Reference language for Higher Computing Science question papers

Note: This document replaces the documents "Pseudocode for Higher Computing Science Question Papers" published in June 2013 and “Reference language for Higher Computing Science question papers” published in August 2014.

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, Prof. 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 related documents, 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 of the reference language suitable for the SQA Higher Course. Versions of the specification for National 4/5 and for Advanced Higher are also available. Documents 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 taught 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 display it 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 (INTEGER) KEYBOARD
DECLARE f INITIALLY ( 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.

Here is a slightly more complex problem:

> *Read in 10 numbers and display the average of those numbers as a floating-point number.*

This would be represented as follows:

```
DECLARE total INITIALLY 0
DECLARE count INITIALLY 0

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

SEND total / 10.0 TO DISPLAY
```

Here is 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:

```
DECLARE allTimes INITIALLY [ 10.23, 10.1, 9.9, 10.34 ]
DECLARE fastestTime INITIALLY 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
```

Possibly, the only 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 have an array of sprites (graphical objects) we have already created, declared as follows:*
>
> ```
> DECLARE sprites INITIALLY [ frog, cow, kangaroo ]
> ```
>
> *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*.*

A 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, remembering that there is a formal definition that should be adhered to.

The reference language 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 it 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, all values in the language have a type associated with them — but types need not be 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 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, ... ]` 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 below 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.

The empty string is ""; the empty array is []; the empty record is {}

Types *may* be specified explicitly in variable declarations if necessary (see section 4.1)

Types *must* appear in the definitions of subprogram formal parameters and RECORD fields.

The type names are:

- `INTEGER`, `REAL`, `BOOLEAN`, `CHARACTER`, `STRING`
- `ARRAY OF` *`type`* where *`type`* can be any type name
- *`id`* where *`id`* is the name of a CLASS or RECORD

2 System entities

System entities include:

- `DISPLAY` : the default window or console out
- `KEYBOARD` : the default textbox or console in

3 Identifiers

Identifiers are the usual sequences of letters and digits and "_" starting with a letter. They cannot include . or -, and should not be all uppercase, to avoid confusion with reserved words. 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, or declared, **explicitly** and **must be initialised with a value**, using the syntax:

```
DECLARE id AS type INITIALLY value
```

or

```
DECLARE id INITIALLY value
```

The type of the variable ***need not be provided*** if it can be inferred from the initialising value; if this is not possible, then the type must be provided explicitly.

Examples are given below:

Note: in questions fragments of code may be used that omit variable declarations, as long as the nature of those variables is thoroughly described in the question preamble.

Base types

- `DECLARE counter INITIALLY 0`
  creates a ***counter*** variable, initialised to 0

- `DECLARE a INITIALLY b`
  creates variable ***a***, initialised to the value associated with variable ***b***

- `DECLARE x AS INTEGER INITIALLY 0`
  type not essential but given for clarity

Arrays

- `DECLARE someVals INITIALLY [ 1, 2, 3 ]`
  creates ***someVals*** initialised to an array

- `DECLARE myVals AS ARRAY OF INTEGER INITIALLY []`
  type must be given: it cannot be inferred from the initialising empty array

Note the use of the shorthand for creating large array values (see section 5), eg `[0] * 4` creates an array with four elements, all set to zero.

Assignment

Variables are updated using the following assignment statement:

- `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*

Examples are:

- `SET counter TO counter + 1`
  increments ***counter*** variable

- `SET a TO b`
  assigns variable ***a*** to the value held by variable ***b***

♦ `SET myVals TO [ 1, 2 ]`
assigns ***myVals*** to a new array value

Records
The two statements:

♦ `DECLARE fred INITIALLY { name = "Fred", age = 42 }`
♦ `DECLARE fred INITIALLY 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.

For compatibility with code written for earlier versions of the reference language, if a SET command is found without a prior DECLARE, then the SET implicitly declares the variable in the local scope level.

Scoping is fully discussed in section 4.5 on subprograms.

4.2 Precise meaning of assignment depending on type

When assigning a variable to a value, a *reference* to the value is used if the value contains embedded updateable values — that is, strings, arrays, records and objects. Consider the following code:

```
DECLARE first INITIALLY [ 10, 11, 12 ]      # an array
DECLARE second INITIALLY first   # still just 1 array
SET first[ 0 ] TO 20
SEND second[ 0 ] TO DISPLAY # update to first is seen
```

The output from this code is 20, because, since array values do contain updateable values, **first** contains a *reference* to the array value created in line 1, not a copy of it. **Second** is then initialised with the reference contained in **first**. Updates to the contents of the array via **first** are seen from **second**, since both variables refer to the same array value. On the other hand, consider the following code:

```
DECLARE first INITIALLY 3
DECLARE second INITIALLY first
SET first TO 2
SEND second TO DISPLAY
```

The output from this code is 3, because the INTEGER type does not contain updateable values, and so the second line effectively causes a copy to be made of the integer 3 in **first**, which is then associated with **second**. The update to **first** in the third line therefore has no effect on **second**.

