Formulas

Formulas define logical relations between the free variables used in expressions.

Depending on the values assigned to those free variables, a formula can be true or false. When a formula is true, we often say that the formula *holds*. For example, the formula `x = 4 + 5` holds if the value `9` is assigned to `x`, but it doesn't hold for other assignments to `x`. Some formulas don't have any free variables. For example `1 < 2` always holds, and `1 > 2` never holds.

You usually use formulas in the bodies of classes, predicates, and select clauses to constrain the set of values that they refer to. For example, you can define a class containing all integers `i` for which the formula `i in [0 .. 9]` holds.

The following sections describe the kinds of formulas that are available in QL.

Comparisons

A comparison formula is of the form:

```
<expression> <operator> <expression>
```

See the tables below for an overview of the available comparison operators.

Order

To compare two expressions using one of these order operators, each expression must have a type and those types must be compatible and orderable.

Name	Symbol
Greater than	`>`
Greater than or equal to	`>=`
Less than	`<`
Less than or equal to	`<=`

For example, the formulas `"Ann" < "Anne"` and `5 + 6 >= 11` both hold.

Equality

To compare two expressions using `=`, at least one of the expressions must have a type. If both expressions have a type, then their types must be compatible.

To compare two expressions using `!=`, both expressions must have a type. Those types must also be compatible.

Name	Symbol
Equal to	`=`
Not equal to	`!=`

For example, `x.sqrt() = 2` holds if `x` is `4`, and `4 != 5` always holds.

For expressions `A` and `B`, the formula `A = B` holds if there is a pair of values—one from `A` and one from `B`—that are the same. In other words, `A` and `B` have at least one value in common. For example, `[1 .. 2] = [2 .. 5]` holds, since both expressions have the value `2`.

As a consequence, `A != B` has a very different meaning to the negation `not A = B` [1]:

- `A != B` holds if there is a pair of values (one from `A` and one from `B`) that are different.
- `not A = B` holds if it is *not* the case that there is a pair of values that are the same. In other words, `A` and `B` have no values in common.

Examples

1. If both expressions have a single value (for example `1` and `0`), then comparison is straightforward:
    - `1 != 0` holds.
    - `1 = 0` doesn't hold.
    - `not 1 = 0` holds.
2. Now compare `1` and `[1 .. 2]`:
    - `1 != [1 .. 2]` holds, because `1 != 2`.
    - `1 = [1 .. 2]` holds, because `1 = 1`.
    - `not 1 = [1 .. 2]` doesn't hold, because there is a common value (`1`).
3. Compare `1` and `int empty() { none() }` (a predicate defining the empty set of integers):
    - `1 != empty()` doesn't hold, because there are no values in `empty()`, so no values that are not equal to `1`.
    - `1 = empty()` also doesn't hold, because there are no values in `empty()`, so no values that are equal to `1`.
    - `not 1 = empty()` holds, because there are no common values.

Type checks

A type check is a formula that looks like:

```
<expression> instanceof <type>
```

You can use a type check formula to check whether an expression has a certain type. For example, `x instanceof Person` holds if the variable `x` has type `Person`.

Range checks

A range check is a formula that looks like:

```
<expression> in <range>
```

You can use a range check formula to check whether a numeric expression is in a given range. For example, `x in [2.1 .. 10.5]` holds if the variable `x` is between the values `2.1` and `10.5` (including `2.1` and `10.5` themselves).

Note that `<expression> in <range>` is equivalent to `<expression> = <range>`. Both formulas check whether the set of values denoted by `<expression>` is the same as the set of values denoted by `<range>`.

Calls to predicates

A call is a formula or expression that consists of a reference to a predicate and a number of arguments.

For example, `isThree(x)` might be a call to a predicate that holds if the argument `x` is `3`, and `x.isEven()` might be a call to a member predicate that holds if `x` is even.

A call to a predicate can also contain a closure operator, namely `*` or `+`. For example, `a.isChildOf+(b)` is a call to the transitive closure of `isChildOf()`, so it holds if `a` is a descendant of `b`.

The predicate reference must resolve to exactly one predicate. For more information about how a predicate reference is resolved, see "Name resolution."

If the call resolves to a predicate without result, then the call is a formula.

It is also possible to call a predicate with result. This kind of call is an expression in QL, instead of a formula. For more information, see "Calls to predicates (with result)."

Parenthesized formulas

A parenthesized formula is any formula surrounded by parentheses, `(` and `)`. This formula has exactly the same meaning as the enclosed formula. The parentheses often help to improve readability and group certain formulas together.

Quantified formulas

A quantified formula introduces temporary variables and uses them in formulas in its body. This is a way to create new formulas from existing ones.

Explicit quantifiers

The following explicit "quantifiers" are the same as the usual existential and universal quantifiers in mathematical logic.

`exists`

This quantifier has the following syntax:

```
exists(<variable declarations> | <formula>)
```

You can also write `exists(<variable declarations> | <formula 1> | <formula 2>)`. This is equivalent to `exists(<variable declarations> | <formula 1> and <formula 2>)`.

This quantified formula introduces some new variables. It holds if there is at least one set of values that the variables could take to make the formula in the body true.

For example, `exists(int i | i instanceof OneTwoThree)` introduces a temporary variable of type `int` and holds if any value of that variable has type `OneTwoThree`.

