COMP5111 – Fundamentals of Software Testing and Analysis

Code Coverage and Instrumentation

Shing-Chi Cheung

Computer Science & Engineering

HKUST

Slides adapted from www.introsoftwaretesting.com by Paul Ammann & Jeff Offutt

Overview

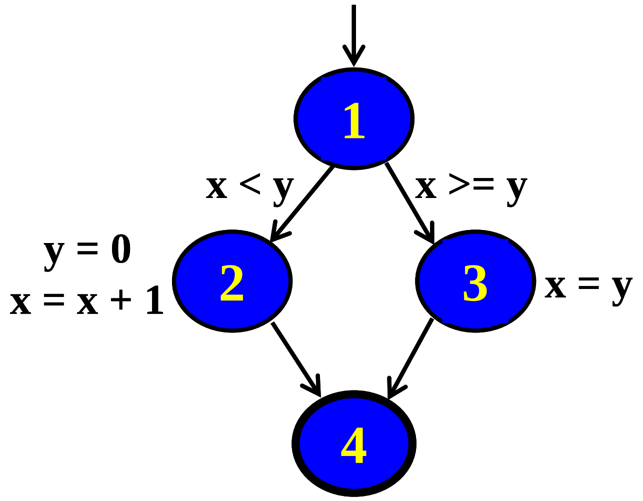
- The most usual application of graph criteria is to program source
- Graph : Usually the control flow graph (CFG)
- Node coverage : execute every statement
- Edge coverage : execute every branch
- Loops : looping structures such as for loops, while loops, etc.
- Data flow coverage : augment the CFG
    - defs are statements that assign values to variables
    - uses are statements that use variables

Control Flow Graphs

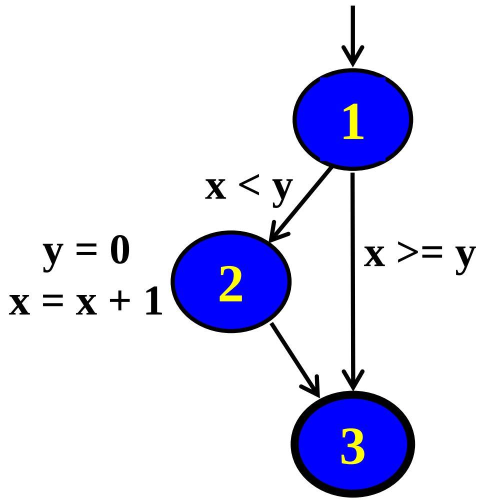
- A CFG models all executions of a method by describing control structures
- Nodes : statements or sequences of statements (basic blocks)
- Edges : transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses

CFG : The if Statement

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

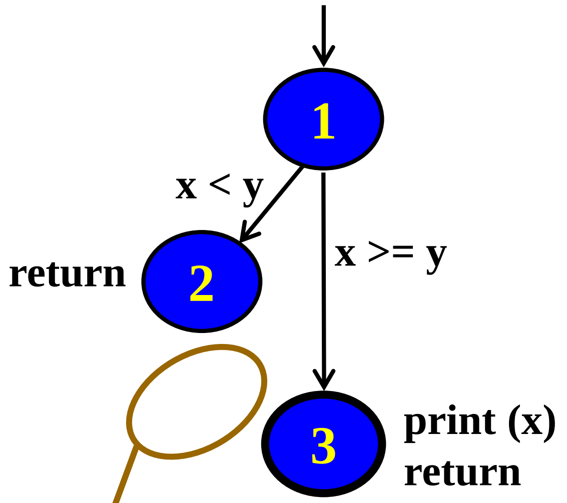


```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG : The if-Return Statement

```
if (x < y)
{
   return;
}
print (x);
return;
```



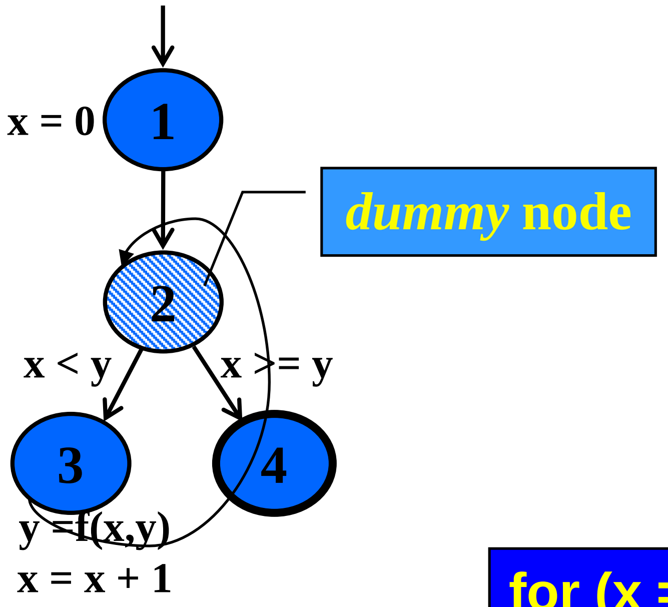
NO edge from node 2 to 3.
The return nodes must be distinct.

Loops

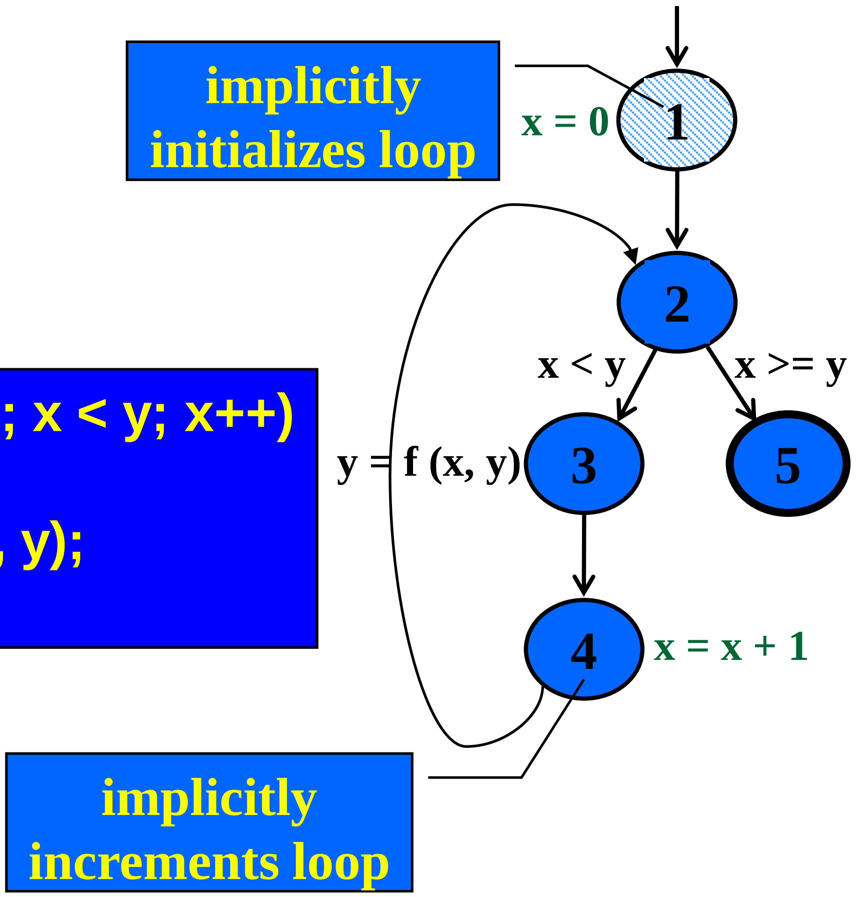
- Loops require “extra” nodes to be added
- Nodes that do not represent statements or basic blocks

CFG : while and for Loops

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
```

