Annotations

An annotation is a string that you can place directly before the declaration of a QL entity or name.

For example, to declare a module `M` as private, you could use:

```
private module M {
    ...
}
```

Note that some annotations act on an entity itself, whilst others act on a particular *name* for the entity:

- Act on an **entity**: `abstract`, `cached`, `external`, `transient`, `override`, `pragma`, `language`, and `bindingset`
- Act on a **name**: `deprecated`, `library`, `private`, `final`, and `query`

For example, if you annotate an entity with `private`, then only that particular name is private. You could still access that entity under a different name (using an alias). On the other hand, if you annotate an entity with `cached`, then the entity itself is cached.

Here is an explicit example:

```
module M {
  private int foo() { result = 1 }
  predicate bar = foo/0;
}
```

In this case, the query `select M::foo()` gives a compiler error, since the name `foo` is private. The query `select M::bar()` is valid (giving the result `1`), since the name `bar` is visible and it is an alias of the predicate `foo`.

You could apply `cached` to `foo`, but not `bar`, since `foo` is the declaration of the entity.

Overview of annotations

This section describes what the different annotations do, and when you can use them. You can also find a summary table in the Annotations section of the QL language specification.

`abstract`

Available for: classes, member predicates

The `abstract` annotation is used to define an abstract entity.

For information about **abstract classes**, see “Classes.”

Abstract predicates are member predicates that have no body. They can be defined on any class, and should be overridden in non-abstract subtypes.

Here is an example that uses abstract predicates. A common pattern when writing data flow analysis in QL is to define a configuration class. Such a configuration must describe, among other things, the sources of data that it tracks. A supertype of all such configurations might look like this:

```
abstract class Configuration extends string {
  ...
  /** Holds if `source` is a relevant data flow source. */
  abstract predicate isSource(Node source);
  ...
}
```

You could then define subtypes of `Configuration`, which inherit the predicate `isSource`, to describe specific configurations. Any non-abstract subtypes must override it (directly or indirectly) to describe what sources of data they each track.

In other words, all non-abstract classes that extend `Configuration` must override `isSource` in their own body, or they must inherit from another class that overrides `isSource`:

```
class ConfigA extends Configuration {
  ...
  // provides a concrete definition of `isSource`
  override predicate isSource(Node source) { ... }
}
class ConfigB extends ConfigA {
  ...
  // doesn't need to override `isSource`, because it inherits it from ConfigA
}
```

cached

Available for: classes, algebraic datatypes, characteristic predicates, member predicates, non-member predicates, modules

The `cached` annotation indicates that an entity should be evaluated in its entirety and stored in the evaluation cache. All later references to this entity will use the already-computed data. This affects references from other queries, as well as from the current query.

For example, it can be helpful to cache a predicate that takes a long time to evaluate, and is reused in many places.

You should use `cached` carefully, since it may have unintended consequences. For example, cached predicates may use up a lot of storage space, and may prevent the QL compiler from optimizing a predicate based on the context at each place it is used. However, this may be a reasonable tradeoff for only having to compute the predicate once.

If you annotate a class or module with `cached`, then all non-private entities in its body must also be annotated with `cached`, otherwise a compiler error is reported.

deprecated

Available for: classes, algebraic datatypes, member predicates, non-member predicates, fields, modules, aliases

The `deprecated` annotation is applied to names that are outdated and scheduled for removal in a future release of QL. If any of your QL files use deprecated names, you should consider rewriting them to use newer alternatives. Typically, deprecated names have a QLDoc comment that tells users which updated element they should use instead.

For example, the name `DataFlowNode` is deprecated and has the following QLDoc comment:

```
/**
 * DEPRECATED: Use `DataFlow::Node` instead.
 *
 * An expression or function/class declaration,
 * viewed as a node in a data flow graph.
 */
deprecated class DataFlowNode extends @dataflownode {
  ...
}
```

This QLDoc comment appears when you use the name `DataFlowNode` in a QL editor.

`external`

Available for: non-member predicates

The `external` annotation is used on predicates, to define an external “template” predicate. This is similar to a database predicate.

`transient`

Available for: non-member predicates

The `transient` annotation is applied to non-member predicates that are also annotated with `external`, to indicate that they should not be cached to disk during evaluation. Note, if you attempt to apply `transient` without `external`, the compiler will report an error.

