Com S 417
Software Testing

Fall 2017 – Week 2, Lecture 4

Announcements

- Lab 1 due Sept 5 midnight.
  - In the future labs will be due just before class so that we can discuss immediately.
- Homework due Sept 7
- My office hours effective now:
  - Tues 2:10 – 3,
  - Wed 4-5, and
  - by appointment
- Reminder: Syllabus schedule is TENTATIVE – and not reliably updated

Reading Quiz

Model-Driven Testing

Selecting tests based on abstractions drawn from the code.

Programs and models

- Good code is more than just a collection of symbols and operators.
  - It expresses a solution design
  - That design is often based on one or more models.
- Control structures
  - Object-centered design - classes
  - Functional decomposition – functions
  - Loops – for, while, do..until
  - Branches – if..then..else, switch, select, choose

Information about design models underlying the code can help us select more effective tests.

Control flow graphs (1.14)

A graphical technique for representing the control structures (and related flow of execution) in a program.

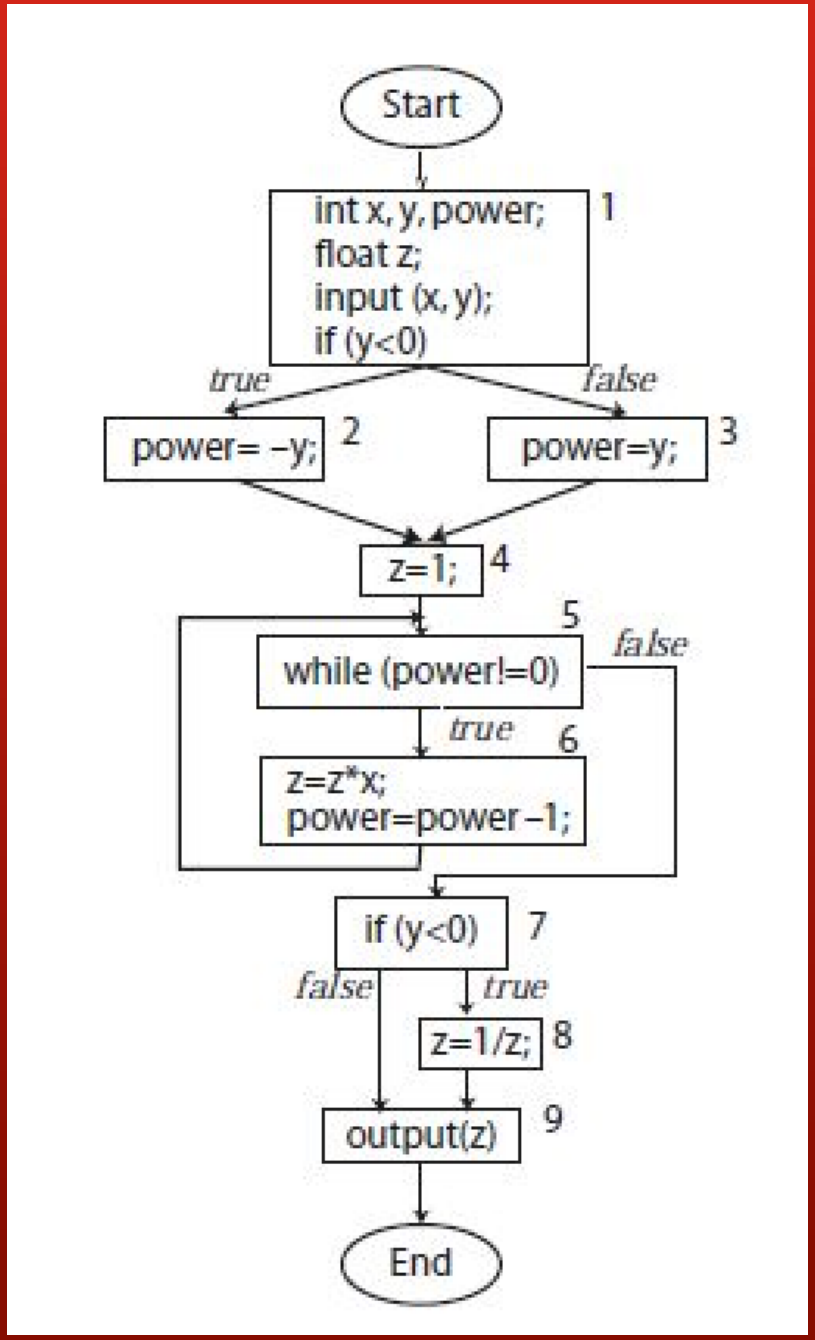
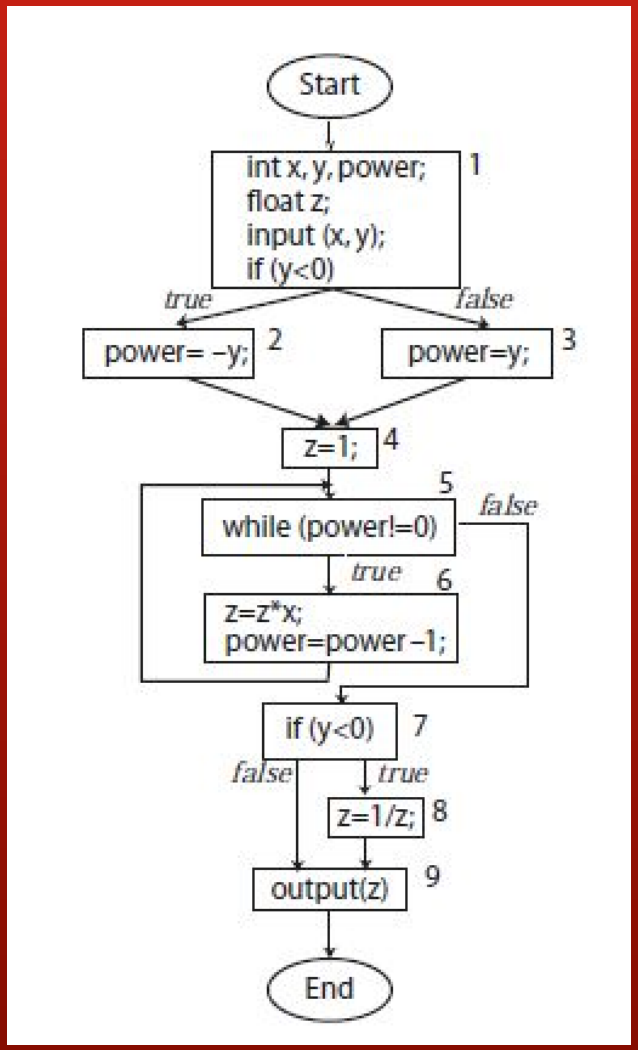
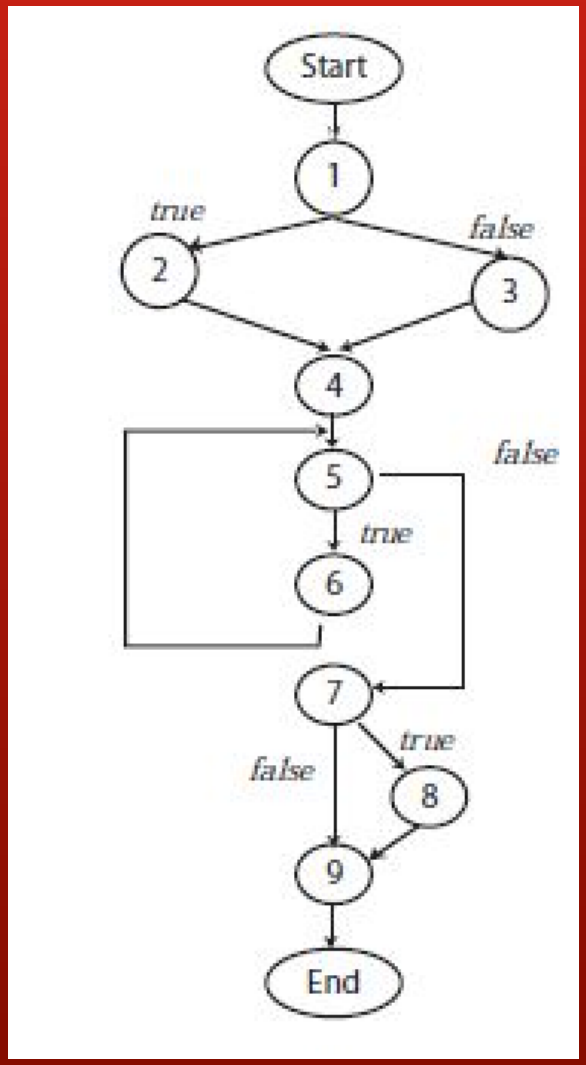


Figure from: Mathur

Reducing detail to coverage related specifics.





