Reference language for Computing Science question papers

This document introduces the reference language used to present code in SQA Computing Science question papers for National 5, Higher and Advanced Higher qualifications. Elements of the language required for Higher and Advanced Higher only are indicated using margin highlights.

This edition: September 2016, version 1.0

Published by the Scottish Qualifications Authority
The Optima Building, 58 Robertson Street, Glasgow G2 8DQ
Lowden, 24 Wester Shawfair, Dalkeith, Midlothian EH22 1FD

www.sqa.org.uk

Contents

1 Purpose of the reference language

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 enables assessors, teachers and candidates to work to one well-defined notation and is suitable for use in schools and further education/higher education institutions.

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 is 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.

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

- Although candidates may be taught using one of a range of languages, this clearly-defined reference language enables code to be presented in a way that candidates can reason about it under closed assessment conditions.
- Candidates are **not** expected to write code in the reference language. Assessors should be able to mark solutions written in a range of languages commonly used for teaching, 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 3, enables the inclusion of steps that have not been fully worked out yet.

The aim of the rest of this 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 language by example

The following typical programming examples show that solutions in the reference language do not differ markedly from those in any programming language.

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 the reference language, the solution would be written as follows:

```
DECLARE c AS INTEGER INITIALLY FROM 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. Readability has been a primary goal in designing the language.

Here is a slightly more complex problem:

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

Using the reference language, the solution would be written as follows:

```
DECLARE total INITIALLY 0
DECLARE count INITIALLY 0
DECLARE nextInput INITIALLY 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 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.*

Using the reference language, the solution would be written as follows:

```
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*.*

Using the reference language, the solution would be written as follows:

```
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 above solution, some of the detail is left out. For example, it is not clear exactly how the frog, cow, and kangaroo are created, but this shouldn't matter. It is expected that candidates will have had experience of this kind of concept using the concrete languages with which they are learning to program. So 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 the clearly-defined reference language do not look radically different from other pseudocodes used for assessment. The aim here is simply to ensure that all parties, particularly exam setters and candidates, are using the same reference language, remembering that there is a formal definition that should be adhered to.

The full specification defined in the following sections may look lengthy, but that is what is required if any language is to be specified accurately. 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!

Remember, candidates are never going to be expected to write this reference language, only to be able to read and understand it.

3 Specification

3.1 Types

Types is 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*'

The structured types are:

- ARRAY : finite length sequence of same type
- STRING : ARRAY OF CHARACTER
- RECORD : collection of labelled, typed values
- CLASS : used in OO programming (see se5ction 3.8)

Note that STRING is really just a specialisation of ARRAY.
A "2-D" array is an ARRAY OF ARRAYs (see section 3.4)

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
- `[ [ 1,2,3,4 ], [ 5,6,7,8 ], [ 9,10,11,12 ] ]`
  is an ARRAY of ARRAY OF INTEGER, which might be described as a “2-D array” of 3 “rows” and 4 “columns”
- `"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:

H

- `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 3.1)

H

Types must appear in the definitions of subprogram formal parameters and RECORD and CLASS fields. The type names are:

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

3.2 System entities

System entities include:

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

3.3 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`

3.4 Commands

Commands include:

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

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.

Note that, 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. Examples are given below:

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

- `DECLARE maze AS ARRAY OF ARRAY OF INTEGER INITIALLY []`
  introduces an empty “2-D array” of integers

AH

To initialise the “2-D” array above with (say) 9 “rows” and 4 “columns” of zeroes would require the following code:

```
SET maze TO [ [] ] * 9    # array with 9 elements,
                          # each an empty array
FOR count FROM 0 TO 8 DO
      # update element to be a 4-element array of zeros
      SET maze[ count ] TO [0] * 4
END FOR
```

Note the use of the shorthand for creating large array values (see section 3.5), eg `[0] * 4` creates an array with four elements, all set to zero.

