递归（Recursion）

递归（Recursion）

QL为递归提供了有力的支持。如果QL中的谓词直接或间接依赖于自身，则谓词是递归的。
为了评估递归谓词，QL编译器会找到递归的最小固定点。特别是，它以空值集开始，并通过重复应用谓词直到值集不再更改来查找新值。该集合是最不固定的点，因此是评估的结果。同样，QL查询的结果是查询中引用的谓词的最小固定点。
在某些情况下，您还可以递归使用聚合。有关更多信息，请参见“单调聚合”。

递归谓词的例子

这是QL中递归谓词的一些示例：

从0到100计数

以下查询使用谓词`getANumber()`列出0到100（含）之间的所有整数：

```
int getANumber() {
  result = 0
  or
  result <= 100 and result = getANumber() + 1
}

select getANumber()
```

谓词`getANumber()`对包含的集合求值，`0`并且该整数比集合中已经存在的数字（最多，包括`100`）多一个。

相互递归

谓词可以是相互递归的，也就是说，您可以拥有一个相互依赖的谓词循环。例如，这是一个使用偶数计数为100的QL查询：

```
int getAnEven() {
  result = 0
  or
  result <= 100 and result = getAnOdd() + 1
}

int getAnOdd() {
  result = getAnEven() + 1
}

select getAnEven()
```

该查询的结果是从0到100的偶数。您可以替换为以列出从1到101的奇数。`select getAnEven()select getAnOdd()`

闭包传递

谓词的闭包传递是递归谓词，其结果是通过重复应用原始谓词获得的。

特别是，原始谓词必须具有两个参数（可能包括a`this`或`result`value），并且这些参数必须具有兼容的type。

由于传递闭包是递归的一种常见形式，因此QL有两个有用的缩写：

1、传递闭包 `+`

例如，假设您有一个`Person`带有成员谓词 的类`getAParent()`。然后`p.getAParent()`返回的任何父母`p`。传递闭包`p.getAParent+()`返回的父母`p`，的父母的父母 `p`，依此类推。

使用此`+`表示法通常比显式定义递归谓词更简单。在这种情况下，显式定义可能如下所示：

```
Person getAnAncestor() {
  result = this.getAParent()
  or
  result = this.getAParent().getAnAncestor()
}
```

谓词`getAnAncestor()`等价于`getAParent+()`。

2、自反传递封闭 `*`

这类似于上面的传递闭包运算符，不同之处在于您可以使用它为谓词对其自身应用零次或多次。

例如，的结果`p.getAParent*()`是`p`（如上）的祖先，或者是`p`它本身。

在这种情况下，显式定义如下所示：

```
1 Person getAnAncestor2() {
2   result = this
3   or
4   result = this.getAParent().getAnAncestor2()
5 }
```

谓词`getAnAncestor2()`等价于`getAParent*()`。

限制和常见错误

虽然QL专为查询递归数据而设计，但有时很难正确定义递归定义。如果递归定义包含错误，则通常不会得到任何结果或编译器错误。

以下示例说明了导致无效递归的常见错误

空递归

首先，有效的递归定义必须具有起点或*基本情况*。如果递归谓词求值为空值集，则通常会有问题。

例如，您可以尝试如下定义谓词`getAnAncestor()`（来自 以上示例）：

```
1 Person getAnAncestor() {
2   result = this.getAParent().getAnAncestor()
3 }
```

在这种情况下，QL编译器给出一个错误，指出这是一个空的递归调用。

由于`getAnAncestor()`最初假定为空，因此无法添加新值。谓词需要递归的起点，例如：

```
1 Person getAnAncestor() {
2   result = this.getAParent()
3   or
4   result = this.getAParent().getAnAncestor()
5 }
```

非单调递归

有效的递归谓词也必须是单调的。这意味着（相互）递归仅在偶数个否定的情况下才允许。
直观上，这可以防止“递归骗子悖论”的情况，在这种情况下无法解决递归问题。例如：

```
predicate isParadox() {
  not isParadox()
}
```

根据此定义，谓词`isParadox()`恰好在不成立时成立。这是不可能的，因此没有递归的定点解决方案。
如果递归在偶数个否定的情况下出现，那么这不是问题。例如，考虑以下（稍微令人震惊）的class成员谓词`Person`：

```
predicate isExtinct() {
  this.isDead() and
  not exists(Person descendant | descendant.getAParent+() = this |
    not descendant.isExtinct()
  )
}
```

`p.isExtinct()`持有，如果`p`并且所有`p`的后代都死了。
对的递归调用`isExtinct()`嵌套在偶数个否定数中，因此这是一个有效的定义。实际上，您可以按如下方式重写定义的第二部分：

```
forall(Person descendant | descendant.getAParent+() = this |
  descendant.isExtinct()
)
```