表达式(Expressions)

表达式(Expressions)

表达式的计算结果为一组值并具有一个类型。
例如，表达式 1+2 计算结果为整数3，表达式"QL" 的计算结果为字符串类型输出"QL" ,1 + 2 这个表达式的类型为int，而"QL"的表达式类型为string

引用变量（Variable references）

变量引用是已声明变量的名称。这种类型的表达式与其所引用的变量具有相同的类型。
例如，如果你已经声明的变量和，然后表达和有类型和分别int i，和 LocalScopeVariable lsv，i和lsv分别是int类型和LocalScopeVariable类型

您还可以引用变量`this`和`result`。这些在谓词定义中使用，并且以与其他变量引用相同的方式起作用。

常量（Literal）

您可以直接在QL中表达某些值，例如数字，布尔值和字符串。

- 布尔文字：这些是`true`和`false`。
- 整数文字：这些是十进制数字（`0`到`9`）的序列，可能以减号（`-`）开头。例如：

```
0
42
-2048
```

- 浮点文字：这些是由点号（`.`）分隔的十进制数字序列，可能以减号（`-`）开头。例如：

```
2.0
123.456
-100.5
```

- 字符串文字：这些是16位字符的有限字符串。您可以通过将字符括在引号（`"..."`）中来定义字符串文字。大多数字符代表自己，但是您需要使用反斜杠“转义”一些字符。以下是字符串文字的示例：

```
"hello"
"They said, \"Please escape quotation marks!\""
```

注意：QL中没有“日期文字”。相反，要指定一个date，您应该使用`toDate()`谓词将字符串转换为它表示的日期。例如，`"2016-04-03".toDate()`日期是2016年4月3日，是2000年新年后一秒的时间点。`"2000-01-01 00:00:01".toDate()`

- 以下字符串格式被识别为日期：
  - **ISO日期**，例如。秒部分是可选的（假设丢失了），整个时间部分也可以丢失（在这种情况下，它被假定为）。`"2016-04-03 17:00:24"` `"00"` `"00:00:00"`
  - **缩写ISO日期**，例如`"20160403"`。
  - **英式日期**，例如`"03/04/2016"`。
  - **详细的日期**，例如。`"03 April 2016"`

括号表达式（Parenthesized expressions）

带括号的表达式是用括号`(`和括起来的表达式`)`。此表达式的类型和值与原始表达式完全相同。括号可用于将表达式分组在一起以消除歧义并提高可读性。

范围表达式(Ranges)

范围表达式表示在两个表达式之间排序的值的范围。它由两个表达式分隔，`..`并用方括号（`[`和`]`）括起来。例如，是有效的范围表达式。它的值是和之间（包括和本身）之间的任何整数 。`[3 .. 7]`表示3和7内所有的整数在有效范围内，开始和结束表达式是整数，浮点数或日期。如果其中之一是日期，则两个都必须是日期。如果其中一个是整数，另一个是浮点数，则两者都将被视为浮点数。

常量范围表达式(Set literal expressions)

可以设置一个常量范围表达式，例如：`[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]`它表示30以内的质数
对于有效的集合文字表达式，所包含表达式的值必须为兼容类型。此外，集合元素中的至少一个元素必须是所有其他包含的表达式的类型的超类型。
从CodeQL CLI的2.1.0版和LGTM Enterprise的1.24版开始支持常量范围表达式。

超级表达式（Super expressions）

QL中的超级表达式类似于其他编程语言（例如Java）中的超级表达式。当您要使用超类型的谓词定义时，可以在谓词调用中使用它们。实际上，当谓词从其超类型继承两个定义时，这很有用。在这种情况下，谓词必须覆盖 这些定义以避免歧义。但是，如果要使用特定超类型的定义而不是编写新定义，则可以使用超级表达式

在下面的示例中，该类`C`继承了谓词的两个定义`getANumber()`-`A`一个来自和一个来自`B`。而不是覆盖两个定义，它使用中的定义`B`

```
class A extends int {
  A() { this = 1 }
  int getANumber() { result = 2 }
}

class B extends int {
  B() { this = 1 }
  int getANumber() { result = 3 }
}

class C extends A, B {
```

```
12   // Need to define `int getANumber()`; otherwise it would be amb
   iguous
13   int getANumber() {
14     result = B.super.getANumber()
15   }
16 }
17
18 from C c
19 select c, c.getANumber()
```

