ELSEVIER

Computer Physics Communications 109 (1998) 250–275

Computer Physics Communications

The Merlin Control Language for strategic optimization

D.G. Papageorgiou [a], I.N. Demetropoulos [a], I.E. Lagaris [b,1]

[a] *Department of Chemistry, University of Ioannina, P.O. Box 1186, GR 45110 Ioannina, Greece*

[b] *Department of Computer Science, University of Ioannina, P.O. Box 1186, GR 45110 Ioannina, Greece*

Received 8 December 1997

Abstract

MCL is the programming language of the MERLIN optimization environment. It can be used for the implementation of efficient optimization strategies, abolishing to a great extend the need for user intervention. The language is simple to learn and its structure is similar to Fortran. We report on successful applications where MCL played an instrumental role, as for example in molecular physics problems and in the training of neural networks. © 1998 Elsevier Science B.V.

Keywords: Optimization; Optimization strategies; Global optimization; Programming language; MERLIN

PROGRAM SUMMARY

Title of program: MCL-3.0

Catalogue identifier: ADHR

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland

Computer for which the program is designed and others on which it is operable:
Computers: Designed to be portable. Developed on a Sun SPARCstation 5. Tested on Sun Classic (SunOS 4.1.3C), Sun Ultra-2 (SunOS 5.5.1), SGI Challenge-M (IRIX 6.2), CD4680 (EP/IX 1.4.3), Intel-based PCs (Linux 2.0.18 and MS-Windows 95), Macintosh (MacOS 7.5); *Installations:* University of Ioannina, Greece

Programming language used: ANSI Fortran-77

Memory required to execute with typical data: Approximately 256 Kwords on a Sun SPARCstation 5

No. of bits in a word: 32

No. of processors used: 1

Has the code been vectorised or parallelized?: No

No. of bytes in distributed program, including test data, etc.: 219621

Distribution format: uuencoded compressed tar file

Keywords: optimization, optimization strategies, global optimization, programming language, MERLIN

Nature of physical problem
Complex optimization problems that cannot be solved efficiently via a single algorithm and rather need to be tackled by a multi-algorithm strategy.

Method of solution
This special purpose language is built to allow the user to devise

[1] Corresponding author. Email: lagaris@cs.uoi.gr.

PII S0010-4655(98)00006-X

successful and efficient strategies based on diverse algorithmic approaches, in a systematic and orderly manner, using the MERLIN optimization environment [1].

Restrictions on the complexity of the problem
None.

Typical running time:
Depending on the MCL program to be compiled. The test run took 1.66 sec on a Sun SPARCstation 5.

Unusual features of the program:
Must be installed using the MERLIN-3.0 [1] installer program. The procedure is described in detail in the MERLIN User Manual.

References
[1] D.G. Papageorgiou, I.N. Demetropoulos, I.E. Lagaris, MERLIN-3.0. A multidimensional optimization environment, Comput. Phys. Commun. 109 (1998) 227–249, preceding article.

LONG WRITE-UP

1. Introduction

MCL stands for MERLIN Control Language and is a high level, special purpose language. Using MCL one can drive the MERLIN optimization environment [1] in a convenient and intelligent manner. Use of MCL abolishes to a great extend the need for user intervention which is particularly annoying when dealing with large and complex problems that require a lot of computing time. In addition, MCL can be used to implement strategies aiming at a host of different objectives. For instance, one can use MCL to substantiate global optimum searches [2], to train artificial neural networks [3,4], or to develop smaller but convenient utility programs.

For a better understanding of this article and for immediate use of MCL, one needs to be familiar with the MERLIN optimization environment at the user level. In order to achieve precise and economical description, we adopt the following conventions: MCL statements, keywords and sample code fragments are printed using a monospaced font. Optional items are enclosed in square brackets ([ ]). Braces ({ }) indicate items that may be repeated zero or more times. Alternative items are separated by a vertical bar (|). Arguments for which the user supplies a value are shown in italics. x_i designates the ith parameter of the MERLIN objective function.

2. The structure of an MCL program

An MCL program consists of exactly one main program and zero or more subprograms. Subprograms are described in Section 6. The main program has the form

```
PROGRAM
      MCL statements
END
```

The main program as well as the subprograms are made up from sequences of lines containing MCL statements. Each line can be up to 120 characters long; longer lines are subject to truncation, and may lead to inexplicable errors. An MCL statement may be split across two or more lines by using the continuation character & as the last character of the line. Up to 10 continued lines may be used. Blank lines, as well as leading blank or tab characters, are ignored. However, they may be used so as to improve the readability of a program. Comments may be entered at the end of any statement, following the percent character %.

Table 1
MCL operators according to their precedence

MCL operator	Precedence
NOT	highest precedence
**	
*, /	
+, -	
>, <, >=, <=, == , #	
AND	
OR	
XOR	lowest precedence

3. Variables, arrays and expressions

3.1. Data types and operators

MCL supports only one data type: floating point numbers. Floating point constants must have at least one digit before the decimal period. For example, `0.15` is an acceptable number, while `.15` is not.

There are three types of operators in MCL: arithmetic, relational and logical. The arithmetic operators are the usual ones: +, -, *, / and ** standing for addition, subtraction, multiplication, division and exponentiation. The relational operators are >, <, >=, <=, == and # and they are equivalent to the Fortran, .GT., .LT., .GE., .LE., .EQ. and .NE. operators. The logical operators are NOT, AND, OR and XOR, standing for the usual logical operations in an obvious notation. All operators are listed in Table 1 in order of descending precedence. The associativity of all the above operators is from left to right.

3.2. Simple variables and arrays

A variable is a name which refers to a storage location in the MERLIN run-time memory. An array is a series of storage locations referenced by the same name. Arrays can have an arbitrary number of dimensions. Each dimension has a specific size and its bounds determine the numbering of the individual elements.

Variable and array names are case insensitive alphanumeric strings of up to 30 characters, the first of which must be a letter. Longer names are subject to truncation without any warning. Underscores can be used anywhere in between for the sake of clarity and are ignored by the compiler. The names JUST, THEN, FROM, BY, and TO are reserved and cannot be used as variable or array names.

All variables and arrays used in an MCL program must be declared using the VAR declaration statement

VAR *name* { ; *name* }

where *name* is either a simple variable or an array specification of the form

array_name [l:u { , l:u }]

The l:u specifiers are the lower and upper dimension bounds, respectively. They must be integer constants (positive negative or zero) with $l \leq u$. For example,

```
VAR a_long_but_acceptable_name; minimum; test_value
VAR root; a[1:20]; b[-10:10,1:100]
```

Most array references are made to a specific array element through its subscripts. The number of subscripts must always match the number of dimensions in the array. An array subscript can be any MCL expression. The

$$\texttt{NOT } expr = \begin{cases} 1 & \text{if } expr = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$expr_1 \texttt{ AND } expr_2 = \begin{cases} expr_1 & \text{if } expr_2 \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$expr_1 \texttt{ OR } expr_2 = \begin{cases} 1 & \text{if } expr_2 \neq 0 \\ expr_1 & \text{otherwise} \end{cases}$$

$$expr_1 \texttt{ XOR } expr_2 = \begin{cases} expr_1 & \text{if } expr_1 \neq 0 \text{ and } expr_2 = 0 \\ expr_2 & \text{if } expr_2 \neq 0 \text{ and } expr_1 = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$expr_1 \; relop \; expr_2 = \begin{cases} 1 & \text{if } expr_1 \; relop \; expr_2 \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Note: *relop* is any of the relational operators >, <, >=, <=, == and #

expr, $expr_1$ and $expr_2$ are MCL expressions

Fig. 1. Rules for evaluating MCL expressions.

entire array can be specified without subscripts only as an actual argument in a subprogram call.

