A keyword based input facility for FORTRAN programs

Friedemann Schautz

October 25, 2018

Abstract

An hopefully easy-to-use and easy-to-extend input facility written in FORTRAN77 is briefly described. It is meant to provide an human readable uniform structure to the input of typical numerical applications. Keyword, namelist-like constructions, macro-like variables, comments and input nesting are supported.

1 User's view

1.1 General structure

The typical input consists of individual lines, which either contain a keyword, optionally followed by arguments, a macro definition, a comment, or namelist like input. In cases where a keyword is associated with a large number of values, additional lines are used for this data, usually ended by the word 'end' on a separate line. A backslash (\) at the end of a line causes it to be joined with the following one and to be treated as a single line. Keywords and macro names *are* case-sensitive.

1.2 Nesting

The keyword `load` *`filename`* causes the file *filename* to be parsed ('included') before processing the of current input resumes. At most eight levels of nesting are allowed (can be changed).

1.3 Special lines

One line comments start with a hash sign (#) and are ignored, as are blank lines. Lines starting with double quotes are simply printed out after macros have been expanded. Some special commands can be used to control the behaviour of the interpreter, they all start with a dot followed by some capital letters.

1.4 Keywords

Keywords are used to trigger some action of the program and pass argument values. (In fact, they are almost equivalent to subroutine calls – see Programmer's view below). Arguments are separated by blanks. Both keyword and arguments can contain macros, which will be expanded first. Arguments can have default values, which are used if the argument is omitted. Omitting arguments is possible from right to left. Keyword names can be abbreviated, provided the abbreviation is not ambiguous.

1.5 Macros

Macros are defined using the special keyword `def` followed by the name of the macro and its definition, e.g.

```
def n 7
def ci-file ci.dat
def parameters 1 2 3 4.0
```

and referred to by affixing a dollar sign, where multi-letter names have to be enclosed in brackets. Examples

```
print $n
update $(parameters)
read-file $(ci-file).$n
```

(In the last example, `ci.dat.7` would be passed to `read-file`) The command `printmacros` prints a list of all currently known macros.

1.6 Namelist like input

It is possible to 'group' macros using the notation

```
&groupname name1 value1 name2 value2  ...  nameN valueN
```

which is equivalent to saying

```
 def groupname:name1 value1
 def groupname:name2 value2
           ...
```

1.7 Indexed macro names

Single letter macros, like `$i` can occur inside multi-letter macro names, so that with `def ab7 100.0` and `def i 7` the macro `$(ab$i)` would expand to `100.0`. Together with loops this can be used to iterate over files, parameter sets, and the like.

1.8 Loops

Simple iterations are supported (vaguely FORTRAN 'do-loops') with the syntax

```
 loop var lower_bound upper_bound
    ...
 end
```

The loop variable is accessible as a normal macro, and the loop body is re-interpreted in each iteration, i.e. macros are expanded again.

1.9 Arithmetics on macros

Macros can be added, subtracted , muiliplied, and divided, provided they expand to numbers. A 'csh-like' @-notation is used. Examples (the blanks are important!):

```
@ i = $i + 1
@ n = $i * 1000
```

2 Programmer's view

2.1 Names

Names of routines and common blocks of this input facility usually start with `p2` (historically for parser 2).

2.2 The read-execute loop

Unlike a 'conventional' FORTRAN program having subroutine calls chained together in a more or less fixed way, the keyword-driven approach amounts to executing a loop which reads a keyword from the input, executes an associated routine, reads the next keyword and so on. This loop is terminated be either end-of-file on the input file or a special keyword like `quit`.

A main program using 'p2' should roughly look like this

```
      program foo
c       ... own initialisation ...

c     init the parser
      call p2init
c     optional call to echo input to output
      call echo(iuin,iuout)
c     the read-execute loop
      call p2go(iuin,0)
c     ... own cleaning up ..
      end
```

2.3 Introducing new keywords

A keyword together with its arguments – either those from input or default values if arguments were omitted – is mapped to a subroutine call (inside the automatically generated subroutine p2call). To indicate that a subroutine corresponds to a keyword, a special comment starting with `C$INPUT` is used, like in the following example:

```
      subroutine foosub
C$INPUT foo
CKEYDOC This keyword is an example and does nothing
      .....
      end
```

Here a new keyword (`foo`) would be created and the subroutine `foosub` would be called (with no arguments) if `foo` occurred in the input. The optional doc comment can be used to attach a brief documentation to be extracted by a utility program (`vardoc`) which generates a pretty-printed list of available keywords. A subroutine can be mapped to different keywords, e.g. with different default values for arguments. See the next example:

```
      subroutine fooarg( ibar, astring, anumber, iflag)
C$INPUT fooa i a d i
CKEYDOC calls fooarg with 4 arguments from input, no defaults
C$INPUT foob i a=hello d=100.d0 i=1
CKEYDOC calls fooarg , 3 arguments having defaults
C$INPUT fooc i a=hello d=100.d0 0
CKEYDOC calls fooarg , 2 arguments having defaults, one the fixed value 0
C$INPUT food inp a=hello d=100.d0 0
CKEYDOC calls fooarg like  fooc, but the current input unit
CKEYDOC (the one the keyword is read from) is passed as the first argument
      integer ibar, iflag
      character astring*(*)
      double precision anumber
        ...
     end
```

As the example shows, arguments are denoted by a letter showing their type (i integer, d double, a string) optionally followed by =default. If one argument has a default value, all arguments right to it must have default values as well (since arguments can only be omitted from right to left). Fixed values for arguments can appear anywhere, implying that the input keyword takes less arguments that the associated subroutine. If there is a mismatch between the number and/or types of arguments between the definition and the use in the input, the interpreter will signal an error.

