M-187-REV-2

MPPL Reference Manual

P. F. Dubois, Z. C . Motteler, P. A. Willmann, R. A. Allsman, C. M. Benedetti, J. A. Crotinger, D. S. Kershaw, A. B. Langdon, A. C. Springer, J. Takemoto, L. Taylor, S. H. Taylor, S. S. Wilson

July 1, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

DISCLAIMER

This work was performed under the auspices of the U. S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

MPPL Reference Manual

Paul F. Dubois
Zane C. Motteler
Peter A. Willmann
Roberta A. Allsman
Cathleen M. Benedetti
James A. Crotinger
David S. Kershaw
A. Bruce Langdon
Allan C. Springer
Janet Takemoto
Lee Taylor
Susan Hockett Taylor
Sharon S. Wilson

July 1, 2002

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Contents

Chapter 1

MPPL Reference Manual

1.1 A More Productive Programming Language

1.1.1 MPPL is a Fortran Preprocessor

MPPL ("More Productive Programming Language") allows programmers to write in a language that is more convenient and powerful than Fortran 77. MPPL then transforms statements written in the MPPL language into standard Fortran 77. This language is essentially an extension to Fortran 77 that provides free-form input and many structured constructs such as "while" and "for" loops. MPPL's macro preprocessor and file-inclusion facility encourage the creation of structured, easy-to-read programs that contain fewer labels. MPPL provides a more productive programming environment for Fortran 77 users on the Unix, Linux, AIX, IRIX, Solairs, HP-UX, Tru64 operating systems.

MPPL can be used independently as well as with Basis.

1.1.2 MPPL's Three Stages

During execution of MPPL, data flows through three ordered steps, or levels. The first level is the token processor; it reads the user's source code and divides it into "tokens", such as names, quoted strings, and punctuation marks. The second level is a macro preprocessor; it takes alphanumeric tokens that the user has defined as macros and replaces them with appropriate text. The third level is the statement processor; it reads tokens after they have been processed by the macro processor. Then the statement processor forms Fortran 77 output text, translating some higher-level programming constructs as it does so.

For most applications, a detailed understanding of the operation of MPPL is not required. The MPPL language is nearly upward-compatible with Fortran 77. The

higher-level programming constructs may be added to existing programs, or not, as the user chooses. MPPL does allow complicated macro definitions, but, the basic usage is very simple:

```
define macroname expansion
```

causes subsequent appearances of the symbol `macroname` to be replaced by the rest of the line on which the define statement occurs.

The user can supply macro arguments like a function call, with the arguments in parenthesis and delimited by commas. The arguments are inserted into the expansion of the macro wherever the definition has a dollar sign followed by an argument number. Thus, an input of

```
define Pop   $1 = $1 - $2
Pop(k,n)
```

yields the statement

```
k = k - n
```

The macro processing facilities are similar to the Unix macro processor m4. The higher-level language facilities are inspired by the C language and the Unix utility Ratfor.

1.1.3 Read the Sample Programs First

Most users will find it suffices to read the next section to learn how to execute MPPL, and then read the MPPL program examples in section 1.7, referring as necessary to the syntax summaries in the Appendix. A more thorough understanding of the MPPL program can be postponed to the day when MPPL does something unexpected.

1.2 Execution

1.2.1 Availability

MPPL is available as `/usr/apps/basis/bin/mppl` at the Secure Computing Facility, the Open Computing Facility, and the A division networks. See http://basis.llnl.gov

1.2.2 Specifying Input and Output Files

To execute MPPL, the user specifies the files to be processed:

```
mppl file1 file2 ... filen
```

The names of the files that MPPL is to translate are delimited by spaces. Output is written to standard out.

A typical mppl-compile-load sequence is:

```
mppl mymacros mysource.m > myout.f
f77  myout.f -o xec
```

If MPPL is executed without a list of files it reads from standard input allowing it to be used as a filter.

Many mistakes in syntax will be caught by MPPL, such as missing `"endif"` statements, but compilation mistakes are possible since MPPL does not check all Fortran syntax.

1.2.3 Specifying Options

Options are entered first on the command line. Options and filenames may not be interspersed. If no files are given, or if a '' alone is given, MPPL reads from stdin. All output is written to stdout, and error output to stderr.

`-`*N* Where *N* is a 1-5 digit integer, specifies the beginning value for MPPL-generated statement labels. The value should be chosen to prevent duplication of existing labels in your code. MPPL restarts the sequence in each subroutine. The default value of *N* is 23000.

`-b` Turn off the output of blank lines and comments. The default is to pass blank and comment lines to the output.

`-c` *string* Set the column 1 comment character to be any of the characters in *string* (up to three characters may be specified). The default value of this option is `cC*`.

`-C`*compiler* Specify the compiler to be used on the MPPL output. This sets the macro `COMPILER` to have the value *compiler*.

`-d` Convert literal character constants enclosed with quotation (") characters to Fortran 77 standard constants using the apostrophe character (') for quoting.

`-D`*name*`[=`*def*`]` Define the macro *name* to have the value *def*, as if MPPL had read the statement

`define name def`

This option may be repeated. Careful quoting is required to embed blanks into *def*:

```
-Dname="'this is a string'"
```

is typical.

`-f` Set free-form input. This disables the usual column 1 comment convention and the column 6 continuation convention. MPPL # comments may still be used in any column. If you only want to disable the column 6 continuation convention, specify a `-ccC*` (or similar `-c` option) after the `-f` option.

`-i` *N* Set the `IntegerSize` to *N*. Legal values are 2, 4, or 8, and imply that an *integer* variable without *kind-selector*, or literal integer constant will be stored in at least 2, 4, or 8 bytes, respectively.

`-I` *directory* Insert *directory* into the search path for include files. Usage is similar to the UNIX C preprocessor. For instance, the options

```
-I.. -I/usr/local/vbasis/pkg
```

tells MPPL to search the parent directory and `/usr/local/vbasis/pkg` for include files, in addition to the current directory. The current directory is always searched first.