There is a number of intrinsic variables and arrays used to monitor the optimization process that should not be declared in a VAR statement. They can be used in an MCL expression as any ordinary variable or array. It is not possible however to change their values through assignment or GET statements since intrinsic variables and arrays assume their values from the MERLIN run-time environment. A complete listing along with their significance is given in Appendix A.

3.3. Expressions and assignments

An expression represents a single value created from the evaluation of its components. The value of an expression is stored in a variable or used as an argument. An expression is composed of numeric constants, variables, array elements, function references, operators and parentheses. Expressions are evaluated in order of precedence of the operators, then from left to right if the operators have equal precedence. Parentheses can be used to override the order of evaluation. All elements in an expression must be defined by the time the expression is evaluated. The rules for evaluating expressions involving the above relational and logical operators are given in Fig. 1. Note that an MCL expression is considered by convention false if its value is zero; true otherwise.

An assignment statement stores a value from an expression into a variable or array element,

var_name = expression

where *var_name* is any variable or array element and *expression* is any MCL expression. A number of sample assignments and expressions is shown in Table 2.

Table 2
Sample MCL assignments and expressions

Plain expressions	MCL expressions
$t = ax^2 + bx + c$	`T = A*X**2 + B*X + C`
$z = m + 1/a$	`Z = M + 1/A`
$a_i = p_i - 1$	`A[I] = P[I] - 1`
$p = (a < 0) \cup (b \geq 1)$	`P = (A < 0) OR (B >= 1)`
$q = (a > 0) \cap (b \leq 1)$	`Q = (A > 0) AND (B <= 1)`

4. Flow control statements

An MCL program runs from the first statement to the last, in sequence, unless directed to do otherwise by a flow control statement. These statements include direct transfer to another statement (`MOVE TO`), repetition of statements (`LOOP`), conditional branching to several different statements (`IF` and `WHEN`) and subprogram calls (`CALL` is described in Section 6.2). In addition one can pause or terminate execution of the MCL program.

4.1. Labels and the MOVE TO statement

The `MOVE TO` statement causes an unconditional branch to another statement:

`MOVE TO` *label*

where *label* is the symbolic name of a label at which control is to be transferred. Both `MOVE TO` and `MOVETO` are acceptable.

A label statement has the form

label :

where *label* is the label name. Labels are used inside a program as addresses where the control may be transferred. Duplicate label names are not allowed in the same program unit.

4.2. The WHEN statement

The `WHEN` statement causes execution of a statement only when specific conditions are met:

`WHEN` *condition* `JUST` *ncnlcs*

with *ncnlcs* being any executable statement besides the `LOOP` and the `IF-THEN` and `WHEN` conditional statements. If *condition* is true, then the *ncnlcs* statement is executed. If *condition* is false, control transfers to the statement at the next line and the *ncnlcs* statement is ignored. As an example one may have

```
WHEN R>BIG JUST BIG=R
```

4.3. The block–if

A block–if consists of two groups of statements, one of which is executed when a logical condition is met:

```
IF condition THEN
      statement block
[ ELSE
```

```
      statement block ]
END IF
```

with *condition* being any valid MCL expression. The ELSE statement and the second block are optional. If *condition* is true, the first block is executed and the control transfers to the statement immediately following the END IF statement. If *condition* is false and if a second block exists, it is executed and the control transfers to the statement immediately following the END IF statement. Each block may contain additional, nested block–if constructs. Since each block–if must be terminated by an END IF, there is no ambiguity in the execution path. Both END IF and ENDIF are acceptable. An example is shown below:

```
IF FUNMODE == 0 THEN
   BFGS ( NOC=2000 )
ELSE
   LEVE ( NOC=500 )
   TERMDIS
END IF
```

Note that control cannot be transferred into an IF block from outside that block. In addition, loop statements must be terminated inside the same IF block where they began.

4.4. The PAUSE and FINISH statements

PAUSE temporarily suspends execution of an MCL program. One can resume execution with an entry from the keyboard. The syntax is

```
PAUSE
```

FINISH terminates execution of an MCL program. Control is then transferred to the MERLIN operating system. The syntax is

```
FINISH
```

An MCL program also terminates when the END statement of the main program is reached.

4.5. The LOOP statement

The LOOP statement provides the means for repeating a sequence of statements. The syntax is

```
LOOP var FROM init_val TO final_val [ BY step ]
      statement block
END LOOP
```

where *var* is a simple variable used to control execution of the LOOP statement, referred to as the *loop control variable*, and *init_val*, *final_val*, and *step* are expressions. If *step* is omitted it defaults to 1.

Upon entry to a LOOP–block, the expressions *init_val*, *final_val*, and *step* are evaluated. These values are used to compute the number of repetitions of the LOOP using the formula

$$n = \max\left\{0, \operatorname{int}\left(\frac{final\_val - init\_val + step}{step}\right)\right\}.$$

Once the count is determined, it does not change, even if *init_val*, *final_val*, and *step* are assigned new values inside the loop. *var_name* is assigned the value of *init_val* and the statements inside the loop–block are executed

```
LOOP I FROM 1 TO DIM
   WHEN FIX[I] == 0 JUST EXIT
END LOOP
```

Fig. 2. Exit statement example

sequentially until the corresponding END LOOP is reached. *var_name* is then incremented by the value of *step* and control transfers to the statement following the LOOP statement. The above process is repeated *n* times.

Each LOOP statement must have a corresponding END LOOP statement, which can be written either as END LOOP or as ENDLOOP. Nested LOOP structures are allowed. An example is shown below:

```
LOOP I FROM 1 TO DIM
   LOOP J FROM 1 TO DIM
      H[I,J] = 0
   END LOOP
   H[I,I] = 1
END LOOP
```

If a variable is used as a loop control variable, it should not be used as a control variable in another loop nested inside the first one. Loop control variables should not be assigned any value, via an assignment or GET statement inside the loop they control. The only way to enter a LOOP is by its initial LOOP statement. Attempting to transfer the control inside the loop–block, from the outside, will result in an error. IF statements inside a loop–block must terminate inside the same block.

4.6. The EXIT statement

The EXIT statement causes a loop to terminate, with execution resuming immediately after the corresponding END LOOP. The syntax is

EXIT

EXIT can be used only inside a LOOP–block. An example of its use is shown in Fig. 2. After exiting the loop, the loop control variable retains its previous value.

