80-009

ML/I Macro Processor: Implementation in PL/I

by
Jeffrey D. Liotta

A Thesis submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1980

Approved by:

Peter Calingaert
Adviser

Gyula Magó
Reader

Stephen M. Pizer
Reader

JEFFREY DEAN LIOTTA
ML/I Macro Processor: Implementation in PL/I
(Under the direction of DR. PETER CALINGAERT.)

ABSTRACT

ML/I is a general purpose macro language which provides for the user specification of the argument delimiters of a macro, where each macro may have several possible patterns of delimiters. Other language features include nested and recursive macro calls and definitions, macro-time assignment and conditional statements, an extensive macro-time environment, the specification of a possibly variable number of delimiters for each macro, and a method of defining multi-atom macro names and delimiters. This particular implementation runs under OS/360 MVT Release 21.8 - Hasp II version 3.1 on the IBM 360/370.

TABLE OF CONTENTS

I. PREFACE 1

II. INTRODUCTION 3

2.1. Description of Syntax 3
2.2. Definitions 4
2.3. Differences Between Macro Processors and Macro Assemblers . . 10
2.4. Special Features of ML/I 11

III. LANGUAGE FEATURES AND THEIR EFFECT ON SYSTEM DESIGN 13

3.1. The Specification of Construction Syntax 13
3.1.1 Fixed Delimiters 13
3.1.2 Option Lists 16
3.1.3 Nodeplaces and Nodegos 18
3.2. Local Symbol Table Manipulation 20
3.2.1 Nesting of Definitions 20
3.2.2 Deleting Local Symbols 22
3.2.3 Organization of the Local Symbol Table 23
3.3. Unbounded Look-Ahead 23
3.4. Lexical Analyzer 24

IV. CONCLUSION 27

4.1. Uses of ML/I 27
4.2. Adaptability of a General-Purpose Macro Processor 28

V. APPENDICES. 29

BIBLIOGRAPHY. 31

LIST OF FIGURES

Figure 1: A Structure with Fixed Delimiters 15

Figure 2: A Structure containing an Option List 17

Figure 3: Structure with an Option List, Nodeplaces, and Nodegos . . 19

1. PREFACE

The purpose of this thesis project was to write an implementation of ML/I, a conditional macro language designed by P.J. Brown [1] supporting both nested and recursive macro calls. Creating a processor for this high-level language provided an opportunity to gain valuable insights into the implementation and design of a general class of translation programs, of which macro processors are but a member. In fact, the implementation of ML/I's translator required many techniques more commonly associated with compilers than with macro processors.

This document describes several of the unusual characteristics of the language and some of the problems which they created during the implementation of the processor. I have focused the discussion on those features which reflect the general-purpose nature of ML/I and directly relate to certain problems which arise when implementing other types of translators.

Material accompanying this document includes an implementation of ML/I to be run on OS/360, a logic manual describing the implementation, and a reference manual describing the basic features and syntax of the macro language.

2. INTRODUCTION

Before examining ML/I, it is necessary to define some of the terms and concepts discussed herein.

2.1 DESCRIPTION OF SYNTAX

The syntactic representations used in this document were borrowed from a notation proposed by Niklaus Wirth [2]. The main characteristics of this notation are as follows:

(a) Alternation is represented by the metasymbol "|".

(b) Repetition is represented by curly brackets. Thus, {atom} implies that atom may optionally be omitted or repeated any number of times.

(c) Optionality is represented by square brackets. Thus, ["+"] implies that the plus sign may be optionally omitted.

(d) Parentheses serve for grouping.

(e) Terminal symbols (literals) are enclosed in quote marks.

(f) Non-terminals (identifiers) are not enclosed within quote marks.

The above syntax can best be illustrated by means of the format used in Wirth's article.

```
syntax     = {production}.

production= identifier "=" expression ".".

expression= term {"|" term}.

term       = factor {factor}.

factor     = identifier | literal | "(" expression ")" |
             "[" expression "]" | "{" expression "}".

literal    = """"character {character}"""".
```

2.2 DEFINITIONS

The following definitions are taken either from Dr. Brown's original user manual or from my own reference manual (Appendix A).

PUNCTUATION CHARACTER - Any character which is not a letter or digit.

ATOM - A single punctuation character or a sequence of letters and digits that is surrounded by punctuation characters. Also referred to as a token.

NEWLINE MARKER - An artificial character inserted by the macro processor at the end of each input record.

MACRO DEFINITION - The specification of a sequence of atoms which subsequently is recognized within a piece of text as a call of this macro. A definition includes

1. An atom or sequence of atoms referred to as the macro name.

2. The specification of zero or more multi-atom delimiters which serve to delimit the various arguments of a macro call.

3. A closing delimiter which marks the end of a macro call.

4. A piece of replacement text which, when evaluated, textually replaces the entire macro call within the scanned text.

MACRO - A sequence of atoms consisting of a macro name, arguments, and delimiters. When encountered during the scanning of text, the entire macro is replaced by the previously defined replacement text.

INSERT - A sequence of atoms consisting of an insert name, argument, and closing delimiter. The insert's argument specifies a certain quantity, such as a macro's argument, a macro's delimiter, or a literal, which is to be inserted into the scanned text. The quantity to be inserted is referred to as the insert text.

SKIP - A sequence of atoms consisting of a skip name, arguments, and delimiters. Skips prevent the recognition of macro names, inserts, and possibly other skips within certain pieces of text. The text contained between the skip name and closing delimiter is essentially "skipped" (i.e. not evaluated). Skips perform a function similar to that of comments within most programming languages.

CONSTRUCTION - A generic term for macro, insert, or skip.

GLOBAL CONSTRUCTION - A construction whose definition applies to all subsequent text evaluation.

LOCAL CONSTRUCTION - A construction which is recognized only within the text in which it was defined. Local constructions are considered global to any text subsequently called from the defining text.

OPERATION MACROS - Macros defined by the processor, not by the user. Operation macros cause some predefined system action to occur.

ARGUMENT - The sequence of atoms occurring between any two delimiters of a construction.

SOURCE TEXT - The text supplied as input to ML/I. The function of ML/I is to evaluate this text.

REPLACEMENT TEXT - Text which, when evaluated, textually replaces a macro call within the scanned text.

INSERT TEXT - Text which, as a result of an insert call, is to be inserted into the scanned text.

SCANNED TEXT - The text presently being evaluated by ML/I. The scanned text can be source text, replacement text, or insert text.

VALUE TEXT - The text resulting from the evaluation of a piece of scanned text.

OUTPUT TEXT - The text resulting from the evaluation of the source text.

SYNTAX OF A CONSTRUCTION - The specification of a construction name, the various delimiters of the construction, and the closing delimiter of the construction. The macro processor uses this syntax in recognizing a construction call within the scanned text.

DELIMITER NAME - A sequence of one or more atoms composing the specification of a single delimiter. A delimiter name is described syntactically as

delimiter_name = atom {("WITH" | "WITHS") atom}.

If two atoms are connected by WITHS, any number of spaces (including zero) may occur between the two atoms during the matching of the delimiter. If WITH is used to connect the two atoms, no spaces may occur between the atoms.

The following are examples of delimiter names:

1. COMPARE

2. ,

3. COMPARE WITH :

4. INTERCHANGE WITHS (

5. END WITHS ;

Each of these examples is a specification of a sequence of atoms which is to be recognized as a delimiter during the scanning of text.

LAYOUT KEYWORDS - Keywords which specify certain characters, or sequence of characters within delimiter names. The layout keywords are:

SPACE - a single space.

SPACES - a sequence of one or more spaces.

NL - the newline marker.

SL - the startline marker.

As an example, a delimiter name representing the atom "DO" followed by exactly two spaces followed by the atom "WHILE" would be written as

DO WITH SPACE WITH SPACE WITH WHILE

Likewise, the specification of a delimiter consisting of the atom ")" followed by one or more spaces followed by the end of line (i.e. newline marker) would be written as

) WITH SPACES WITH NL

If zero or more spaces were to separate ")" and the end of line marker, the delimiter name would be written as

) WITHS NL

DELIMITER SPECIFICATION - The specification of either a delimiter name or an option list (see the subsequent definition).

DELIMITER STRUCTURE - A set of delimiter specifications. A delimiter structure is defined by writing a structure representation, which defines all the delimiters of a construction and the successor(s) of each. Successors specify which delimiters to search for next when scanning the construction call. A successor may be:

1. Null (signifying the closing delimiter).
2. Another delimiter specification within the structure.
3. A set of alternative delimiter specifications within the structure.

If a delimiter is matched during the scanning of a construction call, the construction's delimiter structure is referenced to find the successor(s) of the current delimiter. Subsequent text is then scanned in an attempt to find this successor. This process continues until a closing delimiter is found.

A supplement to the foregoing definitions of macro, insert, and skip is the necessity to define a delimiter structure when defining a new construction. In defining the delimiter structure, the user must specify the name of the construction (often referred to as the name delimiter) and the successor(s) of each delimiter that is not a closing delimiter. By designing a suitable delimiter structure, users may define constructions with a variable number of arguments, constructions with optional arguments, and constructions with an unbounded number of arguments.

FIXED DELIMITERS - Refers to a structure representation consisting of a fixed pattern of delimiters. Each delimiter specification must be a delimiter name. Option lists may not occur within this type of structure. Thus, if delimiter name X is defined as the only successor of delimiter name Y, then upon finding delimiter X, the processor knows that the next delimiter will be Y. This holds for all the delimiter specifications of the fixed delimiter structure. Since each delimiter has but a single possible successor, each call of the construction will match the same pattern of delimiters. An example of a structure representation consisting of fixed delimiters is

INTERCHANGE WITHS (,) WITHS NL

The specification of the construction's name delimiter is

INTERCHANGE WITHS (

the second delimiter specification is

and the closing delimiter specification is

) WITHS NL

Note that for each delimiter specification within the example's structure representation, the succeeding delimiter specification consists of a single delimiter name. Therefore, when the construction is defined, the pattern of delimiters to be matched is fixed.

<u>OPTION LIST</u> - The mechanism used to specify that a delimiter has several optional alternatives as successor. An option list is written as

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

where each branch specifies a possible alternative successor of the delimiter specification preceding the option list. There are two restrictions placed on the branches of an option list.

1. Each branch must specify a unique sequence of atoms.

2. Each branch must begin with a delimiter name. This name is optionally followed by a delimiter specification. Therefore, the syntax of a branch is:

   branch = delimiter_name {delspec}.

The successor of each branch is usually taken to be the delimiter specification following the concluding ALL of the option list.

The structure representation

PRINT OPT X WITH SPACES WITH NL OR Y ALL ;

is an example of a structure containing an option list. The name of the construction is "PRINT" and the closing delimiter is a semi-colon (i.e. ";"). However, the structure's option list specifies two alternative sequences of atoms for the second delimiter of the construction. Thus, the delimiter following the construction name may be either the letter "X" followed by a sequence of one or more spaces followed by the newline marker (NL) or it may be the letter "Y". These sequences are specified by the two branches of the option list, that is, "X WITH SPACES WITH NL" and "Y".

<u>NODES</u> - Entities used to break up the purely sequential manner of searching for succeeding delimiters. Nodes define a successor of a delimiter to be a delimiter name or an option list elsewhere in the structure representation. The syntax of a node is:

node = nodeflag digit.

where the default value for nodeflag is the letter "N".

Nodes have two representations, nodeplaces and nodegos. Syntactically, a nodeplace and a nodego are the same. However, the meaning of a node is implicit in its textual position within the structure representation.

NODEPLACE - Represents the placing of a node at some position within a structure representation. This node will be the object of a nodego. Nodes may be placed only before a delimiter name or an option list.

NODEGO - Represents the action of going to a nodeplace. Nodegos stipulate that the search for delimiters is to continue at some point in the structure representation marked by a nodeplace. Nodegos can occur only at the end of a branch of an option list or at the end of a structure representation.

An example of the use of nodes within a structure representation is

MCSKIP OPT , N1 OR N1 NL ALL

The name delimiter of the construction is "MCSKIP" and the second delimiter is either "," or the newline marker (NL). The nodego N1 following the comma (",") within the first branch of the option list implies that if this branch is matched then the next delimiter specification of the construction is found by going to the nodeplace of the same name (i.e. N1). Note that this nodeplace occurs before the second branch of the option list. Consequently, the construction can have either two delimiters ("MCSKIP" and NL) or three delimiters ("MCSKIP", ",", and NL).

A structure representation specifying an unbounded number of delimiters is

SUM N1 OPT + N1 OR - N1 OR ; ALL

The name of the construction is "SUM" and the structure's option list contains three alternative successors to the name delimiter (i.e. "+", "-", or ";"). If the atoms "+" or "-" are matched then the subsequent nodego N1 specifies that the next delimiter is found by going to the nodeplace N1. Since this nodeplace occurs before the same option list, a looping condition exists such that whenever a "+" or "-" is matched as a delimiter, the next delimiter will be either "+", "-", or ";". If a ";" is matched, then the ";" is recognized as the closing delimiter and the scanning of the construction call is complete. Therefore, the following sequences of atoms match this structure

```
SUM 1 + 2 - 3;

SUM X + Y + Z + A - TOTAL - DIFF + MULT - 4 + 6;

SUM 1 ;
```

SYNTAX - Now that all the sub-components of a structure representation have been defined we can formalize their syntax.

```
structure_representation = delspec {delspec} [nodego].

delspec = [nodeplace] (delimiter_name | option_list).

delimiter_name = atom {("WITH" | "WITHS") atom}.

option_list = "OPT" branch_list "ALL".

branch_list = branch {"OR" [nodeplace] branch}.

branch = delimiter_name {delspec} [nodego].
```

2.3 DIFFERENCES BETWEEN MACRO PROCESSORS AND MACRO ASSEMBLERS

A common misconception encountered when discussing macro processors is the confusion between the function of a general-purpose macro processor and that of a macro assembler. Macro assemblers, by definition, are tied to an assembly language. This association makes the task of expanding macros much simpler. For example, in a macro assembler, the delimiters of a macro call are often predetermined. Thus, the syntax of a call is quite simple, since predetermined delimiters (such as commas) are easily recognized as separating the macro's arguments. However, a general-purpose macro processor, such as ML/I, has no knowledge of the syntax of a construction call and must rely on the user to supply this information when defining a construction.

Another major difference exists in the specification of the number of arguments a construction may have. Many macro assemblers restrict the number of arguments to a fixed amount, determined when the macro is defined. ML/I, however, has an unusual feature which provides for a variable number of arguments and delimiters for each construction. Consequently, several calls of the same construction may have completely different sets of delimiters separating the arguments of the call.

Macro assemblers may also require macro names to appear within certain fields, such as the operation field, of a record. Only names within this field are candidates for macro expansion. Since macro names are restricted to certain fields, they obviously must be of a certain size. A general-purpose macro processor cannot make this restriction. It must process names and delimiters of any size occurring anywhere within the source text.

Of course, some macro assemblers are more flexible than the prototypical one I have described. But, in general, the task of a macro processor is more complex since it must recognize variable length macro names in variable positions within the text, with variable syntax occurring in varying forms of input.

2.4 SPECIAL FEATURES OF ML/I

This document concentrates on several unusual characteristics which differentiate ML/I from other macro languages. The following features were chosen for examination because of their significant effect on the overall system design.

1. Users specify the syntax of a construction call (i.e. users are required to write a specification of the various delimiters when defining a construction). Each newly defined construction may have several different patterns of delimiters, with each pattern signifying a different syntax for a construction call.

2. Users may define constructions having a variable number of arguments and delimiters.

3. Multi-atom names and delimiters are allowed.

4. There is no restriction on nesting and recursion.

5. The format of macro calls is unrestricted (i.e. macros do not have to appear in certain fields of a record).

3. LANGUAGE FEATURES AND THEIR EFFECT ON SYSTEM DESIGN

3.1 THE SPECIFICATION OF CONSTRUCTION SYNTAX

The user specification of the syntax of a construction call was the most difficult feature of ML/I to implement. This specification is used in searching for the delimiters of a construction. Since constructions may be defined as having a variable number of delimiters, as well as a variable pattern of delimiters, the procedure which implements the delimiter search is rather complex.

3.1.1 FIXED DELIMITERS

There seemed to be several alternative methods by which the structure representation of a construction definition could be saved (where the structure representation specifies the delimiters of a construction call). If the representation was saved as a character string and essentially interpreted at macro-expansion time, then the job of representing the syntax within the processor would have been much simpler. However, the ease of implementing this scheme would be offset by the length of time it would take to recognize a call. If the structure representation was processed and transformed into an internal representation when encountered during the definition of a construction, then the task of searching for delimiters would potentially be much faster. Since constructions are defined once but are called an unbounded number of times, the latter approach was applied. Therefore, a parser transforms a structure representation into a directed graph whenever a construction is defined. The nodes of this graph contain information reflecting the various patterns of delimiters which may occur during a construction call.

An example of such a directed graph is illustrated in figure 1. This example shows the graph created for a structure representation containing fixed delimiters. The name of the construction, together with a pointer to the first node of the graph (i.e. ALPHA), is placed in the symbol table. When the construction is called, the various delimiters are found by following the sequence of pointers, beginning at ALPHA, until a null pointer is encountered. Within the graph, a node of type 1 contains 2 pointers: one pointing to a delimiter specification and the other pointing to the next node to be processed during the traversal of the graph. Atoms are read in and compared against the appropriate delimiter specification until a match is found. The search procedure then advances from the corresponding type 1 node to the next node of the graph and attempts to match the next delimiter of the construction. A type 0 node is a pointer node and contains a pointer to the next node to

be processed (the need for type 0 nodes is explained within the subsequent section concerning option lists). A construction has been successfully matched with its delimiters when a type 0 node is encountered which has a null pointer value.

EXAMPLE (1): INTERCHANGE WITHS (,) WITHS NL

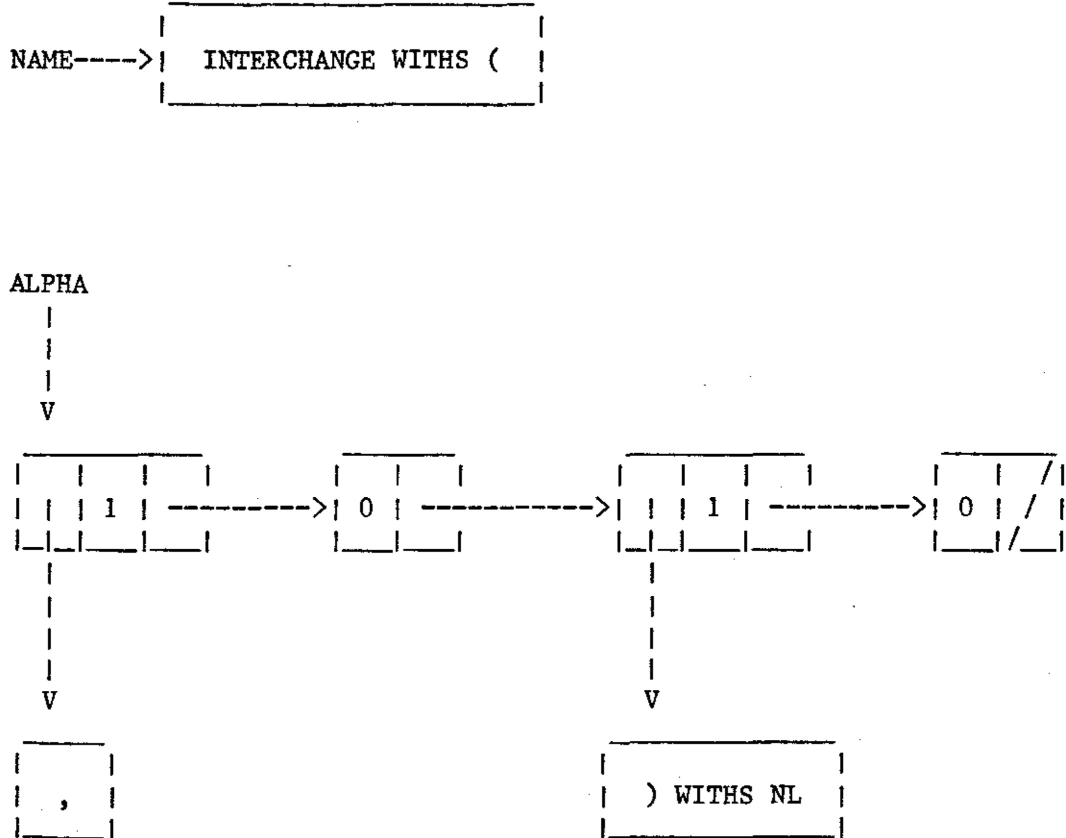


Figure 1. A Structure with Fixed Delimiters

3.1.2 OPTION LISTS

If constructions were required to consist of a fixed pattern of delimiters (i.e. delimiter-0 is followed by delimiter-1 which is followed by delimiter-2 .....), designing the directed graph would have been a fairly simple process. However, a special feature, called an option list, allows users to specify several alternative options as successor to the previous delimiter. The syntax of an option list is:

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

where each branch contains the specification of a possible successor to the previous delimiter. These branches are grouped together within the graph as a logical entity. Upon encountering a collection of branches, the procedure which searches for delimiters will compare the current atom against each branch of the option list for a possible pattern match.

In comparing figure 1 against figure 2, it is apparent that the directed graph becomes more complex whenever option lists are used. Two additional node types have been added to the graph. A type 2 node points to the first branch of an option list (essentially the "head" of the list of nodes which specify the various delimiters of the list). A type 3 node represents a branch of an option list. It contains 2 pointers: a pointer to the delimiter specification node for that branch and a pointer to the next branch of the option list. The matching of a delimiter against the branches of an option list is as follows: the pointer is followed from the head (i.e. type 2 node) to the first branch of the option list (i.e. type 3 node). The "delimiter" pointer of that type 3 node is followed and the current atom is compared against the branch's delimiter specification. If a match is found, the sequence of pointer nodes emanating from the corresponding type 1 node is followed. If a match is not found, the "next branch" pointer from the originating type 3 node is followed and the process is repeated. This continues until either a match is found or until the current atom has been unsuccessfully compared against each delimiter specification of the option list (which is determined whenever a null value is encountered for the "next branch" pointer of a type 3 node). If the atom does not match any of the branches, it is considered to be part of the construction's argument; the next atom is read in and the process is repeated with the first branch of the option list. A type 0 node pointing to another type 0 node (as occurs for the pointer node following the delimiter specifications "X WITH SPACES WITH NL" and "Y") signifies that the processing of an option list has been completed. This situation must be detected by the macro processor, thus necessitating the use of type 0 nodes as special intermediary pointer nodes. If type 0 nodes were not used, it would be impossible to discern the termination of an option list when traversing the nodes of the directed graph.

EXAMPLE (2): PRINT OPT X WITH SPACES WITH NL OR Y ALL ;

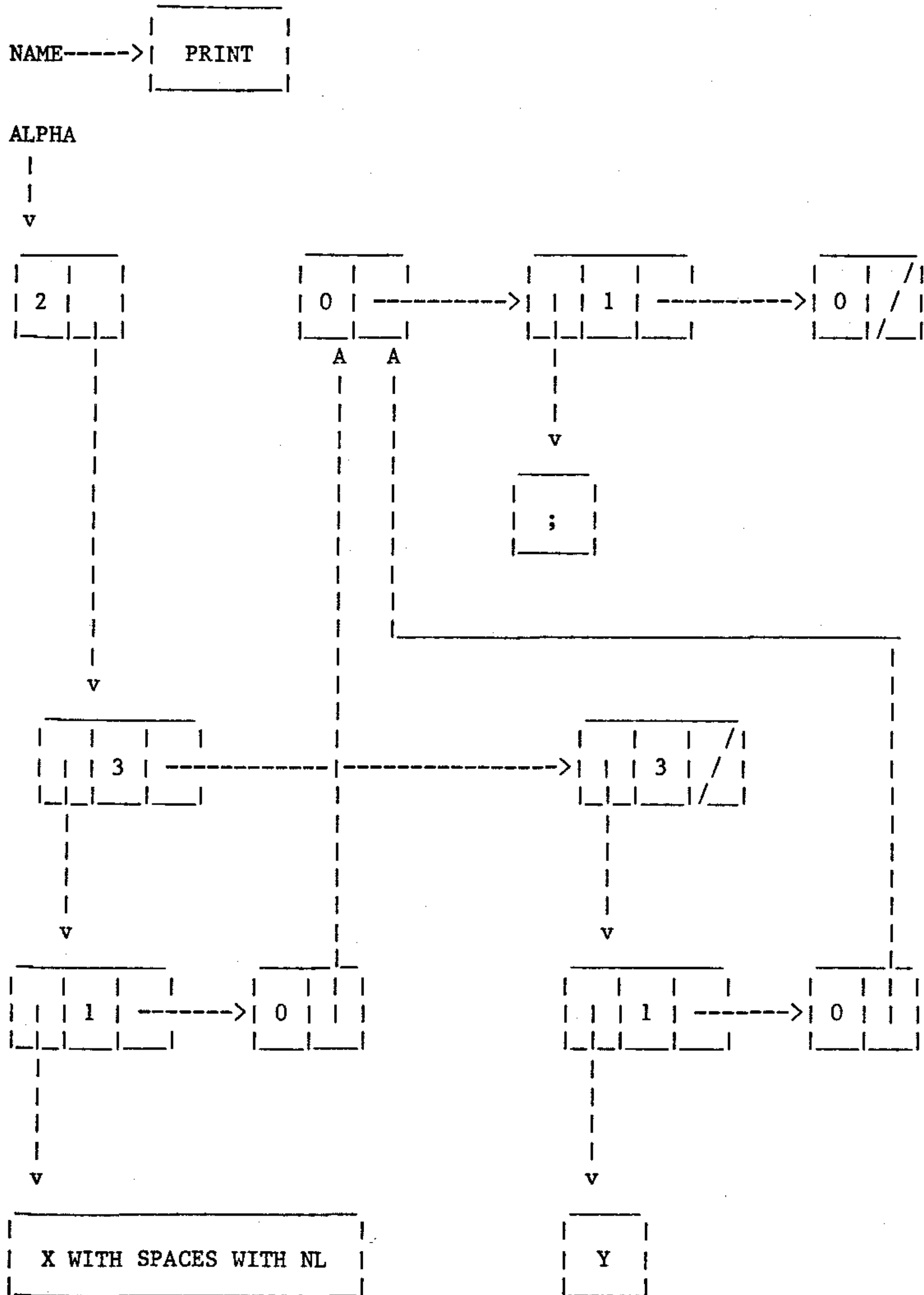


Figure 2. A Structure containing an Option List"

3.1.3 NODEPLACES AND NODEGOS

If a delimiter matches a branch of an option list, the next delimiter usually is assumed to be the delimiter specification following the keyword ALL. However, nodeplaces and nodegos are used to define successors elsewhere in the structure representation. A nodeplace marks the location from which the search for a delimiter pattern begins. A nodego signifies that the search is to continue at the specified nodeplace. Since nodegos and nodeplaces are used frequently within option lists, the representation of option lists becomes increasingly complicated. An interesting problem arises when a delimiter matches a branch of an option list and a nodego following that branch specifies that the next delimiter specification can be found within a subset of the branches of the containing option list. To handle this situation, a mechanism was necessary for restricting searches to subsets of option lists.

