Fortran dependency generator

F. Prill, DWD

11/2016

This text describes the dependency generator which can be applied to the Fortran source files during the ICON `configure` process. The tool recognizes a subset of preprocessor directives and implies a few code conventions.

1 General description

The generator tool provides a list of source files, containing all dependencies of a given "root" file or module. Technically, this is a tokenizer for Fortran files. It filters `USE` dependencies depending on preprocessor `#ifdef`'s, `#define`, and `#undefine`.

1.1 Usage

The general options for the command-line tool are

```
./usedep_parser <root> "<defined symbols>" "<filenames>" \
                [-v] [-m] [-objprefix=xxx]
```

Each list item is given in a separate line.

Usually, the `<root>` argument is the file name of the program source file `icon.f90`. Alternatively, the tool may generate the dependencies of a specific module.

The argument `"<defined symbols>"` is a space-separated list of defined preprocessor symbols.

The `"<filenames>"` argument is a space-separated list of files (full path, without wildcards) where to search for dependencies. Dependency modules which are not defined in this set of files are not listed in the output.

The optional argument `-v` enables some additional output, e. g. dependencies which are outside of the search scope.

The optional argument `-m` enables output of all (direct) dependencies for each of the dependent files as it is required by a `Make` process. In this case, all file names are translated into object file names `%.o` and (optionally) a prefix string is added which can be set through the command-line argument `-objprefix=`*xxx*.

1.1.1 Example

First, a list of files where to search for dependencies must be generated, e. g.

```
export FILELIST=$(ls ~/trunk/icon-nwp-dev/src/*/*f90)
```

Then the Fortran dependency generator may be invoked by

```
./usedep_parser mo_intp_rbf "__ICON__ __NO_JSBACH__ __NO_ICON_OCEAN__" "$FILELIST"
```

2 Recognized preprocessor directives

The tool recognizes only a subset of preprocessor directives (see below). These limitations imply some code conventions which must be met by the preprocessed source files.

2.1 (Known) Limitations and necessary code conventions

- Line breaks: only single-line `#if` conditions. No line break before ”xxx” in ”USE xxx”.
- `#define` directives: for simplicity setting of values is ignored, only (non-)existence matters.
- USE dependencies must not be dynamically generated through preprocessor macros.

Of course, the tool performs only static analysis; it does not inspect if dependencies are actually applied. INTRINSIC modules are not treated as dependencies.

2.2 Recognized grammar

Preprocessor ifdef pattern. nested ifdef's are handled via semantic conditions:

```
hash        = '#' ' '*
condition   = (alnum | [()&\-| _><!])+
open_ifdef  = (hash 'ifdef' | '#if' space) space* condition
open_ifndef = hash 'ifndef' space* condition
open_elif   = hash 'elif'   space* condition
close_ifdef = ( hash 'endif' when { ilevel > 0 } )
else_ifdef  = hash 'else'
```

MODULE/END MODULE name pattern.

```
valid_name  = (alpha|[_]) (alnum | [_])*
module      = space* /MODULE/i space+ valid_name [ ]* '\n'
endmodule   = space* /END/i [ \t]* /MODULE/i space+ valid_name [ ]* '\n'
```

USE dependency pattern.

```
comment     = '!' [^\n]*
use_spec    = (',' space* [^\n]*)
usedep      = space* /USE/i space+ valid_name space* (comment|use_spec)? '\n'
```

INCLUDE dependency pattern.

```
filename    = alpha (alnum | [._])*
includedep  = hash 'include' space* ['"] filename ['"]
```

DEFINE/UNDEFINE pattern.

```
define       = hash 'define' space+ valid_name
undefine     = hash 'undef'  space+ valid_name
cmplx_define = hash 'define' space+ valid_name'(' alnum+ ')' space+ hash
```

Code pattern (the rest).

```
string_literal = (('"' [^"]* '"') |  ("'" [^']* "'"))
code           = (([^\n#!'"]) | string_literal )*  comment?  '\n'
```

3 Technical details

The program contains two different parsers. First, a top-level scanner/parser separates `#ifdef`'s and `USE` dependencies from the remaing content. A second-level scanner/parser then constructs an expression-syntax tree for each `#ifdef` condition.

The program is implemented in C++11. Its parser is generated by the Ragel State Machine Compiler[1].

[1] see `http://www.colm.net/open-source/ragel/`