MCL
&
MCLCOM
Developers Manual
Version 3.0

D.G. Papageorgiou I.E. Lagaris

August 6, 2003

Contents

List of Tables

Chapter 1

Introduction

1.1 Historical Note

Optimization problems were at first tackled by using autonomous routines that were available in the various subroutine libraries. Later on, integrated optimization packages began to emerge, for example the MINUIT [1] and Merlin [3, 4, 5] programs. Merlin is an interactive program that implements a number of different minimization algorithms, and interacts with the user, through a friendly command line interface. It has been under continuous development since the mid 80's. The development of the Merlin Control Language [2] came up naturally, as a tool for implementing complex minimization strategies and automating minimization sessions.

Although Merlin and Mcl are two separate programs, they are in close interaction. Mcl has to recognize and support all Merlin commands and features, while Merlin must provide the means for execution of Mcl programs.

1.2 Manual Organization

This manual in neither an Mcl tutorial, nor a language reference. It describes the internals of the Mcl compiler (version 3.0), and is addressed to those wishing to enhance or modify the compiler program. The reader is assumed to be familiar with both, Merlin and Mcl. In the present manual, we compiled a lot of useful information on Mcl and its compiler program mclcom-3.0. The material is organized as follows: In chapter 2, we present in a rigorous way the syntax definition of Mcl. In chapter 3, the overall structure of the compiler is analyzed. In chapter 4, a detailed description (routine-by-routine) of the compiler is given. These details are mainly of technical importance and they may be skipped during the first reading. In chapter 5 and 6, we describe the mclcom input (filenames and options) and output (moc and Error-reports). In chapter 7 we comment on the mclcom limitations. Finally, in chapter 8, we give some useful hints to the user, aiming to the efficient utilization of the language.

1.3 Typo Conventions

1.4 Acknowledgements

The authors would like to thank Prof. G.A. Evangelakis and Prof. I.N. Demetropoulos for illuminating discussions.

Ioannina, August 6, 2003

Chapter 2

Technical presentation of MCL

The syntactic definition of MCL, stated in EBNF [6] together with the restrictions of the language follows.

2.1 A note on syntax semantics

In order to make this manual self-contained, we give here a brief summary of EBNF, along with a few additional conventions which we use.

- Syntactic units are enclosed in brackets $\langle\ \rangle$.
- The symbols **::=** and | are interpreted as "is defined as" and "or" respectively.
- The implied concatenation operator has priority over the alternation operator. For example, in the hypothetical rule: ⟨a⟩ **::=** ⟨b⟩ ⟨c⟩ | ⟨d⟩, the syntactic entity ⟨a⟩ may be either ⟨b⟩ ⟨c⟩ or ⟨d⟩. In order to alter the order of evaluation, we use parentheses, so that: ⟨a⟩ **::=** ⟨b⟩(⟨c⟩ | ⟨d⟩), would define ⟨a⟩ to be either ⟨b⟩ ⟨c⟩ or ⟨b⟩ ⟨d⟩.
- Spaces, tabs and newline characters, are irrelevant in between syntactic units.
- What is meant by $\{\ \langle x\rangle\ \}_n^m$ is the existence of at least n and at most m occurrences of ⟨x⟩. In the absence of the minimum replication factor n, zero is assumed. In the absence of the maximum replication factor m, an arbitrary number of repetitions is assumed.
- By [⟨x⟩] we denote, at most one occurance of ⟨x⟩ (it is equivalent to £ | ⟨x⟩, where £ denotes the null string).
- Keywords like `VAR`, `SIMPLEX` etc., and key symbols (such as semicolons or parentheses) are enclosed in single quotes and appear as `'VAR'`, `'SIMPLEX'`, `';'`, etc.

- EOS stands for “End Of Statement” and denotes the end of a statement. An MCL statement may be split across several physical lines, using the continuation symbol & at the end of each line. A statement ends, when the last non-blank, non-tab character of a line, is not the continuation symbol &. It also ends when the comment character % is encountered outside a pair of single quotes.

- Finally, note that by ⟨Letter⟩ we refer to any one of ‘A’, ‘B’, . . . ‘Z’, ‘a’, ‘b’, . . . ‘z’, by ⟨Digit⟩, to any one of ‘0’, ‘1’, ‘2’,. . . ‘9’, and by ⟨Special_Character⟩, to all special characters that can be displayed.

2.2 Context-free part of the MCL syntax

1. ⟨MCL_Source⟩ ::= { ⟨Program_Unit⟩ }
2. ⟨Program_Unit⟩ ::= ⟨Program⟩ | ⟨Subprogram⟩
3. ⟨Program⟩ ::= ‘PROGRAM’ { ⟨Var_Declaration⟩ } { ⟨Function_Declaration⟩ }
   ⟨Block_Of_Lines⟩ ‘END’
4. ⟨Subprogram⟩ ::= ⟨Subprogram_Header⟩ { ⟨Var_Declaration⟩ } { ⟨Function_Declaration⟩ }
   ⟨Block_Of_Lines⟩ ‘END’
5. ⟨Subprogram_Header⟩ ::= ‘SUB’ ⟨Identifier⟩ [‘(’ ⟨Var_List⟩ ‘)’]
6. ⟨Var_Declaration⟩ ::= ‘VAR’ ⟨Var_List⟩ EOS
7. ⟨Var_List⟩ ::= ⟨Identifier⟩ | ⟨Simple_Array⟩ { ‘;’ ⟨Identifier⟩ | ⟨Simple_Array⟩ }
8. ⟨Function_Declaration⟩ ::= ‘FUNCTION’ ⟨Identifier⟩ ⟨Argument_List⟩ ‘=’
   ⟨MCL_Expression⟩ EOS
9. ⟨Block_Of_Lines⟩ ::= { (£ | ⟨Statement⟩ | ⟨Label⟩) EOS }
10. ⟨Argument_List⟩ ::= ‘[’ ⟨Identifier⟩ { ‘,’ ⟨Identifier⟩ } ‘]’
11. ⟨Simple_Array⟩ ::= ⟨Identifier⟩ ‘[’ ⟨Lower_Bound⟩ ‘:’ ⟨Upper_Bound⟩ { ‘,’
    ⟨Lower_Bound⟩ ‘:’ ⟨Upper_Bound⟩ } ‘]’
12. ⟨Lower_Bound⟩ ::= ⟨Sign⟩ ⟨Integer⟩
13. ⟨Upper_Bound⟩ ::= ⟨Sign⟩ ⟨Integer⟩ | ‘*’
14. ⟨MCL_Expression⟩ ::= (⟨Sign⟩ | ‘NOT’) ⟨Expression⟩ | ⟨Expression⟩
15. ⟨Expression⟩ ::= ⟨Xterm⟩ [‘XOR’ ⟨Expression⟩]
16. ⟨Xterm⟩ ::= ⟨Rterm⟩ [‘OR’ ⟨Xterm⟩]
17. ⟨Rterm⟩ ::= ⟨Dterm⟩ [‘AND’ ⟨Rterm⟩]

18. ⟨Dterm⟩ ::= ⟨Lterm⟩ [(‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘#’) ⟨Dterm⟩]

19. ⟨Lterm⟩ ::= ⟨Term⟩ [(‘+’ | ‘-’) ⟨Lterm⟩]

20. ⟨Term⟩ ::= ⟨Factor⟩ [(‘*’ | ‘/’) ⟨Term⟩]

21. ⟨Factor⟩ ::= ⟨Base⟩ [‘**’ ⟨Factor⟩]

22. ⟨Base⟩ ::= ⟨Real⟩ | ⟨Variable⟩ | ‘(’ ⟨MCL_Expression⟩ ‘)’

23. ⟨Variable⟩ ::= ⟨Intrinsic_Variable⟩ | ⟨Identifier⟩ | ⟨Function_Reference⟩ | ⟨Simple_Array_Element⟩ | ⟨Intrinsic_Array_Element⟩

24. ⟨Identifier⟩ ⟨Letter⟩ {’_’ | ⟨Letter⟩ | ⟨Digit⟩ }

25. ⟨Real⟩ ::= { ⟨Digit⟩ }$_1$ [‘.’ { ⟨Digit⟩ }$_1$] [‘E’ ⟨Sign⟩ { ⟨Digit⟩ }$_1$]

26. ⟨Sign⟩ ::= ’+’ | ’-’ | £

27. ⟨Statement⟩ ::= ⟨Simple_Statement⟩ | ⟨Block_If⟩ | ⟨When⟩ | ⟨Loop_Statement⟩

28. ⟨Simple_Statement⟩ ::= ⟨Assignment⟩ | ⟨Get⟩ | ⟨Display⟩ | ⟨Move⟩ | ⟨Call⟩ | ⟨Merlin_Control_Statement⟩

29. ⟨Loop_Statement⟩ ::= ‘LOOP’ ⟨Identifier⟩ ‘FROM’ ⟨MCL_Expression⟩ ‘TO’ ⟨MCL_Expression⟩ [‘BY’ ⟨MCL_Expression⟩] EOS ⟨Block_Of_Lines⟩ ‘END’ ‘LOOP’

30. ⟨When⟩ ::= ‘WHEN’ ⟨MCL_Expression⟩ ‘JUST’ ⟨Simple_Statement⟩

31. ⟨Block_If⟩ ::= ‘IF’ ⟨MCL_Expression⟩ ‘THEN’ EOS ⟨Block_Of_Lines⟩ [‘ELSE’ EOS ⟨Block_Of_Lines⟩] ‘END’ ‘IF’

32. ⟨Assignment⟩ ::= ⟨Identifier⟩ | ⟨Simple_Array_Element⟩ ‘=’ ⟨MCL_Expression⟩

33. ⟨Get⟩ ::= ‘GET’ ⟨Identifier⟩ { ‘;’ ⟨Identifier⟩ } [‘FROM’ ⟨Filename⟩]

34. ⟨Display⟩ ‘DISPLAY’ ⟨MCL_Expression⟩ | ⟨String⟩ { ‘;’ ⟨MCL_Expression⟩ | ⟨String⟩ } [‘TO’ ⟨Identifier⟩]

35. ⟨Move⟩ ::= ‘MOVE’ ‘TO’ ⟨Identifier⟩

36. ⟨Label⟩ ::= ⟨Identifier⟩ ‘:’

37. ⟨Call⟩ ::= ‘CALL’ ⟨Identifier⟩ [’(’ ⟨MCL_Expression⟩ { ’;’ ⟨MCL_Expression⟩ } ’)’]

38. ⟨Simple_Array_Element⟩ ::= ⟨Identifier⟩ ‘[’ ⟨MCL_Expression⟩ { ‘,’ ⟨MCL_Expression⟩ } ‘]’

39. ⟨Function_Reference⟩ ::= (⟨Identifier⟩ | ⟨Intrinsic_Function⟩) ‘[’ ⟨MCL_Expression⟩ { ‘,’ ⟨MCL_Expression⟩ } ‘]’

40. ⟨Intrinsic_Variable⟩ ::= ‘VALUE’ | ‘PCOUNT’ | ‘TCOUNT’ | ‘DIM’ | ‘PRECISION’ | ‘BACKUP’ | ‘DERIVA’ | ‘PRINTO’ | ‘PROCES’ | ‘FLAG’ | ‘HTCOUNT’ | ‘HPCOUNT’ | ‘GTCOUNT’ | ‘GPCOUNT’

41. ⟨Intrinsic_Array_Element⟩ ::= (‘X’ | ‘R’ | ‘L’ | ‘FIX’ | ‘GRAD’ | ‘STEP’ | ‘MARG’) ‘[’ ⟨MCL_Expression⟩ ‘]’

42. ⟨Intrinsic_Function⟩ ::= ‘ABS’ | ‘LOG’ | ‘LOG10’ | ‘SIN’ | ‘COS’ | ‘TAN’ | ‘ASIN’ | ‘ACOS’ | ‘ATAN’ | ‘SINH’ | ‘COSH’ | ‘TANH’ | ‘ASINH’ | ‘ACOSH’ | ‘ATANH’ | ‘MIN’ | ‘MAX’ | ‘MEAN’ | ‘MOD’ | ‘TRUNC’ | ‘ROUND’ | ‘FACT’ | ‘EXP’ | ‘GRADNORM’ | ‘RAN’ | ‘SQRT’ | ‘GRMS’

43. ⟨Merlin_Control_Statement⟩ ::= ⟨Non_Parametric⟩ | ⟨I_Parametric⟩ | ⟨IV_Parametric⟩ | ⟨IS_Parametric⟩ | ⟨Panel_Parametric⟩ | ⟨General_Parametric⟩

44. ⟨Non_Parametric⟩ ::= ‘ADJUST’ | ‘FULLBACK’ | ‘LASTBACK’ | ‘NOBACK’ | ‘ANAL’ | ‘FULLPRINT’ | ‘HALFPRINT’ | ‘EVALUATE’ | ‘MODEDIS’ | ‘FAST’ | ‘VALDIS’ | ‘FIXALL’ | ‘LOOSALL’ | ‘GRADDIS’ | ‘STEPALL’ | ‘NOPRINT’ | ‘FLAGDIS’ | ‘RESET’ | ‘NUMER’ | ‘REVEAL’ | ‘SHORTDIS’ | ‘PAUSE’ | ‘FINISH’ | ‘EXIT’ | ‘CFLAGDIS’ | ‘LIMITS’ | ‘NOEVAL’ | ‘NOTARGET’ | ‘STEPDIS’ | ‘GENERAL’ | ‘SOS’ | ‘ALIASDIS’ | ‘SUBRETURN’ | ‘GNORM’ | ‘QUAD’

45. ⟨I_Parametric⟩ ::= ⟨I_Parametric_Name⟩ ‘(’ ⟨Key⟩ ‘.’ ⟨MCL_Expression⟩ { ‘;’ ⟨Key⟩ ‘.’ ⟨MCL_Expression⟩ } ‘)’

46. ⟨I_Parametric_Name⟩ ::= ‘FIX’ | ‘LOOSE’ | ‘LEFTDEMARGIN’ | ‘LDEMARGIN’ | RIGHTDEMARGIN’ | ’RDEMARGIN’ | ‘NONAME’

47. ⟨IV_Parametric⟩ ::= ⟨IV_Parametric_Name⟩ ‘(’ ⟨Key⟩ ‘.’ ⟨MCL_Expression⟩ ‘=’ ⟨MCL_Expression⟩ { ‘;’ ⟨Key⟩ ‘.’ ⟨MCL_Expression⟩ ‘=’ ⟨Value⟩ } ‘)’

48. ⟨IV_Parametric_Name⟩ ::= ‘POINT’ | ‘FLAG’ | ‘LEFTMARGIN’ | ‘LMARGIN’ | ‘RIGHTMARGIN’ | ‘RMARGIN’ | ‘STEP’

49. ⟨IS_Parametric⟩ ::= ⟨IS_Parametric_Name⟩ ‘(’ ⟨Key⟩ ‘.’ ⟨MCL_Expression⟩ ‘=’ ⟨String⟩ { ‘;’ ⟨Key⟩ ‘.’ ⟨MCL_Expression⟩ ‘=’ ⟨String⟩ } ‘)’

50. ⟨IS_Parametric_Name⟩ ::= ‘GODFATHER’ | ‘CFLAG’ | ‘MIXED’

51. ⟨Panel_Parametric⟩ ::= ⟨Panel_Parametric_Name⟩ [‘(’ ⟨Key⟩ ‘=’ ⟨Value⟩ { ‘;’ ⟨Key⟩ ‘=’ ⟨Value⟩ } ‘)’]

52. ⟨Panel_Parametric_Name⟩ ::= ‘AUTO’ | ‘CONGRA’ | ‘GRAPH’ | ‘ACCUM’ | ’MAD’ | ‘TOLMIN’ | ‘LEVE’ | ‘ROLL’ | ‘SIMPLEX’ | ‘DFP’ | ‘BFGS’ | ‘TRUST’ | ‘PSGRAPH’ | ‘CONTROL’ | ‘MEMO’ | ‘INSPECT’ | ‘PICK’ | ‘BACKUP’ | ‘INIT’ | ‘DUMP’ | ‘HISTORY’ | ‘HESSIAN’

53. ⟨General_Parametric⟩ ::= ⟨General_Parametric_Name⟩ ['(' ⟨Key⟩ '=' ⟨Value⟩ { ';' ⟨Key⟩ '=' ⟨Value⟩ } ')']

54. ⟨General_Parametric_Name⟩ ::= 'DELETE' | 'DISCARD' | 'REWIND' | 'GOEOF' | 'HIDEOUT' | 'STOP' | 'RETURN' | 'ALIAS' | 'UNALIAS' | 'EPILOG' | 'TITLE' | 'PDUMP' | 'PANELON' | 'PANELOFF' | 'PSTATUS' | 'GRADCHECK' | 'GRADCHECK' | 'TARGET' | 'QUIT'

55. ⟨String⟩ ::= ' ' ' { ⟨Letter⟩ | ⟨Digit⟩ | ⟨Special_Character⟩ }$_1$ ' ' '

56. ⟨Key⟩ ::= ⟨Identifier⟩

57. ⟨Value⟩ ::= ⟨MCL_Expression⟩ | ⟨String⟩

58. ⟨Filename⟩ ::= ⟨String⟩

2.3 Context-sensitive part of the MCL syntax

The following rules complement the syntax definition of MCL.

1. Keys and corresponding values for ⟨General_Parametric⟩ statements are shown in table 2.1.
2. Keys and corresponding values for ⟨IV_Parametric⟩ statements are shown in table 2.2.
3. Keys and corresponding values for ⟨IS_Parametric⟩ statements are shown in table 2.3.
4. Keys and corresponding values for ⟨I_Parametric⟩ statements are shown in table 2.4.
5. An MCL line is up to 120 characters long counting underscores. An MCL identifier can be at most 30 characters long not counting underscores.
6. Up to 10 continuation lines are allowed. Each continuation line ends with the symbol &.
7. A function declaration, cannot contain a forward reference to another function. Function dummy arguments cannot be array elements or intrinsic items .
8. The identifiers `JUST, TO, BY, THEN,` and `FROM` are reserved words and cannot be used as symbolic names.
9. All operations are performed from left to right.
10. A proper reference to a function or array element, should contain all of the declared arguments.
11. A ⟨Block_Of_Lines⟩ must not contain a label which is referenced from outside the block.
12. All ⟨Block_If⟩ and ⟨Loop_Statement⟩ that belong to a ⟨Block_Of_Lines⟩ must terminate inside the same ⟨Block_Of_Lines⟩.

13. A ⟨Key⟩ may not be specified more than once in a statement.
14. The priority of evaluation of the MCL operators, is implied into definitions 14 through 22. ⟨Base⟩ is evaluated first, then ⟨Factor⟩ etc.
15. A `SUBRETURN` statement is not allowed in the main program.
16. An asterisk, denoting an adjustable ⟨Upper_Bound⟩ is allowed only in the dummy argument list of a subprogram.
17. A single quote inside a string, must be entered as the escape sequence `\'`
18. ⟨Panel_Parametric⟩ statements and their corresponding keys and values, are read in from the Merlin panel description file.
19. Exactly one main program must be present in an MCL source file.

⟨General_Parametric_Name⟩	⟨Key⟩	⟨Value⟩	Must be present
DELETE	FILE	⟨String⟩	Yes
REWIND	FILE	⟨String⟩	Yes
GOEOF	FILE	⟨String⟩	Yes
CLOSE	FILE	⟨String⟩	Yes
DISCARD	FILE	⟨String⟩	Yes
	TYPE	'TEXT' | 'BIN'	No
HIDEOUT	FILE	⟨String⟩	Yes
	APPEND	'YES' | 'NO'	No
STOP	EPILOG	'YES' | 'NO'	No
RETURN	EPILOG	'YES' | 'NO'	No
ALIAS	NAME	⟨String⟩	Yes
	COMMAND	⟨String⟩	Yes
UNALIAS	NAME	⟨String⟩	Yes
EPILOG	COMMAND	⟨String⟩	Yes
TITLE	TITLE	⟨String⟩	Yes
PDUMP	FILE	⟨String⟩	Yes
	COMMAND	⟨String⟩	No
PANELON	COMMAND	⟨String⟩	No
PANELOFF	COMMAND	⟨String⟩	No
PSTATUS	COMMAND	⟨String⟩	No
GRADCHECK	MODE	⟨String⟩	Yes
	MODE2	⟨String⟩	No
TARGET	VALUE	⟨MCL_Expression⟩	Yes
QUIT	FLAG	⟨MCL_Expression⟩	Yes

Table 2.1: Keywords and allowed values for ⟨General_Parametric⟩ statements.

⟨IV_Parametric_Name⟩	⟨Key⟩	⟨Value⟩
POINT	X	⟨MCL_Expression⟩
STEP	S	⟨MCL_Expression⟩
FLAG	F	⟨MCL_Expression⟩
LEFTMARGIN	L	⟨MCL_Expression⟩
LMARGIN	L	⟨MCL_Expression⟩
RIGHTMARGIN	R	⟨MCL_Expression⟩
RMARGIN	R	⟨MCL_Expression⟩

Table 2.2: Keys and corresponding values for ⟨IV_Parametric⟩ statements.

⟨IS_Parametric_Name⟩	⟨Key⟩	⟨Value⟩
`GODFATHER`	`X`	⟨String⟩
`CFLAG`	`X`	⟨String⟩
`MIXED`	`X`	⟨String⟩

Table 2.3: Keys and corresponding values for ⟨IS_Parametric⟩ statements.

⟨I_Parametric_Name⟩	⟨Key⟩
`LEFTDEMARGIN`	`L`
`LDEMARGIN`	`L`
`RIGHTDEMARGIN`	`R`
`RDEMARGIN`	`R`
`FIX`	`X`
`LOOSE`	`X`
`NONAME`	`X`

Table 2.4: Keys for ⟨I_Parametric⟩ statements.

Chapter 3

The overall structure of MCLCOM

3.1 General layout

MCLCOM is the name of the MCL compiler program, and its current version number is 3.0. Each time MCLCOM is activated, it proceeds with *lexical analysis*, *parsing*, *code generation*, and *error recovery* (if needed).

- *Lexical analysis* is the first task. It is the translation of the input stream of characters, into an appropriate form, easy to manipulate. This form, consists of a sequence of lexical items (or "tokens") which are meaningful to the compiler. Each token is represented by a unique numerical code. For every compiler, there exists a different set of meaningful lexical items. MCLCOM distinguishes 34 different lexical items, like ⟨Number⟩ for real numbers, ⟨Identifier⟩ for symbolic names, etc.

- *Parsing* is the transposition from the potentially ambiguous input phrases, to the internal unambiguous structure. The technique used in MCLCOM can be described as "wait-and-see" parsing. The first lexical item of an input phrase, determines the candidate rule to be matched. The rest of the lexical items are processed next, one by one, until a correct grammatical rule is revealed. MCL programs are more than just a sequence of lexical items; they do have structure, as the rules of the MCL grammar imply. A ⟨When⟩ statement for example, consists of the word `WHEN`, a Boolean test expression, the word `JUST` and finally, a ⟨Simple_Statement⟩.

