EECS 4313

Software Engineering Testing

Topic 07:

Path Testing and Test Coverage

Zhen Ming (Jack) Jiang

Relevant Readings

- [Jorgensen] chapter 8
- [Ammann & Offutt] chapter 7

Structural Testing (White-box Testing)

- Also known as glass/white/open box testing
- A software testing technique whereby explicit knowledge of the internal workings of the item being tested are used to select the test data
- Black-box testing uses program specification
- White-box testing is based on specific knowledge of the source code to define the test cases and to examine outputs.

White-box Testing

- White-box testing methods are very amenable to:
  - Rigorous definitions
    - Control flow, data flow, coverage criteria
  - Mathematical analysis
    - Graphs, path analysis
  - Precise measurement
    - Metrics, coverage analysis

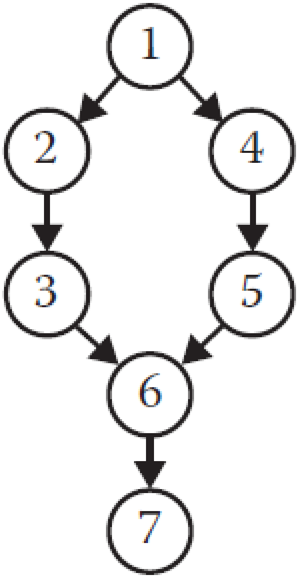
Program Graph - Definition

- Given a program written in an imperative programming language, its **program graph** is a directed graph in which nodes are statement fragments, and edges represent flow of control
- A complete statement is also considered a statement fragment

Program graphs for four structured programming constructs

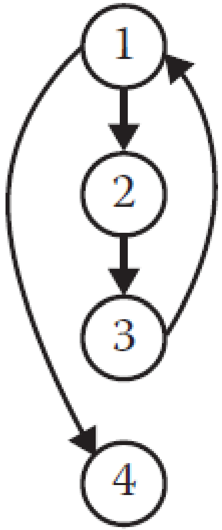
If–Then–Else

```
1  If <condition>
2     Then
3        <then statements>
4     Else
5        <else statements>
6  End If
7  <next statement>
```



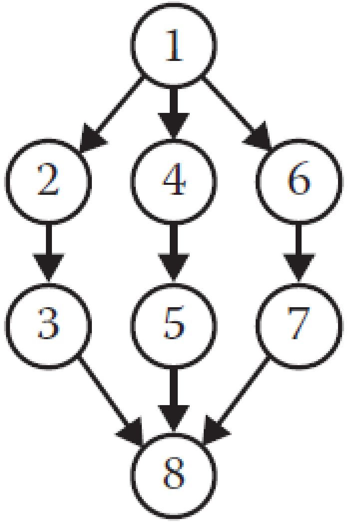
Pretest loop

```
1  While <condition>
2        <repeated body>
3  End While
4  <next statement>
```



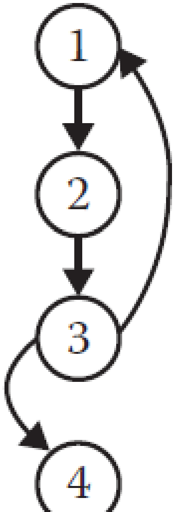
Case/Switch

```
1  Case n of 3
2     n=1:
3        <case 1  statements>
4     n=2:
5        <case 2  statements>
6     n=3:
7        <case 3  statements>
8  End Case
```



Posttest loop

```
1  Do
2        <repeated body>
3  Until <condition>
4  <next statement>
```



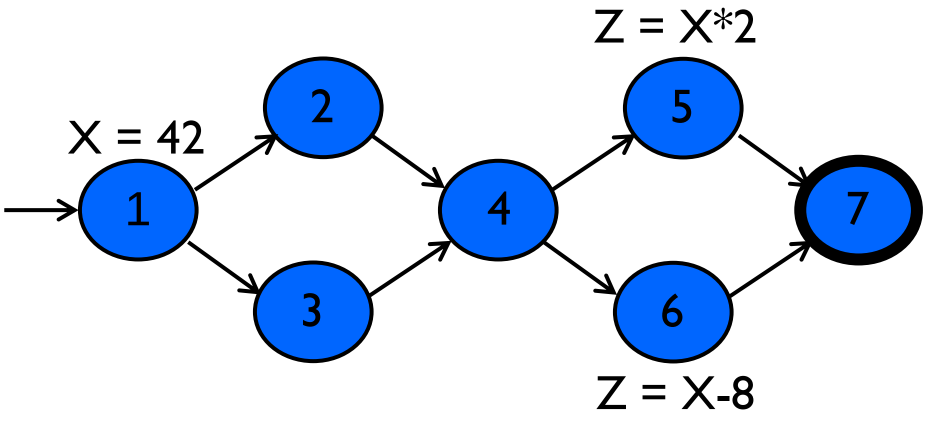
Control Flow Graphs (CFGs)

- 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
- Rules for translating statements into graphs …

Def and Use

- Definition (def):
  - A location where a value for a variable is stored into memory
- Use:
  - A location where a variable's value is accessed

