Software Testing, Quality Assurance & Maintenance (ECE453/CS447/SE465): Midterm

February 17, 2017

This open-book midterm has 5 questions and 80 points. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1: Test Design (10 points)

Sentence: Splitting the test makes it easier to pinpoint the cause of the failure because of the descriptive test name and because the tests run more quickly.

```
1   @Test
2   public void testStatic() throws Throwable {
3     // L3
4   }
5
6   @Test
7   public void testDeprecated() throws Throwable {
8     // L4
9   }
10
11  @Test
12  public void testReturnTypes() throws Throwable {
13    // L5, L6
14  }
15
16  @Test
17  public void testParameterTypes() throws Throwable {
18    // L7
19  }
20
21  @Test
22  public void testExceptionTypes() throws Throwable {
23    // L8
24  }
```

Question 2: Fuzzing (10 points)

```
1   void naiveComputeAngleTest() {
2     Random r = new Random();
3     computeAngle(r.nextInt(), r.nextInt(), r.nextInt());
4   }
```

This is the bottleneck I mentioned in the notes. Most of the time the execution of the method under test would simply stop at the `IllegalArgumentException` and not reach the key `atan2` call. So, no, `naiveComputeAngleTest` would not give you good insight into what `computeAngle` does.

For the next part, I asked the TAs to be fairly lenient, especially with respect to rounding issues. In particular it was OK to assume that the `sqrt` was close enough often enough. There are more clever solutions possible. But I did ask you to never call `computeAngle` with bad args, so you had to filter that out.

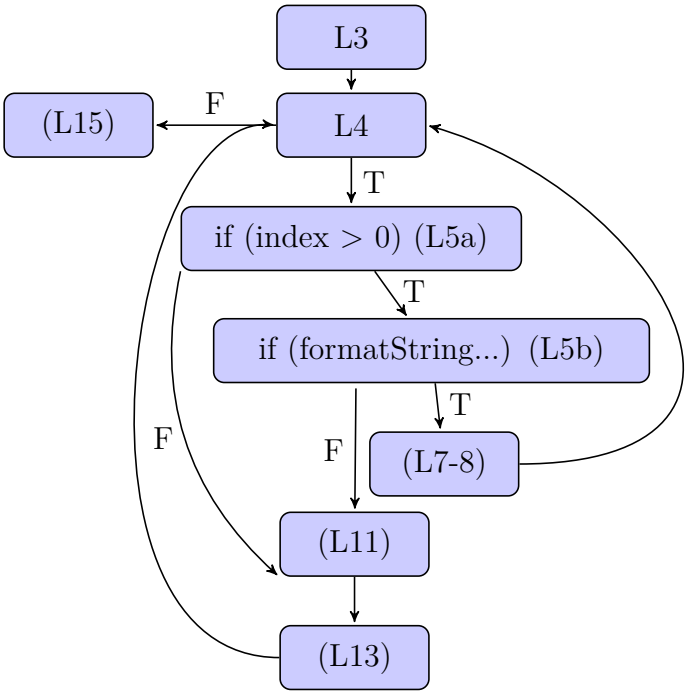
```
1   void smartComputeAngleTest() {
2     Random r = new Random();
3     while (true) {
4       int x = r.nextInt(), y = r.nextInt();
5       int z = Math.sqrt(x*x + y*y);
6       if (z*z == x*x + y*y) {
7         computeAngle(x, y, z);
8       }
9     }
10  }
```

A more clever solution (thanks Jun!) would be:

```
1   void smartComputeAngleTest() {
2     Random r = new Random();
3     while (true) {
4       int a = r.nextInt(), b = r.nextInt();
5       computeAngle(a*a, b*b, a*a + b*b + 2*a*b);
6     }
7   }
```

Question 3: Short-circuit evaluation (15 points)

Whoops. I had 8 nodes. Sorry. Also, there was a missing return, which you might choose to include or not.