`-l` (ell, for "long") Set the length limit for output lines to 80 instead of the standard 72 columns. This limit does not apply to comment lines.

`-m` Prevent MPPL from activating the Basis definitions. Non-Basis users of MPPL should use this option if problems develop from the Basis definitions.

`-M`*machine* Specify the machine we intend to compile on. This sets the value of the macro `MACHINE` to *machine*, and may affect the definition of other predefined macros.

`-r`*N* Set the `RealSize` to *N*. Legal values are 4, 8, or 16, implying that a *real* variable with no *kind-selector* or literal floating point constant of the form `0.0e0` will be stored in an element of at least 4, 8, or 16 bytes, respectively.

`-t` *Sys* Set the intrinsic macro `SYSTEM` to *Sys*, and set the value of other intrinsic macros to the default values for the system named. This allows you to "cross-compile" source for a Fortran compiler on another system. This option sets the macros `MACHINE`, `COMPILER`, `TYPE`, `CHAR_PER_WORD`, `LOCS_PER_WORD`, and `WORDSIZE` to the defaults for the target system. Use the `-C`, `-D`, or `-M` options to over-ride these defaults as required.

`-u` Provide "case insensitivity" for macro names. Either all upper or all lower case (not a mixture) may be used to invoke a macro. This option is required if Fortran keywords in your source code are in upper case.

`-v` Turn on verbose output. Note each input file as it is processed.

`-w` Turn on extra warning messages. In particular, warn if *Size* requests cannot be satisfied in the given target compiler. E.g., if the compiler has no 16 byte wide floating point type, then a request for a *Size16* real object will be mapped into *Size8*, and, if the flag was given, a warning message will be printed to *stderr*.

`--langf77` Convert mppl language macros into Fortran 77. (default)

`--langf90` Convert mppl language macros into Fortran 90. The output will be free source form.

`--nolang` Do not convert mppl language macros.

`--nonumeric` do not convert numbers from f90 format (1.0_Size8), do not process `-r8` or `-r4` macros ($1.0e00$ will not be converted to $1.0d00$) and do not read mppl.std which define integer, real and other related macros.

`--macro` Expand macros. This is the default behavior.

`--nomacro` Do not expand macros.

`--pretty` Pretty print i.e. indent lines. Each level of indention uses the `continuation-indention` value. This is the default.

`--nopretty` use existing white space.

`--relationalf77` convert conditions to use f77 relations operators (.eq., .ne., ...) This is the default behavior.

`--relationalf90` convert conditions to use f90 relations operators (==, /=, ...)

`--honour-new-lines --honor-new-lines -hnl` preserves existing line breaks with `--pretty` option.

`--continuation-indention`*n* `-ci`*n* The width to indent blocks and continued lines. Defaults to 3.

`--comment-indentation`*n* `-comi` The column to start embedded comments (comments using the # character). This is only valid with `--langf90`

The -m option, which must occur before the name of the first input file, prevents MPPL from activating the Basis definitions. Non-Basis users of MPPL should use this option if they find any problems develop from this change. Chances are pretty good that this is not really necessary, since if one of your own definitions collides with the built-in one it will replace it. To see the Basis definitions, run MPPL interactively and enter Dumpdef. Each pair of lines printed are a keyword and its definition. The keywords are:

```
CHAR_PER_WORD COMPILER DEFAULT DONE DYNAM Dumpdef Dynamic ERR FALSE
Filedes Filename GENERATE LOCS_PER_WORD MACHINE Module NO NOTSET
Number_of_Database_Words OK Pi Point Prolog
Quote SITE SLEEPING STDERR STDIN STDOUT SYSTEM TRUE TYPE
UP Use VARNAME WORDSIZE YES
_integer _real _complex _logical _character Ch _Ch _double _Filename
_Filedes  _Varname
SS_WIDTH SS_N SS_TC SS_PTR SS_NAML SS_NS SS_N1 SS_M1 SS_I1
```

1.3 Token Processing

The first internal operation that MPPL performs is the collection of data units, or "tokens". Tokens, or strings of characters, are collected one a time. Some are passed directly to MPPL's output or "translation", and some are checked to see if they require expansion.

1.3.1 Token Descriptions

Alphanumeric An alphanumeric token is any sequence of letters and digits that begins with a letter. The underscore character (_) is treated as a letter.

Digits Tokens can be any one or more digits, 0–9.

Real Numbers Tokens can be Digits followed by a decimal point and exponent.

White Space Tokens can be any sequence of blanks and/or tabs.

Quoted String Tokens can be made up of Hollerith constants or Fortran strings in either single (') or double quotes ("). The same type of quote mark can be used inside a quoted string if the marks are doubled. Or the opposite type of quote may appear.

Comment Everything between a pound sign (#) or an exclamation point (!) and the end of the physical line is a comment. MPPL changes the first character to a lowercase c and writes the token to the output IMMEDIATELY. A new token is then collected. This means that a special method must be used to include comment lines in macro definitions. Refer to the description of the Immediate macro in "Macro Processing" below.

Logical Operators `.eq.`.

Multiple Character Operators exponentiation (**) and concatentation (//).

Any Other Single Character For example, a decimal point is a token. Note, however, that MPPL ignores (and discards) the backslash (\) and collects another token. If the last non-whitespace token on a line is a backslash, MPPL continues the line. The backslash is useful for separating units that must be interpreted separately, but which the user wants adjacent in the output.

"Newline" The invisible "return" character at the physical end of a line of input text is recognized as a token we call "newline". In two cases, however, MPPL discards newline so that two or more physical lines can become one logical line: the column-6 continuation (the Fortran continuation convention) and assumed continuation.

