NAME

ftnchek – Fortran program checker

SYNOPSIS

ftnchek [**–array=***num*] [**–[no]calltree**] [**–columns=***num*] [**–common=***num*] [**–[no]declare**]
[**–[no]division**] [**–[no]extern**] [**–[no]f77**] [**–[no]help**] [**–[no]hollerith**]
[**–include=***str*] [**–[no]library**] [**–[no]linebreak**] [**–[no]list**] [**–[no]novice**]
[**–output=***str*] [**–[no]portability**] [**–[no]pretty**] [**–[no]project**] [**–[no]pure**]
[**–[no]sixchar**] [**–[no]symtab**] [**–[no]truncation**] [**–usage=***num*]
[**–[no]verbose**] [*files ...*]

INTRODUCTION

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 '–'.) 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' 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. Only the first 3 characters of an option name (not counting the '–') need be provided.

The switches which **Ftnchek** currently recognizes are:

–calltree
: Print tree of subprogram call hierarchy. Default = no.

–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.

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

–f77 Warn about extensions to the Fortran 77 standard. Default = no.

–help Print command summary. Default = no.

–hollerith
Warn about hollerith constants under **–port** option. Default = yes.

–library
Begin library mode: do not warn if subprograms in file are defined but never used. Default = no.

–linebreak
Treat linebreaks in continued statements as space. Default = yes.

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

–novice Give warnings suitable for novice users.

–portability
Warn about non-portable usages. Default = no.

–pretty Give certain messages related to appearance of source code. Default = yes.

–project
Create project file (see explanation below). Default = no.

–pure Assume functions have no side effects. Default = yes.

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

–symtab
Print out symbol table. Default = no.

–truncation
Check for possible truncation errors. Default = yes.

–verbose
Produce full amount of output. Default = yes.

There are six settings:

–array=*n*
Set level of strictness in checking array arguments of subprograms. Min is 0 (least checking). Max is 3 (most checking). Default = 3.

–columns=*n*
Set maximum line length to *n* columns. (Beyond this is ignored.) Max is 132. Default = 72.

–common=*n*
Set level of strictness in checking `COMMON` blocks. Min is 0 (no checking). Max is 3 (must be identical). Default = 3.

–include=*path*
Define a directory to search for include files. Cumulative.

–output=*filename*
Send output to the given file. Default is to send output to the screen. (Default filename extension is *.lis*).

–usage=*n*
Control warnings about unused variables, etc. Min is 0 (no checking). Default = 3 (most checking).

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 list of all options and settings.

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.

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

FTNCHEK OPTIONS

This section provides a more detailed discussion of **Ftnchek** command-line options. Options and filenames may be interspersed on a command line. Each option remains in effect from the point it is encountered until it is overridden by a later option. Thus for example, the listing may be suppressed for some files and not for others.

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

–array=*num*

Controls warnings about mismatches between actual and dummy subprogram array arguments. (An actual argument is an argument passed to the subprogram by the caller; a dummy argument is an argument received by the subprogram.) Default = 3. The warnings which can be turned off are for constructions that might legitimately be used by a knowledgeable programmer, but that often indicate programming errors.

The meanings of the setting values are as follows:

0: only the warnings noted below.

1: give warnings 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.

2: give warnings if the arguments are both arrays, but differ in size.

3: give both types of warnings.

Note: no warning is ever given if the actual argument is an array element while the dummy argument is a scalar variable, and 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. Variable-dimensioned arrays match any array size.

–calltree

Causes **Ftnchek** to print out the call structure of the complete program in the form of a tree. The tree 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. Default = no.

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)".

Note that the call tree will be incomplete if any of the input files are project files 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. If no main program is found, a report to that effect is printed out, and no call tree is printed. If multiple main programs are found, the call tree of each is printed separately.

Now that **Ftnchek** recognizes the call tree structure of a program, its checking behavior is somewhat altered from previous versions, which checked the calls of every routine by every other routine, regardless of whether those routines could ever actually be invoked at run time. Now, if a file is read with the **–library** flag in effect, the calls made by a routine in that file will be checked only if the calling routine is in the main program's call tree. Likewise, `COMMON` declarations in a library file will only be checked if the routine is in the call tree. If the **–library** flag is not set,

Ftnchek will check all inter-module calls and all common declarations, as it did formerly. (If there is no main program anywhere in the set of files that **Ftnchek** has read, so that there is no call tree, then library routines will be checked if they are called by any routine in the complete set of files.)

–columns=*n*
Set maximum statement length to *n* 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** option is in effect, a warning will be given for any overlength lines that are processed. Max is 132. Default = 72.

