Essential Fortran

Linda B. Merims
MIT Project Athena
Revision C
September 1, 1987

Table of Contents

1 Introduction

Essential Fortran shows you how to create a short sample Fortran program, compile it, and then run the compiled program. It is intended for students who must run Fortran programs as part of their coursework and who want to get started as quickly and simply as possible. ***Essential Fortran's*** emphasis is on following the example by rote; there is little explanation of what you are doing. The latter part of the document contains more advanced information on run-time debugging, linking, and file I/O. ***Essential Fortran*** does not teach you the Fortran language. For more information on this and any of the steps shown here, see the Athena documents mentioned in the text and in the section labelled "For More Information."

2 Antecedents

We assume that you know all of the following. If you are a student in a class, your instructor should provide you with some of this information. You probably specified your username and password yourself.

your username:	a 3-8 letter "name" that identifies you to the system
your password:	a short password to keep your username private
cluster door combination:	the numeric key to unlock the cluster door

We also assume that you know how to login to a Unix system and that you know the few extras involved in using a workstation. If you need to learn more about how to get started with Unix or how to use a workstation, see ***Athena Workstations***, and the ***Athena Reference Card***. These are available in the Athena clusters.

3 Creating the Fortran Program

Once you have logged in, your first step is to type your Fortran program into a **file**. (The assumption is that you have already written the program out longhand and you just need to type it into a computer file.)

To illustrate what it is like to create a Fortran program, compile and execute it, we will use the following small program that converts fahrenheit temperatures to centrigrade:

```
C
C   Convert Fahrenheit temperatures to Centigrade: C = 5/9 (F - 32)
C
C234567----------------------------------------------------------------|------

      real  ftemp, ctemp

      print *, 'Type in a Fahrenheit temperature, then press RETURN'
      read  *, ftemp

      ctemp = (5.0/9.0) * ( ftemp - 32.0 )

      print *, ftemp, 'Fahrenheit is', ctemp, 'Centrigrade'
      print *, 'Exiting.  Bye.'
      end
```

To type your program into a file, you call up the **emacs text editor** and give it a **filename** to use. You can name your files anything you like. However, all Fortran program files **MUST** be suffixed with a **.f** "file type." If you don't give it a .f type, whenever you try to compile the program it will always fail with the message, "ld:*filename*:Bad magic number."

The file will be named ftoc.f.

3.1 Starting Emacs

3.1.1 Emacs on a Workstation

First, you need to start *emacs* with a **&** as the last character on the command line:

workstation% **emacs ftoc.f &**

If you don't use the **&**, you won't be able to use your original window for anything until you leave emacs. After you hit the RETURN key, Unix will print a message which looks like this:

[2] 10759

Both numbers will be different each time; these are simply Unix's way of letting you know that your emacs has been started.

Once your emacs has started a small flickering box will appear near the mouse cursor. This means that emacs is ready for you to choose the size and location of the editing window.

To choose the location of your window, move the mouse cursor to where you want the upper-left-hand corner of your emacs window to be and press a mouse button.

- Press the left mouse button to select a small, 24-line window which resembles a timesharing emacs screen.
- Press the right mouse button to select a large emacs window.

Pressing the center button on a VAXstation mouse, or both buttons on an RT PC mouse, and dragging the mouse creates a variable sized window.

Once the emacs window appears, move the mouse cursor inside the emacs window. Emacs won't see anything that you type if the mouse cursor is outside the emacs window.

The emacs window looks like the one shown below. Everything above the line is a "blank page" that you can fill up with text. The line shows emacs status information. If emacs has any questions it will prompt you for answers at the bottom of the screen. If you chose to use a large window, the "blank page" area above the line will be much larger.

3.2 Fortran Format

With this and all Fortran programs, you need to be careful about which columns you put things in. If you violate the rules, the errors that result are usually copious, obscure, and hard to diagnose correctly. Notice below how the programmer uses a comment card to help her line things up. The screen is 80 columns wide (like an old data card).

This is the format to follow:

Columns	What it contains
1	If a C is present, for "comment," the rest of the line is ignored.
2-5	Fortran statement numbers.
6	If not blank, this card is a continuation of the previous card.
7-72	Program statements, indented as necessary.
73-80	**MUST** be left blank. Fortran ignores them.

3.3 Typing in Text

Type in the fahrenheit to centrigrade program as shown below. Be sure to move the mouse cursor into the emacs window.

To enter text	Just start typing. When you get to the end of the line, press the RETURN key, just as you would on an electric typewriter.
Blankspace	Use the SPACEBAR to make blanks. In *emacs*, there is a difference between a blank (made with the SPACEBAR) and nothing at all.
Blanklines	Make blank lines just as you would on a typewriter, press the RETURN key an extra time.
Typos	If you make a mistake while typing along, use the DELETE key at the upper right of the main keyboard (sometimes labeled <--) to backup and fix the mistake.

3.4 Fixing Text

To edit existing text (fix typos, add or delete words or lines, etc.), you move the **cursor** around the "page" until it points to where you want to make a change and then you enter a command.

