ECE453/CS447/SE465
Software Testing, Quality Assurance, and Maintenance
Assignment/Lab 2, version 0.96

Patrick Lam
Release Date: February 2, 2017

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

Getting set up

Same as last time, but this time `a2` instead of `a1`. There is a Vagrant VM description, but you shouldn't need it.

Tools used: python (sort of optional); g++/make (very optional); Selenium (required).

Submission summary

Here's what you need to submit in your fork of the repo. Be sure to commit and **push** your changes back to `ecgit`.

1. in directory `q1`, your modified `generate.py` file (unless you've mailed me to tell me that you're not using Python);
2. in directory `q2`, file `mutants.pdf` as described in the question;
3. in directory `q3`, either file `testcases.txt` or `testcases.pdf`, which includes your test cases and the minimized test case and explanation; and, in directory `shared/selenium/src/test/java/se465`, file `CalcSuite.java` containing your test cases;
4. in directory `shared/selenium/src/test/java/se465`, files `CalculatorPageObject.java`, `OriginalCalculatorPageObject.java`, `FrancaisCalculatorPageObject.java`, along with files `RefactoredCalcSuite.java` and `FrancaisCalcSuite.java`.

You may choose to move the `q1` and `q2` files into the `shared` subdirectory if you want to access them in Vagrant. We'll mark submissions of `q1` and `q2` either in the original place or under `shared`.

Question	TA in Charge
1	(mostly machine)
2	TBA
3	TBA
4	TBA
5	TBA

Question 1 (10 points)

Your task is to write code to randomly generate valid iCalendar `.ics` files. You can find the specification at

`https://tools.ietf.org/html/rfc5545`

Although you will be marked against the full specification, following the description of the subset here will suffice.

- an `ics` file starts with a `BEGIN:VCALENDAR` line and ends with an `END:VCALENDAR` line. It must also contain a `VERSION` and `PRODID` line. (The provided code generates these for you.)
- your `ics` file should contain a sequence of events. Each event starts with `BEGIN:VEVENT` and ends with `END:VEVENT` and contains a set of components. Each event must contain UID, DTSTAMP and DTSTART components. Valid iCalendar events may contain other components, and we'll be checking that your events do contain some other components.
- the `UID` must be a string that is unique per-event;
- the `DTSTAMP` and `DTSTART` components must contain dates/times, which are yyyymmddThhmmssZ, with T and Z literal and dates/times as appropriate.
- other valid components include `LOCATION`, `SUMMARY`, and `DTEND`.

You will find a `generate.py` file with skeleton code in the `q1` directory of your repo. You can also find a number of sample `ics` files in the `shared/icalendarlib` directory. I can run the Python file with `python generate.py` at a prompt.

You are not required to use the skeleton `generate.py`; if you choose to write your own generator, email me (Patrick) to let me know. I suspect that the path of least resistance is using the provided skeleton. You just need to add new rules to the CFG at the end of `generate.py` and new special productions to `generate_special_production`. (My solution adds 4 special productions and 9 `add_prod` lines to the CFG; you don't really need to know Python to solve this problem, just pattern-match against what's there already.)

Marking scheme. We will mark this by running your code 20 times to generate `ics` files. We then check that the files are different and that they include more than zero events and also more than zero optional components. Finally, we'll check your generated files for validity against the iCal spec using an automated tool. (Details to come).

Bonus (0 marks). Can you create valid `ics` files that break the app in `shared/icalendarlib`? (I can't.)

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

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)

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. The `^` operator is supposed to denote exponentiation.

$$
\begin{aligned}
\langle expr \rangle ::= \ & \text{number} \\
| \ & \text{`-'} \ \langle expr \rangle \\
| \ & \text{`+'} \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \text{`+'} \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \text{`-'} \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \text{`*'} \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \text{`/'} \ \langle expr \rangle \\
| \ & \langle expr \rangle \ \text{`\^{}'} \ \langle expr \rangle \\
| \ & \text{`('} \ \langle expr \rangle \ \text{`)'}
\end{aligned}
$$

$$
\langle number \rangle ::= [\text{`0'} - \text{`9'}]^{+}
$$

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, but you do need to look at the HTML of `shared/calc/index.html`. Specifically:

- (2 points) Generate 10 test cases for the application. Specify the input along with the actual and expected output. Some of the inputs must be within the language recognized by the application, while others must be outside the language. Be sure to choose interesting test cases.
- (3 points) The following test case exposes a bug in the application:

  ```
  2+(8*3/4)^4*(1+3+5)*2-4*(3/2)+9*3
  ```

  Minimize this test case. That is, produce a minimal-length subset of this test case—not necessarily using contiguous characters—which shows the same error. Explain the cause of the bug as well as the expected output.
- (5 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 this command:

  ```
  mvn test "-Dtest=se465.CalcSuite#test*"
  ```

  We will check that your tests exercise the functionality that you specified in the first part. **HINT:** You can't get the text from an input element with `getText()`. See instead `http://www.w3schools.com/jsref/prop_text_value.asp`.

  Useful reference:

  ```
  https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/WebElement.html
  ```

Question 4 (20 points)

If all goes according to plan, I'll describe the Page Object design pattern in Lecture 19 (and hope to have notes up before that). Your task is to create Page Objects for the JavaScript calculator. This will allow your tests to generalize to `francais.html` (and even `eval.html`, though we won't use that).

- (10 points) In this part, we'll create page objects. Create a generic Page Object interface `se465.CalculatorPage`, along with an implementation `se465.OriginalCalculatorPage`, which encapsulates the UI elements on the `index.html` page. (I created the implementation first and then reverse-engineered the interface). These objects should allow you to access the controls that your tests from Question 4 need. Also, create an additional Page Object `se465.FrancaisCalculatorPage` which implements the same interface but which works with `francais.html`.
- (10 points) Copy `se465.CalcSuite` to `se465.RefactoredCalcSuite`. Modify the refactored suite to use both of the page objects that you created for the first part; you should have two tests in the refactored suite for each test in the original suite. It's OK for your English and French tests to essentially be the same except for the choice of the WebDriver, or you could be more clever. (This could be done via JUnit Parameters, but we haven't talked about that.)

References for this question:

`http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#chapter06-reference`[1]
`https://martinfowler.com/bliki/PageObject.html`

[1] use a `WebDriver` rather than a `Selenium` object as in the example.