Fortran Preprocessor: Introduction

INCITS/Fortran JoR

December 10, 2024

Contents

1 Introduction

The INCITS/Fortran committee has discussed several times in the Plenary meetings, and the Journal of Requirements (JoR) subgroup has met several times, to lay down formal requirements for the proposed Fortran preprocessor (FPP) for Fortran 202y (presumably 2028).

This paper lists the high-level requirements and how JoR came to them.

The first section, "Guiding Principles", describes the value system we use to evaluate which requirements are most import (or unimportant) for the Fortran preprocessor.

The second section, "High-level requirements", describes the peculiarities of Fortran source, C source, C preprocessor (CPP) directives, and which conventions we will follow for FPP. Many of the differences between CPP and FPP arise from these peculiarities.

The third section describes phases of the translation process that lead up to preprocessing and subsequent processing by "the processor" itself. As we have these phase out, the phase for directive execution in FPP should be identical to CPP.

Finally, we identify next steps.

2 Contributions

This paper establishes the foundations for the detailed specification of the new Fortran preprocessor.

It lays out the framing principles, the high-level requirements, and a phased definition model for specifying the detailed FPP requirements, syntax, and semantics.

3 Guiding principles

The most important decisions we make are deciding what is more important and what is less important (or unimportant) for FPP. Guiding principles provide a *value system* for evaluating technical alternatives, and a priority mechanism to help choose among alternatives.

Here are the guiding principles for FPP, in decreasing order of importance.

3.1 Promote portability

Many programs depend on some preprocessing capability. Most of these rely on a preprocessor that is, or behave much like, the C preprocessor (CPP) [ISO99]. Many of these Fortran preprocessors support different versions of the C preprocessor specification, or exhibit different behavior from each other.

The main goal of this project is to promote the portability of programs that rely on CPP-like behavior by standardizing the semantics of a Fortran preprocessor and making it a mandatory feature of standards-complying processors.

Fortran supports separate compilation and use of modules. The use of the preprocessor in C for importing library interfaces is not really needed in Fortran. (Fortran programmers use the `MODULE` and `USE` statements to import these interfaces.)

3.2 Be "Fortran-aware"

Fortran is not C, and has certain issues in its source form that require special attention [KI24a] that aren't needed in CPP.

- Fortran has two distinct source forms: fixed-form and free-form. FPP should support both.
- Some Fortran tokens and C tokens have different meaning: `//` means concatenation in Fortran, but introduces comments in C; `!` is the 'not' operator in C, but introduces comments in Fortran.

- There are source statements in Fortran where we may not want macro expansion, such as in the `IMPLICIT` *implicit-spec-list* ([ISO23] [F2023§8.7 IMPLICIT statement]) and in the `FORMAT` *format-items* ([F2023§13.3R1303 format-items]).

3.3 Support a meaningful subset of CPP

Deliver a viable product. Support the subset of CPP that used by existing Fortran projects. Add the minimum features necessary to support current and near-future use.

Full compatibility with CPP [ISO18] §6.10 is not required.

When Fortran conventions (such as case-insensitivity) conflict with CPP conventions, lean towards CPP.

Accommodate existing use of CPP-like Fortran preprocessor when there is consensus among the existing processor implementations.

3.4 (Additional principles to come)

4 High-level requirements

At a certain altitude, FPP will look like an unholy marriage of some of the syntax and semantics from the C standard, and some of the syntax an semantics from the Fortran standard.

4.1 CPP conventions we should follow

As existing projects use CPP, or a variant of it, FPP should exhibit the following behaviors [ISO18]. These conventions are already in use in Fortran applications today.

4.1.1 Whitespace is significant in delimiting tokens

Rationale: In CPP, the space character (hex 0x20), the tab character, and new-line delimit tokens.

4.1.2 Case-sensitivity in directive names and token names

Rationale: Macro variable names and directive commands are case-sensitive in CPP. The identifiers `i` and `I` are distinct. (In Fortran, identifiers are not

case-sensitive: `i` and `I` are the same identifier.) We are aware of at least one large application that relies on preprocessor identifiers being distinct that differ only in case.

4.1.3 Line continuation with backslash new-line (\ \n) in directives

Rationale: The [C2018§5.1.1.2¶1·2 Translation phases] specifies that "the sequence of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines."

4.1.4 C expressions in `#if` and `#elif` directives

Rationale: C has no built-in Boolean type, so the expressions in conditional directives evaluate to C integers. An integer zero value represents "false", and any non-zero value represents "true". (Fortran supports `LOGICAL` values as a separate type.)

4.1.5 `/* ... */` comments in directives

Rationale: CPP recognizes C-style comments. (Fortran introduces comments with exclamation marks (`!`).)

4.1.6 `&` and `&&` operators in directives

Rationale: CPP has operators `&` and `&&` on integer expressions. These are seen in Fortran applications today. (Fortran free-form lines use `&` at the end of a line to signal continuation.

4.1.7 C tokens allowed in directives

Recognize C tokens in macro definition and conditional directives.

Rationale: Since macro definitions may include C tokens that will later be used in preprocessor expressions, defined macros need to be able to contain valid C tokens. For example,

```
1 #ifdef SOMETHING
2 #define condition (x == 1)
3 #else
4 #define condition (x == 2)
5 #endif
6 ...
7
8 #if condition
```

```
...
#endif
```

4.2 CPP conventions we should *not* follow

FPP should not adopt the following behaviors from CPP.

