An ML/l tutorial

R.D. Eager

May 2016

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. However, there are many, many other possible applications.

There already exists a reference manual for ML/I (*ML/I User's Manual*), and indeed a document for beginners (*A simple introductory guide to ML/I*). However, it was felt that a different approach might be more appropriate for some users, and this tutorial tries to take such an approach, working 'from the ground up'.

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 (this approach was also used in *A simple introductory guide to ML/I*). 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. There is an Appendix to the *ML/I User's Manual* for each implementation of ML/I; it contains general operating instructions, as well as a list of valid layout keywords (this will be needed later).

Getting started

The basic function of ML/I is to copy from input to output, possibly making some changes along the way. Initially, ML/I has received no instructions, so (apart from notable exceptions explained in a moment) it will blindly copy everything unchanged. For example:

```
       1)   This is my first line
Output)   This is my first line
       2)   And this is my second
Output)   And this is my second
```

So far, as can be seen, ML/I appears to be nothing more than a rather overweight program for copying files.

However, ML/I has built in instructions associated with about 20 special words. One of these is the word `MCNOTE`, which is intended for generating messages:

```
       3)   MCNOTE Hello, world
Output)
Output)   Hello, world
Output)
Output)   detected in
Output)   line 4 of source text
```

The immortal phrase is incorporated into a message, accompanied by some text which explains the context in which the `MCNOTE` keyword was found. ML/I does not detect the whole instruction until it has read the whole of the third line, hence the mention of line 4. It is worth noting that the keyword must be all in upper case, or it will not be recognised:

```
       4)   McNote Hello, world
```

```
Output)   McNote Hello, world
```

Now, let's look in detail at what is happening here. First, note that ML/I generated the message as soon as the Enter (or 'carriage return') key was pressed at the end of line 3; in other words, the end of line terminated the message that was typed in following `MCNOTE`. In fact, it is convenient to consider all input to, and output from, ML/I as consisting of a continuous stream of characters, with a notional 'newline' character separating the lines. So, for example:

```
This is line 1
Line 2
One more line
```

would be considered as:

```
This is line 1 newline Line 2 newline One more line
```

Now, ML/I read line 1 of the input, but did not see any of the special instructions that it recognised, so it copied what it saw straight to the output. The same thing happened with line 2.

When reading line 3, ML/I recognised the special word `MCNOTE`. At this point, it started looking for the end of the line (more correctly, for the 'newline' character). When it was found, all of the text between the `MCNOTE` and the newline was taken to be the text of the required message. This was output, as shown above, accompanied by context information.

An important point is that everything between `MCNOTE` and the newline was absorbed by ML/I, and as such did not appear in the output. As it happens, `MCNOTE` generated a message, but nothing else was copied across. Once this line has been processed, ML/I reverts to its normal text copying mode:

```
     5)   Another line of text
Output)   Another line of text
```

As previously mentioned, there are several other special keywords recognised by ML/I; they all start with `MC`, to avoid confusion.

Now let us look at another instruction recognised by ML/I; this is `MCINS`. It is usually followed by two characters, chosen by the user; as before, `MCINS` must be in upper case:

```
     6)   MCINS % .
```

Notice that no output was generated by this line. ML/I expects `MCINS` to handle everything up and including to the next newline, and so the line is absorbed, but there is a side effect, which is the purpose of using `MCINS`. It tells ML/I that, in future, everything between `%` and `.` in its input is to be treated as something called an *insert*. Inserts are used, *inter alia*, to insert values of entities called *macro variables*. Macro variables have various names, but they are all referred to by a single letter (`P`, `S`, `T` and sometimes `C`) followed by a number. In this example, we will use a predefined variable called `S2`:

```
     7)   %S2.
Output)   7
```

`S2` always contains the current input line number, so its value is inserted in place of the `%S2.`. The newline after `%S2.` means nothing special to ML/I, so it is just copied across to the output after the `7`.

