6.2 Statements

Description

A statement describes a computation. Execution of the statement implies that the computation is performed by the computer. Such a computation may be a simple operation or consists of a sequence of operations. We have the following kind of statements: assignment statements, conditional statements, loop statements, break statements, method invocations, and object instantiation.

Conditional statements, loop statements, and break statements are often referred to as *control structures* since they control the order in which the statements of a program are executed based on the state of the objects – this is often referred to as the *control flow* of the program.

Some of the statements are primitives of the qBeta language and others are defined in the qBeta library. We return to this is in the last section on this chapter.

The assignment statement

In the previous chapters, the *assignment statement* has been used to describe operations carried out by methods like `addInterest`, `deposit` and `withdraw`. The assignment statement has the form:

```
VariableDataItem := Expression
```

The left side of the statement must be the name of a variable data-item such as `balance` defined within the various descriptions of accounts. A variable data-item is declared using **`var`** or **`ref`**.

The right-side of the statement is an *expression* which describes the computation of a datum. An expression may be an arithmetic expression like `balance - amount`, as described in the previous section. This expression evaluates to a primitive value of type float. The statement below assigns this value to the variable `balance`:

```
balance := balance + amount
```

Conditional statements

Sometimes the computations of a method depends on the values of data-items of the object or other conditions. One example is `withdraw` where you may not be able to withdraw an amount that is greater than the balance. We may describe this using a conditional statement like an `if:then` statement or an `if:then:else`-statement.

The if:then statement

An `if:then` statement describes a conditional execution of a sequence of items. An example of an `if:then` statement is shown in section . It has the form:

```
if (condition) :then
   thenPart
```

The *condition* is a boolean expression that evaluates to ether true or false. The *thenPart* clause consists of a sequence of items and as mentioned, an item may be a declaration or a statement. The *thenPart* is only executed if the *condition* evaluates to true.

The if:then:else statement

An `if:then:else` statement describes a conditional execution selecting between two sequences of items and it has the form:

```
if (condition) :then
   thenPart
:else
   elsePart
```

As for the `if:then` statement, the `condition` must be an expression that evaluates to the Boolean value true or false. The clauses *`thenPart`* and *`elsePart`* are both a sequence of items. If the condition evaluates to true, then the items describe by *`thenPart`* are executed. Otherwise the items described by *`elsePart`* are executed.

An example of an `if:then:else` statement is given in section where the body of `withdraw` has an `if:then:else` statement:

```
class Account(owner: ref Customer):
   balance: var float;
   ...
   withdraw(amount: var float) -> newB: var
 float:
      if (amount <= balance) :then
         balance := balance - amount
      :else
         console.print("The balance is less than the amount")
      newB := balance
```

In the example, the `condition` is `amount <= balance` where <= means less-than-or-equal. This means that `amount <= balance` evaluates to true if `amount` is less-than-or-equal to `balance`. Otherwise it evaluates to false.

In the example, *`thenPart`* is an assignment statement as in the previous example of `withdraw`. The *`elsePart`* is a statement that prints the `String` `"The balance is less than the amount"` on a console.

We assume that our system has an object `console` that represents a window on the screen of the computer executing the bank system. We also assume that the `console` object has a `print` method that takes a `String` as an argument and prints it in the window.

Loop statements

A loop statement executes a list of statements forever or until some condition is met or interrupted by a break statement. We have three forms of loop statements `cycle`, `while`-loop and `for`-loop:

The cycle statement

A `cycle` statement has the form:

```
cycle
   items
```

where *`Statements`* is a sequence of items. Execution of a `cycle` statement implies that *`items`* are executed forever unless interrupted by a possible `break` statement (see below) in *`Items`*.

In our bank system we assume that interest is added to each account on the first day of every month. To do this, the following cycle statement may be executed by some agent in the bank system:

```
cycle
   if (today.isFirstDayOfMonth) :then
      JohnSmithsAccount.addInterest
      ...
```

The above example is quite tentative. We have not specified what an agent might be – an agent may be a parallel process

running in the bank system – we describe parallel processes in section . Also the condition `today.isFirstDayOfMonth` is left unspecified. Some more immediately useful examples of `cycle` will be shown in the following.

The while-loop

A `while` statements repeatedly executes a list of statements as long as a given condition evaluates to true. It has the form:

```
while (BooleanExp) :repeat
   Items
```

where *`BooleanExp`* is a boolean expression and *`Items`* is a list of statements.

