# Implementing software using the LOWL language

Fourth Edition

April 2004

P.J. Brown, R.D. Eager

# Table of Contents

# Preface to the Third Edition

LOWL has remained almost unchanged for some time now, and hence this Edition contains few alterations to the previous one. The only significant change is the reduction of some statement names from six characters to five or fewer. The only significant addition is the optional facility for aligned data. The specification of LOWL given here corresponds exactly to that given in the book *Macro processors and techniques for portable software*, as published by J. Wiley and Sons in 1974.

# Preface to the Fourth Edition

This Edition has been rewritten in Texinfo, so that it can be published in both printed and machine readable form; this has necessitated some re-wording and re-ordering of the text. The only other substantive change is the removal of mention of paper tape and lineprinter listings as a distribution medium, and the deletion of the table of paper tape codes.

# 1 Introduction

LOWL is a language which can be used to implement portable software. Several pieces of software have been encoded in LOWL, including:

a. *ALGEBRA*, a program for teaching and research in Boolean Algebra and symbolic logic.

b. *UNRAVEL*, a program for putting intelligibility into core dumps.

c. *ML/I*, a macro processor.

d. *SCAN*, a simple conversational language for text analysis.

LOWL consists of a *kernel* of features that are needed by all or almost all of the software that has been encoded in it, plus some *extensions* specially tailored to each piece of software. This manual describes the kernel of LOWL. Each piece of software has its own Supplement, which describes the extensions it needs.

A program in LOWL can be regarded as a sequence of machine-independent macro calls. If an implementation is required for a given machine, each macro is defined as a series of one or more assembly language statements for that machine. The program in LOWL is fed to a macro processor and the effect of the macros will be to map the program into the assembly language of the required machine, thus implementing the program on the machine.

The macros are called *mapping macros* and the machine for which an implementation is required is called the *object machine*. To quote a simple example of a mapping macro, one LOWL statement has the form:

```
        AAV     V
```

where *V* is a variable name. This statement means *A*dd to the *A* register a *V*ariable. On the object machine the equivalent instruction might be `ADD`, so the mapping macro would map the above statement into:

```
        ADD     V
```

The following piece of code should give the reader a flavour of LOWL, though the individual statements will not be fully understood until later:

```
        NB      'Sample LOWL statements'
        LAV     SIZE,X
        CAL     6
        GOGR    BIG,2,X,X
        BUMP    PTR,OF(LCH)
        MESS    'Size is six or less'
[BIG]   GOSUB   CHECK,236
        LAI     PTR,X
        FSTK
        LAV     NUPT,X
        AAV     SIZE
        STV     PTR,X
```

Unfortunately little, if any, software is totally machine-independent. There are always a few routines that depend very closely on the structure of the object machine, on its operating system or even on the way it encodes characters. Thus all software that is to be made portable through LOWL is divided into two parts as follows:

a. the *MI-logic*. The machine-independent parts that are encoded in LOWL.
b. the *MD-logic*. The machine-dependent parts that need to be coded by hand for each implementation.

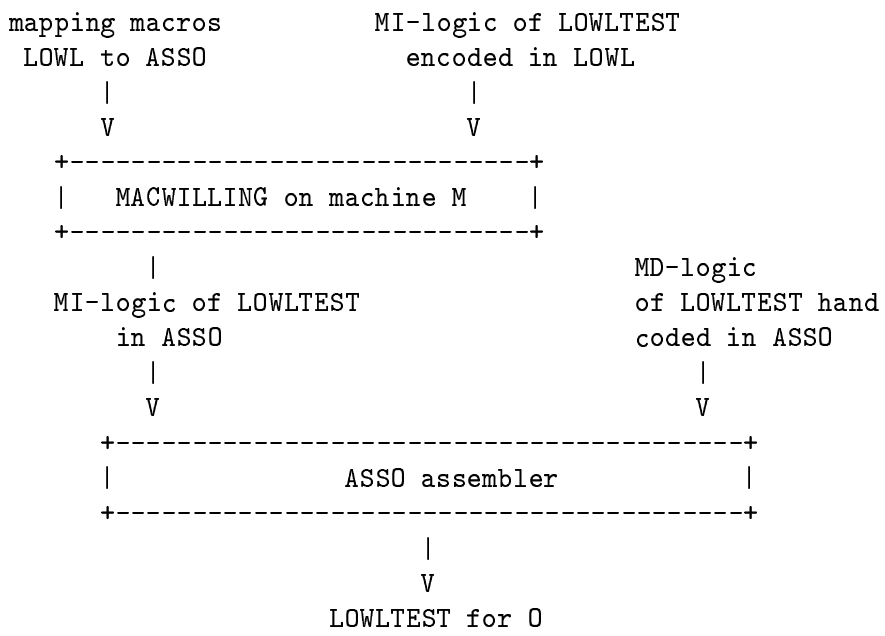
Fortunately the MD-logic is usually smaller than the MI-logic by a factor of twenty or more, and so the amount of hand-coding is relatively small. Each piece of software encoded in LOWL has its own MD-logic, and this is described in the appropriate Supplement.

## 1.1 The implementation procedure

LOWL has been designed so that it is simple enough to be mappable by almost any macro processor, including a macro-assembler, and, in addition, by many string processing languages. It may be necessary to write a simple pre-pass to change characters or formats; for example, the square brackets that surround labels may need to be changed or removed. This is discussed in detail in Chapter 2 [The kernel of LOWL], page 6.

To illustrate how a program is mapped we will assume that software S, which is encoded in LOWL, is to be implemented on the object machine O, which has an assembler called ASSO. The first job of the implementor is to select a suitable mapping tool to map LOWL into ASSO. We will assume that a macro processor called MACWILLING, that runs on machine M, is selected. Part of the flexibility of this technique is that any suitable machine may be selected as M, and in particular M need not be the same as O. In general, however, if a suitable mapping tool is available on O it is best to select this. It is best of all if the macro-assembler for O is able to act as MACWILLING. Several mappings have been done this way and have been accomplished very quickly.

After selecting MACWILLING, the next job is to write mapping macros to map LOWL into ASSO. These need to be tested, and for this purpose a test program called LOWLTEST is available. This is encoded in LOWL, and tests all the statements in the kernel of LOWL (a full description of LOWLTEST appears in Chapter 4 [The LOWL kernel test program], page 30. The procedure for running LOWLTEST is illustrated by the following diagram.




LOWLTEST is then run on machine O, and it will either work or produce error messages. In the latter case, the mapping macros need to be corrected and the process repeated.

When LOWLTEST is working, work can start in earnest on implementing the software S itself. The mapping process is exactly that illustrated in the above diagram but with S in place of LOWLTEST. In particular the MD-logic of S needs to be coded in ASSO. When S has been mapped it needs to be tested and this usually requires some sets of test data. Although LOWLTEST will have paved the way, there may still remain bugs at this stage. These may be due to errors in the MD-logic, in the mapping macros for the extensions to LOWL required by S, or even in in the mapping macros for the kernel of LOWL. Although LOWLTEST will have tested the latter, no test program can cover every possible combination of statements.

In summary, therefore, the implementation process proceeds as follows:

a. If necessary, write a pre-pass to convert LOWL programs to a suitable format and character set.

b. Write mapping macros for LOWL.

c. Test these using LOWLTEST.

d. Encode the MD-logic of S.

e. Map and test S.

