ECE653
Software Testing, Quality Assurance, and Maintenance
Assignment 1 (70 Points), Version 1

Instructor: Werner Dietl
Release Date: May 12, 2019

Due: 21:00, Friday, June 7, 2019
Submit: An electronic copy on GitHub

The GitHub repository with the source code and test cases for the assignment can be obtained using `https://classroom.github.com/a/NmeY6hWI`.

An account on `eceUbuntu.uwaterloo.ca` is available to you. Several of the resources required for this assignment might be already installed on these servers, but can also be downloaded independently. If you are attempting to connect to a server from off campus, remember you will need to connect to the University's VPN first: `https://uwaterloo.ca/information-systems-technology/services/virtual-private-network-vpn/about-virtual-private-network-vpn`

You will need a working Python environment to complete the assignment. Follow the instructions below to set one up on `eceUbuntu` or on your personal machine:

- `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/02/python`
- `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/02/virtual-box`
- `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/02/virtual-env-intro`
- `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/07/docker-tutorial`
- `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/19/docker-for-windows`

Please note to use `eceUbuntu` instead of `ecelinux`.

You can also use a provided Docker container. Instructions on how to install it are available at `https://ece.uwaterloo.ca/~agurfink/stqam/tutorial/2019/01/07/docker-tutorial`. We will expect your assignments to work in the container.

I expect each of you to do the assignment independently. I will follow UW's Policy 71 for all cases of plagiarism.

Submission Instructions:

Please read the following instructions carefully. **If you do not follow the instructions, you may be penalized up to 5 points.** Illegible answers receive no points.

Submit by pushing your changes to the `master` branch of your GitHub repository. The repository must contain the following:

- a `user.yml` file with your UWaterloo user information;

- a single pdf file called `a1_sub.pdf`. The first page must include your full name, 8-digit student number and your uwaterloo email address;
- a directory `a1q3` that includes your code for Question 3; and
- a directory `wlang` that includes your code for Question 4.

After submission, **please view your submissions on GitHub to make sure you have uploaded the right files/versions**.

You can push changes to the repository before and after the deadline. We will use the latest commit at the time of deadline for marking.

Question 1 (10 points)

Below is a faulty *Python* program, which includes a test case that results in a failure. Not all array elements are considered. A possible fix is to change line 7 as follows:

```
    for j in range(0, i + 1):
```

Answer the following questions for this program:

(a) If possible, identify a test case that does not execute the fault.

(b) If possible, identify a test case that executes the fault, but does not result in an error state.

(c) If possible, identify a test case that results in an error, but not a failure.

(d) For the test case `x = [4, 0, -2, 3]` the expected output is 5. Identify the first error state. Describe the complete state.

(e) Using the minimum number of nodes (9), draw a Control Flow Graph (CFG) of the function `max_prefix`. Include your diagram in `a1_sub.pdf`. The CFG should be at the level of basic blocks. Use the line number of the first statement of the basic block to mark the corresponding CFG node. See the lecture notes on *Structural Coverage* and *CFG* for examples.

```
1  def max_prefix(arr):
2      """Ensures: returns the maximum sum of array elements
3                  in a prefix, or 0 if the prefix is empty"""
4      res = 0
5      for i in range(0, len(arr)):
6          temp = 0
7          for j in range(0, i):
8            temp = temp + arr[j]
9
10         if temp > res:
11           res = temp
12
13     return res
14
15 # x = [4, 0, -2, 3]
16 # r = max_prefix(x)
17 # assert(r == 5)
```

Question 2 (15 points)

Recall the WHILE language from the lecture notes. We are going to introduce an additional statement **repeat-until** with the following syntax:

$$\textbf{repeat } S \textbf{ until } b$$

where S is a statement, and b is Boolean expression.

(a) Following the example in the lecture notes, define a Python class `RepeatUntilStmt` to represent the Abstract Syntax Tree node for the repeat-until statement. Include the source code for the class in `a1_sub.pdf`.

(b) Informally, the semantics of the **repeat-until** loop is that in each iteration: (a) the S statements is executed, and (b) the b expression is evaluated. If b evaluates to $false$, the loop continues to the next iteration. Otherwise, the loop terminates and statements following the loop (if any) are executed. Formalize this semantics using the judgment rules of the Natural Operational Semantics (big-step) **WITHOUT** using the rules of the **while** loop.

(c) Use your semantics from part (b) to show that the following judgment is valid:

$$\langle x := 2 \,;\, \textbf{repeat } x := x - 1 \textbf{ until } x \leq 0, [\,]\rangle \Downarrow [x := 0]$$

(d) Use your semantics from part (b) to prove that the statement

$$\textbf{repeat } S \textbf{ until } b$$

is semantically equivalent to

$$S \,;\, \textbf{while not } b \textbf{ do } S$$

Question 3 (15 points)

(Adapted from the original version by Patrick Lam and Lin Tan.)