5. Input–output statements

MCL handles its input via the GET statement and its output via the DISPLAY statement.

5.1. The GET statement

The syntax is

GET *var* { ; *var* } [FROM '*file_name*']

where *var* is either a simple variable or an array element. *file_name* is the name of the file where the corresponding values reside. For every GET statement the input data must be in a single line. When *file_name* is omitted, MERLIN's default input file is assumed. A few examples follow:

```
GET  A; I; B[I-2]; Pressure; First_Root
GET  C; D FROM 'special_file'
```

5.2. The DISPLAY statement

The syntax is

DISPLAY *var* { ; *var* } [TO *'file_name'*]

where *var* is either an expression or a character string delimited by single quotes. Character strings may contain any character except the single quote (’). Single quotes may be output using the escape sequence \’. Each arithmetic expression is evaluated and its value is output using a fixed format. For each delimited character string the output consists of the string itself. *file_name* is the name of the file, to which the output is directed. When *file_name* is omitted, MERLIN's default output file is assumed. A few examples are shown below

```
DISPLAY  '\'Quality\' of fit is'; QF
DISPLAY  'New period is'; SQRT[4*X[1]]; 'msec' TO 'FILE1'
```

Assuming QF=0.93 and x_1=1, we obtain the following output in the default output file, and in FILE1, respectively:

```
'Quality' of fit is 0.93
New period is 2. msec
```

5.3. The CLOSE statement

When a GET or DISPLAY statement that involves a file is executed, the corresponding file is associated (‘opened’) with the currently running MCL program by the MERLIN run-time environment. The file remains ‘open’ for further GET or DISPLAY statements. The CLOSE statement is used to disassociate (‘close’) a file from the currently running MCL program and flush its contents to disk. The file is re-associated (‘reopened’) when another GET or DISPLAY statement contains a reference to it. The syntax is

CLOSE (FILE = *'file_name'*)

Note that a file is never opened as a result of a REWIND or GOEOF MERLIN command.

6. Subprograms

6.1. Writing a subprogram

MCL supports only one type of subprogram (analogous to the Fortran subroutine subprogram). An MCL subprogram has exactly the same overall structure with the main program, except that the first statement is a SUB statement:

SUB *sub_name* [(*list_of_dummy_arguments*)]
    declaration statements
    executable statements
END

where *sub_name* is a symbolic name assigned to the subprogram. It must not coincide with the name of any other subprogram, or with the intrinsic subprograms listed in Section 6.2. The optional *list_of_dummy_arguments* is used to transmit information between the calling program and the subprogram. Its form resembles a VAR declaration (without the word VAR however). Simple variables as well as arrays may be present in the *list_of_dummy_arguments*. Some examples follow:

```
SUB complex ( alpha; beta; point[1:10,1:3] )
SUB another_one ( big[1:200]; trm )
```

The names in the *list_of_dummy_arguments* should not coincide with any of the intrinsic variables, arrays or functions. MCL stores arrays in memory using the convention that their last subscript varies most rapidly. Hence one can use an adjustable upper bound denoted by a *, in the last dimension on an array, as in

```
SUB adj ( arr[-5:5,1:*] )
SUB increase ( v; a[1:10,1:3,-2:*]; dest )
```

All variables and statement functions declared in a subprogram, as well the names in the *list_of_dummy_arguments*, are local to the subprogram. They are not known to the main program or any other subprogram. The only way to communicate information to or from a subprogram is through its arguments.

6.2. Calling a subprogram

A subprogram is accessed by means of a CALL statement which contains the name of the subprogram and optionally a list of arguments,

CALL *sub_name* [(*list_of_actual_arguments*)]

The arguments in the *list_of_actual_arguments* have a one to one correspondence with the dummy arguments in the SUB statement. Arguments are always passed by reference to the subprogram. If the dummy argument is a simple variable, then the actual argument must be a simple variable, array element, or a valid MCL expression. If the dummy argument is an array, the actual argument must be an array; either one defined by the user, or an intrinsic one. Note, however, that when an intrinsic variable or array is used as an actual argument in a CALL statement, any changes made by the subprogram are not communicated back to the calling program.

There are three intrinsic subprograms:

- SUB NORM (A[1:*]; K; L; R)

  This subprogram calculates the Lth norm of the first K elements of array A using the formula

$$\mathrm{R} = \begin{cases} \left(\sum_{i=1}^{\mathrm{K}} |\mathrm{A}(i)|^{\mathrm{L}} \right)^{1/\mathrm{L}} & \text{if } \mathrm{L} > 0, \\ \max_{i=1,\mathrm{K}} \{|\mathrm{A}(i)|\} & \text{if } \mathrm{L} = -1. \end{cases}$$

- SUB GETSEED (A[1:25])
- SUB SETSEED (A[1:25])

  These subprograms are used to manipulate the seed of the MERLIN random number generator. The seed (25 integers stored in array A) contains the information needed to restart the random sequence. More specifically GETSEED returns the seed in array A while SETSEED restarts the random sequence using the contents of array A.

6.3. The SUBRETURN statement

A subprogram returns control to the calling program as soon as its closing END statement is reached. In addition, SUBRETURN can be used wherever appropriate to terminate execution of the subprogram. The syntax is

SUBRETURN

Note that SUBRETURN cannot be used in the main program.

6.4. The FUNCTION statement

The FUNCTION declaration statement allows an expression to be referenced through a symbolic name, in a way reminiscent of the Fortran statement functions. Function declarations are placed immediately after all VAR declarations but before any executable statements,

FUNCTION *function_name* [*arg*, { *arg* }] = *expression*

where *function_name* is the symbolic name used to reference the function, *arg* is a dummy argument and *expression* is any valid MCL expression that involves the arguments from the dummy argument list, variables and functions already declared, and intrinsic variables and functions. A function must not contain a forward reference to another function, and its symbolic name should not coincide with any of the reserved names or with any other symbolic names previously declared. Array elements and intrinsic variables are not allowed into the dummy argument list of a function. Some examples of valid FUNCTION statements are given below:

```
FUNCTION G[A,B,C] = A**B-B**C
FUNCTION Root[A,B,C] =(-B+SQRT[B**2-4*A*C])/(2*A)
FUNCTION F[A,B] = Root[3.0,A,X[4]] + B**VALUE
```

When referencing an intrinsic or a statement function, the arguments can be either simple variables or valid MCL expressions. There should be at least one dummy argument in the declaration and call of a function, even if the function does not need one. For example,

```
VAR dummy
FUNCTION SumIt[dummy] = X[1]**2 + X[2]**2 + X[3]**2
      ...
r = SumIt[dummy]/3
s = SumIt[0]
```

There is a number of intrinsic functions whose names are reserved, and need no declaration. They are all listed in Table 3 along with their meaning.