In the Fortran continuation, the newline token and the first six characters of the next line are discarded if the next physical line begins with five blanks followed by a non-blank character.

Assumed continuation occurs when the last non-whitespace token on a line is `+, -, *, (, comma, &, |, ~, >, < , = or \`. The user may conveniently continue a long quoted string by adding a backslash to a concatenate operator (//), for example:

```
x = "This is a long string"//\
"divided into two parts"
```

Note that MPPL does not treat the forward slash (division) character as an obvious continuation because the forward slash is the final character in DATA statements.

1.3.2 Processing Traditional Comments

MPPL recognizes `c, C, or *` in column 1 of a physical input line as a standard comment line, and writes the entire line immediately to the compiler-ready output. The list of characters that signal comment lines may be altered by means of the `-c` ("minus c") option described above in "Specifying Options."

1.3.3 Free-Form Input

MPPL ignores positioning of statements on a line except for the column-6 continuation convention and the `c, C, or *` in column 1, the comment-line indicator.

1.4 Macro Processing

1.4.1 Basic Features of the Macro Processor

The second internal operation that MPPL performs is to replace the alphanumeric tokens the user has defined as macros with the appropriate text. The macro preprocessor collects any macro arguments, performs macro expansion and translation, generates labels, and then passes translated text to the statement processor.

MPPL macros have the following features:

- Recursivity (a macro can call itself).
- Easy-to-read, functional syntax resembles Fortran.
- Built-in conditional statement.

The built-in macros MPPL has are:

```
define(name,translation)
define name translation
Undefine([name])
ifdef([a],b,c)
ifelse(a,b,c,d)
Errprint(message)
Infoprint(message)
Dumpdef([macroname])
Immediate(argument)
Evaluate(argument)
Remark(message)
Setsuppress(name,char)
include filename
Module
Prolog
SYSTEM
```

The MPPL define macro lets users define their own macros. Macros have many uses; they can:

- Give symbolic names to constants, so global changes need be made in only one place.
- Conditionally compile blocks of code.
- Abbreviate or customize the language of frequently used blocks of coding where a subroutine call is not desired.
- Improve readability of the code to make its structure and purpose more obvious.

1.4.2 Macro Names

A macro name can be a string of alphanumeric characters (upper case and lower case letters, digits, and the underscore character) of any length. Note that the macro processor is sensitive to case. N and n are recognized as different names. The `-u` command line option can override this behavior.

1.4.3 Argument Collection

If a macro has arguments, the macro name is followed by a left parenthesis. Arguments are separated by commas and the argument or argument list is terminated with a right parenthesis. Commas within the second or deeper levels of parentheses, or inside square brackets, are ignored. Each argument in turn is collected, and each alphanumeric token is scanned to see if it is a macro. In the following example, the define macro has just two arguments, `"Jack"` and `"Jill(went,up,hill)"`:

```
define(Jack,Jill(went,up,hill))
```

If a macro name has been specified in a `Setsuppress` macro, then argument collection is supressed.

1.4.4 Macro Expansion

Because macro names are alphanumeric tokens (as defined above), every alphanumeric token must be checked. If a token is a macro name, its arguments (if any) are collected, and the expansion of the macro is "pushed back" onto the input file to be rescanned for tokens as described earlier in "Token Processing."

Square brackets are most often used around the arguments to macros. Macro expansion can be delayed by placing the macro name in one or more pairs of square brackets. Each time brackets are encountered, the outside pair is stripped off. For example:

```
define N 100
[N] = N
```

translates to

```
N = 100
```

In a second example, in line 2 below, the N is expanded to 12 when arguments are collected, so the first argument does equal the second. In line 3, the first argument is N, and the second argument is 12.

```
define(N,12)
ifelse(N,12,true,false)          = true
ifelse([N],12,true,false)          = false
```

1.4.5 Macro Translation

When a macro is invoked in the code, it is translated using information from the macro definition. The following substitutions are made:

- Argument substitution (`$n`).
- Replacement of `$*`.
- Replacement of `$-`
- Label generation (`@n`).

Argument Substitution

Any dollar sign followed by a digit 1–9 in the argument list in the define statement is replaced by the corresponding macro argument: `$1` is the first argument, `$2` the second, etc. `$0` is the name of the macro being expanded.

A dollar sign followed by an asterisk or a minus sign, is treated as explained below. A dollar sign followed by another dollar sign results in the insertion of a single dollar sign into the expansion text. A dollar sign followed by any other character results in the insertion of that other character into the expansion text.

```
define distance sqrt(($1-$3)**2 + ($2-$4)**2)
w = distance(x1,y1,x2,y2)
```

expands to

```
w = sqrt((x1-x2)**2 + (y1-y2)**2)
```

Replacement of $*

The complete argument list, separated by commas, is generated. Thus, if we define Jill as

```
define Jill hill($*) - $1
```

then the macro statement in the code

```
Jill(up,down)
```

is translated as

```
hill(up,down) - up
```

Replacement of $-

The argument list minus the first argument is generated. This can be used to define macros with an arbitrary number of arguments that process the first argument and then call themselves recursively to process the remaining arguments. For example:

```
define Product $1 REST($-)
define REST ifelse($1,,,[* $1 REST($-) ])
w = Product(x,y,z)
q = Product(x)
```

which expands to

```
w = x * y * z
q = x
```

The `ifelse` macro is explained below; the result is simply to terminate the recursion when there are no more arguments left. This is a hard example, but we present it because of the usefulness of the idea.

Label Generation

The combination of an at sign (`@`) followed by a digit 1–9 is replaced by an automatically generated label number. Each occurrence of @n is replaced by the same number within a particular expansion of the macro. The first number assigned is the next number in the automatic label sequence, as described in "Execution: Selecting Options."