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 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

The decision on whether to continue repeating can be placed 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`

The second form is the ***iterator***, of which the ubiquitous FOR loop is technically one example. 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

Note that *id* does not need to be declared explicitly, as its type and initial value can be inferred from the FOR statement in which it first appears.

As an example of the FOR EACH construct:

```
DECLARE myArray INITIALLY [ "The","sun","is","shining" ]
DECLARE sentence INITIALLY ""

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 parameter passing with formal and actual parameters.

Procedure subprogram definitions have the form:

```
PROCEDURE id(…)
      command(s)
END PROCEDURE
```

Function subprogram definitions have the form:

```
FUNCTION id(…) RETURNS type
      command(s)
      RETURN expression   (see note below)
END FUNCTION
```

Note: `RETURN` *`expression`* may be used anywhere inside the function, one or more times, and when executed causes:

- the expression to be evaluated
- the execution of the function to be terminated
- the result of the expression to be returned as the result of calling the function

Subprogram calls

Subprograms may be called as:

- *id*(…)

In the above definitions … is a comma separated list of arguments/parameters, possibly empty. Formal parameters are 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*

The same rules for assigning values from one variable to another (described in section 4.1), apply also to formal parameters being assigned to actual parameter values. Hence, if an array is passed into a subprogram, it is actually a reference to the array value that is passed. Changes to the content of the array inside the subprogram will be seen in the calling context once the subprogram call has completed.

4.5.1 Scope

The scoping rules are as follows:

1. The reference language uses block-level scoping. On entering each block, a new scope level is created, which is destroyed on exit from the block.
2. A variable is said to be *in scope*, that is, available for use in the program, from the DECLARE statement that introduces it to the end of the block containing that declaration.
3. A single scope level may only contain one declaration for a given variable name; but that same name may be declared in many scope levels.
4. On encountering the use of a variable in an expression, the sequence of nested scopes, from the scope immediately containing the variable use outward, is scanned for the first declaration of this variable — and it is this variable instance that is used in the expression.
5. As noted in section 4.1, whenever a SET operation is encountered without any corresponding DECLARE in the immediate or any outer scope level, it is assumed that this is a declaration in the immediate scope level. This is included for backwards compatibility with earlier versions of the reference language.

The following example shows the scope rules in practice:

```
DECLARE a INITIALLY 0              # 1
DECLARE b INITIALLY 0              # 2

PROCEDURE myProc( INTEGER b )      # 3
     SET a TO 3                    # 4
     SET b TO 4                    # 5
     SET c TO 5                    # 6
     DECLARE d INITIALLY 6         # 7
END PROCEDURE                      # 8

SET b to 7                         # 9
SET d to 8                         # 10
```

with the following explanations of the comments:

#1 Declares a variable `a` in the outermost scope level with initialising value `0`. This is often referred to as a global variable.

#2 Similarly for variable `b`.

#3 Declares the formal parameter `b` in `myProc`'s scope level. This will be initialised with the actual parameter value supplied when the procedure is called.

#4 The variable `a` in the outer scope level is updated.

#5 The formal parameter `b` at the inner scope level is updated.

#6 No `c` variable currently exists, either in the outer or inner scope levels, so a `c` variable is declared implicitly in the inner scope level, initialised to `5`.

#7 Variable `d` is declared in the inner scope level, initialised to `6`.

#8 At the end of the procedure, the variables `c` and `d`, and the formal parameter `b`, go out of scope.

#9 The value associated with the outer scope variable `b` is updated to `7`.

#10 There is no variable `d` declared in the outer scope, and the `d` variable declared inside the procedure is no longer in scope, and so a variable `d` is implicitly declared in the outer scope level, initialised to `8`.

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" )
```

For convenience, the * operator can be used in place of repeated concatenation. For example:

```
DECLARE myArray AS ARRAY OF INTEGER INITIALLY [ 0 ] * 20
```

creates a new array variable, myArray, and assigns to it a new array value containing 20 elements all set to zero. The expression after INITIALLY is shorthand for the expression `[0]&[0]&[0]&[0]& … &[0]&[0]`, with twenty `[0]`s and nineteen `&`s.

Items are selected from structured types as follows:

- 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.

Indexing outside the bounds of an array or string value is an error.

6 Comments and elisions

Both comments and elisions enable natural language to be mixed with the formal reference language, in a well-defined manner.

6.1 Comments

These operate in the same way as in most other languages, with an initial comment character, in this case #, followed by text up to the end of the line. Comments are used to clarify, query or explain to a human reader the purpose of nearby constructs. The comment character and text do not form part of the computation being described. For example:

```
# Declarations for the program come next
DECLARE myAge INITIALLY 21     # Can 21 really be true?
```

Typographically, the comment text is presented in non-fixed-width font, to give it the appearance of natural language, compared to formal program text which is always presented in Courier fixed-width font. Where a programming language entity is referred to from within a comment, it is presented in fixed-width font — for example, see the use of `21` in the second comment above.

6.2 Elisions

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.

Typographically, the elision text is presented in the same way as comments.

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 Input/output (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.

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

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

CLOSE "outputFile.txt"
CLOSE sqaData
```