CS 5154

Applying Graph Coverage Criteria to Source Code

Owolabi Legunsen

The following are modified versions of the publicly-available slides for Chapter 7 in the Ammann and Offutt Book, "Introduction to Software Testing"
(http://www.cs.gmu.edu/~offutt/softwaretest)

Overview

- How to apply graph-based criteria to source code?

- Graph : Usually the control flow graph (CFG)
- Node coverage : Execute every statement
- Edge coverage : Execute every branch

- Loops : structures such as for loops, while loops, etc.

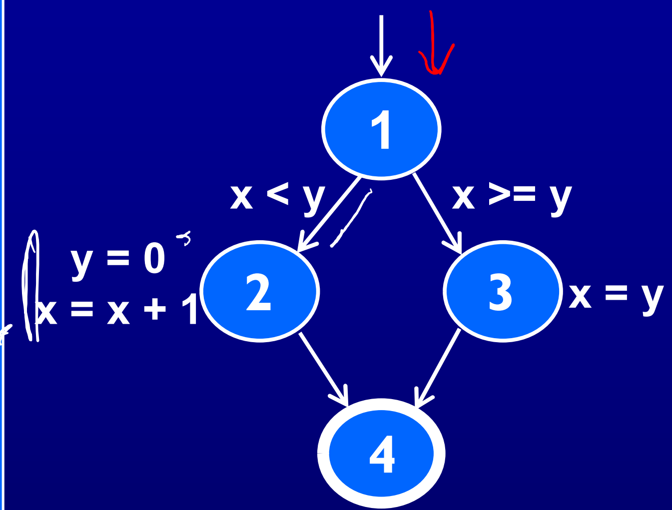
- Data flow coverage : Augment the CFG
  - – defs are statements that assign values to variables
  - – uses are statements that use variables

Control Flow Graphs

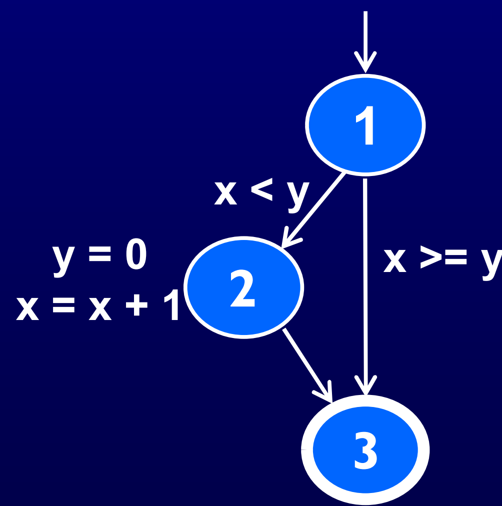
- CFG captures control structures in method executions
- Nodes: Statements or statement sequences (basic blocks)
- Edges : Transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates, defs, uses
- Rules for translating statements into graphs …

CFG : The if Statement

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

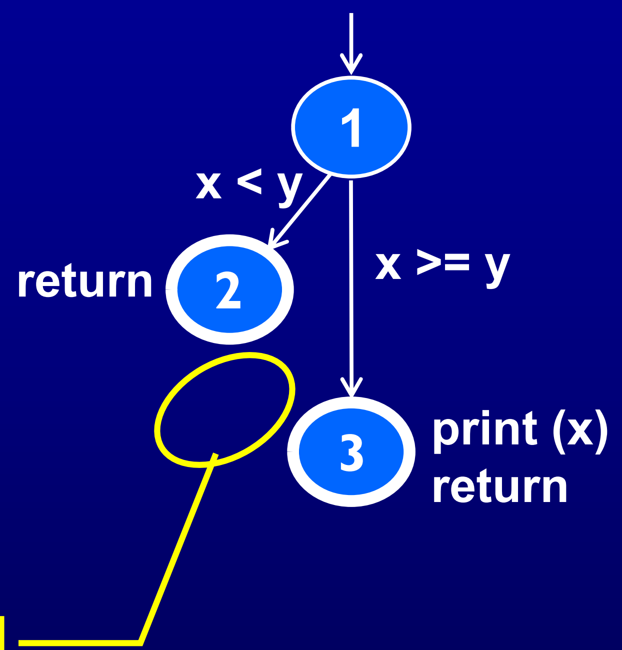



```
if (x < y)
{
  y = 0;
  x = x + 1;
}
```



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.

