Here an unconditional statement plus a conditional expression has become a conditional statement.

Fortunately both of these situations have been ruled out by Section 4.7.5.2.

Conclusion

For centuries astronomers have given the name ALGOL to a star which is also called Medusa's head. The author has tried to indicate every known blemish in [2]; and he hopes that nobody will ever scrutinize any of his own writings as meticulously as he and others have examined the ALGOL Report.

RECEIVED JANUARY 1967; REVISED JULY 1967

REFERENCES

1. NAUR, P. (Ed.) Report on the algorithmic language ALGOL 60. *Comm. ACM 3* (1960), 299–314.
2. NAUR, P., AND WOODGER, M. (Eds.) Revised report on the algorithmic language ALGOL 60. *Comm. ACM 6* (1963), 1–20.
3. ABRAHAMS, P. W. A final solution to the dangling else of ALGOL 60 and related languages. *Comm. ACM 9* (1966), 679–682.
4. MERNER, J. N. Discussion question. *Comm. ACM 5* (1964), 71.
5. KNUTH, D. E. On the translation of languages from left to right. *Inf. Contr. 8* (1965), 607–639.
6. DIJKSTRA, E. W. Letter to the editor. *Comm. ACM 4* (1961), 502–503.
7. LEAVENWORTH, B. M. FORTRAN IV as a syntax language. *Comm. ACM 7* (1964), 72–80.
8. KNUTH, D. E., AND MERNER, J. N. ALGOL 60 confidential. *Comm. ACM 4* (1961), 268–272.
9. INGERMAN P. Z., AND MERNER, J. N. Suggestions on ALGOL 60 (Rome) issues. *Comm. ACM 6* (1963), 20–23.
10. RANDELL, B., AND RUSSELL, L. J. *ALGOL 60 Implementation*. Academic Press, London, 1964.
11. NAUR, P. Questionnaire. *ALGOL Bulletin 14*, Regnecentralen, Copenhagen, Denmark, 1962.
12. B5500 Information processing systems reference manual. Burroughs Corp., 1964.
13. VAN WIJNGAARDEN, A. Switching and programming. In H. Aiken and W. F. Main (EDS.), *Switching Theory in Space Technology*, Stanford U. Press, Stanford, 1963, pp. 275–283.

The ML/I Macro Processor

P. J. BROWN
University Mathematical Laboratory, Cambridge, England

A general purpose macro processor called ML/I is described. ML/I has been implemented on the PDP-7 and I.C.T. Atlas 2 computers and is intended as a tool to allow users to extend any existing programming language by incorporating new statements and other syntactic forms of their own choosing and in their own notation. This allows a complete user-oriented language to be built up with relative ease.

Introduction

A macro is basically a means of extending an existing programming language, called the *base language*, by introducing a new syntactic unit which is describable in terms of the existing syntactic units of the base language. This paper is concerned only with macro processors which operate on text and act as a preprocessor to the compiler for the base language. Most existing macro processors have a fixed base language, usually an assembly language, and the macro processor and the base language processor are regarded as a single piece of software, as exemplified by the term "macro-assembler." One of the best known macro-assemblers is Macro FAP [1]. Most macro-assemblers require that macros expand into a series of statements; it is not possible, for instance, to use a macro to generate the address field of an instruction.

As a very elementary example of the use of a typical macro-assembler, assume that a user wished to introduce a single statement that would move an item from one storage location to another on a machine that required two instructions to achieve this. To do this he would define a macro, give it a name, MOVE, say, and define the two requisite instructions as the *replacement text* of the macro. Having defined his macro, the user could then treat MOVE as if it were an assembly language statement, and the macro processor would expand this into the two given instructions. The MOVE statements are called *macro calls* and have form:

MOVE *argument* 1, *argument* 2

In the typical macro-assembler the syntax of macros is very rigid. Each line of input text must be either a statement in the base language or a macro call. The arguments of macro calls are separated by a fixed delimiter, the comma, and the *closing delimiter* (the symbol used to indicate the end of a call) is typically either a space or the end of a line.

Several macro processors with more general properties than the basic macro-assembler have been produced. Among these are XPOP [2], WISP [3], LIMP [4], GPM [5] and TRAC [6]. These have extended the basic concept in two main ways. Firstly, the user has been allowed to define his own notation for writing macro calls. This cotation might, for instance, be something approaching the English Language or alternatively might be close to the language of algebra. Secondly, the macro processor has been made independent of the base language and its processor. In this case the same macro processor can act as a preprocessor to any number of different languages, and there is no restriction on the syntactic forms of the base language which may be expanded.