Must have an entry point and (at least one) exit node



Defs: def (1) = { X }

def (5) = { Z }

def (6) = { Z }

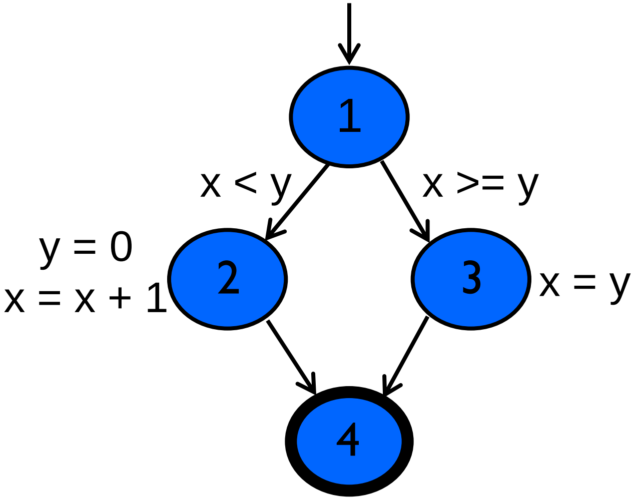
Uses: use (5) = { X }

use (6) = { X }

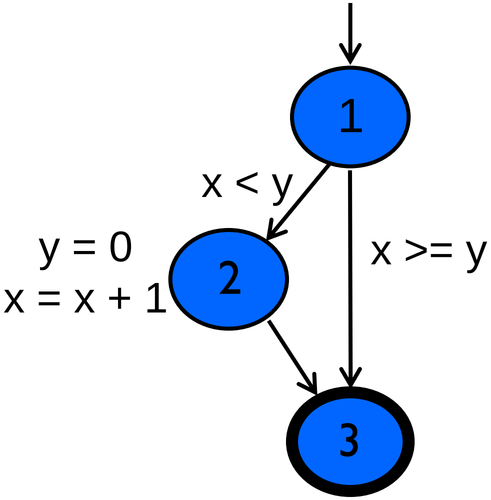
The values given in defs should reach at least one, some, or all possible uses

CFG : The if Statement

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

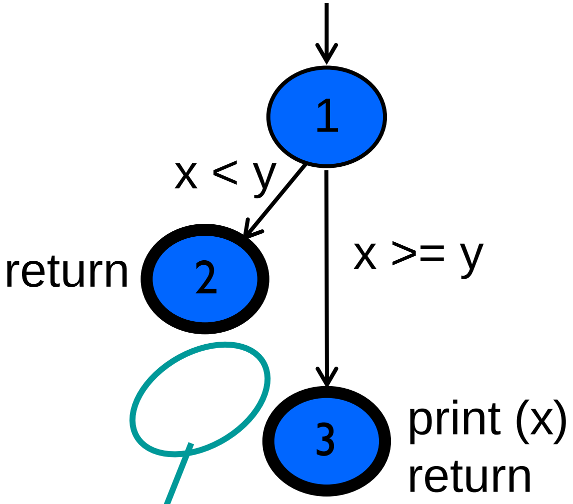


```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG : The if-Return Statement

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



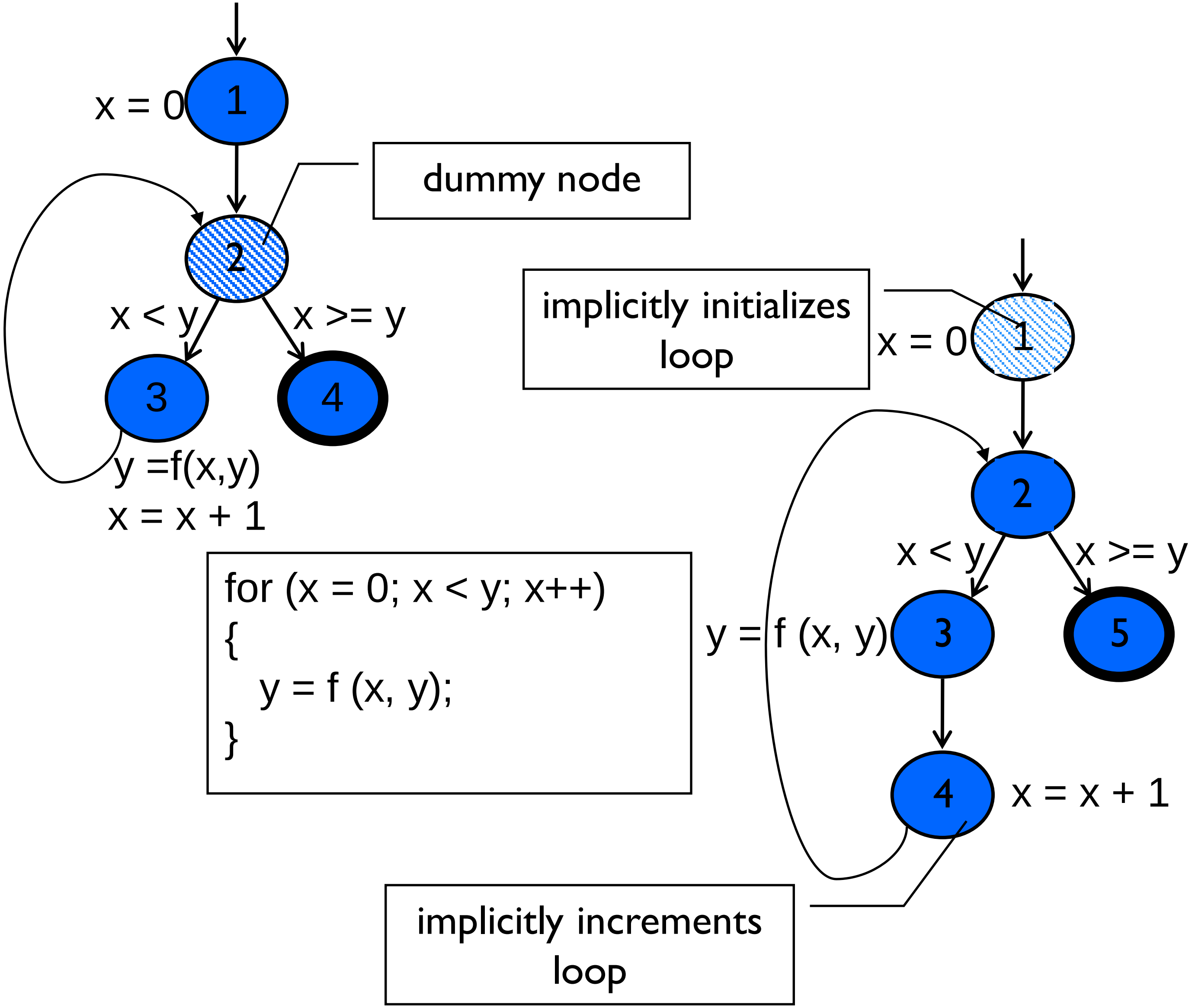
No edge from node 2 to 3.
The return nodes must be distinct.

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;
}
```