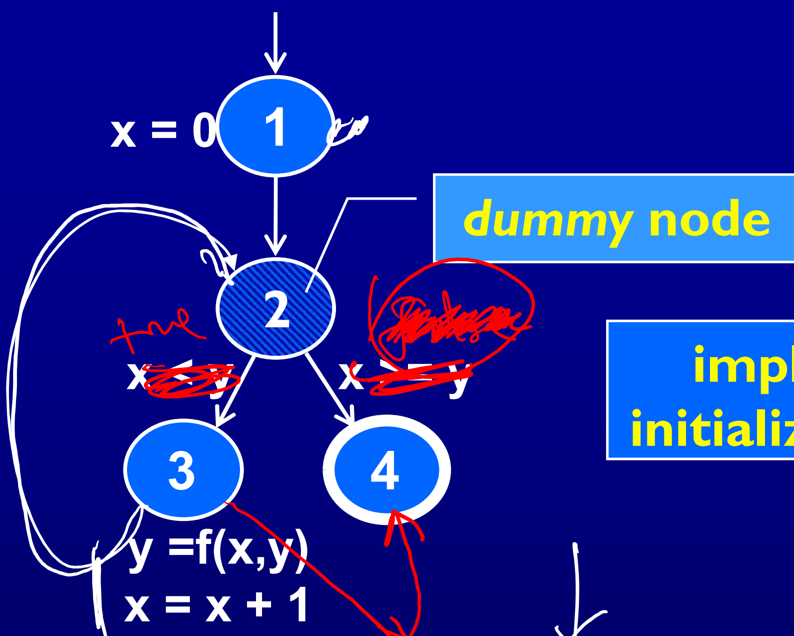
Loops

- Loops require "*extra*" nodes to be added

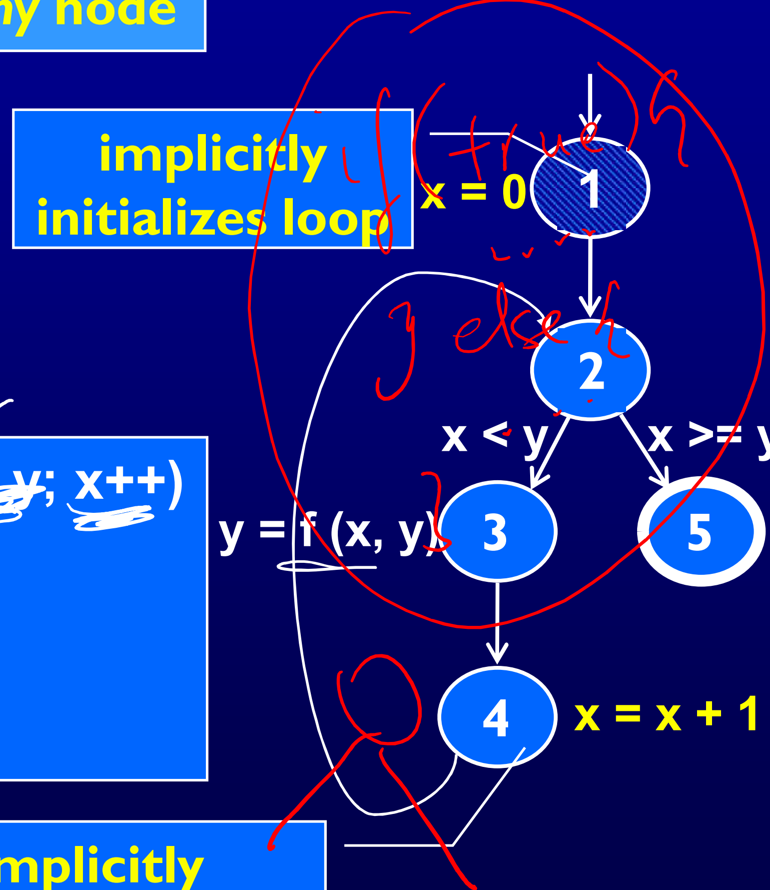
- Nodes that do not represent statements or basic blocks

CFG : while and for Loops

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



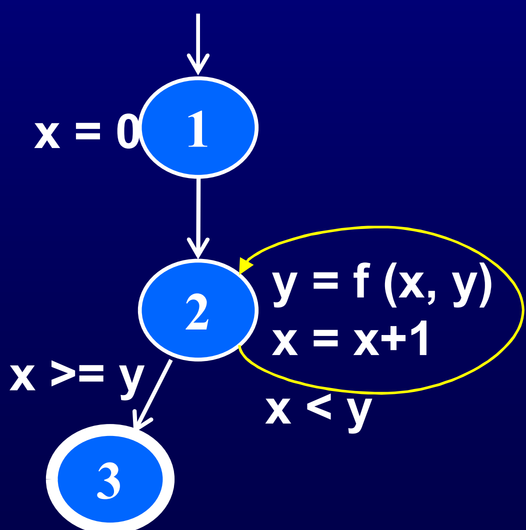

```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
return (x);
```



