NAME

ftnchek − Fortran 77 program checker

SYNOPSIS

ftnchek [**−arguments**[=*list*]] [**−array**[=*list*]] [**−**[**no**]**brief**] [**−calltree**[=*list*]] [**−**[**no**]**check**] [**−columns**[=*num*]] [**−common**[=*list*]] [**−**[**no**]**crossref**[=*list*]] [**−**[**no**]**declare**] [**−**[**no**]**division**] [**−errors**[=*num*]] [**−**[**no**]**extern**] [**−**[**no**]**f77**[=*list*]] [**−**[**no**]**f90**[=*list*]] [**−**[**no**]**f95**[=*list*]] [**−**[**no**]**help**] [**−**[**no**]**identifier−chars**[=*list*]] [**−include=***str*] [**−intrinsic**[=*list*]] [**−**[**no**]**library**] [**−**[**no**]**list**] [**−makedcls**[=*list*]] [**−mkhtml**[=*list*]] [**−**[**no**]**novice**] [**−output=***str*] [**−pointersize**[=*num*]] [**−**[**no**]**portability**[=*list*]] [**−**[**no**]**pretty**[=*list*]] [**−project**[=*list*]] [**−**[**no**]**pure**] [**−**[**no**]**quiet**] [**−**[**no**]**reference**] [**−**[**no**]**resources**] [**−**[**no**]**sixchar**] [**−**[**no**]**sort**] [**−source**[=*list*]] [**−style**[=*list*]] [**−**[**no**]**symtab**] [**−**[**no**]**truncation**[=*list*]] [**−usage**[=*list*]] [**−**[**no**]**vcg**] [**−**[**no**]**version**] [**−**[**no**]**volatile**] [**−wordsize**[=*num*]] [**−wrap**[=*num*]] [*files ...*]

DESCRIPTION

ftnchek (short for Fortran checker) is designed to detect certain errors in a Fortran program that a compiler usually does not. **ftnchek** is not primarily intended to detect syntax errors. Its purpose is to assist the user in finding semantic errors. Semantic errors are legal in the Fortran language but are wasteful or may cause incorrect operation. For example, variables which are never used may indicate some omission in the program; uninitialized variables contain garbage which may cause incorrect results to be calculated; and variables which are not declared may not have the intended type. **ftnchek** is intended to assist users in the debugging of their Fortran program. It is not intended to catch all syntax errors. This is the function of the compiler. Prior to using **ftnchek**, the user should verify that the program compiles correctly.

This document first summarizes how to invoke **ftnchek**. That section should be read before beginning to use **ftnchek**. Later sections describe **ftnchek**'s options in more detail, give an example of its use, and explain how to interpret the output. The final sections mention the limitations and known bugs in **ftnchek**.

INVOKING FTNCHEK

ftnchek is invoked through a command of the form:

```
$ ftnchek [-option -option ...] filename [filename ...]
```

The brackets indicate something which is optional. The brackets themselves are not actually typed. Here options are command-line switches or settings, which control the operation of the program and the amount of information that will be printed out. If no option is specified, the default action is to print error messages, warnings, and informational messages, but not the program listing or symbol tables.

Each option begins with the '−' character. (On VAX/VMS or MS-DOS systems you may use either '/' or '−'.) For the sake of conformity with an increasingly common convention, options can also begin with '−−'. The options are described at greater length in the next section.

ftnchek options fall into two categories: switches, which are either true or false, and settings, which have a numeric or string value. The name of a switch is prefixed by 'no' or 'no-' to turn it off: e.g. **−nopure** would turn off the warnings about impure functions. The 'no' prefix can also be used with numeric settings, having the effect of turning off the corresponding warnings. Settings that control lists of warnings have a special syntax discussed below. Only the first 3 characters of an option name (not counting the '−') need be provided. A colon may be used in place of an equals sign for numeric or string setting assignments; however, we show only the equals sign form below.

The switches and settings which **ftnchek** currently recognizes are listed below. For each option, the default is the value used if the option is not explicitly specified, while the turn-on is the value used if the option is given without assigning it a value.

−arguments=*list*

Control warnings about subprogram type and argument mismatches. Default = turn-on = all.

−array=*list*
Control warnings in checking array arguments of subprograms. Default = turn-on = all.

−brief Use shorter format for some error messages. Default = no.

−calltree=*list*
Produce subprogram call hierarchy in one of 3 formats: text call-tree, who-calls-who and VCG. Default = none, turn-on = tree,prune,sort.

If the **−mkhtml** option is invoked *and* tree is the applied calltree option, a file named `CallTree.html`, will be produced depicting the tree in HTML format.

−check Perform checking. Default = yes.

−columns=*num*
Set maximum line length to *num* columns. (Beyond this is ignored.) Turn-on = max = 132. Default = 72.

−common=*list*
Set degree of strictness in checking COMMON blocks. Default = turn-on = all.

−crossref=*list*
Print cross-reference list of subprogram calls, label usage, and/or COMMON block use. Default = none.

−declare
Print a list of all identifiers whose datatype is not explicitly declared. Default = no.

−division
Warn wherever division is done (except division by a constant). Default = no.

−errors=*num*
Set the maximum number of error messages per cascade. Default = turn-on = 3.

−extern
Warn if external subprograms which are invoked are never defined. Default = yes.

−f77=*list*
Control specific warnings about supported extensions to the Fortran 77 Standard. Default = none, turn-on = all.

−f90=*list*
Control specific warnings about supported extensions to the Fortran 77 Standard that were not adopted as part of the Fortran 90 Standard. Default = none, turn-on = all.

−f95=*list*
Control specific warnings about standard Fortran 77 features that were deleted from the Fortran 95 Standard. Default = none, turn-on = all.

−help Print command summary. Default = no.

−identifier−chars=*list*
Define non-alphanumeric characters that may be used in identifiers. Default = turn-on = dollar sign and underscore.

−include=*path*
Define a directory to search for INCLUDE files before searching in the system-wide directory. Cumulative. Default = turn-on = none.

−intrinsic=*list*
Control treatment of nonstandard intrinsic functions. Default = all except **vms** for Unix version, all except **unix** for VMS version, all except **unix** and **vms** for other versions. Turn-on = all.

−library
Begin library mode: do not warn about subprograms in file that are defined but never used. Default = no.

−list Print source listing of program. Default = no.

−makedcls=*list*
Prepare a file of declarations. The *list* specifies options for the format of this file. Default = none, turn-on = declarations.

−mkhtml=*list*
Create individual HTML document files from ftnchek analysis and code comments. Usually you will also want to specify **−call=tree** to create the root HTML file `CallTree.html`. Default = none, turn-on = documents.

−novice
Give output suitable for novice users. Default = yes.

−output= *filename*
Send output to the given file. Default and turn-on sends output to the screen. (Default filename extension is *. lis*).

−pointersize=*num*
Set the size of ''Cray pointer'' variables to *num* bytes. Min = 1, max = 16. Default = turn-on = 4

−portability=*list*
Warn about non-portable usages. Default = none, turn-on = all.

−pretty=*list*
Give warnings for possibly misleading appearance of source code. Default = turn-on = all.

−project=*list*
Create project file (see explanation below). Default = no.

−pure Assume functions are pure, i.e. have no side effects. Default = yes.

−quiet Produce less verbose output. Default = no.

−reference
Print table of subprograms referenced by each subprogram. Default = no.

−resources
Print amount of resources used in analyzing the program. Default = no.

−sixchar
List any variable names which clash at 6 characters length. Default = no.

−sort Print list of subprograms sorted in prerequisite order. Default = no.

−source=*list*
Select source formatting options: fixed or free form, DEC Fortran tab-formatted lines, VMS-style INCLUDE statement, UNIX-style backslash escape sequences, and implicit typing of parameters. Default = none, turn-on = all.

−style=*list*
Produce extra-picky warnings about obsolescent or old-fashioned programming constructions. Default = none, turn-on = all.

−symtab
Print symbol table and label table for each subprogram. Default = no.

−truncation=*list*
Check for possible loss of accuracy by truncation. Default = turn-on = all.

−usage=*list*
Control warnings about unused or uninitialized variables, common blocks, etc. Default = turn-on = all.

−vcg Produce VCG format of call graph.

−version
Print version number. Default = no.

−volatile
Assume COMMON blocks lose definition between activations. Default = no. (Obsolete. Use **−common=volatile** instead.)

−wordsize=*num*
Set the default word size for numeric quantities to *num* bytes. Default = turn-on = 4 bytes.

−wrap=*num*
Set output column at which to wrap long error messages and warnings to the next line. If set to 0, turn off wrapping. Default = turn-on = 79.

When more than one option is used, they should be separated by a blank space, except on systems such as VMS where options begin with slash (/). No blank spaces may be placed around the equals sign (=) in a setting. `ftnchek "?"` will produce a command summary listing all options and settings.

For settings that take a list of keywords, namely **−arguments**, **−array**, **−calltree**, **−common**, **−crossref**, **−f77**, **−f90**, **−f95**, **−intrinsic**, **−makedcls**, **−mkhtml**, **−portability**, **−pretty**, **−project**, **−source**, **−style**, **−truncation**, and **−usage**, the list consists of keywords separated by commas or colons. If the list of keywords is omitted, the effect is to set the option to its turn-on value (same as ‘‘all’’ in most cases). Also, if the list is omitted, the setting name can be prefixed with **no** or **no−** to turn off all the options it controls. For example, **−f77** turns on all warnings about nonstandard constructions, while **−nof77** turns them all off. Three special keywords are:

help Print out all the option keywords controlled by the setting, with a brief explanation of their meanings. This keyword cannot be given in a list with other keywords.

all Set all options. This turns on all options controlled by the setting.

none Clear all options. This turns off all options controlled by the setting.

These three special keywords must be given in full. For all other keywords, only as many letters of the keyword as are necessary to identify it unambiguously need be given, or a wildcard pattern may be used. Including a keyword in the list turns the corresponding option on. For example, **−f77=intrinsic** would turn on only the warnings about use of nonstandard intrinsic functions. Prefixing a keyword by **no−** turns its option off. For example, **−pretty=no−long−line** turns off warnings about lines exceeding 72 columns in length while leaving all other warnings about misleading appearance in effect. If a setting has default **none**, you can turn on all options except one or two by using **all** first. For example, **−f77=all,no−include** enables warnings about all nonstandard extensions except INCLUDE statements. If a setting has default **all**, you can turn off all warnings except one or two by using **none** first. For example, **−truncation=none,demotion** would turn off all precision related warnings except about demotions. Wildcard patterns contain an asterisk to stand for any string of characters. If a wildcard pattern is used, all the warnings that match it are affected. If **no−** is prefixed to the pattern, all the matching warnings are turned off, otherwise they are all turned on. The minimum unambiguous length rule does not apply to wildcard matching. For example, use **−usage=no−*var*** to turn off all warnings relating to variable usage (both local and common). (Unix users may need to quote any options containing wildcards in order to prevent the shell from attempting to expand them.) Wildcards are recognized only in lists of warning keywords, not in the top-level options themselves.

When **ftnchek** starts up, it looks for environment variables and also for a preferences file. Any options defined in the environment or in the preferences file are used as defaults in place of the built-in defaults. They are over-ridden by any command line options. See the section on changing the defaults for details about the environment options and the preferences file.

When giving a name of an input file, the extension is optional. If no extension is given, **ftnchek** will first look for a project file with extension *.prj*, and will use that if it exists. If not, then **ftnchek** will look for a Fortran source file with the extension *.for* for VMS systems, *.f* for UNIX systems. More than one file name can be given to **ftnchek**, and it will process the modules in all files as if they were in a single file.

Wildcards are allowed in the specification of filenames on the command line for the VMS and MS-DOS versions, as also of course under UNIX and any other system that performs wildcard expansion in the command processor.

If no filename is given, **ftnchek** will read input from the standard input.

OPTIONS

This section provides a more detailed discussion of **ftnchek** command-line options. Options and filenames may be interspersed on a command line. Most options are positional: each option remains in effect from the point it is encountered until it is overridden by a later change. Thus for example, the listing may be suppressed for some files and not for others. Exceptions are: the **−intrinsic**, **−pointersize**, and **−wordsize** settings, which cannot be changed once processing of input files has started; the **−arguments**, **−array**, **−calltree**, **−common**, **−crossref**, **−extern**, **−reference**, **−resources**, **−sort**, **−vcg**, and **−volatile** options, where the action depends only on the value of the option after the processing of input files is finished; and the **−include** setting, which is cumulative.

The option names in the following list are in alphabetical order.

−arguments=*list*

Controls warnings about mismatches between actual and dummy subprogram arguments, and also about mismatches between expected and actual subprogram type. (An actual argument is an argument passed to the subprogram by the caller; a dummy argument is an argument received by the subprogram.) By default, all warnings are turned on.

The *list* consists of keywords separated by commas or colons. Since all these warnings are on by default, include a keyword prefixed by **no−** to turn off a particular warning. There are three special keywords: **all** to turn on all the warnings about arguments, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−arguments** is equivalent to **−arguments=all**, and **−noarguments** is equivalent to **−arguments=none**. The warning keywords with their meanings are as follows:

arrayness:

warn about inconsistent use of arguments that are arrays. These warnings can be further controlled by the **−array** option.

type:

warn about dummy arguments of a different data type from the actual arguments.

function−type:

warn if the invocation assumes the function's return value is a different type than it actually is. Also warns if a function is called as a subroutine, or vice-versa.

number:

warn about invoking a subprogram with a different number of arguments than the subprogram expects.

For compatibility with previous versions of **ftnchek**, a numeric form of this setting is also accepted: the *list* is replaced by a number from 0 to 3. A value of 0 turns all the warnings off, 1 turns on only **number**, 2 turns on all except **number**, and 3 turns all the warnings on.

This setting does not apply to checking invocations of intrinsic functions or statement functions, which can only be turned off by the **−nocheck** option.

See also: **−array**, **−library**, **−usage**.

−array=*list*

Controls the degree of strictness in checking agreement between actual and dummy subprogram arguments that are arrays. The warnings controlled by this setting are for constructions that might legitimately be used by a knowledgeable programmer, but that often indicate programming errors. By default, all warnings are turned on.

The *list* consists of keywords separated by commas or colons. Since all these warnings are on by default, include a keyword prefixed by **no−** to turn off a particular warning. There are three special keywords: **all** to turn on all the warnings about array arguments, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−array** is equivalent to **−array=all**, and **−noarray** is equivalent to **−array=none**. The warning keywords with their meanings are as follows:

dimensions:

warn if the arguments differ in their number of dimensions, or if the actual argument is an array element while the dummy argument is a whole array.

size:

warn if both arguments are arrays, but they differ in number of elements.

For compatibility with previous versions of **ftnchek**, a numeric form of this setting is also accepted: the *list* is replaced by a number from 0 to 3. A value of 0 turns all the warnings off, 1 turns on only **dimensions**, 2 turns on only **size**, and 3 turns all the warnings on.

Note: A warning is always given regardless of this setting if the actual argument is an array while the dummy argument is a scalar variable, or if the actual argument is a scalar variable or expression while the dummy argument is an array. These cases are seldom intentional. (To turn off even these warnings, use **−arguments=no−arrayness**.) No warning is ever given if the actual argument is an array element while the dummy argument is a scalar variable. Variable-dimensioned arrays and arrays dimensioned with 1 or asterisk match any number of array elements. There is no check of whether multi-dimensional arrays agree in the size of each dimension separately.

See also: **−arguments**, **−library**, **−usage**.

−brief

Selects a shorter format for some warning messages. At present, the only warnings controlled by this flag are those that are printed at the end of processing each subprogram. These include warnings about variables that are set but not used or used before set, variable names that do not conform to the Fortran 77 standard, etc. (These warnings may be suppressed entirely depending on other flags, such as the **−usage** or **−f77** flags.) In the default format each variable is listed on a separate line, along with the line number where the variable is declared, set or used, according to the nature of the warning. The briefer format simply lists all variables to which the warning applies, with up to 4 variables per line.

See also: **−quiet**.

−calltree=*list*

Causes **ftnchek** to print out the call structure of the complete program.

The *list* consists of keywords separated by commas or colons. There are two special keywords: **none** to turn off all the options, and **help** to print the list of all the keywords with a brief explanation of each. (The keyword **all** turns on all the options, but should not normally be used since only one format should be specified.) If *list* is omitted, **−calltree** is equivalent to **−calltree=tree**, and **−nocalltree** is equivalent to **−calltree=none**. By default no call graph is printed.

If the -mkhtml option is invoked *and* tree is the applied calltree option, a file named `CallTree.html`, will also be produced depicting the tree in HTML format. This file is useful as a starting point for browsing the HTML files describing each component of the program.

The keywords which control which format is used are as follows:

tree:
produce the call graph in tree format.

reference:
produce the call graph in who-calls-who format (same as **−reference** switch).

vcg:
produce the call graph in VCG format (same as **−vcg** switch).

Only one of the formats **tree**, **reference**, or **vcg** may be specified.

The following keywords control options affecting the output:

prune:
prune repeated subtrees (applicable only with **tree**). This the default.

sort:
sort children of each routine into alphabetical order. This is the default.

See the discussion of the **−reference** and **−vcg** flags for details about these formats.

For **tree** format, The call graph is printed out starting from the main program, which is listed on the first line at the left margin. Then on the following lines, each routine called by the main program is listed, indented a few spaces, followed by the subtree starting at that routine.

