ESD-TR-70-256

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LINCOLN LABORATORY

THE LEAP USER'S MANUAL

LINCOLN MANUAL 93

ESD ACCESSION LIST
Call No. 71233
Copy No. 1 of 1 cys.

P. D. ROVNER
Group 23

11 September 1970

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Advanced Research Projects Agency of the Department of Defense under Air Force Contract AF 19(628)-5167 (ARPA Order 691).

This report may be reproduced to satisfy needs of U.S. Government agencies.

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

LEXINGTON MASSACHUSETTS

AD0713221

ABSTRACT

This document is a user's manual for the LEAP language. LEAP is an extended algebraic programming language which is similar in form to ALGOL.[8] Extensions include language forms for display output and interactive input and facilities for building and manipulating associative information structures. The basic algebraic language is described in Sections I through IX; the extensions to LEAP are presented in the Appendices.

Accepted for the Air Force
Joseph R. Waterman, Lt. Col., USAF
Chief, Lincoln Laboratory Project Office

CONTENTS

Appendices

I. VARIABLES

One may declare and use VARIABLES in LEAP. A variable is an entity which has a NAME, a DATA TYPE, and a VALUE. The NAME of a variable must consist only of alphanumeric characters and must start with a letter. The number of characters allowed in a name is unlimited. The DATA TYPE of a variable must be one of the following data types:

REAL
INTEGER
BOOLEAN
FIXED (i.e. fixed point fraction)
MATRIX
TEXTARRAY
REAL
INTEGER
BOOLEAN
FIXED
} ARRAY

The VALUE of a variable is an algebraic quantity having the specified data type. For example, if X were an INTEGER variable, it might have 46 as its value. If Y were a BOOLEAN ARRAY, it would have an array of BOOLEAN numbers as its value.

A. DECLARATIONS

All variables must be declared. The declaration of a variable may occur either at the beginning of the LEAP program or at the beginning of the outermost COMPOUND STATEMENT within which the variable is used (see the discussion of COMPOUND STATEMENTS in Section V.E). A typical declaration has a data type specification , a list of names, and a semicolon. Examples:

```
REAL X, Y, Z;
INTEGER ARRAY A, B;
```

A dynamic variable (a MATRIX, ARRAY or TEXTARRAY) may be declared with information about its dimensions; for a complete discussion of dynamic variables, see Section III.

II. CONSTANTS

Integer constants are converted to either radix 8 or 10, depending on their form. Including sign, integer constants consist of 36 bits, floating point constants of 27 bits of mantissa and 9 bits of characteristic, and fractions of 36 bits. Omission of a preceding sign indicates a positive number.

1. Decimal INTEGER constants are expressed by 1 to 11 digits written without a decimal point.

Examples:

```
3
527
-321
923
```

2. Octal INTEGER constants are expressed by 1 to 12 octal digits and are written with a terminal decimal point.

Examples:

```
5.
7.
770770770777.
```

3. REAL (i.e., floating point) constants are expressed in two ways, either by digits both before and after the decimal point (for example, 3.5 or -0.3), or by the exponential designation with an optional decimal point:

Examples:

```
-2E-3          equals  -0.002
.2E7           equals  2,000,000.0
2.E10          equals  20,000,000,000.0
```

4. FIXED (i.e., decimal fraction) constants are expressed by a decimal point followed by 1 to 10 digits:

Examples:

```
.2
.37
.002
```

5. There is no facility for octal fraction constants in LEAP.

6. BOOLEAN constants are expressed as either "TRUE" or "FALSE" (Note: this is not valid for typed input to a READ statement).

THUS:

35.0	is	REAL
35	is	decimal INTEGER
35.	is	octal INTEGER
.35	is	FIXED
TRUE	is	BOOLEAN

III. DYNAMIC VARIABLES

A. ARRAYS

An ARRAY is an ordered collection of ELEMENTS. A particular array element is indicated by specifying a unique subscript for the element, as illustrated below:

(1) $A_{E1, E2, E3, \ldots, En}$

In (1), the "Ei" are any INTEGER expressions, "n" is the number of dimensions of the array, and A is the name of the array.

Each array element has a value. The data type of the elements of an array is specified when the array is declared (e.g., REAL ARRAY A;).

An array may be declared with size and dimension information; if this information is specified, then storage will be allocated at program execution time for the array elements. If this information is not specified, then no storage will be allocated until a statement is executed which explicitly assigns storage to the array for its elements (see the discussion of the assignment statement in section V·A.). The following is the form for an array declaration with size and dimension information:

(2) (type) ARRAY (name) $\{a_1$ to a_2, b_1 to b_2, . . ., z_1 to $z_2\}$;

In (2), (type) is either REAL, INTEGER, BOOLEAN, or FIXED. The (name) is the name of the array. The other parameters are explained below:

a_1 is the lower bound on the first dimension (if there is to be only one dimension, then a_1 must equal 1)

a_2 is the upper bound on the first dimension

b_1 is the lower bound on the second dimension

b_2 is the upper bound on the second dimension, etc.

There is no limit on the number of dimensions, and the bounds may be any INTEGER expressions.

B. TEXTARRAYS

A TEXTARRAY is a single dimensional array of characters, each represented by its integer character code. Like the ARRAY, a TEXTARRAY may be declared with information about its size (the maximum number of characters

in the TEXTARRAY, including the 777. character);

(3) TEXTARRAY (name) AE ;

If no size information is given, then no storage will be allocated for the TEXTARRAY elements by the declaration. This storage will be allocated only by a subsequent assignment statement. In (3), (name) is the name of the TEXTARRAY, and AE is an INTEGER expression specifying the size of the TEXTARRAY.

A TEXTARRAY element is indicated by specifying its subscript:

Examples:

IF TA_I = 777. THEN ...

777. → TA_J ;

C. MATRICES

The MATRIX in LEAP is a highly specialized entity. It always has two dimensions, and its elements are always REAL numbers. Only one exponent is kept for all the elements; the elements are scaled appropriately. Thus, information is lost if the values of elements differ by too many orders of magnitude.

Matrices may be declared with no information about the number of rows and columns (e.g., MATRIX(name);), or with such information given:

(4) MATRIX (name) a_1 BY b_1 ;

If no dimension information is specified, then no storage will be allocated for the matrix elements by the matrix declaration. As in the case of the array, this storage will be allocated only when an assignment statement

explicitly assigns storage to the matrix.

If, as in (4), dimension information is specified, then appropriate storage is allocated for the matrix, and all elements are initialized to zero. In (4), a_1 and b_1 are INTEGER expressions. The declared matrix will have a_1 rows and b_1 columns.

A matrix element is indicated by specifying the name of the matrix, the row index, and the column index. These indexes may be any INTEGER expressions between 1 and 256. Examples follow:

M (1, 2)	row 1, column 2 element of M
M (K, J + 1)	row K, column J + 1 element of M

Matrices were introduced into LEAP to implement the parametric homogeneous matrix representation for points, lines, and conics which is described in Reference 9. LEAP has facilities for multiplying, inverting, and adjoining matrices. A complete presentation of the operations which apply to matrices is given in Section IV. C.

LEAP also has a facility for generating the appropriate display instructions from a parametric homogeneous matrix description of a point, line, or conic (see Appendix I).

Note: The word USELEAP must follow START in every LEAP program in which MATRICES are used.

IV. EXPRESSIONS

Variables, constants, elements of dynamic variables, and/or EXPRESSIONS may be combined by OPERATORS (e.g. + and -) to form EXPRESSIONS. An expression has a data-type, and a value. The value is computed by performing the indicated operation. For example, if X is a REAL variable having 3.6 as its value, and Y is a REAL variable having 1.0 as its value, then

X + 4.2 x Y

is a REAL expression with 7.8 as its value.

Note that we would expect the multiplication to be done before the addition when the above expression is evaluated. In LEAP, the multiplication operator (x) is said to have "higher precedence" than the addition operator (+). We can classify the operators in LEAP by specifying their relative precedence, or "binding power." The remainder of this section is a tabulation of the operators in LEAP, organized in groups by operand type, and arranged within groups in order of decreasing precedence. Note that the expression scan is done from left to right. When operators of equal precedence are adjacent, e.g., A + B + C, then the evaluation is performed from left to right, e.g., (A + B) + C. When operators of different precedence are adjacent, the operator of higher precedence is treated first. When in doubt about precedence, parenthesize.

In what follows,

A, A1, A2, etc.	will	represent		ARRAY's
TA, TA1, TA2,	"	"	"	TEXTARRAY's
M, M1, M2,	"	"	"	MATRIX EXPRESSION's
AE, AE1, AE2,	"	"	"	ARITHMETIC EXPRESSION's
B, B1, B2,	"	"	"	BOOLEAN EXPRESSION's

A. ARITHMETIC OPERATORS

The operands for arithmetic operators are of REAL, INTEGER, or FIXED types, and may be mixed indiscriminately in expressions. The result of mixed arithmetic is always REAL.

OPERATOR	MEANING	FORM	PRECEDENCE	RESULT
superscript	exponentiation	$AE1^{AE2}$	8	AE (REAL)
subscript	ARRAY element	$A_{AE1,AE2,\ldots}$	7	B or AE
	TEXTARRAY element	TA_{AE1}	7	AE (INTEGER)
parentheses with comma	MATRIX element	M (AE1, AE2)	7	AE (REAL)
parentheses with "HAND"	SUBWORD (bits AE2 thru AE3, where bits are numbered from 1 to 36, starting at the right)	AE1 (AE2 ⌐ AE3) (Note: AE2 and AE3 are INTEGERS)	7	AE (INTEGER)
parentheses preceded by AE	bit test, TRUE if a 1	AE1 (AE2)	7	B
parentheses	expression delimiter	(AE)	7	AE ∞
FUNCTION call	(SEE SECTION VII)		7	B or AE
↑	FLOAT (convert to a REAL)	↑ AE	6	AE (REAL)
↓	ROUND (convert to an INTEGER)	↓ AE	6	AE (INTEGER)
⇓	TRUNCATE (convert to an INTEGER)	⇓ AE	6	AE (INTEGER)
↓	take the fractional part	↓ AE	6	AE (FIXED)
||	absolute value	|| AE	6	AE
+	unary plus	+AE	6	AE
−	unary minus	−AE	6	AE

OPERATOR	MEANING	FORM	PRECEDENCE	RESULT
x	multiplication	AE1 x AE2	5	AE
/	division	AE1 / AE2	5	AE
⊧	right shift (The 36 bit quantity which is the value of AE1 is shifted to the right N binary places, where N is the value of the INTEGER expression, AE2.)	AE1 ⊧ AE2	5	AE (INTEGER)
⊦	left shift	AE1 ⊦ AE2	5	AE (INTEGER)
+	addition	AE1 + AE2	4	AE
-	subtraction	AE1 - AE2	4	AE
= ≠ or ≠ < > ≤ ≥	arithmetic relations	AE1 = AE2	3	B
∧	logical intersect (AND)	AE1 ∧ AE2	2	AE (bit by bit operations)
∨	logical union (OR)	AE1 ∨ AE2	1	AE (bit by bit operations)
⊕	logical exclusive-or	AE1 ⊕ AE2	1	AE (bit by bit operations)

NOTE: TX-2 is a one's complement machine (i.e. -0 exists ... all ones).

B. BOOLEAN OPERATORS

OPERATOR	MEANING	FORM	PRECEDENCE	RESULT
$\sim$	"NOT"	$\sim$B	3	B
$\wedge$	"AND"	B1 $\wedge$ B2	2	B
$\vee$	"OR"	B1 $\vee$ B2	1	B
$\oplus$	"exclusive OR"	B1$\oplus$B2	1	B

C. MATRIX OPERATORS

OPERATORS	MEANING	FORM	PRECEDENCE	RESULTS
i	inverse	M^{i}	7	M
t	transpose	M^{t}	7	M
x	scalar multiplication	AE x M	6	M
x	matrix multiplication	M1 x M2	5	M
/	adjoin below	M1 / M2	5	M
|	adjoin to the right	M1 | M2	5	M
nR	number of rows	nR M	4	AE (INTEGER)
nC	number of columns	nC M	4	AE (INTEGER)
‖	determinant	‖ M	4	AE (REAL)
-	scalar multiply by -1	- M	3	M
+	MATRIX addition	M1 + M2	2	M
-	MATRIX subtraction	M1 - M2	2	M
=	MATRIX equality test	M1 = M2	1	B
≠ or ∤	MATRIX inequality test	M1 ≠ M2	1	B

D. MISCELLANEOUS MATRIX EXPRESSIONS

FORM	MEANING
$AE1\#_{AE2\ BY\ AE3}$	A MATRIX having AE2 rows and AE3 columns, where all elements have value AE1.
$AE1\Delta_{AE2\ BY\ AE3}$	A MATRIX having AE2 rows and AE3 columns, where all off-diagonal elements have value 0.0, and all diagonal elements have value AE1.
$M\ (AE1,\ AE2)_{AE3\ BY\ AE4}$	Submatrix of M, starting at row AE1 and column AE2, for AE3 rows and AE4 columns.

E. ARRAY AND TEXTARRAY EXPRESSIONS

FORM	MEANING
ARRAY {AE1 to AE2, AE3 to AE4, . . .}	This expression has as its value an ARRAY of the indicated dimensions, where all elements are initialized to zero. See section III A. for details. Note that single dimensional ARRAYS must have AE1 equal to 1.
TEXTARRAY {AE1}	This expression has as its value a TEXTARRAY with AE1 elements, each of which is initiatized to zero. "AE1" should be an INTEGER expression.
'ARBITRARY CHARACTER STRING, EXCLUDING QUOTE'	This is a TEXTARRAY expression.

F. TEXTARRAY OPERATORS

OPERATORS	MEANING	FORM	RESULTS
=	Equality Test	TA1 = TA2	B
≠	Inequality Test	TA1 ≠ TA2	B
‖	Count of Number of Characters (Not including 777. character)	‖ TA	AE (INTEGER)

NOTE: The last character in a TEXTARRAY should be 777_8.

G. CONDITIONAL EXPRESSION

General form: (B ⊃ E1, E2).

This expression has either E1 or E2 as its value, depending on whether the BOOLEAN expression B has value TRUE or FALSE, respectively. E1 and E2 are expressions which must have the same data type. This may be any allowed data type, including MATRIX and ARRAY, for example.

V. STATEMENTS

There are a number of imperatives (called STATEMENTS) in the LEAP language. These are used to modify the values of the program variables and the flow of control through the program. All statements in LEAP must be terminated by one of the following, depending on context:

;

END

ELSE

A. ASSIGNMENT STATEMENT

General Form: <expression> → <variable or element of a dynamic variable>;

This statement causes the value of the indicated variable to be reset to the value of the expression.

Examples:

```
REAL X, Y;
MATRIX M;
4.0 → X;
X x 2.0 → Y;
0.0 #3 by 3 → M;
1.0 → M (3, 3);
```

Data type conversions take place where required and allowed. The following table shows the allowed and resulting conversions. Blanks indicate that the conversion is not allowed.

