15
Functions

"I can call spirits from the vasty deep.
Why so can I, or so can any man; but will they come
when you do call for them?"

William Shakespeare, *King Henry IV, part 1*

Aims

The aims of this chapter are:

- To consider some of the reasons for the inclusion of functions in a programming language.
- To introduce, with examples, some of the predefined functions available in Fortran 95.
- To introduce a classification of intrinsic functions, generic, elemental, transformational.
- To introduce the concept of a user defined function.
- To introduce the concept of a recursive function.
- To introduce the concept of user defined elemental and pure functions.
- To briefly look at scope rules in Fortran 95 for variables and functions.
- To look at internal user defined functions.

15 Functions

The role of functions in a programming language and in the problem-solving process is considerable and includes:

- Allowing us to refer to an action using a meaningful name, e.g., SINE(X) a very concrete use of abstraction.
- Providing a mechanism that allows us to break a problem down into parts, giving us the opportunity to structure our problem solution.
- Providing us with the ability to concentrate on one part of a problem at a time and ignore the others.
- Allowing us to avoid the replication of the same or very similar sections of code when solving the same or a similar subproblem which has the secondary effect of reducing the memory requirements of the final program.
- Allowing us to build up a library of functions or modules for solving particular subproblems, both saving considerable development time and increasing our effectiveness and productivity.

Some of the underlying attributes of functions are:

- They take parameters or arguments.
- The parameter can be an expression.
- A function will normally return a value and the value returned is normally dependent on the parameter(s).
- They can sometimes take arguments of a variety of types.

Most languages provide both a range of predefined functions and the facility to define our own. We will look at the predefined functions first.

15.1 An introduction to predefined functions and their use

Fortran provides over a hundred intrinsic functions and subroutines. For the purposes of this chapter a subroutine can be regarded as a variation on a function. Subroutines are covered in more depth in a later chapter. They are used in a straightforward way. If we take the common trigonometric functions, sine, cosine and tangent, the appropriate values can be calculated quite simply by:

```
X=SIN(Y)
Z=COS(Y)
A=TAN(Y)
```

This is in rather the same way that we might say that X is a function of Y, or X is sine Y. Note that the argument, Y, is in *radians* not *degrees*.

15.1.1 Example 1: Simple function usage

A complete example is given below:

```
PROGRAM ch1501
REAL :: X
   PRINT *,' Type in an angle (in radians)'
   READ *,X
   PRINT *,' Sine of ', X ,' = ',SIN(X)
END PROGRAM ch1501
```

These functions are called *intrinsic functions.* A selection is follows:

Function	Action	Example
INT	conversion to integer	J=INT(X)
REAL	conversion to real	X=REAL(J)
ABS	absolute value	X=ABS(X)
MOD	remaindering remainder when I divided by J	K=MOD(I,J)
SQRT	square root	X=SQRT(Y)
EXP	exponentiation	Y=EXP(X)
LOG	natural logarithm	X=LOG(Y)
LOG10	common logarithm	X=LOG10(Y)
SIN	sine	X=SIN(Y)
COS	cosine	X=COS(Y)
TAN	tangent	X=TAN(Y)
ASIN	arcsine	Y=ASIN(X)
ACOS	arccosine	Y=ACOS(X)
ATAN	arctangent	Y=ATAN(X)
ATAN2	arctangent(a/b)	Y=ATAN2(A,B)

A complete list is given in Appendix D.

15.2 Generic functions

All but four of the intrinsic functions and procedures are generic, i.e., they can be called with arguments of one of a number of kind types.

15.2.1 Example 2: The ABS generic function

The following short program illustrates this with the ABS intrinsic function:

```
PROGRAM ch1502
IMPLICIT NONE
COMPLEX :: C=(1,1)
REAL     :: R=10.9
INTEGER :: I=-27
  PRINT *,ABS(C)
  PRINT *,ABS(R)
  PRINT *,ABS(I)
END PROGRAM ch1502
```

The four nongeneric functions are LGE, LGT, LLE and LLT — the lexical character comparison functions.

