The Quintessential Quandary Guide

Mike Bond

Last updated January 17, 2024

1 Introduction

Quandary is a language that combines elements of functional languages (especially Scheme/Lisp) and imperative languages (especially Java).

Mike Bond created Quandary in Summer 2019 to use in programming languages classes (specifically CSE 3341 and 6341 at Ohio State University), as part of an effort to better connect the implementation projects with the technical material.

This guide tries to cover everything related to Quandary, focused on two main activities:

- how to use Quandary as a programming language, by writing Quandary programs and running them using the Quandary reference interpreter; and
- how to implement a Quandary interpreter, by extending the Quandary skeleton interpreter and testing the modified interpreter using the grading script and grading test cases.

2 Getting Started

Unless stated otherwise, commands in this guide should be run from the root `Quandary-Public` directory.

Some commands may only work if your shell is bash. To change your shell to bash, run

```
chsh -s /bin/bash
```

from any directory.

Unless stated otherwise, run commands using your regular user privileges (i.e., don't use `sudo` except to install Linux packages).

Although the scripts attempt to support having spaces in path names (e.g., `/home/name/My Projects/Quandary-Public`), apparently it doesn't cover all cases (especially on non-Linux platforms?), so the easiest route is to ensure the full path to Quandary doesn't contain spaces or other weird characters.

Complete the steps in this section in order.

Here's an optional video showing how to complete the steps in this section:
`https://mediasite.osu.edu/Mediasite/Play/099a24aa59504d85b162421264a1f0ef1d`

2.1 Choose a platform

Linux. Linux is your best bet. Everything should just work.

macOS. macOS is a solid second choice. The Quandary scripts should work okay. If you get errors about the `realpath` command when running the grading script, update to the latest version of `Quandary-Public`.

Windows. If you have the misfortune of using Windows, you can still make it work. You can either

- run Linux in a system virtual machine such as VirtualBox;
- view and edit the code on Windows, but build and run the interpreter on `stdlinux`; or
- run Windows Subsystem for Linux (WSL).

If you use WSL, here's an issue you may encounter at some point:

You may need to run the following commands on the `quandary` scripts to fix errors regarding trailing `\r` characters:

```
sed -i 's/\r$//' ref/quandary
sed -i 's/\r$//' skeleton/quandary
```

Some people have found that they need to run the commands on other files such as the test case (`.dat`) files and the Quandary program (`.q` and `.calc`) files.

2.2 Get Quandary

Clone `Quandary-Public` by running this command from the directory that you want to contain the `Quandary-Public` directory:

```
git clone https://github.com/mdbond/Quandary-Public.git
```

`Quandary-Public` contains the following directories and files:

- `ref/` contains the Quandary reference interpreter
- `skeleton/` contains the Quandary skeleton interpreter
- `examples/` contains Quandary programs
- `grading/` contains the grading script and test cases
- `quandary.pdf` is this document

Viewing and editing the Quandary code. You should view and edit the Quandary code—particularly the skeleton interpreter—in an IDE. Not sure which IDE to use? Use Visual Studio Code (VS Code).

For VS Code to understand the skeleton interpreter (e.g., to support navigating and detecting errors), you'll want to open the `skeleton` directory as the root folder in VS Code. After you do that, you'll see some build errors related to VS Code not being able to find the JFlex and CUP JARs; you can ignore those build errors, or you can tell VS Code where to find the JFlex and CUP JARs.

Note that you'll still need to *build* the skeleton interpreter using the `Makefile` (see below). Or you can probably configure VS Code to run the `Makefile` every time you change a source file.

VS Code can perform perform syntax highlighting of Quandary programs. Just install the "Quandary language" extension (Id: `cameroncromer.quandary-lang`), created by Cameron Cromer in December 2023, from VS Code's "Extensions" tab. This extension supports syntax highlighting, including for Quandary's extended types (Appendix D), as well as bracket matching and comment toggling.[1]

Updating `Quandary-Public`. Whenever I update `Quandary-Public` (e.g., to add private test cases after a project deadline), you can run

```
git pull
```

in `Quandary-Public` to get the updates.

[1]Thanks also to Vahid Ahmadi Kalkhorani and Luke Balizet, who in years past created extensions that are useful but less fully featured than Cameron's extension.

2.3 Install Java

To run the reference interpreter, you'll need a Java virtual machine (JVM), i.e., the `java` command. To build the skeleton interpreter, you'll need the Java compiler, i.e., the `javac` command. To install both tools, install the Java Development Kit (JDK).

You may already have the JDK. You can skip this part if the `javac` command already works.

Not sure which Java implementation to install? Install OpenJDK.

If you're on Ubuntu, you can install the OpenJDK JDK using the following command:

```
sudo apt install default-jdk
```

If you're on `stdlinux`, run the following (just once in your life):

```
subscribe JDK-CURRENT
```

Then log out and back in.

I believe you'll need a JDK version of at least 11.

2.4 Get the reference interpreter working

Run the reference interpreter, which should print usage information:

```
~/Quandary-Public$ ref/quandary
Expected format: quandary [OPTIONS] QUANDARY_PROGRAM_FILE INTEGER_ARGUMENT
Options:
  -gc (MarkSweep|MarkSweepVerbose|RefCount|Explicit|NoGC)
  -heapsize BYTES
  -ct TIMEOUT_IN_SECONDS
BYTES must be a multiple of the word size (8)
Quandary process returned 0
```

Run the reference interpreter with a Quandary program and argument. For example:

```
~/Quandary-Public$ ref/quandary examples/primes2.q 20
Interpreter returned (2 . (3 . (5 . (7 . (11 . (13 . (17 . (19 . nil))))))))
Quandary process returned 0
```

2.5 Get the skeleton interpreter working

Dependencies. Before building the Quandary skeleton interpreter, you need to download and extract CUP and JFlex and set environment variables that the skeleton's `Makefile` is expecting.