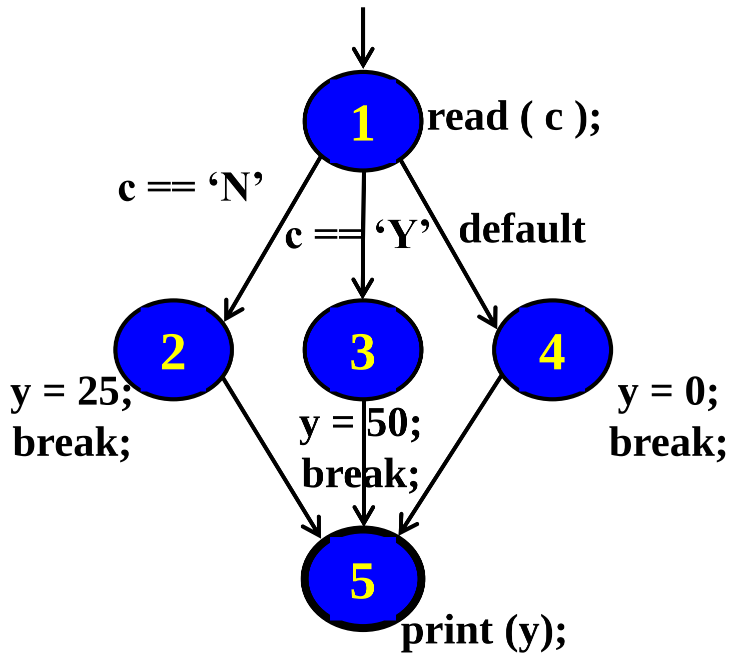


```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```



CFG : The case (switch) Structure

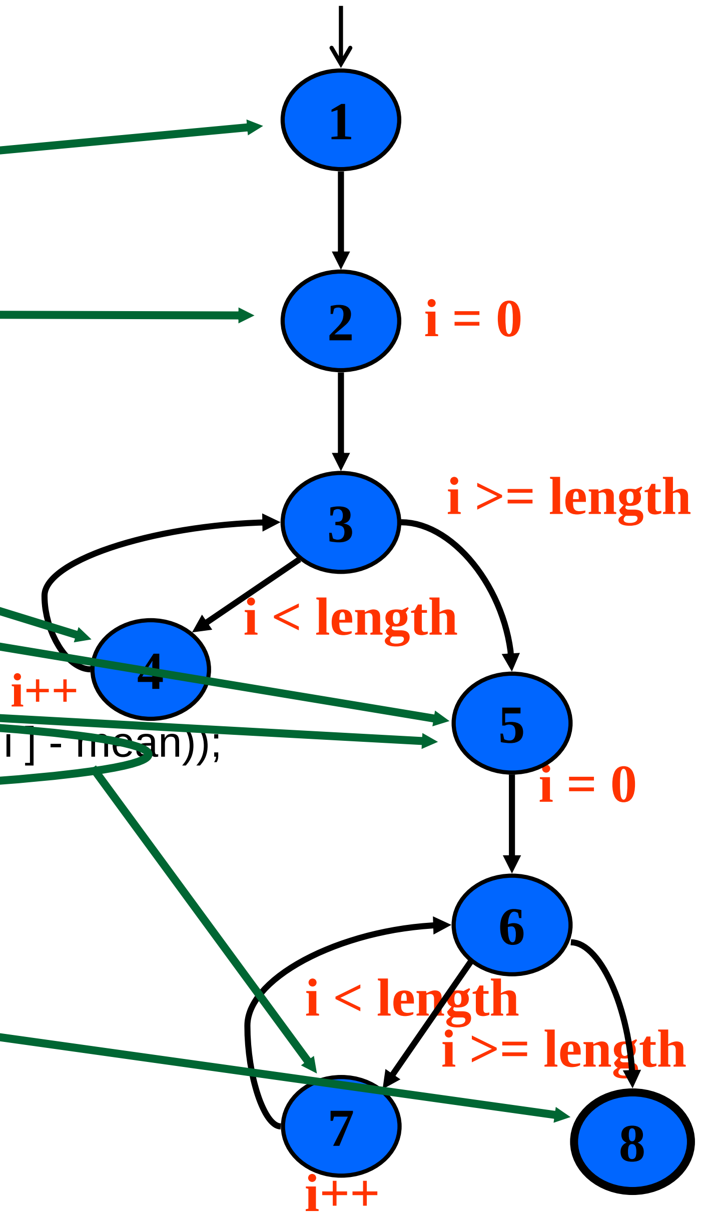
```
read ( c) ;
switch ( c )
{
   case ‘N’:
      y = 25;
      break;
   case ‘Y’:
      y = 50;
      break;
   default:
      y = 0;
      break;
}
print (y);
```



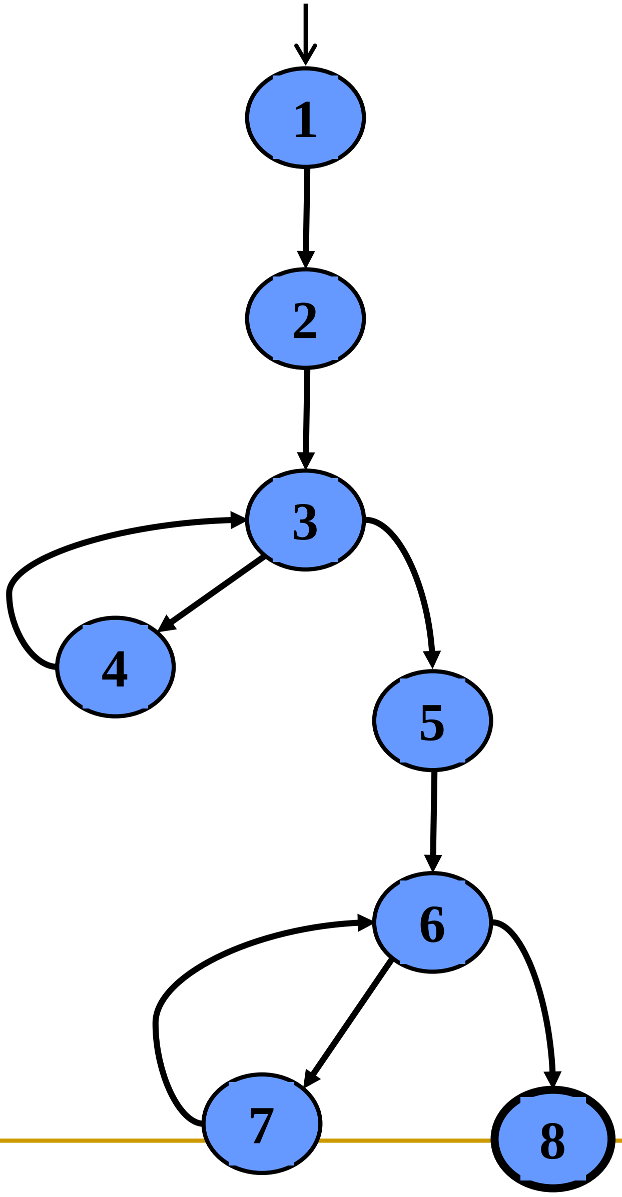
Example Control Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                   " + length);
    System.out.println ("mean:                     " + mean);
    System.out.println ("median:                   " + med);
    System.out.println ("variance:                 " + var);
    System.out.println ("standard deviation: " + sd);
}
```