ML/I incorporates both of these extensions and is intended to allow the user to extend any language, using his own notation. The only restriction on notation is that macro calls must commence with the macro name. ML/I

recognizes a macro by the occurrence of its name and a macro call is taken as the text from the macro name up to its closing delimiter. This contrasts with macro processors such as WISP and LIMP where macro calls are recognized by comparing each line of input text with a number of templates.

Main Features of ML/I

This paper will not attempt to give a detailed description of ML/I but rather will describe its main features. For details the reader is referred to the User's Manual [7]. Firstly, the general form of a macro will be described together with examples of its applications. ML/I allows any sequence of characters to be used for each delimiter. The macro name is regarded as delimiter number zero. Hence the user could specify TO and a semicolon as delimiters of his MOVE macro and write his calls:

MOVE *argument* 1 TO *argument* 2 ;

though this in itself could hardly be claimed as a dramatic improvement. The special feature of ML/I is that the user specifies a *delimiter structure* for each macro which makes it possible for each macro to have any number of alternative patterns of delimiters. The purpose of the delimiter structure is to define the name or names of the macro and to define for each delimiter a *successor* or set of alternative successors. A successor is the next delimiter to be searched for when scanning a call. A closing delimiter may be considered as having a null successor. As an example, consider an IF macro which has alternative forms:

(a) IF *argument* 1 = *argument* 2 THEN *argument* 3 END
or (b) IF *argument* 1 = *argument* 2 THEN *argument* 3 ELSE *argument* 4 END

In the delimiter structure of this macro each delimiter has a unique successor except for the delimiter THEN which has the alternative successors END and ELSE.

The above IF macro could be extended by allowing as the first delimiter a number of alternative relational operators to "equals." This could be done by specifying the set of relational operators as alternative successors to IF. Each relational operator would have THEN as its successor. In order that a meaning can be ascribed to different delimiter names, ML/I contains a facility for placing conditional statements, operative at macro generation time, within the replacement text of a macro. These conditional statements can be used to make the form of the code replacing a macro call dependent on the delimiters used in the call.

Delimiter structures can be designed to allow macros with an indefinitely long list of arguments. Assume, for example, that it is desired to extend further the IF macro by allowing between IF and THEN a whole series of relations connected by, say, AND or OR. This could be achieved by defining the successors of each of the relational operators to be AND, OR, or THEN. The successors of OR and AND would be the set of relational operators, thus causing the delimiter structure to loop back on itself.

ML/I allows macro calls to be nested within the arguments of other macro calls. Hence it would be quite possible to write nested calls of the IF macro. The method of searching for delimiters takes care of nested calls.

Before the more basic details of ML/I are described, a few more examples of its applications will be considered in general terms in order that the reader may get some sort of a feel for the kind of macro that is normally defined.

Example 1. It is possible to design a set of macros that would be useful for referencing individual fields in blocks of data. Thus FIELD and SET macros could be designed that would allow the user to define the fields of his data blocks and then to refer to these fields. His program might read (where, for the sake of clarity, the delimiters of the FIELD macro have been italicized):

FIELD FATHER *IS WORD* 1 *;*
FIELD MOTHER *IS WORD* 3 *;*
FIELD AGE *IS BITS* 6 *TO* 12 *OF WORD* 4 *;*
SET X = AGE(FATHER(MOTHER(Y)));

The action of the FIELD macro would be to set up a macro definition whose name was derived from the first argument of FIELD. Thus the third call of FIELD would define a macro with name "AGE(" and with closing delimiter ")". The replacement text of this macro would generate code to reference the desired field of the designated data block.

Example 2. It is possible to write a macro of form:

LOAD *argument* ;

where the argument is a general arithmetic expression. This macro would generate code to load the value of the arithmetic expression into an accumulator. The argument could be analyzed by defining the character "(" as a macro name with a right parenthesis as its closing delimiter and specifying all the permissible arithmetic operators for the intervening delimiters, of which there could be any number.

Since ML/I is independent of the base language, it is intended that it be used as a common preprocessor to all the compilers and assemblers available at an installation, in the same way as each compiler and assembler might use a common loader. In general it is true to say that the higher level the language, the less need there is for macro facilities. However, even in the most comprehensive high-level languages, macros are useful for introducing statements specially designed for the user's particular field of application.

