Pseudocode

The following information sets out how pseudocode will appear within the examinations of this syllabus. The numbers and letters that appear at the end of a sub-heading provide a cross reference to the relevant section of the subject content.

General style

Font style and size

Pseudocode is presented in `Courier New`. The size of the font will be consistent throughout.

Indentation

Lines are indented by four spaces to indicate that they are contained within a statement in a previous line. Where it is not possible to fit a statement on one line any continuation lines are indented by two spaces from the margin. In cases where line numbering is used, this indentation may be omitted. Every effort will be made to make sure that code statements are not longer than a line of code, unless this is necessary.

Note that the `THEN` and `ELSE` clauses of an `IF` statement are indented by only two spaces. Cases in `CASE` statements are also indented by only two spaces.

Case

Keywords are in upper case, e.g. `IF`, `REPEAT`, `PROCEDURE`.

Identifiers are in mixed case with upper case letters indicating the beginning of new words, e.g. `NumberOfPlayers`.

Meta-variables – symbols in the pseudocode that should be substituted by other symbols are enclosed in angled brackets `< >`.

Example – meta-variables

```
REPEAT
     <Statements>
UNTIL <Condition>
```

Lines and line numbering

Each line representing a statement is numbered. However, when a statement runs over one line of text, the continuation lines are not numbered.

Comments

Comments are preceded by two forward slashes `//`. The comment continues until the end of the line. For multi-line comments, each line is preceded by `//`.

Normally the comment is on a separate line before, and at the same level of indentation as, the code it refers to. Occasionally, however, a short comment that refers to a single line may be at the end of the line to which it refers.

Example – comments

```
// This procedure swaps
// values of X and Y
PROCEDURE SWAP(X : INTEGER, Y : INTEGER)
    Temp ← X    // temporarily store X
    X ← Y
    Y ← Temp
ENDPROCEDURE
```

Variables, constants and data types

Basic data types (8.1.2)

The following keywords are used to designate basic data types:

- `INTEGER` a whole number
- `REAL` a number capable of containing a fractional part
- `CHAR` a single character
- `STRING` a sequence of zero or more characters
- `BOOLEAN` the logical values `TRUE` and `FALSE`

Literals

Literals of the above data types are written as follows:

- Integer written as normal in the denary system, e.g. `5`, `-3`
- Real always written with at least one digit on either side of the decimal point, zeros being added if necessary, e.g. `4.7`, `0.3`, `-4.0`, `0.0`
- Char a single character delimited by single quotes, e.g. `'x'`, `'c'`, `'@'`
- String delimited by double quotes. A string may contain no characters (i.e. the empty string), e.g. `"This is a string"`, `""`
- Boolean `TRUE`, `FALSE`

Identifiers

Identifiers (the names given to variables, constants, procedures and functions) are in mixed case using Pascal case, e.g. `FirstName`. They can only contain letters (`A–Z`, `a–z`) and digits (`0–9`). They must start with a capital letter and not a digit. Accented letters and other characters, including the underscore, should not be used.

As in programming, it is good practice to use identifier names that describe the variable, procedure or function to which they refer. Single letters may be used where these are conventional (such as `i` and `j` when dealing with array indices, or `X` and `Y` when dealing with coordinates) as these are made clear by the convention.

Keywords should never be used as identifier names.

Identifiers should be considered case insensitive, for example, `Countdown` and `CountDown` should not be used as separate variables.

Variable declarations (8.1.1)

Declarations are made as follows:

```
DECLARE <identifier> : <data type>
```

Example – variable declarations

```
DECLARE Counter : INTEGER
DECLARE TotalToPay : REAL
DECLARE GameOver : BOOLEAN
```

Constants (8.1.1)

It is good practice to use constants if this makes the pseudocode more readable, and easier to update if the value of the constant changes.

Constants are declared by stating the identifier and the literal value in the following format:

```
CONSTANT <identifier> ← <value>
```

Example – `CONSTANT` declarations

```
CONSTANT HourlyRate ← 6.50
CONSTANT DefaultText ← "N/A"
```

Only literals can be used as the value of a constant. A variable, another constant or an expression must never be used.

Assignments

The assignment operator is ←

Assignments should be made in the following format:

```
<identifier> ← <value>
```

The identifier must refer to a variable (this can be an individual element in a data structure such as an array or an abstract data type). The value may be any expression that evaluates to a value of the same data type as the variable.