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

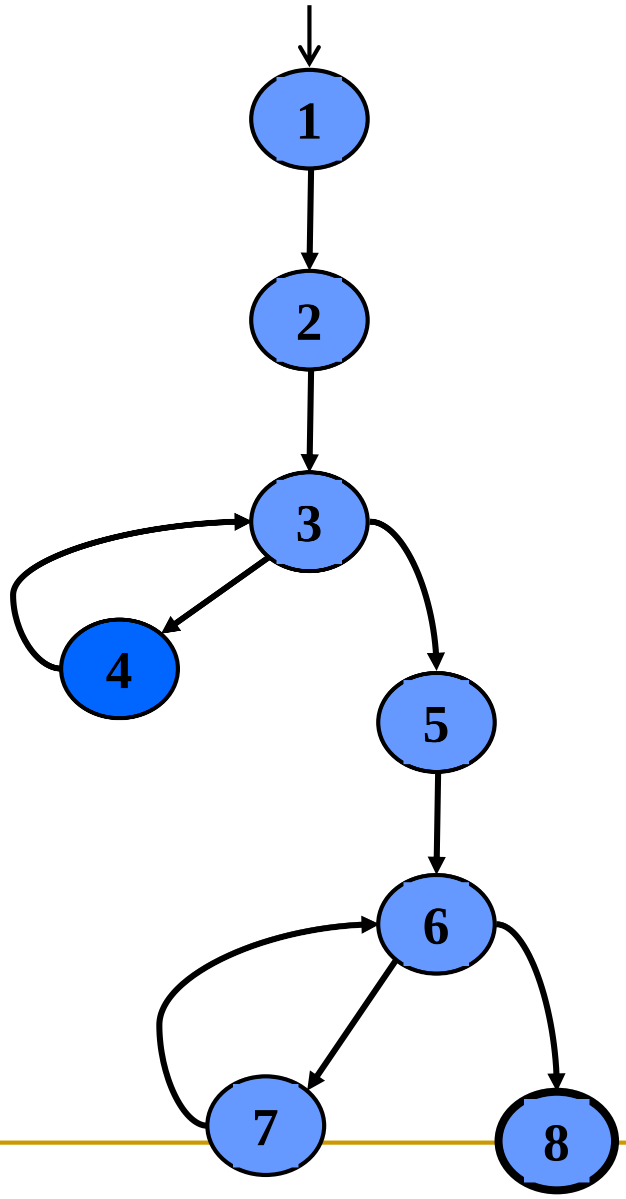


Control Flow TRs and Test Paths – EC



Edge Coverage	
TR	**Test Path**
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	

