Pseudocode Guide for H2 Computing (9569)

This guide includes reference from Cambridge International AS & A Level Computer Science Pseudocode Guide for Teachers and HCI Computing teachers' valuable comments from their teaching experiences.

By definition, pseudocode is not a programming language with a defined, mandatory syntax. This guide will enable students to understand any pseudocode presented in teaching materials and examination papers. It gives a structure to follow so that students can present their algorithms more clearly in pseudocode when required. Any pseudocode presented by candidates will be mainly assessed for the logic of the solution presented – where the logic is understood by the Examiner, and correctly solves the problem addressed, the candidate will be given credit regardless of whether the candidate has followed the style presented here. Using a recommended style will, however, enable the candidate to communicate their solution to the Examiner more effectively.

1. Pseudocode in examined components

1.1 Font Style and size

Pseudocode is presented in a monospaced (fixed-width) font such as `Courier New`. The size of the font will be consistent throughout.

1.2 Uppercase and Lowercase

Keywords are in uppercase, e.g. `IF`, `REPEAT`, `PROCEDURE`. (Different keywords are explained in later sections of this guide.)

Identifiers are in mixed case (sometimes referred to as camelCase or Pascal case) with uppercase letters indicating the beginning of new words, for example `NumberOfPlayers`.

Meta-variables – symbols in the pseudocode that should be substituted by other symbols are enclosed in angled brackets < > (as in Backus-Naur Form). This is also used in this guide.

Example – meta-variables

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

1.3 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(BYREF X : INTEGER, Y INTEGER)
    Temp ← X   // temporarily store X
    X ← Y
    Y ← Temp
ENDPROCEDURE
```

2. Variables, constants, and data types

2.1 Atomic type names

The following keywords are used to designate atomic 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`
• `DATE`:	A valid calendar date

2.2 Variable declarations

It is good practice to declare variables explicitly in pseudocode.

Declarations are made as follows:

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

Example – variable declarations

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

2.3 Constants

It is good practice to use constants if this makes the pseudocode more readable, as an identifier is more meaningful in many cases than a literal. It also makes the pseudocode easier to update if the value of the constant changes.

Constants are normally declared at the beginning of a piece of pseudocode (unless it is desirable to restrict the scope of the constant).

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.

2.4 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
```

3. Arrays

3.1 Declaring arrays

Arrays are considered to be fixed-length structures of elements of identical data type, accessible by consecutive index (subscript) 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.

One-dimensional and two-dimensional 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
```

3.2 Using arrays

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]
```

Arrays can be used in assignment statements (provided they have same size and data type). The following is therefore allowed:

Example – assigning an array

```
SavedGame ← NoughtsAndCrosses
```

A statement should **not**, however, refer to a group of array elements individually. For example, the following construction should not be used.

```
StudentNames [1 TO 30] ← ""
```

Instead, an appropriate loop structure is used to assign the elements individually. For example:

Example – assigning a group of array elements

```
FOR Index = 1 TO 30
    StudentNames[Index] ← ""
ENDFOR Index
```

4. Common operations

4.1 Input and output

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, or a custom data type).

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"
```

4.2 Arithmetic operations

Standard arithmetic operator symbols are used:

+	Addition
-	Subtraction
*	Multiplication
/	Division

Care should be taken with the division operation: the resulting value should be of data type `REAL`, even if the operands are integers.

The integer division operators `MOD` and `DIV` can be used. However, their use should be explained explicitly and not assumed.

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.

Note that augmented assignment in Python, for example, `a += 3`, are not considered as pseudocode. We should use `a ← a + 3` instead.

4.3 Relational operations

The following symbols are used for relational operators (also known as comparison operators):

>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	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.

4.4 Logic operators

The only logic operators (also called relational 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.

4.5 String operations

A statement should not refer to a group of characters in the string. For example, Python codes

```
Extract = Sentence[1:6]
```

canont be written as pseudocode

```
Extract ← Sentence [1 TO 5]
```

Instead, an appropriate loop structure is used.

Example – concatenation of a string

```
Extract ← ""
FOR INDEX ← 1 TO 5
    Extract ← Extract + Sentence[Index]
ENDFOR
```

The string methods in Python are not considered as pseudocode.

5. Selection

5.1 IF statements

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
```

IF statements with more conditions are written as follows:

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

5.2 CASE statements

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 seen 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
  'S': Position ← Position + 10
  'A': Position ← Position - 1
  'D': Position ← Position + 1
  OTHERWISE : Beep
ENDCASE
```

6. Iteration

6.1 Count-controlled (FOR) loops

FOR loop on INTEGER:

Note that Python function `range` is not considered as pseudocode.

Count-controlled loops are written as follows:

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

An increment can be specified as follows:

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

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.

FOR loop on STRING:

Python code `for char in aString` is not applicable to other programming languages and hence not considered as pseudocode. A string is a collection of characters that can be accessed via index. To explain traversal in the string with pseudocode, we will make use of the index.

```
FOR Index ← 1 TO LENGTH(aString)

     <statements using aString[Index]>

ENDFOR
```

6.2 Post-condition (REPEAT UNTIL) loops

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"
```

6.3 Pre-condition (WHILE) loops

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
```

7. Procedures

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 defined as above should be called as follows, respectively:

```
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.

Optional parameters and overloaded procedures (where alternative definitions are given for the same identifier with different sets of parameters) should be avoided in pseudocode.

Unless otherwise stated, it should be assumed that parameters are passed by value. (See section 8.3).

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 DefaultSquare
    CALL Square(100)
ENDPROCEDURE

PROCEDURE Square(Size : integer)
    FOR Side ← 1 TO 4
       MoveForward Size
       Turn 90
    ENDFOR
ENDPROCEDURE

IF Size = Default
  THEN
    CALL DefaultSquare
  ELSE
    CALL Square(Size)
ENDIF
```

8. Functions

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 procedure with no parameters is defined as follows:

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

A procedure 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 Max(Number1:INTEGER, Number2:INTEGER) RETURNS INTEGER
    IF Number1 > Number2
      THEN
        RETURN Number1
      ELSE
        RETURN Number2
    ENDIF
ENDFUNCTION

OUTPUT "Penalty Fine = ", Max(10,Distance*2)
```