NAME

forchek − Fortran program checker

SYNOPSIS

forchek [−**[no]declare**] [−**[no]division**] [−**[no]extern**] [−**[no]f77**]
[−**[no]library**] [−**[no]linebreak**] [−**[no]list**] [−**[no]portability**]
[−**[no]project**] [−**[no]sixchar**] [−**[no]symtab**] [−**[no]usage**] [−**[no]verbose**]
[−**columns**=*num*] [−**common**=*num*] [−**novice**=*num*] [−**output**=*str*]

INTRODUCTION

Forchek (short for Fortran checker) is designed to detect certain errors in a Fortran program that a compiler usually does not. **Forchek** 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. **Forchek** 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 **Forchek**, the user should verify that the program compiles correctly.

This document first summarizes how to invoke **Forchek**. That section should be read before beginning to use **Forchek**. Later sections describe **Forchek**'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 **Forchek**.

INVOKING FORCHEK

Forchek is invoked through a command of the form:

```
$ forchek [-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 systems you may use either '/' or '−'.) The options are described at greater length in the next section.

Forchek 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 can be preceded by 'no' to turn it off: e.g. −**nousage** would turn off the warnings about variable usage. Only the first 3 characters of an option name (not counting the '−') need be provided. The switches which **Forchek** currently recognizes are:

−**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 violations of the Fortran 77 standard. Default = no.

−**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.

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

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

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

−symtab
Print out symbol table. Default = no.

−usage
Warn if variables not used, etc. Default = yes.

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

There are four settings:

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

−common=*n*
Level of strictness in checking *COMMON* blocks. Min is 0 (no checking). Max is 3 (must be identical). Default = 3.

−novice=*n*
Set novice level, which controls certain types of warnings. Min is 1 (tyro). Max is 5 (wizard). Default = 1.

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

When more than one option is used, they should be separated by a blank space. No blank spaces may be placed around the equals (=) in a setting. **Forchek "?"** 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, **Forchek** will first look for a project file with extension *.prj* and will use that if it exists. If not, then **Forchek** 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 **Forchek**, and it will process the modules in all files as if they were in a single file.

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

FORCHEK OPTIONS

This section provides a more detailed discussion of **Forchek** 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.

−columns=*n*
Set maximum line 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 line. According to the standard, all characters past column 72 are ignored. This setting does not affect the reporting of overlength lines under the **−f77** option. Max is 132. Default = 72.

−common=*n*

This setting varies the strictness of checking of *COMMON* blocks. Level 3 is the strictest: it requires that in each declaration of a given *COMMON* block, corresponding variables agree in data type and (if arrays) size and number of dimensions. Levels 1 and 2 require only that corresponding memory locations agree in data type. The difference between Levels 1 and 2 is that Level 2 warns if the blocks are not equal in total length, while Level 1 does not. Level 0 suppresses all checking. Default = 3.

−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 **Forchek** to report whether any subprograms invoked by the program are never defined, or are multiply defined. Ordinarily, if **Forchek** 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. **Forchek** does not report on the use of lowercase letters. Default=no.

−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, **Forchek** 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, **Forchek** 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 **Forchek** 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 **Forchek** 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.

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 **Forchek** 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, **Forchek** will still

print out any line containing an error, to aid the user in determining where the error occurred. Default = no.

−novice=*n*

This setting controls certain messages about conditions which are likely to be errors for novice programmers, but which are often intentional by more sophisticated programmers. Some of these warnings deal with cases in which **Forchek** suspects that what appears to be a function is intended to be an array, which the user forgot to declare in a *DIMENSION* statement. Since a function invocation and an array reference are identical in syntax, undeclared arrays are interpreted by the Fortran compiler and by **Forchek** as functions. Novice users are often confused by the messages which result. **Forchek** attempts to remedy this confusion. Default level = 1.

The novice levels are given below. The warning corresponding to each number will be suppressed if the novice level is set to greater than that value.

1. Warn if arrays passed as arguments to a subprogram do not match the corresponding dummy arguments in both number of dimensions and size. Exception: if the declared size of the dummy array is 1 or if it is dimensioned with a dummy variable, only the number of dimensions will be checked.
2. Warn the user if any argument of a subprogram appears to be a function. This warning is suppressed if the dummy argument is declared in an *EXTERNAL* statement. Novice programmers seldom pass functions as arguments of a subprogram, so it is more likely that such an argument was intended to be an array, but was not dimensioned.
3. If a function was invoked but never defined, advise the user that it may be an array which was not dimensioned. This warning is suppressed if the function is declared in an *EXTERNAL* or *INTRINSIC* statement in any module of the program. This warning is completely suppressed by the **−noextern** option.
4. Warn if a function has side effects: i.e. if it modifies any of its arguments, or modifies a variable in *COMMON*. Ideally, a function has no side effects, and acts only by computing a value based on its arguments, whereas a subroutine normally acts through side effects. Advanced programmers sometimes wish to combine the features of a subroutine and a function in a single module.

−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 **Forchek** 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

Forchek 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.

−project

Forchek 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 *forchek.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 **Forchek** 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 **Forchek** 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.

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. Thus project files contain only information 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.

Naturally, when the **−project** flag is set, **Forchek** 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:
        forchek -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)
        forchek $(PRJS)
```

