Pseudocode for National 5 Computing Science Question Papers

1 Overview

Being able to reason about code is increasingly being seen as a crucial part of learning to program. For example, if you can't explain in precise detail what a fragment of code does, you can't debug. If you can't explain the code you've just written to someone else, how can you justify any of the decisions you made in creating it, and then demonstrate any level of understanding?

To assess candidates' ability to reason about programs, programs must be presented in assessment questions. This document contains a specification for a pseudocode language designed for setting such questions, developed in collaboration with Prof. Greg Michaelson of Heriot Watt University and Dr. Quintin Cutts of the University of Glasgow. It is suitable for use in schools and FE/HE institutions. It enables examiners, assessors and candidates to work to one well-defined pseudocode notation.

The use of pseudocode supports SQA's decision to allow centres to use the programming language of their choice for instruction, as long as assessors ensure that candidates have mapped their understanding from the language of instruction across to the pseudocode. This focus on concepts that are shared between programming languages is potentially a major lever in deepening understanding of computation in general.

Although the idea of a clearly-defined pseudocode may seem daunting, it is not in fact so different from the pseudocode that has been used for years in SQA exam papers. It has simply been regularised, so that current and new assessors, exam-setters, and candidates will all be working to the same definition.

This document presents a reduced specification suitable for the SQA National 5 Course. Documents providing the full specification, and giving more extensive examples of use, will be available in due course.

In reviewing this specification, bear in mind its primary purpose:

- Where candidates may be instructed using one of a range of languages, a clearly-defined pseudocode should enable code **to be presented** to candidates under closed assessment conditions such that they can reason about it.
- Candidates are not expected to **write** code in the clearly-defined pseudocode, given that examiners should be able to mark solutions written in a range of

languages commonly used for teaching — and so candidates can use the language of their choice.

Note that assessors and candidates may choose to use this clearly-defined pseudocode as a tool to support program design, but this is **not its primary purpose**.

The aim in the rest of the document is to present the pseudocode principally via a small number of examples. In reading through the examples and specification, attachment to particular constructs, or to 'my favourite construct in language X', should be avoided — it is the **concepts** that are the major focus.

2 Introducing the pseudocode by example

The following typical programming examples show that solutions in our specified pseudocode do not differ markedly from those in any pseudocode notation.

The first example is for the problem:

> *Read in a number representing a temperature in degrees Celsius and write it out as a value in degrees Fahrenheit. If the Celsius value is C, then the Fahrenheit value, F, is calculated as follows: F = (9 / 5) * C + 32.*

Using our pseudocode, this would be written as follows:

```
RECEIVE c FROM KEYBOARD
SET f TO ( 9 / 5 ) * c + 32
SEND f TO DISPLAY
```

An immediate observation is that the keywords are written in CAPITALS. In any representation of programming language code, it is useful for the reader to distinguish easily between the language's keywords and other names created by the user. Using bold and underline is one technique. Capitalisation is another, and it has been chosen to facilitate keyword highlighting when writing examples on paper or on the board, where emboldening is not practical, and underlining can get messy.

Here's a slightly more complex problem:

> Read *in 10 numbers and write out the average of those numbers.*

This would be represented as follows:

```
SET total TO 0
SET count TO 0

WHILE count < 10 DO
        RECEIVE nextInput FROM KEYBOARD
        SET total TO total + nextInput
        SET count TO count + 1
END WHILE

SEND total / 10 TO DISPLAY
```

Here's a problem that uses an array:

Store and process the race times of the finalists in a 100m sprint so that the winner's time is output.

The solution would look like this:

```
SET allTimes TO [10.23, 10.1, 10.29, 9.9, 10.12, 10.34, 9.99, 9.58]
SET fastestTime TO allTimes [0]
FOR EACH time FROM allTimes DO
    IF fastestTime > time THEN
        SET fastestTime TO time
    END IF
END FOR EACH
SEND ["The winner's time was:" & fastestTime] TO DISPLAY
```

The only possibly slightly new aspect to this code is the FOR EACH iterator, which iterates over anything that is a collection of values, like an array. It is therefore a generalisation of the kind of FOR loop found in most languages, which can iterate over a sequence of integers only. Increasingly, modern programming languages have the FOR EACH style of iterator.

The final example shows how code can be presented in relation to a graphical environment with a library of graphical procedures/functions/subroutines.

