公式（Formulas）

公式（Formulas）

公式用来定义表达式中使用的自由变量之间的逻辑关系
根据分配给这些自由变量的值，公式式可以为true或false。当公式式为真时，我们常说该公式式*成立*。例如，x=4+5 这个公式当x=9时成立，当x为其他值时不成立，有些公式可能没有自变量，比如1<2始终成立，1>2始终不成立

比较运算（Comparisons）

比较公式的形式为：

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

有关可用比较运算符的概述，请参见下表。

排序（Order）

要使用这些顺序运算符之一比较两个表达式，每个表达式必须具有一个类型，并且这些类型必须兼容且 可排序。

名称	符号
大于	`>`
大于或等于	`>=`
少于	`<`
小于或等于	`<=`

例如，公式和两者都成立。`"Ann" < "Anne"` 、 `5 + 6 >= 11`

相等（Equality）

要使用比较两个表达式`=`，至少一个表达式必须具有类型。如果两个表达式都具有类型，则它们的类型必须兼容。

要使用比较两个表达式`!=`，两个表达式必须具有一个类型。这些类型也必须兼容。

名称	符号
等于 （tips：存在交集）	`=`
不等于（tips：存在差集）	`!=`

例如：当x=4时 x.sqrt()=2 成立， 4!=5总是成立

对于表达式`A`和`B`，如果存在一对相同的值（一个来自一个，另一个来自），则公式将成立。换句话说，和具有至少一个共同的值。例如，[1,2] = [2,5]由于两个表达式都具有一个值是2，所以 [1,2] = [2,5] 同时因为[1,2] 和[2,5]存在差集，所以[1,2]!=[2,5]

例子：

1. 如果两个表达式都有一个值（例如`1`和`0`），则比较很简单：
   - `1 != 0` 成立。
   - `1 = 0` 不成立。
   - `not 1 = 0` 成立。
2. 现在，比较`1`和：`[1 .. 2]`
   - `1 != [1 .. 2]`成立，因为。`1 != 2`
   - `1 = [1 .. 2]`成立，因为。`1 = 1`
   - `not 1 = [1 .. 2]`不成立，因为有一个公共值（`1`）。
3. 比较`1`和`none()`（“空集”）：
   - `1 != none()`不成立，因为中没有值`none()`，因此没有不等于的值`1`。
   - `1 = none()`也不成立，因为中没有值`none()`，因此没有等于的值`1`。
   - `not 1 = none()` 成立，因为没有共同值。

类型检查（Type checks）

类型检查用来检查一个表达式的类型,其方法为

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

您可以使用类型检查公式来检查表达式是否具有某种类型。例如，如果变量的类型为，则成立。`x instanceof Person`成立，如果x有一个可用的type是Person
tips：instanceof 在安全规则里是非常常用的一个运算符

范围检查（Range checks）

范围检查是用来检查表达式是否在一定的范围内

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

您可以使用范围检查公式来检查数字表达式是否在给定范围内。例如：x in [2.1 .. 10.5] 成立如果x满足处于2.1和10.5之间
tips：<expression> in <range> 和 <expression> = <range>是一样的，都是代表表达式是否有相同的交集

谓词调用（Calls to predicates）

调用是一个公式或表达式，由对谓词的引用和多个参数组成。
例如，`isThree(x)`可能是谓词isThree对x是否是3进行判断，`x.isEven()`可能是对x是否为偶数进行判断。
谓词的调用也可以包含闭包运算符，即`*`或`+`。例如，`a.isChildOf+(b)`是对的传递闭包的 调用`isChildOf()`，因此如果`a`是b的后代则成立。

带括号的公式（Parenthesized formulas）

括号内的公式是任何用括号`(`和括起来的公式`)`。该公式与所附公式具有完全相同的含义。括号通常有助于提高可读性并将某些公式组合在一起。

量化公式（Quantified formulas）

量化公式会引入临时变量，并在其体内的公式中使用它们。这是从现有公式创建新公式的方法。

内置量化名词（Explicit quantifiers）

以下明确的“量词”与数学逻辑中通常的存在量和通用量词相同。

exists

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

此量化公式引入了一些新变量。它确定变量是否可以使用至少一组值来使主体中的公式为真。
例如：exists(int i | i instanceof OneTwoThree)，引入一个满足OneTwoThree类型的变量i

forall

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

`forall`引入了一些新变量，并且通常在其体内具有两个公式。它对所有保持的值成立则算式成立
例如：`forall(int i | i instanceof OneTwoThree | i < 5)`，当i属于OneTwoThree 并且i<5时，算式才成立，如果i有一个值大于5 则不成立
tips： forall(<vars> | <formula 1> | <formula 2>和not exists(<vars> | <formula 1> | not <formula 2>)逻辑上相同（包含的概念formula2的集合包含formula1）

forex

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

此量词是以下内容的简写形式：

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

举个例子：forall（int i | i=1 and i=2 | i=3）成立，因为i=2 and i=1为空集，表达式i=3 当3时成立，空集是任何集合的子集，所以forall（int i | i=1 and i=2 | i=3）成立 所以forex是非常实用的的，即他可以得出满足条件的非空集合

逻辑连接词（Logical connectives）

您可以在QL中的公式之间使用许多逻辑连接词。它们使您可以将现有公式组合成更长，更复杂的公式。要指出公式的哪个部分优先，可以使用括号。否则，优先级从最高到最低的顺序如下：

1、否定（not）
2、条件选择（if ... then ... else）
3、与 （and）
4、或 （or）
5、蕴含（implies）

您可以`implies`在两个公式之间使用关键字。所得公式称为蕴涵。这只是一种简化的表示法：`A implies B` 和`(not A) or B`.等价
例子：
选择SmallInt里的奇数和4的倍数

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

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

脚注：
A！= B而不是A = B之间的差异是由于底层的数据所致。如果将A和B视为一组集合，则A！= B表示：

```
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 )
```

这相当于forall(a, b | a in A and b in B | a != b)这与第一个公式有很大的不同。