This process is called a *DLIMP*, which stands for *D*escriptive *L*anguage *IM*plemented by *M*acro *P*rocessor. LOWL is a *descriptive language*, being a language specially designed for describing certain pieces of software. The merit of a DLIMP is that portable software can be implemented relatively easily and, most important, efficiently. A paper in *Computer Journal* **12**, 4 (Nov. 1969), pp. 327-331 discusses DLIMPs in more detail, as do various other papers. However, these mostly consider a descriptive language called L, which is set at a much higher level than LOWL. One paper which discusses the relative merits of the LOWL language appears in *Communications of the ACM* **15**, 11 (Dec. 1972), pp. 1059-

1062. Note that some of the details of LOWL described in the latter paper have since been changed.

An implementation of software via a DLIMP may take anything from a few days to several weeks. Variable factors are the experience of the implementor, the quality of ASSO and MACWILLING, the size of the MD-logic and the LOWL extensions, machine availability, and, above all, whether the idea is to get something working quickly or to do a proper professional job.

## 1.2 Distributing software in LOWL

Originally, software in LOWL was distributed as paper listings, and on paper tape; this was due to widely differing machine character codes.

Now that ASCII is understood by virtually every system (even if it is not always the default character code), LOWL software is distributed in ASCII, as files on magnetic media or (more usually) from the ML/I web site (`http://www.ml1.org.uk`).

# 2 The kernel of LOWL

This Chapter gives a complete description of the kernel of LOWL.

## 2.1 Statement formats

Statements in LOWL are written one to a line, and each consists of a mnemonic operation code followed by a number of arguments separated by commas. Operation codes are represented by names that are sequences of up to five letters, and argument lists consist of no more than fifty characters. Each operation code has a fixed number of arguments, some operation codes having no arguments at all. Arguments that are literal character strings (e.g. text to be printed, comments) are enclosed in quotes. No argument is ever null; instead the letter `X` is often used to indicate that a certain argument is not applicable in a given case. The operation code is preceded by a tab character and, if it has any arguments, it is also followed by a tab (these tabs may be replaced by spaces if the implementor thinks this is more convenient). Statements may optionally be preceded by a *label*, which is an identifier of up to six characters enclosed in square brackets occurring at the very start of a line. Blank lines are used to improve layout.

The following are examples of LOWL statements:

```
          NB      'These are LOWL statements'

[ENDST]   LBV     IDPT

          BMOVE

          SUBR    CHEKID,X,1

          MESS    'Error - - stack overflow'
```

Extensions to LOWL follow the same form as statements in the kernel.

The LOWL statement format has been found acceptable to most macro processors. The fact that each type of statement has a fixed number of arguments, and no argument is ever null, helps considerably. There are, however, usually a few problems of which the following are typical:

a. The square brackets round labels are unacceptable. Since label names are usually unaltered on a mapping it is often sufficient simply to remove the square brackets on a pre-pass.

b. Arguments enclosed in quotes cause problems, particularly when the argument itself involves a comma, for example:

```
MESS    'Type your name, age and sex'
```

   Here the entire text in quotes is a single argument, not two arguments separated by commas. Solutions to this problem vary according to the macro processor used.

c. In general each macro call occupies a whole line, but there exists one possible exception to this. Certain constants which occur as arguments to other macros may themselves need to be mapped as macros. For example, in:

```
        CCN     NLREP
```

NLREP stands for the internal code for newline, and in:

```
        LAL     OF(2*LNM+LCH)
```

the argument is the value of twice the length of a number (LNM) plus the length of a character (LCH). (See the full description of the OF macro later.) Fortunately most assemblers have their own mechanism for dealing with such named constants, even when addition and multiplication are involved, so this problem can be solved without the use of macros.

## 2.2 A possible pre-pass algorithm

Because of the type of problem described above, it may be necessary to make certain systematic replacements of characters and symbols in order to make LOWL suitable for MACWILLING. This will apply particularly if MACWILLING is a macro-assembler, since these tend to be inflexible about the formats they will accept. Almost certainly the syntax of labels would require changing, and quite likely the OF macro as well. Most machines possess editors or string manipulation languages that are capable of replacing every occurrence of one string by another one. If such software is available, the implementor need not read the rest of this Section. If it is not, it will be necessary to write a special pre-pass program to make the replacements. Such programs should be trivial in nature, but "trivial" programs often take a long time to get right, and "ten-minute jobs" become ten-hour jobs. Because of this, a possible form of a pre-pass algorithm is shown below. It has been tested in practice and its use may save the implementor a few debugging runs. The algorithm is expressed using a few elementary operations that should be readily convertible into whatever high-level language or assembly language is being used for encoding the pre-pass.

```
            comment Possible form of a program to adapt
            character representations, etc., in LOWL to local needs;

            Set OFSW, QSW = 0;

[LOOP]      Input a character into CHAR, stopping if end of data.

[TEST]      comment Test for unique individual characters to be
            changed, e.g.:;
            If CHAR is '[' then go to LBRAC;
            If CHAR is '<' then go to LESS;
            If CHAR is a tab then go to TAB;
            etc.

            comment To test for quotes, distinguishing opening
            quotes from closing ones;
            If CHAR is not a quote then go to TRYOF;
            Set QSW = 1 - QSW;
            If QSW = 1 then go to OPENQ;
            Go to CLOSEQ;
```

```
[TRYOF] comment To recognise the OF macro and its closing
        parenthesis (it is assumed that the argument of the OF macro
        is to remain unchanged);
        If CHAR is not the letter 'O' then go to TRYRP;
        Input a character into CHAR;
        If CHAR is not 'F' then output 'O' and go to TEST;
        Input a character into CHAR;
        If CHAR is not '(' then output 'O', output 'F' and go to TEST;
        Set OFSW = 1 and go to OFMAC;

[TRYRP] If CHAR is ')' and OFSW is 1 then set OFSW to zero
        and go to ENDOF;

        comment Finally, for characters that do not need
        changing;
        Output CHAR and go to LOOP;

        comment At labels LBRAC, LESS, TAB, OPENQ, CLOSEQ, OFMAC
        and ENDOF, output whatever is to replace the given characters
        and return to LOOP;
```

The nature of test data to verify that the algorithm works will depend on what is being changed. The following might be some useful test cases.

```
[LABEL] LAV     ABC,X
        LAM     OF(LCH+LNM)
        LAL     OF(2*LNM)
        CCL     ')'
        CCL     '<'
[LAB2]  MESS    'Error(s)'
        NB      'Ends in O'
        NB      'Ends in OF'
        CCL     '('
```

## 2.3 Supplementary arguments

Some LOWL statements have what are called *supplementary arguments*. These are used to convey extra information about the statement, which may be used on some mappings. For example a branching statement might be written:

```
        GO      PIG,130,E,X
```

This means branch to label `PIG`. There are three supplementary arguments. The first of these says that the label `PIG` is 130 LOWL statements after the current statement. The `E` means that the branch jumps out of a subroutine, and the final `X` means that it is not a special case (e.g. it is not part of an array of jumps forming a multi-way switch). Most mappings of LOWL would ignore all three supplementary arguments.

