Chapter 12
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.
- 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 for variables and functions.
- To look at internal user defined functions.

12.1 Introduction

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.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_12

- 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 sub-problem 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 sub-problems, 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(s) 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.

12.2 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.

12.2.1 Example 1: Simple Function Usage

A complete example is given below:

```
program ch1201
  implicit none
  real :: x

  print *, ' type in an angle (in radians)'
  read *, x
  print *, ' Sine of ', x, ' = ', sin(x)
end program ch1201
```

These functions are called intrinsic functions. Table 12.1 has details of some of the intrinsic functions available in Fortran.

Table 12.1 Some of the intrinsic functions available in Fortran

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 more complete list is given in Appendix D.

12.3 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.

12.3.1 Example 2: The `abs` Generic Function

The following short program illustrates this with the `abs` intrinsic function:

```
program ch1202
  implicit none
  complex :: c = cmplx(1.0, 1.0)
  real :: r = 10.9
  integer :: i = -27

  print *, abs(c)
  print *, abs(r)
  print *, abs(i)
end program ch1202
```

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

It is now possible with Fortran 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.

12.4 Elemental Functions

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

12.4.1 Example 3: Elemental Function Use

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

```
program ch1203
  implicit none
  real, dimension (5) :: x = (/ 1.0, 2.0, 3.0, &
    4.0, 5.0 /)
```

```
  print *, ' sine of ', x, ' = ', sin(x)
end program ch1203
```

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

12.5 Transformational Functions

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

12.5.1 Example 4: Simple Transformational Use

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

```
program ch1204
  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 ch1204
```

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.

12.5.2 Example 5: Intrinsic **dot_product** *Use*

The following program uses the transformational function dot_product:

```
program ch1205
  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 ch1205
```

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

12.6 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)
```

12.7 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 ch1206
  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
    end if
    if ((epact==25 .and. metcyc>11) .or. &
      epact==24) then
      epact = epact + 1
    end if
  end if
! finding the full moon
  luna = 44 - epact
  if (luna<21) then
    luna = luna + 30
  end if
! 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
```

```
  end if
end program ch1206
```

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.

12.8 Intrinsic Procedures

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

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

This appendix should be consulted for a more complete and thorough understanding of intrinsic procedures and their use in Fortran.

12.9 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.

12.9.1 Example 7: Simple User Defined Function

The following defines a function to achieve this:

```
module gcd_module

contains

  integer function gcd(a, b)
    implicit none
    integer, intent (in) :: a, b
    integer :: temp

    if (a<b) then
      temp = a
    else
      temp = b
    end if
    do while ((mod(a,temp)/=0) .or. (mod(b, &
      temp)/=0))
      temp = temp - 1
    end do
    gcd = temp
  end function gcd

end module gcd_module
```

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

```
program ch1207

  use gcd_module

  implicit none
  integer :: i, j, result

  print *, ' type in two integers'
```

```
    read *, i, j
    result = gcd(i, j)
    print *, ' gcd is ', result

end program ch1207
```

We will start by talking about the actual function and then cover the following statements

```
module gcd_module
contains
..
..
end module gcd_module
```

later in the chapter on modules.

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 `gcd`.
- 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 program unit in Fortran terminology. 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. It is good programming practice for functions not to have side effects, i.e not modify their arguments, and do no i/o.

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 cover the area of argument association in the next section.

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 Chap. 13.

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.

12.10 An Introduction to the Scope of Variables, Local Variables and Interface Checking

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 sub-problems. 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.

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.

A common programming error in Fortran 66 and 77 was mismatches between actual and dummy arguments. Problems caused by this were often very subtle and hard to find.

Fortran 90 introduced a solution to the problem via the use of modules and contains statements. We have added

```
module gcd_module
contains
..
end module gcd_module
```

around the function definition, which contains the function in a module and the following statement in the main program

```
use gcd_module
```

provides an explicit interface (in Fortran terminology) that requires the compiler to check at compile time that the call is correct, i.e. that there are the correct number of parameters, they are of the correct type and in this case that the function return type is correct. We will cover this area in greater depth in later chapters.

12.11 Recursive Functions

There is an additional form of the function header that was required when recursive function support was introduced in Fortran 90. The Fortran 2018 standard makes this form optional. Recursion means the breaking down of a problem into a simpler but identical sub-problem. 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.