- *Code generation* is the creation of the object code. Generally this phase is straightforward, since MCLCOM attempts no code optimization.

- *Error recovery* is the successful reaction of a compiler to an unexpected, weird situation in the source program. It is vital to obtain a correct diagnosis of the problem, to provide meaningful messages, and not to affect subsequent processing. For example, if an undeclared variable is detected, it should be announced. However reannouncing it each time it appears in the MCL source code, adds nothing to the diagnosis. MCLCOM copes with this problem in the following manner: When the undeclared variable is located, it is inserted into the table where all declared variables are kept. After this, the undeclared variable becomes artificially declared. MCLCOM produces two kinds of errors:

– *Fatal errors*, that always inhibit object code generation, and
– *Warnings*, that alert the user for an unusual situation. Code generation proceeds however.

The main input to MCLCOM is the MCL source program. Compilation proceeds in three phases, traditionally called passes.

Pass 1 The MCL source program is read in line by line. Subroutine `LEX` is called to perform lexical analysis on each source line, producing thus a sequence of tokens. Irrelevant spaces, tab and underscore characters are removed. Parsing is the next step. A correct grammatical rule is expected, and exhaustive checks are performed, to ensure compliance with the definition of the language. When all of the MCL source has been read, `IF` and `LOOP` statements are checked for proper nesting. The validity of `MOVE TO` statements (in relation to `IF`s and `LOOP`s) is verified. If everything is is in order, an incomplete object code is created and placed in a temporary scratch file. Passes 2 and 3 are skipped if any errors are detected.

Pass 2 The scratch file produced in pass 1 is read, and filled in with the exact arguments of the `$MOVE`, `$TESTMOVE`, `$ALLOC` and `$DEALLOC` instructions. Some other things that were left out (eg. memory requirements for each subprogram) are filled in as well. This pass produces full object code for each subprogram.

Pass 3 The third pass checks whether there are calls to undefined subprograms, or to MCL predefined subprograms. Calls to subprograms are filled in with the appropriate arguments and the final object code is disposed in a user designated file.

3.2 MCLCOM tables

Tables are one dimensional integer, real or character arrays, used to store certain information. Most of them are connected, in the sense that

- they contain information related to the same subject,
- they are dimensioned by the same parameters and use the same counters, and
- they are updated simultaneously.

Every table has an associated counter indicating the number of entries in the table, and a parameter, that sets the maximum number of entries in the table. As compilation of an MCL source file proceeds, some of the tables are updated, or enlarged. If the number of entries in a table, surpass its maximum (a table overflow), compilation stops and an informative error message appears as:

```
**** The <t> table has overflown in routine <r>.
**** Array <a> was assigned only <n> storage locations.
**** This array is dimensioned by the <p> parameter.
```

where ⟨t⟩ is the name of the table that overflowed, ⟨r⟩ is the routine name which caused the overflow, ⟨a⟩ is the name of the array implementing the table, ⟨n⟩ is the number of its storage locations and ⟨p⟩ is the name of the parameter that dimensions the table. In such a case one may redimension the appropriate table and recompile MCLCOM. In what follows `IDLEN` is the maximum length of an identifier, currently set to 30.

3.2.1 The token table

Declared as: `INTEGER TOKEN(MXTOK)`

Counter: `NTOK` in common block `MCLCNT`.

Maximum: Parameter `MXTOK`.

Description: The token table is constructed by the lexical analysis routine `LEX`. It contains the integer codes, corresponding to the tokens of the MCL statement being processed. For a list of all tokens, see the description of subroutine `LEX` in section 6.3.

3.2.2 The lexical analysis NUM table

Declared as: `REAL NUM(MXNUM) or DOUBLE PRECISION NUM(MXNUM)`

Maximum: Parameter `MXNUM`.

Description: The NUM table holds all numeric constants that appear in an MCL statement. In addition, for an identifier or a string, it stores the starting and ending character positions of the item, in the ALPHA table. The table type (`REAL` or `DOUBLE PRECISION`) is determined at installation time. The table is constructed by the lexical analysis routine `LEX`.

3.2.3 The lexical analysis ALPHA table

Declared as: `CHARACTER*(LINLEN) ALPHA)`

Maximum: Parameter `LINLEN`.

Description: The ALPHA table is constructed by the lexical analysis routine `LEX` and stores all identifiers and strings of the current MCL source statement. Identifiers are always stored in upper case.

3.2.4 The label table

Declared as: `CHARACTER*(IDLEN) LABEL(MXLAB)`

Counter: `NLAB` in common block `MCLCNT`.

Maximum: Parameter `MXLAB`.

Description: The label table stores all label names encountered in an MCL program or subprogram. The names are stored in ascending order.

3.2.5 The label-location table

Declared as: `INTEGER LABLOC(MXLAB)`

Counter: `NLAB` in common block `MCLCNT`.

Maximum: Parameter `MXLAB`.

Description: The label-location table is maintained in conjunction with the label table. Each time an entry is made in the label table, a corresponding entry is made in the label-location table, indicating the position of the label in the scratch file, created during the first pass of the compiler.

3.2.6 The move table

Declared as: `CHARACTER*(MXMOV) MVELAB(MXMOV)`

Counter: `NMOVE` in common block `MCLCNT`.

Maximum: Parameter `MXMOV`.

Description: The move table holds the names of the labels that appear in every `MOVE TO` statement in an MCL program or subprogram. The table is not sorted.

3.2.7 The type table

Declared as: `INTEGER TYPE(MXTYP)`

Counter: `NTYPE` in common block `MCLCNT`.

Maximum: Parameter `MXTYP`.

Description: Each entry in the type table indicates that a statement with special significance has been reached. It is updated sequentially as program compilation proceeds. The entries in this table may be positive or negative. A positive entry means that the statement in question, will generate code (a `$MOVE`, `$TESTMOVE`, `$ALLOC` or `$DEALLOC` instruction) that expects a number written in the line that follows it. The required number, is not calculated immediately, but later on, however during the first pass of the compiler. This process is called a link. A negative entry means either that the statement needs no link, or that the link will be performed during the second pass. The values that the type table elements may take on are:

1 for an `IF` statement. This will be changed later to -1 when the corresponding `END IF` is found.

2 for an `ELSE` statement. This will be changed later to -2 when the corresponding `END IF` is found.

-3 for an `END IF` statement. This entry is negative since an `END IF` statement produces object code, that needs no link.

4 for a `LOOP` statement. This will be changed later to -4 when the corresponding `END LOOP` statement is found.

-5 for an `END LOOP` statement. This entry is negative since an `END LOOP` produces object code that needs no link.

-6 for a `WHEN` statement. This entry is negative since the link is performed by the routine that processes `WHEN`, at the same time.

-7 for a `MOVE TO` statement. The link will be performed in pass 2, when all label definitions have been read.

8 for an `EXIT` statement. This will be changed to later to -8, when the loop inside which the `EXIT` statement resides, is terminated by an `END LOOP`.

9 for a `PROGRAM`, `SUB` or `END` statement, indicating that a memory allocation or deallocation takes place. This will be changed to -9 at the end of pass 1.

3.2.8 The type-definition table

Declared as: `INTEGER TDEF(MXTYP)`

Counter: `NTYPE` in common block `MCLCNT`.

Maximum: Parameter MXTYP.

Description: The type-definition table is updated in conjunction with the type table. For each entry in the type table, the corresponding entry in the type-definition table indicates in which line of the scratch file, the linked number should be written. This line is intentionally left blank in the first pass.

3.2.9 The type-link table

Declared as: `INTEGER TREL(MXTYP)`

Counter: `NTYPE` in common block `MCLCNT`.

Maximum: Parameter `MXTYP`.

Description: For each entry in the type table, the corresponding entry in the type- link table, is the actual number that will be written (in pass 2) in the line left blank. (This line is pointed to by the corresponding `TDEF` entry.) `TREL` values may be calculated either in pass 1 (for `TYPE`s of 1, 2, 4, 5, 6, 8, corresponding to the `IF, ELSE, LOOP, END LOOP, WHEN` and `EXIT` statements), or in pass 2 (for a `TYPE` of 7, corresponding to the `MOVE TO` statement.)

3.2.10 The var table

Declared as: `CHARACTER*(IDLEN) VAR(MXVAR)`

Counter: `NVAR` in common block `MCLCNT`.

Maximum: Parameter `MXVAR`.

Description: The var table keeps the names of the identifiers used as simple variables, arrays and statement functions in an MCL subprogram. It also keeps all MERLIN variables and arrays, as well as the intrinsic functions of the language. These names are sorted in ascending order in the array. Initially, this array contains only items that need not be declared via a `VAR` or `FUNCTION` declaration statement (MERLIN variables, arrays and intrinsic functions), and is in general, enlarged only when MCLCOM processes a `VAR` or `FUNCTION` declaration. However, when a variable that has not been declared is found, it is inserted in the var table, in order to avoid further error messages that might be issued (for the same undeclared variable). This table is always searched via subroutine `LOCAN`, while subroutine `INSERT` inserts information is inserted to it, (and to its related tables).

3.2.11 The kind table

Declared as: `INTEGER KIND(MXVAR)`

Counter: `NVAR` in common block `MCLCNT`.

Maximum: Parameter `MXVAR`.

Description: For each entry in the var table the values of the kind table elements are:

1	for a simple variable, declared in a `VAR` statement.
101	for a simple variable, being a dummy argument in a subprogram.
2	for an array declared in a `VAR` statement.
102	for an array being a dummy argument in a subprogram.
3	for a statement function, declared in a `FUNCTION` statement.
4	for a Merlin variable such as `VALUE, DIM, PCOUNT` etc.
5	for a Merlin array such as `R, L, GRAD` etc.
6	for an intrinsic function such as `ABS, MOD, TRUNC` etc.
-1	for a variable that has not been declared. No other information is available for this variable.

This array is updated simultaneously with the var table.

3.2.12 The memory location table

Declared as: `INTEGER LOC(MXVAR)`

Counter: `NVAR` in common block `MCLCNT`.

Maximum: Parameter `MXVAR`.

Description: The memory location table keeps memory locations, associated with the var table entries. The values it may take on are as follows:

- For a simple variable, (KIND=1) this is the memory location occupied by the variable. As explained in chapter 4, every subprogram has its own private block of memory, for storing local variables. This is the relative position of the variable, in the block.
- For an array, (KIND=2) this is the first memory location that the array occupies. As for KIND=1, this just is a relative position.
- For a dummy subprogram argument (KIND=101 or 102), this is its serial number in the argument list. 1 for the first argument, 2 for the second, etc.

- For a statement function, (KIND=3) this is the total amount of memory, used so far, for variables and statement function arguments. Storage reservation for the arguments of this statement function starts at the next memory location.
- For MERLIN variables and arrays, as well as for intrinsic functions (KIND=4, 5 and 6 respectively) the value of the corresponding entry is 0.

This array is updated in conjunction with the var table.

3.2.13 The dimensionality table

Declared as: `INTEGER DIM(MXVAR)`

Counter: `NVAR` in common block `MCLCNT`.

Maximum: Parameter `MXVAR`.

Description: The dimensionality table keeps the dimensions for each one of the var table entries. Its values are:

- For a simple variable declared in a `VAR` statement (KIND=1), 0.
- For a simple variable, being a dummy argument in a subprogram (KIND=101), 0.
- For an array declared in a `VAR` statement (KIND=2), the dimensionality of the array.
- For an array, being a dummy argument in a subprogram (KIND=102), its dimensionality. If the last dimension of the array, has an adjustable upper bound, the negative of its dimensionality.
- For a statement function (KIND=3), the number of its arguments.
- For a MERLIN variable (KIND=4), 0.
- For a MERLIN array (KIND=5), its dimensionality.
- For an intrinsic function (KIND=6), the number of its arguments; 0 if the function has a variable number of arguments. (such as `MAX, MIN` etc.)

This array is updated in conjunction with the var table.

3.2.14 The position table

Declared as: `INTEGER IPOS(MXVAR)`

Counter: `NVAR` in common block `MCLCNT`.

Maximum: Parameter `MXVAR`.

Description: For each var table entry, the corresponding position table entry represents a pointer to the information table. (Only if KIND=2, 102 or 3.) When KIND=1, 4, 5 or 6 its value is 0. This array is updated in conjunction with the var table.

3.2.15 The information table

Declared as: `INTEGER INFO(MXINFO)`

Counter: `NINF` in common block `MCLCNT`.

Maximum: Parameter `MXINFO`.

Description: Information is kept here for entries in the var table with a KIND of 2, 102 or 3, (arrays, dummy argument arrays, or statement functions). Information in this array is organized as follows: For an array, (KIND=2) with n dimensions, $2n$ table elements are occupied. The first two elements are the lower and upper bounds respectively, for the first array dimension. The next two elements are the lower and upper bounds respectively, for the second array dimension, etc. Consider an array declaration of the form: `VAR A[1:7;-4:10]` This would occupy 4 elements in the information table. Their contents would be: 1, 7, -4 and 10 respectively. For a statement function 2 elements are occupied. They point at the lines where code for this function starts, (the first element) and ends, (the second element) in the statement function file. The position in the information table where these words are placed is pointed at by the corresponding entry in the position table.

3.2.16 The statement function argument table

Declared as: `CHARACTER*(IDLEN) STARG(MXARG)`

Counter: `NFARG` in common block `MCLCNT`.

Maximum: Parameter `MXARG`.

Description: The statement function argument table stores the names of a statement function's arguments, while parsing a `FUNCTION` declaration. In all other cases this array is obsolete. However, since the same routine (`XPR`) is used to parse the expression in a `FUNCTION` declaration, as in any other case, the presence of this array is necessary as an argument to the above routine (even if it not used). The names are stored in this array in the order they appear in the `FUNCTION` declaration. (The first argument occupies the first position in the array, etc.) For each new `FUNCTION` declaration, the array is filled with new names.

3.2.17 The loop table

Declared as: `INTEGER LLOC(MXLOO)`

Counter: `NLOO` in common block `MCLCNT`.

Maximum: Parameter `MXLOO`.

Description: The loop table stores the memory location of the loop control variable. If the loop control variable is a dummy argument in a subprogram, its negative serial number in the argument list is entered in the table.

3.2.18 The loop memory table

Declared as: `INTEGER LMEM(MXLOO)`

Counter: `NLOO` in common block `MCLCNT`.

Maximum: Parameter `MXLOO`.

Description: For each loop in the MCL source, 5 subsequent temporary memory locations are needed. (See also §5.2 and §6.47.) Each entry in this array corresponds to a loop and is the first one of the above 5 memory locations. Entries in the table are always relative to the memory block assigned to the current subprogram

3.2.19 The externals table

Declared as: `CHARACTER*(IDLEN) EXT(MXEXT)`

Counter: `NEXT` in common block `MCLCNT`.

Maximum: Parameter `MXEXT`.

Description: The externals table stores the names of every subprogram found in an MCL source file. Although a main program has no name, it occupies an entry with the name '`..`'.

3.2.20 The externals definition table

Declared as: `INTEGER EXTDEF(MXEXT)`

Counter: `NEXT` in common block `MCLCNT`.

Maximum: Parameter `MXEXT`.

Description: For each entry in the externals table, the externals definition table stores the line number of the scratch file, where object code for the corresponding subprogram starts.

3.2.21 The externals-argument table

Declared as: `INTEGER NARDEF(MXEXT)`

Counter: `NEXT` in common block `MCLCNT`.

Maximum: Parameter `MXEXT`.

Description: The externals-argument table stores the number of arguments, present in the definition of the corresponding subprogram.

3.2.22 The subprogram usage table

Declared as: `INTEGER SUBUSE(MXEXT)`

Counter: `NEXT` in common block `MCLCNT`.

Maximum: Parameter `MXEXT`.

Description: Each entry in the subprogram usage table is either 1 or 0. An value of 1, means that the corresponding entry in the externals table, is referenced by one or more `CALL` statements. 0 means that the corresponding subprogram is never referenced.

3.2.23 The call table

Declared as: `CHARACTER*(IDLEN) CAL(MXCAL)`

Counter: `NCAL` in common block `MCLCNT`.

Maximum: Parameter `MXCAL`.

Description: The call table stores the names of the subprograms, referenced in `CALL` statements. Each entry in this table, corresponds to a different `CALL` statement.

3.2.24 The call-definition table

Declared as: `INTEGER CALDEF(MXCAL)`

Counter: `NCAL` in common block `MCLCNT`.

Maximum: Parameter `MXCAL`.

Description: For each entry in the call table, the call definition table stores the line number of the scratch file, where the argument of the `$GOSUB` instruction should be written.

3.2.25 The call-argument table

Declared as: `INTEGER NARCAL(MXCAL)`

Counter: `NCAL` in common block `MCLCNT`.

Maximum: Parameter `MXCAL`.

Description: For each entry of the call table, the call-argument table, stores the number of arguments, supplied in the `CALL` statement.

3.3 MCLCOM files

All files used by MCLCOM are described here. Their unit numbers are determined at run time (see subroutine `NUNIT`), and are stored in common block `MCLUNT`. If these file cannot be opened or closed properly, the program will issue appropriate informative messages and terminate.

3.3.1 The input file

This is the file in which the MCL source resides. Its name is input to the compiler. This file is opened at the beginning of the program and its contents are never modified.

3.3.2 The error list file

This is the file in which all MCL error messages are written. Its name is input to the compiler. This file is opened at the beginning of the program. Upon a successful compilation (no MCL errors) this file is closed and deleted. If the user invokes the compiler with the default parameters (no error list file), or explicitly specifies `E=*` in the command line, this file is not opened. Instead, all error messages are printed on the standard output.

3.3.3 The object code file

This is the file in which the final MERLIN object code will be written. Its name is input to the compiler. If any fatal errors are detected, this file is closed and deleted.

3.3.4 The scratch file

This is the intermediate object code file, generated during the first pass of the compiler. It is opened with `STATUS='SCRATCH'` and deleted upon program termination. The number of lines written on this file is counted by variable `BLOC` in the common block `MCLDTA`.

3.3.5 The binary file

This is an intermediate object code file, generated after completion of the second pass of the compiler. It is opened with `STATUS='SCRATCH'` and deleted upon program termination.

3.3.6 The statement function file

Object code for all statement functions in the MCL source is written in this file. It is opened with `STATUS='SCRATCH'` and is deleted upon program termination. The line numbers where the code for a statement function starts and ends are kept in the information table. (See also §3.2.15.) Note that the contents of this file are created when the `FUNCTION` statements are compiled.

3.3.7 The spare file

This is a file used in several occasions. It assists code generation in an assignment, in a `GET` statement ...etc. It is always opened with `STATUS='SCRATCH'` and deleted when no longer needed.

3.3.8 The statement function help file

This file is used while generating code for a statement function. It is opened with `STATUS='SCRATCH'` at the beginning of the program and deleted upon program termination.

3.3.9 Other files

Some subroutines require the use of a few temporary files, in order to generate object code, for example subroutine `MULTI`. When needed, unit numbers are allocated by function `NUNIT`, and the temporary files are opened with `STATUS='SCRATCH'`. When the subroutine returns, it closes and deletes its temporary files.

3.4 Common block description

3.4.1 Common block FILES

Declared as: `COMMON /FILES/ IFILE, EFILE, BFILE, PDESCF`

Contains: The names of the input, error list and object code files. Their values are initialized by the `GETPAR` subroutine and are not changed during execution. All variables in this block are of type `CHARACTER*(MXFILE)`.

`IFILE` The input file.

`EFILE` The error list file.

`BFILE` The object code file.

`PDESCF` The panel description file.

3.4.2 Common block MCLCNT

Declared as:
```
COMMON /MCLCNT/ NTOK, NLAB, NVAR, NTYPE, NMOVE, NINF, NFARG, NSTL,
                NLOO, NERR, NWARN, NEXT, NCAL, TSTART
```

Contains: Integer variables, used to count the number of entries in the various tables

`NTOK` Number of tokens in the token table.

`NLAB` Number of labels in the label table.

`NVAR` Number of entries in the var table.

`NTYPE` Number of entries in the type table.

`NMOVE` Number of entries in the move table.

`NINF` Number of entries in the information table. table.

`NFARG` Number of arguments in the statement function under processing. Significant only when compiling a `FUNCTION` declaration statement.

`NSTL` Number of object code lines written in the statement function file.

`NLOO` Loop nesting level. This is also the number of entries in the loop table.

`NERR` Number of MCL errors, encountered so far in the current source line.

`NWARN` Number of warning errors, encountered in the MCL source.

`NEXT` Number of entries in the externals table.

NCAL Number of entries in the call table.

TSTART Entries in the type table, for the current subprogram start at TSTART.

3.4.3 Common block MCLDTA

Declared as: COMMON /MCLDTA/ NL, GENE, BLOC, MEMLOC, BOUNDS, DEBUG, INWHAT, EOPEN, ISERR

Contains: Various parameters, used to control compilation. Their values are initialized in the block data subprogram MCLDEF and changed as compilation proceeds.

NL Integer variable. Number of lines read from the input file, so far.

GENE Logical variable. Object code is generated according to this parameter. A value of .TRUE. is set initially to this variable, which is changed to .FALSE. each time an MCL error is encountered in the source. Subsequent code generation will be disabled.

BLOC Integer variable, counting the number of object code lines written to the spare file during the first pass of the compiler over the MCL source.

MEMLOC Integer variable. This is the number of memory locations occupied so far by simple variables and arrays.

BOUNDS Logical variable. Controls the BOUNDS option of the compiler. It is set via the GETPAR subroutine. A value of .TRUE. indicates that bound checking is to be performed to all array subscripts. (See also section 5.)

DEBUG Logical variable. Controls the DEBUG option of the compiler. It set via the GETPAR subroutine. A value of .TRUE. indicates that additional object code is to be generated for each source line, causing MERLIN to issue informative messages in case an execution time error occurs. (See also section 5.)

INWHAT Integer variable. Indicates what MCLCOM is working on:

- INWHAT = 1 means MCLCOM is compiling a main program.
- INWHAT = 2 means MCLCOM is compiling a subprogram.
- INWHAT = 0 means MCLCOM is currently reading comments and empty lines, in between two subprograms.

EOPEN Logical variable. Is value is set by subroutine FOPEN. EOPEN = .TRUE. indicates that the user supplied the E parameter in the command line, and an error list file is open. All error messages will be written to this file (unit ERL). EOPEN = .FALSE. indicates that there is no error list file, and all error messages are written to the standard output (unit=*).

ISERR Integer variable. Indicates the type of error message to be issued by subroutine ERROR.

- ISERR = 1 means a fatal error will be issued. After a fatal error message, further code generation is inhibited.
- ISERR = 0 means a warning message will be issued.

3.4.4 Common block MCLUNT

Declared as: COMMON /MCLUNT/ INP, ERL, BIN, SCR, STF, SPARE, FHELP

Contains: The unit numbers for the 8 main files the compiler uses. All variables in this block are of type integer and are assigned initial values in subroutine FOPEN.