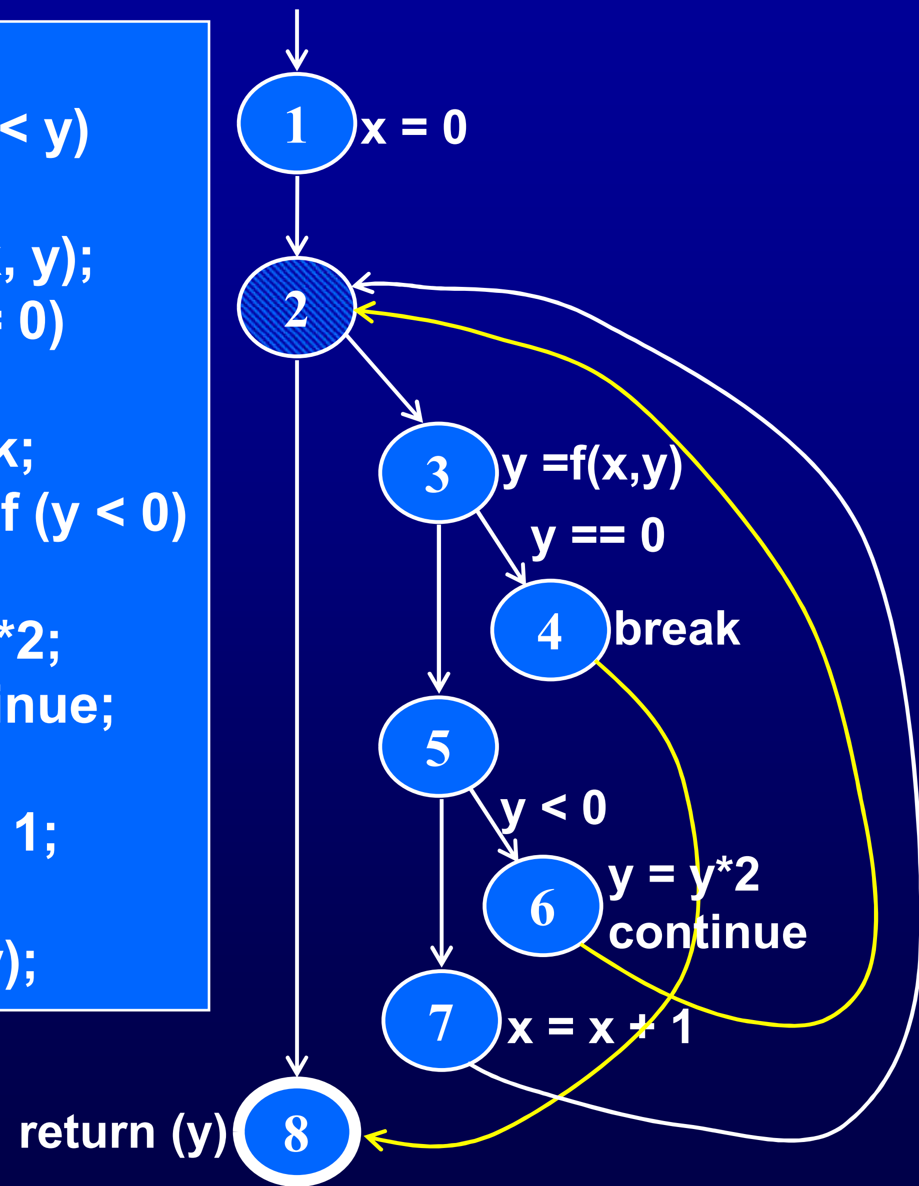
implicitly increments loop

CFG : do Loop, break and continue

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

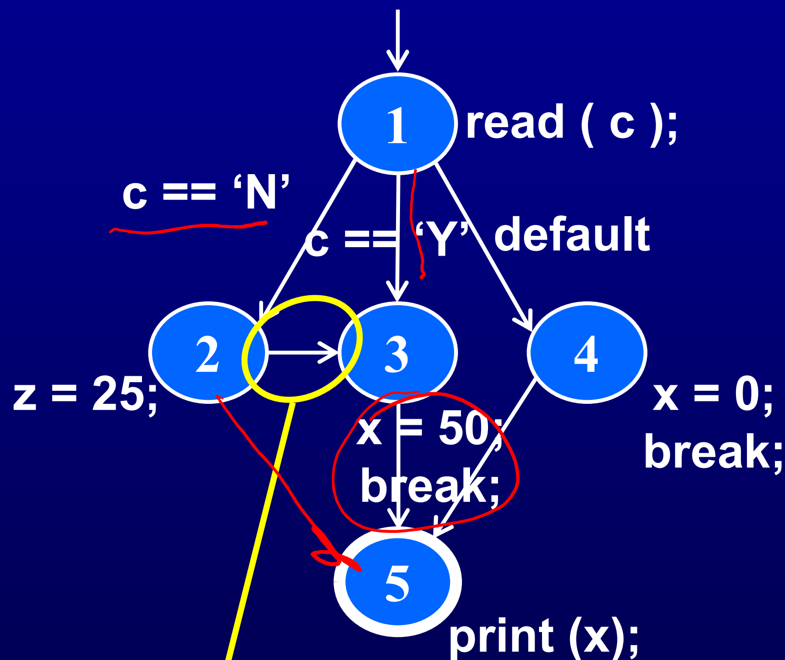



```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
return (y);
```