Before getting down to some terminology, let us try one more example, using another keyword, `MCDEF`: this defines a replacement for a piece of text:

```
8)   MCDEF Robert AS Bob
```

Once again, `MCDEF` must be in upper case, and so must the keyword `AS`, which separates the 'old' and 'new' values. Also notice that, once again, the whole of line 8 is absorbed by ML/I and does not appear in the output; this is the usual situation, so it will not be referred to again.

What we have done here is define a replacement for the text `Robert`; every time that this word is seen in the input, it will now be replaced by `Bob`, as long as it is an exact match:

```
9)   Robert wrote this
Output)   Bob wrote this
10)   Roberta did not help
Output)   Roberta did not help
11)   ROBERT is a different word
Output)   ROBERT is a different word
```

Notice that `Roberta` did not match, as it is a different word; the same is true of `ROBERT`, as it is in upper case.

Terminology

It is now time to give some formal names to some of the things we have seen. We already know the terms *insert* and *macro variable*, of course.

In the example in lines 8 and 9, we defined a *macro* with the name `Robert`. We also defined its *replacement text* to be `Bob`. By doing so, we added `Robert` to the list of instructions recognised by ML/I, the action specified being to output the word `Bob`. Once again, `Robert` was absorbed by ML/I; if we wanted to retain it in the output, we would have had to say so. However, we never mentioned anything about newlines, so the newline at the end of line 9 was copied to the output unchanged, and appeared after `Bob`.

So far, what we have done could be accomplished easily by even a basic editing program. The power of ML/I lies in the more complex macro definitions that it can handle, so we should show a slightly more complex example. Before doing so, we need to set up something called a *skip*, using another keyword, `MCSKIP`:

```
12)   MCSKIP MT,<>
```

The meaning of this will be explained later; for now, we will just use it. Now we can define another macro:

```
13)   MCDEF Promote to NL AS <The %A2. is now %A1.!
14)   >
15)   Promote Robert to Managing Director
Output)   The Managing Director is now Bob!
```

Several new things have been introduced here. The macro has been named `Promote`, but the word `to` was placed after it, followed by the keyword `NL` (which is a built-in ML/I keyword, and means 'newline'). These three words form what is called a *structure representation*, which is a kind of 'picture' of what ML/I is to look for. In essence, it says:

a. Define `Promote` as a macro name.

b. When `Promote` is seen, look for the word `to` and remember everything in between.
c. When `to` is found, look for the end of a line (a newline) and remember everything in between.
d. Replace everything between `Promote` and the newline with the text between `<` and `>`.
e. The whole process of replacement is termed a *call* of the macro named `Promote`.

Before looking in detail at the replacement text for `Promote`, let us look at the components of the call. The boundaries of the various parts of the call are set by `Promote`, `to` and the newline; they are known as the *delimiters* of the call, and are numbered:

- `Promote` is delimiter 0 (ML/I counts from zero), and is sometimes known as the *name delimiter*.
- `to` is delimiter 1.
- The newline is delimiter 2, and is also known as the *closing delimiter*.

All delimiters *except* delimiter 0 are also known as *secondary delimiters*. The way in which delimiters are laid out is defined by the order in which they are listed when using `MCDEF`, as in line 13 above, and the list (as mentioned earlier) is called a *structure representation*, or, more fully, a *delimiter structure representation*. These lists can be made very complex if desired, and this makes macro definition and recognition very flexible.

Going back to the call of `Promote` on line 15, the text `Robert` appeared between delimiters 0 and 1; this is called *argument 1*. Likewise, the text `Managing Director` appeared between delimiters 1 and 2, and is called *argument 2*. In general, it is the case that argument N will appear between delimiter N-1 and delimiter N.

If we look at the definition of `Promote` in lines 13 and 14, it is fairly obvious what is going on. The text `%A1.` means "insert argument 1 here", and the text `%A2.` means "insert argument 2 here". So, inserts are used not just for inserting the values of macro variables, but for inserting the values of arguments. In addition, when an argument is inserted it is normally *evaluated*; that is, it is scanned for further macro calls. In the case of argument 1, this means that `Robert` is replaced by `Bob`, as specified back on line 8. If we did not want this to happen, we would write `%WA1.` instead of `%A1.`, meaning that we wanted the argument "as written", without any further macro replacements.