## 2.4 Character set

The character set used in the kernel of LOWL consists of:

a. The upper case letters 'A' to 'Z'.

b. The lower case letters 'a' to 'z'.

c. The digits '0' to '9'.

d. The following punctuation characters:

```
, + - * / : ' > < [ ] ( ) = $
```

as well as space and tab, the last two being used for layout purposes. Note that `$` is used to signify newline in messages.

## 2.5 Data types

There are two data types: character and numerical. An item of character data can be any single character in the character set for the implementation; an item of numerical data is an integer or an address. The implementor chooses how these should be represented on the object machine. On many machines both data types will be chosen to correspond to a word and there will be no need to differentiate them. There will be some space savings if character data can be stored in a smaller unit of storage than a word, but this should be done only if the smaller unit is directly addressable (i.e. if no packing, unpacking or masking is necessary when characters are manipulated). Some object machines may require alignment of data; this is discussed at the end of this Chapter.

The character code in which the object software is to work may be chosen by the implementor. Most machines have their own preferred internal code, but if there is any choice in the matter it is best to select a code such that the three classes

a. letters

b. digits

c. others

can be simply and quickly differentiated.

Operations on numerical data always yield integer results and hence there are no floating point operations. No allowance has been made in LOWL for integer overflow and the best action is to ignore it if it occurs.

When an item of numerical data is an address it may be that of a variable, a table item or an item on the stacks (see later); alternatively it might have value zero, which means "null" or "end of list". Hence data must be located so that nothing can have zero as its real address.

The terms "address" and "pointer" are used synonymously in LOWL documentation.

## 2.6 Constants

Several LOWL statements have constants as arguments. Constants may be numerical constants or character constants.

Numerical constants are represented as decimal integers or by a call of the OF macro. Before describing the OF macro it is necessary to define its sub-macros, which are machine-dependent constants that define how data is represented on the object machine. They are as follows:

LCH the number of storage units occupied by an item of character data.

LNM the number of storage units occupied by an item of numerical data.

LICH = 1/LCH.

(Another way of looking at these is that LCH and LNM are the amounts a stack pointer would be increased to stack items of character and numerical data, respectively.) On a word machine LCH and LNM would both be one; on a byte machine LCH might be one and LNM four. Some extensions of LOWL use extra sub-macros.

LICH is only used in the context:

```
MULTL   OF(LICH)
```

which means multiply by LICH. In most implementations LCH and LICH will both be one, but if LCH is greater than one then LICH will not be an integer. The problem is best solved by turning multiplication by LICH into division by LCH, thus eliminating LICH altogether.

The OF macro takes the form:

OF(*argument*)

where *argument* is one of the following:

a. *N***S*+*S*

b. *N***S*-*S*

c. *N***S*

d. *S*+*S*

e. *S*-*S*

f. *S*

Here *N* stands for any positive integer, *S* for any of the submacros, and an asterisk represents multiplication. Examples of the OF macro are therefore:

a. OF(3*LNM+LCH)

b. OF(LNM-LCH)

c. OF(LCH)

and an example of its use within a statement is:

```
BUMP    STAKPT,OF(LNM+LCH)
```

The result of the OF macro will never be negative, assuming that LNM is not less than LCH.

If it happens that ML/I is the macro processor being used to effect the mapping, then the mapping macros for OF might take the following form:

```
MCDEF LNM AS 4
MCDEF LCH AS 1
MCDEF LICH AS 1
MCDEF OF WITHS ( ) AS <%%A1..>
```

In ML/I, `%A1.` inserts argument one whereas a `% .` pair on their own evaluate an arithmetic expression. If the argument was, for example, `LNM+2*LCH` then the replacement text of `OF` would be equivalent to `%4+2*1.` — which would yield the result `6`.

There is a facility for "manifest" numerical constants, i.e. numerical constants represented by names. The purpose of manifest constants is simply to make programs easier to understand. For example, if the code 4 were being used to mean "black", then the statement

```
CAL     BLACK
```

is easier to understand than

```
CAL     4
```

In the former case, the name `BLACK` needs to be declared to be identical to `4` and this is done by the `IDENT` statement, which has the form:

```
IDENT   name,decimal integer constant
```

for example:

```
IDENT   BLACK,4
```

`IDENT` statements occur within the declarative statements at the start of a program, and always come before any usage of the name being defined. When a mapping is performed, `IDENT` statements may either be made to perform an explicit replacement of the name by its value, or, perhaps better, to map into a directive in the object assembly language that accomplishes the same effect.

Whether generated by the `OF` macro or not, almost all numerical constants are small. Few pieces of software encoded in LOWL contain constants larger than 100.

A character constant in the kernel of LOWL is represented by a literal character within quotes (e.g. `'P'`, `'+'`) or, for some special characters, by a name. In the latter case the name is one of the following:

`NLREP` meaning newline (i.e. the character noting the end of a line).

`SPREP` meaning space.

`TABREP` meaning tab.

`QUTREP` meaning a double-quote sign (").

(In addition, some further names are used in certain LOWL extensions.) Values corresponding to suitable internal codes for the object machine should be assigned to these names either by means of macro mapping, or, better, by '`EQUATE`' statements in the object machine's assembler (such '`EQUATE`' statements might also profitably be used to supply values for `LNM`, `LCH` and `LICH`, thus adding flexibility to the final object code).

## 2.7 Registers

Almost all statements in LOWL involve at most one storage address, and all assignments, comparisons and arithmetic operations are done via registers. There are notionally three registers, as follows:

`A` is the numerical accumulator.

`B` is the index register.

`C` is the character register.

However, no two of these registers are ever used simultaneously and it is possible to represent all three by the same physical register. The only merit in making them different is a gain in efficiency. In particular, the `LAM` and `LCM` statements (see later) may be faster if B is different from A and C, and, on implementations where character and numerical data is different, character operations may be faster if C is different from A (on one LOWL implementation C was dispensed with altogether, and all character operations were performed by storage-to-storage instructions). Since the registers are never used simultaneously, any operation on one register may clobber any of the others. Moreover, most of the LOWL statements that do not explicitly set registers may use the registers as workspace. The table at the end of this Chapter gives a list of such statements.

All statements that use the A or B registers have numerical operands, and all statements that use the C register have a single character as operand.

## 2.8 Names and scope

The names of all variables, labels, constants and subroutines consist of an identifier (i.e. a letter followed by a sequence of letters and/or digits) of up to six characters, e.g. `SUM`, `BC3`, `STKARG`, `GL13`. No names have local meanings; the meaning is always global to the entire MI-logic.

## 2.9 Variables

All variables are numerical. Each variable is declared by one of the following statements:

```
DCL     name

EQU     name1,name2
```

The `EQU` statement means that *name1* can share the same storage as the previously declared variable called *name2*. It is not, however, imperative that the two be made the same and `EQU` may be treated as if it had been

```
DCL     name1
```

if this is easier to map.

Variables do not need to be given any special initial values.