EXPRESSION TYPE \ VARIABLE TYPE	REAL	FIXED	INTEGER	BOOL.
REAL	Real	Fixed*	Integer* (rounded)	--
FIXED	Real	Fixed	--	--
INTEGER	Real	--	Integer	--
BOOL.	--	--	--	Bool.

The assignment statement may in fact be an expression if it is nested. This facilitates multiple or intermediate stores. For example,

$$1 \rightarrow A \rightarrow B;$$

assigns the value 1 to both A and B.

The subword form may be used as a variable in an assignment statement. Example:

```
INTEGER X;
   :
 3 → X (1 ⌐ 4);
```

A special case of the assignment statement is the sub-matrix store command. Example:

$$M \times N \rightarrow M\ (3, 5);$$

The matrix expression on the left will replace the sub-matrix of M whose upper left-hand element is in row 3, column 5. If the new sub-matrix will not fit into the indicated space, an error will be indicated at run-time.

---

* No check is made for overflow: strange things may occur if a REAL number larger than or equal to 1.0 is converted to a FIXED.

B. TRANSFER-OF-CONTROL STATEMENTS

B1. Unconditional Go

General Form:

```
GO      }
GOTO    }   <statement label>;
GO TO   }
```

The GO statement causes a transfer of control to the statement indicated by the "STATEMENT LABEL." A STATEMENT LABEL is a sequence of alphanumeric characters, starting with a letter, which is assigned to a statement by prefacing the statement with <statement label> ☛.

Example:

```
        1.0 → X;
L1 ☛ X + 1.0 → X;
        :
        :
    GO TO L1;
```

B2. Conditional GO Statement

General Form:

```
GO      }
GOTO    }   <B> ⊃ <label 1>, <label 2>;
GO TO   }
```

This statement causes control to go to either statement label 1 or statement label 2, depending on whether the BOOLEAN expression is true or false.

B3. Switch Statement

General Form: SWITCH VIA <INTEGER expression> TO <list of statement labels>;

This statement causes a transfer of control to the statement label indicated by the value of the INTEGER expression. If this value is out of bounds, an error message will be given.

Example:

```
INTEGER I;
    :
    :
SWITCH VIA I TO L1, L2, L3;
```

If I = 1, then control will go to L1.

If I = 2, then control will go to L2.

If I = 3, then control will go to L3.

C. IF STATEMENTS

General Forms: (1) IF <B> THEN <statement 1> ELSE <statement 2>;

If the BOOLEAN expression is true, <statement 1> is executed; if it is false, <statement 2> is executed. If there is a "dangling ELSE" clause, it is associated with the innermost IF clause. Example (1a and 1b are equivalent):

1a.
```
IF <B1>THEN
IF <B2> THEN
    < statement 1>
ELSE
    <statement 2>;
```

1b.
```
IF <B1> THEN
  BEGIN
    IF < B2> THEN
      <statement 1>
    ELSE<statement 2>
  END;
```

(2) IF <B> THEN <statement>;

the <statement> is executed only if the BOOLEAN expression is true.

The word IFNOT may be used instead of IF in the above forms; in this case, the BOOLEAN expression is complemented, and then examined.

D. ITERATION STATEMENTS

General Forms: (1) FOR E_1 → P STEP E_2 { TO / THRU } E_3 DO S;

where E_1, E_2, E_3 are arithmetic expressions, P is a non-dynamic variable or an array element, and S is a statement.

This statement causes statement S to be executed once for each new value of P, the iteration variable. The statement is executed as if it were written as:

```
        E1 → P;
L1      IF ‖ P > ‖ E3  THEN GOTO L2;  (see note 1 below)
        S;
        P + E2 → P;
        GOTO L1
L2      .
        .
        .
```

(2) FOR E_1 → P STEP E_2 { WHILE / UNTIL } B DO S;

where E_1, E_2, E_3, P and S are as above, and B is any Boolean expression. Execution of this statement is analogous to the previous statement. Executions of statement S continue as long as:

(a) B is true (WHILE)
(b) B is false (UNTIL)

(3) FOR E_1 → P { WHILE / UNTIL } B DO S;

where E_1, P, B, and S are as above. This statement behaves as indicated in (2) above, except that the iteration variable is not incremented.

(4) { WHILE / UNTIL } B DO S;

where B and S are as above. This statement behaves as type (3), but has no iteration variable.

CONTINUE STATEMENT

This is a statement which causes a jump to either the incrementing or testing part of the FOR statement when execution of the remaining body is not desired.

Example:

```
FOR 1 → P STEP 1 TO 10 DO
BEGIN IF P = 7 THEN CONTINUE:
                .
                .
END;
```

would cause execution for values of P = 1 through 6, 8 and 9.

Note 1: For TO, this operator is ≥; for THRU, the operator is >. If the iteration variable changes sign or ever equals zero, then another form of the FOR statement should be used.

E. COMPOUND STATEMENT

It is often desirable to have a number of statements act as a single statement. A group of statements which is preceded by the word BEGIN and followed by the word END is called a COMPOUND STATEMENT. Note that compound statements may be nested.

Compound statements may have "local" declarations of non-dynamic variables (of types REAL, INTEGER, BOOLEAN, and FIXED) immediately following the word BEGIN. These variables are "local" in the sense that they may not be referenced from outside of the compound statement, but they may be referenced anywhere between the current BEGIN-END parentheses. The NAMEs of these variables may have been used in an outer compound statement or in the main program declarations. In this case, a NAME always refers to the variable declared in the current innermost compound statement. Note that one may GO into the middle of a compound statement.

F. BLOCKS

A compound statement in which dynamic variables are declared is called a BLOCK. Iteration statements, [A] . . . [E] statements (see appendix 2), and PROCEDURES (see section VII) are also BLOCKS. One may not GO into the middle of a BLOCK.

VI. COMMENTS

Comments may occur anywhere in a program where a statement or declaration may occur. Comments begin with the word COMMENT, and end with a semi-colon. Any string of characters (excluding semi-colon) may appear in between.

VII. PROCEDURES

A PROCEDURE is a subroutine which may or may not expect input parameters and may or may not return a result. A PROCEDURE must be declared before it is called. A PROCEDURE declaration must occur in a declaration portion of the LEAP program (see section IX) in one of the

following forms:

(1) <REAL, INTEGER, BOOLEAN, or FIXED> PROCEDURE <name of procedure> <plist>; <statement>;

(2) PROCEDURE <name of procedure> <plist>; <statement>;

In the above, the <name> is any string of alphanumeric characters, starting with a letter. The <plist> is a list of "parameter declarations," separated by semi-colons, preceded by {, and followed by }. If the procedure takes no parameters, the <plist> is absent. A "parameter declaration" consists of a data type specification, followed by a list of names which are separated by commas.

For example, the declaration of a PROCEDURE to find the largest number in an array and store it in a specified cell would look like this:

```
PROCEDURE BIG {INTEGER ARRAY A; INTEGER AM, AB};
BEGIN INTEGER I;
A[1] → AB;
FOR 2 → I STEP 1 UNTIL I > AM DO
IF A[I] > AB THEN A[I] → AB;
END;
```

In this procedure, A, AM, and AB are procedure parameters. They represent the true arguments given the procedure when the procedure is "called." Two additional declarations are allowed in a procedure declaration to describe arguments. They are:

LABEL L_1, L_2, . . . L_n;

and

(type) PROCEDURE P_1, P_2, P_3; (Again, type is optional)

Examples:
```
REAL PROCEDURE PYTHAG {REAL A, B};
INTEGER PROCEDURE AVG {INTEGER I, J}
PROCEDURE TEST {REAL PROCEDURE P; LABEL TAG};
```

A procedure "call" may occur as a statement or an expression depending on whether a data type precedes the word PROCEDURE in the procedure

declaration. A procedure which is to be used as an expression is called a FUNCTION. The procedure call has the following general form:

<procedure name> <a list>

The <a list> is a list of expressions, variables and elements of dynamic variables, separated by commas, preceded by {, and followed by }. If the procedure takes no parameters, the <a list> is absent.

The data type of each element in the <a list> is compared with the declared data type of the corresponding element in the <p list>, and an error is given if these do not match. For example, the following is a statement calling the procedure declared above:

BIG {LIST, 100, LARGLST};

where LIST is the name of the array, 100 is the maximum size, and LARGLST will contain the largest element after the procedure is called. Note that there are two kinds of parameters in the above example:

(1) parameters which are not changed by the action of the procedure, but whose values are used (VALUE parameters: LIST and 100, for example)

(2) parameters whose values are changed by the action of the procedure (REFERENCE parameters: LARGLST, in this case).

In LEAP all variables and dynamic variables may be passed to procedures as REFERENCE parameters; also, elements of ARRAYS may be passed as REFERENCE parameters. However, TEXTARRAY elements, subword expressions, and MATRIX elements may NOT be passed as REFERENCE parameters to procedures.

VIII. RETURN STATEMENTS

Normally, procedures and functions return to the calling statement at completion. However, an additional statement is provided to cause the procedure or function to return from anywhere within the procedure body.

General Form: RETURN E;

where E is required for functions and not allowed for other procedures.

E must be of the same data type as the function. This statement causes the procedure to return to the calling statement. If the procedure is a function, then the function value is E.

Example:

```
 FUNCTION  | REAL PROCEDURE LARGEST {REAL X, Y};
DECLARATION| IF X > Y THEN RETURN X ELSE RETURN Y;
                          :
 FUNCTION  |
           | LARGEST {4.0/A, 2.0/B} + 5.0 → A;
   CALL    |
                          :
```

IX. PROGRAM LAYOUT

Each LEAP program must start with the word START and finish with the word FINISH. The remainder of the program consists of two separate parts: a sequence of declarations, followed by a sequence of statements.

Example:

```
START
REAL X, Y, Z;
ARRAY A {1 to 40};
        .
        .
        .
L⌐ X + 3.0 → Y;
        :
        :
 GOTO   L;
FINISH
```

APPENDIX I

PRIMITIVES FOR DISPLAY OUTPUT

The display output facility in LEAP consists entirely of a collection of library procedures for constructing and modifying a "picture data structure". The picture on the screen at the console is generated by a display processor* which accesses and interprets picture-drawing commands from this picture data structure. Typical commands to the display processor are:

1) Place a dot at a specified position** on the screen.

2) Draw a line or conic segment from a specified screen position with a specified slope and length.

3) Display specified text starting at a specified screen position.

4) Call a "display subroutine", to be centered at a specified position relative to the current frame of reference.

The "picture data structure" is simply a collection of display subroutines (called GROUPS), each having a unique 16-bit integer identifier (ID). Each display subroutine (GROUP) consists of a collection of display ITEMS, each having a 16-bit integer identifier (ID) which is unique within that collection of items. There are two kinds of display items:

1) A linear sequence of commands for drawing simple picture fragments and moving the beam, and

---

* Effectively a separate, special purpose computer (see reference 1).

** All positions are REAL expressions, ranging from -1.0 to +1.0.

2) a "use" of a display subroutine, which causes the indicated picture to be displayed as a subpicture of the group.

The library of procedures for constructing and modifying display groups and items is tabulated below. Note the facilities for blanking items, drawing dotted lines, moving the unintensified beam, deleting groups and items, and creating uses. Groups are created automatically when required: e. g. when a use is made of a non-existent group; when an item is "put" into a non-existent group. When a group is deleted, all uses of it are automatically deleted. Display subroutines (groups) are not re-entrant: the "structure" of the picture resembles a tree.

One creates the first kind of display item as follows:

1) Declare the ID of the display item (a 16-bit integer) with a "SETITEM" call,

2) Put points, lines, conics, and/or text into the display item via PUTPNT, PUTLINE,* PUTMAT**, and PUTTEXT calls, and

3) Put the display item into a group via the PUTITEM call. (If the display item is put into group zero, it will be displayed.)

---

* The line will be drawn from the last position of the beam.

** The PUTMAT routine expects as input the parametric homogeneous matrix representation of a point, a line, or a conic. For further information about matrix representations of picture parts, see Reference 9.

As an example of a LEAP program which uses the display output facility, we have written down a program to display the scope diagonals:

```
START
        CLEARSCOPE;
        SETITEM {1};
        LOADPNT {-1.0, -1.0};
        PUTLINE {2.0, 2.0};
        PUTITEM {0};
        SETITEM {2};
        LOADPNT {1.0, -1.0};
        PUTLINE {-2.0, 2.0};
        PUTITEM {0};
FINISH
```

RESERVED PROCEDURES FOR DISPLAY OUTPUT

PROCEDURE NAME	PARAMETERS	
CLEARSCOPE	NONE	(initialize the display structure)
TURNOFF	NONE	
SETITEM	{<display item ID>}	
PUTPNT	{<X position>, <Y position>}	
PUTLINE	{<Δ X>, <Δ Y>}	(REAL expressions)
PUTMAT	{<MATRIX expression>}	
PUTTEXT	{<TEXTARRAY>, <X-position>, <Y-position>}	
LOADPNT	{<X position>, <Y position>}	(position the beam, but don't intensify)
LOADLINE	{<ΔX>, <ΔY>}	(move the beam, but don't intensify)
PUTDOTLINE	{<Δ X>, <Δ Y>}	(draw a dotted line)
PUTDOTMAT	{<MATRIX expression>}	(draw a dotted conic)
SETPENMODE	{<display group ID.><PEN MODE: 0,1,2, or 3>}	(see APPENDIX III)
RESETITEMID	{<display item ID>}	(similar to SETITEM, except the display item buffer is not cleared)
PUTSD	{<36 bit INTEGER>}	(the indicated word is appended to the current display item buffer)
STOREITEM	NONE	(the current display item buffer is sent to the storage tube)

PROCEDURE NAME	PARAMETERS	
PUTITEM	{<display group ID>}	
PUTITEMONCE	{<display group ID>}	(the item is displayed once, then blanked)
BLANKITEM	{< display item ID> , <display group ID>}	
UNBLANKITEM	{<display item ID> , <display group ID>}	
DELITEM	{<display item ID>, , <display group ID>}	(delete a display item)
DELGRP	{<display group ID> }	(delete a display group)
USEGRP	{<ID of prototype group>, <ID of instance>, <ID of host group>, <X center>, <Y center>}	
USEGRPONCE	{<ID of prototype group>, <ID of instance>, < ID of host group>, <X center>, <Y center>}*	
MOVUSE	{<display item ID>, <display group ID>, <new X center>, <new Y center>}	

* The instance is displayed once, then blanked.

NOTES

(1) The X and Y coordinates of the display run from -1.0 to +1.0

(2) All ID's are INTEGER expressions

(3) All positions are REAL expressions

(4) The PUTITEM procedure does not re-initialize the display item buffer. This implies that one may build a display item and copy it into more than one group. Also, one may build a display item, copy it into a group, then add more to the display item, copy it into a group and so forth.

APPENDIX II
THE ASSEMBLY CODE OPTION

A. General Description

A brief version of TX-2 assembly code has been implemented in LEAP allowing the assembly and execution of machine code in LEAP programs. The current implementation has no macro facility.