In the following example, square brackets protect the second argument of the define macro from token interpretation as it is collected. The expansion of the macro named `Errorif0` is given below. It is good practice to use the brackets. They usually produce the desired results, but in this case, they are not really necessary.

```
   define(Errorif0,[
   if ($1.ne.0) go to @1
   write(6,@2)
@2 format("$1 is zero.")
   return
@1 continue
   ])
   Errorif0(x)
```

expands to :

```
        if (x.ne.0) go to 23000
        write(6,23001)
23001           format("x is zero")
        return
23000           continue
```

When a macro is expanded, quoted strings do not protect any arguments (e.g., `$1, $2`) inside them. But when a quoted string is seen by the token processor, macro names inside will not be recognized by the macro processor.

1.4.6 User-Defined Macros

Users define a macro with the built-in MPPL macro define. The two forms of the define macro are:

```
define macroname expansion
define(name,expansion)
```

In the first form, the next token after the define is taken as the macro name. After skipping over any space following the name, MPPL takes the rest of the line as the expansion. Neither the name nor the expansion is scanned for further macros to expand.

In the second form, a define macro with arguments looks like a Fortran function call. The arguments are in parenthesis separated by commas. This form is treated like a normal macro invocation; the arguments are scanned as they are read. If name is already defined then to redefine it using the second form one must surround name with square brackets so that it is not expanded as it is read.

If name has already been defined, the old definition is forgotten. A macro name can be forgotten altogether with the `Undefine` macro.

Undefine([name])

The Undefine macro deletes the definition of a macro name. Note the required square brackets to prevent the name from expanding before we get a chance to Undefine it!

1.4.7 Built-in Macros

In addition to the define macro, the other predefined macros in MPPL are ifdef, ifelse, Errprint, Dumpdef, Immediate, include, COMPILER, SYSTEM, MACHINE, SITE, TYPE, Prolog, Errprint, Infoprint, and Module. The functions they perform cannot be accomplished with user-defined macros.

In addition, the higher-level constructs in MPPL are actually implemented as built-in macros. For example, there is a macro whose translation is a special non-printable character that is interpreted at the statement level.

The built-in macros are described below.

ifdef Macro

```
ifdef([a],b,c)
```

is replaced by either `b` or `c`, depending on whether `a` was defined or not. It becomes `b` if `a` is a defined macro name, and expands to `c` if `a` was not a defined macro name (provided `c` is given). The name `a` needs to be protected with square brackets. For example,

```
ifdef([DEBUG],call trace("x",x))
```

ifelse Macro

If the first argument is identical to the second, the `ifelse` macro,

```
ifelse(a,b,c,d)
```

is replaced by the third argument. Otherwise, it expands to the fourth argument. The second argument `b` can be of the form `b1|b2` in which case, the equality is satisfied if `a` is identical to `b1` or `b2`. Using the notation above, `ifelse(a,b,c,d)` is read as `"if a = b, then c else d."` In making the comparison, leading and trailing spaces in `a` and `b` are ignored. An example of this macro is

```
define Dim real $1[]ifelse($2,,,($2))
Dim(x)
Dim(y,100)
```

which expands to

```
real x
real y(100)
```

The pair of square brackets in the definition of Dim is used as a token separator, so that `ifelse` will be recognized after the name is substituted for `$1`.

Errprint Macro

The Errprint macro immediately writes the argument to the user's terminal in the form MPPL:message and a bell rings. This message goes to the terminal, not to the output file. The syntax is

```
Errprint(message)
```

Infoprint Macro

The Infoprint macro immediately writes the argument to the user's terminal in the form MPPL:message and a bell rings. This message goes to the terminal, not to the output file. The syntax is

```
Infoprint(message)
```

Dumpdef([macroname])

If Dumpdef has no arguments, all macro definitions are displayed to the terminal. If there are arguments, the definition of each macro name given is written to the terminal. The macro name needs to be protected from expansion during argument collection by square brackets, as shown.

Immediate(argument)

Because the token processor writes comments out immediately, the Immediate macro is the best way to delay writing a comment line until it is wanted. For example,

```
define A_comment Immediate([c  this is a comment])
```

is written out as

```
c  this is a comment
```

when the translation for `A_comment` is rescanned. The argument of the Immediate macro is immediately written directly to the output file as a separate line without further interpretation. Note the square brackets surrounding the text of the above `Immediate`. They are recommended in order to suppress the expansion of any macro name, or MPPL reserved word, that might inadvertently been included in the comment.

Comments beginning with "*", "#", and "!" are discarded from macro text upon expansion.

Remark(argument)

The remark macro is used to insert a comment into the code. A limitation of using `Immediate` to insert comment occurs when switching from generating f77 fixed-form to generating f90 free-form. `Remark` will use the correct comment convention based on the `--langf77` and `langf90` command line options.

For example,

```
define A_comment Remark([ this is a comment])
```

is written out as

```
c  this is a comment
```

when using the `--langf77` option; and,

```
!  this is a comment
```

when using the `--langf90` option.

Evaluate(argument)

Evaluate calculates the value of the integer expression represented by argument and returns the character form of the result. If argument is not an integer expression then Evaluate returns argument itself. Example:

```
define N 22
define(NP1, Evaluate(N+1))
define(NP1S, Evaluate(N + 1.0))
x = NP1
y = NP1S
```

expands to

```
x = 23
y = 22 + 1.0
```

Note that in the expression for `y`, `Evaluate(N+1.0)` resulted in a call to Evaluate with argument `"22 + 1.0"` (since the argument was scanned for macros as it was collected), and since this was not an integer expression, Evaluate returned it verbatim.

include filename

The include macro inserts the contents of filename into the input stream. The statement causes the named file to be read before continuing to read the current input file. The included file may itself contain other include statements, to a depth of five files.

