Box 42
Folder 18

2 of 8

10273751 3

DECUS
PROGRAM LIBRARY

DECUS NO.	10-198a
TITLE	IMP - PDP-10 IMP72 COMPILER, VERSION 1.5
AUTHOR	Walter Bilofsky
COMPANY	Submitted by: Sonya Shapiro Bolt, Beranek and Newman, Inc. Cambridge, Massachusetts
DATE	October 3, 1973
SOURCE LANGUAGE	IMP72

IMP - PDP-10 IMP72 COMPILER, VERSION 1.5

DECUS Program Library Write-up

DECUS NO. 10-198a

The IMP72 compiler was designed and implemented at the Yale University Department of Computer Science by Walt Bilofsky(*), Institute for Defense Analyses, Princeton, N.J., based largely on previous work of E. T. Irons. Substantial contributions were made by Steven Weingart and Terry Lyons. This manual reflects version 1.5 of the IMP72 compiler, which was written to run under version 5.Ø6 of the DECsystem-1Ø operating system, and which has been successfully run under version 1.31.23 of TENEX.

(*) Present address: Bolt Beranek and Newman Inc., 5Ø Moulton Street, Cambridge, Mass. Ø2138.

CONTENTS

imp (vt): ... (archaic) to eke out, strengthen.
imp (n): ... (archaic) an evil creature.
[Webster's 1967]

"Things used as language are inexhaustibly attractive."
[Emerson 1849]

" ... The red plague rid you,
For learning me your language!"
[Shakespeare 1612]

1: Introduction.

1.1: Description.

IMP is a simple higher-level language intended primarily for system programming. It has been implemented on the PDP-1Ø and CDC 16Ø4 and 66ØØ computers [Irons 197Ø], [Bilofsky 1972]. It is meant to provide language facilities roughly at the level of FORTRAN II yet allow the programmer the flexibility of machine language programming including use of all the machine's registers and instructions and arbitrary control of the program and data areas while the program is running. IMP72 is a version of IMP for the PDP-1Ø which provides the user with the following facilities:

1. Extensibility. The user may specify extensions to the syntax and semantics of the language in forms ranging from simple "macros" to productions which generate calls to compiler code-generating routines. More efficient object code may easily be specified for special cases.

2. Floating point capabilities. A REAL data type and floating point arithmetic are provided.

3. Byte manipulation capability.

4. No reserved words in the syntax.

5. Syntactic error correction and admissibility of ambiguous syntax.

1.2: Status.

As of June 1973, the compiler has been in use at the Yale Department of Computer Science for a full academic year. It compiles itself. A version of the compiler which generates object code for the PDP-11 computer exists at Yale. The PDP-10 compiler is relatively large (37K minimum, with up to about 50K needed to compile itself) and not particularly fast (compiles at the rate of about 60 input tokens per second), but it is felt that this is not unreasonable for a compiler of the generality of IMP.

The IMP72 compiler is provided (courtesy of the Yale Department of Computer Science) and maintained (as of this writing) by the author, on a purely informal basis. Although the author feels that the IMP72 compiler is practically bug-free, and intends to maintain and update it, the user is reminded that this does not constitute a guarantee that it is or he will. Further details on where to send complaints may be found in Section 5.

1.3: How to Use IMP in One Easy Lesson

This section is intended to give the experienced programmer the flavor of IMP so that he may write simple programs without having to read the entire manual.

IMP is similar to FORTRAN II in power, and a bit like ALGOL in flavor. It has no block structure or reserved words, and all variables are global to all programs in a compilation. Statements may be grouped, using parentheses. Variables need not be declared unless they are arrays. Arrays are declared by, e.g., FOO IS 5Ø LONG and subscripts written, e.g., FOO[I+3]. The assignment operator is "←" and the statement separator is ";". All binary arithmetic operators are of equal precedence, and operations are performed from right to left (as in APL). "←" is of the same precedence as the arithmetic operators and also is performed from right to left. Relational operators are evaluated after arithmetic operators.

Statements may be labeled by prefixing them with LABEL:. The transfer of control is GO TO LABEL and the conditional is A=>B or A=>B ELSE C, where B and C are statements or groups of statements (in parentheses), and A is an expression (Ø is false, non-zero is true), perhaps containing a relational operator (EQ, NE, GE, GT, LE, LT). The basic iteration construct is A FOR B IN C,D,E, where A is a group of statements (in parentheses), B is a variable, and C, D and E are expressions. A is executed for values of B from C, incremented by D, to E.

An IMP source file consists of a number of statements, in free format. The file is terminated by "%%". The last statement executed should be FINI(Ø), which exits from the program. To compile the IMP program on file FOO.IMP, call the compiler by the monitor command RUN IMP (on TENEX, just IMP), and when it types an asterisk, type FOO.IMP<CR>. To execute the resulting object program on DECsystem-1Ø, type the command EX FOO.REL,/LIBRARY,IMPLIB.LIB. On TENEX, execute by calling the loader subsystem with the command LOADER, and then type FOO.REL,/LIMPLIB.LIB<ALTMODE>.

1.4: Differences between IMP72 and the Previous PDP-1Ø IMP.

IMP72 contains all features of old PDP-1Ø IMP with the following exceptions:

1. Trace is not implemented.

2. Relational operators and A=>B ELSE C are implicitly parenthesized differently in a few cases. See sections 2.2 and 2.2.4.

3. The formal syntax of the language has other minor changes which should not affect the compilation of old IMP programs in IMP72. For example, A←B is now permissible as a parameter of a subroutine call. (The parameter passed is B.)

1.5: IMP72 on TENEX.

IMP72 was written to run under the DECsystem-1Ø operating system. Since TENEX supports most features of DECsystem-1Ø, it will be possible for TENEX users to make use of IMP. Certain TENEX features, such as referring to directories by name instead of project,programmer numbers, will not be available.

2: Programmer's Guide to IMP72

2.1: Conventions.

2.1.1: Lexical.

The language is basically free-form, subject to the following constraints. Words representing variable names or special words of the language are delimited by non alpha-numeric characters including space. Therefore, spaces may not appear inside words or numbers, and at least one delimiter must appear between two such words. All characters having an ASCII character code of 40B or less are interpreted as blanks, except when enclosed in !, # or ' signs. Thus, string constants may contain any character, including tabs and returns, and returns and line feeds outside !, # and ' signs serve as delimiters. As a general rule, make the text readable and you will find the words properly distinguished.

Identifiers usually consist of alphanumeric strings, but an identifier containing any special character (except !) may be represented by enclosing the entire name in ! signs; e.g., !Funny*id.! (see Section 2.2.1).

All characters appearing between # signs are treated as comments and ignored by the compiler (except for # signs in string constants and between ! signs).

2.1.2: Expressions and Statements.

In IMP, no distinction is made between expressions and statements. Almost everything is an expression, and has a value, although a few expressions like GO TO X do not have a very useful value. Two consequences of this philosophy are that parentheses may be used for grouping what are usually called statements in other languages, and that the statement separator ';' becomes a binary operator (whose value is the second operand).

The terms "statement" and "expression" will be used almost interchangeably herein. When the term "expression" is used, it should be understood that we may be concerned with the value of the expression, whereas when the term "statement" appears, we are interested primarily in the effect of executing it, and only secondarily, if at all, in its value.

2.1.3: Machine Language Level Programming.

The identifiers ØR,1R,...,15R refer to the 16 registers of the PDP-1Ø: Ø,1,...,15. Therefore you may write down expressions which refer to particular registers. If you write down an expression which can be evaluated with a single instruction like '1R←1R+M', that instruction will almost always be used to implement the expression. Instructions which cannot be expressed in this way may be implemented by user-provided syntax.

Statement labels are used exactly like variable names, so that one can reference and change instructions in the program. Thus 'T←2ØØØØØØØØØØØB; T:GO TO T' puts the octal instruction '2ØØØØØØØØØØ' into the next word and executes it. (We happen to know that GO TO T takes just one instruction, and so use it to set aside a word to store in).

2.1.4: Program Structure and Scope of Variables.

Program structure is more or less like that of the PDP-1Ø FORTRAN system. Thus a main program, if any, comes first, then subroutines, if any, follow.

In contrast to the FORTRAN convention, all variables in IMP are global to the entire set of subroutines compiled at the same time, except that formal arguments of a subroutine are local within that subroutine. That is, any two references to the same variable name within a single IMP compilation will refer to the same memory location, even if the references are within different subroutines. However, references to formal arguments of subroutines will have the effect of references to the memory location containing the actual argument in the call of the subroutine. Thus references to formal subroutine arguments will not affect variables of the same name in other subprograms in the same compilation.

The declaration LOCAL is provided to defeat the fact that variables are normally global to the entire module. An identifier which is declared to be LOCAL will be replaced throughout the subroutine in which it is so declared by a unique name, which will not be used outside the subroutine. The identifier will therefore appear local to the subroutine.

IMP uses the FORTRAN subroutine calling conventions (see sect. 2.2.5). Thus, IMP and FORTRAN programs may call each other at will. However, the IMP and FORTRAN I/O

library subroutines have been known to conflict at times. It is therefore advisable not to perform formatted I/O from both IMP and FORTRAN subroutines in the same program.

2.1.5: Compiler Version Number Conventions.

The compiler version number and creation date appear on its prompt and on the listings it produces. Version 1.5 is release 5 of major version 1. A new release may differ from the previous one by having language additions and/or fewer bugs. A major version contains substantial changes. An extension letter (e.g., 1.5(A)) appears in user-created extensions of the compiler (see Section 3.1 for the method of creating an extended compiler version using the C, U and V switches). The date is the creation date of the current copy of the version being run.

2.2: The Expressions.

The expressions of the language are listed here. Expressions may be grouped explicitly by parentheses. Where parentheses are omitted, the following is the order in which subexpressions are evaluated:

1. Unary operators and functions.
2. Binary operators except ";" (but including "←").
3. Relational operators.
4. Conditionals.
5. ";".

When two operators of equal precedence appear consecutively in an expression, the rightmost is performed first (as in APL). E.g., A*B+C is evaluated as A*(B+C). EXCEPTION: the order of evaluation of relational operators on the same level - e.g., A<B GE C - is not defined.

In "A; B", A is evaluated before B. In a conditional, the implicand will be evaluated after the condition is evaluated, if at all. Other than the restrictions listed above, IMP places no restrictions on the order of evaluation of arguments of operators.

In the following table, if no value is specified for an expression, then the value of that expression is not defined and/or not useful.

2.2.1: Variables and Constants

SAMPLE EXPRESSION	MEANING
19576	Integer numbers (base 10).
777000B	Digits followed by 'B': octal constants
2AFB16	A string of digits and letters beginning with a digit and ending in 'Bnn', where nn is a

decimal number. These are constants in base nn (called "flexadecimal" constants). The letters and digits to the left of the last B are interpreted as a constant in base nn. E.g., 2ABB16 is the base 16 constant 2AB (683 decimal). 1Ø1ØØB2 is the binary constant 1Ø1ØØ, or 2Ø decimal. The base may be arbitrarily large, but only digits and the letters A-Z may be used to represent digits. (A-Z are the digits 1Ø-35.) Note that a constant starting with a letter is not legal. for example, ABB12 is a variable name; the desired constant may be written ØABB12.

3.27"-5

Floating point constant. Consists of a string of digits containing an embedded decimal point (i.e., .Ø1 and 3. are not legal), and followed optionally by " and a signed power-of-ten scaling factor.

A

Variable name. Any string of letters and digits not interpretable as one of the above forms of constants.

9R

Numbers Ø through 15 followed by 'R' name the corresponding PDP-1Ø registers.

'ABCdef'

ASCII strings: stored left justified, zero filled and spilling into successive words. An ASCII string always terminates with at least one zero character. The character ' is represented within a string by two successive quotes: ''. Thus, the string consisting of one single quote would be written '''' (but would print as '). The string '' is the same as Ø.

R'Stg'

A 1 to 5 character ASCII string prefixed by the letter R is a constant containing the ASCII characters stored right-justified. (R may also prefix a string constant written as !'Stg! - see following paragraph.)

!Any thing.!

Special identifier: any string of characters not containing a '!', bracketed by '!'s. The string of characters is considered to be a single identifier or symbol. In particular,

the following interpretations are made:
One special character is recognized as itself. E.g., !+! is the same as +.
Any string starting with a single quote is recognized as the ASCII string containing the remainder of the string. E.g., !'Don't! is the ASCII string constant "Don't".
Any other string is interpreted as an identifier whose name is exactly the string appearing between the ! signs. This allows variable names containing special characters.

The user employing variable names containing special characters is warned that the compiler generates temporary labels prefixed with a %, and syntactic class names in syntax statements prefixed with a #. Use by the user of identical names may produce anomalous results.

A[E]

The E'th word of array 'A' starting from zero. A is any variable name or constant, and E is any expression.

[E]

The E'th word of memory: thus the contents of the word whose address is E.

LOC(A)

The address of the memory location containing A. A may be a simple or subscripted variable.

2.2.2: Unary Operators and Functions.

-A

Negative of A (twos complement)

NOT A

Ones complement of A

(A)

Parentheses used to group expressions: value is A. A may be any expression, including a string of statements.

2.2.3: Binary Operators.

A←B

Stores the value of B in A. The value of this expression is the value stored in A. If A and B are of different arithmetic types, the value of B is converted to the type of A before storing.

A<=B

Stores the value of B in A, without performing any type conversion if A and B are of different arithmetic types.

A; B

; is a binary operator whose value is B. It is used as a statement separator. It always evaluates A first, then B. Example: the value of (A←5; B←3) is 3.

A+B
A-B
A*B
A/B

These are the arithmetic operators. They are either integer (if both operands are of type integer) or rounded floating point (if either operand is real. If one operand is integer, it is converted to real before the operation is performed.). (Caution: when / is used with register names as operands, in general the divide is performed in a different register than that holding either operand, in order to avoid difficulties due to the divide operation requiring two adjacent registers. However, the expression R←R/S, where R is a register name, is compiled to perform the divide in register R.)

A//B

The remainder of A when divided by B. If either A or B or both are floating point, they are converted to fixed point before the computation is performed. The result is always integer.

A<B (or A LT B)
A>B (or A GT B)
A=B (or A EQ B)
A LE B
A GE B
A NE B

Relational operators. These have the value -1 if the relation holds, otherwise 0. The words LE, GE and NE are used for the operators 'less than or equal to', 'greater than or equal to' and 'not equal to'. NOTA BENE: Relational expressions must be enclosed in parentheses wherever they appear except as the condition in a conditional expression.

A OR B
A AND B
A XOR B
A EQV B

Bit-by-bit logical binary operators.

A LS B

A left shifted B bits, end off, zero filled.

A RS B

A right shifted B bits, end off, zero filled.

A LROT B

A left shifted circularly B bits.

A RROT B

A right shifted circularly B bits.

A ALS B

A left shifted B bits, sign extended.

A ARS B

A right shifted B bits, sign extended. (If B is negative in the above instructions, the direction of shift is reversed.)

2.2.4: Control Expressions.

A=>B

If A is not 0, evaluate B (Read as "A implies B"). Thus, K=>X←1 does nothing if K is zero, or sets X to 1 if K is non-zero. X<Y=>(P←1;Q←2) changes P and Q if X is less than Y, otherwise does nothing. NOTA BENE: If A is a relational expression, it need not, and for best object code should not, be enclosed in parentheses. If A is a constant, or an expression involving only constants, then the code generated by the compiler for the expression A=>B will consist either of B,

if A is non-zero, or nothing at all, if A is zero. This allows code to be compiled conditionally if, for example, A is an identifier defined by syntax to be a constant. In the case that B is not compiled due to A being a constant 0, any tags contained within B will not be defined, but declarations contained in B will take effect.

A=>B ELSE C

This expression has the value B if A is nonzero, and C if A is zero. It will evaluate only one of its operands each time it is executed; thus it can be used to replace the construct "A=>(B; GO TO FOO); C; FOO:" if the user does not mind the value being computed and ignored (which costs a few extra instructions of object code). NOTA BENE: The expression A←B=>X ELSE Y, e.g., will be interpreted as (A←B)=>X ELSE Y. To assign the obvious interpretation to it, write A←(B=>X ELSE Y). If C contains another =>-ELSE clause, then C should be enclosed in parentheses.

A FOR B IN C,D,E
A FOR B TO E
A FOR B FROM E

These expressions perform repeated execution of the expression A with different values of the variable B. The IN form executes A with values of B equal to C, C+D, C+2*D,..., E; the TO form for B equal to 0,1,...,E, and the FROM form for B equal to E,E-1,...,0. In every case the expression A is evaluated at least once regardless of the values of C, D and E. In the IN case when D is not a constant, B must exactly reach the value of E. E.g., in

```
I←2;
(A[J]←0) FOR J IN 0,I,9
```

the loop will never terminate. In all other cases, the loop terminates when B passes the terminal value. C, D and E may be any expressions, but in the IN and TO cases D and E will be evaluated once per iteration of the loop. The FROM case generates the most efficient code and is therefore to be preferred. If control is tranferred from within the loop, the value of the index variable B is preserved. The value of B

following normal termination of the loop is undefined.

The special cases (A[I]←B[I]) FOR I TO/FROM V, where A and B are integer arrays and V is a single variable or constant, are compiled as a block transfer instruction. The user is cautioned that in the FROM case, if A and B overlap, the code produced will be incorrect. In more general cases, block transfers may be specified by the MOVE construct (Section 2.2.9).

WHILE A DO B

Repeatedly evaluates the expression A, and, if it it is nonzero (true), evaluates the expression B. If A is zero (false), control passes to the next statement. If A is initially zero, B is never executed.

A UNTIL B

This expression evaluates the expression A, then evaluates B and if B is false (zero) it repeats. Even if B is initially true, A is always executed at least once.

GO TO E

Transfer control to the memory location E. E will usually be a tag, but may be a subscripted variable. for example, GO TO TAG is equivalent to GO TO [LOC(TAG)].

GO TO (L0,L1,...,LN) E

This expression transfers control to the label Li, where i is the value of expression E, and L0,...,LN are identifiers. If E<0 or E>N, the effect is undefined.

T:A

The expression A is tagged with the label T. Two uses of this are: GO TO T transfers control to this point in the program, and references to the variable T will refer to the first instruction word in the expression A. To insert a tag where an expression does not immediately follow, such as before a ')', write TAG:0.

2.2.5: Programs, Subprograms and Subprogram Calls.

A%%

An IMP program consists of an IMP expression followed by two % signs. The % signs terminate the input file as far as the compiler is concerned.

SUBR A(B,C) IS D

Define a subroutine (or function) named A, with formal arguments B and C. The value of the subroutine is D, (unless the subroutine is exited via a RETURN statement, q.v.). The normal exit from the subroutine is by "running off the end" of the expression D. Example:

```
SUBR ABS(A) IS (A<0=>-A ELSE A);
```

is a subroutine whose value is the absolute value of its argument. With this definition P←ABS(Q) calls the subroutine ABS and sets P to the value of Q. The arguments of a subroutine "share" the memory location of the actual arguments with which the subroutine is called; that is, changing the value of one of the parameters in the subroutine will change the value of the variable in the calling program which corresponds to that argument. The linkage generated by the subprogram call F(A1,A2,...,An) is

```
JSA 16,F
JUMP A1
JUMP A2
  ...
JUMP An
(return)
```

The subprogram refers to its i-th argument by @i-1(16), and returns via a JRA @n(16). Consequently, if a subprogram is called with more arguments than it expects, no harm is done, and if it is called with fewer, it will return to a random place following the call (unless it blows up trying to reference a non-existant argument first). Functions return their value in register 0R. This calling sequence is compatible with FORTRAN at the current time (June 1973). In the case of compiling pure (re-entrant) code, the same calling sequence is used, with a two-word

entry block being relegated to the low segment.

SUBR A() IS D

Define a subroutine (or function) named A with no formal arguments, and with value D.

RETURN E

Return from the subroutine in which this statement appears. The value returned by the subroutine is E.

A(B,C,D)

Execute the function A with parameters B,C,D: The value of this expression is the value of the function. (Since there is no distinction in IMP between functions and subroutines, this expression is also used to call A as a subroutine with the indicated parameters.)

A()

Execute the function A with no parameters.

2.2.6: Byte Access.

A<B,C>

This object is called a "byte". A is a variable, either simple or subscripted; B and C are any expressions. A<B,C> designates the portion of the word A consisting of B bits, located C bits from the right hand side of the word. Thus, e.g., A<12,12> designates the center 12 bits of A, A<1,35> is the sign bit. A byte may be used as an operand, or have a value stored in it. Example: the program fragment

```
A←3;
(A<9,I+9>←3+A<9,I>) FOR I IN
 0,9,18;
```

has the effect of setting A to 0140110060003B.

The constructs A<R> and A<L> designate the right and left half words of A. They are equivalent to A<18,0> and A<18,18> respectively.

BYTEP A<I,J>

This expression has the value of a PDP-10 byte pointer to the designated byte. Its usefulness lies in the ability to write X←BYTEP A<B,C> and do byte access by referencing X, as explained in the next paragraph. If I and J are not constants, the byte pointer will be computed with their values at the time the statement is executed and will not be reevaluated if I and J change value later.

<X>

X must contain a byte pointer. (This may be accomplished by X←BYTEP A<I,J>.) Then <X> refers to the byte designated by the byte pointer.

<+X>

X must contain a byte pointer. When <+X> is evaluated, it increments the byte pointer, and its value is the byte then pointed to. Incrementing a byte pointer to A<I,J> means that the pointer will now point to A<I,J-I> if J GE I, or to A[1]<I,36-I> if J<I. Example: Suppose the memory locations starting at STG contain an ASCII string stored five 7-bit characters to a word, with the rightmost bit empty. Then the following program fragment unpacks STG into the locations starting at UNP, one character per word, rightjustified (remember, the ASCII string terminates with a 0 character):

```
X←BYTEP STG<7,36>; I←-1;
LOOP: (UNP[I←I+1]←<+X>)=>GO TO LOOP
```

Byte pointers may also be stored into, with or without first being incremented: <+X>←I.

2.2.7: Input/Output.

PRINT A,B,C
READ A,B,C

A, B, C, etc., are a list of expressions to be transferred from/to input/output files, with format specifiers mixed in. Any expression may appear in a PRINT list; if an expression other than a simple or subscripted

entry block being relegated to the low segment.

SUBR A() IS D

Define a subroutine (or function) named A with no formal arguments, and with value D.

RETURN E

Return from the subroutine in which this statement appears. The value returned by the subroutine is E.

A(B,C,D)

Execute the function A with parameters B,C,D: The value of this expression is the value of the function. (Since there is no distinction in IMP between functions and subroutines, this expression is also used to call A as a subroutine with the indicated parameters.)

A()

Execute the function A with no parameters.

2.2.6: Byte Access.

A<B,C>

This object is called a "byte". A is a variable, either simple or subscripted; B and C are any expressions. A<B,C> designates the portion of the word A consisting of B bits, located C bits from the right hand side of the word. Thus, e.g., A<12,12> designates the center 12 bits of A, A<1,35> is the sign bit. A byte may be used as an operand, or have a value stored in it. Example: the program fragment

```
A←3;
(A<9,I+9>←3+A<9,I>) FOR I IN
 0,9,18;
```

has the effect of setting A to 0140110060003B.

The constructs A<R> and A<L> designate the right and left half words of A. They are equivalent to A<18,0> and A<18,18> respectively.

BYTEP A<I,J>

This expression has the value of a PDP-10 byte pointer to the designated byte. Its usefulness lies in the ability to write X←BYTEP A<B,C> and do byte access by referencing X, as explained in the next paragraph. If I and J are not constants, the byte pointer will be computed with their values at the time the statement is executed and will not be reevaluated if I and J change value later.

<X>

X must contain a byte pointer. (This may be accomplished by X←BYTEP A<I,J>.) Then <X> refers to the byte designated by the byte pointer.

<+X>

X must contain a byte pointer. When <+X> is evaluated, it increments the byte pointer, and its value is the byte then pointed to. Incrementing a byte pointer to A<I,J> means that the pointer will now point to A<I,J-I> if J GE I, or to A[1]<I,36-I> if J<I. Example: Suppose the memory locations starting at STG contain an ASCII string stored five 7-bit characters to a word, with the rightmost bit empty. Then the following program fragment unpacks STG into the locations starting at UNP, one character per word, rightjustified (remember, the ASCII string terminates with a 0 character):

```
X←BYTEP STG<7,36>; I←-1;
LOOP: (UNP[I←I+1]←<+X>)=>GO TO LOOP
```

Byte pointers may also be stored into, with or without first being incremented: <+X>←I.

2.2.7: Input/Output.

PRINT A,B,C
READ A,B,C

A, B, C, etc., are a list of expressions to be transferred from/to input/output files, with format specifiers mixed in. Any expression may appear in a PRINT list; if an expression other than a simple or subscripted

entry block being relegated to the low segment.

SUBR A() IS D

Define a subroutine (or function) named A with no formal arguments, and with value D.

RETURN E

Return from the subroutine in which this statement appears. The value returned by the subroutine is E.

A(B,C,D)

Execute the function A with parameters B,C,D: The value of this expression is the value of the function. (Since there is no distinction in IMP between functions and subroutines, this expression is also used to call A as a subroutine with the indicated parameters.)

A()

Execute the function A with no parameters.

2.2.6: Byte Access.

A<B,C>

This object is called a "byte". A is a variable, either simple or subscripted; B and C are any expressions. A<B,C> designates the portion of the word A consisting of B bits, located C bits from the right hand side of the word. Thus, e.g., A<12,12> designates the center 12 bits of A, A<1,35> is the sign bit. A byte may be used as an operand, or have a value stored in it. Example: the program fragment

```
A←3;
(A<9,I+9>←3+A<9,I>) FOR I IN
 0,9,18;
```

has the effect of setting A to 0140110060003B.

The constructs A<R> and A<L> designate the right and left half words of A. They are equivalent to A<18,0> and A<18,18> respectively.

BYTEP A<I,J>

This expression has the value of a PDP-10 byte pointer to the designated byte. Its usefulness lies in the ability to write X←BYTEP A<B,C> and do byte access by referencing X, as explained in the next paragraph. If I and J are not constants, the byte pointer will be computed with their values at the time the statement is executed and will not be reevaluated if I and J change value later.

<X>

X must contain a byte pointer. (This may be accomplished by X←BYTEP A<I,J>.) Then <X> refers to the byte designated by the byte pointer.

<+X>

X must contain a byte pointer. When <+X> is evaluated, it increments the byte pointer, and its value is the byte then pointed to. Incrementing a byte pointer to A<I,J> means that the pointer will now point to A<I,J-I> if J GE I, or to A[1]<I,36-I> if J<I. Example: Suppose the memory locations starting at STG contain an ASCII string stored five 7-bit characters to a word, with the rightmost bit empty. Then the following program fragment unpacks STG into the locations starting at UNP, one character per word, rightjustified (remember, the ASCII string terminates with a 0 character):

```
X←BYTEP STG<7,36>; I←-1;
LOOP: (UNP[I←I+1]←<+X>)=>GO TO LOOP
```

Byte pointers may also be stored into, with or without first being incremented: <+X>←I.

2.2.7: Input/Output.

PRINT A,B,C
READ A,B,C

A, B, C, etc., are a list of expressions to be transferred from/to input/output files, with format specifiers mixed in. Any expression may appear in a PRINT list; if an expression other than a simple or subscripted

variable appears in a READ list the effect will be to read into a temporary location. The format specifiers are listed below.

The value of a PRINT statement is the number of the last column on a line that was printed into. In particular, the value of PRINT / is 0. The value of a READ statement is -1 if an attempt has been made to read past the end of the file, 1 if an attempt has been made to read past the end of a line, and 0 otherwise.

If no file is explicitly specified, output is to file P0 and input is from file PI. There is no rule against using many statements to print or read one line; the only thing that terminates a line is a /. A format, once specified, remains in effect until another specification is encountered, even through several different PRINT or READ statements. Changing the PRINT format or file does not affect the READ format or file, and vice versa.

In printing, if a data item is presented that is too large for the specified field width, a field just large enough to contain it is used. Consequently, if a field width of 0 is specified, a field just large enough to contain each data item will be used.

In reading, if the field width specification is a single-character string (e.g., ','), the input field will terminate on the first of a) the specified character; b) the end of the line; c) 128 input characters. A field width specification of 0 is equivalent to ','. Blanks are ignored on input (except in STG conversion and when ' ' is the field terminator). Tabs are converted to a single blank on input. All fields are limited to 128 characters on input.

Field widths in format specifiers are usually constants but may be any expression.

It is necessary to close out your output files explicitly in your program some time before it terminates. This is done by the subroutine call FINI(i), for i=0 or -1. This closes all your files. If i=0, FINI exits to the monitor; if i=-1, FINI returns normally.

You may also call FINI(NA,EXT,PN,PJ) where NA & EXT are a specific file name and extension in ASCII, and PN and PJ are a programmer and project number. This causes FINI to close out the specific file named and return normally. If any of EXT, PN or PJ are zero, the usual default case is assumed.

At most four files may be referred to in IMP I/O. If it is desired to refer to more, one of the previous files must be explicitly closed using FINI (or close all of them with FINI(-1)).

Format Specifiers:

IGR N

Conversion is in base 10, in a field N positions wide. On output, leading zeros are suppressed, and a leading minus sign is printed for negative numbers.

OCT N

Conversion is in base 8, unsigned (i.e., negative numbers print with leading 7's), in a field N positions wide. On output, leading zeros are printed. If N is greater than 12, exactly the leftmost N-12 positions will be blank.

STG N

Conversion is in ASCII, in a field N wide. On output, if N is 0 characters are printed until a 0 byte is encountered. Otherwise, characters are printed until a 0 byte is encountered or until N characters have been printed.

FLT M.N

Prints a floating point number in a field M columns wide, with N digits to the right of the decimal point. if N is 0, no decimal point is printed. If the number will not fit in the field as specified, scientific notation is used. If M is negative, scientific notation is always used in a field -M columns wide. (IMP uses " to mean 'times 10 to the'.) Exception: if M is 0, a field just wide enough to contain the number is used. On input, FLT M.N is equivalent to IGR M.

/

New line. On output, causes termination of the current line. If a line is printed but not terminated with a /, then the line will not print. Successive /'s produce blank

lines. On input, a / causes the remainder of the current line to be skipped over.

FILE A
FILE A.B
FILE A[P,R]
FILE A.B[P,R]

Specifies the file for input/output. A and B are file name and extension, and may be either ASCII string constants or unsubscripted variables. P and R are project and programmer numbers. If they are constants, they are interpreted correctly if and only if they are written in octal but without the letter "B" suffixed. If they are expressions, it is the programmer's responsibility to take into account the fact that project and programmer numbers are octal. Caution: FILE FORO3.DAT, e.g., refers to variables FORO3 and DAT. He who wrote this probably meant to write FILE 'FORO3'.'DAT'.

Examples: A←101B; PRINT STG 1,A,IGR 4,A,OCT 6,A,A,/ produces the line
"A 650001010001010"
(PRINT IGR 3,I)FOR I TO 2;PRINT / produces the line
" 0 1 2"
PRINT STG 0, 'This is a Page Heading',/
produces the line
"This is a Page Heading".

DEVICE D

Causes the next FILE specification (on either READ or PRINT) to refer to a file on device D. D may be either an ASCII string constant or a variable containing a string value. Default device is 'DSK:'. I/O to the teletype is a special case. The teletype is designated as the I/O device by DEVICE 'TTY:', and no subsequent file specification. The next file specification will cause the input or output to revert to that file. Caution: device specification will only work for TTY: or for directory devices having physical records of 128 words, such as disk and DECtape.

IMAGE MODE

This format specification causes subsequent data transfer to be in unformatted mode, transferring 36-bit words between the file

and the variables in the i/o statement list. Mixing IMAGE MODE and other format specifications in reading or writing the same file may produce anomalous results.

TAB N

On PRINT only, causes the next character of output to appear in column N (numbering starting with 1). If printing has already gone past column N, no action is taken. that line.

FILL 'c'

On output only, causes leading positions in all fields to be printed as the specified character instead of being left blank. Thus, for example, FILL '0' causes leading zeros to be printed. This specification, unlike the others, affects all output files, not just the current one.

2.2.8: Declarations; DATA and REMOTE statements.

LET I=2R,A=3,BF=B[1]

THE LET statement provides a convenient way of declaring synonyms. It consists of the word LET, followed by a list of synonym declarations of the form N=V, where N is a name and V is a constant, or a simple or subscripted variable. The effect is that all subsequent appearances of N in the program will be interpreted as if V had been written instead(*).

A,B ARE 3 LONG,COMMON
1R IS REGISTER
FOO,3R IS RELEASED

Declarations: the general form for a declaration is a list of names, then a list of characteristics to be associated with these names. The characteristics available are:

(*) LET A=B is in fact just "syntactic sugaring" for the syntax statement <ATOM> ::= A ::="B".

REAL

Specifies that the variables are real numbers, and all arithmetic performed on them is to be floating point. Type conversion between real and integer is always done implicitly.

n LONG

Where n is a constant or constant expression. The variables are declared to be arrays, and n words are reserved for each of them. (The n words may be referred to as A (or A[0]), A[1], ... A[N-1].)

COMMON

Variables are made common. They are assumed to be defined in another program unless they are declared n LONG or appear as a tag in this program, in which case they are defined here. (n may be 1 if desired.) Declaring a variable to be COMMON will make it the same as a FORTRAN common block with that name. Other variables may be located in that block by defining the variables in syntax statements or LET statements.

LOCAL

Makes variables local to the current subroutine from this point to the end of the subroutine, but does not affect the meaning of the identifier outside the current subroutine. This declaration is also useful for making local register assignments.

REGISTER

If A is not among 0R-15R, the declaration A IS REGISTER binds A to a register selected by the compiler, in the range 1R-13R. Until the declaration A IS RELEASED is encountered, all references to A will be replaced by references to the register. If A is a register name 0R-15R, this declaration warns the compiler to avoid using it, because the programmer intends to use it. If the register is already in use, an advisory is issued.

RESERVED

This declaration precludes the use of the associated register by the compiler up to the end of the program or until a RELEASED or AVAILABLE declaration is encountered. RESERVED is distinguished from REGISTER in

that the latter only shields a register from use by the compiler until the last time it is referred to; RESERVED remains in effect for the entire source program.

AVAILABLE

This declaration informs the compiler that the associated register may be used by it for computations. It does not affect the binding between an identifier and a hardware register, and the register will be reserved again beginning at the next reference to it.

RELEASED

For identifiers which have been bound to a hardware register, that binding is terminated, and subsequent references to that identifier will refer to a memory location. The register becomes available to the compiler. For hardware register names, RELEASED is equivalent to AVAILABLE.

SCRATCH

Ordinarily, all registers which are reserved at the point of a subprogram call are saved before the call and restored afterwards. This declaration signals the compiler not to save a register.

PROTECTED

This declaration signals the compiler to preserve through subprogram calls the value of a register which had previously been declared SCRATCH. PROTECTED is the default mode.

REMOTE S

S is any statement. Has the effect of causing the code for statement S to be inserted not at the point in the program where the REMOTE statement appears, but at the end of the program, just before the constants and variables. Useful in quoted semantics where initialized local variables are required, e.g., LOCAL FOO IN "... REMOTE FOO:DATA(20); ...".

DATA (L)

L is a list of variable names and constant expressions. The DATA statement produces

data words at that point in the program containing the values of the constant expressions, and the addresses of the variables, in L. One word is used for each item in L, except that ASCII strings are stored in as many words as are required. DATA statements may be used to preset variables to a value at compile time, viz.: VAR: DATA (3). Avoid putting DATA statements where they might get executed (unless you really want to execute your data).

2.2.9: Miscellaneous Constructs.

CALL ME Ishmael

Ordinarily, the name of the source file is the name which activates the DDT symbol table for a compiled IMP72 program. This statement overrides that name, and assigns the name Ishmael for that purpose.

EXECUTE E

Executes the instruction contained in the variable E.

CALLI(C,V)

Executes the DECsystem-10 UUO CALLI C, with the AC value V. C must be a constant less than 4096, or else this construct will compile as a call to the subroutine CALLI (which fortunately is in the library package FORTIO.REL - see Section 5). The value of this construct is the AC value returned by the CALLI. In addition, if the error return was taken by the CALLI, the variable CALLI will have been set to -1 (but note that if the error return was not taken, its previous value will not have been changed).

XWD A,B

Useful occasionally in system calls. Has the value B<R> OR A<R> LS 18.

IOWD A,B

Has the value XWD -A,B-1.

TWOSEG

This statement will cause reentrant (pure) code to be produced by the program at the

head of which it appears. It is equivalent to the compiler switch /R. It should be the very first statement in the source file.

FIX(A)

Has the value of the expression A, converted to type integer if it was not already of that type.

FLT(A)

Has the value of the expression A, converted to type real if it was not already of that type.

MOVE A THROUGH N TO B

Generates a block transfer instruction, which efficiently performs the operation (B[I]←A[I]) FOR I TO N. N is any positive expression (if real, it is fixed). A and B are any expressions having an address, such as simple or subscripted variables, or [expression].

2.3: Syntactic and Semantic Extension.

This section explains the IMP72 facilities for extending the language. It is possible to add productions to the syntax for the language, defining the semantics for the new constructs in several ways. Semantics may be specified in terms of expressions written in the portion of the language already defined, as explained in Section 2.3.1. Alternatively, semantics may be performed by a series of function calls to semantic subroutines contained within the compiler, as documented in Section 2.3.3.1. The user may use the subroutines provided, or, if he requires operations not available from the current set of semantic subroutines, he may as a last resort go into the compiler to add new ones. Different semantics may be specified in special cases, as explained in Section 2.3.3.2, either to perform different operations upon objects in different contexts, or to generate better object code. When the semantics for several productions are very similar, they may be lumped together under a general case, as noted in Section 2.3.3.3 and 2.3.3.4.

2.3.1: The Easy Way - Syntactic "Macros"

This section is an introduction to syntactic extension in IMP72 for the casual user, and is intended to provide him with sufficient information to utilize the basic facility without informing to the point of total confusion. Some of these features are in fact less restricted than indicated in this section. The full truth emerges in subsequent sections.

IMP72 contains a facility for defining syntactic macros, or patterns which the compiler will recognize in a program and generate specific code for. An example of a definition for the absolute value function using a syntactic macro is

```
<EXP> ::= ABS ( <A> ) ::= "A<0=>-A ELSE A"
```

This statement, when inserted in a program, will cause the compiler to recognize the construct ABS, followed by a "(", followed by any expression, followed by a ")", and to substitute for it the code enclosed in " signs, inserting the actual expression for each instance of A in the quoted expression. Thus, writing

```
X←ABS(R+3)
```

later in the program would be the same as writing

```
X←(R+3<0=>-(R+3) ELSE R+3)
```

A syntax statement fits the following pattern:

```
<EXP> ::= syntax part ::= semantic part
```

The syntax part is the pattern which the compiler is to recognize. It consists of the names and special symbols in the pattern, and, for each expression in the pattern, an identifier in angle brackets: e.g., <FOO>. Single characters in quotes (e.g., '#') are interpreted as that character, without the quotes, and the characters : ; " < and % must be quoted if they appear in the syntax part.

The semantic part consists of an IMP expression in double quotes, perhaps preceded by a list of local variables. It defines the code IMP is to generate when it recognizes an instance of the syntax part, with the actual expressions in the instance to be inserted in the quoted expression in place of the identifiers which appeared in angle brackets in the syntax part.

The quoted expression may be preceded with a list of local variables. For example, notice that the definition of ABS above computes A twice, which may be inefficient if A is an expression. A more efficient way of doing it is

```
<EXP> ::= ABS ( <A> ) ::= LOCAL R IN "R IS REGISTER;
                          (R←A)<0=>-R ELSE R"
```

This definition computes A only once, storing it temporarily in the register R.

Another example is taken from the syntax built into the compiler:

```
<EXP> ::= <A> FOR <B> FROM <C> ::= LOCAL FOR IN "B←C;
                                   FOR: A;
                                   (B←B-1) GE 0=>GO TO FOR";
```

The list of local variables may consist of up to ten names, separated by commas.

This concludes the introduction to syntactic macros.

2.3.2: Specifying Syntax.

The form of the syntax specification statement is a modification of the form of a BNF production (Naur 1963), with alternative right-hand sides not allowed, and a semantics definition added at the right. The general form is:

<class> ::= syntax-part ::= semantic-part

We will use the term "arguments" of a production or syntax statement to refer to the non-terminals appearing in the syntax-part of a syntax statement (i.e., on the right-hand side of the production).

The compiler interprets a program by recognizing instances of certain syntactic classes, such as <EXP> (expression), <VBL> (variable), etc. The function of the syntax part of a syntax statement is to tell the compiler about a new construct that it must recognize as an instance of a certain class, specified by the identifier in <class>. The classes of interest in the IMP language are:

<NAME>: Any variable name or constant. Should generally be avoided in favor of <VBL> unless the user is sure of what he is doing.

<VBL>: Any constant or simple or subscripted variable.

<ION>: Any <VBL>, function call, or an <STL> enclosed in parentheses.

<ATOM>: An <ION> or byte of an <ION>.

<BYTE>: Any byte or byte pointer reference.

<EXP>: Class of most expressions.

<ST>: Includes <EXP>'s, conditional expressions, declarations, etc.

<STL>: List of one or more <ST>s, separated by ;'s.

The full set of syntactic classes in IMP may be determined by reading the syntax of the language (see Appendix II). The user may define his own syntactic classes as the humor falls upon him.

The syntactic part of a syntax statement may contain:

1. Identifiers and special characters, representing themselves. The characters %, ", :, < and ; must

be enclosed in single quotes.

2. Single characters enclosed in single quotes. They are interpreted as the character alone.

3. Syntactic classes, represented by <CLASS,name>, where CLASS is the name of the class and name is an identifier by which to refer to this argument of the production in the semantics.

4. <name>. This is interpreted as <EXP,name> unless it is the very first thing after the ::=, in which case it is interpreted as <ATOM,name>. This has the effect of making operations defined using <name> be right-associative, in keeping with the IMP convention.

Example: In section 2.3.1, an expurgated version of the syntax of FROM loops was given. The adult version illustrates the use of syntactic classes:

```
<ST> ::= <EXP,A> FOR <VBL,B> FROM <EXP,C> ::=
                LOCAL FOR IN "B←C;
                              FOR: A;
                              (B←B-1) GE 0=>GO TO FOR"
```

The interpretation of the syntactic part of this statement is that the compiler is henceforth to recognize as an <ST> the construct consisting of any <EXP>, followed by the word FOR, followed by any <VBL>, followed by the word FROM, followed by any <EXP>.

Example: This example illustrates the use of user-defined syntactic classes and recursive definitions. The construct to be defined is GO TO (L0,L1,..,LN) I which transfers control to Li where i is the value of I. (If I<0 or I>N, the effect is undefined.)

```
<ST> ::= GO TO (<GOLIST,A>) <B>  ::=
           LOCAL GO IN "GO TO GO[B];GO: A";
<GOLIST> ::= <NAM,A>                ::= "GO TO A";
<GOLIST> ::= <GOLIST,A>,<NAM,B>  ::= "A; GO TO B"
```

Then the expression

```
GO TO (A,B,C) J-3
```

produces

```
GO TO %GO1[J-3];
%GO1: GO TO A;
      GO TO B;
      GO TO C
```

(%G01 is a unique name generated by the compiler for the local variable G0.) This example depends on the fact that a GO TO V generates exactly one machine word of code if V is a variable.

It is possible to write a syntax statement with no semantic part:

<class> ::= syntax-part

The semantics implied by this syntax statement is to discard all arguments of the production except the first. If there are no arguments or one argument, no information is lost.

2.3.3: Specifying Semantics.

2.3.3.1: Semantic Routines.

In Section 2.3.1 we saw that the semantic part of a syntax statement could consist of quoted semantics - i.e., an IMP expression enclosed by double quotes. There is one alternate format for semantics: a functional expression consisting of calls to semantic subroutines.

Semantic subroutines are normal compiler subroutines which are also available to the syntax writer in a limited sense. They are called with arguments of the following types:

1. Constants. Usually small positive numbers. These can specify opcodes, switches, or whatever, depending on the routine.

2. Negative directory indices. Identifiers are, by convention, passed to semantic routines as the negative of their index in the directory.

3. Registers. These are in the form of indices in a table of symbolic registers. Assignments to actual machine registers are made during the assembly phase.

4. Objects.

5. Two arguments connected by a + sign. The value is the sum of the arguments. a syntactic ambiguity diagnostic will result from an argument of the form arg+arg+arg; the diagnostic may be ignored.

Objects are pointers into a stack which holds the expressions being processed by the semantics of the current production. An object corresponds to some <class,name> in the production an instance of which is being compiled. Objects either are identifiers from the program being compiled, or have been constructed by semantics operating on and/or combining other objects.

Objects come in five flavors: Name, Register, Constant, Variable and Memory: All but Name will have an arithmetic type (real or integer), and may have associated with them some object code which computes the expression of which the object is the value.

Name - Has no properties other than a name. Is turned into another type of object by semantic routine NAME (q.v.). Names will not pop up as operands; they will have been passed through NAME on their way to becoming <VBL>s.

Register - Has no properties other than a register. A Register object usually designates a register holding the result of a computation.

Constant - May be a constant from the source program (in which case it has a name), or a computed one. In any event, it has a value.

Variable - Designates a simple variable, with maybe a constant subscript. No other properties.

Memory - Designates some address in memory too complicated to be a Variable. May have any or all of a name, a constant subscript, an index register (containing the value of a computed subscript), and even an indirect bit.

2.3.3.1.1: Calling Conventions for Semantic Routines.

