Variables

Variables in QL are used in a similar way to variables in algebra or logic. They represent sets of values, and those values are usually restricted by a formula.

This is different from variables in some other programming languages, where variables represent memory locations that may contain data. That data can also change over time. For example, in QL, `n = n + 1` is an equality formula that holds only if `n` is equal to `n + 1` (so in fact it does not hold for any numeric value). In Java, `n = n + 1` is not an equality, but an assignment that changes the value of `n` by adding `1` to the current value.

Declaring a variable

All variable declarations consist of a type and a name for the variable. The name can be any identifier that starts with a lowercase letter.

For example, `int i`, `SsaDefinitionNode node`, and `LocalScopeVariable lsv` declare variables `i`, `node`, and `lsv` with types `int`, `SsaDefinitionNode`, and `LocalScopeVariable` respectively.

Variable declarations appear in different contexts, for example in a select clause, inside a quantified formula, as an argument of a predicate, and many more.

Conceptually, you can think of a variable as holding all the values that its type allows, subject to any further constraints.

For example, consider the following select clause:

```
from int i
where i in [0 .. 9]
select i
```

Just based on its type, the variable `i` could contain all integers. However, it is constrained by the formula `i in [0 .. 9]`. Consequently, the result of the select clause is the ten numbers between `0` and `9` inclusive.

As an aside, note that the following query leads to a compile-time error:

```
from int i
select i
```

In theory, it would have infinitely many results, as the variable `i` is not constrained to a finite number of possible values. For more information, see “Binding.”

Free and bound variables

Variables can have different roles. Some variables are **free**, and their values directly affect the value of an expression that uses them, or whether a formula that uses them holds or not. Other variables, called **bound** variables, are restricted to specific sets of values.

It might be easiest to understand this distinction in an example. Take a look at the following expressions:

```
"hello".indexOf("l")

min(float f | f in [-3 .. 3])

(i + 7) * 3

x.sqrt()
```

The first expression doesn't have any variables. It finds the (zero-based) indices of where `"l"` occurs in the string `"hello"`, so it evaluates to `2` and `3`.

The second expression evaluates to `-3`, the minimum value in the range `[-3 .. 3]`. Although this expression uses a variable `f`, it is just a placeholder or "dummy" variable, and you can't assign any values to it. You could replace `f` with a different variable without changing the meaning of the expression. For example, `min(float f | f in [-3 .. 3])` is always equal to `min(float other | other in [-3 .. 3])`. This is an example of a **bound variable**.

What about the expressions `(i + 7) * 3` and `x.sqrt()`? In these two cases, the values of the expressions depend on what values are assigned to the variables `i` and `x` respectively. In other words, the value of the variable has an impact on the value of the expression. These are examples of **free variables**.

Similarly, if a formula contains free variables, then the formula can hold or not hold depending on the values assigned to those variables [1]. For example:

```
"hello".indexOf("l") = 1

min(float f | f in [-3 .. 3]) = -3

(i + 7) * 3 instanceof int

exists(float y | x.sqrt() = y)
```

The first formula doesn't contain any variables, and it never holds (since `"hello".indexOf("l")` has values `2` and `3`, never `1`).

The second formula only contains a bound variable, so is unaffected by changes to that variable. Since `min(float f | f in [-3 .. 3])` is equal to `-3`, this formula always holds.

The third formula contains a free variable `i`. Whether or not the formula holds, depends on what values are assigned to `i`. For example, if `i` is assigned `1` or `2` (or any other `int`) then the formula holds. On the other hand, if `i` is assigned `3.5`, then it doesn't hold.

The last formula contains a free variable `x` and a bound variable `y`. If `x` is assigned a non-negative number, then the final formula holds. On the other hand, if `x` is assigned `-9` for example, then the formula doesn't hold. The variable `y` doesn't affect whether the formula holds or not.

For more information about how assignments to free variables are computed, see "evaluation of QL programs."

Footnotes

[1] This is a slight simplification. There are some formulas that are always true or always false, regardless of the assignments to their free variables. However, you won't usually use these when you're writing QL. For example, and `a = a` is always true (known as a tautology), and `x and not x` is always false.