A BRIEF SUMMARY OF FORTRAN

1. FORTRAN STATEMENTS

Most of the statements described here are valid both in the current version of the language (called Fortran-95) and the older version (Fortran-77).

Statements in Fortran-95 are free-format. They may go anywhere from column 1 to 132. Variable names and other parts of a statement may use upper or lower case.

If a statement needs to be continued on more than one line, put an ampersand & at the end of each line except the last one.

Only those statements that are referenced by other statements need to have statement numbers. Blanks may be inserted in a statement to improve its readability and they are ignored by the Fortran compiler, except that blanks may not be inserted into names, keywords like IF, or constants.

2. CONSTANTS AND ARITHMETIC

There are three primary types of Fortran constants:

INTEGER For example 3, -27, 12345.

REAL For example 12.7, -314.0, 2.71828E+20. The last example is in exponential format and it means 2.71828 times ten to the power 20.

DOUBLE For example 3.141592653689793D0. The D exponent indicator means the constant is double precision. Merely giving more digits is not enough to make the constant double precision — the D must be present.

Real constants have about 7 significant digits of precision on a 32-bit computer, and double precision gives about 16 significant digits.

The valid range for both integers and real values may vary from one machine to another. On most 32-bit computers the range is:

INTEGER: -2 147 483 648 to 2 147 483 647

REAL: -3.403E+38 to 3.403E+38

DOUBLE: -1.798D+308 to 1.798D+308

Several other data types are available, including single and double precision complex numbers, character variables, logical variables, and fancier things, but we will concentrate on the three above.

The type of arithmetic that is done in an expression depends on the type of data present. If both operands of an operation are integers then the result is type integer. Division is the main operation where this makes a big difference. If two REAL (also called Floating Point) values are divided a REAL result is obtained that closely approximates the true quotient. If two INTEGER values are divided
the result obtained is the next integer toward zero from the true quotient.

Examples:

13.0/17.0 gives about 0.7647059
5.0/2.0 gives 2.500000
5/2 gives 2 (integer division)
13.0D0/17.0D0 gives about 0.7647058823529412

3. VARIABLE NAMES

A variable name must have between 1 and 63 characters. The first character must be a letter, and subsequent characters may be letters, digits, or underscores (for example X, A6, alpha2, LAST_VALUE).

A variable name is assumed to have only one type of value stored in it. The default rule of Fortran is that names beginning with one of the letters I, J, K, L, M, or N are INTEGER variables, and all others are REAL.

If we want to use LAMBDA as a real variable and Z as an integer, we can explicitly declare them at the top of the code:

```fortran
REAL :: LAMBDA
INTEGER :: Z
```

Using implicitly typed variables does not allow the compiler to do as much error checking as we would like. Putting the following statement before variable declarations and then explicitly declaring all variables tells the compiler that if we try to use a variable that we have not declared, it should give us an error message.

```
IMPLICIT NONE
```

This will catch errors caused by typos like LAMDBA that might be hard to spot.

4. ASSIGNMENT STATEMENTS

To assign a value to a variable, any arithmetic expression may appear on the right side of the equal sign. Examples:

```fortran
X = Y + 2.0
K = 5*(J - LAST)*(14 + M3)/2
J = K/4
```

Note that the division in the last example will be integer division since K is an integer variable. So if K has the value 3 then J will be set to 0.

Operations that combine integer and real values in the same expression are called mixed-mode operations. The rule is that if an integer and a real value are combined the integer is first converted to real and then a real operation is done.
Examples:

\begin{align*}
  X &= Y/2 & \text{This will be done as } Y/2.0. \\
  Z &= K & \text{The integer value is converted to floating point format and then stored in } Z. \\
  D &= 1.000/J & \text{The integer value is converted to double precision format and then divided in double precision.} \\
  L &= B & \text{The value of } B \text{ is chopped to the next integer toward zero and then stored in } L. \\
  X &= K/2 & \text{Be careful of this. If } K \text{ is 7 then } X \text{ will have the value 3.0 since an integer division is done first.} \\
\end{align*}

5. COMMENT LINES

An exclamation point ! means that anything after that on a line is a comment to be read by humans but ignored by the compiler.