The semantic part of a syntax statement may consist of a call to a semantic routine, in standard IMP subroutine call format. The arguments of the call may be either:

1. Constants, unsigned, less than 30 bits. The argument passed to the semantic routine is that number.

2. Identifiers appearing as names of instances of classes in the production (e.g., A in the example of Section 2.3.1). The argument passed to the semantic routine is a "stack pointer" to the object recognized as that part of the production.

3. Identifiers not appearing as names of instances of classes. The argument passed to the semantic routine is the negative of the directory index of the name.

4. Another call to a semantic routine. The argument passed is the value of the semantic routine.

5. Any two of the above connected by a + sign. The argument passed is the indicated sum.

Although a semantic part consisting of calls on semantic routines looks like part of an IMP program, there are certain conventions which the semantics writer may take advantage of here. Semantic routines may have up to 10 arguments. Arguments are evaluated strictly left to right within one function call. If more or fewer arguments are provided than the routine demands, no difficulty is encountered (except that a reference to a missing argument will refer to a random location in user core).

Therefore, although the ";" operator is not provided in the syntax for semantic routine calls, its effects often can be achieved through the use of arguments which will be evaluated but not used. For example, the effect of

```
F(A,B); G(C,D,E)
```

may be obtained by

```
G(C,D,E,F(A,B))
```

if it does not matter that in the latter case C, D and E are evaluated first.

2.3.3.1.1: Calling Conventions for Semantic Routines.

The following is an example of a syntax statement using semantic subroutines to specify semantics. It is suggested that the reader interpret the semantic routine calls using the table below. 214B is the PDP-10 opcode for Move Magnitude.

```
<ATOM>::= ABS ( <ATOM,A> ) ::= DEWOP(214B,AREG1(1,15B),A);
```

2.3.3.1.2: Table of Semantic Routines

In this table, the types of arguments expected are indicated as follows:

S, T, U: Objects (other than Names except where specified).
R, P : Registers.
I, J, N: Constants or (when specified) negative directory indices.

* Indicates a routine which is object machine-dependant (i.e., which must be altered if code for another computer is to be generated.)

NAME AND ARGUMENTS EFFECT AND RESULT RETURNED.

ADDOP(I,S,T)

*Performs a binary operation on S and T. It is defined as HOOK(S,T, DEWOP(I,REGOF(FETCH(T)),S)). This could be written in semantics but using ADDOP saves compiler table space and is clearer besides.

ADDR(S)

*Makes S a Memory-type object whose address is what was previously its value. The use of this is that whereas DEWOP(I,R,S) compiles an instruction with the address of S in the address field, DEWOP(I,R,ADDR(S)) compiles an instruction with the value of S in the address field. Example: <EXP> ::= <ATOM,I> LS <EXP,J> ::= ADDOP(242B,ADDR(J),I) ADDR is smart enough to take any object (except Names) as S. Result of ADDR is S.

AREG(I)

Returns the register index for hardware register I.

AREG1(I,J)

Returns a brand-new register between registers (I AND 37B) and (J AND 37B) inclusive. If I has the 40B bit set, the register is the returned value of a subroutine and may be moved to make room for another such. If J has the 40B bit set, reserve two consecutive registers and

return the first (see also REG2). If I has the 100B bit set, this register is talked about explicitly by the user and he doesn't want any old compiler going around altering its value implicitly.

BYTEP(S,T,U)

*S is a Variable-type object. Makes a byte pointer for S<T,U>, and puts it off at the end of the program with REMOTE (q.v.). Result is S, whose value is now the (variable containing the) byte pointer. Good things to DEWOP on S are byte instructions.

CONOP(S,T,I,U)

Performs an operation designated by I on two Constant-type objects S and T. See file SYNTAX or the source for CONOP for the codes for I. Feel free to add a few more operations if it will generate better code. U is the implicand in cases of CON RELOP CON=>U. Result is S, with code for T hooked in first if there is any.

COPY(S)

Produces a copy of the object S, including a copy of the code associated with it, if any. Useful if S is to be hooked in in two different places in code to be generated. Care must be taken to use the copy of S in the place which is hooked in first, since trying to copy an object that has already been hooked in may imperil the internal tranquility of the compiler.

DATAST(S)

For use in DATA statements. S is any old object; it gets clobbered. Unpacks the current list (see ENLIST), and generates code for the DATA statement whose constants and variables are in the list.

DECL
DECLARE

Special semantic routines to implement declarations.

DEWFUN(S,I,J)

Adds the code to the object S which instructs the assembly pass to perform special function I, with argument J.

DEWOP(I,R,S)

*Performs machine opcode I upon object S and (if nonzero) register R. This is a very smart routine and is happy with any object (except a Name) as S. If possible, it will use an immediate instruction for constant operands. Caution: If DEWOP is used to implement a binary operator (as in DEWOP(OP,REGOF(T),S), the code for T must be hooked on to the code for S after the DEWOP is performed, otherwise it will be lost. See ADDOP. Result of DEWOP is S. If R is nonzero, S is now a Register object, with value in the register designated by R. Normally, if the result of the instruction generated is to destroy, as a side-effect, the value in R, and R is a user-specified register (as in the expression 5R LS 8), a move is inserted to get the value of R into a scratch register first. This move may be suppressed by adding 1000B to the opcode I.

ENLIST(S)

Places the object S at the bottom of the current list. ENLIST is part of a general recursive list mechanism for stacking up lists of things to be fed all at once to some semantic routine. NEWLIST(S) creates an empty list, and, if S is nonzero, puts the object S on it. GETLIST(S) is a non-semantic subroutine which unpacks the current list on a first-in-first-out basis. If the list is not empty, GETLIST(S) puts the top object into the object S (clobbering the old value), and returns a non-zero value. If the list is empty, GETLIST returns 0, and reopens the list which was current at the last call to NEWLIST. See DECLARE (file IMPSEM) for an example of a semantic routine which uses GETLIST. The result of ENLIST is 0.

ENSTACK(I)

Creates an object of type Name, where I is the directory index (either positive or negative) of the name. For example to make the object corresponding to the variable VAR, write NAME(ENSTACK(VAR)).

ERROR(N,V)

Produces the compiler error message which

is the name of the variable V. N is 0, 1 or 2 for a fatal, ordinary, or advisory error respctively. For error messages containing spaces, etc., enclose the message in ! signs to make it into a variable name. Example: ERROR(2,!You Goofed.!).

FETCH(S)

*Forces the object S to be of type Register (i.e., loads it into a register). Result is S.

FETCH2(S)

*Forces the object S to be of type Register, when double registers are being used. Does a fetch into the first register of a newly reserved register pair unless S is programmer-defined register, in which case nothing is done. Result is S.

FIX(S)

*Generates code to convert S from floating point to integer. Value is S.

FLOAT(S)

*Generates code to convert S from integer to floating point. Value is S.

FREEZE(S)

Flags the code associated with S so that the assembly phase will not optimize out any MOVE instructions in it. Useful when generating skip instructions. Result is S.

HOOK(U,S,T)

Hooks the code for T on after the code for S. U gets this code, and the value of T. U may be one of S or T (and usually is). HOOK thus performs the ";" operator, among its other uses. It must be used whenever two or more arguments which might have code attached appear in a production, except in the few cases where another semantic routine invokes HOOK implicitly. The result of HOOK is U.

MAXWELL
MAXEND

Specialized subroutines to handle the job of switching parser output into a

temporary array rather than feeding it to the code generator. Used to implement quoted semantics.

NAME(S)

S is a Name-type object. NAME turns it into a variable, adding an arithmetic type, and making things right if S is a register or subroutine parameter. Result is S.

NEWLIST(S)

See ENLIST.

OJUMPOP(I)

*I is an M-field for a 300-series opcode (conditional) on the PDP-10. Thus I specifies a relational operator. OJUMPOP returns the M-field for the negation of that operator. If I has the 10B bit set, OJUMPOP returns the M-field for the reverse negation (the reverse of >, for example, is <.)

PAR(I)

Used in CASE semantics to refer to the Ith argument (counting from 0) of the CASE argument list (see section 2.3.3.3.).

PRINCAL
PRINPAR

*Specialized routines handling semantics for PRINT and READ statements.

REGOF(S)

Result is the register of the object S, or 0 if it doesn't have one.

REG2S(S)

If S is of type register, and the register was assigned by AREG1 so as to reserve two consecutive registers, then the value of S becomes the contents of the second register. S is otherwise unchanged. If S is not as specified, undefined things happen. The value of REG2S is S.

REMOTE(S)

Causes the code associated with S to be inserted at the end of the compilation instead of at the current point. Result is S, but with no code. (This may lead to incorrect results if an attempt is made to

compute with the value of S.)

REPVAL(S,T)

Part of the semantics for syntax. Generalizes the VALUE class S within the quoted semantics T (see Section 2.3.3.4).

RETURN(S)

*Generates code to return from the current subroutine (JRA) with S as the value of the subroutine. Result is S, not that it matters.

STACKUP(I)

I is either a negative directory index or a constant. This routine is used to create objects and place them on the stack. (If I is a constant, the constant is first entered in the directory, so that a directory index for the item exists.) NAME(ENSTACK(I)) is then performed to make the object. Result is stack index of the item.

SETPRI(I)

Part of the semantics for syntax. Sets the designated priority bits in the semantics being generated.

STACK(I)

Returns the Ith argument of the production. Used to refer to arguments which are VALUE classes in default semantics (i.e., semantics which are not conditional.)

STORE(S,T)

*Stores the value of T in S, hooking the code for S on after that for T. Result is T.

SUBBEG(S)

*Generates code for the beginning of subroutine S, where S is a Name-type object. Result is S, with the code attached.

SUBPRO(I)

Called with I=0 at the beginning, and I=1 at the end, of every subroutine call. Keeps track of the level of calls, and resets the index of the next temporary for subroutine arguments to 0 whenever it gets

up to level 0.

SUBRCALL(S)

*S must be of type Variable. Generates a subroutine call to S (JSA 16,S), plus code to declare S common, and to save any registers in use at the time in temporaries. (Registers are restored by a DEWFUN(T,2,REGOF(T),SUBPRO(1)), where T is the complete subroutine call with arguments. The result of SUBRCALL is S, whose value is now register OR, which is where subroutines return values.

SUBRPAR(S,T)

*S is a subroutine call or some similar thing being built, and T is an argument. A word of code containing a JUMP A, where A is the address of T, is added on to the end of S. If T has no address, A is the address of a temporary containing the value of T. The code for T, and any additional code necessary to get the JUMP A up to specs, is hooked on to the beginning of S. The result is S.

SUBSCRIPT(S,T)

*Generates code for the Variable- or Memory-type object S[T]. Is smart about not computing constant subscripts, etc. S and T may be any type except name. Result is S, with the code for T hooked on before.

SVAL(I)

Creates an object whose VAL is I. (See also VAL).

SWITCH(I)

Turns on the compiler switch which is the Ith letter of the alphabet.

TAG(S)

S had better be an object of type Name, or Variable without a subscript. Adds to S the code which defines the tag S at that point. Result is S.

VAL(S)

Returns the low order 18 bits of the value of the object S. If S in a constant, then this is the low 18 bits of its value. If S is the result of a VALUE semantics, then

this is the value specified there. If S has had a value set by SVAL, then this is that value.

VALU(S,T)

Part of the semantics for syntax. Defines the current semantics to be VALUE S OF T, where S is a Constant object and T is a Name object.

In addition to the above, there are a number of semantic routines which define the semantics for syntax, all on file RSYN. See Section 4.4 for documentation.

2.3.3.2: Conditional Semantics.

We saw above that semantics may consist of either a semantic routine call or quoted semantics. It is possible to invoke one of a number of semantics for a given syntactic production, depending on the particular case of that production being compiled. First an example:

```
<ATOM>::= NOT <ATOM,A>  ::= "NOT BØCON"=>CONOP(B,Ø,6) ELSE
              DEWOP(46ØB,AREG1(1,15B),A);
```

The semantic part of this syntax statement invokes different semantics for the special case of NOT-constant.

Semantics may consist of a number of alternatives, separated by the identifier ELSE. One alternative may consist simply of a semantic routine call or quoted semantics: this is the semantics executed if none of the special cases obtain. The other alternatives are of the form

condition => semantics

where the semantics is a semantic routine call or quoted semantics, and the condition is an IMP expression which is a case of the production, enclosed in double quotes. The arguments of the semantics should be identifiers which designate arguments in the condition. Referring to identifiers which designate arguments in the syntax part will not give a diagnostic but might produce undesired results.

The condition is satisfied if the expression being parsed matches the expression in the condition. Constants in the condition will match any constant with the same value, even if the names are different. (But constant expressions in the condition are not evaluated. Thus, "-18" in a condition would only match in cases where a minus sign was actually used before a constant with value 18. One should trust the compiler to perform constant arithmetic, and write 777777777756B instead, which will match any constant expression which evaluates to -18.) Variable names will match any <EXP>, unless they are tagged with modifiers. If the same name appears twice or more in the condition, the expressions matching all occurrences must be identical.

Modifiers may be added to the first occurrence of a given variable name in a condition by adding the character 'Ø' to the name, followed by the modifiers. Modifiers all consist of three letters,sometimes followed by one or two twodigit octal numbers. For example, AØCON, FOOØREGØ115, IØIGRVARMEMREG. The modifiers are:

Arithmetic type modifiers (Identifiers with neither of these match either type):

IGR: Identifier matches only objects of type integer.
FLT: Identifier matches only objects of type real.

Object type modifiers (Identifiers with none of these match any object type; identifiers with several match any of the indicated types):

REG: Matches objects of type Register.
REGii: Matches objects guaranteed to be in register ii.
REGiijj: Matches objects guaranteed to be in a register between ii and jj. Range for ii and jj is Ø-37 (octal).
CON: Matches objects of type constant.
CONii: Matches objects which are constant, and zero except for the ii rightmost bits.
CONiijj: Matches objects which are constant, and zero except for the ii bits starting jj from the right end of the word. Range for ii and jj is Ø-77 (octal).
CNG: Same as CON (and may have ii or iijj or attached), but matches objects which are constant in the designated field, with all bits outside that field set to 1. CON and CNG should not both be used to modify the same identifier.
VAR: Matches objects of type Variable.
MEM: Matches objects of type Memory.

Code modifier:

WRD: Matches objects with exactly one word of code associated with them. May not always succeed when one might think it should, due to the order in which objects are combined by the code generator, but will always fail when it should.

Numerous examples of conditional semantics may be found on file SYNTAX, which contains the syntax for IMP72.

Care should be taken to specify special cases as cases of the proper production. For example, suppose one wished to refer to the right 18 bits of a variable by the construct /variable/. This might be accomplished by

```
<VBL> ::= / <VBL,A> / ::= DEWOP(55ØB,AREG1(1,13),A)
```

which fetches the right half of A into a register by a Half Right to Right, Zeros instruction. If one wanted also to be able to write

/variable/ ← expression

and store the value of the expression in the right half of variable, one might then write

```
<VBL> ::= / <VBL,A> / ::= DEWOP(55ØB,AREG1(1,13),A)
ELSE "/A/←B" => HOOK(A,B,DEWOP(542B,REGOF(FETCH(B)),A)
```

(542B is a Half Right to Right Memory instruction.) But /A/←B is a special case not of this production but of the producton for "←". The above syntax statement would produce an error diagnostic. The correct definition would be

```
<VBL> ::= / <VBL,A> / ::= DEWOP(55ØB,AREG1(1,13),A);
<EXP>  ::= <VBL,A> ← <EXP,B> ::=
    "/A/←B" => HOOK(A,B,DEWOP(542B,REGOF(FETCH(B)),A)
```

This would produce an advisory diagnostic, since the syntax in the second statement duplicates a production already in the language, but this is permitted in order to allow defining additional special cases of semantics, exactly as is done here.

2.3.3.3: CASEs.

If two or more productions have exactly the same syntax, except for terminal symbols (i.e., they contain the same syntactic classes in the same order, and are instances of the same class), and if their semantics are defined by identical semantic routine calls, with perhaps only a few constants changed (e.g., different opcodes), then it is possible to define a general case, and define the semantics of each production as a special case of it.

Example:

```
<EXP> ::= <ATOM,I> LS <EXP,J> ::= CASE (242B,514B,554B) OF
      SHIFTS  (ADDOP(PAR(Ø),ADDR(J),I)
      ELSE "AØVARMEM LS 18"=>DEWOP(PAR(1),AREG1(1,15B),A)
      ELSE "AØVARMEM LS 777777777756B"=>
              DEWOP(PAR(2),AREG1(1,15B),A))
      ELSE "AØCON LS BØCON"=>CONOP(A,B,5);
<EXP> ::= <ATOM,A>  ALS <EXP,B> ::=
       CASE (240B,514B,574B) OF SHIFTS;
<EXP> ::= <ATOM,A> LROT <EXP,B> ::=
       CASE (241B,204B,204B) OF SHIFTS;
```

The first statement defines the syntax for I LS J. The semantic part defines the case SHIFTS to consist of the set of semantic alternatives within the parentheses following the identifier SHIFTS. (Notice that the AØCON LS BØCON alternative is not part of the SHIFTS definition but is an alternative to it.) The list of constants following the identifier CASE is the "parameter list" for this particular instance (LS) of the case SHIFTS. These constants are referred to in the semantics for SHIFTS by PAR(Ø), PAR(1), and PAR(2).

The other two statements define the syntax and semantics for two other kinds of shifts. The semantics is defined by invoking SHIFTS with different parameter lists. The semantics are the same, but PAR(i) will refer to the i-th element of the appropriate parameter list.

A case, such as SHIFTS, may not be defined in more than one place. It is not necessary to have parameter lists be the same length for different instances of a case, but it is the user's responsibility to insure that parameters that are not supplied are not referred to. In any event, parameter lists must always contain at least one constant.

Several CASE semantics may appear as some or all of the alternatives in a semantic part. A CASE semantics may not be the semantics part of a conditional semantics alternative (i.e., "condition"=>CASE (...) OF FOO is illegal.)

2.3.3.4: The VALUE Kludge.

Occasionally, it is necessary to refer in conditional semantics to an argument of the production which can not be written as an identifier. This is not possible using the mechanisms defined so far. The VALUE kludge is a way to accomplish this. Take for example the syntax for conditionals. Relational operators are defined as follows:

```
<RELOP> ::= NE ::= VALUE 6 OF EQ;
            <RELOP> ::= '<' ::= VALUE 1 OF EQ;
```

and so on. The semantics specifies that these productions are special cases of a class of productions labeled EQ, and gives an 18-bit value to be associated with the particular production.

In syntax statements involving the class <RELOP>, it is possible to specify that any case of EQ may be recognized as matching a particular element in a semantic condition. This is done as follows:

1. In quoted unconditional semantics: prefix the quoted part by EQ/. The corresponding instance of RELOP in the syntax part must be named EQ. (Last line in the example below).

2. In semantic conditions: Prefix the condition by EQ/.

3. In semantics: The element may be referred to in quoted semantics by prefixing the quoted part by EQ/ as in (1). In semantic routine calls, the element may be referenced by STACK(i), where the element comes to the right of exactly i identifiers in the condition (or, in unconditional semantics, i arguments of the production).

The particular instance of EQ may be determined by referring to VAL(STACK(i)), with i as in (2) above. The value of this semantic routine call will be the value specified in the VALUE semantics defining the instance of <RELOP> in the expression being compiled.

Example:

```
<ST>    ::= <EXP,A> <RELOP,EQ> <EXP,B> => <ST,C> ::=
  EQ/"PØREG=Ø=>GO TO
 SØVAR"=>HOOK(P,P,DEWOP(32ØB+VAL(STACK(1)),
                                  REGOF(P),S)) ELSE
  LOCAL IF IN EQ/"NOT(A=B)=>GO TO IF; C; IF: Ø";
```

2.3.3.5: Priority Semantics.

Usually, the semantics for an expression is not executed until enough of the context of the expression has been parsed in order to determine the particular special case involved. If it is known that no special cases involving subexpressions of a production exist, or if it is desired for other reasons to defeat the special case matching mechanism and invoke the semantics for a production immediately it is recognized, the PRIORITY semantics is used.

A PRIORITY semantics is a semantic part consisting entirely of

PRIORITY n s

where n is a digit in the range Ø-7, and s is a semantic part containing no conditional semantics or ELSE alternatives.

n is interpreted as a three-bit mask, and the semantics s for the production is executed immediately, bypassing the special case matching process, in the event that the current priority mask of the matching routine and n have any bits set in the same position.

The priority mask of the matching routine is set to k by the semantic routine SETPRI(k). The interpretations of the bits are:

1. Normally set. Used to force immediate semantic execution under normal circumstances.

2. Set during interpretation of double-quoted expressions. In this mode, semantics are not interpreted unless they are PRIORITY 2. The output of the parser is stored instead, for use as data by the compiler. However, PRIORITY 2 semantics are executed, providing a way to terminate this state.

4. Set during synonym processing (LET statement).

2.3.4: Syntactic Ambiguity and How to Make it Work for You.

The IMP72 parser allows ambiguity in the syntax. For example, notice that in many examples above, we present different definitions of the construct ABS(E). The syntax allows two interpretations of this construct: the special construct we are defining, or a call on the subroutine ABS. The parser notes the two interpretations, and chooses between them, as follows:

1. If an identifier is an expression in one interpretation, and a terminal symbol (i.e., appears as itself in the production, such as ABS in the example of Section 2.3.1), in the other interpretation, the interpretation with the terminal symbol is chosen.

2. Otherwise, the interpretation involving the smallest number of different syntax rules is chosen.

3. Otherwise, the compiler makes an arbitrary choice.

Thus, referring to the example in Section 2.3.3.2 of the construct /variable/, the special case /variable/←expression was defined by a special case in the semantics. It would alternatively be possible to define the case by introducing a deliberate syntactic ambiguity, as follows:

```
<VBL> ::= / <VBL,A> /
                        ::= DEWOP(55ØB,AREG1(1,13),A);
<EXP>  ::= / <VBL,A> / ← <EXP,B>
                        ::=
 HOOK(A,B,DEWOP(542B,REGOF(FETCH(B)),A)
```

The expression /X/←5, e.g., could be recognized in two ways, by using the first syntax statement above plus the rule for <VAR>←<EXP>, or by using the second syntax statement above. But according to rule 2, the parser chooses the second interpretation, as involving one syntax rule as opposed to two rules for the first interpretation. This is precisely the choice that is desired.

2.3.5: Peaceful Co-Existence with Your Extensible Compiler.

This section presents the results of some experience with syntax and semantic definition in IMP72, in order to help others avoid some problems.

The parser works by carrying along all possible parses of the program up to the symbol it is reading at the moment. It usually manages to resolve all ambiguity every couple of symbols at most. If a syntactic production is entered which requires it to read a large number of symbols before it can choose which syntax rule it is following, the parser may require inordinate amounts of space and time. This condition may be diagnosed by looking at the compiler statistics at the bottom of the source program listing. If the Max. Parse Space goes over 1500+500, or the Max. Output Space is over 150, the condition may exist.

An example of syntax which may cause the problem is:

```
<EXP> ::= <EXP,A> => <ST,B>;
<EXP> ::= <EXP,A> => <EXP,B> ELSE <ST,C>
```

When the compiler sees A=>(.....), it is not able to decide until the right parenthesis whether it is trying to form an ST or an EXP. Thus it must carry along two parses while it is parsing the entire contents of the parentheses, which may be an arbitrarily long expression. The syntax IMP uses is

```
<EXP> ::= <EXP,A> => <ST,B>;
<EXP> ::= <EXP,A> => <ST,B> ELSE <ST,C>
```

which avoids the parser problem (although it introduces the necessity for parenthesizing the ELSE clause). The method for storing syntax rules combines all rules whose right-hand parts begin identically, up to the point at which they differ. Thus, the parser stores these two rules as:

```
                        =:: <EXP>
                       /
<EXP,A> => <ST,B>
                       \
                        ELSE <ST,C> =:: <EXP>
```

and only carries along one parse until it has finished parsing the <ST,B>.

Another difficulty which may arise is that special semantic cases may be specified in a syntax statement, but the compiler will refuse to recognize them. This may arise from the way in which the patterns for special cases are

recognized. Semantics are performed for subexpressions of a syntax production from left to right. For an element in a syntactic conditional to be recognized, it must already have been reduced by semantics to one object. An example will illustrate the pitfall:

```
<EXP> ::= <ATOM,A> NE <EXP,B> ::= .....;
<ST>  ::= <EXP,A> => <ST,B> ::=
          "A NE Ø=>B" => ..... ELSE .....
```

where the ..... represents semantics whose exact form is nonessential to the point being illustrated. If the statement

```
A NE Ø=>X←X+4
```

is encountered by the compiler, an attempt to match the special case "A NE Ø=>B" is made, but fails since X←X+4 is not a single object; the semantics for ← and + have not yet been performed. Since the special case fails, the semantics for the subexpressions of the statement are invoked, from left to right. First, the semantics for A NE B is performed. Checks are made at every step to see if the current expression matches some special case, but now A NE Ø is a single object, and will only match patterns of the form "X=>...". Thus the compiler will fail to recognize the special case "A NE Ø=>B".

It is in order to avoid this problem that IMP has separate productions for A=>B and A <RELOP> B=>C (where <RELOP> is the class of relational operators).

3: How to Compile and Run IMP72 Programs.

3.1: Compiling Programs.

The IMP compiler is called by the DECsystem-10 command R IMP (on TENEX, the command IMP). The compiler will respond by typing its version number and an asterisk. It will now accept a command line in the following format:

obj,list←dev:file.ext[pj,pg]/a/b(cd)

All fields are optional. When all are specified, the compiler will compile a program from source file file.ext[pj,pg] on device dev. A listing will be written on file list.LST, and a relocatable object program will be written on file obj.REL. Listings requested by compiler switches will be written on file file.LST if no list file is specified. If the compiler detects an error, a listing will be produced from that point in the source file on. If the extension "ext" is omitted, the compiler will look for a file with null extension, and then for file.IMP.

A, b, c, d are compiler switches, as follows:

/A Produce an Assembly listing.
/C Continue after this file (see below).
/H Help - list the switches available.
/L Produce a source listing.
/R Compile Re-entrant (pure) code.
/U Exit to save compiler.
/V Exit to save low segment of compiler (see below).
/Y List source program on TTY as it is compiled.

If you want to compile the IMP program on file FILE, the simplest way is to give the compiler the command string FILE. If a listing is required, the string FILE/A/L will do.

The /C switch enables a file containing only syntax and declaration statements to be compiled for the purpose of making a version of the compiler for an augmented dialect of IMP. After the file has been compiled, the compiler returns with another prompt, and the /V or /U switches may be used.

The /U switch causes the compiler to exit, after doing some housekeeping. At this point, the monitor command SSAVE NEWIMP will make files NEWIMP.LOW and NEWIMP.SHR (or a file NEWIMP.SAV on TENEX) which will be a new version of the compiler, containing any syntax and declarations just compiled.

The /V switch is similar to the /U switch. It is used to make an augmented version of the compiler while retaining the previous shared high segment. The saved version of the compiler will not contain a sharable high segment, but will, whenever it is run, fetch the high segment from the previous version of the compiler. On TENEX, or if the high segment has been altered, this can not be done. In this case, the error message HIGH SEGMENT NOT SHARABLE is given, and the same action is taken as in the case of the /U switch. It is suggested that the /V switch be used wherever possible, since the size of the high segment is considerable, and it is desirable that two different versions of the compiler be able to share the same high segment, and that identical copies of the high segment not be stored on disk.

If the compiler should encounter a fatal error, such as an illegal memory reference, and terminate abnormally, it is possible to salvage the listing file up to that point. Execute the command REENTER, and look for a file ØØ5IMP.TMP, where ØØ5 is your job number.

If the monitor at your installation has been modified appropriately, it will recognize files with the extension IMP as IMP72 source files, and the monitor commands COMPILE, LOAD and EXECUTE may be used to manipulate IMP programs. The switch "/IMP" may be used to inform these commands that a file with a non-standard extension does in fact contain an IMP program.

3.1.1: The Compiler Listing.

The source listing of the compiler has a number at the left of each line of code. This indicates the level of parenthesis nesting at the beginning of the line. Parentheses contained within # or ' signs are not counted.

3.2: Compilation Error Diagnostics.

The compiler reports on three classes of errors: fatal errors, errors, and advisory errors. Advisory errors indicate a condition which may not be an error but should be brought to the programmer's attention. They are labeled ADVISORY. Fatal errors terminate compilation immediately. Regular errors do not terminate compilation, but subsequent compilation may be affected.

The philosophy behind the error diagnostics in IMP presumes a certain amount of maturity on the part of the programmer. He is less restricted than in a language such as FORTRAN, but he is also less protected from his own errors. Thus, a number of potential error conditions that may be used usefully are not checked for by the compiler. This is especially true of syntax statements. This is not to imply that few useful diagnostics are provided, however.

The following are the error diagnostics provided by the compiler. They are regular errors unless otherwise indicated.

ATTEMPT TO EXECUTE UNDEFINED SEMANTIC ROUTINE (Fatal): It wasn't bad enough you referred to a nonexistant semantic routine in a syntax statement (for which you already got an error message). You had to go and try to compile an instance of the producion. This you won't get away with.

BAD MODIFIER: A semantic condition has an identifier with a bad modifier.

BAD RESULT OF SEMANTICS; IGNORED (Advisory): A semantic routine has returned a value for a semantic part which is not an object on the semantics routine stack. This may or may not cause trouble later on. If you have not added semantic routines to the compiler, report the error to the person responsible for maintaining the compiler.

CANNOT CREATE .REL FILE (Fatal): For some reason, which probably has to do with the operating system, a .REL file for your object program could not be created. Perhaps another user is writing on the file, or there is no space available on the storage device.

CALCULATED CONSTANT IS SUBSCRIPTED (Advisory): An expression has been evaluated to a constant and now an attempt is being made to subscript it. Since the calculation yields only one word, this doesn't make much sense.

CANNOT GET TWO CONSECUTIVE REGISTERS (Fatal): The assembler reports that an operation (such as divide) which requires two consecutive machine registers was not able to find two consecutive registers not in use.

CASE CONTAINS SUBEXPRESSION WHICH IS CASE BUT WONT GENERALIZE (Advisory): You are warned that a semantic condition in a CASE definition contains a subexpression which is also an instance of the production in the syntax part. The main expression will generalize to other instances of the CASE, but the subexpression won't.

CASE UNDEFINED (Fatal): You tried to reference a CASE semantics, but the case was never defined.

CODE GENERATION STACK DID NOT REDUCE TO ONE ITEM: The program parsed correctly, but semantics for some of the program could not be found. If you added your own syntax in the program, check it. Otherwise, report the problem to the person responsible for maintaining the compiler.

DEBUGGING PROGRAM NOT PRESENT (Advisory): A debugging switch has called for a printout not available from the user version of the compiler. Recompile the program using the version containing the debugging routines.

DUPLICATE TAG: The same label was used as a tag in two (or more) places.

ERROR CORRECTOR GIVES UP. (Fatal): The error corrector was unable to corrrect a syntax error.

ERROR CORRECTOR GIVES UP AT % (Fatal): The error corrector was unable to correct a syntax error, and read past the % sign while trying.

ERROR IN ADDCHAR - NAME > 5Ø CHARACTERS: A syntactic class name is too long.

ERROR IN SYNTAX READ-IN: There is a syntax error in the syntax for syntax during the first stage of compiler bootstrapping. Will not occur except when compiler is first being generated.

FREE STORAGE EXHAUSTED: Your program has demanded more space to compile than is available as user core. If you had syntax errors, you may be able to compile when the errors are repaired. Otherwise, make your program smaller or find a bigger machine (or a stiff drink).

FREE STORAGE UNLINKED: Fatal error caused by a compiler bug. Give the listing produced to the person responsible for maintenance of the compiler.

IN AREG1 - I OR J OUT OF RANGE: Argument to AREG1 negative or greater than 177B.

N NOT A CONSTANT IN N LONG: N may be a constant expression but must contain no variables (except those syntactically defined as constants).

NAME TOO LONG (Fatal): You used an identitfier longer than 100 characters in a syntactic condition. Why?

NESTED SUBROUTINES: Due probably to mismatched parentheses, a subroutine definition has been initiated before the previous one was closed out.

NON-OCTAL PJ OR PN: In FILE designator of PRINT or READ statement, that is.

NUMBER MISSING OR OUT OF PLACE IN SEMANTICS IDENTIFIER: In a semantic conditional, you had a number in the wrong format or wrong place.

OUT OF REGISTERS (Fatal): The assembler reports that your program has managed to fill up all available hardware registers and then some. If you are using explicit registers, perhaps you have generated an expression that you think fits in one register but isn't coming out that way. If not, you may just have a very complicated expression somewhere. Split it up.

PREMATURE END OF INPUT FILE: Compiler read the entire input file but did not encounter the %% which terminates IMP programs. Perhaps you: 1. Only had one %. 2. Failed to match # or ' with another one, so that the % sign got absorbed in a constant/comment. Check the numbers down the left margin of the listing to see when they stopped following the parenthesis nesting of the code. The compiler offers a chance at redemption in the form of a prompt for a new file name. Hitting the RETURN key will cause the compiler to insert its own %% signs to complete the compilation.

PRODUCTION DUPLICATES ONE ALREADY IN SYNTAX (advisory): The production in a syntax statement is identical to one already in the syntax. This is not an error, and is used to add special semantic cases to already defined syntax.

PUSHDOWN STACK UNDERFLOW ERROR (Fatal): Indicates a compiler bug. Inform the person responsible for maintaining the compiler.

QUESTIONABLY DEFINED VARIABLE (Advisory): The indicated variable was referenced but never (1) had a value assigned to it, or (2) appeared as an argument of a subroutine call, or (3) was declared COMMON.

QUOTED SEMANTICS NOT AN INSTANCE OF PRODUCTION: A semantic condition was not an instance of the production in the syntax statement in which it appeared.

REGISTER CONFLICT(S) (Advisory): A machine register was referred to which was already in use by the compiler. The operation was performed as requested, but prudence dictates a close inspection of the assembly listing of the code generated. This message appears only once regardless of the number of register conflicts, and is followed on the list file by error messages for each conflict, indicating the register involved and the address of the first reference producing a conflict, relative to the beginning of the object code.

SEMANTICS STACK OVERFLOW (Fatal): Congratulations! There is exactly one major table in the compiler which is not dynamically expandable, and YOUR program has stuffed it full. This can only happen through the use of a very complex semantic routine call in a semantics part of a syntax statement, or maybe a bug in a semantic routine, or a very long list of names in your program. Cognoscenti will appreciate the information that this may result from lots of calls to ENSTACK in one semantics.

STACK UNDERFLOW IN FSUB (Fatal): Indicates a compiler bug. Inform the person responsible for maintaining the compiler.

SUBSCRIPTED REGISTER - IGNORED: The subscript on a register is disregarded.

SYNTACTIC AMBIGUITY; UNRESOLVED AT %: The parser has found a syntactic ambiguity which it was not able to resolve by the end of your program. If your program contains syntax statements, check them; otherwise report the error to the person responsible for maintaining the compiler.

SYNTACTIC AMBIGUITY: The compiler was able to interpret a portion of your program in two conflicting ways, and was not able to decide which way you had in mind.

The ambiguous segment of the program is indicated on the listing (after a fashion) by a list of the identifiers (variable names and constants) which appear in the segment, printed in reverse order. Some constructs which may produce this error are: A=>B=>C ELSE D, which is really ambiguous and should be parenthesized explicitly, and (A RELOP B)=>C, where RELOP is any relational operator, which is not ambiguous, and will generate correct code, but the parentheses may be removed to get rid of the error message.

SYNTAX ERROR: The parser has not been able to interpret your program as a legal IMP expression. It has attempted to continue the compilation by replacing part of your program at the point of the error by what it hopes is is a correction of the error. It is possible that more errors will result from this correction later on. (On the other hand, in contrast to the previous PDP-1Ø IMP compiler, which quit at the first error, there will at least BE a later on.)

The offending line will be typed on your teletype, with a line feed inserted at the point at which the error was detected.

Syntax errors are handled by the error-correcting routine, which may take some time on certain errors. When the error has been corrected, two asterisks are printed after the message ** SYNTAX ERROR. If this seems to be taking an unduly long time, you may want to control-C, execute the command REE, and then look for a file named ØØ5IMP.TMP (where ØØ5 is your job number), which contains your listing up to the point of the error.

TOO MANY ARGS TO SEMANTIC RTN OR PARAM STACK UNDERFLOW (Fatal): Either you tried to execute a semantic routine with more than 1Ø arguments, or else there is a compiler bug.

TOO MANY LOCAL SYMBOLS IN REFERENCED SEMANTICS (Fatal): You tried to reference quoted semantics with more than 1Ø local symbols.

TOO MANY REFERENCES TO ONE REGISTER (Fatal): Either you have devised a way to refer to a register or register variable four thousand ninety-six times in your program (unlikely), or else a linked list of object code has gotten tied into a loop (regrettable, but conceivable). If the latter is the case, report to the person responsible for maintaining the compiler. This error has also been known to crop up when programs which have syntactic errors are being compiled. Although the error corrector always makes

corrections which are SYNTACTICALLY CORRECT, there is no guarantee that they will always be SEMANTICALLY MEANINGFUL. If you get this error message and your program contains syntactic errors, correct the errors and re-compile. This error message should go away.

TOO MANY SYNTAX ERRORS (Fatal): A large number of syntax errors have been detected in your program, and the compiler sees no point in proceeding.

TWO LENGTHS IN SAME DECLARATION: Two lengths in same declaration.

VALUE NOT A CONSTANT: Value in VALUE semantics must be an 18-bit constant.

VECTOR APPEARS IN MORE THAN ONE DECLARATION: The same variable name was declared N LONG more than once.

3.3: Loading and Running.

On DECsystem-1Ø, you may load and run your object programs by using the appropriate monitor command (.EX, .LOAD or .DEB), with a list of the files you wish to load. File IMPLIB.LIB should be loaded in library search mode. Example:

```
.EX URPROG.REL,URSUBR.REL,/LIBRARY,IMPLIB.LIB
```

On TENEX, IMP object programs must be loaded by the LOADER subsystem, and run with the START command. See the TENEX User's Guide and the DECsystem-1Ø Assembly Language Handbook for more information on the loader.

If your program does any I/O, it should call FINI (see Section 2.2.7) to complete the I/O operations.

If your program should bomb out, the monitor command .REE will usually close out your files without losing any output. DDT users may activate the symbol table using the name of the .REL file. This is the name specified in the CALL ME statement, if there was one in the source file, or else the name of the source file.

3.4: Making a New Compiler.

If it is desired to create a new IMP72 compiler from scratch, the following procedure should be followed:

1. Make a version of the compiler containing the syntax on file IMPSYN.IMP. This syntax is necessary to compile each compiler subroutine. The way to make the version of the compiler is: run the compiler. respond to the * prompt with IMPSYN.IMP/C. When another * prompt is given, type /V. The compiler will exit to monitor level. Type SSAVE FOO. Now you have a version of the compiler named FOO. Use it to compile the source files for the compiler. Be sure to specify the compiler switch /R when compiling each source file, in order to generate a sharable high segment for the compiler.

2. Load and execute all .REL files for the compiler, together with the IMP I/O library (IMPLIB.LIB). If you desire a compiler with the compiler debugging facilities, load DEBUG.REL; otherwise ignore the

resulting undefined globals and charge on. In any event, ignore the undefined global PEEK, which will not be called (except on the Yale computer). The file R.CMD may be used to load the compiler by the DECsystem-1Ø command LOAD @R, or the command string R.CMD@<ALTMODE> to the TENEX LOADER subsystem.

3. The compiler will start by reading part of file SYNTAX, and then come back with a SYNTAX BOOTSTRAPPED message and an asterisk. Tell it to compile the rest of the syntax on SYNTAX by typing /C<CR>

4. When (after several minutes) the compiler returns another asterisk, you may type /V<CR>, and then save the core image, which is now an IMP72 compiler.

5 If you wish to add new syntax, say on file NEWSYN, instead of /V type NEWSYN/C<CR>, and proceed to step 4.

4: Internal Documentation of the IMP72 Compiler.

This section provides an overview of the organization of the IMP72 compiler. Documentation of specific routines, data bases, etc., is included at the beginning of each file of source code. The intent of the present section is to provide a frame of reference for the reader who wishes to tinker with the compiler. For those with a more intellectual interest in the compiler's workings and design, [Bilofsky 1973], [Irons 1971] and [Weingart 1973] may be more edifying.

The compiler is programmed modularly wherever possible, at some cost in efficiency. (This cost has been substantially reduced by replacing many subroutine calls by in-line syntactic macros.) This philosophy was adopted in order to enable replacement and modification of portions of the compiler with minimal repercussions. Accordingly, there is no common data base in the compiler (minor exceptions exist for the sake of efficiency in the assembly phase). Necessary information concerning another module's data base is acquired via subroutine calls.

The compiler may be divided conceptually into the following sections:

SOURCE FILES	DATA BASE	FUNCTIONS
I. Housekeeping.		
FREE	Free storage.	Free storage module. Maintains free storage, which is dynamically allocated and used for virtually all tables.
DIR	Directory.	Directory module.
STACK	Semantic routines' stack	Semantic stack module. Maintains a stack for parameters of semantic rtns.
"	(provided by caller)	Bit matrix module. Creates and accesses arbitrary bit matrices.
"	Pushdown stack	Pushdown stack module. Provides a PD stack for anyone who wants.
IMP	(none)	Driver program.
DEBUG		Debugging module, which may be loaded at compiler generation time.
PMOD		Formatted print module.
II. Housekeeping & Parsing.		
LEX		Input and lexical analyzer.
RSYN & RSYN2		Syntax bootstrap routine, and semantics for syntax statements.
III. Parsing.		
PARSE	Parse arrays.	Parse module. Calls heavily upon GRAPH and ERCOR.
ERCOR	Parse arrays.	Error correction and syntactic disambiguation.
GRAPH	Syntax graph & connectivity matrices.	Syntax graph module. Constructs and accesses the syntax data base.

IV. Semantics.

ENTREE	Code generation tree.	Code tree module. Builds and accesses code generation tree.
COTREE	Code generation stack.	Code generation module. Calls heavily on code tree module. Does code tree matching & invokes semantic routines. Maintains code generation stack.
DOSEM	Semantics	Semantics module. Builds and accesses semantics array. Oversees semantics stack.

V. Semantics & Code Generation.

CODE & CODE2 & CODE3	Entries in semantics stack; code arrays.	Semantic routines, mostly target-machine-dependant. Generate intermediate format object code.
IMPSEM	Code and register arrays.	Machine-independant semantic routines. Declarations and register assignment.

VI. Code Generation and Housekeeping.

AMOD	Code array.	Assembly module. Performs some final code optimization.
RMOD	Register array.	Register allocation module.
OMOD	Object code.	Relocatable file generation module.

4.1: Parsing.

The general flow of control within the compiler is as follows: PARSE obtains a stream of input symbols (in the form of directory indices) from LEX, and produces a stream of parser output in the form of terminal symbols (negative directory indices) and production numbers (positive integers n, conventionally indicated by [n]. In fact, n is an index in the semantics array, for historical reasons, but need not be.). The parser output is in Polish Postfix form.

The parser is a bottom-up parser (Irons 1971), driven through one cycle by each input symbol. At the end of the cycle, it has a set of possible parses for the input string so far. It takes one of the following actions, depending on the number of parses in the set.

1. If there are no parses, a syntax error has occurred. ERCOR is called to generate a string of symbols which may be inserted into the parse to form a syntactically correct expression. (ERCOR uses the philosophy, but not the exact method, of (Irons 1963)).

2. If there is more than one parse, AMBIG is called to decide if an ambiguity has generated multiple parses, and to discard the redundant parses if this is the case.

3. If there is one unique parse, the portion of the parser output stream generated since the last time there was one unique parse is passed to the semantics portion of the compiler by calling COTREE.

All other functions performed by the compiler, including addition of syntax and semantics to the compiler tables, and invoking the assembly phase, are performed by semantic routines invoked by the productions which PARSE passes to COTREE for interpretation. Thus the entire flow of control of the compiler is directed by the syntax and semantics of the language.

4.2: Semantics.

COTREE maintains a code generation stack, which starts off as the parser output in postfix form, but is collapsed as the productions in the stack are applied to their arguments, by invoking the appropriate semantics for the productions, to produce a single stack entry (semantic object) which is inserted back in the stack in place of the production and its arguments. This could always be be done at the bottom of the stack (we will consider the working end to be the bottom), as soon as the parser supplied a production, were it not for the special semantic cases. These require that a production not be applied to its arguments until enough additional parser output has been supplied to ensure that the production is not part of some special case.

In order to do this, all special cases are kept in a code generation tree, which is built and accessed by routines in the code tree module, ENTREE. The special cases are entered in the tree as nodes which may match either a particular class of objects (as specified by the modifiers in the semantic condition see Section 2.3.3.2), or a particular production. When a new successor is added to a node, it is carefully inserted among existing successors so that the order of successors of a node, from left to right, is in order of increasing generality. This is done by ENTREE, and by subroutines INSNO and COMPNO. This order insures that the most specific case in the tree is the one to match a given piece of parser output (subject to the difficulties mentioned in Section 2.3.5), since nodes are searched left-to-right while attempting a match.

The definition of unconditional semantics for a production causes the most general possible case of that production to be entered in the code generation tree, ensuring that if no special case is found the semantics for the general case will be invoked.

The terminal nodes of the code generation tree must always match some production. In order to avoid excessive fanout to these nodes, the terminal nodes coming from any given node are condensed into one. The entry for each production in the semantics array contains a pointer to a list for that production, in the array PROSEM (part of the semantics module), containing the names of all the terminal nodes in the code generation tree where that production may match, and a pointer back to the semantics array entry for the semantics to be invoked at that node. The PROSEM entry for a production also contains a bit vector telling at which levels of the code generation tree the production may match some node, either terminal or otherwise. This information makes it possible for COTREE to ascertain in many cases that

no match is possible starting at a given point in the code generation stack, without having to search the entire tree.