In the default mode, if a routine is called by more than one other routine, its call subtree is printed only the first time it is encountered Later calls give only the routine name and the notice ‘‘(see above)’’. To have the subtree printed for each occurrence of the routine, use option **no−prune**.

Note that the call tree will be incomplete if any of the input files are project files containing more than one module that were created in **−library** mode. See the discussion of project files below.

Technical points: Each list of routines called by a given routine is printed in alphabetical order unless the **no−sort** option is given. If multiple main programs are found, the call tree of each is printed separately. If no main program is found, a report to that effect is printed out, and the call trees of any top-level non-library routines are printed. This flag only controls the printing of the call tree: **ftnchek** constructs the call tree in any case because it is used to determine which library modules will be cross-checked. See the discussion of the **−library** flag.

For compatibility with previous versions of **ftnchek**, a numeric form of this setting is also accepted: the *list* is replaced by a number from 0 to 15. This number is formed from 1 for **tree** format, 2 for **reference** format, or 3 for **vcg** format, plus 4 for **no−prune**, and 8 for **no−sort**.

See also: **−crossref**, **−library**, **−reference**, **−sort**, **−symtab**, **−vcg**.

−check

This switch is provided so that errors and warning messages can be turned off when **ftnchek** is used for purposes other than finding bugs, such as making declarations or printing the call tree. It is positional, so after turning all checks off, selected checks can be turned back on. The effect of **−nocheck** is to put all switches, numeric settings, and settings controlling lists of warnings to their turn-off values, as if they had all been specified with the −no prefix. Switches and settings that specify options and modes of operation, rather than controlling warnings, are unaffected. These are **−columns**, **−crossref**, **−include**, **−intrinsic**, **−library**, **−list**, **−makedcls**, **−novice**, **−output**, **−pointersize**, **−project**, **−quiet**, **−reference**, **−resources**, **−sort**, **−source**, **−symtab**, **−vcg**, **−version**, **−wordsize**, and **−wrap**. Default = yes.

Parse errors (syntax errors due to unrecognized or malformed statements) are not suppressed by this switch, since the results may be incorrect if **ftnchek** has not parsed the program correctly.

There are some miscellaneous errors and warning messages that are not controlled by any other switch, and so can be turned off only by this switch. Note that using **−check** following **−nocheck** only has the effect of turning these special warnings back on, and does not restore all the checks it turned off. These warnings are:

- o Module contains no executable statements.
- o In free source form, missing space where space is required (e.g. between a keyword and an identifier) or space present where none is allowed (e.g. within an identifier).
- o Zero or negative length specification in a data type declaration of the form type*len.
- o Array assigned to scalar.
- o Type mismatch between `DO` index and bounds.
- o Undefined common block declared in `SAVE` statement.
- o Intrinsic function explicitly declared with an incompatible type.
- o Unknown intrinsic function explicitly declared in an `INTRINSIC` statement.
- o Intrinsic function passed as a subprogram argument is not declared in an `INTRINSIC` statement.
- o Intrinsic function or statement function invoked incorrectly.
- o Function does not set return value prior to `RETURN` statement.
- o Parameter constant value not evaluated (this is **ftnchek**'s fault, and it is just informing you of the fact).
- o Entry point of a subprogram is later used as a different subprogram's name.
- o Unknown keyword used in an I/O statement.
- o Illegal label reference (e.g. GOTO refers to a non-executable statement; I/O statement refers to a non-format statement).

See also: **−errors**.

−columns=*num*

Set maximum statement length to *num* columns. (Beyond this is ignored.) This setting is provided to allow checking of programs which may violate the Fortran standard limit of 72 columns for the length of a statement. According to the standard, all characters past column 72 are ignored. If this setting is used when the **−f77=long−line** option is in effect, a warning will be given for any lines in which characters past column 72 are processed. Turn-on = max = 132. Default = 72.

This setting does not suppress warnings about the presence of characters beyond column 72. To process code with meaningful program text beyond column 72, use this setting and be sure the **−f77 long−line** option is off. To process code with sequence numbers in columns 73 to 80, leave the the columns setting at the default value and use the **−pretty=no−long−line** flag.

See also: **−f77**, **−pretty**.

−common=*list*

This setting controls the strictness of checking of COMMON blocks. By default, all warnings except **volatile** are turned on.

The *list* consists of keywords separated by commas or colons. Since most of these warnings are on by default, include a keyword prefixed by **no−** to turn off a particular warning. There are three special keywords: **all** to turn on all the warnings, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−common** is equivalent to **−common=dimensions,exact,length,type**, and **−nocommon** is equivalent to **−common=none**. The warning keywords with their meanings are as follows:

dimensions:

corresponding arrays in each declaration of a block must agree in size and number of dimensions. This option only has an effect when used together with **exact**.

exact:
the comparison of two blocks is done variable-by-variable rather than simply requiring agreement between corresponding storage locations. Use this if all declarations of a given COMMON block are supposed to be identical, which is a good programming practice.

length:
warn if different declarations of the same block are not equal in total length. The Fortran 77 Standard requires each named common block, but not blank common, to be the same length in all modules of the program.

type:
in each declaration of a given COMMON block, corresponding memory locations (words or bytes) must agree in data type. If used together with **exact**, this will require that corresponding variables agree in data type.

volatile:
Assume that COMMON blocks are volatile.

Many Fortran programmers assume that variables, whether local or in COMMON, are static, i.e. that once assigned a value, they retain that value permanently until assigned a different value by the program. However, in fact the Fortran 77 Standard does not require this to be the case. Local variables may become undefined between activations of a module in which they are declared. Similarly, COMMON blocks may become undefined if no module in which they are declared is active. (The technical term for entities with this behavior is ‘‘automatic’’, but **ftnchek** uses the word ‘‘volatile’’ since it is clearer to the nonspecialist.) Only COMMON blocks declared in a SAVE statement, or declared in the main program or in a block data subprogram remain defined as long as the program is running. Variables and COMMON blocks that can become undefined at some point are called volatile.

If the **−common=volatile** flag is turned on, **ftnchek** will warn you if it finds a volatile COMMON block. If, at the same time, the **−usage=com−block−volatile** option is turned on (which is the default), **ftnchek** will try to check whether such a block can lose its defined status between activations of the modules where it is declared. **ftnchek** does not do a very good job of this: the rule used is to see whether the block is declared in two separated subtrees of the call tree. For instance, this would be the case if two modules, both called from the main program, shared a volatile COMMON block. A block can also become undefined between two successive calls of the same subprogram, but **ftnchek** is not smart enough to tell whether a subprogram can be called more than once, so this case is not checked for.

The **−common=volatile** flag does not affect the way **ftnchek** checks the usage of local variables.

For compatibility with previous versions of **ftnchek**, a numeric form of this setting is also accepted: the *list* is replaced by a number from 0 to 3. A value of 0 turns all the warnings off, 1 or greater turns on **type**, 2 or greater turns on **length**, and 3 turns on **dimensions** and **exact** also. The numeric form cannot turn on the **volatile** option.

See also: **−library**, **−usage**.

−crossref=*list*
Prints cross-reference tables. Default = none.

The *list* consists of keywords separated by commas or colons. The keywords with their meanings are as follows:

calls: table lists each subprogram followed by a list of routines that call it. This listing omits library modules that are not in the call tree of the main program. The list is alphabetized.

common:
table lists each COMMON block followed by a list of the routines that access it. These listed routines are those in which some variables in the COMMON block are accessed, not simply those routines that declare the block. (To find out what routines declare a

COMMON block but do not use it, see the **−usage** flag.)

labels:

table lists each label followed by a list of all references to it. A label reference is denoted by the line number and statement type of the referring statement. The label list is in sequential order. The references are listed in the order they are encountered in the program.

See also: **−calltree**, **−reference**, **−sort**, **−symtab**, **−vcg**.

−declare

If this flag is set, all identifiers whose datatype is not declared in each module will be listed. This flag is useful for helping to find misspelled variable names, etc. The same listing will be given if the module contains an `IMPLICIT NONE` statement. Default = no.

See also: **−sixchar**, **−usage**.

−division

This switch is provided to help users spot potential division by zero problems. If this switch is selected, every division except by a constant will be flagged. (It is assumed that the user is intelligent enough not to divide by a constant which is equal to zero!) Default = no.

See also: **−portability**, **−truncation**.

−errors=*num*

Set the maximum number of error messages in a ''cascade''. During checking of agreement of subprogram arguments, common block declarations, and so forth, sometimes a single case will generate a long string of warnings. Often this simply indicates some other cause than a genuine item-by-item mismatch, such as for example a varible missing from one list. So in such cases **ftnchek** stops printing the warnings after the cascade limit is reached, and the trailer ''`etc...`'' is printed to indicate that there were more errors not printed. If you think that these warnings are likely to be genuine, use this setting to see more of them. Turn-on = default = 3, max = 999. A value of 0 means no limit.

This setting does not set an overall limit on the number of error messages printed, only the number printed in any one cascade. Most types of warnings and error messages are not subject to the cascade effect and so are not affected by this setting. To turn off warnings generally, use the individual warning control options or the **−nocheck** option.

See also: **−check**.

−extern

Causes **ftnchek** to report whether any subprograms invoked by the program are never defined. Ordinarily, if **ftnchek** is being run on a complete program, each subprogram other than the intrinsic functions should be defined somewhere. Turn off this switch if you just want to check a subset of files which form part of a larger complete program. Subprogram arguments will still be checked for correctness. Default = yes.

The **−extern** flag is now superseded by the **−usage=ext−undefined** option. For the sake of convenience, the **−extern** flag is retained, so that **−noextern** is equivalent to **−usage=no−ext−undefined** option. The **−extern** switch may be retired eventually.

See also: **−library**.

−f77=*list*

Use this setting to catch language extensions which violate the Fortran 77 Standard. Such extensions may cause your program not to be portable. Examples include the use of underscores in variable names; variable names longer than six characters; statement lines longer than 72 characters; and nonstandard statements such as the `DO ... ENDDO` structure. **ftnchek** does not report on the use of lowercase letters. By default, all warnings are turned off.

This setting provides detailed control over the warnings about supported extensions to the Fortran 77 Standard. (Further details about the extensions themselves are given below in the section on Extensions.) The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the warnings about nonstandard extensions, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−f77** is equivalent to **−f77=all**, and **−nof77** is equivalent to **−f77=none**. The warning keywords with their meanings are as follows:

accept−type:
`ACCEPT` and `TYPE` I/O statements.

array−bounds:
Expressions defining array bounds that contain array elements or function references.

assignment−stmt:
Assignment statements involving arrays. In Fortran 90, an array can be assigned to another array of compatible shape, or a scalar can be assigned to an array. Neither of these assignments is permitted in Fortran 77.

A related warning occurs when an array is assigned to a scalar. Since this is illegal also in Fortran 90, it is always warned about regardless of the **−f77** setting (unless all checking is turned off with the **−nocheck** flag).

attribute−based−decl:
Type declarations in the new Fortran 90 attribute-based style. This style of declaration is distinguished by the use of a double colon (::) between the list of attributes and the list of declared variables. This option also controls warnings for use of Fortran 90 length or kind specifiers in type declarations. (Although these specifiers can be used in non-attribute-based declarations, they are controlled by this option to avoid proliferation of **−f77** options.)

automatic−array:
Local (not dummy) arrays which have variable size. These would correspond to arrays whose storage would have to be dynamically allocated at run time.

backslash:
Unix backslash escape in strings. This warning will be given only if the **−source=unix−backslash** setting is specified to cause the escape interpretation of backslash..

byte: `BYTE` data type declaration.

case−construct:
The `SELECT CASE` construct.

character:
Extensions to the Fortran 77 standard regarding character data. At present, this only controls warnings about character variables declared with zero or negative length. In Fortran 77, all character variables must be of positive length. In Fortran 90, they can be zero length, and declarations that specify negative lengths are permitted, turning into zero for the declared length. Note: because negative length specifiers may indicate a programming error, the warning about them is given even if this option is turned off, and is suppressed only by the **−nocheck** flag.

common−subprog−name:
Common block and subprogram having the same name.

construct−name:
Use of a construct-name to label a control statement.

continuation:
More than 19 successive continuation lines.

cpp: Unix C preprocessor directives in the source code.

cray−pointer:
"Cray pointer" syntax.

cycle−exit:
The `CYCLE` and `EXIT` statements.

d−comment:
Debugging comments starting with `D` in the source code.

dec−tab:
DEC Fortran style tab-formatted source code. This warning will be given only if the **−source=dec−tab** setting is specified to cause interpretation of tabs in this style.

do−enddo:
`DO` loop extensions: terminal statement label omitted, `END DO`, and `WHILE`.

double−complex:
Double precision complex datatype.

format−dollarsign:
Dollar sign control code in `FORMAT` statements.

format−edit−descr:
Nonstandard edit descriptors in `FORMAT` statements.

function−noparen:
Function definition without parentheses.

implicit−none:
`IMPLICIT NONE` statement.

include:
`INCLUDE` statement.

inline−comment:
Inline comments starting with an exclamation point.

internal−list−io:
List-directed I/O to or from an internal file.

intrinsic:
Nonstandard intrinsic functions.

io−keywords
Nonstandard keywords used in I/O statements. These fall into three groups. The first group includes keywords that are accepted in Fortran 90:

```
ACTION    PAD        READWRITE
ADVANCE   POSITION   SIZE
DELIM     READ       WRITE
EOR
```

The second group comprises the following VMS Fortran keywords:

```
BLOCKSIZE          EXTENDSIZE      READONLY
BUFFERCOUNT        INITIALSIZE     RECORDSIZE
CARRIAGECONTROL    MAXREC          RECORDTYPE
DEFAULTFILE        NAME (in OPEN)  SHARED
DISP               NOSPANBLOCK     TYPE
DISPOSE            ORGANIZATION
```

(The keyword `NAME` is standard only in the `INQUIRE` statement.) The third group consists of the following IBM/MVS keyword:

`NUM`

This flag also controls a warning about use of `ACCESS='APPEND'`, which is accepted by some compilers. The value of `'APPEND'` is not valid for any I/O specifier in standard Fortran 77, and in Fortran 90 `'APPEND'` should be used as a value of the `POSITION` specifier, not `ACCESS`.

long−line:
Statements with meaningful code past 72 columns. This warning is given only if the **−columns** setting has been used to increase the statement field width.

long−name:
Identifiers over 6 characters long.

mixed−common:
Mixed character and noncharacter data in COMMON block.

mixed−expr:
Nonstandard type combinations in expressions, for example `DOUBLE PRECISION` with `COMPLEX`, assigning hollerith to integer, logical operations on integers.

name−dollarsign:
Dollar sign used as a character in identifiers.

name−underscore:
Underscore used as a character in identifiers.

namelist:
`NAMELIST` statement.

param−implicit−type:
Implicit typing of a parameter by the data type of the value assigned. This warning can only occur if implicit parameter typing has been turned on by the **−source=param−implicit−type** option, or if the `PARAMETER` statement is of the nonstandard form without parentheses. If this option is turned on, then any instances where implicit parameter typing occurs will be warned about. If you want to be warned only in those instances where the implicit data type differs from the default type, use **−portability=param−implicit−type** instead. According to the Fortran 77 standard, the data type of a parameter is given by the same rules as for a variable, and if necessary a type conversion is done when the value is assigned.

param−intrinsic:
Intrinsic function or exponentiation by a real used to define the value of a `PARAMETER` definition.

param−noparen:
`PARAMETER` statement without parentheses. The user should be aware that the semantics of this form of the statement differs from that of the standard form: in this form, the parameter takes its data type from the value assigned, rather than having its default data type based on the first letter of the parameter name. (This form of the `PARAMETER` statement was introduced by DEC before the Fortran 77 standard was defined, and should be avoided.)

pointer:
Fortran 90 standard pointer-related syntax, including `POINTER`, `TARGET` and `ALLOCATABLE` type declarations, `ALLOCATE`, `DEALLOCATE`, and `NULLIFY` statements, and pointer assignment using =>.

quad−constant:
Quad precision real constants, e.g. of the form `1.23Q4`.

quotemark:
Strings delimited by quote marks rather than apostrophes.

relops:
Relational (comparison) operators composed of punctuation, namely: < <= == /= > >=.

semicolon:
Semicolon used as statement separator.

statement−order:
Statements out of the sequence mandated by the Standard. The allowed sequence is illustrated in Table 1 in the section on Interpreting the Output.

typeless−constant:
Typeless constants, for example `Z'19AF'`.

type−size:
Type declarations specifying a size, for example `REAL*8`.

variable−format:
Variable repeat specification or field size in FORMAT. These are of the form < *expr* >.

vms−io:
Obsolete. Now has the same meaning as the **io−keywords** keyword.

See also: **−f90**, **−f95**, **−portability**, **−pretty**, **−style**, **−wordsize**.

−f90=*list*

This setting provides detailed control over the warnings about supported extensions to the Fortran 77 Standard that were not adopted as part of the Fortran 90 Standard. Note that **ftnchek** does not support the full Fortran 90 language. However, it does support some common extensions to Fortran 77 that were prevalent before Fortran 90 was defined. Some of these extensions became part of the Fortran 90 Standard, but others did not. The **−f90** setting warns only about the latter. That is, this flag covers things that are neither legal Fortran 77 nor legal Fortran 90. Therefore, the warnings controlled by this flag are basically a subset of the warnings controlled by **−f77**. There are a few cases, described below, where the circumstances in which the warning is given are slightly different for the two flags.

