EPS USER'S GUIDE

by

E. Kantorowitz, A. Gonen, I. Sagie

Technical Report No. 33

February, 1974
Copyright © E. Kantorowitz, A. Gonen and I. Sagie, 1974.

Chapters 2-5 as well as sections 9.1-9.4 and 8.1-8.3 of this manual are reproduced from

August 1970.

with the kind permission of the author.
Preface

The two principal goals of the EPS (Extensible Programming System) system are to facilitate assembly language programming and implementation of special purpose programming languages, as described in detail in Chapter 1. The EPS system consists of an extended version of P.J. Brown's macro processor ML/I and of a macro library. The macro processor is used as a compiler for the programming languages employed in the system, and the macro library constitutes the implementation of the EPS common core language. This core language has facilities that are useful in most programming languages, such as constructs for coding blocks, procedures, data declarations, control of program flow and debugging aids. A special purpose programming language may relatively easily be implemented by extending the common core language of EPS with the data types and operators that are especially useful for the applications of the new language. These extensions are implemented by defining appropriate macros as explained in this manual.

A discussion of basic ideas on which EPS is based is found in:

The authors wish to acknowledge the originator of ML/I, Dr. P.J. Brown, for his great help and for his kind permission to reproduce parts of his "ML/I users Guide".

The development of the EPS system was sponsored in part by the U.S. Government through the European Research Office of the U.S. Army, and in part by a grant from I.B.M.

Haifa
February 1974
Chapter 1 Introduction

1.1 General description
1.2 Organisation of the manual
1.3 Notation for describing syntax
1.4 Further points of notation

Chapter 2 The Environment and its constituents

2.1 Basic action of ML/I
2.2 Character set
2.3 Text
2.4 Macros and delimiter structures
   2.4.1 Examples of macros
   2.4.2 Delimiter structures
   2.4.3 Optional and repeated delimiters
   2.4.4 Macro definitions
   2.4.5 The difference between macros and subroutines
   2.4.6 Impossible replacements
2.5 Introduction to macro-time variables and statements
2.6 Inserts
   2.6.1 Macro variables
   2.6.2 Initialization of macro variables
   2.6.3 Subscripts and macro expressions
   2.6.4 Integer overflow
   2.6.5 Macro labels
   2.6.6 Macro elements
   2.6.7 Insert definitions
   2.6.8 Examples of inserts
2.7 Skips
   2.7.1 Matched skips and straight skips
   2.7.2 Literal brackets
   2.7.3 Example of a matched skip
2.8 Warning markers
2.9 Summary of the environment
2.10* Normal-scan macros and straight-scan macros
2.11 Name environment used for examples

Continued:
Chapter 3 Text Scanning and evaluation

3.1 Nesting and recursion
3.2 Call by name
3.3 Details of the scanning process
3.4 The method of searching for delimiters
3.5* Exclusive delimiters
3.6* Dynamically generated constructions

Chapter 4 Operation macros and their use in setting up the environment

4.1 Operation macros
4.2 Use of literal brackets for surrounding operation macro arguments
4.3 NEC macros
4.4* Dynamic aspects of the environment
4.5* Protected and unprotected inserts
4.6* Ambiguous use of names
4.7* Implications of rules for name clashes

Chapter 5 Specifications of individual operation macros

5.1 Specifications of delimiter structures
5.1.1 Keywords
5.1.2 The consequences of evaluation
5.1.3* Introduction to more complicated cases
5.1.4* Full syntax of structure representations
5.1.5* Examples of complex structure representations
5.1.6 Possible errors in structure representations

5.2 The NEC macros
5.2.1 MCWARN
5.2.2 MCINS
5.2.3 MCSKIP
5.2.4 MCDEF
5.2.5 MCNOWARN, MCNOINS, MCNOSKIP and MCNODEF
5.2.6 MCWARNG, MCINSG, MCSKIPG and MCDEFG
5.2.7 MCALTER

5.3 System functions
5.3.1 MCLENG
5.3.2 MCSUB

5.4 Further operation macros
5.4.1 MCSET
5.4.2 MCNOTE
5.4.3 MCGO
5.4.3.1* Exact description of a GO TO
5.4.4 MCPVAR

Continued:
### Chapter 6 Data specification macros (declarations)

- **6.1** Introduction
- **6.2** Identifiers
- **6.3** The INT macro
- **6.4** The SHORT INT macro
- **6.5** The REAL macro
- **6.6** The LONG REAL macro
- **6.7** The INT ARRAY macro
- **6.8** The SHORT INT ARRAY macro
- **6.9** The REAL ARRAY macro
- **6.10** The LONG REAL ARRAY macro
- **6.11** The CHAR macro

### Chapter 7 Program structuring macros and activation of the operating system

- **7.1** Introduction
- **7.2** Program organization and scope of names
- **7.3** The PROCER macro
- **7.4** The MAIN PROCER macro
- **7.5** The PROC macro
- **7.6** The BEGIN macro
- **7.7** The CALL macro
- **7.8** The REPEAT MACRO and the FOR WHILE and EXIT constructs
- **7.9** The CASE macro
- **7.10** The EXIT macro
- **7.11** The IF-THEN-ELSE macro
- **7.12** The NOP macro
- **7.13** The COMMENT macro
- **7.14** Activation of the operating system

### Chapter 8 Error messages and macros for program verification and maintenance

- **8.1** Examples of an error message
- **8.2** Notes on context print-outs
- **8.3** Complete list of messages
  - 6.3.1 - 6.3.1 Description of individual messages
- **8.4** Implementation defined messages
  - 8.4.1 - 8.4.4 list of messages
  - 8.4.5 Special syntax errors detection and the TEXT macro
  - 8.4.6 The TEST macro
- **8.5** Error messages which are detected by EPS
  - 8.5.1 - 8.5.9 complete list of errors detected by EPS
- **8.6** Precompilation pass for errors detection
- **8.7** Macros for program debugging and maintenance

Continued:
Chapter 9 How to use EPS

9.1 Introduction
9.2 Possible sources of error
  9.2.1 Jumping over expanded code
  9.2.2 Generation of unique labels
  9.2.3 Lower case letters
  9.2.4 Use of newlines in definitions
  9.2.5 Use of redundant spaces
9.3 Simple techniques
  9.3.1 Interchanging two names
  9.3.2 Removing optional debugging statements
  9.3.3 Inserting extra debugging statements
  9.3.4 Deleting a macro
  9.3.5 Differentiation between special-purpose registers and storage locations
  9.3.6 Testing for macro calls
  9.3.7 Searching
  9.3.8 Bracketing within macro expressions
  9.3.9 Deletion from source text only
9.4 Sophisticated techniques
  9.4.1 Macro-time loop
  9.4.2 Examining optional delimiters
  9.4.3 Dynamically constructed calls
  9.4.4 Arithmetic expression macro
  9.4.5 Formal parameter names
  9.4.6 Intercepting changes of state
  9.4.7 Remembering code for subsequent insertion
  9.4.8 Constructions with restricted scopes
  9.4.9 Optimizing macro-generated code
  9.4.10 Macro to create a macro
9.5 Reserved entities
  9.5.1 Reserved characters
  9.5.2 Reserved identifiers
  9.5.3 Reserved macro variables
  9.5.4 Reserved registers

Continued:
9.6 Practical techniques
9.6.1 How to construct an arithmetic expression
9.6.2 How to construct a logical expression
9.6.3 How to define new data types
9.6.4 Problems due to the layout of Assembly 360 code
9.6.5 Some timing consideration

Chapter 10 The macro library

10.1 Introduction
10.2 The ARITEXP macro
10.3 The LOGEXP macro
10.4 The SET macro
10.5 The SETC macro
10.6 The WRITE macro
10.7 The READ macro

APPENDIX

1. Additional operation macros
   1.1 The macro-time symbol table
   1.2 The macro-time parameter table
   1.3 The MENTRY macro
   1.4 The MTRACE macro
   1.5 The MCTYPE, MCDIME and MCTRACETAG macros
   1.6 The MCCLSMB macro
   1.7 The MCPREENTRY macro
   1.8 The MCPRTYPE macro
   1.9 The MCPRDIME macro
   1.10 The MCPRM and the MCCLPRM macros
   1.11 The MCNOG and MCG macros
   1.12 The MCDEBUG and MCNODEBUG macros
   1.13 The MCMESG macro
   1.14 The MCLINE macro
   1.15 The MCLIST and MCNOLIST macros
   1.16 The MCTEX and MCNOTEXT macros
   1.17 The MCERFLAG macro
   1.18 The MCSTOP macro

Continued:
2. Physical forms of source and output text and initial state of name environment

3. Set of characters

4. The debugging file

5. System variables

6. Extra keywords

7. Permanent variables

ERROR MESSAGES INDEX

INDEX
Chapter 1  Introduction

1.1 General Description

EPS (an Extensible Programming System) is a basic programming system which may be extended into specialized programming systems. These systems may be tailored to fit any application areas, and they may be designed for the use of either computer professionals or for users who know only a little about computing.

The main features of EPS are:

(1) The programming languages of the system have facilities for "structured programming".

(2) The programming language is extensible, such that the problem solver may introduce into the language the concepts that are useful in the problem area encountered and thereafter use these concepts in the formulation of his solution.

(3) The programmer is able to specify the machine instructions to be generated in the target code such that highly efficient code may be produced when needed.

(4) The programmer has the ability to activate any of the functions of the operating system. This ability may be needed in sophisticated applications, and when it is required to utilize the resources of the computing system efficiently.

(5) The programmer has powerful aids for program verification.

EPS has been implemented utilizing an extention of Brown's macro processor ML/l. EPS may be used as the base of a programming system having a number of programming languages. The ML/l processor is used as a common compiler for all the languages of the system. A source program in any of the languages of the system is mostly composed of ML/l macro calls which are translated by the macro processor into an Assembly 360 program.

Any language of the system is implemented by defining a set of ML/l macros. The programmer may extend the language by defining new macros in accordance with this needs.

The EPS is thus designated to function as the common base for all languages in a programming system and is composed of:
1) The ML/l macro processor which serves as a common compiler and as a tool for implementing new languages or language extensions (by defining new macros).

2) A set of common macro definitions which constitute the facilities common for most of the languages.

There are four classes of such common macros, namely:
- basic data specification macros ("declarations");
- macros for program structuring;
- macros for activation of the operating system;
- macros for program verification and maintenance.

1.2 Organization of the manual

Chapters 2, 3, 4 and 5 of this manual describe the ML/l macro processor in full detail. These chapters are reproduced from P.J. Brown, ML/l User's Manual, University of Kent at Canterbury, August 1970. Chapter 6 describes the data specification macros. Chapter 7 describes the macros for program structuring and the techniques for activation of the operating system. Chapter 8 describes error messages, and, macros for program verification and maintenance. Chapter 9 describes in details how to utilize efficiently the EPS. Parts of chapters 8 and 9 are reproductions of chapters 6 and 7 of the ML/l User's Manual. Chapter 10 includes library macros.

1.3 Notation for describing syntax

The notation used to describe syntax should be self-explanatory. An example of its use is the following description of a hypothetical IF statement:

```plaintext
IF condition THEN statement;
```

As can be seen, a syntactic form is defined by concatenating its constituents. A constituent that is itself the name of a syntactic form is underlined. The remaining constituents are literals.

A notation borrowed from Brooker and Morris is used to indicate parts of syntactic forms that may optionally be repeated and/or omitted. In this notation a constituent of series of constituents that may optionally be omitted is written:
[ constituents ? ]

Constituents that may be repeated any desired number of times are written:

[ constituents * ]

and constituents that may be omitted or repeated are written:

[ constituents *? ]

Thus, if the above IF statement had an optional ELSE clause, it would be written:

IF condition THEN statement [ELSE statement ?];

and a hypothetical SUM statement which permitted any number of arguments, provided there were at least two, might be defined:

SUM argument [, argument*];

Lastly, when there are several alternative forms for a constituent, these are written:

(form 1)
(form 2)
(form N)

Thus an expression might be defined as:

(+)
variable [(-) variable*?]
(x)
(/)

Note that the asterisk means that the syntactic forms enclosed within the brackets may be repeated; it is not required that identical text be written at each repetition.

1.4 Further points of notation

(a) When it is desired to emphasize the presence of a space, tab or newline in a piece of text, this is done by writing SPACE, TAB, or NL, respectively. Note that this is simply a point of notation and the reader should be careful not to interpret an occurrence of, say, NL in a specification as requiring that he write 'N' and 'L' and underline them.

(b) An integer is said to be positive only if it is greater than zero, and negative only if it is less than zero. Integers in EPS are represented to a decimal base.
Chapter 2 The environment and its constituents

2.1 Basic action of ML/I

The basic action of ML/I is as follows. The user feeds to ML/I some text and an environment. The purpose of the environment is to specify that certain insertions, deletions, expansions, translations or other modifications are to be made in the text. ML/I performs the textual changes specified by the user. This process is called evaluation of text, and the text generated as a result of the changes is called the value text. The text being evaluated is called the scanned text. In many simple applications of ML/I, the process of evaluation consists of a good deal of straight copying, the value being the same as the original, but periodically a change is made and the generated value text is different from the original scanned text.

The purpose of this Chapter is to explain the mechanisms at the disposal of the user and to give examples of their use. All the possible constituents of the environment will be described and the resultant textual changes will be explained by describing the form of the scanned text and the form of the corresponding value in each case. The mechanisms for setting up the environment will be explained in subsequent Chapters.

2.2 Character set

The character set of ML/I, i.e. the set of allowable characters in the text it processes, is implementation-defined (see Section 3 of the Appendix). However, the character set will normally contain a primary (upper case) set of letters A - Z, the numbers 0 - 9, and a number of characters that are not letters or numbers. Characters that are not letters or numbers are called punctuation characters. (If input is from cards the characters tab and newline will not normally physically exist as characters on the cards. However, the input routine may artificially add the character "newline" at the end of each line and might possibly have some arrangement for inserting tabs as well. See Section 3 of the Appendix for details). If an implementation contains both upper and lower case letters in its character set, then these are treated as entirely different sets of characters and it is not possible to use a lower case letter interchangeably with its upper case equivalent.
2.3 Text

A feature of ML/I is that it does not consider text character by character but in units of atoms. An atom is a single punctuation character or a sequence of letters and digits that is surrounded by punctuation characters (assuming an imaginary punctuation character at the beginning and end of the text). There is no restriction on the length of an atom. To take an example, the text:

\[
\text{PIG, TAB LAC SPACE 4057}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

would be regarded as six atoms as shown.

The following definitions will be used in the rest of this manual. Text is a (possibly null) sequence of atoms. The source text is the text supplied as input to ML/I, and the output text is the text derived form evaluating the source text. The physical form of the source text and output text is implementation-defined (see Section 2 of the Appendix). The action of evaluating a particular piece of source text is called a process.

2.4 Macros and delimiter structures

Before defining a macro it may be useful to consider the sort of text replacement that macros are designed to achieve. PDP-7 Assembly Language will be taken as an example (but it will not be assumed the reader is necessarily familiar with this language). Assume the user wishes to introduce a new instruction of form:

\[
\text{ESUB X meaning "subtract X from the accumulator".}
\]

Now this instruction is not in the PDP-7 instruction set but its effect can be achieved by the three instructions:

- CMA complement accumulator.
- ADD X add X to accumulator.
- CMA complement accumulator.
The introduction of ESUB would be achieved as follows. The user would write his program as if ESUB were an extra machine instruction. Before the program was assembled it would be passed through ML/I with ESUB defined as a macro name with the above three instructions as its replacement text. ML/I would replace each occurrence of ESUB by its expanded form and the resultant output text could then be assembled normally. Each piece of text to be replaced is called a macro call and the text corresponding to X above is called the argument of the call. (Within the replacement text of ESUB it is necessary to specify that the argument of the call should be inserted immediately after ADD. This is done by a constituent of the environment called an "insert", which will be described later.)

This example serves as a simple illustration of the primary use of ML/I, namely to serve as a preprocessor to an existing piece of software to allow the user to introduce new statements of his own design into the existing language. Each new statement must be expansible in terms of the existing language.

Macros may have any number of arguments. Arguments are separated by predefined atoms or sequences of atoms called delimiters. When defining a macro, the user specifies what the delimiters are. The macro name is regarded as a delimiter and is called the name delimiter to distinguish it from the remaining delimiters, which are called secondary delimiters. The delimiter following the last argument of the call is called the closing delimiter. The general form of a macro call can, therefore, be represented as:

name delimiter [argument secondary delimiter * ?]

Arguments may be null but delimiters must consist of at least one atom.

Every time ML/I encounters in the scanned text an atom or series of atoms that has been defined as a macro name, it searches for the secondary delimiters (if any) and then replaces the entire macro call by the value of the replacement text for the macro. More details of the way macro calls are scanned are given in Sections 3.3 and 3.4.

2.4.1 Examples of macros

It may be instructive at this stage to consider a few more examples of macros.
These examples, which are listed below, are all of simple macros with fixed delimiters. Macros with more elaborate patterns of delimiters will be considered later. Note that ML/I could be used to add these macros to any desired programming language, whether high or low level.

Example 1 A macro to generate a loop, which has form

```
DO arg A TIMES arg B REPEAT
```

Here the delimiters are DO, TIMES and REPEAT. DO is the name delimiter, and TIMES and REPEAT are secondary delimiters. REPEAT is the closing delimiter. ML/I does not require that macro calls be written on a single line, and calls of this macro would tend, in practice, to span several lines of text.

Example 2 A macro of form

```
MOVE FROM arg A TO arg B;
```

The name of this macro consists of the two atoms "MOVE FROM".

Example 3 A macro to interchange two variables, which has form

```
INTERCHANGE ( arg A, arg B ) NL
```

In this example both the name and the closing delimiter consist of more than one atom. Note that ML/I does not, like some software, truncate long names such as "INTERCHANGE".

Example 4 Assume that within a program two different names, COUNT and CONT, have inadvertently been used for the same variable. Then this error could be corrected using ML/I with CONT defined as a macro with COUNT as its replacement text. Here the name delimiter, CONT, is also the closing delimiter.
The reader should, at this stage, appreciate why ML/I considers text as a sequence of atoms rather than a sequence of individual characters. If the latter were the case, ML/I would be liable to take names such as DOG and RANDOM as calls of the above macro DO since each name contains the letters "DO". As the situation stands, however, the letters "DO" would only be taken as a macro call if they were surrounded by punctuation characters.

2.4.2 Delimiter structures

The macros considered so far have had fixed delimiters. However, it is possible to have macros with any number of alternative patterns of delimiters. As a very simple example of this consider the ESUB macro. In PDP-7 Assembly Language statements are terminated with either a tab or a newline, and so it would be desirable to allow both of these as alternatives for the closing delimiter of ESUB.

In order to specify the patterns of possible delimiters of a macro the user specifies a delimiter structure. Each macro has its own delimiter structure and other constituents of the environment also have delimiter structures. A delimiter structure is a set of delimiter specifications, each of which is a sequence of one or more atoms. These sequences of atoms need not be distinct. One or more of these delimiter specifications are designated as names of the structure. The remainder are secondary delimiters. With each delimiter specification is associated a specification of its successor(s). This may be

(a) null

or (b) another delimiter specification within the structure,

or (c) a set of alternative delimiter specifications within the structure.

Successors specify what to search for next when scanning. A delimiter with a null successor is a closing delimiter. As an illustration of the use of a delimiter structure consider the scanning of a macro call. During this scanning, each time a delimiter is found, the delimiter structure of the macro being called is referenced to find the successor(s) of the current delimiter and subsequent text is then scanned to try to find this successor. This process continues until a closing delimiter is found.
As an example of a delimiter structure, the delimiter structure of the ESUB macro would contain three delimiter specifications with the following information about them:

(a) ESUB name with (b) or (c) as its successor.
(b) TAB secondary delimiter with no successor.
(c) NL secondary delimiter with no successor.

The rules for setting up delimiter structures (see Section 5.1) ensure that they have certain properties. Among these properties are the following:

(a) If there is more than one name each name is represented by a different sequence of atoms.
(b) If a delimiter specification has alternative successors each is represented by a different sequence of atoms.
(c) The structure is connected. This means that it must be possible to reach each secondary delimiter by a sequence of successors from some name.

2.4.3 Optional and repeated delimiters

It is possible, by designing a suitable delimiter structure, to have a macro with a variable number of arguments, in particular a macro with optional arguments and/or with an indefinitely long list of arguments. For instance, suppose it is desired to implement a macro with alternative forms:

```
IF argument THEN argument
END

and IF argument THEN argument
ELSE argument
END
```

This is done by specifying that either ELSE or END is the successor of THEN. END is a closing delimiter and ELSE has successor END.
As a second example consider a macro of form:

```
SUM argument [+(+) argument*?];
```

This macro has an indefinite number of arguments, separated by plus or minus signs. Its delimiter structure has four members as follows:

(a) SUM name with (b), (c) or (d) as successor.

(b) + secondary delimiter with (b), (c) or (d) as successor.

(c) - secondary delimiter with (b), (c) or (d) as successor.

(d) ; secondary delimiter with no successor.

### 2.4.4 Macro definitions

Now that the basic concepts behind macros have been introduced, it is possible to explain more exactly what makes up a macro definition. Macro definitions are the most important constituents of the environment. A macro definition consists of:

(a) A delimiter structure. The name delimiter(s) of this structure are the macro names.

(b) A piece of replacement text.

(c) An integer exceeding two called the capacity, the purpose of which is explained in Section 2.6.1.

(d) An on/off option. If this option is on, a macro is called a normal-scan macro. Otherwise it is called a straight-scan macro. The effect of this option is explained in Section 2.10.

The reader need not for the moment concern himself with (c) and (d) since nearly all macros will be normal-scan and will have capacity three.

### 2.4.5 The difference between macros and subroutines

There is often confusion between the purpose of macros and the purpose of
subroutines. Macros, however, always generate in-line code and so this code is inserted as many times as the macro is called. Subroutines use out-of-line code and there is only one copy of this code for a particular program. Thus macros are used only when the code to be inserted is short or highly parameterized. It would not be convenient, for instance, to use subroutines to perform the functions of any of the macros used as examples in previous Sections.

2.4.6 Impossible replacements

It is worth noting some of the types of replacement that it is not possible to perform by means of macros. Below are two examples of illegal syntax of macro calls, together with possible correct forms.

(a) Wrong \[ \text{arg A} = \text{arg B}; \] since each macro call must start with a macro name.

Right \set \text{arg A} = \text{arg B};

(b) Wrong \$ \text{character} \] It is not possible to define an argument as the character (or atom) immediately following a given name. Every argument must be followed by some pre-defined delimiter.

Right \$ \text{argument} ;

2.5 Introduction to macro-time variables and statements

The form of the value of a call of such macros as the IF and SUM macros used earlier as examples would have to depend on the particular patterns of delimiters that were used in the call. For instance:

\[ \text{SUM} \quad \text{ALPHA} + \text{BETA} ; \]

must generate an entirely different set of instructions from:

\[ \text{SUM} \quad \text{ALPHA} - \text{BETA} - \text{GAMMA} + \text{X} + \text{Y} - \text{Z} ; \]

and, in the case of IF, the form of the value text must depend upon whether ELSE
was present. Macros such as these, therefore, are more complicated than the ESUB case, where a fixed skeleton of code consisting of three machine instructions is substituted for each call. The only variable element in the ESUB case is the form of its argument. In the more complicated cases, where the delimiters provide a second variable element, the user has to write a little program which is executed by ML/I and tests the form of the delimiters used and generates code accordingly. In the case of SUM, which has an indefinitely long list of arguments and delimiters, this program would involve a simple repetitive loop to iterate through the list. Hence ML/I contains an elementary programming language of its own. This language contains an assignment statement, a conditional GO TO statement, labels and integer variables. All these are called macro-time entities to distinguish them from the corresponding execution-time entities, and the reader must be careful not to confuse the two. The difference is illustrated thus: the DO macro of Section 2.4.1 would generate a loop which was performed at execution-time and controlled by an execution-time variable; on the other hand the value text for the SUM macro would be generated by a macro-time loop controlled by a macro-time variable.

Macro variables and macro labels are considered in the next Section. Macro-time statements are considered in detail in Chapter 4.

2.6 Inserts

This section describes how quantities can be inserted into text. In particular it describes how arguments of macro calls are inserted into replacement text. However, first it is necessary to consider some of the quantities, in addition to arguments, that may be inserted into text.

2.6.1 Macro variables

Macro variables are integer variables available to the user at macro-time, ML/I contains facilities for performing arithmetic on these variables, testing their values, and inserting their values into the text. They are useful as switches and for counting (e.g. in processing macros with a variable number of arguments).
There are three kinds of macro variables, namely:

(a) **permanent variables**, referred to as P1, P2, ...
(b) **system variables**, referred to as S1, S2, ...
(c) **temporary variables**, referred to as T1, T2, ...

Permanent and system variables have global scope; this means they can be referred to anywhere. An implementation-defined number of each is allocated at the start of each process and these remain in existence throughout. The user may allocate extra permanent variables (but not system variables) if he likes (see Section 5.4.4). The difference between permanent and system variables is that the former have no fixed meanings and are free implementation-defined meanings and are free for the user to use as he wishes, but the latter have fixed implementation-defined meanings associated with controlling the operation of ML/I. For example, in a given implementation S20 might control the listing of the source text; if it was zero no listing would be produced and if it was one there would be a listing. Sections 5 and 7 of the Appendix describe the meanings of system variables (if any) and state the number of permanent and system variables that are initially allocated.

Temporary variables, on the other hand, have a more local scope. During the evaluation of the source text there are no temporary variables in existence. However, each time a macro call is made a number of temporary variables is allocated and these remain in existence while the replacement text of the macro is being evaluated. The number of temporary variables allocated at the call of a macro is given by the capacity of the macro (see Section 2.4.4). The capacity is usually three. If temporary variable N is referenced during the evaluation of the replacement text of a macro call, this is taken to mean the Nth temporary variable associated with the call. Since, as will be seen later, it is possible to have macro calls within macro calls, it is possible to have several allocations of temporary variables in existence at the same time.