Certain non-terminal nodes in the code generation tree may match a class of productions defined by a VALUE kludge.

COTREE performs the following algorithm: It is called once from PARSE with each item of parser output. The item is added to the bottom of the code generation stack. If it is a terminal symbol, no other action is taken. If it is a production number, an attempt is made to match some substring in the stack with some pattern in the code generation tree. A match is attempted starting at each node in the stack, starting at the top.

1. If a match is found which might be continued past the end of the stack, no action is taken, and COTREE returns control to the parser for more parser output. In this case, more context is needed to determine whether this is a special case.

2. If no match is found at any position, another symbol is requested from the parser. This is in fact an error condition, but is not checked for until completion of the parse.

3. If a match is found, all the way through to a terminal node of the code generation tree, the appropriate semantics are applied.

The semantics are applied by moving the objects in the matched section of the stack onto the semantics stack, and calling the semantics module, DOSEM, with the index in the semantics array of the semantics to be invoked. DOSEM will usually supply a single object as the result of the semantics. COTREE then inserts this object into the stack in place of all the objects and productions matched, and goes back to look for another pattern match.

Quoted semantics are stored in the semantics array essentially as parser output, with special indicators for local symbols and for places where the arguments are to be inserted (copying the associated code, if any, for all but the first insertion of each argument). DOSEM performs quoted semantics by calling CODENT, in the code generation stack package, to add each item in the quoted semantics onto the bottom of the code generation stack. CODENT also sets a flag for COTREE, so that when DOSEM returns control to COTREE special action is taken to rearrange the stack so that the entire result of the quoted semantics is inserted in place of the matched portion of the stack. Thus, it is quite possible to have semantics generate more items on the stack than were necessary to invoke it. This process is

used in the semantics for IMP72 conditionals, where several quoted semantic rewrites of a conditional expression may be used to put it into a form which generates good object code.

There is also a special switch in COTREE which causes it not to enter the parser output in the code generation stack at all, but rather to store it in an array for later interpretation by certain semantic routines. The semantic routines MAXWELL and MAXEND turn this switch on and off repsectively, the production invoking the latter making use of the PRIORITY mechanism to avoid being stored and ignored itself. This mechanism is used to implement quoted semantics and semantic conditionals.

4.3: Semantics and Code Generation.

Semantic routines are invoked by DOSEM, which merely interprets the entries in the semantics array. The routine is completely explained by the data structure, which is well-documented on the source file.

(The confusing routines in the semantics package, and there are very confusing ones indeed, have to do with the semantics for syntax, which is discussed below.)

4.4: How Extensibility is Implemented.

The syntax of IMP72 is stored in a syntax graph and two associated connectivity matrices, as described in (Irons 1971), except that the graph is not back-optimized, in order to be able to add new productions without great fuss. New productions are added to the graph, and the matrix updated, by subroutine GRAPH, which is called with one production. The parser accesses the graph via other subroutines in the syntax graph module.

Initially, the graph is empty. Subroutine RSYN has built into it a syntax and semantics for very simple syntax statements. The process of creating an IMP compiler begins by calling RSYN (from the driver program IMP) to read a similar syntax and semantics from file SYNTAX. Once this is done, the compiler presents an asterisk, and is normally instructed to continue compiling from file SYNTAX. (See Section 3.4 for the exact procedure.) Now, however, the compiler is bootstrapped, and uses the syntax and semantics in its internal tables. The first statements it reads are the syntax and semantics for more complex syntax statements. After this comes the syntax for the computational portion of the language. When this has been compiled, the compiler is saved. Thus, except for lexical conventions and the syntax built into RSYN (which is only used to bootstrap the compiler), the entire syntax and semantics for IMP72 comes from file SYNTAX and the semantic subroutines referred to on it.

The other routines on files RSYN and RSYN2 implement the semantics for syntax statements. They were at once the first and the most complex semantic routines implemented, and, to put it kindly, are rather obscure. The routines SYTRM, SYNT, and SYNTS build up a production in array SN which is then entered in the syntax graph by semantic routine SYNTAX. At this point, a list of the names of the arguments of the production is in array NA for use in compiling the semantics.

Basically, semantics are compiled into array SEMS in the semantics package via calls on SETSEM. Semantic routine calls are compiled by semantic routines SEMP and SEMR, which check NA to see if identifiers are arguments of the production.

Quoted semantics are processed by semantic routine QUOSEM, which converts parser output into semantics which it enters in the semantics array via SETSEM.

When a semantic condition is encountered, semantic routine QUOSEM enters the pattern in the code generation

tree with a call to ENTREE, which returns a list of the formal arguments of the condition. QUOSEM then calls SEMFIX (in the semantic package) to process the semantics, which have already been entered in the semantics array, to conform with the new list of formal arguments.

When a modifier for a VALUE kludge is encountered on quoted semantics, semantic routine REPVAL performs the necessary alteration on the parser output before QUOSEM gets it.

Semantic package routine SEMFX1 is called to perform a similar alteration on default case semantics in the event that a VALUE kludge applies to them.

5: Distribution Procedure for IMP72.

Until further notice, persons interested in receiving copies of the IMP72 compiler may follow this procedure:

The master copy of the compiler will be maintained by the person whose name and address appear below. From time to time, as improvements are made and bugs are unmade, a new version of the compiler will be issued, containing all updates and corrections known to that point, and compatible with previous versions except in cases of extreme necessity.

A version will be released on two DECtapes, containing the following:

1. Files IMP.LOW and IMP.SHR: the compiler.
2. A file IMP.RNO: the RUNOFF source for this manual.
3. A file IMPLIB.LIB: the I/O library for IMP72 programs.
4. Source files for the compiler (with extension IMC).
5. Source files for the I/O library (with extension IML).
6. A file FORTIO.MAC, which contains utility subroutines included in IMPLIB.LIB. FORTIO.HLP is also included for the curious.
7. A file R.CMD which can be used to load the compiler via .LOAD @R.

Anyone wishing to obtain a copy of the latest release is invited to send two DECtapes to:

WALTER BILOFSKY
BOLT BERANEK AND NEWMAN, INC.
5Ø MOULTON STREET
CAMBRIDGE, MASS. Ø2138

Well-documented information concerning specific bugs in the compiler, if couched in gentle and soothing terms, and suggestions, briefly expressed, will also be received cheerily at the above address. There is no guarantee, however, that the fantastic improvements you have made in YOUR IMP72 will be included in later releases of OUR IMP72. Therefore, those interested in utilizing future releases with minimal anguish factor are advised to confine modifications wherever possible to

1. New syntax, on files to be compiled by IMP72, or, failing that,

2. New semantic subroutines on source files separate from the original compiler source files.

Notwithstanding any of the above or below, neither Yale University, its Department of Computer Science, its graduate students, faculty or employees, Bolt Beranek and Newman, Inc., the author, their immediate families, friends or neighbors, assert or assume any responsibility for the accuracy of the material contained herein, for the continued maintainence, development, or availability of the IMP72 compiler, or for any difficulties resulting from its mis-use, -application, or -understanding, or for your two DECtapes.

Caveat Usor! Good luck!

Appendix I. Utility Library Subroutines.

The IMP run-time library, file IMPLIB.LIB, contains a package of subroutines called FORTIO, originally programmed by Ken Shoemake at the Yale Department of Computer Science to enable FORTRAN programmers to interface with the DECsystem-1Ø monitor from a higher-level language. These routines are useful to IMP programmers working under DECsystem-1Ø, and are therefore described here.

Many of the functions perform the DECsystem-1Ø UUO's of the same name, and more information about them can be found in the DECsystem-1Ø Assembly Language Handbook. In the listing below, optional arguments are enclosed in angle brackets. Unless otherwise stated, string arguments to these functions are IMP ASCII strings.

SAVREG(array): saves all registers in the 16-long array array.

RSTREG(array): restores all registers from the 16-long array array.

ARGCNT(): returns the number of arguments the calling program was called with; undefined for a main program.

CALLI(fn,<v1,v2>): performs the CALLI UUO. Fn is either the function name or the corresponding octal number. V1, and v2 if appropriate, are the values for AC and AC+1. CALLI returns the value returned in the accumulator from the UUO. N.B.: Where fn is a constant less than 4Ø96 and v1 but not v2 is present, this will compile as an in-line CALLI (see Section 2.2.9).

NOSKIP(): returns -1 if the last call to the routine CALLI resulted in a no-skip return, Ø otherwise.

SIXBIT(stg): returns the ASCII string stg (six characters or less) converted to SIXBIT.

ASCII(sixb): returns the sixbit word sixb, converted to ASCII. The second word of the result is returned in register 1R.

RAD5Ø(stg,<bits>): returns the value of string stg as radix 5Ø, with bits in the four code bits.

RADX5Ø(sixb,<bits>): returns the value of the sixbit word sixb as radix 5Ø, with bits in the four code bits.

ASCI5Ø(r5Ø,<bits>): returns the value of the radix 5Ø symbol r5Ø converted to ASCII (second word in register 1R). If bits is present, it is set to the value of the code bits.

SIX5Ø(r5Ø,<bits>): returns the value of the radix 5Ø symbol r5Ø converted to sixbit. If bits is present, it is set to the value of the code bits.

OPEN(ch,ary): performs the OPEN UUO on channel ch with the address of the three-word array ary as the effective address. Returns Ø if no error, -1 otherwise.

INIT(ch,stat,<dev,obuf,ibuf>): performs the INIT UUO with the specified information. 'DSK' is the default device, and if obuf or ibuf contains a left-justified ASCII 'Ø' the corresponding buffer is omitted in the call.

INBUF(chan,n): performs the INBUF UUO with the specified information.

OUTBUF(chan,n): performs the OUTBUF UUO with the specified information.

RENAME(chan,ary)
RENAME(chan,name,ext,<pj,pg>): performs the RENAME UUO using, in the first form, the argument block in the array ary, and in the second form, the arguments shown. Returns Ø if no error, otherwise -1-(error nr.).

LOOKUP(chan,ary)
LOOKUP(chan,name,ext,<pj,pg>): performs the LOOKUP UUO; similar to RENAME q.v.

ENTER(chan,ary)
ENTER(chan,name,ext,<pj,pg>): performs the ENTER UUO; simplar to RENAME q.v.

RELEA(chan): releases the specified channel.

CLOSE(chan,<n>): performs the CLOSE chan,n UUO.

IN(chan,<LOC(arry)>)
INPUT(chan,<arry>): performs the IN UUO . Returns -1 for an error return, Ø otherwise.

OUT(chan,<LOC(arry)>)
OUTPUT(chan,<arry>): performs the OUT UUO. Returns -1 for an error return, Ø otherwise.

GETSTS(chan): returns the result of a GETSTS for the specified channel.

SETSTS(chan,status): performs the SETSTS UUO for the specified channel and status word.

STATO(chan,bits): returns -1 if a STATO for the specified channel gave a skip return, Ø otherwise.

STATZ(chan,bits): returns -1 if a STATZ for the specified channel gave a skip return, Ø otherwise.

GETCHA(): returns the number of a channel which is not open, starting with channel 17B. If no channels are available, returns -1.

MTAPE(chan,n): performs the indicated MTAPE function.

UGETF(chan): returns the result of a UGETF UUO on the specified channel.

USETI(chan,blknum): performs the USETI UUO for the specified channel and block.

USETO(chan,blknum): performs the USETO UUO for the specified channel and block.

INCHRW(): returns the value of a right-justified character from the teletype, waiting until one is typed.

OUTCHAR(char): types the specified right-justified character on the teletype.

INCHRS(): returns a character (right-justified) read from the teletype, or -1 if none has been typed.

INCHWL(): returns a character (right-justified) read from the teletype, waiting until a break character (e.g., carriage return or altmode) has been typed. This is the preferred way to read from the teletype as it allows the person typing to edit his line using rubout and control-U.

INCHSL(): same as INCHWL, but returns -1 if a line has not been typed.

RESCAN(bit): backs up the teletype scan past one break character, in order to allow rereading of the last line (possibly a command) typed. If bit is 1, returns -1 if no command is in the buffer.

SKPINC(): returns -1 if a character has been typed, Ø otherwise.

SKPINL(): returns -1 if a line has been typed, Ø otherwise.

OUTSTR(string): types the specified ASCII string on the teletype.

IONEOU(char): types a single 8-bit character on the teletype.

GETLCH(<line>): returns the line characteristics for the specified line; assumes the caller's line if none is specified.

SETLCH(value): sets the line characteristics for the caller's line.

CLRBFI(): clears the input buffer.

CLRBFO(): clears the output buffer.

Appendix II: Syntax of IMP72.

```
<NAM>       ::= NAM"
<SYM>       ::= SYM"
<SPT>       ::= <SPT,A> <SYM,B>
<SPT>       ::= <SPT,A> '<' <NAM,B>,<NAM,C> >
<SMA>       ::= <NAM,A> ( )
<SMA>       ::= <SPN,A> ( <SPL,B> )
<SPN>       ::= <NAM,A>
<SPL>       ::= <SPI,A>
<SPL>       ::= <SPL,A> , <SPI,B>
<SPI>       ::= <NAM,A>
<CSM>       ::= <SMA,A>
<SMP>       ::= <CSM,A>
<ST>        ::= <SPT,A>
<SAU>       ::= =
<ST>        ::= <SPT,A>':' ':'<SAU,C><SMP,B>
<STL>       ::= <ST,A>
<STL>       ::= <STL,A> ';' <ST,B>
<PG>        ::= <STL,A> '%'
<SPI>       ::= <SPI,A> + <SPI,B>
<SPT>       ::= <SPT,A> '<' <NAM,B> >
<SPT>       ::= '<'<NAM,A>>':' ':'='<'<NAM,B>>
<SSP>       ::= VALUE <NAM,A> OF <NAM,B>
<SSP>       ::= PRIORITY <NAM,A> <SMA,B>
<SSP>       ::= PRIORITY <NAM,A> <SSP,B>
<ST>        ::= <SPT,A>':' ':'<SAU,B><SSP,C>
<SMP>       ::= <SMP,A> ELSE <CSM,C>
<CSM>       ::= CASE (<NLIST,A>) OF <NAM,B>
<CSM>       ::= CASE (<NLIST,A>) OF <CASENAM,B> (<SMP,C>)
<CASENAM>   ::= <NAM,A>
<SQUOTE>    ::= "
<QUOSEM>    ::= <SQUOTE,A> <STL,B> "
<QUOSEM>    ::= <NAM,A> / <QUOSEM,B>
<CSM>       ::= <QUOSEM,A> => <SMA,B>
<CSM>       ::= <QUODEF,A>
<CSM>       ::= <QUOSEM,A> => <QUODEF,B>
<QUODEF>    ::= <QUOSEM,A>
<QUODEF>    ::= LOCAL <NLIST,A> IN <QUOSEM,B>
<VBL>       ::= <NAM,A>
<ION>       ::= <VBL,A>
<ATOM>      ::= <ION,A>
<EXP>       ::= <ATOM,A>
<ST>        ::= <EXP,A>
<ION>       ::= (<STL,A>)
<ST>        ::= GO TO <EXP,A>
<ST>        ::= <NAM,A> ':' <ST,B>
<ION>       ::= <NAM,A> ( )
<ION>       ::= <NAM,A> ( <ELIST,B> )
<ST>        ::= <NLIST,A> IS  <PLIST,B>
<ST>        ::= <NLIST,A> ARE <PLIST,B>
<NLIST>     ::= <NAM,A>
```

```
<NLIST>     ::= <NLIST,A> , <NAM,B>
<PLIST>     ::= <PROP,A>
<PLIST>     ::= <PLIST,A> , <PROP,B>
<PROP>      ::= <NAM,A>
<PROP>      ::= <EXP,A> LONG
<PROP>      ::= COMMON
<PROP>      ::= REAL
<PROP>      ::= INTEGER
<PROP>      ::= REGISTER
<PROP>      ::= RESERVED
<PROP>      ::= SCRATCH
<PROP>      ::= PROTECTED
<PROP>      ::= AVAILABLE
<PROP>      ::= RELEASED
<PROP>      ::= LOCAL
<SYN>       ::= LET
<SYN>       ::= <SYN,A> <NAM,B>=<VBL,C>,
<ST>        ::= <SYN,A> <NAM,B>=<VBL,C>
<ATOM>      ::= - <ATOM,A>
<ATOM>      ::= NOT <ATOM,A>
<EXP>       ::= <ATOM,I>   LS <EXP,J>
<EXP>       ::= <ATOM,A>  ALS <EXP,B>
<EXP>       ::= <ATOM,A> LROT <EXP,B>
<EXP>       ::= <ATOM,A>   RS <EXP,B>
<EXP>       ::= <ATOM,A>  ARS <EXP,B>
<EXP>       ::= <ATOM,A> RROT <EXP,B>
<EXP>       ::= <ATOM,Y> * <EXP,Z>
<EXP>       ::= <ATOM,A> + <EXP,B>
<EXP>       ::= <ATOM,X> - <EXP,Y>
<EXP>       ::= <ATOM,A> / <EXP,B>
<EXP>       ::= <A>//<B>
<EXP>       ::= <ATOM,X> AND <EXP,Y>
<EXP>       ::= <ATOM,A>  OR <EXP,B>
<EXP>       ::= <ATOM,A> XOR <EXP,B>
<EXP>       ::= <ATOM,A> EQV <EXP,B>
<EXP>       ::= <VBL,A> ← <EXP,B>
<EXP>       ::= <VBL,A> '<'= <EXP,B>
<VBL>       ::= <VBL,A> [<EXP,B>]
<VBL>       ::= [<EXP,A>]
<ST>        ::= SUBR <SUBP,A> IS <EXP,B>
<SUBP>      ::= <NAM,A> ( <NLIST,B> )
<SUBP>      ::= <NAM,A> ( )
<ST>        ::= RETURN <A>
<ST>        ::= GO TO ( <GOLST,A> ) <B>
<GOLST>     ::= <NAM,A>
<GOLST>     ::= <GOLST,A> , <NAM,B>
<ELIST>     ::= <EXP,A>
<ELIST>     ::= <ELIST,A>,<B>
<ST>        ::= DATA ( <ELIST,A> )
<ST>        ::= REMOTE <ST,A>
<ATOM>      ::= LOC ( <VBL,A> )
<RELOP>     ::= =
<RELOP>     ::= '<'
```

```
<RELOP>     ::= >
<RELOP>     ::= NE
<RELOP>     ::= LE
<RELOP>     ::= GE
<RELOP>     ::= EQ
<RELOP>     ::= LT
<RELOP>     ::= GT
<ST>        ::= <EXP,A> => <ST,B>
<ST>        ::= <EXP,A> <RELOP,EQ> <EXP,B>
<ST>        ::= <EXP,A> <RELOP,EQ> <EXP,B> => <ST,C>
<ST>        ::= <EXP,A> <RELOP,EQ> <EXP,B> => <ST,C> ELSE <ST,D>
<ST>        ::= <EXP,A> <RELOP,EQ> <EXP,C>
<ST>        ::= MOVE <B> THROUGH <N> TO <A>
<ST>        ::= <EXP,A> FOR <VBL,B> IN <EXP,C>,<EXP,D>,<EXP,E>
<ST>        ::= <EXP,A> FOR <VBL,B> TO <EXP,C>
<ST>        ::= <EXP,A> FOR <VBL,B> FROM <EXP,C>
<ST>        ::= WHILE <EXP,A> DO <ST,B>
<ST>        ::= WHILE <EXP,A><RELOP,EQ><EXP,B> DO <ST,C>
<ST>        ::= <EXP,A> UNTIL <B><RELOP,EQ><C>
<ST>        ::= <EXP,A> UNTIL <EXP,B>
<COND>      ::= (<A> <RELOP,EQ> <B>)
<COND>      ::= (<A> <RELOP,EQ> <B>) OR <COND,C>
<COND>      ::= (<A> <RELOP,EQ> <B>) AND <COND,C>
<ST>        ::= <COND,A> => <ST,B>
<ST>        ::= <COND,A> => <ST,B> ELSE <ST,C>
<ST>        ::= WHILE <COND,A> DO <ST,B>
<ST>        ::= <EXP,A> UNTIL <COND,B>
<IO>        ::= PRINT <PITEM,A>
<IO>        ::= READ  <PITEM,A>
<IO>        ::= <IO,A> , <PITEM,B>
<EXP>       ::= <IO,A>
<PITEM>     ::= <EXP,A>
<PITEM>     ::= OCT <A>
<PITEM>     ::= IGR <A>
<PITEM>     ::= STG <A>
<PITEM>     ::= FILE <NAM,A>
<PITEM>     ::= FILE <NAM,A>.<B>
<PITEM>     ::= FILE <NAM,A>[<C>,<D>]
<PITEM>     ::= FILE <NAM,A>.<B>[<C>,<D>]
<PITEM>     ::= /
<PITEM>     ::= DEVICE <A>
<PITEM>     ::= IMAGE MODE
<PITEM>     ::= TAB <N>
<PITEM>     ::= FILL <N>
<PITEM>     ::= FLT <A>.<B>
<VBL>       ::= <NAM,A>.<NAM,B>
<VBL>       ::= <NAM,A>.<NAM,B>"<ATOM,C>
<ATOM>      ::= FIX(<A>)
<ATOM>      ::= FLT(<A>)
<BYTE>      ::= <VBL,A> '<' <B> , <C> >
<ATOM>      ::= <ION,A> '<' <B> , <C> >
<EXP>       ::= <BYTE,A> ← <B>
<ATOM>      ::= BYTEP <BYTE,A>
```

```
<BYTE>      ::= '<' <EXP,A> >
<ATOM>      ::= '<' <EXP,A> >
<ATOM>      ::= '<' + <VBL,A> >
<EXP>       ::= '<' + <VBL,A> > ← <B>
<ATOM>      ::= <ION,A> '<' R >
<ATOM>      ::= <ION,A> '<' L >
<EXP>       ::= <VBL,A> '<' R > ← <B>
<EXP>       ::= <VBL,A> '<' L > ← <B>
<ST>        ::= EXECUTE <A>
<ST>        ::= CALL ME <NAM,A>
<ST>        ::= TWOSEG
<ATOM>      ::= IDPB(<A>,<B>)
<ATOM>      ::= ILDB(<A>)
<ATOM>      ::= R<VBL,A>
<ION>       ::= CALLI(<A>,<B>)
<ATOM>      ::= XWD <A>,<B>
<ATOM>      ::= IOWD <A>,<B>
```

References

Bilofsky 1972: IMP Reference Manual, Working Paper No. 349, Institute for Defense Analyses, Princeton, N.J.

Bilofsky 1973: Syntax Extension and the IMP Programming Language, unpublished.

Emerson 1849: Representative Men.

Irons 1963: An Error-Correcting Parse Algorithm, Comm. of the ACM, Vol. 6 No. 12, Dec. 1963

Irons 1970: Experience with an Extensible Language, Comm. of the ACM, Vol. 13 No. 1, Jan. 1970

Irons 1971: Syntax Graphs and Fast Context Free Parsing, Research Report No. 71-1, Department of Computer Science, Yale University.

Naur 1963: Revised Report on the Algorithmic Language ALGOL 60. Comm. of the ACM, Vol. 6 No. 1, Jan. 1963. Section 1.1 defines BNF.

Shakespeare 1612: The Tempest

Webster's 1967: Webster's 7th New Collegiate Dictionary, G&C Merriam Co., Pub., 1967.

Weingart 1973: An Efficient and Systematic Method of Code Generation, Ph.D. Thesis, Department of Computer Science, Yale University, June 1973.

INDEX

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) (Revised October 25, 1971)
SOURCE LANGUAGE	BLISS

* Digital Equipment Corporation, Maynard, Mass. 01754

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)
(Revised October 25, 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 mname (parameters) = e ELUDOM

mname → | name

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 'mname' 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. If 'mname' is empty the first four characters of the file name of the object fiel (.REL file) will be used; see II-1.4. 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 |
                              e SEMICOLON 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.

The reserved word SEMICOLON is identical to the semicolon character except that it also serves as a compiler directive with respect to code optimization (see III-1.1.5).

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 | string | 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!

string → stringtype quoted string

stringtype → | ASCII | ASCIZ | SIXBIT | RADIX50

quotedstring → leftadjustedstring | rightadjustedstring

leftadjustedstring → 'string'

rightadjustedstring → "string"

Quoted-string literals may be used to specify bit patterns corresponding to the ASCII, SIXBIT, or RADIX50 codes used on the PDP-10; left or right

justification may be obtained through the use of the single or double quote characters. An empty 'stringtype' implies an ASCII string. Normally quoted strings are constrained to be representable within a single word (five characters for ASCII, six characters for SIXBIT and RADIX50), but strings of arbitrary length may be used in PLIT's (see 1.3.1).

Within a quoted string the quoting character is represented by two successive occurrences of that character. Also, in a quoted string the question mark, "?", is an escape-to-control character -- thus "?M" represents a 'control-M', or carriage return. In addition

?? represents question mark itself

?0 represents the NULL (zero) character

?1 represents the DEL (all ones) 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.

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

p ← e

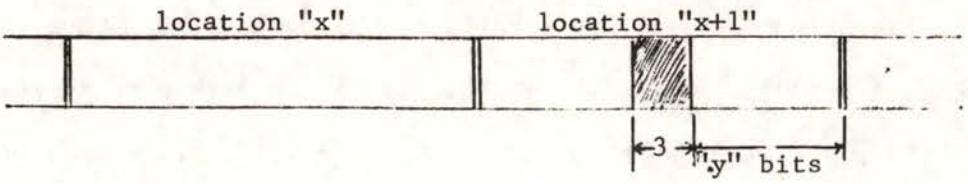
This means "store the value computed from e into the location whose pointer is the value of p". (Further details are given in 2.2.) Thus the Algol statement "A := B" is written "A ← .B". It is impossible to express in Algol BLISS expressions such as: "A ← B", "A ← ..B", ".A ← .B", etc.

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	.ϵ	Value (possibly partial word) pointed at by ϵ.
3	@ϵ	Equivalent to .ϵ<0.36.0.0>.
3	\ϵ	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	-ϵ	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	$\epsilon_1 \leftarrow \epsilon_2$	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.

Examples:

```
if .x then J ← .K else J ← .L;
J ← if .x then .K else .L; !same effect as previous line
if .x then K ← .L else J ← .L;
(if .x then K else J) ← .L; '.same effect as previous line
position ← .position + (if .char eql #11 %tab% then 8 else .T);
```

* 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.

Examples:

```
! find last item of simple list
link ← .beginningoflinkedlist;
while ..link neq ø do link ← ..link;
! link contains address of last item

! add up first N numbers
sum ← ø;
incr j from 1 to .n do sum ← .sum + j;
```

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 → ⌈ [é]

escapevalue → | e

Each of these expressions conveys to its new environment a value, say ∈, obtained by evaluating the escapevalue, which may optionally be omitted implying ∈ = 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 ∈.
EXITBLOCK	terminates the innermost n (where n is the value of the "levels" field) blocks, yielding a value of ∈ for the outermost one exited.
EXITCOMPOUND	terminates the innermost n compound expressions, yielding a value of ∈ for the outermost one exited.
EXITLOOP	terminates the innermost n loop expressions, yielding a value of ∈ for the outermost one exited.
EXITCOND	terminates the innermost n conditional expressions, yielding a value of ∈ for the outermost one exited.
EXIT	terminates the innermost n control scopes (whether blocks, compounds, conditionals, or loops with ∈ as the value of the outermost.

EXITCASE terminates the n innermost case expressions yielding a value of ∈ for the outermost of these.

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

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

Examples:

```
! find index of first space in line image of 80 characters (one per word)
! index = -1 implies none found

index ← incr j from 0 to 79 do
                if .(line + .j) eql #40 then exitloop j;

! to exit the body of a loop and go on to the next iteration
incr j from 1 to 100 do
        ( ...
        ...
        if .condition then exitcompound; !go to next iteration
        ...
        )                                !close compound
        ;                                !close body expression
```

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 $\Pi_0; \Pi_1; \ldots; \Pi_k$. Then the expressions $\Pi_{\epsilon_1}, \Pi_{\epsilon_2}, .., \Pi_{\epsilon_m}$ 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 → xe:e

xe → e | ALWAYS | OTHERWISE

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.

Example:

```
! Suppose we have a teletype input routine

  select .char of nset

      #15: ...          ;          !cr ⇒ echo lf

      #11: ...          ;          !tab ⇒ echo spaces

                       tesn;
```

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 mean 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
sizedchunk → 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 onto a specified structure these identifiers. (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).

Examples:

1. local A,B,C;

   Allocates 3 locals on the stack and binds the names A,B, and C to them dynamically.

2. own A:B:C[3];

   Allocates 9 words, binds the names A,B, and C to the first, fourth and seventh word, respectively and maps the default "VECTOR" structure onto them.

3. structure BITVECTOR[I] = [I↑(-5)+1]...;
   global BITVECTOR A:B[300]:C[200],X[20];

   Allocates 4 chunks of global storage with sizes (300/32+1), (300/32+1), (200/32+1) and 20, respectively, binds the names A,B,C and X to them, and maps BITVECTOR onto A,B and C and the default VECTOR structure onto X, making the names A,B,C and X globally available to other contemporaneously loaded modules.

In addition to the general syntax described above, for REGISTER declarations only, the following is permitted:

register specificreglist

where

specificreglist → specificreg | specificreglist,specificreg

specificreg → structure name size = e

size → | [elist]

The expression, e, must evaluate to a constant at compile time such that $0 \leq e \leq 15$ and the e^{th} register has been reserved in the module head (see section III-1.1.4). Thus, for example, the following declaration is legal:

register r=5, rx=10;

and will cause the specific registers 5 and 10 to be given names 'r' and 'rx', respectively. This mechanism may, for example, be used for global register communication between modules.

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.

Examples:

1. map A:B[10];

   Maps the default structure VECTOR onto A and B, associating 10 with the first incarnation formal.

2. map A;

   Maps the default structure VECTOR onto A, associating 1 with the first incarnation formal.

3. map SOMESTRUCT A:B:C,

   ANOTHERSTRUCT X[3]:Y:Z[5,4]

   SOMESTRUCT is mapped onto A,B and C, with default incarnation actuals of 1; ANOTHERSTRUCT is mapped onto X, Y and Z. Incarnation actuals 3 and (default) 1 are associated with X; 5 and 4 with both Y and Z.

NOTE: In the above, A,B,C,X and Y must have been declared previously.

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 declaration is processed during block entry. Note that these expressions need not evaluate at compile time.

Example:

```
bind  ONE=1,CTWO=.2<0,36>, NAME=ALOCAL,
      Y[5]=QQ[20], R:L:M=7;
```

Future references to:

"ONE" will be equivalent to using "1";

"R", "L", and "M" to using 7;

"NAME" to using "ALOCAL";

"CTWO" to the contents of register 2 at the time the bind is executed;

"Y[∈]" to "QQ[20+∈]" (assuming QQ was also mapped with the vector structure).

Note, the Y[5] only indicates that the default vector structure should be mapped with incarnation actual of 5 - not that of (Y+5)≡ QQ[20].

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)<0,36>;
```

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. Caution: "(" is not a balanced string; there is no matching close parentheses.

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 More Special Functions

(1) $\text{SIGN}(3) \equiv \begin{cases} -1 & \text{if } e < 0 \\ 0 & \text{if } e = 0 \\ +1 & \text{if } e > 0 \end{cases}$

(2) $\text{ABS}(e) \equiv \begin{cases} e & \text{if } e \geqslant 0 \\ -e & \text{if } e < 0 \end{cases}$

(3) $\text{FIRSTONE}(e) \equiv \begin{cases} -1 & \text{if } e = 0 \\ \text{number of zero-bits to the left of the first one-bit in the value of } e & \text{otherwise} \end{cases}$

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
```

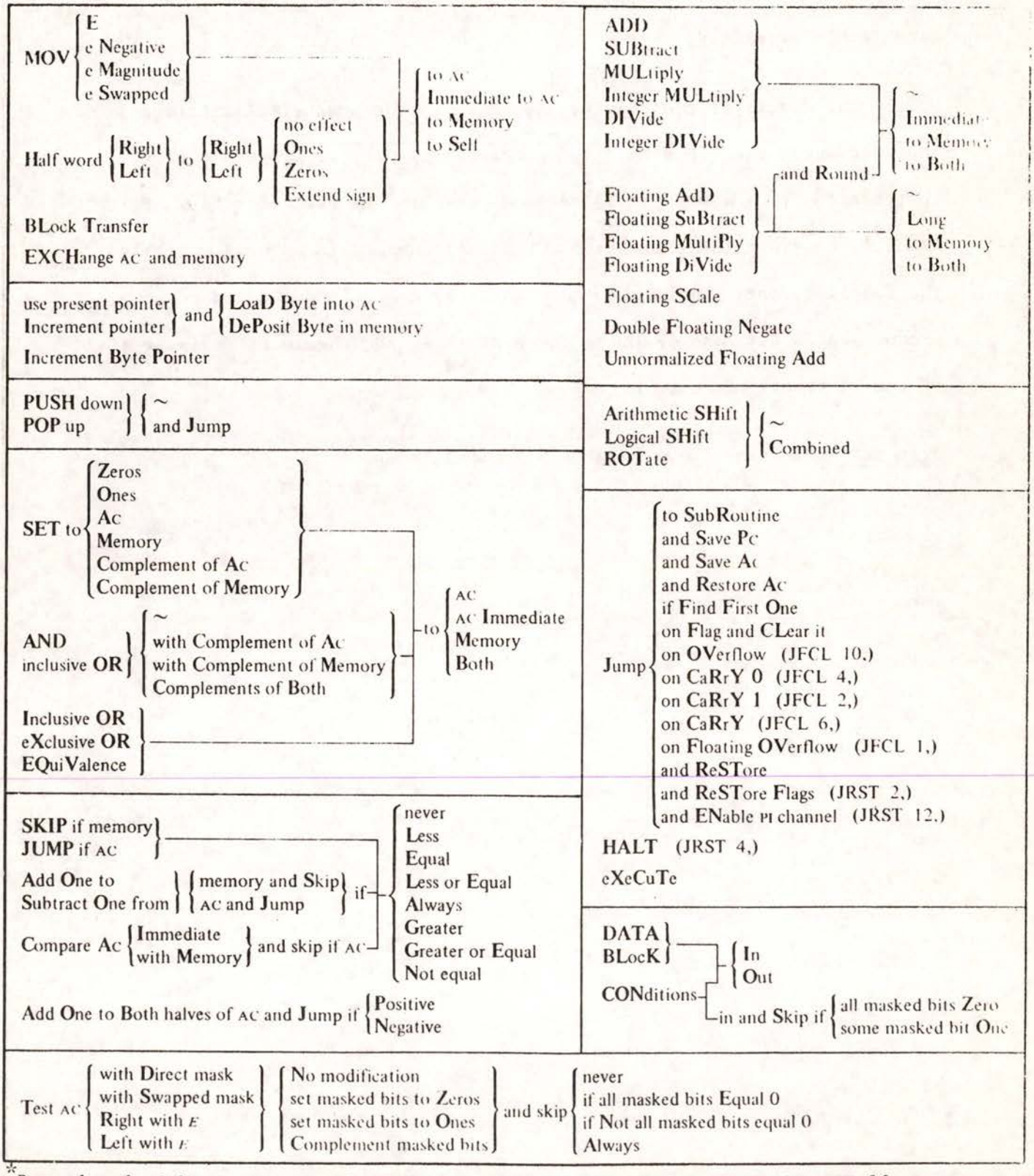
in the head of a block which embraces occurrences of these special functions.

(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.

ALSO NOTE: ϵ_4 must be a constant at compile time.

PDP-10 Instruction Mnemonic Table*



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

1.1 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.

1.1.1 Command Syntax

1.1.1.1 Normal use:

The general format of the command string 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

Switches may be included in the command string as either /x or /-x (where xG{A,B,...,Z}). A switch of the form /-x has the opposite effect of a switch of the form /x (for the same x). A switch may be included anywhere; however, some switches particularly affect the file with which they appear (and usually all those to its right).

Project-programmer numbers (PPN's) may be specified in one of two ways: if a PPN appears to the left of a file name, it applies to that file and all files to its right (unless changed by having a different PPN appear to the left of another file later); if it appears to the right of a file name, it applies only to that file. A PPN to the right of a file name always applies to that file, even if a PPN has appeared to the left of an earlier file name.

1.1.1.2 Use from CCL:

If the COMPIL CUSP has been modified to allow BLISS programs to be processed, then the COMPILE, EXECUTE, LOAD, and DEBUG commands may be used. A few words of warning, however, are necessary.

(1) The BLISS cross-reference (/C switch in the command string) is not invoked by the /CREF switch. /CREF assumes a file with the extension .CRF will be written for later processing by the CREF cusp. BLISS produces its cross-reference listing internally, If /CREF is used, confusion and madness will ensue.

(2) The name of the REL file in the "+" construction is the name of the last file given, i.e., "A+B+C" generates files "C.LST" and "C.REL". Some BLISS-coded systems use a declaration file which forms an outer block, and an "end" file to close the outer block, with the desired programs sandwiched between, e.g.,

"OVBEGIN+H1DECL+END"

The rel and lst files will be END.REL and END.LST. To obtain the proper results, use the "=" construction, i.e.,

"H1DECL=OVBEGIN+H1DECL+END"

1.1.2 Module Head

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

module → MODULE mname(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 III.1.1.4 under the column headed "module head syntax".

1.1.3 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".

1.1.4 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,/-L	NOLIST	NOLIST	Disable listing of the source text.
/N	NOERS	NOERS	Do not print error messages on the TTY.
/-N	-	-	Re-enable error messages on TTY.
/M	MLIST	MLIST	Enable listing of the machine code generated.
/-M	NOMLIST	NOMLIST	Suppress listing of generated machine code.
/H	HISEG	-	Make this module a highsegment module. Initally 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.
/-I	NOINSPECT	NOINSPECT	This sets the inspection switch false.
/S	-	-	Causes the name of each routine to be printed on TTY as it is compiled (unless /N in effect).
/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. '/-X' is illegal.

COMMAND SWITCH	MODULE HEAD SYNTAX	'SWITCHES' DECLARATION	ACTION
-	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).
-	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 or /-O	NOOPTIMIZE	NOOPTIMIZE	Sets the optimization switch (see above) to false.
/E	EXPAND	EXPAND	Give trace of macro expansions.
/-E	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).

COMMAND SWITCH	MODULE HEAD SYNTAX	'SWITCHES' DECLARATION	ACTION
/R /-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. '/-V' is illegal.
-	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 deal → STACK|STACK|STACK=explicit-stack where explicit-stack → stype s-name-sz stype → GLOBAL|OWN|EXTERNAL s-name-sz → (ss-OPTN) ss-OPTN → |e The defaults are 'STACK'= STACK=OWN(STACK,#1000) 'STACK(e)' = STACK-OWN(STACK,e) etc.
/G	GLOROUTINES	GLOROUTINES	All routine names are forced to be 'global'.
/-G	NOGLOROUTINES	NOGLOROUTINES	Non-global routines are <u>not</u> 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.
-	TIMER	-	The syntax of the module head permits automatic inclusions of code to facilitate tracing and timing.

COMMAND SWITCH	MODULE HEAD SYNTAX	'SWITCHES' DECLARATION	ACTION
			timerdecl → TIMER|TIMER(e)| TIMER=explicit timer explicittimer → trtype tname-size trtype → FORWARD|EXTERNAL tname-size → (name size-optn) size-optn →|e The defaults are 'TIMER ≡ TIMER = EXTERNAL (TIMER,4) 'TIMER(e)' ≡ TIMER = EXTERNAL (TIMER,e) etc.
/T	TIMING	-	If the TIMER declaration is given in the module head, routine timing linkages will be generated.
/-T	NOTIMING	-	Turns off generation of timing linkages
-	-	NOTIMING	Suppresses generation of timing linkages
-	-	TIMING	If timing linkages were being generated and have been suppressed by the NOTIMING switch in the SWITCHES declaration, this re-enables generation. Otherwise it has no effect.
-	CCL	-	Generates a CCL-compatible entry linkage as the first two instructions of the main program. A STACK declaration must also be present to enable generation of these instructions.

1.1.5 SEMICOLON Delimiter

The reserved identifier SEMICOLON is identical to the character ";" syntactically. In addition, it is a directive to the compiler declaring that the expression just completed may have side-effects which are unpredictable by the compiler. Consequently no assumptions should be about valid temporary or intermediate results for optimization purposes.

2.1 Unenforced restrictions

There are certain language restrictions that are not (some cannot be) enforced by the BLISS compiler. Let the user take note!

1. Bliss itself uses only the first ten characters of identifiers - distinct identifiers with the same initial ten characters will not be distinguished.

   Further, global symbols processed by the PDP-10 system loader are limited to six characters - so that symbols distinguished by Bliss may not be distinguished at load time.

   In particular, the module name is used to construct certain GLOBAL symbols as explained in section IV-1.3 and hence should be unique from other global symbols in their first four characters.

2. The Bliss compiler distinguishes between two classes of temporary registers - savable (used for declared registers, etc.) and non-savable. The number of savable registers is determined by the DREGS command and its default declaration is DREGS=5. All Bliss modules to be combined by the loader into a load-module must be compiled with identical DREGS declarations - otherwise run-time routine linkage may not work correctly.

3. Care must be exercised in invoking a FUNCTION in an improper enviromment. This might occur if a FUNCTION name is passed as a parameter to a ROUTINE which then executes the FUNCTION. Displays required by the FUNCTION execution will not be accessible.

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.

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

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

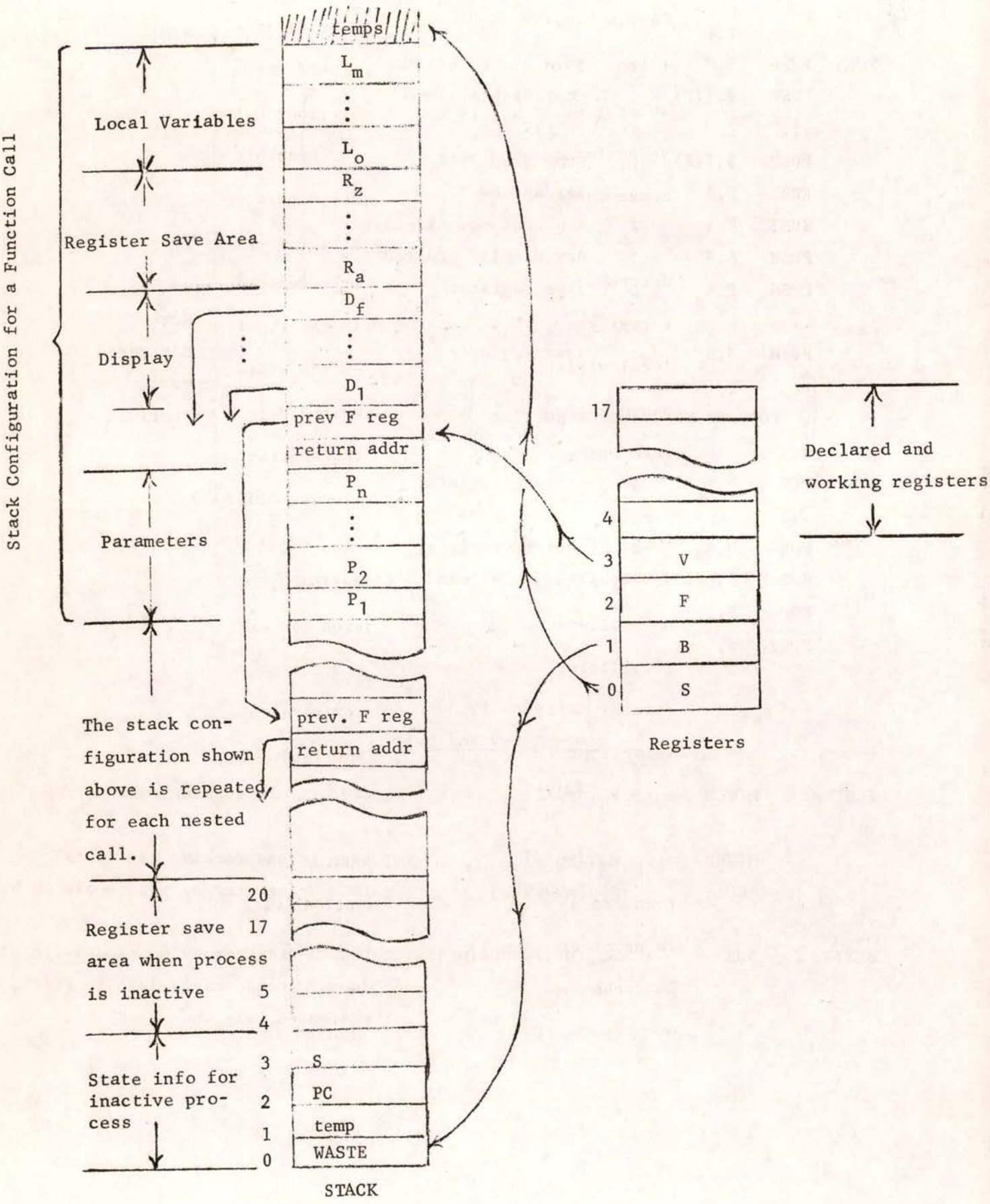


Figure IV.2

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 reg!.s 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.F is the location of the first instruction in the main body of the module, i.e., it is the starting address of the module

XXXX.L 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 DDT and corresponds to the first storage word of the module.

XXXX.P is the first location of the "plit" area.