–common=*n*
This setting varies the strictness of checking of `COMMON` blocks. Default = 3.

The different levels are:

0: no checking.

1: in each declaration of a given `COMMON` block, corresponding memory locations must agree in data type.

2: also warn if different declarations of the same block are not equal in total length.

3: corresponding variables in each declaration of a block must agree in data type and (if arrays) in size and number of dimensions.

–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.

–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.

–extern Causes **Ftnchek** to report whether any subprograms invoked by the program are never defined, or are multiply defined. Ordinarily, if **Ftnchek** is being run on a complete program, each subprogram other than the intrinsic functions should be defined once and only once somewhere. Turn off this switch if you just want to check a subset of files which form part of a larger complete program, or to check all at once a number of unrelated files which might each contain an unnamed main program. Subprogram arguments will still be checked for correctness. Default = yes.

–f77 Use this flag 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. Default=no.

–help This command is identical in function to the **"?"** argument, and is provided simply as a convenience for those systems in which the question mark has special meaning to the command interpreter. Default = no.

–hollerith

Hollerith constants (other than within format specifications) are a source of possible portability problems, so when the **–portability** flag is set, warnings about them will be produced. If your program uses many hollerith constants, these warnings can obscure other more serious warnings. So you can set this flag to "no" to suppress the warnings about holleriths. This flag has no effect when the **–portability** flag is turned off. Default = yes.

–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 systemwide directory (`/usr/include` for Unix, `SYS$LIBRARY` for VMS, and `\include` for MSDOS).

–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.

–linebreak

Normally, when scanning a statement which is continued onto the next line, **Ftnchek** treats the end of the line as a space. This behavior is the same as for Pascal and C, and also corresponds to how humans normally would read and write programs. However, occasionally one would like to use **Ftnchek** to check a program in which identifiers and keywords are split across lines, for instance programs which are produced using a preprocessor. Choosing the option **–nolinebreak** will cause **Ftnchek** to skip over the end of line and also any leading space on the continuation line (from the continuation mark up to the first nonspace character). Default = yes, i.e. treat linebreaks as space. Default = no.

Note that in nolinebreak mode, if token pairs requiring intervening space (for instance, `GOTO 100`) are separated only by a linebreak, they will be rejoined.

Also, tokens requiring more than one character of lookahead for the resolution of ambiguities must not be split across lines. In particular, a complex constant may not be split across a line.

–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 (^) 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.

–novice This flag is intended to provide additional helpful output for beginners. At this time, the only extra message it provides is a comment that any 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). Default = yes.

In earlier versions of **Ftnchek** , this option could take on various numerical values, as a way of controlling various classes of warnings. These warnings are now controlled individually by their own flags. Novice level 1 is now handled by the **–array** flag; level 2 has been eliminated; level 3 is equivalent now to setting **–novice** to yes; level 4 is handled by the **–impure** flag.

–**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.

–**portability**

Ftnchek will give warnings for a variety of non-portable usages. These 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 violations of the Fortran 77 standard, which may also cause portability problems. To catch those, use the –**f77** option. Default = no.

–**pretty** Controls certain messages related to the appearance of the source code. These warn about things that might in some cases be deceptive to the reader. At present, the only warning that is controlled by this flag refers to comments that are interspersed among the continuation lines of a statement. Default = yes.

–**project**

Ftnchek will create a project file from each source file that is input while this flag is in effect. 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 = no.

A project file contains a summary of information from the source file, for use in checking agreement among `FUNCTION`, `SUBROUTINE`, and `COMMON` block 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.

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.

Ordinarily, project files should be created with the –**library** flag in effect. 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. Of course, this means that the calling hierarchy among routines defined within the file is lost. Normally the loss of this information is unimportant. If you wish to retain this information for some reason, you can create the project file 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 intermodule checks that it already did when the project file was created.

Because of the 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.

Naturally, when the **–project** flag is set, **Ftnchek** will not read project files as input.

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. 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.

