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

Patrick Lam
Release Date: January 9, 2026

Due: 11:59 PM, Friday, March 6, 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`.

- TBA

*version 1: initial release

Question 1: Mutation Fuzzing

(2 points) Python's `decimal` module supports "decimal fixed-point and floating-point arithmetic". Write a `decimal_fuzzer()` function 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 includes class `RandomFuzzer` in `fuzzer.py`.

Question 2: Grammar Fuzzing

Question 3: Reducing Inputs

put a delta debugging question

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.
>
> Create a class GenerativeGrammarReducer as subclass of GrammarReducer; extend the method reduce_subtree() accordingly.

Question 4: Property-Based Fuzzing

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)

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.