Note that with (normal) two-segment conventions, XXXX.F, XXXX.L, XXXX.., and XXXX.P will be high-segment addresses while XXXX.O and XXXX.G will be low-segment addresses.

1.4 Main program code

1.4.1 CCL entry linkage

If the "CCL" switch is declared in the module head, the following two instructions are generated:

```
TDZA  $V,$V
MOVEI $V,1
```

Linkage via a RUN UUO with an offset of 1 causes entry to the second instruction of the program (i.e., the MOVEI) whereas normal entry (via a RUN, R, START command) is to the first instruction. The first executable statement in the program must be of the form "X ← .VREG" [assuming the user has not declared the name VREG, and it retains the meaning of the value register], which stores the value set by one of the two instructions. A "0" (false) indicates normal entry while a "1" (true) indicates a CCL-type entry.

1.4.2 Stack initialization

If a STACK declaration occurred in the module head, the following code is generated to initialize the F, S, and B registers. This code follows the CCL entry code, if any (see 1.4.1).

```
HRRZI   $B, stack-address
HRLI    $L, [stack-length]-[coroutine-prefix-length]
HRRZI   $F, [coroutine-prefix-length]
HRR     $S, $F
```

1.4.3 Program termination

All modules terminate with the instruction

```
CALLI  0,#12
```

which is the EXIT UUO for the PDP-10 monitors. This is the standard terminating execution of user programs.

1.5 Timer code

If there is a TIMER declaration in the module head, and it has been activated by the /T switch in the command string or the TIMING switch in the module head, the routine calls below are generated. The entry routine-call precedes the standard prolog code (see figure IV.2), and the exit routine-call precedes the standard epilog code.

entry call: TIMER-routine (routine-descriptor<0,0>)

exit call: TIMER-routine((-1↑18) or routine-descriptor<0,0>)

The actual code generated is:

```
entry call:  HRRZI  R, routine-descriptor
             PUSH   $S, R
             PUSHJ  $S, timer-routine
             SUB    $S, [000001,,000001]

exit call:   PUSH   $S, $V
             HRROI  R, routine-descriptor
             PUSH   $S, R
             PUSHJ  $S, [000001,,000001]
             POP    $S, $V
```

Note that the value of the V-register is preserved across the call. Hence any value returned by the timer-routine is lost.

As of this writing, no routine-descriptor is produced, and the address of the routine-descriptor is always 0.

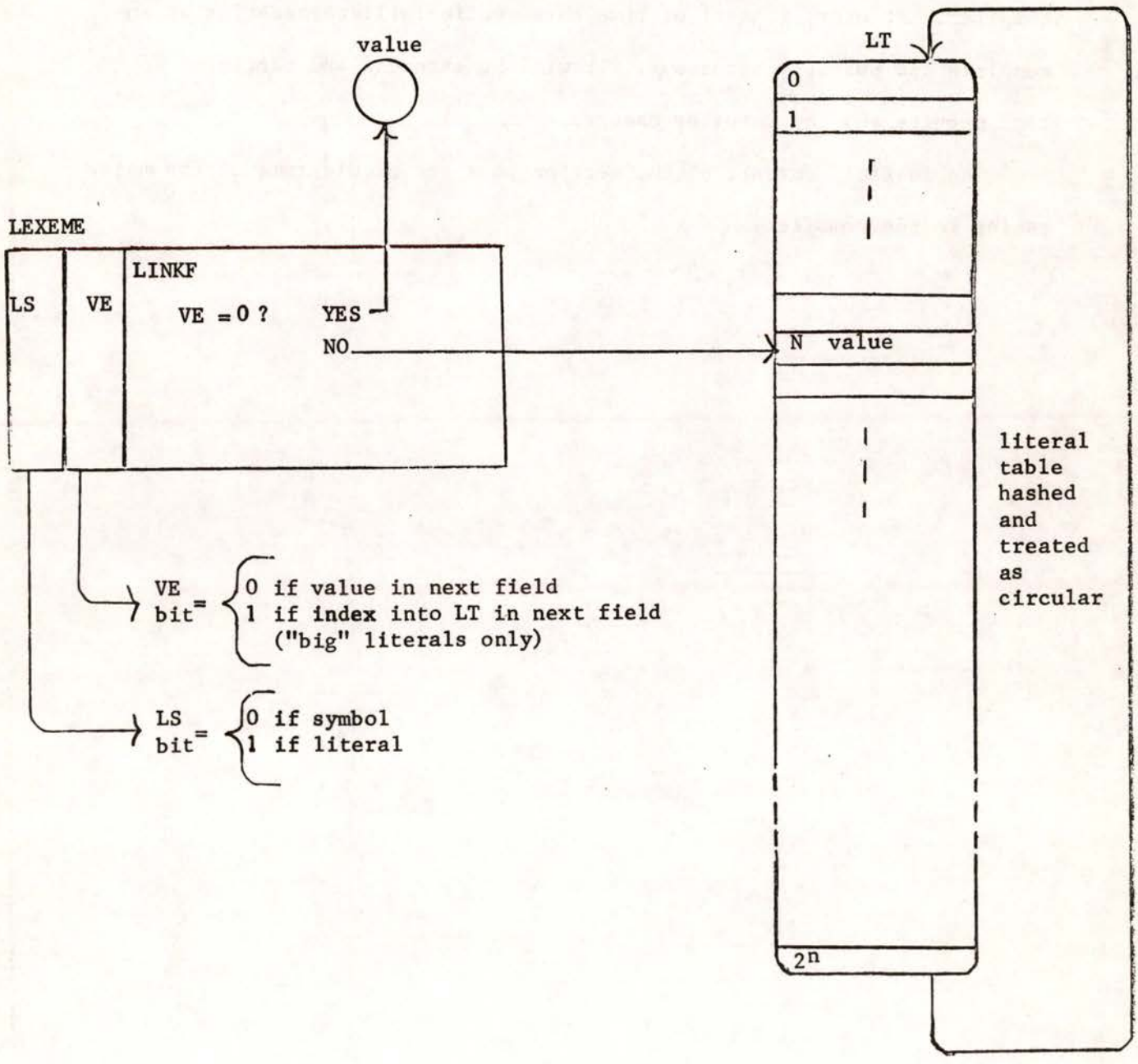
If the module contains a STACK declaration, then additional code is generated in the main body of the program. An entry call is generated

following stack initialization, and an exit call is generated immediately preceding the CALLI which terminates the program. If the CCL switch occurs in the module head, additional code is generated to save the V-register across the entry call.

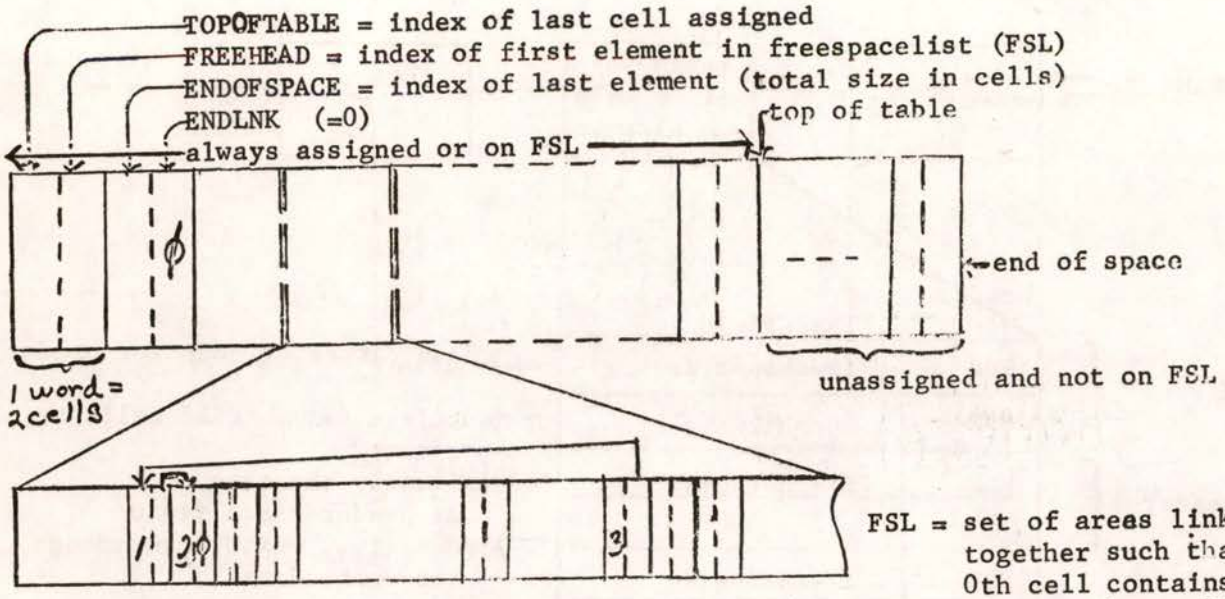
V. COMPILER IMPLEMENTATION

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.

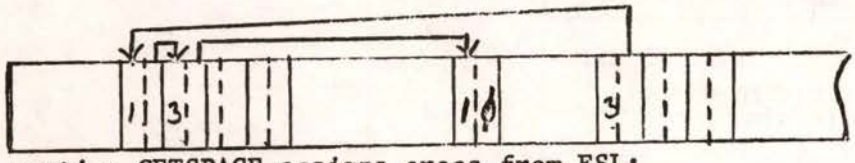


THE LITERAL TABLE

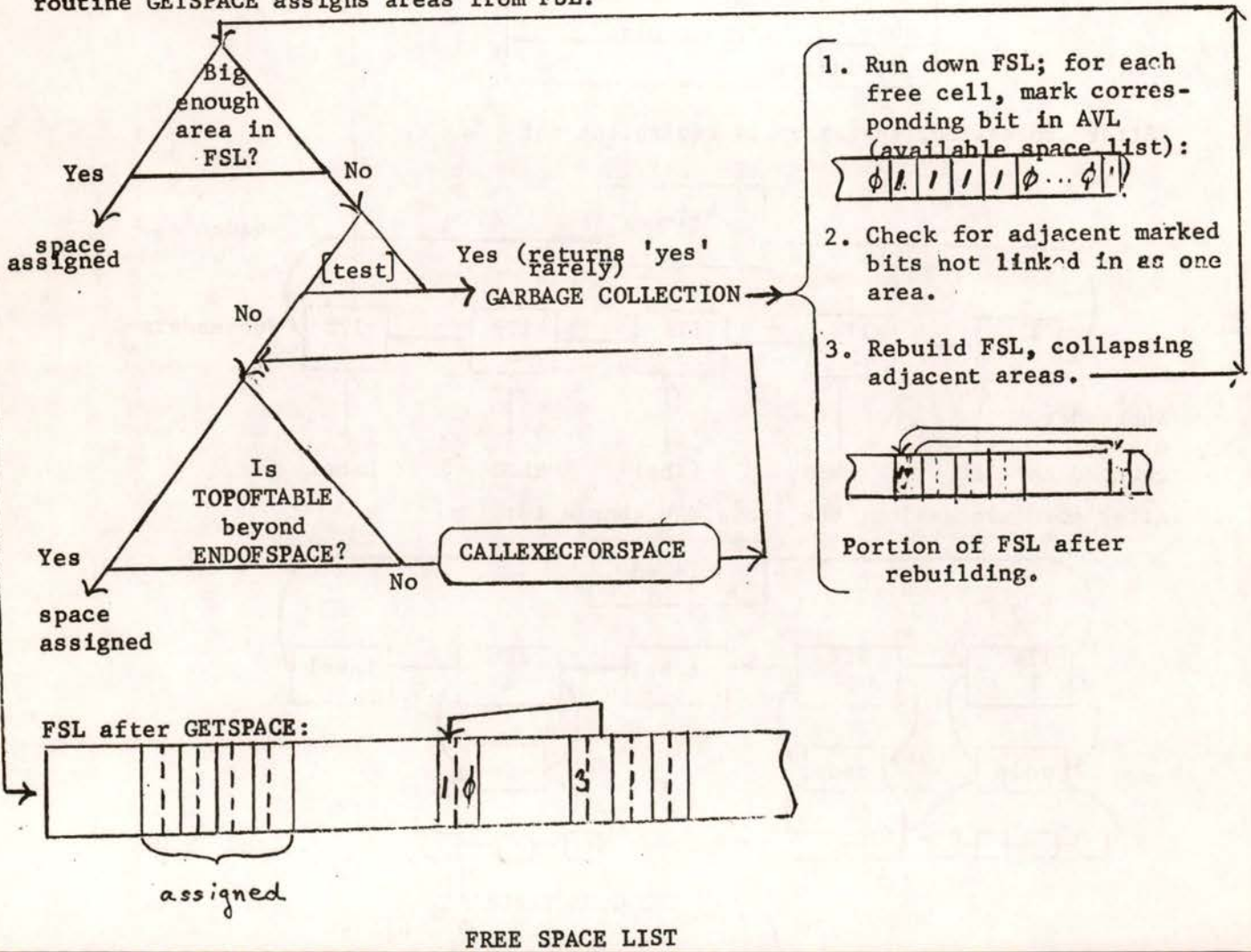


FSL = set of areas linked together such that 0th cell contains size of area, 1st cell has link to next area. (End of chain has 0 link.)

routine RELEASESPACE links areas into FSL:



routine GETSPACE assigns areas from FSL:

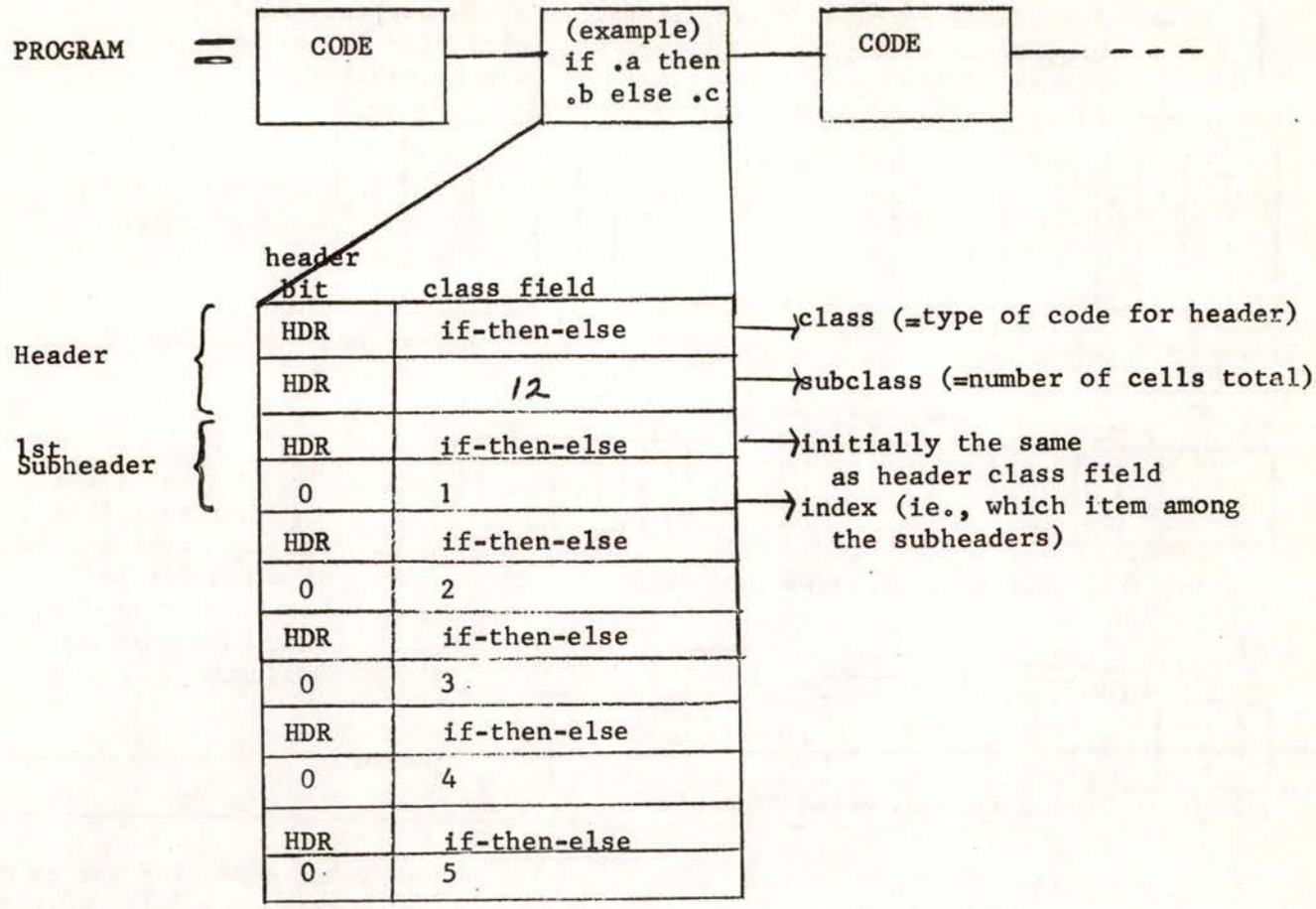


FSL after GETSPACE:

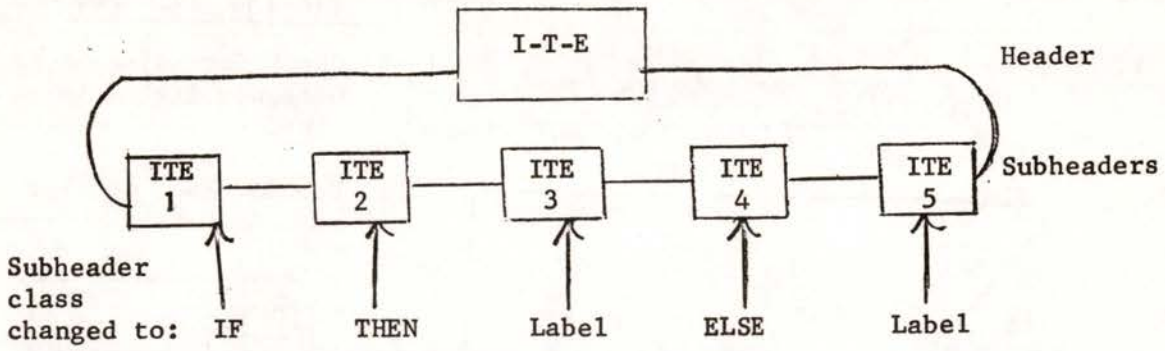
assigned

FREE SPACE LIST

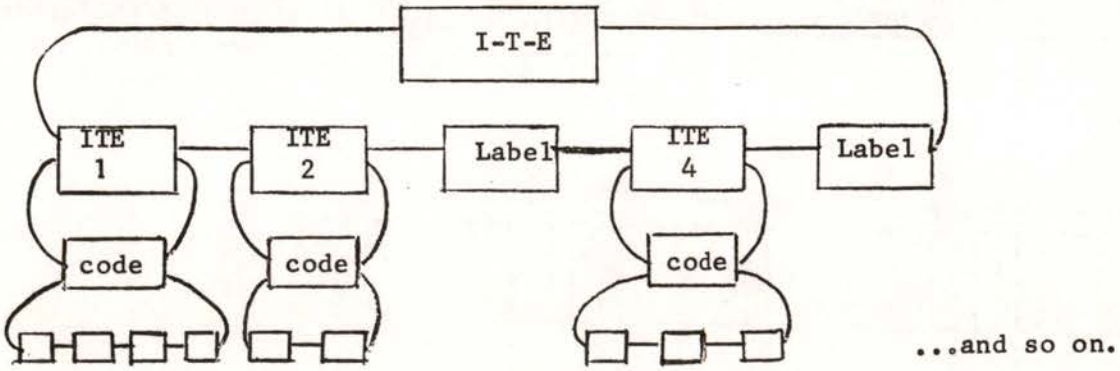
The entire program is a linked list, and is itself linked to global variable PROGRAM:



After processing, the above is equivalent to:



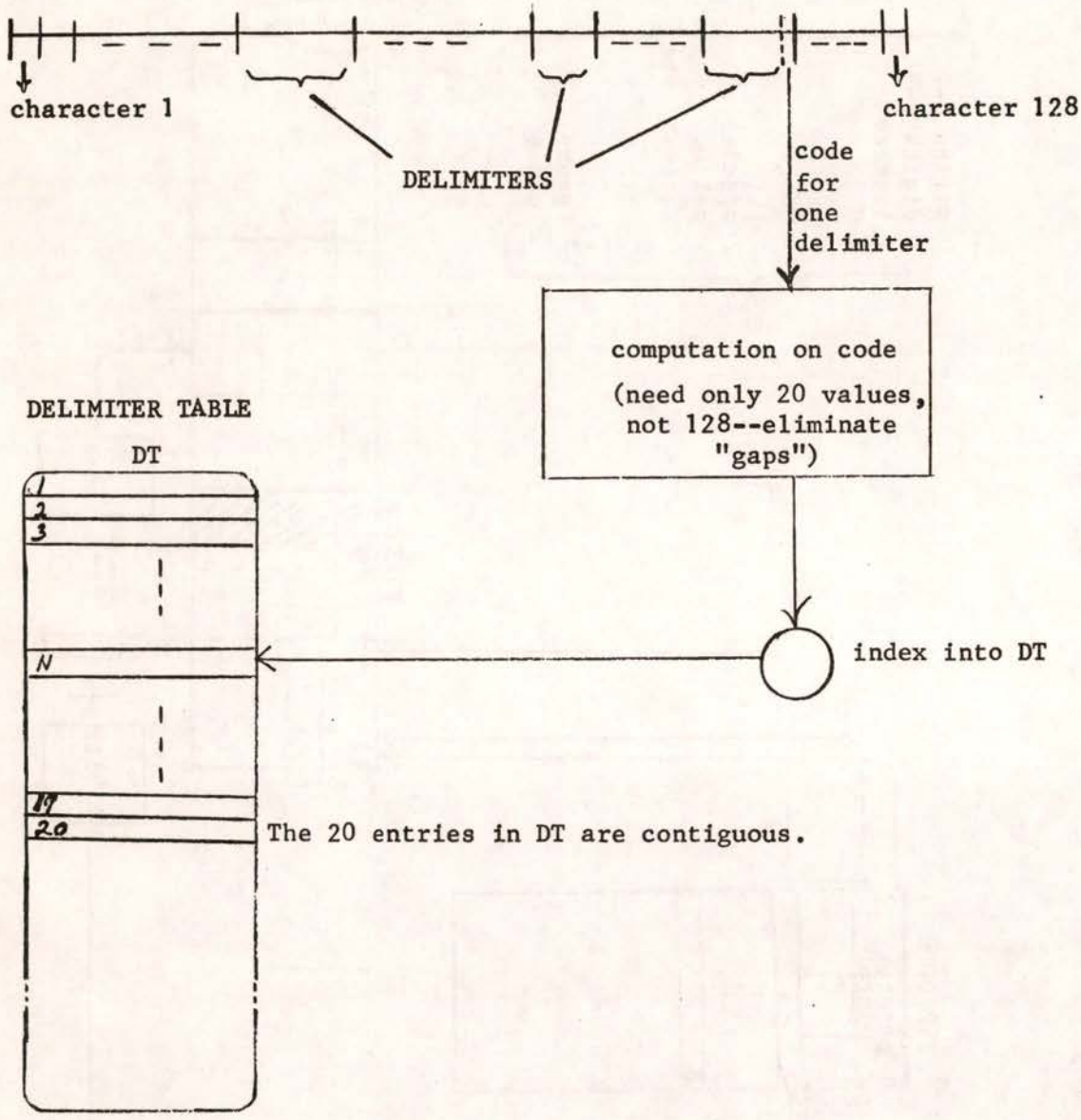
After more processing, the above may change to:



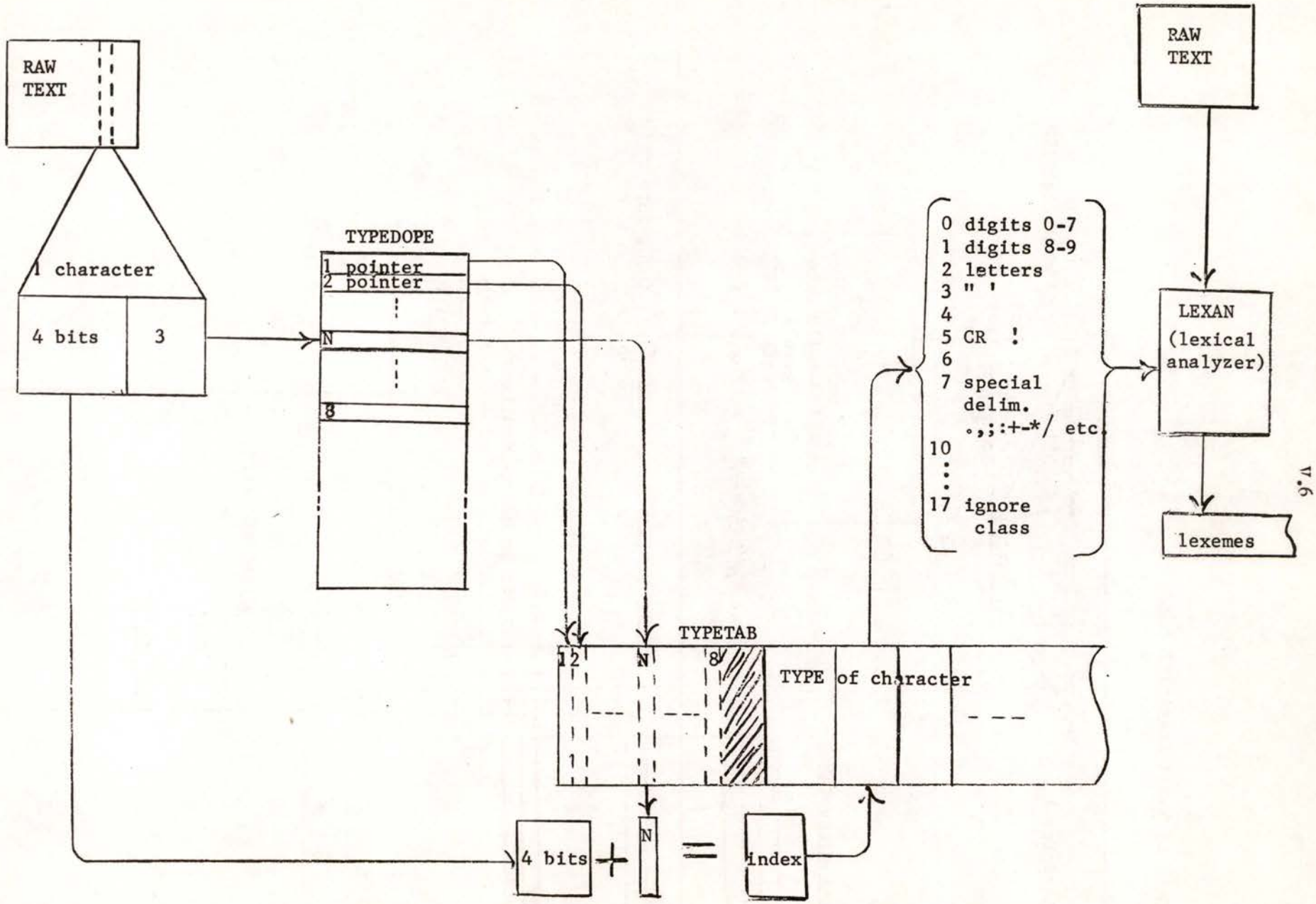
...and so on.

THE CODE TABLE

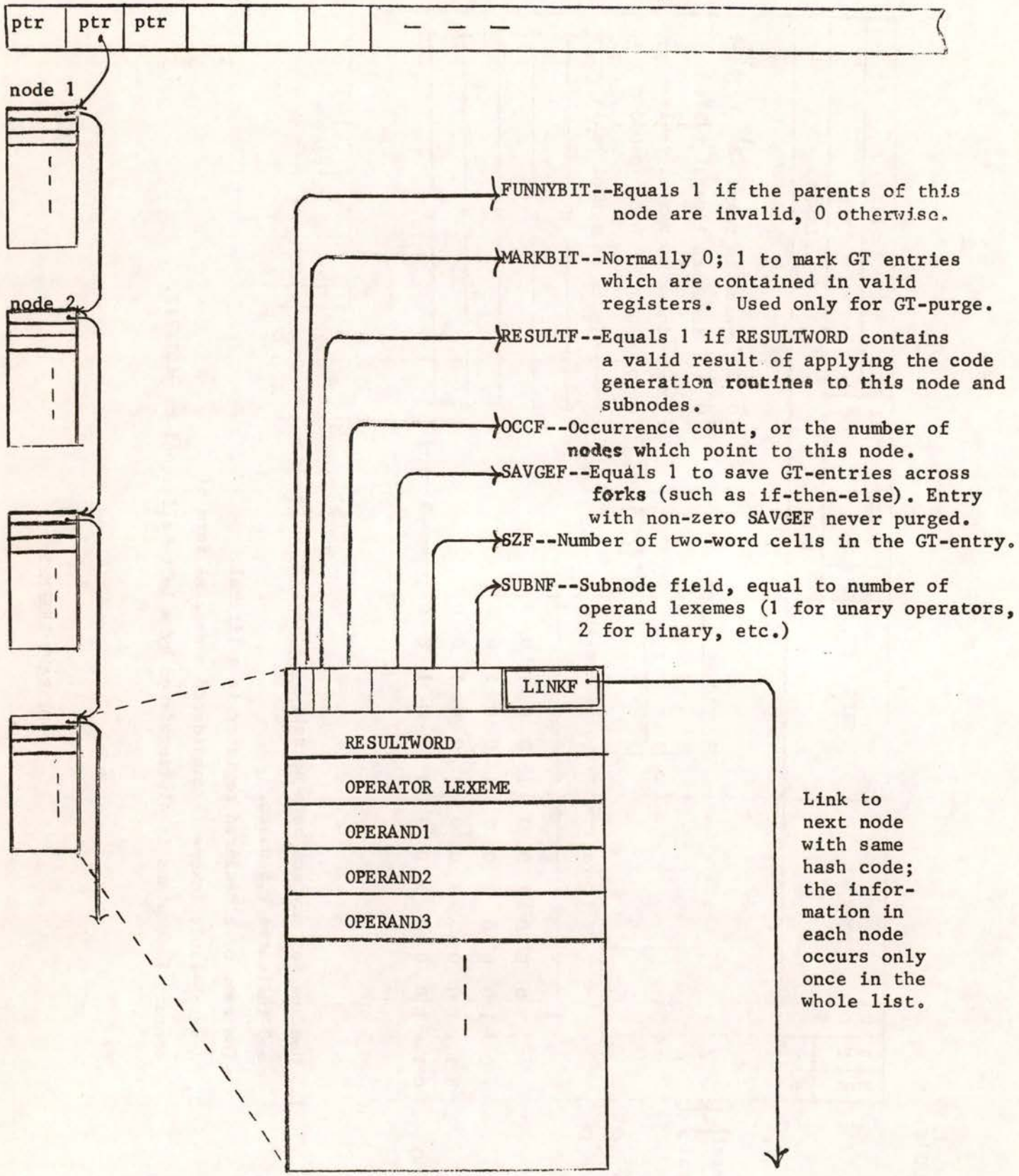
SCALE OF 7-BIT CHARACTER CODES



DELIMITER TABLE

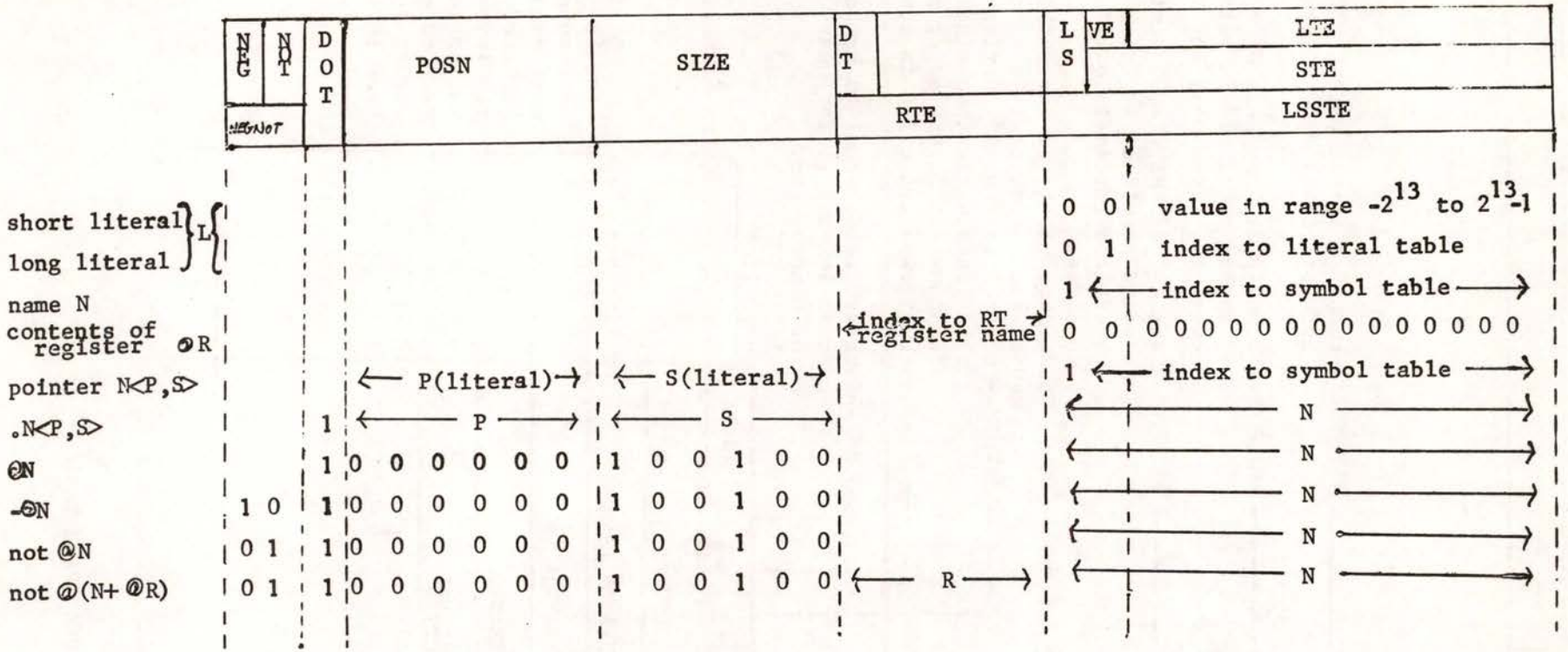


THE TYPE TABLE



THE GRAPH TABLE

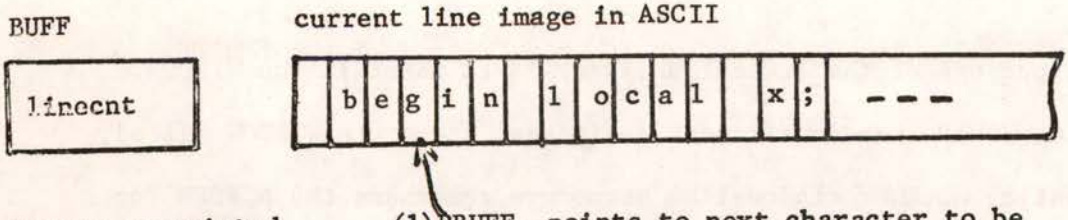
OPERAND LEXEME



NOTES:

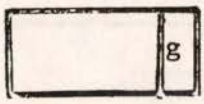
1. Neg and Not may not both be set.
   S=0 indicates P,S unset.
   The name of a declared register is a literal.
2. A graph-table lexeme is considered a special form of operand lexeme and is distinguished by a left-half equal to #077777.

OPERAND LEXEMES

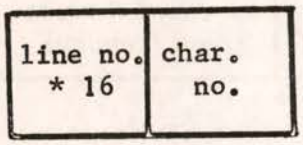


Values associated with BUFF are:

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



(2) CHAR, contains the character to be scanned



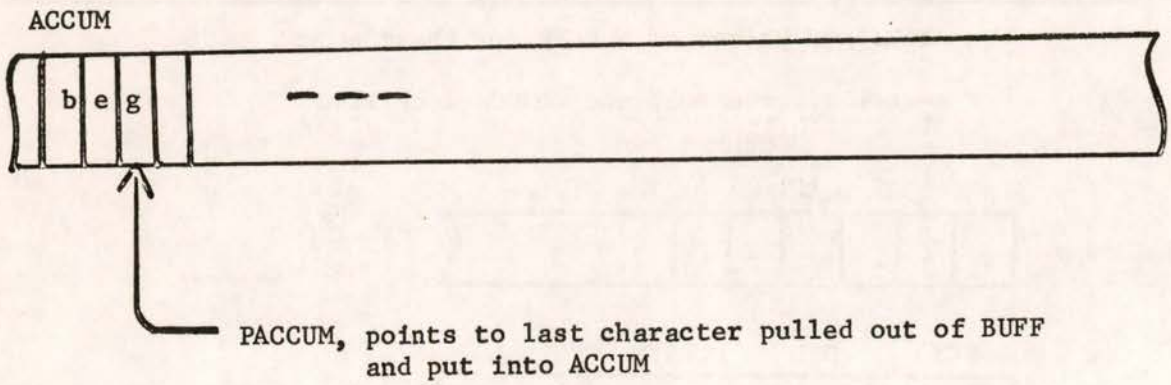
(3) NCBUFF, used for printing error messages-- points to place in line where error was detected

(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)

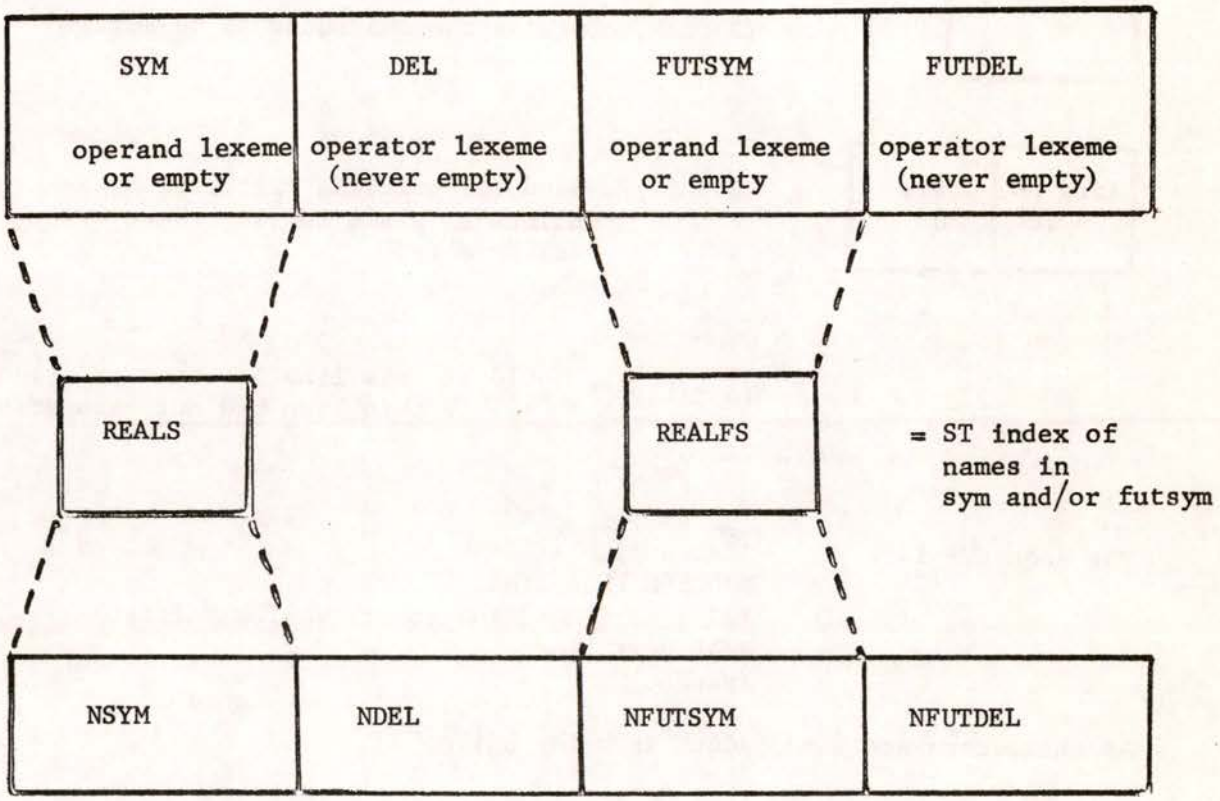
As characters are read, ACCUM is being built:



BUFF and ACCUM

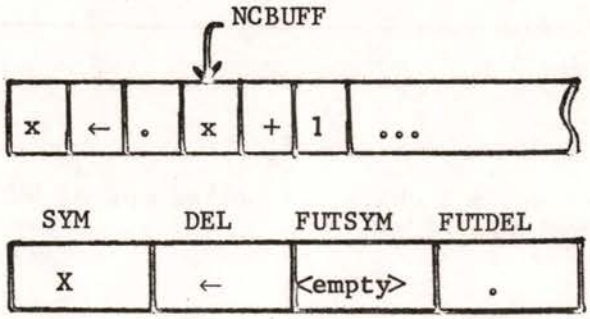
The major purpose of the lexical analyzer is to maintain the WINDOW; routine WRUND (read until next delimiter) keeps the WINDOW filled. Meanwhile, another window-like structure remembers the NCBUFF for lexemes in the WINDOW, for use when errors are detected.

WINDOW

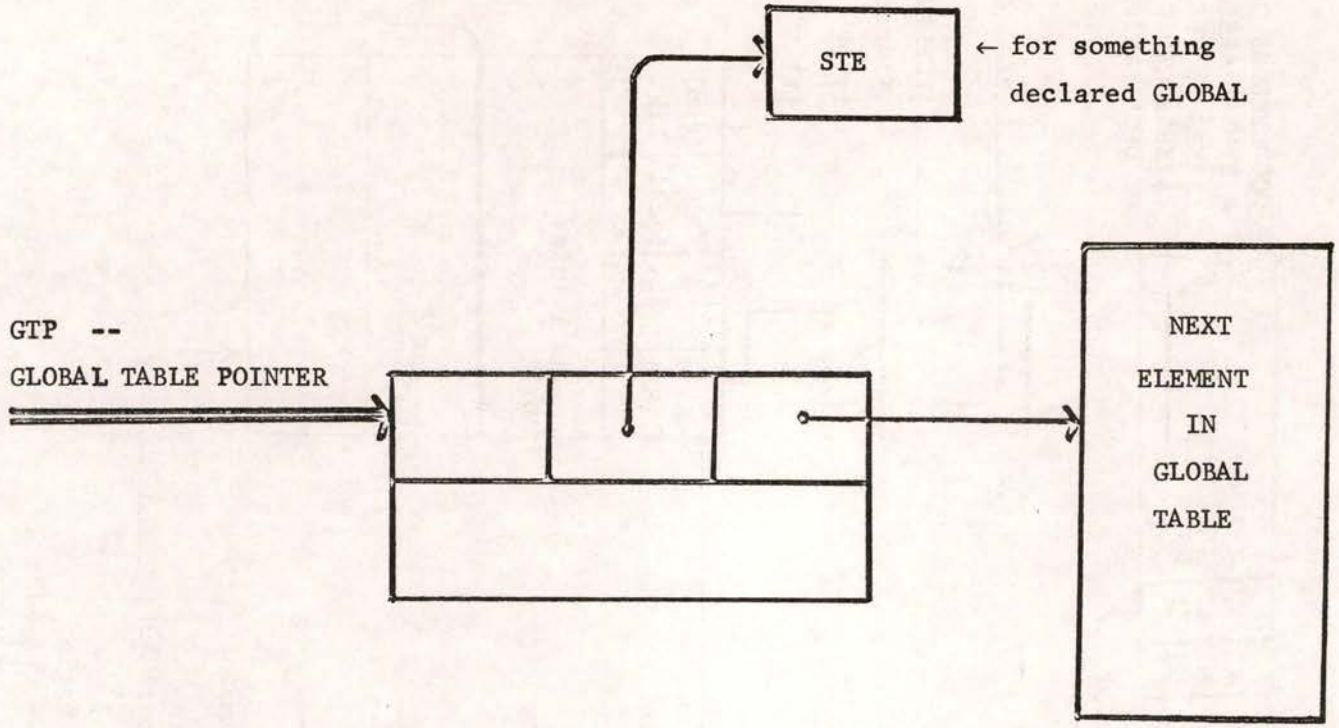


The above are values of NCBUFF for these atoms.

