QL language reference

Release 1.24

Aug 03, 2021

CONTENTS

Learn all about QL, the powerful query language that underlies the code scanning tool CodeQL.

CHAPTER

ONE

ABOUT THE QL LANGUAGE

QL is the powerful query language that underlies CodeQL, which is used to analyze code.

1.1 About query languages and databases

This section is aimed at users with a background in general purpose programming as well as in databases. For a basic introduction and information on how to get started, see Learning CodeQL.

QL is a declarative, object-oriented query language that is optimized to enable efficient analysis of hierarchical data structures, in particular, databases representing software artifacts.

A database is an organized collection of data. The most commonly used database model is a relational model which stores data in tables and SQL (Structured Query Language) is the most commonly used query language for relational databases.

The purpose of a query language is to provide a programming platform where you can ask questions about information stored in a database. A database management system manages the storage and administration of data and provides the querying mechanism. A query typically refers to the relevant database entities and specifies various conditions (called predicates) that must be satisfied by the results. Query evaluation involves checking these predicates and generating the results. Some of the desirable properties of a good query language and its implementation include:

- Declarative specifications - a declarative specification describes properties that the result must satisfy, rather than providing the procedure to compute the result. In the context of database query languages, declarative specifications abstract away the details of the underlying database management system and query processing techniques. This greatly simplifies query writing.
- Expressiveness - a powerful query language allows you to write complex queries. This makes the language widely applicable.
- Efficient execution - queries can be complex and databases can be very large, so it is crucial for a query language implementation to process and execute queries efficiently.

1.2 Properties of QL

The syntax of QL is similar to SQL, but the semantics of QL are based on Datalog, a declarative logic programming language often used as a query language. This makes QL primarily a logic language, and all operations in QL are logical operations. Furthermore, QL inherits recursive predicates from Datalog, and adds support for aggregates, making even complex queries concise and simple. For example, consider a database containing parent-child relationships for people. If we want to find the number of descendants of a person, typically we would:

1. Find a descendant of the given person, that is, a child or a descendant of a child.
2. Count the number of descendants found using the previous step.

When you write this process in QL, it closely resembles the above structure. Notice that we used recursion to find all descendants of the given person, and an aggregate to count the number of descendants. Translating these steps into the final query without adding any procedural details is possible due to the declarative nature of the language. The QL code would look something like this:

```
Person getADescendant(Person p) {
  result = p.getAChild() or
```

(continues on next page)

(continued from previous page)

```
  result = getADescendant(p.getAChild())
}

int getNumberOfDescendants(Person p) {
  result = count(getADescendant(p))
}
```

For more information about the important concepts and syntactic constructs of QL, see the individual reference topics such as *Expressions* and *Recursion*. The explanations and examples help you understand how the language works, and how to write more advanced QL code.

For formal specifications of the QL language and QLDoc comments, see the *QL language specification* and *QLDoc comment specification*.

1.3 QL and object orientation

Object orientation is an important feature of QL. The benefits of object orientation are well known – it increases modularity, enables information hiding, and allows code reuse. QL offers all these benefits without compromising on its logical foundation. This is achieved by defining a simple object model where classes are modeled as predicates and inheritance as implication. The libraries made available for all supported languages make extensive use of classes and inheritance.

1.4 QL and general purpose programming languages

Here are a few prominent conceptual and functional differences between general purpose programming languages and QL:

- QL does not have any imperative features such as assignments to variables or file system operations.

- QL operates on sets of tuples and a query can be viewed as a complex sequence of set operations that defines the result of the query.
- QLs set-based semantics makes it very natural to process collections of values without having to worry about efficiently storing, indexing and traversing them.
- In object oriented programming languages, instantiating a class involves creating an object by allocating physical memory to hold the state of that instance of the class. In QL, classes are just logical properties describing sets of already existing values.

1.5 Further reading

Academic references also provide an overview of QL and its semantics. Other useful references on database query languages and Datalog:

- Database theory: Query languages
- Logic Programming and Databases book - Amazon page
- Foundations of Databases
- Datalog

CHAPTER

TWO

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 (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:

2.1 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.

2.1.1 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.

2.1.2 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"`.

2.2 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 symmetricit 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*.

2.3 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.

2.4 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 arent constrained to a finite number of values. See *Binding* for more information.

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.
*/
int multiplyBy4(int i) {
  result = i * 4
}

/*
  Compilation error:
  ERROR: "str" 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.

2.4.1 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 exan

- 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)
```

2.5 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 cant 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.

CHAPTER

THREE

QUERIES

Queries are the output of a QL program. They evaluate to sets of results.

There are two kinds of queries. For a given *query module*, the queries in that module a

- The *select clause*, if any, defined in that module.
- Any *query predicates* in that modules predicate *namespace*. That is, they can be defined in the module itself, or imported from a different module.

We often also refer to the whole QL program as a query.

3.1 Select clauses

When writing a query module, you can include a **select clause** (usually at the end of the file) of the following form:

```
from /* ... variable declarations ... */
where /* ... logical formula ... */
select /* ... expressions ... */
```

The `from` and `where` parts are optional.

Apart from the expressions described in *Expressions*, you can also include:

- The `as` keyword, followed by a name. This gives a label to a column of results, and allows you to use them in subsequent select expressions.

- The `order by` keywords, followed by the name of a result column, and optionally the keyword `asc` or `desc`. This determines the order in which to display the results.

For example:

```
from int x, int y
where x = 3 and y in [0 .. 2]
select x, y, x * y as product, "product: " + product
```

This select clause returns the following results:

x	y	product	
3	0	0	product: 0
3	1	3	product: 3
3	2	6	product: 6

You could also add `order by y desc` at the end of the select clause. Now the results are ordered according to the values in the `y` column, in descending order:

x	y	product	
3	2	6	product: 6
3	1	3	product: 3
3	0	0	product: 0

3.2 Query predicates

A query predicate is a *non-member predicate* with a `query` annotation. It returns all the tuples that the predicate evaluates to.

For example:

```
query int getProduct(int x, int y) {
  x = 3 and
  y in [0 .. 2] and
```

(continues on next page)

(continued from previous page)

```
  result = x * y
}
```

This predicate returns the following results:

x	y	result
3	0	0
3	1	3
3	2	6

A benefit of writing a query predicate instead of a select clause is that you can call the predicate in other parts of the code too. For example, you can call `getProduct` inside the body of a *class*:

```
class MultipleOfThree extends int {
  MultipleOfThree() { this = getProduct(_, _) }
}
```

In contrast, the select clause is like an anonymous predicate, so you cant call it later.

It can also be helpful to add a `query` annotation to a predicate while you debug code. That way you can explicitly see the set of tuples that the predicate evaluates to.

CHAPTER

FOUR

TYPES

QL is a statically typed language, so each variable must have a declared type.

A type is a set of values. For example, the type `int` is the set of integers. Note that a value can belong to more than one of these sets, which means that it can have more than one type.

The kinds of types in QL are *primitive types*, *classes*, *character types*, *class domain types*, *algebraic datatypes*, *type unions*, and *database types*.

4.1 Primitive types

These types are built in to QL and are always available in the global *namespace*, independent of the database that you are querying.

1. **boolean**: This type contains the values `true` and `false`.
2. **float**: This type contains 64-bit floating point numbers, such as `6.28` and `-0.618`.
3. **int**: This type contains 32-bit twos complement integers, such as `-1` and `42`.
4. **string**: This type contains finite strings of 16-bit characters.
5. **date**: This type contains dates (and optionally times).

QL has a range of built-in operations defined on primitive types. These are available by using dispatch on expressions of the appropriate type. For example, `1.toString()` is the string representation of the integer constant `1`. For a full list of built-in operations available in QL, see the section on built-ins in the QL language specification.

4.2 Classes

You can define your own types in QL. One way to do this is to define a **class**.

Classes provide an easy way to reuse and structure code. For example, you can:

- Group together related values.
- Define *member predicates* on those values.
- Define subclasses that *override member predicates*.

A class in QL doesnt create a new object, it just represents a logical property. A value is in a particular class if it satisfies that logical property.

4.2.1 Defining a class

To define a class, you write:

1. The keyword `class`.
2. The name of the class. This is an identifier starting with an uppercase letter.
3. The types to extend.
4. The *body of the class*, enclosed in braces.

For example:

```
class OneTwoThree extends int {
  OneTwoThree() { // characteristic predicate
    this = 1 or this = 2 or this = 3
  }
```

(continues on next page)

(continued from previous page)

```
  string getAString() { // member predicate
    result = "One, two or three: " + this.toString()
  }

  predicate isEven() { // member predicate
    this = 2
  }
}
```

This defines a class `OneTwoThree`, which contains the values `1`, `2`, and `3`. The *characteristic predicate* captures the logical property of being one of the integers 1, 2, or 3.

`OneTwoThree` extends `int`, that is, it is a subtype of `int`. A class in QL must always extend at least one existing type. Those types are called the **base types** of the class. The values of a class are contained within the intersection of the base types (that is, they are in the *class domain type*). A class inherits all member predicates from its base types.

A class can extend multiple types. See *Multiple inheritance* below.

To be valid, a class:

- Must not extend itself.
- Must not extend a *final* class.
- Must not extend types that are incompatible. (See *Type compatibility*.)

You can also annotate a class. See the list of *annotations* available for classes.

4.2.2 Class bodies

The body of a class can contain:

- A *characteristic predicate* declaration.
- Any number of *member predicate* declarations.

- Any number of *field* declarations.

When you define a class, that class also inherits all non-*private* member predicates and fields from its supertypes. You can *override* those predicates and fields to give them a more specific definition.

Characteristic predicates

These are *predicates* defined inside the body of a class. They are logical properties that use the variable `this` to restrict the possible values in the class.

Member predicates

These are *predicates* that only apply to members of a particular class. You can *call* a member predicate on a value. For example, you can use the member predicate from the *above* class:

```
1.(OneTwoThree).getAString()
```

This call returns the result `"One, two or three:  1"`.

The expression `(OneTwoThree)` is a *cast*. It ensures that `1` has type `OneTwoThree` instead of just `int`. Therefore, it has access to the member predicate `getAString()`.

Member predicates are especially useful because you can chain them together. For example, you can use `toUpperCase()`, a built-in function defined for `string`:

```
1.(OneTwoThree).getAString().toUpperCase()
```

This call returns `"ONE, TWO OR THREE: 1"`.

Note: Characteristic predicates and member predicates often use the variable `this`. This variable always refers to a member of the classin this case a value belonging to the class `OneTwoThree`. In the *characteristic predicate*, the variable `this` constrains the values that are in the class. In a *member predicate*, `this` acts in the same way as any other argument to the predicate.

Fields

These are variables declared in the body of a class. A class can have any number of field declarations (that is, variable declarations) within its body. You can use these variables in predicate declarations inside the class. Much like the *variable* `this`, fields must be constrained in the *characteristic predicate*.

For example:

```
class SmallInt extends int {
  SmallInt() { this = [1 .. 10] }
}

class DivisibleInt extends SmallInt {
  SmallInt divisor;   // declaration of the field `divisor`
  DivisibleInt() { this % divisor = 0 }

  SmallInt getADivisor() { result = divisor }
}

from DivisibleInt i
select i, i.getADivisor()
```

In this example, the declaration `SmallInt divisor` introduces a field `divisor`, constrains it in the characteristic predicate, and then uses it in the declaration of the member predicate `getADivisor`. This is similar to introducing variables in a *select clause* by declaring them in the `from` part.

You can also annotate predicates and fields. See the list of *annotations* that are available.

4.2.3 Concrete classes

The classes in the above examples are all **concrete** classes. They are defined by restricting the values in a larger type. The values in a concrete class are precisely those values in the intersection of the base types that also satisfy the *characteristic predicate* of the class.

4.2.4 Abstract classes

A class *annotated* with `abstract`, known as an **abstract** class, is also a restriction of the values in a larger type. However, an abstract class is defined as the union of its subclasses. In particular, for a value to be in an abstract class, it must satisfy the characteristic predicate of the class itself **and** the characteristic predicate of a subclass.

An abstract class is useful if you want to group multiple existing classes together under a common name. You can then define member predicates on all those classes. You can also extend predefined abstract classes: for example, if you import a library that contains an abstract class, you can add more subclasses to it.

Example

If you are writing a security query, you may be interested in identifying all expressions that can be interpreted as SQL queries. You can use the following abstract class to describe these expressions:

```
abstract class SqlExpr extends Expr {
  ...
}
```

Now define various subclassesone for each kind of database management system. For example, you can define a subclass `class PostgresSqlExpr extends SqlExpr`, which contains expressions passed to some Postgres API that performs a database query. You can define similar subclasses for MySQL and other database management systems.

The abstract class `SqlExpr` refers to all of those different expressions. If you want to add support for another database system later on, you can simply add a new subclass to `SqlExpr`; there is no need to update the queries that rely on it.

> Important
>
> You must take care when you add a new subclass to an existing abstract class. Adding a subclass is not an isolated change, it also extends the abstract class since that is a union of its subclasses.

4.2.5 Overriding member predicates

If a class inherits a member predicate from a supertype, you can **override** the inherited definition. You do this by defining a member predicate with the same name and arity as the inherited predicate, and by adding the `override` *annotation*. This is useful if you want to refine the predicate to give a more specific result for the values in the subclass.

For example, extending the class from the *first example*:

```
class OneTwo extends OneTwoThree {
  OneTwo() {
    this = 1 or this = 2
  }

  override string getAString() {
    result = "One or two: " + this.toString()
  }
}
```

The member predicate `getAString()` overrides the original definition of `getAString()` from `OneTwoThree`.

Now, consider the following query:

```
from OneTwoThree o
select o, o.getAString()
```

The query uses the most specific definition(s) of the predicate `getAString()`, so the results look like this:

o	getAString() result
1	One or two: 1
2	One or two: 2
3	One, two or three: 3

In QL, unlike other object-oriented languages, different subtypes of the same

types dont need to be disjoint. For example, you could define another subclass of `OneTwoThree`, which overlaps with `OneTwo`:

```
class TwoThree extends OneTwoThree {
  TwoThree() {
    this = 2 or this = 3
  }

  override string getAString() {
    result = "Two or three: " + this.toString()
  }
}
```

Now the value 2 is included in both class types `OneTwo` and `TwoThree`. Both of these classes override the original definition of `getAString()`. There are two new most specific definitions, so running the above query gives the following results:

o	getAString() result
1	One or two: 1
2	One or two: 2
2	Two or three: 2
3	Two or three: 3

4.2.6 Multiple inheritance

A class can extend multiple types. In that case, it inherits from all those types.

For example, using the definitions from the above section:

```
class Two extends OneTwo, TwoThree {}
```

Any value in the class `Two` must satisfy the logical property represented by `OneTwo`, **and** the logical property represented by `TwoThree`. Here the class `Two` contains one value, namely 2.

It inherits member predicates from `OneTwo` and `TwoThree`. It also (indirectly)

inherits from `OneTwoThree` and `int`.

Note: If a subclass inherits multiple definitions for the same predicate name, then it must *override* those definitions to avoid ambiguity. *Super expressions* are often useful in this situation.

4.3 Character types and class domain types

You cant refer to these types directly, but each class in QL implicitly defines a character type and a class domain type. (These are rather more subtle concepts and dont appear very often in practical query writing.)

The **character type** of a QL class is the set of values satisfying the *characteristic predicate* of the class. It is a subset of the domain type. For concrete classes, a value belongs to the class if, and only if, it is in the character type. For *abstract classes*, a value must also belong to at least one of the subclasses, in addition to being in the character type.

The **domain type** of a QL class is the intersection of the character types of all its supertypes, that is, a value belongs to the domain type if it belongs to every supertype. It occurs as the type of `this` in the characteristic predicate of a class.

4.4 Algebraic datatypes

Note: The syntax for algebraic datatypes is considered experimental and is subject to change. However, they appear in the standard QL libraries so the following sections should help you understand those examples.

An algebraic datatype is another form of user-defined type, declared with the keyword `newtype`.

Algebraic datatypes are used for creating new values that are neither primitive values nor entities from the database. One example is to model flow nodes when

analyzing data flow through a program.

An algebraic datatype consists of a number of mutually disjoint *branches*, that each define a branch type. The algebraic datatype itself is the union of all the branch types. A branch can have arguments and a body. A new value of the branch type is produced for each set of values that satisfy the argument types and the body.

A benefit of this is that each branch can have a different structure. For example, if you want to define an option type that either holds a value (such as a `Call`) or is empty, you could write this as follows:

```
newtype OptionCall = SomeCall(Call c) or NoCall()
```

This means that for every `Call` in the program, a distinct `SomeCall` value is produced. It also means that a unique `NoCall` value is produced.

4.4.1 Defining an algebraic datatype

To define an algebraic datatype, use the following general syntax:

```
newtype <TypeName> = <branches>
```

The branch definitions have the following form:

```
<BranchName>(<arguments>) { <body> }
```

- The type name and the branch names must be identifiers starting with an uppercase letter. Conventionally, they start with `T`.
- The different branches of an algebraic datatype are separated by `or`.
- The arguments to a branch, if any, are *variable declarations* separated by commas.
- The body of a branch is a *predicate* body. You can omit the branch body, in which case it defaults to `any()`. Note that branch bodies are evaluated fully, so they must be finite. They should be kept small for good performance.

For example, the following algebraic datatype has three branches:

```
newtype T =
  Type1(A a, B b) { body(a, b) }
  or
  Type2(C c)
  or
  Type3()
```

4.4.2 Standard pattern for using algebraic datatypes

Algebraic datatypes are different from *classes*. In particular, algebraic datatypes dont have a `toString()` member predicate, so you cant use them in a *select clause*.

Classes are often used to extend algebraic datatypes (and to provide a `toString()` predicate). In the standard QL language libraries, this is usually done as follows:

- Define a class `A` that extends the algebraic datatype and optionally declares *abstract* predicates.
- For each branch type, define a class `B` that extends both `A` and the branch type, and provide a definition for any abstract predicates from `A`.
- Annotate the algebraic datatype with *private*, and leave the classes public.

For example, the following code snippet from the CodeQL data-flow library for C# defines classes for dealing with tainted or untainted values. In this case, it doesnt make sense for `TaintType` to extend a database type. It is part of the taint analysis, not the underlying program, so its helpful to extend a new type (namely `TTaintType`):

```
private newtype TTaintType =
  TExactValue()
  or
  TTaintedValue()
```

(continues on next page)

(continued from previous page)

```
/** Describes how data is tainted. */
class TaintType extends TTaintType {
  string toString() {
    this = TExactValue() and result = "exact"
    or
    this = TTaintedValue() and result = "tainted"
  }
}

/** A taint type where the data is untainted. */
class Untainted extends TaintType, TExactValue {
}

/** A taint type where the data is tainted. */
class Tainted extends TaintType, TTaintedValue {
}
```

4.5 Type unions

Type unions are user-defined types that are declared with the keyword `class`. The syntax resembles *type aliases*, but with two or more type expressions on the right-hand side.

Type unions are used for creating restricted subsets of an existing *algebraic datatype*, by explicitly selecting a subset of the branches of that datatype and binding them to a new type. Type unions of *database types* are also supported.

You can use a type union to give a name to a subset of the branches from an algebraic datatype. In some cases, using the type union over the whole algebraic datatype can avoid spurious *recursion* in predicates. For example, the following construction is legal:

