fpp: Fortran preprocessor

March 8, 2024

1 Name

`fpp` — the Fortran language preprocessor for the NAG Fortran compiler.

2 `fpp` command line

fpp [option]... [*input-file* [*output-file*]]

3 Description of `fpp`

fpp is the preprocessor used by the NAG Fortran compiler. It optionally accepts two filenames as arguments: *input-file* and *output-file* are, respectively, the input file read and the output file written by the preprocessor. By default standard input and output are used.

When used via `nagfor`, either because the input source file type was automatically recognised as requiring preprocessing (e.g. `.ff90` files), or the *−fpp* option was used, the macro `__NAG_COMPILER_RELEASE` is automatically defined to be the integer release number (major*10+minor, e.g. `61` for release 6.1), and the macro `__NAG_COMPILER_BUILD` is automatically defined to be the build number (for release 6.1 this will have a value greater than `6100`).

4 `fpp` options

−c_com={*yes*|*no*}
: By default, C style comments are recognized. Turn this off by specifying *−c_com=no*.

−D*name*
: Define the preprocessor variable *name* to be `1` (one). This is the same as if a *−Dname=1* option appeared on the fpp command line, or as if a
`#define` *name* `1`
line appeared in the input file.

−D*name=def*
: Define *name* as if by a `#define` directive. This is the same as if a
`#define` *name def*
line appeared at the beginning of the input file. The *−D* option has lower precedence than the *−U* option. Thus, if the same name is used in both a *−U* option and a *−D* option, the name will be undefined regardless of the order of the options.

−e Accept extended source lines, up to 132 characters long.

−fixed Specifies fixed format input source.

−free Specifies free format input source.

−I*pathname*
: Add *pathname* to the list of directories which are to be searched for `#include` files whose names do not begin with '`/`'. If the `#include` file name is enclosed in double-quotes (`"..."`), it is searched

for first in the directory of the file with the `#include` line; if the file name was enclosed in angle brackets (`<...>`) this directory is not searched. Then, the file is searched for in directories named in *−I* options, and finally in directories from the standard list.

-M Generate a list of makefile dependencies and **write** them to the standard output. This list indicates that the object file which would be generated from the input file depends on the input file as well as the include files referenced.

−macro={*yes*|*no_com*|*no*}
By default, macros are expanded **everywhere**. Turn off macro expansion in comments by specifying *−macro=no_com* and turn off macro expansion all together by specifying *−macro=no*

−P Do not put line numbering directives to the output file. Line numbering directives appear as `#`*line-number file-name*

−U*name*
Remove any initial definition of *name*, where *name* is an fpp variable that is predefined by a particular preprocessor. Here is a partial list of variables that might be predefined, depending upon the architecture of the system:

Operating System: `unix`, `__unix` and `__SVR4`;
Hardware: `sun`, `__sun`, `sparc` and `__sparc`.

−undef Remove initial definitions for all predefined symbols.

−w Suppress warning messages.

−w0 Suppress warning messages.

−Xu Convert upper-case letters to lower-case, except within character-string constants. The default is not to convert upper-case letters to lower-case.

−Xw For fixed source form only, treat blanks as insignificant. The default for fpp is that blanks are significant in both source forms.

−Y *directory*
Use the specified *directory* instead of the standard list of directories when searching for `#include` files.

5 Using fpp

5.1 Source files

fpp operates on both fixed and free form source files. Files with the (non-case-sensitive) extension '`.f`', '`.ff`', '`.for`' or '`.ftn`' are assumed to be fixed form source files. All other files (e.g. those with the extension '`.ff90`') are assumed to be free form source files. These assumptions can be overridden by the *−fixed* and *−free* options. Tab format lines are recognised in fixed form.

A source file may contain **fpp** tokens. An fpp token is similar to a Fortran token, and is one of:

- an fpp directive name;
- a symbolic name or Fortran keyword;
- a literal constant;

- a Fortran comment;
- an fpp comment;
- a special character which may be a blank character, a control character, or a graphic character that is not part of one of the previously listed tokens.

5.2 Output

Output consists of a modified copy of the input plus line numbering directives (unless the $-P$ option is used). A line numbering directive has the form
`#`*line-number file-name*
and these are inserted to indicate the original source line number and filename of the output line that follows.

5.3 Directives

All fpp directives start with the hash character (`#`) as the first character on a line. Blank and tab characters may appear after the initial '`#`' to indent the directive. The directives are divided into the following groups:

- macro definitions;
- inclusion of external files;
- line number control;
- conditional source code selection.

5.4 Macro definition

The `#define` directive is used to define both simple string variables and more complicated macros:

`#define` *name token-string*

This is the definition of an fpp variable. Wherever '*name*' appears in the source lines following the definition, '*token-string*' will be substituted for it.