7. Non-parametric statements

These statements are in direct correspondence with the synonymous MERLIN commands an interactive user would issue to guide the minimization process. They are ADJUST, ALIASDIS, ANAL, CFLAGDIS, EVALUATE, FAST, FIXALL, FLAGDIS, FULLBACK, FULLPRINT, GENERAL, GNORM, GRADDIS, HALFPRINT, JANAL, JNUMER, LASTBACK, LIMITS, LOOSALL, MODEDIS, NOBACK, NOEVAL, NOPRINT, NOTARGET, NUMER, QUAD, RESET, REVEAL, SHORTDIS, SOS, STEPALL, STEPDIS, TERMDIS and VALDIS. They accept no parameters. Their syntax is

command_name

where *command_name* is any of the above statements.

8. Parametric statements

These statements use order independent parameters and are distributed among the following categories:

- index parametric statements;
- index–value parametric statements;
- index–string parametric statements;

Table 3
MCL intrinsic functions

Function name	*Operation*	*Remarks*
ABS[z]	$\|z\|$	
ACOS[z]	arccos z	
ACOSH[z]	arccosh z	$z \geq 1$
ASIN[z]	arcsin z	
ASINH[z]	arcsinh z	
ATAN[z]	arctan z	
ATANH[z]	arctanh z	$\|z\| < 1$
COS[z]	cos z	
COSH[z]	cosh z	
EXP[z]	e^z	
FACT[z]	Factorial of the nearest integer to z	$z \geq 0$
GRADNORM[*l*]	L_1, L_2 or L_∞ $(l = 1, 2, -1)$ gradient norm	
GRMS[z]	RMS gradient = $\left(\frac{1}{N}\sum_{i=1}^{N}\frac{\partial f}{\partial x_i}^2\right)^{1/2}$	z is ignored
LOG[z]	ln z	$z > 0$
LOG10[z]	log z	$z > 0$
MAX[$z_1, z_2, \ldots, z_k$]	max $\{z_1, z_2, \ldots, z_k\}$	
MEAN[$z_1, z_2, \ldots, z_k$]	$(1/k)\sum_{i=1}^{k} z_i$	
MIN[$z_1, z_2, \ldots, z_k$]	min $\{z_1, z_2, \ldots, z_k\}$	
MOD[z_1, z_2]	z_1 modulo z_2	
MIN[$z_1, z_2, \ldots, z_k$]	min $\{z_1, z_2, \ldots, z_k\}$	
RAN[z]	Random number in (0,1)	z is ignored
ROUND[z]	Nearest integer to z	
SIN[z]	sin z	
SINH[z]	sinh z	
SQRT[z]	$\sqrt{z}$	$z \geq 0$
TAN[z]	tan z	
TANH[z]	tanh z	
TRUNC[z]	Integer part of z	

- general parametric statements;
- panel parametric statements.

For details on the functionality of each command and the meaning of the keywords one should consult the MERLIN users manual. Only the relevant MCL syntax is presented here.

8.1. Index parametric statements

These are the statements CONFIDENCE, FIX, LOOSE, LDEMARGIN, RDEMARGIN and NONAME. They obey the following syntax:

command_name (*key.index* { ; *key.index* })

index is an expression. The keywords denoted by *key* can be

- X for the FIX, LOOSE and NONAME statements.
- L for the LDEMARGIN statement.
- R for the RDEMARGIN statement.

A few examples follow:

`FIX ( X.1; X.5 )`	Fixes parameters x_1 and x_5.
`LDEMARGIN ( L.2; L.4 )`	Removes the lower bound from x_2 and x_4.
`LOOSE ( X.1; X.7 )`	Looses parameters x_1 and x_7.

8.2. Index–value parametric statements

These are the statements POINT, STEP, LMARGIN, RMARGIN and FLAG. They obey the following syntax:

command_name (*key.index=val* { ; *key.index=val* })

index and *val* are expressions. The keywords denoted by *key* can be
- X for the POINT statement.
- L for the LMARGIN statement.
- R for the RMARGIN statement.
- S for the STEP statement.
- F for the FLAG statement.

Some examples follow

`POINT ( X.I=SQRT(I**2+1)/3 )`	Assigns values to x_i.
`LMARGIN ( L.1=8; L.4=-10 )`	Sets lower bounds to x_1 and x_4.

8.3. Index–string parametric statements

These are the statements GODFATHER, CFLAG and MIXED. They obey the following syntax:

command_name (*key.index=str* { ; *key.index=str* })

index is an expression while *str* is a delimited string. The keywords denoted by *key* can be
- X for the GODFATHER and MIXED statements.
- C for the CFLAG statement.

Some examples follow

`GODFATHER ( X.1='Rho'; X.2='Psi' )`	Assigns symbolic names to x_1, x_2.
`CFLAG ( C.1='/disk1/mc1/out' )`	Assigns a value to the first character flag.

8.4. General parametric statements

These are the statements CLOSE, DELETE, DISCARD, REWIND, GOEOF, HIDEOUT, STOP, RETURN, ALIAS, UNALIAS, EPILOG, TITLE, PDUMP, PANELON, PANELOFF, PSTATUS, GRADCHECK, TARGET and QUIT. Their syntax is

command_name (*key=str* { ; *key=str* })

Some or all the keywords may be omitted, according to the command. Multiple instances of the same keyword are not allowed. In the description that follows, *file_name* is any valid file name and *panel_command* is any of the MERLIN commands that use the panel mechanism.
- DELETE (FILE = *'file_name'*)
  file_type can be either TEXT or BIN corresponding to a text or binary MERLIN file. If omitted, the MERLIN default (TEXT) is chosen at run-time.
- DISCARD (FILE = *'file_name'* [; TYPE = *'file_type'*])
- REWIND (FILE = *'file_name'*)
- CLOSE (FILE = *'file_name'*)

- GOEOF (FILE = *'file_name'*)
- HIDEOUT (FILE = *'file_name'* [; APPEND = *'append_mode'*])
  append_mode must be either YES or NO. If omitted, the MERLIN default (NO) is chosen at run-time.
- STOP [(EPILOG = *'epilog_mode'*)]
  epilog_mode must be either YES or NO. If omitted, the MERLIN default (YES) is chosen at run-time.
- RETURN [(EPILOG = *'epilog_mode'*)]
  epilog_mode must be either YES or NO. If omitted, the MERLIN default (YES) is chosen at run-time.
- ALIAS (NAME = *'alias_name'* ; COMMAND = *'alias_command'*)
  alias_name is any valid MERLIN alias name.
- UNALIAS (NAME = *'alias_name'*)
- EPILOG (COMMAND = *'epilog_command'*)
- TITLE (TITLE = *'short_title'*)
  short_title is a short title, describing the current minimization session.
- PDUMP (FILE = *'file_name'* [; COMMAND = *'panel_command'*])
- PANELON [(COMMAND = *'panel_command'*)]
- PANELOFF [(COMMAND = *'panel_command'*)]
- PSTATUS [(COMMAND = *'panel_command'*)]
- GRADCHECK (MODE = *'gradient_mode'* [; MODE2 = *'gradient_mode*])
  gradient_mode can be ANAL, NUMER, QUAD or FAST. If MODE2 is omitted, the current MERLIN default is used.