Type this program in and run it on the system you use.

It is now possible with Fortran 95 for the arguments to the intrinsic functions to be arrays. It is convenient to categorise the functions into either elemental or transformational, depending on the action performed on the array elements.

15.3 Elemental functions

These functions work with both scalar and array arguments, i.e., with arguments that are either single or multiple valued.

15.3.1 Example 3: Elemental function use

Taking the earlier example with the evaluation of sine as a basis, we have:

```
PROGRAM ch1503
REAL , DIMENSION(5) :: X = (/1.0,2.0,3.0,4.0,5.0/)
  PRINT *,' Sine of ', X ,' = ',SIN(X)
END PROGRAM ch1503
```

In the above example the sine function of each element of the array X is calculated and printed.

15.4 Transformational functions

Transformational functions are those whose arguments are arrays, and work on these arrays to transform them in some way.

15.4.1 Example 4: Simple transformational use

To highlight the difference between an element-by-element function and a transformational function consider the following examples:

```
PROGRAM ch1504
IMPLICIT NONE
REAL , DIMENSION(5) :: X = (/1.0,2.0,3.0,4.0,5.0/)
! Elemental function
  PRINT *,' Sine of ', X ,' = ',SIN(X)
! Transformational function
  PRINT *,' Sum of ', X ,' = ',SUM(X)
END PROGRAM ch1504
```

The SUM function adds each element of the array and returns the SUM as a scalar, i.e., the result is single valued and not an array.

15.4.2 Example 5: Intrinsic DOT_PRODUCT use

The following program uses the transformational function DOT_PRODUCT:

```
PROGRAM ch1505
IMPLICIT NONE
REAL , DIMENSION(5) :: X = (/1.0,2.0,3.0,4.0,5.0/)
   PRINT *,' Dot product of X with X is'
   PRINT *,' ',DOT_PRODUCT(X,X)
END PROGRAM ch1505
```

Try typing these examples in and running them to highlight the differences between elemental and transformational functions.

15.5 Notes on function usage

You should not use variables which have the same name as the intrinsic functions; e.g., what does SIN(X) mean when you have declared SIN to be a real array?

When a function has multiple arguments care must be taken to ensure that the arguments are in the correct position and of the appropriate kind type.

You may also replace arguments for functions by expressions, e.g.,

```
   X = LOG(2.0)
```

or

```
   X = LOG(ABS(Y))
```

or

```
   X = LOG(ABS(Y)+Z/2.0)
```

15.6 Example 6: Easter

This example uses only one function, the MOD (or modulus). It is used several times, helping to emphasise the usefulness of a convenient, easily referenced function. The program calculates the date of Easter for a given year. It is derived from an algorithm by Knuth, who also gives a fuller discussion of the importance of its algorithm. He concludes that the calculation of Easter was a key factor in keeping arithmetic alive during the Middle Ages in Europe. Note that determination of the Eastern churches' Easter requires a different algorithm:

```
PROGRAM ch1506
IMPLICIT NONE
INTEGER :: Year, Metcyc, Century, Error1, Error2, Day
INTEGER :: Epact, Luna, Temp
! A program to calculate the date of Easter
   PRINT *,' Input the year for which Easter'
   PRINT *,' is to be calculated'
   PRINT *,' enter the whole year, e.g. 1978 '
   READ *,Year
! calculating the year in the 19 year
! metonic cycle using variable metcyc
   Metcyc = MOD(Year,19)+1
   IF(Year <= 1582)THEN
      Day = (5*Year)/4
      Epact = MOD(11*Metcyc-4,30)+1
   ELSE
!          calculating the Century-century
      Century = (Year/100)+1
!          accounting for arithmetic inaccuracies
!          ignores leap years etc.
      Error1 = (3*Century/4)-12
      Error2 = ((8*Century+5)/25)-5
!          locating Sunday
      Day = (5*Year/4)-Error1-10
!          locating the epact(full moon)
      Temp = 11 * Metcyc + 20 + Error2 - Error1
      Epact = MOD(Temp,30)
      IF(Epact <= 0) THEN
         Epact = 30 + Epact
      ENDIF
      IF((Epact == 25 .AND. Metcyc > 11) &
      .or. Epact == 24)THEN
         Epact = Epact+1
      ENDIF
   ENDIF
!     finding the full moon
   Luna= 44 - epact
   IF (Luna < 21) THEN
      Luna = Luna+30
   ENDIF
!     locating Easter Sunday
   Luna = Luna+7-(MOD(Day+Luna,7))
!     locating the correct month
   IF(Luna > 31)THEN
      Luna = Luna - 31
      PRINT *,' for the year ',YEAR
      PRINT *,' Easter falls on April ',Luna
   ELSE
      PRINT *,' for the year ',YEAR
      PRINT *,' Easter falls on march ',Luna
   ENDIF
END PROGRAM ch1506
```