Download JFlex and CUP using the following URLs:

- JFlex version 1.7.0: `https://jflex.de/release/jflex-1.7.0.tar.gz`
- CUP version 0.11b-20160615: `http://www2.cs.tum.edu/projects/cup/releases/java-cup-bin-11b-20160615.tar.gz`

Then extract them and set `JFLEX_DIR` and `CUP_DIR` to point to their locations. For example:

```
wget https://jflex.de/release/jflex-1.7.0.tar.gz
wget http://www2.cs.tum.edu/projects/cup/releases/java-cup-bin-11b-20160615.tar.gz
tar -zxf jflex-1.7.0.tar.gz --directory $HOME
mkdir -p $HOME/cup && tar -zxf java-cup-bin-11b-20160615.tar.gz --directory $HOME/cup
```

To be clear, these commands put JFlex's files in `$HOME/jflex-1.7.0` and put CUP's two JAR files in `$HOME/cup`.

You can of course put JFlex and CUP in other places if you like. In any case, be sure that two JAR files end up in `$CUP_DIR`.

Set the environment variables `JFLEX_DIR` and `CUP_DIR` to the locations of JFlex and CUP, respectively. Use absolute, not relative, paths for `JFLEX_DIR` and `CUP_DIR`. For example, if JFlex and CUP are in `$HOME/jflex-1.7.0` and `$HOME/cup`, respectively, and your shell is bash:

```
export JFLEX_DIR=$HOME/jflex-1.7.0
export CUP_DIR=$HOME/cup
```

You should add these commands to your `$HOME/.bashrc`, so they'll be set automatically every time you open a terminal.

Another dependency that may come up for you: You'll need to install `make` if it's not already installed.

Build the skeleton interpreter. Run `make` in the skeleton directory:

```
(cd skeleton && make)
```

which will run `make` in the `skeleton` directory and later return to the parent directory (`Quandary-Public`).

If you get this error,

```
~/Quandary-Public$ (cd skeleton && make)
cd parser && /bin/jflex --nobak Scanner.jflex
/bin/sh: 1: /bin/jflex: not found
make: *** [Makefile:8: parser/Lexer.java] Error 127
```

then `JFLEX_DIR` (and perhaps `CUP_DIR` too) aren't set.

If you get an error related to the `Location` class, I'm not sure why you're seeing this error (it's rarely seen). Anyway, I think it means there's a conflict between `ast.Location` and `java_cup.runtime.ComplexSymbolFactory.Location`. Try changing `Location` to `java_cup.runtime.ComplexSymbolFactory.Location` everywhere in `Scanner.jflex`.

Run the skeleton interpreter. After successfully building the skeleton interpreter, you can run it:

```
~/Quandary-Public$ skeleton/quandary
Expected format: quandary [OPTIONS] QUANDARY_PROGRAM_FILE INTEGER_ARGUMENT
Options:
  -gc (MarkSweep|Explicit|NoGC)
  -heapsize BYTES
BYTES must be a multiple of the word size (8)
Quandary process returned 0
```

Next, run the skeleton interpreter with an input program. As described in Section 3, the skeleton only recognizes "programs" that are simple arithmetic expressions. There are a few such example "programs" provided in `Quandary-Public`: `examples/*.arith`. For example:

```
~/Quandary-Public$ skeleton/quandary examples/simple.arith 42
Interpreter returned 106
Quandary process returned 0
```

Note that the argument to the program (42) is unused because the simple "program" doesn't have a `main` function. However, the argument is still required.

2.6 What to do next

Test more example Quandary programs with the reference interpreter. Write your own Quandary programs and run them with the reference interpreter. Modify the Quandary skeleton in order to implement the projects described in Section 7. Run the grading script (Section 6.1) to evaluate your modified skeleton. Read the rest of this document.

3 Understanding the Skeleton Interpreter

It's hard to describe code in detail without actually pointing to specific parts of the code, so here's a video describing how the skeleton interpreter works at the code level: `https://mediasite.osu.edu/Mediasite/Play/87eb03511eed4791b6d1701427fc321b1d`

Note: The Quandary skeleton interpreter (`Quandary-Public/skeleton`) does not recognize regular Quandary programs. Instead, it only recognizes "programs" that are simple arithmetic expressions (`examples/*.arith`).[2] (See Section 2.5 for info about building and running the skeleton.)

4 Academic Integrity

You'll implement the projects by modifying the skeleton interpreter. You'll want to save your code somewhere like in a GitHub repository. However, you must store your code in a *private* repository. Storing your code in a public repository, or making your code public in any other way, during or after the semester, is a violation of academic integrity. And of course don't share or show your interpreter source code to anyone either. And don't use or look at anyone else's interpreter source code.

Restrictions. For the most part, you can implement the projects however you like.[3] If it works, it works.

For the memory management project, your interpreter must allocate heap objects into "raw memory." Use the provided `RawMemory` class, without modification, to emulate raw memory. `RawMemory` provides only low-level load, store, and compare-and-set operations on an emulated address space.

5 Advice for Modifying the Skeleton Interpreter and Implementing the Projects

To implement the projects you'll want to

- modify the lexer specification (`Scanner.jflex`),
- modify the parser specification (`Parser.cup`),
- modify and add AST files (`ast/*.java`), and
- modify the interpreter's execution `interpreter/Interpreter.java`.

You don't want to modify any files that are generated automatically by JFlex or CUP. To see only files that aren't generated automatically, run `(cd skeleton && make clean)` to eliminate generated files.

5.1 Lexing and parsing

When modifying `Parser.cup`, you'll want to make small changes and then test them. If you get build errors from CUP, or from `javac` when it tries to build the generated parser, they can be quite unhelpful,

[2] The Quandary *reference* interpreter does *not* recognize "programs" that are simple arithmetic expressions (`examples/*.arith`). It *does* recognize calculator programs (`examples/*.calc`; see Section 7.1) and Quandary programs (`examples/*.q`).