Figure 3 is an example of a graph containing nodeplaces and nodegos. The nodeplace N1 occurs before the keyword NL within the structure representation

```
MCSKIP OPT , N1 OR N1 NL ALL
```

and the nodego N1 follows the delimiter "," and precedes the keyword OR of the same structure. The significance of a node such as N1 (i.e. whether it is a nodeplace or a nodego) depends upon its textual position within the structure representation. Thus in this example, if an atom is found which matches the delimiter "," of the option list, the next delimiter specification of the construction is actually contained within a branch of the same option list. Notice that the type 0 node pointed to by the delimiter specification node of "," points to a type 3 node and not to another type 0 node. This configuration represents a nodego to a branch of an option list.

In general, the implementation of option lists was the most difficult task in the design of the directed graph. The elements of nodeplaces and nodegos combined with the nesting of option lists within the branches of other option lists resulted in the procedure which constructs the directed graph being the largest component of the macro processor.

EXAMPLE(3): MCSKIP OPT , N1 OR N1 NL ALL

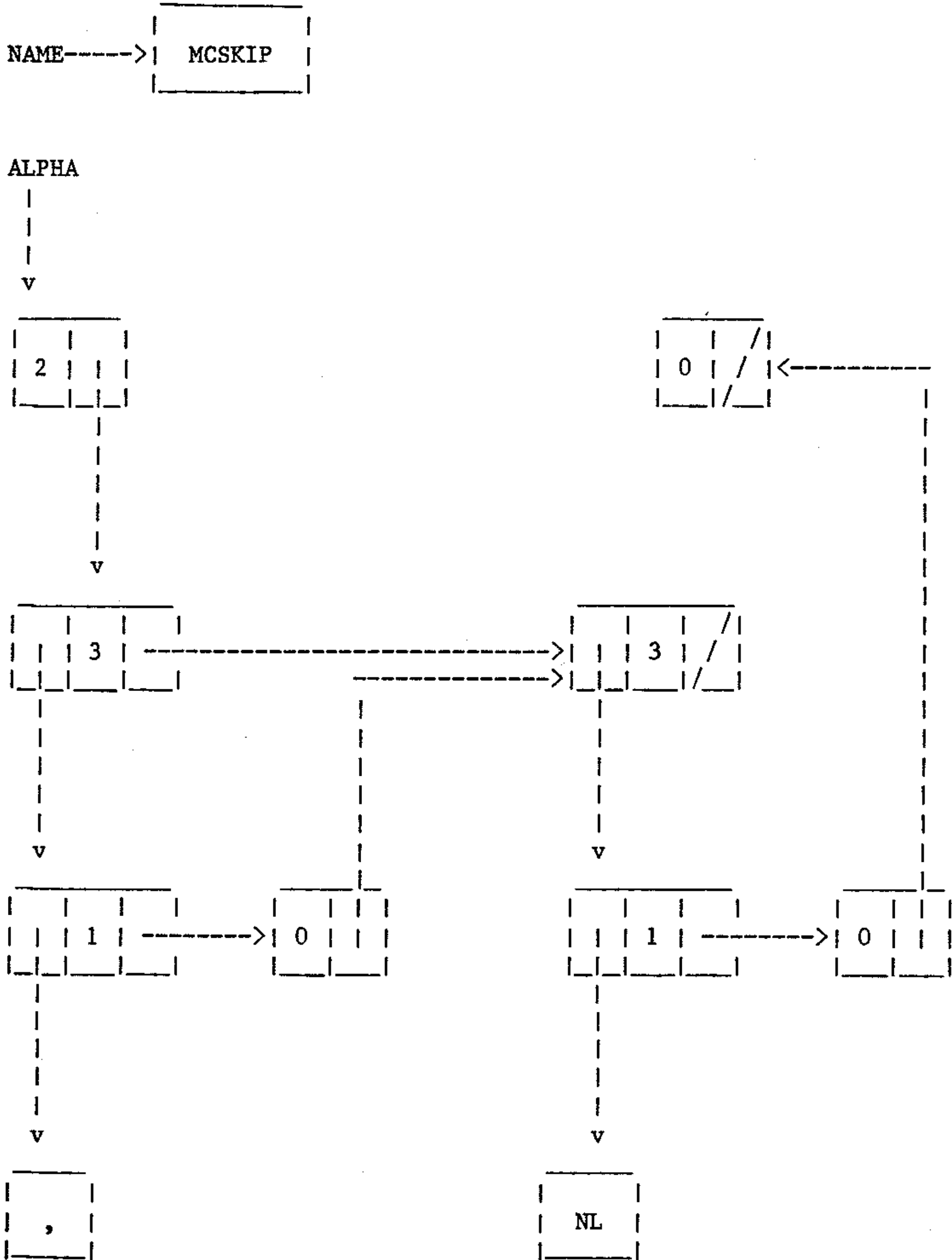


Figure 3. Structure with an Option List, Nodeplaces, and Nodegos

3.2 LOCAL SYMBOL TABLE MANIPULATION

Two language features in particular influenced the design of the local symbol table; the block-structured nature of ML/I construction definitions and the ability to delete randomly local definitions from the name environment.

3.2.1 NESTING OF DEFINITIONS

In ML/I, any construction may be defined within the replacement text of a macro, with "innermost" definitions overriding preceding definitions of the same name. Once a macro's replacement text has been evaluated, local constructions defined within that replacement text must be deleted from the symbol table. The nesting of replacement texts within other replacement texts leads to several levels of text evaluation. Constructions defined at one level are recognized as such within any nested levels unless they are overridden by new definitions.

The following example illustrates how ML/I naming conventions apply to nested levels of text.

```
 _______________________________________
|
|  SOURCE TEXT
|
|
|    local definition of macro A1
|
|    local definition of macro B1
|
|    local definition of macro C1
|
|        ________________
|       |
|       |  REPLACEMENT TEXT OF A1
|       |
|       |    local definition of insert A1
|       |
|       |    local definition of skip D2
|       |
|       |
|       |    B1
|       |
|       |    A1
|       |
|       |________________
|
|
|        ________________
|       |
|       |  REPLACEMENT TEXT OF C1
|       |
|       |    local definition of macro E2
|       |
|       |
|       |    B1
|       |
|       |    D2
|       |
|       |________________
|
|
|    E2
|
|_______________________________________
```

Within the replacement text of A1, the called macro (i.e. A1) is redefined as an insert. Consequently, the call of A1 within A1's replacement text actually refers to the new insert definition. Within the replacement text of C1, the symbol D2 is not recognized as a skip since its definition was local to the replacement text of A1 and thus no longer applies. Likewise, E2 is not recognized as a macro within the

source text since it was defined within C1's replacement text (and subsequently was not recognized within the outermost nesting level).

Symbols may also be redefined within the same level of text evaluation. For instance, in the example

```
|
|   local definition of insert A1
|
|   local definition of skip A1
|
|   local definition of insert A1
|
|_______________________________
```

the symbol A1 is defined successively as an insert, a skip, and another insert with each definition overriding the previous one.

3.2.2 DELETING LOCAL SYMBOLS

The operation macros MCNODEF, MCNOSKIP, and MCNOINS added a certain complexity to the manipulation of the local symbol table. These macros permit users to delete all local macro, skip, or insert definitions. For example, MCNOSKIP deletes all local skip definitions regardless of the level of text evaluation. In the following example, the second definition of SYM (as a skip) overrides the initial macro definition. However, the subsequent call of MCNOSKIP deletes all current skip definitions, resulting in the recognition of the second call of SYM as a macro.

```
 ______________________________
|
|
|  local definition of macro SYM
|
|  local definition of skip SYM
|
|
|  SYM
|
|
|  MCNOSKIP
|
|
|  SYM
|
|______________________________
```

3.2.3 ORGANIZATION OF THE LOCAL SYMBOL TABLE

The scheme used in implementing the symbol table was borrowed from the BLISS-11 compiler [3]. A hash table is used in which each entry contains

1. A pointer to a linked list (NT) of symbol names which hashed to the same location (where each name may have more than one declaration).

2. A pointer to a linked list (ST) of symbol declarations whose names hashed to this particular hash table entry. This list is maintained in reverse declaration order with the newest declaration at the head of the list and the oldest declaration at the tail of the list.

Upon exit from a particular level of text evaluation, the list ST is followed and declarations are deleted until the next outermost nesting level is reached. This is done for each hash table entry. To account for the random deletion of local construction definitions, all linked lists in the symbol table are maintained as doubly linked lists so that any member may be deleted at any time.

3.3 UNBOUNDED LOOK-AHEAD

When using ML/I, users may define constructions having multi-atom names and delimiters. This poses several problems to the procedure which searches for the delimiters of a construction call. For instance, to account for multi-atom delimiters, the searching procedure requires an unbounded amount of look ahead when attempting to match a sequence of atoms against a delimiter specification. This look-ahead is required whenever an atom of the scanned text matches the first atom of a multi-

atom delimiter. The macro processor must then look ahead to attempt a match of the subsequent atoms of the text against the remaining atoms of the delimiter specification. Since several different delimiter specifications may begin with the same atom, an unsuccessful match necessitates the restoration of the scanner to the state previous to the look-ahead. The processor may then attempt to match some other delimiter name.

An example best illustrates the problem. Suppose the text to be scanned contains the sequence

DO WHILE ;

and the construction names

DO WHILE (

and

DO WHILE NEXT (

are contained within the symbol table. The macro processor may first try to match the sequence

DO WHILE (

When the semicolon following the atom sequence "DO WHILE" is scanned, the macro processor determines that the sequence of atoms "DO WHILE ;" does not match the construction name "DO WHILE (". However, it must not discard these atoms but must rescan them (i.e. backup the scanner) in an attempt to match the name delimiter

DO WHILE NEXT (

This process of matching multi-atom delimiters introduced two problems which directly affected the scanner: the need to look ahead for the subsequent atoms of the multi-atom delimiter and the need to back up the scanner whenever it is determined that an inappropriate match has been attempted. The organization of the scanner and its method of responding to this situation are discussed in the next section.

3.4 THE LEXICAL ANALYZER

Owing to the nature of recursive and nested construction calls, the macro processor's scanner does not enjoy the luxury of scanning a single text segment. In fact, it often is called upon to return the next atom from any of a number of different texts in different stages of scanning. A consequence of this feature is that all information needed for scanning must be passed, as parameters, to the lexical analyzer. This includes a flag indicating the text to be scanned and the column in which the beginning of the next atom may be found. Since global variables may not be used in reflecting the state of the scan, any information needed for resuming the scanning process must be returned via the

parameter list.

The unbounded look-ahead which may be necessary when searching for delimiters requires the ability to "reverse" the scanner. The structure of the scanner makes this a fairly simple task. As stated, the scanner is given a flag indicating the text to be scanned and a column number. The flag is actually a pointer containing either the null value (for when the source text is being scanned) or the address of the sequence of text currently being scanned. Since the atoms scanned during the look ahead process are saved, reversing the scanner involves the scanner recursively calling itself to scan the "look-ahead" atoms and resuming the scan at the point where the look ahead terminated. However, this process becomes very complicated when the "look-ahead" atoms themselves try to look ahead during the matching of a construction name.

4. CONCLUSION

Many of the problems encountered during implementation derived from the "generality" of ML/I. A less extensive, and thus more restrictive, macro language probably would have resulted in a simpler and more efficient implementation. However, in judging any macro language, one must always consider the number of applications to which it may be applied.

4.1 USES OF ML/I

In his user manual [4], P. J. Brown mentions several applications for a general-purpose macro processor such as ML/I. Among these is the addition of extra statements and syntactic forms to an existing language so as to adapt it to a particular need. He also discusses program parameterization, such as the inclusion of debugging statements, applications in text editing and data format conversion, and using ML/I to implement the code generation phase of a compiler.

In fact, ML/I is such a powerful macro language that it can be used to produce compilers for high-level languages. Tanenbaum describes a "compiler-compiler", implemented using ML/I macros, for a system programming language [5]. Since ML/I provides automatic parsing, lexical scanning, symbol table manipulation, and syntactic error handling, it is suitable for implementing a compiler in a short amount of time.

The technique for implementing the compiler is fairly obvious. The specification of the syntax of a macro call in ML/I is analogous to the specification of the syntax of a programming language for a parser generator. The parser generator uses the supplied syntax to generate a parser. ML/I uses the syntax to parse the source text - no parser is generated. Code generation is performed by replacing the source statements by the assembly code contained within a macro's replacement text. Tanenbaum points out that the parsing does not use a context-free grammar to define the language to be translated but rather requires the user to provide a set of prototype statements (i.e. macros) which will drive the parser. Each of these macros, which must begin with a unique series of atoms, is a syntactic skeleton of a statement to be recognized.

The "compiler-compiler" implementors discovered several advantages in using ML/I. The macro processor automatically detects certain syntactic errors, such as missing delimiters, and issues appropriate error messages (semantic errors are not automatically detected by ML/I). Since the "compiler" is actually a macro processor, the source language enjoys extensive macro facilities without the aid of a separate preprocessor. ML/I also produces a listing of the source and translated code; additional routines which perform these functions are not needed. The

flexibility of ML/I was demonstrated by the large number of debugging and performance monitoring aids which the implementors added to the compiler. These include: procedure call/return tracing, variable tracing, assertion checking, a symbolic interactive debugger, and call frequency statistics. In fact, the compiler even performs optimization.

As can be expected, using a macro processor to "compile" a program is rather a slow process. However, statistics provided by Tanenbaum show that the generated object programs tend to be both fast and compact when compared with the code produced by compilers for other languages. This is rather surprising when you consider that ML/I was designed to perform the functions of a macro processor and not those of a compiler.

4.2 ADAPTABILITY OF A GENERAL-PURPOSE MACRO PROCESSOR

As noted, ML/I may be adapted to a wide range of uses, many of which are not usually considered to be within the realm of macro processing. This adaptability is, I believe, directly related to the flexibility of the language. In fact, because of the restrictive nature of most macro languages, many users would never consider the possibility of using a macro processor for certain applications mentioned in the previous section. Nevertheless, the ability of ML/I to deal with these situations demonstrates that the text replacement capabilities of a general-purpose macro language can be very powerful.

The environment in which the macro processor exists has a major impact on its usefulness. If certain essential tools, such as a compiler, are absent from a system, they could be implemented using ML/I in a fairly short period of time. ML/I can also provide certain test-related capabilities, such as adding debugging, program-tracing, or code-instrumentation statements to routines. Of course, if such tools are already available, then the attractiveness of using a relatively slow macro processor diminishes. However, in any environment, the general-purpose parsing capabilities of this macro processor make it a unique and beneficial tool.

5. APPENDICES

The documents listed below are appendices to this thesis. These documents are separate publications and as such are individually paginated and formatted.

Appendix A - Reference Manual

Appendix B - IBM System/360 Implementation

Appendix C - Program Logic Manual

Appendix D - Source Listings (on microfiche inserted in pocket in binding)

BIBLIOGRAPHY

[1] Brown, P. J. (1970). ML/I User's Manual, 4th Edition, University of Kent at Canterbury. Available through:
Computing Laboratory
The University of Kent at Canterbury
Kent England CT2 7NF

[2] Wirth, N. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?". Comm. ACM 20, 11 (Nov. 1977) 822-823. Copyright 1977 - reprinting privileges were granted by permission of the Association for Computing Machinery.

[3] Wulf, W., Johnsson, R., Weinstock, C., Hobbs, S., and Geschke, C. _The Design of an Optimizing Compiler_ American Elsevier, New York, N. Y., 1975

[4] Brown, P. J. (1970). ML/I User's Manual, 4th Edition, University of Kent at Canterbury. Page 1/1.

[5] Tanenbaum, A. "A General-Purpose Macro Processor as a Poor Man's Compiler-Compiler" _IEEE Transactions on Software Engineering_ Vol. SE-2, No.2, pp.121-125, June 1976

ML/I Reference Manual

TABLE OF CONTENTS

I. INTRODUCTION 1

- 1.1. Preface 1
- 1.2. General Format and Purpose 1
- 1.3. Exceptions to the Original Definition 1
- 1.4. Description of Syntax 2

II. TEXT 5

III. MACROS 7

- 3.1. Substitution Macros 7
- 3.2. Free Mode and Warning Mode 7
  - 3.2.1 Free Mode 8
  - 3.2.2 Warning Mode 8
- 3.3. Normal-Scan and Straight-Scan Macros 8
  - 3.3.1 Normal-Scan Macros 8
  - 3.3.2 Straight-Scan Macros 9

IV. INSERTS 11

V. SKIPS 15

- 5.1. Literal Brackets 17

VI. MACRO-TIME ENTITIES 19

- 6.1. Macro Variables 19
- 6.2. Syntax of Macro Variables and Macro Expressions 20

VII. THE ENVIRONMENT 23

- 7.1. The Evaluation of Replacement Text 23
- 7.2. Global and Local Environments 24
- 7.3. Protected and Unprotected Inserts 25

VIII. SCANNING AND EVALUATION 27

- 8.1. Nesting and Recursion 27
- 8.2. The Evaluation Process 27
- 8.3. Searching for Delimiters 28

IX. DELIMITERS AND DELIMITER STRUCTURES 31

9.1. Specifying Delimiter Structures 31
9.2. Delimiter Specification 32
9.3. Nodes and Option Lists 33
9.4. Nodes, Nodeplace, and Nodego 34
9.5. Points to Remember About Delimiters and Delimiter Structures 36
9.6. Keywords 38
9.7. Exclusive Delimiters 39

X. OPERATION MACROS 41

10.1. Facts Concerning Operation Macros 41
10.2. The Specifications of the Operation Macros 42
10.3. NEC Macros 43
10.3.1. MCWARN 43
10.3.2. MCINS 44
10.3.3. MCSKIP 45
10.3.4. MCDEF 46
10.3.5. MCNOWARN, MCNOINS, MCNOSKIP, and MCNODEF 47
10.3.6. MCWARNG, MCINSG, MCSKIPG, and MCDEFG 48
10.3.7. MCALTER 49
10.4. System Functions 50
10.4.1. MCLENG 50
10.4.2. MCSUB 51
10.5. Further Operation Macros 52
10.5.1. MCSET 52
10.5.2. MCNOTE 53
10.5.3. MCGO 54
10.5.4. MCPVAR 57

XI. ERROR MESSAGES 59

11.1. The Context of an Error Message 59
11.2. List of Error Messages 59
11.2.1. ILLEGAL MACRO ELEMENT 60
11.2.2. ARITHMETIC OVERFLOW 60
11.2.3. ILLEGAL INPUT CHARACTER 60
11.2.4. ILLEGAL MACRO NAME 60
11.2.5. UNMATCHED CONSTRUCTIONS 61
11.2.6. ILLEGAL SYNTAX OF ARGUMENT VALUE 61
11.2.7. REDEFINED LABEL 62
11.2.8. UNDEFINED LABEL 62
11.2.9. STORAGE EXHAUSTED 62
11.2.10. SYSTEM ERROR 63
11.2.11. SUBSIDIARY MESSAGE 63
11.2.12. STATISTICS 63

XII. HOW TO USE ML/I 65

12.1. The Evaluation of Source Text 65
12.2. The Evaluation of Replacement Text 68
12.3. The Evaluation of Inserted Text 69
12.4. Further Examples 70

XIII. EXCLUDED AND INCLUDED FEATURES 71

13.1. Exclusion List 71
13.2. Inclusion List 72

XIV. APPENDICES 75

BIBLIOGRAPHY. 77

1. INTRODUCTION

1.1 PREFACE

ML/I is a general purpose macro processor which supports both nested and recursive macro calls. The language was designed by P.J. Brown, University of Kent at Canterbury, Canterbury, Kent, England. This manual is based entirely on Brown's original ML/I user manual [1].

1.2 GENERAL FORMAT AND PURPOSE

In writing this manual, I kept two basic purposes in mind. One, I intended it to serve as a supplement to the original, whereby users, who are already familiar with ML/I, can reference the many features and definitions of the language. This manual is not, in any way, meant to replace or improve upon the original ML/I user manual written by P.J Brown. Thus, I have presupposed that anyone using this manual has: (1) read the original manual, and (2) is familiar with the basic constructs of the language. The reader will therefore find very few examples in this text (since it is not meant to be an introductory guide). What he will find, hopefully, is a detailed description of both the semantics and syntax of each of the significant features of the language. Since each description is intended, to some extent, to be independent of any other description, the reader may find some repetition of ideas occurring in different segments of the manual. The principle behind this repetition is to allow the user to reference certain segments of the manual and to find a somewhat complete description, for a particular feature, which contains all the necessary information which the user may need.

My second purpose in writing this manual was a simple one. I wished to familiarize myself with the language definition before attempting to write the program.

1.3 EXCEPTIONS TO THE ORIGINAL DEFINITION

Several of the features mentioned in this manual are not available in my current implementation. These have been marked with a "*" and denoted as "not implemented". Furthermore, any features that later were added to the language by Dr. Brown are not implemented. These include stop markers and controlled line markers. However, startline markers have been implemented. The reader should refer to chapter 13 for a complete list of both the excluded and included language features.

1.4 DESCRIPTION OF SYNTAX

The syntactic representations used in this manual were borrowed from a notation proposed by Niklaus Wirth [2]. The main characteristics of this notation are as follows:

(a) Alternation is represented by the metasymbol "|".

(b) Repetition is represented by curly brackets. Thus, {atom} implies that atom may optionally be omitted or repeated any number of times.

(c) Optionality is represented by square brackets. Thus, ["+"] implies that the plus sign may optionally be omitted.

(d) Parentheses serve for grouping.

(e) Terminal symbols (literals) are enclosed in quote marks.

(f) Non-terminals (identifiers) are not enclosed within quote marks.

The above syntax can best be illustrated by means of the format used in Wirth's article.

```
syntax     = {production}.

production= identifier "=" expression ".".

expression= term {"|" term}.

term       = factor {factor}.

factor     = identifier | literal | "(" expression ")" |
             "[" expression "]" | "{" expression "}".

literal    = """"character {character}"""".
```

In discussing the various features of ML/I, the following syntactic definitions will be used:

(a) {#} - signifies a series of zero or more blanks. In most instances, blanks can occur between the syntactic components of a a production. For those cases where this may not be obvious, I have attempted to clarify this point by inserting {#}.

(b) noblanks - implies that no blanks may occur between two components.

(c) atom= (letter|digit) {(letter|digit)}.

(d) punctuation_character - defined to be any character which is not a letter or digit. (Note that a blank is a punctuation character).

(e) neutral_atom - if an atom is represented by this component, then ML/I will make no attempt to recognize it as an environmental name (i.e. construction name or warning marker).

(f) construction= macro_call | insert | skip.

2. TEXT

The term text refers to a (possible null) sequence of atoms. In discussing text, the following definitions apply:

Source Text - Text supplied as input to ML/I. The function of ML/I is to evaluate this text. Temporary variables are not part of the environment used in the source text's evaluation. Any local construction defined in the source text is equivalent to a global definition (in the sense that it will be recognized in all subsequent text evaluation). The only inserts which are allowed in the source text are those which:

a. insert a numerical value literally

b. insert a numerical value through permanent or system variables.

c. insert a label.

