A Summary of Fortran

Nelson H. F. Beebe
Center for Scientific Computing
Department of Mathematics
University of Utah
Salt Lake City, UT 84112
USA
Tel: (801) 581-5254
FAX: (801) 581-4148
E-mail: <beebe@math.utah.edu>

15 October 1992
Version 1.10

Abstract

This document provides a concise summary of the syntax of Fortran that may be a helpful supplement to a Fortran textbook. Each statement in the language is discussed in separate subsections, and recommended programming practices are emphasized.

Contents

1 A short history of Fortran

The first design of the Fortran (FORmula TRANslating system) language was begun in 1954 by a research group at IBM, under the direction of John Backus, following a December 1953 proposal for the project that Backus had sent to IBM management.

Initial estimates that a compiler could be ready in a few months proved overly optimistic. Although work on the compiler began in early 1955, the first Programmer's Reference Manual was not issued until October 15, 1956, and the first Fortran compiler became available to customers in April 1957, running on the IBM 704, a 36-bit machine.

Fortran was the first high-level language to survive, and is in widespread use today throughout the world.

By the early 1960s, several other vendors had developed Fortran compilers, each supporting a slightly different language. The undesirability of language dialects led to the formation of a standardization committee, X3J3, under the auspices of the American National Standards Institute. The first Fortran language standard was released in 1966 [1], with subsequent clarifications [9, 10]. Fortran was the first language to be standardized; since then, Cobol, MUMPS, Pascal, PL/1, C, Ada, Modula-2, and others have been defined by national, or international, standards.

Deficiencies of Fortran 66 led to a revision of the Standard [11], and the final version was released in April 1978 [2]; the language defined there is called Fortran 77, and is the language level supported by virtually every Fortran compiler available today. Most vendors had Fortran 77 compilers by the early 1980s, but the world's largest computer vendor, IBM, did not have one until about 1985.

Standards work has continued [4], and in the fall of 1990, the first draft of an international standard for what may be called Fortran 90 was released [6]. This is a major overhaul of Fortran 77, and in the view of many, may be unwise; the new language is so large that it will be very costly to implement, and compilers may be too large to run on personal computers. If the standard is adopted, and that is at present debatable, it will likely take many years before compilers for it become routinely available. Only one book, based on an early draft of the Standard, has so far been published [27].

Thus, for the 1980s and 1990s, the Fortran 77 Standard is the final definition of what Fortran is.

Fortran was designed during the punched-card era which lasted from the 1890s until the middle of the 1980s. It was developed about the same time as Noam Chomsky's pioneering work on the mathematical analysis of natural languages, but Chomsky's results were unknown in the computing community until about 1960, where they were adapted to provide a rigorous definition of the language Algol 60, in a notation known as BNF (Backus-Naur Form), named after John Backus, the father of Fortran, and Peter Naur, one of the principal designers of Algol.

Because of these early origins, Fortran does not have a clean syntax, and many peculiar features and restrictions were introduced because it was felt that they would make the job of the compiler writers easier, and at the time, no one knew better, because they had no knowledge of formal language design. With the exception of Cobol and Basic, most languages developed since the early 1960s have been founded on proper grammatical definitions.

John Backus has provided an interesting account of the early developments [12]. Here are three interesting quotes from that article:

> p. 168: "As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs."
>
> p. 169: "Unfortunately we were hopelessly optimistic in 1954 about the problems of debugging FORTRAN programs (thus we find on page 2 of the Report: "Since FORTRAN should virtually eliminate coding and debugging. . . [1]"). . . "
>
> p. 178: ". . . : while it was perhaps natural and inevitable that languages like FORTRAN and its successors should have developed out of the concept of the von Neumann computer as they did, the fact that such languages have dominated our thinking for over twenty years is unfortunate. It is unfortunate because their long-standing familiarity will make it hard for us to understand and adopt new programming styles which one day will offer far greater intellectual and computational power."

In most books, the name of the language is spelled in upper-case, FORTRAN, because it is an acronym; the Fortran 90 Standards Committee has now recommended the spelling Fortran, which we use here.

The two most important contributions of Fortran to the computing world have been the use of high-level statements to replace low-level assembly language coding, and the fact that it permits machine-independent coding, that is, the writing of software that can be run with little or no changes, on a wide variety of computer hardware. The separation of the language used by *programmers* from the language used by the *computer* has been the key to the enormous success of the digital computer.

2 Fortran character set

In Fortran 66, the character set was restricted to upper-case letters and digits, plus these 11 punctuation characters:

- asterisk,
- blank (space),
- comma,

- currency symbol (dollar sign),
- equals,
- minus (hyphen, dash),
- open and close parentheses,
- period,
- plus, and
- slash.

Note that blank is a *character*, just like any other; its printable representation just happens to be empty space.

These 47 characters were universally available on keypunches, and could be represented in the 6-bit 64-character sets used on many computers until the mid-1970s.

It was not until keypunches largely disappeared, and larger character sets (7-bit, 8-bit, and 9-bit) were introduced, that lower-case letters became widely available on computers.

The 1977 Standard extended the Fortran character set with only two new ones: apostrophe and colon, and made no mention of lower-case letters. However, it did permit any character representable in the host processor to be used in the text of a comment statement, or inside quoted strings. Now that lower-case letters are almost universally available, virtually every Fortran compiler permits upper- and lower-case letters to be used interchangeably; letter case is *not* significant, except inside quoted strings. Processors such as `sf3pretty(1)`[1] can be used to normalize letter case, should this ever become necessary.

Because lower-case text is easier to read than upper-case text, many programmers now prefer to write Fortran statements in lower-case, reserving upper-case to distinguish symbolic constants, set in **PARAMETER** statements; this convention is widespread in the C programming language.

3 Fortran names

Names in Fortran must begin with a letter, optionally followed by letters and digits. The name length must not exceed six characters. This rather severe limit is a relic of the 36-bit words of the IBM 704 on which Fortran was first developed. With the 6-bit BCD character set on that machine, at most six characters could fit in one word, and the compiler writers' job was 'simplified'.

[1] Program names followed by a parenthesized digit indicate that documentation can be found on UNIX systems in that section of the manual pages. Type `man 1 sf3pretty` for documentation about that program.

Some compilers permit longer names; however, in the interests of portability of your code, follow the Fortran standard, and ensure that names are never longer than six characters.

4 Fortran statement labels

Executable Fortran statements can be labelled, but only with numeric labels containing one to five digits. Blanks and leading zeroes are not significant in labels, and the position of the label in columns 1–5 of the statement does not matter.

Statement labels serve as reference tags for other Fortran statements, include **DO**, **GOTO**, and *err*= and *end*= exits on I/O statements.

Since every statement label is potentially the target of **GOTO** statement, the difficulty of understanding a program increases sharply when labels are present. A cardinal rule of Fortran programming should always be to *minimize the number of statement labels*.

For readability, statement labels should be kept in ascending order, with uniform increments that allow room for later modifications. An initial label of 10, with increments of 10, is a reasonable choice.

Older code tends to be full of labels, often in apparently random order. Programs known as prettyprinters exist that can clean up Fortran code, relabelling statements so that the labels are in ascending order with constant increments, and unused labels are optionally discarded. Some will also indent **DO** loops and **IF** statements, and do other neatening operations to make code more readable. The author's `pretty(1)` is one such program; it can be usefully combined with `sf3pretty(1)`.

5 Fortran constants

Fortran has six basic data types: real (single precision), double precision, integer, complex, logical, and character.

Integer values contain only an optional leading sign followed by one or more decimal digits. Examples are 1, +17, and -32767.

Floating-point numbers may optionally have a sign, an integer part, a fractional part, and an exponent of ten. Either the integer part or the fractional part must be specified. Examples are 3., .3, 0.3, +3.14, 3E10, 3.14D+00, and -6.27E-27.

The precision of floating-point constants is determined by the exponent letter: E for single precision, and D for double precision. The precision is *not* determined by the number of digits specified (this was the case with a few old compilers); thus, the code

```
DOUBLE PRECISION PI
PI = 3.14159265358979323846264338327 6
```

will assign a severely truncated value to *PI*, because the constant is evaluated in single precision, then converted back to double precision, with zero low-order bits supplied when the fractional part is extended.

Complex constants are represented by a parenthesized comma-separated pair of numbers: (1.0,3.0) and (1,3) are both specifications of the complex value with real part 1.0, and imaginary part 3.0.

Logical constants are either .TRUE. or .FALSE.; these have no particular numerical value, and consequently, logical values may not be used in arithmetic expressions.

Character constants are strings of one or more processor-representable characters delimited by single quotes (apostrophes). If an apostrophe is needed in a string, it must be doubled: the Fortran character constant `'O''Neill'` represents the string O'Neill.

There is no provision for special escape sequence representations for the storage of non-printable characters, such as the C language supports, and since a Fortran statement must fit on one line (the compiler collapses continued statement into a single long line), there is no straightforward way to get a newline character into a string; it can only be done by using string concatenation together with the built-in **CHAR()** function.

The maximum permissible length of a character string constant is implementation dependent; many compilers have very severe length limits, as small as 128, 256, or 500 characters. Such limits are dreadfully low; they do not even allow the assignment of a constant string that spans the 19 permitted continuation lines of a Fortran statement. Arrays have no size limit other than that imposed by addressable memory; character strings ought to have the same limit.

6 Fortran data storage

Because Fortran was designed to be machine-independent, it places only modest requirements on the internal representation of its data types. **INTEGER**, **LOGICAL**, and **REAL** are expected to take one 'storage location'; **DOUBLE PRECISION** and **COMPLEX** take two locations. A storage location is expected to be whatever is convenient for the host computer; it has been as small as 16 bits (minicomputers and microcomputers), and as large as 64 bits (Cray supercomputer).

CHARACTER storage requirements are not specified by the Fortran 77 Standard. Word-addressable machines may have to resort to bit shifting and masking to access characters if several are stored per word. Most modern machines are byte-addressable, so access to individual characters is fast and efficient. Because of these differences, Fortran 77 explicitly prohibits *all* storage association of **CHARACTER** and non-**CHARACTER** data via argument passing, **COMMON** block storage, and **EQUIVALENCE** sharing.

Even though **LOGICAL** data really only require one bit of storage each, Fortran gives them as many bits as an integer uses. The representation of .TRUE. and .FALSE. is up to the compiler; it could be based on the sign bit, the low-order bit (odd or even), zero and non-zero, or any other consistent scheme.

7 Data representation

With the exception of some hand-held calculators that operate in decimal arithmetic, almost all computers today use binary arithmetic, because single bits (*bi*nary digi*ts*: 0 or 1) can be represented by switches that are on or off, or by voltages that are high or low.

Storage addressing is usually not by individual bits, but rather by small units like bytes (usually 8 bits, although other sizes have been used), or words (8, 16, 18, 24, 32, 36, 48, 60, and 64 bits). Byte addressing, and 32-bit words, are the commonest schemes used today.

7.1 Integers

Integers can be represented by a fixed number of bits, usually the number in a machine word. Three schemes are in use for handling signed integers: sign magnitude, one's complement, and two's complement. One bit, usually the left-most, is arbitrarily designated the sign bit. The remaining bits are used for the integer value.