[3] Within reason. Implementing an "interpreter" that just runs the reference interpreter is not allowed.

so it's often best to just revert the last small change you made. Hint: The issue is often a syntax error in `Parser.cup`.

If CUP reports a shift/reduce conflict or reduce/reduce conflict, then precedence has not been unambiguously specified in `Parser.cup`. See Section 8.2 for Quandary precedence rules.

If you get an error about "Syntax error for symbol ... instead expected token classes are ..." then the parser's grammar and the input program don't match. Use the reported line and column numbers to find out where the unexpected token is in the input program, and try to understand why the grammar specified in `Parser.cup` doesn't recognize it.

To help debug errors related to parsing and lexing, you can turn on debugging of the lexer and parser by changing `parse()` to `debug_parse()` in `ParserWrapper.java`.

To give unary minus different precedence than binary minus, CUP allows `Parser.cup` to specify a new terminal that is an alias for another terminal. Search for `UMINUS` in the CUP documentation (`https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html`) for an example.

To give the type cast "operator" the correct precedence, you can use a similar trick by creating a terminal that represents the type cast "operator."

You can use CUP's precedence command to solve the dangling `else` problem, by essentially treating `else` as an operator. I can't explain exactly why the following works, so instead I'll just share what to do. Add this line in `Parser.cup` (ordering with respect to other `precedence` commands shouldn't matter):

```
precedence left ELSE; // resolves dangling else
```

`Scanner.jflex` is order sensitive: Rules earlier in the file are matched first. Thus make sure to put the rules for recognizing keywords *before* the rules for recognizing identifiers; otherwise keywords will be recognized incorrectly as identifiers.

5.2 Adding functionality to your interpreter

It helps to break down the development along two axes:

1. Implement one feature at a time. For example, first conditions, then if-statements, next statement lists, and so on.
2. For each implemented feature, design the AST nodes, then modify `Parser.cup` to construct the AST, next check that a correct-looking AST is getting built (e.g., print it from `Interpreter.main()`), and finally implement the evaluation and execution of the Quandary program by modifying `Interpreter.evaluate()`, `Interpreter.execute()`, and/or similiar methods you might create.

To debug your changes, your interpreter can print the AST after building and before executing it. Uncomment the `println()` in `Interpreter.main()`, and write a `println()` function for all statement-like nodes and implement `toString()` for all expression-like nodes.

Designing the AST. You don't need (or even want) to have a perfect one-to-one relationship between the grammar and your AST nodes. For example, it's up to you whether you want to have a single `IfStmt` class (which represents both if and if–else statements) or separate `IfStmt` and `IfElseStmt` classes. In some cases, you probably want a separate AST node for each non-terminal (e.g., `FuncDef` and `FuncDefList`). In other cases, you probably want a separate AST node for each *production alternative* of a non-terminal (e.g., `IfStmt`, `ReturnStmt`, etc.).

***Building* and *executing* the AST should be kept separate.** To build the AST, the parser invokes actions specified in `Parser.cup`, which create AST nodes. After that, your interpreter shouldn't change the AST (or anything it points to)—the AST represents the static program.

Execution of the program deals with dynamic (run-time) values and control flow. To execute the program, the interpreter calls `Program.executeRoot()`, which recursively calls methods like `Interpreter.execute()`

and `Interpreter.evaluate()` (depending on exactly how you implement things). These methods shouldn't modify any member variables of the AST nodes (or anything they point to).

To keep track of Quandary variables' values, your interpreter can track the currently executing Quandary function's local variables' values by maintaining an "environment," which is a map from local variable names to their current values. (This environment is a map—such as a Java `HashMap`—and it's distinct from the Linux environment variables mentioned elsewhere in this document.) You should probably modify `Interpreter.evaluate()` and `Interpreter.execute()` to take a parameter representing the current environment.

Implementing Quandary `return` statements. There are (at least) two ways to implement `return` statements, which need to exit the current Quandary function immediately with a return value:

- Make `Interpreter.execute()` (or whatever method you use for executing non-expression AST nodes) return a value. The value `null` represents that a `return` statement hasn't executed yet, and a non-`null` v represents that a `return` statement has executed and returned the value v.
- Use Java exception handling. Make a custom exception type that extends `RuntimeException`, not `Exception`, to avoid having to put `catch` blocks all over the place.

5.3 Debugging and testing

Do initial testing manually (i.e., without using `grade.sh`) by running the reference interpreter and your interpreter and comparing the results. Once your interpreter works on most or all of the test cases you've tried manually, then it makes sense to start running `grade.sh` (Section 6.1).

If you get an error like this:

```
Syntax error for input symbol "(" spanning from unknown:1/9(1) to unknown:1/9(9)
instead expected token classes are [MINUS, RETURN, IF, IDENT]
Couldn't repair and continue parse (Symbol: ( (unknown:1/9(1) - unknown:1/9(9)))
Quandary process returned 1
```

it means that there's a symbol "(" at line 1, character 9 of the input Quandary program that the parser does not expect. The grammar specified in `Parser.cup` doesn't recognize the input Quandary program.

If you get an error like this:

```
Error: Syntax error @ Symbol: ID (unknown:54/12(-1) - unknown:54/23(-1))
Error : Internal error: Unexpected exception
Exception in thread "main" java.lang.NullPointerException
  at java_cup.runtime.lr_parser.symbl_name_from_id(lr_parser.java:456)
  ...
make: *** [Makefile:9: classes] Error 1
```

it means there's a parsing error in your `Parser.cup`, at (or related to) line 54 in this case.