6. DO STATEMENT – LOOPING

The DO statement defines a group of statements to be executed repetitively. Examples:

\begin{verbatim}
DO J = 1, 10
   K = K + J
ENDDO
\end{verbatim}

The body of the loop has been indented to make the loop easier to read. This loop will be done 10 times while the index variable J runs through the values 1, 2, 3, \ldots, 10.

The only way to enter a DO loop is from the DO statement itself. Branching into the middle of the range of a DO loop is not allowed. Branching outside the range of the loop from within its range is allowed.

The values defining the starting and stopping values for the DO loop index variable may be any integer values, and integer variables or integer expressions may be used:

\begin{verbatim}
DO NUMBER = KA, KB
   (body of the loop)
ENDDO
\end{verbatim}

Here the loop index variable is NUMBER and it runs through the values KA, KA+1, \ldots, KB.

The most general form of the DO loop is:

\begin{verbatim}
DO JT = KA, KB, INC
   (body of the loop)
ENDDO
\end{verbatim}

JT takes on the values KA, KA+INC, KA+2*INC, \ldots, KA+L*INC, where KA+L*INC is between KA and KB (inclusive), and KA+(L+1)*INC is not between KA and KB.

Expressions are allowed in the DO statement:
\begin{verbatim}
DO J = 1, N+1
   (body of the loop)
ENDDO

Similarly if the index variable is to take on the values N, N-1, N-2, \ldots, 2, 1 then the loop can be written
\begin{verbatim}
DO J = N, 1, -1
   (body of the loop)
ENDDO
\end{verbatim}

The loop index variable should be assumed to be defined only within the range of its loop. To get the value of a loop index variable outside the loop another variable should be defined within the loop that has the desired value.

Two other statements can be used to control loops:
\begin{verbatim}
CYCLE
\end{verbatim}
This causes the program to skip to the next \texttt{ENDDO} without executing the intervening statements.

\begin{verbatim}
EXIT
\end{verbatim}
This causes the program to exit the loop, skipping to the statement after the next \texttt{ENDDO}.
\end{verbatim}

7. BRANCHING STATEMENTS

Statements that are referred to in other statements have statement numbers. A statement does not need a number if it is not referenced elsewhere in the routine.

Unconditional branch:
\begin{verbatim}
GO TO 70
\end{verbatim}
This causes the program to jump to statement 70 next.

Conditional branches:
\begin{verbatim}
IF (J == 7) GO TO 40
\end{verbatim}
Note that there are two equal signs, since this is a logical comparison, not an assignment. This statement means if J is equal to 7 then \texttt{GO TO 40}. If J is not equal to 7 then the \texttt{GO TO 40} is not executed and the statement after this \texttt{IF} statement is executed next.

\begin{verbatim}
IF (XT < 0.57) Y = Y + XT
\end{verbatim}
This means if XT is less than 0.57 then execute the statement \texttt{Y = Y + XT}. Otherwise skip it and proceed to the next statement after this \texttt{IF}.

The six comparisons that can be made are:

\begin{verbatim}
==  Equal
=/= Not equal
>  Greater than
\end{verbatim}
Less than
\geq\quad \text{Greater than or equal to}
\leq\quad \text{Less than or equal to}

In addition, the logical comparisons in these IF statements may be combined using .AND. and .OR.
For example:

\begin{verbatim}
IF (J == 1 .AND. KA >= 5) L = L + 1
\end{verbatim}

Here the \( L = L + 1 \) is done only if both logical expressions are true.

\begin{verbatim}
IF (J < 4 .OR. X >= 3.7) GO TO 120
\end{verbatim}

In this case GO TO 120 will be executed if either one (or both) of the logical expressions is true.

The block IF allows a block of statements to be executed conditionally.

\begin{verbatim}
IF (X > XMAX) THEN
  JMAX = J
  XMAX = X
ENDIF
\end{verbatim}

This example shows the simplest use of a block IF. If the condition is true, the block of two
statements will be executed. Since the number of statements in a block is arbitrary, the ENDIF
statement is used to signal the end of the block.

