Notes and guidance: Pseudo-code

The pseudo-code described below is provided to assist students preparing for their AQA GCSE Computer Science examination (8525).

In all assessment material, AQA will use a consistent style of pseudo-code as described and shown in this document. This will ensure that, given sufficient preparation, candidates will understand the syntax of the pseudo-code used in assessments easily. It is not the intention that candidates must use this style of pseudo-code in their own work or written assessments, although they are free to do so. The only direction to candidates when answering questions or describing algorithms written in pseudo-code is that their code is clear, consistent and unambiguous.

This document may be updated as required and the latest version will always be available on our website. Updates will not be made mid-year unless an error is discovered that must be corrected. If this happens centres will be notified of the changes. Ordinary updates will be made over the summer period with the new version for the following 12 months posted on our website at the start of the academic year, if any updates were made.

The document 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`

Constant assignment	**`CONSTANT`** `IDENTIFIER ← Exp`	**`CONSTANT`** `PI ← 3.141`
		`CONSTANT` `CLASS_SIZE ← 23`
		
		`# Names of constants will always be`
		`# written in capitals`

Arithmetic operations

Standard arithmetic operations	`+`
	`-`
	`*`
	`/`
		Used in the normal way with brackets to indicate precedence where needed. For example, `a + b * c` would multiply `b` and `c` together and then add the result to `a`, whereas `(a + b) * c` would add `a` and `b` together and then multiply the result by `c`.
		
		The `/` symbol is used instead of ÷ for division (for integer division use **`DIV`**)

Integer division	`IntExp` **`DIV`** `IntExp`	`9` **`DIV`** `5` evaluates to `1`
		`5` **`DIV`** `2` evaluates to `2`
		`8` **`DIV`** `4` evaluates to `2`

Modulus operator	`IntExp` **`MOD`** `IntExp`	`9` **`MOD`** `5` evaluates to `4`
		`5` **`MOD`** `2` evaluates to `1`
		`8` **`MOD`** `4` evaluates to `0`

Relational operators for types that can be clearly ordered

Less than	`Exp < Exp`	`4 < 6`
		`'A' < 'B'`
		`'adam' < 'adele'`

Greater than	`Exp > Exp`	`4.1 > 4.0`
Equal to	`Exp = Exp`	`3 = 3`
Not equal to	`Exp ≠ Exp`	`qty ≠ 7`
Less than or equal to	`Exp ≤ Exp`	`3 ≤ 4` `4 ≤ 4`
Greater than or equal to	`Exp ≥ Exp`	`4 ≥ 3` `4.5 ≥ 4.5`

Boolean operations

Logical AND	`BoolExp` **`AND`** `BoolExp`	`(3 = 3)` **`AND`** `(3 ≤ 4)`
Logical OR	`BoolExp` **`OR`** `BoolExp`	`(x < 1)` **`OR`** `(x > 9)`
Logical NOT	**`NOT`** `BoolExp`	**`NOT`** `(a < b)`

Indefinite (condition controlled) iteration

REPEAT-UNTIL (repeat the statements until the Boolean expression is True)	**REPEAT**
	    # statements here
	UNTIL BoolExp
		a ← 1
		REPEAT
		    OUTPUT a
		    a ← a + 1
		UNTIL a = 4
		# will output 1, 2, 3

WHILE-ENDWHILE (while the Boolean expression is True, repeat the statements)	**WHILE** BoolExp
	    # statements here
	ENDWHILE
		a ← 1
		WHILE a < 4
		    OUTPUT a
		    a ← a + 1
		ENDWHILE
		# will output 1, 2, 3

Definite (count controlled) iteration

FOR-TO-[STEP]-ENDFOR
(If **STEP** IntExp is missing it is considered to be 1.)
	FOR Identifier ← IntExp **TO** IntExp [**STEP** IntExp]
	
	  # statements here
	
	ENDFOR
	
	# If **STEP** IntExp is omitted the step value is 1.
		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-IN-ENDFOR
(repeat the statements the number of times that there are characters in a string)
	FOR Identifier **IN** StringExp
	
	  # statements here
	
	ENDFOR
		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

Selection

IF-THEN-ENDIF (execute the statements only if the Boolean expression is True)	`IF BoolExp THEN`
	`  # statements here`
	`ENDIF`
		`a ← 1`
		`IF (a MOD 2) = 0 THEN`
		`  OUTPUT 'even'`
		`ENDIF`