Ambitious implementors for object machines with some spare registers may choose to maintain some variables in registers rather than in storage. However, some blocks of variables need to be stored contiguously, either because they are subjected to block moves or to indexing instructions, and none of these variables should be placed in registers (unless all variables can be placed in registers). If such blocks exist, the relevant declarations are enclosed between the comments:

```
NB      'THE FOLLOWING MUST BE STORED CONTIGUOUSLY'
```

and

```
NB      'END OF CONTIGUOUS BLOCK OF VARIABLES'
```

## 2.10 Table items

Most of the software encoded in LOWL requires certain fixed tables. There is a set of statements in the kernel of LOWL for defining table items. These statements may be labelled, in the same way as program statements may be labelled.

Table items are never changed. They may therefore be placed in read-only storage, and in a multi-access environment the same tables may be shared by all users.

All table items should be stored contiguously, in the order in which they are declared. Table items are always addressed indirectly by means of pointers (see the `LAA` statement).

## 2.11 Statements

We are now ready to enumerate the statements that make up the kernel of LOWL. The following notation is used to describe arguments to statements.

| | | |
|---|---|---|
| a. | *V* | means a variable name. |
| b. | *N* | means a non-negative decimal integer constant (possibly represented by a name). |
| c. | *OF* | means a call of the `OF` macro. |
| d. | *N-OF* | means either *N* or *OF*. |
| e. | *table label* | means a label attached to a table item. |
| f. | *charname* | means one of the names representing literal character constants (e.g. `NLREP`). |
| g. | *character* | means a single character. |
| h. | *characters* | means a string of one or more characters. |
| i. | (*A*)<br>(*B*) | means either *A* or *B*. |

### 2.11.1 Statements for defining table items

The following are the statements used for defining table items:

| | | |
|---|---|---|
| `CON` | `( N-OF)`<br>`(-N-OF)` | defines an item consisting of the single numerical constant represented by the argument. A minus sign indicates a negative constant. |
| `NCH` | `charname` | defines an item consisting of the single character named by the argument. |
| `STR` | `'characters'` | defines an item consisting of the string of characters within the quote signs (this string must be represented by one character per data storage unit, e.g. if character data is stored in words then the string should be represented one character to a word). |

### 2.11.2 Load statements

Three of the load statements shown below have a subsidiary argument that can be `R` or `X`. In these cases `R` means that the load instruction is redundant if compare statements and the conditional branching statements `GOEQ`, `GONE`, `GOGE`, `GOGR`, `GOLE`, `GOLT` do not clobber the register being loaded, e.g.

```
LAV     ABC,X
CAL     6
GOGR    LAB3,...
LAV     ABC,R
```

The complete list of load statements is as follows:

| | | |
|---|---|---|
| `LAV` | *`V`*`,(R)`<br>`(X)` | Load A with value of *`V`*. |
| `LBV` | *`V`* | Load B with value of *V*. |
| `LAL` | *`N-OF`* | Load A with literal value *N-OF*. |
| `LCN` | *`charname`* | Load C with literal named character. |
| `LAM` | *`N-OF`* | Derive the pointer given by adding *N-OF* to the contents of B, and load A with the value pointed at by this (i.e. load A modified). |
| `LCM` | *`N-OF`* | As `LAM`, but load a character into C. |
| `LAI` | *`V`*`,(R)`<br>`(X)` | Load A with value pointed at by *V*. |
| `LCI` | *`V`*`,(R)`<br>`(X)` | Load C with character pointed at by *V*. |
| `LAA` | *`V`*`,D` | Load A with the address of variable *V*. |
| `LAA` | *`table label`*`,C` | Load A with the address of the table label (in most implementations this will be identical to the preceding statement). |

### 2.11.3 Store statements

The complete set of store statements is as follows:

| | | |
|---|---|---|
| `STV` | *`V`*`,(P)`<br>`(X)` | Store A in *V*. |
| `STI` | *`V`*`,(P)`<br>`(X)` | Store A in address pointed at by *V*. |
| `CLEAR` | *`V`* | Set value of *V* to zero (this may clobber A). |

In the first two cases, the second argument specifies whether A needs to be preserved: P means A must be preserved; X means it need not be (on a two-address machine it is possible to implement a load statement followed by a store statement without passing through A, provided that A does not need to be preserved after the store statement).

### 2.11.4 Arithmetic and logical statements

The complete set of arithmetic and logical statements is as follows:

| | | |
|---|---|---|
| AAV | *V* | Add value of *V* to A. |
| ABV | *V* | Add value of *V* to B. |
| AAL | *N-OF* | Add literal value *N-OF* to A. |
| SAV | *V* | Subtract value of *V* from A. |
| SBV | *V* | Subtract value of *V* from B. |
| SAL | *N-OF* | Subtract literal value *N-OF* from A. |
| SBL | *N-OF* | Subtract literal value *N-OF* from B. |
| MULTL | *N-OF* | Multiply A by literal value *N-OF* (which might be one). In no case is the value *N-OF* very large and multiplication can, if desired, be performed by repeated addition. |
| BUMP | *V,N-OF* | Increase value of *V* by literal value *N-OF* (this may clobber A). |
| ANDV | *V* | "and" A with value of *V*. |
| ANDL | *N* | "and" A with literal value *N*. |

### 2.11.5 Compare statements

Compare statements compare A or C with an operand. Each is always followed immediately by a conditional jump statement. Compare statements may clobber registers and hence may be implemented as subtract instructions. The subsidiary argument to the `CAV` and `CAI` statements specifies whether the items to be compared are addresses (`A`) or signed integers (`X`). In the latter case either of the values to be compared may be positive or negative (some machines, for example the PDP-11, use the first bit as a sign bit for numerical values but not for addresses. For most machines, however, there is no difference). If two addresses are compared these may be addresses of variables, table labels, or stack items (see later). No assumptions are made about the relative magnitude of the different kinds of address. For example the stack (see Section 2.11.9 [Stacking and block moving], page 18) can be sited in higher memory or lower memory relative to variables and/or table items.

| | | |
|---|---|---|
| CAV | *V*,(X)<br>(A) | Compare A with value of *V*. |
| CAL | *N-OF* | Compare A with literal value *N-OF*. |
| CCL | *'character'* | Compare C with given character. |
| CCN | *charname* | Compare C with named character. |
| CAI | *V*,(X)<br>(A) | Compare A with value pointed at by *V*. |
| CCI | *V* | Compare C with character pointed at by *V*. |

In the case of the `CAL` statement, the argument may be zero. On some machines, comparing with zero is a redundant operation and the implementor may decide to eliminate such statements. If this is so, it should be borne in mind that `CAL 0` may be the very first instruction in a subroutine (meaning test if the parameter is zero — see later). because of this possible optimisation, no load statements immediately before a `CAL 0` have the subsidiary argument `R`.

### 2.11.6 Subroutines

A subroutine in LOWL may be called from anywhere, including from within another subroutine. However no subroutines are recursive. There may be at most one parameter, which, if it exists, is always numerical, is always passed in the A register, and should always be stored in the variable PARNM.

Some subroutines have multiple exits. In this case exit 1 is a normal return to the point of call, exit 2 returns to the point of call but skips over the next LOWL statement, exit 3 skips two LOWL statements, and so on. The LOWL statements thus skipped over are always GO statements with a special subsidiary argument (see Section 2.11.8 [Branching statements], page 17). Subroutines in the MI-logic never return a value in a register.

