FORTRAN 2018 WITH PARALLEL PROGRAMMING

Subrata Ray

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Fortran 2018 with Parallel Programming

Fortran 2018 with Parallel Programming

Subrata Ray

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-0-367-21843-0 (Hardback)

Library of Congress Control Number: 2019946209

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

This work is dedicated to the memory of

Professor Chanchal Kumar Majumdar

and

Professor Suprokash Mukherjee

Contents

Preface

Since the early days of machine computing, there has been a constant demand for *larger* and *faster* machines. The two terms essentially mean machines with larger memory and more speed than that of the existing available machines. During the past 70 years, there have been dramatic changes in the fields of computer hardware and software—from vacuum tubes to VLSI (very large scale integration) and from no operating system to very sophisticated, time-sharing operating systems. There are three obstacles that computer designers face while aiming to increase the speed of the computer. First, the density of the active components within a chip cannot be increased arbitrarily. Second, with the increase of the density of the active components within VLSI chips, heat dissipation becomes a severe problem. Third, the speed of a signal cannot exceed the speed of light according to the special theory of relativity proposed by Einstein. Thus, a different approach to the problem has been thought of.

Instead of having a single processor, if several processors (each may not be very fast and can be inexpensive) participate in parallel for computation, the speed of calculation can be increased considerably, and in fact, using inexpensive processors controlled by special software and hardware, the speed of a supercomputer can be achieved if hundreds of processors work together in parallel.

This book contains an introduction to parallel computing using Fortran. Fortran supports three types of parallel modes of computation: Coarray, OpenMP and Message Passing Interface (MPI). All three modes of parallel computation have been discussed in this book. In addition, the first part of the book contains a discussion on the current standard of Fortran, namely, Fortran 2018.

The first part of the book can be used to learn the modern Fortran language even if the reader has not yet been exposed to the earlier versions of Fortran. The book should be read sequentially from the beginning. However, a reader who is conversant with the earlier versions of Fortran may skip the introduction to Fortran and go directly to the new features of the language.

As Fortran is mainly used to solve problems related to science and engineering, standard numerical methods have been used as a vehicle to illustrate the application of the language. However, knowledge beyond the level of elementary calculus is not required to understand the numerical examples given in the book. The emphasis of the book is on programming language, not on sophisticated numerical methods. The programming examples given in the book are simple, and to keep the code readable, the codes are not always optimized. It is expected that a reader, after proper understanding of the language, would be able to write *much more efficient* codes than the codes given in the book.

Programming tips and style have been introduced at appropriate places. They serve simply as guidelines. It is well known that every experienced programmer has his or her own programming style.

To keep the size of the book reasonable, all available features of Fortran 2018 have not been discussed. Moreover, only the essential components of Coarray, OpenMP and MPI, which are required to write reasonably useful programs, have been discussed. It is hoped that readers, after going through this book, will refer to relevant manuals and be able to write parallel programs in Fortran to solve their numerical problems.

The book is full of examples. Most of the examples have been tested with Intel Corporation's Fortran compiler, ifort, version 19.3, GCC gfortran version 7.3.0 and the Fortran compiler version 6.2 of Numerical Algorithm Group (NAG). The Fortran part is based on the draft Fortran 2018 report published on July 6, 2017. At the time of writing, these compilers do not support all the proposed features of Fortran 2018, but Intel, Free Software Foundation, Inc., and NAG will add further support for these features over time.

Subrata Ray

Acknowledgments

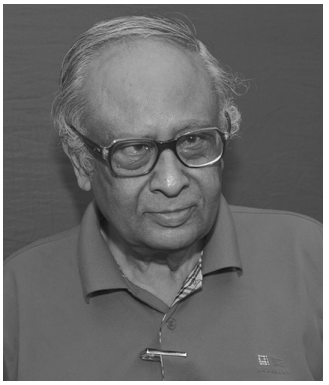
The author wishes to record his deep sense of gratitude to his colleagues, friends and associates who helped him to prepare this manuscript during the various phases of this work.

Abhijit Kumar Das
Ananda Deb Mukherjee
Ankush Bhattacharjee
Ashish Dutta
Biplab Sarkar
Debasis Sengupta
Gayatri Pal
Indrajit Basu
Indrani Bose
Koushik Ray
Minakshi Ghosh
Prasanta Kumar Mukherjee
Pushan Majumdar
Ramaprasad Dey
Ranjit Roy Chowdhury
Robert Dyson
Sandip Ghosh
Sankar Chakravorti
Santosh Kumar Samaddar
Sarbani Saha
Satrajit Adhikari
Satyabrata Roy
Siddhartha Chaudhuri
Souvik Mondal
Swapan Bhattacharjee
Utpal Chattopadhyay

The Numerical Algorithms Group Ltd., Oxford, UK, provided the author with a free license to use their Fortran compiler. Intel Corporation allowed the author to use the trial version of their Fortran compiler ifort. The free GCC gfortran compiler was also used. National Council of Education, Bengal, and Institute of Business Management, NCE, Bengal, allowed the use of their computer laboratory during the preparation of this book.

Finally, the author wishes to thank his family for their encouragement during the preparation of this manuscript.

Author



Dr. Subrata Ray is a retired senior professor of the Indian Association for the Cultivation of Science, Kolkata. In his career spanning over 40 years, he has taught computer software in universities, research institutes, colleges and professional bodies across the country. As a person in charge, he has set up several computer centers, in many universities and research institutes, almost from scratch. Though his field of specialization is scientific computing, he has participated in developing many systems and commercial software.

He has an MSc, a Post MSc (Saha Institute of Nuclear Physics) and PhD from Calcutta University and has served several renowned institutes like Tata Institute of Fundamental Research, Indian Institute of Technology, Kharagpur, Regional Computer Centre, Calcutta, University of Burdwan and Indian Association for the Cultivation of Science.

He is associated with the voluntary blood donation movement of the country and is an active member of Association of Voluntary Blood Donors, West Bengal. He offers his voluntary services to Eye Care & Research Centre and National Council of Education, Bengal.

He is also an amateur photographer.

He is married to Sanghamitra, and they have a daughter, Sumitra. He lives in Kolkata with his brother Debabrata and sister Uma.

1

Preliminaries

A computer is a machine that can perform basic arithmetic operations at a very high speed. It can take logical decisions and select alternative paths depending upon the program logic. It can communicate with the external world via its input/output devices. To get a job done, the computer needs to be instructed through a computer program. A computer program is a set of instructions through which one instructs a computer to perform a specific job. The computer's processor understands only one language, called the machine language. The machine language is machine dependent and is difficult for humans to learn. To circumvent this difficulty, several artificial languages (sometimes called high-level languages) have been developed. These artificial languages are very easy to learn and are practically machine independent. However, this requires translation to the machine language of the particular machine. The translation is done by the computer itself through a system program called a compiler. The compiler, while translating, checks the grammar of the language; if the source program is free from grammatical errors, it generates the machine language version of the *source program*, called the *object program* for the machine, which is subsequently linked (using a system program called the linker) to various libraries of the system. The resultant code, called the *executable code*, is executed by the machine. As different machines use different machine languages, the compilers are naturally machine dependent. Therefore, that a particular machine can *execute* a program written in a high-level language implies that the compiler for that high-level language is available to the computer system.

Fortran—one such programming language—is the abbreviation of **For**mula **tran**slation. It is widely used in solving scientific and engineering problems that require a lot of numerical computation. In this book, Fortran stands for Fortran 2018, the current version of Fortran.

It must be mentioned at this point that no computer can directly execute any program written in Fortran or any other, so-called high-level, language like Fortran. The compiler for the corresponding language must be available to the computer so that the translated version of the program written in a high-level language may be executed by the computer. As this translation—Fortran to machine language of a particular machine—is transparent to the programmer, one may assume that the computer is executing the Fortran program.

The compiler generates an object program only when the source is free from grammatical errors. In case of any grammatical error being flagged by the compiler, the programmer has to go back to the source, make the necessary correction(s) to the

source and recompile the source to get the object program. The object program will not be generated until all grammatical errors are removed from the source program.

A program, free from grammatical errors, may not give a correct result. The program must be free from *logical errors*. A logical error is an error in the program logic at the source level. For example, a particular program requires addition of two numbers, but the programmer, by mistake, has performed multiplication instead of addition. A *runtime error* may occur during the execution of the program. Suppose a program has to divide two numbers. The division process is valid so long as the second number (denominator) is not zero. Division by zero is not a valid arithmetic operation. This error will show up during the execution of the program should the denominator become zero. The program behaves normally so long as the denominator remains nonzero.

Therefore, to obtain a correct result from a program, the following three conditions must be satisfied:

- The program must be free from grammatical errors.
- The program must be free from logical errors.
- There should not be any runtime error.

In this book, we frequently use the term processor. According to the Fortran report, a processor is a combined object, consisting of software (compiler, operating system, etc.) and hardware, that converts a Fortran program into its machine language equivalent and executes the same.

1.1 Character Set

The programming language and its syntax are described by a set of characters. The character set that is available to a Fortran programmer consists of (a) all letters of the English alphabet, both uppercase (A–Z) and lowercase (a–z); (b) underscore character (_); (c) all digits (0–9); (d) several special characters, such as brackets, colon and full stop; and (e) several unprintable characters, such as tab, linefeed and newline characters.

Some characters may appear only within comments, character constants, input/output records and edit descriptors. The English letters, numerals and the underscore character are collectively called alphanumeric characters.

Normally, Fortran is case insensitive; that is, it does not distinguish between the uppercase and lowercase letters. There are, however, exceptions (character strings and input/output). In addition, specifiers like file names in `open` and `inquire` statements (Chapter 9) may make it necessary to distinguish between lowercase and uppercase letters. This is processor dependent and will be discussed at appropriate places in the text. Table 1.1 shows the list of special characters.

In this book, we consider only ASCII set of characters. ASCII, abbreviation of American Standard Code for Information Interchange, is an industry standard for electronic communication. The processor may also support other types of character sets.

TABLE 1.1
Lists of Special Characters

Character	Name	Character	Name
	Blank	;	Semicolon
=	Assignment (equal)	!	Exclamation sign
+	Plus	"	Quote
–	Minus	%	Percent
*	Asterisk	&	Ampersand
/	Slash	~	Tilde
\	Back slash	<	Less than
(	Left parenthesis	>	Greater than
)	Right parenthesis	?	Question mark
[	Left square bracket	'	Apostrophe
]	Right square bracket	`	Grave accent
{	Left curly bracket	^	Circumflex accent
}	Right curly bracket	|	Vertical line
,	Comma	$	Currency sign
.	Decimal point	#	Number sign
:	Colon	@	Commercial at

1.2 Identifiers

Identifiers are used to specify various objects permitted by the language. An identifier (a) must start with a letter of the English alphabet, (b) may contain other digits and letters or the underscore character, (c) must not contain any special characters or blanks and (d) must have a length not exceeding 63 characters. It is obvious that the identifier must comprise at least one character, and in that case, it must be a letter of the English alphabet only. The first character cannot be an underscore character. However, the last character can be an underscore.