We are working in a graphical context and already have an array of sprites (graphical objects) declared as follows, using some sprites we've already created:

SET sprites TO [frog, cow, kangaroo, rhinoceros]

The following subroutines are defined to work on sprites:

getColour: returns the colour of the sprite parameter as a string

move: moves the sprite in the direction and distance specified

Write code to move those objects in the sprites *array that are* red *up by a distance* 0.5*.*

The solution to this problem would be:

```
FOR EACH sprite FROM sprites DO
    IF getColour( sprite ) = "red" THEN
        move( sprite, "up", 0.5 )
    END IF
END FOR EACH
```

Note that in the problem specification, some of the detail is left out. For example, it is not clear exactly how the frog, cow, etc are created. But this shouldn't matter. It is expected that the candidates will have had experience of this kind of concept using the concrete languages with which they are learning to program. Hence the concept of graphical objects, and of subprograms that operate over them, shouldn't be new.

In summary, the purpose here is to show that solutions to problems presented using our clearly-defined pseudocode do not look radically different to other pseudocodes

used for assessment. The aim here is simply to ensure that everyone is using the same pseudocode.

The full specification appropriate for National 5, attached to this document as an appendix, may look lengthy, but that is what is required if any language is to be specified accurately — and is a testament to how much anyone learning a programming language has implicitly picked up, even if they can't articulate all the pieces!

Once again, remember that candidates, or more particularly, examinees, are never going to be expected to write this pseudocode, only to be able to read and understand it.

Appendix: The specification suitable for National 5

1. Types

Types are a major modelling tool for the development of programs, enabling the structure of the data manipulated to be clearly specified. The type system of a language typically contains both base types, such as integers and Booleans, and structured types, such as arrays and records.

The pseudocode language is typed — that is to say, all values in the language have a type associated with them — but types are not exposed if obvious from context.

The base types and their values are:

- INTEGER : -*big* ... + *big* – where *big* is arbitrary
- REAL : -*big*.*small* ... + *big*.*small* — where *big* and *small* are arbitrary
- BOOLEAN: true & false
- CHARACTER : '*character*'

At National 5, the structured types are:

- ARRAY : finite length sequence of same type
- STRING : ARRAY of CHARACTER

Note that STRING is really just a specialisation of ARRAY.

Finite length structured type values may be denoted explicitly as:

- [*value1*, value2, ...] for ARRAY
- "character character ..." for STRING

For example,

- [true, false, true, true] is an ARRAY holding four BOOLEANs
- "Hello, this is a message" is a STRING

2 System entities

System entities include:

- DISPLAY : in effect the default WINDOW or console out.
- KEYBOARD : in effect the default TEXTBOX or console in.

3 Identifiers

Identifiers are the usual sequences of letters and digits and "_", starting with a letter. Examples are:

- myValue My_Value counter2

4 Commands

Commands include:

- variable introduction and assignment
- command sequences
- conditions
- repetitions and iterations
- sub-program calls

4.1 Variable introduction and assignment

There are numerous ways in which variables can be modelled. Considerations are, at the least: do variables need to be explicitly introduced before they can be used; must they always have an initialising value; should the type of the variable be explicitly provided (assuming types are used at all!)?

Variables are introduced **implicitly** by first use on the left of an assignment, and the type of the variable is inferred from the initialising value:

- SET id TO value
  - Introduces *id* of same type as, and initialised to, *value*
  - Includes initialisation of structured types

While this may seem less rigorous than formally requiring a separate declaration, note that this pseudocode language is primarily to be used for the presentation of program fragments in exam questions, and details of variables can be outlined in the question preamble.

Examples are

- SET counter TO 0 creates a *counter* variable, initialised to zero
- SET a TO b creates variable *a*, initialised to the value held by variable *b*
- SET myVals TO [1, 2, 3] creates *myVals* initialised to an array

Assignment looks identical to variable introduction, but note the typing requirement.

- SET id TO expression
  - Change the value associated with *id* to that of *expression*.
  - The type of *expression* must match the type already associated with *id*.

4.2 Command sequences

The concept of a sequence of commands is one of the major control flow structures in any language. These are also known as ‘blocks’ in many languages.

In this pseudocode, commands one line after another are implicitly in top to bottom sequence. Command sequences are made explicit on one line with “;” as a separator, not a terminator.

The extent of a command sequence is implicitly defined, when it is the outermost level of a program, by the beginning and end of the program code; it is explicitly defined everywhere else, by the particular command structure containing it.

