Elements of Fortran

Lloyd D. Fosdick

18 January 1987
Revised
June 21, 1995

High Performance Scientific Computing
University of Colorado at Boulder

The following are members of
the HPSC Group of the Department of Computer Science
at the University of Colorado at Boulder:

Lloyd D. Fosdick
Elizabeth R. Jessup
Carolyn J. C. Schauble
Gitta O. Domik

Contents

Trademark Notice

Elements of Fortran[*]

Lloyd D. Fosdick

18 January 1987
Revised
June 21, 1995

1 Introduction

For many years Fortran has been the language of choice in scientific computing, and, even though C has become increasingly popular, Fortran remains an important language in scientific computing. Indeed, there are Fortran compilers for every supercomputer, and new versions of it exist for vector computers and for parallel computers. For this reason, we use Fortran in most of the laboratory exercises and examples. However, we have found that many students enrolling in our course are unfamiliar with Fortran. For this reason we have found a review of Fortran to be quite useful and therefore have included it in this book.

The brief and elementary review in this tutorial describes enough of Fortran to enable you to read and understand the programs used in our laboratory exercises. Nevertheless, you will probably find a need for a more thorough description of Fortran. Two texts that you may find useful are *Fortran 77 for Humans* [Page 83] and *Effective Fortran 77* [Metcalf 85]. Besides these texts, you may find it useful to refer to the book which defines

[*]This work has been supported by the National Science Foundation under an Educational Infrastructure grant, CDA-9017953. It has been produced by the HPSC Group, Department of Computer Science, University of Colorado, Boulder, CO 80309. Please direct comments or queries to Elizabeth Jessup at this address or e-mail `jessup@cs.colorado.edu`.

standard Fortran 77: *X3.9-1977 Programming Language Fortran* which is available from the American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018.

Fortran 90 and HPF (**H**igh **P**erformance **F**ortran) are more recent versions of Fortran that include vector operations and other features for high performance computing. Descriptions of Fortran 90 can be found in *Programmers Guide to Fortran 90* [Brainerd et al. 90] and in the more complete *Fortran 90 Handbook* [Adams et al. 92]. A reference for HPF is *The High Performance Fortran Handbook* [Koelbel et al. 94].

This tutorial consists of four parts. Section 2 is a brief overview of Fortran. Section 3 contains basic definitions. Section 4 is a description of Fortran statements, organized alphabetically. Section 5 is a short description of the use of the `READ` and `WRITE` statements. Section 6 presents two sample programs.

2 Overview

2.1 Program structure

A Fortran program is composed of statements. Typically these statements are grouped into subprograms, also called procedures. One of the subprograms is the main program and its statements are executed first. Other subprograms are executed by procedure calls, as will be explained later. Small programs may consist of just one subprogram, the main program. An example follows.

```
PROGRAM HELLO
WRITE(*,*) 'HELLO WORLD'
END
```

2.2 Statements

There are two categories of statements: *executable* and *non-executable*. Executable statements specify operations that the computer must perform, or execute. An example of an executable statement is:

```
WRITE(*,*) 'HELLO WORLD'
```

When this statement is executed, the computer writes `HELLO WORLD` on the standard output device, normally your CRT display.

An example of a non-executable statement is:

```
      PROGRAM HELLO
```

This statement declares the statements following it, up to and including the `END` statement to be a main program having the name `HELLO`.

Usually a statement ends at the end of the line, but provision is made for long statements to continue onto additional lines. There is no special mark, such as a semicolon as in C, to denote the end of a statement.

2.3 Control statements

Control statements make it possible to have *loops*, sequences of statements that are executed over and over again, and *branches*, alternate sequences of statements to execute.

The control statements are the `DO` statement, the `IF` statement, and the `GOTO` statement, all of which are described in section 4.

2.4 Expressions

A fundamental component of most executable statements is the *expression*, for example

```
      X + 1.0
```

The meaning of this expression is exactly what you expect: it means the value of `X` plus `1.0`. Here `X` denotes a variable that has some value determined elsewhere, `+` denotes the arithmetic operator for addition, and `1.0` denotes itself, the numerical value one. As this small example illustrates, an expression is composed of operators and operands, the latter being either variables or constants. Besides arithmetic expressions the language also permits logical expressions and character expressions. Different kinds of expressions have different kinds, or *types*, of values. Arithmetic expressions have values that are numbers; logical expressions have only the values *true* or *false*; character expressions have values that are *characters* (i.e., the characters that you type on your keyboard as well as some you cannot) and sequences of characters, or *strings*.

The most common use of an expression is in an assignment statement; for example

```
Y = X + 1.0,
```

where the variable `Y` is given the value of the expression `X + 1.0`. Note that in Fortran the assignment operator is `=`, as it is in C.

2.5 Types

The word *type* refers to the kind of value a variable, constant, or expression has, or is allowed to have. Each variable used in a program has a fixed type, normally defined in a non-executable statement at the beginning of the program. For example, the statement

```
INTEGER A, B
```

declares the variables `A` and `B` to have the type `INTEGER`. This means that the only values `A` and `B` are allowed to have are integer values. The rules for evaluating expressions depend on the types of the operands and the kinds of operators used.

2.6 Compiling

The UNIX command for compiling a Fortran 77 program is `f77`. Three sample uses of this command follow:

```
(1)   f77  hello.f
(2)   f77  hello.f  -o hello
(3)   f77  -c  hello.f
```

The first example merely compiles the program in the file `hello.f`, generating the executable file named `a.out`. The second command also compiles `hello.f` but gives the executable file the name `hello`. The third example compiles the program, but produces only an object file, `hello.o`; it does not produce an executable file. The standard UNIX `f77` command does not produce a program listing. However, many vendors' Fortran compilers do, with a command option such as `-l` or `-list`.

3 Definitions and basic rules

3.1 Character set

The character set consists of the letters of the alphabet

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

and the digits

```
0 1 2 3 4 5 6 7 8 9
```

and the following special characters

```
= + - * / ( ) , . $ ' :
```

There are two important exceptions. `CHARACTER` constants are allowed to contain any character representable on your computer system; the same is true for comment lines.

The following type declarations are equivalent:

```
Integer a
```

and

```
INTEGER A
```

Note, in particular, that there is no distinction between the names `A` and `a` – they stand for the same variable. Fortran is not case sensitive, but, in tutorial, this we write Fortran statements using capital letters in order to make them stand out. (In the past, Fortran allowed only upper case letters, but few, if any, compilers have this restriction now.)

3.2 Names

Names are used to identify objects such as variables, subprograms, and constants. A name consists of one to six characters, the first of which must be a letter, the rest must be letters or digits. Examples are:

```
A  A1  A123  AB4XYZ
```

This atavism on the length of a name harks back to the days when the word length in most computers was 36 bits, and 6 bits represented a character. (A character is now a one-byte construct.) It is fast disappearing, and most Fortran compilers do not have this length restriction.