The above examples have illustrated the primary use of ML/I, namely to allow any existing programming language to be extended to suit a particular user's requirements. This form of extension could be carried to the level where the extended language could be considered as a language in its own right.

The efficiency of code generated by ML/I depends how well the macros are designed. There are macro-time variables and conditional facilities to help eliminate the sort of inefficiencies that occur at boundaries between macros. As regards speed, ML/I would be considerably slower than

a special purpose compiler for a language. Note, however, that it is not intended to be a general purpose compiler. Instead of designing a new language and writing a compiler for it, the idea is for the user to start at the other end, as it were, and extend an existing language to meet his requirements.

The usefulness of ML/I is not confined to language extension. It can also be used for some applications in text editing. For instance, it is now being used as an aid to converting from FORTRAN IV to a dialect of FORTRAN II. Here ML/I performs the transformations of which it is capable, for instance recognizing declarations of logical variables and converting statements involving them into corresponding arithmetic statements. In those cases where it cannot perform the required transformation it places a special marker beside the FORTRAN IV statement.

Introductory Details of ML/I

ML/I is fed some *source text*. It performs some transformations on this text and generates some *output text*, which is, in turn, normally fed to some compiler or assembler. These transformations are specified by *constructions*, which are usually defined at the start of the source text. The most important type of construction is the macro.

The character set of ML/I will vary between implementations but it should contain the letters and numbers and some *punctuation characters*. A punctuation character is any character that is not a letter or number. "Space" or "blank" is treated as a punctuation character and it is usually convenient to assume there is a character called "newline" at the end of each line of text. (This character physically exists if input is from paper tape. In the case of card input it would have to be specially inserted by the input routine.) However, in this paper, for reasons of layout, it will be assumed that "newline" is not a character. ML/I treats text as a sequence of *atoms* rather than as a sequence of individual characters. An atom is defined as a single punctuation character or a sequence of letters and/or numbers surrounded by punctuation characters. Thus the piece of text which follows consists of the six atoms: comma, DOG, space, 23, plus, and C8.

```
,DOG 23+C8
```

Any sequence of atoms may be defined as the name of a construction or as the name of any of the other delimiters. In general every time the name of a macro is encountered during the scanning of text it is taken as the start of a macro call and a search is made for the remaining delimiters. The same applies to other constructions. Thus if DOG were a macro with closing delimiter plus, then the above text would contain a call of it. However, if DO were the macro name, the above text would not contain a call of it since DO is not an atom of the text.

The paragraphs which follow describe the other types of construction in addition to macros.

Inserts

Consider the replacement text of the MOVE macro which has already been used as an example. (This example and some subsequent ones will use PDP-7 Assembly Language. However, it is not assumed that the reader is familiar with the PDP-7 and each instruction will be explained.) The replacement text of the MOVE macro consists of the two following statements:

```
LAC argument 1   /Load accumulator with first argument.
DAC argument 2   /Deposit accumulator at second argument.
```

It is necessary to indicate to ML/I that it must insert the appropriate argument at the appropriate point. This is done by a construction called an *insert* which has a name and a closing delimiter. It will be assumed in the rest of this paper that the character ":" has been defined as an insert name with a period as its closing delimiter. Between the insert name and its delimiter a *designation* is written to indicate what to insert. In particular, "A" stands for argument. The replacement text of MOVE would be written:

```
LAC :A1.
DAC :A2.
```

Several other elements in addition to arguments may be inserted and these are described later.

Skips

It may be that the user does not wish certain occurrences of macro names in his text to be taken to mean the macro is to be called. This might apply, for instance, within comments or character string literals. In this case *skips* are used to inhibit macro replacement within the required context. Thus if the base language were PL/I a skip name "/*" with closing delimiter "*/" might be defined, since these characters are used to enclose PL/I comments. Each skip has an associated set of options which determine whether the skip is to be copied to the output text or deleted during macro expansion. It is possible to copy the delimiters and delete the intervening text or vice-versa. In most applications the user will require a special skip name and closing delimiter called *literal brackets*. The options for literal brackets are set so that the brackets are deleted but the intervening text is copied literally to the output text. It will be assumed in the rest of this paper that the characters "⟨" and "⟩" have been defined as literal brackets. With this assumption, an occurrence of ⟨DOG⟩ in the source text would give rise to DOG in the output text irrespective of whether DOG was a macro name.

