

## Notes and guidance: Pseudo-code

---

The pseudo-code is described in this resource to help students prepare for their AQA GCSE Computer Science exam (8525/1).

We will use this consistent style of pseudo-code in all assessment material. This will ensure that, with enough preparation, students will understand the syntax of the pseudo-code used in assessments. Students do not have to use this style of pseudo-code in their own work or written assessments, although they are free to do so. The only direction to students when answering questions or describing algorithms written in pseudo-code is that their code is clear, consistent and unambiguous.

This resource may be updated as required and the latest version will always be available on our website. It is not confidential and can be freely shared with students.

### General Syntax

- `IntExp`, `RealExp`, `BoolExp`, `CharExp` and `StringExp` mean any expression which can be evaluated to an integer, real, Boolean (`False` or `True`), character or string respectively.
- `Exp` means any expression.
- **Emboldened pseudo-code** is used to indicate the keywords/operators.
- Exam paper questions will assume that indexing for arrays and strings starts at 0 unless specifically stated otherwise.

## Comments

Single line comments	# comment	
Multi-line comments	# comment # comment and so on	

## Variables and constants

Variable assignment	Identifier ← Exp	a ← 3 b ← a + 1 c ← 'Hello'
Constant assignment	<b>CONSTANT</b> IDENTIFIER ← Exp	<b>CONSTANT</b> PI ← 3.141 <b>CONSTANT</b> CLASS_SIZE ← 23  # Names of constants will always be # written in capitals

## Arithmetic operations

Standard arithmetic operations	$+$ $-$ $*$ $/$	<p>Used in the normal way with brackets to indicate precedence where needed. For example, <math>a + b * c</math> would multiply <math>b</math> and <math>c</math> together and then add the result to <math>a</math>, whereas <math>(a + b) * c</math> would add <math>a</math> and <math>b</math> together and then multiply the result by <math>c</math>.</p> <p>The <math>/</math> symbol is used instead of <math>\div</math> for division (for integer division use <b>DIV</b>)</p>
Integer division	IntExp <b>DIV</b> IntExp	$9 \text{ DIV } 5$ evaluates to 1 $5 \text{ DIV } 2$ evaluates to 2 $8 \text{ DIV } 4$ evaluates to 2
Modulus operator	IntExp <b>MOD</b> IntExp	$9 \text{ MOD } 5$ evaluates to 4 $5 \text{ MOD } 2$ evaluates to 1 $8 \text{ MOD } 4$ evaluates to 0

## Relational operators for types that can be clearly ordered

Less than	$\text{Exp} < \text{Exp}$	$4 < 6$ $'A' < 'B'$ $'adam' < 'adele'$
Greater than	$\text{Exp} > \text{Exp}$	$4.1 > 4.0$
Equal to	$\text{Exp} = \text{Exp}$	$3 = 3$
Not equal to	$\text{Exp} \neq \text{Exp}$	$\text{qty} \neq 7$
Less than or equal to	$\text{Exp} \leq \text{Exp}$	$3 \leq 4$ $4 \leq 4$
Greater than or equal to	$\text{Exp} \geq \text{Exp}$	$4 \geq 3$ $4.5 \geq 4.5$

## Boolean operations

Logical AND	$\text{BoolExp} \text{ AND } \text{BoolExp}$	$(3 = 3) \text{ AND } (3 \leq 4)$
Logical OR	$\text{BoolExp} \text{ OR } \text{BoolExp}$	$(x < 1) \text{ OR } (x > 9)$
Logical NOT	$\text{NOT } \text{BoolExp}$	$\text{NOT } (a < b)$

## Indefinite (condition controlled) iteration

<p>REPEAT-UNTIL (repeat the statements until the Boolean expression is True).</p>	<pre><b>REPEAT</b>      # statements here  <b>UNTIL</b> BoolExp</pre>	<pre>a ← 1 <b>REPEAT</b>     <b>OUTPUT</b> a     a ← a + 1 <b>UNTIL</b> a = 4 # will output 1, 2, 3</pre>
<p>WHILE-ENDWHILE (while the Boolean expression is True, repeat the statements).</p>	<pre><b>WHILE</b> BoolExp      # statements here  <b>ENDWHILE</b></pre>	<pre>a ← 1 <b>WHILE</b> a &lt; 4     <b>OUTPUT</b> a     a ← a + 1 <b>ENDWHILE</b> # will output 1, 2, 3</pre>

## Definite (count controlled) iteration

<p><b>FOR-TO-[STEP]-ENDFOR</b> (If <b>STEP</b> IntExp is missing it is considered to be 1).</p>	<pre>FOR Identifier ← IntExp TO IntExp [STEP IntExp]     # statements here ENDFOR  # If <b>STEP</b> IntExp is omitted the step value is 1. # Note that in <b>STEP</b> IntExp the value of IntExp # can be negative (see the third example)</pre>	<pre>FOR a ← 1 TO 3     OUTPUT a ENDFOR # will output 1, 2, 3  FOR a ← 1 TO 5 STEP 2     OUTPUT a ENDFOR # will output 1, 3, 5  FOR a ← 5 TO 1 STEP -2     OUTPUT a ENDFOR # will output 5, 3, 1</pre>
<p><b>FOR-IN-ENDFOR</b> (repeat the statements the number of times that there are characters in a string).</p>	<pre>FOR Identifier IN StringExp     # statements here ENDFOR</pre>	<pre>length ← 0 FOR char IN message     length ← length + 1 ENDFOR # will calculate the # number of characters # in message  reversed ← '' FOR char IN message     reversed ← char + reversed ENDFOR OUTPUT reversed # will output the # string in reverse</pre>

