The Quintessential Quandary Guide

Mike Bond

July 7, 2022

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 technical material and implementation projects.

This guide tries to cover everything related to Quandary, especially 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 packages).

Complete the steps in this section in order.

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 will work mostly fine, except for one issue:

In order for the grading script (`grade.sh`) to work on macOS, you may need to install the correct variant of the `realpath` command using the following command (run from any directory):

```
brew install coreutils
```

If `brew` command fails, you'll need to first get Homebrew (visit `http://brew.sh`).

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

- run Linux in an OS 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 files (`.dat` files) and TODO.

2.2 Get Quandary

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

```
git clone git@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 code examples
- `grading/` contains grading script and test cases
- `quandary.pdf` is this document

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. For VSC 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 VSC.

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

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.

Folks have reported needing a JDK version of at least 11 or 12.

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:

```
~/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 1.7.0: `https://jflex.de/release/jflex-1.7.0.tar.gz`
- CUP 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
```

which should put JFlex 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. 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
```

Even better: Add these commands to your `$HOME/.bashrc`, and they'll be set every time you open a terminal.

Other dependencies that may come up for you: You'll need to install `make`.

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`) aren't set.

Run the skeleton interpreter. Now you can run the skeleton interpreter:

```
~/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 you can run the skeleton interpreter with an input program. As described in Section 3, the skeleton only recognizes "programs" that are simple arithmetic expressions. Thus you'll need to can start by running the skeleton There are a few such example "programs" provided: `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 these simple programs don't have a `main` function. However, the argument is still required.

2.6 What to Do Next

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 **??**. Run the grading script (Section **??**) to evaluate your modified skeleton. Read the rest of this document before attempting the projects.

3 Understanding the Skeleton Interpreter

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

The skeleton is a Java program that starts execution from `Interpreter.main()`...

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.

5 Troubleshooting and Suggestions

5.1 Suggestions for Modifying the Skeleton Interpreter

Don't modify files that are generated automatically by JFlex or CUP. To see which files aren't generated automatically, run `make clean` to eliminate generated files.

You'll need to modify the lexer specification (`sScanner.jflex`) and the parser specification (`Parser.cup`), and modify and add AST files (`ast/*.java`). For later projects you may need to add or modify other Java files (e.g., `interpreter/*.java`).

5.2 Asking for Help

If you can't figure out the answer or find it in this document or on Piazza, the best ways to ask for help are (in order from most to least recommended):

- Make a public post on Piazza.
- Attend instructor or TA office hours.
- Ask in class.

If you're having technical difficulties like getting a weird error, post as much information as possible. If you encounter problems running a script, run it prefaced with `bash -x` 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.

5.3 Finding Bugs in the Reference Interpreter

Students who find a bug in the reference interpreter will receive $20. You must be the first to make a public Piazza post demonstrating the bug.

Often you'll find an issue and not know whether you're doing something wrong or you've found a reference interpreter bug. Don't worry—just make a public Piazza post explaining the issue (you don't need to know you've found a reference interpreter bug to get credit for it).

5.4 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.

6 Grading and Submitting Your Interpreter

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

6.1 The Grading Script and Test Cases

The grading script, `grading/grade.sh`, is the same script that the TA(s) 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
  ```

  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
- `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
```

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 SUCCEED.

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 TODO

When you run the grading script, this test case should

TODO: Note that isrefint.q should FAIL.

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

Test cases. We provide representative public test cases for each project in `grading/*-public.dat`. 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 `.tgz` to Carmen. You can upload as many times as you like—only the latest submission and its timestamp will count.

7 Subsets of Quandary a.k.a. Interpreter Projects

Add some interesting variants

8 Implementation Suggestions and Hints

See last semester's Piazza posts including my pinned post.

9 Quandary Language and Runtime Specification

9.1 Syntax and Semantics

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

```
⟨program⟩ ::= ⟨funcDefList⟩

⟨funcDefList⟩ ::= ⟨funcDef⟩ ⟨funcDefList⟩
               |  ε

⟨funcDef⟩ ::= ⟨varDecl⟩ ( ⟨formalDeclList⟩ ) { ⟨stmtList⟩ }

⟨varDecl⟩ ::= ⟨type⟩ IDENT                       // Variables and functions are immutable by default
           |  mutable ⟨type⟩ IDENT   // Mutable vars can be updated; mutable funcs can perform updates

⟨type⟩ ::= int                                                        // 64-bit signed integer
        |  Ref                // Reference to a heap object with left and right fields of type Q; or nil
        |  Q                                                           // Super type of int and Ref

⟨formalDeclList⟩ ::= ⟨neFormalDeclList⟩
                  |  ε

⟨neFormalDeclList⟩ ::= ⟨varDecl⟩ , ⟨neFormalDeclList⟩
                    |  ⟨varDecl⟩

⟨stmtList⟩ ::= ⟨stmt⟩ ⟨stmtList⟩
            |  ε

⟨stmt⟩ ::= ⟨varDecl⟩ = ⟨expr⟩ ;                               // Declare and initialize variable
        |  IDENT = ⟨expr⟩ ;               // Update to already-declared-and-initialized (mutable) variable
        |  if ( ⟨cond⟩ ) ⟨stmt⟩
        |  if ( ⟨cond⟩ ) ⟨stmt⟩ else ⟨stmt⟩
        |  while ( ⟨cond⟩ ) ⟨stmt⟩                                    // Pointless without mutation
        |  IDENT ( ⟨exprList⟩ ) ;                                 // IDENT must be mutable function
        |  free ⟨expr⟩ ;                        // Frees memory iff explicit memory management enabled
        |  print ⟨expr⟩ ;                              // Prints evaluated value followed by a newline
        |  return ⟨expr⟩ ;
        |  { ⟨stmtList⟩ }