```
newtype InitialValueSource =
  ExplicitInitialization(VarDecl v) { exists(v.getInitializer()) } or
  ParameterPassing(Call c, int pos) { exists(c.getParameter(pos)) } or
```

(continues on next page)

(continued from previous page)

```
  UnknownInitialGarbage(VarDecl v) { not exists(DefiniteInitialization␣
 ↪di | v = target(di)) }

class DefiniteInitialization = ParameterPassing or␣
 ↪ExplicitInitialization;

VarDecl target(DefiniteInitialization di) {
  di = ExplicitInitialization(result) or
  exists(Call c, int pos | di = ParameterPassing(c, pos) and
                           result = c.getCallee().getFormalArg(pos))
}
```

However, a similar implementation that restricts `InitialValueSource` in a class extension is not valid. If we had implemented `DefiniteInitialization` as a class extension instead, it would trigger a type test for `InitialValueSource`. This results in an illegal recursion `DefiniteInitialization -> InitialValueSource -> UnknownInitialGarbage -> ňDefiniteInitialization` since `UnknownInitialGarbage` relies on `DefiniteInitialization`:

```
// THIS WON'T WORK: The implicit type check for InitialValueSource␣
 ↪involves an illegal recursion
// DefiniteInitialization -> InitialValueSource ->␣
 ↪UnknownInitialGarbage -> ňDefiniteInitialization!
class DefiniteInitialization extends InitialValueSource {
  DefiniteInitialization() {
    this instanceof ParameterPassing or this instanceof␣
 ↪ExplicitInitialization
  }
  // ...
}
```

Type unions are supported from release 2.2.0 of the CodeQL CLI.

4.6 Database types

Database types are defined in the database schema. This means that they depend on the database that you are querying, and vary according to the data you are analyzing.

For example, if you are querying a CodeQL database for a Java project, the database types may include `@ifstmt`, representing an if statement in the Java code, and `@variable`, representing a variable.

4.7 Type compatibility

Not all types are compatible. For example, `4 < "five"` doesnt make sense, since you cant compare an `int` to a `string`.

To decide when types are compatible, there are a number of different type universes in QL.

The universes in QL are:

- One for each primitive type (except `int` and `float`, which are in the same universe of numbers).
- One for each database type.
- One for each branch of an algebraic datatype.

For example, when defining a *class* this leads to the following restrictions:

- A class cant extend multiple primitive types.
- A class cant extend multiple different database types.
- A class cant extend multiple different branches of an algebraic datatype.

CHAPTER

FIVE

MODULES

Modules provide a way of organizing QL code by grouping together related types, predicates, and other modules.

You can import modules into other files, which avoids duplication, and helps structure your code into more manageable pieces.

5.1 Defining a module

There are various ways to define moduleshere is an example of the simplest way, declaring an *explicit module* named `Example` containing a class `OneTwoThree`:

```
module Example {
  class OneTwoThree extends int {
    OneTwoThree() {
      this = 1 or this = 2 or this = 3
    }
  }
}
```

The name of a module can be any identifier that starts with an uppercase or lowercase letter.

`.ql` or `.qll` files also implicitly define modules. Read more about the different *kinds of modules* below.

You can also annotate a module. See the list of *annotations* available for modules.

Note that you can only annotate *explicit modules*. File modules cannot be annotated.

5.2 Kinds of modules

5.2.1 File modules

Each query file (extension `.ql`) and library file (extension `.qll`) implicitly defines a module. The module has the same name as the file, but any spaces in the file name are replaced by underscores (`_`). The contents of the file form the *body of the module*.

Library modules

A library module is defined by a `.qll` file. It can contain any of the elements listed in *Module bodies* below, apart from select clauses.

For example, consider the following QL library:

OneTwoThreeLib.qll

```
class OneTwoThree extends int {
  OneTwoThree() {
    this = 1 or this = 2 or this = 3
  }
}
```

This file defines a library module named `OneTwoThreeLib`. The body of this module defines the class `OneTwoThree`.

Query modules

A query module is defined by a `.ql` file. It can contain any of the elements listed in *Module bodies* below.

Query modules are slightly different from other modules:

- A query module cant be imported.

- A query module must have at least one query in its *namespace*. This is usually a *select clause*, but can also be a *query predicate*.

For example:

OneTwoQuery.ql

```
import OneTwoThreeLib

from OneTwoThree ott
where ott = 1 or ott = 2
select ott
```

This file defines a query module named `OneTwoQuery`. The body of this module consists of an *import statement* and a *select clause*.

5.2.2 Explicit modules

You can also define a module within another module. This is an explicit module definition.

An explicit module is defined with the keyword `module` followed by the module name, and then the module body enclosed in braces. It can contain any of the elements listed in *Module bodies* below, apart from select clauses.

For example, you could add the following QL snippet to the library file **OneTwoThreeLib.qll** defined *above*:

```
...
module M {
  class OneTwo extends OneTwoThree {
    OneTwo() {
      this = 1 or this = 2
    }
  }
}
```

This defines an explicit module named `M`. The body of this module defines the class `OneTwo`.

5.3 Module bodies

The body of a module is the code inside the module definition, for example the class `OneTwo` in the *explicit module* `M`.

In general, the body of a module can contain the following constructs:

- *Import statements*
- *Predicates*
- *Types* (including user-defined *classes*)
- *Aliases*
- *Explicit modules*
- *Select clauses* (only available in a *query module*)

5.4 Importing modules

The main benefit of storing code in a module is that you can reuse it in other modules. To access the contents of an external module, you can import the module using an *import statement*.

When you import a module this brings all the names in its namespace, apart from *private* names, into the *namespace* of the current module.

5.4.1 Import statements

Import statements are used for importing modules. They are of the form:

```
import <module_expression1> as <name>
import <module_expression2>
```

Import statements are usually listed at the beginning of the module. Each import statement imports one module. You can import multiple modules by including multiple import statements (one for each module you want to import). An import statement can also be *annotated* with `private`.

You can import a module under a different name using the `as` keyword, for example `import javascript as js`.

The `<module_expression>` itself can be a module name, a selection, or a qualified reference. See *Name resolution* for more details.

For information about how import statements are looked up, see Module resolution in the QL language specification.

CHAPTER

SIX

ALIASES

An alias is an alternative name for an existing QL entity.

Once youve defined an alias, you can use that new name to refer to the entity in the current modules *namespace*.

6.1 Defining an alias

You can define an alias in the body of any *module*. To do this, you should specify:

1. The keyword `module`, `class`, or `predicate` to define an alias for a *module*, *type*, or *non-member predicate* respectively.
2. The name of the alias. This should be a valid name for that kind of entity. For example, a valid predicate alias starts with a lowercase letter.
3. A reference to the QL entity. This includes the original name of the entity and, for predicates, the arity of the predicate.

You can also annotate an alias. See the list of *annotations* available for aliases.

Note that these annotations apply to the name introduced by the alias (and not the underlying QL entity itself). For example, an alias can have different visibility to the name that it aliases.

6.1.1 Module aliases

Use the following syntax to define an alias for a *module*:

```
module ModAlias = ModuleName;
```

For example, if you create a new module `NewVersion` that is an updated version of `OldVersion`, you could deprecate the name `OldVersion` as follows:

```
deprecated module OldVersion = NewVersion;
```

That way both names resolve to the same module, but if you use the name `OldVersion`, a deprecation warning is displayed.

6.1.2 Type aliases

Use the following syntax to define an alias for a *type*:

```
class TypeAlias = TypeName;
```

Note that `class` is just a keyword. You can define an alias for any typenamely, *primitive types*, *database types* and user-defined *classes*.

For example, you can use an alias to abbreviate the name of the primitive type `boolean` to `bool`:

```
class bool = boolean;
```

Or, to use a class `OneTwo` defined in a *module* `M` in `OneTwoThreeLib.qll`, you could create an alias to use the shorter name `OT` instead:

```
import OneTwoThreeLib

class OT = M::OneTwo;

...

from OT ot
select ot
```

6.1.3 Predicate aliases

Use the following syntax to define an alias for a *non-member predicate*:

```
predicate PredAlias = PredicateName/Arity;
```

This works for predicates *with* or *without* result.

For example, suppose you frequently use the following predicate, which calculates the successor of a positive integer less than ten:

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

You can use an alias to abbreviate the name to `succ`:

```
predicate succ = getSuccessor/1;
```

As an example of a predicate without result, suppose you have a predicate that holds for any positive integer less than ten:

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

You could give the predicate a more descriptive name as follows:

```
predicate lessThanTen = isSmall/1;
```

CHAPTER

SEVEN

VARIABLES

Variables in QL are used in a similar way to variables in algebra or logic. They represent sets of values, and those values are usually restricted by a formula.

This is different from variables in some other programming languages, where variables represent memory locations that may contain data. That data can also change over time. For example, in QL, `n = n + 1` is an equality *formula* that holds only if `n` is equal to `n + 1` (so in fact it does not hold for any numeric value). In Java, `n = n + 1` is not an equality, but an assignment that changes the value of `n` by adding `1` to the current value.

7.1 Declaring a variable

All variable declarations consist of a *type* and a name for the variable. The name can be any identifier that starts with an uppercase or lowercase letter.

For example, `int i`, `SsaDefinitionNode node`, and `LocalScopeVariable lsv` declare variables `i`, `node`, and `lsv` with types `int`, `SsaDefinitionNode`, and `LocalScopeVariable` respectively.

Variable declarations appear in different contexts, for example in a *select clause*, inside a *quantified formula*, as an argument of a *predicate*, and many more.

Conceptually, you can think of a variable as holding all the values that its type allows, subject to any further constraints.

For example, consider the following select clause:

```
from int i
where i in [0 .. 9]
select i
```

Just based on its type, the variable `i` could contain all integers. However, it is constrained by the formula `i in [0 .. 9]`. Consequently, the result of the select clause is the ten numbers between `0` and `9` inclusive.

As an aside, note that the following query leads to a compile-time error:

```
from int i
select i
```

In theory, it would have infinitely many results, as the variable `i` is not constrained to a finite number of possible values. See *Binding* for more information.

7.2 Free and bound variables

Variables can have different roles. Some variables are **free**, and their values directly affect the value of an *expression* that uses them, or whether a *formula* that uses them holds or not. Other variables, called **bound** variables, are restricted to specific sets of values.

It might be easiest to understand this distinction in an example. Take a look at the following expressions:

```
"hello".indexOf("l")

min(float f | f in [-3 .. 3])

(i + 7) * 3

x.sqrt()
```

The first expression doesnt have any variables. It finds the (zero-based) indices of where `"l"` occurs in the string `"hello"`, so it evaluates to `2` and `3`.

The second expression evaluates to `-3`, the minimum value in the range `[-3 .. 3]`. Although this expression uses a variable `f`, it is just a placeholder or dummy variable, and you cant assign any values to it. You could replace `f` with a different variable without changing the meaning of the expression. For example, `min(float f | f in [-3 .. 3])` is always equal to `min(float other | other in [-3 .. 3])`. This is an example of a **bound variable**.

What about the expressions `(i + 7) * 3` and `x.sqrt()`? In these two cases, the values of the expressions depend on what values are assigned to the variables `i` and `x` respectively. In other words, the value of the variable has an impact on the value of the expression. These are examples of **free variables**.

Similarly, if a formula contains free variables, then the formula can hold or not hold depending on the values assigned to those variables[1]. For example:

```
"hello".indexOf("l") = 1

min(float f | f in [-3 .. 3]) = -3

(i + 7) * 3 instanceof int

exists(float y | x.sqrt() = y)
```

The first formula doesnt contain any variables, and it never holds (since `"hello".indexOf("l")` has values 2 and 3, never 1).

The second formula only contains a bound variable, so is unaffected by changes to that variable. Since `min(float f | f in [-3 .. 3])` is equal to `-3`, this formula always holds.

The third formula contains a free variable `i`. Whether or not the formula holds, depends on what values are assigned to `i`. For example, if `i` is assigned `1` or `2` (or any other `int`) then the formula holds. On the other hand, if `i` is assigned `3.5`, then it doesnt hold.

[1] This is a slight simplification. There are some formulas that are always true or always false, regardless of the assignments to their free variables. However, you wont usually use these when youre writing QL. For example, and `a = a` is always true (known as a tautology), and `x and not x` is always false.

The last formula contains a free variable `x` and a bound variable `y`. If `x` is assigned a non-negative number, then the final formula holds. On the other hand, if `x` is assigned `-9` for example, then the formula doesnt hold. The variable `y` doesnt affect whether the formula holds or not.

For more information about how assignments to free variables are computed, see *Evaluation of QL programs*.

CHAPTER

EIGHT

EXPRESSIONS

An expression evaluates to a set of values and has a type.

For example, the expression `1 + 2` evaluates to the integer `3` and the expression `"QL"` evaluates to the string `"QL"`. `1 + 2` has *type* `int` and `"QL"` has type `string`.

The following sections describe the expressions that are available in QL.

8.1 Variable references

A variable reference is the name of a declared *variable*. This kind of expression has the same type as the variable it refers to.

For example, if you have *declared* the variables `int i` and `LocalScopeVariable lsv`, then the expressions `i` and `lsv` have types `int` and `LocalScopeVariable` respectively.

You can also refer to the variables `this` and `result`. These are used in *predicate* definitions and act in the same way as other variable references.

8.2 Literals

You can express certain values directly in QL, such as numbers, booleans, and strings.

- *Boolean* literals: These are the values `true` and `false`.

- *Integer* literals: These are sequences of decimal digits (`0` through `9`), possibly starting with a minus sign (`-`). For example:

```
0
42
-2048
```

- *Float* literals: These are sequences of decimal digits separated by a dot (`.`), possibly starting with a minus sign (`-`). For example:

```
2.0
123.456
-100.5
```

- *String* literals: These are finite strings of 16-bit characters. You can define a string literal by enclosing characters in quotation marks (`"..."`). Most characters represent themselves, but there are a few characters that you need to escape with a backslash. The following are examples of string literals:

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

See String literals in the QL language specification for more details.

Note: there is no date literal in QL. Instead, to specify a *date*, you should convert a string to the date that it represents using the `toDate()` predicate. For example, `"2016-04-03".toDate()` is the date April 3, 2016, and `"2000-01-01 00:00:01".toDate()` is the point in time one second after New Year 2000.

The following string formats are recognized as dates:

  - **ISO dates**, such as `"2016-04-03 17:00:24"`. The seconds part is optional (assumed to be `"00"` if its missing), and the entire time part can also be missing (in which case its assumed to be `"00:00:00"`).
  - **Short-hand ISO dates**, such as `"20160403"`.
  - **UK-style dates**, such as `"03/04/2016"`.

– **Verbose dates**, such as `"03 April 2016"`.

8.3 Parenthesized expressions

A parenthesized expression is an expression surrounded by parentheses, `(` and `)`. This expression has exactly the same type and values as the original expression. Parentheses are useful for grouping expressions together to remove ambiguity and improve readability.

8.4 Ranges

A range expression denotes a range of values ordered between two expressions. It consists of two expressions separated by `..` and enclosed in brackets (`[` and `]`). For example, `[3 .. 7]` is a valid range expression. Its values are any integers between 3 and 7 (including 3 and 7 themselves).

In a valid range, the start and end expression are integers, floats, or dates. If one of them is a date, then both must be dates. If one of them is an integer and the other a float, then both are treated as floats.

8.5 Set literal expressions

A set literal expression allows the explicit listing of a choice between several values. It consists of a comma-separated collection of expressions that are enclosed in brackets (`[` and `]`). For example, `[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]` is a valid set literal expression. Its values are the first ten prime numbers.

The values of the contained expressions need to be of *compatible types* for a valid set literal expression. Furthermore, at least one of the set elements has to be of a type that is a supertype of the types of all the other contained expressions.

Set literals are supported from release 2.1.0 of the CodeQL CLI, and release 1.24 of LGTM Enterprise.

8.6 Super expressions

Super expressions in QL are similar to super expressions in other programming languages, such as Java. You can use them in predicate calls, when you want to use the predicate definition from a supertype. In practice, this is useful when a predicate inherits two definitions from its supertypes. In that case, the predicate must *override* those definitions to avoid ambiguity. However, if you want to use the definition from a particular supertype instead of writing a new definition, you can use a super expression.

In the following example, the class `C` inherits two definitions of the predicate `getANumber()` one from `A` and one from `B`. Instead of overriding both definitions, it uses the definition from `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 {
  // Need to define `int getANumber()`; otherwise it would be␣
↪ambiguous
  int getANumber() {
    result = B.super.getANumber()
  }
}

from C c
select c, c.getANumber()
```

The result of this query is `1, 3`.

8.7 Calls to predicates (with result)

Calls to *predicates with results* are themselves expressions, unlike calls to *predicates without results* which are formulas. See *Calls to predicates* in the *Formulas* topic for more general information about calls.

A call to a predicate with result evaluates to the values of the `result` variable of the called predicate.

For example `a.getAChild()` is a call to a predicate `getAChild()` on a variable `a`. This call evaluates to the set of children of `a`.

8.8 Aggregations

An aggregation is a mapping that computes a result value from a set of input values that are specified by a formula.

The general syntax is:

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

The variables *declared* in `<variable declarations>` are called the **aggregation variables**.

Ordered aggregates (namely `min`, `max`, `rank`, `concat`, and `strictconcat`) are ordered by their `<expression>` values by default. The ordering is either numeric (for integers and floating point numbers) or lexicographic (for strings). Lexicographic ordering is based on the Unicode value of each character.

To specify a different order, follow `<expression>` with the keywords `order by`, then the expression that specifies the order, and optionally the keyword `asc` or `desc` (to determine whether to order the expression in ascending or descending order). If you dont specify an ordering, it defaults to `asc`.

The following aggregates are available in QL:

- `count`: This aggregate determines the number of distinct values of `<expression>` for each possible assignment of the aggregation variables.

For example, the following aggregation returns the number of files that have more than 500 lines:

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

If there are no possible assignments to the aggregation variables that satisfy the formula, as in `count(int i | i = 1 and i = 2 | i)`, then `count` defaults to the value `0`.

- `min` and `max`: These aggregates determine the smallest (`min`) or largest (`max`) value of `<expression>` among the possible assignments to the aggregation variables. In this case, `<expression>` must be of numeric type or of type `string`.

  For example, the following aggregation returns the name of the `.js` file (or files) with the largest number of lines:

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

  The following aggregation returns the minimum string `s` out of the three strings mentioned below, that is, the string that comes first in the lexicographic ordering of all the possible values of `s`. (In this case, it returns `"De Morgan"`.)

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

- `avg`: This aggregate determines the average value of `<expression>` for all possible assignments to the aggregation variables. The type of `<expression>` must be numeric. If there are no possible assignments to the aggregation variables that satisfy the formula, the aggregation fails and returns no values. In other words, it evaluates to the empty set.

  For example, the following aggregation returns the average of the integers `0`, `1`, `2`, and `3`:

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

- `sum`: This aggregate determines the sum of the values of `<expression>` over all possible assignments to the aggregation variables. The type of `<expression>` must be numeric. If there are no possible assignments to the aggregation variables that satisfy the formula, then the sum is `0`.

  For example, the following aggregation returns the sum of `i * j` for all possible values of `i` and `j`:

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

- `concat`: This aggregate concatenates the values of `<expression>` over all possible assignments to the aggregation variables. Note that `<expression>` must be of type `string`. If there are no possible assignments to the aggregation variables that satisfy the formula, then `concat` defaults to the empty string.