### 2.6.2 Initialization of macro variables

The initial values of all macro variables are undefined except for the values of the first three temporary variables of each allocation, which are initialized as
follows:

T1  the number of arguments of the current macro call;

T2  the number of macro calls so far performed by ML/I during the current process. The importance of this number is that it is unique to the current call;

T3  the current depth of nesting of macro calls (i.e. the number of calls, including the present one, currently being processed; calls of operation macros (see Section 4.1) are not counted here, though they do count toward the setting of T2).

It is to be emphasized that these are initial values and the user is free to change them if he wishes. (In this way temporary variables are unlike system variables. If the values of system variables, even those without assigned meanings, were changed arbitrarily it might have a tragic effect.)

2.6.3 Subscripts and macro expressions

In the previous Sections macro variables were specified by a letter followed by a number (e.g. P2), but there are other possibilities. The general form of a macro variables is:

(P)
(S) subscript
(T)

where a subscript is an unsigned positive integer or a macro variable. The value of the subscript specifies the macro variable to be referenced. Thus if T3 has value 4, then PT3 would specify P4. As a more complicated example, if T1 had value 2 and P2 had value 6, then TPT1 would specify the sixth temporary variable.

Macro variables can be combined into macro expressions, which are used when it is desired to perform arithmetic calculations during macro generation. Examples of macro expressions are:

1, -6, 3-S1, -TT1-145/P2+P3+6

Multiplication is represented by an asterisk. The general form of a macro expression is:
The result of a macro expression is the integer derived from calculating the expression by the ordinary rules of arithmetic. Unary operators are performed first, followed by the binary operators from left to right with the proviso that multiplication and division take precedence over addition and subtraction. Division is truncated to the greatest integer that does not exceed the exact result. Division by zero is detected as an error. Examples are:

(a) $1 + 2 \times 3$ has result 7.
(b) $3 \times 7/8$ has result 2.
(c) $7/8 \times 3$ has result 0.
(d) $-5/4$ and $5/-4$ both have result -2.
(c) $-4/-3 \times -6$ has result -6.

2.6.4 Integer overflow

Each implementation has a maximum absolute value which must not be exceeded by any integer derived during the calculation of a macro expression or subscript. The effect of exceeding this value is implementation-defined. See Section 5 of the Appendix for details.

2.6.5 Macro labels

Since there is a facility for a macro-time GO TO, there is also a facility for placing macro-time labels. These are called macro labels. Each macro label is designated by a unique positive integer.
2.6.6 Macro elements

Macro variables, macro labels, arguments and delimiters are collectively called macro elements. It is convenient to regard macro elements as part of the environment. The full details of how macro elements are added to the environment are explained in Section 4.4, but in essence the rule is that every time a macro is called its arguments and delimiters plus a set of temporary variables are automatically added to the environment and this supplemented environment is used to evaluate the replacement text of the call. Similarly when a macro label is encountered its position is "remembered" by adding it to the environment.

2.6.7 Insert definitions

It is now possible to define the constituent of the environment, called an insert definition, which is used for such purposes as to tell ML/I to insert a particular argument of a macro at some point in its replacement text. An insert definition consists of:

(a) A delimiter structure. Since all inserts have fixed delimiters and exactly one argument, this delimiter structure will be a simple one. It will consist of a name with a single successor, this successor being a closing delimiter.

(b) An on/off option. If this option is on, an insert is called protected; otherwise it is called unprotected. The use of this option, which need not be of much concern to the average reader, is described in Section 4.5.

At each point where the user wishes something to be inserted he writes the following construction, called an insert:

insert name   argument   delimiter

In the rest of this manual, for the purpose of examples, it will be assumed that the atom "%" is an insert name, with the atom "." as its closing delimiter. With this assumption the following are examples of inserts (the exact meaning of these will become apparent later):

%A6. %P1. %LT2. %WA P9-16*T3.
On encountering an insert, ML/I evaluates the argument of the insert (in case it contains macro calls, etc.) and the resulting value text acts as a specification of what to insert. The value text must consist of a flag followed by a macro expression. In the first above example the flag would be A and the macro expression would be 6. The flag may be null or it may be any of the following: A, B, D, L, WA, WB or WD. Any number of redundant spaces is allowed before, after or within a flag.

The meaning of the various flags are explained below. In each explanation "N" is used to represent the value of the macro expression following the flag. More examples are given in the next Section. An attempt to insert something which does not exist (e.g. the third argument of a macro with only two arguments) results in an error. The meanings of the flags are:

(a) A. This flag is used within the replacement text of a macro to evaluate and insert the Nth argument of a call of the macro. Any spaces at the beginning or end of the argument are deleted before it is evaluated. In the case of this flag and in cases (b) and (c) below the piece of text that is evaluated and inserted is called the inserted text.

(b) B. As case (a) except that spaces are not deleted.

(c) D. As case (b) except that the Nth delimiter, rather than the Nth argument, is inserted. The name of a macro is considered as delimiter zero, and the Nth delimiter is thus the delimiter following the Nth argument.

(d) WA, WB, WD. As cases (a) to (c), respectively, except that the inserted text is not evaluated but is inserted literally, exactly as written. ('W' stands for "written"). The difference between this and the previous cases arises if the inserted text itself involves macro calls, inserts, etc. In the previous cases these are evaluated; in this case they are not.
(e) Null. The numerical value of N, represented as a character string, is inserted. This character string contains no redundant leading zeros. It is preceded by a minus sign if N is negative; otherwise no sign is present.

(f) \( L \). This is used to place a macro label and is rather different from the above cases in that nothing is inserted (i.e. the value of the insert is null). The label N is, if acceptable, added to the current environment and may be the subject of a macro-time GO TO. A macro label is acceptable if it is inserted within a piece of replacement text or inserted text and has not already been defined within that text. It is legal to insert a label in the source text but since, as will be seen later, it is not possible to have a backward GO TO within the source text, such labels are not added to the environment (i.e. they are "forgotten"). Macro labels are local to the piece of text in which they occur, and there is no harm in using the same label numbers within different pieces of text. Label numbers can be chosen arbitrarily, except that they must be positive.

2.6.8 Examples of inserts

The following examples illustrate the use of inserts:

(a) The replacement text of the ESUB macro of Section 2.4 might be written:

\[
\begin{align*}
\text{CMA} \\
\text{ADD} & \%A1. \\
\text{CMA NL}
\end{align*}
\]

or even:

\[
\begin{align*}
\text{CMA} \\
\text{ADD} & \%A1. \\
\text{CMA} & \%D1.
\end{align*}
\]

The latter form would have the advantage of inserting newline or tab
according to which one was written in the call.

(b) In the case of the DO macro of Section 2.4.1 the replacement text would involve an execution-time label. It is imperative that a different execution-time label be generated for each call of DO. This could be achieved by using the initial value of T2. The label could, for example, be written:

\[ ZZ\%T2. \]

In this case if two successive calls of DO occurred at the start of the source text then ZZ1 would be generated at the first call and ZZ2 at the second.

(c) If SWITCH is a macro name with replacement text Pl, then it is possible to write:

\[ \%SWITCH. \]

to insert the first permanent variable. The reason is that the argument of an insert is evaluated before being processed and the call of the SWITCH macro would be performed during this evaluation.

(d) The occurrence of %Al. in the replacement text of the macro call:

\[ MOVE \text{ FROM JACK TO JOHN;} \]

would cause JACK to be inserted, whereas the occurrence of %Bl. would cause JACK enclosed in spaces to be inserted.

(e) If it is desired to insert the name of a macro into its replacement text this can be done by writing "\%WD\%." (The reason for having this facility is that macros can have several alternative names.) In general it would be wrong to use %D\%O. Instead since this form causes any macro calls within the delimiter to be performed. But delimiter zero is the macro name itself and hence an endless recursive loop is likely. In fact when inserting delimiters it is usually better to use a "\w".
This example rather jumps the gun in that it uses the macro-time statements MCSET and MCGO which have not yet been defined. However, if the reader cares to try to understand this example at this stage it may give a useful insight into the purpose of the preceding material. The example shows how the replacement text of the SUM macro could be written. (The comments at the side are for the reader's benefit and do not form part of the replacement text.)

LAC %A1. Generate code to load accumulator with first argument.

MCSET T2 = 1 Use T2 as loop counter.

%L4. MCGO L1 IF%DT2. = + Test if current delimiter is plus

MCGO L2 IF%DT2. = - or minus.

MCGO Lφ If neither then exit. (Lφ has a special meaning, namely "return".)

%L2. ESUB %AT2 + 1. Generate code to substract current argument.

MCGO L3

%L1. ADD %AT2+1. Generate code to add current argument.

%L3. MCSET T2 = T2+1 Increase T2 and continue loop.

MCGO L4

2.7 Skips

The description so far has implied that every occurrence of a macro name in the scanned text is taken as the start of a macro call. This would mean that the user had no easy means of getting macro names or, for that matter, insert names into his value text. Moreover, if he were unfortunate enough to
use a macro name in his comments, then ML/I would take this as a macro call and would start searching for delimiters. To get round these difficulties the user places \textbf{skip definitions} in his environment, and by this means can cause ML/I to ignore comments and to take certain strings as literals.

A skip definition consists of:

(a) A delimiter structure. The names of this structure are called skip names.

(b) Three on/off options. These options are: the text option, the delimiter option and the matched option.

The action of ML/I on finding a skip name is similar to the action on finding a macro name. In both cases a search for delimiters is made until a closing delimiter is found. The text from the skip name to its closing delimiter is called a skip. A skip, therefore, has form:

\textbf{skip name} [argument secondary delimiter \* ?]

In most practical applications of skips there will be exactly one argument. The arguments of skips are treated as literals, exactly as if all macro definitions, insert definitions and warning markers (see later) had been temporarily removed from the environment during the scanning of the skip. There is no replacement text associated with a skip; instead the value of a skip is defined simply by the setting of two of its options. These options which are independent of one another, have the following effect:

(a) If the delimiter option is on, then the delimiters of the skip are copied over to the value text; otherwise they are not.

(b) If the text option is on, then the arguments of the skip are copied over to the value text; otherwise they are not.

As an example of the use of a skip assume the source text contains comments that begin with the word \texttt{COMMENT} and end with a semicolon. In order to skip these comments the user would define \texttt{COMMENT} as a skip name with semicolon as its closing delimiter. In this case if the following comment occurred:

\verb|COMMENT THIS DO LOOP ZEROIZES ARRAY X;|
then its value (i.e. the piece of text copied over to the value text) would be one of the following:

(a) If both options were on, its value would be:

```
COMMENT THIS DO LOOP ZERIOZES ARRAY X;
```

(b) If neither option was on its value would be null.

(c) If only the delimiter option was on its value would be:

```
COMMENT;
```

(d) If only the text option was on then its value would be:

```
THIS DO LOOP ZERIOZES ARRAY X
```

If COMMENT was not defined as a skip at all then comments would normally be copied over to the value text as in case (a). However, if in the above example DO was a macro name then ML/I would try to find the delimiters of DO and replace the call of DO by its replacement text. This is clearly undesirable. The chances are that the entire source text would be scanned without finding the required delimiters. Hence the use of skips is to inhibit the recognition of macro names within certain contexts.

It will be assumed in the rest of this manual that COMMENT is a skip name with semicolon as its closing delimiter.

2.7.1 Matched skips and straight skips

Assume the user has written the comment:

```
COMMENT THIS COMMENT MARKS THE AHLF-WAY STAGE;
```

In this case the skip name COMMENT appears within an argument of the skip COMMENT. However, it is clearly undesirable that ML/I should treat the second COMMENT as a nested skip and try to match it with a semicolon. To prevent this happening COMMENT would be defined as a skip with the matched option off. This is called a straight skip.
However, there are applications of skips where it is desirable for nested skips to be recognised, and such skips have the matched option on. They are called matched skips. Examples of applications of matched skips are "strings" in Algol, which allow nested string quotes and "literal brackets", which are described later. If ML/I encounters any skip name during the scanning of a matched skip it matches the nested skip with its delimiters before matching the containing skip with its delimiters. The scanning process is described in more detail in Section 3.4. In a nest of skips the value is entirely controlled by the options associated with the outermost skip.

2.7.2 Literal brackets

It is usual to have in each environment a skip definition consisting of a name and a closing delimiter with the options set in such a way that at every occurrence of the skip the argument is copied and the delimiters deleted. Such skips are called literal brackets. It will be assumed in the rest of this manual that the name '<' with closing delimiter '>' have been defined as a pair of literal brackets. If it was required to copy a piece of text literally over to the value 'text, ignoring all macro calls and inserts, then the text would be written:

< text >

The process of evaluation would consist simply of removing the literal brackets. Literal brackets always have the matched option on. The reason for this will become apparent in Section 4.2.

2.7.3 Example of a matched skip

The following example, which is rather more complicated than any situation likely to arise in practice, illustrates the full implications of the rules for the matching of skips.

Example In the text:

<AAA < BBB COMMENT < ; CCC > DDD >
the initial "<" is matched with the last ">". (The occurrence of " " after COMMENT is not recognized as a skip name since COMMENT is a straight skip.)

The value of this text is:

```
AAA < BBB COMMENT < ; > CCC DDD
```

This value is independent of how the delimiter and text options for COMMENT are set.

### 2.8 Warning markers

Up to now, ML/I has been described as if every occurrence of a macro name not within a skip is taken as a start of a macro call. In fact this is only true if the environment is in **free mode**.

If he wishes, the user may place the environment in **warning mode** by defining one or more **warning markers**. Any atom or series of atom may be defined as a warning marker. In warning mode each macro call must commence with a warning marker. Optional spaces are allowed between the warning marker and the macro name which follows it. Thus if CALL were a warning marker, the ESUB macro would be called by writing:

```
CALL ESUB X NL
```

In warning mode each occurrence of a warning marker must be followed by a macro name. Any macro name not preceded by a warning marker is not recognized as such.

The essential difference between warning mode and free mode is that in the first case all macro calls have to be specially marked by preceding them with warning markers whereas in the second case all macro names that are not to be taken as macro calls have to be specially marked by enclosing them in skips.

**Note** that warning markers only apply to macro calls, and must not be used to preced inserts or skips. These latter are always recognized, irrespective or the mode of the scan.
2.9 Summary of the environment

All the constituents of the environment have now been defined.
To recap these are:

(a) Macro definitions.
(b) Insert definitions.
(c) Skip definitions.
(d) Warning marker definitions.
(e) Permanent variables.
(f) System variables.
(g) Temporary variables.
(h) Arguments.
(i) Delimiters.
(j) Macro labels.

The term construction is used as a collective name for skips, inserts and macro calls and the term name environment is used as a collective name for constituents (a), (b), (c) and (d) above since the names of these constituents are used to recognize constructions in the scanned text.

2.10* Normal-scan macros and straight-scan macros

This Section explains the difference between normal-scan macros and straight-scan macros. However, straight-scan macros have only limited uses and the reader may choose to skip this Section and assume that all macros are normal-scan.

The difference between the two types of macro arises in the scanning of macro calls. In the case of a normal-scan macro constructions nested within the call are recognized, whereas in the case of a straight-scan macro the effect is as if the name environment were temporarily removed during the scanning of the call. As an example of the use of a straight-scan macro, consider a language where comments are commenced with the word NOTE and ended with a semicolon. Assume it is desired to
use ML/I to map this language into a language where comments are enclosed between the atoms "[" and "]". It is not possible to achieve this transformation by the use of skips since the options on skips do not permit the insertion of extra characters and normal-scan macros are inadequate since it is not desired to recognize macro names within comments. Hence NOTE would be defined as a straight-scan macro. Its replacement text would be:

```
[\%WA1.]
```

The replacement text of a straight-scan macro is evaluated in exactly the same way as that of a normal-scan macro.

The reader will no doubt have noticed that there is an analogy between the two types of macro and the two types of skip. In fact any straight skip can be represented as a straight-scan macro. However, straight skips are preferable, where possible, since they are slightly easier to define and much faster in execution. The analogy between normal-scan macros and matched skips is not so close. Normal-scan macros permit any constructions to be nested within calls of them whereas matched skips only allow further skips to be nested within them.

The straight-scan option can only apply to user-defined macros; it cannot apply to inserts or to operation macros (see Section 4.1).

### 2.11 Name environment used for examples

To avoid unnecessary repetition, a fixed name environment will be assumed in all subsequent examples. This environment consists of:

(a) The atom "%" with closing delimiter "." as an insert definition.
(b) The atoms "<" and ">" as literal brackets.
(c) COMMENT as a straight skip with closing delimiter semicolon.
(d) The DO and MOVE FROM macros of Section 2.4.1.
(e) The ESUB macro of Section 2.4.
(f) No warning markers.

All the macros above are taken to be normal-scan.
Chapter 3 Text scanning and evaluation

3.1 Nesting and recursion

Constructions may be nested to any desired depth, and may appear within replacement text. Furthermore, recursive macro calls are allowed. In other words, any construction is allowed with any piece of replacement text or inserted text, and a macro may be called while evaluating its own replacement text. However, constructions must be properly nested. This means that each construction must lie entirely within a single piece of replacement text, entirely within a single piece of inserted text or entirely within the source text. Apart from this obvious restriction, ML/I contains no restrictions on nesting and recursion.

As a result of nesting and recursion, the process of text evaluation is in general a recursive one. At the beginning of a process ML/I starts evaluating the source text. During this evaluation it will in general encounter a macro call. This will cause it to temporarily suspend the evaluation of the source text and start evaluating the replacement text of the call. While evaluating this replacement text, ML/I may encounter an insert, and this will cause it to suspend the evaluation of the replacement text and start evaluating some inserted text. Alternatively, it may encounter a nested macro call. Thus at any one time several pieces of text may be in the process of evaluation.

This situation is liable to lead to ambiguities in terminology, so it is necessary to clarify some of the terms that will be used. The terms "the scanned text", "the current environment" and "the current point of scan" will always refer to the text actually being evaluated, not to any piece of text whose evaluation has been temporarily suspended. ML/I is said to be evaluating inserted text if the scanned text is inserted text, and a similar definition applies to "evaluating replacement text". ML/I is said to be evaluating the source text if it is not within the evaluation of any macro calls or inserts.

3.2 Call by name

Arguments and delimiters are evaluated each time they are inserted, rather than when the call in which they occur is scanned. In other words they are "called by name" rather than "called by value". In most cases, of course, this choice of
approach makes no difference to the final result, but it does have an effect if the environment changes between the time an argument is scanned and the time it is inserted.

3.3 Details of the scanning process

When text is evaluated it is scanned atom by atom until the end is reached. All text, whether the source text, replacement text or inserted text, is scanned and evaluated in the same way. In general each atom of the scanned text is compared with all the names in the environment to see if a match can be found. However, as was seen in the previous Chapter, some types of name are not recognized under certain circumstances. The complete list of such circumstances is as follows:

(a) No names are recognised within a straight skip or straight-scan macro call.

(b) Apart from skip names, no names are recognized within a matched skip.

(c) In warning mode macro names are not recognized except after warning markers. Immediately after a warning marker no names except macro names and no secondary delimiters are recognized (unless an error occurs, see Section 8.3.4).

When a construction name is found a search is made for its closing delimiter. This process is described in the next Section.

Some names in the environment may consist of more than one atom. In this case when an atom of the scanned text is found to match the first atom of the name the scanning process looks ahead to see if the remaining atoms of the name follow this atom. (This look-ahead is abandoned if the end of the current text is reached.) If a match is found scanning is resumed beyond the last atom of the name. The user can specify for each pair of atoms of a multi-atom name whether spaces between the atoms are to be ignored by the scan. Multi-atom secondary delimiters are matched in exactly the same way as multi-atom names.
Apart from these cases of multi-atom delimiters the scan always proceeds atom by atom. Each atom not within a construction is copied over to the value text. Atoms within skips may or may not be copied according to the option settings. Atoms within macro calls or inserts are never copied over to the value text since the very purpose of these constructions is to perform a replacement.

3.4 The method of searching for delimiters

When ML/I encounters a construction name, it searches for each of the secondary delimiters until the closing delimiter is found (except in the case where the construction name is its own closing delimiter, when no searching is required). In general an error message (see Section 8.3.5) is given if the end of the current piece of text is reached before the closing delimiter has been found. In this case the construction is said to be unmatched. Exclusive delimiters, however, provide a slight exception to this rule (see next Section). If, during the search for the delimiters of a construction, a nested construction is encountered, then the search for the delimiters of the outer construction is suspended until the closing delimiter of the nested construction has been found. Nested constructions can only arise within inserts, matched skips and normal-scan macros. Since arguments are called by name rather than by value, nested constructions are not evaluated when scanned over during the search for delimiters of a containing construction. Evaluation occurs only when the argument containing the nested construction is inserted.

The process of searching for closing delimiters is illustrated by the following rather pathological example (remember that the name environment of Section 2.11 applies to this and all subsequent examples).

```
DO 3 TIMES < REPEAT DO >

ESUB REPEAT

DO REPEAT TIMES

REPEAT

REPEAT
```

In this example the first DO is matched with the last REPEAT, since the search
for the REPEAT for this first DO is suspended during the scanning of the nested constructions \( \langle \), ESUB and DO. Furthermore the occurrence of DO within the literal brackets is not recognized as a macro name.

In general, a single closing delimiter cannot terminate two separate constructions. Thus two successive REPEATs are needed in the above example to close both the DO macros. However, exclusive delimiters again provide an exception to the rule.

As a further example, if the user were foolish enough to write:

```
MOVE FROM TO TO PIG;
```

then the first TO would be taken as the delimiter of MOVE FROM. What he should write to make the second TO the delimiter is:

```
MOVE FROM \(<\) TO \(>\) TO PIG;
```

However, there is nothing wrong with writing:

```
MOVE FROM PIG TO TO;
```

In practice, if delimiter names are chosen sensibly, problems such as the above rarely arise.

3. 5* Exclusive delimiters

(It is highly recommended that this Section be skipped on a first reading as it describes a rather complicated feature which is only occasionally needed.)

In the normal way, after a construction has been scanned over and replaced by its value, scanning is resumed with the atom following the closing delimiter of the construction. Hence the closing delimiter is taken as part of the construction. In a few cases, however, it is more convenient to regard the closing delimiter as external to the construction. Such a delimiter is called an **exclusive delimiter**. Only macros and skips may have exclusive delimiters and exclusive delimiters are always closing delimiters. After a construction with an exclusive delimiter has
been dealt with, scanning is resumed at the exclusive delimiter rather than beyond it.

Exclusive delimiters are useful when it is desired to use a single delimiter as a closing delimiter of several nested constructions. For example an IF macro might have form:

```
IF condition THEN nested macro call NL
```

where the nested macro call is terminated, like IF, by the closing newline. In this case, it would be necessary to define newline as an exclusive delimiter of any macro that could be nested within the IF macro. Then when the scan had used the newline to close the nested macro call it would re-scan it and use it again to close the IF macro.

A difficulty arises in the above example when, within the replacement text of IF, the second argument is inserted. The problem is that the nested macro call is unmatched within this argument, since its closing delimiter, the newline, lies beyond the end of the argument. ML/I resolves this problem by using the following rule; if, when inserting the Nth argument of a macro call, a construction is unmatched then the Nth delimiter is examined and if this delimiter (or a series of atoms at the start of it) is an exclusive closing delimiter which closes the apparently unmatched construction then this construction is considered as matched and processing proceeds normally. If there is a nest of unmatched constructions then this rule is successively applied to all the constructions in turn. (In fact this rule is such a natural one that the user might not realize that there is any logical problem at all.)

Note that it is quite legal to insert an exclusive delimiter in the replacement text of the macro call to which it belongs. It is even legal to define a name delimiter as an exclusive delimiter (though this is almost certain to lead to an endless loop). Furthermore it is quite legal to have both exclusive delimiters and ordinary closing delimiters within the same delimiter structure.

If a skip ends with an exclusive delimiter this closing delimiter is not taken as part of the skip and hence it is not affected by the delimiter option associated with the skip.
Exclusive delimiters are sometimes useful in simple applications where no nesting is involved. For instance it is often desirable for a skip to delete up to, but not including, the next newline.

As a more complicated example, consider a language in which macro calls were one to a line with the macro name coming first. In this case it might be convenient to give newline a double use: firstly as an exclusive delimiter of the macro on the previous line and secondly as a warning marker to precede the macro name on the next line.

The way exclusive delimiters are defined is described at the end of Section 5.1.3.

3.6* Dynamically generated constructions

The method of scanning, with the requirement that calls be properly nested, means that all the delimiters of a construction must be in the same piece of text. This rule, which is very desirable since it leads to the early detection of genuine errors, should be borne in mind by the user who wishes to generate constructions dynamically, for example to combine at macro-time separate pieces of text to build up a macro call. The rule prohibits constructions like:

```
CHOOSENAME A TO B;
```

where CHOOSENAME is a macro with replacement text MOVE FROM, or constructions like:

```
DO A %A1. B REPEAT
```

where %A1. has value TIMES. It is however, quite easy to achieve the object of these examples, namely to generate a delimiter dynamically, and the reader who is interested in doing this should refer to the example in Section 9.4.3.
Chapter 4  Operation macros and their use in setting up the environment

4.1 Operation macros

The macros considered so far have been concerned with making replacements of pieces of text. In fact, strictly speaking, they should have been called substitution macros. There is a second type of macro called an operation macro. A call of an operation macro causes a predefined system action to take place, for example the setting up of a new construction. Operation macros are an integral part of ML/I and are not, like substitution macros, defined by the user. They are, however, part of the name environment and are called in the same way as substitution macros. Examples of operation macros are MCSET (which performs macro-time arithmetic), MCDEF (which defines a macro), and MCGO (which is a macro-time conditional GO TO statement). Examples of their calls are:

\[
\begin{align*}
\text{MCSET} & \quad P_1 = P_2 + 1 \\
\text{MCDEF} & \quad \text{LNG AS LENG} \\
\text{MCGO} & \quad L6 \text{ IF } ZA1. = \text{ACC}
\end{align*}
\]

Chapter 5 contains complete descriptions of all the operation macros. The names of all operation macros begin with MC to minimize confusion with substitution macros. (The user is not forbidden to start his own macro names with MC, but it is probably less confusing not to.)

