Topics for this Lecture

- Test Adequacy Criteria
- Statement, Branch,…. Coverage

Test Adequacy Criteria

- A key problem in software testing is selecting and evaluating test cases
- How do we know a test suite is “good enough”?
- Ideally, we should like an “adequate” test suite to be one that ensures correctness of the program under test.
- But that is **impossible**!
  - Adequacy of test suites, in the sense above, is provably ***undecidable***.
  - As we have seen, we can't try **all possible executions** of a program.
- Judging test suite thoroughness based on the structure of the program itself
  - Also known as “white-box”, “glass-box”, or “code-based” testing

Test Adequacy Criteria

- A test adequacy criterion is a predicate that is true (**satisfied**) or false (**not satisfied**) of a ⟨program, test suite⟩ pair.
- Adequacy criterion is usually expressed in form of a rule (e.g., "all statements must be covered")
- A test suite satisfies an adequacy criterion if
  - all the tests succeed (pass)
  - every test rule in the criterion is satisfied by at least one of the test cases in the test suite.
- Example:

The ***statement coverage adequacy criterion*** *is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was "pass".*

Coverage adequacy criterion

- A coverage criterion describes a finite subset of test cases out of the infinite number of possible tests we should execute.
- Coverage criteria serve two purposes:

1. **Adequacy**: Have we generated enough tests?
   - To know what we have & haven't tested.

2. **Guidance**: Where should we test more?
   - To know when we can "safely" stop testing.

- How can we *measure* "how much testing" we have done and look for more things to test?
  - We could look *structurally* – what aspects of the *source code* have we tested?

Test Coverage

- Test Coverage: a measure of the proportion of a program executed/exercised during testing.
- Advantages:
  - Gives us an objective score.
  - When coverage is <100%, we are given meaningful task.
- Disadvantages:
  - Not very helpful in finding errors of omission.
  - Difficult to interrupt scores <100%.
  - 100% coverage does not mean all bugs were found.

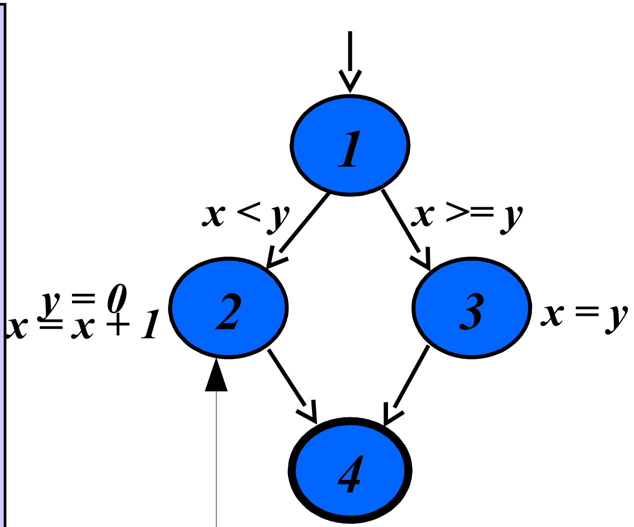
Coverage Metrics

- A large part of the literature of software testing is actually concerned with various notions of coverage.
- Some basic kinds of coverage:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Data flow (def-use) coverage
  - Condition (logic) Coverage
  - Many more – most common kind of coverage, by far.

Statement/Basic Block Coverage

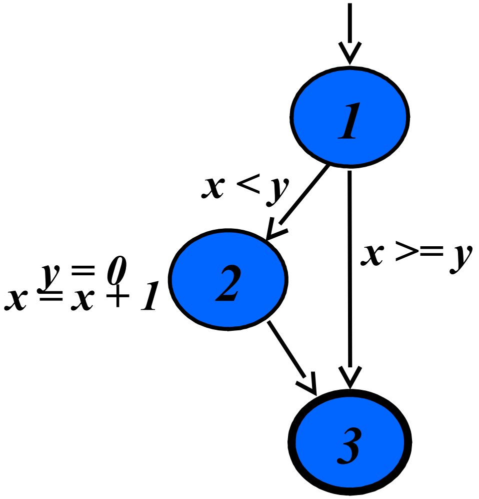
- Statement Coverage Adequacy criterion: each statement (or node in the CFG) must be executed/covered at least once.
- Coverage: $\dfrac{\mathit{Number\ of\ executed\ statements}}{\mathit{Number\ of\ all\ statements}}$

