CSSE 3012
The Software Process

Code Coverage

Coverage Criteria

- Even small programs have too many inputs to fully test them
  - int diff (int A, int B)
  - each variable has over 4 billion possible values (32-bit machine)
- Testers search a huge input space
  - use fewest inputs to find the most problems
- Coverage criteria give structured, practical ways to search the input space
  - search the input space thoroughly
  - reduce overlap in the tests
- Coverage criteria gives testers a “stopping rule” … when testing is finished
- Coverage criteria can be well supported with powerful tools

Example : Jelly Bean Coverage

- Flavors :
  1. Lemon
  2. Pistachio
  3. Cantaloupe
  4. Pear
  5. Tangerine
  6. Apricot

- Colors :
  1. Yellow (Lemon, Apricot)
  2. Green (Pistachio)
  3. Orange (Cantaloupe, Tangerine)
  4. White (Pear)

coverage criteria :

- Taste one jelly bean of each flavor
- Taste one jelly bean of each color

Coverage

Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

- Infeasible test requirements
  - No test case values meet the test requirements
  - Example: Dead code
- 100% coverage is impossible in practice

Comparing Criteria with Subsumption

- **Subsumption**: A test criterion *C1* subsumes *C2* if and only if **every set of test cases** that satisfies C1 also satisfies C2

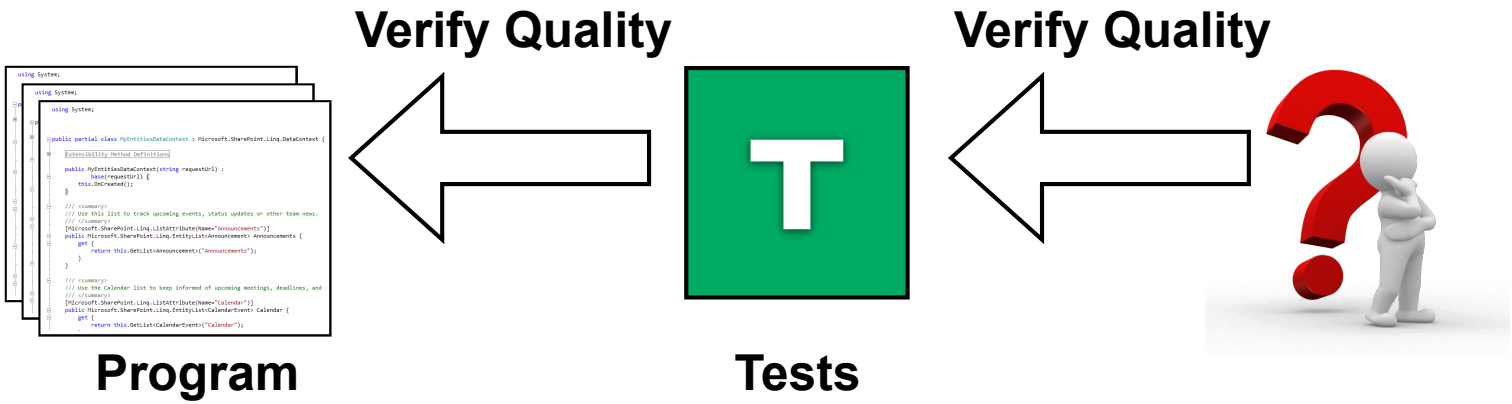
- Examples :
  - The *flavour* criterion on jelly beans subsumes the *colour* criterion
    - if we taste every flavour ==> we taste every colour
  - branch coverage criterion subsumes statement coverage criterion
    - if a test set has covered every branch in a program, then the test set is guaranteed to also have covered every statement

Code coverage

- Control-flow coverage
  - Statement coverage
  - Branch coverage
  - Path coverage

Who will test the tests?

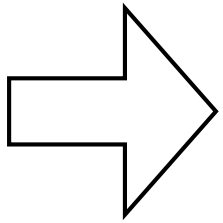
- Code coverage can be a way!
  - Usually, a test covering/executing more code may indicate better test quality

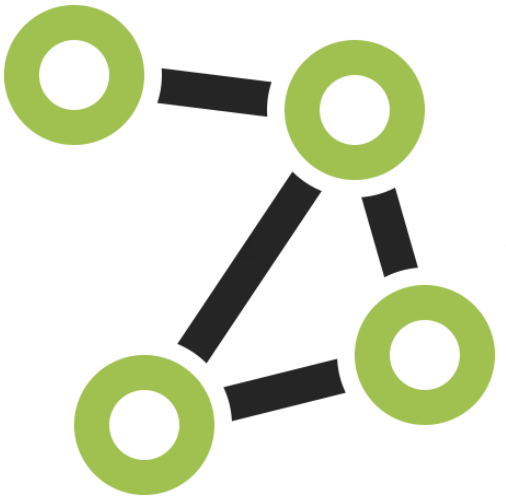


How to measure code coverage?

Overview

- A common way is to abstract program into graphs
  - Graph : Usually the control flow graph (CFG)
  - Node coverage : Execute every statement
  - Edge coverage : Execute every branch



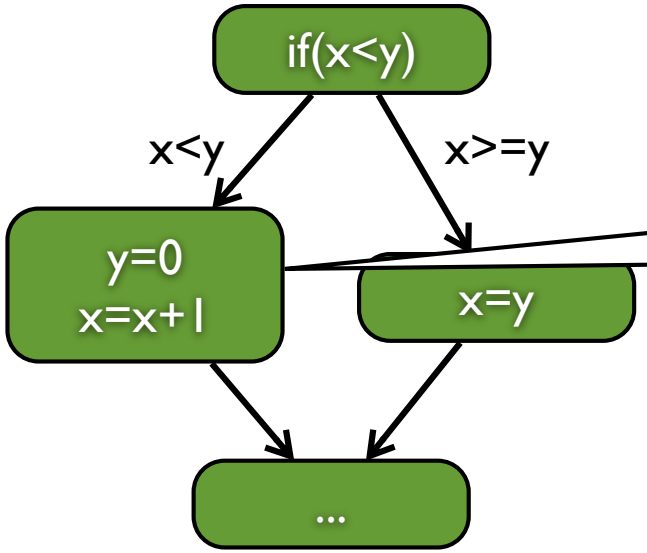


Control flow graphs

- A CFG models all executions of a program by describing control structures
  - Node : Sequences of statements (basic block)
    - Basic Block : A sequence of statements with only one entry point and only one exit point (no branches)
  - Edge : Transfers of control

CFG : The if statement

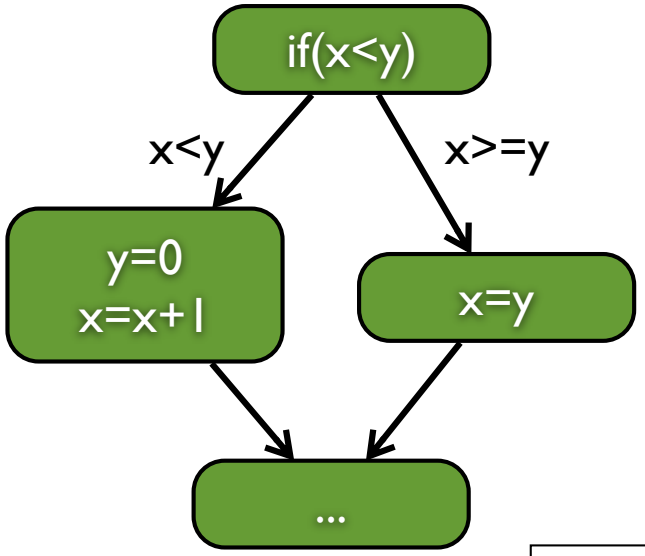
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
...
```



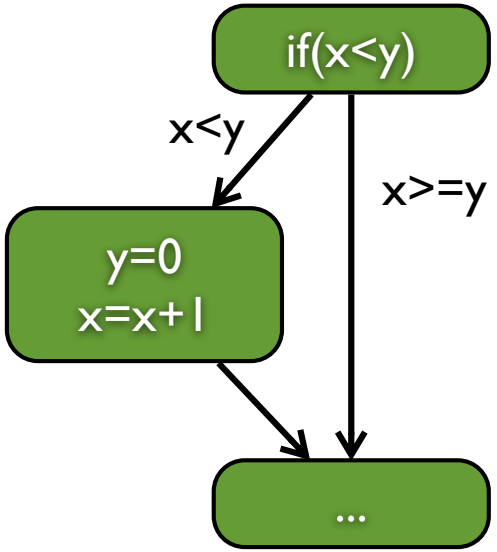
they can be in the same nodes because they are a straight-line code sequence with no branches in except to the entry, no branches out except to the exit (**basic block**)

CFG : The if statement

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

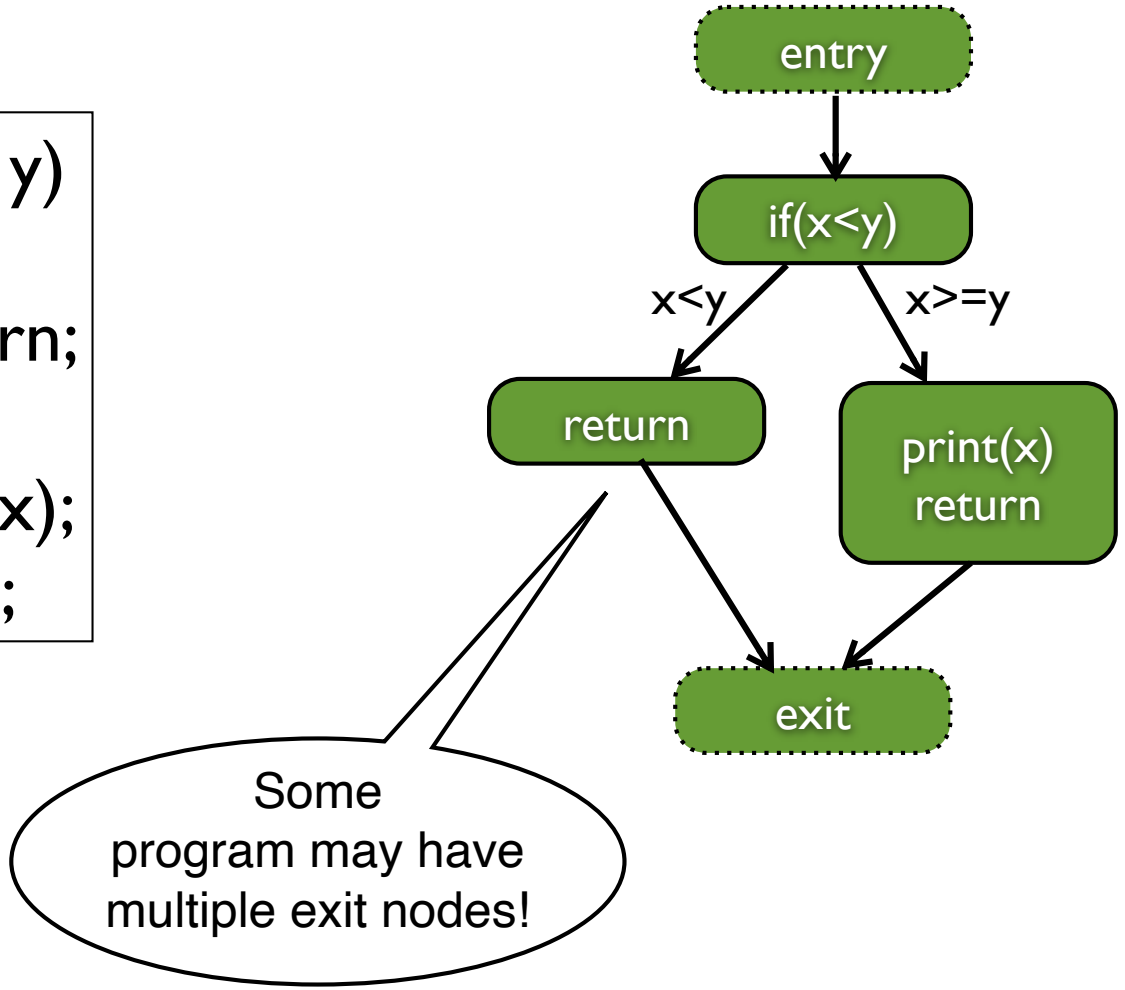



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
...
```