8.5. Panel parametric statements

The following MERLIN commands that use the panel mechanism to obtain their input belong to this category: CONGRA, DFP, BFGS, ACCUM, COVARIANCE, ROLL, SIMPLEX, GRAPH, AUTO, MAD, LEVE, TOLMIN, TRUST, PSGRAPH, CONTROL, MEMO, INSPECT, PICK, BACKUP, INIT, DUMP, HISTORY and HESSIAN. Their syntax is

command_name [(*parameter_pair* { ; *parameter_pair* })]

command_name can be any of the panel parametric statements listed above. A *parameter_pair* is used for communication between the MCL program and the panel. There are two kinds of parameter pairs:

- Parameter pairs used to pass a value from the MCL program to the panel. These are of the form

  key = *val*

  where *key* is a keyword specific to the statement being used and *val* is either an expression or a character string depending on the nature of the keyword. For example,

  ```
  PICK ( FILE = 'Previous_Run' ; REC = NR+1 )
  ```

- Parameter pairs used to pass a value from the panel, back to the MCL program. These are of the form

  key ?= *var*

  where *var* is a simple variable or array element. For example,

  ```
  CONGRA ( FCALLS ?= NC )
  ```

Both types of parameter pairs can be used in a single statement. For example,

```
BFGS ( NOC=1000 ; PRINT=0 ; FCALLS?=N ; ITERDONE?=IT )
```

A complete list of the keywords allowed for each statement is given in the MERLIN user manual. Note that each keyword can be specified at most once.

9. Description of the MCL compiler

The MCL compiler translates MCL programs to instructions that are to be given to MERLIN as input. The compiler is written in ANSI Fortran-77, and it is truly portable. The present version of the software has evolved directly from its predecessor MCL-1.0 [6]. A substantial part of the source code has been totally rewritten while large parts of new code were added in order to support the new features. The source code is distributed in a form which is not directly compilable. It comes with an installation program that takes the distributed files as input and produces a standard ANSI Fortran-77 source code. The installation procedure is described in the MERLIN-3.0 [1] user's manual.

During operation the MCL compiler performs two passes. On the first pass, three operations take place on each line of the MCL program, with the exception of comments and blank lines which are skipped:

(i) Lexical analysis, that transforms each line into tokens, i.e. unambiguous syntactic entities that ease the code generation.
(ii) Parsing, that performs syntax checks and arranges for appropriate error messages.
(iii) Assembling, that creates the basic object code, which however may not be executable at this stage.

The second pass takes care of the loops, the jumps and the associated labels, checks the validity of the nesting structures and generates the MERLIN Object Code (MOC), to be executed by the MERLIN environment.

Upon invocation the compiler issues an informative message and expects an input line of the form

keyword = *value* { , *keyword* = *value* }

that specifies the MCL source file to be compiled along with some additional options. The allowed keywords are

- I specifies the MCL source code file. This keyword cannot be omitted since it has no default value.
- B specifies a file where the MOC is disposed at. This file can be later executed by MERLIN using the RUNMCL command. If omitted, B defaults to MOC.
- E specifies a file where error messages are issued. This file contains the incorrect lines of an MCL program, along with a brief explanation of the error. If omitted, E defaults to the standard output.
- BOUNDS which if set equal to TRUE instructs the compiler to generate code that checks for violation of array boundaries. If omitted, BOUNDS defaults to FALSE.
- DEBUG which if set equal to TRUE instructs the compiler to generate code that provides debugging information when run-time errors are encountered. If omitted, DEBUG defaults to FALSE.

Instead of TRUE and FALSE, one may simply use T and F correspondingly. A sample invocation of the MCL compiler is shown below:

```
------------------------------------------------------------------------------
              The Merlin Control Language compiler v. 3.0
          D.G. Papageorgiou, I.E. Lagaris, I.N. Demetropoulos
                    University of Ioannina, GREECE
                      http://nrt.cs.uoi.gr/merlin
------------------------------------------------------------------------------
Enter compilation parameters   - I, B, E, DEBUG, BOUNDS -
I=prog.mcl, B=runit, E=errors, DEBUG=true, BOUNDS=true
```

10. Description of the test run

The test run was performed in a Sun SPARCstation 5 in double precision. It illustrates a simple strategy that fixes all parameters whose partial derivative is below a given threshold (lines 8–16), and minimizes the

objective function with respect to the rest of them (line 24). Variations of this scheme have been successfully used on several occasions [3,4].

Acknowledgements

This work has been partially supported by the Greek Government through the General Secretariat of Research and Technology, under contracts ΠΕΝΕΔ 89ΕΔ53 and ΠΕΝΕΔ 91ΕΔ959.

Appendix A. Intrinsic variables and arrays

The following is a complete list of all intrinsic MERLIN variables and arrays. These can be used in an MCL expression as any ordinary variable or array. It is not possible, however, to change their values through assignment or GET statements. Intrinsic variables and arrays assume their values from the MERLIN run-time environment and should not be declared through VAR statements. Note that N denotes the number of parameters and M the number of squared terms of the MERLIN objective function.

A.1. Intrinsic variables

DIM — The dimensionality of the objective function. It is set by the user when SUBROUTINE MERLIN is called.

TERMS — The number of terms, if the objective function is a sum of squared terms. It is set by the user when SUBROUTINE MERLIN is called.

PRINTOUT — The current MERLIN printout mode. Possible values are

$$\text{PRINTOUT} = \begin{cases} 1 & \text{NOPRINT} \\ 2 & \text{HALFPRINT} \\ 3 & \text{FULLPRINT} \end{cases}$$

DERIVA — The current MERLIN derivative mode. Possible values are

$$\text{DERIVA} = \begin{cases} 1 & \text{ANAL} \\ 2 & \text{NUMER} \\ 3 & \text{QUAD} \\ 4 & \text{FAST} \\ 5 & \text{MIXED} \end{cases}$$

Its value is changed by the MERLIN commands ANAL, NUMER, QUAD, FAST, MIXED, or after a minimization command that used automatic derivatives, terminates.

PROCESS — The current MERLIN processing mode. Possible values are

$$\text{PROCESS} = \begin{cases} 1 & \text{BATCH} \\ 2 & \text{IAF} \end{cases}$$

Its value is changed by the MERLIN commands IAF and BATCH.

BACKUP — The current MERLIN backup mode. Possible values are

$$\text{BACKUP} = \begin{cases} 1 & \text{NOBACK} \\ 2 & \text{LASTBACK} \\ 3 & \text{FULLBACK} \end{cases}$$

Its value is changed by the MERLIN commands BACKUP, FULLBACK, LASTBACK and NOBACK.

JACOMO The current MERLIN Jacobian mode. Possible values are

$$\texttt{JACOMO} = \begin{cases} 1 & \texttt{JANAL} \\ 2 & \texttt{JNUMER} \end{cases}$$

Its value is changed by the MERLIN commands JANAL and JNUMER.

PRECISION The relative machine's precision as estimated at MERLIN startup or as specified the user with the MACHINE_DIGITS configuration directive.