The rules for the IF statement are the same as for a Logical IF, except the executable statement
after the IF is replaced by THEN. The statements in the block are all on separate lines, as is the
ENDIF.

As usual the rules of Fortran do not require that the block be indented as the example above, but it
makes the code much more readable if blocks of code such as DO loops and IF blocks are indented.

The next version of the block IF allows one of two blocks to be executed:

\begin{verbatim}
IF (Y >= 0.0) THEN
  Z = SQRT(Y) + T
  J = J + 1
ELSE
  WRITE (KW,10) Y
10    FORMAT(/' ERROR DUE TO NEGATIVE Y VALUE. Y =',E15.7/)
ENDIF
\end{verbatim}

Here if the condition \( Y \geq 0.0 \) is true the first block is executed, and if it is false the second
block is executed.

For selecting one of several cases the ELSE IF statement can be used.

\begin{verbatim}
IF (A < B .AND. X < Y) THEN
  (case 1)
ELSE IF(A < B .AND. X >= Y) THEN
  (case 2)
\end{verbatim}
ELSE IF(A \geq B .AND. X < Y) THEN
  (case 3)
ELSE
  (case 4)
ENDIF

8. FUNCTIONS

Most of the standard mathematical functions are part of the built-in Fortran library:

\[
\begin{align*}
A &= \text{ABS}(X - 2.0) & \text{Absolute value.} \\
J &= \text{ABS}(K + L1) & \text{Integer absolute value.} \\
Y &= \text{SQRT}(D \cdot D) & \text{Square root.} \\
Z &= \text{SIN}(T) & \text{Sine. Argument in Radians.} \\
R &= \text{EXP}(Y + 1.0) & \text{Exponential base e.} \\
B &= \text{LOG}(C \cdot C) & \text{Natural logarithm.} \\
Z &= X \cdot Y & X \text{ to the power } Y. \\
Z &= X \cdot K & X \text{ to integer power.} \\
\Pi &= 4.0 \cdot \text{ATAN}(1.0) & \text{Inverse Tangent. Result in Radians.}
\end{align*}
\]

Many other functions are available. Functions such as \texttt{ABS} that can be used for different types of arguments are called \textit{generic} functions. The type of argument used determines which version of the function is called. Generic function names do not have to follow the default rules for \texttt{REAL} or \texttt{INTEGER} type arithmetic. Note the \texttt{LOG} function above returns a \texttt{REAL} value. Some compilers consider expressions such as \texttt{SQRT(K)} or \texttt{EXP(3)} to be errors, since those functions expect \texttt{REAL} arguments.

Functions may appear in any arithmetic expression, and they may have any arithmetic expression for an argument. These are legal statements:

\[
\begin{align*}
Z &= \text{LOG}(2.0 + \text{ABS}(X - Z)) \\
A &= \text{LOG}(\text{SQRT}(\text{ABS}(\text{SIN}(T + \pi/4) + 2.6)))
\end{align*}
\]

The programmer may also define functions to be used in the program. This is done by using a function subprogram. In Fortran each routine in a program is terminated by an \texttt{END} statement. The entire program consists of a main program and (optionally) one or more subprograms. The following example shows a main program that uses a function subprogram to calculate the distance between two points:

```
PROGRAM D
  ! Main program.
  IMPLICIT NONE
  REAL :: X1,X2,Y1,Y2,D,D2
  X1 = 1.2
  Y1 = 2.6
  X2 = 3.1
  Y2 = 4.1
  D = DIST(X1,Y1,X2,Y2)
  D2 = DIST(X1,Y1,5.6,Y2+3.0)
  STOP
END PROGRAM D
```
FUNCTION DIST(A,B,C,D)

! Compute the distance between (A,B) and (C,D).

IMPLICIT NONE
REAL :: DIST,A,B,C,D,XSQ,YSQ
XSQ = (C - A)**2
YSQ = (D - B)**2
DIST = SQRT(XSQ + YSQ)
RETURN
END FUNCTION DIST

The two END statements tell the compiler where the last statement of each routine is. A given routine has only one END statement, and it is always the last statement of that routine.