We have introduced a new statement here, the IF THEN ENDIF, and a variant the IF THEN ELSE ENDIF. A more complete coverage is given in the chapter on control structures. The main point of interest is that the normal sequential flow from top to bottom can be varied. In the following case,

```
IF (expression) THEN
    block of statements
ENDIF
```

if the expression is true the block of statements between the IF THEN and the ENDIF is executed. If the expression is false then this block is skipped, and execution proceeds with the statements immediately after the ENDIF.

In the following case,

```
IF (expression) THEN
  block 1
ELSE
```

  block 2

ENDIF

if the expression is true block 1 is executed and block 2 is skipped. If the expression is false then block 2 is executed and block 1 is skipped. Execution then proceeds normally with the statement immediately after the ENDIF.

As well as noting the use of the MOD generic function in this program, it is also worth noting the structure of the decisions. They are *nested*, rather like the nested DO loops we met earlier.

15.7 Complete list of predefined functions

Due to the large number of predefined functions it is useful to classify them, and the following is one classification.

15.7.1 Inquiry functions

These functions return information about their arguments. They can be further subclassified into BIT, CHARACTER, NUMERIC, ARRAY, POINTER, ARGUMENT PRESENCE:

Bit	BIT_SIZE
Character	LEN
Numeric	DIGITS, EPSILON, EXPONENT, FRACTION, HUGE, KIND, MAXEXPONENT, MINEXPONENT, NEAREST, PRECISION, RADIX, RANGE, RRSPACING, SCALE, SET_EXPONENT, SELECTED_INT_KIND, SELECTED_REAL_KIND, SPACING, TINY
Array	ALLOCATED, LBOUND, SHAPE, SIZE, UBOUND,
Pointer	ASSOCIATED, NULL
Argument Presence	PRESENT

15.7.2 Transfer and conversion functions

These functions convert data from one type and kind type to another type and kind type. Most of them are by necessity generic.

Transfer and Conversion	ACHAR, AIMAG, AINT, ANINT, CHAR, CMPLX, CONJG, DBLE, IACHAR, IBITS, ICHAR, INT, LOGICAL, NINT, REAL, TRANSFER

15.7.3 Computational functions

These functions actually carry out a computation of some sort and return the result of that computation:

Numeric	ABS, ACOS, ASIN, ATAN, ATAN2, CEILING, COS, COSH, DIM, DOT_PRODUCT, DPROD, EXP, FLOOR, LOG, LOG10, MATMUL, MAX, MIN, MOD, MODULO, SIGN, SIN, SINH, SQRT, TAN, TANH
Character	ADJUSTL, ADJUSTR, INDEX, LEN_TRIM, LGE, LGT, LLE, LLT, REPEAT, SCAN, TRIM, VERIFY
Bit	BTEST, IAND, IBCLR, IBSET, IEOR, IOR, ISHFT, ISHFTC, NOT

15.7.4 Array functions

Reduction	ALL, ANY, COUNT, MAXVAL, MINVAL, PRODUCT, SUM
Construction	MERGE, PACK, SPREAD, UNPACK
Reshape	RESHAPE
Manipulation	CSHIFT, EOSHIFT, TRANSPOSE
Location	MAXLOC, MINLOC

15.7.5 Predefined subroutines

Date and Time	CPU_TIME, DATE_AND_TIME, SYSTEM_CLOCK
Random Number	RANDOM_NUMBER, RANDOM_SEED
Other	MVBITS

An alphabetical list of all intrinsic functions and procedures is given in Appendix D. This list provides the following information:

- Function name.
- Description.
- Argument name and type.
- Result type.
- Classification.
- Examples of use.