Move the cursor	You move the cursor with the arrow keys at the right of the keyboard. Notice how the cursor "wraps around" when you go over the end of a line. Please note that on some terminals there is a <-- delete key and a <-- arrow key. They are different.
Inserting letters	Move the cursor to where you want the new characters. Type them. The rest of the text shifts over to accommodate them.
Inserting lines	Move the cursor to where you want the new line to be. Press RETURN. A new line will open up. You may have to move the cursor one spot to the beginning of the new line.
Deleting letters	Move to just *after* the character you want to remove. Press the DELETE key.
Deleting lines	You can delete a whole line by just typing enough DELETEs to devour it. Alternatively, move the cursor to the beginning of the line, hold down the CTRL key (lower left of the keyboard), and while still holding it down, press k. Press CTRL-k again to get rid of the space the line occupied.

3.5 Exit Emacs

To leave *emacs*, type **CTRL-x** followed by a **CTRL-c**. Hold down the **CTRL** key at the lower left of the keyboard while you press **x**. Release the **x** key and (while still holding down the **CTRL** key) press the **c** key. At the bottom of the screen you'll see:

Save file /mit/user/ftoc.f? (y or n)

Nothing you have done is permanent until you type **y** for yes to this question.

If you reply **n** for no, emacs will ask you another question:

1 modified buffer exists, do you really want to exit? (yes or no)

A **yes** answer to this question will cause all your changes to be thrown away.

Answer **no**, and emacs will neither save your file nor will it exit. It will wait for you to enter more text, or to type another emacs command.

For more information on emacs, see the Athena documents *Essential Emacs* and its accompanying *Emacs Quick Reference* card, available in most Athena terminal clusters.

4 Compiling the Program

The next step is to use a **compiler** to translate the Fortran ftoc.f **source file** that you can read into the **binary file** form that the machine can understand. This is called "compiling a program." The Unix Fortran compiler is called *f77*. It usually takes several tries--perhaps many--to successfully compile a program because the compiler will find mistakes in the program. Here is a first try at compiling the file ftoc.f:

```
yourhost%  f77 ftoc.f                                  <--run f77 on the ftoc.f file
ftoc.f:                                                <--compiler starts sending messages of its progress
  MAIN:
Error on line 9 of ftoc.f: bad format descriptor       <--compiler finds error


Error.  No assembly.                                   <--compiler finishes without making a binary file
yourhost%
```

The *f77* compiler found an error in line 9 of the program, printed a message, "bad format descriptor," and then quit. The "Error. No assembly." message means that *f77* did not bother to make a machine-readable binary file because it wouldn't have worked.

You should now look at line 9 of your source program in ftoc.f with the *emacs* text editor to see what could be wrong.

5 Debugging

Whenever you debug a program there are two main questions to answer: ***where*** did the error occur and ***what*** was the mistake.

Fortran *f77* compiler error messages tell you the line number where it found the error. But how, when there is more than just a few lines, do you find this line in the original file, short of counting them with your finger?

5.1 Emacs Editor

You can get the *emacs* editor to go directly to a particular line if you know its line number. Call up emacs with the file ftoc.f as you did when you created your file. Don't forget the **&** at the end of the emacs command. As always when you start *emacs* there will be a short pause, and then ftoc.f is displayed.

Now, tell emacs to go to line number 9 as follows:

1. Press the **ESC** key. You may have to hunt for it. It is on the row of buttons across the top. It is the F11 key or the key marked ESC.
2. Press **x**. Emacs will prompt M-x at the bottom of the screen.
3. Now type out: **goto-line**, and press RETURN. Emacs will then ask you *Goto line:* and you should type the number **9** as your answer, and press the RETURN key. Emacs will move the cursor to the first character of line number 9.

Your screen will look like this:

The error message read, "Error on line 9 of ftoc.f: bad format descriptor." Line 9 is the **read** statement. You must now puzzle out what the error message means. "Bad format descriptor" means that *f77* doesn't like something about the format part of the read statement. The format part of a read statement is nearly always the next thing on the line after the word "read." It tells Fortran what the output should look like (how it should be spaced, where to put the decimal point, etc.) In this case, the * ("use your default formatting rules") was left out by mistake. Fortran is having trouble figuring how what an "ftemp" format should be, not realizing it is a variable, not a format.

To fix this error, you must type the "*, " in with *emacs*. Move the cursor with the arrow keys to the "f" of "ftemp" and type the "*, ". To exit *emacs*, type CTRL-x CTRL-c.

5.2 Hardcopy Output

It is often much easier to debug a program off a paper copy; you see more of the program at once. To produce a line-numbered paper copy of your source program, type:

host% **cat -n** ***filename.f*** **| lpr**

The "cat -n" lists the file out line-numbered. The "lpr" sends it to the default printer for your cluster. Don't forget the **|** or it won't work. You won't see anything on the terminal screen. Look for your username on the front page of the output. When you know the problem, fix it with *emacs*. See *Printing from an Athena Workstation* for more information on *lpr*.

5.3 Another Technique

Because emacs opens a new window on your workstation, you are able to use the original login window that you typed the *emacs* command in for other tasks. You can use the *cat -n* command described above to display your line-numbered Fortran program in this window. To have your program displayed in your Unix window one screen at a time, move the mouse cursor to this window and type:

host% **cat -n** ***filename.f*** **| more**

If your program won't all fit on one screen, *more* will stop when the screen fills up. Hit the space bar to see the next screen of your program. Be sure to move the mouse cursor back to the emacs window when you are finished reviewing your program.