However, in a question, it would be acceptable to use any of the following or similar alternatives:

- `DECLARE maze AS ARRAY OF ARRAY OF INTEGER INITIALLY` *<9 x 4 array, all set to zero>*
- `DECLARE maze AS ARRAY OF ARRAY OF INTEGER INITIALLY` *<9-element array, each containing a 4-element array, all set to zero>*

AH

or, describe the array in the question preamble, for example 'Assume a 2D array named maze which has 9 columns and 4 rows, with all elements set to zero.' and then have:

- `DECLARE maze AS ARRAY OF ARRAY OF INTEGER INITIALLY` *<as described above>*

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 )`

H

where the second makes use of the named *Person* type from section 3.1, each set up a variable *fred* containing equivalent record values.

Scoping is fully discussed in section 3.4 on subprograms.

Meaning of assignment

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*.

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 appearing one line after another are implicitly in top to bottom sequence. Command sequences are made explicit on a single 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.

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
```

Repetition

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

a) 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`*

b) 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, iteration is being used when examining/processing 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
```

Subprograms

H

Higher requires that candidates can define and use subprograms with and without parameters. Subprograms could be procedures, functions, and may appear in class definitions (see section 3.8). Candidates must be able to understand the concept of parameter passing with formal and actual parameters.

a) Procedure subprogram definitions have the form:

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

b) 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

c) Subprogram calls

Subprograms may be called as:

- `id(…)`

d) Parameters

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

```
PROCEDURE id( type1 id1, type2 id2, … )
  or
FUNCTION id( type1 id1, type2 id2, … ) RETURNS type
```

The same rules for assigning values from one variable to another (described in section 3.4) apply also to formal parameters being assigned to actual parameter values. So, 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. This is *call by value*; *call by reference* is not supported.

Scope

The language is statically scoped and the scoping rules are as follows:

- 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.
- A variable is *in scope*, that is, available for use, from the DECLARE statement introducing it to the end of the block containing that declaration.
- 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.
- 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 most recent declaration of this variable — and it is this variable instance that is used in the expression.

The following example shows the scope rules in practice:

```
DECLARE a INITIALLY 0              # 1
DECLARE b INITIALLY 0              # 2
DECLARE c INITIALLY 0              # 3

PROCEDURE myProc( INTEGER b )      # 4
     DECLARE c INITIALLY 5         # 5
     DECLARE d INITIALLY 6         # 6
     SET a TO 3                    # 7
     SET b TO 4                    # 8
     SET c TO 5                    # 9
END PROCEDURE                      # 10

SET b TO 7                         # 11
SET d TO 8                         # 12 (error)
```

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 Similarly for variable `c`.
#4 Declares the formal parameter `b` in `myProc`'s scope level. This will be initialised with the actual parameter value supplied on calls to the procedure.
#5 Variable `c` is declared in the inner scope level, initialised to `5`.
#6 Variable `d` is declared in the inner scope level, initialised to `6`.
#7 The variable `a` in the outer scope level is updated.
#8 The formal parameter `b` at the inner scope level is updated.
#9 The local variable `c` at the inner scope level is updated.
#10 At the end of the procedure, the variables `c` and `d`, and the formal parameter `b`, go out of scope.
#11 The value associated with the outer scope variable `b` is updated to `7`.
#12 There is no variable `d` in the outer scope, and the `d` declared inside the procedure is no longer in scope, and so this line is an error.

3.5 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`

Division, /, is integer division if both arguments are of type INTEGER.

The ***comparison operators*** aim to model their mathematical counterparts:

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

The ***logical operators*** are:

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

Expressions may be bracketed by (...).

The precedence rules are as follows:

Unary minus
^
*, /, MOD
+, -
comparison operators
NOT
AND
OR

Where operators are of the same precedence, they are evaluated left-to-right.