`#define` *name([argname1[,argname2]...]) token-string*

This is the definition of a function-like macro. Occurrences of the macro '*name*' followed by a comma-separated list of arguments within parentheses are substituted by the token string produced from the macro definition. Every occurrence of an argument name from the macro definition's argument list is substituted by the token sequence of the corresponding macro actual argument.

Note that there must be no space or tab between the macro name and the left parenthesis of the argument list in this directive; otherwise, it will be interpreted as a simple macro definition with the left parenthesis treated as the first character of the replacement *token-string*.

`#undef` *name*

Remove any macro definition for *name*, whether such a definition was produced by a $-D$ option, a `#define` directive or by default. No additional tokens are permitted on the directive line after the name.

The macro NAGFOR is defined by default.

5.5 Including external files

There are two forms of file inclusion:
`#include "`*filename*`"`
and
`#include <`*filename*`>`

Read in the contents of *filename* at this location. The lines read in from the file are processed by fpp as if they were part of the current file.

When the `<`*filename*`>` notation is used, *filename* is only searched for in the standard "include" directories. See the $-I$ and $-Y$ options above for more detail. No additional tokens are permitted in the directive line after the final '`"`' or '`>`'.

5.6 Line number control

`#`*line-number* *[*`"`*filename*`"`*]*

Generate line control information for the next pass of the compiler. The *line-number* must be an unsigned integer literal constant, and specifies the line number of the following line. If `"`*filename*`"` does not appear, the current filename is unchanged.

5.7 Conditional selection of source text

There are three forms of conditional selection of source text:

1. ```
   #if condition_1
     block_1
   #elif condition_2
     block_2
   #else
     block_n
   #endif
   ```

2. ```
   #ifdef name
     block_1
   #elif condition
     block_2
   #else
     block_n
   #endif
   ```

3. ```
   #ifndef name
     block_1
   #elif condition
     block_2
   #else
     block_n
   #endif
   ```

The "#else" and "#elif" parts are optional. There may be more than one "#elif" part. Each condition is an expression consisting of fpp constants, macros and macro functions. Condition expressions are similar to cpp expressions, and may contain any cpp operations and operands with the exception of C long, octal
   ```

and hexadecimal constants. Additionally, fpp will accept and evaluate the Fortran logical operations `.NOT.`, `.AND.`, `.OR.`, `.EQV.`, `.NEQV.`, the relational operators `.GT.`, `.LT.`, `.LE.`, `.GE.`, and the logical constants `.TRUE.` and `.FALSE.`.

6 Preprocessing details

6.1 Scope of macro or variable definitions

The scope of a definition begins from the place of its definition and encloses all the source lines (and source lines from `#include`d files) from that definition line to the end of the current file.

However, it does not affect:

- files included by Fortran `INCLUDE` lines;
- fpp and Fortran comments;
- `IMPLICIT` single letter specifications;
- `FORMAT` statements;
- numeric and character constants.

The scope of the macro effect can be limited by means of the `#undef` directive.

6.2 End of macro definition

A macro definition can be of any length but is only one logical line. These may be split across multiple physical lines by ending each line but the last with the macro continuation character '\' (backslash). The backslash and newline are not part of the replacement text. The macro definition is ended by a newline that is not preceded by a backslash.

For example:

```
#define long_macro_name(x,\
y) x*y
```

6.3 Function-like macro definition

The number of macro call arguments must be the same as the number of arguments in the corresponding macro definition. An error is produced if a macro is used with the wrong number of arguments.

6.4 Cancelling macro definitions

`#undef` *name*

After this directive, '*name*' will not be interpreted by fpp as a macro or variable name. This directive has no effect if '*name*' is not a macro name.

6.5 Conditional source code selection

`#if` *condition*

Condition is a constant expression, as specified below. Subsequent lines up to the first matching `#elif`, `#else` or `#endif` directive appear in the output only if the *condition* is true.

The lines following a `#elif` directive appear in the output only if

- the condition in the matching `#if` directive was false,
- the conditions in all previous matching `#elif` directives were false, and
- the condition in this `#elif` directive is true.

If the condition is true, all subsequent matching `#elif` and `#else` directives are ignored up to the matching `#endif`.

The lines following a `#else` directive appear in the output only if all previous conditions in the construct were false.

The macro function 'defined' can be used in a constant expression; it is true if and only if its argument is a defined macro name.

The following operations are allowed.

- C language operations: `<`, `>`, `==`, `!=`, `>=`, `<=`, `+`, `-`, `/`, `*`, `%`, `<<`, `>>`, `&`, `~`, `|`, `!`, `&&` and `||`. These are interpreted in accordance with C language semantics, for compatibility with cpp.
- Fortran language operations: `.AND.`, `.OR.`, `.NEQV.`, `.XOR.`, `.EQV.`, `.NOT.`, `.GT.`, `.LT.`, `.LE.`, `.GE.`, `.NE.`, `.EQ.` and `**`.
- Fortran logical constants: `.TRUE.` and `.FALSE.`.

