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.

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.

Notation

This section describes the notation used in the specification.

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 character's code point. As an example of both, the first character in the name QL is "Q" (U+0051).

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.

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.

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 `qlpack.yml` or `codeql-pack.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 `qlpack.yml` file defines a CodeQL pack. The content of a `qlpack.yml` file is described in the CodeQL CLI documentation. `codeql-pack.yml` is an alias for `qlpack.yml`.

The CodeQL CLI and tools based on it (such as, GitHub code scanning and the CodeQL extension for Visual Studio Code) construct a library path using CodeQL packs. For each CodeQL pack added to the library path, the CodeQL packs named in its `dependencies` 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 CodeQL packs. This process depends on a mechanism for finding CodeQL packs by pack name and version, as described in the CodeQL CLI documentation.

When the query directory contains neither a `qlpack.yml` nor `codeql-pack.yml` file, it is considered to be a CodeQL 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 don't use information from the database.

Name resolution

All modules have six 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.
- The *module signature environment*, whose keys are module signature names and whose values are module signatures.
- The *type signature environment*, whose keys are type signature names and whose values are type signatures.
- The *predicate signature environment*, whose keys are pairs of predicate signature names and arities and whose values are predicate signatures.

For each module, some namespaces are enforced to be disjoint:

- No keys may be shared between the **module namespace** and the **module signature namespace**.
- No keys may be shared between the **type namespace** and the **type signature namespace**.
- No keys may be shared between the **module namespace** and the **type signature namespace**.

- No keys may be shared between the **type namespace** and the **module signature namespace**.
- No keys may be shared between the **predicate namespace** and the **predicate signature namespace**.
- No keys may be shared between the **module signature namespace** and the **type signature namespace**.

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.

Global environments

The global module environment has a single entry `QlBuiltins`.

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 three global signature environments are empty.

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

Module environments

For each of modules, types, predicates, module signatures, type signatures, and predicates signatures, we distinguish four environments: *publically declared*, *privately declared*, *exported*, and *visible*. These are defined as follows (with X denoting the type of entity we are currently considering):

- The *privately declared X environment* of a module is the multimap of X declarations and aliases in the module itself that are annotated as `private`.
- The *publically declared X environment* of a module is the multimap of X declarations and aliases in the module itself that are not annotated as `private`.
- The *exported X environment* of a module is the union of
  1. its *publically declared X environment*, and
  2. for each module which the current module directly imports (excluding `private` imports - see "Import directives"): all entries from the *exported X environment* that have a key not present in the *publically declared X environment* of the current module, and
  3. if X is `predicates`, then for each module signature `S` that is implemented by the current module: an entry for each module signature default predicate in `S` that does not have the same name and arity as any of the entries in the **publically declared predicate environment** of the current module.
- The *visible X environment* of a module is the union of
  1. its *exported X environment*, and
  2. its *privately declared X environment*, and
  3. the *global X environment*, and
  4. for each module which the current module privately imports: all entries from the *exported X environment* that have a key not present in the *publically declared X environment* of the current module, and

5. if there is an enclosing module: all entries from the *visible X environment* of the enclosing module that have a key not present in the *publically declared X environment* of the current module, and
6. all parameters of the current module that are of type X.

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.

Modules

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 module's 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.

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.

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.

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 module's 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 file's 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 directive's 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 name's 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.

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.

Types

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

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`.

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 module's visible type environment.

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.

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.

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.

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.

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 two's 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.

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 two's complement integers.

For strings, the ordering is lexicographic.

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 lowerId
```

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.

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 a named tuple. A fact is written as the predicate name or type name followed immediately by the tuple. Here are some examples of facts:

```
successor(fst: 0, snd:1)
Tree.toString(this:@method_tree(12), result:"def println")
Location.class(this:@location(43))
Location.getURL(this: @location(43), result:"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 named tuple *directly satisfies* a predicate or type with a given tuple 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.
- There is a tuple with `this` component `v` that directly satisfies `t`.

An ordered tuple `v` *directly satisfies* a predicate with a given tuple if there is a fact in the store with the given predicate and a named tuple `v'` such that taking the ordered tuple formed by the `this` component of `v'` followed by the component for each argument equals the ordered tuple.

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 the named tuple satisfies the tuple.

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.

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.

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.

Whitespace

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

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 and does not start with an asterisk. Here is an example multiline comment:

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

QLDoc (qldoc)

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

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

The "content" of a QLDoc comment is the comment body of the comment, omitting the initial `/**`, the trailing `*/`, and the leading whitespace followed by `*` on each internal line.

For more information about how the content is interpreted, see "QLDoc" below.

