

1 F95 to T_EXconverter

The `f95totex` program converts free-format Fortran programs with T_EX-coded comments into typesettable L^AT_EX form, with the actual program lines formatted using the `listings` package. The program lines are numbered according to the position in the F95 file, counting comment lines.

2 Module: filespecs

This module contains the relevant numbers and data for the input and output files.

```
13 module filespecs
14   implicit none
```

Maximum line length

```
18   integer, parameter :: MAXLEN = 200
```

Input and output file numbers.

```
22   integer :: infile
23   integer :: outfile
24
25 end module
```

3 Module: texfileparts

This module handles writing of the various parts of the T_EX output file.

```
29 module texfileparts
30   use filespecs
31   implicit none
```

The current file state is tracked via a set of modes. At present, there are only two modes, T_EX output and F95 output. (Line skipping before F95 code is handled by leaving the blank lines in T_EX mode; `listings` skips blank lines after F95 code.)

```
38   integer, parameter :: texmode = 0
39   integer, parameter :: f95mode = 1
40
41   integer :: currentmode
```

The current line number in the F95 file is tracked in a module variable.

```
45   integer :: currentlinenumber
46
47 contains
```

3.1 Subroutine: writeheader

The header is written as a single preplanned lump.

```
51   subroutine writeheader(outfile)
52     integer, intent(in) :: outfile
```

We use the standard `article` class, with enlarged margins. The `amsmath` package is available for in-comment mathematics. The actual program listings are set with the `listings` package. The `textcomp` package is needed for upright single quotes.

```

59   write( outfile, ' (A)' ) "\documentclass[twoside]{article}"
60   write( outfile, ' (A)' ) "\usepackage{geometry}"
61   write( outfile, ' (A)' ) "\geometry{left=1in,right=1in,top=1in,bottom=1in}"
62   write( outfile, ' (A)' ) "\usepackage{amsmath}"
63   write( outfile, ' (A)' ) "\usepackage{textcomp}"
64   write( outfile, ' (A)' ) "\usepackage{listings}"

```

The default Fortran 95 language file does not include strings delimited by single-quote characters. We define a new language spec that includes them.

```

70   write( outfile, ' (A)' ) "\lstdefinlanguage{MyFortran}[95]{Fortran}%"
71   write( outfile, ' (A)' ) "\_{}_morestring=[d]'"

```

All “actual code” listings are set in the `standardcode` style, as defined here.

```

76   write( outfile, ' (A)' ) "\lstdefinestyle{standardcode}{%"
77   write( outfile, ' (A)' ) "\_{}_language=MyFortran,"
78   write( outfile, ' (A)' ) "\_{}_columns=flexible,"
79   write( outfile, ' (A)' ) "\_{}_xleftmargin=36pt,"
80   write( outfile, ' (A)' ) "\_{}_numbers=left,"
81   write( outfile, ' (A)' ) "\_{}_numberstyle={\tiny},"
82   write( outfile, ' (A)' ) "\_{}_identifierstyle={\itshape},"
83   write( outfile, ' (A)' ) "\_{}_stringstyle={\ttfamily},"
84   write( outfile, ' (A)' ) "\_{}_texcl=true,"
85   write( outfile, ' (A)' ) "\_{}_upquote=true,"
86   write( outfile, ' (A)' ) "\_{}_commentstyle={}"
87   write( outfile, ' (A)' ) "%}"

```

Definitions for program element macros. (This could be improved.)

```

91   write( outfile, ' (A)' ) "\newcommand{\module}[1]"
92   write( outfile, ' (A)' ) "\_{}_{\section{Module:\_{}\_{}texttt{#1}}}"
93   write( outfile, ' (A)' ) "\newcommand{\program}[1]"
94   write( outfile, ' (A)' ) "\_{}_{\section{Program:\_{}\_{}texttt{#1}}}"
95   write( outfile, ' (A)' ) "\newcommand{\subroutine}[1]"
96   write( outfile, ' (A)' ) "\_{}_{\subsection{Subroutine:\_{}\_{}texttt{#1}}}"
97   write( outfile, ' (A)' ) "\newcommand{\function}[1]"
98   write( outfile, ' (A)' ) "\_{}_{\subsection{Function:\_{}\_{}texttt{#1}}}"

```

We define `\code` as a short command for inline code. Also, we define a `\var` command for variables only, which can be included in end-of-line comments without causing problems by nesting listings code. For variables, the only objectionable character is the underscore, which we handle by making it active – which causes it to call the `_{}_` macro that already exists. The standard indirection is required to set the `catcode` before the argument is processed.

```

108   write( outfile, ' (A)' ) "\newcommand{\code}{\lstinline[style=standardcode]}"
109   write( outfile, ' (A)' ) "\newcommand{\var}{\bgroup\catcode`\_{}=13\relax\doovar}"
110   write( outfile, ' (A)' ) "\newcommand{\doovar}[1]{\itshape\_{}#1\egroup}"
111   ! A test of var:  $x_{yz}$  and math:  $x^2 + y^2$  in code comments.