INP The unit number assigned to the input file.

ERL The unit number assigned to the error list file.

BIN The unit number assigned to the binary file.

SCR The unit number assigned to the scratch file.

STF The unit number assigned to the statement function file.

SPARE The unit number assigned to the spare file.

FHELP The unit number assigned to the statement function help file.

ABSFI The unit number assigned to the object code file.

3.4.5 Common block MCLSCD

Declared as: COMMON /MCLSCD/ LEFTP, RIGHTP, PERIOD, COMMA, GT, LT, NE, EQ, GE, LE, PLUS, MINUS, STAR, SLASH, BISTAR, COLON, EOL, IDENT, UNSI, NOTREC, STRING, QUOTE, LBRA, RBRA, SEMI, PERCE, AND, OR, XOR, NOT, EQUAL, LBRACE, RBRACE, QMARK

Contains: The codes for each type of token recognized by the compiler. All variables in this block are of type integer and initialized in the block data subprogram LEXINI.

LEFTP Code for the left parenthesis. (( )

RIGHTP Code for the right parenthesis. ( ))

PERIOD Code for the period. (.)

COMMA Code for the comma. (,)

`GT` Code for the “greater than” symbol. (`>`)

`LT` Code for the “less than” symbol. (`<`)

`NE` Code for the “not equal” symbol. (`#`)

`EQ` Code for the assignment symbol. (`=`)

`GE` Code for the “greater or equal” symbol. (`>=`)

`LE` Code for the “less or equal” symbol. (`<=`)

`PLUS` Code for the addition sign. (`+`)

`MINUS` Code for the subtraction sign. (`-`)

`STAR` Code for the multiplication sign. (`*`)

`SLASH` Code for the division sign. (`/`)

`BISTAR` Code for the raise to a power sign. (`**`)

`COLON` Code for the colon. (`:`)

`EOL` Code for the end of a statement.

`IDENT` Code for an identifier.

`UNSI` Code for an unsigned number.

`NOTREC` Code for any symbol that is not recognized by the language.

`STRING` Code for a string.

`APOST` Code for a single quote. (`'`)

`LBRA` Code for the left bracket. (`[`)

`RBRA` Code for the right bracket. (`]`)

`SEMI` Code for the semicolon. (`;`)

`PERCE` Code for the percent sign. (`%`)

`AND` Code for the `AND` operator.

`OR` Code for the `OR` operator.

`XOR` Code for the `XOR` operator.

`NOT` Code for the `NOT` operator.

EQUAL Code for the equality operator. (`==`)

LBRACE Code for the left brace. ({)

RBRACE Code for the right brace. (})

QMARK Code for the questionmark. (`?`)

3.4.6 Common block MCLERB

Declared as: `COMMON /MCLERB/ EBUFF`

Contains: The error messages to be issued for the line under processing are kept here.

EBUFF Character*90 array. It is dimensioned by the `MAXERR` parameter defined in subroutines `ERROR` and `FEBUFF`.

3.4.7 Common block CTABLE

Declared as: `COMMON /CTABLE/ ALPHA, VAR, LABEL, MVELAB, EXT, CAL`

Contains: All character type tables.

ALPHA The lexical analysis ALPHA table.

VAR The var table.

LABEL The label table.

MVELAB The move table.

EXT The externals table.

CAL The call table.

3.4.8 Common block RTABLE

Declared as: `COMMON /RTABLE/ NUM`

Contains: The lexical analysis NUM table.

NUM The lexical analysis NUM table.

3.4.9 Common block ITABLE

Declared as: `COMMON /ITABLE/ TOKEN, KIND, LOC, DIM, IPOS, INFO, LABLOC, TYPE, TDEF, TREL, LLOC, LMEM, EXTDEF, CALDEF, NARDEF, NARCAL, SUBUSE`

Contains: All integer type tables.

`TOKEN` The token table.

`KIND` The kind table.

`LOC` The memory location table.

`DIM` The dimensionality table.

`IPOS` The position table.

`INFO` The information table.

`LABLOC` The label location table.

`TYPE` The type table.

`TDEF` The type definition table.

`TREL` The type link table.

`LLOC` The loop table.

`LMEM` The loop memory table.

`EXTEDF` The externals definition table.

`CALDEF` The call definition table.

`NARDEF` The externals argument table.

`NARCAL` The call argument table.

`SUBUSE` The subprogram usage table.

3.4.10 Common block UALLOC

Declared as: `COMMON /UALLOC/ MINU, MAXU, LASTU`

Contains: Variables used by the unit allocation routine `NUNIT`. Variables `MINU` and `MAXU` specify a range of allowed units. When function `NUNIT` is called, it will return a unit number in between these limits.

MINU The minimum unit number, the unit allocator will return.

MAXU The maximum unit number, the unit allocator will return.

LASTU The unit number returned by the last call to function `NUNIT`.

3.4.11 Common block COM1

Declared as: `COMMON /SPECHA/ STAT`

Contains: The names of all executable statements. An executable statement is one that can appear next to a `WHEN` statement (`IF`, `LOOP`, `VAR`, etc. are excluded). See chapter 2.

STAT Dimensioned as `CHARACTER*14 STAT(NES)`. Holds all executable statements. Initial values are assigned in the block data subprogram `MCLDEF`. Although the initial values have no specific order, the array is sorted before compilation begins.

3.4.12 Common block COM2

Declared as: `COMMON /COM2/ NAME, INDE`

Contains: Integer codes, used in conjunction with array `STAT`, to identify the MCL executable statements.

NAME Dimensioned as `INTEGER NAME(NES)`. Each entry of this array stores a numerical code, used to identify the corresponding `STAT` command, when a subroutine is designed to parse more that one MCL statement.

INDE Dimensioned as `INTEGER INDE(NES)`. Each entry contains the initial position of the corresponding entry in array `STAT`. For example, if `INDE(I) = K`, then the statement now stored in `STAT(I)` was originally (before `STAT` was sorted) stored in `STAT(K)`.

3.4.13 Common block SPECHA

Declared as: `COMMON /SPECHA/ TAB, ESC`

Contains: The tab and backslash characters. Tab is ignored outside a string, while the backslash is used to enter a single quote in a string. Their values are set in subroutine `INIT` and are specific for the ASCII character set.

TAB Character*1 variable. The tab character, ASCII 9.

ESC Character*1 variable. The backslash character, ASCII 92.

3.4.14 Common blocks related to the panel description file

Declared as: `COMMON /PCOMS1/ KPOINT, NCHELP, NKHELP, TPOINT, SPOINT, PONF`

Declared as: `COMMON /PCOMS2/ PIR`

Declared as: `COMMON /PCOMS3/ PKEY, KTYPE, PANSTR, PCOM, KIO`

Declared as: `COMMON /PCOMS4/ NPCOM, NPKEY, NPIR, NPSP, NPCH`

Contains: The data structures used to store panel related information. All variables are set in subroutine `PARSP` that reads and parses the MERLIN panel description file. Because the same subroutine is used by MERLIN too, some of the arrays are not used in MCL. In general, all panel commands are stored in array `PCOM`. Keywords from all panels are stored in array `PKEY`. Strings and keyword descriptions, are stored in the large character variable `PANSTR`. Entries in array `SPOINT` are used as pointers for the starting and ending positions of each entry in `PANSTR`. Default and allowed values for integer or real keywords from all panels, are stored in array `PIR`.

`KPOINT` Dimensioned as `INTEGER KPOINT(MXPCOM)`.
`KPOINT(I)` is a pointer to the first keyword entry in array `PKEY`, for the $\mathtt{I}^{th}$ panel command (`PCOM(I)`. All keywords for the $\mathtt{I}^{th}$ panel, are stored in succession in array `PKEY`. The number of keywords for `PCOM(I)` is determined by `KPOINT(I+1)` as: `KPOINT(I+1)-KPOINT(I)`. A dummy entry in always present at the end of array `KPOINT`, to allow calculation of the number of keywords for the last (`NPCOM`) panel command: `KPOINT(NPCOM+1) = NPKEY+1`.

`NCHELP` Dimensioned as `INTEGER NCHELP(2*MXPCOM)`.
MERLIN specific, it is not used by MCLCOM.

`NKHELP` Dimensioned as `INTEGER NKHELP(2*MXPKEY)`.
MERLIN specific, it is not used by MCLCOM.

`TPOINT` Dimensioned as `INTEGER TPOINT(MXPKEY)`.
`TPOINT(J)` is a pointer, to the first entry for this keyword, in array `PIR`.

`SPOINT` Dimensioned as `INTEGER SPOINT(MXPPST)`.
Array `SPOINT` stores the starting and ending character positions for all panel related strings. For the $\mathtt{L}^{th}$ string `SPOINT(2*L-1)` is the starting character position of the string in the character variable `PANSTR`. `SPOINT(2*L)` is the ending character position. Hence the contents of the $\mathtt{L}^{th}$ string are `PANSTR(SPOINT(2*L-1):SPOINT(2*L))`. Entries in array `PIR` that are pointers to `SPOINT`, always point to the starting character position.

`PONF` Dimensioned as `INTEGER PONF(MXPCOM)`.
MERLIN specific, it is not used by MCLCOM .

`PIR` Dimensioned as `REAL PIR(MXPIR)` or `DOUBLE PRECISION PIR(MXPIR)` according to installation. Array `PIR` stores information related to the allowed values of a keyword. Assuming that `K = TPOINT(J)`, is the first entry in array `PIR`, for keyword `PKEY(J)`, then:

- For a keyword that accepts integer or real values:
  - `PIR(K) = 1` The keyword has a range of allowed values, declared, for example as `[0,10]`. The entries that follow are:

`PIR(K+1)`	`1`	Interval is open at the left.
	`2`	Interval is closed at the left.
	`3`	Left limit is $-\infty$.
`PIR(K+2)`	The left limit; 0 for $-\infty$.	
`PIR(K+3)`	`1`	Interval is open at the right.
	`2`	Interval is closed at the right.
	`3`	Right limit is ∞.
`PIR(K+4)`	The right limit; 0 for $-\infty$.	
`PIR(K+5)`	The current value of the keyword.	
`PIR(K+6)`	Pointer to array `SPOINT` for the description of the keyword.	

  - `PIR(K) = 2` The keyword has a set of allowed values, declared, for example as {`1,2,3,4`}. The entries that follow are:

`PIR(K+1)`	The number of items in the set.
`PIR(K+2)`	The first one of the allowed values.
	⋮
`PIR(K+1+PIR(K+1))`	The last one of the allowed values.
`PIR(K+2+PIR(K+1))`	The current value of the keyword.
`PIR(K+3+PIR(K+1))`	Pointer to array `SPOINT` for the description of the keyword.

- For a keyword that accepts string or character values:
  - `PIR(K) = 1` Any string is allowed as value.

`PIR(K+1)`	Pointer to array `SPOINT` for the current value.
`PIR(K+2)`	Pointer to array `SPOINT` for the description of the keyword.

  - `PIR(K) = 2` This keyword has a set of allowed values.

`PIR(K+1)`	The number of items in the set.
`PIR(K+2)`	Pointer to array `SPOINT` for the first item in the set.
	⋮
`PIR(K+1+PIR(K+1))`	Pointer to array `SPOINT` for the last item in the set.

PIR(K+2+PIR(K+1))	Pointer to array SPOINT for current value.
PIR(K+3+PIR(K+1))	Pointer to array SPOINT for the description of the keyword.

PKEY Dimensioned as CHARACTER*(KEYLEN) PKEY(MXPKEY).
Array PKEY stores the name of keywords for all panels. The first keyword for the $\mathtt{I}^{th}$ panel, is KPOINT(I).

KTYPE Dimensioned as CHARACTER KTYPE(MXPKEY).
KTYPE(J) is the type of keyword PKEY(J) as defined in the panel configuration file. The following values are allowed:

- KTYPE(J) = 'R' for a keyword that accepts real values.
- KTYPE(J) = 'I' for a keyword that accepts integer values.
- KTYPE(J) = 'S' for a keyword that accepts string values.
- KTYPE(J) = 'C' for a keyword that accepts string values, without considering character case.

PANSTR Declared as CHARACTER*(MXPSTR) PANSTR.
PANSTR is a large character variable, used to store all panel related strings. Allowed values for string keywords, descriptions of keywords, etc. are stored here. PANSTR is always addressed through the pointers in array SPOINT, which in turn are addressed by entries in array PIR.

PCOM Dimensioned as CHARACTER*(KEYLEN) PCOM(MXPCOM).
PCOM(I) contains the name of the $\mathtt{I}^{th}$ panel command.

KIO Dimensioned as CHARACTER KIO(MXPKEY).
KIO(J) specifies whether keyword PKEY(J) is a normal, or MCL-returned keyword:

- KIO(J) = ' ' for a normal keyword. Keywords of this type can only accept values, for example BFGS (NOC=500).
- KIO(J) = '?' for a keyword that can return values to an MCL program. For example BFGS (FCALLS?=A). Although it is allowed to assign values to such keywords, for example BFGS (FCALLS=500), this is a waste of time, since currently, MERLIN commands only set values to these keywords, they never use them.

NPCOM Number of entries in array PCOM.

NPKEY Number of entries in array KPOINT.

NPIR Number of entries in array PIR.

NPSP Number of entries in array SPOINT.

NPCH Number of characters in PANSTR.

3.4.15 Common block EXPRST

Declared as: `COMMON /EXPRST/ TOPBUF, STACK`

Contains: Used for communication among the routines `PFIX, PUSH` and `POP`.

`TOPBUF` Top of the array `STACK`. Integer variable.

`STACK` The structure stack which is constructed and manipulated by subroutines `PUSH` and `POP`. It is an integer array dimensioned in subroutine `PFIX` as `INTEGER STACK(LINLEN)`.

3.4.16 Common block TSTXST

Declared as: `COMMON /TSTXST/ TOPST, STACK, SUCCES`

Contains: Used for communication among subroutines `XCHECK`, `PUSHDN` and `POPUP`.

`TOPST` Integer variable. Top of the array `STACK`.

`STACK` The location-of-parentheses stack which is constructed and manipulated by subroutines `PUSHDN` and `POPUP`. It is an integer array dimensioned in subroutine `XCHECK` as `INTEGER STACK(LINLEN)`.

`SUCESS` Logical variable connected to the operations done via subroutines `POPUP` and `PUSHDN` on array `STACK`.

3.4.17 Common block TSTBST

Declared as: `COMMON /TSTBST/ TOPBST, BSTACK, OK`

Contains: Used for communication among subroutines `XCHECK`, `PUSHDN` and `POPUP`.

`TOPBST` Integer variable. Top of the array `BSTACK`.

`BSTACK` The location-of-brackets stack which is constructed and manipulated by subroutines `PUSHBR` and `POPBR`. It is an integer array dimensioned in subroutine `XCHECK`.

`OK` Logical variable connected to the operations done via subroutines `POPBR` and `PUSHBR` on array `BSTACK`.

Chapter 4

The Merlin object code

4.1 Introduction

After a succesfull compilation, the Mcl compiler, outputs a series of instructions, to be executed by Merlin. For example the following short program:

```
program
   shortdis
   graddis
end
```

compiles to:

```
MCL-3.0
$PUSHC
 0
$ALLOC
$POPMB
SHORTDIS
GRADDIS
$FINISH
```

Output from the compiler, consists of all Merlin commands, plus some additional instructions, recognized only when Merlin is running an Mcl program. These special instructions begin with a dollar sign $ and are collectively called Merlin object code (moc). moc instructions are neccesary for performing tasks that are not part of the standard Merlin operating system, such as the addition of two numbers. In order to be able to store and manipulate data, moc instructions need their own private storage, and operate on special data structures. All moc instructions are recognized and executed by Merlin's subroutine `USEMCL`. The storage and data structures necessary for running an MCL program, are defined in the same subroutine, and constitute the MCL *runtime environment*. Specifically:

- For performing arithmetic and logic operations, MCL uses a data structure called a *stack*. Abstractly, a stack is a sequence of numbers, which are manipulated by two operations, `PUSH` and `POP`. Pushing a number onto the stack, adds it to the end of the sequence, called the top of the stack. The `POP` operation removes the top element from the stack. MCL uses a stack of real numbers, and the necessary storage is provided by the runtime environment in the form of a Fortran array, dimensioned as `DIMENSION STACK(LSTA)`. The exact amount of storage `LSTA` is a Merlin installation option.

- MCL allows the definition of simple variables and multidimensional arrays. The necessary storage is provided on the stack by an `ALLOC` instruction. Allocation of n storage positions on the stack increments its top by n, thus creating a free block of memory. The block is deallocated, when it is no longer needed, decrementing the top of the stack by n positions. Temporary variables, needed during execution of `LOOP` and `CALL` statements, the arguments of a statement function, etc. are allocated on the stack as well.

- The runtime environment keeps track of the top of the stack, the current MOC instruction being executed, the current memory block, etc., using a set of special integer variables, called *registers*:

  `PC` The *program counter* contains the line number of the MOC instruction being executed.

  `SP` The *stack pointer* contains a pointer to the current top of the stack. Most of the MOC instructions affect the `SP` register, by incrementing or decrementing its contents.

  `MB` The *memory base* register is a pointer to the block of memory, containing the local variables of the MCL program or subprogram, currently being executed. The `MB` register can be set or interrogated by the `$PUSHMB` and `$POPMB` instructions correspondingly. These instructions are used in conjunction with `$ALLOC`, `$DEALLOC` to provide the local storage for a main program or subprogram.

  `AL` The *argument list* register, is a pointer to a block of memory, where the addresses of the current subrpogram's arguments reside. Specifically, the address of the first argument is in `STACK(AL)`, the address of the second argument in `STACK(AL+1)`, etc. See §5.4 for a description of how arguments are passed to subprograms.

4.2 Memory and stack

4.3 Memory model for a subprogram

4.4 Object code instructions

4.4.1 Conventions

In the following paragraphs, we give a detailed description of all MOC instructions. The description of the various subroutines in chapter 6 depends on these instructions. The instructions are categorized, according to the operation, they perform. The following conventions are used:

- If an instruction needs to read one or more arguments from the MOC file, they appear separated by semicolons. For example `$IN` ; N ; *Filename*.
- The four registers are referred to as `PC`, `SP`, `MB` and `AL`.
- The top of the stack is reffered to as `STACK(SP)`, the second top of the stack as `STACK(SP-1)`, etc.
- To achieve precise and economical description, Fortran-like pseudocode is used. Two or more statements are separated by semicolons, for example `SP = SP+1` ; `STACK(SP) =` N. Emphasized names, denote temporary variables, for example *Address*. In general, *Top* refers to the value stored at the top of the stack, *Address* to some arbitrary address (relative or absolute), and *Value* to the contents of some memory or stack location.
- Memory locations are either *relative* to the contents of the `MB` register, or *absolute*. Instructions that access relative memory locations, translate them into absolute addresses through register `MB`. For example, relative memory address 3 is absolute `MB`+3.

4.4.2 Stack manipulation instructions

- `$PUSHC` ; N
  Pushes the constant N on the top of the stack.
  `SP = SP+1` ; `STACK(SP) =` N
- `$PUSH` ; *Address*
  Pushes the contents of a relative memory location on the stack.
  `SP = SP+1` ; `STACK(SP) = STACK(`*Address*`+MB)`
- `$POP` ; *Address*
  Stores the top of the stack in a relative memory location.
  `STACK(`*Address*`+MB) = STACK(SP)` ; `SP = SP-1`

- `$POPLV`
  Stores the second top of the stack, in the absolute address specified by the top of the stack.
  Address `= STACK(SP)` ; `SP = SP-1` ; *Value* `= STACK(SP)` ; `SP = SP-1` ; `STACK(`*Address*`) =` *Value*

- `$POPCONT`
  Pops the top of the stack, and stores it in the absolute address specified by the second top of the stack.
  Value `= STACK(SP)` ; `SP = SP-1` ; *Address* `= STACK(SP)` ; `SP = SP-1` ;
  `STACK(`*Address*`) =` *Value*

- `$GETCONT`
  Pushes on the stack, the contents of the absolute address, specified by the top of the stack.
  Address `= STACK(SP)` ; `STACK(SP) = STACK(`*Address*`)`

- `$PUSHMB`
  Pushes the contents of the `MB` register, on the stack.
  `SP = SP+1` ; `STACK(SP) = MB`

- `$POPMB`
  Pops the top of the stack, and stores it on the `MB` register.
  `MB = STACK(SP)` ; `SP = SP-1`

- `$PUSHINAD`
  Pushes the contents of the `AL` register, on the stack.
  `SP = SP+1` ; `STACK(SP) = AL`

- `$POPINAD`
  Pops the top of the stack, and stores it on the `AL` register.
  `AL = STACK(SP)` ; `SP = SP-1`

- `$IPUSH` ; N
  Pushes the contents of the N^{th} subprogram argument, on the stack.
  Address `= STACK(AL+`N`)` ; *Value* `= STACK(`*Address*`)` ; `SP = SP+1` ; `STACK(SP) =` *Value*

- `$IPOP` ; N
  Pops the top of the stack, and stores it in the absolute memory location assigned, to the N^{th} subprogram argument.
  Address `= STACK(AL+`N`)` ; *Value* `= STACK(SP)` ; `STACK(`*Address* `=` *Value*

- `$PUSHA` ; N
  Translates the relative address N into an absolute address, and pushes the result on the stack.
  `SP = SP+1` ; `STACK(SP) = MB+`N

4.4.3 Instructions that access MERLIN internal variables

- `$BACKUP`
  Pushes on the stack a constant, that identifies the current backup mode.
  `SP = SP+1` ; `STACK(SP) =` *Backup_Mode*

Backup_Mode is	1	for `NOBACK`.
	2	for `LASTBACK`.
	3	for `FULLBACK`.

- `$DERIVA`
  Pushes on the stack a constant, that identifies the current derivative mode.
  `SP = SP+1` ; `STACK(SP) =` *Derivative_Mode*
  Derivative_Mode is

  1 for `NOPRINT`.

  2 for `HALFPRINT`.

  3 for `FULLPRINT`.

- `$PRINTO`
  Pushes on the stack a constant, that identifies the current printout mode.
  `SP = SP+1` ; `STACK(SP) =` *Printout_Mode*
  Printout_Mode is

  1 for `ANAL`.

  2 for `NUMER`.

  3 for `FAST`.

  4 for `MIXED`.

- `$PROCES`
  Pushes on the stack a constant, that identifies the current processing mode (`IAF` or `BATCH`).
  `SP = SP+1` ; `STACK(SP) =` *Processing_Mode*
  Processing_Mode is

  1 for `BATCH`.

  2 for `IAF`.

- `$FUNMODE`
  Pushes on the stack a constant, that identifies the current functional form (`GENERAL` or `SOS`).
  `SP = SP+1` ; `STACK(SP) =` *Functional_Form*
  Functional_Form is

  0 for `GENERAL`.