Example – assignments

```
Counter ← 0
Counter ← Counter + 1
TotalToPay ← NumberOfHours * HourlyRate
```

Arrays

Declaring arrays (8.2.1)

Arrays are fixed-length structures of elements of identical data type, accessible by consecutive index numbers. It is good practice to explicitly state what the lower bound of the array (i.e. the index of the first element) is because this defaults to either 0 or 1 in different systems. Generally, a lower bound of 1 will be used.

Square brackets are used to indicate the array indices.

1D and 2D arrays are declared as follows (where `l`, `l1`, `l2` are lower bounds and `u`, `u1`, `u2` are upper bounds):

```
DECLARE <identifier> : ARRAY[<l>:<u>] OF <data type>
DECLARE <identifier> : ARRAY[<l1>:<u1>, <l2>:<u2>] OF <data type>
```

Example – array declaration

```
DECLARE StudentNames : ARRAY[1:30] OF STRING
DECLARE NoughtsAndCrosses : ARRAY[1:3, 1:3] OF CHAR
```

Using arrays (8.2.1)

In the main pseudocode statements, only one index value is used for each dimension in the square brackets.

Example – using arrays

```
StudentNames[1] ← "Ali"
NoughtsAndCrosses[2,3] ← 'X'
StudentNames[n+1] ← StudentNames[n]
```

An appropriate loop structure is used to assign the elements individually.

Example – assigning a group of array elements

```
FOR Index ← 1 TO 30
    StudentNames[Index] ← ""
NEXT Index
```

Common operations

Input and output (8.1.3)

Values are input using the `INPUT` command as follows:

```
INPUT <identifier>
```

The identifier should be a variable (that may be an individual element of a data structure such as an array).

Values are output using the `OUTPUT` command as follows:

```
OUTPUT <value(s)>
```

Several values, separated by commas, can be output using the same command.

Examples – INPUT and OUTPUT statements

```
INPUT Answer
OUTPUT Score
OUTPUT "You have ", Lives, " lives left"
```

Arithmetic operations (8.1.4 (f))

Standard arithmetic operator symbols are used:

+	addition
–	subtraction
*	multiplication
/	division
^	raised to the power of

Examples – arithmetic operations

```
Answer ← Score * 100 / MaxMark
Answer ← Pi * Radius ^ 2
```

The integer division operators MOD and DIV can also be used.

DIV(<identifier1>, <identifier2>)
Returns the quotient of identifier1 divided by identifier2 with the fractional part discarded.

MOD(<identifier1>, <identifier2>)
Returns the remainder of identifier1 divided by identifier2

The identifiers are of data type integer.

Examples – MOD and DIV

```
DIV(10, 3) returns 3
MOD(10, 3) returns 1
```

Multiplication and division have higher precedence over addition and subtraction (this is the normal mathematical convention). However, it is good practice to make the order of operations in complex expressions explicit by using parentheses.

Logical operators (8.1.4 (f))

The following symbols are used for logical operators:

=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to

The result of these operations is always of data type BOOLEAN.

In complex expressions, it is advisable to use parentheses to make the order of operations explicit.

Boolean operators (8.1.4 (f))

The only Boolean operators used are AND, OR and NOT. The operands and results of these operations are always of data type BOOLEAN.

In complex expressions, it is advisable to use parentheses to make the order of operations explicit.

Examples – Boolean operations

```
IF Answer < 0 OR Answer > 100
  THEN
    Correct ← FALSE
  ELSE
    Correct ← TRUE
ENDIF
```

String operations (8.1.4 (e))

`LENGTH(<identifier>)`

Returns the integer value representing the length of string. The identifier should be of data type string.

`LCASE(<identifier>)`

Returns the string/character with all characters in lower case. The identifier should be of data type string or char.

`UCASE(<identifier>)`

Returns the string/character with all characters in upper case. The identifier should be of data type string or char.

`SUBSTRING(<identifier>, <start>, <length>)`

Returns a string of length `length` starting at position `start`. The identifier should be of data type string, length and start should be positive and data type integer.

Generally, a start position of 1 is the first character in the string.

Example – string operations

`LENGTH("Happy Days")` will return `10`
`LCASE('W')` will return `'w'`
`UCASE("Happy")` will return `"HAPPY"`
`SUBSTRING("Happy Days", 1, 5)` will return `"Happy"`

Other library routines (8.1.7)

`ROUND(<identifier>, <places>)`

Returns the value of the identifier rounded to `places` number of decimal places.
The identifier should be of data type real, `places` should be data type integer.

`RANDOM()`

Returns a random number between 0 and 1 inclusive.