dummy node

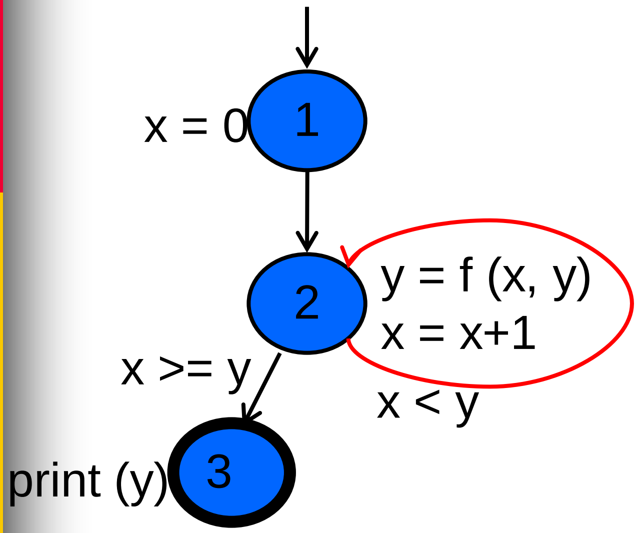
implicitly initializes loop

implicitly increments loop

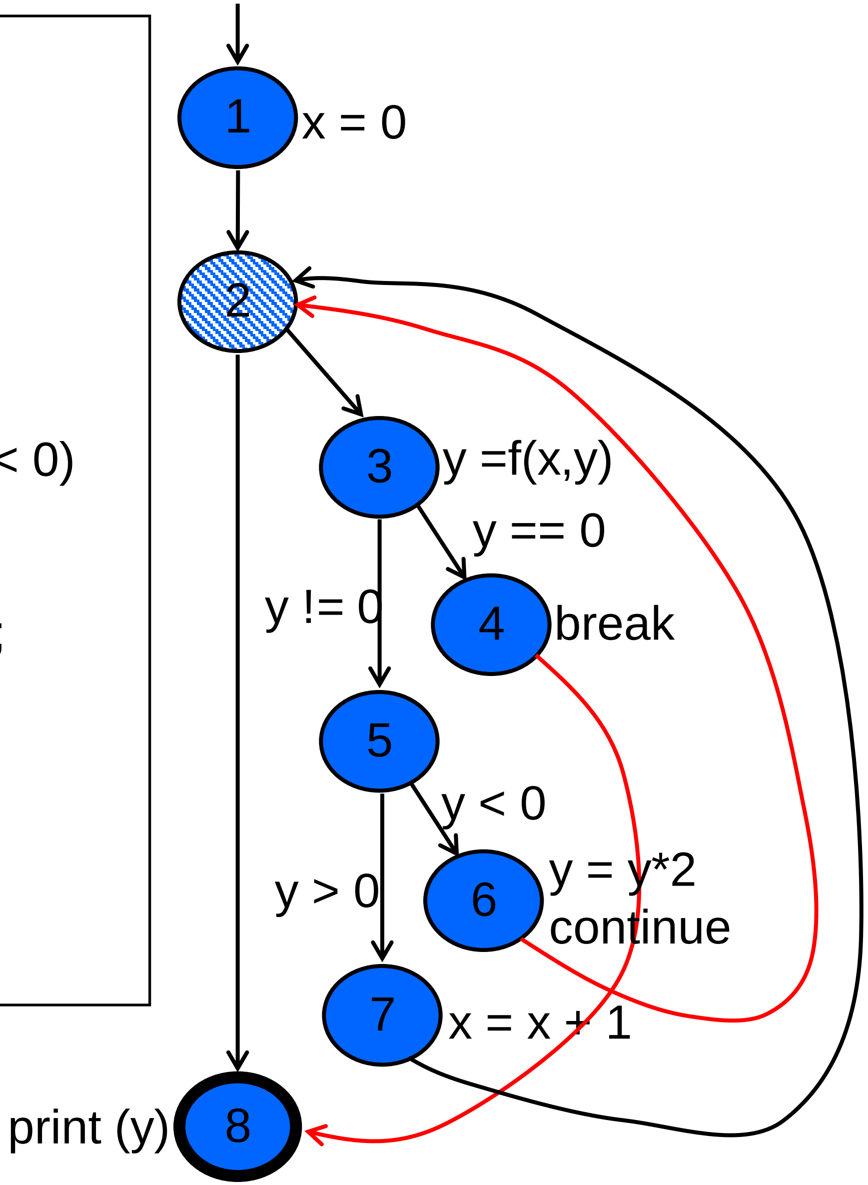
```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

CFG: do Loop, break and continue

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

