ECE453/CS447/SE465
Software Testing, Quality Assurance, and Maintenance
Assignment/Lab 2, version 0.9

Patrick Lam
Release Date: January 30, 2017

Due: 11:59 PM, Monday, February 27, 2017
Submit: via ecgit

Question 1 (10 points)

In the `shared/icalendarlib` directory, you will find an implementation of an iCalendar `ics` file parser.

Your task is to generate `ics` files to crash `icalendarlib` or to cause memory errors as measured by `valgrind` or other tools. You can do this either by manually hand-crafting `ics` files (not recommended) or else by writing a fuzzing tool to create them.

You can find some sample `ics` files here:
`http://apple.stackexchange.com/questions/125338/calendar-ical-ics-format`

and the `ics` file format specification here:
`https://tools.ietf.org/html/rfc5545`

TODO: more details about `ics`; edit the test harness to take an input file.

Question 2 (10 points)

Consider the following implementation of the cycle-finding algorithm from `http://en.literateprograms.org/Floyd%27s_cycle-finding_algorithm_%28C%29`. (We've included the complete implementation from there in the skeleton at `q2/cycle-finder.c`, including a test harness.) In your `q2/mutants.pdf` file, propose two non-stillborn and non-equivalent mutants of this function. Write down test inputs which strongly kill these mutants (syntax doesn't matter) and the expected output of your test cases on the original code and on the mutant.

```
typedef struct node_s {
        void *data;
        struct node_s *next;
} NODE;

int list_has_cycle(NODE *list)
{
        NODE *fast=list;
        while(1) {
                if(!(fast=fast->next)) return 0;
                if(fast==list) return 1;
                if(!(fast=fast->next)) return 0;
                if(fast==list) return 1;
                list=list->next;
        }
        return 0;
}
```

Question 3 (10 points)

Something about test automation?

Question 4 (10 points)

In your repository, you will find a JavaScript application at `shared/calc/index.html`. I wrote this application (or at least ported it from the Internet). It uses a Pratt parser to parse a simple expression language and evaluate the given expression.

Here is a grammar for this application.

⟨*expr*⟩ ::= number
| `` `-' ⟨*expr*⟩ ``
| `` `+' ⟨*expr*⟩ ``
| ⟨*expr*⟩ `` `+' ⟨*expr*⟩ ``
| ⟨*expr*⟩ `` `-' ⟨*expr*⟩ ``
| ⟨*expr*⟩ `` `*' ⟨*expr*⟩ ``
| ⟨*expr*⟩ `` `/' ⟨*expr*⟩ ``
| ⟨*expr*⟩ `` `^' ⟨*expr*⟩ ``
| `` `(' ⟨*expr*⟩ `)' ``

⟨*number*⟩ ::= [`` `0' - `9' ``]$^+$

Your task is to manually (or otherwise) generate a set of test cases for this application and to use Selenium to automate these test cases. You shouldn't need to look at the application source code. Specifically:

- (2 points) Generate 10 test cases for the application. Specify the input along with the actual and expected output. Some of the inputs should be within the language recognized by the application, while others should be outside the language. Be sure to choose interesting test cases.
- (8 points) Use Selenium to automate your 10 test cases. In the `shared/selenium` directory in your repo, you will find a `SeleniumExample`. You can run tests from this example using the command:

  ```
  mvn test "-Dtest=se465.SeleniumExample#test*" -Dwebdriver.base.url=http://www.google.com
  ```

  Your test suite should be called `se465.CalcSuite` and we should be able to run your tests with the command:

  ```
  mvn test "-Dtest=se465.CalcSuite#test*" \
       -Dwebdriver.base.url=file:///<put your a2 directory here>/shared/calc/index.html
  ```

  We will check that your tests exercise the functionality that you specified in the first part.
- (5 bonus points) Fix bugs in the application (submit a diff).

Question 5 (10 points)

If all goes according to plan, I'll describe the Page Object design pattern in Lecture 19. Your task is to create Page Objects for the JavaScript calculator. This will allow your tests to generalize to `francais.html` and `eval.html`.

- (6 points) In this part, we'll create page objects. Create a generic Page Object interface `CalculatorPageObject`, along with an implementation `OriginalCalculatorPageObject`, which summarizes the UI elements on the `index.html` page. It should allow you to access the controls that your tests from Question 4 need. Also, create an additional Page Object `FrancaisCalculatorPageObject` which implements the same interface but which works with `francais.html`.

- (4 points) Copy `se465.CalcSuite` to `se465.RefactoredCalcSuite` and `se465.RefactoredFrancaisCalcSuite`. Modify both of the refactored suites to use the page objects that you created for the first part. (This could be done via dependency injection as well, but we haven't talked about that. However, I will accept a single `RefactoredCalcSuite` if you include instructions on how to inject dependencies.)

References for this question:

`http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#chapter06-reference`
`https://martinfowler.com/bliki/PageObject.html`