GUIDE TO THE LITTLE LANGUAGE

David Shields

October 4, 1982

(A Table of Contents appears at the end.)

PREFACE

This report is the basic document describing the LITTLE Programming Language. The central part of this system is the programming language LITTLE (the name reflects what at one time seemed to be rather modest project goals), and a standard compiler for LITTLE written in LITTLE.

The goal of the LITTLE project is to provide a means for writing software which is portable and efficient. An early version of LITTLE was defined in 1968; serious work began in 1971, when LITTLE was chosen as the implementation language for SETL, a very high level language with finite sets as its fundamental data type. The relation between the two projects is important. The SETL project has provided an active group of users for LITTLE; these users expect a quality compiler. The need to support the 'production' use of LITTLE has given much of the work on LITTLE a pragmatic flavor; this has been a real benefit, as a major problem in language design is to balance the desire for 'abstract consistency' with the actual needs of the users. The SETL implementation work has had minimal impact on the LITTLE language, as both groups agree that LITTLE should be a separate programming language, able to stand on its own merits.

LITTLE has been implemented on several systems, including CDC 6000, IBM System/370, Digital Equipment DECsystem-10, and Digital Equipment VAX-11/780. A substantial amount of software has been written in LITTLE, including the LITTLE system itself, several implementations of SETL, an operating system, and a translator for the MINIMAL language.

This version adds support of floating point operations, input/output facilities, and makes character strings easier to use. Recent work has made it even more clear that an essential part of a portable programming system is a standard, well-specified operating system interface; much of this interface, called the LITTLE 'library' is written in LITTLE.

There remain several troublespots in the language. Problems remain in the handling of arithmetic precision, particularly for negative integers. The previous version provided large precision (up to 2047 bits), but did not support large negative integers, and did not include word-size arithmetic as a proper subset; a more fruitful approach seems to provide a parameterized 'portable integer' with size between 16 and 100 bits, but the details remain unclear. The handling of array elements, particularly in calling sequence, remains unusual. This version retains the peculiar value semantics since the design of the standard compiler makes the implementation of the more usual 'pointer' semantics a difficult task.

PREFACE

The Guide serves both as a language definition and reference manual. Although the LITTLE system includes a standard compiler, LITTLE is not a language 'defined' by its compiler. A programming system is a contract, and this Guide attempts to provide all needed fine print. However, language definition is no easy task; numerous errors and omissions undoubtedly remain. The style used in the Guide emphasizes use as a reference manual; the discipline of writing in this way has aided in the elimination of needless digressions and extraneous justifications. The organization of the Guide is based in large part on that used by Lecht (/1/) in his admirable book 'The Programmer's PL/I'.

In summary, LITTLE provides a congenial, effective environment for the construction of portable, efficient software. Portability remains an elusive goal, but can, with some thought, be achieved.

Acknowledgments

The LITTLE project represents the work of many people. Jacob Schwartz designed the first version of the language. The principal authors of the LITTLE system are Robert Abes, Edith Deak, Richard Kenner, David Shields, Aaron Stein and Thomas Stuart; their contributions are too numerous to detail. Henry Warren was the first major user and was most patient in accepting the weaknesses of the bootstrap compiler. Paul Schneck provided the first implementation for the IBM System/370. The members of the SETL project have made numerous comments and suggestions.

INTRODUCTION

The present document is a comprehensive introduction to the LITTLE programming language, and contains the specifications of the language.

Goals

The goal of the LITTLE project is to provide a means for creating portable and efficient software. Software is portable if it contains no implicit assumptions about the environment in which it is to be executed. Environmental dependencies are unavoidable for all but the simplest programs, but if they are made to appear as explicit parameters of a program, then that program may be made portable. There are three basic classes of environmental dependencies: machine dependencies, system dependencies and compiler dependencies. Machine dependencies reflect the basic machine architecture - word size, character size, addressing modes, etc. System dependencies reflect the operating system conventions: file structure, input/output schemes, loader facilities, calling conventions, etc. Compiler dependencies reflect variations in compiler performance and features: the nature and quality of generated code, compiler options, diagnostic level, listing format, etc.

The basic tool needed to achieve program portability is a language which provides systematic mechanisms for making explicit the crucial environmental dependencies; ideally, the language should force the explicit statement of all such dependencies. The language must then be coupled with a processor which deals automatically, in a manner invisible to the user, with those dependencies which it can handle better than the user himself could. 'Better' in this case means more systematically and more efficiently.

Our tool for producing portable, efficient software consists of the following:

A programming language, LITTLE.
A standard compiler for LITTLE, written in LITTLE.
A library of procedures, written as much as possible in LITTLE, which provides a standard operating system interface.

As an example of explicit machine dependencies, the following special tokens designate three basic parameters of a target machine:

.WS.	size of machine word
.PS.	size of machine pointer (address)
.CS.	size of character.

The parameter .PS. reflects the possibility that a machine may have full-word and address arithmetics of differing precisions. For example, for the IBM System/370, .WS. has value 32 and .PS. has value 24. These parameters can be used in a LITTLE source program. On the other hand, storage allocation is handled by the compiler itself, and nothing is provided in the language to give the user any control over this process.

INTRODUCTION

System dependencies are mainly a reflection of the lack of standard specifications for operating systems features; for example, the notion of 'record' is so diffuse that it is not mentioned in the definition of the LITTLE input/output features. LITTLE deals with system dependencies mainly by providing a standard interface with the host operating system. Much of this interface is written in LITTLE. This body of code, called the LITTLE Library, is itself a fruitful product of our work on portability.

A program is efficient if it is presented at a level of detail such that its translation into assembly language is a routine, albeit time-consuming task. This requires that the compiler translate well the data structure representations, and produce good code. The use of a common compiler and library means that some constructs are not always realized as efficiently as with a machine-tailored compiler. This is the price of portability. However, the cost of alternate strategies is almost invariably higher. Furthermore, the language is designed with optimizaton techniques in mind. Several language features were chosen (or rejected) according to the applicability of program optimizaton to them.

Origins

The language was created by Jacob Schwartz in 1968; the initial description is contained in Cocke and Schwartz (/2/). In brief, LITTLE was first defined as a quite low-level language, similar to FORTRAN, which had bitstrings as its single data type, and which contained the minimal set of operators and statements needed to express the compiler for the language in LITTLE itself. The basic concepts are thus machine-independence, efficiency, and self-definition. These concepts are explained in detail below. Further goals include the desirable features of any programming language, such as readability, modularity, ease of use, and ease of debugging.

Machine independence is the fundamental goal of LITTLE. Methods of attaining this goal may be divided into several broad categories:

1. The 'complete' approach defines an abstract environment which is natural to users and which can be implemented. Examples of this approach include PL/I, SETL and SNOBOL. Although every programming language necessarily defines an environment of the sort just described, the 'complete approach' uses languages of a quite high level, and the compilers for these languages are typically written in other languages.

2. The 'abstract' machine approach defines an abstract machine which can be faithfully, and acceptably, modelled on the available hardware. For example, this approach is used in the STAGE2 programming system (/3/), and by the SIMPL implementation language for GRAAL (/4/).

3. The 'macro' approach expresses a program as a set of macro-calls. Implementation involves the realization of the macros. For example, this approach is used in the SIL implementation of

SNOBOL4 (/5/).

4. The ’unknown machine’ approach expresses programs as ’machine’ language for a machine whose detailed features are not known at the time of program construction.

The above categories are not exclusive; rather they indicate some of the various approaches taken to achieve machine independence. The approach used in LITTLE is quite similar to the ’abstract’ machine approach; however LITTLE makes minimal assumptions about the structure of its target machine, and LITTLE programs are written with machine features as explicit program parameters.

PRECIS

LITTLE is a low-level language for the production of machine independent software. Correct use of the language requires careful parameterization of machine-dependencies. To express these dependencies, and to simplify coding, LITTLE includes a simple macro processor; for example:

```
+* BUMP(I) = I = I + 1; **   /* INCREMENT I. */
+* YES = 1 **  +* NO = 0 **   $  AIDS READABILITY.
+* HAMAX = 787 **  $ DIMENSION OF HA.
+* ERRORLIMIT = 50 **  $ MAXIMUM ALLOWED ERRORS.
```

As just shown, LITTLE programs may contain comments, both in the delimited '/*...text...*/' style of PL/I, and the 'rest-of-line' form, which begins with '$' and includes the rest of the line.

The single data type is the bitstring. Bitstring variables are declared with the SIZE statement, which defines the length of the bitstring; one dimensional arrays are supported, and are declared with the DIMS statement; for example:

```
SIZE  LINE(CS);  DIMS LINE(80);  /* ARRAY OF CHARACTERS */
```

Bitstrings extend from right to left, counting from 1; for example, the bitstring '10' has bit 1 = 0, bit 2 = 1, the leftmost (most significant) bit is 1, and the rightmost (least significant) bit is 0.

LITTLE permits the use of a constant expression, i.e., an expression containing only constant operands, in most cases where a single constant may appear, such as 'SIZE BIG(3*WS+1)'.

LITTLE provides the standard arithmetic operators for bitstrings, viewed as the binary representation of integers: + - * / . LITTLE includes the usual comparison operators, in both their FORTRAN and PL/I form:

```
 =       ^=      <       >       <=      >=
.EQ.    .NE.    .LT.    .GT.    .LE.    .GE.      .
```

LITTLE also includes the standard bitstring primitives:

```
 &       !       ^
.AND.   .OR.    .NOT.   .EXOR.
```

Additional basic operators include

```
.FB. X   - index of leftmost nonzero bit in X
.NB. X   - number of nonzero bits in X
```

PRECIS

LITTLE provides extractors to access subparts of bitstrings; the basic form is

```
.E.  3, 5, W
```

which specifies that field in W which begins at bit 3, and extends 5 bits to the left. Fixed fields are often defined by macros; for example to access the left and right parts of a word, we might write

```
+* LEFT = .E. WS/2+1, WS/2, **
+* RIGHT = .E. 1, WS/2, **
X = LEFT A .EXOR. RIGHT B;
LEFT Y = 10;  .
```

LITTLE represents character strings as bitstrings which contain fixed fields defining the number of characters in the string, and the position of the first (leftmost) character. LITTLE provides character and substring extractors, which may be used on both the left and right side of an assignment statement; for example

```
.S. 1, 3, STR = 'AB1'; $ ASSIGN SUBSTRING.
X = .CH. 2, STR;  $ X IS NOW CHARACTER CODE FOR LETTER B.
```

LITTLE enumerates characters from left to right, starting from one, and includes string concatenation,'!!', and a search operator, of form 'X .IN. Y', which returns the index in character string Y of the first occurrence of character string X.

In summary, LITTLE has bitstring variables, possibly indexed, as the basic data type. LITTLE provides extractors to access parts of bitstrings, and includes the usual arithmetic and boolean operators.

LITTLE supports floating point (real) arithmetic. This is done by associating an arithmetic mode - integer or real - with each variable, array or function procedure. The arithmetic mode of an operation is generally integer unless both inputs are of real mode. Real variables are declared using the REAL declaration instead of the SIZE declaration used to declare variables with integer arithmetic mode. There are no implicit mode conversions, and no conversions on assignment. LITTLE supports the standard FORTRAN real operations and functions.

Statements are terminated by a semicolon, and may be written in a free form. LITTLE has no fixed column assignments for input text. Statements may be simple or compound. Compound statements begin the definition of a compound group; the body may contain other statements. The body is terminated by an END statement.

LITTLE includes the standard assignment statement, and the assignment target may be qualified by an extractor, to allow access to subparts of items; e.g.,

```
.F.  4, 12, X = Y/7;
```

PRECIS

LITTLE provides the usual IF, DO, WHILE, and UNTIL statements. Statements may be labeled. Labels may be subscripted. LITTLE includes the GO TO, in both its simple and indexed form.

```
        GO TO ERRORCASE; ...
        GO TO L(I) IN 1 TO 3;
    /L(1)/ ...
    /L(2)/ ...
    /L(3)/ ...
         /ERRORCASE/...            .
```

LITTLE uses a static namescoping scheme similar to FORTRAN. By default, variables are local to the procedure in which they are defined. Global variables may be defined, and are grouped together in named groups using the NAMESET statement. The ACCESS statement is used to name the NAMESETs which a particular procedure may use. The DATA statement specifies the initial values of variables.

Gross program structure is similar to that of FORTRAN; subprograms and functions may be defined with the SUBR and FNCT statements, respectively. Execution begins with the PROG statement group.

LITTLE includes input/output facilities for unformatted (binary) and formatted files. Formatted files may be external sequential files or internal character strings. Formatted IO may use fixed fields (edit mode) or free-form fields (list mode).

The following program fragment indicates the flavor of LITTLE. Appendix G contains a more comprehensive example.

```
     /*   S A M P L E   L I T T L E   P R O C E D U R E  */

$        MACRO SECTION - DEFINE MACHINE PARAMETERS, CODE SEQUENCES
    +*  SWAP(A,B) =  $ MACRO TO SWAP TWO ITEMS, A COMMON OPERATION
        SIZE  ZZZA(.WS.);  $ TEMPORARY FOR MACRO
        ZZZA = A;  A = B;  B = ZZZA;  **

    SUBR SORTER(A, N);
   $ THIS PROCEDURE SORTS THE ARRAY A OF N ITEMS USING A
   $ SORTING ALGORITHM, DUE TO J. SCHWARTZ, WHICH IS BASED
   $ ON THE ELEGANT HEAPSORT ALGORITHM.

    SIZE  A(.WS.); DIMS A(2);  $ ARRAY TO SORT.
    SIZE  I(.PS.);          $ DO LOOP INDEX
    SIZE  N(.PS.);          $ NUMBER OF ELEMENTS TO SORT.
    SIZE  M(.PS.);          $ CURRENT NODE BEING EXAMINED.
    SIZE  TOP(.PS.);        $ CURRENT TOP OF TREE DURING PHASE 2.
    SIZE  TARG(.PS.);       $ INDEX OF LARGEST CHILD.

    DO  I = 2 TO N;  $ MAKE INTO HEAP, I IS CURRENT PARENT
        M = I;
        WHILE  M > 1;  $ EXAMINE PARENTS IN TURN.
            IF  (A(M/2) >= A(M)) QUIT WHILE;  $ IF PARENT NO SMALLER,
            SWAP(A(M), A(M/2));  $ PROMOTE LARGE CHILD,
            M = M / 2;  $ MOVE TO GRANDPARENT.
            END WHILE;
        END DO I;
```

```
    DO  TOP = N TO 2 BY -1;  $ SORT SUBTREES IN TURN.
        SWAP(A(1), A(TOP));  $ EXTRACT LARGEST ELEMENT.
        M = 1;  $  FORCE REMAINING SUBTREE TO BE HEAP.
        WHILE  M*2 < TOP;  $ FOR ALL SUBTREES
            IF  (A(M*2) < A(M*2+1) ) & (M*2+1 < TOP)
                THEN  TARG = M*2+1;
                ELSE  TARG = M*2;  END IF;
            IF  A(M) < A(TARG)  THEN
                SWAP(A(M), A(TARG));  $ CHILD TOO BIG, EXCHANGE.
                ELSE  QUIT WHILE;  END IF;
            M = TARG;  $ MOVE TO SUBTREE OF LARGEST CHILD.
            END WHILE;
        END DO TOP;
    END SUBR SORTER;
```

TERMS AND NOTATION.

This section defines the terms and notation used in the remainder of this document.

Source text structure, directives and lines.

LITTLE source text consists of a sequence of lines. Each line contains at least 72 characters. The first 72 characters of each line are LITTLE text; remaining characters may be used for identification. Every compilation directive line has a blank as the first character of the line and a period as the second character. A rest-of-line comment begins with the dollar character.

Formation of names

There are two types of names in LITTLE: simple names and name constants. A simple name consists of an alphabetic character followed by zero or more alphameric characters. A name constant consists of an integer of one to three digits, followed by the letter N, followed by a value part. Let L be the value of the integer. If L is nonzero, the value part consists of the L characters immediately following the letter N. If L is zero, the first character after the letter N defines a delimiter, and the value part consists of the one or more characters which occur before the next following instance of the delimiter. For example, the following symbols each define the name LITTLE:

```
LITTLE    6NLITTLE    0N/LITTLE/
```

(Comment: Name constants are typically used to define a name which contains non-alphameric characters. Such nonstandard names are often used to define names which have a low probability of conflicting with simple names found in programs. For example, the procedures used to implement the LITTLE IO features have names which end in '$IO', as in '7NGETC$IO'.)

The Guide uses name to indicate that either a simple name or a name constant may be written. If only a simple name may be used, this restriction is explicitly noted.

Variables and arrays

LITTLE provides bitstring variables and one dimensional arrays, and requires that every variable be declared. The initial declaration is either a SIZE declaration or a REAL declaration; a subsequent DIMS declaration defines an array and gives the number of elements in the array. The DATA statement gives the initial values of variables. The phrase 'variable' indicates that either a simple variable or array element can be used in a construct; any construct which requires only simple variables, and does not permit array elements, is noted by using the phrase 'simple variable'.

TERMS AND NOTATION.

Counting conventions

The bits in a bitstring are enumerated from right to left. The least significant bit has index one and is the rightmost bit. The characters in a character string are counted from left to right. The first character has index one and is the leftmost character. The first element of an array has index one.

Arithmetic mode

There are two arithmetic modes: integer and real. Every variable, array and function procedure has an arithmetic mode, established by declarations. Arithmetic is in general done in the integer mode, but is done in real mode if both operands are of real mode, or if the operation definition specifies that an operand may be of real mode. For example, A+B is integer add unless A and B both have real mode, while SQRT(X) is valid only if X is of real mode.

There are no implicit conversions of mode within expressions or as part of assignment process. For example, if I is of integer mode and R is of real mode, the assignment 'I=R' just copies the value of R to I without conversion. The standard functions IFIX and FLOAT are provided, and must be explicitly written, to effect mode conversion; for example, 'I = IFIX(R)'.

Global and local variables

Variables in LITTLE are local or global. All variables must be declared within a procedure body. A variable is global if it is declared within the body of a NAMESET statement group, and is said to be a member of the NAMESET.

A procedure may reference global variables which are bound to a formal argument of the procedure or which are members of an accessible NAMESET. A NAMESET is accessible to a procedure if the procedure contains the NAMESET group defining the NAMESET, of if the procedure contains an ACCESS statement which includes the NAMESET name.

(Comment: The standard compiler assists in the use of global variables by providing two options, one to generate a NAMESET consisting of the otherwise 'local' variables declared in the first procedure compiled, another to grant each procedure access to all NAMESETs defined in the first procedure compiled. Both these options are enabled by default.)

TERMS AND NOTATION.

Files

Within a program a file is identified by an integer. The integer must be greater than zero and no greater than some implementation limit (typically ten). The FILE statement defines (connects) a file to an external medium; the TITLE clause of the FILE statement identifies the external medium, the ACCESS clause indicates the I/O features to be used. Initially, file one is preconnected as the standard input file, and file two is preconnected as the standard output file. The ACCESS option STRING permits the use of a character string variable as a single line file, so that a program can use the data conversion and editing features without performing IO on an external medium.

Array blocks

The input/output features permit the use of an 'array block' to indicate transmission of several elements of an array. The form of an array block is 'ARA(LO) to ARA(HI)' where ARA is an array name, and LO and HI index elements of ARA. HI must be greater than or equal to LO. The array block specifies transmission of the elements ARA(LO), ARA(LO+1),...,ARA(HI). The array block consisting of all the array elements may be specified by giving the array name alone: 'ARA' without an index corresponds to the array block 'ARA(1) TO ARA(D)', where D is the dimension of ARA. If HI=(LO-1), no element is transmitted. This is a null slice.

Compound groups

A compound group is a sequence of statements, called a statement group, which serve a given purpose. The first statement in the group defines the purpose of the group. An END statement terminates the group. The keyword END in an END statement may be followed by several tokens; if given, they must match the tokens in the statement which begins the group. The compound statements are as follows:

DO FNCT IF NAMESET PROG SUBR UNTIL WHILE.

The FNCT, PROG and SUBR statement groups define a procedure group, or procedure. The DO, UNTIL and WHILE statement groups define iteration groups.

TERMS AND NOTATION.

Compound groups may be nested and may include CONT and QUIT statements which refer to the group. The group referred to is determined by a list of tokens in the statement; if this list is empty, the group referred to is the innermost group of the desired type; otherwise the group is the innermost group whose initial statement begins with the same tokens as are in the token list.

Procedures

A procedure is a named sequence of statements. In LITTLE, a procedure is a compound group which begins with a FNCT, PROG or SUBR statement. A CALL statement directs the execution of a SUBR group. The appearance of a FNCT group name within an expression directs the execution of a FNCT group. Program execution begins with the program group defined by a PROG statement. A procedure group may not contain another procedure group, nor may a compound statement contain a procedure group.

Iteration groups, iterators

An iteration group begins with a DO, UNTIL or WHILE statement. The body of an iteration group is executed a varying number of times according to the value of an iteration condition. Various statements in the body may direct whether to continue or terminate the iteration. 'To terminate an iteration' is to continue processing with the statement which follows the END statement which ends the iteration group. 'To continue an iteration' is to continue processing with the first statement in the group body.

