CodeQL documentation

unknown

Nov 20, 2023

CONTENTS

CHAPTER

ONE

CODEQL OVERVIEW

Learn more about how CodeQL works, the languages and libraries supported by CodeQL analysis, and the tools you can use to run CodeQL on open source projects.

- *About CodeQL*: CodeQL is the analysis engine used by developers to automate security checks, and by security researchers to perform variant analysis.
- *Supported languages and frameworks*: View the languages, libraries, and frameworks supported in the latest version of CodeQL.
- *CodeQL tools*: GitHub provides the CodeQL command-line interface and CodeQL for Visual Studio Code for performing CodeQL analysis on open source codebases.
- *CodeQL glossary*: An overview of the technical terms and concepts in CodeQL.

1.1 About CodeQL

CodeQL is the analysis engine used by developers to automate security checks, and by security researchers to perform variant analysis.

In CodeQL, code is treated like data. Security vulnerabilities, bugs, and other errors are modeled as queries that can be executed against databases extracted from code. You can run the standard CodeQL queries, written by GitHub researchers and community contributors, or write your own to use in custom analyses. Queries that find potential bugs highlight the result directly in the source file.

1.1.1 About variant analysis

Variant analysis is the process of using a known security vulnerability as a seed to find similar problems in your code. It's a technique that security engineers use to identify potential vulnerabilities, and ensure these threats are properly fixed across multiple codebases.

Querying code using CodeQL is the most efficient way to perform variant analysis. You can use the standard CodeQL queries to identify seed vulnerabilities, or find new vulnerabilities by writing your own custom CodeQL queries. Then, develop or iterate over the query to automatically find logical variants of the same bug that could be missed using traditional manual techniques.

1.1.2 CodeQL analysis

CodeQL analysis consists of three steps:

1. Preparing the code, by creating a CodeQL database
2. Running CodeQL queries against the database
3. Interpreting the query results

Database creation

To create a database, CodeQL first extracts a single relational representation of each source file in the codebase.

For compiled languages, extraction works by monitoring the normal build process. Each time a compiler is invoked to process a source file, a copy of that file is made, and all relevant information about the source code is collected. This includes syntactic data about the abstract syntax tree and semantic data about name binding and type information.

For interpreted languages, the extractor runs directly on the source code, resolving dependencies to give an accurate representation of the codebase.

There is one *extractor* for each language supported by CodeQL to ensure that the extraction process is as accurate as possible. For multi-language codebases, databases are generated one language at a time.

After extraction, all the data required for analysis (relational data, copied source files, and a language-specific *database schema*, which specifies the mutual relations in the data) is imported into a single directory, known as a *CodeQL database*.

Query execution

After you've created a CodeQL database, one or more queries are executed against it. CodeQL queries are written in a specially-designed object-oriented query language called QL. You can run the queries checked out from the CodeQL repo (or custom queries that you've written yourself) using the *CodeQL for VS Code extension* or the *CodeQL CLI*. For more information about queries, see "*About CodeQL queries*."

Query results

The final step converts results produced during query execution into a form that is more meaningful in the context of the source code. That is, the results are interpreted in a way that highlights the potential issue that the queries are designed to find.

Queries contain metadata properties that indicate how the results should be interpreted. For instance, some queries display a simple message at a single location in the code. Others display a series of locations that represent steps along a data-flow or control-flow path, along with a message explaining the significance of the result. Queries that don't have metadata are not interpreted—their results are output as a table and not displayed in the source code.

Following interpretation, results are output for code review and triaging. In CodeQL for Visual Studio Code, interpreted query results are automatically displayed in the source code. Results generated by the CodeQL CLI can be output into a number of different formats for use with different tools.

1.1.3 About CodeQL databases

CodeQL databases contain queryable data extracted from a codebase, for a single language at a particular point in time. The database contains a full, hierarchical representation of the code, including a representation of the abstract syntax tree, the data flow graph, and the control flow graph.

Each language has its own unique database schema that defines the relations used to create a database. The schema provides an interface between the initial lexical analysis during the extraction process, and the actual complex analysis using CodeQL. The schema specifies, for instance, that there is a table for every language construct.

For each language, the CodeQL libraries define classes to provide a layer of abstraction over the database tables. This provides an object-oriented view of the data which makes it easier to write queries.

For example, in a CodeQL database for a Java program, two key tables are:

- The `expressions` table containing a row for every single expression in the source code that was analyzed during the build process.
- The `statements` table containing a row for every single statement in the source code that was analyzed during the build process.

The CodeQL library defines classes to provide a layer of abstraction over each of these tables (and the related auxiliary tables): `Expr` and `Stmt`.

1.2 Supported languages and frameworks

View the languages, libraries, and frameworks supported in the latest version of CodeQL.

1.2.1 Languages and compilers

CodeQL supports the following languages and compilers.

Language	Variants	Compilers	Extensions
C/C++	C89, C99, C11, C18, C++98, C++03, C++11, C++14, C++17, C++20[1]	Clang (and clang-cl[2]) extensions (up to Clang 12.0), GNU extensions (up to GCC 11.1), Microsoft extensions (up to VS 2019), Arm Compiler 5[3]	`.cpp`, `.c++`, `.cxx`, `.hpp`, `.hh`, `.h++`, `.hxx`, `.c`, `.cc`, `.h`
C#	C# up to 10.0	Microsoft Visual Studio up to 2019 with .NET up to 4.8, .NET Core up to 3.1 .NET 5, .NET 6	`.sln`, `.csproj`, `.cs`, `.cshtml`, `.xaml`
Go (aka Golan	Go up to 1.17	Go 1.11 or more recent	`.go`
Java	Java 7 to 16[4]	javac (OpenJDK and Oracle JDK), Eclipse compiler for Java (ECJ)[5]	`.java`
JavaS	ECMAScript 2021 or lower	Not applicable	`.js`, `.jsx`, `.mjs`, `.es`, `.es6`, `.htm`, `.html`, `.xhtm`, `.xhtml`, `.vue`, `.hbs`, `.ejs`, `.njk`, `.json`, `.yaml`, `.yml`, `.raml`, `.xml`[6]
Pytho	2.7, 3.5, 3.6, 3.7, 3.8, 3.9	Not applicable	`.py`
Ruby[7]	up to 3.0.2	Not applicable	`.rb`, `.erb`, `.gemspec`, `Gemfile`
Type-Script[8]	2.6-4.5	Standard TypeScript compiler	`.ts`, `.tsx`

1.2.2 Frameworks and libraries

The libraries and queries in the current version of CodeQL have been explicitly checked against the libraries and frameworks listed below.

> Tip
>
> If you're interested in other libraries or frameworks, you can extend the analysis to cover them. For example, by extending the data flow libraries to include data sources and sinks for additional libraries or frameworks.

[1] C++20 support is currently in beta. Supported for GCC on Linux only. Modules are *not* supported.
[2] Support for the clang-cl compiler is preliminary.
[3] Support for the Arm Compiler (armcc) is preliminary.
[4] Builds that execute on Java 7 to 16 can be analyzed. The analysis understands Java 16 standard language features.
[5] ECJ is supported when the build invokes it via the Maven Compiler plugin or the Takari Lifecycle plugin.
[6] JSX and Flow code, YAML, JSON, HTML, and XML files may also be analyzed with JavaScript files.
[7] Requires glibc 2.17.
[8] TypeScript analysis is performed by running the JavaScript extractor with TypeScript enabled. This is the default for LGTM.

C and C++ built-in support

Name	Category
Bloomberg Standard Library	Utility library
Berkeley socket API library	Network communicator
string.h	String library

C# built-in support

Name	Category
ASP.NET	Web application framework
ASP.NET Core	Web application framework
ASP.NET Razor templates	Web application framework
Dapper	Database ORM
EntityFramework	Database ORM
EntityFramework Core	Database ORM
Json.NET	Serialization
NHibernate	Database ORM
WinForms	User interface

Go built-in support

Name	Category
beego	Web/logging/database framework
Chi	Web framework
Couchbase (gocb and go-couchbase)	Database
Echo	Web framework
Gin	Web framework
glog	Logging library
go-pg	Database
go-restful	Web application framework
go-sh	Utility library
go-spew	Logging library
GoKit	Microservice toolkit
Gokogiri	XPath library
golang.org/x/crypto/ssh	Network communicator
golang.org/x/net/websocket	Network communicator
goproxy	HTTP proxy library
Gorilla mux	HTTP request router and dispatcher
Gorilla websocket	Network communicator
GORM	Database
GoWebsocket	Network communicator
goxpath	XPath library
htmlquery	XPath library
json-iterator	Serialization
jsonpatch	Serialization

continues on next page

Table 1 – continued from previous page

Name	Category
jsonquery	XPath library
klog	Logging library
Logrus	Logging library
Macaron	Web framework
mongo	Database
nhooyr.io/websocket	Network communicator
protobuf	Serialization
Revel	Web framework
sqlx	Database
SendGrid	Email library
Squirrel	Database
ws	Network communicator
xmlpath	XPath library
xmlquery	XPath library
xpath	XPath library
xpathparser	XPath library
yaml	Serialization
zap	Logging library

Java built-in support

Name	Category
Apache Commons Lang	Utility library
Apache Commons Collections	Data structure utility library
Apache HTTP components	Network communicator
Guava	Utility and collections library
Hibernate	Database
iBatis / MyBatis	Database
Jackson	Serialization
JSON-java	Serialization
Java Persistence API (JPA)	Database
JaxRS	Jakarta EE API specification
JDBC	Database
Protobuf	Serialization
Kryo deserialization	Serialization
SnakeYaml	Serialization
Spring JDBC	Database
Spring MVC	Web application framework
Struts	Web application framework
Thrift	RPC framework
XStream	Serialization

JavaScript and TypeScript built-in support

Name	Category
angular (modern version)	HTML framework
angular.js (legacy version)	HTML framework
axios	Network communicator
browser	Runtime environment
EJS	templating language
electron	Runtime environment
express	Server
handlebars	templating language
hapi	Server
hogan	templating language
jquery	Utility library
koa	Server
lodash	Utility library
mongodb	Database
mssql	Database
mustache	templating language
mysql	Database
node	Runtime environment
nest.js	Server
nunjucks	templating language
postgres	Database
ramda	Utility library
react	HTML framework
react native	HTML framework
request	Network communicator
sequelize	Database
socket.io	Network communicator
sqlite3	Database
superagent	Network communicator
swig	templating language
underscore	Utility library
vue	HTML framework

Python built-in support

Name	Category
aiohttp.web	Web framework
Django	Web framework
djangorestframework	Web framework
FastAPI	Web framework
Flask	Web framework
Tornado	Web framework
Twisted	Web framework
Flask-Admin	Web framework
starlette	Asynchronous Server Gateway Interface (ASGI)
python-ldap	Lightweight Directory Access Protocol (LDAP)

continues on next page

Table 3 – continued from previous page

Name	Category
ldap3	Lightweight Directory Access Protocol (LDAP)
requests	HTTP client
dill	Serialization
PyYAML	Serialization
ruamel.yaml	Serialization
simplejson	Serialization
toml	Serialization
ujson	Serialization
fabric	Utility library
idna	Utility library
invoke	Utility library
jmespath	Utility library
multidict	Utility library
pydantic	Utility library
yarl	Utility library
aioch	Database
aiomysql	Database
aiopg	Database
asyncpg	Database
clickhouse-driver	Database
mysql-connector-python	Database
mysql-connector	Database
MySQL-python	Database
mysqlclient	Database
psycopg2	Database
sqlite3	Database
Flask-SQLAlchemy	Database ORM
peewee	Database ORM
SQLAlchemy	Database ORM
cryptography	Cryptography library
pycryptodome	Cryptography library
pycryptodomex	Cryptography library
rsa	Cryptography library
MarkupSafe	Escaping Library

1.3 CodeQL tools

GitHub provides the CodeQL command-line interface and CodeQL for Visual Studio Code for performing CodeQL analysis on open source codebases.

1.3.1 CodeQL command-line interface

The CodeQL command-line interface (CLI) is primarily used to create databases for security research. You can also query CodeQL databases directly from the command line or using the Visual Studio Code extension. For more information, see "*CodeQL CLI*."

1.3.2 CodeQL for Visual Studio Code

You can analyze CodeQL databases in Visual Studio Code using the CodeQL extension, which provides an enhanced environment for writing and running custom queries and viewing the results. For more information, see "*CodeQL for Visual Studio Code*."

1.4 CodeQL glossary

An overview of the technical terms and concepts in CodeQL.

1.4.1 `.bqrs` file

A binary query result set (BQRS) file. BQRS is the binary representation of the raw result of a query, with the extension `.bqrs`. A BQRS file can be interpreted into meaningful results and related to your source code. For example, alert query results are interpreted to display a string at the location in the source code where the alert occurs, as specified in the query. Similarly, path query results are interpreted as pairs of locations (sources and sinks) between which information can flow. These results can be exported as a variety of different formats, including SARIF.

1.4.2 CodeQL database

A database (or CodeQL database) is a directory containing:

- queryable data, extracted from the code.
- a source reference, for displaying query results directly in the code.
- query results.
- log files generated during database creation, query execution, and other operations.

1.4.3 DIL

DIL stands for Datalog Intermediary Language. It is an intermediate representation between QL and relation algebra (RA) that is generated during query compilation. DIL is useful for advanced users as an aid for debugging query performance. The DIL format may change without warning between CLI releases.

When you specify the `--dump-dil` option for `codeql query compile`, CodeQL prints DIL to standard output for the queries it compiles. You can also view results in DIL format when you run queries in VS Code. For more information, see "*Analyzing your projects*" in the CodeQL for VS Code help.

1.4.4 Extractor

An extractor is a tool that produces the relational data and source reference for each input file, from which a CodeQL database can be built.

1.4.5 QL database schema

A QL database schema is a file describing the column types and extensional relations that make up a raw QL dataset. It is a text file with the `.dbscheme` extension.

The extractor and core QL pack for a language each declare the database schema that they use. This defines the database layout they create or expect. When you create a CodeQL database, the extractor copies its schema into the database. The CLI uses this to check whether the CodeQL database is compatible with a particular CodeQL library. If they aren't compatible you can use `database upgrade` to upgrade the schema for the CodeQL database.

There is currently no public-facing specification for the syntax of schemas.

1.4.6 `.qlo` files

`.qlo` files are optionally generated during query compilation. If you specify the `--dump-qlo` option for `codeql query compile`, CodeQL writes `.qlo` files for the queries it compiles. They can be used as an aid for debugging and performance tuning for advanced users.

`.qlo` is a binary format that represents a compiled and optimized query in terms of relational algebra (RA) or the intermediate *DIL* format. `.qlo` files can be expanded to readable text using `codeql query decompile`.

The exact details of the `.qlo` format may change without warning between CLI releases.

1.4.7 SARIF file

Static analysis results interchange format (SARIF) is an output format used for sharing static analysis results. For more information, see "*SARIF output*."

1.4.8 Source reference

A source reference is a mechanism that allows the retrieval of the contents of a source file, given an absolute filename at which that file resided during extraction. Specific examples include:

- A source archive directory, within which the requested absolute filename maps to a UTF8-encoded file.
- A source archive, typically in ZIP format, which contains the UTF8-encoded content of all source files.
- A source archive repository, typically in `git` format, typically bare, which contains the UTF8-encoded content of all source files.

Source references are typically included in CodeQL databases.

1.4.9 TRAP file

A TRAP file is a UTF-8 encoded file generated by a CodeQL extractor with the extension `.trap`. To save space, they are usually archived. They contain the information that, when interpreted relative to a QL database schema, is used to create a QL dataset.

CHAPTER

TWO

CODEQL FOR VISUAL STUDIO CODE

The CodeQL extension for Visual Studio Code adds rich language support for CodeQL and allows you to easily find problems in codebases.

- *About CodeQL for Visual Studio Code*: CodeQL for Visual Studio Code is an extension that lets you write, run, and test CodeQL queries in Visual Studio Code.
- *Setting up CodeQL in Visual Studio Code*: You can install and configure the CodeQL extension in Visual Studio Code.
- *Analyzing your projects*: You can run queries on CodeQL databases and view the results in Visual Studio Code.
- *Exploring the structure of your source code*: You can use the AST viewer to display the abstract syntax tree of a CodeQL database.
- *Exploring data flow with path queries*: You can run CodeQL queries in VS Code to help you track the flow of data through a program, highlighting areas that are potential security vulnerabilities.
- *Testing CodeQL queries in Visual Studio Code*: You can run unit tests for CodeQL queries using the Visual Studio Code extension.
- *Working with CodeQL packs in Visual Studio Code*: You can view and edit CodeQL packs in Visual Studio Code.
- *Customizing settings*: You can edit the settings for the CodeQL extension to suit your needs.
- *Troubleshooting CodeQL for Visual Studio Code*: You can use the detailed information written to the extension's log files if you need to troubleshoot problems.
- *About telemetry in CodeQL for Visual Studio Code*: If you specifically opt in to permit GitHub to do so, GitHub will collect usage data and metrics for the purposes of helping the core developers to improve the CodeQL extension for VS Code.

2.1 About CodeQL for Visual Studio Code

CodeQL for Visual Studio Code is an extension that lets you write, run, and test CodeQL queries in Visual Studio Code.

2.1.1 Features

CodeQL for Visual Studio Code provides an easy way to run queries from the large, open source repository of CodeQL security queries. With these queries, or your own custom queries, you can analyze databases generated from source code to find errors and security vulnerabilities. The Results view shows the flow of data through the results of path queries, which is essential for triaging security results.

The CodeQL extension also adds a **CodeQL** sidebar view to VS Code. This contains a list of databases, and an overview of the queries that you have run in the current session.

The extension provides standard IntelliSense features for query files (extension `.ql`) and library files (extension `.qll`) that you open in the Visual Studio Code editor.

- Syntax highlighting
- Right-click options (such as **Go To Definition**)
- Autocomplete suggestions
- Hover information

You can also use the VS Code **Format Document** command to format your code according to the CodeQL style guide.

2.1.2 Data and telemetry

If you specifically opt in to permit GitHub to do so, GitHub will collect usage data and metrics for the purposes of helping the core developers to improve the CodeQL extension for VS Code. For more information, see "*About telemetry in CodeQL for Visual Studio Code*."

2.1.3 Further reading

- "*Setting up CodeQL in Visual Studio Code*"
- "*Analyzing your projects*"

2.2 Setting up CodeQL in Visual Studio Code

You can install and configure the CodeQL extension in Visual Studio Code.

> License notice
>
> If you don't have an Enterprise license then, by installing this product, you are agreeing to the GitHub CodeQL Terms and Conditions.
>
> GitHub CodeQL is licensed on a per-user basis. Under the license restrictions, you can use CodeQL to perform the following tasks:
>
> - To perform academic research.
> - To demonstrate the software.
> - To test CodeQL queries that are released under an OSI-approved License to confirm that new versions of those queries continue to find the right vulnerabilities.
>
> where "OSI-approved License" means an Open Source Initiative (OSI)-approved open source software license.
>
> If you are working with an Open Source Codebase (that is, a codebase that is released under an OSI-approved License) you can also use CodeQL for the following tasks:

- To perform analysis of the Open Source Codebase.
- If the Open Source Codebase is hosted and maintained on GitHub.com, to generate CodeQL databases for or during automated analysis, continuous integration, or continuous delivery.

CodeQL can't be used for automated analysis, continuous integration or continuous delivery, whether as part of normal software engineering processes or otherwise, except in the express cases set forth herein. For these uses, contact the sales team.

2.2.1 Prerequisites

The CodeQL extension requires a minimum of Visual Studio Code 1.39. Older versions are not supported.

2.2.2 Installing the extension

You can install the CodeQL extension using any of the normal methods for installing a VS Code extension:

- Go to the Visual Studio Code Marketplace in your browser and click **Install**.
- In the Extensions view (**Ctrl+Shift+X** or **Cmd+Shift+X**), search for `CodeQL`, then select **Install**.
- Download the CodeQL VSIX file. Then, in the Extensions view, click **More actions** > **Install from VSIX**, and select the CodeQL VSIX file.

2.2.3 Configuring access to the CodeQL CLI

The extension uses the CodeQL CLI to compile and run queries.

If you already have the CLI installed and added to your `PATH`, the extension uses that version. This might be the case if you create your own CodeQL databases instead of downloading them from LGTM.com. For more information, see "*CodeQL CLI*."

Otherwise, the extension automatically manages access to the executable of the CLI for you. This ensures that the CLI is compatible with the CodeQL extension. You can also check for updates with the **CodeQL: Check for CLI Updates** command.

> Note
>
> The extension-managed CLI is not accessible from the terminal. If you intend to use the CLI outside of the extension (for example to create databases), we recommend that you install your own copy of the CLI. To avoid having two copies of the CLI on your machine, you can point the CodeQL CLI **Executable Path** setting to your existing copy of the CLI.

If you want the extension to use a specific version of the CLI, set the CodeQL CLI **Executable Path** to the location of the executable file for the CLI. That is, the file named `codeql` (Linux/Mac) or `codeql.exe` (Windows). For more information, see "*Customizing settings*."

If you have any difficulty setting up access to the CodeQL CLI, check the CodeQL Extension Log for error messages. For more information, see "*Troubleshooting CodeQL for Visual Studio Code*."

2.2.4 Setting up a CodeQL workspace

When you're working with CodeQL, you need access to the standard CodeQL libraries. This also makes a wide variety of queries available to explore.

There are two ways to do this:

- Recommended, use the "starter" workspace. This is maintained as a Git repository which makes it easy to keep up to date with changes to the libraries. For more information, see "*Using the starter workspace*" below.
- More advanced, add the CodeQL libraries and queries to an existing workspace. For more information, see "*Updating an existing workspace for CodeQL*" below.

  Note

  For CLI users there is a third option: If you have followed the instructions in "*Getting started with the CodeQL CLI*" to create a CodeQL directory (for example `codeql-home`) containing the CodeQL libraries, you can open this directory in VS Code. This also gives the extension access to the CodeQL libraries.

Click to show information for LGTM Enterprise users

Your local version of the CodeQL queries and libraries should match your version of LGTM Enterprise. For example, if you use LGTM Enterprise 1.27, then you should clone the `1.27.0` branch of the starter workspace (or the appropriate `1.27.x` branch, corresponding to each maintenance release).

This ensures that the queries and libraries you write in VS Code also work in the query console on LGTM Enterprise.

If you prefer to add the CodeQL queries and libraries to an *existing workspace* instead of the starter workspace, then you should clone the appropriate branch of the general CodeQL repository and the CodeQL repository for Go and add them to your workspace.

Using the starter workspace

The starter workspace is a Git repository. It contains:

- The repository of CodeQL libraries and queries for C/C++, C#, Java, JavaScript, Python, and Ruby. This is included as a submodule, so it can be updated without affecting your custom queries.
- The repository of CodeQL libraries and queries for Go. This is also included as a submodule.
- A series of folders named `codeql-custom-queries-<language>`. These are ready for you to start developing your own custom queries for each language, using the standard libraries. There are some example queries to get you started.

To use the starter workspace:

1. **Clone the https://github.com/github/vscode-codeql-starter/ repository to your computer:**
   - Make sure you include the submodules, either by using `git clone --recursive`, or using by `git submodule update --init --remote` after cloning.
   - Use `git submodule update --remote` regularly to keep the submodules up to date.
2. In VS Code, use the **File > Open Workspace** option to open the `vscode-codeql-starter.code-workspace` file from your checkout of the workspace repository.

Updating an existing workspace for CodeQL

You can add the CodeQL libraries to an existing workspace by making a local clone of the CodeQL repository directly: https://github.com/github/codeql.

To make the standard libraries available in your workspace:

1. Select **File** > **Add Folder to Workspace**, and choose your local checkout of the `github/codeql` repository.
2. Create one new folder per target language, using either the **New Folder** or **Add Folder to Workspace** options, to hold custom queries and libraries.
3. Create a `qlpack.yml` file in each target language folder. This tells the CodeQL CLI the target language for that folder and what its dependencies are. (The `main` branch of `github/codeql` already has these files.) CodeQL will look for the dependencies in all the open workspace folders, or on the user's search path.

For example, to make a custom CodeQL folder called `my-custom-cpp-pack` depend on the CodeQL standard library for C++, create a `qlpack.yml` file with the following contents:

```
name: my-custom-cpp-pack
version: 0.0.0
libraryPathDependencies: codeql/cpp-all
```

For more information about why you need to add a `qlpack.yml` file, see "*About QL packs*."

> Note
>
> The CodeQL libraries for Go are not included in the `github/codeql` repository, but are stored separately. To analyze Go projects, clone the repository at https://github.com/github/codeql-go and add it to your workspace as above.

2.2.5 Further reading

- "*Analyzing your projects*"
- "*CodeQL CLI*"

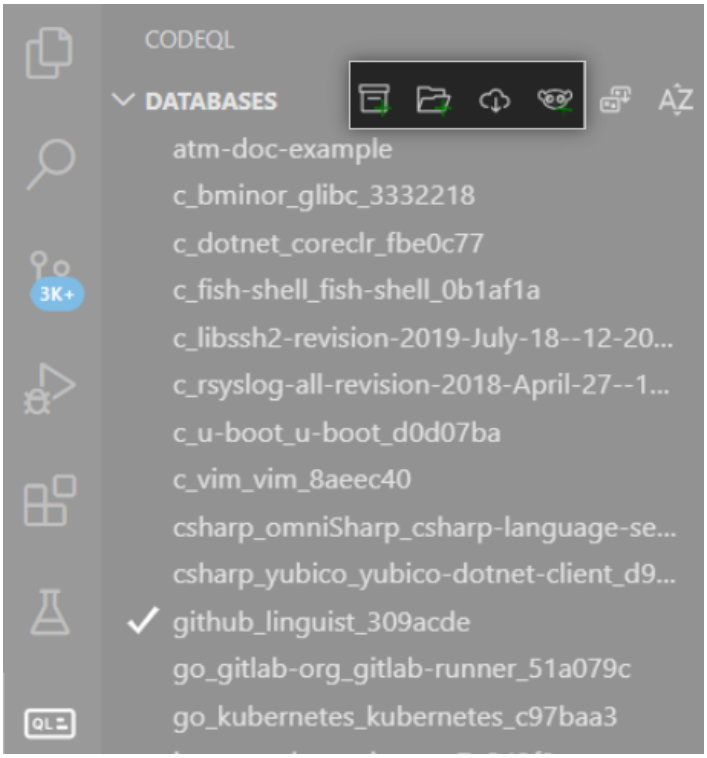
2.3 Analyzing your projects

You can run queries on CodeQL databases and view the results in Visual Studio Code.

2.3.1 Choosing a database

To analyze a project, you need to add a *CodeQL database* for that project.

1. Open the CodeQL Databases view in the sidebar.
2. Hover over the **Databases** title bar and click the appropriate icon to add your database. You can add a database from a local ZIP archive or folder, from a public URL, or from a project slug or URL on LGTM.com.



For more information about obtaining a local database, see below.

3. Once you've chosen a database, it is displayed in the Databases view. To see the menu options for interacting with a database, right-click an entry in the list. You can select multiple databases using **Ctrl/Cmd+click**.

Obtaining a local database

If you have a CodeQL database saved locally, as an unarchived folder or as a ZIP file, you can add it to Visual Studio Code. There are several ways to obtain a local CodeQL database.

- To create a database with the CodeQL CLI, see "*Creating CodeQL databases*."
- To download a database from LGTM.com:
  1. Log in to LGTM.com.
  2. Find a project you're interested in and display the Integrations tab (for example, Apache Kafka).
  3. Scroll to the **CodeQL databases for local analysis** section at the bottom of the page.
  4. Download databases for the languages that you want to explore.
- To analyze a test database, add a `.testproj` folder to the Databases view. Test databases (that is, folders with a `.testproj` extension) are generated when you run regression tests on custom queries using the *CodeQL CLI*. If a query fails a regression test, you may want to analyze the test database in Visual Studio Code to debug the failure.

  For more information about running query tests, see "*Testing custom queries*" in the CodeQL CLI help.

2.3.2 Running a query

The CodeQL repository on GitHub contains lots of example queries. If you have that folder (or a different QL pack) available in your workspace, you can access existing queries under `<language>/ql/src/<category>`, for example `java/ql/src/Likely Bugs`.

1. Open a query (`.ql`) file. It is displayed in the editor, with IntelliSense features such as syntax highlighting and autocomplete suggestions.
2. Right-click in the query window and select **CodeQL: Run Query**. (Alternatively, run the command from the Command Palette.)

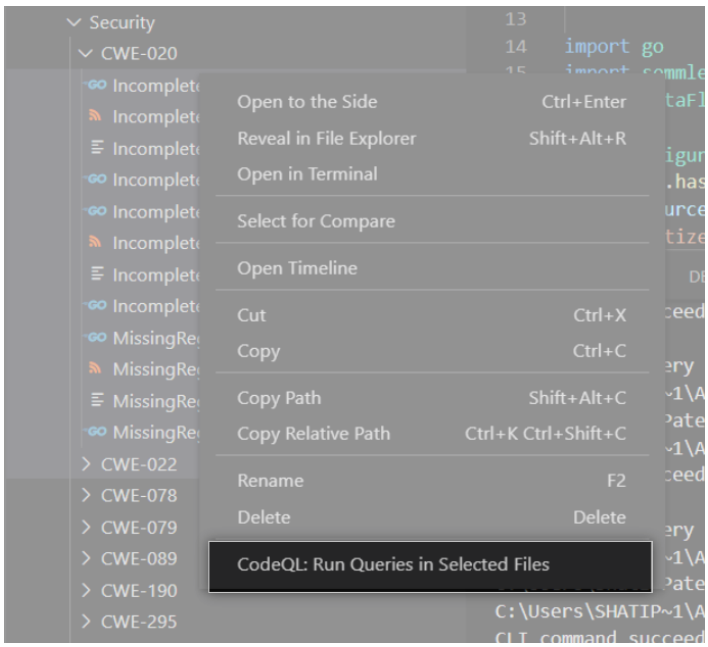
The CodeQL extension runs the query on the current database and reports progress in the bottom right corner of the application. When the results are ready, they're displayed in the Results view.

If there are any problems running a query, a notification is displayed in the bottom right corner of the application. In addition to the error message, the notification includes details of how to fix the problem. For more information, see "*Troubleshooting CodeQL for Visual Studio Code*."

2.3.3 Running multiple queries

You can run multiple queries with a single command.

1. Go to the File Explorer.
2. Select multiple files or folders that contain queries.
3. Right-click and select **CodeQL: Run Queries in Selected Files**.



2.3.4 Running a quick query

When working on a new query, you can open a "quick query" tab to easily execute your code and view the results, without having to save a `.ql` file in your workspace. Open a quick query editing tab by selecting **CodeQL: Quick Query** from the Command Palette. To run the query, use **CodeQL: Run Query**.

You can see all quick queries that you've run in the current session in the Query History view. Click an entry to see the exact text of the quick query that produced the results.

Once you're happy with your quick query, you should save it in a QL pack so you can access it later. For more information, see "*About QL packs*."

2.3.5 Running a specific part of a query or library

This is helpful if you're debugging a query or library and you want to locate the part that is wrong. Instead of using **CodeQL: Run Query** to run the whole query (the *select clause* and any *query predicates*), you can use **CodeQL: Quick Evaluation** to run a specific part of a `.ql` or `.qll` file.

CodeQL: Quick Evaluation evaluates a code snippet (instead of the whole query) and displays results of that selection in the Results view. Possible targets for quick evaluation include:

- Selecting the name of a CodeQL entity (such as a *class* or *predicate*) to evaluate that entity.
- Selecting a *formula* or *expression* with free variables to evaluate that formula or expression.

For example, in the following snippet, you could select the predicate name `foo` or the formula `s = "bar"` for quick evaluation.

```
predicate foo(string s) { s = "bar" }
```

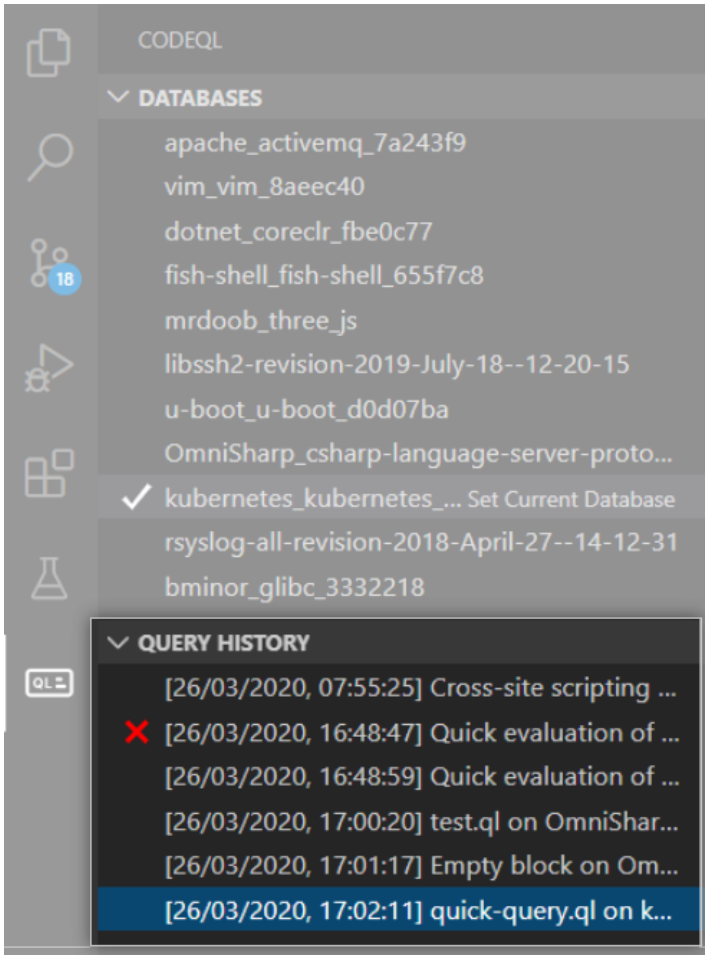
2.3.6 Running a query on multiple databases

This is helpful if you want to test your query on multiple codebases, or find a vulnerability in multiple projects.

1. Open a query (`.ql`) file.
2. Right-click and select **CodeQL: Run Query on Multiple Databases**.
3. From the dropdown menu, select the databases that you want to run the query on.

2.3.7 Viewing previous queries

To see the queries that you have run in the current session, open the Query History view.



The Query History contains information including the date and time when the query was run, the name of the query, the database on which it was run, and how long it took to run the query. To customize the information that is displayed, right-click an entry and select **Set Label**.

Click an entry to display the corresponding results in the Query Results view, and double-click to display the query itself in the editor (or right-click and select **Open Query**). To display the exact text that produced the results for a particular entry, right-click it and select **Show Query Text**. This can differ from **Open Query** as the query file may have been modified since you last ran it.

To remove queries from the Query History view, select all the queries you want to remove, then right-click and select **Remove History Item**.

2.3.8 Viewing query results

1. Click a query in the Query History view to display its results in the Results view.

   Note

   Depending on the query, you can also choose different views such as CSV, *SARIF*, or *DIL format*. For example, to view the DIL format, right-click a result and select **View DIL**. The available output views are determined by the format and the metadata of the query. For more information, see "*CodeQL queries*."

2. Use the dropdown menu in the Results view to choose which results to display, and in what form to display them, such as a formatted alert message or a table of raw results.

3. To sort the results by the entries in a particular column, click the column header.

If a result links to a source code element, you can click it to display it in the source.

To use standard code navigation features in the source code, you can right-click an element and use the commands **Go to Definition** or **Go to References**. This runs a CodeQL query over the active file, which may take a few seconds. This query needs to run once for every file, so any additional references from the same file will be fast.

> Note
>
> If you're using an older database, code navigation commands such as **Go to Definition** and **Go to References** may not work. To use code navigation, try unzipping the database and running `codeql database cleanup <database>` on the unzipped database using the CodeQL CLI. Then, re-add the database to Visual Studio Code. For more information, see the database cleanup reference documentation.

2.3.9 Comparing query results

When you're writing or debugging a query, it's useful to see how your changes affect the results. You can compare two sets of results to see exactly what has changed. To compare results, the two queries must be run on the same database.

1. Right-click a query in the Query History view and select **Compare Results**.
2. A Quick Pick menu shows all valid queries to compare with. Select a query.
3. The Compare view shows the differences in the results of the two queries.

2.3.10 Further reading

- "*CodeQL queries*"
- "*Exploring data flow with path queries*"

2.4 Exploring the structure of your source code

You can use the AST viewer to display the abstract syntax tree of a CodeQL database.

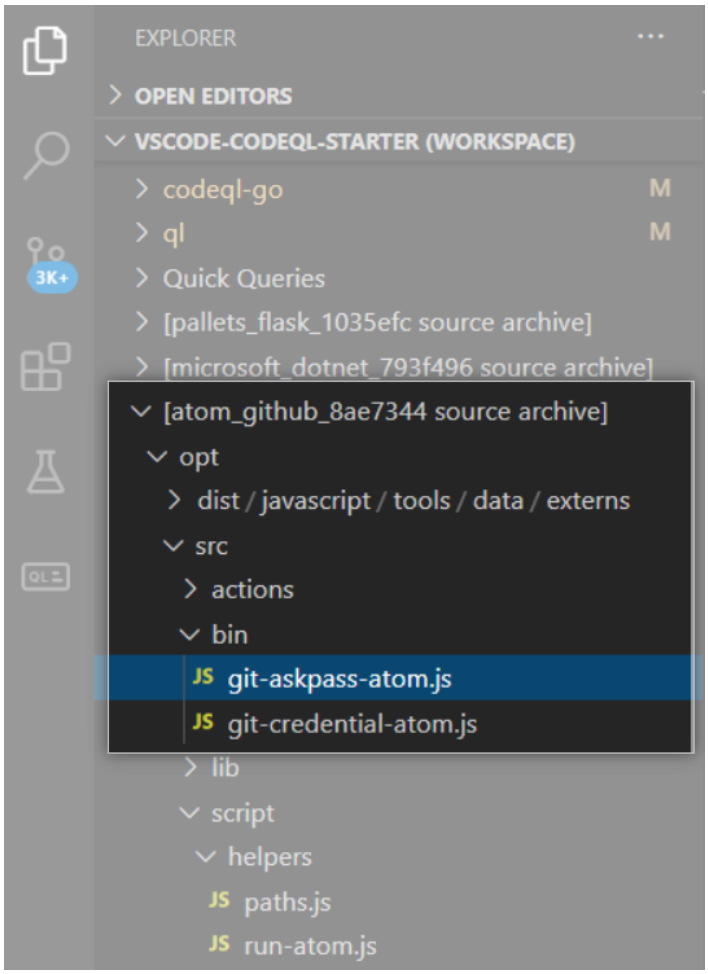
2.4.1 About the abstract syntax tree

The abstract syntax tree (AST) of a program represents the program's syntactic structure. Nodes on the AST represent elements such as statements and expressions. A CodeQL database encodes these program elements and the relationships between them through a *database schema*.

CodeQL for Visual Studio Code contains an AST viewer. The viewer consists of a graph visualization view that lets you explore the AST of a file in a CodeQL database. This helps you see which CodeQL classes correspond to which parts of your source files.

2.4.2 Viewing the abstract syntax tree of a source file

1. Open a source file from a CodeQL database. For example, you can navigate to a source file in the File Explorer.

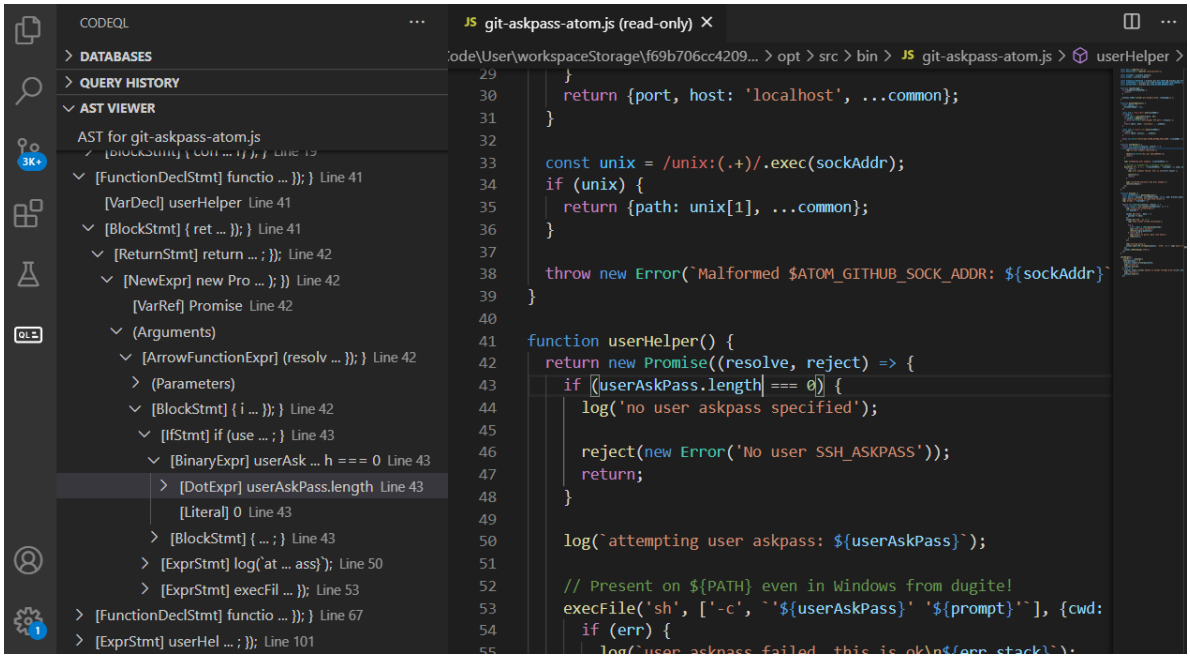


2. Run **CodeQL: View AST** from the Command Palette. This runs a CodeQL query (usually called `printAST.ql`) over the active file, which may take a few seconds.

   Note

   If you don't have an appropriate `printAST.ql` query in your workspace, the **CodeQL: View AST** command won't work. To fix this, you can update your copy of the CodeQL repository (or CodeQL for Go repository) from `main`. If you do this, you may need to upgrade your databases. Also, query caches may be discarded and your next query runs could be slower.

3. Once the query has run, the AST viewer displays the structure of the source file.
4. To see the nested structure, click the arrows and expand the nodes.



You can click a node in the AST viewer to jump to it in the source code. Conversely, if you click a section of the source code, the AST viewer displays the corresponding node.

2.5 Exploring data flow with path queries

You can run CodeQL queries in VS Code to help you track the flow of data through a program, highlighting areas that are potential security vulnerabilities.

2.5.1 About path queries

A path query is a CodeQL query with the property `@kind path-problem`. You can find a number of these in the standard CodeQL libraries, for example, a security query that finds cross-site scripting vulnerabilities in Java projects: Cross-site scripting.

You can run the standard CodeQL path queries to identify security vulnerabilities and manually look through the results. You can also modify the existing queries to model data flow more precisely for the specific framework of your project, or write completely new path queries to find a different vulnerability.

To ensure that your path query uses the correct format and metadata, follow the instructions in "*Creating path queries*." This topic also contains detailed information about how to define new sources and sinks, as well as templates and examples of how to extend the CodeQL libraries to suit your analysis.

2.5.2 Running path queries in VS Code

1. Open a path query in the editor.
2. Right-click in the query window and select **CodeQL: Run Query**. (Alternatively, run the command from the Command Palette.)
3. Once the query has finished running, you can see the results in the Results view as usual (under `alerts` in the dropdown menu). Each query result describes the flow of information between a source and a sink.
4. Expand the result to see the individual steps that the data follows.
5. Click each step to jump to it in the source code and investigate the problem further.
6. To navigate the path from your keyboard, you can bind shortcuts to the **CodeQL: Show Previous Step on Path** and **CodeQL: Show Next Step on Path** commands.

2.5.3 Further reading

- "*About data flow analysis*"
- "*Creating path queries*"

2.6 Testing CodeQL queries in Visual Studio Code

You can run unit tests for CodeQL queries using the Visual Studio Code extension.

2.6.1 About testing queries in VS Code

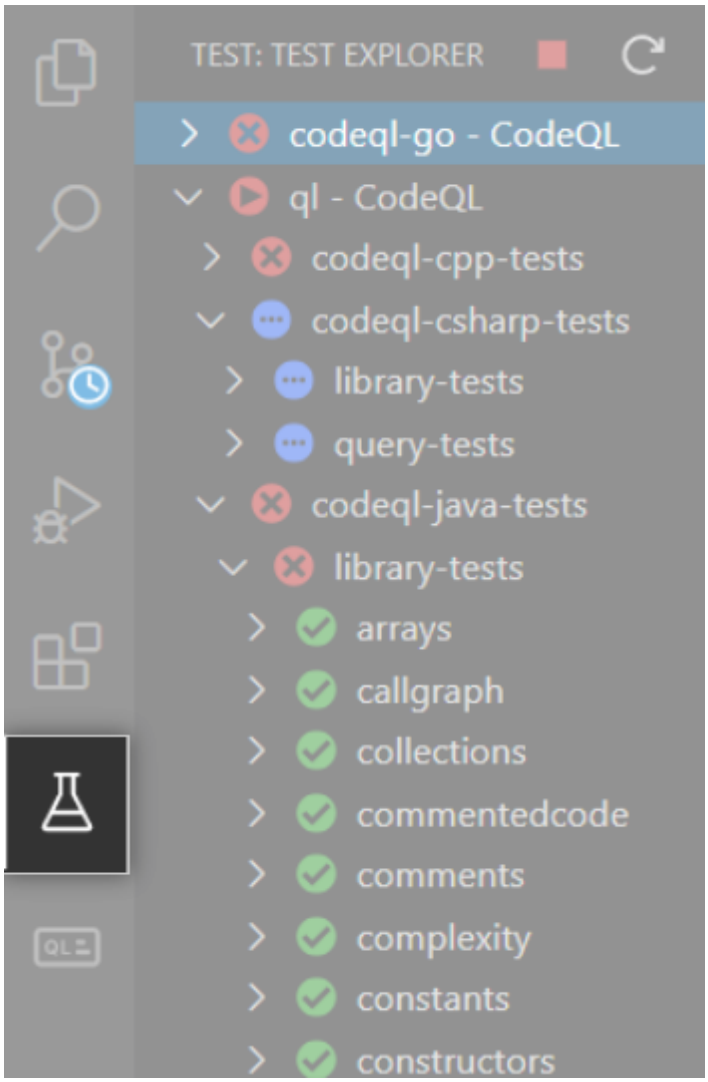
To ensure that your CodeQL queries produce the expected results, you can run tests that compare the expected query results with the actual results.

The CodeQL extension automatically prompts VS Code to install the Test Explorer extension as a dependency. The Test Explorer displays any workspace folders with a name ending in `-tests` and provides a UI for exploring and running tests in those folders.

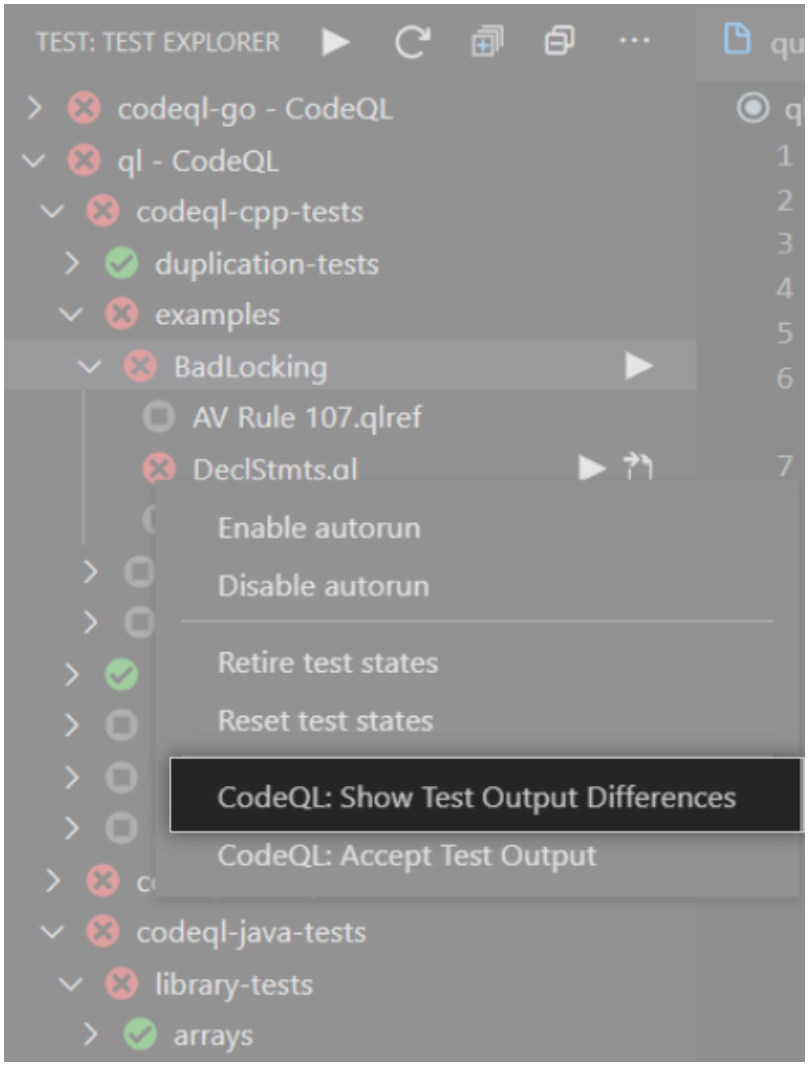
For more information about how CodeQL tests work, see "*Testing custom queries*" in the CLI help.

2.6.2 Testing the results of your queries

1. Open the Test Explorer view in the sidebar.



2. To run a specific test, hover over the file or folder name and click the play button. To run all tests in your workspace, click the play button at the top of the view. If a test takes too long to run, you can click the stop button at the top of the view to cancel the test.
3. The icons show whether a test passed or failed. If it failed, right-click the failed test and click **CodeQL: Show Test Output Differences** to display the differences between the expected output and the actual output.



4. Compare the results. If you want to update the test with the actual output, click **CodeQL: Accept Test Output**.

2.6.3 Monitoring the performance of your queries

Query performance is important when you want to run a query on large databases, or as part of your continuous integration system.

If you want to examine query performance, enable the **Running Queries: Debug** setting to include timing and tuple counts. This is shown in the logs in the CodeQL Query Server tab of the Output view. The tuple count is useful because it indicates the size of the *predicates* computed by the query.

When a query is evaluated, the query server caches the predicates that it calculates. So when you want to compare the performance of two evaluations, you should clear the query server's cache before each run (**CodeQL: Clear Cache** command). This ensures that you're comparing equivalent data.

For more information, see "*Troubleshooting query performance*" and "*Evaluation of QL programs*."

2.6.4 Further reading

- "*Testing custom queries*"

2.7 Working with CodeQL packs in Visual Studio Code

> Note
>
> The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

You can view CodeQL packs and write and edit queries for them in Visual Studio Code.

2.7.1 About CodeQL packs

CodeQL packs are used to create, share, depend on, and run CodeQL queries and libraries. You can publish your own CodeQL packs and download packs created by others. For more information, see "*About CodeQL packs*."

2.7.2 Using standard CodeQL packs in Visual Studio Code

To install dependencies for a CodeQL pack in your Visual Studio Code workspace, run the **CodeQL: Install Pack Dependencies** command from the Command Palette and select the packs you want to install dependencies for.

You can write and run query packs that depend on the CodeQL standard libraries, without needing to check out the standard libraries in your workspace. Instead, you can install only the dependencies required by the query packs you want to use.

2.7.3 Creating and editing CodeQL packs in Visual Studio Code

To create a new CodeQL pack, you will need to use the CodeQL CLI from a terminal, which you can do within Visual Studio Code or outside of it with the `codeql pack init` command. Once you create an empty pack, you can edit the `qlpack.yml` file or run the `codeql pack add` command to add dependencies or change the name or version. For more information, see "*Creating and working with CodeQL packs*."

You can create or edit queries in a CodeQL pack in Visual Studio Code as you would with any CodeQL query, using the standard code editing features such as autocomplete suggestions to find elements to use from the pack's dependencies.

You can then use the CodeQL CLI to publish your pack to share with others. For more information, see "*Publishing and using CodeQL packs*."

2.7.4 Viewing CodeQL packs and their dependencies in Visual Studio Code

To download a CodeQL pack that someone else has created, run the **CodeQL: Download Packs** command from the Command Palette. You can download all the core CodeQL query packs, or enter the full name of a specific pack to download. For example, to download the core queries for analyzing Java, enter `codeql/java-queries`.

Whether you have downloaded a CodeQL pack or created your own, you can open the `qlpack.yml` file in the root of a CodeQL pack directory in Visual Studio Code and view the dependencies section to see what libraries the pack depends on.

If you want to understand a query in a CodeQL pack better, you can open the query file and view the code, using the IntelliSense code editing features of Visual Studio Code. For example, if you hover over an element from a library depended on by the pack, Visual Studio Code will resolve it so you can see documentation about the element.

To view the full definition of an element of a query, you can right-click and choose **Go to Definition**. If the library pack is present within the same Visual Studio Code workspace, this will take you to the definition within the workspace. Otherwise it will take you to the definition within your package cache, the shared location where downloaded dependencies are stored, which is in your home directory by default.

2.8 Customizing settings

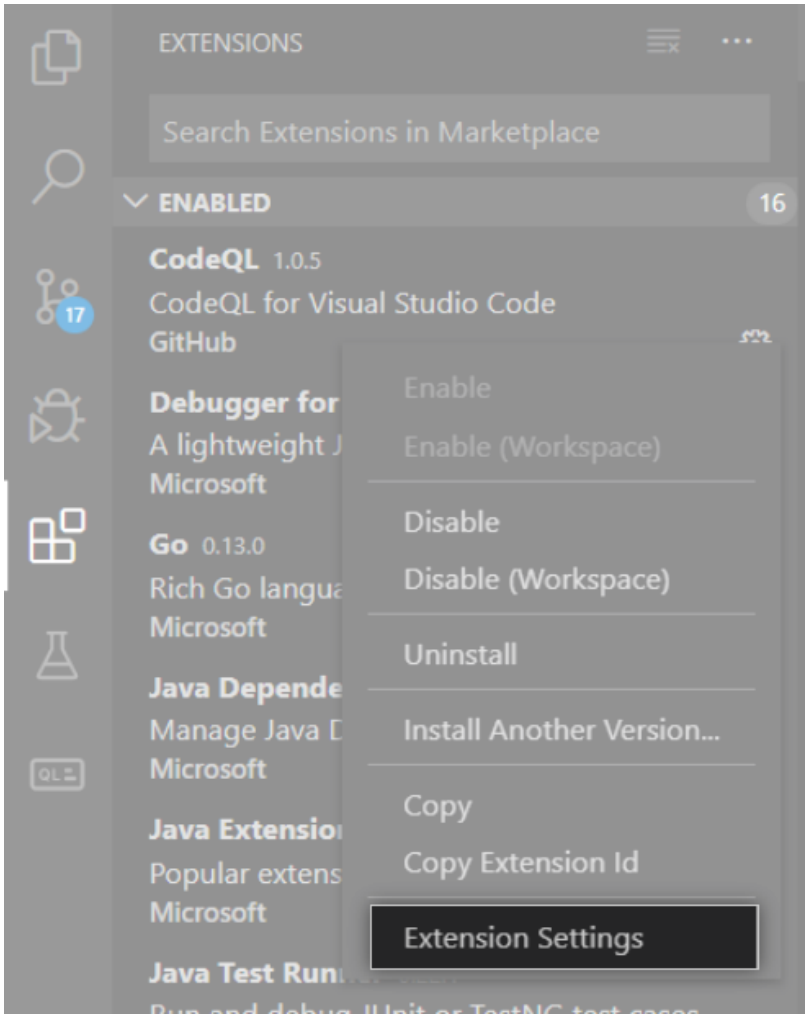
You can edit the settings for the CodeQL extension to suit your needs.

2.8.1 About CodeQL extension settings

The CodeQL extension comes with a number of settings that you can edit. These determine how the extension behaves, including: which version of the CodeQL CLI the extension uses, how the extension displays previous queries, and how it runs queries.

2.8.2 Editing settings

1. Open the Extensions view and right click **CodeQL**.
2. Click **Extension Settings**.



3. Edit a setting. The new settings are saved automatically.

2.8.3 Choosing a version of the CodeQL CLI

The CodeQL extension uses the CodeQL CLI to run commands. If you already have the CLI installed and added to your `PATH`, the extension uses that version. This might be the case if you create your own CodeQL databases instead of downloading them from LGTM.com. Otherwise, the extension automatically manages access to the executable of the CLI for you. For more information about creating databases, see "*Creating CodeQL databases*" in the CLI help.

To override the default behavior and use a different CLI, you can specify the CodeQL CLI **Executable Path**.

2.8.4 Changing the labels of query history items

The query history **Format** setting controls how the extension lists queries in the query history. By default, each item has a label with the following format:

```
%q on %d - %s, %r result count [%t]
```

- `%q` is the query name
- `%d` is the database name
- `%s` is a status string
- `%r` is the number of results
- `%t` is the time the query was run

To override the default label, you can specify a different format for the query history items.

2.8.5 Configuring settings for running queries

There are a number of settings for **Running Queries**. If your queries run too slowly and time out frequently, you may want to increase the memory.

If you want to examine query performance, enable the **Running Queries: Debug** setting to include timing and tuple counts. This is shown in the logs in the CodeQL Query Server tab of the Output view. The tuple count is useful because it indicates the size of the *predicates* computed by the query.

To save query server logs in a custom location, edit the **Running Queries: Custom Log Directory** setting. If you use a custom log directory, the extension saves the logs permanently, instead of deleting them automatically after each workspace session. This is useful if you want to investigate these logs to improve the performance of your queries.

2.8.6 Configuring settings for testing queries

To increase the number of threads used for testing queries, you can update the **Running Tests > Number Of Threads** setting.

To pass additional arguments to the CodeQL CLI when running tests, you can update the **Running Tests > Additional Test Arguments** setting. For more information about the available arguments, see "test run" in the CodeQL CLI help.

2.8.7 Configuring settings for telemetry and data collection

You can configure whether the CodeQL extension collects telemetry data. This is disabled by default. For more information, see "*About telemetry in CodeQL for Visual Studio Code*."

2.8.8 Further reading

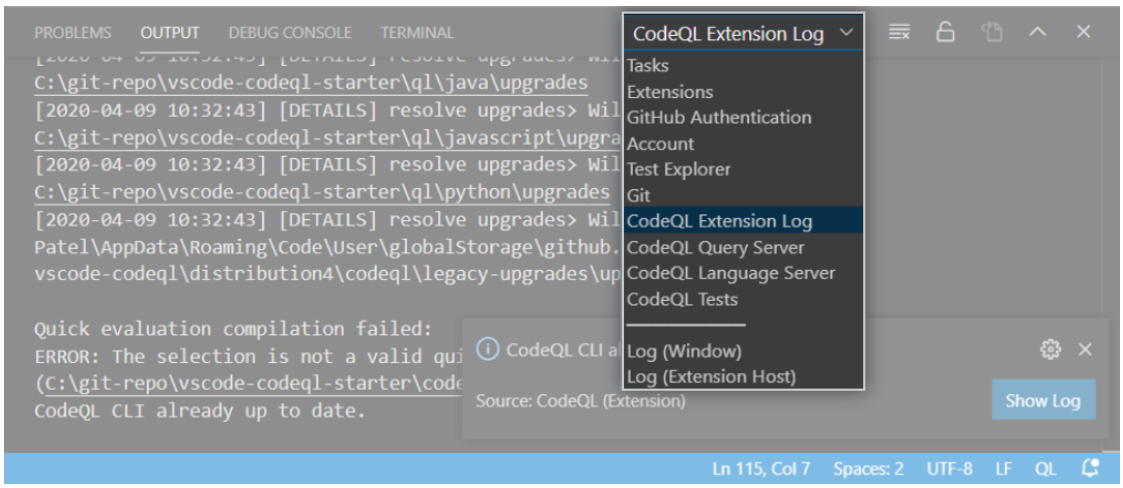
- User and workspace settings in the Visual Studio Code help
- "*CodeQL CLI*"

2.9 Troubleshooting CodeQL for Visual Studio Code

You can use the detailed information written to the extension's log files if you need to troubleshoot problems.

2.9.1 About the log files

Progress and error messages are displayed as notifications in the bottom right corner of the workspace. These link to more detailed logs and error messages in the Output panel. You can use the dropdown list to select the logs you need.



2.9.2 Troubleshooting installation and configuration problems

If you encounter any problems when installing and configuring the extension, check the CodeQL Extension Log to see general extension logging messages, including details about the CodeQL CLI and the commands invoked by the extension.

In particular, you can see the location of the CLI that is being used. This is useful if you want to see whether this is an extension-managed CLI or an external one.

If you use the extension-managed CLI, the extension checks for updates automatically (or with the **CodeQL: Check for CLI Updates** command) and prompts you to accept the updated version. If you use an external CLI, you need to update it manually (when updates are necessary).

2.9.3 Exploring problems with queries and databases

For details about compiling and running queries, as well as information about database upgrades, check the CodeQL Query Server log.

If you see behavior or errors that suggest problems, you can use the **CodeQL: Restart Query Server** command to restart the query server. This restarts the server without affecting your CodeQL session history. You are most likely to need to restart the query server if you make external changes to files that the extension is using. For example, regenerating a CodeQL database that's open in VS Code. In addition to problems in the log, you might also see: errors in code highlighting, incorrect results totals, or duplicate notifications that a query is running.

To see the logs from running a particular query, right-click the query in the Query History and select **Show Query Log**. If the log file is too large for the extension to open in the VS Code editor, the file will be displayed in your file explorer so you can open it with an external program.

By default, the extension deletes logs after each workspace session. To override this behavior, you can specify a custom directory for query server logs. For more information, see "*Customizing settings*."

2.9.4 Exploring problems with running tests

To see more detailed output from running unit tests, open the CodeQL Tests log. For more information about tests, see "*Testing CodeQL queries in Visual Studio Code*."

2.9.5 Generating a bug report for GitHub

The CodeQL Language Server contains more advanced debug logs for CodeQL language maintainers. You should only need these to provide details in a bug report.

2.10 About telemetry in CodeQL for Visual Studio Code

If you specifically opt in to permit GitHub to do so, GitHub will collect usage data and metrics for the purposes of helping the core developers to improve the CodeQL extension for VS Code.

This data will not be shared with any parties outside of GitHub. IP addresses and installation IDs will be retained for a maximum of 30 days. Anonymous data will be retained for a maximum of 180 days.

2.10.1 Why we collect data

GitHub collects aggregated, anonymous usage data and metrics to help us improve CodeQL for VS Code. IP addresses and installation IDs are collected only to ensure that anonymous data is not duplicated during aggregation.

2.10.2 What data is collected

If you opt in, GitHub collects the following information related to the usage of the extension. The data collected are:

- The identifiers of any CodeQL-related VS Code commands that are run.
- For each command: the timestamp, time taken, and whether or not the command completed successfully.
- VS Code and extension version.
- Randomly generated GUID that uniquely identifies a CodeQL extension installation. (Discarded before aggregation.)
- IP address of the client sending the telemetry data. (Discarded before aggregation.)
- Whether or not the `codeQL.canary` setting is enabled and set to `true`.

2.10.3 How long data is retained

IP address and GUIDs will be retained for a maximum of 30 days. Anonymous, aggregated data that includes command identifiers, run times, and timestamps will be retained for a maximum of 180 days.

2.10.4 Access to the data

IP address and GUIDs will only be available to the core developers of CodeQL. Aggregated data will be available to GitHub employees.

2.10.5 What data is NOT collected

We only collect the minimal amount of data we need to answer the questions about how our users are experiencing this product. To that end, we do not collect the following information:

- No GitHub user ID
- No CodeQL database names or contents
- No contents of CodeQL queries
- No filesystem paths

2.10.6 Disabling telemetry reporting

Telemetry collection is *disabled* by default.

When telemetry collection is disabled, no data will be sent to GitHub servers.

You can disable telemetry collection by setting `codeQL.telemetry.enableTelemetry` to `false` in your settings. For more information about CodeQL settings, see "*Customizing settings*."

Additionally, telemetry collection will be disabled if the global `telemetry.enableTelemetry` setting is set to `false`. For more information about global telemetry collection, see "Microsoft's documentation."

2.10.7 Further reading

For more information, see GitHub's "Privacy Statement" and "Terms of Service."

CHAPTER

THREE

CODEQL CLI

The CodeQL command-line interface (CLI) is used to create databases for security research. You can query CodeQL databases directly from the command line or using the Visual Studio Code extension.

- *Using the CodeQL CLI*: Software developers and security researchers can secure their code using the CodeQL CLI.
- *CodeQL CLI reference*: Learn more about the files you can use when running CodeQL processes and the results format and exit codes that CodeQL generates.
- CodeQL CLI manual: Detailed information about all the commands available with the CodeQL CLI.

3.1 Using the CodeQL CLI

The CodeQL command-line interface (CLI) is used to create databases for security research. You can query CodeQL databases directly from the command line or using the Visual Studio Code extension.

See the following links to learn how to get set up and run CodeQL commands:

- *About the CodeQL CLI*: Software developers and security researchers can secure their code using the CodeQL CLI.
- *Getting started with the CodeQL CLI*: Set up the CodeQL CLI so that you can run CodeQL processes from your command line.
- *Creating CodeQL databases*: Create relational representations of source code that can be queried like any other database.
- *Extractor options*: Set options for the behavior of extractors that create CodeQL databases.
- *Analyzing CodeQL databases with the CodeQL CLI*: Analyze your code using queries written in a specially-designed, object-oriented query language.
- *Upgrading CodeQL databases*: Upgrade your databases so that they can be analyzed using the most up to date CodeQL products.
- *Using custom queries with the CodeQL CLI*: Use custom queries to extend your analysis or highlight errors that are specific to a particular codebase.
- *Creating CodeQL query suites*: Define query suite definitions for groups of frequently used queries.
- *Testing custom queries*: Set up regression testing of custom queries to ensure that they behave as expected in your analysis.
- *Testing query help files*: Test query help files by rendering them as markdown to ensure they are valid before adding them to the CodeQL repository or using them in code scanning.
- *Creating and working with CodeQL packs*: Create, share, depend on, and run CodeQL queries and libraries.

- *Publishing and using CodeQL packs*: Publish your own or use others CodeQL packs for code scanning.
- *Specifying command options in a CodeQL configuration file*: You can save default or frequently used options for your commands in a per-user configuration file.

3.1.1 About the CodeQL CLI

Software developers and security researchers can secure their code using the CodeQL CLI.

The CodeQL CLI is a command-line tool used to run CodeQL processes locally on open source software projects. You can use the CodeQL CLI to:

- Run CodeQL analyses using queries provided by GitHub engineers and the open source community
- Create CodeQL databases to use in the CodeQL for Visual Studio Code
- Develop and test custom CodeQL queries to use in your own analyses

For information about using the CodeQL CLI, see "*Getting started with the CodeQL CLI*."

CodeQL CLI commands

The CodeQL CLI includes commands to create and analyze CodeQL databases from the command line. To run a command, use:

```
codeql [command] [subcommand]
```

To view the reference documentation for a command, add the `--help` flag, or visit the "CodeQL CLI manual."

3.1.2 Getting started with the CodeQL CLI

To run CodeQL commands, you need to set up the CLI so that it can access the tools, queries, and libraries required to create and analyze databases.

> License notice
>
> If you don't have an Enterprise license then, by installing this product, you are agreeing to the GitHub CodeQL Terms and Conditions.
>
> GitHub CodeQL is licensed on a per-user basis. Under the license restrictions, you can use CodeQL to perform the following tasks:
>
> - To perform academic research.
> - To demonstrate the software.
> - To test CodeQL queries that are released under an OSI-approved License to confirm that new versions of those queries continue to find the right vulnerabilities.
>
> where "OSI-approved License" means an Open Source Initiative (OSI)-approved open source software license.
>
> If you are working with an Open Source Codebase (that is, a codebase that is released under an OSI-approved License) you can also use CodeQL for the following tasks:
>
> - To perform analysis of the Open Source Codebase.
> - If the Open Source Codebase is hosted and maintained on GitHub.com, to generate CodeQL databases for or during automated analysis, continuous integration, or continuous delivery.

CodeQL can't be used for automated analysis, continuous integration or continuous delivery, whether as part of normal software engineering processes or otherwise, except in the express cases set forth herein. For these uses, contact the sales team.

Setting up the CodeQL CLI

The CodeQL CLI can be set up to support many different use cases and directory structures. To get started quickly, we recommend adopting a relatively simple setup, as outlined in the steps below.

If you use Linux, Windows, or macOS version 10.14 ("Mojave") or earlier, simply follow the steps below. For macOS version 10.15 ("Catalina") or newer, steps 1 and 4 are slightly different—for further details, see the sections labeled **Information for macOS "Catalina" (or newer) users**. If you are using macOS on Apple Silicon (e.g. Apple M1), ensure that the Xcode command-line developer tools and Rosetta 2 are installed.

Note

The CodeQL CLI is currently not compatible with non-glibc Linux distributions such as (muslc-based) Alpine Linux.

For information about installing the CodeQL CLI in a CI system to create results to display in GitHub as code scanning alerts, see Installing CodeQL CLI in your CI system in the GitHub documentation.

1. Download the CodeQL CLI zip package

The CodeQL CLI download package is a zip archive containing tools, scripts, and various CodeQL-specific files. If you don't have an Enterprise license then, by downloading this archive, you are agreeing to the GitHub CodeQL Terms and Conditions.

Important

There are several different versions of the CLI available to download, depending on your use case:

- If you want to use the most up to date CodeQL tools and features, download the version tagged `latest`.
- If you want to create CodeQL databases to upload to LGTM Enterprise, download the version that is compatible with the relevant LGTM Enterprise version number. Compatibility information is included in the description for each release on the CodeQL CLI releases page on GitHub. Using the correct version of the CLI ensures that your CodeQL databases are compatible with your version of LGTM Enterprise. For more information, see Preparing CodeQL databases to upload to LGTM in the LGTM admin help.

If you use Linux, Windows, or macOS version 10.14 ("Mojave") or earlier, simply download the zip archive for the version you require.

If you want the CLI for a specific platform, download the appropriate `codeql-PLATFORM.zip` file. Alternatively, you can download `codeql.zip`, which contains the CLI for all supported platforms.

Information for macOS "Catalina" (or newer) users

macOS "Catalina" (or newer)

If you use macOS version 10.15 ("Catalina"), version 11 ("Big Sur"), or the upcoming version 12 ("Monterey"), you need to ensure that your web browser does not automatically extract zip files. If you use Safari, complete the following steps before downloading the CodeQL CLI zip archive:

i. Open Safari.

ii. From the Safari menu, select **Preferences...**.

iii. Click the **General** Tab.

iv. Ensure the check-box labeled **Open "safe" files after downloading**. is unchecked.

2. Extract the zip archive

For Linux, Windows, and macOS users (version 10.14 "Mojave", and earlier) simply extract the zip archive.

Information for macOS "Catalina" (or newer) users

macOS "Catalina"

macOS "Catalina", "Big Sur", or "Monterey" users should run the following commands in the Terminal, where `${extraction-root}` is the path to the directory where you will extract the CodeQL CLI zip archive:

i. `mv ~/Downloads/codeql*.zip ${extraction-root}`

ii. `cd ${extraction-root}`

iii. `/usr/bin/xattr -c codeql*.zip`

iv. `unzip codeql*.zip`

3. Launch `codeql`

Once extracted, you can run CodeQL processes by running the `codeql` executable in a couple of ways:

- By executing `<extraction-root>/codeql/codeql`, where `<extraction-root>` is the folder where you extracted the CodeQL CLI package.
- By adding `<extraction-root>/codeql` to your `PATH`, so that you can run the executable as just `codeql`.

At this point, you can execute CodeQL commands. For a full list of the CodeQL CLI commands, see the "CodeQL CLI manual."

Note

If you add `codeql` to your `PATH`, it can be accessed by CodeQL for Visual Studio Code to compile and run queries. For more information about configuring VS Code to access the CodeQL CLI, see "*Setting up CodeQL in Visual Studio Code*."

4. Verify your CodeQL CLI setup

CodeQL CLI has subcommands you can execute to verify that you are correctly set up to create and analyze databases:

- Run `codeql resolve languages` to show which languages are available for database creation. This will list the languages supported by default in your CodeQL CLI package.
- (Optional) You can download some "*CodeQL packs*" containing pre-compiled queries you would like to run. To do this, run `codeql pack download <pack-name> [...pack-name]`, where `pack-name` is the name of the pack you want to download. The core query packs are a good place to start. They are:
  - `codeql/cpp-queries`
  - `codeql/csharp-queries`
  - `codeql/java-queries`
  - `codeql/javascript-queries`

- `codeql/python-queries`
- `codeql/ruby-queries`

Alternatively, you can download query packs during the analysis by using the `--download` flag of the `codeql database analyze` command.

Checking out the CodeQL source code directly

Some users prefer working with CodeQL query sources directly in order to work on or contribute to the Open Source shared queries. In order to do this, the following steps are recommended. Note that the following instructions are a slightly more complicated alternative to working with CodeQL packages as explained above.

1. Download the CodeQL CLI zip

Follow *step 1 from the previous section*.

2. Create a new CodeQL directory

Create a new directory where you can place the CLI and any queries and libraries you want to use. For example, `$HOME/codeql-home`.

The CLI's built-in search operations automatically look in all of its sibling directories for the files used in database creation and analysis. Keeping these components in their own directory prevents the CLI searching unrelated sibling directories while ensuring all files are available without specifying any further options on the command line.

3. Obtain a local copy of the CodeQL queries

The CodeQL repository contains the queries and libraries required for CodeQL analysis of C/C++, C#, Java, JavaScript/TypeScript, Python, and Ruby. Clone a copy of this repository into `codeql-home`.

By default, the root of the cloned repository will be called `codeql`. Rename this folder `codeql-repo` to avoid conflicting with the CodeQL CLI that you will extract in step 4. If you use git on the command line, you can clone and rename the repository in a single step by running `git clone git@github.com:github/codeql.git codeql-repo` in the `codeql-home` folder.

The CodeQL libraries and queries for Go analysis live in the CodeQL for Go repository. Clone a copy of this repository into `codeql-home`, and run `codeql-go/scripts/install-deps.sh` to install its dependencies.

The cloned repositories should have a sibling relationship. For example, if the root of the cloned CodeQL repository is `$HOME/codeql-home/codeql-repo`, then the root of the cloned CodeQL for Go repository should be `$HOME/codeql-home/codeql-go`.

Within these repositories, the queries and libraries are organized into QL packs. Along with the queries themselves, QL packs contain important metadata that tells the CodeQL CLI how to process the query files. For more information, see "*About QL packs*."

Important

There are different versions of the CodeQL queries available for different users. Check out the correct version for your use case:

- For the queries used on LGTM.com, check out the `lgtm.com` branch. You should use this branch for databases you've built using the CodeQL CLI, fetched from code scanning on GitHub, or recently downloaded from LGTM.com. The queries on the `lgtm.com` branch are more likely to be compatible

with the `latest` CLI, so you'll be less likely to have to upgrade newly-created databases than if you use the `main` branch. Older databases may need to be upgraded before you can analyze them.

- For the most up to date CodeQL queries, check out the `main` branch. This branch represents the very latest version of CodeQL's analysis. Even databases created using the most recent version of the CLI may have to be upgraded before you can analyze them. For more information, see "*Upgrading CodeQL databases*."
- For the queries used in a particular LGTM Enterprise release, check out the branch tagged with the relevant release number. For example, the branch tagged `v1.27.0` corresponds to LGTM Enterprise 1.27. You must use this version if you want to upload data to LGTM Enterprise. For further information, see Preparing CodeQL databases to upload to LGTM in the LGTM admin help.

4. Extract the zip archive

For Linux, Windows, and macOS users (version 10.14 "Mojave", and earlier) simply extract the zip archive into the directory you created in step 2.

For example, if the path to your copy of the CodeQL repository is `$HOME/codeql-home/codeql-repo`, then extract the CLI into `$HOME/codeql-home/`.

5. Launch `codeql`

See *step 3 from the previous section*.

6. Verify your CodeQL CLI setup

CodeQL CLI has subcommands you can execute to verify that you are correctly set up to create and analyze databases:

- Run `codeql resolve languages` to show which languages are available for database creation. This will list the languages supported by default in your CodeQL CLI package.
- Run `codeql resolve qlpacks` to show which QL packs the CLI can find. This will display the names of all the QL packs directly available to the CodeQL CLI. This should include:
  - Query packs for each supported language, for example, `codeql/{language}-queries`. These packs contain the standard queries that will be run for each analysis.
  - Library packs for each supported language, for example, `codeql/{language}-all`. These packs contain query libraries, such as control flow and data flow libraries, that may be useful to query writers.
  - Example packs for each supported language, for example, `codeql/{language}-examples`. These packs contain useful snippets of CodeQL that query writers may find useful.
  - Legacy packs that ensure custom queries and libraries created using older products are compatible with your version of CodeQL.

Using two versions of the CodeQL CLI

If you want to use the latest CodeQL features to execute queries or CodeQL tests, but also want to prepare databases that are compatible with a specific version of LGTM Enterprise, you may need to install two versions of the CLI. The recommended directory setup depends on which versions you want to install:

- If both versions are 2.0.2 (or newer), you can unpack both CLI archives in the same parent directory.
- If at least one of the versions is 2.0.1 (or older), the unpacked CLI archives cannot be in the same parent directory, but they can share the same grandparent directory. For example, if you unpack version 2.0.2 into `$HOME/codeql-home/codeql-cli`, the older version should be unpacked into `$HOME/codeql-older-version/old-codeql-cli`. Here, the common grandparent is the `$HOME` directory.

3.1.3 Creating CodeQL databases

Before you analyze your code using CodeQL, you need to create a CodeQL database containing all the data required to run queries on your code.

CodeQL analysis relies on extracting relational data from your code, and using it to build a *CodeQL database*. CodeQL databases contain all of the important information about a codebase, which can be analyzed by executing CodeQL queries against it. Before you generate a CodeQL database, you need to:

- Install and set up the CodeQL CLI. For more information, see "*Getting started with the CodeQL CLI*."
- Check out the version of your codebase you want to analyze. The directory should be ready to build, with all dependencies already installed.

For information about using the CodeQL CLI in a third-party CI system to create results to display in GitHub as code scanning alerts, see Configuring CodeQL CLI in your CI system in the GitHub documentation. For information about enabling CodeQL code scanning using GitHub Actions, see Setting up code scanning for a repository in the GitHub documentation.

Running `codeql database create`

CodeQL databases are created by running the following command from the checkout root of your project:

```
codeql database create <database> --language=<language-identifier>
```

You must specify:

- `<database>`: a path to the new database to be created. This directory will be created when you execute the command—you cannot specify an existing directory.
- `--language`: the identifier for the language to create a database for. When used with `--db-cluster`, the option accepts a comma-separated list, or can be specified more than once. CodeQL supports creating databases for the following languages:

Language	Identifier
C/C++	`cpp`
C#	`csharp`
Go	`go`
Java	`java`
JavaScript/TypeScript	`javascript`
Python	`python`
Ruby	`ruby`

You can specify additional options depending on the location of your source file, if the code needs to be compiled, and if you want to create CodeQL databases for more than one language:

- `--source-root`: the root folder for the primary source files used in database creation. By default, the command assumes that the current directory is the source root—use this option to specify a different location.
- `--db-cluster`: use for multi-language codebases when you want to create databases for more than one language.
- `--command`: used when you create a database for one or more compiled languages, omit if the only languages requested are Python and JavaScript. This specifies the build commands needed to invoke the compiler. Commands are run from the current folder, or `--source-root` if specified. If you don't include a `--command`, CodeQL will attempt to detect the build system automatically, using a built-in autobuilder.
- `--no-run-unnecessary-builds`: used with `--db-cluster` to suppress the build command for languages where the CodeQL CLI does not need to monitor the build (for example, Python and JavaScript/TypeScript).

You can specify extractor options to customize the behavior of extractors that create CodeQL databases. For more information, see "*Extractor options*."

For full details of all the options you can use when creating databases, see the database create reference documentation.

Progress and results

Errors are reported if there are any problems with the options you have specified. For interpreted languages, the extraction progress is displayed in the console—for each source file, it reports if extraction was successful or if it failed. For compiled languages, the console will display the output of the build system.

When the database is successfully created, you'll find a new directory at the path specified in the command. If you used the `--db-cluster` option to create more than one database, a subdirectory is created for each language. Each CodeQL database directory contains a number of subdirectories, including the relational data (required for analysis) and a source archive—a copy of the source files made at the time the database was created—which is used for displaying analysis results.

Creating databases for non-compiled languages

The CodeQL CLI includes extractors to create databases for non-compiled languages—specifically, JavaScript (and TypeScript), Python, and Ruby. These extractors are automatically invoked when you specify JavaScript, Python, or Ruby as the `--language` option when executing `database create`. When creating databases for these languages you must ensure that all additional dependencies are available.

> Important
>
> When you run `database create` for JavaScript, TypeScript, Python, and Ruby, you should not specify a `--command` option. Otherwise this overrides the normal extractor invocation, which will create an empty database. If you create databases for multiple languages and one of them is a compiled language, use the `--no-run-unnecessary-builds` option to skip the command for the languages that don't need to be compiled.

JavaScript and TypeScript

Creating databases for JavaScript requires no additional dependencies, but if the project includes TypeScript files, you must install Node.js 6.x or later. In the command line you can specify `--language=javascript` to extract both JavaScript and TypeScript files:

```
codeql database create --language=javascript --source-root <folder-to-extract> <output-
↪folder>/javascript-database
```

Here, we have specified a `--source-root` path, which is the location where database creation is executed, but is not necessarily the checkout root of the codebase.

By default, files in `node_modules` and `bower_components` directories are not extracted.

Python

When creating databases for Python you must ensure:

- You have the all of the required versions of Python installed.
- You have access to the pip packaging management system and can install any packages that the codebase depends on.
- You have installed the virtualenv pip module.

In the command line you must specify `--language=python`. For example::

```
codeql database create --language=python <output-folder>/python-database
```

This executes the `database create` subcommand from the code's checkout root, generating a new Python database at `<output-folder>/python-database`.

Ruby

Creating databases for Ruby requires no additional dependencies. In the command line you must specify `--language=ruby`. For example:

```
codeql database create --language=ruby --source-root <folder-to-extract> <output-folder>/
↪ruby-database
```

Here, we have specified a `--source-root` path, which is the location where database creation is executed, but is not necessarily the checkout root of the codebase.

Creating databases for compiled languages

For compiled languages, CodeQL needs to invoke the required build system to generate a database, therefore the build method must be available to the CLI.

Detecting the build system

The CodeQL CLI includes autobuilders for C/C++, C#, Go, and Java code. CodeQL autobuilders allow you to build projects for compiled languages without specifying any build commands. When an autobuilder is invoked, CodeQL examines the source for evidence of a build system and attempts to run the optimal set of commands required to extract a database.

An autobuilder is invoked automatically when you execute `codeql database create` for a compiled `--language` if don't include a `--command` option. For example, for a Java codebase, you would simply run:

```
codeql database create --language=java <output-folder>/java-database
```

If a codebase uses a standard build system, relying on an autobuilder is often the simplest way to create a database. For sources that require non-standard build steps, you may need to explicitly define each step in the command line.

> Creating databases for Go
>
> For Go, install the Go toolchain (version 1.11 or later) and, if there are dependencies, the appropriate dependency manager (such as dep).
>
> The Go autobuilder attempts to automatically detect code written in Go in a repository, and only runs build scripts in an attempt to fetch dependencies. To force CodeQL to limit extraction to the files compiled by your build script, set the environment variable *CODEQL_EXTRACTOR_GO_BUILD_TRACING=on* or use the `--command` option to specify a build command.

Specifying build commands

The following examples are designed to give you an idea of some of the build commands that you can specify for compiled languages.

> Important
>
> The `--command` option accepts a single argument—if you need to use more than one command, specify `--command` multiple times.
>
> If you need to pass subcommands and options, the whole argument needs to be quoted to be interpreted correctly.

- C/C++ project built using `make`:

```
codeql database create cpp-database --language=cpp --command=make
```

- C# project built using `dotnet build`:

```
For C# projects using either `dotnet build` or `msbuild`, you should specify `/
↪p:UseSharedCompilation=false`
in the build command. It is also a good idea to add `/t:rebuild` to ensure that all␣
↪code will be built (code
that is not built will not be included in the CodeQL database):

codeql database create csharp-database --language=csharp --command='dotnet build /
↪p:UseSharedCompilation=false /t:rebuild'
```

- Go project built using the `CODEQL_EXTRACTOR_GO_BUILD_TRACING=on` environment variable:

```
CODEQL_EXTRACTOR_GO_BUILD_TRACING=on codeql database create go-database --
↪language=go
```

- Go project built using a custom build script:

```
codeql database create go-database --language=go --command='./scripts/build.sh'
```

- Java project built using Gradle:

```
codeql database create java-database --language=java --command='gradle clean test'
```

- Java project built using Maven:

```
codeql database create java-database --language=java --command='mvn clean install'
```

- Java project built using Ant:

```
codeql database create java-database --language=java --command='ant -f build.xml'
```

- Project built using Bazel:

```
# Navigate to the Bazel workspace.

# Before building, remove cached objects
# and stop all running Bazel server processes.
bazel clean --expunge

# Build using the following Bazel flags, to help CodeQL detect the build:
# `--spawn_strategy=local`: build locally, instead of using a distributed build
# `--nouse_action_cache`: turn off build caching, which might prevent recompilation␣
↪of source code
# `--noremote_accept_cached`, `--noremote_upload_local_results`: avoid using a␣
↪remote cache
codeql database create new-database --language=<language> \
  --command='bazel build --spawn_strategy=local --nouse_action_cache --noremote_
↪accept_cached --noremote_upload_local_results //path/to/package:target'

# After building, stop all running Bazel server processes.
# This ensures future build commands start in a clean Bazel server process
# without CodeQL attached.
bazel shutdown
```

- Project built using a custom build script:

```
codeql database create new-database --language=<language> --command='./scripts/
↪build.sh'
```

  This command runs a custom script that contains all of the commands required to build the project.

Using indirect build tracing

If the CodeQL CLI autobuilders for compiled languages do not work with your CI workflow and you cannot wrap invocations of build commands with `codeql database trace-command`, you can use indirect build tracing to create a CodeQL database. To use indirect build tracing, your CI system must be able to set custom environment variables for each build action.

To create a CodeQL database with indirect build tracing, run the following command from the checkout root of your project:

```
codeql database init ... --begin-tracing <database>
```

You must specify:

- `<database>`: a path to the new database to be created. This directory will be created when you execute the command—you cannot specify an existing directory.
- `--begin-tracing`: creates scripts that can be used to set up an environment in which build commands will be traced.

You may specify other options for the `codeql database init` command as normal.

> Note
>
> If the build runs on Windows, you must set either `--trace-process-level <number>` or `--trace-process-name <parent process name>` so that the option points to a parent CI process that will observe all build steps for the code being analyzed.

The `codeql database init` command will output a message:

```
Created skeleton <database>. This in-progress database is ready to be populated by an␣
↪extractor.
In order to initialise tracing, some environment variables need to be set in the shell␣
↪your build will run in.
A number of scripts to do this have been created in <database>/temp/tracingEnvironment.
Please run one of these scripts before invoking your build command.

Based on your operating system, we recommend you run: ...
```

The `codeql database init` command creates `<database>/temp/tracingEnvironment` with files that contain environment variables and values that will enable CodeQL to trace a sequence of build steps. These files are named `start-tracing.{json,sh,bat,ps1}`. Use one of these files with your CI system's mechanism for setting environment variables for future steps. You can:

- Read the JSON file, process it, and print out environment variables in the format expected by your CI system. For example, Azure DevOps expects `echo "##vso[task.setvariable variable=NAME]VALUE"`.
- Or, if your CI system persists the environment, source the appropriate `start-tracing` script to set the CodeQL variables in the shell environment of the CI system.

Build your code; optionally, unset the environment variables using an `end-tracing.{json,sh,bat,ps1}` script from the directory where the `start-tracing` scripts are stored; and then run the command `codeql database finalize <database>`.

Once you have created a CodeQL database using indirect build tracing, you can work with it like any other CodeQL database. For example, analyze the database, and upload the results to GitHub if you use code scanning.

Example of creating a CodeQL database using indirect build tracing

The following example shows how you could use indirect build tracing in an Azure DevOps pipeline to create a CodeQL database:

```
steps:
    # Download the CodeQL CLI and query packs...
    # Check out the repository ...

    # Run any pre-build tasks, for example, restore NuGet dependencies...

    # Initialize the CodeQL database.
    # In this example, the CodeQL CLI has been downloaded and placed on the PATH.
    - task: CmdLine@1
       displayName: Initialize CodeQL database
      inputs:
          # Assumes the source code is checked out to the current working directory.
          # Creates a database at `<current working directory>/db`.
          # Running on Windows, so specifies a trace process level.
          script: "codeql database init --language csharp --trace-process-name Agent.
↪Worker.exe --source-root . --begin-tracing db"

    # Read the generated environment variables and values,
    # and set them so they are available for subsequent commands
    # in the build pipeline. This is done in PowerShell in this example.
    - task: PowerShell@1
       displayName: Set CodeQL environment variables
       inputs:
          targetType: inline
          script: >
             $json = Get-Content $(System.DefaultWorkingDirectory)/db/temp/
↪tracingEnvironment/start-tracing.json | ConvertFrom-Json
             $json.PSObject.Properties | ForEach-Object {
                 $template = "##vso[task.setvariable variable="
                 $template += $_.Name
                 $template += "]"
                 $template += $_.Value
                 echo "$template"
             }

    # Execute the pre-defined build step. Note the `msbuildArgs` variable.
    - task: VSBuild@1
        inputs:
          solution: '**/*.sln'
          # Disable MSBuild shared compilation for C# builds.
          msbuildArgs: /p:OutDir=$(Build.ArtifactStagingDirectory) /
↪p:UseSharedCompilation=false
          platform: Any CPU
          configuration: Release
          # Execute a clean build, in order to remove any existing build artifacts prior␣
↪to the build.
          clean: True
       displayName: Visual Studio Build
```

(continues on next page)

(continued from previous page)

```
   # Read and set the generated environment variables to end build tracing. This is␣
↪done in PowerShell in this example.
   - task: PowerShell@1
      displayName: Clear CodeQL environment variables
      inputs:
         targetType: inline
         script: >
            $json = Get-Content $(System.DefaultWorkingDirectory)/db/temp/
↪tracingEnvironment/end-tracing.json | ConvertFrom-Json
            $json.PSObject.Properties | ForEach-Object {
                $template = "##vso[task.setvariable variable="
                $template += $_.Name
                $template += "]"
                $template += $_.Value
                echo "$template"
            }

   - task: CmdLine@2
      displayName: Finalize CodeQL database
      inputs:
         script: 'codeql database finalize db'

   # Other tasks go here, for example:
   # `codeql database analyze`
   # then `codeql github upload-results` ...
```

Obtaining databases from LGTM.com

LGTM.com analyzes thousands of open-source projects using CodeQL. For each project on LGTM.com, you can download an archived CodeQL database corresponding to the most recently analyzed revision of the code. These databases can also be analyzed using the CodeQL CLI or used with the CodeQL extension for Visual Studio Code.