Only these items, integer literal constants, and names can be used within a constant expression. Names that are not macro names are treated as if they were '0'. The C operation '`!=`' (not equal) can be used in `#if` or `#elif` directives, but cannot be used in a `#define` directive, where the character '`!`' is interpreted as the start of a Fortran comment.

`#ifdef` ***name***
: Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if the name has been defined, either by a #define directive or by the $-D$ option, and in the absence of an intervening #undef directive. No additional tokens are permitted on the directive line after name.

`#ifndef` ***name***
: Subsequent lines up to the matching #else, #elif, or #endif appear in the output only if name has not been defined, or if its definition has been removed with an #undef directive. No additional tokens are permitted on the directive line after name.

`#elif` ***constant-expression***
: Any number of #elif directives may appear between an #if, #ifdef, or #ifndef directive and a matching #else or #endif directive.

`#else`
: This inverts the sense of the conditional directive otherwise in effect. If the preceding conditional would indicate that lines are to be included, then lines between the `#else` and the matching `#endif` are ignored. If the preceding conditional indicates that lines would be ignored, subsequent lines are included in the output.

#endif
: End a section of lines begun by one of the conditional directives #if, #ifdef, or #ifndef. Each such directive must have a matching #endif.

6.6 Including external files

Files are searched as follows:

for `#include "`*filename*`"`:

- in the directory, in which the processed file has been found;
- in the directories specified by the $-I$ option;
- in the default directory.

for `#include <`*filename*`>`:

- in the directories specified by the $-I$ option;
- in the default directory.

Fpp directives (lines beginning with the `#` character) can be placed anywhere in the source code, in particular immediately before a Fortran continuation line. The only exception is the prohibition of fpp directives within a macro call divided on several lines by means of continuation symbols.

6.7 Comments

Fpp permits comments of two kinds:

1. Fortran language comments. A source line containing one of the characters '`C`', '`c`', '`*`', '`d`' or '`D`' in the first column is considered to be a comment line. Within such lines macro expansions are not performed. The '`!`' character is interpreted as the beginning of a comment extending to the end of the line. The only exception is the case when this symbol occurs within a constant expression in a `#if` or `#elif` directive.
2. Fpp comments enclosed in the '`/*`' and '`*/`' character sequences. These are excluded from the output. Fpp comments can be nested so that for each opening sequence '`/*`' there must be a corresponding closing sequence '`*/`'. Fpp comments are suitable for excluding the compilation of large portions of source instead of commenting every line with Fortran comment symbols. Using "`#if 0 ... #endif`" achieves the same effect without being ridiculous.

6.8 Macro functions

The macro function

`defined`(*name*) or `defined` *name*

expands to `.TRUE.` if *name* is defined as a macro, and to `.FALSE.` otherwise.

6.9 Macro expression

If, during expansion of a macro, the column width of a line exceeds column 72 (for fixed form) or column 132 (for free form), fpp inserts appropriate continuation lines.

In fixed form there are limitations on macro expansion in the label part of the line (columns 1-5):

- a macro call (together with possible arguments) should not extend past column 5;
- a macro call whose name begins with one of the Fortran comment characters is treated as a comment;
- a macro expansion may produce text extending beyond the column 5 position. In such a case a warning will be issued.

In the fixed form when the $-Xw$ option has been specified an ambiguity may appear if a macro call occurs in a statement position and a macro name begins or coincides with a Fortran keyword. For example, in the following text:

```
#define call p(x)   call f(x)
        call p(0)
```

fpp can not determine with certainty how to interpret the 'call p' token sequence. It could be considered as a macro name. The current implementation does the following:

- the longer identifier is chosen (callp in this case);
- from this identifier the longest macro name or keyword is extracted;
- if a macro name has been extracted a macro expansion is performed. If the name begins with some keyword **fpp** outputs an appropriate warning;
- the rest of the identifier is considered as a whole identifier.

In the above example the macro expansion would be performed and the following warning would be output:

warning: possibly incorrect substitution of macro callp

It should be noted that this situation appears only when preprocessing fixed form source code and when the blank character is not being interpreted as a token delimiter. It should be said also that if a macro name coincides with a keyword beginning part, as in the following case:

```
#define INT    INTEGER*8
        INTEGER k
```

then in accordance with the described algorithm, the INTEGER keyword will be found earlier than the INT macro name. Thus, there will be no warning when preprocessing such a macro definition.

7 `fpp` diagnostics

There are three kinds of diagnostic messages:

- warnings. preprocessing of source code is continued and the return value remains to be 0.
- errors. Fpp continues preprocessing but sets the return code to a nonzero value, namely the number of errors.
- fatal error. Fpp cancels preprocessing and returns a nonzero return value.

The messages produced by fpp are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

8 See Also

nagfor(1).