Statement labels

A statement label identifies a statement. A statement label consists of a name optionally followed by an integer constant enclosed in parentheses. A statement label definition consists of a slash character followed by a statement label, followed by a slash character, written before a statement. A statement may be prefixed with one or more statement label definitions. Statement labels are used in the simple and indexed GO TO statements to explicitly select the next statement to be processed.

TERMS AND NOTATION.

Instance symbols

Most of the constructs of the LITTLE language have parameters which represent the symbols, variables and constants needed to define an actual LITTLE program. The following symbols are used to indicate the form of the items which may occur in a given construct. An instance symbol consists of a letter followed by a digit. The letter gives the type of the symbol, the digit gives an instance number. The digit '9' is used only where a varying number of instances can be written, and then to indicate the last instance. The letters used are as follows:

A	array
C	constant, or expression in constants
E	expression
I	integer expression (arithmetic mode integer)
N	name
R	real expression (arithmetic mode real)
V	variable, or array element

Comments within the guide

The construct '(Comment: ...)' indicates an internal comment. Such comments are not part of the definition of LITTLE, but are included to improve readability and to direct the reader to related material.

CHARACTER SET

LITTLE uses the following characters:

1. Alphabetic characters:
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _

(The character '_' is called the break character.)

2. Numeric characters: 0 1 2 3 4 5 6 7 8 9

The alphabetic and numeric characters are referred to collectively as the alphanumeric (alphameric) characters.

3. Special characters:

Symbol	ASCII	Name
	32	blank
=	61	equal
+	44	plus
-	45	minus
*	42	times, asterisk
/	47	divide, slash
(	40	left parenthesis
)	41	right parenthesis
,	44	comma
.	46	period, point
;	59	semicolon
:	55	colon
$	36	dollar sign, comment character
^	94	not
&	38	and
!	33	or
<	60	less than
>	62	greater than
'	39	apostrophe, string delimiter

The ASCII code for break character is 95.
Implementations may support both upper and lower case letters.
If so, case is significant only within string constants.

COMMENTS

A LITTLE program may contain comments. Comments may occur between, or even within, program statements. Comments provide for the internal documentation of a program, and have no effect on the manner in which the program is executed. Comments are lexical tokens, and so may not occur within other tokens.

Comments may also occur, in certain cases, in the datasets processed by the LITTLE input/output features, as described in section 7.

Comments are of two types, as follows:

1. End-of-line comment, which begins with the character '$' and includes the remaining characters on the line.

2. Delimited comment, which begins with the characters '/*' (no intervening blanks), and consists of an arbitrary number of characters, possibly extending over several lines. The comment ends with the first occurrence of the characters '*/' (no intervening blanks).

Examples of comments are

```
I = 1;  $ PREPARE FOR SEARCH.

I = 1;  /* PREPARE FOR SEARCH. */

I = 1  /* PREPARE FOR SEARCH. */;

/*PREPARE*/ I /*FOR*/ = 1;  $ SEARCH
```

All of the lines just given are equivalent in that, after comments have been processed, each contains the single statement 'I=1;'.

MACRO PROCESSOR

LITTLE includes a simple macro processor. Macros with arguments are declared in the form

(1) +*MACRONAME(ARG1, ARG2,...,ARGk) = MACROBODY ** .

Macros with no arguments are declared in the form

(2) +* MACRONAME = MACROBODY ** .

In the above, MACRONAME is a name, ARG1,...,ARGk are names denoting the macro arguments, and MACROBODY is a sequence of zero or more lexical tokens.

After its definition, a macro may be invoked at any point by writing

(3) MACRONAME (SUB1, SUB2,...,SUBk)

for a macro with arguments, where each SUBn is a sequence of one more tokens.

A macro without arguments is invoked by just writing its name

(4) MACRONAME

The number of arguments in (1) and (3) must match, although null arguments are allowed. Each argument SUBk may be any sequence of tokens which is balanced with respect to parentheses and which contains no exposed commas, i.e., no commas not enclosed in parentheses.

Macro invocations are expanded by substituting SUBk for each occurrence of ARGk in the macro body, and issuing the resultant stream of tokens instead of the the tokens which invoked the macro. If this stream contains macro invocations, these inner invocations are expanded, and so on recursively.

Macro definitions may not explicitly contain other macro definitions; however, the macro processor does allow macros to be defined within other macros in an indirect fashion, as follows:

(5) Define macro Q3 by +*Q3(A,B,C) = A B C **.

(6) Define macro MACDEF by

```
+* MACDEF(TEXT) = Q3(+, *TEXT*, *) ** .
```

(7) Macros may then be defined within other macros by using

```
+* Outermacro = ... MACDEF(Name=Innerbody) ... ** .
```

MACRO PROCESSOR

The macro processor also supports a simple scheme for generating names and integers with values unique to a particular macro expansion. For example, such values are useful to generate statement labels within macros. The macro generation symbols have the form

```
ZZZA,ZZZB,..., ZZZZ, ZZZ_   (for names)
ZZYA,ZZYB,..., ZZYZ, ZZY_  (for integers)
```

Associated with each such name is a counter variable. When the name is first encountered during a macro expansion, the appropriate counter is incremented and the name is replaced by the name or integer so generated. Subsequent instances of the name in the macro body are replaced by the value generated on encountering the first instance. Generated names consist of the counter value appended to the name; for example, 'ZZZA' might be replaced by 'ZZZA01020'. If a counter variable is encountered when no macro is being expanded, it is replaced with the value last generated during a macro expansion, i.e. its current value.

Once a name is given macro status, it retains that status until it is 'dropped'. Macros are dropped by redefining the name as a macro with a null macro body. For example,

```
+* MACRONAME = **  .
```

The ZZYORG directive line resets selected ZZY symbols to have value zero. The directive line begins with a blank, followed by a period, followed by ZZYORG, followed by one or more blanks, followed by one of more alphabetic characters. The alphabetic characters give the last character of each ZZY symbol which is to be reset to zero.

(Comment: All macroprocessing is done at the lexical level, prior to parsing. Thus, macro definitions are 'global' in that they persist over procedure boundaries.)

The following example shows use of macros to define fields:

```
$   FIELDS OF LEXICAL SCANNER SYMBOL TABLE.
    +*  LEXTYP    = .E. 01, 04, **  $ LEXICAL TYPE.
    +*  LEXLEN    = .E. 05, 07, **  $ LEXICAL LENGTH.
    +*  HALENTYP  = .E. 01, 11, **  $ LENGTH AND TYPE FIELDS.
    +*  LITCOD    = .E. 12, 07, **  $ LITERAL CODE.
    +*  CAB       = .E. 19, 01, **  $ CONDITIONAL ASSEMBLY BIT.
    +*  NAMEPTR   = .E. 20, 13, **  $ NAMES INDEX.
    +*  MACORG    = .E. 33, 13, **  $ MACRO ORIGIN.
    +*  NUSES     = .E. 46, 02, **  $ NUMBER OF USES.
    +*  HALINK    = .E. 48, 13, **  $ LINK FOR HASH CHAIN.
```

Appendix D contains an informal introduction to the macro processor.

TEXT DEFINITION: CONDITIONAL ASSEMBLY, REMOTE TEXT

LITTLE provides conditional assembly to conditionally select the input lines to be processed, and an INCLUDE directive to request substitution of remotely defined text.

Conditional assembly

LITTLE supports a simple scheme for the conditional assembly of source text. Input lines with a blank in column one, a period in column two, one of the characters '+' or '-' or '.' in column three, and an alphabetic character in column four (which begins a simple name) are conditional assembly directives. Such lines take one of the forms

```
(1)      .+NAME   conditional assembly of rest of line
(2)      .-NAME   conditional negative assembly of rest of line
(3)      .+NAME.  conditional assembly of group
(4)      .-NAME.  conditional negative assembly of group
(5)      ..NAME   end of conditional group
```

For each name used in one of the above forms there is an associated conditional assembly bit, CAB. The CAB is initially zero. When form (1) is seen, the rest of the line is processed only if the CAB for the name is one; otherwise, the rest of the line is ignored. When form (2) is seen, the rest of the line is processed only if the CAB for the name is zero; otherwise the line is ignored.

Forms (3) and (4) are treated similarly, except that the scope of the conditional action is the next conditional assembly line referring to the same name. Form (5) is used to indicate the end of a conditional assembly group.

The CAB values for all names are initially zero. Values may be set by using conditional assembly lines with the name 'SET', which has a special interpretation. Such lines contain the conditional name SET followed by a list of names separated by commas. The CAB for the name is set to one in the case '.+SET', or is set to zero in the case '.-SET'. For example, the line

```
         .+SET  HATRACE  $ ENABLE TRACING OPTION.
```

enables the conditional name HATRACE.

Conditional directives can be nested; for example

```
    .+S66.
    .+SET EXTIME   $ Display execution time
    .+SET WSM3     $ Word size is multiple of three.
    ..S66
```

Appendix G contains an example of the use of conditional assembly.

Inclusion of remote text

The text inclusion feature permits the collection of text lines into named groups, called MEMBERs, and subsequent insertion of MEMBERs into a text file by use of the INCLUDE directive. The inclusion feature is typically used for text fragments shared by several programs; examples of such text fragments include macro definitions for codes, field structures and procedure definitions. The inclusion feature can also be used if the same text fragment occurs several times in a program, although the macro processor is more commonly used for this function.

An INCLUDE directive is a line which begins with a blank, followed by a period, followed by an equal sign, followed by INCLUDE, followed by one or more blanks, followed by a member specification. A member specification begins with the first nonblank character after column twelve and ends with the next nonblank character which is followed by a blank character. The member specification defines a member name M according to the following rules:

1. Remove the first character if it is an apostrophe.
2. Remove the first character if it is a left parenthesis.
3. Remove the last character if it is an apostrophe.
4. Remove the last character if it is a right parenthesis.

The above reduction rules permit several ways of specifying a MEMBER name; for example, the following each refer to LTL:

```
LTL  'LTL'  '(LTL)'  (LTL)    'LTL)
```

In effect, the lines of the named MEMBER replace the INCLUDE directive.

The format of a sequential inclusion text file is as follows:

1. A member definition line of the the form ' .=MEMBER M' begins the definition of member M.

2. The member consists of all following lines up to, but not including the next member definition line, or the end of the file, whichever occurs first.

3. A member may have no lines.

TEXT DEFINITION: CONDITIONAL ASSEMBLY, REMOTE TEXT

(Comment: The standard sequential form defines a machine independent representation of member definitions which can be used for program interchange. However, a LITTLE implementation may represent text libraries in a system dependent manner, particularly if direct access input/output is available.)

(Comment: The standard compiler option 'IMEM=M' directs the inclusion of member M before the first line of the input.)

(Comment: The standard compiler accepts MEMBER directives in the compiler input file to permit the trial compilation of text libraries. This also permits the writing of program text in a form which suggests the form a text library would take, without requiring the construction of the text library in order to compile the text. Input lines containing MEMBER directives are skipped.)

DATA TYPES AND CONSTANTS

LITTLE provides the bitstring as the basic data type and supports the use of bitstrings to represent integers, floating point numbers, character codes and character strings. The notion of data type in LITTLE is less rigid than that found in most programming languages, and is closer in spirit to the assembly language level. LITTLE exposes the bitstring representation of data type values, and does not include any implicit conversions from one data type to another.

Bitstrings

The bitstring is the basic data type of LITTLE. A bitstring is a sequence of binary digits, or bits. The length of a bitstring is its size. LITTLE enumerates bitstrings from right to left, starting from one. For example, in the bitstring ’10’, the size is two, the rightmost bit is zero, the leftmost bit is one, the first bit is zero and the second bit is one.

A byte constant specifies the bits in a bit string. The constant begins with a single digit which specifies the byte width. The byte width must be 1, 2, 3 or 4. The byte width is followed by the letter B and then by a string of characters (the value part) delimited by apostrophes. The byte width gives the number of bits defined by each nonblank character in the value part. Blanks may occur within the value part; if present, they do not affect the value. For example, the following bitstring constants have the same value: 1B’1101’ 2B’31’ 3B’15’ 4B’D’

Byte digits are interpreted according to the byte width, as shown in the following table:

SYMBOL	4	3	2	1 (BYTE WIDTH)
0	0000	000	00	0
1	0001	001	01	1
2	0010	010	10	NV
3	0011	011	11	NV
4	0100	100	NV	NV
5	0101	101	NV	NV
6	0110	110	NV	NV
7	0111	111	NV	NV
8	1000	NV	NV	NV
9	1001	NV	NV	NV
A	1010	NV	NV	NV
B	1011	NV	NV	NV
C	1100	NV	NV	NV
D	1101	NV	NV	NV
E	1110	NV	NV	NV
F	1111	NV	NV	NV

The entry NV in the table indicates that the symbol is not valid in byte constants of the corresponding byte width. A byte constant is said to be ’binary’, ’octal’ or ’hexadecimal’ according as the byte width is 1, 3 or 4, respectively.

DATA TYPES AND CONSTANTS

Integers

LITTLE internally represents nonnegative decimal integers using the standard base two bitstring representation. For example, the bitstring ’100’ corresponds to the integer four, the integer fifteen corresponds to the bitstring ’1111’.

LITTLE bitstrings and nonnegative integers may be viewed as the rightmost part of an arbitrarily long representation. The assignment of a ’short’ value to a ’longer’ value implies that the leftmost bits of the result not defined by the assignment source are set to zero.

LITTLE permits the use of signed integers, although all such integers have a fixed size, which is the machine word size (.WS.), and the leftmost bit of a negative value is always one. The representation of negative integers is processor dependent, typically either one’s or two’s complement.

A decimal integer constant consists of one or more digits, optionally preceded by a sign character. One or more blanks may occur between two digits; such blanks do not affect the value; for example:

```
100,   10,  -123,     100 456 789
```

Reals

A floating point number is a processor-dependent approximation to a real number. Real quantities have an implementation-defined size. LITTLE provides explicit conversion operators, as well as several mathematical functions, such as SQRT for ’square root’.

A simple real constant consists of an optional sign, an integer part, a decimal point represented by ’.’, and a fractional part. The integer part and the fractional part consist of digits. Either part may be omitted, but at least one of the parts must be given. A real exponent consists of the letter E followed by an optionally signed integer, and represents a power of ten. A real constant is either a simple real constant, a simple real constant followed by a real exponent, or an integer constant followed by a real exponent. A real constant may contain one or more blanks between two digits; for example ’3.1416’ and ’3.1 4 16’ define the same value.; for example:

```
3.1416   .31416E+01  31.416E-1  31416E-04   3E0
```

(Comment: It is good practice to write the decimal point, and to write a digit before and after the decimal point. For example, write 43.0, 0.1 and 3.0E-02 instead of writing 43., .1 and 3E-02.)

Logicals (Booleans)

LITTLE does not include a separate logical (boolean) data type, but follows the convention that ’nonzero’ is ’true’ and zero is ’false’.

The standard form for 'true' is one, the standard form for 'false' is zero. The standard comparison operators, such as '>' always return as result a bitstring of length one. LITTLE provides the bitstring operations of And, Not, Inclusive Or and Exclusive Or. These operators also serve as logical operators using the standard form operands of size one.

(Comment: A standard coding convention is to use macros YES and NO defined by +* YES = 1 ** +* NO = 0 ** to clarify use of bitstrings for logical values.)

(Comment: On a machine which uses one's complement arithmetic, the quantity '-0' is considered to be nonzero, or 'true', since it has at least one nonzero bit.)

Character codes and character strings

LITTLE provides both character string constants and character code constants. Character string constants are instances of the most portable form of character strings, character code constants support the manipulation of the codes of individual characters as internal integers. For example, the character string constant '0' specifies the string of length one containing the single character corresponding to the digit zero, while the character code constant 1R0 specifies the internal integer code of the character for the digit zero.

A 'character code set' of width W with N symbols is the association of N distinct symbols with distinct bitstrings each of size W. The character code set is 'complete' if N is the W-th power of two, so that a symbol is associated with each possible bitstring. Each symbol is either a 'graphic symbol' or a 'control symbol'. The LITTLE language uses 56 distinct graphic symbols. The term 'character' is used in the guide to refer to one of these graphic symbols. LITTLE requires no control symbols; this Guide does not define the results of their use in character strings. The Guide also does not define the use of characters other than those used by the LITTLE language.

Since LITTLE uses 56 characters, each LITTLE implementation requires a character set with a width of at least six bits. The width of the environment character set is an essential parameter of a LITTLE program; this width is called the 'character size' and is written '.CS.'.

A character string is a 'sequence of characters'. In LITTLE a character string is a bitstring with three parts. The 'length part' gives the number of characters in the string. The 'origin part' gives the position of the character codes within the bitstring. The 'value part' contains, in order, the bitstring codes of the graphic symbols of the character string. The bitstring representation of a character string provides most of the properties of 'varying length character strings' as this term is commonly used. Appendix E contains a more detailed explanation of the use of this representation.

The unique character string which has length zero and contains no characters is the 'null' character string. The LITTLE operations on

character strings permit the use of the null string so that, for example, the result of concatenating a character string S to the null string is just the string S.

Character code constants

A character code constant defines a bitstring as a function of the graphic symbols. A character code constant consists of a length given by an unsigned integer, followed immediately by the letter R, followed by a value part.

If the length, L, is nonzero, then the value part consists of the L characters immediately following the letter R. If the length is zero, the first character following R is taken as a delimiter, and the value consists of following characters up to the next instance of the delimiter. Each character in the 'value part' of the constant defines a bitstring of length .CS. which has as value the bitstring code for the graphic character.

The size of a character code constant is the product of the character size, .CS., and the number of characters in the constant. Character code constants are stored right-justified with zero fill.

Examples:

```
        1RX   6RLITTLE       0R/DELIMITED CASE/      .
```

Character string constants

Character string constants consist of a sequence of zero or more characters enclosed in apostrophes. Such strings may contain internal apostrophes; if so, two apostrophes must be written to indicate each apostrophe in the string. For example, the character string containing the letter A, an apostrophe and the letter B is written

```
            'A''B' .
```

The string '' denotes the null character string.

Character string constants may also be given as Q-type constants. This form consists of a length part, followed by the letter Q, followed by a value part. The length part consists of one to three numeric characters with no intervening blanks; let L be the value of the length part. If L is nonzero, the value part consists of the L characters following the letter Q. Otherwise, the first character following the letter Q defines a delimiter, and the value part consists of the zero or more characters which occur before the next instance of the delimiter. For example, the following are equivalent:

```
        'LITTLE'  6QLITTLE  0Q/LITTLE/
```

Q constants permit the definition of character string constants which contain apostrophes without the need to double the internal

apostrophes. For example, both

'A''B' and 3QA'B

define a constant character string consisting of the letter A, followed by the apostrophe character, followed by the letter B.

EXPRESSIONS

Expressions are constructed using constants, names and operators in the usual manner. The rules for expression formation permit the use of parentheses to explicitly delimit operands and the use of precedence levels to simplify the writing of expressions. For example, the operator * has higher precedence than the operator +, so that A*B+C is taken to mean (A*B)+C). The following table summarizes the standard operators of LITTLE and gives the operator precedence levels.

Every expression has an arithmetic mode which is either integer or real. The phrase 'integer expression' denotes an expression which must have arithmetic mode integer, the phrase 'real expression' denotes an expression which must have arithmetic mode real.

Precedence	Symbol Synonyms	Function
7	.E. E1,E2,E3	Subfield of E3 with length E2 beginning at bit E1
	.F. E1,E2,E3	Subfield of E3 with length E2 beginning at bit E1 (may not cross word boundaries.)
	.S.E1,E2,E3	Substring of character string E3 with length E2 beginning at position E1
	.CH. E1,E2	Character E1 of character string E2
	.LEN. E1	Length of character string
	+E1	Unary sign prefix: +E1 same as E1
	-E1	Unary sign prefix: -E1 same as (0-E1)
	.NB. E1	Number of nonzero bits in E1
	.FB. E1	Position of leftmost nonzero bit in E1.
	.SDS. E1	Size of character string of E1 characters.
6	E1 * E2	Multiply
	E1 / E2	Divide
	E1 .IN. E2	Index in character string E2 of first occurrence of character string E1
5	E1 + E2	Add
	E1 - E2	Subtract

EXPRESSIONS

4	E1 = E2	.EQ.	Equality
	E1 ^= E2	.NE.	Inequality
	E1 > E2	.GT.	Greater than
	E1 >= E2	.GE.	Greater than or equal
	E1 < E2	.LT.	Less than
	E1 <= E2	.LE.	Less than or equal
	E1 .SEQ. E2		Character string equality
	E1 .SNE. E2		Character string inequality
3	.NOT. E1	.N. ^	Bit by bit inverse
2	E1 & E2	.AND. .A.	Bitstring logical product
1	E1 ! E2	.OR.	Bitstring inclusive or
	E1.EXOR.E2	.EX.	Bitstring exclusive or
	E1 !! E2	.CC.	Character string concatenation
	C1 .PAD. C2		Pad character string C1 to length C2

The operands of an expression may be evaluated in any order, and only as many operands as are required to determine the expression result need be evaluated.

EXTRACTION OPERATORS

The following operators extract part of a bitstring. There are also corresponding forms of the assignment statement to assign new values to part of a bitstring.