A last observation concerns the newline that was the closing delimiter. This was absorbed during the scan of the whole call of `Promote`, so we needed to add one back at the end of the replacement text; this is why `>` appears at the start of line 14 rather than at the end of line 13.

Skips

It is time to explain the use of `MCSKIP`, and in particular what was typed in line 12. However, some simpler examples will be considered first.

ML/I is very enthusiastic; it will attempt a scan and macro replacement at almost every conceivable opportunity. Sometimes this is a nuisance, so there is a way to tell ML/I to copy text "as is" regardless of current macro definitions. This mechanism uses something called a *skip*. In its simplest form, a skip merely deletes some text, like this:

```
16)   MCSKIP Delete ;
17)   Delete this and this; but not this
```

```
Output)    but not this
```

which should need no further explanation, except perhaps to point out that this skip is terminated by a closing delimiter of semicolon, not the end of a line. However, it is often useful to be able to keep the intervening text, but to ignore any macro calls in that text. To do so, the skip is given the *text option* by preceding its structure representation with the key letter `T`, separated by a comma:

```
    18)   MCSKIP T, [ ]
    19)   [This was really done by Robert]
Output)   This was really done by Robert
```

Here, the skip name happens to be `[` rather than a word; this use of a punctuation character as a skip name is quite legal and frequently very useful. Notice that the text in line 19 was copied across unchanged (`Robert` was not replaced by `Bob`), and the skip delimiters (`[` and `]`) were just absorbed. Once again, the newline after the `]` was not part of the skip and was copied unchanged.

Sometimes, we might want to copy across the skip delimiters as well, perhaps in the context of comments in a programming language (we would want the comments left unaffected by macros, so we need a skip, but the comments should not be affected by the skip either). In this case, the *delimiter option* is used

```
    20)   MCSKIP DT, COMMENT ;
    21)   COMMENT Robert wants comments left alone;
Output)   COMMENT Robert wants comments left alone;
```

The delimiter option is specified with `D`, and just concatenated with the `T` for the text option. One could of course omit the `T` to delete the intervening text and leave the delimiters, if desired.

There is one last skip option to be explained. Consider the situation when ML/I has recognised the start of a skip, and is scanning for its closing delimiter. If it found a macro name, or another skip, for example, it would not try to match them with any nested closing delimiters first, and this might cause a problem if these were the same as those for the outermost skip. Hence, a skip can have a *matched option*, introduced by `M`. Now, if ML/I encounters a skip and starts scanning for its closing delimiter, and it finds the start of (say) another skip, it stops looking for the closing delimiter of the "outer" skip and starts looking for the closing delimiter of the newly discovered one. When it is found, it resumes the scan for the outer skip. Nothing else is done with the inner skip; it is recognised only for the purpose of matching its delimiters first.

It is often useful to tell ML/I not to evaluate some text straight away, but to do so next time it is scanned; this is applicable particularly to the replacement text of a macro. The macro `Promote` defined in lines 13 and 14 is a case in point, and uses a skip named `<` with both the matched and text options set (see line 12). If this skip were not used, ML/I would try to evaluate `%A1.` and `%A2.` at the time the macro was defined; however, at this point there are no arguments as there is no macro call in progress, and ML/I would report an error. The skip causes ML/I to store the replacement text unchanged (removing the enclosing `<` and `>` because the delimiter option is not set on the skip). When the macro `Promote` is actually called, `%A1.` and `%A2.` are no longer enclosed in a skip and are thus evaluated normally. The characters `<` and `>` are commonly called *literal brackets* and are frequently used in this way. They have a secondary use in the definition of `Promote`, since

normally the definition (and therefore the replacement text) would be terminated at the end of line 13 (as is usual for `MCDEF`); the literal brackets "hide" the newline and allow it to be included in the replacement text, meaning that the end of line 14 terminates the `MCDEF`.

