A simple introductory guide to ML/I

Fourth Edition

R.D. Eager, P.J. Brown

Introduction

ML/I is a general purpose macro processor that is available on many different systems. It has many uses; probably the most common are the extension of existing programming languages, and systematic editing.

There already exists a reference manual for ML/I (*ML/I User's Manual*), and also a paper describing ML/I (*The ML/I macro processor, Comm. ACM 10, 10 (October 1967)*). However, the former is comparatively large and hence rather forbidding, while the latter tries to show off some of the more advanced features of ML/I. This Guide provides a short and simple introduction to ML/I. Obviously, it has been necessary to omit many features and to over-simplify others, but nevertheless it is hoped that the reader will get a feel of what ML/I is all about, and an idea of some of its uses.

In order to show exactly what goes into ML/I and, as a result, what comes out, examples in this Guide have been written as if the reader were using ML/I at an interactive terminal or workstation. The examples represent a sequence of lines of input to ML/I, starting from scratch. The input lines are numbered sequentially to aid cross-referencing. Lines of output are labelled with the word `Output`, e.g.

```
        23)   THIS IS A LINE OF INPUT
    Output)   THIS IS THE CORRESPONDING OUTPUT
```

Before actually using ML/I, you should find out the exact form and manner of the input to ML/I on the system which you are using.

Basic principles

ML/I operates on characters, not numbers. It is fed, as input, a string of characters called the *source text*. There is no restriction on the form of the source text; it may, for example, be a program in any programming language, a scientific paper, a circular letter or some data for a program. What ML/I does to the source text is to scan through it making systematic replacements; the resultant text is the output. ML/I produces its output as it goes along. Its usual sequence of operation is: read a line; perform the necessary replacements; output the line. However, ML/I is inherently stream-based rather than line-based, so line boundaries are merely treated as part of its input (but it can easily handle text that is inherently line-based). In some uses of ML/I, the output is subsequently fed to a compiler or an assembler.

Macros

The way of making a replacement is by means of a *macro*. A macro is defined by writing:

```
    MCDEF x AS y
```

where `x` describes what is to be replaced, and `y`, which can be any string of characters, is what it is to be replaced by. `y` is called the *replacement text*.

We shall now start our imaginary session with ML/I, and will illustrate some simple macros.

The first two lines of input to ML/I are normally used to define some special symbols. We shall supply these without explanation now, and will refer back to them later:

```
         1)   MCINS %.
```

```
      2)   MCSKIP MT,<>
```

We shall now define a macro:

```
      3)   MCDEF JONES AS <SMITH>
```

The above definition causes every subsequent occurrence of JONES to be replaced by SMITH, for example:

```
      4)   PEOPLE SHOULD CONSULT MR. JONES
 Output)   PEOPLE SHOULD CONSULT MR. SMITH
      5)   ....JONES....JONES....
 Output)   ....SMITH....SMITH....
```

Similarly, macros can be defined to replace punctuation characters, for example:

```
      6)   MCDEF & AS <;>
      7)   X = Y & Y = Z;
 Output)   X = Y ; Y = Z;
```

The recognition of a macro, and the substitution of its replacement text, is termed a *call* of the macro.

The normal way of using ML/I is first to feed it the definitions of the macros (and other similar *constructions* — see later) to be used, and then to feed it the source text in which replacements are to be made. However, it is possible to intersperse macro definitions with the rest of the source text, as indeed is being done in this Guide.

Atoms

To show how ML/I splits up its source text, consider the following:

```
      8)   MCDEF READ AS <INPUT TO THE COMPUTER>
      9)   THEN READ YOUR DATA, MR. JONES
 Output)   THEN INPUT TO THE COMPUTER YOUR DATA, MR. JONES
     10)   THE DREADED READER SHOULD READ
 Output)   THE DREADED READER SHOULD INPUT TO THE COMPUTER
```

Note that, in the line above, the sequence READ occurs within DREADED and within READER, but in neither case has it been replaced. This is because ML/I does not, in fact, scan character by character but rather atom by atom, where an *atom* is usually a single punctuation character (i.e. any character other than a letter or a digit) or a sequence of letters and/or digits bounded on each side by punctuation characters. Thus in the above line, DREADED is an atom and hence ML/I does not recognise the letters READ within it.