The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the warnings about nonstandard extensions, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−f90** is equivalent to **−f90=all**, and **−nof90** is equivalent to **−f90=none**.

The following keywords have identical meanings for **−f90** as for **−f77**. The reader is referred to the explanations under **−f77**.

accept−type	**double−complex**	**param−noparen**
backslash	**format−dollarsign**	**cray−pointer**
byte	**format−edit−descr**	**quad−constant**
cpp	**function−noparen**	**type−size**
d−comment	**name−dollarsign**	**variable−format**
dec−tab	**param−implicit−type**	**vms−io**

The keywords which differ somewhat from the corresponding **−f77** keywords are as follows.

continuation:
The limit on the number of continuation lines for a statement in fixed source form is the same, namely 19, in Fortran 90 as in Fortran 77. For free source form the limit is 39 continuation lines, and a line containing a continuation mark cannot be otherwise empty or contain only a comment.

intrinsic:

This is the same as for **−f77** except for the intrinsic functions defined in MIL-STD 1753, which are all included in Fortran 90, and so are not warned about. (See **−intrinsic** for a list.)

io−keywords:

This is the same as for **−f77** except that no warnings are given for the I/O keywords that are standard in Fortran 90.

long−line:

Although the Fortran 90 Standard allows lines longer than 72 characters in free source form, this restriction still applies to fixed source form. In free source form the line length limit is 132 characters, and unlike fixed form, **ftnchek** does not allow this limit to be increased.

mixed−expr:

This is the same as for **−f77** except for expressions mixing extended precision real with complex data types, which are permitted in Fortran 90.

statement−order:

This is similar to the corresponding **−f77** warning, but applies the somewhat looser restrictions on statement order of the Fortran 90 Standard. In particular, Fortran 90 allows `DATA` statements and statement-function definitions to be intermixed with specification statements.

typeless−constant:

In Fortran 90, binary, octal, and hexadecimal constants of the form `B'ddd'`, `O'ddd'`, and `Z'ddd'`, respectively, are permitted. Here `'ddd'` represents a string of digits. **ftnchek** recognizes these forms, as well as a variant of the form `X'ddd'` for a hexadecimal constant, and other variants in which the base indicator `B`, `O`, `Z`, or `X` follows the digit string. These variants were not adopted in Fortran 90, so only they are warned about when this flag is turned on.

See also: **−f77**, **−f95**, **−portability**, **−pretty**, **−style**, **−wordsize**.

−f95=*list*

This setting provides detailed control over warnings about standard Fortran 77 features that were deleted from the Fortran 95 Standard. Unlike the **−f77** and **−f90** settings, these warnings apply to syntax which is legal Fortran 77. However, since these features have been deleted from the Standard, it is possible that programs containing them will be unacceptable to some newer compilers.

The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the warnings about nonstandard extensions, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−f95** is equivalent to **−f95=all**, and **−nof95** is equivalent to **−f95=none**. The warning keywords with their meanings are as follows.

real−do:

A `DO` variable of any real numeric type.

pause:

The `PAUSE` statement.

assign:

The `ASSIGN` statement, assigned `GOTO`, or assigned format.

h−edit:

The `H` edit descriptor in a format.

There is one other Fortran 77 syntax feature that was deleted in Fortran 95, namely branching to an `ENDIF` from outside the `IF` block. However, **ftnchek** is unable to analyze program flow, and so it does not provide a warning for this.

See also: **−f77**, **−f90**, **−portability**, **−pretty**, **−style**, **−wordsize**.

−help

Prints a list of all the command-line options with a short description of each along with its default value. This command is identical in function to the **"?"** argument, and is provided as a convenience for those systems in which the question mark has special meaning to the command interpreter. Default = no.

The help listing also prints the version number and patch level of **ftnchek** and a copyright notice.

Note: the ''default'' values printed in square brackets in the help listing are, strictly speaking, not the built-in defaults but the current values after any environment options and any command-line options preceding the **−help** option have been processed.

See also: **−novice**, **−version**, and **help** option of all settings that take a list of keywords.

−identifier−chars=*list*

Define non-alphanumeric characters that may be used in identifiers. By default, **ftnchek** only accepts the dollar sign and underscore as non-alphanumeric characters in identifier names. The characters in the *list* replace whatever set of accepted non-alphanumeric characters was previously in effect. Thus, if dollar sign or underscore are not included in the list, they lose their status as acceptable characters.

This option is provided to enable **ftnchek** to handle source files containing non-standard identifer names that may be needed, for example, to access certain operating system services. See the section on Limitations and Extensions for the treatment of identifiers containing these characters in implicit typing.

Using **−noidentifer−chars** turns off acceptance of non-alphanumeric characters entirely.

See also: **−source**.

−include=*path*

Specifies a directory to be searched for files specified by `INCLUDE` statements. Unlike other command-line options, this setting is cumulative; that is, if it is given more than once on the command line, all the directories so specified are placed on a list that will be searched in the same order as they are given. The order in which **ftnchek** searches for a file to be included is: the current directory; the directory specified by environment variable `FTNCHEK_INCLUDE` if any; the directories specified by any **−include** options; the directory specified by environment variable `INCLUDE`; and finally in a standard system-wide directory (`/usr/include` for UNIX, `SYS$LIBRARY` for VMS, and `\include` for MSDOS).

See also: **−f77**, **−source**.

−intrinsic=*list*

Controls whether **ftnchek** recognizes certain nonstandard intrinsic functions as intrinsic. The *list* consists of keywords separated by commas or colons. Some of the keywords control whether to recognize certain groups of functions, and other keywords control the expected syntax for invoking some nonstandard intrinsics. Include a keyword to turn on recognition of the corresponding set of intrinsics or to allow the corresponding syntax. Include a keyword prefixed by **no−** to turn off that recognition.

There are three special keywords: **all** turns on recognition of all the nonstandard intrinsics (listed below) and accepts either syntax for those that have variations. Use **none** to turn off recognition of all nonstandard intrinsics except those noted below. Use **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−intrinsic** is equivalent to

−intrinsic=all, and **−nointrinsic** is equivalent to **−intrinsic=none**.

The nonstandard intrinsic functions needed to support the nonstandard extended precision data types (double complex and quad precision) are always recognized. The intrinsics for the double complex data type are:

```
CDABS   CDSQRT  DREAL   ZLOG
CDCOS   DCMPLX  IMAG    ZSIN
CDEXP   DCONJG  ZABS    ZSQRT
CDLOG   DIMAG   ZEXP    ZCOS
CDSIN
```

The intrinsics for the quad precision and quad complex types are:

```
CQABS    QARCOS   QEXT     QNINT
CQCOS    QARSIN   QEXTD    QPROD
CQEXP    QATAN    QFLOAT   QREAL
CQLOG    QATAN2   QIMAG    QSIGN
CQSIN    QCMPLX   QINT     QSIN
CQSQRT   QCONJG   QLOG     QSINH
DBLEQ    QCOS     QLOG10   QSQRT
IQINT    QCOSH    QMAX1    QTAN
IQNINT   QDIM     QMIN1    QTANH
QABS     QEXP     QMOD     SNGLQ
```

The keywords controlling recognition of other nonstandard intrinsic functions are as follows:

extra:
recognize the following commonly available nonstandard intrinsics (all except `EXIT` and `LOC` are defined in MIL-STD 1753):

```
BTEST   IBCLR   IEOR    ISHFTC
EXIT    IBITS   IOR     LOC
IAND    IBSET   ISHFT   NOT
```

unix: recognize these common Unix-specific intrinsic functions:

```
ABORT    GMTIME   LTIME    SRAND
AND      IARGC    OR       SYSTEM
GETARG   IRAND    RAND     TIME
GETENV   LSHIFT   RSHIFT   XOR
```

vms: recognize these common VMS-specific intrinsic functions:

```
DATE     IDATE   SECNDS   TIME
ERRSNS   RAN     SIZEOF
```

iargc−no−argument:
specify that `IARGC` may be invoked with no arguments.

iargc−one−argument:
specify that `IARGC` may be invoked with one argument.

rand−no−argument:
specify that `RAND` and `IRAND` may be invoked with no arguments.

rand−one−argument:
specify that `RAND` and `IRAND` may be invoked with one argument.

The **no−argument** and **one−argument** keywords work as follows: turning the option on causes

ftnchek to accept the corresponding syntax for invocation of the function, without excluding the possibility of the alternative syntax. Turning the option off causes the corresponding syntax not to be accepted. If both options are turned on at once (the default), then either syntax is accepted. Turning both options off at once would not be meaningful. These options have no effect if recognition of Unix intrinsics has been turned off.

Note that this setting does not control whether non-standard warnings are issued about these functions. It controls whether the functions are assumed to be intrinsic or not, which determines how their usage is checked. When functions in any of these sets are included, their invocations will be checked according to the rules for the intrinsic functions; otherwise they will be checked as normal (user-written) external functions. The non-standard warnings are controlled by the **−f77=intrinsic** option.

The default value of this setting is equivalent to **−intrinsic=all** followed by **−intrinsic=no−vms** for the Unix version, **−intrinsic=no−unix** for the VMS version, and **−intrinsic=no−unix,no−vms** for other versions.

Note: In versions of **ftnchek** prior to 2.10, the **−intrinsic** flag took a numeric argument instead of a list of options. For the sake of users who may have written scripts invoking **ftnchek** in this way, the numeric form is still accepted. The numeric form of the setting consists of three digits. The ones digit selects the set of intrinsic functions to be supported. The digit 0 selects only Fortran 77 standard intrinsics plus those needed to support the nonstandard extended precision data types. The digit 1 is equivalent to **extra**, 2 is equivalent to **extra,unix**, and 3 is equivalent to **extra,vms**. The tens digit of this setting controls the syntax of the `RAND` intrinsic function, and the hundreds digit controls the syntax of the `IARGC` function. For these digits, specify 0 to require invocation with no argument, 1 to require one argument, and 2 to allow either form.

See also: **−f77**.

−library

This switch is used when a number of subprograms are contained in a file, but not all of them are used by the application. Normally, **ftnchek** warns you if any subprograms are defined but never used. This switch will suppress these warnings. Default = no.

This switch also controls which subprogram calls and COMMON block declarations are checked. If a file is read with the **−library** flag in effect, the subprogram calls and COMMON declarations contained in a routine in that file will be checked only if that routine is in the main program's call tree. On the other hand, if the **−library** switch is turned off, then **ftnchek** checks the calls of every routine by every other routine, regardless of whether those routines could ever actually be invoked at run time, and likewise all COMMON block declarations are compared for agreement.

The difference between this switch and the **−usage=no−ext−unused** option for subprograms is that the latter suppresses only the warning about routines being declared but not used. The **−library** switch goes further and excludes unused routines processed while it is in effect from all cross-checking of arguments and COMMON block declarations as well.

(If there is no main program anywhere in the set of files that **ftnchek** has read, so that there is no call tree, then **ftnchek** will look for any non-library routines that are not called by any other routine, and use these as substitutes for the main program in constructing the call tree and deciding what to check. If no such top-level non-library routines are found, then all inter-module calls and all COMMON declarations will be checked.)

See also: **−arguments**, **−calltree**, **−common**, **−extern**, **−usage**.

−list

Specifies that a listing of the Fortran program is to be printed out with line numbers. If **ftnchek** detects an error, the error message follows the program line with a caret (ˆ) specifying the location of the error. If no source listing was requested, **ftnchek** will still print out any line containing an error, to aid the user in determining where the error occurred. Default = no.

See also: **−output**, \fB−symtab**, fB−quiet**.

−makedcls=*list*

Prepare a neatly-formatted file of declarations of variables, common blocks, and namelist lists, for possible merging into the source code. The declarations are stored in a file of the same name as the source code, but with the extension changed to *.dcl*. If no declarations are written to the file, it is deleted to reduce clutter from empty files.

If input comes from standard input, instead of a named file, then declarations are written to standard output.

Variables are declared in alphabetical order within each declaration class and type, with integer variables first, because of their later possible use in array dimensions.

`PARAMETER` statements are an exception to the alphabetical order rule, because the Fortran 77 Standard requires that the expressions defining parameter values refer only to constants and already-defined parameter names. This forces the original source file order of such statements to be preserved in the declaration files.

Explicit declaration of *all* variables is considered good modern programming practice. By using compiler options to reject undeclared variables, misspelled variable names (or names extending past column 72) can be caught at compile time. Explicit declarations also greatly facilitate changing floating-point precision with filters such as **dtoq**(1L), **dtos**(1L), **fd2s**(1L), **fs2d**(1L), **qtod**(1L), and **stod**(1L). These programs are capable of changing types of explicit floating-point type declarations, intrinsic functions, and constants, but because they do not carry out rigorous lexical and grammatical analysis of the Fortran source code, they cannot provide modified type declarations for undeclared variables. Default setting = 0, turn-on = 1.

Various options for the form of the declarations file are controlled by the *list*, which consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the options, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−makedcls** is equivalent to **−makedcls=declarations** (i.e. produce the declarations file using the default options), and **−nomakedcls** is equivalent to **−makedcls=none**.

For compatibility with previous versions of **ftnchek**, a numeric form of this setting is also accepted: the *list* is replaced by a number which is the sum of the numbers in parentheses beside the keywords in the following list. The warning keywords with their meanings are as follows:

declarations (1):

Write a declaration file. (This is implied by any of the other options, and can be omitted if any other options are given.)

undeclared−only (2):

By default, all variables are included in the declaration file. With this option, include only *undeclared* variables. This setting is useful if you want to check for undeclared variables, since Fortran source files with all variables properly declared will not result in a *.dcl* file. With this option, common blocks and namelist lists will not be included in the declaration file, since by their nature they cannot be undeclared.

compact (4):

The declarations are normally prettyprinted to line up neatly in common columns, as in the declaration files output by the Extended PFORT Verifier, **pfort**(1L). This option value selects instead compact output, without column alignment.

use−continuation−lines (8):

Causes continuation lines to be used where permissible. The default is to begin a new declaration on each line. This option is appropriate to use together with **compact**.

keywords−lowercase (16):
Output Fortran keywords in lowercase, instead of the default uppercase.

vars−and−consts−lowercase (32):
Output variables and constants in lowercase, instead of the default uppercase. Character string constants are not affected by this option.

exclude−sftran3 (64):
Omit declarations of internal integer variables produced by the SFTRAN3 preprocessor, **xsf3**(1L), as part of the translation of structured Fortran statements to ordinary Fortran. These variables have six-character names of the form *NPRddd*, *NXdddd*, *N2dddd*, and *N3dddd*, where *d* is a decimal digit. Because they are invisible in the SFTRAN3 source code, and will change if the SFTRAN3 code is modified, such variables should not be explicitly declared. Instead, they should just assume the default Fortran INTEGER data type based on their initial letter, *N*.

asterisk−comment (128):
Use an asterisk as the comment character; the default is otherwise 'C'.

comment−char−lowercase (256):
Use 'c' instead of 'C' or '*' as the comment character.

suppress−array−dimensions (512):
Suppress dimensioning of arrays in the generated declarations. This option is for use with code lacking type declarations, to allow the declaration files to be inserted without change into the code. Since the code will have dimension statements already, dimensioning the array variables in the type statements of the declaration file is redundant. This option should be used only in conjunction with option 2 = undeclared-only because otherwise any arrays that were dimensioned in a type statement will lose their dimensioning.

free−form (1024):
Produce declarations in free source form. This mode is automatically used if the input source is free form. Use this option to produce declarations in free form even if the input is in fixed form. Free form declarations are indented only 2 columns instead of 6, use the exclamation mark as the comment character, and indicate continuation lines by an ampersand at the end of the line to be continued.

The declaration files contain distinctive comments that mark the start and end of declarations for each program unit, to facilitate using text editor macros for merging the declarations back into the source code.

The **ftnchek** distribution includes a program, **dcl2inc**, which processes declaration files to produce files containing declarations of all COMMON blocks, in a form suitable for use as INCLUDE files. See the **dcl2inc**(1L) man page for the details of its use.

See also: **−mkhtml**.

−mkhtml=*list*
Produce HTML documentation from source. Creates individual HTML files from ftnchek analysis and code comments. All comments immediately preceding and following the function or subroutine definition are captured to the HTML file. No reformatting of source comments is performed other than stripping of FORTRAN comment characters. In addition, the HTML file lists the local variables declared, common block variables used, functions and subroutines called, I/O unit usage, and other information about each subprogram. Usually you will also want to specify **−call=tree** to create the root HTML file `CallTree.html`. (Perhaps this file should be named `index.html`.)

Various options for the form of the HTML files are controlled by the *list*, which consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the options, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−mkhtml** is equivalent to **−mkhtml=documents** (i.e. produce the HTML document files using the default options), and **−nomkhtmls** is equivalent to **−mkhtml=none**.

For the sake of simplicity, the options for **−mkhtml** are the same as those for **−makedcls** except for those that are inapplicable. Likewise, a numeric form of this setting can be used, formed as the sum of the numbers in parentheses in the list below. The warning keywords with their meanings are as follows:

documents (1):
: Create the HTML documents. (This is implied by any of the other options, and can be omitted if any other options are given.)

compact (4):
: The declarations are normally prettyprinted to line up neatly in common columns. This option value selects instead compact output, without column alignment.

use−continuation−lines (8):
: Causes continuation lines to be used instead of beginning a new declaration on each line. This option is appropriate to use together with **compact**.

keywords−lowercase (16):
: Output Fortran keywords in lowercase, instead of the default uppercase.

vars−and−consts−lowercase (32):
: Output variables and constants in lowercase, instead of the default uppercase. Character string constants are not affected by this option.

exclude−sftran3 (64):
: Omit declarations of internal integer variables produced by the SFTRAN3 preprocessor, **xsf3**(1L). (See **−makedcls** for discussion.)

suppress−array−dimensions (512):
: Suppress dimensioning of arrays in the generated declarations. This is normally undesirable, but is available if for some reason you do not want the array dimensions to appear in the HTML.

free−form (1024):
: Produce variable declarations in free source form. This mode is automatically used if the input source is free form. This mainly affects the form of continuation lines if they are used.