General extractor

Purpose: To extract part of a bitstring.

Form: .E. I1, I2, E1

Rules:

1. I2 must be greater than or equal to zero.

2. If I2 is zero, the result is zero.

3. I1 must be greater than zero, and (I1+I2-1) must be less than or equal to the size of E1.

4. The I-th bit of the result is the (I1+I-1)-th bit of E1.

Partword extractor

Purpose: To extract a bitstring from a machine word.

Form: .F. I1, I2, E1

Rules:

1. I2 must be greater than or equal to zero.

2. If I2 is zero, the result is zero.

3. I1 must be greater than zero, and I2 must be less than or equal to the machine word size WS. ((I1-1)/WS) must be equal to ((I1+I2-1)/WS).

4. The I-th bit of the result is bit (I1+I-1) of E1.

EXTRACTION OPERATORS

Character substring extractor

Purpose: To extract a substring of a character string.

Form: .S. I1, I2, E1

Rules:

1. I2 must be greater than or equal to zero.

2. If I2 is zero, the result is the null character string.

3. E1 must be a character string. Let LE1 be the length in characters of E1.

4. I1 must be greater than zero, and (I1+I2) must be less than or equal to (LE1+1).

5. The result is a character string of I2 characters. The I-th character of the result is the (I1+I-1)-th character of E1.

Character code extractor

Purpose: To extract a character from a character string.

Form: .CH. I1, E1

Rules:

1. E1 must be a character string. Let LE1 be the length in characters of E1.

2. I1 must be greater than zero, and I1 must be less than or equal to LE1.

3. The result is the I1-th character of E1.

EXTRACTION OPERATORS

Character string length operator

Purpose: To determine the current length of a character string.

Form: .LEN. E1

Rules:

1. E1 must be a character string.

2. The result is the current length in characters of E1.

Examples:

```
.LEN. ''     is  0
.LEN. 'LTL'  is  3
```

UNARY OPERATORS

Unary minus operator

The expression '-E1' is equivalent to the subtraction of E1 from zero.

Unary plus operator

The expression '+E1' is same as (E1).

Bit Inversion operator

Purpose: To invert a bit string.

Form: ^ E1

Rules:

1. Determine bit I of the result as follows:
    1. If bit I of E1 is zero, bit I of the result is one.
    2. If bit I of E1 is one, bit I of the result is zero.

2. '^E1' may be written '.NOT. E1' or '.N. E1'.

Examples:

```
.NOT. 1B'10'   is  1B'01'
.NOT. 1B'0'    is  1B'1'
```

First bit operator

Purpose: To determine index of leftmost nonzero bit.

Form: .FB. E1

Rules:

1. Determine the result as follows:
    1. If E1 is zero, the result is zero.
    2. If E1 is not zero, the result is the largest integer I such that bit I of E1 is one.

Examples:

```
.FB. 1B'0'       is  0
.FB. 1B'01'      is  1
.FB. 1B'01001'   is  4
```

UNARY OPERATORS

Number of bits operator

Purpose: To determine the number of nonzero bits.

Form: .NB. E1

Rules:

1. The result is the number of bits in E1 which are one.

Examples:

```
.NB. 1B'0'      is  0
.NB. 1B'0101'   is  2
.NB. 1B'1000'   is  1
```

Character string size operator

Purpose: To determine the number of bits needed for a character string.

Form: .SDS. I1

Rules:

1. Assert that I1 is greater than or equal to zero.

2. The result is the size of a character string which may contain at most I1 characters. The result is always a multiple of the machine word size.

The standard LITTLE compiler assumes that the word size WS is a multiple of the character size CS, so that

```
.SDS. N  = WS * ((N + (.SL.+.SO.)/CS + CPW -1) / CPW)
```

where .SO. and .SL. are the symbols which denote the length in bits of the string origin and string length fields respectively. Assuming WS is 32, CS is 8, .SL. is 8 and .SO. is 16, .SDS. evaluates as follows:

```
.SDS. 0       is  32
.SDS. 1       is  32
.SDS. 2       is  64
.SDS. 5       is  64
.SDS. 8       is  96
.SDS. 80      is  672
```

Appendix E describes the representation of character strings in more detail.

BINARY OPERATORS

Arithmetic operators

LITTLE includes the following standard arithmetic operators:

1. Add, written 'E1+E2'.
2. Subtract, written 'E1-E2'.
3. Multiply, written 'E1*E2'.
4. Divide, written 'E1/E2'.

The result has arithmetic mode real only if both operands are of arithmetic mode real, otherwise the result has arithmetic mode integer.

Comparison operators

LITTLE includes the following standard comparison operators:

1. Equal to, written 'E1=E2' or 'E1.EQ.E2'.
2. Not equal to, written 'E1^=E2' or 'E1.NE.E2'.
3. Greater than, written 'E1>E2' or 'E1.GT.E2'.
4. Greater than or equal to, written 'E1>=E2' or 'E1.GE.E2'.
5. Less than, written 'E1<E2' or 'E1.LT.E2'.
6. Less than or equal to, written 'E1<=E2' or 'E1.LE.E2'.

The result is always zero or one.

The operands must have the same arithmetic mode.

In operators such as '<=' where two symbols are used to indicate the operator, the symbols are normally written with no intervening spaces; however, intervening spaces are permitted.

Character string comparison operators

Purpose: To compare two character strings for equality (inequality)

Form: E1 .SEQ. E2 (E1 .SNE. E2)

Rules:

1. Assert that E1 and E2 are character strings.
2. The result is one (zero) only if E1 and E2 have the same length and contain the same characters. Otherwise, the result is zero

(one).

Examples:

```
'' .SEQ. 'ABC'    is  0
'AB' .SEQ. 'AB'   is  1
'AB' .SNE. 'AC'   is  1
```

Character string concatenation operation

Purpose: To concatenate two character strings into a single string.

Form: E1 !! E2

Rules:

1. Assert that E1 and E2 are character strings.

2. If either input is the null character string, the result is the other input.

3. Otherwise, let L1 be the length in characters of E1, and let L2 be the length in characters of E2. The result is a character string of length (L1+L2). The first L1 characters are the characters of E1; the remaining L2 characters are the characters of E2.

4. 'E1!!E2' may also be written 'E1.CC.E2'.

Examples:

```
'' !! 'ABC'       is  'ABC'
'AB' !! ''        is  'AB'
'ABC' !! 'LTL'    is  'ABCLTL'
```

Character string instance operator

Purpose: To find an instance of one character string within another

Form: E1 .IN. E2

Rules:

1. E1 and E2 must be character strings.

2. If either E1 or E2 is the null character string, the result is zero.

3. Otherwise, let L1 be the length in characters of E1, L2 the length in characters of E2.

4. The result is zero unless string E2 contains an instance of E1, in which case the result is the index in E1 of the start of the first such instance.

BINARY OPERATORS

Examples:

```
'' .IN. 'ABC'        is  0
'AB' .IN. ''         is  0
'BC' .IN. 'ABCD'     is  2
'BC' .IN. 'ABCDBC'   is  2
```

Character string padding operator

Purpose: To pad a character string constant to a given length.

Form: C1 .PAD. C2

Rules:

1. C1 must be a character constant and C2 must be an integer constant greater than or equal to zero. Let L1 be the length in characters of C1.

2. The result is a character string constant of length C2. If C2 is less than or equal to L1, the I-th character of the result is the I-th character of C1. If C2 is greater than L1, the first C2 characters of the result are the characters of C1, the remaining (C2-L1) characters of the result are blank.

Examples:

```
'' .PAD. 0                    is  ''
'' .PAD. 2                    is  '  '
'ABC' .PAD. 6                 is  'ABC   '
'ABC' .PAD. 2                 is  'AB'
'ABC' .PAD. 0                 is  ''
'ABC'.PAD.6 !! 'XY'.PAD.10    is  'ABC   XY  '
```

(Comment: The .PAD. operator requires constant operands and is evaluated at compilation time. The .PAD. operator simplifies the writing of character string constants which end with several blanks.)

BINARY OPERATORS

Binary bitstring operators

LITTLE includes the following standard binary bitstring operators:

1. And, written 'E1&E2', or 'E1.AND.E2' or 'E1.A.E2'.

2. Inclusive Or, written 'E1!E2', or 'E1.OR.E2'.

3. Exclusive Or, written 'E1.EXOR.E2' or 'E1.EX.E2'.

Examples:

```
1B'1100' .AND.  1B'1010'   is   1B'1000'
1B'1100' .EXOR. 1B'1010'   is   1B'0110'
1B'1100' .OR.   1B'1010'   is   1B'1110'
```

STANDARD MATHEMATICAL FUNCTIONS.

LITTLE includes the following standard mathematical functions. Names I1 and I2 represent integer arguments. Names R1 and R2 represent real arguments. The function value is of arithmetic mode integer if the first character of the function name is I or M; otherwise the function value is of arithmetic mode real.

ABS(R1)	Real absolute value.
AINT(R1)	Real to integer truncation. If the absolute value of R1 is less than one the result is zero; otherwise the result is the sign of R1 times the largest integer whose absolute value is not greater than the absolute value of R1.
ALOG(R1)	Natural logarithm of R1 Assert that R1 is greater than zero.
ALOG10(R1)	Common (base ten) logarithm of R1. Assert that R1 is greater than zero.
AMOD(R1, R2)	Remainder: R1 - R2 * FLOAT(INT(R1/R2)). Assert that R2 is nonzero.
ATAN(R1)	Arctangent of R1 radians. Result RV in range -PI/2.0<=RV<=PI/2.0
ATAN2(R1, R2)	Arctangent of R1/R2 radians. Result RV in range -PI<RV<=PI.
COS(R1)	Cosine of R1 radians.
DIM(R1, R2)	Positive difference. If R1 is greater than R2, the result is (R1-R2); otherwise the result is zero.
EXP(R1)	E to the power R1
FLOAT(I1)	Integer to real conversion.
IABS(I1)	Integer absolute value.
IDIM(I1, I2)	Integer positive difference. If I1 is greater than I2, the result is (I1-I2), otherwise the result is zero.

STANDARD MATHEMATICAL FUNCTIONS.

IFIX(R1)	Real to integer conversion. If the absolute value of R1 is less than one, the result is zero. Otherwise the result is the sign of R1 times the largest integer whose magnitude does not exceed the absolute value of R1.
INT(R1)	Same as IFIX(R1)
ISIGN(I1, I2)	Sign of I2 times absolute value of I1. If I1 is zero, result is zero. I2 must not be zero.
MOD(I1, I2)	Remainder: I1 - I2 * (I1/I2).
SIGN(R1, R2)	Sign of R2 times absolute value of R1. If R1 is zero, result is zero. R2 must be not zero.
SIN(R1)	Sine of R1 radians.
SQRT(R1)	Square root of R1. R1 must be greater than or equal to zero.
TANH(R1)	Hyperbolic tangent of R1.

SIZING RULES

This section gives the rules which determine the size in bits of the result of an operation. The following general rules apply:

1. The size of a comparison is always one.
2. The size of standard arithmetic functions such as EXP or LOG is determined by the type of the result, and is always either the size of a signed integer or the size of a real.

In the following table, SZ(X) denotes the size of X, MIN and MAX denote the minimum and maximum, respectively.

The following table summarizes remaining size rules:

Operation	Result Size
.E. I1, I2, V1	If I2 is a constant, size is I2. Otherwise, size is SZ(V1).
.F. I1, I2, V1	If I2 is a constant, size is I2. Otherwise, size is MIN(.WS.,SZ(V1)).
.S. I1, I2, V1	If I2 is constant, size is (.SDS. I2). Otherwise, size is SZ(V1).
.CH. I1, V1	.CS.
.LEN. E1	.SL.
^ E1	SZ(E1)
.FB. E1	.PS.
.NB. E1	.PS.
.SDS. E1	.PS.
E1 * E2	MAX(SZ(E1), SZ(E2))
E1 / E2	SZ(E1)
E1 .IN. E2	.PS.
E1 + E2	MAX(SZ(E1), SZ(E2))
E1 - E2	MAX(SZ(E1), SZ(E2))
.NOT. E1	SZ(E1)
E1 & E2	MAX(SZ(E1), SZ(E2))
E1 ! E2	MAX(SZ(E1), SZ(E2))
E1 .EXOR. E2	MAX(SZ(E1), SZ(E2))
E1 !! E2	.SDS.(L1+L2) where L1 is largest integer such that (.SDS.L1 <= SZ(E1)) and L2 is the largest integer such that ((.SDS. L1) <= SZ(E2)).
C1 .PAD. C2	.SDS. C2

FORMAT AND DESCRIPTION OF LITTLE STATEMENTS

This section presents the statements of the LITTLE language, in alphabetical order, beginning with an index of the statement names and formats:

NAME	FORMAT
ACCESS	ACCESS N1, N2 ... N9;
ASSERT (MON)	ASSERT E1;
ASSIGNMENT	
1. SIMPLE	V1 = E1;
2. PARTWORD	.F. I1, I2, V1 = E1;
3. EXTENDED	.E. I1, I2, V1 = E1;
4. CHARACTER	.CH. I1, V1 = E1;
5. SUBSTRING	.S. I1, I2, V1 = E1;
6. CHARACTER STRING LENGTH	.LEN. V1 = I1;
CALL	CALL N1(E1, E2 ... E9);
CHECK (MON)	CHECK INDEX A1,A2 ... A9;; NOCHECK INDEX A1,A2 ... A9;
CONTINUE ITERATION	CONT;
DATA	DATA V1 = C1: V2 = C2 ... C9:
DIMENSION	DIMS N1(C1), N2(C2) ... N9(C9);
DO	
1. POSITIVE	DO V1 = I1 TO I2 BY I3; BLOCK; END DO;
2. NEGATIVE	DO V1 = I1 TO I2 BY - I3; BLOCK; END DO;
3. BY ONE	DO V1 = I1 TO I2; BLOCK; END DO;
END	END;
FILE	FILE I1 N1=E1,N2=E2,N3=E3;
FUNCTION	FNCT N1(N2, N3 ... N9);
GET	GET I1 IO_LIST;
GO TO	GO TO N1; GO TO N1(I1) IN C1 TO C2;
IF	
1. SIMPLE	IF E1 SIMPLESTATEMENT
2. COMPOUND	IF E1 THEN B1 ELSEIF B2 THEN ...END;
MONITOR (MON)	MONITOR OPTIONLIST;
NAMESET	NAMESET N1;

NULL STATEMENT	;
PROGRAM	PROG N1;
PUT	PUT I1 IO_LIST;
QUIT	QUIT;
READ	READ I1, V1, V2 ... V9;
REAL	REAL N1, N2 ... N9;
RETURN	RETURN;
REWIND	REWIND I1;
SIZE	SIZE N1(C1), N2(C2) ... N9(C9);
SUBROUTINE	SUBR N1(N2, N3 ... N9);
TRACE (MON)	TRACE OPTIONLIST; NOTRACE OPTIONLIST;
UNTIL	UNTIL E1; BLOCK; END UNTIL;
WHILE	WHILE E1; BLOCK; END UNTIL;
WRITE	WRITE I1, E1, E2 ... E9;

Statements marked (MON) are used to monitor program execution, and are discussed in Section 8, Monitor Facility.

ACCESS STATEMENT

Purpose: To permit references to variables which are members of a previously defined NAMESET;

Form: ACCESS N1, N2,...,N9;

Rules:

1. The ACCESS statement contains a list of variable names, separated by commas.

2. Each name must identify a previously defined NAMESET.

3. On encountering a reference to a variable not declared in the current procedure, search the list of accessible NAMESETs for a variable of the same name. If found, bind the name to the NAMESET member, so that subsequent references to the variable name are taken as references to the NAMESET member variable.

Examples:

```
PROG MAIN;
NAMESET SYMTABNS;
SIZE  SYMTABPTR(PS);  $ SYMBOL TABLE POINTER.
SIZE  SYMTAB(SYMTABSZ);  DIMS SYMTAB(SYMTABMAX);
END NAMESET;
...
END PROG MAIN;
...
SUBR ADDSYM(SI);
ACCESS SYMTABNS;
SYMTABPTR = SYMTABPTR + 1;
...
END SUBR ADDSYM;
```

ASSIGNMENT STATEMENT.

Purpose: To assign a new value to a variable or a subpart of a variable.

Forms:
```
V1 = E1;
.E. I1, I2, V1 = E1;
.F. I1, I2, V1 = E1;
.S. I1, I2, V1 = E1;
.CH. I1, V1 = E1;
.LEN. V1 = I1;
```

Rules:

1. The expression following the equal sign gives the source value of the assignment. The variable immediately preceding the equal sign is the target variable of the assignment. Execution of the assignment uses the value of the source expression to determine the value of some or perhaps all of the bits of the target variable.

2. The arithmetic mode of the source and target may differ; however, execution of the assignment statement includes no implicit conversions of arithmetic mode.

3. On execution of the simple assignment 'V1 = E1':
   1. Let LE1 be the length in bits of E1. Let LV1 be the length in bits of V1.
   2. If LE1 is greater than or equal to LV1, then for I from one to LV1, set the I-th of V1 to the I-th bit of E1.
   3. If LE1 is less than LV1, then for I from one to LE1, set the I-th bit of V1 to the I-th bit of E1. Then for I from (LE1+1) to LV1, set the I-th bit of V1 to zero.

4. On execution of the character string length assignment '.LEN. V1 = E1', set the length in characters of V1 to E1.

5. On execution of the character string assignment '.S. I1, I2, V1 = E1', which assigns the I2 characters of V1 starting at position I1 according to the value of E1:
   1. V1 must be a character string. Let LV1 be the length in characters of V1.
   2. If the value of I2 is zero, the assignment statement does not change the value of V1.
   3. E1 must be a character string. Let LE1 be the length in characters of E1.
   4. I1 must be greater than zero, and I2 must be greater than zero. (I1+I2-1) must be less than or equal to LE1.
   5. If LE1 is greater than or equal to I2, then for I from one to I2, set the (I1+I-1)-th character of V1 to the I-th character of E1.
   6. If LE1 is less than I2, then for I from one to LE1, set the (I1+I-1)-th character of V1 to the I-th character of E1. Then, for I from (LE1+1) to I2, set the (I1+I-1)-th character of V1 to be blank.

ASSIGNMENT STATEMENT.

6. On execution of the character assignment '.CH. I1, V1 = E1', which assigns the I1-th character of character string V1 to be the character whose internal code is E1:
   1. V1 must be a character string. Let LV1 be the length in characters of V1.
   2. I1 must be greater than zero, and I1 must be less than or equal to LV1.
   3. Set the I1-th character of V1 to E1.

7. On execution of the extract assignment '.E. I1, I2, V1 = E1', which assigns the I2 bits of V1 starting with bit I1 according to the value of E1:
   1. Assert that I2 is greater than or equal to zero. If I2 is zero, execution of the assignment statement terminates with no change to the value of V1.
   2. Let SV1 be the size of V1. Let SE1 be the size of E1.
   3. I1 must be greater than zero, and (I1+I2) must be less than or equal to (SV1+1).
   4. If SE1 is greater than or equal to I2 then, for I from one to I2, set the (I1+I-1)-th bit of V1 to the I-th bit of E1.
   5. If SE1 is less than I2 then, for I from one to SE1, set the (I1+I-1)-th bit of V1 to the I-th bit of E1. Then, for I from (SE1+1) to I2, set the (I1+I-1)-th bit of V1 to zero.

8. On execution of the field assignment '.F. I1, I2, V1 = E1', which assigns the I2 bits of V1 starting with bit I1 according to the value of E1 (subject to the restriction that all the assigned bits are in a single machine word):
   1. I2 must be greater than or equal to zero. If I2 is zero, execution of the assignment statement terminates without change to the value of V1.
   2. Let SV1 be the size V1. Let SE1 be the size of E1.
   3. I1 must be greater than zero, and (I1+I2-1) must be less than or equal to SV1.
   4. I2 must be less than or equal to the machine word size WS. ((I1-1)/WS) must equal ((I1+I1-2)/WS). (Comment: The field must be in a single machine word.)
   5. If SE1 is greater than or equal to I2 then, for I from one to I2, set the (I1+I-1)-th bit of V1 to the I-th bit of E1.
   6. If SE1 is less than I2 then, for I from one to SE1, set the (I1+I-1)-th bit of V1 to the I-th bit of E1. Then, for I from (SE1+1) to I2, set the I-th bit of V1 to zero.

CALL STATEMENT

Purpose: To initiate execution of a procedure, and to supply the parameters for that execution.

Form: CALL N1(E1, E2,...,E9);
CALL N1;

Rules:

1. The argument list is optional. If present, it consists of a list, enclosed in parentheses, of actual arguments, separated by commas.

2. An actual argument is a simple variable, array name or expression.

3. On execution of the CALL statement:
    1. Evaluate each actual argument which is an expression.
    2. Proceed to the first executable statement in the body of procedure N1.
    3. On execution of a RETURN statement or the END statement which terminates the procedure N1, continue execution with the statement following the CALL statement.

4. Assert that the number of arguments given in the CALL statement agrees with the number of formal arguments given in the procedure definition.