Setsuppress(name,char)

Setsuppress is used to suppress argument collection for a macro when it is followed by a specific character.

```
define RealSize  Size4
define(real,\
[ifelse(RealSize,Size4,[[real]([$*])],[[dble]([$*])])]\
)

real(b)
real*8 foo
Setsuppress([real],[*])
real*8 foo
```

expands to

```
real(b)
real()*8 foo

real*8 foo
```

The `Setsuppress` macro prevents the `real` macro from being expanded when used in the `real*8` context.

CHAR_PER_WORD

CHAR_PER_WORD evaluates to the number of characters per machine word. Present machines have either 4 or 8 characters per word.

COMPILER

COMPILER evaluates to the name of the Fortran compiler we are planning to use.

LOCS_PER_WORD

LOCS_PER_WORD evaluates to the number of locations per machine word. Present machines have either 4 or 1 locations per word.

MACHINE

MACHINE evaluates to the name of the machine we are planning to use.

Module

Module evaluates to the name of the current subroutine, function or program module. It evaluates to ? if between modules or if in a main program which does not contain a program statement.

Prolog

After each subroutine, function, or program statement, MPPL adds a line containing the statement Prolog. Prolog is predefined to be simply a comment. The user may redefine Prolog in order to include certain statements in every subroutine and function, such as:

```
define Prolog implicit integer(a-z)
```

SYSTEM

SYSTEM evaluates to the name of the operating system on which MPPL is being run. Currently available systems include AXP,LINUX,LINUXA,HP700,SGI,IRIX64,SOL

WORDSIZE

WORDSIZE evaluates to the length of a word in bits. Currently available wordsizes are 32 and 64.

1.4.8 Error Messages

MPPL error messages are written both to the terminal and to the output file. Where possible, MPPL tries to continue processing after an error (e.g., an `endif` statement with no matching `if` statement). MPPL tries to begin again at the next physical line. As is common in such cases, one error may cause several error messages because the first error confuses MPPL.

Errors in the macro processor are often extremely difficult to handle, and many of these errors cause MPPL to halt immediately. Since the higher-level constructs are macros, mistakes involving their keywords can lead to errors that are reported as errors in the macro processor. For example, a missing right parenthesis in a `return` statement

```
return(value
```

leads eventually to an error as MPPL proceeds to eat up text looking for the end to the argument list for return. MPPL tries to help in this case by informing you that it was collecting arguments when the error occurred, and naming the macros involved.

1.5 Statement Processing

In statement processing, the third internal process, MPPL collects Fortran statements and writes them to MPPL's output file in standard form. During this operation, MPPL indents do loops and if-then statements, and continues long lines using the column-6 convention.

Another major part of statement processing is the transformation of the nonstandard constructions listed below into standard Fortran:

Looping Constructs

```
do ; ... ; enddo
do ; ... ; until (condition)
while ; ... ; endwhile
for( initial, condition, reinitial); ... ;  endfor
break (or break n)
next          (or next n)
```

Module Declarations and Function Value Return

```
subroutine, program  or function
return
return(value)
end
```

Conditional and Case Statements

```
if(condition) then;...; else ; ... ; endif
if(condition) return(value)
select(expression) case default endselect
symbols for logical operators: >, <, >=, <=, <> or ~=, = or ==
```

Free-Form Input

```
; is a logical newline
# and ! begin comments
Automatic continuation if line ends in +, -, *,  comma,
  (, &, |, ~ ,=,>,<
```

These extensions to the Fortran language allow the user to write programs with clearer structure and meaning, and to reduce the use of goto statements and labels.

The keywords listed above are macro names that are translated to special non-printable characters recognized by the statement processor. When using these macro names, it is important to be aware of the considerations discussed below.

1.5.1 Cautions on the Use of Keywords

No Spaces in Macro Names

Do not include spaces within the names. Like all macro names, they cannot be separated internally. The statement

```
d o 100 i = 1,n
```

is not recognized as a do statement in MPPL, even though standard fixed-form Fortran allows the space. The user may separate `end do`, `end while`, `end for`, `end if`, and `end select`, however.

Error If Name Out of Context

These macro names cannot be used in other contexts (e.g., a variable named do is incorrect). If misplaced in the input, these macro names cause an error message, usually "Unprintable character or misplaced keyword in output."

How the Statement Processor Sees Keywords

An expression in parentheses that follows one of these macro keywords is macroexpanded during argument collection, and is rescanned in cases where the argument is supplied. For instance, the built-in definition of `if` is not just a special nonprintable character `X`, but rather is `X($1)`. Understanding the way the keywords are seen internally is important, as the next example shows. Given

```
define n x
define x 10
```

then

```
 if ([n]>9) goto 70
```

translates to

```
 if(10.gt.9) goto 70
```

but

```
if([[n]]>9) goto 70
```

translates to

```
if(n.gt.9) goto 70
```

Protected Token Interpretation

The user should protect keywords with square brackets inside macro definitions to prevent early interpretation. For example,

```
define(zero_out,do i=1,n;$1(i)=0.;enddo)
zero_out(x)
zero_out(y)
```

will result in two do loops with the same label. Instead, to obtain the correct result, write the definition as

```
define(zero_out,[do i=1,n;$1(i)=0;enddo])
```

1.5.2 Symbols for Logical Operators

In the `if`, `for`, `while`, and `until` statements you can use symbols for the standard logical operators (e.g., < for .lt., > for `.gt.`). The complete list of acceptable symbol substitutions is given below in "Conditional Statements."

1.5.3 Multiple Statements on a Line

MPPL treats a semicolon (;) as a logical newline only. Note that column-1 conventions only refer to physical lines. Thus, in this example, a `c` that follows a semicolon is not the start of a comment. Also, as shown here, a label is allowed in the middle of a line:

```
x=0;c=0;100 format(i5)
```

Of course, just because you can do something doesn't mean you should.