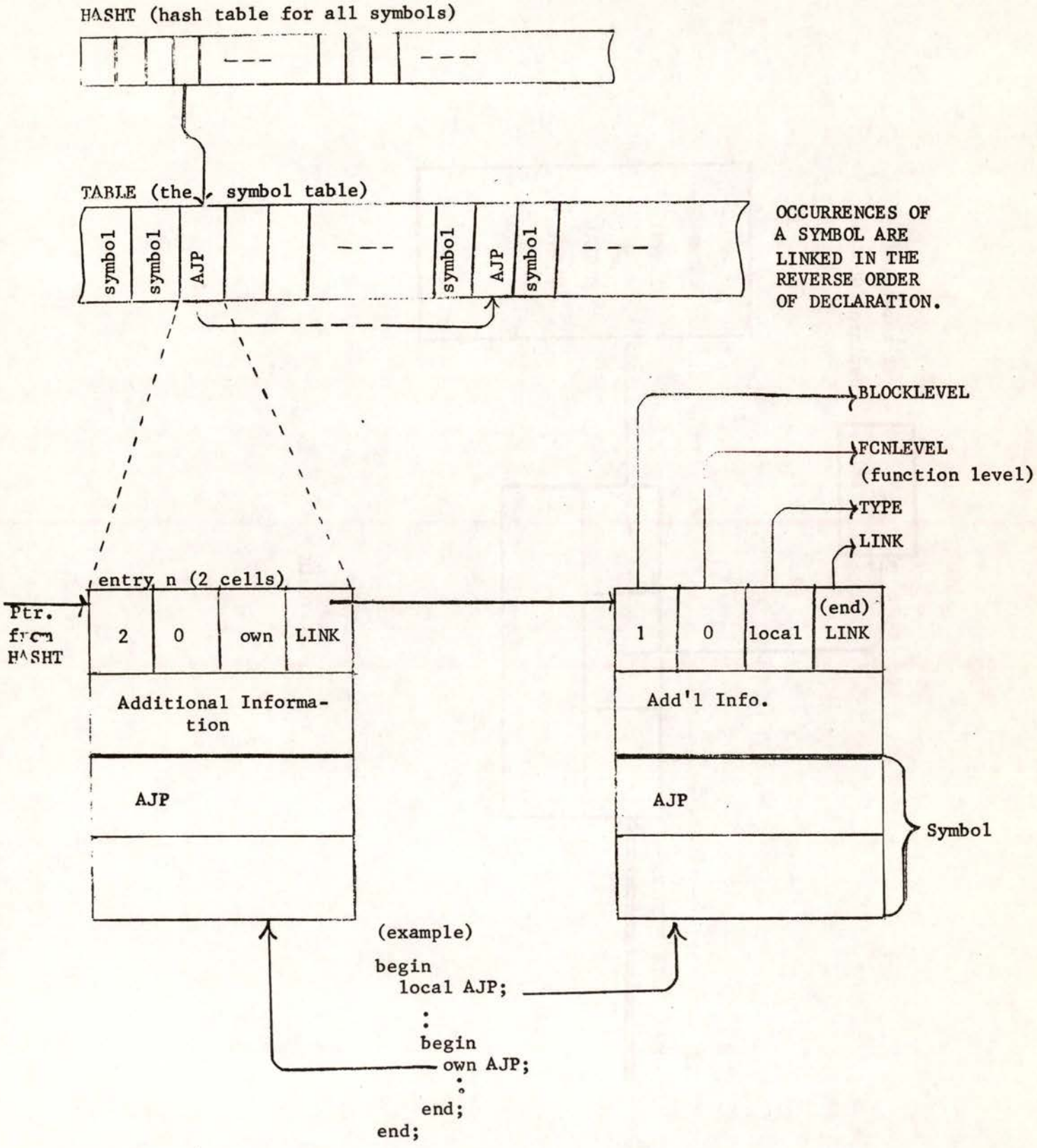
For x←.x+1 ... the BUFF and WINDOW look like:



THE WINDOW



THE GLOBAL TABLE

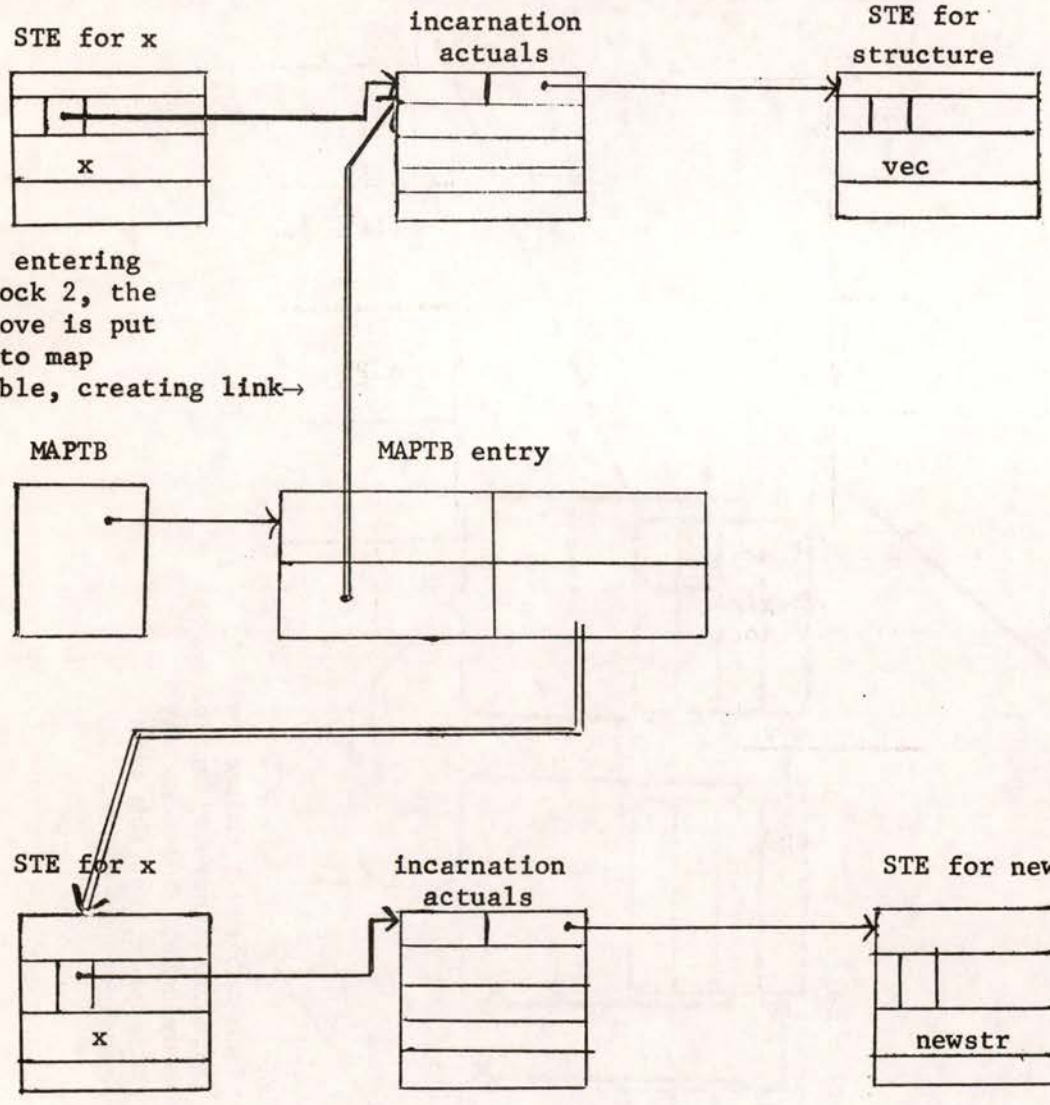


THE SYMBOL TABLE

For the code:

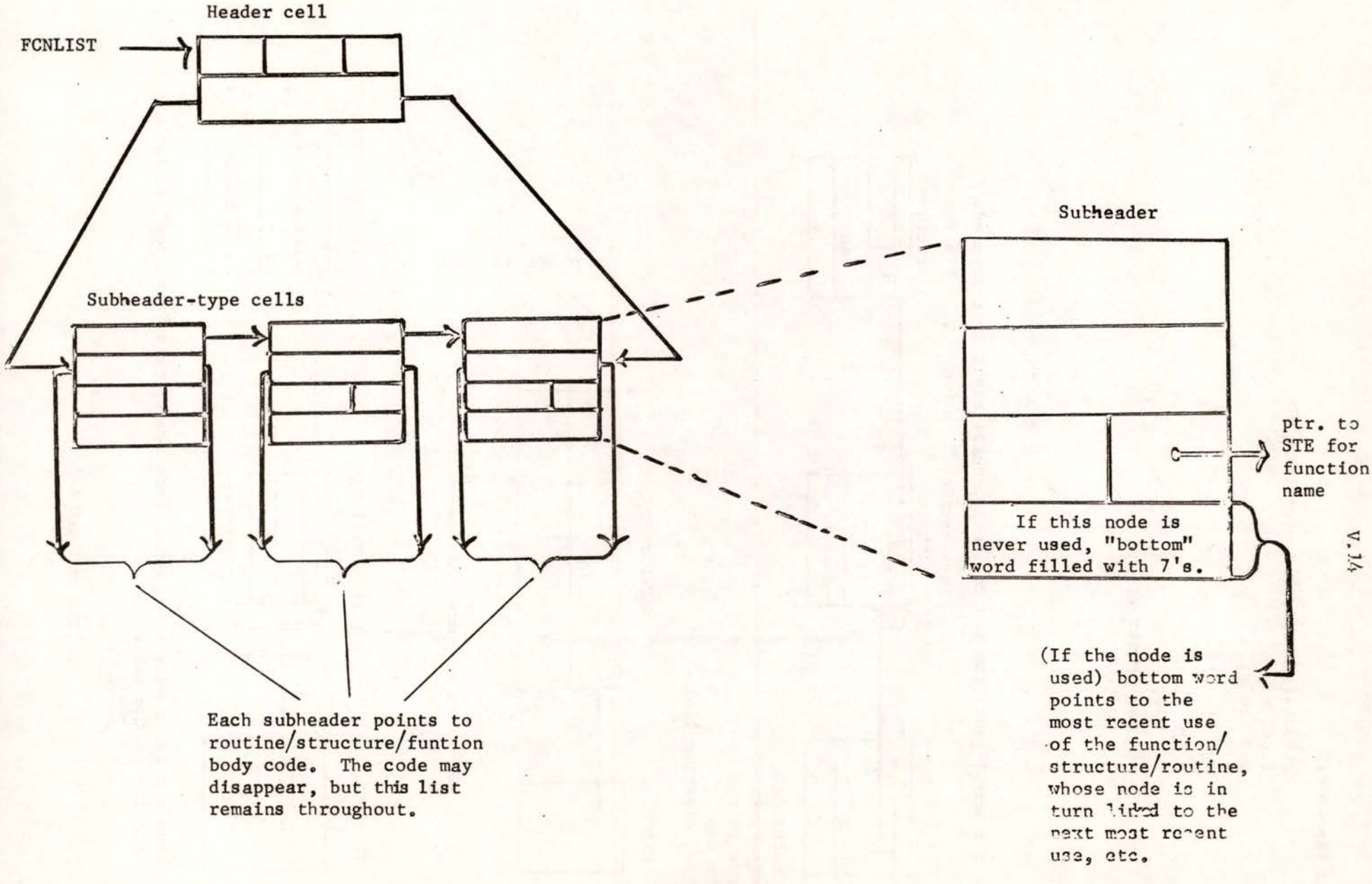
```
structure vec[i] = (.vec+.i)<0,36>;
local x;
map vec x;
:
:
x[5]← 1;
    begin
    map newstr x;
    end;
x[3]← 4;
:
:
```

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.

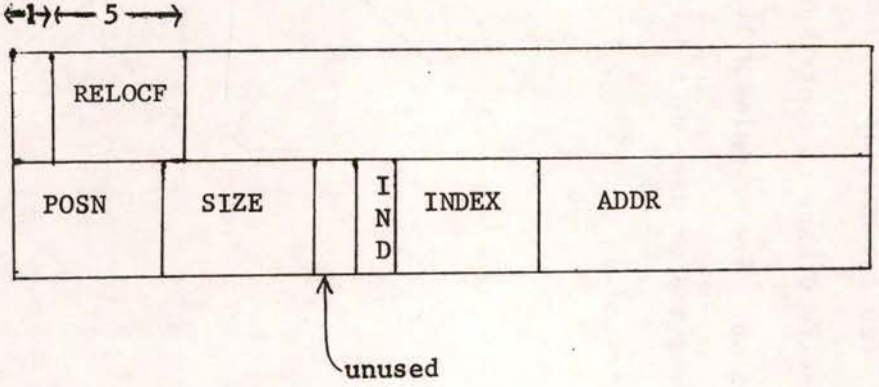
THE MAP TABLE



THE FUNCTION LIST

POINTER

RELOCF indicates whether ADDR is OWN, GLOBAL, etc.



POINTER TABLE

SECTION VI

BLISS EXAMPLES

This section contains a set of examples which illustrate the use of Bliss. Each example is intended to be fairly complete and self contained, and to illustrate one or more features of the language.

The authors would like to invite others to contribute further examples for inclusion in this section. New examples will be included if they clearly illustrate features and/or uses of the language which are not already adequately illustrated.

EXAMPLE 1: A TT-CALL I/O PACKAGE

Contributors: C. Geschke and W. Wulf

The following set of declarations defines a set of teletype input/output routines using the PDP-10 monitor TT-call mechanism. The set of functions is not complete, but adequate to illustrate the approach.

The declarations below provide the following functions:

INC	Input one character - wait for EOL before returning
OUTC	Output one character
OUTSA	Output ASCIZ-type string beginning at specified address
OUTS	Output ASCIZ-type string specified as the parameter
OUTM	Output multiple copies of a specified character
CR	Output carriage return
LF	Output line feed
NULL	Output null character
CRLF	Output carriage return and line-feed followed by 2 nulls
TAB	Output tab
OUTN	Output number in specified base and minimum number of digits
OUTD	Output decimal number with at least one digit
OUTO	Output octal number with at least one digit
OUTDR	Output decimal number with at least specified number of digits
OUTOR	Same as OUTDR except octal

```
MODULE TTIO(STACK)=BEGIN

MACHOP TTCALL=#51;

MACRO    INC= (REGISTER Q; TTCALL(4,Q); .Q)$,
         OUTC(Z)= (REGISTER Q; Q<-(Z); TTCALL(1,Q))$,
         OUTSA(Z)= TTCALL(3,Z)$,
         OUTS(Z)= OUTSA(PLIT ASCIZ Z)$,
         OUTM(C,N)= DECR I FROM (N)-1 TO 0 DO OUTC(C)$,
         CR= OUTC(#15)$,  LF= OUTC(#12)$,  NULL= OUTC(0)$,
         CRLF= OUTS('?M?J?0?0')$,
         TAB= OUTC(#11)$;


ROUTINE OUTN(NUM,BASE,REQD)=
  BEGIN OWN N,B,RD,T;
    ROUTINE XN=
      BEGIN LOCAL R;
        IF .N EQL 0 THEN RETURN OUTM("0",.RD-.T);
        R<-.N MOD .B; N<-.N/.B; T<-.T+1; XN();
        OUTC(.R+"0")
      END;

    IF .NUM LSS 0 THEN OUTC("-");
    B<-.BASE; RD<-.REQD; T<-0; N<-ABS(.NUM); XN()
  END;



MACRO    OUTD(Z)= OUTN(Z,10,1)$,
         OUTO(Z)= OUTN(Z,8,1)$,
         OUTDR(Z,N)= OUTN(Z,10,N)$,
         OUTOR(Z,N)= OUTN(Z,8,N)$;


!        THE PROGRAM BELOW PRINTS A TABLE OF INTEGERS, THEIR SQUARES, AND
!  THEIR CUBES;


    OWN N,C;

    CRLF; OUTS('INPUT AN INTEGER PLEASE ...');
    N<-0; WHILE (C<-INC) GTR "0" AND .C LSS "9" DO N<-.N*10+(.C-"0");

    CRLF; OUTS('A TABLE OF THE SQUARES AND CUBES OF 1-'); OUTD(.N);

    CRLF; INCR I FROM 1 TO 3 DO (TAB; OUTS(' X**'); OUTD(.I));

    CRLF; INCR I FROM 1 TO 3 DO (TAB; OUTM("-",5));

    INCR I FROM 1 TO .N DO
      BEGIN OWN X;
        X<-.I; CRLF;
        DECR J FROM 2 TO 0 DO (TAB; OUTD(.X); X<-.X*.I)
      END

END ELUDOM
```

Although the example is quite simple, there are several things about it which should be noted:

1. The use of a MACHOP declaration and embedded assembly code.
2. The use of macros to add a level of "syntactic sugar" and general cleanliness to the code.
3. The use of the escape character "?" in the CRLF macro to obtain control characters (e.g., carriage-return) in strings.
4. Parenthesization of macro parameters, as in OUTM, to insure proper hierarchy relations in the expansion.
5. The use of "DECR-TO-ZERO" in OUTM because it produces better code than "INCR-TO-EXPRESSION".
6. The use of own variables and the parameterless procedure XN in OUTN in order to avoid passing redundant parameters through the recursive levels of XN.
7. The fact that the local variable "R" is local to each recursive level of XN and hence its value is preserved at each level.

EXAMPLE 2: QUEUE MANAGEMENT MODEL

Contributors: C. Geschke and W. Wulf

This module contains routines to insert and delete items on doubly-linked queues. In addition it contains space management routines implementing the "Buddy System" (cf: Knuth: Vol. 1).

Buddy System

This is not intended to be a detailed description of the buddy system model of space management. We will simply give a brief description of this implementation of the scheme. The vector of allocatable space is called MEM. Space is allocated and deallocated from MEM by the routines GET and RELEASE, respectively. The basic unit of allocatable space is an item. Items are of size 2**ITEMSIZE where 0 < ITEMSIZE ≤ LOG2MEMSIZE. The first two words of an item are formatted:

ITEMSIZE	RLINK
<NOT-USED>	LLINK

Available items of size N are elements of a doubly linked list whose header is the two word cell SPACE[N]. The routines LINK and DELINK are called to enter and remove items from lists. The routine COLLAPSE is used to compactify two adjacent available items of size 2**N into an item of size 2**(N+1). The COLLAPSE routine iterates this process until no more compactification can take place.

Queue Model

In this model a queue is defined to be a doubly-linked list suspended from a header whose first three words are formatted as follows:

HEADERSIZE	RLINK
<NOT-USED>	LLINK
REMOVE	ENTER

The fields REMOVE and ENTER contain the addresses of the routines to be invoked when removing and entering items on the queue. To enter item X on queue Q, one simply makes the call ENQ(X,Q). ENQ then invokes the enter routine in Q's header which returns the address of the item in Q after which X is to be inserted. In a similar manner one removes the "next" item from queue Q by the call DEQ(Q). DEQ then invokes the remove routine in Q's header to return the address of the "next" item. The advantage of this scheme is that the queueing discipline is queue specific, and the same primitives (ENQ and DEQ) may be used independent of the discipline used for that queue. Examples of the enter and remove routines for LIFO, FIFO, and PRIORITY type queues appear at the end of this example module.

```
MODULE BMS(STACK)=

! BUDDY SYSTEM
!---------------

BEGIN

BIND MEMSIZE=1^12;

GLOBAL VECTOR MEM[MEMSIZE];

BIND LOG2MEMSIZE=35-FIRSTONE(MEMSIZE);

STRUCTURE ITEM[I,J,P,S]=
        CASE .I OF
          SET
            (.ITEM)<.P,.S>;
            (0.ITEM+.J)<.P,.S>;
            (00.ITEM+.J)<.P,.S>;
            (0(0.ITEM+1)+.J)<.P,.S>
          TES;

STRUCTURE VECTOR2[I]=
        [2*I](.VECTOR2+2*.I)<0,36>;

MACRO   BASE=0,0,0,18$,
        RLINK=1,0,0,18$,
        LLINK=1,1,0,18$,
        ITEMSIZE=1,0,18,18$,
        NXTRLINK=2,0,0,18$,
        NXTLLINK=2,1,0,18$,
        PRVRLINK=3,0,0,18$,
        PRVLLINK=3,1,0,18$;

GLOBAL VECTOR2 SPACE[LOG2MEMSIZE+1];

BIND VECTOR SIZE =
        PLIT(1^0,1^1,1^2,1^3,1^4,1^5,1^6,1^7,1^8,1^9,1^10,
             1^11,1^12);

MACRO   PARTNER(B1,B2,S)= ((((B1)-MEM<0,0>) XOR ((B2)-MEM<0,0>))
                           EQL .SIZE[S])$,
        REPEAT= WHILE 1 DO$,
        BASEADDR(B,S)= MEM[((B)-MEM<0,0>) AND NOT .SIZE[S]]<0,0>$,
        ERRMSG(S)= ERROR(PLIT ASCIZ S)$;
```

```
! SPACE-MANAGEMENT-ROUTINES
!----------------------------

FORWARD EMPTY,ERROR,LINK,DELINK,COLLAPSE;

GLOBAL ROUTINE GET(N)=

  !RETURNS THE ADDRESS OF AN ITEM OF SIZE 2**N

  BEGIN REGISTER ITEM R;
    IF .N LEQ 0 OR .N GTR LOG2MEMSIZE
      THEN ERRMSG('INVALID SPACE REQ');
    IF NOT EMPTY(SPACE[.N]<0,0>)
      THEN R[BASE]=DELINK(.SPACE[.N])
      ELSE
        BEGIN
          R[BASE]=GET(.N+1);
          COLLAPSE(.R[BASE]+.SIZE[.N],.N)
        END;
    R[ITEMSIZE]=.N;
    .R[BASE]
  END;


ROUTINE COLLAPSE(A,N)=

  !CALLED BY RELEASE AND GET TO ATTEMPT TO COMPACTIFY SPACE
  !IF ADJACENT ITEMS ARE FREE

  BEGIN MAP ITEM A; REGISTER ITEM L;
    REPEAT
      BEGIN
        L[BASE]=SPACE[.N]<0,0>;
        WHILE .L[RLINK] NEQ SPACE[.N]<0,0> DO
          IF PARTNER(.L[RLINK],.A[BASE],.N)
            THEN
              BEGIN
                A[BASE]=BASEADDR(DELINK(.L[RLINK]),.N);
                N=.N+1;
                EXITCOMPOUND[2]
              END
            ELSE L[BASE]=.L[RLINK];
        RETURN (A[ITEMSIZE]=.N; LINK(.A[BASE],.L[BASE]))
      END;
  END;

 GLOBAL ROUTINE RELEASE(A)=

  !CALLED TO RELEASE ITEM A

  BEGIN
    MAP ITEM A;
    COLLAPSE(.A[BASE],.A[ITEMSIZE])
  END;
```

```
! SIMPLE-LIST-ROUTINES
!----------------------

ROUTINE DELINK(A)=

  !REMOVES ITEM A FROM THE LIST TO WHICH IT IS APPENDED

  BEGIN MAP ITEM A;
    A[PRVRLINK]←.A[RLINK]; A[NXTLLINK]←.A[LLINK];
    A[RLINK]←A[LLINK]←.A[BASE]
  END;

ROUTINE LINK(A,TOO)=

  ! INSERTS ITEM A INTO A LIST IMMEDIATELY AFTER THE ITEM TOO

  BEGIN
    MAP ITEM A,TOO;
    A[LLINK]←.TOO[BASE]; A[RLINK]←.TOO[RLINK];
    TOO[NXTLLINK]←TOO[RLINK]←.A[BASE]
  END;


ROUTINE RELINK(A,TOO)=

  ! REMOVES ITEM FROM ITS PRESENT LIST AND INSERTS IT AFTER TOO

  LINK(DELINK(.A),.TOO);


ROUTINE EMPTY(L)=

  !PREDICATE INDICATING EMPTY LIST

  BEGIN MAP ITEM L;
    .L[BASE] EQL .L[RLINK]
  END;
```

```
! QUEUE-HANDLING-ROUTINES
!--------------------------

MACRO    QHDR=ITEM$;

MACRO    ENTER=1,2,0,18$,
         REMOVE=1,2,18,18$;


GLOBAL ROUTINE ENQ(A,Q)=

  ! ENTERS ITEM A ON QUEUE Q ACCORDING TO THE INSERTION DISCIPLINE
  ! EVOKED BY Q'S ENTER ROUTINE

  BEGIN
    MAP QHDR Q;
    RELINK(.A,(.Q[ENTER])(.Q[BASE],.A))
  END;


GLOBAL ROUTINE DEQ(Q)=

  ! REMOVES AN ITEM FROM QUEUE Q ACCORDING TO THE REMOVAL DISCIPLINE
  ! EVOKED BY Q'S REMOVE ROUTINE

  BEGIN
    MAP QHDR Q;
    DELINK((.Q[REMOVE])(.Q[BASE]))
  END;


! MISC SERVICE ROUTINES
!-----------------------


ROUTINE ERROR(A)=
  BEGIN MACHOP TTCALL=#051;
    TTCALL(3,.A)
  END;


ROUTINE INITIALIZE=

  !INITIALIZES THE SPACE MANAGEMENT DATA

  BEGIN REGISTER ITEM X;
    X[BASE]←MEM<0,0>;
    X[RLINK]←X[LLINK]←SPACE[LOG2MEMSIZE]<0,0>;
    X[ITEMSIZE]←LOG2MEMSIZE;
    DECR I FROM LOG2MEMSIZE-1 TO 0 DO
      SPACE[.I]←(SPACE[.I]+1)<0,36>←SPACE[.I]<0,0>;
    SPACE[LOG2MEMSIZE]←(SPACE[LOG2MEMSIZE]+1)<0,36>←MEM<0,0>
  END;
```

```
! EXAMPLES OF VARIOUS QUEUE MODELS
!-----------------------------------

! LIFO QUEUE
!------------

ROUTINE LIFOREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;


ROUTINE LIFOENTER(Q,A)=
  BEGIN
    MAP QHDR Q;
    .Q[BASE]
  END;



! FIFO QUEUE
!------------


ROUTINE FIFOREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;



ROUTINE FIFOENTER(Q,A)=
  BEGIN
    MAP QHDR Q;
    .Q[LLINK]
  END;
```

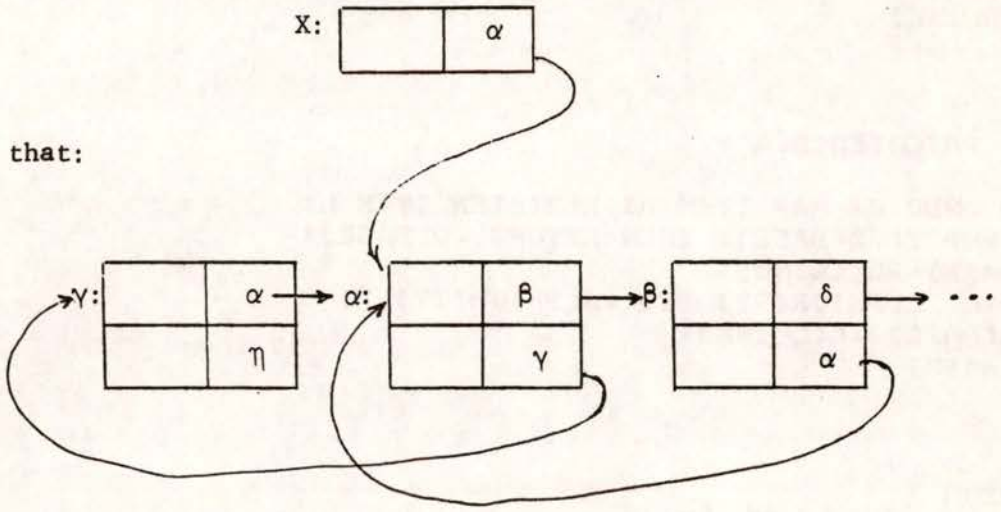
```
! PRIORITY QUEUE
!----------------

MACRO   PRIORITY=1,1,18,18$;

ROUTINE PRIREMOVE(Q)=
  BEGIN
    MAP QHDR Q;
    IF EMPTY(.Q[BASE]) THEN
      ERRMSG('INVALID DEQ REQUEST');
    .Q[RLINK]
  END;


ROUTINE PRIENTER(Q,A)=
  BEGIN
    MAP QHDR Q; MAP ITEM A; REGISTER ITEM L;
    IF EMPTY(.Q[BASE]) THEN RETURN .Q[BASE];
    L[BASE]←.Q[LLINK];
    UNTIL .L[PRIORITY] GEQ .A[PRIORITY] DO
      L[BASE]←.L[LLINK];
    .L[BASE]
  END;


END ELUDOM
```

Comments on the Use of Bliss in the Implementation

(1) The structure ITEM is particularly interesting and perhaps at first a bit obscure.

To illustrate, consider a variable X structured by item:

Assuming that the right half of X contains α:



and that:

Then:

.X[BASE] ≡ α	.X[NXTRLINK] = δ
.X[RLINK] ≡ β	.X[NXTLLINK] = α
.X[LLINK] ≡ γ	.X[PRVRLINK] = α
	.X[PRVLLINK] = η

The structure ITEM uses the "constant case" expression to distinguish between the pointer, the pointee, and the pointee's predecessor and successor.

(2) The structure VECTOR2 has a size expression [2*I] which is used in the allocating declaration:

```
GLOBAL VECTOR2 SPACE[LOG2MEMSIZE+1];
```

(3) Since the addresses of the 'remove' and 'enter' routines are stored in the queue header, the expression

```
(.Q[REMOVE])(.Q[BASE])
```

is a call of the routine whose address is .Q[REMOVE] and passes it to the base address of the queue or its parameter.

(4) The macro 'REPEAT = WHILE 1 DO' defines an infinite loop - its only exit is defined by the RETURN expression in its body.

(5) Notice the 'BIND VECTOR SIZE = PLIT(1↑0,1↑1,1↑2,...' in the space allocator. The value of SIZE is a pointer to this sequence of values, and in particular the value of '.SIZE[.N]' is 2^N.

EXAMPLE 3: DISCRIMINATION NET

Contributor: D. Wile

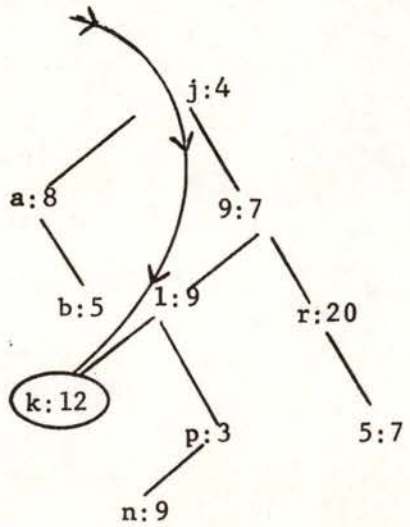
A discrimination net is a mechanism used to associate "information" with "names". The net is actually a tree, each node of which consists of a name and the information associated with that name, as well as a set of pointers to other nodes. To look up a name in the net we start at the root node and see if the name in the node matches our target name. If it does, we return the associated information.

Otherwise, we use a "discrimination function" which determines which subnode to examine next (usually as a function of the target name and the name of the current node). If there is no corresponding subnode, a new node must be created.

For example, a binary net (two subnodes/node) with a discrimination function which chooses the left branch if the target name is alphabetically smaller than the name in the node, is illustrated below:

Name: j, 9, 1, a, b, r, p, n, s, k

Inf: 4, 7, 9, 8, 5, 20,3, 9, 7, 12



In the implementation which follows, there are three globally defined routines:

1. DSCINIT (String address) -- returns a pointer to the information field of the node associated with the string. This must be called first to initialize the net. (The information field will be zeroed when the node is new.)

2. DSCLKP (String address) -- the "lookup" routine. Value returned as above.

3. DSCPNAME (Information field address) -- returns a pointer to the print name associated with the particular information field.

The implementation is designed to allow the user to create a module somewhat "tailored" to his needs. The module is created by passing:

1. the estimated number of entries to be inserted into the table;
2. the average number of words each name will occupy;
3. the number of words in the "information field";
4. the number of subnodes of each node (e.g., binary example above, 2);
5. a string which executes an error routine

in that order, to a macro "DSCRIMINET". Two macros must be defined previous to the DSCRIMINET expansion:

1. DSCIMINATE (Target string address, current node string address) must have a value of -1 if the strings match. Otherwise, its value must be between 0 and 1 less than the number of subnodes.

2. DSCCOPY (To address, From address) copies the string from the "from address" to the "to address", returning the number of words occupied by the copy.

```
MODULE NET(STACK=GLOBAL(STABK,#400))=
BEGIN
MACRO
  DSCRIMINET(MAXNUMENT,AVNAMESIZE,INFSIZE,NOSUBNODES,ERROR)=

    BEGIN
      %N.B.: ALL VECTOR ACCESSES ARE INDIRECT THROUGH THE BASE%
      STRUCTURE VECTOR[I]=(0.VECTOR+.I)<0,36>;

      % NET SPACE ALLOCATION, STRUCTURE DEFINITION AND
        INITIALIZATION DEFINITIONS %
      BIND TABLELEN=MAXNUMENT*((NOSUBNODES+1)/2+INFSIZE+AVNAMESIZE);
      OWN BASENODE[TABLELEN];
      BIND MAXADD=BASENODE+TABLELEN;

      BIND SUBNODE=0, INF=1, PNAME=2,
           INFOFFSET=(NOSUBNODES+1)/2,
           PNAMEOFFSET=INFOFFSET+INFSIZE;

      STRUCTURE NODE[SUBFIELD,INDEX]=CASE .SUBFIELD OF
        SET .NODE[.INDEX*(-1)]<IF .INDEX THEN 18,18>;
            .NODE[INFOFFSET];
            .NODE[PNAMEOFFSET] TES;

      GLOBAL ROUTINE DSCPNAME(INFPOS)=
        (.INFPOS+INFSIZE)<0,36>;

      OWN NODE NEXTCELL;

      ROUTINE INITNODE(CELL,STRING)=
        BEGIN
          DECR I FROM PNAMEOFFSET-1 TO 0 DO CELL[.I]=0;
          IF MAXADD LEQ (NEXTCELL=.NEXTCELL+PNAMEOFFSET+
            (MAP NODE CELL; DSCCOPY(CELL[PNAME],.STRING)))

            THEN ERROR ELSE .CELL
        END;

      GLOBAL ROUTINE DSCINIT(STRING)=
        BEGIN
          LOCAL NODE RETVAL;
          NEXTCELL=BASENODE;
          RETVAL=INITNODE(BASENODE,.STRING);
          RETVAL[INF]
        END;

      ROUTINE NEWCELL(STRING)=INITNODE(.NEXTCELL,.STRING);

      % THE LOOKUP ROUTINE ITSELF %
      GLOBAL ROUTINE DSCLKP(STRING)=
        BEGIN
          LOCAL DISCIND, NODE CURRENT:NEXT;
          NEXT=BASENODE;
```

```
        DO
          BEGIN
            CURRENT←.NEXT;
            IF (DISCIND←DSCIMINATE(.STRING,CURRENT[PNAME])) LSS 0
              THEN RETURN CURRENT[INF];
            NEXT←.CURRENT[SUBNODE,.DISCIND]
          END

          UNTIL .NEXT EQL 0;

        NEXT←CURRENT[SUBNODE,.DISCIND]←NEWCELL(.STRING);
        NEXT[INF]
      END;
  END;;


ROUTINE DSCIMINATE(L,R)=
  BEGIN
    STRUCTURE VECTOR[I]=(0.VECTOR+.I)<0,36>;
    INCR I FROM 0
      DO BEGIN
           BIND LEFT=.L[.I], RIGHT=.R[.I];
           IF LEFT NEQ RIGHT THEN EXITLOOP (LEFT LSS RIGHT);
           IF (LEFT AND #376) EQL 0 THEN EXITLOOP -1
         END
  END;

ROUTINE DSCCOPY(INTO,FRO)=
  BEGIN
    STRUCTURE VECTOR[I]=(0.VECTOR+.I)<0,36>;
    INCR I FROM 0 DO
      IF ((INTO[.I]←.FRO[.I]) AND #376) EQL 0
        THEN EXITLOOP .I+1
  END;

EXTERNAL ERROR;
DSCRIMINET(500,3,1,2,ERROR(PLIT 'LOOKUP TABLE OVERFLOW'))

BEGIN
  BIND NAMES=PLIT(
    PLIT ASCIZ 'FIRSTNAME',
    PLIT ASCIZ 'SECOND',
    PLIT ASCIZ 'SS',
    PLIT ASCIZ 'A LONGISH NAME',
    PLIT ASCIZ 'L',
    PLIT ASCIZ '77788()34');

  EXTERNAL DSCLKP, DSCINIT;
  DSCINIT(PLIT 'ZEROTH NAME')←-3;
  INCR I FROM 0 TO .NAMES[-1]-1 DO DSCLKP(.NAMES[.I])←.I;
  INCR I FROM 0 TO .NAMES[-1]-1 BY 2 DO DSCLKP(.NAMES[.I])←.I+1↑35;
END;
END ELUDOM;;
```

Notes on the Implementation

The Bliss module above implements the example described at the beginning of this section. The test program portion of the module simply initializes the table, inserts the six strings in the plit into the table (associating as information, the index in the plit), and runs through the evenly indexed items in the plit, turning on the sign bit in the information word.

Of interest:

1. The vector structure (which defaults as the structure for all unmapped variables and expressions) is redefined "indirectly"; this is fairly dangerous in any program, and represents an after-the-fact programming decision.

2. The physical structure of the table is kept independent of the logical structure as used by the lookup routine; no reference is made from the lookup routine to the structure other than through the structured nodes.

3. The binds, structures, own declarations and even the initialization function - requiring knowledge of the physical structure are kept grouped and separate. Note, for example, that INITNODE uses both a vector mapping on contiguous fields of CELL and the NODE structure.

4. The physical structure of the tree is kept isolated from the user of the routines to the extent that only knowledge

that the mechanism is associative is of importance -- the particular lookup algorithm and storage management are independent of the functional use of the module.

5. Bliss programming "tricks":

   a. Use of the constant case expression for sub-fields of structures (NODE in this case);
   b. Default use of 0 for the omitted else in the structure case defining the SUBNODE field;
   c. CELL remapped in the INITNODE routine to take advantage of knowledge of the physical layout of the NODE's storage.
   d. "Dynamic" binds of LEFT and RIGHT inside the loop in the test version discrimination function;
   e. The bind to a plit (of NAMES) in the test portion, to prevent duplicate storage allocation for the twice-used plit;
   f. Stores into routine cells in the test program loops;
   g. Use of the plit length word preceding the plit (NAMES[-1]).

Example 4: Simple Monitor I/O

Contributor: Joseph M. Newcomer

These routines supply low-level support for programs which run in the user mode under the PDP-10 timesharing monitor. Although not all I/O facilities are shown in this example, the same methods may be used to provide any I/O facility in BLISS.

These routines compile to about 75 words of code, and form about the smallest set of routines required to read and write files. The restriction they impose is that there may be only one input file and one output file, and the channels associated with these files must be specified at compile time.

```
MODULE IO(ENTRIES=(LOOKUP,ENTER,OPENIN,OPENOUT,CLOSEIN,
                   CLOSEOUT,PURGEOUT,OUTMSG,READ,WRITE))=
BEGIN
%%
%

        THIS MODULE PROVIDES SOME I/O EXAMPLES USING
    BLISS.  FOR THIS SET OF EXAMPLES, WE WILL ASSUME A
    STATIC CHANNEL ASSIGNMENT.  ALL INPUT WILL BE ON
    CHANNEL "INCH", AND ALL OUTPUT ON CHANNEL "OUTCH".
    THE THREE-WORD BUFFER HEADER AREAS WILL BE "IBUFH"
    AND "OBUFH" AND ARE DECLARED GLOBAL IN SOME OTHER
    MODULE.
%
%%

BIND
        INCH=1,
        OUTCH=2;

%%
%
        HERE ARE SOME USEFUL MACHINE OPERATIONS
%
%%

MACHOP
        CALLI=#047,
        OPEN=#050,
        TTCALL=#051,
        IN=#056,
        OUT=#057,
        GETSTS=#062,
        STATZ=#063,
        CLOSEUUO=#070,
        RELEAS=#071,
        LOOKUPUUO=#076,
        ENTERUUO=#077,
        XCT=#256;

%%
%

        THE FOLLOWING MACRO RETURNS "TRUE" IF THE
    INSTRUCTION GIVEN AS ITS PARAMETER SKIPS, AND FALSE
    IF IT DOES NOT.
%
%%

MACRO
        SKIP(OP)=BEGIN
                        VREG←1; OP; VREG←0; .VREG
                ENDS;
```

```
%%
%
        HERE ARE SOME USEFUL MACROS
%
%%

MACRO
        RESET=CALLI(0)$,
        COUNT(BUFH)=BUFH[2]$,
        PTR(BUFH)=BUFH[1]$;

%%
%

        IBUFH AND OBUFH MUST BE DECLARED GLOBAL IN
        ANOTHER MODULE
%
%%

EXTERNAL
        IBUFH[3],
        OBUFH[3];

%%
%

        "OPENIN" AND "OPENOUT" TAKE THREE PARAMETERS:
        THE DEVICE STATUS (INCLUDING DATA MODE), THE LOGICAL
        DEVICE NAME, AND THE BUFFER POINTERS, PRECISELY AS
        SPECIFIED FOR THE OPEN UUO. THESE ROUTINES RETURN
        "TRUE" IF THEY SUCCEEDED, AND "FALSE" IF THEY
        FAILED.
%
%%

GLOBAL ROUTINE OPENIN(STATUS,LDEV,BUF)=
                (SKIP(OPEN(INCH,STATUS)));

GLOBAL ROUTINE OPENOUT(STATUS,LDEV,BUF)=
                (SKIP(OPEN(OUTCH,STATUS)));

%%
%

        THE PARAMETER PASSED TO LOOKUP OR ENTER MUST BE
        THE ADDRESS OF A FOUR-WORD CONTROL BLOCK, AS
        SPECIFIED IN THE MANUAL. THE CONTENTS OF THIS
        CONTROL BLOCK WILL BE ALTERED BY THE UUO, AND
        CONSEQUENTLY MUST NOT RESIDE IN THE HISEG. THESE
        ROUTINES RETURN "TRUE" IF THEY SUCCEED AND "FALSE"
        IF THEY FAIL.
%
%%
```

```
GLOBAL ROUTINE LOOKUP(LOOKUPBLOCK)=
               SKIP(LOOKUPUUO(INCH,LOOKUPBLOCK,0,1));

GLOBAL ROUTINE ENTER(ENTERBLOCK)=
               SKIP(ENTERUUO(OUTCH,ENTERBLOCK,0,1));

%%
%
          THE ROUTINE "READ" RETURNS ONE OF THE FOLLOWING
        VALUES:  -1 IF END-OF-FILE; -2 IF OTHER I/O ERROR; A
        POSITIVE NUMBER IS THE CHARACTER RETURNED.
%
%%

GLOBAL ROUTINE READ=
    BEGIN
        IF (COUNT(IBUFH)-.COUNT(IBUFH)-1) LEQ 0
          THEN
            BEGIN
                IF SKIP(IN(INCH))
                   THEN
                        (IF SKIP (STATZ(INCH,#740000))
                                THEN
                                        -1
                                ELSE
                                        -2)
                    ELSE
                        SCANI(PTR(IBUFH))
            END
          ELSE
                SCANI(PTR(IBUFH))
    END;

%%
%
          THE ROUTINE "WRITE" RETURNS "TRUE" IF IT
        SUCCESSFULLY WROTE OUT THE CHARACTER GIVEN AND
        "FALSE" IF IT DID NOT.
%
%%

GLOBAL ROUTINE WRITE(CHAR)=
    BEGIN
        IF (COUNT(OBUFH)-.COUNT(OBUFH)-1) LEQ 0
          THEN
                BEGIN
                        IF SKIP(OUT(OUTCH)) THEN RETURN 0
                END;
        REPLACEI(PTR(OBUFH),.CHAR);
        1
    END;
```

```
%%
%
        THE FOLLOWING ROUTINE PURGES THE OUTPUT BUFFER.
      THIS IS PARTICULARLY USEFUL FOR TTY OUTPUT.
%
%%

GLOBAL ROUTINE PURGEOUT=
        NOT SKIP(OUT(OUTCH));

%%
%
        THE FOLLOWING ROUTINES CLOSE THE CHANNELS AND
      RELEASE THE DEVICES.  THEIR VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE CLOSEIN=(CLOSEUUO(INCH); RELEAS(INCH));

GLOBAL ROUTINE CLOSEOUT=(CLOSEUUO(OUTCH); RELEAS(OUTCH));

%%
%
        THIS ROUTINE IS USEFUL FOR OUTPUTTING MESSAGES
      TO THE TELETYPE.  IT TAKES ONE PARAMETER, THE
      POINTER TO AN ASCIZ STRING (E.G., PLIT ASCIZ
      'TEXT...').  ITS VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE OUTMSG(TEXT)=
        TTCALL(3,TEXT,,1);


END ELUDOM;
```

Notes:

1. The MACHOP feature is used to provide the opcodes for the UUO's. Note that the names and values need not correspond to existing instruction names.

2. The SKIP macro is used to detect if the machine operation which is its argument skips upon return. Note its use of the VREG to produce better code. This macro is of interest to anyone who will want to write monitor UUO's.

3. Note the use of SCANI and REPLACEI in the READ and WRITE routines, respectively.

4. Note the fact that the parameter lists to OPENIN and OPENOUT appear on the stack in the correct format for the OPEN UUO, which merely has to point to the first parameter of the list.

5. Indirect addressing is used in the LOOKUP, ENTER and OUTMSG routines. This produces slightly better code.

6. The ENTRIES declaration is used in the module head to make the routine names available for library search by the loader.

7. The macros defining the count and pointer fields are used to make the code read clearly. Thus the essence of the operations is seen without the details.