`forall`

This quantifier has the following syntax:

```
forall(<variable declarations> | <formula 1> | <formula 2>)
```

`forall` introduces some new variables, and typically has two formulas in its body. It holds if `<formula 2>` holds for all values that `<formula 1>` holds for.

For example, `forall(int i | i instanceof OneTwoThree | i < 5)` holds if all integers that are in the class `OneTwoThree` are also less than `5`. In other words, if there is a value in `OneTwoThree` that is greater than or equal to `5`, then the formula doesn't hold.

Note that `forall(<vars> | <formula 1> | <formula 2>)` is logically the same as `not exists(<vars> | <formula 1> | not <formula 2>)`.

`forex`

This quantifier has the following syntax:

```
forex(<variable declarations> | <formula 1> | <formula 2>)
```

This quantifier exists as a shorthand for:

```
forall(<vars> | <formula 1> | <formula 2>) and
exists(<vars> | <formula 1> | <formula 2>)
```

In other words, `forex` works in a similar way to `forall`, except that it ensures that there is at least one value for which `<formula 1>` holds. To see why this is useful, note that the `forall` quantifier could hold trivially. For example, `forall(int i | i = 1 and i = 2 | i = 3)` holds: there are no integers `i` which are equal to both `1` and `2`, so the second part of the body `(i = 3)` holds for every integer for which the first part holds.

Since this is often not the behavior that you want in a query, the `forex` quantifier is a useful shorthand.

Implicit quantifiers

Implicitly quantified variables can be introduced using “don’t care expressions.” These are used when you need to introduce a variable to use as an argument to a predicate call, but don’t care about its value. For further information, see “Don’t-care expressions.”

Logical connectives

You can use a number of logical connectives between formulas in QL. They allow you to combine existing formulas into longer, more complex ones.

To indicate which parts of the formula should take precedence, you can use parentheses. Otherwise, the order of precedence from highest to lowest is as follows:

1. Negation (not)
2. Conditional formula (if ... then ... else)
3. Conjunction (and)
4. Disjunction (or)
5. Implication (implies)

For example, `A and B implies C or D` is equivalent to `(A and B) implies (C or D)`.

Similarly, `A and not if B then C else D` is equivalent to `A and (not (if B then C else D))`.

Note that the parentheses in the above examples are not necessary, since they highlight the default precedence. You usually only add parentheses to override the default precedence, but you can also add them to make your code easier to read (even if they aren’t required).

QL also has two nullary connectives indicating the always true formula, `any()`, and the always false formula, `none()`.

The logical connectives in QL work similarly to Boolean connectives in other programming languages. Here is a brief overview:

any()

The built-in predicate `any()` is a formula that always holds.

Example

The following predicate defines the set of all expressions.

```
Expr allExpressions() {
  any()
}
```

none()

The built-in predicate `none()` is a formula that never holds.

Example

The following predicate defines the empty set of integers.

```
int emptySet() {
  none()
}
```

`not`

You can use the keyword `not` before a formula. The resulting formula is called a negation.

`not A` holds exactly when `A` doesn't hold.

Example

The following query selects files that are not HTML files.

```
from File f
where not f.getFileType().isHtml()
select f
```

> **Note**
>
> You should be careful when using `not` in a recursive definition, as this could lead to non-monotonic recursion. For more information, "Non-monotonic recursion."

`if ... then ... else`

You can use these keywords to write a conditional formula. This is another way to simplify notation: `if A then B else C` is the same as writing `(A and B) or ((not A) and C)`.

Example

With the following definition, `visibility(c)` returns `"public"` if `x` is a public class and returns `"private"` otherwise:

```
string visibility(Class c){
  if c.isPublic()
  then result = "public"
  else result = "private"
}
```

`and`

You can use the keyword `and` between two formulas. The resulting formula is called a conjunction.

`A and B` holds if, and only if, both `A` and `B` hold.

Example

The following query selects files that have the `js` extension and contain fewer than 200 lines of code:

```
from File f
where f.getExtension() = "js" and
```

```
  f.getNumberOfLinesOfCode() < 200
select f
```

`or`

You can use the keyword `or` between two formulas. The resulting formula is called a disjunction.

`A or B` holds if at least one of `A` or `B` holds.

Example

With the following definition, an integer is in the class `OneTwoThree` if it is equal to `1`, `2`, or `3`:

```
class OneTwoThree extends int {
  OneTwoThree() {
    this = 1 or this = 2 or this = 3
  }
}
```

`implies`

You can use the keyword `implies` between two formulas. The resulting formula is called an implication. This is just a simplified notation: `A implies B` is the same as writing `(not A) or B`.

Example

The following query selects any `SmallInt` that is odd, or a multiple of `4`.

```
class SmallInt extends int {
  SmallInt() { this = [1 .. 10] }
}

from SmallInt x
where x % 2 = 0 implies x % 4 = 0
select x
```

Footnotes

[1] The difference between `A != B` and `not A = B` is due to the underlying quantifiers. If you think of `A` and `B` as sets of values, then `A != B` means:

```
exists( a, b | a in A and b in B | a != b )
```

On the other hand, `not A = B` means:

```
not exists( a, b | a in A and b in B | a = b )
```

This is equivalent to `forall( a, b | a in A and b in B | a != b )`, which is very different from the first formula.