  1 for `SOS`.

- `$TCOUNT`
  Pushes on the stack the total number of function calls.
  `SP = SP+1` ; `STACK(SP) =` *Total_Number_Of_Function_Calls*

- `$PCOUNT`
  Pushes on the stack the partial number of function calls.
  `SP = SP+1` ; `STACK(SP) =` *Partial_Number_Of_Function_Calls*

- `$GTCOUNT`
  Pushes on the stack the total number of gradient calls.
  `SP = SP+1` ; `STACK(SP) =` *Total_Number_Of_Gradient_Calls*

- `$GPCOUNT`
  Pushes on the stack the partial number of gradient calls.
  `SP = SP+1` ; `STACK(SP) =` *Partial_Number_Of_Gradient_Calls*

- `$HTCOUNT`
  Pushes on the stack the total number of Hessian calls.
  `SP = SP+1` ; `STACK(SP) =` *Total_Number_Of_Hessian_Calls*

- `$HPCOUNT`
  Pushes on the stack the partial number of Hessian calls.
  `SP = SP+1` ; `STACK(SP) =` *Partial_Number_Of_Hessian_Calls*

- `$DIM`
  Pushes on the stack the dimensionality of the objective function.
  `SP = SP+1` ; `STACK(SP) =` *Dimensionality*

- `$TERMS`
  Pushes on the stack the number of terms, when the objective function has the sum-of-squares form.
  `SP = SP+1` ; `STACK(SP) =` *Number_Of_Terms*

- `$VALUE`
  Pushes on the stack the current value of the objective function.
  `SP = SP+1` ; `STACK(SP) =` *Current_Value*

- `$PRECISION`
  Pushes on the stack the machine's accuracy, as determined by MERLIN.
  `SP = SP+1` ; `STACK(SP) =` *Machine_Accuracy*

- `$X`
  Pushes on the stack the component of the current point, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = POINT(`*Index*`)`

- `$GRAD`
  Pushes on the stack the component of the gradient vector, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = GRAD(`*Index*`)`

- `$MARG`
  Pushes on the stack the component of the margin status array, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = MARG(`*Index*`)`

- `$R`
  Pushes on the stack the left margin of the minimization variable, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = XLL(`*Index*`)`

- `$L`
  Pushes on the stack the right margin of the minimization variable, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = XRL(`*Index*`)`

- `$FIX`
  Pushes on the stack the fix status of the minimization variable, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = XRL(`*Index*`)`

- `$FLAG`
  Pushes on the stack the numerical flag, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = FLAG(`*Index*`)`

- `$STEP`
  Pushes on the stack the search step of the minimization variable, specified by the value on the top of the stack.
  Index `= STACK(SP)` ; `STACK(SP) = STEP(`*Index*`)`

- `$TERM`
  If the objective function has a sum-of-squares functional form, it pushes on the stack, the term specified by the value on the top of the stack. Otherwise is pushes 0.
  Index `= STACK(SP)` ; `STACK(SP) = TERM(`*Index*`)`

4.4.4 Instructions that perform arithmetic and logic operations

All of the following instructions, operate on the top and/or second top of the stack. They pop their arguments from the stack, perform the appropriate operation, and push the result back on the stack. In the description that follows, Top refers to the top of the stack, while $Next$ refers to the second top of the stack.

- `$+` Addition
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ; `STACK(SP) =` $Top + Next$

- `$-` Subtraction
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ; `STACK(SP) =` $Next - Top$

- `$*` Multiplication
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ; `STACK(SP) =` $Top * Next$

- `$/` Division
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ; `STACK(SP) =` $Next/Top$

- `$**` Raise to a power
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ; `STACK(SP) =` $Next^{Top}$

- `$=` Test for equality
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top = Next$ then `STACK(SP) = 1` else `STACK(SP) = 0`

- `$#` Test for non equality
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top \neq Next$ then `STACK(SP) = 1` else `STACK(SP) = 0`

- `$>` Test for greater than
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top > Next$ then `STACK(SP) = 1` else `STACK(SP) = 0`

- `$<` Test for less than
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top < Next$ then `STACK(SP) = 1` else `STACK(SP) = 0`

- `$>=` Test for greater or equal than
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top \geq Next$ then `STACK(SP) = 1` else `STACK(SP) = 0`

- `$<=` Test for less or equal than
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top \leq Next$ then `STACK(SP) = 1` else `STACK(SP) = 0`

- `$OR` Logical OR
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top \neq 0$ then `STACK(SP) = 1` else `STACK(SP) =` $Next$

- `$AND` Logical AND
  Top `= STACK(SP)` ; `SP = SP-1` ; $Next$ `= STACK(SP)` ;
  If $Top \neq 0$ then `STACK(SP) =` $Next$ else `STACK(SP) = 0`

- `$XOR` Logical XOR
  Top `= STACK(SP)` ; `SP = SP-1` ; *Next* `= STACK(SP)` ;
  If $Top = 0$ and $Next \neq 0$ then `STACK(SP) =` *Next*
  else if $Top \neq 0$ and $Next = 0$ then `STACK(SP) =` *Top*
  else `STACK(SP) = 0`

- `$NOT` Logical NOT
  Top `= STACK(SP)` ; If $Top = 0$ then `STACK(SP) = 1` ; else `STACK(SP) = 0`

4.4.5 Instructions that implement predefined functions

The following instructions, implemement the MCL predefined functions.

- `$ABS` Absolute value
  Top `= STACK(SP)` ; `STACK(SP) =` $|\,Top\,|$

- `$LOG10` Base 10 logarithm
  Top `= STACK(SP)` ; `STACK(SP) =` $\log_{10} Top$

- `$LOG` Natural logarithm
  Top `= STACK(SP)` ; `STACK(SP) =` $\log Top$

- `$SIN` Sin
  Top `= STACK(SP)` ; `STACK(SP) =` $\sin Top$

- `$COS` Cosine
  Top `= STACK(SP)` ; `STACK(SP) =` $\cos Top$

- `$TAN` Tangent
  Top `= STACK(SP)` ; `STACK(SP) =` $\tan Top$

- `$ASIN` ArcSin
  Top `= STACK(SP)` ; `STACK(SP) =` $\arcsin Top$

- `$ACOS` ArcCos
  Top `= STACK(SP)` ; `STACK(SP) =` $\arccos Top$

- `$ATAN` ArcTangent
  Top `= STACK(SP)` ; `STACK(SP) =` $\arctan Top$

- `$SINH` Hyperbolic Sin
  Top `= STACK(SP)` ; `STACK(SP) =` $\sinh Top$

- `$COSH` Hyperbolic Cosine
  Top `= STACK(SP)` ; `STACK(SP) =` $\cosh Top$

- `$TANH` Hyperbolic Tangent
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` $\tanh Top$
- `$ASINH` Hyperbolic ArcSin
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` $\text{arcsinh}\, Top$
- `$ACOSH` Hyperbolic ArcCosine
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` $\text{arccosh}\, Top$
- `$ATANH` Hyperbolic ArcTangent
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` $\text{arctanh}\, Top$
- `$MIN` ; N
  Calculates the minimum of the N topmost stack positions. All N numbers are popped from the stack, while the result is pushed back on the stack.
- `$MAX` ; N
  Calculates the maximum of the N topmost stack positions. All N numbers are popped from the stack, while the result is pushed back on the stack.
- `$TRUNC` Truncation of a real number
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` `INT(`*Top*`)`
- `$ROUND` Rounding of real number
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` `NINT(`*Top*`)`
- `$MOD` Modulus
  Top `=` `STACK(SP)` ; *Next* `=` `STACK(SP)` ; `STACK(SP)` `=` $Next \bmod Top$
- `$MEAN` ; N
  Calculates the mean of the N topmost stack positions. All N numbers are popped from the stack, while the result is pushed back on the stack.
- `$SQRT` Square root
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` $\sqrt{Top}$
- `$EXP` Exponential function
  Top `=` `STACK(SP)` ; `STACK(SP)` `=` e^{Top}
- `$RAN` Random number
  Returns a random number, from a uniform distribution in (0,1). The argument on the top of the stack is necessary, but ignored. *Top* `=` `STACK(SP)` ; `STACK(SP)` `=` *Random_Number*
- `$GRMS` RMS gradient
  Returns the RMS gradient, calculated as

$$GRMS = \sqrt{\frac{\sum_{i=1}^{N} g_i^2}{N}}$$

with N being the dimensionality of the objective function, and g_i the i^{th} component of the gradient vector. The argument on the top of the stack is necessary, but ignored.

- `$FACT` Factorial
  Top `= STACK(SP)` ; `STACK(SP) =` *Top* !

- `$GRADNORM`
  Returns the L_1, L_2 or L_∞ norms of the gradient vector. The top of the stack specifies which norm to use:

  -1 Calculate L_∞

  1 Calculate L_1

  2 Calculate L_2

  Top `= STACK(SP)` ; `STACK(SP) =` *Gradient_Norm*

4.4.6 Instructions that implement predefined subprograms

All instructions that implement predefined subprograms, operate exactly as a normal MCL subprogram would. They obtain the address of the arguments, using the `AL` register,

- `$GETSEED`
  Implements the `GETSEED` predefined subprogram. Its MCL definition is `SUB GETSEED ( W[1:25] )`.
  It returns an array of 25 integers, used to restart the random number generator.
  Address `= STACK(AL)` ; Do `I=1,25` ; `STACK(`*Address*`+I-1) =` $Seed_i$; End Do

- `$SETSEED`
  Implements the `SETSEED` predefined subprogram. Its MCL definition is `SUB SETSEED ( W[1:25] )`.
  It requires as input, an array of 25 integers, used to restart the random number generator.
  Address `= STACK(AL)` ; Do `I=1,25` ; $Seed_i$ `= STACK(`*Address*`+I-1)` ; End Do

- `$NORM`
  Implements the `NORM` predefined subprogram. Its MCL definition is
  `SUB NORM ( ARRAY[1:*]; N; L; RESULT )`. The norm is specified by `L` and must be positive, unless $L = -1$, which means the infinity norm L_∞.
  Address_Of_Array `= STACK(AL)` ; *Address_Of_N* `= STACK(AL+1)` ;
  Address_Of_L `= STACK(AL+2)` ; *Address_Of_Result* `= STACK(AL+3)` ;
  N `= STACK(`*Address_Of_N*`)` ; *L* `= STACK(`*Address_Of_L*`)` ;
  Calculate the requested norm ; `STACK(`*Address_Of_Result*`) =` *Norm*

4.4.7 Input/output related instructions

Filename is the name of the file involved in the operation. If the standard MERLIN input/output file is to be used, `.STANDARD` must be supplied in place of *Filename*.

- `$NOTE` ; N ; *Character* ; *Filename*
  The `NOTE` instruction puts the N characters present in *Character*, in the output buffer for the file *Filename*.

- `$OUT` ; *Filename*
  Pops the top of the stack, converts it to its character representation, and puts it in the output buffer for file *Filename*.

- `$FLUSH` ; *Filename*
  Flushes the output buffer for file *Filename*.

- `$IN` ; N ; *Filename*
  Reads N real numbers from file *Filename*, and pushes them on the stack, in reverse order, hence the first number read becomes the top of the stack. The `SP` register is incremented by N.

- `$CLOSE` ; *Filename*
  Closes the file specified by *Filename*.

4.4.8 Instructions that transfer program control

- `$MOVE` ; N
  Unconditionally transfers program control. If $N > 0$ then the object code file is forwarded by $N - 1$ lines. If $N < 0$ the the object code file is backspaced by $\mid N \mid +2$ lines.

- `$TESTMOVE` ; N
  Conditional transfer of program control. The top of the stack is popped, and if it is non-zero, then transfer of control occurs. If $N > 0$ then the object code file is forwarded by $N - 1$ lines. If $N < 0$ the the object code file is backspaced by $\mid N \mid +2$ lines.

- `$CALLSUB` ; N
  Prepares the runtime environment to execute the code for a subprogram. It pushes on the stack, the object code line number, where the subprogram should return. It then moves the object file forwards or backwards, in order to reach the subprogram. If $N > 0$ then the object code file is forwarded by $N - 1$ lines. If $N < 0$ the the object code file is backspaced by $\mid N \mid +2$ lines.
  `SP = SP+1` ; `STACK(SP) = PC+1` ; *Move_The_Object_Code_File*

- `$SUBRETURN`
  Returns from a subprogram. It pops from the stack the object code line number, where it should return. Using the current value of `PC` it calculates how many lines forward or backward it must move, in order to reach the return point.
  Old_PC `= STACK(SP)` ; `SP = SP-1` ; $Lines\_To\_Move = Old\_PC -$`PC` ;
  If $Lines\_To\_Move < 0$ then $Lines\_To\_Move = Lines\_To\_Move + 1$;
  Execute a `$MOVE` *instruction, using Lines_To_Move as an argument*

4.4.9 Miscellaneous instructions

- `$CONTINUE`
  Does nothing. Program execution continues with the next instruction.

- `$PAUSE`
  Executes a Fortran `PAUSE` statement.

- `$FINISH`
  Terminates execution of an MCL program.

- `$>>`
  This instruction introduces a comment in the object code. It is used when the compiler `DEBUG` option is set to `.TRUE.`. The characters next to the `$>>` symbol are an image of the MCL source line, that will produce the MOC instructions until the next `$>>`. It is stored by the runtime environment in a character variable and is printed if an MCL runtime error occurs.

- `$STACKSWAP`
  Swaps the contents of the two topmost stack positions.
  Top `= STACK(SP)` ; *Next* `= STACK(SP-1)` ; `STACK(SP) =` *Next* ; `STACK(SP-1) =` *Top*

- `$GETPP` ; *Panel* ; *Key* ; *Type*
  Obtains the current value of the keyword *Key* from panel *Panel* and pushes it on the stack. *Type* is the type of the keyword (`I` or `R`).

- `$WSUM` ; N
  `$WSUM` is used in order to determine the relative memory location of an array element. It performs the weighted sum of $2N$ numbers, which are popped from the stack. The result is pushed on the stack.

$$WSUM = \sum_{i=0}^{N-1} \mathtt{STACK}(\mathtt{SP} - i) * \mathtt{STACK}(\mathtt{SP} - i - N)$$

- `$LOCCHECK`
  Checks whether an array subscript is within its allowed bounds. The top of the stack contains the subscript to be checked, while the next two positions are the upper an lower bounds

respectively. If the subscript is found to be out of bounds, an MCL runtime error occurs. Otherwise the subscript is pushed back on the stack, and program execution proceeds.
Subscript `= STACK(SP)` ; `SP = SP-1` ;
Upper_Bound `= STACK(SP)` ; `SP = SP-1` ;
Lower_Bound `= STACK(SP)` ; `SP = SP-1` ;
If *Lower_Bound* $\leq$ *Subscript* $\leq$ *Upper_Bound* then `SP = SP+1` ; `STACK(SP) =` *Subscript*
else *Stop execution of the* MCL *program*

- `$BLOCKCOPY` ; *What*
  Copies one of the MERLIN internal arrays in a block of memory, starting at the absolute address, contained on the top of the stack. *What* determines which array is to be copied (`X`, `R`, `L`, `FIX`, `MARG`, `GRAD` or `STEP`).
  Address `= STACK(SP)` ; `SP = SP-1` ; *Copy the* MERLIN *array*

- `$LINE` ; N
  Announces to the runtime environment, that N is the MCL source line number, about to be executed. This instruction is used only when the `DEBUG` option of the compiler is set to `.TRUE.` The line number N is printed if an MCL runtime error occurs.

- `$IOFFSET`
  Calculates and pushes on the stack the absolute address of an array element. The array is assumed to be the N^{th} argument of the subprogram currently executing.
  N `= STACK(SP)` ; `SP = SP-1` ; *Offset* `= STACK(SP)` ; *Argument_Address* `= STACK(`N`+AL-1)` ;
  Element_Address `=` *Offset* `+ STACK(`*Argument_Address*`)` ; `STACK(SP) =` *Element_Address*

- `$ALLOC`
  Allocates a block of memory on the stack. The amount of memory is determined by the top of the stack. The memory base address for the newly created block, is pushed on the (new) top of the stack.
  Number_Of_Locations `= STACK(SP)` ; `SP = SP-1` ; *Memory_Base* `= SP` ;
  `SP = SP+`*Number_Of_Locations* ; `SP = SP+1` ; `STACK(SP) =` *Memory_Base*

- `$DEALLOC`
  Deallocates a block of memory. The amount of memory is determined by the top of the stack. The block to be deallocated is assumed to lie just beyond the top of the stack.
  Number_Of_Locations `= STACK(SP)` ; `SP = SP-1` ; `SP = SP-`*Number_Of_Locations*

Chapter 5

The main MCL structures

5.1 The block-IF statement

The syntax of a block `IF` has the form:
`IF` ⟨MclExpression⟩ `THEN` ⟨Block_Of_Lines1⟩ [`ELSE` ⟨BlockOfLines2⟩] `END IF`
The `IF` statement is reached first. Since the compiler parses the input file (and subsequently generates code) line by line, there is no way to know the existence of the `ELSE` part at this point. The code generated when the `IF` statement is reached is:

```
(code to evaluate the \su{MclExpression})
$NOT
$TESTMOVE
<blank line>
```

To indicate that an `IF` statement has been reached a value of 1 is pushed in the type table. The location, say l_1 of the above ⟨blank line⟩ (i.e. the serial number of that line in the scratch file) is pushed in the type-definition table. The purpose of the `$TESTMOVE` instruction is to cause the program to transfer control (by jumping a number -say n_1- of lines) to the `ELSE` statement following, (or `ENDIF`, if an `ELSE` is not defined for this `IF`) if the ⟨MclExpression⟩ yields a non zero value. The number (n_1) to be written in the ⟨blank line⟩ will be calculated later. As compilation of the program proceeds, the ⟨BlockOfLines1⟩ will be compiled. The code generated when the `ELSE` statement is found is:

```
$MOVE
<blank line>
```

To indicate that an `ELSE` statement has been reached a value of 2 is pushed in the type table. The location (say l_2) of the ⟨blank line⟩ is also pushed in the type-definition table. The purpose of the `$MOVE` instruction is to cause the program to jump a number of lines (say n_2) and transfer control to the corresponding `END IF`, when execution of the `BlockOfLines1` has been completed. At this point the type table will be searched backwards (meaning that the search will start at the current entry, to the beginning of the table) in order to locate a value of 1, indicating the `IF` for this `ELSE`. This entry will be set to -1, thus canceling this `IF` from further processing, and n_1 is now calculated as: $n_1 = l_2 - l_1 + 1$ (since the destination of the `$TESTMOVE` instruction is the one, following `$MOVE` instruction) `END IF` is reached, after ⟨BlockOfLines2⟩ is compiled. One line of object code is generated:

```
$CONTINUE
```

The location (say l_3) of the above line is pushed in the type- definition table, while a value of -3 (indicating the `END IF`) is pushed in the type table. The type table is then searched backwards (after a value of -3 is pushed in it) until a value of 2 (indicating an `ELSE`) or a value of 1 (indicating an `IF`) is reached. n_2 is now calculated as: $n_2 = l_3 - l_2$ for an `IF` ... `ELSE` ... `END IF` construct, while $n_2 = l_3 - l_1$ for an `IF` ... `END IF` construct (since the destination of the `$TESTMOVE`, or the `$MOVE` instruction is the above `$CONTINUE` instruction). This value is pushed in the type-link table in the position corresponding to the `IF` or `ELSE`. As an example, the following part of an MCL program:

```
IF  A > B  THEN
    SHORTDIS
    STOP
ELSE
    MEMO
    RETURN
END IF
```

will generate :

```
$PUSH
 1
$PUSH
 2
$>
$NOT
$TESTMOVE
 5
SHORTDIS
STOP
$MOVE
 3
```

```
MEMO
RETURN
$CONTINUE
```

When all of the MCL source has been processed, the compiler will test (by searching the type table) for any block ifs that do not terminate. (i.e. the corresponding `END IF` is missing.) If such a block if is found, a "fake" `END IF` (a value of -3) will be pushed in the type table, thus terminating this block if.

5.2 The LOOP statement

The loop statement is a means to easily repeat a part of an MCL program. Its syntax is:

`LOOP` ⟨LoopControlVariable⟩ `FROM` ⟨initial⟩ `TO` ⟨final⟩ [`BY` ⟨step⟩]
⟨BlockOfLines⟩
`END LOOP`

where the ⟨LoopControlVariable⟩ is the name of a simple variable, and ⟨initial⟩, ⟨final⟩ and ⟨step⟩ are MCL expressions. The way the compiler handles the `LOOP` repetitions is the following: When the `LOOP` statement is encountered the ⟨initial⟩, ⟨final⟩ and ⟨step⟩ values for the loop are calculated. Then the number of loop repetitions is calculated, (the final trip count) by the formula:

$$ftc = \max\left[0, \mathrm{int}\left(\frac{\langle final\rangle - \langle initial\rangle + \langle step\rangle}{\langle step\rangle}\right)\right]$$

The loop should be repeated ftc times. In case of a final trip count being zero, the loop won't be processed, and programs execution will continue with the statement, immediately following the corresponding `END LOOP`. The loop control variable is initially set to the value of ⟨initial⟩ while a trip counter is set to 0. Each time the loop is repeated, the loop control variable is incremented by the value of ⟨step⟩ and the trip counter is incremented by 1, until the value of ftc is reached. In such a case, the repetitions are terminated and the program proceeds with the statement immediately following the corresponding `END LOOP`. From the above discussion, it follows that 5 storage locations are needed for a single loop. These locations will hold the ⟨initial⟩, ⟨final⟩ and ⟨step⟩ values, as well as the value of ftc and the trip counter. Since these values are used only while the loop is executing, the total storage needed for all loops in the program is: 5 `MAXL`, with `MAXL` being the maximum number of nested loops encountered in the MCL source. For example the following part of an MCL program :

```
LOOP A FROM 1 TO 10 BY 2
    PAUSE
END LOOP
```

will generate:

```
$PUSHC
 1.
$POP
 2
$PUSHC
 10.
$POP
 3
$PUSHC
 2.
$POP
 4
$PUSH
 3
$PUSH
 2
$SUB
$PUSH
 4
$ADD
$PUSH
 4
$DIV
$TRUNC
$PUSHC
 0
$MAX
 2
$POP
 6
$PUSH
 2
$POP
 1
$PUSHC
 1
$POP
 5
$PUSH
 5
```

```
$PUSH
 6
$>
$TESTMOVE
 18
$PAUSE
$PUSH
 1
$PUSH
 4
$ADD
$POP
 1
$PUSHC
 1
$PUSH
 5
$ADD
$POP
 5
$MOVE
 -22
$CONTINUE
```

When all of the MCL source has been processed, the compiler will test (by searching the type table) for any loops that do not terminate. (i.e. the corresponding `END LOOP` is missing.) If such a loop is found a "fake" `END LOOP` (a value of -6) will be pushed in the type table, thus terminating this loop.