3.3 Types

The types in Fortran are:

```
        INTEGER,
        REAL,
        DOUBLE PRECISION,
        LOGICAL,
        CHARACTER,
    and CHARACTER*n
```

where `n` is an integer in the range (1, 2, ..., 127).

3.4 Labels

Labels are used to identify particular statements. They consist of one to five digits, and they must be in columns[1] 1-5 of the statement line. The leading digit may begin in any of these columns, but the last digit must not extend beyond column 5. An example of a labelled statement is:

```
   99  RETURN
```

Many Fortran compilers do not have this restriction on label position.

3.5 Keywords

These are words that have a special meaning in a program. Examples are

```
      DO
      REAL
      INTEGER
      COMMON
      FORMAT
      SUBROUTINE
```

and so forth. Although Fortran does not prohibit it, you should not give names to variables that are the same as keywords, otherwise you make the program hard to read.

[1]By Fortran tradition, the position of a character in a line of input or output is called the *column* where that character is located; this terminology goes back to the time when most input files and programs were kept on punched cards.

3.6 Variables

A variable that has not been assigned a value is said to be *undefined*. While most computing systems assign the value zero to all variables before execution of a program begins, it is unwise to assume that this is done. Once a variable is assigned a value it retains that value until a new value is assigned to it, usually by execution of an assignment statement.

We distinguish three kinds of variables: simple variables, array elements, and character strings. Simple variables are denoted with a name, for example

```
V1
SOLN
ROOT
```

Array elements are denoted with a name followed by *subscripts*, for example

```
A1(2,3)
MAP(J,K,L)
SCORE(100)
```

An array element denotes a value in an array of values, thus `A1(2,3)` denotes the value in row 2 and column 3 of the array `A1`. Character strings are denoted with a name, just like simple variables, or by a name followed by a pair of values inside parentheses, and separated by a colon, denoting the location of the first character and the location of the last character in a string. Thus `NAME(4:12)` denotes the string of characters starting at character position 4 and extending to position 12 (including position 12) within the character string `NAME`.

3.7 Arrays

An array is an n-dimensional object that in one-dimensional form is analogous to a vector and in two-dimensional form is analogous to a matrix. Thus a one-dimensional array is thought of as a sequence of values like:

```
3.45
1.0095
2.2299
```

and a two-dimensional array is thought of a set of values arranged in rows and columns like:

```
 3.45    1.004     0.998
-2.11    0.888    -0.333
```

The number of elements in each dimension is arbitrary, and up to seven dimensions are allowed.

An array has a name and a type. An element in an array is identified with *subscripts*. Thus if ARR is the name of the two-dimensional array above, the element ARR(2,1) has the value -2.11 and the element ARR(1,2) has the value 1.004. All of the elements in an array must have the same type.

The subscripts are, in general, INTEGER expressions. Thus we can write ARR(I, J), ARR(I-1, J+1) and so forth, assuming that the types of I and J are INTEGER.

An *array declarator* is used to declare the name and size of an array. It appears in type statements. Thus in the type statement

```
REAL A(10,20)
```

A(10,20) is an array declarator. It declares an array consisting of 10 rows and 20 columns that has the name A. As illustrated here an array declarator consists of a name (the name of the declared array) followed by a sequence of values separated by commas, and enclosed in parentheses, that specify the number of elements in each dimension. The numbering of elements in each dimension begins at 1, unlike C where it begins at 0. However, Fortran also allows arbitrary array bounds; for example

```
REAL A(-1:8,0:19)
```

declares a 10×20 array with row subscripts running $-1, 0, \ldots 8$ and column subscripts running $0, 1, \ldots 19$.

An array declarator like A(N, M) is allowed only if N and M are symbolic constants or if the declarator appears in a SUBROUTINE or FUNCTION and A, N and M are formal parameters.

3.8 Constants

There are two forms of constants, *symbolic* and *literal*. Symbolic constants have names and are defined with a `PARAMETER` statement. Literal constants represent themselves; e.g., `1.2`, `-6.9856`, `0.004`, etc.

A constant has a type: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `CHARACTER`, or `LOGICAL`.

- A literal `INTEGER` constant is written without a decimal point, thus `35`, `1999`, and `-456` are examples of literal `INTEGER` constants. A comma is not allowed: thus `1999` not `1,999`.

- A literal `REAL` constant may be expressed in ordinary decimal form, or in *floating-point* form. Examples of the first form are `3.1415927`, `-0.004475`, and `136.0084`. Examples of the second form (floating-point) are `0.31415927e1`, `-4.475e-3`, `0.1360084e3`. In floating-point form, the integer following the `e` denotes a power of `10` that is to multiply the number standing before the `e`. Thus `0.35e6` means `0.35` times `10` raised to the power `6`, so `0.35e6 = 350000.0`, and `0.35e-6 = 0.00000035`.

- A literal `DOUBLE PRECISION` constant is written like a `REAL` floating-point constant except that `d` is used in place of `e`. Thus

  ```
  0.31415926535897930d1
  ```

  is a `DOUBLE PRECISION` constant representing the mathematical constant π.

- A literal `CHARACTER` constant is written with apostrophes as delimiters. Thus `'C'`, `'CAT'`, `'Monkey'`, and `'pi equals 3.1415927'` are `CHARACTER` constants. A character constant has a length. The lengths of the four constants just given are 1, 3, 6, and 19. Note that the apostrophe is not part of the value of the constant but blanks appearing inside the apostrophes are. If an apostrophe must appear in a `CHARACTER` constant as in O'Malley it is expressed `'O''Malley'`; i.e., the embedded ' is written twice.

- A literal `LOGICAL` constant is written `.TRUE.` (for the value *true*) or `.FALSE.` (for the value *false*). Thus an assignment statement assigning the `LOGICAL` variable `L` the value *false* would be written:

```
        L = .FALSE.
```

3.9 Operators

The arithmetic operators are: + (addition); - (subtraction); * (multiplication); / (division); ** (exponentiation). The relational operators are: `.LT.` (less than); `.LE.` (less than or equal); `.EQ.` (equal); `.NE.` (not equal); `.GE.` (greater than or equal); `.GT.` (greater than). Relational operators are used with operands of type `INTEGER`, `REAL`, `DOUBLE PRECISION`, `CHARACTER`, or `CHARACTER*n`. The value of a relational expression has type `LOGICAL`. Thus the expression

```
X .LT. Y
```

has the value *true* or *false*.

