Fortran preprocessor requirements

INCITS/Fortran JoR

[2023-10-24 Tue 11:27]

Contents

1 Introduction

INCITS/Fortran has talked several times in Plenary, and the 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 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, "On Fortran source form", describes the peculiarities of Fortran source, C source, C preprocessor (CPP) directives, and which conventions we will follow for FPP.

The third section describes phases of the translation process that lead up to preprocessing and subesequent processing by "the processor" itself.

The last section details individual requirements.

1.1 Citing standards and *de facto* standards

For brevity we write references to the Fortran 2023 standard [ISO/IEC JTC 1, 2023] as follows:

- `[F2023`,
- the section marker `§` and the clause number, then one of
  - `R` followed by a rule number, or
  - `C` followed by a constraint number, or
  - `¶` followed by the paragraph number, and (optionally) a bullet symbol · and the bullet number or letter
- (optionally) the section title for redundant context,
- a closing square bracket `]`.

For example, `[F2023§13.2.1R1301 FORMAT statement]` refers to subclause 13.2.1 (‘`FORMAT statement`’) of the Fortran 2023 standard, rule 1301.

We follow a similar convention for the 2018 C programming language standard [ISO/IEC, 2018].

- `[C2018`,
- the section marker `§` and the subclause number,

- ¶ character and the paragraph number,
- (optionally) a bullet symbol · and the bullet number or letter,
- (optionally) the subclause title for redundant context,
- a closing square bracket `]`.

`[C2018§5.1.1.2¶1·2 Translation phases]` refers to subclause 5.1.1.2 ("Translation phases") of the C 2018 standard, paragraph 1, second bullet point.

We follow a similar convention for the GNUCC '`The C Preprocessor`' manual [Stallman and Weinberg, 2024].

- `[G14`,
- the section marker § and the section number,
- ¶ character and the paragraph number,
- (optionally) a bullet symbol · and the bullet number or letter.
- (optionally) the subclause title for redundant context,
- a closing square bracket `]`.

`[G14§3.3¶2 Macro Arguments]` refers to paragraph 2 of section 3.3 ("Macro Arguments") of version 13 of the GNU CPP reference.

2 Guiding principles

We start by deciding what is important and what is less important (or unimportant) up front for FPP. Guiding principles provide a *value system* for evaluating technical alternatives.

2.1 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) [ISO/IEC, 1999]. 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.

2.2 Be "Fortran-aware"

Fortran is not C, and has certain issues in its source form that require special attention [Klimowicz and JoR, 2024a] 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 don't necessarily want macro expansion, such as in the `IMPLICIT` *implicit-spec-list* ([ISO/IEC JTC 1, 2023] §8.7) and in the `FORMAT` *format-items* (§13.3).

2.3 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.

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

Full compatibility with CPP [ISO/IEC, 2018] §6.10, though, is not required.

3 On Fortran source form

Fortran projects today still exist in the obsolescent fixed-form [ISO/IEC JTC 1, 2023]. Roughly half the projects and lines of code we have collected so far, in fact contain fixed-form Fortran [Klimowicz and JoR, 2024b].

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

Compromises must be made.

3.1 CPP conventions we should follow

As existing projects use CPP, or a variant of it, FPP should exhibit the following behaviors. [ISO/IEC, 2018]

Whitespace is significant The space character (hex 0x20), the tab character, and new-line delimit tokens. (In fixed-form Fortran, spaces are ignored.)

Case-sensitivity Macro variable names and directive commands are case-sensitive. The identifiers `i` and `I` are distinct. (In Fortran, identifiers are not case-sensitive: `i` and `I` are the same identifier.)

Directive line continuation with backslash The C convention is that the sequence of a backslash character (`\textbackslash`) immediately followed by a new-line character is deleted. (Fortran defines line continuation using column 6 in fixed-form, and the ampersand character (`&`) in free-form.)

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

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

`&` and `&&` operators CPP has operators `&` and `&&` on integer expressions. (Fortran free-form lines use `&` at the end of a line to signal continuation.