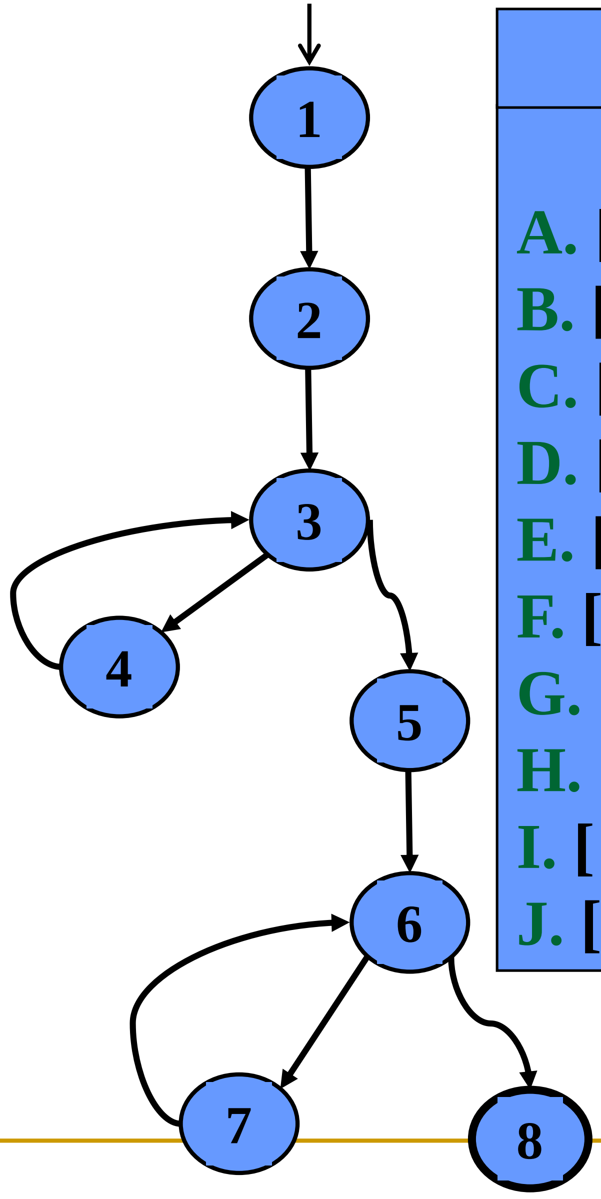
Control Flow TRs and Test Paths – EPC



Edge-Pair Coverage	
TR	**Test Paths**
A. [1, 2, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3, 4]	ii. [1, 2, 3, 5, 6, 8]
C. [2, 3, 5]	iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
D. [3, 4, 3]	
E. [3, 5, 6]	
F. [4, 3, 5]	
G. [5, 6, 7]	
H. [5, 6, 8]	
I. [6, 7, 6]	
J. [7, 6, 8]	
K. [4, 3, 4]	
L. [7, 6, 7]	

TP	TRs toured
~~i~~	~~A, B, D, E, F, G, I J~~
ii	A, C, E, H
iii	A, B, D, E, F, G, I, J, K, L

Control Flow TRs and Test Paths – PPC



Prime Path Coverage	
TR	**Test Paths**
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

TP	TRs toured	
~~i~~	~~A, D, E, F, G~~	
ii	A, B, C, D, E, F, G,	two
iii	A, F, H	one
iv	D, E, F, I	one
v	J	one