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Treat as one node because if one statement executes the other must also execute (code is a basic block)

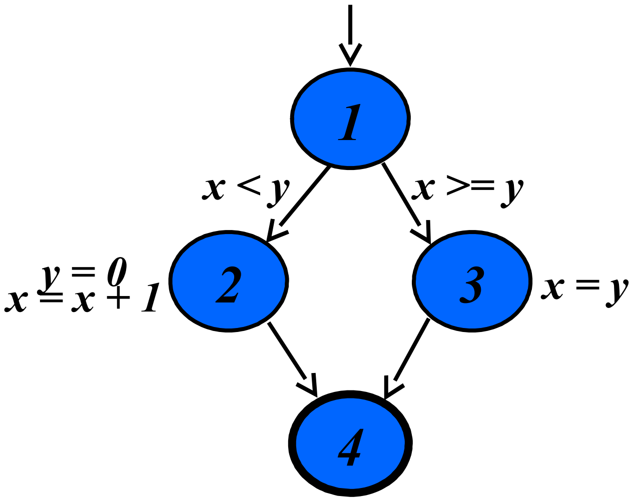
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



Branch Coverage

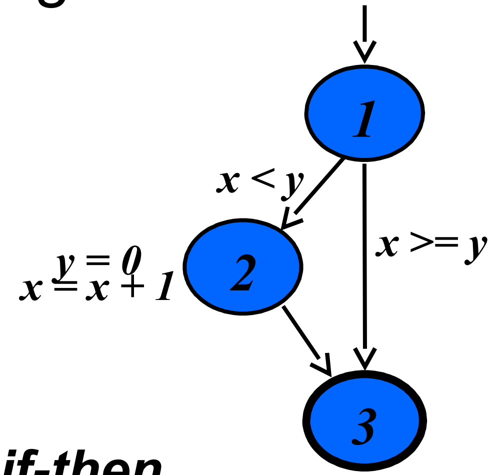
- Branch Coverage Adequacy criterion: each branch in the CFG must be executed at least once
- Coverage: $\dfrac{\textit{Number of executed branches}}{\textit{Number of all branches}}$
- ***Subsumes*** statement testing criterion
  - because traversing all edges implies traversing all nodes

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Branch coverage vs. statement coverage: Same for if-then-else

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



Consider this if-then structure. For branch coverage can't just cover all nodes, but must cover all edges – get to node 3 both after 2 and without executing 2!

Why is branch testing useful?