For an n-bit word, with one's complement, and sign-magnitude, both of which are now uncommon, there are $2^{n-1} - 1$ positive numbers, and the same number of negative numbers. There are two zeroes, one positive, and one negative. The more common two's complement representation has 2^{n-1} negative numbers, a single zero, and $2^{n-1} - 1$ positive numbers.

For example, with a 16-bit word, these systems can represent either the range -32767 ... -0 +0 ... +32767 (one's complement and sign-magnitude), or -32768 ... 0 ... +32767 (two's complement).

Having two representations of zero adds extra complexity to the hardware, but in the alternative two's-complement system, which has only one kind of zero, the absolute value of the most negative number cannot be taken, since the corresponding positive value cannot be represented without having an extra bit. A program that does take the absolute value of such a number may produce negative results, because the integer overflow results in wrap-around from a positive value to a negative one, and integer overflow detection is usually disabled in Fortran.

To give a flavor for the decimal precision supported by common word sizes, here is a table of the largest signed integers that can be represented:

Word Size	Largest Signed Integer	Decimal Figures
8	127	2
16	32 767	4
24	8 388 607	6
32	2 147 483 647	9
36	34 359 738 367	10
48	140 737 488 355 327	14
60	576 460 752 303 423 487	17
64	9 223 372 036 854 775 807	18
128	170 141 183 460 469 231 731 687 303 715 884 105 727	38

For example, you would need 10 decimal figures to count the current world population of about 5 billion people. The U. S. national debt is of the order of a few trillion dollars (1 trillion = 10^{12}); you would need at least 14 decimal figures to total it up in dollars and cents. 32-bit integers are inadequate for both of these jobs.

7.2 Floating-point values

In order to represent numbers needed in scientific calculations, it is necessary to store both a fractional part and an exponent. Such a system is called *floating-point*.

One of the new features of the IBM 704, the machine on which Fortran was first implemented in 1956, was hardware floating-point arithmetic. Without such arithmetic, programming becomes exceedingly tedious, because every arithmetic operation requires many machine instructions, usually relegated to a subroutine, and because each multiplication or division can increase the number of bits that must be retained, so frequent rescaling is necessary.

Curiously, many personal computers, including all of the low-end models of the Apple Macintosh, lack floating-point hardware; compilers on such systems permit programs to use floating-point arithmetic, but compile each arithmetic operation into a subroutine call, at the expense of roughly a factor of 100 in reduced performance.

To represent a floating-point number, we need to divide a word into three parts: a sign, s, an exponent, e, of some base, B, and a fraction, f. The fraction is constrained to lie in the interval $1/B \leq f < 1$. A floating-point number, x, is then given by $x = (-1)^s \times f \times B^e$. The most common base today is $B = 2$, but the IBM mainframe architecture uses $B = 16$. A few older machines used $B = 8$.

We need to represent numbers that are both larger and smaller than one, so in order to avoid storing an additional sign for the exponent, the stored exponent is treated as an unsigned integer, and a *bias* of about half the largest exponent is subtracted from it to obtain the true exponent.

For reasons of hardware design, the same size of word that is used to hold an integer is also used to hold a floating-point number. However, since some of the bits have to be given up for the exponent field, fewer bits are available for the fraction, so precision is sacrificed.

The number of bits in the exponent and the fraction varies between architectures. With 32-bit words, typically 6 to 8 bits are chosen for the exponent, and 23 to 25 bits for the fraction. This gives 6 to 7 decimal figures, whereas a 32-bit integer could represent 9 decimal figures.

The number range depends on the base chosen; a larger base increases the number range, but has the undesirable effect of introducing *wobbling precision*.

To see why, consider the IBM 360 choice of $B = 16$, with a 7-bit exponent and a 24-bit fraction. The fraction lies in the range $1/16 \leq f < 1$, which in binary is $0.0001_2 \leq f \leq 0.1111\ldots111_2$. Notice that up to three leading zero bits may be present in some numbers. This reduces the effective average precision of the fraction from 24 bits to 21 bits, a loss of about one decimal figure. The gain is that the exponent range is increased by a factor of two.

Accurate programming in such a system requires particular care. For example, if you are summing a rapidly convergent series of the form $1+ax+bx^2+\cdots$, intermediate sums will have three leading zero bits; if you rewrite it as $8 + a'x + b'x^2 + \cdots$, intermediate sums have no leading zero bits, and you can gain three extra bits of accuracy in the intermediate computations.

Similarly, in such a system, it is better to divide by $2/\pi = 0.\text{A2F9}\ldots_{16}$ than it is to multiply by $\pi/2 = 1.\text{921F}\ldots_{16}$, because the first form has three more significant bits.

Choosing base $B = 2$ eliminates the phenomenon of wobbling precision, and most modern architectures now make that choice.

The 6 to 7 decimal figures supplied by 32-bit floating-point is often inadequate. Almost all floating-point hardware therefore provides double precision arithmetic as well. Two machine words are used to hold a single number, and the number of fractional bits may be more than doubled; the exponent may or may not be increased.

On 32-bit machines, 64-bit double precision numbers may have from 7 to 15 bits for the exponent, and from 48 to 57 bits for the fraction. This provides number ranges as small as $10^{-76}\ldots10^{78}$ (IBM 360) and as large as $10^{-2466}\ldots10^{2465}$ (Cray). The corresponding decimal figures are 17 (IBM 360), and 14 (Cray supercomputer).

Floating-point numbers that are too small to represent are said to *underflow*; such numbers may be silently forced to zero, or they may cause a hardware interrupt which on a few antisocial Fortran implementations terminates the job.

Floating-point numbers that are too large to represent are said to *overflow*. Some machines reset them to the largest floating-point number. A few architectures have special representations for infinity which are used instead.

Indeterminate results, such as division of zero by zero, may produce unpredictable results.

7.3 IEEE P754 floating-point arithmetic

Because of numerous undesirable features of older floating-point systems, during the early 1980s, a committee under the IEEE was established to define a standard for floating-point arithmetic. They produced two such specifications, IEEE 754 for binary arithmetic [23], and IEEE P854 for arithmetic in any radix [15].

The IEEE 754 standard defines three floating point formats:

Name	Total bits	Exponent bits	Fraction bits	Decimal figures	Number range
short real	32	8	24	7	8.43E-37 ... 3.37E+38
long real	64	11	53	15	4.19E-307 ... 1.67E+308
temporary real	80	15	64	19	3.4E-4932 ... 1.2E+4932

The sharp-eyed reader will have noticed that in the short and long real forms, the sum of the numbers of sign, exponent, and fraction bits is one more than the total in the second column. The reason is that these two forms have an extra hidden bit which is not stored; in the gradual underflow region (see below), the hidden bit is not used. The hidden bit is a hardware trick to pick up an extra bit of precision for free. It is possible to do this by requiring the fraction to be normalized; that is, when the base is 2, there are no leading zero bits in the fraction. A few other floating-point architectures use the same trick.

Almost all new computer architectures introduced since 1985 have used the IEEE 754 format, including Intel 80x8x, Motorola 68xxx and 88xxx, MIPS, Stardent, and Sun SPARC chips, and at least one vendor, Convex, has retrofitted support for the IEEE 754 format. These CPUs are the ones used in almost all personal computers and workstations. Most vendors have implemented only the short real and long real forms; only Intel has implemented the 80-bit temporary real format.

Prior to the IEEE 754 standard, both DEC and IBM had introduced 128-bit floating-point formats to satisfy demands for even more decimal figures.

DEC uses a 15-bit exponent of two, and a 113-bit fraction, for a precision of about 34 decimal figures, and a range of 8.405E-4931 ... 5.949E+4931.

IBM's representation allocates a 7-bit exponent of sixteen, and a 112-bit fraction (three leading bits of which may be zero; see the wobbling precision discussion in Section 7.2 on page 8), for a precision of 32 decimal figures, and a range of 5.398E-77 ... 7.237E+76. One byte of the 16-byte number is not used.

Fortran of course only offers two floating-point precisions, **REAL** and **DOUBLE PRECISION**, so vendors have had to extend Fortran to make these higher precisions available to the programmer. The 1983 ANSI Ada

Standard [3] and the 1989 ANSI C Standard [5] are the first to explicitly incorporate support for three (or more) floating-point precisions. The Fortran 90 proposal does too.

Most IEEE 754 arithmetic implementations carry out all intermediate computations in the highest supported precisions, then, if lower precision is needed, round the results to the smaller size. The extra accuracy of intermediate results is usually beneficial, and importantly, the cost of doing computations in higher precision is negligible (lower precision can even be slower, because of the additional conversions needed).

IEEE 754 introduced several new ideas for floating-point arithmetic, excellent discussions of which can be found elsewhere [14, 16, 17, 18, 22].

The most important features of IEEE 754 arithmetic are:

- Explicit representation of signed and unsigned infinity.
- Explicit representation of undefined values, called NaNs (Not A Number); they can be quiet (non-interrupting), or signalling (interrupting).
- Gradual underflow, so that below the normal underflow limit, precision is allowed to decrease to widen the floating-point range.
- Programmer control of rounding or truncating, with round-to-nearest, round-up, round-down, and round-to-zero. The default is round-to-nearest; the other modes permit the efficient implementation of interval arithmetic.

Infinities are generated by dividing non-zero values by zero, or by binary operations involving an infinite value (except non-zero/infinity, which produces zero). NaNs are generated by dividing zero by zero, or by binary operations involving a NaN. The default action is that both infinities and NaNs propagate without interrupting execution.

The presence of an infinity or a NaN in final results is an indication that something abnormal has happened. Floating-point modes can be set to cause interrupts when either is generated, so the offending code can be identified and corrected.

NaNs have a unique property: they are the only floating-point values that compare *unequal* with themselves. This is easy to express in Fortran code:

```
IF (x .ne. x) PRINT *,'x is a NaN'
```

NaNs are useful return values for functions whose values are undefined for certain arguments, such as the square root or logarithm of negative numbers. You can generate a compile-time NaN by dividing zero by zero:

```
REAL NaN
...
NaN = 0.0/0.0
```

Curiously, several compilers I tried refused to accept 0.0/0.0 in a **PARAMETER** statement.

To generate a NaN at run time that can be trapped by an interrupt handler, you have to trick the compiler into delaying the division until run time. On the current SunOS Fortran compiler, this code suffices:

```
x = 0.0
x = x / x
```

A really smart compiler might still handle this at compile time; in such a case, you could do something like this:

```
x = divide(0.0,0.0)
...
REAL FUNCTION divide (x,y)
REAL x,y
divide = x/y
END
```

where the *divide* routine is compiled separately, so the compiler is ignorant of what it really does, and cannot compute its result at compile time.

8 Fortran statement layout

Fortran statements are laid out according to the now-obsolete 80-column punched card. Except inside character strings, blanks are *not* significant anywhere in the Fortran language; it is legal (if of dubious value) to embed blanks in names. The one place where embedded blanks are occasionally useful is to improve the readability of long numeric constants:

```
*       MAXINT = 2**31 - 1
        MAXINT = 2 147 483 647

*       ROOTHF = sqrt(0.5)
        ROOTHF = 0.70710 67811 86547 52440 08443 62104 E+00
```

In comment statements, column 1 is used for the comment starter, which is a C or an asterisk.