The character ‘space’ is treated as an atom, and so is the character ‘newline’, the latter being an imaginary character that occurs at the end of each line. As regards this newline character, users should be careful only to use it in replacement text exactly where they want it, for example:

```
     11)   MCDEF INTEGER AS <FIXED(15)
     12)   BINARY>
     13)   BEGIN INTEGER A,B;
 Output)   BEGIN FIXED(15)
 Output)   BINARY A,B;
```

Note how a newline appears in the output after FIXED(15).

Multi-atom names

The macros defined so far have replaced a single atom. However, it is possible, by using the keyword `WITHS`, to specify that a multi-atom sequence be replaced. For example:

```
     14)   MCDEF THE WITHS PRIME WITHS MINISTER
     15)   AS <MR. GLADSTONE>
     16)   THE PRIME MINISTER CHAIRS THE CABINET
 Output)   MR. GLADSTONE CHAIRS THE CABINET
```

Calls within calls

Macro replacement takes place within the replacement text of other macros as well as within the source text (in fact recursion, i.e. a macro calling itself, is allowed). For example:

```
     17)   MCDEF THE WITHS CHAIRMAN
     18)   AS <THE PRIME MINISTER OR HIS DEPUTY>
     19)   THE CHAIRMAN SPEAKS FIRST
 Output)   MR. GLADSTONE OR HIS DEPUTY SPEAKS FIRST
```

Here, within the replacement text of `THE CHAIRMAN`, `THE PRIME MINISTER` has been replaced.

Literal brackets

In all the examples of `MCDEF`, the replacement text has been enclosed within the characters `<` and `>`. These are called *literal brackets*. They mean 'copy the enclosed text as it stands'. You should choose, as your literal brackets, a pair of atoms (or multi-atom sequences) that do not occur naturally in the text to be processed. In this Guide, the literal brackets were specified in input line `2)` above. If you wanted to use multi-atom sequences (e.g. `*(` and `)*`) as literal brackets, this would be done thus:

```
     20)   MCSKIP MT, * WITHS ( ) WITHS *
```

With this definition (which has, in fact, supplemented rather than replaced the previous literal brackets) we can now write

```
     21)   MCDEF cur AS *(dog)*
     22)   cur
 Output)   dog
```

This macro has a name in lower case letters. Generally we have used upper case letters in examples in this Guide since they stand out better. However, ML/I works quite happily with lower case letters, and these are treated as entirely separate from their upper case equivalents. Thus `cur` does not match `CUR` or `Cur`. Built-in ML/I names such as `MCDEF` and `AS` must always be given in upper case.

Literal brackets have a secondary use, as illustrated by:

```
     23)   MCDEF REWIND AS <PRINT  'Finished with tape'
     24)   <REWIND>>
     25)   REWIND
 Output)   PRINT  'Finished with tape'
 Output)   REWIND
```

The occurrence of `REWIND` within the replacement text of the `REWIND` macro is enclosed in literal brackets (which are additional to those that enclose the entire replacement text) to cause it to be copied literally over to the output; if these literal brackets had been omitted, it would have been taken as a recursive call of the `REWIND` macro, and ML/I would have been set in an endless loop of replacing `REWIND` at successively deeper levels.

Arguments

The macros that have been defined so far have been of a rather simple kind, in that the thing to be replaced was always a single pre-defined atom or series of atoms. Now we shall consider more powerful macros, which reflect the true usefulness of ML/I. Assume we wish to replace:

```
UNSTACK x;
```

for any `x`, by:

```
x = STACK[PTR];
PTR = PTR - 1;
```

Here the macro has an argument, i.e. an arbitrary string between two predefined delimiters (in this case `UNSTACK` and semicolon). It is possible to have more than one argument to a macro, for instance one might want to replace:

```
APPEND X TO LIST Y.
```

by:

```
Y[0] = Y[0]+1;
Y[Y[0]] = X;
```

This macro has two arguments, which are delineated by the three delimiters: `APPEND`, `TO LIST` and full-stop. Delimiters are numbered 0, 1, 2, etc. Delimiter zero (in this case `APPEND`) is called the *macro name* and the last delimiter is called the *closing delimiter* (in the case of a macro with no arguments, the macro name is also its closing delimiter).

When a macro is defined, the delimiters are simply listed in the order in which they are to occur; they may be separated by one or more spaces or newlines. This is called a *structure representation*. Thus, the structure representation of the `APPEND` macro is (preceded by the necessary call of `MCDEF`):

```
    26)    MCDEF APPEND TO WITHS LIST .
```

Before specifying the replacement text of this macro, we will give some further explanation. When ML/I is given a definition of a macro that has arguments, each subsequent occurrence of the macro name in the source text is taken as a call of the macro; ML/I then searches for the first delimiter, then the second, and so on until it has found the closing delimiter. The arbitrary string occurring between delimiter *n* and delimiter *n+1* is called *argument n+1*. Thus, the first argument is argument one.