Appendix D should be consulted for a more complete and thorough understanding of intrinsic functions and their use in Fortran 95.

15.8 Supplying your own functions

There are two stages here: firstly, to define the function and, secondly, to reference or use it. Consider the calculation of the greatest common divisor of two integers.

15.8.1 Example 7: Simple user defined function

The following defines a function to achieve this:

```
INTEGER FUNCTION GCD(A,B)
IMPLICIT NONE
INTEGER , INTENT(IN) :: A,B
INTEGER :: Temp
  IF (A < B) THEN
    Temp=A
  ELSE
    Temp=B
  ENDIF
  DO WHILE ((MOD(A,Temp) /= 0) .OR. (MOD(B,Temp) /=0))
    Temp=Temp-1
  END DO
  GCD=Temp
END FUNCTION GCD
```

To use this function, you reference or call it with a form like:

```
PROGRAM ch1507
IMPLICIT NONE
INTEGER :: I,J,Result
INTEGER :: GCD
```

```
   PRINT *,' Type in two integers'
   READ *,I,J
   Result=GCD(I,J)
   PRINT *,' GCD is ',Result
END PROGRAM ch1507
```

The first line of the function

```
INTEGER FUNCTION GCD(A,B)
```

has a number of items of interest:

- Firstly the function has a type, and in this case the function is of type INTEGER, i.e., it will return an integer value.
- The function has a name, in this case GGD.
- The function takes arguments or parameters, in this case A and B.

The structure of the rest of the function is the same as that of a program, i.e., we have declarations, followed by the executable part. This is because both a program and a function can be regarded as a so-called *program unit*. We will look into this more fully in later chapters.

In the declaration we also have a new attribute for the INTEGER declaration. The two parameters A and B are of type integer, and the INTENT(IN) attribute means that these parameters will NOT be altered by the function.

The value calculated is returned through the function name somewhere in the body of the executable part of the function. In this case GCD appears on the left-hand side of an arithmetic assignment statement at the bottom of the function. The end of the function is signified in the same way as the end of a program:

```
END FUNCTION GCD
```

We then have the program which actually uses the function GCD. In the program the function is called or invoked with I and J as arguments. The variables are called A and B in the function, and references to A and B in the function will use the values that I and J have respectively in the main program. We will look into the whole area of argument association in much greater depth in later chapters.

Note also a new control statement, the DO WHILE ENDDO. In the following case,

DO WHILE (expression)

  block of statements

ENDDO

the block of statements between the DO WHILE and the ENDDO is executed whilst the expression is true. There is a more complete coverage in Chapter 16.

We have two options here regarding compilation. Firstly, to make the function and the program into one file, and invoke the compiler once. Secondly, to make the function and program into separate files, and invoke the compiler twice, once for each file. With large programs comprising one program and several functions it is probably worthwhile to keep the component parts in different files and compile individually, whereas if it consists of a simple program and one function then keeping things together in one file makes sense.

Try this program out on the system you work with.

15.9 An introduction to the scope of variables and local variables

One of the major strengths of Fortran is the ability to work on parts of a problem at a time. This is achieved by the use of *program units* (a main program, one or more functions and one or more subroutines) to solve discrete subproblems. Interaction between them is limited and can be isolated, for example, to the arguments of the function. Thus variables in the main program can have the same name as variables in the function and they are completely separate variables, even though they have the same name. Thus we have the concept of a local variable in a program unit. We will look into this area again after a coverage of recursion and very thoroughly after the coverage of subroutines and modules.

