**SE 465 – Software Testing and Quality Assurance**
**ECE 653 – Software Testing, Quality Assurance & Maintenance**
Instructor: Patrick Lam

**Instructions:**

- You have **80 minutes** to complete the exam.
- You can refer to **printed or handwritten** material, e.g., books, slides, notes, etc., as well as **reference** material saved on your computer.
- You may not do searches for information on the Internet (no browsers, please; local document viewers only).
- You may use Python and Python coverage tools.
- You may not ask anyone or any automated system (eg LLMs) for help with your exam, either in person or online.
- If information appears to be missing from a question, make a reasonable assumption, state your assumption, and proceed.

## Question 1: Short Answer (15 points)

Answer these questions in one or two sentences. Each is worth 3 points.

(a) In general terms, why is static analysis subject to false positives but not dynamic analysis?

(b) In metamorphic testing, what do we do to validate an observed test output?

(c) When we compute a test suite's statement coverage, we often report a result as a percentage of lines covered. Why is a percentage a more useful number than an absolute number of lines covered?

(d) Conversely, coverage-guided fuzzing does not need to know the number of statements that can potentially be covered. Why not?

(e) With respect to extracting configuration grammars, what are 2 assumptions that the technique from the *Fuzzing Book* rely on?

## Question 2: Mutation Analysis (20 points)

Consider the following function witten in the Python programming language.

```
1  def string_conversion(inpt, literal="^", separator="|", override="@"):
2      """ Example:
3      >>> print(string_coversion('@xone|uno^^|'))
4      ['one', 'unox', 'x', '', '']
5      """
6      result = []
7      token = ""
8      state = 0
9      literal_char = '*'
10     for c in inpt:
11         if state == 0:
12             if c == override:
13                 state = 1
14             elif c == separator:
15                 result.append(token)
16                 token = ""
17             elif c == literal:
18                 token += literal_char # line 18
19                 result.append(token)
20                 token = ""
21             else:
22                 token += c
23         elif state == 1:
24             literal_char = c
25             state = 0
26     result.append(token)
27     return result
```

(a) (10 points) Propose inputs that achieve 100% statement coverage of `add_lists` as a series of calls to that function. Give the inputs in the same format as "Examples" above.

(b) (8 points) In class, we talked about how, in practice, tools generate mutants only for lines that have changed. Here, assume that line 18 was changed in the most recent commit. Propose two non-stillborn and non-equivalent mutants of line 18. Write down mutated versions of the code. Clearly indicate (with comments in the code) how you mutated the program and give a name for the mutation operator you applied (which you may make up).

(c) (10 points) Additionally, write unit tests that kill both of your mutants, in the same format as for part (a). Show that your tests succeed on the original program and fail on the mutant, e.g. by calculating and writing down outputs of the tests on the original and new code. You have a computer, so you are allowed to use Python to run code. If you added new test cases for this part, discuss whether they provide more insight than the ones for part (a). If you did not need to add new test cases, discuss what that says about statement coverage and mutation analysis on this example.

## Question 3: Random and Coverage-Guided Fuzzing (20 points)

Here is a method.

```
1  def crash_midterm(inp:str) -> None:
2      c = 0
3      if len(inp) == 0:
4          return
5      if inp[0] >= '1' and inp[0] <= '9':
6          c = int(inp[0])
7      if c+4 <= len(inp) and inp[c] == '%':
8          if inp[c+1] == '*':
9              if inp[c+2] == '&':
10                 if inp[c+3] == '<':
11                     raise Exception()
```

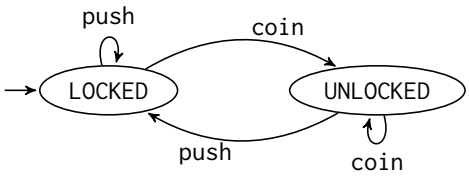
(a) (2 points) What is an input that would cause this method to reach the `raise Exception()` statement? (Don't forget that you can check your answer on your computer!)