See also: **−calltree**, **−makedcls**.

−novice

This flag is intended to provide more helpful output for beginners. It has two effects:

(a) provides an extra message to the effect that a function that is used but not defined anywhere might be an array which the user forgot to declare in a `DIMENSION` statement (since the syntax of an array reference is the same as that of a function reference).

(b) modifies the form of the error messages and warnings. If the flag is turned off by **−nonovice**, these messages are printed in a style more resembling UNIX **lint**.

Default = yes.

−output=*filename*

This setting is provided for convenience on systems which do not allow easy redirection of output from programs. When this setting is given, the output which normally appears on the screen will be sent instead to the named file. Note, however, that operational errors of **ftnchek** itself (e.g. out of space or cannot open file) will still be sent to the screen. The extension for the filename is optional, and if no extension is given, the extension . *lis* will be used.

−pointersize=*num*

Specifies the size of a ‘‘Cray pointer’’ variable to be *num* bytes. Default = turn-on = 4 bytes.

The pointer size is used to inform precision mismatch warnings involving pointer variables, for example when a pointer is assigned a value from an allocation routine, or passed as a subprogram parameter.

See also: **−f77**, **−portability**, **−truncation**, **−wordsize**.

−portability=*list*

ftnchek will give warnings for a variety of non-portable usages. Examples include the use of tabs except in comments or inside strings, the use of Hollerith constants, and the equivalencing of variables of different data types. This option does not produce warnings for supported extensions to the Fortran 77 Standard, which may also cause portability problems. To catch those, use the **−f77** setting. By default, all warnings are turned off.

This setting provides detailed control over the warnings about possible portability problems. The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the warnings about nonportable usages, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−portability** is equivalent to **−portability=all**, and **−noportability** is equivalent to **−portability=none**. The warning keywords with their meanings are as follows:

backslash:

Backslash character in strings. Since some compilers treat the backslash as an escape character, its presence can cause problems even though it is used in a standard-conforming way.

common−alignment:

COMMON block variables not in descending order of storage size. Some compilers require this ordering because of storage alignment requirements.

hollerith:

Hollerith constants (other than within FORMAT specifications). The Hollerith data type is a feature of Fortran IV that has been deleted in the Fortran 77 standard. It is superseded by the character data type. Storing Hollerith data in variables of a numeric or logical data type is nonportable due to differing word sizes.

long−string:

String constants, variables, or expressions over 255 chars long.

mixed−equivalence:

Variables of different data types equivalenced.

mixed−size:

Variables declared with default precision used with variables given explicit precision, in expressions, assignments, or as arguments. For example, if a variable declared as `REAL*8` is treated as equivalent to `DOUBLE PRECISION`.

real−do:

Non-integer DO loop index and bounds. These can cause a program’s results to depend on the hardware characteristics of the particular computer used.

param−implicit−type:

Implicit typing of a parameter by the data type of the value assigned, if it differs from the default type. This warning can only occur if implicit parameter typing has been turned on by the **−source=param−implicit−type** option, or if the `PARAMETER` statement is of the nonstandard form without parentheses. If this option is turned on, then any instances where implicit parameter typing occurs and where the implicit type is different from the default type based on the first letter of the parameter name, will be warned about. Implicit parameter typing can change the semantics of statements where the parameter is used, causing portability problems.

tab: Tabs in source code. Tabs are interpreted differently by different compilers. This warning will be given only once, at the end of the file.

See also: **−f77**, **−f90**, **−f95**, **−pretty**, **−style**, **−wordsize**.

−pretty=*list*

Controls certain messages related to the appearance of the source code. These warn about things that might make a program less readable or be deceptive to the reader. By default, all warnings are turned on.

This setting provides detailed control over the warnings about appearance. The *list* consists of keywords separated by commas or colons. Since all warnings are on by default, include a keyword prefixed by **no−** to turn off a particular warning. There are three special keywords: **all** to turn on all the warnings about misleading appearances, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−pretty** is equivalent to **−pretty=all**, and **−nopretty** is equivalent to **−pretty=none**. The warning keywords with their meanings are as follows:

alternate−return:
A RETURN statement has a constant specifying an alternate return point that is not between 0 and the number of dummy arguments that are labels. This is legal, and has the same effect as a RETURN with no alternate return expression, but suggests that the programmer intended to use an alternate return label that is not provided.

embedded−space:
Space embedded in variable names or in multi-character operators such as `**`.

continuation:
Continuation mark following a comment line.

long−line:
Lines (except comments) over 72 columns in width (beyond 72 is normally ignored by compiler).

missing−space:
Lack of space between variable and a preceding keyword.

multiple−common:
COMMON block declared in multiple statements. No warning is given if the statements are consecutive except for comment lines.

multiple−namelist:
NAMELIST declared in multiple statements. No warning is given if the statements are consecutive except for comment lines.

parentheses:
Parentheses around a variable by itself. As a subprogram argument, this makes the argument an expression, not modifiable by the subprogram.

Note that in free source form, extra space and missing space are forbidden by the Fortran 90 Standard, and are not mere style violations. In this case the warnings are replaced by syntax error messages, and can be turned off only by using **−nocheck**.

See also: **−f77**, **−portability**, **−style**.

−project=*list*

ftnchek will create a project file from each source file that is input while this option is turned on. The project file will be given the same name as the input file, but with the extension *.f* or *.for* replaced by *.prj*. (If input is from standard input, the project file is named *ftnchek.prj*.) Default = none.

The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the options, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−project** is equivalent to **−project=all**, and **−noproject** is equivalent to **−project=none**. The keywords with their meanings are as follows:

create:

Produce a project file. The default is not to produce a project file. If this option is not turned on, the other options have no effect.

trim−calls:

Trim the amount of information stored in the project file about subprogram declarations and calls. This is the default. Turn this option off only in rare situations. (See discussion below.) The amount of trimming varies depending on the **−library** flag. More information is trimmed if that flag is turned on.

trim−common:

Trim the number of common block declarations stored in the project file. This is the default. Turn this option off only in rare situations. (See discussion below.) This option has no effect if the **−library** flag is turned off: when not in library mode, no trimming of common block declarations is done regardless of this option.

A project file contains a summary of information from the source file, for use in checking agreement among `FUNCTION`, `SUBROUTINE`, and `COMMON` usages in other files. It allows incremental checking, which saves time whenever you have a large set of files containing shared subroutines, most of which seldom change. You can run **ftnchek** once on each file with the **−project** flag set, creating the project files. Usually you would also set the **−library** and **−noextern** flags at this time, to suppress messages relating to consistency with other files. Only error messages pertaining to each file by itself will be printed at this time. Thereafter, run **ftnchek** without these flags on all the project files together, to check consistency among the different files. All messages internal to the individual files will now be omitted. Only when a file is altered will a new project file need to be made for it.

Naturally, when the **−project** option is turned on, **ftnchek** will not read project files as input.

Ordinarily, the trim options should be left on when you intend to create project files for future input to **ftnchek**. Since trimming is on by default, this means that simply giving the command **−project** with no option list is the recommended mode. The trim options are provided only as a convenience for those who want to make use of project files for purposes other than checking the program with **ftnchek**. To use project files for their intended purpose, the trim options should not be turned off.

Project files contain only information needed for checking agreement between files. This means that a project file is of no use if all modules of the complete program are contained in a single file.

A more detailed discussion is given in the section on Using Project Files.

−pure

Assume functions are ''pure'', i.e., they will not have side effects by modifying their arguments or variables in a COMMON block. When this flag is in effect, **ftnchek** will base its determination of set and used status of the actual arguments on the assumption that arguments passed to a function are not altered. It will also issue a warning if a function is found to modify any of its arguments or any COMMON variables. Default = yes.

When this flag is turned off, actual arguments passed to functions will be handled the same way as actual arguments passed to subroutines. This means that **ftnchek** will assume that arguments may be modified by the functions. No warnings will be given if a function is found to have side effects. Because stricter checking is possible if functions are assumed to be pure, you should turn this flag off only if your program actually uses functions with side effects.

−quiet

This option reduces the amount of output relating to normal operation, so that error messages are more apparent. This option is provided for the convenience of users who are checking large suites of files. The eliminated output includes the names of project files, and the message reporting that no syntax errors were found. It also eliminates some blank lines that are ordinarily included for clarity. (Some of this output is turned back on by the **−list** and **−symtab** options.) Default = no.

Note: the way to remember the difference between the **−quiet** and **−brief** is that **−quiet** doesn't suppress any warning-related information, whereas **−brief** does.

See also: **−brief**.

−reference

Specifies that a who-calls-who table be printed. This table lists each subprogram followed by a list of the routines it calls. This switch is equivalent to **−calltree=reference**. Default = no.

The reference list omits routines called by unused library modules. Thus it contains the same information as for the call-tree format, namely the hierarchy of subprogram calls, but printed in a different way. This prints out a breadth-first traversal of the call tree whereas **−calltree=tree** prints out a depth-first traversal.

See also: **−calltree**, **−crossref**, **−library**, **−sort**, **−symtab**, **−vcg**.

−resources

Prints the amount of resources used by **ftnchek** in processing the program. This listing may be useful in analyzing the size and complexity of a program. It can also help in choosing larger sizes for **ftnchek**'s internal tables if they are too small to analyze a particular program. Default = no.

In this listing, the term "chunk size" is the size of the blocks of memory allocated to store the item in question, in units of the size of one item, not necessarily in bytes. When the initially allocated space is filled up, more memory is allocated in chunks of this size. The following is an explanation of the items printed:

Source lines processed:

Total number of lines of code, with separate totals for statement lines and comment lines. Comment lines include lines with 'C' or '*' in column 1 as well as blank lines and lines containing only an inline comment. Statement lines are all other lines, including lines that have an inline comment following some code. Continuation lines are counted as separate lines. Lines in include files are counted each time the file is included.

Total executable statements:

Number of statements in the program, other than specification, data, statement-function, FORMAT, ENTRY, and END statements.

Total number of modules:

A module is any external subprogram, including the main program, subroutines, functions, and block data units. This count is of modules defined within the source, not modules referenced. Statement functions are not included. A subprogram with multiple entry points is only counted once.

Total statement labels defined

Number of labels attached to statements (often called statement numbers). The total label count for the entire program is given, as well as the maximum number in any single subprogram.

Max identifier name chars:
Number of characters used for storing identifier names. An identifier is a variable, subprogram, or common block name. Local names are those of local variables in a subprogram, whereas global names refer to subprogram and common block names, as well as dummy argument names and common variable names. Actual argument text (up to 15 characters for each argument) is also included here. The space used for local names is not recovered at the end of each module, so this number, like global space, grows until the whole program is analyzed. Unfortunately, this figure may include some text stored more than once, although a heuristic is used that will avoid duplicates in many cases.

Max token text chars:
A token is the smallest syntactic unit of the FORTRAN language above the level of individual characters. For instance a token can be a variable name, a numerical constant, a quoted text string, or a punctuation character. Token text is stored while a module is being processed. For technical reasons, single-character tokens are not included in this total. Items that are not represented in the symbol table may be duplicated. The space for token text is recovered at the end of each module, so this figure represents the maximum for any one module.

Max local symbols:
This is the largest number of entries in the local symbol table for any module. Local symbol table entries include all variables and parameters, common block names, statement functions, external subprograms and intrinsic functions referenced by the module. Literal constants are not stored in the local symbol table.

Max global symbols:
This is the number of entries in the global symbol table at the end of processing. Global symbol table entries include external subprogram and common block names. Intrinsic functions and statement functions are not included.

Max number of tokenlists:
A token list is a sequence of tokens representing the actual or dummy argument list of a subprogram, or the list of variables in a common block or namelist. Therefore this number represents the largest sum of COMMON, CALL, NAMELIST and ENTRY statements and function invocations for any one module. The space is recovered at the end of each module.

Max token list/tree space:
This is the largest number of tokens in all the token lists and token trees of any one module. A token tree is formed when analyzing an expression: each operand is a leaf of the tree, and the operators are the nodes. Therefore this number is a measure of the maximum complexity of an individual module. For instance a module with many long arithmetic expressions will have a high number. Note that unlike token text described above, the number of tokens is independent of the length of the variable names or literal constants in the expressions.

Number of subprogram invocations:
This is the sum over all modules of the number of CALL statements and function invocations (except intrinsic functions and statement functions).

Number of common block decls:
This is the sum over all modules of the number of common block declarations. That is, each declaration of a block in a different module is counted separately. (The standard allows multiple declarations of a block within the same module; these are counted as only one declaration since they are equivalent to a single long declaration.)

Number of array dim & param ptrs:
This is the sum over all modules of the number of array dimension and parameter definition text strings saved for use by the **−makedcls** option. The length of the text strings is not counted. Each dimension of a multidimensional array is counted separately.

These numbers are obviously not the same when project files are used in place of the original source code. Even the numbers for global entities may be different, since some redundant information is eliminated in project files.

−sixchar

One of the goals of the **ftnchek** program is to help users to write portable Fortran programs. One potential source of nonportability is the use of variable names that are longer than six characters. Some compilers just ignore the extra characters. This behavior could potentially lead to two different variables being considered as the same. For instance, variables named `AVERAGECOST` and `AVERAGEPRICE` are the same in the first six characters. If you wish to catch such possible conflicts, use this flag. Default = no.

Use the **−f77=long−names** if you want to list *all* variables longer than six characters, not just those pairs that are the same in the first six.

See also: **−f77**, **−portability**.

−sort

Specifies that a sorted list of all modules used in the program be printed. This list is in ‘‘prerequisite’’ order, i.e. each module is printed only after all the modules from which it is called have been printed. This is also called a ‘‘topological sort’’ of the call tree. Each module is listed only once. Routines that are not in the call tree of the main program are omitted. If there are any cycles in the call graph (illegal in standard Fortran) they will be detected and diagnosed. Default = no.

See also: **−calltree**, **−crossref**, **−reference**, **−symtab**, **−vcg**.

−source=*list*

This setting controls certain options about the form of the Fortran source code. The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the options, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−source** is equivalent to **−source=all**, and **−nosource** is equivalent to **−source=none**.

For compatibility with previous versions of **ftnchek**, a numeric form of this setting is also accepted: the *list* is replaced by a number which is the sum of the numbers in parentheses beside the keywords in the following list. (The **fixed** and **free** options do not have numeric values.) The warning keywords with their meanings are as follows:

fixed:
Interpret the source as fixed form (with supported extensions such as exclamation mark for comments). Statements must be in columns 7 to 72 (unless the **−cols** setting has been used to change this), and blanks are not significant outside character context (but warned about under the **−pretty** option). This is the default mode unless the source file extension is *.f90* or *.F90*. this option cannot be given together with **−source=free**.

free: Interpret the source as free form. Statements may be anywhere in columns 1 to 132, comments can only begin with an exclamation mark, and blanks are required in some places such as between identifiers and keywords. This is the default mode if the source file extension is *.f90* or *.F90*. This option cannot be given together with **−source=fixed** or **−source=dec−tab**

dec−tab (1):
Accept DEC-style tab-formatted source. A line beginning with an initial tab will be treated as a new statement line unless the character after the tab is a nonzero digit, in which case it is treated as a continuation line. The next column after the tab or continuation mark is taken as column 7. A warning will be given in the case where the line is a continuation, if **−f77=dec−tab** is in effect.

vms−include (2):

Accept VMS-style `INCLUDE` statements. These follow the normal syntax, but with the following additional features: (1) the file extension, if not given, defaults to the same as a normal source file extension; and (2) the option `/LIST` or `/NOLIST` can be appended to the include-file name, to control listing of its contents.

unix−backslash (4):

Handle UNIX-style backslash escapes in character strings. The escape sequence following the backslash will be evaluated according to the ANSI standard for strings in C: up to three digits signify an octal value, an `x` signifies the start of a hexadecimal constant, any of the letters `a b f n r t` signify special control codes, and any other character (including newline) signifies the character itself. When this source code option is in effect, a warning will be given if the **−f77=backslash** setting is specified.

The default behavior is to treat the backslash like any other normal character, but a warning about portability will be generated if the **−portability** flag is set. Because of the fact that some compilers treat the backslash in a nonstandard way, it is possible for standard-conforming programs to be non-portable if they use the backslash character in strings.

Since **ftnchek** does not do much with the interpreted string, it is seldom necessary to use this option. It is needed in order to avoid spurious warnings only if (a) the program being checked uses backslash to embed an apostrophe or quote mark in a string instead of using the standard mechanism of doubling the delimiter; (b) the backslash is used to escape the end-of-line in order to continue a string across multiple source lines; or (c) a `PARAMETER` definition uses an intrinsic string function such as `LEN` with such a string as argument, and that value is later used to define array dimensions, etc.

param−implicit−type (8):

Implicit typing of a parameter by the data type of the value assigned. Some non-standard compilers may allow the data type of the value to override the Fortran 77 default type of a parameter that is based on the first letter of the parameter name. This option only applies to `PARAMETER` statements of the standard form which has parentheses. A parameter that has been explicitly declared in a type statement prior to the `PARAMETER` statement is not affected by this option. A warning will be given under the **−f77=param−implicit−type** or **−portability=param−implicit−type** option.