5.4 Rules of Thumb

It takes practice and experience to learn how to debug. Most errors are not so straightforward. "Appendix A" is an alphabetic list of most of *f77*'s compiler error messages along with a short explanation of what specific problem causes the error. After a while, you will become quite adept at interpreting error messages. You will be able to just glance at most messages and see the real mistake. Here are a few guidelines on debugging:

- Accept that *there really is something wrong* with your program. Don't insist that the compiler is making a mistake.
- Work from the top down. Often later error messages are just secondary effects of earlier errors. Fix the first error, and the rest may disappear.
- Study the error message. Look for a keyword that rings a bell in your head to isolate which piece of the statement may be causing the trouble. If none of the words sound familiar, look them up in your Fortran book's index. In the example above, the keyword was "format." Compare what you have to examples in your Fortran book. If necessary, go character-by-character. A single comma in the wrong place can be fatal.
- Like a doctor diagnosing a patient's symptoms, be a little open-minded and skeptical about what the compiler tells you. A patient who has been playing tennis may come in asking for codeine for a sore arm--when they may actually be experiencing angina. Compilers are like that, what they complain about is not always what's really wrong, just sort-of.

  Error messages only tell you how the real problem manifested itself as a symptom. If you get error messages complaining about bad arrays and you don't *have* any arrays, it means that you've got something that, because of an error, *looks* like an array.

  Error messages only tell you when the compiler *first noticed* something was wrong. It may be that there is nothing wrong with the statement itself, but there could be something wrong with the statement immediately before it, or even earlier. This may just be the first place that the consequences of the mistake show up.

6 A Successful Compilation

To compile the ftoc.f program again:

```
yourhost%  f77 ftoc.f                          <--compile again
ftoc.f
  MAIN:
host%                                          <--no news is good news.  no compile errors found
```

When there are no serious errors, only warnings, the *f77* compiler produces a new output file containing the compiled version of the ftoc program. This **binary file** is what you run to actually execute the ftoc program. You may also hear binary files referred to as "executable images," and, occasionally, "load modules." The binary file will, by default, be named **a.out**.

You can get *f77* to name the binary file something different. You will have to begin doing this when you have more than one program file because whenever you *f77* a file it will overwrite the existing a.out. When you compile the program, add the **-o *filename*** option. This tells *f77* to call the output file *filename* instead of a.out. For example:

yourhost% **f77 -o ftoc ftoc.f**

will produce a binary called just ftoc. (This follows Unix naming conventions.) Notice that the "-o ftoc" comes *before* the ftoc.f file.

To see what files you have, use the Unix ***ls*** command (for "list"):

yourhost% **ls**
a.out ftoc.f ftoc.o

(The ftoc.o file is an intermediate version of your program that *f77* produces called an **object** file. You can ignore it for now.)

The Athena document *Athena Workstation* explains more about Unix file maintenance commands.

7 Executing the Program

After a "clean compile," you are ready to actually execute your program and convert some fahrenheit temperatures to centigrade. To execute any program, you type the name of its binary file--in this case a.out--just as though it were another Unix command:

```
yourhost%  a.out                                        <--type: a.out and press RETURN
Type in a Fahrenheit temperature, then press RETURN     <--program prints instructions
32                                                      <--you type in a number
32.0000 Fahrenheit is 0.0000 Centrigrade                <--program prints results
Exiting. Bye.                                           <--and exits
yourhost%
```

Things may not go this smoothly. Even though it "compiled cleanly," the program may still have so-called **run-time** errors.

8 Run-Time Debugging

Run-time errors are not errors in syntax, but errors in logic that the compiler cannot check. They only show up when you try to execute the program. If you are *un*lucky, they do not show up at all. Run-time errors are hard to diagnose. There are very few run-time error messages, so it is hard to tell *what* went wrong; and the Unix system does not tell you in which statement an error occurred, so it is also hard to tell *where* it went wrong. "Appendix B" lists such run-time error messages as there are, and we will show you one way to find out the line number where an error occurred.

Run-time debugging is a large topic. Run-time errors manifest themselves in myriad ways. We cover only the most obvious problems here.

8.1 Typical Problems

When you run a program, you usually see one of four things:

- Nothing happens for a *long* time. You don't get back a system prompt indicating that execution has finished, you don't get anything. This usually means one of two things is happening:

    1. You have a **read** statement that is waiting for you to type something and press RETURN. You didn't precede it with a **write** statement to tell you what to type.
    2. Your program is infinite looping. Kill it with a CTRL-C. You can find out where problems are by using the *dbx* debugger and the step-through technique shown below.

- Your program starts spilling copious output at you and won't stop. The output looks repetitious. This usually means your program is infinite looping. Kill it with a CTRL-C. Notice which **write** or **print** lines keep repeating as they show indirectly which loop's control variable or which **if-then-else** has gone awry.

- Nothing really bad happens. There are no error messages, you just don't get the results you expect. The only way to solve this is to trace the execution of the program: either by adding print statements, or by using the *dbx* symbolic debugger described below.
- When you run the program, you get a message which ends in the phrase "*core dumped.*" Your program has encountered an error so severe that the Unix system has terminated its execution. In the vernacular, your program has "bombed" or "died." The "core dumped" means specifically that Unix has created a file called ***core*** in the current directory that is a hex core dump of your program when it died. Experienced programmers can examine core dumps to find out what went wrong with a program, but it is not an easy thing to do. The main thing to realize is that the *core* file takes up a great deal of file space. You will want to delete it unless you have a purpose in mind for it.

The first two types of run-time errors are easier to find because the incorrect program output gives clues as to what went wrong and where it happened. The last two groups of run-time errors are more difficult to find since the program output only tells you that something went wrong. There is no message telling you *where* the problem is. Using a program called a ***debugger*** can help you find out more about where runtime errors are happening.