```

Page headings, using the standard `headings` style.

```

115   write( outfile, ' (A)' ) "\pagestyle{headings}"

```

And, finally, the beginning of the document.

```

119   write( outfile, ' (A)' ) "\begin{document}"

```

Set current state as appropriate for the beginning of a new F95 file.

```

123   call startf95file ( outfile )
124
125   end subroutine

```

3.2 Subroutine: *switchtotex*

Switch from F95 mode to T_EX mode.

```
130  subroutine switchtotex(outfile)
131      integer, intent(in) :: outfile
```

`lstlisting` environments should only be closed if they are currently open. We insert a blank line after the listing in order to start a new paragraph, and set the new paragraph to `\noindent`.

(Note that we have to break up the `lstlisting` name in order to avoid triggering the end of a listing when typesetting this program!)

```
142      if (currentmode == f95mode) then
143          write(outfile, '(A)') "\end{lst" // "listing}"
144          write(outfile, '(A)') ""
145          write(outfile, '(A)') "\noindent"
146      end if
147
148      currentmode = texmode
149
150  end subroutine
```

3.3 Subroutine: *switchtof95*

Switch from T_EX mode to F95 mode.

```
154  subroutine switchtof95(outfile)
155      integer, intent(in) :: outfile
```

`lstlisting` environments should only be started if we are not currently in one.

```
160      if (currentmode /= f95mode) then
161          write(outfile, '(A)') "\begin{lstlisting}[%]"
162          write(outfile, '(A,I,A)') "\_firstnumber={", currentlinenumber, "},"
163          write(outfile, '(A)') "\_style=standardcode]"
164      end if
165
166      currentmode = f95mode
167
168  end subroutine
```

3.4 Subroutine: *writefooter*

Finishes out the document.

```
173  subroutine writefooter(outfile)
174      integer, intent(in) :: outfile
```

Close out any open `lstlisting` environment.

```
178      call switchtotex( outfile )
```

End the document.

```
182      write(outfile, '(A)') "\end{document}"
183
184  end subroutine
```

3.5 Subroutine: startf95file

Resets the file state as appropriate for a new F95 file.

```
189  subroutine startf95file( outfile )
190      integer, intent(in) :: outfile
```

Close any open `lstlisting` environment.

```
193      call switchtotex( outfile )
```

Reset the file number counter.

```
196      currentlinenumber = 0
```

Write a header for the new file, if appropriate. (Not yet implemented.)

```
200  end subroutine
```

3.6 Subroutine: dof95line

Takes in a line from the F95 file and processes it as appropriate.

```
205  subroutine dof95line(outfile, programline)
206      integer, intent(in) :: outfile
207      character (len=MAXLEN), intent(in) :: programline
208      integer :: i
```

Increment the current line number.

```
211      currentlinenumber = currentlinenumber + 1
```

If the line is blank, we should just print it out without switching mode.

```
216      if (len_trim(programline) == 0) then
217          write(outfile, '(A)') ""
```

Check for a commented `TeX`-mode line, and print out the comment line without the preceding comment character.

```
222      elseif (programline(1:2) == '!␣') then
223          call switchtotex( outfile )
224          write(outfile, '(A)') trim(programline(3:MAXLEN))
```

If neither of the above are true, the line is a line of code, and should be treated appropriately.

This may eventually include extra parsing to check for module and function definitions and the like, and include automatic section headers and such for those.

```
233      else
234          call switchtof95( outfile )
235          write(outfile, '(A)') trim(programline)
236      endif
237
238  end subroutine
239
240 end module
```

4 Module: `commandline_filenames`

This module inspects the command line for input and output filenames, and returns them.

```
246 module commandline_filenames
```

Use the Intel Fortran POSIX routines (for command-line arguments)

```
249 use ifposix
250 use filespecs
251 implicit none
```

We define the filename variables as module variables, for simplicity (so they don't have to be declared twice).

```
256 integer :: n_infiles
257 character (len=MAXLEN), dimension(:), allocatable :: infile_names
258 character (len=MAXLEN) :: outfile_name
```

File unit numbers for currently connected input and output files.

```
262 integer :: unit_infile , unit_outfile
263
264 contains
```

4.1 Subroutine: `getfilenames`

Gets the file names from the command line.

```
268 subroutine getfilenames
269
270 character (len=MAXLEN) :: currentarg
271 integer :: arglen, ierror
272 integer :: i, i_currentarg
```