`final`

Available for: classes, type aliases, member predicates, fields

The `final` annotation is applied to names that can’t be overridden or extended. In other words, a final class or a final type alias can’t act as a base type for any other types, and a final predicate or field can’t be overridden in a subclass.

This is useful if you don’t want subclasses to change the meaning of a particular entity.

For example, the predicate `hasName(string name)` holds if an element has the name `name`. It uses the predicate `getName()` to check this, and it wouldn’t make sense for a subclass to change this definition. In this case, `hasName` should be final:

```
class Element ... {
  string getName() { result = ... }
```

```
    final predicate hasName(string name) { name = this.getName() }
  }
```

library

Available for: classes

> **Important**
>
> This annotation is deprecated. Instead of annotating a name with `library`, put it in a private (or privately imported) module.

The `library` annotation is applied to names that you can only refer to from within a `.qll` file. If you try to refer to that name from a file that does not have the `.qll` extension, then the QL compiler returns an error.

override

Available for: member predicates, fields

The `override` annotation is used to indicate that a definition overrides a member predicate or field from a base type.

If you override a predicate or field without annotating it, then the QL compiler gives a warning.

private

Available for: classes, algebraic datatypes, member predicates, non-member predicates, imports, fields, modules, aliases

The `private` annotation is used to prevent names from being exported.

If a name has the annotation `private`, or if it is accessed through an import statement annotated with `private`, then you can only refer to that name from within the current module's namespace.

query

Available for: non-member predicates, aliases

The `query` annotation is used to turn a predicate (or a predicate alias) into a query. This means that it is part of the output of the QL program.

Compiler pragmas

The following compiler pragmas affect the compilation and optimization of queries. You should avoid using these annotations unless you experience significant performance issues.

Before adding pragmas to your code, contact GitHub to describe the performance problems. That way we can suggest the best solution for your problem, and take it into account when improving the QL optimizer.

Inlining

For simple predicates, the QL optimizer sometimes replaces a call to a predicate with the predicate body itself. This is known as **inlining**.

For example, suppose you have a definition `predicate one(int i) { i = 1 }` and a call to that predicate `... one(y) ...`. The QL optimizer may inline the predicate to `... y = 1 ...`.

You can use the following compiler pragma annotations to control the way the QL optimizer inlines predicates.

`pragma[inline]`

Available for: characteristic predicates, member predicates, non-member predicates

The `pragma[inline]` annotation tells the QL optimizer to always inline the annotated predicate into the places where it is called. This can be useful when a predicate body is very expensive to compute entirely, as it ensures that the predicate is evaluated with the other contextual information at the places where it is called.

`pragma[inline_late]`

Available for: non-member predicates

The `pragma[inline_late]` annotation must be used in conjunction with a `bindingset[...]` pragma. Together, they tell the QL optimiser to use the specified binding set for assessing join orders both in the body of the annotated predicate and at call sites and to inline the body into call sites after join ordering. This can be useful to prevent the optimiser from choosing a sub-optimal join order.

For instance, in the example below, the `pragma[inline_late]` and `bindingset[x]` annotations specifiy that calls to `p` should be join ordered in a context where `x` is already bound. This forces the join orderer to order `q(x)` before `p(x)`, which is more computationally efficient than ordering `p(x)` before `q(x)`.

```
bindingset[x]
pragma[inline_late]
predicate p(int x) { x in [0..100000000] }

predicate q(int x) { x in [0..10000] }

from int x
where p(x) and q(x)
select x
```

`pragma[noinline]`

Available for: characteristic predicates, member predicates, non-member predicates

The `pragma[noinline]` annotation is used to prevent a predicate from being inlined into the place where it is called. In practice, this annotation is useful when you've already grouped certain variables together in a "helper" predicate, to ensure that the relation is evaluated in one piece. This can help to improve performance. The QL optimizer's inlining may undo the work of the helper predicate, so it's a good idea to annotate it with `pragma[noinline]`.

`pragma[nomagic]`

Available for: characteristic predicates, member predicates, non-member predicates

The `pragma[nomagic]` annotation is used to prevent the QL optimizer from performing the "magic sets" optimization on a predicate.