8.2 Using a Debugger

Here is a Fortran program that has several mistakes that are going to cause runtime errors:

```
C                                          2
C   Program to solve quadratic equation:  ax  + bx + c
C
C               ---------
C          +   / 2
C       -b - \/ b  - 4ac
C   x = -----------------
C              2a
C
C234567*********************************************************************
C Declarations

      real  a, b, c, x1, x2, underradical

C Ask for initial values from user, then read what he typed

      print *, 'Type in a b and c separated by blanks, then hit RETURN'
      read  *, a, b, c

C Now go ahead and compute values

      underradical = ( b**2 ) - ( 4 * a * c)
      x1 = ( -(b) - (sqrt(underradical)) )  /  ( 2 * a )
      x2 = ( -(b) + (sqrt(underradical)) )  /  ( 2 * a )

C Print results

      print *, 'The two roots are:', x1, x2

C And leave

 200  print *, 'Exiting.  Bye.'
      end
```

This program calculates the roots of a quadratic equation. You may be able to see the problem immediately. In trivial programs such as this, that is often true. But you will need the techniques shown below when you have a program with dozens of statements.

The program compiled cleanly:

```
host%  f77  quad.f
quad.f:
  MAIN:
host%
```

When executed on a VAXstation, the following occurred:

```
host%  a.out
 Type in a b and c separated by blanks, then hit RETURN
0 5 7                                  <--user replies to query
*** Arithmetic Exception: Floating divide by zero
Illegal instruction (core dumped)      <--program dies
host%
```

We know what type of problem to look for (a division by zero), but we don't know exactly which statement is causing the problem.

When run on RT PC workstations, the problem shows up differently:

```
rthost%   a.out
  Type in a b and c separated by blanks, then hit RETURN
0 5 7
  The two roots are:   -INF   NAN()             <- program produces strange
  Exiting, Bye!                                 <- output, but no error message
rthost%
```

The RT PC example doesn't generate any error message, but it produces a result that wasn't expected.

8.3 Where Is the Problem?

To find out which statement caused the run time error we can use a debugger called *dbx*. *dbx* is a powerful tool which can be used to find many run time errors. There are different debugging strategies to find different kinds of run time errors. Two of these strategies are shown below.

8.3.1 Finding the Statement Where a Program Core Dumps

To find out which statement caused the core dump in the VAXstation example:

1. Recompile the program, but use the **-g** option. This tells the compiler to produce debugging information. You *must* add the -g option.

   ```
   host%   f77 -g quad.f
   ```

2. Start up the ***dbx*** debugger program on the *binary file*.

   ```
   host%  dbx a.out                          <--start dbx on a.out
   dbx version 3.24 of 6/14/87 22:31 (paris)
   type 'help' for help.                     <--dbx announces itself
   reading symbolic information ...             ending with the (dbx) prompt
   (dbx)                                     <--you now type dbx debug commands
   ```

3. Tell *dbx* to start running the program, and respond *exactly* as before:

   ```
   (dbx)  run                                <--run the program
    type in a b and c separated by blanks, then hit return
   0 5 7                                     <--reply exactly as before

   floating point exception in main at line 23
     23     x1 = ( -(b) - (sqrt(underradical)) ) / ( 2 * a )
   (dbx)                                     <--aha! the offending line
   ```

   The error message is more obscure than before, but the line that causes the problem prints. By looking at the divisor, we can deduce that *a* must be zero. Well, yes, of course, that's what we said. Clearly, the problem with *quad.f* is that we need to test for the case where *a* is zero (figure is not a parabola, but a line) and do something special about it. We could also tell *dbx* to print the value of variables:

```
(dbx) print a
0.0
```

4. When you have found your error, be sure to leave *dbx* with the *quit* command:

   ```
   (dbx) quit                                  <--leave dbx with quit command
   ```

8.3.2 Stepping Through a Program

In the RT PC example, you will need to proceed a little differently, since you know very little about the error. The program doesn't stop and there is no message saying what happened or where it happened. To find the problem here, you will want to "slow down" the program's execution, and look at the values of each variable after each statement is executed. Using this technique to find your runtime errors is like watching a movie one frame at a time instead of at full speed. While you don't get to enjoy the movie as it was intended, you can look at all the details you would otherwise miss.

The only clues you can work with are the values output in the example: *-INF* and *NAN()*. The value *-INF* is used to represent an infinite number; these values are generated to replace values that are too large (overflow) or too small (underflow) to be stored in the computer. Also, *INF* is the value returned by calculations such as division by zero. The *NAN* (which stands for **N**ot **A** **N**umber) is output as the result of a calculation performed with *INF* as one of its terms. The *INF* and *NAN* values indicate errors in the statements that compute values.

To find this type of error:

1. Compile your program with the **-g** option; this option tells *f77* to include debugging information in the binary file *a.out* that it creates:

   ```
   rthost% f77 -g quad.f
   quad.f:
     MAIN:
   rthost%
   ```

2. Create a line-numbered listing of your program:

   ```
   rthost% cat -n quad.f | lpr
   ```

   This line-numbered listing will help you to see which statement *dbx* is currently looking at, and what has happened up to that point. You will need this listing as you step through the program. If you are using a workstation, you may want to display the line-numbered listing in your login window; use the procedure described earlier, called "Another Technique," to display the line-numbered program in your login window.