5.3 The main program

5.4 Subprograms

The definition of an MCL subprogram has the form:
`SUB` ⟨Subprogram_Name⟩ [`(` ⟨Argument_List⟩ `)`]
The ⟨Argument_List⟩ resembles a `VAR` declaration, the only difference being that arrays may have an adjustable upper bound for their last dimension.

As for a main program, an entry is made in the externals table.

Chapter 6

Detailed description of the program

6.1 Some conventions

In what follows, we give a detailed description of every subprogram.

6.2 Frequently used variables

6.3 LEX

Calling sequence

```
CALL LEX ( LINE, TOKENS, ALLTOK, NUMBC, ALPHAC )
```

Purpose

Lexical analyzer: This routine scans a given sequence of characters and determines its structure (by keeping a code for every token found, together with some additional information). Packing is also performed here: blanks and underscores are ignored when detected outside strings.

Description of arguments

Input arguments

`LINE` Declared as `CHARACTER*(*) LINE`. The sequence of characters to be lexically analyzed.

Output arguments

TOKENS Integer one dimensional array, which upon return, holds the structure of LINE in the form of numeric codes.

ALLTOK Integer variable, holding upon return the total number of different syntactic entities (tokens) detected in LINE.

NUMBC Real one dimensional array, which holds the numeric contents (if any) of LINE.

ALPHAC Character*(IDLEN) one dimensional array, which holds the alphabetic contents (if any) of LINE.

Other subprograms called

Subroutines PSHSTR, PSHNUM, PSHALP.

Functions LENGTH, LXTRAN.

Global data accessed (A) or modified (M): TTSTR (M), TTALP (M), TTNUM (M), ALPH (M), STRUC (M), NUMB (M), MCLSCD (A).

Information on principal data structures and work areas used

FINGER A pointer to the character of LINE currently under processing. IDBUF A character*(IDLEN) variable, where the symbolic name of a detected identifier is stored prior to its final assignment to ALPHAC. STBUF Character*(2*IDLEN) variable, serving as a temporary storage for strings.

Notes

A string (anything between single quotes) is kept as a doublet, into ALPHAC on two adjacent positions. Its length is kept in NUMBC. A mini-parsing is performed so as to determine if a character sequence constitutes a valid string. The additional information, which we need to know in order to completely and unambiguously define a lexical item is the so- called value of the lexical item. It depends on its nature. Specifically, the values are character-sequences for identifiers and real-numbers for numeric-constants. For a string, its value consists of both its length and the delimited by quotes character-sequence (without the quotes). Operators and key- symbols have special codes and no value. The token-codes used are shown in table 6.3.

Code	Token	Code	Token
1	(	18	Identifier
2	)	19	Real number
3	.	20	Unrecognizable item
4	,	21	String
5	>	22	' (Apostrophe)
6	<	23	[
7	#	24	]
8	=	25	;
9	>=	26	%
10	<=	27	AND
11	+	28	OR
12	-	29	XOR
13	*	30	NOT
14	/	31	:=:
15	**	32	{
16	:	33	}
17	End of line		

Table 6.1: Lexical token etc

Outline of the Lexical Analysis Algorithm

1. Initially a stream of characters (named `LINE`) is input to `LEX`.
2. Then for each character of `LINE`, do the following:
   (a) Check whether it is alphabetic. If so, accept any sequence of characters which obey the syntax of ⟨Identifier⟩. Then, append the formulated Identifier to the `ALPHAC` array, and update the `TOKENS` array. Proceed to the next character.
   (b) Check whether it is numeric. If so, accept any sequence of characters which obey the syntax of ⟨Real⟩. Then, append the formulated Real Number to `NUMBC` array, and update the `TOKENS` array. Proceed to the next character.
   (c) Check whether it is a Special Character or a Space. If so, then if a space, ignore it, else, if a colon, check if it is part of '`:=:`', and act accordingly. Update the `TOKENS` array. Proceed to the next character.
   (d) Check whether it is a '`*`'. If so, check if it is part of '`**`', and act accordingly. Update the `TOKENS` array. Proceed to the next character.
   (e) Check whether it is '>'. If so, check if it is part of '>=', and act accordingly. Update the `TOKENS` array. Proceed to the next character.

(f) Check whether it is '<'. If so, check if it is part of `'<='`, and act accordingly. Update the `TOKENS` array. Proceed to the next character.

(g) Check whether it is a Single Quote. If so, locate the next single quote, and put the string contents to the `ALPHAC` array. Update the `TOKENS` array. Proceed to the next character.

(h) Check whether it is `'%'`. If so, goto 3.

(i) If none of the above is satisfied, then sign that an unrecognizable character has been detected, and proceed to the next character.

3. Update for the last time `TOKENS` by sending an end of line code.

4. Stop.

6.4 XGEN

Calling sequence

```
CALL XGEN ( EXPR, NTOK, NA, NN, CORREC, FDECLR, BLOC, WORKFL, STVAR )
```

Purpose

Code Generator for MCL expressions. This routine generates the necessary code for the evaluation of MCL expressions. It checks for the proper MCL function and array referencing, and performs the necessary operations for linking functions when called. It also calculates the MEMORY (the array of MERLIN -2.0 interface package) location in case a multi-dimensional MCL array is referenced.

Description of arguments

Input arguments

`EXPR` The expression to be processed (integer array) in a hybrid-postfix form (see subroutine PFIX for details on the hybrid form) .

`NTOK` The number of different tokens residing into EXPR (integer).

`NA` Integer pointer to the last used element of ALPHAC.

`NN` Integer pointer to the last used element of NUMBC.

`FDECLR` Logical variable, being .TRUE. only when XGEN is called during an MCL FUNCTION declaration processing.

WORKFL Input variable, keeping the unit number of a scratch file on which code is to be written.

STVAR The statement function argument table.

Input / Output arguments

CORREC Logical variable, being .TRUE. upon entry, signifying that EXPR up to now is correct. If an error occurs during code generation, it becomes .FALSE. and is returned to XPR (the calling routine).

BLOC Integer variable, holding the total number of lines written to the WORKFL file, or in case FDECLR is .TRUE., the number of lines generated for the function under processing.

Where called from: Subroutine XPR.

Other subprograms called

Subroutines LOCAN, SERLOC, ERROR.

Global data accessed (A) or modified (M): MCLDTA (A), MCLUNT (A), MCLSCD (A).

Information on principal data structures and work areas used

DIME Array holding dimensions under processing. QUALIT Array holding quality of each item under processing. COMCNT Array holding commas found. OFFSET Array holding MCL-array offsets in MERLIN-2.0. array MEMORY. VOLUME Auxiliary array for MCL-array-location calculations. LOW Array holding lower bounds for each dimension for MCL- arrays. UP Array holding upper bounds. POSIT Array holding positions of items in the var table. NDIM The maximum number of dimensions of an MCL array, and also the maximum number of arguments of an MCL function. It is closely related to LINLEN: $NDIM = \frac{LINLEN-13}{2}$.

6.4.1 Processing of MCL functions

MCL function processing, is performed by duplicating code whenever this is needed (in contrast to preprocessing source code so as to transform all function calls to the corresponding declared expressions). This is accomplished by inserting the code corresponding to the declared expression, into the position where the function is called and by using a few "linking words" of the MEMORY array of MERLIN -2.0. which rest above all normally declared and intrinsic items (i.e. above the

VARTOP word of array MEMORY). Whenever XGEN is called for function declaration processing, (i.e. when the FDECLR flag is .TRUE.) the code written, addresses to that sequence of dummy locations which are being used as linking words. Later, when a reference to a function is met, (either in executable source code or in a following declaration), a series of $POP and $PUSH instructions link those locations with the locations where the arguments reside.

As an example, consider the following case: FUNCTION F [A,B] = A * B The final code generated when the above function is referenced somewhere in the MCL program, say by the line: DISPLAY F [C1, C2] is (only the code needed for the evaluation of F is shown):

$PUSH ¡location of C1¿ $PUSH ¡location of C2¿ $POP ¡dummy location 2¿ $POP ¡dummy location 1¿ $PUSH ¡dummy location 1¿ $PUSH ¡dummy location 2¿ $MUL

The underdoted code, is the code generated during the function declaration processing. When the function reference was met, the linking pops and pushes were written (underlined code) and the existing function code was copied. The XGEN variable FSPACE is used when more complicated linking is to be done. This is the case of backward reference of a function to a function in a declaration. In this case VARTOP+FSPACE is used.

6.4.2 Processing of multi-dimensional arrays

The elements of a multi-dimensional MCL array, are stored in a specific way in the one dimensional array MEMORY of the new MERLIN -2.0. They are stored in ascending locations by columns. The first subscript increases most rapidly, and the last subscript increases most slowly. For example the elements of the MCL array declared through the next statement: VAR ARR [1 : r, 1 : c, 1 : p] where r, c, p are numbers, are loaded in array MEMORY as shown in the following descriptive FORTRAN-like code: index = 1 (or more generally =offset) DO 10, plane = p , 1, -1 DO 10, column = c, 1, -1 DO 10, row = r , 1, -1 MEMORY(index) = ARR [row, column, plane] index = index + 1 10 Continue The position of an array element can be found in array MEMORY by using the formula:

with sm = value of the subscript expression specified for dimension m jm = lower bound for dimension m kn = upper bound for dimension n. Note that XGENs OFFSET, has already the 1 added to it. Formula (1) can be transformed as:

= 1+ WSUM + LOWSUM

WSUM is formed during MERLIN -2.0. run time through the MOC instruction $WSUM (see sect. 6.1), while LOWSUM is formed by XGEN. By Volumem we denote the product:

6.4.3 The XGEN Algorithm

1. Initially a stream of codes (named Expression) is input to `XGEN`.

2. For every Item in Expression do the following:

   (a) Check whether it is an Identifier. If so, then in case `XGEN` is called for Function_Declaration processing, write special linking code; next locate the identifier in the var table and get necessary attributes: Kind, Dimensionality.

      i. Check whether Kind is Variable, and if so, write a $PUSH instruction and update the `BLOC` counter.

      ii. Check whether Kind is Array, and if so, keep current attributes for later use, in a "stack" manner. In case bounds checking is necessary, write additional checking code.

      iii. Check whether Kind is Function, and if so, keep current attributes for later use.

      iv. Check whether Kind is Intrinsic_Item, and if so, keep current attributes or write code.

   (b) Check whether it is Zero or One. If so, then, write corresponding simple code.

   (c) Check whether it is a Number. If so, then, write $PUSH, number.

   (d) Check whether it is an Operator. If so, then, write the operator.

   (e) Check whether it is a Comma. If so, then, increment the commas-counter; if Bounds checking is necessary, write additional code.

   (f) Check whether it is a Right_Bracket. If so, then, do the following: If commas counted are not equal to (DIM-1) then, Check if its one of the `MIN`, `MAX` or `MEAN` functions and if so, then ok, write code, else error. Else If commas counted are equal to (DIM-1), Check Kind from the attributes kept and if Kind is Function, then transfer and copy code (see sect. 4.2.1), else if Kind is Array, then write location processing code.

   (g) In case Item is something different than the above, then flag an error.

3. Stop.

6.5 LXTRAN

Calling sequence

```
LXTRAN ( CHAR )
```

Purpose

This routine transforms characters to specific codes to be used in routine LEX. The codes are a subset of the table shown at the end of section 4.1.

Description of arguments

Input arguments

`CHAR` The input character*1 variable for which code is asked.

Where called from: Subroutine LEX

6.6 POP, POPUP and POPBR

Calling sequence

```
CALL POP ( ITEM , EMPTY )
```

Calling sequence

```
CALL POPUP ( ITEM )
```

Calling sequence

```
CALL POPBR ( ITEM )
```

Purpose

These routines, pop numbers from a stack.

Description of arguments

Output arguments

`EMPTY` Logical variable being .TRUE. when the stack is empty.

`ITEM` The integer number to be popped.

Where called from: POP is called from subroutine PFIX. POPUP is called from subroutine XCHECK. POP BR is called from subroutine XCHECK.

6.7 PUSH, PUSHDN and PUSHBR

Calling sequence

```
CALL PUSH ( ITEM )
```

Calling sequence

```
CALL PUSHDN ( ITEM )
```

Calling sequence

```
CALL PUSH BR ( ITEM )
```

Purpose

These routines, push numbers to a stack.

Description of arguments

Input arguments

`ITEM` The integer number to be pushed.

Where called from: PUSH is called from subroutine PFIX. PUSHDN is called from subroutine XCHECK. PUSH BR is called from subroutine XCHECK.

6.8 PSHSTR, PSHNUM and PSHALP

Calling sequence

```
CALL PSHSTR ( ITEM )
```

Calling sequence

```
CALL PSHNUM ( ITEM )
```

Calling sequence

```
CALL PSHALP ( ITEM )
```

Purpose

These routines push elements in the three Lexical Analysis tables maintained by routine LEX.

Description of arguments

Input arguments

`ITEM` The input element to be pushed. It is Integer for PSH STR, Real for PSH NUM, and Character*(*) for PSH ALP.

Where called from: They are called only from subroutine LEX

6.9 LENGTH

Calling sequence

```
LENGTH ( LINE )
```

Purpose

This function determines the effective length of a string of characters. By effective length we mean the length of the string ignoring the trailing blanks.

Description of arguments

Input arguments

`LINE` The input string of characters. Character*(*) variable.

Where called from: Subroutines GETPAR, ERROR, FEBUFF, LEX.

Notes

This is a general purpose function.

6.10 PFIX

Calling sequence

`CALL PFIX ( EXPR, TOKNUM, NEWTOK )`

Purpose

Transforms MCL expressions from InFix form to HybridPostFix form.

Description of arguments

Input arguments

`EXPR`	Integer input-output array holding the tokens of the expression.
`TOKNUM`	Integer input variable, keeping the number of tokens in the EXPR array.
`NEWTOK`	Output integer variable, keeping the number of tokens left in the EXPR array.

Where called from: Subroutine XPR.

Other subprograms called

Subroutines `PUSH, POP`.

Global data accessed (A) or modified (M): EXPRST(M), MCLSCD(A).

Information on principal data structures and work areas used

`STACK` This data structure, keeps operators, parentheses, and brackets located during scanning of InFix EXPR. It is manipulated by the PUSH and POP routines. `PFBUF` This data structure holds HybridPostFix EXPR as this is gradually building. `PRIOR` This is an array which keeps the priorities of all MCL operators.

The `PFIX` Algorithm

1. Initially a stream of codes (named Expression) is input to `PFIX`.
2. For each Token Into Expression do the following:
   (a) If Token is an Operand, append Token to PostFix.
   (b) If Token is either a Left parenthesis or a Left bracket, push Token to the Stack.
   (c) If Token is a Right parenthesis, pop out all items from the Stack, until a left parenthesis is met, which is left back.
   (d) If Token is a Right bracket, pop out all items from the Stack, until a Left bracket is met, which is appended to PostFix.
   (e) If Token is a Comma, pop out all items from the Stack, until either a Left bracket or Comma is met. If a Left Bracket is found it is left back. Token is appended to PostFix.
   (f) If Token is an Operator, perform priority- checking among Token and items of Stack. During priority-checking, either push items back or append them to PostFix, depending on their relative priority to Token.
   (g) If none of the above is satisfied, error: stop.
3. Append any items left in Stack to PostFix.
4. Stop.

Notes

The HybridPostFix form : The HybridPostFix form (note the lettering) is as follows: If the expression does not contain MCL function references or MCL array elements, then it coincides with the standard PostFix form (see [4]). For example: (A+B)-C*2 (InFix) is transformed via PFIX to AB+C2*- (PostFix). If expression contains either arrays or functions, then the following is done: A[1,2]*MAX[4,X] (InFix) becomes A1,2]MAX4,X]* (HybridPostFix), i.e., it is PostFix but the arguments and — or subscripts remain in their order as the expression is being transformed. The commas also remain to separate the argument expressions, while the final bracket remains in order to terminate the sequence. Note that any items enclosed in brackets which are expressions are being transformed to standard PostFix form, while as a whole, in the final form they keep their ordering. Additional processing is needed, when code is to be generated from the HybridPostFix form, so that items enclosed in brackets be processed appropriately. Error recovery though, in case MCLCOM faces a situation of anomalous array or function referencing, is immediate.

6.11 XPR

Calling sequence

```
CALL XPR ( EX, NTOKS, NA, NN, CORR, FDECLR, BLOC, SCRUNT, STVAR )
```

Purpose

This routine orchestrates MCL expression processing. It controls the routines PFIX, XCHECK and XGEN.

Description of arguments

Input arguments

`EX`	The expression to be processed.
`NTOKS`	The number of tokens EX is composed of.
`FDECLR`	Logical flag, indicating if true that XPR is called to process a function declaration.
`SCRUNT`	Integer variable, keeping the unit number of a scratch file on which code is to be written.
`STVAR`	The statement function argument table.

Output arguments

`BLOC`	Number of object code lines written.

Input / Output arguments

`NA`	Integer variable, pointer to the last used element of ALPHAC.
`NN`	Integer variable, pointer to the last used element of NUMBC.
`CORR`	Logical variable, being true if expression is — remains correct.

Other subprograms called

Subroutines `PFIX, XCHECK, XGEN, ERROR`.

Global data accessed (A) or modified (M): MCLSCD(A).

The `XPR` algorithm

1. Start.
2. Insert Zeros Whenever needed to transform signs to binary operators.
3. Insert Ones in front of `NOT` operators so as to transform them into binary operators.
4. Call the `XCHECK` routine to test the validity of the expression.
5. If the expression is correct then
   (a) Call routine `PFIX` to transform expression from Infix to Hybrid Postfix form.
   (b) Call routine `XGEN` to generate the code needed for the expression's evaluation.

   else send error message.
6. Stop.

6.11.1 An outline of how XPR manipulates expressions

1. Original expression: A1 * X[K] + (B[K,J] - C) * Y + Z / 2. (entered to `XPR`)
2. Lexical Analysis : A1 * X [K] + (B[K,J] - C) * Y + Z /2. (`LEX`)

   Identify A1, X, K, B, K, J, C, Y, Z as identifiers, identify 2. as a number identify *, +, -, *, +, / as operators, and finally identify [,], (, and) as special marks. What is meant by identification, is detection of the specific lexical class on which each token belongs.
3. Parsing:
   (a) Structure Checking:
      i. (A1 * X[K] + (B[K,J] - C) * Y + Z / 2) (`XCHECK`)
      ii. (A1 * ident + (ident - C) * Y + Z / 2)
      iii. (A1 * ident + ident * Y + Z / 2)
      iv. ident * ident + ident * ident + ident / number
      v. valid subexpression revealed.

   (b) Infix to Hybrid-postfix transformation (`PFIX`):
      i. A1 * X[K] + (B[K,J] - C) * Y + Z / 2
      ii. (A1 * X[K]) + ((B[K,J] - C) * Y) + (Z / 2)
      iii. ((A1 * X[K]) + ((B[K,J] - C) * Y)) + (Z / 2)
      iv. ((A1X[K] *) + ((B[K,J]C-) * Y)) + (Z2 /)
      v. ((A1XK] *) + ((BK,J]C-)Y*))(Z2 /)+

vi. ((A1XK] *)((BK,J]C-)Y*)+)(Z2 /)+

vii. A1XK] * BK,J]C-Y*+Z2 /+ (the final form revealed).

4. Code generation: Routine `XGEN` is called and the above form is processed (see the `XGEN` Algorithm). Code generated:

```
$PUSH
<location of A1>
$PUSH
<location of K>
$X
$MUL
$PUSH
<location of K>
$PUSH
<location of J>
$GETCONT
$PUSH
<location of C>
$SUB
$PUSH
<location of Y>
$MUL
$ADD
$PUSH
<location of Z>
$PUSHC
 2.
$DIV
$ADD
```

6.11.2 A note on the insertions

The philosophy on which we based processing of the +, - and NOT operators (since they may form Unary-Operator constructs: i.e., use only one argument), comes from the observation that the above operators may appear legally only in the following combinations: (£—¡Bracket¿—¡AnyParenthesis¿) ('+'—'-'—'NOT') (¡RightParenthesis¿—¡Operand¿) and ¡Operand¿ ('+'—'-') (¡RightParenthesis¿—¡Operand¿) From the above, it is evident that insertions must not be performed for certain combinations, like: ¡LeftParenthesis¿ ('+'—'-') ¡RightParenthesis¿ or ¡Operand¿ ('+'—'-') ¡Operand¿ which among other combinations are excluded from the process. The algorithm followed can be summarized as:

While Not eol do scan input line to locate operator. If operator found then check if insertion is needed and perform it. Else continue with no insertions. EndWhile.

6.12 SUBX

Calling sequence

```
CALL SUBX ( SIMPX, TOKN, CORR )
```

Purpose

SubExpression checker: This routine checks expressions consisting only of operands and operators. A SubExpression (or simple expression), corresponds to the expression-part of the innermost pair of parentheses or brackets in a complicated MCL expression.

Description of arguments

Input arguments

`SIMPX` The input simple expression to be checked. One dimensional integer array.

`TOKN` The number of tokens, SIMPX consists of.

Output arguments

`CORR` Output logical variable, being true, if SIMPEX is correct.

Where called from: Subroutine XCHECK.

Other subprograms called

Subroutines `ERROR`.

Global data accessed (A) or modified (M): MCLSCD(A)

6.13 XCHECK

Calling sequence

```
CALL XCHECK ( EXPR, TOKNUM, CORR, NA, FDECLR, STVAR )
```

Purpose

Checks the validity of MCL expressions (see rules 8-16 in section 2.2).

Description of arguments

Input arguments

`EXPR`	The expression to be processed.
`TOKNUM`	The number of tokens EXPR is composed of.
`FDECLR`	Logical flag, indicating if true that XPR is called to process a function declaration.
`STVAR`	The statement function argument table.

Input / Output arguments

`CORR`	Logical variable, being true if expression correct.
`NA`	Pointer to the last used element of ALPHAC.

Where called from: Subroutine XPR.

Other subprograms called

Subroutines `ERROR, INSERT, LOCAN, POPBR, POPUP, PUSHBR, PUSHDN,SUBX, SERLOC.`

Global data accessed (A) or modified (M): TSTXST (M) , TSTBST (M) , MCLCNT (M) , MCLSCD (A).

Notes

This routine checks the validity of any MCL expression using the following method :

- The input expression residing in array `EXPR` in the form of integer codes, is set in between parentheses. By doing this, if the expression is a simple- expression (i.e., an expression consisting only of operators and operands) we ensure that it will have a standard treatment as an arbitrary MCL expression. (Note that the forms ¡MclExpression¿ and '('¡MclExpression¿')' are equivalent, and this applies to the coded form too).
- A scanning is performed on `EXPR` and when a left parenthesis is located, its position is pushed into a stack. If a left bracket is located its position is pushed in a second stack (referred from here on as bstack).
- When a right parenthesis is met, the position of the most recently located parenthesis is popped and the code which corresponds to this position is transformed to identifier-code. If a right bracket is met, the position of the most recently located bracket is popped from bstack .
- All items from (position of left parenthesis + 1) up to and including (position of right parenthesis), are assigned a special ignore-code and scanning is continued from the point before the most recent ignore-code assignment. In the case of a bracket, we proceed as above starting however from (position of left bracket).
- Note 1: Ignore-code assignment is as follows: all items inside the area under processing, are being assigned a special code. This code, when met again in a following iteration, forces the routine to ignore it and continue to get the next item.
- Note 2: An operand is a numeric constant, a variable, an array element, or a function reference. An operator is any valid MCL operator from the set of {+, -, *, /, **, ¡=, >=, <, >, :=:, #, NOT, XOR, AND, OR}.