3.2 CPP conventions we should not follow

FPP should not copy the following behaviors from CPP

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

Comments become spaces In C, a comment is replaced with a single space before it executes preprocessor directives. But Fortran has never had any kind of formal `pragma` line in the language, so programmers embed Fortran compiler directives (e.g., regarding vectorizing optimizations, OpenMP and OpenACC parallelism, and legacy extensions) in comments. FPP must in some way preserve these comment-based directives for the processor.

C tokens With the exceptions mentioned above and below, FPP should recognize tokens as Fortran does, not C.

3.3 Fortran conventions we should not follow

`!` comments `!` introduces a comment in Fortran. Unfortunately, this is also the C 'not' operator. To allow conditions like `! defined(MY_FAVORITE_ID)`, FPP must treat `!` as the 'not' operator.

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.

4 Translation phases

The C standard [ISO/IEC, 2018] defines eight translation phases. These phases each perform a well-defined set of operations on the C source code and intermediate representations.

These phases describe how C compilers should behave. They do not describe how C compilers should be written.

We will do the same for Fortran. For FPP, though, we are only concerned with phases 0 through 5. The rest of the Fortran standard defines the responsibilities of "The Processor".

4.1 Phase 1: Remove continuations

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

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

4.2 Phase 2: Translate comment directives

For fixed and free-form source, translate comment-based directives (such as `!dir$`, `!omp$`, `!acc$`) into `#pragma` directives.

Which comment-directives are translated to `#pragma` directives is processor-dependent.

4.3 Phase 3: Remove comments

All remaining comments are replaced with a single space.

4.4 Phase 4: Tokenize the source into preprocessing tokens

The output from Phase 3 is converted to preprocessor tokens according to the rules defined in "On Fortran source form" above.

4.5 Phase 5: Execute preprocessor directives

Preprocessing directives in the output from Phase 4 are executed. Macros are expanded in directives and Fortran source.

4.6 Phase 6: Convert preprocessing tokens to tokens

The output from Phase 5 is converted to tokens suitable for syntactic analysis.

4.7 Phase 7: "The Processor"

4.7.1 Analyze syntax

Analyze the output from Phase 6 for syntactic correctness. Construct any necessary intermediate representation for semantic analysis.

4.7.2 Analyze semantic correctness

Find and report static semantic errors in the Fortran program. Check for Fortran constraint violations.

4.7.3 Generate code

Translate the output from Phase 7 to runnable code.

5 Detailed requirements

We list the detailed requirements for the Fortran 202y preprocessor.
Each requirement is a heading of the form "One-line description"
Item properties for these headings contain

- A requirement unique identifier in square brackets `[]`.
- Current status (TBD, JoR yes, JoR no, WG 5 yes, WG 5 no, etc.).
- Normative references (such as the C standard).

- Where the requirement came from in Fortran discussions and posts.

The requirements came from the following sources.

cpp *cpp* if in the C standard [ISO/IEC, 2018].

cpp-ish if in C standard, but made Fortran-aware.

gcpp [Stallman and Weinberg, 2024]

ble1 JoR Email threads from Rich Bleikamp re: tutorial *[2022-08-08 Mon 21:34]*.

che1 Email from Daniel Chen to JoR *[2022-07-29 Fri 11:08]*.

clu1 Email from Tom Clune *[2022-08-01 Mon 10:48]*.

fla1 LLVM Flang Preprocessing.md [https://github.com/llvm/llvm-project/blob/main/flang/docs/Preprocessing.md]

gak Gary Klimowicz made it up as he wrote these specifications

jor1 JoR meeting on preprocessors *[2022-08-22 Mon 10:00]*.

jor2 JoR meeting on preprocessors *[2022-09-20 Tue 13:00]*.

jor3 JoR meeting on preprocessors *[2023-11-07 Tue 12:00]*.

jor4 JoR meeting on preprocessors *[2022-12-06 Tue 12:00]*.

gak Private communication in his head.

lio1 Email from Steve Lionel *[2022-08-01 Mon 13:52]*.

lio2 JoR discussion forum https://j3-fortran.org/forum/viewtopic.php?p=561

lio3 JoR discussion forum https://j3-fortran.org/forum/viewtopic.php?p=562

References

- INCITS/Fortran JoR subgroup email re: cpp tutorial for October [2023] meeting?
- [ISO/IEC, 2018]
- [ISO/IEC JTC 1, 2023]
- LLVM Flang Preprocessing.md

5.1 Phase 1 Remove continuations

5.1.1 Remove C-style \ line continuations in directives

In fixed-form and free-from source code, delete a backslash \ immediately followed by a new-line character.