Replacement Text - When defining a substitution macro, the user must specify a piece of replacement text. Upon encountering a call of the macro, ML/I replaces the entire macro call by the evaluated form of its replacement text. ML/I uses both the macro's local environment and the global environment in evaluating the text. Arguments of the macro are placed into the replacement text by means of inserts. Any construction (including the replacement text's own macro) can be called from the text, and any insert can appear within the text.

Inserted Text - Refers to a piece of text which is used to determine the result of an insert call. Inserts are most frequently used within replacement text to insert a macro's arguments or delimiters. The text to be considered as inserted text is indicated by certain flags within the insert (see chapter 4). Depending upon the flag used, the inserted text may either be evaluated or inserted literally (see section 7.3 for the evaluation rules pertaining to inserts). Constructions, including other inserts, may be nested within inserted text.

Output Text - The text resulting from the evaluation of the source text.

<u>Scanned Text</u> - Refers to the text presently being evaluated by ML/I. The scanned text can be either source text, replacement text, or inserted text.

<u>Value Text</u> - The text resulting from the evaluation of a piece of scanned text.

3. MACROS

In ML/I there are two types of macros: substitution macros and operation macros. Substitution macros adhere to the more traditional viewpoint of macros. Their purpose is to perform some type of text replacement. Operation macros on the other hand are designed to perform some predefined system action such as adding a macro name to the environment (MCDEF), performing macro time arithmetic and assignment (MCSET), or performing a macro time GO TO to a macro label (MCGO). Except for the operation macros MCLENG and MCSUB, operation macros do not generate any text.

3.1 SUBSTITUTION MACROS

Operation macros will be dealt with in chapter 10. This section is concerned only with the definition of substitution macros.

To define a macro, the user must specify:

1. A delimiter structure (to be explained in chapter 9). The first delimiter, called the name or zero delimiter, is the macro name.

2. The replacement text of the macro. Upon processing a macro, ML/I replaces the entire macro call by the evaluated form of the replacement text.

3. An integer ($\geq$ 3) indicating the capacity of the macro. The capacity specifies the number of temporary variables to be allocated and added to the local environment when the macro is called.

4. An on/off option. If the option is "on" then the macro is a normal-scan macro. If the option is "off" then the macro is a straight-scan macro. The difference is explained in section 3.3.

3.2 FREE MODE AND WARNING MODE

While evaluating a piece of text, ML/I uses the environment for the purpose of recognizing macro calls. The recognition of a macro call depends on whether the environment is in free mode or warning mode. In free mode, every predefined macro name is recognized as the start of a macro call. In warning mode, a warning marker must precede a macro name if it is to be recognized as a macro call.

The syntactic form of a macro call for each of the modes is as follows:

3.2.1 FREE MODE

macro_call= name_delim [arg_list].

arg_list= {arg delim} arg closing_delim.

where name_delim is a predefined macro name of the form

name_delim= atom {atom}.

and the sequence of delimiters, delim, of macro_call is a pattern that can be derived from the delimiter structure of the macro. The syntactic forms of arg, delim, and closing_delim are described later during the explanation of normal_scan and straight_scan macros.

3.2.2 WARNING MODE

* (not implemented)

macro_call= warning_marker name_delim [arg_list].

arg_list= {arg delim} arg closing_delim.

where warning_marker is a predefined warning marker of the form

warning_marker= atom {atom}.

and the definitions of name_delim and the sequence of delimiters in arg_list are the same as they were under free mode.

3.3 NORMAL-SCAN AND STRAIGHT-SCAN MACROS

Upon encountering a piece of text, whether source, replacement or inserted text, ML/I searches for the secondary delimiters (as specified by the macro's delimiter structure) until the closing delimiter is found. The manner in which ML/I scans a macro call depends upon whether the macro is a <u>normal-scan</u> or <u>straight-scan</u> macro.

3.3.1 NORMAL-SCAN MACROS

For normal-scan macros, nested constructions are recognized during the search for the delimiters of the macro call. If a nested construction is encountered during the search, then ML/I will search for the delimiters of the nested construction before continuing the search for the

outermost macro's delimiters.

Therefore, for normal scan macros, we have

arg= {arg_list}.

(Note that arg can be empty)

arg_list= macro_call | insert | skip | neutral_atom.

delim= macro_call {delim_list} | insert {delim_list} |

skip {delim_list} | neutral_atom {delim_list}.

(Note that delim cannot be empty)

delim_list= macro_call | insert | skip | neutral_atom.

closing_delim= delim | name_delim.

where delim is a delimiter which has been specified (by the macro's delimiter structure) as a legal successor of the preceding delimiter of the macro call. The closing_delim of a macro_call is a delimiter which has been specified as the last delimiter of the macro call (i.e. has no successor).

3.3.2 STRAIGHT-SCAN MACROS

Whereas the scanning of a normal-scan macro call involves the recognition of nested constructions, the scanning of a straight-scan macro does not. Therefore, for a straight-scan macro call, the syntactic formats for arg, delim, and closing_delim are:

arg= {neutral_atom}.

(again arg can be empty)

delim= neutral_atom {neutral_atom}.

closing_delim= delim.

(again delim cannot be empty)

where delim and closing_delim have the same specifications imposed on them from the delimiter structure as in the case of normal-scan macros.

In this instance, the form of neutral_atom in a call of a straight-scan macro does not imply that neutral_atom cannot be a macro, insert, skip name or a warning marker. However, it does mean that ML/I will not recognize it as such.

The replacement texts of both normal-scan and straight-scan macros are evaluated in the same manner. The important point to remember about substitution macros is that a macro call is always replaced by some piece of text, possibly null, which is the output text derived from the evaluation of the macro's replacement text. Since this replacement text may itself contain macro calls, the process of evaluating a macro call implies recursion.

4. INSERTS

Inserts are used to insert certain quantities (i.e. arguments, delimiters, literals, labels, or text resulting from the evaluation of arguments or delimiters) into portions of the text (see section 7.3 for the evaluation rules pertaining to inserts).

To insert a quantity, the user writes an insert whose syntactic definition is given by:

insert= insert_name insert_arg closing_delim.

where insert_name is a previously defined name of an insert of the form

insert_name= atom {atom}.

and closing_delim is a previously defined closing delimiter of the insert of the form

closing_delim= atom {atom}.

Upon encountering an insert, ML/I evaluates the insert argument, insert_arg, and obtains the evaluated form of the argument, eval_insert_arg. Therefore, the original insert argument can contain macro calls, inserts, skips, or neutral atoms. Thus its syntactic format can be represented as:

insert_arg= construction {insert_arg} | neutral_atom {insert_arg}.

Once ML/I has obtained eval_insert_arg through evaluation, the following syntactic forms and actions hold:

eval_insert_arg= {#} [flag] {#} macro_expression.

```
flag= "A" | "B" | "D" | "L" | "W" {#} "A" |
                              "W" {#} "B" |
                              "W" {#} "D".
```

Next, ML/I must interpret macro_expression and the interpretation must return a value "N". This value is used to reference the Nth argument, arg_N, or the Nth delimiter, delim_N, of the macro whose replacement text contains the insert which is currently being evaluated by ML/I. If "N" is used incorrectly (e.g. the Nth argument or delimiter is referenced and doesn't exist), then an error message is produced.

The following cases show the relationship between "N" and flag. Cases (a)-(f) are defined only for those inserts which appear within the replacement text of a macro (since they refer to the arguments or delimiters of a macro call).

a. flag= "A" (N must be $\geq$ 1)

   When this flag is used, ML/I takes the Nth argument of the macro call, deletes any leading and trailing spaces, and evaluates it. The text resulting from this evaluation is used as the value of the insert (i.e. the text to be inserted).

b. flag= "B" (N must be $\geq$ 1)

   ML/I does the same as in case (a) except that leading and trailing spaces are not deleted from the Nth argument before it is evaluated.

c. flag= "D" (N must be $\geq$ 0)

   As in case (b) except that the Nth delimiter is being evaluated rather than the Nth argument. The Nth delimiter is defined as the delimiter following the Nth argument. Delimiter zero is taken to be the name of the macro.

d. flag= "W" {#} "A" (N must be $\geq$ 1)

   As in case (a) except that the Nth argument, with leading and trailing blanks deleted, is not evaluated but is inserted literally. If the Nth argument is either a macro call, insert, or skip, then it will not be recognized as such by ML/I (i.e. it is not evaluated).

e. flag= "W" {#} "B" (N must be $\geq$ 1)

   As in case (d) except that leading and trailing blanks are not deleted from the argument before it is inserted literally.

f. flag= "W" {#} "D" (N must be $\geq$ 0)

   As in case (e) except that we are now considering the Nth delimiter, not the Nth argument. Again the Nth delimiter is not evaluated but is inserted literally.

g. flag= "L" (N must be $\geq$ 1)

"L" indicates that a macro label is being defined for the text in which the insert occurs and "N" is taken to be the number of the new macro label. The label, if acceptable, is added to the current environment (most likely to be used later as the subject of a macro-time GO TO statement). In order to be acceptable, a label must not be multiply defined within a piece of either replacement text or inserted text. Therefore, each label to be inserted within a piece of text must be unique to that text (i.e. N must be different for each label). An exception to this rule occurs during the evaluation of the source text. Labels can be inserted into the source text but they are not added to the environment (i.e. they cannot be the subject of a backward GO TO statement). Therefore, there is no requirement that macro-time labels in the source text be unique. Labels are local to the piece of text in which they were defined; consequently, one can use the same label numbers in different pieces of text. Note that in the case of inserts which use this flag, nothing is actually inserted into the text.

h. If flag does not exist, then N can be any integer. In this case, the text to be inserted is N represented as a character string. The character string does not contain any leading blanks and is preceded by a sign only if N is negative.

5. SKIPS

Skips are used to prevent the recognition of macro names, inserts, and possibly other skips within certain pieces of text (i.e. they cause ML/I to skip the evaluation of a piece of text). Comments constitute a primary use of skips. If the beginning and end of a comment are defined as a skip, then macro names can appear within the comment without being subject to recognition and evaluation.

The syntactic form of a skip is:

```
skip= name_delim [{arg delim} arg closing_delim].
```

where name_delim is a previously defined skip name of the form

```
name_delim= atom {atom}.
```

and closing_delim is a closing delimiter which has previously been defined as such in the skip's delimiter structure. As with macro calls, the sequence of delimiters, delim, can be derived from the skip's delimiter structure.

To define a skip, therefore, the user must specify:

1. A delimiter structure.

2. Three on/off options whose values and meanings are given below:

   a. text option:

      on - The arguments of the skip are copied over to the value text.

      off - The arguments are not copied.

   b. delimiter option:

      on - The delimiters of the skip are copied over to the value text.

      off - The delimiters are not copied.

c. matched option:

on - The skip is a matched skip (described below).

off - The skip is a straight-skip (described below).

Note that a skip differs from a macro in that it has no replacement text. The portions (possibly null) of a skip that are copied over to the value text depend entirely upon the independent setting of the text and delimiter options.

How ML/I scans a skip depends on whether it is a matched or straight skip. The process of scanning a skip is similar to that of scanning a macro call. Upon encountering a skip, ML/I uses the skip's delimiter structure to search for the specified delimiters until a closing delimiter is found. With straight skips, however, no other skips are recognized during the scan for the closing delimiter. On the other hand, the scanning of a matched skip does involve the recognition of any nested skips. ML/I will search for the delimiters of any nested skips before continuing the search for the delimiters of the outermost skip. In both cases during the scan, neither macro calls nor inserts will be recognized within the skip.

An analogy can be drawn between the scanning of a matched skip and the scanning of a normal-scan macro. The single difference is that only nested skips are recognized within matched skips, whereas any construction will be recognized during the scan of a normal-scan macro. An even closer comparison can be made between straight skips and straight-scan macros. In fact, a straight skip can be represented as a straight-scan macro (but not vice-versa).

To complete the syntactic definition of skip we have:

1. straight skips

   arg= {neutral_atom}.

   delim= neutral_atom {neutral_atom}.

   closing_delim= delim.

2. matched skips

   arg= {(neutral_atom | skip)}.

where delim and closing_delim are defined as for straight-skips.

Note that in both straight and matched skips, all spaces not defined as part of a delimiter are absorbed into the beginning or end of an argument. Also, as with straight scan macros, the form of neutral_atom does not mean that macro names and insert names cannot be nested within a

skip. However, it does indicate that any nested, non-skip constructions are not to be recognized as such (i.e. ML/I makes no attempt to initiate a search for their closing delimiters).

5.1 LITERAL BRACKETS

When using ML/I, it is recommended that the user has, as part of his environment, a skip definition whose matched and text options are enabled and whose delimiter option is disabled. This type of skip, called a literal bracket, is used to copy a piece of text literally over to the value text. In processing such a skip, ML/I will drop the literal brackets and take the skip's argument to be the result of the skip. The only constructions that will be recognized within this argument are other skips (remember that macro calls and inserts are never recognized as such within skips).

6. MACRO-TIME ENTITIES

Since ML/I is a conditional macro processor, it contains certain features known as macro-time entities. These entities include a macro-time assignment statement (MCSET), a macro-time conditional GO TO statement (MCGO), macro labels, and macro variables. Very often, the result of evaluating a piece of replacement text depends upon the delimiters used in the macro's call. By writing a macro's replacement text in an appropriate manner, the user can use these macro-time entities to test the delimiters of the call and generate text accordingly. Thus ML/I provides for the implementation of a simple programming language which may be utilized during text evaluation.

6.1 MACRO VARIABLES

Macro variables are entities which can be used, at macro time, for a variety of purposes. Certain applications of macro variables include the controlling of the operation of ML/I, the insertion of a macro variable's integer value into a piece of text, and the testing of conditions for the alteration of processing during the evaluation of replacement text. The MCSET operation macro allows the user to perform arithmetic operations on macro variables; the MCGO operation macro permits the testing of these macro-time variables and the possible alteration of the flow of evaluation, depending on the outcome of the test. Thus, macro variables are very often used as switches during the evaluation of the replacement text of either a macro with a variable number of arguments or a macro with a variable pattern of delimiters.

There are three types of macro variables:

1. Permanent variables - denoted as P1,P2,P3... Before beginning the evaluation of the source text, ML/I will allocate an implementation-defined number of permanent variables which are added to the global environment and remain in the environment for the duration of the evaluation. Additional permanent variables may be allocated and added to the environment by means of the operation macro MCPVAR (see section 10.5.4). Permanent variables have no initial values and can be used for whatever purposes the user desires.

2. System variables - denoted as S1,S2,S3....... System variables control the operation of ML/I. ML/I adds an implementation-defined number of system variables to the global environment before beginning the evaluation of the source text. Unlike permanent variables, system variables are assigned initial values. These values can be changed at any time by the user (by means of the MCSET

macro) but care must be taken in doing so. The user is not allowed to add or delete system variables from the global environment. The appendix describes the initial values and meanings of the system variables.

3. Temporary variables - denoted as T1,T2,T3....... Before evaluating the replacement text of a macro call, ML/I adds a certain number of temporary variables (usually three) to the macro's local environment. These temporary variables remain in the macro's local environment for the duration of the replacement text's evaluation. The number of temporary variables allocated is given by the capacity of the macro (see section 10.3.4). The first three temporary variables of each local environment are assigned the initial values described in section 7.1. The user can use the MCSET operation macro to assign values to any of the temporary variables (including T1, T2, and T3). Note that temporary variables can be used only within the replacement text of a macro; therefore, there are no temporary variables in the environment during the evaluation of the source text. Since macro calls can be nested within other macro calls or appear within a macro's replacement text, there may be several allocations of temporary variables existing at the same time. Each allocation is local to a different environment.

6.2 SYNTAX OF MACRO VARIABLES AND MACRO EXPRESSIONS

To reference a macro variable, the user specifies a letter ("P", "S", or "T") and an unsigned positive integer. Thus the Nth permanent variable is indicated by

```
"P" noblanks "N"
```

where "N" is an unsigned positive integer. The syntactic specification of a macro variable is therefore:

```
macro_variable= "T" noblanks subscript |
                "S" noblanks subscript |
                "P" noblanks subscript.

subscript= macro_variable | positive_integer.

positive_integer= "1" | "2" | "3" |.....
```

where the macro variable being referenced is indicated by the value of subscript.

During evaluation, the user can perform macro-time arithmetic by means of macro expressions. The syntactic format of a macro expression is:

```
macro_expression= term {("+"|"-") term}.

term= primary {("*" | "/") primary}.

primary= [("+"|"-") noblanks] operand.

operand= unsigned_integer | macro_variable.

unsigned_integer= "0" | "1" | "2" | "3" |....
```

where "*" denotes multiplication and "/" denotes division. The result of division is rounded off to the largest integer not exceeding the exact result. Upon performing the arithmetic operations of a macro expression, ML/I will return an integer value. This value must not exceed an implementation defined maximum integer value. Overflow and division by zero are detected and reported as errors.

7. THE ENVIRONMENT

During evaluation, ML/I uses what is called the environment to determine the output text derived from the evaluation of macro calls, inserts, and skips. Therefore, the environment must originally contain any macro names, insert names, skip names, and warning markers which the user wishes to be recognized by ML/I. Additional constituents of the environment are imposed by the requirements of inserts. Inserts may reference the arguments or delimiters of a macro call; therefore, these too must be added to the environment. Macro variables can be used either by inserts or certain operation macros, implying that they are a part of the environment as well.

7.1 THE EVALUATION OF REPLACEMENT TEXT

Now let us consider the case of evaluating the replacement text of a macro call. The environment before the call is encountered consists of:

1. Some macro names and definitions (considered in this context to be global to the upcoming call).

2. Some insert names and definitions (again considered global to the call).

3. Some skip names and definitions (also global to the call).

4. A number of system and permanent variables (also global to the call).

Upon encountering and scanning the call, the environment is supplemented by:

5. The arguments of the call.

6. The delimiters of the call.

7. A number of temporary variables (at least 3) determined by the capacity of the macro. The first three temporary variables have as initial values:

   T1 - The number of arguments of the current macro call.

   T2 - The number of macro calls, including operation macros, so far performed by ML/I. Since this number is unique for each call, it can be used by macros for the purpose of placing labels within the output text.

T3 - The current depth of nesting of macro calls. The depth of nesting is defined as the number of macros currently being processed. For instance, let us suppose that we have a macro FIRST which has within its replacement text a call of a macro SECOND which in turn has in its replacement text a call of the macro THIRD and furthermore these are the only macros which are to be called. If FIRST is embedded in the source text, then T3 for FIRST is 1, T3 for SECOND is 2, and T3 for THIRD is 3. Calls of operation macros do not count in the setting of T3.

While evaluating a macro's replacement text, the environment may be further supplemented by:

8. Macro labels - added to the environment by means of inserts.

9. New macro, insert, and skip definitions. These new definitions are considered local to the called macro's replacement text and global to all macros called from within this replacement text.

This, therefore, is the environment which is used in evaluating the replacement text of a macro. Note that the process of setting up a new, distinct environment will be repeated for any macros called from within this replacement text.

7.2 GLOBAL AND LOCAL ENVIRONMENTS

In the previous section, the environment was described by means of its constituents. Constituents (5) - (9) of this description are considered local to the called macro's environment. They are added to the environment upon encountering, scanning, and evaluating a macro call. They are deleted from the environment upon the completion of the processing of the macro's replacement text. However, constituents (1) - (4) will remain in the environment upon the resumption of the evaluation of the text which follows the macro call. When this evaluation has been completed, some of the members of constituents (1) - (3) may be deleted from the environment (since these constituents in turn may have been local to the text). The members of constituent (4) are never deleted. Constituents (1) - (7) are fixed before the evaluation of the replacement text begins; constituents (8) and (9) may be added to the environment during the evaluation of the replacement text.

Now suppose we are evaluating the replacement text of a macro OLD. During this evaluation, we encounter the definition of a previously undefined macro NEW (i.e. MCDEF NEW AS <.....>). This macro definition is then added to constituent (9) (i.e. it is a local macro definition). In the evaluation of the remaining portion of OLD's replacement text and in the replacement text of any macro called from within this remaining portion, NEW will now be recognized by ML/I as a macro call. But upon the return to the text which originally called OLD, NEW will no longer be recognized as a macro name. In other words, upon encountering a local macro definition, the definition is added to the current text's

local name environment, thus becoming part of the global name environment of all macros called from the subsequent portion of the current text. However, this definition is not added to the local name environment of the containing text. If NEW had been defined as a global macro name (by using MCDEFG instead of MCDEF), then it would be recognized as a macro in all subsequent text evaluations, including the evaluation of the text which originally called OLD. This implies that during the processing of a macro's replacement text, the global name environment can be changed, and any such changes will affect all subsequent text evaluations. To change the global name environment, the operation macros MCWARNG, MCINSG, MCSKIPG, and MCDEFG are used. Note that global constructions defined by these operation macros cannot be deleted from the environment.

7.3 PROTECTED AND UNPROTECTED INSERTS

* (protected inserts are not implemented - all inserts default to unprotected)

An exception to the definitions of the preceding section concerns *protected* inserts. For protected inserts, if the arguments or delimiters of a macro call are to be inserted, they are evaluated under the local environment which was in force when the *macro call* was originally scanned. Any changes in the local name environment made while evaluating the replacement text will not affect the evaluation of the arguments or delimiters specified by the insert. If the insert is *unprotected* then the inserted text is evaluated under the environment which was in force when the *insert* was encountered.

In our example of the OLD macro, if NEW was the first argument of OLD's macro call and an insert with a flag of "A1" was encountered after the processing of MCDEF NEW AS <.....>, NEW would not be recognized as a macro name during the evaluation of the argument. On the other hand, if the insert were unprotected, ML/I would recognize NEW as a macro name during the evaluation of the first argument and the text to be inserted would become the evaluated replacement text of NEW. If NEW had been defined as a global macro name, then it would be recognized as such whether the insert were protected or not.

The same type of action applies to local definitions of inserts, skips, or warning markers. When one of these construction definitions is encountered, it will be added to the current text's local name environment. Whether these local definitions will be recognized as such during the evaluation of any inserted text will again depend upon the type of insert (i.e. protected or unprotected).

8. SCANNING AND EVALUATION

8.1 NESTING AND RECURSION

ML/I supports both nested and recursive construction calls. This means that any construction can be called from within any piece of text (source, replacement, or inserted text). A consequence of this feature is that the arguments of a macro call may themselves contain calls of other constructions, including a call of the containing macro. Likewise, it is possible to call a macro from within its own replacement text (or even redefine a macro, by means of MCDEFG - either from within its own replacement text or from within a piece of inserted text).

8.2 THE EVALUATION PROCESS

The process of evaluation involves the recognition of any environmental names (i.e. construction names or warning markers) which occur in the scanned text. In evaluating text, ML/I must at the same time scan the text. This is done by taking each atom of the text, one by one, and comparing it against a table of environmental names. Any atom which does not match a table entry is copied over to the value text unchanged. If a match is found, then (depending on the match) the following actions are taken:

1. For macros - ML/I searches for the macro's closing delimiter (see next section). The macro call (i.e. macro name, arguments, and delimiters) is not copied over to the value text; rather, the macro's replacement text is evaluated and the result derived from this evaluation is used as the value text resulting from the macro call. Note: leading and trailing blanks are not deleted from the replacement text before evaluation. Therefore, care must be taken in defining the replacement text of any macro which is to be called from a highly structured source program (e.g. assembler language program).

2. For inserts - ML/I searches for the closing delimiter of the insert and evaluates the insert argument. Depending upon the evaluated form of the argument, the insert may or may not be replaced by text. In either case, the insert (i.e. name, argument, and delimiter) is not copied over to the value text.

3. For skips - ML/I searches for the closing delimiter of the skip. The value of the skip (i.e. the text to be printed) depends on whether or not the text and delimiter options have been set.

4. For warning markers (* not implemented) - ML/I attempts to recognize the next atom as a macro name. If a macro name does not follow a warning marker, an error message is produced.

In some cases, the evaluation of a piece of text does not include the recognition of certain environmental names. This can occur in the following situations:

1. During the scanning of straight skips and straight-scan macro calls - no nested constructions are recognized.

2. In matched skips - only nested skips are recognized.

3. In warning mode - macro names are recognized only after warning markers.

When recognizing an environmental name, ML/I must deal with multi-atom names. Upon scanning an atom which is the first atom of a multi-atom name, ML/I will continue the scan, attempting to match the newly scanned atoms with the remaining atoms of the name. The same process is used in matching multi-atom secondary delimiters. If the end of the current piece of text is reached before finding the remaining atoms of the name, the preceding atoms are not considered by ML/I as part of an environmental name (i.e. they are copied over to the value text unchanged).

8.3 SEARCHING FOR DELIMITERS

Upon recognizing a construction name, ML/I references the construction's delimiter structure and initiates a search for the secondary delimiters of the call. This search terminates when the closing delimiter is found. Note that if the construction name is also defined as a closing delimiter, then a search is not necessary. If, during the search, ML/I encounters the end of the current piece of text without finding the closing delimiter, then the construction is considered unmatched and an error message is produced (for an exception to this rule see section 9.7 concerning exclusive delimiters).