该查询的结果是。`1, 3`

包含返回值的谓词调用(Calls to predicates (with result))

带有结果的谓词的调用本身就是表达式，与没有结果的谓词（即公式）的调用不同。有关更多信息，请参见“谓词的调用”。
对具有结果的谓词的调用将求值为`result`被调用谓词的变量的值。
例如`a.getAChild()`，`getAChild()`对变量谓词的调用`a`。此调用计算结果为的子集`a`。

聚合（Aggregations）

聚合是一种映射，它根据公式指定的一组输入值来计算结果值。
通用语法为：

```
1 <aggregate>(<variable declarations> | <formula> | <expression>)
```

变量声明在被称为聚集变量。
有序聚集体（即min，max，rank，concat，和strictconcat）由他们下令值默认。顺序是数字（对于整数和浮点数）或字典（对于字符串）。字典顺序基于 每个字符的Unicode值。

要指定不同的顺序，请紧跟<expression>关键字，然后是指定顺序的表达式，并可选地加上关键字或 （以确定是按升序还是降序对表达式进行排序）。如果您未指定顺序，则默认为。order byascdescasc

QL中提供以下汇总：

- count：此汇总确定汇总变量每个可能分配的的不同值的数量。
  例如，以下聚合返回的文件数多于 500行数：

```
count(File f | f.getTotalNumberOfLines() > 500 | f)
```

如果没有满足公式的聚合变量的可能赋值，例如 ，则默认为value 。`count(int i | i = 1 and i = 2 | i)` count的结果是 0

- min 和 max，这些聚合确定聚合变量可能分配中的最小（min）或最大（max）值。在这种情况下，必须为数字类型或类型string。

例如，以下聚合返回`.js`行数最多的一个或多个文件的名称：

```
max(File f | f.getExtension() = "js" | f.getBaseName() order by f.
getTotalNumberOfLines())
```

以下汇总返回`s`下面提到的三个字符串中的最小字符串，即，按的所有可能值的字典顺序排列的第一个字符串`s`。（在这种情况下，它返回。）`"De Morgan"`

```
min(string s | s = "Tarski" or s = "Dedekind" or s = "De Morgan" |
s)
```

- avg 此汇总确定`<expression>`汇总变量所有可能分配的平均值。的类型`<expression>`必须为数字。如果没有满足公式的聚合变量的可能赋值，则聚合将失败并且不返回任何值。换句话说，它求值为空集。

例如，下面的聚合返回平均整数的`0`，`1`， `2`，和`3`：

```
avg(int i | i = [0 .. 3] | i)
```

- `sum`：此汇总确定`<expression>`汇总变量所有可能分配的值之和。的类型`<expression>`必须为数字。如果没有满足公式的聚合变量的可能赋值，则总和为`0`。

例如，以下聚合返回和的所有可能值的和：`i * j`

```
sum(int i, int j | i = [0 .. 2] and j = [3 .. 5] | i * j)
```

- `concat`：此聚合将`<expression>`所有可能赋值的值连接到聚合变量。请注意，该`<expression>`类型必须为 `string`。如果没有满足公式的聚合变量的可能赋值，则`concat`默认为空字符串。

例如，下面的聚合返回字符串`"3210"`，即，字符串的串联`"0"`， `"1"`， `"2"`，和`"3"`按降序排列：

```
concat(int i | i = [0 .. 3] | i.toString() order by i desc)
```

所述`concat`聚集体还可以采取的第二表达，从用逗号的第一个分离。将第二个表达式作为分隔符插入每个串联值之间。
例如，以下聚合返回`"0|1|2|3"`：

```
concat(int i | i = [0 .. 3] | i.toString(), "|")
```

- `rank`：此汇总采用的可能值`<expression>`并对它们进行排名。在这种情况下，`<expression>`必须为数字类型或类型`string`。聚合返回在**rank表达式**指定的位置中排名的值。您必须在关键字后面的方括号中包括该等级表达式`rank`。

例如，以下聚合返回在所有可能值中排名第四的值。在这种情况下，`8`是范围从`5`到 的第4个整数`15`：

```
rank[4](int i | i = [5 .. 15] | i)
```