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

Operators

The following sequences of characters are operator tokens:

```
<
<=
=
>
>=
_
-
,
;
!=
/
.
..
(
)
[
]
{
}
*
%
+
|
```

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       ::= lowerId
literalId     ::= lowerId | atLowerId
```

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
```

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
```

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...\""
```

Annotations

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

```
annotations ::= annotation*

annotation ::= simpleAnnotation | argsAnnotation

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

argsAnnotation ::= "pragma" "[" ("inline" | "inline_late" | "noinline" | "nomagic" | "noopt" |
"assume_small_delta") "]"
               |   "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.

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				

Annotation	Classes	Characters	Member predicates	Non-member predicates	Imports	Fields	Modules	Aliases
`final`	yes		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.

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				
`inline_late`				yes				
`noinline`		yes	yes	yes				
`nomagic`		yes	yes	yes				
`noopt`		yes	yes	yes				
`assume_small_delta`		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 predicate's arguments such that if those arguments are bound (restricted to a finite range of values), then all of the predicate's 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				

QLDoc

QLDoc is used for documenting QL entities and bindings. QLDoc that is used as part of the declaration is said to be declared.

Ambiguous QLDoc

If QLDoc can be parsed as part of a file module or as part of the first declaration in the file then it is parsed as part of the first declaration.

Inheriting QLDoc

If no QLDoc is provided then it may be inherited.

In the case of an alias then it may be inherited from the right-hand side of the alias.

In the case of a member predicate we collect all member predicates that it overrides with declared QLDoc. If there is a member predicate in that collection that overrides every other member predicate in that collection, then the QLDoc of that member predicate is used as the QLDoc.

In the case of a field we collect all fields that it overrides with declared QLDoc. If there is a field in that collection that overrides every other field in that collection, then the QLDoc of that field is used as the QLDoc.

Content

The content of a QLDoc comment is interpreted as CommonMark, with the following extensions:

- 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. Any sequence of consecutive whitespace characters in the value are replaced by a single space.

Metadata

If the query file starts with whitespace followed by a QLDoc comment, then the tags from that QLDoc comment form the query metadata.

Top-level entities

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

Non-member predicates

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

```
predicate ::= qldoc? annotations head optbody
```

A predicate definition adds a mapping from the predicate name and arity to the predicate declaration to the current module's 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.

Classes

A class definition has the following syntax:

```
class ::= qldoc? annotations "class" classname ("extends" type ("," type)*)? ("instanceof"
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.

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

A class type is said to *inherit* from the base types. 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 module's 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.

A valid class must have at least one base type or instanceof type.

Class dependencies

The program is invalid if there is a cycle of class dependencies.

The following are class dependencies: - `C` depends on `C.C` - `C.C` depends on `C.extends` - If `C` is abstract then it depends on all classes `D` such that `C` is a base type of `D`. - `C.extends` depends on `D.D` for each base type `D` of `C`. - `C.extends` depends on `D` for each instanceof type `D` of `C`.

Class environments

For each of member predicates and fields a class *inherits* and *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 types it inherits from, excluding any elements that are `overridden` by another element.
- 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 *visible X environment* is the overriding union of the declared X environment and the inherited X environment.

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

For each of member predicates and fields a domain type *exports* an environment. We say the *exported X extends environment* is the union of the exported `X` environments of types the class inherits from, excluding any elements that are `overridden` by another element. We say the *exported X instanceof environement* is the union of the exported `X` environments of types that a instanceof type inherits from, excluding any elements that are `overridden` by another element. The *exported X environment* of the domain type is the union of the exported `X` extends environment and the exported `X` instanceof environement.

Members

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

```
member ::= character | predicate | field
character ::= qldoc? annotations classname "(" ")" "{" formula "}"
field ::= qldoc? 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 class's declared member 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 predicate's *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 and also maps any fields on a class 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. The typing environment also maps any field to the type of the field.

Fields

A field declaration introduces a mapping from the field name to the field declaration in the class's declared field environment.

A field `f` with enclosing class `C` *overrides* a field `f'` with enclosing class `D` when `f` is annotated `override`, `C` inherits from `D`, `p'` is visible in `C`, and both `p` and `p'` have the same name.

A valid class may not inherit from two different classes that include a field with the same name, unless either one of the fields overrides the other, or the class defines a field that overrides both of them.

A valid field must override another field if it is annotated `override`.

When field `f` overrides field `g` the type of `f` must be a subtype of the type of `g`. `f` may not be a final field.

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" lowerId)?
```

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 ::= lowerId ("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.

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.