Since ML/I provides for nested constructions, it is possible to encounter a call of another construction during the search for a closing delimiter. If this occurs, ML/I will search for the delimiters of the nested construction before continuing the search for the delimiters of the containing construction. An additional point to be noted is that any nested constructions encountered during the search for a containing construction's closing delimiter are not immediately evaluated. Since many of these nested constructions occur within the arguments of macros, they will be evaluated if and when the argument is inserted with an appropriate flag ("A" or "B").

One further point should be stated concerning the search for secondary delimiters - ML/I requires that all the delimiters of a construction

call lie within the same piece of text (i.e. the delimiters can all be found within either the source text, the same replacement text, or the same inserted text).

9. DELIMITERS AND DELIMITER STRUCTURES

In order to separate the arguments of a construction call, certain atoms or sequences of atoms, called delimiters, must be defined. The name of the construction is known as the name delimiter or delimiter zero. The delimiter following the Nth argument is defined as the Nth delimiter. The last delimiter of the call is called the closing delimiter. Note that the name delimiter is the only delimiter not preceded by an argument. All the other delimiters, called secondary delimiters, are preceded by an argument which in turn is preceded by some other delimiter (possibly the name delimiter).

In some cases, a construction may consist of a single delimiter which simultaneously serves as a name delimiter and a closing delimiter. Other constructions, such as inserts, may have only one argument which is preceded by the construction's name delimiter and followed by the construction's closing delimiter. Still other constructions, such as macros and skips, can have either a variable number of arguments or a conceivably unbounded number of arguments.

The delimiters of a construction are used by ML/I in the following way. During the process of evaluating a piece of text, ML/I will compare each atom of the scanned text with a table of construction names. When a match occurs, ML/I will proceed to search for the secondary delimiters (if any) of the construction. The search is terminated upon finding the construction's closing delimiter. Therefore, when defining a new construction, the user must have some means of specifying the name(s) and possible secondary delimiters of the construction. This is accomplished by means of a delimiter structure.

Since the delimiter structure is referenced during the search for the secondary delimiters of a construction, it must contain not only the name or names of a construction but also those delimiters (if more than one) which are to follow the name delimiter in the construction's call. Thus the delimiter structure must specify a sequence of delimiters which is to be used during the search for the construction's secondary delimiters.

9.1 SPECIFYING DELIMITER STRUCTURES

In order to define a new construction, certain operation macros must be called. In defining a new macro, the user must call MCDEF or MCDEFG; for new inserts, the user calls MCINS or MCINSG; and for new skips, the macros MCSKIP or MCSKIPG are called. Each of the these operation macros has as one of its arguments the specification of the new construction's delimiter structure. This section contains both the syntactic format

and meaning of this specification.

In defining a delimiter structure, the user writes a structure representation which specifies:

1. All the delimiters contained in the structure.

2. The successor(s) of each delimiter.

Before describing the syntactic form of a structure representation, it is first necessary to describe its sub-components.

9.2 DELIMITER SPECIFICATION

The most basic member of a structure representation is the delimiter. A delimiter may be a single atom or a sequence of several atoms. The user specifies a single delimiter by writing a delimiter name, delim_name, of the form

```
delim_name= atom [{atom_list}].

atom_list= "WITH" atom | "WITHS" atom.
```

where "WITH" and "WITHS" are reserved keywords (described along with other keywords in section 9.6) and atom is described syntactically by

```
atom= layout_keyword | non_layout_keyword

layout_keyword= "SPACE"|"SPACES"|"TAB"|"NL"|"SL".
```

The components of layout_keyword are described in section 9.6 and non_layout_keyword is any atom not recognized as a layout_keyword.

The reason for the sub-component layout_keyword is that the user may wish to specify either spaces, tabs, newline markers, or startline markers as part of delim_name. For instance, there are many operation macros which have a newline as their closing delimiter. Thus NL is used to represent the newline marker that is appended, by the input program, to the end of every record. In a similar manner the user may wish to indicate either a tab or a newline as part of a multi-atom delimiter. Layout keywords allow him to do so.

The simplest case of a structure representation is the specification of a delimiter structure with fixed delimiters. This structure is represented by writing down the delimiters in the order in which they are to appear within the construction's call. The first delimiter is the name delimiter, the next delimiter is the successor of the name delimiter, the next delimiter the successor of this delimiter, and so on

until the closing delimiter. A structure representation with fixed delimiters can be written as

structure_representation= delim_name {delim_name}

Note that if a delim_name is followed by another delim_name and neither is a punctuation character, then they must be separated by one or more spaces, tabs, or newlines so that they can both be recognized as separate atoms.

9.3 NODES AND OPTION LISTS

One of the special features of ML/I is its capability to define constructions with a variable number of arguments as well as constructions with various patterns of delimiters where each pattern gives a unique meaning to the construction. Consequently, the user must have some means of specifying constructions whose delimiter structures are not composed solely of fixed delimiters. The devices for permitting the specification of these more intricate structures are known as nodes and option lists. The problem in defining the delimiters of a structure remains the same as that of a structure with fixed delimiters. The structure must still identify the name(s) of the construction and the successor of each secondary delimiter that is not defined to be a closing delimiter. The difference now is that each delimiter can have a list of possible alternative successors and the successor of each delimiter is no longer constrained to being the delimiter specification which directly follows the delimiter.

The option list mechanism is used to designate the alternative successors of a delimiter (as well as designating the alternative names of a construction). The format of an option list is denoted by

option_list= "OPT" branch_list "ALL".

branch_list= branch {"OR" [nodeplace] branch}.

where "OPT", "OR", and "ALL" are reserved keywords. The meanings of branch and nodeplace are described later in this section.

Before going any further, we need to define a delimiter specification, del_spec, which is used to specify either a delim_name or an option_list.

del_spec= [nodeplace] delim_name | [nodeplace] option_list.

Each branch of an option_list specifies either a delimiter or a delimiter followed by the specification of its succeeding delimiters.

branch= delim_name {del_spec} [nodego].

Usually the successor of the last del_spec of each branch is understood to be the del_spec following the terminating keyword "ALL" of the option list. Consequently, when every delim_name within a branch has been matched, ML/I will continue the search for the delimiters at the point following the "ALL" which concludes the containing option list. This rule is overridden by means of nodes.

9.4 NODES, NODEPLACE, AND NODEGO

Nodes are used to break up the purely sequential manner of searching for a succeeding delimiter. Nodes can be placed both by means of a nodeplace (which indicates a location in the structure representation) and gone to by means of a nodego (which stipulates that the search for delimiters is to continue at some point in the structure representation marked by a nodeplace). The point at which the search is to continue must be either a delimiter name or an option list.

The action of placing a node indicates a position in the structure representation which may be the object of a nodego. A nodeplace is represented by:

nodeplace= nodeflag noblanks unsig_pos_integer.

unsig_pos_integer= "1"|"2"|"3"|"4"|.......

where nodeflag is usually the letter "N" but can be changed to either a single letter or digit by the operation macro MCALTER.

Nodes can be placed before, but never after, the "OPT" of a new option list; after an "OR" of an option list; or before a delim_name which is not contained in an option list. To place a node, the user writes a nodeplace at the desired point in the structure representation. Each nodeplace within the structure must have as its unsig_pos_integer a number which is unique to the containing representation. Any positive integer (in any sequence the user wishes) may be used. Any nodeplace within a structure representation is local to that representation and is not associated in any way with any other node occurring within a different structure.

If a node is placed immediately before the "OPT" of an option list, then the action of going to this node specifies that the next delimiter to be searched for is one of the alternatives of the option list. In some cases, a node may be placed immediately after an "OR" of an option list. This signifies that the action of going to the node predicates that the next delimiter is either the delim_name following the nodeplace or else is to be found in one of the subsequent branches of the containing option list.

Now that nodeplace has been defined, the meaning of nodego can be presented in more detail.

The syntactic representation of nodego is:

nodego= nodeflag noblanks zero_or_pos_integer.

zero_or_pos_integer= "0"|"1"|"2"|"3"|"4"|....

where nodeflag is the same flag as was used in the definition of node-place.

Notice that a node of "NO" can be used as a nodego but not as a node-place. The node "NO" is used to designate an _exclusive delimiter_ (described in section 9.7). If "NO" is the successor of some delimiter, then this delimiter is interpreted as an exclusive delimiter. Any other nodego must specify a node which occurs somewhere in the structure representation.

A nodego can occur only at the end of a branch of an option list (i.e. immediately before an "OR" or an "ALL") or at the very end of the structure representation. To go to a node, the user simply writes a nodego at the desired point in the structure representation. The nodego can go to any point in the structure representation at which a nodeplace can legally occur. Syntactically, a nodeplace and a nodego are the same. However, the meaning of a node is implicit in its textual position within the structure representation.

Now that all the sub-components of a structure representation have been defined, its syntactic definition can be presented.

structure_representation= del_spec {del_spec} [nodego].

Note that this definition applies both to delimiter structures with fixed delimiters and to those structures without fixed delimiters.

9.5 POINTS TO REMEMBER ABOUT DELIMITERS AND DELIMITER STRUCTURES

When specifying a delimiter structure, the user should be aware of the following facts.

1. Whereas null arguments are allowed, null delimiters are not. Delimiters must consist of at least one atom.

2. Multi-atom delimiters are allowed. In specifying the delimiter name, the various atoms of the name are connected by either "WITH" or "WITHS" (see section 9.6 for an explanation of the difference between these two keywords).

3. The definition of each construction must include a specification of the construction's delimiter structure.

4. A closing delimiter is a delimiter with no successor in the delimiter structure.

5. The scanning of a construction call continues until a closing delimiter is found.

6. If a construction has several alternative names, then each name must be a unique sequence of atoms.

7. Each branch in an option list must be a unique sequence of atoms.

8. Apart from the preceding two restrictions, the delimiter specifications of a structure need not be unique.

9. For each secondary delimiter defined within a delimiter structure, there must be some sequence of successors leading from the name delimiter to the secondary delimiter.

10. Every argument of a construction must be followed by one of the construction's delimiters.

11. The delimiters of a macro call are added to the macro's environment.

12. Delimiters are not evaluated when a call is scanned. However, they may be evaluated when inserted.

13. A construction is considered unmatched if the conclusion of the current piece of text is encountered while the processor is still searching for the closing delimiter of a construction. An exception is provided by exclusive delimiters (described in section 9.7).

14. Do not choose an argument of a construction to be the same as the next secondary delimiter being searched for by the macro processor. If you must use an argument which is the same as the next delimiter, then enclose the argument in literal brackets.

15. All the delimiters of a construction should be contained in the same piece of text.

16. The insertion of a delimiter cannot alter or affect the containing text's local name environment.

17. Several different delimiter names can all begin with the same atom or series of atoms. The processor will always try to match the longest delimiter name (this applies to name delimiters as well).

18. Generally, the user should choose delimiter names which are different from all construction names or warning markers defined in the current environment. If a series of atoms can be interpreted by the processor in more ways than one (i.e. either as a construction name, warning marker, or secondary delimiter), then the following precedence rules apply:

    a. The macro processor checks to see if the series of atoms is an exclusive delimiter of the current construction call.

    b. If not (a) then the macro processor checks to see if the series of atoms is part of a longer delimiter name.

    c. If not (b) then the macro processor will check to see if the series of atoms is a secondary delimiter of the current construction call.

    d. If not (c) then the macro processor will check to see if the series of atoms is a local environmental name (i.e. construction name or warning name)

    e. If not (d) then the series of atoms is compared against the table of global environmental names.

    f. If a match is found in either (d) or (e), the most recent definition applies.

19. Since arguments of operation macros are evaluated before being processed and delimiter structures occur as the arguments of certain operation macros, it is recommended for most cases that one enclose the delimiter structure in literal brackets when it is being used as an operation macro's argument. This prevents any delimiters occurring in the structure from possibly being recognized as macro calls.

20. If two atoms are connected by "WITH" in delim_name, it is an error if both of the atoms are not punctuation characters.

21. A nodego must refer to a nodeplace which occurs somewhere in the structure.

22. "NO" cannot be placed. However, it can be gone to (signifying that the previous delimiter is an exclusive delimiter).

23. Only closing delimiters can be defined as exclusive delimiters.

24. A nodeplace cannot be placed after an "OPT".

25. The user should try to minimize the number of delimiter specifications within a structure representation in order to minimize the amount of storage needed to store the structure.

26. Keywords cannot be used as delimiters.

27. Two nodes cannot be placed in succession.

28. Each nodeplace within a structure must be unique.

29. Each "OPT" of an option list must have a matching "ALL".

30. Each delimiter structure must have a closing delimiter.

9.6 KEYWORDS

The following is a list of keywords which can be used within a structure representation.

WITH - Reserved keyword, can be used only within the specification of delim_name. Within delim_name, any two atoms joined by "WITH" must not have any intervening spaces between them when used as part of the specified delimiter. It is an error to have two atoms within delim_name connected by "WITH" if neither of the atoms is a punctuation character.

WITHS - Reserved word, can be used only within the specification of delim_name. The two atoms joined by "WITHS" within delim_name can have zero or more intervening spaces between them when used as part of the specified delimiter.

SPACE - Layout keyword, can be used only within the specification of delim_name. The keyword "SPACE" represents the specification of a single space as the value of an atom specified within delim_name.

SPACES - Layout keyword, can be used only within the specification of delim_name. The keyword "SPACES" represents the specification of a sequence of one or more spaces as the value of an atom specified within delim_name.

TAB - (* not implemented). Layout keyword, can be used only within the specification of delim_name. The keyword "TAB" represents the specification of the tab character as the value of the specified atom within delim_name.

NL - Layout keyword, can be used only within the specification of delim_name. The keyword "NL" represents the specification of the

newline marker as the value of an atom specified within delim_name.

SL - Layout keyword, can be used only within the specification of delim_name. The keyword "SL" represents the specification of the startline marker which can be appended optionally to the start of each line of input text. Setting the system variable Sl to 1 will invoke this option.

OPT - Reserved word, signifies the beginning of an option list.

OR - Reserved word, used to separate the branches of an option list.

ALL - Reserved word, indicates the end of an option list.

N noblanks zero_or_pos_integer - Reserved word, represents a node.

Each of the preceding keywords can be changed by the MCALTER operation macro described in section 10.5.4.

9.7 EXCLUSIVE DELIMITERS

* (not implemented)

During the process of evaluating a piece of text (whether source text, replacement text, or inserted text), ML/I will usually encounter a construction call where

```
construction_call= construction_name {delim_list}.

construction_name= macro_name | insert_name | skip_name.

delim_list= {argument delim} argument closing_delimiter.
```

and the format of argument depends upon whether construction_name specifies a macro, insert, or skip.

The construction_call, from the construction_name to the closing_delimiter, is not copied over to the value text. Rather, ML/I isolates the entire construction_call, evaluates it, and copies the evaluated form of the call (possibly null) over to the value text. Thus, the evaluation of the scanned text resumes at the point immediately following the construction's closing delimiter.

Exclusive delimiters provide an exception to the foregoing processing method. If the closing delimiter of a construction call is defined as an exclusive delimiter, then upon isolating the construction_call, the closing delimiter will not be included as part of the call. Therefore, when the evaluation of the scanned text resumes, it proceeds from the construction's exclusive delimiter and not from the point immediately

following it.

The advantage of having such a feature is best exemplified by nested constructions. Suppose we had a macro REPLACE which takes only one argument and has a newline marker as its closing delimiter. Suppose the argument of a call of REPLACE is a nested macro call. Thus the call of REPLACE is of the form:

REPLACE nested macro call NL

Now if the nested macro call has as its closing delimiter a newline, then NL is needed to match both the nested macro call and the call of REPLACE. Since ML/I searches for the delimiters of nested constructions before searching for the delimiters of containing constructions, NL will be used first to match the nested macro call. If it has been declared as an exclusive delimiter of the inner macro, then the search for the closing delimiter of REPLACE will begin with the same NL, thus causing REPLACE to be matched with its closing delimiter. Consequently, by employing exclusive delimiters, the user can use the same delimiter to close out several nested constructions as well as the containing construction.

In the preceding example, the nested macro call, without its closing delimiter of NL, is treated as the first (and only) argument of REPLACE. Now suppose that within the replacement text of REPLACE, the user wishes to insert the argument (i.e. nested macro call) using an insert flag of either "A" or "B". Since NL was not included as part of the nested macro call, the evaluation of the argument will find it to be unmatched. In such a situation, ML/I applies the following rule; if the Nth argument of a macro call is a construction and if during the process of inserting the Nth argument the construction is found to be unmatched, then ML/I will examine the Nth delimiter of the containing macro to see if this delimiter (or the sequence of atoms which begin the delimiter) is defined to be an exclusive delimiter of the nested construction. If so, the nested construction is considered matched. In the case of a nest of apparently unmatched constructions, the same process is applied in turn to each construction.

One further point should be mentioned concerning exclusive delimiters. Exclusive delimiters can be used only within macro calls and skips. In the case of macros, an exclusive delimiter is not taken to be part of the macro's call. However, it is treated like any other delimiter in that it is added to the macro's local environment, thus making it eligible for insertion into the macro's replacement text. Exclusive delimiters, for skips, are not considered as part of the skip and therefore are not affected by the skip's delimiter option.

10. OPERATION MACROS

10.1 FACTS CONCERNING OPERATION MACROS

This chapter describes the operation macros which are used by ML/I to perform certain predefined system actions. In discussing these macros, the user should be cognizant of the following facts:

1. Operation macros are defined by ML/I, not by the user, and serve an important role in the functioning of the macro processor.

2. The definitions of the operation macros are used to initialize the name environment.

3. The names of all operation macros begin with "MC".

4. The arguments of all operation macros are evaluated before being processed (since the arguments may contain calls of substitution macros, operation macros, inserts, or skips). Any leading or trailing spaces are deleted before the evaluation. In most cases the user will wish to inhibit this evaluation, particularly in the case where NL is the closing delimiter of the operation macro and the macro's last argument extends over several lines (i.e. contains one or more NL markers). In situations such as this, the user should enclose the operation macro's arguments in literal brackets.

5. Operation macros, except for the system functions MCSUB and MCLENG, do not generate any value text when called.

6. Operation macros can be called from within any piece of text.

7. The majority of the operation macros are known as NEC (name environment changing) macros. They are used either to add to or to delete from the local and global name environments.

8. Any operation macro which defines a new construction must have a specification of the construction's delimiter structure as an argument.

10.2 THE SPECIFICATIONS OF THE OPERATION MACROS

In describing the individual operation macros, I have chosen to use a format similar to that used in Dr. Brown's original manual. Therefore, the following subsections are used to describe each operation macro.

(1) Purpose

(2) General form - gives the syntactic format of a call of the macro.

(3) Restrictions - gives the syntactic formats of the arguments of the operation macro.

(4) Order of evaluation - describes the order of the evaluation of the arguments. The order is sequential if this subsection is omitted. If a NEC macro has been called, the environment is changed only when all of the arguments have been successfully evaluated.

(5) System action - describes the action performed by ML/I. The term "current environment" refers to the environment in force when the macro was called.

10.3 NEC MACROS

The name environment changing macros are described in this section.

10.3.1 MCWARN

* (not implemented)

Purpose - to define a local warning marker.

General form - "MCWARN" argument NL

Restrictions - argument= delim_name.

System action - the argument is added to the current environment as a local warning marker. The definition of a local warning marker immediately places the current environment in warning mode (see section 3.2.2).

10.3.2 MCINS

* (partially implemented - inserts are unprotected by default; therefore the user need not specify either "P" or "U" when defining an insert)

Purpose - to define a local insert

General form - "MCINS" argb NL

or

"MCINS" arga "," argb NL

Restrictions - argb= delim_name delim_name.

arga= "P" | "U".

System action - argb is taken to be the delimiter structure of a new local insert definition. The name delimiter of the structure is added to the current name environment. The insert is "protected" if arga is absent and is defined to be "unprotected" only if arga is present and is represented by the letter "U".

10.3.3 MCSKIP

Purpose - to define a local skip.

General form - "MCSKIP" argb NL

or

"MCSKIP" arga "," argb NL

Restrictions - argb= structure_representation.

```
arga= "M" {#} skip_list |
      "D" {#} skip_list |
      "T" {#} skip_list.

skip_list= "M" {#} skip_list |
           "D" {#} skip_list |
           "T" {#} skip_list.
```

System action - argb is taken to be the delimiter structure of a new local skip definition. The name delimiter of the structure is added to the current name environment. The setting of the skip options is done by means of arga - if arga contains an "M", the matched option is set; if it contains a "T", the text option is set; and if it contains a "D", the delimiter option is set. If arga is absent, none of the options is set. If argb contains a "," and arga is absent, then the "," should be enclosed in literal brackets. This prevents ML/I from interpreting the "," as the delimiter between arga and argb.

10.3.4 MCDEF

Purpose - to define a local macro.

General form - "MCDEF" argb "AS" argc NL

or

"MCDEF" argb "SSAS" argc NL

or

"MCDEF" arga "VARS" argb "AS" argc NL

or

"MCDEF" arga "VARS" argb "SSAS" argc NL

Restrictions - arga= macro_expression.

argb= structure_representation.

argc must be enclosed in literal brackets if either: (1) the user wishes to delay the evaluation of the replacement text until the macro is called, or (2) argc contains newline markers which can be interpreted as the closing delimiter of MCDEF.

Order of evaluation - arga, argc, argb

System action - argb is taken to be the delimiter structure of a new local macro definition. The name delimiter of the structure is added to the current name environment. The macro's replacement text is specified by argc (remember that leading and trailing blanks are deleted from argc before it is evaluated). The capacity (i.e. the number of temporary variables to be added to the macro's local environment) is taken to be MAX(arga,3). If 'arga "VARS"' is absent, the capacity is three by default. If MCDEF is called with the delimiter "AS", the new macro is considered to be a normal-scan macro; if it is called with the delimiter "SSAS", the new macro is considered a straight-scan macro (see sections 3.3.1 and 3.3.2).

10.3.5 MCNOWARN, MCNOINS, MCNOSKIP, AND MCNODEF

* (MCNOWARN is not implemented, since there are no warning markers)

Purpose - to delete local constituents of the current environment.

General form -

(a) "MCNOWARN"

(b) "MCNOINS"

(c) "MCNOSKIP"

(d) "MCNODEF"

System action - MCNOWARN - deletes all local warning markers from the current environment. If no global warning markers are defined, then the current environment is placed in free mode (see section 3.2.1). A call of "MCWARN" must always be preceded by a warning marker.

MCNOINS - deletes all local insert definitions from the current environment.

MCNOSKIP - deletes all local skip definitions from the current environment.

MCNODEF - deletes all local macro definitions from the current environment. The user cannot delete operation macros from the environment because their definitions are global.

Note that for each of the preceding macros, the name delimiter is also the closing delimiter.

10.3.6 MCWARNG, MCINSG, MCSKIPG, AND MCDEFG

* (MCWARNG is not implemented)

Purpose - to define global warning markers, inserts, skips, and macros.

General form - the same syntax as used for the equivalent local macro definitions.

Restrictions - the arguments have the same syntactic restrictions as have the arguments of the corresponding local definitions.

System action - the actions performed by each of the preceding global macros are equivalent to the actions performed by the corresponding local macros. The only difference is that the constituents defined by these macros are added to the global, not the local, name environment. Global constructions cannot be deleted from the environment by any of the macros defined in the previous subsection.

10.3.7 MCALTER

Purpose - to alter either the secondary delimiters of operation macros or the keywords (both layout and reserved) used in structure representations.

General form - "MCALTER" arga "TO" argb NL

Restrictions - arga= atom.

argb= atom.

Furthermore, arga must specify either a secondary delimiter of an operation macro or a keyword used in structure representations. Also, it is required that the length of argb be less than or equal to the length of arga.

Order of evaluation - argb, arga

System action - whenever a situation arises in which arga would have been used, the user must now use argb. It is important to remember that MCALTER has a global effect. If a user wishes MCALTER to have a local effect only, then he must call MCALTER again to change the altered arguments back to their original form.

10.4 SYSTEM FUNCTIONS

This section describes the operation macros known as system functions. There are two things to remember about system functions:

1. They are the only operation macros which return a piece of text when evaluated.

2. They have ")", not NL, as their closing delimiters.

10.4.1 MCLENG

Purpose - to find and return the length of a character string.

General form - "MCLENG" {#} "(" arga ")"

where arga yields, upon evaluation, a character string and "(" is part of the macro name.

System action - ML/I returns a numerical value, represented as a character string, with leading spaces deleted. A sign precedes the value only if it is negative. The value returned represents the number of characters in the evaluated form of arga.

10.4.2 MCSUB

Purpose - to find and return a substring.

General form - "MCSUB" {#} "(" arga "," argb "," argc ")"

where "(" is part of the macro name. The evaluation of arga, argb, and argc will yield, respectively, a character string, a numeric value, and a numeric value.

Order of evaluation - arga, argb, argc

(see below concerning the evaluation of argc)

System action - in describing the substring returned by a call of MCSUB, the following definitions hold:

L = the number of characters in the evaluated form of arga.