The following are valid identifiers:

```
xmax, counter, basic_pay, i, volt, name__of_a_person
```

(Note two successive underscore characters after name.)

The following are invalid identifiers:

```
1xy (starts with a digit)
x)y (contains a special character)
x min (contains a blank character)
```

The identifiers `ABC`, `Abc`, `aBC` and `abC`, or any combination of uppercase and lowercase `A`, `B` and `C`, are equivalent, as Fortran normally does not distinguish between the uppercase and the lowercase letters.

Usually, identifiers are so chosen that they have some relation with the actual objects they refer to. For example, the identifier `volt` is a natural choice for denoting the voltage of an electrical circuit. One can as well choose `z43p8` to represent voltage; however,

it appears to be a poor choice because the readability of the program is diminished once such a choice is made. Needless to say, the length of the identifier must be of reasonable size. Though the language permits `63` characters to represent an identifier, rarely more than `8` or `10` characters are used to represent an identifier. Unnecessarily long identifiers will invite typing errors and perhaps reduce the readability of the program.

Several characters like (`2` and `Z`), (`1` and `I`) and (`O` and `0`) look similar. Therefore, care should be taken while using similar characters within the same identifier. For example, identifiers like `O0` (oh zero) should be avoided. One may invite further trouble if one chooses another identifier `OO` (oh oh) in the same program unit. It must be understood that, for the Fortran compiler, both `O0` (oh zero) and `OO` (oh oh) are valid but different identifiers—it is the human programmer who may mix up these two similar-looking identifiers.

1.3 Intrinsic Data Types

There are five types of intrinsic data in Fortran. They are `integer`, `real`, `complex`, `logical` and `character`. The `integer`, `real` and `complex` types are used for numeric computation; `logical` and `character` are nonnumeric data types. In addition, there are extended precision real data types, known as `double precision` and `quadruple precision` (not a standard Fortran—Intel's extension) and `double precision complex`. All intrinsic data types are associated with a `kind` parameter. The `kind` parameter may be explicit, or if the `kind` parameter is not present, the intrinsic data assume the `default kind` parameter. The intrinsic type `character` is usually associated with a `len` parameter, which determines the length of the character string. The `kind` parameter is discussed in Chapter 5.

1.4 Constants and Variables

In any programming language, we normally use two types of objects—constants and variables.

A quantity whose value remains fixed during the execution of a program is called a constant. The compiler identifies the constant—its type and value—from its appearance. Constants do not have any name associated with them. In other words, a constant conveys both its type and its value to the compiler. On the other hand, a variable may change its value during the lifetime of a program. A variable must have a name attached to it. It may also remain undefined during the lifetime of a program.

As there are five types of intrinsic data (plus two extended types) in Fortran, there are five types (plus two extended types) of intrinsic constants and variables in Fortran.

1.5 Integer Constants

An integer constant is a whole number; that is, it does not contain any decimal point. It contains only digits and a leading sign, if necessary. An integer constant may be positive, negative or zero. Zero is neither a positive nor a negative number. In addition, the numerical values of `0`, `+0` and `-0` are same. Negative constants are prefixed by a minus sign, and positive constants may optionally be prefixed by a plus sign. Unsigned integer constants

are assumed to be positive. For instance, constants `10` and `+10` are equivalent. Normally, a leading positive sign is not used, as it is optional. The integer constant, say, `127`, tells the compiler its type; in this case it is an integer, and its magnitude is `127`. The maximum and minimum values that an integer constant may assume are processor dependent. The typical values are `2147483647` (maximum: `2` to the power `31` minus `1`) and `–2147483648` (minimum: minus `2` to the power `31`). By default, integer constants are treated as decimal numbers (base `10`). Valid integer constants are `2, 35, –7432, 12345, 0` and `–4321`.

Invalid integer constants are as follows:

```
2.0 (contains decimal point)
37- (negative sign is not prefixed)
12345678901234 (most probably exceeds the capacity of the processor but
                might not at some point in future)
2a3 (contains a non-digit character)
```

Leading zeros of an integer constant are ignored. For example, `01, 001, 00001` and `1` are all equivalent.

1.6 Real Constants

A real constant is a real number containing one and only one decimal point. The decimal point may be explicit or implicit (possible in scientific notation—to be discussed shortly). A real constant may be positive, negative or zero. A real negative constant is prefixed by a negative sign. An unsigned real constant is assumed to be positive. The numerical values of `+0.0` and `–0.0` are same even if the processor can make distinction between `+0.0` and `–0.0`. Like for an integer constant, a leading plus sign for a positive real constant is optional. A real constant, in the standard form, contains digits, one decimal point and is prefixed by a plus or minus sign, if necessary. Valid real constants are `3.1415926, –25.3456, 12345.678` and `0.0`.

Invalid real constants are as follows:

```
36 (contains no decimal point)
35.3.2 (contains more than one decimal point)
3a3563.25 (contains a letter 'a')
45.2(2) (contains special characters)
5 6.0 (contains a blank)
```

If there is no digit before or after the decimal point, zero is assumed. For example, `2.` is treated as `2.0`. Similarly, `.25` is a valid real constant, which is same as `0.25`. The use of real constants like `2.` and `.25` is strongly discouraged. This reduces the readability of the program. These numbers should be written as `2.0` and `0.25`, respectively. It is needless to mention that just a decimal point does not represent any real constant, that is, not `0.0`. The maximum and minimum values and the precision of the real number are processor dependent. A real number may be written with more digits after the decimal point than the number of digits the processor can support. The excess digits are not considered by the system. Real constants may also be represented by powers of `10`, known as scientific form. This makes it very convenient to express very small or very large numbers. In this form, a real constant consists of two parts: an integer or real number followed by an exponent. The exponent is denoted by letter `'E'` or `'e'`. The number `1.24e4` is actually 1.24×10^4,

as 'e4' stands for 10 to the power 4. There is no space between 'e' and the real or the integer part (also called fractional part). The exponent must be an integer and may be signed. The sign is placed after the exponent symbol. Unsigned exponents are assumed to be positive. Valid real constants in scientific notation are 1.24e4, -111.90e10, 77.345e-3 and -123.345e-5. The values of the above real constants are, respectively, 1.24×10^4, -111.90×10^{10}, 77.345×10^{-3} and -123.345×10^{-5}.

Invalid real constants are as follows:

```
1.24e4- (wrong position of the minus sign)
3.25 e5 (space between the fraction and the exponent symbol)
777.24e2.5 (exponent must be an integer)
6.935e500 (probably exceeds the capacity of the processor)
```

As mentioned earlier, the decimal point in a real constant may be implicit. For example, 123e4 is a real constant, though it does not contain any decimal point. The default decimal point is assumed between '3' and 'e'. It is, thus, equivalent to 123.0×10^4. Leading zeros of the fractions are ignored. For example, 0123.24e4, 00123.24e4 and 123.24e4 are all equivalent. Similarly, leading zeros in the exponent are also ignored. The real constants 2.345e+2, 2.345e+02, 2.345e02 and 2.345e2 are equivalent.

1.7 Double Precision Constants

Double precision quantities are real numbers but are more precise than their single precision counterparts. It is well known that the computer internally uses binary numbers, and all decimal numbers do not have an exact binary representation. For example, when 0.1 is converted to binary, the binary number may not be exactly 0.1—it is probably 0.09999.... A computer is a finite bit (binary digit) machine, and the number of bits used determines how close the binary number is to the decimal counterpart. For infinite precision arithmetic, the two numbers would have been identical. By increasing the number of bits to store a real number, the binary counterpart can be made closer to the decimal value. Double precision numbers take more memory than do the single precision numbers and consume more central processing unit (CPU) time to perform any arithmetic operations compared with the corresponding single precision numbers.

Double precision constants are more precise than the corresponding single precision values. Such a constant is expressed in scientific notation. To indicate ten to the power for double precision constants, 'D' or 'd' is used; π, correct up to 15 decimal places, is written as 3.141592653589793d0.

The rules discussed in connection with the single precision real numbers (in scientific notation) are equally valid for the double precision quantities.

1.8 Complex Constants

Complex numbers are widely used in science and engineering. A complex number consists of two parts—real and imaginary. The Fortran compiler can handle complex numbers according to the rules of the complex algebra. The real and the imaginary parts may separately be positive, negative or zero.

Both the real and the imaginary parts are integers or real numbers. The real and the imaginary pair are enclosed in brackets.

Valid complex constants are as follows:

```
(2.0, 5.0), (-3.373, 5.397), (-4467.23, 891.45)
(3.124E2, -4.935E-7), (33, 25), (-15.72, -21)
```

Since the real and the imaginary parts are either integers or real numbers, the discussions regarding real and integer numbers are equally valid for the real and the imaginary parts, respectively.

1.9 Double Precision Complex Constants

Double precision complex constants have double precision quantities as the real and the imaginary parts. Valid double precision complex constants are as follows:

```
(1.23456789d0, 9.87654321d0), (-3.14159265d0, 1.987665432d0)
```

1.10 Quadruple (Quad) Precision Constants

Intel's Fortran compiler, ifort, supports quadruple precision real numbers. These numbers are more precise than the corresponding double precision numbers. This is not a standard Fortran feature. Quadruple precision constants have a precision of `33` places after the decimal. To indicate quadruple precision, `'q'` (or `'Q'`) is used in scientific notation in place of `'e'` or `'d'`. Valid quadruple precision constants are as follows:

```
3.14159265358979323846264338327950 2q0,-7.9876543219876543219876543211 2345q1
```

1.11 Logical Constants

Integer and real constants can assume innumerable values. There are only two logical constants: `.true.` and `.false.`. These constants are bound by two periods (uppercase letters may be used).

1.12 Character Constants

Character constants are zero or more characters enclosed in quotes or apostrophes. Examples of character constants are `'A'`, `'ABC'` and `'West Bengal'`.

The delimiters (apostrophe or quote) are not part of the string. Note that a blank is also a character (blank between `West` and `Bengal`). The characters `'A'` and `'a'` are not same. It is case sensitive. In addition, character `'1'` and number `1` are different, and they are stored in different ways inside the machine. Conventional arithmetic operations are not permitted with character constants like `'1'`. To represent the apostrophe as a character constant, either two successive apostrophes are used or it is enclosed in quotes: `'don''t', "don't"`. Similarly, to represent the quote as a character constant, either two successive quotes are used or it is enclosed in apostrophes: `""""', '"'`. Null is represented as two successive apostrophes (or quotes) with nothing in between. Any graphic character supported by the processor can also be part of a character constant.

