Predicates

Predicates are used to describe the logical relations that make up a QL program.

Strictly speaking, a predicate evaluates to a set of tuples. For example, consider the following two predicate definitions:

```
predicate isCountry(string country) {
  country = "Germany"
  or
  country = "Belgium"
  or
  country = "France"
}

predicate hasCapital(string country, string capital) {
  country = "Belgium" and capital = "Brussels"
  or
  country = "Germany" and capital = "Berlin"
  or
  country = "France" and capital = "Paris"
}
```

The predicate `isCountry` is the set of one-tuples `{("Belgium"),("Germany"),("France")}`, while `hasCapital` is the set of two-tuples `{("Belgium","Brussels"),("Germany","Berlin"),("France","Paris")}`. The arity of these predicates is one and two, respectively.

In general, all tuples in a predicate have the same number of elements. The **arity** of a predicate is that number of elements, not including a possible `result` variable. For more information, see "Predicates with result."

There are a number of built-in predicates in QL. You can use these in any queries without needing to import any additional modules. In addition to these built-in predicates, you can also define your own:

Defining a predicate

When defining a predicate, you should specify:

1. The keyword `predicate` (for a predicate without result), or the type of the result (for a predicate with result).
2. The name of the predicate. This is an identifier starting with a lowercase letter.
3. The arguments to the predicate, if any, separated by commas. For each argument, specify the argument type and an identifier for the argument variable.
4. The predicate body itself. This is a logical formula enclosed in braces.

Note

> An abstract or external predicate has no body. To define such a predicate, end the predicate definition with a semicolon (`;`) instead.

Predicates without result

These predicate definitions start with the keyword `predicate`. If a value satisfies the logical property in the body, then the predicate holds for that value.

For example:

```
predicate isSmall(int i) {
  i in [1 .. 9]
}
```

If `i` is an integer, then `isSmall(i)` holds if `i` is a positive integer less than 10.

Predicates with result

You can define a predicate with result by replacing the keyword `predicate` with the type of the result. This introduces the special variable `result`.

For example:

```
int getSuccessor(int i) {
  result = i + 1 and
  i in [1 .. 9]
}
```

If `i` is a positive integer less than 10, then the result of the predicate is the successor of `i`.

Note that you can use `result` in the same way as any other argument to the predicate. You can express the relation between `result` and other variables in any way you like. For example, given a predicate `getAParentOf(Person x)` that returns parents of `x`, you can define a "reverse" predicate as follows:

```
Person getAChildOf(Person p) {
  p = getAParentOf(result)
}
```

It is also possible for a predicate to have multiple results (or none at all) for each value of its arguments. For example:

```
string getANeighbor(string country) {
  country = "France" and result = "Belgium"
  or
  country = "France" and result = "Germany"
  or
  country = "Germany" and result = "Austria"
  or
  country = "Germany" and result = "Belgium"
}
```

In this case:

- The predicate call `getANeighbor("Germany")` returns two results: `"Austria"` and `"Belgium"`.

- The predicate call `getANeighbor("Belgium")` returns no results, since `getANeighbor` does not define a `result` for `"Belgium"`.

Recursive predicates

A predicate in QL can be **recursive**. This means that it depends, directly or indirectly, on itself.

For example, you could use recursion to refine the above example. As it stands, the relation defined in `getANeighbor` is not symmetric—it does not capture the fact that if x is a neighbor of y, then y is a neighbor of x. A simple way to capture this is to call this predicate recursively, as shown below:

```
string getANeighbor(string country) {
  country = "France" and result = "Belgium"
  or
  country = "France" and result = "Germany"
  or
  country = "Germany" and result = "Austria"
  or
  country = "Germany" and result = "Belgium"
  or
  country = getANeighbor(result)
}
```

Now `getANeighbor("Belgium")` also returns results, namely `"France"` and `"Germany"`.

For a more general discussion of recursive predicates and queries, see "Recursion."

Kinds of predicates

There are three kinds of predicates, namely non-member predicates, member predicates, and characteristic predicates.

Non-member predicates are defined outside a class, that is, they are not members of any class.