B. Format

To begin assembly coding, the user types [A] . This character causes the compiler to look for MARK 5* information. [E] marks the end of the assembly information and the compiler returns to normal LEAP processing. The form [A] --- [E] ; is equivalent to a statement in LEAP.

C. Restrictions and Notes

1. Equalities are permissible, but all symexes used in forming the equality must be defined.

2. The special symexes A, B, C, D, E are not automatically available, although they may be defined as equalities by the user.

3. Configs, hold bits, bit instructions, double indexing, and RC's are allowed. When defining a bit, however, it is necessary to separate the quarter-bit number by a comma (not a period).

Example: $SKN_{4,3}$ YB

Configs and subscripts must also be single symexes.

4. When reference is made to a LEAP variable, the address of the variable is used. This means that in the normal case LDA Q will put the value of Q (a LEAP variable) in A.

5. All MARK 5 equalities and instructions must end with semicolons, except for the last where [E] is used.

6. Forward references are allowed in restricted cases. These are:

---

* MARK5 is the assembler for TX-2.

a) No operation is performed on the symex.

b) The symex is defined later by a '⌐' in LEAP or a '⌐' or '¬' in MARK 5.

7. Tags are assigned by use of a '⌐' or '¬' followed by a MARK 5 instruction, constant, etc.

8. There is no comma convention and constants follow the rules of LEAP. Octal integers must therefore be followed by a decimal point.

Example: JED 56· and ^{21}· LDA X

9. One may not refer to a label or equality which has been defined in [A] . . . [E] statement from anywhere outside that statement (e.g., equalities are "local" to the [A] . . . [E] statement in which they are defined.

APPENDIX III

PRIMITIVES FOR INTERACTIVE INPUT

The facility for non-typewritten interactive input to a LEAP program has two parts:

(A) a set of reserved variables and functions which directly indicate the current state of the indicated input device (see Table IIIA), and

(B) a simple sublanguage for communicating with the part of the time-sharing system which handles input interrupts.

The interrupt sublanguage allows LEAP programs to "activate" the various input devices at a TX-2 console,[2, 11] thereby asking the time-sharing executive to gather relevant information at the exact time that an input event occurs, and report this information to the user when he is next active. The user may ask for certain status information to be recorded along with the specified input event. For example, he may ask that the real-time clock reading be recorded whenever a knob changes state:

(1) ACTIVATE βKNOBS REPORTING βRTC*;

The time-sharing executive reports input information to the user by maintaining a list of events, each with appropriate cause and status information. The user may ask for information about the next event; an entry will be removed from the list of events, and the cause and status information will be reported to him. If the list is empty, he will be notified. The user calls a reserved procedure to get information about the next event:

GETNEXTINT;

This procedure stores the appropriate cause code (an INTEGER) into the reserved variable αCAUSE, and device status information into appropriate reserved variables (e.g. if the event were a knob change, the state of the four knobs would be copied into the reserved variables αKNOB 1, αKNOB 2, αKNOB 3, and αKNOB 4). If a request to report the real time clock reading accompanied the knob activation statement (as in (1)), the reading taken at the time of the event would be stored into the reserved variable αRTC. If the list of events

---

*reserved words in the language are in CAPITALS.

is empty, the GETNEXTINT procedure would store zero into αCAUSE and then return.

The input sublanguage consists of three special statement forms, and a number of reserved variables, procedures, and functions.

(1) Statements in the input sublanguage:

(a) ACTIVATE <input device name> ;

(b) ACTIVATE <input device name>REPORTING< report list>;

(c) DEACTIVATE <input device name> ;

The "input device names" are listed in Table III B. Note that there are four interval timer device names, each of which may be activated with a unique interval time, in milliseconds. The minimum interval time is 64 milliseconds; the maximum is 2^{18} milliseconds.

The "report list" consists of one or more "report specifications," separated by commas (see Table III C).

(2) Reserved variables in the input sublanguage are presented in Table III D. Reserved functions are presented in Table III E, and reserved procedures are shown in Table III F.

NAME	DATA TYPE	NOTES
RTC	INTEGER	real time clock register
KNOBS	INTEGER	knob register
TOGS	INTEGER	reg. 377621_8
EIR	INTEGER	reg. 377621_8
COR	INTEGER	reg. 377622_8
KNOB1	FIXED	quarter 1 of knob reg.
KNOB2	FIXED	quarter 2 of knob reg.
KNOB3	FIXED	quarter 3 of knob reg.
KNOB4	FIXED	quarter 4 of knob reg.
TBLTX	FIXED	x-coordinate of tablet stylus[12]
TBLTY	FIXED	y-coordinate of tablet stylus
TBLTSW1	BOOLEAN	switches which become TRUE as the tablet stylus moves toward the tablet surface.
TBLTSW2	BOOLEAN	
TBLTSW3	BOOLEAN	
TOG1	BOOLEAN	BITS 1.1 thru 1.9 of reg. 377621_8. if the bit is a 1, value is TRUE 0 => FALSE.
TOG2	BOOLEAN	
:	:	
TOG9	BOOLEAN	
META	BOOLEAN	META bit on knob register

TABLE III A: RESERVED VARIABLES AND FUNCTIONS FOR INTERACTIVE INPUT.

DEVICE NAMES	VALUE OF αCAUSE (in octal)	AUTOMATIC REPORT (in addition to αCAUSE)
βTARGET	1	αITEMSEEN, αGRPSEEN
βTRACKSTART	2	αTBLTX, αTBLTY
βTRACKEND	3	αTBLTX, αTBLTY
βSWCHANGE	4	αTBLTX, αTBLTY
βINTM1 {<# millisecs>}	5	- - - - - - - - -
βINTM2 {<# millisecs>}	6	- - - - - - - - -
βINTM3 {<# millisecs>}	7	- - - - - - - - -
βINTM4 {<# millisecs>}	10	- - - - - - - - -
βINTMS	11	- - - - - - - - -
βKNOBS	12	αKNOB1, αKNOB2, αKNOB3, αKNOB4
βEIR	13	αEIR
βKEYBOARD	14	- - - - - - - - -
βINKING	17	αNUMSTROKES
βTRACKING		- - - - - - - - -

TABLE III B: INPUT DEVICE NAMES AND THEIR OCTAL CODES, AND RESERVED VARIABLES AUTOMATICALLY REPORTED.

REPORT SPECIFICATIONS	NOTES
βKNOBS	causes αKNOB1 thru αKNOB4 to be set up.
βEIR	causes αEIR to be set up.
βRTC	causes αRTC to be set up.
β TBLTPOS	causes αTBLTX and αTBLTY to be set up.

TABLE III C: REPORT SPECIFICATIONS

RESERVED VARIABLES	DATA TYPE
αKNOB1	FIXED
αKNOB2	"
αKNOB3	"
αKNOB4	"
αTBLTX	"
αTBLTY	"
αEIR	INTEGER
αRTC	"
αITEMSEEN	"
αCAUSE	"
αGRPSEEN	"
αNUMSTROKES	"

TABLE III D: RESERVED VARIABLES FOR THE INTERACTIVE INPUT SUBLANGUAGE.

FUNCTION AND PARAMETERS	DATA TYPE	NOTES
NUMPOINTS {<stroke number>}	INTEGER	value is the number of ink points in the indicated stroke
INKX {<stroke number>,<point number>}	FIXED	X and Y coordinates of the indicated ink point
INKY {<stroke number>,<point number>}		

TABLE III E: RESERVED FUNCTIONS WHICH ARE RELATED TO THE INKING EVENT.

PROCEDURE NAME	NOTES
CLEARINK (no parameters)	This causes the ink to be removed from the display, and the inking machinery to be re-armed.
TURNOFFINTS (no parameters)	Turn off all interrupt devices.
CLEARINTS (no parameters)	Clear out the list of input events.
SETPENMODE {<display group ID>, <pen mode: 0,1,2, or 3>}	See Note 5 below.
GETNEXTINT (no parameters)	The GETNEXTINT reserved procedure reports the next occurrence of an input event by setting up αCAUSE with the appropriate code and setting up the appropriate reserved variables. If there is no event recorded, αCAUSE will be set to zero.

TABLE III F: RESERVED PROCEDURES FOR THE INTERACTIVE INPUT SUBLANGUAGE.

SAMPLE PROGRAM

This program displays a smooth line for every line drawn in with the tablet stylus.

```
START
INTEGER        ITEMNUM;
               ACTIVATE βINKING;
               0 →ITEMNUM;
               CLEARINK;
TAG1⌐          GETNEXTINT;
               IF αCAUSE = 0 THEN BEGIN SHADE; GOTO TAG1 END;
               IF αCAUSE ≠ 17. THEN HELP;
               IF αNUMSTROKES ≠ 1 THEN GOTO TAG2;
               SETITEM {ITEMNUM + 1 →ITEMNUM };
               LOADPNT { ↑INKX {1,1 }, ↑INKY {1,1 } };
               PUTLINE { ↑INKX {1, NUMPOINTS {1 }}- INKX {1, 1 },
                              ↑ INKY{1, NUMPOINTS {1}}- INKY {1,1}};
               PUTITEM {0 };
TAG2⌐          CLEARINK;
               GOTO TAG1
FINISH
```

MISCELLANEOUS NOTES

1) One can optionally specify an "inking wait duration" (i.e. time delay between lifting the pen from the tablet surface and receiving the inking interrupt) by specifying an integer value between 0 and 100 when activating inking:

ACTIVATE βINKING {<INTEGER >}

e.g. ACTIVATE βINKING {40} REPORTING βEIR;

The increment is 5 ms; the default (normal) delay is 500 ms (1/2 second).

2) -1 < coordinate value <1

3) "ACTIVATE βTRACKING" simply renders the tracking dot visible; no input event is associated with this input device.

4) CLEARINK must be executed (after activating inking) before inking will occur.

5) The "pen mode" attribute of a display group specifies the relationship between the picture indicated by the group and information to be reported to the user when a target is "seen" by the pen. In the case where a target has subpictures (uses) as parts, the user must specify which item in which group is to be reported when a target is seen. He does this by specifying a "pen mode" for each display group; this indicates which group is the "working level": item ID's from this group are reported when a target is seen. There are four pen modes:

   0) Normal (default) mode: look above here for the working level.

   1) Picture parts here and below are invisible to the pen.

   2) (Unused).

   3) Working level: this group contains a group of targets; report the ID of this group and the ID of the item seen by the pen.

6) a) 'α' prefixes denote reserved variables which are stuffed by GETNEXTINT (e.g. αKNOB1).

   b) "β" prefixes denote device names (e.g. βKNOBS).

   c) No prefix (see TABLE III A) denotes a reserved variable or function whose value is a direct reading of the indicated device status when the reference is made (e.g. KNOB1).

APPENDIX IV

TEXT AND NUMERICAL I/O

CONTENTS:

IV. A. TEXT AND NUMERICAL INPUT

The facility for typewritten input to a LEAP program is line-oriented and format-free. Normally (see SETSMACKER procedure), a line which is being typed in is not processed until a read-in key or carriage return key is pushed. Five special function keys are allowed:

a) The DELETE key: deletes the previous character typed, unless there is no previous character on this line.

b) the WORD EXAM key: delete the previous input word on this line, and any trailing spaces or tabs.

c) the NO key: delete all previous characters on this line.

d) the YES key: types a clean version of the input line so far.

e) the READ-IN key: terminates the line, using it as a text file name, and pushes the contents of that text file onto the source of input characters.

An input line consists of a sequence of input words, separated by spaces and/or tabs. The READ statement takes a list of variables as its argument, and attempts to read one input word into each variable, working from left to right, until the argument list is exhausted. If there are not enough input words to satisfy the argument list, the system will wait for sufficient input from completed input lines to be typed. As each input word is read into a variable, allowed data-type conversions are made (see Table A3). The data-type of the input word is determined from its format (see the discussion of constants in Section I. A), and the data-type of the variable is known from its declaration. Only variables of the following types may be arguments to a READ statement:

REAL
INTEGER
BOOLEAN
FIXED
TEXTARRAY

If a TEXTARRAY variable is the argument, an input word will be copied character by character into the indicated textarray, starting with the first element in the textarray. The value of each textarray element will be the integer character code for the indicated character. The next available

element in the textarray will have the value 777_8 to indicate end-of-word. The reserved INTEGER variable αCHARCNT will contain the number of characters read into the TEXTARRAY (not including the 777_8 character).

There are two general forms for the READ statement:

a) READ <list of variables separated by commas>;
(example: READ X, Y, I, IBA;)

b) READ {<ID: an integer expression>} <list of variables>;
(example: READ {37} X, Y, I, IBA;)

The second of the above forms is used to indicate an identifying integer for the READ statement; in case of a read error, this integer is reported to the user along with the appropriate read error code (see Table A4.).

The READ statement reads input words; there is another statement for reading input characters:

a) READCHAR <list of INTEGER variables>;

b) READCHAR {ID} <list of INTEGER variables>;

This statement takes a list of INTEGER variables as its argument, and attempts to read one input character into each variable, going from left to right, until the argument list is exhausted. The indicated integer character code is stored into each variable. Spaces, tabs, and carriage returns ARE treated as input characters. If there are not enough input characters to satisfy the argument list, the system will wait for sufficient input from completed input lines to be typed.

The second READCHAR statement form is similar to the second READ statement form; in case of a read error, the indicated ID is reported to the user along with the appropriate read error code (see Table A4.).

The user may disable the built-in facilities for reporting a read error by executing a statement of the following form:

SETRDERLBL <label>;

This causes the system to note the indicated label, and transfer control to it instead of printing an error message when the next read error occurs. Appropriate information is stored into reserved variables when a read

error occurs (see Table A1.).

The user may cause his program to take its input from a text file rather than from the keyboard. At execution time, he may type the name of a text file, and then hit the READ-IN key. This causes the indicated text to be read in exactly as if it were typed in. When the text file is exhausted, a message will be printed out, and input will again be taken from the keyboard. Note that no change need be made to the user program.

The user may re-read an input word or input character on the current input line by storing away and later resetting the system's input pointer.* This pointer is kept in the reserved variable INPTR (see Table A1.).

TABLE A1. RESERVED VARIABLES AND PROCEDURES FOR THE LEAP INPUT FACILITY

(1) READNUM (INTEGER)

The value of this variable is set to the ID of the offending statement (if specified) when a read error occurs.

(2) RDERRCODE (INTEGER)

The value of this variable is set to the read error code number (see Table A4.) when a read error occurs.

(3) RDTATYPE (INTEGER)

The value of this variable is set to the data type code of the input word if an illegal mode conversion is requested.

(4) ENDOFLINE (BOOLEAN)

This variable is set to FALSE at the beginning of each READ and READCHAR statement execution, and set to TRUE at the end of the execution if there is no more input on the current line.

(5) INPTR (INTEGER)

The value of this variable is a pointer to the next character on the current input line.

(6) LASTINPTR (INTEGER)

This is an integer reserved variable which is used to store the previous value of INPTR. Each time an input character or input word is to be read from the current input line, the value

---

* Note that INPTR may not be reset to point into a previous input line.

of INPTR is assigned to LASTINPTR. If a new input line must be fetched, LASTINPTR is reset to the beginning of the new line. The system uses the value of INPTR as its pointer into the current input line; the user may save LASTINPTR or INPTR, and reset INPTR if desired. Note that INPTR may not be reset to point into a previous input line.