Generally speaking, if you get a run-time exception such as a `NullPointerException`, e.g.,

```
Exception in thread "main" java.lang.NullPointerException
  at interpreter.Interpreter.evaluate(Interpreter.java:234)
  at ...
Quandary process returned 1
```

then the interpreter tried to dereference a `null` reference at line 234 of `Interpreter.evaluate()`. Step one is to go look at that line of code, figure out which reference was `null` and why it was `null` and/or why the code assumes it should be non-`null`. Similarly for other types of errors; for example, for a `ClassCastException`, go to the line of code that threw the exception and understand why the run-time type and the downcast expression don't match.

If your interpreter takes a long time executing a recursive test case like `oddeven.q 50`, it could be because your interpreter is evaluating some expressions twice instead of once.

5.4 Asking for help

If you have questions or encounter issues, first check this document and Piazza for possible solutions. Then the best ways to get help are:

- Make a public post on Piazza (see below for tips)
- Attend instructor or TA office hours
- Ask in class

For Piazza posts, post as much information as possible. When showing output from running the reference interpreter or your interpreter, be sure to show both the command you ran as well the full output. If you're seeing an issue related to a Quandary program you wrote or modified, include the full Quandary program.

If you encounter problems running a script, run it prefaced with `bash -x` to see debugging information. For example:

```
bash -x skeleton/quandary examples/primes2.q 20
```

Another example:

```
bash -x grading/grade.sh skeleton/myproject.tgz ref/quandary grading/calc-public.dat examples
```

and then post the full output of the command along with other information including your platform.

6 Grading and Submitting Your Interpreter

The `Makefile` automatically generates a "submission" `myproject.tgz`, which you can test using the grading script and eventually submit on Carmen.

Unless you changed something in the skeleton that will require a different `Makefile` or `quandary` script *in the TA's environment*, you should submit using the original `Makefile` and `quandary` script.

6.1 The grading script and test cases

The grading script, `grading/grade.sh`, is the same script that the TA will use to grade your submission.

Grading script. Run the grading script with the following command:

```
grading/grade.sh SUBMISSION_TGZ REF_IMPL TESTCASE_LIST TESTCASE_DIR
```

where

- `SUBMISSION_TGZ` is the `.tgz` being submitted
- `REF_IMPL` is the Quandary reference interpreter script
- `TESTCASE_LIST` is a file that specifies a list of test cases; each test case is on its own line and has the following format:

  ```
  POINTS PROGRAM INPUT [QUANDARY_ARGUMENTS]
  ```

  where

  - `POINTS` is the number of points the test case is worth

  - `PROGRAM` is the file containing the Quandary program (must be located in `TESTCASE_DIR`)
  - `INPUT` is the integer input to the program
  - `QUANDARY_ARGUMENTS` are optional arguments to the interpreter
- `TESTCASE_DIR` is the location of the program files listed in `TESTCASE_LIST`

Here's an example:

```
grading/grade.sh skeleton/myproject.tgz ref/quandary grading/calc-public.dat examples
```

`JFLEX_DIR` and `CUP_DIR` must be set correctly when running `grade.sh`.

After your submission has been graded. After the deadline, we'll make the private (grading) test cases available to you. You should be able to reproduce the score that the TA computes (modulo nondeterminism, if applicable) by running the interpreter you submitted with `grade.sh` and the private test cases.

If you can't figure out how you got the score you did, make a (ideally public) Piazza post showing the full output of running `grade.sh` with the private test cases giving a score different than what your submission received in Carmen.

Sanity-checking the grading script. The grading script runs the reference interpreter and your interpreter (`myproject.tgz`) and compares the output. Note that if they both fail with an error, that's considered success. So if something is wrong with your setup, all test cases will appear to have PASSED.

To help with understanding whether the grading script is giving trustworthy results, every test case file (`.dat`) contains at least one test case for `isrefint.q`, e.g.,

```
0 isrefint.q 42
```

When the grading script runs this test case, it should report FAILED. If it reports PASSED, then something is wrong with how the grading script is being run—probably both the reference and skeleton interpreters are failing, perhaps because none of the test case files can be found. To debug this issue, uncomment these lines in `grade.sh` to see the output of the reference interpreter when run by the grading script:

```
# Uncomment to see reference interpreter output only:
echo ""
$REF_IMPL $OPTIONS $TESTCASE_DIR/$PROGRAM $INPUT
echo ""
return
```

Test cases. I've provided representative public test cases for each project in `grading/*-public.dat` (see Section 7). The point values are arbitrary/meaningless. *These test cases are useful for understanding concretely what features are needed for each project.*

6.2 Submitting your project

Upload your `myproject.tgz` (generated by running `make`) to Carmen. You don't need to upload anything else. You can upload as many times as you like—only the latest submission and its timestamp will count.

7 Interpreter Projects

Except when specified in the project description (like for the checking project!), your interpreter may assume that input programs are statically and dynamically correct, i.e., your interpreter does not need to do any static or dynamic checking unless it's required by the project description.

For each project, you may assume that your interpreter will not be tested using Quandary programs that use features that are not required for the project. That is, executions of such programs have undefined behavior for theproject, and your interpreter may do anything for these programs.

The rest of this section describes each project and provides the name of the test cases (`.dat`) file for the project. This file lists example programs that are representative input for the project. *To understand concretely what you need to implement for each project, look through the Quandary program files listed in the relevant `.dat` file.* A few caveats:

- Be aware that some of the public test cases are intended to fail when run with the interpreter: They may generate one of the errors listed in Section 8.3. You want to be aware of such erroneous test cases, especially when you're looking at the test cases to understand concretely what kinds of Quandary programs should be handled by your interpreter.
- While the public test cases are intended to be representative of the private test cases that will be used for grading, the *distribution of behaviors* may not be representative. For example, `calc-public.dat` includes several test cases with parsing errors. The private test cases for the Calculator project will probably include only about one test case with a parsing error.
- While the public test cases are intended to be representative of the private test cases that will be used for grading, the private test cases may have some behaviors that weren't tested by the public test cases.