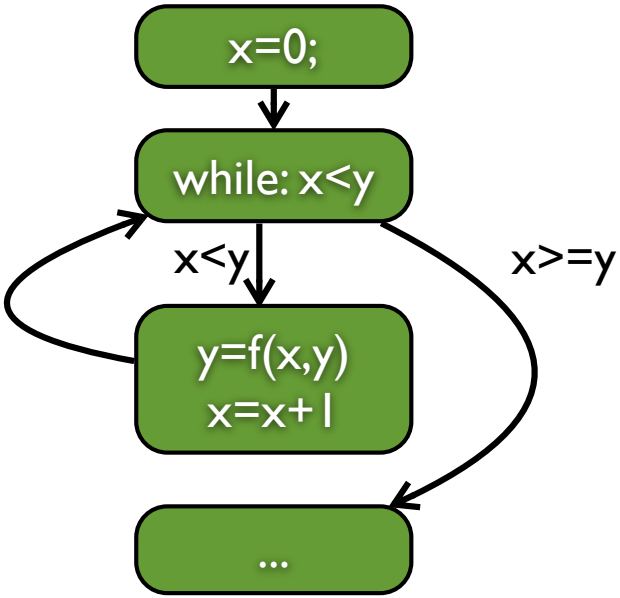
CFG : The dummy nodes

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



CFG : while and for loops

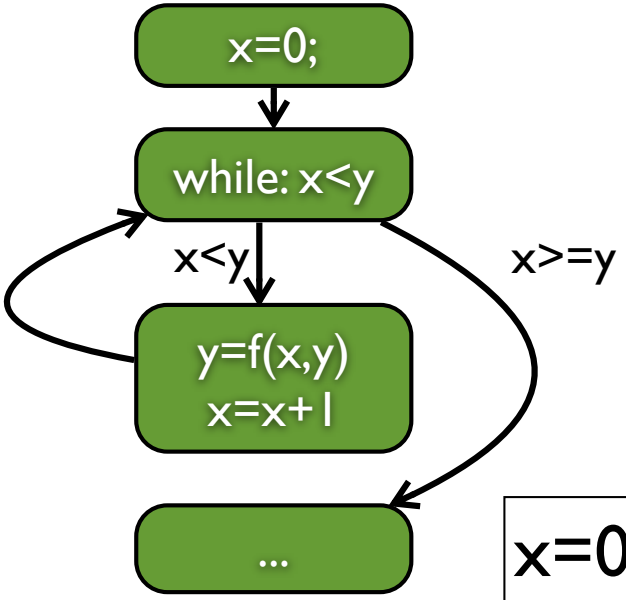
```
x=0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
...
```




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

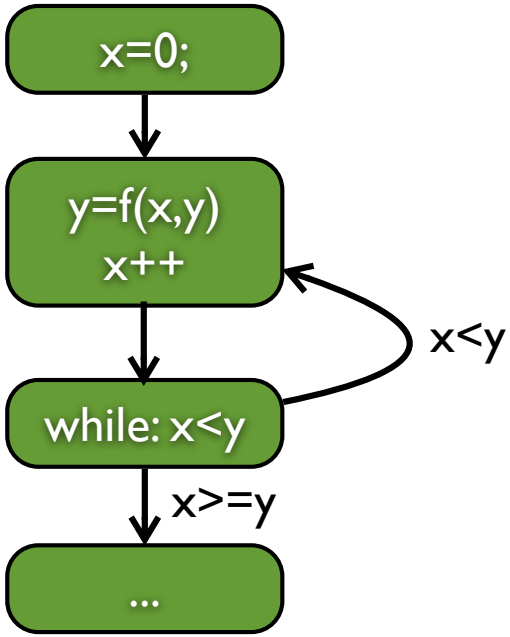
CFG : while and for loops

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



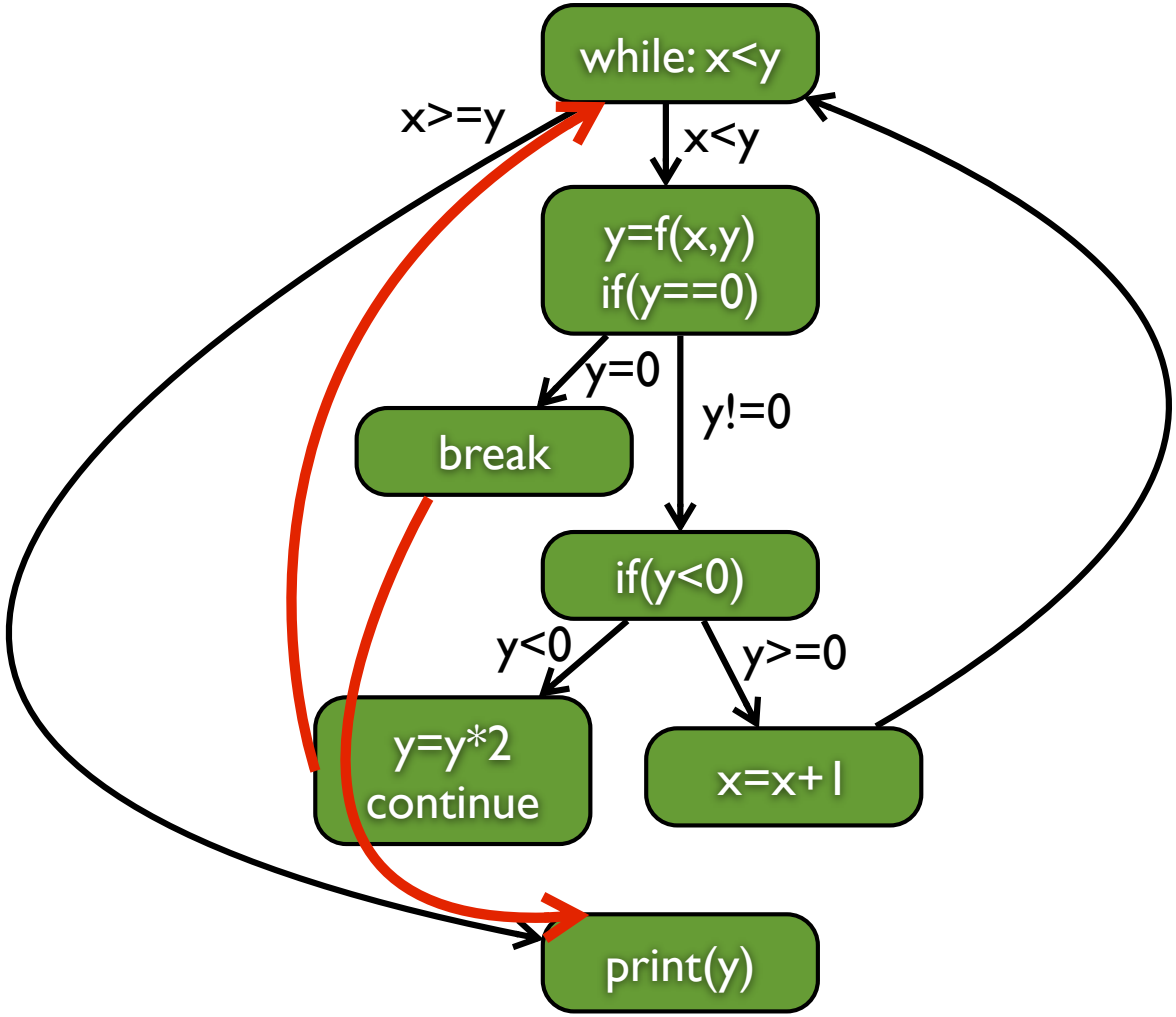

```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

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



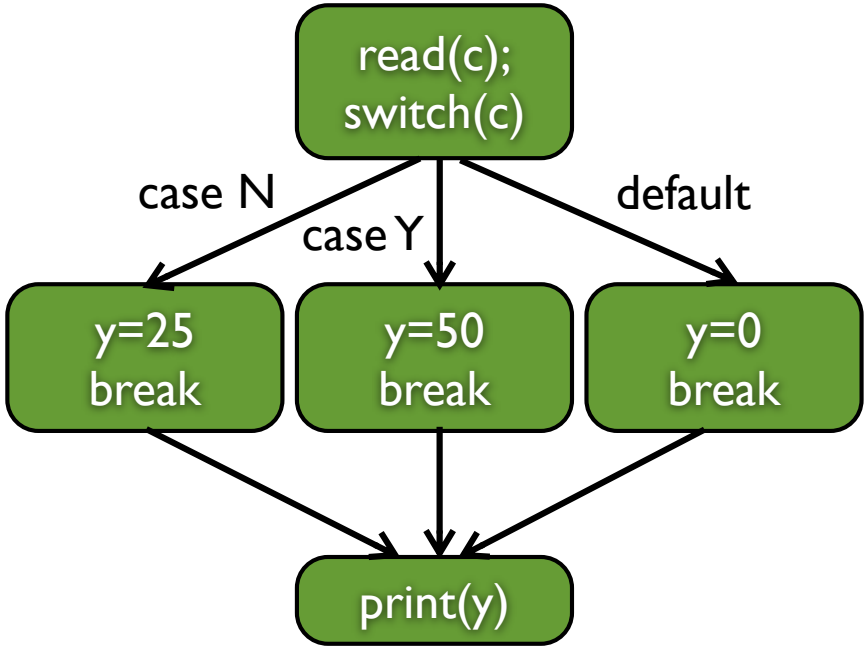
CFG: break and continue

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



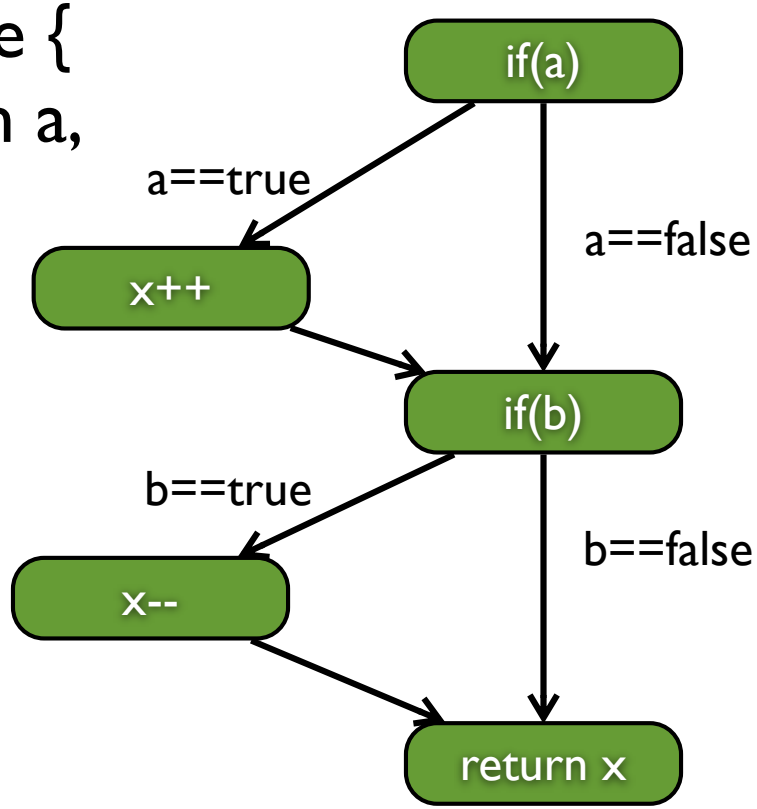
CFG: switch

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



CFG-based coverage: example

```
public class CFGCoverageExample {
    public int testMe(int x, boolean a,
boolean b){
        if(a)
            x++;
        if(b)
            x--;
        return x;
    }
}
```



CFG-based coverage: a JUnit test