CFG : The case (switch) Structure

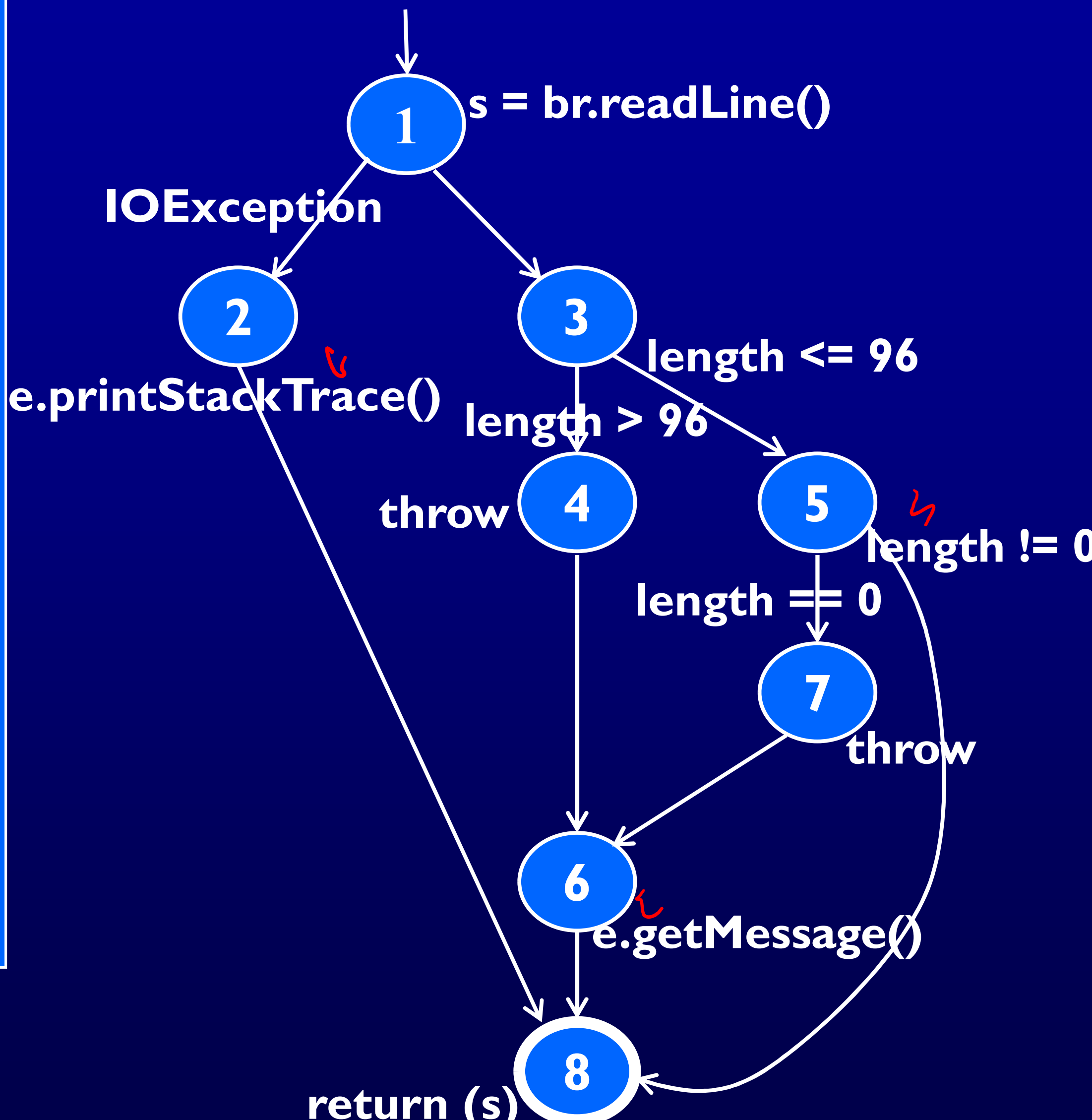
```
read ( c) ;
switch ( c )
{
   case 'N':
      z = 25;
   case 'Y':
      x = 50;
      break;
   default:
      x = 0;
      break;
}
print (x);
```



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);
```



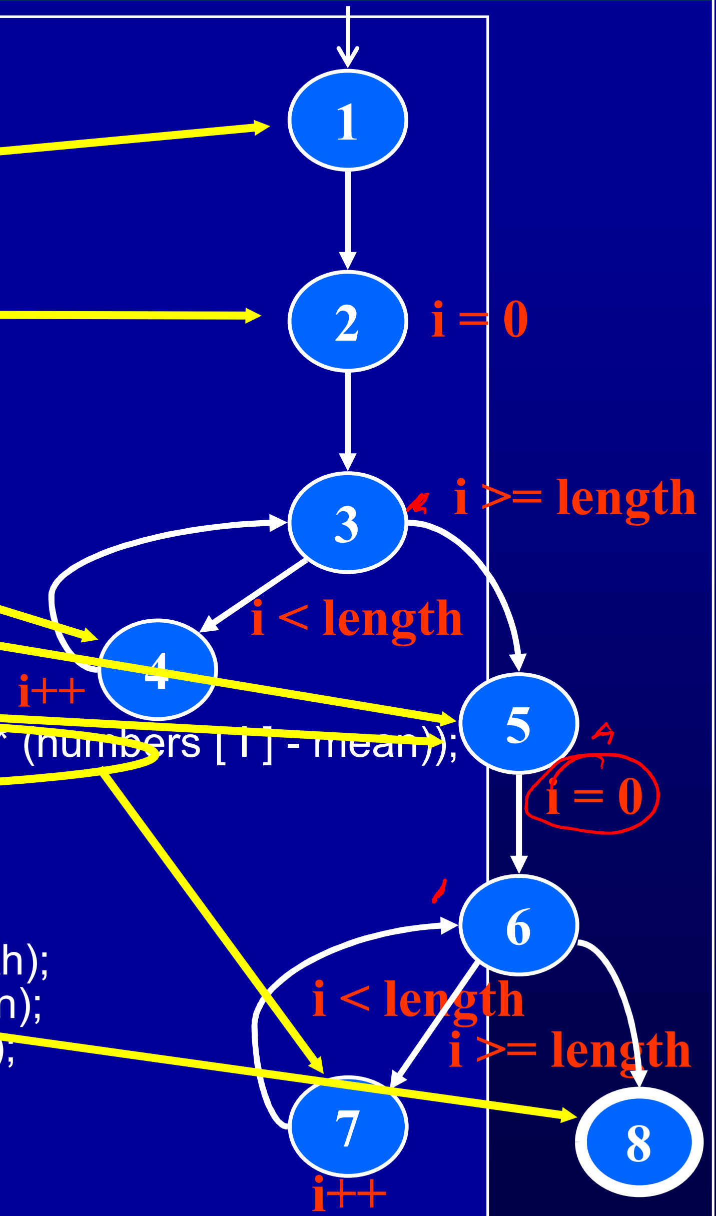
Example Control Flow – Stats