Note that this implicit typing is treated as equivalent to an explicit type declaration for the parameter. Therefore, if you use **−makedcls=undeclared−only** to generate declarations only of undeclared variables, these parameters will *not* be included.

dec−param−standard−type (16):

Follow the Fortran 77 rule for data typing of DEC Fortran style parameters. These are declared using a nonstandard form of the `PARAMETER` statement which lacks parentheses. According to DEC Fortran, parameters defined by this form of the statement have their data type given by the data type of the value assigned. Use this option to tell **ftnchek** not to follow this rule but instead to use the same rule as for standard `PARAMETER` statements. This option does not apply to `PARAMETER` statements of the standard form.

By default, all these source code options are turned off, except for the **vms−include** option, which is on by default in the VMS version..

See also: **−f77**, **−include**, **−portability**.

−style=*list*

Provides extra-picky warnings about obsolescent or old-fashioned programming constructions. This option is helpful for efforts to follow a modern programming style. (Most of the things complained about under this option are forbidden in the **F** `subset language.) By default, all warnings are turned off.`

The *list* consists of keywords separated by commas or colons. There are three special keywords: **all** to turn on all the options, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−style** is equivalent to **−style=all**, and **−nostyle** is equivalent to **−style=none**. The warning keywords with their meanings are as follows:

block−if:
Complain about arithmetic IF statement. Accept block IF or logical IF (which controls a single statement).

construct−name:
Complain about unnamed block constructs: IF, DO, and SELECT CASE. Note that if a construct name is present on the opening statement of a construct, then it is required to be present on all other component statements (ELSE, END IF, etc.) of the construct. In that case a missing construct name on those statements generates a syntax error regardless of this option. The purpose of this option is to warn if the construct completely lacks the optional name.

distinct−do:
Complain if two DO loops share a common terminator statement.

do−construct:
Complain if terminator of a DO loop is anything other than an END DO or CONTINUE statement. This is the requirement in order for the loop to meet the Fortran 90 definition of a do-construct.

do−enddo:
Complain if terminator of a DO loop is anything other than an END DO statement. (This option overrides the **do−construct** option, being even stricter.)

end−name:
Complain about the absence of the subprogram name on structured END statements.

format−stmt:
Complain about the presence of FORMAT statements. Only the FORMAT statements themselves are flagged, not the references to them in I/O lists.

goto: Complain about the presence of unconditional, computed or assigned GOTO statements. Also complain about alternate returns (but not about labels as subprogram arguments).

labeled−stmt:
Complain about the presence of labels (numbers) on statements other than FORMAT statements. (Since FORMAT statements are arguably convenient and not readily abused, complaints about them are controlled by the separate **format−stmt** keyword.)

program−stmt:
Complain about the absence of a PROGRAM statement at the head of the main program.

structured−end:
Complain about the use of a plain END statement to end a subprogram, rather than a structured END statement (END PROGRAM, END SUBROUTINE, END FUNCTION, or END BLOCK DATA).

See also: **−f77**, **−f90**, **−f95**, **−pretty**, **−portability**.

−symtab

A symbol table will be printed out for each module, listing all identifiers mentioned in the module. This table gives the name of each variable, its datatype, and the number of dimensions for arrays. An asterisk (*) indicates that the variable has been implicitly typed, rather than being named in an explicit type declaration statement. The table also lists all subprograms invoked by the module, all COMMON blocks declared, etc. Default = no.

Also, for each module, a label table will be printed. The table lists each label defined in the module; the line on which said statement label is defined; and the statement type (executable, format, or specification). The labels are listed in sequential order.

Also printed is a table describing the I/O units used by the module, together with information about how they are used: what operations are performed, whether the access is sequential or direct, and whether the I/O is formatted or unformatted.

See also: **−calltree**, **−crossref**, **−list**, **−reference**, **−sort**, **−vcg**.

−truncation=*list*

Warn about possible truncation (or roundoff) errors. Most of these are related to integer arithmetic. By default, all warnings are turned on.

This setting provides detailed control over the warnings about possible truncation errors. The *list* consists of keywords separated by commas or colons. Since all warnings are on by default, include a keyword prefixed by **no−** to turn off a particular warning. There are three special keywords: **all** to turn on all the warnings about truncation, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−truncation** is equivalent to **−truncation=all**, and **−notruncation** is equivalent to **−truncation=none**. The warning keywords with their meanings are as follows:

int−div−exponent:

use of the result of integer division as an exponent. This suggests that a real quotient is intended. An example would be writing `X**(1/3)` to evaluate the cube root of `X`. The correct expression is `X**(1./3.)`.

int−div−real:

Conversion of an expression involving an integer division to real. This suggests that a real quotient is intended.

int−div−zero:

division in an integer constant expression that yields a result of zero.

int−neg−power:

exponentiation of an integer by a negative integer (which yields zero unless the base integer is 1 in magnitude). This suggests that a real base is intended.

promotion:

automatic conversion of a lower precision quantity to one of higher precision. The loss of accuracy for real variables in this process is comparable to the corresponding demotion. No warning is given for promotion of integer quantities to real since this is ordinarily exact.

real−do−index:

use of a non-integer DO index in a loop with integer bounds. An integer DO index with real bounds is always warned about regardless of this setting.

real−subscript:

use of a non-integer array subscript.

significant−figures:

overspecifying a single precision constant. This may indicate that a double precision constant was intended.

size−demotion:

automatic conversion of a higher precision quantity to one of lower precision of the same type. This warning only occurs when an explicit size is used in declaring the type of one or both operands in an assignment. For example, a warning wil be issued where a `REAL*8` variable is assigned to a `REAL` variable, if the default wordsize of 4 is in effect. A warning is also issued if a long integer is assigned to a shorter one, for example, if an `INTEGER` expression is assigned to an `INTEGER*2` variable. There is one exception to this last case, namely if the right hand side of the assignment is a small literal constant (less than 128).

type−demotion: automatic conversion of a higher precision quantity to one of lower precision of different type. This warning includes conversion of real quantities to integer, double precision to single precision real, and assignment of a longer character string to a shorter one.

The warnings about promotion and demotion also apply to complex constants, considering the precision to be that of the real or imaginary part. Warnings about promotions and demotions are given only when the conversion is done automatically, e.g. in expressions of mixed precision or in an assignment statement. If intrinsic functions such as `INT` are used to perform the conversion, no warning is given.

See also: **−portability**, **−wordsize**.

−usage=*list*

Warn about unused or possible uninitialized variables, unused common blocks, undefined or unused statement labels, and unused or undefined subprograms. By default, all warnings are turned on.

This setting provides detailed control over the warnings about possible usage errors. The *list* consists of keywords separated by commas or colons. Since all warnings are on by default, include a keyword prefixed by **no−** to turn off a particular warning. There are three special keywords: **all** to turn on all the warnings about usage, **none** to turn them all off, and **help** to print the list of all the keywords with a brief explanation of each. If *list* is omitted, **−usage** is equivalent to **−usage=all**, and **−nousage** is equivalent to **−usage=none**. These warnings cover four main categories of objects: subprogram dummy arguments, common blocks and variables, subprograms and functions, and local variables. Warnings include undefined items, multiply defined items, unused items, etc. The warning keywords with their meanings are as follows:

arg−alias:
a scalar dummy argument is actually the same as another and is (or may be) modified. The Fortran 77 standard (section 15.9.3.6) prohibits modifying an argument that is aliased to another.

arg−array−alias:
a dummy argument which is an array or array element is in the same array as another and is modified. This flag is similar to **arg−alias** but provides separate control over array arguments. It is harder to tell if aliasing is occurring in the case of arrays, so if **ftnchek** gives too many false warnings, this flag allows the array-related ones to be turned off without suppressing the warnings for scalars.

arg−common−alias:
a scalar dummy argument is the same as a common variable in the subprogram, and either is modified. This is also prohibited by the Fortran 77 standard. If common checking is not exact (see the **−common** setting), it is harder to tell if aliasing is occurring, so the warning is given if the variable is anywhere in a common block that is declared by the subprogram.

arg−common−array−alias:
a dummy argument which is an array or array element is in the same array as a common variable, and either is modified. If common checking is not exact, the variable can be anywhere in a common block that is declared by the subprogram.

arg−const−modified:
a subprogram modifies an argument which is a constant or an expression. Such an action could cause anomalous behavior of the program.

arg−unused:
a dummy argument is declared but never used. This is similar to the **var−unused** keyword described below, but applies only to arguments.

com−block−unused:
a common block is declared but none of the variables in it are used by any subprogram. This warning is suppressed if the common strictness setting is 0.

com−block−volatile:
a common block may lose the definition of its contents if common blocks are volatile. This option only has an effect if the **−common=volatile** flag is in effect. See the discussion of the **−common** setting above.

com−var−set−unused:
a common variable is assigned a value, but its value is not used by any subprogram. This warning is suppressed if the common strictness setting is 0.

com−var−uninitialized:
a common variable's value is used in some subprogram, but is not set anywhere. Unfortunately, **ftnchek** does not do a thorough enough analysis of the calling sequence to know which routines are called before others. So warnings about this type of error will only be given for cases in which a variable is used in some routine but not set in any other routine. Checking of individual COMMON variables is done only if the **−common** setting is 3 (variable by variable agreement). This warning is suppressed if the common strictness setting is 0.

com−var−unused:
a common variable is declared but not used by any subprogram. This warning is suppressed if the common strictness setting is 0.

do−index−modified:
a variable that is the index of a DO loop is modified by some statement within the range of the loop. The Standard permits an active DO variable to be modified only by the incrementation mechanism of the DO statement.

ext−multiply−defined:
an external (a subroutine or function) is defined more than once. Definition of an external means providing the body of its source code.

ext−declared−only:
a name is declared in an `EXTERNAL` statement in some module, but is not defined or used anywhere.

ext−undefined:
an external is used (invoked) but not defined anywhere. This option is equivalent to the **−external** flag. If the subprogram is invoked more than once, those invocations will still be checked for consistency.

ext−unused:
an external is defined (its subprogram body is present) but it is not used. A subprogram is considered unused even if it is invoked by some other subprogram, if it cannot be called from any thread of execution starting with the main program. The agreement of the subprogram's arguments with its invocations is still checked even if this warning is turned off. If there is no main program, then this warning is issued only if the subprogram is not invoked anywhere. This warning is suppressed in library mode, but library mode has the additional effect of suppressing argument checking for unused routines.

label−undefined:
a statement refers to a label that has not been defined.

label−unused:
a statement label is defined, but never referred to.

var−set−unused:
a local variable is assigned a value, but that value is not used.

var−uninitialized:

a local variable's value may be used before it is assigned. Sometimes **ftnchek** makes a mistake in the warnings about local variable usage. Usually it errs on the side of giving a warning where no problem exists, but in rare cases it may fail to warn where the problem does exist. See the section on Bugs for examples. If variables are equivalenced, the rule used by **ftnchek** is that a reference to any variable implies the same reference to all variables it is equivalenced to. For arrays, the rule is that a reference to any array element is treated as a reference to all elements of the array.

var−unused:

a local variable is declared (for instance, in a type declaration) but is not used in the module. Does not apply to dummy arguments: warnings about them are controlled by the keyword **arg−unused** described above.

Note: In versions of **ftnchek** prior to 2.10, the **−usage** flag took a numeric argument instead of a list of options. For the sake of users who may have written scripts invoking **ftnchek** in this way, the numeric form is still accepted. The numeric setting is composed of three digits. The first digit (hundreds place) controls warnings about subprograms (functions and subroutines), the second digit (tens place) warnings about common blocks and common variables,, and the third digit (ones place) warnings about local variables. Each digit controls warnings according to the convention that a 1 means warn about undefined items and variables that are used before set, a 2 means warn about items that are unused, and a 3 means warn about both types. These numbers are now converted to the appropriate values for the above-listed keywords, except for **com−block−volatile**, which is not affected by the numeric argument.

See also: **−common**, **−declare**, **−extern**, **−library**.

−vcg

Produce the call graph in the form of a VCG graph description. This description is written to a separate file, with the same stem as the file containing the main program, and suffix *.vcg*. This file is able to be given directly to **xvcg**(1L) to visualize the call graph. (If input is from the standard input, then the graph description is sent to standard output.) This switch is equivalent to **−calltree=vcg**. Default = no.

The VCG description as created is more complex than it need be. VCG allows graphs and nested subgraphs: each subroutine is created as a subgraph nested inside its calling routines. This allows you to interactively display subgraphs or summarise them.

The **−vcg** option for **ftnchek** was written by Dr. Philip Rubini of Cranfield University, UK.

xvcg is a graph visualisation tool which runs under the X windows system. It is freely available from ftp.cs.uni−sb.de. It was written by G. Sander of the University of Saarland, Germany.

See also: **−calltree**, **−crossref**, **−reference**, **−sort**.

−version

This option causes **ftnchek** to print a line giving the version number, release date, and patch level of the program. If no files are given, it then exits. If files are given, the effect of this option is to include the patch level (normally omitted) in the version information printed at the start of processing. Default = no.

See also: **−help**.

−volatile

Assume that COMMON blocks are volatile. Default = no.

This flag is superseded by **−common=volatile**, and should no longer be used. It may be eliminated in a future release of **ftnchek**.

See also: **−common**, **−usage**.

−wordsize=*num*

Specifies the default word size to be *num* bytes. This is the size of logical and single-precision numeric variables that are not given explicit precisions. (Explicit precisions for non-character variables are an extension to the Fortran 77 Standard, and are given by type declarations such as `REAL*8 X`.) Double-precision and complex variables will be twice this value, and double complex variables four times. Quad-precision constants and intrinsic function results will be four times this value. Note that variables declared as `REAL*16` will be regarded as quad precision only if the word size is 4 bytes. Default = turn-on = 4 bytes.

The word size value does not matter for checking standard-conforming programs that do not declare explicit precisions for non-character variables or store Hollerith data in variables. This setting also does not affect the default size of character variables, which is always 1 byte. Hollerith constants also are assumed to occupy 1 byte per character.

The word size is used to determine whether truncation occurs in assignment statements, and to catch precision mismatches in subprogram argument lists and common block lists. The exact warnings that are issued will depend on the status of other flags. Under both the **−portability=mixed−size** and the **−nowordsize** flag, any mixing of explicit with default precision objects (character expressions not included) is warned about. This applies to arithmetic expressions containing both types of objects, and to subprogram arguments and COMMON variables. Under control of the **−truncation=demotion** and **promotion** options, a warning is given for assignment of an expression to a shorter variable of the same type, or for promotion of a lower precision value to higher precision in an arithmetic expression or an assignment statement.

Giving a word size of 0, or equivalently, using **−nowordsize** means that no default value will be assumed. This is equivalent to specifying **−portability=mixed−size**. Use it to find cases of mixing default and explicit precision, for example to flag places where `REAL*8` is treated as equivalent to `DOUBLE PRECISION`.

See also: **−pointersize**, **−portability**, **−truncation**.

−wrap=*col*

Controls the wrapping of error messages. Long error messages that would run past the specified column will be broken up into separate lines between the words of the message for better readability. If turned off with **−nowrap**, each separate error message will be printed on one line, leaving it up to the display to wrap the message or truncate it. Default = turn-on = 79.

CHANGING THE DEFAULTS

ftnchek includes two mechanisms for changing the default values of all options: by defining environment variables or by creating a preferences file. When **ftnchek** starts up, it looks in its environment for any variables whose names are composed by prefixing the string `FTNCHEK_` onto the uppercased version of the option name. If such a variable is found, its value is used to specify the default for the corresponding switch or setting. In the case of settings (for example, the **−common** strictness setting) the value of the environment variable is read as the default setting value. In the case of switches, the default switch will be taken as true or yes unless the environment variable has the value `0` or `NO`.

Note that the environment variable name must be constructed with the full-length option name, which must be in uppercase. For example, to make **ftnchek** print a source listing by default, set the environment variable `FTNCHEK_LIST` to `1` or `YES` or anything other than `0` or `NO`. The names `FTNCHEK_LIS` (not the full option name) or `ftnchek_list` (lower case) would not be recognized.

Here are some examples of how to set environment variables on various systems. For simplicity, all the examples set the default **−list** switch to `YES`.

1. UNIX, Bourne shell: `$ FTNCHEK_LIST=YES`

```
$ export FTNCHEK_LIST
```

2. UNIX, C shell: `% setenv FTNCHEK_LIST YES`

3. VAX/VMS: `$ DEFINE FTNCHEK_LIST YES`

4. MSDOS: `$ SET FTNCHEK_LIST=YES`

After processing any environment variables, ftnchek looks for a preferences file containing options and settings. It will search in the following order, using only the first file found: (1) *.ftnchekrc* in the current directory, (2) *ftnchek.ini* in the current directory, (3) *.ftnchekrc* in the user's home directory, (4) *ftnchek.ini* in the home directory. If such a file is found, the options defined in it are used as defaults in place of the built-in defaults and overriding any defaults set in the environment..

Each option or setting in the preferences file must be on a separate line. They are given in the same form as on the command line, except without the initial dash. The preferences file can contain blank lines and comments. Comments are introduced at any point in a line by a space character (blank or tab) or the '#' character, and are terminated by the end of the line.

Command-line options override the defaults set in the environment or in the preferences file, in the same way as they override the built-in defaults.

