Part II — A Critique of ML/I

8 A Critique of ML/I

It is the purpose of this Chapter, which is the only Chapter in Part II of this discussion, to perform the following functions:

a. to examine what is new about ML/I and what it has derived from other macro processors.

b. to state the deficiencies of ML/I and how they might be remedied.

c. to consider briefly how ML/I works.

d. to mention some applications of ML/I.

The *ML/I User's Manual*, which describes ML/I in full detail, is included as Appendix A.

8.1 Innovations in ML/I

The main feature of ML/I, by which it stands or falls, is its facility for *delimiter structures*. This facility has never been offered before. Other attempts to accomplish the same end, namely to allow the user to design his own notation for macro calls, have been discussed in Chapter 3. Broadly speaking the advantage of ML/I over systems such as XPOP and LIMP, which have different notational mechanisms, is that ML/I is suitable for applications where it is required to nest macro calls within macro calls or to have macros with a large number of possible forms, each requiring a different replacement. XPOP and LIMP, on the other hand, are more suitable than ML/I in applications where macro calls can conveniently be written one to a line and are hence never nested within one another.

As well as making it possible for users to communicate with the machine in their own terminology, the notational flexibility made possible by delimiter structure has another advantage. This advantage is that it makes ML/I notation-independent. As was explained in Chapter 3, notation-independence opens up a new vista of applications for macro processors in such fields as editing and symbol manipulation. Every notation-independent macro processor has its failings, but ML/I has considerable advantages over the "one call to a line" macro processors for a large number of applications.

The basic concept of a macro as in ML/I is not new and in this respect ML/I belongs to the family of macro processors consisting of GPM, TRAC and 803 Macro-generator. Of these ML/I has leaned most heavily on GPM.

Another new feature of ML/I is its *skips*. These are a generalisation of the literal brackets used in GPM. Skips allow the user to inhibit macro replacement within contexts defined by himself.

Inserts in ML/I are not new but combine into one mechanism a number of facilities that have been available in macro processors for some time.

Macro-time statements in ML/I are also not new and represent a cross between the ideas of GPM and the PL/I Macro Processor.

8.2 Defects of ML/I

The most obvious defect of ML/I is its slowness. This is nothing unusual in a macro processor or indeed in any text manipulation processor, but it does limit its usefulness for some applications. The most wasteful activity in ML/I is the interpreting of the macro-time statements `MCSET` and `MCGO` every time they are performed. However if these statements were to be compiled, it would necessitate some restrictions in ML/I which, though not taxing in practice, would be aesthetically unpleasing and would make the User's Manual longer and more complicated.

ML/I is clumsy in dealing with text where macros are written one to a line and not in prefix notation, but there exist macro processors, for example LIMP and WISP, that are very good for this type of application and ML/I fairly neatly complements the facilities offered by these macro processors, since these latter are clumsy in dealing with text where macro calls are not written one to a line.

Another defect of ML/I is its lack of character variables, or put another way, its lack of a facility to redefine a macro and throw the old version away. This facility would not be very difficult to add to ML/I, at least in a restricted way.

Defects that are more defects of individual implementations than of ML/I itself are the lack of a facility to divide the output into separate streams, the lack of a library facility and the lack of a convenient facility for overriding all the operation macro definitions within certain contexts, thus making ML/I capable of editing text involving operation macro calls, i.e., of editing instructions to itself.

It is often said that a language should be such that what is easy to write is easy to process and those operations which require complicated and time-consuming processing should be complicated to write, thus forcing the user, through his own laziness, to use the language in the optimal way. It can, therefore, be held as a defect of ML/I that it is very easy to define more and more macros, involving deeper and deeper nesting, with the result that macro processing becomes excessively slow. In many practical cases the slowness of ML/I in performing a task has been due to the unnecessary use of its most time-consuming facilities and these applications can be speeded up by an order of magnitude by judicious changes.

Lastly an irritating feature of ML/I is that it is necessary for the user to be very careful with layout characters (i.e., spaces, newlines, etc.). This applies to all general-purpose macro processors and is something the user must live with. However in future versions of ML/I the problem will be slightly alleviated by using keywords to stand for these characters when they are needed in structure representations and ignoring the layout characters themselves. With hindsight, it is now clear that it would have been better to use a newline rather than a semicolon as the standard closing delimiter for operation macros.

8.3 The Logic of ML/I

Like most other general-purpose macro processors, ML/I makes only one pass through the source text. The input routine for ML/I normally reads the source text character by character or line by line, and source text is not retained inside the machine after it has been

evaluated. There are only a few features of the logic of ML/I that are worthy of special comment.

Firstly it should be mentioned that one important factor in the development of ML/I has been that it should remain small enough to operate on the PDP-7. Thus features have not been added unless they represent solid improvements and could be implemented fairly easily. This smallness of ML/I means that ML/I can, when appropriate, act as a co-routine with other processors. Mr. T.C. O'Brien of Honig Associates states, in a private communications, "I view the macro processor as belonging in the 'card reader' or 'printer' category rather than as a preprocessor. Furthermore, it should be possible to stack several different translators, one behind the other, without the use of an intermediate file". This is a view I fully support and is a powerful argument for a macro processor to be small.

The requirement for compactness was one reason why list processing techniques were not used in the logic of ML/I. Instead dynamic storage allocation is provided by two stacks, one stack working forwards and the other backwards (see Section 2.8 of Appendix B). Character strings are stored contiguously. This method is much faster than list processing, especially on machines that have instructions for moving about or comparing contiguous blocks of data, but it does, of course, make ML/I hard to extend in some directions. Many of the most useful features in LIMP, for example, depend on the use of list processing techniques. Nevertheless the decision not to use list processing techniques in ML/I has not been regretted.

It has been found from tests on the PDP-7 implementation of ML/I that the most time-consuming activity in ML/I is the comparison of each atom of the scanned text with all possible macro names. This is in spite of the use of hashing techniques (see Section 6.2.3 of Appendix B). As a result of these tests, the speed of the PDP-7 implementation has been improved by enlarging its hash-table, and it is certainly wise to make hash-tables as large as is reasonably possible, subject to the storage available. With a hash-table of 64 entries each atom will, in an average job, need to be compared with only one or two possible macro (or other construction) names and this sort of overhead is quite reasonable. However the hash-tables should not be made excessively large since the logic of ML/I is such that there may be as many as four hash-tables in existence at any one time, and, in exceptional applications, more than this.

Nothing else in the logic of ML/I is unusual or worthy of special comment.

8.4 Uses of ML/I

Some of the uses to which ML/I has been put are indicated in Chapter 7 of the User's Manual and in the published paper describing ML/I [4]. The uses outlined in these publications include the implementation of a language to describe fields within data structures, the compilation of arithmetic expressions, the partial conversion of FORTRAN IV to FORTRAN II, and a number of applications in text generation, searching and systematic editing. Further uses of ML/I have been in conventional macro-assembly, the implementation of application-oriented languages, symbolic differentiation, the extension of Titan Autocode (see Hopewell [22]), the conversion of Elliott ALGOL I/O statements to those of another ALGOL dialect, and the conversion of a program from PDP-7 Assembly Language to Titan Assembly Language.

As a result of the published paper on ML/I, over sixty computer installations have expressed interest. These include computer manufacturers, industrial organisations, universities and research establishments and between them they cover a very wide range of applications.