



Reference language for Higher Computing Science Question Papers

Note: This document replaces the document "Pseudocode for Higher Computing Science Question Papers", published in June 2013.

1 Overview

The ability 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 reference language designed for setting such questions, developed in collaboration with Prof. Greg Michaelson of Heriot Watt University, Dr. Quintin Cutts of the University of Glasgow, and Prof. Richard Connor of Strathclyde University. It is suitable for use in schools and FE/HE institutions. It enables assessors, teachers and candidates to work to one well-defined notation.

Note: in earlier versions of this document, this reference language was referred to as "pseudocode". Given that the language presented here is formally-defined, and pseudocode is not, the term "reference language" is preferred. A formally-defined language is required because it is a candidate's ability to understand and analyse code in such languages (including programming languages) that should be assessed.

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

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

This document presents a reduced specification suitable for the SQA Higher 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, this clearly-defined reference language 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 reference language, given that assessors 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 reference language as a tool to support program design, but this is **not its primary purpose**. The elision feature `<...>` presented in Section 6 enables the inclusion of steps that have not been fully worked out yet.

The aim in the rest of the document is to present the reference language 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 reference language by example

The following typical programming examples show that solutions in our specified reference language 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 reference language, this would be written as follows:

```
RECEIVE c FROM KEYBOARD
SET f TO ( 9.0 / 5.0 ) * 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 / 10.0 TO DISPLAY
```

Here's a problem that uses an array:

Store and process the race times of the finalists in a 100 m 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 reference language do not look radically different from other pseudocodes used for assessment. The aim here is simply to ensure that everyone is using the same reference language.

The full specification appropriate for Higher, 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 are never going to be expected to write this reference language, only to be able to **read** and **understand** it.

Appendix: Specification suitable for Higher

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 reference 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 : $-big \dots + big$ — where *big* is arbitrary
- ◆ REAL : $-big.small \dots + big.small$ — where *big* and *small* are arbitrary
- ◆ BOOLEAN : true & false
- ◆ CHARACTER : *'character'*

At Higher, the structured types are:

- ◆ ARRAY : finite length sequence of same type
- ◆ STRING : ARRAY of CHARACTER
- ◆ RECORD : collection of labelled, typed values

Note that STRING is really just a specialisation of ARRAY.

Finite length structured type values may be denoted explicitly as:

- ◆ $[value1, value2, \dots]$ for ARRAY
- ◆ *"character1 character2 ..."* for STRING

For example,

- ◆ $[true, false, true, true]$ is an ARRAY holding four BOOLEANs
- ◆ *"Hello, this is a message"* is a STRING
- ◆ $\{ name = "Fred", age = 42 \}$ is a RECORD with two fields name and age of types STRING and INTEGER respectively, and with values "Fred" and 42.

Record types can be named as shown in the following example for a record type that holds the same information as in the example above:

- ◆ RECORD Person IS { STRING name, INTEGER age }

and values can then be constructed using the new record name. For example

- ◆ Person("Fred", 42)

creates a value that is equivalent to the record example given above.

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. They should not include . or -. Examples are:

- ◆ myValue My_Value counter2

4 Commands

Commands include:

- ◆ variable introduction and assignment
- ◆ command sequences
- ◆ conditions
- ◆ repetitions and iterations
- ◆ subprogram calls

4.1 Variable introduction and assignment

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 reference language is primarily to be used for the presentation of program fragments in question paper 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

The two statements

- ◆ SET fred TO { name = "Fred", age = 42 }
- ◆ SET fred TO Person("Fred", 42)

where the second makes use of the named *Person* type from Section 1, each set up a variable *fred* containing equivalent record values.

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 reference language, 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 FOR EACH loop is the more general iterator, operating over any structured type value.

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 FOR EACH construct:

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

FOR EACH word FROM myArray DO
    SET sentence TO sentence & word & " "
END FOR EACH
```

4.5 Subprograms

Higher requires that candidates can define and use subprograms with and without parameters. Subprograms could be procedures, functions or methods. Candidates must be able to understand the concept of appropriate parameter passing, by value and by reference, and of both formal and actual parameters.

Procedure subprogram definitions have the form:

```
PROCEDURES id(...)
    command(s)
END PROCEDURE
```

Function subprogram definitions have the form:

```
FUNCTION id(...) RETURNS type
    command(s)
    RETURN expression
END FUNCTION
```

Subprograms may be called as:

- ◆ *id*(...)

In the above definitions ... is a comma separated list of arguments/parameters, possibly empty. Formal parameters may be optionally preceded by their types:

PROCEDURE *id*(*type*₁ *id*₁, *type*₂ *id*₂, ...)

or

FUNCTION *id*(*type*₁ *id*₁, *type*₂ *id*₂...) RETURNS *type*

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 for both ARRAY and STRING starts from zero, unless otherwise stated.

- ◆ Fields in record types may be accessed using dot notation as follows

- *id.fieldname*

where *fieldname* is one of the valid field names in the record type.

6 Elision

Since this reference language is primarily to be used for the presentation of code in exam contexts, there is a need to be able to avoid unnecessary detail in the code fragments shown to candidates. That is, the specification of some parts of a program may be left for further refinement, by using the following notation.

<text>

For example:

```
SET myArray TO <an array with a random collection of 10 numbers>
FOR EACH number FROM myArray DO
    <perform some action on number>
END FOR EACH
```

could be used as an example in a question that asked candidates to explain precisely how the FOR EACH construct worked. The code rigorously specifies the creation of a variable *myArray* and the framework of a FOR EACH loop iterating over *myArray*, but does not specify the precise detail of the array itself and the action to be performed on the elements of the array inside the loop body.

Note that this construct brings the flexibility of a pseudocode into the domain of a rigorously defined language. All code written *outside* instances of the *<...>* construct **must adhere absolutely** to the language definition. Only *inside* the *<...>* can English-like writing be used.

The *<...>* construct can be used instead of any command or expression.

7 I/O (including file operations)

To input next value from input device or file:

```
RECEIVE id FROM (type) source
```

- ◆ *source* may be a device (eg KEYBOARD) or a *file name represented as a STRING*
- ◆ (*type*) specifies the type of the value that will be read from *source*
- ◆ *file name* may include a file path or file URL, for example:
 - C:july/dataFile.txt
 - july/personal/scores.csv
 - file://july/dataFiles.txt
 - http://www.sqa.org/dataFile.txt

To append output value to an output device, file or stream:

```
SEND expression TO target
```

- ◆ *target* may be a device (eg DISPLAY) or a *file name, represented as a STRING*
- ◆ *target file name* may include a file path or file URL, as above

When source or target is an existing file, it must be opened and closed using:

```
OPEN file name  
CLOSE file name
```

When a target file does not exist, it can be created using:

```
CREATE file name
```

For example, the following is a complete sequence opening and reading from one file and then creating and writing to a second file, and finally closing both files.

```
SET sqaData TO "http://www.sqa.org/dataFile.txt"  
RECEIVE data FROM (STRING) sqaData
```

```
CREATE "outputFile.txt"  
SEND data TO "outputFile.txt"
```

```
CLOSE "outputFile.txt"  
CLOSE sqaData
```