To download a database from LGTM.com:

1. Log in to LGTM.com.
2. Find a project you're interested in and display the Integrations tab (for example, Apache Kafka).
3. Scroll to the **CodeQL databases for local analysis** section at the bottom of the page.
4. Download databases for the languages that you want to explore.

Before running an analysis, unzip the databases and try *upgrading* the unzipped databases to ensure they are compatible with your local copy of the CodeQL queries and libraries.

> Note
>
> The CodeQL CLI currently extracts data from additional, external files in a different way to the legacy QL tools. For example, when you run `codeql database create` the CodeQL CLI extracts data from some relevant XML files for Java and C#, but not for the other supported languages, such as JavaScript. This means that CodeQL databases created using the CodeQL CLI may be slightly different from those obtained from LGTM.com or created using the legacy QL command-line tools. As such, analysis results generated from databases created using the CodeQL CLI may also differ from those generated from databases obtained from elsewhere.

Further reading

- "*Analyzing your projects in CodeQL for VS Code*"

3.1.4 Extractor options

The CodeQL CLI uses special programs, called extractors, to extract information from the source code of a software system into a database that can be queried. You can customize the behavior of extractors by setting extractor configuration options through the CodeQL CLI.

About extractor options

Each extractor defines its own set of configuration options. To find out which options are available for a particular extractor, you can run `codeql resolve languages` or `codeql resolve extractor` with the `--format=betterjson` option. The `betterjson` output format provides the root paths of extractors and additional information. The output of `codeql resolve extractor --format=betterjson` will often be formatted like the following example:

```
{
    "extractor_root" : "/home/user/codeql/java",
    "extractor_options" : {
        "option1" : {
            "title" : "Java extractor option 1",
            "description" : "An example string option for the Java extractor.",
            "type" : "string",
            "pattern" : "[a-z]+"
        },
        "group1" : {
            "title" : "Java extractor group 1",
            "description" : "An example option group for the Java extractor.",
            "type" : "object",
            "properties" : {
                "option2" : {
                    "title" : "Java extractor option 2",
                    "description" : "An example array option for the Java extractor",
                    "type" : "array",
                    "pattern" : "[1-9][0-9]*"
                }
            }
        }
    }
}
```

The extractor option names and descriptions are listed under `extractor_options`. Each option may contain the following fields:

- `title` (required): The title of the option
- `description` (required): The description of the option
- `type` (required): The type of the option, which can be
  - `string`: indicating that the option can have a single string value
  - `array`: indicating that the option can have a sequence of string values

  - `object`: indicating that it is not an option itself, but a grouping that may contain other options and option groups

- `pattern` (optional): The regular expression patterns that all values of the option should match. Note that the extractor may impose additional constraints on option values that are not or cannot be expressed in this regular expression pattern. Such constraints, if they exist, would be explained under the description field.
- `properties` (optional): A map from extractor option names in the option group to the corresponding extractor option descriptions. This field can only be present for option groups. For example, options of `object` type.

In the example above, the extractor declares two options:

- `option1` is a `string` option with value matching `[a-z]+`
- `group1.option2` is an `array` option with values matching `[1-9][0-9]*`

Setting extractor options with the CodeQL CLI

The CodeQL CLI supports setting extractor options in subcommands that directly or indirectly invoke extractors. These commands are:

- `codeql database create`
- `codeql database start-tracing`
- `codeql database trace-command`
- `codeql database index-files`

When running these subcommands, you can set extractor options with the `--extractor-option` CLI option. For example:

- `codeql database create --extractor-option java.option1=abc ...`
- `codeql database start-tracing --extractor-option java.group1.option2=102 ...`

`--extractor-option` requires exactly one argument of the form `extractor_option_name=extractor_option_value`. `extractor_option_name` is the name of the extractor (in this example, `java`) followed by a period and then the name of the extractor option (in this example, either `option1` or `group1.option2`). `extractor_option_value` is the value being assigned to the extractor option. The value must match the regular expression pattern of the extractor option (if it exists), and it must not contain newline characters.

Using `--extractor-option` to assign an extractor option that does not exist is an error.

The CodeQL CLI accepts multiple `--extractor-option` options in the same invocation. If you set a `string` extractor option multiple times, the last option value overwrites all previous ones. If you set an *array* extractor option multiple times, all option values are concatenated in order.

You can also specify extractor option names without the extractor name. For example:

- `codeql database create --extractor-option option1=abc ...`
- `codeql database start-tracing --extractor-option group1.option2=102 ...`

If you do not specify an extractor name, the extractor option settings will apply to all extractors that declare an option with the given name. In the above example, the first command would set the extractor option `option1` to `abc` for the `java` extractor and every extractor that has an option of `option1`, for example the `cpp` extractor, if the `option1` extractor option exists for that extractor.

Setting extractor options from files

You can also set extractor options through a file. The CodeQL CLI subcommands that accept `--extractor-option` also accept `--extractor-options-file`, which has a required argument of the path to a YAML file (with extension `.yaml` or `.yml`) or a JSON file (with extension `.json`). For example:

- `codeql database create --extractor-options-file options.yml ...`
- `codeql database start-tracing --extractor-options-file options.json ...`

Each option file contains a tree structure of nested maps. At the root is an extractor map key, and beneath it are map keys that correspond to extractor names. Starting at the third level, there are extractor options and option groups.

In JSON:

```
{
     "extractor" : {
        "java": {
            "option1" : "abc",
            "group1" : {
                "option2" : [ 102 ]
            }
        }
    }
}
```

In YAML:

```
extractor:
    java:
        option1: "abc"
        group1:
            option2: [ 102 ]
```

The value for a `string` extractor option must be a string or a number (which will be converted to a string before further processing).

The value for an `array` extractor option must be an array of strings or numbers.

The value for an option group (of type `object`) must be a map, which may contain nested extractor options and option groups.

Each extractor option value must match the regular expression pattern of the extractor option (if it exists), and it must not contain newline characters.

Assigning an extractor option that does not exist is an error. You can make the CodeQL CLI ignore unknown extractor options by using a special `__allow_unknown_properties` Boolean field. For example, the following option file asks the CodeQL CLI to ignore all unknown extractor options and option groups under `group1`:

```
extractor:
    java:
        option1: "abc"
        group1:
            __allow_unknown_properties: true
            option2: [ 102 ]
```

You can specify `--extractor-options-file` multiple times. The extractor option assignments are processed in the following order:

1. All extractor option files specified by `--extractor-options-file` are processed in the order they appear on the command line, then
2. All extractor option assignments specified by `--extractor-option` are processed in the order they appear on the command line

The same rules govern what happens when the same extractor option is set multiple times, regardless of whether the assignments are done using `--extractor-option`, using `--extractor-options-file`, or some combination of the two. If you set a `string` extractor option multiple times, the last option value overwrites all previous values. If you set an `array` extractor option multiple times, all option values are concatenated in order.

3.1.5 Analyzing databases with the CodeQL CLI

To analyze a codebase, you run queries against a CodeQL database extracted from the code.

CodeQL analyses produce *interpreted results* that can be displayed as alerts or paths in source code. For information about writing queries to run with `database analyze`, see "*Using custom queries with the CodeQL CLI*."

> Other query-running commands
>
> Queries run with `database analyze` have strict *metadata requirements*. You can also execute queries using the following plumbing-level subcommands:
>
> - database run-queries, which outputs non-interpreted results in an intermediate binary format called *BQRS*.
> - query run, which will output BQRS files, or print results tables directly to the command line. Viewing results directly in the command line may be useful for iterative query development using the CLI.
>
> Queries run with these commands don't have the same metadata requirements. However, to save human-readable data you have to process each BQRS results file using the bqrs decode plumbing subcommand. Therefore, for most use cases it's easiest to use `database analyze` to directly generate interpreted results.

Before starting an analysis you must:

- *Set up the CodeQL CLI* so that it can find the queries and libraries included in the CodeQL repository.
- *Create a CodeQL database* for the source code you want to analyze.

Running `codeql database analyze`

When you run `database analyze`, it:

1. Optionally downloads any referenced CodeQL packages that are not available locally.
2. Executes one or more query files, by running them over a CodeQL database.
3. Interprets the results, based on certain query metadata, so that alerts can be displayed in the correct location in the source code.
4. Reports the results of any diagnostic and summary queries to standard output.

You can analyze a database by running the following command:

```
codeql database analyze <database> --format=<format> --output=<output> <queries>
```

You must specify:

- `<database>`: the path to the CodeQL database you want to analyze.

- `--format`: the format of the results file generated during analysis. A number of different formats are supported, including CSV, *SARIF*, and graph formats. For more information about CSV and SARIF, see *Results*. To find out which other results formats are supported, see the database analyze reference.
- `--output`: the output path of the results file generated during analysis.

You can also specify:

- `...<query-specifications>`: a list of queries to run over your database. This is a list of arguments. Where each argument can be:
  - a path to a query file
  - a path to a directory containing query files
  - a path to a query suite file
  - the name of a CodeQL query pack If omitted, the default query suite for the language of the database being analyzed will be used. For more information, see the *examples* below.
- `--sarif-category`: an identifying category for the results. Used when you want to upload more than one set of results for a commit. For example, when you use `github upload-results` to send results for more than one language to the GitHub code scanning API. For more information about this use case, see Configuring CodeQL CLI in your CI system in the GitHub documentation.
- `--sarif-add-query-help`: (supported in version 2.7.1 onwards) adds any custom query help written in markdown to SARIF files (v2.1.0 or later) generated by the analysis. Query help stored in `.qhelp` files must be converted to `.md` before running the analysis. For further information, see "*Including query help for custom CodeQL queries in SARIF files*."
- `--download`: a boolean flag that will allow the CLI to download any referenced CodeQL packages that are not available locally. If this flag is missing and a referenced CodeQL package is not available locally, the command will fail.
- `--threads`: optionally, the number of threads to use when running queries. The default option is `1`. You can specify more threads to speed up query execution. Specifying `0` matches the number of threads to the number of logical processors.

  Upgrading databases

  If the CodeQL queries you want to use are newer than the extractor used to create the database, then you may see a message telling you that your database needs to be upgraded when you run `database analyze`. You can quickly upgrade a database by running the `database upgrade` command. For more information, see "*Upgrading CodeQL databases*."

For full details of all the options you can use when analyzing databases, see the database analyze reference documentation.

Examples

The following examples assume your CodeQL databases have been created in a directory that is a sibling of your local copies of the CodeQL and CodeQL for Go repositories.

Running a single query

To run a single query over a CodeQL database for a JavaScript codebase, you could use the following command from the directory containing your database:

```
codeql database analyze <javascript-database> ../ql/javascript/ql/src/Declarations/
↪UnusedVariable.ql --format=csv --output=js-analysis/js-results.csv
```

This command runs a simple query that finds potential bugs related to unused variables, imports, functions, or classes—it is one of the JavaScript queries included in the CodeQL repository. You could run more than one query by specifying a space-separated list of similar paths.

The analysis generates a CSV file (`js-results.csv`) in a new directory (`js-analysis`).

You can also run your own custom queries with the `database analyze` command. For more information about preparing your queries to use with the CodeQL CLI, see "*Using custom queries with the CodeQL CLI*."

Running a CodeQL pack

> Note
>
> The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

To run an existing CodeQL query pack from the GitHub Container registry, you can specify one or more pack names and use the `--download` flag:

```
codeql database analyze <database> microsoft/coding-standards@1.0.0 github/security-
↪queries --format=sarifv2.1.0 --output=query-results.sarif --download
```

The `analyze` command above runs the default suite from `microsoft/coding-standards v1.0.0` and the latest version of `github/secutiry-queries` on the specified database. For further information about default suites, see "*Publishing and using CodeQL packs*".

For more information about CodeQL packs, see *About CodeQL Packs*.

Running query suites

To run a query suite over a CodeQL database for a C/C++ codebase, you could use the following command from the directory containing your database:

```
codeql database analyze <cpp-database> cpp-code-scanning.qls --format=sarifv2.1.0 --
↪output=cpp-results.sarif
```

The analysis generates a file in the v2.1.0 SARIF format that is supported by all versions of GitHub. This file can be uploaded to GitHub by executing `codeql github upload-results` or the code scanning API. For more information, see Analyzing a CodeQL database or Code scanning API in the GitHub documentation.

CodeQL query suites are `.qls` files that use directives to select queries to run based on certain metadata properties. The standard QL packs have metadata that specify the location of the query suites used by code scanning, so the CodeQL CLI knows where to find these suite files automatically, and you don't have to specify the full path on the command line. For more information, see "*About QL packs*."

The standard query suites are stored at the following paths in the CodeQL repository:

```
ql/<language>/ql/src/codeql-suites/<language>-code-scanning.qls
```

and at the following path in the CodeQL for Go repository:

```
ql/src/codeql-suites/go-code-scanning.qls
```

The repository also includes the query suites used by LGTM.com. These are stored alongside the query suites for code scanning with names of the form: `<language>-lgtm.qls`.

For information about creating custom query suites, see "*Creating CodeQL query suites*."

Diagnostic and summary information

When you create a CodeQL database, the extractor stores diagnostic data in the database. The code scanning query suites include additional queries to report on this diagnostic data and calculate summary metrics. When the `database analyze` command completes, the CLI generates the results file and reports any diagnostic and summary data to standard output. If you choose to generate SARIF output, the additional data is also included in the SARIF file.

If the analysis found fewer results for standard queries than you expected, review the results of the diagnostic and summary queries to check whether the CodeQL database is likely to be a good representation of the codebase that you want to analyze.

Integrating a CodeQL pack into a code scanning workflow in GitHub

> Note
>
> The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

You can use CodeQL query packs in your code scanning setup. This allows you to select query packs published by various sources and use them to analyze your code. For more information, see "Using CodeQL query packs in the CodeQL action" or "Downloading and using CodeQL query packs in your CI system."

Running all queries in a directory

You can run all the queries located in a directory by providing the directory path, rather than listing all the individual query files. Paths are searched recursively, so any queries contained in subfolders will also be executed.

> Important
>
> You shouldn't specify the root of a *QL pack* when executing `database analyze` as it contains some special queries that aren't designed to be used with the command. Rather, to run a wide range of useful queries, run one of the LGTM.com query suites.

For example, to execute all Python queries contained in the `Functions` directory you would run:

```
codeql database analyze <python-database> ../ql/python/ql/src/Functions/ --format=sarif-
↪latest --output=python-analysis/python-results.sarif
```

A SARIF results file is generated. Specifying `--format=sarif-latest` ensures that the results are formatted according to the most recent SARIF specification supported by CodeQL.

Including query help for custom CodeQL queries in SARIF files

If you use the CodeQL CLI to to run code scanning analyses on third party CI/CD systems, you can include the query help for your custom queries in SARIF files generated during an analysis. After uploading the SARIF file to GitHub, the query help is shown in the code scanning UI for any alerts generated by the custom queries.

From CodeQL CLI 2.7.1 onwards, you can include markdown-rendered query help in SARIF files by providing the `--sarif-add-query-help` option when running `codeql database analyze`. For more information, see Configuring CodeQL CLI in your CI system in the GitHub documentation.

You can write query help for custom queries directly in a markdown file and save it alongside the corresponding query. Alternatively, for consistency with the standard CodeQL queries, you can write query help in the `.qhelp` format. Query help written in `.qhelp` files can't be included in SARIF files, and they can't be processed by code scanning so must be converted to markdown before running the analysis. For more information, see "*Query help files*" and "*Testing query help files*."

Results

You can save analysis results in a number of different formats, including SARIF and CSV.

The SARIF format is designed to represent the output of a broad range of static analysis tools. For more information, see *SARIF output*.

If you choose to generate results in CSV format, then each line in the output file corresponds to an alert. Each line is a comma-separated list with the following information:

Property	Description	Example
Name	Name of the query that identified the result.	`Inefficient regular expression`
Description	Description of the query.	`A regular expression that requires exponential time to match certain inputs can be a performance bottleneck, and may be vulnerable to denial-of-service attacks.`
Severity	Severity of the query.	`error`
Message	Alert message.	`This part of the regular expression may cause exponential backtracking on strings containing many repetitions of '\\\\'.`
Path	Path of the file containing the alert.	`/vendor/codemirror/markdown.js`
Start line	Line of the file where the code that triggered the alert begins.	`617`
Start column	Column of the start line that marks the start of the alert code. Not included when equal to 1.	`32`
End line	Line of the file where the code that triggered the alert ends. Not included when the same value as the start line.	`64`
End column	Where available, the column of the end line that marks the end of the alert code. Otherwise the end line is repeated.	`617`

Results files can be integrated into your own code-review or debugging infrastructure. For example, SARIF file output can be used to highlight alerts in the correct location in your source code using a SARIF viewer plugin for your IDE.

Further reading

- "*Analyzing your projects in CodeQL for VS Code*"

3.1.6 Upgrading CodeQL databases

As the CodeQL CLI tools and queries evolve, you may find that some of your CodeQL databases become out of date. You must upgrade out-of-date databases before you can analyze them.

Databases become out of date when:

- For databases created using the CodeQL CLI, the version of CLI tools used to create them is older than your copy of the CodeQL queries.
- For databases downloaded from LGTM.com, the CodeQL tools used by LGTM.com to create that revision of the code are older than your copy of the CodeQL queries.

If you have a local checkout of the `github/codeql` repository, please note that the `main` branch of the CodeQL queries is updated more often than both the CLI and LGTM.com, so databases are most likely to become out of date if you use the queries on this branch. For more information about the different versions of the CodeQL queries, see "*Getting started with the CodeQL CLI*."

Out-of-date databases must be upgraded before they can be analyzed. This topic shows you how to upgrade a CodeQL database using the `database upgrade` subcommand.

Prerequisites

Archived databases downloaded from LGTM.com must be unzipped before they are upgraded.

Running `codeql database upgrade`

CodeQL databases are upgraded by running the following command:

```
codeql database upgrade <database>
```

where <database>, the path to the CodeQL database you want to upgrade, must be specified.

For full details of all the options you can use when upgrading databases, see the "database upgrade" reference documentation.

Progress and results

When you execute the `database upgrade` command, CodeQL identifies the version of the *schema* associated with the database. From there, it works out what (if anything) is required to make the database work with your queries and libraries. It will rewrite the database, if necessary, or make no changes if the database is already compatible (or if it finds no information about how to perform an upgrade). Once a database has been upgraded it cannot be downgraded for use with older versions of the CodeQL products.

3.1.7 Using custom queries with the CodeQL CLI

You can customize your CodeQL analyses by writing your own queries to highlight specific vulnerabilities or errors.

This topic is specifically about writing queries to use with the database analyze command to produce *interpreted results*.

> Other query-running commands
>
> Queries run with `database analyze` have strict *metadata requirements*. You can also execute queries using the following plumbing-level subcommands:
>
> - database run-queries, which outputs non-interpreted results in an intermediate binary format called *BQRS*.
> - query run, which will output BQRS files, or print results tables directly to the command line. Viewing results directly in the command line may be useful for iterative query development using the CLI.
>
> Queries run with these commands don't have the same metadata requirements. However, to save human-readable data you have to process each BQRS results file using the bqrs decode plumbing subcommand. Therefore, for most use cases it's easiest to use `database analyze` to directly generate interpreted results.

Writing a valid query

Before running a custom analysis you need to write a valid query, and save it in a file with a `.ql` extension. There is extensive documentation available to help you write queries. For more information, see "*CodeQL queries*."

Including query metadata

Query metadata is included at the top of each query file. It provides users with information about the query, and tells the CodeQL CLI how to process the query results.

When running queries with the `database analyze` command, you must include the following two properties to ensure that the results are interpreted correctly:

- Query identifier (`@id`): a sequence of words composed of lowercase letters or digits, delimited by `/` or `-`, identifying and classifying the query.
- Query type (`@kind`): identifies the query as a simple alert (`@kind problem`), an alert documented by a sequence of code locations (`@kind path-problem`), for extractor troubleshooting (`@kind diagnostic`), or a summary metric (`@kind metric` and `@tags summary`).

For more information about these metadata properties, see "*Metadata for CodeQL queries*" and the Query metadata style guide.

> Note
>
> Metadata requirements may differ if you want to use your query with other applications. For more information, see "*Metadata for CodeQL queries* ."

Packaging custom QL queries

> Note
>
> The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

When you write your own queries, you should save them in a custom QL pack directory. When you are ready to share your queries with other users, you can publish the pack as a CodeQL pack to GitHub Packages - the GitHub Container registry.

QL packs organize the files used in CodeQL analysis and can store queries, library files, query suites, and important metadata. Their root directory must contain a file named `qlpack.yml`. Your custom queries should be saved in the QL pack root, or its subdirectories.

For each QL pack, the `qlpack.yml` file includes information that tells CodeQL how to compile the queries, which other CodeQL packs and libraries the pack depends on, and where to find query suite definitions. For more information about what to include in this file, see "*About QL packs*."

CodeQL packages are used to create, share, depend on, and run CodeQL queries and libraries. You can publish your own CodeQL packages and download ones created by others via the the Container registry. For further information see "*About CodeQL packs*."

Contributing to the CodeQL repository

If you would like to share your query with other CodeQL users, you can open a pull request in the CodeQL repository. For further information, see Contributing to CodeQL.

Further reading

- "*CodeQL queries*"

3.1.8 Creating CodeQL query suites

CodeQL query suites provide a way of selecting queries, based on their filename, location on disk or in a QL pack, or metadata properties. Create query suites for the queries that you want to frequently use in your CodeQL analyses.

Query suites allow you to pass multiple queries to CodeQL without having to specify the path to each query file individually. Query suite definitions are stored in YAML files with the extension `.qls`. A suite definition is a sequence of instructions, where each instruction is a YAML mapping with (usually) a single key. The instructions are executed in the order they appear in the query suite definition. After all the instructions in the suite definition have been executed, the result is a set of selected queries.

> Note
>
> Any custom queries that you want to add to a query suite must be in a *QL pack* and contain the correct query metadata. For more information, see "*Using custom queries with the CodeQL CLI*."

Locating queries to add to a query suite

When creating a query suite, you first need to specify the locations of the queries that you want to select. You can define the location of one or more queries using:

- A `query` instruction—tells CodeQL to look for one or more specified `.ql` files:

```
- query: <path-to-query>
```

  The argument must be one or more file paths, relative to the QL pack containing the suite definition.

- A `queries` instruction—tells CodeQL to recursively scan a directory for `.ql` files:

```
- queries: <path-to-subdirectory>
```

  The path of the directory must be relative to the root of the QL pack that contains the suite definition file. To find the queries relative to a different QL pack, add a `from` field:

```
- queries: <path-to-subdirectory>
  from: <ql-pack-name>
```

- A `qlpack` instruction—tells CodeQL to resolve queries in the default suite of the named QL pack:

```
- qlpack: <qlpack-name>
```

  The default suite of a query pack includes a recommended set of queries inside of that query pack. Not all query packs have a default suite. If the given query pack does not define a default suite, the *qlpack* instruction will resolve to all of the queries within the pack.

  Note

  When pathnames appear in query suite definitions, they must always be given with a forward slash, `/`, as a directory separator. This ensures that query suite definitions work on all operating systems.

You must add at least one `query`, `queries`, or `qlpack` instruction to your suite definition, otherwise no queries will be selected. If the suite contains no further instructions, all the queries found from the list of files, in the given directory, or in the named QL pack are selected. If there are further filtering instructions, only queries that match the constraints imposed by those instructions will be selected.

Filtering the queries in a query suite

After you have defined the initial set of queries to add to your suite by specifying `query`, `queries`, or `qlpack` instructions, you can add `include` and `exclude` instructions. These instructions define selection criteria based on specific properties:

- When you execute an `include` instruction on a set of queries, any queries that match your conditions are retained in the selection, and queries that don't match are removed.
- When you execute an `exclude` instructions on a set of queries, any queries that match your conditions are removed from the selection, and queries that don't match are retained.

The order of your filter instructions is important. The first filter instruction that appears after the locating instructions determines whether the queries are included or excluded by default. If the first filter is an `include`, the initially located queries will only be part of the suite if they match an explicit `include` filter. If the first filter is an `exclude`, the initially located queries are part of the suite unless they are explicitly excluded.

Subsequent instructions are executed in order and the instructions that appear later in the file take precedence over the earlier instructions. So, `include` instructions can be overridden by a later `exclude` instructions that match the same query. Similarly, `exclude`s can be overridden by a later `include`.

For both instructions, the argument is a constraint block—that is, a YAML map representing the constraints. Each constraint is a map entry, where the key is typically a query metadata property. The value can be:

- A single string.
- A /-enclosed regular expression.
- A list containing strings, regular expressions, or both.

To match a constraint, a metadata value must match one of the strings or regular expressions. When there is more than one metadata key, each key must be matched. For more information about query metadata properties, see "*Metadata for CodeQL queries*."

In addition to metadata tags, the keys in the constraint block can also be:

- `query filename`—matches on the last path component of the query file name.
- `query path`—matches on the path to the query file relative to its enclosing QL pack.
- `tags contain`—one of the given match strings must match one of the space-separated components of the value of the `@tags` metadata property.
- `tags contain all`—each of the given match strings must match one of the components of the `@tags` metadata property.

Examples

To define a suite that selects all queries in the default suite of the `codeql/cpp-queries` QL pack, and then refines them to only include security queries, use:

```
- qlpack: codeql/cpp-queries
- include:
    tags contain: security
```

To define a suite that selects all queries with `@kind problem` and `@precision high` from the `my-custom-queries` directory, use:

```
- queries: my-custom-queries
- include:
    kind: problem
    precision: very-high
```

To create a suite that selects all queries with `@kind problem` from the `my-custom-queries` directory except those with `@problem.severity recommendation`, use:

```
- queries: my-custom-queries
- include:
    kind: problem
- exclude:
    problem.severity: recommendation
```

To create a suite that selects all queries with `@tag security` and `@problem.severity high` or `very-high` from the `codeql/cpp-queries` QL pack, use:

```
- queries: .
  from: codeql/cpp-queries
- include:
```

(continues on next page)

(continued from previous page)

```
    tags contain: security
    problem.severity:
    - high
    - very-high
```

Reusing existing query suite definitions

Existing query suite definitions can be reused by specifying:

- An `import` instruction—adds the queries selected by a previously defined `.qls` file to the current suite:

  ```
  - import: <path-to-query-suite>
  ```

  The path to the imported suite must be relative to the QL pack containing the current suite definition. If the imported query suite is in a different QL pack you can use:

  ```
  - import: <path-to-query-suite>
    from: <ql-pack>
  ```

  Queries added using an `import` instruction can be filtered using subsequent `exclude` instructions.

- An `apply` instruction—adds all of the instructions from a previously defined `.qls` file to the current suite. The instructions in the applied `.qls` file are executed as if they appear in place of `apply`. Any `include` and `exclude` instructions from the applied suite also act on queries added by any earlier instructions:

  ```
  - apply: <path-to-query-suite>
  ```

  The `apply` instruction can also be used to apply a set of reusable conditions, saved in a `.yml` file, to multiple query definitions. For more information, see the *example* below.

- An `eval` instruction—performs the same function as an `import` instruction, but takes a full suite definition as the argument, rather than the path to a `.qls` file on disk.

Example

To use the same conditions in multiple query suite definitions, create a separate `.yml` file containing your instructions. For example, save the following in a file called `reusable-instructions.yml`:

```
- include:
    kind:
    - problem
    - path-problem
    tags contain: security
    precision:
    - high
    - very-high
```

Add `reusable-instructions.yml` to the same QL pack as your current query suite (for example, `my-custom-queries`). Apply the reusable instructions to the queries in your current suite using:

```
- qlpack: my-custom-queries
- apply: reusable-instructions.yml
```

To apply the same conditions to a different suite or directory within the same QL pack, create a new definition and change (or replace) the `qlpack` instruction. For example:

```
- queries: queries/cpp/custom
- apply: reusable-instructions.yml
```

You can also create a suite definition using `reusable-instructions.yml` on queries in a different QL pack. If the `.qls` file is in the same QL pack as the queries, you can add a `from` field immediately after the `apply` instruction:

```
- qlpack: my-other-custom-queries
- apply: reusable-instructions.yml
  from: <name-of-ql-pack>
```

Naming a query suite

You can provide a name for your query suite by specifying a `description` instruction:

```
- description: <name-of-query-suite>
```

This value is displayed when you run codeql resolve queries, if the suite is added to a "well-known" directory. For more information, see "*Specifying well-known query suites*."

Saving a query suite

Save your query suite in a file with a `.qls` extension and add it to a QL pack. For more information, see "*About QL packs*."

Specifying well-known query suites

You can use QL packs to declare directories that contain "well-known" query suites. You can use "well-known" query suites on the command line by referring to their file name, without providing their full path. This gives you a simple way of specifying a set of queries, without needing to search inside QL packs and distributions. To declare a directory that contains "well-known" query suites, add the directory to the `suites` property in the `qlpack.yml` file at the root of your QL pack. For more information, see "*About QL packs*."

Using query suites with CodeQL

You can specify query suites on the command line for any command that accepts `.qls` files. For example, you can compile the queries selected by a suite definition using `query compile`, or use the queries in an analysis using `database analyze`. For more information about analyzing CodeQL databases, see "*Analyzing databases with the CodeQL CLI*."

Viewing the query suites used on LGTM.com

The query suite definitions used to select queries to run on LGTM.com can be found in the CodeQL repository. For example, to view the CodeQL queries for JavaScript, visit https://github.com/github/codeql/tree/main/javascript/ql/src/codeql-suites.

These suite definitions apply reusable filter patterns to the queries located in the standard QL packs for each supported language. For more information, see the suite-helpers in the CodeQL repository.

Further reading

- "*CodeQL queries*"

3.1.9 Testing custom queries

CodeQL provides a simple test framework for automated regression testing of queries. Test your queries to ensure that they behave as expected.

During a query test, CodeQL compares the results the user expects the query to produce with those actually produced. If the expected and actual results differ, the query test fails. To fix the test, you should iterate on the query and the expected results until the actual results and the expected results exactly match. This topic shows you how to create test files and execute tests on them using the `test run` subcommand.

Setting up a test QL pack for custom queries

All CodeQL tests must be stored in a special "test" QL pack. That is, a directory for test files with a `qlpack.yml` file that defines:

```
name: <name-of-test-pack>
version: 0.0.0
libraryPathDependencies: <codeql-libraries-and-queries-to-test>
extractor: <language-of-code-to-test>
```

The `libraryPathDependencies` value specifies the CodeQL queries to test. The `extractor` defines which language the CLI will use to create test databases from the code files stored in this QL pack. For more information, see "*About QL packs*."

You may find it useful to look at the way query tests are organized in the CodeQL repository. Each language has a `src` directory, `ql/<language>/ql/src`, that contains libraries and queries for analyzing codebases. Alongside the `src` directory, there's a `test` directory with tests for these libraries and queries.

Each `test` directory is configured as a test QL pack with two subdirectories:

- `query-tests` a series of subdirectories with tests for queries stored in the `src` directory. Each subdirectory contains test code and a QL reference file that specifies the query to test.
- `library-tests` a series of subdirectories with tests for QL library files. Each subdirectory contains test code and queries that were written as unit tests for a library.

Setting up the test files for a query

For each query you want to test, you should create a sub-directory in the test QL pack. Then add the following files to the subdirectory before you run the test command:

- A query reference file (`.qlref` file) defining the location of the query to test. The location is defined relative to the root of the QL pack that contains the query. Usually, this is a QL pack specified by the `libraryPathDependencies` for the test pack. For more information, see "*Query reference files*."

  You don't need to add a query reference file if the query you want to test is stored in the test directory, but it's generally good practice to store queries separately from tests. The only exception is unit tests for QL libraries, which tend to be stored in test packs, separate from queries that generate alerts or paths.

- The example code you want to run your query against. This should consist of one or more files containing examples of the code the query is designed to identify.

You can also define the results you expect to see when you run the query against the example code, by creating a file with the extension `.expected`. Alternatively, you can leave the test command to create the `.expected` file for you. (If you're using CodeQL CLI 2.0.2–2.0.6, you need to create an `.expected` file otherwise the command will fail to find your test query. You can create an empty `.expected` file to workaround this limitation.)

For an example showing how to create and test a query, see the *example* below.

> Important
>
> Your `.ql`, `.qlref`, and `.expected` files must have consistent names.
>
> If you want to directly specify the `.ql` file itself in the test command, it must have the same base name as the corresponding `.expected` file. For example, if the query is `MyJavaQuery.ql`, the expected results file must be `MyJavaQuery.expected`.
>
> If you want to specify a `.qlref` file in the command, it must have the same base name as the corresponding `.expected` file, but the query itself may have a different name.
>
> The names of the example code files don't have to be consistent with the other test files. All example code files found next to the `.qlref` (or `.ql`) file and in any subdirectories will be used to create a test database. Therefore, for simplicity, we recommend you don't save test files in directories that are ancestors of each other.

Running `codeql test run`

CodeQL query tests are executed by running the following command:

```
codeql test run <test|dir>
```

The `<test|dir>` argument can be one or more of the following:

- Path to a `.ql` file.
- Path to a `.qlref` file that references a `.ql` file.
- Path to a directory that will be searched recursively for `.ql` and `.qlref` files.

You can also specify:

- `--threads`: optionally, the number of threads to use when running queries. The default option is `1`. You can specify more threads to speed up query execution. Specifying `0` matches the number of threads to the number of logical processors.

For full details of all the options you can use when testing queries, see the test run reference documentation.

Example

The following example shows you how to set up a test for a query that searches Java code for `if` statements that have empty `then` blocks. It includes steps to add the custom query and corresponding test files to separate QL packs outside your checkout of the CodeQL repository. This ensures when you update the CodeQL libraries, or check out a different branch, you won't overwrite your custom queries and tests.

Prepare a query and test files

1. Develop the query. For example, the following simple query finds empty `then` blocks in Java code:

```
import java

from IfStmt ifstmt
where ifstmt.getThen() instanceof EmptyStmt
select ifstmt, "This if statement has an empty then."
```

2. Save the query to a file named `EmptyThen.ql` in a directory with your other custom queries. For example, `custom-queries/java/queries/EmptyThen.ql`.

3. If you haven't already added your custom queries to a QL pack, create a QL pack now. For example, if your custom Java queries are stored in `custom-queries/java/queries`, add a `qlpack.yml` file with the following contents to `custom-queries/java/queries`:

```
name: my-custom-queries
version: 0.0.0
libraryPathDependencies: codeql-java
```

   For more information about QL packs, see "*About QL packs*."

4. Create a QL pack for your Java tests by adding a `qlpack.yml` file with the following contents to `custom-queries/java/tests`, updating `libraryPathDependencies` to match the name of your QL pack of custom queries:

```
name: my-query-tests
version: 0.0.0
libraryPathDependencies: my-custom-queries
extractor: java
tests: .
```

   This `qlpack.yml` file states that `my-query-tests` depends on `my-custom-queries`. It also declares that the CLI should use the Java `extractor` when creating test databases. Supported from CLI 2.1.0 onward, the `tests: .` line declares that all `.ql` files in the pack should be run as tests when `codeql test run` is run with the `--strict-test-discovery` option.

5. Within the Java test pack, create a directory to contain the test files associated with `EmptyThen.ql`. For example, `custom-queries/java/tests/EmptyThen`.

6. In the new directory, create `EmptyThen.qlref` to define the location of `EmptyThen.ql`. The path to the query must be specified relative to the root of the QL pack that contains the query. In this case, the query is in the top level directory of the QL pack named `my-custom-queries`, which is declared as a dependency for `my-query-tests`. Therefore, `EmptyThen.qlref` should simply contain `EmptyThen.ql`.

7. Create a code snippet to test. The following Java code contains an empty `if` statement on the third line. Save it in `custom-queries/java/tests/EmptyThen/Test.java`.

```
class Test {
  public void problem(String arg) {
    if (arg.isEmpty())
    ;
    {
        System.out.println("Empty argument");
    }
```

(continues on next page)

(continued from previous page)

```
  }

  public void good(String arg) {
    if (arg.isEmpty()) {
        System.out.println("Empty argument");
    }
  }
}
```

Execute the test

To execute the test, move into the `custom-queries` directory and run `codeql test run java/tests/EmptyThen`.

When the test runs it:

1. Finds one test in the `EmptyThen` directory.
2. Extracts a CodeQL database from the `.java` files stored in the `EmptyThen` directory.
3. Compiles the query referenced by the `EmptyThen.qlref` file.

   If this step fails, it's because the CLI can't find your custom QL pack. Re-run the command and specify the location of your custom QL pack, for example:

   ```
   codeql test run --search-path=java java/tests/EmptyThen
   ```

   For information about saving the search path as part of your configuration, see "*Specifying command options in a CodeQL configuration file*."
4. Executes the test by running the query and generating an `EmptyThen.actual` results file.
5. Checks for an `EmptyThen.expected` file to compare with the `.actual` results file.
6. Reports the results of the test — in this case, a failure: `0 tests passed; 1 tests failed:`. The test failed because we haven't yet added a file with the expected results of the query.

View the query test output

CodeQL generates the following files in the `EmptyThen` directory:

- `EmptyThen.actual`, a file that contains the actual results generated by the query.
- `EmptyThen.testproj`, a test database that you can load into VS Code and use to debug failing tests. When tests complete successfully, this database is deleted in a housekeeping step. You can override this step by running `test run` with the `--keep-databases` option.

In this case, the failure was expected and is easy to fix. If you open the `EmptyThen.actual` file, you can see the results of the test:

```
| Test.java:3:5:3:22 | stmt | This if statement has an empty then. |
```

This file contains a table, with a column for the location of the result, along with separate columns for each part of the `select` clause the query outputs. Since the results are what we expected, we can update the file extension to define this as the expected result for this test (`EmptyThen.expected`).

If you rerun the test now, the output will be similar but it will finish by reporting: `All 1 tests passed.`.

If the results of the query change, for example, if you revise the `select` statement for the query, the test will fail. For failed results, the CLI output includes a unified diff of the `EmptyThen.expected` and `EmptyThen.actual` files. This information may be sufficient to debug trivial test failures.

For failures that are harder to debug, you can import `EmptyThen.testproj` into CodeQL for VS Code, execute `EmptyThen.ql`, and view the results in the `Test.java` example code. For more information, see "*Analyzing your projects*" in the CodeQL for VS Code help.

Further reading

- "*CodeQL queries*"
- "*Testing CodeQL queries in Visual Studio Code*"

3.1.10 Testing query help files

Test query help files by rendering them as markdown to ensure they are valid before uploading them to the CodeQL repository or using them in code scanning.

Query help is documentation that accompanies a query to explain how the query works, as well as providing information about the potential problem that the query identifies. It is good practice to write query help for all new queries. For more information, see Contributing to CodeQL in the CodeQL repository.

The CodeQL CLI includes a command to test query help and render the content as markdown, so that you can easily preview the content in your IDE. Use the command to validate query help files before uploading them to the CodeQL repository or sharing them with other users. From CodeQL CLI 2.7.1 onwards, you can also include the markdown-rendered query help in SARIF files generated during CodeQL analyses so that the query help can be displayed in the code scanning UI. For more information, see "*Analyzing databases with the CodeQL CLI*."

Prerequisites

- The query help (`.qhelp`) file must have an accompanying query (`.ql`) file with an identical base name.
- The query help file should follow the standard structure and style for query help documentation. For more information, see the Query help style guide in the CodeQL repository.

Running `codeql generate query-help`

You can test query help files by running the following command:

```
codeql generate query-help <qhelp|query|dir|suite> --format=<format> [--output=<dir|file>
↪]
```

where <qhelp|query|dir|suite> is one of:

- the path to a `.qhelp` file.
- the path to a `.ql` file.
- the path to a directory containing queries and query help files.
- the path to a query suite, or the name of a well-known query suite for a QL pack. For more information, see "Creating CodeQL query suites."

You must specify a `--format` option, which defines how the query help is rendered. Currently, you must specify `markdown` to render the query help as markdown.

The `--output` option defines a file path where the rendered query help will be saved.

- For directories containing `.qhelp` files or a query suites defining one or more `.qhelp` files, you must specify an `--output` directory. Filenames within the output directory will be derived from the `.qhelp` file names.
- For single `.qhelp` or `.ql` files, you may specify an `--output` option. If you don't specify an output path, the rendered query help is written to `stdout`.

For full details of all the options you can use when testing query help files, see the generate query-help reference documentation.

Results

When you run the command, CodeQL attempts to render each `.qhelp` file that has an accompanying `.ql` file. For single files, the rendered content will be printed to `stdout` if you don't specify an `--output` option. For all other use cases, the rendered content is saved to the specified output path.

By default, the CodeQL CLI will print a warning message if:

- Any of the query help is invalid, along with a description of the invalid query help elements
- Any `.qhelp` files specified in the command don't have the same base name as an accompanying `.ql` file
- Any `.ql` files specified in the command don't have the same base name as an accompanying `.qhelp` file

You can tell the CodeQL CLI how to handle these warnings by including a `--warnings` option in your command. For more information, see the generate query-help reference documentation.

Further reading

- "*Query help files*"

3.1.11 Creating and working with CodeQL packs

You can use CodeQL packs to create, share, depend on, and run CodeQL queries and libraries.

> Note
>
> The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

About CodeQL packs and the CodeQL CLI

With CodeQL packs and the package management commands in the CodeQL CLI, you can publish your custom queries and integrate them into your codebase analysis.

There are two types of CodeQL packs: query packs and library packs.

- Query packs are designed to be run. When a query pack is published, the bundle includes all the transitive dependencies and a compilation cache. This ensures consistent and efficient execution of the queries in the pack.
- Library packs are designed to be used by query packs (or other library packs) and do not contain queries themselves. The libraries are not compiled and there is no compilation cache included when the pack is published.

You can use the `pack` command in the CodeQL CLI to create CodeQL packs, add dependencies to packs, and install or update dependencies. You can also publish and download CodeQL packs using the `pack` command. For more information, see "*Publishing and using CodeQL packs*."

Creating a CodeQL pack

You can create a CodeQL pack by running the following command from the checkout root of your project:

```
codeql pack init <scope>/<pack>
```

You must specify:

- `<scope>`: the name of the GitHub organization or user account that you will publish to.
- `<pack>`: the name for the pack that you are creating.

The `codeql pack init` command creates the directory structure and configuration files for a CodeQL pack. By default, the command creates a query pack. If you want to create a library pack, you must edit the `qlpack.yml` file to explicitly declare the file as a library pack by including the `library:true` property.

Modifying an existing QL pack to create a CodeQL pack

If you already have a `qlpack.yml` file, you can edit it manually to convert it into a CodeQL pack.

1. Edit the `name` property so that it matches the format `<scope>/<name>`, where `<scope>` is the name of the GitHub organization or user account that you will publish to.
2. In the `qlpack.yml` file, include a `version` property with a semver identifier, as well as an optional `dependencies` block.

For more information about the properties, see "*About CodeQL packs*."

Adding and installing dependencies to a CodeQL pack

You can add dependencies on CodeQL packs using the command `codeql pack add`. You must specify the scope, name, and version range.

```
codeql pack add <scope>/<name>@x.x.x <scope>/<other-name>
```

The version range is optional. If you leave off the version range, the latest version will be added. Otherwise, the latest version that satisfies the requested range will be added.

This command updates the `qlpack.yml` file with the requested dependencies and downloads them into the package cache. Please note that this command will reformat the file and remove all comments.

You can also manually edit the `qlpack.yml` file to include dependencies and install the dependencies with the command:

```
codeql pack install
```

This command downloads all dependencies to the shared cache on the local disk.

> Note
>
> Running the `codeql pack add` and `codeql pack install` commands will generate or update the `qlpack.lock.yml` file. This file should be checked-in to version control. The `qlpack.lock.yml` file contains the precise version numbers used by the pack.

3.1.12 Publishing and using CodeQL packs

You can publish your own CodeQL packs and use packs published by other people.

> Note
>
> The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

Configuring the `qlpack.yml` file before publishing

You can check and modify the configuration details of your CodeQL pack prior to publishing. Open the `qlpack.yml` file in your preferred text editor.

```
library: # set to true if the pack is a library. Set to false or omit for a query pack
name: <scope>/<pack>
version: <x.x.x>
description: <Description to publish with the package>
default-suite: # optional, one or more queries in the pack to run by default
    - query: <relative-path>/query-file>.ql
default-suite-file: default-queries.qls # optional, a pointer to a query-suite in this␣
↪pack
license: # optional, the license under which the pack is published
dependencies: # map from CodeQL pack name to version range
```

- `name:` must follow the <scope>/<pack> format, where <scope> is the GitHub organization that you will publish to and <pack> is the name for the pack.
- A maximum of one of `default-suite` or `default-suite-file` is allowed. These are two different ways to define a default query suite to be run, the first by specifying queries directly in the *qlpack.yml* file and the second by specifying a query suite in the pack.

Running `codeql pack publish`

When you are ready to publish a pack to the GitHub Container registry, you can run the following command in the root of the pack directory:

```
codeql pack publish
```

The published package will be displayed in the packages section of GitHub organization specified by the scope in the `qlpack.yml` file.

Running `codeql pack download <scope>/<pack>`

To run a pack that someone else has created, you must first download it by running the following command:

```
codeql pack download <scope>/<pack>@x.x.x
```

- <scope>: the name of the GitHub organization that you will download from.
- <pack>: the name for the pack that you want to download.
- `@x.x.x`: an optional version number. If omitted, the latest version will be downloaded.

This command accepts arguments for multiple packs.

Using a CodeQL pack to analyze a CodeQL database

To analyze a CodeQL database with a CodeQL pack, run the following command:

```
codeql database analyze <database> <scope>/<pack>@x.x.x
```

- <database>: the CodeQL database to be analyzed.
- <scope>: the name of the GitHub organization that the pack is published to.
- <pack>: the name for the pack that you are using.
- @x.x.x: an optional version number. If omitted, the latest version will be used.

The analyze command will run the default suite of any specified CodeQL packs. You can specify multiple CodeQL packs to be used for analyzing a CodeQL database. For example:

```
codeql <database> analyze <scope>/<pack> <scope>/<other-pack>
```

3.1.13 Specifying command options in a CodeQL configuration file

You can save default or frequently used options for your commands in a per-user configuration file.

You can specify CodeQL CLI command options in two ways:

- Directly in the command line, using the appropriate flag.
- In a configuration (or config) file that CodeQL scans for relevant options each time a command is executed.

For options that are likely to change each time you execute a command, specifying the value on the command line is the most convenient way of passing the information to CodeQL. Saving options in a config file is a good way to specify options you use frequently. It's also a good way to add custom QL packs that you use regularly to your search path.

Using a CodeQL configuration file

You need to save the config file under your home (Linux and macOS) or user profile (Windows) directory in the .config/codeql/ subdirectory. For example, $HOME/.config/codeql/config.

The syntax for specifying options is as follows:

```
<command> <subcommand> <option> <value>
```

To apply the same options to more than one command you can:

- Omit the <subcommand>, which will specify the option for every <subcommand> to which it's relevant.
- Omit both <command> and <subcommand>, which will globally specify the option for every <command> and <subcommand> to which it's relevant.

  Note

  - config files only accept spaces between between option flags and values—CodeQL will throw an error if you use = to specify an option value.
  - If you specify an option in the command line, this overrides the config value defined for that option.

- If you want to specify more than one option for a <command>, <subcommand> or globally, use one line per option.

Examples

- To output all analysis results generated by `codeql database analyze` as CSV format, you would specify:

```
database analyze --format csv
```

  Here, you have to specify the command and subcommand to prevent any of the low-level commands that are executed during `database analyze` being passed the same `--format` option.

- To define the RAM (4096 MB) and number of threads (4) to use when running CodeQL commands, specify the following, on separate lines:

```
--ram 4096
--threads 4
```

- To globally specify a directory for CodeQL to scan for QL packs (which is not a sibling of the installation directory), use:

```
--search-path <path-to-directory>
```

3.2 CodeQL CLI reference

Learn more about the files you can use when running CodeQL processes and the results format and exit codes that CodeQL generates.

3.2.1 About CodeQL packs

Note

The CodeQL package management functionality, including CodeQL packs, is currently available as a beta release and is subject to change. During the beta release, CodeQL packs are available only using GitHub Packages - the GitHub Container registry. To use this beta functionality, install version 2.6.0 or higher of the CodeQL CLI bundle from: https://github.com/github/codeql-action/releases.

CodeQL packs are used to create, share, depend on, and run CodeQL queries and libraries. You can publish your own CodeQL packs and download packs created by others. CodeQL packs contain queries, library files, query suites, and metadata.

There are two types of CodeQL packs: query packs and library packs.

- Query packs are designed to be run. When a query pack is published, the bundle includes all the transitive dependencies and a compilation cache. This ensures consistent and efficient execution of the queries in the pack.
- Library packs are designed to be used by query packs (or other library packs) and do not contain queries themselves. The libraries are not compiled and there is no compilation cache included when the pack is published.

You can use the package management commands in the CodeQL CLI to create CodeQL packs, add dependencies to packs, and install or update dependencies. For more information, see "*Creating and working with CodeQL packs*." You can also publish and download CodeQL packs using the CodeQL CLI. For more information, see "*Publishing and using CodeQL packs*."

CodeQL pack structure

A CodeQL pack must contain a file called `qlpack.yml` in its root directory. In the `qlpack.yml` file, the `name:` field must have a value that follows the format of <scope>/<pack>, where <scope> is the GitHub organization or user account that the pack will be published to and <pack> is the name of the pack. The other files and directories within the pack should be logically organized. For example, typically:

- Queries are organized into directories for specific categories.
- Queries for specific products, libraries, and frameworks are organized into their own top-level directories.

About `qlpack.yml` files

When executing query-related commands, CodeQL first looks in siblings of the installation directory (and their subdirectories) for `qlpack.yml` files. Then it checks the package cache for CodeQL packs which have been downloaded. This means that when you are developing queries locally, the local packages in the installation directory override packages of the same name in the package cache, so that you can test your local changes.

The metadata in each *qlpack.yml`* file tells CodeQL how to compile any queries in the pack, what libraries the pack depends on, and where to find query suite definitions.

The contents of the CodeQL pack (queries or libraries used in CodeQL analysis) is included in the same directory as `qlpack.yml`, or its subdirectories.

The location of `qlpack.yml` defines the library path for the content of the CodeQL pack. That is, for all `.ql` and `.qll` files in the pack, CodeQL will resolve all import statements relative to the `qlpack.yml` at the pack's root.

`qlpack.yml` properties

The following properties are supported in `qlpack.yml` files.

Property	Example	Required	Purpose
`name`	`octo-org/security-que`	All packs	The scope, where the CodeQL pack is published, and the name of the pack defined using alphanumeric characters and hyphens. It must be unique as CodeQL cannot differentiate between CodeQL packs with identical names. Name components cannot start or end with a hyphen. Additionally, a period is not allowed in pack names at all. Use the pack name to specify queries to run using `database analyze` and to define dependencies between QL packs (see examples below).
`vers`	`0.0.0`	All packs	A version range for this CodeQL pack. This must be a valid semantic version that meets the SemVer v2.0.0 specification.
`depe`	`codeql/javascript-a ^1.2.3`	Optional	The names and version ranges of any CodeQL packs that this pack depends on, as a mapping. This gives the pack access to any libraries, database schema, and query suites defined in the dependency. For more information, see SemVer ranges in the NPM documentation.
`suit`	`octo-org-que`	Optional	The path to a directory in the pack that contains the query suites you want to make known to the CLI, defined relative to the pack directory. QL pack users can run "well-known" suites stored in this directory by specifying the pack name, without providing their full path. This is not supported for CodeQL packs downloaded from a package registry. For more information about query suites, see "*Creating CodeQL query suites*."
`extr`	`javascript`	All test packs	The CodeQL language extractor to use when the CLI creates a database in the pack. For more information about testing queries, see "*Testing custom queries*."
`test`	`.`	Optional for test packs	The path to a directory within the pack that contains tests, defined relative to the pack directory. Use `.` to specify the whole pack. Any queries in this directory are run as tests when `test run` is run with the `--strict-test-discovery` option. These queries are ignored by query suite definitions that use `queries` or `qlpack` instructions to ask for all queries in a particular pack.
`dbsc`	`semmlecode.python.dbscheme`	Core language packs only	The path to the *database schema* for all libraries and queries written for this CodeQL language (see example below).
`upgr`	`.`	Core language packs only	The path to a directory within the pack that contains upgrade scripts, defined relative to the pack directory. The `database upgrade` action uses these scripts to update databases that were created by an older version of an extractor so they're compatible with the current extractor (see *Upgrade scripts for a language* below.)
`auth`	`example@gith com`	All packs	Metadata that will be displayed on the packaging search page in the packages section of the account that the CodeQL pack is published to.
`lice`	`(LGPL-2.1 AND MIT)`	All packs	Metadata that will be displayed on the packaging search page in the packages section of the account that the CodeQL pack is published to. For a list of allowed licenses, see SPDX License List in the SPDX Specification.
`desc`	`Human-readab description of the contents of the CodeQL pack.`	All packs	Metadata that will be displayed on the packaging search page in the packages section of the account that the CodeQL pack is published to.

3.2.2 About QL packs

QL packs are used to organize the files used in CodeQL analysis. They contain queries, library files, query suites, and important metadata.

The CodeQL repository contains QL packs for C/C++, C#, Java, JavaScript, Python, and Ruby. The CodeQL for Go repository contains a QL pack for Go analysis. You can also make custom QL packs to contain your own queries and libraries.

QL pack structure

A QL pack must contain a file called `qlpack.yml` in its root directory. The other files and directories within the pack should be logically organized. For example, typically:

- Queries are organized into directories for specific categories.
- Queries for specific products, libraries, and frameworks are organized into their own top-level directories.
- There is a top-level directory named `<owner>/<language>` for query library (`.qll`) files. Within this directory, `.qll` files should be organized into subdirectories for specific categories.

About `qlpack.yml` files

When executing commands, CodeQL scans siblings of the installation directory (and their subdirectories) for `qlpack.yml` files. The metadata in the file tells CodeQL how to compile queries, what libraries the pack depends on, and where to find query suite definitions.

The content of the QL pack (queries and libraries used in CodeQL analysis) is included in the same directory as `qlpack.yml`, or its subdirectories.

The location of `qlpack.yml` defines the library path for the content of the QL pack. That is, for all `.ql` and `.qll` files in the QL pack, CodeQL will resolve all import statements relative to the `qlpack.yml` at the pack's root.

For example, in a QL pack with the following contents, you can import `CustomSinks.qll` from any location in the pack by declaring `import mycompany.java.CustomSinks`.

```
qlpack.yml
mycompany/
  java/
    security/
      CustomSinks.qll
Security/
CustomQuery.ql
```

For more information, see "*Importing modules*" in the QL language reference.

`qlpack.yml` properties

The following properties are supported in `qlpack.yml` files.

Property	Example	Required	Purpose
`name`	`org-queries`	All packs	The name of the QL pack defined using alphanumeric characters, hyphens, and periods. It must be unique as CodeQL cannot differentiate between QL packs with identical names. If you intend to distribute the pack, prefix the name with your (or your organization's) name followed by a hyphen. Use the pack name to specify queries to run using `database analyze` and to define dependencies between QL packs (see examples below).
`version`	`0.0.0`	All packs	A version number for this QL pack. This must be a valid semantic version that meets the SemVer v2.0.0 specification.
`libraryPathDependenci`	`codeql/javascript-all`	Optional	The names of any QL packs that this QL pack depends on, as a sequence. This gives the pack access to any libraries, database schema, and query suites defined in the dependency.
`suites`	`suites`	Optional	The path to a directory in the pack that contains the query suites you want to make known to the CLI, defined relative to the pack directory. QL pack users can run "well-known" suites stored in this directory by specifying the pack name, without providing their full path. For more information about query suites, see "*Creating CodeQL query suites*."
`extractor`	`javascript`	All test packs	The CodeQL language extractor to use when the CLI creates a database from test files in the pack. For more information about testing queries, see "*Testing custom queries*."
`tests`	`.`	Optional for test packs	Supported from release 2.1.0 onwards. The path to a directory within the pack that contains tests, defined relative to the pack directory. Use `.` to

Examples of custom QL packs

When you write custom queries or tests, you should save them in custom QL packs. For simplicity, try to organize each pack logically. For more information, see *QL pack structure*. Save files for queries and tests in separate packs and, where possible, organize custom packs into specific folders for each target language.

QL packs for custom queries

A custom QL pack for queries must include a `qlpack.yml` file at the pack root, containing `name`, `version`, and `libraryPathDependencies` properties. If the pack contains query suites, you can use the `suites` property to define their location. Query suites defined here are called "well-known" suites, and can be used on the command line by referring to their name only, rather than their full path. For more information about query suites, see "*Creating CodeQL query suites*."

For example, a `qlpack.yml` file for a QL pack featuring custom C++ queries and libraries may contain:

```
name: my-custom-queries
version: 0.0.0
libraryPathDependencies: codeql/cpp-all
suites: my-custom-suites
```

where `codeql/cpp-all` is the name of the QL pack for C/C++ analysis included in the CodeQL repository.

> Note
>
> When you create a custom QL pack, it's usually a good idea to add it to the search path in your CodeQL configuration. This will ensure that any libraries the pack contains are available to the CodeQL CLI. For more information, see "*Specifying command options in a CodeQL configuration file*."

QL packs for custom test files

For custom QL packs containing test files, you also need to include an `extractor` property so that the `test run` command knows how to create test databases. You may also wish to specify the `tests` property.

```
name: my-query-tests
version: 0.0.0
libraryPathDependencies: my-custom-queries
extractor: java
tests: .
```

This `qlpack.yml` file states that `my-query-tests` depends on `my-custom-queries`. It also declares that the CLI should use the Java `extractor` when creating test databases. Supported from CLI 2.1.0 onward, the `tests: .` line declares that all `.ql` files in the pack should be run as tests when `codeql test run` is run with the `--strict-test-discovery` option.

For more information about running tests, see "*Testing custom queries*."

Examples of QL packs in the CodeQL repository

Each of the languages in the CodeQL repository has four main QL packs:

- Core library pack for the language, with the *database schema* used by the language, and CodeQL libraries, and queries at `ql/<language>/ql/lib`
- Core query pack for the language that includes the default queries for the language, along with their query suites at `ql/<language>/ql/src`
- Tests for the core language libraries and queries at `ql/<language>/ql/test`
- Upgrade scripts for the language at `ql/<language>/upgrades`

Core library pack

The `qlpack.yml` file for a core library pack uses the following properties: `name`, `version`, `dbscheme`. The `dbscheme` property should only be defined in the core QL pack for a language.

For example, the `qlpack.yml` file for C/C++ analysis libraries contains:

```
name: codeql/cpp-all
version: 0.0.0
dbscheme: semmlecode.cpp.dbscheme
upgrades: upgrades
```

Core query pack

The `qlpack.yml` file for a core query pack uses the following properties: `name`, `version`, `suites`, `defaultSuiteFile`, `dependencies` .

For example, the `qlpack.yml` file for C/C++ analysis queries contains:

```
name: codeql/cpp-queries
version: 0.0.0
suites: codeql-suites
defaultSuiteFile: codeql-suites/cpp-code-scanning.qls
dependencies:
 codeql/cpp-all: "*"
 codeql/suite-helpers: "*"
```

Tests for the core QL pack

The `qlpack.yml` file for the tests for the core QL packs use the following properties: `name`, `version`, and `dependencies`. The `dependencies` always specifies the core QL pack.

For example, the `qlpack.yml` file for C/C++ analysis tests contains:

```
name: codeql/cpp-tests
version: 0.0.0
dependencies:
 codeql/cpp-all: "*"
 codeql/cpp-queries: "*"
```

3.2.3 Query reference files

A query reference file is text file that defines the location of one query to test.

You use a query reference file when you want to tell the `test run` subcommand to run a query that's not part of a test directory. There are two ways to specify queries that you want to run as tests:

1. Use a query reference file to specify the location of a query to test. This is useful when you create tests for alert and path queries that are intended to identify problems in real codebases. You might create several directories of test code, each focusing on different aspects of the query. Then you would add a query reference file to each directory of test code, to specify the query to test.
2. Add the query directly to a directory of tests. These is typically useful when you're writing queries explicitly to test the behavior of QL libraries. Often these queries contain just a few calls to library predicates, wrapping them in a `select` statement so their output can be tested.

Defining a query reference file

Each query reference file, `.qlref`, contains a single line that defines where to find one query. The location must be defined relative to the root of the QL pack that contains the query. Usually, this is a QL pack specified by the `libraryPathDependencies` for the test pack.

You should use forward slashes in the path on all operating systems to ensure compatibility between systems.

Example

A query reference file to test a JavaScript alert query: DeadAngularJSEventListener.qlref

The QL pack for the `javascript/ql/test` directory defines the `codeql-javascript` queries as a dependency. So the query reference file defines the location of the query relative to the `codeql-javascript` QL pack:

```
AngularJS/DeadAngularJSEventListener.ql
```

For another example, see *Testing custom queries*.

3.2.4 SARIF output

CodeQL supports SARIF as an output format for sharing static analysis results.

SARIF is designed to represent the output of a broad range of static analysis tools, and there are many features in the SARIF specification that are considered "optional". This document details the output produced when using the format type `sarifv2.1.0`, which corresponds to the SARIF v2.1.0.csd1 specification. For more information on selecting a file format for your analysis results, see the database analyze reference.

SARIF specification and schema

This topic is intended to be read alongside the detailed SARIF specification. For more information on the specification and the SARIF schema, see the SARIF specification documentation on GitHub.

Change notes

Changes between versions

CodeQL version	Format type	Changes
2.0.0	`sarifv2.1.0`	First version of this format.

Future changes to the output

The output produced for a given specific format type (for example `sarifv2.1.0`) may change in future CodeQL releases. We will endeavor to maintain backwards compatibility with consumers of the generated SARIF by ensuring that:

- No field which is marked as "Always" being generated will be removed.
- The circumstances under which "Optional" fields are generated may change. Consumers of the CodeQL SARIF output should be robust to the presence or absence of these fields.

New output fields may be added in future releases under the same format type–these are not considered to break backwards compatibility, and consumers should be robust to the presence of newly added fields.

New format argument types may be added in future versions of CodeQL—for example, to support new versions of SARIF. These have no guarantee of backwards compatibility, unless explicitly documented.

Generated SARIF objects

This details each SARIF component that may be generated, along with any specific circumstances. We omit any properties that are never generated.

`sarifLog` object

JSON property name	When is this generated?	Notes
`$schema`	Always	Provides a link to the SARIF schema.
`version`	Always	The version of the SARIF used to generate the output.
`runs`	Always	An array containing a single run object, for one language.

`run` object

JSON property name	When is this generated?	Notes
`tool`	Always	–
`origin`	Always	A dictionary of `uriBaseIds` to artifactLocations representing the original locations on the analysis machine. At a minimum, this will contain the `%SRCROOT%` `uriBaseId`, which represents the root location on the analysis machine of the source code for the analyzed project. Each `artifactLocation` will contain the `uri` and `description` properties.
`artifa`	Always	An array containing at least one artifact object for every file referenced in a result.
`result`	Always	–
`newLin`	Always	–
`column`	Always	–
`proper`	Always	The properties dictionary will contain the `semmle.formatSpecifier`, which identifies the format specifier passed to the CodeQL CLI.

`tool` object

JSON property name	When is this generated?	Notes
`driver`	Always	–

`toolComponent` object

JSON property name	When is this generated?	Notes
`name`	Always	Set to "CodeQL command-line toolchain" for output from the CodeQL CLI tools. Note, if the output was generated using a different tool a different `name` is reported, and the format may not be as described here.
`organiz`	Always	Set to "GitHub".
`version`	Always	Set to the CodeQL release version e.g. "2.0.0".
`rules`	Always	An array of `reportingDescriptor` objects that represent rules. This array will contain, at a minimum, all the rules that were run during this analysis, but may contain rules which were available but not run. For more detail about enabling queries, see `defaultConfiguration`.

`reportingDescriptor` object (for rule)

`reportingDescriptor` objects may be used in multiple places in the SARIF specification. When a `reportingDescriptor` is included in the rules array of a `toolComponent` object it has the following properties.

JSON property name	When is this generated?	Notes
`id`	Always	Will contain the `@id` property specified in the query that defines the rule, which is usually of the format `language/rule-name` (for example `cpp/unsafe-format-string`). If your organization defines the `@opaqueid` property in the query it will be used instead.
`name`	Always	Will contain the `@id` property specified in the query. See the `id` property above for an example.
`shortDes`	Always	Will contain the `@name` property specified in the query that defines the rule.
`fullDesc`	Always	Will contain the `@description` property specified in the query that defines the rule.
`defaultC`	Always	A `reportingConfiguration` object, with the enabled property set to true or false, and a level property set according to the `@severity` property specified in the query that defines the rule. Omitted if the `@severity` property was not specified.

`artifact` object

JSON property name	When is this generated?	Notes
`locatior`	Always	An `artifactLocation` object.
`index`	Always	The index of the `artifact` object.
`contents`	Optionally	If results are generated using the `--sarif-add-file-contents` flag, and the source code is available at the time the SARIF file is generated, then the `contents` property is populated with an `artifactContent` object, with the `text` property set.

`artifactLocation` object

JSON property name	When is this generated?	Notes
`uri`	Always	–
`index`	Always	–
`uriBaseId`	Optionally	If the file is relative to some known abstract location, such as the root source location on the analysis machine, this will be set.

`result` object

The composition of the results is dependent on the options provided to CodeQL. By default, the results are grouped by unique message format string and primary location. Thus, two results that occur at the same location with the same underlying message, will appear as a single result in the output. This behavior can be disabled by using the flag `--ungroup-results`, in which case no results are grouped.

JSON property name	When is this generated?	Notes
`ruleId`	Always	See the description of the `id` property in `reportingDescriptor` object (for rule) .
`ruleInde`	Always	–
`message`	Always	A message describing the problem(s) occurring at this location. This message may be a SARIF "Message with placeholder", containing links that refer to locations in the `relatedLocations` property.
`location`	Always	An array containing a single `location` object.
`partialF`	Always	A dictionary from named fingerprint types to the fingerprint. This will contain, at a minimum, a value for the `primaryLocationLineHash`, which provides a fingerprint based on the context of the primary location.
`codeFlow`	Optionally	This array may be populated with one or more `codeFlow` objects if the query that defines the rule for this result is of `@kind path-problem`.
`relatedL`	Optionally	This array will be populated if the query that defines the rule for this result has a message with placeholder options. Each unique location is included once.
`suppress`	Optionally	If the result is suppressed, then this will contain a single `suppression` object, with the `@kind` property set to `IN_SOURCE`. If this result is not suppressed, but there is at least one result that has a suppression, then this will be set to an empty array, otherwise it will not be set.

`location` object

JSON property name	When is this generated?	Notes
`physicalLocation`	Always	–
`id`	Optionally	`location` objects that appear in the `relatedLocations` array of a `result` object may contain the `id` property.
`message`	Optionally	`location` objects may contain the `message` property if: • They appear in the `relatedLocations` array of a `result` object may contain the `message` property. • They appear in the `threadFlowLocation.location` property.

`physicalLocation` object

JSON property name	When is this generated?	Notes
`artifactLocat`	Always	–
`region`	Optionally	If the given `physicalLocation` exists in a text file, such as a source code file, then the `region` property may be present.
`contextRegio`	Optionally	May be present if this location has an associated `snippet`.

`region` object

There are two types of `region` object produced by CodeQL:

- Line/column offset regions
- Character offset and length regions

Any region produced by CodeQL may be specified in either format, and consumers should robustly handle either type.

For line/column offset regions, the following properties will be set:

JSON property name	When is this generated?	Notes
`startLine`	Always	–
`startColumn`	Optionally	Not included if equal to the default value of 1.
`endLine`	Optionally	Not included if identical to `startLine`.
`endColumn`	Always	–
`snippet`	Optionally	–

For character offset and length regions, the following properties will be set:

JSON property name	When is this generated?	Notes
`charOffset`	Optionally	Provided if `startLine`, `startColumn`, `endLine`, and `endColumn` are not populated.
`charLength`	Optionally	Provided if `startLine`, `startColumn`, `endLine`, and `endColumn` are not populated.
`snippet`	Optionally	–

`codeFlow` object

JSON property name	When is this generated?	Notes
`threadFlows`	Always	–

`threadFlow` object

JSON property name	When is this generated?	Notes
`locations`	Always	–

`threadFlowLocation` object

JSON property name	When is this generated?	Notes
`location`	Always	–

3.2.5 Exit codes

The CodeQL CLI reports the status of each command it runs as an exit code. This exit code provides information for subsequent commands or for other tools that rely on the CodeQL CLI.

0

Success, normal termination.

1

The command successfully determined that the answer to your question is "no".

This exit code is only used by a few commands, such as codeql test run, codeql database check, codeql query format,and codeql resolve extractor. For more details, see the documentation for those commands.

2

Something went wrong.

The CLI writes a human-readable error message to stderr. This includes cases where an extractor fails with an internal error, because the `codeql` driver can't distinguish between internal and user-facing errors in extractor behavior.

3

The launcher was unable to find the CodeQL installation directory.

In this case, the launcher can't start the Java code for the CodeQL CLI at all. This should only happen when something is severely wrong with the CodeQL installation.

32

The extractor didn't find any code to analyze when running codeql database create or codeql database finalize.

33

One or more query evaluations timed out.

It's possible that some queries that were evaluated in parallel didn't time out. The results for those queries are produced as usual.

98

Evaluation was explicitly canceled.

99

The CodeQL CLI ran out of memory.

This doesn't necessarily mean that all the machine's physical RAM has been used. If you don't use the `--ram` option to set a limit explicitly, the JVM decides on a default limit at startup.

100

A fatal internal error occurred.

This should be considered a bug. The CLI usually writes an abbreviated error description to stderr. If you can reproduce the bug, it's helpful to use `--logdir` and send the log files to GitHub in a bug report.

Other

In the case of really severe problems within the JVM that runs `codeql`, it might return a nonzero exit code of its own choosing. This should only happen if something is severely wrong with the CodeQL installation.

- *About CodeQL packs*: CodeQL packs are created with the CodeQL CLI and are used to create, depend on, publish, and run CodeQL queries and libraries.
- *About QL packs*: QL packs are used to organize the files used in CodeQL analysis. They contain queries, library files, query suites, and important metadata.
- *Query reference files*: A query reference file is text file that defines the location of one query to test.
- *SARIF output*: CodeQL supports SARIF as an output format for sharing static analysis results.
- *Exit codes*: The CodeQL CLI reports the status of each command it runs as an exit code. This exit code provides information for subsequent commands or for other tools that rely on the CodeQL CLI.
- *Extractor options*: You can customize the behavior of extractors by setting options through the CodeQL CLI.

CHAPTER

FOUR

WRITING CODEQL QUERIES

Get to know more about queries and learn some key query-writing skills by solving puzzles.

- *CodeQL queries*: CodeQL queries are used in code scanning analyses to find problems in source code, including potential security vulnerabilities.
- *QL tutorials*: Solve puzzles to learn the basics of QL before you analyze code with CodeQL. The tutorials teach you how to write queries and introduce you to key logic concepts along the way.

4.1 CodeQL queries

CodeQL queries are used in code scanning analyses to find problems in source code, including potential security vulnerabilities.

4.1.1 About CodeQL queries

CodeQL queries are used to analyze code for issues related to security, correctness, maintainability, and readability.

Overview

CodeQL includes queries to find the most relevant and interesting problems for each supported language. You can also write custom queries to find specific issues relevant to your own project. The important types of query are:

- **Alert queries**: queries that highlight issues in specific locations in your code.
- **Path queries**: queries that describe the flow of information between a source and a sink in your code.

You can add custom queries to *QL packs* to analyze your projects with "Code scanning", use them to analyze a database with the "*CodeQL CLI*," or you can contribute to the standard CodeQL queries in our open source repository on GitHub.

This topic is a basic introduction to query files. You can find more information on writing queries for specific programming languages in the "*CodeQL language guides*," and detailed technical information about QL in the "*QL language reference*." For more information on how to format your code when contributing queries to the GitHub repository, see the CodeQL style guide.

Basic query structure

Queries written with CodeQL have the file extension `.ql`, and contain a `select` clause. Many of the existing queries include additional optional information, and have the following structure:

```
/**
 *
 * Query metadata
 *
 */

import /* ... CodeQL libraries or modules ... */

/* ... Optional, define CodeQL classes and predicates ... */

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

The following sections describe the information that is typically included in a query file for alerts. Path queries are discussed in more detail in "*Creating path queries*."

Query metadata

Query metadata is used to identify your custom queries when they are added to the GitHub repository or used in your analysis. Metadata provides information about the query's purpose, and also specifies how to interpret and display the query results. For a full list of metadata properties, see "*Metadata for CodeQL queries*." The exact metadata requirement depends on how you are going to run your query:

- If you are contributing a query to the GitHub repository, please read the query metadata style guide.
- If you are adding a custom query to a query pack for analysis using LGTM , see Writing custom queries to include in LGTM analysis.
- If you are analyzing a database using the *CodeQL CLI*, your query metadata must contain `@kind`.
- If you are running a query in the query console on LGTM or with the CodeQL extension for VS Code, metadata is not mandatory. However, if you want your results to be displayed as either an 'alert' or a 'path', you must specify the correct `@kind` property, as explained below. For more information, see Using the query console on LGTM.com and "*Analyzing your projects*" in the CodeQL for VS Code help.

  Note

  Queries that are contributed to the open source repository, added to a query pack in LGTM, or used to analyze a database with the *CodeQL CLI* must have a query type (`@kind`) specified. The `@kind` property indicates how to interpret and display the results of the query analysis:

  - Alert query metadata must contain `@kind problem` to identify the results as a simple alert.
  - Path query metadata must contain `@kind path-problem` to identify the results as an alert documented by a sequence of code locations.
  - Diagnostic query metadata must contain `@kind diagnostic` to identify the results as troubleshooting data about the extraction process.
  - Summary query metadata must contain `@kind metric` and `@tags summary` to identify the results as summary metrics for the CodeQL database.

When you define the `@kind` property of a custom query you must also ensure that the rest of your query has the correct structure in order to be valid, as described below.

Import statements

Each query generally contains one or more `import` statements, which define the *libraries* or *modules* to import into the query. Libraries and modules provide a way of grouping together related *types*, *predicates*, and other modules. The contents of each library or module that you import can then be accessed by the query. Our open source repository on GitHub contains the standard CodeQL libraries for each supported language.

When writing your own alert queries, you would typically import the standard library for the language of the project that you are querying, using `import` followed by a language:

- C/C++: `cpp`
- C#: `csharp`
- Go: `go`
- Java: `java`
- JavaScript/TypeScript: `javascript`
- Python: `python`

There are also libraries containing commonly used predicates, types, and other modules associated with different analyses, including data flow, control flow, and taint-tracking. In order to calculate path graphs, path queries require you to import a data flow library into the query file. For more information, see "*Creating path queries*."

You can explore the contents of all the standard libraries in the CodeQL library reference documentation or in the GitHub repository.

Optional CodeQL classes and predicates

You can customize your analysis by defining your own predicates and classes in the query. For further information, see *Defining a predicate* and *Defining a class*.

From clause

The `from` clause declares the variables that are used in the query. Each declaration must be of the form `<type> <variable name>`. For more information on the available *types*, and to learn how to define your own types using *classes*, see the *QL language reference*.

Where clause

The `where` clause defines the logical conditions to apply to the variables declared in the `from` clause to generate your results. This clause uses *aggregations*, *predicates*, and logical *formulas* to limit the variables of interest to a smaller set, which meet the defined conditions. The CodeQL libraries group commonly used predicates for specific languages and frameworks. You can also define your own predicates in the body of the query file or in your own custom modules, as described above.

Select clause

The `select` clause specifies the results to display for the variables that meet the conditions defined in the `where` clause. The valid structure for the select clause is defined by the `@kind` property specified in the metadata.

Select clauses for alert queries (`@kind problem`) consist of two 'columns', with the following structure:

```
select element, string
```

- `element`: a code element that is identified by the query, which defines where the alert is displayed.
- `string`: a message, which can also include links and placeholders, explaining why the alert was generated.

You can modify the alert message defined in the final column of the `select` statement to give more detail about the alert or path found by the query using links and placeholders. For more information, see "*Defining the results of a query*."

Select clauses for path queries (`@kind path-problem`) are crafted to display both an alert and the source and sink of an associated path graph. For more information, see "*Creating path queries*."

Select clauses for diagnostic queries (`@kind diagnostic`) and summary metric queries (`@kind metric` and `@tags summary`) have different requirements. For examples, see the diagnostic queries and the summary metric queries in the CodeQL repository.

Viewing the standard CodeQL queries

One of the easiest ways to get started writing your own queries is to modify an existing query. To view the standard CodeQL queries, or to try out other examples, visit the CodeQL and CodeQL for Go repositories on GitHub.

You can also find examples of queries developed to find security vulnerabilities and bugs in open source software projects on the GitHub Security Lab website and in the associated repository.

Contributing queries

Contributions to the standard queries and libraries are very welcome. For more information, see our contributing guidelines. If you are contributing a query to the open source GitHub repository, writing a custom query for LGTM, or using a custom query in an analysis with the CodeQL CLI, then you need to include extra metadata in your query to ensure that the query results are interpreted and displayed correctly. See the following topics for more information on query metadata:

- "*Metadata for CodeQL queries*"
- Query metadata style guide on GitHub

Query contributions to the open source GitHub repository may also have an accompanying query help file to provide information about their purpose for other users. For more information on writing query help, see the Query help style guide on GitHub and the "*Query help files*."

Query help files

When you write a custom query, we also recommend that you write a query help file to explain the purpose of the query to other users. For more information, see the Query help style guide on GitHub, and the "*Query help files*."

4.1.2 Metadata for CodeQL queries

Metadata tells users important information about CodeQL queries. You must include the correct query metadata in a query to be able to view query results in source code.

About query metadata

Any query that is run as part of an analysis includes a number of properties, known as query metadata. Metadata is included at the top of each query file as the content of a QLDoc comment. This metadata tells LGTM and the CodeQL *extension for VS Code* how to handle the query and display its results correctly. It also gives other users information about what the query results mean. For more information on query metadata, see the query metadata style guide in our open source repository on GitHub.

> Note
>
> The exact metadata requirement depends on how you are going to run your query. For more information, see the section on query metadata in "*About CodeQL queries*."

Metadata properties

The following properties are supported by all query files:

Property	Value	Description
`@description`	`<text>`	A sentence or short paragraph to describe the purpose of the query and *why* the result is useful or important. The description is written in plain text, and uses single quotes (`'`) to enclose code elements.
`@id`	`<text>`	A sequence of words composed of lowercase letters or digits, delimited by `/` or `-`, identifying and classifying the query. Each query must have a **unique** ID. To ensure this, it may be helpful to use a fixed structure for each ID. For example, the standard LGTM queries have the following format: `<language>/<brief-description>`.
`@kind`	`problem` `path-problem`	Identifies the query is an alert (`@kind problem`) or a path (`@kind path-problem`). For more information on these query types, see "*About CodeQL queries*."
`@name`	`<text>`	A statement that defines the label of the query. The name is written in plain text, and uses single quotes (`'`) to enclose code elements.

`@tags`	`correctness`
	`maintainability`
	`readability`
	`security`
		These tags group queries together in broad categories to make it easier to search for them and identify them. In addition to the common tags listed here, there are also a number of more specific categories. For more information, see the Query metadata style guide.

`@precision`	`low`
	`medium`
	`high`
	`very-high`
		Indicates the percentage of query results that are true positives (as opposed to false positive results). This, along with the `@problem.severity` property, determines whether the results are displayed by default on LGTM.

`@problem.severity`	`error`
	`warning`
	`recommendation`
		Defines the level of severity of any alerts generated by a non-security query. This, along with the `@precision` property, determines whether the results are displayed by default on LGTM.

`@security-severity`	`<score>`	Defines the level of severity, between 0.0 and 10.0, for queries with `@tags security`. For more information about calculating `@security-severity`, see the GitHub changelog.

Example

Here is the metadata for one of the standard Java queries:

```
/**
 * @name Type mismatch on container modification
 * @description Calling container modification methods such as 'Collection.remove'
 *              or 'Map.remove' with an object of a type that is incompatible with
 *              the corresponding container element type is unlikely to have any effect.
 * @kind problem
 * @problem.severity error
 * @precision very-high
 * @id java/type-mismatch-modification
 * @tags reliability
 *       correctness
 *       logic
 */
```

For more examples of query metadata, see the standard CodeQL queries in our GitHub repository.

4.1.3 Query help files

Query help files tell users the purpose of a query, and recommend how to solve the potential problem the query finds.

This topic provides detailed information on the structure of query help files. For more information about how to write useful query help in a style that is consistent with the standard CodeQL queries, see the Query help style guide on GitHub.

> Note
>
> You can access the query help for CodeQL queries by visiting CodeQL query help. You can also access the raw query help files in the GitHub repository. For example, see the JavaScript security queries and C/C++ critical queries.
>
> For queries run by default on LGTM, there are several different ways to access the query help. For further information, see Where do I see the query help for a query on LGTM? in the LGTM user help.

Overview

Each query help file provides detailed information about the purpose and use of a query. When you write your own queries, we recommend that you also write query help files so that other users know what the queries do, and how they work.

Structure

Query help files are written using a custom XML format, and stored in a file with a `.qhelp` extension. Query help files must have the same base name as the query they describe, and must be located in the same directory. The basic structure is as follows:

```
<!DOCTYPE qhelp SYSTEM "qhelp.dtd">
<qhelp>
    CONTAINS one or more section-level elements
</qhelp>
```

The header and single top-level `qhelp` element are both mandatory. The following sections explain additional elements that you may include in your query help files.

> Code scanning does not process `.qhelp` files for custom CodeQL queries, so to show query help for custom queries in the code scanning UI you must convert the `.qhelp` files to markdown and then include the markdown-rendered query help in SARIF files generated during an analysis. For more information, see "*Analyzing databases with the CodeQL CLI*."

Section-level elements

Section-level elements are used to group the information in the help file into sections. Many sections have a heading, either defined by a `title` attribute or a default value. The following section-level elements are optional child elements of the `qhelp` element.

Element	Attributes	Children	Purpose of section
`example`	None	Any block element	Demonstrate an example of code that violates the rule implemented by the query with guidance on how to fix it. Default heading.
`fragment`	None	Any block element	See "*Query help inclusion*" below. No heading.
`hr`	None	None	A horizontal rule. No heading.
`include`	`src` The query help file to include.	None	Include a query help file at the location of this element. See "*Query help inclusion*" below. No heading.
`overview`	None	Any block element	Overview of the purpose of the query. Typically this is the first section in a query document. No heading.
`recommen`	None	Any block element	Recommend how to address any alerts that this query identifies. Default heading.
`referenc`	None	`li` elements	Reference list. Typically this is the last section in a query document. Default heading.
`section`	`title` Title of the section	Any block element	General-purpose section with a heading defined by the `title` attribute.
`semmleNo`	None	Any block element	Implementation notes about the query. This section is used only for queries that implement a rule defined by a third party. Default heading.

Block elements

The following elements are optional child elements of the `section`, `example`, `fragment`, `recommendation`, `overview`, and `semmleNotes` elements.

Element	Attributes	Children	Purpose of block
`blockquot`	None	Any block element	Display a quoted paragraph.
`img`	`src` The image file to include. `alt` Text for the image's alt text. `height` Optional, height of the image. `width` Optional, the width of the image.	None	Display an image. The content of the image is in a separate image file.
`include`	`src` The query help file to include.	None	Include a query help file at the location of this element. See *Query help inclusion* below for more information.
`ol`	None	`li`	Display an ordered list. See List elements below.
`p`	None	Any inline content	Display a paragraph, used as in HTML files.
`pre`	None	Text	Display text in a monospaced font with preformatted whitespace.
`sample`	`language` The language of the in-line code sample. `src` Optional, the file containing the sample code.	Text	Display sample code either defined as nested text in the `sample` element or defined in the `src` file specified. When `src` is specified, the language is inferred from the file extension. If `src` is omitted, then language must be provided and the sample code provided as nested text.
`table`	None	`tbody`	Display a table. See Tables below.
`ul`	None	`li`	Display an unordered list. See List elements below.
`warning`	None	Text	Display a warning that will be displayed very visibly on the resulting page. Such warnings are sometimes used on queries that are known to have low precision for many code bases; such queries are often disabled by default.

List elements

Query help files support two types of block elements for lists: `ul` and `ol`. Both block elements support only one child elements of the type `li`. Each `li` element contains either inline content or a block element.

Table elements

The `table` block element is used to include a table in a query help file. Each table includes a number of rows, each of which includes a number of cells. The data in the cells will be rendered as a grid.

Element	Attributes	Children	Purpose
`tbody`	None	`tr`	Defines the top-level element of a table.
`tr`	None	`th` `td`	Defines one row of a table.
`td`	None	Any inline content	Defines one cell of a table row.
`th`	None	Any inline content	Defines one header cell of a table row.

Inline content

Inline content is used to define the content for paragraphs, list items, table cells, and similar elements. Inline content includes text in addition to the inline elements defined below:

Element	Attributes	Children	Purpose
a	href The URL of the link.	text	Defines hyperlink. When a user selects the child text, they will be redirected to the given URL.
b	None	Inline content	Defines content that should be displayed as bold face.
code	None	Inline content	Defines content representing code. It is typically shown in a monospace font.
em	None	Inline content	Defines content that should be emphasized, typically by italicizing it.
i	None	Inline content	Defines content that should be displayed as italics.
img	src alt height width	None	Display an image. See the description above in Block elements.
strong	None	Inline content	Defines content that should be rendered more strongly, typically using bold face.
sub	None	Inline content	Defines content that should be rendered as subscript.
sup	None	Inline content	Defines content that should be rendered as superscript.
tt	None	Inline content	Defines content that should be displayed with a monospace font.

Query help inclusion

To reuse content between different help topics, you can store shared content in one query help file and then include it in a number of other query help files using the `include` element. The shared content can be stored either in the same directory as the including files, or in `SEMMLE_DIST/docs/include`. When a query help file is only included by other help files but does not belong to a specific query, it should have the file extension `.inc.qhelp`.

The `include` element can be used as a section or block element. The content of the query help file defined by the `src` attribute must contain elements that are appropriate to the location of the `include` element.

Section-level include elements

Section-level `include` elements can be located beneath the top-level `qhelp` element. For example, in StoredXSS.qhelp, a full query help file is reused:

```
<qhelp>
    <include src="XSS.qhelp" />
</qhelp>
```

In this example, the XSS.qhelp file must conform to the standard for a full query help file as described above. That is, the `qhelp` element may only contain non-`fragment`, section-level elements.

Block-level include elements

Block-level `include` elements can be included beneath section-level elements. For example, an `include` element is used beneath the `overview` section in ThreadUnsafeICryptoTransform.qhelp:

```
<qhelp>
    <overview>
        <include src="ThreadUnsafeICryptoTransformOverview.inc.qhelp" />
    </overview>
    ...
</qhelp>
```

The included file, ThreadUnsafeICryptoTransformOverview.inc.qhelp, may only contain one or more `fragment` sections. For example:

```
<!DOCTYPE qhelp SYSTEM "qhelp.dtd">
<qhelp>
   <fragment>
      <p>
         ...
      </p>
   </fragment>
</qhelp>
```

4.1.4 Defining the results of a query

You can control how analysis results are displayed in source code by modifying a query's `select` statement.

About query results

The information contained in the results of a query is controlled by the `select` statement. Part of the process of developing a useful query is to make the results clear and easy for other users to understand. When you write your own queries in the query console or in the CodeQL *extension for VS Code* there are no constraints on what can be selected. However, if you want to use a query to create alerts in LGTM or generate valid analysis results using the *CodeQL CLI*, you'll need to make the `select` statement report results in the required format. You must also ensure that the query has the appropriate metadata properties defined. This topic explains how to write your select statement to generate helpful analysis results.

Overview

Alert queries must have the property `@kind problem` defined in their metadata. For more information, see "*Metadata for CodeQL queries*." In their most basic form, the `select` statement must select two 'columns':

- **Element**—a code element that's identified by the query. This defines the location of the alert.
- **String**—a message to display for this code element, describing why the alert was generated.

If you look at some of the LGTM queries, you'll see that they can select extra element/string pairs, which are combined with `$@` placeholder markers in the message to form links. For example, Dereferenced variable may be null (Java), or Duplicate switch case (JavaScript).

> Note
>
> An in-depth discussion of `select` statements for path queries is not included in this topic. However, you can develop the string column of the `select` statement in the same way as for alert queries. For more specific information about path queries, see "*Creating path queries*."

Developing a select statement

Here's a simple query that uses the standard CodeQL `CodeDuplication.qll` library to identify similar files.

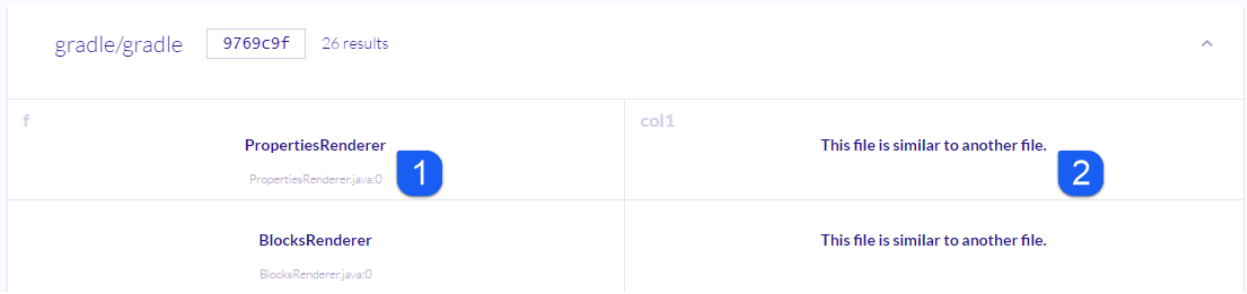
Basic select statement

```
import java
import external.CodeDuplication

from File f, File other, int percent
where similarFiles(f, other, percent)
select f, "This file is similar to another file."
```

This basic select statement has two columns:

1. Element to display the alert on: `f` corresponds to `File`.
2. String message to display: `"This file is similar to another file."`

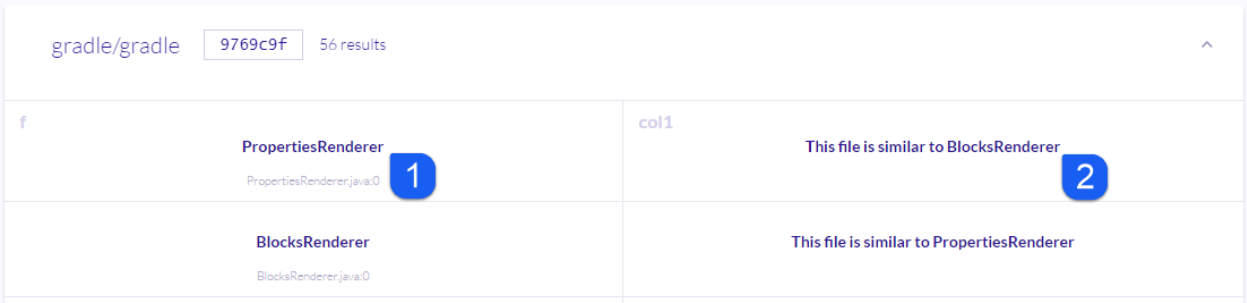


Including the name of the similar file

The alert message defined by the basic select statement is constant and doesn't give users much information. Since the query identifies the similar file (`other`), it's easy to extend the `select` statement to report the name of the similar file. For example:

```
select f, "This file is similar to " + other.getBaseName()
```

1. Element: `f` as before.
2. String message: `"This file is similar to "`—the string text is combined with the file name for the `other`, similar file, returned by `getBaseName()`.



While this is more informative than the original select statement, the user still needs to find the other file manually.

Adding a link to the similar file

You can use placeholders in the text of alert messages to insert additional information, such as links to the similar file. Placeholders are defined using `$@`, and filled using the information in the next two columns of the select statement. For example, this select statement returns four columns:

```
select f, "This file is similar to $@.", other, other.getBaseName()
```

1. Element: `f` as before.
2. String message: `"This file is similar to $@."`—the string text now includes a placeholder, which will display the combined content of the next two columns.
3. Element for placeholder: `other` corresponds to the similar file.
4. String text for placeholder: the short file name returned by `other.getBaseName()`.

When the alert message is displayed, the `$@` placeholder is replaced by a link created from the contents of the third and fourth columns defined by the `select` statement.

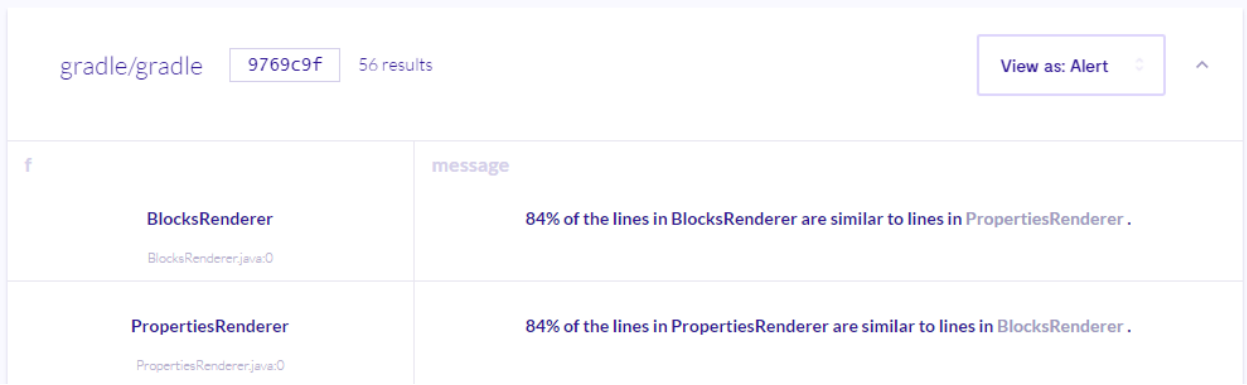
If you use the `$@` placeholder marker multiple times in the description text, then the `N`th use is replaced by a link formed from columns `2N+2` and `2N+3`. If there are more pairs of additional columns than there are placeholder markers, then the trailing columns are ignored. Conversely, if there are fewer pairs of additional columns than there are placeholder markers, then the trailing markers are treated as normal text rather than placeholder markers.

Adding details of the extent of similarity

You could go further and change the `select` statement to report on the similarity of content in the two files, since this information is already available in the query. For example:

```
select f, percent + "% of the lines in " + f.getBaseName() + " are similar to lines in
↪$@.", other, other.getBaseName()
```

The new elements added here don't need to be clickable, so we added them directly to the description string.



Further reading

- CodeQL repository

4.1.5 Providing locations in CodeQL queries

CodeQL includes mechanisms for extracting the location of elements in a codebase. Use these mechanisms when writing custom CodeQL queries and libraries to help display information to users.

About locations

When displaying information to the user, LGTM needs to be able to extract location information from the results of a query. In order to do this, all QL classes which can provide location information should do this by using one of the following mechanisms:

- *Providing URLs*
- *Providing location information*
- *Using extracted location information*

This list is in priority order, so that the first available mechanism is used.

> Note
>
> Since QL is a relational language, there is nothing to enforce that each entity of a QL class is mapped to precisely one location. This is the responsibility of the designer of the library (or the extractor, in the case of the third option below). If entities are assigned no location at all, users will not be able to click through from query results to the source code viewer. If multiple locations are assigned, results may be duplicated.

Providing URLs

A custom URL can be provided by defining a QL predicate returning `string` with the name `getURL` – note that capitalization matters, and no arguments are allowed. For example:

```
class JiraIssue extends ExternalData {
    JiraIssue() {
        getDataPath() = "JiraIssues.csv"
    }

    string getKey() {
        result = getField(0)
    }

    string getURL() {
        result = "http://mycompany.com/jira/" + getKey()
    }
}
```

File URLs

LGTM supports the display of URLs which define a line and column in a source file.

The schema is `file://`, which is followed by the absolute path to a file, followed by four numbers separated by colons. The numbers denote start line, start column, end line and end column. Both line and column numbers are **1-based**, for example:

- `file://opt/src/my/file.java:0:0:0:0` is used to link to an entire file.
- `file:///opt/src/my/file.java:1:1:2:1` denotes the location that starts at the beginning of the file and extends to the first character of the second line (the range is inclusive).
- `file:///opt/src/my/file.java:1:0:1:0` is taken, by convention, to denote the entire first line of the file.

By convention, the location of an entire file may also be denoted by a `file://` URL without trailing numbers. Optionally, the location within a file can be denoted using three numbers to define the start line number, character offset and character length of the location respectively. Results of these types are not displayed in LGTM.

Other types of URL

The following, less-common types of URL are valid but are not supported by LGTM and will be omitted from any results:

- **HTTP URLs** are supported in some client applications. For an example, see the code snippet above.
- **Folder URLs** can be useful, for example to provide folder-level metrics. They may use a file URL, for example `file:///opt/src:0:0:0:0`, but they may also start with a scheme of `folder://`, and no trailing numbers, for example `folder:///opt/src`.
- **Relative file URLs** are like normal file URLs, but start with the scheme `relative://`. They are typically only meaningful in the context of a particular database, and are taken to be implicitly prefixed by the database's source location. Note that, in particular, the relative URL of a file will stay constant regardless of where the database is analyzed. It is often most convenient to produce these URLs as input when importing external information; selecting one from a QL class would be unusual, and client applications may not handle it appropriately.

Providing location information

If no `getURL()` member predicate is defined, a QL class is checked for the presence of a member predicate called `hasLocationInfo(..)`. This can be understood as a convenient way of providing file URLs (see above) without constructing the long URL string in QL. `hasLocationInfo(..)` should be a predicate, its first column must be `string`-typed (it corresponds to the "path" portion of a file URL), and it must have an additional 3 or 4 `int`-typed columns, which are interpreted like a trailing group of three or four numbers on a file URL.

For example, let us imagine that the locations for methods provided by the extractor extend from the first character of the method name to the closing curly brace of the method body, and we want to "fix" them to ensure that only the method name is selected. The following code shows two ways of achieving this:

```
class MyMethod extends Method {
    // The locations from the database, which we want to modify.
    Location getLocation() { result = super.getLocation() }

    /* First member predicate: Construct a URL for the desired location. */
    string getURL() {
        exists(Location loc | loc = this.getLocation() |
            result = "file://" + loc.getFile().getFullName() +
                ":" + loc.getStartLine() +
                ":" + loc.getStartColumn() +
                ":" + loc.getStartLine() +
                ":" + (loc.getStartColumn() + getName().length() - 1)
        )
    }

    /* Second member predicate: Define hasLocationInfo. This will be more
       efficient (it avoids constructing long strings), and will
       only be used if getURL() is not defined. */
    predicate hasLocationInfo(string path, int sl, int sc, int el, int ec) {
        exists(Location loc | loc = this.getLocation() |
            path = loc.getFile().getFullName() and
            sl = loc.getStartLine() and
            sc = loc.getStartColumn() and
            el = sl and
            ec = sc + getName().length() - 1
        )
    }
}
```

Using extracted location information

Finally, if the above two predicates fail, client applications will attempt to call a predicate called `getLocation()` with no parameters, and try to apply one of the above two predicates to the result. This allows certain locations to be put into the database, assigned identifiers, and picked up.

By convention, the return value of the `getLocation()` predicate should be a class called `Location`, and it should define a version of `hasLocationInfo(..)` (or `getURL()`, though the former is preferable). If the `Location` class does not provide either of these member predicates, then no location information will be available.

The `toString()` predicate

All classes except those that extend primitive types, must provide a `string toString()` member predicate. The query compiler will complain if you don't. The uniqueness warning, noted above for locations, applies here too.

Further reading

- CodeQL repository

4.1.6 About data flow analysis

Data flow analysis is used to compute the possible values that a variable can hold at various points in a program, determining how those values propagate through the program and where they are used.

Overview

Many CodeQL security queries implement data flow analysis, which can highlight the fate of potentially malicious or insecure data that can cause vulnerabilities in your code base. These queries help you understand if data is used in an insecure way, whether dangerous arguments are passed to functions, or whether sensitive data can leak. As well as highlighting potential security issues, you can also use data flow analysis to understand other aspects of how a program behaves, by finding, for example, uses of uninitialized variables and resource leaks.

The following sections provide a brief introduction to data flow analysis with CodeQL.

See the following tutorials for more information about analyzing data flow in specific languages:

- "*Analyzing data flow in C/C++*"
- "*Analyzing data flow in C#*"
- "*Analyzing data flow in Java*"
- "*Analyzing data flow in JavaScript/TypeScript*"
- "*Analyzing data flow in Python*"

  Note

  Data flow analysis is used extensively in path queries. To learn more about path queries, see "*Creating path queries*."

Data flow graph

The CodeQL data flow libraries implement data flow analysis on a program or function by modeling its data flow graph. Unlike the abstract syntax tree, the data flow graph does not reflect the syntactic structure of the program, but models the way data flows through the program at runtime. Nodes in the abstract syntax tree represent syntactic elements such as statements or expressions. Nodes in the data flow graph, on the other hand, represent semantic elements that carry values at runtime.

Some AST nodes (such as expressions) have corresponding data flow nodes, but others (such as `if` statements) do not. This is because expressions are evaluated to a value at runtime, whereas `if` statements are purely a control-flow construct and do not carry values. There are also data flow nodes that do not correspond to AST nodes at all.

Edges in the data flow graph represent the way data flows between program elements. For example, in the expression `x || y` there are data flow nodes corresponding to the sub-expressions `x` and `y`, as well as a data flow node corresponding to the entire expression `x || y`. There is an edge from the node corresponding to `x` to the node corresponding to `x`

`|| y`, representing the fact that data may flow from `x` to `x || y` (since the expression `x || y` may evaluate to `x`). Similarly, there is an edge from the node corresponding to `y` to the node corresponding to `x || y`.

Local and global data flow differ in which edges they consider: local data flow only considers edges between data flow nodes belonging to the same function and ignores data flow between functions and through object properties. Global data flow, however, considers the latter as well. Taint tracking introduces additional edges into the data flow graph that do not precisely correspond to the flow of values, but model whether some value at runtime may be derived from another, for instance through a string manipulating operation.

The data flow graph is computed using *classes* to model the program elements that represent the graph's nodes. The flow of data between the nodes is modeled using *predicates* to compute the graph's edges.

Computing an accurate and complete data flow graph presents several challenges:

- It isn't possible to compute data flow through standard library functions, where the source code is unavailable.
- Some behavior isn't determined until run time, which means that the data flow library must take extra steps to find potential call targets.
- Aliasing between variables can result in a single write changing the value that multiple pointers point to.
- The data flow graph can be very large and slow to compute.

To overcome these potential problems, two kinds of data flow are modeled in the libraries:

- Local data flow, concerning the data flow within a single function. When reasoning about local data flow, you only consider edges between data flow nodes belonging to the same function. It is generally sufficiently fast, efficient and precise for many queries, and it is usually possible to compute the local data flow for all functions in a CodeQL database.
- Global data flow, effectively considers the data flow within an entire program, by calculating data flow between functions and through object properties. Computing global data flow is typically more time and energy intensive than local data flow, therefore queries should be refined to look for more specific sources and sinks.

Many CodeQL queries contain examples of both local and global data flow analysis. For more information, see CodeQL query help.

Normal data flow vs taint tracking

In the standard libraries, we make a distinction between 'normal' data flow and taint tracking. The normal data flow libraries are used to analyze the information flow in which data values are preserved at each step.

For example, if you are tracking an insecure object `x` (which might be some untrusted or potentially malicious data), a step in the program may 'change' its value. So, in a simple process such as `y = x + 1`, a normal data flow analysis will highlight the use of `x`, but not `y`. However, since `y` is derived from `x`, it is influenced by the untrusted or 'tainted' information, and therefore it is also tainted. Analyzing the flow of the taint from `x` to `y` is known as taint tracking.

In QL, taint tracking extends data flow analysis by including steps in which the data values are not necessarily preserved, but the potentially insecure object is still propagated. These flow steps are modeled in the taint-tracking library using predicates that hold if taint is propagated between nodes.

Further reading

- "*Exploring data flow with path queries*"

4.1.7 Creating path queries

You can create path queries to visualize the flow of information through a codebase.

Overview

Security researchers are particularly interested in the way that information flows in a program. Many vulnerabilities are caused by seemingly benign data flowing to unexpected locations, and being used in a malicious way. Path queries written with CodeQL are particularly useful for analyzing data flow as they can be used to track the path taken by a variable from its possible starting points (`source`) to its possible end points (`sink`). To model paths, your query must provide information about the `source` and the `sink`, as well as the data flow steps that link them.

This topic provides information on how to structure a path query file so you can explore the paths associated with the results of data flow analysis.

> Note
>
> The alerts generated by path queries are displayed by default in LGTM and included in the results generated using the *CodeQL CLI*. You can also view the path explanations generated by your path query directly in LGTM or in the CodeQL *extension for VS Code*.

To learn more about modeling data flow with CodeQL, see "*About data flow analysis*." For more language-specific information on analyzing data flow, see:

- "*Analyzing data flow in C/C++*"
- "*Analyzing data flow in C#*"
- "*Analyzing data flow in Java*"
- "*Analyzing data flow in JavaScript/TypeScript*"
- "*Analyzing data flow in Python*"

Path query examples

The easiest way to get started writing your own path query is to modify one of the existing queries. For more information, see the CodeQL query help.

The Security Lab researchers have used path queries to find security vulnerabilities in various open source projects. To see articles describing how these queries were written, as well as other posts describing other aspects of security research such as exploiting vulnerabilities, see the GitHub Security Lab website.

Constructing a path query

Path queries require certain metadata, query predicates, and `select` statement structures. Many of the built-in path queries included in CodeQL follow a simple structure, which depends on how the language you are analyzing is modeled with CodeQL.

You should use the following template:

```
/**
 * ...
 * @kind path-problem
 * ...
 */

import <language>
// For some languages (Java/C++/Python) you need to explicitly import the data flow␣
↪library, such as
// import semmle.code.java.dataflow.DataFlow
import DataFlow::PathGraph
...

from MyConfiguration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "<message>"
```

Where:

- `DataFlow::Pathgraph` is the path graph module you need to import from the standard CodeQL libraries.
- `source` and `sink` are nodes on the path graph, and `DataFlow::PathNode` is their type.
- `MyConfiguration` is a class containing the predicates which define how data may flow between the `source` and the `sink`.

The following sections describe the main requirements for a valid path query.

Path query metadata

Path query metadata must contain the property `@kind path-problem`–this ensures that query results are interpreted and displayed correctly. The other metadata requirements depend on how you intend to run the query. For more information, see "*Metadata for CodeQL queries*."

Generating path explanations

In order to generate path explanations, your query needs to compute a path graph. To do this you need to define a *query predicate* called `edges` in your query. This predicate defines the edge relations of the graph you are computing, and it is used to compute the paths related to each result that your query generates. You can import a predefined `edges` predicate from a path graph module in one of the standard data flow libraries. In addition to the path graph module, the data flow libraries contain the other `classes`, `predicates`, and `modules` that are commonly used in data flow analysis.

```
import DataFlow::PathGraph
```

This statement imports the `PathGraph` module from the data flow library (`DataFlow.qll`), in which `edges` is defined.

You can also import libraries specifically designed to implement data flow analysis in various common frameworks and environments, and many additional libraries are included with CodeQL. To see examples of the different libraries used in data flow analysis, see the links to the built-in queries above or browse the standard libraries.

For all languages, you can also optionally define a `nodes` query predicate, which specifies the nodes of the path graph that you are interested in. If `nodes` is defined, only edges with endpoints defined by these nodes are selected. If `nodes` is not defined, you select all possible endpoints of `edges`.

Defining your own `edges` predicate

You can also define your own `edges` predicate in the body of your query. It should take the following form:

```
query predicate edges(PathNode a, PathNode b) {
/** Logical conditions which hold if `(a,b)` is an edge in the data flow graph */
}
```

For more examples of how to define an `edges` predicate, visit the standard CodeQL libraries and search for `edges`.

Declaring sources and sinks

You must provide information about the `source` and `sink` in your path query. These are objects that correspond to the nodes of the paths that you are exploring. The name and the type of the `source` and the `sink` must be declared in the `from` statement of the query, and the types must be compatible with the nodes of the graph computed by the `edges` predicate.

If you are querying C/C++, C#, Java, JavaScript, Python, or Ruby code (and you have used `import DataFlow::PathGraph` in your query), the definitions of the `source` and `sink` are accessed via the `Configuration` class in the data flow library. You should declare all three of these objects in the `from` statement. For example:

```
from Configuration config, DataFlow::PathNode source, DataFlow::PathNode sink
```

The configuration class is accessed by importing the data flow library. This class contains the predicates which define how data flow is treated in the query:

- `isSource()` defines where data may flow from.
- `isSink()` defines where data may flow to.

For more information on using the configuration class in your analysis see the sections on global data flow in "*Analyzing data flow in C/C++*," "*Analyzing data flow in C#*," and "*Analyzing data flow in Python*."

You can also create a configuration for different frameworks and environments by extending the `Configuration` class. For more information, see "*Types*" in the QL language reference.

Defining flow conditions

The `where` clause defines the logical conditions to apply to the variables declared in the `from` clause to generate your results. This clause can use *aggregations*, *predicates*, and logical *formulas* to limit the variables of interest to a smaller set which meet the defined conditions.

When writing a path queries, you would typically include a predicate that holds only if data flows from the `source` to the `sink`.

You can use the `hasFlowPath` predicate to specify flow from the `source` to the `sink` for a given `Configuration`:

```
where config.hasFlowPath(source, sink)
```

Select clause

Select clauses for path queries consist of four 'columns', with the following structure:

```
select element, source, sink, string
```

The `element` and `string` columns represent the location of the alert and the alert message respectively, as explained in "*About CodeQL queries*." The second and third columns, `source` and `sink`, are nodes on the path graph selected by the query. Each result generated by your query is displayed at a single location in the same way as an alert query. Additionally, each result also has an associated path, which can be viewed in LGTM or in the *CodeQL extension for VS Code*.

The `element` that you select in the first column depends on the purpose of the query and the type of issue that it is designed to find. This is particularly important for security issues. For example, if you believe the `source` value to be globally invalid or malicious it may be best to display the alert at the `source`. In contrast, you should consider displaying the alert at the `sink` if you believe it is the element that requires sanitization.

The alert message defined in the final column in the `select` statement can be developed to give more detail about the alert or path found by the query using links and placeholders. For more information, see "*Defining the results of a query*."

Further reading

- "*Exploring data flow with path queries*"
- CodeQL repository

4.1.8 Troubleshooting query performance

Improve the performance of your CodeQL queries by following a few simple guidelines.

About query performance

This topic offers some simple tips on how to avoid common problems that can affect the performance of your queries. Before reading the tips below, it is worth reiterating a few important points about CodeQL and the QL language:

- CodeQL *predicates* and *classes* are evaluated to database tables. Large predicates generate large tables with many rows, and are therefore expensive to compute.
- The QL language is implemented using standard database operations and relational algebra (such as join, projection, and union). For more information about query languages and databases, see "*About the QL language*."
- Queries are evaluated *bottom-up*, which means that a predicate is not evaluated until *all* of the predicates that it depends on are evaluated. For more information on query evaluation, see "*Evaluation of QL programs*."

Performance tips

Follow the guidelines below to ensure that you don't get tripped up by the most common CodeQL performance pitfalls.

Eliminate cartesian products

The performance of a predicate can often be judged by considering roughly how many results it has. One way of creating badly performing predicates is by using two variables without relating them in any way, or only relating them using a negation. This leads to computing the Cartesian product between the sets of possible values for each variable, potentially generating a huge table of results. This can occur if you don't specify restrictions on your variables. For instance, consider the following predicate that checks whether a Java method `m` may access a field `f`:

```
predicate mayAccess(Method m, Field f) {
  f.getAnAccess().getEnclosingCallable() = m
  or
  not exists(m.getBody())
}
```

The predicate holds if `m` contains an access to `f`, but also conservatively assumes that methods without bodies (for example, native methods) may access *any* field.

However, if `m` is a native method, the table computed by `mayAccess` will contain a row `m, f` for *all* fields `f` in the codebase, making it potentially very large.

This example shows a similar mistake in a member predicate:

```
class Foo extends Class {
  ...
  // BAD! Does not use 'this'
  Method getToString() {
    result.getName() = "ToString"
  }
  ...
}
```

Note that while `getToString()` does not declare any parameters, it has two implicit parameters, `result` and `this`, which it fails to relate. Therefore, the table computed by `getToString()` contains a row for every combination of `result` and `this`. That is, a row for every combination of a method named `"ToString"` and an instance of `Foo`. To avoid making this mistake, `this` should be restricted in the member predicate `getToString()` on the class `Foo`.

Use specific types

"*Types*" provide an upper bound on the size of a relation. This helps the query optimizer be more effective, so it's generally good to use the most specific types possible. For example:

```
predicate foo(LoggingCall e)
```

is preferred over:

```
predicate foo(Expr e)
```

From the type context, the query optimizer deduces that some parts of the program are redundant and removes them, or *specializes* them.

Determine the most specific types of a variable

If you are unfamiliar with the library used in a query, you can use CodeQL to determine what types an entity has. There is a predicate called `getAQlClass()`, which returns the most specific QL types of the entity that it is called on.

For example, if you were working with a Java database, you might use `getAQlClass()` on every `Expr` in a callable called `c`:

```
import java

from Expr e, Callable c
where
    c.getDeclaringType().hasQualifiedName("my.namespace.name", "MyClass")
    and c.getName() = "c"
    and e.getEnclosingCallable() = c
select e, e.getAQlClass()
```

The result of this query is a list of the most specific types of every `Expr` in that function. You will see multiple results for expressions that are represented by more than one type, so it will likely return a very large table of results.

Use `getAQlClass()` as a debugging tool, but don't include it in the final version of your query, as it slows down performance.

Avoid complex recursion

"*Recursion*" is about self-referencing definitions. It can be extremely powerful as long as it is used appropriately. On the whole, you should try to make recursive predicates as simple as possible. That is, you should define a *base case* that allows the predicate to *bottom out*, along with a single *recursive call*:

```
int depth(Stmt s) {
  exists(Callable c | c.getBody() = s | result = 0) // base case
  or
  result = depth(s.getParent()) + 1 // recursive call
}
```

> Note
>
> The query optimizer has special data structures for dealing with *transitive closures*. If possible, use a transitive closure over a simple recursive predicate, as it is likely to be computed faster.

Fold predicates

Sometimes you can assist the query optimizer by "folding" parts of large predicates out into smaller predicates.

The general principle is to split off chunks of work that are:

- **linear**, so that there is not too much branching.
- **tightly bound**, so that the chunks join with each other on as many variables as possible.

In the following example, we explore some lookups on two `Elements`:

```
predicate similar(Element e1, Element e2) {
  e1.getName() = e2.getName() and
  e1.getFile() = e2.getFile() and
```

(continues on next page)

(continued from previous page)

```
  e1.getLocation().getStartLine() = e2.getLocation().getStartLine()
}
```

Going from `Element -> File` and `Element -> Location -> StartLine` is linear–that is, there is only one `File`, `Location`, etc. for each `Element`.

However, as written it is difficult for the optimizer to pick out the best ordering. Joining first and then doing the linear lookups later would likely result in poor performance. Generally, we want to do the quick, linear parts first, and then join on the resultant larger tables. We can initiate this kind of ordering by splitting the above predicate as follows:

```
predicate locInfo(Element e, string name, File f, int startLine) {
  name = e.getName() and
  f = e.getFile() and
  startLine = e.getLocation().getStartLine()
}

predicate sameLoc(Element e1, Element e2) {
  exists(string name, File f, int startLine |
    locInfo(e1, name, f, startLine) and
    locInfo(e2, name, f, startLine)
  )
}
```

Now the structure we want is clearer. We've separated out the easy part into its own predicate `locInfo`, and the main predicate `sameLoc` is just a larger join.

Further reading

- "*QL language reference*"
- "*CodeQL tools*"

4.1.9 Debugging data-flow queries using partial flow

If a data-flow query doesn't produce the results you expect to see, you can use partial flow to debug the problem.

In CodeQL, you can use *data flow analysis* to compute the possible values that a variable can hold at various points in a program. A typical data-flow query looks like this:

```
class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "MyConfig" }

  override predicate isSource(DataFlow::Node node) { node instanceof MySource }

  override predicate isSink(DataFlow::Node node) { node instanceof MySink }
}

from MyConfig config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "Sink is reached from $@.", source.getNode(), "here"
```

The same query can be slightly simplified by rewriting it without *path explanations*:

```
from MyConfig config, DataFlow::Node source, DataFlow::Node sink
where config.hasPath(source, sink)
select sink, "Sink is reached from $@.", source.getNode(), "here"
```

If a data-flow query that you have written doesn't produce the results you expect it to, there may be a problem with your query. You can try to debug the potential problem by following the steps described below.

Checking sources and sinks

Initially, you should make sure that the source and sink definitions contain what you expect. If either the source or sink is empty then there can never be any data flow. The easiest way to check this is using quick evaluation in CodeQL for VS Code. Select the text `node instanceof MySource`, right-click, and choose "CodeQL: Quick Evaluation". This will evaluate the highlighted text, which in this case means the set of sources. For more information, see *Analyzing your projects* in the CodeQL for VS Code help.

If both source and sink definitions look good then we will need to look for missing flow steps.

`fieldFlowBranchLimit`

Data-flow configurations contain a parameter called `fieldFlowBranchLimit`. If the value is set too high, you may experience performance degradation, but if it's too low you may miss results. When debugging data flow try setting `fieldFlowBranchLimit` to a high value and see whether your query generates more results. For example, try adding the following to your configuration:

```
override int fieldFlowBranchLimit() { result = 5000 }
```

If there are still no results and performance is still useable, then it is best to leave this set to a high value while doing further debugging.

Partial flow

A naive next step could be to change the sink definition to `any()`. This would mean that we would get a lot of flow to all the places that are reachable from the sources. While this approach may work in some cases, you might find that it produces so many results that it's very hard to explore the findings. It can also dramatically affect query performance. More importantly, you might not even see all the partial flow paths. This is because the data-flow library tries very hard to prune impossible paths and, since field stores and reads must be evenly matched along a path, we will never see paths going through a store that fail to reach a corresponding read. This can make it hard to see where flow actually stops.

To avoid these problems, a data-flow `Configuration` comes with a mechanism for exploring partial flow that tries to deal with these caveats. This is the `Configuration.hasPartialFlow` predicate:

```
/**
 * Holds if there is a partial data flow path from `source` to `node`. The
 * approximate distance between `node` and the closest source is `dist` and
 * is restricted to be less than or equal to `explorationLimit()`. This
 * predicate completely disregards sink definitions.
 *
 * This predicate is intended for dataflow exploration and debugging and may
 * perform poorly if the number of sources is too big and/or the exploration
 * limit is set too high without using barriers.
 *
```

(continues on next page)

(continued from previous page)

```
 * This predicate is disabled (has no results) by default. Override
 * `explorationLimit()` with a suitable number to enable this predicate.
 *
 * To use this in a `path-problem` query, import the module `PartialPathGraph`.
 */
final predicate hasPartialFlow(PartialPathNode source, PartialPathNode node, int dist) {
```

There is also a `Configuration.hasPartialFlowRev` for exploring flow backwards from a sink.

As noted in the documentation for `hasPartialFlow` (for example, in the CodeQL for Java documentation) you must first enable this by adding an override of `explorationLimit`. For example:

```
override int explorationLimit() { result = 5 }
```

This defines the exploration radius within which `hasPartialFlow` returns results.

To get good performance when using `hasPartialFlow` it is important to ensure the `isSink` predicate of the configuration has no results. Likewise, when using `hasPartialFlowRev` the `isSource` predicate of the configuration should have no results.

It is also useful to focus on a single source at a time as the starting point for the flow exploration. This is most easily done by adding a temporary restriction in the `isSource` predicate.

To do quick evaluations of partial flow it is often easiest to add a predicate to the query that is solely intended for quick evaluation (right-click the predicate name and choose "CodeQL: Quick Evaluation"). A good starting point is something like:

```
predicate adhocPartialFlow(Callable c, PartialPathNode n, Node src, int dist) {
  exists(MyConfig conf, PartialPathNode source |
    conf.hasPartialFlow(source, n, dist) and
    src = source.getNode() and
    c = n.getNode().getEnclosingCallable()
  )
}
```

If you are focusing on a single source then the `src` column is superfluous. You may of course also add other columns of interest based on `n`, but including the enclosing callable and the distance to the source at the very least is generally recommended, as they can be useful columns to sort on to better inspect the results.

If you see a large number of partial flow results, you can focus them in a couple of ways:

- If flow travels a long distance following an expected path, that can result in a lot of uninteresting flow being included in the exploration radius. To reduce the amount of uninteresting flow, you can replace the source definition with a suitable `node` that appears along the path and restart the partial flow exploration from that point.
- Creative use of barriers and sanitizers can be used to cut off flow paths that are uninteresting. This also reduces the number of partial flow results to explore while debugging.

Further reading

- *About data flow analysis*
- *Creating path queries*
- *About CodeQL queries*: CodeQL queries are used to analyze code for issues related to security, correctness, maintainability, and readability.
- *Metadata for CodeQL queries*: Metadata tells users important information about CodeQL queries. You must include the correct query metadata in a query to be able to view query results in source code.
- *Query help files*: Query help files tell users the purpose of a query, and recommend how to solve the potential problem the query finds.
- *Defining the results of a query*: You can control how analysis results are displayed in source code by modifying a query's `select` statement.
- *Providing locations in CodeQL queries*: CodeQL includes mechanisms for extracting the location of elements in a codebase. Use these mechanisms when writing custom CodeQL queries and libraries to help display information to users.
- *About data flow analysis*: Data flow analysis is used to compute the possible values that a variable can hold at various points in a program, determining how those values propagate through the program and where they are used.
- *Creating path queries*: You can create path queries to visualize the flow of information through a codebase.
- *Troubleshooting query performance*: Improve the performance of your CodeQL queries by following a few simple guidelines.
- *Debugging data-flow queries using partial flow*: If a data-flow query doesn't produce the results you expect to see, you can use partial flow to debug the problem..

4.2 QL tutorials

Solve puzzles to learn the basics of QL before you analyze code with CodeQL. The tutorials teach you how to write queries and introduce you to key logic concepts along the way.

4.2.1 Introduction to QL

Work through some simple exercises and examples to learn about the basics of QL and CodeQL.

Basic syntax

The basic syntax of QL will look familiar to anyone who has used SQL, but it is used somewhat differently.

QL is a logic programming language, so it is built up of logical formulas. QL uses common logical connectives (such as `and`, `or`, and `not`), quantifiers (such as `forall` and `exists`), and other important logical concepts such as predicates.

QL also supports recursion and aggregates. This allows you to write complex recursive queries using simple QL syntax and directly use aggregates such as `count`, `sum`, and `average`.

Running a query

You can try out the following examples and exercises using *CodeQL for VS Code*, or you can run them in the query console on LGTM.com. Before you can run a query on LGTM.com, you need to select a language and project to query (for these logic examples, any language and project will do).

Once you have selected a language, the query console is populated with the query:

```
import <language>

select "hello world"
```

This query returns the string `"hello world"`.

More complicated queries typically look like this:

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

For example, the result of this query is the number 42:

```
from int x, int y
where x = 6 and y = 7
select x * y
```

Note that `int` specifies that the **type** of `x` and `y` is 'integer'. This means that `x` and `y` are restricted to integer values. Some other common types are: `boolean` (`true` or `false`), `date`, `float`, and `string`.

Simple exercises

You can write simple queries using the some of the basic functions that are available for the `int`, `date`, `float`, `boolean` and `string` types. To apply a function, append it to the argument. For example, `1.toString()` converts the value `1` to a string. Notice that as you start typing a function, a pop-up is displayed making it easy to select the function that you want. Also note that you can apply multiple functions in succession. For example, `100.log().sqrt()` first takes the natural logarithm of 100 and then computes the square root of the result.

Exercise 1

Write a query which returns the length of the string `"lgtm"`. (Hint: here is the list of the functions that can be applied to strings.)

See answer in the query console on LGTM.com

There is often more than one way to define a query. For example, we can also write the above query in the shorter form:

```
select "lgtm".length()
```

Exercise 2

Write a query which returns the sine of the minimum of `3^5` (3 raised to the power `5`) and `245.6`.

See answer in the query console on LGTM.com

Exercise 3

Write a query which returns the opposite of the boolean `false`.

See answer in the query console on LGTM.com

Exercise 4

Write a query which computes the number of days between June 10 and September 28, 2017.

See answer in the query console on LGTM.com

Example query with multiple results

The exercises above all show queries with exactly one result, but in fact many queries have multiple results. For example, the following query computes all Pythagorean triples between 1 and 10:

```
from int x, int y, int z
where x in [1..10] and y in [1..10] and z in [1..10] and
      x*x + y*y = z*z
select x, y, z
```

See this in the query console on LGTM.com

To simplify the query, we can introduce a class `SmallInt` representing the integers between 1 and 10. We can also define a predicate `square()` on integers in that class. Defining classes and predicates in this way makes it easy to reuse code without having to repeat it every time.

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

from SmallInt x, SmallInt y, SmallInt z
where x.square() + y.square() = z.square()
select x, y, z
```

See this in the query console on LGTM.com

Example CodeQL queries

The previous examples used the primitive types built in to QL. Although we chose a project to query, we didn't use the information in that project's database. The following example queries *do* use these databases and give you an idea of how to use CodeQL to analyze projects.

Queries using the CodeQL libraries can find errors and uncover variants of important security vulnerabilities in codebases. Visit GitHub Security Lab to read about examples of vulnerabilities that we have recently found in open source projects.

To import the CodeQL library for a specific programming language, type `import <language>` at the start of the query.

```
import python

from Function f
where count(f.getAnArg()) > 7
select f
```

See this in the query console on LGTM.com. The `from` clause defines a variable `f` representing a Python function. The `where` part limits the functions `f` to those with more than 7 arguments. Finally, the `select` clause lists these functions.

```
import javascript

from Comment c
where c.getText().regexpMatch("(?si).*\\bTODO\\b.*")
select c
```

See this in the query console on LGTM.com. The `from` clause defines a variable `c` representing a JavaScript comment. The `where` part limits the comments `c` to those containing the word `"TODO"`. The `select` clause lists these comments.

```
import java

from Parameter p
where not exists(p.getAnAccess())
select p
```

See this in the query console on LGTM.com. The `from` clause defines a variable `p` representing a Java parameter. The `where` clause finds unused parameters by limiting the parameters `p` to those which are not accessed. Finally, the `select` clause lists these parameters.

Further reading

- For a more technical description of the underlying language, see the "*QL language reference*."

4.2.2 Find the thief

Take on the role of a detective to find the thief in this fictional village. You will learn how to use logical connectives, quantifiers, and aggregates in QL along the way.

Introduction

There is a small village hidden away in the mountains. The village is divided into four parts—north, south, east, and west—and in the center stands a dark and mysterious castle… Inside the castle, locked away in the highest tower, lies the king's valuable golden crown. One night, a terrible crime is committed. A thief breaks into the tower and steals the crown!

You know that the thief must live in the village, since nobody else knew about the crown. After some expert detective work, you obtain a list of all the people in the village and some of their personal details.

Name	Age	Hair color	Height	Location
…	…	…	…	…

Sadly, you still have no idea who could have stolen the crown so you walk around the village to find clues. The villagers act very suspiciously and you are convinced they have information about the thief. They refuse to share their knowledge with you directly, but they reluctantly agree to answer questions. They are still not very talkative and **only answer questions with 'yes' or 'no'**.

You start asking some creative questions and making notes of the answers so you can compare them with your information later:

	Question	Answer
(1)	Is the thief taller than 150 cm?	yes
(2)	Does the thief have blond hair?	no
(3)	Is the thief bald?	no
(4)	Is the thief younger than 30?	no
(5)	Does the thief live east of the castle?	yes
(6)	Does the thief have black or brown hair?	yes
(7)	Is the thief taller than 180cm and shorter than 190cm?	no
(8)	Is the thief the oldest person in the village?	no
(9)	Is the thief the tallest person in the village?	no
(10)	Is the thief shorter than the average villager?	yes
(11)	Is the thief the oldest person in the eastern part of the village?	yes

There is too much information to search through by hand, so you decide to use your newly acquired QL skills to help you with your investigation…

1. Open the query console on LGTM.com to get started.
2. Select a language and a demo project. For this tutorial, any language and project will do.
3. Delete the default code `import <language> select "hello world"`.

QL libraries

We've defined a number of QL *predicates* to help you extract data from your table. A QL predicate is a mini-query that expresses a relation between various pieces of data and describes some of their properties. In this case, the predicates give you information about a person, for example their height or age.

Predicate	Description
`getAge()`	returns the age of the person (in years) as an `int`
`getHairColor()`	returns the hair color of the person as a `string`
`getHeight()`	returns the height of the person (in cm) as a `float`
`getLocation()`	returns the location of the person's home (north, south, east or west) as a `string`

We've stored these predicates in the QL library `tutorial.qll`. To access this library, type `import tutorial` in the query console.

Libraries are convenient for storing commonly used predicates. This saves you from defining a predicate every time you need it. Instead you can just `import` the library and use the predicate directly. Once you have imported the library, you can apply any of these predicates to an expression by appending it.

For example, `t.getHeight()` applies `getHeight()` to `t` and returns the height of `t`.

Start the search

The villagers answered "yes" to the question "Is the thief taller than 150cm?" To use this information, you can write the following query to list all villagers taller than 150cm. These are all possible suspects.

```
from Person t
where t.getHeight() > 150
select t
```

The first line, `from Person t`, declares that `t` must be a `Person`. We say that the *type* of `t` is `Person`.

Before you use the rest of your answers in your QL search, here are some more tools and examples to help you write your own QL queries:

Logical connectives

Using *logical connectives*, you can write more complex queries that combine different pieces of information.

For example, if you know that the thief is older than 30 *and* has brown hair, you can use the following `where` clause to link two predicates:

```
where t.getAge() > 30 and t.getHairColor() = "brown"
```

> Note
>
> The predicate `getHairColor()` returns a `string`, so we need to include quotation marks around the result `"brown"`.

If the thief does *not* live north of the castle, you can use:

```
where not t.getLocation() = "north"
```

If the thief has brown hair *or* black hair, you can use:

```
where t.getHairColor() = "brown" or t.getHairColor() = "black"
```

You can also combine these connectives into longer statements:

```
where t.getAge() > 30
  and (t.getHairColor() = "brown" or t.getHairColor() = "black")
  and not t.getLocation() = "north"
```

> Note
>
> We've placed parentheses around the `or` clause to make sure that the query is evaluated as intended. Without parentheses, the connective `and` takes precedence over `or`.

Predicates don't always return exactly one value. For example, if a person `p` has black hair which is turning gray, `p.getHairColor()` will return two values: black and gray.

What if the thief is bald? In that case, the thief has no hair, so the `getHairColor()` predicate simply doesn't return any results!

If you know that the thief definitely isn't bald, then there must be a color that matches the thief's hair color. One way to express this in QL is to introduce a new variable `c` of type `string` and select those `t` where `t.getHairColor()` matches a value of `c`.

```
from Person t, string c
where t.getHairColor() = c
select t
```

Notice that we have only temporarily introduced the variable `c` and we didn't need it at all in the `select` clause. In this case, it is better to use `exists`:

```
from Person t
where exists(string c | t.getHairColor() = c)
select t
```

`exists` introduces a temporary variable `c` of type `string` and holds only if there is at least one `string c` that satisfies `t.getHairColor() = c`.

> Note
>
> If you are familiar with logic, you may notice that `exists` in QL corresponds to the existential *quantifier* in logic. QL also has a universal quantifier `forall(vars | formula 1 | formula 2)` which is logically equivalent to `not exists(vars | formula 1 | not formula 2)`.

The real investigation

You are now ready to track down the thief! Using the examples above, write a query to find the people who satisfy the answers to the first eight questions:

	Question	Answer
1	Is the thief taller than 150 cm?	yes
2	Does the thief have blond hair?	no
3	Is the thief bald?	no
4	Is the thief younger than 30?	no
5	Does the thief live east of the castle?	yes
6	Does the thief have black or brown hair?	yes
7	Is the thief taller than 180cm and shorter than 190cm?	no
8	Is the thief the oldest person in the village?	no

Hints

1. Don't forget to `import tutorial`!
2. Translate each question into QL separately. Look at the examples above if you get stuck.
3. For question 3, remember that a bald person does not have a hair color.
4. For question 8, note that if a person is *not* the oldest, then there is at least one person who is older than them.
5. Combine the conditions using logical connectives to get a query of the form:

```
import tutorial

from Person t
where <condition 1> and
  not <condition 2> and
  ...
select t
```

Once you have finished, you will have a list of possible suspects. One of those people must be the thief!

See the answer in the query console on LGTM.com

> Note
>
> In the answer, we used `/*` and `*/` to label the different parts of the query. Any text surrounded by `/*` and `*/` is not evaluated as part of the QL code, but is just a *comment*.

You are getting closer to solving the mystery! Unfortunately, you still have quite a long list of suspects… To find out which of your suspects is the thief, you must gather more information and refine your query in the next step.

More advanced queries

What if you want to find the oldest, youngest, tallest, or shortest person in the village? As mentioned in the previous topic, you can do this using `exists`. However, there is also a more efficient way to do this in QL using functions like `max` and `min`. These are examples of *aggregates*.

In general, an aggregate is a function that performs an operation on multiple pieces of data and returns a single value as its output. Common aggregates are `count`, `max`, `min`, `avg` (average) and `sum`. The general way to use an aggregate is:

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

For example, you can use the `max` aggregate to find the age of the oldest person in the village:

```
max(int i | exists(Person p | p.getAge() = i) | i)
```

This aggregate considers all integers `i`, limits `i` to values that match the ages of people in the village, and then returns the largest matching integer.

But how can you use this in an actual query?

If the thief is the oldest person in the village, then you know that the thief's age is equal to the maximum age of the villagers:

```
from Person t
where t.getAge() = max(int i | exists(Person p | p.getAge() = i) | i)
select t
```

This general aggregate syntax is quite long and inconvenient. In most cases, you can omit certain parts of the aggregate. A particularly helpful QL feature is *ordered aggregation*. This allows you to order the expression using `order by`.

For example, selecting the oldest villager becomes much simpler if you use an ordered aggregate.

```
select max(Person p | | p order by p.getAge())
```

The ordered aggregate considers every person `p` and selects the person with the maximum age. In this case, there are no restrictions on what people to consider, so the `<logical formula>` clause is empty. Note that if there are several people with the same maximum age, the query lists all of them.

Here are some more examples of aggregates:

Example	Result
`min(Person p | p.getLocation() = "east" | p order by p.getHeight())`	shortest person in the east of the village
`count(Person p | p.getLocation() = "south" | p)`	number of people in the south of the village
`avg(Person p | | p.getHeight())`	average height of the villagers
`sum(Person p | p.getHairColor() = "brown" | p.getAge())`	combined age of all the villagers with brown hair

Capture the culprit

You can now translate the remaining questions into QL:

	Question	Answer
…	…	…
9	Is the thief the tallest person in the village?	no
10	Is the thief shorter than the average villager?	yes
11	Is the thief the oldest person in the eastern part of the village?	yes

Have you found the thief?

See the answer in the query console on LGTM.com

Further reading

- "*QL language reference*"
- "*CodeQL tools*"

4.2.3 Catch the fire starter

Learn about QL predicates and classes to solve your second mystery as a QL detective.

Just as you've successfully found the thief and returned the golden crown to the castle, another terrible crime is committed. Early in the morning, a few people start a fire in a field in the north of the village and destroy all the crops!

You now have the reputation of being an expert QL detective, so you are once again asked to find the culprits.

This time, you have some additional information. There is a strong rivalry between the north and south of the village and you know that the criminals live in the south.

Read the examples below to learn how to define predicates and classes in QL. These make the logic of your queries easier to understand and will help simplify your detective work.

Select the southerners

This time you only need to consider a specific group of villagers, namely those living in the south of the village. Instead of writing `getLocation() = "south"` in all your queries, you could define a new *predicate* `isSouthern`:

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

The predicate `isSouthern(p)` takes a single parameter `p` and checks if `p` satisfies the property `p.getLocation() = "south"`.

Note

- The name of a predicate always starts with a lowercase letter.
- You can also define predicates with a result. In that case, the keyword `predicate` is replaced with the type of the result. This is like introducing a new argument, the special variable `result`. For example, `int getAge() { result = ... }` returns an `int`.

You can now list all southerners using:

```
/* define predicate `isSouthern` as above */

from Person p
where isSouthern(p)
select p
```

This is already a nice way to simplify the logic, but we could be more efficient. Currently, the query looks at every `Person p`, and then restricts to those who satisfy `isSouthern(p)`. Instead, we could define a new *class* `Southerner` containing precisely the people we want to consider.

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

A class in QL represents a logical property: when a value satisfies that property, it is a member of the class. This means that a value can be in many classes—being in a particular class doesn't stop it from being in other classes too.

The expression `isSouthern(this)` defines the logical property represented by the class, called its *characteristic predicate*. It uses a special variable `this` and indicates that a `Person` "`this`" is a `Southerner` if the property `isSouthern(this)` holds.

Note

If you are familiar with object-oriented programming languages, you might be tempted to think of the characteristic predicate as a *constructor*. However, this is **not** the case—it is a logical property which does not create any objects.

You always need to define a class in QL in terms of an existing (larger) class. In our example, a `Southerner` is a special kind of `Person`, so we say that `Southerner` *extends* ("is a subset of") `Person`.

Using this class you can now list all people living in the south simply as:

```
from Southerner s
select s
```

You may have noticed that some predicates are appended, for example `p.getAge()`, while others are not, for example `isSouthern(p)`. This is because `getAge()` is a member predicate, that is, a predicate that only applies to members of a class. You define such a member predicate inside a class. In this case, `getAge()` is defined inside the class `Person`. In contrast, `isSouthern` is defined separately and is not inside any classes. Member predicates are especially useful because you can chain them together easily. For example, `p.getAge().sqrt()` first gets the age of `p` and then calculates the square root of that number.

Travel restrictions

Another factor you want to consider is the travel restrictions imposed following the theft of the crown. Originally there were no restrictions on where villagers could travel within the village. Consequently the predicate `isAllowedIn(string region)` held for any person and any region. The following query lists all villagers, since they could all travel to the north:

```
from Person p
where p.isAllowedIn("north")
select p
```

However, after the recent theft, the villagers have become more anxious of criminals lurking around the village and they no longer allow children under the age of 10 to travel out of their home region.

This means that `isAllowedIn(string region)` no longer holds for all people and all regions, so you should temporarily *override* the original predicate if `p` is a child.

Start by defining a class `Child` containing all villagers under 10 years old. Then you can redefine `isAllowedIn(string region)` as a member predicate of `Child` to allow children only to move within their own region. This is expressed by `region = this.getLocation()`.

```
class Child extends Person {
  /* the characteristic predicate */
  Child() { this.getAge() < 10 }

  /* a member predicate */
  override predicate isAllowedIn(string region) {
    region = this.getLocation()
  }
}
```

Now try applying `isAllowedIn(string region)` to a person `p`. If `p` is not a child, the original definition is used, but if `p` is a child, the new predicate definition overrides the original.

You know that the fire starters live in the south *and* that they must have been able to travel to the north. Write a query to find the possible suspects. You could also extend the `select` clause to list the age of the suspects. That way you can

clearly see that all the children have been excluded from the list.

See the answer in the query console on LGTM.com

You can now continue to gather more clues and find out which of your suspects started the fire…

Identify the bald bandits

You ask the northerners if they have any more information about the fire starters. Luckily, you have a witness! The farmer living next to the field saw two people run away just after the fire started. He only saw the tops of their heads, and noticed that they were both bald.

This is a very helpful clue. Remember that you wrote a QL query to select all bald people:

```
from Person p
where not exists (string c | p.getHairColor() = c)
select p
```

To avoid having to type `not exists (string c | p.getHairColor() = c)` every time you want to select a bald person, you can instead define another new predicate `isBald`.

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

The property `isBald(p)` holds whenever `p` is bald, so you can replace the previous query with:

```
from Person p
where isBald(p)
select p
```

The predicate `isBald` is defined to take a `Person`, so it can also take a `Southerner`, as `Southerner` is a subtype of `Person`. It can't take an `int` for example—that would cause an error.

You can now write a query to select the bald southerners who are allowed into the north.

See the answer in the query console on LGTM.com

You have found the two fire starters! They are arrested and the villagers are once again impressed with your work.

Further reading

- "*QL language reference*"
- "*CodeQL tools*"

4.2.4 Crown the rightful heir

This is a QL detective puzzle that shows you how to use recursion in QL to write more complex queries.

King Basil's heir

Phew! No more crimes in the village—you can finally leave the village and go home.

But then… During your last night in the village, the old king—the great King Basil—dies in his sleep and there is chaos everywhere!

The king never married and he had no children, so nobody knows who should inherit the king's castle and fortune. Immediately, lots of villagers claim that they are somehow descended from the king's family and that they are the true heir. People argue and fight and the situation seems hopeless.

Eventually you decide to stay in the village to resolve the argument and find the true heir to the throne.

You want to find out if anyone in the village is actually related to the king. This seems like a difficult task at first, but you start work confidently. You know the villagers quite well by now, and you have a list of all the parents in the village and their children.

To find out more about the king and his family, you get access to the castle and find some old family trees. You also include these relations in your database to see if anyone in the king's family is still alive.

The following predicate is useful to help you access the data:

Predicate	Description
`parentOf(Person p)`	returns a parent of `p`

For example, you can list all children `p` together with their parents:

```
from Person p
select parentOf(p) + " is a parent of " + p
```

There is too much information to search through by hand, so you write a QL query to help you find the king's heir.

We know that the king has no children himself, but perhaps he has siblings. Write a query to find out:

```
from Person p
where parentOf(p) = parentOf("King Basil") and
  not p = "King Basil"
select p
```

He does indeed have siblings! But you need to check if any of them are alive… Here is one more predicate you might need:

Predicate	Description
`isDeceased()`	holds if the person is deceased

Use this predicate to see if the any of the king's siblings are alive.

```
from Person p
where parentOf(p) = parentOf("King Basil") and
  not p = "King Basil"
  and not p.isDeceased()
select p
```

Unfortunately, none of King Basil's siblings are alive. Time to investigate further. It might be helpful to define a predicate `childOf()` which returns a child of the person. To do this, the `parentOf()` predicate can be used inside the definition of `childOf()`. Remember that someone is a child of `p` if and only if `p` is their parent:

```
Person childOf(Person p) {
  p = parentOf(result)
}
```

Note

As illustrated by the example above, you don't have to directly write `result = <expression involving p>` in the predicate definition. Instead you can also express the relation between `p` and `result` "backwards" by writing `p` in terms of `result`.

Try to write a query to find out if any of the king's siblings have children:

```
from Person p
where parentOf(p) = parentOf("King Basil") and
  not p = "King Basil"
select childOf(p)
```

The query returns no results, so they have no children. But perhaps King Basil has a cousin who is alive or has children, or a second cousin, or...

This is getting complicated. Ideally, you want to define a predicate `relativeOf(Person p)` that lists all the relatives of `p`.

How could you do that?

It helps to think of a precise definition of *relative*. A possible definition is that two people are related if they have a common ancestor.

You can introduce a predicate `ancestorOf(Person p)` that lists all ancestors of `p`. An ancestor of `p` is just a parent of `p`, or a parent of a parent of `p`, or a parent of a parent of a parent of `p`, and so on. Unfortunately, this leads to an endless list of parents. You can't write an infinite QL query, so there must be an easier approach.

Aha, you have an idea! You can say that an ancestor is either a parent, or a parent of someone you already know to be an ancestor.

You can translate this into QL as follows:

```
Person ancestorOf(Person p) {
  result = parentOf(p) or
  result = parentOf(ancestorOf(p))
}
```

As you can see, you have used the predicate `ancestorOf()` inside its own definition. This is an example of *recursion*.

This kind of recursion, where the same operation (in this case `parentOf()`) is applied multiple times, is very common in QL, and is known as the *transitive closure* of the operation. There are two special symbols + and * that are extremely useful when working with transitive closures:

- `parentOf+(p)` applies the `parentOf()` predicate to `p` one or more times. This is equivalent to `ancestorOf(p)`.
- `parentOf*(p)` applies the `parentOf()` predicate to `p` zero or more times, so it returns an ancestor of `p` or `p` itself.

Try using this new notation to define a predicate `relativeOf()` and use it to list all living relatives of the king.

Hint:

Here is one way to define `relativeOf()`:

```
Person relativeOf(Person p) {
  parentOf*(result) = parentOf*(p)
}
```

Don't forget to use the predicate `isDeceased()` to find relatives that are still alive.

See the answer in the query console on LGTM.com

Select the true heir

At the next village meeting, you announce that there are two living relatives.

To decide who should inherit the king's fortune, the villagers carefully read through the village constitution:

"The heir to the throne is the closest living relative of the king. Any person with a criminal record will not be considered. If there are multiple candidates, the oldest person is the heir."

As your final challenge, define a predicate `hasCriminalRecord` so that `hasCriminalRecord(p)` holds if `p` is any of the criminals you unmasked earlier (in the "*Find the thief*" and "*Catch the fire starter*" tutorials).

See the answer in the query console on LGTM.com

Experimental explorations

Congratulations! You have found the heir to the throne and restored peace to the village. However, you don't have to leave the villagers just yet. There are still a couple more questions about the village constitution that you could answer for the villagers, by writing QL queries:

- Which villager is next in line to the throne? Could you write a predicate to determine how closely related the remaining villagers are to the new monarch?
- How would you select the oldest candidate using a QL query, if multiple villagers have the same relationship to the monarch?

You could also try writing more of your own QL queries to find interesting facts about the villagers. You are free to investigate whatever you like, but here are some suggestions:

- What is the most common hair color in the village? And in each region?
- Which villager has the most children? Who has the most descendants?
- How many people live in each region of the village?
- Do all villagers live in the same region of the village as their parents?
- Find out whether there are any time travelers in the village! (Hint: Look for "impossible" family relations.)

Further reading

- "*QL language reference*"
- "*CodeQL tools*"

4.2.5 Cross the river

Use common QL features to write a query that finds a solution to the "River crossing" logic puzzle.

Introduction

> River crossing puzzle
>
> A man is trying to ferry a goat, a cabbage, and a wolf across a river. His boat can only take himself and at most one item as cargo. His problem is that if the goat is left alone with the cabbage, it will eat it. And if the wolf is left alone with the goat, it will eat it. How does he get everything across the river?

A solution should be a set of instructions for how to ferry the items, such as "First ferry the goat across the river, and come back with nothing. Then ferry the cabbage across, and come back with ..."

There are lots of ways to approach this problem and implement it in QL. Before you start, make sure that you are familiar with how to define *classes* and *predicates* in QL. The following walkthrough is just one of many possible implementations, so have a go at writing your own query too! To find more example queries, see the list *below*.

Walkthrough

Model the elements of the puzzle

The basic components of the puzzle are the cargo items and the shores on either side of the river. Start by modeling these as classes.

First, define a class `Cargo` containing the different cargo items. Note that the man can also travel on his own, so it helps to explicitly include `"Nothing"` as a piece of cargo.

Show/hide code

```
/** A possible cargo item. */
class Cargo extends string {
  Cargo() {
    this = "Nothing" or
    this = "Goat" or
    this = "Cabbage" or
    this = "Wolf"
  }
}
```

Second, any item can be on one of two shores. Let's call these the "left shore" and the "right shore". Define a class `Shore` containing `"Left"` and `"Right"`.

It would be helpful to express "the other shore" to model moving from one side of the river to the other. You can do this by defining a member predicate `other` in the class `Shore` such that `"Left".other()` returns `"Right"` and vice versa.

Show/hide code

```
/** One of two shores. */
class Shore extends string {
  Shore() {
    this = "Left" or
    this = "Right"
```

(continues on next page)

(continued from previous page)

```
  }

  /** Returns the other shore. */
  Shore other() {
    this = "Left" and result = "Right"
    or
    this = "Right" and result = "Left"
  }
}
```

We also want a way to keep track of where the man, the goat, the cabbage, and the wolf are at any point. We can call this combined information the "state". Define a class `State` that encodes the location of each piece of cargo. For example, if the man is on the left shore, the goat on the right shore, and the cabbage and wolf on the left shore, the state should be `Left, Right, Left, Left`.

You may find it helpful to introduce some variables that refer to the shore on which the man and the cargo items are. These temporary variables in the body of a class are called *fields*.

Show/hide code

```
/** A record of where everything is. */
class State extends string {
  Shore manShore;
  Shore goatShore;
  Shore cabbageShore;
  Shore wolfShore;

  State() { this = manShore + "," + goatShore + "," + cabbageShore + "," + wolfShore }
}
```

We are interested in two particular states, namely the initial state and the goal state, which we have to achieve to solve the puzzle. Assuming that all items start on the left shore and end up on the right shore, define `InitialState` and `GoalState` as subclasses of `State`.

Show/hide code

```
/** The initial state, where everything is on the left shore. */
class InitialState extends State {
  InitialState() { this = "Left" + "," + "Left" + "," + "Left" + "," + "Left" }
}

/** The goal state, where everything is on the right shore. */
class GoalState extends State {
  GoalState() { this = "Right" + "," + "Right" + "," + "Right" + "," + "Right" }
}
```

> Note
>
> To avoid typing out the lengthy string concatenations, you could introduce a helper predicate `renderState` that renders the state in the required form.

Using the above note, the QL code so far looks like this:

Show/hide code

```
/** A possible cargo item. */
class Cargo extends string {
  Cargo() {
    this = "Nothing" or
    this = "Goat" or
    this = "Cabbage" or
    this = "Wolf"
  }
}

/** One of two shores. */
class Shore extends string {
  Shore() {
    this = "Left" or
    this = "Right"
  }

  /** Returns the other shore. */
  Shore other() {
    this = "Left" and result = "Right"
    or
    this = "Right" and result = "Left"
  }
}

/** Renders the state as a string. */
string renderState(Shore manShore, Shore goatShore, Shore cabbageShore, Shore wolfShore)
↪{
  result = manShore + "," + goatShore + "," + cabbageShore + "," + wolfShore
}

/** A record of where everything is. */
class State extends string {
  Shore manShore;
  Shore goatShore;
  Shore cabbageShore;
  Shore wolfShore;

  State() { this = renderState(manShore, goatShore, cabbageShore, wolfShore) }
}

/** The initial state, where everything is on the left shore. */
class InitialState extends State {
  InitialState() { this = renderState("Left", "Left", "Left", "Left") }
}

/** The goal state, where everything is on the right shore. */
class GoalState extends State {
  GoalState() { this = renderState("Right", "Right", "Right", "Right") }
}
```

Model the action of "ferrying"

The basic act of ferrying moves the man and one cargo item to the other shore, resulting in a new state.

Write a member predicate (of `State`) called `ferry`, that specifies what happens to the state after ferrying a particular cargo. (Hint: Use the predicate `other`.)

Show/hide code

```
/** Returns the state that is reached after ferrying a particular cargo item. */
State ferry(Cargo cargo) {
  cargo = "Nothing" and
  result = renderState(manShore.other(), goatShore, cabbageShore, wolfShore)
  or
  cargo = "Goat" and
  result = renderState(manShore.other(), goatShore.other(), cabbageShore, wolfShore)
  or
  cargo = "Cabbage" and
  result = renderState(manShore.other(), goatShore, cabbageShore.other(), wolfShore)
  or
  cargo = "Wolf" and
  result = renderState(manShore.other(), goatShore, cabbageShore, wolfShore.other())
}
```

Of course, not all ferrying actions are possible. Add some extra conditions to describe when a ferrying action is "safe". That is, it doesn't lead to a state where the goat or the cabbage get eaten. For example, follow these steps:

1. Define a predicate `isSafe` that holds when the state itself is safe. Use this to encode the conditions for when nothing gets eaten.
2. Define a predicate `safeFerry` that restricts `ferry` to only include safe ferrying actions.

Show/hide code

```
/**
 * Holds if the state is safe. This occurs when neither the goat nor the cabbage
 * can get eaten.
 */
predicate isSafe() {
  // The goat can't eat the cabbage.
  (goatShore != cabbageShore or goatShore = manShore) and
  // The wolf can't eat the goat.
  (wolfShore != goatShore or wolfShore = manShore)
}

/** Returns the state that is reached after safely ferrying a cargo item. */
State safeFerry(Cargo cargo) { result = this.ferry(cargo) and result.isSafe() }
```

Find paths from one state to another

The main aim of this query is to find a path, that is, a list of successive ferrying actions, to get from the initial state to the goal state. You could write this "list" by separating each item by a newline (`"\n"`).

When finding the solution, you should be careful to avoid "infinite" paths. For example, the man could ferry the goat back and forth any number of times without ever reaching an unsafe state. Such a path would have an infinite number of river crossings without ever solving the puzzle.

One way to restrict our paths to a finite number of river crossings is to define a *member predicate* `State reachesVia(string path, int steps)`. The result of this predicate is any state that is reachable from the current state (`this`) via the given path in a specified finite number of steps.

You can write this as a *recursive predicate*, with the following base case and recursion step:

- If `this` *is* the result state, then it (trivially) reaches the result state via an empty path in zero steps.
- Any other state is reachable if `this` can reach an intermediate state (for some value of `path` and `steps`), and there is a `safeFerry` action from that intermediate state to the result state.

To ensure that the predicate is finite, you should restrict `steps` to a particular value, for example `steps <= 7`.

Show/hide code

```
/**
 * Returns all states that are reachable via safe ferrying.
 * `path` keeps track of how it is achieved and `steps` keeps track of the number of␣
↪steps it takes.
 */
State reachesVia(string path, int steps) {
  // Trivial case: a state is always reachable from itself
  steps = 0 and this = result and path = ""
  or
  // A state is reachable using pathSoFar and then safely ferrying cargo.
  exists(int stepsSoFar, string pathSoFar, Cargo cargo |
    result = this.reachesVia(pathSoFar, stepsSoFar).safeFerry(cargo) and
    steps = stepsSoFar + 1 and
    // We expect a solution in 7 steps, but you can choose any value here.
    steps <= 7 and
    path = pathSoFar + "\n Ferry " + cargo
  )
}
```

However, although this ensures that the solution is finite, it can still contain loops if the upper bound for `steps` is large. In other words, you could get an inefficient solution by revisiting the same state multiple times.

Instead of picking an arbitrary upper bound for the number of steps, you can avoid counting steps altogether. If you keep track of states that have already been visited and ensure that each ferrying action leads to a new state, the solution certainly won't contain any loops.

To do this, change the member predicate to `State reachesVia(string path, string visitedStates)`. The result of this predicate is any state that is reachable from the current state (`this`) via the given path without revisiting any previously visited states.

- As before, if `this` *is* the result state, then it (trivially) reaches the result state via an empty path and an empty string of visited states.
- Any other state is reachable if `this` can reach an intermediate state via some path, without revisiting any previous states, and there is a `safeFerry` action from the intermediate state to the result state. (Hint: To check whether

a state has previously been visited, you could check if there is an index of `visitedStates` at which the state occurs.)

Show/hide code

```
/**
 * Returns all states that are reachable via safe ferrying.
 * `path` keeps track of how it is achieved.
 * `visitedStates` keeps track of previously visited states and is used to avoid loops.
 */
State reachesVia(string path, string visitedStates) {
  // Trivial case: a state is always reachable from itself.
  this = result and
  visitedStates = this and
  path = ""
  or
  // A state is reachable using pathSoFar and then safely ferrying cargo.
  exists(string pathSoFar, string visitedStatesSoFar, Cargo cargo |
    result = this.reachesVia(pathSoFar, visitedStatesSoFar).safeFerry(cargo) and
    // The resulting state has not yet been visited.
    not exists(visitedStatesSoFar.indexOf(result)) and
    visitedStates = visitedStatesSoFar + "/" + result and
    path = pathSoFar + "\n Ferry " + cargo
  )
}
```

Display the results

Once you've defined all the necessary classes and predicates, write a *select clause* that returns the resulting path.

Show/hide code

```
from string path
where any(InitialState i).reachesVia(path, _) = any(GoalState g)
select path
```

The *don't-care expression* (_), as the second argument to the `reachesVia` predicate, represents any value of `visitedStates`.

For now, the path defined in `reachesVia` just lists the order of cargo items to ferry. You could tweak the predicate and the select clause to make the solution clearer. Here are some suggestions:

- Display more information, such as the direction in which the cargo is ferried, for example `"Goat to the left shore"`.
- Fully describe the state at every step, for example `"Goat: Left, Man: Left, Cabbage: Right, Wolf: Right"`.
- Display the path in a more "visual" way, for example by using arrows to display the transitions between states.

Alternative solutions

Here are some more example queries that solve the river crossing puzzle:

1. This query uses a modified `path` variable to describe the resulting path in more detail.

   See solution in the query console on LGTM.com

2. This query models the man and the cargo items in a different way, using an *abstract* class and predicate. It also displays the resulting path in a more visual way.

   See solution in the query console on LGTM.com

3. This query introduces *algebraic datatypes* to model the situation, instead of defining everything as a subclass of `string`.

   See solution in the query console on LGTM.com

Further reading

- "*QL language reference*"
- "*CodeQL tools*"
- *Introduction to QL*: Work through some simple exercises and examples to learn about the basics of QL and CodeQL.
- *Find the thief*: Take on the role of a detective to find the thief in this fictional village. You will learn how to use logical connectives, quantifiers, and aggregates in QL along the way.
- *Catch the fire starter*: Learn about QL predicates and classes to solve your second mystery as a QL detective.
- *Crown the rightful heir*: This is a QL detective puzzle that shows you how to use recursion in QL to write more complex queries.
- *Cross the river*: Use common QL features to write a query that finds a solution to the "River crossing" logic puzzle.

CHAPTER

FIVE

CODEQL LANGUAGE GUIDES

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from the languages supported in CodeQL analysis.

5.1 CodeQL for C and C++

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from C and C++ codebases.

5.1.1 Basic query for C and C++ code

Learn to write and run a simple CodeQL query using LGTM.

About the query

The query we're going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they have an empty then branch. For example, code such as:

```
if (error) { }
```

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **C/C++** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import cpp

from IfStmt ifstmt, BlockStmt block
where ifstmt.getThen() = block and
  block.getNumStmt() = 0
select ifstmt, "This 'if' statement is redundant."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis (...) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import cpp`	Imports the standard CodeQL libraries for C/C++.	Every query begins with one or more `import` statements.
`from IfStmt ifstmt, BlockStmt block`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `IfStmt` variable for `if` statements • a `BlockStmt` variable for the statement block
`where ifstmt.getThen() = block and block.getNumStmt() = 0`	Defines a condition on the variables.	`ifstmt.getThen() = block` relates the two variables. The block must be the `then` branch of the `if` statement. `block.getNumStmt() = 0` states that the block must be empty (that is, it contains no statements).
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if (...) {
  ...
} else if (!strcmp(option, "-verbose") {
  // nothing to do - handled earlier
} else {
  error("unrecognized option");
}
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to ignore empty `then` branches if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Extend the `where` clause to include the following extra condition:

```
and not ifstmt.hasElse()
```

   The `where` clause is now:

```
where ifstmt.getThen() = block and
  block.getNumStmt() = 0 and
  not ifstmt.hasElse()
```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer reported.

See this in the query console

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.2 CodeQL library for C and C++

When analyzing C or C++ code, you can use the large collection of classes in the CodeQL library for C and C++.

About the CodeQL library for C and C++

There is an extensive library for analyzing CodeQL databases extracted from C/C++ projects. The classes in this library present the data from a database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks. The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `cpp.qll` imports all the core C/C++ library modules, so you can include the complete library by beginning your query with:

```
import cpp
```

The rest of this topic summarizes the available CodeQL classes and corresponding C/C++ constructs.

Commonly-used library classes

The most commonly used standard library classes are listed below. The listing is broken down by functionality. Each library class is annotated with a C/C++ construct it corresponds to.

Declaration classes

This table lists Declaration classes representing C/C++ declarations.

Example syntax	CodeQL class	Remarks
`int var ;`	GlobalVariable	
`namespace N { ... float var ; ... }`	NamespaceVariable	
`int func ( void ) { ... float var ; ... }`	LocalVariable	See also Initializer
`class C { ... int var ; ... }`	MemberVariable	
`int func (const char param );`	Function	
`template < typename T > void func ( T param);`	TemplateFunction	
`int func (const char* format , ...) { ... }`	FormattingFunction	
`func < int, float > ( ... );`	FunctionTemplateInstantiation	
`template < typename T > func < int, T > ( ... ) { ... }`	FunctionTemplateSpecialization	
`class C { ... int func ( float param ); ... };`	MemberFunction	
`class C { ... int func ( float param ) const; ... };`	ConstMemberFunction	
`class C { ... virtual int func ( ... ) { ... } };`	VirtualFunction	
`class C { ... C ( ... ) { ... } ... };`	Constructor	
`C::operator float () const;`	ConversionOperator	
`class C { ... ~C ( void ) { ... } ... };`	Destructor	
`class C { ... C ( const D & d ) { ... } ... };`	ConversionConstructor	
`C & C :: operator= (const C & );`	CopyAssignmentOperator	
`C & C :: operator= ( C && );`	MoveAssignmentOperator	

continues on next page

Table 1 – continued from previous page

Example syntax	CodeQL class	Remarks
C :: *C* (const *C* &);	CopyConstructor	
C :: *C* (*C* &&);	MoveConstructor	
C :: *C* (void);	NoArgConstructor	Default constructor
enum *en* { *val1* , *val2* ... }	EnumConstant	
friend void *func* (int); friend class *B* ;	FriendDecl	
int *func* (void) { ... enum *en* { *val1* , *val2* ... }; ... }	LocalEnum	
class *C* { ... enum *en* { *val1* , *val2* ... } ... }	NestedEnum	
enum class *en* : short { *val1* , *val2* ... }	ScopedEnum	
class *C* { ... virtual void *func* (int) = 0; ... };	AbstractClass	
template < int , float > class *C* { ... };	ClassTemplateInstantiation	
template < > class *C* < Type > { ... };	FullClassTemplateSpecialization	
template < typename *T* > class *C* < *T* , *5* > { ... };	PartialClassTemplateSpecialization	
int *func* (void) { ... class *C* { ... }; ... }	LocalClass	
class *C* { ... class *D* { ... }; ... };	NestedClass	
class *C* { Type *var* ; Type *func* (Parameter...) { ... }... };	Class	
struct *S* { ... Type *var* ; Type *func* (Parameter...) { ... }... };	Struct Class	

continues on next page

Table 1 – continued from previous page

Example syntax	CodeQL class	Remarks
`union U { Type var1 ; Type var2 ; ... };`	Union Struct Class	
`template < typename T > struct C : T { ... };`	ProxyClass	Appears only in *uninstantiated* templates
`int func ( void ) { ... struct S { ... }; ... }`	LocalStruct	
`class C { ... struct S { ... }; ... };`	NestedStruct	
`int * func ( void ) { ... union U { ... }; ... }`	LocalUnion	
`class C { ... union U { ... }; ... };`	NestedUnion	
`typedef int T ;`	TypedefType	
`int func ( void ) { ... typedef int T ; ... }`	LocalTypedefType	
`class C { ... typedef int T ; ... };`	NestedTypedefType	
`class V : ... public B ... { ... };`	ClassDerivation	
`class V : ... virtual B ... { ... };`	VirtualClassDerivation	
`template < typename T > class C { ... };`	TemplateClass	
`int foo ( Type param1 , Type param2 ... );`	Parameter	
`template <typename T > T t ;`	TemplateVariable	Since C++14

Statement classes

This table lists subclasses of Stmt representing C/C++ statements.

Example syntax	CodeQL class	Remarks
`__asm__ ("` *movb %bh, (%eax)* `");`	AsmStmt	Specific to a given CPU instruction set
`{` Stmt… `}`	BlockStmt	
`catch (` Parameter `)` BlockStmt	CatchBlock	
`catch ( ... )` BlockStmt	CatchAnyBlock	
`goto *` *labelptr* `;`	ComputedGotoStmt	GNU extension; use with LabelLiteral
Type *i* `,` *j* `;`	DeclStmt	
`if (` Expr `)` Stmt `else` Stmt	IfStmt	
`switch (` Expr `) {` SwitchCase… `}`	SwitchStmt	
`do` Stmt `while (` Expr `)`	DoStmt	
`for (` DeclStmt `;` Expr `;` Expr `)` Stmt	ForStmt	
`for (` DeclStmt `:` Expr `)` Stmt	RangeBasedForStmt	
`while (` Expr `)` Stmt	WhileStmt	
Expr `;`	ExprStmt	
`__try {`… `} __except (` Expr `) {`… `}`	MicrosoftTryExceptStmt	Structured exception handling (SEH) under Windows
`__try {`… `} __finally {`… `}`	MicrosoftTryFinallyStmt	Structured exception handling (SEH) under Windows
`return` Expr `;`	ReturnStmt	
`case` Expr `:`	SwitchCase	
`try {` Stmt… `}` CatchBlock… CatchAnyBlock	TryStmt	
`void` *func* `(void) try {` Stmt… `}` CatchBlock… CatchAnyBlock	FunctionTryStmt	
`;`	EmptyStmt	
`break;`	BreakStmt	
`continue;`	ContinueStmt	
`goto` LabelStmt `;`	GotoStmt	
slabel `:`	LabelStmt	
`float` *arr* `[` Expr `] [` Expr `];`	VlaDeclStmt	C99 variable-length array

Expression classes

This table lists subclasses of Expr representing C/C++ expressions.

Example syntax	CodeQL class(es)	Remarks
{ Expr... }	ArrayAggregateLiteral ClassAggregateLiteral	
alignof (Expr)	AlignofExprOperator	
alignof (Type)	AlignofTypeOperator	
Expr [Expr]	ArrayExpr	
__assume (Expr)	AssumeExpr	Microsoft extension
static_assert (Expr , StringLiteral) _Static_assert (Expr , StringLiteral)	StaticAssert	C++11 C11
__noop;	BuiltInNoOp	Microsoft extension
Expr (Expr...)	ExprCall	
func (Expr...) *instance* . *func* (Expr...)	FunctionCall	
Expr , Expr	CommaExpr	
if (Type *arg* = Expr)	ConditionDeclExpr	
(Type) Expr	CStyleCast	
const_cast < Type > (Expr)	ConstCast	
dynamic_cast < Type > (Expr)	DynamicCast	
reinterpret_cast < Type > (Expr)	ReinterpretCast	
static_cast < Type > (Expr)	StaticCast	
template < typename... *T* > auto *sum* (*T* ... *t*) { return (*t* + ... + 0); }	FoldExpr	Appears only in *uninstantiated* templates
int *func* (*format* , ...);	FormattingFunctionCall	
[=] (float *b*) -> float { return *captured* * *b* ; }	LambdaExpression	C++11

^ int (int *x* , int *y*) {
{ Stmt... ; return *x* + *y* ;
}
	BlockExpr	Apple extension

void * *labelptr* = && *label* ;	LabelLiteral	GNU extension; use with ComputedGotoStmt
"%3d %s\n"	FormatLiteral	
0xdbceffca	HexLiteral	

continues on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
0167	OctalLiteral	
'c'	CharLiteral	
"abcdefgh", L"wide"	StringLiteral	
`new` Type `[` Expr `]`	NewArrayExpr	
`new` Type	NewExpr	
`delete [ ]` Expr `;`	DeleteArrayExpr	
`delete` Expr `;`	DeleteExpr	
`noexcept (` Expr `)`	NoExceptExpr	
Expr = Expr	AssignExpr	See also Initializer
Expr += Expr	AssignAddExpr AssignPointerAddExpr	
Expr /= Expr	AssignDivExpr	
Expr *= Expr	AssignMulExpr	
Expr %= Expr	AssignRemExpr	
Expr -= Expr	AssignSubExpr AssignPointerSubExpr	
Expr &= Expr	AssignAndExpr	
Expr <<= Expr	AssignLShiftExpr	
Expr ``	=`` Expr	AssignOrExpr
Expr >>= Expr	AssignRShiftExpr	
Expr ^= Expr	AssignXorExpr	
Expr + Expr	AddExpr PointerAddExpr ImaginaryRealAddExpr RealImaginaryAddExpr	 C99 C99
Expr / Expr	DivExpr ImaginaryDivExpr	 C99
Expr >? Expr	MaxExpr	GNU extension
Expr <? Expr	MinExpr	GNU extension
Expr * Expr	MulExpr ImaginaryMulExpr	 C99
Expr % Expr	RemExpr	
Expr - Expr	SubExpr PointerDiffExpr PointerSubExpr ImaginaryRealSubExpr RealImaginarySubExpr	 C99 C99

continues on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
Expr & Expr	BitwiseAndExpr	
Expr | Expr	BitwiseOrExpr	
Expr ^ Expr	BitwiseXorExpr	
Expr << Expr	LShiftExpr	
Expr >> Expr	RShiftExpr	
Expr && Expr	LogicalAndExpr	
Expr || Expr	LogicalOrExpr	
Expr == Expr	EQExpr	
Expr != Expr	NEExpr	
Expr >= Expr	GEExpr	
Expr > Expr	GTExpr	
Expr <= Expr	LEExpr	
Expr < Expr	LTExpr	
Expr ? Expr : Expr	ConditionalExpr	
& Expr	AddressOfExpr	
* Expr	PointerDereferenceExpr	
Expr --	PostfixDecrExpr	
-- Expr	PrefixDecrExpr	
Expr ++	PostfixIncrExpr	
++ Expr	PrefixIncrExpr	
__imag (Expr)	ImaginaryPartExpr	GNU extension
__real (Expr)	RealPartExpr	GNU extension
- Expr	UnaryMinusExpr	
+ Expr	UnaryPlusExpr	
~ Expr	ComplementExpr	
	ConjugationExpr	GNU extension
! Expr	NotExpr	
int *vect* __attribute__ ((vector_size (*16*))) = { *3* , *8* , *32* , *33* };	VectorFillOperation	GNU extension
sizeof (Expr)	SizeofExprOperator	
sizeof (Type)	SizeofTypeOperator	
template < typename... *T* > int *count* (*T* &&... *t*) { return sizeof... (*t*); }	SizeofPackOperator	
({ Stmt... ; Expr })	StmtExpr	GNU/Clang extension
this	ThisExpr	
throw (Expr);	ThrowExpr	
throw;	ReThrowExpr	

continues on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
`typeid ( Expr )` `typeid ( Type )`	TypeidOperator	
`__uuidof ( Expr )`	UuidofOperator	Microsoft extension

Type classes

This table lists subclasses of Type representing C/C++ types.

Example syntax	CodeQL class	Remarks
`void`	VoidType	
`_Bool` or `bool`	BoolType	
`char16_t`	Char16Type	C11, C++11
`char32_t`	Char32Type	C11, C++11
`char`	PlainCharType	
`signed char`	SignedCharType	
`unsigned char`	UnsignedCharType	
`int`	IntType	
`long long`	LongLongType	
`long`	LongType	
`short`	ShortType	
`wchar_t`	WideCharType	
`nullptr_t`	NullPointerType	
`double`	DoubleType	
`long double`	LongDoubleType	
`float`	FloatType	
`auto`	AutoType	
`decltype ( Expr )`	Decltype	
Type `[ n ]`	ArrayType	
Type `( ^ blockptr ) ( Parameter... )`	BlockType	Apple extension
Type `( * funcptr ) ( Parameter... )`	FunctionPointerType	
Type `( &funcref ) ( Parameter... )`	FunctionReferenceType	
Type `__attribute__ ( ( vector_size ( n ) ) )`	GNUVectorType	
Type `*`	PointerType	
Type `&`	LValueReferenceType	
Type `&&`	RValueReferenceType	
Type `( Class *:: membptr ) ( Parameter... )`	PointerToMemberType	
`template < template < typename > class C >`	TemplateTemplateParameter	
`template < typename T >`	TemplateParameter	

Preprocessor classes

This table lists Preprocessor classes representing C/C++ preprocessing directives.

Example syntax	CodeQL class	Remarks
`#elif` *condition*	PreprocessorElif	
`#if` *condition*	PreprocessorIf	
`#ifdef` *macro*	PreprocessorIfdef	
`#ifndef` *macro*	PreprocessorIfndef	
`#else`	PreprocessorElse	
`#endif`	PreprocessorEndif	
`#line` *line_number file_name*	PreprocessorLine	
`#pragma` *pragma_property*	PreprocessorPragma	
`#undef` *macro*	PreprocessorUndef	
`#warning` *message*	PreprocessorWarning	
`#error` *message*	PreprocessorError	
`#include` *file_name*	Include	
`#import` *file_name*	Import	Apple/NeXT extension
`#include_next` *file_name*	IncludeNext	Apple/NeXT extension
`#define` *macro ...*	Macro	

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.3 Functions in C and C++

You can use CodeQL to explore functions in C and C++ code.

Overview

The standard CodeQL library for C and C++ represents functions using the `Function` class (see *CodeQL libraries for C and C++*).

The example queries in this topic explore some of the most useful library predicates for querying functions.

Finding all static functions

Using the member predicate `Function.isStatic()` we can list all the static functions in a database:

```
import cpp

from Function f
where f.isStatic()
select f, "This is a static function."
```

This query is very general, so there are probably too many results to be interesting for most nontrivial projects.

Finding functions that are not called

It might be more interesting to find functions that are not called, using the standard CodeQL `FunctionCall` class from the **abstract syntax tree** category (see *CodeQL libraries for C and C++*). The `FunctionCall` class can be used to identify places where a function is actually used, and it is related to `Function` through the `FunctionCall.getTarget()` predicate.

```
import cpp

from Function f
where not exists(FunctionCall fc | fc.getTarget() = f)
select f, "This function is never called."
```

See this in the query console on LGTM.com

The new query finds functions that are not the target of any `FunctionCall`—in other words, functions that are never called. You may be surprised by how many results the query finds. However, if you examine the results, you can see that many of the functions it finds are used indirectly. To create a query that finds only unused functions, we need to refine the query and exclude other ways of using a function.

Excluding functions that are referenced with a function pointer

You can modify the query to remove functions where a function pointer is used to reference the function:

```
import cpp

from Function f
where not exists(FunctionCall fc | fc.getTarget() = f)
  and not exists(FunctionAccess fa | fa.getTarget() = f)
select f, "This function is never called, or referenced with a function pointer."
```

See this in the query console on LGTM.com

This query returns fewer results. However, if you examine the results then you can probably still find potential refinements.

For example, there is a more complicated LGTM query that finds unused static functions. To see the code for this query, click **Open in query console** at the top of the page.

> You can explore the definition of an element in the standard libraries and see what predicates are available. Use the keyboard **F3** button to open the definition of any element. Alternatively, hover over the element and click **Jump to definition** in the tooltip displayed. The library file is opened in a new tab with the definition highlighted.

Finding a specific function

This query uses `Function` and `FunctionCall` to find calls to the function `sprintf` that have a variable format string—which is potentially a security hazard.

```
import cpp

from FunctionCall fc
where fc.getTarget().getQualifiedName() = "sprintf"
  and not fc.getArgument(1) instanceof StringLiteral
select fc, "sprintf called with variable format string."
```

See this in the query console on LGTM.com

This uses:

- `Declaration.getQualifiedName()` to identify calls to the specific function `sprintf`.
- `FunctionCall.getArgument(1)` to fetch the format string argument.

Note that we could have used `Declaration.getName()`, but `Declaration.getQualifiedName()` is a better choice because it includes the namespace. For example: `getName()` would return `vector` where `getQualifiedName` would return `std::vector`.

The LGTM version of this query is considerably more complicated, but if you look carefully you will find that its structure is the same. See Non-constant format string and click **Open in query console** at the top of the page.

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.4 Expressions, types, and statements in C and C++

You can use CodeQL to explore expressions, types, and statements in C and C++ code to find, for example, incorrect assignments.

Expressions and types in CodeQL

Each part of an expression in C becomes an instance of the `Expr` class. For example, the C code `x = x + 1` becomes an `AssignExpr`, an `AddExpr`, two instances of `VariableAccess` and a `Literal`. All of these CodeQL classes extend `Expr`.

Finding assignments to zero

In the following example we find instances of `AssignExpr` which assign the constant value zero:

```
import cpp

from AssignExpr e
where e.getRValue().getValue().toInt() = 0
select e, "Assigning the value 0 to something."
```

See this in the query console on LGTM.com

The `where` clause in this example gets the expression on the right side of the assignment, `getRValue()`, and compares it with zero. Notice that there are no checks to make sure that the right side of the assignment is an integer or that it has a value (that is, it is compile-time constant, rather than a variable). For expressions where either of these assumptions is wrong, the associated predicate simply does not return anything and the `where` clause will not produce a result. You could think of it as if there is an implicit `exists(e.getRValue().getValue().toInt())` at the beginning of this line.

It is also worth noting that the query above would find this C code:

```
yPtr = NULL;
```

This is because the database contains a representation of the code base after the preprocessor transforms have run. This means that any macro invocations, such as the `NULL` define used here, are expanded during the creation of the database. If you want to write queries about macros then there are some special library classes that have been designed specifically for this purpose (for example, the `Macro`, `MacroInvocation` classes and predicates like `Element.isInMacroExpansion()`). In this case, it is good that macros are expanded, but we do not want to find assignments to pointers. For more information, see Database generation on LGTM.com.

Finding assignments of 0 to an integer

We can make the query more specific by defining a condition for the left side of the expression. For example:

```
import cpp

from AssignExpr e
where e.getRValue().getValue().toInt() = 0
  and e.getLValue().getType().getUnspecifiedType() instanceof IntegralType
select e, "Assigning the value 0 to an integer."
```

See this in the query console on LGTM.com

This checks that the left side of the assignment has a type that is some kind of integer. Note the call to `Type.getUnspecifiedType()`. This resolves `typedef` types to their underlying types so that the query finds assignments like this one:

```
typedef int myInt;
myInt i;

i = 0;
```

Statements in CodeQL

We can refine the query further using statements. In this case we use the class `ForStmt`:

- `Stmt` - C/C++ statements
  - `Loop`
    - `WhileStmt`
    - `ForStmt`
    - `DoStmt`
  - `ConditionalStmt`
    - `IfStmt`
    - `SwitchStmt`
  - `TryStmt`
  - `ExprStmt` - expressions used as a statement; for example, an assignment
  - `Block` - { } blocks containing more statements

Finding assignments of 0 in 'for' loop initialization

We can restrict the previous query so that it only considers assignments inside `for` statements by adding the `ForStmt` class to the query. Then we want to compare the expression to `ForStmt.getInitialization()`:

```
import cpp

from AssignExpr e, ForStmt f
// the assignment is the for loop initialization
where e = f.getInitialization()
...
```

Unfortunately this would not quite work, because the loop initialization is actually a `Stmt` not an `Expr`—the `AssignExpr` class is wrapped in an `ExprStmt` class. Instead, we need to find the closest enclosing `Stmt` around the expression using `Expr.getEnclosingStmt()`:

```
import cpp

from AssignExpr e, ForStmt f
// the assignment is in the 'for' loop initialization statement
where e.getEnclosingStmt() = f.getInitialization()
  and e.getRValue().getValue().toInt() = 0
  and e.getLValue().getType().getUnspecifiedType() instanceof IntegralType
select e, "Assigning the value 0 to an integer, inside a for loop initialization."
```

See this in the query console on LGTM.com

Finding assignments of 0 within the loop body

We can find assignments inside the loop body using similar code with the predicate `Loop.getStmt()`:

```
import cpp

from AssignExpr e, ForStmt f
// the assignment is in the for loop body
where e.getEnclosingStmt().getParentStmt*() = f.getStmt()
  and e.getRValue().getValue().toInt() = 0
  and e.getLValue().getType().getUnderlyingType() instanceof IntegralType
select e, "Assigning the value 0 to an integer, inside a for loop body."
```

See this in the query console on LGTM.com

Note that we replaced `e.getEnclosingStmt()` with `e.getEnclosingStmt().getParentStmt*()`, to find an assignment expression that is deeply nested inside the loop body. The transitive closure modifier `*` here indicates that `Stmt.getParentStmt()` may be followed zero or more times, rather than just once, giving us the statement, its parent statement, its parent's parent statement etc.

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.5 Conversions and classes in C and C++

You can use the standard CodeQL libraries for C and C++ to detect when the type of an expression is changed.

Conversions

In C and C++, conversions change the type of an expression. They may be implicit conversions generated by the compiler, or explicit conversions requested by the user.

Let's take a look at the Conversion class in the standard library:

- `Expr`
  - `Conversion`
    - `Cast`
      - `CStyleCast`
      - `StaticCast`
      - `ConstCastReinterpretCast`
      - `DynamicCast`
    - `ArrayToPointerConversion`

* VirtualMemberToFunctionPointerConversion

Exploring the subexpressions of an assignment

Let us consider the following C code:

```
typedef signed int myInt;
int main(int argc, char *argv[])
{
    unsigned int i;
    i = (myInt)1;
    return 0;
}
```

And this simple query:

```
import cpp

from AssignExpr a
select a, a.getLValue().getType(), a.getRValue().getType()
```

The query examines the code for assignments, and tells us the type of their left and right subexpressions. In the example C code above, there is just one assignment. Notably, this assignment has two conversions (of type `CStyleCast`) on the right side:

1. Explicit cast of the integer 1 to a `myInt`.
2. Implicit conversion generated by the compiler, in preparation for the assignment, converting that expression into an `unsigned int`.

The query actually reports the result:

```
... = ... | unsigned int | int
```

It is as though the conversions are not there! The reason for this is that `Conversion` expressions do not wrap the objects they convert; instead conversions are attached to expressions and can be accessed using `Expr.getConversion()`. The whole assignment in our example is seen by the standard library classes like this:

```
AssignExpr, i = (myInt)1
VariableAccess, i
Literal, 1
     CStyleCast, myInt (explicit)
          CStyleCast, unsigned int (implicit)
```

Accessing parts of the assignment:

- Left side—access value using `Assignment.getLValue()`.
- Right side—access value using `Assignment.getRValue()`.
- Conversions of the `Literal` on the right side—access both using calls to `Expr.getConversion()`. As a shortcut, you can use `Expr.GetFullyConverted()` to follow all the way to the resulting type, or `Expr.GetExplicitlyConverted()` to find the last explicit conversion from an expression.