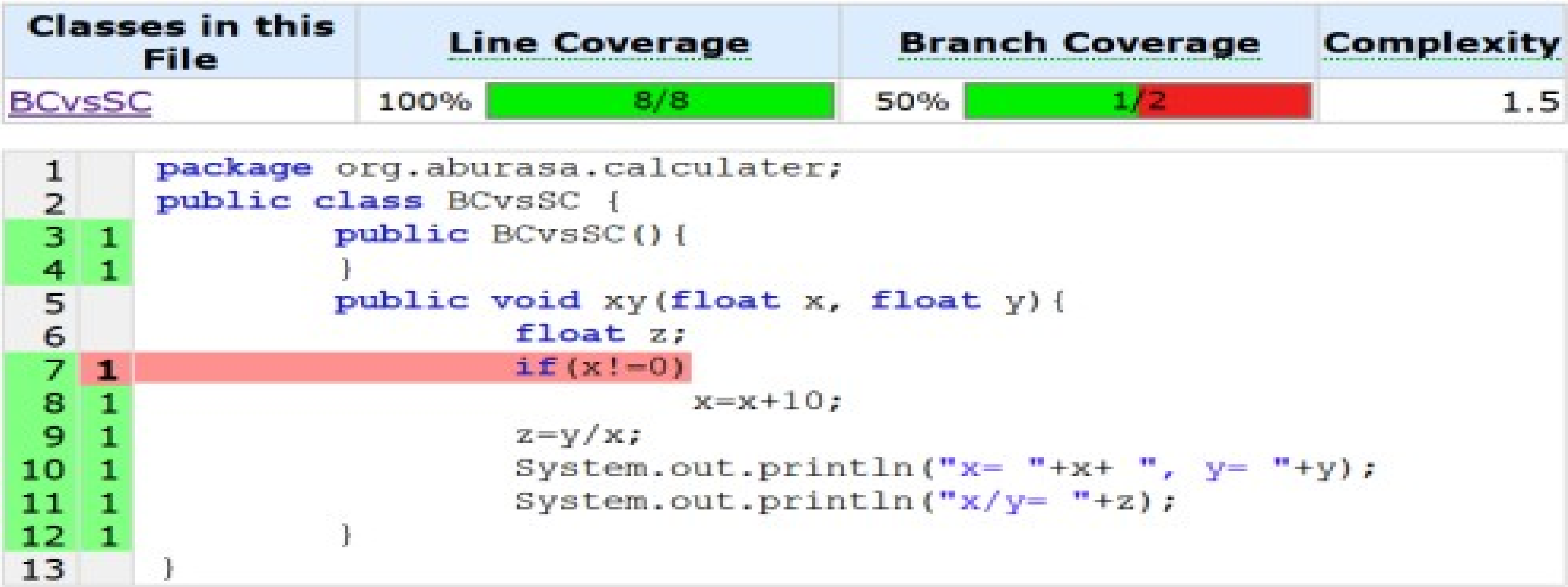
- Consider the method shown below.

```
1.public class BCvsSC {
2.   public BCvsSC(){
3.    }
4.    public void xy(float x, float y){
5.      float z;
6.      if(x!=0)
7.           x=x+10;
8.      z=y/x; //a fault statement
9.      System.out.println("x= "+x+ ", y= "+y);
10.      System.out.println("x/y= "+z);
11.    }
12.}
```

- If you test this method with the goal of 100% statement coverage, you would need to run only a single test case with x=10 and y=10.
- And you would quit testing, right?
- **Wrong**, such test does not reveal the fault @ statement 7
- To reveal it, we need to traverse edge 6-8 ==> Branch Coverage.

Why is branch testing useful?

This is output of the cobertura tool for the test case (x=10, y=10)



Classes in this File	Line Coverage	Branch Coverage	Complexity
BCvsSC	100% 8/8	50% 1/2	1.5

```
 1      package org.aburasa.calculater;
 2      public class BCvsSC {
 3   1          public BCvsSC(){
 4   1          }
 5              public void xy(float x, float y){
 6                      float z;
 7   1                  if(x!=0)
 8   1                          x=x+10;
 9   1                  z=y/x;
10   1                  System.out.println("x= "+x+ ", y= "+y);
11   1                  System.out.println("x/y= "+z);
12   1          }
13      }
```

Each statement (each line) is annotated with the number of executions. The red color stands for branches/lines without executable code.