1.6 Looping Constructs

1.6.1 Do Loops

do i=1,n;...;enddo

The `do-enddo` construct is available in addition to the traditional `do` loop of the form

```
      do 100 i = 1,n
100   continue
```

The user omits do-loop labels (100 in the example above) and MPPL supplies them during creation of compiler-ready output. The user may specify the lowest number with which MPPL begins numbering (the default is 23000; see "Execution Options"). The syntax is

```
do i=1,n
. . .
enddo
```

The numbering sequence restarts at the beginning of each module.

do/enddo

MPPL allows a `do/enddo` loop without a variable, which is a "do forever" construct with the form

```
do
. . .
enddo
```

In this construct, MPPL generates a labeled continue statement on the `do` line, and replaces `enddo` with a `go to` statement transferring back to that label. The user must provide an exit within this loop by means of a `go to` statement, a `return` statement, or a `break` statement. The last three statements are explained later in this section.

do/until

The user may also select the `do/until` construct, which causes the loop to repeat until the condition given is true:

```
do
.
.
.
until(condition)
```

Note that the body of this loop is always executed at least once.

1.6.2 While Loops

A `while` loop allows the user to repeatedly execute a block of statements while the condition remains true (e.g., while an error is too large, or a desired element has not been found in a table). This statement replaces the traditional `do` loop with an `if-test/goto` inside it. The condition is tested at the top of the loop:

```
while(condition)
 . . .
endwhile
```

For Loops

The `for` loop (modified from the for loop in the C language) is a versatile construct that handles many problems not suited to processing by ordinary looping constructs. It is useful for loops in which the changing element is not merely incremented, but rather may be a call to a function, multiple statements, or another nonlinear process. Note the use of commas instead of the semicolons used in C. The second argument, the condition, must always be given. The third argument is optional.

```
for(initial,condition,reinitial)
. . .
endfor
```

MPPL translates the construct as shown below. First, the initial clause is executed, and then the condition evaluated. If the condition is true, the body of the loop is executed. Then the reinitial clause is executed, and the condition reevaluated. The loop terminates when the condition becomes false.

```
   initial
   go to L3
L2 reinitial
L3 if(.not.(condition))go to L1
   . . .
   go to L2
L1 continue
```

The first and third arguments may contain multiple statements, and the first argument can be null, for example,

```
for(,n<10,n=n+1)
  i = i + n
endfor
```

1.6.3 Leaving and Skipping

break

The MPPL `break` statement can be used inside any of the looping constructs discussed above. It is invoked in any one of three forms:

```
break
break(n)
break n
```

where `n` is an integer that specifies the number of loops from which a breakout is desired. The `break` statement translates to a `go to L` statement, where `L` is the supplied label of a continue statement that follows the end of the loop. If $n > 1$, the transfer is to the end of the n'th enclosing loop, e.g.,

```
do i = 1,10
   do j = 1,10
        if(x(   i,j).eq.0)break 2
   enddo
enddo
```

Here, the `break 2` statement causes a transfer out of both loops. If the 2 is omitted, transfer would be just out of the j loop.

The `next` statement can also be used inside any of the looping constructs. It causes the next iteration of the loop to begin.

The slight differences in implementation for each kind of loop are shown below:

Type of Loop	Go to:
Traditional do loop	labeled statement
Label-less do loop	enddo
do/enddo, do/until	do
while/endwhile	while
for/endfor	reinitial

Note that, in each case, the transfer is to the top of the loop. However, the labeled loops go to the label to increment the variable. In traditional loops, where the labeled statement is not continue, the labeled statement is executed, which may be surprising. Note also that the `do/until` loop executes the loop body at least once after the use of a `next` statement.

1.6.4 Module/Return Statements

Modules may begin with standard `program`, `subroutine`, or `function` statements. These three words are MPPL macro names so that MPPL can issue good error messages and so that functions can return a value from a function in a more natural way.

Inside a function, the user may provide an argument to the `return` statement:

```
return(value)
```

MPPL expands this to

```
functionname = (value)
return
```

It is an error to use an argument with the `return` statement inside a program module or subroutine module. In that case, MPPL displays an error message, but continues execution. A statement of the following form is allowed:

```
if(condition) return(value)
```

1.6.5 Conditional Statements

MPPL supports all the standard `if` and `if-then-elseif-else-endif` constructs of Fortran 77. It also adds some extra features to these statements.

Symbol Substitution

In addition to processing Fortran 77 forms of the `if` statement, MPPL allows the user to enter the following symbols for equals, greater than, etc., instead of the traditional notation.

User enters	translation
`>`	`.gt.`
`>=`	`.ge.`
`<`	`.lt.`
`<=`	`.le.`
`~=`	`.ne.`
`<>`	`.ne.`
`~`	`.not.`
`=`	`.eq.`
`==`	`.eq.`

if(condition) enhancement

MPPL also allows placing the last part of an `if(condition)` statement on a new line. For example:

```
if ( ierr > 0 )
   call goof
```

or

```
if ( ierr > 0 )
then
   call goof
endif
```

if(condition) return(value) statement

This special single-statement

```
if (condition) return(value)
```

appears to be a statement of the form

```
if (condition) statement
```

However, `return(value)` translates to two statements. MPPL handles this in a special way in order to translate it correctly to:

```
if(condition) then
  functionname=(value)
  return
endif
```

1.6.6 Case Selection Statement

The syntax for the select macro is:

```
select(expression)
case casenum:
. . .
default:
. . .
endselect
```

where

`select(expression)`

compares an integer expression to the values listed in the case statements that follow, and executes at most one of the cases. The first `case` stated must immediately follow the select statement. An optional `default` section can be executed if the expression fails to match any of the `case`s.

`case casenum:`

labels the beginning of the statements to execute if the select expression matches the case expression casenum. For casenum, the user must insert either an integer, a range (two integers separated by a minus sign), or a comma-delimited list of integers and ranges. The expression must end with a colon. For example:

```
case 7-10,12:
```

Statements may follow on the same line, after the colon. Multiple statements may be separated by a colon, or appear on new lines.