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

# tell make how to create a .prj file from a .f file
.f.prj:
        ftnchek -project -noextern -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)
```

–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.

–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.

–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.

–truncation

Warn about possible truncation (or roundoff) errors. Most of these are related to integer arithmetic. The warnings enabled when this flag is in effect are: conversion of a complex or double

precision value to single precision; conversion of any real type to integer; use of the result of integer division where a real result seems intended (namely as an exponent, or if the quotient is later converted to real); division in an integer constant expression that yields a result of zero; exponentiation of an integer by a negative integer (which yields zero unless the base integer is 1 in magnitude); and use of a non-integer array subscript, DO index or DO loop bounds. Default=yes.

Note: warnings about truncating type conversions are given only when the conversion is done automatically, i.e. by an assignment statement. If intrinsic functions such as `INT` are used to perform the conversion, no warning is given.

−usage=*n*

Warn about unused or possible uninitialized variables. Default=3.

The meanings of the setting values are as follows:

0: no warnings.

1: warn if variables are (or may be) used before they are set.

2: warn if variables are declared or set but never used.

3: give both types of warnings.

Sometimes **Ftnchek** makes a mistake about these warnings. 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.

−verbose

This option is on by default. Turning it off 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. (Some of this output is turned back on by the **−list** and **−symtab** options.) Default = yes.

CHANGING THE DEFAULTS

Ftnchek includes a mechanism for changing the default values of all options by defining environment variables. 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 novice level) 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`. Of course, command-line options will override these defaults the same way as they override the built-in defaults.

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`

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 2.6 December 1992

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 converted to real
     17         END
     18

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.
```

```
Variables declared but never referenced in module COMPAV:
         J

Variables may be used before set in module COMPAV:
       SUM


     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.


Variables set but never used in module AVENUM:
       AVG
```

```
 0 syntax errors detected in file average.f
 1 warning issued in file average.f

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

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 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

Ftnchek will print out four main types of messages. They are portability warnings, other warnings, informational messages, and syntax errors. Portability warnings specify nonstandard usages that may not be accepted by other compilers. Other warning messages report potential errors that are not normally flagged by a compiler. Informational messages consist of warnings which may assist the user in the debugging of their Fortran program.

Syntax errors are violations of the Fortran language. The user should have already eliminated any that are flagged by the Fortran compiler.

Ftnchek does not detect all syntax errors. Generally, **Ftnchek** only does as much local syntactic error checking as is necessary in order for it to work properly.

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**. 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.**

Most error messages are self-explanatory. Those which need a brief explanation are listed below. Please note that any error messages which begin with "oops" refer to technical conditions and indicate bugs in **Ftnchek** or that its resources have been exceeded.

The following messages warn about possible portability problems, including common but nonstandard usages:

Nonstandard format item

Ftnchek will flag nonstandard items in a `FORMAT` statement which may not be compatible with other systems. Controlled by **–f77** option.

Characters past 72 columns

A statement has been read which has nonblank characters past column 72. Standard Fortran ignores all text in those columns, but many compilers do not. Thus the program may be treated differently by different compilers. Controlled by **–f77** option and **–columns** setting.

Warning: file contains tabs. May not be portable.

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 strings. Controlled by **–portability** option.

Nonstandard type usage in expression

The program contains an operation such as a logical operation between integers, which is not standard, and may not be acceptable to some compilers.

Common block 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 **–portability** option.

Common block has long data type following short data type

Some compilers require that if a `COMMON` block contains mixed data types, all long types (namely `DOUBLE PRECISION` and `COMPLEX`) must precede all short types (namely `INTEGER`, `REAL`, etc.) Controlled by **–portability** option.

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 controlled by the **–portability** option, since it affects **Ftnchek**'s analysis of the program.

Identifiers which are not unique in first six chars

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

The following messages warn of possible errors (bugs) that could cause incorrect operation of the program:

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** option.

Integer quotient expr used in exponent
Similarly, if the quotient of two integers is used as an exponent, it is quite likely that a real type division was intended. Controlled by –**truncation** option.

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** option.

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** option.

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 –**truncation** option.

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** option.

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.

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.

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. Given for –**usage** setting 1 or 3.

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. Given for –**usage** setting 1 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.

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 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 said

```
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.

