Appendix I: Testing Your Programs

This appendix describes the `lb unit` unit-testing framework. An important part of constructing programs in any language is testing them, and one purpose of this appendix is to get you thinking about constructing tests as you write your programs.

The `lb unit` is invoked by typing `lb unit args`, where `args` is used to indicate the tests you want to run and other options you can supply. Individual tests can be run by using the `--test` *fileName(s)* option, where *fileName(s)* gives the name of the file(s) containing the tests that you want to run.

You can also organize your tests into *suites*, collections of related tests. Similar to the above, you would use the `--suite` *suiteName(s)* option to run suite(s) of tests. Each of them names a directory containing a set of test files to execute. You can even run a series of suites by using the `--suiteDir` *suiteDirectory(ies)* option. In this case, `lb unit` will recursively execute the tests in each of the named directories.

Regardless of how you invoke `lb unit`, it first runs any user-specified setup instructions common to the tests in the directory containing them. The instructions are placed in a file named `setUp.lb`. Similarly, after the tests are run, `lb unit` runs the instructions in the file `tearDown.lb`. Note that all test script files should use the `.lb` suffix.

`setUp.lb`, `tearDown.lb`, and your test files comprise two types of content: LogiQL code and instructions in the form accepted by the `lb` command. (See Appendix A.) For example, here is the setup file as found in the `setUp.lb` file in the resource directory for this appendix:

```
create --unique
```

This instruction tells `lb unit` to create a new workspace with a made up and unique name. Alternatively, instead of `--unique`, you can specify a name for the workspace, such as `ws`.

Similarly, `tearDown.lb` contains the following line:

```
close --destroy
```

which closes and removes the workspace used in the tests.

Besides opening and closing workspaces, `lb unit` instructions can load LogiQL code blocks and data. For example, say we wanted to make sure that a simple declaration like the following had no problems. The following instruction could be used to do this:

```
addblock 'Monarch(m), hasMonarchName(m:s) ->
  string(s).'
exec --file b.logic
```

`addblock` is the `lb` command for adding declarations, rules, and constraints to a workspace. The LogiQL code is contained within apostrophes to prevent inadvertent interpretation by the shell from which you execute `lb unit`. In this example, the `addblock` transaction adds the declarations of `Monarch` and `hasMonarchName` to the workspace created with `setUp.lb`. The second transaction above shows an alternative way to include information in a test. The `--file` option to `exec` requests that `lb unit` obtain information from a file `b.logic`, which contains the following content:

```
+Monarch("George VI").
```

Note the suffix `.logic` should be used for files containing only LogiQL statements.

Of course, the above two transactions do not comprise a very interesting test. If they were executed by `lb unit`, the compiler and runtime would check for simple errors, such as syntax problems. While this can be valuable information, normally, you will want to determine that the program actually computes the intended output. One way to make this check is to define a new predicate and populate it with the output expected. Then the two predicates (the original one and the new one) can be compared.

The following test, based on `chapter1/tests/CE1/q1c.lb`, illustrates this technique. It assumes the same `setUp.lb` and `tearDown.lb` files as shown above. The test's main purpose is to check whether

the computation expressed in the `foreignMonarch` predicate derives the expected answers. Here is the first part of the test providing declarations, constraints, derivation rules, and basic data:

```
addblock '
  Monarch(m), hasMonarchName(m:s) -> string(s).
  Country(c), hasCountryCode(c:cc) -> string(cc).
  wasBornIn[m] = c -> Monarch(m), Country(c).
  Monarch(m) -> wasBornIn[m] = _ .
  foreignMonarch(m) -> Monarch(m).
  foreignMonarch(m) <- wasBornIn[m] = c, c ! = "GB".
'

exec '
  +wasBornIn["Anne"] = "GB".
  +wasBornIn["George I"] = "DE".
  +wasBornIn["George II"] = "DE".
  +wasBornIn["George III"] = "GB".
  +wasBornIn["George IV"] = "GB".
  +wasBornIn["William IV"] = "GB".
  +wasBornIn["Victoria"] = "GB".
  +wasBornIn["Edward VII"] = "GB".
  +wasBornIn["George V"] = "GB".
  +wasBornIn["Edward VIII"] = "GB".
  +wasBornIn["George VI"] = "GB".
  +wasBornIn["Elizabeth II"] = "GB".
'
```

Based upon the above declarations and the derivation rule for `foreignMonarch`, you would expect the derived answers to be "`George I`" and "`George II`". In order to make this check, you can express these expectations by introducing a new predicate, as shown in the following code segments:

```
addblock 'foreignMonarch:expected(m) -> Monarch(m).'
```

The segment defines a new predicate named `foreignMonarch:expected`. The name is conventionally constructed from the name of the predicate being checked, `foreignMonarch`, and the suffix, `expected`, separated by a colon (':').

You can now populate the new predicate with the facts you expect to be contained in the `foreignMonarch` predicate: that is, that George I and George II were not born in Great Britain:

```
exec '
  +foreignMonarch:expected("George II").
  +foreignMonarch:expected("George I").
'
```

If the code above is correct, you would expect the two predicates, `foreignMonarch` and `foreignMonarch:expected`, to be identical. The final segment of the test comprises two constraints that make this check:

```
addblock '
  foreignMonarch:expected(m) -> foreignMonarch(m).
  foreignMonarch(m) -> foreignMonarch:expected(m).
'
```

The first constraint expresses the expectation that all of our predicted answers were, in fact, derived. The second checks that no other outputs were produced.

There is some flexibility into which transactions the various statements are placed. For example, the declaration of `foreignMonarch` could have been placed with the declarations for `Monarch` and `hasMonarchName` in the first transaction above. Also, the first of the two constraints in the last segment could have been placed there. Stylistically, however, we recommend that you place your testing code separate from and after the code being tested. Note however, we could not have put the second constraint with the declarations in the first transaction. Can you see why?

If you place the second constraint in any transaction before the `foreignMonarch:expected` facts were asserted, the test would fail, even if the underlying code was correct. This is because at the moment when the transaction containing the constraints completed, there would, in fact, be two `foreignMonarch` facts for which there were not yet `foreignMonarch:expected` facts, violating the second constraint. The code for this example can be found in the file `example.lb`.

Information about other `lb unit` options can be found by executing `lb unit --help`.