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 don't-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
        |   postfix_cast
        |   callwithresults
        |   aggregation
        |   expression_pragma
```

```
          |   any
          |   range
          |   setliteral
```

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.

Don't-care expressions

A don't-care expression is written as a single underscore:

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

All values are values of a don't-care expression.

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.

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.

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 two's-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 `%`.

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.

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 domain type of the type of `this` in the typing environment.

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

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 two's complement integers and 64-bit IEEE 754 floats. Other values will not be included in the values of the cast expression.

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.

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 *resolves* to a set of predicates. The ways a call can resolve are as follows:

- If the call has no receiver and the predicate name is a simple identifier, then the predicate is resolved by looking up its name and arity in the visible member-predicate environment of the enclosing class.
- If the call has no receiver and 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 module.
- If the call has no receiver and 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 type of the receiver is the same as the enclosing class, the predicate is resolved by looking up its name and arity in the visible predicate environment of the class.
- If the type of the receiver is not the same as the enclosing class, the predicate is resolved by looking up its name and arity in the exported predicate environment of the class or domain type.

If all the predicates that the call resolves to are declared on a primitive type, we then restrict to the set of predicates where each argument of the call is a subtype of the corresponding predicate argument type. Then we find all predicates `p` from this new set such that there is not another predicate `p'` where each argument of `p'` is a subtype of the corresponding argument in `p`. We then say the call resolves to this set instead.

A valid call must only resolve to a single predicate.

For each argument other than a don't-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 includes a closure, then all declared predicate arguments, the enclosing type of the declaration (if it exists), and the result type of the declaration (if it exists) must be compatible. If one of those types is a subtype of `int`, then all the other arguments must be a subtype of `int`.

A call to a member predicate may be a *direct* call:

- If the receiver is not a super expression it is not direct.
- If the receiver is `A.super` and `A` is an instanceof type and not a base type then it is not direct.
- If the receiver is `A.super` and `A` is a base type type and not an instanceof type then it is direct.
- If the receiver is `A.super` and `A` is a base type and an instanceof type then the call is not valid.
- If the receiver is `super` and the member predicate is in the exported member predicate environment of an instanceof type and not in the exported member predicate environment of a base type then it isn't direct.
- If the receiver is `super` and the member predicate is in the exported member predicate environment of a base type and not in the exported member predicate environment of an instanceof type then it is direct.
- If the receiver is `super` and the member predicate is in the exported member predicate environment of a base type and in the exported member predicate environment of an instanceof type then the call is not valid.

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 call is direct 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 call's matching tuples.

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)*

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`.

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 expression's name to the earlier expression's 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 expression's name to the expression's 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.

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.

Expression Pragma

Expression pragmas can be used to guide optimization.

::

expression_pragma ::= “pragma” “[” expression_pragma_type “]” “(” expr “)”

expression_pragma_type ::= “only_bind_out” | “only_bind_into”

The values of an expression pragma are the values of the contained expression.

The type *only_bind_out* hints that uses of the result of the expression pragma should not be used to guide the evaluation of the result of the contained expression. When checking to see that all values are bound the compiler does not assume that if the result of the expression pragma is bound then the result of the contained expression is bound.

The type *only_bind_into* hints that uses of the contained expression should not be used to guide the evaluation of the result of the expression pragma. When checking to see that all values are bound the compiler does not assume that if the result of the contained expression is bound then the result of the expression pragma is bound.

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.

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.

Since release 2.7.1 of the CodeQL CLI, a trailing comma is allowed in a set literal.

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 `-`

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.

Whenever a sequence of tokens can be interpreted either as arithmetic with a parenthesized variable or as a prefix cast of a unary operation, the syntax is interpreted as a cast.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

Range checks

A range check has the following syntax:

```
inrange ::= expr "in" (range | setliteral)
```

The formula is equivalent to `expr "=" range` or `expr "=" setliteral`.

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").

A call may be direct using the same definition of direct 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 is direct, 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.

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

Aliases

Aliases define new names for existing QL entities.

```
alias ::= qldoc? annotations "predicate" literalId "=" predicateRef "/" int ";"
      |   qldoc? annotations "class" classname "=" type ";"
      |   qldoc? 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 module's declared predicate, type, or module environment respectively.

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.

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`.

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."

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.

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 or equal to the receiver.
`copySign`	float	float	The result is the floating point number with the magnitude of the receiver and the sign of the argument.
`cos`	float		The result is the cosine of the receiver.
`cosh`	float		The result is the hyperbolic cosine of the receiver.
`exp`	float		The result is the value of e, the base of the natural logarithm, raised to the power of the receiver.
`floor`	int		The result is the largest integer that is not greater than the receiver.
`log`	float		The result is the natural logarithm of the receiver.
`log`	float	float	The result is the logarithm of the receiver with the base of the argument.
`log`	float	int	The result is the logarithm of the receiver with the base of the argument.
`log10`	float		The result is the base-10 logarithm of the receiver.