Warning Markers

In some applications it is inconvenient to have every occurrence of a macro name outside a skip taken as the beginning of a call. In these cases ML/I can be placed in *warning mode* by defining some atom as a *warning marker*. If ML/I is in warning mode all macro calls must be preceded by a warning marker and all macro names not preceded by a warning marker are treated literally. If ML/I

is not in warning mode, macro names are essentially reserved words and if they are used for any other purpose they must be enclosed in skips. It will be assumed in examples in the rest of this paper that ML/I is not in warning mode.

Text Evaluation

The *environment* consists of all the constructions defined by the user, together with the system macros, which will be described later. The process of scanning a piece of text to expand macro calls and deal with skips and inserts is called *evaluating* the text. The text generated as a result of this evaluation is called the *value*. The value of a piece of text depends, of course, on the environment.

Constructions may be nested in any desired way. When ML/I encounters a macro call, it evaluates the replacement text in the same way as it evaluates the source text. The replacement text may itself contain macro calls and recursion is permitted. The arguments of macros are evaluated when they are inserted rather than when the call containing them is scanned. Thus they are "called by name" rather than "called by value." This fact is useful if ML/I is generating assembly code as it is possible to examine the context in which the code is to be placed and generate optimal code accordingly.

Macro Variables

ML/I contains some "macro-time" integer variables, i.e., variables operative during macro generation. There are facilities for the user to perform arithmetic on these variables and to insert their values into his text. There are two types of macro variables as follows.

Permanent variables. These are called P1, P2, etc. They are reserved at the start of a program and remain in existence throughout. They have global scope.

Temporary variables. These are called T1, T2, etc. Each time a user-defined macro is called, a number (defined by the user) of temporary variables is allocated. These are local to the current call. The first three temporary variables are initialized by ML/I in the following way:

T1 contains the number of arguments.
T2 contains the number of macro calls performed so far.
T3 contains the current depth of nesting of macro calls.

The initial value of T2 is a number unique to the current call and is useful for generating unique labels. If the replacement text of a macro contains a label it is imperative that a different name be generated for the label each time the macro is called. This can be achieved by writing the label as:

```
LAB:T2.
```

(Remember that the colon is assumed to be an insert name in this and all subsequent examples.) A later example illustrates the use of this feature.

Further Facilities for Inserts

It has been seen that arguments and macro variables may be inserted into text. This section describes the complete facilities offered by inserts.

The general form of the designation of an insert consists of a flag followed by a subscript expression. The subscript expression may consist simply of a positive integer, as in the previous examples, or it may be the name of a macro variable or an arithmetic expression involving macro variables and/or constants. Thus if T1 had its initial value then AT1 would reference the last argument and AT1–1 would reference the next to last argument. The following are the possible flags that can be used for inserts, together with a description of each:

A. —argument.
D. —delimiter.
WA, WD. —argument or delimiter in its written form. The argument or delimiter is not evaluated but is inserted literally.
null. —the numerical value of the subscript expression is inserted.
L. —macro label. This type of insert "places" a macro-time label which may be the subject of a macro-time GO TO statement; it is a special type of insert in that it does not cause any output text to be generated, i.e., its value is null.

Operation Macros

ML/I contains a number of built-in macros called *operation macros.* Operation macros are used for such purposes as defining new constructions, performing macro-time arithmetic and changing the position of scan (using the macro-time GO TO statement). The names of operation macros all begin with the letters "MC" so that they may readily be differentiated from user-defined macros. An example of an operation macro is MCDEF, which is used for defining macros. This has form:

```
MCDEF argument 1 AS argument 2 ;
```

The first argument must be in the form of a *structure representation,* which specifies the delimiter structure of the macro being defined. In the simplest case, where all the delimiters are fixed, a structure representation is specified by writing the delimiters in the order in which they are to occur. The macro name, which is regarded as delimiter number zero, comes first. The second argument of MCDEF specifies the replacement text of the macro being defined. Thus if "TO" and semicolon were chosen to delimit the ends of the two arguments of MOVE, its definition would be written:

```
MCDEF MOVE TO ; AS
⟨ LAC :A1.
  DAC :A2.
⟩;
```

and a call of form:

```
MOVE X TO TABLE + 6;
```

would expand into:

```
LAC X
DAC TABLE + 6
```

Note that the replacement text of MOVE has been enclosed in literal brackets in the above definition. This is because all arguments to operation macros are evaluated before being processed. Assume that the literal brackets had been omitted. Then in evaluating the second argument of MCDEF, ML/I would have tried to perform the inserts. This would have unfortunate results since the inserts should be performed when MOVE is called, not when it is defined. In its correct form, with the literal brackets present, the evaluation of the argument leads to the literal brackets being deleted and the enclosed text being copied literally. This text then becomes the replacement text of the MOVE macro. The reader is probably tempted to ask why operation macro arguments are not treated literally by the system in order to avoid the necessity of using literal brackets, but there are many examples where the dynamic element is vital.