VALUE The current value of the objective function.

FUNMODE The functional form of the objective function. Possible values are

$$\texttt{FUNMODE} = \begin{cases} 0 & \texttt{GENERAL} \\ 1 & \texttt{SOS} \end{cases}$$

Its value is changed by the MERLIN commands GENERAL and SOS.

TCOUNT The total number of calls to the objective function.

PCOUNT Number of calls to the objective function, since the last RESET command was issued.

GTCOUNT The total number of calls to the user supplied routine GRANAL that calculates the gradient vector.

GPCOUNT Number of calls to the user supplied routine GRANAL since the last RESET command was issued.

HTCOUNT The total number of calls to the user supplied routine HANAL that calculates the second derivative matrix.

HPCOUNT Number of calls to the user supplied routine HANAL since the last RESET command was issued.

JTCOUNT The total number of calls, since MERLIN startup, to the user supplied routine JANAL that calculates the Jacobian matrix.

JPCOUNT Number of calls to the user supplied routine JANAL since the last RESET command was issued.

NFLAGS Number of numerical flags.

NCFLAGS Number of character flags.

A.2. Intrinsic arrays

X[1:N] The current values of the minimization parameters

GRAD[1:N] The gradient vector at the current point, calculated using the current derivative mode.

FIX[1:N] The fix status of the minimization parameters. Possible values are

$$\texttt{FIX}[i] = \begin{cases} 0 & \text{Parameter } x_i \text{ is fixed.} \\ 1 & \text{Parameter } x_i \text{ is not fixed.} \end{cases}$$

MARG[1:N] MARG[i] indicates whether a lower and/or upper bound has been set for parameter x_i. Possible values are

$$\texttt{MARG}[i] = \begin{cases} -1 & \text{Only a lower bound has been set.} \\ 0 & \text{Neither an upper nor a lower bound has been set.} \\ 1 & \text{Only an upper bound has been set.} \\ 2 & \text{Both bounds have been set.} \end{cases}$$

L[1:N] When MARG[i] = −1 or MARG[i] = 2, L[i] is the lower bound. Otherwise L[i] is set to a large negative number.

R[1:N] When MARG[i] = 1 or MARG[i] = 2, R[i] is the upper bound. Otherwise R[i] is set to a large positive number.

STEP[1:N] STEP[i] is the Roll search step that corresponds to parameter x_i.

TERM[1:M] When the objective function is a sum of squared terms, TERM[i] is the ith term.

FLAG[1:nf] FLAG[i] is the value of the ith flag. nf is the number of available MERLIN flags.

Appendix B. Technical presentation of MCL

B.1. A note on syntax semantics

In order to make this article self-contained, we give here a brief summary of EBNF [5], along with a few additional conventions which we use.

- Syntactic units are enclosed in brackets ⟨ ⟩.
- 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.
- The existence of at least n and at most m occurrences of ⟨x⟩ is denoted by $\{\ \langle x\rangle\ \}_n^m$. 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.
- One at most occurrence of ⟨x⟩ (an optional item) is denoted by [⟨x⟩].
- Terminal symbols 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.