In the example above I, J, Result, are local to the main program. The declaration of GCD is to tell the compiler that it is an integer, and in this case it is an external function.

A and B in the function GCD do not exist in any real sense; rather they will be replaced by the actual variable values from the calling routine, in this case by whatever values I and J have. Temp is local to GCD.

15.10 Recursive functions

There is an additional form of the function header that must be used when the function is recursive. Recursion means the breaking down of a problem into a simpler but identical subproblem. The concept is best explained with reference to an actual example. Consider the evaluation of a factorial, e.g., 5!. From simple mathematics we know that the following is true:

5!=5*4!

4!=4*3!

3!=3*2!

2!=2*1!

1!=1

and thus 5! = 5*4*3*2*1 or 120.

15.10.1 Example 8: Recursive factorial evaluation

Let us look at a program with recursive function to solve the evaluation of factorials.

```
PROGRAM ch1508
IMPLICIT NONE
INTEGER :: I, F, Factorial
  PRINT *,' Type in the number, integer only'
  READ *,I
  DO WHILE(I<0)
    PRINT *,' Factorial only defined for '
    PRINT *,' positive integers: Re-input'
    READ *,I
  END DO
  F=Factorial(I)
  PRINT *,' Answer is', F
END PROGRAM ch1508

RECURSIVE INTEGER FUNCTION Factorial(I) RESULT(Answer)
IMPLICIT NONE
INTEGER , INTENT(IN):: I
  IF (I==0) THEN
    Answer=1
  ELSE
    Answer=I*Factorial(I-1)
  END IF
END FUNCTION Factorial
```

What additional information is there? Firstly, we have an additional attribute on the function header that declares the function to be recursive. Secondly, we must return the result in a variable, in this case Answer. Let us look now at what happens when we compile and run the whole program (both function and main program). If we type in the number 5 the following will happen:

- The function is first invoked with argument 5. The ELSE block is then taken and the function is invoked again.

- The function now exists a second time with argument 4. The ELSE block is then taken and the function is invoked again.
- The function now exists a third time with argument 3. The ELSE block is then taken and the function is invoked again.
- The function now exists a fourth time with argument 2. The ELSE block is then taken and the function is invoked again.
- The function now exists a fifth time with argument 1. The ELSE block is then taken and the function is invoked again.
- The function now exists a sixth time with argument 0. The IF BLOCK is executed and Answer=1. This invocation ends and we return to the previous level, with Answer=1*1.
- We return to the previous invocation and now Answer=2*1.
- We return to the previous invocation and now Answer=3*2.
- We return to the previous invocation and now Answer=4*6.
- We return to the previous invocation and now Answer=5*24.

The function now terminates and we return to the main program or calling routine. The answer 120 is the printed out.

Add a PRINT *,I statement to the function after the last declaration and type the program in and run it. Try it out with 5 as the input value to verify the above statements.

Recursion is a very powerful tool in programming, and remarkably simple solutions to quite complex problems are possible using recursive techniques. We will look at recursion in much more depth in the later chapters on dynamic data types, and subroutines and modules.

15.11 Example 9: Recursive version of GCD

The following is another example of the earlier GCD function but with the algorithm in the function replaced with an alternate recursive solution:

```
PROGRAM ch1509
IMPLICIT NONE
INTEGER :: I,J,Result
INTEGER :: GCD
  PRINT *,' Type in two integers'
  READ *,I,J
  Result=GCD(I,J)
  PRINT *,' GCD is ',Result
```

```
END PROGRAM ch1509

RECURSIVE INTEGER FUNCTION GCD(I,J) RESULT(Answer)
IMPLICIT NONE
INTEGER , INTENT(IN) :: I,J
   IF (J==0) THEN
      Answer=I
   ELSE
      Answer=GCD(J,MOD(I,J))
   ENDIF
END FUNCTION GCD
```

Try this program out on the system you work with, look at the timing information provided, and compare the timing with the previous example. The algorithm is a much more efficient algorithm than in the original example, and hence should be much faster. On one system there was a twentyfold decrease in execution time between the two versions.