In non-comment statements, columns 1–5 hold an integer statement label. Leading zeroes, and blanks before, after, or inside, the statement number are *not* significant. Column 6 is a continuation!column; on the initial line of a multi-line statement, it must be blank or zero. Columns 7–72 hold the text of the statement.

For multi-line statements, on continuation lines, columns 1–5 must be blank, and column 6 must hold a character from the 49-member Fortran character set, other than blank or zero. Up to 19 continuation!line can be specified, so the longest legal Fortran statement is $72 + 19 \times (72 - 6) = 1326$ characters.

Anything appearing after column 72 is *ignored* by the compiler; on punched cards, columns 73–80 were used to hold an 8-digit sequence number that you were grateful for on the day you dropped your deck of punched cards on the floor.

On modern display screens, it is usually not possible to see whether text extends beyond column 72. It is therefore useful to have an intelligent text editor that can test for such instances, or a utility that can do so. For example, in the local GNU Emacs installation, the functions `M-x check-line-length` and `M-x show-long-lines` written by this author will find lines longer than their numeric argument, which defaults to 72 characters if omitted. On any system for which the `awk(1)` utility is available, the simple command

awk 'length($0) > 72 { print $0 }' file1 file2 ... filen

will find problem lines in the listed files.

9 Fortran statement order

Like most programming languages, Fortran has definite rules about the order in which statements must appear in a program. In general, definition must precede usage.

This table is taken from the Fortran 77 Standard [2, p. 3-4]:

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY statements</td><td rowspan="2">PARAMETER statements</td><td>IMPLICIT statements</td></tr>
<tr><td>Other specification statements</td></tr>
<tr><td rowspan="2">DATA statements</td><td>Statement function statements</td></tr>
<tr><td>Executable statements</td></tr>
<tr><td colspan="3">END statement</td></tr>
</table>

From the table, one can conclude that comment statements can go anywhere before the **END** statement. **IMPLICIT** statements precede other specification statement. **PARAMETER** statements can come anywhere in the specification statement section.

10 Fortran statement syntax

In the following subsections, we give a short summary of the syntax of each of the various Fortran statements. **Boldface** text is used for Fortran keywords and other symbols that are required; lower-case text represents things you supply. *Italics* mark phrases that must be replaced by a Fortran keyword. Ellipses (. . .) mark the presence of other Fortran statements.

The order of discussion largely follows the precise descriptions in the Fortran 77 standard [2], but is considerably less formal.

10.1 Comments

Comments are written by placing a C in column 1, followed by the text of the comment. Blank lines were explicitly forbidden in Fortran 66, but were legalized in Fortran 77, which considers any line in which columns 1–72 are blank to be a comment.

Fortran 77 made it legal to use an asterisk in column 1 instead of a C, a rather unnecessary and useless extension. What is needed is a facility for an in-line comment. A few existing compilers, and Fortran 90, provide this in the form of an exclamation point ('bang' for short in computer jargon); the comment continues from that character to the end of the current line. However, such usage should be strictly avoided so as not to compromise code portability.

Fortran 77 permits comments to appear between continuation lines of a multi-line statement. However, earlier compilers, and some tools that process Fortran text, do not, so the practice should be scrupulously avoided.

10.2 Routine header statements

Fortran program modules can be classed into one of four cases: main programs, subroutines, functions, and **BLOCK DATA** routines. These cases are distinguished by a header statement.

10.2.1 PROGRAM statement

A main program may optionally be started with a **PROGRAM** statement:

PROGRAM program-name

If there is no header statement (the usual case), then a main program is assumed.

Every executable program must have exactly one main program. When a job is initiated, the operating system kernel starts it executing at its initial entry point, which is always somewhere in the run-time library. After performing whatever initialization chores are needed, the Fortran run-time library calls the main program, and regains control when the main program terminates, or a **STOP** statement or a **CALL EXIT** statement is executed.

The run-time library does whatever cleanup actions are needed, such as closing files still in use, and then returns to the operating system kernel.

Standard Fortran does not permit a **RETURN** statement to appear in a main program.

10.2.2 SUBROUTINE statement

A subroutine can only be invoked by a **CALL** statement. Arguments can be passed to the subroutine from the calling program; the associations between arguments in the calling and called routines is made at run time, at the point of the call.

SUBROUTINE subrtn-name
SUBROUTINE subrtn-name ( )
SUBROUTINE subrtn-name (arg1 , arg2, . . . , argn)

If the subroutine has no arguments, then either of the first two forms can be used, and the **CALL** statement may be written with or without the parentheses:

CALL subrtn-name
CALL subrtn-name ( )

If arguments are declared on the **SUBROUTINE** statement, they must be matched one-for-one by arguments in a corresponding **CALL** statement; it is an *error* for the number of arguments in the two statements to be different.

A subroutine does not return any value, but it may change its arguments, as well as any global data in Fortran **COMMON**.

Recursion is *not* supported in Fortran; no routine may call itself, either directly, or indirectly.

10.2.3 FUNCTION statement

A function is a routine that returns a single value of a particular data type. As with a **SUBROUTINE**, its arguments must be matched one-for-one in the calling and called program units, and recursion is forbidden. Functions are invoked by name; they may *not* be invoked by a **CALL** statement. Like subroutines, functions may change their arguments, as well as any global data in Fortran **COMMON**. However, such action is generally considered bad programming practice; as a rule, functions should be free of side effects and I/O statements, modify neither their arguments nor any global values, and return a single value to their caller.

Prior to Fortran 77, functions were required to have at least one argument, even if that argument was an unused dummy variable. Fortran 77 extended the language to permit functions with zero arguments, but unlike subroutines, such functions must be declared and referenced with an empty

pair of parentheses. These empty parentheses are needed to distinguish between a variable name and a function name, since Fortran does not require prior declaration of names.

type-name **FUNCTION** function-name ( )
type-name **FUNCTION** function-name (arg1 , arg2, . . . , argn)

The *type-name* value must be one of the six standard Fortran data types: **CHARACTER** or **CHARACTER***n or **CHARACTER***(*), **COMPLEX**, **DOUBLE PRECISION**, **INTEGER**, **LOGICAL**, or **REAL**.

Inside the function, the function-name is treated like an ordinary typed variable, and the last value assigned to it is the value that the function returns to the program unit that invoked it.

Control returns from a **FUNCTION** or **SUBROUTINE** to their caller by execution of a **RETURN** statement. Fortran 77 extended the language to allow the **END** statement to do the same thing; previously, its sole purpose was to inform the compiler where the end of the program unit was. Modern programming practice is to avoid the use of the **RETURN** as much as possible, and let **END** do the job.

10.2.4 BLOCK DATA statement

The last program unit class is the **BLOCK DATA**:

BLOCK DATA
BLOCK DATA block-data-name

This program unit is not executable. It cannot be called, or in any way referenced by name from another program unit. It may contain only specification statements, and **DATA** statements; executable statements are *forbidden*.

Fortran 77 extended the language to permit named **BLOCK DATA** program units, but they are of limited utility, except on systems where the loader or linker refuses to allow more than one such program unit, in which case, they must be given unique names.

The primary use of **BLOCK DATA** is for initializing global data in **COMMON**, but this is fraught with peril in large programs. Since the **BLOCK DATA** routine cannot be called, there is no way to ensure that it will be loaded from a library by the linker or loader; it must be explicitly loaded by name, and if the user forgets to do so, the data initializations will all be lost.[2]

[2]As a graduate student, I once spent several days debugging a large program that had suddenly stopped executing correctly; the error turned out to be the loss of a **BLOCK DATA** routine in the loading phase.

10.3 Specification Statements

Specification statements tell the compiler about the attributes of variables, but are not themselves executable. They must appear before all statement functions, **DATA** statements, and executable statements in the program, and after any header statement.

10.3.1 Type declarations

The names of Fortran scalars, arrays, and function names can be declared by specification statements of the form

type-name var1, var2, . . . , varn

type-name must be one of **CHARACTER** or **CHARACTER***n or **CHARACTER***(*), **COMPLEX**, **DOUBLE PRECISION**, **INTEGER**, **LOGICAL**, or **REAL**.

One or more variable names may be given following the *type-name*; if there is more than one, they are separated by commas.

For character variables, if the *n length modifier is omitted, **CHARACTER*1** is assumed. The length modifier may be replaced by (*) in two circumstances: for a variable which is an argument of the current program unit, and for a constant string in a **PARAMETER** statement. In either case, the length is determinable.

If the variables are arrays, they should be followed by a parenthesized dimension list, as described elsewhere. Names of scalars and *functions* must *not* be followed by parenthesized lists.

Any number of specifications statements can be given in a single program unit, and they can be in any order.

To make life easier on the human reader of the program, it is always a good idea to order variables and declarations systematically. Alphabetical order is usually best. In all but trivial programs, it will be helpful if some commentary is included to describe what each variable is used for. Here are samples of a style I have found useful:

```
C
C       AFMFIL.....Base name of the configuration file.
C
        CHARACTER*12            AFMFIL
        PARAMETER               (AFMFIL = 'afmfiles.dat')
C
C       MAXLIN.....Longest input line.
C
        INTEGER                 MAXLIN
        PARAMETER               (MAXLIN = 255)
```

The author's Extended PFORT verifier [20, 28], `pfort(1)`, has an option to produce declarations organized like this:

```
C ---------------------------------
C
C     EXTERNAL REFERENCES (FUNCTION,SUBROUTINE,COMMON)
C
C     EXTERNAL REFS          ERRAI,          ERRCK,          ERRMS,          MAX0
C     EXTERNAL REFS          MOD,            UTRCA
C
C ---------------------------------
C
C     INTRINSIC FUNCTIONS
C
      INTEGER                MAX0,           MOD
C ---------------------------------
C
C     STATEMENT FUNCTIONS
C
      LOGICAL                ODD
C ---------------------------------
C
C     NON-COMMON VARIABLES
C
C     SETS                   DASHSV,         DIGIT,          IERVAL,         INSTYL
C     SETS                   IPATSV,         IPT1SV,         K,              L
C     SETS                   LAST,           LSSOFT,         MAXPAT,         NPATSV
C     SETS                   PATLEN,         SAVOFF,         SAVPAT,         SIZE
C
      INTEGER                DIGIT,          IARG,           IERVAL(1),      INSTYL
      INTEGER                IPATSV,         IPT1SV,         K,              L
      INTEGER                LAST,           LSSOFT(9),      LSTYLE,         MAXPAT
      INTEGER                NPATSV
      LOGICAL                DASHSV
      REAL                   PATLEN,         SAVOFF,         SAVPAT(25)
      REAL                   SIZE(10)
```

Alphabetical order is adhered to; the initial comments indicate whether the variables are changed in the current program unit. External references are given in comments, and functions are declared before scalars and arrays.

Unfortunately, PFORT does not know about Fortran 77 block-**IF** statements, or the **CHARACTER** data type, so it is not as useful as it once was.

10.3.2 DIMENSION statement

Fortran permits arrays to be declared without a dimension list in a type declaration, if the dimensions are supplied in a specification statement of the form

DIMENSION var1(dimlist), . . . , varn(dimlist)

where dimlist is a comma-separated list of constant dimension limits.