6.14 LBDEF

Calling sequence

```
CALL LBDEF ( NAME )
```

Purpose

This routine processes a label definition. If the specified label name is already in the label table, this will cause an MCL-error to be issued (see sect. 6.2). Otherwise, the label is inserted in the label table. Only the following one line of object code is generated :

$CONTINUE

The location assigned to the label in the label-location table is the location of the above line in the scratch file.

Description of arguments

Input arguments

`NAME` Character*(*) variable. This is the name of the label to be processed.

Other subprograms called

Subroutines `LOCAN, ERROR, ABORT`.

Abort conditions that may occur

If the label table overflows this will cause the program to stop and issue an informative message.

6.15 LOCAN

Calling sequence

```
CALL LOCAN ( TABLE, ELEM, NAME, FOUND, POSIT )
```

Purpose

This routine locates the position of a given name in a character-type table, sorted in ascending order. The method used, is a binary search [8, 7].

Description of arguments

Input arguments

`TABLE` Character array. Dimensioned as CHARACTER*(*) TABLE(*) This is the table to be searched. It is assumed to be sorted in ascending order.

`ELEM` Integer variable. Number of elements in the table.

`NAME` Character variable. The name whose position in the table we are looking for.

Output arguments

- `FOUND` Logical variable indicating whether the specified name, was (FOUND=.TRUE.), or was not found (FOUND=.FALSE.) in the given table.
- `POSIT` Integer variable. If the name was found in the table, this is its position; otherwise, this is the position that the given name should occupy.

6.16 SERLOC

Calling sequence

```
CALL SERLOC ( TABLE, ELEM, NAME, FOUND, POSIT )
```

Purpose

This routine locates the position of a given name in a character-type table. The table need not to be sorted, since a linear search is applied to it.

Description of arguments

Input arguments

- `TABLE` Character array. Dimensioned as : CHARACTER*(*) TABLE(*) This is the table to be searched for the specified entry.
- `ELEM` Integer variable. The number of elements in the table.
- `NAME` The name whose position in the table we are looking for.

Output arguments

- `FOUND` Logical variable indicating whether the specified name, was (FOUND=.TRUE.), or was not found (FOUND=.FALSE.) in the given table.
- `POSIT` Integer variable. If the name was found in the table, this is its position; otherwise, POSIT=0

6.17 COPY

Calling sequence

```
CALL COPY ( INIT, FINAL )
```

Purpose

Copies a part of the scratch file (generated by the compiler during the first pass over the MCL source) to the final object code file. The part of the file to be copied is specified by the initial and final line counters. These counters have nothing to do with the actual location of the lines in the file. They are used to specify how many lines are to be copied. Lines are copied from the current position of the scratch file until the initial counter + 1 reaches the final counter - 1.

Description of arguments

Input arguments

`INIT` Integer variable. This is the initial line counter. Copying starts at INIT+1.

`FINAL` Integer variable. This is the final line counter. Copying stops at FINAL-1.

Notes

Note that actually, (FINAL-1)-(INIT+1)+1 = FINAL-INIT-1 lines are copied, starting at the current position of the scratch file.

6.18 FEBUFF

Calling sequence

```
CALL FEBUFF ( LINE, TOTAL)
```

Purpose

Flushes the error buffer (see subroutine ERROR for a description of the error buffer) to the error list file. The faulty line, along with its line number will also be printed on the error list file. The routine assumes that upon entry, the error buffer contains at least one entry. The number of entries in the error buffer is counted by variable NERR in the MCLCNT common block.

Description of arguments

Input arguments

- `LINE` Character*(*) variable. This is the MCL source line that is currently under processing by the compiler.
- `TOTAL` Integer variable. This is the total number of errors encountered in the program, so far.

6.19 NOPAR

Calling sequence

```
CALL NOPAR ( NTOK, NT, NAME, OPTION )
```

Purpose

This routine processes statements with no parameters. If OPTION=0 the command name (variable NAME) is written on the scratch file. Otherwise the command is prefixed by a dollar ($) sign, before it is written on the scratch file. For example, for the MCL source line: SHORTDIS this routine will be called with OPTION=0 and will generate: SHORTDIS while for the MCL source line: PAUSE this routine will be called with OPTION0 and will generate: $PAUSE

Description of arguments

Input arguments

- `NTOK` Integer variable. Number of tokens in the current line.
- `NAME` Character*(*) variable. This is the command to be processed.
- `OPTION` Integer variable. This is an option. If OPTION=0, the command already resides in the MERLIN operating system; otherwise the command is implemented for MCL use only.

Input / Output arguments

- `NT` Integer variable. This is the number of tokens used by the previously called parsing routines. Checks on command syntax are performed, starting at token NT+1. Upon exit, this variable points to the token immediately following the command.

Other subprograms called

Subroutines `ERROR`.

6.20 MOVE

Calling sequence

```
CALL MOVE ( NT, NA )
```

Purpose

This routine processes the MOVE TO statement. The following two lines of object code are generated :

$MOVE ¡blank line¿

where ¡blank line¿ is a line left blank. The destination label of the statement is inserted in the move table. A value of -7 is pushed in the TYPE array, while the location of the blank line is also pushed in the TDEF array. The actual argument for the $MOVE instruction will be calculated in pass 2, after all label definitions have been read from the source.

Description of arguments

Input / Output arguments

- `NT` Integer variable. This is the number of tokens used by the previously called parsing routines. It is used as a pointer to the TOKEN array. Checks on command syntax are performed, starting at token NT+1. Upon exit, this variable points to the token where syntax checks have stopped.

- `NA` Integer variable. This is the number of identifiers used by the previously called parsing routines. It is used as a pointer to the ALPHA array. Upon exit this is the number of identifiers used by this routine.

Other subprograms called

Subroutines `ERROR, ABORT`.

Abort conditions that may occur

If the move table overflows this will cause the program to stop and issue an informative message.

6.21 EXITL

Calling sequence

```
CALL EXITL ( NT )
```

Purpose

The EXITL routine processes the EXIT statement. If the statement is outside a loop structure, this will cause an MCL-error to be issued. The following two lines of object code are generated :

$MOVE ¡blank line¿

where ¡blank line¿ is a line left blank. A value of 8 is pushed in the TYPE array, while the location of the blank line is also pushed in the TDEF array. The actual argument for the $MOVE instruction will be calculated later when the END LOOP that terminates the current loop is reached.

Description of arguments

Input / Output arguments

`NT` Integer variable. This is the number of tokens used by the previously called parsing routines. Checks on command syntax are performed, starting at token NT+1. Upon exit, this is the number of identifiers used by this routine.

Other subprograms called

Subroutines `ERROR`.

Notes

Note that the target of the $MOVE instruction generated, is the statement immediately following the ENDLOOP statement of the loop.

6.22 RESE

Calling sequence

```
RESE ( NAME )
```

Purpose

The Logical function RESE checks whether a given name, may be used as a variable or function symbolic name, in a VAR or FUNCTION declaration. The identifiers FROM, TO, BY, JUST and THEN can not be used as symbolic names. The value returned by the function is .FALSE. if there is no problem with the given name; .TRUE. otherwise.

Description of arguments

Input arguments

`NAME` Character*(*) variable. This is the symbolic name to check.

6.23 BLOCK DATA MCLDEF

Calling sequence

```
None
```

Purpose

Initializes some of the common blocks used in the program.

Notes

The following blocks are assigned initial values that will not be changed, throughout programs execution : MCLUNT, MCLSCD, MCLMFL, FILES. The values assigned to the MCLDTA common block are changed, during programs execution.

6.24 IF

Calling sequence

`CALL IF`

Purpose

This routine parses the IF statement. The logical expression following the IF statement is evaluated (this means, that code appropriate to its evaluation is generated) and two more instructions are written in the scratch file:

$NOT $TESTMOVE ¡blank line¿ If the logical condition holds, a non-zero value will be left on the top of the stack. The $NOT instruction will cause it to become 0. Subsequently, the $TESTMOVE instruction will not cause the program to jump to the ENDIF (or ELSE) that follows. (See also 3.3.1) The code generated for a block if structure of the form:

IF ¡logical expression¿ THEN ¡block1¿ ELSE ¡block2¿ END IF

is as:

(code for the evaluation of the ¡logical expression¿) $NOT $TESTMOVE ¡blank line¿ (code for ¡block1¿) $MOVE ¡blank line¿ (code for ¡block2¿) $CONTINUE

The destination of the $TESTMOVE instruction is either the first instruction of the (code for ¡block2¿) or the $CONTINUE instruction, depending whether the ELSE part exists for this block if, or not. The destination of the $MOVE instruction, is always the $CONTINUE instruction. A value of 1 is pushed in the TYPE array, to indicate that an IF has been reached. The value of the location counter in the scratch file, is also pushed in the TDEF array.

Other subprograms called

Subroutines `EXTRA, XPR, ERROR, OVER`.

Abort conditions that may occur

If the type table overflows this will cause the program to stop and issue an informative message.

Notes

The number to be written in the ¡blank line¿, left under the $TESTMOVE instruction will be calculated later, when the ENDIF (or ELSE) corresponding to this IF is reached.

6.25 ELSE

Calling sequence

```
CALL ELSE
```

Purpose

Parses the ELSE statement. Only 2 lines of object code are generated (see also the description of subroutine IF and section 3.3.1) :

$MOVE ¡blank line¿

The type table is searched backwards, (from the current entry, to the beginning of the table) until a value of 1 is found. This indicates the IF corresponding to this ELSE statement. The number of lines to jump (the number that should be written after the $TESTMOVE instruction for this IF) is now calculated and stored in the type-link table as : TREL(I) = BLOC-TDEF(I)+1 where I is the position of the IF corresponding to this ELSE in the type table, and BLOC is the location of the blank line after the $MOVE instruction in the scratch file. Subsequently, this IF is assigned a value of -1 in the type table to indicate that it has been processed. If an IF is not found, while searching the type table backwards, this will cause an MCL error to be issued by the compiler.

Other subprograms called

Subroutines `ERROR, OVER`.

Abort conditions that may occur

If the type table overflows this will cause the program to stop and issue an informative message.

6.26 ENDIF

Calling sequence

```
CALL ENDIF ( NT )
```

Purpose

Processes the ENDIF statement. Only one line of object code is generated (see also 3.3.1) :

$CONTINUE

The type table is searched backwards until an entry of 1 or 2 (indicating the presence of an IF or ELSE statement) is reached. The number of lines to jump so as to reach this ENDIF statement will be calculated and pushed in the appropriate position, in the type-link table. The entry in the type table for the IF or ELSE located will be set to -1 or -2 respectively, indicating that they have been processed. A value of -3 is pushed in the type table, (indicating the presence of the ENDIF statement) while the location of the above $CONTINUE instruction is pushed in the type-definition table, in order to be used later on.

Input / Output arguments

`NT` Integer variable. This is the number of tokens that have been used so far to parse the current source line. This routine will start syntax checks at token NT+1.

Other subprograms called

Subroutines `ERROR, OVER`.

Abort conditions that may occur

If the type table overflows this will cause the program to stop and issue an informative message.

Notes

Note that only two values are possible for NT upon entry: 1 or 2. If the statement is entered as "ENDIF" (one word), this will cause NT to take on the value 1, since only one token (an identifier) is found. However if the statement is entered as "END" "IF" (two distinct words) , this will cause NT to take on the value 2, since two tokens (two identifiers) are found. Both of them are used by the PCG subroutine in order to recognize the statement.

6.27 VARIA

Calling sequence

```
CALL VARIA ( )
```

Purpose

Processes the VAR declaration statement. Reserved names, as well as names already present in the var table, should not be declared. The routine checks for syntactic validity and updates the related tables. These are: The var table, The memory location table, The dimensionality table, The information table, The position table and The kind table.

Description of arguments

Input arguments

`ee`

Input / Output arguments

`ee`

Other subprograms called

Subroutines `LOCAN, CHARR, INSERT, ERROR`.

Functions `RESE, MEMSPC`.

Notes

If a syntactic error is encountered while a simple variable is processed, the routine will continue parsing. If an error is encountered while an array is processed, the routine will return, since it has no means to determine in which point of the input line, the next variable declaration starts.

6.28 WHEN

Calling sequence

`CALL WHEN`

Purpose

Processes the WHEN statement. The object code generated is as:

¡code to evaluate the expression involved in the statement¿ $NOT $TESTMOVE ¡blank line¿

If the syntax up to the JUST keyword is correct, a value of -6 is pushed in the type table, to indicate the WHEN statement. Subroutine EXE is called then to process the statement following the JUST keyword. Upon return of the above routine, the number of object code lines, that it has written, is pushed in the type-link table, in the position corresponding to this WHEN.

Other subprograms called

Subroutines `EXTRA, XPR, ERROR, EXE, OVER`.

Abort conditions that may occur

If the type table overflows this will cause the program to stop and issue an informative message.

6.29 EXTRA

Calling sequence

```
CALL EXTRA ( NT, EX, NEX, IOPT, NA )
```

Purpose

Extracts from a line, all tokens, valid in an MCL expression and stores them in an array that is returned as output. The line is input in the form of tokens. Expression scanning always stops when an invalid token is reached. Additionally according to an input option, expression scanning stops when different situations are met. A token is accepted in an expression sequence, if it's one of the following:

right parenthesis) left parenthesis (multiplication * division / addition + subtraction - identifier unsigned number raise to a power ** comma , right bracket] left bracket [greater than ¿ less than ¡ greater or equal than ¿= less or equal than ¡= exclusive or operator XOR not operator NOT and operator AND or operator OR

Description of arguments

Input arguments

`IOPT` Integer variable. This is an option. It may take on the values: 0 expression scanning stops at the first invalid token reached. 1 expression scanning stops also when one of the identifiers: FROM, TO, BY, JUST or THEN is encountered. 2 expression scanning stops also when a right parenthesis is reached before the end of the line. 3 expression scanning stops also at the equality sign (=) appearing after a right bracket.

`NA` Integer variable. This is a pointer to the ALPHA array, and is the number of identifiers in the current line that have been used by the previously called parsing routines.

Output arguments

`EX` Integer array. Upon return, contains the valid tokens that have been found in the TOKEN array.

`NEX` Integer variable. The number of entries in the EX array.

Input / Output arguments

`NT` Integer variable. When the routine is entered, this is the number of tokens that have been used by the previously called parsing modules. When this routine returns, this is the total number of tokens (including the ones extracted as an expression) used from the current source line.

Other subprograms called

Subroutines `ABORT`.

Functions `RESE`.

6.30 PCG

Calling sequence

```
CALL PCG ( IEND )
```

Purpose

This is the main routine that drives the others according to the input line. This routine is called for executable statements. (i.e. statements that produce object code) At this point all declaration statements have been processed. The first token encountered in the input line, should be an identifier, otherwise an MCL error is issued. If a colon follows, then the routine will assume that this is a label definition line. If an equality sign, or a left bracket follows, the routine will assume, that this is an assignment. If its neither a label, nor an assignment, the routine will test for one of the following statements: WHEN, IF, ELSE, ENDIF, LOOP, ENDLOOP. If these tests fail also, subroutine EXE will be called to test for the rest of the MCL statements.

Description of arguments

Input arguments

`IEND`

Other subprograms called

Subroutines `LBDEF, EXE, WHEN, IF, ELSE, ENDIF, LOOP, ENDLOO`.

Notes

Note that this routine processes directly all MCL statements that may appear next to a WHEN statement. Subroutine EXE will take care of the rest of the statements.

6.31 TERMIN

Calling sequence

`CALL TERMIN`

Purpose

This routine will check for any block ifs, or loops that do not terminate (ENDIF or ENDLOOP absent). An ENDIF statement causes the entries for the corresponding IF and ELSE statement in the type table, to become negative. (-1 or -2) An ENDLOOP statement causes the entry for the corresponding LOOP in the type table to become also negative. (-4) Thus, when the first pass over the MCL source has been completed, a value of 1 or 2, indicates a block if that does not terminate, while a value of 4 indicates a loop that does not terminate. The routine will search the type table backwards and if a value of 1 or 2 is found a "fake" ENDIF is pushed in the type table and the above two values become negative. If a 4 is reached, a "fake" ENDLOOP is pushed in the type table, and the previous value becomes also negative. In both cases the location pushed in the type-definition table, is the current location counter of the scratch file. Since the first pass has been completed this is the last line of object code written. (Note that no more object code will be written in the scratch file) If any non-terminating block ifs or loops are found, appropriate error messages are issued.

Other subprograms called

Subroutines `ERROR, OVER`.

Abort conditions that may occur

If the type table overflows, this will cause the program to stop and issue an informative message.

6.32 VALIJ

Calling sequence

```
CALL VALIJ
```

Purpose

This routine will check the validity of the MOVE TO statements in the program, in relation with the block ifs and loops. A block is defined to be a sequence of statements residing between:

IF ... ENDIF, IF ... ELSE, ELSE ... ENDIF and LOOP ... ENDLOOP statements.

A jump inside a block is not allowed, unless the corresponding MOVE TO statement, belongs to the block. Jumps outside a block, are always allowed. When this routine is entered, it is assumed that block ifs and loops are properly nested and properly terminated. It is also assumed that all MOVE TO statements have been processed. (i.e. linked with the corresponding labels.) Thus only negative entries occur in the type table. The routine will start searching the type table. For each block encountered it will assign a value of 7 for all MOVE TO statements that belong to the block as well as their target label. Then it will check the validity of the rest of the MOVE TO statements, belonging to the block. Upon exit of the routine all entries in the type table that indicate an IF, ELSE, ENDIF, LOOP, ENDLOOP or MOVE TO statement will yield positive values: 1, 2, 3, 4, 5 and 7 respectively.

Other subprograms called

Subroutines `LOCIF, LOCLOO, ZEROM, JUMPIN`.

6.33 ZEROM

Calling sequence

`CALL ZEROM ( IND1, IND2 )`

Purpose

This routine will assign a value of 7 to all MOVE TO entries in the type table, residing in a block specified by two pointers in the table, if their target label lies also in the same block.

Description of arguments

Input arguments

`IND1` Integer variable. This is a pointer to the type table, and is the lower limit from where the search for any MOVE TO statements will start.

`IND2` Integer variable. This is a pointer to the type table, and is the upper limit to where the search for any MOVE TO statements will terminate.

6.34 JUMPIN

Calling sequence

`CALL JUMPIN ( F1, F2, T1, T2 )`

Purpose

This routine checks whether any jumps are performed (via a MOVE TO statement) from a block of statements (specified by two pointers in the type table) to another block of statements. (Specified also by two pointers in the type table.) Upon exit, all MOVE TO statements that have been processed will assume a value of 7. (See also the description of subroutine VALIJ)

Description of arguments

Input arguments

- `F1` Integer variable. This is a pointer to the type table, and is the lower limit from where the search for any MOVE TO statements will start.
- `F2` Integer variable. This is a pointer to the type table, and is the upper limit to where the search for any MOVE TO statements will terminate.
- `T1` Integer variable. This is a pointer to the type table, and is the lower limit of the block that shouldn't be the target of a MOVE TO statement residing in the first block.
- `T2` Integer variable. This is a pointer to the type table, and is the upper limit of the block that shouldn't be the target of a MOVE TO statement residing in the first block.

6.35 LOCIF

Calling sequence

```
CALL LOCIF ( IF, ELSE, END )
```

Purpose

This routine will locate an IF and an ELSE entry in the type table, given the position of the ENDIF entry. It is used while checking the validity of the MOVE TO statements and assumes that entries in the type table that have not been processed are negative; positive otherwise.

Description of arguments

Input arguments

- `END` Integer variable. This is the position of the ENDIF in the type table.

Output arguments

- `IF` Integer variable. This is the position of the IF in the type table, corresponding to the given ENDIF.

ELSE Integer variable. This is the position of the ELSE in the type table, corresponding to the given ENDIF. A value of 0 is returned if this is an IF without an ELSE.

Other subprograms called

Subroutines `ABORT`.

6.36 LOCLOO

Calling sequence

```
CALL LOCLOO ( LOO, ENDLOO )
```

Purpose

This routine will locate a LOOP entry in the type table, given the position of the ENDLOOP entry. It is used while checking the validity of the MOVE TO statements and assumes that entries in the type table that have not been processed are negative; positive otherwise.

Description of arguments

Input arguments

`ENDLOO` Integer variable. This is the position of the ENDLOOP in the type table.

Output arguments

`LOO` Integer variable. This is the position of the LOOP in the type table, corresponding to the given ENDLOOP.

Other subprograms called

Subroutines `ABORT`.

6.37 ERROR

Calling sequence

```
CALL ERROR ( CODE, MESS )
```

Purpose

Handles error processing. When a syntactic error is detected during the parsing of an MCL source line, this routine is called to issue the appropriate message. According to the input code, a message will be placed in the error buffer. Another message (a short one, up to 10 characters), will also be placed in the error buffer. If the actual argument in the subroutine call is longer than 10 characters, it will be truncated, and 3 dots will be appended to it. An option is available that controls display of the error code to the output file. This may be useful for debugging purposes, or for further development. When this routine is called, further code generation will be disabled.

Description of arguments

Input arguments

- `CODE` Integer variable. This is the code number of the error to be issued.
- `MESS` Character variable. This a short (up to 10 characters) description of the problem, to be displayed along with the full error message description.

Other subprograms called

Subroutines `ABORT`.

Abort conditions that may occur

If the error buffer is full, the program will stop and issue an informative message. In this case set the MAXERR parameter to a higher value, or correct your program.

Notes

The main structure used in this subroutine (and subroutine FEBUFF also) is the error buffer. This is an one dimensional character*90 array named EBUFF. The array is dimensioned by the MAXERR parameter. The number of entries in the buffer is counted by the NERR variable in the MCLCNT common block. Each entry in this array is an one line error message to be written on the error list file. As syntax errors are detected (for the same source line) by the compiler, they are placed in the error buffer and NERR is incremented. When parsing of the input line has been completed the error buffer is flushed (if it contains any messages) on the error list file. An option is available that controls display of the error code. The logical variable CDIS when set to .TRUE. (via a DATA statement in the routine) will allow display of the error code.