Recursion is sometimes said to be inefficient, and the following example looks at a nonrecursive version of the second algorithm.

15.12 Example 10: After removing recursion

The following is a variant of the above, with the same algorithm, but with the recursion removed:

```
PROGRAM ch1510
IMPLICIT NONE
INTEGER :: I,J,Result
INTEGER :: GCD
   PRINT *,' Type in two integers'
   READ *,I,J
   Result=GCD(I,J)
   PRINT *,' GCD is ',Result
END PROGRAM ch1510

INTEGER FUNCTION GCD(I,J)
IMPLICIT NONE
INTEGER , INTENT(INOUT) :: I,J
INTEGER :: Temp
   DO WHILE (J/=0)
      Temp=MOD(I,J)
      I=J
      J=Temp
   END DO
   GCD=I
END FUNCTION GCD
```

15.13 Pure functions

Within the world of mathematics there is the concept of a pure function. This means that the function only returns a value, and has no effect on the arguments. Fortran 95 introduced the ability to write user defined pure functions. We will provide examples in Chapter 26, when we have covered the additional syntax that is required.

15.14 Elemental functions

Fortran 77 introduced the concept of generic intrinsic functions. Fortran 90 added elemental intrinsic functions and the ability to write generic user defined functions. Fortran 95 squares the circle and enables us to write elemental user defined functions. We will show how this can be done in Chapter 26 when we have covered the additional syntax that is required.

15.15 Internal functions

An internal function is a more restricted and hidden form of the normal function definition.

Since the internal function is specified within a program segment, it may only be used within that segment and cannot be referenced from any other functions or subroutines, unlike the intrinsic or other user defined functions.

15.15.1 Example 11: Stirling's approximation

In this example we use Stirling's approximation for large *n*,

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

and a complete program to use this internal function is given below:

```
PROGRAM ch1511
IMPLICIT NONE
REAL :: Result,N,R
   PRINT *,' Type in N and R'
   READ *,N,R
! NUMBER OF POSSIBLE COMBINATIONS THAT CAN
! BE FORMED WHEN
! R OBJECTS ARE SELECTED OUT OF A GROUP OF N
! N!/R!(N-R)!
  Result=Stirling(N)/(Stirling(R)*Stirling(N-R))
  PRINT *,Result
  PRINT *,N,R
CONTAINS
REAL FUNCTION Stirling (X)
  REAL , INTENT(IN) :: X
  REAL , PARAMETER :: PI=3.1415927, E =2.7182828
  Stirling=SQRT(2.*PI*X) * (X/E)**X
END FUNCTION Stirling
END PROGRAM ch1511
```

The difference between this example and the earlier ones lies in the CONTAINS statement. The function is now an integral part of the program and could not, for example, be used elsewhere in another function. This provides us with a very powerful way of information hiding and making the construction of larger programs more secure and bug free.

15.16 Resumé

There are a large number of Fortran supplied functions and subroutines (intrinsic functions) which extend the power and scope of the language. Some of these functions are of *generic* type, and can take several different types of arguments. Others are restricted to a particular type of argument. Appendix D should be consulted for a fuller coverage concerning the rules that govern the use of the intrinsic functions and procedures.

When the intrinsic functions are inadequate, it is possible to write *user defined* functions. Besides expanding the scope of computation, such functions aid in problem visualisation and logical subdivision, may reduce duplication, and generally help in avoiding programming errors.

In addition to separately defined user functions, internal functions may be employed. These are functions which are used within a program segment.

Although the normal exit from a user defined function is through the END, other, *abnormal,* exits may be defined through the RETURN statement.

Communication with nonrecursive functions is through the function name and the function arguments. The function *must* contain a reference to the function name on the left-hand side of an assignment. Results may also be returned through the argument list.

We have also covered briefly the concept of scope for a variable, local variables, and argument association. This area warrants a much fuller coverage and we will do this after we have covered subroutines and modules.