−sixchar

One of the goals of the **Forchek** 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.

−usage

This switch is on by default. It causes **Forchek** to list all variables which may be used before they are initialized, or which are given a value but never subsequently used, or which are declared but never used. Sometimes **Forchek** makes a mistake about this. Usually it errs on the side of giving a warning where no problem exists, but in rare cases

it will fail to warn where the problem does exist. See the section on bugs for examples. If variables are equivalenced, the rule used by **Forchek** is that a reference to any variable implies the same reference to all variables it is equivalenced to. Default = yes.

−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

Forchek includes a mechanism for changing the default values of all options by defining environment variables. When **Forchek** starts up, it looks in its environment for any variables whose names are composed by prefixing the string "FORCHEK_" onto the uppercased version of the option name (the quote marks are not part of the 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" (again, the quotes are not part of the value). 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 **Forchek** print a source listing by default, set the environment variable "FORCHEK_LIST" to "1" or "YES" or anything other than "0" or "NO". The names "FORCHEK_LIS" (not the full option name) or "forchek_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: `$ FORCHEK_LIST=YES  $ export FORCHEK_LIST`

2. Unix, C shell: `% setenv FORCHEK_LIST YES`

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

4. MSDOS: `$ SET FORCHEK_LIST=YES`

AN EXAMPLE

The following simple Fortran program illustrates the messages given by **Forchek**. 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 **Forchek** when it is used to debug the above program:

```
$ forchek -list -symtab average

FORCHEK Version 2.4 August 1991

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 **Forchek**, the program contains variables which may be used before they are assigned an initial value, and variables which are not needed. **Forchek** 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, **Forchek** prints out a table containing identifiers from the local module and their corresponding datatype and number of dimensions. Finally, **Forchek** warns that the function is not used with the proper type of arguments.

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

INTERPRETING THE OUTPUT

Forchek 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 these by using the Fortran compiler. **Forchek** does not detect all syntax errors. Generally, **Forchek** only does as much syntactic error checking as is necessary in order for it to work properly.

If **Forchek** gives you a syntax error message when the compiler does not, it is probably because your program contains an extension to standard Fortran which is accepted by the compiler but not by **Forchek**. On a VAX/VMS system, you can use the compiler option `/STANDARD` to cause the compiler to accept only standard Fortran. On most Unix 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 **Forchek** or that its resources have been exceeded.

The following messages warn about portability or nonstandard usages:

Nonstandard format item
: **Forchek** will flag nonstandard items in a *FORMAT* statement which may not be compatible with other systems.

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.

Warning: file contains tabs. May not be portable.
: **Forchek** 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. **Forchek** does not give this message if tabs only occur within comments or strings.

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

Common block has long data type following short data type
: 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*. Some compilers additionally 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.*).

The following messages are warning messages:

Integer quotient expr converted to real

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 involving division is later converted to a real datatype, it may be that a real type division had been intended. Likewise, if it is used as an exponent, it is likely that a real type division was intended.

real truncated to intg
: **Forchek** 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 real variable, etc.

Continuation follows comment or blank line

Forchek issues this warning message to alert the user that a continuation of a statement is interspersed with comments, making it easy to overlook.

Possible division by zero

This message is printed out wherever division is done (except division by a constant), if the **−division** option was selected.

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.

Unknown intrinsic function

This message warns the user that a name declared in an *INTRINSIC* statement is unknown to **Forchek**. Probably it is a nonstandard intrinsic function, and so the program will not be portable. The function will be treated by **Forchek** as a user-defined function.

The following messages are syntax errors:

syntax error

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 **Forchek**.

No path to this statement

Forchek 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.

Forchek 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 **Forchek**, to prevent "cascades" of error messages.

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

Table 1

The following messages are informational messages:

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.

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.

Variables may be used before set

Similar to used before set except that **Forchek** is not able to determine its status with certainty. **Forchek** assumes a variable may be used before set if the first usage of the variable occurs prior in the program text to its assignment.

Variables set but never used

Forchek 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.

Type has been implicitly defined

Forchek 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.

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. This message is printed only if the −**sixchar** option was selected.

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. **Forchek** 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)
```