```
public class JUnitStatementCov {
    CFGCoverageExample tester;
    @Before
    public void initialize() {
        tester = new CFGCoverageExample();
    }
    @Test
    public void testCase() {
        assertEquals(1, tester.testMe(0, true, false));
    }
}
```

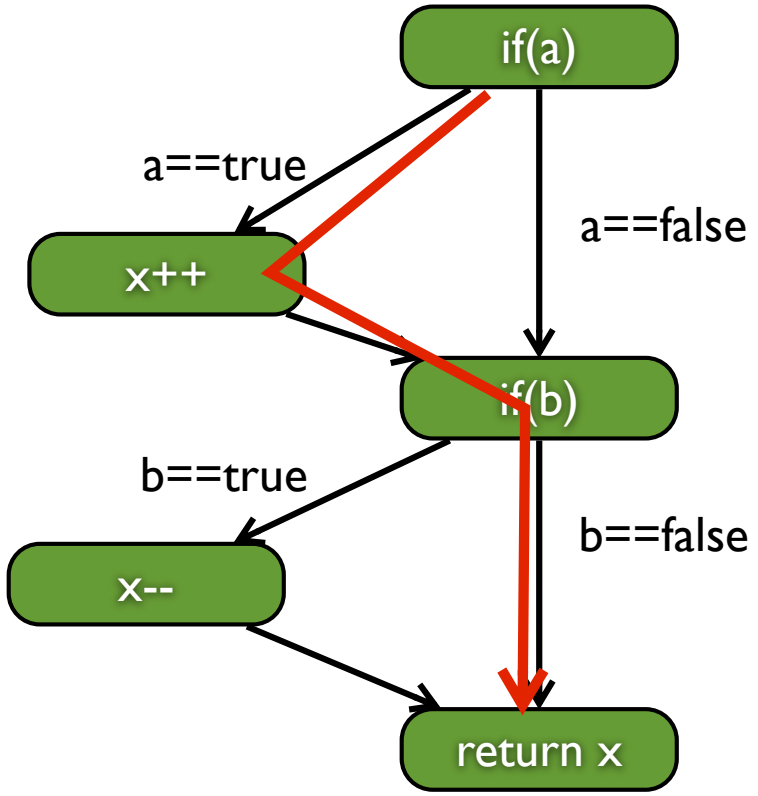
How good is it??

CFG-based coverage: statement coverage

- The percentage of statements covered by the test

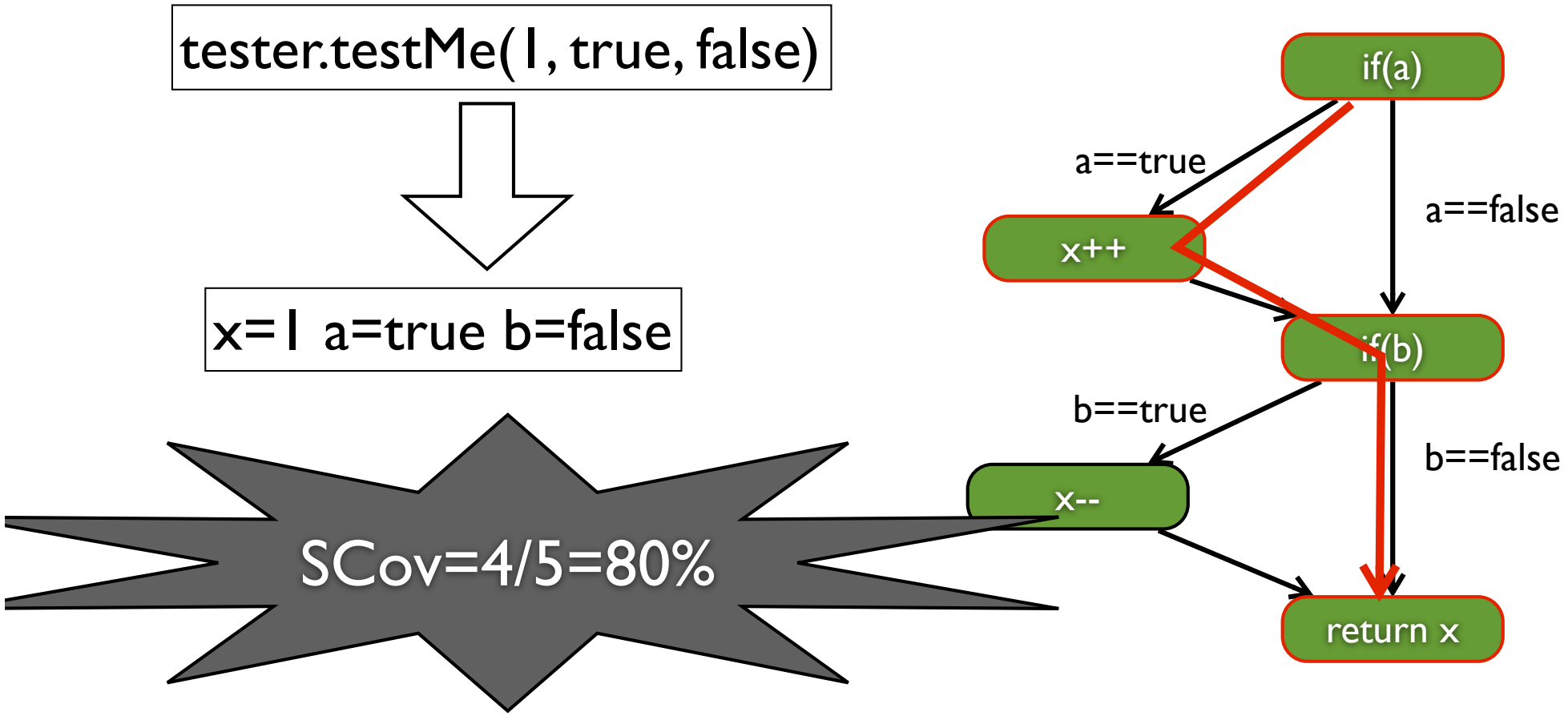
tester.testMe(1, true, false)

x=1 a=true b=false



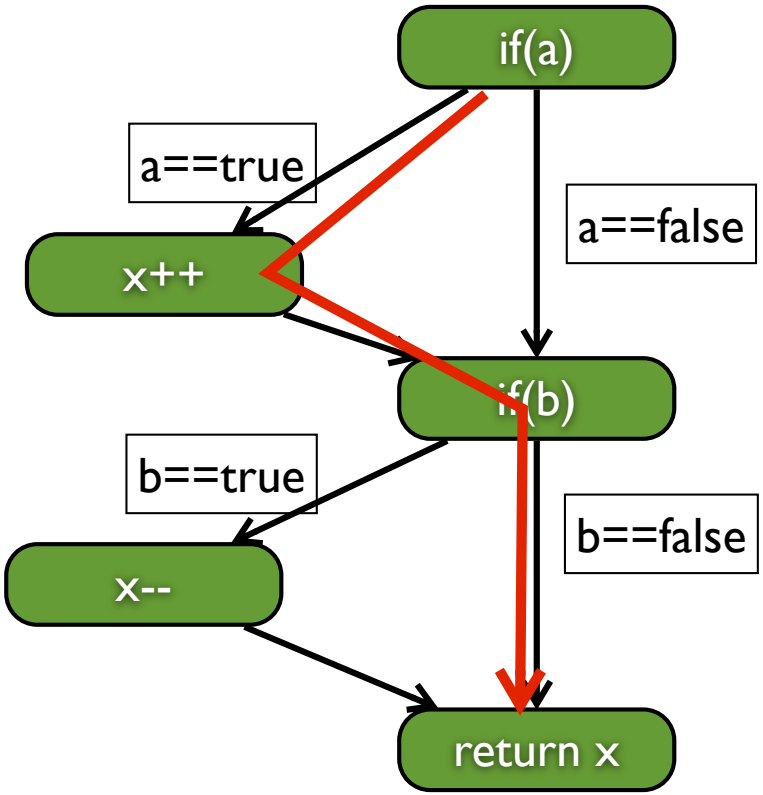
CFG-based coverage: statement coverage

- The percentage of statements covered by the test



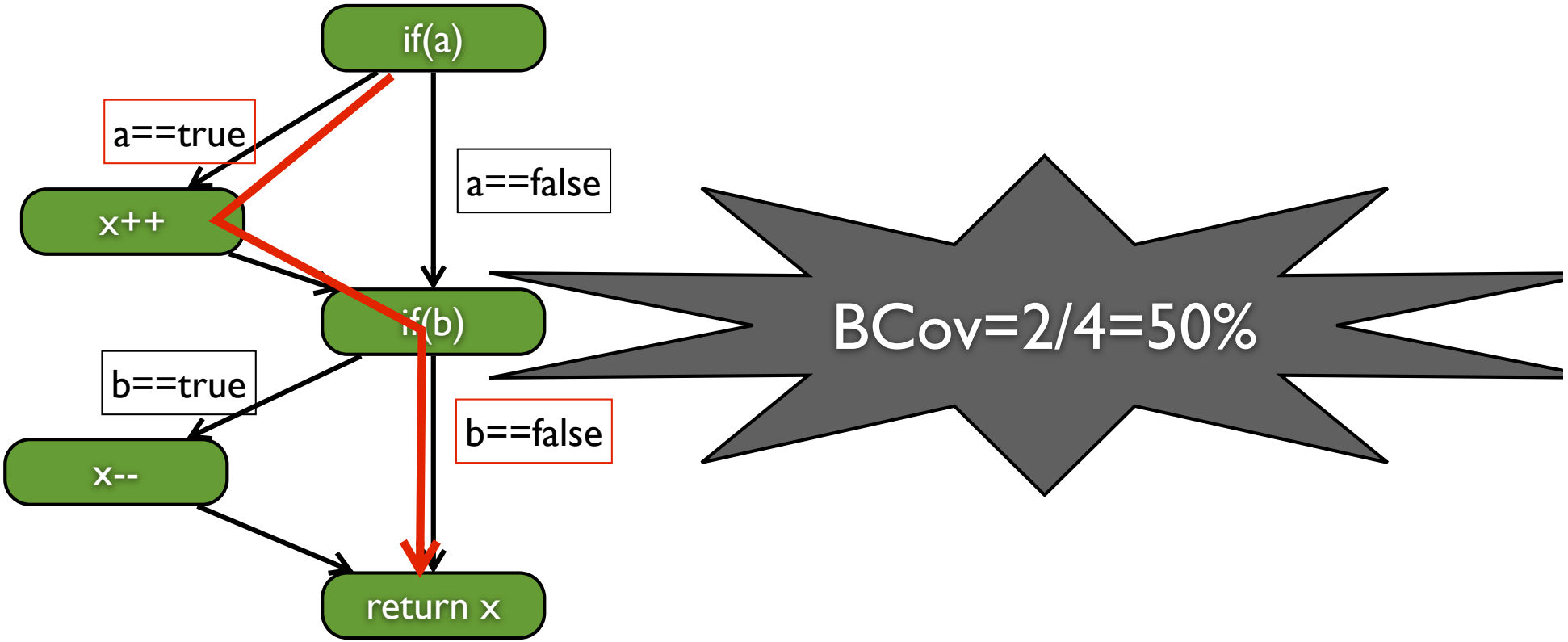
CFG-based coverage: branch coverage