12.11.1 Example 8: Recursive Factorial Evaluation

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

```
module factorial_module
  implicit none

contains
  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
end module factorial_module

program ch1208
  use factorial_module
  implicit none
  integer :: i, f

  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 ch1208
```

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.

12.12 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:

```
module gcd_module
  implicit none

contains
  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))
    end if
  end function gcd
end module gcd_module

program ch1209
  use gcd_module
  implicit none
  integer :: i, j, result

  print *, ' type in two integers'
  read *, i, j
  result = gcd(i, j)
```

```
  print *, ' gcd is ', result
end program ch1209
```

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 non-recursive version of the second algorithm.

12.13 Example 10: gcd After Removing Recursion

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

```
module gcd_module
  implicit none

contains
  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
end module gcd_module

program ch1210
  use gcd_module
  implicit none
  integer :: i, j, result

  print *, ' type in two integers'
  read *, i, j
  result = gcd(i, j)
  print *, ' gcd is ', result
end program ch1210
```

12.14 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.

12.14.1 Example 11: Stirling's Approximation

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

$$n! = \sqrt{2\pi n}(n/e)^n$$

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

```
program ch1211
  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 ch1211
```

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.

12.15 Pure Functions

We mentioned earlier that functions should not have side effects. If your functions do have side effects and are running the code on parallel systems we have the additional problem that it may not actually work! We would also like to be able to take advantage of automatic parallelisation if possible. In the following example we show how to do this using the pure prefix specification.

```
module gcd_module
  implicit none

contains
  pure integer function gcd(a, b)
    implicit none
    integer, intent (in) :: a, b
    integer :: temp

    if (a<b) then
      temp = a
    else
      temp = b
    end if
    do while ((mod(a,temp)/=0) .or. (mod(b, &
      temp)/=0))
      temp = temp - 1
    end do
    gcd = temp
  end function gcd
end module gcd_module

program ch1212
  use gcd_module
  implicit none
  integer :: i, j, result

  print *, ' type in two integers'
  read *, i, j
  result = gcd(i, j)
  print *, ' gcd is ', result
end program ch1212
```

Subroutines can also be made pure.

12.15.1 Pure Constraints

The following are some of the constraints on pure procedures

- a dummy argument must be `intent(in)`
- local variables may not have the `save` attribute
- no i/o must be done in the procedure
- any procedures referenced must be pure
- you cannot have a `stop` statement in a pure procedure

The above information should be enough to write simple pure functions.

12.16 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 squared the circle and enabled us to write elemental user defined functions. Here is an example to illustrate this.

```
module reciprocal_module

contains
  real elemental function reciprocal(a)
    implicit none
    real, intent (in) :: a

    reciprocal = 1.0/a
  end function reciprocal
end module reciprocal_module

program ch1213
  use reciprocal_module
  implicit none
  real :: x = 10.0
  real, dimension (5) :: y = [ 1.0, 2.0, 3.0, &
    4.0, 5.0 ]

  print *, ' reciprocal of x is ', reciprocal(x)
  print *, ' reciprocal of y is ', reciprocal(y)
end program ch1213
```

Here is the output from one compiler.

```
reciprocal of x is   0.1000000
reciprocal of y is   0.9999999
0.5000000      0.3333333
0.2500000      0.2000000
```

Hence we can call our own elemental functions with both scalar and array arguments.

Elemental functions require the use of explicit interfaces, and we have therefore used modules to achieve this.

12.17 Resume

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` statement, other, abnormal, exits may be defined through the `return` statement.

Communication with non-recursive 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.

12.18 Formal 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.
The simple syntax for a module as we have used them in this chapter is

```
module module_name
...
end module_name
```

and

```
use module_name
```

in the calling routine.

12.19 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.

12.20 Problems

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

12.2 Create a table which gives the sines, cosines and tangents for –1 to 91° in 1° intervals. Remember that the arguments have to be in radians. What value will you give *pi*? One possibility is `pi=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° ? What do you notice about the tangent of 90° ? Why do you think this is?

Use a calculator to evaluate the sine, cosine at 0 and 90°. do the same for the tangent at 90°. 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.

12.3 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.

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

12.5 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 Chap. 5 as a basis for the solution.

12.21 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 specialised algorithms. Early algorithms tended to be in Algol, Fortran now predominates.

12.21.1 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. Sadly no longer available. We found one at over 100 on Abebooks!

```
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.