Although Fortran supports other types of character sets, in this book only the ASCII character set is considered. All these character sets have one thing in common: all of them have one blank character.

1.13 Literal Constants

The constants mentioned earlier are also known as literal constants. Literal constants do not have names attached to them.

1.14 Variables

An object whose value may vary during the execution of a program is called a variable. A variable can store only one value at a time, which may change during the execution of the program. A variable must have a name attached to it. A variable is identified by its name, type and value. If no value is assigned to it, it remains unassigned or undefined. Note the word *may* in the definition. The variable may or may not change its value during the execution of the program and may even remain undefined during the lifetime of the program.

By default, if the variable name starts with `i, j, k, l, m` or `n` (or uppercase letters), it is an integer variable. All other variables that start with letters other than `i–n` are real variables. An integer variable can store only an integer quantity, and similarly, a real variable can store only a real quantity. In spite of the preceding default rule, it is a good programming practice to define each variable explicitly. The modern programming practice is to switch off this default feature, that is, `i–n` rule, by appropriate declaration, and in that case, it is mandatory to declare each variable.

1.15 Variable Declarations

An integer variable can store only an integer quantity. It is declared as follows:

```
integer :: a
integer :: b
integer :: c, d
```

In the preceding declarations, `a, b, c` and `d` are declared as integer variables, and therefore, these variables can store only integer quantities. It is apparent from these declarations that more than one variable may be declared by a single declaration—in that case, the variables are separated by a comma. The first two declarations may be combined as follows:

```
integer :: a, b
```

Blanks between `integer` and `'::'` and between `'::'` and the variable name can be introduced to increase the readability.

An identifier (variable name) may be treated like a box. The name of the box is the name of the variable. The content of the box is undefined. When a value is assigned to a variable, the content of the box is the value of the variable.

In an identical manner, other intrinsic types are defined.

```
real :: x
real :: p, q
double precision :: d1
double precision :: d3, d4
complex :: c
complex :: z, y
double complex :: dc1
double complex :: dc2, dc3
character :: ch
character :: e, f
logical :: l3, l4
```

By default, the number of characters that a character variable can store is 1; that is, the length is 1. A character variable may store more than one character if it is declared in an appropriate manner.

```
character (len=10) :: ch
```

The variable `ch` can store `10` characters. The length can also be specified as `character *10 ch`. Also, `character(4) :: ch` and `character(len=4) :: ch` are equivalent. In this case, just an integer without `len=` is assumed to be the length of the string.

1.16 Meaning of a Declaration

A declaration is a placeholder for a variable; it merely reserves location(s) for a variable and defines the type of the variable. No value is assigned to the variable. A suitable Fortran statement must be used to assign a value to a variable. A variable can store only one value at a time. The same variable cannot be declared more than once in a program unit. For example, the declarations

```
integer :: x
real :: x
```

within the same program unit will give rise to Fortran error because the variable `'x'` cannot be an integer and a real variable at the same time. Unassigned variables should not be used, as the result of such computation is unpredictable.

1.17 Assignment Operator

The assignment operator `(=)` is used to assign a value to a variable. For example, if a variable `first` is declared as integer, the variable `first` is assigned to a value in the following manner:

```
integer :: first
...
first = 10
```

Subsequently, if `20` is assigned to `first`, the old value `10` will be lost and now `first` will contain `20`. The instruction `first = 20` assigns `20` to `first;` the old value `10` is lost.

We will discuss this assignment operator in detail in Chapter 2.

1.18 Named Constants

A symbolic name may be attached to a constant. The symbolic name becomes an alias for the constant. The alias behaves just like a literal constant, and it cannot be modified during the execution of the program. A true constant, say, π, may thus be used in this manner.

```
real, parameter :: pi=3.1415926
```

In the preceding declaration, `pi` is the symbolic name of `3.1415926` because of the presence of the attribute `'parameter'` with the real declaration; a comma separates `real` and `parameter`. In the preceding declaration, `pi` is not a real variable—it is just another name of `3.1415926`. During compilation, `3.1415926` will replace each occurrence of `pi`. Since `pi` is alias of `3.1415926`, `pi` cannot be assigned to a different value; `pi=4.25` is not allowed as named constants by definition cannot be modified. The reason is not difficult to guess. During compilation, `3.1415926` will replace `pi`, so the statement `pi=4.25` will become

```
3.1415926 = 4.25
```

which is clearly not a valid Fortran statement.

Moreover, the program unit cannot have any variable named `pi` as `pi` has already been declared as an alias for `3.1415926`. The following will generate compilation error:

```
real, parameter :: pi = 3.1415926
integer :: pi
```

It is a good programming practice to assign a symbolic name to a true constant like π so that even by mistake the constant cannot be modified during the execution of the program. An alternative way to represent a named constant is through the parameter statement:

```
parameter (named constant=value,...)
```

Example: `parameter (pi=3.1415926)`

Either the type of the named constant, declared by the parameter statement, is declared or it follows the default `i–n` rule. For example, in the case of `parameter (ip=2.3)`, each occurrence of `ip` is substituted by `2` and not `2.3` since, without any declaration, `ip`, being an integer can store only an integer quantity. Therefore, truncation will take place. A single parameter statement may define more than one named constant: `parameter (pi=3.1415926, e=2.303, lpt=6)`.

Named constants are assigned values at the time of compilation. Therefore, it cannot contain anything whose value is not known during compilation. In an identical manner, other intrinsic type constants can be attached to a symbolic name.

```
integer, parameter :: limit=100
double precision, parameter :: dpi=3.1415926589793d0
complex, parameter :: zpar=(10.0,30.0)
logical, parameter :: l3=.true.
character, parameter :: start= 'a'
```

For a character named constant, an asterisk may be used as the length of the named constant; the compiler from the declaration can find out the length of the named constant (allocates locations to store the constant):

```
character (len=*), parameter :: city= 'kolkata'
```

From the declaration, the compiler can ascertain that the named constant `city` should have a length `7` to accommodate the string `'kolkata'` and allocates locations accordingly.

The preceding declaration is same as `character (len=7), parameter :: city= 'kolkata'`.

1.19 Keywords

Fortran contains several keywords like `integer` and `real`. However, the keywords are not reserved words and may be used as identifiers. This is strongly discouraged. For example, `'do'` is a Fortran statement and also a Fortran keyword. It is permitted to have an identifier named `do`. The compiler will identify the `do` statement from its appearance; it will also correctly treat the `do` identifier. However, for the sake of readability, this should be avoided. These types of keywords are called statement keywords. Normally, a keyword cannot have embedded space in free form. The keyword, say, `'read'` cannot be written as `'re ad'`. However, this is allowed in fixed form (not discussed in this book). If a name follows a keyword, the keyword and the name must be separated by a blank. Blank is optional for some single keywords that consist of two

TABLE 1.2

Adjacent Keywords

Blanks Are Optional		
`block data`	`end file`	`end team`
`double complex`	`end forall`	`end type`
`double precision`	`end function`	`end where`
`else if`	`end if`	`error stop`
`else where`	`end interface`	`go to`
`end associate`	`end module`	`in out`
`end block`	`end procedure`	`select case`
`end block data`	`end program`	`select rank`
`end critical`	`end select`	`select type`
`end do`	`end submodule`	
`end enum`	`end subroutine`	
Blanks Are Mandatory		
`case default`	`interface assignment`	`recursive subroutine`
`do while`	`interface operator`	`recursive` *`type-spec`*
`implicit` *`type-spec`*	`module procedure`	*`type-spec`* `function`
`implicit none`	`recursive function`	*`type-spec`* `recursive`

keywords, such as `'end do'`. In this case, `enddo` and `end do` are the same. However, blank is mandatory for keywords like `do while` and `implicit none`. Table 1.2 lists such adjacent keywords where blanks are optional and mandatory.

Argument keywords are discussed along with subprograms. Keywords are also used to identify an item within a list. They are used as `keyword=value` so that the position of the keyword within the list is not important.

1.20 Lexical Tokens

If a Fortran statement is broken into basic language elements, the smallest meaningful objects are called lexical tokens. The lexical tokens consist of names, operators, literals, keywords, labels, assignment signs, commas, etc. In other words, a Fortran statement is a combination of lexical tokens. Names, constants and labels are usually separated from adjacent lexical tokens by one or more blanks or an end of line. For example, a+b consists of three lexical tokens: a, b and +.

1.21 Delimiters

A delimiter consists of a pair of symbols, which determines a part of a Fortran statement. Examples of delimiters are

```
/....../
(......)
[......]
(/....../)
```

These are discussed at appropriate places in the text.

1.22 Source Form

A Fortran source program consists of one or more lines. A line may contain zero or more characters. Fortran statements may be written in two different forms: (a) fixed form and (b) free form. The current trend is to write programs in free form; therefore, fixed form is not discussed in this book.

1.23 Free Form

In free form, a Fortran statement can be extended to `132` characters per line if characters of default kind are used. However, if the line contains characters other than the default kind, the number of characters that a line can accommodate is processor dependent. The statement may start anywhere within this field. A line is usually divided into several fields: (a) statement number field, (b) statement field, (c) comment field and (d) continuation field. A line may not contain all fields; even a line may be totally empty. If the last non-blank character of a particular line is `'&'`, the next line (if it is not a comment line) is considered as the continuation of the previous line. A total number of `255` continuation lines are allowed (per statement). Note that `'&'` character is not a part of the statement when used as a continuation character.

Figure 1.1 is equivalent to `X=Y+Z`. No line can contain a single `'&'` as the only non-blank character. In addition, no line can contain one `'&'` character followed by `'!'` character. A statement number, if any, should be placed at the beginning of the line. There must be a blank or a tab character after the statement number. There may be any number of blanks before the statement number. A statement may have a statement number between `1` and `99999`. Leading zeros of the statement numbers are ignored. No two statements in a program unit can have the same statement number (one exception is discussed in Chapter 12). It is not required to assign a statement number to every statement. However, there are statements that must have a statement number. The statement number is used when a statement is required to be referred by other statement(s). If a line is continued, only the first line can have a statement number. There cannot be any blank within the statement number (Figures 1.2 through 1.4).