Before defining the statements for subroutine linkage it is best to show an example. Assume the subroutine FACT calculates the factorial of its parameter and places the result in VALUE. If the parameter is negative it uses exit 1 and in normal cases it uses exit 2. The calling sequence for FACT might be:

```
            LAV     ARG,X
            GOSUB   FACT,...
            GO      ERROR,...
```

and the declaration might be:

```
            SUBR    FACT,PARNM,2
            LAL     1
            STV     VALUE,X
            LAV     PARNM,X
            CAL     0
            GOGR    FCT1,1,X,X
            EXIT    1,FACT
[FCT1]       .
             .
             .
             .
            STV     VALUE,X
            EXIT    2,FACT
```

The declaration of a subroutine may come before or after the first call of the subroutine.

As can be seen from the example, subroutines are declared by a LOWL statement of form

```
            SUBR    subroutine name,(PARNM),N
                                    (  X  )
```

where the second argument is X if there is no parameter. The third argument gives the number of exits. The SUBR statement should be mapped into code to place A into PARNM if there is a parameter, and store the return link in all cases. If there is a parameter it should remain in A, i.e. the action of setting PARNM and preserving the link should not clobber A. The code would normally be labelled with the subroutine name so that the mapped version of the GOSUB statements can reference it.

Since there is no recursion, each subroutine may preserve its link in a fixed variable unique to that subroutine. Alternatively subroutine links can be stored in a stack (though

*not* the stack used by the MI-logic). Such a stack need only have room for a dozen items since this is the maximum depth of subroutine nesting.

The statement to return from a subroutine takes the form:

```
EXIT    N,subroutine name
```

where $N$ is the number of the exit to be taken.

Subroutines are called by the statement:

```
GOSUB   subroutine name,(distance)
                        (   X    )
```

A call may reference a routine in the MD-logic, in which case the second argument is `X`. Otherwise the second argument gives the number of LOWL statements between the call and the subroutine declaration. The distance is negative if the declaration precedes the call.

The purpose of giving the distance is to allow optimisation on machines with special instructions for short-distance jumps. The distance does not include LOWL statements that are comments, and is measured from the statement following the given one — i.e. the following statement is at distance 0 and a statement is at distance $-1$ from itself. When LOWL is mapped into another language, distances will change unless the mapping is one-to-one. However the distance in terms of LOWL statements will still be a useful approximation.

### 2.11.7 The GOADD statement

The statement:

```
GOADD   V
```

is a multi-way branch statement. It is always followed immediately by a series of unconditional `GO` statements (see Section 2.11.8 [Branching statements], page 17). If the value of $V$ is zero the first such `GO` is executed; if the value of $V$ is one the second `GO` is executed, and so on.

### 2.11.8 Branching statements

LOWL contains both conditional and unconditional branching statements. The former always immediately follow a compare statement. Any conditional branching statement may clobber any register, but an unconditional branch must leave the registers unchanged. There are no branches into a subroutine from outside, but there are branches out of subroutines to the main logic (i.e. those parts of the logic not within a subroutine). If subroutine return links are being stored on a stack, these branches may cause problems, so there is a special statement of form:

```
CSS
```

(which means *C*lear *S*ubroutine *S*tack) to alleviate this situation. `CSS` statements appear after each label in the main logic that is referenced from within a subroutine. On implementations which do not use a stack for return links, the `CSS` statement will, of course, be mapped into a null instruction.

All branching statements have a set of four arguments called a *label spec*, which takes the following form:

*Argument 1*
: Name of designated label.

*Argument 2*
: distance of designated label (as for second argument to `GOSUB`).

*Argument 3*
: `E` if the branch goes out of a subroutine; `X` otherwise.

*Argument 4*
: `C` if the branch is an exit following a `GOSUB` statement; `T` if the branch is one of a sequence following a `GOADD` statement; `X` otherwise.

The purpose of argument 4 is that, if it is not `X`, the branch must be mapped into an instruction of a standard form; this may be significant on a machine with different lengths of jump instruction.

The full list of branching statements is as follows:

`GO` unconditional branch.
`GOEQ` branch if equal.
`GONE` branch if not equal.
`GOGE` branch if greater than or equal (i.e. if A is greater than or equal to the compared operand).
`GOGR` branch if greater than.
`GOLE` branch if less than or equal.
`GOLT` branch if less than.

There are, in addition, two statements which test the C register. These do not follow compare statements, but rather they follow a statement that loads the C register. The statements are:

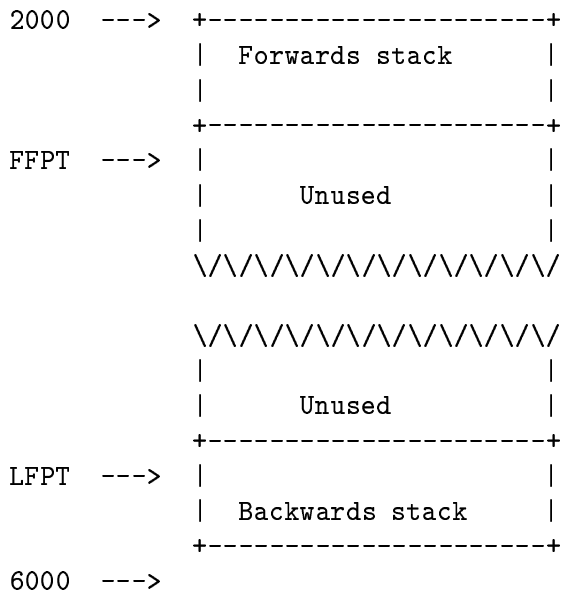
`GOPC` branch if character in C is a punctuation character, i.e. not a letter or a digit.
`GOND` branch if character in C is not a digit. If it *is* a digit, load the A register with the value of the digit (e.g. if C contained the value 51, which happened to be the internal code of the digit '3', then A should be set to the value 3).

The following example illustrates the use of label specs.

```
            SUBR    SHOW,X,1
            LAV     PIG,X
            CAV     COW,X
            GOEQ    SAME,3,X,X
            GOSUB   INVERT,-620
            GO      SAME,1,X,C
            GO      OUT,516,E,X
    [SAME]  EXIT    1,SHOW
```

### 2.11.9 Stacking and block moving statements

All software encoded in LOWL uses a double stack within a single contiguous block of storage. If this block of storage runs from, say, addresses 2000 to 6000 the stacks might look like this:




In particular the variable FFPT points at the first free location of the forwards stack and LFPT points at the last location in use on the backwards stack. Stacks are initialised by routines in the MD-logic.

There are four statements in LOWL concerned with stacking. These are:

| | | |
|---|---|---|
| FSTK | | stack A on forwards stack. |
| BSTK | | stack A on backwards stack. |
| CFSTK | | stack C on forwards stack. |
| UNSTK | *V* | unstack number at top of backwards stack and put it in *V*. |

In terms of other LOWL statements, the action of these is as follows. The label ERLSO is the label of the code that deals with stack overflow; it is present in every MI-logic. First, FSTK:

```
[FSTK]  STI     FFPT,X
        LAV     FFPT,X
        AAL     OF(LNM)
        STV     FFPT,P
        CAV     LFPT,A
        GOGE    ERLSO,...
```

CFSTK is essentially the same as FSTK, except that C is stored and FFPT is incremented by OF(LCH). Next, BSTK:

```
[BSTK]  preserve A
        LAV     LFPT,X
        SAL     OF(LNM)
        STV     LFPT,X
        restore A
        STI     LFPT,X
        LAV     FFPT,X
        CAV     LFPT,A
        GOGE    ERLSO,...
```

Lastly, UNSTK:

```
[UNSTK] LAI     LFPT,X
        STV     V,X
        BUMP    LFPT,OF(LNM)
```

On most mappings of LOWL, these rather clumsy instruction sequences can be improved upon; indeed the very purpose of the stacking statements is to permit the use of specially optimised code.