It is usually necessary to insert arguments into the replacement text of a macro, and this is done by writing `%An.` where `n` is the number of the argument to be inserted. Hence, continuing the definition of the `APPEND` macro:

```
    27)    AS <%A2.[0] = %A2.[0]+1;
    28)    %A2.[%A2.[0]] = %A1.;>
    29)    APPEND PATIENT TO LIST WAIT.
Output)    WAIT[0] = WAIT[0]+1;
Output)    WAIT[WAIT[0]] = PATIENT;
```

```
     30)   APPEND X/Y+9 TO LIST ARRAY.
 Output)   ARRAY[0] = ARRAY[0]+1;
 Output)   ARRAY[ARRAY[0]] = X/Y+9;
```

As a second example, the `UNSTACK` macro can be defined and used thus:

```
     31)   MCDEF UNSTACK ;
     32)   AS <%A1. = STACK[PTR];
     33)   PTR = PTR - 1;>
     34)   L:UNSTACK OP;
 Output)   L:OP = STACK[PTR];
 Output)   PTR = PTR - 1;
```

It is often convenient to use the imaginary character 'newline' as a delimiter, particularly as a closing one. Newlines themselves are ignored in structure representations (hence the fact that `AS` in the previous definitions started on a new line was not significant), so when it is required to specify newline as a delimiter it is necessary to use a keyword, namely `NL` (similarly, most implementations of ML/I have other keywords such as `SL`, `SPACE`, `SPACES` and perhaps `TAB`).

The following example defines a macro `CALL` with newline as its closing delimiter:

```
     35)   MCDEF CALL NL
     36)   AS <   BSR   %A1.
     37)   >
     38)   CALL Pig
 Output)      BSR   Pig
     39)   UNSTACK Y; LAB: CALL PIGGY
 Output)   Y = STACK[PTR];
 Output)   PTR = PTR - 1; LAB:   BSR   PIGGY
```

Optional delimiters

We will now go one step further in the elaboration of macros, and will introduce one of the most important concepts in ML/I, namely optional delimiters.

Assume one wishes to define a macro which has two alternative forms:

```
SET a = b + c
```

which is replaced by

```
      LSS   b
      ADD   c
      ST    a
```

and

```
SET a = b - c
```

which is replaced by

```
      LSS   b
      SUB   c
      ST    a
```

i.e. delimiter two can be either a plus sign or a minus sign. Options such as this are specified in structure representations by writing:

```
OPT branch1 OR branch2 OR . . . OR branchN ALL
```

where each branch can itself be any structure representation. In practice, a branch is usually a single delimiter. Hence, the structure representation of the `SET` macro is written:

```
      40)    MCDEF SET = OPT + OR - ALL NL
```

Within the replacement text of this macro, it is necessary to test whether delimiter two of the current call was a plus or minus sign, and to generate code accordingly. This is done thus:

```
      41)    AS <   LSS   %A2.
      42)    MCGO L1 IF %D2. = +
      43)       SUB   %A3.
      44)    MCGO L2
      45)    %L1.   ADD   %A3.
      46)    %L2.   ST    %A1.
      47)    >
```

Two new features have been introduced here. The first is *macro-time statements*, i.e. statements that are executed by ML/I when it encounters them at the time it is macro processing. The above example shows one such macro-time statement, the 'goto' statement `MCGO`. As can be seen, `MCGO` has an optional conditional clause (`MCGO ... IF ...`).

A second new feature is the extended use of the `%` notation to include the insertion of delimiters (e.g. `%D2.` meaning delimiter two) and to place labels that are the destination of the `MCGO` statements (e.g. `%L1.` and `%L2.` above). Unlike other types of insert, the 'inserting' of a label does not generate any text — i.e. it has a 'null' value.

We will now call `SET` to show that it works:

```
      48)    SET X1 = Y1+Z1
  Output)       LSS   Y1
  Output)       ADD   Z1
  Output)       ST    X1
      49)    SET BROWN = JONES - ROBINSON
  Output)       LSS   SMITH
  Output)       SUB   ROBINSON
  Output)       ST    BROWN
```

(Note how the `JONES` macro, defined back in input line 3, is still in existence.)