The STOP statement is an executable statement that terminates the execution of the program. There can be more than one STOP statement in the program, and the first one that is executed stops the program.

The RETURN statement is similar to the STOP statement, but it means the currently executing subprogram is done and control is then returned to the calling routine. A given subprogram may have more than one RETURN statement. A STOP or RETURN statement that occurs just before the END statement on a routine is optional. If omitted the compiler assumes one was present.

The way a function subprogram defines the function value that is returned to the calling routine is by setting the function name (DIST in the example) to the desired value. Since it has a value associated with it, a function name follows the same rules as variable names regarding REAL or INTEGER type.

The way values are communicated to a subprogram is by establishing a correspondence between a variable or expression in the calling program and a variable in the subprogram. The first time function DIST is used in the example above, control goes to the function and while the statements in DIST are executed variable A in the function corresponds to variable X1 in the main program, B corresponds to Y1, etc.

When XSQ is computed during the first call of the function it will be (3.1 - 1.2)**2, so XSQ is given the value 3.61. Similarly YSQ will have the value (4.1 - 2.6)**2, which is 2.25. So DIST is computed to be about 2.420744. This is the only value that is returned to the main program. When the RETURN is executed, control passes back to the statement in the main program from which this call was made. Then D is assigned the value 2.420744 in the main program.

The second call to DIST causes another (possibly different) correspondence to be set up between variables in DIST and values in the main program. Upon return from the second call D2 will have the value SQRT((5.6-1.2)**2 + (7.1-2.6)**2), or about 6.293648.

9. SUBROUTINES

A function subprogram may only return a single value to the calling program. If a more complicated result is desired, the subroutine is a more general type of subprogram. As with a function it starts
with a **SUBROUTINE** statement that gives the name of the routine and the argument list, and it
ends with an **END** statement. One or more **RETURN** statements appear in the subroutine. Since the
subroutine does not return one function value its name has no value associated with it upon return
and so whether an integer or real name is chosen does not matter.

As with a function subprogram the variable names in the argument list are identified with the
corresponding variable names in the calling routine. This means a given variable must be of the
same type (Real or Integer) as the variable or constant it corresponds to in the calling routine.

Any variables that are defined in a subprogram and are not in the argument list are considered
local to that subprogram. This means that the same variable name can be used in more than one
routine with no connection between them.

Statement numbers are also local to a given subprogram, so that the same statement number can
be used in more than one routine.

For example:

```fortran
PROGRAM S
   (first part of the main program)
   10 CALL SUB(X,Y,A,B)
   (more main program)
   20 CALL SUB(X1,X2,X3,X4)
   (rest of the main program)
END PROGRAM S

SUBROUTINE SUB(A,B,C,D)
   IMPLICIT NONE
   REAL :: A,B,C,D
   IF (A < B) GO TO 10
   C = C + A - B
   D = A
   RETURN
9   10 C = C + B - A
   D = B
   RETURN
END SUBROUTINE SUB
```

Here the three basic types of communication between a calling program and a subroutine are
illustrated. The values of **A** and **B** are “sent” to the subroutine by the calling program since they
are used but not re-defined within the subroutine. The value of **D** is “returned” to the calling
program since it does not have to be defined when the subroutine is called, but it is defined within
the subroutine and the value is available to the calling program in the corresponding variable upon
return from the subroutine. The variable **C** is used for “two-way” communication, since it must be
defined when the subroutine is called, and it is re-defined within the subroutine.

Notice that the use of variables **A** and **B** in the subroutine does not conflict with the variables **A**
and **B** in the main program. In the first call **A** in the subroutine refers to **X** in the main program,
**B** refers to **Y**, and **C** refers to **A** (in main), and **D** refers to **B** (in main). Since **C** and **D** are assigned new
values in the subroutine, **A** and **B** in the main program will be changed accordingly upon return
from the first call. X and Y in main will retain their original values since the subroutine does not change A or B.