```
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);
}
```

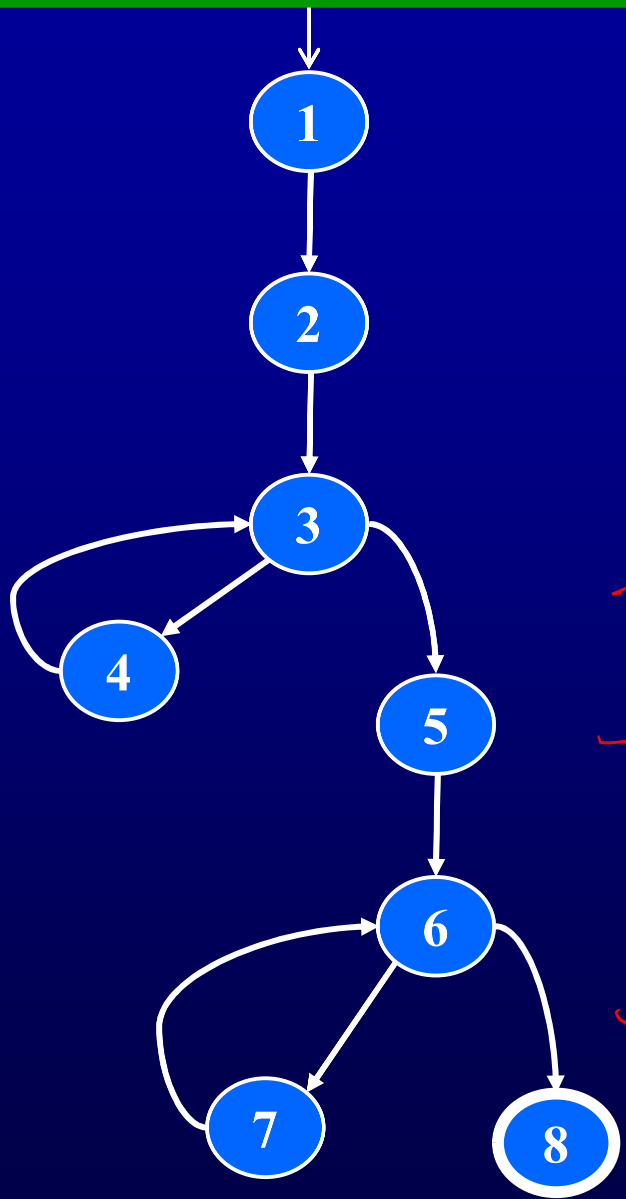
Draw the graph and label the edges.

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);
}
```