`default:`

labels the beginning of the statements to execute if the select expression fails to match any of the case values.

`endselect`

marks the end of the case list. Here is an example of a complete select/case/default/endselect construction:

```
select(x)
case 0: y = 1
   x = 1
case 1-4: y = 2
case 5,6: y = 3
case 7-10,12:
   y = 4
default: y = 0;x = 0
endselect
```

A `select` statement is translated either into a series of `if` statements or into a computed go to. The latter is more efficient and so is used if there are enough consecutive case values to make it desirable. A few gaps in the sequence will be filled in and the sequence need not start from one. A computed go to is a statement of the type

```
go to (1000,1001,1002, ...) ivar
```

where control goes to label 1000 if ivar = 1, to label 1001 if ivar = 2, etc. While efficient, such statements are opaque, annoying to modify, and have undefined behavior if ivar is out of bounds. The select statement is both clearer and safer.

1.7 Sample Input File Showing Major MPPL Features

```
#LOOPING CONSTRUCTS
#
define N 100
define M  20
   function shoot(j)
c This subroutine shows the six different looping constructs
   real xx(N),y(M),x,y
# there are four kinds of DO loops plus WHILE and FOR loops.
#
# TRADITIONAL LABELED DO LOOP
   do 100 i=1,10
        if(x(i) = 4) then
             break          # same as go to next stmt after 100
```

```
        endif
        if(x(i) = 5 then
             next # same as go to loop label (100)
        endif
100          y(2) = x(i)          # this gets executed on a next
#
# DO LOOPS WITHOUT LABELS
# next gets you to next iteration; break gets you out
# SIMPLE LOOP
#
   do i=1,M
        if(y(i) < 0) break
        if(y(i) >= 10.) next
        y(i) = sqrt(10.-y(i))
   enddo
#
# NESTED LOOPS
#
   do i = 1,M
        do j = 1,N
             if(x.eq.10)then #next iteration of inner loop
                  next
             endif
             if(x.eq.20)then #next iteration of outer loop
                  next(2)
             endif
             xx(j) = 8
             if(y(i) > x(j))then
                  break     # get out of inner loop
             else
                  break(2)  # get out of inner loop
             endif
        enddo j   # end inner loop
                  # ignores anything after enddo
   enddo i            # end outer loop
#
# DO FOREVER
# repeats forever; get out with break, return, or goto.
#
   i=0
   do
        i = i + 1
```

```
        if(i > M) break
        if(x(i) == 32)
             next
        x(i)=1/(x(i)-32)
   enddo
#
# WHILE/ENDWHILE
# does a loop as long as the condition is satisfied
#
   i = N
   while(x(i)-x(i+1) > 1.e-5 & i <> 0)   #&=.and.  <>=.ne.
        i = i - 1
   endwhile
#
# DO/UNTIL
# repeats until the condition is satisfied. Note that unlike
# a while loop, the loop body is always done once
#
   do
        i = i - 1
        if(i = 0 | x(i) <= 0.) break        # | = .or.
   until(x(i)-x(i+1) < 1.e-3)
#
# FOR/ENDFOR
# has three arguments separated by commas:
# a) initialization statements to be executed before the
# loop, b) the condition under which the loop is to be
# executed while true, and c) the reinitialization
# statements to be executed at the start of each loop after
# the first before the condition is tested. The condition,
# argument 2, must be present; other arguments are optional.
#
# The following example is the same as do i=1,N;x(i)=i;enddo
#
   for (i-1, i<=N, i=i+1)
        x(i) = 1
   endfor
#
# FOR loops are good for things DO LOOPS can't do:
# the hard way to find the square root of two is:
#
   for(t = 1.,abs(t**2 - 2.) > 1.e-6, t=(t+2./t)/2.)
```

```
   endfor
#
# FUNCTIONS
# The return statement can have an argument to give the
# returned value.
#
   return(t)
   end
   real function boxo(w,z)
   real w,z,a,b
#
# IF STATEMENTS
# There are two basic kinds of IF; this routine shows some
# of the variations allowed.
#
# IF(CONDITION) THEN ...ENDIF
#
   if( a<b)            #ok if then is on next line
   then
        call odd("this is a string; try it");return(b-a)
   endif
   if(a <> b) then         #if a .ne. b
        x = y
   else if (b > a-1)         #ok if you forget the then here
        x=y/2 +         #statements continued if they end
        golf(tango,         #in +,-,*,comma,=,(,&,|,caret, or
        bravo \         #backslash; backslash is deleted
        -1)
        y="This is a quoted string "" with a quote in it\"
                        #...but not inside strings
   else if("the sky is blue >")then #or put in to be neat
        howdy = 1
   else
        if(a == w) call junko
   endif
#
# IF(CONDITION)STATEMENT/RETURN(VALUE)
# is correct even if it expands to more than one statement.
#
   if(a > b)
        b = b/2
   if( a<> b) return(gas)
```

```
   return(bug)
   end
   program testme
#
# SELECT/CASE/DEFAULT/ENDSELECT
# You can put things after an ENDDO that are ignored.
#
   real x(N)
   do i=1,10
        x(i) = i - 1
   enddo i --end of loop setting initial values for x
#
# You can have multiple statements by separating them
# with semicolons, even in the arguments of a FOR statement.
#
   i0 = 0 ; j0 = M
   for( i = i0; j = j0 , j < 9 , i=i+1;j=j-1)
        x(i) = y(j)
        for(k=j, k<i+5 , k=k+1)
             z(k) = y(j) + x(i)
        endfor
   endfor
#
# SELECT allows you to test an integer variable against
# different cases.
#
   select(j)
   case 5: y=5          #if j is 5 do these two statements
        z = 4                ! exclamation points are also comments
   case 6: y=6          ! if j is 6 do this one
   case 7,8,10:          ! statements can follow on next line
        y=8;z=4          ! if j is 7, 8, or 10
   case 11-20,9: y=9         #if j is between 11 and 20
                        #inclusive or is 9
   default:
        y=0                  #do if j is none of the above
   endselect
#
   call exit
   end
```

1.8 Examples of Advanced MPPL Macro Usage

The following examples show how to use the macro processor. Most MPPL users will use macros only in the simple sense of using a name as a symbol for a constant value, as in

```
define pi 3.14159
```

and as in the first example below, to enable the specification of variables to be confined to just one place. Another common problem is conditional compilation, which we cover in the second example. The third and fourth examples show a user inventing language extensions.