- The percentage of branches covered by the test
  - Consider both false and true branch for each conditional statement



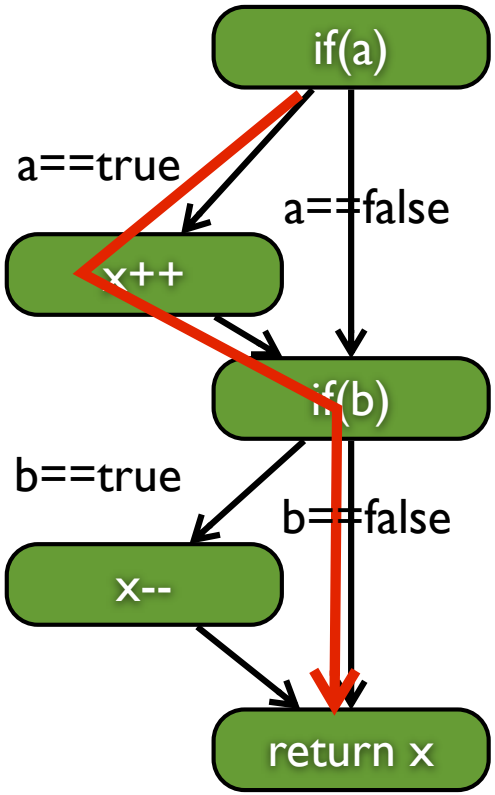
CFG-based coverage: branch coverage

- The percentage of branches covered by the test
  - Consider both false and true branch for each conditional statement



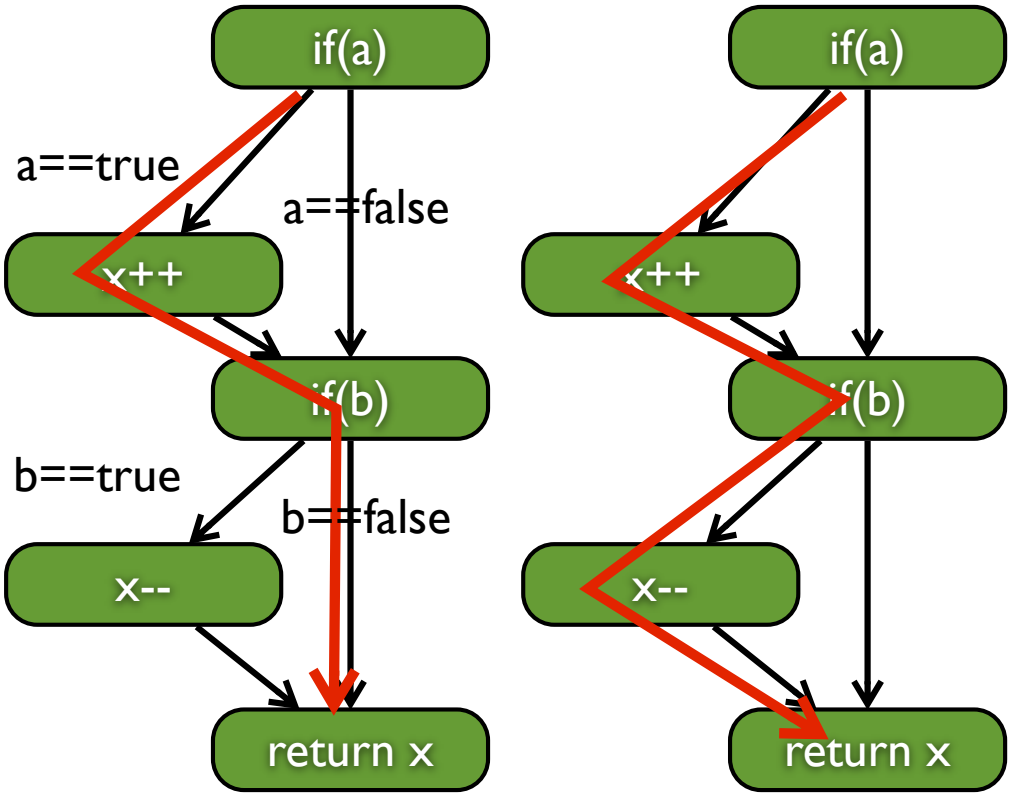
CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



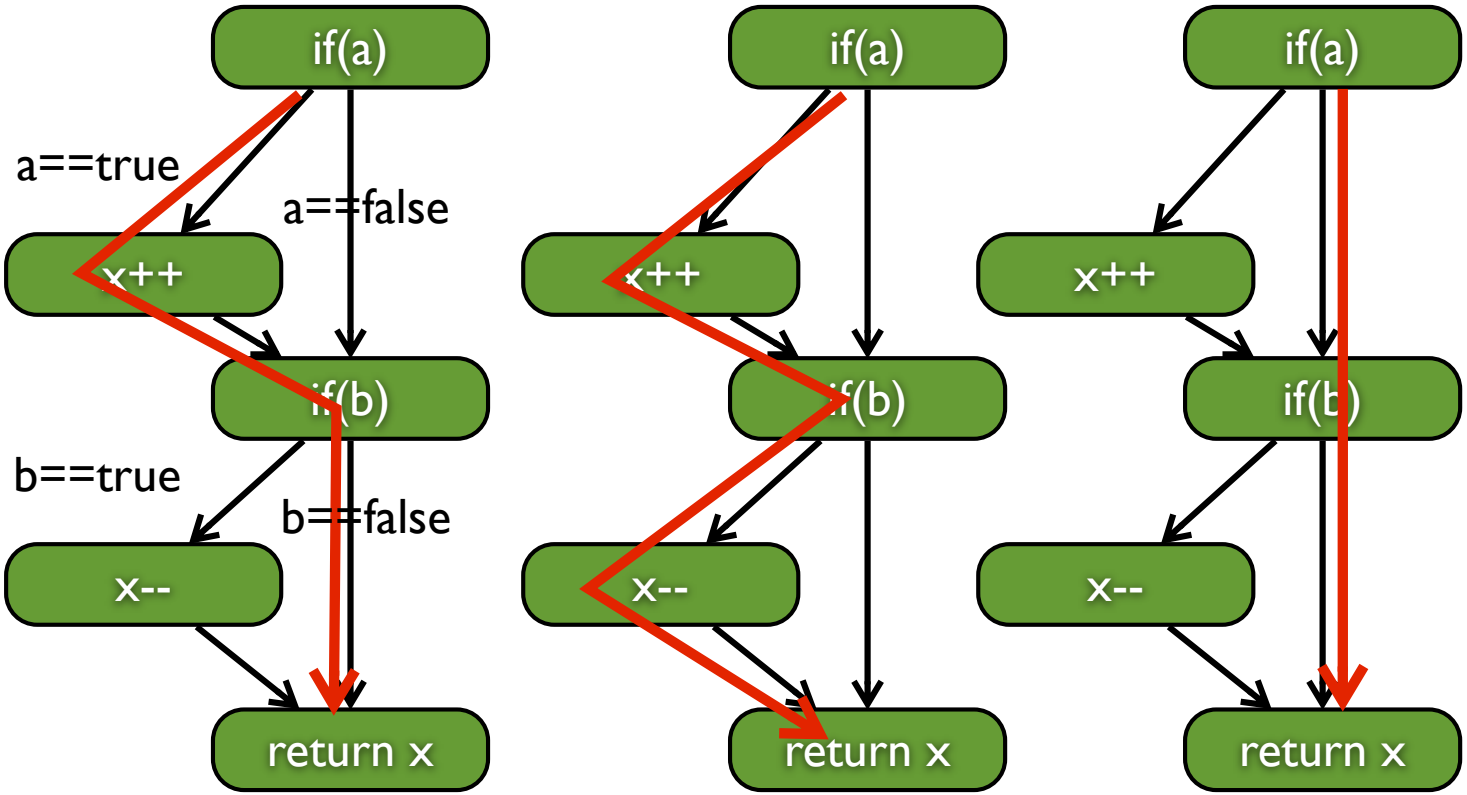
CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



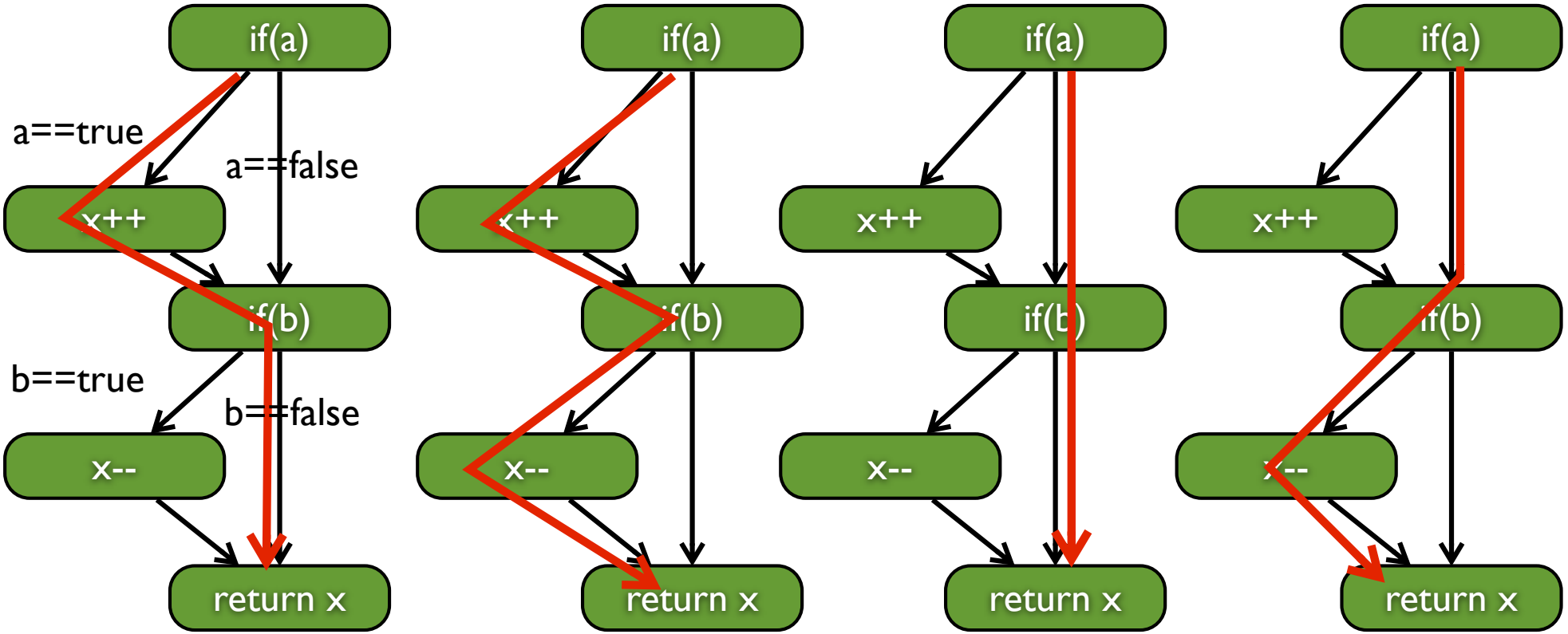
CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



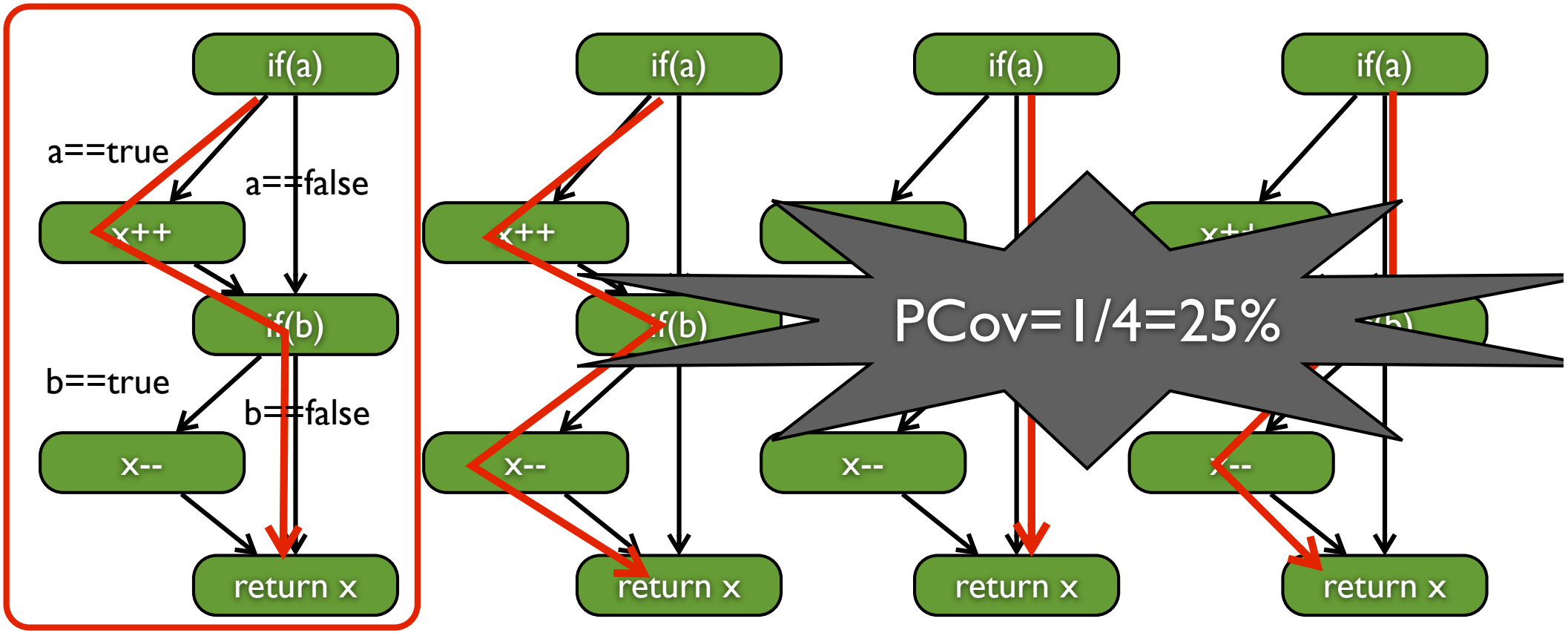
CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



CFG-based coverage: comparison

