

# Findent: design

Willem Vermin\*

Jan 09, 2025

## Abstract

*Findent* is a computer program to indent Fortran sources. *Findent* can also be used to convert from fixed format to free format and *vice versa*. The program *findent* became more complex than originally foreseen, that is why I wrote this document covering the inner workings of *findent*.

## Contents

<b>1</b>	<b><i>Findent</i>: Design objectives</b>	<b>2</b>
1.1	Reliability . . . . .	2
1.2	Usability . . . . .	2
<b>2</b>	<b>Building <i>findent</i></b>	<b>3</b>
2.1	End users . . . . .	3
2.2	Program development and maintenance . . . . .	4
<b>3</b>	<b>Usage</b>	<b>4</b>
3.1	Flags to influence the working of <i>findent</i> . . . . .	4
3.2	Generating documentation . . . . .	5
3.3	Miscellaneous other functions . . . . .	5
<b>4</b>	<b>Detailed overview of the internals of <i>findent</i></b>	<b>5</b>
4.1	Starting the machinery . . . . .	5
4.2	The main driver: <code>fortran.run()</code> . . . . .	6
4.3	Collecting a full Fortran statement . . . . .	6
4.4	Preprocessing the full Fortran statement . . . . .	7
4.4.1	Parsing the preprocessed full statement . . . . .	7
4.5	Keeping track of indents . . . . .	8
4.6	Handling <code>cpp</code> and <code>coco</code> preprocessor statements . . . . .	8
4.7	Relabelling . . . . .	10
4.8	Generate miscellaneous text files . . . . .	10
<b>5</b>	<b>Self replication</b>	<b>10</b>
<b>6</b>	<b>Copyright</b>	<b>11</b>

---

\*Email: [contact@ratrabbbit.nl](mailto:contact@ratrabbbit.nl). Website: <https://ratrabbbit.nl>.

# 1 *Findent*: Design objectives

## 1.1 Reliability

Assuming that *findent* will be used in serious projects, involving large Fortran programs, the code of *findent* should be as reliable as possible, therefore it is kept as simple as possible:

- *Findent* handles only two files: standard input and standard output. The input contains the Fortran program to be handled, the output contains the modified program.
- The programming language is C++, a well maintained and documented language.
- No multithreading is used.
- Parsing the input is done with the aid of `bison` and `flex`: well known and maintained tools.
- *Findent* mostly works on a line-by-line basis. (Exceptions: labelled `DO`-loops require a simple administration, and relabelling needs the complete source of a program unit.)
- *Findent* uses no configuration files: it is steered by command-line parameters and an environment variable containing command-line parameters.
- A comprehensive test-suite is part of the distribution.

## 1.2 Usability

*Findent* is easy to use, yet offers the possibility to tweak the indentation to the user's taste:

- All options have reasonable defaults, for example, usage can be as simple as:

```
findent < program.f90 > newprogram.f90
```

Furthermore, the *findent* distribution comes with a wrapper script *wfindent* that can be used like (to indent all `.f90` files):

```
wfindent *.f90
```

Normally, *findent* detects if the input is in fixed or free form.

- All types of indentation (`DO`, `SUBROUTINE`, ...) can be specified on the command line, for example to use 5 spaces after `DO`:

```
findent --indent-do=5 <program.f90 > newprogram.f90
```

- *Findent* ignores white space outside strings and label fields.
- Fixed and free format Fortran are supported.
- Conversion from fixed to free form is implemented, as well as the other way around.
- All kinds of DO-loops are recognized, even nested DO-loops using the same label.
- *Findent* has been tested on legacy Fortran sources, going back to Fortran IV. Hollerith's are parsed correctly.
- Unrecognized constructions are allowed and are written on the output as-is. Incomplete Fortran sources are handled gracefully.
- *Findent* can relabel the Fortran source. The man page contains a warning: 'use this only on correct programs'. If *findent* detects a problem (missing label definition; incomplete program unit; ...), relabelling is abandoned.
- High speed: *findent* indents about 100.000 lines per second.

## 2 Building *findent*

### 2.1 End users

Building *findent* is easy and is based on standard tools:

- The distribution tar ball is based on autoconf, a mature program suite to distribute program sources.
- The distribution tar ball contains the output files of `flex` and `bison`, so the user doesn't need to install these programs. (If they are installed, they will be used).
- On Linux *findent* is built by unpacking the tar-ball, and issue the commands:

```
cd findent-xx.yy.zz
./configure
make
make check # to run the test-suite
sudo make install
```

- For MacOS, building *findent* is the same as for Linux.
- A Windows version can be obtained by the following:

```
a=i686-w64-mingw32
b=`gcc -dumpmachine`
export CXX="$a-g++ -static"

./configure --build=$b --host=$a
```

```
make clean
make
```

You need to have `g++-mingw-w64-i686-win32` or something like that available. Probably, using WSL or Cygwin on Windows should make it possible to do the build on the Windows system.