3. We're now going to start *dbx* and step through the program one statement at a time, and examine the values in the variables until the *INF* or *NAN* show up. The process is rhythmic--execute a statement, look at the variables, execute the next statement, look at the variables, etc. There is one oddity to what you'll see below. When *dbx* executes a statement, it doesn't then print out *that* statement; instead it prints out the next statement it would execute. When *dbx* executes 20, it prints statement 21. Remember that the action is really one statement back from the one that you see.

- Run the debugger on the *binary file*:

```
rthost%  dbx a.out                               <- start dbx on a.out
dbx version of 1986-03-16 5:24 (garp.mit.edu)
Type 'help' for help.                            <- dbx announces itself
reading symbolic information ...                   ending with the (dbx) prompt
(dbx)                                            <- now type dbx debug commands
```

- After *dbx* loads the binary file, it will prompt you with *(dbx)* and wait for you to type a command. Find the first *executable* statement in the program; executable statements are statements which **do** things. Comment lines, declaration statements, and blank lines aren't executable, but statements which read and write information, or do computations *are* executable statements. Since we know that the error happened as a result of some bad arithmetic, we should slow down the program execution where the calculations begin. The first executable statement which we are interested in is line number 22, since this is where the first calculation happens. Tell *dbx* to stop just *before* executing line 22:

```
(dbx)  stop at 22                                <- stop before the first calculation
(1) stop at 22
```

- Now run the program as you did before:

```
(dbx)  run                                       <- now run the program
```

- When dbx stops and gives you the *(dbx)* prompt, you can look at the values of the variables with the dbx *print* command:

```
stopped in MAIN at line 22
   22         underradical = (b**2) - ( 4 * a * c )
(dbx)  print a, b, c
0 5 7
(dbx)
```

- When you are done looking at the variables, you can tell dbx to execute this statement and stop before executing the next statement:

```
(dbx)  step                                      <- execute line 22
```

- Repeat the last two steps until you find where your error is.

Here is an example of a *dbx* session which uses this strategy:

```
rthost%  dbx a.out
dbx version of 1986-03-16 5:24 (garp.mit.edu).
type 'help' for help.
reading symbolic information ...
(dbx)  stop at 22                                <- tell dbx to stop just before the first computation
(1) stop at 22
(dbx)  run                                       <- now run the program
  Type in a b and c separated by blanks, then hit RETURN
0 5 7                                            <- answer just as you did before
stopped in MAIN at line 22                       <- dbx has executed line 18 and stopped
   22         underradical = (b**2) - (4 * a * c )
(dbx)  print a, b, c                             <- look at the values for a, b, and c
0 5 7
(dbx)  step                                      <- now run line 22 and stop before line 23
```

```
stopped in MAIN at line 23
  23          x1 = ( -(b) - (sqrt(underradical)) ) / ( 2 * a)
(dbx)  print underradical                          <- now that underradical has a value you can
25                                                    examine it
(dbx)  step                                        <- execute line 23 now
stopped in MAIN at line 24
  24          x2 = ( -(b) + (sqrt(underradical)) ) / ( 2 * a)
(dbx)  print x1
-INF                                               <- Aha!  Here's one of the errors
(dbx)  print a
0                                                  <-  (2 * a) = 0; this causes a "divide by zero" error
(dbx)  quit                                        <- we've found the error, so quit
rthost%
```

To fix the error you need to add statements to check *a* to see if its value is zero. The modified *quad.f* would look like this:

```
C                                        2
C   Program to solve quadratic equation:  ax  + bx + c
C
C               ---------
C          +   / 2
C       -b - \/ b  - 4ac
C   x = -----------------
C              2a
C
C234567*********************************************************************
C Declarations

      real  a, b, c,  x2, underradical

C Ask for initial values from user, then read what he typed

      print *, 'Type in a b and c separated by blanks, then hit RETURN'
      read  *, a, b, c

C Check if a=0  (line, not parabola)

      if ( a .EQ. 0) then                                     <- here's the fix!
         print *, 'a is 0, this is a line'
         go to 200
      endif

C Now go ahead and compute values

      underradical = (b**2) - (4 * a * c)
      x1 = ( -(b) - (sqrt(underradical)) )  /  ( 2 * a )
      x2 = ( -(b) + (sqrt(underradical)) )  /  ( 2 * a )

C Print results

      print *, 'The two roots are:', x1, x2

C And leave

200   print *, 'Exiting.  Bye.'
      end
```

There is another run-time bug hiding in this program. The program does not check to see if the value under the radical is negative, leading to an attempt to take the square root of a negative number, producing an irrational result. On some computer systems, the built-in *sqrt* function would check for this case and flag an error. On the Unix system, the *sqrt* function just returns 0. Without an explicit, programmer-supplied check, this program will quietly produce incorrect results.

Type *man dbx* for more information on the *dbx* debugger. The Athena document *More C* will also be available in the Athena terminal clusters; this document has an extensive description of the *dbx* debugger.

9 Linking to Libraries

You don't have to write Fortran code "from scratch" for everything you want to do. Any respectable system comes with an assortment of libraries that you can call from Fortran to do anything from take a square root to compute an eigenvalue to draw an x-y plot from an array of data. There are many such libraries on Athena. Some come with the Unix system, others Athena added as extras, such as the NAG library of mathematical routines and the Penplot graphics package. [1]Most of the "public" libraries are kept in the directories */lib*, */usr/lib*, or */usr/athena/lib*. You can construct your own private libraries with the *ar* utility. Many courses have done this.