```
public class JUnitStatementCov {
    CFGCoverageExample tester;
    @Before
    public void initialize() {
        tester = new CFGCoverageExample();
    }
    @Test
    public void testCase() {
        assertEquals(1, tester.testMe(0, true, false));
    }
```

Statement coverage: 80%
Branch coverage: 50%
Path coverage: 25%

If we achieve 100% branch coverage, do we get 100% statement coverage for free?

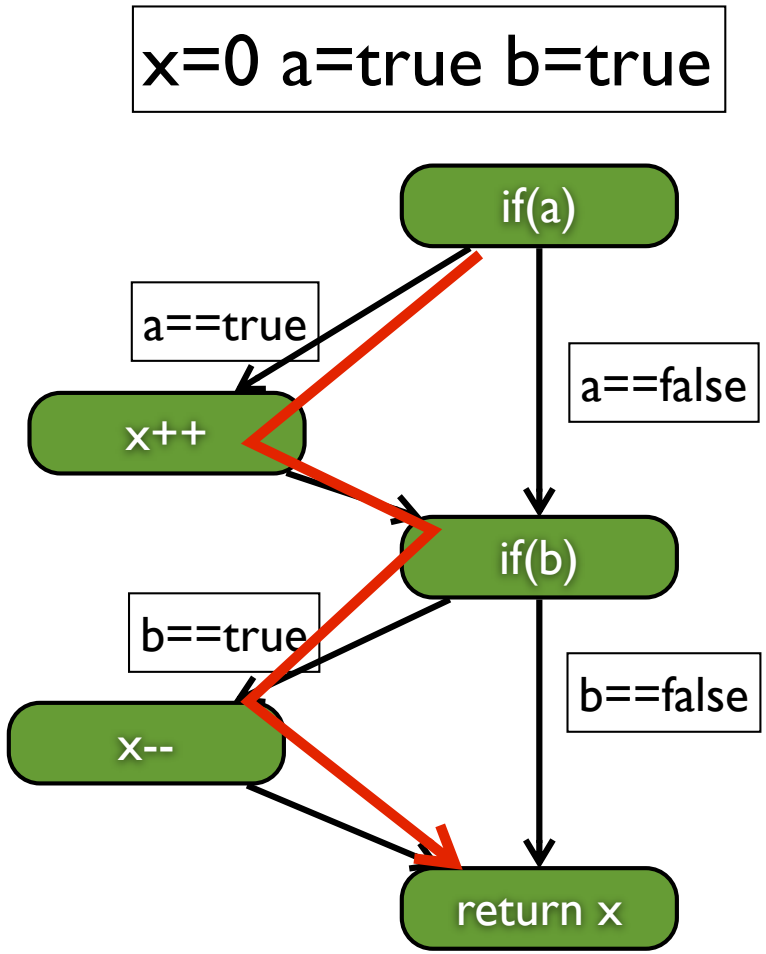
If we achieve 100% path coverage, do we get 100% branch coverage for free?

Statement coverage VS. branch coverage

- If a test suite achieve 100% b-coverage, it must achieve 100% s-coverage
  - The statements not in branches will be covered by any test
  - All other statements are in certain branch
- If a test suite achieve 100% s-coverage, will it achieve 100% b-coverage?

Statement coverage VS. branch coverage

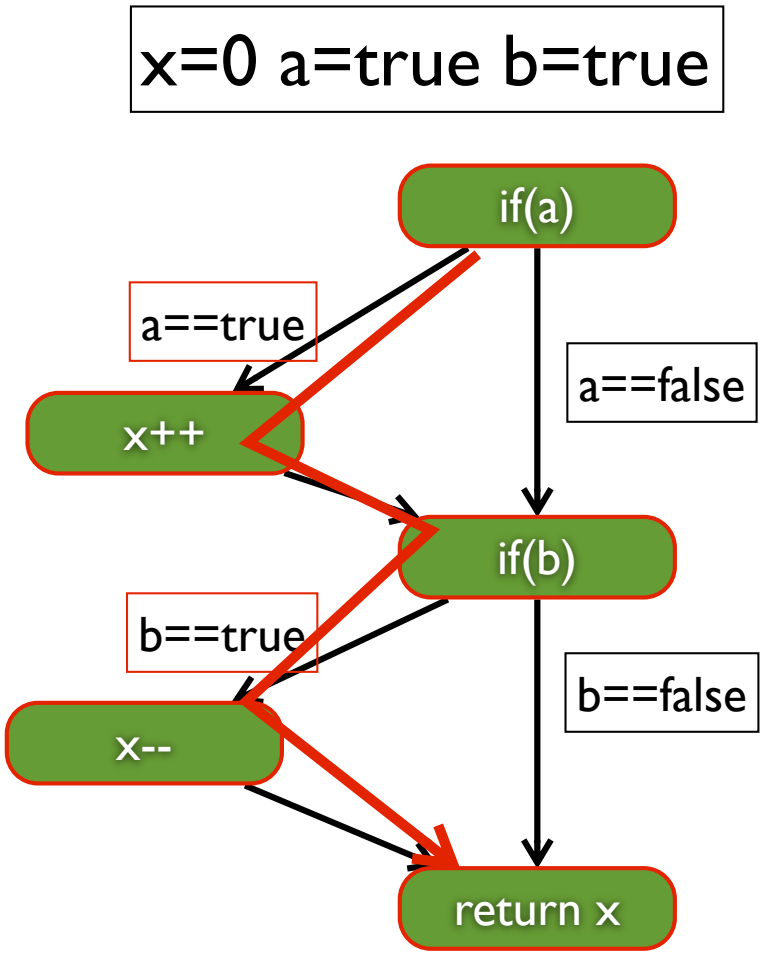
- If a test suite achieve 100% b-coverage, it must achieve 100% s-coverage
  - The statements not in branches will be covered by any test
  - All other statements are in certain branch
- If a test suite achieve 100% s-coverage, will it achieve 100% b-coverage?



Statement coverage VS. branch coverage

- If a test suite achieve 100% b-coverage, it must achieve 100% s-coverage
  - The statements not in branches will be covered by any test
  - All other statements are in certain branch
- If a test suite achieve 100% s-coverage, will it achieve 100% b-coverage?

Branch coverage strictly subsumes statement coverage

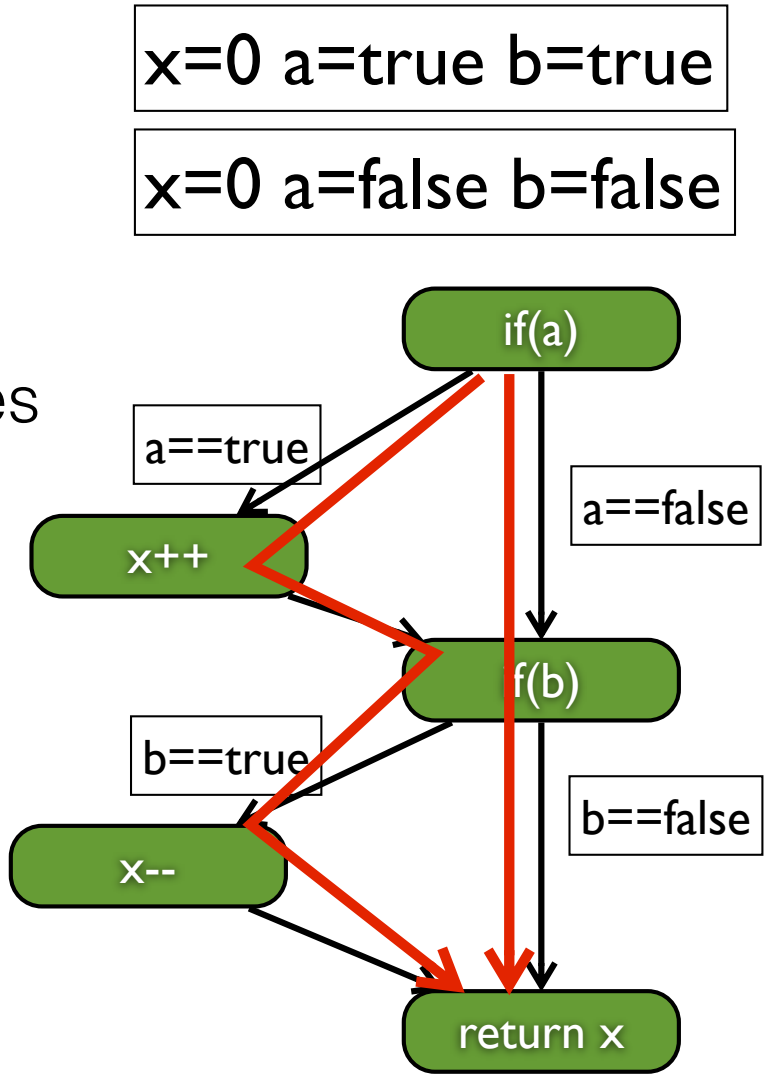


Branch coverage VS. path coverage

- If a test suite achieve 100% p-coverage, it must achieve 100% b-coverage
  - All the branch combinations have been covered indicate all branches are covered
- If a test suite achieve 100% b-coverage, will it achieve 100% p-coverage?

Branch coverage VS. path coverage

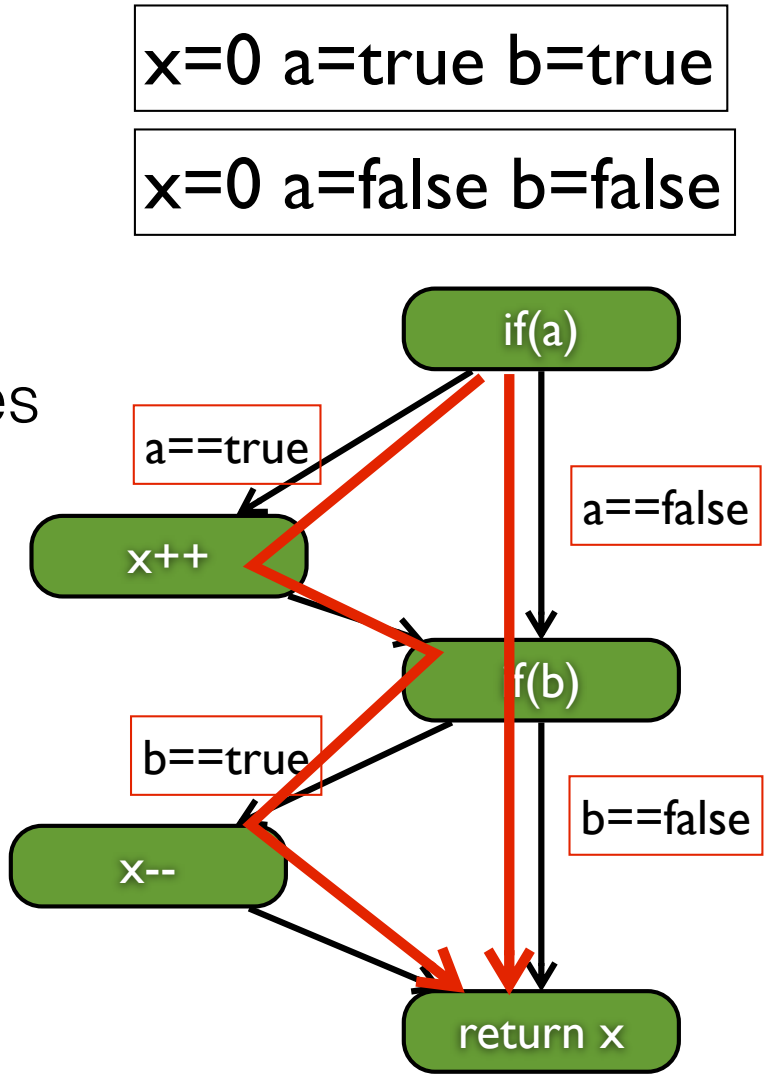
- If a test suite achieve 100% p-coverage, it must achieve 100% b-coverage
  - All the branch combinations have been covered indicate all branches are covered
- If a test suite achieve 100% b-coverage, will it achieve 100% p-coverage?



Branch coverage VS. path coverage

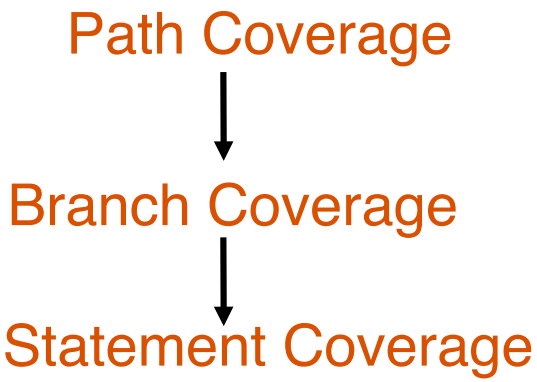
- If a test suite achieve 100% p-coverage, it must achieve 100% b-coverage
  - All the branch combinations have been covered indicate all branches are covered
- If a test suite achieve 100% b-coverage, will it achieve 100% p-coverage?

Path coverage strictly subsumes branch coverage



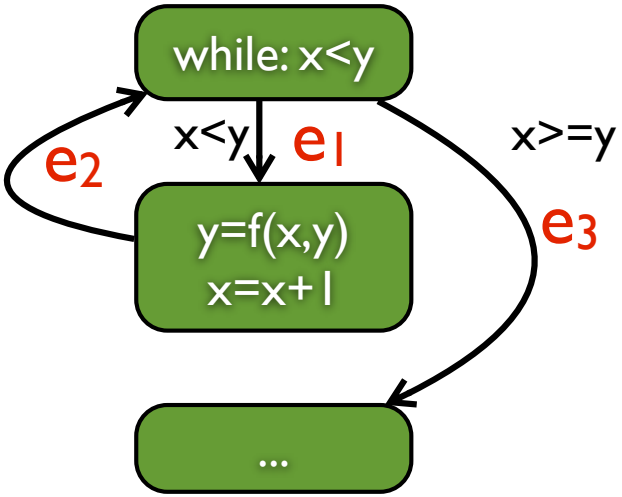
CFG-based coverage: comparison summary

Path coverage
strictly subsumes branch coverage
strictly subsumes statement coverage



Should we just use path coverage?

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



Possible Paths

e_3

$e_1e_2e_3$

$e_1e_2e_1e_2e_3$

$e_1e_2e_1e_2e_1e_2e_3$

...

Path coverage can be infeasible for real-world programs

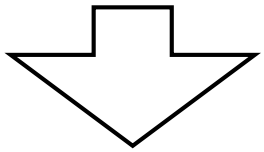
CFG-based coverage: effectiveness

- About 65% of all bugs can be caught in unit testing
- Unit testing is dominated by control-flow testing methods
- Statement and branch testing dominates control-flow testing

CFG-based coverage: limitation

- 100% coverage of some aspect is never a guarantee of bug-free software

Test: assertEquals(1, sum(1,0))