The logical operators are: `.NOT.` (negation); `.AND.` (logical *and*); `.OR.` (logical *or*). Logical operators are used with operands of type `LOGICAL`; for example,

```
(X .LT. Y) .OR. (X .LT. Z)
```

3.10 Evaluation of expressions

The evaluation of expressions is done in the way we normally expect in mathematical work. Thus in the arithmetic expression

```
X + Y*Z
```

the multiplication is performed first then the addition. Parentheses are used to group subexpressions; for example,

```
(X + Y)*Z
```

where now the addition is performed first, then the multiplication. The arithmetic operators thus have an order of precedence. The operator with the highest order of precedence is performed first within any parenthesis-free subexpression; the operator with the lowest order of precedence is performed last. The order of precedence from lowest to highest is (+ -) (* /) **. The grouped operators have the same precedence level. In the expression

```
A + B*C**3
```

the evaluation proceeds as follows: raise `C` to the power `3`, multiply the result by `B`, and then add `A`. If two operations are at the same level, they are performed in left-to-right order, excepting exponentiation that is done right-to-left (`C**2**3` is equivalent to `C**(2**3) = C**8`).

The order of precedence for logical operators from lowest to highest is: `.OR. .AND. .NOT.` .

Relational operators all have the same level.

Across the various types of operators the order of precedence from lowest to highest is: logical, relational, arithmetic. Thus

```
X + Y .LT. Z .OR. A .LT. B
```

is equivalent to

```
((X + Y) .LT. Z) .OR. (A .LT. B)
```

The latter form is preferable since it makes the order of evaluation explicit.

3.11 Statements

Each statement is usually written on a single line but if the statement is too long to fit on a line it may be extended onto one or more (up to nineteen) lines. The statement must begin in column 7 of the line or to the right of column 7 and cannot extend beyond column 72 (another atavism, frequently ignored by compilers). Many Fortran systems accept a tab character at the front of the line as equivalent to 7 or more spaces; otherwise you must explicitly type at least 7 blanks at the front of the line. A long statement can extend to the next line, with the continuation line beginning at column 7 or to the right of column 7 and not extending beyond column 7. The continuation line has a non-blank character, except 0, in column 6 to identify it as a continuation line.

3.12 Statement order

The declarations, which are nonexecutable statements, precede the executable statements within a subprogram. The `PROGRAM`, `FUNCTION`, or `SUBROUTINE` statement must come first in the subprogram. The `IMPLICIT` statement, if it is used, should come next. Then type statements, `PARAMETER` statements, and `EXTERNAL` statements follow. It is important (See section 4 for more information on the meanings of these statements.) to note that the type of a symbolic constant must be declared in a type statement before it is given a value in a `PARAMETER` statement. Also, a symbolic constant should be given a value in a `PARAMETER` statement before it is used; for example,

```
      INTEGER N
      PARAMETER (N = 30)
      REAL A(N)
```

is the correct order for these three statements. For the sake of clarity it is a good idea to group declaration statements of the same kind together.

Executable statements (Assignment, `CALL`, `CONTINUE`, `DO`, `GOTO`, `IF`, `OPEN`, `READ`, `RETURN`, `STOP`, `WRITE`) follow the declarations. `FORMAT` statements may appear anywhere within the subprogram. For clarity, it is a good idea to group them in a single place, say just before `END`.

Fortran has implicit typing: names beginining with the letters I, J, K, L, M, and N are implicitly typed as `INTEGER`; names beginning with any other letter are implicitly typed `REAL`. This atavism, which may once have had some convenience, is a source of programming errors. Implicit typing can be turned off with the statement

```
      IMPLICIT NONE
```

which must precede any type declarations. With this statement in place, the type of every variable must be declared explicitly, as in C.

3.13 Comments

A comment line is identified by the letter `C` or an asterisk `*` in column 1 of the line.

3.14 Blanks

Strictly speaking blanks are ignored by the compiler, however it is unwise to put meaningless blanks in a program or to not use them when they could improve legibility. Thus

```
        RE AL X, Y  and   REALX,Y
```

are allowed but

```
        REAL X, Y
```

is clearer.

3.15 Intrinsic functions

Certain common functions like the square root are part of Fortran. They are called *intrinsic functions*. A list of some of these functions appears in table 1. The type `DOUBLE PRECISION` can replace `REAL` everywhere in this table.

4 Description of statements

In this section we systematically describe the statement types, proceeding in alphabetical order. First a few words about the notation we use in these descriptions.

4.1 Notation

To describe the syntax, or form, of statements we use a certain notation and conventions that are described below.

1. Special characters (except as noted below) and capitalized words appear in statements exactly as shown.
2. Lower case letters and words stand for objects defined elsewhere.
3. Square brackets are used to indicate optional items.
4. An ellipsis ... is used to denote one or more repetitions of an item.
5. Lower case words or phrases that appear in the syntax descriptions are in bold letters (e.g., `name`) when they appear in the running text.

Thus the syntax of a `CALL` statement is described by the expression:

Name	Definition	Type of parameter	Type of result
`ICHAR(C)`	Convert to integer	`CHARACTER`	`INTEGER`
`CHAR(K)`	Convert to character	`INTEGER`	`CHARACTER`
`ABS(X)`	Absolute magnitude	`INTEGER` or `REAL`	`INTEGER` or `REAL`
`MOD(J,K)`	Remainder of J/K	`INTEGER`	`INTEGER`
`SQRT(X)`	Square root	`REAL`	`REAL`
`EXP(X)`	Exponential function	`REAL`	`REAL`
`LOG(X)`	Natural logarithm	`REAL`	`REAL`
`SIN(X)`	Trig. sine function (X in radians)	`REAL`	`REAL`
`COS(X)`	Trig. cosine function (X in radians)	`REAL`	`REAL`
`TAN(X)`	Trig. tangent function (X in radians)	`REAL`	`REAL`
`ASIN(X)`	Trig. arcsine	`REAL`	`REAL`
`ACOS(X)`	Trig. arccosine	`REAL`	`REAL`
`ATAN(X)`	Trig. arctangent	`REAL`	`REAL`

Table 1: A partial list of Fortran intrinsic functions.

```
        CALL name [(parameter [, parameter]...)]
```

The form of `name` is described elsewhere (a letter followed by letters or digits, possibly with a maximum of six characters). The outermost pair of square brackets implies that this statement is valid:

```
        CALL MYSUB
```

The innermost pair of square brackets followed by the ellipsis imply that inside the parentheses there are one or more `parameter`s separated by commas. Thus the following are all valid:

```
        CALL SUB2(X, Y)
        CALL SUB3(X, 1.0)
        CALL SUB4('MYNAME', W**2, A(J))
```

4.2 Assignment statement

Syntax:

```
    variable = expression
```

Purpose: Assign the value of `expression` to `variable`.
Examples:

```
    (1)     X = Y + 3.2*Z
    (2)     C = 'A String'
    (3)     L = X .LT. Y
    (4)     U(K) = U(K)*EXP(SQRT(2.0/W))
```

Remarks:

1. The type of `expression` and the type of `variable` must be the same excepting between numeric types, where `REAL` and `INTEGER` can be paired, and between character types, where `CHARACTER` types of different length can be paired. Thus in example (1), `X` must be `REAL` or `INTEGER`; in example (2), `C` must be `CHARACTER` (of any length); in example (3), `L` must be `LOGICAL`; and in example (4), `U` must be `REAL` or `INTEGER`.