If the character `'!'` is typed anywhere in a line, the rest of the line, except within a character string, is treated as a comment (Figure 1.5). Comments are used for documentation. The compiler does not try to translate the comment. Comments can be

1	2	3	4	5	6	7	8	9	10	11	12
						X	=	Y	+		&
						Z					

1	2	3	4	5	6	7	8	9	10	11	12
1	4	7				X	=	Y	+	Z	

1	2	3	4	5	6	7	8	9	10	11	12
			1	0	5		X	=	Y	+	z

1	2	3	4	5	6	7	8	9	10	11	12
1		2	5		X	=	Y	!	e	r	r

1	2	3	4	5	6	7	8	9	10	11	12
5		A	=	2	!	c	m	t			

FIGURE 1.1 through 1.5
(see text).

placed anywhere within the program unit; it may be placed before the first statement. It may also be placed after the last statement. It may be placed between two continuation lines. Comments placed between continuation lines do not contribute to the calculation of continuation lines.

A comment line cannot have a statement number (gfortran gives a warning):

```
100 ! This is a comment
```

This is not a valid statement. Furthermore, a comment line cannot be continued.

A blank line is treated as a comment. In fact, judicious use of blank lines increases the readability of the program. If the continuation symbol `'&'` is typed after the comment character `'!'`, the character `'&'` becomes a part of the comment and is not considered a continuation character (Figure 1.6). This will generate compilation error; `'&'` is not considered as the continuation character in this case, so the next line is not treated as continuation of the previous line. A comment may appear after the line continuation character:

```
a=b+ & ! This is a comment
c
```

1	2	3	4	5	6	7	8	9	10	11	12
						X	=	Y	+	!	&
							Z				

1	2	3	4	5	6	7	8	9	10	11	12
						X	=	Y	+	&	
	Z										

1	2	3	4	5	6	7	8	9	10	11	12
						X	=	Y	+	&	
&	Z										

FIGURE 1.6 through 1.8 (see text).

Normally, continuation starts from the first character of the next non-commented line. However, if it is necessary to start the continuation from a particular position of the next line, then the `'&'` character must be typed just before the desired character of the next line.

The statement where the continuation line starts with a blank (Figure 1.7) will be treated as follows:

```
X=Y+ Z (one blank between '+' and Z)
```

On the other hand, where the continuation line starts with `'&'`, the statement will be treated as follows (Figure 1.8):

```
X=Y+Z (no blank between '+' and 'Z')
```

In these situations, both mean the same, as normally Fortran ignores blanks. This effect will be felt when character strings are used where the presence or absence of a blank within a character string may result in a different meaning.

In free form, there cannot be any imbedded blanks within a lexical token. Blanks are used to separate various items, such as names, constants and labels, from the keywords; `read 20, a, b, c` cannot be written as `read20, a, b, c`.

However, multiple blanks may be used between tokens. This may improve the readability of the source code. Most of the times, multiple blanks between tokens are treated as a single blank.

In free form, a normal statement (a statement without continuation) terminates when either `'!'` character or the end of line is reached. Each compiler has a mechanism to identify the free and the fixed form. This is done in two ways. The first method is to use an appropriate compiler directive, say, `-free` or `-fixed`. The second method is to use the file extension. A file having extension `.f` (say, `a.f`) is considered in fixed form, and a file having extension `.f90` (say, `a.f90`) is considered in free form.

1.24 Continuation of Character Strings

Normally, Fortran ignores blanks. A blank is considered as a character within a character string. Therefore, special consideration is needed for continuing a character string to the next line. For this purpose, an ampersand character `('&')` must be the last non-blank character of the first line of the character string, and each continuation line must have an ampersand character. Continuation begins from the character following the ampersand of the continuation line. Consider the following (Figure 1.9).

```
character(len=80) :: ch = 'ASSOCIATION OF VOLUNTARY &
'ASSOCIATION OF VOLUNTARY &
&          BLOOD DONORS'
```

FIGURE 1.9
Continuation of Character String.

The variable `ch` is initialized along with its declaration. Lines `2` and `3` are continuation of line `1`. We now consider lines `2` and `3`. As the continuation starts from the first character of a line (in this case line `3`), six blanks will be added before the string `'BLOOD DONORS'`. The character variable `ch` will be initialized to

`"ASSOCIATION OF VOLUNTARY          BLOOD DONORS"` (6 blank characters), and the system will add the required number of trailing blanks. However, if the intention of the programmer is to initialize the variable to `"ASSOCIATION OF VOLUNTARY BLOOD DONORS"`, that is, only one blank between `'VOLUNTARY'` and `'BLOOD'`, the third line should have an ampersand character as shown next (Figure 1.10).

```
character(len=80) :: ch = 'ASSOCIATION OF VOLUNTARY &
'ASSOCIATION OF VOLUNTARY &
&BLOOD DONORS'
```

FIGURE 1.10
Continuation of Character String.

The ampersand in the third line ensures that continuation starts from the character following the ampersand, which is just a blank in this case.

If a keyword or other attributes of the language (technically called token) are split across the line for which no embedded blank is allowed, there should not be any space between the ampersand and the rest of the token in the continued line as shown in the following:

```
re&
&ad *, x
```

The preceding code is treated as `read *, x`. Note that the position of the ampersand in this case ensures that there is no space between `'re'` and `'ad'`. A character constant may contain an ampersand. The last one is considered the continuation character, as shown next. If it contains more than one ampersand, the other ampersands become the part of the character string. For

```
print *, 'M/s Roy & Chatterjee & &
  &Gupta'
end
```

the output is `M/s Roy & Chatterjee & Gupta`. The ampersand shown in bold is considered as the continuation character. The ampersand before the continuation character is a part of the character string.

1.25 Structure of a Program

A program unit contains one or more lines, which comprises the Fortran declaration, statement, comment and included line. A line may contain zero or more characters. The `end` statement terminates a program unit. A program may contain more than one program unit. An executable unit must have one and only one program unit called the main program. Execution always begins from the first executable statement of the main program. In addition to the main program, an executable unit may have external subprograms, internal subprograms, modules, submodules and block data (now declared as obsolete). These topics are discussed at appropriate places. For the time being, we shall consider only one program unit; that is, the executable unit would contain only the main program.

1.26 IMPLICIT NONE

It was mentioned earlier that if the variables are not declared explicitly, Fortran applies certain default rule (`i–n` rule) in selecting the variable type. This default rule can be switched off by placing `implicit none` at the beginning of the program unit. In this case, all variables are to be declared explicitly. If `implicit none` is present in a program unit, the unit cannot have any other `implicit` statement. For example, if a variable `i1` (`i` and `1`) is declared as an integer and if it is typed as `ii` (`i` and `i`) in the body of the program

(typing error), `implicit none` will force the compiler to generate a Fortran error (undefined variable). If `implicit none` is absent, it will be treated as another integer variable following the default `i–n` rule, and since it is undefined (no value is possibly assigned), the result is unpredictable. The modern trend of programming is to use `implicit none` in every program unit so the programmer is forced to declare all variables explicitly. Any typing error, similar to that shown above, will be flagged as an error at the compilation stage.

1.27 IMPLICIT

This statement can be used to treat variables that start with certain letter to be of a particular type.

```
implicit integer (a)              ! (1)
implicit integer (b, c)           ! (2)
implicit integer (d-f)            ! (3)
implicit integer (g-i, x-z)       ! (4)
```

Statement (1) directs the compiler to treat all variables that start with `'a'` as integers. That is, the compiler will treat `am, ax1, ap,` etc., as integers. Statement (2) tells the compiler to assume the variables that start with `'b'` or `'c'` to be integers. Statement (3) contains `d-f,` which is equivalent to `implicit integer (d, e, f)`. Statement (4) states that all variables that start with `g, h, i, x, y` and `z` are integers. The dash sign (minus sign) indicates a set of contiguous letters; the first and the last letter in the set are placed on the left and the right of the dash sign. The discussion of this section is equally applicable to other types of variables. So it will not be repeated again.

```
implicit real (a)
implicit real (x-z)
implicit real (a-h, o-z)
implicit double precision (d)
implicit double precision (e, f)
implicit complex (p-q)
implicit logical (m-n)
implicit double complex (x-z)
implicit character (r, s)
```

For a character variable, if the length parameter is not present along with the `implicit` declaration, the length is assumed to be equal to `1`. The length parameter can be specified along with the implicit declaration:

```
implicit character (len=4) (u-v)
```

This declaration needs some explanation. It states that variables that start with `u` or `v` are character variables of length `4`; that is, they can store four characters.

1.28 Rules of IMPLICIT

1. If a program unit contains `implicit none`, it must not contain any other `implicit` statement.
2. A program unit cannot contain two implicit statements with the same letter:

```
implicit integer (c)
implicit real (c)
```

or

```
implicit integer (c-f)  ! this includes d
implicit real (d)
```

are not valid.

3. An explicit declaration overrides an `implicit` declaration.

```
implicit integer (i)
real :: i
```

Here, `'i'` will be treated as a real variable because of the explicit real declaration.

1.29 Type Declarations

Fortran allows declaring intrinsic variables through `type` declarations.

```
    integer :: a
```

and

```
type(integer) :: a
```

are equivalent.

Similarly, real, complex, double precision, double complex, logical and character variables may be declared through `type` declarations as shown in the following:

```
type(real) :: b
type(double precision) :: c
type(complex) :: d
type (double complex) :: e     ! allowed in ifort, not allowed in NAG
type (logical) :: l
type (character) :: ch
implicit type(integer) (a-h, o-z)! allowed in NAG, not in ifort
```

and

```
implicit integer (a-h, o-z)
```

are equivalent.

1.30 Comments on IMPLICIT Statement

Readers must have noticed that the discussion related to `implicit none` goes against the declaration `implicit integer / real / double precision / complex / character / logical / double complex`. A guideline may be formulated. `Implicit none` is certainly very safe; it isolates all undefined variables and helps to eliminate most of the typing errors related to variable names. On the other hand, `implicit integer` saves a lot of typing, especially if the program unit contains many integer variables. Some programmers use the first letter of variables to indicate the type of variables. For example, one may choose `'c'` as the first letter for all complex variables and `'d'` as the first letter for all double precision variables. In such a situation, `implicit double precision(d)` and `implicit complex(c)` are convenient. Therefore, it is a matter of choice. The present author prefers `implicit none`, and he feels that if more time is spent during the development phase of the program (that is, coding and typing), then debugging time is substantially reduced and the problems mentioned related to the undefined variables never crop up.

1.31 PROGRAM Statement

The optional `program` statement is the first statement of a program. It supplies the name of the program:

```
program my_first_program
```

where `my_first_program` is the name of the program.

1.32 END Statement

The `end` statement signifies the end of a program unit. It may contain a program and the name of the program. However, this is optional.