Subprogram 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
```

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, i.e. its value will be 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. Given for **–usage** setting 1 or 3.

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. Given for **–usage** setting 1 or 3.

Common block NAME: varying length

A `COMMON` block declared in different subprograms has different numbers of variables in it in different declarations. This is not necessarily an error, but it may indicate that a variable is missing from the list. Note that according to the Fortran 77 standard, it is an error for named common blocks (but not blank common) to differ in length in declarations in different modules. Given for **–common** setting 2 or 3.

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.

The following messages refer to local syntax errors:

Syntax 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 the compiler does not report a syntax error at the same place, the most common explanations are: (1) use of a reserved word as an array or character variable (see Table 2 in the section entitled "Limitations and Extensions"), or (2) use of an extension to ANSI standard Fortran that is not recognized by **Ftnchek**.

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.).

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.

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.

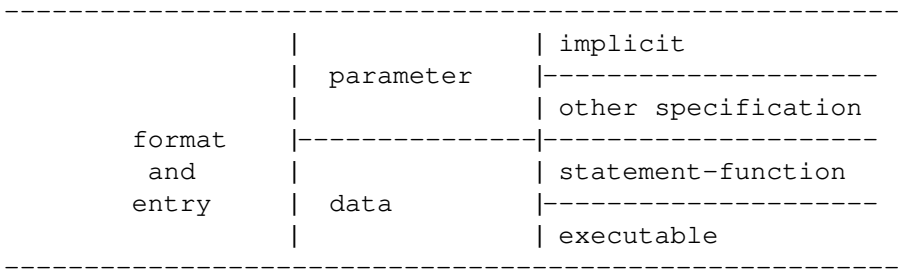

Table 1

The following messages are informational messages, which probably do not indicate bugs, but may indicate carelessness or oversights during modification of a program.

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** option.

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. Given for **–usage** setting 2 or 3.

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. Given for **–usage** setting 2 or 3.

Type has been implicitly defined

Ftnchek will flag 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 is printed only when the **–symtab** option is in effect. Alternatively, use the **–declare** flag if you want to get a list of all undeclared variables.

Possibly it is an array which was not declared

This message refers 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.

LIMITATIONS AND EXTENSIONS

Ftnchek accepts ANSI standard Fortran-77 programs with the following exceptions:

Restrictions:

Ftnchek is sensitive to blank spaces. This encourages the user to use good programming style. The rules are similar to Pascal or C where a blank space is required between identifiers or keywords and not allowed inside identifiers or keywords. Any keywords which occur in pairs may be written as either one or two words, e.g. `ELSE IF` or `ELSEIF`. Unlike Pascal and C, **Ftnchek** allows blanks inside numeric constants, except within the exponent part of E and D form numbers. Also, if the **–nolinebreak** option is selected, the end of line in continued statements is ignored.

Complex constants are subject to a special restriction: they may not be split across lines, even in **–nolinebreak** mode.

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

Some keywords and identifiers are partially reserved. See Table 2 for details.

The following keywords may be freely used as variables:

```
ASSIGN           BLOCK          BLOCKDATA      BYTE
CALL             CHARACTER      COMMON         COMPLEX
CONTINUE         DIMENSION      DO             DOUBLE
DOUBLEPRECISION  ELSE           END            ENDDO
ENDIF            ENTRY          EXTERNAL       FILE
FUNCTION         GO             IMPLICIT       INCLUDE
INTEGER          INTRINSIC      LOGICAL        NAMELIST
PAUSE            PRECISION      PROGRAM        REAL
SAVE             STOP           SUBROUTINE     THEN
TO
```

The following keywords may be used in scalar contexts only, for example, not as arrays or as character variables used in substring expressions.

```
ACCEPT           BACKSPACE      CLOSE          DATA
DOWHILE          ELSEIF         ENDFILE        EQUIVALENCE
FORMAT           GOTO           IF             INQUIRE
OPEN             PARAMETER      PRINT          READ
RETURN           REWIND         TYPE           WRITE
WHILE
```

Table 2

Extensions:

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.

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

Hollerith constants are permitted, in accordance with the ANSI Manual, appendix C. They should not be used in expressions, or confused with datatype `CHARACTER`.

Statements may be longer than 72 columns provided that the setting **−column** 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.

Variable names may contain underscores, which are treated the same as alphabetic letters. The VMS version of **Ftnchek** also allows dollar signs in variable names, but not as the initial character.

The `DO` ... `ENDDO` control structure is permitted. The syntax which is recognized is according to either of the following two forms:

```
DO [label [,]] var = expr , expr [, expr]
 ...
END DO
```

or

```
DO [label [,]] WHILE ( expr )
 ...
END DO
```

where square brackets indicate optional elements.

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

Statements may have any number of continuation lines. The standard allows a maximum of 19.

Inline comments, beginning with an exclamation mark, are permitted.

`NAMELIST` I/O is supported. The syntax is the same as in VAX/VMS or IBM 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.

Data types `INTEGER`, `REAL`, `COMPLEX`, and `LOGICAL` are allowed to have an optional length specification in type declarations. For instance, `REAL*8` means an 8-byte floating point data type. The `REAL*8` datatype is interpreted by **Ftnchek** as equivalent to `DOUBLE PRECISION`. **Ftnchek** ignores length specifications on all other types. The standard allows a length specification only for `CHARACTER` data.

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