Using these predicates we can refine our query so that it reports the results that we expected:

```
import cpp

from AssignExpr a
select a, a.getLValue().getExplicitlyConverted().getType(), a.getRValue().
↪getExplicitlyConverted().getType()
```

The result is now:

```
... = ... | unsigned int | myInt
```

We can refine the query further by adding `Type.getUnderlyingType()` to resolve the `typedef`:

```
import cpp

from AssignExpr a
select a, a.getLValue().getExplicitlyConverted().getType().getUnderlyingType(), a.
↪getRValue().getExplicitlyConverted().getType().getUnderlyingType()
```

The result is now:

```
... = ... | unsigned int | signed int
```

If you simply wanted to get the values of all assignments in expressions, regardless of position, you could replace `Assignment.getLValue()` and `Assignment.getRValue()` with `Operation.getAnOperand()`:

```
import cpp

from AssignExpr a
select a, a.getAnOperand().getExplicitlyConverted().getType()
```

Unlike the earlier versions of the query, this query would return each side of the expression as a separate result:

```
... = ... | unsigned int
... = ... | myInt
```

> Note
>
> In general, predicates named `getAXxx` exploit the ability to return multiple results (multiple instances of `Xxx`) whereas plain `getXxx` predicates usually return at most one specific instance of `Xxx`.

Classes

Next we're going to look at C++ classes, using the following CodeQL classes:

- `Type`
  - `UserType`—includes classes, typedefs, and enums
    - `Class`—a class or struct
      - `Struct`—a struct, which is treated as a subtype of `Class`
      - `TemplateClass`—a C++ class template

Finding derived classes

We want to create a query that checks for destructors that should be `virtual`. Specifically, when a class and a class derived from it both have destructors, the base class destructor should generally be virtual. This ensures that the derived class destructor is always invoked. In the CodeQL library, `Destructor` is a subtype of `MemberFunction`:

- `Function`
  - `MemberFunction`
    - `Constructor`
    - `Destructor`

Our starting point for the query is pairs of a base class and a derived class, connected using `Class.getABaseClass()`:

```
import cpp

from Class base, Class derived
where derived.getABaseClass+() = base
select base, derived, "The second class is derived from the first."
```

See this in the query console on LGTM.com

Note that the transitive closure symbol + indicates that `Class.getABaseClass()` may be followed one or more times, rather than only accepting a direct base class.

A lot of the results are uninteresting template parameters. You can remove those results by updating the `where` clause as follows:

```
where derived.getABaseClass+() = base
  and not exists(base.getATemplateArgument())
  and not exists(derived.getATemplateArgument())
```

See this in the query console on LGTM.com

Finding derived classes with destructors

Now we can extend the query to find derived classes with destructors, using the `Class.getDestructor()` predicate:

```
import cpp

from Class base, Class derived, Destructor d1, Destructor d2
where derived.getABaseClass+() = base
  and not exists(base.getATemplateArgument())
  and not exists(derived.getATemplateArgument())
  and d1 = base.getDestructor()
  and d2 = derived.getDestructor()
select base, derived, "The second class is derived from the first, and both have a␣
↪destructor."
```

See this in the query console on LGTM.com

Notice that getting the destructor implicitly asserts that one exists. As a result, this version of the query returns fewer results than before.

Finding base classes where the destructor is not virtual

Our last change is to use `Function.isVirtual()` to find cases where the base destructor is not virtual:

```
import cpp

from Class base, Destructor d1, Class derived, Destructor d2
where derived.getABaseClass+() = base
  and d1.getDeclaringType() = base
  and d2.getDeclaringType() = derived
  and not d1.isVirtual()
select d1, "This destructor should probably be virtual."
```

See this in the query console on LGTM.com

That completes the query.

There is a similar built-in query on LGTM.com that finds classes in a C/C++ project with virtual functions but no virtual destructor. You can take a look at the code for this query by clicking **Open in query console** at the top of that page.

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.6 Analyzing data flow in C and C++

You can use data flow analysis to track the flow of potentially malicious or insecure data that can cause vulnerabilities in your codebase.

About data flow

Data flow analysis computes the possible values that a variable can hold at various points in a program, determining how those values propagate through the program, and where they are used. In CodeQL, you can model both local data flow and global data flow. For a more general introduction to modeling data flow, see "*About data flow analysis*."

Local data flow

Local data flow is data flow within a single function. Local data flow is usually easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. `Node`s are divided into expression nodes (`ExprNode`) and parameter nodes (`ParameterNode`). It is possible to map between data flow nodes and expressions/parameters using the member predicates `asExpr` and `asParameter`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the parameter corresponding to this node, if any. */
  Parameter asParameter() { ... }

  ...
}
```

or using the predicates `exprNode` and `parameterNode`:

```
/**
 * Gets the node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }

/**
 * Gets the node corresponding to the value of parameter `p` at function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. The predicate can be applied recursively (using the + and * operators), or through the predefined recursive predicate `localFlow`, which is equivalent to `localFlowStep*`.

For example, finding flow from a parameter `source` to an expression `sink` in zero or more local steps can be achieved as follows:

```
DataFlow::localFlow(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
int i = tainted_user_input();
some_big_struct *array = malloc(i * sizeof(some_big_struct));
```

In this case, the argument to `malloc` is tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint propagation edge from the node `nodeFrom` to the node `nodeTo`. The predicate can be applied recursively (using the + and * operators), or through the predefined recursive predicate `localTaint`, which is equivalent to `localTaintStep*`.

For example, finding taint propagation from a parameter `source` to an expression `sink` in zero or more local steps can be achieved as follows:

```
TaintTracking::localTaint(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Examples

The following query finds the filename passed to `fopen`.

```
import cpp

from Function fopen, FunctionCall fc
where fopen.hasQualifiedName("fopen")
  and fc.getTarget() = fopen
select fc.getArgument(0)
```

Unfortunately, this will only give the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import cpp
import semmle.code.cpp.dataflow.DataFlow

from Function fopen, FunctionCall fc, Expr src
where fopen.hasQualifiedName("fopen")
  and fc.getTarget() = fopen
  and DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(fc.getArgument(0)))
select src
```

Then we can vary the source, for example an access to a public parameter. The following query finds where a public parameter is used to open a file:

```
import cpp
import semmle.code.cpp.dataflow.DataFlow

from Function fopen, FunctionCall fc, Parameter p
where fopen.hasQualifiedName("fopen")
  and fc.getTarget() = fopen
  and DataFlow::localFlow(DataFlow::parameterNode(p), DataFlow::exprNode(fc.
↪getArgument(0)))
select p
```

The following example finds calls to formatting functions where the format string is not hard-coded.

```
import semmle.code.cpp.dataflow.DataFlow
import semmle.code.cpp.commons.Printf

from FormattingFunction format, FunctionCall call, Expr formatString
where call.getTarget() = format
  and call.getArgument(format.getFormatParameterIndex()) = formatString
  and not exists(DataFlow::Node source, DataFlow::Node sink |
    DataFlow::localFlow(source, sink) and
    source.asExpr() instanceof StringLiteral and
    sink.asExpr() = formatString
  )
select call, "Argument to " + format.getQualifiedName() + " isn't hard-coded."
```

Exercises

Exercise 1: Write a query that finds all hard-coded strings used to create a `host_ent` via `gethostbyname`, using local data flow. (*Answer*)

Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see *Creating path queries*.

Using global data flow

The global data flow library is used by extending the class `DataFlow::Configuration` as follows:

```
import semmle.code.cpp.dataflow.DataFlow

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

The following predicates are defined in the configuration:

- `isSource`—defines where data may flow from
- `isSink`—defines where data may flow to
- `isBarrier`—optional, restricts the data flow
- `isBarrierGuard`—optional, restricts the data flow
- `isAdditionalFlowStep`—optional, adds additional flow steps

The characteristic predicate `MyDataFlowConfiguration()` defines the name of the configuration, so `"MyDataFlowConfiguration"` should be replaced by the name of your class.

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Data flow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow as local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. The global taint tracking library is used by extending the class `TaintTracking::Configuration` as follows:

```
import semmle.code.cpp.dataflow.TaintTracking

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

The following predicates are defined in the configuration:

- `isSource`—defines where taint may flow from
- `isSink`—defines where taint may flow to
- `isSanitizer`—optional, restricts the taint flow
- `isSanitizerGuard`—optional, restricts the taint flow
- `isAdditionalTaintStep`—optional, adds additional taint steps

Similar to global data flow, the characteristic predicate `MyTaintTrackingConfiguration()` defines the unique name of the configuration, so `"MyTaintTrackingConfiguration"` should be replaced by the name of your class.

The taint tracking analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Examples

The following data flow configuration tracks data flow from environment variables to opening files in a Unix-like environment:

```
import semmle.code.cpp.dataflow.DataFlow

class EnvironmentToFileConfiguration extends DataFlow::Configuration {
  EnvironmentToFileConfiguration() { this = "EnvironmentToFileConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    exists (Function getenv |
      source.asExpr().(FunctionCall).getTarget() = getenv and
      getenv.hasQualifiedName("getenv")
    )
  }

  override predicate isSink(DataFlow::Node sink) {
```

(continues on next page)

(continued from previous page)

```
    exists (FunctionCall fc |
      sink.asExpr() = fc.getArgument(0) and
      fc.getTarget().hasQualifiedName("fopen")
    )
  }
}

from Expr getenv, Expr fopen, EnvironmentToFileConfiguration config
where config.hasFlow(DataFlow::exprNode(getenv), DataFlow::exprNode(fopen))
select fopen, "This 'fopen' uses data from $@.",
  getenv, "call to 'getenv'"
```

The following taint-tracking configuration tracks data from a call to `ntohl` to an array index operation. It uses the `Guards` library to recognize expressions that have been bounds-checked, and defines `isSanitizer` to prevent taint from propagating through them. It also uses `isAdditionalTaintStep` to add flow from loop bounds to loop indexes.

```
import cpp
import semmle.code.cpp.controlflow.Guards
import semmle.code.cpp.dataflow.TaintTracking

class NetworkToBufferSizeConfiguration extends TaintTracking::Configuration {
  NetworkToBufferSizeConfiguration() { this = "NetworkToBufferSizeConfiguration" }

  override predicate isSource(DataFlow::Node node) {
    node.asExpr().(FunctionCall).getTarget().hasGlobalName("ntohl")
  }

  override predicate isSink(DataFlow::Node node) {
    exists(ArrayExpr ae | node.asExpr() = ae.getArrayOffset())
  }

  override predicate isAdditionalTaintStep(DataFlow::Node pred, DataFlow::Node succ) {
    exists(Loop loop, LoopCounter lc |
      loop = lc.getALoop() and
      loop.getControllingExpr().(RelationalOperation).getGreaterOperand() = pred.
↪asExpr() |
      succ.asExpr() = lc.getVariableAccessInLoop(loop)
    )
  }

  override predicate isSanitizer(DataFlow::Node node) {
    exists(GuardCondition gc, Variable v |
      gc.getAChild*() = v.getAnAccess() and
      node.asExpr() = v.getAnAccess() and
      gc.controls(node.asExpr().getBasicBlock(), _)
    )
  }
}

from DataFlow::Node ntohl, DataFlow::Node offset, NetworkToBufferSizeConfiguration conf
where conf.hasFlow(ntohl, offset)
select offset, "This array offset may be influenced by $@.", ntohl,
```

(continues on next page)

(continued from previous page)

```
  "converted data from the network"
```

Exercises

Exercise 2: Write a query that finds all hard-coded strings used to create a `host_ent` via `gethostbyname`, using global data flow. (*Answer*)

Exercise 3: Write a class that represents flow sources from `getenv`. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query which finds all global data flows from `getenv` to `gethostbyname`. (*Answer*)

Answers

Exercise 1

```
import semmle.code.cpp.dataflow.DataFlow

from StringLiteral sl, FunctionCall fc
where fc.getTarget().hasName("gethostbyname")
  and DataFlow::localFlow(DataFlow::exprNode(sl), DataFlow::exprNode(fc.getArgument(0)))
select sl, fc
```

Exercise 2

```
import semmle.code.cpp.dataflow.DataFlow

class LiteralToGethostbynameConfiguration extends DataFlow::Configuration {
  LiteralToGethostbynameConfiguration() {
    this = "LiteralToGethostbynameConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists (FunctionCall fc |
      sink.asExpr() = fc.getArgument(0) and
      fc.getTarget().hasName("gethostbyname"))
  }
}

from StringLiteral sl, FunctionCall fc, LiteralToGethostbynameConfiguration cfg
where cfg.hasFlow(DataFlow::exprNode(sl), DataFlow::exprNode(fc.getArgument(0)))
select sl, fc
```

Exercise 3

```
import cpp

class GetenvSource extends FunctionCall {
  GetenvSource() {
    this.getTarget().hasQualifiedName("getenv")
  }
}
```

Exercise 4

```
import semmle.code.cpp.dataflow.DataFlow

class GetenvSource extends DataFlow::Node {
  GetenvSource() {
    this.asExpr().(FunctionCall).getTarget().hasQualifiedName("getenv")
  }
}

class GetenvToGethostbynameConfiguration extends DataFlow::Configuration {
  GetenvToGethostbynameConfiguration() {
    this = "GetenvToGethostbynameConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof GetenvSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists (FunctionCall fc |
      sink.asExpr() = fc.getArgument(0) and
      fc.getTarget().hasName("gethostbyname"))
  }
}

from DataFlow::Node getenv, FunctionCall fc, GetenvToGethostbynameConfiguration cfg
where cfg.hasFlow(getenv, DataFlow::exprNode(fc.getArgument(0)))
select getenv.asExpr(), fc
```

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.7 Refining a query to account for edge cases

You can improve the results generated by a CodeQL query by adding conditions to remove false positive results caused by common edge cases.

Overview

This topic describes how a C++ query was developed. The example introduces recursive predicates and demonstrates the typical workflow used to refine a query. For a full overview of the topics available for learning to write queries for C/C++ code, see "*CodeQL for C and C++*."

Finding every private field and checking for initialization

Writing a query to check if a constructor initializes all private fields seems like a simple problem, but there are several edge cases to account for.

Basic query

We can start by looking at every private field in a class and checking that every constructor in that class initializes them. Once you are familiar with the library for C++ this is not too hard to do.

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    and not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() =␣
↪c)
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

1. `f.getDeclaringType() = c.getDeclaringType()` asserts that the field and constructor are both part of the same class.
2. `f.isPrivate()` checks if the field is private.
3. `not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)` checks that there is no assignment to the field in the constructor.

This code looks fairly complete, but when you test it on a project, there are several results that contain examples that we have overlooked.

Refinement 1—excluding fields initialized by lists

You may see that the results contain fields that are initialized by constructor initialization lists, instead of by assignment statements. For example, the following class:

```
class BoxedInt {
public:
  BoxedInt(int value) : m_value(value) {}

private:
  int m_value;
};
```

These can be excluded by adding an extra condition to check for this special constructor-only form of assignment.

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    and not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() =␣
↪c)
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.
↪getEnclosingFunction() = c)
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

Refinement 2—excluding fields initialized by external libraries

When you test the revised query, you may discover that fields from classes in external libraries are over-reported. This is often because a header file declares a constructor that is defined in a source file that is not analyzed (external libraries are often excluded from analysis). When the source code is analyzed, the CodeQL database is populated with a `Constructor` entry with no body. This `constructor` therefore contains no assignments and consequently the query reports that any fields initialized by the constructor are "uninitialized." There is no particular reason to be suspicious of these cases, and we can exclude them from the results by defining a condition to exclude constructors that have no body:

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    and not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() =␣
↪c)
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.
↪getEnclosingFunction() = c)
    // ignore cases where the constructor source code is not available
    and exists(c.getBlock())
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

This is a reasonably precise query—most of the results that it reports are interesting. However, you could make further refinements.

Refinement 3—excluding fields initialized indirectly

You may also wish to consider methods called by constructors that assign to the fields, or even to the methods called by those methods. As a concrete example of this, consider the following class.

```
class BoxedInt {
public:
  BoxedInt(int value) {
    setValue(value);
  }

  void setValue(int value) {
    m_value = value;
```

(continues on next page)

(continued from previous page)

```
  }

private:
  int m_value;
};
```

This case can be excluded by creating a recursive predicate. The recursive predicate is given a function and a field, then checks whether the function assigns to the field. The predicate runs itself on all the functions called by the function that it has been given. By passing the constructor to this predicate, we can check for assignments of a field in all functions called by the constructor, and then do the same for all functions called by those functions all the way down the tree of function calls. For more information, see "*Recursion*" in the QL language reference.

```
import cpp

predicate getSubAssignment(Function c, Field f){
    exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)
    or exists(Function fun | c.calls(fun) and getSubAssignment(fun, f))
}
from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.
↪getEnclosingFunction() = c)
    // check for initializations performed indirectly by methods called
    // as a result of the constructor being called
    and not getSubAssignment(c, f)
    // ignore cases where the constructor source code is not available
    and exists(c.getBlock())
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

Refinement 4—simplifying the query

Finally we can simplify the query by using the transitive closure operator. In this final version of the query, `c.calls*(fun)` resolves to the set of all functions that are `c` itself, are called by `c`, are called by a function that is called by `c`, and so on. This eliminates the need to make a new predicate all together. For more information, see "*Transitive closures*" in the QL language reference.

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.
↪getEnclosingFunction() = c)
    // check for initializations performed indirectly by methods called
    // as a result of the constructor being called
    and not exists(Function fun, Assignment a |
      c.calls*(fun) and a = f.getAnAssignment() and a.getEnclosingFunction() = fun)
    // ignore cases where the constructor source code is not available
    and exists(c.getBlock())
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

See this in the query console on LGTM.com

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.8 Detecting a potential buffer overflow

You can use CodeQL to detect potential buffer overflows by checking for allocations equal to `strlen` in C and C++. This topic describes how a C/C++ query for detecting a potential buffer overflow was developed.

Problem—detecting memory allocation that omits space for a null termination character

The objective of this query is to detect C/C++ code which allocates an amount of memory equal to the length of a null terminated string, without adding +1 to make room for a null termination character. For example the following code demonstrates this mistake, and results in a buffer overflow:

```
void processString(const char *input)
{
    char *buffer = malloc(strlen(input));

    strcpy(buffer, input);

    ...
}
```

Basic query

Before you can write a query you need to decide what entities to search for and then define how to identify them.

Defining the entities of interest

You could approach this problem either by searching for code similar to the call to `malloc` in line 3 or the call to `strcpy` in line 5 (see example above). For our basic query, we start with a simple assumption: any call to `malloc` with only a `strlen` to define the memory size is likely to cause an error when the memory is populated.

Calls to `strlen` can be identified using the library StrlenCall class, but we need to define a new class to identify calls to `malloc`. Both the library class and the new class need to extend the standard class `FunctionCall`, with the added restriction of the function name that they apply to:

```
import cpp

class MallocCall extends FunctionCall
{
```

(continues on next page)

(continued from previous page)

```
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
}
```

Note

You could easily extend this class to include similar functions such as `realloc`, or your own custom allocator. With a little effort they could even include C++ `new` expressions (to do this, `MallocCall` would need to extend a common superclass of both `FunctionCall` and `NewExpr`, such as `Expr`).

Finding the `strlen(string)` pattern

Before we start to write our query, there's one remaining task. We need to modify our new `MallocCall` class, so it returns an expression for the size of the allocation. Currently this will be the first argument to the `malloc` call, `FunctionCall.getArgument(0)`, but converting this into a predicate makes it more flexible for future refinements.

```
class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
    Expr getAllocatedSize() {
        result = this.getArgument(0)
    }
}
```

Defining the basic query

Now we can write a query using these classes:

```
import cpp

class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
    Expr getAllocatedSize() {
        result = this.getArgument(0)
    }
}

from MallocCall malloc
where malloc.getAllocatedSize() instanceof StrlenCall
select malloc, "This allocation does not include space to null-terminate the string."
```

Note that there is no need to check whether anything is added to the `strlen` expression, as it would be in the corrected C code `malloc(strlen(string) + 1)`. This is because the corrected code would in fact be an `AddExpr` containing a `StrlenCall`, not an instance of `StrlenCall` itself. A side-effect of this approach is that we omit certain unlikely patterns such as `malloc(strlen(string) + 0)`. In practice we can always come back and extend our query to cover this pattern if it is a concern.

Tip

For some projects, this query may not return any results. Possibly the project you are querying does not have any problems of this kind, but it is also important to make sure the query itself is working properly. One solution is to set up a test project with examples of correct and incorrect code to run the query against

(the C code at the very top of this page makes a good starting point). Another approach is to test each part of the query individually to make sure everything is working.

When you have defined the basic query then you can refine the query to include further coding patterns or to exclude false positives:

Improving the query using the 'SSA' library

The `SSA` library represents variables in static single assignment (SSA) form. In this form, each variable is assigned exactly once and every variable is defined before it is used. The use of SSA variables simplifies queries considerably as much of the local data flow analysis has been done for us. For more information, see Static single assignment on Wikipedia.

Including examples where the string size is stored before use

The query above works for simple cases, but does not identify a common coding pattern where `strlen(string)` is stored in a variable before being passed to `malloc`, as in the following example:

```
int len = strlen(input);
buffer = malloc(len);
```

To identify this case we can use the standard library `SSA.qll` (imported as `semmle.code.cpp.controlflow.SSA`).

This library helps us identify where values assigned to local variables may subsequently be used.

For example, consider the following code:

```
void myFunction(bool condition)
{
    const char* x = "alpha"; // definition #1 of x

    printf("x = %s\n", x); // use #1 of x

    if (condition)
    {
        x = "beta"; // definition #2 of x
    } else {
        x = "gamma"; // definition #3 of x
    }

    printf("x = %s\n", x); // use #2 of x
}
```

If we run the following query on the code, we get three results:

```
import cpp
import semmle.code.cpp.controlflow.SSA

from Variable var, Expr defExpr, Expr use
where exists(SsaDefinition ssaDef |
    defExpr = ssaDef.getAnUltimateDefiningValue(var)
    and use = ssaDef.getAUse(var))
select var, defExpr.getLocation().getStartLine() as dline, use.getLocation().
↪getStartLine() as uline
```

Results:

var	dline	uline
x	3	5
x	9	14
x	11	14

It is often useful to also display the defining expression `defExpr`, if there is one. For example we might adjust the query above as follows:

```
import cpp
import semmle.code.cpp.controlflow.SSA

from Variable var, Expr defExpr, Expr use
where exists(SsaDefinition ssaDef |
    defExpr = ssaDef.getAnUltimateDefiningValue(var)
    and use = ssaDef.getAUse(var))
select var, defExpr.getLocation().getStartLine() as dline, use.getLocation().
↪getStartLine() as uline, defExpr
```

Now we can see the assigned expression in our results:

var	dline	uline	defExpr
x	3	5	alpha
x	9	14	beta
x	11	14	gamma

Extending the query to include allocations passed via a variable

Using our experiments above we can expand our simple implementation of `MallocCall.getAllocatedSize()`. With the following refinement, if the argument is an access to a variable, `getAllocatedSize()` returns a value assigned to that variable instead of the variable access itself:

```
Expr getAllocatedSize() {
    if this.getArgument(0) instanceof VariableAccess then
        exists(LocalScopeVariable v, SsaDefinition ssaDef |
                result = ssaDef.getAnUltimateDefiningValue(v)
                and this.getArgument(0) = ssaDef.getAUse(v))
    else
        result = this.getArgument(0)
}
```

The completed query will now identify cases where the result of `strlen` is stored in a local variable before it is used in a call to `malloc`. Here is the query in full:

```
import cpp

class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
```

(continues on next page)

(continued from previous page)

```
    Expr getAllocatedSize() {
        if this.getArgument(0) instanceof VariableAccess then
            exists(LocalScopeVariable v, SsaDefinition ssaDef |
                result = ssaDef.getAnUltimateDefiningValue(v)
                and this.getArgument(0) = ssaDef.getAUse(v))
        else
            result = this.getArgument(0)
    }
}

from MallocCall malloc
where malloc.getAllocatedSize() instanceof StrlenCall
select malloc, "This allocation does not include space to null-terminate the string."
```

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.9 Using the guards library in C and C++

You can use the CodeQL guards library to identify conditional expressions that control the execution of other parts of a program in C and C++ codebases.

About the guards library

The guards library (defined in `semmle.code.cpp.controlflow.Guards`) provides a class GuardCondition representing Boolean values that are used to make control flow decisions. A `GuardCondition` is considered to guard a basic block if the block can only be reached if the `GuardCondition` is evaluated a certain way. For instance, in the following code, `x < 10` is a `GuardCondition`, and it guards all the code before the return statement.

```
if(x < 10) {
  f(x);
} else if (x < 20) {
  g(x);
} else {
  h(x);
}
return 0;
```

The `controls` predicate

The `controls` predicate helps determine which blocks are only run when the `GuardCondition` evaluates a certain way. `guard.controls(block, testIsTrue)` holds if `block` is only entered if the value of this condition is `testIsTrue`.

In the following code sample, the call to `isValid` controls the calls to `performAction` and `logFailure` but not the return statement.

```
if(isValid(accessToken)) {
  performAction();
  succeeded = 1;
} else {
  logFailure();
  succeeded = 0;
}
return succeeded;
```

In the following code sample, the call to `isValid` controls the body of the `if` statement, and also the code after the `if`.

```
if(!isValid(accessToken)) {
  logFailure();
  return 0;
}
performAction();
return succeeded;
```

The `ensuresEq` and `ensuresLt` predicates

The `ensuresEq` and `ensuresLt` predicates are the main way of determining what, if any, guarantees the `GuardCondition` provides for a given basic block.

The `ensuresEq` predicate

When `ensuresEq(left, right, k, block, true)` holds, then `block` is only executed if `left` was equal to `right + k` at their last evaluation. When `ensuresEq(left, right, k, block, false)` holds, then `block` is only executed if `left` was not equal to `right + k` at their last evaluation.

The `ensuresLt` predicate

When `ensuresLt(left, right, k, block, true)` holds, then `block` is only executed if `left` was strictly less than `right + k` at their last evaluation. When `ensuresLt(left, right, k, block, false)` holds, then `block` is only executed if `left` was greater than or equal to `right + k` at their last evaluation.

In the following code sample, the comparison on the first line ensures that `index` is less than `size` in the "then" block, and that `index` is greater than or equal to `size` in the "else" block.

```
if(index < size) {
  ret = array[index];
} else {
  ret = nullptr
}
return ret;
```

The `comparesEq` and `comparesLt` predicates

The `comparesEq` and `comparesLt` predicates help determine if the `GuardCondition` evaluates to true.

The `comparesEq` predicate

`comparesEq(left, right, k, true, testIsTrue)` holds if `left` equals `right + k` when the expression evaluates to `testIsTrue`.

The `comparesLt` predicate

`comparesLt(left, right, k, isLessThan, testIsTrue)` holds if `left < right + k` evaluates to `isLessThan` when the expression evaluates to `testIsTrue`.

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.10 Using range analysis for C and C++

You can use range analysis to determine the upper or lower bounds on an expression, or whether an expression could potentially over or underflow.

About the range analysis library

The range analysis library (defined in `semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis`) provides a set of predicates for determining constant upper and lower bounds on expressions, as well as recognizing integer overflows. For performance, the library performs automatic widening and therefore may not provide the tightest possible bounds.

Bounds predicates

The `upperBound` and `lowerBound` predicates provide constant bounds on expressions. No conversions of the argument are included in the bound. In the common case that your query needs to take conversions into account, call them on the converted form, such as `upperBound(expr.getFullyConverted())`.

Overflow predicates

`exprMightOverflow` and related predicates hold if the relevant expression might overflow, as determined by the range analysis library. The `convertedExprMightOverflow` family of predicates will take conversions into account.

Example

This query uses `upperBound` to determine whether the result of `snprintf` is checked when used in a loop.

```
from FunctionCall call, DataFlow::Node source, DataFlow::Node sink, Expr convSink
where
  // the call is an snprintf with a string format argument
  call.getTarget().getName() = "snprintf" and
  call.getArgument(2).getValue().regexpMatch(".*%s.*") and

  // the result of the call influences its size argument in later iterations
  TaintTracking::localTaint(source, sink) and
  source.asExpr() = call and
  sink.asExpr() = call.getArgument(1) and

  // there is no fixed bound on the snprintf's size argument
  upperBound(convSink) = typeUpperBound(convSink.getType().getUnspecifiedType()) and
  convSink = call.getArgument(1).getFullyConverted()

select call, upperBound(call.getArgument(1).getFullyConverted())
```

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"

5.1.11 Hash consing and value numbering

You can use specialized CodeQL libraries to recognize expressions that are syntactically identical or compute the same value at runtime in C and C++ codebases.

About the hash consing and value numbering libraries

In C and C++ databases, each node in the abstract syntax tree is represented by a separate object. This allows both analysis and results display to refer to specific appearances of a piece of syntax. However, it is frequently useful to determine whether two expressions are equivalent, either syntactically or semantically.

The hash consing library (defined in `semmle.code.cpp.valuenumbering.HashCons`) provides a mechanism for identifying expressions that have the same syntactic structure. The global value numbering library (defined in `semmle.code.cpp.valuenumbering.GlobalValueNumbering`) provides a mechanism for identifying expressions that compute the same value at runtime. Both libraries partition the expressions in each function into equivalence classes repre-

sented by objects. Each `HashCons` object represents a set of expressions with identical parse trees, while `GVN` objects represent sets of expressions that will always compute the same value. For more information, see Hash consing and Value numbering on Wikipedia.

Example C code

In the following C program, `x + y` and `x + z` will be assigned the same value number but different hash conses.

```
int x = 1;
int y = 2;
int z = y;
if(x + y == x + z) {
  ...
}
```

However, in the next example, the uses of `x + y` will have different value numbers but the same hash cons.

```
int x = 1;
int y = 2;
if(x + y) {
  ...
}

x = 2;

if(x + y) {
  ...
}
```

Value numbering

The value numbering library (defined in `semmle.code.cpp.valuenumbering.GlobalValueNumbering`) provides a mechanism for identifying expressions that compute the same value at runtime. Value numbering is useful when your primary concern is with the values being produced or the eventual machine code being run. For instance, value numbering might be used to determine whether a check is being done against the same value as the operation it is guarding.

The value numbering API

The value numbering library exposes its interface primarily through the `GVN` class. Each instance of `GVN` represents a set of expressions that will always evaluate to the same value. To get an expression in the set represented by a particular `GVN`, use the `getAnExpr()` member predicate.

To get the `GVN` of an `Expr`, use the `globalValueNumber` predicate.

> Note
>
> While the `GVN` class has `toString` and `getLocation` methods, these are only provided as debugging aids. They give the `toString` and `getLocation` of an arbitrary `Expr` within the set.

Why not a predicate?

The obvious interface for this library would be a predicate `equivalent(Expr e1, Expr e2)`. However, this predicate would be very large, with a quadratic number of rows for each set of equivalent expressions. By using a class as an intermediate step, the number of rows can be kept linear, and therefore can be cached.

Example query

This query uses the `GVN` class to identify calls to `strncpy` where the size argument is derived from the source rather than the destination

```
from FunctionCall strncpy, FunctionCall strlen
where
  strncpy.getTarget().hasGlobalName("strncpy") and
  strlen.getTarget().hasGlobalName("strlen") and
  globalValueNumber(strncpy.getArgument(1)) = globalValueNumber(strlen.getArgument(0))␣
↪and
  strlen = strncpy.getArgument(2)
select ci, "This call to strncpy is bounded by the size of the source rather than the␣
↪destination"
```

Hash consing

The hash consing library (defined in `semmle.code.cpp.valuenumbering.HashCons`) provides a mechanism for identifying expressions that have the same syntactic structure. Hash consing is useful when your primary concern is with the text of the code. For instance, hash consing might be used to detect duplicate code within a function.

The hash consing API

The hash consing library exposes its interface primarily through the `HashCons` class. Each instance of `HashCons` represents a set of expressions within one function that have the same syntax (including referring to the same variables). To get an expression in the set represented by a particular `HashCons`, use the `getAnExpr()` member predicate.

> Note
>
> While the `HashCons` class has `toString` and `getLocation` methods, these are only provided as debugging aids. They give the `toString` and `getLocation` of an arbitrary `Expr` within the set.

To get the `HashCons` of an `Expr`, use the `hashCons` predicate.

Example query

```
import cpp
import semmle.code.cpp.valuenumbering.HashCons

from IfStmt outer, IfStmt inner
where
  outer.getElse+() = inner and
  hashCons(outer.getCondition()) = hashCons(inner.getCondition())
select inner.getCondition(), "The condition of this if statement duplicates the␣
```

(continues on next page)

(continued from previous page)

```
↪condition of $@",
  outer.getCondition(), "an enclosing if statement"
```

Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- "*QL language reference*"
- "*CodeQL tools*"
- *Basic query for C and C++ code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for C and C++*: When analyzing C or C++ code, you can use the large collection of classes in the CodeQL library for C and C++.
- *Functions in C and C++*: You can use CodeQL to explore functions in C and C++ code.
- *Expressions, types, and statements in C and C++*: You can use CodeQL to explore expressions, types, and statements in C and C++ code to find, for example, incorrect assignments.
- *Conversions and classes in C and C++*: You can use the standard CodeQL libraries for C and C++ to detect when the type of an expression is changed.
- *Analyzing data flow in C and C++*: You can use data flow analysis to track the flow of potentially malicious or insecure data that can cause vulnerabilities in your codebase.
- *Refining a query to account for edge cases*: You can improve the results generated by a CodeQL query by adding conditions to remove false positive results caused by common edge cases.
- *Detecting a potential buffer overflow*: You can use CodeQL to detect potential buffer overflows by checking for allocations equal to `strlen` in C and C++.
- *Using the guards library in C and C++*: You can use the CodeQL guards library to identify conditional expressions that control the execution of other parts of a program in C and C++ codebases.
- *Using range analysis for C and C++*: You can use range analysis to determine the upper or lower bounds on an expression, or whether an expression could potentially over or underflow.
- *Hash consing and value numbering*: You can use specialized CodeQL libraries to recognize expressions that are syntactically identical or compute the same value at runtime in C and C++ codebases.

5.2 CodeQL for C#

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from C# codebases.

5.2.1 Basic query for C# code

Learn to write and run a simple CodeQL query using LGTM.

About the query

The query we're going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they have an empty then branch. For example, code such as:

```
if (error) { }
```

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **C#** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

   ```
   import csharp

   from IfStmt ifstmt, BlockStmt block
   where ifstmt.getThen() = block and
     block.isEmpty()
   select ifstmt, "This 'if' statement is redundant."
   ```

   LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   > Note
   >
   > Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

Example query results

Note

An ellipsis (...) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import csharp`	Imports the standard CodeQL libraries for C#.	Every query begins with one or more `import` statements.
`from IfStmt ifstmt, BlockStmt block`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `IfStmt` variable for `if` statements • a `BlockStmt` variable for the then block
`where ifstmt.getThen() = block and block.isEmpty()`	Defines a condition on the variables.	`ifstmt.getThen() = block` relates the two variables. The block must be the `then` branch of the `if` statement. `block.isEmpty()` states that the block must be empty (that is, it contains no statements).
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if (...)
{
    ...
}
else if (option == "-verbose")
{
    // nothing to do - handled earlier
}
else
{
    error("unrecognized option");
}
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to ignore empty `then` branches if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Add the following to the where clause:

   ```
   and not exists(ifstmt.getElse())
   ```

   The `where` clause is now:

   ```
   where ifstmt.getThen() = block and
     block.isEmpty() and
     not exists(ifstmt.getElse())
   ```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer included.

See this in the query console

Further reading

- CodeQL queries for C#
- Example queries for C#
- CodeQL library reference for C#
- "*QL language reference*"
- "*CodeQL tools*"

5.2.2 CodeQL library for C#

When you're analyzing a C# program, you can make use of the large collection of classes in the CodeQL library for C#.

About the CodeQL libraries for C#

There is an extensive core library for analyzing CodeQL databases extracted from C# projects. The classes in this library present the data from a database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks. The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `csharp.qll` imports all the core C# library modules, so you can include the complete library by beginning your query with:

```
import csharp
```

Since this is required for all C# queries, it's omitted from code snippets below.

The core library contains all the program elements, including *files*, *types*, methods, *variables*, *statements*, and *expressions*. This is sufficient for most queries, however additional libraries can be imported for bespoke functionality such as control flow and data flow. For information about these additional libraries, see "*CodeQL for C#*."

Class hierarchies

Each section contains a class hierarchy, showing the inheritance structure between CodeQL classes. For example:

- `Expr`
  - `Operation`
    - `ArithmeticOperation`
      - `UnaryArithmeticOperation`
      - `UnaryMinusExpr`, `UnaryPlusExpr`
      - `MutatorOperation`
      - `IncrementOperation`
      - `PreIncrExpr`, `PostIncrExpr`
      - `DecrementOperation`
      - `PreDecrExpr`, `PostDecrExpr`
      - `BinaryArithmeticOperation`
      - `AddExpr`, `SubExpr`, `MulExpr`, `DivExpr`, `RemExpr`

This means that the class `AddExpr` extends class `BinaryArithmeticOperation`, which in turn extends class `ArithmeticOperation` and so on. If you want to query any arithmetic operation, use the class `ArithmeticOperation`, but if you specifically want to limit the query to addition operations, use the class `AddExpr`.

Classes can also be considered to be *sets*, and the `extends` relation between classes defines a subset. Every member of class `AddExpr` is also in the class `BinaryArithmeticOperation`. In general, classes overlap and an entity can be a member of several classes.

This overview omits some of the less important or intermediate classes from the class hierarchy.

Each class has predicates, which are logical propositions about that class. They also define navigable relationships between classes. Predicates are inherited, so for example the `AddExpr` class inherits the predicates `getLeftOperand()`

and `getRightOperand()` from `BinaryArithmeticOperation`, and `getType()` from class `Expr`. This is similar to how methods are inherited in object-oriented programming languages.

In this overview, we present the most common and useful predicates. For the complete list of predicates available on each class, you can look in the CodeQL source code, use autocomplete in the editor, or see the C# reference.

Exercises

Each section in this topic contains exercises to check your understanding.

Exercise 1: Simplify this query:

```
from BinaryArithmeticOperation op
where op instanceof AddExpr
select op
```

(*Answer*)

Files

Files are represented by the class File, and directories by the class Folder. The database contains all of the source files and assemblies used during the compilation.

Class hierarchy

- `File` - any file in the database (including source files, XML and assemblies)
  - `SourceFile` - a file containing source code
- `Folder` - a directory

Predicates

- `getName()` - gets the full path of the file (for example, `C:\Temp\test.cs`).
- `getNumberOfLines()` - gets the number of lines (for source files only).
- `getShortName()` - gets the name of the file without the extension (for example, `test`).
- `getBaseName()` - gets the name and extension of the file (for example, `test.cs`).
- `getParent()` - gets the parent directory.

Examples

Count the number of source files:

```
select count(SourceFile f)
```

Count the number of lines of code, excluding the directory `external`:

```
select sum(SourceFile f |
  not exists(Folder ext | ext.getShortName() = "external" |
             ext.getAFolder*().getAFile() = f) |
  f.getNumberOfLines())
```

Exercises

Exercise 2: Write a query to find the source file with the largest number of lines. Hint: Find the source file with the same number of lines as the `max` number of lines in any file. (*Answer*)

Elements

The class Element is the base class for all parts of a C# program, and it's the root of the element class hierarchy. All program elements (such as types, methods, statements, and expressions) ultimately derive from this common base class.

`Element` forms a hierarchical structure of the program, which can be navigated using the `getParent()` and `getChild()` predicates. This is much like an abstract syntax tree, and also applies to elements in assemblies.

Predicates

The `Element` class provides common functionality for all program elements, including:

- `getLocation()` - gets the text span in the source code.
- `getFile()` - gets the `File` containing the `Element`.
- `getParent()` - gets the parent `Element`, if any.
- `getAChild()` - gets a child `Element` of this element, if any.

Examples

To list all elements in `Main.cs`, their QL class and location:

```
from Element e
where e.getFile().getShortName() = "Main"
select e, e.getAQlClass(), e.getLocation()
```

Note that `getAQlClass()` is available on all entities and is a useful way to figure out the QL class of something. Often the same element will have several classes which are all returned by `getAQlClass()`.

Locations

Location represents a section of text in the source code, or an assembly. All elements have a `Location` obtained by their `getLocation()` predicate. A `SourceLocation` represents a span of text in source code, whereas an `Assembly` location represents a referenced assembly.

Sometimes elements have several locations, for example if they occur in both source code and an assembly. In this case, only the `SourceLocation` is returned.

Class hierarchy

- `Location`
  - `SourceLocation`
  - `Assembly`

Predicates

Some predicates of `Location` include:

- `getFile()` - gets the `File`.
- `getStartLine()` - gets the first line of the text.
- `getEndLine()` - gets the last line of the text.
- `getStartColumn()` - gets the column of the start of the text.
- `getEndColumn()` - gets the column of the end of the text.

Examples

Find all elements that are one character wide:

```
from Element e, Location l
where l = e.getLocation()
  and l.getStartLine() = l.getEndLine()
  and l.getStartColumn() = l.getEndColumn()
select e, "This element is a single character."
```

Declarations

Declaration is the common class of all entities defined in the program, such as types, methods, variables etc. The database contains all declarations from the source code and all referenced assemblies.

Class hierarchy

- `Element`
  - `Declaration`
    - `Callable`
    - `UnboundGeneric`
    - `ConstructedGeneric`
    - `Modifiable` - a declaration which can have a modifier (for example `public`)
      - `Member` - a declaration that is member of a type
    - `Assignable` - an element that can be assigned to
      - `Variable`
      - `Property`

        - Indexer
        - Event

Predicates

Useful member predicates on `Declaration` include:

- `getDeclaringType()` - gets the type containing the declaration, if any.
- `getName()`/`hasName(string)` - gets the name of the declared entity.
- `isSourceDeclaration()` - whether the declaration is source code and is not a constructed type/method.
- `getSourceDeclaration()` - gets the original (unconstructed) declaration.

Examples

Find declarations containing a username:

```
from Declaration decl
where decl.getName().regexpMatch("[uU]ser([Nn]ame)?")
select decl, "A username."
```

Variables

The class Variable represents C# variables, such as fields, parameters and local variables. The database contains all variables from the source code, as well as all fields and parameters from assemblies referenced by the program.

Class hierarchy

- `Element`
    - `Declaration`
        - `Variable` - any type of variable
            - `Field` - a field in a `class`/`struct`
            - `MemberConstant` - a `const` field
            - `EnumConstant` - a field in an `enum`
            - `LocalScopeVariable` - a variable whose scope is limited to a single `Callable`
            - `LocalVariable` - a local variable in a `Callable`
            - `LocalConstant` - a locally defined constant in a `Callable`
            - `Parameter` - a parameter to a `Callable`

Predicates

Some common predicates on `Variable` are:

- `getType()` - gets the `Type` of this variable.
- `getAnAccess()` - gets an expression that accesses (reads or writes) this variable, if any.
- `getAnAssignedValue()` - gets an expression that is assigned to this variable, if any.
- `getInitializer()` - gets the expression used to initialize the variable, if any.

Examples

Find all unused local variables:

```
from LocalVariable v
where not exists(v.getAnAccess())
select v, "This local variable is unused."
```

Types

Types are represented by the CodeQL class Type and consist of builtin types, interfaces, classes, structs, enums, and type parameters. The database contains types from the program and all referenced assemblies including mscorlib and the .NET framework.

The builtin types (`object`, `int`, `double` etc.) have corresponding types (`System.Object`, `System.Int32` etc.) in mscorlib.

Class `ValueOrRefType` represents defined types, such as a `class`, `struct`, `interface` or `enum`.

Class hierarchy

- `Element`
  - `Declaration`
    - `Modifiable` - a declaration which can have a modifier (for example `public`)
      - `Member` - a declaration that is member of a type
      - `Type` - all types
      - `ValueOrRefType` - a defined type
      - `ValueType` - a value type (see below for further hierarchy)
      - `RefType` - a reference type (see below for further hierarchy)
      - `NestedType` - a type defined in another type
      - `VoidType` - `void`
      - `PointerType` - a pointer type

The `ValueType` class extends further:

- `ValueType` - a value type
  - `SimpleType` - a simple built-in type

        * BoolType - bool
        * CharType - char
        * IntegralType
            - UnsignedIntegralType
            - ByteType - byte
            - UShortType - unsigned short/System.UInt16
            - UIntType - unsigned int/System.UInt32
            - ULongType - unsigned long/System.UInt64
            - SignedIntegralType
            - SByteType - signed byte
            - ShortType - short/System.Int16
            - IntType - int/System.Int32
            - LongType - long/System.Int64
            - FloatingPointType
            - FloatType - float/System.Single
            - DoubleType - double/System.Double
            - DecimalType - decimal/System.Decimal
        * Enum - an enum
        * Struct - a struct
        * NullableType
        * ArrayType

The RefType class extends further:

- RefType
    - Class - a class
        * AnonymousClass
        * ObjectType - object/System.Object
        * StringType - string/System.String
    - Interface - an interface
    - DelegateType
    - NullType - the type of null
    - DynamicType - dynamic
- NestedType - a type defined in another type

These class hierarchies omit generic types for simplicity.

Predicates

Useful members of `ValueOrRefType` include:

- `getQualifiedName()/hasQualifiedName(string)` - gets the qualified name of the type (for example, `"System.String"`).
- `getABaseInterface()` - gets an immediate interface of this type, if any.
- `getABaseType()` - gets an immediate base class or interface of this type, if any.
- `getBaseClass()` - gets the immediate base class of this type, if any.
- `getASubType()` - gets an immediate subtype, a type which directly inherits from this type, if any.
- `getAMember()` - gets any member (field/method/property etc), if any.
- `getAMethod()` - gets a method, if any.
- `getAProperty()` - gets a property, if any.
- `getAnIndexer()` - gets an indexer, if any.
- `getAnEvent()` - gets an event, if any.
- `getAnOperator()` - gets an operator, if any.
- `getANestedType()` - gets a nested type.
- `getNamespace()` - gets the enclosing namespace.

Examples

Find all members of `System.Object`:

```
from ObjectType object
select object.getAMember()
```

Find all types which directly implement `System.Collections.IEnumerable`:

```
from Interface ienumerable
where ienumerable.hasQualifiedName("System.Collections.IEnumerable")
select ienumerable.getASubType()
```

List all simple types in the `System` namespace:

```
select any(SimpleType t | t.getNamespace().hasName("System"))
```

Find all variables of type `PointerType`:

```
from Variable v
where v.fromSource()
  and v.getType() instanceof PointerType
select v
```

List all classes in source files:

```
from Class c
where c.fromSource()
select c
```

Exercises

Exercise 3: Write a query to list the methods in `string`. (*Answer*)

Exercise 4: Adapt the example to find all types which indirectly implement `IEnumerable`. (*Answer*)

Exercise 5: Write a query to find all classes starting with the letter `A`. (*Answer*)

Callables

Callables are represented by the class Callable and are anything that can be called independently, such as methods, constructors, destructors, operators, anonymous functions, indexers, and property accessors.

The database contains all of the callables in your program and in all referenced assemblies.

Class hierarchy

- `Element`
  - `Declaration`
    - `Callable`
      - `Method`
      - `ExtensionMethod`
      - `Constructor`
      - `StaticConstructor`
      - `InstanceConstructor`
      - `Destructor`
      - `Operator`
      - `UnaryOperator`
      - `PlusOperator`, `MinusOperator`, `NotOperator`, `ComplementOperator`, `IncrementOperator`, `DecrementOperator`, `FalseOperator`, `TrueOperator`
      - `BinaryOperator`
      - `AddOperator`, `SubOperator`, `MulOperator`, `DivOperator`, `RemOperator`, `AndOperator`, `OrOperator`, `XorOperator`, `LShiftOperator`, `RShiftOperator`, `EQOperator`, `NEOperator`, `LTOperator`, `GTOperator`, `LEOperator`, `GEOperator`
      - `ConversionOperator`
      - `ImplicitConversionOperator`
      - `ExplicitConversionOperator`
      - `AnonymousFunctionExpr`
      - `LambdaExpr`
      - `AnonymousMethodExpr`
      - `Accessor`
      - `Getter`

- `Setter`
- `EventAccessor`
- `AddEventAccessor`, `RemoveEventAccessor`

Predicates

Here are a few useful predicates on the `Callable` class:

- `getParameter(int)`/`getAParameter()` - gets a parameter.
- `calls(Callable)` - whether there's a direct call from one callable to another.
- `getReturnType()` - gets the return type.
- `getBody()`/`getExpressionBody()` - gets the body of the callable.

Since `Callable` extends `Declaration`, it also has predicates from `Declaration`, such as:

- `getName()`/`hasName(string)`
- `getSourceDeclaration()`
- `getName()`
- `getDeclaringType()`

Methods have additional predicates, including:

- `getAnOverridee()` - gets a method that is immediately overridden by this method.
- `getAnOverrider()` - gets a method that immediately overrides this method.
- `getAnImplementee()` - gets an interface method that is immediately implemented by this method.
- `getAnImplementor()` - gets a method that immediately implements this interface method.

Examples

List all types which override `ToString`:

```
from Method m
where m.hasName("ToString")
select m
```

Find methods that look like `ToString` methods but don't override `Object.ToString`:

```
from Method toString, Method falseToString
where toString.hasQualifiedName("System.Object.ToString")
 and falseToString.getName().toLowerCase() = "tostring"
 and not falseToString.overrides*(toString)
 and falseToString.getNumberOfParameters() = 0
select falseToString, "This method looks like it overrides Object.ToString but it doesn
↪'t."
```

Find all methods which take a pointer type:

```
from Method m
where m.getAParameter().getType() instanceof PointerType
select m, "This method uses pointers."
```

Find all classes which have a destructor but aren't disposable:

```
from Class c
where c.getAMember() instanceof Destructor
  and not c.getABaseType*().hasQualifiedName("System.IDisposable")
select c, "This class has a destructor but is not IDisposable."
```

Find `Main` methods which are not `private`:

```
from Method m
where m.hasName("Main")
  and not m.isPrivate()
select m, "Main method should be private."
```

Statements

Statements are represented by the class Stmt and make up the body of methods (and other callables). The database contains all statements in the source code, but does not contain any statements from referenced assemblies where the source code is not available.

Class hierarchy

- Element
  - ControlFlowElement
    - Stmt
      - BlockStmt - { ... }
      - ExprStmt
      - SelectionStmt
      - IfStmt - if
      - SwitchStmt - switch
      - LabeledStmt
      - ConstCase
      - DefaultCase - default
      - LabelStmt
      - LoopStmt
      - WhileStmt - while(...) { ... }
      - DoStmt - do { ... } while(...)
      - ForStmt - for
      - ForEachStmt - foreach

- JumpStmt
- BreakStmt - break
- ContinueStmt - continue
- GotoStmt - goto
- GotoLabelStmt
- GotoCaseStmt
- GotoDefaultStmt
- ThrowStmt - throw
- ReturnStmt - return
- YieldStmt
- YieldBreakStmt - yield break
- YieldReturnStmt - yield return
- TryStmt - try
- CatchClause - catch
- SpecificCatchClause
- GeneralCatchClause
- CheckedStmt - checked
- UncheckedStmt - unchecked
- LockStmt - lock
- UsingStmt - using
- LocalVariableDeclStmt
- LocalConstantDeclStmt
- EmptyStmt - ;
- UnsafeStmt - unsafe
- FixedStmt - fixed

Examples

Find long methods:

```
from Method m
where m.getBody().(BlockStmt).getNumberOfStmts() >= 100
select m, "This is a long method!"
```

Find `for(;;)`:

```
from ForStmt for
where not exists(for.getAnInitializer())
  and not exists(for.getUpdate(_))
  and not exists(for.getCondition())
select for, "Infinite loop."
```

Find `catch(NullDefererenceException)`:

```
from SpecificCatchClause catch
where catch.getCaughtExceptionType().hasQualifiedName("System.NullReferenceException")
select catch, "Catch NullReferenceException."
```

Find an `if` statement with a constant condition:

```
from IfStmt ifStmt
where ifStmt.getCondition().hasValue()
select ifStmt, "This 'if' statement is constant."
```

Find an `if` statement with an empty "then" block:

```
from IfStmt ifStmt
where ifStmt.getThen().(BlockStmt).isEmpty()
select ifStmt, "If statement with empty 'then' block."
```

The `(BlockStmt)` is an inline cast, which restricts the query to cases where the result of `getThen()` has the QL class `BlockStmt`, and allows predicates on `BlockStmt` to be used, such as `isEmpty()`.

Exercises

Exercise 6: Write a query to list all empty methods. (*Answer*)

Exercise 7: Modify the last example to also detect empty statements (`;`) in the "then" block. (*Answer*)

Exercise 8: Modify the last example to exclude chains of `if` statements, where the `else` part is another `if` statement. (*Answer*)

Expressions

The Expr class represents all C# expressions in the program. An expression is something producing a value such as `a+b` or `new List<int>()`. The database contains all expressions from the source code, but no expressions from referenced assemblies where the source code is not available.

The `Access` class represents any use or cross-reference of another `Declaration` such a variable, property, method or field. The `getTarget()` predicate gets the declaration being accessed.

The `Call` class represents a call to a `Callable`, for example to a `Method` or an `Accessor`, and the `getTarget()` method gets the `Callable` being called. The `Operation` class consists of arithmetic, bitwise operations and logical operations.

Some expressions use a qualifier, which is the object on which the expression operates. A typical example is a `MethodCall`. In this case, the `getQualifier()` predicate is used to get the expression on the left of the `.`, and `getArgument(int)` is used to get the arguments of the call.

Class hierarchy

- Element
  - ControlFlowElement
    - Expr
      - LocalVariableDeclExpr
      - LocalConstantDeclExpr
      - Operation
      - UnaryOperation
      - SizeofExpr, PointerIndirectionExpr, AddressOfExpr
      - BinaryOperation
      - ComparisonOperation
      - EqualityOperation
      - EQExpr, NEExpr
      - RelationalOperation
      - GTExpr, LTExpr, GEExpr, LEExpr
      - Assignment
      - AssignOperation
      - AddOrRemoveEventExpr
      - AddEventExpr
      - RemoveEventExpr
      - AssignArithmeticOperation
      - AssignAddExpr, AssignSubExpr, AssignMulExpr, AssignDivExpr, AssignRemExpr
      - AssignBitwiseOperation
      - AssignAndExpr, AssignOrExpr, AssignXorExpr, AssignLShiftExpr, AssignRShiftExpr
      - AssignExpr
      - MemberInitializer
      - ArithmeticOperation
      - UnaryArithmeticOperation
      - UnaryMinusExpr, UnaryPlusExpr
      - MutatorOperation
      - IncrementOperation
      - PreIncrExpr, PostIncrExpr
      - DecrementOperation
      - PreDecrExpr, PostDecrExpr
      - BinaryArithmeticOperation

- AddExpr, SubExpr, MulExpr, DivExpr, RemExpr
- BitwiseOperation
- UnaryBitwiseOperation
- ComplementOperation
- BinaryBitwiseOperation
- LShiftExpr, RShiftExpr, BitwiseAndExpr, BitwiseOrExpr, BitwiseXorExpr
- LogicalOperation
- UnaryLogicalOperation
- LogicalNotOperation
- BinaryLogicalOperation
- LogicalAndExpr, LogicalOrExpr, NullCoalescingExpr
- ConditionalExpr
- ParenthesisedExpr, CheckedExpr, UncheckedExpr, IsExpr, AsExpr, CastExpr, TypeofExpr, DefaultValueExpr, AwaitExpr, NameofExpr, InterpolatedStringExpr
- Access
- ThisAccess
- BaseAccess
- MemberAccess
- MethodAccess
- VirtualMethodAccess
- FieldAccess, PropertyAccess, IndexerAccess, EventAccess, MethodAccess
- AssignableAccess
- VariableAccess
- ParameterAccess
- LocalVariableAccess
- LocalScopeVariableAccess
- FieldAccess
- MemberConstantAccess
- PropertyAccess
- TrivialPropertyAccess
- VirtualPropertyAccess
- IndexerAccess
- VirtualIndexerAccess
- EventAccess
- VirtualEventAccess
- TypeAccess

- ArrayAccess
- Call
- PropertyCall
- IndexerCall
- EventCall
- MethodCall
- VirtualMethodCall
- ElementInitializer
- ConstructorInitializer
- OperatorCall
- MutatorOperatorCall
- DelegateCall
- ObjectCreation
- DefaultValueTypeObjectCreation
- TypeParameterObjectCreation
- AnonymousObjectCreation
- ObjectOrCollectionInitializer
- ObjectInitializer
- CollectionInitializer
- DelegateCreation
- ExplicitDelegateCreation, ImplicitDelegateCreation
- ArrayInitializer
- ArrayCreation
- AnonymousFunctionExpr
- LambdaExpr
- AnonymousMethodExpr
- Literal
- BoolLiteral, CharLiteral, IntegerLiteral, IntLiteral, LongLiteral, UIntLiteral, ULongLiteral, RealLiteral, FloatLiteral, DoubleLiteral, DecimalLiteral, StringLiteral, NullLiteral

Predicates

Useful predicates on `Expr` include:

- `getType()` - gets the `Type` of the expression.
- `getValue()` - gets the compile-time constant, if any.
- `hasValue()` - whether the expression has a compile-time constant.
- `getEnclosingStmt()` - gets the statement containing the expression, if any.
- `getEnclosingCallable()` - gets the callable containing the expression, if any.
- `stripCasts()` - remove all explicit or implicit casts.
- `isImplicit()` - whether the expression was implicit, such as an implicit `this` qualifier (`ThisAccess`).

Examples

Find calls to `String.Format` with just one argument:

```
from MethodCall c
where c.getTarget().hasQualifiedName("System.String.Format")
  and c.getNumberOfArguments() = 1
select c, "Missing arguments to 'String.Format'."
```

Find all comparisons of floating point values:

```
from ComparisonOperation cmp
where (cmp instanceof EQExpr or cmp instanceof NEExpr)
  and cmp.getAnOperand().getType() instanceof FloatingPointType
select cmp, "Comparison of floating point values."
```

Find hard-coded passwords:

```
from Variable v, string value
where v.getName().regexpMatch("[pP]ass(word|wd|)")
  and value = v.getAnAssignedValue().getValue()
select v, "Hard-coded password '" + value + "'."
```

Exercises

Exercise 9: Limit the previous query to string types. Exclude empty passwords or null passwords. (*Answer*)

Attributes

C# attributes are represented by the class Attribute. They can be present on many C# elements, such as classes, methods, fields, and parameters. The database contains attributes from the source code and all assembly references.

The attribute of any `Element` can be obtained via `getAnAttribute()`, whereas if you have an attribute, you can find its element via `getTarget()`. These two query fragments are identical:

```
attribute = element.getAnAttribute()
element = attribute.getTarget()
```

Class hierarchy

- `Element`
  - `Attribute`

Predicates

- `getTarget()` - gets the `Element` to which this attribute applies.
- `getArgument(int)` - gets the given argument of the attribute.
- `getType()` - gets the type of this attribute. Note that the class name must end in `"Attribute"`.

Examples

Find all obsolete elements:

```
from Element e, Attribute attribute
where e = attribute.getTarget()
  and attribute.getType().hasName("ObsoleteAttribute")
select e, "This is obsolete because " + attribute.getArgument(0).getValue()
```

Model NUnit test fixtures:

```
class TestFixture extends Class
{
  TestFixture() {
    this.getAnAttribute().getType().hasName("TestFixtureAttribute")
  }

  TestMethod getATest() {
    result = this.getAMethod()
  }
}

class TestMethod extends Method
{
  TestMethod() {
    this.getAnAttribute().getType().hasName("TestAttribute")
  }
```

(continues on next page)

(continued from previous page)

```
}

from TestFixture f
select f, f.getATest()
```

Exercises

Exercise 10: Write a query to find just obsolete methods. (*Answer*)

Exercise 11: Write a query to find all places where the `Obsolete` attribute is used without a reason string (that is, `[Obsolete]`). (*Answer*)

Exercise 12: In the first example, what happens if the `Obsolete` attribute doesn't have a reason string? How could the query be fixed to accommodate this? (*Answer*)

Answers

Exercise 1

```
from AddExpr op
select op
```

or

```
select any(AddExpr op)
```

Exercise 2

```
from File f
where f.getNumberOfLines() = max(any(File g).getNumberOfLines())
select f
```

Exercise 3

```
from StringType s
select s.getAMethod()
```

Exercise 4

```
from Interface ienumerable
where ienumerable.hasQualifiedName("System.Collections.IEnumerable")
select ienumerable.getASubType*()
```

Exercise 5

```
from Class a
where a.getName().toLowerCase().matches("a%")
select a
```

Exercise 6

```
select any(Method m | m.getBody().(BlockStmt).isEmpty())
```

Exercise 7

```
from IfStmt ifStmt
where ifStmt.getThen().(BlockStmt).isEmpty() or ifStmt.getThen() instanceof EmptyStmt
select ifStmt, "If statement with empty 'then' block."
```

Exercise 8

```
from IfStmt ifStmt
where (ifStmt.getThen().(BlockStmt).isEmpty() or ifStmt.getThen() instanceof EmptyStmt)
  and not ifStmt.getElse() instanceof IfStmt
select ifStmt, "If statement with empty 'then' block."
```

Exercise 9

```
from Variable v, StringLiteral value
where v.getName().regexpMatch("[pP]ass(word|wd|)")
  and value = v.getAnAssignedValue()
  and value.getValue() != ""
select v, "Hard-coded password '" + value.getValue() + "'."
```

Exercise 10

```
from Method method, Attribute attribute
where method = attribute.getTarget()
  and attribute.getType().hasName("ObsoleteAttribute")
select method, "This is obsolete because " + attribute.getArgument(0).getValue()
```

Exercise 11

```
from Attribute attribute
where attribute.getType().hasName("ObsoleteAttribute")
  and not exists(attribute.getArgument(0))
select attribute, "Missing reason in 'Obsolete' attribute."
```

Exercise 12

The query does not return results where the argument is missing.

Here is the fixed version:

```
from Element e, Attribute attribute, string reason
where e = attribute.getTarget()
  and attribute.getType().hasName("ObsoleteAttribute")
  and if exists(attribute.getArgument(0))
    then reason = attribute.getArgument(0).getValue()
    else reason = "(not given)"
select e, "This is obsolete because " + reason
```

Further reading

- CodeQL queries for C#
- Example queries for C#
- CodeQL library reference for C#
- "*QL language reference*"
- "*CodeQL tools*"

5.2.3 Analyzing data flow in C#

You can use CodeQL to track the flow of data through a C# program to its use.

About this article

This article describes how data flow analysis is implemented in the CodeQL libraries for C# and includes examples to help you write your own data flow queries. The following sections describe how to use the libraries for local data flow, global data flow, and taint tracking. For a more general introduction to modeling data flow, see "*About data flow analysis*."

Local data flow

Local data flow is data flow within a single method or callable. Local data flow is easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. `Node`s are divided into expression nodes (`ExprNode`) and parameter nodes (`ParameterNode`). You can map between data flow nodes and expressions/parameters using the member predicates `asExpr` and `asParameter`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the parameter corresponding to this node, if any. */
  Parameter asParameter() { ... }

 ...
}
```

or using the predicates `exprNode` and `parameterNode`:

```
/**
 * Gets the node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }

/**
 * Gets the node corresponding to the value of parameter `p` at function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively, by using the + and * operators, or you can use the predefined recursive predicate `localFlow`.

For example, you can find flow from a parameter `source` to an expression `sink` in zero or more local steps:

```
DataFlow::localFlow(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
var temp = x;
var y = temp + ", " + temp;
```

If `x` is a tainted string then `y` is also tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint propagation edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively, by using the `+` and `*` operators, or you can use the predefined recursive predicate `localTaint`.

For example, you can find taint propagation from a parameter `source` to an expression `sink` in zero or more local steps:

```
TaintTracking::localTaint(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Examples

This query finds the filename passed to `System.IO.File.Open`:

```
import csharp

from Method fileOpen, MethodCall call
where fileOpen.hasQualifiedName("System.IO.File.Open")
  and call.getTarget() = fileOpen
select call.getArgument(0)
```

Unfortunately this will only give the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import csharp

from Method fileOpen, MethodCall call, Expr src
where fileOpen.hasQualifiedName("System.IO.File.Open")
  and call.getTarget() = fileOpen
  and DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(call.
↪getArgument(0)))
select src
```

Then we can make the source more specific, for example an access to a public parameter. This query finds instances where a public parameter is used to open a file:

```
import csharp

from Method fileOpen, MethodCall call, Parameter p
where fileOpen.hasQualifiedName("System.IO.File.Open")
  and call.getTarget() = fileOpen
  and DataFlow::localFlow(DataFlow::parameterNode(p), DataFlow::exprNode(call.
↪getArgument(0)))
  and call.getEnclosingCallable().(Member).isPublic()
select p, "Opening a file from a public method."
```

This query finds calls to `String.Format` where the format string isn't hard-coded:

```
import csharp

from Method format, MethodCall call, Expr formatString
where format.hasQualifiedName("System.String.Format")
  and call.getTarget() = format
  and formatString = call.getArgument(0)
  and formatString.getType() instanceof StringType
  and not exists(StringLiteral source | DataFlow::localFlow(DataFlow::exprNode(source),␣
↪DataFlow::exprNode(formatString)))
select call, "Argument to 'string.Format' isn't hard-coded."
```

Exercises

Exercise 1: Write a query that finds all hard-coded strings used to create a `System.Uri`, using local data flow. (*Answer*)

Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see *Creating path queries*.

Using global data flow

The global data flow library is used by extending the class `DataFlow::Configuration`:

```
import csharp

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "..." }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource` - defines where data may flow from.
- `isSink` - defines where data may flow to.
- `isBarrier` - optionally, restricts the data flow.

- `isAdditionalFlowStep` - optionally, adds additional flow steps.

The characteristic predicate (`MyDataFlowConfiguration()`) defines the name of the configuration, so `"..."` must be replaced with a unique name.

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguation dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Dataflow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow what local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. The global taint tracking library is used by extending the class `TaintTracking::Configuration`:

```
import csharp

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "..." }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource` - defines where taint may flow from.
- `isSink` - defines where taint may flow to.
- `isSanitizer` - optionally, restricts the taint flow.
- `isAdditionalTaintStep` - optionally, adds additional taint steps.

Similar to global data flow, the characteristic predicate (`MyTaintTrackingConfiguration()`) defines the unique name of the configuration and the taint analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Flow sources

The data flow library contains some predefined flow sources. The class `PublicCallableParameterFlowSource` (defined in module `semmle.code.csharp.dataflow.flowsources.PublicCallableParameter`) represents data flow from public parameters, which is useful for finding security problems in a public API.

The class `RemoteFlowSource` (defined in module `semmle.code.csharp.dataflow.flowsources.Remote`) represents data flow from remote network inputs. This is useful for finding security problems in networked services.

Example

This query shows a data flow configuration that uses all public API parameters as data sources:

```
import csharp
import semmle.code.csharp.dataflow.flowsources.PublicCallableParameter

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() {
    this = "..."
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof PublicCallableParameterFlowSource
  }

  ...
}
```

Class hierarchy

- `DataFlow::Configuration` - base class for custom global data flow analysis.
- `DataFlow::Node` - an element behaving as a data flow node.
  - `DataFlow::ExprNode` - an expression behaving as a data flow node.
  - `DataFlow::ParameterNode` - a parameter data flow node representing the value of a parameter at function entry.
    - `PublicCallableParameter` - a parameter to a public method/callable in a public class.
  - `RemoteFlowSource` - data flow from network/remote input.
    - `AspNetRemoteFlowSource` - data flow from remote ASP.NET user input.
      - `AspNetQueryStringRemoteFlowSource` - data flow from `System.Web.HttpRequest`.
      - `AspNetUserInputRemoveFlowSource` - data flow from `System.Web.IO.WebControls.TextBox`.
    - `WcfRemoteFlowSource` - data flow from a WCF web service.
    - `AspNetServiceRemoteFlowSource` - data flow from an ASP.NET web service.
- `TaintTracking::Configuration` - base class for custom global taint tracking analysis.

Examples

This data flow configuration tracks data flow from environment variables to opening files:

```
import csharp

class EnvironmentToFileConfiguration extends DataFlow::Configuration {
  EnvironmentToFileConfiguration() { this = "Environment opening files" }

  override predicate isSource(DataFlow::Node source) {
    exists(Method m |
      m = source.asExpr().(MethodCall).getTarget() and
      m.hasQualifiedName("System.Environment.GetEnvironmentVariable")
    )
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodCall mc |
      mc.getTarget().hasQualifiedName("System.IO.File.Open") and
      sink.asExpr() = mc.getArgument(0)
    )
  }
}

from Expr environment, Expr fileOpen, EnvironmentToFileConfiguration config
where config.hasFlow(DataFlow::exprNode(environment), DataFlow::exprNode(fileOpen))
select fileOpen, "This 'File.Open' uses data from $@.",
  environment, "call to 'GetEnvironmentVariable'"
```

Exercises

Exercise 2: Find all hard-coded strings passed to `System.Uri`, using global data flow. (*Answer*)

Exercise 3: Define a class that represents flow sources from `System.Environment.GetEnvironmentVariable`. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query to find all global data flow from `System.Environment.GetEnvironmentVariable` to `System.Uri`. (*Answer*)

Extending library data flow

Library data flow defines how data flows through libraries where the source code is not available, such as the .NET Framework, third-party libraries or proprietary libraries.

To define new library data flow, extend the class `LibraryTypeDataFlow` from the module `semmle.code.csharp.dataflow.LibraryTypeDataFlow`. Override the predicate `callableFlow` to define how data flows through the methods in the class. `callableFlow` has the signature

```
predicate callableFlow(CallableFlowSource source, CallableFlowSink sink,␣
↪SourceDeclarationCallable callable, boolean preservesValue)
```

- `callable` - the `Callable` (such as a method, constructor, property getter or setter) performing the data flow.
- `source` - the data flow input.

- `sink` - the data flow output.
- `preservesValue` - whether the flow step preserves the value, for example if `x` is a string then `x.ToString()` preserves the value where as `x.ToLower()` does not.

Class hierarchy

- `Callable` - a callable (methods, accessors, constructors etc.)
  - `SourceDeclarationCallable` - an unconstructed callable.
- `CallableFlowSource` - the input of data flow into the callable.
  - `CallableFlowSourceQualifier` - the data flow comes from the object itself.
  - `CallableFlowSourceArg` - the data flow comes from an argument to the call.
- `CallableFlowSink` - the output of data flow from the callable.
  - `CallableFlowSinkQualifier` - the output is to the object itself.
  - `CallableFlowSinkReturn` - the output is returned from the call.
  - `CallableFlowSinkArg` - the output is an argument.
  - `CallableFlowSinkDelegateArg` - the output flows through a delegate argument (for example, LINQ).

Example

This example is adapted from `LibraryTypeDataFlow.qll`. It declares data flow through the class `System.Uri`, including the constructor, the `ToString` method, and the properties `Query`, `OriginalString`, and `PathAndQuery`.

```
import semmle.code.csharp.dataflow.LibraryTypeDataFlow
import semmle.code.csharp.frameworks.System

class SystemUriFlow extends LibraryTypeDataFlow, SystemUriClass {
  override predicate callableFlow(CallableFlowSource source, CallableFlowSink sink,␣
↪SourceDeclarationCallable c, boolean preservesValue) {
    (
      constructorFlow(source, c) and
      sink instanceof CallableFlowSinkQualifier
      or
      methodFlow(c) and
      source instanceof CallableFlowSourceQualifier and
      sink instanceof CallableFlowSinkReturn
      or
      exists(Property p |
        propertyFlow(p) and
        source instanceof CallableFlowSourceQualifier and
        sink instanceof CallableFlowSinkReturn and
        c = p.getGetter()
      )
    )
    and
    preservesValue = false
  }
```

(continues on next page)

(continued from previous page)

```
  private predicate constructorFlow(CallableFlowSourceArg source, Constructor c) {
    c = getAMember()
    and
    c.getParameter(0).getType() instanceof StringType
    and
    source.getArgumentIndex() = 0
  }

  private predicate methodFlow(Method m) {
    m.getDeclaringType() = getABaseType*()
    and
    m = getSystemObjectClass().getToStringMethod().getAnOverrider*()
  }

  private predicate propertyFlow(Property p) {
    p = getPathAndQueryProperty()
    or
    p = getQueryProperty()
    or
    p = getOriginalStringProperty()
  }
}
```

This defines a new class `SystemUriFlow` which extends `LibraryTypeDataFlow` to add another case. It extends `SystemUriClass` (the class representing `System.Uri`, defined in the module `semmle.code.csharp.frameworks.System`) to access methods such as `getQueryProperty`.

The predicate `callableFlow` declares data flow through `System.Uri`. The first case (`constructorFlow`) declares data flow from the first argument of the constructor to the object itself (`CallableFlowSinkQualifier`).

The second case declares data flow from the object (`CallableFlowSourceQualifier`) to the result of calling `ToString` on the object (`CallableFlowSinkReturn`).

The third case declares data flow from the object (`CallableFlowSourceQualifier`) to the return (`CallableFlowSinkReturn`) of the getters for the properties `PathAndQuery`, `Query` and `OriginalString`. Note that the properties (`getPathAndQueryProperty`, `getQueryProperty` and `getOriginalStringProperty`) are inherited from the class `SystemUriClass`.

In all three cases `preservesValue = false`, which means that these steps will only be included in taint tracking, not in (normal) data flow.

Exercises

Exercise 5: In `System.Uri`, what other properties could expose data? How could they be added to `SystemUriFlow`? (*Answer*)

Exercise 6: Implement the data flow for the class `System.Exception`. (*Answer*)

Answers

Exercise 1

```
import csharp

from Expr src, Call c
where DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(c.getArgument(0)))
  and c.getTarget().(Constructor).getDeclaringType().hasQualifiedName("System.Uri")
  and src.hasValue()
select src, "This string constructs 'System.Uri' $@.", c, "here"
```

Exercise 2

```
import csharp

class Configuration extends DataFlow::Configuration {
  Configuration() { this="String to System.Uri" }

  override predicate isSource(DataFlow::Node src) {
    src.asExpr().hasValue()
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call c | c.getTarget().(Constructor).getDeclaringType().hasQualifiedName(
↪"System.Uri")
    and sink.asExpr()=c.getArgument(0))
  }
}

from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This string constructs a 'System.Uri' $@.", sink, "here"
```

Exercise 3

```
class EnvironmentVariableFlowSource extends DataFlow::ExprNode {
  EnvironmentVariableFlowSource() {
    this.getExpr().(MethodCall).getTarget().hasQualifiedName("System.Environment.
↪GetEnvironmentVariable")
  }
}
```

Exercise 4

```
import csharp

class EnvironmentVariableFlowSource extends DataFlow::ExprNode {
  EnvironmentVariableFlowSource() {
    this.getExpr().(MethodCall).getTarget().hasQualifiedName("System.Environment.
↪GetEnvironmentVariable")
  }
}

class Configuration extends DataFlow::Configuration {
  Configuration() { this="Environment to System.Uri" }

  override predicate isSource(DataFlow::Node src) {
    src instanceof EnvironmentVariableFlowSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call c | c.getTarget().(Constructor).getDeclaringType().hasQualifiedName(
↪"System.Uri")
    and sink.asExpr()=c.getArgument(0))
  }
}

from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This environment variable constructs a 'System.Uri' $@.", sink, "here"
```

Exercise 5

All properties can flow data:

```
private predicate propertyFlow(Property p) {
  p = getAMember()
}
```

Exercise 6

This can be adapted from the `SystemUriFlow` class:

```
import semmle.code.csharp.dataflow.LibraryTypeDataFlow
import semmle.code.csharp.frameworks.System

class SystemExceptionFlow extends LibraryTypeDataFlow, SystemExceptionClass {
  override predicate callableFlow(CallableFlowSource source, CallableFlowSink sink,␣
↪SourceDeclarationCallable c, boolean preservesValue) {
    (
      constructorFlow(source, c) and
      sink instanceof CallableFlowSinkQualifier
```

(continues on next page)

(continued from previous page)

```
      or
      methodFlow(source, sink, c)
      or
      exists(Property p |
        propertyFlow(p) and
        source instanceof CallableFlowSourceQualifier and
        sink instanceof CallableFlowSinkReturn and
        c = p.getGetter()
      )
    )
    and
    preservesValue = false
  }

  private predicate constructorFlow(CallableFlowSourceArg source, Constructor c) {
    c = getAMember()
    and
    c.getParameter(0).getType() instanceof StringType
    and
    source.getArgumentIndex() = 0
  }

  private predicate methodFlow(CallableFlowSourceQualifier source,␣
↪CallableFlowSinkReturn sink, SourceDeclarationMethod m) {
    m.getDeclaringType() = getABaseType*()
    and
    m = getSystemObjectClass().getToStringMethod().getAnOverrider*()
  }

  private predicate propertyFlow(Property p) {
    p = getAProperty() and p.hasName("Message")
  }
}
```

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for C#
- Example queries for C#
- CodeQL library reference for C#
- "*QL language reference*"
- "*CodeQL tools*"
- *Basic query for C# code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for C#*: When you're analyzing a C# program, you can make use of the large collection of classes in the CodeQL library for C#.
- *Analyzing data flow in C#*: You can use CodeQL to track the flow of data through a C# program to its use.

5.3 CodeQL for Go

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Go codebases.

5.3.1 Basic query for Go code

Learn to write and run a simple CodeQL query using LGTM.

About the query

The query we're going to run searches the code for methods defined on value types that modify their receiver by writing a field:

```
func (s MyStruct) valueMethod() { s.f = 1 } // method on value
```

This is problematic because the receiver argument is passed by value, not by reference. Consequently, valueMethod is called with a copy of the receiver object, so any changes it makes to the receiver will be invisible to the caller. To prevent this, the method should be defined on a pointer instead:

```
func (s *MyStruct) pointerMethod() { s.f = 1 } // method on pointer
```

For further information on using methods on values or pointers in Go, see the Go FAQ.

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Go** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

   ```
   import go

   from Method m, Variable recv, Write w, Field f
   where
     recv = m.getReceiver() and
     w.writesField(recv.getARead(), f, _) and
     not recv.getType() instanceof PointerType
   select w, "This update to " + f + " has no effect, because " + recv + " is not a␣
   ↪pointer."
   ```

   LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   > Note
   >
   > Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to `w`, which is the location in the source code where the receiver `recv` is modified. The second column is the alert message.

   Example query results

   > Note
   >
   > An ellipsis (...) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click a link in the `w` column to view it in the code viewer.

   The matching `w` is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import go`	Imports the standard CodeQL libraries for Go.	Every query begins with one or more `import` statements.
`from Method m, Variable recv, Write w, Field f`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We declare: • `m` as a variable for all methods • a `recv` variable, which is the receiver of `m` • `w` as the location in the code where the receiver is modified • `f` as the field that is written when `m` is called
`where recv = m.getReceiver() and w.writesField(recv.getARead(), f, _) and not recv.getType() instanceof PointerType`	Defines a condition on the variables.	`recv = m.getReceiver()` states that `recv` must be the receiver variable of `m`. `w.writesField(recv.getARead(), f, _)` states that `w` must be a location in the code where field `f` of `recv` is modified. We use a *'don't-care' expression* `_` for the value that is written to `f`—the actual value doesn't matter in this query. `not recv.getType() instanceof PointerType` states that `m` is not a pointer method.
`select w, "This update to " + f + " has no effect, because " + recv + " is not a pointer."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports `w` with a message that explains the potential problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Among the results generated by the first iteration of this query, you can find cases where a value method is called but the receiver variable is returned. In such cases, the change to the receiver is not invisible to the caller, so a pointer method is not required. These are false positive results and you can improve the query by adding an extra condition to remove them.

To exclude these values:

1. Extend the where clause to include the following extra condition:

```
not exists(ReturnStmt ret | ret.getExpr() = recv.getARead().asExpr())
```

The `where` clause is now:

```
where e.isPure() and
  recv = m.getReceiver() and
  w.writesField(recv.getARead(), f, _) and
  not recv.getType() instanceof PointerType and
  not exists(ReturnStmt ret | ret.getExpr() = recv.getARead().asExpr())
```

2. Click **Run**.

   There are now fewer results because value methods that return their receiver variable are no longer reported.

See this in the query console

Further reading

- CodeQL queries for Go
- Example queries for Go
- CodeQL library reference for Go
- "*QL language reference*"
- "*CodeQL tools*"

5.3.2 CodeQL library for Go

When you're analyzing a Go program, you can make use of the large collection of classes in the CodeQL library for Go.

Overview

CodeQL ships with an extensive library for analyzing Go code. The classes in this library present the data from a CodeQL database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `go.qll` imports most other standard library modules, so you can include the complete library by beginning your query with:

```
import go
```

Broadly speaking, the CodeQL library for Go provides two views of a Go code base: at the *syntactic level*, source code is represented as an abstract syntax tree (AST), while at the *data-flow level* it is represented as a data-flow graph (DFG). In between, there is also an intermediate representation of the program as a control-flow graph (CFG), though this representation is rarely useful on its own and mostly used to construct the higher-level DFG representation.

The AST representation captures the syntactic structure of the program. You can use it to reason about syntactic properties such as the nesting of statements within each other, but also about the types of expressions and which variable a name refers to.

The DFG, on the other hand, provides an approximation of how data flows through variables and operations at runtime. It is used, for example, by the security queries to model the way user-controlled input can propagate through the program. Additionally, the DFG contains information about which function may be invoked by a given call (taking

virtual dispatch through interfaces into account), as well as control-flow information about the order in which different operations may be executed at runtime.

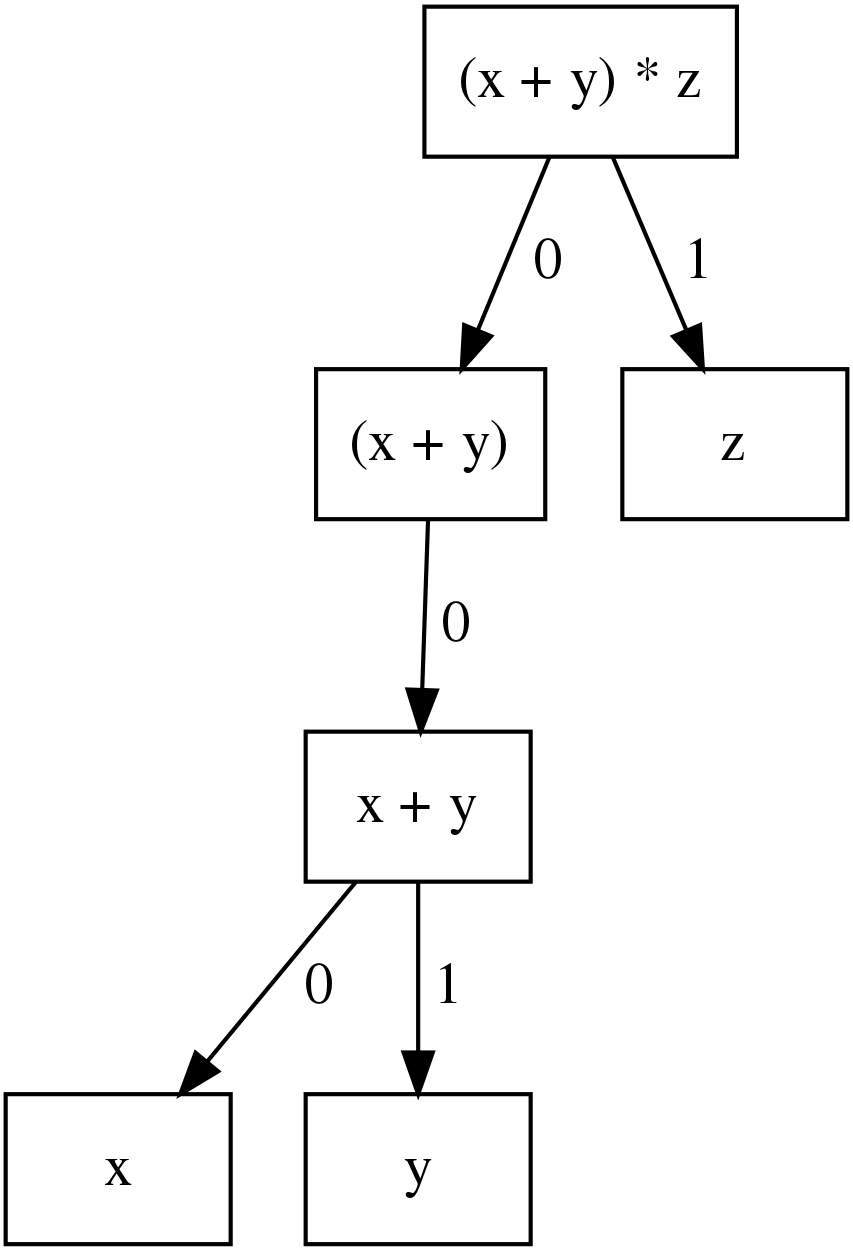
As a rule of thumb, you normally want to use the AST only for superficial syntactic queries. Any analysis involving deeper semantic properties of the program should be done on the DFG.

The rest of this tutorial briefly summarizes the most important classes and predicates provided by this library, including references to the detailed API documentation where applicable. We start by giving an overview of the AST representation, followed by an explanation of names and entities, which are used to represent name-binding information, and of types and type information. Then we move on to control flow and the data-flow graph, and finally the call graph and a few advanced topics.

Abstract syntax

The AST presents the program as a hierarchical structure of nodes, each of which corresponds to a syntactic element of the program source text. For example, there is an AST node for each expression and each statement in the program. These AST nodes are arranged into a parent-child relationship reflecting the nesting of syntactic elements and the order in which inner elements appear in enclosing ones.

For example, this is the AST for the expression `(x + y) * z`:



It is composed of six AST nodes, representing `x`, `y`, `x + y`, `(x + y)`, `z` and the entire expression `(x + y) * z`, respectively. The AST nodes representing `x` and `y` are children of the AST node representing `x + y`, `x` being the zeroth child and `y` being the first child, reflecting their order in the program text. Similarly, `x + y` is the only child of `(x + y)`, which is the zeroth child of `(x + y) * z`, whose first child is `z`.

All AST nodes belong to class AstNode, which defines generic tree traversal predicates:

- `getChild(i)`: returns the `i`th child of this AST node.
- `getAChild()`: returns any child of this AST node.
- `getParent()`: returns the parent node of this AST node, if any.

These predicates should only be used to perform generic AST traversal. To access children of specific AST node types, the specialized predicates introduced below should be used instead. In particular, queries should not rely on the numeric indices of child nodes relative to their parent nodes: these are considered an implementation detail that may change

between versions of the library.

The predicate `toString()` in class `AstNode` nodes gives a short description of the AST node, usually just indicating what kind of node it is. The `toString()` predicate does *not* provide access to the source text corresponding to an AST node. The source text is not stored in the dataset, and hence is not directly accessible to CodeQL queries.

The predicate `getLocation()` in class `AstNode` returns a Location entity describing the source location of the program element represented by the AST node. You can use its member predicates `getFile()`, `getStartLine()`, `getStartColumn`, `getEndLine()`, and `getEndColumn()` to obtain information about its file, start line and column, and end line and column.

The most important subclasses of AstNode are Stmt and Expr, which represent statements and expressions, respectively. This section briefly discusses some of their more important subclasses and predicates. For a full reference of all the subclasses of Stmt and Expr, see *Abstract syntax tree classes for Go*.

Statements

- `ExprStmt`: an expression statement; use `getExpr()` to access the expression itself
- `Assignment`: an assignment statement; use `getLhs(i)` to access the `i`th left-hand side and `getRhs(i)` to access the `i`th right-hand side; if there is only a single left-hand side you can use `getLhs()` instead, and similar for the right-hand side
  - `SimpleAssignStmt`: an assignment statement that does not involve a compound operator
    - `AssignStmt`: a plain assignment statement of the form `lhs = rhs`
    - `DefineStmt`: a short-hand variable declaration of the form `lhs := rhs`
  - `CompoundAssignStmt`: an assignment statement with a compound operator, such as `lhs += rhs`
- `IncStmt`, `DecStmt`: an increment statement or a decrement statement, respectively; use `getOperand()` to access the expression being incremented or decremented
- `BlockStmt`: a block of statements between curly braces; use `getStmt(i)` to access the `i`th statement in a block
- `IfStmt`: an `if` statement; use `getInit()`, `getCond()`, `getThen()`, and `getElse()` to access the (optional) init statement, the condition being checked, the "then" branch to evaluate if the condition is true, and the (optional) "else" branch to evaluate otherwise, respectively
- `LoopStmt`: a loop; use `getBody()` to access its body
  - `ForStmt`: a `for` statement; use `getInit()`, `getCond()`, and `getPost()` to access the init statement, loop condition, and post statement, respectively, all of which are optional
  - `RangeStmt`: a `range` statement; use `getDomain()` to access the iteration domain, and `getKey()` and `getValue()` to access the expressions to which successive keys and values are assigned, if any
- `GoStmt`: a `go` statement; use `getCall()` to access the call expression that is evaluated in the new goroutine
- `DeferStmt`: a `defer` statement; use `getCall()` to access the call expression being deferred
- `SendStmt`: a send statement; use `getChannel()` and `getValue()` to access the channel and the value being sent over the channel, respectively
- `ReturnStmt`: a `return` statement; use `getExpr(i)` to access the `i`th returned expression; if there is only a single returned expression you can use `getExpr()` instead
- `BranchStmt`: a statement that interrupts structured control flow; use `getLabel()` to get the optional target label
  - `BreakStmt`: a `break` statement
  - `ContinueStmt`: a `continue` statement

  - `FallthroughStmt`: a `fallthrough` statement at the end of a switch case
  - `GotoStmt`: a `goto` statement
- `DeclStmt`: a declaration statement, use `getDecl()` to access the declaration in this statement; note that one rarely needs to deal with declaration statements directly, since reasoning about the entities they declare is usually easier
- `SwitchStmt`: a `switch` statement; use `getInit()` to access the (optional) init statement, and `getCase(i)` to access the `i`th `case` or `default` clause
  - `ExpressionSwitchStmt`: a `switch` statement examining the value of an expression
  - `TypeSwitchStmt`: a `switch` statement examining the type of an expression
- `CaseClause`: a `case` or `default` clause in a `switch` statement; use `getExpr(i)` to access the `i`th expression, and `getStmt(i)` to access the `i`th statement in the body of this clause
- `SelectStmt`: a `select` statement; use `getCommClause(i)` to access the `i`th `case` or `default` clause
- `CommClause`: a `case` or `default` clause in a `select` statement; use `getComm()` to access the send/receive statement of this clause (not defined for `default` clauses), and `getStmt(i)` to access the `i`th statement in the body of this clause
- `RecvStmt`: a receive statement in a `case` clause of a `select` statement; use `getLhs(i)` to access the `i`th left-hand side of this statement, and `getExpr()` to access the underlying receive expression

Expressions

Class `Expression` has a predicate `isConst()` that holds if the expression is a compile-time constant. For such constant expressions, `getNumericValue()` and `getStringValue()` can be used to determine their numeric value and string value, respectively. Note that these predicates are not defined for expressions whose value cannot be determined at compile time. Also note that the result type of `getNumericValue()` is the QL type `float`. If an expression has a numeric value that cannot be represented as a QL `float`, this predicate is also not defined. In such cases, you can use `getExactValue()` to obtain a string representation of the value of the constant.

- `Ident`: an identifier; use `getName()` to access its name
- `SelectorExpr`: a selector of the form `base.sel`; use `getBase()` to access the part before the dot, and `getSelector()` for the identifier after the dot
- `BasicLit`: a literal of a basic type; subclasses `IntLit`, `FloatLit`, `ImagLit`, `RuneLit`, and `StringLit` represent various specific kinds of literals
- `FuncLit`: a function literal; use `getBody()` to access the body of the function
- `CompositeLit`: a composite literal; use `getKey(i)` and `getValue(i)` to access the `i`th key and the `i`th value, respectively
- `ParenExpr`: a parenthesized expression; use `getExpr()` to access the expression between the parentheses
- `IndexExpr`: an index expression `base[idx]`; use `getBase()` and `getIndex()` to access `base` and `idx`, respectively
- `SliceExpr`: a slice expression `base[lo:hi:max]`; use `getBase()`, `getLow()`, `getHigh()`, and `getMax()` to access `base`, `lo`, `hi`, and `max`, respectively; note that `lo`, `hi`, and `max` can be omitted, in which case the corresponding predicates are not defined
- `ConversionExpr`: a conversion expression `T(e)`; use `getTypeExpr()` and `getOperand()` to access `T` and `e`, respectively
- `TypeAssertExpr`: a type assertion `e.(T)`; use `getExpr()` and `getTypeExpr()` to access `e` and `T`, respectively

- `CallExpr`: a call expression `callee(arg0, ..., argn)`; use `getCalleeExpr()` to access `callee`, and `getArg(i)` to access the `i`th argument
- `StarExpr`: a star expression, which may be either a pointer-type expression or a pointer-dereference expression, depending on context; use `getBase()` to access the operand of the star
- `TypeExpr`: an expression that denotes a type
- `OperatorExpr`: an expression with a unary or binary operator; use `getOperator()` to access the operator
  - `UnaryExpr`: an expression with a unary operator; use `getAnOperand()` to access the operand of the operator
  - `BinaryExpr`: an expression with a binary operator; use `getLeftOperand()` and `getRightOperand()` to access the left and the right operand, respectively
    - `ComparisonExpr`: a binary expression that performs a comparison, including both equality tests and relational comparisons
      - `EqualityTestExpr`: an equality test, that is, either == or `!=`; the predicate `getPolarity()` has result `true` for the former and `false` for the latter
      - `RelationalComparisonExpr`: a relational comparison; use `getLesserOperand()` and `getGreaterOperand()` to access the lesser and greater operand of the comparison, respectively; `isStrict()` holds if this is a strict comparison using < or >, as opposed to <= or >=

Names

While `Ident` and `SelectorExpr` are very useful classes, they are often too general: `Ident` covers all identifiers in a program, including both identifiers appearing in a declaration as well as references, and does not distinguish between names referring to packages, types, variables, constants, functions, or statement labels. Similarly, a `SelectorExpr` might refer to a package, a type, a function, or a method.

Class `Name` and its subclasses provide a more fine-grained mapping of this space, organized along the two axes of structure and namespace. In terms of structure, a name can be a `SimpleName`, meaning that it is a simple identifier (and hence an `Ident`), or it can be a `QualifiedName`, meaning that it is a qualified identifier (and hence a `SelectorExpr`). In terms of namespacing, a `Name` can be a `PackageName`, `TypeName`, `ValueName`, or `LabelName`. A `ValueName`, in turn, can be either a `ConstantName`, a `VariableName`, or a `FunctionName`, depending on what sort of entity the name refers to.

A related abstraction is provided by class `ReferenceExpr`: a reference expression is an expression that refers to a variable, a constant, a function, a field, or an element of an array or a slice. Use predicates `isLvalue()` and `isRvalue()` to determine whether a reference expression appears in a syntactic context where it is assigned to or read from, respectively.

Finally, `ValueExpr` generalizes `ReferenceExpr` to include all other kinds of expressions that can be evaluated to a value (as opposed to expressions that refer to a package, a type, or a statement label).

Functions

At the syntactic level, functions appear in two forms: in function declarations (represented by class `FuncDecl`) and as function literals (represented by class `FuncLit`). Since it is often convenient to reason about functions of either kind, these two classes share a common superclass `FuncDef`, which defines a few useful member predicates:

- `getBody()` provides access to the function body
- `getName()` gets the function name; it is undefined for function literals, which do not have a name
- `getParameter(i)` gets the `i`th parameter of the function
- `getResultVar(i)` gets the `i`th result variable of the function; if there is only one result, `getResultVar()` can be used to access it
- `getACall()` gets a data-flow node (see below) representing a call to this function

Entities and name binding

Not all elements of a code base can be represented as AST nodes. For example, functions defined in the standard library or in a dependency do not have a source-level definition within the source code of the program itself, and built-in functions like `len` do not have a definition at all. Hence functions cannot simplify be identified with their definition, and similarly for variables, types, and so on.

To smooth over this difference and provide a unified view of functions no matter where they are defined, the Go library introduces the concept of an *entity*. An entity is a named program element, that is, a package, a type, a constant, a variable, a field, a function, or a label. All entities belong to class `Entity`, which defines a few useful predicates:

- `getName()` gets the name of the entity
- `hasQualifiedName(pkg, n)` holds if this entity is declared in package `pkg` and has name `n`; this predicate is only defined for types, functions, and package-level variables and constants (but not for methods or local variables)
- `getDeclaration()` connects an entity to its declaring identifier, if any
- `getAReference()` gets a `Name` that refers to this entity

Conversely, class `Name` defines a predicate `getTarget()` that gets the entity to which the name refers.

Class `Entity` has several subclasses representing specific kinds of entities: `PackageEntity` for packages; `TypeEntity` for types; `ValueEntity` for constants (`Constant`), variables (`Variable`), and functions (`Function`); and `Label` for statement labels.

Class `Variable`, in turn, has a few subclasses representing specific kinds of variables: a `LocalVariable` is a variable declared in a local scope, that is, not at package level; `ReceiverVariable`, `Parameter` and `ResultVariable` describe receivers, parameters and results, respectively, and define a predicate `getFunction()` to access the corresponding function. Finally, class `Field` represents struct fields, and provides a member predicate `hasQualifiedName(pkg, tp, f)` that holds if this field has name `f` and belongs to type `tp` in package `pkg`. (Note that due to embedding the same field can belong to multiple types.)

Class `Function` has a subclass `Method` representing methods (including both interface methods and methods defined on a named type). Similar to `Field`, `Method` provides a member predicate `hasQualifiedName(pkg, tp, m)` that holds if this method has name `m` and belongs to type `tp` in package `pkg`. Predicate `implements(m2)` holds if this method implements method `m2`, that is, it has the same name and signature as `m2` and it belongs to a type that implements the interface to which `m2` belongs. For any function, `getACall()` provides access to call sites that may call this function, possibly through virtual dispatch.

Finally, module `Builtin` provides a convenient way of looking up the entities corresponding to built-in functions and types. For example, `Builtin::len()` is the entity representing the built-in function `len`, `Builtin::bool()` is the `bool` type, and `Builtin::nil()` is the value `nil`.

Type information

Types are represented by class `Type` and its subclasses, such as `BoolType` for the built-in type `bool`; `NumericType` for the various numeric types including `IntType`, `Uint8Type`, `Float64Type` and others; `StringType` for the type `string`; `NamedType`, `ArrayType`, `SliceType`, `StructType`, `InterfaceType`, `PointerType`, `MapType`, `ChanType` for named types, arrays, slices, structs, interfaces, pointers, maps, and channels, respectively. Finally, `SignatureType` represents function types.

Note that the type `BoolType` is distinct from the entity `Builtin::bool()`: the latter views `bool` as a declared entity, the former as a type. You can, however, map from types to their corresponding entity (if any) using the predicate `getEntity()`.

Class `Expr` and class `Entity` both define a predicate `getType()` to determine the type of an expression or entity. If the type of an expression or entity cannot be determined (for example because some dependency could not be found during extraction), it will be associated with an invalid type of class `InvalidType`.

Control flow

Most CodeQL query writers will rarely use the control-flow representation of a program directly, but it is nevertheless useful to understand how it works.

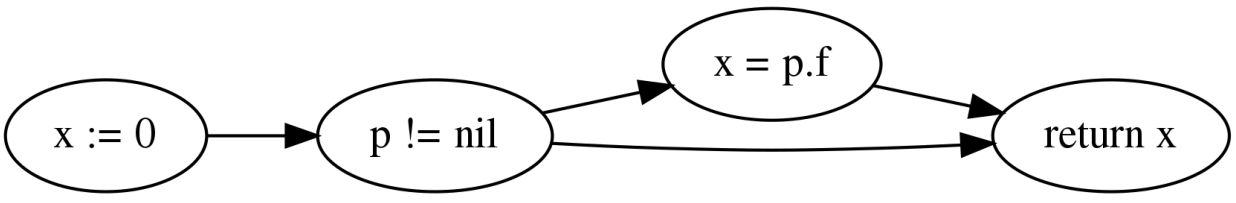
Unlike the abstract syntax tree, which views the program as a hierarchy of AST nodes, the control-flow graph views it as a collection of *control-flow nodes*, each representing a single operation performed at runtime. These nodes are connected to each other by (directed) edges representing the order in which operations are performed.

For example, consider the following code snippet:

```
x := 0
if p != nil {
  x = p.f
}
return x
```

In the AST, this is represented as an `IfStmt` and a `ReturnStmt`, with the former having an `NeqExpr` and a `BlockStmt` as its children, and so on. This provides a very detailed picture of the syntactic structure of the code, but it does not immediately help us reason about the order in which the various operations such as the comparison and the assignment are performed.

In the CFG, there are nodes corresponding to `x := 0`, `p != nil`, `x = p.f`, and `return x`, as well as a few others. The edges between these nodes model the possible execution orders of these statements and expressions, and look as follows (simplified somewhat for presentational purposes):



For example, the edge from `p != nil` to `x = p.f` models the case where the comparison evaluates to `true` and the "then" branch is evaluated, while the edge from `p != nil` to `return x` models the case where the comparison evaluates to `false` and the "then" branch is skipped.

Note, in particular, that a CFG node can have multiple outgoing edges (like from `p != nil`) as well as multiple incoming edges (like into `return x`) to represent control-flow branching at runtime.

Also note that only AST nodes that perform some kind of operation on values have a corresponding CFG node. This includes expressions (such as the comparison `p != nil`), assignment statements (such as `x = p.f`) and return statements (such as `return x`), but not statements that serve a purely syntactic purpose (such as block statements) and statements whose semantics is already reflected by the CFG edges (such as `if` statements).

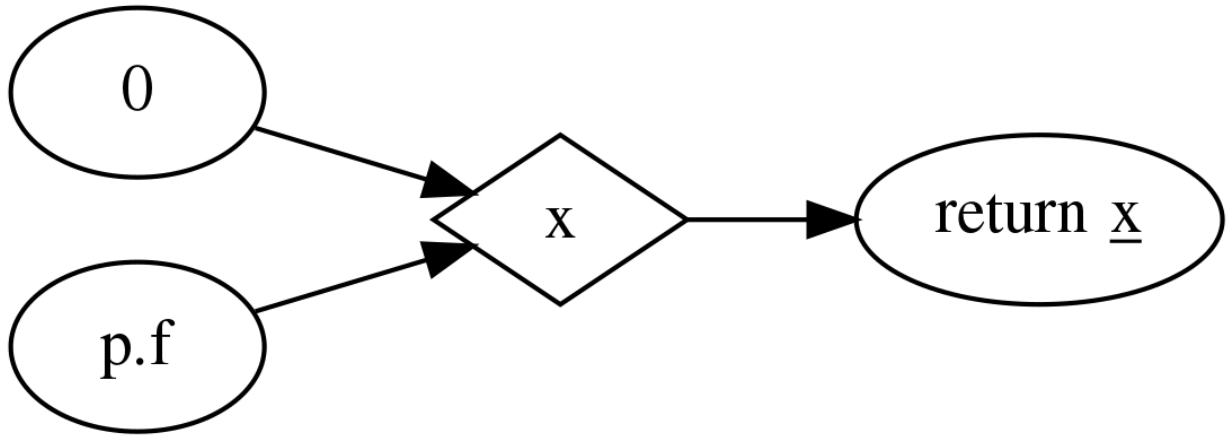
It is important to point out that the control-flow graph provided by the CodeQL libraries for Go only models *local* control flow, that is, flow within a single function. Flow from function calls to the function they invoke, for example, is not represented by control-flow edges.

In CodeQL, control-flow nodes are represented by class `ControlFlow::Node`, and the edges between nodes are captured by the member predicates `getASuccessor()` and `getAPredecessor()` of `ControlFlow::Node`. In addition to control-flow nodes representing runtime operations, each function also has a synthetic entry node and an exit node, representing the start and end of an execution of the function, respectively. These exist to ensure that the control-flow graph corresponding to a function has a unique entry node and a unique exit node, which is required for many standard control-flow analysis algorithms.

Data flow

At the data-flow level, the program is thought of as a collection of *data-flow nodes*. These nodes are connected to each other by (directed) edges representing the way data flows through the program at runtime.

For example, there are data-flow nodes corresponding to expressions and other data-flow nodes corresponding to variables (SSA variables, to be precise). Here is the data-flow graph corresponding to the code snippet shown above, ignoring SSA conversion for simplicity:



Note that unlike in the control-flow graph, the assignments `x := 0` and `x = p.f` are not represented as nodes. Instead, they are expressed as edges between the node representing the right-hand side of the assignment and the node representing the variable on the left-hand side. For any subsequent uses of that variable, there is a data-flow edge from the variable to that use, so by following the edges in the data-flow graph we can trace the flow of values through variables at runtime.

It is important to point out that the data-flow graph provided by the CodeQL libraries for Go only models *local* flow, that is, flow within a single function. Flow from arguments in a function call to the corresponding function parameters, for example, is not represented by data-flow edges.

In CodeQL, data-flow nodes are represented by class `DataFlow::Node`, and the edges between nodes are captured by the predicate `DataFlow::localFlowStep`. The predicate `DataFlow::localFlow` generalizes this from a single flow step to zero or more flow steps.

Most expressions have a corresponding data-flow node; exceptions include type expressions, statement labels and other expressions that do not have a value, as well as short-circuiting operators. To map from the AST node of an expression to the corresponding DFG node, use `DataFlow::exprNode`. Note that the AST node and the DFG node are different entities and cannot be used interchangeably.

There is also a predicate `asExpr()` on `DataFlow::Node` that allows you to recover the expression underlying a DFG node. However, this predicate should be used with caution, since many data-flow nodes do not correspond to an expression, and so this predicate will not be defined for them.

Similar to `Expr`, `DataFlow::Node` has a member predicate `getType()` to determine the type of a node, as well as predicates `getNumericValue()`, `getStringValue()`, and `getExactValue()` to retrieve the value of a node if it is constant.

Important subclasses of `DataFlow::Node` include:

- `DataFlow::CallNode`: a function call or method call; use `getArgument(i)` and `getResult(i)` to obtain the data-flow nodes corresponding to the `i`th argument and the `i`th result of this call, respectively; if there is only a single result, `getResult()` will return it
- `DataFlow::ParameterNode`: a parameter of a function; use `asParameter()` to access the corresponding AST node
- `DataFlow::BinaryOperationNode`: an operation involving a binary operator; each `BinaryExpr` has a corresponding `BinaryOperationNode`, but there are also binary operations that are not explicit at the AST level, such as those arising from compound assignments and increment/decrement statements; at the AST level, `x + 1`, `x += 1`, and `x++` are represented by different kinds of AST nodes, while at the DFG level they are all modeled as a binary operation node with operands `x` and `1`
- `DataFlow::UnaryOperationNode`: analogous, but for unary operators
  - `DataFlow::PointerDereferenceNode`: a pointer dereference, either explicit in an expression of the form `*p`, or implicit in a field or method reference through a pointer
  - `DataFlow::AddressOperationNode`: analogous, but for taking the address of an entity
  - `DataFlow::RelationalComparisonNode`, `DataFlow::EqualityTestNode`: data-flow nodes corresponding to `RelationalComparisonExpr` and `EqualityTestExpr` AST nodes

Finally, classes `Read` and `Write` represent, respectively, a read or a write of a variable, a field, or an element of an array, a slice or a map. Use their member predicates `readsVariable`, `writesVariable`, `readsField`, `writesField`, `readsElement`, and `writesElement` to determine what the read/write refers to.

Call graph

The call graph connects function (and method) calls to the functions they invoke. Call graph information is made available by two member predicates on `DataFlow::CallNode`: `getTarget()` returns the declared target of a call, while `getACallee()` returns all possible actual functions a call may invoke at runtime.

These two predicates differ in how they handle calls to interface methods: while `getTarget()` will return the interface method itself, `getACallee()` will return all concrete methods that implement the interface method.

Global data flow and taint tracking

The predicates `DataFlow::localFlowStep` and `DataFlow::localFlow` are useful for reasoning about the flow of values in a single function. However, more advanced use cases, particularly in security analysis, will invariably require reasoning about global data flow, including flow into, out of, and across function calls, and through fields.

In CodeQL, such reasoning is expressed in terms of *data-flow configurations*. A data-flow configuration has three ingredients: sources, sinks, and barriers (also called sanitizers), all of which are sets of data-flow nodes. Given these three sets, CodeQL provides a general mechanism for finding paths from a source to a sink, possibly going into and out of functions and fields, but never flowing through a barrier.

To define a data-flow configuration, you can define a subclass of `DataFlow::Configuration`, overriding the member predicates `isSource`, `isSink`, and `isBarrier` to define the sets of sources, sinks, and barriers.

Going beyond pure data flow, many security analyses need to perform more general *taint tracking*, which also considers flow through value-transforming operations such as string operations. To track taint, you can define a subclass of `TaintTracking::Configuration`, which works similar to data-flow configurations.

A detailed exposition of global data flow and taint tracking is out of scope for this brief introduction. For a general overview of data flow and taint tracking, see "*About data flow analysis*."

Advanced libraries

Finally, we briefly describe a few concepts and libraries that are useful for advanced query writers.

Basic blocks and dominance

Many important control-flow analyses organize control-flow nodes into basic blocks, which are maximal straight-line sequences of control-flow nodes without any branching. In the CodeQL libraries, basic blocks are represented by class `BasicBlock`. Each control-flow node belongs to a basic block. You can use the predicate `getBasicBlock()` in class `ControlFlow::Node` and the predicate `getNode(i)` in `BasicBlock` to move from one to the other.

Dominance is a standard concept in control-flow analysis: a basic block `dom` is said to *dominate* a basic block `bb` if any path through the control-flow graph from the entry node to the first node of `bb` must pass through `dom`. In other words, whenever program execution reaches the beginning of `bb`, it must have come through `dom`. Each basic block is moreover considered to dominate itself.

Dually, a basic block `postdom` is said to *post-dominate* a basic block `bb` if any path through the control-flow graph from the last node of `bb` to the exit node must pass through `postdom`. In other words, after program execution leaves `bb`, it must eventually reach `postdom`.

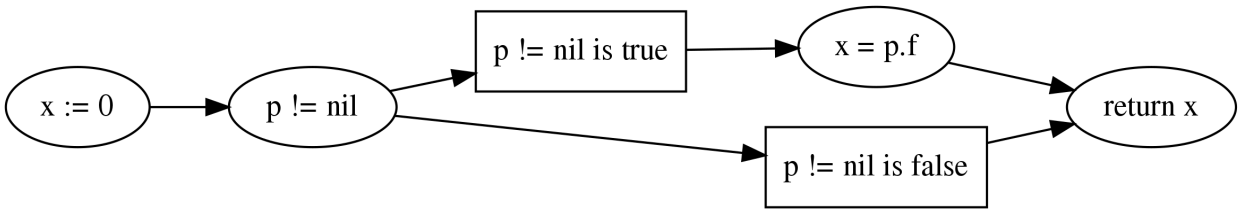
These two concepts are captured by two member predicates `dominates` and `postDominates` of class `BasicBlock`.

Condition guard nodes

A condition guard node is a synthetic control-flow node that records the fact that at some point in the control-flow graph the truth value of a condition is known. For example, consider again the code snippet we saw above:

```
x := 0
if p != nil {
  x = p.f
}
return x
```

At the beginning of the "then" branch `p` is known not be `nil`. This knowledge is encoded in the control-flow graph by a condition guard node preceding the assignment to `x`, recording the fact that `p != nil` is `true` at this point:



A typical use of this information would be in an analysis that looks for `nil` dereferences: such an analysis would be able to conclude that the field read `p.f` is safe because it is immediately preceded by a condition guard node guaranteeing that `p` is not `nil`.

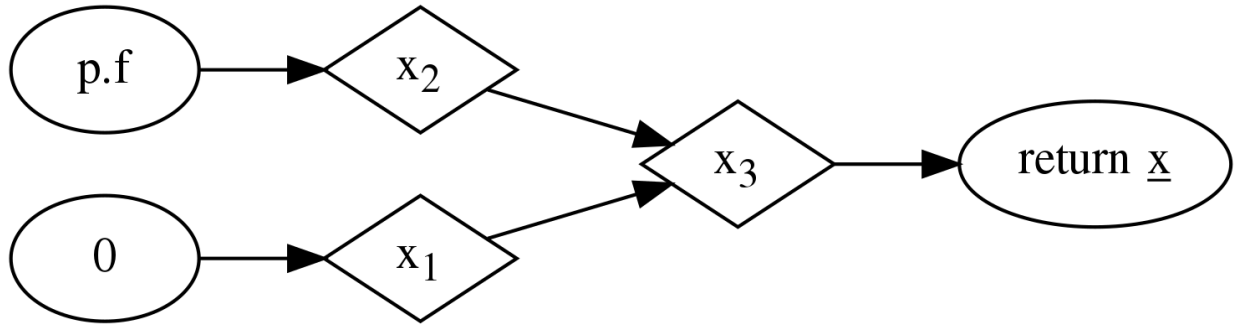
In CodeQL, condition guard nodes are represented by class `ControlFlow::ConditionGuardNode` which offers a variety of member predicates to reason about which conditions a guard node guarantees.

Static single-assignment form

Static single-assignment form (SSA form for short) is a program representation in which the original program variables are mapped onto more fine-grained *SSA variables*. Each SSA variable has exactly one definition, so program variables with multiple assignments correspond to multiple SSA variables.

Most of the time query authors do not have to deal with SSA form directly. The data-flow graph uses it under the hood, and so most of the benefits derived from SSA can be gained by simply using the data-flow graph.

For example, the data-flow graph for our running example actually looks more like this:



Note that the program variable `x` has been mapped onto three distinct SSA variables `x1`, `x2`, and `x3`. In this case there is not much benefit to such a representation, but in general SSA form has well-known advantages for data-flow analysis for which we refer to the literature.

If you do need to work with raw SSA variables, they are represented by the class `SsaVariable`. Class `SsaDefinition` represents definitions of SSA variables, which have a one-to-one correspondence with `SsaVariable`s. Member predicates `getDefinition()` and `getVariable()` exist to map from one to the other. You can use member predicate `getAUse()` of `SsaVariable` to look for uses of an SSA variable. To access the program variable underlying an SSA variable, use member predicate `getSourceVariable()`.

Global value numbering

Global value numbering is a technique for determining when two computations in a program are guaranteed to yield the same result. This is done by associating with each data-flow node an abstract representation of its value (conventionally called a *value number*, even though in practice it is not usually a number) such that identical computations are represented by identical value numbers.

Since this is an undecidable problem, global value numbering is *conservative* in the sense that if two data-flow nodes have the same value number they are guaranteed to have the same value at runtime, but not conversely. (That is, there may be data-flow nodes that do, in fact, always evaluate to the same value, but their value numbers are different.)

In the CodeQL libraries for Go, you can use the `globalValueNumber(nd)` predicate to compute the global value number for a data-flow node `nd`. Value numbers are represented as an opaque QL type `GVN` that provides very little information. Usually, all you need to do with global value numbers is to compare them to each other to determine whether two data-flow nodes have the same value.

Further reading

- CodeQL queries for Go
- Example queries for Go
- CodeQL library reference for Go
- "*QL language reference*"
- "*CodeQL tools*"

5.3.3 Abstract syntax tree classes for working with Go programs

CodeQL has a large selection of classes for representing the abstract syntax tree of Go programs.

The abstract syntax tree (AST) represents the syntactic structure of a program. Nodes on the AST represent elements such as statements and expressions.

Statement classes

This table lists all subclasses of Stmt.

Statement syntax	CodeQL class	Superclasses	Remarks
`;`	EmptyStmt		
Expr	ExprStmt		
`{` Stmt `...` `}`	BlockStmt		
`if` Expr BlockStmt	IfStmt		
`if` Expr BlockStmt `else` Stmt			
`if` Stmt`;` Expr BlockStmt			
`for` Expr BlockStmt	ForStmt	LoopStmt	
`for` Stmt`;` Expr`;` Stmt BlockStmt			
`for` Expr `...` `=` `range` Expr BlockStmt	RangeStmt	LoopStmt	
`switch` Expr `{` CaseClause `...` `}`	ExpressionSwitchStmt	SwitchStmt	
`switch` Stmt`;` Expr `{` CaseClause `...` `}`			
`switch` Expr`.(type)` `{` CaseClause `...` `}`	TypeSwitchStmt	SwitchStmt	
`switch` SimpleAssignStmt`.(type)` `{` CaseClause `...` `}`			

Table 3 – continued from previous page

Statement syntax	CodeQL class	Superclasses	Remarks
`switch` Stmt; Expr.`(type)` { CaseClause ... }			
`select` { CommClause ... }	SelectStmt		
`return`	ReturnStmt		
`break`	BreakStmt	BranchStmt	
`continue`	ContinueStmt	BranchStmt	
`goto` LabelName	GotoStmt	BranchStmt	
`fallthrough`	FallthroughStmt	BranchStmt	can only
LabelName: Stmt	LabeledStmt		
`var` VariableName TypeName	DeclStmt		
`const` VariableName = Expr			
`type` TypeName TypeExpr			
`type` TypeName = TypeExpr			
Expr ... = Expr ...	AssignStmt	SimpleAssignStmt, Assignment	
VariableName ... := Expr ...	DefineStmt	SimpleAssignStmt, Assignment	
Expr += Expr	AddAssignStmt	CompoundAssignStmt, Assignment	
Expr -= Expr	SubAssignStmt	CompoundAssignStmt, Assignment	
Expr *= Expr	MulAssignStmt	CompoundAssignStmt, Assignment	
Expr /= Expr	QuoAssignStmt	CompoundAssignStmt, Assignment	
Expr %= Expr	RemAssignStmt	CompoundAssignStmt, Assignment	
Expr *= Expr	MulAssignStmt	CompoundAssignStmt, Assignment	
Expr &= Expr	AndAssignStmt	CompoundAssignStmt, Assignment	
Expr |= Expr	OrAssignStmt	CompoundAssignStmt, Assignment	
Expr ^= Expr	XorAssignStmt	CompoundAssignStmt, Assignment	
Expr <<= Expr	ShlAssignStmt	CompoundAssignStmt, Assignment	
Expr >>= Expr	ShrAssignStmt	CompoundAssignStmt, Assignment	
Expr &^= Expr	AndNotAssignStmt	CompoundAssignStmt, Assignment	
Expr ++	IncStmt	IncDecStmt	
Expr --	DecStmt	IncDecStmt	
`go` CallExpr	GoStmt		
`defer` CallExpr	DeferStmt		
Expr <- Expr	SendStmt		
`case` Expr ...: Stmt ...	CaseClause		can only
`case` TypeExpr ...: Stmt ...			a SwitchS
`default`: Stmt ...			
`case` SendStmt: Stmt ...	CommClause		can only
`case` RecvStmt: Stmt ...			a SelectS
`default`: Stmt ...			
Expr ... = RecvExpr	RecvStmt		can only
VariableName ... := RecvExpr			a CommC
(anything unparseable)	BadStmt		

Expression classes

There are many expression classes, so we present them by category. All classes in this section are subclasses of Expr.

Literals

Expression syntax example	CodeQL class	Superclass
`23`	IntLit	BasicLit
`4.2`	FloatLit	BasicLit
`4.2 + 2.7i`	ImagLit	BasicLit
`'a'`	CharLit	BasicLit
`"Hello"`	StringLit	BasicLit
`func(x, y int) int { return x + y }`	FuncLit	FuncDef
`map[string]int{"A": 1, "B": 2}`	MapLit	CompositeLit
`Point3D{0.5, -0.5, 0.5}`	StructLit	CompositeLit

Unary expressions

All classes in this subsection are subclasses of UnaryExpr.

Expression syntax	CodeQL class	Superclasses
+Expr	PlusExpr	ArithmeticUnaryExpr
-Expr	MinusExpr	ArithmeticUnaryExpr
!Expr	NotExpr	LogicalUnaryExpr
^Expr	ComplementExpr	BitwiseUnaryExpr
&Expr	AddressExpr	
<-Expr	RecvExpr	

Binary expressions

All classes in this subsection are subclasses of BinaryExpr.

Expression syntax	CodeQL class	Superclasses
Expr * Expr	MulExpr	ArithmeticBinaryExpr
Expr / Expr	QuoExpr	ArithmeticBinaryExpr
Expr % Expr	RemExpr	ArithmeticBinaryExpr
Expr + Expr	AddExpr	ArithmeticBinaryExpr
Expr - Expr	SubExpr	ArithmeticBinaryExpr
Expr << Expr	ShlExpr	ShiftExpr
Expr >> Expr	ShrExpr	ShiftExpr
Expr && Expr	LandExpr	LogicalBinaryExpr
Expr || Expr	LorExpr	LogicalBinaryExpr
Expr < Expr	LssExpr	RelationalComparisonExpr
Expr > Expr	GtrExpr	RelationalComparisonExpr
Expr <= Expr	LeqExpr	RelationalComparisonExpr
Expr >= Expr	GeqExpr	RelationalComparisonExpr
Expr == Expr	EqlExpr	EqualityTestExpr
Expr != Expr	NeqExpr	EqualityTestExpr
Expr & Expr	AndExpr	BitwiseBinaryExpr
Expr | Expr	OrExpr	BitwiseBinaryExpr
Expr ^ Expr	XorExpr	BitwiseBinaryExpr
Expr &^ Expr	AndNotExpr	BitwiseBinaryExpr

Type expressions

These classes represent different expressions for types. They do not have a common superclass.

Expression syntax	CodeQL class	Superclasses
`[`Expr`]` TypeExpr	ArrayTypeExpr	
`struct { ... }`	StructTypeExpr	
`func` FunctionName`(...) (...)`	FuncTypeExpr	
`interface { ... }`	InterfaceTypeExpr	
`map[`TypeExpr`]`TypeExpr	MapTypeExpr	
`chan<-` TypeExpr	SendChanTypeExpr	ChanTypeExpr
`<-chan` TypeExpr	RecvChanTypeExpr	ChanTypeExpr
`chan` TypeExpr	SendRecvChanTypeExpr	ChanTypeExpr

Name expressions

All classes in this subsection are subclasses of Name.

The following classes relate to the structure of the name.

Expression syntax	CodeQL class	Superclasses
Ident	SimpleName	Ident
Ident`.`Ident	QualifiedName	SelectorExpr

The following classes relate to what sort of entity the name refers to.

- PackageName

- TypeName
- LabelName
- ValueName
  - ConstantName
  - VariableName
  - FunctionName

Miscellaneous

Expression syntax	CodeQL class	Superclasses	Remarks
`foo`	Ident		
_	BlankIdent		
`...`	Ellipsis		
`(`Expr`)`	ParenExpr		
Ident`.`Ident	SelectorExpr		
Expr`[`Expr`]`	IndexExpr		
Expr`[`Expr`:`Expr`:`Expr	SliceExpr		
Expr`.(`TypeExpr`)`	TypeAssert-Expr		
*Expr	StarExpr		can be a ValueExpr or TypeExpr depending on context
Expr`:` Expr	KeyValue-Expr		
TypeExpr`(`Expr`)`	Conversion-Expr	CallOrConversion-Expr	
Expr`(...)`	CallExpr	CallOrConversion-Expr	
(anything un-parseable)	BadExpr		

The following classes organize expressions by the kind of entity they refer to.

CodeQL class	Explanation
TypeExpr	an expression that denotes a type
Refer-enceExpr	an expression that refers to a variable, a constant, a function, a field, or an element of an array or a slice
Value-Expr	an expression that can be evaluated to a value (as opposed to expressions that refer to a package, a type, or a statement label). This generalizes ReferenceExpr

Further reading

- CodeQL queries for Go
- Example queries for Go
- CodeQL library reference for Go
- "*QL language reference*"
- "*CodeQL tools*"

5.3.4 Modeling data flow in Go libraries

When analyzing a Go program, CodeQL does not examine the source code for external packages. To track the flow of untrusted data through a library, you can create a model of the library.

You can find existing models in the `ql/src/semmle/go/frameworks/` folder of the CodeQL for Go repository. To add a new model, you should make a new file in that folder, named after the library.

Sources

To mark a source of data that is controlled by an untrusted user, we create a class extending `UntrustedFlowSource::Range`. Inheritance and the characteristic predicate of the class should be used to specify exactly the dataflow node that introduces the data. Here is a short example from `Mux.qll`.

```
class RequestVars extends DataFlow::UntrustedFlowSource::Range, DataFlow::CallNode {
  RequestVars() { this.getTarget().hasQualifiedName("github.com/gorilla/mux", "Vars") }
}
```

This has the effect that all calls to the function Vars from the package mux are treated as sources of untrusted data.

Flow propagation

By default, we assume that all functions in libraries do not have any data flow. To indicate that a particular function does have data flow, create a class extending `TaintTracking::FunctionModel` (or `DataFlow::FunctionModel` if the untrusted user data is passed on without being modified).

Inheritance and the characteristic predicate of the class should specify the function. The class should also have a member predicate with the signature `override predicate hasTaintFlow(FunctionInput inp, FunctionOutput outp)` (or `override predicate hasDataFlow(FunctionInput inp, FunctionOutput outp)` if extending `DataFlow::FunctionModel`). The body should constrain `inp` and `outp`.

`FunctionInput` is an abstract representation of the inputs to a function. The options are:

- the receiver (`inp.isReceiver()`)
- one of the parameters (`inp.isParameter(i)`)
- one of the results (`inp.isResult(i)`, or `inp.isResult` if there is only one result)

Note that it may seem strange that the result of a function could be considered as a function input, but it is needed in some cases. For instance, the function `bufio.NewWriter` returns a writer `bw` that buffers write operations to an underlying writer `w`. If tainted data is written to `bw`, then it makes sense to propagate that taint back to the underlying writer `w`, which can be modeled by saying that `bufio.NewWriter` propagates taint from its result to its first argument.

Similarly, `FunctionOutput` is an abstract representation of the outputs to a function. The options are:

- the receiver (`outp.isReceiver()`)
- one of the parameters (`outp.isParameter(i)`)
- one of the results (`outp.isResult(i)`, or `outp.isResult` if there is only one result)

Here is an example from `Gin.qll`, which has been slightly simplified.

```
private class ParamsGet extends TaintTracking::FunctionModel, Method {
  ParamsGet() { this.hasQualifiedName("github.com/gin-gonic/gin", "Params", "Get") }

  override predicate hasTaintFlow(FunctionInput inp, FunctionOutput outp) {
    inp.isReceiver() and outp.isResult(0)
  }
}
```

This has the effect that calls to the `Get` method with receiver type `Params` from the `gin-gonic/gin` package allow taint to flow from the receiver to the first result. In other words, if `p` has type `Params` and taint can flow to it, then after the line `x := p.Get("foo")` taint can also flow to `x`.

Sanitizers

It is not necessary to indicate that library functions are sanitizers. Their bodies are not analyzed, so it is assumed that data does not flow through them.

Sinks

Data-flow sinks are specified by queries rather than by library models. However, you can use library models to indicate when functions belong to special categories. Queries can then use these categories when specifying sinks. Classes representing these special categories are contained in `ql/src/semmle/go/Concepts.qll` in the CodeQL for Go repository. `Concepts.qll` includes classes for logger mechanisms, HTTP response writers, HTTP redirects, and marshaling and unmarshaling functions.

Here is a short example from `Stdlib.qll`, which has been slightly simplified.

```
private class PrintfCall extends LoggerCall::Range, DataFlow::CallNode {
  PrintfCall() { this.getTarget().hasQualifiedName("fmt", ["Print", "Printf", "Println
↪"]) }

  override DataFlow::Node getAMessageComponent() { result = this.getAnArgument() }
}
```

This has the effect that any call to `Print`, `Printf`, or `Println` in the package `fmt` is recognized as a logger call. Any query that uses logger calls as a sink will then identify when tainted data has been passed as an argument to `Print`, `Printf`, or `Println`.

- *Basic query for Go code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for Go*: When you're analyzing a Go program, you can make use of the large collection of classes in the CodeQL library for Go.
- *Abstract syntax tree classes for working with Go programs*: CodeQL has a large selection of classes for representing the abstract syntax tree of Go programs.
- *Modeling data flow in Go libraries*: When analyzing a Go program, CodeQL does not examine the source code for external packages. To track the flow of untrusted data through a library, you can create a model of the library.

5.4 CodeQL for Java

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Java codebases.

5.4.1 Basic query for Java code

Learn to write and run a simple CodeQL query using LGTM.

About the query

The query we're going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they have an empty then branch. For example, code such as:

```
if (error) { }
```

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Java** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

   ```
   import java

   from IfStmt ifstmt, Block block
   where ifstmt.getThen() = block and
     block.getNumStmt() = 0
   select ifstmt, "This 'if' statement is redundant."
   ```

   LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   > Note
   >
   > Your query is always run against the most recently analyzed commit to the selected project.

The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

Example query results

Note

An ellipsis (...) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import java`	Imports the standard CodeQL libraries for Java.	Every query begins with one or more `import` statements.
`from IfStmt ifstmt, Block block`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `IfStmt` variable for `if` statements • a `Block` variable for the then block
`where ifstmt.getThen() = block and block.getNumStmt() = 0`	Defines a condition on the variables.	`ifstmt.getThen() = block` relates the two variables. The block must be the `then` branch of the `if` statement. `block.getNumStmt() = 0` states that the block must be empty (that is, it contains no statements).
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if (...) {
  ...
} else if ("-verbose".equals(option)) {
  // nothing to do - handled earlier
} else {
  error("unrecognized option");
}
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to ignore empty `then` branches if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Extend the where clause to include the following extra condition:

   ```
   and not exists(ifstmt.getElse())
   ```

   The `where` clause is now:

   ```
   where ifstmt.getThen() = block and
     block.getNumStmt() = 0 and
     not exists(ifstmt.getElse())
   ```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer included.

See this in the query console

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.2 CodeQL library for Java

When you're analyzing a Java program, you can make use of the large collection of classes in the CodeQL library for Java.

About the CodeQL library for Java

There is an extensive library for analyzing CodeQL databases extracted from Java projects. The classes in this library present the data from a database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `java.qll` imports all the core Java library modules, so you can include the complete library by beginning your query with:

```
import java
```

The rest of this article briefly summarizes the most important classes and predicates provided by this library.

> Note
>
> The example queries in this article illustrate the types of results returned by different library classes. The results themselves are not interesting but can be used as the basis for developing a more complex query. The other articles in this section of the help show how you can take a simple query and fine-tune it to find precisely the results you're interested in.

Summary of the library classes

The most important classes in the standard Java library can be grouped into five main categories:

1. Classes for representing program elements (such as classes and methods)
2. Classes for representing AST nodes (such as statements and expressions)
3. Classes for representing metadata (such as annotations and comments)
4. Classes for computing metrics (such as cyclomatic complexity and coupling)
5. Classes for navigating the program's call graph

We will discuss each of these in turn, briefly describing the most important classes for each category.

Program elements

These classes represent named program elements: packages (`Package`), compilation units (`CompilationUnit`), types (`Type`), methods (`Method`), constructors (`Constructor`), and variables (`Variable`).

Their common superclass is `Element`, which provides general member predicates for determining the name of a program element and checking whether two elements are nested inside each other.

It's often convenient to refer to an element that might either be a method or a constructor; the class `Callable`, which is a common superclass of `Method` and `Constructor`, can be used for this purpose.

Types

Class `Type` has a number of subclasses for representing different kinds of types:

- `PrimitiveType` represents a primitive type, that is, one of `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`; QL also classifies `void` and `<nulltype>` (the type of the `null` literal) as primitive types.
- `RefType` represents a reference (that is, non-primitive) type; it in turn has several subclasses:
  - `Class` represents a Java class.
  - `Interface` represents a Java interface.
  - `EnumType` represents a Java `enum` type.
  - `Array` represents a Java array type.

For example, the following query finds all variables of type `int` in the program:

```
import java

from Variable v, PrimitiveType pt
where pt = v.getType() and
    pt.hasName("int")
select v
```

See this in the query console on LGTM.com. You're likely to get many results when you run this query because most projects contain many variables of type `int`.

Reference types are also categorized according to their declaration scope:

- `TopLevelType` represents a reference type declared at the top-level of a compilation unit.
- `NestedType` is a type declared inside another type.

For instance, this query finds all top-level types whose name is not the same as that of their compilation unit:

```
import java

from TopLevelType tl
where tl.getName() != tl.getCompilationUnit().getName()
select tl
```

See this in the query console on LGTM.com. This pattern is seen in many projects. When we ran it on the LGTM.com demo projects, most of the projects had at least one instance of this problem in the source code. There were many more instances in the files referenced by the source code.

Several more specialized classes are available as well:

- `TopLevelClass` represents a class declared at the top-level of a compilation unit.
- `NestedClass` represents a class declared inside another type, such as:
  - A `LocalClass`, which is a class declared inside a method or constructor.
  - An `AnonymousClass`, which is an anonymous class.

Finally, the library also has a number of singleton classes that wrap frequently used Java standard library classes: `TypeObject`, `TypeCloneable`, `TypeRuntime`, `TypeSerializable`, `TypeString`, `TypeSystem` and `TypeClass`. Each CodeQL class represents the standard Java class suggested by its name.

As an example, we can write a query that finds all nested classes that directly extend `Object`:

```
import java

from NestedClass nc
where nc.getASupertype() instanceof TypeObject
select nc
```

See this in the query console on LGTM.com. You're likely to get many results when you run this query because many projects include nested classes that extend `Object` directly.

Generics

There are also several subclasses of `Type` for dealing with generic types.

A `GenericType` is either a `GenericInterface` or a `GenericClass`. It represents a generic type declaration such as interface `java.util.Map` from the Java standard library:

```
package java.util.;

public interface Map<K, V> {
    int size();

    // ...
}
```

Type parameters, such as `K` and `V` in this example, are represented by class `TypeVariable`.

A parameterized instance of a generic type provides a concrete type to instantiate the type parameter with, as in `Map<String, File>`. Such a type is represented by a `ParameterizedType`, which is distinct from the `GenericType` representing the generic type it was instantiated from. To go from a `ParameterizedType` to its corresponding `GenericType`, you can use predicate `getSourceDeclaration`.

For instance, we could use the following query to find all parameterized instances of `java.util.Map`:

```
import java

from GenericInterface map, ParameterizedType pt
where map.hasQualifiedName("java.util", "Map") and
    pt.getSourceDeclaration() = map
select pt
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects contain parameterized instances of `java.util.Map` in their source code, but they all have results in reference files.

In general, generic types may restrict which types a type parameter can be bound to. For instance, a type of maps from strings to numbers could be declared as follows:

```
class StringToNumMap<N extends Number> implements Map<String, N> {
    // ...
}
```

This means that a parameterized instance of `StringToNumberMap` can only instantiate type parameter `N` with type `Number` or one of its subtypes but not, for example, with `File`. We say that N is a bounded type parameter, with `Number` as its upper bound. In QL, a type variable can be queried for its type bound using predicate `getATypeBound`. The type bounds themselves are represented by class `TypeBound`, which has a member predicate `getType` to retrieve the type the variable is bounded by.

As an example, the following query finds all type variables with type bound `Number`:

```
import java

from TypeVariable tv, TypeBound tb
where tb = tv.getATypeBound() and
    tb.getType().hasQualifiedName("java.lang", "Number")
select tv
```

See this in the query console on LGTM.com. When we ran it on the LGTM.com demo projects, the *neo4j/neo4j*, *hibernate/hibernate-orm* and *apache/hadoop* projects all contained examples of this pattern.

For dealing with legacy code that is unaware of generics, every generic type has a "raw" version without any type parameters. In the CodeQL libraries, raw types are represented using class `RawType`, which has the expected subclasses `RawClass` and `RawInterface`. Again, there is a predicate `getSourceDeclaration` for obtaining the corresponding generic type. As an example, we can find variables of (raw) type `Map`:

```
import java

from Variable v, RawType rt
where rt = v.getType() and
    rt.getSourceDeclaration().hasQualifiedName("java.util", "Map")
select v
```

See this in the query console on LGTM.com. Many projects have variables of raw type `Map`.

For example, in the following code snippet this query would find `m1`, but not `m2`:

```
Map m1 = new HashMap();
Map<String, String> m2 = new HashMap<String, String>();
```

Finally, variables can be declared to be of a wildcard type:

```
Map<? extends Number, ? super Float> m;
```

The wildcards `? extends Number` and `? super Float` are represented by class `WildcardTypeAccess`. Like type parameters, wildcards may have type bounds. Unlike type parameters, wildcards can have upper bounds (as in `? extends Number`), and also lower bounds (as in `? super Float`). Class `WildcardTypeAccess` provides member predicates `getUpperBound` and `getLowerBound` to retrieve the upper and lower bounds, respectively.

For dealing with generic methods, there are classes `GenericMethod`, `ParameterizedMethod` and `RawMethod`, which are entirely analogous to the like-named classes for representing generic types.

For more information on working with types, see the *Types in Java*.

Variables

Class `Variable` represents a variable in the Java sense, which is either a member field of a class (whether static or not), or a local variable, or a parameter. Consequently, there are three subclasses catering to these special cases:

- `Field` represents a Java field.
- `LocalVariableDecl` represents a local variable.
- `Parameter` represents a parameter of a method or constructor.

Abstract syntax tree

Classes in this category represent abstract syntax tree (AST) nodes, that is, statements (class `Stmt`) and expressions (class `Expr`). For a full list of expression and statement types available in the standard QL library, see "*Abstract syntax tree classes for working with Java programs*."

Both `Expr` and `Stmt` provide member predicates for exploring the abstract syntax tree of a program:

- `Expr.getAChildExpr` returns a sub-expression of a given expression.
- `Stmt.getAChild` returns a statement or expression that is nested directly inside a given statement.
- `Expr.getParent` and `Stmt.getParent` return the parent node of an AST node.

For example, the following query finds all expressions whose parents are `return` statements:

```
import java

from Expr e
where e.getParent() instanceof ReturnStmt
select e
```

See this in the query console on LGTM.com. Many projects have examples of `return` statements with child expressions.

Therefore, if the program contains a return statement `return x + y;`, this query will return `x + y`.

As another example, the following query finds statements whose parent is an `if` statement:

```
import java

from Stmt s
where s.getParent() instanceof IfStmt
select s
```

See this in the query console on LGTM.com. Many projects have examples of `if` statements with child statements.

This query will find both `then` branches and `else` branches of all `if` statements in the program.

Finally, here is a query that finds method bodies:

```
import java

from Stmt s
where s.getParent() instanceof Method
select s
```

See this in the query console on LGTM.com. Most projects have many method bodies.

As these examples show, the parent node of an expression is not always an expression: it may also be a statement, for example, an `IfStmt`. Similarly, the parent node of a statement is not always a statement: it may also be a method or a constructor. To capture this, the QL Java library provides two abstract class `ExprParent` and `StmtParent`, the former representing any node that may be the parent node of an expression, and the latter any node that may be the parent node of a statement.

For more information on working with AST classes, see the *article on overflow-prone comparisons in Java*.

Metadata

Java programs have several kinds of metadata, in addition to the program code proper. In particular, there are annotations and Javadoc comments. Since this metadata is interesting both for enhancing code analysis and as an analysis subject in its own right, the QL library defines classes for accessing it.

For annotations, class `Annotatable` is a superclass of all program elements that can be annotated. This includes packages, reference types, fields, methods, constructors, and local variable declarations. For every such element, its predicate `getAnAnnotation` allows you to retrieve any annotations the element may have. For example, the following query finds all annotations on constructors:

```
import java

from Constructor c
select c.getAnAnnotation()
```

See this in the query console on LGTM.com. The LGTM.com demo projects all use annotations, you can see examples where they are used to suppress warnings and mark code as deprecated.

These annotations are represented by class `Annotation`. An annotation is simply an expression whose type is an `AnnotationType`. For example, you can amend this query so that it only reports deprecated constructors:

```
import java

from Constructor c, Annotation ann, AnnotationType anntp
where ann = c.getAnAnnotation() and
    anntp = ann.getType() and
    anntp.hasQualifiedName("java.lang", "Deprecated")
select ann
```

See this in the query console on LGTM.com. Only constructors with the `@Deprecated` annotation are reported this time.

For more information on working with annotations, see the *article on annotations*.

For Javadoc, class `Element` has a member predicate `getDoc` that returns a delegate `Documentable` object, which can then be queried for its attached Javadoc comments. For example, the following query finds Javadoc comments on private fields:

```
import java

from Field f, Javadoc jdoc
where f.isPrivate() and
    jdoc = f.getDoc().getJavadoc()
select jdoc
```

See this in the query console on LGTM.com. You can see this pattern in many projects.

Class `Javadoc` represents an entire Javadoc comment as a tree of `JavadocElement` nodes, which can be traversed using member predicates `getAChild` and `getParent`. For instance, you could edit the query so that it finds all `@author` tags in Javadoc comments on private fields:

```
import java

from Field f, Javadoc jdoc, AuthorTag at
where f.isPrivate() and
```

(continues on next page)

(continued from previous page)

```
    jdoc = f.getDoc().getJavadoc() and
    at.getParent+() = jdoc
select at
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects uses the `@author` tag on private fields.

> Note
>
> On line 5 we used `getParent+` to capture tags that are nested at any depth within the Javadoc comment.

For more information on working with Javadoc, see the *article on Javadoc*.

Metrics

The standard QL Java library provides extensive support for computing metrics on Java program elements. To avoid overburdening the classes representing those elements with too many member predicates related to metric computations, these predicates are made available on delegate classes instead.

Altogether, there are six such classes: `MetricElement`, `MetricPackage`, `MetricRefType`, `MetricField`, `MetricCallable`, and `MetricStmt`. The corresponding element classes each provide a member predicate `getMetrics` that can be used to obtain an instance of the delegate class, on which metric computations can then be performed.

For example, the following query finds methods with a cyclomatic complexity greater than 40:

```
import java

from Method m, MetricCallable mc
where mc = m.getMetrics() and
    mc.getCyclomaticComplexity() > 40
select m
```

See this in the query console on LGTM.com. Most large projects include some methods with a very high cyclomatic complexity. These methods are likely to be difficult to understand and test.

Call graph

CodeQL databases generated from Java code bases include precomputed information about the program's call graph, that is, which methods or constructors a given call may dispatch to at runtime.

The class `Callable`, introduced above, includes both methods and constructors. Call expressions are abstracted using class `Call`, which includes method calls, `new` expressions, and explicit constructor calls using `this` or `super`.

We can use predicate `Call.getCallee` to find out which method or constructor a specific call expression refers to. For example, the following query finds all calls to methods called `println`:

```
import java

from Call c, Method m
where m = c.getCallee() and
    m.hasName("println")
select c
```

See this in the query console on LGTM.com. The LGTM.com demo projects all include many calls to methods of this name.

Conversely, `Callable.getAReference` returns a `Call` that refers to it. So we can find methods and constructors that are never called using this query:

```
import java

from Callable c
where not exists(c.getAReference())
select c
```

See this in the query console on LGTM.com. The LGTM.com demo projects all appear to have many methods that are not called directly, but this is unlikely to be the whole story. To explore this area further, see "*Navigating the call graph*."

For more information about callables and calls, see the *article on the call graph*.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.3 Analyzing data flow in Java

You can use CodeQL to track the flow of data through a Java program to its use.

About this article

This article describes how data flow analysis is implemented in the CodeQL libraries for Java and includes examples to help you write your own data flow queries. The following sections describe how to use the libraries for local data flow, global data flow, and taint tracking.

For a more general introduction to modeling data flow, see "*About data flow analysis*."

Local data flow

Local data flow is data flow within a single method or callable. Local data flow is usually easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. `Node`s are divided into expression nodes (`ExprNode`) and parameter nodes (`ParameterNode`). You can map between data flow nodes and expressions/parameters using the member predicates `asExpr` and `asParameter`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the parameter corresponding to this node, if any. */
  Parameter asParameter() { ... }

  ...
}
```

or using the predicates `exprNode` and `parameterNode`:

```
/**
 * Gets the node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }

/**
 * Gets the node corresponding to the value of parameter `p` at function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively by using the + and * operators, or by using the predefined recursive predicate `localFlow`, which is equivalent to `localFlowStep*`.

For example, you can find flow from a parameter `source` to an expression `sink` in zero or more local steps:

```
DataFlow::localFlow(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
String temp = x;
String y = temp + ", " + temp;
```

If `x` is a tainted string then `y` is also tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint propagation edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively by using the + and * operators, or by using the predefined recursive predicate `localTaint`, which is equivalent to `localTaintStep*`.

For example, you can find taint propagation from a parameter `source` to an expression `sink` in zero or more local steps:

```
TaintTracking::localTaint(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Examples

This query finds the filename passed to `new FileReader(..)`.

```
import java

from Constructor fileReader, Call call
where
  fileReader.getDeclaringType().hasQualifiedName("java.io", "FileReader") and
  call.getCallee() = fileReader
select call.getArgument(0)
```

Unfortunately, this only gives the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import java
import semmle.code.java.dataflow.DataFlow

from Constructor fileReader, Call call, Expr src
where
  fileReader.getDeclaringType().hasQualifiedName("java.io", "FileReader") and
  call.getCallee() = fileReader and
  DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(call.getArgument(0)))
select src
```

Then we can make the source more specific, for example an access to a public parameter. This query finds where a public parameter is passed to `new FileReader(..)`:

```
import java
import semmle.code.java.dataflow.DataFlow

from Constructor fileReader, Call call, Parameter p
where
  fileReader.getDeclaringType().hasQualifiedName("java.io", "FileReader") and
  call.getCallee() = fileReader and
  DataFlow::localFlow(DataFlow::parameterNode(p), DataFlow::exprNode(call.
↪getArgument(0)))
select p
```

This query finds calls to formatting functions where the format string is not hard-coded.

```
import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.StringFormat

from StringFormatMethod format, MethodAccess call, Expr formatString
where
  call.getMethod() = format and
  call.getArgument(format.getFormatStringIndex()) = formatString and
  not exists(DataFlow::Node source, DataFlow::Node sink |
    DataFlow::localFlow(source, sink) and
    source.asExpr() instanceof StringLiteral and
    sink.asExpr() = formatString
```

(continues on next page)

(continued from previous page)

```
  )
select call, "Argument to String format method isn't hard-coded."
```

Exercises

Exercise 1: Write a query that finds all hard-coded strings used to create a `java.net.URL`, using local data flow. (*Answer*)

Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see *Creating path queries*.

Using global data flow

You use the global data flow library by extending the class `DataFlow::Configuration`:

```
import semmle.code.java.dataflow.DataFlow

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource`—defines where data may flow from
- `isSink`—defines where data may flow to
- `isBarrier`—optional, restricts the data flow
- `isAdditionalFlowStep`—optional, adds additional flow steps

The characteristic predicate `MyDataFlowConfiguration()` defines the name of the configuration, so `"MyDataFlowConfiguration"` should be a unique name, for example, the name of your class.

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Data flow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow as local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. You use the global taint tracking library by extending the class `TaintTracking::Configuration`:

```
import semmle.code.java.dataflow.TaintTracking

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource`—defines where taint may flow from
- `isSink`—defines where taint may flow to
- `isSanitizer`—optional, restricts the taint flow
- `isAdditionalTaintStep`—optional, adds additional taint steps

Similar to global data flow, the characteristic predicate `MyTaintTrackingConfiguration()` defines the unique name of the configuration.

The taint tracking analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Flow sources

The data flow library contains some predefined flow sources. The class `RemoteFlowSource` (defined in `semmle.code.java.dataflow.FlowSources`) represents data flow sources that may be controlled by a remote user, which is useful for finding security problems.

Examples

This query shows a taint-tracking configuration that uses remote user input as data sources.

```
import java
import semmle.code.java.dataflow.FlowSources

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() {
    this = "..."
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  ...
}
```

Exercises

Exercise 2: Write a query that finds all hard-coded strings used to create a `java.net.URL`, using global data flow. (*Answer*)

Exercise 3: Write a class that represents flow sources from `java.lang.System.getenv(..)`. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query which finds all global data flows from `getenv` to `java.net.URL`. (*Answer*)

Answers

Exercise 1

```
import semmle.code.java.dataflow.DataFlow

from Constructor url, Call call, StringLiteral src
where
  url.getDeclaringType().hasQualifiedName("java.net", "URL") and
  call.getCallee() = url and
  DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(call.getArgument(0)))
select src
```

Exercise 2

```
import semmle.code.java.dataflow.DataFlow

class Configuration extends DataFlow::Configuration {
  Configuration() {
    this = "LiteralToURL Configuration"
  }

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
      call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("java.net", "URL
↪")
    )
  }
}

from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This string constructs a URL $@.", sink, "here"
```

Exercise 3

```
import java

class GetenvSource extends MethodAccess {
  GetenvSource() {
    exists(Method m | m = this.getMethod() |
      m.hasName("getenv") and
      m.getDeclaringType() instanceof TypeSystem
    )
  }
}
```

Exercise 4

```
import semmle.code.java.dataflow.DataFlow

class GetenvSource extends DataFlow::ExprNode {
  GetenvSource() {
    exists(Method m | m = this.asExpr().(MethodAccess).getMethod() |
      m.hasName("getenv") and
      m.getDeclaringType() instanceof TypeSystem
    )
```

(continues on next page)

(continued from previous page)

```
  }
}

class GetenvToURLConfiguration extends DataFlow::Configuration {
  GetenvToURLConfiguration() {
    this = "GetenvToURLConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof GetenvSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
      call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("java.net", "URL
↪")
    )
  }
}

from DataFlow::Node src, DataFlow::Node sink, GetenvToURLConfiguration config
where config.hasFlow(src, sink)
select src, "This environment variable constructs a URL $@.", sink, "here"
```

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.4 Types in Java

You can use CodeQL to find out information about data types used in Java code. This allows you to write queries to identify specific type-related issues.

About working with Java types

The standard CodeQL library represents Java types by means of the `Type` class and its various subclasses.

In particular, class `PrimitiveType` represents primitive types that are built into the Java language (such as `boolean` and `int`), whereas `RefType` and its subclasses represent reference types, that is classes, interfaces, array types, and so on. This includes both types from the Java standard library (like `java.lang.Object`) and types defined by non-library code.

Class `RefType` also models the class hierarchy: member predicates `getASupertype` and `getASubtype` allow you to find a reference type's immediate super types and sub types. For example, consider the following Java program:

```
class A {}

interface I {}

class B extends A implements I {}
```

Here, class `A` has exactly one immediate super type (`java.lang.Object`) and exactly one immediate sub type (`B`); the same is true of interface `I`. Class `B`, on the other hand, has two immediate super types (`A` and `I`), and no immediate sub types.

To determine ancestor types (including immediate super types, and also *their* super types, etc.), we can use transitive closure. For example, to find all ancestors of `B` in the example above, we could use the following query:

```
import java

from Class B
where B.hasName("B")
select B.getASupertype+()
```

See this in the query console on LGTM.com. If this query were run on the example snippet above, the query would return `A`, `I`, and `java.lang.Object`.

> Tip
>
> If you want to see the location of `B` as well as `A`, you can replace `B.getASupertype+()` with `B.getASupertype*()` and re-run the query.

Besides class hierarchy modeling, `RefType` also provides member predicate `getAMember` for accessing members (that is, fields, constructors, and methods) declared in the type, and predicate `inherits(Method m)` for checking whether the type either declares or inherits a method `m`.

Example: Finding problematic array casts

As an example of how to use the class hierarchy API, we can write a query that finds downcasts on arrays, that is, cases where an expression `e` of some type `A[]` is converted to type `B[]`, such that `B` is a (not necessarily immediate) subtype of `A`.

This kind of cast is problematic, since downcasting an array results in a runtime exception, even if every individual array element could be downcast. For example, the following code throws a `ClassCastException`:

```
Object[] o = new Object[] { "Hello", "world" };
String[] s = (String[])o;
```

If the expression `e` happens to actually evaluate to a `B[]` array, on the other hand, the cast will succeed:

```
Object[] o = new String[] { "Hello", "world" };
String[] s = (String[])o;
```

In this tutorial, we don't try to distinguish these two cases. Our query should simply look for cast expressions `ce` that cast from some type `source` to another type `target`, such that:

- Both `source` and `target` are array types.
- The element type of `source` is a transitive super type of the element type of `target`.

This recipe is not too difficult to translate into a query:

```
import java

from CastExpr ce, Array source, Array target
where source = ce.getExpr().getType() and
    target = ce.getType() and
    target.getElementType().(RefType).getASupertype+() = source.getElementType()
select ce, "Potentially problematic array downcast."
```

See this in the query console on LGTM.com. Many projects return results for this query.

Note that by casting `target.getElementType()` to a `RefType`, we eliminate all cases where the element type is a primitive type, that is, `target` is an array of primitive type: the problem we are looking for cannot arise in that case. Unlike in Java, a cast in QL never fails: if an expression cannot be cast to the desired type, it is simply excluded from the query results, which is exactly what we want.

Improvements

Running this query on old Java code, before version 5, often returns many false positive results arising from uses of the method `Collection.toArray(T[])`, which converts a collection into an array of type `T[]`.

In code that does not use generics, this method is often used in the following way:

```
List l = new ArrayList();
// add some elements of type A to l
A[] as = (A[])l.toArray(new A[0]);
```

Here, `l` has the raw type `List`, so `l.toArray` has return type `Object[]`, independent of the type of its argument array. Hence the cast goes from `Object[]` to `A[]` and will be flagged as problematic by our query, although at runtime this cast can never go wrong.

To identify these cases, we can create two CodeQL classes that represent, respectively, the `Collection.toArray` method, and calls to this method or any method that overrides it:

```
/** class representing java.util.Collection.toArray(T[]) */
class CollectionToArray extends Method {
    CollectionToArray() {
        this.getDeclaringType().hasQualifiedName("java.util", "Collection") and
        this.hasName("toArray") and
        this.getNumberOfParameters() = 1
    }
}

/** class representing calls to java.util.Collection.toArray(T[]) */
class CollectionToArrayCall extends MethodAccess {
    CollectionToArrayCall() {
        exists(CollectionToArray m |
            this.getMethod().getSourceDeclaration().overridesOrInstantiates*(m)
        )
    }

    /** the call's actual return type, as determined from its argument */
    Array getActualReturnType() {
        result = this.getArgument(0).getType()
    }
}
```

Notice the use of `getSourceDeclaration` and `overridesOrInstantiates` in the constructor of `CollectionToArrayCall`: we want to find calls to `Collection.toArray` and to any method that overrides it, as well as any parameterized instances of these methods. In our example above, for instance, the call `l.toArray` resolves to method `toArray` in the raw class `ArrayList`. Its source declaration is `toArray` in the generic class `ArrayList<T>`, which overrides `AbstractCollection<T>.toArray`, which in turn overrides `Collection<T>.toArray`, which is an instantiation of `Collection.toArray` (since the type parameter `T` in the overridden method belongs to `ArrayList` and is an instantiation of the type parameter belonging to `Collection`).

Using these new classes we can extend our query to exclude calls to `toArray` on an argument of type `A[]` which are then cast to `A[]`:

```
import java

// Insert the class definitions from above

from CastExpr ce, Array source, Array target
where source = ce.getExpr().getType() and
    target = ce.getType() and
    target.getElementType().(RefType).getASupertype+() = source.getElementType() and
    not ce.getExpr().(CollectionToArrayCall).getActualReturnType() = target
select ce, "Potentially problematic array downcast."
```

See this in the query console on LGTM.com. Notice that fewer results are found by this improved query.

Example: Finding mismatched contains checks

We'll now develop a query that finds uses of `Collection.contains` where the type of the queried element is unrelated to the element type of the collection, which guarantees that the test will always return `false`.

For example, Apache Zookeeper used to have a snippet of code similar to the following in class `QuorumPeerConfig`:

```
Map<Object, Object> zkProp;

// ...

if (zkProp.entrySet().contains("dynamicConfigFile")){
    // ...
}
```

Since `zkProp` is a map from `Object` to `Object`, `zkProp.entrySet` returns a collection of type `Set<Entry<Object, Object>>`. Such a set cannot possibly contain an element of type `String`. (The code has since been fixed to use `zkProp.containsKey`.)

In general, we want to find calls to `Collection.contains` (or any of its overriding methods in any parameterized instance of `Collection`), such that the type `E` of collection elements and the type `A` of the argument to `contains` are unrelated, that is, they have no common subtype.

We start by creating a class that describes `java.util.Collection`:

```
class JavaUtilCollection extends GenericInterface {
    JavaUtilCollection() {
        this.hasQualifiedName("java.util", "Collection")
    }
}
```

To make sure we have not mistyped anything, we can run a simple test query:

```
from JavaUtilCollection juc
select juc
```

This query should return precisely one result.

Next, we can create a class that describes `java.util.Collection.contains`:

```
class JavaUtilCollectionContains extends Method {
    JavaUtilCollectionContains() {
        this.getDeclaringType() instanceof JavaUtilCollection and
        this.hasStringSignature("contains(Object)")
    }
}
```

Notice that we use `hasStringSignature` to check that:

- The method in question has name `contains`.
- It has exactly one argument.
- The type of the argument is `Object`.

Alternatively, we could have implemented these three checks more verbosely using `hasName`, `getNumberOfParameters`, and `getParameter(0).getType() instanceof TypeObject`.

As before, it is a good idea to test the new class by running a simple query to select all instances of `JavaUtilCollectionContains`; again there should only be a single result.

Now we want to identify all calls to `Collection.contains`, including any methods that override it, and considering all parameterized instances of `Collection` and its subclasses. That is, we are looking for method accesses where the source declaration of the invoked method (reflexively or transitively) overrides `Collection.contains`. We encode this in a CodeQL class `JavaUtilCollectionContainsCall`:

```
class JavaUtilCollectionContainsCall extends MethodAccess {
    JavaUtilCollectionContainsCall() {
        exists(JavaUtilCollectionContains jucc |
            this.getMethod().getSourceDeclaration().overrides*(jucc)
        )
    }
}
```

This definition is slightly subtle, so you should run a short query to test that `JavaUtilCollectionContainsCall` correctly identifies calls to `Collection.contains`.

For every call to `contains`, we are interested in two things: the type of the argument, and the element type of the collection on which it is invoked. So we need to add two member predicates `getArgumentType` and `getCollectionElementType` to class `JavaUtilCollectionContainsCall` to compute this information.

The former is easy:

```
Type getArgumentType() {
    result = this.getArgument(0).getType()
}
```

For the latter, we proceed as follows:

- Find the declaring type `D` of the `contains` method being invoked.
- Find a (reflexive or transitive) super type `S` of `D` that is a parameterized instance of `java.util.Collection`.
- Return the (only) type argument of `S`.

We encode this as follows:

```
Type getCollectionElementType() {
    exists(RefType D, ParameterizedInterface S |
        D = this.getMethod().getDeclaringType() and
        D.hasSupertype*(S) and S.getSourceDeclaration() instanceof JavaUtilCollection and
        result = S.getTypeArgument(0)
    )
}
```

Having added these two member predicates to `JavaUtilCollectionContainsCall`, we need to write a predicate that checks whether two given reference types have a common subtype:

```
predicate haveCommonDescendant(RefType tp1, RefType tp2) {
    exists(RefType commondesc | commondesc.hasSupertype*(tp1) and commondesc.
↪hasSupertype*(tp2))
}
```

Now we are ready to write a first version of our query:

```
import java

// Insert the class definitions from above

from JavaUtilCollectionContainsCall juccc, Type collEltType, Type argType
where collEltType = juccc.getCollectionElementType() and argType = juccc.
↪getArgumentType() and
    not haveCommonDescendant(collEltType, argType)
select juccc, "Element type " + collEltType + " is incompatible with argument type " +␣
↪argType
```

See this in the query console on LGTM.com.

Improvements

For many programs, this query yields a large number of false positive results due to type variables and wild cards: if the collection element type is some type variable `E` and the argument type is `String`, for example, CodeQL will consider that the two have no common subtype, and our query will flag the call. An easy way to exclude such false positive results is to simply require that neither `collEltType` nor `argType` are instances of `TypeVariable`.

Another source of false positives is autoboxing of primitive types: if, for example, the collection's element type is `Integer` and the argument is of type `int`, predicate `haveCommonDescendant` will fail, since `int` is not a `RefType`. To account for this, our query should check that `collEltType` is not the boxed type of `argType`.

Finally, `null` is special because its type (known as `<nulltype>` in the CodeQL library) is compatible with every reference type, so we should exclude it from consideration.

Adding these three improvements, our final query becomes:

```
import java

// Insert the class definitions from above

from JavaUtilCollectionContainsCall juccc, Type collEltType, Type argType
where collEltType = juccc.getCollectionElementType() and argType = juccc.
↪getArgumentType() and
    not haveCommonDescendant(collEltType, argType) and
    not collEltType instanceof TypeVariable and not argType instanceof TypeVariable and
    not collEltType = argType.(PrimitiveType).getBoxedType() and
    not argType.hasName("<nulltype>")
select juccc, "Element type " + collEltType + " is incompatible with argument type " +␣
↪argType
```

See the full query in the query console on LGTM.com.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.5 Overflow-prone comparisons in Java

You can use CodeQL to check for comparisons in Java code where one side of the comparison is prone to overflow.

About this article

In this tutorial article you'll write a query for finding comparisons between integers and long integers in loops that may lead to non-termination due to overflow.

To begin, consider this code snippet:

```
void foo(long l) {
    for(int i=0; i<l; i++) {
        // do something
    }
}
```

If `l` is bigger than 2^{31}- 1 (the largest positive value of type `int`), then this loop will never terminate: `i` will start at zero, being incremented all the way up to 2^{31}- 1, which is still smaller than `l`. When it is incremented once more, an arithmetic overflow occurs, and `i` becomes -2^{31}, which also is smaller than `l`! Eventually, `i` will reach zero again, and the cycle repeats.

> More about overflow
>
> All primitive numeric types have a maximum value, beyond which they will wrap around to their lowest possible value (called an "overflow"). For `int`, this maximum value is 2^{31}- 1. Type `long` can accommodate larger values up to a maximum of 2^{63}- 1. In this example, this means that `l` can take on a value that is higher than the maximum for type `int`; `i` will never be able to reach this value, instead overflowing and returning to a low value.

We're going to develop a query that finds code that looks like it might exhibit this kind of behavior. We'll be using several of the standard library classes for representing statements and functions. For a full list, see "*Abstract syntax tree classes for working with Java programs*."

Initial query

We'll start by writing a query that finds less-than expressions (CodeQL class `LTExpr`) where the left operand is of type `int` and the right operand is of type `long`:

```
import java

from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
```

(continues on next page)

(continued from previous page)

```
    expr.getRightOperand().getType().hasName("long")
select expr
```

See this in the query console on LGTM.com. This query usually finds results on most projects.

Notice that we use the predicate `getType` (available on all subclasses of `Expr`) to determine the type of the operands. Types, in turn, define the `hasName` predicate, which allows us to identify the primitive types `int` and `long`. As it stands, this query finds *all* less-than expressions comparing `int` and `long`, but in fact we are only interested in comparisons that are part of a loop condition. Also, we want to filter out comparisons where either operand is constant, since these are less likely to be real bugs. The revised query looks like this:

```
import java

from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long") and
    exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr) and
    not expr.getAnOperand().isCompileTimeConstant()
select expr
```

See this in the query console on LGTM.com. Notice that fewer results are found.

The class `LoopStmt` is a common superclass of all loops, including, in particular, `for` loops as in our example above. While different kinds of loops have different syntax, they all have a loop condition, which can be accessed through predicate `getCondition`. We use the reflexive transitive closure operator `*` applied to the `getAChildExpr` predicate to express the requirement that `expr` should be nested inside the loop condition. In particular, it can be the loop condition itself.

The final conjunct in the `where` clause takes advantage of the fact that *predicates* can return more than one value (they are really relations). In particular, `getAnOperand` may return *either* operand of `expr`, so `expr.getAnOperand().isCompileTimeConstant()` holds if at least one of the operands is constant. Negating this condition means that the query will only find expressions where *neither* of the operands is constant.

Generalizing the query

Of course, comparisons between `int` and `long` are not the only problematic case: any less-than comparison between a narrower and a wider type is potentially suspect, and less-than-or-equals, greater-than, and greater-than-or-equals comparisons are just as problematic as less-than comparisons.

In order to compare the ranges of types, we define a predicate that returns the width (in bits) of a given integral type:

```
int width(PrimitiveType pt) {
    (pt.hasName("byte") and result=8) or
    (pt.hasName("short") and result=16) or
    (pt.hasName("char") and result=16) or
    (pt.hasName("int") and result=32) or
    (pt.hasName("long") and result=64)
}
```

We now want to generalize our query to apply to any comparison where the width of the type on the smaller end of the comparison is less than the width of the type on the greater end. Let's call such a comparison *overflow prone*, and introduce an abstract class to model it:

```
abstract class OverflowProneComparison extends ComparisonExpr {
    Expr getLesserOperand() { none() }
    Expr getGreaterOperand() { none() }
}
```

There are two concrete child classes of this class: one for <= or < comparisons, and one for >= or > comparisons. In both cases, we implement the constructor in such a way that it only matches the expressions we want:

```
class LTOverflowProneComparison extends OverflowProneComparison {
    LTOverflowProneComparison() {
        (this instanceof LEExpr or this instanceof LTExpr) and
        width(this.getLeftOperand().getType()) < width(this.getRightOperand().getType())
    }
}

class GTOverflowProneComparison extends OverflowProneComparison {
    GTOverflowProneComparison() {
        (this instanceof GEExpr or this instanceof GTExpr) and
        width(this.getRightOperand().getType()) < width(this.getLeftOperand().getType())
    }
}
```

Now we rewrite our query to make use of these new classes:

```
import Java

// Insert the class definitions from above

from OverflowProneComparison expr
where exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr) and
not expr.getAnOperand().isCompileTimeConstant()
select expr
```

See the full query in the query console on LGTM.com.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.6 Navigating the call graph

CodeQL has classes for identifying code that calls other code, and code that can be called from elsewhere. This allows you to find, for example, methods that are never used.

Call graph classes

The CodeQL library for Java provides two abstract classes for representing a program's call graph: `Callable` and `Call`. The former is simply the common superclass of `Method` and `Constructor`, the latter is a common superclass of `MethodAccess`, `ClassInstanceExpression`, `ThisConstructorInvocationStmt` and `SuperConstructorInvocationStmt`. Simply put, a `Callable` is something that can be invoked, and a `Call` is something that invokes a `Callable`.

For example, in the following program all callables and calls have been annotated with comments:

```
class Super {
    int x;

    // callable
    public Super() {
        this(23);       // call
    }

    // callable
    public Super(int x) {
        this.x = x;
    }

    // callable
    public int getX() {
        return x;
    }
}

    class Sub extends Super {
    // callable
    public Sub(int x) {
        super(x+19);    // call
    }

    // callable
    public int getX() {
        return x-19;
    }
}

class Client {
    // callable
    public static void main(String[] args) {
        Super s = new Sub(42);  // call
        s.getX();               // call
    }
}
```

Class `Call` provides two call graph navigation predicates:

- `getCallee` returns the `Callable` that this call (statically) resolves to; note that for a call to an instance (that is, non-static) method, the actual method invoked at runtime may be some other method that overrides this method.
- `getCaller` returns the `Callable` of which this call is syntactically part.

For instance, in our example `getCallee` of the second call in `Client.main` would return `Super.getX`. At runtime, though, this call would actually invoke `Sub.getX`.

Class `Callable` defines a large number of member predicates; for our purposes, the two most important ones are:

- `calls(Callable target)` succeeds if this callable contains a call whose callee is `target`.
- `polyCalls(Callable target)` succeeds if this callable may call `target` at runtime; this is the case if it contains a call whose callee is either `target` or a method that `target` overrides.

In our example, `Client.main` calls the constructor `Sub(int)` and the method `Super.getX`; additionally, it `polyCalls` method `Sub.getX`.

Example: Finding unused methods

We can use the `Callable` class to write a query that finds methods that are not called by any other method:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee))
select callee
```

See this in the query console on LGTM.com. This simple query typically returns a large number of results.

> Note
>
> We have to use `polyCalls` instead of `calls` here: we want to be reasonably sure that `callee` is not called, either directly or via overriding.

Running this query on a typical Java project results in lots of hits in the Java standard library. This makes sense, since no single client program uses every method of the standard library. More generally, we may want to exclude methods and constructors from compiled libraries. We can use the predicate `fromSource` to check whether a compilation unit is a source file, and refine our query:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource()
select callee, "Not called."
```

See this in the query console on LGTM.com. This change reduces the number of results returned for most projects.

We might also notice several unused methods with the somewhat strange name `<clinit>`: these are class initializers; while they are not explicitly called anywhere in the code, they are called implicitly whenever the surrounding class is loaded. Hence it makes sense to exclude them from our query. While we are at it, we can also exclude finalizers, which are similarly invoked implicitly:

```
import java

from Callable callee
```

(continues on next page)

(continued from previous page)

```
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize")
select callee, "Not called."
```

See this in the query console on LGTM.com. This also reduces the number of results returned by most projects.

We may also want to exclude public methods from our query, since they may be external API entry points:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize") and
    not callee.isPublic()
select callee, "Not called."
```

See this in the query console on LGTM.com. This should have a more noticeable effect on the number of results returned.

A further special case is non-public default constructors: in the singleton pattern, for example, a class is provided with private empty default constructor to prevent it from being instantiated. Since the very purpose of such constructors is their not being called, they should not be flagged up:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize") and
    not callee.isPublic() and
    not callee.(Constructor).getNumberOfParameters() = 0
select callee, "Not called."
```

See this in the query console on LGTM.com. This change has a large effect on the results for some projects but little effect on the results for others. Use of this pattern varies widely between different projects.

Finally, on many Java projects there are methods that are invoked indirectly by reflection. So, while there are no calls invoking these methods, they are, in fact, used. It is in general very hard to identify such methods. A very common special case, however, is JUnit test methods, which are reflectively invoked by a test runner. The CodeQL library for Java has support for recognizing test classes of JUnit and other testing frameworks, which we can employ to filter out methods defined in such classes:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize") and
    not callee.isPublic() and
    not callee.(Constructor).getNumberOfParameters() = 0 and
    not callee.getDeclaringType() instanceof TestClass
select callee, "Not called."
```

See this in the query console on LGTM.com. This should give a further reduction in the number of results returned.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.7 Annotations in Java

CodeQL databases of Java projects contain information about all annotations attached to program elements.

About working with annotations

Annotations are represented by these CodeQL classes:

- The class `Annotatable` represents all entities that may have an annotation attached to them (that is, packages, reference types, fields, methods, and local variables).
- The class `AnnotationType` represents a Java annotation type, such as `java.lang.Override`; annotation types are interfaces.
- The class `AnnotationElement` represents an annotation element, that is, a member of an annotation type.
- The class `Annotation` represents an annotation such as `@Override`; annotation values can be accessed through member predicate `getValue`.

For example, the Java standard library defines an annotation `SuppressWarnings` that instructs the compiler not to emit certain kinds of warnings:

```
package java.lang;

public @interface SuppressWarnings {
    String[] value;
}
```

`SuppressWarnings` is represented as an `AnnotationType`, with `value` as its only `AnnotationElement`.

A typical usage of `SuppressWarnings` would be this annotation for preventing a warning about using raw types:

```
class A {
    @SuppressWarnings("rawtypes")
    public A(java.util.List rawlist) {
    }
}
```

The expression `@SuppressWarnings("rawtypes")` is represented as an `Annotation`. The string literal `"rawtypes"` is used to initialize the annotation element `value`, and its value can be extracted from the annotation by means of the `getValue` predicate.

We could then write this query to find all `@SuppressWarnings` annotations attached to constructors, and return both the annotation itself and the value of its `value` element:

```
import java

from Constructor c, Annotation ann, AnnotationType anntp
where ann = c.getAnAnnotation() and
    anntp = ann.getType() and
    anntp.hasQualifiedName("java.lang", "SuppressWarnings")
select ann, ann.getValue("value")
```

See the full query in the query console on LGTM.com. Several of the LGTM.com demo projects use the `@SuppressWarnings` annotation. Looking at the `values` of the annotation element returned by the query, we can see that the *apache/activemq* project uses the `"rawtypes"` value described above.

As another example, this query finds all annotation types that only have a single annotation element, which has name `value`:

```
import java

from AnnotationType anntp
where forex(AnnotationElement elt |
    elt = anntp.getAnAnnotationElement() |
    elt.getName() = "value"
)
select anntp
```

See the full query in the query console on LGTM.com.

Example: Finding missing `@Override` annotations

In newer versions of Java, it's recommended (though not required) that you annotate methods that override another method with an `@Override` annotation. These annotations, which are checked by the compiler, serve as documentation, and also help you avoid accidental overloading where overriding was intended.

For example, consider this example program:

```
class Super {
    public void m() {}
}

class Sub1 extends Super {
    @Override public void m() {}
}

class Sub2 extends Super {
    public void m() {}
}
```

Here, both `Sub1.m` and `Sub2.m` override `Super.m`, but only `Sub1.m` is annotated with `@Override`.

We'll now develop a query for finding methods like `Sub2.m` that should be annotated with `@Override`, but are not.

As a first step, let's write a query that finds all `@Override` annotations. Annotations are expressions, so their type can be accessed using `getType`. Annotation types, on the other hand, are interfaces, so their qualified name can be queried using `hasQualifiedName`. Therefore we can implement the query like this:

```
import java

from Annotation ann
where ann.getType().hasQualifiedName("java.lang", "Override")
select ann
```

As always, it is a good idea to try this query on a CodeQL database for a Java project to make sure it actually produces some results. On the earlier example, it should find the annotation on `Sub1.m`. Next, we encapsulate the concept of an `@Override` annotation as a CodeQL class:

```
class OverrideAnnotation extends Annotation {
    OverrideAnnotation() {
        this.getType().hasQualifiedName("java.lang", "Override")
    }
}
```

This makes it very easy to write our query for finding methods that override another method, but don't have an `@Override` annotation: we use predicate `overrides` to find out whether one method overrides another, and predicate `getAnAnnotation` (available on any `Annotatable`) to retrieve some annotation.

```
import java

from Method overriding, Method overridden
where overriding.overrides(overridden) and
    not overriding.getAnAnnotation() instanceof OverrideAnnotation
select overriding, "Method overrides another method, but does not have an @Override␣
↪annotation."
```

See this in the query console on LGTM.com. In practice, this query may yield many results from compiled library code, which aren't very interesting. It's therefore a good idea to add another conjunct `overriding.fromSource()` to restrict the result to only report methods for which source code is available.

Example: Finding calls to deprecated methods

As another example, we can write a query that finds calls to methods marked with a `@Deprecated` annotation.

For example, consider this example program:

```
class A {
    @Deprecated void m() {}

    @Deprecated void n() {
        m();
    }

    void r() {
        m();
    }
}
```

Here, both `A.m` and `A.n` are marked as deprecated. Methods `n` and `r` both call `m`, but note that `n` itself is deprecated, so we probably should not warn about this call.

As in the previous example, we'll start by defining a class for representing `@Deprecated` annotations:

```
class DeprecatedAnnotation extends Annotation {
    DeprecatedAnnotation() {
        this.getType().hasQualifiedName("java.lang", "Deprecated")
    }
}
```

Now we can define a class for representing deprecated methods:

```
class DeprecatedMethod extends Method {
    DeprecatedMethod() {
        this.getAnAnnotation() instanceof DeprecatedAnnotation
    }
}
```

Finally, we use these classes to find calls to deprecated methods, excluding calls that themselves appear in deprecated methods:

```
import java

from Call call
where call.getCallee() instanceof DeprecatedMethod
    and not call.getCaller() instanceof DeprecatedMethod
select call, "This call invokes a deprecated method."
```

In our example, this query flags the call to `A.m` in `A.r`, but not the one in `A.n`.

For more information about the class `Call`, see "*Navigating the call graph*."

Improvements

The Java standard library provides another annotation type `java.lang.SupressWarnings` that can be used to suppress certain categories of warnings. In particular, it can be used to turn off warnings about calls to deprecated methods. Therefore, it makes sense to improve our query to ignore calls to deprecated methods from inside methods that are marked with `@SuppressWarnings("deprecated")`.

For instance, consider this slightly updated example:

```
class A {
    @Deprecated void m() {}

    @Deprecated void n() {
        m();
    }

    @SuppressWarnings("deprecated")
    void r() {
        m();
    }
}
```

Here, the programmer has explicitly suppressed warnings about deprecated calls in `A.r`, so our query should not flag the call to `A.m` any more.

To do so, we first introduce a class for representing all `@SuppressWarnings` annotations where the string `deprecated` occurs among the list of warnings to suppress:

```
class SuppressDeprecationWarningAnnotation extends Annotation {
    SuppressDeprecationWarningAnnotation() {
        this.getType().hasQualifiedName("java.lang", "SuppressWarnings") and
        this.getAValue().(Literal).getLiteral().regexpMatch(".*deprecation.*")
    }
}
```

Here, we use `getAValue()` to retrieve any annotation value: in fact, annotation type `SuppressWarnings` only has a single annotation element, so every `@SuppressWarnings` annotation only has a single annotation value. Then, we ensure that it is a literal, obtain its string value using `getLiteral`, and check whether it contains the string `deprecation` using a regular expression match.

For real-world use, this check would have to be generalized a bit: for example, the OpenJDK Java compiler allows `@SuppressWarnings("all")` annotations to suppress all warnings. We may also want to make sure that `deprecation` is matched as an entire word, and not as part of another word, by changing the regular expression to `".*\\bdeprecation\\b.*"`.

Now we can extend our query to filter out calls in methods carrying a `SuppressDeprecationWarningAnnotation`:

```
import java

// Insert the class definitions from above

from Call call
where call.getCallee() instanceof DeprecatedMethod
    and not call.getCaller() instanceof DeprecatedMethod
    and not call.getCaller().getAnAnnotation() instanceof␣
↪SuppressDeprecationWarningAnnotation
select call, "This call invokes a deprecated method."
```

See this in the query console on LGTM.com. It's fairly common for projects to contain calls to methods that appear to be deprecated.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.8 Javadoc

You can use CodeQL to find errors in Javadoc comments in Java code.

About analyzing Javadoc

To access Javadoc associated with a program element, we use member predicate `getDoc` of class `Element`, which returns a `Documentable`. Class `Documentable`, in turn, offers a member predicate `getJavadoc` to retrieve the Javadoc attached to the element in question, if any.

Javadoc comments are represented by class `Javadoc`, which provides a view of the comment as a tree of `JavadocElement` nodes. Each `JavadocElement` is either a `JavadocTag`, representing a tag, or a `JavadocText`, representing a piece of free-form text.

The most important member predicates of class `Javadoc` are:

- `getAChild` - retrieves a top-level `JavadocElement` node in the tree representation.
- `getVersion` - returns the value of the `@version` tag, if any.
- `getAuthor` - returns the value of the `@author` tag, if any.

For example, the following query finds all classes that have both an `@author` tag and a `@version` tag, and returns this information:

```
import java

from Class c, Javadoc jdoc, string author, string version
where jdoc = c.getDoc().getJavadoc() and
    author = jdoc.getAuthor() and
    version = jdoc.getVersion()
select c, author, version
```

`JavadocElement` defines member predicates `getAChild` and `getParent` to navigate up and down the tree of elements. It also provides a predicate `getTagName` to return the tag's name, and a predicate `getText` to access the text associated with the tag.

We could rewrite the above query to use this API instead of `getAuthor` and `getVersion`:

```
import java

from Class c, Javadoc jdoc, JavadocTag authorTag, JavadocTag versionTag
where jdoc = c.getDoc().getJavadoc() and
    authorTag.getTagName() = "@author" and authorTag.getParent() = jdoc and
    versionTag.getTagName() = "@version" and versionTag.getParent() = jdoc
select c, authorTag.getText(), versionTag.getText()
```

The `JavadocTag` has several subclasses representing specific kinds of Javadoc tags:

- `ParamTag` represents `@param` tags; member predicate `getParamName` returns the name of the parameter being documented.
- `ThrowsTag` represents `@throws` tags; member predicate `getExceptionName` returns the name of the exception being documented.
- `AuthorTag` represents `@author` tags; member predicate `getAuthorName` returns the name of the author.

Example: Finding spurious @param tags

As an example of using the CodeQL Javadoc API, let's write a query that finds `@param` tags that refer to a non-existent parameter.

For example, consider this program:

```
class A {
    /**
    * @param lst a list of strings
    */
    public String get(List<String> list) {
        return list.get(0);
    }
}
```

Here, the `@param` tag on `A.get` misspells the name of parameter `list` as `lst`. Our query should be able to find such cases.

To begin with, we write a query that finds all callables (that is, methods or constructors) and their `@param` tags:

```
import java

from Callable c, ParamTag pt
where c.getDoc().getJavadoc() = pt.getParent()
select c, pt
```

It's now easy to add another conjunct to the `where` clause, restricting the query to `@param` tags that refer to a non-existent parameter: we simply need to require that no parameter of `c` has the name `pt.getParamName()`.

```
import java

from Callable c, ParamTag pt
where c.getDoc().getJavadoc() = pt.getParent() and
    not c.getAParameter().hasName(pt.getParamName())
select pt, "Spurious @param tag."
```

Example: Finding spurious @throws tags

A related, but somewhat more involved, problem is finding `@throws` tags that refer to an exception that the method in question cannot actually throw.

For example, consider this Java program:

```
import java.io.IOException;

class A {
    /**
    * @throws IOException thrown if some IO operation fails
    * @throws RuntimeException thrown if something else goes wrong
    */
    public void foo() {
        // ...
    }
}
```

Notice that the Javadoc comment of `A.foo` documents two thrown exceptions: `IOException` and `RuntimeException`. The former is clearly spurious: `A.foo` doesn't have a `throws IOException` clause, and therefore can't throw this kind of exception. On the other hand, `RuntimeException` is an unchecked exception, so it can be thrown even if there is no explicit `throws` clause listing it. So our query should flag the `@throws` tag for `IOException`, but not the one for `RuntimeException.`

Remember that the CodeQL library represents `@throws` tags using class `ThrowsTag`. This class doesn't provide a member predicate for determining the exception type that is being documented, so we first need to implement our own version. A simple version might look like this:

```
RefType getDocumentedException(ThrowsTag tt) {
    result.hasName(tt.getExceptionName())
}
```

Similarly, `Callable` doesn't come with a member predicate for querying all exceptions that the method or constructor may possibly throw. We can, however, implement this ourselves by using `getAnException` to find all `throws` clauses of the callable, and then use `getType` to resolve the corresponding exception types:

```
predicate mayThrow(Callable c, RefType exn) {
    exn.getASupertype*() = c.getAnException().getType()
}
```

Note the use of `getASupertype*` to find both exceptions declared in a `throws` clause and their subtypes. For instance, if a method has a `throws IOException` clause, it may throw `MalformedURLException`, which is a subtype of `IOException`.