```
public int sum(int x, int y){
        return x-y; //should be x+y
    }
```

Statement coverage: 100%

Branch coverage: 100%

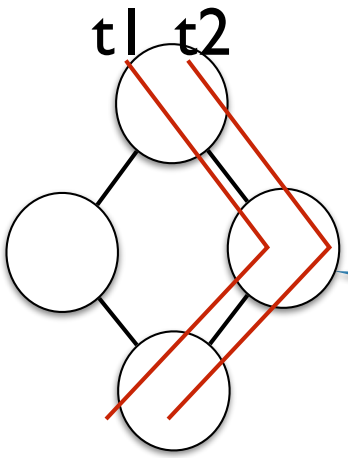
Path coverage: 100%

Failed to detect the bug...

Code coverage

- Data-flow coverage
  - All-Defs
  - All-Uses
  - All-DU-Paths
  - All-P-Uses/Some-C-Uses
  - All-C-Uses/Some-P-Uses
  - All-P-Uses
  - All-C-Uses

Motivation



Control-flow graph

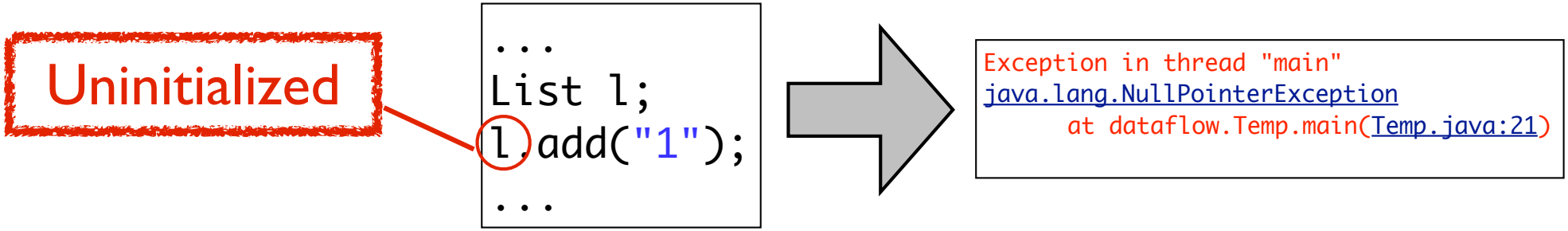
Are t1 and t2 identical?

although the paths are the same, different tests may have different variable values defined/used

- Basic idea:
  - Existing control-flow coverage criteria only consider the execution path (structure)
  - In the program paths, which variables are defined and then used should also be covered (data)
- A family of dataflow criteria is then defined, each providing a different degree of data coverage

Dataflow coverage

- Considers how data gets accessed and modified in the system and how it can get corrupted
- Common access-related bugs
  - Using an undefined or uninitialized variable
  - Deallocating or reinitializing a variable before it is constructed, initialized, or used
  - Deleting a collection object leaving its members unaccessible (garbage collection helps here)



Variable definition

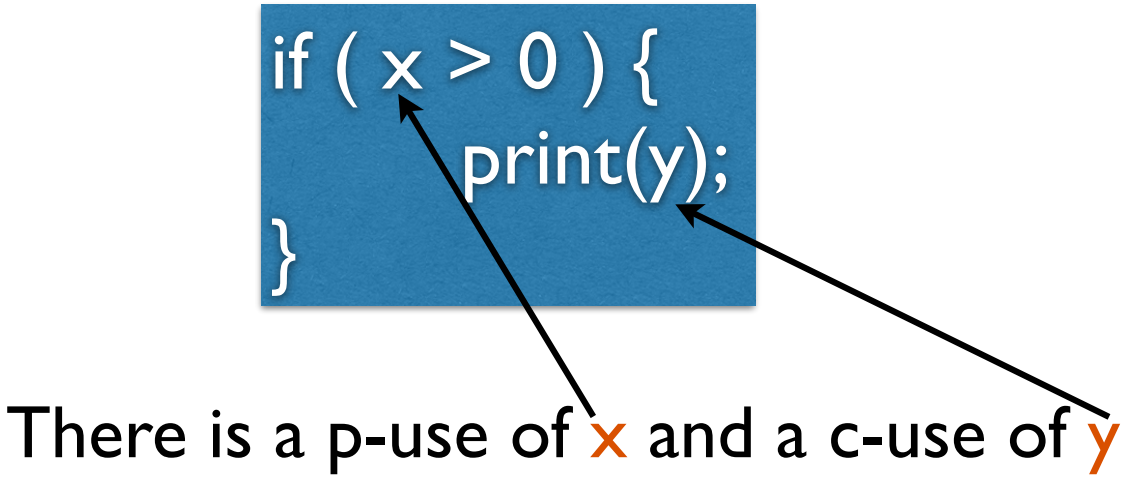
- A program variable is DEFINED whenever its value is modified:
  - on the *left* hand side of an assignment statement
    - *e.g.,* **y** = 17
  - in an input statement
    - *e.g.,* read(**y**)
  - as an call-by-reference parameter in a subroutine call
    - *e.g.,* update(x, &**y**);

Variable use

- A program variable is USED whenever its value is read:
  - on the right hand side of an assignment statement
    - *e.g.,* y = **x**+17
  - as an call-by-value parameter in a subroutine or function call
    - *e.g.,* y = sqrt(**x**)
  - in the predicate of a branch statement
    - *e.g.,* if (**x** > 0) { … }

Variable use: p-use and c-use

- Use in the predicate of a branch statement is a predicate-use or "p-use"
- Any other use is a computation-use or "c-use"
- For example, in the program fragment:

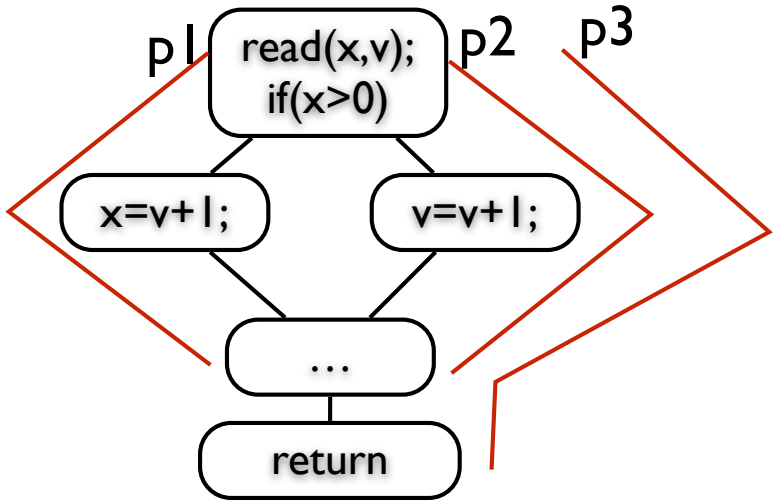


Variable use

- A variable can also be used and then re-defined in a single statement when it appears:
  - on both sides of an assignment statement
    - *e.g.,* **y** = **y** + x
  - as an call-by-reference parameter in a subroutine call
    - *e.g.,* increment(&**y**)

More dataflow terms and definitions

- A path is **definition clear** ("def-clear") with respect to a variable **v** if it has no variable re-definition of **v** on the path
- A **complete path** is a path whose initial node is a entry node and whose final node is an exit node



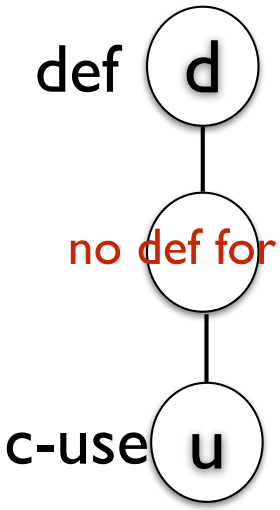
Control-flow graph

p1 is def-clear for **v**, while p2 is not

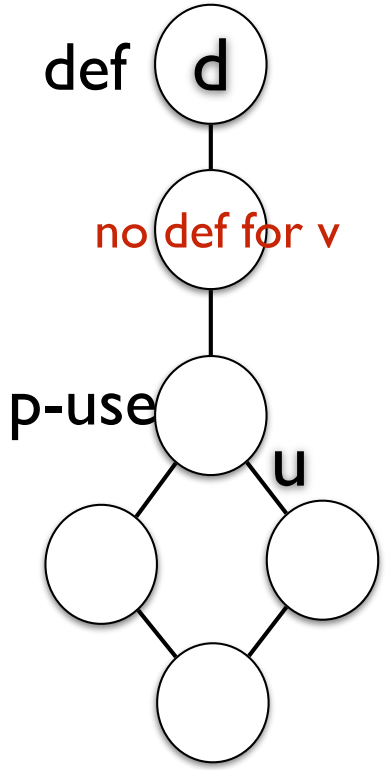
p1, p2 are not complete, while p3 is

Definition-use pair (du-pair)

- A **definition-use pair** ("du-pair") with respect to a variable **v** is a pair (**d**,**u**) such that
  - **d** is a node defining **v**
  - **u** is a node or edge using **v**
    - when it is a p-use of **v**, **u** is an outgoing edge of the predicate statement
  - there is a def-clear path with respect to **v** from **d** to **u**



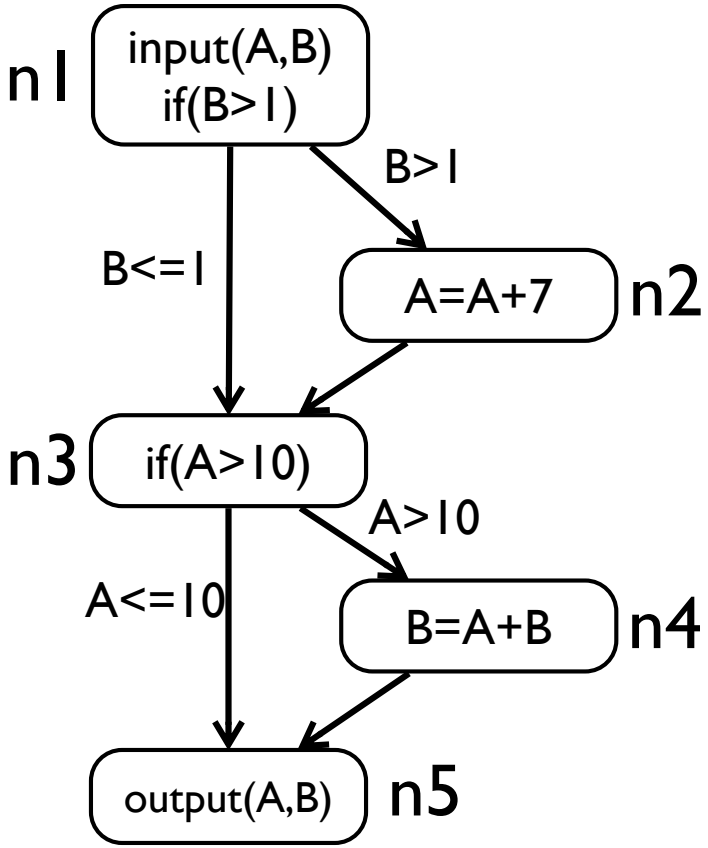
c-use example



p-use example

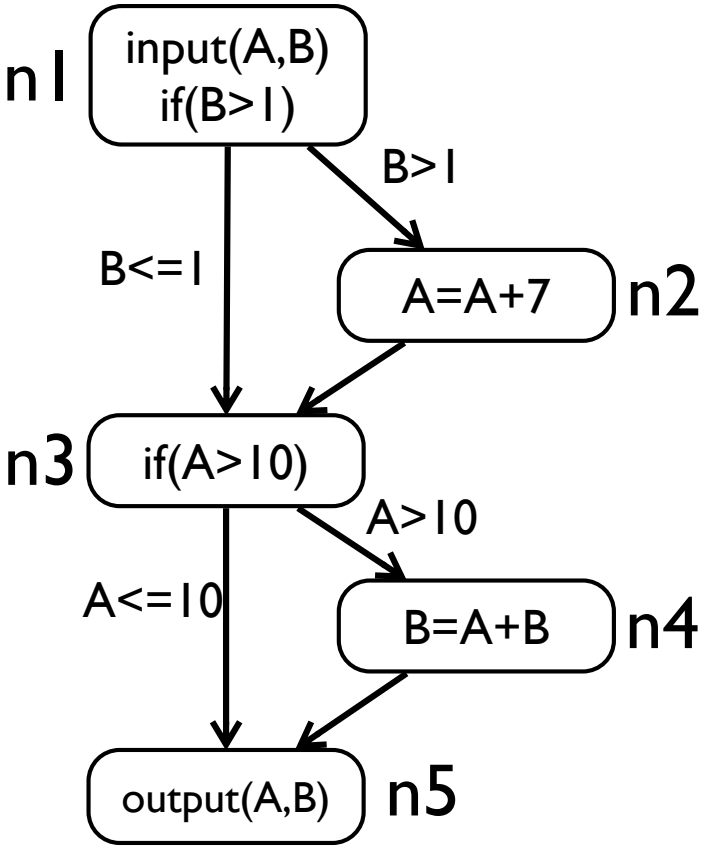
Du-pair: example 1

