Recursion

QL provides strong support for recursion. A predicate in QL is said to be recursive if it depends, directly or indirectly, on itself.

To evaluate a recursive predicate, the QL compiler finds the least fixed point of the recursion. In particular, it starts with the empty set of values, and finds new values by repeatedly applying the predicate until the set of values no longer changes. This set is the least fixed point and hence the result of the evaluation. Similarly, the result of a QL query is the least fixed point of the predicates referenced in the query.

In certain cases, you can also use aggregates recursively. For more information, see "Monotonic aggregates."

Examples of recursive predicates

Here are a few examples of recursive predicates in QL:

Counting from 0 to 100

The following query uses the predicate `getANumber()` to list all integers from 0 to 100 (inclusive):

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

select getANumber()
```

The predicate `getANumber()` evaluates to the set containing `0` and any integers that are one more than a number already in the set (up to and including `100`).

Mutual recursion

Predicates can be mutually recursive, that is, you can have a cycle of predicates that depend on each other. For example, here is a QL query that counts to 100 using even numbers:

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

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

select getAnEven()
```

The results of this query are the even numbers from 0 to 100. You could replace `select getAnEven()` with `select getAnOdd()` to list the odd numbers from 1 to 101.

Transitive closures

The transitive closure of a predicate is a recursive predicate whose results are obtained by repeatedly applying the original predicate.

In particular, the original predicate must have two arguments (possibly including a `this` or `result` value) and those arguments must have compatible types.

Since transitive closures are a common form of recursion, QL has two helpful abbreviations:

1. **Transitive closure** `+`

   To apply a predicate **one** or more times, append `+` to the predicate name.

   For example, suppose that you have a class `Person` with a member predicate `getAParent()`. Then `p.getAParent()` returns any parents of `p`. The transitive closure `p.getAParent+()` returns parents of `p`, parents of parents of `p`, and so on.

   Using this `+` notation is often simpler than defining the recursive predicate explicitly. In this case, an explicit definition could look like this:

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

   The predicate `getAnAncestor()` is equivalent to `getAParent+()`.

2. **Reflexive transitive closure** `*`

   This is similar to the above transitive closure operator, except that you can use it to apply a predicate to itself **zero** or more times.

   For example, the result of `p.getAParent*()` is an ancestor of `p` (as above), or `p` itself.

   In this case, the explicit definition looks like this:

   ```
   Person getAnAncestor2() {
     result = this
     or
     result = this.getAParent().getAnAncestor2()
   }
   ```

   The predicate `getAnAncestor2()` is equivalent to `getAParent*()`.

Restrictions and common errors

While QL is designed for querying recursive data, recursive definitions are sometimes difficult to get right. If a recursive definition contains an error, then usually you get no results, or a compiler error.

The following examples illustrate common mistakes that lead to invalid recursion:

Empty recursion

Firstly, a valid recursive definition must have a starting point or *base case*. If a recursive predicate evaluates to the empty set of values, there is usually something wrong.

For example, you might try to define the predicate `getAnAncestor()` (from the above example) as follows:

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

In this case, the QL compiler gives an error stating that this is an empty recursive call.

Since `getAnAncestor()` is initially assumed to be empty, there is no way for new values to be added. The predicate needs a starting point for the recursion, for example:

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

Non-monotonic recursion

A valid recursive predicate must also be monotonic. This means that (mutual) recursion is only allowed under an even number of negations.

Intuitively, this prevents “liar’s paradox” situations, where there is no solution to the recursion. For example:

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

According to this definition, the predicate `isParadox()` holds precisely when it doesn’t hold. This is impossible, so there is no fixed point solution to the recursion.

If the recursion appears under an even number of negations, then this isn’t a problem. For example, consider the following (slightly macabre) member predicate of class `Person`:

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

`p.isExtinct()` holds if `p` and all of `p`’s descendants are dead.

The recursive call to `isExtinct()` is nested in an even number of negations, so this is a valid definition. In fact, you could rewrite the second part of the definition as follows:

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