A typical Fortran program is of the following form:

```
program my_first_program
.
end program my_first_program
```

Once the `'end'` statement is reached, the compiler starts compiling that particular unit. The `end` statement terminates a program unit. If the `end` statement contains the program name, the corresponding program name must be present with the `program` statement. In fact, it is always better to use the name of the subroutine, module and

function along with the `end` statement. Each program unit, module subprogram and internal subprogram can have only one `end` statement. Such `end` statements are executable statement, and it is possible to jump to the statement by suitable statements (`goto, if`—Chapter 3). The execution of the `end` statement in the main program terminates the job. The execution of the `end` statement within a function or subprogram or subroutine subprogram or separate module subprogram is equivalent to a `return` statement (Chapter 12). The `end` statement of a module, submodule and block data is a non-executable statement.

1.33 Initialization

A variable may be initialized to a value along with its declaration. In this case, when the execution begins, the corresponding variable is not undefined; it has an initial value.

```
integer :: a=10
real :: x=1.34
integer :: b=10, c=20
```

In the first case, not only `'a'` is declared as an integer but it is also initialized to `10`. Similarly, `x, b` and `c` are also initialized to `1.34, 10` and `20`, respectively. In case more than one variable is declared by a single declaration, all variables are to be initialized individually. For example,

```
integer :: d, e=200
```

will initialize `e` to `200`, but `d` will remain uninitialized. If it is necessary to initialize both variables, it is to be done separately:

```
integer :: d=200, e=200
```

1.34 Number System

Strictly speaking, the digits are just symbols; the positions of a digit within a number determine its value. For example, when the base of the number system is `10`, the number `123` is actually $1\times10^2 + 2\times10^1 + 3\times10^0$. Therefore, if a digit, say, `3`, appears at the unit position, its value is `3`. On the other hand, if the same digit appears at the position of `10`, its value is `30`. A number may be represented in terms of a base other than `10`. The most popular bases are binary (base of `2`), octal (base of `8`) and hexadecimal or hex (base of `16`). However, the numerical value of a particular number is independent of the number system (base)—the value of a particular number is same in all systems. In the next few sections, we indicate a base other than base `10` by means of a subscript—$(1110)_2$ stands for a binary number.

1.35 Binary Numbers

A bit is the abbreviation of **bi**nary digi**t**. In the binary system, that is, when the base is `2`, the available digits are `0` and `1`. For example, a number $(1101)_2$ is equal to `13` in the decimal system: $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$.

1.36 Octal Numbers

Octal numbers have a base of `8`. The available digits are `0` through `7`. Three binary digits constitute one octal digit (the highest value is `7`, that is, $(111)_2$). An octal number $(101)_8$ is equal to `65` in the decimal system: $1\times8^2 + 0\times8^1 + 1\times8^0$.

1.37 Hexadecimal Numbers

Hexadecimal numbers—popularly known as hex numbers—have a base of `16`. The available digits are `0` through `9` and `a, b, c, d, e` and `f` (capital letters may also be used). The last six symbols are equivalent to decimal `10, 11, 12, 13, 14` and `15`, respectively. Four binary digits constitute one hex digit. The highest value of a hex digit is `'f'`, that is, `15` in the decimal system. The hex number $(101)_{16}$ is equal to `257` in the decimal system: $1\times16^2 + 0\times16^1 + 1\times16^0$.

1.38 Initialization Using DATA Statement

`Data` statements are used to initialize variables. They are usually placed at the beginning of the program unit along with other specification statements. It is normally placed after the declaration of the variables. However, it can be placed anywhere within the program unit, but this should not be done. Two integer variables `i` and `j` may be initialized to `10` and `20`, respectively, by a `data` statement as follows:

```
     data i /10/
     data j /20/
```

or

```
     data i /10/, j /20/
```

or

```
     data i, j /10, 20/
```

The last two declarations are equivalent. Note that either the variables are declared by appropriate declaration or the default `i-n` rule is followed. The following are the `data` statements to initialize variables `l, d` and `c`:

```
logical :: l
double precision :: d
complex :: c
data l /.true./
data d /3.1415926589d0/
data c /(2.0, 3.0)/
```

For the complex variable `c`, the first constant corresponds to the real part, and the second constant corresponds to the imaginary part of the variable. In the case of a complex variable, the real and the imaginary parts are enclosed in parentheses.

Character variables are also initialized in a similar manner:

```
character (len=4) :: ch
data ch / 'iacs'/
```

If the number of characters is less than the size of the variable, blanks are added at the end. If the number of characters is more than the size of the variable, the constant is truncated from the right:

```
character (len=8) :: ch
data ch / 'iacs calcutta'/
```

The variable `ch` is initialized to `'iacs cal'` as it can store a maximum of `8` characters.

1.39 BOZ Numbers

The binary, octal or hex numbers (also known as boz numbers) are represented by the respective digits enclosed in apostrophes or quotes and prefixed by `b`, `o` or `z`, respectively. The following are binary, octal and hex numbers:

```
b'1001' (decimal 9)
o'127'  (decimal 87)
z'1b7'  (decimal 439)
```

The uppercase letters `B, O` and `Z` may be substituted for their corresponding lowercase counterparts – `b, o` and `z`, respectively.

1.40 Integer Variables and BOZ Numbers

An integer variable may be initialized by a binary, octal or hex number:

```
integer :: a=b'111'
integer :: b=o'171'
integer :: c=z'1a'
```

A BOZ constant can be used to initialize an integer variable by a `data` statement:

```
integer :: p, q,r
data p/b'1111'/
data q/o'247'/
data r/z'12a'/
```

1.41 Executable and Non-Executable Statements

Fortran statements are basically of two types: executable and non-executable statements. The first one means some action. For example, `x=10` is an executable statement, where the variable `x` is assigned to a value `10`. The statement `integer :: y` is a non-executable statement. It is a declaration and merely passes information to the compiler to reserve locations for an integer variable `y`. The non-executable statements configure the programming environment where executions of executable statements are performed. Non-executable statements cannot be the targets of any branch statement (Chapter 3).

1.42 INCLUDE Directive

Strictly speaking `include` is not a Fortran statement; it is a directive to the compiler. The syntax of `include` is

```
include 'char-constant'
```

where the `character-constant` is usually the name of a file. The compiler replaces the `include` statement by the content of the file. The `include` statement cannot be labeled; it may be nested; that is, it may contain another `include` statement (nested `include`). The maximum number of nested `include` is processor dependent. The `include` cannot `'include'` itself directly or indirectly—`include 'a'` may contain `include 'b'`, but then the file `'b'` cannot contain `include 'a'` (a recursive "call"). The `include` statement must be typed on a separate line, and it may have a trailing comment:

```
include 'myfile.f90'
```

The content of `'myfile.f90'` is included at the point of inclusion. The `include` statement cannot be continued. The first included line should not be a continuation line; the last included line cannot be continued. The following program segment is unacceptable:

```
a=b+ &
include 'myfile.f90'
...
```

The file `myfile.f90` cannot have its first line as

```
&c
```

Similarly,

```
include 'myfile.f90'
&c
```

with the last line of the file `'myfile.f90'` as

```
a=b+&
```

is also not acceptable.

1.43 Statement Ordering

In a program unit, statements are ordered as shown in Appendix B. Usually, the declarations come at the beginning of the program unit.

1.44 Processor Dependencies

If the source line is created with characters other than the default type, the number of characters that a source line can have is processor dependent. There is no guideline how the compiler will identify the free-form and fixed-form sources. Though, normally, ASCII is the default character set, there is no specific guideline in the Fortran report in this matter. The interpretation of the char-literal-constant used with the `include` compiler directive is processor dependent. Also, the maximum number of `include` directives that may be nested is not specified in the Fortran report.

The maximum and minimum values (numeric) are processor dependent.

1.45 Compilation and Execution of Fortran Programs

The programs in this book have been tested with three compilers: ifort, nagfor and gfortran.

To compile a Fortran program using the `ifort` compiler, the instruction is (name of the source file—`x.f90`)

```
ifort x.f90
```

This will create an executable x.exe, which can be executed.

A Fortran program using Numerical Algorithm Group's `nagfor` can be compiled as follows:

```
nagfor -o x.exe x.f90 [x.exe is the executable file]
```

A Fortran program using the GCC gfortran compiler can be compiled as follows:

```
gfortran  -o x.exe x.f90 [x.exe is the executable file]
```

2

Arithmetic, Relational and Logical Operators and Expressions

As the computer is a machine that can perform basic arithmetic operations at a high speed, naturally Fortran is provided with arithmetic operators to perform these operations on arithmetic expressions. Alongside, relational operators can test a relation. They return either a `true` or `false` value. For example, if a question is asked, "Is `x` greater than `y`?" The answer is either yes (`true`) or no (`false`). In addition to this, five logical operators are also available that return either a `true` or `false` value. We first consider arithmetic operators.

2.1 Arithmetic Operators

Two types of arithmetic operators are available to a Fortran programmer – binary and unary operators. Binary operators require two operands. The binary arithmetic operators are shown in Table 2.1.

Examples of binary operators are as follows:

```
a + b   (add a to b)
a * b   (multiply a by b)
a - b   (subtract b from a)
a / b   (divide a by b)
a ** b (a to the power of b)
```

Unary operators require a single operand. Table 2.2 shows the unary arithmetic operators. As unsigned integers, real constants or variables are treated as positive numbers, unary plus is rarely used; `+5` is same as `5`. The unary minus changes the sign of a variable or a constant. An example of the unary minus is `-5`, where the sign of `5` is changed. Similarly, the magnitude of `-x` is the value of `x` with its sign reversed.

TABLE 2.1

Arithmetic Operators (Binary)

Symbol	Meaning
**	Exponentiation (to the power)
/	Division
*	Multiplication
+	Addition
-	Subtraction

TABLE 2.2

Unary Operators

Symbol	Meaning
+	Unary plus
-	Unary minus

It may be noted that the same symbols '+' and '-' are used to indicate both the unary and binary operations. The compiler can determine from the context the meaning of the operators—whether it is a binary or a unary operator.

2.2 Arithmetic Expressions

Arithmetic expressions are formed using constants, variables and other objects as permitted by the language and arithmetic operators discussed in the previous section. Examples of arithmetic expressions are as follows:

```
x + y + z
5 * j + k
p + q - c**3 / f + 27.35
```

2.3 Assignment Sign

The symbol '=' is used to assign a value to a variable. The general form of an assignment statement is as follows:

```
variable = expression
```

The expression on the right-hand side of the assignment sign is evaluated, and the value thus obtained is stored in the variable. As a variable can store only one value at a time, the current value of the variable is lost, and a new value is stored in its place. Examples of assignments are as follows:

```
i = 2
area = length * width
s = u * t + 0.5 * f * t**2
```

Consider the following expression:

```
c = a + b
```

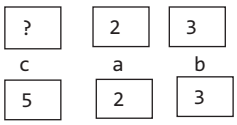


FIGURE 2.1
Arithmetic operation.