There are two "block moving" statements for moving blocks of contiguous locations to and from positions in the stacks. In each case SRCPT points at the start of the source field, DSTPT points to the start of the destination field and A contains the length of the field (which exceeds zero). The statements are:

FMOVE perform forwards move.
BMOVE perform backwards move.

Each should perform a character by character move from source to destination. FMOVE should start with the first character and end with the last, while BMOVE should do the reverse. The effect only differs, of course, when the two fields overlap. The values of DSTPT and SRCPT need not be preserved.

### 2.11.10 I/O statements

Almost all I/O in LOWL is performed through machine-dependent subroutines. There is just one statement in the kernel of LOWL which deals with I/O, and this has the form:

```
        MESS    'characters'
```

The MESS statement means output the given message, which may be an error message or an informatory message. The output device will normally be a printer or a workstation screen. Each character of the message stands for itself except for a dollar sign, which means a newline. Thus:

```
        MESS    'ONE$TWO$$THREE'
        MESS    'FOUR'
```

should produce the output

```
ONE
TWO

THREEFOUR
```

The way the message is represented and stored is entirely up to the implementor. If it is possible to pack the message then this should be done.

Although all the other I/O routines belong to the MD-logic it is worth noting a few general points. Most messages contain some variable information, for example:

```
Identifier ... in line ... not declared
```

The fixed parts are generated by MESS statements and, because of variations in character sets, the variable parts are generated by MD-logic subroutines. The output from the two is therefore interspersed. Because the variable information is indeterminate in length as well as in form, lines may be arbitrarily long. The identifier name in the above message might, for example, be one character or a hundred characters in length. Hence both the

MESS routine and the associated MD-logic subroutines may, in some implementations, need to check for overflow of the output buffer.

Most LOWL software caters for two output streams; the *message stream* described above, and a *results stream*, which is used for the main results. In all cases the logic is defined so that the two may or may not go to the same physical device. Two possible situations are, for example, messages going to a workstation screen and results to a disk file, or both messages and results being interspersed on some printed medium.

### 2.11.11 Comment and layout statements

Comments in LOWL are written:

```
NB      'characters'
```

There are two other layout statements, namely:

```
PRGST   'characters'
```

which occurs once, at the very start of the MD-logic, the *characters* giving the program name, and:

```
PRGEN
```

which occurs at the end. PRGST and PRGEN will probably map into null instructions on most implementations. None of NB, PRGST or PRGEN is ever labelled.

The MI-logic is always organised so that the variable declarations come first, the table items (if any) come second and the executable statements come last. The first table item, if one exists, is labelled TABFST and the first executable statement is labelled BEGIN. The overall layout of the MI-logic is therefore:

```
          PRGST   'characters'
          (declarations of variables/manifest constants)
[TABFST] (table items)
[BEGIN]  (executable statements)
          PRGEN
```

## 2.12 Uniqueness of names

All variable names, constant names, labels, subroutine names and statement names in LOWL are unique. Thus, for example, GO, being a statement name is never used as the name of anything else and all occurrences of GO can be taken as a call of the mapping macro for the GO statement unless the GO occurs in string quotes, e.g.:

```
MESS    'Illegal GO TO'
```

## 2.13 Interface with the MD-logic

The MD-logic consists of some initialisation code together with a set of subroutines. The initialisation code is entered before the MI-logic. When the MD-logic has performed the necessary initialisation it branches to the label BEGIN in the MI-logic, which then takes control. Communication with the MD-logic is then by means of subroutine calls. At the end

of execution, either by natural termination or because of a fatal error such as stack overflow, the MI-logic calls the MD-logic subroutine `MDQUIT`, which performs the final tidying up.

All subroutines in the MD-logic have names beginning with the letters MD. These subroutines may clobber registers, but should not, except where otherwise stated, change the values of any MI-logic variables.

The nature of the initialisation code varies according to the software concerned, but always includes the following tasks, which are called the *common initialisation code*.

a. Reserve an area of contiguous storage for the stacks. A suitable size for this depends on the nature of the software being implemented, but clearly the bigger the better. The software must not be entered with a stack smaller than two words, and for most software the practical minimum is about fifty words. `FFPT` should be set to point to the start of this area and `LFPT` should be set to point immediately beyond the end.

b. Perform any necessary I/O initialisation.

c. If the `GOSUB` statement is being implemented using a stack, then initialise the stack.

d. Output an introductory message if desired, e.g.:

```
Supersys version 1 entered
```

e. Branch to the label `BEGIN` in the MI-logic.

All initialisation in the MI-logic is performed dynamically (i.e. by means of explicit assignments rather than by preset initial values) so that the software can be re-used without being reloaded, if this is appropriate on the object machine. Ideally the MD-logic initialisation should be performed in the same way.

The duties of the `MDQUIT` subroutine are to close all I/O (there might be incomplete lines in output buffers), to release resources (e.g. storage areas "borrowed" from the operating system) and to quit (`MDQUIT` does not, and should not, return to its point of call in the MI-logic). The way `MDQUIT` is encoded depends, of course, on the software being implemented.

## 2.14 Alignment

On implementations where `LNM` and `LCH` are unequal it may be necessary to consider the problem of data alignment. Implementors without alignment problems can skip this Section — all they need to do is map the `ALIGN` statement into a null instruction.

We will discuss alignment problems with reference to a specific example. Assume that the object machine works in units of words, each word being divided into four bytes. An item of character data occupies one byte and an item of numerical data four bytes. Problems may arise in a table of data when, for example, a single character is followed by a number (the problem applies both to table items and to dynamically created data on a stack). One way of storing such data is to place the character in the first byte of one word and the number in the next three bytes of that word and the first byte of the next word. However some object machines cannot directly address numbers that straddle word boundaries. For these machines LOWL statements that address numbers indirectly, such as `LAI`, would need to be mapped into instructions to assemble the number into a word before loading it. A similar problem exists for statements that store indirectly addressed numbers. This may be very slow and cumbersome.

To combat this problem LOWL provides a statement called:

```
ALIGN
```

As can be seen, this takes no arguments. This can be used to force numerical data to be aligned to any desired boundary. Taking the example of the number following a single character, the number could be stored in the next word following the word containing the character, the last three bytes of the latter being unused. This might eliminate addressing problems.

The following are the rules for the implementor to follow, depending on whether the data is to be aligned or not.