(7) ISCHARINPUT (BOOLEAN PROCEDURE; no parameters)

This returns the value TRUE if there are any characters left on the current input line, or if there is another completed input line available; the value FALSE is returned otherwise.

(8) ISWORDINPUT (BOOLEAN PROCEDURE; no parameters)

This returns the value TRUE if there are any input words left on the current input line, or if there are input words on any new, completed input line; the value FALSE is returned otherwise.

(9) CLEARKBDLINE (PROCEDURE; no parameters)

This removes all input from the current input line.

(10) CLEARKBD (PROCEDURE; no parameters)

This removes all completed input lines from the source of typed input.

(11) READINTEXTFILE (PROCEDURE; Textarray parameter)

This procedure pushes the textfile whose file name is given onto the stack of input character sources. If the parameter is not a correct textfile name, a READ ERROR #12 will result.

Example: READINTEXTFILE { 'STANDARDTEXT' };

(12) SETSMACKER (PROCEDURE; boolean parameter)

This procedure allows the user to access single characters typed on the keyboard before a carriage return is typed. Only the functions READCHAR and ISCHARINPUT are changed. After a call of the form SETSMACKER {FALSE }, READCHAR will return any character typed,

including the five function keys which, obviously, have no effect when accessed in such a manner. This is a special mode of operation, primarily for those who wish to use the keyboard as a set of control keys, rather than as a source of input text lines or words. Under this mode, READ acts as it always does, but INPTR, LINPTR, and ISWORDINPUT should not be used.

The normal mode for the read package is restored by executing a

```
SETSMACKER{TRUE};
```

statement.

DATA TYPE	CODE
BOOLEAN	1
INTEGER	2
FIXED (FRACTION)	3
REAL	4
ALPHANUMERIC	5

TABLE A2. DATA TYPE CODES

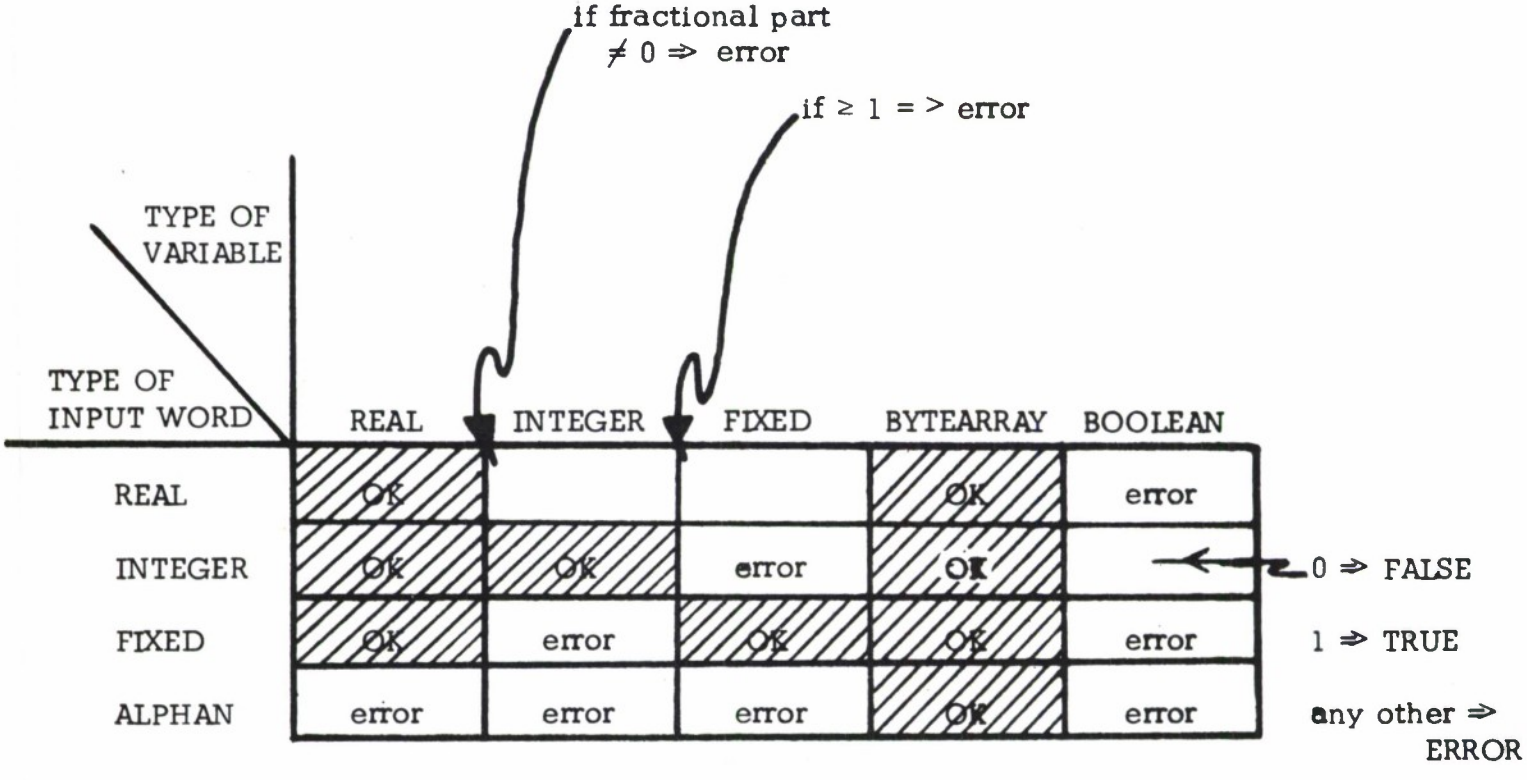

TABLE A3. ALLOWED DATA TYPE CONVERSIONS FOR THE READ STATEMENT

TABLE A4. READ ERROR CODES

CODE (in OCTAL)	ERROR
1	illegal mode conversion - example: you tried to read an INTEGER into a FIXED (fraction) variable.
2	too many characters on this line
3	used ISWORDINPUT while SMACKER was off
4	you tried to do a READCHAR into a variable of different type then INTEGER
12	tried to read-in a nonexistent text file

B. TEXT AND NUMERICAL OUTPUT

1. Statements which output a text array:

a) PRINT <text array>;

b) XEROX <text array>;

This statement causes the indicated text to be appended to the XEROX buffer. This buffer is maintained by the APEX executive. The following statement causes the XEROX buffer to be printed and then cleared:

DUMP XEROX;

c) SHOWTEXT {<text array>, <display item ID>, <display group ID>, <X position>, <Y position> };

This is a reserved procedure which causes the indicated text to be added as a display item to the current display structure. The indicated position coordinates specify the position of the lower left corner of the first character.

d) STORETEXT {<text array> , <X position> , <Y position>};

This is a reserved procedure which causes the indicated text to be displayed on the storage tube at the indicated position.

2. FORMAT Statements

The FORMAT statement is used to define a format descriptor, and associate it with a format description. A format description is used to specify the manner in which printed output is to be formatted. For example, a format description may indicate the number of digits to be printed after the decimal point of a real number, or the number of spaces between fields on an output line, or whether to print or suppress leading zeros.

The FORMAT statement has the following general form:

FORMAT <name of format descriptor> (<format description>);

A FORMAT statement should appear as a declaration in a declaration portion of a LEAP program.

In general, a format description consists of several data descriptors which are separated by vertical bar or slash. In addition to separating data descriptors, a slash causes a carriage return to be inserted on the

output line when the format description is applied to data to be output.

Data descriptors in a format specification are matched to data arguments on a one-to-one basis. A full discussion of the format scan and list matching follows this section.

In general, a data descriptor consists of a combination of designators to specify the different portions of the data argument which is to be printed. Nesting of data descriptors is accomplished by parentheses preceded by an optional replicator (see the examples). The general form for a number specification is:

[SIGN] [WHOLE DESIGNATOR] [POINT] [FRACTIONAL DESIGNATOR] [CONVERSION] [MODIFIER].

Some of these fields are optional (see the examples).

Numbers are converted to characters according to the conversion designator. These are:

$\overline{K}$ for octal integer.
$\overline{I}$ for decimal integer,
$\overline{F}$ for fraction,
$\overline{E}$ for mixed plus exponent of 10,
$\overline{R}$ for mixed number, and
$\overline{A}$ for alphanumeric

The modifier is an integer constant specifying the power of ten (or eight for octal integers) which multiplies the number before it is placed for output. For example, $\overline{I}$ 2, would cause the integer to be multiplied by 100 (10^2) before processing.

The sign of a number is specified by an optional portion of the specification. The sign may have either a fixed or floating position. A fixed sign is declared by having only a single + or - sign. A floating sign is declared by preceding the sign with a replicator larger than 1. This defines the sign field. The + causes the sign to be printed regardless of its value; the - causes only negative signs to be printed.

A fixed sign is printed in the specified position at the left of the field. A floating sign is printed either at the left of the first significant

digit or at the right of the sign field.

A decimal point is indicated by a comma.

Both the whole and fractional parts of a number are used to describe the digit positions before and after the decimal point. The two digit designators are:

$\overline{D}$ Print digits, but suppress leading or trailing zeros

$\overline{Z}$ Print digits with leading or trailing zeros.

These designators must be ordered if both are used to describe either whole or fractional parts. For the whole part of a number, ($\overline{D}$) must precede ($\overline{Z}$), and for the fractional part, ($\overline{Z}$) must precede ($\overline{D}$).

There are two special output descriptors which may be used in a format description:

(a) $\overline{S}$ (insert a space character)

(b) $\overline{T}$ (insert a tab character)

Examples of the FORMAT statement follow:

(a) FORMAT F1 (6 $\overline{D}$ $\overline{I}$);

Specifies a six digit decimal integer with leading zeros supressed. If a sign is not specified, + is assumed.

(b) FORMAT F2 (- 7 $\overline{D}$, 3 $\overline{Z}$ $\overline{R}$);

Specifies a real number having seven integer digits, and three fractional digits, with trailing zeros. A sign will be printed only if the number is negative.

(c) FORMAT F3 (7 $\overline{A}$);

Specifies seven integer numbers, which will be treated as character codes, and printed as the indicated characters.

(d) FORMAT F4 (3 (4 $\overline{D}$ $\overline{I}$ | -5 $\overline{D}$, 6 $\overline{D}$ $\overline{E}$) | 2 $\overline{A}$);

Specifies three pairs of numbers (the first of each pair an integer, the second a real) followed by two character codes.

3. Statements for Formatted Output

There are three statements which generate formatted output:

(a) PRINT FORMAT <format descriptor> , <output list>;

This causes the indicated output to be printed on the Lincoln

writer (see the discussion of the output list below).

(b) XEROX FORMAT <format descriptor>, <output list>;

This causes the indicated output characters to be put into the APEX Xerox buffer. The user program must force this buffer to be dumped by executing a

DUMP XEROX;

statement.

(c) GATHER FORMAT <format descriptor>, <output list>;

This causes the indicated output characters to be appended to a special reserved textarray named OUTPUT. This textarray may be used as a parameter to the statements described in section B of this appendix, for example. The following special statement clears and reinitializes the OUTPUT reserved textarray:

CLEAROUTPUT;

There are several restrictions on the use of this textarray:

(i) Storage for the elements of OUTPUT is automatically allocated, and is of a fixed length (500 characters). Do not attempt to re-declare or assign storage to OUTPUT.

(ii) References may be made to the elements of OUTPUT, but do not attempt to move the 777. character if subsequent GATHER statements are to be executed before a CLEAROUTPUT is done.

4. The Output List

The output argument list in a formatted output statement consists of arithmetic expressions and braced FOR statements. The comma is used to separate list elements.

The braced ({ }) FOR statement is an iterative output argument. This means that several elements of the argument list may be indicated by one FOR statement. The braced FOR statement has the same form as the regular FOR except that the DO clause is an arithmetic expression or another braced FOR statement.

Examples: {FOR 1 → I STEP 1 THRU 10 DO A_I}

would be equivalent to listing arguments A_1 . . . A_{10}.

{FOR 1 → I STEP 1 UNTIL I> 10 DO
{FOR 1 → J STEP 1 UNTIL J> 10 DO $A_{I,J}$, $B_{J,I}$}}

would be equivalent to listing elements $A_{1,1}$, $B_{1,1}$, $A_{1,2}$, $B_{2,1}$...

$A_{10,10}$, $B_{10,10}$.

```
{FOR 1 → I STEP 1 UNTIL I > J DO
{FOR 1 → K STEP 1 UNTIL K> 3 DO A I,K }}
```

would cause the variables $A_{1,1}$, $A_{1,2}$, $A_{1,3}$, $A_{2,1}$, etc. to be used.

The processing for formatted output is controlled by the output list. The format description is scanned and processed until a data descriptor is found. The next output argument is then fetched and processed; the format scan is continued until there are no more arguments. If the end of the format description is reached before the output list is exhausted, a carriage return is automatically inserted, and the scan restarts from the beginning.

Examples:

```
FORMAT F (5 D̄ Ī)
PRINT FORMAT F, A, B, C;
```

causes A, B and C to be printed as 5-digit integers on separate lines.

```
FORMAT F (5(5 D̄ Ī);
XEROX FORMAT F, X, Y;
```

cause X and Y to be placed in Xerox buffer as 5-digit integers on one line.

```
FORMAT F (2(5D̄, 3D̄ Ē))
PRINT FORMAT F, (FOR 1 → I STEP 1 UNTIL I > 6 DO
{FOR 1 → J STEP 1 UNTIL J > 2 DO A I,J }};
```

causes array elements A_{11}, A_{12}, A_{21}, A_{22}, . . . A_{52} to be printed as real numbers, two to a line.

APPENDIX V

SUBPROGRAM LINKAGE FACILITY*

A. GOUPTO AND PEELBACK

There is a facility for going up to a LEAP program from a LEAP program with input parameters and output parameters. The calling program executes a statement of the form:

GOUPTO <TEXTARRAY expression> <argument list>;

where the TEXTARRAY contains the name of the LEAP program to be called, and the argument list may be:

a. null, if there are no parameters.

b. {<INPUT parameter list>}, if there are only input parameters.

c. {; <OUTPUT parameter list>}, if there are only output parameters, and

d. {<INPUT parameter list>; <OUTPUT parameter list>}, if there are both.

Input parameters may be variables or expressions; output parameters must be variables.

In the called program, if there are any input parameters, a declaration of the form

INPUT {<declaration list>};

must appear immediately after USELEAP, or after START if there is no USELEAP. The declaration list is similar to the declaration list for a PROCEDURE declaration, with the exceptions that LABEL and PROCEDURE parameters are not allowed, and a program may use the "FILE" declaration to pass the name of a file (or any name) in the public or private directory as a parameter in the GOUPTO statement. A "directory item" parameter is put into the connector, and the INPUT declaration on the upper map causes the text of the file name to be made available. The "FILE" declaration is used on both maps as follows:

Examples:

lower map: GOUPTO 'BLOP' {FILE 'SAM', . . .};

---

* For an introduction to the APEX time-sharing executive and features of the time-shared virtual machine, see references 6 and 11.

upper map

(in the program BLOP):

```
START
  INPUT {FILE X, . . .};
```

After the INPUT declaration on the upper map, X behaves like a declared TEXTARRAY variable, having the FILE NAME as its value.