Example – `ROUND` and `RANDOM`

```
Value ← ROUND (RANDOM() * 6, 0) // returns a whole number between 0 and 6
```

Selection

IF statements (8.1.4 (b) and 8.1.5)

IF statements may or may not have an ELSE clause.

IF statements without an ELSE clause are written as follows:

```
IF <condition>
  THEN
    <statements>
ENDIF
```

IF statements with an ELSE clause are written as follows:

```
IF <condition>
  THEN
    <statements>
  ELSE
    <statements>
ENDIF
```

Note that the THEN and ELSE clauses are only indented by two spaces. (They are, in a sense, a continuation of the IF statement rather than separate statements.)

When IF statements are nested, the nesting should continue the indentation of two spaces.

Example – nested IF statements

```
IF ChallengerScore > ChampionScore
  THEN
    IF ChallengerScore > HighestScore
      THEN
        OUTPUT ChallengerName, " is champion and highest scorer"
      ELSE
        OUTPUT Player1Name, " is the new champion"
    ENDIF
  ELSE
    OUTPUT ChampionName, " is still the champion"
    IF ChampionScore > HighestScore
      THEN
        OUTPUT ChampionName, " is also the highest scorer"
    ENDIF
ENDIF
```

CASE statements (8.1.4 (b))

CASE statements allow one out of several branches of code to be executed, depending on the value of a variable.

CASE statements are written as follows:

```
CASE OF <identifier>
  <value 1> : <statement>
  <value 2> : <statement>
  ...
ENDCASE
```

An OTHERWISE clause can be the last case:

```
CASE OF <identifier>
  <value 1> : <statement>
  <value 2> : <statement>
  ...
  OTHERWISE <statement>
ENDCASE
```

It is best practice to keep the branches to single statements as this makes the pseudocode more readable. Similarly, single values should be used for each case. If the cases are more complex, the use of an IF statement, rather than a CASE statement, should be considered.

Each case clause is indented by two spaces. They can be considered as continuations of the CASE statement rather than new statements.

Note that the case clauses are tested in sequence. When a case that applies is found, its statement is executed, and the CASE statement is complete. Control is passed to the statement after the ENDCASE. Any remaining cases are not tested.

If present, an OTHERWISE clause must be the last case. Its statement will be executed if none of the preceding cases apply.

Example – formatted CASE statement

```
INPUT Move
CASE OF Move
  'W' : Position ← Position - 10
  'E' : Position ← Position + 10
  'A' : Position ← Position - 1
  'D' : Position ← Position + 1
  OTHERWISE OUTPUT "Beep"
ENDCASE
```

Iteration

Count-controlled (FOR) loops (8.1.4 (c))

Count-controlled loops are written as follows:

```
FOR <identifier> ← <value1> TO <value2>
    <statements>
NEXT <identifier>
```

The identifier must be a variable of data type INTEGER, and the values should be expressions that evaluate to integers.

The variable is assigned each of the integer values from value1 to value2 inclusive, running the statements inside the FOR loop after each assignment. If value1 = value2 the statements will be executed once, and if value1 > value2 the statements will not be executed.

An increment can be specified as follows:

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
    <statements>
NEXT <identifier>
```

The increment must be an expression that evaluates to an integer. In this case the identifier will be assigned the values from value1 in successive increments of increment until it reaches value2. If it goes past value2, the loop terminates. The increment can be negative.

Example – nested FOR loops

```
Total ← 0
FOR Row ← 1 TO MaxRow
    RowTotal ← 0
    FOR Column ← 1 TO 10
        RowTotal ← RowTotal + Amount[Row, Column]
    NEXT Column
    OUTPUT "Total for Row ", Row, " is ", RowTotal
    Total ← Total + RowTotal
NEXT Row
OUTPUT "The grand total is ", Total
```

Post-condition (REPEAT) loops (8.1.4 (c))

Post-condition loops are written as follows:

```
REPEAT
    <Statements>
UNTIL <condition>
```

The condition must be an expression that evaluates to a Boolean. The statements in the loop will be executed at least once. The condition is tested after the statements are executed and if it evaluates to TRUE the loop terminates, otherwise the statements are executed again.

Example – REPEAT UNTIL statement

```
REPEAT
    OUTPUT "Please enter the password"
    INPUT Password
UNTIL Password = "Secret"
```

Pre-condition (WHILE) loops (8.1.4 (c))

Pre-condition loops are written as follows:

```
WHILE <condition> DO
   <statements>
ENDWHILE
```

The condition must be an expression that evaluates to a Boolean. The condition is tested before the statements, and the statements will only be executed if the condition evaluates to TRUE. After the statements have been executed the condition is tested again. The loop terminates when the condition evaluates to FALSE.

The statements will not be executed if, on the first test, the condition evaluates to FALSE.

Example – WHILE loop

```
WHILE Number > 9 DO
      Number ← Number - 9
ENDWHILE
```

Procedures and functions

Procedures and functions are defined at the start of the code.

Defining and calling procedures (8.1.6 (b))

A procedure with no parameters is defined as follows:

```
PROCEDURE <identifier>
    <statements>
ENDPROCEDURE
```

A procedure with parameters is defined as follows:

```
PROCEDURE <identifier>(<param1>:<datatype>, <param2>:<datatype>...)
    <statements>
ENDPROCEDURE
```

The `<identifier>` is the identifier used to call the procedure. Where used, `param1`, `param2`, etc. are identifiers for the parameters of the procedure. These will be used as variables in the statements of the procedure.

Procedures should be called as follows:

```
CALL <identifier>
```

```
CALL <identifier>(Value1,Value2...)
```

These calls are complete program statements.

When parameters are used, `Value1`, `Value2...` must be of the correct data type as in the definition of the procedure.

When the procedure is called, control is passed to the procedure. If there are any parameters, these are substituted by their values, and the statements in the procedure are executed. Control is then returned to the line that follows the procedure call.

Example – use of procedures with and without parameters

```
PROCEDURE DefaultLine
    CALL LINE(60)
ENDPROCEDURE

PROCEDURE Line(Size : INTEGER)
    DECLARE Length : INTEGER
    FOR Length ← 1 TO Size
        OUTPUT '-'
    NEXT Length
ENDPROCEDURE

IF MySize = Default
  THEN
    CALL DefaultLine
  ELSE
    CALL Line(MySize)
ENDIF
```

Defining and calling functions (8.1.6 (b))

Functions operate in a similar way to procedures, except that in addition they return a single value to the point at which they are called. Their definition includes the data type of the value returned.

A function with no parameters is defined as follows:

```
FUNCTION <identifier> RETURNS <data type>
   <statements>
ENDFUNCTION
```

A function with parameters is defined as follows:

```
FUNCTION <identifier>(<param1>:<datatype>, <param2>:<datatype>...) RETURNS <data type>
   <statements>
ENDFUNCTION
```

The keyword `RETURN` is used as one of the statements within the body of the function to specify the value to be returned. Normally, this will be the last statement in the function definition.

Because a function returns a value that is used when the function is called, function calls are not complete program statements. The keyword `CALL` should not be used when calling a function. Functions should only be called as part of an expression. When the `RETURN` statement is executed, the value returned replaces the function call in the expression and the expression is then evaluated.

Example – definition and use of a function

```
FUNCTION SumSquare(Number1:INTEGER, Number2:INTEGER) RETURNS INTEGER
    RETURN Number1 * Number1 + Number2 * Number2
ENDFUNCTION

OUTPUT "Sum of squares = ", SumSquare(10, 20)
```

File handling

Handling files (8.3.2)

It is good practice to explicitly open a file, stating the mode of operation, before reading from or writing to it. This is written as follows:

```
OPENFILE <File identifier> FOR <File mode>
```

The file identifier will be the name of the file with data type string. The following file modes are used:

- `READ` for data to be read from the file
- `WRITE` for data to be written to the file. A new file will be created and any existing data in the file will be lost.

A file should be opened in only one mode at a time.

Data is read from the file (after the file has been opened in `READ` mode) using the `READFILE` command as follows:

```
READFILE <File Identifier>, <Variable>
```

When the command is executed, the data item is read and assigned to the variable.

Data is written into the file after the file has been opened using the `WRITEFILE` command as follows:

```
WRITEFILE <File identifier>, <Variable>
```

When the command is executed, the data is written into the file. Files should be closed when they are no longer needed using the `CLOSEFILE` command as follows:

```
CLOSEFILE <File identifier>
```

Example – file handling operations

This example uses the operations together, to copy a line of text from `FileA.txt` to `FileB.txt`

```
DECLARE LineOfText : STRING
OPENFILE FileA.txt FOR READ
OPENFILE FileB.txt FOR WRITE
READFILE FileA.txt, LineOfText
WRITEFILE FileB.txt, LineOfText
CLOSEFILE FileA.txt
CLOSEFILE FileB.txt
```