Structure Representations for More Complicated Cases

The exact details of how to write structure representations in the general case are to be found in the User's Manual and this paper will confine itself to a basic outline. There are certain reserved words in structure representations. Among these are: WITH, OPT, OR, ALL and any atom consisting of the letter "N" followed by an integer. (The names of reserved words can be changed dynamically by the user, if he desires.) WITH is used to specify delimiters that consist of more than one atom. For example, if a delimiter consists of the atoms A1, A2, . . ., Ak, then this is specified by:

```
A1 WITH A2 WITH ... WITH Ak
```

The remaining reserved words are used to specify branches and nodes of the delimiter structure.

As has been seen, the purpose of a delimiter structure is to specify a successor or set of alternative successors for each delimiter. In a structure representation each specification of a delimiter is immediately followed by a specification of its successor. This successor may be defined in any of the following ways:

(a) If there is a single possible successor, as in the case of the delimiters of the MOVE example, this is specified simply by writing its name. (It is convenient to imagine a special symbol "null" occurs at the end of each structure representation. Any delimiter with "null" as its successor is a closing delimiter.)

(b) If there are alternative successors, this is specified by writing:

OPT *branch* 1 OR *branch* 2 OR *branch n* ALL

A branch may be a single delimiter specification or a structure in itself. An example of the use of this facility is the following macro to convert fully parenthesized algebraic notation to Polish Prefix notation.

```
MCDEF (OPT + OR - OR * OR / ALL) AS ( :D1.:A1.:A2.);
```

Here the left parenthesis is the macro name. The replacement text consists of the first delimiter followed by the first argument followed by the second argument.

(c) If a delimiter is at the end of a branch, its successor is normally taken as the successor of the occurrence of ALL corresponding to the branch. Thus in the above Polish Prefix example each of the alternative arithmetic operators has a right parenthesis as its successor. However, a delimiter at the end of a branch can be given a different successor by writing the name of a *node* at the end of a branch. A node is designated by the letter "N" followed by a positive integer. The successor of the delimiter is then taken as the successor of the node, which is specified by writing the name of the node in front of some delimiter specification in the structure representation. The following example illustrates the use of nodes. Assume a macro called MIN allows any number of arguments separated by commas and terminated by a semicolon. Its structure representation would be written: MIN N1 OPT, N1 OR; ALL

This is interpreted thus. The successor of MIN is either a comma or a semicolon. Node N1 is to be associated with these alternatives since N1 has been written before the specification of the alternatives. If a comma is found as a delimiter, its successors are the successors of N1, namely either a comma or a semicolon. If a semicolon is found, its successor is the successor of ALL, namely "null."

Use of Macro-Time Statements

In simple macros such as MOVE, the replacement text is a fixed skeleton of code which is to replace each call. This is obviously not adequate for more sophisticated cases. Consider the IF macro used earlier as an example, which had alternative forms:

(a) IF *argument* 1 = *argument* 2 THEN *argument* 3 END
or (b) IF *argument* 1 = *argument* 2 THEN *argument* 3 ELSE *argument* 4 END

Clearly the replacement text of IF must be made to generate different code in the two cases. This can be done by using the macro-time GO TO statement, MCGO, in its form: MCGO *argument* 1 UNLESS *argument* 2 = *argument* 3;

The definition of IF is written as follows:

```
MCDEF IF = THEN OPT ELSE END OR END ALL AS
(       LAC :A1         /Load first argument.
        SAD :A2.        /Skip if it differs from the second
                         argument.
        SKP             /Equal case: skip one instruction.
        JMP XX:T2.      /Unequal case: jump over inserted
                         code.
        :A3.            /Inserted code.
MCGO L1 UNLESS :D3. = ELSE; /Macro-time jump if
                             else clause is absent.
        JMP YY:T2.      /Jump over else clause.
```

```
XX:T2.,     :A4.          /Insert else clause.
YY:T2., MCGO L2;          /Macro-time jump to end of replace-
                           ment text.
:L1.XX:T2., :L2. );       /Placing of macro-time labels.
```