RB = the numeric result derived from the evaluation of argb.

VB = (RB if RB > 0
(
(L+RB if RB $\leq$ 0

RC = the numeric result derived from the evaluation of argc.

VC = (RC if RC > 0
(
(L+RC if RC $\leq$ 0

The values of VB and VC describe a valid substring of arga only if

$1 \leq VB \leq VC \leq L$

If this relation holds, then MCSUB returns a substring of arga which begins at character position VB and extends for VC-VB+1 characters. The first character of arga is defined to be in character position one. If the relation does not hold, MCSUB returns a null string.

10.5 FURTHER OPERATION MACROS

The remainder of the operation macros are described in this section.

10.5.1 MCSET

Purpose - to perform macro-time assignments to macro variables.

General form - "MCSET" arga "=" argb NL

Restrictions - arga= macro_variable.

where the evaluation of arga must yield the name of a macro variable which exists in the current environment.

argb= macro_expression.

System action - the result returned by argb is assigned to the macro variable returned from the evaluation of arga.

10.5.2 MCNOTE

Purpose - to allow the user to generate his own error and debugging messages.

General form - "MCNOTE" arga NL

System action - arga is treated as if it were a system message (i.e. it is printed on the ERROR file). Three lines are skipped before printing the message. After printing the message, ML/I prints the context of the call of MCNOTE (see section 11.1 for an explanation of the context of an error message).

10.5.3 MCGO

Purpose - to perform a macro-time GO TO or conditional GO TO.

General form - (a) "MCGO" arga NL

(b) "MCGO" arga del_1 argb cond argc NL

del_1 = "IF" | "UNLESS".

cond = "=" | "BC" | "EN" | "GE" | "GR".

where "BC" means Belongs to Class,
"EN" means Equals Numerically,
"GE" means Greater than or Equal to,
"GR" means GReater than.

Restrictions - arga= {#} "L" noblanks macro_expression.

where macro_expression must

1. yield a non-negative result
2. yield a positive (i.e. >0) result if MCGO is called from the source text.

If cond= "BC" then argc= "I" | "L" | "N" where "I" stands for the class of identifiers, "L" stands for the class of letters, and "N" stands for the class of numbers.

If cond= "EN" | "GE" | "GR" then

argb= macro_expression.

and

argc= macro_expression.

Order of evaluation - argb, argc, arga

In the conditional form of MCGO (form (a)), arga is evaluated only if the condition is true.

FORM (a)

System action - a comparison, yielding either a true or false value, is made between the results derived from the evaluation of arga and argc. The comparison made depends upon the value of cond in the following way:

1. If cond= "=" then a character comparison is made. The comparison yields a true value if and only if argb and argc are identical strings of characters.

2. If cond= "BC" then a character comparison is made. If argc= "I", the comparison yields a true value if and only if:

    argb= {letter | digit}.

   If argc= "L", the comparison is true if and only if

    argb= {letter}.

   If argc= "N", the comparison is true if and only if

    argb= {["+" | "-"]} {digit}.

3. If cond= "EN" | "GE" | "GR" , a numerical comparison is made. The comparison is true if and only if the evaluated form of argb is respectively equal to, greater than or equal to, or greater than the evaluated form of argc.

The relationship between del_1 and the value which the comparison yields is described below:

1. If del_1 = "IF" and the comparison yields a false value, then no further action is taken by ML/I. If the comparison is true, then the actions of form (b) are performed.

2. If del_1 = "UNLESS" and the comparison is true, then no further action is taken. If the comparison is false, then the actions of form (b) are performed.

FORM (b) System action

Let "N" be the value derived from the evaluation of arga (remember that arga is a macro_expression). Then the action performed by ML/I depends upon "N" in the following way:

1. If N>0, ML/I changes the point of scan to the point represented by macro label N. There are two ways in which ML/I determines this point:

   a. If macro label N is present in the current environment, ML/I takes the new point of scan as the point associated with the macro label. A macro label is present in the current environment if it has been previously placed in the environment by means of an insert.

   b. If macro label N is not in the current environment, ML/I will initiate a forward search for an insert which places macro label N in the environment. If during the search, a macro call or skip is encountered, ML/I will suspend the search for the macro label and will begin a search for the closing delimiter of the encountered construction. The only inserts which are evaluated during the search for the macro label are those which occur outside any macro calls. During this search, no value text is generated and no macros are called (except possibly during the evaluation of an insert argument). An error message is produced if the end of the current piece of text is found without encountering an insert which places macro label N. Upon encountering the designated label, ML/I will begin the scanning and evaluation of the text at the point immediately following the insert which placed the label. At the same time, the macro label is associated with this point and added to the current environment.

2. If N=0, ML/I discontinues the processing of the current piece of text and continues the processing of the previous piece of text. As a result, a MCGO to a label of "L0" can only occur within inserted text or replacement text. An error message is produced if the user attempts to MCGO to "L0" within the source text.

10.5.4 MCPVAR

Purpose - to allocate extra permanent variables.

General form - "MCPVAR" arga NL

Restrictions - arga= macro_expression.

System action - let N be the result derived from evaluating arga and let M be the number of permanent variables currently in existence. If N>M, N-M extra permanent variables are allocated and added to the environment. These new permanent variables are not assigned initial values.

11. ERROR MESSAGES

11.1 THE CONTEXT OF AN ERROR MESSAGE

Upon detecting an error, ML/I will print an error message on the ERROR file. This message consists of a description of the error together with a print-out of the context in which the error occurred. The context indicates:

1. The line number in which the error occurred. This line number refers to the text which was being scanned at the time the error was detected.

2. A list of all the macro calls and inserts which were in the process of being evaluated. If the name delimiters and the closing delimiters of these macro calls and inserts are not on the same line of text, the line numbers of both the beginning and end of the calls and inserts will be printed.

3. A list of all the arguments of any macro in which an error has occurred. These arguments are printed literally (i.e. not evaluated).

4. When printing a layout character (such as a series of spaces, a newline marker, etc.), the corresponding layout keyword (enclosed in parentheses) is used. See section 9.6 for a list of layout keywords. A piece of null text is printed as "(NULL)".

5. If two atoms of a delimiter were connected by "WITHS" in the structure representation, a space will be inserted between the two atoms when the multi-atom delimiter is printed.

6. If, when printing an error message, ML/I finds that the message is greater than 60 characters in length, the first and last 26 characters (separated by "......   ") will be printed.

11.2 LIST OF ERROR MESSAGES

The following is a list of the error messages produced by ML/I (regardless of the implementation).

11.2.1 ILLEGAL MACRO ELEMENT

Message - flag number IS ILLEGAL MACRO ELEMENT

Description - this error can occur either: (1) when using an insert which references the arguments or delimiters of a macro call and these arguments or delimiters do not exist; or (2) when using number to reference a macro variable which does not exist.

System action - the current operation macro or insert is aborted.

11.2.2 ARITHMETIC OVERFLOW

Message - ARITHMETIC OVERFLOW

Description - this error can occur during the evaluation of a macro expression or a subscript when either: (1) attempted division by zero has been detected; or (2) overflow has occurred (i.e. the calculated number exceeds the implementation-defined maximum value).

System action - the current operation macro or insert is aborted.

11.2.3 ILLEGAL INPUT CHARACTER

Message - ILLEGAL INPUT CHARACTER

Description - a character of the source text is not a member of the implementation's character set.

System action - the illegal character is replaced by "?" (defined to be the error character).

11.2.4 ILLEGAL MACRO NAME

* (not implemented, since there are no warning markers)

Message - ILLEGAL MACRO NAME AFTER WARNING, VIZ atom

Description - an atom which follows a warning marker is neither a macro name, nor the beginning of a multi-atom macro name. If this error occurs within an argument of a normal-scan macro, it will be detected both when the macro call is scanned and when the argument is inserted.

System action - the warning marker is ignored (i.e. it is not copied

over to the value text and the atom which follows it is not treated as a macro name).

11.2.5 UNMATCHED CONSTRUCTIONS

Message - DELIMITER name [{OR name }] OF (MACRO) name
(SKIP)
(INSERT)
IN LINE number OF CURRENT TEXT NOT FOUND

Description - this error occurs when an unmatched construction has been encountered. The line number printed indicates the line in which the construction's name delimiter can be found. The list of delimiters printed indicates the delimiter(s) for which ML/I was searching when the error occurred. This error is detected during the search for a delimiter only upon reaching the end of the scanned text.

Possible causes - there are two conditions most likely to cause this error:

1. A construction nested within an argument or delimiter of the given construction may have caused ML/I to mismatch delimiters (i.e. it may have taken a delimiter of the outer construction to be a delimiter of the inner construction).

2. The construction's structure representation may have been specified incorrectly by the user.

System action - the text, from the construction's name delimiter to the current point of scan, is ignored by ML/I (i.e. the result from evaluating the construction is the null string).

11.2.6 ILLEGAL SYNTAX OF ARGUMENT VALUE

Message - ARGUMENT number HAS ILLEGAL VALUE, VIZ value

Description - the syntax of an argument of an operation macro or insert is not correct. See the appropriate sections for the correct syntax.

System action - the current operation macro or insert is aborted.

11.2.7 REDEFINED LABEL

Message - LABEL number IS MULTIPLY-DEFINED

Description - an insert which placed a label specified a label number not unique to the current environment.

System action - the new label definition is ignored.

11.2.8 UNDEFINED LABEL

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

Description - an attempt has been made to MCGO to an undefined label. This error is detected only when the scanning process reaches the end of the current text (remember that ML/I initiates a forward search for the label if it is not in the current environment).

Possible causes - the user may have attempted a backward MCGO in the source text. Since labels placed in the source text are not added to the environment, the user can MCGO only forward to a label. Another possible cause of error could be the user's attempt to MCGO from one piece of text to another. Unmatched constructions are another source of error. If an unmatched construction is encountered, any MCGO currently being processed will be unable to find the desired label.

System action - ML/I changes the point of scan to the end of the current text.

11.2.9 STORAGE EXHAUSTED

Message - PROCESS ABORTED FOR LACK OF STORAGE [POSSIBLY DUE TO other messages?]

Description - ML/I has used up all the available storage. If the current text is the source text, the messages of 11.1.5 and 11.1.8 will be printed to indicate any unmatched constructions or unfound labels.

Possible causes - storage must be allocated for construction definitions, macro variables, the arguments and delimiters of macro calls and skips (including the arguments and delimiters of nested macros), and the name environment. Thus, an unmatched macro call in the source text (or a call with a very long argument)

can readily use up all the available storage. A very deep nest of calls or an endless recursive call can also cause this problem.

11.2.10 SYSTEM ERROR

Message - SYSTEM ERROR

Description - a machine or operating system error has occurred. An error in the implementation of ML/I may also cause this error.

System action - the current process is aborted.

11.2.11 SUBSIDIARY MESSAGE

Message - (MACRO) name ABORTED BECAUSE OF FOREGOING ERROR
(INSERT)

Description - this message will be printed whenever an error causes an operation macro or insert to be aborted. It is printed in addition to the message which describes the error. A null value is returned whenever a construction is aborted.

11.2.12 STATISTICS

Message - AT END OF PROCESS: number LINES, number CALLS

Description - this message is printed at the end of the processing of the source text. The values printed include the number of source text lines scanned and the number of macro calls performed.

12. HOW TO USE ML/I

Before applying the macro processor to a piece of text, the user must first initialize the name environment. A certain amount of initialization is done by the processor itself. For instance, the operation macro definitions and default values for certain system variables are pre-supplied. But any macro, insert, or skip definitions that the user wishes to employ must be added to the environment before their calls are encountered in the text.

12.1 THE EVALUATION OF SOURCE TEXT

It is recommended that the user append to the start of his source text any construction definitions which he plans to use in the remainder of the text processing. These definitions must be processed before the associated construction calls are encountered. If a large number of definitions are to be processed, then they may be stored in a macro library (see the appendix concerning macro libraries).

Normally, the user will wish to have in the environment certain construction definitions which can be used during the processing of all subsequent text. A good example of such a definition is a literal bracket. The name delimiter of the literal bracket should be an atom (or sequence of atoms) that normally would not appear in the original source text. For example, in pre-processing a PL/I program, the atoms /$ (with $/ as the closing delimiter) would be a good choice for a literal bracket. This particular definition would be

```
MCSKIP MT, / WITH $ $ WITH /
```

Another construction which is useful in the environment is an insert definition. Since macros would be very limited if they did not have the ability to insert arguments (and, in ML/I's case, delimiters) into their replacement text, the user should define an insert definition which can be used in all subsequent source or replacement text processing. The same constraints are imposed on choosing the insert name as with the choice of the name of a literal bracket.

In addition to the two foregoing construction types, the user should define those macros which he wishes to be recognized and expanded. These macros (as well as any previously defined skips and inserts) will be recognized as such during the subsequent evaluation of either the source text or any replacement text or inserted text whose evaluation is brought about due to the evaluation of previous text.

The following text serves as an example of the processing involved in the evaluation of source text. Note that construction definitions usually appear on (but are not constrained to) one line of source text per definition.

```
MCSKIP MT, / WITH $ $ WITH /
MCINS %.
MCDEF # AS /$ WITH SPACE WITH $/
MCDEF INSERT#THE#FIRST#ARGUMENT INSERT#THE#SECOND
#ARGUMENT END AS /$ %A1.
%A2. %D2. $/
THE TEXT WILL NOW BE PROCESSED. THE FIRST MACRO CALL
INSERT THE FIRST ARGUMENT /$ THIS IS THE FIRST ARGUMENT $/
INSERT THE SECOND ARGUMENT /$ THIS IS THE SECOND
ARGUMENT $/  END
MCNODEF
$MCDEF DON WITH ' WITH T RECOGNIZE A SKIP AS /$ %WA1. #
%WA2. # %WA3. $/
TRY THE MACRO DON'T /$ ARG1 $/ RECOGNIZE /$ ARG2 $/ A
/$ ARG3 $/ SKIP. THE TEXT RESULTING FROM THE EVALUATION
OF THIS MACRO (WITH # FOLLOWING EACH OF THE INSERTED
ARGUMENTS) IS:
/$ /$ ARG1 $/ #
/$ ARG2 $/ # /$ ARG3 $/ #  (NOTE: THE MACRO CALL $DON'T
RECOGNIZE A SKIP IS NOT RECOGNIZED WITHIN THIS SKIP) $/
```

The output resulting from the evaluation of the preceding text would be:

```
THE TEXT WILL NOW BE PROCESSED. THE FIRST MACRO CALL
  THIS IS THE FIRST ARGUMENT
 THIS IS THE SECOND
ARGUMENT    END

TRY THE MACRO /$ ARG1 $/ #
/$ ARG2 $/ #
/$ ARG3 $/ . THE TEXT RESULTING FROM THE EVALUATION
OF THIS MACRO (WITH # FOLLOWING EACH OF THE INSERTED
ARGUMENTS) IS:
 /$ ARG1 $/ #
/$ ARG2 $/ # /$ ARG3 $/ #  (NOTE: THE MACRO CALL DON'T
RECOGNIZE A SKIP IS NOT RECOGNIZED WITHIN THIS SKIP)
```

Several points should be mentioned concerning this evaluation.

1. The skip definition MCSKIP MT, / WITH $ $ WITH / is a matched skip with the text option enabled. Any occurrence of this skip will

cause the text between its delimiters to be copied over to the output text with the delimiters dropped. The importance of a matched skip is best illustrated by the last example of this skip. It is imperative that the atoms "$/" of one of the innermost skips not be recognized as the closing delimiter of the outermost skip. Therefore, since the outermost skip is a matched skip, the skips nested within the first skip are matched with their delimiters before the outermost skip is matched with its delimiters. Furthermore, since only other skips are recognized within matched skips, the warning marker and macro call "$ DON'T RECOGNIZE A SKIP" are not recognized as such (note: if this call had been recognized, the macro would have had null arguments).

2. The definition MCDEF # AS /$ WITH SPACE WITH $/ will cause every occurrence of "#" to be replaced by "WITH SPACE WITH". Since the arguments of all operation macros are evaluated, the specification of "INSERT#THE#FIRST#...." will be expanded into "INSERT WITH SPACE WITH THE WITH SPACE WITH FIRST....." and thus results in the intended structure representation. Notice that " WITH SPACE WITH " was enclosed within literal brackets in order to include the leading and trailing blanks.

3. The structure representation of the definition MCDEF INSERT#THE#FIRST..... will be expanded as previously mentioned. The replacement text of the macro is enclosed within literal brackets in order to prevent the attempted insertion of the non-existing arguments and delimiters (since the definition occurs within the source text). Again this situation is due to the evaluation of the arguments of an operation macro. Note that a call of this macro results in the insertion into the output text of the first argument on one line and the second argument and closing delimiter on the next line. This is caused by the occurrence of the newline marker between %A1. and %A2.

   When this macro is called, the arguments will be evaluated (since an insert flag of "A" is used). Thus, the text resulting from the macro call occurring within the source text consists of "THIS IS THE FIRST ARGUMENT.
   THIS IS THE SECOND
   ARGUMENT END"

   Since the arguments were enclosed within literal brackets, their evaluation resulted in the dropping of the enclosing brackets.

4. The macro definition MCDEF DON'T RECOGNIZE A ..... has several interesting features. Again the replacement text must be enclosed within literal brackets in order to prevent the evaluation of the inserts. When the replacement text of the macro is evaluated, the resulting text will consist of the unevaluated first argument followed by "#", the unevaluated second argument followed by "#", and the unevaluated third argument followed by "#". Notice that "#" is no longer recognized as a macro since the operation macro MCNODEF had previously deleted all macro definitions from the local

environment.

12.2 THE EVALUATION OF REPLACEMENT TEXT

The evaluation of replacement text is very similar to the evaluation of source text. There are, however, a few differences which prove to be unique to its evaluation.

In defining a construction definition within the source text, the definition will be applicable to all subsequent text evaluation. However, this is not true for any definition encountered during the evaluation of a piece of replacement text. The new construction will be recognized during the evaluation of any inserted text resulting from the processing of an insert encountered within the replacement text, or during the evaluation of the replacement text of any macros called from within the original replacement text. But upon the completion of the evaluation of the replacement text in which the definition occurred, the definition (as well as any other definitions which occurred in that replacement text) will be deleted from the name environment.

This type of construction definition nesting is analogous to the definition of symbols in a block structured programming language. Any symbol defined at a certain block level will be recognized at that level and within any blocks nested within that block (unless shielded by an inner definition of the same symbol). However, the symbol will no longer be recognized once the block in which it was defined has been terminated. The only difference between this type of symbol definition and the definition of constructions in ML/I is that the same construction name can have several different definitions at the same level of macro nesting. The latest definition will always be used.

The final significant contrast between the evaluation of the two types of text involves the relationship between the insertion of macro labels and the operation macro MCGO. It is legitimate to insert a macro label into the source text but the label itself will not be added to the environment. Thus, MCGO cannot be used to perform a backward macro-time GO TO to a macro label within the source text. However, any macro label inserted into a piece of replacement text will be added to the macro's environment and can be the object of a backward MCGO.

The following example is a possible way of writing the replacement text of the DO macro used in Dr. Brown's original user's manual [3]. The macro name is DO, the second delimiter is TIMES, and the closing delimiter is REPEAT. The constructions defined in the section on source text evaluation will be used (since they apply to the replacement text of any macros called from within the source text).

```
%L1.MCSET T1=%A1.
%A2.
MCSET T1=T1-1
MCGO L1 IF T1 GE 1
```

The macro could be used in conjunction with the following code to compute the factorial of 5.

```
J=1
K=1

DO 5 TIMES
   J=J*K
   K=K+1
REPEAT
```

The expansion of the macro would produce the following code:

```
J=1
K=1
J=J*K
K=K+1
J=J*K
K=K+1
J=J*K
K=K+1
J=J*K
K=K+1
J=J*K
K=K+1
```

12.3 THE EVALUATION OF INSERTED TEXT

Inserted text refers to the arguments and delimiters of a macro call which, because of an insert with a flag of either A, B, or D, may be evaluated and inserted into a piece of replacement text.

The rules pertaining to the evaluation of inserted text are the same as those relating to the evaluation of replacement text.

12.4 FURTHER EXAMPLES

This chapter was intended to serve as a rudimentary guide to the definition of constructions and the use of these constructions (along with the operation macros) in the evaluation of text. For more detailed examples on how to use ML/I, the reader should refer to Dr. Brown's original user manual, particularly chapter 7. For any discrepancies between the original definition of ML/I and my implementation, the exclusion list of chapter 13 should be consulted.

13. EXCLUDED AND INCLUDED FEATURES

13.1 EXCLUSION LIST

The following language features have been excluded from my implementation of ML/I:

1. Exclusive delimiters - the absence of exclusive delimiters prevents a user from using the same closing delimiter in the matching of several nested construction calls. If a construction does not have either a newline or startline marker as its closing delimiter, the omission of exclusive delimiters will not be a serious drawback. The user must simply specify the correct number of closing delimiters when matching nested calls. If a construction has either newline or startline as its closing delimiter (and the construction may be used in a situation where exclusive delimiters will be needed), then it is recommended that the construction be defined with some other closing delimiter.

2. A consequence of not having exclusive delimiters is that a structure representation cannot be followed by a nodego (i.e. NO). Therefore, the last delimiter specification of a structure representation must specify the closing delimiter.

3. The first delimiter specification of a structure representation cannot be an option list - it must be a delimiter name specification. This prevents the user from using a single definition to define several different names for the same construction. The user can achieve the same effect by repeating the construction definition separately for each of the names.

4. Because of the preceding two exceptions, the syntax of a structure representation is now:

   structure_representation= delim_name {del_spec}.

5. When matching a series of atoms against the construction names in the environment, longer environmental names do not necessarily take precedence over shorter names. If two names begin with the same series of atoms, then the processor will attempt to match the most recently defined name first. If the user wishes the longer name to take precedence, its definition should occur after the definition of any other name having the same initial atoms. The same concept

applies to the matching of a secondary delimiter. If several branch names of an option list begin with the same atoms, the processor will first attempt to match the branch names closest to the beginning "OPT" of the option list. The precedence of a longer secondary delimiter within an option list can be forced by defining it as the first branch of the list.

6. A space or a series of spaces cannot be defined as the initial atom of a construction name. This feature was excluded mainly to improve the speed of the processor. However, the user can still specify that a space or sequence of spaces should be searched for during the matching of the secondary atoms of a multi-atom delimiter. Therefore, the name delimiters

    MOVE WITH SPACE WITH SPACE WITH FROM

    and

    MCSUB WITHS (

   are still legal delimiter definitions.

7. All inserts are unprotected. Therefore, the environment used in evaluating a piece of inserted text is the one that was in effect when the insert was encountered, not the environment which was in effect when the call containing the inserted text was encountered.

8. Warning markers are not implemented. If a warning marker were added to the environment (thus placing the environment in warning mode), then macros would be recognized only if they were preceded by a warning marker. The same effect can be achieved by using literal brackets. If the user does not wish a macro to be recognized within a piece of text, then he should enclose the macro name within literal brackets.

13.2 INCLUSION LIST

The following features are present in my implementation but do not appear in the original definition:

1. In the operation macro MCGO, the numerical comparison operators are supplemented by:

   NT - meaning Not Equal

   LT - meaning Less Than

   LE - meaning Less than or Equal to

2. Since the writing of a correct structure representation may be difficult for someone unfamiliar with ML/I, the following error messages pertaining to structure representations have been included in the implementation. Each message is self-explanatory.

   A. AN "OPT" WHICH BEGINS AN OPTION LIST DOES NOT HAVE A TERMINATING "ALL".

   B. TWO NODES OCCUR IN SUCCESSION WITHIN AN OPTION LIST

   C. A NODE OCCURS AFTER THE BEGINNING "OPT" OF AN OPTION LIST

   D. A BRANCH OF AN OPTION LIST DOES NOT HAVE A NAME

   E. TWO OF THE BRANCH NAMES OF AN OPTION LIST CAN MATCH THE SAME PATTERN OF DELIMITERS (I.E. THEY ARE NOT UNIQUE)

   F. A KEYWORD HAS BEEN USED AS A DELIMITER

   G. TWO NON-PUNCTUATION CHARACTERS ARE CONNECTED BY "WITH" WITHIN A DELIMITER NAME SPECIFICATION

   H. A NODE OCCURS WITHIN A DELIMITER NAME SPECIFICATION

   I. A NODE IS MULTIPLY DEFINED WITHIN AN OPTION LIST

   J. A NODEGO WITHIN AN OPTION LIST REFERS TO AN UNDEFINED NODE

   K. THE STRUCTURE AS WRITTEN IS UNCONNECTED

   L. A DELIMITER NAME SPECIFICATION CONTAINS AN ILLEGAL ATOM

   M. A NODEGO FOLLOWS A STRUCTURE REPRESENTATION

   N. WARNING - A NODEPLACE PRECEDES THE SINGLE DELIMITER NAME OF A STRUCTURE REPRESENTATION

14. APPENDICES

The following appendix contains instructions on the use of this implementation. Certain system dependencies (such as the initial values of system variables) are presented.