Now we can write a query for finding all callables `c` and `@throws` tags `tt` such that:

- `tt` belongs to a Javadoc comment attached to `c`.
- `c` can't throw the exception documented by `tt`.

```
import java

// Insert the definitions from above

from Callable c, ThrowsTag tt, RefType exn
where c.getDoc().getJavadoc() = tt.getParent+() and
    exn = getDocumentedException(tt) and
    not mayThrow(c, exn)
select tt, "Spurious @throws tag."
```

See this in the query console on LGTM.com. This finds several results in the LGTM.com demo projects.

Improvements

Currently, there are two problems with this query:

1. `getDocumentedException` is too liberal: it will return *any* reference type with the right name, even if it's in a different package and not actually visible in the current compilation unit.
2. `mayThrow` is too restrictive: it doesn't account for unchecked exceptions, which do not need to be declared.

To see why the former is a problem, consider this program:

```
class IOException extends Exception {}

class B {
    /** @throws IOException an IO exception */
    void bar() throws IOException {}
}
```

This program defines its own class `IOException`, which is unrelated to the class `java.io.IOException` in the standard library: they are in different packages. Our `getDocumentedException` predicate doesn't check packages, however, so it will consider the `@throws` clause to refer to both `IOException` classes, and thus flag the `@param` tag as spurious, since `B.bar` can't actually throw `java.io.IOException`.

As an example of the second problem, method `A.foo` from our previous example was annotated with a `@throws RuntimeException` tag. Our current version of `mayThrow`, however, would think that `A.foo` can't throw a `RuntimeException`, and thus flag the tag as spurious.

We can make `mayThrow` less restrictive by introducing a new class to represent unchecked exceptions, which are just the subtypes of `java.lang.RuntimeException` and `java.lang.Error`:

```
class UncheckedException extends RefType {
    UncheckedException() {
        this.getASupertype*().hasQualifiedName("java.lang", "RuntimeException") or
        this.getASupertype*().hasQualifiedName("java.lang", "Error")
    }
}
```

Now we incorporate this new class into our `mayThrow` predicate:

```
predicate mayThrow(Callable c, RefType exn) {
    exn instanceof UncheckedException or
    exn.getASupertype*() = c.getAnException().getType()
}
```

Fixing `getDocumentedException` is more complicated, but we can easily cover three common cases:

1. The `@throws` tag specifies the fully qualified name of the exception.
2. The `@throws` tag refers to a type in the same package.
3. The `@throws` tag refers to a type that is imported by the current compilation unit.

The first case can be covered by changing `getDocumentedException` to use the qualified name of the `@throws` tag. To handle the second and the third case, we can introduce a new predicate `visibleIn` that checks whether a reference type is visible in a compilation unit, either by virtue of belonging to the same package or by being explicitly imported. We then rewrite `getDocumentedException` as:

```
predicate visibleIn(CompilationUnit cu, RefType tp) {
    cu.getPackage() = tp.getPackage()
    or
    exists(ImportType it | it.getCompilationUnit() = cu | it.getImportedType() = tp)
}

RefType getDocumentedException(ThrowsTag tt) {
    result.getQualifiedName() = tt.getExceptionName()
    or
    (result.hasName(tt.getExceptionName()) and visibleIn(tt.getFile(), result))
}
```

See this in the query console on LGTM.com. This finds many fewer, more interesting results in the LGTM.com demo projects.

Currently, `visibleIn` only considers single-type imports, but you could extend it with support for other kinds of imports.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.9 Working with source locations

You can use the location of entities within Java code to look for potential errors. Locations allow you to deduce the presence, or absence, of white space which, in some cases, may indicate a problem.

About source locations

Java offers a rich set of operators with complex precedence rules, which are sometimes confusing to developers. For instance, the class `ByteBufferCache` in the OpenJDK Java compiler (which is a member class of `com.sun.tools.javac.util.BaseFileManager`) contains this code for allocating a buffer:

```
ByteBuffer.allocate(capacity + capacity>>1)
```

Presumably, the author meant to allocate a buffer that is 1.5 times the size indicated by the variable `capacity`. In fact, however, operator + binds tighter than operator >>, so the expression `capacity + capacity>>1` is parsed as `(capacity + capacity)>>1`, which equals `capacity` (unless there is an arithmetic overflow).

Note that the source layout gives a fairly clear indication of the intended meaning: there is more white space around + than around >>, suggesting that the latter is meant to bind more tightly.

We're going to develop a query that finds this kind of suspicious nesting, where the operator of the inner expression has more white space around it than the operator of the outer expression. This pattern may not necessarily indicate a bug, but at the very least it makes the code hard to read and prone to misinterpretation.

White space is not directly represented in the CodeQL database, but we can deduce its presence from the location information associated with program elements and AST nodes. So, before we write our query, we need an understanding of source location management in the standard library for Java.

Location API

For every entity that has a representation in Java source code (including, in particular, program elements and AST nodes), the standard CodeQL library provides these predicates for accessing source location information:

- `getLocation` returns a `Location` object describing the start and end position of the entity.
- `getFile` returns a `File` object representing the file containing the entity.
- `getTotalNumberOfLines` returns the number of lines the source code of the entity spans.
- `getNumberOfCommentLines` returns the number of comment lines.
- `getNumberOfLinesOfCode` returns the number of non-comment lines.

For example, let's assume this Java class is defined in the compilation unit `SayHello.java`:

```
package pkg;

class SayHello {
    public static void main(String[] args) {
        System.out.println(
            // Display personalized message
            "Hello, " + args[0];
        );
    }
}
```

Invoking `getFile` on the expression statement in the body of `main` returns a `File` object representing the file `SayHello.java`. The statement spans four lines in total (`getTotalNumberOfLines`), of which one is a comment line (`getNumberOfCommentLines`), while three lines contain code (`getNumberOfLinesOfCode`).

Class `Location` defines member predicates `getStartLine`, `getEndLine`, `getStartColumn` and `getEndColumn` to retrieve the line and column number an entity starts and ends at, respectively. Both lines and columns are counted starting from 1 (not 0), and the end position is inclusive, that is, it is the position of the last character belonging to the source code of the entity.

In our example, the expression statement starts at line 5, column 3 (the first two characters on the line are tabs, which each count as one character), and it ends at line 8, column 4.

Class `File` defines these member predicates:

- `getAbsolutePath` returns the fully qualified name of the file.
- `getRelativePath` returns the path of the file relative to the base directory of the source code.
- `getExtension` returns the extension of the file.
- `getStem` returns the base name of the file, without its extension.

In our example, assume file `A.java` is located in directory `/home/testuser/code/pkg`, where `/home/testuser/code` is the base directory of the program being analyzed. Then, a `File` object for `A.java` returns:

- `getAbsolutePath` is `/home/testuser/code/pkg/A.java`.
- `getRelativePath` is `pkg/A.java`.
- `getExtension` is `java`.
- `getStem` is `A`.

Determining white space around an operator

Let's start by considering how to write a predicate that computes the total amount of white space surrounding the operator of a given binary expression. If `rcol` is the start column of the expression's right operand and `lcol` is the end column of its left operand, then `rcol - (lcol+1)` gives us the total number of characters in between the two operands (note that we have to use `lcol+1` instead of `lcol` because end positions are inclusive).

This number includes the length of the operator itself, which we need to subtract out. For this, we can use predicate `getOp`, which returns the operator string, surrounded by one white space on either side. Overall, the expression for computing the amount of white space around the operator of a binary expression `expr` is:

```
rcol - (lcol+1) - (expr.getOp().length()-2)
```

Clearly, however, this only works if the entire expression is on a single line, which we can check using predicate `getTotalNumberOfLines` introduced above. We are now in a position to define our predicate for computing white space around operators:

```
int operatorWS(BinaryExpr expr) {
    exists(int lcol, int rcol |
        expr.getNumberOfLinesOfCode() = 1 and
        lcol = expr.getLeftOperand().getLocation().getEndColumn() and
        rcol = expr.getRightOperand().getLocation().getStartColumn() and
        result = rcol - (lcol+1) - (expr.getOp().length()-2)
    )
}
```

Notice that we use an `exists` to introduce our temporary variables `lcol` and `rcol`. You could write the predicate without them by just inlining `lcol` and `rcol` into their use, at some cost in readability.

Find suspicious nesting

Here's a first version of our query:

```
import java

// Insert predicate defined above

from BinaryExpr outer, BinaryExpr inner,
    int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

See this in the query console on LGTM.com. This query is likely to find results on most projects.

The first conjunct of the `where` clause restricts `inner` to be an operand of `outer`, the second conjunct binds `wsinner` and `wsouter`, while the last conjunct selects the suspicious cases.

At first, we might be tempted to write `inner = outer.getAnOperand()` in the first conjunct. This, however, wouldn't be quite correct: `getAnOperand` strips off any surrounding parentheses from its result, which is often useful, but not what we want here: if there are parentheses around the inner expression, then the programmer probably knew what they were doing, and the query should not flag this expression.

Improving the query

If we run this initial query, we might notice some false positives arising from asymmetric white space. For instance, the following expression is flagged as suspicious, although it is unlikely to cause confusion in practice:

```
i< start + 100
```

Note that our predicate `operatorWS` computes the **total** amount of white space around the operator, which, in this case, is one for the < and two for the +. Ideally, we would like to exclude cases where the amount of white space before and after the operator are not the same. Currently, CodeQL databases don't record enough information to figure this out, but as an approximation we could require that the total number of white space characters is even:

```
import java

// Insert predicate definition from above

from BinaryExpr outer, BinaryExpr inner,
    int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

See this in the query console on LGTM.com. Any results will be refined by our changes to the query.

Another source of false positives are associative operators: in an expression of the form `x + y+z`, the first plus is syntactically nested inside the second, since + in Java associates to the left; hence the expression is flagged as suspicious. But since + is associative to begin with, it does not matter which way around the operators are nested, so this is a false positive. To exclude these cases, let us define a new class identifying binary expressions with an associative operator:

```
class AssociativeOperator extends BinaryExpr {
    AssociativeOperator() {
        this instanceof AddExpr or
        this instanceof MulExpr or
        this instanceof BitwiseExpr or
        this instanceof AndLogicalExpr or
        this instanceof OrLogicalExpr
    }
}
```

Now we can extend our query to discard results where the outer and the inner expression both have the same, associative operator:

```
import java

// Insert predicate and class definitions from above

from BinaryExpr inner, BinaryExpr outer, int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    not (inner.getOp() = outer.getOp() and outer instanceof AssociativeOperator) and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

See this in the query console on LGTM.com.

Notice that we again use `getOp`, this time to determine whether two binary expressions have the same operator. Running our improved query now finds the Java standard library bug described in the Overview. It also flags up the following suspicious code in Hadoop HBase:

```
KEY_SLAVE = tmp[ i+1 % 2 ];
```

Whitespace suggests that the programmer meant to toggle `i` between zero and one, but in fact the expression is parsed as `i + (1%2)`, which is the same as `i + 1`, so `i` is simply incremented.

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.4.10 Abstract syntax tree classes for working with Java programs

CodeQL has a large selection of classes for representing the abstract syntax tree of Java programs.

The abstract syntax tree (AST) represents the syntactic structure of a program. Nodes on the AST represent elements such as statements and expressions.

Statement classes

This table lists all subclasses of Stmt.

Statement syntax	CodeQL class	Superclasses	Remarks
`;`	EmptyStmt		
Expr `;`	ExprStmt		
`{` Stmt `...` `}`	BlockStmt		
`if (` Expr `)` Stmt `else` Stmt	IfStmt	ConditionalStmt	
`if (` Expr `)` Stmt			
`while (` Expr `)` Stmt	WhileStmt	ConditionalStmt, LoopStmt	
`do` Stmt `while (` Expr `)`	DoStmt	ConditionalStmt, LoopStmt	
`for (` Expr `;` Expr `;` Expr `)` Stmt	ForStmt	ConditionalStmt, LoopStmt	
`for (` VarAccess `:` Expr `)` Stmt	EnhancedForStmt	LoopStmt	
`switch (` Expr `) {` SwitchCase `...` `}`	SwitchStmt		
`try {` Stmt `...` `} finally {` Stmt `...` `}`	TryStmt		
`return` Expr `;`	ReturnStmt		
`return ;`			
`throw` Expr `;`	ThrowStmt		
`break ;`	BreakStmt	JumpStmt	
`break` label `;`			
`continue ;`	ContinueStmt	JumpStmt	
`continue` label `;`			
label `:` Stmt	LabeledStmt		
`synchronized (` Expr `)` Stmt	SynchronizedStmt		
`assert` Expr `:` Expr `;`	AssertStmt		
`assert` Expr `;`			
TypeAccess `name ;`	LocalVariableDeclStmt		
`class name {` Member `...` `}` `;`	LocalClassDeclStmt		
`this (` Expr `, ...` `) ;`	ThisConstructorInvocationStmt		
`super (` Expr `, ...` `) ;`	SuperConstructorInvocationStmt		
`catch (` TypeAccess `name` `) {` Stmt `...` `}`	CatchClause		can only occur as child of a TryStmt
`case` Literal `:` Stmt `...`	ConstCase		can only occur as child of a SwitchStmt
`default :` Stmt `...`	DefaultCase		can only occur as child of a SwitchStmt

Expression classes

There are many expression classes, so we present them by category. All classes in this section are subclasses of Expr.

Literals

All classes in this subsection are subclasses of Literal.

Expression syntax example	CodeQL class
`true`	BooleanLiteral
`23`	IntegerLiteral
`23l`	LongLiteral
`4.2f`	FloatingPointLiteral
`4.2`	DoubleLiteral
`'a'`	CharacterLiteral
`"Hello"`	StringLiteral
`null`	NullLiteral

Unary expressions

All classes in this subsection are subclasses of UnaryExpr.

Expression syntax	CodeQL class	Superclasses	Remarks
Expr++	PostIncExpr	UnaryAssignExpr	
Expr--	PostDecExpr	UnaryAssignExpr	
++Expr	PreIncExpr	UnaryAssignExpr	
--Expr	PreDecExpr	UnaryAssignExpr	
~Expr	BitNotExpr	BitwiseExpr	see below for other subclasses of BitwiseExpr
-Expr	MinusExpr		
+Expr	PlusExpr		
!Expr	LogNotExpr	LogicExpr	see below for other subclasses of LogicExpr

Binary expressions

All classes in this subsection are subclasses of BinaryExpr.

Expression syntax	CodeQL class	Superclasses
Expr * Expr	MulExpr	
Expr / Expr	DivExpr	
Expr % Expr	RemExpr	
Expr + Expr	AddExpr	
Expr - Expr	SubExpr	
Expr << Expr	LShiftExpr	
Expr >> Expr	RShiftExpr	
Expr >>> Expr	URShiftExpr	
Expr && Expr	AndLogicalExpr	LogicExpr
Expr || Expr	OrLogicalExpr	LogicExpr
Expr < Expr	LTExpr	ComparisonExpr
Expr > Expr	GTExpr	ComparisonExpr
Expr <= Expr	LEExpr	ComparisonExpr
Expr >= Expr	GEExpr	ComparisonExpr
Expr == Expr	EQExpr	EqualityTest
Expr != Expr	NEExpr	EqualityTest
Expr & Expr	AndBitwiseExpr	BitwiseExpr
Expr | Expr	OrBitwiseExpr	BitwiseExpr
Expr ^ Expr	XorBitwiseExpr	BitwiseExpr

Assignment expressions

All classes in this table are subclasses of Assignment.

Expression syntax	CodeQL class	Superclasses
Expr = Expr	AssignExpr	
Expr += Expr	AssignAddExpr	AssignOp
Expr -= Expr	AssignSubExpr	AssignOp
Expr *= Expr	AssignMulExpr	AssignOp
Expr /= Expr	AssignDivExpr	AssignOp
Expr %= Expr	AssignRemExpr	AssignOp
Expr &= Expr	AssignAndExpr	AssignOp
Expr |= Expr	AssignOrExpr	AssignOp
Expr ^= Expr	AssignXorExpr	AssignOp
Expr <<= Expr	AssignLShiftExpr	AssignOp
Expr >>= Expr	AssignRShiftExpr	AssignOp
Expr >>>= Expr	AssignURShiftExpr	AssignOp

Accesses

Expression syntax examples	CodeQL class
`this`	ThisAccess
`Outer.this`	
`super`	SuperAccess
`Outer.super`	
`x`	VarAccess
`e.f`	
`a[i]`	ArrayAccess
`f(...)`	MethodAccess
`e.m(...)`	
`String`	TypeAccess
`java.lang.String`	
`? extends Number`	WildcardTypeAccess
`? super Double`	

A VarAccess that refers to a field is a FieldAccess.

Miscellaneous

Expression syntax examples	CodeQL class	Remarks
`(int) f`	CastExpr	
`o instanceof String`	InstanceOfExpr	
Expr `?` Expr `:` Expr	ConditionalExpr	
`String. class`	TypeLiteral	
`new A()`	ClassInstanceExpr	
`new String[3][2]`	ArrayCreationExpr	
`new int[] { 23, 42 }`		
`{ 23, 42 }`	ArrayInit	can only appear as an initializer or as a child of an ArrayCreationExpr
`@Annot(key=val)`	Annotation	

Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"
- *Basic query for Java code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for Java*: When analyzing Java code, you can use the large collection of classes in the CodeQL library for Java.

- *Analyzing data flow in Java*: You can use CodeQL to track the flow of data through a Java program to its use.
- *Java types*: You can use CodeQL to find out information about data types used in Java code. This allows you to write queries to identify specific type-related issues.
- *Overflow-prone comparisons in Java*: You can use CodeQL to check for comparisons in Java code where one side of the comparison is prone to overflow.
- *Navigating the call graph*: CodeQL has classes for identifying code that calls other code, and code that can be called from elsewhere. This allows you to find, for example, methods that are never used.
- *Annotations in Java*: CodeQL databases of Java projects contain information about all annotations attached to program elements.
- *Javadoc*: You can use CodeQL to find errors in Javadoc comments in Java code.
- *Working with source locations*: You can use the location of entities within Java code to look for potential errors. Locations allow you to deduce the presence, or absence, of white space which, in some cases, may indicate a problem.
- *Abstract syntax tree classes for working with Java programs*: CodeQL has a large selection of classes for representing the abstract syntax tree of Java programs.

5.5 CodeQL for JavaScript

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from JavaScript codebases.

5.5.1 Basic query for JavaScript code

Learn to write and run a simple CodeQL query using LGTM.

About the query

In JavaScript, any expression can be turned into an expression statement. While this is sometimes convenient, it can be dangerous. For example, imagine a programmer wants to assign a new value to a variable `x` by means of an assignment `x = 42`. However, they accidentally type two equals signs, producing the comparison statement `x == 42`. This is valid JavaScript, so no error is generated. The statement simply compares `x` to `42`, and then discards the result of the comparison.

The query you will run finds instances of this problem. The query searches for expressions `e` that are pure—that is, their evaluation does not lead to any side effects—but appear as an expression statement.

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

Note

Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **JavaScript** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import javascript

from Expr e
where e.isPure() and
  e.getParent() instanceof ExprStmt
select e, "This expression has no effect."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

Note

Your query is always run against the most recently analyzed commit to the selected project.

The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `e` and is linked to the location in the source code of the project where `e` occurs. The second column is the alert message.

Example query results

Note

An ellipsis (...) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click one of the links in the `e` column to view the expression in the code viewer.

The matching statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import javascript`	Imports the standard CodeQL libraries for JavaScript.	Every query begins with one or more `import` statements.
`from Expr e`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	`e` is declared as a variable that ranges over expressions.
`where e.isPure() and e.getParent() instanceof ExprStmt`	Defines a condition on the variables.	`e.isPure()`: The expression is side-effect-free. `e.getParent() instanceof ExprStmt`: The parent of the expression is an expression statement.
`select e, "This expression has no effect."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Report the expression with a string that explains the problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find `use strict` directives. These are interpreted specially by modern browsers with strict mode support and so these expressions *do* have an effect.

To remove directives from the results:

1. Extend the `where` clause to include the following extra condition:

   ```
   and not e.getParent() instanceof Directive
   ```

   The `where` clause is now:

   ```
   where e.isPure() and
     e.getParent() instanceof ExprStmt and
     not e.getParent() instanceof Directive
   ```

2. Click **Run**.

   There are now fewer results as `use strict` directives are no longer reported.

The improved query finds several results on the example project including this result:

```
point.bias == -1;
```

As written, this statement compares `point.bias` against `-1` and then discards the result. Most likely, it was instead meant to be an assignment `point.bias = -1`.

Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"

5.5.2 CodeQL library for JavaScript

When you're analyzing a JavaScript program, you can make use of the large collection of classes in the CodeQL library for JavaScript.

Overview

There is an extensive CodeQL library for analyzing JavaScript code. The classes in this library present the data from a CodeQL database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `javascript.qll` imports most other standard library modules, so you can include the complete library by beginning your query with:

```
import javascript
```

The rest of this tutorial briefly summarizes the most important classes and predicates provided by this library, including references to the detailed API documentation where applicable.

Introducing the library

The CodeQL library for JavaScript presents information about JavaScript source code at different levels:

- **Textual** — classes that represent source code as unstructured text files
- **Lexical** — classes that represent source code as a series of tokens and comments
- **Syntactic** — classes that represent source code as an abstract syntax tree
- **Name binding** — classes that represent scopes and variables
- **Control flow** — classes that represent the flow of control during execution
- **Data flow** — classes that you can use to reason about data flow in JavaScript source code
- **Type inference** — classes that you can use to approximate types for JavaScript expressions and variables
- **Call graph** — classes that represent the caller-callee relationship between functions
- **Inter-procedural data flow** — classes that you can use to define inter-procedural data flow and taint tracking analyses

- **Frameworks** — classes that represent source code entities that have a special meaning to JavaScript tools and frameworks

Note that representations above the textual level (for example the lexical representation or the flow graphs) are only available for JavaScript code that does not contain fatal syntax errors. For code with such errors, the only information available is at the textual level, as well as information about the errors themselves.

Additionally, there is library support for working with HTML documents, JSON, and YAML data, JSDoc comments, and regular expressions.

Textual level

At its most basic level, a JavaScript code base can simply be viewed as a collection of files organized into folders, where each file is composed of zero or more lines of text.

Note that the textual content of a program is not included in the CodeQL database unless you specifically request it during extraction. In particular, databases on LGTM (also known as "snapshots") do not normally include textual information.

Files and folders

In the CodeQL libraries, files are represented as entities of class File, and folders as entities of class Folder, both of which are subclasses of class Container.

Class Container provides the following member predicates:

- `Container.getParentContainer()` returns the parent folder of the file or folder.
- `Container.getAFile()` returns a file within the folder.
- `Container.getAFolder()` returns a folder nested within the folder.

Note that while `getAFile` and `getAFolder` are declared on class Container, they currently only have results for Folders.

Both files and folders have paths, which can be accessed by the predicate `Container.getAbsolutePath()`. For example, if `f` represents a file with the path `/home/user/project/src/index.js`, then `f.getAbsolutePath()` evaluates to the string `"/home/user/project/src/index.js"`, while `f.getParentContainer().getAbsolutePath()` returns `"/home/user/project/src"`.

These paths are absolute file system paths. If you want to obtain the path of a file relative to the source location in the CodeQL database, use `Container.getRelativePath()` instead. Note, however, that a database may contain files that are not located underneath the source location; for such files, `getRelativePath()` will not return anything.

The following member predicates of class Container provide more information about the name of a file or folder:

- `Container.getBaseName()` returns the base name of a file or folder, not including its parent folder, but including its extension. In the above example, `f.getBaseName()` would return the string `"index.js"`.
- `Container.getStem()` is similar to `Container.getBaseName()`, but it does *not* include the file extension; so `f.getStem()` returns `"index"`.
- `Container.getExtension()` returns the file extension, not including the dot; so `f.getExtension()` returns `"js"`.

For example, the following query computes, for each folder, the number of JavaScript files (that is, files with extension `js`) contained in the folder:

```
import javascript

from Folder d
select d.getRelativePath(), count(File f | f = d.getAFile() and f.getExtension() = "js")
```

See this in the query console on LGTM.com. When you run the query on most projects, the results include folders that contain files with a `js` extension and folders that don't.

Locations

Most entities in a CodeQL database have an associated source location. Locations are identified by five pieces of information: a file, a start line, a start column, an end line, and an end column. Line and column counts are 1-based (so the first character of a file is at line 1, column 1), and the end position is inclusive.

All entities associated with a source location belong to the class Locatable. The location itself is modeled by the class Location and can be accessed through the member predicate `Locatable.getLocation()`. The Location class provides the following member predicates:

- `Location.getFile()`, `Location.getStartLine()`, `Location.getStartColumn()`, `Location.getEndLine()`, `Location.getEndColumn()` return detailed information about the location.
- `Location.getNumLines()` returns the number of (whole or partial) lines covered by the location.
- `Location.startsBefore(Location)` and `Location.endsAfter(Location)` determine whether one location starts before or ends after another location.
- `Location.contains(Location)` indicates whether one location completely contains another location; `l1.contains(l2)` holds if, and only if, `l1.startsBefore(l2)` and `l1.endsAfter(l2)`.

Lines

Lines of text in files are represented by the class Line. This class offers the following member predicates:

- `Line.getText()` returns the text of the line, excluding any terminating newline characters.
- `Line.getTerminator()` returns the terminator character(s) of the line. The last line in a file may not have any terminator characters, in which case this predicate does not return anything; otherwise it returns either the two-character string `"\r\n"` (carriage-return followed by newline), or one of the one-character strings `"\n"` (newline), `"\r"` (carriage-return), `"\u2028"` (Unicode character LINE SEPARATOR), `"\u2029"` (Unicode character PARAGRAPH SEPARATOR).

Note that, as mentioned above, the textual representation of the program is not included in the CodeQL database by default.

Lexical level

A slightly more structured view of a JavaScript program is provided by the classes Token and Comment, which represent tokens and comments, respectively.

Tokens

The most important member predicates of class Token are as follows:

- `Token.getValue()` returns the source text of the token.
- `Token.getIndex()` returns the index of the token within its enclosing script.
- `Token.getNextToken()` and `Token.getPreviousToken()` navigate between tokens.

The Token class has nine subclasses, each representing a particular kind of token:

- EOFToken: a marker token representing the end of a script
- NullLiteralToken, BooleanLiteralToken, NumericLiteralToken, StringLiteralToken and RegularExpressionToken: different kinds of literals
- IdentifierToken and KeywordToken: identifiers and keywords (including reserved words) respectively
- PunctuatorToken: operators and other punctuation symbols

As an example of a query operating entirely on the lexical level, consider the following query, which finds consecutive comma tokens arising from an omitted element in an array expression:

```
import javascript

class CommaToken extends PunctuatorToken {
    CommaToken() {
        getValue() = ","
    }
}

from CommaToken comma
where comma.getNextToken() instanceof CommaToken
select comma, "Omitted array elements are bad style."
```

See this in the query console on LGTM.com. If the query returns no results, this pattern isn't used in the projects that you analyzed.

You can use predicate `Locatable.getFirstToken()` and `Locatable.getLastToken()` to access the first and last token (if any) belonging to an element with a source location.

Comments

The class Comment and its subclasses represent the different kinds of comments that can occur in JavaScript programs:

- Comment: any comment
  - LineComment: a single-line comment terminated by an end-of-line character
    - SlashSlashComment: a plain JavaScript single-line comment starting with `//`
    - HtmlLineComment: a (non-standard) HTML comment
      - HtmlCommentStart: an HTML comment starting with `<!--`
      - HtmlCommentEnd: an HTML comment ending with `-->`
  - BlockComment: a block comment potentially spanning multiple lines
    - SlashStarComment: a plain JavaScript block comment surrounded with `/*...*/`

        * DocComment: a documentation block comment surrounded with `/**...*/`

The most important member predicates are as follows:

- `Comment.getText()` returns the source text of the comment, not including delimiters.
- `Comment.getLine(i)` returns the `i`th line of text within the comment (0-based).
- `Comment.getNumLines()` returns the number of lines in the comment.
- `Comment.getNextToken()` returns the token immediately following a comment. Note that such a token always exists: if a comment appears at the end of a file, its following token is an EOFToken.

As an example of a query using only lexical information, consider the following query for finding HTML comments, which are not a standard ECMAScript feature and should be avoided:

```
import javascript

from HtmlLineComment c
select c, "Do not use HTML comments."
```

See this in the query console on LGTM.com. When we ran this query on the *mozilla/pdf.js* project in LGTM.com, we found three HTML comments.

Syntactic level

The majority of classes in the JavaScript library is concerned with representing a JavaScript program as a collection of abstract syntax trees (ASTs).

The class ASTNode contains all entities representing nodes in the abstract syntax trees and defines generic tree traversal predicates:

- `ASTNode.getChild(i)`: returns the `i`th child of this AST node.
- `ASTNode.getAChild()`: returns any child of this AST node.
- `ASTNode.getParent()`: returns the parent node of this AST node, if any.

  Note

  These predicates should only be used to perform generic AST traversal. To access children of specific AST node types, the specialized predicates introduced below should be used instead. In particular, queries should not rely on the numeric indices of child nodes relative to their parent nodes: these are considered an implementation detail that may change between versions of the library.

Top-levels

From a syntactic point of view, each JavaScript program is composed of one or more top-level code blocks (or *top-levels* for short), which are blocks of JavaScript code that do not belong to a larger code block. Top-levels are represented by the class TopLevel and its subclasses:

- TopLevel
  - Script: a stand-alone file or HTML `<script>` element
    * ExternalScript: a stand-alone JavaScript file
    * InlineScript: code embedded inline in an HTML `<script>` tag
  - CodeInAttribute: a code block originating from an HTML attribute value

        * EventHandlerCode: code from an event handler attribute such as `onload`
        * JavaScriptURL: code from a URL with the `javascript:` scheme
    - Externs: a JavaScript file containing externs definitions

Every TopLevel class is contained in a File class, but a single File may contain more than one TopLevel. To go from a `TopLevel tl` to its File, use `tl.getFile()`; conversely, for a `File f`, predicate `f.getATopLevel()` returns a top-level contained in `f`. For every AST node, predicate `ASTNode.getTopLevel()` can be used to find the top-level it belongs to.

The TopLevel class additionally provides the following member predicates:

- `TopLevel.getNumberOfLines()` returns the total number of lines (including code, comments and whitespace) in the top-level.
- `TopLevel.getNumberOfLinesOfCode()` returns the number of lines of code, that is, lines that contain at least one token.
- `TopLevel.getNumberOfLinesOfComments()` returns the number of lines containing or belonging to a comment.
- `TopLevel.isMinified()` determines whether the top-level contains minified code, using a heuristic based on the average number of statements per line.

  Note

  By default, LGTM filters out alerts in minified top-levels, since they are often hard to interpret. When writing your own queries in the LGTM query console, this filtering is *not* done automatically, so you may want to explicitly add a condition of the form `and not e.getTopLevel().isMinified()` or similar to your query to exclude results in minified code.

Statements and expressions

The most important subclasses of ASTNode besides TopLevel are Stmt and Expr, which, together with their subclasses, represent statements and expressions, respectively. This section briefly discusses some of the more important classes and predicates. For a full reference of all the subclasses of Stmt and Expr and their API, see Stmt.qll and Expr.qll.

- Stmt: use `Stmt.getContainer()` to access the innermost function or top-level in which the statement is contained.
    - ControlStmt: a statement that controls the execution of other statements, that is, a conditional, loop, `try` or `with` statement; use `ControlStmt.getAControlledStmt()` to access the statements that it controls.
        * IfStmt: an `if` statement; use `IfStmt.getCondition()`, `IfStmt.getThen()` and `IfStmt.getElse()` to access its condition expression, "then" branch and "else" branch, respectively.
        * LoopStmt: a loop; use `Loop.getBody()` and `Loop.getTest()` to access its body and its test expression, respectively.
            · WhileStmt, DoWhileStmt: a "while" or "do-while" loop, respectively.
            · ForStmt: a "for" statement; use `ForStmt.getInit()` and `ForStmt.getUpdate()` to access the init and update expressions, respectively.
            · EnhancedForLoop: a "for-in" or "for-of" loop; use `EnhancedForLoop.getIterator()` to access the loop iterator (which may be a expression or variable declaration), and `EnhancedForLoop.getIterationDomain()` to access the expression being iterated over.
            · ForInStmt, ForOfStmt: a "for-in" or "for-of" loop, respectively.

        * WithStmt: a "with" statement; use `WithStmt.getExpr()` and `WithStmt.getBody()` to access the controlling expression and the body of the with statement, respectively.
        * SwitchStmt: a switch statement; use `SwitchStmt.getExpr()` to access the expression on which the statement switches; use `SwitchStmt.getCase(int)` and `SwitchStmt.getACase()` to access individual switch cases; each case is modeled by an entity of class Case, whose member predicates `Case.getExpr()` and `Case.getBodyStmt(int)` provide access to the expression checked by the switch case (which is undefined for `default`), and its body.
        * TryStmt: a "try" statement; use `TryStmt.getBody()`, `TryStmt.getCatchClause()` and `TryStmt.getFinally` to access its body, "catch" clause and "finally" block, respectively.
    - BlockStmt: a block of statements; use `BlockStmt.getStmt(int)` to access the individual statements in the block.
    - ExprStmt: an expression statement; use `ExprStmt.getExpr()` to access the expression itself.
    - JumpStmt: a statement that disrupts structured control flow, that is, one of `break`, `continue`, `return` and `throw`; use predicate `JumpStmt.getTarget()` to determine the target of the jump, which is either a statement or (for `return` and uncaught `throw` statements) the enclosing function.
        * BreakStmt: a "break" statement; use `BreakStmt.getLabel()` to access its (optional) target label.
        * ContinueStmt: a "continue" statement; use `ContinueStmt.getLabel()` to access its (optional) target label.
        * ReturnStmt: a "return" statement; use `ReturnStmt.getExpr()` to access its (optional) result expression.
        * ThrowStmt: a "throw" statement; use `ThrowStmt.getExpr()` to access its thrown expression.
    - FunctionDeclStmt: a function declaration statement; see below for available member predicates.
    - ClassDeclStmt: a class declaration statement; see below for available member predicates.
    - DeclStmt: a declaration statement containing one or more declarators which can be accessed by predicate `DeclStmt.getDeclarator(int)`.
        * VarDeclStmt, ConstDeclStmt, LetStmt: a `var`, `const` or `let` declaration statement.
- Expr: use `Expr.getEnclosingStmt()` to obtain the innermost statement to which this expression belongs; `Expr.isPure()` determines whether the expression is side-effect-free.
    - Identifier: an identifier; use `Identifier.getName()` to obtain its name.
    - Literal: a literal value; use `Literal.getValue()` to obtain a string representation of its value, and `Literal.getRawValue()` to obtain its raw source text (including surrounding quotes for string literals).
        * NullLiteral, BooleanLiteral, NumberLiteral, StringLiteral, RegExpLiteral: different kinds of literals.
    - ThisExpr: a "this" expression.
    - SuperExpr: a "super" expression.
    - ArrayExpr: an array expression; use `ArrayExpr.getElement(i)` to obtain the `i`th element expression, and `ArrayExpr.elementIsOmitted(i)` to check whether the `i`th element is omitted.
    - ObjectExpr: an object expression; use `ObjectExpr.getProperty(i)` to obtain the `i`th property in the object expression; properties are modeled by class Property, which is described in more detail below.
    - FunctionExpr: a function expression; see below for available member predicates.
    - ArrowFunctionExpr: an ECMAScript 2015-style arrow function expression; see below for available member predicates.
    - ClassExpr: a class expression; see below for available member predicates.

- ParExpr: a parenthesized expression; use `ParExpr.getExpression()` to obtain the operand expression; for any expression, `Expr.stripParens()` can be used to recursively strip off any parentheses
- SeqExpr: a sequence of two or more expressions connected by the comma operator; use `SeqExpr.getOperand(i)` to obtain the `i`th sub-expression.
- ConditionalExpr: a ternary conditional expression; member predicates `ConditionalExpr.getCondition()`, `ConditionalExpr.getConsequent()` and `ConditionalExpr.getAlternate()` provide access to the condition expression, the "then" expression and the "else" expression, respectively.
- InvokeExpr: a function call or a "new" expression; use `InvokeExpr.getCallee()` to obtain the expression specifying the function to be called, and `InvokeExpr.getArgument(i)` to obtain the `i`th argument expression.
  - CallExpr: a function call.
  - NewExpr: a "new" expression.
  - MethodCallExpr: a function call whose callee expression is a property access; use `MethodCallExpr.getReceiver` to access the receiver expression of the method call, and `MethodCallExpr.getMethodName()` to get the method name (if it can be determined statically).
- PropAccess: a property access, that is, either a "dot" expression of the form `e.f` or an index expression of the form `e[p]`; use `PropAccess.getBase()` to obtain the base expression on which the property is accessed (`e` in the example), and `PropAccess.getPropertyName()` to determine the name of the accessed property; if the name cannot be statically determined, `getPropertyName()` does not return any value.
  - DotExpr: a "dot" expression.
  - IndexExpr: an index expression (also known as computed property access).
- UnaryExpr: a unary expression; use `UnaryExpr.getOperand()` to obtain the operand expression.
  - NegExpr ("-"), PlusExpr ("+"), LogNotExpr ("!"), BitNotExpr ("~"), TypeofExpr, VoidExpr, DeleteExpr, SpreadElement ("..."): various types of unary expressions.
- BinaryExpr: a binary expression; use `BinaryExpr.getLeftOperand()` and `BinaryExpr.getRightOperand()` to access the operand expressions.
  - Comparison: any comparison expression.
    - EqualityTest: any equality or inequality test.
    - EqExpr ("=="), NEqExpr ("!="): non-strict equality and inequality tests.
    - StrictEqExpr ("==="), StrictNEqExpr ("!=="): strict equality and inequality tests.
    - LTExpr ("<"), LEExpr ("<="), GTExpr (">"), GEExpr (">="): numeric comparisons.
  - LShiftExpr ("<<"), RShiftExpr (">>"), URShiftExpr (">>>"): shift operators.
  - AddExpr ("+"), SubExpr ("-"), MulExpr ("*"), DivExpr ("/"), ModExpr ("%"), ExpExpr ("**"): arithmetic operators.
  - BitOrExpr ("|"), XOrExpr ("^"), BitAndExpr ("&"): bitwise operators.
  - InExpr: an `in` test.
  - InstanceofExpr: an `instanceof` test.
  - LogAndExpr ("&&"), LogOrExpr ("||"): short-circuiting logical operators.
- Assignment: assignment expressions, either simple or compound; use `Assignment.getLhs()` and `Assignment.getRhs()` to access the left- and right-hand side, respectively.

        * AssignExpr: a simple assignment expression.
        * CompoundAssignExpr: a compound assignment expression.
            - AssignAddExpr, AssignSubExpr, AssignMulExpr, AssignDivExpr, AssignModExpr, AssignLShiftExpr, AssignRShiftExpr, AssignURShiftExpr, AssignOrExpr, AssignXOrExpr, AssignAndExpr, AssignExpExpr: different kinds of compound assignment expressions.
    - UpdateExpr: an increment or decrement expression; use `UpdateExpr.getOperand()` to obtain the operand expression.
        * PreIncExpr, PostIncExpr: an increment expression.
        * PreDecExpr, PostDecExpr: a decrement expression.
    - YieldExpr: a "yield" expression; use `YieldExpr.getOperand()` to access the (optional) operand expression; use `YieldExpr.isDelegating()` to check whether this is a delegating `yield*`.
    - TemplateLiteral: an ECMAScript 2015 template literal; `TemplateLiteral.getElement(i)` returns the `i`th element of the template, which may either be an interpolated expression or a constant template element.
    - TaggedTemplateExpr: an ECMAScript 2015 tagged template literal; use `TaggedTemplateExpr.getTag()` to access the tagging expression, and `TaggedTemplateExpr.getTemplate()` to access the template literal being tagged.
    - TemplateElement: a constant template element; as for literals, use `TemplateElement.getValue()` to obtain the value of the element, and `TemplateElement.getRawValue()` for its raw value
    - AwaitExpr: an "await" expression; use `AwaitExpr.getOperand()` to access the operand expression.

Stmt and Expr share a common superclass ExprOrStmt which is useful for queries that should operate either on statements or on expressions, but not on any other AST nodes.

As an example of how to use expression AST nodes, here is a query that finds expressions of the form `e + f >> g`; such expressions should be rewritten as `(e + f) >> g` to clarify operator precedence:

```
import javascript

from ShiftExpr shift, AddExpr add
where add = shift.getAnOperand()
select add, "This expression should be bracketed to clarify precedence rules."
```

See this in the query console on LGTM.com. When we ran this query on the *meteor/meteor* project in LGTM.com, we found many results where precedence could be clarified using brackets.

Functions

JavaScript provides several ways of defining functions: in ECMAScript 5, there are function declaration statements and function expressions, and ECMAScript 2015 adds arrow function expressions. These different syntactic forms are represented by the classes FunctionDeclStmt (a subclass of Stmt), FunctionExpr (a subclass of Expr) and ArrowFunctionExpr (also a subclass of Expr), respectively. All three are subclasses of Function, which provides common member predicates for accessing function parameters or the function body:

- `Function.getId()` returns the Identifier naming the function, which may not be defined for function expressions.
- `Function.getParameter(i)` and `Function.getAParameter()` access the `i`th parameter or any parameter, respectively; parameters are modeled by the class Parameter, which is a subclass of BindingPattern (see below).
- `Function.getBody()` returns the body of the function, which is usually a Stmt, but may be an Expr for arrow function expressions and legacy expression closures.

As an example, here is a query that finds all expression closures:

```
import javascript

from FunctionExpr fe
where fe.getBody() instanceof Expr
select fe, "Use arrow expressions instead of expression closures."
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects uses expression closures, but you may find this query gets results on other projects.

As another example, this query finds functions that have two parameters that bind the same variable:

```
import javascript

from Function fun, Parameter p, Parameter q, int i, int j
where p = fun.getParameter(i) and
    q = fun.getParameter(j) and
    i < j and
    p.getAVariable() = q.getAVariable()
select fun, "This function has two parameters that bind the same variable."
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects has functions where two parameters bind the same variable.

Classes

Classes can be defined either by class declaration statements, represented by the CodeQL class ClassDeclStmt (which is a subclass of Stmt), or by class expressions, represented by the CodeQL class ClassExpr (which is a subclass of Expr). Both of these classes are also subclasses of ClassDefinition, which provides common member predicates for accessing the name of a class, its superclass, and its body:

- `ClassDefinition.getIdentifier()` returns the Identifier naming the function, which may not be defined for class expressions.
- `ClassDefinition.getSuperClass()` returns the Expr specifying the superclass, which may not be defined.
- `ClassDefinition.getMember(n)` returns the definition of member `n` of this class.
- `ClassDefinition.getMethod(n)` restricts `ClassDefinition.getMember(n)` to methods (as opposed to fields).
- `ClassDefinition.getField(n)` restricts `ClassDefinition.getMember(n)` to fields (as opposed to methods).
- `ClassDefinition.getConstructor()` gets the constructor of this class, possibly a synthetic default constructor.

Note that class fields are not a standard language feature yet, so details of their representation may change.

Method definitions are represented by the class MethodDefinition, which (like its counterpart FieldDefinition for fields) is a subclass of MemberDefinition. That class provides the following important member predicates:

- `MemberDefinition.isStatic()`: holds if this is a static member.
- `MemberDefinition.isComputed()`: holds if the name of this member is computed at runtime.
- `MemberDefinition.getName()`: gets the name of this member if it can be determined statically.

- `MemberDefinition.getInit()`: gets the initializer of this field; for methods, the initializer is a function expressions, for fields it may be an arbitrary expression, and may be undefined.

There are three classes for modeling special methods: ConstructorDefinition models constructors, while GetterMethodDefinition and SetterMethodDefinition model getter and setter methods, respectively.

Declarations and binding patterns

Variables are declared by declaration statements (class DeclStmt), which come in three flavors: `var` statements (represented by class VarDeclStmt), `const` statements (represented by class ConstDeclStmt), and `let` statements (represented by class LetStmt). Every declaration statement has one or more declarators, represented by class VariableDeclarator.

Each declarator consists of a binding pattern, returned by predicate `VariableDeclarator.getBindingPattern()`, and an optional initializing expression, returned by `VariableDeclarator.getInit()`.

Often, the binding pattern is a simple identifier, as in `var x = 42`. In ECMAScript 2015 and later, however, it can also be a more complex destructuring pattern, as in `var [x, y] = arr`.

The various kinds of binding patterns are represented by class BindingPattern and its subclasses:

- VarRef: a simple identifier in an l-value position, for example the `x` in `var x` or in `x = 42`
- Parameter: a function or catch clause parameter
- ArrayPattern: an array pattern, for example, the left-hand side of `[x, y] = arr`
- ObjectPattern: an object pattern, for example, the left-hand side of `{x, y: z} = o`

Here is an example of a query to find declaration statements that declare the same variable more than once, excluding results in minified code:

```
import javascript

from DeclStmt ds, VariableDeclarator d1, VariableDeclarator d2, Variable v, int i, int j
where d1 = ds.getDecl(i) and
    d2 = ds.getDecl(j) and
    i < j and
    v = d1.getBindingPattern().getAVariable() and
    v = d2.getBindingPattern().getAVariable() and
    not ds.getTopLevel().isMinified()
select ds, "Variable " + v.getName() + " is declared both $@ and $@.", d1, "here", d2,
↪"here"
```

See this in the query console on LGTM.com. This is not a common problem, so you may not find any results in your own projects. The *angular/angular.js* project on LGTM.com has one instance of this problem at the time of writing.

> Notice the use of `not ... isMinified()` here and in the next few queries. This excludes any results found in minified code. If you delete `and not ds.getTopLevel().isMinified()` and re-run the query, two results in minified code in the *meteor/meteor* project are reported.

Properties

Properties in object literals are represented by class Property, which is also a subclass of ASTNode, but neither of Expr nor of Stmt.

Class Property has two subclasses ValueProperty and PropertyAccessor, which represent, respectively, normal value properties and getter/setter properties. Class PropertyAccessor, in turn, has two subclasses PropertyGetter and PropertySetter representing getters and setters, respectively.

The predicates `Property.getName()` and `Property.getInit()` provide access to the defined property's name and its initial value. For PropertyAccessor and its subclasses, `getInit()` is overloaded to return the getter/setter function.

As an example of a query involving properties, consider the following query that flags object expressions containing two identically named properties, excluding results in minified code:

```
import javascript

from ObjectExpr oe, Property p1, Property p2, int i, int j
where p1 = oe.getProperty(i) and
    p2 = oe.getProperty(j) and
    i < j and
    p1.getName() = p2.getName() and
    not oe.getTopLevel().isMinified()
select oe, "Property " + p1.getName() + " is defined both $@ and $@.", p1, "here", p2,
↪"here"
```

See this in the query console on LGTM.com. Many projects have a few instances of object expressions with two identically named properties.

Modules

The JavaScript library has support for working with ECMAScript 2015 modules, as well as legacy CommonJS modules (still commonly employed by Node.js code bases) and AMD-style modules. The classes ES2015Module, NodeModule, and AMDModule represent these three types of modules, and all three extend the common superclass Module.

The most important member predicates defined by Module are:

- `Module.getName()`: gets the name of the module, which is just the stem (that is, the basename without extension) of the enclosing file.
- `Module.getAnImportedModule()`: gets another module that is imported (through `import` or `require`) by this module.
- `Module.getAnExportedSymbol()`: gets the name of a symbol that this module exports.

Moreover, there is a class Import that models both ECMAScript 2015-style `import` declarations and CommonJS/AMD-style `require` calls; its member predicate `Import.getImportedModule` provides access to the module the import refers to, if it can be determined statically.

Name binding

Name binding is modeled in the JavaScript libraries using four concepts: *scopes*, *variables*, *variable declarations*, and *variable accesses*, represented by the classes Scope, Variable, VarDecl and VarAccess, respectively.

Scopes

In ECMAScript 5, there are three kinds of scopes: the global scope (one per program), function scopes (one per function), and catch clause scopes (one per `catch` clause). These three kinds of scopes are represented by the classes GlobalScope, FunctionScope and CatchScope. ECMAScript 2015 adds block scopes for `let`-bound variables, which are also represented by class Scope, class expression scopes (ClassExprScope), and module scopes (ModuleScope).

Class Scope provides the following API:

- `Scope.getScopeElement()` returns the AST node inducing this scope; undefined for GlobalScope.
- `Scope.getOuterScope()` returns the lexically enclosing scope of this scope.
- `Scope.getAnInnerScope()` returns a scope lexically nested inside this scope.
- `Scope.getVariable(name)`, `Scope.getAVariable()` return a variable declared (implicitly or explicitly) in this scope.

Variables

The Variable class models all variables in a JavaScript program, including global variables, local variables, and parameters (both of functions and `catch` clauses), whether explicitly declared or not.

It is important not to confuse variables and their declarations: local variables may have more than one declaration, while global variables and the implicitly declared local `arguments` variable need not have a declaration at all.

Variable declarations and accesses

Variables may be declared by variable declarators, by function declaration statements and expressions, by class declaration statements or expressions, or by parameters of functions and `catch` clauses. While these declarations differ in their syntactic form, in each case there is an identifier naming the declared variable. We consider that identifier to be the declaration proper, and assign it the class VarDecl. Identifiers that reference a variable, on the other hand, are given the class VarAccess.

The most important predicates involving variables, their declarations, and their accesses are as follows:

- `Variable.getName()`, `VarDecl.getName()`, `VarAccess.getName()` return the name of the variable.
- `Variable.getScope()` returns the scope to which the variable belongs.
- `Variable.isGlobal()`, `Variable.isLocal()`, `Variable.isParameter()` determine whether the variable is a global variable, a local variable, or a parameter variable, respectively.
- `Variable.getAnAccess()` maps a Variable to all VarAccesses that refer to it.
- `Variable.getADeclaration()` maps a Variable to all VarDecls that declare it (of which there may be none, one, or more than one).
- `Variable.isCaptured()` determines whether the variable is ever accessed in a scope that is lexically nested within the scope where it is declared.

As an example, consider the following query which finds distinct function declarations that declare the same variable, that is, two conflicting function declarations within the same scope (again excluding minified code):

```
import javascript

from FunctionDeclStmt f, FunctionDeclStmt g
where f != g and f.getVariable() = g.getVariable() and
    not f.getTopLevel().isMinified() and
    not g.getTopLevel().isMinified()
select f, g
```

See this in the query console on LGTM.com. Some projects declare conflicting functions of the same name and rely on platform-specific behavior to disambiguate the two declarations.

Control flow

A different program representation in terms of intraprocedural control flow graphs (CFGs) is provided by the classes in library CFG.qll.

Class ControlFlowNode represents a single node in the control flow graph, which is either an expression, a statement, or a synthetic control flow node. Note that Expr and Stmt do not inherit from ControlFlowNode at the CodeQL level, although their entity types are compatible, so you can explicitly cast from one to the other if you need to map between the AST-based and the CFG-based program representations.

There are two kinds of synthetic control flow nodes: entry nodes (class ControlFlowEntryNode), which represent the beginning of a top-level or function, and exit nodes (class ControlFlowExitNode), which represent their end. They do not correspond to any AST nodes, but simply serve as the unique entry point and exit point of a control flow graph. Entry and exit nodes can be accessed through the predicates `StmtContainer.getEntry()` and `StmtContainer.getExit()`.

Most, but not all, top-levels and functions have another distinguished CFG node, the *start node*. This is the CFG node at which execution begins. Unlike the entry node, which is a synthetic construct, the start node corresponds to an actual program element: for top-levels, it is the first CFG node of the first statement; for functions, it is the CFG node corresponding to their first parameter or, if there are no parameters, the first CFG node of the body. Empty top-levels do not have a start node.

For most purposes, using start nodes is preferable to using entry nodes.

The structure of the control flow graph is reflected in the member predicates of ControlFlowNode:

- `ControlFlowNode.getASuccessor()` returns a ControlFlowNode that is a successor of this ControlFlowNode in the control flow graph.
- `ControlFlowNode.getAPredecessor()` is the inverse of `getASuccessor()`.
- `ControlFlowNode.isBranch()` determines whether this node has more than one successor.
- `ControlFlowNode.isJoin()` determines whether this node has more than one predecessor.
- `ControlFlowNode.isStart()` determines whether this node is a start node.

Many control-flow-based analyses are phrased in terms of basic blocks rather than single control flow nodes, where a basic block is a maximal sequence of control flow nodes without branches or joins. The class BasicBlock from BasicBlocks.qll represents all such basic blocks. Similar to ControlFlowNode, it provides member predicates `getASuccessor()` and `getAPredecessor()` to navigate the control flow graph at the level of basic blocks, and member predicates `getANode()`, `getNode(int)`, `getFirstNode()` and `getLastNode()` to access individual control flow nodes within a basic block. The predicate `Function.getEntryBB()` returns the entry basic block in a function, that is, the basic block containing the function's entry node. Similarly, `Function.getStartBB()` provides access to the start basic block, which contains the function's start node. As for CFG nodes, `getStartBB()` should normally be preferred over `getEntryBB()`.

As an example of an analysis using basic blocks, `BasicBlock.isLiveAtEntry(v, u)` determines whether variable `v` is live at the entry of the given basic block, and if so binds `u` to a use of `v` that refers to its value at the entry. We can use it to find global variables that are used in a function where they are not live (that is, every read of the variable is preceded by a write), suggesting that the variable was meant to be declared as a local variable instead:

```
import javascript

from Function f, GlobalVariable gv
where gv.getAnAccess().getEnclosingFunction() = f and
    not f.getStartBB().isLiveAtEntry(gv, _)
select f, "This function uses " + gv + " like a local variable."
```

See this in the query console on LGTM.com. Many projects have some variables which look as if they were intended to be local.

Data flow

Definitions and uses

Library DefUse.qll provides classes and predicates to determine def-use relationships between definitions and uses of variables.

Classes VarDef and VarUse contain all expressions that define and use a variable, respectively. For the former, you can use predicate `VarDef.getAVariable()` to find out which variables are defined by a given variable definition (recall that destructuring assignments in ECMAScript 2015 define several variables at the same time). Similarly, predicate `VarUse.getVariable()` returns the (single) variable being accessed by a variable use.

The def-use information itself is provided by predicate `VarUse.getADef()`, that connects a use of a variable to a definition of the same variable, where the definition may reach the use.

As an example, the following query finds definitions of local variables that are not used anywhere; that is, the variable is either not referenced at all after the definition, or its value is overwritten:

```
import javascript

from VarDef def, LocalVariable v
where v = def.getAVariable() and
    not exists (VarUse use | def = use.getADef())
select def, "Dead store of local variable."
```

See this in the query console on LGTM.com. Many projects have some examples of useless assignments to local variables.

SSA

A more fine-grained representation of a program's data flow based on Static Simple Assignment Form (SSA) is provided by the library `semmle.javascript.SSA`.

In SSA form, each use of a local variable has exactly one (SSA) definition that reaches it. SSA definitions are represented by class SsaDefinition. They are not AST nodes, since not every SSA definition corresponds to an explicit element in the source code.

Altogether, there are five kinds of SSA definitions:

1. Explicit definitions (SsaExplicitDefinition): these simply wrap a VarDef, that is, a definition like `x = 1` appearing explicitly in the source code.
2. Implicit initializations (SsaImplicitInit): these represent the implicit initialization of local variables with `undefined` at the beginning of their scope.
3. Phi nodes (SsaPhiNode): these are pseudo-definitions that merge two or more SSA definitions where necessary; see the Wikipedia page linked to above for an explanation.
4. Variable captures (SsaVariableCapture): these are pseudo-definitions appearing at places in the code where the value of a captured variable may change without there being an explicit assignment, for example due to a function call.
5. Refinement nodes (SsaRefinementNode): these are pseudo-definitions appearing at places in the code where something becomes known about a variable; for example, a conditional `if (x === null)` induces a refinement node at the beginning of its "then" branch recording the fact that `x` is known to be `null` there. (In the literature, these are sometimes known as "pi nodes.")

Data flow nodes

Moving beyond just variable definitions and uses, library `semmle.javascript.dataflow.DataFlow` provides a representation of the program as a data flow graph. Its nodes are values of class DataFlow::Node, which has two subclasses `ValueNode` and `SsaDefinitionNode`. Nodes of the former kind wrap an expression or a statement that is considered to produce a value (specifically, a function or class declaration statement, or a TypeScript namespace or enum declaration). Nodes of the latter kind wrap SSA definitions.

You can use the predicate `DataFlow::valueNode` to convert an expression, function or class into its corresponding `ValueNode`, and similarly `DataFlow::ssaDefinitionNode` to map an SSA definition to its corresponding `SsaDefinitionNode`.

There is also an auxiliary predicate `DataFlow::parameterNode` that maps a parameter to its corresponding data flow node. (This is really just a convenience wrapper around `DataFlow::ssaDefinitionNode`, since parameters are also considered to be SSA definitions.)

Going in the other direction, there is a predicate `ValueNode.getAstNode()` for mapping from `ValueNode`s to `ASTNode`s, and `SsaDefinitionNode.getSsaVariable()` for mapping from `SsaDefinitionNode`s to `SsaVariable`s. There is also a utility predicate `Node.asExpr()` that gets the underlying expression for a `ValueNode`, and is undefined for all nodes that do not correspond to an expression. (Note in particular that this predicate is not defined for `ValueNode`s wrapping function or class declaration statements!)

You can use the predicate `DataFlow::Node.getAPredecessor()` to find other data flow nodes from which values may flow into this node, and `getASuccessor` for the other direction.

For example, here is a query that finds all invocations of a method called `send` on a value that comes from a parameter named `res`, indicating that it is perhaps sending an HTTP response:

```
import javascript

from SimpleParameter res, DataFlow::Node resNode, MethodCallExpr send
where res.getName() = "res" and
      resNode = DataFlow::parameterNode(res) and
      resNode.getASuccessor+() = DataFlow::valueNode(send.getReceiver()) and
      send.getMethodName() = "send"
select send
```

See this in the query console on LGTM.com. The query finds HTTP response sends in the AMP HTML project.

Note that the data flow modeling in this library is intraprocedural, that is, flow across function calls and returns is *not* modeled. Likewise, flow through object properties and global variables is not modeled.

Type inference

The library `semmle.javascript.dataflow.TypeInference` implements a simple type inference for JavaScript based on intraprocedural, heap-insensitive flow analysis. Basically, the inference algorithm approximates the possible concrete runtime values of variables and expressions as sets of abstract values (represented by the class AbstractValue), each of which stands for a set of concrete values.

For example, there is an abstract value representing all non-zero numbers, and another representing all non-empty strings except for those that can be converted to a number. Both of these abstract values are fairly coarse approximations that represent very large sets of concrete values.

Other abstract values are more precise, to the point where they represent single concrete values: for example, there is an abstract value representing the concrete `null` value, and another representing the number zero.

There is a special group of abstract values called *indefinite* abstract values that represent all concrete values. The analysis uses these to handle expressions for which it cannot infer a more precise value, such as function parameters (as mentioned above, the analysis is intraprocedural and hence does not model argument passing) or property reads (the analysis does not model property values either).

Each indefinite abstract value is associated with a string value describing the cause of imprecision. In the above examples, the indefinite value for the parameter would have cause `"call"`, while the indefinite value for the property would have cause `"heap"`.

To check whether an abstract value is indefinite, you can use the `isIndefinite` member predicate. Its single argument describes the cause of imprecision.

Each abstract value has one or more associated types (CodeQL class InferredType corresponding roughly to the type tags computed by the `typeof` operator. The types are `null`, `undefined`, `boolean`, `number`, `string`, `function`, `class`, `date` and `object`.

To access the results of the type inference, use class DataFlow::AnalyzedNode: any DataFlow::Node can be cast to this class, and additionally there is a convenience predicate `Expr::analyze` that maps expressions directly to their corresponding `AnalyzedNode`s.

Once you have an `AnalyzedNode`, you can use predicate `AnalyzedNode.getAValue()` to access the abstract values inferred for it, and `getAType()` to get the inferred types.

For example, here is a query that looks for `null` checks on expressions that cannot, in fact, be null:

```
import javascript

from StrictEqualityTest eq, DataFlow::AnalyzedNode nd, NullLiteral null
where eq.hasOperands(nd.asExpr(), null) and
      not nd.getAValue().isIndefinite(_) and
      not nd.getAValue() instanceof AbstractNull
select eq, "Spurious null check."
```

To paraphrase, the query looks for equality tests `eq` where one operand is a `null` literal and the other some expression that we convert to an `AnalyzedNode`. If the type inference results for that node are precise (that is, none of the inferred values is indefinite) and (the abstract representation of) `null` is not among them, we flag `eq`.

You can add custom type inference rules by defining new subclasses of `DataFlow::AnalyzedNode` and overriding `getAValue`. You can also introduce new abstract values by extending the abstract class `CustomAbstractValueTag`, which is a subclass of `string`: each string belonging to that class induces a corresponding abstract value of type `CustomAbstractValue`. You can use the predicate `CustomAbstractValue.getTag()` to map from the abstract

value to its tag. By implementing the abstract predicates of class `CustomAbstractValueTag` you can define the semantics of your custom abstract values, such as what primitive value they coerce to and what type they have.

Call graph

The JavaScript library implements a simple call graph construction algorithm to statically approximate the possible call targets of function calls and `new` expressions. Due to the dynamically typed nature of JavaScript and its support for higher-order functions and reflective language features, building static call graphs is quite difficult. Simple call graph algorithms tend to be incomplete, that is, they often fail to resolve all possible call targets. More sophisticated algorithms can suffer from the opposite problem of imprecision, that is, they may infer many spurious call targets.

The call graph is represented by the member predicate `getACallee()` of class DataFlow::InvokeNode, which computes possible callees of the given invocation, that is, functions that may at runtime be invoked by this expression.

Furthermore, there are three member predicates that indicate the quality of the callee information for this invocation:

- `DataFlow::InvokeNode.isImprecise()`: holds for invocations where the call graph builder might infer spurious call targets.
- `DataFlow::InvokeNode.isIncomplete()`: holds for invocations where the call graph builder might fail to infer possible call targets.
- `DataFlow::InvokeNode.isUncertain()`: holds if either `isImprecise()` or `isIncomplete()` holds.

As an example of a call-graph-based query, here is a query to find invocations for which the call graph builder could not find any callees, despite the analysis being complete for this invocation:

```
import javascript

from DataFlow::InvokeNode invk
where not invk.isIncomplete() and
      not exists(invk.getACallee())
select invk, "Unable to find a callee for this invocation."
```

See this in the query console on LGTM.com

Inter-procedural data flow

The data flow graph-based analyses described so far are all intraprocedural: they do not take flow from function arguments to parameters or from a `return` to the function's caller into account. The data flow library also provides a framework for constructing custom inter-procedural analyses.

We distinguish here between data flow proper, and *taint tracking*: the latter not only considers value-preserving flow (such as from variable definitions to uses), but also cases where one value influences ("taints") another without determining it entirely. For example, in the assignment `s2 = s1.substring(i)`, the value of `s1` influences the value of `s2`, because `s2` is assigned a substring of `s1`. In general, `s2` will not be assigned `s1` itself, so there is no data flow from `s1` to `s2`, but `s1` still taints `s2`.

The simplest way of implementing an interprocedural data flow analysis is to extend either class `DataFlow::TrackedNode` or `DataFlow::TrackedExpr`. The former is a subclass of `DataFlow::Node`, the latter of `Expr`, and extending them ensures that the newly added values are tracked interprocedurally. You can use the predicate `flowsTo` to find out which nodes/expressions the tracked value flows to.

For example, suppose that we are developing an analysis to find hard-coded passwords. We might start by writing a simple query that looks for string constants flowing into variables named `"password"`. To do this, we can extend `TrackedExpr` to track all constant strings, `flowsTo` to find cases where such a string flows into a (SSA) definition of a password variable:

```
import javascript

class TrackedStringLiteral extends DataFlow::TrackedNode {
    TrackedStringLiteral() {
        this.asExpr() instanceof ConstantString
    }
}

from TrackedStringLiteral source, DataFlow::Node sink, SsaExplicitDefinition def
where source.flowsTo(sink) and sink = DataFlow::ssaDefinitionNode(def) and
      def.getSourceVariable().getName().toLowerCase() = "password"
select sink
```

Note that `TrackedNode` and `TrackedExpr` do not restrict the set of "sinks" for the inter-procedural flow analysis, tracking flow into any expression that they might flow to. This can be expensive for large code bases, and is often unnecessary, since usually you are only interested in flow to a particular set of sinks. For example, the above query only looks for flow into assignments to password variables.

This is a particular instance of a general pattern, whereby we want to specify a data flow or taint analysis in terms of its *sources* (where flow starts), *sinks* (where it should be tracked), and *barriers* or *sanitizers* (where flow is interrupted). The example does not include any sanitizers, but they are very common in security analyses: for example, an analysis that tracks the flow of untrusted user input into, say, a SQL query has to keep track of code that validates the input, thereby making it safe to use. Such a validation step is an example of a sanitizer.

The classes `DataFlow::Configuration` and `TaintTracking::Configuration` allow specifying a data flow or taint analysis, respectively, by overriding the following predicates:

- `isSource(DataFlow::Node nd)` selects all nodes `nd` from where flow tracking starts.
- `isSink(DataFlow::Node nd)` selects all nodes `nd` to which the flow is tracked.
- `isBarrier(DataFlow::Node nd)` selects all nodes `nd` that act as a barrier for data flow; `isSanitizer` is the corresponding predicate for taint tracking configurations.
- `isBarrierEdge(DataFlow::Node src, DataFlow::Node trg)` is a variant of `isBarrier(nd)` that allows specifying barrier *edges* in addition to barrier nodes; again, `isSanitizerEdge` is the corresponding predicate for taint tracking;
- `isAdditionalFlowStep(DataFlow::Node src, DataFlow::Node trg)` allows specifying custom additional flow steps for this analysis; `isAdditionalTaintStep` is the corresponding predicate for taint tracking configurations.

Since for technical reasons both `Configuration` classes are subtypes of `string`, you have to choose a unique name for each flow configuration and equate `this` with it in the characteristic predicate (as in the example below).

The predicate `Configuration.hasFlow` performs the actual flow tracking, starting at a source and looking for flow to a sink that does not pass through a barrier node or edge.

To continue with our above example, we can phrase it as a data flow configuration as follows:

```
class PasswordTracker extends DataFlow::Configuration {
    PasswordTracker() {
        // unique identifier for this configuration
        this = "PasswordTracker"
    }

    override predicate isSource(DataFlow::Node nd) {
        nd.asExpr() instanceof StringLiteral
```

(continues on next page)

(continued from previous page)

```
    }

    override predicate isSink(DataFlow::Node nd) {
        passwordVarAssign(_, nd)
    }

    predicate passwordVarAssign(Variable v, DataFlow::Node nd) {
        exists (SsaExplicitDefinition def |
            nd = DataFlow::ssaDefinitionNode(def) and
            def.getSourceVariable() = v and
            v.getName().toLowerCase() = "password"
        )
    }
}
```

Now we can rephrase our query to use `Configuration.hasFlow`:

```
from PasswordTracker pt, DataFlow::Node source, DataFlow::Node sink, Variable v
where pt.hasFlow(source, sink) and pt.passwordVarAssign(v, sink)
select sink, "Password variable " + v + " is assigned a constant string."
```

Note that while analyses implemented in this way are inter-procedural in that they track flow and taint across function calls and returns, flow through global variables is not tracked. Flow through object properties is only tracked in limited cases, for example through properties of object literals or CommonJS `module` and `exports` objects.

Syntax errors

JavaScript code that contains syntax errors cannot usually be analyzed. For such code, the lexical and syntactic representations are not available, and hence no name binding information, call graph or control and data flow. All that is available in this case is a value of class JSParseError representing the syntax error. It provides information about the syntax error location (JSParseError is a subclass of Locatable) and the error message through predicate `JSParseError.getMessage`.

Note that for some very simple syntax errors the parser can recover and continue parsing. If this happens, lexical and syntactic information is available in addition to the JSParseError values representing the (recoverable) syntax errors encountered during parsing.

Frameworks

AngularJS

The `semmle.javascript.frameworks.AngularJS` library provides support for working with AngularJS (Angular 1.x) code. Its most important classes are:

- AngularJS::AngularModule: an Angular module
- AngularJS::DirectiveDefinition, AngularJS::FactoryRecipeDefinition, AngularJS::FilterDefinition, AngularJS::ControllerDefinition: a definition of a directive, service, filter or controller, respectively
- AngularJS::InjectableFunction: a function that is subject to dependency injection

HTTP framework libraries

The library `semmle.javacript.frameworks.HTTP` provides classes modeling common concepts from various HTTP frameworks.

Currently supported frameworks are Express, the standard Node.js `http` and `https` modules, Connect, Koa, Hapi and Restify.

The most important classes include (all in module `HTTP`):

- `ServerDefinition`: an expression that creates a new HTTP server.
- `RouteHandler`: a callback for handling an HTTP request.
- `RequestExpr`: an expression that may contain an HTTP request object.
- `ResponseExpr`: an expression that may contain an HTTP response object.
- `HeaderDefinition`: an expression that sets one or more HTTP response headers.
- `CookieDefinition`: an expression that sets a cookie in an HTTP response.
- `RequestInputAccess`: an expression that accesses user-controlled request data.

For each framework library, there is a corresponding CodeQL library (for example `semmle.javacript.frameworks.Express`) that instantiates the above classes for that framework and adds framework-specific classes.

Node.js

The `semmle.javascript.NodeJS` library provides support for working with Node.js modules through the following classes:

- NodeModule: a top-level that defines a Node.js module; see the section on *Modules* for more information.
- Require: a call to the special `require` function that imports a module.

As an example of the use of these classes, here is a query that counts for every module how many other modules it imports:

```
import javascript

from NodeModule m
select m, count(m.getAnImportedModule())
```

See this in the query console on LGTM.com. When you analyze a project, for each module you can see how many other modules it imports.

NPM

The `semmle.javascript.NPM` library provides support for working with NPM packages through the following classes:

- PackageJSON: a `package.json` file describing an NPM package; various getter predicates are available for accessing detailed information about the package, which are described in the online API documentation.
- BugTrackerInfo, ContributorInfo, RepositoryInfo: these classes model parts of the `package.json` file providing information on bug tracking systems, contributors and repositories.

- PackageDependencies: models the dependencies of an NPM package; the predicate `PackageDependencies.getADependency(pkg, v)` binds `pkg` to the name and `v` to the version of a package required by a `package.json` file.
- NPMPackage: a subclass of Folder that models an NPM package; important member predicates include:
  - `NPMPackage.getPackageName()` returns the name of this package.
  - `NPMPackage.getPackageJSON()` returns the `package.json` file for this package.
  - `NPMPackage.getNodeModulesFolder()` returns the `node_modules` folder for this package.
  - `NPMPackage.getAModule()` returns a Node.js module belonging to this package (not including modules in the `node_modules` folder).

As an example of the use of these classes, here is a query that identifies unused dependencies, that is, module dependencies that are listed in the `package.json` file, but which are not imported by any `require` call:

```
import javascript

from NPMPackage pkg, PackageDependencies deps, string name
where deps = pkg.getPackageJSON().getDependencies() and
deps.getADependency(name, _) and
not exists (Require req | req.getTopLevel() = pkg.getAModule() | name = req.
↪getImportedPath().getValue())
select deps, "Unused dependency '" + name + "'."
```

See this in the query console on LGTM.com. It is not uncommon for projects to have some unused dependencies.

React

The `semmle.javascript.frameworks.React` library provides support for working with React code through the ReactComponent class, which models a React component defined either in the functional style or the class-based style (both ECMAScript 2015 classes and old-style `React.createClass` classes are supported).

Databases

The class `SQL::SqlString` represents an expression that is interpreted as a SQL command. Currently, we model SQL commands issued through the following npm packages: mysql, pg, pg-pool, sqlite3, mssql and sequelize.

Similarly, the class `NoSQL::Query` represents an expression that is interpreted as a NoSQL query by the `mongodb` or `mongoose` package.

Finally, the class `DatabaseAccess` contains all data flow nodes that perform a database access using any of the packages above.

For example, here is a query to find SQL queries that use string concatenation (instead of a templating-based solution, which is usually safer):

```
import javascript

from SQL::SqlString ss
where ss instanceof AddExpr
select ss, "Use templating instead of string concatenation."
```

See this in the query console on LGTM.com, showing two (benign) results on strong-arc.

Miscellaneous

Externs

The `semmle.javascript.Externs` library provides support for working with externs through the following classes:

- ExternalDecl: common superclass modeling all different kinds of externs declarations; it defines two member predicates:
  - `ExternalDecl.getQualifiedName()` returns the fully qualified name of the declared entity.
  - `ExternalDecl.getName()` returns the unqualified name of the declared entity.
- ExternalTypedef: a subclass of ExternalDecl representing type declarations; unlike other externs declarations, such declarations do not declare a function or object that is present at runtime, but simply introduce an alias for a type.
- ExternalVarDecl: a subclass of ExternalDecl representing a variable or function declaration; it defines two member predicates:
  - `ExternalVarDecl.getInit()` returns the initializer associated with this declaration, if any; this can either be a Function or an Expr.
  - `ExternalVarDecl.getDocumentation()` returns the JSDoc comment associated with this declaration.

Variables and functions declared in an externs file are either globals (represented by class ExternalGlobalDecl), or members (represented by class ExternalMemberDecl).

Members are further subdivided into static members (class ExternalStaticMemberDecl) and instance members (class ExternalInstanceMemberDecl).

For more details on these and other classes representing externs, see the API documentation.

HTML

The `semmle.javascript.HTML` library provides support for working with HTML documents. They are represented as a tree of `HTML::Element` nodes, each of which may have zero or more attributes represented by class `HTML::Attribute`.

Similar to the abstract syntax tree representation, `HTML::Element` has member predicates `getChild(i)` and `getParent()` to navigate from an element to its `i`th child element and its parent element, respectively. Use predicate `HTML::Element.getAttribute(i)` to get the `i`th attribute of the element, and `HTML::Element.getAttributeByName(n)` to get the attribute with name `n`.

For `HTML::Attribute`, predicates `getName()` and `getValue()` provide access to the attribute's name and value, respectively.

Both `HTML::Element` and `HTML::Attribute` have a predicate `getRoot()` that gets the root `HTML::Element` of the document to which they belong.

JSDoc

The `semmle.javascript.JSDoc` library provides support for working with JSDoc comments. Documentation comments are parsed into an abstract syntax tree representation closely following the format employed by the Doctrine JSDoc parser.

A JSDoc comment as a whole is represented by an entity of class JSDoc, while individual tags are represented by class JSDocTag. Important member predicates of these two classes include:

- `JSDoc.getDescription()` returns the descriptive header of the JSDoc comment, if any.
- `JSDoc.getComment()` maps the JSDoc entity to its underlying Comment entity.
- `JSDocTag.getATag()` returns a tag in this JSDoc comment.
- `JSDocTag.getTitle()` returns the title of his tag; for instance, an `@param` tag has title `"param"`.
- `JSDocTag.getName()` returns the name of the parameter or variable documented by this tag.
- `JSDocTag.getType()` returns the type of the parameter or variable documented by this tag.
- `JSDocTag.getDescription()` returns the description associated with this tag.

Types in JSDoc comments are represented by the class JSDocTypeExpr and its subclasses, which again represent type expressions as abstract syntax trees. Examples of type expressions are JSDocAnyTypeExpr, representing the "any" type *, or JSDocNullTypeExpr, representing the null type.

As an example, here is a query that finds `@param` tags that do not specify the name of the documented parameter:

```
import javascript

from JSDocTag t
where t.getTitle() = "param" and
not exists(t.getName())
select t, "@param tag is missing name."
```

See this in the query console on LGTM.com. Of the LGTM.com demo projects analyzed, only *Semantic-Org/Semantic-UI* has an example where the `@param` tag omits the name.

For full details on these and other classes representing JSDoc comments and type expressions, see the API documentation.

JSX

The `semmle.javascript.JSX` library provides support for working with JSX code.

Similar to the representation of HTML documents, JSX fragments are modeled as a tree of JSXElements, each of which may have zero or more JSXAttributes.

However, unlike HTML, JSX is interleaved with JavaScript, hence JSXElement is a subclass of Expr. Like `HTML::Element`, it has predicates `getAttribute(i)` and `getAttributeByName(n)` to look up attributes of a JSX element. Its body elements can be accessed by predicate `getABodyElement()`; note that the results of this predicate are arbitrary expressions, which may either be further JSXElements, or other expressions that are interpolated into the body of the outer element.

JSXAttribute, again not unlike `HTML::Attribute`, has predicates `getName()` and `getValue()` to access the attribute name and value.

JSON

The `semmle.javascript.JSON` library provides support for working with JSON files that were processed by the JavaScript extractor when building the CodeQL database.

JSON files are modeled as trees of JSON values. Each JSON value is represented by an entity of class JSONValue, which provides the following member predicates:

- `JSONValue.getParent()` returns the JSON object or array in which this value occurs.
- `JSONValue.getChild(i)` returns the `i`th child of this JSON object or array.

Note that JSONValue is a subclass of Locatable, so the usual member predicates of Locatable can be used to determine the file in which a JSON value appears, and its location within that file.

Class JSONValue has the following subclasses:

- JSONPrimitiveValue: a JSON-encoded primitive value; use `JSONPrimitiveValue.getValue()` to obtain a string representation of the value.
  - JSONNull, JSONBoolean, JSONNumber, JSONString: subclasses of JSONPrimitiveValue representing the various kinds of primitive values.
- JSONArray: a JSON-encoded array; use `JSONArray.getElementValue(i)` to access the `i`th element of the array.
- JSONObject: a JSON-encoded object; use `JSONObject.getValue(n)` to access the value of property `n` of the object.

Regular expressions

The `semmle.javascript.Regexp` library provides support for working with regular expression literals. The syntactic structure of regular expression literals is represented as an abstract syntax tree of regular expression terms, modeled by the class RegExpTerm. Similar to ASTNode, class RegExpTerm provides member predicates `getParent()` and `getChild(i)` to navigate the structure of the syntax tree.

Various subclasses of RegExpTerm model different kinds of regular expression constructs and operators; see the API documentation for details.

YAML

The `semmle.javascript.YAML` library provides support for working with YAML files that were processed by the JavaScript extractor when building the CodeQL database.

YAML files are modeled as trees of YAML nodes. Each YAML node is represented by an entity of class YAMLNode, which provides, among others, the following member predicates:

- `YAMLNode.getParentNode()` returns the YAML collection in which this node is syntactically nested.
- `YAMLNode.getChildNode(i)` returns the `i`th child node of this node, `YAMLNode.getAChildNode()` returns any child node of this node.
- `YAMLNode.getTag()` returns the tag of this YAML node.
- `YAMLNode.getAnchor()` returns the anchor associated with this YAML node, if any.
- `YAMLNode.eval()` returns the YAMLValue this YAML node evaluates to after resolving aliases and includes.

The various kinds of scalar values available in YAML are represented by classes YAMLInteger, YAMLFloat, YAMLTimestamp, YAMLBool, YAMLNull and YAMLString. Their common superclass is YAMLScalar, which has a member predicate `getValue()` to obtain the value of a scalar as a string.

YAMLMapping and YAMLSequence represent mappings and sequences, respectively, and are subclasses of YAMLCollection.

Alias nodes are represented by class YAMLAliasNode, while YAMLMergeKey and YAMLInclude represent merge keys and `!include` directives, respectively.

Predicate `YAMLMapping.maps(key, value)` models the key-value relation represented by a mapping, taking merge keys into account.

Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"

5.5.3 CodeQL library for TypeScript

When you're analyzing a TypeScript program, you can make use of the large collection of classes in the CodeQL library for TypeScript.

Overview

Support for analyzing TypeScript code is bundled with the CodeQL libraries for JavaScript, so you can include the full TypeScript library by importing the `javascript.qll` module:

```
import javascript
```

CodeQL libraries for JavaScript covers most of this library, and is also relevant for TypeScript analysis. This document supplements the JavaScript documentation with the TypeScript-specific classes and predicates.

Syntax

Most syntax in TypeScript is represented in the same way as its JavaScript counterpart. For example, `a+b` is represented by an AddExpr; the same as it would be in JavaScript. On the other hand, `x as number` is represented by TypeAssertion, a class that is specific to TypeScript.

Type annotations

The TypeExpr class represents anything that is part of a type annotation.

Only type annotations that are explicit in the source code occur as a `TypeExpr`. Types inferred by the TypeScript compiler are `Type` entities; for details about this, see the section on *static type information*.

There are several ways to access type annotations, for example:

- `VariableDeclaration.getTypeAnnotation()`
- `Function.getReturnTypeAnnotation()`
- `BindingPattern.getTypeAnnotation()`
- `Parameter.getTypeAnnotation()` (special case of `BindingPattern.getTypeAnnotation()`)
- `VarDecl.getTypeAnnotation()` (special case of `BindingPattern.getTypeAnnotation()`)
- `FieldDeclaration.getTypeAnnotation()`

The TypeExpr class provides some convenient member predicates such as `isString()` and `isVoid()` to recognize commonly used types.

The subclasses that represent type annotations are:

- TypeAccess: a name referring to a type, such as `Date` or `http.ServerRequest`.
  - LocalTypeAccess: an unqualified name, such as `Date`.
  - QualifiedTypeAccess: a name prefixed by a namespace, such as `http.ServerRequest`.
  - ImportTypeAccess: an `import` used as a type, such as `import("./foo")`.
- PredefinedTypeExpr: a predefined type, such as `number`, `string`, `void`, or `any`.
- ThisTypeExpr: the `this` type.
- InterfaceTypeExpr, also known as a literal type, such as `{x: number}`.
- FunctionTypeExpr: a type such as `(x: number) => string`.
- GenericTypeExpr: a named type with type arguments, such as `Array<string>`.
- LiteralTypeExpr: a string, number, or boolean constant used as a type, such as `'foo'`.
- ArrayTypeExpr: a type such as `string[]`.
- UnionTypeExpr: a type such as `string | number`.
- IntersectionTypeExpr: a type such as `S & T`.
- IndexedAccessTypeExpr: a type such as `T[K]`.
- ParenthesizedTypeExpr: a type such as `(string)`.
- TupleTypeExpr: a type such as `[string, number]`.
- KeyofTypeExpr: a type such as `keyof T`.
- TypeofTypeExpr: a type such as `typeof x`.
- IsTypeExpr: a type such as `x is string`.
- MappedTypeExpr: a type such as `{ [K in C]: T }`.

There are some subclasses that may be part of a type annotation, but are not themselves types:

- TypeParameter: a type parameter declared on a type or function, such as `T` in `class C<T> {}`.

- NamespaceAccess: a name referring to a namespace from inside a type, such as `http` in `http.ServerRequest`.
  - LocalNamespaceAccess: the initial identifier in a prefix, such as `http` in `http.ServerRequest`.
  - QualifiedNamespaceAccess: a qualified name in a prefix, such as `net.client` in `net.client.Connection`.
  - ImportNamespaceAccess: an `import` used as a namespace in a type, such as in `import("http").ServerRequest`.
- VarTypeAccess: a reference to a value from inside a type, such as `x` in `typeof x` or `x is string`.

Function signatures

The Function class is a broad class that includes both concrete functions and function signatures.

Function signatures can take several forms:

- Function types, such as `(x: number) => string`.
- Abstract methods, such as `abstract foo(): void`.
- Overload signatures, such as `foo(x: number): number` followed by an implementation of `foo`.
- Call signatures, such as in `{ (x: string): number }`.
- Index signatures, such as in `{ [x: string]: number }`.
- Functions in an ambient context, such as `declare function foo(x: number): string`.

We recommend that you use the predicate `Function.hasBody()` to distinguish concrete functions from signatures.

Type parameters

The TypeParameter class represents type parameters, and the TypeParameterized class represents entities that can declare type parameters. Classes, interfaces, type aliases, functions, and mapped type expressions are all `TypeParameterized`.

You can access type parameters using the following predicates:

- `TypeParameterized.getTypeParameter(n)` gets the nth declared type parameter.
- `TypeParameter.getHost()` gets the entity declaring a given type parameter.

You can access type arguments using the following predicates:

- `GenericTypeExpr.getTypeArgument(n)` gets the nth type argument of a type.
- `TypeAccess.getTypeArgument(n)` is a convenient alternative for the above (a TypeAccess with type arguments is wrapped in a GenericTypeExpr).
- `InvokeExpr.getTypeArgument(n)` gets the nth type argument of a call.
- `ExpressionWithTypeArguments.getTypeArgument(n)` gets the nth type argument of a generic superclass expression.

To select references to a given type parameter, use `getLocalTypeName()` (see *Name binding* below).

Examples

Select expressions that cast a value to a type parameter:

```
import javascript

from TypeParameter param, TypeAssertion assertion
where assertion.getTypeAnnotation() = param.getLocalTypeName().getAnAccess()
select assertion, "Cast to type parameter."
```

See this in the query console on LGTM.com.

Classes and interfaces

The CodeQL class ClassOrInterface is a common supertype of classes and interfaces, and provides some TypeScript-specific member predicates:

- `ClassOrInterface.isAbstract()` holds if this is an interface or a class with the `abstract` modifier.
- `ClassOrInterface.getASuperInterface()` gets a type from the `implements` clause of a class or from the `extends` clause of an interface.
- `ClassOrInterface.getACallSignature()` gets a call signature of an interface, such as in `{ (arg: string): number }`.
- `ClassOrInterface.getAnIndexSignature()` gets an index signature, such as in `{ [key: string]: number }`.
- `ClassOrInterface.getATypeParameter()` gets a declared type parameter (special case of `TypeParameterized.getATypeParameter()`).

Note that the superclass of a class is an expression, not a type annotation. If the superclass has type arguments, it will be an expression of kind ExpressionWithTypeArguments.

Also see the documentation for classes in the "CodeQL libraries for JavaScript."

To select the type references to a class or an interface, use `getTypeName()`.

Statements

The following are TypeScript-specific statements:

- NamespaceDeclaration: a statement such as `namespace M {}`.
- EnumDeclaration: a statement such as `enum Color { red, green, blue }`.
- TypeAliasDeclaration: a statement such as `type A = number`.
- InterfaceDeclaration: a statement such as `interface Point { x: number; y: number; }`.
- ImportEqualsDeclaration: a statement such as `import fs = require("fs")`.
- ExportAssignDeclaration: a statement such as `export = M`.
- ExportAsNamespaceDeclaration: a statement such as `export as namespace M`.
- ExternalModuleDeclaration: a statement such as `module "foo" {}`.
- GlobalAugmentationDeclaration: a statement such as `global {}`

Expressions

The following are TypeScript-specific expressions:

- ExpressionWithTypeArguments: occurs when the `extends` clause of a class has type arguments, such as in `class C extends D<string>`.
- TypeAssertion: asserts that a value has a given type, such as `x as number` or `<number> x`.
- NonNullAssertion: asserts that a value is not null or undefined, such as `x!`.
- ExternalModuleReference: a `require` call on the right-hand side of an import-assign, such as `import fs = require("fs")`.

Ambient declarations

Type annotations, interfaces, and type aliases are considered ambient AST nodes, as is anything with a `declare` modifier.

The predicate `ASTNode.isAmbient()` can be used to determine if an AST node is ambient.

Ambient nodes are mostly ignored by control flow and data flow analysis. The outermost part of an ambient declaration has a single no-op node in the control flow graph, and it has no internal control flow.

Static type information

Static type information and global name binding is available for projects with "full" TypeScript extraction enabled. This option is enabled by default for projects on LGTM.com and when you create databases with the *CodeQL CLI*.

Basic usage

The Type class represents a static type, such as `number` or `string`. The type of an expression can be obtained with `Expr.getType()`.

Types that refer to a specific named type can be recognized in various ways:

- `type.(TypeReference).hasQualifiedName(name)` holds if the type refers to the given named type.
- `type.(TypeReference).hasUnderlyingType(name)` holds if the type refers to the given named type or a transitive subtype thereof.
- `type.hasUnderlyingType(name)` is like the above, but additionally holds if the reference is wrapped in a union and/or intersection type.

The `hasQualifiedName` and `hasUnderlyingType` predicates have two overloads:

- The single-argument version takes a qualified name relative to the global scope.
- The two-argument version takes the name of a module and qualified name relative to that module.

Example

The following query can be used to find all `toString` calls on a Node.js `Buffer` object:

```
import javascript

from MethodCallExpr call
where call.getReceiver().getType().hasUnderlyingType("Buffer")
  and call.getMethodName() = "toString"
select call
```

Working with types

`Type` entities are not associated with a specific source location. For instance, there can be many uses of the `number` keyword, but there is only one `number` type.

Some important member predicates of `Type` are:

- `Type.getProperty(name)` gets the type of a named property.
- `Type.getMethod(name)` gets the signature of a named method.
- `Type.getSignature(kind,n)` gets the nth overload of a call or constructor signature.
- `Type.getStringIndexType()` gets the type of the string index signature.
- `Type.getNumberIndexType()` gets the type of the number index signature.

A `Type` entity always belongs to exactly one of the following subclasses:

- `TypeReference`: a named type, possibly with type arguments.
- `UnionType`: a union type such as `string | number`.
- `IntersectionType`: an intersection type such as `T & U`.
- `TupleType`: a tuple type such as `[string, number]`.
- `StringType`: the `string` type.
- `NumberType`: the `number` type.
- `AnyType`: the `any` type.
- `NeverType`: the `never` type.
- `VoidType`: the `void` type.
- `NullType`: the `null` type.
- `UndefinedType`: the `undefined` type.
- `ObjectKeywordType`: the `object` type.
- `SymbolType`: a `symbol` or `unique symbol` type.
- `AnonymousInterfaceType`: an anonymous type such as `{x: number}`.
- `TypeVariableType`: a reference to a type variable.
- `ThisType`: the `this` type within a specific type.
- `TypeofType`: the type of a named value, such as `typeof X`.
- `BooleanLiteralType`: the `true` or `false` type.

- `StringLiteralType`: the type of a string constant.
- `NumberLiteralType`: the type of a number constant.

Additionally, `Type` has the following subclasses which overlap partially with those above:

- `BooleanType`: the type `boolean`, internally represented as the union type `true | false`.
- `PromiseType`: a type that describes a promise such as `Promise<T>`.
- `ArrayType`: a type that describes an array object, possibly a tuple type.
  - `PlainArrayType`: a type of form `Array<T>`.
  - `ReadonlyArrayType`: a type of form `ReadonlyArray<T>`.
- `LiteralType`: a boolean, string, or number literal type.
- `NumberLikeType`: the `number` type or a number literal type.
- `StringLikeType`: the `string` type or a string literal type.
- `BooleanLikeType`: the `true`, `false`, or `boolean` type.

Canonical names and named types

`CanonicalName` is a CodeQL class representing a qualified name relative to a root scope, such as a module or the global scope. It typically represents an entity such as a type, namespace, variable, or function. `TypeName` and `Namespace` are subclasses of this class.

Canonical names can be recognized using the `hasQualifiedName` predicate:

- `hasQualifiedName(name)` holds if the qualified name is `name` relative to the global scope.
- `hasQualifiedName(module,name)` holds if the qualified name is `name` relative to the given module name.

For convenience, this predicate is also available on other classes, such as `TypeReference` and `TypeofType`, where it forwards to the underlying canonical name.

Function types

There is no CodeQL class for function types, as any type with a call or construct signature is usable as a function. The type `CallSignatureType` represents such a signature (with or without the `new` keyword).

Signatures can be obtained in several ways:

- `Type.getFunctionSignature(n)` gets the `n`th overloaded function signature.
- `Type.getConstructorSignature(n)` gets the `n`th overloaded constructor signature.
- `Type.getLastFunctionSignature()` gets the last declared function signature.
- `Type.getLastConstructorSignature()` gets the last declared constructor signature.

Some important member predicates of `CallSignatureType` are:

- `CallSignatureType.getParameter(n)` gets the type of the `n`th parameter.
- `CallSignatureType.getParameterName(n)` gets the name of the `n`th parameter.
- `CallSignatureType.getReturnType()` gets the return type.

Note that a signature is not associated with a specific declaration site.

Call resolution

Additional type information is available for invocation expressions:

- `InvokeExpr.getResolvedCallee()` gets the callee as a concrete `Function`.
- `InvokeExpr.getResolvedCalleeName()` get the callee as a canonical name.
- `InvokeExpr.getResolvedSignature()` gets the signature of the invoked function, with overloading resolved and type arguments substituted.

Note that these refer to the call target as determined by the type system. The actual call target may differ at runtime, for instance, if the target is a method that has been overridden in a subclass.

Inheritance and subtyping

The declared supertypes of a named type can be obtained using `TypeName.getABaseTypeName()`.

This operates at the level of type names, hence the specific type arguments used in the inheritance chain are not available. However, these can often be deduced using `Type.getProperty` or `Type.getMethod` which both take inheritance into account.

This only accounts for types explicitly mentioned in the `extends` or `implements` clause of a type. There is no predicate that determines subtyping or assignability between types in general.

The following two predicates can be useful for recognising subtypes of a given type:

- `Type.unfold()` unfolds unions and/or intersection types and get the underlying types, or the type itself if it is not a union or intersection.
- `Type.hasUnderlyingType(name)` holds if the type is a reference to the given named type, possibly after unfolding unions/intersections and following declared supertypes.

Example

The following query can be used to find all classes that are React components, along with the type of their `props` property, which generally coincides with its first type argument:

```
import javascript

from ClassDefinition cls, TypeName name
where name = cls.getTypeName()
  and name.getABaseTypeName+().hasQualifiedName("React.Component")
select cls, name.getType().getProperty("props")
```

Name binding

In TypeScript, names can refer to variables, types, and namespaces, or a combination of these.

These concepts are modeled as distinct entities: Variable, TypeName, and Namespace. For example, the class `C` below introduces both a variable and a type:

```
class C {}
let x = C; // refers to the variable C
let y: C;  // refers to the type C
```

The variable `C` and the type `C` are modeled as distinct entities. One is a Variable, the other is a TypeName.

TypeScript also allows you to import types and namespaces, and give them local names in different scopes. For example, the import below introduces a local type name `B`:

```
import {C as B} from "./foo"
```

The local name `B` is represented as a LocalTypeName named `B`, restricted to just the file containing the import. An import statement can also introduce a Variable and a LocalNamespaceName.

The following table shows the relevant classes for working with each kind of name. The classes are described in more detail below.

Kind	Local alias	Canonical name	Definition	Access
Value	Variable			VarAccess
Type	LocalTypeName	TypeName	TypeDefinition	TypeAccess
Namespace	LocalNamespaceName	Namespace	NamespaceDefinition	NamespaceAccess

Note: `TypeName` and `Namespace` are only populated if the database is generated using full TypeScript extraction. `LocalTypeName` and `LocalNamespaceName` are always populated.

Type names

A TypeName is a qualified name for a type and is not bound to a specific lexical scope. The TypeDefinition class represents an entity that defines a type, namely a class, interface, type alias, enum, or enum member. The relevant predicates for working with type names are:

- `TypeAccess.getTypeName()` gets the qualified name being referenced (if any).
- `TypeDefinition.getTypeName()` gets the qualified name of a class, interface, type alias, enum, or enum member.
- `TypeName.getAnAccess()`, gets an access to a given type.
- `TypeName.getADefinition()`, get a definition of a given type. Note that interfaces can have multiple definitions.

A LocalTypeName behaves like a block-scoped variable, that is, it has an unqualified name and is restricted to a specific scope. The relevant predicates are:

- `LocalTypeAccess.getLocalTypeName()` gets the local name referenced by an unqualified type access.
- `LocalTypeName.getAnAccess()` gets an access to a local type name.
- `LocalTypeName.getADeclaration()` gets a declaration of this name.
- `LocalTypeName.getTypeName()` gets the qualified name to which this name refers.

Examples

Find references that omit type arguments to a generic type.

It is best to use TypeName to resolve through imports and qualified names:

```
import javascript

from TypeDefinition def, TypeAccess access
where access.getTypeName().getADefinition() = def
  and def.(TypeParameterized).hasTypeParameters()
  and not access.hasTypeArguments()
select access, "Type arguments are omitted"
```

See this in the query console on LGTM.com.

Find imported names that are used as both a type and a value:

```
import javascript

from ImportSpecifier spec
where exists (LocalTypeAccess access | access.getLocalTypeName().getADeclaration() =␣
↪spec.getLocal())
  and exists (VarAccess access | access.getVariable().getADeclaration() = spec.
↪getLocal())
select spec, "Used as both variable and type"
```

See this in the query console on LGTM.com.

Namespace names

Namespaces are represented by the classes Namespace and LocalNamespaceName. The NamespaceDefinition class represents a syntactic definition of a namespace, which includes ordinary namespace declarations as well as enum declarations.

Note that these classes deal exclusively with namespaces referenced from inside type annotations, not through expressions.

A Namespace is a qualified name for a namespace, and is not bound to a specific scope. The relevant predicates for working with namespaces are:

- `NamespaceAccess.getNamespace()` gets the namespace being referenced by a namespace access.
- `NamespaceDefinition.getNamespace()` gets the namespace defined by a namespace or enum declaration.
- `Namespace.getAnAccess()` gets an access to a namespace from inside a type.
- `Namespace.getADefinition()` gets a definition of this namespace. Note that namespaces can have multiple definitions.
- `Namespace.getNamespaceMember(name)` gets an inner namespace with a given name.
- `Namespace.getTypeMember(name)` gets a type exported under a given name.
- `Namespace.getAnExportingContainer()` gets a StmtContainer whose exports contribute to this namespace. This can be a the body of a namespace declaration or the top-level of a module. Enums have no exporting containers.

A LocalNamespaceName behaves like a block-scoped variable, that is, it has an unqualified name and is restricted to a specific scope. The relevant predicates are:

- `LocalNamespaceAccess.getLocalNamespaceName()` gets the local name referenced by an identifier.
- `LocalNamespaceName.getAnAccess()` gets an identifier that refers to this local name.
- `LocalNamespaceName.getADeclaration()` gets an identifier that declares this local name.
- `LocalNamespaceName.getNamespace()` gets the namespace to which this name refers.

Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"

5.5.4 Analyzing data flow in JavaScript and TypeScript

This topic describes how data flow analysis is implemented in the CodeQL libraries for JavaScript/TypeScript and includes examples to help you write your own data flow queries.

Overview

The various sections in this article describe how to utilize the libraries for local data flow, global data flow, and taint tracking. As our running example, we will develop a query that identifies command-line arguments that are passed as a file path to the standard Node.js `readFile` function. While this is not a problematic pattern as such, it is typical of the kind of reasoning that is frequently used in security queries.

For a more general introduction to modeling data flow, see "*About data flow analysis*."

Data flow nodes

Both local and global data flow, as well as taint tracking, work on a representation of the program known as the *data flow graph*. Nodes on the data flow flow graph may also correspond to nodes on the abstract syntax tree, but they are not the same. While AST nodes belong to class `ASTNode` and its subclasses, data flow nodes belong to class `DataFlow::Node` and its subclasses:

- `DataFlow::ValueNode`: a *value node*, that is, a data flow node that corresponds either to an expression, or to a declaration of a function, class, TypeScript namespace, or TypeScript enum.
- `DataFlow::SsaDefinitionNode`: a data flow node that corresponds to an SSA variable, that is, a local variable with additional information to reason more precisely about different assignments to the same variable. This kind of data flow node does not correspond to an AST node.
- `DataFlow::PropRef`: a data flow node that corresponds to a read or a write of an object property, for example, in an assignment, in an object literal, or in a destructuring assignment.
- `DataFlow::PropRead`, `DataFlow::PropWrite`: subclasses of `DataFlow::PropRef` that correspond to reads and writes, respectively.

Apart from these fairly general classes, there are some more specialized classes:

- `DataFlow::ParameterNode`: a data flow node that corresponds to a function parameter.
- `DataFlow::InvokeNode`: a data flow node that corresponds to a function call; its subclasses `DataFlow::NewNode` and `DataFlow::CallNode` represent calls with and without `new` respectively, while `DataFlow::MethodCallNode` represents method calls. Note that these classes also model reflective calls using `.call` and `.apply`, which do not correspond to any AST nodes.
- `DataFlow::ThisNode`: a data flow node that corresponds to the value of `this` in a function or top level. This kind of data flow node also does not correspond to an AST node.
- `DataFlow::GlobalVarRefNode`: a data flow node that corresponds to a direct reference to a global variable. This class is rarely used directly, instead you would normally use the predicate `globalVarRef` (introduced below), which also considers indirect references through `window` or global `this`.
- `DataFlow::FunctionNode`, `DataFlow::ObjectLiteralNode`, `DataFlow::ArrayLiteralNode`: a data flow node that corresponds to a function (expression or declaration), an object literal, or an array literal, respectively.
- `DataFlow::ClassNode`: a data flow node corresponding to a class, either defined using an ECMAScript 2015 `class` declaration or an old-style constructor function.
- `DataFlow::ModuleImportNode`: a data flow node corresponding to an ECMAScript 2015 import or an AMD or CommonJS `require` import.

The following predicates are available for mapping from AST nodes and other elements to their corresponding data flow nodes:

- `DataFlow::valueNode(x)`: maps `x`, which must be an expression or a declaration of a function, class, namespace or enum, to its corresponding `DataFlow::ValueNode`.
- `DataFlow::ssaDefinitionNode(ssa)`: maps an SSA definition `ssa` to its corresponding `DataFlow::SsaDefinitionNode`.
- `DataFlow::parameterNode(p)`: maps a function parameter `p` to its corresponding `DataFlow::ParameterNode`.
- `DataFlow::thisNode(s)`: maps a function or top-level `s` to the `DataFlow::ThisNode` representing the value of `this` in `s`.

Class `DataFlow::Node` also has a member predicate `asExpr()` that you can use to map from a `DataFlow::ValueNode` to the expression it corresponds to. Note that this predicate is undefined for other kinds of nodes, and for value nodes that do not correspond to expressions.

There are also some other predicates available for accessing commonly used data flow nodes:

- `DataFlow::globalVarRef(g)`: gets a data flow node corresponding to an access to global variable `g`, either directly or through `window` or (top-level) `this`. For example, you can use `DataFlow::globalVarRef("document")` to find references to the DOM `document` object.
- `DataFlow::moduleMember(p, m)`: gets a data flow node that references a member `m` of a module loaded from path `p`. For example, you can use `DataFlow::moduleMember("fs", "readFile")` to find references to the `fs.readFile` function from the Node.js standard library.

Local data flow

Local data flow is data flow within a single function. Data flow through function calls and returns or through property writes and reads is not modeled.

Local data flow is faster to compute and easier to use than global data flow, but less complete. It is, however, sufficient for many purposes.

To reason about local data flow, use the member predicates `getAPredecessor` and `getASuccessor` on `DataFlow::Node`. For a data flow node `nd`, `nd.getAPredecessor()` returns all data flow nodes from which data flows to `nd` in one local step. Conversely, `nd.getASuccessor()` returns all nodes to which data flows from `nd` in one local step.

To follow one or more steps of local data flow, use the transitive closure operator `+`, and for zero or more steps the reflexive transitive closure operator `*`.

For example, the following query finds all data flow nodes `source` whose value may flow into the first argument of a call to a method with name `readFile`:

```
import javascript

from DataFlow::MethodCallNode readFile, DataFlow::Node source
where
  readFile.getMethodName() = "readFile" and
  source.getASuccessor*() = readFile.getArgument(0)
select source
```

Source nodes

Explicit reasoning about data flow edges can be cumbersome and is rare in practice. Typically, we are not interested in flow originating from arbitrary nodes, but from nodes that in some sense are the "source" of some kind of data, either because they create a new object, such as object literals or functions, or because they represent a point where data enters the local data flow graph, such as parameters or property reads.

The data flow library represents such nodes by the class `DataFlow::SourceNode`, which provides a convenient API to reason about local data flow involving source nodes.

By default, the following kinds of data flow nodes are considered source nodes:

- classes, functions, object and array literals, regular expressions, and JSX elements
- property reads, global variable references and `this` nodes
- function parameters
- function calls
- imports

You can extend the set of source nodes by defining additional subclasses of `DataFlow::SourceNode::Range`.

The `DataFlow::SourceNode` class defines a number of member predicates that can be used to track where data originating from a source node flows, and to find places where properties are accessed or methods are called on them.

For example, the following query finds all references to properties of `process.argv`, the array through which Node.js applications receive their command-line arguments:

```
import javascript

select DataFlow::globalVarRef("process").getAPropertyRead("argv").getAPropertyReference()
```

First, we use `DataFlow::globalVarRef` (mentioned above) to find all references to the global variable `process`. Since global variable references are source nodes, we can then use the predicate `getAPropertyRead` (defined in class `DataFlow::SourceNode`) to find all places where the property `argv` of that global variable is read. The results of this predicate are again source nodes, so we can chain it with a call to `getAPropertyReference`, which is a predicate that finds all references to any property (even references with a computed name) on its base source node.

Note that many predicates on `DataFlow::SourceNode` have source nodes as their result in turn, allowing calls to be chained to concisely express the relationship between several data flow nodes.

Most importantly, predicates like `getAPropertyRead` implicitly follow local data flow, so the above query not only finds direct property references like `process.argv[2]`, but also more indirect ones as in this example:

```
var args = process.argv;
var firstArg = args[2];
```

Analogous to `getAPropertyRead` there is also a predicate `getAPropertyWrite` for identifying property writes.

Another common task is to find calls to a function originating from a source node. For this purpose, `DataFlow::SourceNode` offers predicates `getACall`, `getAnInstantiation` and `getAnInvocation`: the first one only considers invocations without `new`, the second one only invocations with `new`, and the third one considers all invocations.

We can use these predicates in combination with `DataFlow::moduleMember` (mentioned above) to find calls to the function `readFile` imported from the standard Node.js `fs` library:

```
import javascript

select DataFlow::moduleMember("fs", "readFile").getACall()
```

For identifying method calls there is also a predicate `getAMethodCall`, and the slightly more general `getAMemberCall`. The difference between the two is that the former only finds calls that have the syntactic shape of a method call such as `x.m(...)`, while the latter also finds calls where `x.m` is first stored into a local variable `f` and then invoked as `f(...)`.

Finally, the predicate `flowsTo(nd)` holds for any node `nd` into which data originating from the source node may flow. Conversely, `DataFlow::Node` offers a predicate `getALocalSource()` that can be used to find any source node that flows to it.

Putting all of the above together, here is a query that finds (local) data flow from command line arguments to `readFile` calls:

```
import javascript

from DataFlow::SourceNode arg, DataFlow::CallNode call
where
  arg = DataFlow::globalVarRef("process").getAPropertyRead("argv").
↪getAPropertyReference() and
  call = DataFlow::moduleMember("fs", "readFile").getACall() and
  arg.flowsTo(call.getArgument(0))
select arg, call
```

There are two points worth making about the source node API:

1. All data flow tracking is purely local, and in particular flow through global variables is not tracked. If `args` in our `process.argv` example above is a global variable, then the query will not find the reference through `args[2]`.
2. Strings are not source nodes and cannot be tracked using this API. You can, however, use the `mayHaveStringValue` predicate on class `DataFlow::Node` to reason about the possible string values flowing into a data flow node.

For a full description of the `DataFlow::SourceNode` API, see the JavaScript standard library.

Exercises

Exercise 1: Write a query that finds all hard-coded strings used as the `tagName` argument to the `createElement` function from the DOM `document` object, using local data flow. (*Answer*).

Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow. That is, the analysis may report spurious flows that cannot in fact happen. Moreover, global data flow analysis typically requires significantly more time and memory than local analysis.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see *Creating path queries*.

Using global data flow

For performance reasons, it is not generally feasible to compute all global data flow across the entire program. Instead, you can define a data flow *configuration*, which specifies *source* data flow nodes and *sink* data flow nodes ("sources" and "sinks" for short) of interest. The data flow library provides a generic data flow solver that can check whether there is (global) data flow from a source to a sink.

Optionally, configurations may specify extra data flow edges to be added to the data flow graph, and may also specify *barriers*. Barriers are data flow nodes or edges through which data should not be tracked for the purposes of this analysis.

To define a configuration, extend the class `DataFlow::Configuration` as follows:

```
class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) { /* ... */ }

  override predicate isSink(DataFlow::Node sink) { /* ... */ }

  // optional overrides:
  override predicate isBarrier(DataFlow::Node nd) { /* ... */ }
  override predicate isBarrierEdge(DataFlow::Node pred, DataFlow::Node succ) { /* ... */␣
↪}
  override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ) { /*␣
↪... */ }
}
```

The characteristic predicate `MyDataFlowConfiguration()` defines the name of the configuration, so `"MyDataFlowConfiguration"` should be replaced by a suitable name describing your particular analysis configuration.

The data flow analysis is performed using the predicate `hasFlow(source, sink)`:

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Data flow from $@ to $@.", source, source.toString(), sink, sink.
↪toString()
```

Using global taint tracking

Global taint tracking extends global data flow with additional non-value-preserving steps, such as flow through string-manipulating operations. To use it, simply extend `TaintTracking::Configuration` instead of `DataFlow::Configuration`:

```
class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

  override predicate isSource(DataFlow::Node source) { /* ... */ }

  override predicate isSink(DataFlow::Node sink) { /* ... */ }
}
```

Analogous to `isAdditionalFlowStep`, there is a predicate `isAdditionalTaintStep` that you can override to specify custom flow steps to consider in the analysis. Instead of the `isBarrier` and `isBarrierEdge` predicates, the taint tracking configuration includes `isSanitizer` and `isSanitizerEdge` predicates that specify data flow nodes or edges that act as taint sanitizers and hence stop flow from a source to a sink.

Similar to global data flow, the characteristic predicate `MyTaintTrackingConfiguration()` defines the unique name of the configuration, so `"MyTaintTrackingConfiguration"` should be replaced by an appropriate descriptive name.

The taint tracking analysis is again performed using the predicate `hasFlow(source, sink)`.

Examples

The following taint-tracking configuration is a generalization of our example query above, which tracks flow from command-line arguments to `readFile` calls, this time using global taint tracking.

```
import javascript

class CommandLineFileNameConfiguration extends TaintTracking::Configuration {
  CommandLineFileNameConfiguration() { this = "CommandLineFileNameConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    DataFlow::globalVarRef("process").getAPropertyRead("argv").getAPropertyRead() =␣
↪source
  }

  override predicate isSink(DataFlow::Node sink) {
    DataFlow::moduleMember("fs", "readFile").getACall().getArgument(0) = sink
  }
```

(continues on next page)

(continued from previous page)

```
}

from CommandLineFileNameConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

This query will now find flows that involve inter-procedural steps, like in the following example (where the individual steps have been marked with comments `#1` to `#4`):

```
const fs = require('fs'),
      path = require('path');

function readFileHelper(p) {     // #2
  p = path.resolve(p);           // #3
  fs.readFile(p,                 // #4
    'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
  });
}

readFileHelper(process.argv[2]); // #1
```

Note that for step #3 we rely on the taint-tracking library's built-in model of the Node.js `path` library, which adds a taint step from `p` to `path.resolve(p)`. This step is not value preserving, but it preserves taint in the sense that if `p` is user-controlled, then so is `path.resolve(p)` (at least partially).

Other standard taint steps include flow through string-manipulating operations such as concatenation, `JSON.parse` and `JSON.stringify`, array transformations, promise operations, and many more.

Sanitizers

The above JavaScript program allows the user to read any file, including sensitive system files like `/etc/passwd`. If the program may be invoked by an untrusted user, this is undesirable, so we may want to constrain the path. For example, instead of using `path.resolve` we could implement a function `checkPath` that first makes the path absolute and then checks that it starts with the current working directory, aborting the program with an error if it does not. We could then use that function in `readFileHelper` like this:

```
function readFileHelper(p) {
  p = checkPath(p);
  ...
}
```

For the purposes of our above analysis, `checkPath` is a *sanitizer*: its output is always untainted, even if its input is tainted. To model this we can add an override of `isSanitizer` to our taint-tracking configuration like this:

```
class CommandLineFileNameConfiguration extends TaintTracking::Configuration {

  // ...

  override predicate isSanitizer(DataFlow::Node nd) {
    nd.(DataFlow::CallNode).getCalleeName() = "checkPath"
```

(continues on next page)

(continued from previous page)

```
  }
}
```

This says that any call to a function named `checkPath` is to be considered a sanitizer, so any flow through this node is blocked. In particular, the query would no longer flag the flow from `process.argv[2]` to `fs.readFile` in our updated example above.

Sanitizer guards

A perhaps more natural way of implementing the path check in our example would be to have `checkPath` return a Boolean value indicating whether the path is safe to read (instead of returning the path if it is safe and aborting otherwise). We could then use it in `readFileHelper` like this:

```
function readFileHelper(p) {
  if (!checkPath(p))
    return;
  ...
}
```

Note that `checkPath` is now no longer a sanitizer in the sense described above, since the flow from `process.argv[2]` to `fs.readFile` does not go through `checkPath` any more. The flow is, however, *guarded* by `checkPath` in the sense that the expression `checkPath(p)` has to evaluate to `true` (or, more precisely, to a truthy value) in order for the flow to happen.

Such sanitizer guards can be supported by defining a new subclass of `TaintTracking::SanitizerGuardNode` and overriding the predicate `isSanitizerGuard` in the taint-tracking configuration class to add all instances of this class as sanitizer guards to the configuration.

For our above example, we would begin by defining a subclass of `SanitizerGuardNode` that identifies guards of the form `checkPath(...)`:

```
class CheckPathSanitizerGuard extends TaintTracking::SanitizerGuardNode,␣
↪DataFlow::CallNode {
  CheckPathSanitizerGuard() { this.getCalleeName() = "checkPath" }

  override predicate sanitizes(boolean outcome, Expr e) {
    outcome = true and
    e = getArgument(0).asExpr()
  }
}
```

The characteristic predicate of this class checks that the sanitizer guard is a call to a function named `checkPath`. The overriding definition of `sanitizes` says such a call sanitizes its first argument (that is, `getArgument(0)`) if it evaluates to `true` (or rather, a truthy value).

Now we can override `isSanitizerGuard` to add these sanitizer guards to our configuration:

```
class CommandLineFileNameConfiguration extends TaintTracking::Configuration {

  // ...

  override predicate isSanitizerGuard(TaintTracking::SanitizerGuardNode nd) {
    nd instanceof CheckPathSanitizerGuard
```

(continues on next page)

(continued from previous page)

```
  }
}
```

With these two additions, the query recognizes the `checkPath(p)` check as sanitizing `p` after the `return`, since execution can only reach there if `checkPath(p)` evaluates to a truthy value. Consequently, there is no longer a path from `process.argv[2]` to `readFile`.

Additional taint steps

Sometimes the default data flow and taint steps provided by `DataFlow::Configuration` and `TaintTracking::Configuration` are not sufficient and we need to add additional flow or taint steps to our configuration to make it find the expected flow. For example, this can happen because the analyzed program uses a function from an external library whose source code is not available to the analysis, or because it uses a function that is too difficult to analyze.

In the context of our running example, assume that the JavaScript program we are analyzing uses a (fictitious) npm package `resolve-symlinks` to resolve any symlinks in the path `p` before passing it to `readFile`:

```
const resolveSymlinks = require('resolve-symlinks');

function readFileHelper(p) {
  p = resolveSymlinks(p);
  fs.readFile(p,
  ...
}
```

Resolving symlinks does not make an unsafe path any safer, so we would still like our query to flag this, but since the standard library does not have a model of `resolve-symlinks` it will no longer return any results.

We can fix this quite easily by adding an overriding definition of the `isAdditionalTaintStep` predicate to our configuration, introducing an additional taint step from the first argument of `resolveSymlinks` to its result:

```
class CommandLineFileNameConfiguration extends TaintTracking::Configuration {

  // ...

  override predicate isAdditionalTaintStep(DataFlow::Node pred, DataFlow::Node succ) {
    exists(DataFlow::CallNode c |
      c = DataFlow::moduleImport("resolve-symlinks").getACall() and
      pred = c.getArgument(0) and
      succ = c
    )
  }
}
```

We might even consider adding this as a default taint step to be used by all taint-tracking configurations. In order to do this, we need to wrap it in a new subclass of `TaintTracking::SharedTaintStep` like this:

```
class StepThroughResolveSymlinks extends TaintTracking::SharedTaintStep {
  override predicate step(DataFlow::Node pred, DataFlow::Node succ) {
    exists(DataFlow::CallNode c |
      c = DataFlow::moduleImport("resolve-symlinks").getACall() and
      pred = c.getArgument(0) and
```

(continues on next page)

(continued from previous page)

```
      succ = c
    )
  }
}
```

If we add this definition to the standard library, it will be picked up by all taint-tracking configurations. Obviously, one has to be careful when adding such new additional taint steps to ensure that they really make sense for *all* configurations.

Analogous to `TaintTracking::SharedTaintStep`, there is also a class `DataFlow::SharedFlowStep` that can be extended to add extra steps to all data-flow configurations, and hence also to all taint-tracking configurations.

Exercises

Exercise 2: Write a query that finds all hard-coded strings used as the `tagName` argument to the `createElement` function from the DOM `document` object, using global data flow. (*Answer*).

Exercise 3: Write a class which represents flow sources from the array elements of the result of a call, for example the expression `myObject.myMethod(myArgument)[myIndex]`. Hint: array indices are properties with numeric names; you can use regular expression matching to check this. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query which finds all global data flows from array elements of the result of a call to the `tagName` argument to the `createElement` function. (*Answer*)

Answers

Exercise 1

```
import javascript

from DataFlow::CallNode create, string name
where
  create = DataFlow::globalVarRef("document").getAMethodCall("createElement") and
  create.getArgument(0).mayHaveStringValue(name)
select name
```

Exercise 2

```
import javascript

class HardCodedTagNameConfiguration extends DataFlow::Configuration {
  HardCodedTagNameConfiguration() { this = "HardCodedTagNameConfiguration" }

  override predicate isSource(DataFlow::Node source) { source.asExpr() instanceof␣
↪ConstantString }

  override predicate isSink(DataFlow::Node sink) {
    sink = DataFlow::globalVarRef("document").getAMethodCall("createElement").
↪getArgument(0)
  }
```

(continues on next page)

(continued from previous page)

```
}

from HardCodedTagNameConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

Exercise 3

```
import javascript

class ArrayEntryCallResult extends DataFlow::Node {
  ArrayEntryCallResult() {
    exists(DataFlow::CallNode call, string index |
      this = call.getAPropertyRead(index) and
      index.regexpMatch("\\d+")
    )
  }
}
```

Exercise 4

```
import javascript

class ArrayEntryCallResult extends DataFlow::Node {
  ArrayEntryCallResult() {
    exists(DataFlow::CallNode call, string index |
      this = call.getAPropertyRead(index) and
      index.regexpMatch("\\d+")
    )
  }
}

class HardCodedTagNameConfiguration extends DataFlow::Configuration {
  HardCodedTagNameConfiguration() { this = "HardCodedTagNameConfiguration" }

  override predicate isSource(DataFlow::Node source) { source instanceof␣
↪ArrayEntryCallResult }

  override predicate isSink(DataFlow::Node sink) {
    sink = DataFlow::globalVarRef("document").getAMethodCall("createElement").
↪getArgument(0)
  }
}

from HardCodedTagNameConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- "*QL language reference*"
- "*CodeQL tools*"

5.5.5 Using flow labels for precise data flow analysis

You can associate flow labels with each value tracked by the flow analysis to determine whether the flow contains potential vulnerabilities.

Overview

You can use basic inter-procedural data-flow analysis and taint tracking as described in "*Analyzing data flow in JavaScript and TypeScript*" to check whether there is a path in the data-flow graph from some source node to a sink node that does not pass through any sanitizer nodes. Another way of thinking about this is that it statically models the flow of data through the program, and associates a flag with every data value telling us whether it might have come from a source node.

In some cases, you may want to track more detailed information about data values. This can be done by associating flow labels with data values, as shown in this tutorial. We will first discuss the general idea behind flow labels and then show how to use them in practice. Finally, we will give an overview of the API involved and provide some pointers to standard queries that use flow labels.

Limitations of basic data-flow analysis

In many applications we are interested in tracking more than just the reachability information provided by inter-procedural data flow analysis.

For example, when tracking object values that originate from untrusted input, we might want to remember whether the entire object is tainted or whether only part of it is tainted. The former happens, for example, when parsing a user-controlled string as JSON, meaning that the entire resulting object is tainted. A typical example of the latter is assigning a tainted value to a property of an object, which only taints that property but not the rest of the object.

While reading a property of a completely tainted object yields a tainted value, reading a property of a partially tainted object does not. On the other hand, JSON-encoding even a partially tainted object and including it in an HTML document is not safe.

Another example where more fine-grained information about tainted values is needed is for tracking partial sanitization. For example, before interpreting a user-controlled string as a file-system path, we generally want to make sure that it is neither an absolute path (which could refer to any file on the file system) nor a relative path containing `..` components (which still could refer to any file). Usually, checking both of these properties would involve two separate checks. Both checks taken together should count as a sanitizer, but each individual check is not by itself enough to make the string safe for use as a path. To handle this case precisely, we want to associate two bits of information with each tainted value, namely whether it may be absolute, and whether it may contain `..` components. Untrusted user input has both bits set initially, individual checks turn off individual bits, and if a value that has at least one bit set is interpreted as a path, a potential vulnerability is flagged.

Using flow labels

You can handle these cases and others like them by associating a set of *flow labels* (sometimes also referred to as *taint kinds*) with each value being tracked by the analysis. Value-preserving data-flow steps (such as flow steps from writes to a variable to its reads) preserve the set of flow labels, but other steps may add or remove flow labels. Sanitizers, in particular, are simply flow steps that remove some or all flow labels. The initial set of flow labels for a value is determined by the source node that gives rise to it. Similarly, sink nodes can specify that an incoming value needs to have a certain flow label (or one of a set of flow labels) in order for the flow to be flagged as a potential vulnerability.

Example

As an example of using flow labels, we will show how to write a query that flags property accesses on JSON values that come from user-controlled input where we have not checked whether the value is `null`, so that the property access may cause a runtime exception.

For example, we would like to flag this code:

```
var data = JSON.parse(str);
if (data.length > 0) {  // problematic: `data` may be `null`
  ...
}
```

This code, on the other hand, should not be flagged:

```
var data = JSON.parse(str);
if (data && data.length > 0) { // unproblematic: `data` is first checked for nullness
  ...
}
```

We will first try to write a query to find this kind of problem without flow labels, and use the difficulties we encounter as a motivation for bringing flow labels into play, which will make the query much easier to implement.

To get started, let's write a query that simply flags any flow from `JSON.parse` into the base of a property access:

```
import javascript

class JsonTrackingConfig extends DataFlow::Configuration {
  JsonTrackingConfig() { this = "JsonTrackingConfig" }

  override predicate isSource(DataFlow::Node nd) {
    exists(JsonParserCall jpc |
      nd = jpc.getOutput()
    )
  }

  override predicate isSink(DataFlow::Node nd) {
    exists(DataFlow::PropRef pr |
      nd = pr.getBase()
    )
  }
}

from JsonTrackingConfig cfg, DataFlow::Node source, DataFlow::Node sink
```

(continues on next page)

(continued from previous page)

```
where cfg.hasFlow(source, sink)
select sink, "Property access on JSON value originating $@.", source, "here"
```

Note that we use the `JsonParserCall` class from the standard library to model various JSON parsers, including the standard `JSON.parse` API as well as a number of popular npm packages.

Of course, as written this query flags both the good and the bad example above, since we have not introduced any sanitizers yet.

There are many ways of checking for nullness directly or indirectly. Since this is not the main focus of this tutorial, we will only show how to model one specific case: if some variable `v` is known to be truthy, it cannot be `null`. This kind of condition is easily expressed using a `BarrierGuardNode` (or its counterpart `SanitizerGuardNode` for taint-tracking configurations). A barrier guard node is a data-flow node `b` that blocks flow through some other node `nd`, provided that some condition checked at `b` is known to hold, that is, evaluate to a truthy value.

In our case, the barrier guard node is a use of some variable `v`, and the condition is that use itself: it blocks flow through any use of `v` where the guarding use is known to evaluate to a truthy value. In our second example above, the use of `data` on the left-hand side of the `&&` is a barrier guard blocking flow through the use of `data` on the right-hand side of the `&&`. At this point we know that `data` has evaluated to a truthy value, so it cannot be `null` anymore.

Implementing this additional condition is easy. We implement a subclass of `DataFlow::BarrierGuardNode`:

```
class TruthinessCheck extends DataFlow::BarrierGuardNode, DataFlow::ValueNode {
  SsaVariable v;

  TruthinessCheck() {
    astNode = v.getAUse()
  }

  override predicate blocks(boolean outcome, Expr e) {
    outcome = true and
    e = astNode
  }
}
```

and then use it to override predicate `isBarrierGuard` in our configuration class:

```
override predicate isBarrierGuard(DataFlow::BarrierGuardNode guard) {
  guard instanceof TruthinessCheck
}
```

With this change, we now flag the problematic case and don't flag the unproblematic case above.

However, as it stands our analysis has many false negatives: if we read a property of a JSON object, our analysis will not continue tracking it, so property accesses on the resulting value will not be checked for null-guardedness:

```
var root = JSON.parse(str);
if (root) {
  var payload = root.data;   // unproblematic: `root` cannot be `null` here
  if (payload.length > 0) {  // problematic: `payload` may be `null` here
    ...
  }
}
```

We could try to remedy the situation by overriding `isAdditionalFlowStep` in our configuration class to track values through property reads:

```
override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ) {
  succ.(DataFlow::PropRead).getBase() = pred
}
```

But this does not actually allow us to flag the problem above as once we have checked `root` for truthiness, all further uses are considered to be sanitized. In particular, the reference to `root` in `root.data` is sanitized, so no flow tracking through the property read happens.

The problem is, of course, that our sanitizer sanitizes too much. It should not stop flow altogether, it should simply record the fact that `root` itself is known to be non-null. Any property read from `root`, on the other hand, may well be null and needs to be checked separately.

We can achieve this by introducing two different flow labels, `json` and `maybe-null`. The former means that the value we are dealing with comes from a JSON object, the latter that it may be `null`. The result of any call to `JSON.parse` has both labels. A property read from a value with label `json` also has both labels. Checking truthiness removes the `maybe-null` label. Accessing a property on a value that has the `maybe-null` label should be flagged.

To implement this, we start by defining two new subclasses of the class `DataFlow::FlowLabel`:

```
class JsonLabel extends DataFlow::FlowLabel {
  JsonLabel() {
    this = "json"
  }
}

class MaybeNullLabel extends DataFlow::FlowLabel {
  MaybeNullLabel() {
    this = "maybe-null"
  }
}
```

Then we extend our `isSource` predicate from above to track flow labels by overriding the two-argument version instead of the one-argument version:

```
override predicate isSource(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
  exists(JsonParserCall jpc |
    nd = jpc.getOutput() and
    (lbl instanceof JsonLabel or lbl instanceof MaybeNullLabel)
  )
}
```

Similarly, we make `isSink` flow-label aware and require the base of the property read to have the `maybe-null` label:

```
override predicate isSink(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
  exists(DataFlow::PropRef pr |
    nd = pr.getBase() and
    lbl instanceof MaybeNullLabel
  )
}
```

Our overriding definition of `isAdditionalFlowStep` now needs to specify two flow labels, a predecessor label `predlbl` and a successor label `succlbl`. In addition to specifying flow from the predecessor node `pred` to the successor node `succ`, it requires that `pred` has label `predlbl`, and adds label `succlbl` to `succ`. In our case, we use this to add both the `json` label and the `maybe-null` label to any property read from a value labeled with `json` (no matter whether it has the `maybe-null` label):

```
override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ,
                                        DataFlow::FlowLabel predlbl, DataFlow::FlowLabel succlbl) {
  succ.(DataFlow::PropRead).getBase() = pred and
  predlbl instanceof JsonLabel and
  (succlbl instanceof JsonLabel or succlbl instanceof MaybeNullLabel)
}
```

Finally, we turn `TruthinessCheck` from a `BarrierGuardNode` into a `LabeledBarrierGuardNode`, specifying that it only removes the `maybe-null` label (but not the `json` label) from the sanitized value:

```
class TruthinessCheck extends DataFlow::LabeledBarrierGuardNode, DataFlow::ValueNode {
  ...

  override predicate blocks(boolean outcome, Expr e, DataFlow::FlowLabel lbl) {
    outcome = true and
    e = astNode and
    lbl instanceof MaybeNullLabel
  }
}
```

Here is the final query, expressed as a *path query* so we can examine paths from sources to sinks step by step in the UI:

```
/** @kind path-problem */

import javascript
import DataFlow::PathGraph

class JsonLabel extends DataFlow::FlowLabel {
  JsonLabel() {
    this = "json"
  }
}

class MaybeNullLabel extends DataFlow::FlowLabel {
  MaybeNullLabel() {
    this = "maybe-null"
  }
}

class TruthinessCheck extends DataFlow::LabeledBarrierGuardNode, DataFlow::ValueNode {
  SsaVariable v;

  TruthinessCheck() {
    astNode = v.getAUse()
  }

  override predicate blocks(boolean outcome, Expr e, DataFlow::FlowLabel lbl) {
    outcome = true and
    e = astNode and
    lbl instanceof MaybeNullLabel
  }
}
```

(continues on next page)

(continued from previous page)

```
class JsonTrackingConfig extends DataFlow::Configuration {
  JsonTrackingConfig() { this = "JsonTrackingConfig" }

  override predicate isSource(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
    exists(JsonParserCall jpc |
      nd = jpc.getOutput() and
      (lbl instanceof JsonLabel or lbl instanceof MaybeNullLabel)
    )
  }

  override predicate isSink(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
    exists(DataFlow::PropRef pr |
      nd = pr.getBase() and
      lbl instanceof MaybeNullLabel
    )
  }

  override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ,
                                          DataFlow::FlowLabel predlbl, DataFlow::FlowLabel succlbl) {
    succ.(DataFlow::PropRead).getBase() = pred and
    predlbl instanceof JsonLabel and
    (succlbl instanceof JsonLabel or succlbl instanceof MaybeNullLabel)
  }

  override predicate isBarrierGuard(DataFlow::BarrierGuardNode guard) {
    guard instanceof TruthinessCheck
  }
}

from JsonTrackingConfig cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink, source, sink, "Property access on JSON value originating $@.", source, "here
↪"
```

Here is a run of this query on the plexus-interop project on LGTM.com. Many of the 19 results are false positives since we currently do not model many ways in which a value can be checked for nullness. In particular, after a property reference `x.p` we implicitly know that `x` cannot be null anymore, since otherwise the reference would have thrown an exception. Modeling this would allow us to get rid of most of the false positives, but is beyond the scope of this tutorial.

API

Plain data-flow configurations implicitly use a single flow label "data", which indicates that a data value originated from a source. You can use the predicate `DataFlow::FlowLabel::data()`, which returns this flow label, as a symbolic name for it.

Taint-tracking configurations add a second flow label "taint" (`DataFlow::FlowLabel::taint()`), which is similar to "data", but includes values that have passed through non-value preserving steps such as string operations.

Each of the three member predicates `isSource`, `isSink` and `isAdditionalFlowStep`/`isAdditionalTaintStep` has one version that uses the default flow labels, and one version that allows specifying custom flow labels through additional arguments.

For `isSource`, there is one additional argument specifying which flow label(s) should be associated with values originating from this source. If multiple flow labels are specified, each value is associated with *all* of them.

For `isSink`, the additional argument specifies which flow label(s) a value that flows into this source may be associated with. If multiple flow labels are specified, then any value that is associated with *at least one* of them will be considered by the configuration.

For `isAdditionalFlowStep` there are two additional arguments `predlbl` and `succlbl`, which allow flow steps to act as flow label transformers. If a value associated with `predlbl` arrives at the start node of the additional step, it is propagated to the end node and associated with `succlbl`. Of course, `predlbl` and `succlbl` may be the same, indicating that the flow step preserves this label. There can also be multiple values of `succlbl` for a single `predlbl` or vice versa.

Note that if you do not restrict `succlbl` then it will be allowed to range over all flow labels. This may cause labels that were previously blocked on a path to reappear, which is not usually what you want.

The flow label-aware version of `isBarrier` is called `isLabeledBarrier`: unlike `isBarrier`, which prevents any flow past the given node, it only blocks flow of values associated with one of the specified flow labels.

Standard queries using flow labels

Some of our standard security queries use flow labels. You can look at their implementation to get a feeling for how to use flow labels in practice.

In particular, both of the examples mentioned in the section on limitations of basic data flow above are from standard security queries that use flow labels. The Prototype pollution query uses two flow labels to distinguish completely tainted objects from partially tainted objects. The Uncontrolled data used in path expression query uses four flow labels to track whether a user-controlled string may be an absolute path and whether it may contain `..` components.

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"

5.5.6 Specifying additional remote flow sources for JavaScript

You can model potential sources of untrusted user input in your code without making changes to the CodeQL standard library by specifying extra remote flow sources in an external file.

> Note
>
> Specifying remote flow sources in external files is currently in beta and subject to change.

As mentioned in the *Data flow cheat sheet for JavaScript*, the CodeQL libraries for JavaScript provide a class RemoteFlowSource to represent sources of untrusted user input, sometimes also referred to as remote flow sources.

To model a new source of untrusted input, such as a previously unmodelled library API, you can define a subclass of `RemoteFlowSource` that covers all uses of that API. All standard analyses will then automatically pick up this new source of remote flow.

However, this approach requires writing QL code and adding it to the standard library, which is not always easy to do. Instead, you can also add a JSON file describing custom sources of untrusted input to your code base and have it picked up without needing to modify the standard library. This JSON file can be hand-written or generated by another tool.

The custom remote flow sources are only available to the code base containing the JSON file. This means that you need to copy the JSON file into each code base that requires the customizations.

Specification format

The JSON file must be called `codeql-javascript-remote-flow-sources.json` and can be located anywhere in your code base. It should consist of a single JSON object. The property names of this object are interpreted as *source types*. The values they map to should be arrays of strings. Each string should be of the form `window.props`, where `props` is a sequence of one or more property names separated by dots. This notation specifies that any value reachable from the global window object by this sequence of property names should be considered as untrusted user input of the associated source type.

Example

Consider the following specification:

```
{
  "user input": [ "window.user.name", "window.user.address", "window.dob" ]
}
```

It declares that the contents of global variable `dob`, as well as the contents of properties `name` and `address` of global variable `user`, should be considered as remote flow sources, with source type "user input".

5.5.7 Using type tracking for API modeling

You can track data through an API by creating a model using the CodeQL type-tracking library for JavaScript.

Overview

The type-tracking library makes it possible to track values through properties and function calls, usually to recognize method calls and properties accessed on a specific type of object.

This is an advanced topic and is intended for readers already familiar with the SourceNode class as well as taint tracking. For TypeScript analysis also consider reading about static type information first.

The problem of recognizing method calls

We'll start with a simple model of the Firebase API and gradually build on it to use type tracking. Knowledge of Firebase is not required.

Suppose we wish to find places where data is written to a Firebase database, as in the following example:

```
var ref = firebase.database().ref("forecast");
ref.set("Rain"); // <-- find this call
```

A simple way to do this is just to find all method calls named "`set`":

```
import javascript
import DataFlow

MethodCallNode firebaseSetterCall() {
```

(continues on next page)

(continued from previous page)

```
  result.getMethodName() = "set"
}
```

The obvious problem with this is that it finds calls to *all* methods named `set`, many of which are unrelated to Firebase.

Another approach is to use local data flow to match the chain of calls that led to this call:

```
MethodCallNode firebaseSetterCall() {
  result = globalVarRef("firebase")
      .getAMethodCall("database")
      .getAMethodCall("ref")
      .getAMethodCall("set")
}
```

This will find the `set` call from the example, but no spurious, unrelated `set` method calls. We can split it up so each step is its own predicate:

```
SourceNode firebase() {
  result = globalVarRef("firebase")
}

SourceNode firebaseDatabase() {
  result = firebase().getAMethodCall("database")
}

SourceNode firebaseRef() {
  result = firebaseDatabase().getAMethodCall("ref");
}

MethodCallNode firebaseSetterCall() {
  result = firebaseRef().getAMethodCall("set")
}
```

The code above is equivalent to the previous version, but it's easier to tinker with the individual steps.

The downside is that the model relies entirely on local data flow, which means it won't look through properties and function calls. For instance, `firebaseSetterCall()` fails to find anything in this example:

```
function getDatabase() {
  return firebase.database();
}
var ref = getDatabase().ref("forecast");
ref.set("Rain");
```

Notice that the predicate `firebaseDatabase()` still finds the call to `firebase.database()`, but not the `getDatabase()` call. This means `firebaseRef()` has no result, which in turn means `firebaseSetterCall()` has no result.

As a simple remedy, let's try to make `firebaseDatabase()` recognize the `getDatabase()` call:

```
SourceNode firebaseDatabase() {
  result = firebase().getAMethodCall("database")
  or
  result.(CallNode).getACallee().getAReturn().getALocalSource() = firebaseDatabase()
}
```

The second clause ensures `firebaseDatabase()` finds not only `firebase.database()` calls, but also calls to functions that *return* `firebase.database()`, such as `getDatabase()` seen above. It's recursive, so it handles flow out of any number of nested function calls.

However, it still only tracks *out* of functions, not *into* functions through parameters, nor through properties. Instead of adding these steps by hand, we'll use type tracking.

Type tracking in general

Type tracking is a generalization of the above pattern, where a predicate matches the value to track, and has a recursive clause that tracks the flow of that value. But instead of us having to deal with function calls/returns and property reads/writes, all of these steps are included in a single predicate, SourceNode.track, to be used with the companion class TypeTracker.

Predicates that use type tracking usually conform to the following general pattern, which we explain below:

```
SourceNode myType(TypeTracker t) {
  t.start() and
  result = /* SourceNode to track */
  or
  exists(TypeTracker t2 |
    result = myType(t2).track(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeTracker::end())
}
```

We'll apply the pattern to our example model and use that to explain what's going on.

Tracking the database instance

Applying the above pattern to the `firebaseDatabase()` predicate we get the following:

```
SourceNode firebaseDatabase(TypeTracker t) {
  t.start() and
  result = firebase().getAMethodCall("database")
  or
  exists(TypeTracker t2 |
    result = firebaseDatabase(t2).track(t2, t)
  )
}

SourceNode firebaseDatabase() {
  result = firebaseDatabase(TypeTracker::end())
}
```

There are now two predicates named `firebaseDatabase`. The one with the `TypeTracker` parameter is the one actually doing the global data flow tracking – the other predicate exposes the result in a convenient way.

The new `TypeTracker t` parameter is a summary of the steps needed to track the value of interest to the resulting data flow node.

In the base case, when matching `firebase.database()`, we use `t.start()` to indicate that no steps were needed, that is, this is the starting point of type tracking:

```
t.start() and
result = firebase().getAMethodCall("database")
```

In the recursive case, we apply the `track` predicate on a previously-found Firebase database node, such as `firebase.database()`. The `track` predicate maps this to a successor of that node, such as `getDatabase()`, and binds `t` to the continuation of `t2` with this extra step included:

```
exists(TypeTracker t2 |
  result = firebaseDatabase(t2).track(t2, t)
)
```

To understand the role of `t` here, note that type tracking can step *into* a property, which means the data flow node returned from `track` is not necessarily a Firebase database instance, it could be an object *containing* a Firebase database in one of its properties.

For example, in the program below, the `firebaseDatabase(t)` predicate includes the `obj` node in its result, but with `t` recording the fact that the actual value being tracked is inside the `DB` property:

```
let obj = { DB: firebase.database() };
let db = obj.DB;
```

This brings us to the last predicate. This uses `TypeTracker::end()` to filter out the paths where the Firebase database instance ended up inside a property of another object, so it includes `db` but not `obj`:

```
SourceNode firebaseDatabase() {
  result = firebaseDatabase(TypeTracker::end())
}
```

Here's see an example of what this can handle now:

```
class Firebase {
  constructor() {
    this.db = firebase.database();
  }

  getDatabase() { return this.db; }

  setForecast(value) {
    this.getDatabase().ref("forecast").set(value); // found by firebaseSetterCall()
  }
}
```

Tracking in the whole model

We applied this pattern to `firebaseDatabase()` in the previous section, and it can just as easily apply to the other predicates. For reference, here's our simple Firebase model with type tracking on every predicate:

```
SourceNode firebase(TypeTracker t) {
  t.start() and
  result = globalVarRef("firebase")
  or
  exists(TypeTracker t2 |
    result = firebase(t2).track(t2, t)
  )
}

SourceNode firebase() {
  result = firebase(TypeTracker::end())
}

SourceNode firebaseDatabase(TypeTracker t) {
  t.start() and
  result = firebase().getAMethodCall("database")
  or
  exists(TypeTracker t2 |
    result = firebaseDatabase(t2).track(t2, t)
  )
}

SourceNode firebaseDatabase() {
  result = firebaseDatabase(TypeTracker::end())
}

SourceNode firebaseRef(TypeTracker t) {
  t.start() and
  result = firebaseDatabase().getAMethodCall("ref")
  or
  exists(TypeTracker t2 |
    result = firebaseRef(t2).track(t2, t)
  )
}

SourceNode firebaseRef() {
  result = firebaseRef(TypeTracker::end())
}

MethodCallNode firebaseSetterCall() {
  result = firebaseRef().getAMethodCall("set")
}
```

Here is a run of an example query using the model to find *set* calls on one of the Firebase sample projects. It's been modified slightly to handle a bit more of the API, which is beyond the scope of this tutorial.

Tracking associated data

By adding extra parameters to the type-tracking predicate, we can carry along extra bits of information about the result.

For example, here's a type-tracking version of `firebaseRef()`, which tracks the string that was passed to the `ref` call:

```
SourceNode firebaseRef(string name, TypeTracker t) {
  t.start() and
  exists(CallNode call |
    call = firebaseDatabase().getAMethodCall("ref") and
    name = call.getArgument(0).getStringValue() and
    result = call
  )
  or
  exists(TypeTracker t2 |
    result = firebaseRef(name, t2).track(t2, t)
  )
}

SourceNode firebaseRef(string name) {
  result = firebaseRef(name, TypeTracker::end())
}

MethodCallNode firebaseSetterCall(string refName) {
  result = firebaseRef(refName).getAMethodCall("set")
}
```

So now we can use `firebaseSetterCall("forecast")` to find assignments to the forecast.

Back-tracking callbacks

The type-tracking predicates we've seen above all use *forward* tracking. That is, they all start with some value of interest and ask "where does this flow?".

Sometimes it's more useful to work backwards, starting at the desired end-point and asking "what flows to here?".

As a motivating example, we'll extend our model to look for places where we *read* a value from the database, as opposed to writing it. Reading is an asynchronous operation and the result is obtained through a callback, for example:

```
function fetchForecast(callback) {
  firebase.database().ref("forecast").once("value", callback);
}

function updateReminders() {
  fetchForecast((snapshot) => {
    let forecast = snapshot.val(); // <-- find this call
    addReminder(forecast === "Rain" ? "Umbrella" : "Sunscreen");
  })
}
```

The actual forecast is obtained by the call to `snapshot.val()`.

Looking for all method calls named `val` will in practice find many unrelated methods, so we'll use type tracking again to take the receiver type into account.

The receiver `snapshot` is a parameter to a callback function, which ultimately escapes into the `once()` call. We'll extend our model from above to use back-tracking to find all functions that flow into the `once()` call. Backwards type tracking is not too different from forwards type tracking. The differences are:

- The `TypeTracker` parameter instead has type `TypeBackTracker`.
- The call to `.track()` is instead a call to `.backtrack()`.
- To ensure the initial value is a source node, a call to `getALocalSource()` is usually required.

```
SourceNode firebaseSnapshotCallback(string refName, TypeBackTracker t) {
  t.start() and
  result = firebaseRef(refName).getAMethodCall("once").getArgument(1).getALocalSource()
  or
  exists(TypeBackTracker t2 |
    result = firebaseSnapshotCallback(refName, t2).backtrack(t2, t)
  )
}

FunctionNode firebaseSnapshotCallback(string refName) {
  result = firebaseSnapshotCallback(refName, TypeBackTracker::end())
}
```

Now, `firebaseSnapshotCallback("forecast")` finds the function being passed to `fetchForecast`. Based on that we can track the `snapshot` value and find the `val()` call itself:

```
SourceNode firebaseSnapshot(string refName, TypeTracker t) {
  t.start() and
  result = firebaseSnapshotCallback(refName).getParameter(0)
  or
  exists(TypeTracker t2 |
    result = firebaseSnapshot(refName, t2).track(t2, t)
  )
}

SourceNode firebaseSnapshot(string refName) {
  result = firebaseSnapshot(refName, TypeTracker::end())
}

MethodCallNode firebaseDatabaseRead(string refName) {
  result = firebaseSnapshot(refName).getAMethodCall("val")
}
```

With this addition, `firebaseDatabaseRead("forecast")` finds the call to `snapshot.val()` that contains the value of the forecast.

Here is a run of an example query using the model to find *val* calls.

Summary

We have covered how to use the type-tracking library. To recap, use this template to define forward type-tracking predicates:

```
SourceNode myType(TypeTracker t) {
  t.start() and
  result = /* SourceNode to track */
  or
  exists(TypeTracker t2 |
    result = myType(t2).track(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeTracker::end())
}
```

Use this template to define backward type-tracking predicates:

```
SourceNode myType(TypeBackTracker t) {
  t.start() and
  result = (/* argument to track */).getALocalSource()
  or
  exists(TypeBackTracker t2 |
    result = myType(t2).backtrack(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeBackTracker::end())
}
```

Note that these predicates all return `SourceNode`, so attempts to track a non-source node, such as an identifier or string literal, will not work. If this becomes an issue, see TypeTracker.smallstep.

Also note that the predicates taking a `TypeTracker` or `TypeBackTracker` can often be made `private`, as they are typically only used as an intermediate result to compute the other predicate.

Limitations

As mentioned, type tracking will track values in and out of function calls and properties, but only within some limits.

For example, type tracking does not always track *through* functions. That is, if a value flows into a parameter and back out of the return value, it might not be tracked back out to the call site again. Here's an example that the model from this tutorial won't find:

```
function wrapDB(database) {
  return { db: database }
}
let wrapper = wrapDB(firebase.database())
wrapper.db.ref("forecast"); // <-- not found
```

This is an example of where data-flow configurations are more powerful.

When to use type tracking

Type tracking and data-flow configurations are different solutions to the same problem, each with their own tradeoffs.

Type tracking can be used in any number of predicates, which may depend on each other in fairly unrestricted ways. The result of one predicate may be the starting point for another. Type-tracking predicates may be mutually recursive. Type-tracking predicates can have any number of extra parameters, making it possible, but optional, to construct source/sink pairs. Omitting source/sink pairs can be useful when there is a huge number of sources and sinks.

Data-flow configurations have more restricted dependencies but are more powerful in other ways. For performance reasons, the sources, sinks, and steps of a configuration should not depend on whether a flow path has been found using that configuration or any other configuration. In that sense, the sources, sinks, and steps must be configured "up front" and can't be discovered on-the-fly. The upside is that they track flow through functions and callbacks in some ways that type tracking doesn't, which is particularly important for security queries. Also, path queries can only be defined using data-flow configurations.

Prefer type tracking when:

- Disambiguating generically named methods or properties.
- Making reusable library components to be shared between queries.
- The set of source/sink pairs is too large to compute or has insufficient information.
- The information is needed as input to a data-flow configuration.

Prefer data-flow configurations when:

- Tracking user-controlled data – use taint tracking.
- Differentiating between different kinds of user-controlled data – see "*Using flow labels for precise data flow analysis*."
- Tracking transformations of a value through generic utility functions.
- Tracking values through string manipulation.
- Generating a path from source to sink – see "*Creating path queries*."

Lastly, depending on the code base being analyzed, some alternatives to consider are:

- Using static type information, if analyzing TypeScript code.
- Relying on local data flow.
- Relying on syntactic heuristics such as the name of a method, property, or variable.

Type tracking in the standard libraries

Type tracking is used in a few places in the standard libraries:

- The DOM predicates, documentRef, locationRef, and domValueRef, are implemented with type tracking.
- The HTTP server models, such as Express, use type tracking to track the installation of router handler functions.
- The Firebase and Socket.io models use type tracking to track objects coming from their respective APIs.

Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"

5.5.8 Abstract syntax tree classes for working with JavaScript and TypeScript programs

CodeQL has a large selection of classes for representing the abstract syntax tree of JavaScript and TypeScript programs.

The abstract syntax tree (AST) represents the syntactic structure of a program. Nodes on the AST represent elements such as statements and expressions.

Statement classes

This table lists subclasses of Stmt representing ECMAScript and TypeScript statements.

Statement syntax	CodeQL class	Superclasses
Expr ;	ExprStmt	
Label : Stmt	LabeledStmt	
;	EmptyStmt	
break Label ;	BreakStmt	JumpStmt, BreakOrContinueStmt
case Expr : Stmt…	Case	
catch(Identifier) { Stmt… }	CatchClause	ControlStmt
class Identifier extends Expr { MemberDeclaration… }	ClassDeclStmt	ClassDefinition, ClassOrInterface, T
const Identifier = Expr ;	ConstDeclStmt	DeclStmt
continue Label ;	ContinueStmt	JumpStmt, BreakOrContinueStmt
debugger;	DebuggerStmt	
declare global { Stmt… }	GlobalAugmentationDeclaration	
declare module StringLiteral { Stmt… }	ExternalModuleDeclaration	
default: Stmt…	Case	
do Stmt while (Expr)	DoWhileStmt	ControlStmt, LoopStmt
enum Identifier { MemberDeclaration… }	EnumDeclaration	NamespaceDefinition
export * from StringLiteral	BulkReExportDeclaration	ReExportDeclaration, ExportDeclar
export default ClassDeclStmt	ExportDefaultDeclaration	ExportDeclaration
export default Expr ;	ExportDefaultDeclaration	ExportDeclaration
export default FunctionDeclStmt	ExportDefaultDeclaration	ExportDeclaration
export { ExportSpecifier… };	ExportNamedDeclaration	ExportDeclaration
export DeclStmt	ExportNamedDeclaration	ExportDeclaration
export = Expr ;	ExportAssignDeclaration	
export as namespace Identifier ;	ExportAsNamespaceDeclaration	
for (Expr ; Expr ; Expr) Stmt	ForStmt	ControlStmt, LoopStmt
for (VarAccess in Expr) Stmt	ForInStmt	ControlStmt, LoopStmt, EnhancedF
for (VarAccess of Expr) Stmt	ForOfStmt	ControlStmt, LoopStmt, EnhancedF
function Identifier (Parameter…) { Stmt… }	FunctionDeclStmt	Function
if (Expr) Stmt else Stmt	IfStmt	ControlStmt

Table 4 – continued from prev

Statement syntax	CodeQL class	Superclasses
`import { ImportSpecifier... from StringLiteral`	ImportDeclaration	Import
`import Identifier = Expr ;`	ImportEqualsDeclaration	
`interface Identifier { MemberDeclaration... }`	InterfaceDeclaration	InterfaceDefinition, ClassOrInterfac
`let Identifier = Expr ;`	LetStmt	DeclStmt
`namespace Identifier { Stmt... }`	NamespaceDeclaration	NamespaceDefinition
`return Expr ;`	ReturnStmt	JumpStmt
`switch ( Expr ) { Case... }`	SwitchStmt	ControlStmt
`throw Expr ;`	ThrowStmt	JumpStmt
`try { Stmt... } CatchClause... finally { Stmt... }`	TryStmt	ControlStmt
`type Identifier = TypeExpr ;`	TypeAliasDeclaration	TypeParameterized
`var Identifier = Expr ;`	VarDeclStmt	DeclStmt
`while ( Expr ) Stmt`	WhileStmt	ControlStmt, LoopStmt
`with ( Expr ) Stmt`	WithStmt	ControlStmt
`{ Stmt... }`	BlockStmt	

Expression classes

There is a large number of expression classes, so we present them by category. All classes in this section are subclasses of Expr, except where noted otherwise.

Literals

All classes in this subsection are subclasses of Literal.

Expression syntax	CodeQL class
`true`	BooleanLiteral
`23`	NumberLiteral
`4.2`	NumberLiteral
`"Hello"`	StringLiteral
`/ab*c?/`	RegExpLiteral
`null`	NullLiteral

Identifiers

All identifiers are represented by the class Identifier, which has subclasses to represent specific kinds of identifiers:

- VarAccess: an identifier that refers to a variable
- VarDecl: an identifier that declares a variable, for example `x` in `var x = "hi"` or in `function(x) { }`
- VarRef: a VarAccess or a VarDecl
- Label: an identifier that refers to a statement label or a property, not a variable; in the following examples, `l` and `p` are labels:
  - `break l;`
  - `l:  for(;;) {}`
  - `x.p`

```
- { p:  42 }
```

Primary expressions

All classes in this subsection are subclasses of Expr.

Expression syntax	CodeQL class	Superclasses	Remarks
`this`	ThisExpr		
`[` Expr`...` `]`	ArrayExpr		
`{` Property`...` `}`	ObjectExpr		
`function (` Parameter`...` `)` `{` Stmt`...` `}`	FunctionExpr	Function	
`(` Parameter`...` `) =>` Expr	ArrowFunctionExpr	Function	
`(` Expr `)`	ParExpr		
`` ` ... ` ``	TemplateLiteral		an element in a TemplateLiteral is either a TemplateElement representing a constant template element, or some other expression representing an interpolated expression of the form `${` Expr `}`
Expr `` ` ... ` ``	TaggedTemplateExpr		an element in a TaggedTemplateExpr is either a TemplateElement representing a constant template element, or some other expression representing an interpolated expression of the form `${` Expr `}`

Properties

All classes in this subsection are subclasses of Property. Note that Property is not a subclass of Expr.

Property syntax	CodeQL class	Superclasses
Identifier `:` Expr	ValueProperty	
`get` Identifier `()` `{` Stmt`...` `}`	PropertyGetter	PropertyAccessor
`set` Identifier `(` Identifier `)` `{` Stmt`...` `}`	PropertySetter	PropertyAccessor

Property accesses

All classes in this subsection are subclasses of PropAccess.

Expression syntax	CodeQL class
Expr `.` Identifier	DotExpr
Expr `[` Expr `]`	IndexExpr

Function calls and `new`

All classes in this subsection are subclasses of InvokeExpr.

Expression syntax	CodeQL class	Remarks
Expr `(` Expr... `)`	CallExpr	
Expr `.` Identifier `(` Expr... `)`	MethodCallExpr	this also includes calls of the form Expr `[` Expr `]` `(` Expr... `)`
`new` Expr `(` Expr... `)`	NewExpr	

Unary expressions

All classes in this subsection are subclasses of UnaryExpr.

Expression syntax	CodeQL class
`~` Expr	BitNotExpr
`-` Expr	NegExpr
`+` Expr	PlusExpr
`!` Expr	LogNotExpr
`typeof` Expr	TypeofExpr
`void` Expr	VoidExpr
`delete` Expr	DeleteExpr
`...` Expr	SpreadElement

Binary expressions

All classes in this subsection are subclasses of BinaryExpr.

Expression syntax	CodeQL class	Superclasses
Expr * Expr	MulExpr	
Expr / Expr	DivExpr	
Expr % Expr	ModExpr	
Expr ** Expr	ExpExpr	
Expr + Expr	AddExpr	
Expr - Expr	SubExpr	
Expr << Expr	LShiftExpr	
Expr >> Expr	RShiftExpr	
Expr >>> Expr	URShiftExpr	
Expr && Expr	LogAndExpr	
Expr || Expr	LogOrExpr	
Expr < Expr	LTExpr	Comparison
Expr > Expr	GTExpr	Comparison
Expr <= Expr	LEExpr	Comparison
Expr >= Expr	GEExpr	Comparison
Expr == Expr	EqExpr	EqualityTest, Comparison
Expr != Expr	NEqExpr	EqualityTest, Comparison
Expr === Expr	StrictEqExpr	EqualityTest, Comparison
Expr !== Expr	StrictNEqExpr	EqualityTest, Comparison
Expr & Expr	BitAndExpr	
Expr | Expr	BitOrExpr	
Expr ^ Expr	XOrExpr	
Expr in Expr	InExpr	
Expr instanceof Expr	InstanceofExpr	

Assignment expressions

All classes in this table are subclasses of Assignment.

Expression syntax	CodeQL class	Superclasses
Expr = Expr	AssignExpr	
Expr += Expr	AssignAddExpr	CompoundAssignExpr
Expr -= Expr	AssignSubExpr	CompoundAssignExpr
Expr *= Expr	AssignMulExpr	CompoundAssignExpr
Expr **= Expr	AssignExpExpr	CompoundAssignExpr
Expr /= Expr	AssignDivExpr	CompoundAssignExpr
Expr %= Expr	AssignModExpr	CompoundAssignExpr
Expr &= Expr	AssignAndExpr	CompoundAssignExpr
Expr |= Expr	AssignOrExpr	CompoundAssignExpr
Expr ^= Expr	AssignXOrExpr	CompoundAssignExpr
Expr <<= Expr	AssignLShiftExpr	CompoundAssignExpr
Expr >>= Expr	AssignRShiftExpr	CompoundAssignExpr
Expr >>>= Expr	AssignURShiftExpr	CompoundAssignExpr

Update expressions

All classes in this table are subclasses of UpdateExpr.

Expression syntax	CodeQL class
Expr `++`	PostIncExpr
Expr `--`	PostDecExpr
`++` Expr	PreIncExpr
`--` Expr	PreDecExpr

Miscellaneous

All classes in this table are subclasses of Expr.

Expression syntax	CodeQL class
Expr `?` Expr `:` Expr	ConditionalExpr
Expr `,` `...` `,` Expr	SeqExpr
`await` Expr	AwaitExpr
`yield` Expr	YieldExpr

Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"

5.5.9 Data flow cheat sheet for JavaScript

This article describes parts of the JavaScript libraries commonly used for variant analysis and in data flow queries.

Taint tracking path queries

Use the following template to create a taint tracking path query:

```
/**
 * @kind path-problem
 */
import javascript
import DataFlow
import DataFlow::PathGraph

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "MyConfig" }
```

(continues on next page)

(continued from previous page)

```
  override predicate isSource(Node node) { ... }
  override predicate isSink(Node node) { ... }
  override predicate isAdditionalTaintStep(Node pred, Node succ) { ... }
}

from MyConfig cfg, PathNode source, PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "taint from $@.", source.getNode(), "here"
```

This query reports flow paths which:

- Begin at a node matched by isSource.
- Step through variables, function calls, properties, strings, arrays, promises, exceptions, and steps added by isAdditionalTaintStep.
- End at a node matched by isSink.

See also: "Global data flow" and "*Creating path queries*."

DataFlow module

Use data flow nodes to match program elements independently of syntax. See also: "*Analyzing data flow in JavaScript and TypeScript*."

Predicates in the `DataFlow::` module:

- moduleImport – finds uses of a module
- moduleMember – finds uses of a module member
- globalVarRef – finds uses of a global variable

Classes and member predicates in the `DataFlow::` module:

- **Node – something that can have a value, such as an expression, declaration, or SSA variable**
  - getALocalSource – find the node that this came from
  - getTopLevel – top-level scope enclosing this node
  - getFile – file containing this node
  - getIntValue – value of this node if it's is an integer constant
  - getStringValue – value of this node if it's is a string constant
  - mayHaveBooleanValue – check if the value is `true` or `false`
- **SourceNode extends Node – function call, parameter, object creation, or reference to a property or global variable**
  - getALocalUse – find nodes whose value came from this node
  - getACall – find calls with this as the callee
  - getAnInstantiation – find `new`-calls with this as the callee
  - getAnInvocation – find calls or `new`-calls with this as the callee
  - getAMethodCall – find method calls with this as the receiver
  - getAMemberCall – find calls with a member of this as the callee

  - getAPropertyRead – find property reads with this as the base
  - getAPropertyWrite – find property writes with this as the base
  - getAPropertySource – find nodes flowing into a property of this node
- **InvokeNode, NewNode, CallNode, MethodCallNode extends SourceNode – call to a function or constructor**
  - getArgument – an argument to the call
  - getCalleeNode – node being invoked as a function
  - getCalleeName – name of the variable or property being called
  - getOptionArgument – a "named argument" passed in through an object literal
  - getCallback – a function passed as a callback
  - getACallee - a function being called here
  - (MethodCallNode).getMethodName – name of the method being invoked
  - (MethodCallNode).getReceiver – receiver of the method call
- **FunctionNode extends SourceNode – definition of a function, including closures, methods, and class constructors**
  - getName – name of the function, derived from a variable or property name
  - getParameter – a parameter of the function
  - getReceiver – the node representing the value of `this`
  - getAReturn – get a returned expression
- **ParameterNode extends SourceNode – parameter of a function**
  - getName – the parameter name, if it has one
- **ClassNode extends SourceNode – class declaration or function that acts as a class**
  - getName – name of the class, derived from a variable or property name
  - getConstructor – the constructor function
  - getInstanceMethod – get an instance method by name
  - getStaticMethod – get a static method by name
  - getAnInstanceReference – find references to an instance of the class
  - getAClassReference – find references to the class itself
- **ObjectLiteralNode extends SourceNode – object literal**
  - getAPropertyWrite – a property in the object literal
  - getAPropertySource – value flowing into a property
- **ArrayCreationNode extends SourceNode – array literal or call to `Array` constructor**
  - getElement – an element of the array
- **PropRef, PropRead, PropWrite – read or write of a property**
  - getPropertyName – name of the property, if it is constant
  - getPropertyNameExpr – expression holding the name of the property
  - getBase – object whose property is accessed

  - (PropWrite).getRhs – right-hand side of the property assignment

StringOps module

- StringOps::Concatenation – string concatenation, using a plus operator, template literal, or array join call
- StringOps::StartsWith – check if a string starts with something
- StringOps::EndsWith – check if a string ends with something
- StringOps::Includes – check if a string contains something
- StringOps::RegExpTest – check if a string matches a RegExp

Utility

- ExtendCall – call that copies properties from one object to another
- JsonParserCall – call that deserializes a JSON string
- JsonStringifyCall – call that serializes a JSON string
- PropertyProjection – call that extracts nested properties by name

System and Network

- ClientRequest – outgoing network request
- DatabaseAccess – query being submitted to a database
- FileNameSource – reference to a filename
- **FileSystemAccess – file system operation**
  - FileSystemReadAccess – reading the contents of a file
  - FileSystemWriteAccess – writing to the contents of a file
- PersistentReadAccess – reading from persistent storage, like cookies
- PersistentWriteAccess – writing to persistent storage
- SystemCommandExecution – execution of a system command

Untrusted data

- **RemoteFlowSource – source of untrusted user input**
  - isUserControlledObject – is the input deserialized to a JSON-like object? (as opposed to just being a string)
- **ClientSideRemoteFlowSource extends RemoteFlowSource – input specific to the browser environment**
  - getKind – is this derived from the `path`, `fragment`, `query`, `url`, or `name`?
- **HTTP::RequestInputAccess extends RemoteFlowSource – input from an incoming HTTP request**
  - getKind – is this derived from a `parameter`, `header`, `body`, `url`, or `cookie`?
- **HTTP::RequestHeaderAccess extends RequestInputAccess – access to a specific header**
  - getAHeaderName – the name of a header being accessed

Note: some RemoteFlowSource instances, such as input from a web socket, belong to none of the specific subcategories above.

Files

- File, Folder extends Container – file or folder in the database
  - getBaseName – the name of the file or folder
  - getRelativePath – path relative to the database root

AST nodes

See also: "*Abstract syntax tree classes for working with JavaScript and TypeScript programs*."

Conversion between DataFlow and AST nodes:

- Node.asExpr() – convert node to an expression, if possible
- Expr.flow() – convert expression to a node (always possible)
- DataFlow::valueNode – convert expression or declaration to a node
- DataFlow::parameterNode – convert a parameter to a node
- DataFlow::thisNode – get the receiver node of a function

String matching

- x.matches("escape%") – holds if x starts with "escape"
- x.regexpMatch("escape.*") – holds if x starts with "escape"
- x.regexpMatch("(?i).*escape.*") – holds if x contains "escape" (case insensitive)

Access paths

When multiple property accesses are chained together they form what's called an "access path".

To identify nodes based on access paths, use the following predicates in AccessPath module:

- AccessPath::getAReferenceTo – find nodes that refer to the given access path
- AccessPath::getAnAssignmentTo – finds nodes that are assigned to the given access path
- AccessPath::getAnAliasedSourceNode – finds nodes that refer to the same access path

`getAReferenceTo` and `getAnAssignmentTo` have a 1-argument version for global access paths, and a 2-argument version for access paths starting at a given node.

Type tracking

See also: "*Using type tracking for API modeling*."

Use the following template to define forward type tracking predicates:

```
import DataFlow

SourceNode myType(TypeTracker t) {
  t.start() and
  result = /* SourceNode to track */
  or
  exists(TypeTracker t2 |
    result = myType(t2).track(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeTracker::end())
}
```

Use the following template to define backward type tracking predicates:

```
import DataFlow

SourceNode myType(TypeBackTracker t) {
  t.start() and
  result = (/* argument to track */).getALocalSource()
  or
  exists(TypeBackTracker t2 |
    result = myType(t2).backtrack(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeBackTracker::end())
}
```

Troubleshooting

- Using a call node as as sink? Try using getArgument to get an *argument* of the call node instead.
- Trying to use moduleImport or moduleMember as a call node? Try using getACall to get a *call* to the imported function, instead of the function itself.
- Compilation fails due to incompatible types? Make sure AST nodes and DataFlow nodes are not mixed up. Use asExpr() or flow() to convert.

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- "*QL language reference*"
- "*CodeQL tools*"
- *Basic query for JavaScript code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for JavaScript*: When you're analyzing a JavaScript program, you can make use of the large collection of classes in the CodeQL library for JavaScript.
- *CodeQL library for TypeScript*: When you're analyzing a TypeScript program, you can make use of the large collection of classes in the CodeQL library for TypeScript.
- *Analyzing data flow in JavaScript and TypeScript*: This topic describes how data flow analysis is implemented in the CodeQL libraries for JavaScript/TypeScript and includes examples to help you write your own data flow queries.
- *Using flow labels for precise data flow analysis*: You can associate flow labels with each value tracked by the flow analysis to determine whether the flow contains potential vulnerabilities.
- *Specifying remote flow sources for JavaScript*: You can model potential sources of untrusted user input in your code without making changes to the CodeQL standard library by specifying extra remote flow sources in an external file.
- *Using type tracking for API modeling*: You can track data through an API by creating a model using the CodeQL type-tracking library for JavaScript.
- *Abstract syntax tree classes for working with JavaScript and TypeScript programs*: CodeQL has a large selection of classes for representing the abstract syntax tree of JavaScript and TypeScript programs.
- *Data flow cheat sheet for JavaScript*: This article describes parts of the JavaScript libraries commonly used for variant analysis and in data flow queries.

5.6 CodeQL for Python

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Python codebases.

5.6.1 Basic query for Python code

Learn to write and run a simple CodeQL query using LGTM.

About the query

The query we're going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they only include a `pass` statement. For example, code such as:

```
if error: pass
```

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Python** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

   ```
   import python

   from If ifstmt, Stmt pass
   where pass = ifstmt.getStmt(0) and
     pass instanceof Pass
   select ifstmt, "This 'if' statement is redundant."
   ```

   LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   > Note
   >
   > Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

   Example query results

   > Note
   >
   > An ellipsis (…) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import python`	Imports the standard CodeQL libraries for Python.	Every query begins with one or more `import` statements.
`from If ifstmt, Stmt pass`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `If` variable for `if` statements • a `Stmt` variable for the statement
`where pass = ifstmt.getStmt(0) and pass instanceof Pass`	Defines a condition on the variables.	`pass = ifstmt.getStmt(0)`: `pass` is the first statement in the `if` statement. `pass instanceof Pass`: `pass` must be a pass statement. In other words, the first statement contained in the `if` statement is a `pass` statement.
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where a `pass` statement does serve a purpose. For example:

```
if cond():
  pass
else:
  do_something()
```

In this case, identifying the `if` statement with the `pass` statement as redundant is a false positive. One solution to this is to modify the query to ignore `pass` statements if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Extend the `where` clause to include the following extra condition:

   ```
   and not exists(ifstmt.getOrelse())
   ```

   The `where` clause is now:

   ```
   where pass = ifstmt.getStmt(0) and
     pass instanceof Pass and
     not exists(ifstmt.getOrelse())
   ```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer included.

See this in the query console

Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"

5.6.2 CodeQL library for Python

When you need to analyze a Python program, you can make use of the large collection of classes in the CodeQL library for Python.

About the CodeQL library for Python

The CodeQL library for each programming language uses classes with abstractions and predicates to present data in an object-oriented form.

Each CodeQL library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `python.qll` imports all the core Python library modules, so you can include the complete library by beginning your query with:

```
import python
```

The CodeQL library for Python incorporates a large number of classes. Each class corresponds either to one kind of entity in Python source code or to an entity that can be derived from the source code using static analysis. These classes can be divided into four categories:

- **Syntactic** - classes that represent entities in the Python source code.
- **Control flow** - classes that represent entities from the control flow graphs.
- **Data flow** - classes that represent entities from the data flow graphs.
- **API graphs** - classes that represent entities from the API graphs.

The first two categories are described below. For a description of data flow and associated classes, see "*Analyzing data flow in Python*". For a description of API graphs and their use, see "*Using API graphs in Python*."

Syntactic classes

This part of the library represents the Python source code. The `Module`, `Class`, and `Function` classes correspond to Python modules, classes, and functions respectively, collectively these are known as `Scope` classes. Each `Scope` contains a list of statements each of which is represented by a subclass of the class `Stmt`. Statements themselves can contain other statements or expressions which are represented by subclasses of `Expr`. Finally, there are a few additional classes for the parts of more complex expressions such as list comprehensions. Collectively these classes are subclasses of `AstNode` and form an Abstract syntax tree (AST). The root of each AST is a `Module`. Symbolic information is attached to the AST in the form of variables (represented by the class `Variable`). For more information, see Abstract syntax tree and Symbolic information on Wikipedia.

Scope

A Python program is a group of modules. Technically a module is just a list of statements, but we often think of it as composed of classes and functions. These top-level entities, the module, class, and function are represented by the three CodeQL classes Module, Class and Function which are all subclasses of `Scope`.

- `Scope`
  - `Module`
  - `Class`
  - `Function`

All scopes are basically a list of statements, although `Scope` classes have additional attributes such as names. For example, the following query finds all functions whose scope (the scope in which they are declared) is also a function:

```
import python

from Function f
where f.getScope() instanceof Function
select f
```

See this in the query console on LGTM.com. Many projects have nested functions.

Statement

A statement is represented by the Stmt class which has about 20 subclasses representing the various kinds of statements, such as the `Pass` statement, the `Return` statement or the `For` statement. Statements are usually made up of parts. The most common of these is the expression, represented by the `Expr` class. For example, take the following Python `for` statement:

```
for var in seq:
    pass
else:
    return 0
```

The For class representing the `for` statement has a number of member predicates to access its parts:

- `getTarget()` returns the `Expr` representing the variable `var`.
- `getIter()` returns the `Expr` resenting the variable `seq`.
- `getBody()` returns the statement list body.
- `getStmt(0)` returns the pass `Stmt`.
- `getOrElse()` returns the `StmtList` containing the return statement.

Expression

Most statements are made up of expressions. The Expr class is the superclass of all expression classes, of which there are about 30 including calls, comprehensions, tuples, lists and arithmetic operations. For example, the Python expression `a+2` is represented by the `BinaryExpr` class:

- `getLeft()` returns the `Expr` representing the `a`.
- `getRight()` returns the `Expr` representing the `2`.

As an example, to find expressions of the form `a+2` where the left is a simple name and the right is a numeric constant we can use the following query:

Finding expressions of the form "a+2"

```
import python

from BinaryExpr bin
where bin.getLeft() instanceof Name and bin.getRight() instanceof Num
select bin
```

See this in the query console on LGTM.com. Many projects include examples of this pattern.

Variable

Variables are represented by the Variable class in the CodeQL library. There are two subclasses, `LocalVariable` for function-level and class-level variables and `GlobalVariable` for module-level variables.

Other source code elements

Although the meaning of the program is encoded by the syntactic elements, `Scope`, `Stmt` and `Expr` there are some parts of the source code not covered by the abstract syntax tree. The most useful of these is the Comment class which describes comments in the source code.

Examples

Each syntactic element in Python source is recorded in the CodeQL database. These can be queried via the corresponding class. Let us start with a couple of simple examples.

1. Finding all `finally` blocks

For our first example, we can find all `finally` blocks by using the `Try` class:

Find all `finally` blocks

```
import python

from Try t
select t.getFinalbody()
```

See this in the query console on LGTM.com. Many projects include examples of this pattern.

2. Finding `except` blocks that do nothing

For our second example, we can use a simplified version of a query from the standard query set. We look for all `except` blocks that do nothing.

A block that does nothing is one that contains no statements except `pass` statements. We can encode this as:

```
not exists(Stmt s | s = ex.getAStmt() | not s instanceof Pass)
```

where `ex` is an `ExceptStmt` and `Pass` is the class representing `pass` statements. Instead of using the double negative, "**no** *statements that are* **not** *pass statements"*, this can also be expressed positively, *"all statements must be pass statements."* The positive form is expressed using the `forall` quantifier:

```
forall(Stmt s | s = ex.getAStmt() | s instanceof Pass)
```

Both forms are equivalent. Using the positive expression, the whole query looks like this:

Find pass-only `except` blocks

```
import python

from ExceptStmt ex
```

(continues on next page)

(continued from previous page)

```
where forall(Stmt s | s = ex.getAStmt() | s instanceof Pass)
select ex
```

See this in the query console on LGTM.com. Many projects include pass-only `except` blocks.

Summary

The most commonly used standard classes in the syntactic part of the library are organized as follows:

`Module`, `Class`, `Function`, `Stmt`, and `Expr` - they are all subclasses of AstNode.

Abstract syntax tree

- `AstNode`
  - `Module` – A Python module
  - `Class` – The body of a class definition
  - `Function` – The body of a function definition
  - `Stmt` – A statement
    - `Assert` – An `assert` statement
    - `Assign` – An assignment
      - `AssignStmt` – An assignment statement, `x = y`
      - `ClassDef` – A class definition statement
      - `FunctionDef` – A function definition statement
    - `AugAssign` – An augmented assignment, `x += y`
    - `Break` – A `break` statement
    - `Continue` – A `continue` statement
    - `Delete` – A `del` statement
    - `ExceptStmt` – The `except` part of a `try` statement
    - `Exec` – An exec statement
    - `For` – A `for` statement
    - `If` – An `if` statement
    - `Pass` – A `pass` statement
    - `Print` – A `print` statement (Python 2 only)
    - `Raise` – A raise statement
    - `Return` – A `return` statement
    - `Try` – A `try` statement
    - `While` – A `while` statement
    - `With` – A `with` statement
  - `Expr` – An expression

* `Attribute` – An attribute, `obj.attr`
* `Call` – A function call, `f(arg)`
* `IfExp` – A conditional expression, `x if cond else y`
* `Lambda - A lambda expression`
* `Yield` – A `yield` expression
* `Bytes` – A bytes literal, `b"x"` or (in Python 2) `"x"`
* `Unicode` – A unicode literal, `u"x"` or (in Python 3) `"x"`
* `Num` – A numeric literal, `3` or `4.2`
    - `IntegerLiteral`
    - `FloatLiteral`
    - `ImaginaryLiteral`
* `Dict` – A dictionary literal, `{'a': 2}`
* `Set` – A set literal, `{'a', 'b'}`
* `List` – A list literal, `['a', 'b']`
* `Tuple` – A tuple literal, `('a', 'b')`
* `DictComp` – A dictionary comprehension, `{k: v for ...}`
* `SetComp` – A set comprehension, `{x for ...}`
* `ListComp` – A list comprehension, `[x for ...]`
* `GenExpr` – A generator expression, `(x for ...)`
* `Subscript` – A subscript operation, `seq[index]`
* `Name` – A reference to a variable, `var`
* `UnaryExpr` – A unary operation, `-x`
* `BinaryExpr` – A binary operation, `x+y`
* `Compare` – A comparison operation, `0 < x < 10`
* `BoolExpr` – Short circuit logical operations, `x and y`, `x or y`

Variables

- `Variable` – A variable
  - `LocalVariable` – A variable local to a function or a class
  - `GlobalVariable` – A module level variable

Other

- `Comment` – A comment

Control flow classes

This part of the library represents the control flow graph of each `Scope` (classes, functions, and modules). Each `Scope` contains a graph of `ControlFlowNode` elements. Each scope has a single entry point and at least one (potentially many) exit points. To speed up control and data flow analysis, control flow nodes are grouped into basic blocks. For more information, see Basic block on Wikipedia.

Example

If we want to find the longest sequence of code without any branches, we need to consider control flow. A `BasicBlock` is, by definition, a sequence of code without any branches, so we just need to find the longest `BasicBlock`.

First of all we introduce a simple predicate `bb_length()` which relates `BasicBlock`s to their length.

```
int bb_length(BasicBlock b) {
    result = max(int i | exists(b.getNode(i))) + 1
}
```

Each `ControlFlowNode` within a `BasicBlock` is numbered consecutively, starting from zero, therefore the length of a `BasicBlock` is equal to one more than the largest index within that `BasicBlock`.

Using this predicate we can select the longest `BasicBlock` by selecting the `BasicBlock` whose length is equal to the maximum length of any `BasicBlock`:

Find the longest sequence of code without branches

```
import python

int bb_length(BasicBlock b) {
    result = max(int i | exists(b.getNode(i)) | i) + 1
}

from BasicBlock b
where bb_length(b) = max(bb_length(_))
select b
```

See this in the query console on LGTM.com. When we ran it on the LGTM.com demo projects, the *openstack/nova* and *ytdl-org/youtube-dl* projects both contained source code results for this query.

> Note
>
> The special underscore variable _ means any value; so `bb_length(_)` is the length of any block.

Summary

The classes in the control-flow part of the library are:

- ControlFlowNode – A control-flow node. There is a one-to-many relation between AST nodes and control-flow nodes.
- BasicBlock – A non branching list of control-flow nodes.

Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"

5.6.3 Analyzing data flow in Python

You can use CodeQL to track the flow of data through a Python program to places where the data is used.

About this article

This article describes how data flow analysis is implemented in the CodeQL libraries for Python and includes examples to help you write your own data flow queries. The following sections describe how to use the libraries for local data flow, global data flow, and taint tracking. For a more general introduction to modeling data flow, see "*About data flow analysis*."

Local data flow

Local data flow is data flow within a single method or callable. Local data flow is easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. The `Node` class has a number of useful subclasses, such as `ExprNode` for expressions, `CfgNode` for control-flow nodes, `CallCfgNode` for function and method calls, and `ParameterNode` for parameters. You can map between data flow nodes and expressions/control-flow nodes using the member predicates `asExpr` and `asCfgNode`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the control-flow node corresponding to this node, if any. */
  ControlFlowNode asCfgNode() { ... }

 ...
}
```

or using the predicate `exprNode`:

```
/**
 * Gets a node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }
```

Due to the control-flow graph being split, there can be multiple data-flow nodes associated with a single expression.

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively, by using the + and * operators, or you can use the predefined recursive predicate `localFlow`.

For example, you can find flow from an expression `source` to an expression `sink` in zero or more local steps:

```
DataFlow::localFlow(DataFlow::exprNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
temp = x
y = temp + ", " + temp
```

If `x` is a tainted string then `y` is also tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint propagation edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively, by using the + and * operators, or you can use the predefined recursive predicate `localTaint`.

For example, you can find taint propagation from an expression `source` to an expression `sink` in zero or more local steps:

```
TaintTracking::localTaint(DataFlow::exprNode(source), DataFlow::exprNode(sink))
```

Using local sources

When asking for local data flow or taint propagation between two expressions as above, you would normally constrain the expressions to be relevant to a certain investigation. The next section will give some concrete examples, but there is a more abstract concept that we should call out explicitly, namely that of a local source.

A local source is a data-flow node with no local data flow into it. As such, it is a local origin of data flow, a place where a new value is created. This includes parameters (which only receive global data flow) and most expressions (because they are not value-preserving). Restricting attention to such local sources gives a much lighter and more performant data-flow graph and in most cases also a more suitable abstraction for the investigation of interest. The class `LocalSourceNode` represents data-flow nodes that are also local sources. It comes with a useful member predicate `flowsTo(DataFlow::Node node)`, which holds if there is local data flow from the local source to `node`.

Examples

Python has builtin functionality for reading and writing files, such as the function `open`. However, there is also the library `os` which provides low-level file access. This query finds the filename passed to `os.open`:

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call
where
  call = API::moduleImport("os").getMember("open").getACall()
select call.getArg(0)
```

See this in the query console on LGTM.com. Two of the demo projects make use of this low-level API.

Notice the use of the `API` module for referring to library functions. For more information, see "*Using API graphs in Python*."

Unfortunately this will only give the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call, DataFlow::ExprNode expr
where
  call = API::moduleImport("os").getMember("open").getACall() and
  DataFlow::localFlow(expr, call.getArg(0))
select call, expr
```

See this in the query console on LGTM.com. Many expressions flow to the same call.

We see that we get several data-flow nodes for an expression as it flows towards a call (notice repeated locations in the `call` column). We are mostly interested in the "first" of these, what might be called the local source for the file name. To restrict attention to such local sources, and to simultaneously make the analysis more performant, we have the QL class `LocalSourceNode`. We could demand that `expr` is such a node:

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call, DataFlow::ExprNode expr
where
  call = API::moduleImport("os").getMember("open").getACall() and
  DataFlow::localFlow(expr, call.getArg(0)) and
  expr instanceof DataFlow::LocalSourceNode
select call, expr
```

However, we could also enforce this by casting. That would allow us to use the member function `flowsTo` on `LocalSourceNode` like so:

```
import python
import semmle.python.dataflow.new.DataFlow
```

(continues on next page)

(continued from previous page)

```
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call, DataFlow::ExprNode expr
where
  call = API::moduleImport("os").getMember("open").getACall() and
  expr.(DataFlow::LocalSourceNode).flowsTo(call.getArg(0))
select call, expr
```

As an alternative, we can ask more directly that `expr` is a local source of the first argument, via the predicate `getALocalSource`:

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call, DataFlow::ExprNode expr
where
  call = API::moduleImport("os").getMember("open").getACall() and
  expr = call.getArg(0).getALocalSource()
select call, expr
```

See this in the query console on LGTM.com. All these three queries give identical results. We now mostly have one expression per call.