Statement number 10 is used in both the main program and the subroutine. This causes no trouble since a reference to a statement number always means that statement number in the same routine.

10. ARRAYS — VARIABLES WITH SUBSCRIPTS

Subscripts are used to refer to a vector or a matrix in mathematical notation. In Fortran all characters appear on the same line, so subscripts are enclosed in parentheses. The third element of vector X is written \( x_3 \) in mathematical notation, and is referred to as \( X(3) \) in Fortran. The subscript can be an integer constant or expression such as \( X(J) \) or \( Y(L*N - 7) \).

Array names are chosen like regular variable names to specify whether the values stored in the array are Integer or Real. Since the one array name refers to many values the number of elements of an array must be declared, so that the right number of memory locations can be reserved. The declaration statement must go at the beginning of a routine, before any executable statement.

When an array name appears in the argument list of a subprogram the name must be declared to be an array in that subprogram also. Example:

```fortran
PROGRAM X
 IMPLICIT NONE
 INTEGER :: J
 REAL :: X(100), A(20,21)
 DO J = 1, 100
   X(J) = 0.0
 END DO
 CALL XSUB(X)  
 (rest of the main program)
 END PROGRAM X

SUBROUTINE XSUB(T)
 IMPLICIT NONE
 REAL :: T(100)
 (rest of the subroutine)
 END SUBROUTINE XSUB
```

In this example X is a one-dimensional array with 100 elements. They are \( X(1), X(2), \ldots, X(100) \). A is a 20 by 21 matrix. 420 memory locations are reserved for the elements of A. They are stored in this order:

\[
A(1,1), A(2,1), A(3,1), \ldots, A(20,1), A(1,2), A(2,2), \ldots, A(20,2), \ldots, A(1,21), A(2,21),
A(3,21), \ldots, A(20,21).
\]

In many mathematical applications, it is more convenient if subscripts start with 0 instead of 1. To declare an array \( X(0), X(1), \ldots, X(100) \), here is the statement:

```fortran
REAL X(0:100)
```

Using this feature, arrays with negative subscripts can be defined, as well as arrays whose subscripts
run only from 1950 to 2050, etc.

Fortran also allows the entire array or some part of it to be referred to as a single object. After \( X \) is defined as above we can initialize the entire array to zero with a single statement:

\[
X = 0.0
\]

We can also refer to an array section as a single object and use it in an arithmetic expression.

\[
Y(1:10) = 2*X(21:30) + 3
\]

This statement takes the ten items \( X(21), \ldots, X(30) \), multiplies by 2 and adds 3 to each one, then stores the resulting values in \( Y(1), \ldots, Y(10) \).

11. INPUT/OUTPUT

This is usually one of the more complicated parts of a high-level programming language. To be able to specify precisely the format of the input data that is read or the output data that is printed requires a detailed description of that format be given. In Fortran input and output is controlled using \texttt{READ}, \texttt{WRITE}, and \texttt{FORMAT} statements:

\[
\texttt{READ (5,40) A,B,C,I,J} \\
\texttt{40 FORMAT(3F10.4,2I5)}
\]

This pair of statements causes values to be read from an input file and assigned to the variables \( A, B, C, I, \) and \( J \).

The 5 in the \texttt{READ} statement is the “unit number” on which the input is to be done. It is possible for a program to be reading data from several different input files at the same time, and the unit number in a \texttt{READ} statement identifies which input file is used.

The 40 in the \texttt{READ} statement is the statement number of a \texttt{FORMAT} statement that specifies the format of the input data.

\( A,B,C,I,J \) is the “read list” of variable names that are defined by the read operation.

\texttt{3F10.4} means there are 3 fields of Fixed-point real numbers in the input record. Each field is 10 columns wide with 4 digits after the decimal point. This tells how to read \( A, B, \) and \( C \).

\texttt{2I5} means there are then 2 fields of Integer numbers, and each of these fields is 5 columns wide. This tells how to read \( I \) and \( J \).

These 5 numbers are all on one record (line) of the input file.