4.2.1 `//` introducing comments in directives

Rationale: The `//` operator is the Fortran character concatenation operator, and is a valid token in FPP.

4.3 Fortran conventions we should follow

4.3.1 Macro expansion in fixed-form Fortran

Rationale: Fortran projects today still exist in the obsolescent fixed-form [ISO23]. Roughly half the projects and lines of code we have collected so far, in fact contain fixed-form Fortran [KI24b].

FPP must behave reasonably on this large body of code, but also must bend to conventions used by CPP.

4.4 Fortran conventions we should *not* follow

4.4.1 In fixed-form Fortran, blanks are not significant for determining token boundaries

Rationale: CPP treats blanks and comments as significant for determining token boundaries (they are significant up until translation phase 7 [C2018§5.1.1.2¶8·7]). We follow the CPP convention for tokenizing identifiers. Existing program surely rely on this, but we haven't analyzed this yet in the corpus of existing programs.

4.4.2 `!` comments in directives

Rationale: `!` introduces a comment in Fortran. Unfortunately, this is also the C 'not' operator. To allow conditions with the C `!` not operator (such as below), FPP must treat `!` as the C 'not' operator.

```
#if ! defined(MY_FAVORITE_ID)
```

4.5 Features to decide or revisit

4.5.1 Should we allow Fortran operators in directives?

For FPP directives, we should use C-style expressions, not Fortran expressions. Operators such as `=`, `/=`, `.AND.`, `.OR.`, `.NOT.`, `**` should not appear in `#if` and `#elif` directives. These operators, of course, can appear in the replacement text of `#define` directives.

Rationale for using Fortran-style expressions in definitions: It can be useful to define macros that are legal Fortran and that can also be evaluated by the preprocessor.

```
1 #define INTERESTING(x) ((x) > 1 .AND. (x) < 8)
2
3 #if INTERESTING(SOMETHING)
4         IF (INTERESTING(ANOTHER_THING)) THEN
5             ...
6         END IF
7 #endif
```

4.5.2 Should comments become spaces in Fortran source?

In C, a comment is replaced with a single space before it executes preprocessor directives.

Fortran has never had any kind of formal `pragma` line in the language, programmers embed Fortran compiler directives (e.g., regarding vectorizing optimizations, OpenMP and OpenACC parallelism, and legacy extensions) in comments. FPP may need to preserve these comment-based directives for the processor.

We may define a mechanism for converting these comments into proper directives, to be available to later phases of the compiler.

Or we may decide that passing directives through is implementation-defined behavior.

5 Translation phases

The C standard [ISO24] defines eight translation phases. These phases each perform a well-defined set of operations on the C source code and intermediate representations. They define a processing pipeline where one phase transforms its input in some way, and its output becomes the input to the next phase.

While these phase descriptions explain how C compilers *should behave*, they do not prescribe how C compilers should *be written*.

We do the same for Fortran. For FPP, though, we are only concerned with phases through interpreting preprocessor directives. (The rest of the Fortran standard defines the responsibilities of "The Processor" that validates and transforms Fortran source.)

5.1 Phase 1: Remove continuations

For fixed-form Fortran source, follow the column-6 conventions to produce a sequence of logical lines.

For free-form Fortran source, follow the `&` conventions to produce a sequence of logical lines.

In either form, remove continuations from directive lines (those lines beginning with `#`).

5.2 Phase 2: Process comments

For fixed and free-form source, translate comment-based directives (such as `!dir$`, `!omp$`, `!acc$`, and `CDIR$`, `COMP$`, and `Cacc$`) into some kind of formal pragma (such as a `#pragma` directive).

Which comment-directives are translated to pragmas is processor-dependent.

The Fortran standard should specify whether and how macros are expanded in pragmas.

All remaining comments are replaced with a single space.

5.3 Phase 3: Tokenize the source into preprocessing tokens

The output from Phase 2 is converted to preprocessor tokens according to the rules defined in the Fortran standard and in "High-level requirements" above.

5.4 Phase 4: Execute preprocessor directives

Preprocessing directives in the output from Phase 4 are executed. As in C, the execution of preprocessor directives and interpretation of macro defini-

tion and expansion is a *token-replacement* process, not a *text replacement* process.

Macros are expanded in Fortran source.

Source code is included, excluded, or modified based on the directives.

This phase removes all directives before generating output for subsequent processor phases.

6 Next steps

In subsequent papers, we list the detailed requirements for each phase of the Fortran 202y preprocessor.

References

[ISO99] ISO/IEC. Programming language – C. Standard ISO/IEC 9899:1999, International Organization for Standardization, Geneva, CH, December 1999.

[ISO18] ISO/IEC. Information technology – programming languages – C. Standard ISO/IEC 9899:2018, International Organization for Standardization, Geneva, CH, July 2018.

[ISO23] ISO/IEC JTC 1. Information technology – programming languages – Fortran. Standard ISO/IEC 1539-1:2023, International Organization for Standardization, Geneva, CH, November 2023.

[ISO24] ISO/IEC. Information technology – programming languages – C. Standard ISO/IEC 9899:2018 (R2024), International Organization for Standardization, Geneva, CH, October 2024.

[KI24a] Gary Klimowicz and INCITS/Fortran JoR. On Fortran awareness in a Fortran preprocessor, February 2024.

[KI24b] Gary Klimowicz and INCITS/Fortran JoR. Preprocessor directives seen in existing Fortran programs, February 2024.