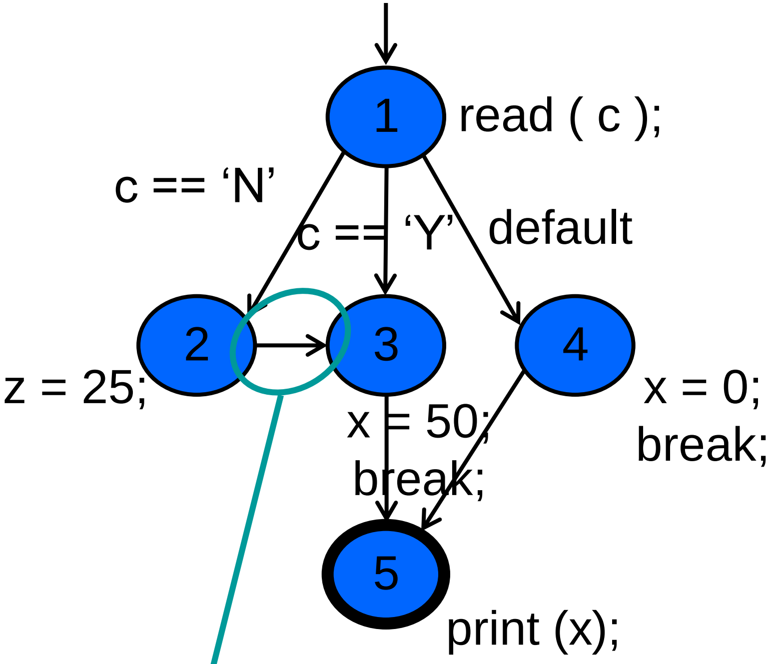


```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```



CFG: The case (switch) Structure

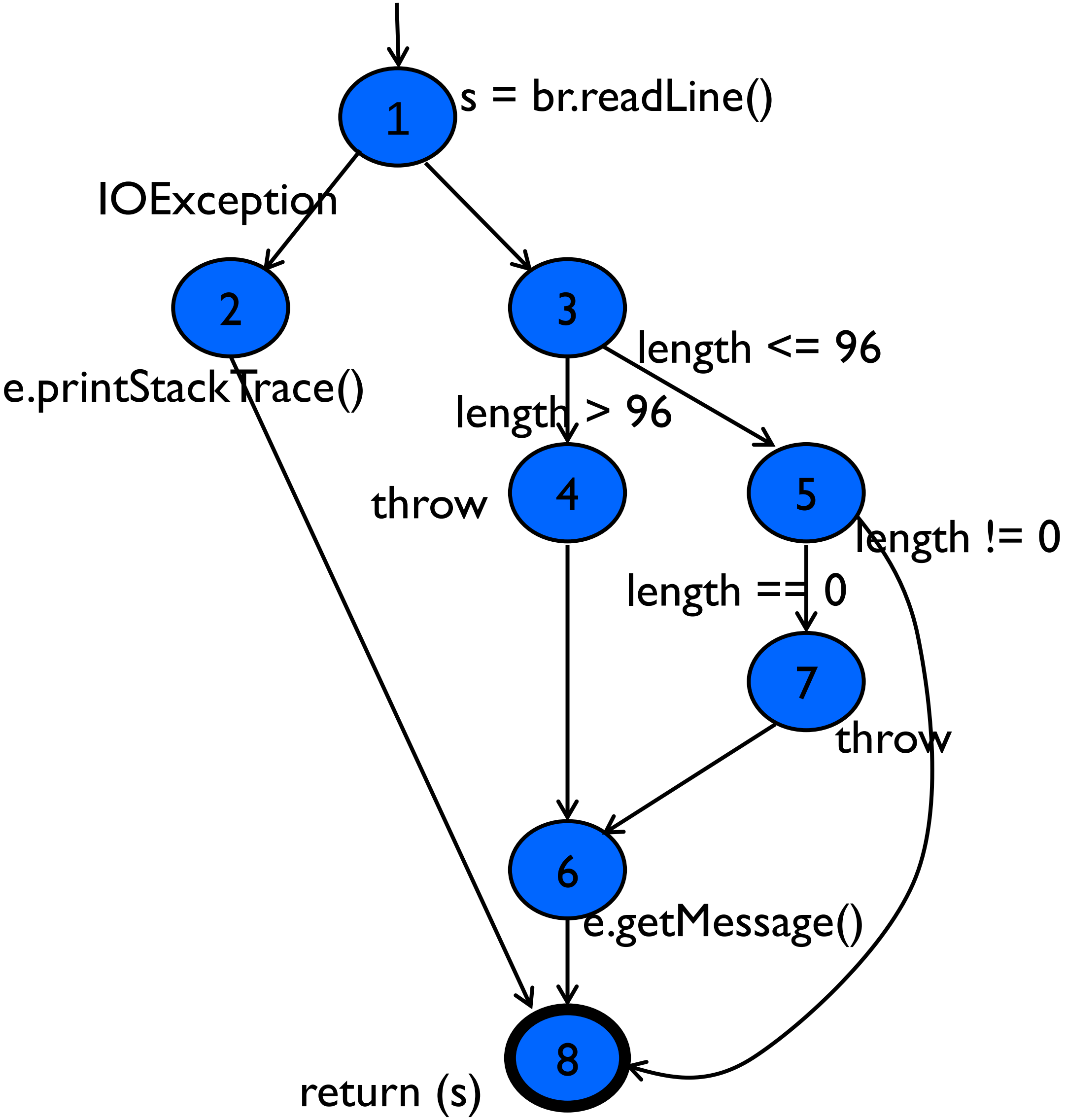
```
read ( c) ;
switch ( c )
{
   case ‘N’:
      z = 25;
   case ‘Y’:
      x = 50;
      break;
   default:
      x = 0;
      break;
}
print (x);
```