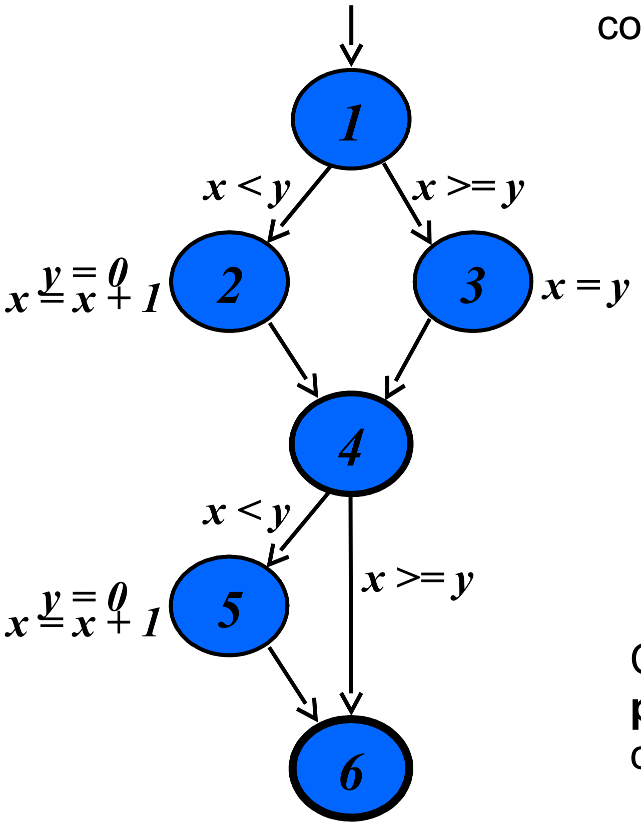
Path Coverage

Path coverage adequacy criterion: each path must be executed at least once

Coverage: $\dfrac{\textit{Number of executed paths}}{\textit{Number of all paths}}$

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}

if (x < y)
{
   y = 0;
   x = x + 1;
}
```



How many **paths** through this code are there? Need one test case for each to get path coverage

To get **statement** and **branch** coverage, we only need two test cases:
1 2 4 5 6 and 1 3 4 6

Path coverage needs two more:
1 2 4 5 6
1 3 4 6
1 2 4 6
1 3 4 5 6

Creating and executing tests for **all possible paths** results (**subsumes**) in 100% statement coverage and 100% branch coverage.

In general: Number of paths is **exponential** in the number of conditional branches. Therefore, path coverage cost may be very **expensive!**

OSU
Oregon State
UNIVERSITY

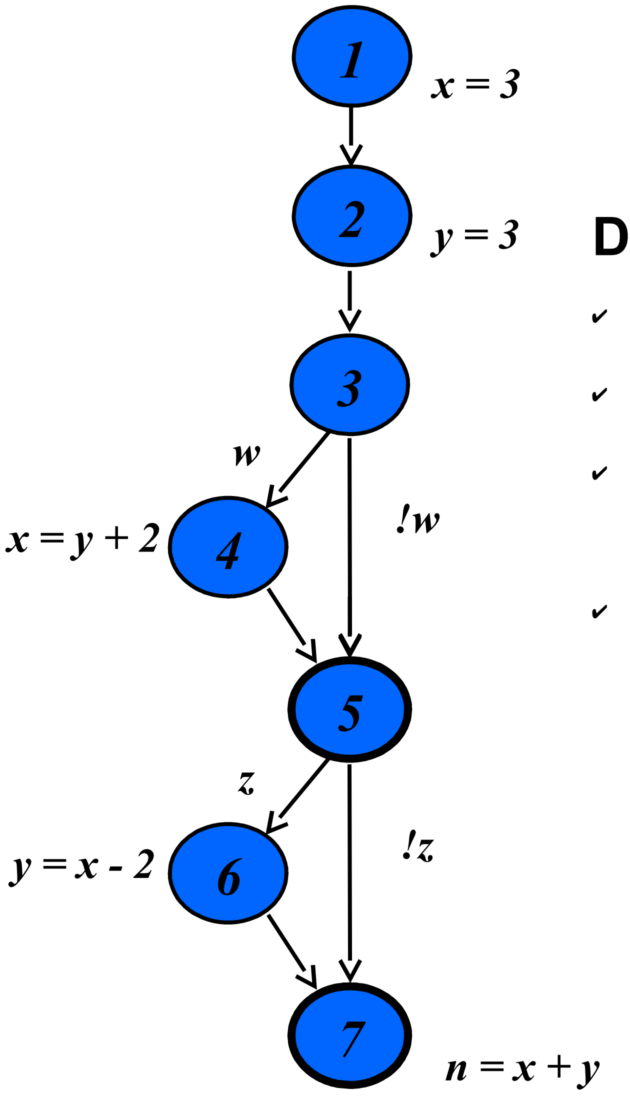
Data Flow (Def-Use) Coverage