Variable numbers of arguments

We will now take the last step in the elaboration of macros, and will describe how to use macros that have a variable number of arguments.

Assume, therefore, that it is required to define a macro called `LET`, which is similar to the `SET` macro except that it can have, to the right of the equals sign, an arbitrary expression involving additions and/or subtractions. Thus typical calls of `LET` might be:

```
LET A = B + C - D + F - G
LET X = Y
LET X = X + Y + C + D
```

When scanning a call of this macro, ML/I first encounters `LET` and then an equals sign. It then searches for plus, minus or a newline. If it finds either of the first two it recycles, i.e. again looks for plus, minus or a newline. If it finds a newline, then that is the closing delimiter, and thus the call is complete and the replacement text can be generated.

This scheme for searching for delimiters is specified in structure representations by the use of *nodes* (the word *node* is used since the delimiter structure of a macro can conveniently be represented as a directed graph). A node is *placed* at a given point in a structure representation, and can be *gone to* from the end of any branch in the same structure representation. Nodes, which are local to the structure representation in which they occur, are written `N1`, `N2`, `N3`, etc. A node is placed just by writing its name at the appropriate point within a structure representation, and is gone to simply by placing its name at the end of a branch; note that there is no explicit 'goto'. Hence the structure representation of the `LET` macro is written:

```
        50)   MCDEF LET = N1 OPT + N1 OR - N1 OR NL ALL
```

Here there is one node, called `N1`, which is placed before the alternative 'plus, minus or newline'. If either the plus branch or the minus branch is taken, the branch ends by going back to `N1`. If, on the other hand, the newline branch is taken, the next delimiter is taken as the one following the `ALL`. In this case, nothing follows `ALL`, so this newline is the closing delimiter.

We shall now consider the problems that arise in specifying the replacement text for the `LET` macro. The main problem is that one does not know in advance how many arguments there will be. The way to deal with this is to write a macro-time loop that takes the arguments one by one until they have run out. To do this, one needs variables for counting and subscripting; ML/I caters for this need by supplying three integer variables called `T1`, `T2` and `T3` which are local to each macro call (there also exist permanent, global variables called `P1`, `P2`, etc., as well as others, but these are not of immediate interest here). These variables are called *macro variables*, and ML/I contains an assignment statement, `MCSET`, for manipulating them. `MCGO` can be used for testing them. Macro variables or expressions involving them can be used as subscripts; e.g. if `T1` had value `3`, then `%AT1.` would mean 'insert argument three' and `%DT1 - 1.` would mean 'insert delimiter two'. In addition, values of macro variables can be inserted as they stand; e.g. `%T1.` would generate the character `3`.

Using these facilities, the replacement text of the `LET` macro is written:

```
        51)   AS <   LSS   %A2.
        52)   MCSET T1 = 3
        53)   %L4.MCGO L2 IF %DT1-1. = +
        54)   MCGO L5 UNLESS %DT1-1. = -
        55)      SUB   %AT1.
        56)   MCGO L3
        57)   %L2.   ADD   %AT1.
        58)   %L3.MCSET T1 = T1 + 1
        59)   MCGO L4
        60)   %L5.   ST    %A1.
        61)   >
```

and a sample call is:

```
        62)   LET A = B-C + D
    Output)      LSS   B
```

```
Output)      SUB   C
Output)      ADD   D
Output)      ST    A
```

Specialised example

As a last example, we shall show a macro which illustrates no new concepts, but which may be of interest to readers familiar with Polish notation. The macro converts from fully parenthesised algebraic notation to Polish prefix notation:

```
    63)   MCDEF ( OPT + OR - OR * OR / ALL )
    64)   AS <%D1. %A1. %A2.>
    65)   (A + B)
Output)   + A B
    66)   ((A-(B*C))/(X/Y))
Output)   / - A * B C / X Y
```

In this example, the macro name is a left parenthesis and its closing delimiter is a right parenthesis. Line 66 above shows a nested and, in fact, recursive call of this macro.

Skips and inserts

This concludes the description of macros as such, and we will end this Guide by clearing up a few isolated features not already covered. Firstly, we shall consider *inserts* and *skips*.

Insert is the name for the `%` facility. All that remains to be said about this is that, in a similar manner to literal brackets, the user chooses as the insert marker (i.e. what we have used `%` for) an atom or sequence of atoms that do not occur naturally in the text to be processed. The insert marker is normally defined at the start of the source text, as in input line 1 above. If it was desired to use `//` as an insert marker, it would have been defined:

```
    67)   MCINS /WITH/ .
```

(`WITH` is similar to `WITHS`, but means that no spaces are allowed between the two atoms it connects.)