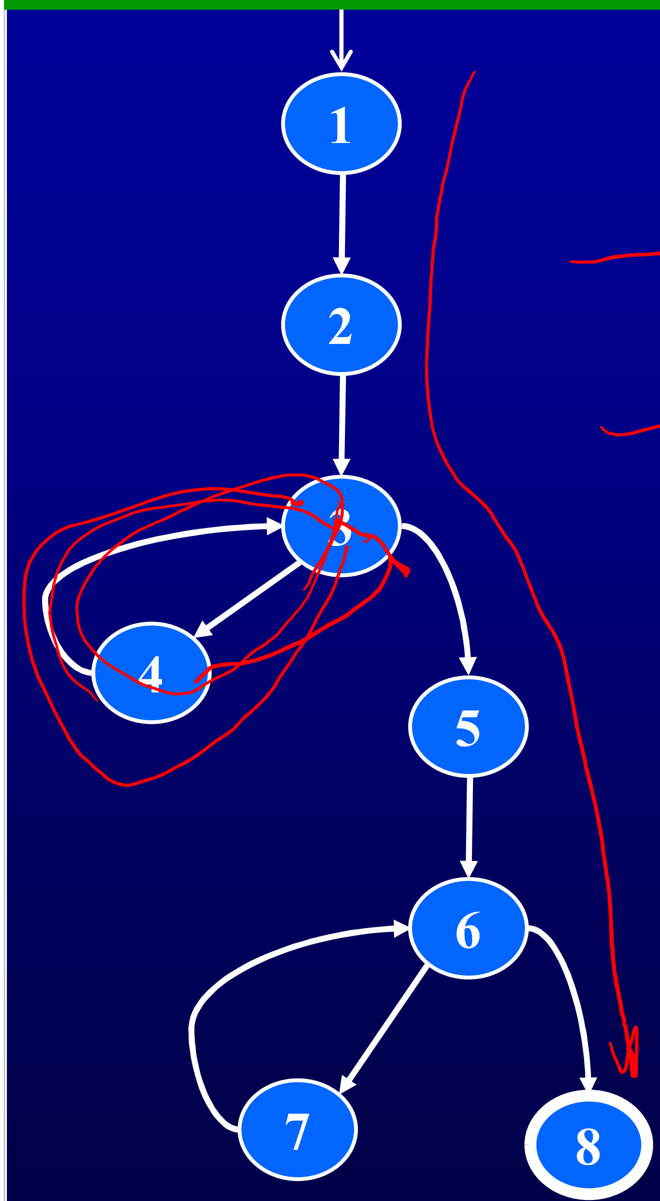


Control Flow TRs and Test Paths—EC



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

Control Flow TRs and Test Paths—EPC



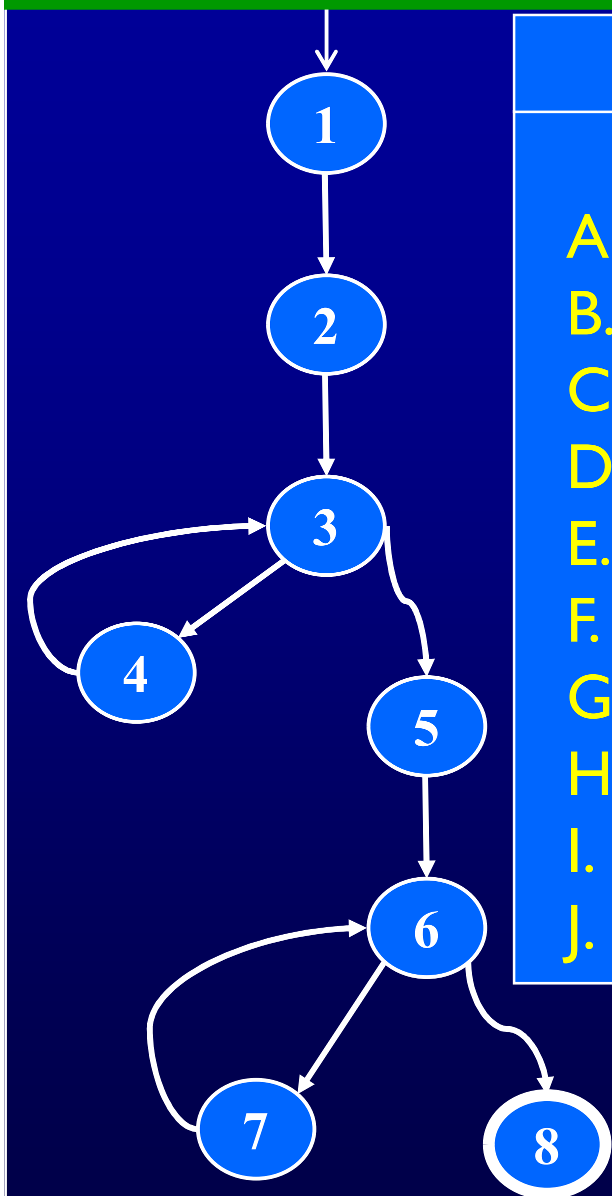
Edge-Pair Coverage

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	*sidetrips*
~~i~~	~~A, B, D, E, F, G, I, J~~	~~C, H~~
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

TP iii makes TP i redundant. A *minimal* set of TPs is cheaper.

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	*sidetrips*
~~i~~	~~A, D, E, F, G~~	~~H, I, J~~
ii	A, B, C, D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	

TP ii makes TP i redundant.