2. In the case

   ```
   integer_variable = real_expression
   ```

   the integer part of `real_expression` is assigned to `integer_variable`. Thus in the statement `K = -3.95` the value assigned `K` is `-3`.

3. In the case

   ```
   real_variable = integer_expression
   ```

   the integer part of `real_variable` is assigned the value of `integer_expression` and the fractional part of `real_variable` is assigned the value zero.

4. In the case

   ```
   character_variable = character_expression
   ```

   we may have the length of `character_variable` less than the length of `character_expression`. In this case, characters are chopped from the right end of `character_expression`. If the length of `character_variable`, is greater than the length of `character_expression`, the excess space on the right end of `character_variable` is filled with blanks.

In the above, `DOUBLE PRECISION` may replace `REAL`.

4.3 CALL statement

Syntax:

```
CALL name [(parameter [, parameter]...)]
```

Purpose: Execute the subroutine `name`.
Examples:

```
(1)     CALL SUB1
(2)     CALL SUB2(X, Y)
(3)     CALL SUB3(X, 1.0)
(4)     CALL SUB4('MYNAME', W**2, A(J))
```

Remarks:

1. A `parameter` may be any of the following: `variable`, `expression`, `subroutine_name`, `function_name`, `array_name`.

2. If `parameter` is `subroutine_name` or `function_name` then an `EXTERNAL` statement must declare the `subroutine_name` or `function_name`. The `EXTERNAL` statement must be located in the same program unit as the `CALL` statement.

3. The `parameter`s must agree in number and type with the `parameter`s in the corresponding `SUBROUTINE` statement; i.e., the k-th `parameter` in each list must have the same type and each list must have the same number of `parameter`s.

4. The `parameter`s appearing here are called `actual` parameters to distinguish them from the `formal` parameters appearing in the corresponding `SUBROUTINE` statement.

4.4 `COMMON` statement

Syntax:

```
COMMON [/name/] common_item [, common_item]...
```

Purpose: Share data between program units.
Examples:

```
(1)     COMMON X, Y
(2)     COMMON /PARAMS/ A, B, C
```

Remarks:

1. `common_item` may be a simple variable or an array name.

2. If the `name` part is absent as in example (1), the statement is called a *blank* `COMMON` statement, otherwise it is called a *labelled* `COMMON` statement.

3. If two program units have labelled `COMMON` statements with the same `name` then `common_items` in corresponding positions refer to the same data regardless of whether or not they have the same name. The two `COMMON` statements should have the same number of `common_items` and corresponding `common_items` should have the same type.

4. If two program units have blank common statements then `common_items` in corresponding positions refer to the same data regardless of whether or not they have the same name. The two `COMMON` statements should have the same number of `common_items` and corresponding `common_items` should have the same type.

5. If one `COMMON` statement follows another in the same program unit and both have the same `name` or both are blank `COMMON` statements then the lists of `common_items` are concatenated; e.g.,

   ```
   COMMON /CPARMS/ X, Y
   COMMON /CPARMS/ Z
   ```

   is equivalent to

   ```
   COMMON /PARMS/ X, Y, Z
   ```

   The latter form is preferred because it is clearer.

4.5 CONTINUE statement

Syntax:

```
CONTINUE
```

Purpose: This is a null statement, it doesn't do any computation.
Examples:

```
(1)    10   CONTINUE
```

Remarks:

1. This statement is often used with a label as the last statement in a `DO` loop or as the target of a `GOTO` statement. Example (1) shows a `CONTINUE` statement with a label of 10.

4.6 `DO` statement

Syntax:

```
DO  do_variable = expressn_1, expressn_2 [,expressn_3]
```

Purpose: Controls repeated execution of a sequence of statements.
Examples:

```
(1)        DO J = 1, 20
             X(J) = 0
          END DO


(2)        DO  K = 0, N, 2
             WRITE(*,*) K, K**2, K**3
           END DO

(3)        DO  J = 1, 100
             P(J) = 1
             DO  K = 1, N
                 P(J) = P(J) + SQRT(J*K)
             END DO
           END DO
```

Remarks:

1. The `DO` statement causes the sequence of statements following the `DO` statement, up to the matching `END DO`, to be executed repetitively. This sequence of statements is called the `range` of the `DO`. If `expressn_3` is absent it is assumed to have the value 1. Initially, `do_variable` is assigned the value of `expressn_1`; and the iteration count is given the value

$$\max(\lfloor \texttt{(expressn\_2 - expressn\_1 + expressn\_3)/ expressn\_3} \rfloor, 0)$$

If the iteration count is not zero the range is executed. After each execution of the range the `do_variable` is incremented by `expressn_3`, and the iteration count is decremented by 1; then the range is executed again and this continues until the iteration count reaches 0, at which point the iteration is terminated.

2. `do_variable` is a simple variable of type `INTEGER`, and the types of `expressn_1`, `_2`, `_3` are `INTEGER`.

3. The effect of example (1) is to set the values of `X(1)`, `X(2)`, ..., `X(20)` equal to zero.

4. The effect of example (2) is to write, on successive lines, the values: 0 0 0; 2 4 8; 4 16 64; .... The last line has the values `N`, `N**2`, and `N**3` if `N` is even; otherwise it has the values `(N-1)`, `(N-1)**2`, and `(N-1)**3`.

5. The effect of example (3) is to evaluate expressions:

```
1 + SQRT(1) + SQRT(1*2) + ... + SQRT(1*N);
1 + SQRT(2*1) + SQRT(2*2) + ... + SQRT(2*N);
     ... ;
1 + SQRT(100*1) + SQRT(100*2) + ... + SQRT(100*N).
```

   These values are assigned to `P(1)`, `P(2)`, ..., `P(100)`, respectively. Example (3) illustrates that one `DO` can be contained in the range of another `DO`. This so-called nesting of `DO` statements can be arbitrarily deep.

6. No statement in the range is permitted to change the value of `do_variable`, `expressn_1`, `expressn_2`, `expressn_3`.

7. Execution of the range must begin with executing the `DO`; that is, execution of a statement in the range by jumping to it from outside the range, using a `GOTO`, is forbidden. On the other hand, it is permitted to jump out of the range using a `GOTO`.

4.7 `END` statement

Syntax:

```
END
```

Purpose: Marks the end of a program unit.
Examples:

```
(1)    END
```

Remarks:

1. Every program unit must have this statement as its last statement.

4.8 `EXTERNAL` statement

Syntax:

```
EXTERNAL name [, name]...
```

Purpose: Declares names of `FUNCTION` and `SUBROUTINE` subprograms that are passed as parameters in calls to subprograms.
Examples:

```
(1)    EXTERNAL MYFUNC, MYSUB
```

Remarks:

1. `name` is the name of a `FUNCTION` or `SUBROUTINE` appearing as an actual argument in a call to a subprogram.