Skips have already been mentioned, in that literal brackets are a special case of a skip. A skip has, like a macro, an associated structure representation. When ML/I encounters the name of a skip, it 'switches off' all the macros until it comes to the closing delimiter of the skip. The only things that may be recognised within a skip are other skips, and these are only recognised if the first skip has the `M` (for *matched*) option set. What ML/I does to skips is controlled by two further options: the `D` option means 'copy the delimiters to the output' and the `T` option means 'copy the intervening pieces of text' (i.e. in macro terms, the arguments). Hence if, for example, neither `D` nor `T` is set, the skip is totally deleted. A skip is defined thus:

```
MCSKIP options, structure representation
```

For example:

```
    68)   MCSKIP DT, ' '
```

means define the quote sign as a skip name with another quote as the closing delimiter, and set the `D` and `T` options (but not the `M` option) for it. The following skip would, on the other hand, be totally deleted since no options are set:

```
        69)   MCSKIP , COMMENT ;
```

The following examples show how these two skips work:

```
        70)   'LET SET JONES'
    Output)   'LET SET JONES'
        71)   COMMENT LET IS A MACRO; 'COMMENT'
    Output)   'COMMENT'
```

Searching for delimiters

If ML/I has found a macro name and is searching for its delimiters, it may encounter a nested macro call or skip. In this case it 'goes down a level' and searches for the delimiters of the nested construction; only when the closing delimiter of this is found does it return to the original search. Thus, if one wrote the nonsense line:

```
        72)   LET A <=> = UNSTACK L + 1;  + 7
```

the second equals and the second plus would be delimiters of the `LET` macro, since the first equals is within a skip and the first plus is within a call of the `UNSTACK` macro. The nonsensical output from the above has not been shown.

Exclusive delimiters

The reader may wish to skip this Section initially, as it describes a rather complicated feature of ML/I.

It is sometimes convenient to define a macro or skip so that, after the macro or skip has been processed, scanning resumes *at* rather than *beyond* the closing delimiter; a closing delimiter with this property is called an *exclusive delimiter*. An exclusive delimiter is specified as such by placing the imaginary node `NO` (`N` zero) after it.

For example, if it is desired to delete all lines commencing with an asterisk, this can be done as follows:

```
        73)   MCSKIP , NL WITH * NL NO
        74)   AAA**OK
        75)   * THIS WILL VANISH
        76)   * SO WILL THIS
        77)   END
    Output)   AAA**OK
    Output)   END
```

Here the skip name is a newline immediately followed by an asterisk, and the closing delimiter is the next newline. However, when the skip has been processed, scanning resumes at this closing newline, so this newline is free to form part of a further skip name if an asterisk occurs at the start of the next line.

In fact, defining newline as part of a macro name or skip name, though often useful, has many potential pitfalls and it is best avoided by the novice (it will also, for the reader who is actually using ML/I interactively, tend to upset the interrelation of input lines with output ones since ML/I will need to keep 'looking ahead' a line). It is generally better to use the 'startline' facility, which is described in the ML/I User's Manual.

Operation macros

All the built-in ML/I statements like `MCDEF`, `MCSKIP`, `MCSET`, `MCGO` etc., have the generic name of *operation macro*. Operation macros are analogous to ordinary user-defined macros in the way they are scanned, but they are different in that they perform some predefined system action instead of effecting a replacement (they generate no actual output in most cases). There are many operation macros that have not been covered in this Guide. One example is `MCNOTE`, which prints a message together with the current line number, and is useful for scanning documents and for generating error messages.

Replacement

Note that macro replacement takes place everywhere in the text scanned by ML/I (except within skips). For example, it takes place within arguments to operation macros (e.g. in structure representations) and within inserts. In fact, ML/I contains virtually no restrictions on what one can do and where one can do it. This in many ways contributes to the power of ML/I, but it does mean that you are not restricted as to the depths of the logical mires that you can get into, nor in the machine time that you can use trying to make ML/I do things it was not designed for.

Concluding remarks

We have now come to the end of this Guide. Hopefully, you have reached a stage where, although your understanding of ML/I is of necessity rather patchy and in some cases superficial, you can still use ML/I in some simple applications and perhaps in some not-so-simple ones too. After having some experience with ML/I, you may wish to refer to the User's Manual to fill in some of the gaps in your knowledge.