Chapter 2.2: The structure of procedural programs

2.2 (a) Statement, subroutine, procedure, function, parameter, loop

Procedural programs are ones in which instructions are executed in the order defined by the programmer.

Procedural languages are often referred to as third generation languages and include FORTRAN, ALGOL, COBOL, BASIC, and PASCAL.

**Statement**
A statement is a single instruction in a program, which can be converted into machine code and executed.

In most languages a statement is written on a single line, but some languages allow multiple lines for single statements.

Examples of statements are:

```
DIM name As String
A=X*X
While x < 10
```

**Subroutine**
A subroutine is a self-contained section of program code that performs a specific task, as part of the main program.

**Procedure**
A procedure is a subroutine that performs a specific task without returning a value to the part of the program from which it was called.

**Function**
A function is a subroutine that performs a specific task and returns a value to the part of the program from which it was called.

Note that a function is ‘called’ by writing it on the right hand side of an assignment statement.

**Parameter**
A parameter is a value that is ‘received’ in a subroutine (procedure or function).

The subroutine uses the value of the parameter within its execution. The action of the subroutine will be different depending upon the parameters that it is passed.

Parameters are placed in parenthesis after the subroutine name. For example:

- `Square(5)` ‘passes the parameter 5 – returns 25
- `Square(8)` ‘passes the parameter 8 – returns 64
- `Square(x)` ‘passes the value of the variable x
2.2 (b) Identify the three basic programming constructs used to control the flow of execution: sequence, selection and iteration

Control structures
Computer programs execute statements one line after the next, unless there is a command that instructs the program to ‘jump’ backwards or forwards to an alternative line.

Sequence
Sequence is when the programming statements are executed one after the other, in the order in which they appear in the program.

Selection
Selection is a control structure in which there is a test to decide if certain instructions are executed.

\[
\text{If } x < 10 \text{ then} \\
\quad \text{Do this line} \\
\quad \text{And this line} \\
\text{Else} \\
\quad \text{Do this line} \\
\text{End if}
\]

Iteration
Iteration is a control structure in which a group of statements is executed repeatedly – either a set number of times, or until a specific condition is True.

\[
\text{Repeat} \\
\quad \text{Do this line} \\
\quad \text{And this line} \\
\text{Until } n=5
\]
2.2 (c) Understand and use selection in pseudocode and a procedural programming language, including the use of IF statements and CASE/SELECT statements

**IF-THEN-ELSE**
This selection method is used if there are two possible outcomes to a test:

```plaintext
IF x < 0 THEN
    OUTPUT “Sorry, you can’t have negative values”
ELSE
    a = x*x
    OUTPUT a
END
```

**SELECT-CASE**
This selection method is used if there are more than two possible outcomes to a test:

```plaintext
SELECT CASE KeyPress
    CASE LeftArrow
        Move one character backwards
    CASE RightArrow
        Move one character forwards
    CASE UpArrow
        Move one character up
    CASE DownArrow
        Move one character down
END SELECT
```
2.2 (d) Understand and use iteration in pseudocode and a procedural programming language, including the use of count-controlled loops (FOR-NEXT loops) and condition-controlled loops (WHILE-ENDWHILE and REPEAT-UNTIL loops)

**FOR-NEXT**
This is an unconditional loop in which the number of repetitions is set at the beginning.

```
FOR X = 1 TO 5
    Answer = X*3
    OUTPUT X, Answer
NEXT
```

**WHILE-ENDWHILE**
This is a conditional loop, which has a test at the start and repeats until the condition is false:

```
X = 0
WHILE X < 6 DO
    X = X + 1
    Answer = X*3
    OUTPUT X, Answer
ENDWHILE
```

**REPEAT-UNTIL**
This is a conditional loop, which has a test at the end and repeats until the condition is true:

```
X = 0
REPEAT
    X = X + 1
    Answer = X*3
    OUTPUT X, Answer
UNTIL X > 4
```
Comparison of the different iterations

<table>
<thead>
<tr>
<th>FOR-NEXT</th>
<th>WHILE-WEND</th>
<th>REPEAT-UNTIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>An unconditional loop – it will repeat itself a set number of times.</td>
<td>A conditional loop – it will repeat until a variable has reached a stated value.</td>
<td>A conditional loop – it will repeat until a variable has reached a stated value.</td>
</tr>
<tr>
<td>The condition is tested at the start of the loop.</td>
<td>The condition is tested at the end of the loop.</td>
<td></td>
</tr>
<tr>
<td>Iteration statements will always execute at least once.</td>
<td>Iteration statements may not actually be executed.</td>
<td>Iteration statements will always execute at least once.</td>
</tr>
</tbody>
</table>
| For x = 1 To 10            
  a=x*5
  Output x, a
  Next x                    | x-1                          
  While x < 11               
  a=x*5
  Output x, a
  x=x+1
  End While                  | x-1                          
  Repeat                     
  a=x*5
  Output x, a
  x=x+1
  Until x > 10                |
2.2 (e) Understand and use nested selection and nested iteration statements