2.4 Preprocessors and files

A preprocessor (`genp2_defaults.awk`) is used to create the routines `p2init,` `p2inid` and `p2call` from the special comments. These routines are typically located in a file called `p2prog.F`. The mappings between keywords and routines are assembled in a file called `commands.p2`. The parser code itself resides in two file called `p2_defaults.F` and `p2etc.F`. The script `vardoc` can be used to extract keywords and documentation comments to produce a (latex or ascii) list.

2.5 Accessing macros from within the program

All macros defined with `def` or the name list like input can be accessed from within the program, they can be modified as well and new macros can be created. The two routines used for this purpose are `p2setv` to define a macro (or change the value of an existing one) and `p2var` to retrieve a value. In both cases the values are character strings. For the common case where macros (parameters etc) are supposed to contain numerical values, two routines are available to retrieve these numbers, `p2gtid` and `p2gtfd`, for integer and floating point (double precision) values. Here is an example how to use `p2gtid`, `p2gtfd` being analogous.

```
      call p2gtid('qmc:walkers',nwalker,100,1)
CVARDOC Number of walkers
```

The first argument is the name of the macro. In the example above `walkers` is a 'member' of a namelist group (`qmc`), so it could be given in the input by

```
&qmc walkers 500
```

or

```
def qmc:walkers 500
```

The next argument is the corresponding FORTRAN variable, followed by a default value and a flag, specifying what should happen if the macro is not defined when p2gtid is called. Possible values are: 0 use default value if the macro is not present, -1 raise an fatal error if the macro is not present, 1 use default value and define the macro with this value. In cases 0 and 1 a warning will be issued when the default value is used. The optional documentation following `CVARDOC` can again be extracted with the `vardoc` utility.

2.6 Routines that read their own input

Sometimes a keyword needs a lot of data and it should appear in the input rather than in a separate file. In such a case the current input unit can be passed as a keyword argument (unsing `inp` instead of the type specifier, as in the foo-example above) and the routine can continue reading its own data. To be consistent with the rest it should, however, not use `read(..)` but a call to `gpnln2`. This routine will read a line from the input (or some other FORTRAN unit), expand macros and spilt the line into fields (much like awk). Self-read input should be terminated by the word 'end' on a single line. The functions `ifroms` and `dfroms` can be used to convert the output of `gpnln2` to intergers and double precision floats. The following code fragment demonstrates this use. Assuming an fictitious input fragment to read in some square matrix

```
  read_sqmat 3
   1.0 2.0 3.0
   4.0 5.0 6.0
   7.0 8.0 9.0
  end
```

the corresponding routine could look like

```
      subroutine rdsqmt(iu,ndim)
C$INPUT read_sqmat inp i
        ....
      include 'inc/p2_dim.inc'
      include 'inc/p2.inc'
      dimension idx1(MXF),idx2(MXF)
        ....
      do i=1,ndim
```

```
    call gpnln2(iu,il,lne,MXLNE,idx1,idx2,nf,lbuf,'rdsqmt')
    if(nf.ne.ndim) then
      call fatal('read_sqmat: wrong number of elements')
    else
      do  j=1,ndim
        a(j,i)=dfroms(lne(idx1(j):idx2(j)),1)
      enddo
    endif
  enddo
  call gpnln2(iu,il,lne,MXLNE,idx1,idx2,nf,lbuf,'rdsqmt')
  if((nf.ne.1).or.(lne(idx1(1):idx2(1)).ne.'end')) then
    call fatal('read_sqmat: <end> expected')
  endif
    ....
  end
```

Here `iu` is the current input unit , `lne` is a buffer holding the input line being processed, nf is the number of fields in the line after splitting it on whitespace, `idx1` points to the beginnings of these fields, `idx2` to their ends. Comments and blank lines are 'eaten' by `gpnln2`, so subsequent call result in new 'data-lines'. Mentioning the name of the keyword in the call to `gpnln2` is just to include it in error messages possibly arising while the line is processed.

2.7 Using 'file' instead of 'open'

The interpreter uses calls to `file` to open input files. This routine *assigns* FORTRAN unit numbers and keeps track of them. It is possible to limit the unit numbers `file` assigns by changing `MINUNT` and `MXUNT` in `files.inc`. As an alternative, calls to `open` can be replaced by calls to `file`.

3 Miscellaneous

3.1 Changing the interpreter's behaviour

The following directives are understood: (all beginning with a dot!) Default settings marked with a star.

directive	meaning
.VERSION	print version of the interpreter
.ABBREV	keywords can be abbreviated *
.NOABBREV	keywords must be spelled out
.SEP	print the keyword and a separator before calling *
.NOSEP	dont
.NODEFAULTS	dont allow omitting keyword arguments with defaults
.DEFAULTS	allow omitting keyword arguments *
.VARDEF_SILENT	use default values in macro retrievals without notice
.VARDEF_WARN	issue warnings if undefined macros are retrieved *
.VARDEF_ERROR	make the above fatal errors
.DEBUG	print (lots of) debugging info
.NODEBUG	dont print debugging info *
.COMMENTSc	change the character which starts one line comments to some value c (if the hash sign is a bad choice)
.ONLYc	if set, *only* lines starting with some character c will be parsed
.ALL	reset from .ONLY

3.2 Utility keywords

printmacros	print list of currently defined macros
savemacros file	save macros as def-statements to file
skipto x	ignore input until it contains xxx
loop var a b [s]	loop from a to b (step s if given)
? or ??	print list of keywords (? short, ?? long format)
load file	load file (nested input)
@@	floating point operation
@	integer operation