- *Unaligned data.* The `ALIGN` statement is mapped into a null instruction. Statements for defining numerical table items (e.g. a `CON` statement) are *not* aligned but follow immediately after the preceding table item.
- *Aligned data.* The `ALIGN` statement is mapped into instructions to align the contents of the A register up to the desired boundary. Statements defining numerical table items are aligned to the same boundary, as are the initial values of `FFPT` and `LFPT` together with any numerical pointers created by the MD-logic (when alignment is performed, no special markers need to be put in the unused portions of words; software written in LOWL always works with the true lengths of character strings and never looks at padding beyond the end). Given this alignment, the implementor can always assume that, when a number is to be addressed indirectly, the requisite pointer is correctly aligned. This applies for example to statements such as `LAI`, `LAM`, `STI`, `FSTK`, `UNSTK` and to pointers supplied as arguments to MD-logic routines. The implementor can thus forget about alignment problems when mapping such statements.

In either situation, variable declarations created by `DEC` or `EQU` statements should be aligned to the boundary most convenient for direct addressing.

## 2.15 Summary of LOWL

To summarise, the basic elements in the kernel of LOWL are as follows:

*Data types*
: Character (single character), number (may be integer value or pointer).

*Variables*
: Represented by identifiers. No character variables.

*Constants*
: Numerical: decimal integer or call of `OF` macro. Character: single character in quotes, or name.

*Registers*
: Three: A, B and C.

*Labels*
: Represented by identifiers. Enclosed in square brackets where placed.

*Subroutines*
: Names are identifiers. At most one argument.

The following is a complete list of the statements in the kernel of LOWL. Those marked with an asterisk may clobber any of the registers.

| | | |
|---|---|---|
| `DCL` | *`V`* | declare variable. |
| `EQU` | *`V,V`* | equate two variables. |
| `IDENT` | *`V,decimal integer`* | equate name to integer. |

```
CON     N-OF                        numerical constant.
NCH     charname                    character constant.
STR     'characters'                character string constant.
LAV     V,(R)                       load A with variable.
          (X)
LBV     V                           load B with variable.
LAL     N-OF                        load A with literal.
LCN     charname                    load C with named character.
LAM     N-OF                        load A modified.
LCM     N-OF                        load C modified.
LAI     V,(R)                       load A indirect.
          (X)
LCI     V,(R)                       load C indirect.
          (X)
LAA     V,D                         load A modified (variable).
LAA     table label,C               load A modified (table item).
STV     V,(P)                       store A in variable.
          (X)
STI     V,(P)                       store A indirectly in variable.
          (X)
*CLEAR  V                           set variable to zero.
AAV     V                           add to A a variable.
ABV     V                           add to B a variable.
AAL     N-OF                        add to A a literal.
SAV     V                           subtract from A a variable.
SBV     V                           subtract from B a variable.
SAL     N-OF                        subtract from A a literal.
SBL     N-OF                        subtract from B a literal.
MULTL   N-OF                        multiply A by a literal.
*BUMP   V,N-OF                      increase a variable.
ANDV    V                           "and" A with a variable.
ANDL    N                           "and" A with a literal.
*CAV    V,(X)                       compare A with variable.
          (A)
*CAL    N-OF                        compare A with literal.
*CCL    'character'                 compare C with literal.
*CCN    charname                    compare C with named character.
*CAI    V,(X)                       compare A indirect.
          (A)
*CCI    V                           compare C indirect.
SUBR    subroutine name,(PARNM),N   declare subroutine.
                        (  X  )
*EXIT   N,subroutine name           exit from subroutine.
GOSUB   subroutine name,(distance)  call subroutine.
                        (   X   )
*GOADD  V                           multi-way branch.
*CSS                                clear subroutine stack (if any).
```

| | | |
|---|---|---|
| GO | *label spec* | unconditional branch. |
| *GOEQ | *label spec* | branch if equal. |
| *GONE | *label spec* | branch if not equal. |
| *GOGE | *label spec* | branch if greater than or equal. |
| *GOGR | *label spec* | branch if greater than. |
| *GOLE | *label spec* | branch if less than or equal. |
| *GOLT | *label spec* | branch if less than. |
| *GOPC | *label spec* | branch if C is a punctuation character. |
| *GOND | *label spec* | branch if C is not a digit; otherwise put value in A. |
| *FSTK | | stack A on forwards stack. |
| *BSTK | | stack A on backwards stack. |
| *CFSTK | | stack C on forwards stack. |
| *UNSTK | *V* | unstack from backwards stack. |
| *FMOVE | | forwards block move. |
| *BMOVE | | backwards block move. |
| *MESS | *'characters'* | output a message. |
| NB | *'characters'* | comment. |
| PRGST | *'characters'* | start of logic. |
| PRGEN | | end of logic. |
| ALIGN | | align A up to next boundary. |

A mapping of LOWL simply requires mapping macros for each of the above statements plus possible subsidiary macros to deal with labels and constants.

# 3 Mapping and documentation

Some of the attributes that mark out professional software writers from the cowboys are:

a. their software is completely tested.
b. their software is completely documented.
c. their software is easy to use and operate. In particular the operating system interface is smooth.
d. the implementation process is adaptable to future changes.
e. the implementation process is sufficiently well documented for someone else to take it over at any time.

Anyone who thinks that they have completed an implementation when the software has been coded up and a few test cases run can be equated with a man who thinks he has overcome all the problems of marriage when he has finished his speech at the wedding reception.

The purpose of this Chapter is to cover b), d) and e) of the above points. Of the other two points, a) is covered by LOWLTEST and the test data for the software to be implemented, and point c) largely by the way the MD-logic is encoded. Implementors might add a further point to the above list, which is that others are expected to adopt the same professional standards. They should thus feel free to hit back and point out errors and inadequacies in LOWL itself, its documentation or the software issued in it.

## 3.1 The mapping

The most important point about a mapping is that it should not be regarded as a one-off job. Several pieces of software have been encoded in LOWL, and, although the immediate aim may be to implement only one of these, others may be implemented at a later date. Moreover the software might be extended and improved. It may even be that the actual implementor, after seeing the software in use for a while, may design some extensions to it. In this case they would do well to make the changes to the LOWL encoding of the software (rather than the mapped version) and re-map this, thus retaining portability. Since the mapping macros might be re-used, perhaps by someone else, the implementor should write them in a well organised and well documented manner.

An extension of this point is that LOWL itself might be developed and improved. In particular, subsidiary arguments might be added to some existing LOWL statements. If at all possible, mapping macros should be written so that if an extra subsidiary argument is added to the end of the argument list, this should not affect the working of the macro. To take an explicit example of this that arose in the past, the `CAV` statement in LOWL was originally written in the form:

```
CAV     V
```

It subsequently became clear that, for at least one object machine, it would be useful to know whether $V$ was an address or simply a number. Hence a subsidiary argument was added, the CAV statement being written:

```
CAV     V,A
```

or

```
CAV     V,X
```

depending on whether $V$ was an address or not. All previous mappings of LOWL were not interested in this argument, and, since the mapping macros had been written to allow for extra redundant arguments, these mappings still worked although LOWL had been changed. The point is this: it is not worth taking a lot of trouble to make mapping macros allow for extra arguments, but if they have this property already — as is the case for most macro-assembler macros — then it is foolish to do anything to destroy it (like assuming that a certain argument is the last argument). Since flexibility is the key factor in portability projects it is certainly an advantage to be able to make slight changes to descriptive languages painlessly.

## 3.2 Common mapping problems

It may be worth bringing the implementor's attention to two rather mundane problems which, nevertheless, cause difficulties on many mappings.