2. Every `FUNCTION` and `SUBROUTINE` name used as an actual parameter in a call to a subprogram must appear in an `EXTERNAL` statement in the program unit in which it is so used.

4.9 FORMAT statement

Syntax:

```
FORMAT (edit_descriptor [, edit_descriptor]...)
```

Purpose: Defines the input or output format (number of columns used, floating-point form, etc.) of values of `iolist` items. (cf., `READ` and `WRITE` statements.)

Examples:

```
(1)     99  FORMAT(1X, I10)
(2)     98  FORMAT(I10, 5X, E15.8, 5X, F10.2)
(3)     97  FORMAT(A, 3(2X, I5))
(4)     96  FORMAT(A, 2X, I5, 2X, I5, 2X, I5)
```

Remarks:

1. A repeatable `edit_descriptor` is one of: `Iw`, `Fw.d`, `Ew.d`, `A`. These edit descriptors are associated with `iolist` items: `I` with items of type `INTEGER`, `F` and `E` with items of type `REAL`, A with items of type character. `F` is used to specify conventional decimal format (e.g., `0.003956`), `E` is used to specify floating-point format (e.g., `0.3956e-02`). The lowercase letters `w` and `d` denote unsigned integers: `w` specifies the width, number of columns, occupied by the item; `d` specifies the number of digits after the decimal point. The width of an item associated with `A` is the length of the `CHARACTER` type. The value of `d` is ignored when reading `REAL` values; it is only meaningful when writing.

2. A repeatable `edit_descriptor` may be preceded by an unsigned integer (viz. `3I10`) denoting multiple descriptors. Thus `3I10` and `I10, I10, I10` are equivalent.

3. A nonrepeatable `edit_descriptor` is one of: `nX / nP`. These descriptors are not associated with `iolist` items. `X` denotes a blank, `/` denotes end of line, `P` denotes a scale factor. The edit descriptor `3PE15.7` prints a floating-point value with 3 places before the decimal point. For example, writing the value `-0.01255` with the edit descriptor `E15.7` produces

```
-0.1255000E-01
```

Using the edit descriptor `3PE15.7` results in

```
        -125.5000000E-04
```

4. The `FORMAT` statements in examples (3) and (4) are equivalent, example (3) being a more compact form of example (4).

4.10 `FUNCTION` statement

Syntax:

```
[type] FUNCTION name ([parameter [, parameter]...])
```

Purpose: Declares a subprogram to be a `FUNCTION` subprogram. It is the first statement in the subprogram.

Examples:

```
(1)     FUNCTION FUN1(X)
(2)     REAL FUNCTION FUN2(X1, X2)
(3)     CHARACTER*8 FUNCTION FUN3(CHR1, CHR2, XYZ)
```

4.11 `GOTO` statement

Syntax:

```
    GOTO label
```

Purpose: Jump to the statement labelled `label` and resume executing statements there.

Examples:

```
    GOTO 50
```

Remarks:

1. `GOTO` statements should be used with care. Indiscriminate use of these statements results in programs with tangled control paths that are hard to understand.

4.12 IF statement

Syntax:

```
IF (logical_expression) THEN
    [statement]...
ENDIF
```

or

```
IF (logical_expression) THEN
    [statement]...
ELSE
    [statement]...
ENDIF
```

or

```
IF (logical_expression) THEN
    [statement]...
ELSEIF (logical_expression) THEN
    [statement]...
[ELSEIF (logical_expression) THEN
    [statement]...]
ELSE
    [statement]...
ENDIF
```

Purpose: Allows conditional execution of a sequence of statements.

Examples:

```
(1)   IF (X .LT. Y) THEN
          WRITE(*,*) 'X IS LESS THAN Y'
      ENDIF

(2)   IF (A(J) .GT. A(J+1)) THEN
          T = A(J)
          A(J) = A(J+1)
          A(J+1) = T
      ENDIF
```

```
(3)    IF (X .LT. Y) THEN
           WRITE(*,*) 'X IS LESS THAN Y'
       ELSE
           WRITE(*,*) 'X IS GREATER THAN OR EQUAL TO Y'
       ENDIF

(4)    IF (ABS(X-Y) .LE. ABS(X)*EPS) THEN
           GOTO 20
       ELSE
           Y = X
           J = J + 1
       ENDIF

(5)    IF (C .EQ. 'A') THEN
           CALL SUBA(X)
           A(1) = A(1) + 1
       ELSEIF (C .EQ. 'B') THEN
           CALL SUBB(X)
           A(2) = A(2) + 1
       ELSEIF (C .EQ. 'C') THEN
           CALL SUBC(X)
           A(3) = A(3) + 1
       ELSE
           CALL ERROR(X)
           A(4) = A(4) + 1
       ENDIF
```

Remarks:

1. Execution of this statement proceeds as follows. If the value of `logical_expression` is true then the sequence of statements following `THEN` and before `ELSE`, `ELSEIF`, or `ENDIF` (whichever appears first) is executed. If the value of `logical_expression` is false then the statements following `THEN` and before `ELSE`, `ELSEIF`, or `ENDIF` (whichever appears first) are skipped and:

- if `ELSE` comes first then the sequence of statements following `ELSE` is executed;
- if `ELSEIF` comes first then, if the associated `logical_expression` is true, the statements following `THEN` are executed; otherwise they are skipped.
- if `ENDIF` comes first then the statements immediately following `ENDIF` are executed.

2. In example (1), the `WRITE` statement is executed if and only if the value of `X` is less than the value of `Y`.
3. In example (2), the sequence of three statements in the body of the `IF` is executed if and only if the value of `A(J)` is greater than the value of `A(J+1)`.
4. In example (3), the message `X IS LESS THAN Y` is written if and only if the value of `X` is less than the value of `Y`; otherwise the message `X IS GREATER THAN OR EQUAL TO Y` is written.
5. In example (4), the `GOTO` is executed if and only if the value of the absolute magnitude of `(X-Y)` is less than or equal to the product of the absolute magnitude of `X` and the value of `EPS`.
6. In example (5), the subroutine `ERROR` is called if and only if the value of `C` is not equal to '`A`', or to '`B`', or to '`C`'.
7. Every `IF` must be terminated with an `ENDIF`.

4.13 IMPLICIT statement

Syntax:

```
IMPLICIT type(range [, range]...) [, type(range [, range]...]
```

Purpose: To associate a type with all names starting with a particular letter, or range of letters, excepting names of intrinsic functions.