Forchek 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, **Forchek** must be able to determine whether it is a function or a subroutine. The rules used by **Forchek** 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
```

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.

Subprogram NAME: argument usage mismatch

Forchek 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. **Forchek** 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.

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.

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.

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), **Forchek** 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 **Forchek** 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.

LIMITATIONS AND EXTENSIONS

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

Restrictions:

Forchek 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. The following keywords which occur in pairs may be written as either one or two words: *DO WHILE* or *DOWHILE*, *ELSE IF* or *ELSEIF*, *END DO* or *ENDDO*, *END IF* or *ENDIF*, *GO TO* or *GOTO*. Unlike Pascal and C, **Forchek** 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           CALL            CHARACTER
COMMON          COMPLEX         CONTINUE        DIMENSION
DO              DOUBLE          ELSE            END
ENDDO           ENDIF           ENTRY           EXTERNAL
FUNCTION        GO              IMPLICIT        INCLUDE
INTEGER         INTRINSIC       LOGICAL         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 VAX version of **Forchek** 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 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.

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, **Forchek** 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 **Forchek** as equivalent to *DOUBLE PRECISION*. **Forchek** ignores length specifications on all other types. The standard allows a length specification only for *CHARACTER* data.

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

```
INCLUDE 'filename'
```

When compiled for VMS, **Forchek** 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 2.3 to Version 2.4:

1. Fixed bugs: the *SAVE* statement was incorrectly parsed, a *CALL* of a user function with the same name as an intrinsic function assumed by default to refer to the intrinsic function, and adjustable-size arrays passed as arguments were not correctly checked. Also, the usage of variables in some I/O control-list specifiers was handled incorrectly.
2. New options **−declare**, **−f77**, **−linebreak** and **−verbose**.
3. Allow embedded space in numeric constants.
4. Support *ACCEPT*, *IMPLICIT NONE* and *INCLUDE* statements.
5. Analyze *EQUIVALENCE* statements.
6. Common block checking levels 1 and 2.
7. *Important:* the project-file format has been changed. Project files created by Version 2.3 are not compatible with Version 2.4, and will need to be remade.

Here are the changes from Version 2.2 to Version 2.3:

1. Three bugs were fixed: Version 2.2 crashed if a real constant exceeding the magnitude limit was encountered; the computation of hash codes was not portable to 64-bit machines; and spurious used-before-set messages were generated by statement functions. We thank Greg

Flint of Purdue University and Warren J. Wiscombe of NASA Goddard for pointing some of these out.

2. Allow complex constants in expressions.
3. Allow *DO... ENDDO* structure
4. Allow *TYPE* statement.
5. Allow underscores in variable names.
6. Allow inline comments.
7. Provide project-file capability.
8. Suppress used-before-set messages for implied-do index (see Bugs section).

BUGS

Forchek 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 **Forchek** issues an error message where no error exists, but also if **Forchek** fails to issue a warning when it ought to. Note, however, that **Forchek** 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. **Forchek** 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 **Forchek** 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 **Forchek**'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 **Forchek**, 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 **Forchek** is running.
2. The version of **Forchek**.
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 **Forchek** to crash. Bugs involving incorrect warnings or error messages may take longer to fix.

The following is a list of known bugs.

1. Bug: Functions which modify their arguments may cause used-before-set warning. For example, *A* in the statement

```
X = FUN(A)
```

if *A* is not previously set. If *FUN* has the purpose of setting *A*, this situation will not be a bug. Generally, however, it is not considered good style for functions to modify their arguments, and so the warning may be to good purpose.

Prognosis: Probably will not be fixed anytime soon.

2. 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 **Forchek** 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 **Forchek** to report a spurious used-before-set warning for *J*. Since this report is usually in error and occurs fairly commonly, **Forchek** suppresses the warning for *J* altogether.

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

3. 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 **Forchek**.

CONCLUSION

Forchek 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 **Forchek** 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 *forchek.man*. The **Forchek** program can be obtained by sending the message `send forchek from fortran` to the Internet address: `netlib@ornl.gov` . Installation requires a C compiler for your computer.