1.8.1 Specifying a common block

This example shows how to specify a common block in one place, then use it as needed in subroutines. We include an Immediate comment so that in the expanded source the common block is marked with a comment.

```
define(Distribution_parameters,[
Immediate([c Distribution variables])
     integer alpha,sigma,beta
     common /c1/ alpha,sigma,beta
])

     subroutine x
Distribution_parameters
     . . .
     end
     subroutine y
Distribution_parameters
     . . .
     end
```

1.8.2 Conditional compilation

Depending on whether or not the first define(DEBUG,) line is present or not, the write statement is or is not compiled.

```
define(DEBUG,)
ifdef([DEBUG],[
     write(6,100) x,y,z
])
```

1.8.3 Vector operations

The following example shows a macro that expands to a do-loop that adds the last two arrays together and stores the result in the first array. The fourth argument is the length of the arrays. I do not advocate this kind of programming but it can be done.

```
define (Vector_add,[do i=1,$4 ; $1(i)=$2(i)+$3(i) ; enddo])
Vector_add(a,b,c,n)
Vector_add(d,e,f,n)
```

1.8.4 Alphanumeric Labels

Some people enjoy the LRLTRAN feature of using names as labels. This can be done with MPPL as long as we use a macro to change the names into statement labels. The Label macro is recursive so that several labels can be specified at once. The definition for Label can be read: if Label is called with an empty argument list, do nothing. Otherwise, define the first argument (`$1`) to be a macro name standing for the next available label (`@1`) and then apply Label to the rest of the arguments (`$`). Thus Label chews its arguments from left to right. Note that the `$1` is surrounded by square brackets in case this name was used as a label already in another subroutine.

```
define Label ifelse($1,,,[define([$1],@1)Label($-)])

      function boom(x)
c return 1, 0, -1 depending on the sign of x
      integer boom
      real x
Label(Negative,Positive) #must appear before first use of names
      if( x < 0) go to Negative
      if( x > 0) go to Positive
      return(0)
Positive  return(1)
Negative  return(-1)
      end
```

Conversion to MPPL

Those users who want to convert a code to precompile with MPPL instead of Precomp, but who do not plan to utilize the rest of Basis will have to make simple changes in their cliches. If a cliche is called `Abc` change the statement `cliche Abc` to `define{[UseAbc],[` and change the statement `endcliche` to `])\`. In the source every `use Abc` must be replaced by `Use(Abc)`. Basis does not support `dif` and

`.if` directives. Replace them with combinations of the Basis macros `ifelse` and `define`.

1.9 Migration to Fortran 90 syntax

In the years since MPPL was first written, the Fortran standard has advanced to where the language processing features of MPPL can be replace by Fortran 90 syntax.

1.9.1 Command Line Options

A typical mppl-compile-load sequence is:

```
mppl mymacros mysource.m > myout.f
f77  myout.f -o xec
```

Often, the input file mysource.m and the output file myout.f are significantly different. All macros and real numbers have been processed and the output has been indented to a consistent form.

A line similar to

```
mppl --langf90 --nomacro --nonumeric --nopretty -l78
    mysource.m > mysource1.m
```

can be used to convert only the language macros.

The `--nolang` command line option can then be used to prevent the future expansion of MPPL language constructs.

```
mppl --nolang mymacros mysource1.m > myout.f90
f90  myout.f90 -o xec
```

1.9.2 Statement Processing

The `--langf90` option will produce free-form output. All comments start with an exclamation point (!). Embeded comments will replace the # with ! without creating a new line. Continued lines end with an ampersand (&).

By default, f77 compatiable relation operators are used. `--relational90` can be used to generate symbols <, <=, ==, / =, >, and >=

1.9.3 Macros

`include filename` is process by mppl. *filename* is read by mppl and processed. `include "filename"` is process by f90. *filename* is ignored by mppl.

The `Remark` macro should be use instead of `Immediate` to insert comments from macros. This will use the correct comment convention.

1.9.4 Loop Constructs

Indexed Loops

```
do i=1,n
...
enddo
```

This loop requires no conversion since it is valid f90.

do/until

```
do
  ...
until(condition)
```

`do/until` requires an explicit `exit`.

```
do
  ...
  if (condition) exit
enddo
```

While Loops

```
while(condition)
 . . .
endwhile
```

The `endwhile` is replaced with `enddo`.

```
while(condition)
 . . .
enddo
```

For Loops

```
for(initial,condition,reinitial)
  . . .
endfor
```

The `initial`, `condition` and `reintial` clauses are moved to the appropriate parts of a `while` loop.

```
initial
do while (condition)
  . . .
  reinitial
endfor
```

1.9.5 Leaving and Skipping

`next` and `next` are replace by `cycle` and `exit`.

The `next 2`, syntax is converted to use `goto`'s as with `--langf77`. A motivated user can manually convert this to:

```
outer: do
         do
           ...
           exit outer
         enddo
       enddo outer
```

1.9.6 Case Selection Statement

```
select(expression)
case casenum:
. . .
default:
. . .
endselect
```

`casenum` is enclosed in parenthesis. `default` becomes `case default`.

```
select(expression)
case (casenum)
. . .
case default
. . .
endselect
```