Let us assume that the values of a and b are 2 and 3, respectively. The value of c is not our concern at this moment. Before the expression is evaluated, the contents of a, b and c are as shown in the upper panel of Figure 2.1.

When a is added to b, the result is 5, and it is stored in location c. However, a and b will retain their old values. Therefore, at the end of the operation, the values are as shown in the lower panel of Figure 2.1.

The symbol for the assignment sign, that is, '=', must not be confused with the equal sign used in algebra.

For example, i = i + 1 is a valid Fortran statement, which is not an algebraic equation. Had it been so, canceling i from both sides would give us, 0 = 1, which is clearly not acceptable. The proper meaning of this statement is to increment i by 1. To be more specific, in this

case the current value of `i` is taken, `1` is added to it and the result is stored in the same location, `i`. If, for example, the value of `i` is `10` before the execution of the statement, it is `11` at the end of the operation, and the result is stored in location `i`.

One can write similar statements:

```
i = i - 1
i = i * j
```

In an assignment operation, unless the same variable appears on both sides of the assignment sign, the variables appearing on the right-hand side of the assignment sign are not modified – they retain their old values, and the variable on the left-hand side of the assignment sign gets a new value.

2.4 Rules for Arithmetic Expressions

The following rules must be followed while writing arithmetic expressions:

Rule 1: Arithmetic operations are not allowed on the left-hand side of the assignment sign. For example, `a + b = c` is not a valid arithmetic expression. The left-hand side of the assignment sign must be a variable. Arithmetic operation on the left-hand side of the assignment sign is allowed only to calculate the address of a variable and the like. This is discussed in Chapter 6, when we discuss array.

Rule 2: No arithmetic operation is assumed like algebra: `(a+b)(a-b)` is not taken as `(a+b)*(a-b)`. The multiplication operator in this case must be specified explicitly.

Rule 3: No two arithmetic operators may appear side by side: `c = a * -b` is not a valid Fortran statement. Should such situation arise, it must be enclosed in parentheses: `c = a *(-b)`. However, some compilers do not flag this as error. This rule appears to have been violated in case of exponentiation operator `'**'`. However, it must be remembered that the exponentiation operator is a single entry – it is not two successive multiplication operators.

Rule 4: Arithmetic expressions may contain parentheses and also nested parentheses (i.e., parentheses within parentheses). In case of nested parentheses, the nearest left and right parentheses form a pair. If an expression contains parentheses, the number of left parentheses must be equal to the number of right parentheses. In case of nested parentheses, computation proceeds from the inner to the outer parentheses.

The Fortran statement `f = a + (b + c * (d + e)` will be rejected by the compiler because of unmatched parenthesis. It should have been, `f = a + (b + c * (d + e))`.

Rule 5: If `a` and `b` are real numbers, `a ** b` can be evaluated only if `a` is a positive quantity. This is because when both `a` and `b` are real numbers, `a ** b` is calculated as $e^{b\ln(a)}$, where `ln` is logarithm to the base `e`. If `a` is negative, `ln(a)` is not defined.

2.5 Precedence of the Arithmetic Operators

An arithmetic expression may contain different kinds of operators. Therefore, it is necessary to specify a rule regarding how the expressions like `d = a / b * c` are going to be evaluated. If the division is performed before the multiplication, the expression becomes algebraically `d = (a / b) × c`. However, if the multiplication is performed before the division, the expression becomes `d = a / (b × c)`. Needless to say, the results of the two sets of calculations are different. This may be verified by assuming the values of `a`, `b` and `c` as `6`, `3` and `2`, respectively. In the first case, `d = (6 / 3) × 2 = 4`, and in the second case, `d = 6 / (3 × 2) = 1`.

Arithmetic operators are assigned different priorities. The priority of the exponentiation operator is the highest and that of the assignment operator is the lowest. The priority of the multiplication and the division operators is same and less than that of the exponentiation operator. The priority of the addition and the subtraction operators is same and lower than that of the multiplication and the division operators. The priority of unary plus and minus operators is in between the multiplication/division and the addition/subtraction operators. In any arithmetic expression, the high-priority operators are evaluated before the low-priority operators. For example, in case of the expression `e = a + b * d`, multiplication, `b * d`, is performed first and then `a` is added to get the result. If an arithmetic expression contains operators having same priority, computation proceeds from left to right. In case of `d = a + b - c`, the addition will be performed before the subtraction (Table 2.3).

There is one exception to the preceding rule. For exponentiation, the evaluation proceeds from right to left; for `a ** b ** c`, `b ** c` is performed first and then the result is used as the power of `a`. If brackets are used to indicate the order of evaluation, then `(a ** b) ** c` and `a ** (b ** c)` are not equivalent. This may be verified by assuming `a = 2`, `b = 3` and `c = 4`. Substituting these values, `(a ** b) ** c` becomes 2^{12} and `a ** (b ** c)` = 2^{81}. When a complex number `c1` is raised to the power of another complex number `c2`, the result is the principle value of $c1^{c2}$.

TABLE 2.3

Precedence of Arithmetic Operators

High	Exponentiation	**
↓	Multiplication and division	/, *
	Unary minus and plus	-, +
	Addition and subtraction	+, -
Low	Assignment	=

The parentheses have the highest priority. Inside the parentheses, the preceding rules are followed. It may be noted that `-2**2` is `-4` but `(-2)**2` is `4`. Also, `a/(b*c)` and `a/b*c` are not the same. In the first case, `b*c` is evaluated first and then `a` is divided by the result. In the second case, without the parentheses, computation proceeds according to the default priority rules. The priority of the division and the multiplication being equal, computation proceeds from the left to the right and the division is performed before the multiplication. The result is multiplied by `c`.

In the case of nested parentheses, computation starts from the innermost one.

Consider the expression `a + (b * (c * (d + e / f)))`. The innermost parentheses containing expressions involving the division and the addition are evaluated first, the default priority rules being used (division before addition). The result is then multiplied by `c`, which is then multiplied by `b` and the result is added to `a`. In Figure 2.2, the numbers indicate the order of evaluation. The first is indicated by 1 and the second by 2 and so on. The rule of thumb is that in case of any doubt, parentheses may be used to indicate the intention. Extra balanced parentheses do not cause any harm. The expression `d = a / b * c` is same as `d = (a / b) * c`.

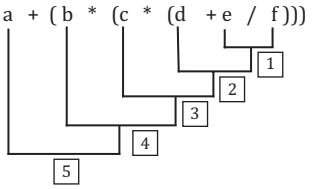


FIGURE 2.2
Evaluation of arithmetic expression.

Sometimes compilers are smart enough to change the order of the evaluation of the arithmetic expressions to make it more efficient. In Table 2.4, expressions and allowable alternative forms used by the compiler are shown. In these expressions, `x`, `y` and `z` are any type of numeric operands and `a`, `b` and `c` represent any arbitrary real or complex variables. Table 2.5 shows expressions that the compiler will never convert to the alternative forms. In this case, `i` and `j` are integers.

TABLE 2.4
Allowable Alternative

Expression	Alternative Form
`x + y`	`y + x`
`x * y`	`y * x`
`-x + y`	`y - x`
`x + y + z`	`x + (y + z)`
`x - y + z`	`x - (y - z)`
`x * a / z`	`x * (a / z)`
`x * y - x * z`	`x * (y - z)`
`a / b / c`	`a * (b * c)`
`a / 5.0`	`0.2 * a`

TABLE 2.5
Non-allowable Alternative

Expression	Non-allowable Alternative Form
`i / 2`	`i * 0.5`
`x * i / j`	`x * (i / j)`
`i / j / a`	`i / (j * a)`
`(x + y) + z`	`x + (y + z)`
`(x * y) - (x * z)`	`x * (y - z)`
`x * (y - z)`	`x * y - x * z`

2.6 Multiple Statements

Two or more Fortran statements, separated by a semicolon, may be placed in a line:

```
c = a + b; d = 10
```

This is same as follows:

```
c = a + b
d = 10
```

In this case, only the first statement may have a statement number. The semicolon is not a part of the Fortran statement. Two or more successive semicolons separated by zero or more blanks constitute a single semicolon; `a=2;; b=3` is same as `a=2; b=3`. If a line containing

a complete Fortran statement is terminated by a semicolon, it is ignored. It is treated as a statement separator; `a=10;` is the same as `a=10`. Some compilers give compilation error if the first non-blank character in a line is a semicolon.

As multiple statements decrease the program readability, this is not recommended. However, in this book we use this semicolon to save some spaces in the book.

2.7 Mixed-Mode Operations

For a numeric expression involving variables or constants of different types, conversion takes place before the expression is evaluated. First, we consider expressions involving reals and integers.

In an expression involving a real and an integer constant or variable on the two sides of a binary arithmetic operator, the integer is converted into a real number before the calculation takes place. For example, the expression `a+2` will be calculated as (`a` is a real variable): integer `2` will be converted to real `2.0` by the processor and `2.0` will be added to `a`. The result of `(a+2)` will be real.

Similarly, during the assignment operation, if the type of the variable (or the result) on the right-hand side of the assignment is different from the type of the variable on the left-hand side, an automatic type conversion takes place. An integer is converted into a real number, keeping the magnitude same – integer `2` is converted to real `2.0`. However, a real number is converted into an integer by truncating the fractional part—real `4.56` is converted to integer `4`.

One important point should be noted, the operands determine the type of the operation and accordingly type conversion takes place. Consider the expression `i = j + a * 2`, where `i` and `j` are integers and `a` is a real number (Figure 2.3). The steps required to perform this computation are as follows: (a) integer `2` is converted to a real number and stored in a temporary location, (b) priority of multiplication operator is more than that of addition, (c) `2.0` is multiplied by `a`—the result of the computation is real and is stored in a temporary location within the system, (d) `j` is converted to a real number because the result of the computation `a * 2` is real and is stored in a temporary location, (e) the addition is performed in the

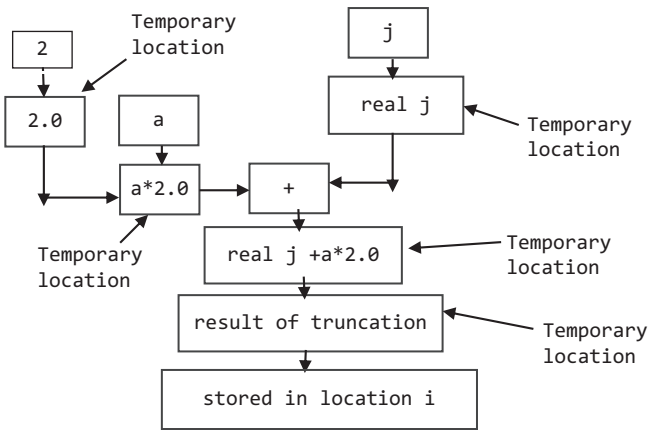


FIGURE 2.3
Mixed-mode arithmetic. `i = j + a * 2`

real mode—the result of addition is stored in a temporary location and (f) as the left-hand side of the assignment operator is an integer, the result of the computation is converted to an integer and is stored in location `i`.