Appendix B - IBM System/360 Implementation

BIBLIOGRAPHY

[1] Brown, P. J. (1970). ML/I User's Manual, 4th Edition, University of Kent at Canterbury.
Available through:
Computing Laboratory
The University of Kent at Canterbury
Kent, England CT2 7NF

[2] Wirth, N. "What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?". Comm. ACM 20, 11 (Nov. 1977) 822-823. Copyright 1977 - reprinting privileges were granted by permission of the Association for Computing Machinery.

[3] page 2/4, ML/I User's Manual, 4th Edition, University of Kent at Canterbury.

IBM System/360 Implementation

TABLE OF CONTENTS

I. RESTRICTIONS AND ADDITIONS 1

II. OPERATING INSTRUCTIONS AND I/O 1

III. MACRO LIBRARIES 2

IV. RESTRICTING THE CHARACTER POSITIONS WITHIN A RECORD 3

V. PROGRAM PARAMETERS 3

VI. CHARACTER SET 4

VII. ERROR MESSAGES 4

VIII. INTEGER CALCULATIONS 4

IX. LAYOUT KEYWORDS 4

X. S-VARIABLES 5

1. RESTRICTIONS AND ADDITIONS

This implementation of ML/I contains all the features described in the ML/I User's Manual (4th edition, August 1970) with the exception of those features mentioned in the exclusion list of Chap. 13 of the Reference Manual. The inclusion list of that chapter describes several additional features.

2. OPERATING INSTRUCTIONS AND I/O

To run ML/I under OS/360 MVT Release 21.8 - Hasp II Version 3.1 (taking the input form a disk dataset and sending the results to another disk dataset), the following JCL must be used:

```
//MLI EXEC PGM=MLI,PARM='/....,....,....'
//STEPLIB DD DSN=.......,DISP=SHR,UNIT=DISK
//SOURCE DD SYSOUT=A,
// DCB=(RECFM=VB,BLKSIZE=141,LRECL=137)
//SYSPRINT DD SYSOUT=A,
// DCB=(RECFM=VBA,BLKSIZE=141,LRECL=137)
//RESULTS DD DSN=.......,DISP=SHR,UNIT=DISK
//DEBUG DD SYSOUT=A,
// DCB=(RECFM=VB,BLKSIZE=141,LRECL=137)
//PLIDUMP DD SYSOUT=A
//INPUT DD DSN=.......,DISP=SHR,UNIT=DISK
```

If no parameters are used, the EXEC card is simply

```
//MLI EXEC PGM=MLI
```

(see section 5 for a list and description of the program parameters).

The following files are used by the macro processor:

1. INPUT - the file containing the source text to be evaluated.
2. SOURCE - the file on which the listing of the source text is printed.

3. RESULTS - the file on which the result of the evaluation of the source text is printed.

4. SYSPRINT - the system print file (also used to hold the results of the evaluation of the source text).

5. DEBUG - the file on which all error messages are printed.

Output can be suppressed on any of the four output files (SOURCE, RESULTS, SYSPRINT, DEBUG) by specifying

```
//filename DD DUMMY
```

where "filename" is the particular file not to be used for output.

Output may be controlled during processing by changing the values of S20 - S22 (via the operation macro "MCSET").

```
S20 = 0 turns SYSPRINT listing off
      1  "      "        "       " on
S21 = 0 turns RESULTS listing off
      1  "      "       "      " on
S22 = 0 turns SOURCE listing off
      1  "      "      "      " on
```

The initial values of the S-variables are described in section 10.

3. MACRO LIBRARIES

If a dataset containing a set of construction definitions (i.e. a macro library) is to be used, then the INPUT card might appear as:

```
//INPUT DD DSN=MACRO.LIB,DISP=SHR,UNIT=DISK
//      DD DSN=MLI.INPUT,DISP=SHR,UNIT=DISK
```

where "MACRO.LIB" is the macro library and "MLI.INPUT" contains the source text to be evaluated.

4. RESTRICTING THE CHARACTER POSITIONS WITHIN A RECORD

The S-variables S11 and S12 can be set to restrict the character positions within a line of input. This is generally used for card input to ignore (i.e. not evaluate) the first column and the last eight columns (73 to 80) of the record, but it may be used on any input device. The processor ignores a character unless

$$S11 \leq \text{column number} \leq S12$$

where 'column number' means 'the character position within a record'. Initially, S11 has the value 2 and S12 is set to 72 (note: if S12<S11, then a newline marker is the only character evaluated for each input record).

5. PROGRAM PARAMETERS

The following parameters may be used when calling ML/I

Parameter		Default Value
LRECL -	the length of an input record	80
SYSVAR# -	the number of system variables to be added to the environment	24
HT_SIZE -	the size of the local and global construction definition hash tables	31
S11 -	the value of system variable 11	2
S12 -	the value of system variable 12	72
S22 -	the value of system variable 22	1

An example of parameter usage is

```
//MLI EXEC PGM=MLI,PARM='/LRECL=133,S11=1,S12=133'
```

which causes input records of 133 bytes to be read in by the macro processor. Furthermore, evaluation will begin in column 1 and end in column 133. All input records must be of fixed length.

6. CHARACTER SET

The character set is the full EBCDIC character set (including lower case letters and control characters).

7. ERROR MESSAGES

Error messages are sent to the file DEBUG. In printing any text (such as arguments and delimiters) within an error message, only the first 50 characters of the text (followed by ".....") will be printed.

8. INTEGER CALCULATIONS

The initial environment contains ten permanent variables and (as a default) 24 system variables. All permanent variables are initialized to zero. All integers in, or derived from macro expressions, have a precision of 31 bits. Overflow is detected and reported as an error.

9. LAYOUT KEYWORDS

The following layout keywords are used in this implementation:

SPACE	meaning a space
SPACES	meaning a sequence of one or more spaces
NL	meaning a newline
SL	meaning a startline marker

10. S-VARIABLES

There are 24 S-variables available. This number can be increased (but not decreased) through the use of the program parameter "SYSVAR#". The meanings and initial values of these variables are the following.

Variable		Initial Value
S1	not used	0
S2	the source line count	1
S3	not used	0
S4	if 0 then print the context of an error message after a call of "MCNOTE"	0
S5-S10	not used	0
S11	starting column	2
S12	ending column	72
S13-S19	not used	0
S20	(see section 2)	1
S21	(see section 2)	1
S22	(see section 2)	1
S23-S24	not used	0

ML/I Logic Manual

TABLE OF CONTENTS

I. INTRODUCTION 1

II. SYSTEM ORGANIZATION 3

2.1. Initialization Routines 3
2.2. Scanning Routines 3
2.3. Symbol Table Routines 4
2.4. Operation Macro Routines 4
2.5. Construction Routines 4
2.6. Text Forming Routines 4
2.7. Storage Releasing Routines 5
2.8. Evaluation Routines 5
2.9. Printing Routines 5
2.10. Macro Expression Routines 5
2.11. Error Routines 5
2.12. Statistical Information 6

III. MODULES 7

3.1. MLI 8
3.2. PUNCINT 8
3.3. INIT 8
3.4. SCANSRC 8
3.5. SCANNER 8
3.6. SCAN 8
3.7. MATCHNM 9
3.8. DELS 10
3.9. ALTRDEL 11
3.10. LOCDEF 11
3.11. GLOBAL 16
3.12. MATCH 19
3.13. PRINT 19
3.14. FORMTXT 19
3.15. OUTPUT 20
3.16. MACRO 21
3.17. SKIP 21
3.18. INSERT 21
3.19. INSTEXT 22
3.20. ARGFREE 22
3.21. MCINS 23
3.22. MCSKIP 23
3.23. MCDEF 24
3.24. MCNO 24
3.25. MCALTER 24

3.26. MCLENG 25
3.27. MCSUB 25
3.28. MCSET 26
3.29. MCNOTE 26
3.30. MCGO 26
3.31. MCPVAR 27
3.32. DELETE 27
3.33. EVALARG 28
3.34. RELEASE 28
3.35. STRUC 28
3.36. MACEXPR 33
3.37. TERM 33
3.38. PRIMARY 34
3.39. OPERAND 34
3.40. MACVAR 35
3.41. FETCH 35
3.42. PRNTDEL 36
3.43. PRINTXT 36
3.44. CONTEXT 36
3.45. MCERRS 36
3.46. ERR1 37
3.47. ERR2 37
3.48. ERR3 38
3.49. ERROR 38
3.50. STATS 38

IV. EXTERNAL VARIABLES 39

4.1. Secondary Delimiters 39
4.2. Reserved Keywords 41
4.3. Layout Keywords 43
4.4. Character Sets 44
4.5. Input/Output Variables 44
4.6. Environmental Variables 46
4.7. Scanning Variables 48

V. SYSTEM FILES 51

5.1. INPUT 51
5.2. SYSPRINT 51
5.3. RESULTS 51
5.4. SOURCE 51
5.5. DEBUG 51

LIST OF FIGURES

Figure 1: Name Table Associations 14

Figure 2: Threaded Lists 15

Figure 3: Global Symbol Table 18

Figure 4: A Structure with Fixed Delimiters 30

Figure 5: A Structure containing an Option List 31

Figure 6: Structure with an Option List, Nodeplaces, and Nodegos . . 32

1. INTRODUCTION

This manual describes the organization and external interfaces of the various modules which constitute the ML/I macro processor. The manual does not explain the algorithms or internal characteristics of the programs, but relies on the internal documentation of each program for an explanation of the module's behavior.

In describing the individual modules of the macro processor, the basic function and external interfaces of each module will be stated. Diagrams illustrating certain data structures (such as the symbol table or certain directed graphs) are provided to assist in the modification of the routine.

The ML/I macro language itself is described in the language reference manual. Briefly, ML/I is a nested, recursive, conditional macro language in which multi-atom names and delimiters may be defined by the user. The language can be classified as general purpose since it accepts any form of input.

2. SYSTEM ORGANIZATION

The basic function of the macro processor is to scan the source text and evaluate each atom as it is scanned. The evaluation of an atom determines whether the atom is the beginning of a local or global construction (i.e. macro, skip, or insert). If a construction is found, then the construction is matched with its delimiters and the appropriate action is taken. The process is recursive since both the replacement text and the arguments of a macro may be scanned and evaluated.

This section describes the system organization by classifying routines according to their functions. Subsequent sections describe the external interfaces of each module, external variables, and the files used during the processing of text.

2.1 INITIALIZATION ROUTINES

MLI INIT PUNCINT

Before evaluating the source text, the macro processor must initialize the macro-time environment. The initialization process includes: reading in the run-time parameters, allocating and initializing the system variables, allocating the permanent variables, initializing the number of temporary variables to zero, initializing the reserved keywords, the layout keywords, the node flag, and the secondary delimiters of the operation macros, initializing such special characters as the end of text marker (OMEGA), the newline marker (NL), etc., initializing the character sets, setting flags to be used during subsequent text evaluation, allocating the local and global symbol tables, and adding the operation macro definitions to the local symbol table.

2.2 SCANNING ROUTINES

SCANSRC SCANNER SCAN MATCHNM DELS OUTPUT ALTRDEL

After the environment is initialized, SCANSRC is called to scan and evaluate the source text. In evaluating any piece of text, SCANNER and SCAN are called repetitively. SCANNER returns the next atom from a piece of text and SCAN evaluates this atom. If SCAN determines that an atom matches the first atom of a possibly multi-atom construction name, then MATCHNM is called in an attempt to match the remaining atoms of the construction name. If a match is discovered, then the routine DELS will find the remaining delimiters of the construction. Note that when a construction name is matched with its subsequent delimiters, the construction is not necessarily evaluated. For instance, constructions

nested within the arguments of a macro are evaluated only when the argument is inserted. If a matched construction is not to be evaluated, the routine OUTPUT is called to transmit the arguments and delimiters to the appropriate medium.

2.3 SYMBOL TABLE ROUTINES

LOCDEF GLOBAL MATCH MCNO STRUC

There are three routines which manipulate the symbol table: LOCDEF, GLOBAL, and MCNO. LOCDEF adds a new construction definition to the local symbol table, GLOBAL adds a new definition to the global symbol table, and MCNO deletes entries from the local table. The routine MATCH determines whether a construction name is already in a symbol table and the routine STRUC constructs a directed graph to be used in searching for the arguments and delimiters of constructions contained in the symbol tables.

2.4 OPERATION MACRO ROUTINES

MCINS MCDEF MCSKIP MCNO MCALTER MCLENG
MCSUB MCNOTE MCGO MCPVAR MCSET

The names of the operation macro routines are self-explanatory in that they correspond to the operation macros described in the reference manual. The only difference is the routine MCNO which implements the operation macros MCNODEF, MCNOINS, and MCNOSKIP.

2.5 CONSTRUCTION ROUTINES

MACRO SKIP INSERT INSTEXT

As with the operation macros, the routines which implement the various constructions are given the obvious names MACRO, SKIP, and INSERT. For instance, MACRO performs the actions associated with the processing of a macro. The routine INSTEXT is used to insert either an argument or a delimiter of a macro into the output text.

2.6 TEXT FORMING ROUTINES

FORMTXT

The routine FORMTXT forms a linked list of text by concatenating an atom to a piece of previously formed text. Linked lists of text are used frequently within the macro processor. Such lists are used in forming: the arguments and delimiters of construction calls, the replacement text of macros, the names of constructions, etc.

2.7 STORAGE RELEASING ROUTINES

RELEASE ARGFREE

Linked lists of text, containing various types of information, can quickly deplete available storage. Such a situation can occur when the arguments of a construction are being stored. If the construction is called several times during the evaluation of text, then the linked lists containing the arguments of the various activations of the construction may occupy considerable space. Thus, storage must be released once it is no longer needed.

The routine RELEASE releases the based storage containing the arguments and delimiters of a construction which has just been processed. ARGFREE is used to release the storage containing the evaluated argument of a construction call.

2.8 EVALUATION ROUTINES

SCAN EVALARG DELETE

Evaluation involves the comparison of an atom against the construction names contained within the local and global symbol tables. The routine SCAN performs this comparison and takes the appropriate actions when a match is found. The module EVALARG evaluates the arguments of an operation macro and the routine DELETE deletes leading and trailing blanks from a piece of text before it is evaluated.

2.9 PRINTING ROUTINES

PRINT

The routine PRINT prints an atom on the output files SYSPRINT and RESULTS.

2.10 MACRO EXPRESSION ROUTINES

MACEXPR TERM PRIMARY OPERAND MACVAR FETCH

The preceding routines are used in the evaluation of macro expressions.

2.11 ERROR ROUTINES

ERROR ERR1 ERR2 ERR3
MCERRS CONTEXT PRINTXT PRNTDEL

Errors reported during macro processing are those relating to macro expressions, illegal arguments, erroneous structure representations, and unfound closing delimiters. The reference manual provides more

information concerning these error messages.

2.12 STATISTICAL INFORMATION

STATS

The routine STATS prints statistical information at the end of the processing of the source text. This information includes the number of source lines scanned, the number of macros called, and the constructions remaining in the environment at the end of the text processing.

3. MODULES

The macro processor consists of 50 external routines. The division of the entire program into external routines was formed, for the most part, on the basis of function. For example, each of the operation macros has a corresponding module designed to implement that operation macro. There are modules which process a macro, print error messages, evaluate arguments, and form a sequence of text.

There are two obvious interfaces to any external routine; the interface as specified by the formal parameters of a routine, and the interface suggested by the use of external variables. A subsequent section of the manual describes the external variable interface; this section illustrates the various module interfaces. In describing the module interfaces, each module is listed together with a statement of the function of the module, the formal parameters of the module, and lists of those routines which call the module and those routines called by the module.

3.1 MLI

Function: the main routine for the ML/I macro processor. This routine initializes the environment to be used in all subsequent text evaluation.

Parameters:

1. LRECL - the maximum length of an input card.
2. SYSVAR# - the number of system variables to be added to the environment.
3. HT_SIZE - the size of the hash tables.
4. S11 - the value of system variable 11.
5. S12 - the value of system variable 12.
6. S22 - the value of system variable 22.

Called by: (none)

Calls: INIT, PUNCINT, SCANSRC, STATS

3.2 PUNCINT

Function: initializes the set of punctuation characters.

Parameters: (none)

Called by: MLI

Calls: (none)

3.3 INIT

Function: adds the operation macro definitions to the local name environment.

Parameters:

1. STRUC_REP - the structure representation of an operation macro.

Called by: MLI

Calls: FORMTXT, LOCDEF, STRUC

3.4 SCANSRC

Function: scans and evaluates the source text.

Parameters: (none)

Called by: MLI

Calls: SCAN, SCANNER

3.5 SCANNER

Function: returns the next atom from a piece of text.

Parameters:

1. SRCPTR - points to the text card being scanned.
2. COL - the column in which the scan for the next atom of the text will resume.
3. ATOM - the atom to be returned.
4. SCANTYPE - the type of the scan.

5. TEMPCARD - the card currently being scanned.

Called by: SCANSRC, MATCHNM, DELS, LOCDEF, GLOBAL, MATCH, OUTPUT, MACRO, SKIP, INSERT, INSTEXT, MCALTER, MCSUB, MCNOTE, MCGO, EVALARG, STRUC, FETCH, PRNTDEL, PRINTXT

Calls: ARGFREE, CONTEXT, SCANNER

3.6 SCAN

Function: evaluates an atom to see whether it is the beginning of a construction name.

Parameters:

1. ATOM - the atom to be evaluated.

2. SRCPTR - indicates the input medium.

3. TEXTPTR - points to the piece of text in which the evaluated atom is to be stored.

4. OLDLNGTH - the length of the text card in which the evaluated atom is to be stored.

5. COL - the column in which the scan for the next atom of the text will resume.

6. TEMPCARD - the card currently being scanned.

Called by: SCANSRC, DELS, MACRO, INSERT, INSTEXT, EVALARG

Calls: MATCHNM, DELS, MACRO, SKIP, INSERT, MCINS, MCSKIP, MCDEF, MCNO, MCALTER, MCLENG, MCSUB, MCSET, MCNOTE, MCGO, MCPVAR, OUTPUT, FORMTXT, PRINT

3.7 MATCHNM

Function: matches the remaining atoms of a possibly multi-atom delimiter.

Parameters:

1. ATOM - an atom of the scanned text.

2. SRCPTR - points to the text in which the subsequent atoms of the delimiter may be found.

3. COL - the column in which the scan for the next atom of the

delimiter will resume.

4. TEMPCARD - the text card currently being scanned for the atoms of the delimiter.

5. MATCHED - flag indicating that the procedure matched a delimiter.

6. DELIM_NAME - points to the delimiter name specification for the delimiter which the procedure is trying to match.

7. DELIM - points to the head of the linked list of text containing the delimiter which the procedure has matched.

8. LAST_ATOM - the last atom scanned during the matching of a delimiter.

Called by: DELS, SCAN

Calls: FORMTXT, SCANNER

3.8 DELS

Function: finds the remaining delimiters of a construction call.

Parameters:

1. ALPHA - points to the head of the directed graph to be used in searching for the delimiters of the construction.

2. ARGHEAD - points to the head of the argument list of the construction.

3. DELHEAD - points to the head of the delimiter list of the construction.

4. #_OF_ARGS - the number of arguments of the construction.

5. SRCPTR - points to the text currently being scanned.

6. COL - the column in which the scan for the next atom of the text will resume.

7. TEMPCARD - the card currently being scanned.

8. ERR - flag indicating that the end of the text was reached before the construction's closing delimiter was found.

9. CONSTRUC_TYPE - the type of construction for whose delimiters are being searched.

10. CONSTRUC_NAME - the name of the construction.

11. LEVEL - indicates whether the construction is an operation macro (i.e. has a nesting level of -1).

12. LAST_ATOM - the last atom scanned during the matching of a delimiter.

13. MTSKIP - flag indicating whether the construction whose arguments and delimiters are being formed is nested within a matched skip.

14. STRSCAN - flag indicating whether the construction whose arguments and delimiters are being formed is a straight-scan macro.

Called by: SCAN

Calls: ALTRDEL, ERR3, FORMTXT, MATCHNM, SCAN, SCANNER

3.9 ALTRDEL

Function: converts the representation of an operation macro's secondary delimiter to the delimiter's current value.

Parameters:

1. DELATOM - the secondary delimiter of an operation macro.

Called by: DELS

Calls: (none)

3.10 LOCDEF

Function: adds a construction name to the local environment.

Parameters:

1. NAMEPTR - points to the text containing the new construction name.

2. CONSTRUCQ - qualifier for "CONSTRUC".

Called by: INIT, MCDEF, MCINS, MCSKIP

Calls: MATCH, SCANNER

Resultant Data Structure:

LOCDEF builds the symbol table of local construction names. The local symbol table consists of a hash table (containing two pointers) and three types of doubly linked lists. The operations performed on the symbol table are table look-up, insertion, and deletion. The doubly

linked lists of the symbol table are necessary in order to provide for the easy deletion of table entries.

The symbol table resulting from this procedure is utilized by SCAN in matching an atom with a construction name, by MCNO in deleting constructions from the environment, and by STATS in providing statistical information (such as the current local constructions) at the end of the evaluation of the source text.

The following examples illustrate the data structures which may be associated with a particular hash table entry. In this example, the construction names "MCSUB (", "X (", and "X;" have all hashed to the same location. The construction "MCSUB (" is an operation macro and was the first construction associated with this entry. Next the skip "X (" was entered at block level 0 (i.e. in the source text). This definition was followed by the insert definition "X (" also occurring within the source text. The last entry made from the source text was the macro "X;". Finally, in block level 1 the skip "X (" was entered and in block level 2 the macro "X;" was entered. The example illustrates the data structure as it would appear during the evaluation of text at block level 2. For simplicity, the example has been split into two parts. The first part (figure 1) shows the name table associated with the hash table entry and the constructions associated with each name table entry. The second part (figure 2) shows the threaded lists used in determining the most recent construction entries (see the internal documentation of LOCDEF for further details).