Consult your Fortran text for the general rules on how to call functions and subroutines within a program. The important things to get right are the name of the function/subroutine, and the number and type of its arguments. *f77* and the run-time system do not check that you have called a function with the proper number of arguments or that the types (integer, real, etc.) match. This frequently causes obscure run-time bugs.

To use a library, you must link to it at compile time, as follows:

f77 *options sourcefile.f* **-l*libname***

The **-l** option specifies the library to link to. All -l options must come at the *end* of the compile command, after all filenames. There can be *no blanks* between the -l and the library name. You can specify more than one library; they will be linked in the order given.

The -l option, by default, searches the */lib, /usr/lib*, and */usr/athena/lib* directories for files of the form: **lib*name*.a**. For example, -lnag will look in the three library directories until it finds a file named *libnag.a*.

9.1 Linking to the NAG Library

To link to the NAG library:

host% **f77** ***options sourcefile.f*** **-lnag**

---

[1]As of this printing, none of these extra libraries are available on the RT PC workstations. Only the libraries supplied with Unix exist for the RT. If you will need to use these extra libraries you should plan on using a VAXstation.

9.2 Linking to the Penplot Library

To link to Penplot:

host% **f77** ***options sourcefile.f*** **-lpenplot**

9.3 Linking to Private Libraries

You can link to private libraries, such as those created for a class, provided that you have read permission on the file. To link to a private library, just give the complete filename of the file that contains the library. Do not use -l. For example:

host% **f77 mysource.f /mit/user/lib/draw.a**

would link *mysource.f* to the *draw.a* library under the user's library subdirectory.

9.4 Linking Errors: "Undefined" and "Permission Denied"

-l is actually an option to a separate linking program, ***ld***, not to the *f77* compiler. *f77* always calls *ld* as its last step (unless you specify -c for "compile only"). When *f77* calls *ld*, it tells it which libraries to search. *f77* always has *ld* search: */lib/libc.a* (Unix Programmer's Manual sections 3, 3S, 3N, 3X, and 3C), */usr/lib/libm.a* (the math library, manual section 3M), */usr/lib/libU77.a, /usr/lib/libF77.a*, and */usr/lib/libI77.a* (the Fortran libraries, manual section 3F). You should realize that, if you ever use *ld* separately, without *f77*, *ld* will *not* look in these files automatically; many things will become undefined. If you have doubts about which libraries are being linked in, use the **-t** (for "trace") option on the compile command.

If *ld* finds linking errors ("Undefined:" error messages), it will still produce a binary file named *a.out* (or whatever you specified), but it will set its file protection to "no owner execute." If you try to run such a file anyway, you will get a "Permission denied." error message. Verify that this is the problem with:

```
ls -l a.out
-rw-------  1  you   37888 Sep 1 16:15 a.out*
   |
   | No "x" for owner execute
```

10 Simple File I/O

By default, Fortran reads its input from the keyboard, and writes its output and errors to the screen. More precisely, Unit 0 (errors) is assigned to the Unix data stream *stderr*, Unit 5 (input) is assigned to Unix data stream *stdin*, and Unix 6 (output) is assigned to the Unix stream *stdout*. This means that you can easily redirect your input and output away from the keyboard and screen and have it come to and from files using the Unix I/O redirection mechanisms.

If a compiled program *myfile* contained the two statements:

```
read  *,a
write *,a
```

and was subsequently executed as follows:

```
host% myprogram
```

it would read its input data from the keyboard and write it to the screen.

If it were invoked as follows:

```
host%  myprogram  <testdata   >results
```

it would read from the file *testdata* in the current directory and write to a file *results*, also in the current directory. Much more elaborate I/O operations to files are possible. See the document "Introduction to the f77 I/O Library," part of the Athena *Fortran* manual for details.

11 For More Information

If you plan to do much Fortran programming, you will need to buy a good book on Fortran 77, both to teach yourself the language, and to use as a reference manual for syntax questions. There are dozens of Fortran 77 books. We recommend *Fortran 77 for Humans*, Second Edition, by Page, Didday and Alpert, West Publishing Co., ISBN 0-314-69672-5. [2] Note in particular the "Quick Reference Index" on its last page.

The IS Publications Office in 11-209 (M-F, 10 am to 4 pm) sells this book. It is used as a text for 2.10, and so the Coop usually has it as well. Reference copies are available in each cluster in the manual rack.

To learn more about the *f77* compiler, there is:

- Type: *man f77* -- this will show you the other options on the *f77* command.
- The Athena *Fortran* manual contains two documents: "A Portable Fortran 77 Compiler" and "An Introduction to the f77 I/O Library." You should get these if you will be doing serious Fortran work. They tell you things such as this compiler's extensions to the Fortran 77 standard, and details of I/O to files. Reference copies are in each terminal cluster manual rack. You can buy the manual at the IS Publications Office.

Very specific information on libraries is found in the Unix Programmer's Manual, section 3. This is sold as *The Unix Programmer's Reference Manual* through IS Publications. Reference copies are in the racks. The easiest way, though, is to examine the material online. Type: *man 3 intro* for the introduction to libraries. *man ar* describes sketchily how to make libraries. *man nm* describes how to examine their contents. Penplot and NAG documentation is available in the cluster reference racks. The NAG manual is a large green, 6-volume set. The small Penplot manual is also sold at IS Publications.