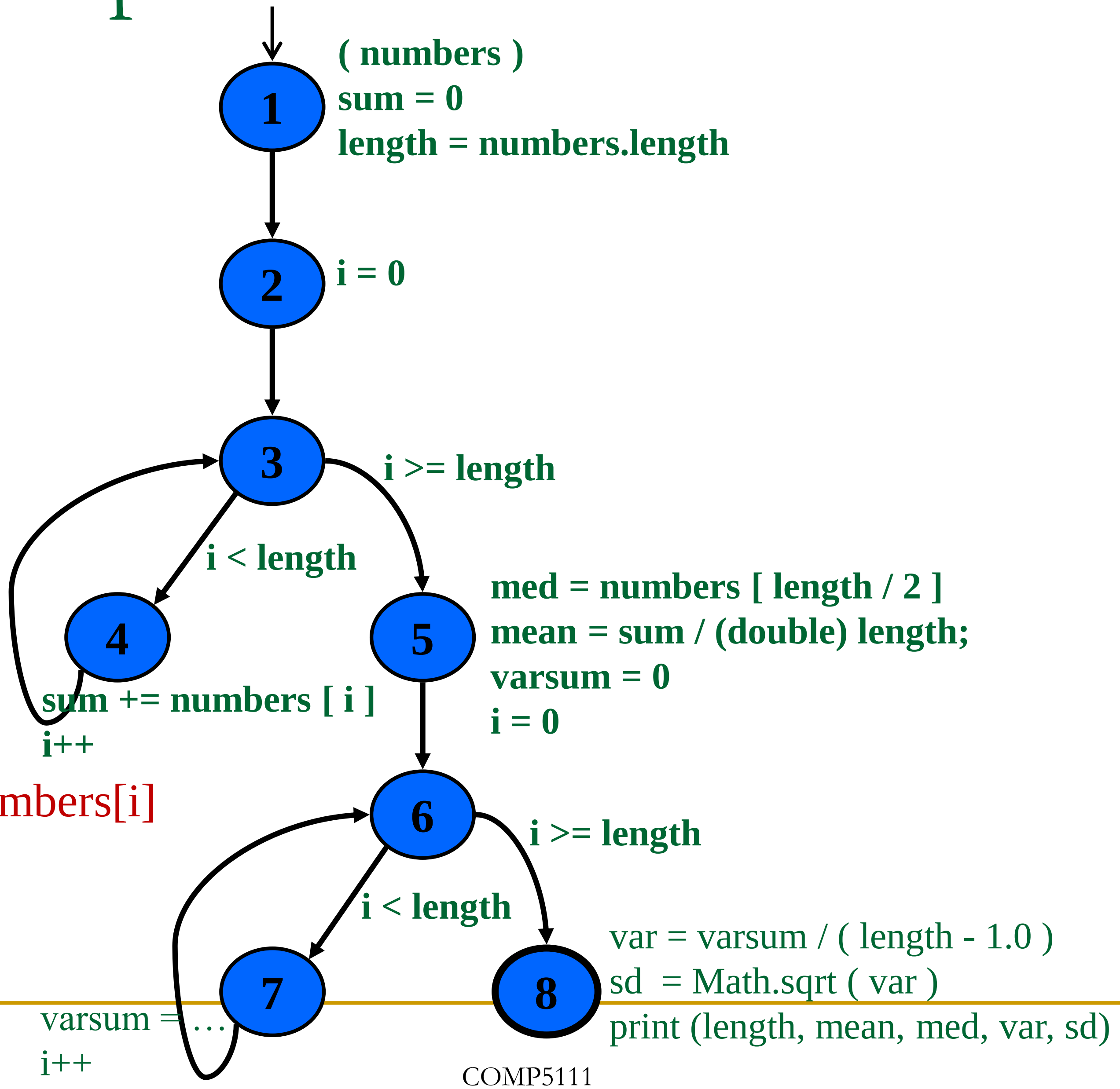
Data Flow Coverage for Source

- def : a location where a value is stored into memory
  - x appears on the left side of an assignment (x = 44;)
  - x is an actual parameter in a call and the method changes its value
  - x is a formal parameter of a method (implicit def when method starts)
  - x is an input to a program
- use : a location where variable's value is accessed
  - x appears on the right side of an assignment
  - x appears in a conditional test
  - x is an actual parameter to a method
  - x is an output of the program
  - x is an output of a method in a return statement
- If a def and a use appear on the same node, then it is only a DU-pair if the def occurs after the use and the node is in a loop