This statement is completely unnecessary, and should be avoided in Fortran programming. Put the dimension information in the type declaration statement.

10.3.3 EQUIVALENCE statement

On older machines with limited memory, storage economization was of prime importance in large programs. Newer machines have very much larger memory spaces, and except for personal computers and the Cray supercomputer, almost all have virtual memory, so this problem is much less severe than it used to be.

Fortran 77's **SAVE** statement (see Section 10.3.9 on page 24) together with stack allocation provides a means to reduce data storage requirements.

The Fortran **EQUIVALENCE** statement was introduced as a means to inform the compiler that two or more different variables (usually arrays) could share the same storage locations, because the programmer had arranged to complete the use of one of them before using any of the others. This statement looks as follows:

EQUIVALENCE (var-a1, . . . , var-an), . . . , (var-k1, . . . , var-kn)

Each variable may be either a scalar, or an array name, or an array element. Because this statement merely informs the compiler to assign the variables to the same starting location in memory, it is illegal to produce conflicting requests, such as this one:

```
REAL A(30)
INTEGER B(30)
EQUIVALENCE (A(1), B(1))
EQUIVALENCE (A(21), B(17)) ⇐= illegal!
```

The use of storage equivalencing can introduce serious, and often difficult-to-find, bugs if the same storage location is used simultaneously for more than one variable. Consequently, such use of the **EQUIVALENCE** statement is *strongly discouraged* in modern programming.

However, the **EQUIVALENCE** statement does have a *legitimate* use in low-level primitives for numerical software. For example, in order to write a function to set or get the sign, exponent, or fraction fields of a floating-point number, one requires bit primitives to access the data, and storage equivalencing to allow access to floating-point values as integer words or bytes.

Here is an example, taken from a set of primitives required for the implementation of the Fortran elementary functions [14]:

```
      double precision function dsetxp (x,n)
*     (Set exponent of x)
*     SunOS UNIX version for Sun 3 Motorola or Sun 4 SPARC
*     architectures, but not for the Sun 386i, because the Intel 386
*     has a different byte order.
*     [14-Nov-1990]
      double precision wf, x
```

```
      integer n, wi(2)

*     Force storage overlay so we can twiddle bits
      equivalence (wi(1), wf)

      wf = x

*     Zero exponent field
      wi(1) = and(wi(1),z'800fffff')

*     Or in the new exponent field
      wi(1) = or(wi(1),lshift(and(1022 + n,z'7ff'),20))

      dsetxp = wf

      end
```

In this function, the double precision value, *wf*, is storage equivalenced with an integer array, *wi(2)*, which allows access to the high-order and low-order words of *wf*.

10.3.4 COMMON statement

In Fortran, non-argument variables declared in a routine are local to that routine; they cannot be known elsewhere, unless they are received or passed as arguments. This is a form of *information hiding*, which is absolutely essential for reducing the complexity of programming. In suspense novels, spies are told only what they 'need to know' in order to limit damage if they are interrogated by the enemy. In programming, reducing the degree of visibility of each item of data likewise reduces the potential for damage.

When routines that call one another need to share data, it is possible to pass the data as routine arguments. However, in large programs, it may be necessary to share data between routines that do not call one another, and some mechanism is needed to provide that capability. Scratch files can of course be used, but I/O is much slower than direct memory accesses. The solution is *global data*. In Fortran, the **COMMON** statement provides a means for declaring such data. It defines an area of memory, either unnamed ('blank' **COMMON**), or named, where the name is specified in the **COMMON** statement:

```
COMMON var1, var2, ..., varn
COMMON /     / var1, var2, ..., varn
COMMON / name / var1, var2, ..., varn
```

The first two forms define a blank **COMMON**, and the third form, a **COMMON** named *name*. If multiple **COMMON** statements with the same name

are given, then their variable lists are implicitly appended in order of their occurrence.

The compiler will output information in the object file that tells the loader or linker to create an area of memory named by the **COMMON** name, and then to place the **COMMON** variables in that area, in the same order in which they were listed in the **COMMON** statement. It is important to note that only the *name* of the memory area is known to other routines; the individual storage locations are *unnamed*. This is quite different than the global storage mechanisms of more modern languages, where each global *variable* names a storage location.

Multiple routines that declare the same **COMMON** blocks can then share the same data, without having to call one another, or write scratch files to exchange the data.

Consider, for example, a graphics library. It needs to maintain a considerable amount of 'state information' (whether the current output device is open, what its name is, what the current point is, what the current transformation matrix is, and so on). It would be unacceptable if all of this information had to be passed through arguments of the graphics routines. This would expose the details of the internal state to every routine in the call chain. Changes in the amount of state information would require changes in a large number of places. Execution would be slowed by all of the additional arguments that had to be passed. In such a case, global data provides a satisfactory solution.

Fortran's **COMMON** mechanism is far from ideal however. First of all, it is not required that each routine declare the same set of variables in a particular **COMMON** block. Older code often used this as another trick for storage economization, just like the **EQUIVALENCE** statement. However, imagine that the **COMMON** variables *were* expected to be the same everywhere, but that in one routine, they accidentally differ: you have an instant bug that may be very hard to find.

The second problem is that many architectures, including all modern RISC architectures, impose constraints on storage alignment. **DOUBLE PRECISION** values must start on a double-word boundary, **REAL**, **INTEGER**, and **LOGICAL** values on a single-word boundary, and **CHARACTER** values on a byte boundary. If the **COMMON** block contains data of mixed types, it is possible to create illegal storage alignments. The compiler may refuse to accept the code, or an expensive run-time fixup may be needed to access the misaligned data.

The third problem is that undisciplined use of **COMMON** tends to introduce serious bugs, and make too much data visible in too many places. Consider for example, the novice programmer who places a variable named *K* in a **COMMON** block; such a variable is very likely to be used as a loop index in many places, and chaos will result if it is reused in a routine called from a loop where it is in use as an index.

The fourth problem is that standard Fortran does not have an **INCLUDE** statement that would permit the programmer to put the **COM-**

MON declarations in a file that was then referenced in an **INCLUDE** statement everywhere the **COMMON** block was needed. Keeping the information in *one* place is the *only* way to ensure that it does not become inconsistent. This is a case where the issues of reliability and maintainability override portability: if your compiler supports an **INCLUDE** statement, *use it* for all **COMMON** declarations. It is not hard to write a simple preprocessor to process **INCLUDE** statements on machines that lack them. Fortran 77 should have introduced this statement, but did not. Fortran 90 does.

The fifth problem is one of data initialization, discussed further in Section 10.2.4 on page 15 and Section 10.5 on page 27. Named **COMMON** variables may only be initialized by **DATA** statements in **BLOCK DATA** routines, and variables in blank (unnamed) **COMMON** cannot be initialized by **DATA** statements anywhere. Blank **COMMON** is a relic of older loaders that reserved a single unnamed area of memory for such storage; on some systems, the size of the area could be adjusted at run time, giving a very primitive means for dynamic storage allocation in Fortran. The object code for large programs, where **COMMON** is most often needed, tends to be stored in load libraries, but since **BLOCK DATA** routines cannot be referenced by any Fortran routine, there is no way to guarantee that the initializations in a **BLOCK DATA** routine will be loaded. This makes **BLOCK DATA** almost useless in practice, and the programmer is forced to resort to run-time initializations, which can be thwarted if the programmer then forgets to arrange for a call to the initializing routine.

In summary, then, **COMMON** blocks are one of the most dangerous areas of Fortran, and the programmer is advised to avoid them except in the rarest circumstances, and when they are unavoidable, to use them with the utmost discipline and discretion, keeping their definitions in separate files that are merged into the source code at compile time with an **INCLUDE** statement.

10.3.5 IMPLICIT statements

Fortran does not require type declarations for scalar or function names. Untyped names are assigned a type based on the first letter of their names: initial I through N means an **INTEGER** variable, and initial A through H or O through Z implies that the variable is **REAL**.

Most languages invented since Fortran have required explicit variable typing, for two reasons. First, when typing is optional, spelling errors may go undetected, leading to very hard-to-find bugs when a mis-spelt variable is assigned a value, and then later used under its correct name, or vice versa. Second, modern languages have a much richer repertoire of data types, and it is unreasonable to single out two of them, as Fortran does, as default types.

Fortran 77, and a few earlier compilers, introduced the **IMPLICIT** statement to provide a means to change the default typing rules:

IMPLICIT *type-name* (*a*, *a*, . . .), *type-name* (*a*, *a*, . . .), . . .

Each *a* is a single letter, or a range of single letters in alphabetical order, denoted by a letter pair separated by a hyphen. Variables beginning with those letters have default types given by the preceding *type-name*. Thus,

IMPLICIT INTEGER (A-Z)

makes all variables default to **INTEGER** type, and

IMPLICIT INTEGER (I - N), **REAL** (A - H , O - Z)

corresponds to Fortran's default typing.

The major use of the **IMPLICIT** statement in older programs is converting between single and double precision:

IMPLICIT DOUBLE PRECISION (A-H,O-Z)
IMPLICIT REAL (A-H,O-Z)

Modern practice is to use explicit typing, supported by compiler options that check for undeclared variables.[3]

Fortran 90, and a few existing compilers, permit an extension of the **IMPLICIT** statement in the form

IMPLICIT NONE

This disables all default typing, so untyped variables will cause a compilation error; in this form, there can be only one **IMPLICIT** statement in the program unit, and it must precede all **PARAMETER** and specification statements. However, since it is not standard, its use is discouraged, except for debugging purposes with compilers that lack a compile-time option to check for untyped variables.

Avoid use of the **IMPLICIT** statement, except possibly for debugging purposes.

10.3.6 PARAMETER statements

Fortran 77 introduced the **PARAMETER** statement as a means of declaring symbolic constants in Fortran programs. Before 1978, a few compilers had already implemented such a facility using a syntax similar to that of a type declaration (**PARAMETER** var = value); the variable was simply another name for the value, and automatically took on the type of the value. Regrettably, Fortran 77 changed the syntax, and introduced typed parameters, so that the value is converted to the type of the variable. The statement looks as follows:

PARAMETER (name = constant-expression)

[3] On Berkeley UNIX, IBM AIX, and SunOS, the Fortran compiler option `-u` provides this service. On Stardent UNIX, `-implicit` does the job. On Apollo systems, use `-type`. On the IBM PC with the Lahey compiler, use `/0`. On VAX VMS, `/WARNINGS:DECLARATIONS` suffices.

As a general rule, the type of the name should be declared in an immediately preceding type statement, with some helpful commentary:

```
C       MAXCHR is the number of characters in the host character set.
        INTEGER MAXCHR
        PARAMETER (MAXCHR = 256)
C       HELLO is the startup banner message
        CHARACTER*(*) HELLO
        PARAMETER (HELLO = 'This is Matlab, version 1.2')
```

10.3.7 EXTERNAL statement

Fortran permits routines to be passed as arguments. An example might be an integration routine that can integrate any function passed by the user.