- If building as presented above does not succeed, the script `simplemake.sh`, containing usage instructions, can be used.

## 2.2 Program development and maintenance

The following is for developers and maintainers:

- The script `bootstrap` runs `autoreconf`, replaces the copyright statements in nearly all sources and generates the output of the `flex` and `bison`. This output will be contained in the distribution tar ball, generated with

```
make distcheck
```

- An esoteric option is `--with-esope`. This causes *findent* to recognize `esope` constructs, see <http://www-cast3m.cea.fr/html/esope/esope.html>.
- In the files `src/debug.h` and `src/debug.cpp` macros and functions are defined to be used when debugging. There is comment in those files how to use them.
- *Findent* comes with a comprehensive test suite, located in the directory `test`. The tests will exercise every flag, and check if solved bugs are still solved. Testing is activated by running:

```
make check
```

## 3 Usage

### 3.1 Flags to influence the working of *findent*

Options to *findent* can be given on the command line, like:

```
findent -ifixed -ofree -i2 < prog.f > prog.f90
```

and/or in the environment variable `FINDENT_FLAGS`, like:

```
export FINDENT_FLAGS='-i4 -I8'
findent -ifixed -ofree < prog.f > prog.f90
```

Most flags relate to the format of the input file and output file. However, some options arrange that *findent* does not output an indented Fortran source, but other information. These flags are marked in the man page with the string `[NO_ENV]` and

are ignored when present in the environment variable `FINDENT_FLAGS`. Invalid flags, both on the command line and in the environment, are silently ignored<sup>1</sup>. Flags are read first from `FINDENT_FLAGS` and secondly from the command line. The flags are handled in the files `src/flags.cpp` and `src/flags.h`. See the man page for a description of the flags.

## 3.2 Generating documentation

*Findent* can generate the following documentation:

- A text file ('help-file'), describing all flags.
- A man page, suitable for processing with the program `man`.
- A text file, containing the `ChangeLog`.
- Text files, describing the usage in an editor.
- Text file describing how to use *findent* in editor, for example `vim`.

## 3.3 Miscellaneous other functions

- Print version of *findent*.
- Print 'free' or 'fixed', depending on what *findent* deduces from the input.
- Print dependency information, based on:
  - Usage and definitions of modules.
  - Usage of include files.
- Print a shell script to be used in combination with the dependencies.
- Print the amount of indentation of the last line read.
- Print the line number of the last usable line as a start for indenting.
- Print a report of defined and used labels.
- Print scripts to incorporate *findent* in the editors `Vim`, `Emacs` or `Gedit`.

# 4 Detailed overview of the internals of *findent*

## 4.1 Starting the machinery

The main program is in `findent.cpp`. The flags are read (`get_flags()`), and if some kind of documentation has to be produced (`docs.print()`), the program prints the documentation and returns. Otherwise, the class `Findent` from `findentclass.h` is instantiated as `findent` and `findent.run()` from `findentruncpp` is called.

---

<sup>1</sup>Since *findent* only writes to standard output, error messages would clutter the indented Fortran program.

## 4.2 The main driver: `fortran.run()`

`fortran.run` executes the following tasks (trivia are omitted here):

- If standard input is connected to a terminal, take appropriate actions.
- Read all of the input and store the lines in a buffer (`input_buffer`).
- If the input format is not forced to be fixed or free, call `determine_fix_or_free()` to determine the format.
- Instantiate class `indent` to either `Free` or `Fixed` as appropriate. These are subclasses of class `Fortran` in `fortran.h` to define the free or fixed alternatives of certain functions. See `free.cpp`, `free.h`, `fixed.cpp` and `fixed.h`.
- Take actions if the user wants to relabel the input.
- Enter the indenting loop (trivia omitted here):
  - Call `get_full_statement()` to create full Fortran line `full_statement` by collating the possible continuation lines to the first (see below).
  - Call `indent_and_output()` to determine the indentation and output the lines that define the full Fortran line.