7.1 Calculator

Programs consist of a `return` statement containing constants, plus, binary minus, times, unary minus, and parentheses.

See `calc-public.dat` for a list of example programs.

Watch this video for more info about how to get started on the Calculator project: `https://mediasite.osu.edu/Mediasite/Play/1cf5ddcbe52948b68269518972efb66d1d`. Note: In the video I forgot to mention it, but you'll want to modify `Interpreter.java` (in addition to `Scanner.jflex`, `Parser.cup`, and files in the `ast` package) for this project and other projects. See also Section 5.

7.2 Simplified Quandary Without Calls

A programs consists of a single `main` function; there are no calls and no other function definitions. A program uses any of the non-heap, non-mutation, non-concurrency functionality (i.e., the functionality in the default color in Section 8.1) *except* function definition lists, formal declaration lists, expression lists, and call expressions.

See `simplified-without-calls-public.dat` for a list of example programs.

7.3 Simplified Quandary With Calls

A program uses any of the non-heap, non-mutation, non-concurrency functionality (i.e., the functionality in the default color in Section 8.1). A program may call the built-in `randomInt()` function (Section 8.4).

See `simplified-with-calls-public.dat` for a list of example programs.

7.4 Basic Quandary Without Checking

A program uses any of the functionality for Simplified Quandary With Calls, plus functionality for heap and mutation, *excluding* `free` statements. A program may call any built-in function *except* `acq()` and `rel()`.

See `basic-without-checking-public.dat` for a list of example programs.

7.5 Basic Quandary With Checking

A program uses the same functionality as Basic Quandary Without Checking. In addition, the interpreter performs static and dynamic checking (Sections 8.5 and 8.3).

See `basic-with-checking-public.dat` for a list of example programs.

7.6 Extended Quandary

This project extends Basic Quandary With Checking with two features:

- Support for memoization (see Appendix C)
- Support for more types (see Appendix D)

See `extended-public.dat` for a list of example programs.

7.7 Memory Management

A program uses the same functionality as Basic Quandary Without Checking, plus `free` statements. The interpreter represents the heap using raw memory (use `RawMemory.java` without modification), and it fails with an out-of-memory error if the heap is exhausted. The interpreter supports mark–sweep GC (`-gc MarkSweep`), explicit memory management (`-gc Explicit`), and no memory management (`-gc NoGC`).

See `memory-management.dat` for a list of example programs.

Temporary roots: To work correctly, your mark–sweep implementation should consider not only functions' local variables, but also temporary values that aren't referenced by any local variable when allocation might trigger a GC, to be GC roots. This applies to two kinds of expressions:

- `expr1 . expr2` — After evaluating `expr1`, the interpreter should add the resulting value as a temporary root while evaluating `expr2`. While allocating the new object for `expr1 . expr2`, the interpreter should keep the temporary root for `expr1`'s result and add a temporary root for `expr2`'s result. After allocating the new object for `expr1 . expr2`, the interpreter should remove both temporary roots.
- `f(arg1, arg2, ..., argn)` — After evaluating an argument expression, the interpreter should add the resulting value as a temporary root while evaluating the rest of the argument expressions. After evaluating the last argument expression `argn` (i.e., just before making the call), the interpreter should remove all $n-1$ temporary roots for the argument expressions.

7.8 Concurrency

A program uses the same functionality as Basic Quandary Without Checking, plus concurrent binary expressions. A program may call any built-in function.

See `concurrency-public.dat` for a list of example programs.

Important:

- If you're using a system virtual machine such as VirtualBox, be sure to set the number of cores to at least 4. If it is set lower—particularly if it is set to 1—then concurrency in the reference interpreter (and likely in your interpreter) won't behave correctly: Certain tests will be very slow because of the interpreters' simple spin locks implementation, and certain Quandary programs will behave unexpectedly because threads' operations will interleave infrequently.
- Your interpreter must pass test cases that actually implement concurrent evaluation of expressions (as opposed to sequentially evaluating concurrent binary expressions). Specifically, your interpreter's earned fraction of max points will max out at the fraction of "relies on concurrent execution" (see `concurrency-public.dat`) test cases that it passes.

Other tips:

- To test your `acq()`/`rel()` implementation, use simple test cases (simpler than `steps*.q` and `pcfixed*.q`, for example). Two recommended simple test cases:

```
examples/counter2.q 10000
examples/drfcounter4.q 10000
```

- If your interpreter crashes/fails nondeterministically on some `pcfixed*.q` tests cases, it's likely because your interpreter doesn't handle empty statement lists correctly.

8 Quandary Language and Runtime Specification

8.1 Syntax and semantics

This part tries to show the syntax (structure) and some of the semantics (behavior) of Quandary, by showing a context-free grammar for Quandary along with some comments about the meaning and behavior of various syntactic elements.

Colors denote productions used only for heap (including the `Q` and `Ref` types), concurrency, and mutation. The list of built-in functions (Section 8.4) uses the same color coding.