  For example, the following aggregation returns the string `"3210"`, that is, the concatenation of the strings `"0"`, `"1"`, `"2"`, and `"3"` in descending order:

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

  The `concat` aggregate can also take a second expression, separated from the first one by a comma. This second expression is inserted as a separator between each concatenated value.

  For example, the following aggregation returns `"0|1|2|3"`:

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

- `rank`: This aggregate takes the possible values of `<expression>` and ranks them. In this case, `<expression>` must be of numeric type or of type `string`. The aggregation returns the value that is ranked in the position specified by the **rank expression**. You must include this rank expression in brackets after the keyword `rank`.

For example, the following aggregation returns the value that is ranked 4th out of all the possible values. In this case, `8` is the 4th integer in the range from `5` through `15`:

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

Note that the rank indices start at `1`, so `rank[0](...)` returns no results.

- `strictconcat`, `strictcount`, and `strictsum`: These aggregates work like `concat`, `count`, and `sum` respectively, except that they are *strict*. That is, if there are no possible assignments to the aggregation variables that satisfy the formula, then the entire aggregation fails and evaluates to the empty set (instead of defaulting to `0` or the empty string). This is useful if youre only interested in results where the aggregation body is non-trivial.
- `unique`: This aggregate depends on the values of `<expression>` over all possible assignments to the aggregation variables. If there is a unique value of `<expression>` over the aggregation variables, then the aggregate evaluates to that value. Otherwise, the aggregate has no value.

  For example, the following query returns the positive integers `1`, `2`, `3`, `4`, `5`. For negative integers `x`, the expressions `x` and `x.abs()` have different values, so the value for `y` in the aggregate expression is not uniquely determined.

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

  The `unique` aggregate is supported from release 2.1.0 of the CodeQL CLI, and release 1.24 of LGTM Enterprise.

8.8.1 Evaluation of aggregates

In general, aggregate evaluation involves the following steps:

1. Determine the input variables: these are the aggregation variables declared in `<variable declarations>` and also the variables declared outside of the aggregate that are used in some component of the aggregate.

2. Generate all possible distinct tuples (combinations) of the values of input variables such that the `<formula>` holds true. Note that the same value of an aggregate variable may appear in multiple distinct tuples. All such occurrences of the same value are treated as distinct occurrences when processing tuples.

3. Apply `<expression>` on each tuple and collect the generated (distinct) values. The application of `<expression>` on a tuple may result in generating more than one value.

4. Apply the aggregation function on the values generated in step 3 to compute the final result.

Let us apply these steps to the `sum` aggregate in the following query:

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

1. Input variables: `i`, `j`.

2. All possible tuples `(<value of i>, <value of j>)` satisfying the given condition: `(0, 0)`, `(0, 1)`, `(0, 2)`, `(0, 3)`, `(0, 4)`, `(0, 5)`, `(1, 0)`, `(1, 1)`, `...`, `(4, 5)`.

   30 tuples are generated in this step.

3. Apply the `<expression> i` on all tuples. This means selecting all values of `i` from all tuples: `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`.

4. Apply the aggregation function `sum` on the above values to get the final result 60.

If we change `<expression>` to `i + j` in the above query, the query result is `135` since applying `i + j` on all tuples results in following values: `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`.

Next, consider the following query:

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

1. `s` is the input variable of the aggregate.
2. A single tuple `"hello"` is generated in this step.
3. The `<expression> charAt(_)` is applied on this tuple. The underscore `_` in `charAt(_)` is a *dont-care expression*, which represents any value. `s.charAt(_)` generates four distinct values `h, e, l, o`.
4. Finally, `count` is applied on these values, and the query returns 4.

8.8.2 Omitting parts of an aggregation

The three parts of an aggregation are not always required, so you can often write the aggregation in a simpler form:

1. If you want to write an aggregation of the form `<aggregate>(<type> v | <expression> = v | v)`, then you can omit the `<variable declarations>` and `<formula>` parts and write it as follows:

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

   For example, the following aggregations determine how many times the letter `l` occurs in string `"hello"`. These forms are equivalent:

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

2. If there only one aggregation variable, you can omit the `<expression>` part instead. In this case, the expression is considered to be the aggregation variable itself. For example, the following aggregations are equivalent:

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

3. As a special case, you can omit the `<expression>` part from `count` even if there is more than one aggregation variable. In such a case, it counts the

number of distinct tuples of aggregation variables that satisfy the formula. In other words, the expression part is considered to be the constant `1`. For example, the following aggregations are equivalent:

```
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. You can omit the `<formula>` part, but in that case you should include two vertical bars:

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

This is useful if you dont want to restrict the aggregation variables any further. For example, the following aggregation returns the maximum number of lines across all files:

```
max(File f | | f.getTotalNumberOfLines())
```

5. Finally, you can also omit both the `<formula>` and `<expression>` parts. For example, the following aggregations are equivalent ways to count the number of files in a database:

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

8.8.3 Monotonic aggregates

In addition to standard aggregates, QL also supports monotonic aggregates. Monotonic aggregates differ from standard aggregates in the way that they deal with the values generated by the `<expression>` part of the formula:

- Standard aggregates take the `<expression>` values for each `<formula>` value and flatten them into a list. A single aggregation function is applied to all the values.
- Monotonic aggregates take an `<expression>` for each value given by the

<formula>, and create combinations of all the possible values. The aggregation function is applied to each of the resulting combinations.

In general, if the <expression> is total and functional, then monotonic aggregates are equivalent to standard aggregates. Results differ when there is not precisely one <expression> value for each value generated by the <formula>:

- If there are missing <expression> values (that is, there is no <expression> value for a value generated by the <formula>), monotonic aggregates wont compute a result, as you cannot create combinations of values including exactly one <expression> value for each value generated by the <formula>.
- If there is more than one <expression> per <formula> result, you can create multiple combinations of values including exactly one <expression> value for each value generated by the <formula>. Here, the aggregation function is applied to each of the resulting combinations.

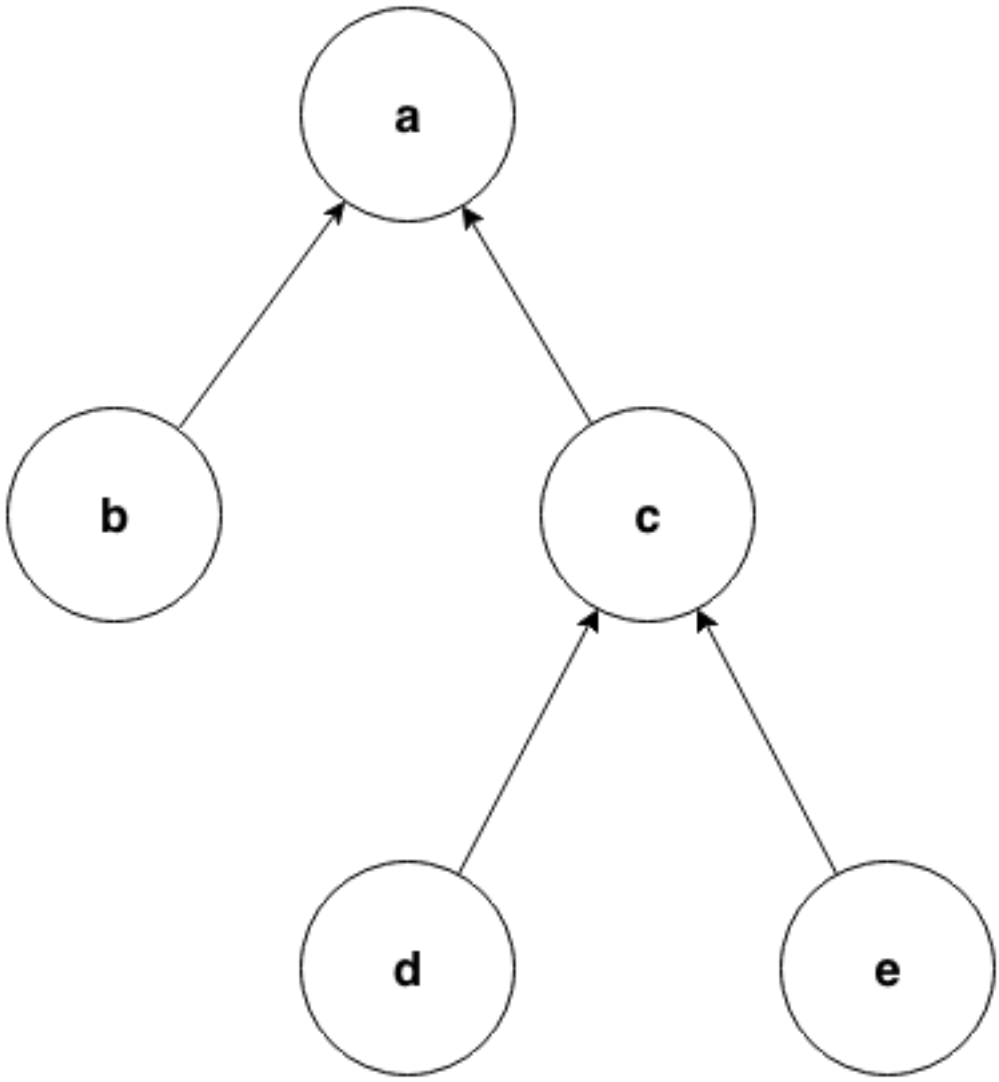
Recursive monotonic aggregates

Monotonic aggregates may be used *recursively*, but the recursive call may only appear in the expression, and not in the range. The recursive semantics for aggregates are the same as the recursive semantics for the rest of QL. For example, we might define a predicate to calculate the distance of a node in a graph from the leaves as follows:

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

Here the recursive call is in the expression, which is legal. The recursive semantics for aggregates are the same as the recursive semantics for the rest of QL. If you understand how aggregates work in the non-recursive case then you should not find it difficult to use them recursively. However, it is worth seeing how the evaluation of a recursive aggregation proceeds.

Consider the depth example we just saw with the following graph as input (arrows point from children to parents):



Then the evaluation of the `depth` predicate proceeds as follows:

Stage	depth	Comments
0		We always begin with the empty set.
1	`(0, b), (0, d), (0, e)`	The nodes with no children have depth 0. The recursive step for **a** and **c** fails to produce a value, since some of their children do not have values for `depth`.
2	`(0, b), (0, d), (0, e), (1, c)`	The recursive step for **c** succeeds, since `depth` now has a value for all its children (**d** and **e**). The recursive step for **a** still fails.
3	`(0, b), (0, d), (0, e), (1, c), (2, a)`	The recursive step for **a** succeeds, since `depth` now has a value for all its children (**b** and **c**).

Here, we can see that at the intermediate stages it is very important for the aggregate to fail if some of the children lack a value - this prevents erroneous values being added.

8.9 Any

The general syntax of an `any` expression is similar to the syntax of an *aggregation*, namely:

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

You should always include the *variable declarations*, but the *formula* and *expression* parts are optional.

The `any` expression denotes any values that are of a particular form and that satisfy a particular condition. More precisely, the `any` expression:

1. Introduces temporary variables.
2. Restricts their values to those that satisfy the `<formula>` part (if its present).
3. Returns `<expression>` for each of those variables. If there is no `<expression>` part, then it returns the variables themselves.

The following table lists some examples of different forms of any expressions:

Expression	Values
`any(File f)`	all `Files` in the database
`any(Element e | e.getName())`	the names of all `Elements` 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`

Note: There is also a built-in predicate `any()`. This is a predicate that always holds.

8.10 Unary operations

A unary operation is a minus sign (-) or a plus sign (+) followed by an expression of type `int` or `float`. For example:

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

A plus sign leaves the values of the expression unchanged, while a minus sign takes the arithmetic negations of the values.

8.11 Binary operations

A binary operation consists of an expression, followed by a binary operator, followed by another expression. For example:

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

You can use the following binary operators in QL:

Name	Symbol
Addition/concatenation	+
Multiplication	*
Division	/
Subtraction	-
Modulo	%

If both expressions are numbers, these operators act as standard arithmetic operators. For example, `10.6 - 3.2` has value `7.4`, `123.456 * 0` has value `0`, and `9 % 4` has value `1` (the remainder after dividing `9` by `4`). If both operands are integers, then the result is an integer. Otherwise the result is a floating-point number.

You can also use `+` as a string concatenation operator. In this case, at least one of the expressions must be a stringthe other expression is implicitly converted to a string using the `toString()` predicate. The two expressions are concatenated, and the result is a string. For example, the expression `221 + "B"` has value `"221B"`.

8.12 Casts

A cast allows you to constrain the *type* of an expression. This is similar to casting in other languages, for example in Java.

You can write a cast in two ways:

- As a postfix cast: A dot followed by the name of a type in parentheses. For example, `x.(Foo)` restricts the type of `x` to `Foo`.

- As a prefix cast: A type in parentheses followed by another expression. For example, `(Foo)x` also restricts the type of `x` to `Foo`.

Note that a postfix cast is equivalent to a prefix cast surrounded by parentheses`x.(Foo)` is exactly equivalent to `((Foo)x)`.

Casts are useful if you want to call a *member predicate* that is only defined for a more specific type. For example, the following query selects Java classes that have a direct supertype called List:

```
import java

from Type t
where t.(Class).getASupertype().hasName("List")
select t
```

Since the predicate `getASupertype()` is defined for `Class`, but not for `Type`, you cant call `t.getASupertype()` directly. The cast `t.(Class)` ensures that `t` is of type `Class`, so it has access to the desired predicate.

If you prefer to use a prefix cast, you can rewrite the `where` part as:

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

8.13 Dont-care expressions

This is an expression written as a single underscore `_`. It represents any value. (You dont care what the value is.)

Unlike other expressions, a dont-care expression does not have a type. In practice, this means that `_` doesnt have any *member predicates*, so you cant call `_.somePredicate()`.

For example, the following query selects all the characters in the string `"hello"`:

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

The `charAt(int i)` predicate is defined on strings and usually takes an `int` argument. Here the dont care expression _ is used to tell the query to select characters at every possible index. The query returns the values `h`, `e`, `l`, and `o`.

CHAPTER

NINE

FORMULAS

Formulas define logical relations between the free variables used in expressions.

Depending on the values assigned to those *free variables*, a formula can be true or false. When a formula is true, we often say that the formula *holds*. For example, the formula `x = 4 + 5` holds if the value `9` is assigned to `x`, but it doesnt hold for other assignments to `x`. Some formulas dont have any free variables. For example `1 < 2` always holds, and `1 > 2` never holds.

You usually use formulas in the bodies of classes, predicates, and select clauses to constrain the set of values that they refer to. For example, you can define a class containing all integers `i` for which the formula `i in [0 .. 9]` holds.

The following sections describe the kinds of formulas that are available in QL.

9.1 Comparisons

A comparison formula is of the form:

```
<expression> <operator> <expression>
```

See the tables below for an overview of the available comparison operators.

9.1.1 Order

To compare two expressions using one of these order operators, each expression must have a type and those types must be *compatible* and orderable.

Name	Symbol
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

For example, the formulas `"Ann" < "Anne"` and `5 + 6 >= 11` both hold.

9.1.2 Equality

To compare two expressions using =, at least one of the expressions must have a type. If both expressions have a type, then their types must be *compatible*.

To compare two expressions using `!=`, both expressions must have a type. Those types must also be *compatible*.

Name	Symbol
Equal to	=
Not equal to	!=

For example, `x.sqrt() = 2` holds if `x` is 4, and `4 != 5` always holds.

For expressions `A` and `B`, the formula `A = B` holds if there is a pair of valuesone from `A` and one from `B`that are the same. In other words, `A` and `B` have at least one value in common. For example, `[1 .. 2] = [2 .. 5]` holds, since both expressions have the value 2.

As a consequence, `A != B` has a very different meaning to the *negation* `not A = B`[1]:

[1] The difference between `A != B` and `not A = B` is due to the underlying quantifiers. If you think of `A` and `B` as sets of values, then `A != B` means:

```
exists( a, b | a in A and b in B | a != b )
```

On the other hand, `not A = B` means:

```
not exists( a, b | a in A and b in B | a = b )
```

This is equivalent to `forall( a, b | a in A and b in B | a != b )`, which is very differ-

- `A != B` holds if there is a pair of values (one from `A` and one from `B`) that are different.
- `not A = B` holds if it is *not* the case that there is a pair of values that are the same. In other words, `A` and `B` have no values in common.

Examples

1. **If both expressions have a single value (for example `1` and `0`), then comparison is s**

    - `1 != 0` holds.
    - `1 = 0` doesnt hold.
    - `not 1 = 0` holds.

2. **Now compare `1` and `[1 .. 2]`:**

    - `1 != [1 .. 2]` holds, because `1 != 2`.
    - `1 = [1 .. 2]` holds, because `1 = 1`.
    - `not 1 = [1 .. 2]` doesnt hold, because there is a common value (`1`).

3. **Compare `1` and `none()` (the empty set):**

    - `1 != none()` doesnt hold, because there are no values in `none()`, so no values that are not equal to `1`.
    - `1 = none()` also doesnt hold, because there are no values in `none()`, so no values that are equal to `1`.
    - `not 1 = none()` holds, because there are no common values.

9.2 Type checks

A type check is a formula that looks like:

ent from the first formula.