It should be apparent that if nested sets of literal brackets are used, one pair will be removed each time the text is processed; there are occasions when this is desirable.

One last piece of terminology should be mentioned. Loosely, the set of things that ML/I recognises while scanning is known as the *environment*, although in fact other entities such as macro variables are also part of the environment. The contents of the environment will change as new macros, inserts, skips etc. are defined, and as macro calls start and finish.

Operation macros

So far, we have seen how macros work; essentially, each has a name which is recognised specially by ML/I while it is copying from the input to the output. The effect of a macro is usually defined by the user; it might trigger a replacement, or it might just absorb the text of a macro and have some kind of side effect.

Keywords such as `MCDEF`, `MCINS`, etc. are also macros. The difference is that they are built in to ML/I, usually have a null replacement text (i.e. they do not generate any output), and mostly have some kind of side effect. For example, the side effect of `MCDEF` is to define a new macro. These built-in macros are known as *operation macros*, and there are usually twenty of them.

Apart from the differences noted above, operation macros work in a similar way to any other (user defined) macro. They have defined structure representations, can be nested, and so on. Most (but not all of them) have a closing delimiter of newline, which turns out to be very convenient. There are three rough groups of operation macros:

a. *Name environment changing macros* (or NEC macros). These operation macros change the "name" part of the environment, i.e. they add new names (macros, skips, etc.) to it, or (sometimes) suppress them. There are fourteen NEC macros.

b. *System functions.* These operation macros provide useful facilities which are often employed within the replacement text of a macro; they allow operations such as the selection of substrings, and counting the length of a piece of text. There are two system function operation macros; they are the exceptions to the rule that operation macros have a null replacement text, and calls of them do not change the environment either.

c. *Other operation macros.* The operation macros that do not fall into the above groups are used to create new macro variables, make decisions, and generate messages. There are four of these operation macros. Some of them change the environment (by creating macro variables) and some do not.

When ML/I is started, only the operation macros are usually recognised. All other text is copied from input to output unchanged. Because all macros, whether operation macros or user macros, are treated the same way, calls of operation macros can occur at any time. In particular, there is no need to define all the user macros before processing an input file; indeed, one of the powerful features of ML/I is that new macros can be defined at any time, even inside the replacement text of another macro.

In summary, operation macros only differ from other macros in that:

a. They are built in to ML/I, and thus always defined.

b. All but two of them (the system functions) have null replacement text.

c. Most of them have side effects on the environment.

A full list of operation macros can be found in Chapter 5 of the *ML/I User's Manual*.

Atoms

So far, all construction names (macros, skips and inserts) have been single words or punctuation characters. It is time to formalise this, and to show how more complex names can be used.

On line 10 of our example input, we saw that the name `Roberta` was not recognised, even though a macro named `Robert` had been defined. This is where ML/I differs from a conventional text editor, since the former works in entities called *atoms*, and the latter usually just recognises sequences of characters.

An atom is generally defined as 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. This turns out to be a very convenient way of handling input.

It is also worth noting that characters such as space, tab and newline are usually referred to here as *layout characters*.

So far, all of the construction names we have used have been single atoms, e.g. `Robert`, `Promote`, `[` and `<`. However, a construction name can be made up of a number of atoms, joined in specified ways. Atoms that are not punctuation characters must be separated by some other atom, otherwise they would coalesce and form a different atom altogether.

The most common requirement is probably a macro name made up of two or more "words", for example `Promote` and `immediately`. To define this, we need to add a space in between the two words. ML/I ignores layout characters inside structure representations, so we use a *layout keyword* to specify an intervening space:

```
22)   MCDEF Promote WITH SPACE WITH immediately
23)   NL AS <Shock! Horror!
24)   >
```

Notice the use of the keyword `WITH` to join the atoms together. Also note that we started a new line before the `NL` keyword; there was no need to do this, but nothing strange happens because ML/I ignores layout characters in structure representations (i.e. before the `AS`). If we started a new line *after* the `NL`, it would be taken as the end of the replacement text, which is delimited by `AS` and a newline (the end of line 23 is not taken as the end of the replacement text, because it is inside the literal brackets).