Examples:

```
CALL READLINE(INPUTFILE, NEXTLINE);
CALL EXIT;
```

CONTINUE STATEMENT

Purpose: To terminate the current execution of an iteration body, and possibly repeat execution of the iteration body, after testing, and perhaps modifying, the iteration control variable or expression.

Form: CONT;

Rules:

1. The CONT statement (CONT stands for continue) must occur within the body of an iteration group.

2. Zero or more tokens may follow the keyword CONT. If none are given, the CONT statement refers to the innermost iterator. If any are given, they must correspond to the tokens which begin an iteration containing the CONT statement, and the CONT statement refers to the innermost such iterator.

3. Execution of a CONT statement proceeds in the same way as execution of the END statement which terminates the iteration group.

Examples:

```
CONT WHILE I;
CONT DO;
CONT;

DO  I = 1 TO N;
    DO  J = 1 TO M;
        IF  (B(J)=0)  CONT DO I;
        A(I) = A(I) / B(J);
        END DO J;
    END DO I;
```

DATA STATEMENT

Purpose: To define the initial value of a variable.

Form: DATA V1 = C1: V2 = C2:...:V9 = C9;

Rules:

1. A DATA statement contains a list of initialization items, separated by colons.

2. An initialization item is either a variable initialization or an array initialization.

3. A variable initialization consists of a variable name followed by the equal symbol, followed by a constant. The constant gives the initial value of the variable when execution begins.

4. An array initialization consists of an array specification, followed by the equal symbol, followed by a list of array initial values.

5. The array specification specifies a starting index in an array. An array specification consisting of just an array name alone specifies a starting index of one. Otherwise, the array specification consists of an array name followed by a constant enclosed in parentheses; the constant gives the starting index.

6. An array initial value is either a constant or a constant followed by a repetition constant enclosed in parentheses.

7. The array initial values define the initial values of array elements, in order, beginning with the starting index. Repetition constants direct the initialization of successive array elements to the same value.

8. DATA initializations must occur within the procedure containining the declaration of the variable.

Examples:

```
 DATA I=1: J(3)=2;
$   THE FOLLOWING DATA STATEMENTS EACH INITIALIZE
$   A(1) TO A(10) TO BE 1,3,5,4,4,4,7,0,0,0.
DATA A = 1, 3, 5, 4, 4, 4, 7, 0, 0, 0;
DATA A = 1, 3, 5, 4(3), 7, 0(3);
DATA A(1) = 1, 3, 5: A(7) = 7: A(4) = 4(3): A(8) = 0(3);
```

DIMENSION STATEMENT

Purpose: To declare that an identifier is an array, and to indicate the number of elements in the array.

Form: DIMS N1(C1), N2(C2),...,N9(C9);

Rules:

1. A DIMS statement contains a list of dimension declarations, separated by commas.

2. A dimension declaration consists of a name N followed by a constant C which is enclosed in parentheses. C must be greater than zero.

3. The dimension declaration must occur after the initial SIZE or REAL declaration for the variable.

4. The dimension declaration declares that N is an array with C elements.

Examples:

```
DIMS LINE(72);
```

DO STATEMENT

Purpose: To mark the start of a DO group; to cause the statements of a DO group to be iterated based on the value of a control variable.

Form:
```
DO V1 = E1 TO E2 BY E3;   (Positive)
DO V1 = E1 TO E2;         (Positive by one)
DO V1 = E1 TO E2 BY - E3;  (Negative)
```

Rules:

1. The DO statement is an opener. The body of the DO group consists of all following statements up to and including the END statement which terminates the DO group.

2. The DO group is an iteration group.

3. The BY clause is optional. If E3 is not given, take E3 to be one.

4. V1 is the control variable and must be a simple integer variable. E1, E2 and E3 are integer expressions. E1 is the initial value, E2 is the final value, and E3 is the magnitude of the increment. E3 must be greater than zero. The loop is said to be increasing (decreasing) if the minus character does not (does) follow the keyword BY.

5. Execution proceeds as follows:
   1. Evaluate E1, E2 and E3; let LV1, LV2 and LV3 denote their respective values.
   2. If the loop is increasing (decreasing), then if LV1 is greater than (less than) LV2, execution proceeds to the statement following the END statement which terminates the DO group. Otherwise V1 is set to be LV1 and execution proceeds to the first statement of the DO group.

6. The expressions E1, E2 and E3 are evaluated only once, so that assignments within the loop body to variables occuring in these expressions do no affect the number of times the loop body is executed.

7. On execution of the END statement which terminates the DO group: If the iteration is increasing (decreasing), add (subtract) E3 to the iteration control variable V1. If the value of the control variable is greater than (less than) the value of E2, then terminate the iteration. Otherwise, continue the iteration.

Examples:

```
DO  I = 1 TO N;  A(I) = 0;  END DO;

DO  I = N TO 1 BY -1;
    IF  (A(I)=0)  CONT DO;
    A(I) = 10 / A(I);
    END DO;
```

END STATEMENT

Purpose: To end a compound statement group.

Form: END;

Rules:

1. The END statement terminates the statement group begun by the most recent compound statement.

2. An END statement terminates one statement group.

3. The keyword END may be followed by up to five tokens. If present, they must match the tokens which begin the compound statement.

4. Execution of an END statement depends on the type of the compound group, and is explained in the rules for the statement.

Examples:

```
IF  X > 0  THEN
    COUNT = 0;
ELSE
    COUNT = COUNT + 1;
    END IF X;
```

FILE STATEMENT

Purpose: To connect a file and give the file attributes to be used for subsequent input/output.

Form: FILE FID FATR1=EXPR1, FATR2=EXPR2,...,FATRn=EXPRn;

Rules:

1. Assert than FID is an integer greater than zero.

2. Execution of the FILE statement associates FID with an entity which can contain representations of bit strings.

3. The FILE statement contains a list of the attributes which apply for subsequent input/output operations on the file. Execution of the FILE statement either alters the current association, or terminates the current association and establishes a new association.

4. The file attributes FATRi are ACCESS, LINESIZE and TITLE.
    1. The ACCESS attribute must be given and must have as value one of the following symbols:
        1. GET: file contains formatted representations. Permit GET statements.
        2. PRINT: file contains formatted representations. Permit PUT statements. Permit use of PAGE control format.
        3. PUT: file contains formatted representations. Permit PUT statements.
        4. READ: file contains unformatted representations. Permit READ statements.
        5. STRING: file is line represented by a character string variable. Permit GET and PUT statements.
        6. WRITE: file is to contain unformatted representations. Permit WRITE statements.
        7. RELEASE: disconnect the file.
    2. If the LINESIZE attribute is given, then the value must be greater than or equal to zero.
        1. If LINESIZE given the file access must be GET, PRINT, PUT or STRING.
        2. If LINESIZE not given, take linesize to be zero.
        3. Assert that LINESIZE is greater than or equal to zero.
        4. If LINESIZE is greater than zero, it gives the number of characters in a line. If LINESIZE is zero, the length of a line is determined from the structure of the external file. ment: FILESTAT(FID, LINESIZE) gives actual line length.)
    3. The TITLE attribute names the file, as follows:
        1. If the ACCESS is STRING, the TITLE attribute specifies the name of a simple variable which has the structure of a character string. This variable contains the single line of the file.
        2. Otherwise, the value of TITLE is a character string which gives the operating system identification of the file. If the null string is specified, the file title is determined as function of file number, in processor-selected manner.

5. Execution of the FILE statement establishes a new association for the file unless the FILE statement is used to read a file that has

just been written. If the file statement contains only the ACCESS attribute, transition from writing to reading, with rewinding of the file, occurs if prior ACCESS was WRITE and new access is READ, or if prior ACCESS was PRINT or PUT, and new access is GET.

6. Two files are initially defined as execution begins.
   1. The standard input file has attributes

```
FILE 1  ACCESS=GET, LINESIZE = 0, TITLE = '' ;
```

      A GET statement which does not explicitly specify a file number implicitly specifies file one. File one is preconnected as the standard input file.
   2. The standard print file has attributes

```
FILE 2  ACCESS=PRINT, LINESIZE = 0, TITLE = '';
```

      A PUT statement which does not explicitly specify a file number implicitly specifies file two. File two is preconnected as the standard output file.

Examples:

```
+*  LISTING = 4 **  +* FS = 5 **  +* SCRATCH = 6 **
FILE  LISTING
      ACCESS = PRINT,
      LINESIZE = 120,
      TITLE = LISTINGNAME;


SIZE  SV(.SDS. 5);
FILE  FS
      ACCESS = STRING,
      TITLE = SV,
      LINESIZE = 5;


FILE  SCRATCH
      ACCESS = WRITE,
      TITLE = 'TAPE1040';
```

FUNCTION STATEMENT

Purpose: To name a function procedure, indicate its parameters, begin its definition.

Form: FNCT N1(N2, N3,...,N9);

Rules:

1. The function statement is an opener. The body of the function group consists of all following statements up to and including the END statement which terminates the function group.

2. The function statement contains the function name N1 followed by a list, enclosed in parentheses, of formal arguments, separated by commas. Each formal argument is a name.

3. The body of the function must contain a declaration of a simple variable of the same name as the function.

4. A function is invoked by writing its name, followed by a parenthesized list of actual arguments. The function procedure is then executed, and the last value assigned within the function to the variable of the same name as the function is used as the value of the function call.

5. A procedure P which invokes a function procedure N must contain a SIZE or REAL declaration for N to indicate the size and arithmetic mode of the function procedure value.

6. The function body must not contain any assignments to formal arguments.

7. On execution within a function procedure of a RETURN statement or of the END statement which terminates the function group, execution of the function procedure terminates. Execution proceeds with the use of the value of the function name variable within the expression which contains the function reference.

Examples:

```
$   FIND INDEX OF LAST NONBLANK IN STRING.
FNCT LASTNB(STR);
SIZE  LASTNB(PS);       $ FUNCTION VALUE.
SIZE  STR(.SDS. 80);    $ STRING.
SIZE  I(PS);            $ LOOP INDEX.
LASTNB = 0;
DO  I = (.LEN. STR) TO 1 BY -1;
    IF  (.CH. I, STR) ^= 1R  THEN
        LASTNB = I;
        QUIT DO;
        END IF;
    END DO;
END FNCT LASTNB;
```

GET STATEMENT

Purpose: To read data from a formatted file.

Form: GET FID Formlist;
GET Formlist;

Rules:

1. FID is an integer expression giving the file number. If FID is not given, take FID to be one.

2. Assert that file FID is connected with access GET or STRING.

3. Formlist is a list of control formats, input data items and data formats written according to the following rules:
   1. A comma precedes each control or data format.
   2. A colon precedes each data item.
   3. A data format or a data item follows a data item.

4. Transmit data according to the following rules:
   1. Transmit each data item according to the associated data format. The data format either immediately follows the data item or follows a list of data items.
   2. Transmit each element of an array block using the associated data format.

Examples:

```
FILE  FF
        TITLE = 'EXAMPLE',
        ACCESS = GET,
        LINESIZE = 100;
GET FF ,SKIP :X,I(5) :A(LO) TO A(HI),B(10,3);

GET :LINE,A(80) ,SKIP;  $ GET LINE FROM STANDARD INPUT.
GET :A:B:C,I(5)  $ READ A,B,C IN I(5) FORMAT.
```

GO TO STATEMENT (SIMPLE)

Purpose: To select the next statement to execute.

Form: GO TO SL;

Rules:

1. SL must be a statement label prefix for exactly one statement in the containing procedure.

2. On execution, proceed to the statement with statement label SL.

Examples:

```
GO TO READLINE;
...
/READLINE/  ...statement processed after GO TO ...
```

GO TO STATEMENT (INDEXED)

Purpose: To select the next statement to be processed according to the value of an integer selection expression.

Form: GO TO N1(E1) IN C1 TO C2;

Rules:

1. C1 and C2 must be integer constants such that C1 is greater than or equal to zero, C2 is greater than or equal to C1, and C2 is less than 1000.

2. The procedure containing the indexed GO TO must contain one statement label prefix N(E) for each integer E from C1 to C2.

3. Assert that E1 is greater than or equal to C1, and less than or equal to C2.

4. On execution, proceed to the statement with statement label prefix equal to E1.

Examples:

```
GO TO L(ECASE) IN 1 TO 4;
    /L(1)/ ...
    /L(3)/ ...
    /L(2)/  /L(4)/  ...
```

IF STATEMENT (SIMPLE)

Purpose: To conditionally determine whether a single, simple statement is to be executed.

Form: IF E1 Simplestatement

Rules:

1. Execution proceeds as follows:
    1. If E1 is nonzero, process the Simplestatement.
    2. If E1 is zero, proceed to the next statement.

2. Simplestatement is any statement except a compound statement, a simple IF statement or an END statement.

(Comment: It is suggested, but not required, that the control expression E1 be enclosed in parentheses.)

Examples

```
IF (X>0) CALL READER;
IF (FOUNDVAL ^= 0) RETURN;
```

IF STATEMENT (COMPOUND)

Purpose: To conditionally determine whether a group of statements is to be processed.

Form:
```
IF  E1  THEN  Block  END IF;
IF  E1  THEN  Block  ELSE  Block END IF;
IF  E1  THEN  Block
    ELSEIF E2  THEN  Block
    ...
    ELSEIF E9  THEN  Block
    ELSE Block END IF;
```

Rules:

1. The compound IF statement is an opener. The body of the IF group consists of all statements following the keyword THEN up to and including the END statement which terminates the IF statement.

2. A compound IF statement is distinguished from a simple IF statement by the occurrence of THEN immediately following the control expression which follows IF.

3. The body consists of an IF_THEN clause, followed by zero or more ELSEIF clauses, optionally followed by an ELSE clause.
   1. An IF_THEN clause consists of the keyword IF followed by a control expression, followed by the keyword THEN, followed by one or more statements.
   2. An ELSEIF clause consists of the keyword ELSEIF followed by a control expression, followed by the keyword THEN, followed by one or more statements.
   3. An ELSE clause consists of the keyword ELSE followed by one or more statements.
   4. A clause is terminated by the next clause or by the END statement which terminates the IF group.

4. On execution of a compound IF statement, perform the following actions for each clause:
   1. On execution of an IF_THEN clause, evaluate the control expression. If the value is zero, proceed to the next clause. Otherwise, proceed to the first statement following the keyword THEN.
   2. On execution of an ELSEIF clause, evaluate the control expression. If the value is zero, proceed to the next clause. Otherwise, proceed to the first statement following the keyword THEN.
   3. On execution of an ELSE clause, proceed to the statement following the keyword ELSE.
   4. After execution of the last statement in a clause, proceed to the statement following the END statement which terminates the IF group.

5. On execution of the END statement which terminates the IF group, proceed to the next following statement.

IF STATEMENT (COMPOUND)

Examples:

```
        IF  X=10  THEN  Y=3;  END IF;

        IF  X=10  THEN  Y=3;
                  ELSE  Y=5;  END IF;

        IF  X=10  THEN  Y=3;
        ELSEIF  X=20  THEN  Y=5;
        ELSEIF  X=40  THEN  Y=7;
        ELSE  Y = 0;  END IF;

    DO  LI = 1 TO ARGMAX;
        H = ARGLIST(LI);
        IF  (H=0)  QUIT DO;
        LC = .F. 1, 8, H;  VV = .F. 9, 8, H;
        IF  LC  THEN
            IF  CC=5 ! CC=6  THEN
                COUNTUP(OPRCODTABLPTR, OPRCODTABLMAX, 'OPRTAB');
                OPRCODTABL(OPRCODTABLPTR) = VV;
                END IF;
            LITTABL(CC) = VV;
        ELSE
            CC = VV;
            END IF;
        END DO;
```

NAMESET STATEMENT

Purpose: To indicate the name of a set of global variables; to begin definition of a set of global variables.

Form: NAMESET N1;

Rules:

1. The NAMESET statement is an opener. The body of the NAMESET group consists of all following statements up to and including the END statement which terminates the NAMESET group.

2. Any declaration within the NAMESET group defines a global variable, which is a member of the NAMESET.

3. The member variables of a NAMESET have distinct names.

4. The same variable name may not occur in more than one NAMESET.

5. Variables in a NAMESET may be referred to in other procedures, using the ACCESS statement.

6. Variables not contained in any NAMESET are local to the procedure in which they are defined.

7. Variables in a NAMESET may be referred to within the procedure in which they are defined. No separate ACCESS statement is needed.

8. On execution of the END statement which terminates the NAMESET group, execution proceeds to the next following statement.

Examples

```
NAMESET SYMTAB;
    SIZE SYMTABPTR(PS);  $ TOP OF SYMBOL TABLE
    SIZE  SYMTAB(WS); DIMS SYMTAB(100);
    END NAMESET SYMTAB;
```

NULL STATEMENT

Purpose: To specify no action other than continuation of processing; to simplify use of macros.

Form: ;

Rules:

1. On execution, proceed to the next statement.

(Comment: LITTLE uses the semicolon to terminate, not separate, statements. Null statements permit the use of more than one semicolon to terminate a single statement, and typically occur as a result of macro processing. Macros often consist of several statements which accomplish a given task. There then arises the question whether the semicolon terminating the last statement is to be written in the macro definition or as part of each macro invocation. LITTLE includes the null statement so that the semicolon may be written in the definition, in the call, or both, without changing the program semantics.)

Examples:

Consider

```
+*  INCR(I) = I = I+1; **

INCR(SYMPTR);
```

which, after macro expansion, yields

```
SYMPTR = SYMPTR + 1;;
```

which contains a null statement.

PROGRAM STATEMENT

Purpose: To define the program procedure.

Form: PROG N1;

Rules:

1. The program statement is an opener. The body of the program group consists of all following statements up to and including the END statement which terminates the program group.

2. An executable LITTLE program consists of one or more procedures. One and only one procedure must be a program procedure. Execution begins with the first executable statement in the program procedure and continues until execution is terminated.

3. On execution within a program procedure of a RETURN statement or of the END statement which terminates the program group, program execution terminates in a normal fashion.

Examples:

```
PROG COPYFILE;  $ LIST STANDARD INPUT FILE.
SIZE  LINE(.SDS. 80);
WHILE 1;
    GET ,SKIP :LINE,A(80);  $ READ LINE.
    IF  (FILESTAT(1,END))  QUIT WHILE;
    PUT  :LINE,A ,SKIP;
    END WHILE;
END PROG COPYFILE;
```

PUT STATEMENT

Purpose: To write data to a formatted file.

Form: PUT FID Formlist;
PUT Formlist;

Rules:

1. FID is an integer greater than zero which identifies the file. If FID is not given, take FID to be two.

2. Assert that file FID is connected with access PRINT, PUT or STRING.

3. Formlist is a list of control formats, output data items and data formats written according to the following rules.
   1. A comma precedes each control or data format.
   2. A colon precedes each data item.
   3. A data format or a data item follows a data item.

4. Transmit data according to the following rules:
   1. Transmit each data item according to the associated data format. The data format either immediately follows the data item or follows a list of data items.
   2. Transmit each element of an array block according to the associated data format.

5. If S is a character string constant, the edit specification :S,A may be abbreviated by writing a comma before the string constant. This abbreviated form ',S' is called an annotation format, as it reflects, and simplifies, the common use of character string constants to describe or annotate formatted output.

Examples:

```
FILE  FF
        ACCESS = PUT,
        TITLE = 'EXAMPLE',
        LINESIZE = 100;
PUT FF ,SKIP :X,I(5) :A(LO) TO A(HI),B(10,3);

PUT :LINE,A ,SKIP;  $ PUT  LINE TO STANDARD PRINT FILE.
PUT ,'EXECUTION TIME ' :XTIME,F(10,3) ,' MILLISECONDS.';
```

QUIT STATEMENT

Purpose: To terminate execution of an iteration.

Form: QUIT;

Rules:

1. A QUIT statement must occur within an iteration group.

2. Zero or more tokens may follow the keyword QUIT. If none are given, the QUIT statement refers to the innermost iterator. If any are given, they must correspond to the tokens which begin an iteration group containing the QUIT statement, and the QUIT statement refers to the innermost such iterator.

3. On execution, terminate the iteration and proceed to the statement following the END statement which terminates the iteration group.

Examples:

```
        QUIT WHILE MORE;
        QUIT DO;

    $ FIND INDEX OF FIRST VOWEL IN STRING STR.
    FIRSTVOWEL = 0;
    DO  I = 1 TO .LEN. STR;
        IF  (.S. I, 1, STR) .IN. 'AEIOU'  THEN  $ IF FOUND.
            FIRSTVOWEL = I;
            QUIT DO;
            END IF;
        END DO;
```

READ STATEMENT

Purpose: To read values from an unformatted file.

Form: READ FID, V1, V2,...,V9;

Rules:

1. FID is an integer greater than zero which identifies the file. The file must be connected with access READ.

2. The READ statement contains a list of read input items, separated by commas.

3. A read input item is either a simple variable or an array block item.

4. On execution, read from file FID the values of the variables and array elements specified in the list.

Examples:

```
FILE 3  ACCESS=READ, TITLE='';
READ 3, I, VOALO, VOA(VOALO) TO VOA(VOALO+10);
```

REAL STATEMENT

Purpose: To declare a real variable.