We still have some cases of more than one expression flowing to a call, but then they flow through different code paths (possibly due to control-flow splitting, as in the second case).

We might want to make the source more specific, for example a parameter to a function or method. This query finds instances where a parameter is used as the name when opening a file:

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call, DataFlow::ParameterNode p
where
  call = API::moduleImport("os").getMember("open").getACall() and
  DataFlow::localFlow(p, call.getArg(0))
select call, p
```

See this in the query console on LGTM.com. Very few results now; these could feasibly be inspected manually.

Using the exact name supplied via the parameter may be too strict. If we want to know if the parameter influences the file name, we can use taint tracking instead of data flow. This query finds calls to `os.open` where the filename is derived from a parameter:

```
import python
import semmle.python.dataflow.new.TaintTracking
import semmle.python.ApiGraphs

from DataFlow::CallCfgNode call, DataFlow::ParameterNode p
where
  call = API::moduleImport("os").getMember("open").getACall() and
  TaintTracking::localTaint(p, call.getArg(0))
```

(continues on next page)

(continued from previous page)

```
select call, p
```

See this in the query console on LGTM.com. Now we get more results and in more projects.

Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see *Creating path queries*.

Using global data flow

The global data flow library is used by extending the class `DataFlow::Configuration`:

```
import python

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "..." }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource` - defines where data may flow from.
- `isSink` - defines where data may flow to.
- `isBarrier` - optionally, restricts the data flow.
- `isAdditionalFlowStep` - optionally, adds additional flow steps.

The characteristic predicate (`MyDataFlowConfiguration()`) defines the name of the configuration, so `"..."` must be replaced with a unique name (for instance the class name).

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguation dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Dataflow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow what local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. The global taint tracking library is used by extending the class `TaintTracking::Configuration`:

```
import python

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "..." }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource` - defines where taint may flow from.
- `isSink` - defines where taint may flow to.
- `isSanitizer` - optionally, restricts the taint flow.
- `isAdditionalTaintStep` - optionally, adds additional taint steps.

Similar to global data flow, the characteristic predicate (`MyTaintTrackingConfiguration()`) defines the unique name of the configuration and the taint analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Predefined sources and sinks

The data flow library contains a number of predefined sources and sinks, providing a good starting point for defining data flow based security queries.

- The class `RemoteFlowSource` (defined in module `semmle.python.dataflow.new.RemoteFlowSources`) represents data flow from remote network inputs. This is useful for finding security problems in networked services.
- The library `Concepts` (defined in module `semmle.python.Concepts`) contain several subclasses of `DataFlow::Node` that are security relevant, such as `FileSystemAccess` and `SqlExecution`.
- The module `Attributes` (defined in module `semmle.python.dataflow.new.internal.Attributes`) defines `AttrRead` and `AttrWrite` which handle both ordinary and dynamic attribute access.

For global flow, it is also useful to restrict sources to instances of `LocalSourceNode`. The predefined sources generally do that.

Class hierarchy

- `DataFlow::Configuration` - base class for custom global data flow analysis.
- `DataFlow::Node` - an element behaving as a data flow node.
  - `DataFlow::CfgNode` - a control-flow node behaving as a data flow node.
    - `DataFlow::ExprNode` - an expression behaving as a data flow node.
    - `DataFlow::ParameterNode` - a parameter data flow node representing the value of a parameter at function entry.
    - `DataFlow::CallCfgNode` - a control-flow node for a function or method call behaving as a data flow node.
  - `RemoteFlowSource` - data flow from network/remote input.
  - `Attributes::AttrRead` - an attribute read as a data flow node.
  - `Attributes::AttrWrite` - an attribute write as a data flow node.
  - `Concepts::SystemCommandExecution` - a data-flow node that executes an operating system command, for instance by spawning a new process.
  - `Concepts::FileSystemAccess` - a data flow node that performs a file system access, including reading and writing data, creating and deleting files and folders, checking and updating permissions, and so on.
  - `Concepts::Path::PathNormalization` - a data-flow node that performs path normalization. This is often needed in order to safely access paths.
  - `Concepts::Decoding` - a data-flow node that decodes data from a binary or textual format. A decoding (automatically) preserves taint from input to output. However, it can also be a problem in itself, for example if it allows code execution or could result in denial-of-service.
  - `Concepts::Encoding` - a data-flow node that encodes data to a binary or textual format. An encoding (automatically) preserves taint from input to output.
  - `Concepts::CodeExecution` - a data-flow node that dynamically executes Python code.
  - `Concepts::SqlExecution` - a data-flow node that executes SQL statements.
  - `Concepts::HTTP::Server::RouteSetup` - a data-flow node that sets up a route on a server.
  - `Concepts::HTTP::Server::HttpResponse` - a data-flow node that creates a HTTP response on a server.
- `TaintTracking::Configuration` - base class for custom global taint tracking analysis.

Examples

This query shows a data flow configuration that uses all network input as data sources:

```
import python
import semmle.python.dataflow.new.DataFlow
import semmle.python.dataflow.new.TaintTracking
import semmle.python.dataflow.new.RemoteFlowSources
import semmle.python.Concepts

class RemoteToFileConfiguration extends TaintTracking::Configuration {
  RemoteToFileConfiguration() { this = "RemoteToFileConfiguration" }
```

(continues on next page)

(continued from previous page)

```
  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  override predicate isSink(DataFlow::Node sink) {
    sink = any(FileSystemAccess fa).getAPathArgument()
  }
}

from DataFlow::Node input, DataFlow::Node fileAccess, RemoteToFileConfiguration config
where config.hasFlow(input, fileAccess)
select fileAccess, "This file access uses data from $@.",
  input, "user-controllable input."
```

This data flow configuration tracks data flow from environment variables to opening files:

```
import python
import semmle.python.dataflow.new.TaintTracking
import semmle.python.ApiGraphs

class EnvironmentToFileConfiguration extends DataFlow::Configuration {
  EnvironmentToFileConfiguration() { this = "EnvironmentToFileConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    source = API::moduleImport("os").getMember("getenv").getACall()
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(DataFlow::CallCfgNode call |
      call = API::moduleImport("os").getMember("open").getACall() and
      sink = call.getArg(0)
    )
  }
}

from Expr environment, Expr fileOpen, EnvironmentToFileConfiguration config
where config.hasFlow(DataFlow::exprNode(environment), DataFlow::exprNode(fileOpen))
select fileOpen, "This call to 'os.open' uses data from $@.",
  environment, "call to 'os.getenv'"
```

Running this in the query console on LGTM.com unsurprisingly yields no results in the demo projects.

Further reading

- "*Exploring data flow with path queries*"
- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"

5.6.4 Using API graphs in Python

API graphs are a uniform interface for referring to functions, classes, and methods defined in external libraries.

About this article

This article describes how to use API graphs to reference classes and functions defined in library code. You can use API graphs to conveniently refer to external library functions when defining things like remote flow sources.

Module imports

The most common entry point into the API graph will be the point where an external module or package is imported. For example, you can access the API graph node corresponding to the `re` library by using the `API::moduleImport` method defined in the `semmle.python.ApiGraphs` module, as the following snippet demonstrates.

```
import python
import semmle.python.ApiGraphs

select API::moduleImport("re")
```

See this in the query console on LGTM.com.

This query selects the API graph node corresponding to the `re` module. This node represents the fact that the `re` module has been imported rather than a specific location in the program where the import happens. Therefore, there will be at most one result per project, and it will not have a useful location, so you'll have to click *Show 1 non-source result* in order to see it.

To find where the `re` module is referenced in the program, you can use the `getAUse` method. The following query selects all references to the `re` module in the current database.

```
import python
import semmle.python.ApiGraphs

select API::moduleImport("re").getAUse()
```

See this in the query console on LGTM.com.

Note that the `getAUse` method accounts for local flow, so that `my_re_compile` in the following snippet is correctly recognized as a reference to the `re.compile` function.

```
from re import compile as re_compile

my_re_compile = re_compile

r = my_re_compile(".*")
```

If you only require immediate uses, without taking local flow into account, then you can use the `getAnImmediateUse` method instead.

Note that the given module name *must not* contain any dots. Thus, something like `API::moduleImport("flask.views")` will not do what you expect. Instead, this should be decomposed into an access of the `views` member of the API graph node for `flask`, as described in the next section.

Accessing attributes

Given a node in the API graph, you can access its attributes by using the `getMember` method. Using the above `re.compile` example, you can now find references to `re.compile`.

```
import python
import semmle.python.ApiGraphs

select API::moduleImport("re").getMember("compile").getAUse()
```

See this in the query console on LGTM.com.

In addition to `getMember`, you can use the `getUnknownMember` method to find references to API components where the name is not known statically. You can use the `getAMember` method to access all members, both known and unknown.

Calls and class instantiations

To track instances of classes defined in external libraries, or the results of calling externally defined functions, you can use the `getReturn` method. The following snippet finds all places where the return value of `re.compile` is used:

```
import python
import semmle.python.ApiGraphs

select API::moduleImport("re").getMember("compile").getReturn().getAUse()
```

See this in the query console on LGTM.com.

Note that this includes all uses of the result of `re.compile`, including those reachable via local flow. To get just the *calls* to `re.compile`, you can use `getAnImmediateUse` instead of `getAUse`. As this is a common occurrence, you can use `getACall` instead of `getReturn` followed by `getAnImmediateUse`.

See this in the query console on LGTM.com.

Note that the API graph does not distinguish between class instantiations and function calls. As far as it's concerned, both are simply places where an API graph node is called.

Subclasses

For many libraries, the main mode of usage is to extend one or more library classes. To track this in the API graph, you can use the `getASubclass` method to get the API graph node corresponding to all the immediate subclasses of this node. To find *all* subclasses, use * or + to apply the method repeatedly, as in `getASubclass*`.

Note that `getASubclass` does not account for any subclassing that takes place in library code that has not been extracted. Thus, it may be necessary to account for this in the models you write. For example, the `flask.views.View` class has a predefined subclass `MethodView`. To find all subclasses of `View`, you must explicitly include the subclasses of `MethodView` as well.

```
import python
import semmle.python.ApiGraphs

API::Node viewClass() {
  result =
    API::moduleImport("flask").getMember("views").getMember(["View", "MethodView"]).
↪getASubclass*()
}

select viewClass().getAUse()
```

See this in the query console on LGTM.com.

Note the use of the set literal `["View", "MethodView"]` to match both classes simultaneously.

Built-in functions and classes

You can access built-in functions and classes using the `API::builtin` method, giving the name of the built-in as an argument.

For example, to find all calls to the built-in `open` function, you can use the following snippet.

```
import python
import semmle.python.ApiGraphs

select API::builtin("open").getACall()
```

Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"

5.6.5 Functions in Python

You can use syntactic classes from the standard CodeQL library to find Python functions and identify calls to them.

These examples use the standard CodeQL class Function. For more information, see "*CodeQL library for Python*."

Finding all functions called "get..."

In this example we look for all the "getters" in a program. Programmers moving to Python from Java are often tempted to write lots of getter and setter methods, rather than use properties. We might want to find those methods.

Using the member predicate `Function.getName()`, we can list all of the getter functions in a database:

> Tip
>
> Instead of copying this query, try typing the code. As you start to write a name that matches a library class, a pop-up is displayed making it easy for you to select the class that you want.

```
import python

from Function f
where f.getName().matches("get%")
select f, "This is a function called get..."
```

See this in the query console on LGTM.com. This query typically finds a large number of results. Usually, many of these results are for functions (rather than methods) which we are not interested in.

Finding all methods called "get..."

You can modify the query above to return more interesting results. As we are only interested in methods, we can use the `Function.isMethod()` predicate to refine the query.

```
import python

from Function f
where f.getName().matches("get%") and f.isMethod()
select f, "This is a method called get..."
```

See this in the query console on LGTM.com. This finds methods whose name starts with `"get"`, but many of those are not the sort of simple getters we are interested in.

Finding one line methods called "get..."

We can modify the query further to include only methods whose body consists of a single statement. We do this by counting the number of lines in each method.

```
import python

from Function f
where f.getName().matches("get%") and f.isMethod()
 and count(f.getAStmt()) = 1
select f, "This function is (probably) a getter."
```

See this in the query console on LGTM.com. This query returns fewer results, but if you examine the results you can see that there are still refinements to be made. This is refined further in "*Expressions and statements in Python*."

Finding a call to a specific function

This query uses `Call` and `Name` to find calls to the function `eval` - which might potentially be a security hazard.

```
import python

from Call call, Name name
where call.getFunc() = name and name.getId() = "eval"
select call, "call to 'eval'."
```

See this in the query console on LGTM.com. Some of the demo projects on LGTM.com use this function.

The `Call` class represents calls in Python. The `Call.getFunc()` predicate gets the expression being called. `Name.getId()` gets the identifier (as a string) of the `Name` expression. Due to the dynamic nature of Python, this query will select any call of the form `eval(...)` regardless of whether it is a call to the built-in function `eval` or not. In a later tutorial we will see how to use the type-inference library to find calls to the built-in function `eval` regardless of name of the variable called.

Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"

5.6.6 Expressions and statements in Python

You can use syntactic classes from the CodeQL library to explore how Python expressions and statements are used in a code base.

Statements

The bulk of Python code takes the form of statements. Each different type of statement in Python is represented by a separate CodeQL class.

Here is the full class hierarchy:

- `Stmt` – A statement
  - `Assert` – An `assert` statement
  - `Assign`
    - `AssignStmt` – An assignment statement, `x = y`
    - `ClassDef` – A class definition statement
    - `FunctionDef` – A function definition statement
  - `AugAssign` – An augmented assignment, `x += y`
  - `Break` – A `break` statement
  - `Continue` – A `continue` statement

- `Delete` – A `del` statement
- `ExceptStmt` – The `except` part of a `try` statement
- `Exec` – An `exec` statement
- `For` – A `for` statement
- `Global` – A `global` statement
- `If` – An `if` statement
- `ImportStar` – A `from xxx import *` statement
- `Import` – Any other `import` statement
- `Nonlocal` – A `nonlocal` statement
- `Pass` – A `pass` statement
- `Print` – A `print` statement (Python 2 only)
- `Raise` – A `raise` statement
- `Return` – A `return` statement
- `Try` – A `try` statement
- `While` – A `while` statement
- `With` – A `with` statement

Example finding redundant 'global' statements

The `global` statement in Python declares a variable with a global (module-level) scope, when it would otherwise be local. Using the `global` statement outside a class or function is redundant as the variable is already global.

```
import python

from Global g
where g.getScope() instanceof Module
select g
```

See this in the query console on LGTM.com. None of the demo projects on LGTM.com has a global statement that matches this pattern.

The line: `g.getScope() instanceof Module` ensures that the `Scope` of `Global g` is a `Module`, rather than a class or function.

Example finding 'if' statements with redundant branches

An `if` statement where one branch is composed of just `pass` statements could be simplified by negating the condition and dropping the `else` clause.

```
if cond():
    pass
else:
    do_something
```

To find statements like this that could be simplified we can write a query.

```
import python

from If i, StmtList l
where (l = i.getBody() or l = i.getOrelse())
  and forall(Stmt p | p = l.getAnItem() | p instanceof Pass)
select i
```

See this in the query console on LGTM.com. Many projects have some `if` statements that match this pattern.

The line: `(l = i.getBody() or l = i.getOrelse())` restricts the `StmtList l` to branches of the `if` statement.

The line: `forall(Stmt p | p = l.getAnItem() | p instanceof Pass)` ensures that all statements in `l` are `pass` statements.

Expressions

Each kind of Python expression has its own class. Here is the full class hierarchy:

- `Expr` – An expression
  - `Attribute` – An attribute, `obj.attr`
  - `BinaryExpr` – A binary operation, `x+y`
  - `BoolExpr` – Short circuit logical operations, `x and y`, `x or y`
  - `Bytes` – A bytes literal, `b"x"` or (in Python 2) `"x"`
  - `Call` – A function call, `f(arg)`
  - `Compare` – A comparison operation, `0 < x < 10`
  - `Dict` – A dictionary literal, `{'a': 2}`
  - `DictComp` – A dictionary comprehension, `{k: v for ...}`
  - `Ellipsis` – An ellipsis expression, `...`
  - `GeneratorExp` – A generator expression
  - `IfExp` – A conditional expression, `x if cond else y`
  - `ImportExpr` – An artificial expression representing the module imported
  - `ImportMember` - An artificial expression representing importing a value from a module (part of an `from xxx import *` statement)
  - `Lambda` - A lambda expression
  - `List` – A list literal, `['a', 'b']`
  - `ListComp` – A list comprehension, `[x for ...]`
  - `Name` – A reference to a variable, `var`
  - `Num` – A numeric literal, `3` or `4.2`
    - `FloatLiteral`
    - `ImaginaryLiteral`
    - `IntegerLiteral`
  - `Repr` – A backticks expression, `x` (Python 2 only)
  - `Set` – A set literal, `{'a', 'b'}`

- `SetComp` – A set comprehension, `{x for ...}`
- `Slice` – A slice; the `0:1` in the expression `seq[0:1]`
- `Starred` – A starred expression, `*x` in the context of a multiple assignment: `y, *x = 1,2,3` (Python 3 only)
- `StrConst` – A string literal. In Python 2 either bytes or unicode. In Python 3 only unicode.
- `Subscript` – A subscript operation, `seq[index]`
- `UnaryExpr` – A unary operation, `-x`
- `Unicode` – A unicode literal, `u"x"` or (in Python 3) `"x"`
- `Yield` – A `yield` expression
- `YieldFrom` – A `yield from` expression (Python 3.3+)

Example finding comparisons to integer or string literals using 'is'

Python implementations commonly cache small integers and single character strings, which means that comparisons such as the following often work correctly, but this is not guaranteed and we might want to check for them.

```
x is 10
x is "A"
```

We can check for these using a query.

```
import python

from Compare cmp, Expr literal
where (literal instanceof StrConst or literal instanceof Num)
  and cmp.getOp(0) instanceof Is and cmp.getComparator(0) = literal
select cmp
```

See this in the query console on LGTM.com. Two of the demo projects on LGTM.com use this pattern: *saltstack/salt* and *openstack/nova*.

The clause `cmp.getOp(0) instanceof Is and cmp.getComparator(0) = literal` checks that the first comparison operator is "is" and that the first comparator is a literal.

> Tip
>
> We have to use `cmp.getOp(0)` and `cmp.getComparator(0)`as there is no `cmp.getOp()` or `cmp.getComparator()`. The reason for this is that a `Compare` expression can have multiple operators. For example, the expression `3 < x < 7` has two operators and two comparators. You use `cmp.getComparator(0)` to get the first comparator (in this example the `x`) and `cmp.getComparator(1)` to get the second comparator (in this example the `7`).

Example finding duplicates in dictionary literals

If there are duplicate keys in a Python dictionary, then the second key will overwrite the first, which is almost certainly a mistake. We can find these duplicates with CodeQL, but the query is more complex than previous examples and will require us to write a `predicate` as a helper.

```
import python

predicate same_key(Expr k1, Expr k2) {
  k1.(Num).getN() = k2.(Num).getN()
  or
  k1.(StrConst).getText() = k2.(StrConst).getText()
}

from Dict d, Expr k1, Expr k2
where k1 = d.getAKey() and k2 = d.getAKey()
  and k1 != k2 and same_key(k1, k2)
select k1, "Duplicate key in dict literal"
```

See this in the query console on LGTM.com. When we ran this query on LGTM.com, the source code of the *saltstack/salt* project contained an example of duplicate dictionary keys. The results were also highlighted as alerts by the standard "Duplicate key in dict literal" query. Two of the other demo projects on LGTM.com refer to duplicate dictionary keys in library files. For more information, see Duplicate key in dict literal on LGTM.com.

The supporting predicate `same_key` checks that the keys have the same identifier. Separating this part of the logic into a supporting predicate, instead of directly including it in the query, makes it easier to understand the query as a whole. The casts defined in the predicate restrict the expression to the type specified and allow predicates to be called on the type that is cast-to. For example:

```
x = k1.(Num).getN()
```

is equivalent to

```
exists(Num num | num = k1 | x = num.getN())
```

The short version is usually used as this is easier to read.

Example finding Java-style getters

Returning to the example from "*Functions in Python*," the query identified all methods with a single line of code and a name starting with `get`.

```
import python

from Function f
where f.getName().matches("get%") and f.isMethod()
    and count(f.getAStmt()) = 1
select f, "This function is (probably) a getter."
```

This basic query can be improved by checking that the one line of code is a Java-style getter of the form `return self.attr`.

```
import python

from Function f, Return ret, Attribute attr, Name self
where f.getName().matches("get%") and f.isMethod()
    and ret = f.getStmt(0) and ret.getValue() = attr
    and attr.getObject() = self and self.getId() = "self"
select f, "This function is a Java-style getter."
```

See this in the query console on LGTM.com. Of the demo projects on LGTM.com, only the *openstack/nova* project has examples of functions that appear to be Java-style getters.

```
ret = f.getStmt(0) and ret.getValue() = attr
```

This condition checks that the first line in the method is a return statement and that the expression returned (`ret.getValue()`) is an `Attribute` expression. Note that the equality `ret.getValue() = attr` means that `ret.getValue()` is restricted to `Attribute`s, since `attr` is an `Attribute`.

```
attr.getObject() = self and self.getId() = "self"
```

This condition checks that the value of the attribute (the expression to the left of the dot in `value.attr`) is an access to a variable called `"self"`.

Class and function definitions

As Python is a dynamically typed language, class, and function definitions are executable statements. This means that a class statement is both a statement and a scope containing statements. To represent this cleanly the class definition is broken into a number of parts. At runtime, when a class definition is executed a class object is created and then assigned to a variable of the same name in the scope enclosing the class. This class is created from a code-object representing the source code for the body of the class. To represent this the `ClassDef` class (which represents a `class` statement) subclasses `Assign`. The `Class` class, which represents the body of the class, can be accessed via the `ClassDef.getDefinedClass()`. `FunctionDef` and `Function` are handled similarly.

Here is the relevant part of the class hierarchy:

- `Stmt`
  - `Assign`
    - `ClassDef`
    - `FunctionDef`
- `Scope`
  - `Class`
  - `Function`

Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"

5.6.7 Analyzing control flow in Python

You can write CodeQL queries to explore the control-flow graph of a Python program, for example, to discover unreachable code or mutually exclusive blocks of code.

About analyzing control flow

To analyze the control-flow graph of a `Scope` we can use the two CodeQL classes `ControlFlowNode` and `BasicBlock`. These classes allow you to ask such questions as "can you reach point A from point B?" or "Is it possible to reach point B *without* going through point A?". To report results we use the class `AstNode`, which represents a syntactic element and corresponds to the source code - allowing the results of the query to be more easily understood. For more information, see Control-flow graph on Wikipedia.

The `ControlFlowNode` class

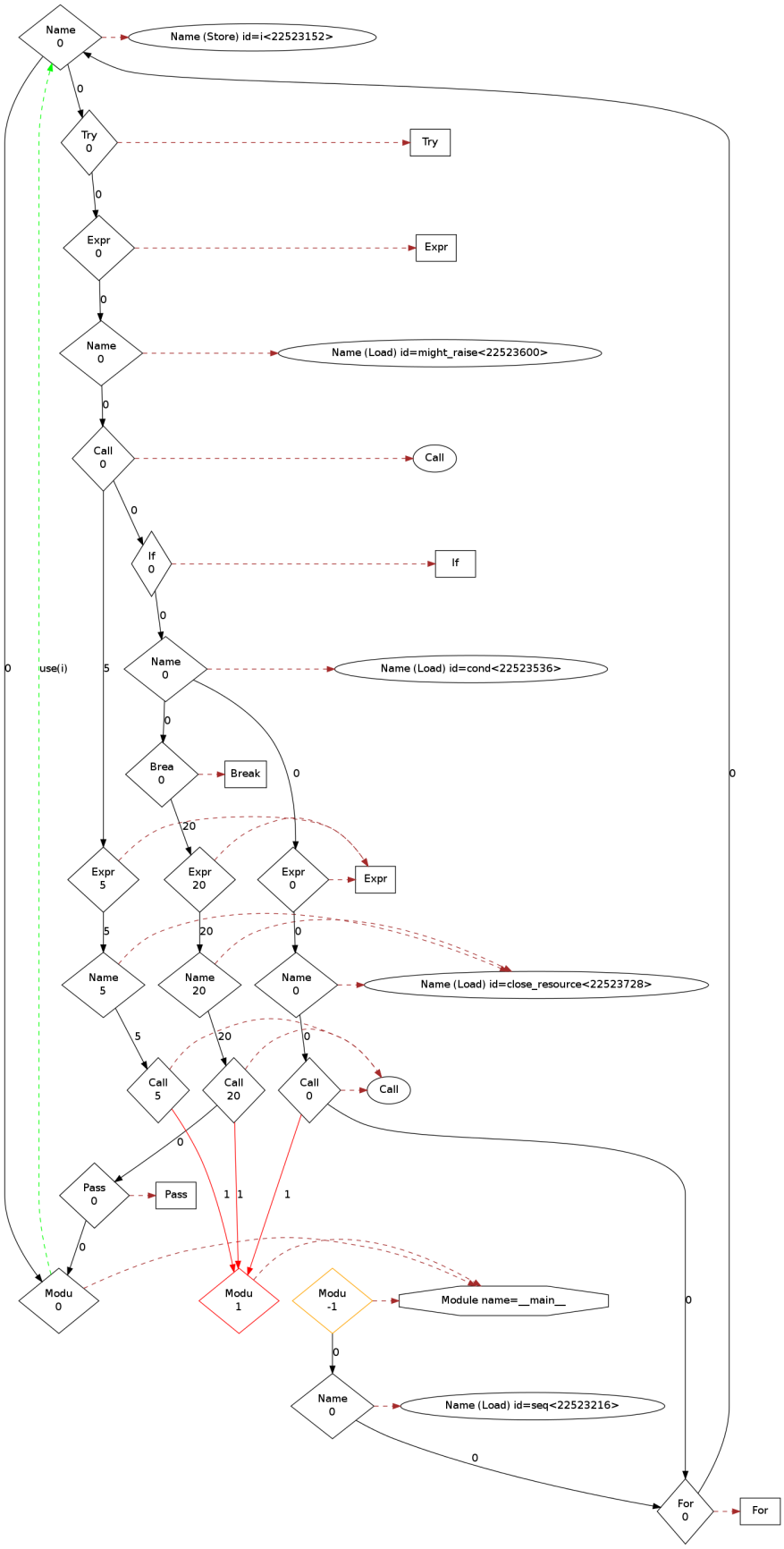
The `ControlFlowNode` class represents nodes in the control flow graph. There is a one-to-many relation between AST nodes and control flow nodes. Each syntactic element, the `AstNode`, maps to zero, one, or many `ControlFlowNode` classes, but each `ControlFlowNode` maps to exactly one `AstNode`.

To show why this complex relation is required consider the following Python code:

```
try:
    might_raise()
    if cond:
        break
finally:
    close_resource()
```

There are many paths through the above code. There are three different paths through the call to `close_resource()`; one normal path, one path that breaks out of the loop, and one path where an exception is raised by `might_raise()`.

An annotated flow graph:


Name 0
Name (Store) id=i<22523152>
Try 0
Try
Expr 0
Expr
Name 0
Name (Load) id=might_raise<22523600>
Call 0
Call
If 0
If
Name 0
Name (Load) id=cond<22523536>
Brea 0
Break
Expr 5
Expr 20
Expr 0
Expr
Name 5
Name 20
Name 0
Name (Load) id=close_resource<22523728>
Call 5
Call 20
Call 0
Call
Pass 0
Pass
Modu 0
Modu 1
Modu -1
Module name=__main__
Name 0
Name (Load) id=seq<22523216>
For 0
For
use(i)

The simplest use of the `ControlFlowNode` and `AstNode` classes is to find unreachable code. There is one `ControlFlowNode` per path through any `AstNode` and any `AstNode` that is unreachable has no paths flowing through it. Therefore, any `AstNode` without a corresponding `ControlFlowNode` is unreachable.

Example finding unreachable AST nodes

```
import python

from AstNode node
where not exists(node.getAFlowNode())
select node
```

See this in the query console on LGTM.com. The demo projects on LGTM.com all have some code that has no control flow node, and is therefore unreachable. However, since the `Module` class is also a subclass of the `AstNode` class, the query also finds any modules implemented in C or with no source code. Therefore, it is better to find all unreachable statements.

Example finding unreachable statements

```
import python

from Stmt s
where not exists(s.getAFlowNode())
select s
```

See this in the query console on LGTM.com. This query gives fewer results, but most of the projects have some unreachable nodes. These are also highlighted by the standard "Unreachable code" query. For more information, see Unreachable code on LGTM.com.

The `BasicBlock` class

The `BasicBlock` class represents a basic block of control flow nodes. The `BasicBlock` class is not that useful for writing queries directly, but is very useful for building complex analyses, such as data flow. The reason it is useful is that it shares many of the interesting properties of control flow nodes, such as, what can reach what, and what dominates what, but there are fewer basic blocks than control flow nodes - resulting in queries that are faster and use less memory. For more information, see Basic block and Dominator on Wikipedia.

Example finding mutually exclusive basic blocks

Suppose we have the following Python code:

```
if condition():
    return 0
pass
```

Can we determine that it is impossible to reach both the `return 0` statement and the `pass` statement in a single execution of this code? For two basic blocks to be mutually exclusive it must be impossible to reach either of them from the other. We can write:

```
import python

from BasicBlock b1, BasicBlock b2
where b1 != b2 and not b1.strictlyReaches(b2) and not b2.strictlyReaches(b1)
select b1, b2
```

However, by that definition, two basic blocks are mutually exclusive if they are in different scopes. To make the results more useful, we require that both basic blocks can be reached from the same function entry point:

```
exists(Function shared, BasicBlock entry |
    entry.contains(shared.getEntryNode()) and
    entry.strictlyReaches(b1) and entry.strictlyReaches(b2)
)
```

Combining these conditions we get:

Example finding mutually exclusive blocks within the same function

```
import python

from BasicBlock b1, BasicBlock b2
where b1 != b2 and not b1.strictlyReaches(b2) and not b2.strictlyReaches(b1) and
exists(Function shared, BasicBlock entry |
    entry.contains(shared.getEntryNode()) and
    entry.strictlyReaches(b1) and entry.strictlyReaches(b2)
)
select b1, b2
```

See this in the query console on LGTM.com. This typically gives a very large number of results, because it is a common occurrence in normal control flow. It is, however, an example of the sort of control-flow analysis that is possible. Control-flow analyses such as this are an important aid to data flow analysis. For more information, see "*Analyzing data flow in Python*."

Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- "*QL language reference*"
- "*CodeQL tools*"
- *Basic query for Python code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for Python*: When you need to analyze a Python program, you can make use of the large collection of classes in the CodeQL library for Python.
- *Analyzing data flow in Python*: You can use CodeQL to track the flow of data through a Python program to places where the data is used.
- *Using API graphs in Python*: API graphs are a uniform interface for referring to functions, classes, and methods defined in external libraries.

- *Functions in Python*: You can use syntactic classes from the standard CodeQL library to find Python functions and identify calls to them.
- *Expressions and statements in Python*: You can use syntactic classes from the CodeQL library to explore how Python expressions and statements are used in a codebase.
- *Analyzing control flow in Python*: You can write CodeQL queries to explore the control-flow graph of a Python program, for example, to discover unreachable code or mutually exclusive blocks of code.

5.7 CodeQL for Ruby

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Ruby codebases.

5.7.1 Basic query for Ruby code

Learn to write and run a simple CodeQL query.

About the query

The query we're going to run performs a basic search of the code for `if` expressions that are redundant, in the sense that they have an empty `then` branch. For example, code such as:

```
if error
  # Handle the error
```

Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Ruby** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

   ```
   import ruby

   from IfExpr ifexpr
   where
     not exists(ifexpr.getThen())
   select ifexpr, "This 'if' expression is redundant."
   ```

   LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifexpr` and is linked to the location in the source code of the project where `ifexpr` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis (...) at the bottom of the table indicates that the entire list is not displayed—click it to show more results.

6. If any matching code is found, click a link in the `ifexpr` column to view the `if` statement in the code viewer.

   The matching `if` expression is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import ruby`	Imports the standard CodeQL libraries for Ruby.	Every query begins with one or more `import` statements.
`from IfExpr ifexpr`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: an `IfExpr` variable for `if` expressions.
`where not exists(ifexpr.getThen())`	Defines a condition on the variables.	`ifexpr.getThen()`: gets the `then` branch of the `if` expression. `exists(...)`: requires that there is a matching element, in this case a `then` branch.
`select ifexpr, "This 'if' expression is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` expression with a string that explains the problem.

Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if option == "-verbose"
  # nothing to do - handled earlier
else
  error "unrecognized option"
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to select `if` statements where both the `then` and `else` branches are missing.

To exclude `if` statements that have an `else` branch:

1. Add the following to the where clause:

   ```
   and not exists(ifstmt.getElse())
   ```

   The `where` clause is now:

   ```
   where
      not exists(ifexpr.getThen()) and
      not exists(ifexpr.getElse())
   ```

2. Click **Run**.

   There are now fewer results because `if` expressions with an `else` branch are no longer included.

See this in the query console

Further reading

- CodeQL queries for Ruby
- Example queries for Ruby
- CodeQL library reference for Ruby
- "*QL language reference*"
- "*CodeQL tools*"

5.7.2 CodeQL library for Ruby

When you're analyzing a Ruby program, you can make use of the large collection of classes in the CodeQL library for Ruby.

Overview

CodeQL ships with an extensive library for analyzing Ruby code. The classes in this library present the data from a CodeQL database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of CodeQL modules, that is, files with the extension `.qll`. The module ruby.qll imports most other standard library modules, so you can include the complete library by beginning your query with:

```
import ruby
```

The CodeQL libraries model various aspects of Ruby code, depending on the type of query you want to write. For example the abstract syntax tree (AST) library is used for locating program elements, to match syntactic elements in the source code. This can be used to find values, patterns and structures.

The control flow graph (CFG) is imported using

```
import codeql.ruby.CFG
```

The CFG models the control flow between statements and expressions, for example whether one expression can flow to another expression, or whether an expression "dominates" another one, meaning that all paths to an expression must flow through another expression first.

The data flow library is imported using

```
import codeql.ruby.DataFlow
```

Data flow tracks the flow of data through the program, including through function calls (interprocedural data flow). Data flow is particularly useful for security queries, where untrusted data flows to vulnerable parts of the program to exploit it. Related to data flow, is the taint-tracking library, which finds how data can *influence* other values in a program, even when it is not copied exactly.

The API graphs library is used to locate methods in libraries. This is particuarly useful when locating particular functions or parameters that could be used as a source or sink of data in a security query.

To summarize, the main Ruby modules are:

Table 5: Main Ruby modules

Import	Description
`ruby`	The standard Ruby library
`codeql.ruby.AST`	The abstract syntax tree library (also imported by *ruby.qll*)
`codeql.ruby.ApiGraphs`	The API graphs library
`codeql.ruby.CFG`	The control flow graph library
`codeql.ruby.DataFlow`	The data flow library
`codeql.ruby.TaintTracking`	The taint tracking library

The CodeQL examples in this article are only excerpts and are not meant to represent complete queries.

Abstract syntax

The abstract syntax tree (AST) represents the elements of the source code organized into a tree. The AST viewer in Visual Studio Code shows the AST nodes, including the relevant CodeQL classes and predicates.

All CodeQL AST classes inherit from the *AstNode* class, which provides the following member predicates to all AST classes:

Table 6: Main predicates in `AstNode`

Predicate	Description
`getEnclosingModule(`	Gets the enclosing module, if any.
`getEnclosingMethod(`	Gets the enclosing method, if any.
`getLocation()`	Gets the location of this node.
`getAChild()`	Gets a child node of this node.
`getParent()`	Gets the parent of this *AstNode*, if this node is not a root node.
`getDesugared`	Gets the desugared version of this AST node, if any.
`isSynthesized()`	Holds if this node was synthesized to represent an implicit AST node not present in the source code.

Modules

Modules represent the main structural elements of Ruby programs, and include modules (`Module`), namespaces (`Namespace`) and classes (`ClassDeclaration`).

Table 7: Callable classes

CodeQL class	Description and selected predicates
`Module`	A representation of a runtime *module* or *class* value. • *getADeclaration()* - Gets a declaration • *getSuperClass()* - Gets the super class of this module, if any. • *getAPrependedModule()* - Gets a prepended module. • *getAnIncludedModule()* - Gets an included module.
`Namespace`	A class or module definition. • *getName()* - Gets the name of the module/class. • *getAMethod()*, *getMethod(name)* - Gets a method in this namespace. • *getAClass()*, *getClass(name)* - Gets a class in this namespace. • *getAModule()*, *getModule(name)* - Gets a module in this namespace.
`ClassDeclaration`	A class definition.
`SingletonClass`	A definition of a singleton class on an object.
`ModuleDeclaration`	A module definition.
`Toplevel`	The node representing the entire Ruby source file.

The following example lists all methods in the class *ApiController*:

```
import ruby

from ClassDeclaration m
where m.getName() = "ApiController"
select m, m.getAMethod()
```

Callables

Callables are elements that can be called, including methods and blocks.

Table 8: Callable classes

CodeQL class	Description and main predicates
`Callable`	A callable. • *getAParameter()* - gets a parameter of this callable. • *getParameter(n)* - gets the nth parameter of this callable.
`Private`	A call to `private`.
`Method`	A method. • *getName()* - gets the name of this method
`SingletonMethod`	A singleton method.
`Lambda`	A lambda (anonymous method).
`Block`	A block.
`DoBlock`	A block enclosed within *do* and *end*.
`BraceBlock`	A block defined using curly braces.

Parameters are the values that are passed into callables. Unlike other CodeQL language models, parameters in Ruby are not variables themselves, but can introduce variables into the callable. The variables of a parameter are given by the *getAVariable()* predicate.

Table 9: Parameter classes

CodeQL class	Description and main predicates
`Parameter`	A parameter. • *getCallable()* - Gets the callable that this parameter belongs to. • *getPosition()* - Gets the zero-based position of this parameter. • *getAVariable()*, *getVariable(name)* - Gets a variable introduced by this parameter.
`PatternParameter`	A parameter defined using a pattern.
`TuplePatternParameter`	A parameter defined using a tuple pattern.
`NamedParameter`	A named parameter. • *getName()*, *hasName(name)* - Gets the name of this parameter. • *getAnAccess()* - Gets an access to this parameter. • *getDefiningAccess()* - Gets the access that defines the underlying local variable.
`SimpleParameter`	A simple (normal) parameter.
`BlockParameter`	A parameter that is a block.
`HashSplatParameter`	A hash-splat (or double-splat) parameter.
`KeywordParameter`	A keyword parameter, including a default value if the parameter is optional. • *getDefaultValue()* - Gets the default value, i.e. the value assigned to the parameter when one is not provided by the caller.
`OptionalParameter`	An optional parameter. • *getDefaultValue()* - Gets the default value, i.e. the value assigned to the parameter when one is not provided by the caller.
`SplatParameter`	A splat parameter.

Example

```
import ruby

from Method m
where m.getName() = "show"
select m.getParameter(0)
```

Statements

Statements are the elements of code blocks. Statements that produce a value are called *expressions* and have CodeQL class *Expr*. The remaining statement types (that do not produce values) are listed below.

Table 10: Statement classes

CodeQL class	Description and main predicates
`Stmt`	The base class for all statements. • *getAControlFlowNode()* - Gets a control-flow node for this statement, if any. • *getEnclosingCallable()* - Gets the enclosing callable, if any.
`EmptyStmt`	An empty statement.
`BeginExpr`	A *begin* statement.
`BeginBlock`	A *BEGIN* block.
`EndBlock`	An *END* block.
`UndefStmt`	An *undef* statement.
`AliasStmt`	An *alias* statement.
`ReturningStmt`	A statement that may return a value: *return*, *break* and *next*.
`ReturnStmt`	A *return* statement.
`BreakStmt`	A *break* statement.
`NextStmt`	A *next* statement.
`RedoStmt`	A *redo* statement.
`RetryStmt`	A *retry* statement.

The following example finds all literals that are returned by a *return* statement.

```
import ruby

from ReturnStmt return, Literal lit
where lit.getParent() = return
select lit, "Returning a literal " + lit.getValueText()
```

Expressions

Expressions are types of statement that evaluate to a value. The CodeQL class *Expr* is the base class of all expression types.

Table 11: Expressions

CodeQL class	Description and main predicates
`Expr`	An expression. This is the root class for all expressions. • *getValueText()* - Gets the textual (constant) value of this expression, if any.
`Self`	A reference to the current object.
`Pair`	A pair expression.
`RescueClause`	A *rescue* clause.
`RescueModifierExpr`	An expression with a *rescue* modifier.
`StringConcatenation`	A concatenation of string literals. • *getConcatenatedValueText()* - Gets the result of concatenating all the string literals, if and only if they do not contain any interpolations.

Table 12: Statement sequences

CodeQL class	Description
`StmtSequence`	A sequence of expressions. • *getAStmt()*, *getStmt(n)* - Gets a statement in this sequence. • *isEmpty()* - Holds if this sequence has no statements. • *getNumberOfStatements()* - Gets the number of statements in this sequence.
`BodyStmt`	A sequence of statements representing the body of a method, class, module, or do-block. • *getARescue()*, *getRescue(n)* - Gets a rescue clause in this block. • *getElse()* - Gets the *else* clause in this block, if any. • *getEnsure()* - Gets the *ensure* clause in this block, if any.
`ParenthesizedExpr`	A parenthesized expression sequence, typically containing a single expression.

Literals are expressions that evaluate directly to the given value. The CodeQL Ruby library models all types of Ruby literal.

Table 13: Literals

CodeQL class	Description
`Literal`	A literal. This is the base class for all literals. • *getValueText()* - Gets the source text for this literal, if this is a simple literal.
`NumericLiteral`	A numerical literal. The literal types are `IntegerLiteral`, `FloatLiteral`, `RationalLiteral`, and `ComplexLiteral`.
`NilLiteral`	A *nil* literal.
`BooleanLiteral`	A Boolean value. The classes `TrueLiteral` and `FalseLiteral` match *true* and *false* respectively.
`StringComponent`	A component of a string. Either a `StringTextComponent`, `StringEscapeSequenceComponent`, or `StringInterpolationComponent`.
`RegExpLiteral`	A regular expression literal.
`SymbolLiteral`	A symbol literal.
`SubshellLiteral`	A subshell literal.
`CharacterLiteral`	A character literal.
`ArrayLiteral`	An array literal.
`HashLiteral`	A hash literal.
`RangeLiteral`	A range literal.
`MethodName`	A method name literal.

The following example defines a string literal class containing the text "username":

```
class UsernameLiteral extends Literal
{
  UsernameLiteral() { this.getValueText().toLowerCase().matches("%username%") }
}
```

Operations are types of expression that typically perform some sort of calculation. Most operations are `MethodCalls` because often there is an underlying call to the operation.

Table 14: Operations

CodeQL class	Description
Opera	An operation.
Unary(	A unary operation. Types of unary operation include UnaryLogicalOperation, NotExpr, UnaryPlusExpr, UnaryMinusExpr, SplatExpr, HashSplatExpr, UnaryBitwiseOperation, and ComplementExpr.
Defin	A call to the special *defined?* operator
Binar	A binary operation, that includes many other operation categories such as BinaryArithmeticOperation, BinaryBitwiseOperation, ComparisonOperation, SpaceshipExpr, and Assignment.
Binar	A binary arithmetic operation. Includes: AddExpr, SubExpr, MulExpr, DivExpr, ModuloExpr, and ExponentExpr.
Binar	A binary logical operation. Includes: LogicalAndExpr and LogicalOrExpr.
Binar	A binary bitwise operation. Includes: LShiftExpr, RShiftExpr, BitwiseAndExpr, BitwiseOrExpr, and BitwiseXorExpr.
Compa	A comparison operation, including the classes EqualityOperation, EqExpr, NEExpr, CaseEqExpr, RelationalOperation, GTExpr, GEExpr, LTExpr, and LEExpr.
RegEx	A regexp match expression.
NoRegl	A regexp-doesn't-match expression.
Assig	An assignment. Assignments are simple assignments (AssignExpr), or assignment operations (AssignOperation). The assignment arithmetic operations (AssignArithmeticOperation) are AssignAddExpr, AssignSubExpr, AssignMulExpr, AssignDivExpr, AssignModuloExpr, and AssignExponentExpr. The assignment logical operations (AssignLogicalOperation) are AssignLogicalAndExpr and AssignLogicalOrExpr. The assignment bitwise operations (AssignBitwiseOperation) are AssignLShiftExpr, AssignRShiftExpr, AssignBitwiseAndExpr, AssignBitwiseOrExpr, and AssignBitwiseXorExpr.

The following example finds "chained assignments" (of the form A=B=C):

```
import ruby

from Assignment op
where op.getRightOperand() instanceof Assignment
select op, "This is a chained assignment."
```

Calls pass control to another function, include explicit method calls (MethodCall), but also include other types of call such as *super* calls or *yield* calls.

Table 15: Calls

CodeQL class	Description and main predicates
`Call`	A call. • *getArgument(n)*, *getAnArgument()*, *getKeywordArgument(keyword)* - Gets an argument of this call. • *getATarget()* - Gets a potential target of this call, if any.
`MethodCall`	A method call. • *getReceiver()* - Gets the receiver of this call, if any. This is the object being invoked. • *getMethodName()* - Gets the name of the method being called. • *getBlock()* - Gets the block of this method call, if any.
`SetterMethodCall`	A call to a setter method.
`ElementReference`	An element reference; a call to the *[]* method.
`YieldCall`	A call to *yield*.
`SuperCall`	A call to *super*.
`BlockArgument`	A block argument in a method call.

The following example finds all method calls to a method called *delete*.

```
import ruby

from MethodCall call
where call.getMethodName() = "delete"
select call, "Call to 'delete'."
```

Control expressions are expressions used for control flow. They are classed as expressions because they can produce a value.

Table 16: Control expressions

CodeQL class	Description and main predicates
`ControlExpr`	A control expression, such as a *case*, *if*, *unless*, ternary-if (*?:*), *while*, *until* (including expression-modifier variants), and *for*.
`ConditionalExpr`	A conditional expression. • *getCondition()* - Gets the condition expression.
`IfExpr`	An *if* or *elsif* expression. • *getThen()* - Gets the *then* branch. • *getElse()* - Gets the *elseif* or *else* branch.
`UnlessExpr`	An *unless* expression.
`IfModifierExpr`	An expression modified using *if*.
`UnlessModifierExpr`	An expression modified using *unless*.
`TernaryIfExpr`	A conditional expression using the ternary (*?:*) operator.
`CaseExpr`	A *case* expression.
`WhenExpr`	A *when* branch of a *case* expression.
`Loop`	A loop. That is, a *for* loop, a *while* or *until* loop, or their expression-modifier variants.
`ConditionalLoop`	A loop using a condition expression. That is, a *while* or *until* loop, or their expression-modifier variants. • *getCondition()* - Gets the condition expression of this loop.
`WhileExpr`	A *while* loop.
`UntilExpr`	An *until* loop.
`WhileModifierExpr`	An expression looped using the *while* modifier.
`UntilModifierExpr`	An expression looped using the *until* modifier.
`ForExpr`	A *for* loop.

The following example finds *if*-expressions that are missing a *then* branch.

```
import ruby

from IfExpr expr
where not exists(expr.getThen())
select expr, "This if-expression is redundant."
```

Variables

Variables are names that hold values in a Ruby program. If you want to query *any* type of variable, then use the `Variable` class, otherwise use one of the subclasses `LocalVariable`, `InstanceVariable`, `ClassVariable` or `GlobalVariable`.

Local variables have the scope of a single function or block, instance variables have the scope of an object (like member variables), *class* variables have the scope of a class and are shared between all instances of that class (like static variables), and *global* variables have the scope of the entire program.

Table 17: Variable classes

CodeQL class	Description and main predicates
`Variable`	A variable declared in a scope. • *getName()*, *hasName(name)* - Gets the name of this variable. • *getDeclaringScope()* - Gets the scope this variable is declared in. • *getAnAccess()* - Gets an access to this variable.
`LocalVariable`	A local variable.
`InstanceVariable`	An instance variable.
`ClassVariable`	A class variable.
`GlobalVariable`	A global variable.

The following example finds all class variables in the class *StaticController*:

```
import ruby

from ClassDeclaration cd, ClassVariable v
where
  v.getDeclaringScope() = cd and
  cd.getName() = "StaticController"
select v, "This is a static variable in 'StaticController'."
```

Variable accesses are the uses of a variable in the source code. Note that variables, and *uses* of variables are different concepts. Variables are modelled using the `Variable` class, whereas uses of the variable are modelled using the `VariableAccess` class. `Variable.getAnAccess()` gets the accesses of a variable.

Variable accesses come in two types: *reads* of the variable (a `ReadAccess`), and *writes* to the variable (a `WriteAccess`). Accesses are a type of expression, so extend the `Expr` class.

Table 18: Variable access classes

CodeQL class	Description and main predicates
`VariableAccess`	An access to a variable. • *getVariable()* - Gets the variable that is accessed.
`VariableReadAccess`	An access to a variable where the value is read.
`VariableWriteAccess`	An access to a variable where the value is updated.
`LocalVariableAccess`	An access to a local variable.
`LocalVariableWriteAccess`	An access to a local variable where the value is updated.
`LocalVariableReadAccess`	An access to a local variable where the value is read.
`GlobalVariableAccess`	An access to a global variable where the value is updated.
`InstanceVariableAccess`	An access to a global variable where the value is read.
`InstanceVariableReadAccess`	An access to an instance variable.
`InstanceVariableWriteAccess`	An access to an instance variable where the value is updated.
`ClassVariableAccess`	An access to a class variable.
`ClassVariableWriteAccess`	An access to a class variable where the value is updated.
`ClassVariableReadAccess`	An access to a class variable where the value is read.

The following example finds writes to class variables in the class *StaticController*:

```
import ruby

from ClassVariableWriteAccess write, ClassDeclaration cd, ClassVariable v
where
  v.getDeclaringScope() = cd and
  cd.getName() = "StaticController" and
  write.getVariable() = v
select write, "'StaticController' class variable is written here."
```

- *Basic query for Ruby code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for Ruby*: When you're analyzing a Ruby program, you can make use of the large collection of classes in the CodeQL library for Ruby.

  Note

  CodeQL analysis for Ruby is currently in beta. During the beta, analysis of Ruby code, and the accompanying documentation, will not be as comprehensive as for other languages.

CHAPTER

SIX

QL LANGUAGE REFERENCE

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

- *About the QL language*: QL is the powerful query language that underlies CodeQL, which is used to analyze code.
- *Predicates*: Predicates are used to describe the logical relations that make up a QL program.
- *Queries*: Queries are the output of a QL program. They evaluate to sets of results.
- *Types*: QL is a statically typed language, so each variable must have a declared type.
- *Modules*: Modules provide a way of organizing QL code by grouping together related types, predicates, and other modules.
- *Aliases*: An alias is an alternative name for an existing QL entity.
- *Variables*: Variables in QL are used in a similar way to variables in algebra or logic. They represent sets of values, and those values are usually restricted by a formula.
- *Expressions*: An expression evaluates to a set of values and has a type.
- *Formulas*: Formulas define logical relations between the free variables used in expressions.
- *Annotations*: An annotation is a string that you can place directly before the declaration of a QL entity or name.
- *Recursion*: QL provides strong support for recursion. A predicate in QL is said to be recursive if it depends, directly or indirectly, on itself.
- *Lexical syntax*: The QL syntax includes different kinds of keywords, identifiers, and comments.
- *Name resolution*: The QL compiler resolves names to program elements.
- *Evaluation of QL programs*: A QL program is evaluated in a number of different steps.
- *QL language specification*: A formal specification for the QL language. It provides a comprehensive reference for terminology, syntax, and other technical details about QL.

6.1 About the QL language

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

6.1.1 About query languages and databases

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

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

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

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

6.1.2 Properties of QL

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

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

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

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

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

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

For a formal specification of the QL language, see the "*QL language specification*."

6.1.3 QL and object orientation

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

6.1.4 QL and general purpose programming languages

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

- QL does not have any imperative features such as assignments to variables or file system operations.
- QL operates on sets of tuples and a query can be viewed as a complex sequence of set operations that defines the result of the query.
- QL's set-based semantics makes it very natural to process collections of values without having to worry about efficiently storing, indexing and traversing them.
- In object oriented programming languages, instantiating a class involves creating an object by allocating physical memory to hold the state of that instance of the class. In QL, classes are just logical properties describing sets of already existing values.

6.1.5 Further reading

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

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

6.2 Predicates

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

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

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

predicate hasCapital(string country, string capital) {
  country = "Belgium" and capital = "Brussels"
```

(continues on next page)

(continued from previous page)

```
  or
  country = "Germany" and capital = "Berlin"
  or
  country = "France" and capital = "Paris"
}
```

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

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

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

6.2.1 Defining a predicate

When defining a predicate, you should specify:

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

   Note

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

Predicates without result

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

For example:

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

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

Predicates with result

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

For example:

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

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

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

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

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

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

In this case:

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

6.2.2 Recursive predicates

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

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

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

(continues on next page)

(continued from previous page)

```
  country = "Germany" and result = "Belgium"
  or
  country = getANeighbor(result)
}
```

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

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

6.2.3 Kinds of predicates

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

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

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

Here is an example showing a predicate of each kind:

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

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

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

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

6.2.4 Binding behavior

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

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

Here are a few examples of infinite predicates:

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

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

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

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

Binding sets

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

For example:

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

6.2.5 Database predicates

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

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

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

6.3 Queries

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

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

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

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

6.3.1 Select clauses

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

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

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

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

- The `as` keyword, followed by a name. This gives a "label" to a column of results, and allows you to use them in subsequent select expressions.
- The `order by` keywords, followed by the name of a result column, and optionally the keyword `asc` or `desc`. This determines the order in which to display the results.

For example:

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

This select clause returns the following results:

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

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

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

6.3.2 Query predicates

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

For example:

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

This predicate returns the following results:

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

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

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

In contrast, the select clause is like an anonymous predicate, so you can't call it later.

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

6.4 Types

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

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

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

6.4.1 Primitive types

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

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

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

6.4.2 Classes

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

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

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

A class in QL doesn't "create" a new object, it just represents a logical property. A value is in a particular class if it satisfies that logical property.

Defining a class

To define a class, you write:

1. The keyword `class`.
2. The name of the class. This is an identifier starting with an uppercase letter.
3. The supertypes that the class is derived from via *extends* and/or *instanceof*
4. The *body of the class*, enclosed in braces.

For example:

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

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

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

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

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

A class can extend multiple types. For more information, see "*Multiple inheritance*." Classes can also specialise other types without extending the class interface via *instanceof*, see "*Non-extending subtypes*.".

To be valid, a class:

- Must not extend itself.
- Must not extend a *final* class.
- Must not extend types that are incompatible. For more information, see "*Type compatibility*."

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

Class bodies

The body of a class can contain:

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

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

Characteristic predicates

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

Member predicates

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

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

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

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

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

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

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

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

Fields

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

For example:

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

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

  SmallInt getADivisor() { result = divisor }
}

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

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

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

Concrete classes

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

Abstract classes

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

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

Example

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

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

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

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

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

Overriding member predicates

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

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

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

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

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

Now, consider the following query:

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

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

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

In QL, unlike other object-oriented languages, different subtypes of the same types don't need to be disjoint. For example, you could define another subclass of `OneTwoThree`, which overlaps with `OneTwo`:

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

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

(continues on next page)

(continued from previous page)

```
  }
}
```

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

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

Multiple inheritance

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

For example, using the definitions from the above section:

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

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

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

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

Non-extending subtypes

Besides extending base types, classes can also declare *instanceof* relationships with other types. Declaring a class as *instanceof Foo* is roughly equivalent to saying *this instanceof Foo* in the characteristic predicate. The main differences are that you can call methods on Bar via *super* and you can get better optimisation.

```
class Foo extends int {
  Foo() { this in [1 .. 10] }

  string foo_method() { result = "foo" }
}

class Bar instanceof Foo {
  string toString() { result = super.foo_method() }
}
```

In this example, the characteristic predicate from *Foo* also applies to *Bar*. However, *foo_method* is not exposed in *Bar*, so the query *select any(Bar b).foo_method()* results in a compile time error. Note from the example that it is still possible to access methods from instanceof supertypes from within the specialising class with the *super* keyword.

Crucially, the instanceof **supertypes** are not **base types**. This means that these supertypes do not participate in overriding, and any fields of such supertypes are not part of the new class. This has implications on method resolution when complex class hierarchies are involved. The following example demonstrates this.

```
class Interface extends int {
  Interface() { this in [1 .. 10] }
  string foo() { result = "" }
}

class Foo extends int {
  Foo() { this in [1 .. 5] }
  string foo() { result = "foo" }
}

class Bar extends Interface instanceof Foo {
  override string foo() { result = "bar" }
}
```

Here, the method *Bar::foo* does not override *Foo::foo*. Instead, it overrides only *Interface::foo*. This means that *select any(Foo f).foo()* yields only *foo*. Had *Bar* been defined as *extends Foo*, then *select any(Foo b)* would yield *bar*.

6.4.3 Character types and class domain types

You can't refer to these types directly, but each class in QL implicitly defines a character type and a class domain type. (These are rather more subtle concepts and don't appear very often in practical query writing.)

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

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

6.4.4 Algebraic datatypes

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

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

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

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

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

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

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

Defining an algebraic datatype

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

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

The branch definitions have the following form:

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

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

For example, the following algebraic datatype has three branches:

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

Standard pattern for using algebraic datatypes

Algebraic datatypes are different from *classes*. In particular, algebraic datatypes don't have a `toString()` member predicate, so you can't use them in a *select clause*.

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
    or
    this = TTaintedValue() and result = "tainted"
  }
}

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

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

6.4.5 Type unions

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

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

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

```
newtype InitialValueSource =
  ExplicitInitialization(VarDecl v) { exists(v.getInitializer()) } or
  ParameterPassing(Call c, int pos) { exists(c.getParameter(pos)) } or
  UnknownInitialGarbage(VarDecl v) { not exists(DefiniteInitialization di | v =␣
↪target(di)) }

class DefiniteInitialization = ParameterPassing or ExplicitInitialization;

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

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

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

(continues on next page)

(continued from previous page)

```
  }
  // ...
}
```

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

6.4.6 Database types

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

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

6.4.7 Type compatibility

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

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

The universes in QL are:

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

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

- A class can't extend multiple primitive types.
- A class can't extend multiple different database types.
- A class can't extend multiple different branches of an algebraic datatype.

6.5 Modules

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

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

6.5.1 Defining a module

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

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

(continues on next page)

(continued from previous page)

```
    }
  }
}
```

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

`.ql` or `.qll` files also implicitly define modules. For more information, see "*Kinds of modules*."

You can also annotate a module. For more information, see of "*Overview of annotations*."

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

6.5.2 Kinds of modules

File modules

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

Library modules

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

For example, consider the following QL library:

OneTwoThreeLib.qll

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

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

Query modules

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

Query modules are slightly different from other modules:

- A query module can't be imported.
- A query module must have at least one query in its *namespace*. This is usually a *select clause*, but can also be a *query predicate*.

For example:

OneTwoQuery.ql

```
import OneTwoThreeLib

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

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

Explicit modules

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

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

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

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

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

6.5.3 Module bodies

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

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

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

6.5.4 Importing modules

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

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

Import statements

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

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

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

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

The `<module_expression>` itself can be a module name, a selection, or a qualified reference. For more information, see "*Name resolution*."

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

6.6 Aliases

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

Once you've defined an alias, you can use that new name to refer to the entity in the current module's *namespace*.

6.6.1 Defining an alias

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

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

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

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

Module aliases

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

```
module ModAlias = ModuleName;
```

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

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

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

Type aliases

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

```
class TypeAlias = TypeName;
```

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

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

```
class bool = boolean;
```

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

```
import OneTwoThreeLib

class OT = M::OneTwo;

...

from OT ot
select ot
```

Predicate aliases

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

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

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

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

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

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

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

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

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

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

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

6.7 Variables

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

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

6.7.1 Declaring a variable

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

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

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

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

For example, consider the following select clause:

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

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

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

```
from int i
select i
```

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

6.7.2 Free and bound variables

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

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

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

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

(i + 7) * 3

x.sqrt()
```

The first expression doesn't have any variables. It finds the (zero-based) indices of where `"l"` occurs in the string `"hello"`, so it evaluates to `2` and `3`.

The second expression evaluates to `-3`, the minimum value in the range `[-3 .. 3]`. Although this expression uses a variable `f`, it is just a placeholder or "dummy" variable, and you can't assign any values to it. You could replace `f` with a different variable without changing the meaning of the expression. For example, `min(float f | f in [-3 .. 3])` is always equal to `min(float other | other in [-3 .. 3])`. This is an example of a **bound variable**.

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

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

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

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

(i + 7) * 3 instanceof int

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

The first formula doesn't contain any variables, and it never holds (since `"hello".indexOf("l")` has values `2` and `3`, never `1`).

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

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

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

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

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

6.8 Expressions

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

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

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

6.8.1 Variable references

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

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

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

6.8.2 Literals

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

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

```
0
42
-2048
```

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

```
2.0
123.456
-100.5
```

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

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

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

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

  The following string formats are recognized as dates:

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

6.8.3 Parenthesized expressions

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

6.8.4 Ranges

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

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

6.8.5 Set literal expressions

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

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

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

6.8.6 Super expressions

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

In the following example, the class `C` inherits two definitions of the predicate `getANumber()`—one from `A` and one from `B`. Instead of overriding both definitions, it uses the definition from `B`.

```
class A extends int {
  A() { this = 1 }
  int getANumber() { result = 2 }
}

class B extends int {
```

(continues on next page)

(continued from previous page)

```
  B() { this = 1 }
  int getANumber() { result = 3 }
}

class C extends A, B {
  // Need to define `int getANumber()`; otherwise it would be ambiguous
  int getANumber() {
    result = B.super.getANumber()
  }
}

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

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

6.8.7 Calls to predicates (with result)

Calls to *predicates with results* are themselves expressions, unlike calls to *predicates without results* which are formulas. For more information, see "*Calls to predicates*."

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

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

6.8.8 Aggregations

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

The general syntax is:

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

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

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

To specify a different order, follow `<expression>` with the keywords `order by`, then one or more comma-separated expressions that specify the order, and optionally the keyword `asc` or `desc` after each expression (to determine whether to order the expression in ascending or descending order). If you don't specify an ordering, it defaults to `asc`. For example, `order by o.getName() asc, o.getSize() desc` might be used to order some object by name, breaking ties by descending size.

The following aggregates are available in QL:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note

- Rank indices start at 1, so `rank[0](...)` has no result.
- `rank[1](...)` is the same as `min(...)`.

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

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

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

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

Evaluation of aggregates

In general, aggregate evaluation involves the following steps:

1. Determine the input variables: these are the aggregation variables declared in `<variable declarations>` and also the variables declared outside of the aggregate that are used in some component of the aggregate.
2. Generate all possible distinct tuples (combinations) of the values of input variables such that the `<formula>` holds true. Note that the same value of an aggregate variable may appear in multiple distinct tuples. All such occurrences of the same value are treated as distinct occurrences when processing tuples.
3. Apply `<expression>` on each tuple and collect the generated (distinct) values. The application of `<expression>` on a tuple may result in generating more than one value.
4. Apply the aggregation function on the values generated in step 3 to compute the final result.

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

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

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

   30 tuples are generated in this step.
3. Apply the `<expression>` `i` on all tuples. This means selecting all values of `i` from all tuples: `0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4`.
4. Apply the aggregation function `sum` on the above values to get the final result `60`.

If we change <expression> to `i + j` in the above query, the query result is `135` since applying `i + j` on all tuples results in following values: `0, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6, 2, 3, 4, 5, 6, 7, 3, 4, 5, 6, 7, 8, 4, 5, 6, 7, 8, 9`.

Next, consider the following query:

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

1. `s` is the input variable of the aggregate.
2. A single tuple `"hello"` is generated in this step.
3. The <expression> `charAt(_)` is applied on this tuple. The underscore `_` in `charAt(_)` is a *don't-care expression*, which represents any value. `s.charAt(_)` generates four distinct values `h, e, l, o`.
4. Finally, `count` is applied on these values, and the query returns `4`.

Omitting parts of an aggregation

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

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

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

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

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

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

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

3. As a special case, you can omit the <expression> part from `count` even if there is more than one aggregation variable. In such a case, it counts the number of distinct tuples of aggregation variables that satisfy the formula. In other words, the expression part is considered to be the constant `1`. For example, the following aggregations are equivalent:

   ```
   count(int i, int j | i in [1 .. 3] and j in [1 .. 3] | 1)
   count(int i, int j | i in [1 .. 3] and j in [1 .. 3])
   ```

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

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

   This is useful if you don't want to restrict the aggregation variables any further. For example, the following aggregation returns the maximum number of lines across all files:

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

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

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

Monotonic aggregates

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

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

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

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

Example of monotonic aggregates

Consider this query:

```
string getPerson() { result = "Alice" or
                     result = "Bob" or
                     result = "Charles" or
                     result = "Diane"
                   }
string getFruit(string p) { p = "Alice"   and result = "Orange" or
                            p = "Alice"   and result = "Apple" or
                            p = "Bob"     and result = "Apple" or
                            p = "Charles" and result = "Apple" or
                            p = "Charles" and result = "Banana"
                          }
int getPrice(string f) { f = "Apple"  and result = 100 or
                         f = "Orange" and result = 100 or
                         f = "Orange" and result =   1
                       }

predicate nonmono(string p, int cost) {
  p = getPerson() and cost = sum(string f | f = getFruit(p) | getPrice(f))
}

language[monotonicAggregates]
predicate mono(string p, int cost) {
  p = getPerson() and cost = sum(string f | f = getFruit(p) | getPrice(f))
```

(continues on next page)

(continued from previous page)

```
}

from string variant, string person, int cost
where variant = "default"  and nonmono(person, cost) or
      variant = "monotonic" and mono(person, cost)
select variant, person, cost
order by variant, person
```

The query produces these results:

variant	person	cost
default	Alice	201
default	Bob	100
default	Charles	100
default	Diane	0
monotonic	Alice	101
monotonic	Alice	200
monotonic	Bob	100
monotonic	Diane	0

The two variants of the aggregate semantics differ in what happens when `getPrice(f)` has either multiple results or no results for a given `f`.

In this query, oranges are available at two different prices, and the default `sum` aggregate returns a single line where Alice buys an orange at a price of 100, another orange at a price of 1, and an apple at a price of 100, totalling 201. On the other hand, in the the *monotonic* semantics for `sum`, Alice always buys one orange and one apple, and a line of output is produced for each *way* she can complete her shopping list.

If there had been two different prices for apples too, the monotonic `sum` would have produced *four* output lines for Alice.

Charles wants to buy a banana, which is not for sale at all. In the default case, the sum produced for Charles includes the cost of the apple he *can* buy, but there's no line for Charles in the monontonic `sum` output, because there *is no way* for Charles to buy one apple plus one banana.

(Diane buys no fruit at all, and in both variants her total cost is 0. The `strictsum` aggregate would have excluded her from the results in both cases).

In actual QL practice, it is quite rare to use monotonic aggregates with the *goal* of having multiple output lines, as in the "Alice" case of this example. The more significant point is the "Charles" case: As long as there's no price for bananas, no output is produced for him. This means that if we later do learn of a banana price, we don't need to *remove* any output tuple already produced. The importance of this is that the monotonic aggregate behavior works well with a fixpoint-based semantics for recursion, so it will be meaningul to let the `getPrice` predicate be mutually recursive with the count aggregate itself. (On the other hand, `getFruit` still cannot be allowed to be recursive, because adding another fruit to someone's shopping list would invalidate the total costs we already knew for them).

This opportunity to use recursion is the main practical reason for requesting monotonic semantics of aggregates.

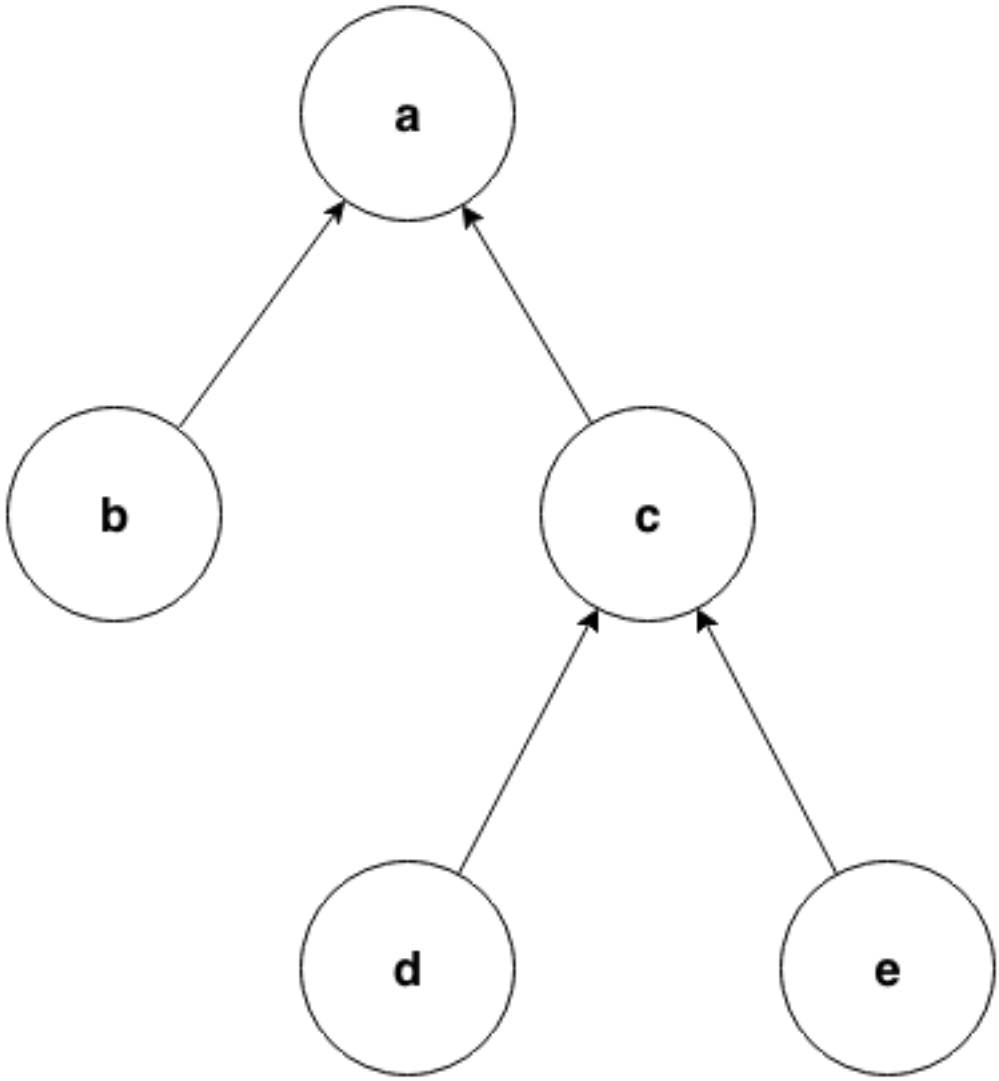
Recursive monotonic aggregates

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

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

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

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



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

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

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

6.8.9 Any

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

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

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

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

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

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

Expression	Values
`any(File f)`	all `File`s in the database
`any(Element e | e.getName())`	the names of all `Element`s in the database
`any(int i | i = [0 .. 3])`	the integers `0`, 1, 2, and 3
`any(int i | i = [0 .. 3] | i * i)`	the integers `0`, 1, 4, and 9

Note

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

6.8.10 Unary operations

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

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

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

6.8.11 Binary operations

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

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

You can use the following binary operators in QL:

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

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

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

6.8.12 Casts

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

You can write a cast in two ways:

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

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

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

```
import java

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

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

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

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

6.8.13 Don't-care expressions

This is an expression written as a single underscore `_`. It represents any value. (You "don't care" what the value is.)

Unlike other expressions, a don't-care expression does not have a type. In practice, this means that `_` doesn't have any *member predicates*, so you can't call `_.somePredicate()`.

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

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

The `charAt(int i)` predicate is defined on strings and usually takes an `int` argument. Here the don't care expression `_` is used to tell the query to select characters at every possible index. The query returns the values `h`, `e`, `l`, and `o`.

6.9 Formulas

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

Depending on the values assigned to those *free variables*, a formula can be true or false. When a formula is true, we often say that the formula *holds*. For example, the formula `x = 4 + 5` holds if the value `9` is assigned to `x`, but it doesn't hold for other assignments to `x`. Some formulas don't have any free variables. For example `1 < 2` always holds, and `1 > 2` never holds.

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

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

6.9.1 Comparisons

A comparison formula is of the form:

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

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

Order

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

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

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

Equality

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

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

Name	Symbol
Equal to	=
Not equal to	!=

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

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

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

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

Examples

1. **If both expressions have a single value (for example `1` and `0`), then comparison is straightforward:**
   - `1 != 0` holds.
   - `1 = 0` doesn't hold.
   - `not 1 = 0` holds.
2. **Now compare `1` and `[1 .. 2]`:**
   - `1 != [1 .. 2]` holds, because `1 != 2`.
   - `1 = [1 .. 2]` holds, because `1 = 1`.
   - `not 1 = [1 .. 2]` doesn't hold, because there is a common value (`1`).
3. **Compare `1` and `none()` (the "empty set"):**
   - `1 != none()` doesn't hold, because there are no values in `none()`, so no values that are not equal to `1`.
   - `1 = none()` also doesn't hold, because there are no values in `none()`, so no values that are equal to `1`.
   - `not 1 = none()` holds, because there are no common values.

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

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

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

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

This is equivalent to `forall( a, b | a in A and b in B | a != b )`, which is very different from the first formula.

6.9.2 Type checks

A type check is a formula that looks like:

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

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

6.9.3 Range checks

A range check is a formula that looks like:

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

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

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

6.9.4 Calls to predicates

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

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

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

The predicate reference must resolve to exactly one predicate. For more information about how a predicate reference is resolved, see "*Name resolution*."

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

It is also possible to call a predicate with result. This kind of call is an expression in QL, instead of a formula. For more information, see "*Calls to predicates (with result)*."

6.9.5 Parenthesized formulas

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

6.9.6 Quantified formulas

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

Explicit quantifiers

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

exists

This quantifier has the following syntax:

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

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

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

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

forall

This quantifier has the following syntax:

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

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

For example, `forall(int i | i instanceof OneTwoThree | i < 5)` holds if all integers that are in the class `OneTwoThree` are also less than `5`. In other words, if there is a value in `OneTwoThree` that is greater than or equal to `5`, then the formula doesn't hold.

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

forex

This quantifier has the following syntax:

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

This quantifier exists as a shorthand for:

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

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

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

Implicit quantifiers

Implicitly quantified variables can be introduced using "don't care expressions." These are used when you need to introduce a variable to use as an argument to a predicate call, but don't care about its value. For further information, see "*Don't-care expressions*."

6.9.7 Logical connectives

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

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

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

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

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

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

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

`not`

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

`not A` holds exactly when `A` doesn't hold.

Example

The following query selects files that are not HTML files.

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

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

if ... then ... else

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

Example

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

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

and

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

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

Example

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

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

or

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

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

Example

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

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

implies

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

Example

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

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

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

6.10 Annotations

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

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

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

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

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

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

Here is an explicit example:

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

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

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

6.10.1 Overview of annotations

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

abstract

Available for: *classes*, *member predicates*

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

For information about **abstract classes**, see "*Classes*."

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

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

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

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

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

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

cached

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

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

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

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

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

deprecated

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

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

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

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

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

external

Available for: *non-member predicates*

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

transient

Available for: *non-member predicates*

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

final

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

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

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

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

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

library

Available for: *classes*

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

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

override

Available for: *member predicates*, *fields*

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

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

private

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

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

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

query

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

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

Compiler pragmas

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

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

Inlining

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

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

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

pragma[inline]

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

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

pragma[noinline]

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

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

pragma[nomagic]

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

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

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

Note that `nomagic` implies `noinline`.

pragma[noopt]

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

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

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

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

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

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

   For example, suppose you have the following definitions:

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

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

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

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

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

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

pragma[only_bind_out]

Available for: *expressions*

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

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

For more information, see "*Binding*."

`pragma[only_bind_into]`

Available for: *expressions*

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

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

For more information, see "*Binding*."

Language pragmas

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

`language[monotonicAggregates]`

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

For more information, see "*Monotonic aggregates*."

Binding sets

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

`bindingset[...]`

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

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

- When you annotate a predicate, each variable must be an argument of the predicate, possibly including `this` (for characteristic predicates and member predicates) and `result` (for predicates that return a result). For more information, see "*Binding behavior*."
- When you annotate a class, each variable must be `this` or a field in the class. Binding sets for classes are supported from release 2.3.0 of the CodeQL CLI, and release 1.26 of LGTM Enterprise.

6.11 Recursion

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

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

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

6.11.1 Examples of recursive predicates

Here are a few examples of recursive predicates in QL:

Counting from 0 to 100

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

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

select getANumber()
```

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

Mutual recursion

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

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

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

select getAnEven()
```

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

Transitive closures

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

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

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

1. **Transitive closure** +

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

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

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

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

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

2. **Reflexive transitive closure** *

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

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

   In this case, the explicit definition looks like this:

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

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

6.11.2 Restrictions and common errors

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

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

Empty recursion

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

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

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

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

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

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

Non-monotonic recursion

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

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

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

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

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

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

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

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

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

6.12 Lexical syntax

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

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

6.12.1 Comments

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

The following example uses these three different kinds of comments:

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

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

6.13 Name resolution

The QL compiler resolves names to program elements.

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

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

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

In summary, the kinds of expressions are:

- **Module expressions**
  - These refer to modules.
  - They can be simple *names*, *qualified references* (in import statements), or *selections*.
- **Type expressions**
  - These refer to types.
  - They can be simple *names* or *selections*.
- **Predicate expressions**
  - These refer to predicates.
  - They can be simple *names* or names with arities (for example in an *alias* definition), or *selections*.

6.13.1 Names

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

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

```
import javascript
```

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

6.13.2 Qualified references

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

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

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

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

1. The `.`s in the qualified reference correspond to file path separators, so it first looks up `examples/security/MyLibrary.qll` from the directory containing `Example.ql`.
2. If that fails, it looks up `examples/security/MyLibrary.qll` relative to the query directory, if any. The query directory is the first enclosing directory containing a file called `qlpack.yml`. (Or, in legacy products, a file called `queries.xml`.)
3. If the compiler can't find the library file using the above two checks, it looks up `examples/security/MyLibrary.qll` relative to each library path entry. The library path is usually specified using the `libraryPathDependencies` of the `qlpack.yml` file, though it may also depend on the tools you use to run your query, and whether you have specified any extra settings. For more information, see "Library path" in the QL language specification.

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

6.13.3 Selections

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

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

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

Example

Consider the following *library module*:

CountriesLib.qll

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

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

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

```
import CountriesLib

from M::EuropeanCountries ec
select ec
```

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

```
import CountriesLib::M

from EuropeanCountries ec
select ec
```

That gives the query access to everything within `M`, but nothing within `CountriesLib` that isn't also in `M`.

6.13.4 Namespaces

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

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

Each module in QL has three namespaces:

- The **module namespace**, where the keys are module names and the entities are modules.
- The **type namespace**, where the keys are type names and the entities are types.

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

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

Global namespaces

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

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

In practice, this means that you can use the built-in types and predicates directly in a QL module (without importing any libraries). You can also use any database predicates and types directly—these depend on the underlying database that you are querying.

Local namespaces

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

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

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

This is easiest to understand in an example:

OneTwoThreeLib.qll

```
import MyFavoriteNumbers

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

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

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

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

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

Example

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

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

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

Villagers.qll

```
import tutorial

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

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

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

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

Module namespace

The module namespace of `Villagers` has entries for:

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

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

Type namespace

The type namespace of `Villagers` has entries for:

- The class `Child`.
- The types exported by the module `tutorial`.

- The built-in types, namely `int`, `float`, `string`, `date`, and `boolean`.

The type namespace of `S` has entries for:

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

Predicate namespace

The predicate namespace of `Villagers` has entries for:

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

The predicate namespace of `S` has entries for:

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

6.14 Evaluation of QL programs

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

6.14.1 Process

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

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

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

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

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

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

6.14.2 Validity of programs

The result of a query must always be a **finite** set of values, otherwise it can't be evaluated. If your QL code contains an infinite predicate or query, the QL compiler usually gives an error message, so that you can identify the error more easily.

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

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

```
from int i
select i
```

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

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

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

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

To fix these errors, it's useful to think about **range restriction**: A predicate or query is **range-restricted** if each of its variables has at least one *binding* occurrence. A variable without a binding occurrence is called **unbound**. Therefore, to perform a range restriction check, the QL compiler verifies that there are no unbound variables.

Binding

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

1. **Finite types**: Variables of a finite *type* are bound. In particular, any type that is not *primitive* is finite. To give a finite type to a variable, you can *declare* it with a finite type, use a *cast*, or use a *type check*.
2. **Predicate calls**: A valid *predicate* is usually range-restricted, so it *binds* all its arguments. Therefore, if you *call* a predicate on a variable, the variable becomes bound.

   Important

   If a predicate uses non-standard binding sets, then it does **not** always bind all its arguments. In such a case, whether the predicate call binds a specific argument depends on which other arguments are bound, and what the binding sets say about the argument in question. For more information, see "*Binding sets*."
3. **Binding operators**: Most operators, such as the *arithmetic operators*, require that all their operands are bound. For example, you can't add two variables in QL unless you have a finite set of possible values for both of them.

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

Intuitively, a binding occurrence restricts the variable to a finite set of values, while a non-binding occurrence doesn't. Here are some examples to clarify the difference between binding and non-binding occurrences of variables:

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

While the occurrence of a variable can be binding or non-binding, the variable's property of being "bound" or "unbound" is a global concept—a single binding occurrence is enough to make a variable bound.

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

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

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

6.15 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.

6.15.1 Introduction

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

6.15.2 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.

6.15.3 Architecture

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

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

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

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

6.15.4 Library path

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

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

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

A `qlpack.yml` file defines a *QL pack*. The content of a `qlpack.yml` file is described in the CodeQL CLI documentation.

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

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

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

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

6.15.5 Name resolution

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

The environments are:

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

If not otherwise specified, then the environment for a piece of syntax is the same as the environment of its enclosing syntax.

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

Environments may be combined as follows:

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

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

Global environments

The global module environment is empty.

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

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

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

Module environments

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

- The *imported X environment* of a module is defined to be the union of the exported X environments of all the modules which the current module directly imports (see "*Import directives*").
- The *declared X environment* of a module is the multimap of X declarations in the module itself.
- The *exported X environment* of a module is the union of the exported X environments of the modules which the current module directly imports (excluding `private` imports), and the declared X environment of the current module (excluding `private` declarations).
- The *external X environment* of a module is the visible X environment of the enclosing module, if there is one, and otherwise the global X environment.
- The *visible X environment* is the union of the imported X environment, the declared X environment, and the external X environment.

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

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

6.15.6 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.

6.15.7 Types

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

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.

6.15.8 Values

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

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 simpleId
```

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

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

6.15.9 The store

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

A *fact* is a predicate or type along with 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.

6.15.10 Lexical syntax

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

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

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
```

(continues on next page)

(continued from previous page)

```
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:

```
<
<=
=
>
>=
_
-
,
```

(continues on next page)

(continued from previous page)

```
;
!=
/
.
..
(
)
[
]
{
}
*
%
+
|
```

Identifiers

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

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

Here are some examples of identifiers:

```
width
Window_width
window5000_mark_II
@expr
```

There are several kinds of identifiers:

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

Identifiers are used in following syntactic constructs:

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

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...\""
```

6.15.11 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" | "noinline" | "nomagic" | "noopt") "]"
                 |   "language" "[" "monotonicAggregates" "]"
                 |   "bindingset" "[" (variable ( "," variable)*)? "]"
```

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

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

Simple annotations

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

Annotation	Classes	Characters	Member predicates	Non-member predicates	Imports	Fields	Modules	Aliases
`abstract`	yes		yes					
`cached`	yes	yes	yes	yes			yes	
`external`				yes				
`final`	yes		yes			yes		
`transient`				yes				
`library`	yes							
`private`	yes		yes	yes	yes	yes	yes	yes
`deprecate`	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				
`noinline`		yes	yes	yes				
`nomagic`		yes	yes	yes				
`noopt`		yes	yes	yes				

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

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

6.15.12 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.

6.15.13 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)* "{" member* "}"
```

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

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

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

A class adds a mapping from the class name to the class declaration to the current 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.

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. This is the union of the exported `X` environments of types the class inherits from, excluding any elements that are `overridden` by another element.

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" simpleId)?
```

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

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

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

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

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

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.

6.15.14 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
        |   callwithresult
        |   postfix_cast
        |   aggregation
        |   any
```

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 type of `this` in the typing environment.

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

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

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 call has a super expression as the receiver, then it resolves to a member predicate in a class that the enclosing class inherits from:**
  - If the super expression is unqualified and there is a single class that the current class inherits from, then the super-class is that class.
  - If the super expression is unqualified and there are multiple classes that the current class inherits from, then the super-class is the domain type.
  - Otherwise, the super-class is the class named by the qualifier of the super expression.

The predicate is resolved by looking up its name and arity in the exported predicate environment of the super-class.

- If 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`.

If the call resolves to a member predicate, then the *receiver values* are as follows. If the call has a receiver, then the receiver values are the values of that receiver. If the call does not have a receiver, then the single receiver value is the value of `this` in the contextual named tuple.

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

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

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

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.

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

Since release 2.7.1 of the CodeQL CLI, and release 1.30 of LGTM Enterprise, a trailing comma is allowed in a set literal.

6.15.15 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.

6.15.16 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
```

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

Calls

A call has the following syntax:

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

The identifier is called the *predicate name* of the call.

A call must resolve to a predicate, using the same definition of resolve as for calls with results (see "*Calls with results*").

The resolved predicate must not have a result type.

If the resolved predicate is a built-in member predicate of a primitive type, then the call may not include a closure. If the call does have a closure, then the call must resolve to a predicate with relational arity of 2.

The *candidate tuples* of a call are the ordered tuples formed by selecting a value from each of the arguments of the call.

If the call has no closure, then it matches whenever one of the candidate tuples satisfies the resolved predicate of the call, unless the call has a super expression as a receiver, in which case the candidate tuple must *directly* satisfy the resolved predicate. If the call has * or + closure, then the call matches whenever one of the candidate tuples satisfies or directly satisfies the associated closure of the resolved predicate.

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

6.15.17 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.

6.15.18 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:

Nam	Re-sult type	Argument types	Content
any			The empty tuple.
none			No tuples.
toUr		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
daysT	int	date	The result is the number of days between but not including the receiver and the argument.
getDa	int		The result is the day component of the receiver.
getHo	int		The result is the hours component of the receiver.
getMi	int		The result is the minutes component of the receiver.
getMo	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`.
getSe	int		The result is the seconds component of the receiver.
getYe	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.
toStr	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 si
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
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.
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, -
sin	float		The result is the sine of the receiver.

C

Table 1 – continued from previous page

Name	Result type	Argument types	Content
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,
gcd	int	int	The result is the greatest common divisor of the receiver and the argu
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 argume
log	float	int	The result is the logarithm of the receiver with the base of the argume
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, mod
bitShiftRight	int	int	The result is the bitwise right shift of the receiver by the argument, mo
bitShiftRightSigned	int	int	The result is the signed bitwise right shift of the receiver by the argum
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 c

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
charA	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.
index	int	string	The result is an index into the receiver where the argument occurs.
index	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`.
isLow			The receiver contains no upper-case letters.
isUpp			The receiver contains no lower-case letters.
lengt	int		The result is the number of characters in the receiver.
match		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.
prefi	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.
regex	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.
regex	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.
regex		string	The receiver matches the argument as a regex.
regex	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.
repla	string	string, string	The result is obtained by replacing all occurrences in the receiver of the first argument by the second.
split	string	string	The result is one of the strings obtained by splitting the receiver at every occurrence of the argument.
split	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`.
subst	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.
suffi	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 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()]`.
toDat	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`.
toFlo	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.
toLow	string		The result is the receiver with all letters converted to lower case.
toStr	string		The result is the receiver.
toUpp	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.

6.15.19 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 and does not have a `super` expression as a qualifier, depends on the dispatch dependencies of the root definitions of the target of the call. If the call has negative or zero polarity then the dependencies are strict. The predicate strictly depends on the strict dispatch dependencies of the root definitions.
- For each class `C` in the program, for each base class `B` of `C`, `C.extends` depends on `B.B`.
- For each class `C` in the program, for each base type `B` of `C` that is not a class type, `C.extends` depends on `B`.
- For each class `C` in the program, `C.class` depends on `C.C`.
- For each class `C` in the program, `C.C` depends on `C.extends`.
- For each class `C` in the program that declares a field of class type `B`, `C.C` depends on `B.class`.
- For each class `C` with a characteristic predicate, `C.C` depends on the characteristic predicate.
- For each abstract class `A` in the program, for each type `C` that has `A` as a base type, `A.class` depends on `C.class`.
- A predicate with a higher-order body may strictly depend or depend on each predicate reference within the body. The exact dependencies are left unspecified.

A valid stratification must have no predicate that depends on a predicate in a later layer. Additionally, it must have no predicate that strictly depends on a predicate in the same layer.

If a QL program has no valid stratification, then the program itself is not valid. If it does have a stratification, a QL implementation must choose exactly one stratification. The precise stratification chosen is left unspecified.

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 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.

6.15.20 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" simpleId)?

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

orderby ::= simpleId ("asc" | "desc")?

predicate ::= qldoc? annotations head optbody

annotations ::= annotation*
```

(continues on next page)

(continued from previous page)

```
annotation ::= simpleAnnotation | argsAnnotation

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

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

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

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

class ::= qldoc? annotations "class" classname "extends" 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 simpleId

formula ::= fparen
        |   disjunction
        |   conjunction
        |   implies
```

(continues on next page)

(continued from previous page)

```
        |   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

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
```

(continues on next page)

(continued from previous page)

```
        |   any
        |   range
        |   setliteral

eparen ::= "(" expr ")"

dontcare ::= "_"

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

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

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

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

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

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

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

aggregation ::= aggid ("[" expr "]")? "(" var_decls ("|" (formula)? ("|" as_exprs ("order
↪" "by" aggorderbys)?)?)? ")"
            |   aggid ("[" expr "]")? "(" as_exprs ("order" "by" aggorderbys)? ")"
            |   "unique" "(" var_decls "|" (formula)? ("|" as_exprs)? ")"

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

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
```

(continues on next page)

(continued from previous page)

```
classname ::= upperId

dbasetype ::= atLowerId

predicateRef ::= (moduleId "::")? literalId

predicateName ::= lowerId

varname ::= simpleId

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