Here is a sample input record. The first line serves only to show which columns the data items are in. It would not be part of the input file.

```
---+--- 1  ---+--- 2  ---+--- 3  ---+--- 4  ---+--- 5
-17.6 3.1416 10.0 187 -31
```

If this line is read by the \texttt{READ} statement above then the values assigned are:

\[
A = -17.6 \quad B = 3.1416 \quad C = 10.0 \quad I = 187 \quad J = -31.
\]
In addition to the I and F format descriptors described above, there is an E format for real values in Exponential format (scientific notation). There are also ways to skip columns, to skip to a new line, and to print character strings. These are illustrated in the following WRITE statement:

```
WRITE (6,50) A,B,C,I,J
50 FORMAT(/' A,B,C=',3E15.7//' I,J=',I6,3X,I7)
```

In the WRITE statement 6 is the output unit number, 50 is the FORMAT statement to be used, and A,B,C,I,J is the output list.

/ is the FORMAT command to go to a new line. Each time a READ or WRITE statement is executed a new line command is automatically given, so the initial / in this FORMAT statement will leave a blank line between this output line and the last one.

'A,B,C=' is a character string that will be printed. Anything that appears between quotes in a FORMAT statement will be printed at that point in the output. For defining character strings we can use either single quotes ('A,B,C=') or double quotes ("A,B,C=") to delimit the string.

Either a comma or a slash (/) must separate items in a FORMAT statement. Since the character string 'A,B,C=' and the values of A, B, and C are to be printed on the same line, the comma separator is used there.

3E15.7 means that the next 3 items in the write list are to appear in Exponential format in fields 15 columns wide with 7 significant digits. This format is often used when the size of the output values is not known in advance.

// means skip to a new line twice. This will leave one blank line between the one with A, B, and C and the one with I and J.

'I,J=' is another character string used to label the output.

I6 means the next item is an integer to be printed in a field 6 columns wide.

3X means skip 3 spaces on the line.

I7 will print J in I7 format.

If the variables have the values read in the input example above, then the output will be as follows. The first line shows which column the output is in, and does not appear in the output.

```
-----+----1----+----2----+----3----+----4----+----5----+----6
A,B,C= -0.1760000E+02 0.3141600E+01 0.1000000E+02
I,J= 187 -31
```

On some computers (mostly mainframes), the blank that is the first character of the character strings to be printed at the start of each line does not appear in the output. Some printers use the first character of each output line for “carriage control”. It is not printed, but is used to specify options for the printer. In this case a blank signifies that the printer is to single space before printing. Here is a list of some of the carriage control characters:

- blank - single space
Most desktop computers and printers do not support carriage control commands, and anything in column 1 is just printed.

For maximum portability, it is a good idea to leave column 1 blank in each output line. Then the program can be run without change on mainframes or desktop machines.

In E format the field width is always at least 7 more than the number of significant digits printed. This is to allow room for the “0.”, the “E+” or “E-”, two exponent digits, and a minus sign.

An error occurs if the field width specified is not big enough to hold the number. For example, if the integer value 12345 is printed using an I4 format specification, the value cannot be printed. Most Fortran compilers signify this error by filling the field with asterisks, so that ****** will appear.

When reading or writing arrays it is often necessary to refer to several of the elements at once so they will be read from one line or printed on one line. Consider this example:

```fortran
DO J = 1, 100
    WRITE (6,60) X(J)
60   FORMAT(1X,6F12.5)
ENDDO
```

This will write the array one element per line. Even though the FORMAT specifies 6 fields per line, only one is used since each time through the loop the WRITE statement automatically goes to a new line.

One way to print the numbers six per line is with an array section:

```fortran
WRITE (6,60) X(1:100)
60 FORMAT (1X,6F12.5)
```

This can also be done with an implied DO loop:

```fortran
WRITE (6,60) (X(J),J=1,100)
60 FORMAT (1X,6F12.5)
```

In both cases the code tries to write the entire array on one line. The rule in Fortran is that when the FORMAT specification is used up (after the first 6 items are printed on the first line) then the FORMAT statement is scanned backwards until a left parenthesis is found, the output is advanced to a new line, and the FORMAT scan resumes. This will continue until all the items in the WRITE list have been printed. In this case the array X will be printed 6 items per line for 17 lines. The last line has only 4 items printed.