This kind of optimization involves taking information from the context of a predicate call and pushing it into the body of a predicate. This is usually beneficial, so you shouldn't use the `pragma[nomagic]` annotation unless recommended to do so by GitHub.

Note that `nomagic` implies `noinline`.

`pragma[noopt]`

Available for: characteristic predicates, member predicates, non-member predicates

The `pragma[noopt]` annotation is used to prevent the QL optimizer from optimizing a predicate, except when it's absolutely necessary for compilation and evaluation to work.

This is rarely necessary and you should not use the `pragma[noopt]` annotation unless recommended to do so by GitHub, for example, to help resolve performance issues.

When you use this annotation, be aware of the following issues:

1. The QL optimizer automatically orders the conjuncts of a complex formula in an efficient way. In a `noopt` predicate, the conjuncts are evaluated in exactly the order that you write them.

2. The QL optimizer automatically creates intermediary conjuncts to "translate" certain formulas into a conjunction of simpler formulas. In a `noopt` predicate, you must write these conjunctions explicitly. In particular, you can't chain predicate calls or call predicates on a cast. You must write them as multiple conjuncts and explicitly order them.

   For example, suppose you have the following definitions:

   ```
   class Small extends int {
     Small() { this in [1 .. 10] }
     Small getSucc() { result = this + 1}
   }

   predicate p(int i) {
     i.(Small).getSucc() = 2
   }

   predicate q(Small s) {
     s.getSucc().getSucc() = 3
   }
   ```

   If you add `noopt` pragmas, you must rewrite the predicates. For example:

```
pragma[noopt]
predicate p(int i) {
  exists(Small s | s = i and s.getSucc() = 2)
}

pragma[noopt]
predicate q(Small s) {
  exists(Small succ |
    succ = s.getSucc() and
    succ.getSucc() = 3
  )
}
```

`pragma[only_bind_out]`

Available for: expressions

The `pragma[only_bind_out]` annotation lets you specify the direction in which the QL compiler should bind expressions. This can be useful to improve performance in rare cases where the QL optimizer orders parts of the QL program in an inefficient way.

For example, `x = pragma[only_bind_out](y)` is semantically equivalent to `x = y`, but has different binding behavior. `x = y` binds `x` from `y` and vice versa, while `x = pragma[only_bind_out](y)` only binds `x` from `y`.

For more information, see "Binding."

`pragma[only_bind_into]`

Available for: expressions

The `pragma[only_bind_into]` annotation lets you specify the direction in which the QL compiler should bind expressions. This can be useful to improve performance in rare cases where the QL optimizer orders parts of the QL program in an inefficient way.

For example, `x = pragma[only_bind_into](y)` is semantically equivalent to `x = y`, but has different binding behavior. `x = y` binds `x` from `y` and vice versa, while `x = pragma[only_bind_into](y)` only binds `y` from `x`.

For more information, see "Binding."

`pragma[assume_small_delta]`

Available for: characteristic predicates, member predicates, non-member predicates

The `pragma[assume_small_delta]` annotation changes the compilation of the annotated recursive predicate. If the compiler normally generates the join orders `order_<1>`, `order_<2>`, `order_<3>`, and `standard_order`, applying this annotation makes `standard_order` the same as `order_<3>` and removes the (now redundant) `order_<3>` join order.

Language pragmas

Available for: classes, characteristic predicates, member predicates, non-member predicates

`language[monotonicAggregates]`

This annotation allows you to use **monotonic aggregates** instead of the standard QL aggregates.

For more information, see "Monotonic aggregates."

Binding sets

Available for: classes, characteristic predicates, member predicates, non-member predicates

`bindingset[...]`

You can use this annotation to explicitly state the binding sets for a predicate or class. A binding set is a subset of a predicate's or class body's arguments such that, if those arguments are constrained to a finite set of values, then the predicate or class itself is finite (that is, it evaluates to a finite set of tuples).

The `bindingset` annotation takes a comma-separated list of variables.

- When you annotate a predicate, each variable must be an argument of the predicate, possibly including `this` (for characteristic predicates and member predicates) and `result` (for predicates that return a result). For more information, see "Binding behavior."
- When you annotate a class, each variable must be `this` or a field in the class.