Figures from: Mathur

CFG Details

Basic Block: a section of code which will always execute sequentially.

- If any statement in the block executes, then all statements in the block will execute.

Nodes represent basic blocks.

Arcs represent non-sequential control structures; i.e., transitions between basic blocks

- if, switch, while, etc.

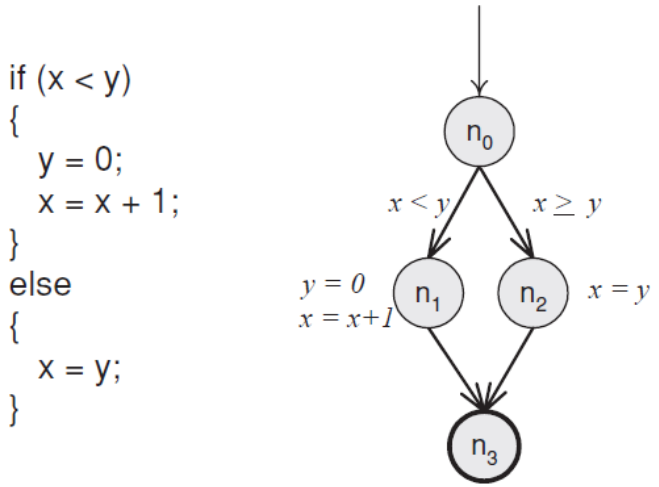


Figure 2.16. CFG fragment for the if-else structure.

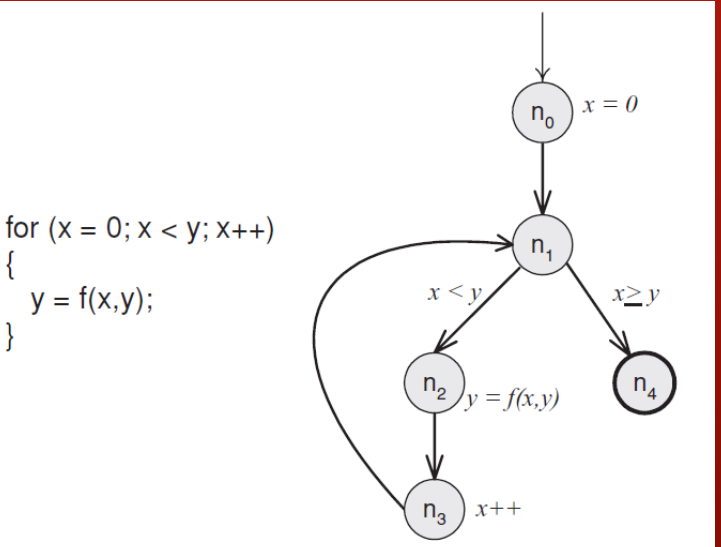


Figure 2.19. CFG fragment for the for loop structure.

Figures from Ammann and Offutt

Graph coverage with Flow Graphs

How do the graphs help us select tests?

You can think of each basic block or node as an “equivalence class” based on execution code.

- Any input that gets you to one of the lines in the block, automatically exercises all of the other lines.
- You can easily focus on inputs that affect the conditions controlling flow.

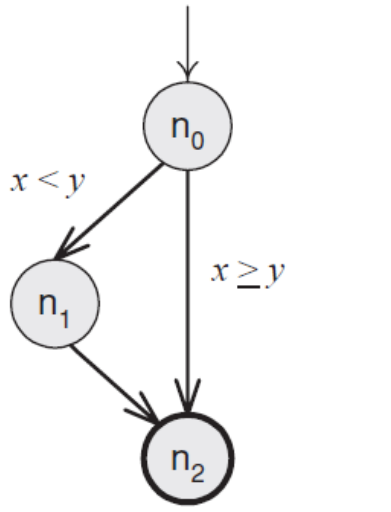
What rule would you use to select high-coverage tests?