Examples:

```
(1)     IMPLICIT CHARACTER*32 (C), REAL (I-L, N)
(2)     IMPLICIT NONE
```

Remarks:

1. `type` is a Fortran type (`REAL`, `LOGICAL`, etc.).
2. `range` is a single letter or a pair of letters separated by a dash.
3. Example (1) declares all variables with names starting with the letter `C` to have the type `CHARACTER*32` and all variables with names starting with the letters `I`, `J`, `K`, `L`, `N` to have the type `REAL`.
4. Example (2) declares no variables to have an implicit type; that is, all variables must be explicitly typed.
5. If a name is explicitly typed as in

   ```
   REAL CENTER
   ```

   then this type declaration overides the effect of an implicit type declaration. Thus in a program unit containing the declaration line shown in example (1) and this `REAL` declaration, `CENTER` would have they type `REAL` but `COURSE` would have the type `CHARACTER*32`.
6. The scope of this statement is the program unit in which it appears.
7. Programming errors associated with wrong types can be more easily detected by using the `IMPLICIT` statement shown in example (2) in each program unit and explicitly declaring the types of all variables.

4.14 OPEN statement

Syntax:

```
OPEN (unit_spec [, FILE = 'file_name'] [, STATUS = 'status'])
```

Purpose: Associates a file with a unit number used in a `READ` or `WRITE` statement.

Examples:

```
(1)    OPEN (3, FILE = 'MYDATA', STATUS = 'OLD')
(2)    OPEN (UNIT = 7, FILE = 'MYOUT', STATUS = 'NEW')
(3)    OPEN (8)
```

Remarks:

1. `unit_spec` is an integer, a unit number alone, or a unit number preceded by `UNIT =` (cf., example (2)).

2. `file_name` is the name of the file that is to be associated with the unit number given in `unit_spec`. Thus, referring to example (1), if a subsequent `READ` statement had the form

   ```
   READ(3, 99) X
   ```

   then the value of `X` would be read from the file named `MYDATA`. In this situation, case is important; that is, the file name is `MYDATA`, not `mydata`.

3. `status` is `NEW`, `OLD`, `SCRATCH`, and `UNKNOWN`. `NEW` is used for files that are to be created, `OLD` is for files that already exist. Thus, the statement in example (2) might be used in conjunction with a `WRITE` statement of the form

   ```
   WRITE(7, 98) RESULT
   ```

   to write the value of RESULT on the new file `MYOUT`, but it could not be used in conjunction with a `READ` statement of the form

   ```
   READ(7, 98) VALUE
   ```

   which presupposes the existence of the file `MYOUT`. `SCRATCH` is used for files that are only temporary, for example to save some data during a computation; they are removed when program execution is terminated, or when the file is closed. `UNKNOWN` is processor dependent; in some systems (e.g., DEC Fortran) the system tries `OLD` and if it cannot find the file it creates a `NEW` file. The default status is `OLD`.

4. The form used in example (3) is for creating a scratch file to hold intermediate results during a computation. It is destroyed when the program stops.

4.15 PARAMETER statement

Syntax:

```
PARAMETER (name = expression [, name = expression]...)
```

Purpose: Gives a name to a constant. Thus it allows you to use `PI` instead of 3.141592653 and to be protected against accidentally changing `PI`.

Examples:

```
(1)     PARAMETER (PI = 3.141592653, ALPHA = 'ABC')
(2)     PARAMETER (MAXVAL = 100, MINVAL = MAXVAL-50)
```

Remarks:

1. The type of `name` must agree with the type of `expression`.

2. A name used in a `PARAMETER` statement cannot have its value changed during execution of the program. An attempt to change it normally causes an error message.

3. `expression` is either a constant, or an expression containing constants. In the latter case the expression must be of type `INTEGER`.

4. Previously named constants can be used in `expression` (cf., example (2)).

5. It is recommended that named constants be used rather than literal constants. Programming errors are likely to be reduced in cases where the constant is used more than once (using a name assures that the same value is used everywhere). Also, it is easy to modify the value of the constant in editing the program because its value appears only in the `PARAMETER` statement (cf., type statement, remark 9).

6. Any variable used in a `PARAMETER` statement must have had its type previously declared.

4.16 PROGRAM statement

Syntax:

```
PROGRAM name
```

Purpose: Gives the name `name` to the main subprogram.
Examples:

```
(1)     PROGRAM MYPROG
```

Remarks:

1. This statement is optional. If it is used then it must be the first statement of the main subprogram, and in this case, `name` is the name of the main subprogram. If it is not used, then the main subprogram has the default name `MAIN`.
2. It is recommended that this statement be used to improve program readability.

4.17 READ statement

Syntax:

```
READ (unit, format [, END = label]) iolist
```

Purpose: Reads data from a file or the keyboard.
Examples:

```
(1)     READ (5, 99) X
(2)     READ (5, 99) X1, X2, C(I)
(3)     READ (*, 99) X
(4)     READ (*, *) X
(5)     READ (7, *) W1, A(K)
(6)     READ (7, *) K, (A(J), J = 1, 5), Y
(7)     READ (7, '(A)') C
(8)     READ (9, *, END = 100) K, A(I,J)
```

Remarks:

1. unit is an integer, greater than zero, that has been defined as the unit number for a file by an OPEN statement. Thus, given that the OPEN statement

   ```
   OPEN(5, FILE = 'MYDATA', STATUS = 'OLD')
   ```

   has already been executed, the effect of the statement in example (1) is to read one value from the file MYDATA and assign it to X. Similarly, the effect of the statement in example (2) is to read three values from MYDATA and assign them to X1, X2, and C(I), respectively.

2. unit may also be *. In this case, the data is read from the keyboard. Thus the effect of the statement in example (3) is to read one value entered from the keyboard and assign it to X. When reading from the keyboard, execution of the READ statement is not completed until the return key has been depressed: depressing this key signals the end of the line.

3. format is a label on a format statement or it is *. Examples (1)–(3) illustrate the first alternative; examples (4)–(6) illustrate the second alternative. In the second alternative, a default format specifier is used for each item in the iolist consistent with the type of the item; the phrase list directed input is used to describe the style of input determined by using this alternative.

4. format may also be a character constant as illustrated in example (7). In this case, the string of characters is treated just as if it appeared in a FORMAT statement. The effect of executing the statement in example (7) is the same as the effect of executing

   ```
          READ(7, 88) C
   88     FORMAT (A)
   ```

5. label denotes a labelled statement in the same program unit as the READ statement. If the READ statement is executed and there is no more data on the file, i.e., the end of the file has been reached, then the effect of END = label is to cause a GOTO label. Thus the effect of executing the following when all data has been read from unit 9 is to GOTO the statement X = 0.

```
            READ (9, *, END = 100) K, A(I,J)
            ...
      100   X = 0
```

6. `iolist` is a list of one or more variables separated by commas; these variables are to be assigned the values read. An `iolist` element can be an implied `DO`, as illustrated in example (6). The effect of the statement in this example is the same as the effect of

```
            READ (7, *) K, A(1), A(2), A(3), A(4), A(5), Y
```

7. There are many other options for the `READ` statement. Consult your computer manual for further information.

4.18 `RETURN` statement

Syntax:

```
      RETURN
```

Purpose: Return control to the calling subprogram from a subprogram.