Example Data Flow – Stats

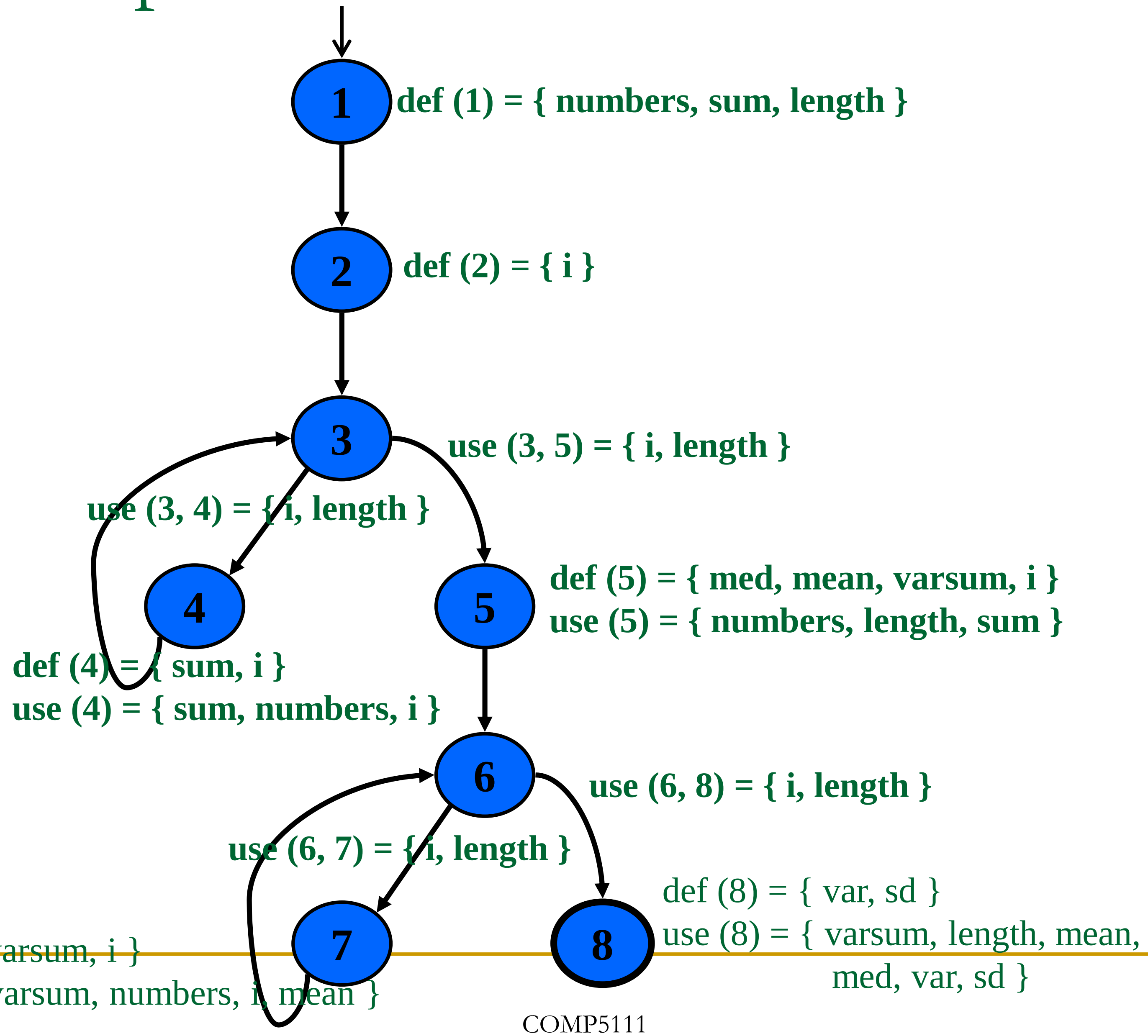
```java
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Data Flow Graph for Stats



tmp = sum + numbers[i]
sum = tmp

Data Flow Graph for Stats


1
def (1) = { numbers, sum, length }
2
def (2) = { i }
3
use (3, 5) = { i, length }
use (3, 4) = { i, length }
4
5
def (5) = { med, mean, varsum, i }
use (5) = { numbers, length, sum }
def (4) = { sum, i }
use (4) = { sum, numbers, i }
6
use (6, 8) = { i, length }
use (6, 7) = { i, length }
7
8
def (8) = { var, sd }
use (8) = { varsum, length, mean, med, var, sd }
def (7) = { varsum, i }
use (7) = { varsum, numbers, i, mean }

Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) ~~(2, 7) (2, (6,7)) (2, (6,8))~~ (4, 4) (4, (3,4)) (4, (3,5)) ~~(4, 7) (4, (6,7)) (4, (6,8))~~ (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

defs come <u>before</u> uses, do not count as DU pairs

defs <u>after</u> use in loop, these are valid DU pairs

No def-clear path … different scope for i

No path through graph from nodes 5 and 7 to 4 or 3

DU Paths for Stats

variable	DU Pairs	DU Paths
numbers	(1, 4) (1, 5) (1, 7)	[1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7]
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	[1, 2, 3, 5] [1, 2, 3, 5, 6, 8] [1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7] [1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	*No path needed*
sd	(8, 8)	*No path needed*
sum	(1, 4) (1, 5) (4, 4) (4, 5)	[1, 2, 3, 4] [1, 2, 3, 5] [4, 3, 4] [4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7) (5, 8)	[5, 6, 7] [5, 6, 8]
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	[5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 8]
i	(2, 4) (2, (3,4)) (2, (3,5)) (4, 4) (4, (3,4)) (4, (3,5)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	[2, 3, 4] [2, 3, 4] [2, 3, 5] [4, 3, 4] [4, 3, 4] [4, 3, 5] [5, 6, 7] [5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 7] [7, 6, 8]

DU Paths for Stats – No Duplicates

There are 38 DU paths for Stats, but only 12 unique

✦ [1, 2, 3, 4]	[4, 3, 4] ✸
★ [1, 2, 3, 5]	[4, 3, 5] ✦
★ [1, 2, 3, 5, 6, 7]	[5, 6, 7] ✦
★ [1, 2, 3, 5, 6, 8]	[5, 6, 8] ★
✦ [2, 3, 4]	[7, 6, 7] ✸
★ [2, 3, 5]	[7, 6, 8] ✦

★ **5 expect a loop not to be “entered”**

✦ **5 require at least one iteration of a loop**

✸ **2 require at least two iterations of a loop**

Test Cases and Test Paths

Test Case : numbers = (44) ; length = 1
Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
Additional DU Paths covered (no sidetrips)
[1, 2, 3, 4]   [2, 3, 4]   [4, 3, 5]   [5, 6, 7]   [7, 6, 8]
The five stars ✦ that require at least one iteration of a loop

Test Case : numbers = (2, 10, 15) ; length = 3
Test Path : [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]
DU Paths covered (no sidetrips)
[4, 3, 4]   [7, 6, 7]
The two stars ✪ that require at least two iterations of a loop

Other DU paths ☆ require arrays with length 0 to skip loops
But the method fails with divide by zero on the statement …