**Nested selection**
This is where there is an IF statement contained within an IF statement.
The following algorithm allows a maximum of four attempts to login to a computer system:

```
INPUT Password
IF NumberOfTries < 5 THEN
  IF Password is correct THEN
    OUTPUT “Successful Login”
  ELSE
    OUTPUT “Password was incorrect”
  ENDIF
ELSE
  OUTPUT “You have made too many attempts”
ENDIF
```

**Nested iteration**
This is where there is a loop within a loop.
A nested iteration is needed to initialise a two-dimensional array:

```
FOR row = 0 TO 7
  FOR column = 0 TO 5
    SET MyArray (row, column) = 0
  NEXT column
NEXT row
```
2.2 (f) Understand, create and use subroutines (procedures and functions), including the passing of parameters and the appropriate use of the return value of functions

2.2 (g) Use subroutines to modularise the solution to a problem

Subroutine/sub-program
A subroutine is a self-contained section of program code which performs a specific task and is referenced by a name.

A subroutine resembles a standard program in that it will contain its own local variables, data types, labels and constant declarations.

There are two types of subroutine. These are procedures and functions.

- procedures are subroutines that input, output or manipulate data in some way;
- functions are subroutines that return a value to the main program.

A subroutine is executed whenever its name is encountered in the executable part of the main program. The execution of a subroutine by referencing its name in the main program is termed ‘calling’ the subroutine.

The advantages of subroutines.
The advantage of using procedures and functions are that:

- the same lines of code are re-used whenever they are needed – they do not have to be repeated in different sections of the program.
- a procedure or function can be tested/improved/rewritten independently of other procedures or functions.
- it is easy to share procedures and functions with other programs – they can be incorporated into library files which are then ‘linked’ to the main program;
- a programmer can create their own routines that can be called in the same way as any built-in command.
2.2 (h) Identify and use recursion to solve problems; show an understanding of the structure of a recursive subroutine, including the necessity of a stopping condition.

2.2 (i) Trace the execution of a recursive subroutine.

2.2 (j) Discuss the relative merits of iterative and recursive solutions to the same problem.

Recursion is when a function or a procedure contains a call to itself. Recursion is used when calculating factorials:

```
FUNCTION Factorial(n)
 IF n = 1 THEN
   RETURN 1
 ELSE
   RETURN n * Factorial(n - 1)
 END IF
END
```

Note that a recursive function needs a ‘stopping condition’, i.e., a statement that will stop it from repeating infinitely.

In the Factorial function above the stopping condition is IF n = 1.

**Tracing the execution of x = Factorial(3)**

```
01 FUNCTION Factorial(n)
02   IF n = 1 THEN
03     RETURN 1
04   ELSE
05     RETURN n * Factorial(n - 1)
06   END IF
07 END FUNCTION
```

Note that in the above algorithm, the recursion ‘happens’ on line 05. See tracing table on next page for this Recursion.
<table>
<thead>
<tr>
<th>Line Number</th>
<th>Function Call</th>
<th>Returned Value</th>
<th>Condition</th>
<th>State of Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Factorial(3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>Factorial(2)</td>
<td></td>
<td>False</td>
<td>New Call: n=2</td>
</tr>
<tr>
<td>05</td>
<td>Factorial(1)</td>
<td>3 * Factorial(2)</td>
<td>False</td>
<td>New Call: n=1</td>
</tr>
<tr>
<td>01</td>
<td>Factorial(1)</td>
<td></td>
<td>True</td>
<td>n=1: Completed</td>
</tr>
<tr>
<td>02</td>
<td>Factorial(1)</td>
<td></td>
<td></td>
<td>n=2: Completed</td>
</tr>
<tr>
<td>07</td>
<td>Factorial(1)</td>
<td>2 * 1 = 2</td>
<td></td>
<td>n=2: Resumed</td>
</tr>
<tr>
<td>05</td>
<td>Factorial(1)</td>
<td></td>
<td></td>
<td>n=3: Completed</td>
</tr>
<tr>
<td>07</td>
<td>Factorial(1)</td>
<td></td>
<td></td>
<td>n=3: Completed</td>
</tr>
</tbody>
</table>
## Iteration compared to recursion

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Recursion</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 INPUT n</td>
<td>01 FUNCTION Factorial(n)</td>
</tr>
<tr>
<td>02 a=1</td>
<td>02 IF n=1 THEN</td>
</tr>
<tr>
<td>03 FOR i=1 TO n</td>
<td>03 RETURN 1</td>
</tr>
<tr>
<td>04 a=a*n</td>
<td>04 ELSE</td>
</tr>
<tr>
<td>05 NEXT i</td>
<td>05 RETURN n*Factorial(n-1)</td>
</tr>
<tr>
<td>06 OUTPUT a</td>
<td>06 END IF</td>
</tr>
<tr>
<td>07 END</td>
<td>07 END</td>
</tr>
</tbody>
</table>

There is only one set of variable values that are used within the iteration. A recursive subroutine is called repeatedly and each call has its own set of variables that are distinct from the variables in the other function calls.

Iteration requires less memory than recursion – because it only uses one set of variable values. There needs to be a stopping condition that causes the subroutine to stop without calling itself.

Iteration variables may need to be initialised and the value of the tracking variable may need to be tested for each iteration. Recursive subroutines are often more elegant and easier to understand than the equivalent iterations.