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" )
```

If one of the arguments used with the `&` operator is of type string, then the other argument will be coerced to a string. For example:

```
SEND "Number " & 3 TO DISPLAY
```

will result in `Number 3` being output.

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.

3.6 Comments and elisions

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

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.

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.

3.7 Input/output (including file operations)

Values can be read in from input devices and files and stored in variables, array elements, and record / class fields. For example, reading an integer from the keyboard into an integer variable `myInt` that has already been declared:

```
RECEIVE myInt FROM KEYBOARD
```

Similarly for array elements and record fields:

```
RECEIVE myArrayOfNumbers[ 4 ] FROM KEYBOARD
RECEIVE myPerson.name FROM KEYBOARD
```

Currently, `KEYBOARD` is the only defined input device. If writing example code using another kind of device, eg a sensor, then elision should be used, for example:

```
<sensor device>
```

The type of the value read from the input device/file is inferred from the variable / array element / record field type.

H

A file can be specified using a file name, path, or URL, represented as a `STRING`. Examples are:

- `"myNumbers.txt"`
- `"C:july/dataFile.txt"`
- `"july/personal/scores.csv"`
- `"file://july/dataFiles.txt"`
- `"http://www.sqa.org/dataFile.txt"`

For convenience, declaring a variable and initialising it to a value from an input device or file can be combined in a single `DECLARE` statement. For example:

```
DECLARE userInput AS STRING INITIALLY FROM KEYBOARD
```

Output values can be appended to an output device or file. For example, sending an integer, variable or the result of an expression to the screen:

```
SEND 4 TO DISPLAY
SEND myVariable TO DISPLAY
SEND "The new value is " & newValue TO DISPLAY
```

Currently, `DISPLAY` is the only defined output device. As for input, if another kind of device is to be used in example code, eg a controller, then elision should be used, for example:

```
<controller>
```

H

♦ A file can be specified in the same way as for input, as above.

When working with files that have already been created, they must be opened and closed, for example:

```
OPEN "myNumbers.txt"
CLOSE "myNumbers.txt"
```

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

```
CREATE "myUpdatedNumbers.txt"
```

H

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

```
DECLARE sqaData INITIALLY "http://www.sqa.org/dataFile.txt"
OPEN sqaData
DECLARE data AS STRING INITIALLY FROM sqaData

CREATE "outputFile.txt"
SEND data TO "outputFile.txt"
RECEIVE data FROM sqaData
SEND data TO "outputFile.txt"

CLOSE "outputFile.txt"
CLOSE sqaData
```

3.8 Object-oriented (OO) programming

H

Records

Records provide the ability to aggregate name:value pairs (fields) into a single value accessible for read and update using a dot notation.

```
RECORD Person IS { STRING name, INTEGER age }

DECLARE me INITIALLY Person( "Quintin", 47 )
DECLARE myAge INITIALLY me.age
SET me.age TO myAge + 1
```

AH

Classes, objects, instance variables and methods

As the scale of programs increases, mechanisms are required to partition a program so as to limit the extent by which one section of the program can manipulate data in another section of the program. Using abstract data types, or object orientation, the program is partitioned according to aggregations of related data (like records) and the associated operations over those aggregations. The partitioned data can only be accessed directly by the associated operations. This is encapsulation.

When viewing an object as an extension to a record instance, access to the data items in the record instance is restricted to a defined set of operations, or methods — these are just subprograms (procedures and functions) as outlined above. Code elsewhere in the program can only access the data items in an object via the set of operations defined for that object, and not directly, as in the case of a local variable or a record field.

Just as the type/structure of a record requires a definition in the program, so the type structure of an object, in terms of both the data items and the valid operations, needs a definition — a *class* definition.

In a *record* definition, the data items are referred to as fields. Many names are used for the data items in a class definition, such as attribute, property and instance variable. Instance variable is used, as it is the most general — values for each of the data items exist in an *instance* of the class, and since the data items have a name and are updatable, like a *variable*, the name instance variable makes sense.

In OO languages generally, instance variables can optionally be hidden from external view, only being accessible to the defined operations. This allows aspects of the underlying implementation of the class to be hidden, a key aspect of large system engineering.

In this reference language, all instance variables are inaccessible/invisible outside the methods that are defined within the class. This simplifies the language, removing the need for access modifiers such as Java's *private*, *protected* and *public* keywords.

A class can be defined as follows:

```
CLASS Person IS { STRING name, INTEGER age }

METHODS

    PROCEDURE introduce()
    # Note the use of THIS to access the object on
    # which this procedure has been invoked.
        SEND "Hello, my name is " & THIS.name TO DISPLAY
    END PROCEDURE

    FUNCTION getAge() RETURNS INTEGER
        RETURN THIS.age
    END FUNCTION

END CLASS
```

AH

Note the consistency with records in the form of procedures or functions.
A suitable model for a learner is to consider an object to be a record with associated functionality.

Also, as is typical in OO languages, the predefined name THIS is used to access the object on which a method has been invoked.

From a minimalist point of view (although see below for extensions), we do not require a constructor function explicitly. Instead, the class name can be used as with records:

```
DECLARE quintin INITIALLY Person( "Quintin", 47 )
```

and a method is selected in just the same way as a record field, and then invoked just as any subprogram would be:

```
quintin.introduce()
```

You **cannot** write the following lines because of the encapsulation restricting access to the data elements:

```
DECLARE qAge INITIALLY quintin.age
SET quintin.age TO 48
```

The same instance variable and method names may be used across different classes, as their scope is restricted to the class in which they are defined only. Method *overloading*, where many methods within a single class definition have the same name but with different parameter lists, is not permitted.

Inheritance

A class can be extended with additional data elements and behaviour. This is reflected in the syntax as shown below:

```
CLASS Employee INHERITS Person WITH { INTEGER empID }

METHODS

    FUNCTION getID() RETURNS INTEGER
        RETURN THIS.empID
    END FUNCTION

END CLASS
```

AH

All instance variables and methods in the superclass are accessible to the code in any new subclass methods, as well as newly-added instance variables and methods.

A value of a subclass can be created by using the subclass name and all the data elements of the superclass, followed by the data elements of the subclass. For example:

```
SET aWorker TO Employee( "Fred", 18, 1401234 )
SEND "This employee's ID is " & aWorker.getID() TO
DISPLAY
```

A superclass variable may be assigned to a value of a subclass. For example:

```
DECLARE aPerson INITIALLY Person( "Harry", 22 )
SET aPerson TO aWorker
```

This is an example of polymorphism, since the superclass variable may be associated with objects of many different subclasses, although only the methods associated with the superclass may be applied to those values. This is particularly useful when working over a collection of values of different subclasses, but with a common superclass, as in the following:

```
DECLARE myPeople AS ARRAY OF Person INITIALLY
                                [ quintin, aWorker ]

FOR EACH guy FROM myPeople DO
    guy.introduce()
END FOR EACH
```

Without the polymorphism supported by inheritance, we would not be able to create an array with values of two different types (given that in the reference language, all values in an array must be of the same type).

Overriding a method in a subclass

A method may be declared in a subclass with the same name as a method already existing in a superclass. The new method is said to *override* the superclass method. Such overriding is noted explicitly as follows:

```
CLASS Student INHERITS Person WITH
                                { ARRAY OF STRING courses }

METHODS

    OVERRIDE PROCEDURE introduce()
                SEND "Hi, I am a student, my name is " &
                     THIS.name TO DISPLAY
             END PROCEDURE

END CLASS
```

Constructor functions

AH

Constructor functions enable the initial values of instance variables to be set, without exposing the particular implementation of those instance variables and/or having to explicitly provide an initial value for every variable. For example:

```
CLASS Student INHERITS Person WITH
                                { ARRAY OF STRING courses }

METHODS

    CONSTRUCTOR ( STRING name, INTEGER age )
        DECLARE THIS.name INITIALLY name
        DECLARE THIS.age INITIALLY age
        DECLARE THIS.courses INITIALLY []
    END CONSTRUCTOR

END CLASS
```

In this example, a student can now only be created via the constructor function, which hides the original implicit constructor:

```
DECLARE aStudent INITIALLY Student( "Hazel", 18 )
```

Only one constructor may appear in a class definition. If a constructor is included then it must contain declarations for all the class's instance variables, including those of all superclasses.

4 Further resources

A checker run-time system and full formal specification of the language are available at

http://haggisforsqa.appspot.com/haggisparser.html?variant=higher