Example 5: A Sample Program Using Example 4

Contributor: Joseph M. Newcomer

This is a somewhat trivial example of a program which transfers an ASCII file from the disk to the printer. We will assume in this example that the name of the file is "FILE.EXT". Note that no test is made for the existence of the file. If it does not exist (i.e., LOOKUP fails) this will appear as an input error to READ.

```
MODULE DSKLPT(STACK)=
BEGIN
%%
%
          THIS  MODULE  TRANSFERS  ASCII  FILES  TO  THE
        PRINTER.  FOR ILLUSTRATIVE PURPOSES HERE, THE INPUT
        FILE IS ALWAYS CALLED "FILE.EXT".
%
%%

EXTERNAL
        CLOSEIN, CLOSEOUT,
        OPENIN, OPENOUT, LOOKUP, OUTMSG, READ, WRITE;

OWN
        LBLOCK[4],
        T;

GLOBAL
        OBUFH[3], IBUFH[3];

MACRO
        RESET=CALLI(0,0)$,
        STOP=CALLI(1,#12)$;

MACHOP
        CALLI=#047;

%%
%
        HERE IS THE MAIN PROGRAM
%
%%

        RESET;

UNTIL OPENOUT(1, SIXBIT 'LPT', OBUFH^18) DO
        (OUTMSG(PLIT ASCIZ '?? NO LPT?M?J'); STOP);

UNTIL OPENIN(1, SIXBIT 'DSK', IBUFH<0,0>) DO
        (OUTMSG(PLIT ASCIZ '?? NO DSK?M?J'); STOP);


        LBLOCK[0]←SIXBIT 'FILE';
        LBLOCK[1]←SIXBIT 'EXT';
        LBLOCK[2]←LBLOCK[3]←0;

        LOOKUP(LBLOCK<0,0>);

        WHILE (T←READ()) GEQ 0 DO
                (IF .T NEQ 0 THEN IF NOT WRITE(.T)
                                  THEN EXITLOOP(T←-3));
```

```
        CLOSEIN(); CLOSEOUT();

        CASE .T+3 OF
                SET
                % -3 % OUTMSG(PLIT ASCIZ '??LPT ERROR?M?J');
                % -2 % OUTMSG(PLIT ASCIZ '??DSK ERROR?M?J');
                % -1 % OUTMSG(PLIT ASCIZ 'DONE?M?J');
                TES;

ENDS
```

Notes:

1. The STACK declaration is used because this is the main program. Since no parameters are given, the stack is an OWN array, size 1000 words, with the name STACK.

2. Macros define the CALLI UUO's for RESET and EXIT ("STOP").

3. The STOP macro is used in loops, which allows the user to issue an ASSIGN command and type CONT to resume if an error occurs.

4. EXITLOOP is used to terminate processing in the case of an output error. The loop condition is used to terminate processing in the case of an input error.

5. The CASE expression is used to output the appropriate message. Note the use of comments to show which value of T will execute a given expression within the CASE. This method of commenting CASE expressions makes them very readable.

6. The attributes SIXBIT and ASCIZ are used to define strings.

7. The escape convention (? character) is used in the messages to obtain control characters. Note the use of the double question mark (??) to obtain a single question mark.

Example 6: Monitor I/O

Contributor: Joseph M. Newcomer

These routines supply low-level support for programs which run in the user mode under the PDP-10 timesharing monitor. Although not all I/O facilities are shown in this example, the same methods may be used to provide any I/O facility in BLISS.

These routines differ from those in Example 4 in that they allow the channel number to be supplied as one parameter to each routine. There is no restriction to doing input or output on only one channel. These routines use about 128 words.

```
MODULE IO(ENTRIES=(LOOKUP,ENTER,OPEN,CLOSE,
                   PURGEOUT,OUTMSG,READ,WRITE))=

BEGIN
%%
%
        THIS MODULE PROVIDES SOME I/O EXAMPLES USING
        BLISS.  FOR THIS SET OF EXAMPLES, WE WILL USE
        DYNAMIC CHANNEL ASSIGNMENT.
%
%%


%%
%
        HERE ARE SOME USEFUL MACHINE OPERATIONS
%
%%

MACHOP
        CALLI=#047,
        TTCALL=#051,
        XCT=#256;

BIND
        OPENUUO=#050,
        IN=#056,
        OUT=#057,
        GETSTS=#062,
        STATZ=#063,
        CLOSEUUO=#070,
        RELEAS=#071,
        LOOKUPUUO=#076,
        ENTERUUO=#077;

%%
%
        THE FOLLOWING MACRO RETURNS "TRUE" IF THE
        INSTRUCTION GIVEN AS ITS PARAMETER SKIPS, AND FALSE
        IF IT DOES NOT.
%
%%

MACRO
        SKIP(OP)=BEGIN
                        VREG←1; OP; VREG←0; .VREG
                END$;
```

```
%%
%
        HERE ARE SOME USEFUL MACROS
%
%%

MACRO
        RESET=CALLI(0)$,

        ICOUNT(CHNL)=(.BUFH[CHNL]<0,18>+2)<0,36>$,
        IPTR(CHNL)=(.BUFH[CHNL]<0,18>+1)<0,36>$,
        OCOUNT(CHNL)=(.BUFH[CHNL]<18,18>+2)<0,36>$,
        OPTR(CHNL)=(.BUFH[CHNL]<18,18>+1)<0,36>$;
%%
%

        THIS VECTOR KEEPS THE BUFFER HEADER POINTERS
        PASSED TO "OPEN".
%
%%

OWN
        BUFH[16];

%%
%

        THE FOLLOWING MACROS ARE USED TO CONSTRUCT
        INSTRUCTIONS AND EXECUTE THEM.
%
%%

MACRO
        MAKEOP(OP,REG,ADDR)=(OP<0,0>^27+REG^23+ADDR)$,
        EXECUTE(X)=(REGISTER Q; Q←X; SKIP(XCT(0,Q)))$,
        IND=0,0,0,1$;

%%
%

        "OPEN" TAKES FOUR PARAMETERS: THE CHANNEL
        NUMBER, THE DEVICE STATUS (INCLUDING DATA MODE), THE
        LOGICAL DEVICE NAME, AND THE BUFFER POINTERS, THE
        LATTER THREE PRECISELY AS SPECIFIED FOR THE OPEN
        UUO. THIS ROUTINE RETURNS "TRUE" IF IT SUCCEEDED,
        AND "FALSE" IF IT FAILED.
%
%%

GLOBAL ROUTINE OPEN(CHNL,STATUS,LDEV,BUF)=
        (BUFH[.CHNL]←.BUF;
         EXECUTE(MAKEOP(OPENUUO,.CHNL,STATUS<0,0>)));
```

```
%%
%
        THE SECOND PARAMETER PASSED TO LOOKUP OR ENTER
      MUST BE THE ADDRESS OF A FOUR-WORD CONTROL BLOCK, AS
      SPECIFIED IN THE MANUAL.  THE CONTENTS OF THIS
      CONTROL BLOCK WILL BE ALTERED BY THE UUO, AND
      CONSEQUENTLY MUST NOT RESIDE IN THE HISEG.  THESE
      ROUTINES RETURN "TRUE" IF THEY SUCCEED AND "FALSE"
      IF THEY FAIL.
%
%%

GLOBAL ROUTINE LOOKUP(CHNL,LOOKUPBLOCK)=
       EXECUTE(MAKEOP(LOOKUPUUO,.CHNL,LOOKUPBLOCK<IND>));

GLOBAL ROUTINE ENTER(CHNL,ENTERBLOCK)=
       EXECUTE(MAKEOP(ENTERUUO,.CHNL,ENTERBLOCK<IND>));

%%
%
        THE ROUTINE "READ" RETURNS ONE OF THE FOLLOWING
      VALUES:  -1 IF END-OF-FILE; -2 IF OTHER I/O ERROR; A
      POSITIVE NUMBER IS THE CHARACTER RETURNED.
%
%%

GLOBAL ROUTINE READ(CHNL)=
    BEGIN
       IF (ICOUNT(.CHNL)←.ICOUNT(.CHNL)-1) LEQ 0
         THEN
           BEGIN
               IF EXECUTE(MAKEOP(IN,.CHNL,0))
                THEN
                 RETURN
                 (IF EXECUTE(MAKEOP(STATZ,.CHNL,#740000))
                       THEN
                                 -1
                       ELSE
                                 -2)
           END;
       SCANI(IPTR(.CHNL))
    END;

%%
%
        THE ROUTINE "WRITE" RETURNS "TRUE" IF IT
      SUCCESSFULLY WROTE OUT THE CHARACTER GIVEN AND
      "FALSE" IF IT DID NOT.
%
%%
```

```
GLOBAL ROUTINE WRITE(CHNL,CHAR)=
    BEGIN
        IF (OCOUNT(.CHNL)←.OCOUNT(.CHNL)-1) LEQ Ø
          THEN
            (IF EXECUTE(MAKEOP(OUT,.CHNL,Ø)) THEN RETURN Ø);
        REPLACEI(OPTR(.CHNL),.CHAR);
        1
    END;

%%
%
            THIS ROUTINE PURGES THE OUTPUT BUFFER.  THIS IS
        PARTICULARLY USEFUL FOR TTY OUTPUT.
%
%%

GLOBAL ROUTINE PURGEOUT(CHNL)=
        NOT EXECUTE(MAKEOP(OUT,.CHNL,Ø));

%%
%
            THE FOLLOWING ROUTINE CLOSES  THE  CHANNEL  AND
        RELEASES THE DEVICE.  ITS VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE CLOSE(CHNL)=
        (EXECUTE(MAKEOP(CLOSEUUO,.CHNL,Ø));
         EXECUTE(MAKEOP(RELEAS,.CHNL,Ø)));

%%
%
            THIS ROUTINE IS USEFUL FOR OUTPUTTING  MESSAGES
        TO  THE  TELETYPE.   IT  TAKES  ONE  PARAMETER,  THE
        POINTER  TO  AN  ASCIZ  STRING  (E.G.,  PLIT  ASCIZ
        'TEXT...').  ITS VALUE IS UNDEFINED.
%
%%

GLOBAL ROUTINE OUTMSG(TEXT)=
        TTCALL(3,TEXT,,1);


END ELUDOM;
```

Notes:

1. The MACHOP feature is used to provide the opcodes for the UUO's CALLI and TTCALL, as well as the machine operation XCT.

2. The BIND declaration is used to associate names with the values of the UUO opcodes. The reason this is done, instead of declaring these names as MACHOPs (as is done in example 4) will be discussed below.

3. The SKIP macro is used to detect if the machine operation which is its argument skips upon return. Note its use of the VREG to produce better code. This macro is of interest to anyone who will want to write monitor UUO's.

4. In any UUO dealing with I/O, the register field is the channel number. In order to provide for varying channel numbers, the value of this field must be evaluated at execution time. However, the BLISS compiler requires that this field evaluate to a constant at compile time. In order to allow this field to vary at execution time, we construct the instruction and execute it by means of the XCT instruction. The macro MAKEOP constructs an instruction, given the parameters of its opcode, register field, and address field. The macro EXECUTE takes the expression given as its parameter, considers it an intruction, and executes it. Note that EXECUTE assumes the possibility that the instruction will skip upon return, and its value is the value of the SKIP macro. A minor restriction is that the name "Q" may not be used in the argument to the EXECUTE macro. However, the choice of this name is arbitrary and any name may be used in coding the macro.

5. Note the use of SCANI and REPLACEI in the READ and WRITE routines, respectively.

6. Note the fact that the parameter list to OPEN appears on the stack in the correct format for the OPEN UUO, which merely has to point to the first parameter of the list.

7. Indirect addressing is used in the LOOKUP, ENTER and OUTMSG routines. This produces slightly better code.

8. The ENTRIES declaration is used in the module head to make the routine names available for library search by the loader.

9. The macros defining the count and pointer fields are used to make the code read clearly. The essence of the operations is seen without the details.

Example 7: A Sample Program Using Example 6

Contributor: Joseph M. Newcomer

This is a somewhat trivial example of a program which transfers an ASCII file from the disk to the printer. We will assume in this example that the name of the file is "FILE.EXT". This program is basically similar to the program in example 5.

```
MODULE DSKLPT(STACK)=
BEGIN
%%
%
        THIS MODULE TRANSFERS ASCII FILES TO THE
      PRINTER.  FOR ILLUSTRATIVE PURPOSES HERE, THE INPUT
      FILE IS ALWAYS CALLED "FILE.EXT".
%
%%

EXTERNAL
        CLOSE, OPEN, LOOKUP, OUTMSG, READ, WRITE;

BIND
        INCH=1,                         % INPUT CHANNEL NUMBER %
        OUCH=2;                         % OUTPUT CHANNEL NUMBER %

OWN
        LBLOCK[4],
        T,
        OBUFH[3], IBUFH[3];

MACRO
        MSG(X)=OUTMSG(PLIT ASCIZ X)$,
        RESET=CALLI(0,0)$,
        HALT=CALLI(0,#12)$,
        STOP=CALLI(1,#12)$;

MACHOP
        CALLI=#047;

%%
%
        HERE IS THE MAIN PROGRAM
%
%%

        RESET;

UNTIL OPEN(OUCH,1, SIXBIT 'LPT', OBUFH+18) DO
        (MSG('?? NO LPT?M?J'); STOP);

UNTIL OPEN(INCH,1, SIXBIT 'DSK', IBUFH<0,0>) DO
        (MSG('?? NO DSK?M?J'); STOP);


        LBLOCK[0]←SIXBIT 'FILE';
        LBLOCK[1]←SIXBIT 'EXT';
        LBLOCK[2]←LBLOCK[3]←0;
```

```
        IF NOT LOOKUP(INCH,LBLOCK<0,0>)
                THEN
                (MSG('??FILE.EXT NOT FOUND?M?J');
                HALT);

        WHILE (T←READ(INCH)) GEQ 0 DO
                (IF .T NEQ 0 THEN
                        IF NOT WRITE(OUCH,.T)
                                THEN EXITLOOP(T←-3));

        CLOSE(INCH); CLOSE(OUCH);

        CASE .T+3 OF
                SET
                % -3 % MSG('??LPT ERROR?M?J');
                % -2 % MSG('??DSK ERROR?M?J');
                % -1 % MSG('DONE?M?J');
                TES;
END;
```

NOTES:

1. THE STACK declaration is used because this is the main program. Since no parameters are given, the stack is an OWN array, size 1000 words, with the name STACK.

2. Macros define the CALLI UUO's for RESET and EXIT ("STOP" and "HALT"). Note that the EXIT UUO is given two different names, depending on whether its accumulator field is zero or nonzero (see the DDP-10 monitor manual for the significance of this).

3. The STOP macro is used in the loops, which allows the user to issue an ASSIGN command and type CONT to resume if an error occurs.

4. EXITLOOP is used to terminate processing in the case of an output error. The loop condition is used to terminate processing in the case of an input error.

5. The CASE expression is used to output the appropriate message after the loop is terminated. Note the use of comments to show which value of I will execute a given expression within the CASE. This method of commenting CASE expressions makes them very readable.

6. The attributes SIXBIT and ASCIZ are used to define strings.

7. The escape convention (? character) is used in the messages to obtain control characters. Note the use of the double question mark (??) to obtain a single question mark.

8. Note the use of the MSG macro, which reduces a great deal of clutter in coding the messages. Compare the messages in this example with those of example 5.

APPENDIX A: SYNTAX

module → MODULE mname (parameters) e ELUDOM

mname → | name

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 | string | plit

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

string → stringtype quotedstring

stringtype → | ASCII | ASCIZ | SIXBIT | RADIX50

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 → xe:e

xe → e | ALWAYS | OTHERWISE

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 mname(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	0	
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
	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.

INTERFACE

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.

CARD CODES

Column Punch	Character	Octal
None	*Space*	0000
0	0	1000
1	1	0400
2	2	0200
3	3	0100
4	4	0040
5	5	0020
6	6	0010
7	7	0004
8	8	0002
9	9	0001
12 1	A	4400
12 2	B	4200
12 3	C	4100
12 4	D	4040
12 5	E	4020
12 6	F	4010
12 7	G	4004
12 8	H	4002

Column Punch	Character	Octal
12 9	I	4001
11 1	J	2400
11 2	K	2200
11 3	L	2100
11 4	M	2040
11 5	N	2020
11 6	O	2010
11 7	P	2004
11 8	Q	2002
11 9	R	2001
0 1	/	1400
0 2	S	1200
0 3	T	1100
0 4	U	1040
0 5	V	1020
0 6	W	1010
0 7	X	1004
0 8	Y	1002
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

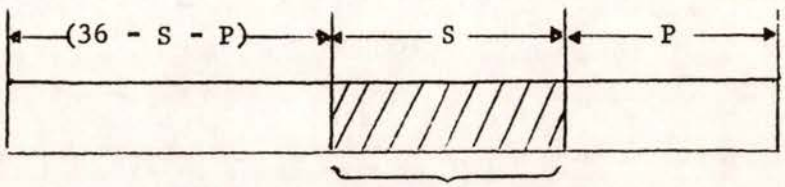
olumn Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12 8 7			|	'	↑	4006
11 8 2			!	:	!	2202
11 8 3	$	$	$	$	$	2102
11 8 4	*	*	*	*	*	2042
11 8 5			)	[	)	2022
11 8 6			;	>	;	2012
11 8 7			¬	&	\	2006
0 8 2			*See note*	;	]	1202
0 8 3	.	,	.	,	,	1102
0 8 4	%	(	%	(	%	1042
0 8 5			←	"	←	1022
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	primary precedes declarator
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: ∅ assumed
201	compile time (floating) division by zero
250	
300	
350	
400	
450	
500	input error while reading a source file (prints)
600	
613	module declaration within module body
614	invalid type in stack or timer declaration
615	syntax error in stack or timer declaration
616	invalid size expression in stack or timer declaration
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	MESSAGE
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)*

Carnegie-Mellon University
Department of Computer Science
Pittsburgh, Pennsylvania 15213

2a. REPORT SECURITY CLASSIFICATION

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	152	

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-70-C-0107	
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 (SRMA) 1400 Wilson Boulevard 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

DECUS
PROGRAM LIBRARY

DECUS NO.	10-177
TITLE	SIGN MAKER
AUTHOR	Irwin L. Goverman
COMPANY	Brandeis University Waltham, Massachusetts
DATE	November 13, 1972
SOURCE LANGUAGE	FORTRAN

BRANDEIS UNIVERSITY
WALTHAM, MASSACHUSETTS 02154

SIGN WRITING PROGRAM

Operating Instructions:

The program is loaded with the normal commands; 'R SIGN', if the program is on device SYS:, 'RUN SIGN' if it is in the user's area.

The program indicates its readiness to accept a command by typing '*'.

The information that the user wants to put in his sign can be input by either of two commands. The 'INLINE' command will accept one line from the user and then return to command mode. The 'INSIGN' command will keep looping back for more lines until either the user types a ↑Z (control-Z) or the tenth line has been input.

If an error is detected by the user after a line has been processed, he can replace the line by typing the command 'ERASE n', where n is the number of the line to be replaced. The user will be asked to type in the replacement line. The command 'WIPE' erases the entire current sign and resets the line counter.

The 'DEFSYM' command allows the user to define his own nonstandard symbol. Instructions are offered and then the user is asked to input a matrix of characters that will appear in his signs when he includes the symbol '@' (at sign) in a string input to 'INSIGN' or 'INLINE'.

The program contains its own help text and further instructions for using the 'DEFSYM' and 'INSIGN' commands.

The command 'PRINT' outputs the assembled sign, closes the output file and resets the line counter.

The user can exit the program by typing 'EXIT' or by typing ↑C (control-C) after the last 'PRINT' command. When 'EXIT' is used, a check is made to insure that there are no open signs as yet un'PRINT'ed and the user is informed as to how many signs he has created.

Output is in the form of files with the name 'SGNxx.DAT', where the xx is replaced with the number of the sign. If the user has set spool for device LPT:, the system will make up a name with the extension '.LPT'.

The two current limits imposed by the program are: 12 characters per line and 10 lines per sign.

Program Description:

The program accepts user input strings of up to 12 characters and accesses a random access file to create a centered line made up of 0.7" x 1.3" characters corresponding to the characters that are input. Each record of the random-access file contains the data that is used to generate one character of the full character set. In addition, the user can define his own non-standard symbols. The supplied symbol library contains the data for all the characters in the full FORTRAN character set and in addition, the backarrow and the square brackets.

As supplied, the program will look for the symbol library file 'SIGN.LIB' in the system area [1,4] , but this can be changed by recompiling the program with the variables 'PROJ' and 'PROG' set to other octal values. The file must have the proper protection so that the program can read it.

Output is in the form of automatically centered signs, made up of one or two pages. Each sign can contain up to ten lines of twelve characters each. The output can go directly to the printer, or be spooled for later printing.

```
      C     ********************************SIGN MAKER*************************
      C     WRITTEN BY IRWIN L. GOVERMAN  BRANDEIS '75
      C     USES RANDOM ACCESS FILE IN 1,4 CALLED "SIGN.LIB"
      C     PROGRAM ASSUMES UNIT#5 IS USER TELETYPE
      C     READS SYMBOL LIBRARY ON UNIT#1 AND WRITES SIGN
      C     ON DEVICE "LPT" WHICH IS NORMALLY 3.
      C
      C
      C
      C
      C
      C
      C
      C
      C
      C----------------MAIN PROGRAM--------------------------
      C
            DOUBLE PRECISION LINBF,BLANKS,NEWSYM
            DIMENSION LINBF(10,12,8),NEWSYM(8),
           1 IBUFF(13),ISYM(62), INDREM(10),ICMND(9)
            LOGICAL FFORM,ERASE,DEFLG,NOFLG
            DATA NEWSYM/8*'???????'/
      C
      C     THIS IS THE AREA THAT THE PROGRAM EXPECTS SIGN.LIB TO BE IN
            DATA PROJ/01/,PROG/04/
      C
      C     INITIALIZE COUNTER OF SIGNS AND THE FLAG THAT TELLS THAT A
      C     PRINT HAS JUST BEEN DONE
            DATA ISICNT/0/,IPRFLG/1/
      C
      C     THESE ARE THE AVAILABLE SYMBOLS, BACKARROW AND SQUARE
      C     BRACKETS ARE IN OCTAL BECAUSE THE COMPILER DOESN'T RECOGNIZE
      C     THEM.
            DATA ISYM/'A','B','C','D','E','F','G','H','I',
           1 'J','K','L','M','N','O','P','Q','R','S',
           2 'T','U','V','W','X','Y','Z',' ','0','1','2','3','4',
           3 '5','6','7','8','9','.',',','-','?','$','*',
           4 '=','&','<','>','+','/','!',''','(',')',
           5 '"','↑',057500402010Ø ,'%',';','[',']',
           6  055500402010Ø,056500402010Ø/
      C
      C     THIS IS THE LIST OF AVAILBLE COMMANDS.
            DATA ICMND/'ERASE','INLIN','H','PRINT','    ','EXIT','DEFSY',
           1 'INSIG','WIPE '/
      C
      C     THIS IS THE DEV, NO, THAT THE SIGNS ARE WRITTEN ON.
            LPT=3
      C
            DATA BLANKS/'          '/
            DATA FFORM,ERASE,DEFLG,NOFLG/4*.FALSE./
      C
      C     OPEN UP THE LIBRARY OF SYMBOLS
```

```
                      CALL DEFINE FILE(1,56,NERD,'SIGN.LIB',PROJ,PROG)
                C
                C     NOTIFY USER THAT HELP IS AVAILABLE
                      WRITE(5,2)
              2       FORMAT(' TYPE "H" FOR HELP',/)
                C
                C     THIS IS WHERE THE INTERACTIVE PART STARTS, INDICATE READINESS
                C     FOR COMMAND BY TYPING A STAR, AND WAIT.
              1       WRITE (5,3)
              3       FORMAT (' *',$)
                C
                C     ACCEPT A COMMAND
                      READ(5,4)INDO
              4       FORMAT(A5)
                C
                C     CHECK FOR BLANK, NO NEED TO SEARCH COMMAND LIST,
                      IF(INDO,EQ,'    ')GO TO 1
                C
                C     CHECK THE COMMAND INPUT AGAINST VALID COMMANDS.
                      DO 5 J=1,9
              5       IF(INDO,EQ,ICMND(J)) GO TO 6
                C
                C     NOT A RECOGNIZED COMMAND
                      WRITE(5,7)INDO
              7       FORMAT(' ?NOT A COMMAND: ',A5)
                      WRITE(5,2)
                C
                C     LOOP BACK FOR ANOTHER COMMAND
                      GO TO 1
                C
                C     JUMP TO SPECIFIED PLACE
              6       GO TO (100,200,300,400,900,600,700,800,1000),J
                C
                C
                C
                C
                C
                C     ***ERASE***
                C
                C     CHECK TO SEE IF THERE IS AN UNPRINTED SIGN OPEN, AND
                C     THAT IT CONTAINS DATA (LINCNT NOT EQUAL TO ZERO)
            100 IF(LINCNT,NE,0,AND,IPRFLG,EQ,0)GO TO 101
                      WRITE(5,103)
            103 FORMAT(' ?NO SIGN IN CORE',/)
                C
                C     LOOP BACK FOR ANOTHER COMMAND
                      GO TO 1
                C
                C     DID HE INCLUDE NUMBER IN COMMAND STRING?
            101 REREAD 113,INDO
                      IF(INDO,NE,0)GO TO 117
            113 FORMAT(5X,I)
                C
```

```
      C     NO, GET IT NOW
            WRITE(5,102)
102   FORMAT(' WHAT LINE NUMBER? ',$)
            READ (5,106)INDO
106   FORMAT(I)
      C
      C     CHECK FOR EXISTENCE OF SUCH A LINE
117         IF(INDO.LE.0)GO TO 111
            IF(LINCNT-INDO)111,120,120
111   WRITE(5,112)INDO
112   FORMAT(' ?LINE #',I2,' DOES NOT EXIST')
      C
      C     LOOP BACK FOR ANOTHER COMMAND
            GO TO 1
      C
      C     STORE CURRENT VALUE OF LINCNT IN TEMP, JUMP TO INLINE
      C     TO REPLACE LINE, ERASE IS FLAG TO RETURN HERE
120         ITEMP=LINCNT
            WRITE(5,126)
126   FORMAT('+(NEW) ',$)
            LINCNT=INDO-1
            ERASE=.TRUE.
            GO TO 200
      C
      C     JUMP BACK HERE AND RESTORE EVERYTHING TO ORIGINAL STATE
125         LINCNT=ITEMP
            ERASE=.FALSE.
      C
      C     LOOP BACK FOR ANOTHER COMMAND
            GO TO 1
      C
      C
      C
      C
      C
      C
      C     ***INLINE***
      C
      C     INCREMENT LINCNT
200         LINCNT=LINCNT+1
      C
      C     CHECK AGAINST THE MAXIMUM NO. OF LINES.
            IF(LINCNT.LE.10)GO TO 297
            WRITE(5,242)
242   FORMAT(' ?TOO MANY LINES- "PRINT" THE SIGN.',/)
            GO TO 298
      C
      C     INITIALIZE CHARACTER COUNTER
297         ICRCNT=0
      C
      C     CHECK TO SEE IF WE HAVE TO JUMP TO FILE-OPEN SECTION.
      C     IF IPRFLG=1 THEN A PRINT HAS BEEN DONE AND WE MUST OPEN
            IF(IPRFLG)202,203,202
```

```
160             C
161             C       SET IRET SO WE RETURN HERE AFTER THE OPENING.
162             202     IRET=1
163             C
164             C       JUMP TO OPENING PROCEDURE.
165                     GO TO 500
166             C
167             C       IF THIS IS AN INSIGN, USE THE SECOND OUTPUT FORMAT.
168             203     IF(FFORM)GO TO 280
169                     WRITE(5,204)ISIGNT,LINCNT
170             204     FORMAT(!+SIGN:'I2,'  LINE:',I2,' STRING:',$)
171                     GO TO 283
172             280     WRITE(5,855)LINCNT
173             C
174             C       READ THE INPUT STRING, CHECK FOR ↑Z (EOF)
175             C       THE EOF MEANS THAT THE USER HAS FINISHED INPUT TO INSIGN.
176             283     READ(5,205,END=298,ERR=298)IBUFF
177             205     FORMAT(13A1)
178             C
179             C       CHECK FOR STRING DELIMITER
180                     IF (IBUFF(13).EQ.' ')GO TO 276
181             206     WRITE(5,207)
182             207     FORMAT(' ?TOO MANY CHARACTERS --LINE NOT COMPILED')
183             C
184             C       ERROR TRAP FOR INLINE AND INSIGN
185             C       CLEARS LOGICAL FLAGS ,DECREMENTS LINCNT, GOES TO COMMAND MODE
186             C       EXCEPT IF WE CAME FROM ERASE (WE HAVE TO RESTORE LINCNT)
187             298     IF(ERASE)GO TO 125
188                     LINCNT=LINCNT-1
189                     FFORM=.FALSE.
190             C
191             C       LOOP BACK FOR ANOTHER COMMAND
192                     GO TO 1
193             C
194             C       GET ACTUAL LENGTH OF LINE
195             276     DO 270 KK=13,1,-1
196                     IF(IBUFF(KK).NE.' ')GO TO 290
197             270     CONTINUE
198                     KK=1
199             C
200             C       COMPUTE THE AMOUNT OF INDENTATION NESS.
201             290     INDENT=(12-KK)/2
202             C
203             C       SET THE INDENT VARIABLE FOR THIS LINE (1=ODD,0=EVEN # OF CHARS.
204                     INDREM(LINCNT)=0
205                     IF(KK.NE. (KK/2)*2) INDREM(LINCNT)=1
206             C
207             C       CHECK TO SEE IF LAST CHARACTER HAS BEEN PROCESSED
208             271     IF(ICRCNT-KK)272,230,230
209             C
210             C       INCREMENT CHARACTER COUNTER
211             272     ICRCNT=ICRCNT+1
212             C
```

```
                C       CALCULATE THE POSITION IN THE BUFFER THAT THIS CHARACTER
                C       SHOULD GO INTO.
                        IPOS=INDENT+ICRCNT
                C
                C       CHECK FOR USER DEFINED SYMBOL
                        IF(IBUFF(ICRCNT).EQ.'@')GO TO 750
                C
                C       CHECK FOR VALID CHARACTER AND GET INDEX
                        DO 210 J=1,62
            211 210     IF(IBUFF(ICRCNT).EQ.ISYM(J)) GO TO 214
                        WRITE(5,211)IBUFF(ICRCNT)
            211         FORMAT(' ?NOT A RECOGNIZED SYMBOL: 'A2,'  TRY AGAIN')
                        GO TO 298
                C
                C       GET THAT CHARACTER'S DATA FROM SYMBOL LIBRARY
            214         READ(1#J,215,END=240,ERR=240)(LINBF(LINCNT,IPOS,I),I=1,8)
            215         FORMAT(8A7)
                C
                C       LOOP BACK FOR ANOTHER CHARACTER
                        GO TO 271
                C
                C       BLANK OUT THE PARTS OF THE BUFFER NOT USED TO HOLD CHARACTERS
            237         IF(INDENT)252,252,249
            249         DO 250 KL=1,INDENT
                        DO 250 KM=1,8
            250         LINBF(LINCNT,KL,KM)= BLANKS
                C
                C       CHECK IF THERE IS BLANK SPACE AT END OF LINE
            252         IF(12-(INDENT+ICRCNT))262,262,256
            256         DO 258 KL=INDENT+ICRCNT+1,12
                        DO 258 KM=1,8
                C
                C       USE NULLS INSTEAD OF BLANKS TO CONSERVE SPACE.
            258         LINBF(LINCNT,KL,KM)= 0
                C
                C       IF FASTFORM (INSIGN)AND MAX. NUMBER OF LINES IS NOT
                C       EXCEEDED GO BACK FOR ANOTHER LINE.
            262         IF((FFORM).AND.(LINCNT.LT.10))GO TO 200
                C
                C       CLEAR THE INSIGN FLAG SO ERASES AND INLINES WONT LOOP
                        FFORM=.FALSE.
                C
                C       JUMP BACK TO ERASE IF WE CAME FROM THERE
                        IF(ERASE)GO TO 125
                C
                C       LOOP BACK FOR ANOTHER COMMAND
                        GO TO 1
                C
                C       ERROR TRAP FOR TROUBLE READING SYMBOL LIBRARY.
            240         WRITE(5,241)
            241         FORMAT(' ?CANNOT ACCESS SYMBOL DATA FILE SYS:SIGN.LIB ')
                C
                C       LOOP BACK FOR ANOTHER COMMAND
```

```
                      GO TO 1
                C
                C
                C
                C
                C
                C     ***H***
                C     THIS IS THE HELP TEXT.
           300  WRITE(5,301)
           301  FORMAT(' SIGN MAKER  V.4',/,' COMMANDS:'/,
                2' INLINE-TYPED TO INPUT EACH LINE OF A SIGN',/,
                3' INSIGN- TYPED TO INPUT A WHOLE SIGN, RATHER THAN',/,
                3'     A LINE AT A TIME.',/,
                4' PRINT-PRINTS ASSEMBLED SIGN AND CLOSES FILE',/,
                5' DEFSYM-LETS USER DEFINE A CHARACTER OR SYMBOL',/,
                7' ERASE N- TO REPLACE THE NTH LINE OF A SIGN',/,
                7' WIPE- ZEROES OUT THE SIGN IN CORE AND RESETS LINE-COUNTER',/
                8' H-THIS TEXT',/,
                9' EXIT -TO EXIT FROM THE PROGRAM',/,
                1' PARAMETERS:',/,
                2' 12 CHARS/LINE 10 LINES/SIGN',/,
                3'  THE  USER DEFINED SYMBOL IS',/,
                4' INCLUDED IN A SIGN BY TYPE THE CHARACTER "@" IN ITS',/,
                5' PLACE WHEN YOU INPUT A STRING TO "INLINE"',/,
                6' OR "INSIGN" AFTER USING "DEFSYM"'/,
                7' FURTHER INSTRUCTIONS FOR USE OF "INSIGN " AND "DEFSYM"',/,
                8' WILL BE OFFERED AT APPROPRIATE TIMES.',/,
                7' ALL SIGNS ARE CENTERED AUTOMATICALLY'/)
                C
                      WRITE(5,350)
           350  FORMAT(' DO YOU WISH TO SEE CHARACTER SET?(Y OR N)',$)
                      READ(5,302)IANSW
           302  FORMAT(A1)
                C
                C     LOOP BACK FOR ANOTHER COMMAND
                      IF(IANSW.EQ.'N') GO TO 1
                      WRITE(5,303)ISYM
           303  FORMAT(1X,30A2,/,1X,32A2,/)
                C
                C     LOOP BACK FOR ANOTHER COMMAND
                      GO TO 1
                C
                C
                C
                C
                C
                C     ***PRINT***
                C
                C     CHECK TO SEE IF THERE IS A SIGN OPEN (IPRFLG=0)
           400  IF(IPRFLG)401,401,433
           433  WRITE(5,444)
           444  FORMAT(' ?NOTHING TO PRINT',/)
                C
```

```
                C       LOOP BACK FOR ANOTHER COMMAND
                        GO TO 1
                C
                C       SKIP LINES ON PAGE TO CENTER SIGN
            401         ISKIP=(60-(LINCNT*12))/2
                C
                C       SPACING, USED TO ADJUST SO THAT CHARRIAGE CONTROL IS NOT
                C       NESS.
                        WRITE(LPT,404)
                        WRITE(LPT,404)
                        IF (ISKIP)410,410,402
            402         DO 403 KK=1,ISKIP-2
            403         WRITE(LPT,404)
            404         FORMAT(' ')
                C
                C       ACTUAL WRITING TAKES PLACE
            410         DO 499 J=1,LINCNT
                C
                C       CHECK FOR ODD NUMBER OF CHARACTERS (USE SECOND FORMAT)
                        IF(INDREM(J))415,415,450
            415         WRITE(LPT,408)((LINBF(J,K,I),K=1,12),I=1,8)
            408         FORMAT(8(' ',7X,12(A7,3X)/)/)
                C
                C       NO SKIP AFTER FIFTH LINE BECAUSE PAGE THROW DOES THAT
                        IF(J.NE.5)WRITE(LPT,482)
            482         FORMAT(' ',/,' ')
                        GO TO 499
            450         WRITE(LPT,455)((LINBF(J,K,I),K=1,12),I=1,8)
                C
                C       THIS IS THE SECOND FORMAT
            455         FORMAT(8(' ',11X,12(A7,3X)/)/)
                        IF(J.NE.5)WRITE(LPT,482)
            499         CONTINUE
                C
                C       CLOSE THE FILE
                C
                C       THIS RELEASE IS DONE BECAUSE USER MAY CONTROL C OUT
                C       OF PROGRAM, AND AN END-FILE LOSES THE LAST BIT OF THE BUFFE
                        CALL RELEAS(LPT)
                C
                C       INITIALIZE LINE COUNTER
                        LINCNT=0
                C
                C       SET IPRFLG TO INDICATE THAT A PRINT WAS JUST DONE
                        IPRFLG=1
                C
                C       LOOP BACK FOR ANOTHER COMMAND
                        GO TO 1
                C
                C
                C
                C
                C
```

```
372             C
373             C-----OPENING PROCEDURE USED BY BOTH INLINE AND INSIGN
374             C-----WHEN THERE IS NO OPEN FILE (I.E. A PRINT WAS DONE)
375             C
376             C       INCREMENT SIGN COUNTER BY 1
377             500     ISICNT=ISICNT+1
378             C
379             C       CLEAR IPRFLG TO INDICATE THERE IS AN OPEN SIGN
380             508     IPRFLG=0
381             C
382             C       MAKE A NAME FOR THE FILE.
383                     ENCODE(5,504,INAME)ISICNT
384             504     FORMAT('SGN',I2)
385             C
386             C       OPEN UP THE FILE
387                     CALL OFILE(LPT,INAME)
388             C
389             C       JUMP BACK TO INLINE OR INSIGN.
390                     GO TO (203,805),IRET
391             C
392             C
393             C
394             C
395             C
396             C
397             C
398             C       ***EXIT***
399             C
400             C       CHECK FOR OPEN SIGNS
401             600     IF(IPRFLG.EQ.1.OR. LINCNT.EQ.0)GO TO 605
402                     WRITE(5,537)ISICNT
403             537     FORMAT(' ?YOU HAVE NOT PRINTED SIGN',I3,' YET!')
404             C
405             C       LOOP BACK FOR ANOTHER COMMAND
406                     GO TO 1
407             605     WRITE(5,601)ISICNT
408             601     FORMAT(' TOTAL OF ',I3,' SIGNS CREATED')
409                     STOP
410             C
411             C
412             C
413             C
414             C
415             C
416             C
417             C       ***DEFSYM***
418             C
419             C       ASK IF THEY WANT INSTRUCTIONS
420             700     WRITE(5,804)
421                     READ(5,4)INDO
422                     IF(INDO.EQ.'N')GO TO 780
423                     WRITE(5,710)
424             710     FORMAT(' EACH SYMBOL IS 7 CHARACTERS ACROSS AND 8',/,
```

```
425                     1 ' CHARACTERS HIGH,  YOUR DEFINED SYMBOL IS CALLED BY',/,
426                     2 ' TYPING THE CHARACTER "@" IN ITS PLACE WHEN YOU',/,
427                     3 ' INPUT A STRING TO THE COMMAND "INLINE", NOW TYPE IN A',/
428                     4 ' ROW OF YOUR SYMBOL AFTER EACH "↑" APPEARS.',/,
429                     5 ' (<CR> AFTER EACH ROW)')
430           780       WRITE(5,785)
431           785       FORMAT('   1234567',/)
432           C
433           C         GET THE USER DEFINED SYMBOL FROM THE TTY
434                     DO 725 KL=1,8
435                     WRITE(5,713)KL
436           713       FORMAT(!↑',I1,'↑',$)
437           725       READ(5,715)NEWSYM(KL)
438           715       FORMAT(A7)
439                     WRITE(5,730)
440           730       FORMAT(' NEW SYMBOL DEFINED'/)
441           C
442           C         SET DEFLG TO INDICATE THAT THERE IS A USER DEFINED SYMBOL
443                     DEFLG=.TRUE.
444           C
445           C         LOOP BACK FOR ANOTHER COMMAND
446                     GO TO 1
447           C
448           C         INSIGN AND INLINE JUMP HERE WHEN '@' IS FOUND IN INPUT STRI
449           C
450           C         CHECK IF THERE IS A USER DEFINED SYMBOL
451           750       IF(.NOT.DEFLG)GO TO 790
452           C
453           C         TRANSFER USER SYMBOL TO CHARACTER BUFFER
454                     DO 775 J=1,8
455           775       LINBF(LINCNT,IPOS,J)=NEWSYM(J)
456           C
457           C         RETURN TO INLINE PORTION.
458                     GO TO 271
459           790       WRITE(5,793)
460           793       FORMAT(' "@" MUST NOT APPEAR IN AN INPUT STRING UNTIL',/,
461                    1 ' YOU HAVE GONE THRU DEFSYM DIALOG TO DEFINE THE SYMBOL',/
462                    2 ' THE LINE HAS NOT BEEN COMPILED,'/)
463                     GO TO 298
464           C
465           C
466           C
467           C
468           C
469           C         ***INSIGN***
470           C
471           C         IF A PRINT WAS JUST DONE JUMP TO OPENING ROUTINE.
472           800       IF(IPRELG)803,805,803
473           803       IRET=2
474                     GO TO 500
475           C
476           C         CHECK IF USER HAS SAID HE DOESN'T WANT INSRUCTIONS
477           805       IF(NOFLG)GO TO 850
```

```
                      WRITE(5,804)
804                   FORMAT(' INSTRUCTIONS? (Y OR N):',$)
                      READ(5,806)INDO
806                   FORMAT(A1)
                      WRITE(5,801)
801                   FORMAT(' ')
                      IF(INDO.EQ.'N')GO TO 840
                      WRITE(5,810)
810                   FORMAT(' FAST FORM OF INLINE COMMAND',/,
                     1' KEEPS YOU IN INLINE MODE. TYPE IN A LINE OF THE',/,
                     2' SIGN AFTER EACH "LINE NO.:" APPEARS, TO RETURN TO',/,
                     4' COMMAND LEVEL, TYPE  A ↑Z, ANY ERROR RETURNS',/,
                     5' YOU TO COMMAND MODE . THERE IS AN AUTOMATIC',/,
                     1' RETURN TO COMMAND MODE AFTER THE',/,
                     2' TENTH LINE OF A SIGN IS INPUT,'//)
C
C     SET FLAG TO INDICATE THAT USER HAS EITHER HAD INSTRUCTION
C     OR DOESN'T WANT IT
840   NOFLG=.TRUE.
C
C     SET FAST FORM FLAG FOR INSIGN TO USE INLINE
850   FFORM=.TRUE.
C
C     FORMAT USED BY INLINE WHEN FFORM IS SET
855   FORMAT('+LINE ',I2,'! ',$)
C
C     JUMP FROM HERE INTO THE INLINE PORTION.
                      GO TO 200
C
C
C
C
C
C     ***FOR EXPANSION***
900   CONTINUE
C
C     LOOP BACK FOR ANOTHER COMMAND
                      GO TO 1
C
C
C
C
C
C
C     *** WIPE ***
1000  LINCNT=0
C
C     LOOP BACK FOR ANOTHER COMMAND
                      GO TO 1
                      END
```

CONSTANTS

DECUS
PROGRAM LIBRARY

DECUS NO.	10-107
TITLE	CFILE
AUTHOR	Walter Metcalf
COMPANY	Submitted by: Kay Latven Brookings Institution Washington, D.C.
DATE	December 7, 1970
SOURCE LANGUAGE	MACRO-10

ATTENTION

This is a USER program. Other than requiring that it conform to submittal and review standards, no quality control has been imposed upon this program by DECUS.

The DECUS Program Library is a clearing house only; it does not generate or test programs. No warranty, express or implied, is made by the contributor, Digital Equipment Computer Users Society or Digital Equipment Corporation as to the accuracy or functioning of the program or related material, and no responsibility is assumed by these parties in connection therewith.

DECUS Program Library Write-up DECUS No. 10-107

CFILE will run as a regular CUSP under the 4S72 Monitor with no monitor modifications required if the assembly parameter DEBSW is set nonzero. In this case the argument is typed following the asterisk that CFILE responds with when called, as follows:

```
.R CFILE
*dev:filename.ext [ proj,prog ]
```

If it is desired to implement CFILE as a monitor command, then the assembly parameter DEBSW must be set to zero and the monitor modified to accept the command "CFILE". (This can be avoided by setting the assembly switch HELPSW non-zero and naming the file HELP.SHR, in which case CFILE can be invoked by the HELP command.) In addition, if the monitor and LOGIN are modified so that (a.) a CFILE command may be given without logging in (the job is logged in under [1, 2]), and (b.) remote logins are permitted over a PTY, then CFILE will deposit the transaction file in disk area [1, 2] and can be used for initializing jobs such as PRINTR, CHKPNT, and OPFILE when the system is brought up. Under these circumstances, CFILE checks to make sure it is running from the CTY and, if not, exits, telling the user to login so that security is maintained. The exit is by means of the LOGOUT UUO so that the job number is freed up.

Program Name: CFILE

How to Call: After logging in, type

```
. CFILE dev:file.ext [ Proj,Prog ]
```

as a Monitor command. Each of the subfields has default valve which is assumed if the field is omitted. The defaults are:

dev	-	DSK
file	-	OBJECT
ext	-	(null)
[Proj,Prog]	-	0, meaning the PPN under which the user is currently logged in.

The file specified by these parameters is called the command file.

Purpose:

CFILE was written to allow users to create a file (the command file) containing various "lines of data" that he would normally type on the teletype, and then to specify at some later time that these "lines of data", i.e., commands, be processed. This permits a quasi-batch mode of running certain programs whose need for interaction is limited. In addition,

it permits certain frequently executed procedures to be saved and collected, so that the user may run them when the need arises without having to type in all the commands each time.

Procedures executed by means of CFILE are initiated as separate jobs. Therefore the first three commands of a command file will normally be:

1. The word LOGIN
2. The user's project programmer number
3. The user's password

An example of a simple command file follows:

The command file, called RUNJOB, consists of the following:

```
LOGIN
615,122
PASSWORD
ASSIGN MTAØ DEV1
ASSIGN MTA1 DEV2
ASSIGN DSK  DEV3
ASSIGN DSK  DEV4
EXECUTE CONSIS
R PIP
DSK:CONSIS.LST/P<—DSK:FOR 21.DAT,FOR22.DAT,FOR27.DAT,
    FOR28.DAT
↑CR PRINT
CONSIS.LST
↑CKJOB
```

This example assigns all the devices needed to run the program "CONSIS", executes the program, and prints the output. To run this command file, the user types the following on his teletype:

```
.LOGIN↵

#615,122↵

PASSWORD:  PASSWORD↵

.CFILE RUNJOB

EXIT

↑C

. KJOB
```

Method of Operation:

CFILE operates by initializing an entity called a pseudo teletype, (PTY). A PTY is a non-sharable device in the Monitor and has all the properties common to other non-sharable devices (line printer (LPT), DECtape (DTA), card reader (CDR), etc.,) except that there is no corresponding piece of hardware. In particular a PTY has the properties of a teletype (TTY) in that jobs may be logged in via a PTY, and all the regular communication that takes place between a user and his TTY can take place between CFILE and a PTY. In effect, then, CFILE runs jobs using the PTY.