When the called program finishes, it may execute a PEELBACK statement:

```
PEELBACK {output parameter list};
```

or simply execute the FINISH statement.

B. OVERLAYS

A LEAP program may be segmented into one main program and several subprograms (called OVERLAYs). At execution time, the main program is set up on the user's map, and remains set up until execution terminates. Overlays may be set up and dropped from the map under program control. Only one overlay at a time may be set up. The main program must be no larger than one book of code, and each overlay is similarly restricted. At compile time, the user must use the BBIN[7] command to compile his program if overlays are declared within.

The overlay facility was implemented for three reasons:

(1) to help reduce the maximum core requirement both at compile-time and at run-time,

(2) to provide an alternative to the GOUPTO facility, which may cause large inefficiencies if much information is passed between maps, and

(3) to get around the requirement (imposed by the VITAL[7] system) that the total code compiled for any one program not exceed two books.

Overlay declarations should appear immediately before FINISH in a LEAP program. Overlay declarations may not be nested. A LEAP

program in which overlays are declared should have the following general form:

```
START
      < entire main program >
      DEFINE OVERLAY '<character string >';
      < statement >;
      DEFINE OVERLAY '<character string >';
      < statement >;
                  .
                  .
                  .
FINISH
```

Example:

```
START
      REAL  X, Y;
      3.0 → X;
          .
          .
          .
      DEFINE OVERLAY 'OVL1';
            BEGIN
                  .
                  .
                  .
            END ;
FINISH
```

There are three statements in LEAP which are related to the overlay feature:

(1) CALL < textarray expression >;

This statement causes the overlay with the indicated name to be set up, and control to be transferred to the first statement in the overlay. If a different overlay is already set up when this statement is executed, it will be dropped from the map.

(2) OVERLAYRETURN;

This statement causes control to return from an overlay to the statement following the CALL statement last executed. Note that one overlay may call another overlay; the calling

overlay is re-set up before control is returned.

A GOTO statement which transfers control to a label in the main program may be executed from within an overlay (if the label is not within a block). Note that labels declared within an overlay may not be referenced from outside the overlay.

(3) DROPOVERLAY;

This statement causes the current overlay (if any) to be dropped from the current map.

APPENDIX VI

ERROR DETECTION IN LEAP

A. PRODUCTION ERRORS

These errors appear in the syntax phase of compiling.* They are noted by the following comment:

```
PERRXXXX  EDITARG
--------  (line in error)
```

where, XXXX is the error number, EDITARG is a standard argument defining the line in error, and asterisks mark the current scan pointer at the occurrance of the error.

HINT: If the error occurs on the first word of the line, then the error may be caused by an incorrect end to the preceeding line.

B. SEMANTIC ERRORS

These errors are caused by the VITAL mechanisms and may indicate an error in the compiler. They are noted by the comment:

```
SERR XXXX  EDITARG
---------  (line in error)
```

where XXXX and EDITARG are as previously defined. If these errors occur, the user should consult the staff.

C. SEMANTIC FAULTS

These faults occur in the semantics of the language and are noted by the comment:

```
FAULT XXXX  EDITARG
----------  (line in error)
```

where XXXX and EDITARG are as defined for production errors.

A complete tabulation of both compile-time and run-time errors and probable causes is presented below.

---

*The LEAP compiler was written using the VITAL compiler-compiler, and is housed in the VITAL system. For information about VITAL, see reference 7.

EAULIS.SERRS.AND_PERRS

A. EAULIS

301	COMPILER TABLE OVERFLOW - SEE PDR
34	TOO MANY PROPERTIES DECLARED (5 MAX)
3410	USELEAP MISSING
3411	TOO MANY LOCALS DECLARED (20 MAX)
3412	BAD DECLARATION
1611	CAN ONLY DECLARE IIEMS AND LOCALS AND PROPERIIES AT THE BEGINNING OF A PROGRAM
31	ILLEGAL CONSTANT
36	'REIURN' MUST BE USED WITHIN A PROCEDURE DECLARATION ONLY
37	ILLEGAL USE OF RETURN
41	ONLY SIMPLE DATA TYPES ALLOWED FOR A FUNCTION DATA TYPE
411	CAN'T PASS A PROCEDURE OR LABEL AS A PARAMETER WHEN GOING UP
42	A FILE DECLARATION CAN OCCUR ONLY IN AN INPUI STATEMENT
431	ILLEGAL LABEL PARAMETER DECLARATION
60	PARAMETER IS NOT A MATRIX
76	TEXTARRAYS HAVE ONLY ONE SUBSCRIPT
100	EXPRESSIONS HAVE INCOMPATIBLE TYPES
1001	EXPRESSIONS HAVE INCOMPATIBLE TYPES
1002	ONLY ONE EXPRESSION IS IN THE DYNAMIC ACCUMULATOR - BOTH MUST BE
101	PARAMETER IS NOT A MATRIX
1011	EXPRESSIONS HAVE INCOMPATIBLE TYPES
1012	EXPRESSIONS HAVE INCOMPATIBLE TYPES
1013	EXPRESSIONS HAVE INCOMPATIBLE TYPES
1014	ILLEGAL REFERENCE VARIABLE
103	ILLEGAL PROCEDURE ARGUMENT
114	ILLEGAL ITERATION VARIABLE
124	ILLEGAL ITERATION VARIABLE
126	CONIINUE CANNOT BE USED OUTSIDE OF AN ITERATION STATEMENT
1311	THIS PROCEDURE IS CALLED WITH THE WRONG NUMBER OF ARGUMENTS
1312	ILLEGAL VARIABLE
1313	YOU CANNOT USE A LOCAL OUTSIDE OF THE FOREACH STATEMENT WHICH BINDS IT
134	SAVE-RESIORE CANNOT BE USED OUTSIDE OF A PROCEDURE DEFINITION
136	ILLEGAL PROPERTY
1361	ILLEGAL IIEM EXPRESSION
137	BAD ARGUMENT FOR SAVE
1371	BAD ARGUMENT FOR SAVE
152	ILLEGAL Ms INDEX (MUST BE A CONSTANT OR A Ms EQUALITY)

```
154    YOU CAN'T CHAIN A Ms EQUALITY
170    YOU CAN'T CHAIN Ms EXPRESSIONS
174    YOU CANT CHAIN Ms EXPRESSIONS
202    THIS PROCEDURE IS CALLED WITH THE WRONG NUMBER OF ARGUMENTS
205    THIS PROCEDURE IS CALLED WITH THE WRONG NUMBER OF ARGUMENTS
206    ARGUMENT TO PRINI STATEMENT IS NOT A IEXIABRAY
207    ILLEGAL PROPERIY
212    CAN'T USE A FUNCTION AS A STATEMENT
215    CAN'T USE A PROCEDURE AS AN EXPRESSION
234    CAN'T TAKE THE FRACTIONAL PART OF AN INIEGER
241    REPLICATOR IS NEITHER A CONSTANT NOR A VARIABLE
243    REPLICATOR IS NOT A CONSTANT
250    ILLEGAL USE OF A REPLICATOR
251    ILLEGAL USE OF A REPLICATOR
252    THE EXPONENT IS NOT A CONSTANT
253    THE EXPONENT IS NOT A CONSTANT
271    ILLEGAL FORMAI DESCRIPTOR
274    ILLEGAL FORMAI DESCRIPTOR
306    ILLEGAL VARIABLE
307    ILLEGAL PARAMETER (MUST BE INTEGER)
326    ILLEGAL REFERENCE VARIABLE
335    EITHER THIS IRIPLE HAS NO FREE LOCALS OR IT HAS ALL LOCALS
3351   EITHER THIS TRIPLE HAS NO FREE LOCALS OR IT HAS ALL LOCALS
337    CAN'T HAVE A SEI BE PART OF A IRIPLE WHICH IS USED AS AN IIEM EXPRESSION
340    BOTH PARAMETERS ARE LOCALS (CAN'T HANDLE THIS YET)
3401   TOO MANY LOCALS DECLARED (20 MAX.)
347    ILLEGAL SEI PARAMETER
350    ILLEGAL IIEMVAR
355    THE ITERATION VARIABLE IS NOT A LOCAL
3571   CAN'T RECLAIM THIS
363    THE ITERATION VARIABLE IS NOT A LOCAL
3631   THIS ITERATION VARIABLE IS ALREADY BOUND
374    SYNONYM TO AN UNDEFINED
430    BAD ARGUMENT TO READ
4301   BAD ARGUMENT TO READ
431    BAD ARGUMENT TO READCHAR (MUST BE AN INTEGER VARIABLE)
440    THE MAIN PROGRAM IS TOO BIG TO USE OVERLAYS
4401   BAD OVERLAY NAME
442    CAN'T CALL AN OVERLAY FROM WITHIN A PROCEDURE
```

B. SEMANTIC ERRORS (SERR'S)

23	ERROR IN STORE ROUTINE...USUALLY CAUSED BY INCOMPATIBLE DATA TYPES
100XXX	YOU PROBABLY FORGOT A QUOTE

C.PRODUCTION ERRORS (PERR'S)

1	ILLEGAL PROGRAM BEGINNING
2	ILLEGAL DECLARATION (SIMPLE)
3	ILLEGAL DECLARATION ENDING
4	ILLEGAL MATRIX DECLARATION
5	ILLEGAL DECLARATION
6	ILLEGAL DECLARATION (TEXT) ARRAY
7	ILLEGAL DECLARATION
10	ILLEGAL DECLARATION EXTERNAL
11	ILLEGAL DECLARATION PROCEDURE
12	ILLEGAL DECLARATION (PROCEDURE)
13	ILLEGAL DECLARATION (''PARAMETERS)
14	ILLEGAL DECLARATION
15	ILLEGAL DECLARATION
16	ILLEGAL DECLARATION
17	ILLEGAL DECLARATION
20	ILLEGAL STATEMENT START
21	ILLEGAL STATEMENT START
22	ILLEGAL EXPRESSION START
23	ILLEGAL USE OF AN ARRAY
24	ILLEGAL USE OF IS
25	ILLEGAL USE OF Y
26	ILLEGAL USE OF AN IIEM EXPRESSION
27	ILLEGAL USE OF A MATRIX
31	ILLEGAL USE OF AN EXPRESSION
32	THIS IS NOT A VARIABLE
33	THIS IS NOT AN IIEM EXPRESSION
34	ILLEGAL USE OF AN IIEM EXPRESSION
35	ILLEGAL USE OF A VARIABLE
36	ILLEGAL FORMAT FOR SWITCH
37	ILLEGAL FORMAT FOR CONDITIONAL GOIO
40	ILLEGAL USE OF A PROCEDURE CALL
41	ILLEGAL FORMAT FOR A PROCEDURE CALL
42	ILLEGAL FORMAT FORMAT

43 ILLEGAL FORMAT DECLN END
44 ILLEGAL I/O STATEMENT
45 SYSERR CALL FOR
46 ILLEGAL I/O STATEMENT
47 ILLEGAL I/O STATEMENT
50 ILLEGAL I/O STATEMENT
51 ILLEGAL I/O STATEMENT
52 ILLEGAL ARRAY OR IEXIARRAY EXPRESSION BEGINNING
53 ILLEGAL USE OF A STATEMENT - YOU PROBABLY FORGOT AN END
54 ILLEGAL CONSTRUCT IN MS
55 ILLEGAL CONSTRUCT IN MS
56 ILLEGAL CONSTRUCT IN MS
57 ILLEGAL CONSTRUCT IN MS
60 ILLEGAL CONSTRUCT IN MS
61 ILLEGAL CONSTRUCT IN MS
62 ILLEGAL CONSTRUCT IN MS
63 ILLEGAL CONSTRUCT IN MS
64 ILLEGAL CONSTRUCT IN MS
65 BAD MS STATEMENT END
66 ILLEGAL FORMAT DECLARATION FORMAT
67 ILLEGAL FORMAT DECLARATION FORMAT
70 ILLEGAL FORMAT DECLARATION FORMAT
71 ILLEGAL FORMAT DECLARATION FORMAT
72 ILLEGAL FORMAT DECLARATION FORMAT
73 ILLEGAL FORMAT DECLARATION FORMAT
74 ILLEGAL FORMAT DECLARATION FORMAT
76 ILLEGAL FORMAT DECLARATION FORMAT
77 REELBACK FORMAT ERROR
100 ILLEGAL END TO FILE PARAMETER IN GOUP
101 SYSERR_CALL FOR
102 SYSERR CALL FOR
103 SYSERR CALL FOR
104 SYSERR CALL FOR
105 ILLEGAL USE OF AN ARRAY, IEXIARRAY, OR SEI ITEM

NOIE

USUALLY, PRODUCTION ERRORS ARISE FROM EITHER

(1) FORGETTING A SEMI-COLON

(2) NOT PROPERLY MATCHING BEGIN-ENDS
OR
(3) HAVING A DECLARATION FOLLOW A STATEMENT WITHOUT STARTING A NEW BLOCK WITH BEGIN

RUN-TIME ERRORS

LEAP ERRORS

PLEASE REPORT UNDEFINED INSTRUCTIONS, BDE'S, ETC. TO PDB
NORMALLY, LEAP ERRORS ARE TYPED OUT WITH AN ERROR NUMBER. THE FOLLOWING IS A GUIDE TO THESE NUMBERS.