```
<expression> instanceof <type>
```

You can use a type check formula to check whether an expression has a certain type. For example, `x instanceof Person` holds if the variable `x` has type `Person`.

9.3 Range checks

A range check is a formula that looks like:

```
<expression> in <range>
```

You can use a range check formula to check whether a numeric expression is in a given *range*. For example, `x in [2.1 .. 10.5]` holds if the variable `x` is between the values `2.1` and `10.5` (including `2.1` and `10.5` themselves).

Note that `<expression> in <range>` is equivalent to `<expression> = <range>`. Both formulas check whether the set of values denoted by `<expression>` is the same as the set of values denoted by `<range>`.

9.4 Calls to predicates

A call is a formula or *expression* that consists of a reference to a predicate and a number of arguments.

For example, `isThree(x)` might be a call to a predicate that holds if the argument `x` is 3, and `x.isEven()` might be a call to a member predicate that holds if `x` is even.

A call to a predicate can also contain a closure operator, namely `*` or `+`. For example, `a.isChildOf+(b)` is a call to the *transitive closure* of `isChildOf()`, so it holds if `a` is a descendent of `b`.

The predicate reference must resolve to exactly one predicate. See *Name resolution* for more information about how a predicate reference is resolved.

If the call resolves to a predicate without result, then the call is a formula.

It is also possible to call a predicate with result. This kind of call is an expression in QL, instead of a formula. See *Calls to predicates (with result)* for the corresponding topic.

9.5 Parenthesized formulas

A parenthesized formula is any formula surrounded by parentheses, `(` and `)`. This formula has exactly the same meaning as the enclosed formula. The parentheses often help to improve readability and group certain formulas together.

9.6 Quantified formulas

A quantified formula introduces temporary variables and uses them in formulas in its body. This is a way to create new formulas from existing ones.

9.6.1 Explicit quantifiers

The following explicit quantifiers are the same as the usual existential and universal quantifiers in mathematical logic.

`exists`

This quantifier has the following syntax:

```
exists(<variable declarations> | <formula>)
```

You can also write `exists(<variable declarations> | <formula 1> | <formula 2>)`. This is equivalent to `exists(<variable declarations> | <formula 1> and <formula 2>)`.

This quantified formula introduces some new variables. It holds if there is at least one set of values that the variables could take to make the formula in the body true.

For example, `exists(int i | i instanceof OneTwoThree)` introduces a temporary variable of type `int` and holds if any value of that variable has type `OneTwoThree`.

forall

This quantifier has the following syntax:

```
forall(<variable declarations> | <formula 1> | <formula 2>)
```

`forall` introduces some new variables, and typically has two formulas in its body. It holds if `<formula 2>` holds for all values that `<formula 1>` holds for.

For example, `forall(int i | i instanceof OneTwoThree | i < 5)` holds if all integers that are in the class `OneTwoThree` are also less than `5`. In other words, if there is a value in `OneTwoThree` that is greater than or equal to `5`, then the formula doesnt hold.

Note that `forall(<vars> | <formula 1> | <formula 2>)` is logically the same as `not exists(<vars> | <formula 1> | not <formula 2>)`.

forex

This quantifier has the following syntax:

```
forex(<variable declarations> | <formula 1> | <formula 2>)
```

This quantifier exists as a shorthand for:

```
forall(<vars> | <formula 1> | <formula 2>) and
exists(<vars> | <formula 1> | <formula 2>)
```

In other words, `forex` works in a similar way to `forall`, except that it ensures that there is at least one value for which `<formula 1>` holds. To see why this is useful, note that the `forall` quantifier could hold trivially. For example, `forall(int i | i = 1 and i = 2 | i = 3)` holds: there are no integers `i` which are equal to both `1` and `2`, so the second part of the body `(i = 3)` holds for every integer for which the first part holds.

Since this is often not the behavior that you want in a query, the `forex` quantifier is a useful shorthand.

9.6.2 Implicit quantifiers

Implicitly quantified variables can be introduced using dont care expressions. These are used when you need to introduce a variable to use as an argument to a predicate call, but dont care about its value. For further information, see *Dont-care expressions*.

9.7 Logical connectives

You can use a number of logical connectives between formulas in QL. They allow you to combine existing formulas into longer, more complex ones.

To indicate which parts of the formula should take precedence, you can use parentheses. Otherwise, the order of precedence from highest to lowest is as follows:

1. Negation (*not*)
2. Conditional formula (*if then else*)
3. Conjunction (*and*)
4. Disjunction (*or*)
5. Implication (*implies*)

For example, `A and B implies C or D` is equivalent to `(A and B) implies (C or D)`.

Similarly, `A and not if B then C else D` is equivalent to `A and (not (if B then C else D))`.

Note that the *parentheses* in the above examples are not necessary, since they highlight the default precedence. You usually only add parentheses to override the default precedence, but you can also add them to make your code easier to read (even if they arent required).

The logical connectives in QL work similarly to Boolean connectives in other programming languages. Here is a brief overview:

9.7.1 `not`

You can use the keyword `not` before a formula. The resulting formula is called a negation.

`not A` holds exactly when `A` doesnt hold.

Example

The following query selects files that are not HTML files.

```
from File f
where not f.getFileType().isHtml()
select f
```

Note: You should be careful when using `not` in a recursive definition, as this could lead to non-monotonic recursion. For more information, see the section on *Non-monotonic recursion*.

9.7.2 `if ... then ... else`

You can use these keywords to write a conditional formula. This is another way to simplify notation: `if A then B else C` is the same as writing `(A and B) or ((not A) and C)`.

Example

With the following definition, `visibility(c)` returns `"public"` if `x` is a public class and returns `"private"` otherwise:

```
string visibility(Class c){
  if c.isPublic()
  then result = "public"
```

(continues on next page)

(continued from previous page)

```
  else result = "private"
}
```

9.7.3 and

You can use the keyword `and` between two formulas. The resulting formula is called a conjunction.

`A and B` holds if, and only if, both `A` and `B` hold.

Example

The following query selects files that have the `js` extension and contain fewer than 200 lines of code:

```
from File f
where f.getExtension() = "js" and
  f.getNumberOfLinesOfCode() < 200
select f
```

9.7.4 or

You can use the keyword `or` between two formulas. The resulting formula is called a disjunction.

`A or B` holds if at least one of `A` or `B` holds.

Example

With the following definition, an integer is in the class `OneTwoThree` if it is equal to 1, 2, or 3:

```
class OneTwoThree extends int {
  OneTwoThree() {
    this = 1 or this = 2 or this = 3
  }
```

(continues on next page)

(continued from previous page)

```
  ...
}
```

9.7.5 implies

You can use the keyword `implies` between two formulas. The resulting formula is called an implication. This is just a simplified notation: `A implies B` is the same as writing `(not A) or B`.

Example

The following query selects any `SmallInt` that is odd, or a multiple of `4`.

```
class SmallInt extends int {
  SmallInt() { this = [1 .. 10] }
}

from SmallInt x
where x % 2 = 0 implies x % 4 = 0
select x
```

CHAPTER

TEN

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 *nam*

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

(continues on next page)

(continued from previous page)

```
  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.

10.1 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.

10.1.1 `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);
```

(continues on next page)

(continued from previous page)

```
  ...
}
```

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
}
```

10.1.2 `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.

10.1.3 `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.

10.1.4 `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*.

10.1.5 `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.

10.1.6 `final`

Available for: *classes*, *member predicates*, *fields*

The `final` annotation is applied to entities that cant be overridden or extended. In other words, a final class cant act as a base type for any other types, and a final predicate or field cant be overridden in a subclass.

This is useful if you dont 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 wouldnt 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() }
}
```

10.1.7 `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.

10.1.8 `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.

10.1.9 `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 modules *namespace*.

10.1.10 `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.

10.1.11 Compiler pragmas

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

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]`

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[noinline]`

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 youve 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 optimizers inlining may undo the work of the helper predicate, so its a good idea to annotate it with `pragma[noinline]`.

`pragma[nomagic]`

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 shouldnt use the `pragma[nomagic]` annotation unless recommended to do so by GitHub.

Note that `nomagic` implies `noinline`.

`pragma[noopt]`

The `pragma[noopt]` annotation is used to prevent the QL optimizer from optimizing a predicate, except when its 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 cant 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
  )
}
```

10.1.12 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*.

10.1.13 Binding sets

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

`bindingset[...]`

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

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

See *Binding behavior* in the *Predicates* topic for examples and more information.

CHAPTER

ELEVEN

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*.

11.1 Examples of recursive predicates

Here are a few examples of recursive predicates in QL:

11.1.1 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
```

(continues on next page)

(continued from previous page)

```
  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`).

11.1.2 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.

11.1.3 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*()`.

11.2 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:

11.2.1 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()
}
```

11.2.2 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 liars 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 doesnt 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 isnt 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 ps 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()
)
```

CHAPTER

TWELVE

LEXICAL SYNTAX

The QL syntax includes different kinds of keywords, identifiers, and comments.

For an overview of the lexical syntax, see Lexical syntax in the QL language specification.

12.1 Comments

All standard one-line and multiline comments, as described in the QL language specification, are ignored by the QL compiler and are only visible in the source code. You can also write another kind of comment, namely **QLDoc comments**. These comments describe QL entities and are displayed as pop-up information in QL editors. For information about QLDoc comments, see the QLDoc comment specification.

The following example uses these three different kinds of comments:

```
/**
 * A QLDoc comment that describes the class `Digit`.
 */
class Digit extends int {  // A short one-line comment
  Digit() {
    this in [0 .. 9]
  }
}
```

(continues on next page)

(continued from previous page)

```
/*
  A standard multiline comment, perhaps to provide
  additional details, or to write a TODO comment.
*/
```

CHAPTER

THIRTEEN

NAME RESOLUTION

The QL compiler resolves names to program elements.

As in other programming languages, there is a distinction between the names used in QL code, and the underlying QL entities they refer to.

It is possible for different entities in QL to have the same name, for example if they are defined in separate modules. Therefore, it is important that the QL compiler can resolve the name to the correct entity.

When you write your own QL, you can use different kinds of expressions to refer to entities. Those expressions are then resolved to QL entities in the appropriate *namespace*.

In summary, the kinds of expressions are:

- **Module expressions**
  - These refer to modules.
  - They can be simple *names*, *qualified references* (in import statements), or *selections*.
- **Type expressions**
  - These refer to types.
  - They can be simple *names* or *selections*.
- **Predicate expressions**

- These refer to predicates.
- They can be simple *names* or names with arities (for example in an *alias* definition), or *selections*.

13.1 Names

To resolve a simple name (with arity), the compiler looks for that name (and arity) in the *namespaces* of the current module.

In an *import statement*, name resolution is slightly more complicated. For example, suppose you define a *query module* `Example.ql` with the following import statement:

```
import javascript
```

The compiler first checks for a *library module* `javascript.qll`, using the steps described below for qualified references. If that fails, it checks for an *explicit module* named `javascript` defined in the *module namespace* of `Example.ql`.

13.2 Qualified references

A qualified reference is a module expression that uses `.` as a file path separator. You can only use such an expression in *import statements*, to import a library module defined by a relative path.

For example, suppose you define a *query module* `Example.ql` with the following import statement:

```
import examples.security.MyLibrary
```

To find the precise location of this *library module*, the QL compiler processes the import statement as follows:

1. The `.`s in the qualified reference correspond to file path separators, so it first looks up `examples/security/MyLibrary.qll` from the directory containing `Example.ql`.

2. If that fails, it looks up `examples/security/MyLibrary.qll` relative to the query directory, if any. The query directory is the first enclosing directory containing a file called `qlpack.yml`. (Or, in legacy products, a file called `queries.xml`.)

3. If the compiler cant find the library file using the above two checks, it looks up `examples/security/MyLibrary.qll` relative to each library path entry. The library path is usually specified using the `libraryPathDependencies` of the `qlpack.yml` file, though it may also depend on the tools you use to run your query, and whether you have specified any extra settings. For more information, see Library path in the QL language specification.

If the compiler cannot resolve an import statement, then it gives a compilation error.

13.3 Selections

You can use a selection to refer to a module, type, or predicate inside a particular module. A selection is of the form:

```
<module_expression>::<name>
```

The compiler resolves the module expression first, and then looks for the name in the *namespaces* for that module.

13.3.1 Example

Consider the following *library module*:

CountriesLib.qll

```
class Countries extends string {
  Countries() {
    this = "Belgium"
    or
    this = "France"
    or
```

(continues on next page)

(continued from previous page)

```
    this = "India"
  }
}

module M {
  class EuropeanCountries extends Countries {
    EuropeanCountries() {
      this = "Belgium"
      or
      this = "France"
    }
  }
}
```

You could write a query that imports `CountriesLib` and then uses `M::EuropeanCountries` to refer to the class `EuropeanCountries`:

```
import CountriesLib

from M::EuropeanCountries ec
select ec
```

Alternatively, you could import the contents of `M` directly by using the selection `CountriesLib::M` in the import statement:

```
import CountriesLib::M

from EuropeanCountries ec
select ec
```

That gives the query access to everything within `M`, but nothing within `CountriesLib` that isnt also in `M`.

13.4 Namespaces

When writing QL, its useful to understand how namespaces (also known as environments) work.

As in many other programming languages, a namespace is a mapping from **keys** to **entities**. A key is a kind of identifier, for example a name, and a QL entity is a *module*, a *type*, or a *predicate*.

Each module in QL has three namespaces:

- The **module namespace**, where the keys are module names and the entities are modules.
- The **type namespace**, where the keys are type names and the entities are types.
- The **predicate namespace**, where the keys are pairs of predicate names and arities, and the entities are predicates.

Its important to know that there is no relation between names in different namespaces. For example, two different modules can define a predicate `getLocation()` without confusion. As long as its clear which namespace you are in, the QL compiler resolves the name to the correct predicate.

13.4.1 Global namespaces

The namespaces containing all the built-in entities are called **global namespaces**, and are automatically available in any module. In particular:

- The **global module namespace** is empty.
- The **global type namespace** has entries for the *primitive types* `int`, `float`, `string`, `boolean`, and `date`, as well as any *database types* defined in the database schema.
- The **global predicate namespace** includes all the built-in predicates, as well as any *database predicates*.

In practice, this means that you can use the built-in types and predicates directly

in a QL module (without importing any libraries). You can also use any database predicates and types directlythese depend on the underlying database that you are querying.

13.4.2 Local namespaces

In addition to the global module, type, and predicate namespaces, each module defines a number of local module, type, and predicate namespaces.

For a module `M`, its useful to distinguish between its **declared**, **exported**, and **imported** namespaces. (These are described generically, but remember that there is always one for each of modules, types, and predicates.)

- The **declared** namespaces contain any names that are declaredthat is, definedin `M`.
- The **imported** namespaces contain any names exported by the modules that are imported into `M` using an *import statement*.
- The **exported** namespaces contain any names declared in `M`, or exported from a module imported into `M`, except names annotated with `private`. This includes everything in the imported namespaces that was not introduced by a private import.

This is easiest to understand in an example:

OneTwoThreeLib.qll

```
import MyFavoriteNumbers

class OneTwoThree extends int {
  OneTwoThree() {
    this = 1 or this = 2 or this = 3
  }
}

private module P {
  class OneTwo extends OneTwoThree {
    OneTwo() {
```

(continues on next page)

(continued from previous page)

```
      this = 1 or this = 2
    }
  }
}
```

The module `OneTwoThreeLib` **imports** anything that is exported by the module `MyFavoriteNumbers`.

It **declares** the class `OneTwoThree` and the module `P`.

It **exports** the class `OneTwoThree` and anything that is exported by `MyFavoriteNumbers`. It does not export `P`, since it is annotated with `private`.

13.4.3 Example

The namespaces of a general QL module are a union of the local namespaces, the namespaces of any enclosing modules, and the global namespaces. (You can think of global namespaces as the enclosing namespaces of a top-level module.)

Lets see what the module, type, and predicate namespaces look like in a concrete example:

For example, you could define a library module `Villagers` containing some of the classes and predicates that were defined in the QL tutorials:

Villagers.qll

```
import tutorial

predicate isBald(Person p) {
  not exists(string c | p.getHairColor() = c)
}

class Child extends Person {
  Child() {
    this.getAge() < 10
  }
```

(continues on next page)

(continued from previous page)

```
}

module S {
  predicate isSouthern(Person p) {
    p.getLocation() = "south"
  }

  class Southerner extends Person {
    Southerner() {
      isSouthern(this)
    }
  }
}
```

Module namespace

The module namespace of `Villagers` has entries for:

- The module `S`.
- Any modules exported by `tutorial`.

The module namespace of `S` also has entries for the module `S` itself, and for any modules exported by `tutorial`.

Type namespace

The type namespace of `Villagers` has entries for:

- The class `Child`.
- The types exported by the module `tutorial`.
- The built-in types, namely `int`, `float`, `string`, `date`, and `boolean`.

The type namespace of `S` has entries for:

- All the above types.
- The class `Southerner`.

Predicate namespace

The predicate namespace of `Villagers` has entries for:

- The predicate `isBald`, with arity 1.
- Any predicates (and their arities) exported by `tutorial`.
- The built-in predicates.

The predicate namespace of `S` has entries for:

- All the above predicates.
- The predicate `isSouthern`, with arity 1.

CHAPTER

FOURTEEN

EVALUATION OF QL PROGRAMS

A QL program is evaluated in a number of different steps.

14.1 Process

When a QL program is run against a database, it is compiled into a variant of the logic programming language Datalog. It is optimized for performance, and then evaluated to produce results.

These results are sets of ordered tuples. An ordered tuple is a finite, ordered sequence of values. For example, `(1, 2, "three")` is an ordered tuple with two integers and a string. There may be intermediate results produced while the program is being evaluated: these are also sets of tuples.

A QL program is evaluated from the bottom up, so a predicate is usually only evaluated after all the predicates it depends on are evaluated.

The database includes sets of ordered tuples for the built-in predicates and *external predicates*. Each evaluation starts from these sets of tuples. The remaining predicates and types in the program are organized into a number of layers, based on the dependencies between them. These layers are evaluated to produce their own sets of tuples, by finding the least fixed point of each predicate. (For example, see *Recursion*.)

The programs *queries* determine which of these sets of tuples make up the final results of the program. The results are sorted according to any ordering directives (`order by`) in the queries.

For more details about each step of the evaluation process, see the QL language specification.

14.2 Validity of programs

The result of a query must always be a **finite** set of values, otherwise it cant be evaluated. If your QL code contains an infinite predicate or query, the QL compiler usually gives an error message, so that you can identify the error more easily.

Here are some common ways that you might define infinite predicates. These all generate compilation errors:

- The following query conceptually selects all values of type `int`, without restricting them. The QL compiler returns the error `'i' is not bound to a value`:

```
from int i
select i
```

- The following predicate generates two errors: `'n' is not bound to a value` and `'result' is not bound to a value`:

```
int timesTwo(int n) {
  result = n * 2
}
```

- The following class `Person` contains all strings that start with `"Peter"`. There are infinitely many such strings, so this is another invalid definition. The QL compiler gives the error message `'this' is not bound to a value`:

```
class Person extends string {
  Person() {
    this.matches("Peter%")
  }
}
```

To fix these errors, its useful to think about **range restriction**: A predicate or

query is **range-restricted** if each of its variables has at least one *binding* occurrence. A variable without a binding occurrence is called **unbound**. Therefore, to perform a range restriction check, the QL compiler verifies that there are no unbound variables.

14.2.1 Binding

To avoid infinite relations in your queries, you must ensure that there are no unbound variables. To do this, you can use the following mechanisms:

1. **Finite types**: Variables of a finite *type* are bound. In particular, any type that is not *primitive* is finite. To give a finite type to a variable, you can *declare* it with a finite type, use a *cast*, or use a *type check*.

2. **Predicate calls**: A valid *predicate* is usually range-restricted, so it *binds* all its arguments. Therefore, if you *call* a predicate on a variable, the variable becomes bound.

   Important: If a predicate uses non-standard binding sets, then it does **not** always bind all its arguments. In such a case, whether the predicate call binds a specific argument depends on which other arguments are bound, and what the binding sets say about the argument in question. See *Binding sets* for more information.

   ---

3. **Binding operators**: Most operators, such as the *arithmetic operators*, require that all their operands are bound. For example, you cant add two variables in QL unless you have a finite set of possible values for both of them.

   However, there are some built-in operators that can bind their arguments. For example, if one side of an *equality check* (using =) is bound and the other side is a variable, then the variable becomes bound too. See the table below for examples.

Intuitively, a binding occurrence restricts the variable to a finite set of values, while a non-binding occurrence doesnt. Here are some examples to clarify the difference between binding and non-binding occurrences of variables:

Variable occurrence	Details
`x = 1`	Binding: restricts `x` to the value `1`
`x != 1`, `not x = 1`	Not binding
`x = 2 + 3`, `x + 1 = 3`	Binding
`x in [0 .. 3]`	Binding
`p(x, _)`	Binding, since `p()` is a call to a predicate.
`x = y`, `x = y + 1`	Binding for `x` if and only if the variable `y` is bound. Binding for `y` if and only if the variable `x` is bound.
`x = y * 2`	Binding for `x` if the variable `y` is bound. Not binding for `y`.
`x > y`	Not binding for `x` or `y`
`"string".matches(x)`	Not binding for `x`
`x.matches(y)`	Not binding for `x` or `y`
`not (... x ...)`	Generally non-binding for `x`, since negating a binding occurrence typically makes it non-binding. There are certain exceptions: `not not x = 1` is correctly recognized as binding for `x`.
`sum(int y | y = 1 and x = y | y)`	Not binding for `x`. `strictsum(int y | y = 1 and x = y | y)` would be binding for `x`. Expressions in the body of an *aggregate* are only binding outside of the body if the aggregate is *strict*.
`x = 1 or y = 1`	Not binding for `x` or for `y`. The first subexpression, `x = 1`, is binding for `x`, and the second subexpression, `y = 1`, is binding for `y`. However, combining them with *disjunction* is only binding for variables for which **all** disjuncts are bindingin this case, thats no variable.