B.2. Context-free part of the MCL syntax

(1) ⟨MCL_Source⟩ ::= $\{$ ⟨Program_Unit⟩ $\}_1$
(2) ⟨Program_Unit⟩ ::= ⟨Program⟩ | ⟨Subprogram⟩
(3) ⟨Program⟩ ::= 'PROGRAM' EOS { ⟨Var_Declaration⟩ } { ⟨Function_Declaration⟩ } ⟨Block_Of_Lines⟩ 'END' EOS
(4) ⟨Subprogram⟩ ::= ⟨Subprogram_Header⟩ { ⟨Var_Declaration⟩ } { ⟨Function_Declaration⟩ } ⟨Block_Of_Lines⟩ 'END' EOS
(5) ⟨Subprogram_Header⟩ ::= 'SUB' ⟨Subprogram_Name⟩ ['(' ⟨Var_List⟩ ')'] EOS
(6) ⟨Var_Declaration⟩ ::= 'VAR' ⟨Var_List⟩ EOS
(7) ⟨Var_List⟩ ::= ⟨Simple_Variable⟩ | ⟨Simple_Array⟩ { ';' ⟨Simple_Variable⟩ | ⟨Simple_Array⟩ }
(8) ⟨Function_Declaration⟩ ::= 'FUNCTION' ⟨Function_Name⟩ ⟨Argument_List⟩ '=' ⟨MCL_Expression⟩ EOS
(9) ⟨Argument_List⟩ ::= '[' ⟨Simple_Variable⟩ { ',' ⟨Simple_Variable⟩ } ']'
(10) ⟨Block_Of_Lines⟩ ::= { [⟨Statement⟩ | ⟨Label_Definition⟩] EOS }
(11) ⟨Simple_Array⟩ ::= ⟨Simple_Array_Name⟩ '[' ⟨Lower_Bound⟩ ':' ⟨Upper_Bound⟩ { ',' ⟨Lower_Bound⟩ ':' ⟨Upper_Bound⟩ } ']'
(12) ⟨Lower_Bound⟩ ::= [⟨Sign⟩] ⟨Integer⟩
(13) ⟨Upper_Bound⟩ ::= [⟨Sign⟩] ⟨Integer⟩ | '*'
(14) ⟨MCL_Expression⟩ ::= [⟨Sign⟩ | 'NOT'] ⟨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⟩ | ⟨Simple_Variable⟩ | ⟨Function_Reference⟩ | ⟨Simple_Array_Element⟩ | ⟨Intrinsic_Array_Element⟩
(24) ⟨Real⟩ ::= { ⟨Digit⟩ }$_1$ ['.' { ⟨Digit⟩ }$_1$] [('E' | 'D') [⟨Sign⟩] { ⟨Digit⟩ }$_1$]
(25) ⟨Sign⟩ ::= '+' | '-'
(26) ⟨Statement⟩ ::= ⟨Simple_Statement⟩ | ⟨Block_If⟩ | ⟨When⟩ | ⟨Loop_Statement⟩
(27) ⟨Simple_Statement⟩ ::= ⟨Assignment⟩ | ⟨Get⟩ | ⟨Display⟩ | ⟨Move⟩ | ⟨Call⟩ | ⟨Merlin_Control_Statement⟩
(28) ⟨Loop_Statement⟩ ::= 'LOOP' ⟨Loop_Control_Variable⟩ 'FROM' ⟨Starting_Value⟩ 'TO' ⟨Ending_Value⟩ ['BY' ⟨Step_Value⟩] EOS ⟨Block_Of_Lines⟩ ('END' 'LOOP' | 'ENDLOOP')
(29) ⟨Starting_Value⟩ ::= ⟨MCL_Expression⟩
(30) ⟨Ending_Value⟩ ::= ⟨MCL_Expression⟩
(31) ⟨Step_Value⟩ ::= ⟨MCL_Expression⟩
(32) ⟨Loop_Control_Variable⟩ ::= ⟨Simple_Variable⟩
(33) ⟨When⟩ ::= 'WHEN' ⟨MCL_Expression⟩ 'JUST' ⟨Simple_Statement⟩
(34) ⟨Block_If⟩ ::= 'IF' ⟨MCL_Expression⟩ 'THEN' EOS ⟨Block_Of_Lines⟩ ['ELSE' EOS ⟨Block_Of_Lines⟩] ('END' 'IF' | 'ENDIF')
(35) ⟨Assignment⟩ ::= ⟨Simple_Variable⟩ | ⟨Simple_Array_Element⟩ '=' ⟨MCL_Expression⟩
(36) ⟨Get⟩ ::= 'GET' ⟨Get_Item⟩ { ';' ⟨Get_Item⟩ } ['FROM' ⟨Filename⟩]
(37) ⟨Get_Item⟩ ::= ⟨Simple_Variable⟩ | ⟨Simple_Array_Element⟩
(38) ⟨Display⟩ ::= 'DISPLAY' ⟨Display_Item⟩ { ';' ⟨Display_Item⟩ } ['TO' ⟨Filename⟩]
(39) ⟨Display_Item⟩ ::= ⟨MCL_Expression⟩ | ⟨String⟩
(40) ⟨Move⟩ ::= ('MOVE' 'TO' | 'MOVETO') ⟨Label⟩
(41) ⟨Label_Definition⟩ ::= ⟨Label⟩ ':'
(42) ⟨Call⟩ ::= 'CALL' ⟨Subprogram_Name⟩ ['(' ⟨MCL_Expression⟩ { ';' ⟨MCL_Expression⟩ } ')']
(43) ⟨Simple_Array_Element⟩ ::= ⟨Simple_Array_Name⟩ '[' ⟨MCL_Expression⟩ { ',' ⟨MCL_Expression⟩ } ']'
(44) ⟨Function_Reference⟩ ::= (⟨Function_Name⟩ | ⟨Intrinsic_Function_Name⟩) '[' ⟨MCL_Expression⟩ { ',' ⟨MCL_Expression⟩ } ']'
(45) ⟨Intrinsic_Variable⟩ ::= 'BACKUP' | 'DERIVA' | 'DIM' | 'FUNMODE' | 'GPCOUNT' | 'GTCOUNT' | 'HPCOUNT' | 'HTCOUNT' | 'JACOMO' | 'JPCOUNT' | 'JTCOUNT' | 'NCFLAGS' | 'NFLAGS' | 'PCOUNT' | 'PRECISION' | 'PRINTOUT' | 'PROCESS' | 'TCOUNT' | 'TERMS' | 'VALUE'
(46) ⟨Intrinsic_Array_Element⟩ ::= ⟨Intrinsic_Array_Name⟩ '[' ⟨MCL_Expression⟩ ']'
(47) ⟨Intrinsic_Array_Name⟩ ::= 'FIX' | 'FLAG' | 'GRAD' | 'L' | 'MARG' | 'R' | 'STEP' | 'TERM' | 'X'
(48) ⟨Intrinsic_Function_Name⟩ ::= 'ABS' | 'ACOS' | 'ACOSH' | 'ASIN' | 'ASINH' | 'ATAN' | 'ATANH' | 'COS' | 'COSH' | 'EXP' | 'FACT' | 'GRADNORM' | 'GRMS' | 'LOG' | 'LOG10' | 'MAX' | 'MEAN' | 'MIN' | 'MOD' | 'RAN' | 'ROUND' | 'SIN' | 'SINH' | 'SQRT' | 'TAN' | 'TANH' | 'TRUNC'
(49) ⟨Merlin_Control_Statement⟩ ::= ⟨Non_Parametric⟩ | ⟨I_Parametric⟩ | ⟨IV_Parametric⟩ | ⟨IS_Parametric⟩ | ⟨Panel_Parametric⟩ | ⟨General_Parametric⟩

(50) ⟨Non_Parametric⟩ ::= 'ADJUST' | 'ALIASDIS' | 'ANAL' | 'CFLAGDIS' | 'EVALUATE' | 'EXIT' | 'FAST' | 'FINISH' | 'FIXALL' | 'FLAGDIS' | 'FULLBACK' | 'FULLPRINT' | 'GENERAL' | 'GNORM' | 'GRADDIS' | 'HALFPRINT' | 'JANAL' | 'JNUMER' | 'LASTBACK' | 'LIMITS' | 'LOOSALL' | 'MODEDIS' | 'NOBACK' | 'NOEVAL' | 'NOPRINT' | 'NOTARGET' | 'NUMER' | 'PAUSE' | 'QUAD' | 'RESET' | 'REVEAL' | 'SHORTDIS' | 'SOS' | 'STEPALL' | 'STEPDIS' | 'SUBRETURN' | 'TERMDIS' | 'VALDIS'

(51) ⟨I_Parametric⟩ ::= ⟨I_Parametric_Name⟩ '(' ⟨Key⟩ '.' ⟨MCL_Expression⟩ { ';' ⟨Key⟩ '.' ⟨MCL_Expression⟩ } ')'

(52) ⟨I_Parametric_Name⟩ ::= 'CONFIDENCE' | 'FIX' | 'LDEMARGIN' | 'LEFTDEMARGIN' | 'LOOSE' | 'RDEMARGIN' | 'RIGHTDEMARGIN' | 'NONAME'

(53) ⟨IV_Parametric⟩ ::= ⟨IV_Parametric_Name⟩ '(' ⟨Key⟩ '.' ⟨MCL_Expression⟩ '=' ⟨MCL_Expression⟩ { ';' ⟨Key⟩ '.' ⟨MCL_Expression⟩ '=' ⟨MCL_Expression⟩ } ')'

(54) ⟨IV_Parametric_Name⟩ ::= 'FLAG' | 'LEFTMARGIN' | 'LMARGIN' | 'POINT' | 'RIGHTMARGIN' | 'RMARGIN' | 'STEP'

(55) ⟨IS_Parametric⟩ ::= ⟨IS_Parametric_Name⟩ '(' ⟨Key⟩ '.' ⟨MCL_Expression⟩ '=' ⟨String⟩ { ';' ⟨Key⟩ '.' ⟨MCL_Expression⟩ '=' ⟨String⟩ } ')'

(56) ⟨IS_Parametric_Name⟩ ::= 'CFLAG' | 'GODFATHER' | 'MIXED'

(57) ⟨Panel_Parametric⟩ ::= ⟨Panel_Parametric_Name⟩ ['(' ⟨Key_Value_Pair⟩ { ';' ⟨Key_Value_Pair⟩ } ')']

(58) ⟨Key_Value_Pair⟩ ::= ⟨Key⟩ '=' ⟨Value⟩ | ⟨Key⟩ '?=' (⟨Simple_Variable⟩ | ⟨Simple_Array_Element⟩)