```
x = 3;
y = 3;

if (w) {
   x = y + 2;
}

if (z) {
  y = x – 2;
}

n = x + y
```



During the life time of a **variable**, it can be defined and used.

Definition

- ✓ Variable declaration
- ✓ Variable initialization
- ✓ Variable assignment - left hand side of an expression
- ✓ Values received by a parameter

Use

- ✓ Expressions
- ✓ Parameter passing
- ✓ Conditional statements
- ✓ Returns

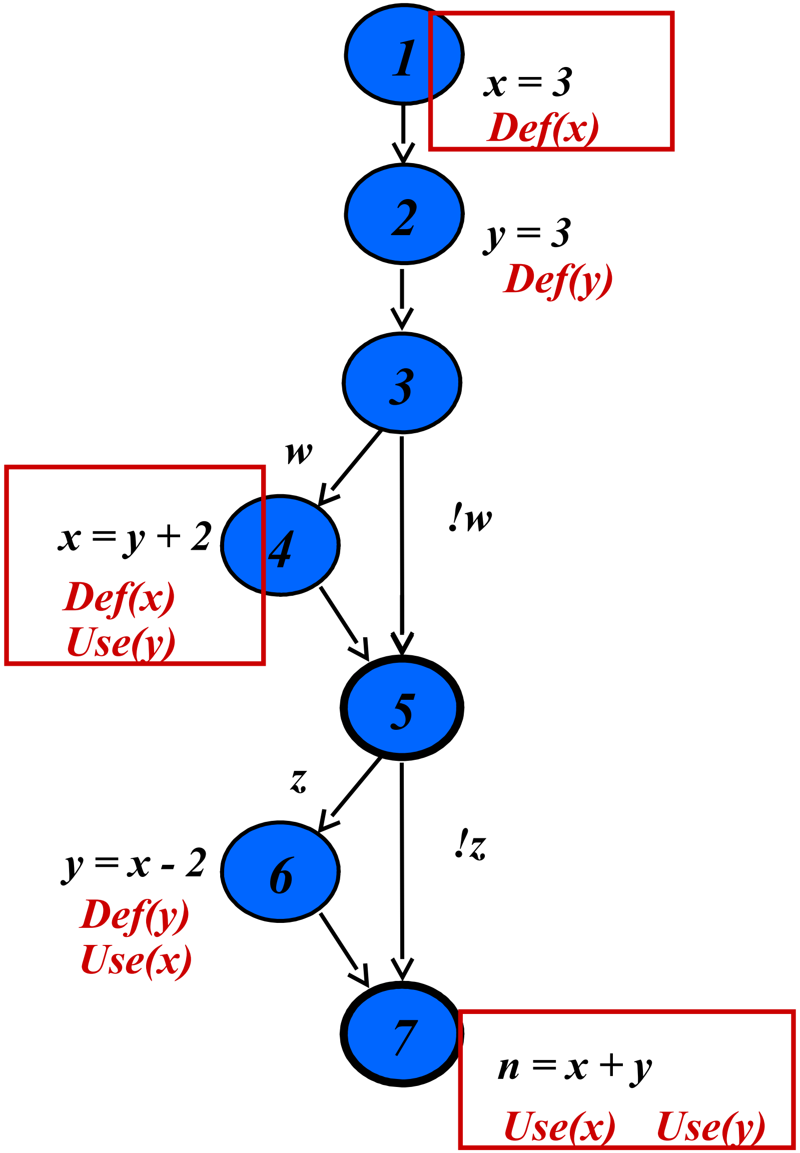
Data Flow (Def-Use) Coverage

```
x = 3;
y = 3;