When a routine name appears in an argument list of a **CALL** statement or a function reference, the compiler cannot tell whether the name is a variable or a routine. The programmer must supply that information in a specification statement of the form

EXTERNAL rtn1, rtn2, . . . , rtnk

Here, each *rtni* is the name of a routine which is received or passed as an argument in the current routine.

Used for this purpose, the **EXTERNAL** statement is both legitimate, and essential.

However, the **EXTERNAL** statement does one other job: it tells the compiler that the routine is not a built-in (intrinsic) function. For example, in the code

```
EXTERNAL SQRT
. . .
PRINT *,SQRT(0.5)
```

the *SQRT* routine must be one defined by the user; it need not even compute a square root.

This second usage of the **EXTERNAL** statement is absolutely not recommended, because it can lead to code that is hard to understand. Fortran programmers are used to assuming that references to intrinsic functions are just that; an **EXTERNAL** statement declaring them otherwise violates that assumption.

10.3.8 INTRINSIC statement

Prior to Fortran 77, it was not possible in standard Fortran to pass an intrinsic function as an argument to a routine, because the **EXTERNAL** statement also made it a user-defined function.

Fortran 77 introduced the **INTRINSIC** statement to remedy this deficiency:

INTRINSIC rtn1, rtn2, . . . , rtnk

Each *rtni* must be the name of an intrinsic function. The effect is that each such function is now available to pass as an argument.

Unfortunately, Fortran 77 introduced yet another arbitrary rule—these functions are not permitted to appear in an **INTRINSIC** statement, or as arguments to another routine:

- type conversion functions (**INT**, **IFIX**, **IDINT**, **FLOAT**, **SNGL**, **REAL**, **DBLE**, **CMPLX**, **ICHAR**, **CHAR**),
- lexical relationship functions (**LGE**, **LGT**, **LLE**, **LLT**), and
- minimax functions (**MAX**, **MAX0**, **AMAX1**, **DMAX1**, **AMAX0**, **MAX1**, **MIN**, **MIN0**, **AMIN1**, **DMIN1**, **AMIN0**, **MIN1**).

10.3.9 SAVE statement

Until modern stack-based architectures became routinely available, Fortran was always implemented in such a way that *local* variables (those declared inside routines that are neither arguments, nor in **COMMON**) were placed in static storage, that is, storage that was used for only one purpose during the execution of a single job.

Thus, programmers could assume that a local variable set on one call would retain its last value on a subsequent call. An example of where this is useful is a routine that needs some first-time initializations that cannot be handled by **DATA** statements:

```
SUBROUTINE sample
LOGICAL first
DATA first / .true. /
IF ( first ) THEN
        . . . initializations. . .
        first = .false.
END IF
```

The initializing code would be executed only on the first call, because at the end, the flag *first* is reset to .false., so subsequent calls will not execute the body of the block **IF** statement.

This practice is widely used in older code. Unfortunately, with modern stack architectures, compilers may choose to place all local variables on the stack, which is an area of memory that is provided afresh to each invocation of a routine. This offers reduced storage requirements for the program, and on many systems, improved execution speed when the hardware has been designed to make stack accesses faster than normal memory accesses.

A compiler that used stack allocation, instead of static storage, for local variables would break a lot of existing code, and several compilers do exactly that. Good ones offer a compile-time option to force static allocation for locals.

Fortran 77 came to the rescue with a new concept: the **SAVE** statement. As a simple declaration of the form

SAVE

it declares that all local variables must be placed in static storage, so their values are saved across invocations of the program unit.

Alternatively, a list of scalar and array variables, or *named* **COMMON** block names enclosed in slashes, can be given by

SAVE var1 , var2 , . . . , varn

so that only those variables are allocated statically; other locals may be stored on the stack.

Here is an example:

SAVE foo, bar, /glob01/, / stuff /

This saves the local variables *foo* and *bar*, as well as the contents of the two **COMMON** blocks *glob01* and *stuff*.

10.3.10 NAMELIST statement

The **NAMELIST** statement defines a collection of one or more variables that may be input or output with namelist I/O statements. Its syntax somewhat resembles that of a **COMMON** statement:

NAMELIST / namelist-name / var1, var2, . . . , vark

However, unlike **COMMON** blocks, namelist groups must be defined in a *single* **NAMELIST** statement. This is unfortunate: the syntax is unusual, and the length of the namelist group is limited by how many variables can be put in a single Fortran statement. Fortran 90 fortunately remedies that deficiency.

When namelist input is read, the *namelist-name* is used in the input stream to identify the namelist group (␣ marks spaces visibly):

```
␣$namelist-name
␣var2 = value2,
␣var1 = value1, var17 = value17,
...
␣$END
```

The *variable* = *value* pairs are in free format, in any order, even duplicated (the last assignment is the one that survives), with the exception that column 1 may *not* be used, because on namelist output, it is used for Fortran carriage control, and output files must be readable as input files.

When namelist output is written, the variables are printed in the order they appear in the **NAMELIST** statement. The appearance of the namelist output depends on the implementation; it may have one variable per line,

or several, and Repeat factors may or may not be used in front of repeated data values. However, it is guaranteed to be readable by a namelist **READ** statement.

Namelist I/O was introduced by IBM in the early 1960s, and has been implemented by many vendors, even though it was not included in the Fortran 77 Standard (it *is* in Fortran 90). It has a several important advantages over list-directed and formatted I/O:

- Programs with numerous input options can supply default values before issuing a namelist **READ**, so that only the variables for which the defaults are not adequate need be specified in the input. This feature can be exceedingly useful in reducing the complexity of user input to programs with many input options.
- Input values are associated with a *name*, rather than an arbitrary column position or list position.
- Because duplicate assignments are permitted, namelist input can be modified by simply adding new assignments before the **$END**; the original assignments can remain unmodified.
- On a **READ**, the file is flushed until the *$namelist-name* is found in column 2; these flushed lines can contain arbitrary comments if desired.
- Namelist output is very handy for debugging, since it provides variable names as well as their values.

For arrays, the input stream can contain multiple values, including values prefixed by a repeat count and an asterisk, just as for list-directed I/O. If an array element is specified in the input stream with multiple data values, the values go into the array in memory order, starting at the location of the specified element. This makes it possible to input only part of an array.

10.4 Statement functions

Fortran provides a very rudimentary one-statement internal function called a *statement function*. It is written as follows:

fun(arg1, arg2, . . .) = expression

The function name, *fun*, should be declared in a preceding type declaration, and the argument variables are dummies that may occur in the right-hand-side expression.

The statement function must be placed after all specification statements, and before all executable statements; see the diagram on page 12.

A statement function is local to the routine it is defined in; it cannot be called from other routines.

Statement functions are sometimes useful in shortening, and clarifying, complicated expressions. For example, suppose you needed the cube root of expressions in several places in one program unit. You could write something like this:

```
DOUBLE PRECISION cubert, xarg
...
cubert(xarg) = (xarg)**(1.0/3.0)
...
CALL foo (cubert(x), cubert(y), cubert(z**3/x**2))
```

Another example might be a statement function to test for a NaN in several places:

```
LOGICAL isNaN
REAL argnan
...
isNaN(argnan) = (argnan .ne. argnan)
...
IF (isNaN(alpha)) PRINT *,'alpha is a NaN'
```

10.5 DATA statements

Before they are explicitly set by an assignment, local Fortran variables have *indeterminate* values.

On some machines, memory is cleared to zero before starting a new job; variables will then start out with zero values.

Other machines leave memory untouched when a new job is initiated, so on them, uninitialized variables will have unpredictable values.

Uninitialized variables are the source of insidious bugs in a large amount of software, and unfortunately, they are less likely to be noticed on those machines that zero memory.

Perhaps someday, computer architects will begin designing memory systems with an extra bit in each addressable storage location; the extra bit would be turned on throughout memory before a job was started, and turned off as values were stored. Any attempt to use an uninitialized value could then be caught by the hardware, so that a software change could be made to fix the error.

One sometimes has variables whose values remain unchanged, or which are reset after the first execution of a program unit, but must have some particular value on the first entry.

Fortran offers the **DATA** statement to handle this case:

```
DATA var / constant-value /
DATA var1 / constant-value1 /, var2 / constant-value2 /, ...
DATA arrayvar / constant-value1, ..., constant-valuen /
DATA var1, var2, ..., varn / constant-value1, ..., constant-valuen /
```

The compiler matches the constant data values with the variables, and then outputs information in the object file that will cause the linker or loader to initialize the memory locations of those variables with the specified values.

This initialization happens *only* when the program is first loaded into memory for execution. If the variables are subsequently assigned new values, their original values from the **DATA** statement are lost.

The types of the variables and the data items must match; if they do not, the action taken is compiler-dependent.

Fortran 77 allowed a variable in a **DATA** statement to be an array name, in which case as many constant values will be collected as there are storage locations in the array. Earlier versions of Fortran permitted only individual array elements to be initialized, which is clearly tedious when there are a lot of them.

Because of Fortran's statement length limit, initializations of large arrays pose a problem. Fortran **DATA** statements allow a repetition factor in front of a constant value, so a short code fragment like

```
INTEGER N(256)
DATA N / 128 * 32767, 128 * -32767 /
```

could be used to preset large numbers of array elements.

What does one do when there are more initializations than will fit in one long continued statement? Fortran 77, and a few earlier Fortran compilers, allow an *implied*-**DO** list, like those in **I/O** statements:

```
INTEGER N(256)
DATA (N(K), K = 1,128) / 128 * 32767/
DATA (N(K), K = 129,256) / 128 * -32767/
```

This permits cross-sections of an array to be initialized, so long initialization statements can always be broken up into several shorter ones.

10.6 Assignment statement

Fortran assignments are reasonably simple:

```
var = expression
```

The left-hand side must be a scalar variable, an array element, or a character substring; array assignments are illegal.

The right-hand side can be any legal Fortran expression.

There is no guarantee which side of the statement is evaluated first; this could matter if the left-hand side contained function references and those functions had side effects.

Regrettably, Fortran 77 introduced another unnecessary restriction. In a character assignment, the left-hand-side variable may *not* appear anywhere on the right-hand side. This was done to eliminate the ambiguity of how assignment of overlapping strings is handled. While this is easy

enough for the programmer to enforce when the left-hand-side variable is local, it is *impossible* when both sides contain variables that are arguments. In such cases, character assignment must be handled through local temporary variables in two steps.

10.7 Control statements

Fortran's control statements are noticeably weak, and structured Fortran preprocessors such as Ratfor [24] and SFTRAN3 [26] have been advocated by many, and are widely used. It is very regrettable that Fortran 77 did not introduce enough new control structures to make such preprocessors unnecessary; that has only happened with the Fortran 90 draft proposal.

Fortran's weak control statements lead to programming practices that produce code that is unmaintainable and unverifiable. I am fond of quoting a particularly bad example that I once encountered: a 150-line routine with about 50 statement labels, and 29 three-way branch **IF** statements. That code has $3^{29} = 6.86 \times 10^{13}$ execution paths.

If you cannot use a structured Fortran preprocessor, then you must exercise particularly strong discipline and restraint in coding flow control in Fortran.

10.7.1 GOTO statement

The **GOTO** (or **GO TO**, since blanks are not significant) statement is in the view of many the root of most flow-control problems in programming languages. E. W. Dijkstra in 1968 published a famous letter [19] entitled *Go to statement considered harmful* which spawned a large number of papers in response, the most important of which is possibly Don Knuth's [25].

