Simplified Fortran Guide

K. R. Jackson

Department of Computer Science

Originally Written November 1984
Last Revised January 1992

This is a guide to a subset of Fortran 77, the current standard version of Fortran. For a complete description of the language, refer to one of the many Fortran texts.

Throughout this guide, items enclosed in square brackets (i.e., [ ]) are optional and items enclosed in braces (i.e., { }) may occur zero or more times.

Fortran key-words are capitalized; all other words are uncapitalized. Standard Fortran contains only capital letters. Although the UNIX† Fortran compiler f77 allows both capital and small letters in programs, the capitals are mapped to small letters (except for those occurring in character strings) before the program is compiled, so the case of variables and key-words is immaterial.

To compile a Fortran program on a UNIX system use

```
f77 name.f {others.f}
```

where ‘‘name.f’’ is the name of the file containing your Fortran main program and ‘‘{others.f}’’ are the names of other optional files containing Fortran subprograms. Each *source* file name must end with ‘‘.f’’. If a program or subprogram has not been changed since it was last compiled, then using the extension ‘‘.o’’ instead of ‘‘.f’’ links the precompiled *object* code (produced earlier by f77) to the other program segments being compiled and thus saves some computer time.

In addition, f77 can link your program to precompiled subroutine libraries. For example, to compile a program that calls subroutines from the Teapack library, use

```
f77 name.f -lteapack
```

If the f77 compiler finds no errors, it produces a file called ‘‘a.out’’. To run your program, simply type the command

```
a.out [ < input ] [ > output ]
```

where ‘‘input’’ and ‘‘output’’ are optional input and output files. If these are not supplied, the default is to read from and write to the terminal. You may rename a.out if you wish and execute the renamed file in a similar manner.

The f77 compiler has many options. Two that you might find helpful in debugging programs are -u and -C. The first makes all undeclared variables *undefined* and second checks for *subscripts out of range.* For a program compiled on a Sun 2 or 3 that does a significant amount of floating-point computation, it is also worth setting the -f option for the appropriate floating-point hardware if available, since the default is to do all floating-point computations in software. See

```
man f77
```

for further discussion of these and other options.

1. Textual Layout of a Program.

Fortran is a positional language: each line of the program file is divided into subfields. Columns 1–5 are reserved for statement labels, which are unsigned integers having at most 5 digits. Statements and declarations are written in columns 7–72 inclusive. ‘‘Card numbers’’, which are not of much use now that punch cards are rarely used, may appear in columns 73–80. (Characters beyond column 72 will not be read by the compiler. This is a very common source of error.) Each new line begins a new statement unless there is a nonblank character (often a +) in column 6. In this case, the statement from the preceding line is continued to this line. A comment begins with a C in column 1: the rest of the line may contain any comment and is ignored by the compiler. Each comment must have a C in column 1: comments may not be continued with a continuation mark in column 6.

† UNIX is a trademark of Bell Laboratories.

Blanks may be used freely in Fortran. Write your programs so that their physical layout reflects their logical structure.

2. Program Structure.

A Fortran program has the following structure:

```
{ declarations }
{ statements }
END
{ subprograms }
```

A *subprogram* is either

```
SUBROUTINE name [ ( ident { , ident } ) ]
{ declarations }
{ statements }
END
```

or

```
type FUNCTION name  (  ident { , ident } )
{ declarations }
{ statements }
END
```

where ident above is an abbreviation for an identifier. The parameter list for a subroutine may be empty, in which case the parentheses following the name of the subroutine are optional. However, parentheses are always required for a function even if the parameter list is empty. Valid *types* are listed below.

Assign the value that the function is to return to the function name. This is just as if the function name were a variable; Pascal and many other languages use the same convention. Also declare the function name in any program or subprogram that uses it. Declarations are described below.

If you pass either a function or subroutine X as a parameter to another subprogram Y, then X's name should be declared to be EXTERNAL in the program or subprogram that contains the call to Y. The EXTERNAL declaration is described below.

The main program must contain a STOP statement and a subprogram must contain a RETURN (or STOP) statement. When STOP is executed, the program terminates; when RETURN is executed, the subprogram terminates and execution resumes at the point where the subprogram was called. STOP or RETURN is the last statement executed in the program segment. Often, STOP or RETURN immediately precedes the END statement.