The above is a little inflexible; it requires exactly one space between `Promote` and `immediately`. To allow any number of spaces (but at least one), we could use the `SPACES` keyword:

```
25)   MCDEF Promote WITH SPACES WITH soon
26)   NL AS <More like it...
27)   >
```

Actually, this can be abbreviated using the `WITHS` keyword, in which case we could have written:

```
MCDEF Promote WITHS soon ...
```

There are various other layout keywords; see the appropriate Appendix to the *ML/I User's Manual* for a list of those accepted on the particular implementation of ML/I that you are using.

There is no need to put spaces between atoms if one is a punctuation character; for example, line 5 above is usually written without the space between the `%` and the `.`:

```
MCINS %.
```

Linking of atoms is not restricted to construction names; it can be used for secondary delimiters as well.

More about delimiter structures

Until now, we have considered macro definitions (and indeed skips and inserts) with fixed delimiter structures; that is, the number of delimiters is completely defined when the construction is set up, and the delimiters are also fixed.

One of the most useful features in ML/I is the ability to define variable delimiter structures. The variation comes in several forms, for example:

a. Different (alternate) names for name delimiters and also secondary delimiters.
b. Variable numbers of delimiters, some being repeated an arbitrary number of times.
c. As a consequence of b): optional delimiters.

We will now examine how a variable delimiter structure is specified. Any delimiter may have any number of alternate forms; the list of alternatives is bracketed by the keywords `OPT` and `ALL`, and the possibilities are separated by the keyword `OR`. So, we might say:

```
    28)   MCDEF OPT William OR Mary ALL AS One of the twins
    29)   William
Output)   One of the twins
    30)   Mary
Output)   One of the twins
```

This defines a single macro with two alternate names. It is also possible to have multi-atom alternates:

```
    31)   MCDEF OPT Twin WITHS one OR Twin WITHS two ALL
    32)   AS One of the twins
    33)   Twin one
Output)   One of the twins
```

A multi-atom delimiter may not be split across options, so something like:

```
Twin WITH OPT one OR two ALL
```

is not permitted; each delimiter must be written in full. Secondary delimiters can also have alternatives, and these are written in just the same way.

Alternate names are useful, but it is even better if the number of delimiters can vary. For example, ML/I might be required to handle a macro called `Demote`, which took an arbitrary number of arguments which were separated by commas, and terminated by the end of the line. In this case, we need to direct ML/I to consider possible alternatives of "comma" or "newline" for each secondary delimiter, looping back if a comma is found, but finishing if the newline is encountered. Loops in a delimiter structure are specified by *nodes*, which are merely the letter `N` followed by a number. A node acts both as a label, and as

a "go to", within a delimiter structure; the context makes it clear which one is intended. So, for the example just described, the delimiter structure would be as follows (we will not type it in just yet):

```
Demote N1 OPT , N1 OR NL ALL
```

Notice that `OPT` is *preceded* by node `N1`, and so it is considered to be "labelled" with node 1. The list of options is as specified (comma or newline), but the comma option is *followed* by a mention of node `N1`; this is considered to be a "go to" to the node label before the `OPT`. Thus, repeated commas are accepted. However, the other alternate, `NL`, occurs at the end of the delimiter structure, so when a newline is encountered, it terminates the delimiter structure. A more formal description of this mechanism may be found in the *ML/I User's Manual*. Nested `OPT ... ALL` pairs are permitted, as well as any number of options.

A variable number of delimiters poses new problems; how to find out how many delimiters are present in an actual call, and how to access them. ML/I provides suitable facilities, of course.

First, every macro call has at least three temporary macro variables available to the replacement text; these are called `T1`, `T2` and `T3`. These can be used for any purpose at all, but they are initialised to useful values; in particular, `T1` is set to the number of *arguments* of the current call. So, if `T1` has the value (say) 3, there will be three arguments (1 to 3) and four delimiters (0 to 3).