Examples:

```
  (1)      RETURN
```

Remarks:

1. This statement is used only in `SUBROUTINE` or `FUNCTION` subprograms. When it is executed, it causes execution of the subprogram to stop and returns control to the program unit that called the subprogram.

2. If the control path in a subprogram reaches an `END` statement, the `END` has the same effect as `RETURN`.

4.19 `STOP` statement

Syntax:

```
        STOP
```

Purpose: Stops program execution and prints an optional message on the screen.

Examples:

```
  (1)     STOP
```

Remarks:

1. When this statement is executed, the program stops.

4.20 `SUBROUTINE` statement

Syntax:

```
        SUBROUTINE name [(parameter [, parameter]...)]
```

Purpose: Declares a subprogram to be a `SUBROUTINE`. This statement must be the first statement in the subprogram.

Examples:

```
  (1)     SUBROUTINE MYSUB
  (2)     SUBROUTINE MYSUB1(X, Y)
```

Remarks:

1. `parameter` is a `name` and `name` may identify a simple variable, an array, a `FUNCTION` subprogram, or a `SUBROUTINE` subprogram.

2. When the subroutine is called, the actual parameters must match the parameters in the `SUBROUTINE` statement (called the `formal parameters`) in number and type (cf., `CALL` statement).

4.21 Type statements

Syntax:

type `var [, var]...`

Purpose: Defines the type of a variable.
Examples:

```
 (1)     INTEGER X1, X2, K
 (2)     REAL ROOT, SOLN
 (3)     CHARACTER C1, C2, LET
 (4)     CHARACTER*20 MESSG1, MESSG2
 (5)     INTEGER X1, X2(50)
 (6)     REAL ROOTS(4), A(10, 20)
 (7)     CHARACTER C1(10)
 (8)     CHARACTER*80 LINES(50)
 (9)     LOGICAL TEST, BOOL(10)
(10)     REAL A(N,*), Z
(11)     DOUBLE PRECISION HPSOLN, Z
```

Remarks:

1. *type* may be `REAL`, `DOUBLE PRECISION`, `INTEGER`, `LOGICAL`, `CHARACTER`, or `CHARACTER*n` where `n` is an integer in the range (1, 2, ..., 127).

2. `var` may be a `name` or an array declarator.

3. All items in the `var` list have the specified type. Thus, in example (1), `X1`, `X2`, and `K` are defined to have the type `INTEGER`.

4. In example (4), `MESSG1` and `MESSG2` are defined to be character strings of length 20.

5. The type `DOUBLE PRECISION` is used to declare variables whose values are about twice as accurate as those declared `REAL`. Eight bytes are used to store a value of type `DOUBLE PRECISION`; four bytes are used to store a value of type `REAL`.

6. In example (6), `ROOTS` is defined to be a one-dimensional array of 4 elements, each element being of type `REAL`; and `A` is defined to be a two-dimensional array consisting of 10 rows and 20 columns having elements of type `REAL`.

7. In example (8), `LINES` is defined to be a one-dimensional array of character strings each of length 80.

8. The declaration in example (10) would appear in a `SUBROUTINE` or `FUNCTION` subprogram. It declares `A` to be an array of `N` rows and an indefinite number of columns; `N` would have to appear as a formal parameter in the `SUBROUTINE` or `FUNCTION` statement.

9. The sequence

   ```
   PARAMETER (N = 10, M = 20)
   REAL A(N, M)
   ```

   is equivalent to

   ```
   REAL A(10, 20)
   ```

   The former is longer but is preferred when the row and column dimensions are frequently used in the program text because modification of `N` and `M` is simpler – they only have to be changed in the `PARAMETER` statement, not every place 10 and 20 appear in the program text.

4.22 `WRITE` statement

Syntax:

```
WRITE (unit, format) iolist
```

Purpose: To write values on a file or the screen.
Examples:

```
(1)     WRITE (5, 99) X
(2)     WRITE (5, 99) X1, X2, C(I)
(3)     WRITE (*, 99) X
```

```
(4)     WRITE (*, *) X
(5)     WRITE (7, *) W1, A(K)
(6)     WRITE (7, *) K, (A(J), J = 1, 5), Y
(7)     WRITE (7, '(A)') C
(8)     WRITE (9, *) K, A(I,J)
```

Remarks:

1. `unit` is as defined for the `READ` statement, except that `*` denotes the screen. Thus in example (3), the value of `X` is written on the screen.

2. `format` is as defined for the `READ` statement.

3. `iolist` is as defined for the `READ` statement except the items listed have their values written on the designated unit or screen.

5 Reading and writing

5.1 Reading

List directed input (cf., `READ` statement, Remark 3) should be used for reading data. The other alternative, `FORMAT` directed input, is troublesome and likely to result in errors. The remaining discussion in this section is concerned with list directed input.

When the `READ` statement is executed, the data on one line of the input file is read. The correspondence between the data on the line and items in the `iolist` is left-to-right. For example, assume that the next line to be read from the input file is

```
'My Data is' 3.96 5
```

and that the following `READ` statement is executed:

```
READ(5,*) MESSG, X1, K1
```

The effect is that the string constant `'My Data is'` (without the quotes of course) is assigned to `MESSG` (which must be of type `CHARACTER`); the value 3.96 is assigned to `X1` (which must be of type `REAL`); the integer value 5 is assigned to `K1` (which must be of type `INTEGER`).

As the example above illustrates, a blank is used to separate the data on the input line. One or more blanks may be used, thus

```
'My Data is'      3.96   5
```

would give the same result when read by the above `READ` statement.

The number of items read from the input line is just the number of items in the `iolist`. Extra items are ignored. Thus if we read the two lines

```
12.1  -13.2E-5  44.0
-3.1  445.2     99.9
```

using the two statements

```
READ(5,*) X1, X2
READ(5,*) Y1, Y2, Y3
```

then the values assigned are as follows:

```
X1 = 12.1,  X2 = -13.2E-5
Y1 = -3.1, Y2 = 445.2, Y3 = 99.9
```

Note that the value 44.0 on the first line is not read.

If there are more items in the `iolist` than on the line of data, the excess items are not be assigned values. Thus if we read

```
12.1  -13.2E-5 44.0
```

with the statement

```
READ(5,*) X1, X2, X3, X4
```

then the values assigned are as follows:

```
X1 = 12.1, X2 = -13.2E-5, X3 = 44.0
```

and the value of `X4` is unchanged.

When a character constant is read and the length of the constant differs from the length of the `CHARACTER` type of the item in the `iolist`, the rules are like those for the assignment statement (cf., Assignment statement, Remark 4); i.e., characters are chopped from the right end of the constant if it is too long, and blanks are filled in on the right if it is too short.

5.2 Writing

While list directed output can be used with the `WRITE` statement, `FORMAT` directed output is the preferred mode since it allows more control over the form of the output. With list directed output, the format is the default format provided by the system. This is usually adequate for a quick look at the results, but not adequate for nice presentations of results in tables. The rest of the discussion is concerned with `FORMAT` directed output.

