SE465/ECE653
Software Testing and Quality Assurance
Assignment 2, version 1*

Patrick Lam

Release Date: January 9, 2026

Due: 11:59 PM, Friday, March 13, 2026
Submit: via git.uwaterloo.ca

Getting set up

We will create a copy of the starter repo for you in your `git.uwaterloo.ca` account. You need to log in to `git.uwaterloo.ca` for that to work.

I expect each of you to do the assignment independently. As stated in the course outline, you can ask questions of generative AI, but you cannot submit text or code that comes from GenAI. I will follow UW's Policy 71 for all cases of plagiarism.

Submission instructions:

Commit **and push** your modifications back to your fork on `git.uwaterloo.ca`. It's git, so you can submit multiple times. After submission, **please make a fresh clone of your submission to make sure you have uploaded all necessary files**.

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 `git.uwaterloo.ca`.

- Q1: (a), (c) to your repo in `a2-mutation-fuzzing`, (b) to Crowdmark
- Q2: (a) and (b) to your repo in `a2-grammar-fuzzing`; (c) to Crowdmark; (d, i), (d, ii), and (d, iii) go into your repo; (d, iv) to Crowdmark.
- Q3: (a), (b) to Crowdmark, (c) to your repo in `a2-reducing-inputs`.
- Q4: all to your repo in `a2-property-based-testing`.
- Q5: all to Crowdmark.

[*] version 1: initial release

Question 1: Mutation Fuzzing

(a) (2 points) Python's `decimal` module supports "decimal fixed-point and floating-point arithmetic". Write a `decimal_fuzzer()` function in `a2-mutation-fuzzing/decimal_fuzzer.py` that creates a `decimal.Decimal` as follows: it draws between 1 and 5 decimal digits to form an integral part, then between 0 and 3 decimal digits to form a fractional part, and a random sign (+ or -). The skeleton also includes class `RandomFuzzer` in `fuzzer.py`, and `decimal_fuzzer.py` includes a `__main__` that invokes it.

(b) (4 points) In this question we'll work on the `MutationCoverageFuzzer` and the `crashme` example from Lecture 8. Work through enough calls to `MutationCoverageFuzzer.run()`, to reach the `Exception`. You must show at least 3 calls to `run()`. Start with the seed `["good"]`. As you are working through the algorithm, you can choose any mutated input you want, but it has to be something that could be generated by the `MutationCoverageFuzzer` (explain how). Show the newly-generated input, along with the population and the coverage at each step. You can show the coverage as a set of line numbers from the above listing. Submit this answer to Crowdmark.

(c) (14 points) (From the *Fuzzing Book*:) When adding a new element to the list of candidates, AFL actually does not compare the coverage, but instead adds an element if it exercises a new branch.

In the `a2-mutation-fuzzing` directory you'll find `mutation_coverage_fuzzer_branches.py` as well as the original `mutation_coverage_fuzzer.py`. You can call either of these from the Python command line. Using branch coverage from the exercises of the "Coverage" chapter, implement this "branch" strategy in the `run()` method of `mutation_coverage_fuzzer_branches.py`. Can you notice any difference compared to the previous strategy?

Question 2: Grammar Fuzzing

In the `a2-grammar-fuzzing` directory in your repo you'll find `symbol_costs.py` and other files.

(a) (2 points) Change the implementation of `grammar_and_symbol_with_cost_7` so that it returns a grammar and symbol with symbol cost 7, when evaluated with empty `seen` set.

(b) (2 points) Change the implementation of `second_grammar_and_symbol()`, `symbol_cost_with_seen_Z()`, as well as `symbol_cost_with_seen_Y()`, such that the `symbol_cost()` of the returned symbol varies with different `seen` sets. The `seen` sets have to be sets that would get computed by an actual invocation to the `GrammarFuzzer`.

(c) (2 points) If you run `fuzz_process_numbers.py` as found in the repo (under `code/L10`) a bunch of times, you'll notice that the last number is pretty much always the longest number. How could you modify the provided `fuzz_process_numbers.py` to get more balanced number lengths?

(d) (14 points total) We'll now modify `GrammarFuzzer` in `a2-grammar-fuzzing/grammar_fuzzer.py` to perhaps get more balanced number lengths (maybe?).

(i) (4 of 14) I've provided a stub `descendants()` implementation. Write code that meets my specification: given `tree`, return a list of triples `(d, parent, index)` for each descendant in `tree`. Implementation hint: append direct children first, then recursively add descendants.

(ii) (6 of 14) Modify `expand_tree_once` to use this new `descendants()` function instead of picking a direct child. (In some ways this simplifies the existing code a bit).

(iii) (2 of 14) I've provided two test suites: `test_descendants.py` and `test_expand_tree_with_descendants.py`. But, testing `expand_tree` is tricky, because it randomly chooses a descendant. Modify the test case to be deterministic and to check the output of `expand_tree_once`.

(iv) (2 of 14) Is it actually true that these modifications produce more balanced inputs? Make some sort of argument for or against and provide some evidence.

Parts (a) and (b) go into your repo; (c) is for Crowdmark; (d, i), (d, ii), and (d, iii) go into your repo; (d, iv) is for Crowdmark.