Form: REAL N1, N2,...,N9;

Rules:

1. A REAL statement contains a list of names, separated by commas.

2. The REAL statement declares each name to be a real variable with arithmetic mode real and an implementation-defined size.

Examples:

```
REAL SUMX, SUMY;
```

RETURN STATEMENT

Purpose: To terminate execution of a procedure.

Form: RETURN;

Rules:

1. On execution within a program (PROG) procedure, terminate program execution in a normal manner.

2. On execution within a subroutine (SUBR) procedure, proceed to the statement which follows the CALL statement which invoked the procedure.

3. On execution within a function (FNCT) procedure, return as value the value of the local variable of the same name as the function procedure, and continue evaluation of the expression which invoked the function procedure.

Example:

```
FNCT LASTNB(STR);  $ FIND LAST NON-BLANK CHAR IN STR.
SIZE  LASTNB(.PS.);
SIZE  STR(.SDS. 80);  $ STRING TO SEARCH.
SIZE  I(.PS.);               $ LOOP INDEX.

LASTNB = 0;  $ ASSUME STRING ALL BLANK.
DO  I = (.LEN. STR) TO 1 BY -1;
    IF  .CH. I, STR ^= 1R  THEN  $ IF NON BLANK FOUND.
        LASTNB = I;
        RETURN;
        END IF;
    END DO I;
END FNCT LASTNB;
```

REWIND STATEMENT

Purpose: To position a file at its initial point.

Form: REWIND FID;

Rules:

1. FID is an integer greater than zero which identifies the file. The file must be connected.

2. Position file FID at its initial point.

Examples:

```
REWIND SCRFILE;
```

(Comment: An implicit rewind occurs when the FILE statement is used to change access from WRITE to READ, or from PUT to GET. For example in

```
FILE 3 TITLE='', ACCESS = WRITE;
WRITE 3, A(1) TO A(10);
FILE 3 ACCESS = READ;
```

there is an implicit rewind performed as part of the second FILE statement.)

SIZE STATEMENT

Purpose: To declare a variable and give its size in bits.

Form: SIZE N1(C1), N2(C2),...,N9(C9) ;

Rules:

1. A SIZE statement contains a list of size declarations, separated by commas.

2. A size declaration consists of a name N followed by a constant C enclosed in parentheses.

3. The size declaration declares N to be of arithmetic mode integer.

3. Assert that C is greater than zero. C gives the length of N in bits.

Examples:

```
SIZE LINE (80*CS);  $ CS IS MACRO FOR CHARACTER-SIZE
SIZE ONBIT(1), LOCKBIT(1);
```

SUBR STATEMENT

Purpose: To give the name of a subroutine procedure, to give its definition.

Form: SUBR N1(N2, N3,...,N9);

Rules:

1. The subroutine statement is an opener. The body of the subroutine group consists of all following statements up to and including the END statement which terminates the subroutine group.

2. The subroutine statement contains the name of the subroutine. Subroutines may have arguments. If so, the SUBR statement contains a list, enclosed in parentheses, of the names of the formal arguments.

3. On execution of a RETURN statement or of the END statement which terminates the subroutine, execution continues with the statement following the CALL statement which invoked the subroutine procedure.

Examples:

```
     SUBR LSTLIN;
 $   LIST CURRENT INPUT LINE IF NOT YET LISTED.

     IF  LINELISTED = 0  THEN
         PUT :LINENOW,A(80) ,SKIP;
         LINELISTED = 1;
         END IF;
     END SUBR LSTLIN;
```

UNTIL STATEMENT

Purpose: To repeatedly execute a group of statements until the value of a control expression becomes nonzero.

Form: UNTIL E1;

Rules:

1. The UNTIL statement is an opener. The body of the UNTIL group consists of all following statements up to and including the END statement which terminates the UNTIL statement.

2. The UNTIL group is an iteration group.

3. On execution of the UNTIL statement, proceed to the first statement in the body of the until group.

4. On execution of the END statement which terminates the UNTIL group:

    1. If E1 is zero, continue the iteration.

    2. If E1 is nonzero, terminate the iteration.

Examples:

```
$ CONVERT INTEGER TO STREAM OF CHARACTERS.
+*  CHAROFDIG(D) = (.CH. D, '0123456789') **
UNTIL N=0;
    D = N - 10*(N/10);
    CALL PUTCHAR(CHAROFDIG(D));
    N = N / 10;
    END UNTIL;
```

WHILE STATEMENT

Purpose: To repeatedly execute a group of statements while the value of a control expression remains nonzero.

Form: WHILE E1;

Rules:

1. The WHILE statement is an opener. The body of the WHILE group consists of all following statements up to and including the END statement which terminates the WHILE group.

2. The WHILE group is an iteration group.

3. Execution proceeds as follows:
   1. If E1 is zero, terminate the iteration. If E1 is nonzero, proceed to the statement which follows the WHILE statement.

4. On execution of the END statement which terminates the WHILE group:
   1. If E1 is zero, terminate the iteration.
   2. If E1 is nonzero, continue the iteration.

Examples:

```
WHILE 1;
    CALL READLINE;
    IF (FILESTAT(INPUTFILE, END)) QUIT WHILE;
    CALL PROCESSLINE;
    END WHILE;
```

WRITE STATEMENT

Purpose: To write values to an unformatted file.

Form: WRITE FID, E1, E2,...E9;

Rules:

1. FID is an integer greater than zero which identifies the file. File FID must be connected with access WRITE.

2. The WRITE statement contains a list of write output items, separated by commas.

3. A write output item is either a variable, expression or array block item.

4. On execution, write to file FID, in order, the values of the write output items.

Examples:

```
FILE 3  ACCESS=WRITE, TITLE='';
WRITE 3, I, VOALO, VOA(VOALO) TO VOA(VOALO+10);
```

PROCEDURES AND PROGRAMS

LITTLE provides three kinds of procedures: subroutine (SUBR), function (FNCT) and program (PROG). The most basic is the subroutine. Subroutines may have arguments. A program procedure is similar to a subroutine, except that it may have no arguments, and marks the starting point of program execution. A function has a value, and is used in expressions. Procedures communicate by the association of arguments or by shared access to global variables. In LITTLE, global variables are grouped into named collections, called NAMESETs. The ACCESS statement permits one procedure to use the variables in a specified NAMESET.

Subroutine (SUBR) procedure

The compound SUBR statement defines a subroutine procedure. A subroutine may have arguments. The CALL statement initiates execution of a subroutine and supplies the actual arguments to be associated with the formal arguments during the execution. On execution within the subroutine of a RETURN statement or of the END statement which terminates the subroutine, execution continues with the statement following the CALL statement which invoked the subroutine.

Function (FNCT) procedure

The compound FNCT statement defines a function procedure. A function reference consists of the occurrence within an expression of the function name followed by list of actual arguments enclosed in parentheses. A function must have at least one argument. The function procedure must contain a local variable of the same name as the function. On execution within the function of a RETURN statement or of the END statement which terminates the function procedure, the last value assigned to the function name variable is used to continue evaluation of the expression containing the function reference.

Program (PROG) procedure

The compound PROG statement defines a program procedure. A program procedure cannot have formal arguments. (Comment: The standard LITTLE library includes procedures GETIPP and GETSPP which obtain values from the execution environment.) A LITTLE program text is executable only if it contains exactly one program procedure. Program execution begins with this program procedure. On execution within the program procedure of a RETURN statement or of the END statement which terminates the program procedure, program execution terminates in a normal manner.

PROCEDURES AND PROGRAMS

Intrinsic function procedures

LITTLE provides a number of standard functions which have a fixed interpretation. These functions should not be declared. The standard functions include the FILESTAT input/output status function and the following standard mathematical functions defined in section 4:

ABS	AINT	ALOG	ALOG10	AMOD	ATAN	ATAN2
COS	DIM	EXP	FLOAT	IABS	IDIM	IFIX
INT	ISIGN	MOD	SIGN	SQRT	TAN	

Association of arguments

A function procedure must have at least one argument; a subroutine procedure may have arguments. Every argument must be declared. The declarations define the size, arithmetic mode and possibly the dimension of the argument. The process of procedure invocation associates actual arguments with the formal arguments of the procedure. This section defines the rules of valid argument association.

Actual arguments are divided into the following classes:

Exprarg	expression, including constants and array elements
Vararg	simple variable (undimensioned variable)
Arrayarg	array name

An Exprarg can only provide a value, so that a procedure should not execute an assignment to all or part of a formal argument associated with an Exprarg. A procedure may only reference as an array a formal argument associated with an Arrayarg. The sizes of actual and formal arguments which are arrays must be equal. In any reference to a formal argument which is not qualified by an extractor, the sizes of the formal and actual arguments must be equal.

INPUT/OUTPUT: TERMS AND CONCEPTS

The input/output process (IO) associates the internal processor representaton of bit strings with external representations on various media. On input, the external representation defines the internal value; on output, the internal value defines the external representation.

A file is a sequence of external representations. A FORMATTED file consists of a sequence of lines; each line is a sequence of characters.

A formatted file is accessed with data formats and control formats. Control formats are used to position to a specific line or position within a line. Data formats specify the form of the external representation. Each data format corresponds to a type of constant; for example, the I format indicates repesentation as integer constant.

There are two types of data formats: edit and list. Each defines a field, consisting of a sequence of characters. The field may occupy more than one line. Edit data formats specify the width of the field and the structure of the data in the field. List formats are used for fields which contain data in the same form as a constant. On input, the field is determined by searching for a constant of the desired type. On output, the field width is chosen to permit a correct representation of the internal value.

The edit formats permit the use of group control to insert or ignore blanks within a field in order to represent long strings in a more readable form. For example, division of string ’EXAMPLE’ into groups of two gives ’EX AM PL E’, division of ’12378912’ into groups of three gives ’12 378 912’, and division of ’133.414’ into groups of two gives ’1 33.41 4’.

SUMMARY OF IO STATEMENTS AND FORMATS

This section summarizes the input/output statements. Files are referenced within the program as small integers, indicated by FID in the summary:

```
FILE  FID  ATTR1=VAL1, ATTR2=VAL2, ...  ATTRn=VALn;
```

The attributes and interpretation are as follows:

ACCESS	GET PRINT PUT READ STRING WRITE RELEASE
LINESIZE	Length of line in characters. If zero, processor determines length.
TITLE	If ACCESS is STRING, then variable name else string giving external name (if null then processor determines name)

```
GET FID Iolist;            Read from file.

PUT FID Iolist;            Write to file.

READ FID, V1, V2,...,V9;   Read from unformatted file.

REWIND  FID;               Rewind file.

WRITE FID, E1, E2,...,E9;  Write to unformatted file.
```

The intrinsic function FILESTAT returns the current value of a file attribute. FILESTAT has two arguments. The first argument specifies the file; the second is a keyword naming the attribute. The codes are as follows:

ACCESS	Return file type, encoded as follows: GET 1 PRINT 2 PUT 3 READ 4 STRING 5 WRITE 6 Return zero if file not connected.
COLUMN	Return column position.
END	Return one if at end of file.
ERR	If last operation had error, return error code; otherwise return zero. ERR may be written ERROR.
LINESIZE	Return current line size.
STREAM	Return one if prior operation forced streaming.

The control formats establish a position within a formatted file:

PAGE	Begin new page
COLUMN(E)	Set current column position to E
SKIP(E)	Skip E lines
X(E)	Reset current column position by adding E.

SUMMARY OF IO STATEMENTS AND FORMATS

Data formats specify the conversion of data items for formatted files:

Code	Type	Alignment	Parameters	Required parameters.
A	Edit	Left	A(FW, GW)	
AL	List		AL	
B	Edit	Right	B(FW, BW, GW)	FW, BW
BL	List		B(BW)	
E	Edit	Right	E(FW, DW, GW)	FW
EL	List		EL(SD)	SD not specified on input
F	Edit	Right	F(FW, DW, GW)	FW
FL	List		FL(SD)	SD not specified on input
I	Edit	Right	I(FW, DW, GW)	
IL	List		IL	
R	Edit	Left	R(FW, GW)	FW
RL	List		RL(NC)	

Parameter types are as follows:

FW	field width, nonnegative integer
BW	byte width, either 1,2,3 or 4
DW	decimal width, positive nonzero integer
GW	group width, positive nonzero integer
NC	number of characters
SD	number of significant digits

On output, if FW is zero, let processor determine width.
On output, if NC is zero or not given, take NC to be one.
On output, if SD is zero or not given, take SD to be six.
On output, may prefix any data format with N to indicate that name is to generated in A format.

Streaming in formatted files

Formatted files consist of a sequence of characters, grouped into lines. The end of one line is logically followed by the start of the next. The action of going from the end of one line to the start of the next is called streaming; when this occurs implicitly during processing of an Edit or List format, streaming is said to have been forced. The STREAM option of the FILESTAT inquiry can be used to determine if streaming has been forced in the prior IO operation. The SKIP control format positions to the start of a new line.

EDIT FIELDS

Edit fields represent data within a specified number of characters. The following general rules apply:

1. The edit formats have the general form

   Edit_code(Expr_list)

   where Edit_code is a single character which gives the format type and Expr_list is a list of expressions separated by commas.

2. Each edit format permits a number of parameters. In some cases, not all parameters need be written and default values are then implied for the parameters. The enclosing parentheses are written only if parameters are specified explicitly.

3. The first parameter, FW, always specifies the length of the edit field. Some edit formats permit FW to be zero on output, in which case FW is the minimum value required to correctly represent the edited value.

4. Associated with each edit format is an alignment, either left or right, as follows: A(left), B(right), E(right), F(right), I(right), R(left).

5. The last parameter specifies group control. Grouping consists of inserting (on output) or ignoring (on input) the blanks in an edit field in order to break up the represented value into readable groups. The following general rules apply:

   1. If the group width is not given, take group width to be zero.
   2. If the group width is zero, do not form groups.
   3. The group width must be greater than or equal to zero.
   4. Determine group structure according to the following rules. (exceptions are noted in the description of the individual edit formats):
      1. If the edit item is left adjusted, begin the field at the left and form groups from the left.
      2. If the edit item is right adjusted, end the field on the right and form groups from the right.
      3. On input, group control has no effect for the I and B formats, as blanks are allowed in constants of these types.
      4. On output using the numeric formats (E, F and I), insert the group separating blanks only between digits. For the E and F formats, center the groups around the decimal point.

LIST FIELDS

List fields are used to transmit data in the form of a constant. The following general rules apply:

1. The list field contains a string in the same form as a constant.

2. On output, write the value as a constant, and then write one or more blanks.

3. For PUT access, include the enclosing string delimiters for character strings; for PRINT access, do not include the enclosing string delimiters for character strings.

3. On input:

    1. Advance to the start of the data field by skipping over blanks, commas and comments.
    2. If the end of the file is encountered during the advancing, raise the end of file condition and set the value of the receiving item to zero.
    3. Accumulate characters as long as the accumulated characters define a valid representation of a constant of the desired type.
    4. Convert the accumulated characters to an internal value in the same manner as a constant.
    5. Blanks may not occur within numeric constants (integers and real quantities).
    6. Set the column position at the character following the last character accumulated, which must be a blank or comma.

CONTROL FORMATS

Control formats specify a position within a line or page. The control formats are COLUMN, PAGE, SKIP and X. Control formats require ACCESS of GET, PRINT, PUT or STRING.

Column format

Purpose: To set the current column position.

Form: COLUMN(E1)

Rules:

1. E1 is greater than zero and less than or equal to the (LINESIZE+1) of the file.

2. Set the value of the current column position to E1. Setting the position to (LINESIZE+1) indicates that the next operation is to begin at the start of the next record.

Page format

Purpose: To begin a new page on a print file.

Form: PAGE

Rules:

1. Assert that file ACCESS is PRINT.

2. Complete the current line and begin a new one.

3. The new line begins a new page.

Skip format

Purpose: To establish the next line to be processed.

Form: SKIP(E1)

Rules:

1. If E1 is not given, take E1 to be one.

2. Assert that E1 is greater than or equal to zero.

3. If E1 is zero, the skip request has no effect.

4. On input, read E1 lines. The last line read becomes the current line.

5. On output, end the current line. If E1 is greater than one, then write (E1-1) blank lines.

7. Begin a new line.

X format

Purpose: To set the current column relative to its current value.

Form: X(E1)

Rules:

1. If E1 is not given, take E1 to be one.

2. Define the new value of the current column position by adding E1 to the current value.

3. Assert that the new value is within the current line.

4. Set the current column position to the new value.

A FORMAT

A format: Edit form

Purpose: To edit data in the form of a character string.

Form: A(FW, GW)

Rules:

1. The edit field is left adjusted. GW gives the group width.

2. On input:
    1. Let RL be the largest integer such that the value of '.SDS. RL' does not exceed the size in characters of the input item.
    2. The effective field width, EFW, is the minimum of RL and FW. EFW must be greater than zero.
    3. The first EFW characters in the field define a character string constant.

3. On output:
    1. Assert that the output item is a character string. Let SL be the length of this string.
    2. If FW is not given, or is given with value zero, take FW to be SL.
    3. The effective field width, EFW, is the minimum of SL and FW.
    4. The first EFW characters of the output item define the data in the edit field.

A format: List form

Purpose: To transmit data in the form of a character string constant

Form: AL

Rules:

1. On input, the list field contains a character string constant.

2. On output:
    1. If the ACCESS is PRINT, the list field contains the characters in the sending item.
    2. Otherwise, the list field contains the character string constant defined by the sending item.

B FORMAT

B format: Edit form

Purpose: To edit data in the form of a bitstring constant.

Form: B(FW, BW, GW)

Rules:

1. The edit field is right adjusted.

2. BW is the byte width and must have a value of 1, 2, 3 or 4. BW gives the byte width of each character in the edit field in the same manner as for a byte constant. BW must be given.

3. On input:
   1. The edit field must contain only characters that are valid within the value part of a bitstring constant of byte width BW.
   2. The group width GW has no effect.

4. On output:
   1. If the field width FW is not given, take FW to be zero.
   2. If the field width is zero, then the field width is EFW, where EFW is the least integer such that the value of 'EFW*BW' is greater than or equal to the value of '.FB. SI', where SI is the output item. If EFW is zero, take EFW to be one.
   3. The edit field contains the delimited part of the bitstring constant of byte width BW defined by the sending item.
   4. If BW not given, take BW to be a processor-defined value. (Comment: the implied value will typically be the standard byte width used for the machine.)

B format: List form

Purpose: To transmit data in the form of a bitstring constant.

Form:

BL	input
BL(BW)	output

Rules:

1. The list field has the form of a bitstring constant .

2. On output, BW specifies the byte width to use (1, 2, 3 or 4). If BW is not given, take BW to be a processor-defined value.

E FORMAT

E format: Edit form

Purpose: To edit data in the form of a real constant.

Form: E(FW, DW, GW)

Rules:

1. The edit field is right adjusted, and contains numeric data. GW gives the group width.

2. FW is the field width and must be given.

3. On input:
    1. The edit field contains a real constant.
    2. DW is the decimal width. If not given, take DW to be zero.
    3. If the edit field does not contain a decimal point, the position of the decimal point is implied by DW and the internal value corresponds to division of the constant defined in the field by the value of 10 raised to the power DW.

4. On output:
    1. The value of the sending item approximately defines a real constant.
    2. DW is the decimal width. If not given, take DW to be zero.
    3. The edit field contains in order the following parts: a sign representation, a decimal representation and an exponent representation.
    4. If the sending item has value zero, the field contains '0.'.
    5. If the value is nonzero the decimal representation consists of a nonzero digit followed by a decimal point followed by DW digits.
    6. The exponent representation consists of the letter E followed by a signed integer.

E FORMAT

E format: List form

Purpose: To transmit data in the form of a real constant.

Form:		
	EL	Input
	EL(SD)	Output

Rules:

1. On input, the list field contains a real constant.

2. On output:
    1. SD gives the number of significant digits. If SD is not given or has value zero, take SD to be six.
    2. Represent the internal value as a floating point constant with SD significant digits and with an exponent of at least two significant digits. The decimal point follows the first digit.

F FORMAT

F format: Edit form

Purpose: To edit data in the form of a real constant.

Form: F(FW, DW, GW)

Rules:

1. The field is right adjusted and contains numeric data. GW specifies the group width.

2. F input format is the same as E input format.

3. On output:
    1. DW is the decimal width. If DW is not given, take DW to be zero.
    2. The field consists of a real constant with no exponent part and DW digits after the decimal point.

F format: List form

Purpose: To transmit data in the form of a real constant.

Form:

FL	Input
FL(SD)	Output

Rules:

1. The list field contains a real constant.

2. On output:
    1. SD gives the number of significant digits. If SD is not given or is given with value zero, take SD to be six.
    2. Represent the internal value as a floating point constant with SD significant digits.

I FORMAT

I format: Edit form

Purpose: To edit data in the form of an integer constant.

Form: I(FW, DW, GW)

Rules:

1. The edit field is right adjusted and contains numeric data. GW gives the group width.

2. The edit field contains an integer constant.

3. DW gives the decimal width. If DW is not given, take DW to be one. The field contains at least DW digits with leading zeros added if necessary.

I format: List form

Purpose: To transmit data in the form of an integer constant.

Form: IL

Rules:

1. The list field contains an integer constant.

R FORMAT

R format: Edit form

Purpose: To edit data in the form of character codes.