Subprograms are not recursive in Fortran 77. That is, they must not call themselves either directly or indirectly. Although many compilers, including f77, allow recursion, it is not wise to use it, since this will limit your program's portability.

Subroutines are called using the call statement

```
CALL name [ ( expr {, expr } ) ]
```

where expr is an abbreviation for an expression. Functions are called by using their name (followed by an appropriate argument list, of the same form as in the CALL statement) in an expression. The argument list of a subprogram must have as many arguments as there are parameters to the subprogram. They are matched on a one-to-one basis just as in Turing and many other languages. The types of arguments must match appropriately, although this is not enforced in Fortran as rigidly as it is in Turing. This is a source of many seemingly inexplicable bugs in Fortran programs.

If a value is to be returned through an argument, then that argument must be a variable or an array element.

3. Declarations.

The declarations in a program or subprogram should be in the same order as they are described below.

Fortran identifiers must begin with a letter and may contain up to six letters and digits. Using such short names effectively requires some practice. Many compilers, including f77, allow longer names, but using this extension limits portability.

3.1. Type.

Each variable used in a program should be declared in a type statement of the form

```
type variable {, variable }
```

where *type* is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, EXTERNAL or CHARACTER*n for a positive integer n. EXTERNAL is used only to declare a subprogram name in a program segment that passes that subprogram name as an argument to another subprogram. Note that there is no double precision complex in standard Fortran 77, although many compilers, including f77, provide it. Again, using this extension limits portability.

A variable may be a simple identifier or an array identifier declared as

```
identifier ( range {, range } )
```

where *range* is

n0 | n1 : n2

In the first case, n0 must be ≥ 1 and the array has elements 1..n0. In the second case, n1 must be ≤ n2 and the array has elements n1..n2.

An array element is referenced in the program as

identifier (expression {, expression })

where each expression must lie within the declared range of that dimension. A common error is *subscripts out of range* which arises when an array index falls outside the declared range of the array. This error usually causes the program to crash with a *segmentation fault error* and no helpful error diagnostics. The f77 −C compiler option mentioned at the start of this guide may help to locate the error in this case.

3.2. Parameters.

A parameter statement has the form

PARAMETER (name = expr {, ... })

where ... is another instance of name = expr and expr is an abbreviation for an expression. The expressions may contain the arithmetic operators +, –, *, / and **; the exponents with ** must be integers. The parameter declaration corresponds to the *const* declaration in Pascal: the value assigned to the identifier is fixed throughout the program. (This is more rigid than Turing's *const*.) Unlike those languages, though, the parameter statement does not declare a variable. If you do not declare it in a type statement (described above), the variable will be given a default type, which may cause errors if this type is not what you intended.

3.3. Data.

A data statement has the form

DATA nlist / clist / {, nlist / clist / }

Each *nlist* is a list of variable names, array names, array element names, and implied DO lists. (See READ and PRINT below for implied DO lists.) Each *clist* is a list of constants or symbolic names of constants, and may be preceded by an r* where r is an unsigned positive integer indicating r occurrences of the following constant in the list. The values in the *clist* are assigned to the variables in the *nlist* on a one-for-one basis.

3.4. Common.

A common statement has the form

COMMON / blockname / list

where *list* is a list of identifiers or array elements. No subprogram argument or function name may appear in the list. Fortran has no global variables. Common provides a means of sharing variables between program segments. The *blockname* must be the same in each segment. The identifiers in the *list* are matched one-for-one by position proceeding from left to right, without respect to their names or types in the different program segments.

4. Statements.

The SUBROUTINE, FUNCTION, RETURN, CALL, STOP, and END statements are described above.

4.1. Assignment.

An assignment statement has the form

variable = expression

The left side may be either a simple variable or an array element. Whole arrays cannot be assigned all at once: they must be assigned an element at a time in a DO loop (described below). If the type of the left and right sides do not match, the value of the expression is converted to the type of the left side variable (if possible) after the expression has been evaluated.

4.2. GO TO.

A GO TO statement is of the form

GO TO label

and transfers control to the statement having that label in columns 1–5. (Remember, as stated above, a label is an unsigned integer having at most 5 digits.)

GO TOs should not be used indiscriminately in Fortran programs. They should be used only to construct well-structured loops as described below.