The arguments of all operation macros are evaluated before being processed. Thus if TEMPNO were a macro with replacement text P1, then the following would be equivalent to the previous example of MCSET:

\[
\text{MCSET} \quad \text{TEMPNO} = P_2 + 1
\]

In most cases a call of an operation macro does not cause any value text to be generated. No value text would be generated, for instance, in any of the examples above. However, there are two operation macros, MCSUB and MCLENG, which do cause value text to be generated. These two macros are called system functions. MCSUB is used for generating substrings of longer pieces of text and MCLENG is used to calculate the length of a piece of text.
There are no general restrictions on the use of operation macros. They may be called from within any type of text, even from within arguments to other operation macros.

4.2 Use of literal brackets for surrounding operation macro arguments

The fact that arguments of operation macros are evaluated before being processed has several advantages but it also has its dangers, and in many cases the user will wish to inhibit this argument evaluation. Consider as an example the last argument of MCDEF, which specifies the replacement text of the macro being defined. A definition might be written:

```
MCDEF ... AS <...%Al. ...>
```

If the above literal brackets had been omitted, ML/I would have tried to insert the value of argument one at the time the macro was defined (called definition time) rather than when it was called, and an error would probably result. Occasionally, however, a user might want to do this, in particular when one macro is defined within another and the arguments of the outer one figure in the definition. Apart from cases like this it is a good plan to use literal brackets whenever specifying the replacement text of a macro.

Another reason for the usage of literal brackets arises when the replacement text involves one or more newlines, e.g.

```
MCDEF ... AS <LINE 1
LINE 2
>
```

In this case, since newline is also the closing delimiter of MCDEF, the newlines within the replacement text need to be prevented from closing the MCDEF. The literal brackets, being a construction nested within the call of MCDEF, achieve this.

It is now possible to see why literal brackets must be defined as matched skips rather than straight skips. Consider the following example, where a piece of replacement text itself contains a call of MCDEF:

```
MCDEF MAC1 AS <...
MCDEF MAC2 AS <...>
... COMMENT >;
```
It is vital that the first "<" be matched with the last ">" and not with the occurrence of this symbol in a comment nor with the occurrence in the nested MCDEF. The definition of literal brackets as a matched skip accomplishes this.

4.3 NEC macros

Many of the operation macros have the effect of adding to or deleting from the name environment. These macros are called NEC (name environment changing) macros. The name environment is set up dynamically by calls of NEC macros during text evaluation. The initial state of the name environment is implementation-defined (see Section 2 of the Appendix) but it will usually contain just the operation macros. Changes in the environment affect subsequent text evaluation but have no effect on value text already generated. Constructions may be defined as either local or global. Global constructions apply to all subsequent text evaluation, whereas local constructions apply only to the text in which they are defined, together with any macros called from within this text (for exact details see next Section). A local definition occurring in the source text has the same effect as a global definition.

To start with, most users will probably not be very interested in defining new macros in the middle of text evaluation. In this case the entire name environment can be set up by a series of NEC macro calls at the start of the source text, and all the rest of the text can be evaluated using this name environment. Local definitions should be used in preference to global ones where possible since the setting up of global definitions involves more work for ML/I. (Normally global definitions are only necessary when it is desired to use one macro to set up the definition of another.)
A reader who is not interested in changing the name environment dynamically can skip the next two Sections. He can, in fact, totally ignore global definitions and he need not worry about the difference between protected and unprotected inserts.

4.4* Dynamic aspects of the environment

The value of a piece of text depends upon the state of the environment when its evaluation is started. The purpose of this Section is to define the initial state of the environment when replacement text or inserted text is evaluated, and to explain the effect of dynamic changes in the name environment.
It is convenient to divide the name environment into two parts:

(a) The **global name environment**, which contains the names of global constructions. Operation macro names are treated as global.

(b) The **local name environment**, which contains the names of local construction.

If a substitution macro is called or if an argument or delimiter is inserted, this cannot change the local name environment of the containing text. However, any change in the global name environment applies to the subsequent evaluation of the containing text. In other words there is a single global name environment but each piece of text in the process of evaluation has its own particular local name environment.

When a substitution macro is called, the replacement text is evaluated under the following initial environment:

(a) the global name environment in effect when the call is made.

(b) the local name environment in effect when the call is made.

(c) the permanent and system variables.

(d) the arguments and delimiters of the call.

(e) a set of temporary variables. These are allocated when the call is made. The number allocated is given by the capacity of the macro called.

(f) no macro labels.

When an operation macro is called, no special environment is set up and no temporary variables are allocated. The arguments of the operation macro are evaluated under the environment in force when the call was scanned. The same applies to the argument of an insert.

Before considering the initial environment for the evaluation of inserted text, it is instructive to consider an example that will illustrate the reasons behind the rules. This example involves passing arguments down from one macro to another.
Assume ABC is called with TEMP as its first argument. Then if "%" has been defined as a protected insert the value of %Al. is TEMP. If it has been defined as an unprotected insert the value is LMN. (MCDEF defines a local macro. If MCDEFG, which defines a global macro, had been used in place of MCDEF then the value of %Al. would always be LMN.) Hence the purpose of a protected insert is to protect the insertion of a macro's arguments or delimiters from any changes in the local environment of the macro's replacements text. It is often useful, for instance, to switch into warning mode when entering the replacement text of a macro but still to evaluate its arguments in free mode. In some applications the user may wish to define two insert names, one protected and the other unprotected. In most applications, however, it will be entirely immaterial which sort of insert is defined.

To complete the definition of the previous Section, the initial local name environment when inserted text is evaluated is as follows:

(a) If the insert is a protected insert then it is the local name environment that was in force when the call containing the inserted text was encountered.

(b) If the insert is an unprotected insert then it is the local name environment that was in force when the insert was encountered.

4.6* Ambiguous use of names

When defining new constructions the user should be careful to avoid certain clashes of name. It would obviously be foolish, for instance, to choose the name MCDEF for a new construction. ML/I has a fixed set of priority rules for dealing with multiply-defined names, and these are listed below. However, for the reader who is not interested in these complications the following simple rule for defining new constructions is sufficient to avoid difficulty: choose the delimiters to be different from all other environmental names (i.e. the names of macros, inserts, skips and warning markers in the current environment). It is quite acceptable, of course, to choose the same representation for the secondary delimiters of different constructions. For example, all macros could have a newline as their closing delimiter. Furthermore it is perfectly in order to have several different names all
Assume that within the replacement text of a macro XYZ it is desired to call the MOVE FROM macro to move the second argument of XYZ into a place called TEMP. This call of MOVE FROM would be written:

```
MOVE FROM %A2. TO TEMP;
```

This call would cause the replacement text of the MOVE FROM macro to be evaluated and during this evaluation it would be necessary to insert the first argument of MOVE FROM. The insertion of this argument involves the performing of the insert "%A2.". Now in this case ML/I takes A2 to mean the second argument of XYZ, not the second argument of MOVE FROM. The initial state of the environment for the evaluation of inserted text is set to make this so. This initial environment consists of:

(a) the current global name environment.

(b) a local name environment. This depends on whether the insert is protected or unprotected. See next Section.

(c) the permanent and system variables.

(d), (e) the arguments, delimiters and temporary variables that were in the environment when the call containing the text to be inserted was encountered.

(f) no macro labels.

The reader may have noticed that no initial environment contains any macro labels. This is because it is not possible to use the MCGO macro to jump from one piece of text to another. Thus each piece of text has its own macro labels, and macro labels are not carried down from one piece of text to another.

4.5* Protected and unprotected inserts

The difference between protected and unprotected inserts is best illustrated by an example. Consider a macro ABC whose replacement text starts as follows:

```
MCDEF TEMP AS LMN
%A1.
```
beginning with the same atom(s); for example, three separate macros could have names RETURN, RETURN TO and RETURN IF. ML/I always tries to find the longest name it can, so in this example it would only call the RETURN macro if RETURN was not followed by TO or IF. The reader who is prepared to adopt the simple rule above can skip the rest of this Section.

A name clash is considered to occur if an atom or series of atoms of the scanned text can be interpreted in more than one way. Note that some environmental names are ignored within certain contexts (see Section 3.4 for a complete list) and thus a name can sometimes be multiply-defined without a clash occurring. For example, in warning mode it is unambiguous to have a macro name the same as an insert name since each is recognized in a different context.

When a name clash does occur, the following rules are applied in order until all ambiguity is removed:

(a) Exclusive delimiters take precedence over everything else.
(b) A longer delimiter takes precedence over a shorter one (as illustrated by the above RETURN example).
(c) Secondary delimiters take precedence over environmental names.
(d) Local environmental names takes precedence over global ones.
(e) The most recently defined environmental name takes precedence.

4.7* Implications of rules for name clashes

Some implications of the rules in the previous Section are:

(a) A construction may be overridden by redefining it. It is even possible to redefine a macro within its own replacement text. If it is desired to achieve the effect of deleting a macro name PQR from the environment this can be achieved by defining PQR as a skip using the MCSKIP macro of Section 5.2.3 as follows:

MCSKIP D, <PQR>

(PQR is enclosed in literal brackets to prevent it being called.)
This technique can be used for all construction names. Note that when a construction is redefined its old use is not completely deleted (no storage is released) and it is possible under some circumstances to re-incarnate the old usage. For example the overriding use may have restricted scope or it may be deleted by one of the macros of Section 5.2.5, such as MCNOSKIP.

(b) It is usually acceptable to choose a construction name to be the same as the secondary delimiter of another construction. For instance there is no harm in choosing IF as a macro name even though it is a delimiter of MCGO. The only restriction on the use of IF would be that it could not be called within the first argument of MCGO. (This restriction only applies in free mode. In warning mode there would be no restriction.)

(c) A technique (described in Section 9.4.8) can be designed to give constructions different meanings in different scopes.

(d) If it is desired to design a language where each macro call occupies one line, it is practicable to define newline as an exclusive delimiter of each macro and also as a warning marker or as a part of a composite macro name (for instance NL GO TO could be a macro name).

(e) If each of GO, GO TO, and TO THE END are macro names then

\[
\text{GO TO THE END}
\]

is interpreted as a call of GO TO, not as a call of GO and a call of TO THE END. This is because the rules of the previous Section are applied at each step in the scan. There is no mechanism for looking ahead and thus deciding, for instance, to take a shorter delimiter at one step in order to get a longer one later.
This chapter contains descriptions of the operation macros which should be present in every implementation. In addition, each implementation may have its own particular operation macros (see Section 1 of the Appendix).

Arguments of operation macros are evaluated before being processed in the same way as arguments of substitution macros. Leading and trailing spaces are deleted before evaluation in all cases.

Descriptions of the operation macros have been arranged in a standard format which consists of a number of subsections. These subsections in order of occurrence are described below.

1. **Purpose.**
2. **General form.**
3. **Examples.** Examples may not be comprehensible until further subsections have been read. Each example is independent of all others.
4. **Restrictions.** This subsection describes any restrictions on the form that the values of the arguments of the macro can take. If this subsection is omitted there are no restrictions. The notation "ARG X" is used to represent the value of the argument corresponding to arg X in the General Form.
5. **Order of evaluation.** This subsection describes the order in which arguments are evaluated. It is omitted if the order is sequential. The order of evaluation is, of course, immaterial in all but the most pathological cases. Note that any change in the name environment caused by the call of a NEC macro does not come into effect until after all its arguments have been evaluated. It is possible for an operation macro to be aborted due to an error before all its arguments have been evaluated.
6. **System action.** This subsection describes the action performed by ML/I at a call of the macro. A reference to "the current environment" means the environment in force when the macro was called. Apart from
the system functions, all operation macros have a null value.

(7) Notes. This subsection contains nothing new, but attempts to bring out more clearly points implied by the preceding material.

Before describing the individual operation macros it is necessary to describe how to define delimiter structures, since all the operation macros which define new constructions have an argument which specifies the delimiter structure of the construction.

5.1 Specification of delimiter structures

Delimiter structures are defined by writing a structure representation, which defines all the delimiters in the structure and the successor(s) of each. The atoms that make up a delimiter are specified by a delimiter name, which is written in the following way:

\[
\text{(WITH )} \quad \text{atom} \quad [ \quad \text{atom}^* \ ? \ ]
\]

\[
\text{(WITHS )}
\]

The difference between WITH and WITHS is as follows. If two atoms are linked by WITHS, this means that any number of spaces (including none) may occur between the atoms when the delimiter is used. WITH, on the other hand, means that no intervening spaces are allowed.

As an example, the delimiter names of a macro of form:

\[
\text{COMPARE CHARACTERS argument 1 /// argument 2;}
\]

would be:

(1) COMPARE WITHS CHARACTERS
(2) / WITH / WITH/
(3) ;

If, for some reason, it was desired to restrict the number of permissible spaces between COMPARE and CHARACTERS to one, then this would be specified by:

(1a) COMPARE WITH SPACE WITH CHARACTERS
Note that at least one space must be allowed between COMPARE and CHARACTERS because otherwise they would not be recognized as separate atoms. Thus, in the general case, a delimiter name is in error if two atoms are connected by WITH and neither atom is a punctuation character.

It is now necessary to consider how delimiter names are combined to form a structure representation. In the simplest case, the case of a construction with fixed delimiters, this is done simply by concatenating the delimiter names in the order in which they are to occur. Thus the complete structure representations of some of the constructions used as examples in this manual (see Section 2.11) are:

(a) %.
(b) < >
(c) COMMENT;
(d) DO TIMES REPEAT
(e) MOVE WITHS FROM TO;

5.1.1. Keywords

Within a structure representation the atoms are separated out by layout characters, i.e. spaces, newlines, tabs, etc. (In the above examples spaces have been used.) Apart from acting as separators, layout characters are totally ignored within structure representations. Thus a problem arises when it is desired to specify a layout character as a delimiter, or as a constituent atom of a multi-atom delimiter. This problem is overcome by using layout keywords to stand for layout characters. In particular:

SPACE means a space
TAB means a tab
SPACES means a sequence of one or more spaces
NL means a newline

In addition each implementation may have its own extra layout keywords. See Section 6 of the Appendix for details. The characters represented by these keywords are treated as layout characters and hence, within structure representations, are exactly equivalent to newlines or spaces. Note that layout keywords only apply within structure representations.
The following are examples of delimiter structures using layout keywords

(a) ESUB NL
(b) SPACE
(c) SPACE WITH SPACES (means two or more spaces)
(d) LD WITH SPACES SPACES NL

A construction defined using (d) above, would be analysed thus:

```
LD | X | Y NL
```
delimiter 0          delimiter 1          delimiter 2

Note how all the spaces following LD are absorbed into the name; if they had not been defined to be part of the name they would have been taken as the first delimiter.

It is permissible to use SPACES before or after WITHS; in these cases it is exactly equivalent to SPACE.

In addition to these layout keywords, there are other keywords that apply within structure representations. These are: WITH, WITHS, OPT, OR, ALL and any atom commencing with the letter 'N' followed by a digit. Keywords are reserved words and can not be used as the atoms of delimiters. However, if it is necessary to define, say, WITH as a delimiter name, then the keyword WITH could be changed to something else (e.g. '+') by using the MCALTER macro described in Section 5.2.7.

5.1.2 The consequences of evaluation

Since structure representations occur as arguments to operation macros they are evaluated before being processed. Two consequences of this, one beneficial to the user and the other a nuisance, are as follows.

The beneficial consequence is that much-used alternatives can be artificially generated. Assume, for example, that a large number of macros have the form:

```
NAME (argument) NL
```

where NAME varies from macro to macro. In this case it would be useful to define a macro PARENS with replacement text:
WITH ( ) WITH NL

Then a macro DOG of the above form could be defined by writing:

```
DOG PARENS
```

The mischievous consequence arises if an attempt is made to redefine a macro. Assume that a macro EMPLOYEE is defined thus:

```
MCDEF EMPLOYEE AS < J. SMITH >
```

and then subsequently an attempt is made to redefine it by writing:

```
MCDEF EMPLOYEE AS < J. BLOGGS >
```

In this second definition the structure representation is J. SMITH since EMPLOYEE is replaced by its value. Hence a macro J would be defined with secondary delimiters "." and SMITH. The end result would probably be a puzzling error message, perhaps that a delimiter of the macro J was missing.

To avoid problems such as this it is imperative to enclose a name in literal brackets if it is being redefined. The same applies if the name of one macro occurs as a delimiter of another. In fact it is not a bad rule to enclose all structure representations in literal brackets except where constructions such as PARENS are being used. The correct way to redefine EMPLOYEE would be:

```
MCDEF < EMPLOYEE > AS < J. BLOGGS >
```

5.1.3* Introduction to more complicated cases

The Sections which follow describe facilities for setting up more and more elaborate delimiter structures. The reader is recommended to read on until he knows enough for his own applications and then to skip the rest. Readers who are only interested in fixed delimiters may give up now.

In order to specify the delimiter structure of a construction it is necessary to specify the name(s) of the construction and the successor(s) of each delimiter that is not a closing delimiter. In the simple cases described above the structure representation consisted of the name of the construction and then each succeeding delimiter followed by its successor until the closing delimiter. In more complicated cases it is necessary to have two other mechanisms for specifying successors, namely
option lists and nodes. Furthermore it is convenient to imagine that a special symbol $\alpha$ occurs at the start of each structure representation and another symbol $\omega$ at the end. With this convention any successor of $\alpha$ is a name of the construction and any delimiter with $\omega$ as successor is a closing delimiter. The paragraphs which follow contain informal introductions to the concepts of option lists and nodes. More exact details are given in the next Section.

Option lists are used to specify that a delimiter has several optional alternatives as successor. The essential form of an option list is:

$$\text{OPT branch}_1 \text{ OR branch}_2 \text{ OR } \ldots \text{ OR branch}_N \text{ ALL}$$

The ordering of the branches is immaterial. An example of the use of an option list is in the following structure representation for the ESUB macro:

$$\text{ESUB OPT TAB OR NL ALL}$$

If, in addition, it was decided to allow SUBTRACT as an alternative name to ESUB, then its structure representation would be:

$$\text{OPT ESUB OR SUBTRACT ALL OPT TAB OR NL ALL}$$

In the ordinary way the successor of the delimiter at the end of a branch is taken as the delimiter following the ALL concluding the option list. In other words the branches may be thought of as coalescing at the delimiter following ALL. (Thus in the example above both ESUB and SUBTRACT have either tab or newline as alternative successors and both tab and newline have the imaginary symbol $\omega$ as successor and are therefore closing delimiters.) However, as will be seen, it is possible to override this coalescing effect by the use of nodes.

Nodes are used for defining the successor of a delimiter to be a delimiter or option list elsewhere in the structure representation. The use of nodes in structure representations is analogous to the use of labels in programming languages. As the reader will know, the statements in a programming language are written in sequence and the "successor" of each statement is normally taken as the statement which follows. However, the user can specify a different successor by the use of labels. A label is "placed" on one program statement and is then "gone to" after any program statement which requires the labelled statement as successor. In exactly the same way, nodes are used to specify the successors of delimiters.
A node is represented by a **node flag** followed by a positive integer. The normal node flag is the letter 'N' but this can be changed if desired using the MCALTER macro of Section 5.2.7. It will be assumed in this manual that the node flag is 'N'. A node is placed by writing its name before any delimiter name or option list. A node can be "gone to" only from the end of a branch of an option list or at the end of a structure representation. A "go to" is indicated simply by placing the name of the appropriate node at the desired point. (Although the name of a node is used both to place it and to go to it, there is no ambiguity owing to the different context in which each occurs.) As a simple example of the use of nodes, consider the structure representation of a SUM macro which allows any number of arguments separated by plus or minus signs and terminated by a semicolon. A typical call of SUM would be:

```
SUM A + B - C + D ;
```

The structure representation of SUM is:

```
SUM N1 OPT + N1 OR - N1 OR ; ALL
```

This is interpreted thus. SUM is followed by either a plus sign, a minus sign or a semicolon. Node N1 is placed before the option list. The successor of both plus and minus is defined by going to N1, and N1 is associated with the alternative plus, minus and semicolon. The successor of semicolon, on the other hand, is taken as the delimiter which follows ALL, which is ω. Hence the semicolon is a closing delimiter.

There are no particular restrictions on the use of nodes. Any number of nodes may be placed within a structure representation provided, of course, that they have different numbers. Any positive integers may be chosen to designate nodes; no particular sequence is required. Node numbers are local to the structure representation in which they occur and hence there is no relation between the nodes of one structure representation and those of another. Thus the same node numbers may be used in each case. There are no restrictions on the scope of "go to"; thus it may dive into an option list or alternatively come out of one.

The node N₀ (N zero) has a special usage, namely to denote an exclusive delimiter. Node N₀ may be gone to but it may not be placed. If the successor of a delimiter is specified by N₀ then this delimiter is taken as an exclusive delimiter. Apart from N₀, it is illegal to go to a node without placing it.
5.1.4* Full syntax of structure representations

Before describing the general form of a structure representation it is necessary to describe a number of syntactic sub-components. These are:

(a) A nodeplace represents the placing of a node and is specified by the node flag followed by an unsigned positive integer.

(b) A nodego represents the action of going to a node and is also specified by the node flag followed by an unsigned integer. (In this case and case (a) above any redundant leading zeros are ignored.)

(c) A delspec represents the specification of a delimiter or an option list and is of form:

\[
\begin{array}{l}
\text{[ nodeplace ? ] (delimiter name) (OPT branch [ OR [ nodeplace ?] branch *?] ALL) }
\end{array}
\]

where a branch is of form:

\[
\text{delimiter name [ delspec * ?] [ nodego ? ]}
\]

(The reader may like to look ahead to the examples in the next Section at this point.) Note that each branch must begin with a delimiter name, called the branch name. The branch names are the possible alternative successors of the delimiter preceding the option list, and must all be different. Thus no sequence of atoms must match more than one branch name, and the following option list is therefore incorrect:

\[
\text{OPT X WITH SPACE WITH Y ... OR X WITHS Y ... ALL}
\]
since "X Y" could be the name of either branch.

As was seen from the preceding example of the SUM macro, nodeplaces immediately preceding an option list associate the node with all the options of the list. The syntax forbids a nodeplace immediately after OPT and a nodeplace immediately following OR has a special meaning in that it associates the node not only with the delimiter name that follows it but also with the names of all subsequent branches of the option list. As an example, assume that the SUM macro was extended to allow the user the option of assigning the answer by writing, for example:

\[
\text{SUM X = Y + Z; to calculate Y + Z and assign the answer X,}
\]
or

\[
\text{SUM Y + Z; to calculate Y + Z and leave the answer in an accumulator.}
\]
Here \( \text{SUM} \) has an optional first argument delimited by an equals sign. Its structure representation could be written:

\[
\text{SUM} \ \text{OPT} = \text{N1 OR N1} + \text{N1 OR - N1 OR; ALL}
\]

In this case \( \text{N1} \), which is placed after the first OR, is associated with the alternatives plus, minus and semicolon.

Now that the sub-components have been described it is possible to give the general form of a structure representation. This is:

\[
[ \text{delspec} \ 0 ] \ [ \text{nodego} \ ? ]
\]

One last point should be made about the writing of structure representations. This concerns minimizing the amount of storage that it is needed to store a delimiter structure. The storage used is a function of the number of delimiter names in the structure. Thus it is advisable to try to link a structure together in such a way that it contains the minimum number of delimiters. As an example of redundancy, consider the following structure representation:

\[
\text{BUMP OPT TIMES OR ALL}
\]

This represents a construction of form:

\[
\text{BUMP} \ [ \text{argument TIMES?} ] \ \text{argument}
\]

Note that the semicolon is repeated within the structure representation of \( \text{BUMP} \). However, this repetition can be avoided by writing the structure representation in the following improved way:

\[
\text{BUMP OPT TIMES N1 OR N1; ALL}
\]

5.1.5* Examples of complex structure representations

This section contains the general forms of some possible constructions together with the structure representation of each.

**Example 1**

**General form**

Either \( \text{BUY} \ \text{arg A} \ \$ \ \text{arg B. arg C;} \)

or \( \text{BUY} \ \text{arg A POUNDS arg B S arg C D arg D;} \)

**Structure representation**

\( \text{BUY} \ \text{OPT \$;} \ \text{OR POUNDS S D ALL;} \)

In the second form, if it is desired to allow the \( S \) and \( D \) fields optionally to be omitted, then the structure representation could be written:
BUY OPT $ . ; OR POUNDS OPT S N1 OR N1 D ; OR ; ALL ALL

Here N1 is associated with the possibilities D and semicolon. In this form the semicolon is mentioned three times. The structure representation is therefore improved by writing it in the following form, there semicolon only occurs once:

BUY OPT $ . N2 OR POUNDS OPT S N1 OR N1 D N2 OR N2 OR N2 ; ALL ALL

(The diagram in the next Section may be an aid to understanding this.)

Example 2

General form  [ / argument * ? ] END

Structure representation  N1 OPT / N1 OR END ALL

This macro has two possible names: "/" and "END".

Example 3

General form  ( LOAD ) arg A, arg B NL
( STORE )

where the newline is an exclusive delimiter.

Structure representation  OPT LOAD OR LOAD WITHS Q OR STORE ALL, NL

5.1.6 Possible errors in structure representations

Great care must be taken in writing structure representations as errors can have very unfortunate results. In complex cases it may be useful to use a diagram. For example the following represents the BUY macro of the previous Section in its final improved form.
Special points to be watched in writing structure representations are the use of keywords and the possible differences between the structure representation as written and its evaluated form. Remember that key words cannot be used as delimiter names.

If ML/I does reject a structure representation as illegal (giving the message of Section 8.3.6), then the following are some of the possible causes:

(a) Illegal syntax, for example: unmatched OPT, node after OPT, two nodes in succession, branch without a name, placing of node zero, names such as NIA.

(b) Keyword used as delimiter.

(c) Undefined or multiply-defined node.

(d) Two branches with the same name.

(e) Misuse of WITH or WITHS e.g. GO WITH TO, X WITHS N1.