Note that the second argument of MCGO, which is ":D3.", is evaluated before being compared with the character string "ELSE". This evaluation causes the text of the third delimiter to be inserted and to take part in the comparison. Note also the use of the initial value of T2 to generate unique labels. The following two successive calls of the IF macro:

```
IF A = B THEN JMS SUB
END
IF PIG = DOG THEN LAC C
                  DAC D
ELSE LAC Y
     DAC Z
END
```

would generate the code:

```
      LAC A       /First call.
      SAD B
      SKP
      JMP XX1
      JMS SUB
XX1,  LAC PIG     /Second call.
      SAD DOG
      SKP
      JMP XX2
      LAC C
      DAC D
      JMP YY2
XX2,  LAC Y
      DAC Z
YY2,
```

The two macro-time statements MCSET (the macro-time assignment statement) and MCGO can be used to form macro-time loops, which are useful in processing macros with a variable number of arguments. The following macro, which is useful in a number of applications, illustrates the macro-time loop. The macro, which has the composite name "INDEX (", successively compares its first argument with each succeeding argument until a match is found, and returns as its value the number of the matching argument.

```
MCDEF INDEX WITH (N1 OPT, N1 OR) ALL AS
(     MCSET T2 = 2;
:L2.  MCGO  L1 IF :AT2. = :A1.;
      MCSET T2 = T2 + 1;
      MCGO  L2;
:L1.  :T2. );    /Insert value of T2.
```

This macro can be used for switching. Assume it is desired to test the second delimiter of a macro and branch to macro-time label L2 if the delimiter is a plus sign, to L3 if it is a minus sign, and so on. Then the following statement achieves this:

```
MCGO L INDEX (:D2., +, -, *, /);
```

Scope of Definitions

New constructions may be defined at any time during text evaluation but, ML/I being a one-pass system, definitions only apply to text that comes after them. It is possible to redefine a construction and individual constructions may be deleted by redefining them to have a null effect. Constructions may be defined as local to a piece of replacement text or even local to the evaluation of an argument. It is possible to use a macro to set up the definition of another macro, and this is useful in dealing with declarative statements. Thus DECLARE could be a macro such that if, for example, the statement:

```
DECLARE X REAL
```

were encountered, then the DECLARE macro would set up a definition of X as a macro, which might, for instance, take the form:

```
MCDEFG X AS ( 100 MCSET P1 = 6 ; );
```

(MCDEFG is the same as MCDEF except that the definition is global rather than local to any containing macro.) This definition would cause each subsequent occurrence of X to be replaced by 100 and each call of X would, as a side-effect, set the permanent variable P1 to value six. The side-effect could be used to convey the information that X was of type "real".

Implementation

ML/I has been implemented on the PDP-7 and I.C.T. Atlas 2 computers. It occupies about three thousand words of storage. It is planned to write the logic of ML/I in a macro language so that, by suitably coding the basic macros and running the logical description through ML/I itself, it is possible to generate code for any machine. It is hoped that this technique will enable ML/I to be transferred to a new machine in about one man-month, even allowing for the multitude of unforeseen difficulties this kind of operation always involves.

RECEIVED JANUARY, 1967; REVISED MAY, 1967

REFERENCES

1. FAP: reference manual. Form C28-6235, IBM Programming Sys. Publ., Poughkeepsie, N.Y., Sept. 1962.
2. HALPERN, M. I. XPOP: a meta-language without metaphysics. Proc. AFIPS 1964 Fall Joint Comput. Conf., Vol. 26, pp. 57–68.
3. WILKES, M. V. An experiment with a self-compiling compiler for a simple list-processing language. *Annual Revue in Automatic Programming, Vol. 4*. Pergamon Press, Oxford, England, 1964.
4. WAITE, W. M. A language-independent macro processor. *Comm. ACM 10*, 7 (July, 1967), 433–440.
5. STRACHEY, C. A general purpose macrogenerator. *Comput. J. 8*, 3 (Oct. 1965), 225–241.
6. MOOERS, C. N. TRAC, a procedure describing language for the reactive typewriter. *Comm. ACM 9*, 3 (Mar. 1966), 215–219.
7. BROWN, P. J. ML/I user's manual. University Math. Lab., Cambridge, England, July 1966.