While the occurrence of a variable can be binding or non-binding, the variables property of being bound or unbound is a global concepta single binding occurrence is enough to make a variable bound.

Therefore, you could fix the infinite examples above by providing a binding occurrence. For example, instead of `int timesTwo(int n) { result = n * 2 }`, you could write:

```
int timesTwo(int n) {
  n in [0 .. 10] and
  result = n * 2
}
```

The predicate now binds `n`, and the variable `result` automatically becomes bound by the computation `result = n * 2`.

CHAPTER

FIFTEEN

QL LANGUAGE SPECIFICATION

This is a formal specification for the QL language. It provides a comprehensive reference for terminology, syntax, and other technical details about QL.

15.1 Introduction

QL is a query language for CodeQL databases. The data is relational: named relations hold sets of tuples. The query language is a dialect of Datalog, using stratified semantics, and it includes object-oriented classes.

15.2 Notation

This section describes the notation used in the specification.

15.2.1 Unicode characters

Unicode characters in this document are described in two ways. One is to supply the character inline in the text, between double quote marks. The other is to write a capital U, followed by a plus sign, followed by a four-digit hexadecimal number representing the characters code point. As an example of both, the first character in the name QL is Q (U+0051).

15.2.2 Grammars

The syntactic forms of QL constructs are specified using a modified Backus-Naur Form (BNF). Syntactic forms, including classes of tokens, are named using bare identifiers. Quoted text denotes a token by its exact sequence of characters in the source code.

BNF derivation rules are written as an identifier naming the syntactic element, followed by `::=`, followed by the syntax itself.

In the syntax itself, juxtaposition indicates sequencing. The vertical bar (`|`, U+007C) indicates alternate syntax. Parentheses indicate grouping. An asterisk (`*`, U+002A) indicates repetition zero or more times, and a plus sign (`+`, U+002B) indicates repetition one or more times. Syntax followed by a question mark (`?`, U+003F) indicates zero or one occurrences of that syntax.

15.3 Architecture

A *QL program* consists of a query module defined in a QL file and a number of library modules defined in QLL files that it imports (see *Import directives*). The module in the QL file includes one or more queries (see *Queries*). A module may also include *import directives* (see *Import directives*), non-member predicates (see *Non-member predicates*), class definitions (see *Classes*), and module definitions (see *Modules*).

QL programs are interpreted in the context of a *database* and a *library path* . The database provides a number of definitions: database types (see *Types*), entities (see *Values*), built-in predicates (see *Built-ins*), and the *database content* of built-in predicates and external predicates (see *Evaluation*). The library path is a sequence of file-system directories that hold QLL files.

A QL program can be *evaluated* (see *Evaluation*) to produce a set of tuples of values (see *Values*).

For a QL program to be *valid*, it must conform to a variety of conditions that are described throughout this specification; otherwise the program is said to be *invalid*. An implementation of QL must detect all invalid programs and refuse to

evaluate them.

15.4 Library path

The library path is an ordered list of directory locations. It is used for resolving module imports (see *Module resolution*). The library path is not strictly speaking a core part of the QL language, since different implementations of QL construct it in slightly different ways. Most QL tools also allow you to explicitly specify the library path on the command line for a particular invocation, though that is rarely done, and only useful in very special situations. This section describes the default construction of the library path.

First, determine the *query directory* of the `.ql` file being compiled. Starting with the directory containing the `.ql` file, and walking up the directory structure, each directory is checked for a file called `queries.xml` or `qlpack.yml`. The first directory where such a file is found is the query directory. If there is no such directory, the directory of the `.ql` file itself is the query directory.

A `queries.xml` file that defines a query directory must always contain a single top-level tag named `queries`, which has a `language` attribute set to the identifier of the active database schema (for example, `<queries language="java"/>`).

A `qlpack.yml` file defines a QL pack. The content of a `qlpack.yml` file is described in the CodeQL CLI documentation. This file will not be recognized when using legacy tools that are not based on the CodeQL CLI (that is, LGTM.com, LGTM Enterprise, ODASA, CodeQL for Eclipse, and CodeQL for Visual Studio).

If both a `queries.xml` and a `qlpack.yml` exist in the same directory, the latter takes precedence (and the former is assumed to exist for compatibility with older tooling).

In legacy QL tools that dont recognize `qlpack.yml` files, the default value of the library path for each supported language is hard-coded. The tools contain directories within the ODASA distribution that define the default CodeQL libraries for the selected language. Which language to use depends on the `language` attribute of the `queries.xml` file if not overridden with a `--language` option to the ODASA CLI.

On the other hand, the CodeQL CLI and newer tools based on it (such as GitHub Code Scanning and the CodeQL extension for Visual Studio Code) construct a library path using QL packs. For each QL pack added to the library path, the QL packs named in its `libraryPathDependencies` will be subsequently added to the library path, and the process continues until all packs have been resolved. The actual library path consists of the root directories of the selected QL packs. This process depends on a mechanism for finding QL packs by pack name, as described in the CodeQL CLI documentation.

When the query directory contains a `queries.xml` file but no `qlpack.yml`, the QL pack resolution behaves as if it defines a QL pack with no name and a single library path dependency named `legacy-libraries-LANGUAGE` where `LANGUAGE` is taken from `queries.xml`. The `github/codeql` repository provides packs with names following this pattern, which themselves depend on the actual CodeQL libraries for each language.

When the query directory contains neither a `queries.xml` nor `qlpack.yml` file, it is considered to be a QL pack with no name and no library dependencies. This causes the library path to consist of *only* the query directory itself. This is not generally useful, but it suffices for running toy examples of QL code that dont use information from the database.

15.5 Name resolution

All modules have three environments that dictate name resolution. These are multimaps of keys to declarations.

The environments are:

- The *module environment*, whose keys are module names and whose values are modules.
- The *type environment*, whose keys are type names and whose values are types.
- The *predicate environment*, whose keys are pairs of predicate names and arities and whose values are predicates.

If not otherwise specified, then the environment for a piece of syntax is the same

as the environment of its enclosing syntax.

When a key is resolved in an environment, if there is no value for that key, then the program is invalid.

Environments may be combined as follows:

- *Union*. This takes the union of the entry sets of the two environments.
- *Overriding union*. This takes the union of two environments, but if there are entries for a key in the first map, then no additional entries for that key are included from the second map.

A *definite* environment has at most one entry for each key. Resolution is unique in a definite environment.

15.5.1 Global environments

The global module environment is empty.

The global type environment has entries for the primitive types `int`, `float`, `string`, `boolean`, and `date`, as well as any types defined in the database schema.

The global predicate environment includes all the built-in classless predicates, as well as any extensional predicates declared in the database schema.

The program is invalid if any of these environments is not definite.

15.5.2 Module environments

For each of modules, types, and predicates, a module *imports*, *declares*, and *exports* an environment. These are defined as follows (with X denoting the type of entity we are currently considering):

- The *imported X environment* of a module is defined to be the union of the exported X environments of all the modules which the current module directly imports (see *Import directives*).
- The *declared X environment* of a module is the multimap of X declarations in the module itself.

- The *exported X environment* of a module is the union of the exported X environments of the modules which the current module directly imports (excluding `private` imports), and the declared X environment of the current module (excluding `private` declarations).
- The *external X environment* of a module is the visible X environment of the enclosing module, if there is one, and otherwise the global X environment.
- The *visible X environment* is the union of the imported X environment, the declared X environment, and the external X environment.

The program is invalid if any of these environments is not definite.

Module definitions may be recursive, so the module environments are defined as the least fixed point of the operator given by the above definition. Since all the operations involved are monotonic, this fixed point exists and is unique.

15.6 Modules

15.6.1 Module definitions

A QL module definition has the following syntax:

```
module ::= annotation* "module" modulename "{" moduleBody "}"

moduleBody ::= (import | predicate | class | module | alias | select)*
```

A module definition extends the current modules declared module environment with a mapping from the module name to the module definition.

QL files consist of simply a module body without a name and surrounding braces:

```
ql ::= moduleBody
```

QL files define a module corresponding to the file, whose name is the same as the filename.

15.6.2 Kinds of modules

A module may be:

- A *file module*, if it is defined implicitly by a QL file.
- A *query module*, if it is defined by a QL file.
- A *library module*, if it is not a query module.

A query module must contain one or more queries.

15.6.3 Import directives

An import directive refers to a module identifier:

```
import ::= annotations "import" importModuleId ("as" modulename)?

qualId ::= simpleId | qualId "." simpleId

importModuleId ::= qualId
                 | importModuleId "::" simpleId
```

An import directive may optionally name the imported module using an `as` declaration. If a name is defined, then the import directive adds to the declared module environment of the current module a mapping from the name to the declaration of the imported module. Otherwise, the current module *directly imports* the imported module.

15.6.4 Module resolution

Module identifiers are resolved to modules as follows.

For simple identifiers:

- First, the identifier is resolved as a one-segment qualified identifier (see below).
- If this fails, the identifier is resolved in the current modules visible module environment.

For selection identifiers (`a::b`):

- The qualifier of the selection (`a`) is resolved as a module, and then the name (`b`) is resolved in the exported module environment of the qualifier module.

For qualified identifiers (`a.b`):

- Build up a list of *candidate search paths*, consisting of the current files directory, then the *query directory* of the current file, and finally each of the directories on the *library path* (in order).
- Determine the first candidate search path that has a *matching* QLL file for the import directives qualified name. A QLL file in a candidate search path is said to match a qualified name if, starting from the candidate search path, there is a subdirectory for each successive qualifier in the qualified name, and the directory named by the final qualifier contains a file whose base name matches the qualified names base name, with the addition of the file extension `.qll`. The file and directory names are matched case-sensitively, regardless of whether the filesystem is case-sensitive or not.
- The resolved module is the module defined by the selected candidate search path.

A qualified module identifier is only valid within an import.

15.6.5 Module references and active modules

A module `M` *references* another module `N` if any of the following holds:

- `M` imports `N`.
- `M` defines `N`.
- `N` is `M`s enclosing module.

In a QL program, the *active* modules are the modules which are referenced transitively by the query module.

15.7 Types

QL is a typed language. This section specifies the kinds of types available, their attributes, and the syntax for referring to them.

15.7.1 Kinds of types

Types in QL are either *primitive* types, *database* types, *class* types, *character* types or *class domain* types.

The primitive types are `boolean`, `date`, `float`, `int`, and `string`.

Database types are supplied as part of the database. Each database type has a *name*, which is an identifier starting with an at sign (`@`, U+0040) followed by lower-case letter. Database types have some number of *base types*, which are other database types. In a valid database, the base types relation is non-cyclic.

Class types are defined in QL, in a way specified later in this document (see *Classes*). Each class type has a name that is an identifier starting with an upper-case letter. Each class type has one or more base types, which can be any kind of type except a class domain type. A class type may be declared *abstract*.

Any class in QL has an associated class domain type and an associated character type.

Within the specification the class type for `C` is written `C.class`, the character type is written `C.C` and the domain type is written `C.extends`. However the class type is still named `C`.

15.7.2 Type references

With the exception of class domain types and character types (which cannot be referenced explicitly in QL source), a reference to a type is written as the name of the type. In the case of database types, the name includes the at sign (`@`, U+0040).

```
type ::= (moduleId "::")? classname | dbasetype | "boolean" | "date" |
↪"float" | "int" | "string"

moduleId ::= simpleId | moduleId "::" simpleId
```

A type reference is resolved to a type as follows:

- If it is a selection identifier (for example, `a::B`), then the qualifier (`a`) is resolved as a module (see *Module resolution*). The identifier (`B`) is then resolved in the exported type environment of the qualifier module.
- Otherwise, the identifier is resolved in the current modules visible type environment.

15.7.3 Relations among types

Types are in a subtype relationship with each other. Type A is a *subtype* of type B if one of the following is true:

- A and B are the same type.
- There is some type C, where A is a subtype of C and C is a subtype of B.
- A and B are database types, and B is a base type of A.
- A is the character type of C, and B is the class domain type of C.
- A is a class type, and B is the character type of A.
- A is a class domain type, and B is a base type of the associated class type.
- A is `int` and B is `float`.

Supertypes are the converse of subtypes: A is a *supertype* of B if B is a subtype of A.

Types A and B are *compatible* with each other if they either have a common supertype, or they each have some supertype that is a database type.

15.7.4 Typing environments

A *typing environment* is a finite map of variables to types. Each variable in the map is either an identifier or one of two special symbols: `this`, and `result`.

Most forms of QL syntax have a typing environment that applies to them. That typing environment is determined by the context the syntax appears in.

Note that this is distinct from the type environment, which is a map from type names to types.

15.7.5 Active types

In a QL program, the *active* types are those defined in active modules. In the remainder of this specification, any reference to the types in the program refers only to the active types.

15.8 Values

Values are the fundamental data that QL programs compute over. This section specifies the kinds of values available in QL, specifies the sorting order for them, and describes how values can be combined into tuples.

15.8.1 Kinds of values

There are six kinds of values in QL: one kind for each of the five primitive types, and *entities*. Each value has a type.

A boolean value is of type `boolean`, and may have one of two distinct values: `true` or `false`.

A date value is of type `date`. It encodes a time and a date in the Gregorian calendar. Specifically, it includes a year, a month, a day, an hour, a minute, a second, and a millisecond, each of which are integers. The year ranges from -16777216 to 16777215, the month from 0 to 11, the day from 1 to 31, the hour from 0 to 23, the minutes from 0 to 59, the seconds from 0 to 59, and the milliseconds from 0 to 999.

A float value is of type `float`. Each float value is a binary 64-bit floating-point value as specified in IEEE 754.

An integer value is of type `int`. Each value is a 32-bit twos complement integer.

A string is a finite sequence of 16-bit characters. The characters are interpreted as Unicode code points.

The database includes a number of opaque entity values. Each such value has a type that is one of the database types, and an identifying integer. An entity value is written as the name of its database type followed by its identifying integer in parentheses. For example, `@tree(12)`, `@person(16)`, and `@location(38132)` are entity values. The identifying integers are left opaque to programmers in this specification, so an implementation of QL is free to use some other set of countable labels to identify its entities.

15.8.2 Ordering

Values in general do not have a specified ordering. In particular, entity values have no specified ordering with entities or any other values. Primitive values, however, have a total ordering with other primitive values in the same type. Primitives types and their subtypes are said to be *orderable*.

For booleans, `false` is ordered before `true`.

For dates, the ordering is chronological.

For floats, the ordering is as specified in IEEE 754 when one exists, except that NaN is considered equal to itself and is ordered after all other floats, and negative zero is considered to be strictly less than positive zero.

For integers, the ordering is as for twos complement integers.

For strings, the ordering is lexicographic.

15.8.3 Tuples

Values can be grouped into tuples in two different ways.

An *ordered tuple* is a finite, ordered sequence of values. For example, (`1`, `2`, `"three"`) is an ordered sequence of two integers and a string.

A *named tuple* is a finite map of variables to values. Each variable in a named tuple is either an identifier, `this`, or `result`.

A *variable declaration list* provides a sequence of variables and a type for each one:

```
var_decls ::= var_decl ("," var_decl)*
var_decl ::= type simpleId
```

A valid variable declaration list must not include two declarations with the same variable name. Moreover, if the declaration has a typing environment that applies, it must not use a variable name that is already present in that typing environment.

An *extension* of a named tuple for a given variable declaration list is a named tuple that additionally maps each variable in the list to a value. The value mapped by each new variable must be in the type that is associated with that variable in the given list; see *The store* for the definition of a value being in a type.

15.9 The store

QL programs evaluate in the context of a *store*. This section specifies several definitions related to the store.

A *fact* is a predicate or type along with an ordered tuple. A fact is written as the predicate name or type name followed immediately by the tuple. Here are some examples of facts:

```
successor(0, 1)
Tree.toString(@method_tree(12), "def println")
Location.class(@location(43))
Location.getURL(@location(43), "file:///etc/hosts:2:0:2:12")
```

A *store* is a mutable set of facts. The store can be mutated by adding more facts to it.

An ordered tuple *directly satisfies* a predicate or type with a given if there is a fact in the store with the given tuple and predicate or type.

A value `v` is in a type `t` under any of the following conditions:

- The type of `v` is `t` and `t` is a primitive type.
- The tuple `(v)` directly satisfies `t`.

An ordered tuple *satisfies a predicate* `p` under the following circumstances. If `p` is not a member predicate, then the tuple satisfies the predicate whenever it directly satisfies the predicate.

Otherwise, the tuple must be the tuple of a fact in the store with predicate `q`, where `q` shares a root definition with `p`. The first element of the tuple must be in the type before the dot in `q`, and there must be no other predicate that overrides `q` such that this is true (see *Classes* for details on overriding and root definitions).

An ordered tuple `(a0, an)` satisfies the `+` closure of a predicate if there is a sequence of binary tuples `(a0, a1)`, `(a1, a2)`, , `(an-1, an)` that all satisfy the predicate. An ordered tuple `(a, b)` satisfies the `*` closure of a predicate if it either satisfies the `+` closure, or if `a` and `b` are the same, and if moreover they are in each argument type of the predicate.

15.10 Lexical syntax

QL and QLL files contain a sequence of *tokens* that are encoded as Unicode text. This section describes the tokenization algorithm, the kinds of available tokens, and their representation in Unicode.

Some kinds of tokens have an identifier given in parentheses in the section title. That identifier, if present, is a terminal used in grammar productions later in the specification. Additionally, the *Identifiers* section gives several kinds of identifiers, each of which has its own grammar terminal.

15.10.1 Tokenization

Source files are interpreted as a sequence of tokens according to the following algorithm. First, the longest-match rule, described below, is applied starting at the beginning of the file. Second, all whitespace tokens and comments are discarded from the sequence.

The longest-match rule is applied as follows. The first token in the file is the longest token consisting of a contiguous sequence of characters at the beginning of the file. The next token after any other token is the longest token consisting of contiguous characters that immediately follow any previous token.

If the file cannot be tokenized in its entirety, then the file is invalid.

15.10.2 Whitespace

A whitespace token is a sequence of spaces (U+0020), tabs (U+0009), carriage returns (U+000D), and line feeds (U+000A).

15.10.3 Comments

There are two kinds of comments in QL: one-line and multiline.