Question 3: Reducing Inputs (20 points)

(a) (2 points) Give an example of a class of inputs for which grammar-based reduction works well, but which is not a programming/expression language (e.g. arithmetic expressions, Python, Rust, Java, etc.) Let's say: something that is not Turing-complete and not something that appears as a subset of some other language. Submit this answer to Crowdmark.

(b) (4 points) Here is a grammar-based reduction (using the depth concept) of `1 + (2 / (3 - 3)) * (5 + 7)`:

```
Test #1 '(2 / (3 - 3)) * (5 + 7)' 23 FAIL
Test #2 '(5 + 7)' 7 FAIL
Test #3 '(7)' 3 FAIL
Test #4 '7' 1 PASS
(7)
```

Explain which reductions have taken place to obtain each of the tests, including the type of reduction, the node being reduced, and what it is replaced by. This is also a Crowdmark answer.

(c) (14 points) (From the *Fuzzing Book*):

Grammar-based input reduction, as sketched above, might be a good algorithm, but is by no means the only alternative. One interesting question is whether "reduction" should only be limited to elements already present, or whether one would be allowed to also create new elements. These would not be present in the original input, yet still allow producing a much smaller input that would still reproduce the original failure.

As an example, consider the following grammar:

```
<number> ::= <float> | <integer> | <not-a-number>
<float> ::= <digits>.<digits>
<integer> ::= <digits>
<not-a-number> ::= NaN
<digits> ::= [0-9]+
```

Assume the input 100.99 fails. We might be able to reduce it to a minimum of, say, 1.9. However, we cannot reduce it to an <`integer`> or to <`not-a-number`>, as these symbols do not occur in the original input. By allowing to create alternatives for these symbols, we could also test inputs such as `1` or `NaN` and further generalize the class of inputs for which the program fails.

In file `reducer.py` in the `a2-reducing-inputs` subdirectory of your `a2` repo, you'll find `GenerativeGrammarReducer`, a subclass of `GrammarReducer`. Implement overridden method `symbol_reductions()` which uses the `GrammarFuzzer` (already imported) to generate new subtrees as one more option that `symbol_reductions()` can create.

I've also provided a driver `generative_grammar_example.py` in the same directory. It sets a deterministic seed. You can run `python3 generative_grammar_example.py` to test your implementation.

```
~/c/s/a/a/a2-reducing-inputs> python3 generative_grammar_example.py
Test #1 '9.36' 4 PASS
Test #2 '5.8' 3 PASS
Test #3 '84.99' 5 PASS
Test #4 '100.9' 5 PASS
Test #5 '7.99' 4 PASS
Test #6 '100.8' 5 PASS
Test #7 'NaN' 3 FAIL
NaN
```

Marking note: I don't know how many different implementations we'll see, but any implementation that satisfies the specification here gets full marks. We will first try to mark your implementation by setting the seed to a value for marking and running your implementation; if it matches ours on that seed, then you get full marks. Otherwise we'll have to hand mark.

Question 4: Property-Based Testing (15 points)

The Hypothesis library supports generating Python `decimals` and `fractions`. Use the equality operator == for all comparisons in this question.

(a) (6 points) Write property-based tests for the three tests in `a2-property-based-testing/decimals.py`; they check that $d = -(-d)$; $d + 0 = 0$; and $d_1 + d_2 = d_2 + d_1$. You need to add a `@given` annotation to make these functions into Hypothesis tests, and you'll need to use `.normalize()` on the `Decimal`. You also want your tests to pass, so you'll need the right parameters at `@given`.

(b) (2 points) It turns out that + is not associative for `Decimal`, even when `NaN` and `inf` are excluded. Fill in the property-based test in `failing_decimals.py` that demonstrates this.

(c) (4 points) Moving on to fractions, fill in the implementations in `fractions.py`, which check that $q + (-q) = 0$ and that $(q_1 + (q_2 + q_3)) = ((q_1 + q_2) + q_3)$.

(d) (3 points) Coming back full circle, write a composite Hypothesis strategy that creates a `Decimal` similarly to the one in Question 1. However, this time, for the integral part, concatenate (as strings) between 1 and 5 non-negative integers; for the fractional part, concatenate between 0 and 3 non-negative integers; draw the sign randomly; and construct the `Decimal` from these.

Then, write a property-based test that again checks whether add is associative for the `Decimals` that you create with your strategy.

Question 5: Symbolic Execution (10 points)

Submit the answer to this question on Crowdmark. Consider the following program Prog1:

```
1   havoc (x, y)
2   if x + y > 15:
3     x = x + 7
4     y = y - 12
5   else:
6     y = y + 10
7     x = x - 2
8
9   x = x + 2
10
11  if 2 * (x + y) > 21:
12    x = x * 3
13    y = y * 2
14  else:
15    x = x * 4
16    y = y * 3 + x
17  pass
```

(a) (3 points) How many execution paths does Prog1 have? List all the paths as a sequence of line numbers taken on the path.

(b) (4 points) Symbolically execute each path and provide the resulting path condition. Show the steps of symbolic execution as a table. An example of executing the first line is given below:

Edge	**Symbolic State**	**Path Condition (PC)**
$1 \to 2$	$\mathrm{x} \mapsto X_0, y \mapsto Y_0$	true
...	...	...

(c) (3 points) For each path in part (b), indicate whether it is feasible or not. For each feasible path, give values for X_0 and Y_0 that satisfy the path condition.