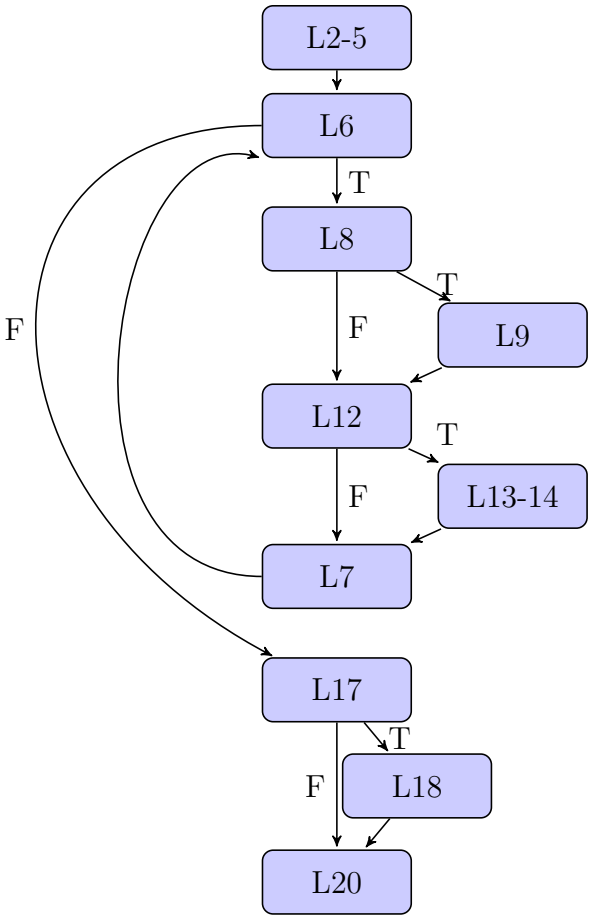
The missing branch is L5a true, L5b false (i.e. `(index > 0) && !formatString.charAt(index - 1) == '!'`); the case `"$"` is L5a false and done, while the case `"!$"` is L5a true, L5b true and done.

The question was somewhat ambiguous about whether the case should achieve 100% branch coverage on its own or combined with the two previous test cases. Due to ambiguity, TAs should accept either one, but it's pretty easy to make a standalone case that achieves 100% branch coverage: `"$!$$"` achieves all of the previous branches and also includes a case where `formatString.charAt(index - 1) == '!'`.

Question 4: Statement and Branch Coverage (20 points)

Sorry again. I added a node at the last minute. For reference:

```
1  public static List<String> wrap(String input, int line_length) {
2    List<String> rv = new ArrayList<String>();
3    int last_break = -1, last_space = 0;
4
5    for (int i = 0;
6         i < input.length();
7         i++) {
8      if (input.charAt(i) == ' ') {
9        last_space = i;
10     }
11
12     if (i - last_break > line_length) {
13       rv.add(input.substring(last_break + 1, last_space));
14       last_break = i;
15     }
16   }
17   if (last_space >= last_break + 1) {
18     rv.add(input.substring(last_break + 1, last_space));
19   }
20   return rv;
21 }
```



The argument should proceed node-by-node. Lines 2--5 and L6 are obvious enough that they don't need to be mentioned. Continuing to Line 8, we initially have `i = 0` and `input.length > 0`, so we definitely enter the loop. We also observe that the loop iterates over every possible `i` (it doesn't skip any). We reach line 9 because the input string contains spaces. Once we're in the loop, Line 12 is unavoidable. `last_break` starts at -1 and is only changed inside the if, and `i - 1` exceeds `line_length` because we observe more than one line in the output, so we definitely execute Lines 13--14. Finally, Line 7 is unavoidable inside the loop as well. Because the program terminates, we definitely exit the loop and reach Line 17. Finally, we reach Line 18 because `last_space` is 13 on exit while `last_break` is 9. Line 20 is obvious.

The branches that might not be covered given statement coverage are L8--L12, L12--L7, and L17--L20. L8--L12 is executed on a non-space input, which the input clearly includes. L12--L7 also gets executed for characters that don't cause line breaks, which clearly exists because we observe lines that are longer than 1 letter long. This input does not execute the branch L17--L20 because we only execute L17 once and, as we argued above, that execution takes the true branch.

Question 5: Understanding Mutation (25 points)

For part (a), there are lots of non-trivial mutants. One such mutant is to replace the constant "1" on line 18 by the constant "0". This will cause the second line of the output to start with a space. The mutant is non-trivial: any input which doesn't wrap (e.g. same `input`, `line_length = 40`) does not kill the mutant.

Part (b): Test case `input = "one three two", line_length = 9` without trailing space has the same expected output (`one three\ntwo`) and actual output `one three`.

Part (c) depends on your answer from part (b). Another mutant is on Line 17: changing `"+ 1"` to `"+ 0"` will cause a crash on my input from (b). On the other hand, input `input = "one three two ", line_length 9` shows that the mutant is non-trivial. (I'd intended a fix for the bug but that's actually harder than it seems, especially without a computer.)