USING PROJECT FILES

This section contains detailed information on how to use project files most effectively, and how to avoid some pitfalls.

One can divide the checks **ftnchek** does into two categories, local and global. Local checking is restricted to within a single routine, and catches things like uninitialized variables, unintended loss of precision in arithmetic expressions, etc. This sort of checking can be done on each subprogram independently. Furthermore, local checking of a subprogram does not need to be repeated when some other subprogram is changed. Global checking catches things like calling a subroutine with the wrong argument types, or disagreeing in common block declarations. It requires looking at the whole set of subprograms interacting with each other.

The purpose of project files is to allow the local checking and global checking steps to be separated. Assuming that each subprogram is in its own source file, you can run **ftnchek** once on each one to do local checking while suppressing global checking. Then **ftnchek** can be run once on all the project files together to do the global checking. The sample makefile below shows how to automate this task. The "`.f.prj`" target updates a project file for a particular file any time the source file changes. The information needed for global checking is saved in the project file. The "`check`" target does the combined global checking. Typically "`make check`" would repeat the "`ftnchek -project`" step only on changed source files, then do the global check. This is obviously a big advantage for large programs, when many subprograms seldom if ever change.

It is best when using project files to place each subprogram in a separate source file. If each source file may contain more than one subprogram, it complicates the definition of "local" and "global" checking because there is some inter-module checking that is contained within a file. **ftnchek** tries to do the right thing in this case, but there are some complications (described below) due to the trade-off between avoiding re-doing cross-checks and preserving information about the program's structure.

Ordinarily, to do the least amount of re-checking, project files should be created with the **–library** flag in effect and trimming turned on. In this mode, the information saved in the project file consists of all subprogram declarations, all subprogram invocations not resolved by declarations in the same file, and one instance of each COMMON block declaration. This is the minimum amount of information needed to check agreement between files.

If the source file contains more than one routine, there are some possible problems that can arise from creating the project file in library mode, because the calling hierarchy among routines defined within the file is lost. Also, if the routines in the file make use of COMMON blocks that are shared with routines in other files, there will not be enough information saved for the correct checking of set and used status of COMMON blocks and COMMON variables according to the **−usage** setting. Therefore if you plan to use project files when **−usage** checking is turned on (which is the default situation), and if multiple routines in one project file share COMMON blocks with routines in other files, the project files should be created with the **−library** flag turned off. In this mode, **ftnchek** saves, besides the information listed above, one invocation of each subprogram by any other subprogram in the same file, and all COMMON block declarations. This means that the project file will be larger than necessary, and that when it is read in, **ftnchek** may repeat some inter-module checks that it already did when the project file was created. If each project file contains only one module, there is no loss of information in creating the project files in library mode.

Because of the possible loss of information entailed by creating a project file with the **−library** flag in effect, whenever that project file is read in later, it will be treated as a library file regardless of the current setting of the **−library** flag. On the other hand, a project file created with library mode turned off can be read in later in either mode.

The foregoing discussion assumes that the trimming options of the **−project** setting are turned on when the project file is created. This is the normal situation. The **no−trim** options of the **−project** setting are provided in case one wants to use the project files for purposes other than checking the program with **ftnchek**. For instance, one could write a Perl script to analyze the project files for information about how the different subprograms are called. You should not use the **no−trim** options to deal with the issues of information loss discussed above, since they cause more information than necessary to be stored. This makes the project files bigger and causes **ftnchek** to do more work later when it reads them to check your complete program. Ordinarily, you should use the **−library** option to control how much information to store for later use by **ftnchek** in checking your program.

Here is an example of how to use the UNIX **make** utility to automatically create a new project file each time the corresponding source file is altered, and to check the set of files for consistency. Add these lines to your `makefile`. The example assumes that a macro `OBJS` has been defined which lists all the names of object files to be linked together to form the complete executable program. (In this `makefile`, the indented lines should each begin with a tab, not blanks.) If any source file contains multiple routines that share common blocks among themselves, then the `no−com−\*` option should be removed from `NOGLOBAL`, and/or drop the `−library` flag.

```
# tell make what a project file suffix is
.SUFFIXES: .prj

# these options suppress global checks.
NOGLOBAL=-usage=no-ext-undefined,no-com-\*

# tell make how to create a .prj file from a .f file
.f.prj:
	ftnchek −project $(NOGLOBAL) −library $<

# set up macro PRJS containing project filenames
PRJS= $(OBJS:.o=.prj)

# "make check" will check everything that has been changed.
check: $(PRJS)
	ftnchek $(PRJS)
```

When a program uses many routines defined in a large number of different source files in different directories, it can be cumbersome to specify all the different project files needed to check the program properly. To deal with such cases, **ftnchek** allows project files to be concatenated into a single large file. This single file can then be given to **ftnchek** to provide the information for checking the use of any or all of the routines defined in the combined project files. When using such a ''library'' project file, you may want **ftnchek**'s error reports to document precisely the name of the file where the specific function is defined. If the various source files are in several directories, an error report that gives only the file name may be ambiguous, and rather should include the path to the file. The solution is to create each of the individual project files by giving the complete path to the source file. Then this complete path will appear in the error reports. For example, suppose that all of the library subprogram source files are in subdirectories of a directory named `/util/lib`. Then the individual project files could first be created by a command such as

```
find /util/lib -name '*.f' -exec ftnchek -project '{}' ';'
```

(Possibly other options would be provided to **ftnchek** as discussed above. Also, this step could be handled instead by a revised `makefile` rule that would provide the complete source file path instead of just the local name when invoking **ftnchek**.) Next, concatenate all of these project files manually.

```
find /util/lib -name '*.prj' -exec cat '{}' ';' > ourlib.prj
```

Then a program source file can be checked by using the command

```
ftnchek prog.f ... -lib ourlib.prj
```

and an error message related to any library routine will include the full path to the routine's source file.

At present, there is no archive utility like **ar** to manage the contents of a concatenated project file like the one in the illustration above. If changes are made to one of the library routines, the only way to update the combined project file is to concatenate all the individual project files once again. Such a utility would be quite easy to write. Someone should do so and contribute it to the **ftnchek** effort.

AN EXAMPLE

The following simple Fortran program illustrates the messages given by **ftnchek**. The program is intended to accept an array of test scores and then compute the average for the series.

```
C       AUTHORS: MIKE MYERS AND LUCIA SPAGNUOLO
C       DATE:    MAY 8, 1989

C       Variables:
C               SCORE -> an array of test scores
C               SUM ->   sum of the test scores
C               COUNT -> counter of scores read in
C               I ->     loop counter

        REAL FUNCTION COMPAV(SCORE,COUNT)
            INTEGER SUM,COUNT,J,SCORE(5)

            DO 30 I = 1,COUNT
                SUM = SUM + SCORE(I)
30          CONTINUE
            COMPAV = SUM/COUNT
        END


        PROGRAM AVENUM
C
C                       MAIN PROGRAM
C
C       AUTHOR:   LOIS BIGBIE
C       DATE:     MAY 15, 1990
C
C       Variables:
C               MAXNOS -> maximum number of input values
C               NUMS    -> an array of numbers
```

```
C               COUNT   -> exact number of input values
C               AVG     -> average returned by COMPAV
C               I       -> loop counter
C

            PARAMETER(MAXNOS=5)
            INTEGER I, COUNT
            REAL NUMS(MAXNOS), AVG
            COUNT = 0
            DO 80 I = 1,MAXNOS
                READ (5,*,END=100) NUMS(I)
                COUNT = COUNT + 1
80          CONTINUE
100         AVG = COMPAV(NUMS, COUNT)
        END
```

The compiler gives no error messages when this program is compiled. Yet here is what happens when it is run:

```
$ run average
70
90
85
<EOF>
$
```

What happened? Why didn't the program do anything? The following is the output from **ftnchek** when it is used to debug the above program:

```
$ ftnchek -list -symtab average

FTNCHEK Version 3.3 November 2004

File average.f:

      1 C       AUTHORS: MIKE MYERS AND LUCIA SPAGNUOLO
      2 C       DATE:    MAY 8, 1989
      3
      4 C       Variables:
      5 C               SCORE -> an array of test scores
      6 C               SUM ->   sum of the test scores
      7 C               COUNT -> counter of scores read in
      8 C               I ->     loop counter
      9
     10         REAL FUNCTION COMPAV(SCORE,COUNT)
     11             INTEGER SUM,COUNT,J,SCORE(5)
     12
     13             DO 30 I = 1,COUNT
     14                 SUM = SUM + SCORE(I)
     15 30          CONTINUE
     16             COMPAV = SUM/COUNT
                           ^
```

```
Warning near line 16 col 20: integer quotient expr SUM/COUNT  converted to
 real
     17         END

Module COMPAV: func: real

Variables:

      Name Type Dims     Name Type Dims     Name Type Dims     Name Type Dims
    COMPAV real         COUNT intg             I intg*            J intg
     SCORE intg  1        SUM intg

* Variable not declared. Type has been implicitly defined.


Warning in module COMPAV: Variables declared but never referenced:
    J declared at line 11

Warning in module COMPAV: Variables may be used before set:
    SUM used at line 14
    SUM set at line 14


Statement labels defined:

    Label   Line  StmtType
     <30>     15      exec

     18
     19
     20         PROGRAM AVENUM
     21 C
     22 C                       MAIN PROGRAM
     23 C
     24 C       AUTHOR:   LOIS BIGBIE
     25 C       DATE:     MAY 15, 1990
     26 C
     27 C       Variables:
     28 C               MAXNOS -> maximum number of input values
     29 C               NUMS    -> an array of numbers
     30 C               COUNT   -> exact number of input values
     31 C               AVG     -> average returned by COMPAV
     32 C               I       -> loop counter
     33 C
     34
     35             PARAMETER(MAXNOS=5)
     36             INTEGER I, COUNT
     37             REAL NUMS(MAXNOS), AVG
     38             COUNT = 0
     39             DO 80 I = 1,MAXNOS
     40                 READ (5,*,END=100) NUMS(I)
     41                 COUNT = COUNT + 1
     42 80          CONTINUE
     43 100         AVG = COMPAV(NUMS, COUNT)
```

```
     44          END

Module AVENUM: prog

External subprograms referenced:

    COMPAV: real*

Variables:

      Name Type Dims      Name Type Dims      Name Type Dims      Name Type Dims
       AVG real          COUNT intg              I intg        MAXNOS intg*
      NUMS real  1

* Variable not declared. Type has been implicitly defined.


Warning in module AVENUM: Variables set but never used:
    AVG set at line 43

I/O Operations:

     Unit ID Unit No. Access Form Operation   Line
             5          SEQ  FMTD READ         40

Statement labels defined:

    Label   Line  StmtType    Label   Line  StmtType
     <80>     42      exec    <100>     43      exec


 0 syntax errors detected in file average.f
 6 warnings issued in file average.f

Warning: Subprogram COMPAV argument data type mismatch at position 1:
    Dummy arg SCORE in module COMPAV line 10 file average.f is type intg
    Actual arg NUMS in module AVENUM line 43 file average.f is type real
```

According to **ftnchek**, the program contains variables which may be used before they are assigned an initial value, and variables which are not needed. **ftnchek** also warns the user that an integer quotient has been converted to a real. This may assist the user in catching an unintended roundoff error. Since the **−symtab** flag was given, **ftnchek** prints out a table containing identifiers from the local module and their corresponding datatype and number of dimensions. Finally, **ftnchek** warns that the function `COMPAV` is not used with the proper type of arguments.

With **ftnchek**'s help, we can debug the program. We can see that there were the following errors:

1. `SUM` and `COUNT` should have been converted to real before doing the division.
2. `SUM` should have been initialized to 0 before entering the loop.
3. `AVG` was never printed out after being calculated.
4. `NUMS` should have been declared `INTEGER` instead of `REAL`.

We also see that `I`, not `J`, should have been declared `INTEGER` in function `COMPAV`. Also, `MAXNOS` was not declared as `INTEGER`, nor `COMPAV` as `REAL`, in program `AVENUM`. These are not errors, but they may indicate carelessness. As it happened, the default type of these variables coincided with the intended type.

Here is the corrected program, and its output when run:

```
C       AUTHORS: MIKE MYERS AND LUCIA SPAGNUOLO
C       DATE:    MAY 8, 1989
C
C       Variables:
C               SCORE -> an array of test scores
C               SUM ->   sum of the test scores
C               COUNT -> counter of scores read in
C               I ->     loop counter
C
       REAL FUNCTION COMPAV(SCORE,COUNT)
            INTEGER SUM,COUNT,I,SCORE(5)
C
            SUM = 0
            DO 30 I = 1,COUNT
                SUM = SUM + SCORE(I)
30          CONTINUE
            COMPAV = FLOAT(SUM)/FLOAT(COUNT)
        END
C
C
        PROGRAM AVENUM
C
C                       MAIN PROGRAM
C
C       AUTHOR:   LOIS BIGBIE
C       DATE:     MAY 15, 1990
C
C       Variables:
C               MAXNOS -> maximum number of input values
C               NUMS    -> an array of numbers
C               COUNT   -> exact number of input values
C               AVG     -> average returned by COMPAV
C               I       -> loop counter
C
C
            INTEGER MAXNOS
            PARAMETER(MAXNOS=5)
            INTEGER I, NUMS(MAXNOS), COUNT
            REAL AVG,COMPAV
            COUNT = 0
            DO 80 I = 1,MAXNOS
                READ (5,*,END=100) NUMS(I)
                COUNT = COUNT + 1
80          CONTINUE
100         AVG = COMPAV(NUMS, COUNT)
            WRITE(6,*) 'AVERAGE =',AVG
        END
$ run average
70
90
85
<EOF>
```

```
AVERAGE =   81.66666
$
```

With **ftnchek**'s help, our program is a success!

INTERPRETING THE OUTPUT

The messages given by **ftnchek** include not only syntax errors but also warnings and informational messages about things that are legal Fortran but that may indicate errors or carelessness. Most of these messages can be turned off by command-line options. Which option controls each message depends on the nature of the condition being warned about. See the descriptions of the command-line flags in the previous sections, and of individual messages below. Each message is prefixed with a word or phrase indicating the nature of the condition and its severity.

''Error'' means a syntax error. The simplest kind of syntax errors are typographical errors, for example unbalanced parentheses or misspelling of a keyword. This type of error is caught by the parser and appears with the description ''parse error'' or ''syntax error'' (depending on the version of the parser generator and whether it is GNU **bison** or UNIX **yacc**). This type of error message cannot be suppressed. Be aware that this type of error often means that **ftnchek** has not properly interpreted the statement where the error occurs, so that its subsequent checking operations will be compromised. You should eliminate all syntax errors before proceeding to interpret the other messages **ftnchek** gives.

''Warning: Nonstandard syntax'' indicates an extension to Fortran that **ftnchek** supports but that is not according to the Fortran 77 Standard. The extensions that **ftnchek** accepts are described in the section on Extensions below. One example is the `DO` ... `ENDDO` construction. If a program uses these extensions, warnings will be given according to specifications under the **–f77** setting. The default behavior is to give no warnings.

''Warning'' in other cases means a condition that is suspicious but that may or may not be a programming error. Frequently these conditions are legal under the standard. Some are illegal but do not fall under the heading of syntax errors. Usage errors are one example. These refer to the possibility that a variable may be used before it has been assigned a value (generally an error), or that a variable is declared but never used (harmless but may indicate carelessness). The amount of checking for usage errors is controlled by the **–usage** flag, which specifies the maximum amount of checking by default.

Truncation warnings cover situations in which accuracy may be lost unintentionally, for example when a double precision value is assigned to a real variable. These warnings are controlled by the **–truncation** setting, which is on by default.

''Nonportable usage'' warns about some feature that may not be accepted by some compilers even though it is not contrary to the Fortran 77 Standard, or that may cause the program to perform differently on different platforms. For example, equivalencing real and integer variables is usually a non-portable practice. The use of extensions to the standard language is, of course, another source of non-portability, but this is handled as a separate case. To check a program for true portability, both the **–portability** and the **–f77** flags should be used. They are both turned off by default. The **–wordsize** setting is provided to check only those nonportable usages that depend on a particular machine wordsize.

''Possibly misleading appearance'' is used for legal constructions that may not mean what they appear to mean at first glance. For example, Fortran is insensitive to blank space, so extraneous space within variable names or the lack of space between a keyword and a variable can convey the wrong impression to the reader. These messages can be suppressed by turning off the **–pretty** flag, which is on by default.

Other messages that are given after all the files are processed, and having to do with agreement between modules, do not use the word ''warning'' but generally fall into that category. Examples include type mismatches between corresponding variables in different COMMON block declarations, or between dummy and actual arguments of a subprogram. These warnings are controlled by the **–common** and **–arguments** settings respectively. By default both are set for maximum strictness of checking.

Another group of warnings about conditions that are often harmless refer to cases where the array properties of a variable passed as a subprogram argument differ between the two routines. For instance, an array element might be passed to a subroutine that expects a whole array. This is a commonly-used technique for processing single rows or columns of two-dimensional arrays. However, it could also indicate a programming error. The **–array** setting allows the user to adjust the degree of strictness to be used in checking this kind of agreement between actual and dummy array arguments. By default the strictness is maximum.

''Oops'' indicates a technical problem, meaning either a bug in **ftnchek** or that its resources have been exceeded.