Secondly, some decision making mechanism is needed; this is provided by a conditional "go to" statement in ML/I, implemented by the operation macro `MCGO`.

Thirdly, we need some way of labelling parts of the replacement text, to provide a target for "go to"s. ML/I implements labels by using the "insert" mechanism; all labels are numeric, and introduced by the letter `L`. So, to place a label numbered 3 (say):

```
%L3.
```

This "inserts" a label, and provides a target for "go to" statements, but it has a null value, so nothing is sent to the final output.

Next, we need to be able to assign values to macro variables. This is done using the operation macro `MCSET`, as in:

```
MCSET T2 = 99
```

Lastly, we need some way of accessing the arguments via an "index". This is easily done, as macro variables and arguments can be subscripted. For example, if `T2` had the value 3, then `%AT2.` would specify argument 3.

Putting all this together, we can iterate through the arguments, but before doing so it may be useful to define a skip to allow us to add comments:

```
34)   MCSKIP "WITH" NL
```

Now, the `Demote` macro itself. Its purpose, in this example, is to generate one line of output for each person being demoted:

```
35)   MCDEF Demote N1 OPT , N1 OR NL ALL
36)   AS <"" initialise argument counter
37)   MCSET T2 = 1
38)   "" place label and insert argument and text
39)   %L1.Note that %AT2. is to be demoted
40)   "" increment counter
41)   MCSET T2 = T2 + 1
42)   "" remember that T1 contains number of arguments
43)   "" jump back if more to do
44)   MCGO L1 UNLESS T2 GR T1
45)   >
```

Now we can try it out:

```
    46)   Demote Tom, Dick, Harry
Output)   Note that Tom is to be demoted
Output)   Note that Dick is to be demoted
Output)   Note that Harry is to be demoted
```

Each macro call has its own set of temporary variables, and label numbers are also local to each macro, so conflicts do not usually happen. Incidentally, label zero (`L0`) is always valid as the target of a `MCGO`, and forces an immediate exit from the current text (i.e. the current macro call).

System functions

There are two operation macros which are *system functions*. These are particularly useful at macro time (i.e. in the replacement text of a macro).

The first one is `MCLENG`; this is followed by a pair of parentheses, and in fact its structure representation would be

```
MCLENG WITHS ( )
```

The value of a call of `MCLENG` is a minimum-length string of digits giving the length of its argument. Note that the closing delimiter of `MCLENG` is the closing parenthesis, not the newline which is more usual for an operation macro.

```
    47)   MCLENG(ABcDEfg)
Output)   7
```

The second system function is `MCSUB`; this extracts substrings of a piece of text. In this case there are three arguments, separated by commas but enclosed overall in parentheses. The arguments are (in order):

1. The text of which a substring is to be taken.
2. The offset of the starting character of the required substring.
3. The offset of the ending character of the required substring.

The offsets start at 1 for the first character of the text. A zero offset means the last character of the text, -1 means the next to last, and so on, so it is possible to extract substrings from either end of the text without much work.

```
    48)   MCSUB(KLMNOPQR, 3, 6)
Output)   MNOP
    49)   MCSUB(KLMNOPQR, -2, 0)
Output)   PQR
```

Macro variables

Macro variables have been mentioned previously, but this section summarises the various kinds of variables that are supported by ML/I.

Each call of a macro is allocated three integer *temporary variables*, named `T1`, `T2` and `T3`. This number can be increased, at the time the macro is defined, if so desired. For full details of initial values, see the *ML/I User's Manual*; we have already seen that `T1` is preset to the number of arguments to the macro call.

There are also a number of integer *permanent variables*, named `P1`, `P2`, `...`, etc. The number of these initially allocated, and their initial values, is documented in the Appendix for each implementation. Permanent variables are globally accessible and can be used for any purpose. Additional permanent variables can be created by calling the `MCPVAR` operation macro.

Each implementation has a fixed number of integer *system variables*, usually at least nine, named `S1`, `S2`, `...`, etc. They are used to control the way ML/I works, and their function is documented in the *ML/I User's Manual*, and also in each Appendix. They are also globally accessible.