The *dbx* debugger is described on its man page; type *man dbx*. It is also described in the Athena *More C* document, which will be available in the cluster reference racks and at the IS Publications Office.

---

[2] Be careful not to get *Fortran for Humans* by the same author; it is for a very different, outdated version of Fortran

12 Appendix A: Compile Errors

These appendices showing f77 error messages were originally compiled for subject 2.10 by John Kyratzoglou and Dan Ottenheimer

ASSIGNMENT TO AN UNDIMENSIONED ARRAY

An assumed array name was used, where the array name was not declared in a declaration statement.

BAD ARGUMENT TYPE TO INTRINSIC FUNCTION

A function reference was made, using an intrinsic function name, but the argument list does not agree in order, number or type with the intrinsic function requirements.

BAD MAGIC NUMBER

For a FORTRAN file the extension must be **.f**. For a C file the extension must be **.c**.

DECLARATION ERROR DECLARATION AMONG EXECUTABLES

The order of the statements was not as specified in the FORTRAN Language Reference Manual. The statement found out of order is ignored.

```
Example :
        N = 10
        REAL X(N)
```

Here an executable statement precedes a declaration statement.

DECLARATION ERROR DECLARATION: INCOMPATIBLE TYPE DECLARATION

The same variable appears in more than one but different data type declaration statements. As an example,

```
REAL X(5)
INTEGER X(5)
```

DECLARATION ERROR FOR AN ENTITY, USED AS VARIABLE

An entity has been used incorrectly; for example, the name of a subprogram was used where an arithmetic expression is required.

DECLARATION ERROR FOR ENTITY: INCONSISTENT EQUIVALENCE

EQUIVALENCE statements specify inconsistent relationships between variables or array elements. As an example,

```
EQUIVALENCE (A(1), A(2))
```

DECLARATION ERROR LENGTH MUST BE A POSITIVE INTEGER CONSTANT

An unacceptable length has been specified in a data type declaration. This statement is not allowed.

```
Examples:
        INTEGER NUM*3
        CHARACTER*  NAME
```

DECLARATION ERROR FOR VARIABLE, IMPROPERLY DIMENSIONED ARRAY

A reference was made to an adjustable array in a context where such a reference is not allowed. Example:

```
parameter(nc=20, nchar=3)
real x(n)
```

The variable n specified giving the array size must be either initialized in the parameter statement or must be subprogram dummy argument or appear in common.

DO LOOP OR BLOCK IF NOT CLOSED

The terminal statement of a DO loop or the END IF statement of an IF block was not found.

EXECUTION ERROR ENDIF OUT OF PLACE

Invalid control structure using ELSEIF, ELSE, or ENDIF. The order of the above statements is incorrect.

EXECUTION ERROR LABEL ALREADY DEFINED

The same label appears on more than one statement.

EXECUTION ERROR, UNCLASSIFIABLE STATEMENT

Compiler cannot figure out what kind of statement it is. Sometimes caused by misspelled keywords or unbalanced quotes that create nonsense statement fragments.

EXECUTION ERROR, UNLABELED FORMAT STATEMENT

A required label in a format statement was ommitted.

FLOATING EXCEPTION DURING CONSTANT EVALUATION

The specified value of a constant is too large or too small to be represented.

IMPOSSIBLE CONVERSION

An expression that must be arithmetic(INTEGER, REAL, LOGICAL) was of type CHARACTER.

An expression that must be of type CHARACTER was another data type.

A variable is not declared properly.

IMPOSSIBLE STORAGE CLASS 8 IN ROUTINE

Often caused by attempt to use a non-array variable as an array. Often paired with "DECLARATION ERROR" message.

INVALID CONTROL ACCESS FOR STATEMENT

The ACCESS= keyword specified in an I/O statement was the one assumed when the file was created.

INVALID CONTROL END FOR STATEMENT

The END=keyword was used illegally in a WRITE or direct access READ statement.

LABEL ALREADY THAT OF A FORMAT

Labels of executable statements have been confused with labels of FORMAT statements or with labels of non executable statements.

LOGICAL DATUM ASSIGNED TO A REAL VARIABLE

In a data statement a logical constant was assigned to a real variable. Example:

```
data bzro /.true./
```

LD:*filename*: CANNOT OPEN

File type of *filename* was not specified during the creation of a FORTRAN or C file. As a result during the execution of the shell command

```
% f77 for.
or
% f77 for
```

the file could not be opened and compiled.

LOCAL VARIABLE "X" NEVER USED

A particular variable was referenced but never used in the program.

MISSING END STATEMENT

An END statement was missing at the end of the last input file.

MISSING STATEMENT NUMBER

Reference has been made to statement label that is not defined in the program unit.

MORE DATA VALUES THAN DATA ITEMS

The number of variables or array elements to be initialized must match the number of constants specified in data initialization.

NAME " name " TRUNCATED TO 16

" name " was longer than 16 characters and has been truncated.

NONDIGIT IN STATEMENT NUMBER FIELD

An improperly formed statement label(namely one containing letters) has been detected in columns 1 to 5 of an initial line. The statement label was ignored.

NONLOGICAL EXPRESSION IN IF STATEMENT

An expression that must be of type LOGICAL was another data type.

NONLOGICAL OPERANT OF LOGICAL OPERATOR

An invalid operation was specified as .AND. of two real variables.