We will describe our coverage criteria in terms of sets of paths with certain properties.

Node Coverage

We select test sets that correspond to certain paths which, combined, visit all reachable nodes in the graph.

- simple
- corresponds to 100% branch coverage.



path (t_1) = [n_0, n_1, n_2]
path (t_2) = [n_0, n_2]

Edge Coverage

We select test sets that traverse all reachable edges in the graph.

- Augmentation of node coverage
- corresponds to a simple form of conditional coverage

$T_1 = \{ t_1 \}$
T_1 satisfies node coverage on the graph

$T_2 = \{ t_1 , t_2 \}$
T_2 satisfies edge coverage on the graph

Junit take 2 – scaling tests

Lab 2

Scalable test sets (Lab 2)

- It should be clear from our look at black box tests that test sets can grow very rapidly.
- Do we really want to write something like this for every member of a test set?

```
@Test
public void testNotReachedx() {
    CharACounter test = new CharACounter();
    int output = test.countA("");
    assertEquals(0,output);
}
```

- Does it make sense to force someone to understand java just to add a new entry to a test set?
- Java's "RunWith" capability and the helper class "Parameterized" make it much easier to construct and maintain large automated test sets.

Enhanced test management

Junit supports “pluggable” runners.

- runners are responsible for identifying the tests to be run and helping junit run each.

Using the @RunWith annotation, you can replace the default runner with a runner better suited to your particular testing strategy/problems.

@RunWith(Paramaterized.class) let’s you “reuse” a single test with any number of test values.

- Think of this option as allowing you to create table-driven test sets.

Basic Structure for RunWith

- RunWith annotation
- global variables for individual test values
- a constructor which receives values for one test case.
- An Object[][] structure (the parameter table)
- The test.

Skeletal Code

```
/* Runner-related imports.
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.junit.runner.RunWith;


@RunWith(Parameterized.class)
public class CounterTest {

   public String input;
   public Integer expected Result;

  /* a constructor ... Parameterized uses this to "inject"
     the test inputs for each test case .
     Parameterized will expect the width of the param table
     to match the number of constructor arguments.

   */

   public CounterTest(String input, Integer expectedResult) {
     this.input = input;
     this.expectedResult = expectedResult;
   }
```

Skeletal Code – part 2

```
  @Parameterized.Parameters
  /*
  public static Collection getTestSet() {
     return Arrays.asList(new Object[][] {
        {"ok", 1 },
        { "book", 2 },
        { "flat", 0 }
     });
  }

  @Test
  public void testCounter() {
     int result = counter.countO(input);
     System.out.format("test %02d %-14s %2d%n", testId++, input, result);
     assertEquals(expectedResult, counter.countO(input));
  }
}
```

Making test sets maintainable

- Lab 2 will require you to go two steps beyond the basic Parameterized behavior.
  - First you will move all the test values into a tab-delimited text file. This allows the test set to be maintained (without risk of breaking code) in a spreadsheet and then exported to tab-delimited text for execution.
  - Second, you will eliminate the “expected results” from the file and instead, rely upon an “Oracle” to determine if the SUT generated the right result.

Lab 1 questions/information

- 1) tests must be capable of failing, or they aren't testing anything.
- 2) you are supplying the each test with a specific input, but each test must be capable of producing the correct outcome with any input value we might choose to use.
- 3) you can put instrumentation in your SUT, but not junit assertions. Remember, the production code will not include *any* code from the test source hierarchy. The production package will NOT include the junit library.

Questions

Reading, Last Lecture, Lab?

Reading Assignment

Junit RunWith information from the web:

- **Understanding JUnit's Runner architecture**

  http://www.mscharhag.com/java/understanding-junits-runner-architecture

  Nice description of the Runner api and how it fits into the Junit architecture.

- **Extended Parameterized Example**

  http://blog.icodejava.com/727/junit-a-complete-java-unit-testing-example-with-advance-features-source-code-and-diagram/

  Also links to other junit materials.