请注意，排名索引从开始`1`，因此不`rank[0](...)`返回任何结果。

- `strictconcat`，`strictcount`和`strictsum`：这些聚集的工作一样`concat`， `count`和`sum`分别，除了他们是*严格的*。也就是说，如果没有满足公式的聚合变量的可能赋值，则整个聚合将失败并计算为空集（而不是默认为`0`或空字符串）。如果您只对聚合主体不重要的结果感兴趣，这将很有用。
- `unique`：此聚合取决于`<expression>`聚合变量的所有可能分配的值。如果`<expression>`在聚合变量中有一个唯一的值，则聚合将求值到该值。否则，合计没有价值。

例如，下面的查询返回的正整数`1`， `2`， `3`，`4`，`5`。对于负整数`x`，表达式`x`和`x.abs()`具有不同的值，因 此`y`聚合表达式中的值不是唯一确定的。

```
from int x
where x in [-5 .. 5] and x != 0
select unique(int y | y = x or y = x.abs() | y)
```

如何检查评估聚合（Evaluation of aggregates）

总的来说，综合评估涉及以下步骤：

1. 确定输入变量：这些是在聚合中声明的聚合变量，以及在聚合的某些组件中使用的在聚合外部声明的变量。`<variable declarations>`

2. 生成输入变量的值的所有可能的不同元组（组合），使 `<formula>` 成立。请注意，聚合变量的相同值可能会出现在多个不同的元组中。处理元组时，具有相同值的所有此类出现均被视为不同的出现。
3. `<expression>` 在每个元组上应用并收集生成的（不同的）值。`<expression>` 在元组上应用可能会导致生成多个值。
4. 将聚合函数应用于步骤3中生成的值以计算最终结果。

让我们将这些步骤应用于 `sum` 以下查询中的汇总：

```
select sum(int i, int j |
    exists(string s | s = "hello".charAt(i)) and exists(string s |
  s = "world!".charAt(j)) | i)
```

- 输入变量i.j
- 满足给定条件的所有可能元组：（<value of i><value of j>）`(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 0), (1, 1), ..., (4, 5)`.在此步骤中将生成30个元组
- 将<expression>应用在所有元组的值中，这意味着筛选出所有符合条件i和j的值0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4,4, 4.
- 将使用sum公式聚合搜索出来的值，总额为60

如果我们改变表达式为i+j，这个查询的结果将是135，i+j 在所有的元组内的值是：0, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 2, 3, 4, 5, 6, 7, 3, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8, 9

然后我们来看下一个查询：

```
select count(string s | s = "hello" | s.charAt(_))
```

- s 是聚合的输入变量。
- "hello"在此步骤中将生成一个元组
- 该应用在这个元组。下划线其表示任意值,意味着不在乎他的值，也是一种内置的表达式。 产生四个不同的值 h，e，l，o

  tips：这里的charAt(_)表示的是组成s的元素集合，所以只会输出h，e，l，o，但是如果在s.charAt(3)输出结果为l（游标从0开始）
- 最后，`count` 对这些值应用，查询返回 `4`

省略聚合的一部分（Omitting parts of an aggregation）

聚合的三个部分并不总是必需的，因此您通常可以以更简单的形式编写聚合

1、如果要编写形式的集合，则可以省略和部分，并按如下方式编写：<aggregate>(<type> v | <expression> = v | v)<variable declarations><formula>

```
<aggregate>(<expression>)
```

例如，以下聚合确定字母l在string中出现的次数"hello"。这些形式是等效的：

```
count(int i | i = "hello".indexOf("l") | i)
count("hello".indexOf("l"))
```

2、如果只有一个聚合变量，则可以省略该`<expression>`部分。在这种情况下，表达式被视为聚合变量本身。例如，以下聚合是等效的：

```
avg(int i | i = [0 .. 3] | i)
avg(int i | i = [0 .. 3])
```

3、作为一种特殊情况，即使存在多个聚集变量，也可以省略该`<expression>`部分`count`。在这种情况下，它将计算满足该公式的聚合变量的不同元组的数量。换句话说，表达式部分被认为是常量`1`。例如，以下聚合是等效的：

```
count(int i, int j | i in [1 .. 3] and j in [1 .. 3] | 1)
count(int i, int j | i in [1 .. 3] and j in [1 .. 3])
```

4、您可以省略`<formula>`零件，但是在这种情况下，您应该包括两个垂直条：

```
<aggregate>(<variable declarations> | | <expression>)
```

5、最后，您也可以同时省略`<formula>`和`<expression>`部分。例如，以下聚合是计算数据库中文件数量的等效方法：

```
count(File f | any() | 1)
count(File f | | 1)
count(File f)
```