IF-THEN-ELSE-ENDIF (execute the statements following the THEN if the Boolean expression is True, otherwise execute the statements following the ELSE)	`IF BoolExp THEN`
	`  # statements here`
	`ELSE`
	`  # statements here`
	`ENDIF`
		`a ← 1`
		`IF (a MOD 2) = 0 THEN`
		`  OUTPUT 'even'`
		`ELSE`
		`  OUTPUT 'odd'`
		`ENDIF`

NESTED IF-THEN-ELSE ENDIF (use nested versions of the above to create more complex conditions)
 	
Note that IF statements can be nested inside the THEN part, the ELSE part of both
	`IF BoolExp THEN`
	`  # statements here`
	`ELSE`
	`  IF BoolExp THEN`
	`  # statements here`
	`  ELSE`
	`  # statements here`
	`  ENDIF`
	`ENDIF`
		`a ← 1`
		`IF (a MOD 4) = 0 THEN`
		`  OUTPUT 'multiple of 4'`
		`ELSE`
		`  IF (a MOD 4) = 1 THEN`
		`  OUTPUT 'leaves a remainder of 1'`
		`  ELSE`
		`  IF (a MOD 4) = 2 THEN`
		`  OUTPUT 'leaves a remainder of 2'`
		`  ELSE`
		`  OUTPUT 'leaves a remainder of 3'`
		`  ENDIF`
		`  ENDIF`
		`ENDIF`

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

Arrays

Assignment	`Identifier ← [Exp, … ,Exp]`	`primes ← [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] ← Exp`	`primes[5] ← 17`
		
		`# array is now [2, 3, 5, 7, 11, 17]`

Accessing an element in a two-dimensional array	`Identifier[IntExp][IntExp]`	`table ← [[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] ← Exp`	`table[3][1] ← 16`
		
		`# table is now`
		`#[ [1, 2],`
		`#  [2, 4],`
		`#  [3, 6],`
		`#  [4, 16] ]`

Array length	**LEN**(Identifier)	**LEN**(primes) # evaluates to 6 using example above **LEN**(table) # evaluates to 4 using example above **LEN**(table[0]) # evaluates to 2 using example above

Records

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

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	**SUBROUTINE** Identifier**(**parameters**)**
	
	  # statements here
	
	ENDSUBROUTINE
		SUBROUTINE showAdd(a, b)
		  result ← a + b
		  OUTPUT result
		ENDSUBROUTINE
		
		SUBROUTINE sayHi()
		  OUTPUT 'Hi'
		ENDSUBROUTINE
		
		# Both of these subroutines are procedures

Subroutine return value	**RETURN** Exp	**SUBROUTINE** add(a, b)
		  result ← a + b
		  RETURN result
		ENDSUBROUTINE
		
		# This subroutine is a function

Calling subroutines	# Subroutines without a return value Identifier**(**parameters**)** # Subroutines with a return value Identifier ← Identifier**(**parameters**)**	showAdd(2, 3) answer ← add(2, 3) * 6

String handling

String length	**LEN(**StringExp**)**	**LEN**('computer science') # evaluates to 16(including space)
Position of a character	**POSITION(**StringExp, CharExp**)**	**POSITION**('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).	**SUBSTRING(**IntExp, IntExp, StringExp**)**	**SUBSTRING**(2, 9, 'computer science') # evaluates to 'mputer s'
Concatenation	StringExp + StringExp	'computer' + 'science' # evaluates to 'computerscience'

String and Character Conversion

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

Input/output

User input	**USERINPUT**	a ← **USERINPUT**
Output	**OUTPUT** StringExp, … StringExp	**OUTPUT** a **OUTPUT** 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 ← **RANDOM_INT(**IntExp, IntExp)	diceRoll ← **RANDOM_INT**(1, 6) # will randomly generate an # integer between 1 and 6 # inclusive

Error detection

Detecting and responding to errors	**TRY**
	
	  # statements that may produce
	  # an error here
	
	CATCH
	
	  # statements to execute when an error
	  # is detected in the TRY section
	
	ENDTRY
		TRY
		
		  int_input ← **STRING_TO_INT**(string_input)
		
		CATCH
		
		  OUTPUT 'You must input an integer'
		
		ENDTRY