A one-line comment is two slash characters (/, U+002F) followed by any sequence of characters other than line feeds (U+000A) and carriage returns (U+000D). Here is an example of a one-line comment:

```
// This is a comment
```

A multiline comment is a *comment start*, followed by a *comment body*, followed by a *comment end*. A comment start is a slash (/, U+002F) followed by an asterisk (*, U+002A), and a comment end is an asterisk followed by a slash. A comment body is any sequence of characters that does not include a comment end. Here is an example multiline comment:

```
/*
  It was the best of code.
```

(continues on next page)

(continued from previous page)

```
  It was the worst of code.
  It had a multiline comment.
*/
```

15.10.4 Keywords

The following sequences of characters are keyword tokens:

```
and
any
as
asc
avg
boolean
by
class
concat
count
date
desc
else
exists
extends
false
float
forall
forex
from
if
implies
import
in
instanceof
int
max
min
```

(continues on next page)

(continued from previous page)

```
module
none
not
or
order
predicate
rank
result
select
strictconcat
strictcount
strictsum
string
sum
super
then
this
true
where
```

15.10.5 Operators

The following sequences of characters are operator tokens:

```
<
<=
=
>
>=

_
-

,
;
!=
/
.
```

(continues on next page)

(continued from previous page)

```
..
(
)
[
]
{
}
*
%
+
|
```

15.10.6 Identifiers

An identifier is an optional @ sign (U+0040) followed by a sequence of identifier characters. Identifier characters are lower-case ASCII letters (`a` through `z`, U+0061 through U+007A), upper-case ASCII letters (`A` through `Z`, U+0041 through U+005A), decimal digits (`0` through `9`, U+0030 through U+0039), and underscores (`_`, U+005F). The first character of an identifier other than any @ sign must be a letter.

An identifier cannot have the same sequence of characters as a keyword, nor can it be an @ sign followed by a keyword.

Here are some examples of identifiers:

```
width
Window_width
window5000_mark_II
@expr
```

There are several kinds of identifiers:

- `lowerId`: an identifier that starts with a lower-case letter.
- `upperId`: an identifier that starts with an upper-case letter.

- `atLowerId`: an identifier that starts with an @ sign and then a lower-case letter.
- `atUpperId`: an identifier that starts with an @ sign and then an upper-case letter.

Identifiers are used in following syntactic constructs:

```
simpleId      ::= lowerId | upperId
modulename    ::= simpleId
classname     ::= upperId
dbasetype     ::= atLowerId
predicateRef  ::= (moduleId "::")? literalId
predicateName ::= lowerId
varname       ::= simpleId
literalId     ::= lowerId | atLowerId
```

15.10.7 Integer literals (int)

An integer literal is a possibly negated sequence of decimal digits (`0` through `9`, U+0030 through U+0039). Here are some examples of integer literals:

```
0
42
123
-2147483648
```

15.10.8 Float literals (float)

A floating-point literals is a possibly negated two non-negative integers literals separated by a dot (`.`, U+002E). Here are some examples of float literals:

```
0.5
2.0
123.456
-100.5
```

15.10.9 String literals (string)

A string literal denotes a sequence of characters. It begins and ends with a double quote character (U+0022). In between the double quotes are a sequence of string character indicators, each of which indicates one character that should be included in the string. The string character indicators are as follows.

- Any character other than a double quote (U+0022), backslash (U+005C), line feed (U+000A), carriage return (U+000D), or tab (U+0009). Such a character indicates itself.
- A backslash (U+005C) followed by one of the following characters:
  - Another backslash (U+005C), in which case a backslash character is indicated.
  - A double quote (U+0022), in which case a double quote is indicated.
  - The letter n (U+006E), in which case a line feed (U+000A) is indicated.
  - The letter r (U+0072), in which case a carriage return (U+000D) is indicated.
  - The letter t (U+0074), in which case a tab (U+0009) is indicated.

Here are some examples of string literals:

```
"hello"
"He said, \"Logic clearly dictates that the needs of the many...\""
```

15.11 Annotations

Various kinds of syntax can have *annotations* applied to them. Annotations are as follows:

```
annotations ::= annotation*

annotation ::= simpleAnnotation | argsAnnotation
```

(continues on next page)

(continued from previous page)

```
simpleAnnotation ::= "abstract"
                   |   "cached"
                   |   "external"
                   |   "final"
                   |   "transient"
                   |   "library"
                   |   "private"
                   |   "deprecated"
                   |   "override"
                   |   "query"

argsAnnotation ::= "pragma" "[" ("inline" | "noinline" | "nomagic" |
 ↪"noopt") "]"
                |   "language" "[" "monotonicAggregates" "]"
                |   "bindingset" "[" (variable ( "," variable)*)? "]"
```

Each simple annotation adds a same-named attribute to the syntactic entity it precedes. For example, if a class is preceded by the `abstract` annotation, then the class is said to be abstract.

A valid annotation list may not include the same simple annotation more than once, or the same parameterized annotation more than once with the same arguments. However, it may include the same parameterized annotation more than once with different arguments.

15.11.1 Simple annotations

The following table summarizes the syntactic constructs which can be marked with each annotation in a valid program; for example, an `abstract` annotation preceding a character is invalid.

Annotation	Classes	Characters	Member predicates	Non-member predicates	Imports	Fields	Modules	Aliases
`abstract`	yes		yes					
`cached`	yes	yes	yes	yes			yes	
`external`				yes				
`final`	yes		yes			yes		
`transient`				yes				
`library`	yes							
`private`	yes		yes	yes	yes	yes	yes	yes
`deprecated`	yes		yes	yes		yes	yes	yes
`override`			yes			yes		
`query`				yes				yes

The `library` annotation is only usable within a QLL file, not a QL file.

Annotations on aliases apply to the name introduced by the alias. An alias may, for example, have different privacy to the name it aliases.

15.11.2 Parameterized annotations

Parameterized annotations take some additional arguments.

The parameterized annotation `pragma` supplies compiler pragmas, and may be applied in various contexts depending on the pragma in question.

Pragma	Classes	Characters	Member predicates	Non-member predicates	Imports	Fields	Modules	Aliases
`inline`		yes	yes	yes				
`noinline`		yes	yes	yes				
`nomagic`		yes	yes	yes				
`noopt`		yes	yes	yes				

The parameterized annotation `language` supplies language pragmas which change the behavior of the language. Language pragmas apply at the scope level, and are inherited by nested scopes.

Pragma	Classes	Characters	Member predicates	Non-member predicates	Imports	Fields	Modules	Aliases
`monotonicAggregates`	yes	yes	yes	yes			yes	

A binding set for a predicate is a subset of the predicates arguments such that if those arguments are bound (restricted to a finite range of values), then all of the predicates arguments are bound.

The parameterized annotation `bindingset` can be applied to a predicate (see *Non-member predicates* and *Members*) to specify a binding set.

This annotation accepts a (possibly empty) list of variable names as parameters. The named variables must all be arguments of the predicate, possibly including `this` for characteristic predicates and member predicates, and `result` for predicates that yield a result.

In the default case where no binding sets are specified by the user, then it is assumed that there is precisely one, empty binding set - that is, the body of the predicate must bind all the arguments.

Binding sets are checked by the QL compiler in the following way:

1. It assumes that all variables mentioned in the binding set are bound.
2. It checks that, under this assumption, all the remaining argument variables are bound by the predicate body.

A predicate may have several different binding sets, which can be stated by using multiple `bindingset` annotations on the same predicate.

Pragma	Classes	Characters	Member predicates	Non-member predicates	Imports	Fields	Modules	Aliases
`bindingset`		yes	yes	yes				

15.12 Top-level entities

Modules include five kinds of top-level entity: predicates, classes, modules, aliases, and select clauses.

15.12.1 Non-member predicates

A *predicate* is declared as a sequence of annotations, a head, and an optional body:

```
predicate ::= annotations head optbody
```

A predicate definition adds a mapping from the predicate name and arity to the predicate declaration to the current modules declared predicate environment.

When a predicate is a top-level clause in a module, it is called a non-member predicate. See below for *member predicates*.

A valid non-member predicate can be annotated with `cached`, `deprecated`, `external`, `transient`, `private`, and `query`. Note, the `transient` annotation can only be applied if the non-member predicate is also annotated with `external`.

The head of the predicate gives a name, an optional *result type*, and a sequence of variables declarations that are *arguments*:

```
head ::= ("predicate" | type) predicateName "(" (var_decls)? ")"
```

The body of a predicate is of one of three forms:

```
optbody ::= ";"
        |  "{" formula "}"
        |  "=" literalId "(" (predicateRef "/" int ("," predicateRef "/
↪" int)*)? ")" "(" (exprs)? ")"
```

In the first form, with just a semicolon, the predicate is said to not have a body. In the second form, the body of the predicate is the given formula (see *Formulas*). In the third form, the body is a higher-order relation.

A valid non-member predicate must have a body, either a formula or a higher-order relation, unless it is external, in which case it must not have a body.

The typing environment for the body of the formula, if present, maps the variables in the head of the predicate to their associated types. If the predicate has a result type, then the typing environment also maps `result` to the result type.

15.12.2 Classes

A class definition has the following syntax:

```
class ::= annotations "class" classname "extends" type ("," type)* "{"␣
↪member* "}"
```

The identifier following the `class` keyword is the name of the class.

The types specified after the `extends` keyword are the *base types* of the class.

A class domain type is said to *inherit* from the base types of the associated class type, a character type is said to *inherit* from its associated class domain type and a class type is said to *inherit* from its associated character type. In addition, inheritance is transitive: If a type `A` inherits from a type `B`, and `B` inherits from a type `C`, then `A` inherits from `C`.

A class adds a mapping from the class name to the class declaration to the current modules declared type environment.

A valid class can be annotated with `abstract`, `final`, `library`, and `private`. Any other annotation renders the class invalid.

A valid class may not inherit from a final class, from itself, or from more than one primitive type.

15.12.3 Class environments

For each of modules, types, predicates, and fields a class *inherits*, *declares*, and *exports* an environment. These are defined as follows (with X denoting the type of entity we are currently considering):

- The *inherited X environment* of a class is the union of the exported X environments of its base types.
- The *declared X environment* of a class is the multimap of X declarations in the class itself.
- The *exported X environment* of a class is the overriding union of its declared X environment (excluding `private` declaration entries) with its inherited X environment.
- The *external X environment* of a class is the visible X environment of the enclosing module.
- The *visible X environment* is the overriding union of the declared X environment and the inherited X environment; overriding unioned with the external X environment.

The program is invalid if any of these environments is not definite.

15.12.4 Members

Each member of a class is either a *character*, a predicate, or a field:

```
member ::= character | predicate | field
character ::= annotations classname "(" ")" "{" formula "}"
field ::= annotations var_decl ";"
```

Characters

A valid character must have the same name as the name of the class. A valid class has at most one character provided in the source code.

A valid character can be annotated with `cached`. Any other annotation renders the character invalid.

Member predicates

A predicate that is a member of a class is called a *member predicate*. The name of the predicate is the identifier just before the open parenthesis.

A member predicate adds a mapping from the predicate name and arity to the predicate declaration in the classs declared predicate environment.

A valid member predicate can be annotated with `abstract`, `cached`, `final`, `private`, `deprecated`, and `override`.

If a type is provided before the name of the member predicate, then that type is the *result type* of the predicate. Otherwise, the predicate has no result type. The types of the variables in the `var_decls` are called the predicates *argument types*.

A member predicate `p` with enclosing class `C` *overrides* a member predicate `p'` with enclosing class `D` when `C` inherits from `D`, `p'` is visible in `C`, and both `p` and `p'` have the same name and the same arity. An overriding predicate must have the same sequence of argument types as any predicates which it overrides, otherwise the program is invalid.

Member predicates have one or more *root definitions*. If a member predicate overrides no other member predicate, then it is its own root definition. Otherwise, its root definitions are those of any member predicate that it overrides.

A valid member predicate must have a body unless it is abstract or external, in which case it must not have a body.

A valid member predicate must override another member predicate if it is annotated override.

When member predicate `p` overrides member predicate `q`, either `p` and `q` must both have a result type, or neither of them may have a result type. If they do have result types, then the result type of `p` must be a subtype of the result type of `q`. `q` may not be a final predicate. If `p` is abstract, then `q` must be as well.

A class may not inherit from a class with an abstract member predicate unless it either includes a member predicate overriding that abstract predicate, or it inherits from another class that does.

A valid class must include a non-private predicate named `toString` with no arguments and a result type of `string`, or it must inherit from a class that does.

A valid class may not inherit from two different classes that include a predicate with the same name and number of arguments, unless either one of the predicates

overrides the other, or the class defines a predicate that overrides both of them.

The typing environment for a member predicate or character is the same as if it were a non-member predicate, except that it additionally maps `this` to a type. If the member is a character, then the typing environment maps `this` to the class domain type of the class. Otherwise, it maps `this` to the class type of the class itself.

Fields

A field declaration introduces a mapping from the field name to the field declaration in the classs declared field environment.

15.12.5 Select clauses

A QL file may include at most one *select clause*. That select clause has the following syntax:

```
select ::= ("from" var_decls)? ("where" formula)? "select" select_
 ↪exprs ("order" "by" orderbys)?
```

A valid QLL file may not include any select clauses.

A select clause is considered to be a declaration of an anonymous predicate whose arguments correspond to the select expressions of the select clause.

The `from` keyword, if present, is followed by the *variables* of the formula. Otherwise, the select clause has no variables.

The `where` keyword, if present, is followed by the *formula* of the select clause. Otherwise, the select clause has no formula.

The `select` keyword is followed by a number of *select expressions*. Select expressions have the following syntax:

```
as_exprs ::= as_expr ("," as_expr)*
as_expr ::= expr ("as" simpleId)?
```

The keyword `as` gives a *label* to the select expression it is part of. No two select expressions may have the same label. No expression label may be the same as one of the variables of the select clause.

The `order` keyword, if present, is followed by a number of *ordering directives*. Ordering directives have the following syntax:

```
orderbys ::= orderby ("," orderby)*
orderby ::= simpleId ("asc" | "desc")?
```

Each identifier in an ordering directive must identify exactly one of the select expressions. It must either be the label of the expression, or it must be a variable expression that is equivalent to exactly one of the select expressions. The type of the designated select expression must be a subtype of a primitive type.

No select expression may be specified by more than one ordering directive. See *Ordering* for more information.

15.12.6 Queries

The queries in a QL module are:

- The select clause, if any, defined in that module.
- Any predicates annotated with `query` which are in scope in that module.

The target predicate of the query is either the select clause or the annotated predicate.

Each argument of the target predicate of the query must be of a type which has a `toString()` member predicate.

15.13 Expressions

Expressions are a form of syntax used to denote values. Every expression has a typing environment that is determined by the context where the expression occurs. Every valid expression has a type, as specified in this section, except if it is a dont-care expression.

Given a named tuple and a store, each expression has one or more *values*. This section specifies the values of each kind of expression.

There are several kinds of expressions:

```
exprs ::= expr ("," expr)*

expr ::= dontcare
     |   unop
     |   binop
     |   cast
     |   primary

primary ::= eparen
        |   literal
        |   variable
        |   super_expr
        |   callwithresult
        |   postfix_cast
        |   aggregation
        |   any
```

15.13.1 Parenthesized expressions

A parenthesized expression is an expression surrounded by parentheses:

```
eparen ::= "(" expr ")"
```

The type environment of the nested expression is the same as that of the outer expression. The type and values of the outer expression are the same as those of the nested expression.

15.13.2 Dont-care expressions

A dont-care expression is written as a single underscore:

```
dontcare ::= "_"
```

All values are values of a dont-care expression.

15.13.3 Literals

A literal expression is as follows:

```
literal ::= "false" | "true" | int | float | string
```

The type of a literal expression is the type of the value denoted by the literal: `boolean` for `false` or `true`, `int` for an integer literal, `float` for a floating-point literal, or `string` for a string literal. The value of a literal expression is the same as the value denoted by the literal.

15.13.4 Unary operations

A unary operation is the application of + or − to another expression:

```
unop ::= "+" expr
     |   "-" expr
```

The + or − in the operation is called the *operator*, and the expression is called the *operand*. The typing environment of the operand is the same as for the unary operation.

For a valid unary operation, the operand must be of type `int` or `float`. The operation has the same type as its operand.

If the operator is +, then the values of the expression are the same as the values of the operand. If the operator is −, then the values of the expression are the arithmetic negations of the values of the operand.

15.13.5 Binary operations

A binary operation is written as a *left operand* followed by a *binary operator*, followed by a *right operand*:

```
binop ::= expr "+" expr
      |   expr "-" expr
      |   expr "*" expr
      |   expr "/" expr
      |   expr "%" expr
```

The typing environment for the two environments is the same as for the operation. If the operator is +, then either both operands must be subtypes of `int` or `float`, or at least one operand must be a subtype of `string`. If the operator is anything else, then each operand must be a subtype of `int` or `float`.

The type of the operation is `string` if either operand is a subtype of `string`. Otherwise, the type of the operation is `int` if both operands are subtypes of `int`. Otherwise, the type of the operation is `float`.

If the result is of type `string`, then the *left values* of the operation are the values of a call with results expression with the left operand as the receiver, `toString` as the predicate name, and no arguments (see *Calls with results*). Otherwise the left values are the values of the left operand. Likewise, the *right values* are either the values from calling `toString` on the right operand, or the values of the right operand as it is.

The binary operation has one value for each combination of a left value and a right value. That value is determined as follows:

- If the left and right operand types are subtypes of string, then the operation has a value that is the concatenation of the left and right values.
- Otherwise, if both operand types are subtypes of `int`, then the value of the operation is the result of applying the twos-complement 32-bit integer operation corresponding to the QL binary operator.
- Otherwise, both operand types must be subtypes of `float`. If either operand is of type `int` then they are converted to a float. The value of the operation is then the result of applying the IEEE 754 floating-point operator that corresponds to the QL binary operator: addition for +, subtraction for -, multiplication for *, division for /, or remainder for %.

15.13.6 Variables

A variable has the following syntax:

```
variable ::= varname | "this" | "result"
```

A valid variable expression must occur in the typing environment. The type of the variable expression is the same as the type of the variable in the typing environment.

The value of the variable is the value of the variable in the named tuple.

15.13.7 Super

A super expression has the following syntax:

```
super_expr ::= "super" | type "." "super"
```

For a super expression to be valid, the `this` keyword must have a type and value in the typing environment. The type of the expression is the same as the type of `this` in the typing environment.

A super expression may only occur in a QL program as the receiver expression for a predicate call.

If a super expression includes a `type`, then that type must be a class that the enclosing class inherits from.

If the super expression does not include a type, then the enclosing class must have a single declared base type, and that base type must be a class.

The value of a super expression is the same as the value of `this` in the named tuple.

15.13.8 Casts

A cast expression is a type in parentheses followed by another expression:

```
cast ::= "(" type ")" expr
```

The typing environment for the nested expression is the same as for the cast expression. The type of the cast expression is the type between parentheses.

The values of the cast expression are those values of the nested expression that are in the type given within parentheses.

For casts between the primitive `float` and `int` types, the above rule means that for the cast expression to have a value, it must be representable as both 32-bit twos complement integers and 64-bit IEEE 754 floats. Other values will not be included in the values of the cast expression.

15.13.9 Postfix casts

A postfix cast is a primary expression followed by a dot and then a class or primitive type in parentheses:

```
postfix_cast ::= primary "." "(" type ")"
```

All the rules for ordinary casts apply to postfix casts: a postfix cast is exactly equivalent to a parenthesized ordinary cast.

15.13.10 Calls with results

An expression for a call with results is of one of two forms:

```
callwithresult ::= predicateRef (closure)? "(" (exprs)? ")"
                |   primary "." predicateName (closure)? "(" (exprs)? ")
 ↪"
closure        ::= "*" | "+"
```

The expressions in parentheses are the *arguments* of the call. The expression before the dot, if there is one, is the *receiver* of the call.

The type environment for the arguments is the same as for the call.