From The C standard:

> Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place.

5.1.2 Process Fortran line 6 continuation in fixed-form Fortran text

In fixed-form input, a character in column 6 that is not blank or the digit zero continues the line with the previous line, even if the previous line is a directive line, or the continuation of a directive line.

5.1.3 Process Fortran `&` line continuation in free-form Fortran text

In free-form input, an `&` character as the last character on a directive line indicates the directive continues on the next line. The handling of the continuation is as described in [F2023§6.3.2.4].

5.1.4 Remove leading spaces before `&` in Fortran line continuation in free form

In free-form input, an `&` character as the last character on a directive line indicates the directive continues on the next line. When the first non-blank character on the next line is also an `&`, the characters between the ampersands are deleted.

5.1.5 Comment lines in definitions with continuation lines

5.2 Phase 2 Translate comment directives

5.2.1 Convert Fortran line comments to `#pragma` directives in fixed-form

5.3 Phase 3 Remove comments

5.3.1 Strip C-style `/* ... */` comments in directive lines

5.4 Phase 4 Tokenization

5.4.1 Case sensitive tokens

:PROPERTIES: :reqid: [tokens-case-sensitive] :status: TBD :source: fla1

Fortran is not case-sensitive. The preprocessor *is* case-sensitive when recognizing identifiers. The text fragment

```
#define abc XYZ
#define ABC foo
      subroutine abc
```

should expand to

```
      subroutine XYZ
```

Note that this creates (perhaps astonishing) behavior, such as

```
#define ABC var_1
#define abc var_2
      abc = ABC + 1       ! Normally, Fortran treats these as the same identi
```

expanding to

```
      var_2 = var_1 + 1     ! These identifiers are now different
```

Fortran programmers may expect it to expand to

```
      var_2 = var_2 + 1     ! Only the second definition matters
```

5.4.2 Case insensitive tokens

:PROPERTIES: :reqid: [tokens-case-insensitive] :status: TBD :source: fla1

Fortran is not case-sensitive. The preprocessor is not case-sensitive when recognizing identifiers. The text fragment

```
#define abc XYZ
#define ABC foo
      subroutine abc
```

should expand to

```
      subroutine foo
```

If the preprocessor were case sensitive, we would have behavior, such as

```
#define ABC var_1
#define abc var_2
      abc = ABC + 1       ! Normally, Fortran treats these as the same identi
```

expanding to

```
      var_2 = var_1 + 1     ! These identifiers are now different
```

We should expect it to expand to

```
      var_2 = var_2 + 1     ! Only the second definition matters
```

5.4.3 Identifier tokens are not broken by line continuations

In fixed-form, there are only 66 characters available for statement text (columns 7-72). The maximum length of an identifier is 63 characters. It is not practical to have identifiers end at a fixed-form line boundary at column 72.

5.4.4 Spaces significant in determining tokens

In order to simplify the preprocessor tokenization, spaces are significant, even in fixed-form source.

5.5 Phase 5 Directive processing

5.5.1 A # in column 6 in fixed-form is not a directive

This is seen in some existing projects that use `#` in column 6 for a conventional continuation line. These are the Fortran files in fortran-examples that have `#` in column 6. Note that some expect to run through the preprocessor (extension is `.F`).

```
ALBUS_ionosphere@twillis449/FORTRAN/IRI/igrf.f
CMAQ@USEPA/POST/sitecmp_dailyo3/src/process.F
E3SM@E3SM-Project/components/mpas-ocean/src/mode_forward/mpas_ocn_time_integration_si.F
```