The format of the data structure is:

```
HASH TABLE
ENTRY
-----------------------
|            |         |
| THRD_HEAD  | NT_HEAD |------------> NAME TABLE ENTRIES
|            |         |
-----------------------
                                       |        |        |
                                       |        |        |
                                       |        |        |
                                       |        |        |
                                       V        V        V

                                     CONSTRUCTIONS ASSOCIATED WITH
                                     THE NAME TABLE ENTRIES
```

where each construction entry is of the form:

```
 -------------------------------------------
|         |              |                  |
|   UP    | CONSTRUCTION | BLOCK LEVEL OF   |
| POINTER |     TYPE     | THE DEFINITION   |
|_________|______________|__________________|
|                                           |
|         CONSTRUCTION INFORMATION          |
|___________________________________________|
|             |              |              |
|  BACKWARD   |     DOWN     |   FORWARD    |
|   THREAD    |   POINTER    |   THREAD     |
|_____________|______________|______________|
```

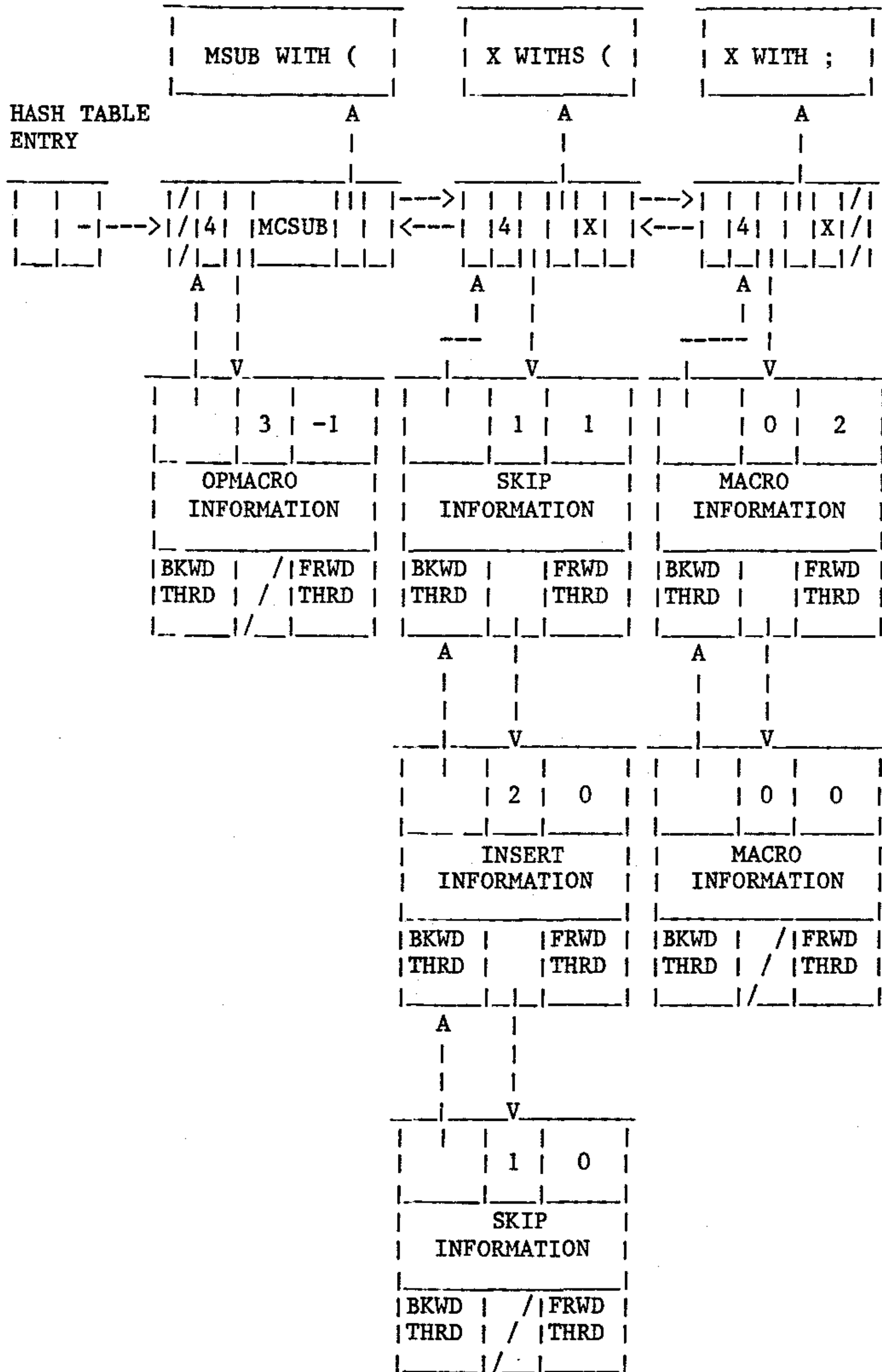


Figure 1. Name Table Associations

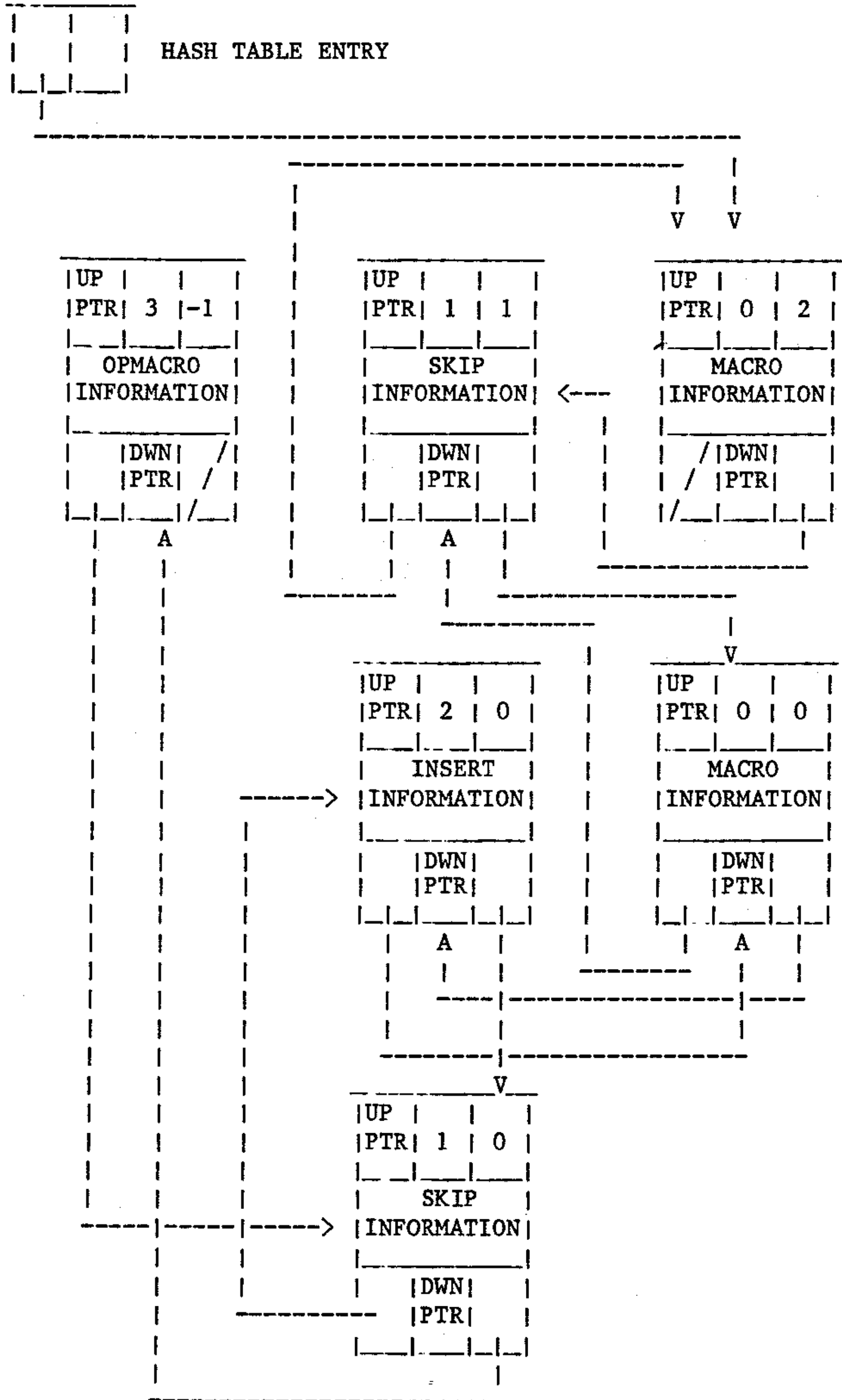


Figure 2. Threaded Lists

3.11 GLOBAL

Function: adds a construction name to the global environment.

Parameters:

1. NAMEPTR - points to the text containing the new construction name.
2. GLOBAL_CONSTRUCQ - qualifier for "GLOBAL_CONSTRUC".

Called by: MCDEF, MCINS, MCSKIP

Calls: MATCH, SCANNER

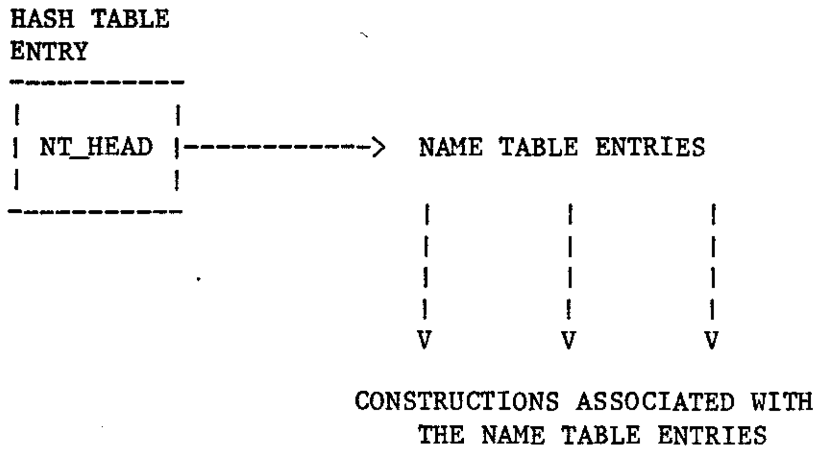
Resultant Data Structure:

Since the only operations performed on the global symbol table are table lookup and insertion, the data structure used to represent the global symbol table is much simpler than the one used to represent the local symbol table. The global symbol table consists of a hash table in which each entry contains a single pointer, and two types of singly linked lists.

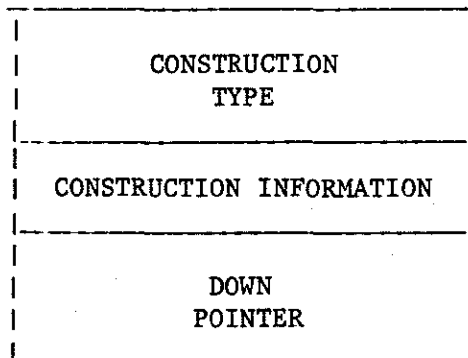
The symbol table resulting from this procedure is used by SCAN in matching an atom with a construction name and by STATS in providing statistical information (such as the current global constructions) at the end of the source text evaluation.

The following example displays a hash table entry (with associated linked lists) as it might appear after the processing of at least 3 different nesting levels of text evaluation.

The format of the data structure is:



where each construction entry is of the form:



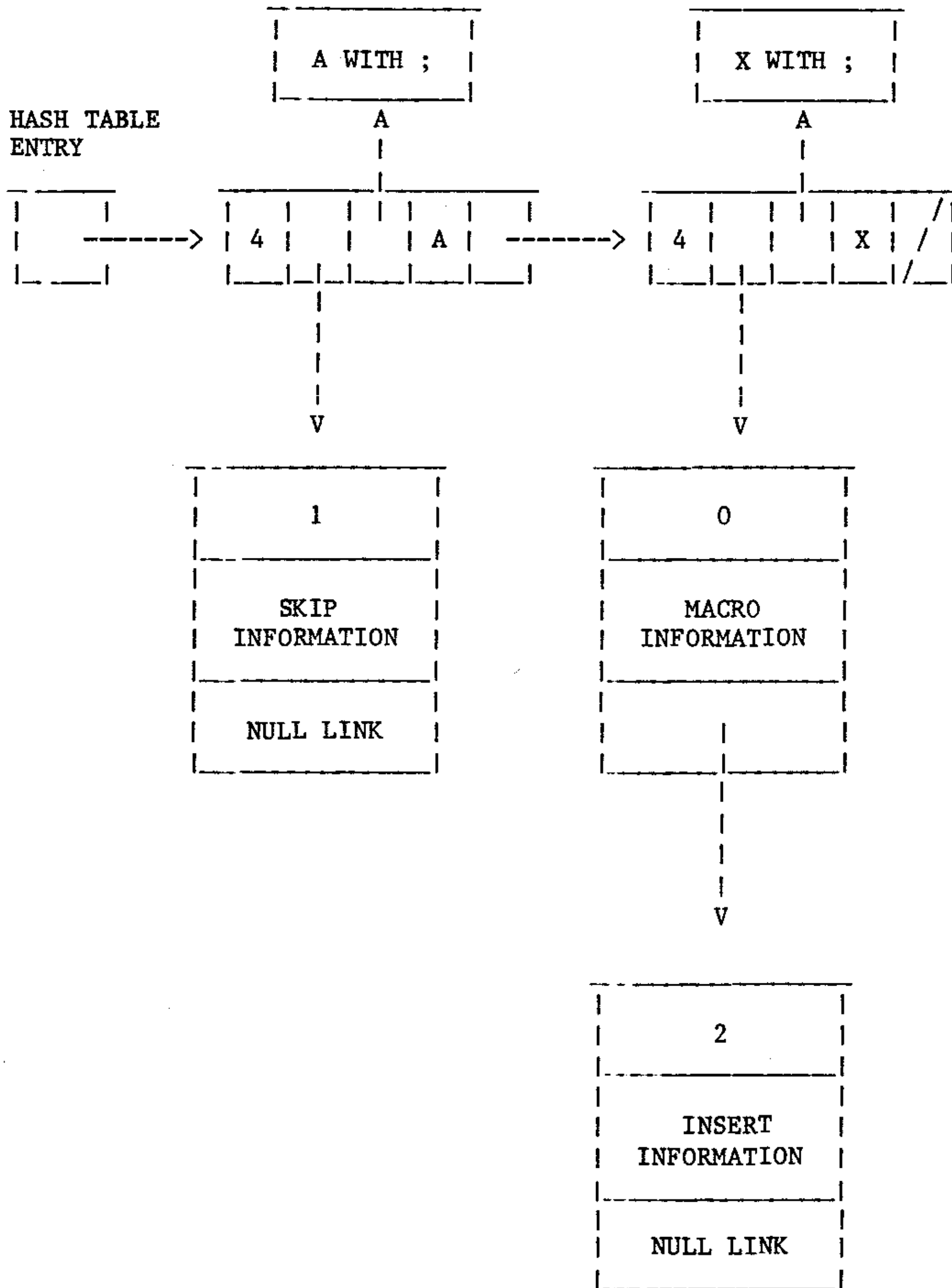


Figure 3. Global Symbol Table

3.12 MATCH

Function: compares two construction names to see whether they match.

Parameters:

1. NAMEPTR# - points to the text containing the new construction name currently being compared against each entry in a name table.

2. CONPTR# - points to the text containing the construction name associated with a name table entry.

Called by: GLOBAL, LOCDEF

Calls: SCANNER

3.13 PRINT

Function: prints an atom on the output file.

Parameters:

1. ATOM - the atom to be printed.

Called by: SCAN, SKIP, INSERT, INSTEXT, MCLENG, MCSUB

Calls: (none)

3.14 FORMTXT

Function: adds an atom to a piece of text being formed.

Parameters:

1. ATOM - the atom to be added.

2. CARDLNGTH - the length of the text card that is currently being formed.

3. TQ# - a pointer to the piece of text in which the atoms are currently being stored.

Called by: DELS, EVALARG, INIT, INSERT, INSTEXT, MATCHNM, MCLENG, MCSUB, OUTPUT, SCAN, SKIP, STRUC

Calls: (none)

3.15 OUTPUT

Function: transmits the delimiters and arguments of a construction call to a sequence of text (maintained internally as a linked list).

Parameters:

1. TEXTPTR - points to the text card currently being used to store the argument of the outermost construction.

2. OLDLNGTH - the length of the text card currently being used to store the argument of the outermost construction.

3. ARGHEAD - points to the head of the argument list of the nested construction.

4. DELHEAD - points to the head of the delimiter list of the nested construction.

5. #_OF_ARGS - the number of arguments of the nested construction.

Called by: SCAN

Calls: FORMTXT, RELEASE, SCANNER

3.16 MACRO

Function: processes a macro call.

Parameters:

1. #_OF_ARGS - the number of arguments of the macro.

2. ARGHEAD - points to the head of the argument list of the macro.

3. DELHEAD - points to the head of the delimiter list of the macro.

4. TEXTPTR - indicates the output medium.

5. OLDLNGTH - the length of the text card in which the atoms of the evaluated replacement text are to be stored.

6. MACROQ - points to the structure containing the information necessary for the processing of the macro.

7. LOCAL - flag indicating whether the macro's definition is local or global.

Called by: SCAN

Calls: MCNO, RELEASE, SCAN, SCANNER

3.17 SKIP

Function: processes a skip.

Parameters:

1. #_OF_ARGS - the number of arguments of the skip.

2. ARGHEAD - points to the head of the argument list of the skip.

3. DELHEAD - points to the head of the delimiter list of the skip.

4. TEXTPTR - indicates the output medium.

5. OLDLNGTH - the length of the text card in which the text resulting from the skip is to be stored.

6. SKIPQ - points to the structure containing the information necessary for the processing of the skip.

7. LOCAL - flag indicating whether the skip's definition is local or global.

Called by: SCAN

Calls: FORMTXT, PRINT, RELEASE, SCANNER

3.18 INSERT

Function: processes an insert.

Parameters:

1. ARGHEAD - points to the head of the argument list of the insert.

2. DELHEAD - points to the head of the delimiter list of the insert.

3. TEXTPTR - indicates the output medium.

4. OLDLNGTH - the length of the text card in which the text resulting from the insert is to be stored.

5. PREVSRCPTR - indicates the input medium.

6. PREVCOL - the column (within the card that was being scanned when the insert was encountered) where the search for the atom immediately following the insert will begin.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, FORMTXT, INSTEXT, MACEXPR, MCERRS, MCNO, PRINT, PRINTXT, RELEASE, SCAN, SCANNER

3.19 INSTEXT

Function: inserts either an argument or a delimiter of a macro call into the appropriate output medium.

Parameters:

1. INARG - indicates whether the text to be inserted is an argument or a delimiter.
2. EVALUATE - indicates whether the text is to be evaluated before it is inserted.
3. DELETE_SPACES - indicates whether leading and trailing spaces should be deleted from the text before it is inserted.
4. N - indicates the argument or delimiter to be inserted.
5. TEXTPTR - indicates the output medium.
6. OLDLNGTH - the length of the text card in which the value of the insert is to be placed.
7. ERR - indicates that an error occurred during the attempted insertion of a piece of text.

Called by: INSERT

Calls: DELETE, FORMTXT, MCNO, PRINT, SCAN, SCANNER

3.20 ARGFREE

Function: frees the storage associated with an evaluated argument of a construction call.

Parameters:

1. HEADARG - points to the sequence of text cards used to store an evaluated argument of a construction call.

Called by: INSERT, MCDEF, MCINS, MCSKIP, SCANNER, MCALTER, MCLENG, MCSUB, MCSET, MCNOTE, MCGO, MCPVAR

Calls: (none)

3.21 MCINS

Function: processes an insert definition.

Parameters:

1. OPMAC - the name of the operation macro (either "MCINS" or "MCINSG").

2. ARGHEAD - points to the head of the argument list of the operation macro.

3. DELHEAD - points to the head of the delimiter list of the operation macro.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, ERROR, EVALARG, GLOBAL, LOCDEF, RELEASE, STRUC

3.22 MCSKIP

Function: processes a skip definition.

Parameters:

1. OPMAC - the name of the operation macro (either "MCSKIP" or "MCSKIPG").

2. ARGHEAD - points to the head of the argument list of the operation macro.

3. DELHEAD - points to the head of the delimiter list of the operation macro.

4. #_OF_ARGS - the number of arguments of the operation macro.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, ERROR, EVALARG, GLOBAL, LOCDEF, RELEASE, STRUC

3.23 MCDEF

Function: processes a macro definition.

Parameters:

1. OPMAC - the name of the operation macro (either "MCDEF" or "MCDEFG").

2. ARGHEAD - points to the head of the argument list of the operation macro.

3. DELHEAD - points to the head of the delimiter list of the operation macro.

4. #_OF_ARGS - the number of arguments of the operation macro.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, ERROR, EVALARG, GLOBAL, LOCDEF, MACEXPR, MCERRS, RELEASE, STRUC

3.24 MCNO

Function: deletes construction definitions from the local name environment.

Parameters:

1. OPMAC - indicates whether a particular construction or all constructions defined at the current level of text evaluation should be deleted.

Called by: EVALARG, INSERT, INSTEXT, MACRO, SCAN

Calls: (none)

3.25 MCALTER

Function: alters either the secondary delimiters of operation macros or the keywords used in structure representations.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, EVALARG, RELEASE, SCANNER

3.26 MCLENG

Function: determines the length of its argument.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

3. TEXTPTR - indicates the output medium.

4. OLDLNGTH- the length of the text card in which the value of the operation macro is to be inserted.

Called by: SCAN

Calls: ARGFREE, EVALARG, FORMTXT, PRINT, RELEASE

3.27 MCSUB

Function: accesses a substring.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

3. TEXTPTR - indicates the output medium.

4. OLDLNGTH- the length of the text card in which the value of the operation macro is to be inserted.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, EVALARG, MACEXPR, MCERRS, RELEASE, SCANNER, PRNTXT, PRINT

3.28 MCSET

Function: implements the macro-time assignment statement.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, EVALARG, FETCH, MACEXPR, MCERRS, OPERAND, RELEASE

3.29 MCNOTE

Function: generates a user defined error message.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

3. SRCPTR - points to the text that was being scanned at the time the call of "MCNOTE" was encountered.

Called by: SCAN

Calls: ARGFREE, CONTEXT, EVALARG, RELEASE, SCANNER

3.30 MCGO

Function: implements the macro-time "GO TO" statement.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

3. #_OF_ARGS - the number of arguments of the operation macro.

4. PREVSRCPTR - points to the text that was being scanned at the time the call of "MCGO" was encountered.

5. PREVCOL - the column within the scanned text where the scan for the next atom following "MCGO" will resume.

6. PREVTEMPCARD - the card that was being scanned at the time the call of "MCGO" was encountered.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, EVALARG, MACEXPR, MCERRS, RELEASE, SCANNER, SCAN

3.31 MCPVAR

Function: allocates extra permanent variables.

Parameters:

1. ARGHEAD - points to the head of the argument list of the operation macro.

2. DELHEAD - points to the head of the delimiter list of the operation macro.

Called by: SCAN

Calls: ARGFREE, CONTEXT, ERR2, EVALARG, MACEXPR, MCERRS, RELEASE

3.32 DELETE

Function: provides for the deletion of trailing blanks from a piece of text.

Parameters:

1. ARG - pointer to the head of a linked list of text whose trailing blanks are to be deleted.

2. PTR - pointer to the text card in the linked list which contains the last non-blank atom.

3. LAST_POSITION - the length of the text card with deleted trailing blanks.

Called by: INSTEXT, EVALARG

Calls: (none)

3.33 EVALARG

Function: evaluates an argument of an operation macro.

Parameters:

1. TEMP - points to the head of a text card sequence containing an argument to be evaluated.

Called by: MCALTER, MCDEF, MCGO, MCINS, MCLENG, MCNOTE, MCPVAR, MCSET, MCSKIP, MCSUB

Calls: DELETE, FORMTXT, MCNO, SCAN, SCANNER

3.34 RELEASE

Function: releases the storage associated with the arguments and delimiters of a construction call.

Parameters:

1. #_OF_ARGS - the number of construction arguments.

2. ARGHEAD - points to the head of the construction argument list.

3. DELHEAD - points to the head of the construction delimiter list.

Called by: INSERT, MACRO, MCALTER, MCDEF, MCGO, MCINS, MCLENG, MCNOTE, MCPVAR, MCSET, MCSKIP, MCSUB, OUTPUT, SKIP

Calls: (none)

3.35 STRUC

Function: transforms a structure representation into a directed graph which can be used to find the delimiters of a construction call.

Parameters:

1. STRUC - the structure representation used to construct the directed graph.

2. ERR - flag used to indicate an illegal structure representation.

3. NAME - points to the delimiter specification of the construction name.

4. #_OF_DELNMS - the number of delimiter name specifications within the structure representation.

5. #_OF_OPTS - the number of option lists within the structure representation.

Called by: MCDEF, MCINS, MCSKIP

Calls: ERROR, FORMTXT, SCANNER

RESULTANT DATA STRUCTURE: The directed graph must be constructed so as to allow for option lists, option lists nested within other option lists, nodegos, and fixed delimiters. It must also take into consideration the specification of multi-atom delimiters.

The graph resulting from this procedure will be applied in the routine DELS during the search for the delimiters of a construction call. Thus, if any modifications are made to the structure of the graph, corresponding alterations must be made within DELS.

The following are examples of structure representations and the graphs built by STRUC from these representations. The first example (figure 4) shows a structure with fixed delimiters, the next (figure 5) shows a structure containing an option list, and the last example (figure 6) depicts a structure with an option list, nodeplaces, and nodegos. Note that, when building a graph, the specification of the construction's name delimiter is not included as part of the graph but is returned separately (pointed to by the parameter NAME). The head of the graph (pointed to by ALPHA) is actually the second delimiter specification of the structure representation.