Cases without breaks fall through to the next case

CFG : Exceptions (try-catch)

```
try
{
   s = br.readLine();
   if (s.length() > 96)
      throw new Exception
         (“too long”);
   if (s.length() == 0)
      throw new Exception
         (“too short”);
} (catch IOException e) {
   e.printStackTrace();
} (catch Exception e) {
   e.getMessage();
}
return (s);
```



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);
}
```

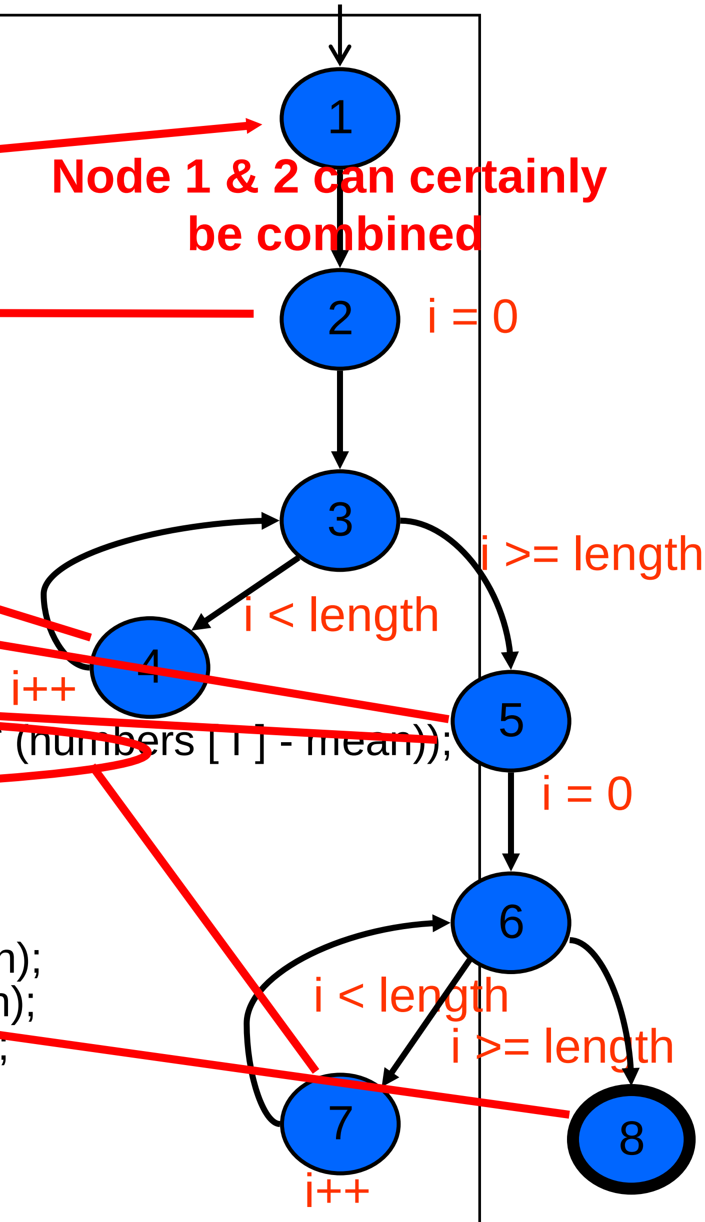
Control Flow Graph for Stats

