ECE653
Software Testing, Quality Assurance, and Maintenance
Assignment 1 (70 Points), Version 2

Instructor: Arie Gurfinkel
Release Date: September 11, 2023

Due: October 6, 2024
Submit: An electronic copy on GitLab

Any source code and test cases for the assignment will be released in the skeleton repository at `https://git.uwaterloo.ca/stqam-1239/skeleton`.

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 `main` branch of your GitLab repository in directory `a1/`. Make sure to use a web browser to check that your changes have been committed! The submission 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, **review your submissions on GitLab web interface 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 (the example shows the expected result)[1]. The dimensions of matrix `b` are not computed correctly. A possible fix is to change line 8 as follows:

```
p1, q = len(b), len(b[0])
```

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 cause an error.

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

(d) For the test case `a = [[5, 7], [8, 21]], b = [[8], [4]]` the expected output is `[[68], [148]]`:

$$\begin{bmatrix} 5 & 7 \\ 8 & 21 \end{bmatrix} \begin{bmatrix} 8 \\ 4 \end{bmatrix} = \begin{bmatrix} 8 \cdot 5 + 4 \cdot 7 \\ 8 \cdot 8 + 4 \cdot 21 \end{bmatrix} = \begin{bmatrix} 68 \\ 148 \end{bmatrix}$$

Identify the first error state. Describe the complete state that includes the process counter `pc`.

(e) Using the minimum number of nodes (8), draw a Control Flow Graph (CFG) of the function `matmul`. Treat *list comprehension*[2] as a single statement, explicit exceptions with **raise** as function exit, and ignore implicit exceptions. 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. You can use GraphViz[3] to produce the graph.

```
1  def matmul(a, b):
2      """
3      Returns the result of multiply two input matrices.
4
5      Raises an exception when the input is not valid.
6      """
7      n, p = len(a), len(a[0])
8      q, p1 = len(b), len(b[0])
9      if p != p1:
10         raise ValueError("Incompatible dimensions")
11     c = [[0] * q for i in range(n)]
12     for i in range(n):
13         for j in range(q):
14             c[i][j] = sum(a[i][k] * b[k][j] for k in range(p))
15     return c
16
17 # >>> a = [[5, 7], [8, 21]]
18 # >>> b = [[8], [4]]
19 # >>> matmul(a, b)
20 # [[68], [148]]
```

[1] The program is adapted from `http://www.rosettacode.org/wiki/Rosetta_Code`.

[2] `https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions`

[3] `https://dreampuf.github.io/GraphvizOnline`

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:

1. S is executed;
2. b is evaluated;
3. if the current value of b is *false*, the loop continues to the next iteration;
4. if the current value of b is *true*, the loop terminates (and statements following the loop are executed).

Formalize this semantics using the judgment rules of the Natural Operational Semantics (big-step) **WITHOUT** using the rules of the **while** loop. That is, it should be possible to use your semantics to create a language with **repeat-until** loop, even if the language does not already have a **while** loop.

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

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

That is, construct a derivation tree using your rules and other rules for the language with the above judgment being the consequence.

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

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

is semantically equivalent to

$$S\,;\textbf{if } b \textbf{ then skip else } (\textbf{repeat } S \textbf{ until } b)$$

That is, show that **reapeat-until** can be unfolded once without changing its meaning in any execution.

Hint: The proof of Lemma 2.5 in "Semantics with Applications: An Appetizer" is useful for this question: `https://link.springer.com/chapter/10.1007/978-1-84628-692-6_2`[4].

[4] To access, use `http://testtube.uwaterloo.ca/makelink.cfm`.

Question 3 (15 points)

Consider the following program from `http://www.rosettacode.org/wiki/Tokenize_a_string_with_escaping`:

```
1  def token_with_escape(inpt, escape="^", separator="|"):
2      """
3      Issue python -m doctest thisfile.py to run the doctests.
4
5      >>> print(token_with_escape('one^|uno||three^^^^|four^^^|^cuatro|'))
6      ['one|uno', '', 'three^^', 'four^|cuatro', '']
7      """
8      result = []
9      token = ""
10     state = 0
11     for c in inpt:
12         if state == 0:
13             if c == escape:
14                 state = 1
15             elif c == separator:
16                 result.append(token)
17                 token = ""
18             else:
19                 token += c
20         elif state == 1:
21             token += c
22             state = 0
23     result.append(token)
24     return result
```

(a) Using the minimal number of nodes (11), draw a Control Flow Graph (CFG) for it 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). In other words, write three sets: TR_{NC}, TR_{EC}, TR_{EPC}. 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 each of the following coverage: (1) node coverage but not edge coverage; (2) edge coverage but not edge-pair coverage; (3) edge-pair coverage but not prime path coverage. In other words, you will write three test sets (groups of test functions) in total. One test set satisfies (1), one satisfies (2), and one satisfies (3), if possible. If it is not possible to write a test set to satisfy (1), (2), or (3), 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. For this part of the question consider feasible test requirements only. That is, if some TRs are infeasible, exclude them form the list of requirements and document why they have been excluded.

You can execute the tests using the following command:

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

Question 4 (30 points)

The skeleton GitLab 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) $ python3 -m wlang.int wlang/test1.prg
x: 10
```

A sample program is provided for your convenience 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 achieve 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) $ python3 -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-5.3.1/`.

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`.

The grammar is produced using Tatsu parser generator: `https://github.com/neogeny/TatSu`.

(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 the 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.