```
mean = sum / (double) length;
```

A fault is found

Instrumentation for Test Coverage

Tools Instrumentation

- Coverage analysis is measured with instrumentation
- Instrument : One or more statements inserted into the program to monitor some aspect of the program
  - Must not affect the behavior
  - May affect timing
  - Source level or object code level

Mark: "if body is reached"

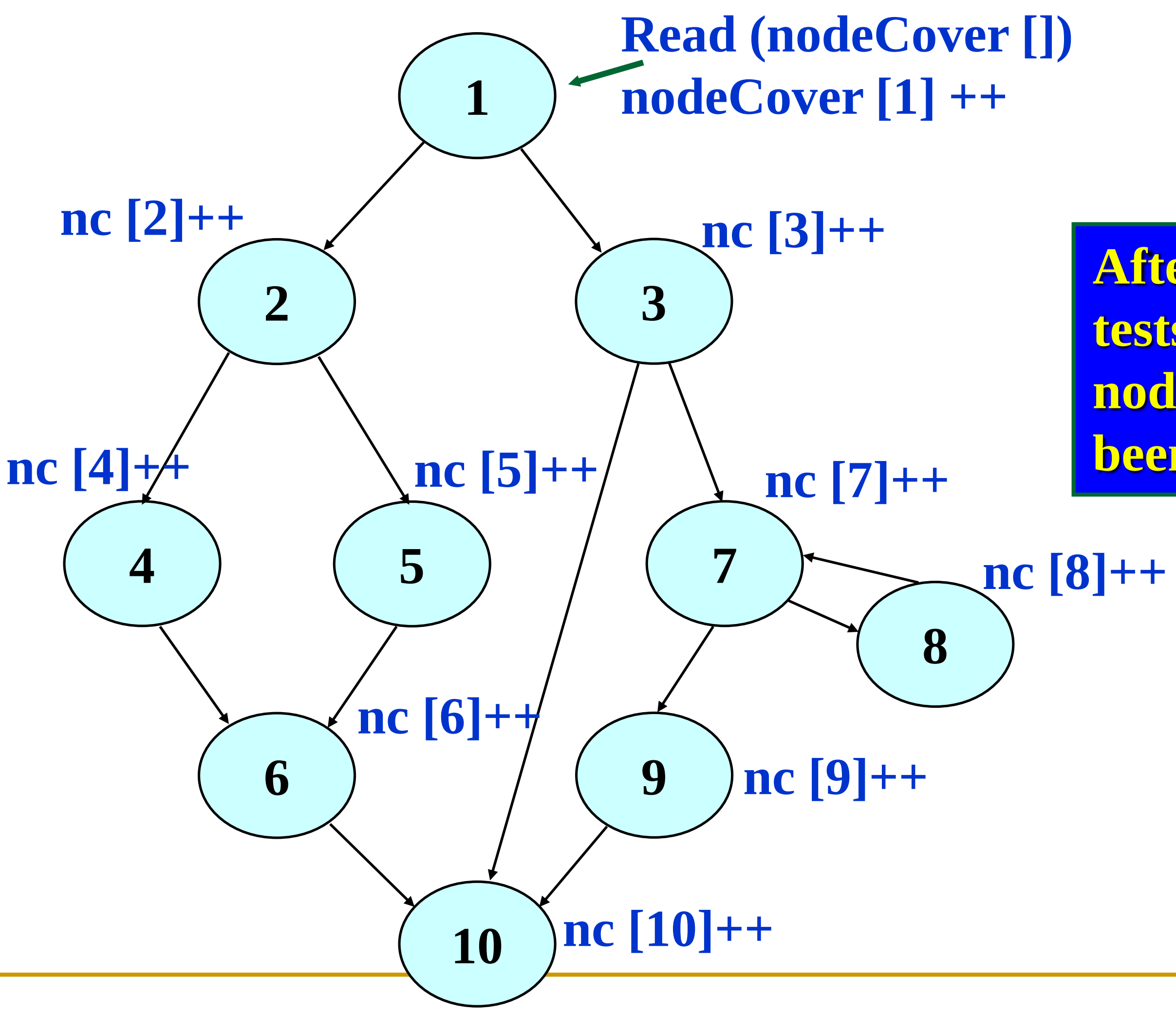
```
public int min (int A, B)
{
    int m = A;
    if (A > B)
    {
        m = B;
    }
    return (m);
}
```

Instrumenting for Statement Coverage

1. Each node is given a unique id #
   - Node # or statement #
2. Create an array indexed by id #s – nodeCover [ ]
3. Insert an <u>*instrument*</u> at each node
   - nodeCover [i] ++;
4. Save nodeCover [ ] after each execution
   - Must <u>accumulate</u> results across multiple test cases

Statement Coverage Example

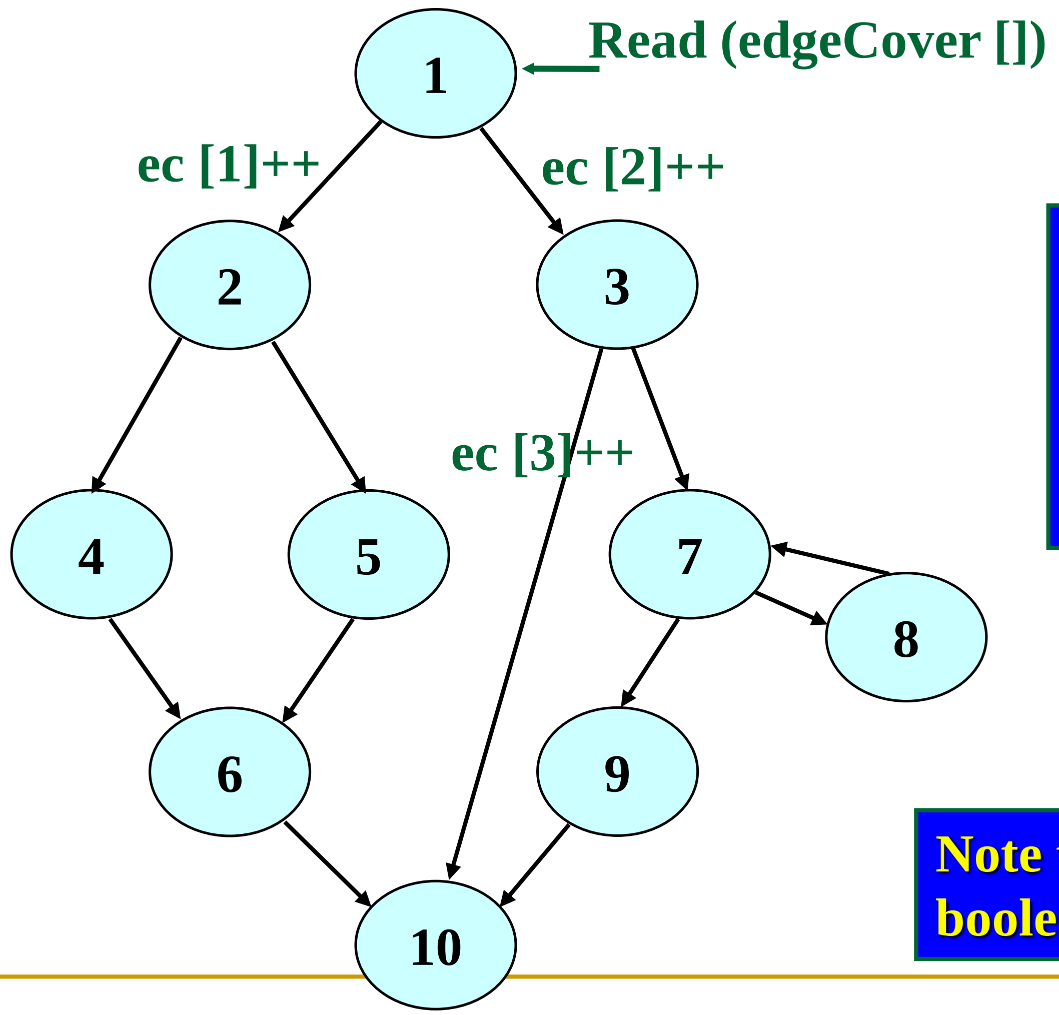