Name	Result type	Argument types	Content
`log2`	float		The result is the base-2 logarithm of the receiver.
`maximum`	float	float	The result is the larger of the receiver and the argument.
`maximum`	float	int	The result is the larger of the receiver and the argument.
`minimum`	float	float	The result is the smaller of the receiver and the argument.
`minimum`	float	int	The result is the smaller of the receiver and the argument.
`nextAfter`	float	float	The result is the number adjacent to the receiver in the direction of the argument.
`nextDown`	float		The result is the number adjacent to the receiver in the direction of negative infinity.
`nextUp`	float		The result is the number adjacent to the receiver in the direction of positive infinity.
`pow`	float	float	The result is the receiver raised to the power of the argument.
`pow`	float	int	The result is the receiver raised to the power of the argument.
`signum`	float		The result is the sign of the receiver: zero if it is zero, 1.0 if it is greater than zero, -1.0 if it is less than zero.
`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 receiver.
`toString`	string		The decimal representation of the number as a string.
`ulp`	float		The result is the ULP (unit in last place) of the receiver.

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 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.
`cos`	float		The result is the cosine of the receiver.
`cosh`	float		The result is the hyperbolic cosine of the receiver.
`exp`	float		The result is the value of value of e, the base of the natural logarithm, raised to the power of the receiver.
`gcd`	int	int	The result is the greatest common divisor of the receiver and the argument.

Name	Result type	Argument types	Content
`log`	float		The result is the natural logarithm of the receiver.
`log`	float	float	The result is the logarithm of the receiver with the base of the argument.
`log`	float	int	The result is the logarithm of the receiver with the base of the argument.
`log10`	float		The result is the base-10 logarithm of the receiver.
`log2`	float		The result is the base-2 logarithm of the receiver.
`maximum`	float	float	The result is the larger of the receiver and the argument.
`maximum`	int	int	The result is the larger of the receiver and the argument.
`minimum`	float	float	The result is the smaller of the receiver and the argument.
`minimum`	int	int	The result is the smaller of the receiver and the argument.
`pow`	float	float	The result is the receiver raised to the power of the argument.
`pow`	float	int	The result is the receiver raised to the power of the argument.
`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 receiver.
`bitAnd`	int	int	The result is the bitwise and of the receiver and the argument.
`bitOr`	int	int	The result is the bitwise or of the receiver and the argument.
`bitXor`	int	int	The result is the bitwise xor of the receiver and the argument.
`bitNot`	int		The result is the bitwise complement of the receiver.
`bitShiftLeft`	int	int	The result is the bitwise left shift of the receiver by the argument, modulo 32.
`bitShiftRight`	int	int	The result is the bitwise right shift of the receiver by the argument, modulo 32.
`bitShiftRightSigned`	int	int	The result is the signed bitwise right shift of the receiver by the argument, modulo 32.
`toString`	string		The result is the decimal representation of the number as a string.
`toUnicode`	string		The result is the unicode character for the receiver seen as a unicode code point.

The leftmost bit after `bitShiftRightSigned` depends on sign extension, whereas after `bitShiftRight` it is zero.

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 receiver's 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.
`splitAt`	string	string	The result is one of the strings obtained by splitting the receiver at every occurrence of the argument.
`splitAt`	string	string, int	Let the arguments be `delim` and `i`. The result is field number `i` of the fields obtained by splitting the receiver at every occurrence of `delim`.
`substring`	string	int, int	The result is the `substring` of the receiver starting at the index of the first argument and ending just before the index of the second argument.
`suffix`	string	int	The result is the suffix of the receiver that has a length exactly equal to the receiver's length minus the argument. If the argument

Name	Result type	Argument types	Content
			is negative or greater than the receiver's length, then there is no result. As a result, the identity `s.prefix(i)+s.suffix(i)=s` holds for `i` in `[0, s.length()]`.
`toDate`	date		The result is a date value determined by the receiver. The format of the receiver is unspecified, except that if `(d, s)` is in `date.toString`, `(s, d)` is in `string.toDate`.
`toFloat`	float		The result is the float whose value is represented by the receiver. If the receiver cannot be parsed as a float then there is no result.
`toInt`	int		The result is the integer whose value is represented by the receiver. If the receiver cannot be parsed as an integer or cannot be represented as a QL `int`, then there is no result. The parser accepts an optional leading `-` or `+` character, followed by one or more decimal digits.
`toLowerCase`	string		The result is the receiver with all letters converted to lower case.
`toString`	string		The result is the receiver.
`toUpperCase`	string		The result is the receiver with all letters converted to upper case.
`trim`	string		The result is the receiver with all whitespace removed from the beginning and end of the string.