Genetic-Algorithm-for-Causeway-Modification@stevenmeyersusf/Code/genmain.f
MCFM-RE@lcarpino/src/Parton/eks98r.f
MITgcm@MITgcm/pkg/openad/externalDummies.F
NCEP_Shared@GEOS-ESM/NCEP_w3/w3ersunb.f
OEDGE@ORNL-Fusion/lim3/comsrc/sysaix.f
PublicRelease_2020@FLOSIC/flosic/scan.f
STELLOPT@PrincetonUniversity/LIBSTELL/Sources/NCLASS/nclass_mod.f
ShirleyForQE@subhayanrc/yambo-stable/src/real_time_common/RT_driver.F
cernlib@apc-llc/2005/src/graflib/higz/imac/f_readwi.F
cernlib@apc-llc/2006/src/graflib/higz/imac/f_readwi.F
cfdtools@nasa/app/traj_opt/numerics.f
cfdtools@nasa/lib/searchlib/hsortcc.f
dynamite@dynamics-of-stellar-systems/legacy_fortran/galahad-2.3/src/ma27/ma27d.f
forestclaw@ForestClaw/applications/clawpack/euler/2d/rp/rpn2euq3.f
hompack90@vtopt/src/MAINP.f
legacy-mars-global-climate-model@nasa/code/cmp3out.f
nosofs-NCO@ioos/sorc/SELFE.fd/utility/Combining_Scripts/combine_outHA.f
nwchem@nwchemgit/src/nwpw/band/lib/psi/cpsi_KS.F
nwchem@nwchemgit/src/tce/mrcc/tce_mrcc_energy.F
pyOpt@madebr/pyOpt/pySLSQP/source/slsqp.f
pyclaw@clawpack/development/rp_approaches/rpn2_euler_5wave.f
scream@E3SM-Project/components/mpas-ocean/src/mode_forward/mpas_ocn_time_integration_si
starlink@Starlink/applications/echomop/ech_kdhsubs.f
starlink@Starlink/applications/obsolete/iras90/misc/ffield.f
starlink@Starlink/thirdparty/caltech/pgplot/examples/pgdemo17.f

5.5.2 # non-directive

5.5.3 Conditional inclusion

5.5.3.1 # if *constant-expression*

5.5.3.2 # ifdef *identifier*

5.5.3.3 # ifndef *identifier*

5.5.3.4 # elif *constant-expression*

5.5.3.5 # else

5.5.3.6 # endif

5.5.4 Source file inclusion

5.5.4.1 # include *char-literal-constant*

5.5.4.2 # include *pp-tokens*

5.5.5 Macro replacement

5.5.5.1 # define id replacement-list

5.5.5.2 # define id (id-list) replacement-list

5.5.5.3 # define id (...) replacement-list

5.5.5.4 # define id (id-list , ...) replacement-list

5.5.5.5 # undef

5.5.6 Line control

5.5.6.1 # line

5.5.7 Error directive

5.5.7.1 # error

5.5.8 Error directive

5.5.8.1 # warning

5.5.9 Pragma directive

5.5.9.1 # pragma

5.5.10 Null directive

5.5.10.1 # newline :reqid: [#null] :status: TBD

- **Source:**

:references: [C§6.10.7 Null directive]

5.5.11 Additional requests

5.5.11.1 # show

#show prints a table of all the macros; and a "#show namea nameb name*" does the same with just the listed names, but allows simple globbing. Simple and sometimes useful for distinguishing between output files when actually retaining the intermediate files. All the output is written to the output file with lines starting with ! (not standard, but even a lot of pre-f90 compilers allowed ! as a comment)

so the output is still valid Fortran.

5.5.11.2 # import VARNAME

which imports an environment variable as if it had been defined with -DVARNAME=VALUE

5.5.11.3 # output filename [–append]

which makes it easy to have a single file that outputs Fortran, C, markdown ... sections but would complicate a preprocessor being "inline" in the compiler, which I hope is an expected feature of a standard preprocessor, thus being able to eliminate having to generate (or at least retain) intermediate files, being able to define reusable blocks of plain test and reuse them or loop over them applying an optional filter that can convert them all to comments, convert them all the a character variable definition, or convert them to WRITE statements. Makes maintaining comments and help text a lot easier, as you can just type it as plain text, for example. I would be content with just cpp-like functionality, but those are features I use a lot cpp(1) does not do, except typically (not in all fpp flavors) block comments are supported. I think a #import would be useful and simple though. Perhaps a group concensus would be it is problematic, making sure it is not inadvertently the wrong value, ... so not sure even that would make it into a first-generation standard utility.