Data Flow Coverage for Source

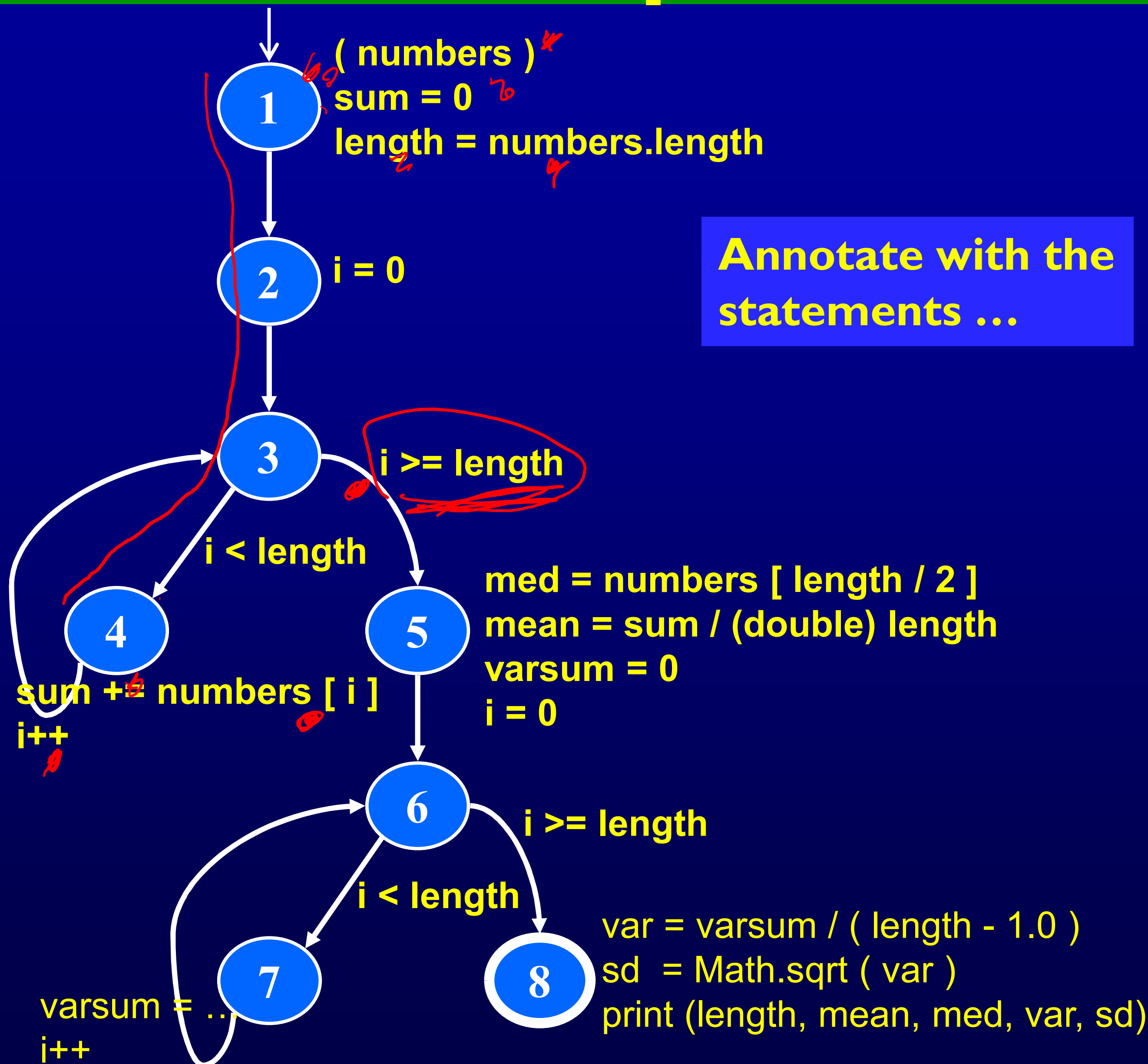
- **def** : a location where a value is stored into **memory**
  - x appears on the **left side** of an assignment (e.g., x = 44;)
  - x is an **actual parameter** in a call site & method **changes** x's value
  - x is a method's **formal parameter** (implicit def on method start)
  - x is an **input** to a program
- **use** : a location where variable's value is **accessed**
  - x appears on the **right side** of an assignment (e.g., y = sqrt(x);)
  - 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

- A def and a use on the **same node** is only a DU-pair if the def occurs **after** the use and the node is in a loop