A valid call with results must *resolve* to exactly one predicate. The ways a call can resolve are as follows:

- If the call has no receiver, then it can resolve to a non-member predicate. If the predicate name is a simple identifier, then the predicate is resolved by looking up its name and arity in the visible predicate environment of the enclosing class or module.

  If the predicate name is a selection identifier, then the qualifier is resolved as a module (see *Module resolution*). The identifier is then resolved in the exported predicate environment of the qualifier module.

- If the call has a super expression as the receiver, then it resolves to a member predicate in a class the enclosing class inherits from. If the super expression is unqualified, then the super-class is the single class that the current class inherits from. If there is not exactly one such class, then the program is invalid. Otherwise the super-class is the class named by the qualifier of the super expression. The predicate is resolved by looking up its name and arity in the exported predicate environment of the super-class. If there is more than one such predicate, then the predicate call is not valid.

For each argument other than a dont-care expression, the type of the argument must be compatible with the type of the corresponding argument type of the predicate, otherwise the call is invalid.

A valid call with results must resolve to a predicate that has a result type. That result type is also the type of the call.

If the resolved predicate is built in, then the call may not include a closure. If the call does have a closure, then it must resolve to a predicate where the *relational arity* of the predicate is 2. The relational arity of a predicate is the sum of the following numbers:

- The number of arguments to the predicate.
- The number 1 if the predicate is a member predicate, otherwise 0.
- The number 1 if the predicate has a result, otherwise 0.

If the call resolves to a member predicate, then the *receiver values* are as follows.

If the call has a receiver, then the receiver values are the values of that receiver. If the call does not have a receiver, then the single receiver value is the value of `this` in the contextual named tuple.

The *tuple prefixes* of a call with results include one value from each of the argument expressions values, in the same order as the order of the arguments. If the call resolves to a non-member predicate, then those values are exactly the tuple prefixes of the call. If the call instead resolves to a member predicate, then the tuple prefixes additionally include a receiver value, ordered before the argument values.

The *matching tuples* of a call with results are all ordered tuples that are one of the tuple prefixes followed by any value of the same type as the call. If the call has no closure, then all matching tuples must additionally satisfy the resolved predicate of the call, unless the receiver is a super expression, in which case they must *directly* satisfy the resolved predicate of the call. If the call has a `*` or `+` closure, then the matching tuples must satisfy or directly satisfy the associated closure of the resolved predicate.

The values of a call with results are the final elements of each of the calls matching tuples.

15.13.11 Aggregations

An aggregation can be written in one of two forms:

```
aggregation ::= aggid ("[" expr "]")? "(" (var_decls)? ("|" (formula)?␣
↪("|" as_exprs ("order" "by" aggorderbys)?)?)? ")"
            |   aggid ("[" expr "]")? "(" as_exprs ("order" "by"␣
↪aggorderbys)? ")"
            |   "unique" "(" var_decls "|" (formula)? ("|" as_exprs)?
↪")"

aggid ::= "avg" | "concat" | "count" | "max" | "min" | "rank" |
↪"strictconcat" | "strictcount" | "strictsum" | "sum"

aggorderbys ::= aggorderby ("," aggorderby)*
```

(continues on next page)

(continued from previous page)

```
aggorderby ::= expr ("asc" | "desc")?
```

The expression enclosed in square brackets (`[` and `]`, U+005B and U+005D), if present, is called the *rank expression*. It must have type `int` in the enclosing environment.

The `as_exprs`, if present, are called the *aggregation expressions*. If an aggregation expression is of the form `expr as v` then the expression is said to be *named* v.

The rank expression must be present if the aggregate id is `rank`; otherwise it must not be present.

Apart from the presence or absence of the rank variable, all other reduced forms of an aggregation are equivalent to a full form using the following steps:

- If the formula is omitted, then it is taken to be `any()`.
- If there are no aggregation expressions, then either:
  - The aggregation id is `count` or `strictcount` and the expression is taken to be `1`.
  - There must be precisely one variable declaration, and the aggregation expression is taken to be a reference to that variable.
- If the aggregation id is `concat` or `strictconcat` and it has a single expression then the second expression is taken to be `""`.
- If the `monotonicAggregates` language pragma is not enabled, or the original formula and variable declarations are both omitted, then the aggregate is transformed as follows:
  - For each aggregation expression `expr_i`, a fresh variable `v_i` is declared with the same type as the expression in addition to the original variable declarations.
  - The new range is the conjunction of the original range and a term `v_i = expr_i` for each aggregation expression `expr_i`.

  - Each original aggregation expression `expr_i` is replaced by a new aggregation expression `v_i`.

The variables in the variable declarations list must not occur in the typing environment.

The typing environment for the rank expression is the same as for the aggregation.

The typing environment for the formula is obtained by taking the typing environment for the aggregation and adding all the variable types in the given `var_decls` list.

The typing environment for an aggregation expression is obtained by taking the typing environment for the formula and then, for each named aggregation expression that occurs earlier than the current expression, adding a mapping from the earlier expressions name to the earlier expressions type.

The typing environment for ordering directives is obtained by taking the typing environment for the formula and then, for each named aggregation expression in the aggregation, adding a mapping from the expressions name to the expressions type.

The number and types of the aggregation expressions are restricted as follows:

- A `max`, `min`, `rank` or `unique` aggregation must have a single expression.
- The type of the expression in a `max`, `min` or `rank` aggregation without an ordering directive expression must be an orderable type.
- A `count` or `strictcount` aggregation must not have an expression.
- A `sum`, `strictsum` or `avg` aggregation must have a single aggregation expression, which must have a type which is a subtype of `float`.
- A `concat` or `strictconcat` aggregation must have two expressions. Both expressions must have types which are subtypes of `string`.

The type of a `count`, `strictcount` aggregation is `int`. The type of an `avg` aggregation is `float`. The type of a `concat` or `strictconcat` aggregation is `string`. The type of a `sum` or `strictsum` aggregation is `int` if the aggregation expression is a subtype of `int`, otherwise it is `float`. The type of a `rank`, `min` or `max`

aggregation is the type of the single expression.

An ordering directive may only be specified for a `max`, `min`, `rank`, `concat` or `strictconcat` aggregation. The type of the expression in an ordering directive must be an orderable type.

The values of the aggregation expression are determined as follows. Firstly, the *range tuples* are extensions of the named tuple that the aggregation is being evaluated in with the variable declarations of the aggregation, and which *match* the formula (see *Formulas*).

For each range tuple, the *aggregation tuples* are the extension of the range tuples to *aggregation variables* and *sort variables*.

The aggregation variables are given by the aggregation expressions. If an aggregation expression is named, then its aggregation variable is given by its name, otherwise a fresh synthetic variable is created. The value is given by evaluating the expression with the named tuple being the result of the previous expression, or the range tuple if this is the first aggregation expression.

The sort variables are synthetic variables created for each expression in the ordering directive with values given by the values of the expressions within the ordering directive.

If the aggregation id is `max`, `min` or `rank` and there was no ordering directive, then for each aggregation tuple a synthetic sort variable is added with value given by the aggregation variable.

The values of the aggregation expression are given by applying the aggregation function to each set of tuples obtained by picking exactly one aggregation tuple for each range tuple.

- If the aggregation id is `avg`, and the set is non-empty, then the resulting value is the average of the value for the aggregation variable in each tuple in the set, weighted by the number of tuples in the set, after converting the value to a floating-point number.
- If the aggregation id is `count`, then the resulting value is the number of tuples in the set. If there are no tuples in the set, then the value is the integer `0`.

- If the aggregation id is `max`, then the values are the those values of the aggregation variable which are associated with a maximal tuple of sort values. If the set is empty, then the aggregation has no value.
- If the aggregation id is `min`, then the values are the those values of the aggregation variable which are associated with a minimal tuple of sort values. If the set is empty, then the aggregation has no value.
- If the aggregation id is `rank`, then the resulting values are values of the aggregation variable such that the number of aggregation tuples with a strictly smaller tuple of sort variables is exactly one less than an integer bound by the rank expression of the aggregation. If no such values exist, then the aggregation has no values.
- If the aggregation id is `strictcount`, then the resulting value is the same as if the aggregation id were `count`, unless the set of tuples is empty. If the set of tuples is empty, then the aggregation has no value.
- If the aggregation id is `strictsum`, then the resulting value is the same as if the aggregation id were `sum`, unless the set of tuples is empty. If the set of tuples is empty, then the aggregation has no value.
- If the aggregation id is `sum`, then the resulting value is the same as the sum of the values of the aggregation variable across the tuples in the set, weighted by the number of times each value occurs in the tuples in the set. If there are no tuples in the set, then the resulting value of the aggregation is the integer `0`.
- If the aggregation id is `concat`, then there is one value for each value of the second aggregation variable, given by the concatenation of the value of the first aggregation variable of each tuple with the value of the second aggregation variable used as a separator, ordered by the sort variables. If there are multiple aggregation tuples with the same sort variables then the first distinguished value is used to break ties. If there are no tuples in the set, then the single value of the aggregation is the empty string.
- If the aggregation id is `strictconcat`, then the result is the same as for `concat` except in the case where there are no aggregation tuples in which case the aggregation has no value.

- If the aggregation id is `unique`, then the result is the value of the aggregation variable if there is precisely one such value. Otherwise, the aggregation has no value.

15.13.12 Any

The `any` expression is a special kind of quantified expression.

```
any ::= "any" "(" var_decls ("|" (formula)? ("|" expr)?)? ")"
```

The values of an `any` expression are those values of the expression for which the formula matches.

The abbreviated cases for an `any` expression are interpreted in the same way as for an aggregation.

15.13.13 Ranges

Range expressions denote a range of values.

```
range ::= "[" expr ".." expr "]"
```

Both expressions must be subtypes of `int`, `float`, or `date`. If either of them are type `date`, then both of them must be.

If both expressions are subtypes of `int` then the type of the range is `int`. If both expressions are subtypes of `date` then the type of the range is `date`. Otherwise the type of the range is `float`.

The values of a range expression are those values which are ordered inclusively between a value of the first expression and a value of the second expression.

15.13.14 Set literals

Set literals denote a choice from a collection of values.

```
setliteral ::= "[" expr ("," expr)* "]"
```

Set literals can be of any type, but the types within a set literal have to be consistent according to the following criterion: At least one of the set elements has to be of a type that is a supertype of all the set element types. This supertype is the type of the set literal. For example, `float` is a supertype of `float` and `int`, therefore `x = [4, 5.6]` is valid. On the other hand, `y = [5, "test"]` does not adhere to the criterion.

The values of a set literal expression are all the values of all the contained element expressions.

Set literals are supported from release 2.1.0 of the CodeQL CLI, and release 1.24 of LGTM Enterprise.

15.14 Disambiguation of expressions

The grammar given in this section is disambiguated first by precedence, and second by associating left to right. The order of precedence from highest to lowest is:

- casts
- unary + and –
- binary ∗ , / and %
- binary + and –

Additionally, whenever a sequence of tokens can be interpreted either as a call to a predicate with result (with specified closure), or as a binary operation with operator + or ∗, the syntax is interpreted as a call to a predicate with result.

15.15 Formulas

A formula is a form of syntax used to *match* a named tuple given a store.

There are several kinds of formulas:

```
formula ::= fparen
        |   disjunction
        |   conjunction
        |   implies
        |   ifthen
        |   negated
        |   quantified
        |   comparison
        |   instanceof
        |   inrange
        |   call
```

This section specifies the syntax for each kind of formula and what tuples they match.

15.15.1 Parenthesized formulas

A parenthesized formula is a formula enclosed by a pair of parentheses:

```
fparen ::= "(" formula ")"
```

A parenthesized formula matches the same tuples as the nested formula matches.

15.15.2 Disjunctions

A disjunction is two formulas separated by the `or` keyword:

```
disjunction ::= formula "or" formula
```

A disjunction matches any tuple that matches either of the nested formulas.

15.15.3 Conjunctions

A conjunction is two formulas separated by the `and` keyword:

```
conjunction ::= formula "and" formula
```

A conjunction matches any tuple that also matches both of the two nested formulas.

15.15.4 Implications

An implication formula is two formulas separated by the `implies` keyword:

```
implies ::= formula "implies" formula
```

Neither of the two formulas may be another implication.

An implied formula matches if either the second formula matches, or the first formula does not match.

15.15.5 Conditional formulas

A conditional formula has the following syntax:

```
ifthen ::= "if" formula "then" formula "else" formula
```

The first formula is called the *condition* of the conditional formula. The second formula is called the *true branch*, and the second formula is called the *false branch*.

The conditional formula matches if the condition and the true branch both match. It also matches if the false branch matches and the condition does not match.

15.15.6 Negations

A negation formula is a formula preceded by the `not` keyword:

```
negated ::= "not" formula
```

A negation formula matches any tuple that does not match the nested formula.

15.15.7 Quantified formulas

A quantified formula has several syntaxes:

```
quantified ::= "exists" "(" expr ")"
           |   "exists" "(" var_decls ("|" formula)? ("|" formula)? ")"
           |   "forall" "(" var_decls ("|" formula)? "|" formula ")"
           |   "forex"  "(" var_decls ("|" formula)? "|" formula ")"
```

In all cases, the typing environment for the nested expressions or formulas is the same as the typing environment for the quantified formula, except that it also maps the variables in the variable declaration to their associated types.

The first form matches if the given expression has at least one value.

For the other forms, the extensions of the current named tuple for the given variable declarations are called the *quantifier extensions*. The nested formulas are called the *first quantified formula* and, if present, the *second quantified formula*.

The second `exists` formula matches if one of the quantifier extensions is such that the quantified formula or formulas all match.

A `forall` formula that has one quantified formula matches if that quantified formula matches all of the quantifier extensions. A `forall` with two quantified formulas matches if the second formula matches all extensions where the first formula matches.

A `forex` formula with one quantified formula matches under the same conditions as a `forall` formula matching, except that there must be at least one quantifier extension where that first quantified formula matches.

15.15.8 Comparisons

A comparison formula is two expressions separated by a comparison operator:

```
comparison ::= expr compop expr
compop ::= "=" | "!=" | "<" | ">" | "<=" | ">="
```

A comparison formula matches if there is one value of the left expression that is

in the given ordering with one of the values of the right expression. The ordering used is specified in *Ordering*. If one of the values is an integer and the other is a float value, then the integer is converted to a float value before the comparison.

If the operator is `=`, then at least one of the left and right expressions must have a type; if they both have a type, those types must be compatible.

If the operator is `!=`, then both expressions must have a type, and those types must be compatible.

If the operator is any other operator, then both expressions must have a type. Those types must be compatible with each other. Each of those types must be orderable.

15.15.9 Type checks

A type check formula has the following syntax:

```
instanceof ::= expr "instanceof" type
```

The type to the right of `instanceof` is called the *type-check type*.

The type of the expression must be compatible with the type-check type.

The formula matches if one of the values of the expression is in the type-check type.

15.15.10 Range checks

A range check has the following syntax:

```
inrange ::= expr "in" range
```

The formula is equivalent to `expr "=" range`.

15.15.11 Calls

A call has the following syntax:

```
call ::= predicateRef (closure)? "(" (exprs)? ")"
     |   primary "." predicateName (closure)? "(" (exprs)? ")"
```

The identifier is called the *predicate name* of the call.

A call must resolve to a predicate, using the same definition of resolve as for calls with results (see *Calls with results*).

The resolved predicate must not have a result type.

If the resolved predicate is a built-in member predicate of a primitive type, then the call may not include a closure. If the call does have a closure, then the call must resolve to a predicate with relational arity of 2.

The *candidate tuples* of a call are the ordered tuples formed by selecting a value from each of the arguments of the call.

If the call has no closure, then it matches whenever one of the candidate tuples satisfies the resolved predicate of the call, unless the call has a super expression as a receiver, in which case the candidate tuple must *directly* satisfy the resolved predicate. If the call has `*` or `+` closure, then the call matches whenever one of the candidate tuples satisfies or directly satisfies the associated closure of the resolved predicate.

15.15.12 Disambiguation of formulas

The grammar given in this section is disambiguated first by precedence, and second by associating left to right, except for implication which is non-associative. The order of precedence from highest to lowest is:

- Negation
- Conditional formulas
- Conjunction
- Disjunction
- Implication

15.16 Aliases

Aliases define new names for existing QL entities.

```
alias ::= annotations "predicate" literalId "=" predicateRef "/" int ";
 ↪"
      |   annotations "class" classname "=" type ";"
      |   annotations "module" modulename "=" moduleId ";"
```

An alias introduces a binding from the new name to the entity referred to by the right-hand side in the current modules declared predicate, type, or module environment respectively.

15.17 Built-ins

A QL database includes a number of *built-in predicates* . This section defines a number of built-in predicates that all databases include. Each database also includes a number of additional non-member predicates that are not specified in this document.

This section gives several tables of built-in predicates. For each predicate, the table gives the result type of each predicate that has one, and the sequence of argument types.

Each table also specifies which ordered tuples are in the database content of each predicate. It specifies this with a description that holds true for exactly the tuples that are included. In each description, the result is the last element of each tuple, if the predicate has a result type. The receiver is the first element of each tuple. The arguments are all elements of each tuple other than the result and the receiver.

15.17.1 Non-member built-ins

The following built-in predicates are non-member predicates:

Name	Result type	Argument types	Content
`any`			The empty tuple.
`none`			No tuples.
`toUrl`		string, int, int, int, int, string	Let the arguments be `file`, `startLine`, `startCol`, `endLine`, `endCol`, and `url`. The predicate holds if `url` is equal to the string `file://file:startLine:startCol:endLine:endCol`.

15.17.2 Built-ins for boolean

The following built-in predicates are members of type `boolean`:

Name	Result type	Argument types	Content
`booleanAnd`	boolean	boolean	The result is the boolean and of the receiver and the argument.
`booleanNot`	boolean		The result is the boolean not of the receiver.
`booleanOr`	boolean	boolean	The result is the boolean or of the receiver and the argument.
`booleanXor`	boolean	boolean	The result is the boolean exclusive or of the receiver and the argument.
`toString`	string		The result is true if the receiver is `true`, otherwise false.

15.17.3 Built-ins for date

The following built-in predicates are members of type `date`:

Name	Result type	Argument types	Content
daysTo	int	date	The result is the number of days between but not including the receiver and the argument.
getDay	int		The result is the day component of the receiver.
getHours	int		The result is the hours component of the receiver.
getMinutes	int		The result is the minutes component of the receiver.
getMonth	string		The result is a string that is determined by the month component of the receiver. The string is one of `January`, `February`, `March`, `April`, `May`, `June`, `July`, `August`, `September`, `October`, `November`, or `December`.
getSeconds	int		The result is the seconds component of the receiver.
getYear	int		The result is the year component of the receiver.
toISO	string		The result is a string representation of the date. The representation is left unspecified.
toString	string		The result is a string representation of the date. The representation is left unspecified.

15.17.4 Built-ins for float

The following built-in predicates are members of type `float`:

Name	Result type	Argument types	Content
abs	float		The result is the absolute value of the receiver.
acos	float		The result is the inverse cosine of the receiver.
asin	float		The result is the inverse sine of the receiver.
atan	float		The result is the inverse tangent of the receiver.
ceil	int		The result is the smallest integer greater than o
copySign	float	float	The result is the floating point number with the
cos	float		The result is the cosine of the receiver.