5.6 Predefined macros

5.6.0.1 `__FILE__` Current file name

5.6.0.2 `__LINE__` Current line number

5.6.0.3 `__STDF__` Fortran compiler (1)

5.6.0.4 `__STDF_HOSTED__` Compiler is hosted cross-compiler (e.g., 202311)

5.6.0.5 `__STDF_VERSION__` Fortran standard conformance

5.6.0.6 `__DATE__` of processing

5.6.0.7 `__TIME__` time of processing

- Source :::

:reqid: [macro-file-process-time] :status: TBD :source: cpp :references: [C§6.10.8 Predefined macro names]

5.6.0.8 `STRINGIFY` macro function Tom Clune requested a macro named `stringify`, which apparently is commonly used in some codes. There is no stadard `stringify` macro, so JoR declined to add one.

5.6.0.9 `__SCOPE__` defines current lexical scope

5.6.0.10 `__VENDOR__`

5.6.0.11 No undecorated names (no _) defined by preprocessor

5.6.1 Expressions

5.6.1.1 `#`

5.6.1.2 `##`

5.6.1.3 `defined` *identifier*

5.6.1.4 `defined (` *identifier* `)`

5.6.1.5 `!`

5.6.1.6 [c-expressions] C-style expressions

- Source :::

:status: TBD :references: [C§6.10 Preprocessing directives]

This is problematic, in that there are many files in the wild that have the `!` not operator in `#if` and `#elif` directives. This is a Fortran comment character.

5.6.1.7 [fortran-expressions] Fortran-style expressions

- Source :::

:status: TBD :references: [C§6.10 Preprocessing directives]

5.6.2 Expansion

5.6.2.1 No expansion of `C` in column 1 In fixed-form, `C` or `c` in column 1 indicates a comment. No identifier that begins with a `C` or `c` in column 1 is expanded.

5.6.2.2 No expansion of `D` in column 1 In fixed-form, `D` in column 1 is a common extension to indicate a comment. No identifier that begins with a `D` in column is expanded.

5.6.2.3 No expansion of column 6 In fixed-form, a character in column 6 that is not blank or zero indicates a continuation line. No identifier that begins in column 6 is expanded.

[Note that since we define expansion to occur after continuation handling, this requirement is not necessary.]

5.6.2.4 Strip column 1 `C` comments from expanded text

5.6.2.5 Pass comments from expanded text to the processor This is necessary to pass comments with directives to the Fortran processor.

5.6.2.6 No expansion of function macro names not followed by parenthesis String constants are output without being examined for macro expansion.

5.6.2.7 Function macro name invocation may cross logical line boundaries String constants are output without being examined for macro expansion.

5.6.2.8 No expansion of self-referential macro names String constants are output without being examined for macro expansion.

5.6.2.9 No expansion in strings String constants are output without being examined for macro expansion.

5.6.2.10 No expansion in Hollerith No expansion occurs in the string contained in a Hollerith constant.

5.6.2.11 No expansion in `IMPLICIT` single-character specifiers The letters in an `IMPLICIT` statement are not considered for macro expansion.

Note that this implies the preprocessor recognizes `IMPLICIT` statements.

5.6.2.12 No expansion in `FORMAT` specifiers In `FORMAT` statements, there is no macro expansion in the *format-specification.*.

Note that this implies the preprocessor recognizes `FORMAT` statements.

5.6.2.13 Expansion in comments Outside the exceptions noted elsewhere, the preprocessor expands macros in the text of comments.

[This may be the way the preprocessor preserves directives (such as OpenMP and OpenACC) in the program. We have heard from at least one J3 member that this is on important feature. The good news is that expanding macros always will handle directives, without having to do special processing for all manner of directives. The other good news in that comments without directives are ignored by the processor, so we don't care (maybe) how they are mangled.]