15.17 Function syntax

The syntax of a function is:

[function prefix] function_statement &

[RESULT (Result_name)]

[specification part]

[execution_part]

[internal sub program part]

END [FUNCTION [function name]]

and prefix is:

[type specification] RECURSIVE

or

[RECURSIVE] type specification

and the function_statement is:

FUNCTION function_name ([dummy argument name list])

[ ] represent optional parts to the specification.

15.18 Rules and restrictions

The type of the function must only be specified once, either in the function statement or in a type declaration.

The names must match between the function header and END FUNCTION function name statement.

If there is a RESULT clause, that name must be used as the result variable, so all references to the function name are recursive calls.

The function name must be used to return a result when there is no RESULT clause.

We will look at additional rules and restrictions in later chapters.

15.19 Problems

1. Find out the action of the MOD function when one of the arguments is negative. Write your own modulus function to return only a positive remainder. Don't call it MOD!

2. Create a table which gives the sines, cosines and tangents for –1 to 91 degrees in 1 degree intervals. Remember that the arguments have to be in radians. What value will you give π? One possibility is π=4*atan(1.0). Pay particular attention to the following angle ranges:

- –1,0,+1
- 29,30,31
- 44,45,46
- 59,60,61
- 89,90,91

What do you notice about sine and cosine at 0 and 90 degrees? What do you notice about the tangent of 90 degrees? Why do you think this is?

Use a calculator to evaluate the sine, cosine at 0 and 90 degrees. Do the same for the tangent at 90 degrees. Does this surprise you?

Repeat using a spreadsheet, e.g., Excel.

Are you surprised?

Repeat the Fortran program using one or more real kind types.

3. Write a program that will read in the lengths *a* and *b* of a right-angled triangle and calculate the hypotenuse *c*. Use the Fortran SQRT intrinsic.

4. Write a program that will read in the lengths *a* and *b* of two sides of a triangle and the angle between them θ (in degrees). Calculate the length of the third side *c* using the cosine rule:

$$c^2 = a^2 + b^2 - 2abcos(\theta)$$

5. Write a function to convert an integer to a binary character representation. It should take an integer argument and return a character string that is a sequence of zeros and ones. Use the program in Chapter 8 as a basis for the solution.

15.20 Bibliography

Abramowitz M., Stegun I., *Handbook of Mathematical Functions*, Dover, 1968.

- This book contains a fairly comprehensive collection of numerical algorithms for many mathematical functions of varying degrees of obscurity. It is a widely used source.

Association of Computing Machinery (ACM)

- *Collected Algorithms*, 1960–1974

- *Transactions on Mathematical Software*, 1975 —
  A good source of more specialied algorithms. Early algorithms tended to be in Algol, Fortran now predominates.

15.20.1 Recursion and problem solving

The following are a number of books that look at the role of recursion in problem solving and algorithms.

Hofstader D. R., *Gödel, Escher, Bach — an Eternal Golden Braid*, Harvester Press.

- The book provides a stimulating coverage of the problems of paradox and contradiction in art, music and mathematics using the works of Escher, Bach and Gödel, and hence the title. There is a whole chapter on recursive structures and processes. The book also covers the work of Church and Turing, both of whom have made significant contributions to the theory of computing.

Kruse R.L., *Data Structures and Program Design*, Prentice-Hall, 1994.

- Quite a gentle introduction to the use of recursion and its role in problem solving. Good choice of case studies with explanations of solutions. Pascal is used.

Sedgewick R., *Algorithms in Modula 3*, Addison-Wesley, 1993.

- Good source of algorithms. Well written. The GCD algorithm was taken from this source.

Vowels R.A., *Algorithms and Data Structures in F and Fortran,* Unicomp, 1998.

- The only book currently that uses Fortran 90/95 and F. Visit the Fortran web site for more details. They are the publishers.

  http://www.fortran.com/fortran/market.html

Wirth N., *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

- In the context of this chapter the section on recursive algorithms is a very worthwhile investment in time.

Wood D., *Paradigms and Programming in Pascal*, Computer Science Press.

- Contains a number of examples of the use of recursion in problem solving. Also provides a number of useful case studies in problem solving.