if (w) {
   x = y + 2;
}

if (z) {
  y = x – 2;
}

n = x + y
```



1- Annotate program with locations where variables are defined and used (very basic static analysis)

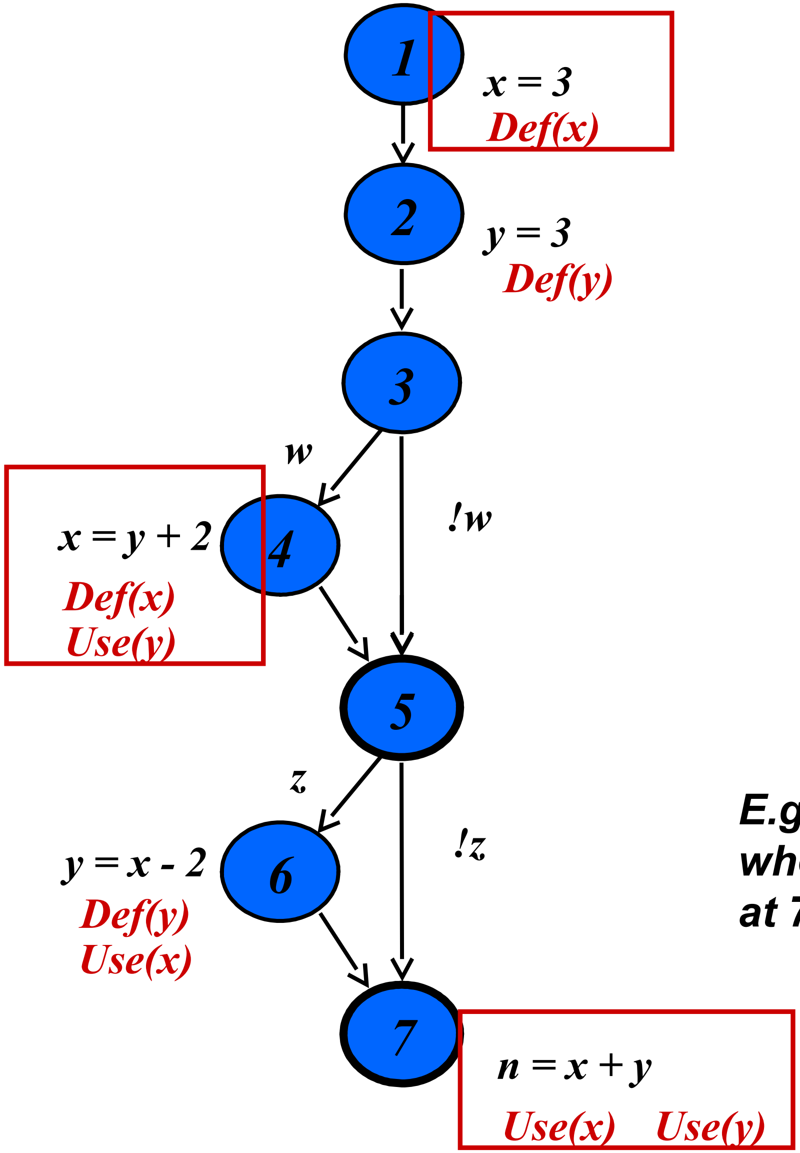
Data Flow (Def-Use) Coverage

```
x = 3;
y = 3;

if (w) {
   x = y + 2;
}

if (z) {
  y = x – 2;
}

n = x + y
```



1- Annotate program with locations where variables are defined and used (very basic static analysis)

2- Def-use pair coverage requires executing all possible pairs of nodes where a variable is first defined and then used, without any intervening re-definitions

E.g., this path covers the pair where x is defined at 1 and used at 7: 1 2 3 5 6 7

But this path does NOT: 1 2 3 4 5 6 7

Oregon State UNIVERSITY

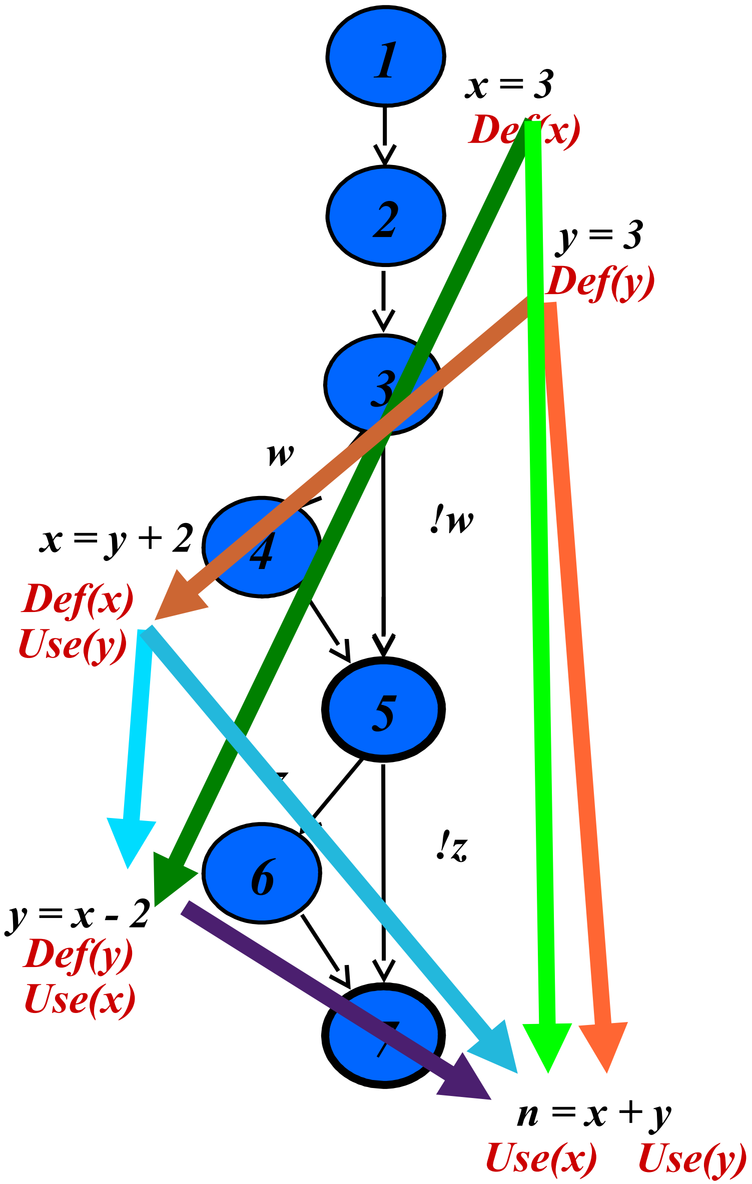
Data Flow (Def-Use) Coverage