⟨*program*⟩ ::= ⟨*funcDefList*⟩ // A program is a list of function definitions

⟨*funcDefList*⟩ ::= ⟨*funcDef*⟩ ⟨*funcDefList*⟩ // A function definition list is zero or more function definitions
| ϵ

⟨*funcDef*⟩ ::= ⟨*varDecl*⟩ `(` ⟨*formalDeclList*⟩ `)` `{` ⟨*stmtList*⟩ `}` // A function definition: decl, params, body

⟨*varDecl*⟩ ::= ⟨*type*⟩ `IDENT` // Declares immutable variable or function
| `mutable` ⟨*type*⟩ `IDENT` // `mutable` vars can be updated; `mutable` funcs can update objects

⟨*type*⟩ ::= `int` // Static type for 64-bit signed integers
| `Ref` // Static type for references to heap objects and `nil`
| `Q` // Supertype of `int` and `Ref`

⟨*formalDeclList*⟩ ::= ⟨*neFormalDeclList*⟩ // Comma-delimited list of function parameters
| ϵ

⟨*neFormalDeclList*⟩ ::= ⟨*varDecl*⟩ `,` ⟨*neFormalDeclList*⟩ // Helps represent comma-delimited list
| ⟨*varDecl*⟩

⟨*stmtList*⟩ ::= ⟨*stmt*⟩ ⟨*stmtList*⟩ // A statement list is zero or more statements
| ϵ

⟨*stmt*⟩ ::= ⟨*varDecl*⟩ `=` ⟨*expr*⟩ `;` // Declaration statement: declares and initializes variable
| `IDENT` `=` ⟨*expr*⟩ `;` // Assignment statement: updates previously declared (`mutable`) variable
| `if` `(` ⟨*cond*⟩ `)` ⟨*stmt*⟩ // Java-style `if` statement without `else`
| `if` `(` ⟨*cond*⟩ `)` ⟨*stmt*⟩ `else` ⟨*stmt*⟩ // Java-style `if` statement with `else`
| `while` `(` ⟨*cond*⟩ `)` ⟨*stmt*⟩ // Java-style `while` loop
| `IDENT` `(` ⟨*exprList*⟩ `)` `;` // Call statement: makes a call and ignores return value
| `free` ⟨*expr*⟩ `;` // Frees memory if explicit memory management enabled
| `print` ⟨*expr*⟩ `;` // Prints evaluated value followed by a newline
| `return` ⟨*expr*⟩ `;` // Java-style `return` statement
| `{` ⟨*stmtList*⟩ `}` // Block of statements

⟨*exprList*⟩ ::= ⟨*neExprList*⟩ // Comma-delimited list of function arguments

$|\ \epsilon$

$\langle neExprList \rangle$ ::= $\langle expr \rangle$ `,` $\langle neExprList \rangle$ // Helps represent comma-delimited list
  | $\langle expr \rangle$

$\langle expr \rangle$ ::= `nil` // Expression evaluating to constant value `nil` with type `Ref`
  | `INTCONST` // Expression evaluating to constant value of type `int`
  | `IDENT` // Expression evaluating to current value of variable
  | `-` $\langle expr \rangle$ // Unary minus expression
  | `(` $\langle type \rangle$ `)` $\langle expr \rangle$ // Typecast expression
  | `IDENT (` $\langle exprList \rangle$ `)` // Call expression
  | $\langle binaryExpr \rangle$ // Binary expression, i.e., expression involving binary operator
  | `[` $\langle binaryExpr \rangle$ `]` // Binary expr for which left and right expressions are evaluated concurrently
  | `(` $\langle expr \rangle$ `)` // Parenthetical expression (for overriding precedence/associativity)

$\langle binaryExpr \rangle$ ::= $\langle expr \rangle$ `+` $\langle expr \rangle$ // Binary plus expression
  | $\langle expr \rangle$ `-` $\langle expr \rangle$ // Binary minus expression
  | $\langle expr \rangle$ `*` $\langle expr \rangle$ // Binary times expression
  | $\langle expr \rangle$ `.` $\langle expr \rangle$ // Expression evaluating to reference to new object

$\langle cond \rangle$ ::= $\langle expr \rangle$ `<=` $\langle expr \rangle$ // Compares two `int` values
  | $\langle expr \rangle$ `>=` $\langle expr \rangle$ // Compares two `int` values
  | $\langle expr \rangle$ `==` $\langle expr \rangle$ // Compares two `int` values
  | $\langle expr \rangle$ `!=` $\langle expr \rangle$ // Compares two `int` values
  | $\langle expr \rangle$ `<` $\langle expr \rangle$ // Compares two `int` values
  | $\langle expr \rangle$ `>` $\langle expr \rangle$ // Compares two `int` values
  | $\langle cond \rangle$ `&&` $\langle cond \rangle$ // Java-style logical AND conditional
  | $\langle cond \rangle$ `||` $\langle cond \rangle$ // Java-style logical OR conditional
  | `!` $\langle cond \rangle$ // Java-style logical NOT conditional
  | `(` $\langle cond \rangle$ `)` // Parenthetical conditional (for overriding precedence/associativity)

Lexical analysis. An `IDENT` is a sequence of letters, digits, and underscores such that the first character is not a digit.

If an `INTCONST` exceeds the bounds of a 64-bit signed integer, the interpreter's behavior is undefined.

Quandary's syntax is case sensitive.

Quandary allows Java/C/C++/Rust-style "block" comments `/* like this */`

8.2 Precedence and dangling `else`

Precedence of operators from high to low:

1. `-` used as a unary operator and `(` $\langle type \rangle$ `)` (cast operator)
2. `*`
3. `-` used as a binary operator and `+`
4. `.`
5. `<=`, `>=`, `==`, `!=`, `<`, and `>`
6. `!`
7. `&&`

8. `||`

All operators are left assocative.

Dangling `else` ambiguity is resolved by matching an `else` with the nearest `if` statement allowed by the grammar, i.e., the same as Java/C/C++/Rust. See Section 5.1 for how to resolve dangling `else` ambiguity.

8.3 Language semantics and interpreter behavior

The interpreter executes the defined function called `main` and passes a command-line parameter as `main`'s argument. Incorrect command-line parameters, such as the program file not being found, have undefined behavior.

The interpreter prints the return value of `main` using the following algorithm:

- a `int` value is printed in decimal, e.g., `42`
- a `nil` value is printed as `nil`
- a non-`nil` `Ref` value v is printed as `(`l `.` r`)` where l is `left(`v`)` of the value and r is `right(`v`)` of the value.

For example:

```
Interpreter returned ((5 . nil) . (-87 . (9 . 3)))
```

Function calls. Function call semantics are pass-by-value.