(f) Structure with no closing delimiter.

(g) Unconnected structure. For example the delimiter D is not connected to the main structure in the following case:

\[
\text{NOGOOD N1 OPT A N1 OR B N1 ALL D}
\]

5.2 The NEC Macros

The operation macros which change the name environment are listed in this Section.

5.2.1 MCWARN

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Definition of a local warning marker.</th>
</tr>
</thead>
<tbody>
<tr>
<td>General form</td>
<td>MCWARN arg A NL</td>
</tr>
<tr>
<td>Examples</td>
<td>(a) MCWARN $</td>
</tr>
<tr>
<td></td>
<td>(b) MCWARN CALL WITHS THIS WITHS MACRO</td>
</tr>
<tr>
<td>Restrictions</td>
<td>ARG A must be a structure representation consisting simply of a single delimiter name.</td>
</tr>
<tr>
<td>System action</td>
<td>ARG A is added to the current environment as a local warning marker and the current environment is placed in warning mode.</td>
</tr>
</tbody>
</table>
5.2.2 MCINS

Purpose
Definition of a local insert.

General form
MCINS [ arg A, ? ] arg B NL

Examples
(a) MCINS *.
(b) MCINS U, INSERT HERE

Restrictions
ARG A, if it exists, must consist of the letter 'P' or the letter 'U'. Redundant spaces are allowed. ARG B must be a structure representation of form:

delimiter name    delimiter name

System action
A new local insert definition is added to the current environment. The delimiter structure of the new insert is represented by ARG B and the option is defined as "protected" unless ARG A exists and consists of the letter 'U'. In this latter case it is defined as "unprotected".

Notes
(a) Unprotected inserts are only needed for sophisticated applications of ML/I and users with simple applications can safely omit arg A.
5.2.3 MCSKIP

**Purpose**
Definition of a local skip.

**General form**
MCSKIP [ arg A, ? ] arg B NL

**Examples**
(a) MCSKIP MT, ( )
defines "(" and ")" as literal brackets.
(b) MCSKIP N WITH. WITH B WITH. ;
deletes comments that commence with N.B. and end with a semicolon.
(c) MCSKIP DT, '
(d) MCSKIP NONL NL
(e) MCSKIP T, NOPUNCT N1 OPT, N1 OR. N1 OR END ALL
causes all commas and periods between NOPUNCT and END to be deleted.
(f) MCSKIP STATIC
deletes all occurrences of STATIC. Note that the delimiter structure of a skip can specify any number of delimiters, although usually there will be one, as in this example, or two.

**Restrictions**
ARG A, if it exists, must have form:

(M) 
[D] *? 
(T)

Redundant spaces are allowed. ARG B must be a structure representation.

**System action**
A new local skip definition is added to the current environment. The delimiter structure of the new skip is represented by ARG B, and the matched option, the text option and the delimiter option are set if ARG A contains the letter M, T, or D, respectively. If arg A is omitted none of the options is set.

**Notes.**
(a) The letters in ARG A may be in any order.
(b) If arg A is omitted and arg B contains a comma, then this comma should be enclosed in literal brackets to prevent it being taken as a delimiter of MCSKIP.
5.2.4 MCDEF

Purpose
Definition of a local macro.

General form
MCDEF [ arg A  VARS ? ] arg B (AS ) arg C NL

Examples
(a) MCDEF ARRSIZE AS 6
(b) MCDEF ESUB NL
   AS < CMA
   ADD %A1.
   CMA
   >
   is a definition of the ESUB macro used in examples.
(c) MCDEF 6 VARS CALCULATE AS ... 
(d) MCDEF (OPT + OR - OR * ALL) AS < %D1. %A1. %A2 > This macro converts fully parenthesized algebraic notation to Polish Prefix notation. Thus, for example, it would convert ((PI*26)-LENGTH) to -* PI 26 LENGTH.
(e) MCDEF PARENS AS WITH ( ) WITH NL defines the PARENS macro used in Section 5.1.2.
(f) MCDEF NOTE ; SSAS [< %WAI. ] > is the definition of the straight-scan macro NOTE used as an example in Section 2.10. "SSAS" stands for "straight-scan AS".
(g) MCDEF CALL NL NØ AS ... defines a CALL macro with newline as an exclusive delimiter.

Restrictions
ARG A, if it exists, must be a macro expression and ARG B must be a structure representation.

Order of evaluation
arg A, arg C, arg B.

System action
A new local macro definition is added to the current environment. The delimiter structure of this new macro is represented by ARG B, the replacement text is specified by ARG C and the capacity (i.e. the number of temporary variables) is the greater of the result of ARG A and three. The capacity is three if ARG A is omitted. The new macro is set up as a normal-scan macro if
MCDEF is called with delimiter AS and as a straight-scan macro if the delimiter SSAS is used.

Notes

(a) The replacement text is normally enclosed in literal brackets to delay evaluation until macro call time and to ensure that any newlines within the replacement text are not taken as the closing delimiter of MCDEF.

(b) If it is desired that the replacement text be treated as a literal when the macro is called as well as when it is defined, then it is necessary to enclose the replacement text in double literal brackets (see example in Section 9.3.1).
5.2.5 **MCNOWARN, MCNOINS, MCNOSKIP, and MCNODEF**

**Purpose**
Deletion of local constituents of the current environment.

**General form**
(a) MCNOWARN
(b) MCNOINS
(c) MCNOSKIP
(d) MCNODEF

**System actions**
These macros respectively delete all local warning markers, all local insert definitions, all local skip definitions and all local macro definitions from the current environment. In addition, MCNOWARN causes the current environment to be placed in free mode unless there are any global warning markers.

**Notes**
(a) Note that these macros do not have newline as a closing delimiter.
(b) In current implementations no storage is released if a constituent of the environment is deleted by one of these macros.
(c) See the example in Section 9.3.4 for a method of deleting individual constructions from the environment.
(d) If MCNOWARN is to be meaningful it must be preceded by a warning marker.
(c) MCNODEF does not cause the operation macros to be deleted from the environment since these latter are global.
5.2.6 MCWARN, MCINS, MCSKIP and MCDEF

**Purpose**
Global equivalents of MCWARN, MCINS, MCSKIP and MCDEF.

**General form**
Similar to those of the corresponding local macros.

**Examples**
(a) MCWARN MACRO
(b) MCINS /.
(c) MCSKIP DT, TEXT::
(d) MCDEF %AL WITH (,)AS <...>

**Restrictions**
The restrictions on the forms of arguments are the same as for the corresponding local macros.

**System actions**
As for the corresponding local macros except that the newly-defined constituents are global rather than local.

**Notes**
(a) If a global NEC macro is called in the source text, the effect is the same as if the corresponding local macro had been called (except for certain differences if the name is multiply-defined). Global constructions are not, however, deleted by the macros MCNOWARN etc. described in Section 5.2.5. For reasons of efficiency the user is recommended to use local macros where possible.
(b) If a call of MCWARN occurs, all subsequent text processing will be in warning mode, since it is impossible to delete a global warning marker.
5.2.7 MCALTER

Purpose
Alteration of the secondary delimiters operation macros or of the keywords used structure representations.

General form
MCALTER \textit{arg A} TO \textit{arg B} NL

Examples
(a) MCALTER
\texttt{TO ;}
MCALTER \texttt{AS TO : ;}
After these two calls of MCALTER, Example (a) of Section 5.2.4 would be written:
\begin{verbatim}
MCDEF ARRSIZE : 6;
\end{verbatim}
(b) MCALTER WITH TO +
MCDEF JOIN + (WITH) AS •••
MCALTER + TO WITH
Here \texttt{WITH} is changed to + and then back to \texttt{WITH} again in order to delimit macro "JOIN(" with delimiter \texttt{WITH}.
(c) MCALTER N TO 9
(d) MCALTER SPACE TO BLANK

Restrictions
ARG A and ARG B must be single atoms. ARG A must be either a secondary delimiter of one or more operation macros or one of the keywords used in structure representations. ARG B must not be longer than the system name of any delimiter or keyword matched by ARG A. If ARG A is the node flag (i.e. the letter 'N' or whatever has replaced it) then ARG B must be a letter or a digit.

Order of evaluation \textit{arg B, arg A}.

System action
ARG B is substituted in place of ARG A wherever ARG A occurs as a secondary delimiter of an operation macro or as a keyword.
Notes

(a) MCALTER cannot be used to change the names of operation macros.

(b) It is very dangerous to change a keyword or delimiter to become the same as another keyword, for instance:

   MCALTER UNLESS TO IF

The effect of an alteration such as the above on subsequent processing is undefined, since it depends upon the order in which delimiters are scanned.

(c) In the unlikely event of a call of MCALTER specifying several replacements some of which are valid, and some of which are invalid because of the length of ARG B, then the number of valid replacements that are performed before the call is aborted is undefined.

(d) In the MCGO macro (and in any other macro where the action taken depends upon the form of the delimiters), the delimiters are examined immediately the macro is called and no call of MCALTER within an argument can affect the action of the containing macro.

(e) Since the operation macros are global, the effect of MCALTER is also global.

(f) It has been assumed in examples throughout this manual (apart from this Section) that no calls of MCALTER have occurred.

(g) Since MCALTER has a global effect, it is not recommended to use it locally to an piece of replacement text. If it is used locally, MCALTER must be called again before leaving the replacement text in order to cancel the changes that have been made.

(h) A layout character can be MCALTERed to be the same as the character it represents, e.g.

   MCALTER NL TO <
   >

   This will effectively delete the layout keyword, e.g. after the above MCALTER, newline would stand for itself within structure representations - it would not act as a separator.
5.3 System functions

The operation macros which return values are listed in this Section. Note that these macros do not have a newline as the closing delimiter.

5.3.1 MCLENG

Purpose
Function to find the length of a character string.

General form
MCLENG ( arg A )
The left parenthesis is part of the macro name. It may optionally be preceded by spaces.

Examples
(a) MCLENG (%A1.)
(b) MCLENG (%A1.%D3.PIG)

System action
The value of this function is the number of characters in ARG A. This number is represented as a character string in the way described in Section 2.6.7 (e).
5.3.2 MCSUB

**Purpose**
Function to access a substring

**General Form**
MCSUB (arg A, arg B, arg C)
The left parenthesis is part of the macro name. It may optionally be preceded by spaces.

**Examples**
(a) MCSUB (ABC/XYZ, 3, 6)
   This function has value C/XY.
(b) MCSUB (ARGUMENT, -2, 0)
   This function has value ENT, since non-positive results of ARG B and ARG C specify offsets from the end of ARG A.
(c) MCSUB (%D2., 1, 1)
   The value of this function is the first character of the inserted delimiter.
(d) MCSUB (%A3. Y%D3., 1, P3 - T6 + 7)

**Order of evaluation** arg A, arg B, arg C. However, arg C is not evaluated if VB (see below) is greater than L (see below) or is less than one.

**System action**
Let L be the number of characters in ARG A, let RB be the result of ARG B, and let VB be derived from these values by the following rule:

\[
\begin{align*}
\text{VB} &= \begin{cases} 
\text{RB} & \text{if } \text{RB} > 0 \\
(L + \text{RB}) & \text{otherwise}
\end{cases} 
\end{align*}
\]

Let VC be derived from the result of ARG C by a similar rule. The value of a call of MCSUB depends upon whether VB and VC describe a valid substring of ARG A. This occurs if:

\[1 \leq \text{VB} \leq \text{VC} \leq L\]

If this relation does not hold the value of MCSUB is null. If the relation holds the value of MCSUB is the substring of ARG A from character position VB up to and including character position.
VC, the first character of ARG A being taken as character position one.

Notes

(a) In the case where the relation holds, the value of MCSUB consists of VC - VB + 1 characters.

(b) The value of MCSUB is not itself evaluated. Thus the value of Example (b) would be ENT even if ENT was a macro.
5.4 Further operation macros

The remaining operation macros, i.e. those not falling into the previous categories, are described below.

5.4.1 MCSET

Purpose
Macro-time assignment statement.

General form
MCSET arg A == arg B NL

Examples
(a) MCSET PIO == 3
(b) MCSET T6 == -4
(c) MCSET TT3 == TP4 - 109 + 25/P1
(d) MCSET T%Al. == %Al. + 17

where the value of the inserted argument is a positive integer.

Restrictions
ARG A must be the name of a macro variable in the current environment. (ARG A may contain redundant spaces at the beginning or the end.) ARG B must be a macro expression.

System action
The result of ARG B is assigned to the macro variable designated by ARG A.
5.4.2 MCNOTE

Purpose
Generation of user's own error and debugging messages.

General form
MCNOTE arg A NL

Examples
(a) MCNOTE %A3. IS ILLEGAL ARGUMENT.
(b) MCNOTE OCCURENCE NUMBER %P1. OF <CONT>

System action
ARG A is printed on the debugging file (see Chapter 8) as if it were a system message. Three newlines are inserted in front of it and it is followed by a printout of the context of the call of MCNOTE.

Notes
(a) If example (b) occurred in line 3 of a macro CONT, then the printout might be:

```plaintext
OCCURRENCE NUMBER 33 OF CONT
DETECTED IN
LINE 3 of MACRO CONT WITH NO ARGUMENTS
CALLED FROM
LINE 267 OF SOURCE TEXT
```

(b) Notes (d) and (f) of Section 6.2 do not apply to the printing of ARG A.
5.4.3 MCGO

Purpose
Macro-time GO TO statement or conditional GO TO statement.

General forms
(a) MCGO arg A NL
(b) MCGO arg A (IF ) arg A (= ) arg C NL
                  (UNLESS)
                  (BC)
                  (EN)
                  (GE)
                  (GR)

The meanings of the respective mnemonic second delimiters are:
Belongs to Class, Equals Numerically, Greater than or
Equals and Greater than.

Examples
(a) MCGO L1
(b) MCGO LT1
(c) MCGO L6 IF%D1. = +
(d) MCGO LO UNLESS P3 - T5 GE - 6
(e) MCGO LT3 - P7 + 4 UNLESS %A6. BC N

This tests whether argument six is a number
(Belongs to the Class of Numbers).

Restrictions
ARG A Must consist of the letter "L" (optionally preceded by
redundant spaces) followed by a macro expression. The result of
this macro expression must never be negative and, furthermore, it
must not be zero if MCGO is called from the source text. If the
second delimiter is BC then ARG C, which is the name of a class,
must consist of one of the following letters:

L (for identifier)
L (for letter)
N (for number)
R (for real number)

together with any desired number of spaces. If the second
delimiter is EN, GE or GR then ARG B and ARG C must both be
macro expressions.
Order of evaluation arg B, arg C, arg A. In Form (b), arg A is evaluated only if the condition holds.

System action for form (b) ARG B and ARG C are compared to yield a true or false value. If the second delimiter is EN, EG or GR, then numerical comparison is performed; otherwise character comparison is performed. The method of comparison depends on the second delimiter in the following way:

(a) =. A true value results only if ARG B and ARG C are identical strings of characters.

(b) BC. If ARG C is the letter I, then a true value results only if ARG B is of form:

(letter) *
(digit )

If ARG C is the letter L, then a true value results only if ARG B is of form:

[ letter * ]

If ARG C is the letter N, then a true value results only if ARG B is of form:

(+)*? [ digit* ]

(-)

(The letter R has been added, see in Appendix).

(c) EN, GE, GR. In these cases a true value results only if the result of ARG B is, respectively, numerically equal to, greater than or equal, or greater than the result of ARG C.

If the comparison yields a false value and the second delimiter is IF or if the comparison yields a true value and the second delimiter is UNLESS, then no further action takes place. Otherwise the system action for FORM (a) is now performed.

System action for form (a) Let N be the result of the macro expression in ARG A. If N is positive, then the point of scan is changed to the point associated with macro label N. (See next Section for a fuller
description.) If \( N \) is zero, the processing of the current piece of text is abandoned and evaluation proceeds as if the end of the current piece of text had been reached. Thus when \( N \) is zero a MCGO serves a similar function to the RETURN statement found in many high-level languages. This 'RETURN' facility may be used within inserted text or replacement text but not within the source text.

**Notes**

(a) Note that leading and trailing spaces are removed before \( \text{arg B} \) and \( \text{arg C} \) are evaluated. If it is required that these spaces take part in the comparison, they should be enclosed in literal brackets.

(b) If it is desired to achieve the effect of a backward GO TO in the source text then the required loop must be defined as the replacement text of a macro call. See Section 9.4.1 for an example.

(c) Sections 9.3.3 and 9.3.5 contain examples of the use of MCGO.

(d) The user should be very careful to differentiate between the two relational operators "=" and \( \text{EN} \). Note that the relation "\( \text{PI EN P2} \)" is true if the first two permanent variables have the same value whereas "\( \text{PI = P2} \)" is, of course, never true. Note that "\( \%\text{PI} = \%\text{P2} \)" is equivalent to "\( \text{PI EN P2} \)".

**5.4.3.1* Exact description of GO TO**

The following is a more exact description of the action of ML/I in performing a GO TO when \( N \) is positive.

If label \( N \), which is called the designated label, is present in the current environment then the action of ML/I is simply to change the point of scan to the point associated with the designated label. Otherwise a forward search for the designated label is performed, starting at the current point of scan. If a macro call or skip is encountered during this search, the search is suspended until the end of the macro call or skip is found. Each time an insert is encountered outside a call or skip,
the argument is evaluated and the search ends when an insert which "places" label N is found (or, in the error case, at the end of the current piece of text). No value text is generated during a search and no macro calls are performed (except conceivably during the evaluation of the argument of an insert). At the end of the search the action of ML/I is concluded by setting the point of scan as the point immediately after the designated label.

The only way the current environment can be changed by a call of MCGO is that any labels encountered in the forward search (including the designated one) are added to the current environment provided that the rules of Section 2.6.7 (f) are satisfied.

If an error is detected during a forward search then the appropriate error message is printed in the normal way.
5.4.4 MCPVAR

Purpose
Allocation of extra permanent variables.

General form
MCPVAR arg A NL

Example
(a) MCPVAR 100

Restrictions
ARG A must be a micro expression.

System action
Let N be the result of ARG A. If N is greater than the current number of permanent variables then the number of permanent variables is increased to N; otherwise no action is taken. The values of the new permanent variables are undefined but the values of the previously allocated ones remain unchanged.
Chapter 6 Data specification macros (declarations).

6.1 Introduction

In EPS all variables must be declared and we employ variable scope rules which are nearly the same as those of ALGOL 60 (see chapter 7). Declarations are made in EPS by calling "Data Specification" macros. These macros are also used for definition of the formal parameters of procedures (see chapter 7). The data specification macros enter the name and type of the data declared into the macro time symbol table (described in appendix section 1) and generate Assembler 360 code for storage allocation (DS or DC). Storage for simple variables non-subscripted is allocated in the same area as the program segment where it is declared, whereas storage for arrays is allocated dynamically in a separate data stack, and is freed upon exit of the block in which the arrays are declared (as described in chapter 7). Non subscripted are upon declaration assigned an initial value which represent "undefined". This enables, later in the program, to check whether a variable employed in an expression has already been assigned a value. Arrays are assigned initial values only when we run in "debugging mode" (described in section 8.6).

The syntax of declarations (calls of data specification macros) in EPS is similar to that of ALGOL 60, e.g.

```
REAL A, B;
```

The existing types of the variables are described in detail in the next sections and are summarized in Table 6.1.

6.2 Identifiers

An identifier is a string of up to six characters, the first of which is a letter and each of the remaining is either a letter or a decimal digit. Blanks or other characters which are not letters or digits are not permitted as an identifier constituent.

6.3 The INT macro

**Purpose**

Declaration of fixed point binary variables.
General form

(INT) [Argument,*?] Argument;
(INTGER)

The Arguments are variable names.

Examples

(a) INT A;
(b) INT Q, R, PL1;
(c) INTEGER S1, EPS;
(d) INTEGER TEPSLY;

Restrictions

A variable name is an identifier as defined in 6.2 and may not collide with any other identifier in the same external procedure.

System action

1. For each variable the code

   variable name DC F'2147483647'

   is generated. The value 2147483647 represents "undefined integer value".

2. For each variable the following information is entered the macro-time symbol table:

   name: variable name
   type: 1
   dimension: 0

6.4 The SHORT INT macro

Purpose

Declaration of half word fixed-point binary variables

General form

SHORT (INT) [Argument,*?] Argument;
(INTGER)

The Arguments are variable names.

Examples

(a) SHORT INT AR;
(b) SHORT INT B, P, XLY;
(c) SHORT INTEGER ASW, ALGOL;

Restrictions

A variable name is an identifier as defined in section 6.2

System action

1. For each variable the code:

   variable name DC H'32767'

   is generated. 32767 represent "undefined short integer value".

2. For each variable the following information is entered
the macro-time symbol table:
name: variable name
type: 2
dimension: 0

6.5 The REAL macro

Purpose
Declaration of short floating-point variables.

General form
REAL [Argument, *?] Argument;
The Arguments are variable names.

Examples
(a) REAL A;
(b) REAL A, B, C1Y, ZPR1Y2;

Restrictions
A variable name is an identifier as defined in section 6.2.

System action
1. For each variable the code:
variable name DC E'2.5E76'
is generated. 2.5E76 represent "undefined real value".
2. For each variable the following information is entered
the macro-time symbol table:
name: variable name
type: 3
dimension: 0

6.6 The LONG REAL macro

Purpose
Declaration of long floating-point variables.

General form
LONG REAL [Argument, *?] Argument;
The Arguments are variable names.

Examples
(a) LONG REAL PØR2;
(b) LONG REAL B, L, R2;

Restrictions
A variable name is an identifier as defined in section 6.2.

System action
1. For each variable the code:
variable name DC D'2.5E76'
is generated. 2.5E76 represent "undefined real value".
2. For each variable the following information is entered
the macro-time symbol table:
name: variable name
type: 4
dimension: 0

6.7 The INT ARRAY macro

Purpose Declaration of one dimensional integer arrays.

General form (INT) ARRAY [Argument A (Argument B), *?]
(INTEGER)

Argument A (Argument B);
1. Arguments A are array names.
2. Arguments B specify the number of elements in each array.

Examples (a) INT ARRAY B(8);
1. Argument A is array B.
2. Argument B is the number of elements 8.

(b) INT ARRAY Q(NP1);
(c) INTEGER ARRAY A(7), C(M), F(MN3);

Restrictions 1. Arguments A are identifiers as defined in 6.2
2. Arguments B must be unsigned, non zero, positive integer
numbers or integer variables which have been given posi­
tive non-zero values.

System action *1. For each array declared Assembly 360 code is generated
(see example below)
a. Allocation of an area of two words whose name (label)
is the array name.
b. Storing into the first word the address of the first
free byte in the data stack. This address is the value
of register 11. Appropriate assignment is performed
if needed.
c. Updating register 11 to the first free byte in the
data stack after the area allocated for the array.
d. Storing in the second word the length in bytes of the
area allocated for the array.

2. Enter the macro-time symbol table the information :
name: array name
type: 1
dimension: 1
Example  This example illustrates what happens during an array declaration.

```
INTEGER ARRAY A(8);
```

The following code will be generated according to the explanation on *1.

```
N 11, = X'FFFFFFFC' ALIGNMENT TO BOUNDARY OF A FULL WORD
A 11, = X'4'
BC 15, #C23
A DS 2F
#C23 NOP
ST 11,A ADDRESS OF FIRST ELEMENT
L 9, = F'8'
NOP
AR 9,9
AR 9,9
ST 9,A+4 LENGTH OF ARRAY (BYTES)
AR 11,9 UPDATE STACK POINTER
```

6.8 The SHORT INT ARRAY macro

**Purpose**  Declaration of one dimensional short integer arrays.
6.9 The REAL ARRAY macro

**Purpose**
Declaration of one dimensional real arrays.

**General form**
REAL ARRAY [Argument A (Argument B), *?]  

**Examples**
(a) REAL ARRAY A(S), B(I), C1Y(J);  
(b) REAL ARRAY BIT1Y3(S8S);

**Restrictions**
1. Arguments A are identifiers as defined in 6.2.  
2. Arguments B must be unsigned non zero integer numbers or integer variables which have been already given a positive non zero value.

**System action** 
1. The same action as in section 6.8.  
2. For each array the following information enters the macro-time symbol table.  
   name: array name  
   type: 2  
   dimension: 1
macro-time symbol table:
name: array name
  type: 3
  dimension: 1

6.10 The LONG REAL ARRAY macro

**Purpose**
Declaration of one dimensional long real arrays.

**General form**
LONG REAL ARRAY [Argument A (Argument B),*?]

**Examples**
(a) LONG REAL ARRAY A(1), B(X);
(b) LONG REAL ARRAY C1Y(G1S);

**Restrictions**
1. Arguments A are identifiers as defined in 6.2.
2. Arguments B must be unsigned non zero integer numbers or integer variables which have been already given a positive non zero value.

**System action**
1. The same action as in 6.8.
2. For each array the following information enters the macro-time symbol table.
   name: array name
   type: 4
   dimension: 1

6.11 The CHAR macro

**Purpose**
Declaration of character variables.

**General form**
(CHAR) [Argument A (Argument B),*?]

**Examples**
(a) CHAR A(1);
(b) CHAR B(2), C(256), B1Y256(C);
(c) CHARACTER B(4);
(d) CHARACTER B(S), C(A);

Restrictions

1. Arguments A are variable names which are identifiers as defined in 6.2.
2. Arguments B are unsigned non zero positive integer numbers or integer variables, which have been already given a positive non zero value.