(b) (4 points) Would you expect random fuzzing, as implemented in `fuzzer()` from L07 (with its default parameters, notably a `char_start` of 32 and a `char_range` of 32) to find the raised exception on `crash_midterm()` in 100,000 trials? In at most two sentences, say why or why not.

(c) (14 points) Work through enough calls to `MutationCoverageFuzzer.run()`, as described in Lecture 8, to reach the `Exception`. You must show at least 3 calls to `run()`. Start with the seed `[""]`. 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.

## Question 4: Oracles (15 points)

Here is a Finite State Machine modelling a subway turnstile, which controls access to the platforms. You have to insert a coin to unlock the turnstile, at which point you can push the turnstile to return to the initial locked position. The turnstile provides methods `push()` and `coin()` for actions and function `get_state()` which returns either `LOCKED` or `UNLOCKED`.




(a) (3 points) Give three specific examples of implicit oracles—observable behaviours of programs that always indicate that something is wrong with the program. An example should be no longer than a sentence.

(b) (4 points) Write test cases (4) that cover all of the state transitions for this FSM and check that the proper state is reached. Each test case may assume that it starts in the `LOCKED` state.

(c) (8 points) In the notes, it says that regression testing uses an earlier version as an oracle. Write a function, a change to that function, and a test case, such that the earlier version is a valid oracle to the new version; and, a change and test case such that the earlier version is not a valid oracle.

## Question 5: Grammar Fuzzing (15 points)

Here is the BNF grammar `PROCESS_NUMBERS` from lecture,

```
{ '<start>': ['<operator> <integers>'],
  '<operator>': ['--sum', '--min', '--max'],
  '<integers>': ['<integer>', '<integers> <integer>'],
  '<integer>': ['<digit-1>'],
  '<digit>': ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'],
  '<digit-1>': ['<digit>', '<digit><digit-1>'] }
```

and here is a modified subset of `EXPR_GRAMMAR`.

```
{ '<start>': ['<factor>'],
  '<factor>': ['+<factor>', '-<factor>', '<integer>.<integer>.<integer>'],
  '<integer>': ['8'] }
```

(a) (5 points) You are allowed to run Python code on this midterm, so you can see that `symbol_cost` of <`integers`> is 4. Write down an explanation of why this is the case.

(b) (8 points) Write down the steps that `expand_node_by_cost` takes, when `choose` is `max` and `choose_node_expansion()` is random. Your `DerivationTree` is { ('<`factor`>', None) }.

(c) (2 points) Give an example of a grammar where <`start`> has symbol cost `inf`.

## Question 6: Delta Debugging (20 points)

This is delta debugger output for the `MysteryRunner` in Lecture 11 on a randomly generated input, rounded up to make the numbers nicer. Call the original input, in test #1, "`inp`".

```
Test #1 ' 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92
              +$1<(3%&5\'\'>#000' 100 FAIL
Test #2 '3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#000' 50 PASS
Test #3 " 7:,>((/$$-/->.;.=;(.%!:50#7*8=$&&=$9!%6(4=&69':'<" 50 PASS
Test #4 '0#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+1<2!4$>92+$1<(3%&5\'\'>#000' 75 FAIL
Test #5 "0#7*8=$&&=$9!%6(4=&69':'<1<2!4$>92+$1<(3%&5''>#000" 50 PASS
Test #6 '0#7*8=$&&=$9!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+' 50 FAIL
Test #7 '3+0-3.24#7=!&60)2/+";+<7+' 25 PASS
Test #8 "0#7*8=$&&=$9!%6(4=&69':'<" 25 PASS
Test #9 '!%6(4=&69\':\'<3+0-3.24#7=!&60)2/+";+<7+' 38 FAIL
```

You probably want to talk about the current value of `n` as computed in the `DeltaDebuggingReducer`.

(a) (2 point) Explain how you can get the substrings for test #2 and test #3, as a function of `inp`.

(b) (4 points) What about test #4? What is it derived from?

(c) (8 points) What is the relationship between test #4 and tests #5/#6? Why are there only 2 tests of length 50 after test #4, rather than 3 tests?

(d) (6 points) How is the input for test #9 computed, as a function of previous tests?