5.6.2.14 Expansion in directives (e.g., OpenMP) This is problematic, as how does the preprocessor know which comments are directives?

5.6.2.15 Expand `INCLUDE` lines as if `#include` :PROPERTIES: :reqid: [preprocess-fortran-include] :status: TBD :source: fla1, jor1, JoR4

Assuming the preprocessor is a mandatory part of the Fortran standard, preprocessor directives are allowed in the file specified in a Fortran `INCLUDE`

line. Therefore, the preprocessor should process the `INCLUDE`-ed file as if it had been invoked via the `#include` directive.

Otherwise, where will the handling of directives the included file be handled, and how can it use any of the macro definitions available at the time the `INCLUDE` statement is encountered. (It is likely to be included in multiple places in the application.)

5.6.2.16 Expand macro names in *kind-param* in literal constants

If the *kind-param* is a *scalar-int-constant-name* following the underscore in an *int-literal-constant*, *real-literal-constant*, and *logical-literal-constant*, that constant name is subject to macro expansion. This needs to be explicit, as otherwise the preprocessor might treat `_kind-name` as an identifier, as many preprocesser predefined macro names begin with an underscore.

In a *char-literal-constant*, if the *kind-param* preceding the underscore (`_`) is a *scalar-int-constant-name*, that constant name is subject to macro expansion. This needs to be explicit, as otherwise the preprocessor might treat `kind-name_` as an identifier.

5.6.3 Output form

The user may request the preprocessor to produce the source representation after preprocessing. How this is requested is processor-dependent. If provided, the source form of the preprocessor output shall be bound by the following requirements.

5.6.3.1 `#line` and `#file` directives in the output

To show source code origin information (such as for error messages), the preprocessor may generate such origin information in the form of `#line` and `#file` directives. These directive lines may be up to 10,000 characters long, as in free-form Fortran input.

5.6.3.2 Directive lines begin with a `#` in column 1

5.6.3.3 Fixed-form input becomes fixed-form output

When the preprocessor produces output corresponding to fixed-form Fortran input (such as with the `-E` option supported by some C compilers), it must produce valid Fortran fixed-form source code. This may require re-flowing the preprocessed output to the 72-column boundary.

Column 1 Comments begin with a `C` or `c` in column 1.

Columns 1-5 Optional statement label.

Column 6 `0` or blank if not a continuation; continuation line otherwise.

Columns 7-72 Fortran statement text

5.6.3.4 Free-form input becomes free-form output When the preprocessor produces output corresponding to free-form Fortran input (such as with the `-E` option supported by some C compilers), it must produce valid free-form Fortran source code. This may require re-flowing the preprocessed output to the 10,000-column boundary.

Columns 1-10,000 Up to 10,000 characters of Fortran source text. If the source text is longer than 10,000 characters, up to 9,999 characters followed by `&` to continue additional source text on the next physical line, continuing up until one million characters of Fortran source.

5.7 Phase 6 Convert preprocessing tokens to tokens

With a suitable representation for preprocessing tokens, this phase may be unnecessary.

6 Further investigation of existing Fortran projects

6.1 Add directives found inside `#include` files

6.2 Add directives found inside `INCLUDE` files

6.3 Count comments embedded between continuation lines

6.4 Are any comment-based directives sandwiched between continuation lines?

6.5 `#if`, `#elif`: Can we tell if `!` is used for comment?

6.6 `#define` Can we tell if `!` is used as a comment?

6.7 What projects depend on case-sensitive macro replacement?

6.8 What projects depend on not expanding in `INCLUDE` files (besides VASP)?

6.9 Examine the unrecognized directives

6.10 Identify if there are `INCLUDE` files that would be hurt by preprocessing

References

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

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

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

[Klimowicz and JoR, 2024a] Klimowicz, G. and JoR, I. (2024a). On fortran awareness in a Fortran preprocessor.

[Klimowicz and JoR, 2024b] Klimowicz, G. and JoR, I. (2024b). Preprocessor directives seen in existing Fortran programs.

[Stallman and Weinberg, 2024] Stallman, R. M. and Weinberg, Z. (2024). The C preprocessor.