System action

1. The same as in 6.8. Obviously declaration of a character variable of length n is identical to declaration of a one dimensional array of n, one byte length, elements.
2. For each variable declared, the information entered the macro-time symbol table is:
   name: variable name
   type: 5
   dimension: 0
Table 6.1 Variable types in EPS

<table>
<thead>
<tr>
<th>Declaration starting with the key word</th>
<th>Description of the type</th>
<th>Object code generated</th>
<th>Entry into the macro time symbol table</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT or INTEGER</td>
<td>Integer stored in a word</td>
<td>DC F'2147489647'</td>
<td>TYPE 1 DIMENSION 0</td>
</tr>
<tr>
<td>SHORT INTEGER or SHORT INT</td>
<td>Integer stored in a half word</td>
<td>DC H'32767'</td>
<td>TYPE 2 DIMENSION 0</td>
</tr>
<tr>
<td>REAL</td>
<td>Floating point number stored in a word</td>
<td>DC E'2.5E76'</td>
<td>TYPE 3 DIMENSION 0</td>
</tr>
<tr>
<td>LONG REAL</td>
<td>Floating point number stored in a double word</td>
<td>DC D'2.5E76'</td>
<td>TYPE 4 DIMENSION 0</td>
</tr>
<tr>
<td>CHAR</td>
<td>Characters stored in n consecutive bytes</td>
<td>see section 6.11</td>
<td>TYPE 5 DIMENSION 0</td>
</tr>
<tr>
<td>INTEGER ARRAY or INT ARRAY</td>
<td>Integers stored in n consecutive words</td>
<td>see section 6.7</td>
<td>TYPE 1 DIMENSION 1</td>
</tr>
<tr>
<td>SHORT INT ARRAY</td>
<td>Short integers stored in n consecutive half words</td>
<td>see section 6.8</td>
<td>TYPE 2 DIMENSION 1</td>
</tr>
<tr>
<td>REAL ARRAY</td>
<td>Floating point numbers in n consecutive words</td>
<td>see section 6.9</td>
<td>TYPE 3 DIMENSION 1</td>
</tr>
<tr>
<td>LONG REAL ARRAY</td>
<td>Floating point numbers in n consecutive double words</td>
<td>see section 6.10</td>
<td>TYPE 4 DIMENSION 1</td>
</tr>
</tbody>
</table>
Chapter 7 Program structuring macros and activation of the operating system

7.1 Introduction

The macros BEGIN, PROC, PROCEX and MAIN PROCEX are employed for defining blocks, procedures and external procedures. The two latter are procedures that may be compiled separately, each into an OS/360 CSECT. The CSECT's may be later linked together by a standard OS/360 linker into a program. The REPEAT macro is used for defining loop compounds whose execution is repeated until certain conditions are met. The EXIT macro is used for an abnormal exit from procedures, blocks, and loop compounds.

Conditional compounds are bracketed by the IF THEN ELSE FI macro. A CASE macro provides a multiple choice compound. In this case, the NOP (no operation) macro may be used for specifying dummy cases. Comments may be written between the symbol \# and the end of the line (the COMMENT macro).

7.2 Program organization and scope of names

An EPS program consists of a main program which is a main external procedure (MAIN PROCEX), followed, optionally, by external procedures.

Each external procedure may contain one or more internal procedures (PROC) which are positioned after the RETURN of the PROCEX and before its END. Each procedure (internal or external) may contain BEGIN-END blocks. It is prohibited to nest external procedures. Internal procedures may however be nested in the way described in section 7.5. BEGIN blocks may be nested in the way it is done in ALGOL 60 and PL/I. There is only one RETURN in a procedure, and if one wished to exit the procedure from any other point the EXIT macro may be used. These features are illustrated in the following example:
This program is composed of two external procedures, the MAIN PROCEX named GO and the PROCEX named OUT.

The MAIN PROCEX GO contains the blocks A and B, where B is nested in A. In addition it contains the internal procedures MAX, which contains the block X, and W, both residing between the RETURN and the END of the MAIN PROCEX GO. The macro EXIT is used twice, in the first time for exiting the block X and in the second time for exiting the MAIN PROCEX GO.
Non subscripted variables declared in an external procedure are known from the declaration statement in which they were declared throughout the external procedure. However, subscripted variables (arrays) are only known from the declaration statement until the end of the block or procedure in which they were declared. This dynamic storage allocation for subscripted variables does not relax, however, the restrictions on the naming convention discussed in the following paragraph.

Since any external procedure is translated into OS/360 CSECT we must require that every entity (a variable, a label, a block, or a procedure) within an external procedure has a unique name (identifier). This is a restriction as compared to ALGOL 60. Different entities in different external procedures may, however, have the same name.

Storage for arrays is allocated in a data stack. Upon exit from a block or a procedure, the storage allocated for the arrays declared in it is released. Since procedure parameters are transformed to relative addresses (which relate to the address allocated in register 1 given by the CALL macro), a parameter name must be unique within its procedure but may be used for any purpose outside the procedure.

7.3 The PROCEX macro

<table>
<thead>
<tr>
<th>Purpose</th>
<th>To define an external procedure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>General form</td>
<td>PROCEX</td>
</tr>
<tr>
<td></td>
<td>Argument C</td>
</tr>
<tr>
<td></td>
<td>[Argument D*?];</td>
</tr>
<tr>
<td>1.</td>
<td>Argument A is the procedure name.</td>
</tr>
<tr>
<td>2.</td>
<td>Argument B is the formal parameter list.</td>
</tr>
<tr>
<td></td>
<td>e.g. REAL A; INT A, B; In case of arrays an asterisk(*) must be placed instead of the array length.</td>
</tr>
<tr>
<td></td>
<td>(e.g. REAL ARRAY A(*);)</td>
</tr>
<tr>
<td>3.</td>
<td>Argument C is the external procedure body.</td>
</tr>
<tr>
<td>4.</td>
<td>Arguments D, if exist are procedures which reside inside this external procedure.</td>
</tr>
</tbody>
</table>
Examples

(a) PROCEX A;

(b) PROCEX A (REAL B,C);

Restrictions

A procedure name and its parameters are identifiers as defined in 6.2, and must not collide with any other identifiers in the same external procedure.

System action

1. Produces a new CSECT named as the PROCEX.
2. Produces code for starting an assembler program such as base register loading, save area allocation and register saving.
3. Produces code that upon exit from the procedure will free the storage allocated in the data stack declarations at the head of the procedure. For details see chapter 6.
4. Produces code for ending an assembly language CSECT.
5. Enters into the macro-time symbol table the information:

7.4 The MAIN PROCEX macro

Purpose

To define an external procedure which is used as the main program.

General form

MAIN PROCEX Argument A;
Argument B
[RETURN;?]
[Argument C*?]
END [Argument A?];

1. Argument A is the procedure name.
2. Argument B is the procedure body.
3. Arguments C, if exist, are internal procedures which reside inside this external procedure.

**Examples**

```
MAIN PROCEX   A;
    =
RETURN;
PROC B(INT C;);
    =
RETURN;
END B;
END A;
```

**Restrictions**

Argument A is an identifier as defined in section 6.2 and must not collide with any other identifiers in the same external procedure.

**System action**

1. Produces a new CSECT named as the procex.
2. Produces code for starting an assembler program such as base register loading, save area allocation and register saving.
3. Produces code for ending an assembler language CSECT.
4. Produces an entry point for the linkage editor.
5. Produces code that upon exit from the procedure will free storage allocated in the data stack by declarations in the head of the procedure.
6. Enters into the macro-time symbol table the information:
   name:  procedure name
type:  6
dimension:  0

How to use the MAIN PROCEX

A program must contain one and only one MAIN PROCEX. The execution of a program starts with the first statement on the MAIN PROCEX. The body of the MAIN PROCEX will
typically contain calls to the remaining PROCESes of the program. The MAIN PROCEX itself may not be called from other PROCESes.

The MAIN PROCEX has no parameters. Therefore all the variables which are used in the MAIN PROCEX body have to be declared in it.

How to convert a MAIN PROCEX into an ordinary PROCEX

A conversion of a MAIN PROCEX into an ordinary PROCEX is needed when it is desired to combine a number of programs, each having its own MAIN PROCEX, into a single program. The MAIN PROCEX of each of these programs must be changed into an ordinary PROCEX, and a new single MAIN PROCEX is written for the combined program.

When designing a MAIN PROCEX, it is recommended to consider the case where the current program is a subprogram in a larger program. Those variables that will be used for interfacing the current program with the larger program, will appear as parameters of the PROCEX into which we convert our current MAIN PROCEX. These variables should appear in the current MAIN PROCEX in variable declaration statements such that they may readily be transformed into the parameters of the eventual PROCEX. In the current MAIN PROCEX it may furthermore be necessary to write code for assigning appropriate initial values to these variables. If this practice is followed, the eventual changing of the MAIN PROCEX into an ordinary PROCEX will involve:

(a) dropping of the MAIN before PROCEX;
(b) adding parameters behind the PROCEX name
(c) removing the variable declarations and initial assignment statements written for variables which are now used as parameters.

The following example illustrates how to convert a MAIN PROCEX into an ordinary PROCEX:
current MAIN procex

MAIN PROCEX A;
REAL A,B,C;
INT I;
REAL ARRAY D(10);
SET A = 50;
SET B = A+1;
SET C = 50+B*A;
SET I = 0;
REPEAT FOR I=1 TO 10;
SET D(I) = 0;
END;
SET A = B+C;
RETURN;
END A;

converted to ordinary PROCEX

PROCEX A(REAL A,B,C; INT I;
REAL ARRAY D(*));
SET A = B+C
-
-
RETURN;
END A;

7.5 The PROC macro

Purpose To define an (internal) procedure.

General form

PROC Argument A [(Argument B)?];
Argument C
[RETURN;?] [Argument D*?]
END [Argument A?];

1. Argument A is the procedure name.
2. Argument B is the formal parameter list which is written in the same way as variable declarations.
3. Argument C is the procedure body.
4. Arguments D, if exist, are internal procedures defined in this internal procedure.
Examples

(a) PROC K(INTEGER I,J;);
    -
END K;

(b) PROC MIN(REAL A,B; INTEGER C;);
    -
RETURN;
PROC A(REAL D);
    -
RETURN;
END A;
PROC Q;
    -
END Q;
END MIN;

Restrictions
A procedure name and its parameters are identifiers as defined in section 6.2. A procedure name must not collide with any other identifier in the same external procedure.

System action
1. Produces code for register saving
2. Produces code for returning to the calling program.
3. Produces code that upon exit from the procedure will be free from the data stack storage, allocated by declarations in the procedure.
4. Enters into the macro-time symbol table the information:
   name: procedure name
   type: 6
   dimension: number of parameters.
7.6 The BEGIN macro

**Purpose**
To define a BEGIN-END block.

**General form**
BEGIN [Argument A?];
[Argument B]
END [Argument A?];

Argument A is the block name.
Argument B is the block body.

**Examples**
(a) BEGIN A1;
   =
   BEGIN B;
   =
   END B;
   =
   END A1;

(b) BEGIN;
   =
   END;

**Restrictions**
Argument A is an identifier as defined in section 6.2 and must not collide with any other identifier in the same external procedure.

**System action**
1. Produces code for freeing storage allocation from the data stack upon exit from the block.

2. Enters into the macro-time symbol table the information:
   - name: *block name*
   - type: 6
   - dimension: 0
7.7 The CALL macro

Purpose Calls a procedure.

General form
CALL Argument A [(Argument B)?];
Argument A is the procedure name.
Argument B is the list of actual parameters of the called procedure.

Examples
(a) CALL MIRA;
(b) CALL A(B);
(c) CALL SIN(A,C,D,Q(3));

Restrictions
The number and type of the actual parameters must match the number and type of the corresponding formal parameters which have been defined in the corresponding PROCEX or PROC macro. The parameters must be either identifiers or constants but not expressions.

System action
Code is generated such that a contiguous storage field is allocated for an "activation record" and such that the values of the actual parameters are stored in this record. Register 1 is loaded with address of the first byte of this field. Furthermore, the called procedure address is stored in register 15 and a BALR instruction generated such that the return address is stored in register 14.

The same CALL macro is used for calling both internal and external procedures.

7.8 The REPEAT macro and the FOR WHILE and EXIT constructs

Purpose Coding loops. The loop compound is bracketed by REPEAT-END and the number of repetitions is controlled by FOR, WHILE and EXIT constructs.

General form
REPEAT [Argument A?] [For Argument B = Argument C1 [BY Argument C2?] TO Argument C3?] [WHILE Argument D?] ;
Argument E

END [Argument A?];

1. Argument A is an optional name for the REPEAT compound, and is an identifier as defined in 6.2.

2. Argument B is the control variable. It is an integer variable which is implicitly declared by this FOR clause, and whose scope is confined to the current REPEAT-END compound. This variable should thus not be declared.

3. Arguments C1, C2, C3 are either integer constants or integer variables.

4. Argument D is a logical expression.

5. Argument E is the REPEAT body.

Examples

(a) REPEAT DIR FOR I = 2 TO 10;
    =
    END DIR;

(b) REPEAT FOR I = 1 BY 2 TO 5;
    =
    REPEAT FOR J = 1 TO P;
    =
    END;
    =
    END;

(c) REPEAT A WHILE A < B;
    =
    END A;

(c) REPEAT SUM FOR I = 1 TO N WHILE A > B;
    =
    END SUM;

(e) REPEAT Q;
    =
    IF S = R THEN EXIT Q; FI;
    =
    END Q;

System action

The meaning of the FOR and WHILE constructions are similar to those in ALGOL 60 and PL/I. If the BY is omitted in the FOR clause a step of 1 is assumed. If both FOR and WHILE clauses
are specified the execution is repeated as long as both the FOR and WHILE conditions are satisfied. The FOR and WHILE conditions are tested before any repetition of the execution of the loop compound. If it is required to exit the loop compound at any other point, the EXIT construct is employed (example e).

It is not required to name the loop compound if there is no EXIT. It is however recommended to name all large loops, both for improving the documentation and for enabling the system to recover in case of missing END's in the program.

Variables for controlling the looping, such as I and J in example b, need not be declared. These names may collide with same names outside the compound.

7.9 The CASE macro

**Purpose**

Defining multiple choice compound

**General form**

```
CASE Argument A OF [Argument E?];
[ (Argument B) Argument C*] 
[OUT; Argument D?] 
END [Argument E?];
```

1. Argument A is an integer variable whose value is a positive integer that specifies which piece of program ("case") will be executed.

2. Argument B is a comment by which the programmer can describe the various cases.

3. Arguments C and D are the pieces of program to be executed in the different cases.

4. Argument E is the name of the whole case compound.

**Examples**

(a) CASE N OF;
   (1) -
   (2) -
   (3) -
   END;
   CASE X OF A;
   (FIRST) -
   (SECOND) NOP;
   (ABNORMAL) -
   OUT; SET ERROR17 = 1
   END A;

Technion - Computer Science Department - Technical Report CS0033 - 1974
Restrictions Argument A must be an integer variable.

System action The cases are numbered 1, 2, ..., M, M being the number of cases. If the value of argument A is n then the n-th piece of text is executed. If n is out of range, i.e. n < 1 or n > M, Argument D, if exists, is executed.

7.10 The EXIT macro

Purpose Transfer of control to an END of a procedure, block, REPEAT or CASE compound whose name is specified as argument of the EXIT macro.

General form EXIT Argument A ;

Argument A is the EXIT name, i.e. the name of the procedure, block or compound that we want to exit.

Examples (a) BEGIN A;
     =
     EXIT A;
     END A;
(b) PROCESQ Q;
     =
     IF---THEN
     EXIT Q;
     FI;
     =
     BEGIN R;
     =
     EXIT Q;
     END R;
     CASE n OF X;
     (1) --- IF --- THEN EXIT X;FI;
     END X;
     =
     END Q;

Restrictions The referenced block, procedure or compound must not be nested within the block in which the EXIT resides.

System action Produces an assembler instruction for branching to the end of the named block or compound.
7.11 The IF-THEN-ELSE macro

**Purpose**
Specifying conditional statements or compound statements.

**General form**
```
IF Argument A THEN Argument B
ELSE Argument C?
FI;
```

Argument A is a logical expression specified by the user. Arguments B and C are the program segments which may again contain IF-THEN-ELSE constructs.

**Examples**
(a) IF A > B THEN
   -
   FI;
(b) IF --- THEN
   -
ELSE
- 
FI;
(c) IF --- THEN
   -
   IF --- THEN
   -
   IF --- THEN
   -
   FI;
ELSE
   IF --- THEN
   -
ELSE
   - 
   FI;
   - 
   FI;

**System action**
1. Calls macro LOGEXP which produce assembler code for evaluation of a logical expression. The LOGEXP macro must be defined by the user in accordance with his needs. (See section 9.6.2.)
2. Produce code for the evaluation of the conditional statements.
3. The "Dangling else" problem in a nested IF-THEN-ELSE construct is solved in the same way as in ALGOL 68. Every IF compound in a nested IF-THEN-ELSE structure is bracketed by the opening IF and a closing FI (see example c). Thus, it is quite clear which piece of code belongs to which IF.

### 7.12 The NOP macro

**Purpose**

Implementation of a dummy statement.

**General form**

NOP ;

**Examples**

(a) CASE N OF
   (1) ---;
   (2) NOP ;
   (3) ---;

   END;

**System action**

Produces the code:

BCR 15,0

### 7.13 The COMMENT macro

**Purpose**

To write comments.

**General form**

` Argument A`

**Examples**

(a) ` FIRST SOLUTION`
(b) ` EXAMPLE 1`
(c) ` LINEAR EQUATION`

**Restrictions**

A comment must be written within one line. If a comment is too large for a line, it must be divided into a number of comments, each covering one line.

**System action**

Produces no code.
7.14 Activation of the operating system

The different functions of the operating system are activated simply by writing calls to the required Assembly language system macros. If the EPS programmer writes a call to a 360/370 system macro in his source program, and this call is not at the same time a call to an EPS macro, then EPS will copy this call into the Assembly program produced by EPS. A user should, therefore, not define EPS macros whose call is identical to calls of system 360/370 macros that may be needed. The object program will thus be composed of Assembly language instructions generated by expansion of the EPS macro calls of which the source program is composed, and of the calls of the 360/370 system macros which have been copied by EPS from the source. The programmer may in conclusion write a call to a 360/370 system macro between any two statements in his source program.

Example: SET A = B;
TIME
SET Q = R;
Chapter 8. Error messages and macros for program verification and maintenance

This chapter deals with two types of errors. Sections 8.1 - 8.4 deal with syntax errors which are detected by the extended ML/I processor. Section 8.5 deals with syntax errors detected by the macros of EPS. Section 8.6 deals with a precompilation pass for syntax errors detection. Sections 8.7-8.12 deal with debugging facilities detecting errors in the program logic ("run time errors").

Syntactic Errors

EPS detects all errors and prints a message at every occurrence. An error message consists of a statement describing the particular error that has been detected with a print-out of the current context. This print-out enumerates all the macro calls and insertions of arguments or delimiters that are currently being processed, together with a line number to indicate the state of the scan in each case. Error messages are printed on an implementation-defined medium (see Section 4 of relevant Appendix) called the debugging file. This is normally a printer or on-line typewriter.

8.1 Example of an error message

An example of an error message is the following. Assume the user has written:

```
MCSET Y10 = 56
```

in the source text. Then the following message would be given:

```
ERROR(S)
ARGUMENT HAS ILLEGAL VALUE, VIX "Y10"
DETECTED IN
MACRO MCSET WITH ARGUMENTS
1) Y10
2) 56
CALLED FROM
LINE ... OF SOURCE TEXT
```

8.2 Notes on context print-outs

The printout of the context should be largely self-explanatory, but the following points should be noted.
(a) The line number is one greater than the number of newlines so far encountered in the piece of text to which it refers. Line numbers refer to scanned text, not to value text.

(b) If a macro call or insert straddles more than one line of text, then the line numbers of both the beginning and the end of the call or insert are printed (e.g. CALLED FROM LINES 6 TO 21 OF SOURCE TEXT).

(c) When the arguments of a call are enumerated, the text of each argument rather than its value is printed.

(d) If a piece of text in an error message consists of a single layout character, then the corresponding layout keyword, enclosed in parentheses is used in its place, for example:

   DELIMITER (NL) OF MACRO X NOT FOUND

In addition a null piece of text is represented by (NULL).

(e) Any multi-atom delimiter occurring in an error message is printed in full. A space is printed between two adjacent atoms if spaces are permitted between the atoms (i.e. if WITHS has been used rather than WITH in their definition).

Note (d) above applies to each atom. As an example, a message involving the multi-atom macro name "MCSUB(" would read:

   MACRO MCSUB ( CALLED FROM ...

(f) There is an implementation-defined number 2N (see Section 4 of the relevant Appendix) which is the maximum length of a piece of text that can be inserted in an error message. If a piece of text is too long, the first N-4 characters and the last N-4 characters are printed, separated by three dashes and some spaces.

(g) If the text of an error message is about to overflow a line, then a newline is artificially inserted.

8.3 Complete list of messages

This Section contains a complete list of all the error messages produced by the original ML/I.
8.3.1 Illegal macro element

Message flag number IS ILLEGAL MACRO ELEMENT

Description The number, which is the value of the subscript or macro expression associated with the flag, is either too large or too small. Alternatively, macro elements of the type designated by the flag do not exist in the current context (e.g. there are no arguments or temporary variables in the source text).

System action The current operation macro or insert is aborted.

8.3.2 Arithmetic overflow

Message ARITHMETIC OVERFLOW

Description Overflow has occurred during the evaluation of a macro expression or subscript. This message occurs when an attempt is made to divide by zero. It may also occur under other circumstances but these are implementation-defined (see Section 5 of the Appendix).

System action The current operation macro or insert is aborted.

8.3.3 Illegal input character

Message ILLEGAL INPUT CHARACTER

Description A character of the source text is not in the character set of the implementation.

System action The illegal character is replaced by a fixed implementation-defined character called the error character (is not implemented in EPS). A typical error character is the question mark.

8.3.4 Illegal macro name

Message ILLEGAL MACRO NAME AFTER WARNING, VIZ 'atom'

Description A warning marker is followed, possibly with intervening spaces, by the given atom which is not a macro name (nor the start of a multi-atom macro name). If this error occurs within an argu-
ment the above message is printed both when the argument is originally scanned and also each time it is inserted.

System action

The warning marker is treated as if it had not been recognised as an environmental name, and the atom which follows is treated as if no warning marker had occurred. Thus, for example, a skip name following a warning marker will be treated as such.

8.3.5 Unmatched construction

Message

DELMITR name [OR name*?] OF (MACRO) name (SKIP) (INSERT)

IN LINE number OF CURRENT TEXT NOT FOUND

Description

The given construction which starts in the given line of the current piece of text is not complete. Note that the line number is relative to the current piece of text. When the error was detected the scan was searching for the given delimiter (or for one of the given alternative delimiters). The error is detected only when the scan reaches the end of the source text or the end of a piece of inserted text or replacement text.

Possible causes

A mismatch of the delimiters of a construction nested within the given one can cause this error since delimiter matching is liable to get "out of phase" as a result. Alternatively, an incorrect specification of a delimiter structure can cause delimiters to be matched in a way not intended by the user and, again, the error may be in a nested construction rather than in the given one.

System action

In the call and insert cases, the effect is as if the text from the macro or insert name to the current point of scan was deleted. In the skip case, text skipped over is treated in the normal way and the skip is artificially terminated.

8.3.6 Illegal syntax of argument value

Message

ARGUMENT number HAS ILLEGAL VALUE, VIZ 'value'
Description The given value of an argument to an operation macro or insert has not the required syntax. For operation macro arguments see appropriate "Restrictions" subsection of Sections 5.2, 5.3 or 5.4, or if the argument is (supposed to be) a structure representation then see Section 5.1.6. For arguments to inserts see Section 2.6.7.

System action The current operation macro or insert is aborted.

8.3.7 Redefined label

Message LABEL number IS MULTIPLY-DEFINED

Description An attempt has been made to re-define a label that has already been defined within the current text.

System action The new definition is ignored.

8.3.8 Undefined label

Message LABEL number REFERENCED IN LINE number OF CURRENT TEXT NOT FOUND

Description A call of MCGO references an undefined label. This error is detected when the scan reaches the end of a piece of text (since it performs a search for the missing label). If any constructions are unmatched, the message(s) of Section 8.3.5 are printed with this message.

Possible causes An attempted backward MCGO in the source text or an attempted MCGO from one piece of text to another can cause this error. Alternatively, it can be caused by an unmatched construction within the scope of a forward MCGO.

System action The effect is as if the designated label had been found at the very end of the current piece of text.

8.3.9 Storage exhausted

Message PROCESS ABORTED FOR LACK OF STORAGE [POSSIBLY DUE TO other messages?]
Description  ML/I has used up all its available storage. If the current text is the source text then the following additional information is given: if there are any constructions currently unmatched, or if a search is being made for a label as a result of a forward MCGO, then the messages of Sections 8.3.5 and 8.3.8 are printed with this message.

Possible causes  Storage is taken up by macro variables, by the name environment, by a macro call or insert in the source text, and by nested calls and/or inserts. Hence an unmatched macro call in the source text or a call with a very long argument can cause this error. Alternatively, it can be caused by an endless or very deep recursive nest, by the name environment being too big, or by a combination of all these factors.

System action  The current process is aborted.

8.3.10 System error

Message  SYSTEM ERROR

Description  There has been a machine error, an operating error or an error in the implementation of ML/I.

System action  The current process is aborted.

8.3.11 Subsidiary message

Message  (MACRO ) name ABORTED DUE TO ABOVE ERROR (INSERT)

Description  This message occurs as a subsidiary message every time an error causes the operation macro or insert currently being performed to be aborted. Any construction that has been aborted is given a null value.
8.4 Implementation-defined messages

In this implementation of ML/I some errors which are concerned with name declarations and scope of names have been added.

8.4.1 Illegal identifier

Message name IS AN ILLEGAL IDENTIFIER

Description An identifier may be composed of no more than six letters or digits, the first of which must be a letter.

8.4.2 Multiple-defined names

Message name IS MULTIPLY DEFINED

Description Names of variables, procedures, blocks and compounds must be unique within the same external procedure. Procex names must be also unique.

8.4.3 Undeclared variables

Message name HAS NOT BEEN DECLARED

Description Usage of an undeclared variable.

8.4.4 Multiple defined parameter

Message name IS MULTIPLY DEFINED

POSSIBLE CAUSES:
(1) TWO PROCEDURE PARAMETERS HAVE THE SAME NAME
(2) A PARAMETER AND A VARIABLE HAVE THE SAME NAME
(3) TWO VARIABLES HAVE THE SAME NAME.
DETECTED IN PROCEDURE procedure name.