## Selection

<p>IF-THEN-ENDIF (execute the statements only if the Boolean expression is True).</p>	<pre> <b>IF</b> BoolExp <b>THEN</b>     # statements here <b>ENDIF</b>         </pre>	<pre> a ← 1 <b>IF</b> (a <b>MOD</b> 2) = 0 <b>THEN</b>     <b>OUTPUT</b> 'even' <b>ENDIF</b>         </pre>
<p>IF-THEN-ELSE-ENDIF (execute the statements following the THEN if the Boolean expression is True, otherwise execute the statements following the ELSE).</p>	<pre> <b>IF</b> BoolExp <b>THEN</b>     # statements here <b>ELSE</b>     # statements here <b>ENDIF</b>         </pre>	<pre> a ← 1 <b>IF</b> (a <b>MOD</b> 2) = 0 <b>THEN</b>     <b>OUTPUT</b> 'even' <b>ELSE</b>     <b>OUTPUT</b> 'odd' <b>ENDIF</b>         </pre>
<p>NESTED IF-THEN-ELSE ENDIF (use nested versions of the above to create more complex conditions).</p> <p>Note that IF statements can be nested inside the THEN part, the ELSE part or both.</p>	<pre> <b>IF</b> BoolExp <b>THEN</b>     # statements here <b>ELSE</b>     <b>IF</b> BoolExp <b>THEN</b>         # statements here     <b>ELSE</b>         # statements here     <b>ENDIF</b> <b>ENDIF</b>         </pre>	<pre> a ← 1 <b>IF</b> (a <b>MOD</b> 4) = 0 <b>THEN</b>     <b>OUTPUT</b> 'multiple of 4' <b>ELSE</b>     <b>IF</b> (a <b>MOD</b> 4) = 1 <b>THEN</b>         <b>OUTPUT</b> 'leaves a remainder of 1'     <b>ELSE</b>         <b>IF</b> (a <b>MOD</b> 4) = 2 <b>THEN</b>             <b>OUTPUT</b> 'leaves a remainder of 2'         <b>ELSE</b>             <b>OUTPUT</b> 'leaves a remainder of 3'         <b>ENDIF</b>     <b>ENDIF</b> <b>ENDIF</b>         </pre>

## Selection (continued)

<p>IF-THEN-ELSE IF ENDIF (removes the need for multiple indentation levels).</p>	<pre><b>IF</b> BoolExp <b>THEN</b>     # statements here <b>ELSE IF</b> BoolExp <b>THEN</b>     # statements here     # possibly more ELSE IFs <b>ELSE</b>     # statements here <b>ENDIF</b></pre>	<pre>a ← 1 <b>IF</b> (a MOD 4) = 0 <b>THEN</b>     <b>OUTPUT</b> 'multiple of 4' <b>ELSE IF</b> (a MOD 4) = 1 <b>THEN</b>     <b>OUTPUT</b> 'leaves a remainder of 1' <b>ELSE IF</b> (a MOD 4) = 2 <b>THEN</b>     <b>OUTPUT</b> 'leaves a remainder of 2' <b>ELSE</b>     <b>OUTPUT</b> 'leaves a remainder of 3' <b>ENDIF</b></pre>
--	---	---



## Arrays

Assignment	Identifier $\leftarrow$ [Exp, ... ,Exp]	primes $\leftarrow$ [2, 3, 5, 7, 11, 13]
Accessing an element	Identifier[IntExp]	primes[0] # evaluates to 2  (questions on exam papers will start indexing at 0 unless specifically stated otherwise)
Updating an element	Identifier[IntExp] $\leftarrow$ Exp	primes[5] $\leftarrow$ 17 # array is now [2, 3, 5, 7, 11, 17]
Accessing an element in a two-dimensional array	Identifier[IntExp][IntExp]	table $\leftarrow$ [[1, 2],[2, 4],[3, 6],[4, 8]]  table[3][1]  # evaluates to 8 as second element # (with index 1) of fourth array # (with index 3) in table is 8
Updating an element in a two-dimensional array	Identifier[IntExp][IntExp] $\leftarrow$ Exp	table[3][1] $\leftarrow$ 16  # table is now #[ [1, 2], # [2, 4], # [3, 6], # [4, 16] ]