6.38 EXE

Calling sequence

```
CALL EXE ( NT, NA, NN )
```

Purpose

This is the main routine that drives other program units according to the statement to be processed. It handles any statement that may appear next to a WHEN statement. Upon entry it assumes that the first token encountered is an identifier. (A command or the name of a variable, in case of an assignment.) Two arrays are defined in this routine: 1) A character array (named STAT) that keeps the statements this routine handles, and 2) an integer array (named NAME) that keeps an identification code for each statement, in case a routine takes care of more than one statement. EXE will first check if the next token in the sequence is an = or a [. In such a case it will assume that an assignment takes place. Otherwise it will check whether the first identifier upon entry is one of the valid commands (defined in the STAT array), and if so it will perform a computed GOTO and call the appropriate routine to process the statement.

Object code is generated by this routine only in case of an assignment as:

¡code to calculate memory address of the element to be assigned the value¿ ¡code to calculate the expression on the right side of the = sign¿ $POPCONT

Description of arguments

Input / Output arguments

- `NT` Integer variable. This is a pointer to the TOKEN array and is the number of tokens used so far by the previously called parsing routines.
- `NA` Integer variable. This is a pointer to the ALPHA array and is the number of identifiers used so far by the previously called parsing routines.
- `NN` Integer variable.This is a pointer to the NUM array and is the number of numbers used so far by the previously called parsing routines.

Other subprograms called

Subroutines `LOCAN, ERROR, INSERT, EXTRA, XPR, ABORT, NOPAR, GRADQ, REWI, DISCA, DFL, PICK, STEPO, GODFA, MULTI, EXECU, MOVE, DISPLA, GET, ACCUM, HIDE, INI, MARGI, EXITL.`

Notes

If an array is being assigned a value, the whole left side of the = sign is considered as an expression and subroutine XPR is called to process it. XPR will generate code to calculate the array element address and push it on the top of the stack. Then a $GETCONT instruction will be generated by the same routine, so as when executed, to replace the top of the Merlin stack with the actual value of the array element. The object code just described will be written in the spare file. As soon as XPR returns, all of the spare file will be copied to the scratch file, excluding the final $GETCONT instruction.

6.39 GET

Calling sequence

```
CALL GET ( NT, NA, NN )
```

Purpose

This routine parses the GET statement and generates appropriate object code. The syntax expected is:

GET var1;var2;...varn or GET var1;var2;...varn FROM ¡filename¿

where var1,var2,...varn is a list of variables and ¡filename¿ is a filename. (An identifier.) The routine will first determine the number of variables to be input by counting the semicolons in the input line. Then it will determine the input ¡filename¿. If this is omitted the default Merlin input file will be used. ¡filename¿ is not allowed to be any of the Merlin files: HELP, DATA, STORE, INIPO, DISPO, BACKUP, MACROF, defined in the block data subprogram MCLDEF, in the common block MCLMFL. Only simple variables (not used as loop control variables) or array elements (declared by the user) may appear in the list of variables. If any of the variables is not declared, or is not of the type mentioned above, an Mcl error will be issued. The object code generated is as:

$IN ¡number of variables¿ ¡filename¿ ¡code to calculate memory address for var1¿ $POPLV ¡code to calculate memory address for var2¿ $POPLV ... ¡code to calculate memory address for varn¿ $POPLV

where ¡number of variables¿ is the number of variables in the variable list and ¡filename¿ is the file name from which the values will be input. (".STANDARD" will be used if ¡filename¿ is omitted from the GET statement.) ¡code to calculate memory address for varn¿ is either just an integer number pushed on the stack (by a $PUSHC instruction) in case of a simple variable, or a sequence of code that will calculate the requested address in case of an array element.

Description of arguments

Input / Output arguments

`NT` Integer variable. This is a pointer to the TOKEN array and is the number of tokens used so far by the previously called parsing routines.

`NA` Integer variable. This is a pointer to the ALPHA array and is the number of identifiers used so far by the previously called parsing routines.

`NN` Integer variable. This is a pointer to the NUM array and is the number of numbers used so far by the previously called parsing routines.

Other subprograms called

Subroutines `ERROR, LOCAN, INSERT, EXTRA, XPR, ABORT.`

Abort conditions that may occur

This routine uses the spare file, so when its opening or closing fails, the program will stop and issue an informative message.

Notes

If an array element appears in the list of variables, it is considered as an expression and subroutine XPR is called to process it. XPR will generate code to calculate the array element address and push it on the top of the stack. Then a $GETCONT instruction will be generated by the same routine, so as when executed, to replace the top of the MERLIN stack with the actual value of the array element. The object code just described will be written in the spare file. As soon as XPR returns, all of the spare file will be copied to the scratch file, excluding the final $GETCONT instruction.

6.40 DISPLA

Calling sequence

```
CALL DISPLA ( NT, NA, NN )
```

Purpose

This routine handles the DISPLAY statement. The syntax expected is:

DISPLAY item1;item2;...itemn or DISPLAY item1;item2;...itemn TO ¡filename¿ where ¡filename¿ is the file to receive the output, and item1;item2;...itemn is a list of items. Each item may be:

1) A valid MCL expression, or 2) a sequence of characters, (a string) delimited by single quotes.

The routine will first determine the input ¡filename¿. If its omitted, the default MERLIN input file will be used. ¡filename¿ is not allowed to be any of the MERLIN files: HELP, DATA, STORE, INIPO, DISPO, BACKUP, MACROF, defined in the block data MCLDEF, in the common block MCLMFL. Then it will determine the number of variables to get by counting the semicolons in the input line. A loop over all items in the item list will start.

For an item being an expression the code generated is:

¡code to evaluate the expression¿ $OUT ¡filename¿

where ¡code to evaluate the expression¿ is the code necessary to evaluate the expression. (Obviously more than one line.) For a string the code generated is:

$NOTE ¡length¿ ¡string¿ ¡filename¿

where ¡length¿ is the length of the string, in characters, ¡string¿ is the string of characters to be displayed, and ¡filename¿ is the file to receive the output. (".STANDARD" will be used if the ¡filename¿ is omitted from the DISPLAY statement.) When all the items in the list have been processed, the following two more lines of object code are written on the scratch file:

$FLUSH ¡filename¿

Description of arguments

Input / Output arguments

`NT` Integer variable. This is a pointer to the TOKEN array and is the number of tokens used so far by the previously called parsing routines.

`NA` Integer variable. This is a pointer to the ALPHA array and is the number of identifiers used so far by the previously called parsing routines.

NN Integer variable. This is a pointer to the NUM array and is the number of numbers used so far by the previously called parsing routines.

Other subprograms called

Subroutines `EXTRA, XPR, ERROR`.

6.41 CHARR

Calling sequence

```
CALL CHARR ( NT, NDIM, ERR, ARR, NN )
```

Purpose

This routine will check if an array is defined in a VAR statement. In this case, it will parse the definition and update the information table. The part of the input line that is actually parsed is:

[L1:U1, L2:U2, ... Ln:Un]

with L1,L2,...Ln being the low dimension bounds, and U1,U2,...Un being the upper dimension bounds of the array respectively.

Description of arguments

Input arguments

XXX

Output arguments

NDIM Integer variable. This is the number of dimensions for the declared array. 0 will be returned if this is not an array definition.

ERR Logical variable. Indicates whether a syntax error occurred (ERR=.TRUE.) in the declaration, or not. (ERR=.FALSE.)

ARR Logical variable. Indicates whether the item in the VAR declaration is an array (ARR=.TRUE.) , or not. (ARR=.FALSE.)

Input / Output arguments

`NT` Integer variable. This is a pointer to the TOKEN array and is the number of tokens used so far by the previously called parsing routines.

`NN` Integer variable. This is a pointer to the NUM array and is the number of numbers used so far by the previously called parsing routines.

Other subprograms called

Subroutines `ERROR`.

6.42 MEMSPC

Calling sequence

```
MEMSPC ( INFO, IP, NDIM )
```

Purpose

Calculates total memory space occupied by an MCL array. This is calculated as a product of the storage required for each one of the array's dimensions.

Description of arguments

Input arguments

`INFO` Integer array. This is the information table.

`IP` Integer variable. Information for the array starts at position IP of the information table.

`NDIM` Integer variable. The dimensionality of the array.

6.43 INSERT

Calling sequence

```
CALL INSERT ( ID, ND, ML, IP, KI, IND )
```

Purpose

Inserts information in the tables related to variables, arrays and statement functions. The tables affected are: 1) the var table 2) the dimensionality table 3) the memory location table 4) the position table and 5) the kind table. The position where information should be inserted is input to the routine.

Description of arguments

Input arguments

- `ID` Character variable. This value is inserted in the var table.
- `ND` Integer variable. This value is inserted in the dimensionality table.
- `ML` Integer variable. This value is inserted in the memory location table.
- `IP` Integer variable. This value is inserted in the position table.
- `KI` Integer variable. This value is inserted in the kind table.
- `IND` Integer variable. This is the position where the input values should be inserted in the tables stated above.

Other subprograms called

Subroutines `ABORT`.

Abort conditions that may occur

In case the tables stated, overflow, the program will stop and issue an informative message.

6.44 STAFU

Calling sequence

`CALL STAFU`

Purpose

This routine will handle a statement function declaration. The declaration is expected in the form:

FUNCTION ¡name¿ [arg1,arg2,...argn] = ¡expression¿

where ¡name¿ is the symbolic name of the statement function, arg1,arg2,...argn is a list of dummy arguments to the function and ¡expression¿ is the actual expression that this function will calculate. The arguments in the list are pushed in the statement function argument table, in the order they appear in the list. The object code generated by the ¡expression¿ is not actually written in the scratch file, but in the statement function file instead.

Other subprograms called

Subroutines	`ERROR, LOCAN, SERLOC, EXTRA, XPR, INSERT.`
Functions	`RESE.`

6.45 MULTI

Calling sequence

```
CALL MULTI ( NT, NA, NN, NAME )
```

Purpose

This routine handles statements with many parameters, of the form:

¡command¿ (key1=expr1; key2=expr2; ... keyn=exprn) or ¡command¿

where ¡command¿ is the command name, key1,key2,...keyn are keywords corresponding to MERLIN-panel parameters to be changed and expr1,expr2,...exprn are expressions. The values they yield, are assigned to the corresponding MERLIN parameters. ¡command¿ may be one of the following:

Description of arguments

Input arguments

`NAME` Character variable. This is the command to be processed.

Input / Output arguments

- `NT` Integer variable. This is a pointer to the TOKEN array and is the number of tokens used so far by the previously called parsing routines.
- `NA` Integer variable. This is a pointer to the ALPHA array and is the number of identifiers used so far by the previously called parsing routines.
- `NN` Integer variable. This is a pointer to the NUM array and is the number of numbers used so far by the previously called parsing routines.

Other subprograms called

Subroutines `LOCAN, ERROR, EXTRA, XPR, ABORT`.

6.46 MARGI

Calling sequence

```
CALL MARGI ( NT, NA, NN )
```

Purpose

Parses the MARGIN statement and generates appropriate object code. The syntax expected is:

MARGIN (¡k1¿.expr1=val1; ¡k2¿.expr2=val2;...¡kn¿=exprn=valn)

where ¡k1¿,¡k2¿,...¡kn¿ are keywords. Their value is either L (for a left margin) or R. (for a right margin) expr1, expr2,... exprn are expressions. val1, val2,... valn are also expressions. Their values will be assigned to the margin variables (array L or R) indexed by expr1,expr2,...exprn correspondingly. The object code generated is as:

¡code to evaluate expr for the 1-st R keyword¿ ¡code to evaluate val for the 1-st R keyword¿ ¡code to evaluate expr for the 2-nd R keyword¿ ¡code to evaluate val for the 2-nd R keyword¿ ... ¡code to evaluate expr for the n-th R keyword¿ ¡code to evaluate val for the n-th R keyword¿ ¡code to evaluate expr for the 1-st L keyword¿ ¡code to evaluate val for the 1-st L keyword¿ ¡code to evaluate expr for the 2-nd L keyword¿ ¡code to evaluate val for the 2-nd L keyword¿ ... ¡code to evaluate expr for the n-th L keyword¿ ¡code to evaluate val for the n-th L keyword¿ $WMARG ¡nl¿ ¡nr¿ $SWAP

where ¡code to evaluate expr for the n-th R keyword¿ is the object code necessary to evaluate expr for the n-th occurrence of an R keyword in the statement. ¡nl¿ is the total number of "¡k¿.expr=val" entries in the statement using the L keyword and ¡nr¿ is the corresponding number for the R keyword. Since the L and R keywords may be appear in any order in the statement, but the object code for each one must be separate, as shown above, the following method is used: When an R keyword is encountered in the statement, the code that evaluates expr and val for this keyword is written normally in the scratch file. However when an L keyword is reached, the object code that evaluates expr and val is written to the spare file. When all keywords have been processed and no error has been encountered, the spare file is copied on the scratch file.

Description of arguments

Input arguments

`TOKEN` Integer array. These are the tokens in the current line.

`NTOK` Integer variable. This is the number of tokens in the current line.

`ALPHA` Character array. The identifiers encountered in the input line.

`NUM` Real array. The numbers encountered in the input line.

Input / Output arguments

`NT` Integer variable. This is a pointer to the TOKEN array and is the number of tokens used so far by the previously called parsing routines.

`NA` Integer variable. This is a pointer to the ALPHA array and is the number of identifiers used so far by the previously called parsing routines.

`NN` Integer variable. This is a pointer to the NUM array and is the number of numbers used so far by the previously called parsing routines.

`VAR` Character array. This is the var table.

`LOC` Integer array. This is the memory location table.

`DIM` Integer array. This is the dimensionality table.

`INFO` Integer array. This is the information table.

`IPOS` Integer array. This is the position table.

`KIND` Integer array. This is the kind table.

Other subprograms called

Subroutines `ERROR, EXTRA, XPR, ABORT`.

Abort conditions that may occur

In case an error occurs while opening or closing the spare file, the routine will stop and issue an informative message.

6.47 LOOP

Calling sequence

```
CALL LOOP
```

Purpose

Parses the LOOP statement and generates appropriate object code. The syntax expected is:

LOOP ¡var¿ FROM init TO final BY step or LOOP ¡var¿ FROM init TO final

where ¡var¿ is a simple variable (referred to as the loop control index) and init, final, step are expressions specifying the initial, final and step values for the loop. For each loop 5 consecutive storage locations are needed. Their contents are listed along with the symbolic name, they will be be referred later in the description of this subroutine:

1) Storage for the value of init. (init_mloc) 2) Storage for the value of final. (final_mloc) 3) Storage for the value of step. (step_mloc) 4) Storage for the current trip count of the loop. (ctc_mloc) 5) Storage for the final trip count of the loop. (ftc_mloc)

Since these memory locations are used only while the loop is executing, the storage required for all loops in the program is: 5*MAXL , where MAXL is the maximum number of nested loops encountered in the source.

Initially the init, final and step expressions are calculated:

¡code to evaluate init¿ $POP init_mloc ¡code to evaluate final¿ $POP final_mloc ¡code to evaluate step¿ $POP step_mloc

Then the final trip count is calculated by the formula: ftc = max(int ((final-init+step)/step) , 0)

$PUSH final_mloc $PUSH init_mloc $SUB $PUSH step_mloc $ADD $PUSH step_mloc $DIV $TRUNC $PUSHC 0 $MAX 2 $POP ftc_mloc

¡var¿ is then initialized to the value of init:

$PUSH init_mloc $POP var_mloc

where var_mloc is the memory location assigned to ¡var¿. The trip counter is then initialized to 1:

$PUSHC 1 $POP ctc_mloc

A check will be made if the trip counter has reached the final trip count:

$PUSH ctc_mloc $PUSH ftc_mloc $¿ $TESTMOVE ¡blank line¿

The target of the $TESTMOVE instruction is the $CONTINUE instruction, generated by the ENDLOOP statement, to be found later on. A value of 4 is pushed in the type table, indicating that a LOOP statement has been reached. The location of the above ¡blank line¿ is also pushed in the type-definition table.

Other subprograms called

Subroutines `ERROR, OVER, LOCAN, INSERT, EXTRA, XPR`.

Abort conditions that may occur

If either the type or the loop table overflows, the program will stop and issue an informative message.

6.48 ENDLOO

Calling sequence

```
CALL ENDLOO ( NT )
```

Purpose

Parses the ENDLOOP statement and generates appropriate object code. The type table is searched backwards until an entry of 4 (indicating the corresponding loop) is reached. At this point, the type-link table entry for the loop is calculated. Meanwhile for every entry of 8 in the type table (indicating an EXIT statement) the corresponding type-link table entry is calculated. 17 lines of object code are generated. Initially the loop control index is incremented by the step value:

$PUSH var_mloc $PUSH step_mloc $ADD $POP var_mloc

where var_mloc is the memory location assigned to the loop control index and step_mloc is the memory location used to store the step for this loop. The trip counter is then incremented by 1:

$PUSHC 1 $PUSH ctc_mloc $ADD $POP ctc_mloc where ctc_mloc is the memory location used to store the trip counter. Finally the loop is repeated once more:

$MOVE ¡blank line¿ $CONTINUE

a value of -5 is pushed in the type table, indicating the ENDLOOP, while the location of the above ¡blank line¿ in the scratch file, is pushed in the type-definition table. The entry in the type-link table for this ENDLOOP is calculated.

Description of arguments

Input / Output arguments

`NT` Integer variable. This is the number of tokens in the input line, used so far by the previously called parsing routines.

Other subprograms called

Subroutines `ERROR, OVER`.

Abort conditions that may occur

If the type table overflows, the program will stop and issue an informative message.

6.49 OVER

Calling sequence

```
CALL OVER ( TABLE, ARRAY, SUB, MAXIM, PARAM )
```

Purpose

When any of the MCLCOM tables overflows, this routine will be called to issue informative messages and stop execution of the program. See 3.2 for a description of the message issued.

Description of arguments

Input arguments

`TABLE` Character variable. This the name of the table that overflowed.

`ARRAY` Character variable. This is the name of the array implementing the table.

`SUB` Character variable. This is the name of the routine in which the overflow occurred.

`MAXIM` Integer variable. This is the maximum number of entries allowed in the table.

`PARAM` Character variable. This is the name of the parameter that dimensions the table.

6.50 ABORT

Calling sequence

```
CALL ABORT ( MESS, SUB )
```

Purpose

This routine will be called to issue informative messages and stop the program when a situation that normally should never happen, is encountered. The message issued is as:

ROUTINE ¡routine¿ ABORTED. ¡mess¿

where ¡routine¿ is the routine name (variable SUB) where the abnormal situation happened and ¡mess¿ is an informative message (variable MESS) that explains the situation.

Description of arguments

Input arguments

`MESS` Character variable. This is an informative message to be displayed.

`SUB` Character variable. This is the routine name in which the abnormal situation happened.

6.51 GETPAR

Calling sequence

```
CALL GETPAR
```

Purpose

Displays the compilers initial message (see section 5) and prompts the user to enter the compilation parameters. These are entered in one line, up to 80 characters long. The input line must have the form:

k1=v1,k2=v2,...,kn=vn

where k1,k2,...,kn are keywords and v1,v2,...,vn is the value assigned to them. The keywords, their significance and the values that should be assigned to them are as:

Keyword Significance Value allowed I The MCL source Any file name E The error list file Any file name B The object code file Any file name DEBUG The debug option TRUE, FALSE, T, F BOUNDS The bounds option TRUE, FALSE, T, F

The file names specified as input to the compiler may contain special characters (such as . / + - * etc) and may be up to 20 characters at most. This is particularly useful, since most operating systems allow special characters in a file name. Note that the comma may not be used in a file name since it serves separation purposes in the input line. Note also that MCL itself does not support special characters in file names. The input file (the I parameter) must be always specified. The other parameters when not explicitly declared assume default values as:

Parameter Default value E ERRORS B MOC DEBUG FALSE BOUNDS FALSE

The above parameters obtain their default values in the block data MCLDEF. Blanks are allowed in the input line at any point. Duplicate entries of keywords are not allowed. Thus the following line is incorrect:

I=INP, B=BIN, E=ERR, B=BIN

If a file name is specified as a value to a keyword, it may not be specified as a value to another keyword. For this reason the following line is incorrect:

I=FILE1, E=XFILE, BOUNDS=T, B=XFILE

If any of the above rules is violated, or the input line is incorrectly formatted, the program will stop and issue an informative message. Upon return of this routine, the values of IFILE, EFILE and BFILE in the FILES common block will be updated. The values of BOUNDS and DEBUG in the MCLDTA common block will also be updated.

Other subprograms called

Subroutines `LENGTH, DEF.`

6.52 DEF

Calling sequence

```
DEF ( V, VALUE, FILE, NF )
```

Purpose

This function will test if a given argument (Variable V) is used as a file name in a previously encountered keyword. In such a case, a value of .TRUE. will be returned; .FALSE. otherwise.

Description of arguments

Input arguments

`V` Character variable. The function will check if this file name has been entered as a value to a previously encountered keyword.

`VALUE` Character array. Contains the values of the keywords entered in the input line.

`FILE` Integer array. Each entry of this array is a pointer to the VALUE array. It points to the positions of the VALUE array, that have been used as file names in the I,E or B parameters.

`NF` Integer variable. Number of entries in the FILE array. This is actually the number of keywords that used a file name as a value, so far.

6.53 MCL

Calling sequence

```
CALL MCL
```

Purpose

This is the main routine that will read the input file, analyze each source line in a sequence of tokens, and then call the appropriate routines to perform syntactic checks and code generation. Initially it will read all source line containing a VAR declaration statement. When all of them have been processed, it will read all source lines containing a FUNCTION declaration statement. Finally it will read the rest of the program. When processing of each line completes, subroutine

FEBUFF will be called to write the error messages (if any) on the error list file. When all of the program has been read and processed the labels and the MOVE TO statements of the program will be linked. This means that the number of object code lines to jump so as to reach the destination label, will be calculated for each MOVE TO statement in the source. (This number is kept in the type-link table.) Subroutine TERMIN is called then to check for any block ifs or loops that do not terminate, while subroutine VALIJ will check the validity of the MOVE TO statements in the program. Finally the scratch file is copied on the object code file, while the numbers in the type-link table are written at the same time in the appropriate positions.

Other subprograms called

Subroutines `LEX, VARIA, FEBUFF, STAFU, LOCAN, ERROR, OVER, TERMIN, VALIJ, COPY`.

6.54 MCLCOM

Calling sequence

`None`

Purpose

The compilers main program. It will call subroutine GETPAR to obtain the compilation parameters, and then `SUBROUTINE MCL` to compile the source. Upon compilation completion it will delete the error list file if no errors occurred; the object code file will be deleted otherwise.

Other subprograms called

Subroutines `GETPAR, MCL`.