(59) ⟨Panel_Parametric_Name⟩ ::= 'ACCUM' | 'AUTO' | 'BACKUP' | 'BFGS' | 'CONGRA' | 'CONTROL' | 'COVARIANCE' | 'DFP' | 'DUMP' | 'GRAPH' | 'HESSIAN' | 'HISTORY' | 'INIT' | 'INSPECT' | 'LEVE' | 'MAD' | 'MEMO' | 'PICK' | 'PSGRAPH' | 'ROLL' | 'SIMPLEX' | 'TOLMIN' | 'TRUST' | ⟨User_Defined_Panel_Name⟩

(60) ⟨User_Defined_Panel_Name⟩ ::= *Any name defined in the panel description file*

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

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

(63) ⟨Value⟩ ::= ⟨MCL_Expression⟩ | ⟨String⟩

(64) ⟨Filename⟩ ::= ⟨String⟩

(65) ⟨String⟩ ::= ' ' { ⟨Letter⟩ | ⟨Digit⟩ | ⟨Special_Character⟩ }$_1$ ' '

(66) ⟨Key⟩ ::= ⟨Identifier⟩

(67) ⟨Label⟩ ::= ⟨Identifier⟩

(68) ⟨Subprogram_Name⟩ ::= ⟨Identifier⟩

(69) ⟨Simple_Variable⟩ ::= ⟨Identifier⟩

(70) ⟨Simple_Array_Name⟩ ::= ⟨Identifier⟩

(71) ⟨Function_Name⟩ ::= ⟨Identifier⟩

(72) ⟨Identifier⟩ ::= ⟨Letter⟩ {'_' | ⟨Letter⟩ | ⟨Digit⟩ }

(73) ⟨Letter⟩ ::= 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'

(74) ⟨Digit⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Table B.1
Keywords and allowed values for ⟨General_Parametric⟩ statements

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

(75) ⟨Special_Char⟩ ::= ' ' | '!' | '"' | '#' | '$' | '%' | '&' | '(' | ')' | '*' | '+' | ',' | '-' | '.' | '/' | '@' ':' | ';' | '<' | '=' | '>' | '?' | '[' | '\' | ']' | '^' | '_' | '{' | '|' | '}' '~' | '`'

B.3. Context-sensitive part of the MCL syntax

The following rules complement the syntax definition of MCL.

(1) Keywords for ⟨General_Parametric⟩ statements are shown in Table B.1.
(2) Keywords for ⟨IV_Parametric⟩ statements are shown in Table B.2.
(3) Keywords for ⟨IS_Parametric⟩ statements are shown in Table B.3.
(4) Keywords for ⟨I_Parametric⟩ statements are shown in Table B.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 continued 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) Identifiers and terminal symbols are case insensitive.
(9) The identifiers JUST, TO, BY, THEN, and FROM are reserved words and cannot be used as symbolic names.
(10) All operations are performed from left to right.
(11) A proper reference to a function or array element should contain all of the declared arguments.
(12) A ⟨Block_Of_Lines⟩ must not contain a label which is referenced from outside the block.

Table B.2
Keywords and corresponding values for ⟨IV_Parametric⟩ statements

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

Table B.3
Keywords and corresponding values for ⟨IS_Parametric⟩ statements

⟨IS_Parametric_Name⟩	⟨Key⟩
CFLAG	X
GODFATHER	X
MIXED	X

Table B.4
Keywords for ⟨I_Parametric⟩ statements

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

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

References

[1] D.G. Papageorgiou, I.N. Demetropoulos, I.E. Lagaris, MERLIN-3.0. A multidimensional optimization environment, Comput. Phys. Commun. 109 (1998) 227–249, preceding article.
[2] D.G. Papageorgiou, I.N. Demetropoulos, I.E. Lagaris, P.T. Papadimitriou, How many conformers of the 1,2,3-Propanetriol Triacetate are present in gas phase and in aqueous solution?, Tetrahedron 52 (1996) 677–686.
[3] D.A. Karras, I.E. Lagaris, A novel neural network training technique based on a multi algorithm constrained optimization strategy, Preprint 14-96 (Dep. Computer Science, Univ. Ioannina, 1996).
[4] A. Likas, D.A. Karras, I.E. Lagaris, Neural network training and simulation using a multidimensional optimization system, Int. J. Comput. Math., to appear.
[5] J.P. Tremblay, P.G. Sorenson, Theory and Practice of Compiler Writing (McGraw-Hill, New York, 1985).
[6] C.S. Chassapis, D.G. Papageorgiou, I.E. Lagaris, MCL – Optimization oriented programming language, Comput. Phys. Commun. 52 (1989) 223–239.

TEST RUN INPUT

```
program
var i; sml; bfgs_calls; nfix; max_calls

sml = 1.e-4           % Gradient threshlod.
bfgs_calls = 1000     % Number of BFGS calls.
max_calls = 10000     % Max. calls to spend.

again:
   loosall
   nfix = 0
   loop i from 1 to dim
      if abs[grad[i]] <= sml then
         fix (x.i)
         nfix = nfix+1
      end if
   end loop

   if nfix == dim then
      display 'Gradient is below the threshold...'
      loosall
      finish
   end if

   bfgs (noc=bfgs_calls)
when pcount < max_calls just move to again
display 'We probably failed...'

end
```

TEST RUN OUTPUT

```
MCL-3.0
$PUSHC
   10
$ALLOC
$POPMB
$PUSHA
   2
$PUSHC
     1.00000000000000D-04
$POPCONT
$PUSHA
   3
$PUSHC
    1000.00000000000
$POPCONT
$PUSHA
   5
$PUSHC
    10000.0000000000
$POPCONT
$CONTINUE
$PUSHA
   4
$PUSHC
   0.
$POPCONT
$PUSHC
     1.0000000000000
$POP
6
$DIM
$POP
7
$PUSHC
1
$POP
8
$PUSH
7
$PUSH
6
$-
$PUSH
8
$+
$PUSH
8
$/
$TRUNC
$PUSHC
0
$MAX
 2
$POP
10
```

```
$PUSH
6
$POP
1
$PUSHC
1
$POP
9
$PUSH
9
$PUSH
10
$>
$TESTMOVE
   41
$PUSH
1
$GRAD
$ABS
$PUSH
2
$<=
$NOT
$TESTMOVE
   14
$PUSH
1
$PUSHC
1
FIX
$PUSHA
   4
$PUSH
4
$PUSHC
     1.0000000000000
$+
$POPCONT
$CONTINUE
$PUSH
1
$PUSH
8
$+
$POP
1
$PUSHC
1
$PUSH
9
$+
$POP
9
$MOVE
   -45
$CONTINUE
$PUSH
```

```
4
$DIM
$=
$NOT
$TESTMOVE
   9
$NOTE
 34
Gradient is below the threshold...
.STANDARD
$FLUSH
.STANDARD
LOOSALL
$FINISH
$CONTINUE
$PUSH
3
BFGS
  1  1
NOC
I
$PCOUNT
$PUSH
5
$<
$NOT
$TESTMOVE
   3
$MOVE
   -120
$NOTE
 21
We probably failed...
.STANDARD
$FLUSH
.STANDARD
$FINISH
```