int nodeCover[] = {0,0,0,0,0,0,0,0,0,0}



After running a sequence of tests, any node for which nodeCover[node]==0 has <u>not</u> been covered.

Edge Coverage Instrumentation

int edgeCover[] = {0,0,0,0,0,0,0,0,0,0,0,0,0}

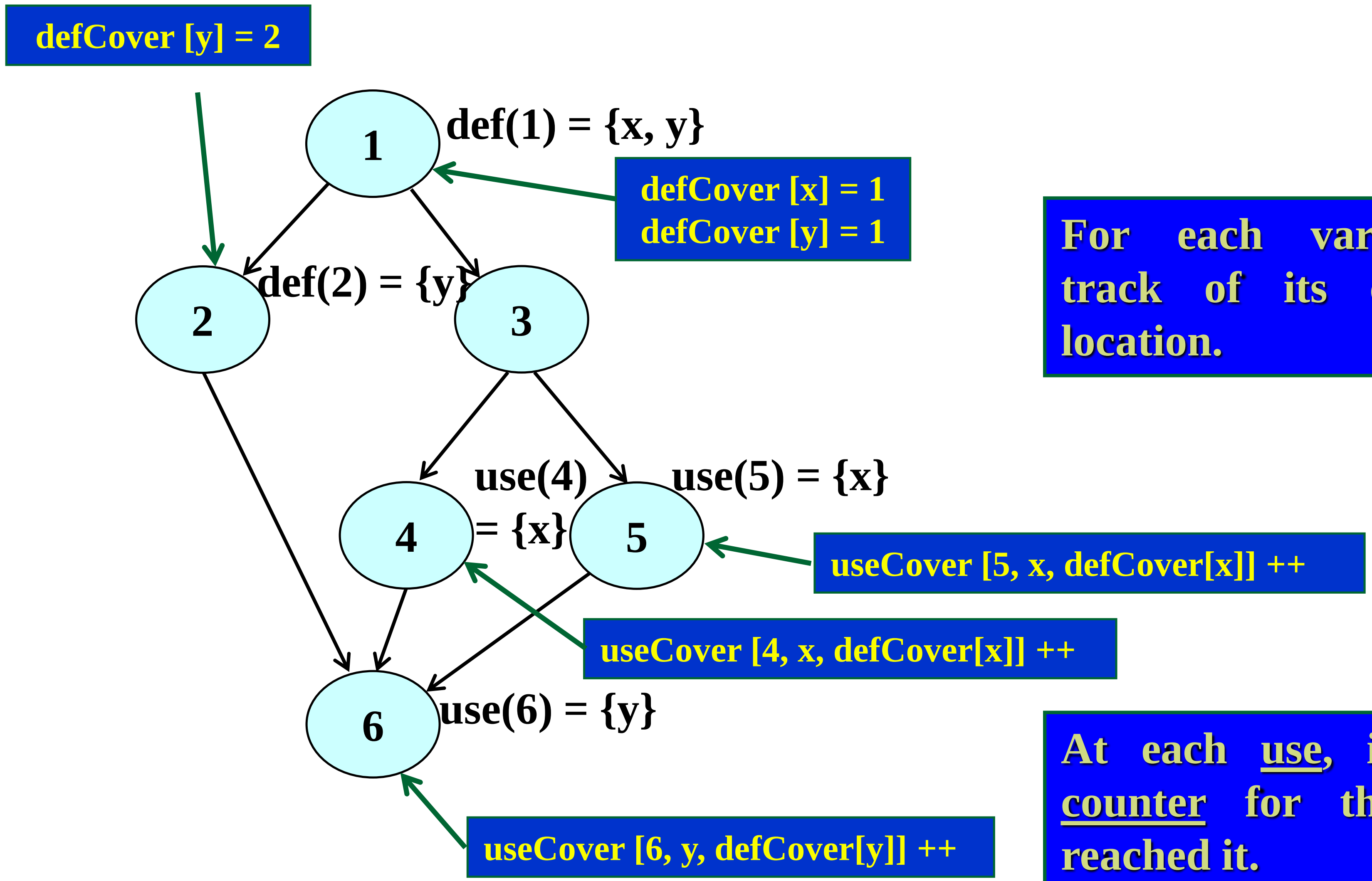


For each edge e, put edgeCover[e]++ on the edge.

If edgeCover[e] == 0, e has not been covered.

Note that the arrays could be boolean

All-Uses Coverage Instrumentation



For each variable, keep track of its current def location.

At each use, increment a counter for the def that reached it.

Instrumentation Summary

- Instrumentation can be added in multiple copies of the program
  - Source code
  - Java byte code (or other intermediate code)
  - Executable
- Instrumentation must not change or delete functionality
  - Only add new functionality
- Instrumentation may affect timing behavior
- Requires the program to be parsed
  - Once parsed, inserting instruments is straightforward