For more information about the other kinds of predicates, see characteristic predicates and member predicates in the "Classes" topic.

Here is an example showing a predicate of each kind:

```
int getSuccessor(int i) {  // 1. Non-member predicate
  result = i + 1 and
  i in [1 .. 9]
}

class FavoriteNumbers extends int {
  FavoriteNumbers() {  // 2. Characteristic predicate
    this = 1 or
    this = 4 or
    this = 9
  }

  string getName() {   // 3. Member predicate for the class `FavoriteNumbers`
    this = 1 and result = "one"
    or
    this = 4 and result = "four"
    or
    this = 9 and result = "nine"
```

```
  }
}
```

You can also annotate each of these predicates. See the list of annotations available for each kind of predicate.

Binding behavior

It must be possible to evaluate a predicate in a finite amount of time, so the set it describes is not usually allowed to be infinite. In other words, a predicate can only contain a finite number of tuples.

The QL compiler reports an error when it can prove that a predicate contains variables that aren't constrained to a finite number of values. For more information, see "Binding."

Here are a few examples of infinite predicates:

```
/*
  Compilation errors:
  ERROR: "i" is not bound to a value.
  ERROR: "result" is not bound to a value.
  ERROR: expression "i * 4" is not bound to a value.
*/
int multiplyBy4(int i) {
  result = i * 4
}

/*
  Compilation errors:
  ERROR: "str" is not bound to a value.
  ERROR: expression "str.length()" is not bound to a value.
*/
predicate shortString(string str) {
  str.length() < 10
}
```

In `multiplyBy4`, the argument `i` is declared as an `int`, which is an infinite type. It is used in the binary operation `*`, which does not bind its operands. `result` is unbound to begin with, and remains unbound since it is used in an equality check with `i * 4`, which is also unbound.

In `shortString`, `str` remains unbound since it is declared with the infinite type `string`, and the built-in function `length()` does not bind it.

Binding sets

Sometimes you may want to define an "infinite predicate" anyway, because you only intend to use it on a restricted set of arguments. In that case, you can specify an explicit binding set using the `bindingset` annotation. This annotation is valid for any kind of predicate.

For example:

```
bindingset[i]
int multiplyBy4(int i) {
  result = i * 4
}

from int i
```

```
  where i in [1 .. 10]
  select multiplyBy4(i)
```

Although `multiplyBy4` is an infinite predicate, the above QL query is legal. It first uses the `bindingset` annotation to state that the predicate `multiplyBy4` will be finite provided that `i` is bound to a finite number of values. Then it uses the predicate in a context where `i` is restricted to to the range `[1 .. 10]`.

It is also possible to state multiple binding sets for a predicate. This can be done by adding multiple binding set annotations, for example:

```
bindingset[x] bindingset[y]
predicate plusOne(int x, int y) {
  x + 1 = y
}

from int x, int y
where y = 42 and plusOne(x, y)
select x, y
```

Multiple binding sets specified this way are independent of each other. The above example means:

- If `x` is bound, then `x` and `y` are bound.
- If `y` is bound, then `x` and `y` are bound.

That is, `bindingset[x] bindingset[y]`, which states that at least one of `x` or `y` must be bound, is different from `bindingset[x, y]`, which states that both `x` and `y` must be bound.

The latter can be useful when you want to declare a predicate with result that takes multiple input arguments. For example, the following predicate takes a string `str` and truncates it to a maximum length of `len` characters:

```
bindingset[str, len]
string truncate(string str, int len) {
  if str.length() > len
  then result = str.prefix(len)
  else result = str
}
```

You can then use this in a select clause, for example:

```
select truncate("hello world", 5)
```

Database predicates

Each database that you query contains tables expressing relations between values. These tables (“database predicates”) are treated in the same way as other predicates in QL.

For example, if a database contains a table for persons, you can write `persons(x, firstName, _, age)` to constrain `x`, `firstName`, and `age` to be the first, second, and fourth columns of rows in that table.

The only difference is that you can’t define database predicates in QL. They are defined by the underlying database. Therefore, the available database predicates vary according to the database that you are querying.