```
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);
}
```



Node 1 & 2 can certainly be combined

Creating test cases using code coverage metrics

- In order to increase the coverage of a test suite, one needs to generate test cases that exercise certain statements or follow a specific path
  - Define test coverage goals in terms of test requirements
  - This results in test specifications and test cases
- This is not always easy to do …

Code Coverage

Code coverage models

- Statement Coverage
- Segment Coverage
- Branch Coverage
- Condition Coverage
- Branch & Condition Coverage
- Modified Condition/Decision Coverage

Statement coverage

- Achieved when all statements in a method have been executed at least once

- Take home exercises
  - How many test cases do we need to achieve statement coverage in our example?

Statement Coverage Measure

```
public void printSum(int a, int b) {
 int result = a + b;
 if (result > 0) {
     System.out.println(″red″, result);
 else if (result < 0)
     System.out.println(″blue″, result);
}
```

$$Statement\ Coverage = \frac{\#\ of\ executed\ statements}{total\ \#\ of\ statements}$$

<u>TC1</u>

- a = 3
- b = 9

Coverage = 5/7 = 71%

<u>TC2</u>

- a = -5
- b = -8

Coverage = 100%

Segment coverage

- **Segment coverage** counts segments rather than statements
- May produce drastically different numbers
  - Assume two segments P and Q
  - P has one statement, Q has nine
  - Exercising only one of the segments will give 10% or 90% statement coverage
  - Segment coverage will be 50% in both cases

Statement coverage in practice

- Statement coverage is most used in industry
- Typical coverage target is 80-90%
  - Why don’t we aim at 100%?

Statement coverage problems

- Predicate may be tested for only one value (misses many bugs)
- Loop bodies may only be iterated once
- Statement coverage can be achieved without branch coverage. Important cases may be missed

```
public void printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
      System.out.println(″red″, result);
  else if (result < 0)
      System.out.println(″blue″, result);
  [else do nothing]
}
```

TC1
- a = 3
- b = 9

TC2
- a = -5
- b = -8

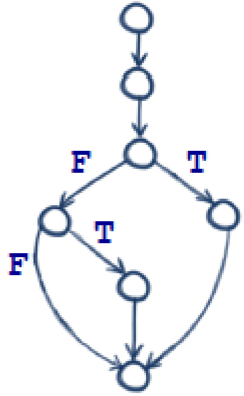
Never in here for the above two test cases!

Branch coverage

- Achieved when every branch from a node is executed at least once
- At least one true and one false evaluation for each predicate
- Can be achieved with D+1 paths in a control flow graph with D 2-way branching nodes and no loops
  - Even less if there are loops

Branch Coverage Measure

```
public void printSum(int a, int b) {
  int result = a + b;
  if (result > 0)
      System.out.println(“red”, result);
  else if (result < 0) {
      System.out.println(“blue”, result);
  [else do nothing]
}
```



$$\mathit{Branch\ Coverage} = \frac{\#\ \mathit{of\ executed\ branches}}{\mathit{total}\ \#\ \mathit{of\ branches}}$$

TC1
- a = 3
- b = 9

Coverage = 1/4 = 25%

TC2
- a = -5
- b = -8

Coverage = 3/4 = 75%

TC3
- a = -5
- b = 5

Coverage = 100%

Branch coverage problems

- Short-circuit evaluation means that many predicates might not be evaluated
- A compound predicate is treated as a single statement. If n clauses, 2^n combinations, but only 2 are tested
- Only a subset of all entry-exit paths is tested

```
public void printResults(int a, int b) {
  if ((a == 0) || (b > 0))
      System.out.println("red", b/a);
  else
      System.out.println("blue", b + 2);
  System.out.println("end");
}
```

<u>TC1:</u> (a = 5, b = 6)
<u>TC2:</u> (a = 5, b = -5)
<u>Branch Coverage = 100%</u>

Can we thoroughly test all the conditions?

Condition coverage

- Condition coverage reports the true or false outcome of each condition.
- Condition coverage measures the conditions **independently** of each other.

Condition Coverage Measure

```
public void printResults(int a, int b) {
  if ((a == 0) || (b > 0))
      System.out.println(″red″, b/a;
  else
      System.out.println(″blue″, y + 2);
  System.out.printlin(″end″);
}
```

$$Condition\ Coverage = \frac{\#\ of\ conditions\ that\ are\ both\ T\ and\ F}{total\ \#\ of\ conditions}$$

TC1: (a = 0, b = -5)
TC2: (a = 5, b = 5)
Branch coverage = 50%
Condition coverage = 100%

Branch & Condition Coverage

- Sometimes branch and condition coverage is also called as “Decision Coverage”
  - It is computed by considering both branch and individual condition coverage measures

Decision Coverage Measure

- How to achieve 100% in this example?