⟨exprList⟩ ::= ⟨neExprList⟩
            |  ε

⟨neExprList⟩ ::= ⟨expr⟩ , ⟨neExprList⟩
              |  ⟨expr⟩
```

```
⟨expr⟩ ::= nil                                  // Special constant value of type Ref
         | INTCONST                             // 64-bit signed integer of type int
         | IDENT
         | - ⟨expr⟩
         | ( ⟨type⟩ ) ⟨expr⟩                    // Need for explicit downcast from Q to int or Ref
         | IDENT ( ⟨exprList⟩ )
         | ⟨binaryExpr⟩
         | [ ⟨binaryExpr⟩ ]     // Evaluates the left and right sides of the binary expression concurrently
         | ( ⟨expr⟩ )

⟨binaryExpr⟩ ::= ⟨expr⟩ + ⟨expr⟩
               | ⟨expr⟩ - ⟨expr⟩
               | ⟨expr⟩ * ⟨expr⟩
               | ⟨expr⟩ . ⟨expr⟩               // Evaluates to a Ref referencing a new heap object

⟨cond⟩ ::= ⟨expr⟩ <= ⟨expr⟩
         | ⟨expr⟩ >= ⟨expr⟩
         | ⟨expr⟩ == ⟨expr⟩                     // For comparing int values only
         | ⟨expr⟩ != ⟨expr⟩                     // For comparing int values only
         | ⟨expr⟩ < ⟨expr⟩
         | ⟨expr⟩ > ⟨expr⟩
         | ⟨cond⟩ && ⟨cond⟩
         | ⟨cond⟩ || ⟨cond⟩
         | ! ⟨cond⟩
         | ( ⟨cond⟩ )
```

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++-style "block" comments `/* like this */`

9.2 Precedence and dangling `else`

Precedence of operators in high-to-low order:

1. Expressions in parentheses (`()`) or brackets (`[]`)
2. `-` used as a unary operator and `(` ⟨*type*⟩ `)` (cast operator)
3. `*`
4. `-` used as a binary operator and `+`
5. `.`
6. `<=`, `>=`, `==`, `!=`, `<`, and `>`
7. `!`
8. `&&` and `||`

All operators are left assocative.

Dangling `else` ambiguity is resolved by matching an `else` with the nearest `if` statement allowed by the grammar.

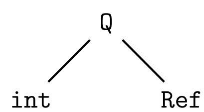
9.3 Static type checking

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 $\langle \mathit{expr} \rangle$ 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., $\boxed{\langle \mathit{expr} \rangle ::= \texttt{(}\ \langle \mathit{type} \rangle\ \texttt{)}\ \langle \mathit{expr} \rangle}$, and are checked at run time.

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

Immutability: Variables and functions are *immutable* unless declared as `mutable`. An immutable variable must not be the assigned-to variable in an assignment statement, i.e., $\boxed{\langle \mathit{stmt} \rangle ::= \texttt{IDENT = } \langle \mathit{expr} \rangle\ \texttt{;}}$.

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: 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).

A function may contain `return` statements that make code statically unreachable. In general, statically unreachable code is not erroneous.

9.4 Built-in functions

`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`

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

9.5 Language semantics (including dynamic type checking) and operation of the interpreter

The interpreter executes the defined function called `main` and passes a command-line parameter as `main`'s argument:

```
$ ./quandary
Expected format: quandary [OPTIONS] QUANDARY_PROGRAM_FILE INTEGER_ARGUMENT
Options:
  -gc (MarkSweep|MarkSweepVerbose|RefCount|Explicit|NoGC)
  -heapsize BYTES
BYTES must be a multiple of the word size (8)
```

The interpreter prints the return value of `main` in the following way:

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

Incorrect command-line parameters, including `QUANDARY_PROGRAM_FILE` not being found, have undefined behavior (i.e., the interpreter may fail in any way).

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 executed 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` should cause a fatal `nil` dereference error at run time.

Memory management: An execution should report 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 and mark–sweep and reference counting garbage collection (and optionally others as well, e.g., semi-space). See command-line arguments above.

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.

Trace-based garbage collection only: An evaluation of an allocation expression ($\langle binaryExpr \rangle$ ::= $\langle expr \rangle$ `.` $\langle expr \rangle$) performs trace-based GC when and only when the non-freed memory exceeds the specified maximum heap size. Trace-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; if trace-based GC is triggered when multiple threads are active, the interpreter has undefined behavior (but ideally it will report an error, to help with debugging).

Concurrency: A concurrently evaluated binary expression `[` $\langle binaryExpr \rangle$ `]` 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* should return one of the following error codes as appropriate:

- 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`) should print this return code. Specifically, the script should handle executions as follows.

1. For a non-erroneous, terminated program, the script should print the following as its last two lines:

   ```
   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),[1] the script should not terminate. Printing anything or nothing is fine.

3. For an execution that should return error code `ERROR_CODE`, the script should print the following as its last line:

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

```
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 fine (including uncaught exceptions). 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.

9.6 Implementing the interpreter

An interpreter written in Java or C++ should allocate heap objects into raw memory (represented by a primitive array in Java, for example), and assume that raw memory provides only low-level load, store, and compare-and-set operations. When writing the interpreter, use the provided `Heap` class to emulate raw memory.