## Arrays (continued)

<p>Array length</p>	<p><b>LEN</b>(Identifier)</p>	<p><b>LEN</b>(primes) # evaluates to 6 using example above</p> <p><b>LEN</b>(table) # evaluates to 4 using example above</p> <p><b>LEN</b>(table[0]) # evaluates to 2 using example above</p>
<p>FOR-IN-ENDFOR (repeat the statements the number of times that there are elements in an array)</p> <p>NOTE: array items <b>cannot</b> be modified using this method</p>	<p><b>FOR</b> Identifier <b>IN</b> array</p> <p>    # statements here</p> <p><b>ENDFOR</b></p>	<pre>primes ← [2, 3, 5, 7, 11, 13] total ← 0 <b>FOR</b> prime <b>IN</b> primes     total ← total + prime <b>ENDFOR</b> <b>OUTPUT</b> 'Sum of the values in primes is' <b>OUTPUT</b> total</pre>

## Records

Record declaration	<pre><b>RECORD</b> Record_identifier      field1 : &lt;data type&gt;     field2 : &lt;data type&gt;     ...  <b>ENDRECORD</b></pre>	<pre><b>RECORD</b> Car     make : String     model : String     reg : String     price : Real     noOfDoors : Integer  <b>ENDRECORD</b></pre>
Variable Instantiation	<pre>varName ← Record_identifier(value1, value2, ...)</pre>	<pre>myCar ← Car('Ford', 'Focus', 'DX17 GYT', 1399.99, 5)</pre>
Assigning a value to a field in a record	<pre>varName.field ← Exp</pre>	<pre>myCar.model ← 'Fiesta'  # The model field of the myCar # record is assigned the value # 'Fiesta'.</pre>
Accessing values of fields within records	<pre>varName.field</pre>	<pre><b>OUTPUT</b> myCar.model  # Will output the value stored in the # model field of the myCar record</pre>

## Subroutines

**Note:** for the purposes of this pseudo-code definition subroutines that contain a **RETURN** keyword are functions. Those that do not contain a **RETURN** keyword are procedures.

Subroutine definition	<pre> <b>SUBROUTINE</b> Identifier(parameters)      # statements here  <b>ENDSUBROUTINE</b>         </pre>	<pre> <b>SUBROUTINE</b> showAdd(a, b)     result ← a + b     <b>OUTPUT</b> result <b>ENDSUBROUTINE</b>  <b>SUBROUTINE</b> sayHi()     <b>OUTPUT</b> 'Hi' <b>ENDSUBROUTINE</b>  # Both of these subroutines are procedures         </pre>
Subroutine return value	<pre> <b>RETURN</b> Exp         </pre>	<pre> <b>SUBROUTINE</b> add(a, b)     result ← a + b     <b>RETURN</b> result <b>ENDSUBROUTINE</b>  # This subroutine is a function         </pre>
Calling subroutines	<pre> # Subroutines without a return value Identifier(parameters)  # Subroutines with a return value Identifier ← Identifier(parameters)         </pre>	<pre> showAdd(2, 3)  answer ← add(2, 3) * 6         </pre>

## String handling

String length	<b>LEN</b> (StringExp)	<b>LEN</b> ('computer science') # evaluates to 16(including space)
Position of a character	<b>POSITION</b> (StringExp, CharExp)	<b>POSITION</b> ('computer science', 'm') # evaluates to 2 (as with arrays # exam papers will start # indexing at 0 unless # specifically stated otherwise)
Substring (the substring is created by the first parameter indicating the start position within the string, the second parameter indicating the final position within the string and the third parameter being the string itself).	<b>SUBSTRING</b> (IntExp, IntExp, StringExp)	<b>SUBSTRING</b> (2, 9, 'computer science') # evaluates to 'mputer s'
Concatenation	StringExp + StringExp	'computer' + 'science' # evaluates to 'computerscience'

## String and character conversion

Converting string to integer	<b>STRING_TO_INT</b> (StringExp)	<b>STRING_TO_INT</b> ('16') # evaluates to the integer 16
Converting string to real	<b>STRING_TO_REAL</b> (StringExp)	<b>STRING_TO_REAL</b> ('16.3') # evaluates to the real 16.3
Converting integer to string	<b>INT_TO_STRING</b> (IntExp)	<b>INT_TO_STRING</b> (16) # evaluates to the string '16'
Converting real to string	<b>REAL_TO_STRING</b> (RealExp)	<b>REAL_TO_STRING</b> (16.3) # evaluates to the string '16.3'
Converting character to character code	<b>CHAR_TO_CODE</b> (CharExp)	<b>CHAR_TO_CODE</b> ('a') # evaluates to 97 using ASCII/Unicode
Converting character code to character	<b>CODE_TO_CHAR</b> (IntExp)	<b>CODE_TO_CHAR</b> (97) # evaluates to 'a' using ASCII/Unicode

## Input/output

User input	<b>USERINPUT</b>	$a \leftarrow \mathbf{USERINPUT}$
Output	<b>OUTPUT</b> StringExp, ... StringExp	<b>OUTPUT</b> a <b>OUTPUT</b> a, g  # The output statement can be followed by multiple StringExp separated by commas

## Random number generation

Random integer generation (between two integers inclusively).	Identifier $\leftarrow \mathbf{RANDOM\_INT}(\text{IntExp}, \text{IntExp})$	diceRoll $\leftarrow \mathbf{RANDOM\_INT}(1, 6)$  # will randomly generate an # integer between 1 and 6 # inclusive
---	--	---