EXAMPLE (1): INTERCHANGE WITHS (,) WITHS NL

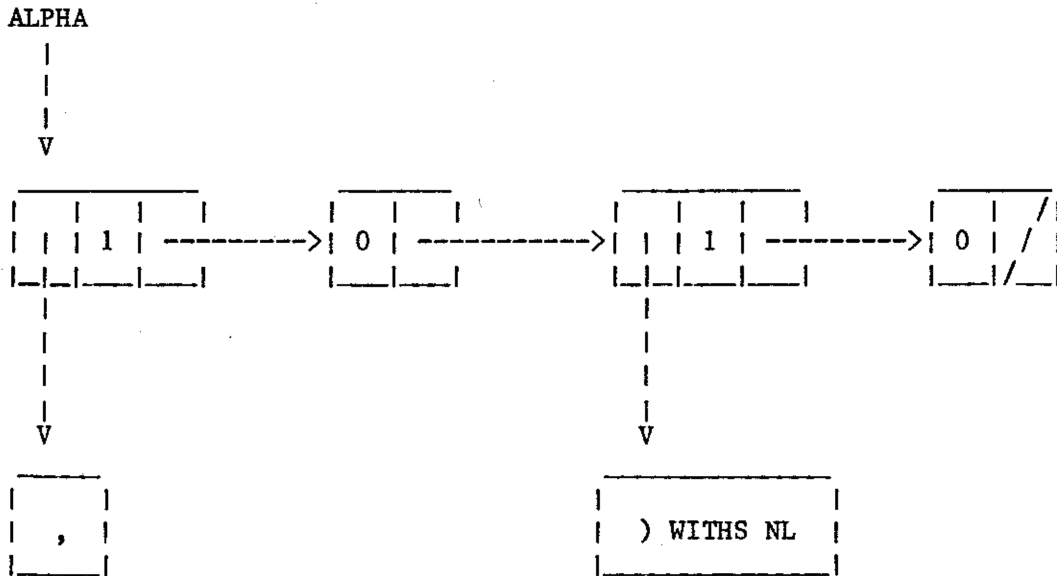


Figure 4. A Structure with Fixed Delimiters

EXAMPLE (2): PRINT OPT X WITH SPACES WITH NL OR Y ALL ;

```
                 _______
                |       |
NAME----->|  PRINT  |
                |_______|

ALPHA
  |
  |
  |
  v
 _______          _______          _____________          _________
|   |   |        |   |   |        |  |  |   |   |        |   |   /|
| 2 |   |        | 0 | --------->|  |  | 1 | --------->| 0 | / |
|___|_|_|        |___|___|        |__|__|___|___|        |___|/__|
      |            A   A            |
      |            |   |            |
      |            |   |            v
      |            |   |          _____
      |            |   |         |     |
      |            |   |         |  ;  |
      |            |   |         |_____|
      |            |   |
      |            |   |______________________________
      v            |                                  |
 ____________      |               ____________       |
|  |  |   |   |    |              |  |  |   |  /|     |
|  |  | 3 | ------|------------->|  |  | 3 | / |     |
|__|__|___|___|    |              |__|__|___|/__|     |
   |               |                 |                |
   |               |                 |                |
   v               |                 v                |
 _____________   __|__            _____________   ____|__
|  |  |   |   | |   |   |        |  |  |   |   | |   |   |
|  |  | 1 | ------>| 0 |   |        |  |  | 1 | ------>| 0 |   |
|__|__|___|___| |___|___|        |__|__|___|___| |___|___|
 |                                  |
 |                                  |
 v                                  v
 _________________________         _____
|                         |       |     |
|  X WITH SPACES WITH NL  |       |  Y  |
|_________________________|       |_____|
```

Figure 5. A Structure containing an Option List

EXAMPLE(3): MCSKIP OPT , N1 OR N1 NL ALL

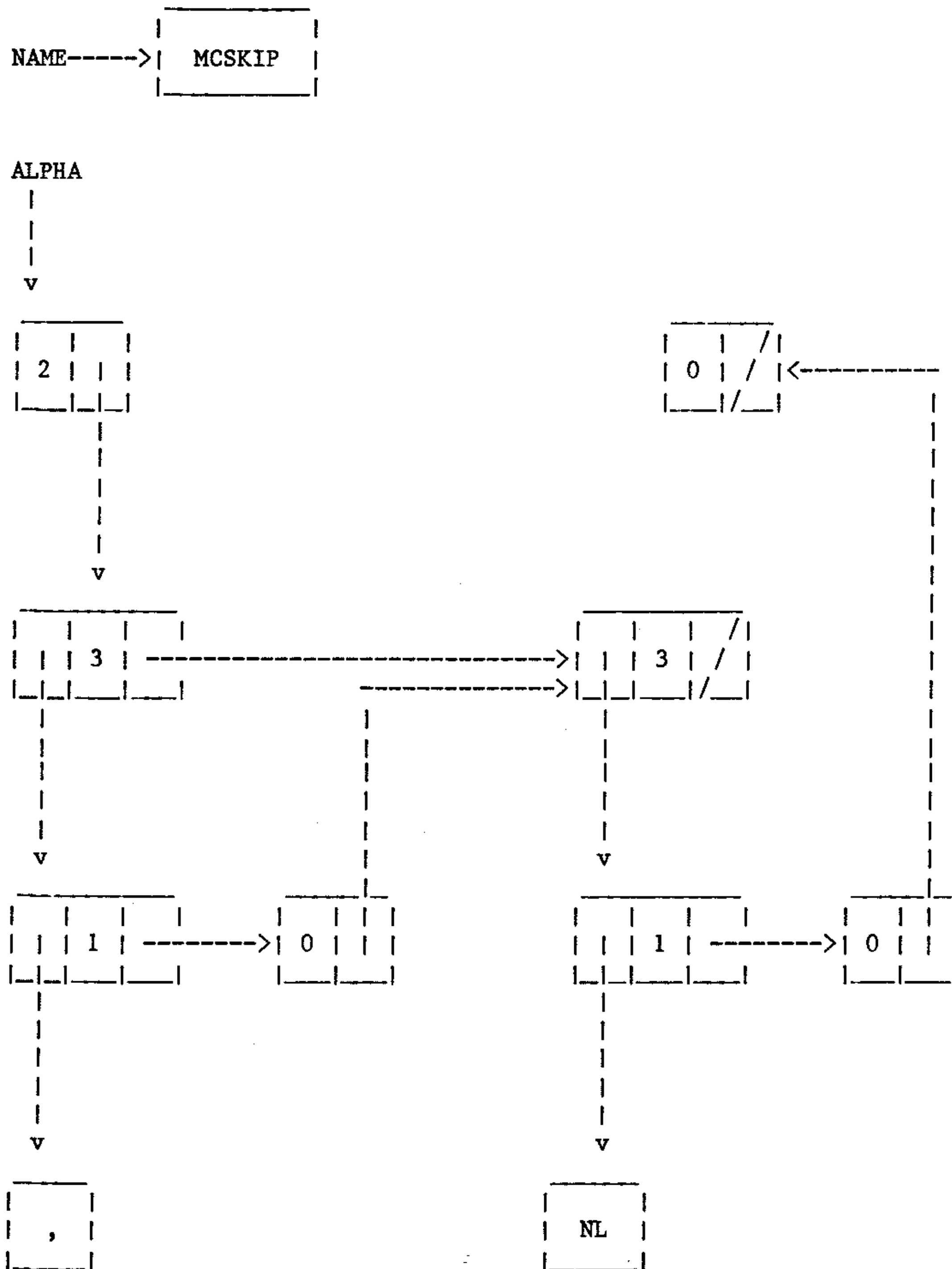


Figure 6. Structure with an Option List, Nodeplaces, and Nodegos

3.36 MACEXPR

Function: evaluates a macro expression.

Parameters:

1. MCEXPR - points to the text containing the macro expression being evaluated.

2. EXPRERR - indicates that an error occurred during the evaluation of the macro expression.

Called by: INSERT, MCDEF, MCGO, MCPVAR, MCSET, MCSUB

Calls: FETCH, TERM

3.37 TERM

Function: evaluates a term contained within a macro expression.

Parameters:

1. MCEXPR - points to the text containing the macro expression being evaluated.

2. EXPRERR - indicates that an error occurred during the evaluation of the macro expression.

3. EXPRCOL - indicates the position within the macro expression where the search for the next terminal will begin.

4. EXPREOF - indicates that the end of the macro expression has been encountered.

5. TERML - the next terminal of the macro expression.

Called by: MACEXPR

Calls: FETCH, PRIMARY

3.38 PRIMARY

Function: evaluates a primary contained within a macro expression.

Parameters:

1. MCEXPR - points to the text containing the macro expression being evaluated.
2. EXPRERR - indicates that an error occurred during the evaluation of the macro expression.
3. EXPRCOL - indicates the position within the macro expression where the search for the next terminal will begin.
4. EXPREOF - indicates that the end of the macro expression has been encountered.
5. TERML - the next terminal of the macro expression.

Called by: PRIMARY, TERM

Calls: FETCH, OPERAND, PRIMARY

3.39 OPERAND

Function: evaluates an operand contained within a macro expression.

Parameters:

1. MCEXPR - points to the text containing the macro expression being evaluated.
2. EXPRERR - indicates that an error occurred during the evaluation of the macro expression.
3. EXPRCOL - indicates the position within the macro expression where the search for the next terminal will begin.
4. EXPREOF - indicates that the end of the macro expression has been encountered.
5. TERML - the next terminal of the macro expression.

Called by: MACVAR, MCSET, PRIMARY

Calls: FETCH, MACVAR

3.40 MACVAR

Function: evaluates a macro variable contained within a macro expression.

Parameters:

1. MCEXPR - points to the text containing the macro expression being evaluated.

2. EXPRERR - indicates that an error occurred during the evaluation of the macro expression.

3. EXPRCOL - indicates the position within the macro expression where the search for the next terminal will begin.

4. EXPREOF - indicates that the end of the macro expression has been encountered.

5. TERML - the next terminal of the macro expression.

Called by: OPERAND

Calls: FETCH, OPERAND

3.41 FETCH

Function: assigns the next terminal of a macro expression to the parameter "TERML".

Parameters:

1. MCEXPR - points to the text containing the macro expression being evaluated.

2. EXPRCOL - indicates the position within the macro expression where the search for the next terminal will begin.

3. EXPREOF - indicates that the end of the macro expression has been encountered.

4. TERML - the next terminal of the macro expression.

Called by: MACEXPR, MACVAR, MCSET, PRIMARY, OPERAND, TERM

Calls: SCANNER

3.42 PRNTDEL

Function: transmits a delimiter name specification to the debugging file.

Parameters:

1. PTR - points to the delimiter name specification to be printed.

Called by: CONTEXT, ERR3, STATS

Calls: SCANNER

3.43 PRINTXT

Function: transmits a piece of text to the debugging file.

Parameters:

1. PTR - points to the head of the sequence of text cards containing the text to be printed.

Called by: INSERT, CONTEXT, ERR2

Calls: SCANNER

3.44 CONTEXT

Function: prints the context of an error message.

Parameters: (none)

Called by: ERR3, INSERT, MCALTER, MCDEF, MCGO, MCINS, MCNOTE, MCPVAR, MCSET, MCSKIP, MCSUB, SCANNER

Calls: PRINTXT, PRNTDEL

3.45 MCERRS

Function: prints the error messages pertaining to the evaluation of macro expressions.

Parameters:

1. ERRFLAG - the number of the error message.

2. VALUE - the value resulting from the evaluation of the macro

expression.

Called by: INSERT, MCDEF, MCGO, MCPVAR, MCSET, MCSUB

Calls: ERR1

3.46 ERR1

Function: prints an error message when an illegal macro variable has been used.

Parameters:

1. FLAG - the flag (either "T", "P", or "S") which identifies the type of the macro variable.

2. N - the value of the subscript associated with the flag.

Called by: MCERRS

Calls: (none)

3.47 ERR2

Function: prints an error message when an argument of an insert or operation macro has an illegal value.

Parameters:

1. N - the number of the illegal argument.

2. ARGHEAD - points to the head of the construction argument list in which the erroneous argument was detected.

Called by: INSERT, MCALTER, MCDEF, MCGO, MCINS, MCPVAR, MCSET, MCSKIP, MCSUB

Calls: PRINTXT

3.48 ERR3

Function: prints an error message when the end of the scanned text is reached during the search for the closing delimiter of a construction.

Parameters:

1. OPTLIST - flag indicating whether the name specifications of the missing delimiters were contained within an option list.

2. NODEPTR - pointer to either the delimiter name specification of a delimiter or the head node of the option list in which the specifications of the missing delimiters are contained.

3. CONSTRUC_TYPE - the type of the unmatched construction.

4. CONSTRUC_NAME - pointer to the delimiter name specification of the unmatched construction.

Called by: DELS

Calls: CONTEXT, PRNTDEL

3.49 ERROR

Function: prints the error messages pertaining to structure representations.

Parameters:

1. MESSAGES - the number of the error message.

Called by: MCDEF, MCINS, MCSKIP

Calls: (none)

3.50 STATS

Function: prints certain statistical information at the termination of the processing of the source text.

Parameters: (none)

Called by: MLI

Calls: PRNTDEL

4. EXTERNAL VARIABLES

This section describes the external variables used by the various routines. The relatively large number (72) of external variables is a by-product of the relatively low usage of many of these variables. The secondary delimiters of the operation macros, for instance, are initialized by MLI, altered by MCALTER, and used by ALTRDEL when scanning for the delimiters of the operation macros. Since the majority of the secondary delimiters are used by only these three routines, it is unnecessary to pass these variables to other routines used in the scanning process. The few variables which are used extensively (such as LRECL, RECSIZE, and OMEGA) are assigned values during the initialization stage of the processing and are not subject to change.

The external variables can be categorized into 7 logical groups: the secondary delimiters of the operation macros, the reserved keywords, the layout keywords, the character sets used in scanning text, the input/output related variables, the variables which constitute the environment used in scanning text, and the variables relating to the scanning process itself.

4.1 SECONDARY DELIMITERS

The following external variables represent the various secondary delimiters of the operation macros. Any of these variables may be changed through a call to the operation macro MCALTER.

CHARACTER(1) EXTERNAL

COMMA The secondary delimiter ",".

Used in: ALTRDEL, MCALTER, MLI

EQUAL The secondary delimiter "=".

Used in: ALTRDEL, MCALTER, MLI

PAREN The secondary delimiter ")".

Used in: ALTRDEL, MCALTER, MLI

CHARACTER(2) VARYING EXTERNAL

AS The secondary delimiter "AS".

Used in: ALTRDEL, MCALTER, MLI

BC The secondary delimiter "BC".

Used in: ALTRDEL, MCALTER, MCGO, MLI

EN The secondary delimiter "EN".

Used in: ALTRDEL, MCALTER, MCGO, MLI

GE The secondary delimiter "GE".

Used in: ALTRDEL, MCALTER, MCGO, MLI

GR The secondary delimiter "GR".

Used in: ALTRDEL, MCALTER, MCGO, MLI

IF The secondary delimiter "IF".

Used in: ALTRDEL, MCALTER, MCGO, MLI

LE The secondary delimiter "LE".

Used in: ALTRDEL, MCALTER, MCGO, MLI

LT The secondary delimiter "LT".

Used in: ALTRDEL, MCALTER, MCGO, MLI

NE The secondary delimiter "NE".

Used in: ALTRDEL, MCALTER, MCGO, MLI

TO The secondary delimiter "TO".

Used in: ALTRDEL, MCALTER, MLI

CHARACTER(4) VARYING EXTERNAL

SSAS The secondary delimiter "SSAS".

Used in: ALTRDEL, MCALTER, MCDEF, MLI

VARS The secondary delimiter "VARS".

Used in: ALTRDEL, MCALTER, MLI

CHARACTER(6) VARYING EXTERNAL

UNLESS The secondary delimiter "UNLESS".

Used in: ALTRDEL, MCALTER, MLI

4.2 RESERVED KEYWORDS

This section contains the external variables which represent the reserved keywords used within structure representations.

CHARACTER(1) EXTERNAL

N The nodeflag

Used in: MCALTER, MLI, STRUC

WITH# The special character used to represent the reserved word "WITH".

Used in: MATCHNM, MLI, PUNCINT, STRUC

WITHS# The special character used to represent the reserved word "WITHS".

Used in: MATCHNM, MLI, PUNCINT, STRUC

CHARACTER(2) VARYING EXTERNAL

OR The reserved word "OR".

Used in: MCALTER, MLI, STRUC

CHARACTER(3) VARYING EXTERNAL

ALL The reserved word "ALL".

Used in: MCALTER, MLI, STRUC

OPT The reserved word "OPT".

Used in: MCALTER, MLI, STRUC

CHARACTER(4) VARYING EXTERNAL

WITH The reserved word "WITH".

Used in: MCALTER, MLI, STRUC

CHARACTER(5) VARYING EXTERNAL

WITHS The reserved word "WITHS".

Used in: MCALTER, MLI, STRUC

4.3 LAYOUT KEYWORDS

The layout keywords, which represent such layout characters as spaces, newlines, and startlines, are listed in this section.

CHARACTER(1) EXTERNAL

NL The newline marker appended to the end of each input record.

Used in: FETCH, INSERT, MCGO, MCLENG, MCNOTE, MCSUB, MLI, PRINT, PRINTXT, PRNTDEL, PUNCINT, SCAN, SCANNER, SCANSRC, STRUC

SL The startline marker optionally appended to the beginning of each input record.

Used in: FETCH, INSERT, INSTEXT, MCGO, MCLENG, MCNOTE, MCSUB, MLI, PRINT, PRINTXT, PRNTDEL, PUNCINT, SKIP, SCAN, SCANNER, SCANSRC, STRUC

SPACE# The special character representing the layout keyword "SPACE".

Used in: MATCHNM, MLI, PUNCINT, PRNTDEL, STRUC

SPACES# The special character representing the layout keyword "SPACES".

Used in: MATCHNM, MLI, PUNCINT, PRNTDEL, STRUC

CHARACTER(2) VARYING EXTERNAL

NL$ The layout keyword for the newline marker.

Used in: MCALTER, MLI, STRUC

SL$ The layout keyword for the startline marker.

Used in: MCALTER, MLI, STRUC

CHARACTER(5) VARYING EXTERNAL

SPACE The layout keyword "SPACE".

Used in: MCALTER, MLI, STRUC

SPACES The layout keyword "SPACES".

Used in: MATCHNM, MCALTER, MLI, STRUC

4.4 CHARACTER SETS

The following variables constitute the character sets used in determining the lexical class of a character. There are only two lexical classes: alphanumeric characters and punctuation characters.

CHARACTER(62) EXTERNAL

ALPHANM The alphanumeric characters.

Used in: MCALTER, MLI, SCANNER

PUNCHAR The punctuation characters (i.e. all non-alphanumeric characters, including the multi-punch characters).

Used in: MCALTER, MLI, SCANNER

4.5 INPUT/OUTPUT VARIABLES

The following external variables are used in transmitting information.

FIXED BINARY EXTERNAL

LRECL The maximum length of an input card (and thus of an atom).

Used in: DELS, EVALARG, GLOBAL, INSERT, INSTEXT, LOCDEF, MACEXPR, MACRO, MATCH, MATCHNM, MCALTER, MCGO, MCNOTE, MCSET, MCSUB, MLI, PRINTXT, PRNTDEL, OUTPUT, SCAN, SCANNER, SCANSRC, SKIP, STRUC

OUTCOL Indicates the column within the current record of the output file in which the writing of any subsequent text will begin.

Used in: SCANSRC, PRINT

RECSIZE The maximum length of a text card.

Used in: DELETE, DELS, EVALARG, FETCH, FORMTXT, INIT, INSERT, INSTEXT, GLOBAL, LOCDEF, MACRO, MATCH, MATCHNM, MCALTER, MCGO, MCLENG, MCNOTE, MCSKIP, MCSUB, MLI, OUTPUT, PRNTDEL, PRINTXT, SCANSRC, SCANNER, SKIP, STRUC

FIXED BINARY CONTROLLED EXTERNAL

LINE# The line number indicating which input card or text card currently is being scanned.

Used in: CONTEXT, ERR3, EVALARG, INSERT, INSTEXT, MACRO, MATCHNM, MCGO, MCNOTE, MCSUB, PRINTXT, OUTPUT, SCAN, SCANNER, SCANSRC, SKIP, STATS

OLDCOL The column in which the scan for the subsequent atoms of a piece of text will commence in following the "rescan" of a previous portion of text.

Used in: MATCHNM, SCANNER

POINTER CONTROLLED EXTERNAL

OLDSRC Indicates the sequence of text which is to be scanned following the "rescan" of a previous portion of text.

Used in: MATCHNM, SCANNER

CHARACTER(LRECL) CONTROLLED EXTERNAL

INCARD A card from the input file.

Used in: SCANNER, SCANSRC

CHARACTER(SYSVAR(11) - 1) CONTROLLED EXTERNAL

HEADER The first (A-1) columns of the input card (where "A" is the value of system variable 11).

Used in: PRINT, SCANNER, SCANSRC

CHARACTER(RECSIZE) CONTROLLED EXTERNAL

SRCARD The text card containing the subsequent atoms of the text which is to be scanned following the "rescan" of a previous portion of text.

Used in: SCANNER, MATCHNM

CHARACTER(LRECL - SYSVAR(12)) CONTROLLED EXTERNAL

TRAILER The portion of the input card beyond column Z (where "Z" is the value of system variable 12).

Used in: PRINT, SCANNER, SCANSRC

4.6 ENVIRONMENTAL VARIABLES

The variables which constitute the macro-time environment are listed below.

FIXED BINARY EXTERNAL

MACALLS The number of macro calls performed.

Used in: MACRO, MLI, SCAN, STATS

PERMVR# The number of permanent variables in the environment.

Used in: MACVAR, MCPVAR, MCSET, MLI

SYSVAR# The number of system variables in the environment.

Used in: MACVAR, MCSET, MLI, SCANNER

FIXED BINARY CONTROLLED EXTERNAL

CURARG# The number of arguments in the environment.

Used in: INSTEXT, MACRO, MLI

TEMPVR# The number of temporary variables in the environment.

Used in: MACVAR, MACRO, MCSET, MLI

FIXED BINARY(31,0) CONTROLLED EXTERNAL

SYSVAR(1:SYSVAR#) The system variables.

Used in: MACVAR, MCNOTE, MCSET, MLI, PRINT,
SCANNER, SCANSRC

TEMPVAR(1:TEMPVR#) The temporary variables.

Used in: MACVAR, MACRO, MCSET

POINTER EXTERNAL

PRMHEAD Points to the head of the list of permanent
variables.

Used in: MACVAR, MCSET, MLI

PRMTAIL Points to the tail of the list of permanent
variables.

Used in: MCPVAR, MLI

POINTER CONTROLLED EXTERNAL

CURARGS Points to the head of the argument list in the environment.

Used in: INSTEXT, MACRO

CURDELS Points to the head of the delimiter list in the environment.

Used in: INSTEXT, MACRO

CURLABS Points to the head of the macro-time label list in the environment.

Used in: EVALARG, INSERT, INSTEXT, MACRO, MCGO, MLI

4.7 SCANNING VARIABLES

The following external variables are used during the scanning and evaluation of text.

FIXED BINARY EXTERNAL

HT_SIZE The size of the hash tables.

Used in: GLOBAL, LOCDEF, MCNO, MLI, SCAN, STATS

LAB# The number of a label being searched for.

Used in: INSERT, MCGO

GOSTART Indicates the line number on which the forward scan for a macro-time label began.

Used in: MCGO, SCAN

MCLEVEL The current nesting level of macro calls.

Used in: MACRO, MLI

NESTLVL The current nesting level of text evaluation.

Used in: EVALARG, INSERT, INSTEXT, MACRO, MCDEF, MCGO, MCINS, MCNO, MCSKIP, MLI

BIT(1) EXTERNAL

EOF The end of file flag.

Used in: DELS, MCGO, SCANNER, SCANSRC

LABSRCH Flag indicating whether a search for a macro-time label is in progress.

Used in: DELS, INSERT, MCGO, MLI, SCAN

BIT(1) CONTROLLED EXTERNAL

EVAL Indicates whether a construction is to be evaluated.

Used in: INSERT, INSTEXT, MACRO, SCAN, SCANSRC

INSKIP Indicates that the construction whose arguments and delimiters are to be formed is nested within a skip.

Used in: MLI, SCAN, DELS

CHARACTER(1) EXTERNAL

RESCAN The special character used to mark the end of a piece of source text to be rescanned.

Used in: MATCHNM, MLI, PUNCINT, SCANNER

CHARACTER(1) VARYING EXTERNAL

OMEGA The end of text marker. This variable is of varying length so that it will be compatible with a formal

argument of FORMTXT.

Used in: DELETE, DELS, EVALARG, FETCH, INIT, INSERT, INSTEXT, MACRO, MATCH, MATCHNM, MCALTER, MCGO, MCLENG, MCNOTE, MCSUB, MLI, PRINTXT, PRNTDEL, PUNCINT, OUTPUT, SKIP, STRUC

POINTER EXTERNAL

CONTXT# Pointer to the top member of the stack of structures containing context information.

Used in: CONTEXT, MCNOTE, MLI, SCAN

EXTERNAL STRUCTURES

```
DECLARE 1 LCTABLE(0:(HT_SIZE-1))  /* THE HASH TABLE CONTAINING  */
                CONTROLLED        /* THE LOCAL CONSTRUCTION     */
                EXTERNAL,         /* DEFINITIONS                */

          2 NT_HEAD               /* POINTS TO THE HEAD OF THE  */
               POINTER,           /* NAME TABLE ASSOCIATED WITH */
                                  /* THIS HASH TABLE ENTRY      */

          2 THRD_HEAD             /* POINTS TO THE HEAD OF THE  */
               POINTER;           /* LINKED LIST OF CONSTRUCTION*/
                                  /* DEFINITIONS THAT WERE      */
                                  /* ENTERED INTO THE NAME TABLE*/
                                  /* ASSOCIATED WITH THIS HASH  */
                                  /* TABLE ENTRY                */


DECLARE 1 GLTABLE(0:(HT_SIZE-1))  /* THE HASH TABLE CONTAINING  */
                CONTROLLED        /* THE GLOBAL CONSTRUCTION    */
                EXTERNAL,         /* DEFINITIONS                */

          2 NT_HEAD               /* POINTS TO THE HEAD OF THE  */
               POINTER;           /* NAME TABLE ASSOCIATED WITH */
                                  /* THIS HASH TABLE ENTRY      */
```

5. SYSTEM FILES

There are 5 files used by the macro processor. This section describes the purpose for which the files are used and the routines which use each file.

5.1 INPUT

Purpose: contains the source text to be scanned.

Used by: SCANSRC, SCANNER

5.2 SYSPRINT

Purpose: to produce a hardcopy listing of the text resulting from the macro expansion of the source text. This file normally defaults to the system print file.

Used by: SCANSRC, PRINT

5.3 RESULTS

Purpose: to retain a machine readable version of the text resulting from the macro expansion of the source text. This file is normally a user supplied data set.

Used by: SCANSRC, PRINT

5.4 SOURCE

Purpose: contains the listing of the original, unexpanded source text.

Used by: SCANSRC, SCANNER

5.5 DEBUG

Purpose: contains the system error messages.

Used by: SCANSRC, SCANNER, INSERT, MCINS, MCSKIP, MCDEF, MCALTER, MCSUB, MCSET, MCNOTE, MCGO, MCPVAR, PRNTDEL, PRINTXT, CONTEXT, MCERRS, ERR1, ERR2, ERR3, ERROR, STATS