## 4.3 Collecting a full Fortran statement

Collecting a full Fortran statement from the first line and continuation lines is done in `src/fortran.cpp`: `get_full_statement()`. This function looks surprisingly complex at first sight. This is because continuation lines can contain:

- comment lines,
- blank lines,
- `cpp` or `coco` preprocessor statements,
- `findentfix` lines (see the man page).

Furthermore, attention must be paid if we are relabelling or not. The full Fortran statement is stored in `full_statement`.

In `src/fortranline.cpp` and `src/fortranline.h` functions are defined to handle lines with Fortran code. Care has been taken that for fixed format, a tab at the start of a line is handled properly (see `do_clean()`).

The call to `build_statement()` has a different implementation for free and fixed format, see `src/free.cpp` and `src/fixed.cpp`, respectively. This function performs the following tasks:

- Add the input `Fortranline` to `c-lines` (a list of `Fortranline`'s).
- Add the line, stripped from all non-fortran stuff to `full_statement`.
- Signal if there are continuation lines to be expected. This is easy in the free-form case, but in the fixed-form case a look-ahead is necessary. See `wizard()` in `fixed.cpp`.

## 4.4 Preprocessing the full Fortran statement

Once a `full_statement` has been obtained, this line is preprocessed to make it suitable for parsing using `flex` and `bison`. This is done in `Line_prep::set_line()`, in file `src/line_prep.cpp`. An example may clarify this.

Below is:

- **s** - The full statement.
  - **sl** - Spaces removed, except in strings and Hollerith's, and after the statement label.
  - **sv** - Strings, Hollerith's, operators and statement label replaced by a space.
  - **sc** - Strings etc. replaced by space number space, the number is the index in `sv`. (`sc` is used for parsing with `bison` and `flex`.)
  - **wv** - A list, `length = sv.size()`. Each entry consists of a struct `whats` (see `line_prep.h`) which tells (type) what this entry contains: `invalid`, `none`, `string`, `statement label` or `operator`. In case of `string` there is `stringtype` which discriminates between Hollerith (`h`), single quoted string (`'`) or double quoted string (`"`). The value of the entry is contained in `value`.
- 

```
s: [123  call sub(5habcde , 5, 'f oo', 'ab c' .concat. "def")]
sl: [123 callsub(5habcde,5,'f oo','ab c'.concat."def")]
sv: [ callsub( ,5, , )]
    [0123456789012345678] ! these are index numbers in sv
sc: [ 0 callsub( 9 ,5, 13 , 15 16 17 )]
wv[0]: statement label      [123]
wv[9]: hollerith string     [abcde]
wv[13]: single quoted string [f oo]
wv[15]: single quoted string [ab c]
wv[16]: operator            [concat]
wv[17]: double quoted string [def]
```

The other entries have `type=none`.

---

### 4.4.1 Parsing the preprocessed full statement

Parsing the preprocessed full statement is done using `bison` and `flex`. Things are arranged that one line at a time is parsed, see `lexer_set(Line_prep p, const int state)` in `lexer.l`. The string `sc` (see above) is used for parsing. Parsing is initiated in `indent_and_output()` in `fortran.cpp` by a call to `parseline()`. This function, returning a `propstruct` (see `prop.h`) containing the results, parses the full statement in two passes:

- The lexer is brought in a state that does not recognize Fortran keywords. For example:  
    `subroutinesub(x)=10`  
will return `kind=ASSIGNMENT`.

- If parsing does is not successful (kind = UNCLASSIFIED), full statement is parsed again, but now the lexer is in a state to recognize relevant Fortran keywords.

For example:

```
subroutinesub(x)
```

will succeed and returning kind=SUBROUTINE.

## 4.5 Keeping track of indents

In `indent_and_output()` (`fortran.cpp`), a stack is maintained containing the indents, along with the current index. The actions are in principle quite simple: if after parsing a relevant keyword is found (SUBROUTINE, DO, ...) the indent is changed as appropriate and put on the stack. If a kind of END (ENDIF, END SUBROUTINE, ...) is found, the indent is pulled from the stack.

Some constructs deserve extra attention:

- Labelled DO-loops: if a labelled DO-loop is encountered, the label involved is stored on a stack. When a corresponding statement label is encountered, appropriate action is taken, also in the case of nested DO-loops sharing to the same label.<sup>2</sup>
- MODULE PROCEDURE statements: at encountering a MODULE PROCEDURE, indentation if the next full statement is classified as an executable statement.