单调聚合（Monotonic aggregates）

除标准聚合外，QL还支持单调聚合。单调聚合与标准聚合的区别在于它们处理`<expression>`公式部分生成的值的方式：

- 标准聚合将`<expression>`每个`<formula>`值的值取整，然后将它们展平到列表中。单个聚合函数将应用于所有值。
- 单调集合`<expression>`对于给出的每个值取一个`<formula>`，并创建所有可能值的组合。聚合函数将应用于每个结果组合。

通常，如果a`<expression>`是合计的并且是功能性的，则单调聚合等于标准聚合。如果：`<expression>` 生成的每个值都不完全相同，则结果不同`<formula>`：

- 如果缺少`<expression>`值（即，`<expression>`生成的值不存在任何 值`<formula>`），则单调聚合将不会计算结果，因为您无法`<expression>`为生成的每个值创建仅包含一个值的值组合`<formula>`。
- 如果`<expression>`每个`<formula>`结果不止一个，则可以创建多个值组合，包括`<expression>`为生成的每个值恰好一个值`<formula>`。在此，将聚合函数应用于每个结果组合。

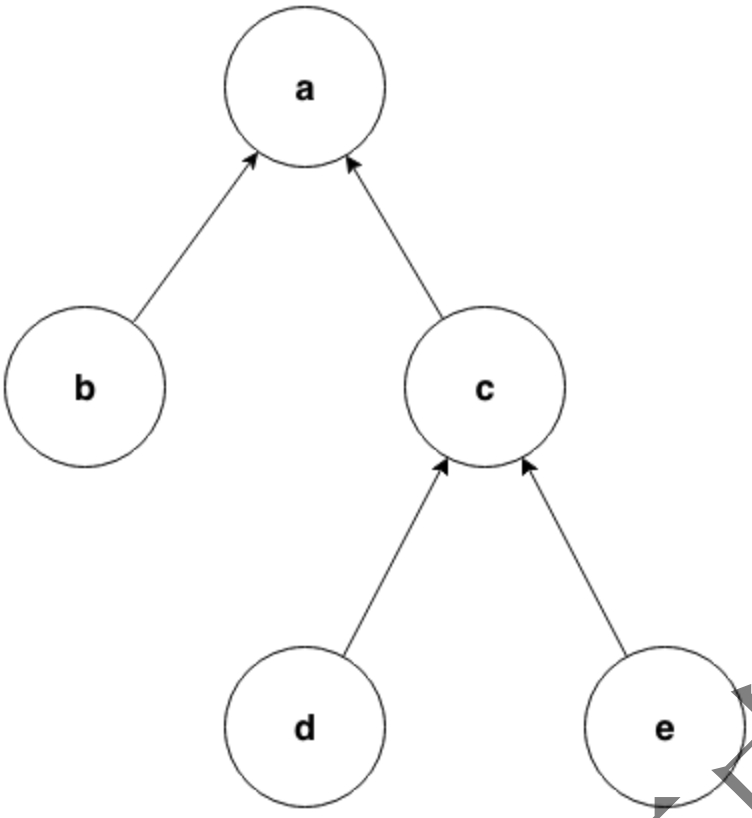
递归单调聚合（Recursive monotonic aggregates）

可以递归使用单调聚合，但是递归调用只能出现在表达式中，而不能出现在范围内。聚合的递归语义与其余QL的递归语义相同。例如，我们可以定义一个谓词来计算图中一个节点到叶子的距离，如下所示：

```
1 int depth(Node n) {
2   if not exists(n.getAChild())
3   then result = 0
4   else result = 1 + max(Node child | child = n.getAChild() | depth
  (child))
5 }
```

这里的递归调用在表达式中，这是合法的。聚合的递归语义与其余QL的递归语义相同。如果您了解聚合在非递归情况下的工作原理，那么您应该不会很难递归地使用它们。但是，值得一看的是递归聚合的评估如何进行。

考虑一下我们刚才用下面的图作为输入的深度示例（箭头从孩子到父母）：



阶段	深度	评论
0		我们总是从空集开始。
1个	`(0, b), (0, d), (0, e)`	无子女的节点具有深度0的递归步骤—和c不能产生价值，因为他们的一些孩子没有值`depth`。
2	`(0, b), (0, d), (0, e), (1, c)`	c的递归步骤成功了，因为`depth`现在对其所有子代（d和e）都有一个值。对于递归步骤一个仍然失败。
3	`(0, b), (0, d), (0, e), (1, c), (2, a)`	对于递归步骤一个成功，因为`depth`现在已经为所有的孩子（一个值b和c）。