8.4.5 Special syntax errors detection and the TEXT macro

There are some common errors which causes much trouble. Most of them are concerned with the delimiter structure. The absence of a delimiter causes in some cases many redundant error messages and in other cases the processor may search for the missing
delimiter until the end of the program. The producing of value text and the generation of object code is, in a program which has such errors, obviously considerable waste of time.

In order to avoid such a waste it is recommended to proceed the 'production' run with some syntax errors detection. In this run no value text is produced. This is accomplished by calling the macro TEST in the beginning of the source program.

8.4.6 The TEST macro

**General form**

```
TEST
```

**System action**
The macro assigns the value zero to the permanent variable p25, and thus inhibits any production of value text in all the EPS macros.

8.5 Error messages which are detected by EPS

EPS provides some error messages detected in macro time but are not concerned with the syntax of ML/I.

These errors concerned with the syntax of EPS are described in chapter 7.

These error messages differ from the messages described in 8.3 and 8.4 by the way the messages refer to the place in the source program where the error was met. In these messages only the consecutive line number, in the source program, where the error was met is given.

The complete error messages detected by EPS are listed in the following sections.

8.5.1 Procedure, block or compound has not been terminated normally

**Message**

```
(PROCEX) (PROC)
(BEGIN) name HAS NOT BEEN TERMINATED
(CASE) (REPEAT)
BY END name;
```
Description: A procedure, block or a compound which is named must be terminated by END name; or by END; when a procedure terminates by mistake with an END which has a name which is different from the procedure name it will cause the above message. When this error appears, compilation is suspended and the system searches for a possible appearance of an END with the correct name. If this END is found an error message is printed (described in 8.4.8) and thereafter compilation continues.

8.5.2 END was found not in place

Message: END name; HAS BEEN FOUND NOT IN PLACE OR APPEARS MORE THAN ONCE

Description:
(1) A named END which is not ending a procedure, block or a compound having the same name causes the above message (see 8.5.1)

(2) Appearance of two or more END's with the same name may also cause the above message.

8.5.3 Undefined procedure block or repeat

Message: (PROCEDURE) name IS NOT FOUND
(BLOCK) (REPEAT) (CASE)

Description: An attempt to reference a not defined procedure block or a repeat compound, by the EXIT macro.

8.5.4 Formal and actual parameters matching

Message: TYPES OF FORMAL AND ACTUAL PARAMETERS OF PROCEDURE name DO NOT MATCH

Description: It is required that types of the formal and actual parameters match. If this is not the case the program will not be executed.

8.5.5 Illegal parameter

Message: name IS AN ILLEGAL PARAMETER
Description
An occurrence of an actual parameter, which is not a variable or a constant causes the above message to be printed.

8.5.6 Unlabeled procedure

Message
UNLABELED PROCEDURE

Description
Procedure with no name cannot be called. Absence of a procedure name is, therefore, an error.

8.5.7 Illegal subscript

Message
name MAY NOT APPEAR AS THE LENGTH OF A DECLARED ARRAY

Description
The length of an array in declaration statement must be either a variable or an integer constant.

8.5.8 Multiple defined procedure name

Message
name IS A MULTIPLE DEFINED PROCEDURE NAME

Description
Two or more procedures having the same name.

8.6 Precompilation pass for errors detection

EPS detects some kinds of errors which require a precompilation pass. Since the macro CALL which calls a procedure may appear before the procedure definition, it is unable to check for matching of actual and formal parameters in one pass. The same applies for the EXIT macro. Using these macros and/or other macros which will be added by the programmer and have the same obstacle requires a precompilation pass. EPS has a precompilation pass. In this pass procedure parameters are entered into the parameter table (see Section 1.2 in Appendix) and enables checking for matching of formal and actual parameters in procedure calls. All the entity names which may be referenced by the macro EXIT are also entered into the parameter table and this enables detection of errors concerned with the EXIT macro. Moreover, this pass detects illegal constituents in the formal parameters list as well as multiple defined entity names. This pass may be skipped if none of the errors concerned with it were met (see chapter 9 for details).
8.7 Macros for program debugging and maintenance

We shall first describe the program verification and maintenance procedure that our aids are designed to support.

The process starts with sets of test data and trace specifications that are prepared at the program design stage to enable a quick and "nearly complete" verification. The insight gained through the debugging process will often result in major revisions and improvements of the program. The professional programmer will, nevertheless, end the process with sets of test data and trace specifications that enable a quite thorough check of the program. These test data and trace specifications are the principal tools for maintenance after the program has been released. When a bug appears in a released program, the program is rerun with the data that caused the bug, but now with the trace specification activated. When the error is repaired, the program is run with the test data to check that no new errors have been introduced through the corrections.

By analysing the process, it is seen that we need a facility by which we may easily switch to either "debugging mode" where the trace facility is "enabled", or to "production mode" where it is "disabled". This goal may quite easily be achieved when the compilation is done by a macro processor. Say that we wish to trace any change in the value of a certain variable. In our language, such a change may occur in a SET or in a READ macro. These macros will, therefore, be defined such that if the "debugging switch" is "on" and if the macro assigns a value to the variable to be traced, then and only then, there will be produced code for printing a message giving the new value of the variable. Note that the code for producing this trace message is not generated when the program is in "production mode", such that there is no overhead in production runs.

The trace specifications may be divided into the temporary traces, which are removed from the program after the initial debugging process, and the permanent ones, which remain in the program as a maintenance tool. The permanent trace specifications may be regarded as an integral part of the source program, although they are not translated into object code in production runs. It is, however, convenient to have these trace specifications in the source program when we have to analyze trace messages in debugging runs.
One of the major advantages of structured programming is that it facilitates the tracing of the flow of the program. It is, therefore, natural to link the facilities for tracing the flow of the program to the macros employed for structuring the program, e.g. BEGIN-END, PROC-END. When a flow trace is specified and the program is in "debugging mode", code is generated to print a message whenever a block or procedure is entered, or when a "THEN part" or "ELSE part" in an IF construct is entered.

8.8 The DEBUG and DEBUGOFF macros

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Specifying that subsequent code is to be run in &quot;debugging mode&quot; or in &quot;production mode&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td>General form</td>
<td>(1) DEBUG;</td>
</tr>
<tr>
<td></td>
<td>(2) DEBUGOFF;</td>
</tr>
</tbody>
</table>

Examples

(a) MAIN PROCEX;
    DEBUG;
    ...
    DEBUGOFF;
    ...
    END MAIN;
(b) DEBUG;
    PROCEX A;
    DEBUG OFF;
    ...
    ...
    END A;

System action

The DEBUG macro switches to "debugging mode" whereas the DEBUGOFF macro switches to "production mode". These macros may appear anywhere in the program and even before MAIN PROCEX. The default mode is "production mode", in other words the program will be in "production mode" if none of these macros calls appear. Each DEBUG macro switches to "debugging mode" only in the procex where it appears, which means that every procex
END causes switch to "production mode".

Each procex to be debugged, has to be preceded by a call of DEBUG. Typically, only one PROCEx will be in "debugging mode" in a certain run.

If one wishes to check only the procedure name and parameters and not to check the procedure body, example (b) may be helpful.

The DEBUG and the DEBUGOFF macros assign the value one or zero respectively to the permanent macro time variable P17, which thus indicate the debugging state.

8.9 The CHECK and the CHECKOFF macros

Purpose Specification of the variables to be traced. The program segment where a certain variable, say R, is to be traced is thus bracketted by the macro calls CHECK and CHECKOFF.

General form

(1) CHECK [Argument A,*?] Argument A;
(2) (CHECKOFF) [Argument A,*?] Argument A;

Arguments A are variable names which may be subscripted.

Examples

(a) CHECK A;
(b) CHECKOFF B,C,A,Dly;

Restrictions Any variable which appears in the call of these macros must of course be known at that stage of the program. These macros are valid only in the procex where they appear.

System action The variables specified in a CHECK or a CHECKOFF macro are tagged by a one or zero, respectively, in the macro time symbol table.
8.11 The FLOW and the FLOWOFF macros

**Purpose**
The FLOW and the FLOWOFF macros are used for bracketting the program segments where it is required to trace the flow of the program.

**General form**
(1) FLOW;
(2) (FLOWOFF;)
(FLOW OFF;)

**Examples**
FLOW;
PROCEX MAIN;
BEGIN;
FLOWOFF;
REPEAT A;
END A;
END;
END MAIN;

**Restrictions**
A number of FLOW-FLOWOFF pairs may be employed in the same program. Each such pair of macro calls delimit a program segment where tracing of program flow is required. These program segments may neither overlap each other nor be nested.
The FLOW macro assigns the value 1 to the "permanent" compile time variable Pl6 which thus indicate the "traceflow state". Similarly FLOWOFF will assign a zero to Pl6.

In the "traceflow state" the macros for controlling the flow of the program described in chapter 7, will in addition to the usual code also generate code for printing run time messages on passing crucial program points, e.g. block entries (see table). Ultimately, these messages will constitute a printed "log book" of the flow of control through the program segment checked.

The "program flow macros" affected by FLOW are:

<table>
<thead>
<tr>
<th>macro</th>
<th>code is generated for printing the message</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>ENTRY TO BLOCK block name</td>
</tr>
<tr>
<td>REPEAT</td>
<td>LOOP loop name IS EXECUTED number TIMES</td>
</tr>
<tr>
<td>IF</td>
<td>(THEN) PART OF IF CONSTRUCT IS EXECUTED (ELSE)</td>
</tr>
<tr>
<td>CASE</td>
<td>SEGMENT number OF CASE name IS EXECUTED</td>
</tr>
<tr>
<td>EXIT</td>
<td>BRANCH TO END OF procedure name</td>
</tr>
<tr>
<td>CALL</td>
<td>PROCEDURE name IS CALLED</td>
</tr>
<tr>
<td>All messages are ended by</td>
<td>IN LINE line number</td>
</tr>
</tbody>
</table>

The REPEAT and BEGIN macros may be "partly" in FLOW state. This may for instance be useful in the case of a loop where we are merely interested in knowing the number of times it is executed, but not in recording of macro calls etc. inside the loop body. An example of such a message is:
FLOW;
SET SUM = 0;
SET I = 1;
REPEAT SUMSINES;
EXIT IF I = N;
FLOWOFF;
SET SUM = SUM + A;
SET I = I + 1;
FLOWON;
END SUMSINES;

8.11 The INIT and INITOFF macros

Purpose To check whether the variables employed in a program have been given a value prior to their usage.

General form INIT;
(INIT OFF;)
(INITOFF;)

System action Assign the value 1 or 0 respectively to the permanent variable P18, which represent the "INIT" flag. The default value of this flag is OFF. The ON option will generate code for assigning the value "undefined" to all variables upon their declaration.
In the macros where the values of variables are used code is generated in order to check whether the value of the variables to be used is "undefined".
The INIT facility involves some overhead and should be switched off when the code has been checked.

8.12 Errors detected by the assembler

The EPS object language is assembly of IBM OS/360. Since EPS does not detect all the possible syntax errors as well as logical errors, some errors may be detected at assembly time by the assembler. The assembler will also detect some errors in run time like division by zero.
9.1 Introduction

The trivial use of EPS is as a convenient tool for writing assembly programs. In this case one utilizes only the "Common Macros" defined in chapters 6,7, which provide quite sophisticated facilities for structured programming and for debugging.

The most important use of EPS is probably to produce new programming languages for special application areas. Concepts which are common in such a problem area will be included in the specialized programming language, so that solution of problems may be expressed in a clear and natural way. To do this, macros for manipulations with the concepts and data types that are special for the problem area in question must be defined and added to EPS. These additional macros may be defined by using the basic macros of ML/1, the common macros of EPS and any macro which has been previously defined by the user. In any case, the object language of EPS is assembly 360.

The rules for defining new macros were described in chapters 2-5. There are some further points that are essential for the proper use of EPS, and which are the subject of the following sections.

9.2 Possible sources of error

The following Sections illustrate some areas where the user of EPS should take special care.

9.2.1 Jumping over expanded code

When macros are expanded into an assembly language program, great care must be taken with instructions of the form "jump to location counter \( \pm N \)", since there may be macros calls within the scope of the jump which expand into several machine instructions. For this reason it is helpful to choose macro names that cannot be mistaken for the MNEMONIC machine instructions.

9.2.2 Generation of unique labels

If a macro generates code which involves an execution-time label, then a
different label must be generated at each call of the macro. The technique described in Section 2.6.8(b) can be used for this purpose. The same applies, in some cases, to execution-time temporary variables.

9.2.3 Lower case letters

Note that in implementations where the character set includes both upper and lower case letters, only upper case letters may be used for vocabulary words of ML/I. This applies to the names and secondary delimiters of operation macros, to keywords and to insert flags. Further note that, for example, "PIG", "Pig" and "pig" are three different atoms.

9.2.4 Use of newlines in definitions

Remember that layout characters within replacement text are treated like any other characters. They should therefore be used with great care as they affect the format of the output text. Thus:

```
MCDEF LOAD AS <LD>

LOAD X
```

would generate:

```
LD X
```

whereas:

```
MCDEF STORE AS <ST>

STORE Y
```

would generate:

```
ST

Y
```

Moreover:

```
MCDEF JUMP AS <B>

JUMP LB6
```

would generate:

```
B

LB6
```
since JUMP would be defined as a null macro.
9.2.5 Use of redundant spaces

As a general rule extra spaces are ignored within text that forms an instruction to ML/I but are treated like any other character within text that ML/I manipulates.

Spaces may be chosen as construction names, but in any context where spaces are ignored they are ignored even if space is a construction name. In particular, spaces are ignored after warning markers so, when in warning mode, it is not possible to have a macro name commencing with a space.

Below is a list of some of the places where spaces are ignored:

(a) At the beginning or end of an argument to an operation macro (before evaluation).
(b) Ditto for an argument to a substitution macro, provided the insert flag B is not used.
(c) After a warning marker.
(d) Within a macro expression (except within variable names or constants).
(e) Within the argument to an insert (except within variable names or constants).
(f) Within the values of those operation macro arguments that specify options.

Within structure representations one or more spaces act as a separator.

9.3 Simple techniques

This Section illustrates a few techniques for solving some simple problems. In general, only one solution is given but there are often several equally good solutions. In some cases a problem has been described in terms of the use of ML/I as a preprocessor to a particular language, but in each case the problem has counterparts in other applications.

9.3.1 Interchanging two names

Problem It is desired to interchange the names PIG and DOG
in a piece of text.

Solution
The complete name environment is set up as follows:

\[
\begin{align*}
\text{MCSKIP} & \quad MT, < > \\
\text{MCDEF} & \quad \text{PIG AS "DOG"} \\
\text{MCDEF} & \quad \text{DOG AS "PIG"}
\end{align*}
\]

and the desired result is achieved by evaluating the given text under this environment.

Notes
(a) In this example there is no necessity to have an insert definition in the environment.
(b) Notice that two pairs of literal brackets are used to surround the pieces of replacement text. One pair is stripped off at definition time and the second at replacement time. If the brackets were omitted, ML/I would endlessly replace one name by the other.

9.3.2 Removing optional debugging statements

Problem
It is desired to include a number of extra statements in a FORTRAN program in order to aid in debugging its execution. These are to be removed when the program is debugged. Each statement ends with a newline.

Solution
Some unique atom, say DEBUG, is written at the beginning of each debugging statement. Before the FORTRAN program is compiled it is passed through ML/I. If it is desired to include the debugging statements then the following skip definition is placed in the name environment:

\[
\begin{align*}
\text{MCSKIP} & \quad \text{DEBUG}
\end{align*}
\]

This causes each occurrence of DEBUG to be deleted. When it is desired to delete the debugging statements then the following skip definition is used:

\[
\begin{align*}
\text{MCSKIP} & \quad \text{DEBUG} \quad \text{NL}
\end{align*}
\]

9.3.3 Inserting extra debugging statements

Problem
It is desired in a PDP-7 Assembly Language program for a
particular variable COW, to replace every occurrence of DAC COW (deposit accumulator at COW) by a call to a subroutine (which perhaps prints the value assigned to COW). This call has form JMS TYP COW.

Solution
MCDEF DAC WITHS COW AS <JMS TYP COW>

9.3.4 Deleting a macro

Problem
It is desired to delete the macro GONE from the current environment.

Solution
The following skip accomplishes this:

MCSKIP D, <GONE>

Notes
(a) The literal brackets prevent GONE being called during the evaluation of the second argument of the above MCSKIP.
(b) Strictly speaking the macro GONE is overridden rather than deleted (see Section 4.7(a)).

9.3.5 Differentiation between special-purpose registers and storage locations

Problem
It is desired to define an INTERCHANGE macro for PDP-7 Assembly Language so that, as well as being used to interchange the values of two storage location it can be used to interchange the accumulator with a storage location. In the latter case "ACC" is written as the first argument of the call.

Solution
Assuming the existence of a MOVE FROM macro, which moves the value of one storage location into another, the definition of INTERCHANGE is written:

MCDEF INTERCHANGE WITH ( , ) WITH NL AS <MCGO LI IF %A1. = ACC
MOVE FROM %A2. TO TEMP; MOVE FROM %A1. TO %A2.;
MOVE FROM TEMP TO %A1.;
MCGO LØ
%L1. DAC TEMPAC
MOVE FROM %A2. TO TEMP; MOVE FROM TEMPAC TO %A2.;
LAC TEMP>
9.3.6 Testing for macro calls

**Problem**
It is desired to find out whether an argument of a macro call itself involves any macro calls, inserts or skips.

**Solution**
Compare the written form of the argument with its evaluated form. (It is assumed that any construction occurring within the argument would cause these two forms to be different.)
The following is an example of how the test might be written:

```
MCGO L1 IF %A1. = %WA1.
```
Alternatively, if it was only required to test if the argument involved any macro calls, the test might be written:

```
MCGO L1 IF MCNODEF%A1. = %A1.
```

9.3.7 Searching

**Problem**
It is desired to search the source text to find all occurrences of given atoms.

**Solution**
Define macros such as:

```
MCDEF X
AS <MCNOTE HERE IS <X>
```

It is best to send the output text itself to a null channel so that the only printed output is the MCNOTE message.

9.3.8 Bracketing within macro expressions

**Problem**
Parentheses cannot be used within macro expressions.

**Solution**
Use nested inserts. For example, to insert the value of $(P1 + 6) / (P3 - 2)$ write:

```
%%P1+6./%P3-2.
```

9.3.9 Deletion from source text only

**Problem**
It is desired to delete a given atom only if it occurs in the source text.
Solution

Use temporary variable three, e.g.:

 MCDEF X AS <MCGO LØ IF T3 EN 1
     %WDØ.>

9.4 * Sophisticated techniques

This Section illustrates some techniques which may be of value to the more sophisticated user.

9.4.1 Macro-time loop

Problem

A macro-time interaction statement is required in order to generate repetitive text.

Solution

The macro MCFOR defined below serves this purpose. It allows the step size to be optionally omitted; in this case a step size of one is assumed. MCFOR should be regarded as a "black box" by the reader who finds the definition below hard to understand. The part labelled "L2" is to deal with a negative step size.

 MCDEF MCFOR = OPT STEP N1 OR N1 TO ALL NL REPEAT  
     AS<MCSET %A1. = %A2.  
     MCSET T3 = 1  
     MCGO L1 IF T1 EN 4  
     MCSET T3 = %A3.  
     MCGO L1 IF T3 GR Ø  
     %L2. MCGO LØ IF %AT1-1. GR %A1.  
     %AT1. MCSET %A1. = %A1. + T3  
     MCGO L2  
     %L1. MCGO LØ IF %A1. GR %AT1-1.  
     %AT1.MCSET %A1. = %A1. +T3  
     MCGO L1

(a) MCFOR P1 = 1 TO 20  
     JMP LAB%P1.  
     REPEAT  
     would generate the twenty instructions  
     JMP LAB1,..., JMP LAB20.

(b) MCFOR P6 = 20 STEP -1 TO 1  
     JMP LAB%P6.  
     REPEAT  
     would generate the above twenty instructions in reverse order.
(c) MCSET P2 = 1
MCFOR P1 = 1 TO 10
%P2.MCSET P2 = P2+P2
REPEAT
would generate the first ten powers of two.

Notes
(a) The controlled variable must be a permanent variable.
(If it were a temporary variable, MCFOR would try to use its own temporary variables rather than those of the calling environment thus causing an error.)
(b) The initial value, step size, and final value must be macro expressions not involving temporary variables.
(c) MCFOR is a substitution macro, not an operation macro.
(d) Calls of MCFOR may be nested.
(e) MCFOR can be used to perform loops within the source text, thus surmounting the restriction that backward MCGOs within the source text are not allowed.

9.4.2 Examining optional delimiters

Problem An IF macro has form:

```
IF arg A (GE) arg B (GR) arg C (LT) arg D (LT) THEN ...
```

Within the replacement text of IF, it is desired to examine the form of the first delimiter and go to L1 if the delimiter is GE, to L2 if it is GR, etc. This problem can obviously be solved by writing a large number of conditional MCGO statements but this would make the IF macro very slow and cumbersome.

Solution The various possible delimiters can be defined as macros thus:

```
MCDEF GE AS 1
MCDEF GR AS 2
etc.
```

and then the requisite switch statement can be written:

```
MCGO L%D1.
```
Notes

(a) The definition of the delimiters of IF as macros does not affect the scanning of a call of the IF macro since the use of an atom as a delimiter takes precedence over its use as a macro name.

(b) It is necessary to place the definitions of GE etc. after the definition of IF or else to enclose the structure representation of IF within literal brackets.

(c) This technique will not, as it stands, work for name delimiters. However, see Section 9.4.8.

9.4.3 Dynamically constructed calls

Problem

It is required to implement a WHILE macro of form:

(GE)

(GR)

WHILE arg A (LT) arg B DO

(=)

(etc.)

arg C

END

Within the replacement text of this macro it is desired to call the IF macro with the first delimiter of this call of IF the same as the delimiter that occurred in the call of WHILE. However, as was seen in Section 3.6, it is not possible to do this by writing:

IF ... %DI. ... THEN ...

Solution

It is necessary to use a temporary macro definition to build up the text for the required call of IF and then to call the temporary macro. This could be achieved thus:

MCDEF TEMP AS IF ...

%WDI. ...

THEN ...

END TEMP

Notes

(a) WDI was used rather than DI since GE etc. are macros and it is not desired to call them at this stage.

(b) Note that the insert %WDI. is not enclosed in literal brackets and is thus inserted when TEMP is defined. Thus if this delimiter were GR, then the replacement text of
9/10

TEMP would be:

```
IF ... GR ... THEN ...  
```

and calling TEMP would then accomplish the required call of IF.

(c) TEMP is enclosed in literal brackets when it is defined in case there is already a TEMP macro in existence. This might arise, for example, if the WHILE macro was called recursively.

(d) TEMP should be a local macro rather than a global one so that the storage it occupies is released when an exit is made from the WHILE macro.

(e) This general technique can be used in all cases where it is required to build up a call dynamically. The next Section contains a further example of the technique.

9.4.4 Arithmetic expression macro

**Problem**

A macro whose name is "(" has been designed so that, when supplied an arithmetic expression as argument, it generates assembly code to calculate the value of the expression and to place the resultant value in an accumulator. This macro will be referred to as "the parenthesis macro". A typical call of the parenthesis macro might be:

\[(PIG + (Y/6)*Z - 16)\]

This involves a nested call of the same macro. The arguments of the outer call are PIG, (Y/6), Z and 16, and the delimiters are +, * and -. It is desired to use this macro to implement a SET macro, which allows a macro expression as argument. Calls of SET might be:

```
SET DOG = Y
SET VAR = (VAR + 6)/13 - PIG
```

**Solution**

The solution to this problem is not to give the SET macro a complicated delimiter structure but rather to regard it as a macro with two arguments. The second argument is then passed
down to the parenthesis macro, which breaks it down into operators and operands. The SET macro is defined:

```
MCDEF SET = NL
AS <MCDEF TEMP AS <(>%WA2.<)> TEMP
[instruction to store the result in %A1.]
```

Notes
(a) Notice the use of TEMP to build up a call of the parenthesis macro. In the second of the above examples of SET, for instance, TEMP would be defined as:

```
((VAR+6)/13 - PIG)
```

When TEMP was called, it would result in a call of the parenthesis macro with arguments (VAR+6), 13 and PIG.

(b) It would have been wrong to call the parenthesis macro from within SET by writing simply (%A2.), since this would have been interpreted as a call with one argument.

9.4.5 Formal parameter names

Problem It is desired to use the name TAXRATE for the first formal parameter of the macro DEDUCT.

Solution The first part of the definition of DEDUCT is written:

```
MCDEF DEDUCT ... AS MCDEF TAXRATE AS %A1.
```

Thereafter within the replacement text of DEDUCT, TAXRATE can be written in place of "%A1."

9.4.6 Intercepting changes of state

Problem It is desired in PDP-7 Assembly Language to generate some decimal constants within the replacement text of a macro SIZE. However, PDP-7 Assembly Language has two statements, OCTAL and DECIMAL, to control the base to which constants are to be written, and this might vary between calls of SIZE. Furthermore, it is desired that a call of SIZE should not change the base
behind the user's back.

**Solution**

A permanent variable, say P10, is used as a switch, the value zero being used to indicate an octal base. The following is written at the start of the source text:

```plaintext
MCDET P10 = 0
MCDEF OCTAL AS <MCSET P10 = 0 %WD0.>
MCDEF DECIMAL AS <MCSET P10 = 1 %WD0.>
```

and the definition of SIZE is written:

```plaintext
MCDEF SIZE AS <MCSET T1 = P10 DECIMAL
:
MCGO L0 IF T1 EN 1 OCTAL
>
```

thus ensuring that the base is returned to its original state.

**Notes**

This technique is also useful for the following problem: the user has written a macro SUBS to generate code for subscripted vectors and it is necessary that SUBS generates different code for the two following calls:

(a) LAC SUBS (V, 1) Load accumulator from element.
(b) DAC SUBS (V, 1) Store accumulator in element.