4.3. CONTINUE.

A CONTINUE statement does nothing, but serves as a convenient point on which to attach a label as described below with respect to loops.

4.4. IF Statements.

There are several types of IF statements in Fortran 77. The following are particularly useful.

4.4.1. Logical IF.

```
IF ( logical expression ) statement
```

(The parentheses around the logical expression are required in the IF statement above and in those below.) If the logical expression is true, then the statement is executed. Otherwise the statement is not executed. Only one executable statement may be included.

4.4.2. IF–THEN.

```
IF ( logical expression ) THEN
   { statements }
END IF
```

If the logical expression is true, then the statements between the IF and the END IF are executed. Otherwise they are skipped.

4.4.3. IF–THEN–ELSE.

```
IF ( logical expression ) THEN
   { statements }
ELSE
   { statements }
END IF
```

If the logical expression is true, then the statements between the IF and the ELSE are executed. Otherwise the statements between the ELSE and the END IF are executed.

4.4.4. IF–THEN–ELSE–IF ...

```
IF ( logical expression ) THEN
   { statements }
{ ELSE IF ( logical expression ) THEN
   { statements }              }
[ ELSE
   { statements }              ]
END IF
```

The statements after the first logical expression that is true are executed; all the others are skipped. If no logical expression is true and there is an ELSE part, then the statements after the ELSE are executed. In any case, execution resumes after the END IF.

IFs may be nested. An ELSE is always associated with the closest preceding IF.

4.4.5. Logical Constants and Expressions.

A logical constant is either .TRUE. or .FALSE. A logical expression is of the form

```
logical constant
logical variable
comparison
.NOT. logical expression
logical expression .AND. logical expression
logical expression .OR. logical expression
```

where a *comparison* is

```
expression comparator expression
```

and a *comparator* is .LT. .LE. .GT. .GE. .EQ. or .NE. Expressions in brackets are evaluated first, then comparisons, then .NOT.s are applied, then .AND.s are performed, followed by .OR.s. Operations of equal precedence are performed left to right.

4.5. Loops.

Unlike Turing, Fortran does not provide a general ‘‘loop ... end loop’’ structure. You can build one as follows using the logical IF and GO TO statements.

```
C       LOOP
label1  CONTINUE
           { statements }
C       EXIT WHEN logical expression
        IF (logical expression) GO TO label2
           { statements }
        GO TO label1
C       END LOOP
label2  CONTINUE
```

4.6. DO loops.

A DO loop (or ‘‘indexed loop’’) is of the form

```
        DO label identifier = start, limit [, step]
           { statements }
label   CONTINUE
```

where *start*, *limit* and *step* are expressions that are evaluated before the loop is executed. The loop is executed with the identifier initialized to the value *start* and incremented by *step* (which has a default of 1) until the value of the identifier is greater than *limit*. In Fortran 77, unlike earlier versions of Fortran, the statements within the loop are not executed if *start* is greater than *limit* and *step* is positive or *start* is less than *limit* and *step* is negative. Usually, all values are integers, but other types are allowed.

4.7. Read and Print.

These statements are used to input and output values. They have the form

```
READ * [, variable {, variable } ]
```

or

```
READ label [, variable {, variable } ]
label FORMAT ( format item {, format item } )
```

and

```
PRINT * [, expression {, expression } ]
```

or

```
PRINT label [, expression {, expression } ]
label FORMAT ( format item {, format item } )
```

An expression may be any valid expression or an implied DO list. A variable is a simple identifier, an array element or an implied DO list. An implied DO list has the form

```
( dlist, identifier = start, limit [,step] )
```

where *dlist* is a list of permissible input or output items and *start*, *limit* and *step* are expressions. The implied DO works just as the DO statement described above does and is frequently used to read and write arrays as, for example,

```
READ *, (a(i), i=1,10)
```

which reads elements a(1), a(2),...,a(10) of the array a.

For both READ and PRINT, the * form uses default formats. Use the * form of the READ whenever possible. However, with PRINT, you may want to supply your own output format rather than using the default.