One problem concerns arguments that are character strings, particularly arguments to the `MESS` and `STR` statements. These strings are literals and should not be subject to macro replacement (except perhaps the `$` which stands for a newline in the `MESS` statement). Moreover, spaces within these character strings are significant, including any that occur at the beginning and end.

Character strings provide the only instance in LOWL where spaces occur within arguments. Tabs never occur within arguments but are used to separate the statement name from the first argument. These tabs (which may be represented by spaces) do not, of course, count as part of the first argument. The amount of difficulty presented by spaces and tabs varies between macro processors. However it is often necessary to plan carefully the layout of the output from a macro processor and where spaces and tabs are to appear in it.

A further problem concerns potential recursion. It may happen, for example, that the object machine has an instruction called `GO`, and that the `GO` statement of LOWL will map into this (although the argument structure will almost certainly be different in the two cases). The replacement text of the `GO` statement will then involve a `GO` instruction. Many macro processors, unless told otherwise, would treat this `GO` instruction as a recursive call of the `GO` macro.

## 3.3 Some mapping macros

This Section contains some samples of mapping macros. Unfortunately it is necessary, when showing examples, to fix on one macro processor, and in this case ML/I has been chosen. It is hoped, however, that the macros will still provide some useful insights for all implementors, although they most likely will not be using ML/I as the mapping tool and will be unfamiliar with ML/I notation.

When using ML/I it is convenient to perform the minor systematic editing on LOWL at the same time as the macros are mapped, thus eliminating a prepass. Assuming, for example, that the square brackets round labels were to be deleted, this would be achieved by defining them as skips:

```
MCSKIP [
MCSKIP ]
```

It is also usually convenient to cause all tab characters in the source text to be ignored. This can also be done by defining a suitable skip.

A mapping macro for a simple statement might be:

```
MCDEF LAV N1 OPT , N1 OR NL ALL
AS <        LOAD     %A1.
>
```

The delimiter structure defines `LAV` as the macro name, which is followed by an indefinitely long sequence of arguments separated by commas. The macro generates a `LOAD` instruction. If we wanted to be clever and eliminate the `LOAD` instruction if the second argument was `R` (for redundant) then we might have written the replacement text:

```
<MCGO L0 IF %A2.=R
        LOAD     %A1.
>
```

Here, `MCGO L0` means exit from the macro. Alternatively it might be preferred to generate a comment in the case when a `LAV` statement was ignored. A suitable comment might be:

```
Accumulator already contains ...
```

While on the subject of comments, it is often helpful to place the source LOWL statement as a comment on the assembly language instructions that it generates.

Some LOWL statements might not be replaced by in-line code, but might generate a call to a subroutine. Thus if it required six instructions to effect the `FMOVE` statement then the mapping macro for `FMOVE` might be written:

```
MCDEF FMOVE NL
AS <        CALL     FMVSUB
>
```

where `FMVSUB` was a subroutine consisting of the six required instructions. (The `NL` in the delimiter structure of `FMOVE` stands for "newline". The text in between, in this case null, counts as the argument.) It may be convenient to combine such subroutines as `FMVSUB` with the MD-logic.

Lastly, we will show the mapping macro for the `MESS` statement, since this has rather special properties. We will assume that

```
MESS    'ABC'
```

is mapped into

```
CALL     MESSUB
TEXT     "ABC~"
```

where the ~, not being a character used in LOWL, acts as an end marker for the message. The mapping macro might be written:

```
MCDEF MESS WITH TAB WITH ' ' NL
SSAS <        CALL     MESSUB
        TEXT     "%WB1.~"
>
```

Here the macro name is `MESS` followed by a tab and quote (even if tab has been defined as a skip this would not affect a tab within a macro name), the first delimiter is quote and the final delimiter is a newline — this allows for any subsidiary arguments that might be added. The `SSAS` in place of the usual `AS` means that the macro is "straight scan", i.e. no account is taken of nested calls when scanning for delimiters of this macro. The notation `%WB1.` means insert argument 1 exactly as written and include any spaces that occur at the beginning and end. It is assumed that the `MESSUB` routine takes care of the conversion of dollar signs within messages to newlines.

## 3.4 Documentation

The implementor will need to provide a *User's Manual* of the software that has been implemented. In general, existing User's Manuals for software written in LOWL have been written in as machine-independent a way as possible. Implementation-dependent features, such as the operating system interface, size limits and character sets, have been placed in separate Chapters or Appendices. Where this is so, the only documentation needed for a new implementation is a re-write of these Chapters and Appendices. Some of the Supplements describing individual pieces of software contain information about more specialised documentation needs.

In addition, the implementor should provide some documentation on the implementation process, as mentioned earlier.

Critical evaluations of the mapping process are also of value. Interesting questions are:

a. how efficient is the final implementation?
b. where do the inefficiencies lie?
c. how long did the implementation take?
d. how much machine time did an implementation take?
e. which features of LOWL proved difficult to map?
f. was the mapping tool adequate?
g. could the software have been better implemented some other way?

# 4 The LOWL kernel test program

LOWLTEST is a program to test the mapping macros for the kernel of LOWL. It was originally designed and written by R.C. Saunders, working on a project supported by a research grant from the Science Research Council.

The first action of LOWLTEST is to print an introductory message using the `MESS` statement. It then tests the following seven statements:

1. `GO`
2. `LAL`
3. `CAL`
4. `GONE`
5. `GOEQ`
6. `STV` with `X` as second argument
7. `LAV`

If any of these tests fail, LOWLTEST prints a suitable error message, for example:

```
MESS    '+++Error in LAL or CAL with non-zero arg'
```

and abandons the run.

After this, LOWLTEST tries to test the remaining LOWL statements independently of one another, relying on the use of the seven statements that have already been tested, together with the `MESS` statement. If an error is found at this stage, a message is printed but the run continues. However some of the later tests may be omitted, since they may be dependent on an incorrect statement (e.g. `UNSTK` depends on `BSTK`). *The implementor must therefore continue to run LOWLTEST until it works completely.*

Mountain walkers, when traversing dangerous territory, should leave behind a message such as "We are climbing to Windy Ridge and then crossing Suckfoot Bog to Creakbridge". This helps the rescue party to find them. LOWLTEST adopts a similar policy, and its output, after the introductory messages, should consist of a series of messages like

```
...Testing GOADD..
```

If the program is working correctly each of these should be followed by the acknowledgement `OK` or `found`. These tests are followed, at the very end, by five lines of output that should read

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789
$.,;:()*/-+=    "
Should be the same as
ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789
.,;:()*/-+= TAB and quote sign
```

If there are any error messages, these will begin with the characters `+++`.

## 4.1 Extensions, the MD-logic and I/O

By its very nature LOWLTEST requires no extensions to LOWL but it does require a small MD-logic. The input/output requirements are minimal, namely one output stream, which is the message stream used by `MESS` statements, and no input stream.

The MD-logic consists of two subroutines and some initialisation code. The subroutines are as follows:

1. `MDERCH`. Output the character in the C register on the message stream. The character set is exactly the LOWL character set.
2. `MDQUIT`. As defined in the description of LOWL.

The initialisation code is exactly the common initialisation code given in the description of LOWL.

# Statement/Macro Index

# Concept Index

## S



## T



## U



## V



## W