NUMERIC DATUM ASSIGNED TO A CHARACTER VARIABLE

Character data with a length greater than one was initialize with a numeric value in a DATA statement. As an example:

```
character*10 name
data name /14/
```

REPEATED KEYWORD

Each keyword subparameter in an I/O statement or auxiliary I/O statement can be specified only once.

SUBSCRIPTS ON A SIMPLE VARIABLE

A variable that is not defined as an array cannot appear with subscripts.

SYNTAX ERROR — This is the catch-all error message for syntax problems. Some frequent causes:

- An expression or a term of an expression has been ommitted (missing variable or constant).

```
Example : RESULT =    * TIME
          SUMMATION =
          WRITE(6, FMT='(1X, F3.1)') SUM
```

- Superfluous text was found at the end of a syntactically correct statement. Check for typing or syntax errors (extra characters following a valid statement).

```
Example :  ICOUNT = ICOUNT+1,
```

- A required variable or subprogram name was not found.
- Two terms of an expression are not sepatated by an operator or a punctuation mark(such as comma) has been omitted.
- A floating-point constant was specified in E notation but the exponent was omitted.
- A required constant was not found.
- A character constant must be enclosed in parenthesis.
- An invalid operation was attempted in an implied-DO list in a DATA statement, for instance a function reference in the subscript or substring expression of an array or character substring reference. Example:

```
DATA (A(SIN(REAL(I))), I=1, 10 )   /100.0/
```

- Extra comma in format list.

TERMINATION CODE 139

The upper bound of a dimension declarator must be equal to or greater than the lower bound.

THERE IS A BRANCH TO STATEMENT LABEL FROM OUTSIDE BLOCK

DO or IF statement incorrectly nested.

UNBALANCED PARENTHESIS, STATEMENT SKIPPED

Two terms of an expression are not separated by an operator or a punctuation mark (such as comma,parenthesis) has been omitted.

UNBALANCED QUOTES; CLOSING QUOTE SUPPLIED, (SYNTAX ERROR)

A missing quote was detected and supplied by the system.

UNDEFINED: — Followed by a list of routine names. The linker *ld* cannot find the routines in the libraries searched. To check which libraries were searched, recompile with -t option.

UNDEFINED ENTITY :

A control parameter expression in an implied-DO list in a DATA statement contains a name which is not the name of a control variable of an implied-DO list which has the name of its scope. Example:

```
DATA (A(J), J=1,10),(B(I), I=J,K) /100./
```

WRITE SFE:[100] ERROR IN FORMAT ILLEGAL INSTRUCTION

- Extra comma in format list
- Extra number in format list
- An invalid character or a syntax error was detected in a FORMAT statement.

WRONG NUMBER OF SUBSCRIPTS
: More or fewer dimensions are referenced than were declared for the array.

ZERO STEP IN IMPLIED DO LOOP
: An invalid control parameter was detected in an implied-DO list in a data statement; for instance an increment of zero.

13 Appendix B: Run-Time Errors

ASSIGNMENT TO AN UNDIMENSIONED ARRAY
: An assumed array name was used, where the array name was not declared.

ARITHMETIC EXCEPTION: FLOATING DIVIDE BY ZERO
: During a floating point arithmetic operation, an attempt was made to divide by zero.

ARITHMETIC EXCEPTION :FLOATING POINT OVERFLOW
: During an arithmetic operation, a floating point value exceeded the largest representable value for that data type.

BUS ERROR
: The number of arguments in the dummy argument list is greater than the number of arguments in the actual argument list in a main/subprogram to subprogram interface (access violation, RM=01).

READ UNEXPECTED CHARACTER
: During a formatted operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable (Input conversion error).

* * * (ASTERISKS)
: During a formatted output operation the value of a particular number could not be output in the specified field length without loss of significant digits. The field is filled with asterisks (Output conversion error).

DOFIO :[-1] END OF FILE DURING READ
: An end-of-file condition was encountered during execution of a READ statement that did not contain an END, ERR or IOSTAT specification.

IERR = -1 — Attempt to access a nonexistent record.

IERR = 104 — Mixed access modes. An attempt was made to use formatted and unformatted operations on the same unit.

IERR = 103 — access='sequential' was performed on a direct access file.

ILLEGAL UNIT NUMBER
: A logical unit number greater than 99 or less than zero was used in an I/O statement.

PERMISSION DENIED.
: There were errors during linking when you compiled the program. ("Undefined:") The linker has set the file's protection to "no owner execute."

14 Appendix C: Errors Undetected by f77

1. The control variable of a DO loop has been altered within the range of the DO statement. Assignment to DO variable within loop. Example:

```
        do 14 i=1,20
          i = 20
          x(i) = 0
     14 continue
```

2. Extra comma in format list. Example :

```
        Format(1x, a, f3.1,)
```

3. Missing separator between format items. A required separator character has been omitted between fields in a FORMAT statement. Example:

```
        Format(1x, a  f3.1)
```

4. A variable appears in more than one data type declaration statement. Or a name appears in two or more inconsistent declarations statements (Multiple declaration of name).

```
          real x(5), x(5)
   or
          real x(5) , ....
          real x(5) , ....
```

5. Missing number in format list

```
   Example :    FORMAT(1x, a, f6. )
```

   The system assumes that the number of characters to the right of decimal point is one and there is an accuracy error involved.

6. Using an array within a program without declaring it with a DIMENSION statement.

```
Example:
  c
         do 14 j=1,20
          x(j) = 10.5
          write(6, fmt=*) x(j)
  14    continue
  c
        stop
        end
```