In a FORMAT statement, a format item is one of the following.

nX	skip the next n columns.
nIw	n integers right justified in fields of width w.
nFw.d	n real or double precision values without exponents right justified in fields of width w with d digits to the right of the decimal point.
nEw.d	n reals (or double precision) values with an exponent right justified in fields of width w with a leading 0, followed by a decimal point, followed by d digits.
nDw.d	like nEw.d except that the exponent is marked by a D rather than an E.
nAw	n groups of w characters.

Fortran uses the first character of each output line for carriage control. The first character should be one of the following.

‘ ’	(blank) start a new line.
‘1’	start a new page.
‘0’	skip a line then start a new line (double space).
‘+’	go back to the beginning of the current line (overprint).

fpr can be used to print a Fortran output file containing these carriage control characters. See

```
man fpr
```

for details.

5. Constants and Expressions.

Logical constants and expressions are described above.

5.1. Arithmetic Constants and Expressions.

Integer constants are of the form

```
[ sign ] digit { digit }
```

Real constants are of the form

```
[ sign ] { digits } . { digits } { E [ sign ] digit { digit } }
```

where at least one of the two groups of digits surrounding the decimal point must be nonempty. A few valid real numbers are

```
1.  .33   1.5E4  0.333E+10  4.2E-10
```

A double precision number is similar to a real number except that the exponent E is replaced by a D. The D is not optional – without it, the number is a single precision real.

Expressions may be formed using the arithmetic operators +, –, *, /, **, where ** represents exponentiation. If a and b are integers, then a/b returns the integer quotient of a divided by b (e.g., 3/2 = 1). Otherwise, the operators are as one would expect. The precedence of these operators is ** highest, * and / next, and + and – lowest. When operators are of equal precedence they are evaluated from left to right, except for ** which is evaluated right to left. A**B is computed by repeated multiplication if B is an integer, but is computed as exp(B*log(A)) if B is a real. Hence, if A is negative and B is real, an exception occurs even though this expression may be mathematically valid.

5.2. Character Constants.

A character constant is any string of characters enclosed in single quotes not containing a single quote. A quote may be included in a character constant by using two single quotes in a row. Some examples are

```
'FRED'  'X = 24'  'MR. O''REILLY'
```

6. Built-in Functions.

Fortran has many built-in functions. Some particularly useful ones are listed below.

MOD(m,n)	the remainder of m divided by n
ABS(x)	the absolute value of x
MAX(x1,...,xn)	the maximum of x1,...,xn
MIN(x1,...,xn)	the minimum of x1,...,xn
SQRT(x)	the square root of x
EXP(x)	e to the x
LOG(x)	log to the base e of x
LOG10(x)	log to the base 10 of x
SIN(x)	the sin of x (x in radians)
COS(x)	the cos of x (x in radians)
TAN(x)	the tan of x (x in radians)
ATAN(x)	the arctan of x (result in radians)

7. Sample Program

```
c  Sample program to compute and print n!
c  for n = 0,...,limit

      integer n, limit
      double precision fac
      parameter ( limit = 20 )

      do 10 n = 0,limit
         print 5, n, fac(n)
 5       format('n = ',i2,3x,'n! = ',d15.5)
10    continue

      stop
      end

      double precision function fac(n)

c     This function computes n!.
c     It assumes without checking
c     that n .ge. 0

      integer i,n

      fac = 1.d0
      do 10 i = 1,n
         fac = fac * dble(i)
10    continue

      return
      end
```

The output from this program follows.

```
n =  0,  n! = 0.1000000D+01
n =  1,  n! = 0.1000000D+01
n =  2,  n! = 0.2000000D+01
n =  3,  n! = 0.6000000D+01
n =  4,  n! = 0.2400000D+02
n =  5,  n! = 0.1200000D+03
n =  6,  n! = 0.7200000D+03
n =  7,  n! = 0.5040000D+04
n =  8,  n! = 0.4032000D+05
n =  9,  n! = 0.3628800D+06
n = 10,  n! = 0.3628800D+07
n = 11,  n! = 0.3991680D+08
n = 12,  n! = 0.4790016D+09
n = 13,  n! = 0.6227021D+10
n = 14,  n! = 0.8717829D+11
n = 15,  n! = 0.1307674D+13
n = 16,  n! = 0.2092279D+14
n = 17,  n! = 0.3556874D+15
n = 18,  n! = 0.6402374D+16
n = 19,  n! = 0.1216451D+18
n = 20,  n! = 0.2432902D+19
```