Numerous languages designed since Dijkstra's letter have been designed without any **GOTO** statement.[4]

The **GOTO** statement is a simple translation of a hardware jump instruction that every computer has. It has been proven mathematically [13] that sequential execution, conditional execution, and loops, are *necessary* and *sufficient* constructs with which to write any program. If **GOTO** statements are eliminated, they must therefore be replaced by at least one looping construct.

Loops can be classed into two major types: loops where the iteration count is known before hand, and loops where the count is not known, but instead, some test must be made at each iteration to determine whether the loop can be exited. Fortran provides only the **GOTO** and **IF** statements to implement this second class.

Fortran **DO** loops are the most obvious example of counted loops; see Section 10.7.4 on page 32.

[4] Knuth joked that he had received a letter from Professor Eiichi Goto in Japan complaining that he was always being eliminated.

The other kind of loop requires a programmer-defined test for termination of the loop. If that test occurs at the beginning, it is called a *while* loop (e.g. while (there is some input data) process that input data); this is the most common case. If the test occurs at the end, it is called an *until* loop (e.g. do (some work) until (the work is done)). A *while* loop will not be executed if the initial test is false, but an *until* loop will always be executed at least once.

Less commonly, the test may come in the middle of the loop.

Termination of loops is always a matter of concern, and every time you program one, you should think about the *loop invariant*: what condition is true during the execution of the loop, and when it becomes false, causes the loop to exit?

With a Fortran **DO** loop, the loop invariant is that the loop index lies between the initial and final limits, and is changed monotonically once each iteration. That change in the index guarantees that eventually, the index will be moved out of the limit range, and the loop will terminate.

In Fortran 77, you can implement a *while* loop as follows:

```
label   IF (test) THEN
                ... do some work...
                GO TO label
        END IF
```

An *until* loop is not much harder:

```
label   CONTINUE
                ... do some work...
        IF (test) GO TO label
```

A loop with the test in the middle looks like

```
lab1    CONTINUE
                ... do some work...
                IF (test) GO TO lab2
                ... do some more work...
                GOTO lab1
lab2    CONTINUE
```

In each of these three cases, statements in the loop body must eventually change the outcome of the **IF** test, so the loops can terminate.

Modern languages, and structured Fortran preprocessors, eliminate the labels and provide exit statements to get out of the loop with the exit in the middle.

The examples so far have all been of the simple **GOTO**. Fortran has two other types, the assigned **GOTO**, and the computed **GOTO**. The assigned **GOTO** is used like this:

```
        INTEGER where
```

```
      ...
lab1  CONTINUE
      ...
      ASSIGN lab1 TO where
      ...
      GOTO where (lab1, lab2, ..., labk)
```

The assigned **GOTO** must have a list of all possible statement labels that the integer variable has received by **ASSIGN** statements.

The computed **GOTO** uses a small integer value to index a list of statement labels to jump to:

```
      INTEGER where
      ...
      where = 3
      ...
      GOTO (lab1, lab2, ..., labk), where
```

If *where* is 1, control passes to the statement having the first label in the list; if 2, to the statement with the second label, and so on.

In both the assigned and the computed **GOTO** statements, it is an error if the integer value requests a label that is not in the list.

Neither of these statements should ever be used in normal programming. They find application primarily in the code generated by structured Fortran preprocessors.

10.7.2 ASSIGN statement

Fortran permits a statement label to be assigned to a variable with an **ASSIGN** statement (*not* an assignment like *var* = *value*), and subsequently used in an assigned **GOTO** statement. An example is given in Section 10.7.1 on page 30.

10.7.3 IF statement

Fortran has three kinds of conditional statements: the logical **IF**, the arithmetic **IF**, and the block **IF**. The latter was first introduced in Fortran 77.

The logical **IF** takes the form

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

where *statement* is any Fortran statement, except a **DO**, a block **IF**, or another logical **IF**.

The logical **IF** is very common, and convenient when only a single statement is governed by the test.

The arithmetic **IF** looks as follows:

```
      IF ( expression ) lab1, lab2, lab3
```

The expression is evaluated, and control then passes to the statement labelled *lab1* if the result is negative, to the statement labelled *lab2* if the result is zero, and otherwise to the statement labelled *lab3*.

The arithmetic **IF** is a direct reflection of a hardware instruction on the IBM 704 in the 1950s which tested a numerical result, and jumped to one of three following addresses, depending on whether the result was negative, zero, or positive.

The arithmetic **IF** statement is strongly deprecated in modern programming. In IEEE arithmetic, if the expression evaluates to a NaN, none of the tests is true, and it is compiler dependent which of the three labels is selected.

The third kind of conditional statement is the block **IF**. It takes the form

IF (expression-1) **THEN**
...
ELSE IF (expression-2) **THEN**
...
ELSE IF (expression-3) **THEN**
...
ELSE
...
END IF

In the block **IF**, there may be zero or more **ELSE IF** statements, and zero or one **ELSE** statement, which must follow all **ELSE IF** statements. The last statement must be **END IF**. There can be zero or more statements following each **IF** or **ELSE IF** statement.

Block **IF** statements must be properly nested.

Each **IF** or **ELSE IF** test is tried in turn, and the first one that is true results in the execution of the immediately following code block. Execution then passes to the statement following the terminating **END IF** statement.

If none of the tests evaluates to true, then the **ELSE** block is executed. If there is no **ELSE** statement, control passes to the statement after the **END IF**.

It is good programming practice to indent the statements in each of the **IF** and **ELSE** blocks.

It is illegal to transfer into the body of a block **IF** statement with a **GOTO** from outside the block **IF**. Jumping out of a block **IF** is permitted.

The availability of the block **IF** has completely removed the need for the arithmetic **IF** statement, and also substantially reduces the number of labels needed in Fortran programming. It is only regrettable that it took 23 years to become a part of the Fortran language.

10.7.4 DO statement

The Fortran **DO** statement provides one useful kind of loop, the counted loop; for other types, see the discussion in Section 10.7.1 on page 29.

The statement is written as follows:

```
      DO label loop-var = initial, final, increment
                .
                .
                .
label statement
```

It is good programming practice to ensure that the terminal labelled statement is always a **CONTINUE** statement, and that nested **DO** loops do not share terminal statements.

The increment in the **DO** defaults to 1 if it is omitted; this is most often the case.

Prior to Fortran 77, **DO** loop index variables were not permitted to cross 0, and negative increments were forbidden. Fortran 77 removes those restrictions.

Before Fortran 77, the behavior of the loop when the initial index value was larger than the final value was compiler dependent: some generated a one-trip loop, and others a zero-trip loop. Fortran 77 remedied this ambiguity by requiring a zero-trip loop.

In Fortran 77, the loop is executed by first computing an iteration count, equal to the value of the expression *max(int((final - initial + increment) / increment), 0)*. The loop index is set equal to the specified initial value, and the iteration count is then compared to zero. The loop is terminated if the count is smaller than one. Otherwise, the loop body is executed, and at the end, the iteration count is reduced by one, the loop index is bumped by the specified increment, and control returns to the top of the loop for another test of the iteration count.

The usual case of

```
      DO 10 K = 1, N
                .
                .
                .
10    CONTINUE
```

results in the loop body being executed exactly *N* times.

It is good programming practice to indent the loop body.

10.7.5 CONTINUE statement

The **CONTINUE** statement serves as a null statement; its normal use is for the terminal statement of a **DO** loop. It is optionally labelled:

```
label CONTINUE
```

10.7.6 STOP statement

In the early days of Fortran, the **STOP** statement actually halted the program and the computer. When multiprocessing became common, it served to terminate execution of just the program.

It takes one of the forms

STOP
STOP nnnnn
STOP 'quoted string'

In the second form, nnnnn is a 1 to 5 digit integer constant. In the second and third forms, the constant is usually displayed at the termination of execution. Regrettably, expressions are forbidden (they would introduce a syntactical ambiguity because of Fortran's insignificant blank rules).

Because execution of the **END** statement in a main program terminates the program, the **STOP** statement should normally be used only for termination of execution in unrecoverable abnormal circumstances.

Many Fortran run-time systems print a message on the screen or job log when a **STOP** statement is executed; the message is not shown if no **STOP** is executed.

10.7.7 PAUSE statement

The **PAUSE** statement takes the forms

PAUSE
PAUSE nnnnn
PAUSE 'quoted string'

In the second and third forms, the integer constant or quoted string is displayed on the screen or job log, and execution pauses. It must be possible to resume execution, but the means for doing so are system-dependent.

On older machines, the **PAUSE** statement served mainly for communication with the computer operator, such as to mount the next input or output tape.

In modern programs, the **PAUSE** statement should be avoided.

10.7.8 RETURN statement

Before Fortran 77, the only way a routine could legally return to its caller was to execute a

RETURN

statement. Fortran 77 permits the **END** statement to perform this job, and modern practice is to severely limit use of **RETURN**.

One of the rules of top-down programming is that routines should have a single entry (at the top), and a single exit (at the bottom); **RETURN** statements in the middle of a routine violate that convention, and are discouraged.

10.7.9 ENTRY statement

The **ENTRY** statement is a rarely used Fortran feature that permits a routine to have multiple names and argument lists. Its use is strongly discouraged, and its syntax will not even be shown here.

10.7.10 END statement

The end of any program unit is signalled by the

END

statement, which must be the last statement in the program unit. Any statements found after the **END** statement in the current input file are assumed to belong to the following program unit.

In Fortran 77, if execution reaches this statement in a function or subroutine, control returns to the caller; in a main program, control returns to the run-time library, and thence to the operating system.

In older Fortrans, the **END** statement was not executable, and it was an error to reach it. Modern practice is to use **END** as the normal way of returning from a subroutine, function, or main program.

10.8 Input/Output statements

Input/output is one of the richer areas of Fortran, and its complexity gives novice programmers considerable difficulty. It is also an area where numerous dialectical differences existed prior to Fortran 77. The 1977 Standard did much to improve the situation, but language extensions, particularly to the **OPEN** and **CLOSE** statements, exist in most compilers.

Fortran input/output is *record*-oriented. Each I/O operation handles one or more records. In text files, Fortran records are *lines*, so that it is impossible in standard Fortran to write partial lines.

Fortran offers support for several kinds of sequential I/O: list-directed (free form), namelist, formatted, and unformatted (binary).

Internal files in the form of character variables are also supported; in **READ** and **WRITE** statements, the character variable replaces the unit number. In this form, I/O is restricted to formatted I/O, and of course, neither an **OPEN** nor a **CLOSE** statement is appropriate.

Formatted and unformatted I/O can also be performed on direct-access files; records in such files can be processed in any order.

With formatted I/O, record delimiters are line delimiters; they may be special control characters, as on most systems that use the ASCII character set, or special markers in the file, or just implicit positions in files made up of fixed-length records. On essentially all systems, such files can be read by programs written in other languages.