The syntax error messages and warnings include the filename along with the line number and column number. **ftnchek** has two different options for the appearance of these error messages. If **–novice** is in effect, which is the default, the messages are in a style approximating normal English. (In default style, the filename is not printed in messages within the body of the program if **–list** is in effect.) The other style of error messages is selected by the **–nonovice** option. In this style, the appearance of the messages is similar to that of the UNIX **lint** program.

ftnchek is still blind to some kinds of syntax errors. The two most important ones are detailed checking of `FORMAT` statements, and almost anything to do with control of execution flow by means of `IF`, `DO`, and `GOTO` statements: namely correct nesting of control structures, matching of opening statements such as `IF ... THEN` with closing statements such as `ENDIF`, and the proper use of statement labels (numbers). Most compilers will catch these errors. See the section on Limitations for a more detailed discussion.

If **ftnchek** gives you a syntax error message when the compiler does not, it may be because your program contains an extension to standard Fortran which is accepted by the compiler but not by **ftnchek**. (See the section on Extensions.) On a VAX/VMS system, you can use the compiler option `/STANDARD` to cause the compiler to accept only standard Fortran. On most UNIX or UNIX-like systems, this can be accomplished by setting the flag **–ansi.**

Many of the messages given by **ftnchek** are self-explanatory. Those that need some additional explanation are listed below in alphabetical order.

Common block NAME: data type mismatch at position n

The *n*-th variable in the COMMON block differs in data type in two different declarations of the COMMON block. By default (**–common** strictness level 3), **ftnchek** is very picky about COMMON blocks: the variables listed in them must match exactly by data type and array dimensions. That is, the legal pair of declarations in different modules:

```
COMMON /COM1/ A,B
```

and

```
COMMON /COM1/ A(2)
```

will cause **ftnchek** to give warnings at strictness level 3. These two declarations are legal in Fortran since they both declare two real variables. At strictness level 1 or 2, no warning would be given in this example, but the warning would be given if there were a data type mismatch, for instance, if `B` were declared `INTEGER`. Controlled by **–common** setting.

Common block NAME has long data type following short data type

Some compilers require alignment of multi-byte items so that each item begins at an address that is a multiple of the item size. Thus if a short (e.g. single-precision real) item is followed by a long (e.g. double precision real) item, the latter may not be aligned correctly. Controlled by **–portability=common–alignment** option.

Common block NAME has mixed character and non-character variables

The ANSI standard requires that if any variable in a COMMON block is of type `CHARACTER`, then all other variables in the same COMMON block must also be of type `CHARACTER`. Controlled by **–f77=mixed–common** option.

Common block NAME: varying length

For **–common** setting level 2, this message means that a COMMON block is declared to have different numbers of words in two different subprograms. A word is the amount of storage occupied by one integer or real variable. For **–common** setting level 3, it means that the two declarations have different numbers of variables, where an array of any size is considered one variable. This is not necessarily an error, but it may indicate that a variable is missing from one of the lists. Note that according to the Fortran 77 Standard, it is an error for named COMMON blocks (but not blank COMMON) to differ in number of words in declarations in different modules. Given for **–common** setting 2 or 3.

Error: Badly formed logical/relational operator or constant

Error: Badly formed real constant

The syntax analyzer has found the start of one of the special words that begin and end with a period (e.g. `.EQ.`), or the start of a numeric constant, but did not succeed in finding a complete item of that kind.

Error: cannot be adjustable size in module NAME

A character variable cannot be declared with a size that is an asterisk in parentheses unless it is a dummy argument, a parameter, or the name of the function defined in the module.

Error: cannot be declared in SAVE statement in module NAME

Only local variables and common blocks can be declared in a `SAVE` statement.

Error: No path to this statement

ftnchek will detect statements which are ignored or by-passed because there is no foreseeable route to the statement. For example, an unnumbered statement (a statement without a statement label), occurring immediately after a `GOTO` statement, cannot possibly be executed.

Error: Parse error

This means that the parser, which analyzes the Fortran program into expressions, statements, etc., has been unable to find a valid interpretation for some portion of a statement in the program. If your compiler does not report a syntax error at the same place, the most common explanations are: (1) use of an extension to ANSI standard Fortran that is not recognized by **ftnchek**, or (2) the statement requires more lookahead than **ftnchek** uses (see section on Bugs).

NOTE: This message means that the affected statement is not interpreted. Therefore, it is possible that **ftnchek**'s subsequent processing will be in error, if it depends on any matters affected by this statement (type declarations, etc.).

Error: Syntax error

This is the same as ‘‘Error: Parse error’’ (see above). It is generated if your version of **ftnchek** was built using the UNIX **yacc** parser generator rather than GNU **bison**.

Identifiers which are not unique in first six chars

Warns that two identifiers which are longer than 6 characters do not differ in the first 6 characters. This is for portability: they may not be considered distinct by some compilers. Controlled by **–sixchar** option.

Nonportable usage: argument precision may not be correct for intrinsic function

The precision of an argument passed to an intrinsic function may be incorrect on some computers. Issued when a numeric variable declared with explicit precision (e.g. `REAL*8 X`) is passed to a specific intrinsic function (e.g. `DSQRT(X)`). Controlled by **–portability=mixed–size** and **–wordsize**.

Nonportable usage: character constant/variable length exceeds 255

Some compilers do not support character strings more than 255 characters in length. Controlled by **−portability=long−string**.

Nonportable usage: File contains tabs

ftnchek expands tabs to be equivalent to spaces up to the next column which is a multiple of 8. Some compilers treat tabs differently, and also it is possible that files sent by electronic mail will have the tabs converted to blanks in some way. Therefore files containing tabs may not be compiled correctly after being transferred. **ftnchek** does not give this message if tabs only occur within comments or character constants. Controlled by **−portability=tab**.

Nonportable usage: non-integer DO loop bounds

This warning is only given when the `DO` index and bounds are non-integer. Use of non-integer quantities in a `DO` statement may cause unexpected errors, or different results on different machines, due to roundoff effects. Controlled by **−portability=real−do**.

Possibly it is an array which was not declared

This message is appended to warnings related to a function invocation or to an argument type mismatch, for which the possibility exists that what appears to be a function is actually meant to be an array. If the programmer forgot to dimension an array, references to the array will be interpreted as function invocations. This message will be suppressed if the name in question appears in an `EXTERNAL` or `INTRINSIC` statement. Controlled by the **−novice** option.

Possibly misleading appearance: characters past 72 columns

The program is being processed with the statement field width at its standard value of 72, and some nonblank characters have been found past column 72. In this case, **ftnchek** is not processing the characters past column 72, and is notifying the user that the statement may not have the meaning that it appears to have. These characters might be intended by the programmer to be significant, but they will be ignored by the compiler. Controlled by **−pretty=long−line**.

Possibly misleading appearance: Common block declared in more than one statement

Such multiple declarations are legal and have the same effect as a continuation of the original declaration of the block. This warning is only given if the two declarations are separated by one or more intervening statements. Controlled by **−pretty=multiple−common**.

Possibly misleading appearance: Continuation follows comment or blank line

ftnchek issues this warning message to alert the user that a continuation of a statement is interspersed with comments, making it easy to overlook. Controlled by **−pretty=continuation**.

Possibly misleading appearance: Extraneous parentheses

Warns about parentheses surrounding a variable by itself in an expression. When a parenthesized variable is passed as an argument to a subprogram, it is treated as an expression, not as a variable whose value can be modified by the called routine. Controlled by **−pretty=parentheses**.

Subprogram NAME: argument data type mismatch at position n

The subprogram's *n*-th actual argument (in the `CALL` or the usage of a function) differs in datatype or precision from the *n*-th dummy argument (in the `SUBROUTINE` or `FUNCTION` declaration). For instance, if the user defines a subprogram by

```
SUBROUTINE SUBA(X)
REAL X
```

and elsewhere invokes `SUBA` by

```
CALL SUBA(2)
```

ftnchek will detect the error. The reason here is that the number 2 is integer, not real. The user should have written

```
CALL SUBA(2.0)
```

When checking an argument which is a subprogram, **ftnchek** must be able to determine whether it is a function or a subroutine. The rules used by **ftnchek** to do this are as follows: If the subprogram, besides being passed as an actual argument, is also invoked directly elsewhere in the same module, then its type is determined by that usage. If not, then if the name of the subprogram does not appear in an explicit type declaration, it is assumed to be a subroutine; if it is explicitly typed it is taken as a function. Therefore, subroutines passed as actual arguments need only be declared by an `EXTERNAL` statement in the calling module, whereas functions must also be explicitly typed in order to avoid generating this error message. Controlled by **−arguments** setting.

Subprogram NAME: argument arrayness mismatch at position n

Similar to the preceding situation, but the subprogram dummy argument differs from the corresponding actual argument in its number of dimensions or number of elements. Controlled by **−array** together with **−arguments** settings.

Subprogram NAME: argument mismatch at position n

A character dummy argument is larger than the corresponding actual argument, or a Hollerith dummy argument is larger than the corresponding actual argument. Controlled by **−arguments** setting.

Subprogram NAME: argument usage mismatch

ftnchek detects a possible conflict between the way a subprogram uses an argument and the way in which the argument is supplied to the subprogram. The conflict can be one of two types, as outlined below.

Dummy arg is modified, Actual arg is const or expr

A dummy argument is an argument as named in a `SUBROUTINE` or `FUNCTION` statement and used within the subprogram. An actual argument is an argument as passed to a subroutine or function by the caller. **ftnchek** is saying that a dummy argument is modified by the subprogram, implying that its value is changed in the calling module. The corresponding actual argument should not be a constant or expression, but rather a variable or array element which can be legitimately assigned to. Controlled by the **−usage=arg−const−modified** option.

Dummy arg used before set, Actual arg not set

Here a dummy argument may be used in the subprogram before having a value assigned to it by the subprogram. The corresponding actual argument should have a value assigned to it by the caller prior to invoking the subprogram. Controlled by the **−usage=var−uninitialized** option.

This warning is not affected by the **−arguments** setting.

Subprogram NAME invoked inconsistently

Here the mismatch is between the datatype of the subprogram itself as used and as defined. For instance, if the user declares

```
INTEGER FUNCTION COUNT(A)
```

and invokes `COUNT` in another module as

```
N = COUNT(A)
```

without declaring its datatype, it will default to real type, based on the first letter of its name. The calling module should have included the declaration

```
INTEGER COUNT
```

Given for **−arguments** setting 2 or 3.

Subprogram NAME: varying length argument lists:

An inconsistency has been found between the number of dummy arguments (parameters) a subprogram has and the number of actual arguments given it in an invocation. **ftnchek** keeps track of all invocations of subprograms (`CALL` statements and expressions using functions) and compares them with the definitions of the subprograms elsewhere in the source code. The Fortran compiler normally does not catch this type of error. Given for **−arguments** setting 1 or 3.

Variable not declared. Type has been implicitly defined

When printing the symbol table for a module, **ftnchek** will flag with an asterisk all identifiers that are not explicitly typed and will show the datatype that was assigned through implicit typing. This provides support for users who wish to declare all variables as is required in Pascal or some other languages. This message appears only when the **−symtab** option is in effect. Alternatively, use the **−declare** flag if you want to get a list of all undeclared variables.

Variables declared but never referenced

Detects any identifiers that were declared in your program but were never used, either to be assigned a value or to have their value accessed. Variables in COMMON are excluded. Controlled by the **−usage=var−unused** option.

Variables set but never used

ftnchek will notify the user when a variable has been assigned a value, but the variable is not otherwise used in the program. Usually this results from an oversight. Controlled by the **−usage=var−set−unused** option.

Variables used before set

This message indicates that an identifier is used to compute a value prior to its initialization. Such usage may lead to an incorrect value being computed, since its initial value is not controlled. Controlled by the **−usage=var−uninitialized** option.

Variables may be used before set

Similar to used before set except that **ftnchek** is not able to determine its status with certainty. **ftnchek** assumes a variable may be used before set if the first usage of the variable occurs prior in the program text to its assignment. Controlled by the **−usage=var−uninitialized** option.

Warning: DO index is not integer

This warning is only given when the `DO` bounds are integer, but the `DO` index is not. It may indicate a failure to declare the index to be an integer. Controlled by **−truncation=real−do** option.

Warning: integer quotient expr ... converted to real

The quotient of two integers results in an integer type result, in which the fractional part is dropped. If such an integer expression involving division is later converted to a real datatype, it may be that a real type division had been intended. Controlled by **−truncation=int−div−real** option.

Warning: Integer quotient expr ... used in exponent

The quotient of two integers results in an integer type result, in which the fractional part is dropped. If such an integer expression is used as an exponent, it is quite likely that a real type division was intended. Controlled by **−truncation=int−div−exponent** option.

Warning: NAME not set when RETURN encountered

The way that functions in Fortran return a value is by assigning the value to the name of the function. This message indicates that the function was not assigned a value before the point where a `RETURN` statement was found. Therefore it is possible that the function could return an undefined value.

Warning: Nonstandard syntax: adjustable size cannot be concatenated here

The Fortran 77 Standard (sec. 6.2.2) forbids concatenating character variables whose size is an asterisk in parentheses, except in an assignment statement. Controlled by **−f77=mixed−expr**.

Warning: Nonstandard syntax : significant characters past 72 columns

This warning is given under the **−f77=long−line** setting if the **−columns** setting has been used to increase the statement field width, and a statement has meaningful program text beyond column 72. Standard Fortran ignores all text in those columns, but some compilers do not. Thus the program may be treated differently by different compilers.

Warning: Nonstandard syntax : Statement out of order.

ftnchek will detect statements that are out of the sequence specified for ANSI standard Fortran 77. Table 1 illustrates the allowed sequence of statements in the Fortran language. Statements which are out of order are nonetheless interpreted by **ftnchek**, to prevent "cascades" of error messages. The sequence counter is also rolled back to prevent repetition of the error message for a block of similar statements. Controlled by the **−f77=statement−order** option.

```
--------------------------------------------------------
                  |               | implicit
                  |  parameter    |---------------------
                  |               | other specification
        format    |---------------|---------------------
         and      |               | statement-function
        entry     |  data         |---------------------
                  |               | executable
--------------------------------------------------------
```

Table 1

Warning: Possible division by zero

This message is printed out wherever division is done (except division by a constant). Use it to help locate a runtime division by zero problem. Controlled by **−division** option.

Warning: real truncated to intg

ftnchek has detected an assignment statement which has a real expression on the right, but an integer variable on the left. The fractional part of the real value will be lost. If you explicitly convert the real expression to integer using the `INT` or `NINT` intrinsic function, no warning will be printed. A similar message is printed if a double precision expression is assigned to a single precision variable, etc. Controlled by **−truncation=demotion** option.

Warning: subscript is not integer

Since array subscripts are normally integer quantities, the use of a non-integer expression here may signal an error. Controlled by **−truncation=real−subscript** option.

Warning: Unknown intrinsic function

This message warns the user that a name declared in an `INTRINSIC` statement is unknown to **ftnchek**. Probably it is a nonstandard intrinsic function, and so the program will not be portable. The function will be treated by **ftnchek** as a user-defined function. This warning is not suppressed by any option, since it affects **ftnchek**'s analysis of the program. However, if the intrinsic function is in one of the supported sets of nonstandard intrinsics, you can use the **−intrinsic** setting to cause **ftnchek** to recognize it.

LIMITATIONS AND EXTENSIONS

ftnchek accepts ANSI standard Fortran-77 programs with some minor limitations and numerous common extensions.

Limitations:

The dummy arguments in statement functions are treated like ordinary variables of the program. That is, their scope is the entire subprogram, not just the statement function definition.

The checking of FORMAT statements is lax, tolerating missing separators (comma, etc.) between format descriptors in places where the Standard requires them, and allowing *.d* fields on descriptors that should not have them. It does warn under **−f77=format−edit−descr** about nonstandard descriptor types (like `O`), and supported extensions.

There are some syntactic extensions and Fortran 90 elements that **ftnchek** accepts but does very little checking. For instance, pointer usage (whether the nonstandard Cray syntax or the Fortran 90 syntax) is not checked other than for set and used status. It is hoped that some day more thorough checking will be implemented, but for now the user should regard the acceptance of these syntactic features simply as a convenience to enable checking of other aspects of code that contains them. See the section Extensions for specifics about what features are accepted but not fully checked.

If a user-supplied subprogram has the same name as one of the nonstandard intrinsic functions recognized by **ftnchek**, it must be declared in an `EXTERNAL` statement in any routine that invokes it. Otherwise it will be subject to the checking normally given to the intrinsic function. Since the nonstandard intrinsics are not standard, this `EXTERNAL` statement is not required by the Fortran 77 Standard. Using the **−intrinsic=none** setting, recognition of most nonstandard intrinsics (excepting only those needed to support the double complex data type) can be turned off. See the lists of supported nonstandard intrinsic functions under the discussion of the **−intrinsic** setting above.

Extensions:

All of these extensions (except lower-case characters) will generate warnings if the relevant **−f77** option is set. Some of the extensions listed below are part of the Fortran-90 Standard. These are indicated by the notation (F90).

Tabs are permitted, and translated into equivalent blanks which correspond to tab stops every 8 columns. The standard does not recognize tabs. Note that some compilers allow tabs, but treat them differently. The treatment defined for DEC FORTRAN can be achieved using the **−source=dec−tab** setting.

Strings may be delimited by either quote marks or apostrophes. A sequence of two delimiter characters is interpreted as a single embedded delimiter character. (F90)

Strings may contain UNIX-style backslash escape sequences. They will be interpreted as such if the **−source=unix−backslash** setting is given. Otherwise the backslash character will be treated as a normal printing character.