```
public void printResults(int a, int b)
  if ((a == 0) || (b > 0))
      System.out.println("red", y/x);
  else
      System.out.println("blue", y + 2);
  System.out.printlin("end");
}
```

<u>TC1:</u> (a = 0, b = -5)
<u>TC2:</u> (a = 5, b = 5)
<u>TC3:</u> (a = 3, b = -2)

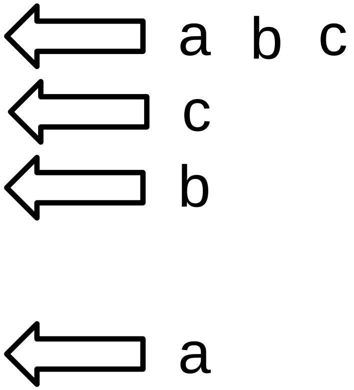
Modified Condition/Decision Coverage (MC/DC)

- **Key idea**: test important combinations of conditions and limiting testing costs
  - Extend branch and decision coverage with the requirement that each condition should affect the decision outcome independently
  - In other words, each condition should be evaluated one time to "true" and one time to "false", and this with *affecting the decision's outcome*.
- Often required for the mission-critical systems

MC/DC Example

a && b && c

Test Case	**a**	**b**	**c**	**Outcome**
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

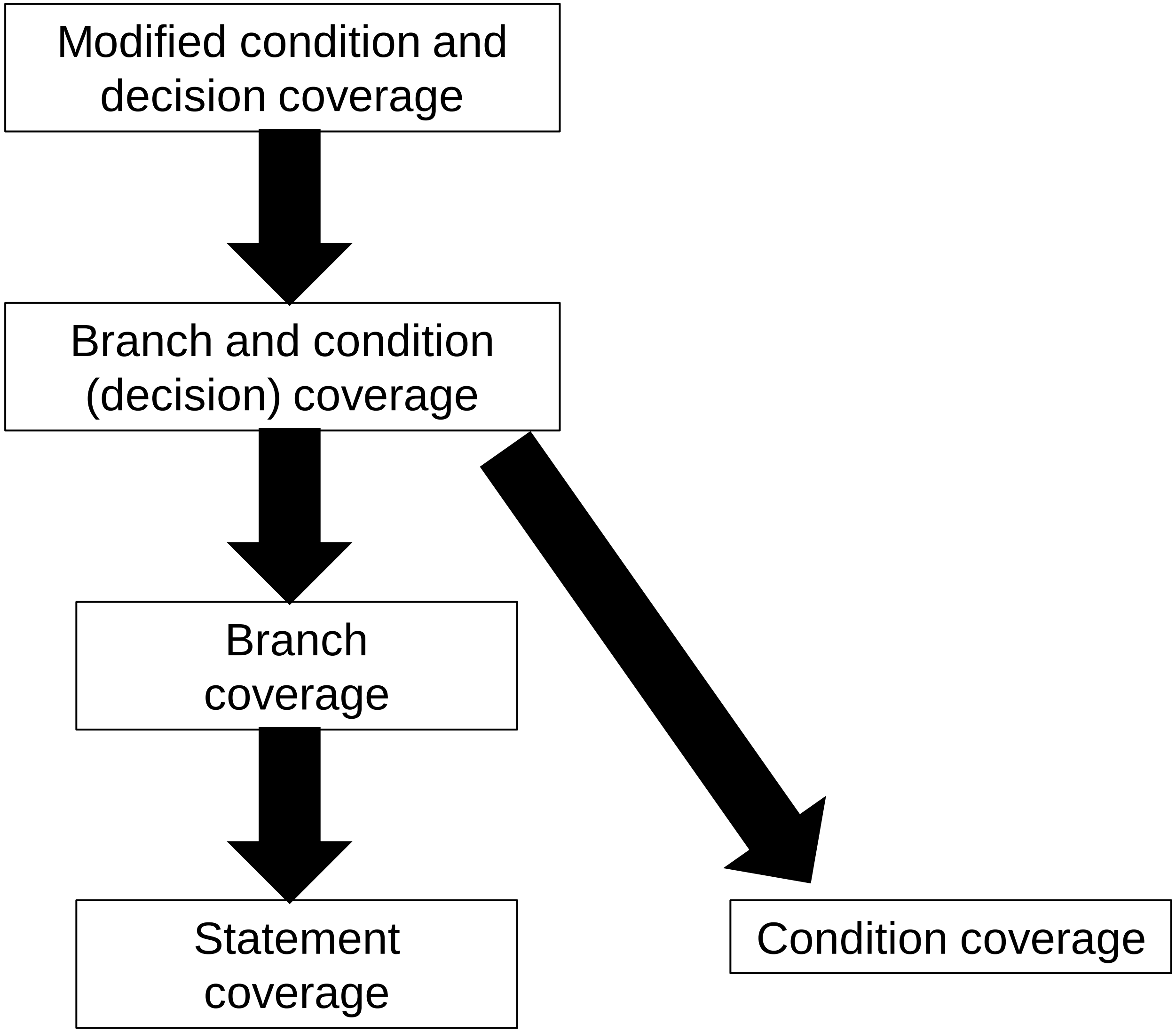


MC/DC (continued)

a && b && c

Test Case	a	b	c	Outcome	
1	T	T	T	T	⇐ a b c
2	T	T	F	F	⇐ c
3	T	F	T	F	⇐ b
~~**4**~~	~~T~~	~~F~~	~~F~~	~~F~~	
5	F	T	T	F	⇐ a
~~**6**~~	~~F~~	~~T~~	~~F~~	~~F~~	
~~**7**~~	~~F~~	~~F~~	~~T~~	~~F~~	
~~**8**~~	~~F~~	~~F~~	~~F~~	~~F~~	

Test Criteria Subsumption



Modified condition and decision coverage

Branch and condition (decision) coverage

Branch coverage

Statement coverage

Condition coverage

Dealing with Loops

- Loops are highly fault-prone, so they need to be tested carefully
- Simple view: Every loop involves a decision to traverse the loop or not
- A bit better: Boundary value analysis on the index variable
- Nested loops have to be tested separately starting with the innermost

Test Case Creation

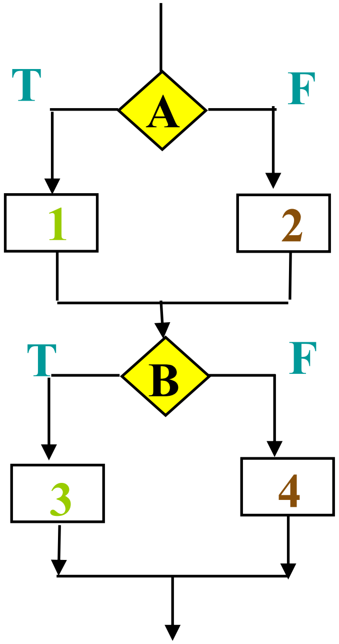
Creating test cases

- In order to increase the coverage of a test suite, one needs to generate test cases that exercise certain statements or follow a specific path
- This is not always easy to do …

CFG question

- **What is the control flow graph for the following?**