After initializing a PTY (the first available one is used), the characters in the command file are "typed" on the PTY, and the response from the computer, if any, is read. In this way CFILE is capable of setting up jobs to do completely arbitrary tasks. Any command 1/ that a user can type on a TTY can be included in the command file for CFILE to "type" on its PTY.

As CFILE is running the job on the PTY (this job is called the object job), it puts the commands from the command file, along with the response to these commands in a file called nnnCFI.TXT

where nnn is job number under which CFILE is logged in (not the job number of the object job).

nnnCFI.TXT is created in the user's own disk area.

When the object job is completed (i.e., the command file is exhausted), CFILE types:

EXIT

↑C

.

on the user's teletype.

At this time, the user may type the file nnnCFI.TXT (called the transaction file) to get a complete record of what took place on the PTY.

If CFILE finds a "?" in the response from the computer to a command, the message

?ERROR FOUND IN OBJECT JOB

is typed on the user's console, but the job continues to run. In general, it is up to the user to decide whether an error is acceptable and let the job run or an error is not acceptable and abort the run.

1/ See Restriction on Commands, page 4 of this bulletin.

In any event, if it should ever become necessary to abort CFILE (by typing two ↑ C's) before it has finished and typed

EXIT

↑ C

on the user's teletype, then the user must ALWAYS follow immediately with a REENTER command. Simply type:

.REENTER↓

After about 30-60 seconds, CFILE will type:

EXIT

↑ C

on the user's teletype. This is necessary for two reasons:

1. The object code is still running after typing the ↑ C's and the REENTER command causes CFILE to KJOB the object job.

2. The transaction file (nnnCFI.TXT) is still open and not available for reading, the REENTER command causes the transaction file to be closed and thus the user can type it out to see what, if anything, went wrong.

Restriction On Commands:

There is one class of programs and CUSP's that cannot be run under control of CFILE (i.e., commands to execute them cannot be included in a command file). This class includes any MACRO-10 program which does input from the user's teletype solely by means of the following TTCALL UUO's (see page 443 of PDP-10 Reference Handbook):

INCHRS

INCHSL

It is permissible to use these UUO's in conjunction with an INCHRW or INCHWL UUO, but they may not be used as the only input commands to a user console.

The PLEASE command is currently the only CUSP that cannot be included in the command file. All other CUSPS, and all FORTRAN programs, may be used freely with CFILE.

If anyone is writing a MACRO-10 program that he wants to be able to use with CFILE and is in doubt about whether he is violating the above restriction, he should contact Walter Metcalf at the Computation Center, Brookings Institution, 483-8919, extension 407.

It is hoped that this restriction will be removed at a future date.

NOTES:

1. If a user wishes to enter a ↑ C in command file, he may do so by typing $$31$ to TECO. TECO is presently the only means by which ↑ C's can be entered into a command file. (Page 508 of PDP-10 Reference Handbook).

2. Since command files will normally contain the user's password, it is advisable to change the protection of the command file to <477> which is read protected against all other users. The instructions to do this are:

```
.R PIP

*DSK:/R*.*<477> <—CMD ↵

* ↑ C
```

where CMD is the name of the command file.

DECUS
PROGRAM LIBRARY

DECUS NO.	10-142
TITLE	MATHLAB
AUTHOR	Carl Engelman
COMPANY	The MITRE Corporation Bedford, Massachusetts
DATE	April 23, 1971
SOURCE LANGUAGE	LISP

ACKNOWLEDGEMENTS

This project was sponsored by The MITRE Corporation's Independent Research Committee.

It was also supported, in part, through access to its computer facilities, by Project MAC, an M. I. T. Research Program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract No. NONR-4102(01).

The MITRE Corporation employees who, in addition to the current author, have contributed to MATHLAB are S. Bloom, B. W. Diffie, L. Ernst, M. Manove, J. Millen, and M. Waters.

The simplification program is due, for the most part, to K. Korsvold of the Stanford Artificial Intelligence Project, more recently of the Norwegian Defense Research Establishment. Parts of the user level routines within the matrix package were programmed by P. Wang of M. I. T.

The system was originally developed within the LISP 1.5 system of CTSS and then within the LISP system of the PDP-6/10 ITS timesharing system of the Artificial Intelligence Group of M. I. T., now the M. I. T. Artificial Intelligence Laboratory.

The current DECUS release of MATHLAB was also supported, in part, through access to its computer facilities, by the Digital

Equipment Corporation. Special thanks are due A. D. Grayson of that corporation for his day-to-day support.

This version is built within the DECUS release of LISP 1.6 due to the Stanford Artificial Intelligence Project. The LISP compiler was contributed by B. W. Diffie of that Project.

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Cont.)

TABLE OF CONTENTS (Conc.)

SECTION I

INTRODUCTION

MATHLAB is an on-line system providing machine aid for the mechanical symbolic processes encountered in analysis. It is capable of performing, automatically and symbolically, such common procedures as simplification, substitution, differentiation, polynomial factorization, indefinite integration, direct and inverse Laplace transforms, the solution of linear differential equations with constant coefficients, the solution of simultaneous linear equations, and the inversion of matrices. It also supplies fairly elaborate bookkeeping facilities appropriate to its on-line operation.

The formal name of the system is MATHLAB 68, and it is so referred to in references [1] and [2]. We shall for the remainder of this manual refer to it simply as MATHLAB. The first of these references is a very brief introduction to our objectives; the second a case study of the key decisions made during the design of the system and an analysis of their consequences. This latter paper is highly recommended to MATHLAB users who would like some idea of what is happening to them and why. There are a few discrepancies between that paper and this manual, e.g., they disagree on how large a MATHLAB integer might be. In such disputes, the manual prevails.

Unlike the above papers, the current manual is meant simply as a detailed set of instructions on how to use MATHLAB. Before delving

into such details, we should like to point out that the system must be regarded as experimental, not because its answers are unreliable, but because the emphasis has been primarily on its symbolic and interactive aspects to the neglect of such practical requirements as numerical computation or programming features. These deficiencies are not absolute, but the facilities provided are often inadequate.

As for the reliability of the answers, we are, of course, offering the program with no warranty, neither explicit nor implicit. Nonetheless, MATHLAB has certainly proved capable of large, accurate computation. On several occasions, it has differed with published results with the discrepancy being resolved in MATHLAB's favor.

This manual also contains instructions on how to construct MATHLAB from its source code.

SECTION II

SYNTAX

We shall try to present here a semi-formal definition of the syntax of MATHLAB. We are doing this, as opposed to writing a formal, e.g., BNF or context-free grammar, definition for several reasons including our lack of appetite for the formal task and our doubt that it would be of much use to someone learning how to use MATHLAB. There is a danger that even this semi-formal definition is too frightening an introduction. It is presented as a logical necessity, but the beginner should bear in mind that it is not difficult to ask MATHLAB to execute some common algorithm and that the input is in a fairly natural algebraic formalism. For example, if the user wanted to know the integral, with respect to x, of

$$X + \frac{3X^3 - C}{X + 1}$$

he need only wait for MATHLAB to type

#

as an indication that it is listening, then type himself

'INTEGRATE(X+(3*X↑3-C)/(X+1),X)

followed by the terminal character of MATHLAB which is the key, labeled on a teletype ALT-MODE, PRE, or ESC. The computer would then respond

$$X^3 - X^2 + 3X + (- C - 3)L0G(X + 1)$$

which is the answer.

A CONVERSATION is a sequence of inputs, by the user, followed by outputs by the machine (exception: MATHLAB occasionally asks questions to which the user responds).

INPUT: an instruction or command.

OUTPUT: An instruction yields for output a two-dimensional display of an evaluated mathematical expression. A command might be, for example, a request for information (such as an inventory of the names of all stored expressions) and the output a statement of relevant information. Commands are discussed in Section VI.

The machine always types a # to indicate it is listening and the user always hits the ALT-MODE or PRE or ESC key to signify that his input is complete. We shall, from now on, denote that key by $, since that is how MATHLAB -- really, the time-sharing system -- echoes it to the user.

INSTRUCTION

We shall build up the definition of an instruction slowly.

NUMBER: A non-negative integer; may be of any size. E.g., the number of ways of shuffling a deck of cards is

80658175170943878571660636856403766975289505440883277824000000000000

All arithmetic in MATHLAB is infinite precision rational.

FORMAL VARIABLE: a 'word' or 'atom' consisting of a string of letters or numerals, the first of which is not a numeral.

ALGEBRAIC OPERATOR: + - * / ↑ **

The last two denote exponentiation. Precedences are conventional. e↑x↑2 denotes e↑(x↑2). " - " is either unary or binary.

PARENTHESES: "(" and ")" are used to denote precedence as in a*(b+c) and, with commas, to construct a

FUNCTIONAL FORM: something of the form f(x) or f(x, y, z). The rule for forming the function name "f" is the same as that for a formal variable.

EXPRESSION: any ordinary combination of the above. E. g., x ↑2*sin(x+y) .

EQUATION: a = b, where a and b are expressions.

FUNCTION: An expression-valued function of a finite number of arguments, e.g., f: (x, y, z) → x + y + z. This is what a function means but, unfortunately, not what we type. We would type the definition of the above function as: f(x, y, z): x + y + z. In addition to user-defined functions, there are a number of system-defined functions discussed in Section V.

DATUM: an expression, equation, or function.

ASSIGNMENT STATEMENT: The above function definition is an exceptional case of an assignment statement. The assignment operator in MATHLAB is the colon. It may be used internally in an instruction, its scope being as far as it continues to make sense. E. g., (v:x+y)↑3 will evaluate to $(x + y)^3$ but will also store an expression whose name is v and whose value is x + y. Data storage will be discussed in Section IV.

EVALUATION: Unless an input symbol is preceded by the mark ', that symbol will evaluate to itself. However, any formal variable preceded by ' is interpreted as a name and its value is substituted.

If

#V: X + Y$

is followed by

#'V ↑ 2 + 'V + A$.

the computer will respond

$(X + Y)^2 + X + Y + A.$

The mark ', read as evaluate or unquote, has two meanings when applied to functions. If f and g are functions, the expression 'f + 'g has a meaning which is discussed in Section III. If, as is more usual, ' appears before the name of a function which appears in a functional form, it causes the execution of the algorithm represented by the function.

The following example is an attempt to show the control a MATHLAB user can exercise by judicious use of the evaluation sign. There is a system-defined function named DERIV which, we may assume for our current purposes, takes two arguments and differentiates the first with respect to the second. Let us suppose that we have previously typed the assignment statement:

#Y:SIN(X)$,

and that, further, we have not declared, cf. Section VII, Y dependent on X. Then:

DERIV(Y,X)	evaluates to	$\frac{DY}{DX}$	;
DERIV('Y,X)	evaluates to	$\frac{D}{DX}$SIN(X)	;
'DERIV(Y,X)	evaluates to	0	;
'DERIV('Y,X)	evaluates to	COS(X)	.

Evaluation may be used to any depth. If v:x and x:5, then ''v evaluates to 5.

User defined functions can only be evaluated with expressions as arguments. So, for example, if f and g are functions, 'f('g) is not allowed but 'f('g(1)) is. System-defined functions are subject to individual rules to be found in Section V.

INSTRUCTION: A legal combination of the above items followed by $. We shall not define legal. In general, common sense prevails but the rules of Section III should be noted.

Spaces and carriage-returns inserted between the components of an instruction are ignored. There is an additional syntactic element, the double-quote mark: ". Discussion of its meaning will be postponed until Section X_b.

SECTION III

RULES FOR ALGEBRAIC OPERATORS

We shall state the rules for + but they apply uniformly to all the operators.

1. The sum of two data of the same type is of the same type.

1a. Expressions are added by concatenating them with a + set between them. However, if SIMP is turned on, cf. Section VI, anything might happen, e.g., terms might combine or cancel.

1b. Equations are summed by equating the sum of the left sides to the sum of the right sides.

1c. One cannot sum system-defined functions. We shall, however, see (cf. Section X_b) that there are other ways of creating a function which is the sum of system functions.

In order to sum user-defined functions, they must have the same number of arguments. The dummy variables of the sum of several functions will normally be chosen by MATHLAB to be the dummy variables of the first of the functions being added in the same order as they appear in its definition. However, should one of these be used as a parameter (not a dummy) by another of the functions being added, its name, as a dummy, will be changed to one generated by the program. MATHLAB generated names are of the form MLn (pronounced MATHLABn) where n is an integer. The following example illustrates these principles. If confused, the reader may take comfort from the fact that most

MATHLAB users never perform arithmetic on functions.

```
#REPEAT F G$
F(X, Y) : X^2Y^3 + A
G(A, X) : SIN(Y) + COS(A)
#'F+'G$
WS(X, ML2) : X^2ML2^3 + A + SIN(Y) + COS(X)
```

REPEAT, here, is a command (cf. Section VI) which causes the indicated data to be re-displayed. The WS(X, ML2) bit is an attempt on MATHLAB's part to tell us that the contents of Workspace (cf. Section IV) is a function.

What happened? MATHLAB first used the X (because it is the first dummy of F) as the first dummy of WS. This expalins both the X^2 (because X is the first dummy of F) and COS(X) (because A is the first dummy of G) in WS. It then tried to use Y (because it is the second dummy of F) as the second dummy of WS, but it ran into trouble since Y was a parameter, not a dummy, of G. Since such a parameter presumably has an external significance, MATHLAB felt it safest to leave it alone. That explains SIN(Y). The program then made up an alternate name, ML2, for the second dummy of WS. That explains $ML2^3$. A was a parameter of F and therefore left alone in that context.

2a. Expressions are summed with an equation by summing the expressions to both sides of the equation. The resulting datum is of type equation.

2b. Functions can only be summed with functions.

SECTION IV

WORKSPACE

The result of each instruction is, of course, a datum, and we say it is stored in workspace. This means that it is temporarily assigned the name WS and may be referred to by that name. This is independent of any other name it may have been assigned. Probably, the most common use of the unquote sign is to evaluate workspace, i.e., to refer to the result of the previous computation. If, for example, the result of the previous computation is an equation, we may now choose to solve that equation for, say, X. We would type

```
'SOLVE ('WS,X) $
```

Not only is the most recent computational result preserved by name in workspace, but all previous results are stored, without name, in a construct called the history of workspace.

We shall see(cf. Section VI) that one of the main purposes of the bookkeeping commands is to enable us to retrieve past results which have current import even if we were not so careful as to store them by name.

SECTION V

SYSTEM FUNCTIONS

We shall list those functions which come pre-defined with the MATHLAB system. For each function, we shall give its syntactic rules, describe the algorithm it is supposed to perform, and display one or two illustrative examples.

The format of a system function is not checked unless the reference to the function is preceded by '. An exception is DERIV. If the unquote is omitted, enough checking is done to cast out expressions such as DERIV(Y). This exception is necessary because unevaluated derivatives have special display and simplification programs.

We shall first discuss those functions which are not intended for matrix manipulation. The functions for matrices are listed, together with a special explanation, in Section V_b.

V_a NON-MATRIX FUNCTIONS

There are fourteen system-defined non-matrix functions which might be categorized as follows:

BASIC

SUBSTITUTE

SIMPLIFY

TERM

DERIV

FUNCTIONS TO PRODUCE ALTERNATIVE RATIONAL FORMS

RATSIMP

PF

PPF

PFE

ADVANCED RATIONAL ROUTINES

INTEGRATE

LTX

ILTX

THE SOLVES

SOLVE

SIMSOLVE

LDESOLVE

SUBSTITUTE

USE: 'substitute($x_1, y_1, x_2, y_2, \cdots, x_n, y_n, z$)

RULES: The x_i, y_i, z can be anything. Hopefully, if the effect would be to create an object that would be illegal if typed in directly, the instruction will be rejected.

MEANING: The algorithm is tricky and powerful:

a) Evaluate z and the y_i. Do not evaluate the x_i.

b) Simultaneously substitute the x_i for the evaluated y_i in the evaluated z.

c) Evaluate the resulting expression. This is the answer.

EXAMPLES:

We present two tiny examples, the first to demonstrate the simultaneity of the substitution and the second to demonstrate how our evaluation strategy allows us to, for example, transform a previously formal (unevaluated) reference to a function to an active (evaluated) reference.

Example 1:

```
#'SUBSTITUTE(X, Y, Y, X, X+Y)$

Y + X
```

Example 2:

```
REPEAT DER Z$

DER :  D Z
       ---
       DX

Z :  X^2

#'SUBSTITUTE('Z, Z, 'DERIV, DERIV, 'DER)$
2X
```

SIMPLIFY

USE: 'simplify(x)

RULES: x can evaluate to anything.

MEANING: We shall not attempt a definition of SIMPLIFY.

In section VI, we shall learn of some of the controls on SIMPLIFY. In particular, we shall see that there is a SWITCH named SIMP which, when ON, causes all data to be simplified upon evaluation. The use of simplify in that circumstance would be redundant; the machine would also perform redundant work. As a result, simplify is a rarely used function. The purposes of including it in the system are:

a) to allow the user to turn simp off and simplify locally.

b) to allow simplification to be built in to more complex user-defined functions. Cf. the definition of definite integration in Section X_b.

EXAMPLE:

```
#1 + 'SIMPLIFY(1-1)$
1 + 0
```

TERM

USE: 'term(n, x)

RULES: n must evaluate to a positive integer.

x must not be a function, number, or formal variable.

MEANING: If x is an equation, term(1, x) will be the left side and term(2, x) the right.

If x is a functional form, term(n, x) is the n'th argument.

Otherwise, x has an algebraic operator at the top level and term (n, x) is the n'th expression bound by that operator.

EXAMPLE:

$$1/5*\text{INTEGRATE}\left(\frac{-X^2 - 2X + 1}{X^3 + 2}, X\right) + 1/10*\text{LOG}(X^2 + 1) + 2/5*\text{ARCTAN}(X)$$

#'TERM(1,'TERM(2,' TERM(1,'WS)))$

$$\frac{-X^2 - 2X + 1}{X^3 + 2}$$

EXPLANATION: WS is, at its top level, the sum of three expressions; the first, i.e., 'TERM(1,'WS), being 1/5*INTEGRATE(...). According to conventional (and MATHLAB) parsing rules, this sub-expression is to be read as (1/5)*INTEGRATE(...). This means that, at its own top level, the sub-expression is to be viewed as a product of two terms; the first being 1/5 and the second, i.e., 'TERM(2,'TERM(1,'WS)), being

$$\text{INTEGRATE}\left(\frac{-X^2 - 2X + 1}{X^3 + 2}, X\right)$$

According to the rule for functional forms, the first term of this last sub-expression, i.e., 'TERM(1,'TERM(2,'TERM(1,'WS))), is its first argument, which is the answer exhibited above.

DERIV

USE: 'deriv(y, x_1, n_1, ... , x_m, n_m)

EXCEPTION TO USE: 'deriv(y, x) is equivalent to 'deriv(y, x, 1).

RULES: The x_i must evaluate to formal variables.

Then n_i must evaluate to non-negative integers.

y must not evaluate to a function.

MEANING: The effect is to differentiate y n_i times with respect to each x_1. Much more is said about this in Section VII.

FUNCTIONS TO PRODUCE ALTERNATIVE RATIONAL FORMS

RATSIMP

USE: 'ratsimp(y, x_1, ... , x_n)

RULES: y may be anything.

The x_1 must be expressions.

MEANING: The program performs the "rational simplification" of an expression by representing it as a rational function. All powers and products of sums are expanded and the entire expression is combined into a rational form with a single denominator, all possible division cancellations being performed in the process. The internal representation is recursive in the number of variables and this can cause the reforming of certain products (cf., example 1 below).

If y is an equation, both sides are RATSIMPed independently. If y is a function, the "body" of its definition is RATSIMPed.

The optional $x_1, \ldots, x_n$ are taken as the main variables in that order. All other variables appearing in y -- these may be either formal variables or non-rational expressions -- are ordered after the x_i in order of appearance.

The reader is encouraged to read [2], or even [3], for a better understanding of RATSIMP, in particular, and the MATHLAB rational function package, in general.

CAUTION: RATSIMP sets DISTEXP (cf. Section VI) to zero.

EXAMPLES:

Example 1:

```
A*B + A*C + B*C

#COMMENT: THIS CAN BE THOUGHT OF AS A POLYNOMIAL
IN A, B, AND C$

#'RATSIMP('WS, A, C)$

(C + B)A + B*C

#COMMENT: POLY IN A WHOSE COEFS ARE POLYS IN C
WHOSE COEFS ARE POLYS IN B$

#'RATSIMP('WS, A, B)$

(B + C)A + C*B

#COMMENT: A--B--C$
```

Example 2:

```
    A(A + X)            1
------------- + ----------
             2
(- A*X + 1)      - A*X + 1
--------------------------
             2
      (A + X)
   ------------- + 1
                2
   (- A*X + 1)

#'RATSIMP('WS)$

   1
--------
 2
X  + 1
```

PF
(Polynomial Factorization)

USE: 'pf(y, x_1, ... , x_n)

RULES: y can be anything.

The x_i must all evaluate to expressions.

MEANING: y will be represented as in RATSIMP and the optional x_i have the same significance. The numerator and denominator of the resulting rational function will be independently factored into polynomials in several variables which are irreducible over the integers.

CAUTION: pf automatically sets distexp to zero.

EXAMPLE:

#REPEAT V$

V: $-B^2A - B^2A*X^2 + A^2X^2 + A^3X^2 - B^2X^2 + A^2X^4 - B^2X^4 + A^3$

#'PF('V, X, A)$

$(A + B)(A - B)(X^2 + A)(X^2 + 1)$

PPF
(Partial Polynomial Factorization)

USE: 'ppf(y, x_1, ... , x_n)

RULES: exactly the same as PF.

MEANING: The only difference is that the factorizations of the numerators and the denominators are with respect to the main variable only. While it is legal to omit all the x_i, the definition of PPF makes the use of at least the main variable customary.

CAUTION: same as PF.

EXAMPLE: (cf.PF example)

#REPEAT V$

V: $-B^2A - B^2A*X^2 + A^2X^2 + A^3X^2 - B^2X^2 + A^2X^4 - B^2X^4 + A^3$

#'PPF('V, X, A)$

$(A^2 - B^2)(X^2 + A)(X^2 + 1)$

PFE
(Partial Fraction Expansion)

USE: 'pfe(y, x_1, ... , x_n)

RULES: y and the x_i as in RATSIMP.

MEANING: The partial fraction expansion of y will be calculated and returned as the answer.

The optional arguments x_1, ... , x_n have the same significance as in RATSIMP.

Equations and functions are handled as in RATSIMP.

EXAMPLE:

$$\frac{X^3 + (Y + Z)X^2 + (Y^2 + 3Z*Y + Z^2)X + Y^3 + Z*Y^2 + Z^2Y + Z^3}{(Y + Z)X^2 + (Y^2 + 2Z*Y + Z^2)X + Z*Y^2 + Z^2Y}$$

#'PFE('WS, X, Y, Z)$

$$\frac{X}{Y + Z} + \frac{Y}{X + Z} + \frac{Z}{X + Y}$$

ADVANCED RATIONAL ROUTINES

INTEGRATE

USE: 'integrate(y, x)

RULES: x must be a formal variable. y must be an expression, rational in x.

MEANING: Integrate returns the integral of y with respect to x. For complete success, the denominator of y must, in general, decompose over the integers into factors which are all linear or quadratic in x.

May interrogate the user as to the signa of certain expressions.

EXAMPLE:

#REPEAT PROB$

$$\text{PROB: } \frac{1}{X^3 + A*X^2 + X}$$

#'INTEGRATE('PROB, X)$
IS THE EXPRESSION

$$A^2 - 4$$

TO BE CONSIDERED POSITIVE NEGATIVE OR ZERO?
#NEGATIVE$

$$- 1/2*LOG(X^2 + A*X + 1) - \frac{A}{SQRT(- A^2 + 4)}ARCTAN(\frac{2X + A}{SQRT(- A^2 + 4)})$$

+

LOG(X)

LTX
(Laplace Transform)

USE: 'ltx(x, t, s)

RULES: t and s must be formal variables. x must be an expression which, were all its products and powers of sums expanded, would be the sum of terms of the form:

$$at^{n}e^{lin_1(t)}trig(lin_2(t))$$

where a is constant with respect to t, n is a positive integer, lin_1 and lin_2 are linear in t, i.e., bt + c, and trig is sin, cos, sinh, or cosh. Any of the factors can be omitted; order is unimportant.

MEANING: returns the direct Laplace transform of x from the t-half-line to the s-plane.

EXAMPLE:

```
E^-A*T SIN(B*T)

#'LTX('WS, T, S)$

                B
-------------------------------
 S^2 + 2A*S + A^2 + B^2
```

ILTX
(Inverse Laplace Transform)

USE: 'iltx(x, s, t)

RULES: s and t must be formal variables and x must be an expression rational in s.

MEANING: returns the inverse Laplace transform of x going from the s-plane to the t-half-line. The program is closely related to that for INTEGRATE and it wins or loses on the same grounds.

In both this function and LTX, expressions like SIN(T) should be regarded as formally restricted to the right-half line.

EXAMPLE:

$$\frac{B}{S^2 + 2A*S + A^2 + B^2}$$

#'ILTX('WS, S, T)$

IS THE EXPRESSION

B^2

TO BE CONSIDERED POSITIVE NEGATIVE OR ZERO?
#POSITIVE$

$E^{-A*T}SIN(B*T)$

THE SOLVES

SOLVE

USE: 'solve(e,x)

RULES: e must evaluate to an equation and x to an expression.

MEANING: Solve will try to solve e for x. The mechanism is discussed in [2] and [3].

It will report any roots it finds, storing them under unique names of the form MLn.

If only partially successful, it will name, report, and store any equations it finds that contain the roots of the original equation.

Roots and Equations are reported together with their multiplicities.

Solve leaves Workspace unchanged.

CAUTION: Solve is a system function, as opposed to a command, for the technical reason that it is capable of evaluating its arguments. It is meaningless to use it internally in an instruction (like 'solve ('e,x) + 1) since solve is basically multivalued and, technically, does not evaluate to anything.

EXAMPLES:

Example 1:

$$\frac{1}{(X-A)(X-B)} + \frac{1}{(X-A)(X-C)} + \frac{1}{(X-B)(X-C)} = 0$$

```
#'SOLVE('WS,X)$
THE ROOTS ARE
```

$$ML1: \frac{A+B+C}{3}$$

```
ONCE
FINISHED
```

Example 2:

$$4A^3X - A^3X^2 + 4X^3 - 4X^4 + X^5 - 4A^3 = 0$$

```
#'SOLVE('WS,X)$
THE ROOTS ARE

ML2 :  2

TWICE AND

ML3:  - A/2 + A/2 SQRT(-3)

ONCE
AND

ML4:  - A/2 - A/2 SQRT(-3)

ONCE  AND
ML5:  A

ONCE
FINISHED
```

Example 3:

$$X^5 = 1$$

```
#'SOLVE('WS,X)$
THE ROOTS ARE

ML6:  1

ONCE AND
THE ROOTS OF

ML7 :  X^4 + X^3 + X^2 + X + 1 = 0

ONCE
FINISHED
```

SIMSOLVE
(Solution of Simultaneous Linear Equations)

USE: 'simsolve(x_1, ... ,x_n)

RULES: Each x must be an equation or a formal variable; at least one an equation and at least one a formal variable. Each equation must be linear in all the formal variables.

MEANING: Solves the equations for the variables. It is not necessary to have the same number of equations as variables. If the system of equations is over-determined, MATHLAB will complain. If it is under-determined in the sense of being of rank k in the n variables specified, n>k, then it will solve for the first k variables mentioned in terms of the n-k others.

CAUTION: Like SOLVE, SIMSOLVE does not alter Workspace nor return a value. Unlike SOLVE, it does not generate names for the answers but, rather, assigns, as names, the desired formal variables. If you have an X and ask SIMSOLVE to solve for X, you will lose the old one -- unless WARNING is ON, cf, Section VI.

EXAMPLES:

Example 1:

```
#E0:Y0=M*X0+B$

Y0 = M*X0 + B

#E1:Y1=M*X1+B$
```

```
Y1 = M*X1 + B

#'SIMSOLVE('E0,'E1,M,B)$

M :  Y0 - Y1
     ---------
     X0 - X1

B :  - X1*Y0 + Y1*X0
     ----------------
          X0 - X1

FINISHED
```

Example 2:

```
#REPEAT E1 E2 E3 E4$
E1 : 3X - 7Y + 14Z - 8W = 24

E2: X - 5Y + 2Z = - 8

E3: Y + Z - W = 6

E4: 2X - 15Y - Z + 5W = - 46

#'SIMSOLVE('E1,'E2,'E3,'E4,X,Y,Z,W)$
X : 5W - 7Z + 22

Y : W - Z + 6

FINISHED
```

LDESOLVE
(Solution of Linear Differential Equations)

USE: 'ldesolve(e,y,x)

RULES: y and x must be formal variables. e must, taking x as the independent variable and y as the dependent one, be an ordinary linear differential equation, of arbitrary order, with constant coefficients. The inhomogeneous term (SIN(2*X) in the first example and 0 in the second example below) must be admissable to LTX, q.v..

MEANING: Solves the differential equation for y in terms of x. Success depends on the factorization of certain polyvariate polynomials over the integers.

The program will immediately type the message:

NEED INITIAL CONDITIONS

followed by a sharp sign. There are two possible responses as this point:

a) ALLFORMAL$

which will cause the machine to use the general initial conditions, $Y(0), Y'(0), \ldots, Y^{(n-1)}(0)$, where Y is the second argument of LDESOLVE and n is the order.

b) ASK$

after which the machine will inquire in succession for the n initial values by typing a message like

Y(0) =

#

and, after receiving an answer, proceeding to the next one. At each step one might answer any of three things:

b_1) any instruction to be normally evaluated.

b_2) FORMAL$

in which case the formal initial condition $Y^{(k)}(0)$ will be used.

b_3) ALLFORMAL$

in which case the current and all succeeding initial conditions are taken as formal. No further inquiry.

In addition to inquiring about initial conditions, LDESOLVE, like INTEGRATE and ILTX, may inquire about the signa of certain expressions.

EXAMPLES:

Example 1:

```
#DERIV(Y, X, 3)+A*DERIV(Y, X)=SIN(2*X)$
```

$$\frac{D^3Y}{DX^3} + A\,\frac{D\,Y}{DX} = \mathrm{SIN}(2X)$$

```
#'LDESOLVE('WS, Y, X)$
NEED INITIAL CONDITIONS
#ALLFORMAL$
IS THE EXPRESSION
A
TO BE CONSIDERED POSITIVE NEGATIVE OR ZERO?
#POSITIVE $
```

$$\frac{2Y(0)A + 2Y''(0) + 1}{2A} + \frac{-Y''(0)A + 4Y''(0) + 2}{A^2 - 4A}\mathrm{COS}(\mathrm{SQRT}(A)X)$$
$$+$$
$$Y'(0)\,\frac{\mathrm{SIN}(\mathrm{SQRT}(A)X)}{\mathrm{SQRT}(A)} - \frac{1}{2A - 8}\mathrm{COS}(2X)$$

Example 2:

```
#SPRING:DERIV(X, T, 2)=-K*X$
```

$$\frac{D^2X}{DT^2} = (-K)X$$

```
#'LDESOLVE('SPRING, X, T)$
NEED INITIAL CONDITIONS
#ASK$
X(0) =
#L$
X'(0) =
#0$
IS THE EXPRESSION
K
TO BE CONSIDERED POSITIVE NEGATIVE OR ZERO?
#POSITIVE$
6*COS(SQRT(K)T)
```

V_b MATRIX FUNCTIONS

The matrix package of MATHLAB is not fully integrated with the other parts of the system. Matrices are officially classified as expressions -- and are so reported by INVENTORY -- but only silly things will happen if one computes matrices with SIMP turned ON or if one employs the arithmetic operators together with matrices. It is, however, possible to perform arithmetic in a more awkward syntax employing the arithmetic functions below. The bookkeeping commands, see Section VI, will all work fine.

There are some seventeen different system functions, all starting with the letters MX, for the manipulation of matrices. To ease the confusion, we shall list them in four categories:

FUNCTIONS TO INTRODUCE OR MODIFY MATRICES

MXENTER

MXELSET

MXIDENT

MXTRANSPOSE

FUNCTIONS TO EXTRACT SUBMATRICES OR ELEMENTS

MXELEMENT

MXROW

MXCOL

MXMINOR

ARITHMETIC FUNCTIONS

MXADD

MXMULT

MXPOWER

MXINVERSE

MXECHELON

FUNCTIONS TO EXTRACT SIGNIFICANT PROPERTIES OF MATRICES

Note: Since the values of these functions are expressions, not matrices, they can be freely mixed with non-matrix functions, arithmetic operators, and SIMP.

MXRANK

MXDETERMINANT

MXCHARPOL

MXTRACE

In the examples that follow we shall use MXENTER to introduce three little matrices, named TWOA, TWOB, and THREE, and then use these throughout the Section for illustrative purposes.

One should note the example accompanying MXPOWER, since it demonstrates the scheme for displaying matrices too large to fit on the page in the normal format.

FUNCTIONS TO INTRODUCE OR MODIFY MATRICES

MXENTER

USE: 'mxenter(m, n)

RULES: m and n are positive integers.

MEANING: permits you to introduce a matrix of m rows and n columns. It asks for each element in turn, by printing: ROW 1 COLUMN 1 : #, etc.; cf. example below. If you make a mistake half way through:

a) If you have not yet hit $ nor carriage return, use rub-out or ↑U as usual, cf., Section XII.

b) If you have not yet hit $ but have hit carriage-return, hit the period followed by carriage-return. The program will ask for the current element again.

c) You have hit $. If input is rejected for any reason, the program will ask for the current element again. If the input is accepted, go on with the rest of the matrix and use MXELSET to correct the defective entry afterwards.

EXAMPLES:

Example 1:

```
#TWOA:'MXENTER(2,2)$
ROW 1 COLUMN 1 : #A11$
ROW 1 COLUMN 2 : #A12$
ROW 2 COLUMN 1 : #A21$
ROW 2 COLUMN 2 : #A22$

( A11  A12)
(          )
( A21  A22)
```

Example 2:

```
#TWOB:'MXENTER(2,2)$
ROW 1 COLUMN 1 : #B11$
ROW 1 COLUMN 2 : #B12$
ROW 2 COLUMN 1 : #B21$
ROW 2 COLUMN 2 : #B22$

( B11   B12)
(           )
( B21   B22)
```

Example 3:

```
#THREE:'MXENTER(3,3)$
ROW 1 COLUMN 1 : #A$
ROW 1 COLUMN 2 : #0$
ROW 1 COLUMN 3 : #B$
ROW 2 COLUMN 1 : #0$
ROW 2 COLUMN 2 : #A$
ROW 2 COLUMN 3 : #B$
ROW 3 COLUMN 1 : #A$
ROW 3 COLUMN 2 : #B$
ROW 3 COLUMN 3 : #0$

( A   0   B )
(           )
( 0   A   B )
(           )
( A   B   0 )
```

MXELSET

USE: 'mxelset(expr, j, k, mat)

RULES: expr must be an expression, j and k positive integers, mat a matrix .

MEANING: returns the matrix obtained by replacing the j, kth element of mat by expr.

EXAMPLE:

```
#'MXELSET(Z, 2, 2, 'TWOA)$

( A11   A12)
(           )
( A21    Z  )
```

MXIDENT

USE: 'mxident(n)

RULE: n must be a positive integer.

MEANING: returns the nxn identity matrix.

EXAMPLE:

```
#'MXIDENT(5)$

( 1   0   0   0   0 )
(                   )
( 0   1   0   0   0 )
(                   )
( 0   0   1   0   0 )
(                   )
( 0   0   0   1   0 )
(                   )
( 0   0   0   0   1 )
```

MXTRANSPOSE

USE: 'mxtranspose(mat)

RULE: mat must be a matrix.

MEANING: returns the transpose of mat.

EXAMPLE:

```
#'MXTRANSPOSE('TWOA)$

( A11   A21 )
(           )
( A12   A22 )
```

FUNCTIONS TO EXTRACT SUBMATRICES OR ELEMENTS

MXELEMENT

USE: 'mxelement(mat, j, k)

RULES: mat must be a matrix, j and k must be positive integers.

MEANING: returns the j, k^{th} element of mat. Since this answer is an expression, not a matrix, MXELEMENT can also be freely mixed with non-matrix functions, arithmetic operators, and SIMP.

EXAMPLE:

```
#'MXELEMENT('TWOA, 2, 1)$

A21
```

MXROW

USE: 'mxrow(k, mat)

RULE: k must be a positive integer, mat must be a matrix.

MEANING: returns the k'th row of mat. This is a matrix.

EXAMPLE:

```
#'MXROW(2, 'TWOA)$

( A21   A22 )
```

MXCOL

USE: 'mxcol(k, mat)

RULES: k is a positive integer; mat a matrix.

MEANING: returns the k^{th} column of mat. This is a matrix.

EXAMPLE:

```
#'MSCOL(2, TWOA)$

( A12 )
(     )
( A22 )
```

MXMINOR

USE: 'mxminor(mat, j, k)

RULES: mat must be a matrix; j and k must be positive integers.

MEANING; returns the matrix obtained by striking out the j^{th} row and k^{th} column of mat.

EXAMPLE:

```
#'MXMINOR('THREE, 1, 2)$
( 0  B )
(      )
( A  0 )
```

MXADD

USE: 'mxadd(x_1, ... ,x_n)

RULES: Each x_i must be either a matrix or an expression.

At least one must be a matrix.

MEANING: In its simplest application the x_i are all matrices. In that case the answer is the (matrix) sum of the x_i.

The user should understand that MATHLAB matrix arithmetic is based upon the rational function package (see RATSIMP, Section V_a or Reference [2] or [3]). If some of x_i are expression, not matrices, then they are taken as the main variables -- in the order in which they appear -- in the representation of each element of the sum matrix. In particular, should only one of the x_i, say x_k, be a matrix, the effect is precisely that of RATSIMPing each element of x_k, with the importance of the variables being derived from the order of appearance of the x_i, $i \neq k$.

EXAMPLES:

Example 1:

```
#'MXADD('TWOA,'TWOB)$

( A11 + B11    A12 + B12 )
(                        )
( A21 + B21    A22 + B22 )
```

Example 2:

```
#'MXADD('TWOA,'TWOB,B12,B21)

( A11 + B11    B12 + A12 )
(                        )
( B21 + A21    A22 + B22 )
```

MXMULT

USE: 'mxmult($x_1, \ldots, x_n$)

RULES: The x_i must be matrices or expressions.

MEANING: Returns the product of the x_i. Matrices are multiplied in the order they occur. If an expression and a matrix are to be multiplied, the expression is considered a scalar which multiplies the matrix. If all the x_i are expressions, the answer is an expression, not a matrix.

EXAMPLES:

Example 1:

```
#'MXMULT('TWOA,'TWOB)$

( B11*A11 + B21*A12    B12*A11 + B22*A12 )
(                                        )
( B11*A21 + B21*A22    B12*A21 + B22*A22 )
```

Example 2:

```
#'MXMULT(1/A,'THREE)$

(          B   )
( 1   0   ---  )
(          A   )
(              )
(          B   )
( 0   1   ---  )
(          A   )
(              )
(     B        )
( 1  ---  0    )
(     A        )
```

MXPOWER

USE: 'mxpower(mat, n)

RULES: mat must be a matrix; n a positive integer.

MEANING: returns mat raised to the n^{th} power.

EXAMPLE:

(Note the format used to display large matrices.)

#'MXPOWER('THREE, 4)$

WS(1, 1) : $A^4 + 3B*A^3 + B^2A^2 + B^3A$

WS(1, 2) : $3B^2A^2 + B^3A + B^4$

WS(1, 3) : $B*A^3 + 2B^2A^2 + 2B^3A$

WS(2, 1) : $3B*A^3 + B^2A^2 + B^3A$

WS(2, 2) : $A^4 + 3B^2A^2 + B^3A + B^4$

WS(2, 3) : $B*A^3 + 2B^2A^2 + 2B^3A$

WS(3, 1) : $A^4 + 2B*A^3 + 2B^2A^2$

WS(3, 2) : $B*A^3 + 2B^2A^2 + 2B^3A$

WS(3, 3) : $B*A^3 + 2B^2A^2 + 2B^3A + B^4$

MXINVERSE

USE: 'mxinverse(mat, $x_1, x_2, \ldots, x_n$)

RULES: mat must be a square matrix; the optional x_i must be expressions.

MEANING: Returns the inverse of mat with the x_i determining the order of the variables in the representation of each element. Cf. MXADD.

If mat is singular, MATHLAB complains.

EXAMPLES:

```
#J:'MXENTER(2,2)$
ROW 1 COLUMN 1 : #DERIV(U,X)$
ROW 1 COLUMN 2 : #DERIV(U,Y)$
ROW 2 COLUMN 1 : #DERIV(V,X)$
ROW 2 COLUMN 2 : #DERIV(V,Y)$
```

```
( D U   D U )
( ---   --- )
( DX    DY  )
(           )
( D V   D V )
( ---   --- )
( DX    DY  )
```

```
#'MXINVERSE('J)$
```

```
(          D V                          D U          )
(          ---                          ---          )
(          DY                           DY           )
(  ------------------          - ------------------  )
(   D V D U   D V D U           D V D U   D V D U    )
(   ---*---  - ---*---          ---*---  - ---*---   )
(   DY  DX    DX  DY            DY  DX    DX  DY     )
(                                                    )
(          D V                          D U          )
(          ---                          ---          )
(          DX                           DX           )
(  - ------------------        ------------------    )
(   D V D U   D V D U           D V D U   D V D U    )
(   ---*---  - ---*---          ---*---  - ---*---   )
(   DY  DX    DX  DY            DY  DX    DX  DY     )
```

```
#'MXINVERSE('J,DERIV(V,Y),DERIV(V,X))$
```

```
(          D V                          D U          )
(          ---                          ---          )
(          DY                           DY           )
(  ------------------          - ------------------  )
(   D U D V   D U D V           D U D V   D U D V    )
(   ---*---  - ---*---          ---*---  - ---*---   )
(   DX  DY    DY  DX            DX  DY    DY  DX     )
(                                                    )
(          D V                          D U          )
(          ---                          ---          )
(          DX                           DX           )
(  - ------------------        ------------------    )
(   D U D V   D U D V           D U D V   D U D V    )
(   ---*---  - ---*---          ---*---  - ---*---   )
(   DX  DY    DY  DX            DX  DY    DY  DX     )
```

USE: 'mxechelon(mat, x_1, ... , x_n)

RULES: mat must be a matrix. The x_i must be expressions.

MEANING: The x_i are optional and serve the same purpose as the extra arguments of MXINVERSE, q.v..

The value is a matrix in echelon form, i.e., which:

1) has 1's on the main diagonal, i.e., the diagonal starting at the upper left-hand corner. In degenerate cases some of the diagonal elements may be 0's;

2) 0's below the main diagonal;

(These two conditions imply that, if mat is square, the answer is triangular.)

3) is row-equivalent to mat. The permissible row operations are:

a) interchange of rows.

b) addition of rows.

c) multiplication of rows by expressions.

EXAMPLE:

```
        ( X    Y    1 )
  MAT:  (               )
        ( X   - Y   0 )
```

```
 #'MXECHELON('MAT)$

 (         Y      1   )
 ( 1     ---    ---  )
 (         X      X   )
 (                    )
 (                1   )
 ( 0       1    ---  )
 (                2Y  )
```

FUNCTIONS TO EXTRACT SIGNIFICANT PROPERTIES OF MATRICES

MXRANK

USE: 'mxrank(mat)

RULE: mat must be a matrix.

MEANING: the value is a non-negative integer, the rank of the matrix computed on the assumption that there are no hidden relationships among the variables appearing in it.

EXAMPLES:

Example 1:

```
#'MXRANK('THREE)$

3
```

Example 2:

```
(                  2      )
(  A - B         A        )
(                         )
(   2     2    3     2    )
( A  - B    A  + A  B     )

#'MXRANK('WS)$

1
```

MXDETERMINANT

USE: 'mxdeterminant(mat)

RULE: mat must be a square matrix.

MEANING: returns the determinant of mat.

EXAMPLE:

```
#'MXDETERMINANT('TWOA)$

A22*A11  -  A21*A12
```

MXCHARPOL

USE: 'mxcharpol(mat, λ)

RULES: mat must be a square matrix; λ a formal variable not appearing in mat.

MEANING: returns the characteristic polynomial (in λ) of mat; i. e., the determinant of (mat - λI), where I is the identity matrix.

EXAMPLE:

```
#'MXCHARPOL('TWOA, L)$
L^2 + (- A22 - A11)L + A11*A22 - A12*A21
```

MXTRACE

USE: 'mxtrace(mat)

RULE: mat must be a square matrix.

MEANING: returns the trace of mat; i. e., the sum of its diagonal elements.

EXAMPLE:

```
#'MXTRACE('THREE)$
2A
```

SECTION VI

COMMANDS

At any time that MATHLAB has printed a # and is waiting for an instruction, the user may instead type in a command. Commands differ from instructions in that their purpose is not to evoke some mathematical computation but rather to serve some less direct function, perhaps a bookkeeping task or the establishment of a different environment for the simplification routines.