1 A ZERO INTERNAL IDENTIFIER WAS PASSED TO THE ASSOCIATIVE ROUTINES
2 STRUCTURE IS TOO LARGE
3 UPDATE STORAGE OVERFLOW
4 CORRESPONDENCE STRUCTURE OVERFLOW
5 ILLEGAL USE OF ANY
6 TRIED TO RECLAIM A DECLARED ITEM
7 TRIED TO RECLAIM A FREE ITEM
10 TOO MANY ITEMS
11 ILLEGAL USE OF A SET
12 SET IS EMPTY IN ⊖ STATEMENT
13 BAD STRUCTURE NAME GIVEN TO READSTRUCTURE
14 YOUR TRIED TO JUMP OUT OF A FOREACH STATEMENT
15 ERROR ON GOUPIQ
16 YOU TRIED TO RETURN FROM A FUNCTION BY FALLING THRU TO THE END
17 NO SUCH TRIPLE EXISTS
41 YOU HAVE REQUESTED TOO MANY BOOKS OF STORAGE
42 YOU HAVE REQUESTED TOO LARGE A BLOCK OF STORAGE
43 SUBSCRIPT OUT OF BOUNDS
45 UNDEFINED OVERLAY
46 ATTEMPT TO JUMP OUT OF AN ASSOCIATIVE FOREACH STATEMENT
47 THIS DYNAMIC VARIABLE HAS NO VALUE
50 THE DECLARED LOWER BOUND ON A SINGLE DIMENSIONAL ARRAY IS NOT 1
51 ERROR IN MULMATS
52 ERROR IN SCALAR MULMATS
53 ERROR IN ADD OR SUBTRACT MATRICES
54 SWITCH VARIABLE OUT OF BOUNDS
55 DETERMINANT ERROR
56 INVERSE OR TRANSPOSE ERROR

```
 57   SUBMATRIX ERROR
 60   ADJOIN MATRICES ERROR
 61   MATRIX IS SINGULAR ON INVERSION ATTEMPT
 62   TRIED TO SETUP 2 WRITETHRU BUFFERS
 63   NO WRITETHRU BUFFER IS SET UP
 64   BAD PARAMETER TO EMPTYBOOK
 65   YOU TRIED TO EDIT AN EMPTY TEXTARRAY
 66   ITEM HAS INCONSISTENT DATA TYPE  (GAMMA OPERATOR)
 67   FILE NAME POINTER BAD OR TEXT FILE NOT IN PROPER FORMAT
 71   TRYING TO OPEN A TEXT FILE WITH ONE ALREADY OPEN
 72   TRYING TO PUT OR CLOSE INTO TEXT FILE WITH NONE OPEN.
 77   MASK PGM SYSTEM ERROR - DEFINITIONS
101   SETUPTEMPFILE -- FILE ALREADY SET UP IN SPECIFIED BOOK
104   SETUPTEMPFILE -- LENGTH GREATER THAN 40
105   SETUPTEMPFILE -- IMPOSSIBLE BOOK NUMBER
106   SETUPTEMPFILE -- LENGTH ZERO OR NEGATIVE
201   SETUPFILE -- NAME IS UNDEFINED
202   SETUPFILE -- NAME IS NOT NAME OF A FILE
203   SETUPFILE -- NAME IS NOT LEGAL DIRECTORY NAME
205   SETUPFILE -- IMPOSSIBLE BOOKNUMBER
210   SETUPFILE -- EXECUTABLE SYSTEM FILE MAY ONLY BE SET UP IN BOOK SPECIFIED IN DIRECTORY.

211   SETUPFILE -- THE BOOK IS NOT EMPTY
213   SETUPFILE -- EXECUTABLE FILE MAY ONLY BE SET UP IN BOOK SPECIFIED IN DIRECTORY.
301   SETUPANDNAMEFILE -- FILE ALREADY SET UP IN SPECIFIED BOOK
302   SETUPANDNAMEFILE -- NAME IS NOT A LEGAL DIRECTORY NAME
304   SETUPANDNAMEFILE -- LENGTH GREATER THAN 40
305   SETUPANDNAMEFILE -- IMPOSSIBLE BOOK NUMBER
306   SETUPANDNAMEFILE -- LENGTH ZERO OR NEGATIVE
316   SETUPANDNAMEFILE -- FILENAME IS PROTECTED
401   WHATSIN -- BAD MAP NUMBER
406   WHATSIN -- BELOW SPECIFIED MAP
501   REPORTSTATUSOF -- NAME IS NOT NAME OF A FILE
701   READTF -- FILE NAMED IS NOT A TEXT FILE
702   READTF -- FILENAME IS NOT LEGAL DIRECTORY NAME
703   READTF -- FILENAME IS NOT NAME OF A FILE
705   READTF -- FILENAME IS UNDEFINED
1002  ASSIGNRECOG -- FILENAME IS NOT LEGAL DIRECTORY NAME
1101  CLOSETF -- THERE IS NO FILE OPEN
1102  CLOSETF -- FILENAME IS NOT LEGAL DIRECTORY NAME
```

1103 CLOSETF -- FILENAME IS NOT NAME OF A FILE
1201 PUTCHARINTF -- THERE IS NO FILE OPEN
1301 PUTTAINTF -- THERE IS NO FILE OPEN
1401 SETSTATUSOF -- FILE IS PROTECTED AGAINST ANY CHANGES
1402 SETSTATUSOF -- NAME IS NOT NAME OF A FILE
1403 SETSTATUSOF -- NAME IS NOT LEGAL DIRECTORY NAME
1404 SETSTATUSOF -- CANNOT ALTER LENGTH OF READ - ONLY FILE
1405 SETSTATUSOF -- NAME IS UNDEFINED
1406 SETSTATUSOF -- CREATING FILE OF ZERO OR NEGATIVE LENGTH
1407 SETSTATUSOF -- CREATING FILE OF MORE THAN 40 PAGES
1410 SETSTATUSOF -- ONLY PRIVILEGED USER CAN ALTER PUBLIC FILE
OTHER NUMBERS- CALL P·D·R·

OUTPUT ERRORS

1 BUFFER OVERFLOW ON GATHER STATEMENT
2 ILLEGAL DATA TYPE CONVERSION ATTEMPTED
3 NUMERICAL OVERFLOW ON OUTPUT ATTEMPT

READ ERRORS

1 ILLEGAL DATA TYPE CONVERSION ATTEMPTED
2 TOO MANY CHARACTERS ON THIS LINE
4 YOU TRIED TO DO A READCHAR INTO A VARIABLE OF DIFFERENT TYPE THAN INTEGER
10 ILLEGAL CHARACTER IN THIS TEXT FILE
11 READIN KEY IN THIS TEXTFILE
12 TRIED TO READ-IN A NON-EXISTENT TEXT FILE

SCOPE ERRORS

1 TOO MUCH STUFF FOR ONE ITEM (600_8 TSD'S MAX)
2 TOO MUCH STUFF FOR ONE ITEM (600_8 TSD'S MAX)
3 ERROR IN MULMAIS
4 ERROR IN GENISD FOR MAIRIX

A. OTHER RESERVED FUNCTIONS

DATA TYPE	FUNCTION NAME	ARGUMENTS	NOTES
FIXED	SIN	FIXED; part of π; i.e., .5 = 90°	SINE
REAL	SINR	REAL number in radians	SINE
FIXED	COS	FIXED; same as SIN	COSINE
REAL	COSR	REAL number in radians	COSINE
FIXED	ATAN	{ΔX, ΔY} both FIXED	ARC TANGENT
REAL	ATANR	REAL number	ARC TANGENT
FIXED	SQRT	FIXED	SQUARE ROOT
REAL	SQRTR	REAL	SQUARE ROOT
FIXED	PYTHAG	{X, Y} both FIXED	computes SQRT $\{X^2 + Y^2\}$
REAL	LOG	REAL	LOG_{10}
REAL	LOGE	REAL	LOG_e
REAL	EXP	REAL	e^x

B. LEAP BIN MODES

The "bin" command to VITAL may be followed by a vertical bar, and an octal number.

1. | 1 causes the LABEL table to be Xeroxed
2. | 2 causes the SYMBOL table to be Xeroxed
3. | 4 causes a formatted listing to be Xeroxed
4. | 10 disables the compilation of code to check subscript bounds and SWITCH bounds.
5. | 20 disables the compilation of code to check the data type of LEAP items when γ is used.

These mode numbers may be combined:

e.g. | 7 causes all three listings to be Xeroxed

C. SYNONYM FEATURE

One may define a synonym to a declared variable or to a procedure in LEAP; for example, if XYZ is a declared variable or a procedure, then

W ≡ XYZ;

is a DECLARATION which will assign the "semantics" of XYZ to W. Subsequent reference to either XYZ or W will have identical meaning.

D. NO KEYBOARD BUFFER OPTION

The following declaration, occurring anywhere in a declaration portion of a LEAP program, will suppress the assignment of a keyboard buffer at run time:

NOKBBF;

E. SEGMENTING A LEAP PROGRAM DIRECTIVE

There is a feature in LEAP which allows the compilation of a LEAP program from text files rather than from a VITAL directive. This is useful if the directive is larger than two books, or if core space at compile time is at a premium. Only one text file is set up at a time during compilation.

Note that:

(1) The user cannot ask VITAL for a program listing, or for a formatted listing.

(2) Compile-time error messages will usually be garbled.

The use of this feature is described below:

The user deals with his program in text file form. He may direct the compiler to take its input from the keyboard, and proceed to specify the text files which are to be read in. The compilation is then performed.

The user directs the compiler's attention to the keyboard by asking to compile a program consisting of one special word:

GETFROMKB

He specifies that a text file be read in by typing the name of

the text file followed by the READ-IN key. For example, if BLOP is a large program, having TAG as a label about halfway through, then the following sequence of events in VITAL will compile BLOP:

TYPED BY	
SYSTEM	CLEAN
USER	←←L 5LEAP
USER	←←C BLOP
USER	←←DIR F|| #→TAG
USER	←←DIR FF|| TAG→ ?
USER	←←FRESH
SYSTEM	FRESH
USER	←←INS #
USER	GETFROMKB
USER	[Y]
USER	←←BBIN
USER	F [R]
USER	FF [R]
SYSTEM	←←

F. OTHER RESERVED PROCEDURES

1. SHADE;
   This causes the user to go into the shade.

2. HELP;
   This causes a HELP call.

3. ASSIGNRECOGNIZER (<TEXTARRAY>);
   The TEXTARRAY parameter indicates the name of the file which is to be used henceforth as the character recognizer.

4. There is a reserved procedure which calls the character recognizer:*
   RECOGNIZE;
   When this procedure is called, a full inking buffer should be available, and the ASSIGNRECOGNIZER procedure should previously have been called.

*For information on the TX-2 drawn character recognition facility, see reference 3.

The following reserved variables will be set up by the procedure:

αCHAR	(INTEGER)	character code (-1 if no recognition)
αXMAX	(FIXED)	maximum X coordinate
αXMIN	(FIXED)	minimum X coordinate
αYMAX	(FIXED)	maximum Y coordinate
αYMIN	(FIXED)	minimum Y coordinate
αXCEN	(FIXED)	X coordinate of center
αYCEN	(FIXED)	Y coordinate of center

EXAMPLE:

(reserved words are underlined)

```
START
INPUT  {FILE REC };
  .
  .
  .
ASSIGNRECOGNIZER  {REC};
  .
  .
  .
GETNEXTINT;
IF αCAUSE = 17. THEN
    BEGIN
        RECOGNIZE;
        IF αCHAR = -1 THEN HELP;
          .
          .
          .
    END;
.
.
.
FINISH
```

5. A reserved procedure for "going up to" the character-recognition trainer (5TRAIN). This procedure expects the ASSIGNRECOGNIZER procedure to have previously been called.

```
TRAIN;
```

6. A procedure which takes a TEXTARRAY as a parameter, and "goes up to" the scope editor

```
EDIT  { < TEXTARRAY >};
```

7. Two reserved procedures for allocating and emptying books at run time:

   (a) FREEBOOK - an INTEGER function which requires no parameters, and returns the number of an empty book (1 thru 17) as its value.
   Book 0 is automatically free for use; allocation of other free books must be done through FREEBOOK.

   (b) EMPTYBOOK {<INTEGER quantity>};
   A procedure which causes the indicated book to be emptied (JED 123).

8. KEYBOARDEDIT {<TEXTARRAY>}; This calls the keyboard editor, with the indicated TEXTARRAY as input. It works just like the EDIT procedure.

9. BASICTRANSLATE {<TEXTARRAY>};
   This passes the indicated text up to 5BTF.

G. EXTERNAL PROCEDURES

This is a facility for defining a procedure or function that during run-time will exist outside the LEAP system. The experienced user will find this useful in linking MARK 5 and LEAP programs. The following is the external procedure declaration form:

EXTERNAL <LOC>, <normal procedure definition header>;

where LOC should be an octal integer constant defining the absolute location of the procedure and a regular procedure definition follows.

Example: EXTERNAL 411.,REAL PROCEDURE SUMSQ (REAL A1, A2);

defines the real function SUMSQ at location 411_8 with two real arguments.

The calling sequence generated by LEAP is

JES_{10} LOC
address of argument 1
address of argument 2
.
.
address of argument n
expected return point →

H. LIST OF RESERVED WORDS AND SYMBOLS

Note that all Mark 5 op-codes are also reserved words in LEAP.

ACTIVATE
ARRAY
BASICTRANSLATE
CALL
CLEARKBDLINE
CLOSETF
COSR
DELITEM
DUMP
END
EXTERNAL
FOR
GATHER
GOUPTO
INKX
IS
ITEM
KNOB3
LOADLINE
MAKE
NEWITEM
OVERLAYRETURN
PROPERTY
PUTDOTMAT
PUTPNT
RDERRCODE
READNUM
RECOGNIZE
RESTORE
SETHASHMASK
SETSMACKER
SETUPWRITEBUF
SQRT
STORETEXT
TBLTX
THRU
TOG4
TOG9
TURNOFFINTS
USEGRP
WHEN
XEROX

AND
ASSIGN
BEGIN
CLEANSTRUCTURE
CLEAROUTPUT
COMMENT
DEACTIVATE
DO
EDIT
ENDOFLINE
FALSE
FOREACH
GETFROMKB
HELP
INKY
ISCHARINPUT
ITEMVAR
KNOB4
LOADPNT
MATRIX
NOKBBF
PEELBACK
PUSHFILEERRORLABEL
PUTITEM
PUTSD
RDTATYPE
READSTRUCTURE
REMOVE
RETURN
SETITEM
SETSTATUSOF
SHADE
SQRTR
SWITCH
TBLTY
TO
TOG5
TOG8
TYPEERROR
USEGRPONCE
WHILE
#C

ANDB
ASSIGNRECOGNIZER
BLANKITEM
CLEARINK
CLEARSCOPE
CONTINUE
DEFINE
DROPOVERLAY
EIR
ENDWHEN
FILE
FORMAT
GETNEXTINT
IF
INPTR
ISSTRUCTURE
KEYBOARDEDIT
KNOB5
LOCAL
MERGESTRUCTURE
NUMPOINTS
POPFILEERRORLABEL
PUT
PUTITEMONCE
PUTTAINTF
READ
READTF
REPORTING
RTC
SETMSCALE
SETUPANDNAMEFILE
SHOWTEXT
START
TBLTSW1
TEXTARRAY
TOG1
TOG6
TRAIN
TYPEOUTERROR
USELEAP
WRITEITEM
#R

ANY
ATAN
BOOLEAN
CLEARINTS
CLEARSTORESCOPE
COR
DELETE
DROPSTRUCTURE
ELSE
ERASE
FINISH
FREEBOOK
GO
IFNOT
INPUT
ISTRIPLE
KNOB1
LABEL
LOG
META
OPENTF
PRINT
PUTCHARINTF
PUTLINE
PUTTEXT
READCHAR
REAL
REPORTSTATUSOF
SAVE
SETPENMODE
SETUPFILE
SIN
STEP
TBLTSW2
THEN
TOG2
TOG7
TRUE
UNBLANKITEM
VIA
WRITESTRUCTURE
@CAUSE

APPEND
ATANR
BY
CLEARKBD
CLEARWRITEBUF
COS
DELGRP
DROPWRITEBUF
EMPTYBOOK
EXP
FIXED
FROM
GOTO
IN
INTEGER
ISWORDINPUT
KNOB2
LASTINPTR
LOGE
MOVUSE
OUTPUT
PROCEDURE
PUTDOTLINE
PUTMAT
PYTHAG
READINTEXTFILE
RECLAIM
RESETITEMID
SET
SETRDERLBL
SETUPTEMPFILE
SINR
STOREITEM
TBLTSW3
THROUGH
TOG3
TOG8
TURNOFF
UNTIL
WHATSIN
WRITETEXT
@CHAR

aCHARCNT
aCONSOLE
aDATATYPE
aEIR
aEXECUTABLE
aEXPANDABLE
aFILEERRORCODE
aFILENAME
aGRPSEEN
aITEMSEEN
aKNOB1
aKNOB2
aKNOB3
aKNOB4
aLENGTH
aMAP
aNUMSTROKES
aPROTECT
aROUTINECODE
aRTC
aSUMMARY
aTBLTSW1
aTBLTSW2
aTBLTSW3
aTBLTX
aTBLTY
aTFCOUNT
aWHICH
aWRITEABLE
aXCEN
aXMAX
aXMIN
aYCEN
aYMAX
aYMIN
BDATATYPE
BEIR
BEXECUTABLE
BEXPANDABLE
BINKING
BINTM1
BINTM2
BINTM3
BINTM4
BINTM5
BKEYBOARD
BKNOB5
BLENGTH
BNONEXECUTABLE
BNONEXPANDABLE
BPROTECT
BREADONLY
BRTC
BSWCHANGE
BTARGET
BTBLTPOS
BTRACKEND
BTRACKING
BTRACKSTART
BUNPROTECT
BWRITEABLE

LEA RESERVED SYMBOLS

⊖ · + − / — ~) A D E F T R R S T V Z U C 0 1 2 3 ← = PSUP PSUB PNOR → ≡ ▶ ↕ ↑ ↓ ≠

APPENDIX VIII
PRIMITIVES FOR DATA-STRUCTURING

This appendix presents a user's-eye view of the data-structuring facilities in LEAP. The first part of the appendix is a condensation of a paper* on this topic, and is included here as a user's introduction to these facilities. The remainder of the appendix is an annotated tabulation of the language forms for data structuring.