Operations involving an integer or a real quantity with a double precision variable or constant are performed by first converting the integer or real quantity into double precision. However, it must be noted that the real number, thus converted, does not become more precise. The digits after the normal precision are meaningless. If single precision `π (3.1415927)` is converted to a double precision number, then the digits after the last digit on the right (say, `6`) are not correct. Similarly, when a single precision expression is equated to a double precision variable, the single precision number is converted into a double precision number before it is stored. Again, extra digits so added to make it a double precision number do not have any significance. Thus, by equating a single precision number to a double precision variable, the resulting double precision number does not become more precise compared to the single precision number. The precision may be illustrated by considering the following program:

```
program testdbl
implicit none
double precision:: d1,d2
d1 = 2.0/3.0
d2 = 2.0d0/3.0d0
print *, d1,d2
end program testdbl
```

It is known that the result of the arithmetic operation is `0.666666666666...`. The outputs of the program (using NAG Fortran compiler) are `0.6666666865348816` and `0.66666666666666`, respectively. Note that the displayed value of `d1` is correct up to `7` significant figures. Consider the following program:

```
program testdbl2
implicit none
double precision:: d1,d2
d1 = 3.1415926535897932
d2 = 3.1415926535897932d0
print *, d1,d2
end program testdbl2
```

In absence of `'d0'`, `3.1415926535897932` is treated as a single precision constant and additional digits after, say, `6` significant digits are removed to make it a single precision constant. Again, this single precision constant, when equated to a double precision variable `d1`, is converted into a double precision constant, but the digits so added do not have any significance. The outputs of the program are, respectively, `3.1415927410125732` and `3.1415926535897931`.

We further illustrate mixed-mode arithmetic with double precision quantities in the following example:

```
double precision:: d1, d2, d3, d4
d1 = 1.1/3.1; d2 = 1.1d0/3.1; d3 = 1.1/3.1d0; d4 = 1.1d0/3.1d0
print *,"d1=",d1; print *,"d2=",d2; print *,"d3=",d3; print *,"d4=",d4
end
```

The results are as follows (shown in the same line):

```
d1 = 0.3548387289047241, d2 = 0.3548387205935669,
d3 = 0.3548387173683413, d4 = 0.3548387096774194
```

The results need explanation. The right-hand side of the expression involving `d1` is in single precision. Therefore, the calculation is performed in single precision mode, and subsequently, the result is converted into double precision. The result is correct up to `7` places of decimal. One of the quantities on the right-hand side of the expression involving `d2` is double precision. Therefore, the other single precision number `3.1` is converted into a double precision number. However, `3.1` when converted into a double precision number is less accurate than `3.1d0`. Similar logic holds for `d3`, where `3.1` is a double precision number but `1.1` is not. Thus, in the case of `d4`, where both `1.1` and `3.1` are double precision numbers, the result is the most accurate and it is correct up to, say, `14` places of decimal.

In mixed-mode operations involving an integer or a real number with a complex quantity, the integer or the real number is converted into a complex number with the imaginary part set to `0`. Thus, `c*2.0`, where `c` is a complex variable calculated as `c * cmplx(2.0, 0.0)`. In a similar way, when a real number or an integer is equated to a complex variable, the real number or the integer becomes the real part of the complex variable with `0` as the imaginary part.

In arithmetic operations involving a double precision real and a complex variable, the double precision variables are first converted into a double complex number (both the real and the imaginary parts are double precision quantities). Subsequently, the complex variable is converted into a double precision complex number. The result is a double complex number.

This may be verified with the help of the following program:

```
double precision:: d=2.1234567891234d0
complex:: c=(2.0, 3.0)
print *, d+c
end
```

2.8 Integer Division

Integer division deserves special attention. Improper use of integer division may invite serious problems. If `i` and `j` are integers, `i` divided by `j` is calculated in the integer mode and the result of the computation is an integer (whole number). Following this logic, `3 / 2` is `1` and `2 / 3` is `0`. Even if `a` is a real variable, the expression `a = 5 / 2` is calculated in the integer mode. The result is `2` and because `a` is real, the result of the computation is converted to `2.0` and `2.0` is stored in `a`. The computer itself does the conversion. Now, consider the expressions `i = 5.0 / 2.0` and `j = 5 / 2`. In the first case, `5.0 / 2.0` is a real number and it is `2.5`. As `i` is an integer, `2.5` is truncated to `2` before it is stored in `i`. In the second case, `2` is stored in `j` and no type conversion takes place as all the operands and the variable on the left-hand side of the assignment sign are of the same type. It may be noted that the result of the computation in the preceding two cases are same; however, the way they are evaluated is different.

Consider the expression `x = y * 10 ** (-2)`. The value of `x` is `0`, irrespective of the value of `y`. This is because `10 ** (-2)` is evaluated as `1 / 100`. As both `1` and `100` are integers, computation is performed in the integer mode and the result is `0`. One should be extremely careful while translating algebraic expressions like

$$a = 4 / 3 \times \pi \times r^3$$
$$s = u \times t + 1 / 2 \times f \times t^2$$

into Fortran; `4 / 3` is `1` and `1 / 2` is `0`. It is necessary to write at least one integer constant as a real number or better both as real numbers:

```
a = 4.0 / 3.0 * 3.1415926 * r ** 3
s = u * t + 1.0 / 2.0 * f * t ** 2 ! 1.0 / 2.0 may be written as 0.5
```

Often the result of computation unexpectedly turns out to be `0`. In such a situation, one should look for an integer division similar to that shown earlier. It is advised to avoid mixed-mode operations as far as practicable.

2.9 List-Directed Input/Output Statement

We now introduce the free-formatted input/output statements. These are also called list-directed input/output statements.

An input statement reads a value of a variable from an external device, namely, the keyboard. Similarly, an output statement displays the value of a variable, the result of a computation or some message on an output device, namely, screen.

Various kinds of input/output devices are available. Some devices are used only for input and some devices are used only for output, while some devices are used for both input and output. Free-formatted input/output statements are

```
read *, list
print *, list
```

respectively, where list is a list of items to be read or written. If the list contains more than one element, the elements are separated by comma:

```
read *, a, b, c
print *, a, b, c
```

Therefore, `read *, x` will read data from the keyboard and store in location `x` erasing the existing value of `x`. Similarly, `print *, y` will display the current value of `y` on the screen. When a `read *` statement is encountered, the computer waits until the required input is supplied through the keyboard.

Normally, for numbers, one or more blanks are used as delimiters; `read *, i, j, k` will have the corresponding data from the keyboard as `10 20 30` so that `10, 20 and 30` will be stored in locations `i, j` and `k`, respectively. Usually, while supplying inputs, list items are separated by one or more blanks. However, characters like comma, tab or carriage return

may also be used as delimiters. For example, in the earlier case, data may be entered in the following manner also:

```
     10, 20, 30 <enter>
or,  10 <enter>
     20 <enter>
     30 <enter>
or,  10, 20 <enter>
     30 <enter>
```

or various such combinations where `<enter>` indicates the enter key of the keyboard.

For real numbers, normally the decimal point is typed. If the decimal point is absent, it is assumed just before the delimiter. If `x, y` and `z` are declared as real, and the data corresponding to the `read` statement is entered as `10 20 30`, it is taken as `10.0, 20.0` and `30.0` for `x, y` and `z`, respectively.

Real numbers may be entered as input in scientific notation also for the earlier case: `1.4e2 1.245e-4 3.25`. In this case, `x, y` and `z` are assigned to 1.4×10^2, 1.245×10^{-4} and `3.25`, respectively. To display a message on the screen, it needs to be enclosed in apostrophes (or quotes).

```
print *, 'The result is = ', r
```

If the value of `r` is, say, `2.5`, this print statement will display `The result is = 2.5` on the screen. Within apostrophes, a blank is also treated as a character and the number of blanks between `'='` and `2.5` on the screen depends on the number of blanks between `'='` and the closing apostrophe within the `print` statement. There must be a comma between the message and the list element.

The `read` and `print` statements are equally valid for double precision variables.

```
double precision:: d1
read *, d1; print *, d1
```

The `read` and `print` statements can be used for complex variables also. As the complex number consists of two parts, two numbers are to be supplied during the input operation for each complex variable. Similarly, two numbers are displayed during the output operation. The first and the second numbers correspond to the real and the imaginary parts, respectively. The data, in response to the `read` statement, is supplied as `(r, i)`, where `r` is the real part and `i` is the imaginary part.

```
complex:: c1
read *, c1
```

If the data supplied from the terminal is `(2.0, 3.0)`, `2.0+i 3.0` will be assigned to `c1`. Similarly, `print *, c1` will display the real and the imaginary parts within brackets.

To read a logical variable from the keyboard,

```
read *, l  ! l is a logical variable
```

is to be used. The data must be of the following type: `.TRUE.` or `TRUE` for the `true` value and `.FALSE.` or `FALSE` (lowercase letters are also allowed) for the `false` value. In addition, if the first non-blank character is `T` or `F` (uppercase or lowercase) or a period followed

by `T` or `F` (uppercase or lowercase), `true` or `false` value is, respectively, read in. Note that if `TREU` (intentional spelling mistake) is typed in place of `TRUE`, `true` value is assumed because the first non-blank character is `T`.

To print a logical variable, `print *, l` is used, where `l` is a logical quantity or expression. The `print` statement displays either `T` or `F` depending on whether `l` is `true` or `false`.

For characters, it is necessary to enclose the data in apostrophes (or quotes) when the data contains leading or trailing or embedded blanks. If the data does not contain leading, trailing or embedded blanks, apostrophes are optional. It is always better to enclose the character data in apostrophes (or quotes):

```
character (len=20):: ch
read *, ch; print *, ch
```

If the data is `Indian Association`, `'Indian'` is stored in `ch` because the blank between `'Indian'` and `'Association'` prevents reading the data beyond `'Indian'`. Here, the blank acts as a separator. Therefore, the `print` statement will print `'Indian'` (without the apostrophe). If the data is enclosed in apostrophes, `ch` becomes `'Indian Association'`. If the data corresponding to `read` statement is `bbbIndian` (`b` stands for a blank; `3` blanks in front), the variable `ch` will be assigned to blank as the second blank acts a separator. However, if it is desired that `ch` should be assigned exactly like the data, the data must be enclosed in apostrophes.