To label the output the following could be used:

```fortran
WRITE (6,60) X(1:100)
60 FORMAT(/' THE X ARRAY IS:'//(1X,6F12.5))
```

The 1X in these examples is another way to put a blank in column 1 for carriage control.
Some people think E-format output looks better without the leading zeros, so that pi prints as 3.14159E+00 instead of the 0.31416E+01 that comes from E15.5. There is an ES-format for scientific notation for doing this. Using ES15.5 format will cause pi to print as 3.14159E+00.

The FORMAT statement is useful if more than one read or write in the same routine refers to the same format, but if a format is to be used just once, it may reduce clutter (and statement numbers) to put the format into the read or write. The first example of this section could be written:

```
READ (5, "(3F10.4,2I5)") A,B,C,I,J
```

12. FREE FORMAT INPUT/OUTPUT

Sometimes it is convenient to be able to read or write without using a FORMAT statement. For example, a main program might prompt the user to enter test data from the keyboard. The user doesn’t want to count columns or align the input values.

```
WRITE (6,*) ' Enter A,B,I,J '
READ (5,*) A,B,I,J
```

The WRITE statement prints the character string, and the asterisk in place of a FORMAT number specifies free format.

The READ statement reads two floating-point numbers and two integers. They can be in any columns, as long as the numbers are separated by one or more blank spaces.

Many compilers assume that reading on unit 5 automatically takes input from the keyboard, and writing on unit 6 puts output on the screen. However, these unit numbers are not the same on all compilers.

One solution that is available on all compilers is to use default unit numbers. This will read from the keyboard and write to the screen on any compiler.

```
WRITE (*,*) ' Enter A,B,I,J '
READ (*,*) A,B,I,J
```

13. FILE INPUT/OUTPUT

The next example shows how to read input from a specific file and write output to another file. The OPEN statement tells the compiler which unit number refers to which file.

```
OPEN (14,FILE='input data')
READ (14,*) A,B,N
OPEN (16,FILE='results')
WRITE (16,*) ' A,B,N= ',A,B,N
```

After a file is opened, any number of reads or writes can be done. The OPEN statement should be executed only once, before any read or write statement refers to that unit.

13
14. DOUBLE PRECISION

Double precision is a 64-bit data type that gives about 16 significant digits on a 32-bit computer. No variable names default to double precision type, so they must be declared. A program or routine that uses variables A,B,X,Y,Z as double precision needs to declare it at the beginning of the routine.

```
DOUBLE PRECISION :: A,B,X,Y,Z
```

One potential pitfall when converting a program from single to double precision is that a variable might be missed and not declared as double in the new version. Then it would still be single precision in the new program and that could ruin the accuracy of the program. This is another reason to use `IMPLICIT NONE` and declare all variables.

Every routine that uses double precision variables must declare them. It is considered a good programming practice to explicitly declare all variables in each routine and to use `IMPLICIT NONE`. Writing bug-free software is a very difficult task, and the compiler can do more error checking when we use `IMPLICIT NONE`.

Constants must also be changed to double precision. In single precision the power of ten follows “E”, as in 1.234567E+5 for the value 123456.7 (= 1.234567 * 10.0**5). For double precision, write 1.234567D+5. The “+” is optional, so this can also be written 1.234567D5.

If no exponent letter is present, as with \( X = 0.1 \), the constant is single precision. In this case, even if \( X \) is a double precision variable, 0.1 is created as a single precision constant before it is converted to a double precision value by putting zeros after the single precision value. Since 0.1 cannot be exactly represented in base 2, the value that ends up stored as the value of \( X \) is accurate only to single precision. Writing \( X = 0.1D0 \) solves this problem.

Similarly, in an expression like \( X = Y + (2.0/3.0)*Z \), the division 2/3 will be done in single precision. To get double, write \( X = Y + (2.0D0/3.0D0)*Z \).