Table 1 – continued from previous p

Name	Result type	Argument types	Content
cosh	float		The result is the hyperbolic cosine of the receiv
exp	float		The result is the value of e, the base of the natu
floor	int		The result is the largest integer that is not great
log	float		The result is the natural logarithm of the receiv
log	float	float	The result is the logarithm of the receiver with
log	float	int	The result is the logarithm of the receiver with
log10	float		The result is the base-10 logarithm of the receiv
log2	float		The result is the base-2 logarithm of the receive
maximum	float	float	The result is the larger of the receiver and the a
maximum	float	int	The result is the larger of the receiver and the a
minimum	float	float	The result is the smaller of the receiver and the
minimum	float	int	The result is the smaller of the receiver and the
nextAfter	float	float	The result is the number adjacent to the receive
nextDown	float		The result is the number adjacent to the receive
nextUp	float		The result is the number adjacent to the receive
pow	float	float	The result is the receiver raised to the power of
pow	float	int	The result is the receiver raised to the power of
signum	float		The result is the sign of the receiver: zero if it is
sin	float		The result is the sine of the receiver.
sinh	float		The result is the hyperbolic sine of the receiver.
sqrt	float		The result is the square root of the receiver.
tan	float		The result is the tangent of the receiver.
tanh	float		The result is the hyperbolic tangent of the recei
toString	string		The decimal representation of the number as a
ulp	float		The result is the ULP (unit in last place) of the

15.17.5 Built-ins for int

The following built-in predicates are members of type `int`:

Name	Result type	Argument types	Content
abs	int		The result is the absolute value of t
acos	float		The result is the inverse cosine of th
asin	float		The result is the inverse sine of the
atan	float		The result is the inverse tangent of
cos	float		The result is the cosine of the recei
cosh	float		The result is the hyperbolic cosine o
exp	float		The result is the value of value of e
gcd	int	int	The result is the greatest common d
log	float		The result is the natural logarithm
log	float	float	The result is the logarithm of the re
log	float	int	The result is the logarithm of the re
log10	float		The result is the base-10 logarithm
log2	float		The result is the base-2 logarithm o
maximum	float	float	The result is the larger of the receiv
maximum	int	int	The result is the larger of the receiv
minimum	float	float	The result is the smaller of the rece
minimum	int	int	The result is the smaller of the rece
pow	float	float	The result is the receiver raised to t
pow	float	int	The result is the receiver raised to t
sin	float		The result is the sine of the receiver
sinh	float		The result is the hyperbolic sine of
sqrt	float		The result is the square root of the
tan	float		The result is the tangent of the rece
tanh	float		The result is the hyperbolic tangent
bitAnd	int	int	The result is the bitwise and of the
bitOr	int	int	The result is the bitwise or of the re
bitXor	int	int	The result is the bitwise xor of the r
bitNot	int		The result is the bitwise complemen
bitShiftLeft	int	int	The result is the bitwise left shift of
bitShiftRight	int	int	The result is the bitwise right shift o
bitShiftRightSigned	int	int	The result is the signed bitwise righ
toString	string		The result is the decimal representa

The leftmost bit after `bitShiftRightSigned` depends on sign extension, whereas after `bitShiftRight` it is zero.

15.17.6 Built-ins for string

The following built-in predicates are members of type `string`:

Name	Result type	Argument types	Content
charAt	string	int	The result is a 1-character string containing the character in the receiver at the index given by the argument. The first element of the string is at index 0.
indexOf	int	string	The result is an index into the receiver where the argument occurs.
indexOf	int	string, int, int	Let the arguments be `s`, `n`, and `start`. The result is the index of occurrence `n` of `substring` `s` in the receiver that is no earlier in the string than `start`.
isLowercase			The receiver contains no upper-case letters.
isUppercase			The receiver contains no lower-case letters.
length	int		The result is the number of characters in the receiver.
matches		string	The argument is a pattern that matches the receiver, in the same way as the LIKE operator in SQL. Patterns may include _ to match a single character and % to match any sequence of characters. A backslash can be used to escape an underscore, a percent, or a backslash. Otherwise, all characters in the pattern other than _ and % and \\ must match exactly.
prefix	string	int	The result is the prefix of the receiver that has a length exactly equal to the argument. If the argument is negative or greater than the receivers length, then there is no result.
regexpCapture	string	string, int	The receiver exactly matches the regex in the first argument, and the result is the group of the match numbered by the second argument.
regexpFind	string	string, int, int	The receiver contains one or more occurrences of the regex in the first argument. The result is the `substring` which matches the regex, the second argument is the occurrence number, and the third argument is the index within the receiver at which the occurrence begins.
regexpMatch		string	The receiver matches the argument as a regex.
regexpReplaceAll	string	string, string	The result is obtained by replacing all occurrences in the receiver of the first argument as a regex by the second argument.
replaceAll	string	string, string	The result is obtained by replacing all occurrences in the receiver of the first argument by the second.

Regular expressions are as defined by `java.util.regex.Pattern` in Java. For more information, see the Java API Documentation.

15.18 Evaluation

This section specifies the evaluation of a QL program. Evaluation happens in three phases. First, the program is stratified into a number of layers. Second, the layers are evaluated one by one. Finally, the queries in the QL file are evaluated to produce sets of ordered tuples.

15.18.1 Stratification

A QL program can be *stratified* to a sequence of *layers*. A layer is a set of predicates and types.

A valid stratification must include each predicate and type in the QL program. It must not include any other predicates or types.

A valid stratification must not include the same predicate in multiple layers.

Formulas, variable declarations and expressions within a predicate body have a *negation polarity* that is positive, negative, or zero. Positive and negative are opposites of each other, while zero is the opposite of itself. The negation polarity of a formula or expression is then determined as follows:

- The body of a predicate is positive.
- The formula within a negation formula has the opposite polarity to that of the negation formula.
- The condition of a conditional formula has zero polarity.
- The formula on the left of an implication formula has the opposite polarity to that of the implication.
- The formula and variable declarations of an aggregate have zero polarity.
- If the `monotonicAggregates` language pragma is not enabled, or the original formula and variable declarations are both omitted, then the expressions and

order by expressions of the aggregate have zero polarity.

- If the `monotonicAggregates` language pragma is enabled, and the original formula and variable declarations were not both omitted, then the expressions and order by expressions of the aggregate have the polarity of the aggregate.
- If a `forall` has two quantified formulas, then the first quantified formula has the opposite polarity to that of the `forall`.
- The variable declarations of a `forall` have the opposite polarity to that of the `forall`.
- If a `forex` has two quantified formulas, then the first quantified formula has zero polarity.
- The variable declarations of a `forex` have zero polarity.
- In all other cases, a formula or expression has the same polarity as its immediately enclosing formula or expression.

For a member predicate `p` we define the *strict dispatch dependencies*. The strict dispatch dependencies are defined as:

- The strict dispatch dependencies of any predicates that override `p`.
- If `p` is not abstract, `C.class` for any class `C` with a predicate that overrides `p`.

For a member predicate `p` we define the *dispatch dependencies*. The dispatch dependencies are defined as:

- The dispatch dependencies of predicates that override `p`.
- The predicate `p` itself.
- `C.class` where `C` is the class that defines `p`.

Predicates, and types can *depend* and *strictly depend* on each other. Such dependencies exist in the following circumstances:

- If `A` strictly depends on `B`, then `A` depends on `B`.
- If `A` depends on `B`, then `A` also depends on anything on which `B` depends.

- If `A` strictly depends on `B`, then `A` and anything depending on `A` strictly depend on anything on which `B` depends (including `B` itself).
- If a predicate has a parameter whose declared type is a class type `C`, it depends on `C.class`.
- If a predicate declares a result type which is a class type `C`, it depends on `C.class`.
- A member predicate of class `C` depends on `C.class`.
- If a predicate contains a variable declaration of a variable whose declared type is a class type `C`, then the predicate depends on `C.class`. If the declaration has negative or zero polarity then the dependency is strict.
- If a predicate contains a variable declaration with negative or zero polarity of a variable whose declared type is a class type `C`, then the predicate strictly depends on `C.class`.
- If a predicate contains an expression whose type is a class type `C`, then the predicate depends on `C.class`. If the expression has negative or zero polarity then the dependency is strict.
- A predicate containing a predicate call depends on the predicate to which the call resolves. If the call has negative or zero polarity then the dependency is strict.
- A predicate containing a predicate call, which resolves to a member predicate and does not have a `super` expression as a qualifier, depends on the dispatch dependencies of the root definitions of the target of the call. If the call has negative or zero polarity then the dependencies are strict. The predicate strictly depends on the strict dispatch dependencies of the root definitions.
- For each class `C` in the program, for each base class `B` of `C`, `C.extends` depends on `B.B`.
- For each class `C` in the program, for each base type `B` of `C` that is not a class type, `C.extends` depends on `B`.
- For each class `C` in the program, `C.class` depends on `C.C`.

- For each class `C` in the program, `C.C` depends on `C.extends`.
- For each class `C` in the program that declares a field of class type `B`, `C.C` depends on `B.class`.
- For each class `C` with a characteristic predicate, `C.C` depends on the characteristic predicate.
- For each abstract class `A` in the program, for each type `C` that has `A` as a base type, `A.class` depends on `C.class`.
- A predicate with a higher-order body may strictly depend or depend on each predicate reference within the body. The exact dependencies are left unspecified.

A valid stratification must have no predicate that depends on a predicate in a later layer. Additionally, it must have no predicate that strictly depends on a predicate in the same layer.

If a QL program has no valid stratification, then the program itself is not valid. If it does have a stratification, a QL implementation must choose exactly one stratification. The precise stratification chosen is left unspecified.

15.18.2 Layer evaluation

The store is first initialized with the *database content* of all built-in predicates and external predicates. The database content of a predicate is a set of ordered tuples that are included in the database.

Each layer of the stratification is *populated* in order. To populate a layer, each predicate in the layer is repeatedly populated until the store stops changing. The way that a predicate is populated is as follows:

- To populate a predicate that has a formula as a body, find all named tuples with the variables of the predicates arguments that match the body formula and the types of the variables. If the predicate has a result, then the matching named tuples should additionally have a value for `result` that is in the result type of the predicate. If the predicate is a member predicate, then the tuples should additionally have a value for `this` that is of the type assigned

to `this` by the typing environment. For each such tuple, convert the named tuple to an ordered tuple by sequencing the values of the tuple, starting with `this` if present, followed by the predicates arguments, followed by `result` if present. Add each such converted tuple to the predicate in the store.

- To populate an abstract predicate, do nothing.
- The population of predicates with a higher-order body is left only partially specified. A number of tuples are added to the given predicate in the store. The tuples that are added must be fully determined by the QL program and by the state of the store.
- To populate the type `C.extends` for a class `C`, identify each value `v` that has the following properties: It is in all non-class base types of `C`, and for each class base type `B` of `C` it is in `B.B`. For each such `v`, add `(v)` to `C.extends`.
- To populate the type `C.C` for a class `C`, if `C` has a characteristic predicate, then add all tuples from that predicate to the store. Otherwise add each tuple in `C.extends` into the store.
- To populate the type `C.class` for a non-abstract class type `C`, add each tuple in `C.C` to `C.class`.
- To populate the type `C.class` for an abstract class type `C`, for each class `D` that has `C` as a base type add all tuples in `D.class` to `C.class`.
- To populate a select clause, find all named tuples with the variables declared in the `from` clause that match the formula given in the `where` clause, if there is one. For each named tuple, convert it to a set of ordered tuples where each element of the ordered tuple is, in the context of the named tuple, a value of one of the corresponding select expressions. Collect all ordered tuples that can be produced from all of the restricted named tuples in this way. Add each such converted tuple to the select clause in the store.

15.18.3 Query evaluation

A query is evaluated as follows:

1. Identify all named tuples in the predicate targeted by the query.

2. Sequence the ordered tuples lexicographically. The first elements of the lexicographic order are the tuple elements specified by the ordering directives of the predicate targeted by the query, if it has any. Each such element is ordered either ascending (`asc`) or descending (`desc`) as specified by the ordering directive, or ascending if the ordering directive does not specify. This lexicographic order is only a partial order, if there are fewer ordering directives than elements of the tuples. An implementation may produce any sequence of the ordered tuples that satisfies this partial order.

15.19 Summary of syntax

The complete grammar for QL is as follows:

```
ql ::= moduleBody

module ::= annotation* "module" modulename "{" moduleBody "}"

moduleBody ::= (import | predicate | class | module | alias | select)*

import ::= annotations "import" importModuleId ("as" modulename)?

qualId ::= simpleId | qualId "." simpleId

importModuleId ::= qualId
               | importModuleId "::" simpleId

select ::= ("from" var_decls)? ("where" formula)? "select" as_exprs (
↪"order" "by" orderbys)?

as_exprs ::= as_expr ("," as_expr)*

as_expr ::= expr ("as" simpleId)?

orderbys ::= orderby ("," orderby)*

orderby ::= simpleId ("asc" | "desc")?
```

(continues on next page)

(continued from previous page)

```
predicate ::= annotations head optbody

annotations ::= annotation*

annotation ::= simpleAnnotation | argsAnnotation

simpleAnnotation ::= "abstract"
                   |   "cached"
                   |   "external"
                   |   "final"
                   |   "transient"
                   |   "library"
                   |   "private"
                   |   "deprecated"
                   |   "override"
                   |   "query"

argsAnnotation ::= "pragma" "[" ("noinline" | "nomagic" | "noopt") "]"
                 |   "language" "[" "monotonicAggregates" "]"
                 |   "bindingset" "[" (variable ( "," variable)*)? "]"

head ::= ("predicate" | type) predicateName "(" (var_decls)? ")"

optbody ::= ";"
          |  "{" formula "}"
          |  "=" literalId "(" (predicateRef "/" int ("," predicateRef "/
 ↪" int)*)? ")" "(" (exprs)? ")"

class ::= annotations "class" classname "extends" type ("," type)* "{"␣
 ↪member* "}"

member ::= character | predicate | field

character ::= annotations classname "(" ")" "{" formula "}"
```

(continues on next page)

(continued from previous page)

```
field ::= annotations var_decl ";"

moduleId ::= simpleId | moduleId "::" simpleId

type ::= (moduleId "::")? classname | dbasetype | "boolean" | "date" |
↪"float" | "int" | "string"

exprs ::= expr ("," expr)*

alias := annotations "predicate" literalId "=" predicateRef "/" int ";"
      |  annotations "class" classname "=" type ";"
      |  annotations "module" modulename "=" moduleId ";"

var_decls ::= var_decl ("," var_decl)*

var_decl ::= type simpleId

formula ::= fparen
        |   disjunction
        |   conjunction
        |   implies
        |   ifthen
        |   negated
        |   quantified
        |   comparison
        |   instanceof
        |   inrange
        |   call

fparen ::= "(" formula ")"

disjunction ::= formula "or" formula

conjunction ::= formula "and" formula

implies ::= formula "implies" formula
```

(continues on next page)

(continued from previous page)

```
ifthen ::= "if" formula "then" formula "else" formula

negated ::= "not" formula

quantified ::= "exists" "(" expr ")"
           |   "exists" "(" var_decls ("|" formula)? ("|" formula)? ")"
           |   "forall" "(" var_decls ("|" formula)? "|" formula ")"
           |   "forex"  "(" var_decls ("|" formula)? "|" formula ")"

comparison ::= expr compop expr

compop ::= "=" | "!=" | "<" | ">" | "<=" | ">="

instanceof ::= expr "instanceof" type

inrange ::= expr "in" range

call ::= predicateRef (closure)? "(" (exprs)? ")"
     |   primary "." predicateName (closure)? "(" (exprs)? ")"

closure ::= "*" | "+"

expr ::= dontcare
     |   unop
     |   binop
     |   cast
     |   primary


primary ::= eparen
        |   literal
        |   variable
        |   super_expr
        |   postfix_cast
        |   callwithresults
```

(continues on next page)

(continued from previous page)

```
        |   aggregation
        |   any
        |   range
        |   setliteral

eparen ::= "(" expr ")"

dontcare ::= "_"

literal ::= "false" | "true" | int | float | string

unop ::= "+" expr
     |   "-" expr

binop ::= expr "+" expr
      |   expr "-" expr
      |   expr "*" expr
      |   expr "/" expr
      |   expr "%" expr

variable ::= varname | "this" | "result"

super_expr ::= "super" | type "." "super"

cast ::= "(" type ")" expr

postfix_cast ::= primary "." "(" type ")"

aggregation ::= aggid ("[" expr "]")? "(" (var_decls)? ("|" (formula)?␣
↪("|" as_exprs ("order" "by" aggorderbys)?)?)? ")"
           |   aggid ("[" expr "]")? "(" as_exprs ("order" "by"␣
↪aggorderbys)? ")"
           |   "unique" "(" var_decls "|" (formula)? ("|" as_exprs)?
↪")"

aggid ::= "avg" | "concat" | "count" | "max" | "min" | "rank" |
↪"strictconcat" | "strictcount" | "strictsum" | "sum"
```

(continues on next page)

(continued from previous page)

```
aggorderbys ::= aggorderby ("," aggorderby)*

aggorderby ::= expr ("asc" | "desc")?

any ::= "any" "(" var_decls ("|" (formula)? ("|" expr)?)? ")"

callwithresults ::= predicateRef (closure)? "(" (exprs)? ")"
                |   primary "." predicateName (closure)? "(" (exprs)?
 ↪")"

range ::= "[" expr ".." expr "]"

setliteral ::= "[" expr ("," expr)* "]"

simpleId ::= lowerId | upperId

modulename ::= simpleId

classname ::= upperId

dbasetype ::= atLowerId

predicateRef ::= (moduleId "::")? literalId

predicateName ::= lowerId

varname ::= simpleId

literalId ::= lowerId | atLowerId | "any" | "none"
```

CHAPTER

SIXTEEN

QLDOC COMMENT SPECIFICATION

This document is a formal specification for QLDoc comments.

16.1 About QLDoc comments

You can provide documentation for a QL entity by adding a QLDoc comment in the source file. The QLDoc comment is displayed as pop-up information in QL editors, for example when you hover over a predicate name.

16.2 Notation

A QLDoc comment is a valid QL comment that begins with `/**` and ends with `*/`.

The content of a QLDoc comment is the textual body of the comment, omitting the initial `/**`, the trailing `*/`, and the leading whitespace followed by `*` on each internal line.

A QLDoc comment precedes the next QL syntax element after it in the file.

16.3 Association

A QLDoc comment may be associated with any of the following QL syntax elements:

- Class declarations

- Non-member predicate declarations
- Member predicate declarations
- Modules

For class and predicate declarations, the associated QLDoc comment (if any) is the closest preceding QLDoc comment.

For modules, the associated QLDoc comment (if any) is the QLDoc comment which is the first element in the file, and moreover is not associated with any other QL element.

16.4 Inheritance

If a member predicate has no directly associated QLDoc and overrides a set of member predicates which all have the same QLDoc, then the member predicate inherits that QLDoc.

16.5 Content

The content of a QLDoc comment is interpreted as standard Markdown, with the following extensions:

- Fenced code blocks using backticks.
- Automatic interpretation of links and email addresses.
- Use of appropriate characters for ellipses, dashes, apostrophes, and quotes.

The content of a QLDoc comment may contain metadata tags as follows:

The tag begins with any number of whitespace characters, followed by an @ sign. At this point there may be any number of non-whitespace characters, which form the key of the tag. Then, a single whitespace character which separates the key from the value. The value of the tag is formed by the remainder of the line, and any subsequent lines until another @ tag is seen, or the end of the content is reached.

INDEX