Source code can be in either Fortran 90 free format or traditional fixed format. (F90)

A semicolon is allowed as a statement separator. (F90)

Lower case characters are permitted, and are converted internally to uppercase except in character strings. The standard specifies upper case only, except in comments and strings. (F90)

Hollerith constants are permitted, in accordance with the Fortran 77 Standard, appendix C. They should not be used in expressions, or confused with datatype CHARACTER.

The letter 'D' (upper or lower case) in column 1 is treated as the beginning of a comment. There is no option to treat such lines as statements instead of comments.

Statements may be longer than 72 columns provided that the setting **−columns** was used to increase the limit. According to the standard, all text from columns 73 through 80 is ignored, and no line may be longer than 80 columns.

Variable names may be longer than six characters. The standard specifies six as the maximum. **ftnchek** permits names up to 31 characters long (F90).

Variable names may contain underscores and dollar signs (or other non-alphabetic characters as specified by the **−identifier−chars** option). These characters are are treated the same as alphabetic letters. The default type for variables beginning with these characters is REAL. In IMPLICIT type statements specifying a range of characters, the dollar sign follows Z and is followed by underscore. (Any other user-defined characters are treated the same as the dollar sign.) Fortran 90 permits underscores in variable names.

The UNIX version tolerates the presence of preprocessor directives, namely lines beginning with the pound sign (#). These are treated as comments, except for `#line` directives, which are interpreted, and are used to set the line number and source file name for warnings and error messages. Note that `#include` directives are not processed by **ftnchek**. Programs that use them for including source files should be passed through the preprocessor before being input to **ftnchek**. As noted below, **ftnchek** does process `INCLUDE` statements, which have a different syntax. An optional program, **ftnpp**(1L) (available separately) provides preprocessing that properly handles `INCLUDE` files.

The Fortran 90 `DO` ... `ENDDO` control structure is permitted. The `CYCLE` and `EXIT` statements are accepted. All of these may have an optional do-construct name, but construct names are not checked for consistency. (F90)

The Fortran 90 `SELECT CASE` construct is accepted. (F90)

Construct names are also accepted on `IF`, `THEN`, `ELSE`, `ENDIF` and `SELECT CASE` statements. (F90)

The `ACCEPT` and `TYPE` statements (for terminal I/O) are permitted, with the same syntax as `PRINT`.

The so-called "Cray pointer" syntax is tolerated. It is not the same as the Fortran 90 `POINTER` statement. There is no real checking of the statement other than basic syntax. The form of this statement is

`POINTER (`*pointer*`, `*pointee*`) [,(`*pointer*`, `*pointee*`)]`

The pointer variables are assigned a data type of `INTEGER *4`. Usage checking of the pointee variables is suppressed, since in practice they are accessed indirectly via the pointers.

The following Fortran 90 pointer related syntaxes are accepted: `ALLOCATABLE`, `POINTER`, and `TARGET` statements and the same attributes in type declarations; `ALLOCATE`, `DEALLOCATE`, and `NULLIFY` executable statements; pointer assignment using => operator; and the intrinsic functions `ALLOCATED` and `ASSOCIATED`. Little semantic checking of pointer variables and operations is done beyond basic set and used status. For instance, there is no checking for such errors as dangling pointers, or use of unallocated arrays.

Statements may have any number of continuation lines. The Fortran 77 and Fortran 90 standards allow a maximum of 19 in fixed source form. The Fortran 90 standard allows a maximum of 39 in free source form.

Relational (comparison) operators composed of punctuation, namely: < <= == /= > >= are allowed. (F90)

Inline comments, beginning with an exclamation mark, are permitted. (F90)

NAMELIST I/O is supported. The syntax is the same as in Fortran 90.

FORMAT statements can contain a dollar sign to indicate suppression of carriage-return. An integer expression enclosed in angle brackets can be used anywhere in a FORMAT statement where the Fortran 77 Standard allows an integer constant (except for the length of a Hollerith constant), to provide a run-time value for a repeat specification or field width.

Nonstandard keywords are allowed in I/O statements, corresponding to those in VMS Fortran.

The `IMPLICIT NONE` statement is supported. The meaning of this statement is that all variables must have their data types explicitly declared. Rather than flag the occurrences of such variables with syntax error messages, **ftnchek** waits till the end of the module, and then prints out a list of all undeclared variables, as it does for the **−declare** option. (F90)

Data types `INTEGER`, `REAL`, `COMPLEX`, and `LOGICAL` are allowed to have an optional precision specification in type declarations. For instance, `REAL*8` means an 8-byte floating point data type. The `REAL*8` datatype is not necessarily considered equivalent to `DOUBLE PRECISION`, depending on the **−wordsize** setting. The Fortran 77 Standard allows a length specification only for `CHARACTER` data.

ftnchek supports the `DOUBLE COMPLEX` type specification for a complex quantity whose real and imaginary parts are double precision. Mixed-mode arithmetic involving single-precision complex with double-precision real data, prohibited under the Standard, yields a double complex result.

Combined type declarations and data-statement-like initializers are accepted. These have the form of a standard Fortran 77 type declaration, followed by a slash-delimited list of constants like that used in a `DATA` statement. An example of the syntax is

```
INTEGER  N / 100 /
```

This bastard form of initializing declaration was not adopted in Fortran 90. Such declarations should be written using the standard form described below, which is accepted by **ftnchek**.

There is limited support for Fortran 90 attribute-based type declarations. This style of declaration is distinguished by the use of a double colon (::) between the list of attributes and the list of declared variables. The features supported may be adequate for novice programmers, but are not yet sufficient for professional-quality Fortran 90 programs. I hope to add support for more features in future releases. I invite volunteers to assist in this task. See the ToDo file in the source code distribution for details. The attributes currently accepted, besides all the usual data types, are `DIMENSION`, `EXTERNAL`, `INTRINSIC`, `PARAMETER`, and `SAVE`. The new form of declaration also allows assignment of values to the variables declared. At present, the `(LEN=`*value*`)` form of specifying character lengths is also accepted. Kind specifications, using `(KIND=`*value*`)` are parsed but are not processed: all kinds are treated as default kind. Also, there is little checking of these declarations beyond basic syntax.

Many commonly found nonstandard intrinsic functions are provided. See the discussion of **−intrinsic** for a list of functions and how to control which ones are recognized.

Argument checking is not tight for those nonstandard intrinsics that take arrays or mixed argument types.

ftnchek permits the `INCLUDE` statement, which causes inclusion of the text of the given file. The syntax is

```
INCLUDE 'filename'
```

This is compatible with Fortran 90. If the **−source=vms−include** option is given, **ftnchek** follows VMS conventions with respect to this statement: it assumes a default extension of *.for* if no filename extension is given, and allows the qualifier `/[NO]LIST` following the filename, to control the listing of the included file. There is no support for including VMS text modules.

In diagnostic output relating to items contained in include files, the location of the error is specified by both its location in the include file and the location in the parent file where the file was included.

ftnchek accepts `PARAMETER` statements which lack parentheses. These will be warned about if the **−f77=param−noparen** flag is given.

ftnchek accepts `PARAMETER` definitions that involve intrinsic functions and exponentiation by a non-integer exponent. Both of these cases are prohibited by the Fortran 77 Standard, and will be warned about if the **−f77=param−intrinsic** flag is given. If an intrinsic function value is a compile-time integer constant, **ftnchek** will evaluate it. This allows better checking if the parameter is used in declaring array sizes. Fortran 90 allows intrinsic functions in `PARAMETER` definitions.

The intrinsic functions that are evaluated are:

```
ABS     IABS   DIM     IDIM    MAX
MAX0    MIN    MIN0    MOD     SIGN
ISIGN   LEN    ICHAR   INDEX
```

The functions of integer arguments are evaluated only if the arguments are integer constant expressions. (These may involve integer constants, parameters, and evaluated intrinsic functions.) The function `LEN` is evaluated if its argument is an expression involving only character constants and variables whose length is not adjustable. The functions `ICHAR` and `INDEX` are evaluated only if the arguments are character constants. **ftnchek** gives a warning if it needs the value of some intrinsic function that is not evaluated.

NEW FEATURES

Here are the changes from Version 3.2 to Version 3.3:

1. Front-end has been rewritten for unlimited lookahead, eliminating the longstanding bug that caused incorrect interpretation of statements whose ambiguity was not resolved in the first line.
2. The **−mkhtml** option is now available in the MS-DOS version.
3. Added support for Fortran 90 pointer related syntax: `ALLOCATE`, `DEALLOCATE`, and `NULLIFY` statements; the `ALLOCATABLE`, `POINTER` and `TARGET` attributes in type declarations; the pointer assignment operator `=>` and intrinsic functions `ALLOCATED` and `ASSOCIATED`; and deferred-shape array declarations. At present these new syntax features are accepted but not properly checked. This feature was added by Robert Landrito.
4. The **−f77** and **−f90 pointer** option controlling warnings about "Cray pointers" has been renamed to **cray−pointer**. The **−f77=pointer** option now instead controls warnings for code containing Fortran 90 pointer-related syntax.
5. Re-implemented **−mkhtml** processing so it is now much faster on source files containing many routines.
6. Changed the arrangement of the test directory so there is no longer any need to modify the distribution in order to run the test suite (*check. bat*) under MS-DOS.
7. Fixed bug in reading numeric settings on command line when setting name abbreviated to 3 characters.
8. Fixed bug causing spurious warning for a `GOTO` referring to a labeled `END` statement when the statement before `END` was a `FORMAT`.
9. New flag **−f77=character** to control warnings about extensions to the Fortran 77 character data type. Accompanying this new flag is support for Fortran 90 rules for character variable declarations that evaluate to zero or negative length, allowing them and treating negative length values as zero.
10. Fixed minor bug in printing of comments and blank lines following last `END` statement in **−list** mode.

BUGS

ftnchek still has much room for improvement. Your feedback is appreciated. We want to know about any bugs you notice. Bugs include not only cases in which **ftnchek** issues an error message where no error exists, but also if **ftnchek** fails to issue a warning when it ought to. Note, however, that **ftnchek** is not intended to catch all syntax errors (see section on Limitations). Also, it is not considered a bug for a variable to be reported as used before set, if the reason is that the usage of the variable occurs prior in the text to where the variable is set. For instance, this could occur when a `GOTO` causes execution to loop backward to some previously skipped statements. **ftnchek** does not analyze the program flow, but assumes that statements occurring earlier in the text are executed before the following ones.

We especially want to know if **ftnchek** crashes for any reason. It is not supposed to crash, even on programs with syntax errors. Suggestions are welcomed for additional features which you would find useful. Tell us if any of **ftnchek**'s messages are incomprehensible. Comments on the readability and accuracy of this document are also welcome.

You may also suggest support for additional extensions to the Fortran language. These will be included only if it is felt that the extensions are sufficiently widely accepted by compilers.

If you find a bug in **ftnchek**, first consult the list of known bugs below to see if it has already been reported. Also check the section entitled "Limitations and Extensions" above for restrictions that could be causing the problem. If you do not find the problem documented in either place, then send a report including

1. The operating system and CPU type on which **ftnchek** is running.
2. The version of **ftnchek** and values of any environment options or settings defined in startup file. (Capturing the output of `ftnchek -help` is useful for this.)
3. A brief description of the bug.
4. If possible, a small sample program showing the bug.

The report should be sent to Dr. Robert Moniot (see contact information in section entitled "Installation and Support").

Highest priority will be given to bugs which cause **ftnchek** to crash.

Certain problems that arise when checking large programs can be fixed by increasing the sizes of the data areas in **ftnchek**. (These problems are generally signaled by error messages beginning with "Oops".) The simplest way to increase the table sizes is by recompiling **ftnchek** with the `LARGE_MACHINE` macro name defined. Consult the `makefile` and `README` file for the method of doing this.

The following is a list of known bugs.

1. Bug: Used-before-set message is suppressed for any variable which is used as the loop index in an implied-do loop, even if it was in fact used before being set in some earlier statement. For example, consider `J` in the statement

   ```
   WRITE(5,*) (A(J), J=1,10)
   ```

   Here **ftnchek** parses the I/O expression, `A(J)`, where `J` is used, before it parses the implied loop where `J` is set. Normally this would cause **ftnchek** to report a spurious used-before-set warning for `J`. Since this report is usually in error and occurs fairly commonly, **ftnchek** suppresses the warning for `J` altogether.

   Prognosis: A future version of **ftnchek** is planned which will handle implied-do loops correctly.

2. Bug: Variables used (not as arguments) in statement-function subprograms do not have their usage status updated when the statement function is invoked.

   Prognosis: To be fixed in a future version of **ftnchek**.

3. Bug: VAX version does not expand wildcards in filenames on the command line if they are followed without space by an option, e.g. `ftnchek *.f/calltree` would not expand the `*.f`. This is because VMS-style options without intervening space are not supported by the GNU `shell_mung` routine that is used to expand wildcards.

   Prognosis: unlikely to be fixed.

4. Bug: checking for nonstandard format edit descriptors is done only in `FORMAT` statements, not in character strings used as formats.

   Prognosis: may be fixed someday.

ACKNOWLEDGEMENTS

ftnchek was designed by Dr. Robert Moniot, professor at Fordham University. During the academic year of 1988-1989, Michael Myers and Lucia Spagnuolo developed the program to perform the variable usage checks. During the following year it was augmented by Lois Bigbie to check subprogram arguments and COMMON block declarations. Brian Downing assisted with the implementation of the `INCLUDE` statement. John Quinn wrote the common block usage checks. Heba Elsayed wrote the label table printout and label usage checks. Nelson H. F. Beebe of the University of Utah added most of the new code to implement the **–makedcls** feature and wrote the dcl2inc script. The **–mkhtml** feature was contributed by Mark McVeigh of Framatome ANP, Inc. The **–reference** feature was contributed by Gerome Emmanuel, Ecole des mines, U. Nancy (slightly modified). The **–vcg** option was contributed by Dr. Philip Rubini of Cranfield University, UK. The support for Cray pointer syntax was provided by John Dannenhoffer of United Technologies Research Center. John C. Bollinger of Indiana University added the parser syntax for the `SELECT CASE` construct. Robert Landrito added the parser syntax for F90 pointer-related features. Additional features will be added as time permits. As of Version 2.5, the name was changed from **forchek** to **ftnchek**, to avoid confusion with a similar program named **forcheck**, developed earlier at Leiden University.

We would like to thank John Amor of the University of British Columbia, Reg Clemens of the Air Force Phillips Lab in Albuquerque, Markus Draxler of the University of Stuttgart, Victor Eijkhout of the University of Tennessee at Knoxville, Greg Flint of Purdue University, Daniel P. Giesy of NASA Langley Research Center, Fritz Keinert of Iowa State University, Judah Milgram of the University of Maryland College Park, Hugh Nicholas of the Pittsburgh Supercomputing Center, Dan Severance of Yale University, Phil Sterne of Lawrence Livermore National Laboratory, Larry Weissman of the University of Washington, Warren J. Wiscombe of NASA Goddard, and Nelson H. F. Beebe of the University of Utah, for pointing out bugs and suggesting some improvements. Stefan A. Deutscher, Gunnar Duus, Clive Page of the University of Leicester, Stephan Wefing of Heidelberg University, and Bob Wells of Oxford University were extremely helpful as alpha testers. We also thank Jack Dongarra for putting **ftnchek** into the `netlib` library of publicly available software.

INSTALLATION AND SUPPORT

The **ftnchek** program is free software. It can be obtained by anonymous ftp from many software servers, including ftp://netlib.org/fortran . Note that on Netlib the distribution is named *ftnchek.tar.gz* whereas on most other servers the file name includes the version number, e.g. *ftnchek-3.3.0.tar.gz*. If the file extension is *.Z*, uncompress with the Unix **uncompress**(1) utility. If the file extension is *.gz*, uncompress with the GNU **gunzip**(1L) program. Then use **tar**(1) to unpack the files into a subdirectory.

Installation requires a C compiler for your computer. See the `INSTALL` file provided with the distribution for instructions on installing **ftnchek** on your system. Executable binary for particular systems such as IBM PC or Macintosh, as available, can be obtained by anonymous ftp from ftp://ftp.dsm.fordham.edu/pub/ftnchek . Assistance in preparing such executable binary forms is welcome.

The **nroff** version of this document is named *ftnchek.man*. On UNIX systems, this file can be used as the man page, but actually it is a multi-purpose source file which is used to produce the other forms of the documentation. The cleaned-up man page document, created during installation of **ftnchek**, is named *ftnchek.1*. The distribution also includes a plain ASCII version named *ftnchek.doc*, a PostScript version named *ftnchek.ps*, an HTML version in directory *html*, and a VMS HELP version named *ftnchek.hlp*.

Information about the latest version and the status of the project can be obtained by visiting **ftnchek**'s home page, http://www.dsm.fordham.edu/~ftnchek . For further information and to report bugs, you may contact Dr. Robert Moniot, whose contact information can be found by a Web search for his name and Fordham University. (E-mail address is not provided here because it attracts unsolicited commercial e-mail, but it is easily constructed by combining his last name with the name of the university and the edu domain.)

SEE ALSO

dcl2inc(1L), **dtoq**(1L), **dtos**(1L), **f77**(1), **fd2s**(1L), **fs2d**(1L), **ftnpp**(1L), **pfort**(1L), **qtod**(1L), **sf3**(1L), **stod**(1L). **xsf3**(1L), **xvcg**(1L).