Regular expressions are as defined by `java.util.regex.Pattern` in Java. For more information, see the Java API Documentation.

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.

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` which is not a variable reference, 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, where the call is not direct, 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 instanceof type `B` of `C`, `C.extends` depends on `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.

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 each named tuple `t` that has the following properties:
  - The tuple matches the body formula.
  - The variables should be the predicate's arguments.
  - If the predicate has a result, then the tuples should additionally have a value for `result`.
  - If the predicate is a member predicate or characteristic predicate of a class `C` then the tuples should additionally have a value for `this` and each visible field on the class.
  - The values corresponding to the arguments should all be a member of the declared types of the arguments.
  - The values corresponding to `result` should all be a member of the result type.
  - The values corresponding to the fields should all be a member of the declared types of the fields.
  - If the predicate is a member predicate of a class `C` and not a characteristic predicate, then the tuples should additionally extend some tuple in `C.class`.
  - If the predicate is a characteristic predicate of a class `C`, then there should be a tuple `t'` in `C.extends` such that for each visible field in `C`, any field that is equal to or overrides a field in `t'` should have the same value in `t`. `this` should also map to the same value in `t` and `t'`.

  For each such tuple remove any components that correspond to fields and add it 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 named tuple that has the following properties:
  - The value of `this` is in all non-class base types of `C`.
  - The value of `this` is in all instanceof types of `C`.
  - The keys of the tuple are `this` and the union of the public fields from each base type.
  - For each class base type `B` of `C` there is a named tuple with variables from the public fields of `B` and `this` that the given tuple and some tuple in `B.B` both extend.

  For each such tuple add it 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 all tuples `t` such that:
  - The variables of `t` should be `this` and the visible fields of `C`.
  - The values corresponding to the fields should all be a member of the declared types of the fields.
  - If the predicate is a characteristic predicate of a class `C`, then there should be a tuple `t'` in `C.extends` such that for each visible field in `C`, any field that is equal to or overrides a field in `t'` should have the same value in `t`. `this` should also map to the same value in `t` and `t'`.
- 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`, identify each named tuple that has the following properties:
  - It is a member of `C.C`.
  - For each class `D` that has `C` as a base type then there is a named tuple with variables from the public fields of `C` and `this` that the given tuple and a tuple in `D.class` both extend.

Query evaluation

A query is evaluated as follows:

1. Identify all facts about query predicates.
2. If there is a select clause then 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. Then 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.
3. The result is the facts from the query predicates plus the list of ordered tuples from the select clause if it exists.

Summary of syntax

The complete grammar for QL is as follows:

```
ql ::= qldoc? 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" lowerId)?

orderbys ::= orderby ("," orderby)*
```

```
orderby ::= lowerId ("asc" | "desc")?

predicate ::= qldoc? annotations head optbody

annotations ::= annotation*

annotation ::= simpleAnnotation | argsAnnotation

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

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

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

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

class ::= qldoc? annotations "class" classname ("extends" type ("," type)*)? ("instanceof"
type ("," type)*)?  "{" member* "}"

member ::= character | predicate | field

character ::= qldoc? annotations classname "(" ")" "{" formula "}"

field ::= qldoc? annotations var_decl ";"

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

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

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

alias ::= qldoc? annotations "predicate" literalId "=" predicateRef "/" int ";"
      |  qldoc? annotations "class" classname "=" type ";"
      |  qldoc? annotations "module" modulename "=" moduleId ";"

var_decls ::= (var_decl ("," var_decl)*)?

var_decl ::= type lowerId

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

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 | setliteral)

call ::= predicateRef (closure)? "(" (exprs)? ")"
     |   primary "." predicateName (closure)? "(" (exprs)? ")"

closure ::= "*" | "+"

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


primary ::= eparen
        |   literal
        |   variable
        |   super_expr
        |   postfix_cast
        |   callwithresults
        |   aggregation
        |   expression_pragma
        |   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)? ")"

  expression_pragma ::= "pragma" "[" expression_pragma_type "]" "(" expr ")"

  expression_pragma_type ::= "only_bind_out" | "only_bind_into"

  aggid ::= "avg" | "concat" | "count" | "max" | "min" | "rank" | "strictconcat" | "strictcount"
  | "strictsum" | "sum"

  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 ::= lowerId

  literalId ::= lowerId | atLowerId | "any" | "none"
```