```
x = 3;
y = 3;

if (w) {
   x = y + 2;
}

if (z) {
  y = x – 2;
}

n = x + y
```



May be many pairs,
some not actually executable

Condition (logic)Coverage

For each clause in a predicate:

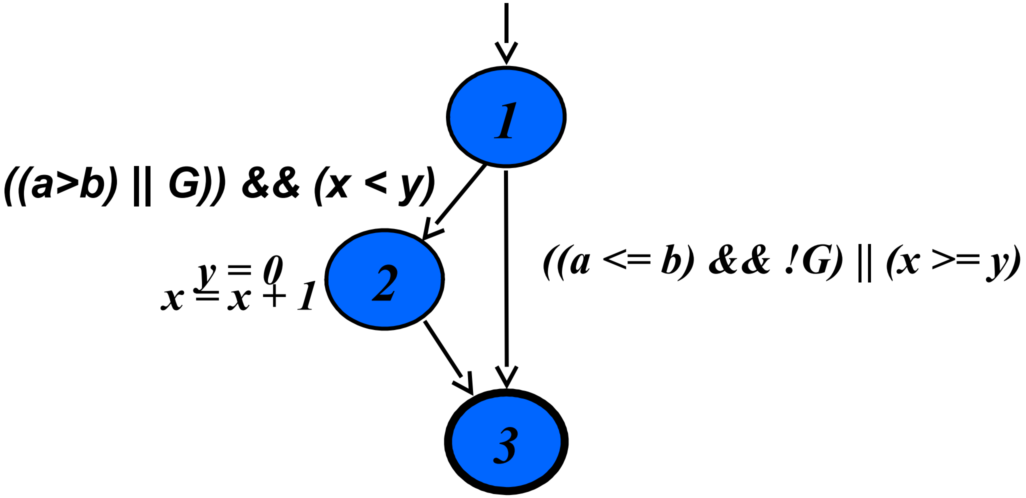
- Evaluate to true
- Evaluate to false

What if, instead of simple conditions:

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```

we have much more complicated:

```
if (((a>b) || G)) && (x < y))
{
  y = 0;
  x = x + 1;
}
```



Now, branch coverage will guarantee that we cover all the edges, but does not guarantee we will do so for all the different logical reasons

We want to test the logic of the guard of the if statement

Condition Coverage

With these values for G and (x<y), (a>b) determines the value of the predicate

With these values for (a>b) and (x<y), G determines the value of the predicate

With these values for (a>b) and G, (x<y) determines the value of the predicate

((a > b) or G) and (x < y)

1	T	F	T	T	
2	F	F	T	F	
3	F	T	T	T	*duplicate*
4	F	F	T	F	
5	T	T	T	T	
6	T	T	F	F	

Coverage and Subsumption

- Sometimes one coverage approach *subsumes* another
  - If you achieve 100% coverage of criteria A, you are guaranteed to satisfy B as well
    - For example, consider node and edge coverage (statement and coverage)
      - (there's a subtlety here, actually – can you spot it?)
- What does this mean?
  - Unfortunately, not a great deal
  - If test suite X satisfies "stronger" criteria A and test suite Y satisfies "weaker" criteria B
    - **Y may still reveal bugs that X does not!**

  - *It means we should take coverage with a grain of salt, for one thing. It means we should NOT just take coverage numbers as a magic indication how good our test suite is*

Test Coverage

- Test coverage always tells you where you haven’t tested
- What does code that does not get covered mean?

1.Infeasible code

```
// we are testing a balance tree data structure!
 if(leftsubTree.height() != rightSubTree.height())
        return false;
```

2. Code Not Worth Covering

```
// we are testing a function that calls an abort function which
simply aborts the execution of the program
    if(flag==true)
         abort();
```

3. Test Suite is inadequate

We can decide to ship the code without having 100% of coverage.

Testing “for” Coverage

- Two purposes: to know what we have & haven’t tested, and to know when we can “safely” stop testing
- Never seek to improve coverage *just for the sake of increasing coverage*
  - Well, unless it’s a command from-on-high
- Coverage is not the goal
- Finding failures that expose faults is the goal
  - Developers write test only to satisfy coverage
  - No amount of coverage will prove that the program cannot fail
- Coverage measures what is executed, not what is checked.

What’s So Good About Coverage?

- Consider a fault that causes failure every time the code is executed
- Don’t execute the code: cannot possibly find the fault!

```
int findLast (int a[], int n, int
   x) {
   // Returns index of last element

   // in a equal to x, or -1 if no
   // such.  n is length of a

   int i;
   for (i = n-1; i >= 0; i--) {
     if (a[i] = x)
       return i;
   }
   return 0;
}
```

Using *Cobertura* tool to Collect Coverage

- ***Cobertura*** is a free Java tool for collecting and anlyzying coverage.
- To integrate ***Cobertura*** report into the Maven, you need to add the following dependency to the **pom.xml** file.

```
<dependencies>
  ….
<dependency>
    <groupId>net.sourceforge.cobertura</groupId>
    <artifactId>cobertura</artifactId>
    <version>2.1.1</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.codehaus.mojo/findbugs-maven-plugin -->
<dependency>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>findbugs-maven-plugin</artifactId>
    <version>3.0.4</version>
</dependency>
</dependencies>
```

- **To generate code coverage report**
  - **mvn cobertura:cobertura**
  - Maven will generate the HTML code coverage report at ./target/site/cobertura/index.html.

Oregon State
UNIVERSITY

Maven + code coverage example

- **Note**: **we always run Maven commands in the directory that contains the pol.xml file**
- You need to compile your project
  - **mvn compile**
    - Note: if you run this for the first time, it might take a while to finish!
- Build the Project
  - **mvn package**
  - Note: if you want to run the main file
    - java -cp target/Dominion-1.0-SNAPSHOT.jar org.cs362.dominion.YourMainFileName
    - Java -cp ./target/classes/ org.cs362.dominion.YourMainFileName
- **Run your JUnit test cases**
  - **mvn test**
- **To import the project to Eclipse**
  - **mvn eclipse:eclipse**
    - **Then open eclipse and select File, Import and General, Existing projects to workspace, go to the Dominion folder and press OK**
- **To generate code coverage report**
  - **mvn cobertura:cobertura**
  - Maven will generate the HTML code coverage report at ./Dominion/target/site/cobertura/index.html.

References:
Young, Michal, and Mauro Pezze. "Software Testing and Analysis: Process, Principles and Techniques." (2005).Chapters 9, 12
http://classes.engr.oregonstate.edu/eecs/summer2015/cs362-002/
http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/Web/Testing/PathTest.html
www.st.cs.uni-saarland.de/edu