Check that there are arguments, and set unit numbers to the standard input and output if not.

```
277 if (ipxfargc() == 0) then
278   n_infiles = 1
279   unit_infile = 5
280   unit_outfile = 6
281 return
282 end if
283
284   i_currentarg = 1
```

Get the first argument.

```
288 call pxfgetarg(i_currentarg, currentarg, arglen, ierror)
```

Check for whether the first argument is a “-o” flag, and retrieve the file name from the following argument.

```
293 if (currentarg(1:2) == '-o') then
```

Error checking: the argument should be simply “-o”, with nothing after.

```
298 if (arglen>2) then
299   write(*,*) "Unknown option: " // trim(currentarg)
300 stop
301 end if
```

Retrieve outfile name.

```

305     call pxfgetarg(i_currentarg + 1, outfile, arglen, ierror)
306     if (ierror > 0) then
307         write(*,*) "Error obtaining output file name."
308         stop
309     end if
310     i_currentarg = i_currentarg + 2

```

Append “.tex” to outfile name if it has no extensions.

```

314     if (index(outfile, ".") == 0) then
315         outfile = trim(outfile) // ".tex"
316     end if

```

Alternately, set outfile name to blank, for filling in later.

```

318     else
319         outfile = ""
320     end if

```

Get count of input file names

```

324     n_infiles = ipxfargc() - i_currentarg + 1
325     allocate(infilenames( n_infiles ))

```

Read input file names from the command line.

```

329     do i=1, n_infiles
330         call pxfgetarg(i_currentarg, infilenames(i), arglen, ierror)
331         i_currentarg = i_currentarg + 1

```

Append “.f90” to infile name if it has no extensions.

```

335     if (index(infilenames(i), ".") == 0) then
336         infilenames(i) = trim(infilenames(i)) // ".f90"
337     end if
338 end do

```

If outfile name is undefined, define it from the first infile name.

```

342     if (len_trim(outfile) == 0) then
343         outfile = infilenames(1)
344         outfile = outfile(1:index(outfile, '.', back=.true.)-1) // ".tex"
345     end if

```

At this point, we have an outfile; we may or may not have an infile. We set the unit numbers appropriately.

```

350     if ( n_infiles > 0) then
351         unit_infile = 10
352     else
353         unit_infile = 5
354     endif
355     unit_outfile = 11
356 end subroutine

```

4.2 Function: connectinputfile

Connects the *i*th input file name to a file handle, and returns the handle. If *i* is zero, this function simply closes the file.

```

362     function connectinputfile( i_infile ) result( u_infile )
363         integer, intent(in) :: i_infile
364         integer :: u_infile

```

If the infile unit is 5 (standard input), we do nothing but return the unit number.

```

369     if ( unit_infile == 5) then
370         u_infile = 5
371     return
372 end if

```

We close the input file handle. (If it's not open, this is a no-op.)

```

376     close( unit_infile )

```

Open the new infile, if within range, and return the unit number; else, return zero.

```

381     if (( i_infile > 0) .AND. ( i_infile <= n_infiles)) then
382         open ( unit_infile, file = infilenames( i_infile ), &
383             action = 'read', status = 'old' )
384         u_infile = unit_infile
385     else
386         u_infile = 0
387     end if
388 end function

```

4.3 Function: connectoutputfile

Connects the output file name to a file handle, and returns the handle.

```

393     function connectoutputfile result( u_outfile )
394     integer :: u_outfile

```

If the outfile unit is 6 (standard output), we do nothing but return the unit number.

```

399     if ( unit_outfile == 6) then
400         u_outfile = 6
401     return
402 end if

```

We close the output file handle. (If it's not open, this is a no-op.)

```

406     close( unit_outfile )

```

Open the new outfile and return the unit number.

```

410     open ( unit_outfile, file = outfilename, &
411         action = 'write', status = 'replace' )
412     u_outfile = unit_outfile
413 end function
414 end module

```

5 Program: f95totex

The main program handles opening the input and output files, and reading the individual lines and passing them to dof95line.

```

420 program f95totex
421
422     use filespecs
423     use texfileparts
424     use commandline_filenames
425     implicit none

```

```

426
427   character (len=MAXLEN) :: programline
428   integer :: i

```

Collect filenames from the command line.

```

432   call getfilenames

```

Open output file, as appropriate.

```

436   outfile = connectoutputfile()

```

Start T_EXfile.

```

440   call writeheader( outfile )

```

Loop over input files.

```

444   do i=1, n_infiles
445     infile = connectinputfile( i )
446
447     call startf95file ( outfile )

```

Loop until end of input file, reading lines and processing them.

```

451     10 continue
452       read( infile, '(A)', end=20) programline
453       call dof95line( outfile, programline )
454       goto 10
455       20 continue ! end of file.
456     enddo

```

Finish T_EXfile.

```

460     call writefooter( outfile )
461
462     end program

```