```
if (a < b) { c = a + b ; d = a * b }
else { c = a * b ; d = a + b}
if (c < d) { x = a + c ; y = b + d }
else { x = a * c ; y = b * d }
```



Creating a test case

- The key question for creating a test for a path is:
  - How to make the path execute, if possible.
    - Generate input data that satisfies all the conditions on the path
- The key items you need to generate a test case for a path:
  - Input vector
  - Predicate
  - Path predicate
  - Predicate interpretation
  - Path predicate expression
  - Create test input from path predicate expression

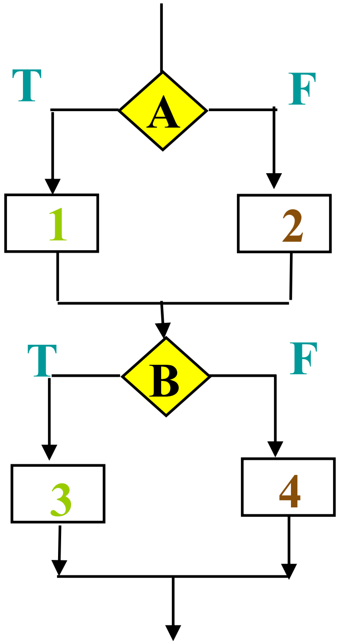
Input Vector

- Input vector is:
  - A collection of all data entities read by the routine whose values must be fixed prior to entering the routine.
- The members of an input vector are:
  - Input arguments to the routine
  - Global variables and constants
  - Files
  - Network connections
  - Timers

Predicate

- A predicate is
  - A logical function evaluated at a decision point.
    - In the following example, each of a < b and c < d are predicates

```
if (a < b) { c = a + b ; d = a * b }
else { c = a * b ; d = a + b }
if (c < d) { x = a + c ; y = b + d }
else { x = a * c ; y = b * d }
```



Path Predicate Expression

- A **path predicate expression** is
  - An interpreted path predicate
- A **path predicate interpretation** is
  - A path predicate may contain local variables.
  - Local variables cannot be selected independently of the input variables
  - Local variables are eliminated with symbolic execution
- A **symbolic execution** is
  - Symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
  - A predicate may have different interpretations depending on how control reaches the predicate.

Attributes of a Path Predicate Interpretation

- The attributes of a path predicate interpretation are:
  - No local variables
  - A set of constraints in terms of the input vector, and, maybe, constants
  - Path forcing inputs are generated by solving the constraints
  - If a path predicate expression has no solution, the path is infeasible

Path Predicate
Generating Input Values

```
if (a < b) { c = a + b ; d = a * b }
else { c = a * b ; d = a + b }
if (c < d) { x = a + c ; y = b + d }
else { x = a * c ; y = b * d }
```

- Path predicate $a < b = \text{true} \wedge c < d = \text{false}$
- Substitute for c and d $c = a + b \quad d = a * b$

$$a < b = \text{true} \;\wedge\; a + b < a * b = \text{false}$$

$$a < b \;\wedge\; a + b \geq a * b$$

Path Predicate Generating Input Values (Continued)

$$a < b \wedge a + b \geq a * b$$

- Solve for a and b $a = 0 \wedge b = 1$
  - Solutions are not unique
- A solution exists
  - We have a feasible path
- No solution to the constraints
  - Have an infeasible path

Organizing path predicates

- We can organize the set of path predicates using a decision table
  - How would a decision table be used?

	A1B3	A1B4	A2B3	A2B4
$A < B$	T	T	F	F
$C < D$	T	F	T	F
A value	2	0	1	5
B value	5	1	0	2

Paths **A1B3** and **A2B4** give statement coverage
or Paths **A1B4** and **A2B3** give statement coverage

Selecting paths

- A program unit may contain a large number of paths.
  - Path selection becomes a problem
  - Some selected paths may be infeasible

- What strategy would you use to select paths?
  - Select as many short paths as possible
    - Tradeoffs?
  - Choose longer paths
    - Tradeoffs?

- What about infeasible paths?
  - What would you do about them?
  - Make an effort to write program text with fewer or no infeasible paths.