Commands also differ from instructions in certain technical respects:

1. They do not evaluate their arguments.

2. They are written parentheses-free. They take the general form:

commandname argument1 ... argumentn $

3. With the exception of KILL and RESTORE, they do not affect the contents of workspace or its history.

4. With the exception of REPEAT, they do not respond with the display of a mathematical expression. They respond with either nothing or some relevant information.

SWITCHES

Certain commands may be regarded as switches -- either double-throw or infinite-positioned -- whose purpose is not to do anything at the moment they are thrown, but, rather to affect the future flow of MATHLAB by modifying the meaning of subsequent instructions. Double-throw switches all obey the following rules:

1. Each double-throw switch is initially OFF.
2. Each double-throw switch can be set by using its name as a command.
3. There are three legitimate forms for this command.

   For example, there is a double-throw switch named SIMP. One may type:

   SIMP ON $

   in which case, SIMP is set ON. Or, one may type:

   SIMP OFF $

   in which case, SIMP is set OFF. Or, one may type:

   SIMP $

   in which case, SIMP is thrown to the position opposite to its previous one, i.e., OFF if it was ON and ON if it was OFF.
4. Double-throw switches produce no response.

We shall group the commands as follows:

SIMPLIFICATION

- SIMP
- VARORDER
- DISTEXP

DIFFERENTIATION

- DEPENDENT
- INDEPENDENT
- DEPENDENCIES

BOOKKEEPING: CORE

- REPEAT
- STORE
- RESTORE
- KILL
- RENAME
- INVENTORY
- WARNING

BOOKKEEPING: DISK

- DSKDUMP
- DSKLOAD
- DSKDISP
- LPTWIDTH

DISPLAY

- TALLPAR
- ALLSTAR

MISCELLANEOUS

- ALIAS
- ALIASKILL
- COMMENT
- EV

SIMPLIFICATION

SIMP

A double-throw switch. The most important in MATHLAB.

If SIMP is ON, the result of each evaluated instruction is simplified before being stored and displayed. If OFF, not. One reason for allowing SIMP to be switched OFF is to prevent it from performing gratuitous transformations such as changing SQRT(-X) to SQRT(-1)SQRT(X). Another, more important, reason is to allow the simplifications to be handled by the rational function routines, e.g., RATSIMP without interference from SIMP.

VARORDER

An infinite-positioned switch.

USE: VARORDER $var_1 \ldots var_n$\$

Each var_i must be a formal variable. Establishes the pecking order of the variables during any simplification. Has no effect unless SIMP is ON or the system function SIMPLIFY is employed. The order of variables is preserved, though in a dormant state, through those periods in which SIMP is turned OFF. The named variables will take precedence in the order listed and all of these will take precedence over any not listed. In any case, the algorithm is neither clear nor satisfactory.

USE: VARORDER \$

Puts the program back on its own resources.

EXAMPLE:
```
#SIMP ON$
#VARORDER A B C$
#A*B + A*C + B*C$
C*B + C*A + B*A
#VARORDER A C B$
#'WS$
B*C + B*A + C*A
```

DISTEXP

An infinite-positioned switch.

USE: DISTEXP n $

RULE: n a non-negative integer.

REPLY: none.

This governs, except when superceded by calls to the rational function system functions, the expansion of products and powers of sums during simplification.

n = 0; no expansion.

n = 1; expansion only of products of different sums.

n = k; k > 1; all sums to a power $\leq$ k (as well as products of different sums) will be expanded.

USE: DISTEXP $

Equivalent to DISTEXP 0 $

For DISTEXP to have any effect, either SIMP must be ON or the system function SIMPLIFY must be employed. Its value remains unchanged, but dormant, when SIMP is OFF. It is, however, reset to zero by the rational function routines RATSIMP, PF, AND PPF.

EXAMPLE:

```
#SIMP ON$
#DISTEXP$
#D:(A + B)*C + (A +B) ↑ 2$
C(B + A) + (B + A)^2
#DISTEXP 1$
#'D$
B*C + C*A + (B + A)^2
#DISTEXP 2$
#'D$
B^2 + A^2 + B*C + 2B*A + C*A
```

DIFFERENTIATION

DEPENDENT

An infinite-positioned switch.

USE: DEPENDENT Y X1 ... Xn $

RULE: Y and the Xi must be formal variables.

Declares, for the purpose of differentiation, that Y is dependent on all the Xi. See Section VII.

REPLY: none.

INDEPENDENT

An infinite-positioned switch.

USE: INDEPENDENT Y X1 ... Xn $

RULE: Y and all the Xi must be formal variables.

Removes the dependency of Y on the Xi. See Section VII.

REPLY: none

DEPENDENCIES

USE: DEPENDENCIES Y1 ... Yn $

RULE: The Yi must be formal variables.

Inventories all the formal variables on which each Yi is dependent. See Section VII.

REPLY: The requested information.

USE: DEPENDENCIES $

Inventories all dependencies.

REPLY: information. For Example,

Z: X Y
W: U V
means Z depends on X and Y; W on U and V.

REPEAT

USE: REPEAT NAME1 ... NAMEn $

REPLY: This data stored under each of the requested names is displayed, labelled with its name.

USE: REPEAT n $ (n a non-negative integer)

REPLY: The n'th object back in the history of workspace is displayed. If n = 0, no effect and no reply. The object currently in Workspace is number 1.

USE: REPEAT $

REPLY: Displays the contents of workspace.

STORE

USE: STORE NAME $

The datum in workspace is stored under the name NAME.

REPLY: none

Note that the sequence:

RESTORE n $

STORE NAME $

will serve to store the n'th object back in the history of workspace under the name NAME.

If one objects to its moving into workspace,

RESTORE n $

STORE NAME $

KILL $

RESTORE

USE: RESTORE NAME $

The object stored under the name NAME is brought back into workspace.

REPLY: none.

USE: RESTORE n $ (where n is a non-negative integer)

The n'th item back in the history of workspace is brought back into workspace -- also stays back in what is now the $(n + 1)^{st}$ place.

REPLY: none.

KILL

USE: KILL NAME1 ... NAMEn $

The data stored under the given names are destroyed.

REPLY: none.

USE: KILL n $ (where n is a non-negative integer)

The n most recent objects are removed from the history of workspace. The object currently in workspace is number 1. If n = 0, no effect.

REPLY: none.

USE: KILL HISTORY $

The history of workspace is wiped out. The only remaining stored objects are those stored by name.

REPLY: none.

USE: KILL ALL $

The history of workspace and all objects stored by name are destroyed. This is almost a complete re-initialization of MATHLAB. Only the switches are not reset.

REPLY: none.

USE: KILL $

Destroys object currently in workspace.

REPLY: none.

It is important to understand the synchronization of REPEAT, KILL, and RESTORE. To bring back an old expression into workspace, experiment with REPEAT until REPEAT n $ displays the correct expression. Then RESTORE n $ will bring it back.

To destroy everything which occurred after a particular expression, experiment with REPEAT until REPEAT n $ produces the most recent datum you wish to preserve. Then KILL n-1 $ throws away everything after the good one.

RENAME

USE: RENAME OLDNAME NEWNAME $

The datum that was stored under the name OLDNAME is now stored under the name NEWNAME.

REPLY: none.

INVENTORY

USE: INVENTORY OPTIONALWORD OPTIONALWORD OPTIONALWORD $

The optionalwords should be chosen from the list:

EXPRESSION EQUATION FUNCTION.

Any 1, 2, or all 3 of these may be used.

REPLY: A list of the names of those data stored by name which are of the types requested.

USE: INVENTORY $

REPLY: A list of the names of all data stored by name, sorted by type.

WARNING

A double-throw switch.

If WARNING is ON and you try to store a datum by a name already used for another datum, the machine will ask you if you want to lose the old one. If you say "YES $", it will continue. If you say "NO $", it will ask you for a new name for the datum being stored at present.

An exception is made for the name "ws", since workspace changes with the completion of each evaluated instruction. No warning is given, but the old workspace recedes into the history of workspace and may be retrieved by use of the command RESTORE.

If warning is off, the instruction:

V : 'V + 1 $

will lose the old V without comment.

BOOKKEEPING: DISK

DSKDUMP

USE: DSKDUMP$

All data stored by name -- with the exception of workspace -- are written on a file in the user's file directory. The format used is a general MATHLAB internal representation expected by the complementary command DSKLOAD. The program assigns the file a generated name of the form MLn. DMP, where n is an integer.

REPLY: the name of the file.

CAUTION: Either at the end of each MATHLAB session or at the beginning of each new one, one should -- using the time-sharing system, not MATHLAB -- delete all the DMP files no longer of value and change the first names of those to be preserved. A fresh MATHLAB initiates the count; so that issuing the command DSKDUMP could construct a file MLn. DMP with the same name as one from a previous session. This would destroy the older one.

DSKLOAD

USE: DSKLOAD NAME1 ... NAMEn $

Loads the data in files NAME1. DMP, ... , NAMEn. DMP into core.

REPLY: The names of the files loaded.

USE: DSKDISP $x_1 \ldots x_n\$$

RULE: Each x_i must be one of the following:

a) the name of some datum.

b) a positive integer.

c) the special word TTY.

d) the special word LPT.

e) the special word FIN.

MEANING: The data named will be displayed, in the usual two-dimensional format, on the disk; presumably for later printing by a teletype or line printer. The width allotted to the display, which governs how lines will be split, is determined as follows:

1) It is initialized in a fresh MATHLAB to 72.

2) Whatever value it has stays in effect until changed by some use of DSKDISP.

3) If one of the x_i, is an integer, that is the value.

4) If one of the x_i is TTY, the value is 72.

5) If one of the x_i is LPT, the value is that associated with the line printer. See LPTWIDTH.

The file name used is of the form MLn. DSP, with MLn being generated by the program. If the word FIN is included in the command, the file will be closed after the current display requests are complied

with. If not, the file remains open and a future use of DSKDISP will display in the same file.

REPLY: The file name followed by the word OPEN if the file is left OPEN and CLOSED if it is closed.

CAUTION: Same danger as with DSKDUMP.

LPTWIDTH

USE: LPTWIDTH n$

n must be a positive integer. The width associated with the use of LPT in DSKDISP, q.v., is set at n. In a fresh MATHLAB, this is initialized at 132.

REPLY: none.

TALLPAR

A double-throw switch.

If TALLPAR is ON, the display will draw lots of parentheses to indicate the vertical extent of any subexpression. If TALLPAR is OFF, the display will include only those parentheses needed to delimit the horizontal extent of each subexpression.

EXAMPLE:

```
#(1+X↑2)↑3$
```

$$(1 + X^2)^3$$

```
#TALLPAR ON$
#REPEAT$
```

$$\begin{pmatrix} & 2 \\ 1 + X & \end{pmatrix}^3$$

ALLSTAR

A double-throw switch.

If ALLSTAR is ON, all multiplicative asterisks will be displayed. If ALLSTAR is OFF, the display will omit the unambiguous ones.

EXAMPLE:

```
#5*X$

5X

#ALLSTAR ON$
#REPEAT$
5*X
```

MISCELLANEOUS

ALIAS

USE: ALIAS INTERNALNAME TYPEDNAME $

This puts a whammy on the machine so that, whenever you type TYPEDNAME, the machine will think you typed INTERNALNAME. The main purpose is to allow the user to choose personal abbreviations like D for DERIV.

REPLY: none.

ALIASKILL

USE: ALIASKILL TYPEDNAME1 ... TYPEDNAMEn $

Removes the aliases. See ALIAS.

REPLY: none.

COMMENT

USE: COMMENT text$

A convenience, text is any text (sequence of characters) including spaces and carriage returns but no periods. It is ignored.

REPLY: none.

EV

USE: EV $

followed by a LISP S-Expression.

This is a command for the use of the LISP initiate. It reads one S-expression from the typewriter and hands it to EVAL. See Section XII for more on the use of LISP from within MATHLAB and vice versa.

REPLY: The value of the S-Expression.

SECTION VII

DIFFERENTIATION

If a fresh MATHLAB were asked to differentiate $R^2 = X^2 + Y^2$, it would respond 0 = 2*X. This sort of differentiation, in which the derivative of a subexpression is considered to be zero unless the subexpression explicitly contains the variable of differentiation, is almost useless.

The way in which a mathematician would differentiate that equation would depend on what it meant to him. The machine cannot guess this meaning nor can it understand: 'This is the equation of a circle.' What we have done is to allow the user a simple method of telling the machine which formal variables depend on which. It is up to the user to understand which declarations of dependencies will guide the program towards getting the right answer for his problem.

Let there be stored an equation whose name is E and whose value is $R^2 = X^2 + Y^2$. If the user understands this as the equation of a circle, then R is a constant and the equation defines Y (implicitly) as a function of X. In that case, the user should issue the command:

DEPENDENT Y X $

after which, the instruction:

'DERIV('E,X) $

will produce the response:

$$0 = 2X + 2Y \frac{D\ Y}{DX}$$

If, on the other hand, the user understands E as one of a pair of equations which define the transformation from rectangular coordinates to polar coordinates in the plane, then Y and X are independent and the equation defines R (implicitly) as function of both of them. In that case he should type the command:

DEPENDENT R X $

after which, the instruction:

'DERIV('E,X) $

would yield:

$$2R\frac{D\ R}{DX} = 2X$$

There are some other rules about differentiation that the user should be familiar with.

1. The expression $\frac{D\ Y}{DX}$ does not simplify to zero, even if Y has not been declared dependent on X. It will however, under the same hypothesis, be simplified to zero if it is part of an expression being differentiated with respect to X.

2. The differentiation of formal derivatives is always formal: The derivative of $\frac{D\ Y}{DX}$ with respect to X is $\frac{D^2\ Y}{DX^2}$ if Y depends on X and zero otherwise.

The derivative of $\frac{D\ Y}{DX}$ with respect to T is $\frac{D^2\ Y}{DT\ DX}$ if Y depends on both X and T and zero otherwise.

3. When, in the course of a differentiation, a functional form -- such as $g(x_1, \ldots, x_n)$ -- is encountered, the program will first look to see if a gradient is stored for g. If it is, this will be used.

If the gradient is not stored, the program will query the user by typing:

NEED GRADIENT OF G
#

At this point the user has a choice of three responses:

1) He can type: ALLFORMAL$, in which case the machine will generate ugly formal functional forms D1G(.), ... , DnG(.) for the n components of the gradient.

2) If there is stored a function named G and the user would like the machine to differentiate its definition to obtain the gradient, he can type: USEDEF$

3) He can type ASK$. The machine will respond by typing a message like:

CONSIDER G(X1, X2, ... , Xn)

$$\frac{D}{DX1} G(X1, X2, \ldots, Xn) =$$

#

At this point the user has a choice of three responses:

a) He can type any MATHLAB <u>instruction</u> whose <u>value</u> will be taken as the requested component of the gradient.

b) He can type FORMAL$. The machine will use DIG(.) for this component. In either of these first two cases the machine would ask in a similar way for the next component and the same choices would be available.

c) He can type ALLFORMAL. The current and all succeeding components will be FORMAL. No further inquiry.

The following patently artificial example exhibits most of the possibilities:

```
#H(X):SIN(X/A)↑2$
```

$$WS(X) : \quad SIN^2\left(\frac{X}{A}\right)$$

```
#'DERIV(G(H(X↑2),X↑3,X↑4,X↑5)↑3+ARCTAN(X),X)$
NEED GRADIENT OF G
#ASK$
CONSIDER G(X1,X2,X3,X4)
D
----G(X1,X2,X3,X4) =
DX1
#FORMAL$
D
----G(X1,X2,X3,X4) =
DX2
#Z$
D
----G(X1,X2,X3,X4) =
DX3
#ALLFORMAL$
NEED GRADIENT OF H
#USEDEF$
```

$$3G^2(H(X^2),X^3,X^4,X^5)$$

$$*$$

$$3Z*X^2 + \frac{4X*SIN(-\frac{X^2}{A})DIG(H(X^2),X^3,X^4,X^5)COS(-\frac{X^2}{A})}{A}$$

$$+$$

$$4X^3D3G(H(X^2),X^3,X^4,X^5) + 5X^4D4G(H(X^2),X^3,X^4,X^5)$$

$$+$$

$$\frac{1}{X^2+1}$$

4. The program will simplify

$$\frac{DZ^5}{DX^2\,DY^3} + \frac{DZ^5}{DY^3\,DX^2} \text{ to } 2\frac{DZ^5}{DX^2\,DY^3}$$

SECTION VIII

THE TWO-DIMENSIONAL DISPLAYS

The MATHLAB display program [4] is designed to produce legible two-dimensional displays of mathematical expressions within the limitations of character set and positional placement imposed by such common devices as teletypes and line-printers. This means, for example, that it must use dashes, not underlines, to draw its division bars and that exponents must be a full line above their bases as opposed to the fractional height employed in the professional setting of mathematical text.

The results are certainly of sufficient quality to encourage the interactive utilization of MATHLAB, our primary goal. By designing for minimal facilities we have ensured that our program is operative on teletypes, line-printers, fixed character display terminals, graphical terminals with character generators, and even, as it was on one accidental occasion, a plotter. Larger expressions are particularly amenable to line-printer output. See DSKDISP, Section VI, for instructions on how to arrange for the display of your results on off-line line-printers.

MATHLAB is set up for a line-width of 72 characters, standard to teletypes and many teletype-compatible displays. If you are operating with a display having a width of n columns, $n \neq 72$, issue the command:

EV$

followed by:

(LINELENGTH n)

followed by a carriage-return.

There is one aspect of the MATHLAB displays which is unconvential and which must, therefore, be quite carefully explained. The examples used throughout this manual are all minute and, with the exception of that accompanying MXPOWER and the missing gradient example of the previous section, all fit comfortably within the confines of the 72 column width provided by a teletype. Even the MXPOWER example fails to fit only in the sense that the entire matrix does not fit. It is still true that each element of the matrix, as a mathematical expression, can be displayed within the allotted width. In real applications, the expressions that can be written on one line are more the exception than the rule. When MATHLAB has to break an expression over several lines, it employs an unconventional -- though unambiguous and, we believe, revelatory -- syntax for the output. This syntax may be defined by the following rule:

Consider any expression as consisting of a top-level operator with two operands. Slight modifications are needed when the expression is a functional form or when the operator has multiple arguments; cf, the multi-summand denominator in the example below.

If the expression fits, use conventional syntax.

If the expression does not fit, display the first operand indented from the current left origin; display the operator at the current left origin; display the second operand indented like the first operand.

Apply the rule recursively to each operand.

As an example, consider this MATHLAB display:

$$
\begin{aligned}
&PHI(X)\\
&+\\
&\quad A0*X^6 + A1*X^5 + A2*X^4 + A3*X^3 + A4*X^2 + A5*X + A6\\
&\quad /\\
&\qquad B0^2X^{12} + 2B0*B1*X^{11} + (2B0*B2 + B1^2)X^{10}\\
&\qquad +\\
&\qquad (2B0*B3 + 2B1*B2)X^9 + (2B0*B4 + 2B1*B3 + B2^2)X^8\\
&\qquad +\\
&\qquad (2B0*B5 + 2B1*B4 + 2B2*B3)X^7\\
&\qquad +\\
&\qquad (2B0*B6 + 2B1*B5 + 2B2*B4 + B3^2)X^6\\
&\qquad +\\
&\qquad (2B1*B6 + 2B2*B5 + 2B3*B4)X^5 + (2B2*B6 + 2B3*B5 + B4^2)X^4\\
&\qquad +\\
&\qquad (2B3*B6 + 2B4*B5)X^3 + (2B4*B6 + B5^2)X^2 + 2B5*B6*X + B6^2
\end{aligned}
$$

How do we read this? At the extreme left, we see a +. This means the expression is a sum of the expression above the +, namely PHI(X), and that below the +, which is considerably more complicated. Below the +, at the same indentation level as PHI(X) is a /. This means

the second summand is a fraction. The numerator is the polynomial with the A's for coefficients displayed in conventional Syntax. Or the same indentation level with this polynomial is a vertical sequence of +'s. The denominator is then a polynomial extending over six lines, with each line read conventionally. The entire expression can thus be seen to be of the form:

$$\mathrm{PHI}(X) + \frac{A0*X^{6} + \ldots + A6}{B0^{2}X^{12} + \ldots + (2B0*B3 + 2B1*B2)X^{9} + \ldots + B6^{2}}$$

By printing some additional parentheses -- in our example, one pair surrounding the numerator and one the denominator of the fraction -- we could ensure that, reading the expression top-to-bottom left-to-right, we would have a conventional sequence of operators and sub-expressions with the customary precedence conventions. However, we believe our displays are less cluttered and, once the user has disciplined himself to the rule that sub-expressions are always split at the top level, more transparent.

SECTION IX

ERRORS

So much effort has been invested in MATHLAB to provide adequate, self-explanatory error messages, that we shall take the risk, here, of offering no individual explanations. These messages apply primarily to the evaluation of functional forms and may be concerned with syntactic difficulties, e.g.,

YOU ARE GIVING ME 2 ARGUMENTS FOR F1 WHICH HAS 3 ARGUMENTS

or with those of a semantic nature, e.g.,

CAN ONLY INTEGRATE RATIONAL FUNCTIONS.

There is, however, one aspect of MATHLAB for which the error protection is grossly inadequate: the parsing of ungrammatical instructions. While a few violations will be caught, e.g., X Y$ will be rejected, the general tendency is for the parser to valiantly and treacherously try to parse any input. It will, for example, parse 5*(X + Y $ the same as it would 5*(X + Y)$, which is nice if that is what you meant but a potential catastrophe if it is not. Beware, in particular, if you receive the message "NO DECIMALS". This means you have typed a period, which is illegal; but it also means that the parser has attempted a new translation immediately to the right of the period.

While the greatest danger is an undetected improvisation by the parser, there is also the possibility of MATHLAB simply losing its cool

when processing ill-formed expressions. The following example was discovered by my ten year old son (SIMP is ON and he is using MATHLAB as a desk calculator):

```
#145-5852*25-$
3423 ILL MEM REF FROM NUMBERP
BACKTRACE
TALKP-NUMBERP CLEAN1-TALKP CLEAN1-CLEAN1 CLEAN1-CLEAN1
CLEAN1-CLEAN1
CLEAN1-CLEAN1 CLEAN1-CLEAN1 CLEAN-CLEAN1 MLABSET-CLEAN
ERRSET-*EVAL
#9658+3214-9*4$

12836
```

His reaction was, "The machine went crazy and ended in evil." Such messages from the bowels of LISP are not likely to mean much more to most MATHLAB users. Notice, however that MATHLAB did regain control, print a #, and function correctly afterwards.

There are two other known circumstances where the complaint is from LISP, not MATHLAB. One occurs when DSKLOAD is asked to load a non-existent file:

```
#DSKLOAD WHIT3$
(DSK: (WHIT3 . DMP) )
CAN'T FIND FILE - INPUT
BACKTRACE
INPUT-INPUT DSKLOAD-*EVAL UNQUOTE-*EVAL ERRSET-*EVAL
```

At least here the message is clear and MATHLAB has once more regained control. The other circumstance is when space in core is exhausted, cf. Section XI.

SECTION X

EXAMPLES

This section has two purposes. The first, the subject of part a, is the presentation of the complete solution of a little problem chosen primarily for its succinctness and for the variety of the most commonly used MATHLAB facilities -- substitution, differentiation, simplification, bookkeeping, etc. -- that it brings into play. The purpose of part b is to introduce, primarily by example, the meaning of the syntactic element " and the limited opportunity it provides for programming in MATHLAB.

X_a A COMPLETE MATHLAB SESSION

The problem we have chosen is the solution of a differential equation, the identical one solved so expertly by the machine in Example 1, LDESOLVE, Section V_a. Our solution will, in fact, be inferior to that found by the machine in that ours will not reveal so explicitly its dependence on the initial values. We shall log-on, enter and start MATHLAB, solve the problem, save the differential equation and its solution on the disk, leave MATHLAB, rename the disk file and log off, preserving the new file in the process.

```
SYSTEM #124 5S03A3 10:21:57

PLEASE LOGIN OR ATTACH.

.LOG
JOB 16 SYSTEM #124 5S03A3 TTY15
#104,337
PASSWORD:
1022        03-AUG-71        TUE

****PLEASE CLEAN UP YOUR DISK AREA ****
```

```
.R MATHLAB

(MATHLAB)

# MEANS MATHLAB IS LISTENING

#COMMENT WE SHALL INTRODUCE TWO PET ALIASES, THE DIFFERENTIAL
EQUATION, AND THE HOMOGENEOUS EQUATION$
#ALIAS DERIV D$
#ALIAS SUBSTITUTE SUB$
#DE:D(Y,X,3)+A*D(Y,X)=SIN(2*X)$
```

$$\frac{D^3Y}{DX^3} + A\frac{DY}{DX} = SIN(2X)$$

```
#HDE: 'TERM(1,'DE)=0$
```

$$\frac{D^3Y}{DX^3} + A\frac{DY}{DX} = 0$$

```
#COMMENT SHALL LOOK FOR THREE VALUES OF L SUCH THAT
E↑(L*X) SATISFIES HDE; FIRST LET US SUBSTITUTE$

#'SUB(E↑(L*X),Y,'HDE)$
```

$$\frac{D^3}{DX^3}E^{L*X} + A\frac{D}{DX}E^{L*X} = 0$$

```
#'SUB('D,D,'WS)$
```

$$L^3E^{X*L} + A*L*E^{X*L} = 0$$

```
#COMMENT MUST PREPARE FOR SOLVE; NOTE USE OF NON-RATIONAL
EXPRESSION AS OPTIONAL ARGUMENT OF RATSIMP$
#'RATSIMP('WS, E↑(X*L), L)$
```

$$(L^3 + A*L)E^{X*L} = 0$$

```
#'TERM(1,'TERM(1,'WS) )=0$
```

$$L^3 + A*L = 0$$

```
#'SOLVE('WS, L)$
THE ROOTS ARE

ML1 :  0

ONCE AND
ML2 :  SQRT(- A)

ONCE
AND

ML3 :  -SQRT(- A)


ONCE
FINISHED

#COMMENT WE INTRODUCE A LITTLE FUNCTION TO HELP US PASTE THE
THREE ROOTS TOGETHER$
#F(C, L):C*E↑(L*X)$
```

$$WS(C,L) : C*E^{L*X}$$

```
#'F(CO,'ML1) + 'F(C1,'ML2)+'F(C2,'ML3)$
```

$$CO*E^{0X} + C1*E^{SQRT(-A)X} + C2*E^{(-SQRT(-A))X}$$

```
#COMMENT CO*E↑(0*X)  ?  TIME FOR SIMPLIFICATION$
#SIMP$
#'WS$
```

$$CO + C2*E^{-SQRT(-1)X*SQRT(A)} + C1*E^{SQRT(-1)X*SQRT(A)}$$

```
#COMMENT THAT IS OUR SOLUTION TO HDE; LET US NAME IT AND THEN
CHECK THAT EVERYTHING WE NEED IS STORED BY NAME AND
THEN THROW AWAY THE HISTORY OF WORKSPACE$
#STORE YH$
#INVENTORY$

EXPRESSION : YH WS ML3 ML2 ML1


EQUATIONS : HDE DE


FUNCTIONS : F


#KILL ML3 ML2 ML1 F$
#KILL HISTORY $
#COMMENT BACK TO THE BEGINNING$
#REPEAT DE$

        3
      D Y          D Y
DE :  ---- + A*---- = SIN(2X)
        3        DX
      DX

#NEED A PARTICULAR SOLUTION; BY INSPECTION K*COS(2*X)  ; IN ONE
INSTRUCTION WE WILL TELL MATHLAB TO FIND THE K$
#'SOLVE('SUB('D, D, K*COS(2*X), Y, 'DE), K)$
THE ROOTS ARE


          1
ML4 :  ----------
       -2A + 8

ONCE
FINISHED

#COMMENT CAN WRAP IT UP$
#Y:'YH+'ML4*COS(2*X)$

          -SQRT(- 1)X*SQRT(A)        SQRT(- 1)X*SQRT(A)    COS(2X)
CO + C2*E                     + C1*E                     + ----------
                                                           -2A + 8
```

```
#INVENTORY$

EXPRESSIONS : WS Y ML4 YH


EQUATIONS : HDE DE


#KILL ML4 YH HDE$
#DSKDUMP$
ML5.DMP

#↑C

.REMEMBER THE DSKDUMP WARNING!   ↑U
↑C

.RENAME DE.DMP=ML5.DMP
FILES RENAMED:
ML5.DMP

.K
CONFIRM: U
DSKC:
DE      .DMP   <055>  5. BLKS  :P
JOB 1, USER [104,337] LOGGED OFF TTY15  1102  3-AUG-71
SAVED ALL 46 FILES (2470. DISK BLOCKS)
RUNTIME 0 MIN, 13.80 SEC
```

The double-quote mark " (read quote) was introduced into MATHLAB to provide the user the ability to extend the set of common mathematical procedures available to him by name. Unquote marks ' within the scope of a quote mark " are inhibited from immediate evaluation. The intention is to forestall certain evaluations within function definitions so that they take place when the function is evaluated, not when it is defined.

We shall present three small examples. For a more heroic exercise in what can be done without conditional branching, GOTO statements, or DO loops, see the Runge-Kutta example in [2].

Example 1. A Differential Operator.

```
#LAPLACE(U):"('SIMPLIFY
('DERIV(U,X,2) + 'DERIV(U,Y,2) + 'DERIV(U,Z,2) )  )$

WS(U) :  'SIMPLIFY('DERIV(U,X,2) + 'DERIV(U,Y,2) + 'DERIV(U,Z,2) )

#'LAPLACE(X↑2+Y↑2+Z↑2)$

6
```

Example 2. Definite Integration.

The user can take the system function INTEGRATE which computes indefinite integrals and use it to define a function, which we shall call DI, which computes definite integrals. This will only work, though, for those cases where the substitution of the limits into the

indefinite integral involves no real limit computation. Note the internal assignment statement, V: ..., introduced to save the machine from computing the indefinite integral twice.

```
#DI(LOWER,UPPER,EXPR,DUMMY):
"('SIMPLIFY(
'SUBSTITUTE(UPPER,DUMMY,V:'INTEGRATE(EXPR,DUMMY) )
- 'SUBSTITUTE(LOWER,DUMMY,'V)  ))$

   WS(LOWER,UPPER,EXPR,DUMMY)
:
   'SIMPLIFY
         'SUBSTITUTE(UPPER,DUMMY,V : 'INTEGRATE(EXPR,DUMMY))
      +
         - 'SUBSTITUTE(LOWER,DUMMY,'V)

#'DI(0,1,X/(1+X↑2),X)$

1/2*LOG(2)
```

Example 3. Eigenvalues of a Matrix.

It is very easy to introduce a function which finds the eigenvalues of a matrix and reports each accompanied by the dimension of its associated eigenspace. The success of the function depends of the luck encountered by the polynomial factorization program.

```
#EIGEN(MAT):"('SOLVE('MXCHARPOL(MAT,Z)=0,Z))$

WS(MAT) :  'SOLVE('MXCHARPOL(MAT,Z)= 0    ,Z)

#REPEAT A$
      ( 1  0  0  E )
      (            )
      ( 0  1  0  0 )
A :   (            )
      ( 0  0  1  0 )
      (            )
      ( 1  0  0  1 )
```

```
#'EIGEN('A)$
THE ROOTS ARE

ML1 :   1

TWICE AND
ML2 :   1 + SQRT(E)

ONCE
AND

ML3 :   1 - SQRT(E)

ONCE
FINISHED

#
```

SECTION XI

HINTS

1. Do not type 'WS$ just to find out what is in Workspace. If SIMP is ON, you might get a different answer. In any case, since 'WS$ is an instruction, its value will be placed at the top of History, causing it to appear twice. Use REPEAT$

2. If all you want to do is differentiate something, i.e., if your instruction looks like 'DERIV(...)$, in distinction to 'DERIV(...) + ... $, turn SIMP OFF. DERIV always simplifies anyhow and having SIMP on causes redundant effort.

3. Think of RATSIMP as an alternative to SIMP. Not only is it expert at division, but it will often collect terms better than SIMP and is usually much faster. Contrast, for example,

```
#SIMP ON $
#DISTEXP 20$
#(X+Y)↑20$
```

and

```
#SIMP OFF$
#'RATSIMP((X+Y)↑20)$
```

The latter is four or five times faster.

4. Think of using RATSIMP with non-rational expressions for the optional arguments. Cf. the line #'RATSIMP('WS,E↑(X*L),L)$ in Section X_a.

5. Turn SIMP OFF when defining functions.

6. If time-sharing is precarious, use DSKDUMP frequently. It's built for speed. Do clean up the disk afterwards.

7. When performing large computations, care must be taken to control the process. Among the more expensive mistakes are the unnecessary expansion of powers of sums, the premature substitution of complex expressions for variables, and the failure to prohibit the simplification routines from looking for combinations among terms that can be seen, on some a priori grounds, not to combine.

8. WHAT TO DO WHEN YOU RUN OUT OF CORE:

Sooner or later the dread message will come:

```
NO FREE STG LEFT
BACKTRACE

#
```

What should you try before giving up?

a) Go back to 7.

b) The display of a complex expression can be a large computation in its own right. It is not uncommon for the program to succeed in computing and storing the value of an instruction but then fail in the display. This can be detected in several ways:

1) The machine printed a line-feed in anticipation of a display.

2) If the instruction were an assignment statement with a new variable name, that name's showing up in an INVENTORY.

3) REPEAT not showing the prior Workspace but, instead, running out of core once more.

If this is the case, use STORE to give the value a name and then use the remedies below, but only to display the answer, not to recompute it.

c) Use INVENTORY and KILL to throw away what you do not need. Try again.

d) If c) fails, use DSKDUMP and then use KILL to remove from core what you do not need for the immediate computation. Then use DSKLOAD to retrieve the other data from the disk.

e) Get more core. See Section XII.

SECTION XII

MATHLAB WITHIN PDP-10 TIMESHARING

NOTATION:) means carriage-return; $ means 'alt-mode'; ↑x, for any character x, means control-x, i.e., x struck while holding the CTRL button.

We shall assume familiarity with PDP-10 time-sharing in general so that no instructions on how to log on or off or how to list your file directory or rename or delete files will be given.

SIMPLEST INSTRUCTION ON HOW TO START MATHLAB:

If MATHLAB is in SYS:

.R MATHLAB)

If in your DSK: area

.RU MATHLAB)

The machine will take a while loading the program onto the drum. It will then print a few)'s. You then type (MATHLAB))

It replies

MEANS MATHLAB IS LISTENING

#

and you are off.

SIMPLEST INSTRUCTION ON HOW TO LEAVE MATHLAB AT ANY TIME:

↑C↑C

MORE COMPLEX INSTRUCTIONS ON HOW TO COMMUTE BETWEEN THE MONITOR, LISP, AND MATHLAB:

MONITOR→LISP:

Starting fresh:

R MATHLAB↵ OR RU MATHLAB↵

when the program loads, indicated by a) , you are in LISP.

Back to an old LISP (if applicable):

REE↵

MONITOR→MATHLAB:

Starting fresh, go to LISP. Then type:

(MATHLAB)

It will respond:

MEANS MATHLAB IS LISTENING

#

Back to an old MATHLAB (if appropriate):

.CONT↵

$

MATHLAB will respond with

#

LISP→MATHLAB:

(MATHLAB)

If there has been no previous MATHLAB in this LISP, MATHLAB will type:

MEANS MATHLAB IS LISTENING

#

If there has been a previous MATHLAB, MATHLAB will type:

#

and you will be in the old MATHLAB.

MATHLAB→LISP

Note that EV$ will evaluate one LISP S-expression for you. If you choose for that expression:

(PROG() A (PRINT(EVAL(READ)))(GO A)), you will, in effect, be in LISP until the occurence of a LISP error which will take you back to MATHLAB.

Much simpler route to LISP:

↑G

LISP→MONITOR:

↑C

MATHLAB→MONITOR:

↑C

HOW TO ERASE, SQUELCH, AND INTERRUPT:

	MONITOR	LISP	MATHLAB
HOW TO ERASE INPUT IN PROCESS			
n characters in current line	n rubouts	n rubouts	n rubouts
entire current line	↑U	↑U	↑U
entire input going back across ⏎'s	N/A	↑G	.⏎ (period carriage-return)
HOW TO SQUELCH OUTPUT IN PROCESS	↑0↑C	↑0	↑0 $
HOW TO INTERRUPT WHEN MACHINE IS NOT LISTENING	↑C	↑C MONITOR says: . You say REE⏎ LISP says: ⏎	a little difficult; You say: ↑C ↑C Monitor says: . You say: REE⏎ LISP says: ⏎ You say: (MATHLAB)⏎ MATHLAB says: #

HOW TO EXPAND CORE:

It is possible for the MATHLAB user to obtain more core from the monitor. This core will be automatically allocated among the various data areas of the underlying LISP system, with almost 3/4* of the additional space available for MATHLAB computation and storage. Of the 59K standard MATHLAB, about 10K is so available. Expanding core from 59K to, say, 75K would then more than double your working room.

The procedure is:

From either MATHLAB or LISP, you say:

↑C

The monitor says:

.

You say:

COR↵

The monitor says:

59 + 0/nK CORE

.

where n is a positive integer, the total permissible. Choose k such that $59 < k \leq n$. Then type:

COR k↵

Monitor says:

.

* It is possible to arrange for a larger fraction to be made available. The LISP location ACHLOC contains the instruction ASH A, -2. Increase "2".

You say:

REE↵

LISP says:

↵

You say:

(MATHLAB)↵

MATHLAB says:

#

The other users say:

@#$↑↓< #

SECTION XIII

CONSTRUCTING MATHLAB

Our intention is to distribute MATHLAB through DECUS in two extreme forms. One is a SAVEd program, ready to run. All you need is access to 59K of core on a standard PDP-10 time-sharing system and you are in business. At the other extreme, you can obtain MATHLAB in the form of a build-it-yourself kit consisting of the source (LISP) code for MATHLAB and, for safety's sake, of the source (MACRO) code for the exact version of LISP we employed and the source (LISP) code for CMP47X, a LISP compiler due to B. W. Diffie[1] which was used to compile most of MATHLAB. The kit form should serve as a starting point for those who wish to delete parts of MATHLAB or to augment or modify its current capabilities. We now give as detailed a description as we can reconstruct from our notes of how we built MATHLAB.

GET A LISP

If you have one, you can try it. If it does not work, obtain a copy of the LISP 1.6 manual distributed by DECUS. Ours is dated September 1969 and was authored by Lynn H. Quam. We used the source files we are supplying with the kit and the instructions appearing in an appendix to

---

[1] Diffie has compilers which represent improvements over CMP47X. Prospective LISP users should watch for a DECUS release.

the LISP manual entitled, CONSTRUCTION OF THE DISK BASED

LISP SYSTEM, except:

Step 1): omit ALVINE←ALVINE.MAC but add BIGNUM←BIGNUM.MAC.

Steps 2) through 4):

We used the sequence:

.R LOADER)

*LOADER$

It responds and exits.

.R LOADER)

*LISP,/J SYMMAK$

It responds and exits.

.SAVE DSK:LISP 10)

Job saved.

Step 5):

Expect LISP.SAV instead of LISP.DMP.

Steps 6) through 10):

Omit.

GET A COMPILER

Route 1: You may try one you have. But first,

(INC(INPUT DSK: (MACROS.L2)))

Save a copy with this added.

Route 2: Use CMP47X to compile itself.

Take a LISP of about 30K, very little Binary Program Space, no decimals or LAP, and:

(INC(INPUT DSK: CMP47X))↵

Machine responds; go to Route 3:

Route 3: Use any compiler to compile CMP47X.

(COMPL CMP47X)↵

This will be very slow if you got here via Route 2. When done, exit to the monitor. You have just created a file named CMP47X. LAP.

In the following conversation, ␣ denotes a space.

```
.R LISP 30
ALLOC? Y
FULL WDS=3000␣
BIN. PROG. SP=14000␣
SPEC. PDL= ␣
REG. PDL= ␣
HASH=377 ␣
AUXILIARY FILES? Y
SMILE?␣
ALVINE?␣
TRACE?␣
LAP? Y
DECIMAL? N
```

STANFORD AI LISP 1.6 21-NOV-69

* (INC(INPUT DSK: (CMP47X.LAP)))↵

Machine responds.

* (INC(INPUT DSK: (MACROS.L2)))↵

Machine responds.

↑C

.SAVE CMP47X

You have just created and saved a compiler whose name is CMP47X.SAV

COMPILE THE MATHLAB SOURCE CODE

Talking to the monitor:

RU (or R) CMP47X (or whatever compiler you are using)↵

LISP says:

*

You say

(COMPL (BPRE1.L))↵

Machine responds at length, ending with a *. You have just created a file named BPRE1.LAP. Exit and start fresh:

↑C

.RU CMP47X

*(COMPL(DISP9.L))

etc.

This process, starting each time with a fresh compiler, is repeated for the following files:

BSIMP1.L

DIF8.L

BRATS1.L

FACT6.L

MLAB37.L

SOL6.L

INT2.L

TOP9.L

CELT6.L

ILT2.L

LDE6.L

MAT11.L

BUILD A BIG BIGNUM LISP

Basically we wish to allocate a LISP large enough for MATHLAB and follow the procedure delineated in APPENDIX I of the LISP 1.6 Manual for the incorporation of the arbitrary precision integer arithmetic package. Our notes suggest that at this point instruction 5) -- Copy to SYS: , etc. -- found in the LISP 1.6 Manual appendix entitled CONSTRUCTION OF THE DISK BASED LISP SYSTEM is of some import. Probably the loader looks in SYS: for LISP.SYM when assembling the BIGNUMs. If

moving files to SYS: is prohibited, it can be faked by:

```
.AS DSK: SYS:)
```

although this does have side effects, e.g., the DIR command no longer works.

The procedure is

```
.AS DSK:  SYS:)
DSK ASSIGNED
.R LISP 58)
ALLOC? Y
FULL WDS=4400␣
BIN. PROG. SP=60000␣
SPEC. PDL=2000␣
REG. PDL=2000␣
HASH=␣
AUXILIARY FILES? Y
SMILE?␣
ALVINE?␣
TRACE?␣
LAP? Y
DECIMAL?␣
STANFORD AI LISP 1.6          21-NOV-69
*(INC(INPUT SYS:  (BIGNUM. LSP)))
NIL
BIGINIT
APNINIT
```

```
NIL
*(LOAD T)
**SYS:BIGNUMS$
LOADER 2K CORE
*(APNINIT)
NIL
*↑C
·SAVE BILISP
You have just created a file named BILISP.SAV
```

ASSEMBLE MATHLAB

There is a sequence in the procedure below which reads:

```
*(EXAMINE 6457)
51040006455
*(DEPOSIT 6457 203000000000)
203000000000
```

The purpose of this sequence is to replace the instruction which causes LISP to print a * on each carriage return -- which would be quite confusing in the middle of MATHLAB input -- with a no-op. If the result of the EXAMINE is anything but 51040006455, your LISP has assembled the pertinent instruction in a different location -- which you have to find -- than has the LISP used to build MATHLAB. The actual instruction you will be looking for will be of the form 51040xxxxxx where xxxxxx is an address two less than the address of the instruction itself. If your LISP differs from the LISP.MAC supplied with MATHLAB in only a few places, you could look for the instruction in the vicinity of 6457.

In general, you must replace the location in which the line

```
TTYUU0 1,  ["*"]; OUTPUT *
```

(which appears five text lines down from the label TTYI:) assembled with a no-op. Note, however, that the text line in question is included within the brackets of

```
JRST            [TALK ...
        ...
        ...
        ...
        ...]
```

and is hence assembled out-of-line.

The procedure for building MATHLAB from BILISP.SAV is:

```
.RU BILISP↵
*(INC(INPUT DSK:(MACROS.L2)))↵
```

Machine responds and you continue

```
*(INC(INPUT DSK:(BHAND1.LAP) (BPRE1.LAP) (DISP9.LAP)))↵
```

Machine responds and you continue in a similar vein until you have assembled, in addition to the above files,

BSIMP1.LAP

DIF8.LAP

BRATS1.LAP

FACT6.LAP

MLAB37.LAP

SOL6.LAP

INT2.LAP

TOP9.LAP

CELT6.LAP

ILT2.LAP

LDE6.LAP

MAT11.LAP

Then,

*(EXAMINE 6457)↵

51040006455

*(DEPOSIT 6457 203000000000)↵

203000000000

(SETQ *NOPOINT(SETQ BASE(SETQ IBASE(PLUS 5 5))))↵

10

(NOUUO NIL)↵

T

(LINELENGTH 72)↵

72

↑C

.SAVE MATHLAB↵

You have just created a file named MATHLA.SAV that will be called whenever you say RU (or R) MATHLAB

SECTION XIV

REFERENCES

[1] Engelman, C., MATHLAB 68. In A. J. H. Morrell (Ed.), Information Processing 68, North-Holland, Amsterdam, 1969, pp. 462-467.

[2] Engelman, C., THE LEGACY OF MATHLAB 68. Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation, Los Angeles, 23-25 March 1971. (ACM, New York). pp 29-41.

[3] Manove M., Bloom, S., and Engelman, C., Rational Functions in MATHLAB. In Bobrow, D. G., (Ed.), Symbolic Manipulation Languages and Techniques, North Holland, Amsterdam, 1968, pp 86-102.

[4] Millen, J. K., CHARYBDIS: A LISP Program to Display Mathematical Expressions on Typewriter-like Devices. In Klerer, M., and Reinfelds, J., (Eds.), Interactive Systems for Experimental Applied Mathematics, Academic Press, New York and London, 1968, pp. 155-163.

SECTION XV

INDEX