在这里，我们可以看到，在中间阶段，如果某些子项缺少值，则聚合失败非常重要–这可以防止添加错误的值。

Any表达式（Any）

`any`表达式的一般语法类似于聚合的语法，即：

```
any(<variable declarations> | <formula> | <expression>)
```

您应该始终包括变量声明，但是 公式和表达式部分是可选的。

该`any`表达式表示具有特定形式并满足特定条件的任何值。更准确地说，`any`表达式是：

1. 介绍临时变量。
2. 将其值限制为满足该`<formula>`部分的值（如果存在）。
3. `<expression>`为每个变量返回。如果没有任何`<expression>`部分，那么它将返回变量本身。

下表列出了一些不同形式的`any`表达式的示例：

Expression	Values
`any(File f)`	all `File`s in the database
`any(Element e | e.getName())`	the names of all `Element`s in the database
`any(int i | i = [0 .. 3])`	the integers `0`, `1`, `2`, and `3`
`any(int i | i = [0 .. 3] | i * i)`	the integers `0`, `1`, `4`, and `9`

```
1 还有一个内置谓词 any()。这是一个始终成立的谓词。
```

一元运算（Unary operations）

一元运算是减号（`-`）或加号（`+`），后跟类型`int`或的表达式`float`。例如：

```
1 -6.28
2 +(10 - 4)
3 +avg(float f | f = 3.4 or f = -9.8)
4 -sum(int i | i in [0 .. 9] | i * i)
```

加号使表达式的值保持不变，而减号使值的算术求反。

二进制运算（Binary operations）

```
1 5 % 2
2 (9 + 1) / (-2)
3 "0" + "L"
4 2 * min(float f | f in [-3 .. 3])
```

如果两个表达式都是数字，则这些运算符将充当标准算术运算符。例如:10.6 - 3.2 结果值为7.4，123.456 * 0 值为0，9%4 值为1（取余数）。如果两个操作数均为整数，则结果为整数。否则，结果为浮点数。
您还可以将其`+`用作字符串连接运算符。在这种情况下，至少一个表达式必须是字符串，而另一个表达式将使用`toString()`谓词隐式转换为字符串 。这两个表达式是连接在一起的，结果是一个字符串。例如，表达式具有value 。`221 + "B"` 的结果为`"221B"`

强制转换（Casts）

强制转换可让您限制表达式的类型。这类似于使用其他语言（例如Java）进行转换。
您可以通过两种方式编写演员表：

- 作为“后缀”强制转换：在圆括号后加点号和类型名称。例如，`x.(Foo)`将类型限制`x`为`Foo`。
- 作为“前缀”强制转换：括号中的类型，后跟另一个表达式。例如，`(Foo)x`还将的类型限制`x`为`Foo`。

请注意，后缀强制转换等同于用括号括起来的前缀强制转换-`x.(Foo)` 完全等同于`((Foo)x)`。
如果要调用仅针对更特定类型定义的成员谓词，则强制转换非常有用。例如，以下查询选择 具有直接超类型“ List”的Java 类：

```
1 import java
2
3 from Type t
4 where t.(Class).getASupertype().hasName("List")
5 select t
```

由于谓词`getASupertype()`是为定义的`Class`，但不是为定义的，因此`Type`您不能`t.getASupertype()`直接调用。强制转换`t.(Class)`确保`t`类型为`Class`，因此它可以访问所需的谓词。
如果您更喜欢使用前缀转换，则可以将`where`子句重写为：

```
1 where ((Class)t).getASupertype().hasName("List")
```

"不在乎"表达式(Don’t-care expressions)

这是写为单个下划线的表达式`_`。它代表任何值。（您“不在乎”值是什么。）
与其他表达式不同，“无关”表达式没有类型。实际上，这意味着`_`没有任何成员谓词，因此您不能调用`_.somePredicate()`。
例如，以下查询选择字符串中的所有字符`"hello"`：

```
1 from string s
2 where s = "hello".charAt(_)
3 select s
```

charAt（int i）谓词是在字符串上定义的，通常采用int参数。这里的无关表达式_用来告诉查询在每个可能的索引处选择字符,这个查询返回的值是h、e、l和o

```
1 tips: _ 在规则中是一个经常用到的表达式，需要理解该表达式的含义才能在编写规则时不
  出现问题
```