Chapter 7

MCLCOM **input**

Once the compiler is activated the following message is displayed:

——————————————————————————- ————— M C L C O M 1.0 C.S. CHASSAPIS, D.G. PAPAGEORGIOU and I.E. LAGARIS. PHYSICS DEPT., UNIVERSITY OF IOANNINA, IOANNINA - GREECE. ——————————————————————————- ————— ENTER COMPILATION PARAMETERS - I, B, E, DEBUG, BOUNDS -

A line then is expected to be input by the user, describing the files and options to be used. For a description of the input line, see section 4.60 (Subroutine GETPAR) Two options may be specified to ease debugging of an Mcl program:

The BOUNDS option, which if selected will cause the compiler to generate appropriate object code, to check if array subscripts are within the limits specified in the corresponding VAR declaration statement.

The DEBUG option, which if selected will cause the compiler to generate 3 more object code lines for each executable source statement as:

$LINE < n >$¿¿ ¡line¿

where ¡line¿ is the image of the source line that will be compiled and ¡n¿ is its serial number in the input file. Thus when any execution time error is detected by Merlin-2.0, the line number in which it happened will be reported to the user.

Chapter 8

MCLCOM **output**

In this section we discuss the compilers output. MCLCOM outputs either MERLIN Object Code or, in case that errors have been encountered, an error list.

8.1 About the MERLIN Object Code

MCLCOM translates a program written in MCL, into a vertical sequence of the low-level directives of mocd. These are described in the table at the end of this section. There are two kinds of MOCinstructions. Those which use arguments and those which do not. Arguments either rest in the same file with the MOC, or they are generated dynamically at run time at the top of a stack (described below). One, by reading the table with the MOCdescription should be able to read or write MOC. We hope that this information will enable the MCL programmer to understand what MCLCOM's output is all about. This information will be most valuable to anyone technically involved with MCLCOM , since understanding the instructions, will be very useful for further development of the compiler. We must point out here that the add-on MERLIN interface uses a stack mechanism. All arithmetic and other operations are performed through the use of MERLIN's stack and MERLIN's memory (array CMEM into MERLIN-2.0). Note that there is an easy way to see, the sequence of MOC instructions in which any MCL line is transformed to. Simply turn ON the DEBUG option, and observe the MOC file. An image of each MCL line appears (prefixed by $¿¿) along with the corresponding MOC (ignore though the `$LINE` instruction which also appears if `DEBUG` is turned ON, and which has no connection with the particular MCL line). STACK and CMEM (often referenced as MEMORY) are two REAL-type one dimensional arrays used by the interface package of MERLIN-2.0 to facilitate various run time operations. For array STACK, there exists a pointer (ISTACK), that points to its topmost element (in the beginning of each run, this pointer is set to zero). Two kinds of manipulations are allowed on array STACK.

- Operation `POP`. By this we mean a reference to the top of the stack, followed by a decrementation of its pointer by one.

- Operation PUSH. By this we mean an incrementation of the pointer by one followed by an assignment for the new top of the stack.

The MEMORY array, is used to store the values of all variables, arrays and function arguments of an MCL program. The correspondense of a symbolic name to the memory location is established indirectly through the various compiler tables. In the following table, we present all MOC instructions. The appearance of the symbol / signifies new line (in no other case a new line is implied). For example $PUSH / ¡number¿ appears in the file with MOC as: $PUSH and at next line some integer number. Note that ¡number¿ stands for any number, MOS for MERLIN Operating System, INFILE for the MERLIN input file, and ISTACK stands for the pointer to the top of the stack. We occasionally use FORTRAN like statements to achieve precise and economical description.

8.2 About the Error Listing

If any syntax errors occur during compilation, code generation stops, and a listing with the error description, line locations and line images is generated. The default file name for the error list file is: ERRORS. The following table contains the error codes together with their meaning and the name of the issuing routine.

CODE EXPLANATION ISSUED BY ROUTINE

1 LABEL ALREADY DEFINED. LBDEF 2 LABEL DEFINITION NOT FOLLOWED BY "EOL". LBDEF 3 IDENTIFIER EXPECTED IN THE LIST OF VARIABLES. VARIA

GET 4 VARIABLE ALREADY DEFINED. VARIA 5 SEMICOLON EXPECTED IN "VAR" DECLARATION. VARIA 6 "JUST" EXPECTED. WHEN 7 STATEMENT NOT RECOGNIZED. PCG 8 "THEN" EXPECTED. IF 9 "EOL" EXPECTED AFTER "ELSE". ELSE 10 "EOL" EXPECTED AFTER "ENDIF". ENDIF 11 LABEL NOT DEFINED. MCL 12 RESERVED NAME USED IN DECLARATION STATEMENT. VARIA

STAFU 13 THIS IS NOT AN MCL STATEMENT. EXE 14 STATEMENT NOT FOLLOWED BY "EOL". NOPAR 15 "MOVE TO" STATEMENT NOT FOLLOWED BY "EOL". MOVE 16 LABEL NAME EXPECTED IN "MOVE TO" STATEMENT. MOVE 17 "TO" EXPECTED AFTER "MOVE". EXE 18 RESERVED VARIABLES MAY NOT BE ASSIGNED A VALUE. EXE

GET 19 VARIABLE NOT DECLARED. EXE

GET 20 "EOL" EXPECTED AFTER THE ASSIGNMENT. EXE 21 INITIAL DOT NOT FOUND. EXECU 22 IDENTIFIER EXPECTED AS MACRO NAME. EXECU 23 STATEMENT NOT FOLLOWED BY "EOL". EXECU 24 MACRO FILE COULD NOT BE OPENED. EXECU 25 INITIAL MACRO "*" NOT FOUND IN THE MACRO FILE. EXECU 26 MACRO NOT FOUND IN THE MACRO FILE. EXECU 27 THE MACRO SPECIFIED IS EMPTY. EXECU 28 FINAL ""

NOT FOUND. DISPLA 29 SEMICOLON EXPECTED. DISPLA 30 STRING EXPECTED AFTER "". DISPLA 31 LEFT PARENTHESIS SHOULD FOLLOW THE STATEMENT. STEPO 32 KEYWORD EXPECTED. STEPO

MARGI 33 KEYWORD NOT VALID FOR THIS COMMAND. STEPO

MARGI 34 "." EXPECTED AFTER KEYWORD. STEPO

MARGI 35 "=" NOT FOUND. STEPO

MARGI 36 RIGHT PARENTHESIS EXPECTED. STEPO

MARGI 37 LEFT PARENTHESIS EXPECTED. ACCUM 38 ¡KEYWORD¿ "=" EXPECTED. ACCUM 39 KEYWORD ILLEGAL IN THE ACCUM STATEMENT. ACCUM 40 RIGHT PARENTHESIS EXPECTED. ACCUM 41 KEYWORD MISSING. ACCUM 42 "VAR" / "FUNCTION" STATEMENT MISPLACED. PCG 43 OPERAND MISSING, OR OPERATOR TROUBLES IN EXPRESSION. SUBX 44 MISSING OPERAND, OR MISSING OPERATOR IN EXPRESSION. SUBX 45 EXPRESSION CONTAINS ILLEGAL SEQUENCE. SUBX 46 EMPTY EXPRESSION INTO PARENTHESES. XCHECK 47 VARIABLE NOT DECLARED. XCHECK 48 UNMATCHED ")" OR "]" OR COMMA IN WRONG PLACE. XCHECK 49 UNMATCHED LEFT PARENTHESIS OR LEFT BRACKET. XCHECK 50 IDENTIFIER NOT FOUND BEFORE "[". XCHECK 51 LEFT PARENTHESIS NOT FOUND AFTER PARAMETRIC STATEMENT NAME. MULTI 52 KEYWORD EXPECTED AFTER LEFT PARENTHESIS."EOL" FOUND INSTEAD. MULTI 53 KEYWORD ILLEGAL FOR THIS STATEMENT. MULTI 54 RIGHT PARENTHESIS EXPECTED. MULTI 55 LOW ARRAY BOUND NOT NUMERICAL. CHARR 56 COLON EXPECTED AFTER LOW ARRAY BOUND. CHARR 57 ARRAY, HIGH BOUND NOT NUMERICAL. CHARR 58 RIGHT BRACKET NOT FOUND IN ARRAY DECLARATION. CHARR 59 LOW BOUND GREATER THAN HIGH BOUND. CHARR 60 IDENTIFIER EXPECTED AS FUNCTION NAME. STAFU 61 LEFT BRACKET MISSING. STAFU 62 ARGUMENT IN ERROR. STAFU 63 RIGHT BRACKET MISSING IN ARRAY DECLARATION. STAFU 64 "=" EXPECTED. STAFU 65 FUNCTION NAME ALREADY DECLARED. STAFU 66 FUNCTION ARGUMENT DUPLICATED. STAFU 67 "TO" NOT FOUND. DISPLA 68 FILE NAME EXPECTED AFTER "TO". DISPLA 69 "EOL" NOT FOUND AFTER FILE NAME. DISPLA 70 LEFT PARENTHESIS NOT FOUND. MARGI 71 "=" EXPECTED AFTER KEYWORD. GRADQ 72 LEFT PARENTHESIS EXPECTED. GRADQ 73 "FLAG" MISSING. GRADQ 74 KEYWORD INTO ILLEGAL COMBINATION. GRADQ 75 STATEMENT SYNTACTICALLY INCOMPLETE. GRADQ 76 STATEMENT MUST END WITH THE SEQUENCE: ¡ARITHM-EXPR¿, ")". GRADQ 77 I/O OPERATIONS SHOULD NOT BE PERFORMED ON MERLIN FILES. GET

DISPLA 78 NUMBER OF SUBSCRIPTS OR ARGUMENTS, DIFFERENT THAN DECLARED. XGEN 79 FILE NAME WAS EXPECTED AFTER "=". PICK 80 PICK STATEMENT MUST END WITH RIGHT PARENTHESIS. PICK 81 "=" EXPECTED AFTER THE "FILE" KEYWORD. PICK 82 "FILE" EXPECTED AFTER ";". PICK 83 SEMICOLON EXPECTED AFTER

EXPRESSION. PICK 84 ”=” EXPECTED AFTER THE ”REC” KEYWORD. PICK 85 SEMI-COLON WAS EXPECTED AFTER FILE NAME. PICK 86 ”REC” OR ”FILE” EXPECTED AFTER ”(”, OR ”=” NOT DETECTED. PICK 87 LEFT PARENTHESIS EXPECTED AFTER ”PICK”. PICK 88 PICK STATEMENT SYNTACTICALLY INCOMPLETE. PICK 89 ”=” EXPECTED AFTER ”REC”. PICK 90 ”REC” EXPECTED. PICK 91 ”=” EXPECTED. GODFA 92 THIS STATEMENT MUST END WITH A RIGHT PARENTHESIS. GODFA 93 IDENTIFIER EXPECTED. GODFA 94 ”.” EXPECTED. GODFA 95 ”X” EXPECTED. GODFA 96 LEFT PARENTHESIS EXPECTED. GODFA 97 GODFATHER STATEMENT SYNTACTICALLY INCOMPLETE. GODFA 98 THIS STATEMENT MUST END WITH A RIGHT PARENTHESIS. REWI 99 IDENTIFIER EXPECTED AFTER ”=”. REWI 100 REWIND STATEMENT SYNTACTICALLY INCOMPLETE. REWI 101 ”=” EXPECTED AFTER FILE INDICATOR. REWI 102 ”REWIND” MUST BE FOLLOWED BY LEFT PARENTHESIS. REWI 103 THIS STATEMENT MUST END MITH A RIGHT PARENTHESIS. HIDE 104 IDENTIFIER EXPECTED AFTER ”=”. HIDE 105 HIDEOUT STATEMENT SYNTACTICALLY INCOMPLETE. HIDE 106 ”=” EXPECTED AFTER FILE INDICATOR. HIDE 107 ”HIDEOUT” MUST BE FOLLOWED BY LEFT PARENTHESIS. HIDE 108 ”DISCARD” MUST END WITH A RIGHT PARENTHESIS. DISCA 109 INVALID FILE NAME TO DISCARD. DISCA 110 FILE NAME EXPECTED. DISCA 111 ”=” EXPECTED. DISCA 112 ”DISCARD” MUST BE FOLLOWED BY LEFT PARENTHESIS. DISCA 113 DISCARD STATEMENT SYNTACTICALLY INCOMPLETE. DISCA 114 ”=” EXPECTED AFTER ARRAY ELEMENT. EXE 115 ”FROM” SHOULD FOLLOW MACRO NAME. EXECU 116 FILE NAME EXPECTED AFTER ”FROM”. EXECU 117 ”FROM” OR ”EOL” EXPECTED AFTER MACRO NAME. EXECU 118 LEFT PARENTHESIS EXPECTED. INI 119 IDENTIFIER EXPECTED. INI 120 BE CAREFULL, YOU RE-INITIALIZE ARRAY X. INI 121 ”.” EXPECTED AFTER ”X”. INI 122 IDENTIFIER EXPECTED AS FILE NAME, AFTER ”X.”. INI 123 BE CAREFULL, YOU RE-INITIALIZE ARRAY L. INI 124 ”.” EXPECTED AFTER ”L”. INI 125 IDENTIFIER EXPECTED AS FILE NAME AFTER ”L.”. INI 126 BE CAREFULL, YOU RE-INITIALIZE ARRAY R. INI 127 ”.” EXPECTED AFTER ”R”. INI 128 IDENTIFIER EXPECTED AS FILE NAME AFTER ”R.”. INI 129 ”X”, ”L” OR ”R” KEYWORD EXPECTED. INI 130 ILLEGAL CHARACTER SEQUENCE DETECTED. INI 131 INIT STATEMENT SYNTACTICALLY INCOMPLETE. INI 132 ARITHMETIC EXPRESSION EXPECTED INTO STATEMENT. XPR 133 IDENTIFIER EXPECTED AS LOOP CONTROL INDEX. LOOP 134 LOOP CONTROL INDEX NOT DECLARED. LOOP 135 NESTED LOOPS, HAVE THE SAME CONTROL INDEX. LOOP 136 ”FROM” EXPECTED. LOOP 137 ”TO” EXPECTED. LOOP 138 ”BY” EXPECTED. LOOP 139 ”ENDLOOP” NOT FOLLOWED BY EOL. ENDLOO 140 ”ENDLOOP” FOUND, WITH NO CORRESPONDING ”LOOP” STATEMENT. ENDLOO 141 A LOOP CONTROL VARIABLE SHOULD NOT BE ASSIGNED A VALUE. EXE 142 ”EXIT” NOT FOLLOWED BY EOL. EXITL 143 ”EXIT” ILLEGAL OUTSIDE A LOOP STRUCTURE. EXITL 144 ”ELSE” FOUND WITHOUT A CORRESPONDING ”IF”. ELSE 145 ”ENDIF” FOUND WITHOUT A CORRESPONDING ”IF”. ENDIF 146 ONE OR MORE LOOPS DO NOT TERMINATE. (”ENDLOOP” MISSING) TERMIN 147 ONE OR MORE BLOCK IFS DO NOT TERMINATE. (”ENDIF” MISS-

ING) TERMIN 148 ILLEGAL JUMP INTO A BLOCK IF OR LOOP. JUMPIN 149 "FROM" EXPECTED AFTER VARIABLE LIST. GET 150 IDENTIFIER EXPECTED AS FILE NAME AFTER "FROM". GET 151 "FROM" EXPECTED AFTER VARIABLE LIST. GET 152 "GET" STATEMENT NOT FOLLOWED BY "EOL". GET 153 IDENTIFIER IN ASSIGNMENT, IS NOT A SIMPLE VARIABLE. EXE

GET 154 STATEMENT MUST END WITH RIGHT PARENTHESIS. DFL 155 "." EXPECTED AFTER KEYWORD. DFL 156 INCOMPATIBLE INDEXED PARAMETER WITH STATEMENT. DFL 157 LEFT PARENTHESIS EXPECTED. DFL 158 STATEMENT SYNTACTICALLY INCOMPLETE. DFL 159 SIMPLE VARIABLE FOLLOWED BY "(", OR "[", INTO AN EXPRESSION. XCHECK 160 INTRINSIC VARIABLE FOLLOWED BY "(",OR "[". XCHECK 161 REFERENCE TO A FUNCTION WITH NO "[". XCHECK 162 SIMPLE ARRAY CALLED WITH NO "[" AFTER ITS NAME. XCHECK 163 INTRINSIC ARRAY CALLED WITH NO "[" AFTER ITS NAME. XCHECK 164 MORE THAN ONE SUBSCRIPT IN INTRINSIC ARRAY. XGEN 165 THE WHOLE MACRO FILE READ. TERMINATING "CLEAR" NOT FOUND. EXECU 166 MULTIPLE ENTRY OF A KEYWORD. MULTI

ACCUM 167 A LOOP CONTROL VARIABLE SHOULD NOT BE INPUT A VALUE. GET

Chapter 9

Program limitations and parameters

The only limitation of this program is the length of its tables. They are all dimensioned by parameters that may be easily set to any desired value. We list the parameters that dimension the tables used in the program.

9.1 Parameter MXLAB

Maximum number of labels allowed in a program. This parameter dimensions the label and label-location tables. Current value: 100 Defined in subroutine MCL.

9.2 Parameter MXMOV

Maximum number of `MOVE TO` statements allowed in a program. This parameter dimensions the move table. Current value: 100 Defined in subroutine MCL.

9.3 Parameter MXTYP

Maximum number of entries allowed in the type table. This parameter dimensions the type, type definition and type-link tables. Current value: 500 Defined in subroutine MCL.

9.4 Parameter MXVAR

Maximum number of variables, arrays and statement functions allowed in a program. (Merlin variables and arrays are also included.) This parameter dimensions the var, kind, memory- location, dimensionality and position tables. Current value: 500 Defined in subroutine MCL.

9.5 Parameter MXINFO

Maximum number of entries allowed in the information table. Current value: 200 Defined in subroutine MCL.

9.6 Parameter MXLOO

Maximum number of loop nesting level allowed in a program. This parameter dimensions the loop table. Current value: 100 Defined in subroutine MCL.

The following parameters are used to define various properties of the program.

9.7 Parameter NPRE

The number of Merlin variables together with arrays and intrinsic functions. Current value: 42 Defined in subroutine MCL.

9.8 Parameter MAXMES

Total number of error messages that the compiler may issue. Current value: 167 Defined in subroutine ERROR.

9.9 Parameter MAXERR

Maximum number of error messages that may be issued for one source line. This parameter dimensions the EBUFF array in the MCLERB common block. Current value: 20 Defined in subroutines ERROR, FEBUFF.

9.10 Parameter NES

Number of statements that subroutine EXE handles. This is actually the number of statements that may appear next to the `JUST` keyword of the `WHEN` statement. (Excluding the assignment statement.) Current value: 57 Defined in subroutine EXE.

9.11 Parameter NNUI

Number of identifiers that may not be used as variable, array or statement function names. Current value: 5 Defined in function RESE.

9.12 Parameter NK

Number of keywords allowed in the `ACCUM` statement. Current value: 3 Defined in subroutine: ACCUM.

9.13 Parameter NRA

Number of valid keywords for the `RANDOM` statement. Current value: 8 Defined in subroutine MULTI.

9.14 Parameter NCO

Number of valid keywords for the `CONGRA` statement. Current value: 7 Defined in subroutine MULTI.

9.15 Parameter NDF

Number of valid keywords for the `DFP` statement. Current value: 8 Defined in subroutine MULTI.

9.16 Parameter NBF

Number of valid keywords for the `BFGS` statement. Current value: 8 Defined in subroutine MULTI.

9.17 Parameter NRO

Number of valid keywords for the RQLL statement. Current value: 7 Defined in subroutine MULTI.

9.18 Parameter NGR

Number of valid keywords for the `GRAPH` statement. Current value: 7 Defined in subroutine MULTI.

9.19 Parameter NSI

Number of valid keywords for the `SIMPLEX` statement. Current value: 7 Defined in subroutine MULTI.

9.20 Parameter NAU

Number of valid keywords for the `AUTO` statement. Current value: 3 Defined in subroutine MULTI.

9.21 Parameter IDLEN

The length of an identifier (counted in characters). The length of a string is always: 2*IDLEN. Current value: 30 Defined in subroutines MCL, PCG, IF, VARIA, WHEN, EXE, GET, EXECU, DISPLA, STAFU,MULTI, STEPO, MARGI, ACCUM, LOOP, LEX, XGEN, PSHALP, PFIX, XCHECK.

9.22 Parameter LINLEN

The length of each source line. (Counted in characters.) Several other parameters are calculated by the program using `LINLEN`: Maximum number of identifiers in an MCL source line: (LINLEN+1)/2 Maximum number of number in an MCL source line: (LINLEN+1)/2 Maximum number of arguments in a statement function: (LINLEN- 13)/2 Current value: 80 Defined in subroutines MCL, IF, WHEN, EXE, GET, DISPLA, STAFU, MULTI, STEPO, MARGI, ACCUM, LOOP, LEX, XGEN, POPUP, PUSHDN, PUSH, POP, PSHALP, PSHSTR, PSHNUM, POPBR, PUSHBR, PFIX, XPR, XCHECK.

Chapter 10

Efficient use of MCLCOM

We state a few general guidelines that intent to help the user to write efficient MCL programs.

10.1 Simplify complicated expressions

Since MCLCOM 1.0, does not perform any object code optimization, the MCL user should avoid situations like : A = 3*4/ 5 ; instead one should prefer to write: A = 2.4

10.2 Use Functions and Loops with care.

Since both Function declarations and Function calls generate quite a few lines of code, keep in mind that excessive Function usage will lengthen the produced MOC. Similarly the LOOP statements, while contributing to the clarity of the program, they produce more code than what a combination of WHEN and MOVE TO statements would.

10.3 Use BOUNDS and DEBUG options appropriately

If both options are ON, a longer MOC will result. One should use the DEBUG options for debugging purposes only. To a lesser extend, the above is true for the BOUNDS option as well.

Bibliography

[1] F. James and M. Roos, Comp. Phys. Commun. 10 (1975) 343.

[2] MCL, Optimization oriented programming language, C.S. Chassapis, D.G. Papageorgiou, and I.E. Lagaris, Comp. Phys. Comm. 52 (1989) 223-239.

[3] MERLIN, A portable system for multidimensional optimization, G.A. Evangelakis, J.P. Rizos, I.E. Lagaris and I.N. Demetropoulos, Comp. Phys. Comm. 46 (1987) 401-415.

[4] MERLIN-2.0, Enhanced and programmable version, D.G. Papageorgiou, C.S. Chassapis, and I.E. Lagaris, Comp. Phys. Comm. 52 (1989) 241-247.

[5] Merlin-2.1, Double precision, D.G. Papageorgiou and I.E. Lagaris, Comp. Phys. Comm. 58 (1990) 119-125.

[6] Theory and practice of compiler writing, J.P. Tremblay and P.G. Sorenson, McGraw-Hill, 1985, section 2-4.4.

[7] Algorithms and data structures, N. Wirth, Prentice-Hall, 1986.

[8] The art of computer programming, vol. 3, Sorting and searching, D.E. Knuth, Addison-Wesley, 1973.

[9] American National Standard programming Language FORTRAN (ANSI X3.9-1978), American National Standards Institute, New York, 1978.