A `WRITE` statement normally writes one line of output. The connection between the iolist and the list of format edit descriptors can be described as follows. Assume we have a pair of statements of the form:

```
      WRITE(5, 99) iolist
 99   FORMAT(descriptors)
```

For each item in `iolist` there must be a corresponding edit descriptor in `descriptors`. The correspondence is left-to-right. The beginning of execution of the `WRITE` statement initiates `format control`. Format control proceeds from left to right through the `descriptors`. Each action of format control depends on the next edit descriptor and the next item in the `iolist`. If format control encounters a nonrepeatable edit descriptor it performs the action specified by that descriptor. If it encounters a repeatable edit descriptor it writes, in the output file, the value of the corresponding item in `iolist`.

To illustrate this, consider the pair of statements

```
      WRITE(6,99) X1, X2, C, K1, 'the end'
 99   FORMAT(1X, E15.7, 2X, F15.7, 2X, A, 2X, I5, 2X, A)
```

The output line consists of a blank in column 1, the value of `X1` (assumed `REAL`) in columns 2-16, blanks in columns 17 and 18, the value of `X2` (assumed `REAL`) in columns 19-33, blanks in columns 34 and 35, the value of C (assumed `CHARACTER*10`) in columns 35-45, blanks in columns 46 and 47, the value of `K1` in columns 48-52, blanks in columns 53 and 54, and the string constant 'the end' (without the quotes) in columns 55-61. The value of `X1` is written in floating-point form with seven digits after the decimal point (e.g., –.1234567e+02), the value of `X2` is written in ordinary decimal form, i.e., *fixed point* form with seven digits after the decimal point (e.g., –12.3456789); the value of `K1` is written in integer form (e.g., 29). If the number of characters required to express the value is less than the field width, as is the case for X1, X2, and K1, the left end is padded with blanks to fill out the field; thus the value is always *right-justified* (i.e., moved as far to the right as possible) in the field.

WARNING: When writing numbers that are small or large compared with 1 you should use the E format descriptor. Inexperienced programmers often use F instead. If it is used to write small numbers, the number written may be 0 even though the actual value is not zero but simply too small to show up with this format descriptor. For example, if you try to write the value 10^{-12} with the format specification F10.7 the number printed will be zero. With E12.4 the correct value will be printed. If the number you try to print is too large then the result is system dependent. Sometimes a block of asterisks is printed.

6 Examples

We close this with two short example programs. The first of these, CIRCLE, reads from a file the radii of circles and computes their circumference and area. This program is shown in figure 1. It illustrates opening, reading and writing files, and symbolic constants. The second, SORT, illustrates the use of control statements. It is shown in figure 2. Note the backward counting in the DO. It also illustrates the use of an array, and also how to declare an array that starts with row index 0, rather than the default value 1. Finally, it shows the value of a parameter statment: if you need to sort more numbers, you need only reassign the value of the parameter NX. For more examples, look in the textbooks already referenced.

```
      PROGRAM CIRCLE
* This program reads a number, the radius of a circle,
* from the file 'circle.dat', computes the area and
* circumference of the circle, and writes the results on
* the file 'circle.out'. It repeats these steps until
* all of the numbers on the file 'circle.in' have been
* read.
      IMPLICIT NONE
      REAL RADIUS, CIRCUM, AREA, PI
      PARAMETER(PI = 3.1415927)
* Open input file.
      OPEN(UNIT=7, FILE='circle.in', STATUS='OLD')
* Open output file.
      OPEN(UNIT=8, FILE='circle.out', STATUS='NEW')
* Write headers on output file.
      WRITE(8,*) 'Program CIRCLE output'
      WRITE(8,99)'RADIUS', 'CIRCUM', 'AREA'
* Begin main loop.
 10   CONTINUE
         READ(7,*,END=100) RADIUS
         CIRCUM = 2*PI*RADIUS
         AREA = PI*RADIUS*RADIUS
         WRITE(8,98) RADIUS, CIRCUM, AREA
         GOTO 10
* End main loop, write completion message to stdout and stop.
 100  WRITE(*,*) 'Program CIRCLE done.'
      STOP
  98  FORMAT(3(1PE15.7,1X))
  99  FORMAT(9X,A,10X,A,12X,A)
      END
```

Figure 1: The CIRCLE program. A simple example showing the reading and writing of files.

```
       PROGRAM SORT
* This program reads a list of integers and sorts them
* using a simple insertion sort algorithm. The maximum
* number of integers allowed is 20.
      IMPLICIT NONE
      INTEGER NX
      PARAMETER(NX = 20)
      INTEGER I, J, LAST, N(0:NX)
* Open input file.
      OPEN(UNIT=7, FILE='sort.in', STATUS='OLD')
* Open output file.
      OPEN(UNIT=8, FILE='sort.out', STATUS='NEW')
* Write header on output file.
      WRITE(8,*) 'Program SORT output'
* Read the input file and sort on the fly.
      READ(7,*) N(1)
      DO I = 2,NX
         READ(7,*,END=20) N(0)
 LAST = I
 DO J = I, 1, -1
    IF(N(0) .LT. N(J-1)) THEN
       N(J) = N(J-1)
            ELSE
       N(J) = N(0)
       GOTO 10
            ENDIF
         ENDDO
 10      CONTINUE
      ENDDO
* End reading input file and sorting.
 20   CONTINUE
      WRITE(8,99) (N(I), I=1,LAST)
* End main loop, write completion message to stdout and stop.
       WRITE(*,*) 'Program SORT done.'
       STOP
  99   FORMAT(I10)
       END
```

Figure 2: The SORT program. An example of an insertion sort program showing the use of control statements.

References

[Adams et al. 92] ADAMS, JEANNE C., WALTER S. BRAINERD, JEANNE T. MARTIN, BRIAN T. SMITH, AND JERROLD L. WAGENER. [1992]. *Fortran 90 Handbook.* Intertext Publications. McGraw-Hill Book Company, New York, NY.

[Brainerd et al. 90] BRAINERD, WALTER S., CHARLES GOLDBERG, AND JEANNE C. ADAMS. [1990]. *Programmers Guide to Fortran 90.* McGraw-Hill Book Company, New York, NY.

[Koelbel et al. 94] KOELBEL, CHARLES H., DAVID B. LOVEMAN, ROBERT S. SCHREIBER, JR. GUY L. STEELE, AND MARY E. ZOSEL. [1994]. *The High Performance Fortran Handbook.* Scientific and Engineering Computation. MIT Press, Cambridge, MA.

[Metcalf 85] METCALF, MICHAEL, editor. [1985]. *Effective FORTRAN 77.* Oxford University Press, Oxford.

[Page 83] PAGE, REX L., editor. [1983]. *FORTRAN 77 for Humans.* West Publishing Co., St. Paul, MN, 2nd edition.