Form: R(FW, GW)

Rules:

1. The edit field is left adjusted. GW specifies the group width.

2. FW is the field width. If FW is not given or has value zero, take FW to be one.

3. On input, the edit field defines a character code constant of length FW.

4. On output:
    1. Let SL be the largest integer such that '.CS. * SL' does not exceed the size of the sending item. If SL is zero, set SL to one.
    2. The effective field width, EFW, is the minimum of FW and SL. The edit field contains the EFW characters defined by the rightmost EFW character codes in the sending item.

R format: List form

Purpose: To transmit data in the form of a character code constant.

Form:

RL	Input
RL(FW)	Output

Rules:

1. On input, the list field contains a character code constant.

2. On output:
    1. FW gives the number of character codes to transmit.
    2. If FW is not given, take FW to be one.
    3. The list field contains a character code constant with FW codes. codes.

NAMING OUTPUT ITEMS

Purpose: To write the name of a data item.

Form: The letter N occurs before the data format code.

Rules:

1. On output to a formatted file, may prefix any data format with the letter N.

2. Let DI be the data item to which the data format applies. Define a character string S which names DI as follows:
   1. If DI is an expression, S is the null string ''.
   2. If DI is a simple variable, S is the name of the variable.
   3. If DI is an array element, S is the name of the array followed by a left parenthesis followed by the subscript value represented as an integer, followed by a right parenthesis.

3. Write the name string S defined by rule 2 and then write the string ' = '.

Examples:

```
PUT   :S, NI(3);

PUT  :A(1) TO A(AMAX),NI(6) ,SKIP;
```

STRING FILES

The formatted IO features support the construction of lines by conversion and editing, and the transmission of lines to external media. The file access STRING permits use of the editing and conversion features without the cost of transmitting data to external media. The following rules govern the use of file access STRING:

1. Establish access STRING by a FILE statement of the form

```
FILE  FID ACCESS=STRING, TITLE=V1, LINESIZE=I1;
```

   where V1 is a character string, and I1 defines the length of the line.

2. Both PUT and GET statements can be used if access is STRING. Assert that on execution of a GET or PUT statement, the current length in characters of V1 is greater than or equal to the LINESIZE I1.

3. The following actions set the column position of a STRING file to one:
   1. Execution of the FILE statement.
   2. Execution of a REWIND statement.
   3. Execution of a SKIP or PAGE format.
   4. Execution of a PUT statement which forces streaming.

4. Execution of a PUT statement which forces streaming does not clear the string to blanks.

5. Execution of a GET statement which forces streaming raises the file END condition.

Examples:

```
$   SHOW HOW TO OBTAIN VALUES USING
$   STRING IO.  THIS METHOD COULD BE USED TO OBTAIN
$   PROGRAM PARAMETERS OR DATA IN 'FREE FIELD' FORM.

    SIZE  SFILE(.SDS. 80);
    SIZE  I(.PS.);      $ LOOP INDEX.
    SIZE  ARA(.WS.);  DIMS  ARA(100);
    SIZE  ARAPTR(.PS.);     $ NUMBER OF VALUES OBTAINED.
    FILE  3 ACCESS=STRING, TITLE=SFILE, LINESIZE = 80;
    SFILE = ' 1 3 5 -20  123 ' .PAD. 80;

    ARAPTR = 0;
    DO  I = 1 TO 100;
        GET 3 :V,IL;  $ GET NEXT VALUE.
        IF  (FILESTAT(3, END)) QUIT DO;
        ARAPTR = ARAPTR + 1;
        ARA(ARAPTR) = V;
        END DO;
```

FILESTAT REQUEST.

Purpose: To obtain file attributes or status of last input/output operation.

Form: FILESTAT(FID, SCODE)

Rules:

1. FILESTAT is an intrinsic function.

2. FID is the file number as established by a FILE statement.

3. SCODE names the type of the request and must be one of the following:

ACCESS COLUMN END ERROR ERR LINESIZE STREAM

4. The result is zero, except as noted below.

5. ACCESS returns the file access mode, as follows:

Type	Value
	0 (file not connected) (Must exeucute FILE statement before any other IO operation on file.)
GET	1
PRINT	2
PUT	3
READ	4
STRING	5
WRITE	6

6. COLUMN returns the current column position of formatted file.

7. END returns one if the end of an input file has been encountered.

8. ERROR is nonzero if an error condition exists, as follows:

value	meaning
1	truncation or conversion error.
2	format error or error doing operation

FILESTAT(F, ERROR) may also be written FILESTAT(F, ERR).

9. LINESIZE returns the LINESIZE of a formatted file.

10. STREAM is one if the preceding IO action forced streaming to a new line.

MONITOR FACILITIES

```
        /*  Mr. Edison, I was informed, had been up the two previous
            nights discovering a 'bug' in his phongraph - an
            expression for solving a difficulty, and implying
            that some imaginary insect has secreted itself inside
            and is causing all the trouble.
           --Supplement to Oxford English Dictionary, under 'bug'  */
```

LITTLE provides several tools to monitor program execution. These tools may assist in the identification of program errors, and provide many of the features usually found in a 'debugging' package. The tools permit the following:

```
    trace of procedure entry and exit
    report of control flow during procedure invocation
    trace of stores (assignments) to selected variables
    check that index valid on array assignment
    verification of program 'assertion'
```

The monitor package includes the following compilation directives:

```
    CHECK (NOCHECK) INDEX;      Check (do not check) index on array
                                assignment.

    CHECK (NOCHECK) INDEX A1,...,A9;  Check (do not check) index
                                on assignments to selected arrays.
    TRACE (NOTRACE) ENTRY;      Trace (do not trace) procedure
                                entry and exit.
    TRACE (NOTRACE) FLOW;       Trace (do not trace) procedure
                                control flow.
    TRACE (NOTRACE) STORES;     Trace (do not trace) assignments
                                to all variables.
    TRACE (NOTRACE) STORES N1,...,N9;  Trace (do not trace)
                                assignments to selected variables.
```

The monitor package includes the following executable statements:

```
    ASSERT E1;                  Verify that E1 is nonzero.

    MONITOR OPTIONLIST;         Set monitor option, as follows:
        BYTE (NOBYTE)           Guarantee (do not guarantee) that
                                bitstring value always given.
        ENTRY (NOENTRY)         Enable (disable) listing of results
                                of TRACE ENTRY directives.
        FLOW (NOFLOW)           Enable (disable) listing of results
                                of TRACE FLOW directives.
        LIMIT = E1              Set monitor line limit to E1.
        STORES (NOSTORES)       Enable (disable) listing of results
                                of TRACE STORES directives.
```

Monitor conventions.

Several of the monitor directives have two forms - one to enable a feature and one to disable it. The prefix NO systematically identifies the disabling case. For example, TRACE ENTRY enables tracing of procedure entry, while NOTRACE ENTRY disables this trace.

MONITOR FACILITIES

The directives CHECK and TRACE may contain a list of names. If no names are given, the directive has maximal scope in that all names are referred to. If a list is given, only the named items are indicated. For example, 'TRACE STORES A,X;' directs tracing of stores to 'A' and 'X', while the directive 'NOTRACE STORES;' disables stores tracing for all variables.

The scope of a CHECK or TRACE directive is determined as follows: an interlude of a program consists of the text before the start of the first procedure definition, or of the text between the END statement terminating a procedure definition and the start of the next procedure definition. Directives within an interlude establish defaults for following procedures; directives within a procedure definition apply only to that procedure. The TRACE ENTRY and TRACE FLOW directives must appear in an interlude.

Checking index range on array assignment

The CHECK directive requests checking that the index is valid on an array assignment. The NOCHECK directive suppresses this check. The CHECK directive either requests checking of all arrays, or contains a list of names of the arrays to check.

Tracing procedure entry and exit

The TRACE ENTRY directive enables tracing of procedure entry and exit. The NOTRACE ENTRY directive disables this trace. This directive must occur in an interlude.

Tracing control flow

The TRACE FLOW directive enables the tracing of control flow within a procedure invocation. The NOTRACE FLOW directive disables this trace. This directive must occur in an interlude.

Tracing assignments.

The TRACE STORES directive enables tracing of assignments. The NOTRACE STORES directive disables this trace. The directive either enables tracing of assignments to all variables or contains a list of the variables to be traced.

MONITOR FACILITIES

Verification of program assertions

If a program is valid only if certain conditions hold, use the ASSERT statement to verify these assertions at execution time. The ASSERT statement has the form

```
ASSERT E1;
```

On execution, if E1 is nonzero, execution proceeds to the next statement; otherwise, execution terminates abnormally.

Execution time monitor options

The MONITOR statement defines execution-time options of the monitor package. A MONITOR statement must occur within a procedure definition. The MONITOR statement consists of a list of options separated by commas. The options are as follows:

1. The BYTE option enables listing of values in byte form. The NOBYTE option disables this listing. The monitor package picks a format to print a value. If the BYTE option is on, the value as a bitstring is also listed if another format is chosen.
2. The ENTRY option enables listing of the output of TRACE ENTRY directives. The NOENTRY option disables this listing.
3. The FLOW option enables listing of output of TRACE FLOW directives. The NOFLOW option disables this listing.
4. The LIMIT option has the form LIMIT = I1 where I1 is an integer giving the monitor line limit. When the number of lines produced by monitor directives equals or exceeds this limit, the options ENTRY, FLOW and STORES are disabled, to minimize further monitor output.
5. The STORES option enables listing of the output of TRACE STORES directives. The NOSTORES option disables this listing.

Monitor level

There are several levels of monitor processing, as follows:

0. Ignore all monitor statements and directives.
1. Process only ASSERT statements.
2. Process all Monitor directives and statements.

The default level is one. The compiler option MLEV sets the level, as does the compiler option HELP.

Compiler option HELP

The compiler option HELP aids use of the monitor package for debugging. The parameters of HELP consist of a sequence of character codes, which correspond to initial monitor directives, as follows:

C CHECK INDEX on
E TRACE ENTRY on
F TRACE FLOW on
S TRACE STORES on

The default is 'HELP=0'. If 'HELP' alone given, then 'HELP=ES' is taken. If any of the options are selected, the monitor level is set to two.

REFERENCES

(/1/) P. Lecht. The Programmer's PL/I. McGraw Hill (1970).

(/2/) J. Cocke and J. Schwartz. Programming Languages and Their Compilers. Computer Science Department, Courant Institute of Mathematical Sciences, New York University (1970).

(/3/) P.C. Poole and W.M. Waite. 'Portability and Adaptability'. in: Advanced Course on Software Engineering, ed. Bauer, Springer-Verlag, Berlin and New York (1973).

(/4/) V. Basili. SIMPL-X - A Language for Writing Structured Programs. Technical Report TR-223, University of Maryland (1973).

(/5/) R. Griswold. The Macro Implementation of SNOBOL 4. W.H. Freeman and Co, San Francisco (1972).

(/6/) W.A. Wulf, et. al. BLISS Reference Manual. Computer Science Department Report, Carnegie Mellon University, Pittsburgh, Penn., (1969).

(/7/) G. Goos. 'Language Characteristics: Programming Languages as a Tool in Writing Systems Software'. In: Beekmann, op. Cit.

(/8/) P. J. Brown. Macro Processors and Techniques for Portable Software. John Wiley and Sons, New York (1974).

(/9/) R. Griswold, J. Poage and I. Polonsky. The SNOBOL4 Programming Language. Second Edition, Prentice-Hall (1971).

LITTLE GRAMMAR

The following grammar for LITTLE may aid in understanding its syntactic details. The notations used in the grammar are as follows:

```
    <stype>        denotes a syntactic type
    <*ltype>       denotes a lexical type
    lit  'lit'     denotes literal
    <-comment>     denotes comment about the grammar
    <stype*>       denotes varying number of instances (maybe none)
                   of a syntactic type
    <stype(m,n)>   denotes limited number of repetitions of
                   a syntactic type
                     m : minimum number required
                     n : maximum number allowed
```

Successive alternative expansions of a syntactic type are indicated by successive equal signs, as follows:

```
    <stype>  := (first alternative)
             := (second alternative)
             :=...(remaining alternatves)     .
```

The following lexical types appear in the grammar

```
<*name>      variable or procedure name
<*con>       a constant, integer, bit or string
             Examples:   10    'A STRING'   3B'33'    1RC
<*notsemicolon>  any character except semicolon
```

```
<program>           := <block>

<block>             := <statement> <statement*>

<statement>         := <declstatement>
                    := <compstatement>
                    := <simplstatement>
                    := <simplifstatement>

<declstatement>     := SIZE <attrspec> <cattrspec*> ;
                    := REAL <*name> <cname*> ;
                    := DIMS <attrspec> <cattrspec*> ;
                    := DATA <dataspec> <coldataspec*> ;
                    := ACCESS <*name> <cname*> ;

<compstatement>     := <opener> <block> <ender>
                    := <ifstatement>

<ifstatement>       := IF <expr> THEN <block> <elseifblock*>
                       <ender>
                    := IF <expr> THEN <block> <elseifblock*>
                       ELSE <block> <ender>

<elseifblock>       := ELSEIF <expr> THEN <block>

<opener>            := NAMESET <*name> ;
```

LITTLE GRAMMAR

```
                    := PROG <*name>;
                    := SUBR <*name> <arglist> ;
                    := SUBR <*name> ;
                    := FNCT <*name> <arglist> ;
                    := WHILE <expr> ;
                    := UNTIL <expr>;
                    := DO <*name>  = <expr> TO <expr> BY - <expr> ;
                    := DO <*name>  = <expr> TO <expr> BY <expr> ;
                    := DO <*name>  = <expr> TO <expr> ;

<simplifstatement>  := IF <expr> <simplstatement>

<simplstatement>    := CALL <*name> ( <expr> <cexpr*> );
                    := CALL <*name> ;
                    := CONT <notsemi(0,5)> ;
                    := GO TO <*name> ;
                    := GO TO <*name> ( <expr> ) ;
                    := GO TO <*name> ( <expr> )
                       IN <const> TO <const> ;
                    := QUIT <notsemi(0,5)> ;
                    := RETURN ;
                    := REWIND <expr> ;
                    := FILE <expr> <filelist> ;
                    := GET <expr> <iolist> ;
                    := GET <iolist> ;
                    := PUT <expr> <iolist> ;
                    := PUT <iolist> ;
                    := READ <expr> , <expr> <cexpr*> ;
                    := WRITE <expr> , <expr> <cexpr*> ;
                    := <assignstatement> ;

<arglist>           := ( <*name> <cname*> )

<cname>             := , <*name>

<ender>             := END <*notsemi(0,5)> ;

<notsemi>           := <*notsemicolon>

<attrspec>          := <*name> ( <constexpr> )

<cattrspec>         := , <attrspec>

<file>              := <constexpr>

<assignstatement>   := <targpart> <target>  = <expr>
                    := <target>  = <expr>

<targpart>          := .F. <expr> , <expr> ,
                    := .E. <expr> , <expr> ,
                    := .S. <expr> , <expr> ,
                    := .CH. <expr> ,
                    := .LEN.

<target>            := <*name> ( <expr> )
                    := <*name>
```

LITTLE GRAMMAR

```
<dataspec>          := <*name> ( <constexpr> )  = <dataval*>
                    := <*name>  = <dataval*>

<dataval>           := <dataexpr> <cdataexpr*>

<cdataexpr>         := , <dataexpr>

<coldataspec>       := : <dataspec>

<dataexpr>          := <constexpr> ( <constexpr> )
                    := <constexpr>

<iolist>            := <ioitem> <ioitem*>

<ioitem>            := , <controlformat>
                    := <iodataitem>

<controlformat>     := SKIP (<expr>)
                    := SKIP
                    := X (<expr>)
                    := X
                    := PAGE
                    := COLUMN ( <expr> )
                    := <*string>

<cexpr>             := , <expr>

<expr>              := <-described in section 4>

<constexpr>         := <- expression with only constants as operands>

<dataitem>          := : <dataitem> , <dataformat>
                    := : <*name>
```

CODING CONVENTIONS

This section describes the coding conventions used in writing software for the LITTLE system. Systematic coding style improves readability. The conventions include basic text format, choice of names, comments and program organization. The programs in appendix G follow these conventions.

Basic text format

A line of LITTLE text contains 72 columns. Leave column one blank, and begin text of each line in column seven, except as noted below.

Except for procedure definitions and perhaps NAMESET definitions, indent each statement in the body of a compound group four spaces, including the terminating END statement. The standard 'tabs' are thus 7, 11, 15, 19, ...

Begin the ELSEIF and ELSE clauses of a compound IF in the same column as the opening IF. In general, put the ELSE of an ELSE clause on a separate line.

Spacing rules

1. Space twice after the following if they begin a line

   DATA DIMS DO IF REAL SIZE +*

2. Space twice before and after the condition of an IF or ELSEIF clause. Space twice before and after THEN.
3. Space twice after the following: ; ** .
4. Space twice before '$' which begins comment and space at least once after it.
5. Space at least once after every comma, except as noted in next rule.
6. In GET or PUT statements,
   1. Space once before each comma which is followed by a control format.
   2. Space once before each colon which marks a data item.
   3. Do not space before or after a comma which is followed by a data format.

   For example, 'PUT ,SKIP ,X :A :B,NIL ,SKIP;'.

Never terminate two compound groups on the same line if neither begins on that line.

CODING CONVENTIONS

Blank lines

Use blank lines to improve readability, but do not use them in excess. Use a single blank line to indicate a break in the program. Use two blank lines to indicate a major break. Use of more than two blank lines at a time is generally no more informative than a single blank line, and serves only to waste paper.

Put a blank line before and after compound groups which contain more than a few lines, to indicate the scope of the group.

Avoid banners

A banner consists of one or more lines which do nothing other than announce their presence. A common use is to announce comments, as in

```
/* * * * * * * * * * * * * * * * * */   .
```

Comments are an integral part of a well-written program and so need no banners to announce their presence. A better approach is to use blank lines as noted above, or to construct programs which extract and list special text groups.

Statement labels

Use as few statement labels as necessary. Use a separate line for labels and begin the label in column two, unless need for indenting indicates otherwise. Use descriptive names for labels, except that L is commonly used for an indexed GO TO if a procedure contains a single indexed GO TO.

Identify terminating END

Always put at least the opening keyword after the END in an END statement. Do this also for CONT and QUIT statements; for example:

```
END PROG MAIN;    QUIT DO;   END IF;    CONT WHILE;    .
```

Choice of names

1. Use short names for items used many times and longer names for items infrequently referenced.
2. When a name is not obvious, include a comment which explains the derivation of the name; for example:

   ```
   LTLDOC   $ LTL-LITTLE DOC-UMENTATION PROGRAM.
   LCS  $ L-IST C-OMPILATION S-TATISTICS.
   ```
3. In choosing related names, use a common prefix, as this makes

cross-reference list easier to read. For example, write 'HAPTR, HAMAX and HAORG' and not 'PTRHA, MAXHA and ORGHA'.

4. Avoid names which are common words, such as 'table'. Invented names stand out in text and require no special delimiters to distinguish the name from the common word.

Program organization

A LITTLE program text consists of the following sections:

1. INTRO: an introductory section consisting entirely of comments which name the program, give its purpose and identify the authors.
2. MODS: a section consisting entirely of comments which describes each change to the program. The text of each mod should contain the name of the author, the date of change, the purpose of the change, and a list of the procedures affected.
3. GLOSSARY: the glossary is needed for large programs, and contains an alphabetical list of names of macros, procedures and variables, with one name to a line; for example,

   $ HAMAX: DIMENSION OF HA.'

4. MACROS: this section contains the standard macros as follows:
   1. Conditional assembly options.
   2. Machine parameters.
   3. Program parameters, such as table lengths.
   4. Program codes, typically macros naming constants.
   5. Code macros.
5. MAIN: this section defines the first procedure, typically a program procedure and contains the declarations for global variables.
6. PROCS: remaining procedure definitions.

Comments

1. Use rest-of-line comments in preference to delimited comments. Use delimited comments only for comments of several lines.
2. Every variable declaration requires a comment. Use a paragraph of comments just before the declarations of several related variables, or an end-of-line comment for a single variable.
3. Put a period at the end of every comment.
4. Write comments in active voice. Comments should define the task to be done.
5. Good comments are an essential part of a quality program.
6. Begin each procedure with a paragraph of comments which define the purpose of procedure.

INTRODUCTION TO MACROS IN LITTLE

Macro processors are not commonly found in high level languages, and this appendix expands on the definition of the LITTLE macro processor given in section 2.3 to provide an informal introduction to the LITTLE macro processor. A book by Brown (/8/) contains an excellent introduction to macro processors and techniques for writing portable software.

A macro processor provides a tool for transforming one set of symbols into another. A macro definition specifies a transformation rule. A macro call applies the transformation rule to a specific sequence of symbols. The symbols resulting from the transformation replace the macro call.

The simplest form of a macro definition is shown by

```
+*  MAXTOKLEN = 127 **  $ MAXIMUM TOKEN LENGTH.
```

The successive plus and asterisk characters mark the start of the macro definition, and are followed by the macro name MAXTOKLEN. The macro body begins with the symbol following the equal character after the macro name and is ended by the two successive asterisk characters that terminate a macro definition.