Order of evaluation. The interpreter evaluates expressions in left-to-right order, i.e., it evaluates the left side of (non-concurrent) binary expressions before the right side, and it evaluates function call actual expressions in left-to-right order.

Binary boolean operators (`&&` and `||`) use short-circuit evaluation.

Dynamic type checking. The interpreter should check evaluated type downcasts at run time and report a fatal dynamic type checking error on a type downcast failure.

Heap mutation. A new heap object's left and right fields are each initialized to an `int` or `Ref` value, and must remain as either an `int` or `Ref` value, respectively, for the duration of the execution. Thus the interpreter should fail with a dynamic type checking error if the `setLeft()` or `setRight()` function attempts to overwrite an `int` slot with a `Ref` value, or a `Ref` slot with an `int` value.

The purpose of this restriction is to avoid the implementation challenge of updating both the value and associated type metadata atomically (which is an issue if implementing objects using "raw" memory).

`nil` dereference. Calling `left()`, `right()`, `setLeft()`, `setRight()`, `acq()`, or `rel()` with a first argument evaluating to `nil` causes a fatal `nil` dereference error at run time.

Memory management. An execution should generate an "out of memory" error if and only if the non-freed memory exceeds the specified maximum heap size.

The interpreter potentially supports explicit memory management, tracing-based mark–sweep garbage collection, and reference-counting-based garbage collection.

Tracing-based garbage collection only: An evaluation of an allocation expression ($\langle binaryExpr \rangle ::= \langle expr \rangle$ `.` $\langle expr \rangle$) performs tracing-based GC when and only when the non-freed memory exceeds the specified maximum heap size. Tracing-based GC frees objects that are transitively unreachable from the roots (functions' local variables and intermediate values). Implementing support for stopping multiple threads at GC-safe points is not required because it is tricky to implement; if tracing-based GC is triggered when multiple threads are active, the interpreter has undefined behavior (the reference interpreter reports an error in this case).

Explicit memory management only: An execution that accesses a freed object has undefined semantics. An execution that performs double-free on a reference has undefined semantics. An execution that tries to free `nil` has undefined semantics.

Concurrency. A concurrently evaluated binary expression `[` ⟨*binaryExpr*⟩ `]` evaluates the left and right child expressions in two new concurrent threads (i.e., thread fork), and waits for both threads to finish (i.e., thread join). Every thread that is not blocked eventually makes progress.

Thread fork and join and lock acquire and release are synchronization operations that induce happens-before edges.

Conflicting accesses unordered by happens-before constitute a data race. An execution of a program with a data race has undefined semantics.

An execution in which a thread performs a `rel()` of a lock it does not hold, has undefined semantics.

Error checking. To help with grading, the interpreter *process* returns one of the following error codes:

- 0 – success
- 1 – lexical analysis or parsing error
- 2 – static checking error
- 3 – dynamic type checking error
- 4 – `nil` dereference error
- 5 – Quandary heap out-of-memory error

The interpreter script (`quandary`) prints this return code. Specifically, the output should be as follows.

1. For a non-erroneous, terminated program, the last two lines of output should be:

   ```
   Interpreter returned RETURN_VALUE_OF_MAIN
   Quandary process returned 0
   ```

   where `RETURN_VALUE_OF_MAIN` is return value of the Quandary program's `main` function.

   Printing anything or nothing before that is fine.

2. For a non-erroneous, non-terminating execution (e.g., a program execution with an infinite loop),[4] the interpreter should not terminate. Printing anything or nothing is fine.

3. For an execution that should return an error code `ERROR_CODE`, the last line of output should be:

   ```
   Quandary process returned ERROR_CODE
   ```

   Printing anything or nothing before that is fine.

4. For an execution that has undefined behavior, any behavior and output is allowed (including termination with an uncaught Java exception). That is, an interpreter can safely assume that programs and inputs with undefined behavior will *not* be executed.

If the *interpreter program itself* runs out of stack memory, runs out of heap memory, or allocates too many threads, then behavior is undefined (any behavior is acceptable). For a reasonable Quandary input program, the interpreter should succeed if given enough stack memory, heap memory, and thread count limit.

[4]Execution with unbounded call depth has undefined behavior.

8.4 Built-in functions

The Quandary interpreter provides the following built-in functions:

`int randomInt(int n)` – Returns a random `int` in $[0, n)$

`Q left(Ref r)` – Returns the left field of the object referenced by `r`

`Q right(Ref r)` – Returns the right field of the object referenced by `r`

`int isAtom(Q x)` – Returns `1` if `x`'s value is `nil` or an `int`; returns `0` otherwise (if `x`'s value is a non-`nil` `Ref`)

`int isNil(Q x)` – Returns `1` if `x`'s value is `nil`; returns `0` otherwise (if `x`'s value is an `int` or a non-`nil` `Ref`)

`mutable int setLeft(Ref r, Q value)` – Sets the left field of the object referenced by `r` to `value`, and returns `1`

`mutable int setRight(Ref r, Q value)` – Sets the right field of the object referenced by `r` to `value`, and returns `1`

`mutable int acq(Ref r)` – Acquires the lock of the object referenced by `r` and returns `1`

`mutable int rel(Ref r)` – Releases the lock of the object referenced by `r` and returns `1`

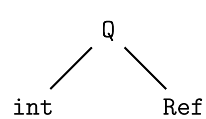
8.5 Static type-checking rules

The Quandary interpreter checks the following rules prior to executing the program.

Declarations. A program must not define a function with the same name as another function, including the built-in functions. A program must only call functions defined in the program or built-in functions. A program must define a function named `main` that takes a single argument of type `int`.