Where *command* appears in command definitions below, this stands for a single command or a command sequence.

4.3 Condition

Conditional commands have the form:

- IF expression THEN command END IF
- IF expression THEN command ELSE command END IF

An example of a simple one-armed conditional is:

```
IF a > 3 THEN
   SEND “more than three” TO DISPLAY
END IF
```

4.4 Repetition

Repetition may be specified to take place a fixed number of times, or it may continue until a condition is reached.

4.4.1 Unbounded/Conditional repetition

Conditional repetition can place the decision on whether to continue repeating at the start or end of the command sequence to be repeated. These commands are:

- WHILE *expression* DO *command* END WHILE
- REPEAT *command* UNTIL *expression*

4.4.2 Bounded/Fixed repetition

These take two forms. In the first, code is repeated a specified number of times:

- REPEAT *expression* TIMES *command* END REPEAT

Note that the ubiquitous FOR loop is technically an ***iterator***, the second form. The terms repetition and iteration are often used interchangeably. However, technically, one iterates *over something*. That is, we are using iteration when we examine/process items in a structured data value, one by one.

The FOR loop is the most familiar iterator — it effectively creates a list of integers from the lower to upper bounds specified, using a step if available, and then makes each element of that list available to the code body by placing it in the loop variable. The FOREACH loop is the more general iterator, operating over any structured type value.

In Haggis, iteration commands have the form:

- FOR *id* FROM *expr* TO *expr* DO *command* END FOR
- FOR *id* FROM *expr* TO *expr* STEP *expr* DO *command* END FOR
- FOR EACH *id* FROM *expression* DO *command* END FOR EACH
  - *expression* returns a structured value - an ARRAY or STRING
  - the order of value extraction from the structured value is first to last

As an example of the FOREACH construct:

```
SET myArray TO [ "The","sun","is","shining","today" ]
SET sentence TO ""

FOREACH word FROM myArray DO
        SET sentence TO sentence & word & " "
END FOREACH
```

4.5 Subprograms

National 5 requires only that candidates can *use* libraries of subprograms, with and without parameters. The subprograms can return values. It is expected that the specification of any subprograms used in questions will be specified in the question pre-amble. This enables a wide range of contexts to be used in question setting.

Subprograms may be called as:

- *id*(...) where … is a comma separated list of arguments, possibly empty

For example

```
SET t TO currentTimeIn( "New York" )
```

makes use of a subprogram *currentTimeIn*, taking a city as parameter, and returning the current time in that city.

5 Operations

The usual *infix and prefix* operations on INTEGER and REAL are provided:

- minus: - unary
- add: +
- subtract: -
- multiply: *
- divide: /
- exponent: ^

In addition, INTEGER has:

- modulo: mod

The *binary comparison operators* aim to model their mathematics counterparts:

- equality: =
- inequality: ≠
- less than: <
- less than or equal: <=
- greater than: >
- greater than or equal: >=

Comparisons apply to all finite types, where equality is defined to be element by element.

Order comparisons on structured types imply alphabetic order or equivalent.

The *logical operators* are:

- conjunction: AND
- disjunction: OR
- negation: NOT

Expressions are bracketed by (...).

STRINGS and ARRAYS may be concatenated using the & operator, and their length found using the standard subprogram *length*. For example:

```
SET myLength TO length( "Quintin" & "Cutts" )
```

Selecting items from structured types:

- Both ARRAY and STRING types may be accessed by:
  - *id*[*index*]
- Indexing starts from zero.

6 Elision

Since this is a pseudocode, and the specification of some parts of a program may be left for further refinement,

<text>

may be used instead of any command to express such an item.

7 I/O

For National 5, simple abstractions are given for screen and keyboard based I/O

To input next value from keyboard:

RECEIVE *id* FROM (*type*) KEYBOARD

For example

RECEIVE s1 FROM (STRING) KEYBOARD	takes all characters on the line
RECEIVE s2 FROM (INTEGER) KEYBOARD	jumps whitespace, reads an integer
RECEIVE s3 FROM (CHARACTER) KEYBOARD	reads a single character

To append output value to display:

SEND *expression* TO DISPLAY

For example

- SEND "\n" TO DISPLAY takes the output to a new line
- SEND 23 TO DISPLAY prints the integer 23
- SEND ["h ",1] TO DISPLAY prints out "h 1" without the quote marks