A subsequent instance of MAXTOKLEN in the source text causes the replacement of MAXTOKLEN by the macro body so that, for example, the statement

```
ASSERT TOKLEN <= MAXTOKLEN;
```

becomes

```
ASSERT TOKLEN <= 127;
```

This use of the macro processor improves the program in two ways. First, the statement 'ASSERT TOKLEN <= MAXTOKLEN;' is more readable than 'ASSERT TOKLEN <= 127;'. Second, and of more importance, in order to change the program to accept tokens of a different maximum length it is only necessary to change the macro definition, as in

```
+*  MAXTOKLEN = 255  **  $ MAXIMUM TOKEN LENGTH.
```

The macro processor can be used to simplify coding, as shown by

```
+*  WS = .WS. **  +* CS = .CS. **
```

which permits the writing of WS and CS to express the environment word and character sizes, respectively. Macros can be used within macros, as shown by

```
+*  CPW = (WS/CS) **  $ CHARACTERS PER WORD.
+*  HASHTOKORG = $ ORIGIN OF HASH TOKEN STRING.
    (1 + .SDS. (MAXTOKLEN+CPW)) **
```

Note that a macro definition may be written on a single line or on several lines, and a line may contain several macro definitions.

Another common use of the macro processor is to express

environment-dependent program parameters, as shown by

```
        +*  NCHARS =  $ NUMBER OF CHARACTER CODES.
 .+S37      256  $ IBM SYSTEM/370.
 .+S66       64  $ CDC 6000 SERIES.
            **
```

This macro uses the conditional assembly feature to select the macro body. If an attempt is made to compile the program in a new environment, the macro definition becomes

```
        +*  NCHARS = **
```

which has an empty (null) macro body. The macro processor treats this definition form as a request to drop the macro status of the name, and subsequent instances of the name are not transformed. For example, NCHARS just remains NCHARS. An attempt to compile the program will result in compilation errors since NCHARS will appear to be an undeclared variable. This use of the macro processor thus makes environment-dependent parameters very visible, and requires that a new value be specified when attempting to move the program to a new environment.

Another common use of the macro processor is to name the fields of a bitstring, as shown by the macros

```
        +*  HA_LEXLEN  = .E. 05, 07, **  $ TOKEN LENGTH.
        +*  HA_NAMEPTR = .E. 20, 13, **  $ NAMES INDEX.
        +*  HA_MACORG  = .E. 35, 13, **  $ MACRO ORIGIN.
```

which define some of the fields in the symbol table of the LITTLE lexical scanner. Uses of such macros have the form

```
        HA_LEXLEN SYMTAB(I)
```

Instances of consecutive names, with no intervening operators or delimiters, usually indicate the use of an extractor in a LITTLE program.

The macro processor permits macros to have arguments. For example, suppose the array TOKARA is used to accumulate the characters of a token. Before adding a new character to the array, it is necessary to check that there is room for it, by writing code of the form

```
        ASSERT TOKARAPTR < MAXTOKLEN;
        TOKARAPTR = TOKARAPTR + 1;
        TOKARA(TOKARAPTR) = NEWCHAR;
```

The basic check can be expressed as a macro by writing

```
        +*  COUNTUP(VAR, LIM) = $ INCREMENT WITH LIMIT CHECK.
            ASSERT VAR < LIM;
            VAR = VAR + 1;  **
```

This macro definition has the same form as a simple definition except that the macro name is followed by a parenthesized list of names separated by commas. The code can now be written

```
COUNTUP(TOKARAPTR, MAXTOKLEN);
TOKARA(TOKARAPTR) = NEWCHAR;
```

However, the basic action is just to add the character, and can be written as

```
+*  ADDCHAR(C) = $ ADD C TO TOKARA.
    COUNTUP(TOKARAPTR, MAXTOKLEN);
    TOKARA(TOKARAPTR) = C;  **
```

so that the original code can now be written as just

```
ADDCHAR(NEWCHAR);
```

which, as a result of expanding macros ADDCHAR and COUNTUP, yields

```
ASSERT TOKARAPTR < MAXTOKLEN;
TOKARAPTR = TOKARAPTR + 1;;
TOKARA(TOKARAPTR) = NEWCHAR;;
```

The consecutive semicolons result from the writing of a semicolon both in the macro definition and after the macro call. However, a statement consisting only of the statement-terminating semicolon is a null statement in LITTLE. Indeed, LITTLE includes the null statement just to avoid this problem.

As another example, suppose the array STK is used to simulate a stack, with STKPTR the index of the top of the stack. The stack primitives PUSH and POP can be implemented by writing

```
+*  PUSH(I) =  $ PUSH I ON STACK.
    COUNTUP(STKPTR, STKLIM);  $ TEST FOR OVERFLOW.
    STK(STKPTR) = I;  **

+*  POP(I) =   $ POP STACK TO I.
    I = STK(STKPTR);
    ASSERT STKPTR > 0;  $ TEST FOR UNDERFLOW.
    STKPTR = STKPTR - 1;  **
```

The ADDCHAR, PUSH and POP macros illustrate the use of the macro processor to permit the writing of code in a form close to the programmer's view of the algorithm. This adaption can also be done by using procedures, writing code of the form

```
CALL PUSH(ELM); ...  CALL POP(LAST);
```

A merit of the macro processor is that it permits the use of both direct 'in-line' code and 'off-line' procedures, or combinations thereof. For example, if speed is of the essence, the POP macro can be defined, as above, to expand into code; however, if it is more important to reduce program size, and there are many calls to POP, the POP macro can be defined to expand into a procedure call.

INTRODUCTION TO MACROS IN LITTLE

The definition of a macro in which the body is procedural in nature and requires its own local variables or statement labels must be done with care. For example, consider writing a macro NBLANKS which is to return the number of blank characters in a character string. A first attempt is to write

```
+*  NBLANKS(S, NB) = $ SET NB TO NUM. OF BLANKS IN S.
    NB = 0;
    DO  I = 1 TO .LEN. S;
        NB = NB + (.CH. I, S = 1R );
        END DO;
    **
```

The loop index I is local to the macro, and the user of the macro should not even know of the existence of I. LITTLE provides special symbols ZZZA, ZZZB, ..., ZZZZ (and also ZZZ_, since _ can be used in names) which can be written in a macro definition and which cause the generation of new names on macro expansion. The correct definition is

```
+*  NBLANKS(S, NB) = $ SET NB TO NUM. OF BLANKS IN S.
    SIZE  ZZZI(PS);  $ LOOP INDEX.
    NB = 0;
    DO  ZZZI = 1 TO .LEN. S;
        NB = NB + (.CH. ZZZI, S = 1R );
        END DO;
    **
```

LITTLE also provides ZZY symbols which generate integer constants private to a macro expansion.

As has been shown, macros can be used within macro definitions. An obvious question is to ask whether macro definitions can be written within macro definitions. The LITTLE macro processor answers yes, but this point requires some discussion. For example, consider the macro definition

```
+*  M1(A, B) =
    +*  M2 = B **
    A = B + 1;
    **
```

One approach is to have the macro processor absorb the definition of M2 while processing the definition of M1. This is not very useful, as M2 is then just a macro giving the name B of the second argument of M1. A second and much more powerful approach is to defer the definition of M2 until M1 is called. The LITTLE macro processor takes the second approach. For various technical reasons, this requires the use of the following macros

```
+*  Q3(A, B, C) = A B C **
+*  MACDEF(MACTXT) = Q3(+, *MACTXT*, *)  **
```

to effect definition of macros within macros. The main point is that during expansion of a macro containing a call of MACDEF, the expansion of the MACDEF macro provides the '+*' and '**' delimiters before and after MACTXT, so that MACTXT is recognized as a macro definition.

INTRODUCTION TO MACROS IN LITTLE

As an example of the use of macro definitions within macros, consider the coding of an interpreter for a hypothetical machine which has fifteen opcodes. The crudest approach would be just to assign the codes and use their integer values, although the resulting code would not be very readable. A simple use of the macro processor is to write macros of the form

```
+*  OP_ACT = 01 **
+*  OP_BAK = 02 **
+*  OP_END = 03 **
```

The MACDEF device permits the assigning of codes without the need to specify the values. This is done by writing

```
+*  DEFC(C) = MACDEF(C=ZZYA) **
DEFC(OP_ACT)  $ ASSIGN CONSTANT CODE FOR ACT.
DEFC(OP_BAK)
DEFC(OP_END)
```

This method permits the addition of new codes, and changing the order of the codes simply by adding a new DEFC macro call or by changing the order of the calls. There is a problem if DEFC macro used to define several sets of codes, as each set would begin with a code corresponding to one more than the code assigned to the last element in the prior set. LITTLE permits the resetting of the ZZY symbols to their initial value, by use of the ZZYORG directive line, which has the form

```
.=ZZYORG   AXB
```

and which, in this example, resets the ZZYA, ZZYX and ZZYB symbol origins. The period must be written in column two.

Although LITTLE provides only one-dimensional arrays, the macro processor can be used to provide much of the effect of multi-dimensional arrays. For example, consider the macro

```
+*  DEFARA2(MAC, ARA, NR, NC, DIM) =
    MACDEF(MAC(I,J) = ARA((I)+((J)-1)*NR))
    MACDEF(DIM = (NR*NC))  $ DIMENSION OF ARA.
    **
```

which permits the writing of MAC(I,J) to refer to an element of the two-dimensional array with NR rows and NC columns. The elements are stored in the one-dimensional array ARA and DIM gives the dimension of ARA. For example

```
DEFARA2(MAT, MATARA, 10, 20, MATARADIM)

SIZE  MATARA(WS);
DIMS  MATARA(MATARADIM);
DATA  MATARA = 0(MATARADIM)
```

sets up an 10 by 20 matrix whose elements can be referenced by writing MAT(4,I), MAT(3*J, K), etc. Similar methods can be used to define ’zero origin’ arrays, and so forth.

Instances of macro arguments within expressions in the macro body should be enclosed in parentheses. To see the need for this consider the macro

```
+*  MUL5(X) = X*5  **  $ MULTIPLY BY FIVE.
```

The macro call MUL5(N) expands to N*5; however, the macro call MUL5(A+B) expands to A+B*5, and since multiplication has a higher precedence than addition, the expression is interpreted as A+(B*5) instead of the desired (A+B)*5. Such problems are avoided by writing the macro as

```
+*  MUL5(X) = (X)*5 **  $ MULTIPLY BY FIVE.
```

Macro arguments may in general consist of an arbitrary sequence of tokens. The arguments may contain calls to other macros. However, since macro arguments are separated by commas, some care must be taken in writing arguments which contain commas. For example, the call MUL5(.E. 5, 6, A(I)) would be detected as erroneous by the macro processor, as it is taken to be a call with the three arguments '.E. 5', '6', and 'A(I)', while MUL5 has only one argument. The correct way to write this macro call is to write MUL5((.E. 5, 6, A(I))) as the macro processor permits commas in arguments which are enclosed in parentheses. If the extractor were defined by a macro such as '+* FLD = .E. 5, 6, **' then the macro call could be written MUL5(FLD A(I)).

Yet another use of the macro processor is to assist in the initialization of program data. For example, the following text fragment taken from the IBM System/370 code generator for LITTLE shows how some of the machine attributes are specified in a readable fashion.

```
$   FIELDS IN -MOPTAB-.

+*  MT_OP     = .F. 01, 8, **  $ MACHINE OPERATION CODE.
+*  MT_CCTYPE = .F. 09, 8, **  $ CONDITION CODE TYPE.
+*  MT_MODR1  = .F. 17, 1, **  $ 'CHANGES INPUT REGISTER'


SIZE  MOPTAB(PS);  DIMS MOPTAB(NUM_MO);  $ OPERATION TABLE.
SIZE  MOPNAME(.SDS. 4); DIMS MOPNAME(NUM_MO);  $ NAMES.

$   MACRO TO INITIALIZE -MOPTAB-.
+*  MOP(VAL, CODE, CC, M1, NAME) =
    MOPNAME(VAL) = NAME:
    MOPTAB(VAL) = M1*4B'10000'+CC*4B'100'+CODE **

DATA    $ INITIALIZE TABLE.

$    MOP       CODE     CCTYPE      MR1   NAME
$    ---       ----     ------      ---   ----

MOP(MOP_BALR, 4B'05', MC_NOCHANGE, YES, 'BALR'):
MOP(MOP_BCTR, 4B'06', MC_NOCHANGE, YES, 'BCTR'):
MOP(MOP_BCR,  4B'07', MC_NOCHANGE, NO,   'BCR'):
MOP(MOP_LPR,  4B'10', MC_FULL,     YES,  'LPR'):
```

```
   MOP(MOP_LNR,  4B'11', MC_FULL,     YES,  'LNR'):
   MOP(MOP_LTR,  4B'12', MC_FULL,     NO,   'LTR'):
   MOP(MOP_LCR,  4B'13', MC_FULL,     YES,  'LCR'):
```

In conclusion, macros are quite powerful, and are of great assistance in the writing of well-structured programs. Indeed, much of the art of programming in LITTLE is in the design of systematic macros which clarify program structure and enhance portability.

INTRODUCTION TO CHARACTER STRINGS

The construction of portable software which uses character strings in an efficient manner remains a difficult problem. Although the desired primitives - input/output, concatenation, substring extraction and insertion - are quite standard, there is no obvious character string representation which is both portable and efficient. This appendix gives an informal introduction to the processing of character strings in LITTLE, and also describes the representation of character strings used by the standard LITTLE compiler.

The character string features may be summarized as follows:

1. Constants
    1. R constants for character codes.
    2. Quoted and Q constants for character strings.

2. Unary operator .SDS. I1 to determine size of character string of I1 characters.

3. Binary operators:
    1. S1 .IN. S2 to search character string S2 for instance of character string S1.
    2. S1 .PAD. I1 to pad character string S1 to length I1. S1 and I1 must be constants.
    3. S1 .SEQ. S2 to compare character strings for equality.
    4. S1 .SNE. S2 to compare character strings for inequality.

4. Access (extractor, assignment) qualifiers:
    1. .CH. I1, S1 to access I1-th character code of character string S1.
    2. .S. I1, I2, S1 to access substring of character string S1 which is I2 characters long and begins at position I1.
    3. .LEN. S1 to access length of character string S1.

5. Input/output formats:
    1. A format for character strings.
    2. R format for character codes.

6. Environment symbols:
    1. .CS. for size of character code.
    2. .SL. for size of character string length field.
    3. .SO. for size of character string origin field.

7. String search, case conversion and replacement procedures (described in section 9.6.4).

A character string is a sequence of characters. Each character has both an external graphic symbol and an internal bitstring code. The internal code has size CS. An R-constant specifies the internal code in terms of the external graphic symbol; for example, the value of the constant 1RA is the internal code of the character which has the letter A as its graphic symbol (01 on S66, 193 on S37). Character string constants are written using the common convention of enclosing the characters with the apostrophe character, using two successive apostrophes to represent an apostrophe within the string. Q constants simplify the writing of character string constants with internal apostrophes; for example, both 3Q''' and '''''''' represent the same character string constant.

INTRODUCTION TO CHARACTER STRINGS

Character strings are represented as bitstrings according to the following rules:

1. The size of the bitstring is a multiple of the machine word size WS.

2. The rightmost part of the bitstring contains two fields. The first field has size .SL. and gives the length of the character string in characters. The second field has size .SO. and gives the relative position of the start of the string. These fields are conventionally referenced by the macros

```
+* SLEN = .E. 01, .SL., **  $ STRING LENGTH.
+* SORG = .E. .SL.+1, .SO., **  $ STRING ORIGIN.
```

   The combined length of these fields (.SL.+.SO.) must be a multiple of the character size CS.

3. The character codes are arranged in the bitstring so that the extractor for the I-th character of S is

```
.F. (SORG S)-I*CS, CS, S
```

   The .F. extractor is used since the representation requires that characters not cross machine word boundaries. Note that .LEN. S is just an abbreviation of SLEN S, so that

```
.LEN. S  =  SLEN S  =  .E. 1, .SL., S
```

The representation rules constrain the values of the length and origin fields, and so permit a validity test to be performed on a bitstring used to represent a character string. The validity test is defined as follows (failure of a validity test causes abnormal program termination):

1. Let L be the value of .LEN. S. If L is zero, S has a valid form and represents the null string.

2. The value of (SORG S - 1) must be less than or equal to the size of S; otherwise, the validity test fails. The value of (SORG S) must be one more than a multiple of the word size; otherwise the validity test fails. The value of (SORG S) must be greater than the value of (.SL.+.SO.); otherwise the validity test fails.

3. The capacity C of S is defined as

```
(SORG S - 1)/(WS/CS) - (.SL.+.SO.)/CS
```

   and must be greater than or equal to the length L; otherwise the validity test fails. C is just the number of characters that can be correctly represented by S.

The standard compiler performs the validity test in the following situations:

1. In S1 .IN. S2, both S1 and S2 are validated.

2. In S1 !! S2, both S1 are S2 are validated.
3. In S1 .SEQ. S2, both S1 and S2 are validated.
4. In S1 .SNE. S2, both S1 and S2 are validated.
5. In S1 .PAD. I1, S1 is validated. Since S1 must be a character string constant, validation is done as part of compilation.
6. In .S. I1, I2, S1 = S2, both S1 and S2 are validated. Moreover, the index of the last character in the substring, (I1+I2-1), must not exceed the capacity of S1 and must not exceed the current length of S1. Substring assignment replaces existing characters, and cannot be used to extend the length of the target string.
7. In the extraction .S. I1, I2, S1, S1 is validated.
8. If S1 is PUT using the A format, S1 is validated.

The validity test detects most, but not all, of the attempts to operate on a bitstring which does not in fact represent a character string.

The representation rules do not define a unique representation, as two valid representations of the same string may have different origins. Moreover, the character string representation does not always specify the values of all the bits in the representing bitstring, but only specifies the values of the length and origin fields, and the values of the internal character codes. As a result, the bitstring comparison operators almost always return incorrect results when applied to character strings. LITTLE provides the binary operators .SEQ. and .SNE. to test for character string equality and inequality, respectively. As there is no standard ordering of character codes across different machines, LITTLE does not attempt to define the other comparison operators on character strings.

Certain actions completely define a character string in that they set both the origin and length fields. These operations always provide a valid representation with smallest origin. These operations are string concatenation (!!), the use of the A input format to read in a character string, the .S. substring extractor and the .PAD. operator.