The problem is solved by using the above technique to cause LAC and DAC to set a switch which the SUBS macro can then test to find out which instruction preceded its call.

### 9.4.7 Remembering code for subsequent insertion

**Problem**

It is desired to design two macros, REMEMBER and INSERT, to enable the user to remember text for subsequent insertion. These macros are used in the following way. REMEMBER is called with a piece of text as argument. REMEMBER does not generate any code but remembers its argument for subsequent insertion. When the INSERT macro is called all the pieces of text that have been remembered are inserted.
Solution

A sequence of global macros I1, I2, \ldots IN is used, the value of N being given by a permanent variable, say P1\emptyset. Each macro represents a piece of text that is to be remembered. The definitions of REMEMBER and INSERT would be written:

\begin{verbatim}
MCSET P1\emptyset = 0
MCDEF REMEMBER; AS <MCSET P1\emptyset = P1\emptyset + 1
MCDEFG I2P1\emptyset. AS %A1.
> MCDEF INSERT AS <MCFOR P1 = 1 TO P1\emptyset
RECALL I2P1.
REPEAT>
\end{verbatim}

where MCFOR is the macro of Section 7.4.1 and RECALL is a macro defined thus:

\begin{verbatim}
MCDEF RECALL
AS <MCDEF TEMP AS%A1.
TEMP>
\end{verbatim}

Notes

(a) The above solution tries to minimize the amount of storage used. It would have been possible to do without the RECALL macro. but this would have involved redefining TEMP N times within the MCFOR loop and so, albeit temporarily, using up rather more storage.

(b) Note that the macros I1 etc. must be global whereas the macro TEMP should be local.

(c) An apparently promising technique for this problem which fails because of excessive use of storage is the following. The entire remembered text is maintained by redefining the INSERT macro as below each time REMEMBER is called:

\begin{verbatim}
MCDEF REMEMBER; AS <MCDEFG<INSERT> AS INSERT%A1.
> The trouble with this approach is that old versions of INSERT can never be released, thus using up a very considerable amount of storage.
\end{verbatim}

9.4.8 Constructions with restricted scopes

Problem

It is desired to assign different meanings to a macro X within
different scopes. One meaning is to apply within the replacement text of a set of macros $M_1, \ldots, M_N$ whereas another meaning is to apply elsewhere.

Solution

One solution is to redefine X as a local macro within each of $M_1$ to $M_N$, but this is tiresome if $N$ is large and slower than the method below even if $N$ is one. A better solution is to place the two following definitions at the start of the source text:

\begin{verbatim}
MCDEFG X ... AS <replacement to be used in M1 to MN>
MCDEF <X ... > AS <replacement to be used elsewhere>
\end{verbatim}

The second definition overrides the first. Within the macros $M_1$ to $M_N$ the first definition can be re-incarnated by writing MCNODEF, which deleted the second definition. Any macros besides X that were used within $M_1$ to $M_N$ should also be defined as global.

Notes

(a) This technique can be used in a variety of applications. It is the best solution in almost all situations where a macro or set of macros has restricted scope, but where this scope does not consists simply of the replacement text of a single macro. Even in the latter case the technique is useful as it is faster than setting up the local definitions every time a macro is called.

(b) This technique can be used to extend the technique described in Section 7.4.2 to make it work for name delimiters. For example, if a macro had alternative names A and B and, within the replacement text of this macro, it was desired to insert the number 206 if the name was A and the number 15 if the name was B then this could be achieved, assuming "%" to be an unprotected insert, by writing :

\begin{verbatim}
MCDEFG A AS 206
MCDEFG B AS 15
MCDEF <OPT A OR B ALL ...> AS <... MCNODEF%DØ. ...>
\end{verbatim}
9.4.9 Optimizing macro-generated code

**Problem**
It is desired to optimize the code generated by EPS, in particular to cut down possible inefficiencies at the boundary between successive macros.

**Solution**
There are basically two approaches to producing optimal code:
(a) Code can be optimized as it is produced.

Typically this would involve using the permanent variables to maintain some sort of indication of the previous instruction(s) generated.
(b) A second pass can be made through the macro generated code, to search for various inefficient sequences of instructions. Except in simple cases, the second method is usually the better.

In many machines considerable optimization can be performed by maintaining where possible an indication of the contents of the accumulator(s) or other special-purpose registers and thus cutting out redundant loading instructions. This can be done by defining macros to map into numbers all the variables used in the code being generated. A permanent variable, say P1, could be used to indicate whether the accumulator was known to contain the current value of a particular execution-time variable. If so, P1 could contain the number of the variable, otherwise it could be zero. P1 would need to be zeroized when a label was placed, a subroutine was called, etc.

This might be achieved by defining a macro with many alternative names, covering all the situations where the accumulator was clobbered. The macro might be:

```
MCDEF OPT, OR JMS OR ADD OR ... ALL AS<MCSET P1 = 0
```

9.4.10 Macro to create a macro

**Problem**
This problem illustrates the use of a macro to set up the definition of another macro. The problem is as follows.

It is desired to design a macro EQUATE which equates one
vector to part of another. Thus the call:

EQUATE VEC1 TO VEC2 OFFSET 3

would cause each subsequent reference to an element of VEC1 which has form, say:

VECl (script)

to be translated into a reference to the corresponding element of VEC2, namely:

VEC2 (script + 3)

Solution

MCDEF EQUATE TO OFFSET NL AS <MCDEFG%Al. WITH () ASW. «%Al.)+%A3.)>

Examples

(a) The call:

EQUATE VEC1 TO VEC2 OFFSET 3

would be equivalent to writing the definition:

MCDEFG VEC1 WITH ( ) AS <VEC2(%A+3)>

Notes

(a) The main source of error in this sort of problem is to confuse the arguments of the macro that creates the definition with the arguments of the new macro being defined. The rule is that the latter should be enclosed doubly in literal brackets. Hence in the replacement text of EQUATE, the arguments within single literal brackets are the arguments of EQUATE, which are inserted when the new macro is defined, and the argument within double literal brackets is the argument of the new macro, which is inserted when the new macro is called.
9.5 Reserved entities

There are some reserved characters, macro names, registers and macro variables which are used in EPS for special purposes.

Instructions and restrictions on the use of these entities are listed below.

9.5.1 Reserved characters

The following symbols are used for special purposes and therefore may not be used for any other purpose.

a. The sequence %. is defined as the delimiter structure of a "protected global insert".

b. The sequence :: is defined as the delimiter structure of an "unprotected global insert".

c. The sequence &? is defined as the delimiter structure of a global skip with MT options on.

d. The sequence \ NL is defined as "global skip" with no option on. (This skip is used for writing comments).

e. The $ is used as new line representation. Any appearance of this character in the source text causes an immediate skip to the next line in the value text.

f. The character ! is used for start line representation. Any appearance of this character in the source text forces the next value text to begin at the start of the line, erasing any information which should have previously been printed in that line.

g. The character @ is used as a "stopcode" and is inserted automatically by the system at the end of the source text.
9.5.2 Reserved identifiers

The following identifiers are used for naming various system macros and they should therefore not be employed for naming new macros:

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>MAIN</th>
<th>PROCEX</th>
<th>MCNOOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>MCINS</td>
<td></td>
<td>MCMMSG</td>
</tr>
<tr>
<td>BEGIN</td>
<td>MCSKIP</td>
<td></td>
<td>MCENTRY</td>
</tr>
<tr>
<td>CASE</td>
<td>MCDEF</td>
<td></td>
<td>MCTYPE</td>
</tr>
<tr>
<td>CHAR</td>
<td>MCWARN</td>
<td></td>
<td>MCDIME</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>MCINS</td>
<td></td>
<td>MCTRACE</td>
</tr>
<tr>
<td>CHECK</td>
<td>MCSKIPG</td>
<td></td>
<td>MCTRACETAG</td>
</tr>
<tr>
<td>CHECKOFF</td>
<td>MCDEFG</td>
<td></td>
<td>MCG</td>
</tr>
<tr>
<td>DEBUG</td>
<td>MCWARN</td>
<td></td>
<td>MCNG</td>
</tr>
<tr>
<td>DEBUGOFF</td>
<td>MCNOSKIP</td>
<td></td>
<td>PROC</td>
</tr>
<tr>
<td>INT</td>
<td>MCNOINS</td>
<td></td>
<td>PROCEX</td>
</tr>
<tr>
<td>FLOW</td>
<td>MCNODEF</td>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td>FLOWOFF</td>
<td>MCNOWARN</td>
<td></td>
<td>REPEAT</td>
</tr>
<tr>
<td>IF</td>
<td>MCALTER</td>
<td></td>
<td>SHRT INT</td>
</tr>
<tr>
<td>INTEGER</td>
<td>MCLENG</td>
<td></td>
<td>SHRT INTEGER</td>
</tr>
<tr>
<td>INITTESTØN</td>
<td>MCSUB</td>
<td></td>
<td>TEST</td>
</tr>
<tr>
<td>INITTESTØFF</td>
<td>MCSET</td>
<td></td>
<td>VARIABLE</td>
</tr>
<tr>
<td>LØNG</td>
<td>MCPVAR</td>
<td></td>
<td>VALUE</td>
</tr>
<tr>
<td>REAL</td>
<td>MCGØ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following identifiers are defined as delimiters in system macros and should not be employed for naming new macros.

<table>
<thead>
<tr>
<th>BY</th>
<th>PI</th>
<th>OF</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>END</td>
<td>FOR</td>
<td>WHILE</td>
<td>;</td>
</tr>
</tbody>
</table>

Note: There is no automatic check for improper use of the above identifiers! A misuse of these identifiers may cause trouble, and a manual check is therefore recommended.
9.5.3 Reserved macro variables

EPS provides 30 permanent macro variables. This number can be increased by calling the macro MCPVAR.

The first ten variables, P1 to P10 may be freely used whereas the next twenty variables, namely, P11 to P30 are used by the system and should therefore not be used for other purposes.

9.5.4 Reserved registers

The general purpose registers of the IBM OS/360 Computer, specified below, are used for special purposes and it is therefore necessary to preserve their contents before using them for other purposes.

Register 1 is used as pointer to the memory area where the actual parameters of a called procedure are stored.

Register 10 is the base register for an auxiliary CSECT used by the system as storage area for subroutines and special variables.

Register 11 is the base register for the stack area allocated for the arrays declared in the current block/procedure.

Register 12 is used as base register for referencing within the code area, e.g. for calculation of addresses of branch instructions. It is required that each external procedure occupies less than 4096 bytes, and a call of an external procedure thus involves the assignment of the start address of the new procedure to register 12.

Register 13 is used as a pointer to the storage area for register saving.

Register 14 and Register 15 are employed for procedure call. Register 15 contains the entry address to the new procedure and Register 14 the return address. Note that our use of registers 1, 13, 14, 15 follow the common conventions for subroutine calls on the IBM 360/370 computers.
9.6.1 How to construct an arithmetic expression

Most of the common programming languages employ assignments, which involve the evaluation of arithmetic expressions. One way is to define a single arithmetic assignment macro for handling both the assignment and the evaluation of the arithmetic expression. A call of such a macro may, for instance, look like:

```
SET A-B + C * D;
```

The delimiter structure of this macro is:

```
SET = + * ;
```

An alternative way is to have two separate macros: An assignment macro with the delimiter structure:

```
SET = ;
```

and an arithmetic expression macro with the delimiter structure:

```
( += )
```

The arithmetic expression macro called \( ( \) will be employed as the second argument of the SET macro, e.g.

```
SET A = (B + C * D) ;
```

The latter way has a great advantage since the separate macro for evaluation of arithmetic expressions macro may be used not only in the assignment statements but also, in logical expression evaluation, subscript evaluation, etc.

Selected problems in the definition of the arithmetic expression macro and suggested solutions are listed below.

a. It is convenient to define left paranthnesis (as the arithmetic expression like

```
((( A * x + B) + C) * x + D)
```

The nesting mechanism of the ML/I is efficiently utilized. There are, however, also some difficulties in using (as a name for the arithmetic expression macro.
One problem is that the contents of registers or variables may be destroyed when passing, from one nesting level to another. This problem may be solved by saving these variables. One must distinguish carefully between the scopes of temporary and permanent macro variables and use them accordingly, otherwise macro time information may be destroyed.

b. If left parenthesis (is defined as the name of the arithmetic expression macro, undesired effects may sometimes result. E.g. the case of subscripted variables where the subscript is delimited by parenthesis, e.g. in the expression

\[(A(I+1) + B)\]

we want to treat the subscript expression \((I+1)\) separately and not as a nested part of the entire expression.

c. An arithmetic expression may be composed of variables of different types, e.g. real, integer, strings. The types of the operands may be retrieved from the macro time symbol table by the MCTYPE macro (see appendix), such that the assembly instructions that correspond to the types of the operand may be generated.

d. In most languages arithmetic expressions are defined such that some operators, for example *, has priority over +.

A simple way of implementing such a priority is to define intermediate macros in a scheme nearly identical to that employed for implementing arithmetic expressions using an operator precedence grammar.

The following example illustrates this technique. Consider the following syntax rules for an arithmetic expression:

\[\langle ae \rangle ::= \langle term \rangle | \langle term \rangle + \langle ae \rangle\]
\[\langle term \rangle ::= \langle var \rangle | \langle var \rangle + \langle term \rangle\]

Notice that this grammar doesn't include parenthesis. Moreover, let us assume that only real variables are considered.
Two macros which correspond to the two syntax rules will be defined:

1. MCDEF 4 VARS AE N1 OPT + N1 OR ; ALL
   AS & MCSET T4 = 1
         ! LE 2, = 'E0.'
   %L1. MCDEF & TEMP ? AS & TERM? %WAT4.;
   TEMP
   MCGO L2 IF P1 EN 0
   AER 2, 4
   MCGO L3
   %L2. AE 2,%AT4.
   %L3. MCSET T4 = T4+1
       MCGO L1 UNLESS T4 EN T1+1

2. MCDEF TERM N1 OPT N1 OR ; ALL
   AS & MCSET P1 = 0
   MCGO L0 F T1 EN 1
         ! LE 4,% AP1+1.
   %L1. MCSET P1 = P1+1
       ME 4,% AP1 + 1.
       MCGO L1 UNLESS P1 EN T1+1

The code generated for the arithmetic expression is AE X+Y*2 + W*P ;

LE 2, = 'E0.'
AE 2, X
LE 4, Y
ME 4, Z
AER 2, 4
LE 4, W
ME 4, P
AER 2, 4
The macro AE may be used in the definition of a macro that assigns the value of an arithmetic expression to a variable. The definition of such a macro, called SET, is shown below:

```
MCDEF SET = ;
AS & MCDEF & TEMP ?
    AS & &AE ? %WA2. ; ?
    TEMP
    ! STE 2, %A1.
    ?
```

An example of a call of this macro is:

```
SET A=B+Q*D;
```

### 9.6.2 How to construct a logical expression

The macro IF in EPS calls a macro named LOGEXP, which has to be defined by the user.

The user is free to define this macro to fit his special needs. A number of requirements must however be met:

1. The macro name has to be LOGEXP.
2. The closing delimiter has to be ;
3. Register 1 has to get eventually the value 1 for "true" and 0 for "false".

The following example illustrates a simple definition, of such a logical expression macro.

Assume for instance that the syntax rules for logical expression are:

```
<logical expression> ::= <arithmetic expression><op><arithmetic expression>
<op> ::= >| = |<
```
The LOGEXP macro satisfying these rules may be defined as:

```
MCDEF LOGEXP OPT OR = OR ALL;
AS & MCDEF & TEMP?
    AS & & AE? %WA1. ;?
    TEMP
    LER 6,2
MCDEF & TEMP?
    AS &&AE? %WA2.;?
    TEMP
    CER 6,2
    LA 1,1
    BC %DL., LXT2.
    LA 1,0
LXT2. NOP
?
MCDEF > AS 4
MCDEF =AS 8
MCDEF < AS 11
```

The generated code for the logical expression:

```
LOGEXP A = B ;
IS:
    LE 2,E'0.'
    AE 2, A
    LER 6, 2
    LE 2, = E'0.'
    AE 2, B
    CER 6, 2
    LA 1, 1
    BC 8, L84
    LA 1, 0
L84 NOP
```
9.6.3 How to define new data types

Two systems macros, VARIABLE and ARRAY are defined within EPS for data type declaration. The macro VARIABLE is used for defining simple variables and takes the form VARIABLE (ArgA, ArgB, ArgC)

ArgA is the numerical value that represents the value "undefined". In debugging mode (see chapter 8) this value is assigned to variables of this type upon declaration. This is a debugging aid which enables one to check at run time whether the variables when employed in computations have already been assigned values (if not their value is still "undefined"). A highly unlikely value should obviously be chosen for "undefined".

ArgB is a natural number to be used as numerical code that will represent the new data type in the macro time symbol table.

ArgC specifies the length in bytes that a variable of the defined type occupies. For example the macro INT is defined in the following way.

```
MCDEF INT N1 OPT, N1 OR; ALL
AS VARIABLE (F'2147483647',1,4)
```

The macro ARRAY is used for defining one dimension arrays. It takes the following form:

```
ARRAY (ArgA, ArgB, ArgD)
```

where

ArgA is a natural number to be used as the numerical code by which the new data type will be specified in the macro time symbol table.

ArgB specifies the number of bytes needed for storing of each element of the array.

ArgC is the digit 1.
Arg D is an assembler 360 constant which specifies the value "undefined" which is assigned to each element of the array upon its declaration. The value "undefined" is required for the debugging facility.

For example the macro INT ARRAY is defined in the following way.

```
MCDEF & INT ARRAY ? N1 OPT, N1 OR ; ALL
AS & ARRAY (1, 4, 1,F'2147483647')
```

9.6.4 Problems due to the layout of Assembly 360 code

The code generated must be of course in the format assumed by the assembler. To enable this a number of aids are provided,

a. Some constituents of the assembly language such as labels and comments must begin in column 1. Continuation character must be placed in column 72 etc. Moreover, it is recommended for sake of clarity to write all assembly instructions in a fixed format namely - operation codes starting in column 10, operands in column 16.

These requirements may be satisfied by employing the character ! Any appearance of this character in the source text will cause the generated value text to begin in column 1. The character itself will not appear in the value text. Consider the following example:

```
MCDEF ADD,; AS &!* ADDITION
      L 4,%A1
      A 4,%A2
      ST 4,%A3
```

Note that ! before the *, ensures that the * will be positioned in column 1 as required by the Assembler.
b. The assembly language prohibits blank lines in programs. The EPS programmer must keep this peculiarity in mind, and avoid casual appearance of such lines in value text.

Consider the "innocent" case:

```
MCNODEF
MCGO L15
```

It will cause the generation of a blank line because NL is not the closing delimiter of the MCNODEF macro so a skip to the next line is copied, and a blank line is thus generated. The sequence

```
MCNODEF MCGO L15
```

will on the other hand avoid this phenomenon.

c. Assembly code may include undesired calls for macros. For example, the assembly instruction

```
L 5, 0(1)
```

calls to the arithmetic expression macro "(\(\), if such a macro has been defined, which will usually be the case. To prevent such undesired calls it is recommended to delimit each piece of assembly code in the value text by literal brackets. e.g.

```
& L 5,0(1)
```

9.6.5 Some timing considerations

In order to minimize compile time the following considerations have to be taken. The greatest time-consumers in EPS are MCGO and MCSET, and the number of calls of these macros should be reduced. Long distance forward MCGO's are particularly expensive and should be avoided. It is in some cases possible to avoid MCGO's by building extra alternatives into delimiter structures.
Thus

\[
\text{MCDEF } x \text{ WITHS } Y \text{ WITHS } NL \text{ AS } \& \text{ +++ SPECIAL+++?}
\]

\[
\text{MCDEF } x \text{ NL AS } \& \text{ +++ } \%A1.+++?
\]

is better than

\[
\text{MCDEF } X \text{ NL AS } \& \text{ MCGO } \text{ L1 IF} \%A1.= \text{Y}
\]

\[
\text{+++ } \%A1. \text{ +++ MCGO } \text{ LO}
\]

\[
\%L1. \text{ +++ SPECIAL +++ ?}
\]

Long sequences of MCGO's which test an argument or delimiter against several possible alternatives can be avoided by using the technique described in Section 9.4.2.

The use of local NEC Macros can also save time. For example

\[
\text{MCDEF } X \text{ NL AS } \& \text{ MCDEF } Y \text{ AS SPECIAL }
\]

\[
\text{+++ } \%A1.+++
\]

avoids using MCGO's to test if argument one contains the atom Y, which is to be replaced by SPECIAL. In this case % would need to be defined as an unprotected insert. The use of local NEC macros, however, has its dangers, particularly when several different arguments are to be inserted. One safeguard is to have two insert definitions, % and %U, say, where only one of them is unprotected. The unprotected one should be used only when local NEC macros are to apply.
9.7 How to run a program in EPS (JCL Instructions)

A program in EPS is executed in four steps:

1. Processing by the EPS macro processor
2. Compilation by the 360 Assembler
3. Link editing by the OS/360 linkage editor
4. Execution

The first step may be executed either in one pass or in two passes. The latter is needed in the early debugging stages. There are five catalogued procedures to handle these steps.

- **EPSP2** Process the source program in two passes. The first pass is needed only to check for matching of formal and actual parameters. The final product is an assembly program left in a data set specified by the user.

- **EPSP1** Process the source program in one pass, without checking for matching of parameters. The final product is the same as in EPSP2.

- **EPSPC** Process (in one pass) and compile the source program. The final product is an object module left in a data set specified by the user.

- **EPSPCL** Process (in one pass), compile and link with other object modules the source program. The final product is a load module left in a data set specified by the user.

- **EPSPCLG** Process (in one pass), compile, link and execute (go) the source program.

The following sections describe how to use each procedure.
9.7.1 EPSP2

// EXEC EPSP2

// GENER.PROGRAM DD *

{ source program

/*

// EPS. PRODUCT DD dsname

dasename is the data set name in which the assembly language program will reside.

9.7.2 EPSP1

// EXEC EPSP1

// GENER.PROGRAM DD *

{ source program

/*

// EPS. PRODUCT DD dsname

dasename is the data set name in which the assembly language program will reside.

9.7.3 EPSPC

// EXEC EPSPC [LST = LIST]

// GENER.PROGRAM DD *

{ source program

/*

// ASM.OBJECT DD dsname

dsname is the data set name in which the object module will reside. LST is a parameter which determines whether we want to obtain the assembly language program.

9.7.4 EPSPCL

// EXEC EPSPCL [LST = LIST]
// GENER.PROGRAM DD *

{source program

/*

//LKED.SYSIN DD dsname 1
//LKED.LDM DD dsname 2

dsname 1 is the data set name to be linked with the produced object module. If it doesn't exist a DUMMY dd card is required.

dsname 2 is the data set name in which the load module will reside.

9.7.5 EPS PCLG

// EXEC EPSPCLG [LST = LIST]
// GENER.PROGRAM DD *

{source program

/*

//LKED.SYSIN DD dsname

dsname 1 is defined as in EPSPCL.
Chapter 10 The macro library

10.1 Introduction

The macro library contains macros for evaluation of arithmetic expressions, logical expressions and character string expression. There are, furthermore, macros for I/O. The logical expression is, of course, designed such that it may be used in the IF and WHILE constructs of EPS.

10.2 The ARITEXP macro

**Purpose**

To evaluate an arithmetic expression

**General form**

ARITEXP Argument A;

Argument A is an arithmetic expression written in accordance with the following BNF rules:

\[
E = T | E + T | E - T | T \times T \\
T = F | F \times T | F \div T | F \text{ REM } T | F \div T \\
F = I | (E) \\
I = \text{identifier} | \text{constant} | \text{identifier (E)}
\]

where the non-terminal symbols are:

- E-expression
- T-term
- F-factor
- I-simple variable or a single array component

Identifiers are written in accordance with EPS rules, while constants are written in accordance with Assembly 360/370 rules.

**Restrictions**

1. This macro applies only to integer and real variables and constants.
2. REM and DIV apply only to integer expression.
3. The subscript of a subscripted variable (Identifier (E)) must be a positive non-zero integer expression.

**Examples**

(a) ARITEXP A+B*C;
(b) ARITEXP 5.3*(A-2);
(c) ARITEXP I;
(d) ARITEXP -5;
(e) ARITEXP A+R(J+K DIV M);
(f) ARITEXP S(1);
1. Code is generated for evaluation of the arithmetic expression. If the result is an integer it is left in register 1, and if it is floating point number it is left in floating point register 2. The contents of all registers are destroyed during the evaluation.

2. It is assumed that the subscript of the first component of an array is 1, e.g. the variable A(3) is the third element of the array A.

3. The following table illustrates the automatic type conversions which take place during the evaluation of an arithmetic expression.

<table>
<thead>
<tr>
<th>op (operator)</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>DIV</th>
<th>REM</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT op INT</td>
<td>INT</td>
<td>INT</td>
<td>INT</td>
<td>INT</td>
<td>INT</td>
<td>INT</td>
</tr>
<tr>
<td>INT op REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>REAL op REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
<tr>
<td>REAL op REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>REAL</td>
<td>ERROR</td>
<td>ERROR</td>
</tr>
</tbody>
</table>

4. The usual hierarchy of operators is employed, i.e. the operations *, / REM and DIV are performed before the operations + and -. Consecutive operators with same priority are executed from left to right.

10.3 The LOGEXP macro

Purpose To evaluate a logical expression

General form

\[
\text{LOGEXP} \text{ Argument A ; (\text{< or > or = or \text{<= or >=}}) \text{ Argument B;}}
\]

Arguments A and B are arithmetic expressions as defined in Section 10.2.
Examples  (a) LOGEXP A+B C*F;
(b) LOGEXP 5 = A;

System action Code is generated for evaluation of the logical expression.
The value 1 is left in register 1 if the value of the logical expression is true, and 0 otherwise. The contents of all registers is destroyed during the evaluation.

Restrictions The types of the arithmetic expression on both sides of the relational operation must be the same. The type of an arithmetic expression is the type of the result of its evaluation. The two expressions may therefore have the same type even in cases where they are composed of operands of different types.