If the list item is a pointer (Chapter 16) or an allocatable variable (Chapter 15), the pointer should point to a target and the allocatable variable must have its status allocated.

List-directed input/output statements are very convenient. However, the programmer has practically no control over its appearance on the screen. For example, while using the `print` statement, the programmer has little control where the value will appear on the screen and how many digits will be displayed after the decimal point for a real number.

2.10 Variable Assignment—Comparative Study

We have just seen that a variable may be assigned in three different ways—through initialization, through assignment and through the `read` statement. A guideline may be prescribed as follows:

- True constants like `π (3.1415926)` and `e (2.303)` should be declared as named constants.
- If the initial value is required for a variable, then it should be initialized along with the declaration.
- If the same program is to be executed for different sets of values, then the corresponding variables should be read from outside; `read` statements should be used.

2.11 Library Functions

Several commonly used functions are available in the system as library functions to calculate square root, absolute value, trigonometric functions, etc. A library function is called (invoked) by its name and correct number and type of arguments are supplied

within parentheses. For example, the library function `sqrt` calculates square root of a real (double precision, complex, etc.) quantity and takes one argument. The function `nint` takes one real number as its argument and returns an integer nearest to its argument; `nint(3.7)` returns `4` and `nint (2.3)` returns `2`. The function `floor` takes one real number as its argument and returns the greatest integer less than or equal to its argument; `floor(9.8)` returns `9`.

2.12 Memory Requirement of Intrinsic Data Types

Memory requirements of intrinsic data types (in binary digits, Chapter 10) can be obtained using the library function `storage_size`. This function takes one intrinsic data type as its argument. It returns the size of the data type in memory (binary digits).

```
integer:: a
real::b
double precision::c
complex::d
character::e
double complex:: f
print *, 'Integer, Size =          ',storage_size(a)
print *, 'Real, Size =             ',storage_size(b)
print *, 'Double Precision, Size = ',storage_size(c)
print *, 'Complex, Size =          ',storage_size(d)
print *, 'Character, Size =        ',storage_size(e)
print *, 'Double Complex, Size =   ',storage_size(f)
end
```

The outputs are as follows:

```
Integer, Size =                   32
Real, Size =                      32
Double Precision, Size =          64
Complex, Size =                   64
Character, Size =                  8
Double Complex, Size =           128
```

2.13 Programming Examples

The following example converts miles to kilometers:

```
program miletokm
implicit none
real, parameter:: factor=1.609 ! mile to km conversion factor
integer:: mile, yard
```

```
real:: km ! Marathon distance - 26 miles 385 yards=42.185 km
mile=26; yard=385
km=factor*(mile+yard/1760.0) ! 1 mile=1760 yards
print *, mile, 'Mile and ', Yard, 'yards = ', km, 'Kilometers'
end program miletokm
```

In this program, `yard/1760.0` is very crucial; `yard` is an integer and if `1760` is written in place of `1760.0`, then the result of the division would be `0`.

The next program calculates the escape velocity from the earth. The escape velocity is defined as the minimum velocity that a projectile requires to escape from the earth. It is dependent on the mass `(m)`, radius `(r)` of the earth and the universal gravitational constant `G`. It is given by `sqrt (2.0*G*m/r)`.

```
program escape
implicit none
real::rearth = 6378.0e3, earthm = 5.98e24 ! radius in meter, mass in kg
real::ev, G=6.67300e-11 ! gravitational constant
ev=sqrt(2.0*G*earthm/rearth)
print *,'Escape Velocity - Earth: ', ev/1000.0, 'km/sec'
end program escape
```

Output:

```
Escape Velocity - Earth: 11.18623 km/sec
```

If the argument is an expression, then the expression is evaluated and the square root of the result is calculated. The final program converts seconds to hours, minutes and seconds. It uses the property of integer division—an integer divided by another integer is an integer (whole number). The library function `mod(i, j)` returns the reminder of `i/j`.

```
program convt
implicit none
integer:: hh, mm, ss, t, second=7540
t=second/60
ss=mod(second,60) ! reminder of second/60. mod is a library function
hh=t/60; mm=mod(t,60)
print *, second, 'seconds = ', hh, 'hour ', mm,' minute ', ss, 'second'
end program convt
```

The output is:

```
7540 seconds = 2 hour 5 minute 40 second
```

2.14 BLOCK Construct

A `block` construct usually contains both declarations and statements. The construct is terminated by `end block`.

```
block
.
end block
```

The `block` construct may have a label. If `end block` has a label, it should be same as the label used with `block` construct.

```
thisblock: block
           .
           end block thisblock
```

Certain statements like `common, equivalence, implicit, intent, namelist, optional, statement function` and `value` cannot be used within a `block` construct. We have not yet introduced these statements. Some of these statements will be introduced later. Consider the following block construct:

```
block
 integer:: i
 .
 do i=1,10
 .
 enddo
 .
end block
```

The variables declared within a `block` construct are lost when the `block` is exited. In case the variable declared above the `block` construct (as shown later—known as global variable) has the same name as the variable declared within the `block`, called local variable for the `block`, the local variable always prevails (visible) over the global variable having the same name within the block. The global variable is not available within the `block` when there is a name conflict. The global variable reappears when the `block` is exited.

```
program block_demo
integer:: i          ! global to the block
i=27
 block
  integer:: i        ! local to the block
  i=77; print *, i
end block
print *, i
end program block_demo
```

Inside the `block`, the value of `i` is `77`. The global `i` is not available within the `block`. When the `block` is exited, the global `i` with its value as `27` reappears.

2.15 Assignment of BOZ Numbers

Normally, `boz` numbers cannot be used directly to assign a value to a variable. Integer, real, double precision and complex variables may be assigned to binary, octal and hex constants through the library functions `int, real, dble` and `cmplx`, respectively.

This is illustrated through the program shown next. The decimal value corresponding to the binary constants are indicated through in-line comments.

```
program num_sys
implicit none
integer:: i, j,k
real:: a, b,c
complex:: d
i=int(b'11')              ! i=3
j=int(o'16')              ! j=14
k=int(z'a1')              ! k=161
a=real(int(b'111'))       ! a=7.0
b=real(int(o'73'))        ! b=59.0
c=real(int(z'ab'))        ! c=171.0
d=cmplx(real(int(b'11')), real(int(o'16'))) ! (3.0, 14.0)
end program num_sys
```

If a real variable `x` is equated to `real (z'123456789')` (total `9` hex digits), both the gfortran and nagfor compilers give compilation error. However, ifort compiler takes the rightmost `8` hex digits. This is true for other `boz` numbers. The number of binary, octal, or hex digits should be such that it should not exceed the capacity of the processor (32 bits or 64 bits as the case may be).

2.16 Initialization and Library Functions

A variable may be initialized with standard library functions, which can be evaluated at the compilation time.

```
real:: a=sqrt(3.0)
```

In this case, `a` is initialized to the square root of `3.0`, that is, `1.7320508`.

2.17 Relational Operators

A relational operator tests a relation. It returns either `true` or `false`. For example, if a question is asked, "Is `x` greater than `y`?" The answer is either yes (`true`) or no (`false`). The relational operators are `lt, le, gt, ge, eq` and `ne`. These operators are bound by periods. There should not be any space between the periods and the operator. The operators are, respectively, less than, less than or equal to, greater than, greater than or equal to, equal to and not equal to. Either the symbolic notations (in both uppercase and lowercase letters) or the equivalent mathematical notations may be used. The symbol and alternative symbols may be mixed freely within an expression (Table 2.6).

TABLE 2.6

Relational Operators

Symbol	Math Symbol	Meaning
`.lt.`	<	Less than
`.le.`	<=	Less than or equal to
`.gt.`	>	Greater than
`.ge.`	>=	Greater than or equal to
`.eq.`	==	Equal to
`.ne.`	/=	Not equal to

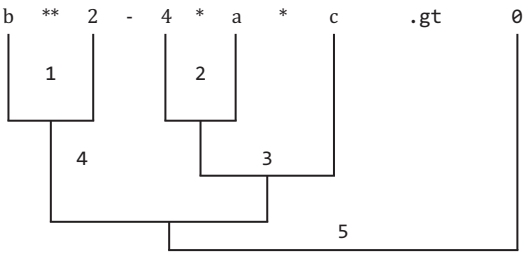


FIGURE 2.4
Precedence rules of relational operators. (1) `b**2`, **(2)** `4*a`, **(3)** `4*a*c`, **(4)** `b**2-4*a*c`, **(5)** `b**2-4*a*c .gt. 0 .0`

2.18 Precedence Rule of Relational Operators

The priority of all the relational operators are same, and it is lower than the priority of arithmetic operators. In an expression involving arithmetic and relational operators, the arithmetic operators are evaluated first and then the relations are tested (Figure 2.4). For `b**2 - 4.0 * a * c.gt. 0`, the order of evaluation is shown using serial numbers `1, 2, 3, 4` and `5`.

2.19 Relational Operators and Complex Numbers

The relational operators `.lt., .le., .gt.` and `.ge.` cannot be used for comparison between two complex numbers or variables. Only `.eq.` (equal) and `.ne.` (not equal) may be used to compare two complex numbers or variables. Two complex numbers are considered to be equal if both the real and the imaginary parts are separately equal. If `z1` and `z2` are complex variables with `z1=cmplx(2.0,3.0)` and `z2=cmplx(2.0,3.0)`, then `z1` and `z2` are equal but the complex variables `c1` and `c2`, where `c1=cmplx(3.0,4.0)` and `c2=cmplx(4.0,3.0)`, are not equal, as their real and imaginary parts are separately not equal.

Although the `arithmetic if` has been declared as obsolete feature, it may be mentioned that the arithmetic expression involving a complex number cannot be used with `arithmetic if`.

2.20 Logical Operators

There are five logical operators. All logical operators are bound by periods (Table 2.7). There should not be any space between the periods and the logical operators.

NOT: The logical operator `.not.` is a logical negation. It changes `true` to `false` and vice versa; `.not. (a>b)` is `true` if `a` is not greater than `b`, that is, `a>b` is `false`. It is `false` if `a` greater than `b` is `true`. For `l =.not. (a>b)`, `true` value is assigned to `l` if `a` is not greater than `b`.