A function must not declare a variable with the same name as a variable that has been defined earlier (including as a parameter) in the same or an outer/contiaining lexical scope (demarcated by curly braces, i.e., {}, or by being a single conditional statement in an `if`/`else` statement or `while` loop). An expression may only access variables declared within the same or an outer/containing 1exical scope. *Thus for any variable name v at any program point, either v can be accessed or declared, but never both.*

Types and conversions. All $\langle expr \rangle$ evaluation—including function actuals, return values, and `free` statements—must be statically type-checked as much as possible, according to the following type hierarchy:



Upcasts and same-casts are permitted to be either implicit (silent) or explicit. Downcasts must be explicit (i.e., must use a cast expression). Any cast that is statically infeasible, i.e., statically between `int` and `Ref` is a static checking error. Statically feasible downcasts are checked at run time.

Immutability. Variables and functions are *immutable* unless declared as `mutable`. An immutable variable must not be assigned a new value using an assignment statement.

An immutable function's body must not contain calls to `mutable` functions (including built-in `mutable` functions).

A call *statement* may only call a `mutable` function.

Miscellaneous. Function calls must have the same number of actuals as the function definition's number of formals.

Every function must be statically guaranteed to return a value. The interpreter's static checking may verify this property by simply checking that the function's last statement is a `return` statement (and reporting an error if not). The reference interpreter performs this simple check.

A function may contain `return` statements that make some code statically unreachable. Statically unreachable code is not erroneous.

A Finding Bugs in the Reference Interpreter

Students who find a bug in the reference interpreter—not including behavior related to first-class functions (Appendix B), memoization (Appendix C), or extended types (Appendix D)—will receive $20 from me. You must be the first to make a public Piazza post demonstrating the bug. You might find a reference interpreter bug without knowing you've found one. It doesn't matter: As long as you've made a public Piazza post explaining an issue that turns out to be a reference interpreter bug, you get credit for it.

B Quandary Extension: First-Class Functions

The Quandary reference interpreter supports an extension to the language described in Section 8: *first-class functions*. Basically, functions can be used as program values, and such values can be used to make function calls.

The reference interpreter's implementation of first-class functions is ad hoc. Briefly, evaluating a function name as a variable identifier yields an integer corresponding to the function. Integer-valued identifiers in turn can be used as function names in call expressions.

As a result, some programs that are incorrect according to Section 8 are accepted by the reference interpreter—in particular, programs that use function names as variable identifiers or make calls using variable identifiers. Furthermore, since the reference interpreter's handling of first-class functions is ad hoc, there are no guarantees for the reference interpreter's handling of programs that use first-class functions incorrectly.

C Quandary Extension: Memoization

The Quandary reference interpreter supports memoization. To enable memoization, use the command-line option `-memoize FUNCTION_NAME` for each function name that memoization should be performed on.

For each call to a memoized function, the interpreter checks if the function has already been called with the same inputs (based on value and/or reference equivalence as appropriate), and return the prior call's return value if so.

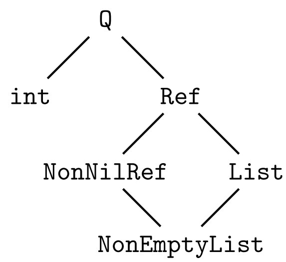
Memoization is intended for programs that do *not* use `mutable` functions. For any program that uses `mutable` functions, memoization should only be used with care; memoization produces unpredictable results if applied to a function that is not pure, i.e., that gives different output for the same inputs.

For correctness, memoization data—function arguments and return values—are used as GC roots in the reference interpreter The interpreter should *not* run out of memory because of memoization. If GC fails to reclaim memory, the interpreter should clear all memoization data and try GC again (or use another technique, such as Java soft references). In the reference interpreter, memoization is incompatible with reference counting GC and with explicit memory management.

D Quandary Extension: More Types

The Quandary reference interpreter supports extended types. To enable extended types, use the command-line option `-et`. By convention, programs that use extended types have file names matching `*.qet`.

With this extension, the following static type hierarchy exists:



The following static rules apply:

- The static type of `nil` is `List`.
- The static type of `expr1 . expr2` is
  - `NonEmptyList` if `expr2`'s static type is `List` (or its subtype `NonEmptyList`); or
  - `NonNilRef` otherwise.
- `free` statements expect an operand of type `NonNilRef`.
- The following built-in functions have different type signatures:
  ```
  Q left(NonNilRef r)
  Q right(NonNilRef r)
  mutable int setLeft(NonNilRef r, Q value)
  mutable int setRight(NonNilRef r, Q value)
  mutable int acq(NonNilRef r)
  mutable int rel(NonNilRef r)
  ```
- The following new built-in functions exist:
  ```
  Q first(NonEmptyList r)
  List rest(NonEmptyList r)
  ```
  They have the same behavior as `left` and `right`, respectively.

The following dynamic (run-time) rules apply:

- The run-time type of value `nil` is `List`.
- The run-time type of a non-`nil` reference value is
  - `NonEmptyList` if the value references an object whose right field was initialized with a value with run-time type `List` (or `NonEmptyList`); or
  - `NonNilRef` otherwise.

Extended types are intended for programs that do not use `mutable` functions. For any program that uses `mutable` functions, using extended types produces unpredictable results—particularly if an object is mutated from a list to a non-list or vice versa.

E Implementation Language

Because the skeleton is written in Java, it's natural to extend the skeleton to implement your interpreter in Java. However, you don't have to extend the skeleton, and you don't even have to use Java—you can use C/C++ or Rust if you like (using any other language requires prior instructor approval).

If you're interested in porting the skeleton to C/C++, note that JFlex and CUP have close equivalents for C/C++: The JFlex manual (`https://jflex.de/manual.html`; "Porting from lex/flex") says that the input file `Scanner.jflex` is similar in format to the format expected by the C/C++ flex tool. Likewise, Java CUP is based on the C/C++ tool YACC, and they use similar input file formats. So you can probably port `Scanner.jflex` to flex, and `Parser.cup` file to YACC.