```
INCLUDE 'filename'
```

When compiled for VMS, **Ftnchek** will assume a default extension of *.for* if no filename extension is given. Also for compatibility with VMS, the VMS version 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.

At this time, diagnostic output relating to items contained in include files is minimal. Only information about the location in the include file is given. There is no traceback giving the parent file(s), although usually this can be inferred from the context.

NEW FEATURES

Here are the changes from Version 2.5 to Version 2.6:

1. The following bugs in Version 2.5 were fixed: Overflow of large integer parameter values. Inline comment character mistaken in difficult contexts. Unnamed `BLOCK DATA` modules treated as main programs. `DATA` implied-do statements sometimes parsed incorrectly. Size of variable-dimensioned arrays sometimes calculated incorrectly. Documented bug number 1 in the previous documentation, which caused a used-before-set warning if a function modifies an argument, has been fixed. The warning is now controlled by the **–impure** option.
2. New features: Support for NAMELIST I/O. Directories to be searched for include files can be specified. All keyword pairs are now accepted in both split or single-word form. `PARAMETER` definitions may contain intrinsic functions. Cross-module checking within library files is now limited to modules in the call tree. `BYTE` data type is accepted (treated as INTEGER). VMS and MS-DOS versions no longer require command-line flags having the "/" prefix to be separated by space.
3. New command-line flags added and existing flags modified for better control of error and warning reporting. Affected flags are: **–array**, **–calltree**, **–help**, **–hollerith**, **–novice**, **–pretty**, **–pure**, **–trunc**, **–usage**, and the **–no** prefix to turn functions off. See documentation sections for full explanation of these flags.
4. New warnings: if array subscript or DO index is non-integer; if constant value of 0 results from integer division or exponentiation; and if data type of expression in logical or arithmetic `IF` statement is improper. The warning of "variable declared but not used" is now suppressed when the declaration is in an include file. A warning is now given under the **–f77** option if the standard limit of 19 continuation lines is exceeded.

Here are the changes from Version 2.4 to Version 2.5:

1. The name was changed from **Forchek** to **Ftnchek**, to avoid confusion with a similar program named **Forcheck** , developed earlier at Leiden University.
2. Some bugs were fixed: Version 2.4 incorrectly processed `DO` index variable names beginning with D or E. It did not support the `+kP` format descriptor. The VMS version failed to accept the `NOSPANBLOCKS`, `READONLY` or `SHARED` keywords in `OPEN` statements. Also, a couple of error messages were improved.

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. 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**.
3. A brief description of the bug.
4. If possible, a small sample program showing the bug.

The report should be sent to either of the following addresses:

```
MONIOT@FORDMULC.BITNET
moniot@mary.fordham.edu
```

Highest priority will be given to bugs which cause **Ftnchek** to crash. Bugs involving incorrect warnings or error messages may take longer to fix.

Certain problems that arise when checking large programs can be fixed by increasing the sizes of the data areas in **Ftnchek** 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 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: Length declarations of character variables are not correctly handled in `COMMON` block checking. Nonstandard length declarations of other data types, except `REAL*8`, are also not handled correctly.

   Prognosis: We hope to fix this soon, possibly in the next release.

CONCLUSION

Ftnchek was designed by Dr. Robert Moniot, professor at Fordham University, College at Lincoln Center. 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. Additional features will be added as time permits.

We would like to thank Markus Draxler of the University of Stuttgart, Greg Flint of Purdue University, Phil Sterne of Lawrence Livermore National Laboratory, and Warren J. Wiscombe of NASA Goddard for reporting some bugs in Versions 2.1 and 2.2. We also thank John Amor of the University of British Columbia, Daniel P. Giesy of NASA Langley Research Center, Hugh Nicholas of the Pittsburgh Supercomputing Center, Dan Severance of Yale University, and Larry Weissman of the University of Washington for suggesting some improvements. Nelson H. F. Beebe of the University of Utah kindly helped with the documentation, and pointed out several bugs in Version 2.3. Reg Clemens of the Air Force Phillips Lab in Albuquerque and Fritz Keinert of Iowa State University helped debug Version 2.4. We also thank Jack Dongarra for putting **Ftnchek** into the Netlib library of publicly available software.

For further information, you may contact Dr. Robert Moniot at either of the following network addresses:

```
MONIOT@FORDMULC.BITNET
moniot@mary.fordham.edu
```

This document is named *ftnchek.man*.
The **Ftnchek** program can be obtained by sending the message `send ftnchek from fortran`

to the Internet address: `netlib@ornl.gov`.

Installation requires a C compiler for your computer.