Part I. Introduction to the Associative Sub-language

The basic data-structure entity used in LEAP is an associative TRIPLE of the form

ATTRIBUTE of OBJECT is VALUE

(e.g., FATHER of JOE is PETE). The data structure is a store of facts in this form. The hash-coded nature of the data-structure makes it amenable to paging techniques. Programming constructs are available for creating, deleting, and searching for elements in the data store. Of particular importance is the uniformity of the single data form used. A LEAP user does not have to consider the details of a complex structure in computer memory; he can concentrate on what he wishes to represent and not how to represent it. The programming facilities available include set-theoretic operations, a powerful fact search and retrieval facility, and the ability to use a TRIPLE itself as a component of another TRIPLE.

In the discussion below, reserved words in the language are underlined.

A. COMPONENTS OF A LEAP DATA STRUCTURE

Conceptually, a LEAP data structure consists of a universe of ITEMs,** a universe of TRIPLEs, and a number of SETs.

1. ITEMs

An ITEM is an entity whose "internal identifier" (name) is manipulated by the LEAP system. An ITEM may have an associated "datum", this must be specified to have one of the data types of the base language,

*For further details, see references 4, 5, and 10.

**Not to be confused with "display items".

which we will refer to as "algebraic types." Some allowed algebraic types are listed below

real*

integer

boolean

fixed

(real, integer, boolean, or fixed) array

For example, the declaration

real array item LINE4;

would specify an entity, LINE4, whose datum was an array of real numbers, perhaps containing end-point coordinates. The LEAP language contains elements which are used only as algebraic quantities, only as names (ITEMs without algebraic type), and in both ways (ITEMs with algebraic type).

The LEAP language has various statements for creating ITEMs and entering them into the initially empty universe of ITEMs. Declaring an ITEM will enter it at compile-time; the facilities for dynamically entering a new ITEM at execution time are presented in Section B.

2. TRIPLEs

The TRIPLE is an ordered collection of three ITEMs and is used to represent a fact in the relational structure. A TRIPLE is created and entered into the initially empty universe of TRIPLEs via the MAKE statement.

For example, if FATHER, JOHN, and PETE are ITEMs, then execution of

(1) make FATHER·JOHN ≡ PETE; (read "FATHER of JOHN is PETE")

will add the indicated TRIPLE to the universe of TRIPLEs. In (1), "·" and "≡" are reserved symbols.

A TRIPLE may be removed from the universe of TRIPLEs via the ERASE statement:

(2) erase FATHER·JOHN ≡ PETE;

3. SETs

A SET is an unordered collection of ITEMs. SETs are created by a SET declaration, (e.g., set SONS;).

*In this appendix, reserved words are underlined.

Initially, a SET is empty (has no ITEMs). An ITEM may be added to a SET via the PUT statement:

(3) put JOE in SONS;

An ITEM may be removed from a SET via the REMOVE statement:

(4) remove JOE from SONS;

4. ITEMVARs (ITEM variables)

An ITEMVAR has an ITEM as its value. An ITEM (e.g., JOE) may be assigned to an ITEMVAR (e.g., X) via the assignment statement:

(5) JOE → X;

X now "represents" the ITEM JOE in the sense that the following two statements have the same meaning:

(6) make FATHER·JOE ≡ PETE;

(7) make FATHER.X ≡ PETE;

ITEMVARs may be declared with or without an algebraic type (example: real itemvar X;). The algebraic type specification is necessary in case it is ever desirable to retrieve the datum of the ITEM that is currently represented by the ITEMVAR. In such a case, the system assumes that the algebraic type of the ITEM represented is the same as the algebraic type of the ITEMVAR.

An ITEMVAR may always be used in place of an ITEM.

5. LOCALs

A LOCAL also has an ITEM as its value. The LOCAL is used as the iteration variable in the FOREACH statement. It is used as a "local ITEMVAR" within the scope of this statement, hence its name. A discussion of the FOREACH statement and the use of LOCALs is presented in Section G.

B. DYNAMIC CREATION AND DELETION OF ITEMs

1. DYNAMIC ITEM CREATION

There are two ways to create new ITEMs dynamically (at execution time):

(a) via the statement

(8) newitem → X;

where "X" is an ITEMVAR. This statement causes a new ITEM to be generated and assigned to the ITEMVAR and space allocated for its datum (unless this is an array). The algebraic type of this datum is assumed to be the same as the algebraic type of the ITEMVAR. If the ITEMVAR was not declared with an algebraic type, then no space is allocated, and the new ITEM is assumed to have no datum.

(b) via the unary operator, "n ". This operates on an arbitrary algebraic expression and yields a new ITEM having the evaluated expression as its datum.

2. DYNAMIC ITEM DELETION

The following statement removes the ITEM represented by X from the universe of ITEMs.

(9) reclaim X;

Execution of this statement causes the internal identifier of the indicated ITEM to be placed on a list of available internal identifiers, and the storage allocated for the datum of the ITEM (if any) to be returned to free storage. ITEMs which were declared via an ITEM declaration may not be deleted. It is the user's responsibility to make sure that an ITEM is not a member of any SET nor a part of any TRIPLE when it is deleted.

C. THE ITEM EXPRESSION

Thus far, we have mentioned three ways to represent an ITEM:

(a) by a declared ITEM identifier

(b) by an ITEMVAR which has been assigned an ITEM

(c) by "n" applied to an algebraic expression.

We will classify these as "ITEM expressions." An ITEM expression may always be used in place of an ITEM.

In addition, a TRIPLE form can be an ITEM expression. This feature allows the use of a TRIPLE as part of another TRIPLE. For example, the following statement creates a TRIPLE which expresses the idea that "the number of lines in a square is four":

(10) make NUMBER·(PART·SQUARE ≡ LINE) ≡ n 4;

In (10), "NUMBER," "PART,""SQUARE", and "LINE" are ITEMs. The TRIPLE

PART·SQUARE ≡ LINE

should exist in the universe of TRIPLEs before (10) is executed. The ITEM "NUMBER" is meant to represent an attribute which applies to all part-whole relationships.

D. SET EXPRESSIONS

A declared SET is a SET expression.

ϕ is a SET expression (the empty SET).

A list of ITEM expressions separated by commas, all enclosed in brackets "{" and "}", is a SET expression (example: {PETE, JOE}).

Two ITEM expressions combined by one of the binary association operators (., ', *) is a SET expression. The evaluation of these SET expressions requires extracting information from the universe of triples, as follows:

If A and B are the two specified ITEM expressions, then

(a) A·B is the SET of all X such that

A·B ≡ X

(b) A'B is the SET of all X such that

A·X ≡ B

(c) A*B is the SET of all X such that

X·A ≡ B

The special reserved word ANY may be used in place of an ITEM expression in a binary association operation, implying that any ITEM in the indicated position will match.

Example:

(11) FATHER·ANY

is the SET of all fathers.

E. SET STATEMENTS

The SET ASSIGNMENT statement may be used to assign an arbitrary SET expression to a declared SET, (e.g., (12) SONS ∪ BROTHERS → SONS).

There is a special statement in LEAP for performing a task for each ITEM in a SET.

For example, if SONS is a SET and X is a LOCAL,

(13) foreach X in SONS do <STATEMENT>;

will cause the <STATEMENT> to be executed once for each ITEM in the SET. Before each iteration, the next ITEM in the SET is assigned to the LOCAL. Within the scope of the FOREACH statement, the LOCAL behaves like an ITEMVAR. A complete discussion of the FOREACH statement is presented in Section G.

F. LEAP OPERATORS WHICH YIELD ALGEBRAIC RESULTS

The binary operators "ϵ", "$\subset$", and "$=$" and the unary operators istriple, "$||$", and "γ" yield algebraic results. Four of these operators deal with SET expressions:

(a) <ITEM expression> ϵ <SET expression> is a Boolean expression which has the value TRUE if the indicated ITEM is a member of the indicated SET, and FALSE otherwise.

(b) $||$ <SET expression> is an INTEGER expression whose value is the number of ITEMs in the indicated SET.

(c) <SET expression> $\subset$ <SET expression> is a Boolean expression which has the value TRUE if the left operand is a subset of the right operand, and FALSE otherwise.

(d) <SET expression> = <SET expression> is a Boolean expression which has the value TRUE if the left operand equals the right operand (i.e., the left SET is a subset of the right SET, and vice versa), and FALSE otherwise.

The unary operator "γ" (GAMMA) operates on an ITEM expression to yield the datum of the indicated ITEM.

For example, if PETE is an INTEGER ITEM, then the following statement assigns 40 as the datum of PETE:

(14) 40 →γ PETE;

The unary operator istriple operates on a TRIPLE form to yield a Boolean result. This result has the value TRUE if the indicated TRIPLE exists in the store.

G. ASSOCIATIVE FOREACH STATEMENT

There is a special statement for retrieving information from the universe of TRIPLEs. It allows one to specify the context in which the information of interest is to be found rather than a procedure for finding that information.

For example, the following finds PETE's sons:

(15) foreach FATHER·X ≡ PETE and
SEX·X ≡ MALE do
<STATEMENT>;

In (15), X is a LOCAL.

There are two "context specifications" in (15):

(a) SEX·X ≡ MALE

(b) FATHER·X ≡ PETE

These serve to determine the collection of ITEMs represented by the LOCAL X. An ITEM will be in this collection if and only if it satisfies all "context specifications." In general, there may be many "context specifications" in a FOREACH statement.

At execution time, a collection of ITEMs is calculated for the LOCAL from the context specifications. The <STATEMENT> is then executed once for each ITEM in this collection. Before each iteration, the next ITEM is assigned to the LOCAL. Within the limits of the <STATEMENT>, the LOCAL is treated like an ITEMVAR. The difference between an ITEMVAR and a LOCAL is only that the LOCAL has special meaning within the FOREACH state-

ment and no meaning outside of this statement. An ITEM may be assigned to a LOCAL only by the internal action of the FOREACH statement. This action is said to "bind" the LOCAL. Within the FOREACH statement, the LOCAL is termed "bound." Outside the FOREACH statement, the LOCAL is undefined.

FOREACH statements may be nested; a LOCAL which has been "bound" by a FOREACH statement is treated like an ITEMVAR everywhere within the scope of that statement.

More than one LOCAL may be "bound" by a FOREACH statement. In this case, if there are N LOCALs, then a collection of N-tuples of ITEMs is calculated when the context specifications are processed. The <STATEMENT> is executed once for each N-tuple in this collection; the appropriate ITEMs are assigned to the appropriate LOCALs before each iteration.

For example, the following statement would create all paternal grandfather relationships:

```
foreach FATHER·X ≡ Y
and     FATHER·Y ≡ Z  do
make PGRFATHER·X ≡ Z;
```

Usually, the three operands of a "context specification" of the TRIPLE form may be any ITEM expressions. There are cases which are ill-defined; the compiler makes the following restrictions:

(a) At least one operand must be a LOCAL which is being "bound" by this statement.

(b) The three operands cannot all be LOCALs which are being "bound."

(c) The item expression

    n <algebraic expression>

    is not allowed.

The following constructs (specified in BNF) are other allowed operands for a "context specification" of the TRIPLE form:

```
<other allowed operand> : : =
<ITEM expression> <binary association operator> <LOCAL> |
```

<LOCAL> <binary association operator> <ITEM expression> |

<ITEM expression> <binary association operator> <other allowed operand>

An example of the use of these constructs follows. In this example there are TRIPLEs having the following forms:

ABOVE·SQUARE ≡ [OBJECT]

PART·[OBJECT] ≡ [LINE]

"ABOVE," "SQUARE," and "PART" are ITEMs, and "OBJECT" and "LINE" represent the meaning of ITEMs found in the indicated context. The following statement will display all objects above the square:

(16) foreach ABOVE·SQUARE ≡ PART ' Z do DISPLAY {γZ};

In (16), "Z" is a LOCAL with declared data-type REAL ARRAY, the DISPLAY procedure expects a REAL ARRAY (representing a line) as a parameter, and an ITEM which represents a line has a REAL ARRAY as its datum.

The statement (16) may be expressed another way:

(17) foreach PART·Y ≡ Z and
ABOVE·SQUARE ≡ Y do
DISPLAY {γZ};

In addition to the TRIPLE form the following construct is allowed as a "context specification" in the associative FOREACH statement:

<LOCAL> in <SET expression>

This "context specification" restricts the collection of ITEMs represented by the LOCAL by requiring that each such ITEM be an element in the SET expression.

NOTE: Do not attempt to terminate a foreach statememt by a GOTO to a label outside the scope of the statement.