In section on the data type `String`, we have seen an example of a while-loop:

```
while (aName.get[i] <> ' ') :repeat
    console.print(aName.get[i])
    i := i + 1
```

The for-loop

A `for` loop executes a list of statements a specific number of times. It has the form:

```
for (StartValue):to(EndValue):repeat
   Items
```

where *`StartValue`* and `EndValue` are expressions that evaluate to integer values, and *`Items`* is a list of items. If *`StartValue`* evaluates to `n1` and *`EndValue`* evaluates to `n2`, then the *`Items`* are evaluated `(n2 - n1 + 1)` times.

Within *`Items`*, the integer variable `inx` has the value of the current iteration number. That is, `inx` goes through the values `n1`, `n1 + 1`, `n1 + 1`, ... , `n2`.

In section , the following example makes use of a for loop:

```
balanceSum -> bal: var float:
   for (1):to(noOfAccounts):repeat
       bal := bal + accounts.get[inx].balance
```

If `noOfAccounts` evaluates to the value 3, the statement in the for-loop is executed 3 times and `inx` goes through the values 1, 2, and 3. The effect is that the following statements are executed:

```
bal := bal + accounts.get[1].balance
bal := bal + accounts.get[2].balance
bal := bal + accounts.get[3].balance
```

Break statements

A *break-statement* transfers the control-flow to another point in the program and is also called a *jump-statement*. The flow of control may be to the beginning or end of the current method object or the beginning or end of an enclosing method object or object.

We have two kinds of break-statements `leave` and `restart`.

The leave statement

The *leave* statement has the form:

```
leave (AttributeName)
```

where *`AttributeName`* is the name of an enclosing class, method or singular object.

If `L` is the name of an enclosing attribute, then `leave(L)` will transfer control to the end of the statement-part of `L`.

As an example, we may make a more safe version of the method for printing the first name in a given string by testing that there is actually a blank character in the `String` and if not leave the while-loop:

```
L:
   while (aName.get[i] <> ' ') :repeat
       console.print(aName.get[i])
       i := i + 1
       if (i > aName.length) :then
          console.print("No blank char in String")
          leave(L)
```

Here we have given the while-loop the name `L` id we get at the end of `aName` without meeting blank, `leave(L)` will terminate the while-loop.

The restart statement

The *restart* statement has the form:

`restart (`*`AttributeName`*`)`

where *`AttributeName`* is the name of an enclosing class, method or singular object.

If `L` is the name of an enclosing attribute, then `restart(L)` will transfer control to the beginning of the statement-part of `L`.

We may use `restart` to write another variant of the first name in the `String aName`:

```
firstName:
   i: var integer
   i := i + 1
   if (i <= aName.length) :then
      if (aName.get[i] <> ' ') :then
         console.print(aName.get[i])
         restart(firstName)
   :else
      console.print("No space in String")
```

The inner-most `if:then`-statement prints the next character in a possible first name and then restart execution from the beginning of `firstName`. All local data-items of `firstName` – in this case the variable `i`, keeps its values, so `i` is incremented in each iteration.

Method invocation

As mentioned in the previous section, a *method invocation* may be part of an expression. It may, however, also just appear as a statement in case no value is returned by the invocation or if the value is not used:

```
  account_1010.addInterest
```

Here `account_1010.addInterest` is an example of a method invocation being used as a statement.

For further descriptions of object instantiation and method invocation, see chapter and chapter .

Object instantiation

An object instantiation may in principle also be used as a statement although it in most cases appears as an expression since the resulting reference is assigned to a reference variable.

Do objects

This section is a reference section and may be skipped during a first reading.

It is sometimes useful to be able to group statements in a block of program text that has a name that can be referred to from e.g. break statements like leave and restart. A do-object may be used for this purpose and has the form:

```
Name: do SuperMethod
   Items
```

A do-object is executed as a statement. It differs from a data-item defined using `obj` in the sense that attributes in a do-object cannot be accessed and it is not possible to obtain a reference to a do-object. A do object is thus a singular method object.

The following is a sketchy example of using a do-object

```
aGhost: obj
   ...
   L: do
      ...
      if (aCondition) :then
         leave (L)
      ...
```

When the execution of `aGhost` reaches `L`, the do object is executed. If the `aCondition` evaluates to true, `leave(L)` implies that execution of `L` is terminated and execution continues after `L`.

On defining control abstractions

Just a few of the statement types described above are primitives of the qBeta language. These include assignment, the `if:then` statement and the break statements. All the other statement types described are defined in the qBeta library. The user of qBeta may in a similar way define his or her own control statements, defined by methods, which we here refer to as *control abstractions*. This is described in chapter . In this chapter, we also explain that most of the above control statements in fact describe invocations of *singular method objects* similar to singular data-objects declared using **`obj`**.