10.4 The SET macro
Purpose to assign the value of an arithmetic expression to a variable.

General form SET Argument A = Argument B;
1. Argument A is a variable (Subscripted or nonsubscripted)
2. Argument B is an arithmetic expression as defined in section 10.2.

Examples (a) SET A = B + 5.3;
(b) SET R(J+4) = B(I) * C /5.1;

System action Code is generated for evaluation of the arithmetic expression in the right hand side of the equal sign (by calling the ARITEXP macro) and assigning the result to the variable on the left hand side (argument A). The type of the assigned value will be the same as the type of the variable on the left hand side. Automatic conversion is performed if necessary.

10.5 The SETC macro
Purpose To evaluate a string expression and to assign its value to a string variable.
**General form**

SETC Argument A [/Argument B?] = Argument C;

1. Argument A is the name of the string variable to which the value of the expression is assigned.

2. Argument B is an integer variable or an integer constant giving the number of the characters in Argument A, into which the first character of the string expression is inserted.

3. Argument C is a string expression defined as follows:
   
   \[
   \text{Argument D} \ [\text{+Argument D} \ * \ ?]
   \]

   where + is the concatenation operator and Argument D is either a 'string' or identifier [/I1/I2/].

   identifier is the name of the string variable and I1,I2 are integer variables or constants. I1 and I2 specify a substring of the string in identifier. I1 is the number of the characters where the substring begins and I2 is the length of the substring. The numbering of characters in a string is 1 for the first character, etc.

**Examples**

(a) SETC A=B/3/2 + C+'KPL';

(b) SETC x/3=D/2/1 + 'I AM HERE';

**System action**

Generates code for evaluation of the string expression (Argument C) and assigning the result to the string variable specified in the left hand side of the equal sign (Argument A).

To illustrate the actions let us assume that the initial values of the variables in the examples (a) and (b) above are:
10.6 The WRITE macro

Purpose
To write a line of values.

General form
WRITE Argument A [,Argument.B * ?];
Arguments A and B are either variables (integer, real or string type) or a string constant.

Examples
(a) WRITE A;
(b) WRITE I, J, '53.4', 'K=', B;
Notice that numerical constants are written as string constants. The apostrophes are omitted in the printing.

System action
Generates code for writing the values of the variables and constants which are listed. The values are printed consequently with spaces between them. Integers are printed in an integer format (similar to format I in FORTRAN).
Real values which range between + 10 E - 3 to + 10 E + 3 are printed in a fixed point format (similar to format F in FORTRAN), while greater or smaller real values are written in a floating point format (similar to format E in FORTRAN).
The output file name is LINEOUT.
10.7 The READ macro

Purpose
To read a line of values.

General form
READ Argument A [,Argument B * ? ];
Arguments A and B are either previously defined integer variables or real variables or character string variables.

Examples
READ A, SQR, B ;

System action
Generates code for reading values from records (e.g. cards).

The format in the card is free i.e. at least one blank between each value with the exception that before a character string there must be one and only one blank (to allow a character string to begin with blanks) e.g.: CHAR Q(4), R(4);
READ Q,R;

Suppose that the card is punched as:
bABCbbDEFb
After the READ is executed Q='bABC' R='bDEF'.

If a string starts in a new card, it must begin in the first column, however integers and reals may start in any column and after as many blanks as one wishes.
If all the values which are punched on the card have been read and there are more variables in the variable list of the READ macro, the next card will be read until all the variables have been read.
Real numbers may be punched only in a fixed-point format and may not be punched in a floating-point format.

The input file name is CARDIN.
APPENDIX

1. Additional operation macros.

Some operation macros have been added to ML/1 for debugging, and manipulation of the macro-time symbol and parameter tables.

The macro-time symbol table is needed to manipulate different data types in macro-time and to facilitate a macro-time detection of undefined symbols or symbols which have been defined twice. The parameter table is used for detection of mismatching between types and number of formal and actual parameters of procedures.

In addition to the local and global environment another global environment has been created for the needs of the debugging facilities.

For the sake of clarity, the original global environment will be called "main global environment" and the new global environment will be called the "auxiliary global environment".

Only one global environment can be in force at one time. The switching between these two global environments is done by the macros MCG and MCNOG discussed in 1.9. The environment which is first in force is the main global environment. The initial contents of both the main and the auxiliary global environment are merely the operation macros.

1.1 The macro-time symbol table

The macro-time symbol table takes the following form:

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity name</td>
<td>Type attribute</td>
<td>Dimension attribute</td>
<td>Trace attribute</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
The symbol table is composed of four "columns" and 50 rows. The number of rows can be changed by the macro MCSPACE.

The first column contains the entity name which must be an identifier. In EPS it is used for names of variables, arrays, procedures, blocks, cases and compounds. Names may, however, be used for other purposes in macros defined by the programmer.

Columns 2,3,4 contain positive integer numbers. The "type attribute" is used for data type specification or for procedure, block, compound and case identification. The dimension attribute is used for dimension specification in case of variables and arrays. The trace attribute is used for tagging variables to be traced and is therefore only used for variables and arrays.

1.2 The macro-time parameter table

The macro-time parameter table is similar to the symbol table and takes the following form:

<table>
<thead>
<tr>
<th>entity name</th>
<th>entity attribute</th>
<th>dimension attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

The entity name may be either a formal parameter name or a procedure name. The type attribute specifies the parameter or the procedure type. The dimension attribute specifies for arrays the number of dimensions and in the case of procedure names it contains the number of parameters in the procedure.
1.3 The MCENTRY macro

Purpose
Entering the entity name and its attributes into the macro-time symbol table.

General form
MCENTRY (Argument A, Argument B, Argument C)
1. Argument A is the entity name.
2. Argument B is the type attribute.
3. Argument C is the dimension attribute.

Examples
(a) MCENTRY (A,3,1)
(b) MCENTRY (BB1, 8,0)

System action
Inserts in the macro-time symbol table, the code specifying its type (the type attribute), and the code specifying the number of dimensions of the variable (the "array number").

1.4 The MCTRACE macro

Purpose
Specification of variables to be traced.

General form
MCTRACE (Argument A, Argument B)
1. Argument A is the name of a variable whose value we want to start or stop tracing.
2. Argument B is the "trace tag". It is the number 1 when we want to start tracing, and the number 0 when we wish to stop the tracing.

Examples
(a) MCTRACE (A,1)
(b) MCTRACE (B,0)

System action
Inserts a "trace tag" in the macro-time symbol table on the variable specified as Argument A. The trace tag "on" or "off" is represented in the table by the number 1 or 0 respectively in the trace attribute column.
1.5 The MCTYPE, MCDIME and MCTRACETAG macros

**Purpose** Retrieving the value of any attribute from the macro-time symbol table.

**General form**
1. MCTYPE |Argument A|
2. MCDIME |Argument A|
3. MCTRACETAG |Argument A|

Argument A is an entity name.

**Examples**
(a) MCTYPE |A|
(b) MCDIME |B|
(c) MCTRACETAG |C|

**System action**
These macros belong to the system function class. They return values like the MCSUB and the MCLENG macros. The values of the three macros are:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTYPE</td>
<td>Type attribute of the argument</td>
</tr>
<tr>
<td>MCDIME</td>
<td>Dimension attribute of the argument</td>
</tr>
<tr>
<td>MCTRACETAG</td>
<td>Trace attribute of the argument</td>
</tr>
</tbody>
</table>

If the entity name is not found in the table, an error message is produced and a value 0 is returned.

1.6 The MCCLSMB macro

**Purpose** Clearing the macro-time symbol table.

**General form** MCCLSMB
1.7 The MCPREENTRY macro

Purpose
Entering a parameter or procedure name and its attributes into the parameter table.

General form
MCPREENTRY (Argument A, Argument B, Argument C)
1. Argument A is a formal parameter or a procedure name.
2. Argument B is the parameter or procedure type.
3. Argument C is either the number of dimensions in an array or the number of the parameters in a procedure.

Examples
(a) MCPREENTRY (A, 1, 0)
(b) MCPREENTRY (B, 3, 1)
(c) MCPREENTRY (KUKU, 6, 13)

System action
Inserts in the macro-time parameter table the code specifying the type and dimension of a formal parameter. In the case of a procedure name its type and the number of its parameters are inserted instead.

1.8 The MCPRTYPE macro

Purpose
Retrieving the type of a parameter of a procedure from the parameter table.

General form
MCPRTYPE (Argument A, Argument B)
1. Argument A is either a parameter or a procedure name.
2. Argument B is a positive integer number.

Examples
(a) MCPRTYPE D,0
(b) MCPRTYPE E,7

System action
This macro belongs to the system function class and it returns a value. If Argument B is zero then the returned value is the type of the name specified as argument A. If one wishes to get the type of the name which stands N places before the name specified as argument A, argument B must take the value N. Consider the following parameter table:
<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

These values will be returned by the following calls of the MCPRTYPE macros:

- MCPRTYPE [A,0] = 6
- MCPRTYPE [A,1] = 1
- MCPRTYPE [A,2] = 2
- MCPRTYPE [A,3] = 3

1.9 The MCPRDIME macro

**Purpose**
Retrieving the number of dimensions in a parameter or the number of procedure parameters, from the parameter table.

**General form**
MCPRDIME [Argument A]

**Examples**
MCPRDIME [A]

**System action**
Returning the value of the number of dimensions of a parameter or the number of procedure parameters.

1.10 The MCPRM and MCCLPRM macros

**Purpose**
Inserting a parameter in the parameter list and emptying the parameter list.

**General form**
1. MCPRM [Argument A]
2. MCCLPRM

**Examples**
(a) MCPRM [A]
(b) MCCLPRM
System action  The macro MCPRM adds a parameter to the parameter list and checks whether it has been previously entered into this list. If it has, an error message is produced. Each parameter has to be unique in its procedure. In order to detect parameters of the same name in the same procedure, a parameter list is constructed for each procedure. The macros PROCEX or PROC calls the macro MCPRM for each parameter in order to enter it into the parameter list. When the evaluation of the macros PROCEX or PROC is completed the parameter list is no longer needed and it is cleared out by calling the macro MCCLPRM.

1.11 The MCNOG and MCG macros

Purpose  Switching between the two global environment.
General form  
MCNOG  
MCG
System action  The macro MCNOG causes the auxiliary environment to be in force whereas the MCG macro re-establishes the main global environment.
Each global environment constituent (macro, insert, skip) definition is added to the global environment which is currently in force. No information is destroyed by switching from one global environment to another.

1.12 The MCDEBUG and MCNODEBUG macros

Purpose  Specifying whether we wish to run the program in "debugging mode" or in "production mode".
General form  
1. MCDEBUG  
2. MCNODEBUG
System action  Assigns the value one or zero respectively, to a location named BUGPT in ML/1.
1.13 The MCMSG macro

**Purpose**
Generating user's own debugging messages.

**General form**
MCMSG (Argument A)

Argument A is the message to be printed.

**Examples**
MCMSG (MULTIPLE DEFINED PARAMETER) will produce the message:
MULTIPLE DEFINED PARAMETER
FOUND IN LINE line number

**System action**
This macro differs from the MCNOTE.
In this macro only the sequence line number is produced as shown in the example, whereas in the MCNOTE macro additional information is produced (see section 5.4.2). The messages are transferred to list 2 which is the 'debugging file name'.

1.14 The MCLINE macro

**Purpose**
Getting the line number value.

**General form**
MCLINE

**System action**
This macro belongs to the system function class and it returns the value of the current line number of the source text.

1.15 The MCLIST and MCNOLIST macros

**Purpose**
Controlling the option of obtaining source program listing.

**General form**
MCLIST
MCNOLIST

**System action**
An appearance of the macros MCLIST and MCNOLIST anywhere in the program causes the starting or stopping respectively of the listing.
These two macros enable the programmer to get a complete listing of the source program as well as a listing of any program segments. Obviously, the programmer can avoid
listing completely. In EPS the default option is MCLIST, i.e. source program listing is produced and transferred to list 3 which is the "listings file name". If listings are not needed the program is started with a call for MCNOLIST.

Examples

(a) MCNOLIST

(b) MCNOLIST

... MCNOLIST

... MCLIST

1.16 The MCTEXT and MCNOTEXT macros

**Purpose**
Control the production of a value text

**General form**
MCTEXT
MCNOTEXT

**Examples**

(a) MCNOTEXT

(b) MCNOTEXT

... MCNOTEXT

... MCTEXT

**System action**
These macros may be called anywhere in the source program.
MCNOTEXT is called when we wish only to check the source program but not use CPU time on object code production. It is advisable to call MCNOTEXT when bugs are expected, e.g. when running a new program for the first time. MCNOTEXT is automatically called when a bug is detected. The amount of saving in CPU time by calling MCNOTEXT can be considerable.
1.17 The MCERFLAG macro

**Purpose**
To check if any syntax errors were detected

**General form**
MCERFLAG

**System action**
The MCERFLAG returns the value 1 when at least one syntax error has been detected by ML/l in the source program otherwise the value zero is returned.

1.18 The MCSTOP macro

**Purpose**
To terminate EPS processing

**General form**
MCSTOP

**System action**
The MCSTOP macro causes ML/l to stop processing.

2. Physical forms of source and output text and initial state of name environment

The physical forms of source and output text are defined by the user, and are explained in chapter 9.

The logical record length must be 80. Only the first 72 bytes are significant whereas the last 8 bytes are disregarded.

At the initial state of the local name environment no name is included. The initial state of the main and the auxiliary global name environments is the set of the operation macros.

3. Set of characters

The character set of EPS contains a primary (upper case) set of letters A-Z, the numbers 0-9, the character # which will be also called "letter" and other characters: `. , <,+,!,&,$,*,;,\-,\,\,\,\,\,\,\,\,\,\,-,?>,:.@','="

Programmers should be warned that EPS has adopted the convention of using # as
the first character of symbols. To avoid collisions with EPS the programmer is urged to avoid using this symbol.

EPS utilizes several punctuation characters, for the purposes listed below:

<table>
<thead>
<tr>
<th>punctuation character</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>New line representation (NL)</td>
</tr>
<tr>
<td>!</td>
<td>Start line representation</td>
</tr>
<tr>
<td>@</td>
<td>Stop code, termination of processing</td>
</tr>
</tbody>
</table>

4. **The Debugging file**

Error messages can take any physical form defined by the user. In most cases the error messages will be filled for line printer listings.

The name of this file is LIST 2. For details see chapter 9.

5. **System variables**

In the implementation of EPS no system variables were utilized.

6. **Extra keywords**

No extra keywords were implemented in EPS.

7. **Permanent variables**

30 (thirty) permanent variables were initially allocated in EPS. The programmer may increase this number by using the MCPVAR macro. The maximum value of a permanent variable may not exceed $2^{31} - 1$. 
<table>
<thead>
<tr>
<th>Error message</th>
<th>section no.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARITHMETIC OVERFLOW</td>
<td>8.3.2</td>
</tr>
<tr>
<td>ARGUMENT number HAS ILLEGAL VALUE, VIZ &quot;value&quot;</td>
<td>8.3.6</td>
</tr>
<tr>
<td>BRANCH TO END OF procedure name</td>
<td>8.11</td>
</tr>
<tr>
<td>BEGIN name HAS NOT BEEN TERMINATED BY END name;</td>
<td>8.5.1</td>
</tr>
<tr>
<td>CASE name HAS NOT BEEN TERMINATED BY END name;</td>
<td>8.5.1</td>
</tr>
<tr>
<td>DELIMITER name .... OF (SKIP) name .... NOT FOUND (INSERT)</td>
<td>8.3.5</td>
</tr>
<tr>
<td>END name; HAS BEEN FOUND NOT IN PLACE OR APPEARS MORE THAN ONCE</td>
<td>8.5.2</td>
</tr>
<tr>
<td>ENTRY TO BLOCK block name</td>
<td>8.11</td>
</tr>
<tr>
<td>ELSE PART OF IF IS EXECUTED</td>
<td>8.11</td>
</tr>
<tr>
<td>flag number IS ILLEGAL MACRO ELEMENT</td>
<td>8.3.1</td>
</tr>
<tr>
<td>ILLEGAL INPUT CHARACTER</td>
<td>8.3.3</td>
</tr>
<tr>
<td>ILLEGAL MACRO NAME AFTER WARNING, VIZ &quot;atom&quot;</td>
<td>8.3.4</td>
</tr>
<tr>
<td>INSERT name ABORTED DUE TO ERROR</td>
<td>8.3.4</td>
</tr>
<tr>
<td>LABEL number IS MULTIPLY-DEFINED</td>
<td>8.3.7</td>
</tr>
<tr>
<td>LABEL number REFERENCED IN LINE number OF... NOT FOUND</td>
<td>8.3.8</td>
</tr>
<tr>
<td>LOOP loop name IS EXECUTED number TIMES</td>
<td>8.11</td>
</tr>
<tr>
<td>MACRO name ABORTED DUE TO ABOVE ERROR</td>
<td>8.3.11</td>
</tr>
<tr>
<td>name MAY NOT APPEAR AS THE LENGTH OF A DECLARED ARRAY</td>
<td>8.5.7</td>
</tr>
<tr>
<td>name IS A MULTIPLE DEFINED PROCEDURE NAME</td>
<td>8.5.8</td>
</tr>
<tr>
<td>name IS AN ILLEGAL PARAMETER</td>
<td>8.5.5</td>
</tr>
<tr>
<td>name IS AN ILLEGAL IDENTIFIER</td>
<td>8.4.1</td>
</tr>
<tr>
<td>name is MULTIPLY DEFINED</td>
<td>8.4.2</td>
</tr>
<tr>
<td>name HAS NOT BEEN DECLARED</td>
<td>8.4.3</td>
</tr>
<tr>
<td>Error messages</td>
<td>section no.</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>name</strong> is MULTIPLY DEFINED</td>
<td></td>
</tr>
<tr>
<td>POSSIBLE CAUSES:</td>
<td></td>
</tr>
<tr>
<td>1) TWO PROCEDURE PARAMETERS HAVE THE SAME NAME</td>
<td></td>
</tr>
<tr>
<td>2) A PARAMETER AND A VARIABLE HAVE THE SAME NAME</td>
<td></td>
</tr>
<tr>
<td>3) TWO VARIABLES HAVE THE SAME NAME</td>
<td></td>
</tr>
<tr>
<td>DETECTED IN PROCEDURE <strong>procedure name</strong></td>
<td>8.4.4</td>
</tr>
<tr>
<td><strong>P number</strong> IS ILLEGAL MACRO ELEMENT</td>
<td>8.3.1</td>
</tr>
<tr>
<td>PROCESS ABORTED FOR LACK OF STORAGE</td>
<td>8.3.9</td>
</tr>
<tr>
<td>PROCEDURE <strong>name</strong> HAS NOT BEEN TERMINATED BY END <strong>name</strong>;</td>
<td>8.5.1</td>
</tr>
<tr>
<td>PROCEDURE <strong>name</strong> HAS NOT BEEN TERMINATED BY END <strong>name</strong>;</td>
<td>8.5.1</td>
</tr>
<tr>
<td>PROCEDURE <strong>name</strong> IS CALLED PROCEDURE OR BLOCK OR CASE OR REPEAT NAMED <strong>name</strong> IS NOT FOUND</td>
<td>8.5.3</td>
</tr>
<tr>
<td>REPEAT <strong>name</strong> HAS NOT BEEN TERMINATED BY END <strong>name</strong>;</td>
<td>8.5.1</td>
</tr>
<tr>
<td>SEGMENT <strong>number</strong> OF CASE <strong>name</strong> IS EXECUTED</td>
<td>8.11</td>
</tr>
<tr>
<td>TYPES OF FORMAL AND ACTUAL PARAMETERS OF PROCEDURE <strong>name</strong> DO NOT MATCH</td>
<td>8.5.4</td>
</tr>
<tr>
<td>UNLABELED PROCEDURE</td>
<td>8.5.6</td>
</tr>
</tbody>
</table>
INDEX

Activation of the operating system ,7/15
Argument ,2/3
ARITEXP ,10/1
Arithmetic expression ,9/20, 9/21, 9/22, 9/23
Arithmetic overflow ,8/3
Atom ,2/2
BEGIN ,7/1, 7/2, 7/3
Block ,7/1, 7/7, 8/8, 8/9
CALL ,7/10
Call by name (ML/1) ,3/1
Capacity ,2/7, 5/14, 5/15
Case ,7/12, 7/13
CHARACTER ,6/7, 6/8
CHAR ,6/7, 6/18
Character set ,2/1, 8/3, 9/17, A/10, A/11
CHECK ,8/11, 8/13
CHECKOFF ,8/11, 8/13
Closing delimiter ,2/3
COMMENT ,7/14
Common Compiler ,1/1, 1/2
Common Macros Classes ,1/2
Consequences of evaluation ,5/4, 5/5
[Constituent?] ,1/2
[Constituent *] ,1/2
[Constituent *?] ,1/2
Constructions ,2/22
Data stack ,7/3
Debugging mode ,8/11, 8/12, 8/13
DEBUG ,8/11, 8/12, 8/13
DEBUGOFF ,8/11, 8/12, 8/13
Definition name ,4/2
Deleting a macro ,9/5
Delimiter ,2/3
  examples of delimiters 2/4
Delimiter structure ,2/5, 2/6, 5/2, 5/3, 5/4, 5/5, 5/6, 5/7, 8/4
  Optional delimiter ,2/6, 2/7
  Repeated delimiter ,2/6, 2/7
  Searching for delimiter ,3/4
  Exclusive delimiters ,3/4, 3/5
Description of operation macros ,5/1
Dynamically generated construction ,3/6, 9/9

Environment ,2/1
Exclusive delimiters ,3/4, 3/5
Execution time ,2/9
Extensible language ,1/1
External procedure ,7/1, 7/2, 7/3
EXIT ,7/10, 7/11, 7/13
Evaluating inserted text ,3/1
Evaluating replacement text ,3/1
Evaluating source text ,3/1

FLOW ,8/11, 8/14
FLOWOFF ,8/11, 8/14, 8/15
FOR ,7/10, 7/11
Free mode ,2/21
Global macros ,4/3, 4/4

Identifier ,6/1, 8/7, 9/17
INIT ,8/16
INITOFF ,8/16
IF THEN ELSE ,7/14, 7/15
INSERT ,2/9, 2/10, 2/11, 2/12, 2/13, 2/14, 2/15, 2/16, 2/17
Insert definition ,2/13
Insert name ,2/13
Insert flag ,2/14
Inserted text ,2/14
Examples of inserts ,2/15, 2/16, 2/17
Integer (in ML/1) ,1/2
Integer overflow (in ML/1) ,2/12
INT ,6/1
INTEGER ,6/1
INT ARRAY ,6/4
INTEGER ARRAY ,6/4
Internal procedure ,7/7, 7/3

Jumping over expanded code ,9/1

Keywords ,5/3

Layout characters ,5/3
Layout keywords ,5/3
Literal brackets ,2/10, 4/2
Local macros ,4/3, 4/4
LOGEXP ,10/3
Logical expression ,9/23, 9/24
LONG REAL ,6/3
LONG REAL ARRAY ,6/7
Lower case letters ,9/2
Macro name, 2/3, 8/3
Macro call, 2/3, 9/6
Macro name, 2/7, 4/7, 4/8, 9/19
Macro time variables, 2/9, 2/10, 9/17
  Permanent variables
  System variables
  Temporary variables
  Initialization values, 2/11
Macro time loop, 2/9, 9/7
Macro expression, 2/11
  Result of macro expression, 2/12
Macro labels, 2/12, 8/5
Macro elements, 2/13, 8/3
Macro time symbol table, A/1, A/2
Macro time parameter table, A/2
MAIN PROCEX, 7/14, 7/15, 7/6, 7/7
Matched skip, 2/19, 2/20, 5/13
  Examples of matched skip, 2/20, 2/21
Macro redefinition, 4/7, 4/8
MCALTER, 5/18
MCCLPRM, A/6, A/7
MCCLSMB, A/4
MCDEBUG, A/7
MCDEF, 5/14
MCDIME, A/4
MCENTRY, A/3
MCRERFLAG, A/10
MCFOR, 9/7, 9/8
MCG, A/7
MCGO, 5/25, 5/26, 5/27, 5/28
MCINS, 5/12
MCLEN, 5/20
MCLINE ,A/8
MCLIST ,A/8, A/9
MCMSG ,A/8
MCNOLIST ,A/9, A/10
MCNODEBU ,A/7
MCNODEF ,5/6
MCNOG ,A/7
MCNOINS ,5/6
MCNOSKIP ,5/6
MCNOTEXT ,A/9
MCNOTE ,5/24
MCNOWARN ,5/6
MCPRETR ,A/5
MCPRTYPE ,A/5, A/6
MCPRDIME ,A/6
MCPRM ,A/6
MCPVAR ,5/29
MCSKIP ,5/13
MCSUB ,5/21, 5/22
MCSET ,5/23
MCSTOP ,A/10
MCTEXT ,A/9
MCWARN ,5/11
REPEAT 7/7, 7/8, 8/8, 8/9
Repeated delimiters 2/6, 2/7
Replacement text 2/3
Reserved characters 9/17, 9/18
Reserved entities 9/17, 9/18
Reserved identifiers 9/17, 9/18

Scanned text 2/1
Scanning process 3/2
Scope of names 7/3
Searching for delimiters 3/3
Secondary delimiter 2/3
SET 10/4
SETC 10/4
SHORT INT 6/2
SHORT INT ARRAY 6/5, 6/6
SHORT INTEGER 6/2
SHORT INTEGER ARRAY 6/5, 6/6
Skip name 2/18
Matched skip 2/18, 2/19, 2/20
Straight skip 2/18, 2/19, 2/20
Skips 2/17, 2/18
SPACE 1/2, 9/3
Straight scan 2/7, 2/22
Structure representation 5/8, 5/9, 5/10, 5/11
Structured programming 1/1
Subscript (in ML/1) 2/11
Substitution macros 4/1
Syntactic errors 8/1
System function 4/1, 5/20
System variables 2/10