Part II. Annotated Tabulations of the Language Forms for Data-Structuring

DECLARATIONS

property
item
itemvar
local
set

<list of identifiers>;

real
boolean
integer
fixed
set
matrix
textarray

item
itemvar
local

<list of identifiers> ;

real
integer
boolean
fixed

array

ITEM EXPRESSIONS

FORM	NOTES
t <integer expression>	represents the ITEM which has the indicated integer as its internal identifier
n <algebraic expression or set expression>	generates a new ITEM and assigns the evaluated expression as its datum
<declared ITEM, ITEMVAR, or LOCAL>	
(<Boolean expression> ⊃ <item expression>, <item expression>)	
(<item expression>)	
<set expression> → <itemvar> <item expression> → <itemvar>	if a nested store
(<item expression> · <item expression> ≡ <item expression>)	represents the ITEM which has this TRIPLE as its datum (if this is not the argument to istriple).
[1] <item expression>	represents the first item in the triple which is the datum of the indicated item
[2] <item expression>	represents the second item in the triple which is the datum of the indicated item
[3] <item expression>	represents the third item in the triple which is the datum of the indicated item

(from the left)

SET EXPRESSIONS

FORM	NOTES
{<list of item expressions>}	
<declared set>	
<set expr.> ∪ <set expr.>	set union
<set expr.> ∩ <set expr.>	set intersection
<set expr.> - <set expr.>	set subtraction
<set expr.> → <set identifier>	if nested store
(<Boolean expr.> ⊃ <set expr.>, <set expr.>)	
<item expr.> . <item expr.>	if a·b, value is the set {x| a·b ≡ x}
<item expr.> ' <item expr.>	if a'b, value is the set {x| a·x ≡ b}
<item expr.> * <item expr.>	if a*b, value is the set {x| x·a ≡ b}
<item expr.> . <set expr.>	if a·b, value is the union, for every y in b, of the sets {x| a·y ≡ x}
<item expr.> ' <set expr.>	if a'b, value is the union, for every y in b, of the sets {x| a·x ≡ y}
<item expr.> * <set expr.>	if a*b, value is the union, for every y in b, of the sets {x| x·a ≡ y}

ALGEBRAIC EXPRESSIONS

FORM	TYPE	NOTES
i <item expr.>	INTEGER	value is the internal identifier of the indicated item.
‖ <set expr.>	INTEGER	value is the number of items in the indicated set.
<item expr.> ε <set expr.>	BOOLEAN	TRUE if the item is in the set, and false otherwise.
istriple (<item expr.> · <item expr.> ≡ <item expr.>*	BOOLEAN	TRUE if the triple exists in the store of triples.
<item expr.> is <property>	BOOLEAN	TRUE if the item has the property.
γ <item expr.>	(type of the item)	value is the datum of the indicated item.
<item expr.> = <item expr.>	BOOLEAN	TRUE if the two items are the same.
<item expr.> ≠ <item expr.>	BOOLEAN	TRUE if the two items are not the same.
<set expr.> = <set expr.>	BOOLEAN	TRUE if the two sets contain exactly the same items.
<set expr.> ⊂ <set expr.>	BOOLEAN	TRUE if the left set is a subset of the right set.
isstructure {<text array>}	BOOLEAN	TRUE if a LEAP data-structure of the indicated name exists in the user's directory.

---

* The reserved word any is permitted in place of (at most two of) the item expressions in this form.

STATEMENTS

FORM	NOTES
assign <property> to <item expr.>;	
delete <property> from <item expr.>;	
put <item expr.> in <declared set>;	
remove <item expr.> from <declared set>;	
make / erase <item expr.> · <item expr.> ≡ <item expr.>;	add (or delete) the indicated triple to (from) the store of triples.
writestructure / readstructure / dropstructure / mergestructure <text array>;	save the entire data structure under the indicated name. replace the current data structure with the indicated saved data structure. drop the indicated saved data structure. append the indicated structure to the current working data structure
cleanstructure ;	re-initialize the current working data structure
newitem → <itemvar>;	generate a new item
reclaim <itemvar>;	
<item expr.> → <itemvar>;	
<set expr.> → <declared set>;	
<set expr.> ⊖ <itemvar>;	assign one of the items in the indicated set to the itemvar.

foreach <ardlist> do <statement>;

(where)

<ardlist>: : = <ard> | <ardlist> and <ard> | <ardlist> andb <Boolean expr.>

<ard>: : = <item expr.> · <item expr.> ≡ < ardpar.> > | <local> in <set expr.>

<ardpar>:: = <item expr.>|<item expr.><bap><locals> | < local><bap><item expr.> | <item expr.> <bap><ardpar>

<bap>:: = ·| ' | *

NOTES:

(1) Items, properties, and locals may be declared only at the beginning of a LEAP program.

(2) The word useleap must follow start in every LEAP program in which the associative sublanguage is used.

(3) When a procedure is declared, one can specify either an item or an itemvar as a parameter; the first specifies a value parameter, the second a reference parameter. One can pass any item expression as an item parameter; only an itemvar may be passed as an itemvar parameter.

(4) There is a facility for binding locals with an arbitrary Boolean expression in the upper part of the foreach statement. A Boolean expression in this context must be preceded by andb. Example:

```
foreach X in S andb γ X<6 do. . .
```

(5) There is a facility for declaring and using up to six "properties" in LEAP. A property may be assigned to or removed from an item, if the item has a datum. Also, one may ask if a specified item has a specified property.

Examples:

```
begin
        real itemvar X;
        set S;
        property A, B;
        real     local Y;
          :
        newitem → X;
        assign A to X;
          :
        foreach Y in S do
        if Y is A then delete A from X;
          :
end
```

(6) When using writestructure and readstructure, the user should be careful that the program in which a structure is written should be "compatible" with the program in which the structure is read. This simply means that the item declarations in the two programs should correspond. Since there is no symbolic communication between LEAP programs, only the order of declaration of items, the data-type of the corresponding items, and the total number of declared items should match.

(7) The mergestructure statement allows the user to merge "compatible" LEAP data structures.

Example:

```
mergestructure 'ABC';
```

when this statement is executed, the structure named 'ABC' will be appended to the current structure in the following way:

(a) All items in 'ABC' except those declared with no data-type will be added as new items to the store of items, and

(b) All triples in 'ABC' will be adjusted to preserve the relations between these new items and then will be added to the store of triples.

Note that both the current structure and the structure to be merged should have no items which were declared with a data type.

(8) itemvars are treated as simple variables (e.g. real) in declarations (i.e. the existence of an itemvar declaration in a compound statement does not make the compound statement a BLOCK).

(9) sets are treated as dynamic variables in declarations.

APPENDIX IX

PRIMITIVES FOR TEXT AND FILE MANIPULATION

This appendix describes a set of reserved procedures for manipulating textarrays and APEX files. The names of files are textarrays containing the text of the name. Procedures are provided for setting up files, reporting which files are set up, setting and reporting file status information, reading and writing text files, and combining textarrays.

A flexible error handling facility has been implemented to allow for the variety of errors that can occur. Should an error occur, a jump to the label at the top of a stack will be executed after the reserved variables αROUTINECODE and αFILEERRORCODE have been set to indicate the cause of the error. Initially, the stack contains a jump to an internal routine which prints a "canned" message regarding the source of the error and then calls help. The user, however, may push and pop this stack of error labels to set his own handling of errors in various parts of his program.

A description of the routines and reserved variables follows:

SETUPTEMPFILE {LENGTH, BOOK};

This routine sets up a non-executable, auto-expandable, ephemeral file of LENGTH pages in the specified book. The reserved textarray variable αFILENAME is set to the name of the file (e.g., '43E201763').

SETUPFILE {NAME, BOOK};

This routine sets up the indicated file in the specified book using the status information in the directory. If the file doesn't exist, an error is signalled.

SETUPANDNAMEFILE {NAME, LENGTH, BOOK};

This routine sets up a new file with the indicated name, status non-executable, auto-expandable, of LENGTH pages, in the specified book. Any previous uses of the name are dropped.

WHATSIN {BOOK, MAP, NAME};

This routine sets NAME to the name of the file in the specified book and map. If that slot is empty, NAME is set to the null textarray (i.e., ' ').

SETSTATUSOF {NAME, WHICH};

The status of the file NAME is set to be the status specified by the integer WHICH; typically this is an "OR'ing" of the following reserved variables:

βEXECUTABLE	βNONEXECUTABLE
βWRITEABLE	βREADONLY
βUNPROTECT	βPROTECT
βEXPANDABLE	βNONEXPANDABLE

βLENGTH (Set length to αLENGTH pages).
βDATATYPE (Set datatype to αDATATYPE).

examples: Set the status of file RR to be Read-Only and 10 pages in length. Leave other status information unchanged.

```
10 → αLENGTH;
SETSTATUSOF {'RR', βREADONLY ∨ βLENGTH};
```

REPORTSTATUSOF {NAME};

The status of the file NAME is used to set the values of the following reserved variables:

αSUMMARY	INTEGER	1 LEGAL DEFINED FILE 0 LEGAL DIRECTORY NAME, BUT NOT DEFINED -1 NOT LEGAL DIRECTORY NAME
αLENGTH	INTEGER	No. of PAGES
αDATATYPE	INTEGER	DATATYPE, VALUES HAVE THE FOLLOWING MEANINGS: 0 - UNSPECIFIED (BINARY) 1 - LINCOLNWRITER TEXT 2 - PROCESS 3 - ARRAY 4 - LIBRARY FILE 5 - MK4/5 DIRECTIVE 6 - USER DIRECTORY 7 - TABLET RECOGNIZER DICTIONARY 8 - EXTENDED LINCOLNWRITER TEXT FILE 9 - RELOCATABLE BINARY 10 - TAP ASSEMBLER DICTIONARY 11 - COMPILED REGULAR EXPRESSION
αWHICH	INTEGER	This, if supplied in a SETSTATUSOF would set status identical to this file. e.g., To set file RR to identical status with file RA, except that RR is one page longer, the following code would be written: REPORTSTATUSOF {'RA'}; αLENGTH + 1 → αLENGTH; SETSTATUSOF {'RR', αWHICH};
αEXECUTABLE	BOOLEAN	
αWRITEABLE	BOOLEAN	
αPROTECT	BOOLEAN	
αEXPANDABLE	BOOLEAN	

READTF {NAME, TA};

The contents of the textarray TA is set to be the contents of the text file NAME.

OPENTF;

Prepares to build a new text file. Only one text file may be under construction (i.e., open) at one time.

CLOSETF {NAME};

Close the current text file and name it NAME.

PUTCHARINTF {INTGR};

Puts the character INTGR into the next character location in the open text file.

PUTTAINTF {TA};

Appends the indicated text to the open text file.

APPEND {A, B, C};

The textarray B is appended to the textarray A and the result is put into textarray C. Any two or all three may be the same textarray or the null textarray.

PUSHFILEERRORLABEL {LABEL};

If an error is discovered in the file package, or a DOFILEERROR is executed, control will be passed to the last LABEL pushed. The reserved variables αROUTINECODE and αFILEERRORCODE will be set to indicate the cause of the error. These variables are volatile over any reserved procedure and should thus be saved quickly. If no error is detected in a reserved procedure, the values of αROUTINECODE and αFILEERRORCODE are not defined. αROUTINECODE and αFILEERRORCODE are safe over the following reserved procedures: PUSHFILEERRORLABEL, POPFILEERRORLABEL.

POPFILEERRORLABEL;

Cancels the last PUSHFILEERRORLABEL executed. If you try to POP too far, the top of the stack will contain its initial value, namely a label of routine which types the cause of the error and then calls help.

DOFILEERROR {ROUTINECODE, FILEERRORCODE};

Sets the reserved variables αROUTINECODE and αFILEERRORCODE to the specified values and then jumps to the last label pushed onto the file error label stack.

TYPEERROR;

This routine takes the values in αROUTINECODE and αFILEERRORCODE and uses them to print a canned message as to the cause of the error.

RESERVED VARIABLE	DATA TYPE	COMMENTS
αTFCOUNT	INTEGER	contains number of characters inserted into currently open text file. Contains -1 if no text file currently open.
αROUTINECODE	INTEGER	See discussion at PUSHFILE-ERRORLABEL and listing of errors.
αFILEERRORCODE	INTEGER	
αFILENAME	TEXTARRAY	Set to the name of the file set up by the last SETUPTEMPFILE.
αDATATYPE	INTEGER	See REPORTSTATUSOF
αLENGTH	INTEGER	
αSUMMARY	INTEGER	
αWHICH	INTEGER	
αEXECUTABLE	BOOLEAN	
αEXPANDABLE	BOOLEAN	
αWRITEABLE	BOOLEAN	
αPROTECT	BOOLEAN	
αMAP	INTEGER	Set at initialization to the number of the map the user is running on.
αCONSOLE	INTEGER	Set at initialization to the number of the console on which the user is running.

REFERENCES

1) Blatt, H., "Conic Display Generator Using Multiple Digital-Analog Decoders:" PROC FJCC (1967).

2) Clark, W. A. et. al., "The Lincoln TX-2 Computer", Proc. Western JCC (February 1957).

3) Curry, J. E., "A Tablet Input Facility for an Interactive Graphics System", PROC of International Conference on Artificial Intelligence, Washington, D. C. (May 1969).

4) Feldman, J. A. and Rovner, P. D., "An ALGOL-Based Associative Language", CACM (August 1969).

5) Feldman, J. A., "Aspects of Associative Processing", Technical Note 1965-13, Lincoln Laboratory, M.I.T. (21 April 1965), DDC AD-614634

6) Forgie, J. W., "A Time and Memory Sharing Executive Program for Quick-Response on-line Applications", PROC. FJCC (1965).

7) Mondshein, L. F., "VITAL Compiler-Compiler System Reference Manual", Technical Note 1967-12, Lincoln Laboratory, M.I.T. (8 February 1967), DDC AD-649140.

8) Naur, P. et. al., "Revised Report on the Algorithmic Language ALGOL-60. CACM (January 1963).

9) Roberts, L. G., "Homogeneous Matrix Representation and Manipulation of N-Dimensional Constructs", The Computer Display Review, published by Keydata Associates, Bedford, Massachusetts.

10) Rovner, P. D. and Feldman, J. A., "The LEAP Language and Data Structure", IFIP Congress 1968 (August 1968).

11) Sutherland, W. R. et. al., "Graphics in Time-Sharing: A Summary of the TX-2 Experience" PROC SJCC (1969).

12) Teixera, J. F. and Sallen, R. P., "The Sylvania Data Tablet", Proc. SJCC (1968).

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)* Lincoln Laboratory, M.I.T.	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP None

3. REPORT TITLE

The LEAP User's Manual

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

Lincoln Manual

5. AUTHOR(S) *(Last name, first name, initial)*

Rovner, Paul D.

6. REPORT DATE 11 September 1970	7a. TOTAL NO. OF PAGES 100	7b. NO. OF REFS 12
8a. CONTRACT OR GRANT NO. AF 19(628)-5167 b. PROJECT NO. ARPA Order 691 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Lincoln Manual 93 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* ESD-TR-70-256	

10. AVAILABILITY/LIMITATION NOTICES

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

11. SUPPLEMENTARY NOTES None	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency, Department of Defense

13. ABSTRACT

This document is a user's manual for the LEAP language. LEAP is an extended algebraic programming language which is similar in form to ALGOL.8 Extensions include language forms for display output and interactive input and facilities for building and manipulating associative information structures. The basic algebraic language is described in Sections I through IX; the extensions to LEAP are presented in the Appendices.

14. KEY WORDS

computer language | LEAP language | extended algebraic language