```
1. input(A,B)
    if (B>1) {
2.     A = A+7
    }
3. if (A>10) {
4.     B = A+B
    }
5. output(A,B)
```



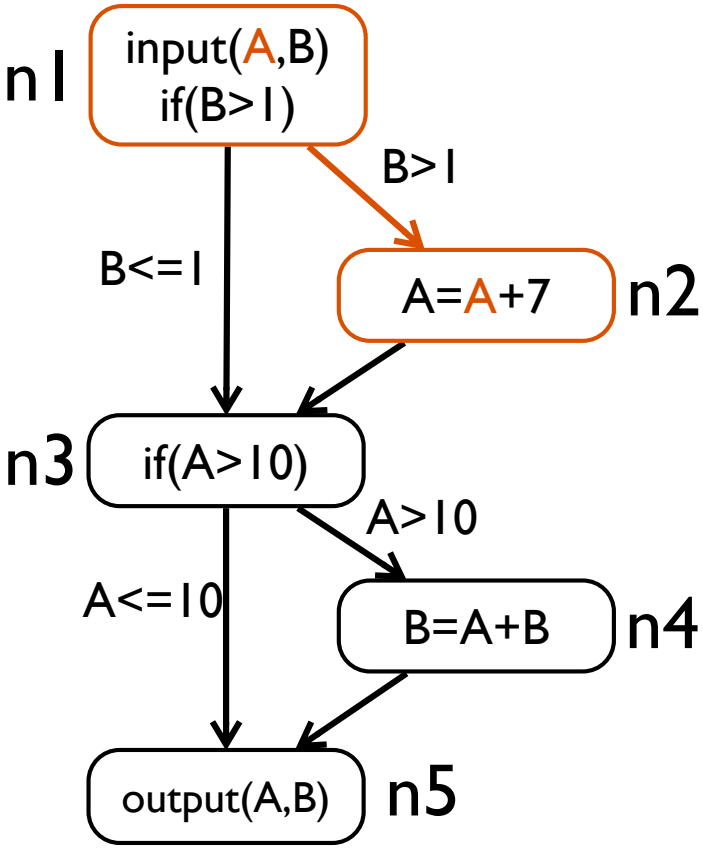
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



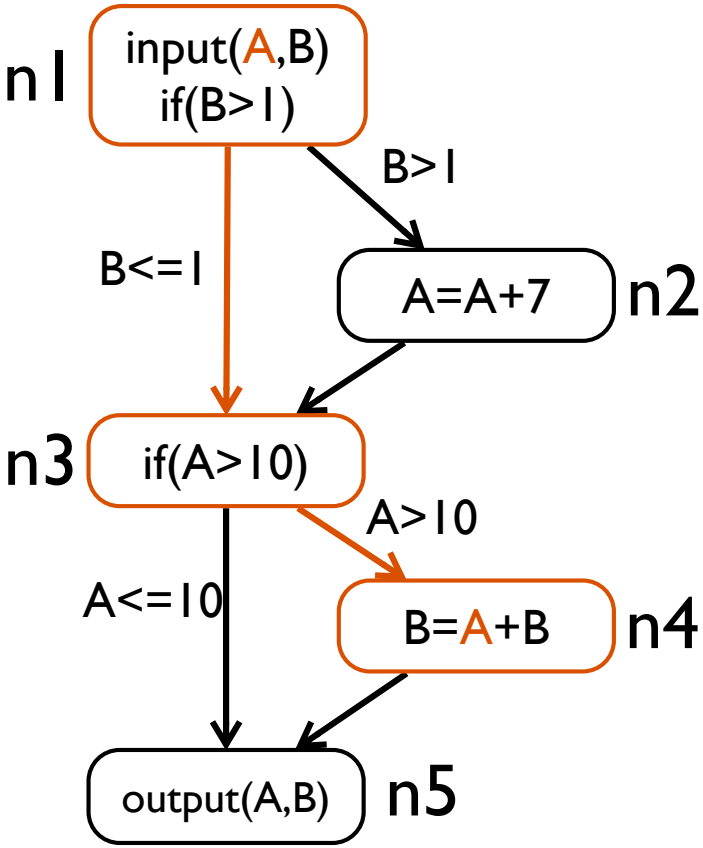
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



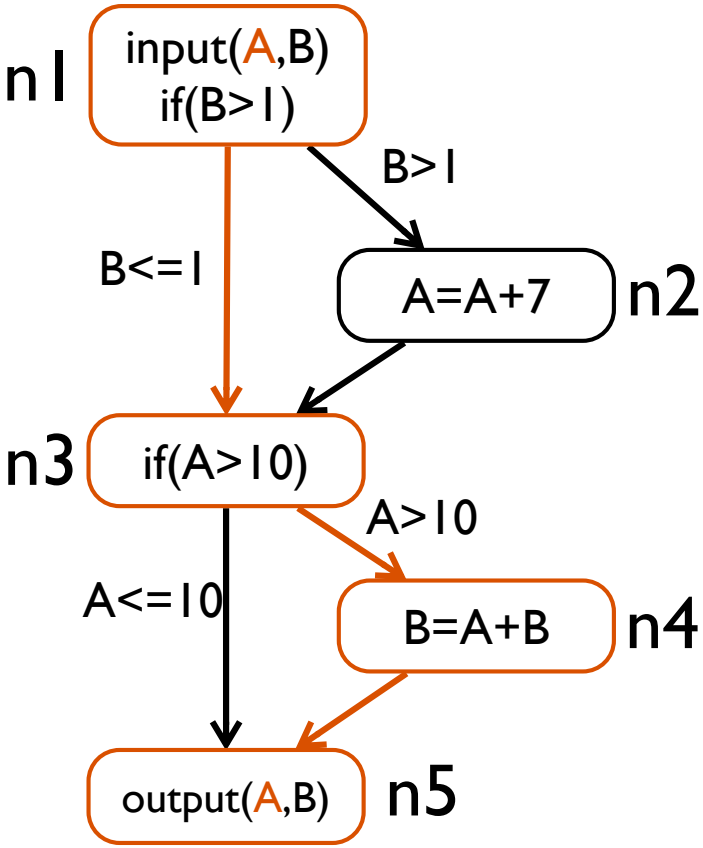
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	**<1,3,4>**
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



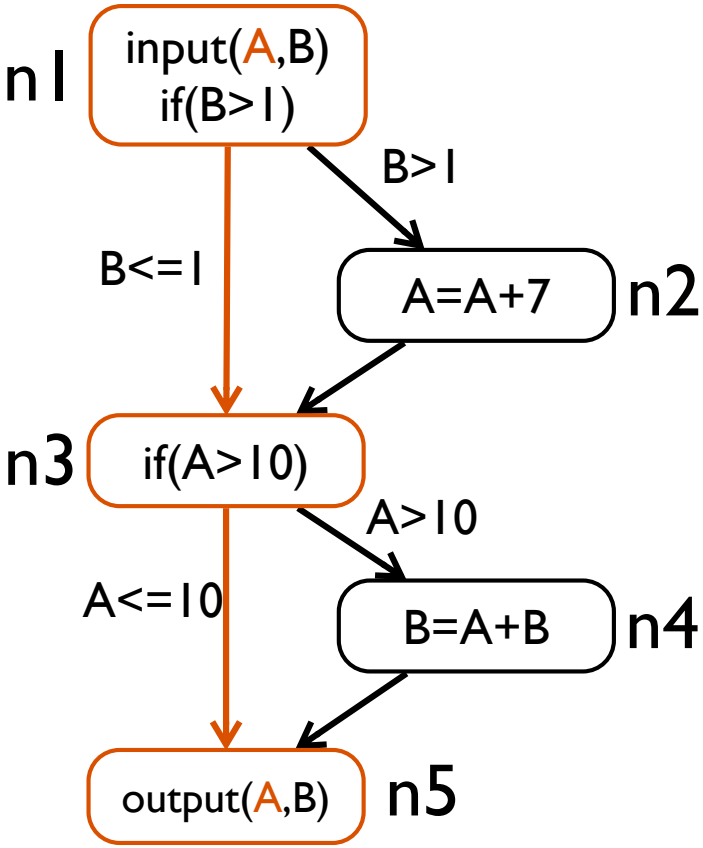
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	**<1,3,4,5>**
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



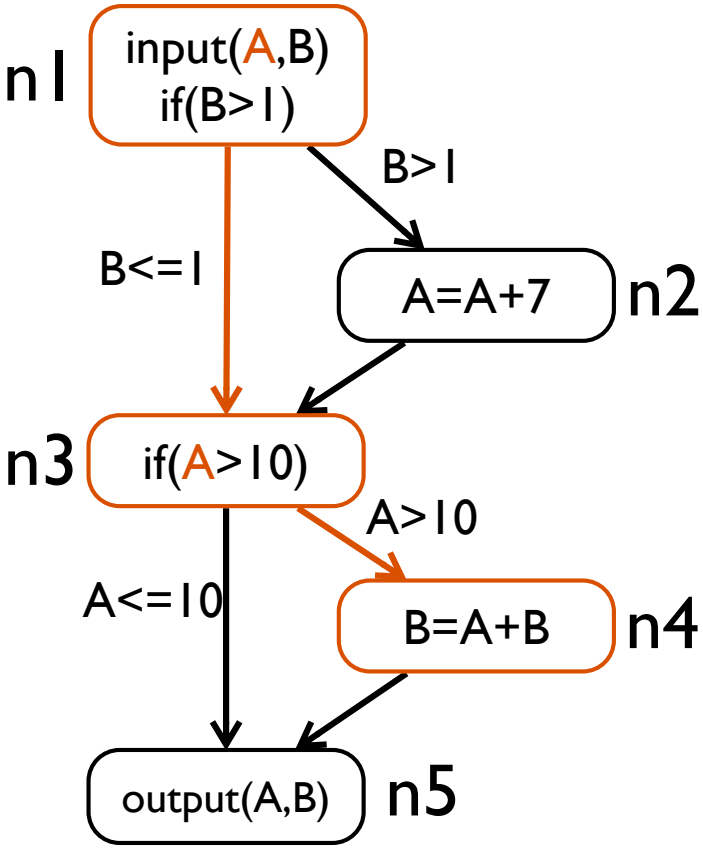
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



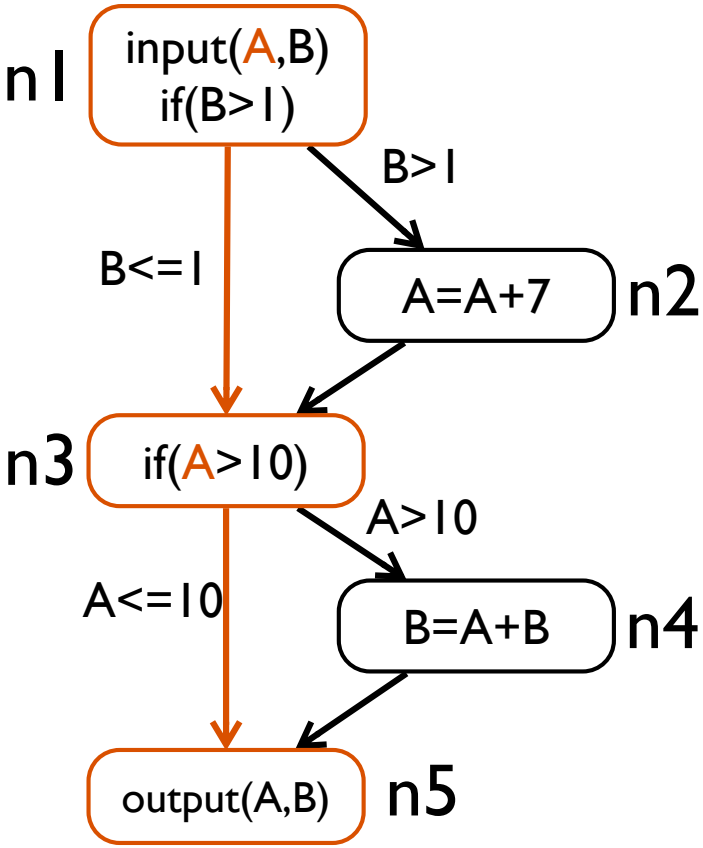
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	**<1,3,4>**
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



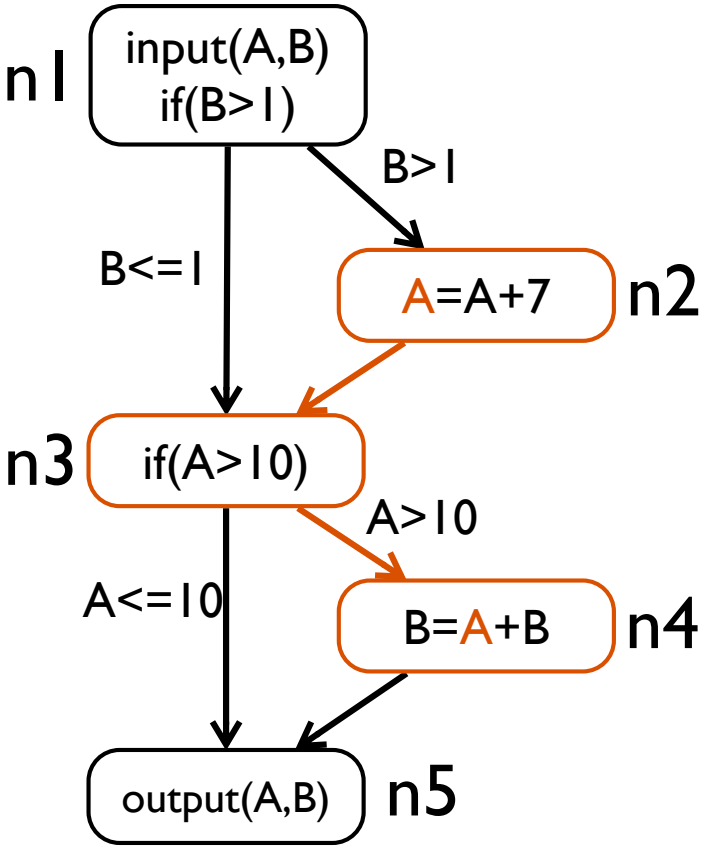
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	**<1,3,5>**
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



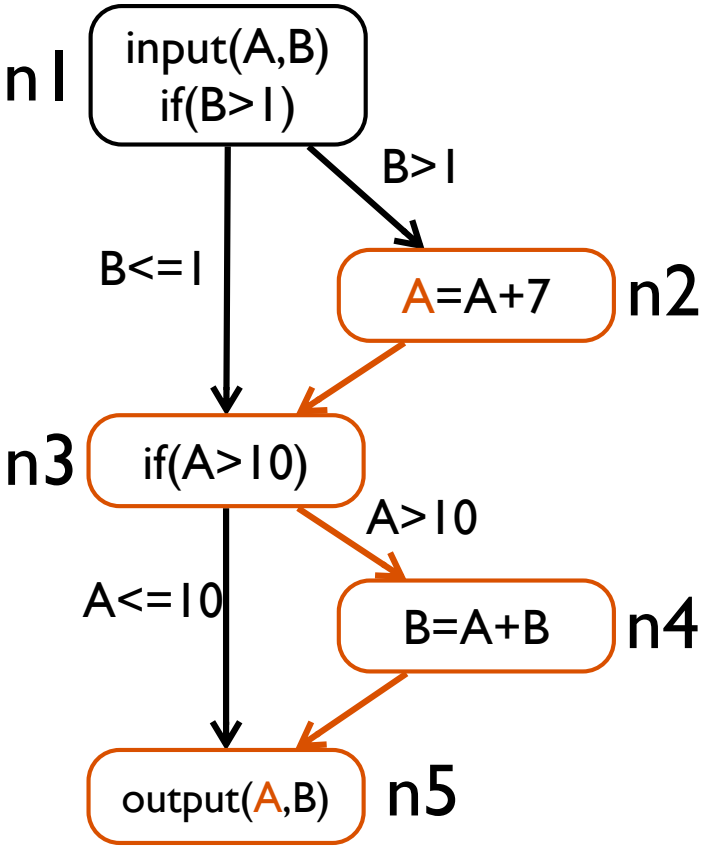
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	**<2,3,4>**
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



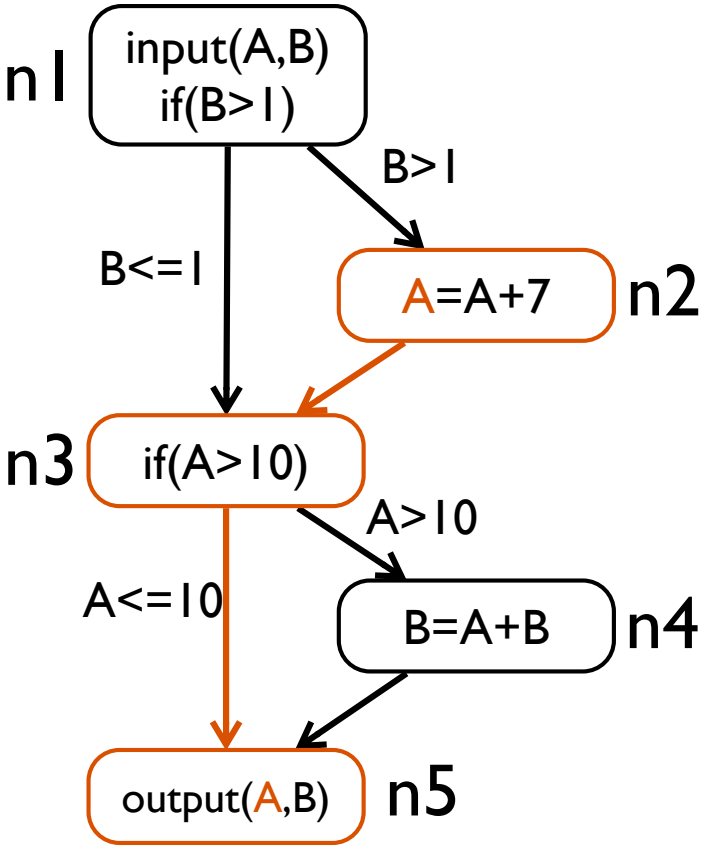