Example Data Flow – Stats

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

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    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);
}
```

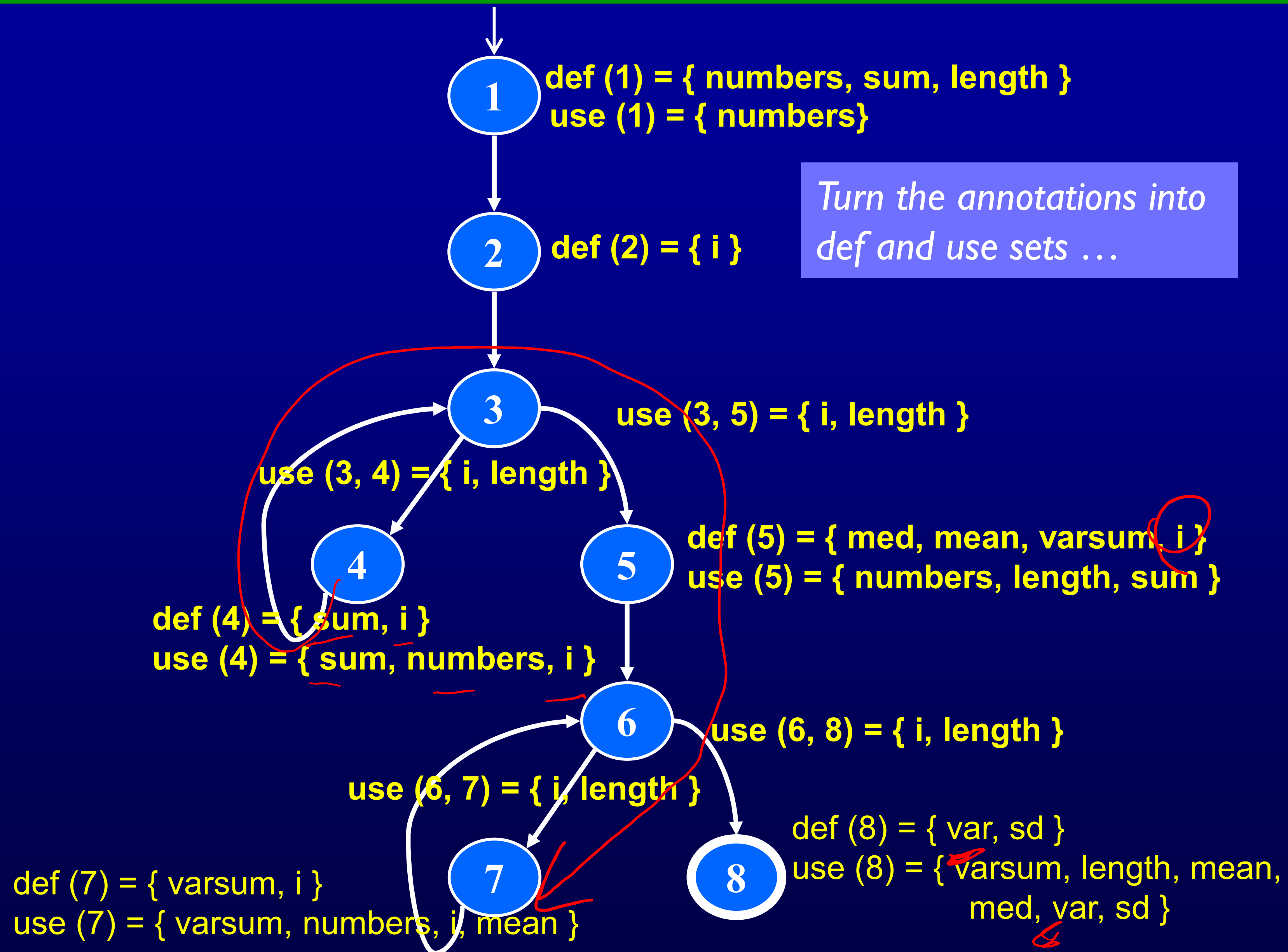
Control Flow Graph for Stats



Annotate with the statements …

CFG for Stats – With Defs & Uses

Turn the annotations into def and use sets …



Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
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 }

Recall: DU Pairs and DU Paths

- def (n) or def (e) : The set of variables that are defined by node n or edge e
- use (n) or use (e) : The set of variables that are used by node n or edge e

- DU pair : A pair of locations (l_i, l_j) s.t. a variable *v* is defined at l_i and used at l_j

- Def-clear : Path from l_i to l_j is *def-clear* w.r.t. *v* if *v* is not given another value on any of the nodes or edges in the path

- Reach : If there is a def-clear path from l_i to l_j with respect to *v*, the def of *v* at l_i reaches the use at l_j

- du-path : A simple subpath that is def-clear w.r.t. *v* from a def of *v* to a use of *v*
- Def-path set, du (n_i, *v*) – the set of du-paths that start at n_i
- Def-pair set, du (n_i, n_j, *v*) – the set of du-paths from n_i to n_j

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 before uses, do not count as DU pairs

defs after 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, 2, 3, 4]
	(1, 5)	[1, 2, 3, 5]
	(1, 7)	[1, 2, 3, 5, 6, 7]
length	(1, 5)	[1, 2, 3, 5]
	(1, 8)	[1, 2, 3, 5, 6, 8]
	(1, (3,4))	[1, 2, 3, 4]
	(1, (3,5))	[1, 2, 3, 5]
	(1, (6,7))	[1, 2, 3, 5, 6, 7]
	(1, (6,8))	[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, 2, 3, 4]
	(1, 5)	[1, 2, 3, 5]
	(4, 4)	[4, 3, 4]
	(4, 5)	[4, 3, 5]

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

★ **4 expect a loop not to be "entered"**

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

✸ **2 require at least <u>two</u> iterations of a loop**

Test Inputs and Test Paths

Test input: 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 Input : 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 index out of bounds exception…

med = numbers [length / 2];

A fault was found

Summary

- Applying the graph test criteria to control flow graphs is relatively straightforward
  - Most of the developmental research work was done with CFGs
- A few subtle decisions must be made to translate control structures into the graph
- Some tools will assign each statement to a unique node
  - These slides and the book uses basic blocks
  - Coverage is the same, although the bookkeeping will differ

Next

- Logic coverage

- Some announcements
  - Sprint 0.2 was due at 9:30am today
  - Talk to me if you are in a distant time zone
  - HW2 has been released on CMS, due 2/29 at 9:30am
  - HW2 is to be done individually, no discussion on Ed