Fortran unformatted (binary) records must contain special codes that mark the start and end of records, so that files can be read forwards or backwards. These extra codes mean that Fortran unformatted files contain additional information beyond what the programmer specified in the I/O list, and it is therefore impossible in standard Fortran to write binary files for which the programmer has complete control over the contents. Since there is a frequent need for such a facility (e.g. to write binary files to be

sent to a graphics device), programmers must resort to language extensions to accomplish that, and the resulting code is not portable.

Because of the extra markers, Fortran binary files are usually not easy to read by programs written in other languages.

Because the data is recorded in its native binary format, Fortran binary files in general cannot be exchanged between unlike architectures, or perhaps even between Fortran programs on the same machine, but compiled with different Fortran compilers.

Binary files are nevertheless very useful, because they offer substantially faster I/O than formatted files. Also, the exact bit patterns of all data are preserved with binary I/O. Formatted I/O involves a conversion between binary and decimal, and in general introduces errors.

The model of I/O processing assumed by Fortran 77 is that files must first be *opened* for processing, either by an explicit **OPEN** statement, or implicitly according to local conventions of the compiler, run-time library, and operating system. Once opened, a file can be read or written using **READ** and **WRITE** statements, and possibly **PRINT** statements. The position of a sequential file can be controlled by the **BACKSPACE** and **REWIND** statements. A file can be truncated by an **END FILE** statement. After processing, the file is closed, either with an explicit **CLOSE** statement, or implicitly when control returns to the run-time library from execution of a **STOP**, **CALL EXIT**, or a main program's **END** statement.

Reference to an open file is made not by the name of the file, but more conveniently by a small integer number, called the *unit number*. The unit number is selected in the **OPEN** statement, and used until the file is closed. After the close operation, the same unit number can be reused, possibly for a different file.

The integers available for unit numbers are regrettably system-dependent, and unfortunately, no standard Fortran library routine exists to obtain an unused unit number. In general, unit numbers in the range 1–19 can be expected to be available. A few systems allow zero or negative unit numbers, and a few place no restrictions at all on the unit number.

The number of files that can be open at one time is system dependent. It may be as few as a half dozen, or as large as several dozen. It is therefore advisable to close files once they are no longer needed.

Some unit numbers may be pre-assigned to certain I/O devices; the local Fortran Programmer's Guide must be consulted for information about such restrictions.

IBM mainframe Fortran since the 1950s has used the convention that unit 5 is text input, unit 6 is the printer, and unit 7 is the card punch (now an obsolete device). Most Fortran vendors follow IBM's custom. On Berkeley UNIX and SunOS, unit 0 is preconnected to *stderr*, unit 5 to *stdin*, and unit 6 to *stdout*; none of these units should ever be opened or closed, and even **REWIND** should be avoided, since that may be illegal if the files are connected to UNIX pipes.

Fortran 77 permits the unit number to be replaced by an asterisk, which

causes default pre-opened input and output units to be used. While this form is adequate for small throw-away programs, it is not advisable in software that might be moved to other machines, because the file assignments can be changed only by revising the source code to use numbers instead of asterisks.

The asterisk form of a unit number can only be used in **PRINT**, **READ**, and **WRITE** statements; it cannot be used in **BACKSPACE**, **CLOSE**, **END FILE**, **INQUIRE**, **OPEN**, or **REWIND** statements.

PRINT, **READ**, and **WRITE** statements have an optional I/O list, designated *iolist* in the following subsections. This is a comma-separated list of variables (scalars or array names). For output, expressions are permitted.

Portions of arrays may be selected with the *implied* **DO**, as illustrated by these examples:

READ (. . .) (A(K), K = K1, K2)
READ (. . .) ((B(K,L), K = K1, K2), L = L1, L2, L3)

The implied loops work just like Fortran **DO** loops, with the computation of an iteration count, a loop index variable, and initial, final, and optional increment values.

Each *iolist* forms one Fortran record on unformatted files, and one or more records on formatted files. On most systems, there is no real limit to the amount of data that can be transmitted with a single Fortran I/O statement.

10.8.1 Fortran carriage control

Some short remarks about Fortran carriage control are in order. In the early days of Fortran, output devices were limited to line printers, card punches, and magnetic tape. Six-bit character sets offered only 64 printable characters, with no provision for any control characters.

On line printers, it is desirable to be able to control the output spacing, as well as to be able to force the start of a new page. The convention was therefore adopted that the first character of each record sent to a line printer is not printed, but instead, is used as a printer carriage control instruction:

Character	Vertical Space Before Printing
blank	one line
0	two lines
-	three lines
+	no advance (i.e. overprint last-written line)
1	to first line of next page

The minus for triple line spacing was not universally available. Some printers accepted other characters, but usually, anything in column 1 other than

those five characters was treated like a blank, and single line spacing resulted.

When printers commanded by ASCII control characters (i.e. most produced since the late 1970s) became common, these conventions were no longer of any use, and today, they are inappropriate for most Fortran output. Nevertheless, they persist in a lot of software, and veteran programmers have become accustomed to leaving an initial space in each output line.

There is little reason to be concerned with Fortran carriage control in most new code, however, with one caveat.

On operating systems like MS-DOS, UNIX, and TOPS-20, files are just a stream of bytes, with no attached attributes, and column 1 of printer files has no special significance.

On other operating systems, like DEC's VAX VMS and IBM's mainframe systems, files have additional attributes, like 'fixed length records', and 'Fortran carriage control'. On those systems, Fortran formatted output files may by default have a Fortran carriage control attribute that gives the first character of every line special interpretation. On such systems, the simple act of copying the file to a terminal, or into a text editor buffer, may cause column 1 to disappear. On VAX VMS, this action can be eliminated by opening the file with the special non-standard **carriagecontrol**=*'list'* attribute in the **OPEN** statement.

10.8.2 BACKSPACE statement

The statement

BACKSPACE n

backspaces unit *n* one Fortran record. If the last **READ** operation reached end-of-file, then the **BACKSPACE** positions the file before the end-of-file, such that a following **WRITE** will add a record to the file.

10.8.3 END FILE statement

The statement

END FILE n

writes an end-of-file marker on unit *n*. Since a **REWIND** after a **WRITE** accomplishes the same thing, the **END FILE** statement receives little use.

10.8.4 PRINT statement

The **PRINT** statement is a holdover from the 1950s when Fortran's I/O statements were device-dependent (there were also **READ INPUT TAPE**, **WRITE OUTPUT TAPE**, and **PUNCH** statements).

It takes the forms

PRINT format-label, iolist
PRINT *, iolist

In the first of these, formatted output of data in the *iolist* to the default print file (on UNIX, *stdout*) is made according to the referenced **FORMAT** statement. In the second form, output is list-directed (free format).

Because the **PRINT** statement is preconnected to a fixed file, it is not advisable for general programming use. Small programs that require only a single output file may find it convenient.

10.8.5 READ statement

The **READ** statement takes the general forms

READ format-label
READ format-label, iolist
READ (cilist)
READ (cilist) iolist

The first two forms are not often used, because they read only from an implementation-defined unit.

cilist is a *control information list*. It is a comma-separated list of values that supplies the Fortran unit number, and other items that depend on the I/O method. The first item is *always* the unit number, which is an integer expression, usually a constant, or a variable name.

For formatted I/O, the second item is a format specification, which is either the numeric statement label of a **FORMAT** statement defined elsewhere in the routine, or else is a character expression which evaluates to a string containing the format, including the outer parentheses.

For list-directed I/O, the second item in the *cilist* is an asterisk.

For namelist I/O, the second item in the *cilist* is a namelist name, defined in a **NAMELIST** statement in the specification statement section.

The remaining entries in the *cilist* are *keyword* = *value* pairs. Here are some examples of *cilist*s:

formatted I/O	(unit-number, format-label)
	(unit-number, format-label, **end**=eof-label)
	(unit-number, format-string)
	(unit-number, format-string, **end**=eof-label)
unformatted I/O	(unit-number)
	(unit-number, **end**=eof-label)
direct-access I/O	(unit-number, format-label, **rec**=recnum)
	(unit-number, format-string, **rec**=recnum)
	(unit-number, **rec**=recnum)
	(unit-number, **rec**=recnum)
list-directed I/O	(unit-number, *)
	(unit-number, *, **end**=eof-label)
namelist I/O	(unit-number, namelist-name)
	(unit-number, namelist-name, **end**=eof-label)

Here are some sample formatted I/O statements:

```
      READ (5,1000) X,Y
      READ (5,'(2F10.3)') X,Y
      READ (5,'(2F10.3)',end=99) X,Y
1000  FORMAT (2F10.3)
```

These read two floating-point variables according to the format 2F10.3. The third will jump to the statement labelled 99 if an end-of-file is reached during the read.

Here are two list-directed (free format) input statements:

```
      READ (5,*) X,Y
      READ (5,*,end=99) X,Y
```

The two input values appear in the input stream separated by whitespace (including line breaks), or a single comma.

Here are some unformatted (binary) input examples:

```
      READ (17) N, (A(K), K = 1,N)
      READ (9,end=500) X, Y
```

Note that in the first statement, the number of elements in the vector is recorded in the input record, and can be used in the reading of the remainder of the record.

Namelist input is straightforward:

```
      NAMELIST /MYDATA/ X, Y, Z
      ...
      READ (5,MYDATA,end=99)
```

No *iolist* is used, because the namelist input contains the variable names and values, in any order.

Direct-access I/O is available in both formatted and unformatted variants:

READ (17,'(2F10.3)',**rec**=23) X,Y
READ (17,**rec**=23) X,Y

Each of these selects record number 23. That record must have previously been written by a corresponding direct-access **WRITE** statement. It is not necessary in a direct-access file to write all records (i.e. holes may exist), but you can legally read only those that have been written.

Since Fortran records are identifiable in files, it is permissible for a **READ** statement to have a shorter *iolist* than the one in the **WRITE** statement that wrote the file. This is sometimes useful; here is a short code fragment that positions to the end of an unformatted file, ready to write more data:

```
10    READ (1,end=20)
      GO TO 10
20    CONTINUE
      ...
      WRITE (1) iolist
```

Reading unwanted records this way is faster than if *iolist*s were used, and also does not require knowledge of what the *iolist*s were. This can be important, because the structure of Fortran files is determined by the *iolist*s used to write them, and that is entirely the responsibility of the programmer.

Fortran records *no information whatever* in the file about the data types or array lengths from the I/O list. For formatted, list-directed, and namelist output, only newlines are added; for unformatted I/O, Fortran only inserts special markers that delineate the start and end of each record. Otherwise, the data in the file is nothing but a raw stream of bits. This makes Fortran files very flexible, but also easily subject to programming errors if the *iolist*s on **WRITE** and corresponding **READ** statements mismatch.

10.8.6 REWIND statement

By default, all Fortran files are normally opened so that the next **READ** statement will read the first record. In order to write a sequential file, then read it back without having to close and reopen it, you can issue a **REWIND** statement:

REWIND unit-number

For example, the three statements

```
WRITE (1) A,B,C
REWIND 1
READ (1) X,Y,Z
```

are effectively the same as the following three assignments, only considerably slower:

X = A
Y = B
Z = C

10.8.7 WRITE statement

Fortran records are written with the **WRITE** statement, which is analogous to the **READ** statement, except that the **end**=*nnn* item is of no use:

WRITE (cilist)
WRITE (cilist) iolist

A single formatted **WRITE** statement produces one or more output records (i.e. lines); a new record is started for every slash format item, and every time the end of the format specification is reached when more data remains to be written.