The following demonstration program illustrates some of the points just mentioned. The program was run an IBM System/370, and uses the Monitor package to show the internal representations. This machine has WS of 32, PS of 24, CS of 8, .SL. of 16, and .SO. of 16.

```
1          +*  WS = .WS. **  +* PS = .PS. **  +* CS = .CS. **
2          TRACE STORES;  $ TRACE ALL STORES.
```

```
1          PROG START;
2          MONITOR BYTE;
3          SIZE  C1(CS), C2(CS*2), C4(CS*4), C8(CS*8);
4          SIZE  S1(.SDS. 1), S4(.SDS. 4), S8(.SDS. 8);
5          SIZE  I(PS);
6
7          C1 = 1RA;
8          C2 = 2RAB;
9          C4 = 0R/ABCD/;
10         S1 = 'A';  $ NOTE C1 AND S1 DIFFER.
```

```
 11          S4 = 'AB  ';  $ SHOULD BE 'AB'.PAD.4
 12          S8 = 'A    CD  ';
 13          S8 = 'COMPILER';
 14          .S. 5, 4, S8 = 'X';
 15          .CH. 1, S4 = 1RC;
 16          .S. 2, 2, S4 = 'OM';
 17          .S. 4, 1, S4 = 'P';
 18          S4 = S4;  $ TO FORCE TRACE LIST.
 19          S8 = S8;  $ TO FORCE TRACE LIST.
 20          I = S4 .SEQ. S8;  $ NOT EQUAL, AS LENGTHS DIFFER.
 21          .LEN. S8 = 4;
 22          I = S4 .SEQ. S8;
 23          I = (S4 .EQ. S8);  $ .EQ. IS NOT SAME AS .SEQ.
 24          I = 'MP' .IN. S4;
 25          END PROG START;

  7 :  C1 =   193 = 4B'000000C1'
  8 :  C2 = 49602 = 4B'0000C1C2'
  9 :  C4 = 4B'C1C2C3C4'
 10 :  S1 = 'A' = 4B'C1000000 00410001'
 11 :  S4 = 'AB  ' = 4B'C1C24040 00410004'
 12 :  S8 = 'A    CD  ' = 4B'C1404040 40C3C440 40000000 00810009'
 13 :  S8 = 'COMPILER' = 4B'C3D6D4D7 C9D3C5D9 00610008'
 14 : .S.    5,    4, S8 = 'X' = 4B'E7000000 00210001'
 15 : .CH.    1, S4 = 1RC = 4B'000000C3'

 16 : .S.    2,    2, S4 = 'OM' = 4B'D6D40000 00410002'
 17 : .S.    4,    1, S4 = 'P' = 4B'D7000000 00210001'
 18 :  S4 = 'COMP' = 4B'C3D6D4D7 00410004'
 19 :  S8 = 'COMPX   ' = 4B'C3D6D4D7 E7404040 00610008'
 20 :  I =     0 = 4B'00000000'
 21 : .LEN. S8 =     4 = 4B'00000004'
 22 :  I =     1 = 4B'00000001'
 23 :  I =     0 = 4B'00000000'
 24 :  I =     3 = 4B'00000003'
```

EXAMPLE - SORT PROCEDURES

```
$    THIS PROGRAM DEFINES TWO SORT ROUTINES AND A TEST ROUTINE.
$    ROUTINE BUBLSRT(A,B,N) IS THE WELL-KNOWN BUBBLE SORT, AS
$    DESCRIBED IN 'THE ART OF COMPUTER PROGRAMMING', VOL. 3,
$    PAGES 106-111.
$    ROUTINE HEAPSRT(A,B,N) IS THE HEAP SORT, CF. PP 145-149 OF KNUTH.
$    THE CODE FOR HEAPSRT IS BASED ON THAT GIVEN IN
$    'ON PROGRAMMING: INSTALLMENT 2' BY J. SCHWARTZ, P. 64.
$    THE MAIN ROUTINE START IS TEST DRIVER FOR SORTERS; IT USES DATA
$    GIVEN IN KNUTH, P. 75.
$    THE CODE CONTAINS CONDITIONAL TEXT, WITH NAME 'T'
$    WHICH MAY BE USED TO OBTAIN TRACE PRINTOUTS OF THE SORTERS
$    IN ACTION, AND TO AID DEBUGGING.
$            AUTHOR: D. SHIELDS  (CIMS)  REVISED 01 JUL 77
$
$    MACRO SECTION - DEFINE MACHINE PARAMETERS, I/O FUNCTIONS, AND
$        USEFUL CODING SEQUENCES.
     +* WS = 60 **  +* PS = 17 **  +* CS = 6 **  $ 6600 PARAMETERS
.+SET  T   $  COMPILE TRACING LINES.
     +* NTEST = 16 **  $ SIZE OF TEST ARRAY.

     +* PRINTIT(A, NA) =
        $ MACRO TO PRINT ARRAY, 4 COLUMNS PER ELEMENT
         PUT ,COLUMN(14) :A(1) TO A(NA),I(5),SKIP;
     **

     +*  TESTSORT(SORTER, SORTERNAME) =  $ TEST SORT PROCEDURE.
     PUT ,SKIP(2),'TEST SORT PROCEDURE: ' :SORTERNAME,A,SKIP;
     DO  I = 1 TO NTEST;  SORTED(I) = TEST(I);  END DO;
     PUT ,'BEFORE SORT: ';  PRINTIT(TEST, NTEST);  $ LIST TEST DATA.
     CALL SORTER(SORTED, NTEST);  $ TEST SORT.
     PUT ,'AFTER SORT:  ';  PRINTIT(SORTED, NTEST);PUT ,SKIP(2);
     **

     +*  SWAP(A,B) =  $ MACRO TO SWAP TWO ITEMS, A COMMON OPERATION
         SIZE  ZZZA(WS);  $ TEMPORARY FOR MACRO.
         ZZZA = A;  A = B;  B = ZZZA;  **

     PROG START;  $ MAIN PROGRAM AND TEST PROCEDURE.
     SIZE TEST(WS); DIMS TEST(NTEST);  $ KNUTH'S TEST DATA
     SIZE  SORTED(WS); DIMS SORTED(NTEST);
     DATA TEST = 503, 087, 512, 061, 908, 170, 897, 275,
         653, 426, 154, 509, 612, 677, 765, 703;
     SIZE  I(PS);  $ DO LOOP INDEX.
     PUT ,SKIP, 'TEST OF SORT ROUTINES ',SKIP(2);
     TESTSORT(BUBLSRT, 'BUBBLE SORT');
     TESTSORT(HEAPSRT, 'HEAP SORT');
     END;

     SUBR BUBLSRT(A, N);  $ BUBBLE SORT OF A.
     SIZE A(WS);
     DIMS A(2);  $ ARGUMENTS ARE ARRAYS (DIMENSIONS ARE DUMMY)
     SIZE  J(PS);  $ DO LOOP INDEX.
     SIZE  N(PS);  $  NUMBER OF ELEMENTS TO SORT.
     SIZE  T(PS);  $ POSITION OF LAST SWAP IN PASS THROUGH ARRAY.
     SIZE  BOUND(PS);  $ HIGHEST INDEX IN ORDER.
     SIZE  I(PS);  $ DO LOOP INDEX.
     BOUND = N;
```

EXAMPLE - SORT PROCEDURES

```
     WHILE BOUND>0;  $ LOOP WHILE POSSIBLY UNSORTED
         T = 0;
         DO  J = 1 TO BOUND-1;  $ BUBBLE ENTRY TO PROPER PLACE.
             IF  A(J) > A(J+1)  THEN    $ OUT OF ORDER, EXCHANGE.
                 SWAP(A(J), A(J+1));
                 T = J;  $ RECORD POINT OF EXCHANGE.
                 END IF;
             END DO;
         BOUND = T;  $ ELEMENTS ABOVE BOUND ARE SORTED.
.+T      PUT :BOUND,NI(5);  PRINTIT(A, N);  $ TRACE LIST.
         END WHILE;
     END SUBR BUBLSRT;


     SUBR HEAPSRT(A, N);  $ HEAPSORT OF A; FROM SETL NOTES.
$        THIS PROCEDURE IS A VARIANT OF HEAP SORT DUE TO
$        J. SCHWARTZ.
$        IN BRIEF, THIS IS A TREE SELECTION SORT IN WHICH ELEMENTS
$        2*I AND 2*I+1 ARE THE DESCENDANTS OF I.  WE BEGIN BY
$        TRANSFORMING THE ARRAY INTO A HEAP, WHERE A(1)...A(N) IS A
$        HEAP IF FOR 1 <= FLOOR(J/2) < J <= N, THEN
$            A( FLOOR(J/2) )  >=  A(J)  .
$        FOR A BINARY TREE, THIS MEANS THAT NO CHILD IS BIGGER THAN
$        A PARENT.  FIRST FORCE THE ARRAY TO HAVE THE HEAP
$        PROPERTY.  AS A RESULT A(1) WILL BE THE MAXIMAL ELEMENT.
$        THEN SWAP THE TOP ELEMENT WITH THE RIGHTMOST, AND
$        THEN REARRANGE THE TREE SO IT REMAINS A HEAP.
$        REPEAT THIS PROCESS UNTIL ALL ELEMENTS ARE SORTED.
$
     SIZE  A(WS);  DIMS A(2);  $ OUTPUT SORTED ARRAY.
     SIZE  I(PS);  $ DO LOOP INDEX.
     SIZE  N(PS);  $ NUMBER OF ELEMENTS TO SORT.
     SIZE  M(PS);  $ CURRENT NODE BEING EXAMINED.
     SIZE  TOP(PS);  $ CURRENT TOP OF TREE DURING PHASE TWO.
     SIZE  TARG(PS);  $ LARGEST CHILD.

.+T. PUT ,SKIP, 'HEAPSORT - FORCE HEAP.', SKIP;
     PUT :I :M,NI(3), COLUMN(14);
         DO I = 1 TO N; PUT :I,I(5);  END DO; PUT ,SKIP;
..T
     DO  I = 2 TO N;  $ MAKE INTO HEAP, I IS CURRENT PARENT
         M = I;
         WHILE  M > 1;  $ EXAMINE PARENTS IN TURN.
             IF  (A(M/2) >= A(M)) QUIT WHILE;  $ IF PARENT NO SMALLER,
                     $ IS HEAP.
             SWAP(A(M), A(M/2));  $ PROMOTE LARGE CHILD.
.+T          PUT :I:M,NI(3); PRINTIT(A, N)  $ TRACE LIST.
             M = M / 2;  $ MOVE TO GRANDPARENT.
             END WHILE;
         END DO I;
.+T  PUT ,SKIP ,' HEAPSORT - PHASE 2' ,SKIP;
     DO  TOP = N TO 2 BY -1;  $ SORT SUBTREES IN TURN.
         SWAP(A(1), A(TOP));  $ EXTRACT LARGEST ELEMENT.
         M = 1;  $  FORCE REMAINING SUBTREE TO BE HEAP.
         WHILE  M*2 < TOP;  $ FOR ALL SUBTREES.
                 $ PICK LARGEST CHILD OF NODE M IN SUBTREE.
             IF  (A(M*2) < A(M*2+1) ) & (M*2+1 < TOP)
```

EXAMPLE - SORT PROCEDURES

```
                THEN  TARG = M*2+1;
                ELSE  TARG = M*2;  END IF;
            IF  A(M) < A(TARG) THEN
                SWAP(A(M), A(TARG));  $ CHILD TOO BIG, EXCHANGE.
.+T         PUT :TOP:M:TARG,I(5); PRINTIT(A, N) $ TRACE LIST
                ELSE
                     $  QUIT SINCE BOTH CHILDREN IN RANGE, AND
                     $  KNOW THAT THEIR CHILDREN ALREADY IN ORDER..
                     QUIT WHILE M;
                END IF;
            M = TARG;  $ MOVE TO SUBTREE OF LARGEST CHILD.
            END WHILE;
        END DO TOP;
    END SUBR HEAPSRT;
```

EXAMPLE - TAUSWORTHE RANDOM NUMBER GENERATOR

```
    +* WS = .WS. **  +* PS = .PS. **  +* CS = .CS. **
    +*  RSZ = WS-1 **       $ SIZE OF RANDOM VALUE.
    PROG START;             $ TEST TAUSWORTHE RANDOM GENERATOR.
    SIZE  I(PS);            $ LOOP INDEX.
    SIZE  RV(PS);           $ RANDOM VALUE.
    SIZE  RANINT(RSZ);      $ RANDOM VALUE FUNCTION.
    SIZE  RANINTSEED(RSZ);    $ RANDOM SEED.
    RANINTSEED = .F. 1, 31, ATAN(1.0);
    PUT :RANINTSEED,NB(20,3,4) ,SKIP;
    PUT ,PAGE, 'TEST OF TAUSWORTHE GENERATOR.',SKIP;
    DO  I = 1 TO 100;
        RV = RANINT(100);
        PUT :RV,I(5);
        IF (MOD(I,10)=0)  PUT ,SKIP;
        END DO;
    END PROG START;
    FNCT RANINT(K);

    /* K IS AN ORDINARY INTEGER.  THE RESULT IS AN
    ORDINARY INTEGER FROM 0 TO K-1, UNIFORMLY DISTRIBUTED.
         THIS IS THE TAUSWORTHE GENERATOR FOR A 32 BIT MACHINE
    (SIGN AND 31 MAGNITUDE).  THE SEQUENCE IS OF MAXIMUM LENGTH
    FOR THE WORD SIZE USED.
         REFERENCES:

         1.  TAUSWORTHE, ROBERT C., MATHEMATICS OF COMPUTATION
             1965 PAGES 201-209.
         2.  WHITTLESLEY, J., CACM SEPTEMBER 1968 PAGES 641-644.
         3.  PAYNE, W. H., CACM JANUARY 1970 PAGE 57.

         THE ROUTINE USES PARAMETERS N AND M, WHICH ARE CHOSEN
    BASED ON THE MACHINE'S WORD SIZE.  ONE NORMALLY CHOOSES N
    EQUAL TO THE WORD SIZE LESS ONE, AND THEN M AS FOLLOWS:

              N       M
              --------------
              11   2
              15   1, 4, OR 7
              17   3, 5, OR 6
              23   5 OR 9
              31   3, 6, 7, OR 13
              63   1, 5, OR 31

         N AND M ARE EXPRESSED AS MACROS, SO THAT EFFICIENT
    CODE IS OBTAINED.
         THE CALCULATION OF SUITABLE VALUES OF M FOR A GIVEN N
    INVOLVES FINDING PRIMITIVE POLYNOMIALS; SEE (1) PAGE 208.
         THIS ROUTINE USES A GLOBAL VARIABLE 'RANINTSEED'
      WHICH MUST BE INITIALIZED TO A BITSTRING,
    NOT ALL ZEROS, OF LENGTH N.

    AUTHOR:  D. SHIELDS  (CIMS)  01 JAN 77.

        THIS IS A REWRITE OF A VERSION WRITTEN
        IN SETL BY HENRY S. WARREN, JR.

    */
```

EXAMPLE - TAUSWORTHE RANDOM NUMBER GENERATOR

```
    +* N = 31 **
    +* M = 13 **
    SIZE  RANINT(N);   $ SIZE NO MORE THAN N.
    SIZE  A(WS), B(WS);  $ TEMPORARIES DURING GENERATION
    SIZE  K(WS);   $  RANGE IN WHICH RANINT MUST FALL.

    SIZE  FBK(PS);    $ MAGNITUDE OF K.
    $ FIRST UPDATE THE VALUE OF 'RANINTSEED'.

    ASSERT K>0;
    FBK = .FB. K;  $ MAGNITUDE OF K.
    ASSERT FBK <= N;
    UNTIL  RANINT <= K;   $ ITERATE TILL GET NUMBER IN RANGE.
        B = .F. M+1, N-M, RANINTSEED;  $ RIGHT SHIFT M
        A = RANINTSEED .EXOR. B;
        .F. N-M+1, M, B = .F. 1, M, A;
        .F. 1, N-M, B = 0;     $   LEFT SHIFT N-M.
        RANINTSEED = A .EXOR. B;

        /* NOW CONVERT 'RANINTSEED' TO AN INTEGER RANGING FROM 0 TO
        K-1, WHERE K < 2 EXP. N.  THIS IS DONE BY TRUNCATING
        'RANINTSEED' TO THE APPROPRIATE NUMBER OF BITS.  IF THE RESULT
        IS LESS THAN K, IT IS RETURNED.  OTHERWISE THE ROUTINE STARTS
        ALL OVER AGAIN.  THIS GUARANTEES A UNIFORMLY DISTRIBUTED
        RESULT.  THE PROCESS MUST TERMINATE, AS 'RANINTSEED' IS OF
        MAXIMUM PERIOD (2 EXP. N - 1). */

        RANINT = .F. (N+1) - FBK, FBK, RANINTSEED;  $ TAKE LEFT BITS.
        END UNTIL;
    END FNCT RANINT;
```

EXAMPLE - LTLDOC: A SIMPLE FORMATTING PROGRAM.

```
/*              LTLDOC - LIST LITTLE DOCUMENT.

    INPUT CONSISTS OF LINES WITH CONTROL CHARACTERS IN
    THE FIRST TWO COLUMNS, AND TEXT IN THE REMAINING SEVENTY COLUMNS.
    CONTROL CHARACTERS ARE AS FOLLOWS:

    D - DOCUMENT: INITIALIZE. SHOULD BE FIRST CONTROL LINE.
    E - EJECT: SET EJECT FLAG, DO NOT LIST TEXT.
    P - PAGE: SET EJECT FLAG, LIST TEXT.
    S - SUBTITLE: USE TEXT TO DEFINE SUBTITLE, SET EJECT FLAG.
    T - TITLE: USE TEXT TO DEFINE MAIN TITLE, SET EJECT FLAG.
    U - UNDERLINE: LIST TEXT, THEN UNDERLINE IT.
    0 - SKIP LINE BEFORE LISTING TEXT.
    1 - SAME AS P.

    LTLDOC INCREASES THE PAGE AND LINE LIMITS TO PERMIT
    UP TO 500 PAGES.

    AUTHOR:  DAVID SHIELDS  (CIMS)  11 JAN 77.
*/

    $   STANDARD MACROS.
    +* WS = .WS. **  +* PS = .PS. **  +* CS = .CS. **
    +* YES = 1 **  +* NO = 0 **

    $   PROGRAM PARAMETERS.
    +*  NLB = 4  **  $ NUMBER OF LEADING BLANKS IN LINE.
    +*  IBL = (''.PAD. NLB) **  $ INITIAL BLANK STRING.

    PROG LTLDOC;            $ LIST LITTLE DOCUMENT.

    SIZE  LINESPERPAGE(PS); $ LINES PER PAGE.
    SIZE  DOTEXT(1);        $ ON TO LIST TEXT OF LINE.
    SIZE  EJECTING(1);      $ ON TO BEGIN NEW PAGE WITH NEXT TEXT
                            $ LINE.
    SIZE  UNDERLINING(1);   $ ON TO UNDERLINE TEXT.
    $   FIRSTNB AND LASTNB DELIMIT TEXT FOR UNDERLINE OPTION.
    SIZE  FIRSTNB(PS), LASTNB(PS);
    SIZE  I(PS);            $ LOOP INDEX.
    SIZE  C1(CS), C2(CS);   $ FIRST TWO CHARACTERS IN LINE.
    SIZE  TEXT(.SDS. 70);   $ TEXT LINE.
    SIZE  SKIPCOUNT(PS);    $ SKIP COUNT.

    $   GET NUMBER OF LINES PER PAGE.
    CALL CONTLPR(10, LINESPERPAGE);

    $   SET LARGE PAGE AND LINE LIMITS.
    CALL CONTLPR(21, 500);  $ UP TO 500 PAGES.
    CALL CONTLPR(19, 500*LINESPERPAGE);

    CALL DOCINI;            $ INITIALIZE FOR NEW DOCUMENT.

    WHILE 1;
        GET ,SKIP :C1 :C2,R(1) :TEXT,A(70);
        IF  (FILESTAT(1,END))  QUIT WHILE;

        IF  C1 = 1R  THEN  $ IF BLANK, LIST TEXT.
```

EXAMPLE - LTLDOC: A SIMPLE FORMATTING PROGRAM.

```
        DOTEXT = YES;

    ELSEIF  C1 = 1RD  THEN  $ IF NEW DOCUMENT
        CALL DOCINI;
        DOTEXT = NO;

    ELSEIF  C1 = 1RE  THEN  $ IF EJECT REQUEST.
        DOTEXT = NO;  EJECTING = YES;

    ELSEIF  C1=1RP ! C1=1R1  THEN  $ IF NEW PAGE.
        DOTEXT = YES;  EJECTING = YES;

    ELSEIF  C1 = 1RS  THEN  $ IF SUBTITLE DEFINITION.
        DOTEXT = NO;  EJECTING = YES;
        CALL STITLR(1, IBL!!TEXT);  $ ENTER SUBTITLE.

    ELSEIF  C1 = 1RT  THEN  $ IF MAIN TITLE DEFINITION
        DOTEXT = NO;  EJECTING = YES;
        CALL STITLR(0, IBL!!TEXT);  $ ENTER MAIN TITLE.
        CALL STITLR(1, '');  $ CLEAR SUBTITLE.

    ELSEIF  C1 = 1RU  THEN  $ IF UNDERLINE REQUEST.
        DOTEXT = YES;  UNDERLINING = YES;

    ELSEIF  C1 = 1R0  THEN  $ IF SKIP LINE REQUEST.
        DOTEXT = YES;  SKIPCOUNT = 1;

    ELSE  DOTEXT = YES;
        END IF;

    IF  (DOTEXT=NO)  CONT WHILE;

    IF  EJECTING  THEN  $ IF STARTING NEW PAGE.
        PUT ,PAGE;
        EJECTING = NO;
        END IF;

    IF  SKIPCOUNT  THEN $ IF SKIPPING LINES BEFORE TEXT.
        PUT ,SKIP(SKIPCOUNT);  SKIPCOUNT = 0;  END IF;

    PUT ,X(NLB) :TEXT,A ,SKIP;
    DOTEXT = YES;

    IF  UNDERLINING  THEN
        UNDERLINING = NO;
        IF  (TEXT .SEQ. (''.PAD.70))  CONT WHILE;
        CALL CONTLPR(5, 3);  $ NEED AT LEAST THREE LINES.
        FIRSTNB = 1;    $ FIND FIRST, LAST NON BLANKS.
        WHILE  .CH. FIRSTNB, TEXT = 1R ;
            FIRSTNB = FIRSTNB + 1;  END WHILE;
        LASTNB = 70;
        WHILE  .CH. LASTNB, TEXT = 1R ;
            LASTNB = LASTNB - 1;  END WHILE;
        TEXT = '' .PAD. 70;
        DO  I = FIRSTNB TO LASTNB;  .CH. I, TEXT = 1R-;  END DO;
        PUT ,X(NLB) :TEXT,A,SKIP;
        END IF;
```

```
        END WHILE;
    END PROG LTLDOC;
    SUBR DOCINI;            $ INITIALIZE FOR NEW DOCUMENT.

    CALL CONTLPR(6, 1);     $ ENABLE PAGING.
    CALL CONTLPR(7,1);      $ ENABLE TITLING, CLEAR MAIN, SUBTITLES.
    CALL ETITLR(0, 'PAGE', 67, 0);  $ ENTER 'PAGE' FIELD.
    CALL CONTLPR(8, 71);    $ SET PAGE FIELD.
    CALL CONTLPR(9, 0);     $ CLEAR DATE FIELD.
    CALL CONTLPR(13,0);     $ SET INITIAL PAGE NUMBER.
    CALL CONTLPR(2,2);      $ SET INITIAL LINE POSITION.

    DOTEXT = NO;  EJECTING = YES;  UNDERLINING = NO;
    SKIPCOUNT = 0;
    END SUBR DOCINI;
```

CONTENTS

CONTENTS

/* FINIS */