- An ambiguity:

```
MODULEPROCEDUREmyproc
```

Should this be interpreted as:

```
MODULE PROCEDUREmyproc
```

or:

```
MODULE PROCEDURE myproc
```

*Findent* assumes the last is correct.<sup>3</sup>

## 4.6 Handling cpp and coco preprocessor statements

It was a design goal that *findent* should handle macro's more or less intelligent.

For example:

Input	Desired	Not desired
<pre>#ifdef DIM2 do y=1,ny #else do y=1,1 #endif do x=1,nx call s(x,y) enddo enddo</pre>	<pre>#ifdef DIM2 do y=1,ny #else do y=1,1 #endif do x=1,nx call s(x,y) enddo enddo</pre>	<pre>#ifdef DIM2 do y=1,ny #else do y=1,1 #endif do x=1,nx call s(x,y) enddo enddo</pre>

<sup>2</sup>Shared DO-termination is flagged as a 'deleted feature' by gfortran.

<sup>3</sup>This ambiguity arises from the fact that all spaces are removed in the preprocessing phase. In fixed format (where spaces do not count), this ambiguity is also present for the compiler.



The following preprocessor statements (defined in `lexer.1`) are recognized:

cpp	coco
# if	?? if
# endif	?? endif
# else	?? else
# elif	?? elseif
# include "..."	?? include "..."
# include <...>	?? include <...>

Note1: the rest of the preprocessing line is ignored, so, for example, `#if` has the same effect as `#ifdef`.

Note2: the `include`'s are only used when generating dependencies, and are ignored when indenting.

Most of the preprocessor-handling code is reached via `handle_pre()` in `Fortran.cpp` and `Pre_analyzer()` in `pre_analyzer.cpp`.

The strategy is as follows:

- A stack is maintained to store the relevant items (e.g. the indentation level and the stack of indentations) (see `push_all()`, `top_all()` and `pop_all()` in `fortran.h`).
- The relevant items are pushed on this stack after `#if`.
- The relevant items are popped off the stack if appropriate after `#endif`, `#else` and `#elif`.
- Handling the preprocessor statements is done recursively.
- After a construct like

```
#if ...
<fortran statements>
#endif
```

the indentation continues from the state before the `#if`, but after a construct like

```
#if ...
<fortran statements>
#else
<fortran statements>
#endif
```

the indentation continues from the state after the `#else`.

In this way, most of the times *findent* will generate sensible indentation. If *findent* makes an error, this can easily be fixed by inclusion of a `findentfix` statement, for example (admittedly somewhat constructed):

Original	Corrected
<pre> program p #ifdef A     do i=1,10 #else     i = 1 #endif     x = x+i #ifdef A     enddo #endif end </pre>	<pre> program p #ifdef A     do i=1,10 #else     i = 1 #endif     x = x+i #ifdef A     !findentfix: do         enddo     endif end </pre>

## 4.7 Relabelling

Relabelling (renumbering of labels) is done in the following stages:

1. Scan the input until a complete program unit (PROGRAM, SUBROUTINE, FUNCTION) is obtained, collecting the defined and used labels.
2. Regenerate the program unit, now with the renumbered labels.
3. Indent and output the renumbered program unit.
4. Go to step 1.

If some error is detected, (not defined label, label spanning continuation lines, ...) relabelling is abandoned for the current and following program units, however, indentation proceeds as normal. If relabelling fails, one can run *findent* with the flag `--query-relabel`, to see the reason of failure.

## 4.8 Generate miscellaneous text files

Help files, man page, scripts for usage in editors *etc.* are generated in the file `src/docs.cpp`. For generating the man page and the help-file, the function `manout()` is used so that generating these files is based on the same input.

The other files are include files, generated from the original text files. For example: `vim_fortran.inc` is generated from `vim/fortran.vim`, using the script `src/tocpp`. Details are available in the file `src/Makefile.am`.

## 5 Self replication

*Findent* has the capability to output a tar ball containing the complete source.<sup>4</sup> The method used is to create an include file for `src/selfrep.cpp` based on the output of `make dist`. See `src/Makefile.am` for details. An issue is to maintain

<sup>4</sup>This can be disabled by giving the flag `--disable-selfrep` to configure.

a reproducible build (see <https://reproducible-builds.org/>), because the tar ball contains time stamps for the containing files. This problem is solved by modifying the standard code to produce a tar ball. Normally, the file `bootdate`, created by `bootstrap` is used as time stamp. Most of the code is contained in `configure.ac`.

## 6 Copyright

*Findent* comes with the BSD-3 license:

---

Copyright: 2015-2025 Willem Vermin

License: BSD-3-Clause

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

---