A single list-directed or namelist **WRITE** statement produces one or more output lines.

A single direct-access **WRITE** statement produces one record in the direct access file. It is an *error* if the length of the *iolist* is longer than the record length declared in the **recl**=*nnn* entry in the **OPEN** statement.

For sequential files, there should be no limit on the amount of data that can be read or written with one I/O statement.

10.8.8 FORMAT statement

Format specifications often present great difficulty for novice programmers. The reason is that the contents of the format specification conform to a little programming language, albeit a special-purpose one, whose syntax is somewhat irregular.

The general idea is that each item in the *iolist* is matched one-for-one with format specifiers, and formatted accordingly. If there are more items in the *iolist* than specifiers in the format, then the format is re-used by returning to the beginning of the specification terminated by the last close parenthesis. If there is no last closing parenthesis, control returns to the beginning of the format specification. *In either case, a new record is started in the file.*

Repeatable edit descriptors are summarized in the following table:

Iw	integer value right-adjusted in field of width w
Iw.m	integer value right-adjusted in field of width w, with at least m digits (leading zeroes are supplied if necessary)
Fw.d	**REAL** value right-adjusted in field of width w, with d fractional digits
Ew.d	**REAL** value right-adjusted in field of width w, with d fractional digits, and a two-digit power-of-ten exponent field
Ew.dEe	**REAL** value right-adjusted in field of width w, with d fractional digits, and a power-of-ten exponent field of width e
Dw.d	**DOUBLE PRECISION** value right-adjusted in field of width w, with d fractional digits, and a two-digit power-of-ten exponent field
Dw.dEe	**DOUBLE PRECISION** value right-adjusted in field of width w, with d fractional digits, and a power-of-ten exponent field of width e
Gw.d	**DOUBLE PRECISION** value right-adjusted in field of width w, with d fractional digits, and an optional two-digit power-of-ten exponent field, which is used only if necessary
Gw.dEe	**DOUBLE PRECISION** value right-adjusted in field of width w, with d fractional digits, and an optional power-of-ten exponent field of width e, which is used only if necessary
Lw	**LOGICAL** value output right-adjusted in a field of width w, as a letter T or F
A	**CHARACTER** value output in a field of the width of the corresponding *iolist* expression
Aw	**CHARACTER** value output right-justified in a field of width w

Any of these may be prefixed with an integer repeat count (e.g. 2E15.7 is equivalent to E15.7, E15.7).

Non-repeatable edit descriptors are as follows:

'cc. . . cc'	character string constant
nHhh. . . hh	Hollerith string constant
Tc	tab to absolute column c for next item
TLc	tab left c columns before next item
TRc	tab right c columns before next item
nX	skip n spaces before next item
/	start a new record
:	terminate format processing if there are no more *iolist* items
S	revert to default sign processing
SP	force plus signs on numeric items
SS	suppress plus signs on numeric items
kP	scale next floating-point items by 10^k
BN	ignore blanks in fields
BZ	treat all blanks as zeroes

When multiple edit descriptors are present, they should be separated by commas; the slash format item does not need delimiting commas.

There are a lot of details that need to be considered, and a textbook treatment is essential.

For the beginner, the Iw, Fw.d, Ew.d, Aw, Lw, 'ccc. . . ccc', nX, and / format items provide a useful subset that can be used effectively. A single example may serve to illustrate them:

```
      WRITE (6,1000) N, X, Y, 'hello', (X .EQ. Y)
1000  FORMAT (1X, I5, F10.2, E15.5, 1X, A, 1X, L3 / ' That was easy.')
```

It might produce output like this (␣ marks spaces visibly):

```
␣␣␣255␣␣␣␣␣␣␣3.14␣␣␣0.27182E+01␣hello␣␣␣F
␣That was easy.
```

10.8.9 OPEN statement

Files can be opened by name or unit number using the **OPEN** statement. There are numerous standard options of the form *key* = *value*, and most implementations provide additional ones to cater to the needs of the local file system.

We shall show only the simplest cases here.

Many Fortran run-time libraries provide a default association between unit numbers and file names, in which case only a unit number is needed.

```
      OPEN (unit=1)
```

For example, on SunOS, this will open a sequential formatted file named `fort.1`; on Stardent UNIX and VAX VMS, this creates a file named `FOR001.DAT`.

Relying on fixed file names is inflexible and unsafe; more commonly, a file name is available, possibly constructed in the program itself:

OPEN (**unit**=1, **file**='myfile.dat')

This would open the file `myfile.dat`.

These simple cases do not provide the Fortran library with all the information that might be necessary, so defaults are assumed. However, some libraries will issue warning messages when this happens. A recommended method is to specify the access method and the file status:

```
      OPEN (unit=1,
     X      file='myfile.dat',
     X      access='sequential',
     X      form='formatted',
     X      status='unknown')
```

If the file is intended for immediate input, change *'unknown'* to *'old'*, so that the program will be terminated if the file does not exist.

10.8.10 CLOSE statement

The

CLOSE (**unit**=unit-number)

statement closes the file currently associated with the specified unit number, and frees whatever internal resources (e.g. buffer memory) that may have been required while the file was open. The unit number is then available for re-use for processing of any other file.

10.8.11 INQUIRE statement

The **INQUIRE** statement provides a means of finding out whether a file is open, and if so, what attributes were requested at open time.

INQUIRE (**unit**=unit-number, . . .)

Like the **OPEN** statement, it has a great many *keyword* = *value* options, only a few of which are commonly needed.

To find out the name of an open file, do

INQUIRE (**unit**=unit-number, **name**=namvar)

The name will be returned in the character variable named *namvar*.

To find out whether a file is open on a particular unit, do

INQUIRE (**unit**=unit-number, **opened**=logvar)

The logical variable *logvar* will be set .TRUE. on return if the file is open, and .FALSE. otherwise.

This last form can be used to good effect in a function to find an unused unit number:

```
      INTEGER FUNCTION getun ()
      LOGICAL isopen
      getun = 1
10    INQUIRE (unit=getun, opened=isopen)
      IF (isopen) THEN
            getun = getun + 1
      ELSE
            RETURN
      END IF
      IF (getun .LE. 50) GOTO 10
      getun = -1
      END
```

This will try units 1, 2, . . . , 50 in turn, returning as a function value the first unused unit number it finds. If after 50 tries, no available unit number can be found, something is probably wrong, and a negative unit number is returned.

Although unit number 0 is available on some Fortran implementations, not all support it, so we start the search with unit number 1.

The maximum number of simultaneously open units varies from system to system, but is often in the range 5 to 20. It is therefore reasonable to terminate the loop after a few dozen tries.

References

[1] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *ANSI Fortran X3.9-1966*, 1966. Approved March 7, 1966 (also known as Fortran 66). See also subsequent clarifications [9] and [10], and history [21].

[2] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *ANSI Fortran X3.9–1978*, 1978. Approved April 3, 1978 (also known as Fortran 77). See also draft [11].

[3] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Military Standard Ada Programming Language*, February 17 1983. Also MIL-STD-1815A. See also [8, 7].

[4] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Draft Proposed ANSI Fortran X3.9–198x*, September 18 1987. See also [27].

[5] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[6] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Draft Proposed American National Standard Programming Language Fortran Extended X3.198–199x*, September 24 1990. See also [27].

[7] Anonymous. Preliminary Ada reference manual. *ACM SIGPLAN Notices*, 14(6A), June 1979. The final standard is [3].

[8] Anonymous. Rationale for the design of the Ada programming language. *ACM SIGPLAN Notices*, 14(6B), June 1979. The final standard is [3].

[9] ANSI Subcommittee X3J3. Clarification of Fortran standards—initial progress. *Communications of the Association for Computing Machinery*, 12:289–294, 1969. See also [1].

[10] ANSI Subcommittee X3J3. Clarification of Fortran standards—second report. *Communications of the Association for Computing Machinery*, 14:628–642, 1971. See also [1].

[11] ANSI Subcommittee X3J3. Draft proposed ANS Fortran. *ACM SIGPLAN Notices*, 11(3), 1976. See also final standard [2].

[12] John Backus. The history of FORTRAN I, II, and III. *ACM SIGPLAN Notices*, 13(8):165–180, August 1978.

[13] C. Boehm and G. Jacopini. Flow diagrams, Turing machines, and languages with only two formation rules. *Communications of the Association for Computing Machinery*, 9(5):366–371, May 1966. CODEN CACMA2. ISSN 0001-0782.

[14] William J. Cody, Jr. Analysis of proposals for the floating-point standard. *Computer*, 14(3):63–69, March 1981. See [23].

[15] William J. Cody, Jr., Jerome T. Coonen, David M. Gay, K. Hanson, David Hough, W. Kahan, R. Karpinski, John F. Palmer, F. N. Ris, and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE Micro*, 4(4):86–100, August 1984.

[16] Jerome T. Coonan. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, March 1981. See [23].

[17] Jerome T. Coonen. An implementation guide to a proposed standard for floating point arithmetic. *Computer*, 13(1):68–79, January 1980. See errata in [18]. See [23].

[18] Jerome T. Coonen. Errata: An implementation guide to a proposed standard for floating point arithmetic. *Computer*, 14(3):62, March 1981. See also [17, 23].

[19] Edsger Wybe Dijkstra. Go to statement considered harmful. *Communications of the Association for Computing Machinery*, 11(3):147–148, March 1968. This paper inspired scores of others, published mainly in SIGPLAN Notices up to the mid-1980s. The best-known is [25].

[20] A. D. Hall. A portable Fortran IV subset. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1969.

[21] W. P. Heising. History and summary of Fortran standardization development for the ASA. *Communications of the Association for Computing Machinery*, 7:590–625, 1966. See also final standard [1].

[22] David Hough. Applications of the proposed IEEE-754 standard for floating point arithmetic. *Computer*, 14(3):70–74, March 1981. See [23].

[23] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles [14, 16, 17, 18, 22, 29]. Available from the IEEE Service Center, Piscataway, NJ, USA.

[24] Brian W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, Reading, MA, USA, 1976. ISBN 0-201-03669-X. 338 pp. LCCN QA76.6 .K42 1976.

[25] D. E. Knuth. Structured programming with go to statements. *Computing Surveys*, 6:261–301, 1974. This paper is a response to [19].

[26] C. L. Lawson and J. A. Flynn. SFTRAN3 programmer's reference manual. Technical Report 1846-98, Jet Propulsion Laboratory, Pasadena, CA, USA, December 1978.

[27] Michael Metcalf and John Reid. *Fortran 8x Explained*. Clarendon Press, Oxford, UK, 1987. ISBN 0-19-853751-4 (hardcover), 0-19-853731-X (paperback). xiv + 262 pp. LCCN QA76.73.F26 M48 1987. See also [4, 6].

[28] Barbara G. Ryder. The PFORT verifier. *Software—Practice and Experience*, 4:359–377, 1974.

[29] David Stevenson. A proposed standard for binary floating-point arithmetic. *Computer*, 14(3):51–62, March 1981. See [23].

Index