Lastly, more recent implementations support *character variables*, named `C1`, `C2`, `...`, etc. These store character strings, and have a fixed maximum length. Initially there are no character variables in the environment; users can create them, and define their maximum length, by calling the operation macro `MCCVAR`. Once created, they are globally accessible.

Startlines

It will be apparent by now that ML/I is essentially character based rather than line based; in many ways this adds to its power, but can be inconvenient when processing input that is line oriented. ML/I addresses this problem by the use of *startlines*.

A startline is an imaginary character that is (optionally) inserted at the beginning of every input line. If a startline is sent to the output, it is silently discarded. There is a layout keyword (`SL`) for a startline character, so a startline can be part of a macro name.

Startlines sometimes get in the way, particularly when defining macros that may use them; thus, they are not inserted unless specifically requested. To "turn on" startlines, set system variable `S1` to one (its initial value is zero).

As an example, suppose it was desired to delete all lines starting with an asterisk, while retaining intact other lines containing asterisks somewhere else in the line. The following skip might be used:

```
    50)   MCSKIP SL WITH * NL
```

Now if we try this out:

```
    51)   * This is a test
Output)   * This is a test
```

Nothing has happened to the line, because the asterisk was not preceded by a startline character. Now we can turn startlines on and try again:

```
      52)   MCSET S1 = 1
      53)   * This one should disappear
      54)   But this line * should be intact
  Output)   But this line * should be intact
```

Notice that the line *starting* with an asterisk is suppressed this time, but the line *containing* an asterisk is preserved, because the asterisk is not immediately preceded by the startline (which is immediately before `But`).

In summary, startlines are a good way of addressing the relative lack of line orientation in ML/I.

Hints and tips

This section contains some general hints and tips about using ML/I.

- 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; this is why the replacement text of a macro is usually enclosed in literal brackets. 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 the user also needs to be careful.
- If it is necessary to place two non-punctuation atoms (e.g. `PIG` and `DOG`) next to each other for some reason, they must be separated by something that has a null replacement text. One good thing to use is an empty pair of literal brackets (`<>`), and another is a dummy label (e.g. `%L99.`).
- Use of an ML/I keyword as a delimiter of a construction is legal, but difficult to set up. It is possible to alter a keyword (but not the name of an operation macro) to something else, using the operation macro `MCALTER`. For example, to define a macro with secondary delimiter `AS`, one could use:

  ```
  MCALTER AS TO IS
  MCDEF NAME AS NL IS ...
  MCALTER IS TO AS
  ```

  The first line alters the secondary delimiter of `MCDEF` from `AS` to `IS`. The second line defines a macro with secondary delimiters `AS` and "newline", with the replacement text after the new keyword `IS`. The third line wisely puts the keyword back to normal, since once the definition has been set up there is no further problem with keyword clashes.
- ML/I normally removes leading and trailing spaces from arguments. It is possible to insert an argument while preserving those spaces, by using `B` instead of `A` in the insert, for example:

  ```
  %B1.
  ```
- Arguments to operation macros have leading and trailing spaces removed. If spaces are to be significant, they should be enclosed in literal brackets. For example, to insert the value of T1, padding it with leading spaces to a width of six characters, we could use:

  ```
  MCSUB(<      >%T1., -5, 0)
  ```

Conclusion

We are now at the end of this tutorial. In it, we have covered many of the common features of ML/I, but much has been omitted and some areas have been slightly over-simplified. It takes time and practice to realise the full power that ML/I can wield!

For further tuition, work through *A simple introductory guide to ML/I*; it is complementary to this tutorial, so some material will be repeated (which is no bad thing).

For reference material, and much more detailed information about such topics as the scanning process, see the *ML/I User's Manual*. Do not be put off by its size, nor by the apparently strange order in which topics are presented (operation macros are not covered until well into Chapter 5). There are reasons for this, and in fact the entire manual is a marvel of logical presentation and lucidity.