Consider the following (contrived) program:

```
1  class M (object):
2      def m (self, arg, i):
3          q = 1
4          o = None
5
6          nothing = Impossible ()
7          if i == 0:
8              q = 4
9          q = q + 1
10         n = len (arg)
11         if n == 0:
12             q = q / 2
13         elif n == 1:
14             o = A ()
15             B ()
16             q = 25
17         elif n == 2:
18             o = A ()
19             q = q * 100
20             o = B ()
21         else:
22             o = B ()
23         if n > 0:
24             o.m ()
25         else:
26             print 'zero'
27         nothing.happened()
28
29 class A (object):
30     def m (self):
31         print 'a'
32 class B (A):
33     def m (self):
34         print 'b'
35 class Impossible (object):
36     def happened(self):
37         pass
38
39 def main (argv):
40     obj = M ()
41     if len (argv) > 1:
42         obj.m (argv [1], len (argv))
```

(a) Using the minimal number of nodes, draw a Control Flow Graph (CFG) for method `M.m()` and include it in your `a1_sub.pdf`. The CFG should be at the level of basic blocks. Use the line number of the first statement of the basic block to mark the corresponding CFG node.

(b) List the sets of Test Requirements (TRs) with respect to the CFG you drew in part (a) for each of the following coverage: node coverage (NC); edge coverage (EC); edge-pair coverage (EPC); and prime path coverage (PPC). In other words, write four sets: TR_{NC}, TR_{EC}, TR_{EPC}, and TR_{PPC}. If there are infeasible test requirements, list them separately and explain why they are infeasible.

(c) Using `a1q3/coverage_tests.py` as a starting point, write unit tests that achieve, for method `M.m()`, each of the following coverages: (1) node coverage but not edge coverage; (2) edge coverage but not edge-pair coverage; (3) edge-pair coverage but not prime path coverage; and (4) prime path coverage. In other words, you will write four test sets (groups of test functions) in total. One test set satisfies (1), one satisfies (2), one satisfies (3), and the last satisfies (4), if possible. If it is not possible to write a test set to satisfy (1), (2), (3), or (4), explain why. For each test written, provide a simple documentation in the form of a few comment lines above the test function, listing which TRs are satisfied by that test. Consider feasible test requirements only for this part.

You can execute the tests using the following command:

```
python -m a1q3.test
```

Question 4 (30 points)

Your GitHub repository includes an implementation of a parser and interpreter for the WHILE language from the lecture notes. Your task is to extend the code with two simple visitor implementations and to develop a test suite that achieves complete branch coverage.

The implementation of the interpreter is located in directory `wlang`. You can execute the interpreter using the following command:

```
(venv) $ python -m wlang.int wlang/test1.prg
x: 10
```

A sample program is provided for your convinience in `wlang/test1.prg`

(a) *Statement coverage.* A sample test suite is provided in `wlang/test_int.py`. Extend it with additional test cases (i.e., test methods) to achive complete statement coverage in `wlang/parser.py`, `wlang/ast.py`, and `wlang/int.py`. If complete statement coverage is impossible (or difficult), provide an explanation for each line that was not covered. Refer to lines using `FILE:LINE`. For example, `ast.py:10` refers to line 10 of `wlang/ast.py`. Fix and report any bugs that your test suite uncovers.

To execute the test suite use the following command:

```
(venv) $ python -m wlang.test
x: 10
```

To compute coverage of your test suite and to generate an HTML report use the following command:

```
(venv) $ coverage run -m wlang.test
(venv) $ coverage html
```

The report is generated into `htmlcov/index.html`. For more information about the `coverage` command see `https://coverage.readthedocs.io/en/coverage-4.3.1`.

For your convenience, a readable version of the grammar is included in `wlang/while.lean.ebnf`, and a picture of the grammar is included in `wlang/while.svg`.

(b) *Branch coverage.* Extend your test suite from part (a) to complete branch coverage. If complete branch coverage is impossible (or difficult), provide an explanation for each line that was not covered. Fix and report any bugs that your test suite uncovers.

To compute branch coverage, use the following command:

```
(venv) $ coverage run --branch -m wlang.test
(venv) $ coverage html
```

Explain what can be concluded about the interpreter after it passes your test suite?

(d) *Statistics Visitor* Complete an implementation of a class `StatsVisitor` in `wlang/stats_visitor.py`. `StatsVisitor` extends `wlang.ast.AstVisitor` to gather number of statements and the number of variables in a program. An example usage is provided in the `wlang/test_stats_visitor.py` test suite.

Extend the test suite to achieve complete statement coverage of your implementation.

(e) *Undefined Use Visitor.* An assignment to a variable is called a *definition*, an appearance of a variable in an expression is called a *use*. For example, in the following statement

```
x := y + z
```

variable `x` is defined and variables `y` and `z` are used. A variable `u` is said to be *used before defined* (or *undefined*) if there exists an execution of the program in which the *use* of the variable appears prior to its definition. For instance, if the statement above is the whole program, then the variables `y` and `z` are undefined.

As another example, consider the program

```
1    havoc x;
2    if x > 10 then
3       y := x + 1
4    else
5      z := 10 ;
6    x := z + 1
```

In this program, z is undefined because it is used before being defined in the execution 1, 2, 3, 6.

Complete an implementation of a class `UndefVisitor` in `wlang/undef_visitor.py` that extends `wlang.ast.AstVisitor` to check a given program for all undefined variables. The class must provide two methods: `check()` to begin the check, and `get_undefs()` that returns the set of all variables that might be used before being defined. An example usage is provided in the `wlang/test_undef_visitor.py` test suite.

Extend the test suite to achieve complete statement coverage of your implementation.