Functions also exist in both single and double precision versions. \( X = SQRT(3.0)*Z \), is accurate to single precision, and \( X = SQRT(3.0D0)*Z \) to double precision.

15. EXAMPLE PROGRAM

```
PROGRAM TESTA
!
! This program tests subroutine ADDMAT.
!
IMPLICIT NONE
INTEGER :: I,J,kr,Kw,N
REAL :: A(20,20),B(20,20),C(20,20)
KR = 5
KW = 6
10 WRITE(KW,*) ' Enter N (matrix size). N<1 to stop. '
READ(KR,*) N
IF (N <= 0) STOP

```

14
! Read two N by N matrices.

DO I = 1, N
   READ (KR,"(5E15.7)") (A(I,J),J=1,N)
ENDDO
DO I = 1, N
   READ (KR,"(5E15.7)") (B(I,J),J=1,N)
ENDDO
!
   Call ADDMAT to add the matrices.

CALL ADDMAT(A,B,C,N)

WRITE (KW,"(//' A='/)")
DO I = 1, N
   WRITE (KW,"(1X,5E15.7)") (A(I,J),J=1,N)
ENDDO
WRITE (KW,"(//' B='/)")
DO I = 1, N
   WRITE (KW,"(1X,5E15.7)") (B(I,J),J=1,N)
ENDDO
WRITE (KW,"(//' C='/)")
DO I = 1, N
   WRITE (7,"(1X,5E15.7)") (C(I,J),J=1,N)
ENDDO
!
   Go back to the top and see if there is another case to test.

GO TO 10
END PROGRAM TESTA

SUBROUTINE ADDMAT(X,Y,Z,N)
!
   Add the two N by N matrices X and Y.
   The sum is returned in Z.
!
   IMPLICIT NONE
   INTEGER :: I,J,N
   REAL :: X(20,20),Y(20,20),Z(20,20)
   DO I = 1, N
      DO J = 1, N
         Z(I,J) = X(I,J) + Y(I,J)
      ENDDO
   ENDDO
   RETURN
END SUBROUTINE ADDMAT
16. NON-STANDARD FEATURES

Most compilers support some features that are not part of the Fortran standard. Using these features will severely limit the portability of a program, since it will probably not run on any computer not using the same compiler. Some features to avoid:

Many editors on Macintosh or PC computers automatically insert \texttt{tab} characters in the file in place of several blanks. This saves space in the file, but tabs are not standard Fortran characters. Since the number of spaces each tab represents may be different for different machines, using tabs can cause portability problems. It is best to turn off any auto-indent or other feature that can put tabs into the file when editing Fortran programs.

The language described above is only a small part of the Fortran language. Many of the more advanced features of the language are not mentioned here.

17. TESTING AND DEBUGGING

Writing the code for a new program usually goes through several stages before we are satisfied that it works as intended. A nontrivial program very seldom works correctly the first time we run it, usually there are some errors to fix.

The first set of errors to correct are \textit{syntax errors}. These are caused by the program not conforming to the rules of the Fortran language. Often these are merely typing mistakes or leaving out a comma or parenthesis. Once we have gained some experience with the language, these errors are usually easy to fix. The compiler will give an error message for each of these. Some compilers are better than others at explaining what caused the error message, but after using a particular compiler for a while most are easy to interpret.

The second set of errors are \textit{logic errors}. Here the program compiles correctly, and so we can run the program, but some of the results are wrong. These errors can be much more difficult to correct. To catch logic errors, the first few times we run a program we test “small” cases where we know the correct answer and can compute it by hand.

When we have found a test case for which the program gives the wrong answer, the next step is to try to track down where the logic error is located. This is like a search procedure where we want to find the first line of code at which some intermediate result was wrong. The usual strategy at this point is to start putting write statements into the program, so we can see a trace of the execution with each step printed out. Then by comparing each step to the results we have computed by hand for this test case, we can see which statement caused the first bad result. That is usually enough for us to understand what caused the error, and then we can fix it.