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	**<2,3,4,5>**
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



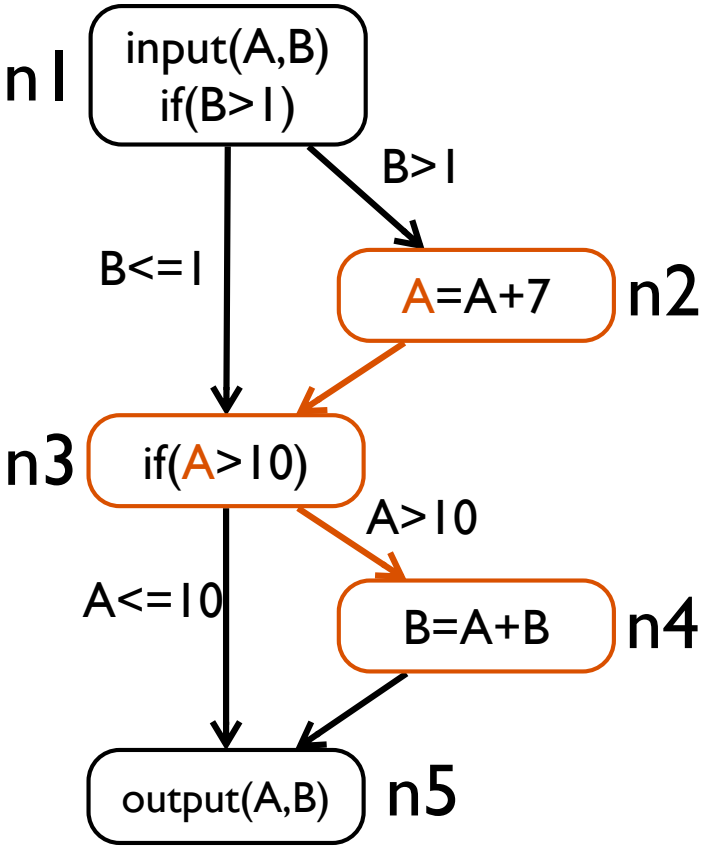
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



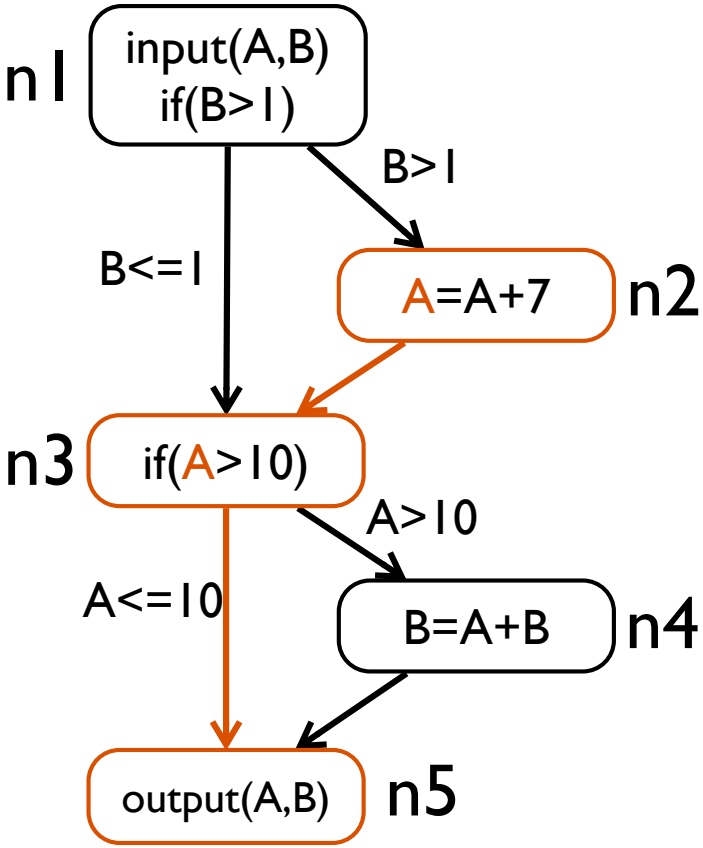
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	**<2,3,4>**
(2,<3,5>)	<2,3,5>



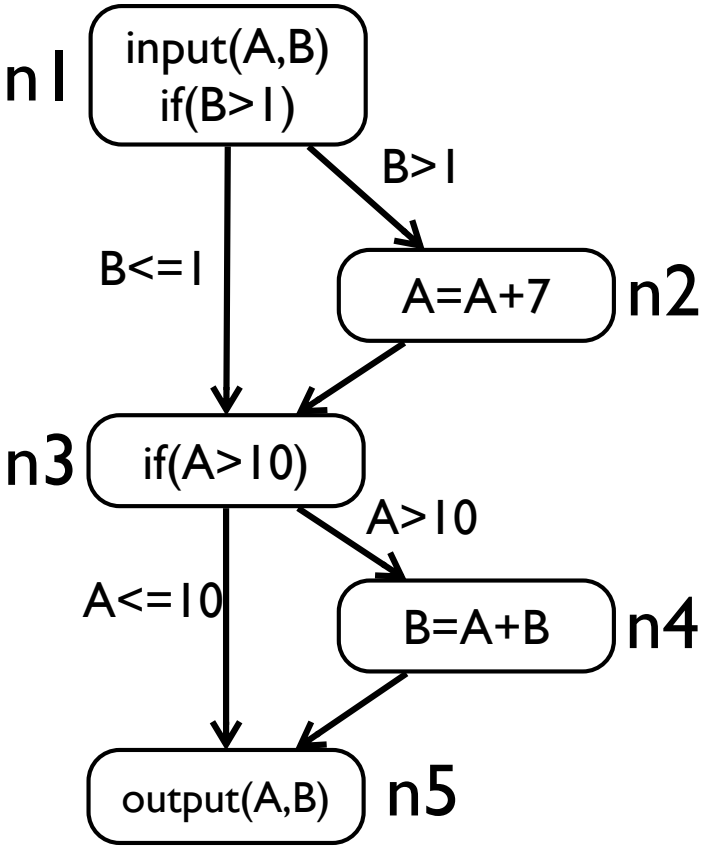
Identifying du-pairs – variable **A**

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	**<2,3,5>**



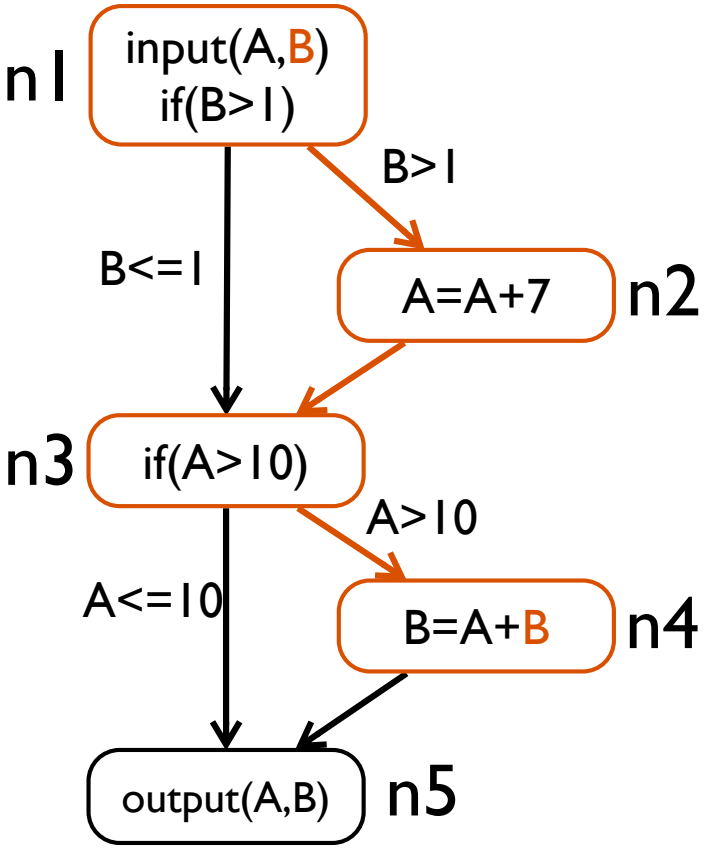
Identifying du-pairs – variable **B**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2>
(1,<1,3>)	<1,3>
(4,5)	<4,5>



Identifying du-pairs – variable **B**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2>
(1,<1,3>)	<1,3>
(4,5)	<4,5>



Dataflow test coverage criteria

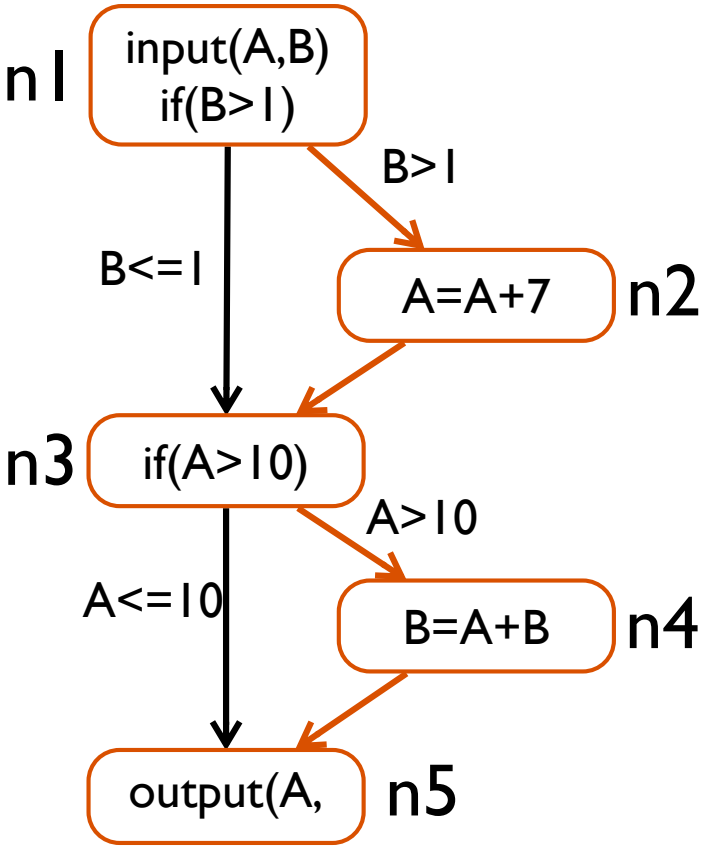
- **All-Defs**
  - for every program variable **v**, at least one def-clear path from every definition of **v** to at least one c-use or one p-use of **v** must be covered

Dataflow test coverage criteria

- Consider a test case executing path:
  - t1: <1,2,3,4,5>
- Identify all def-clear paths covered (i.e., subsumed) by this path for each variable
- Are all definitions for each variable associated with at least one of the subsumed def-clear paths?

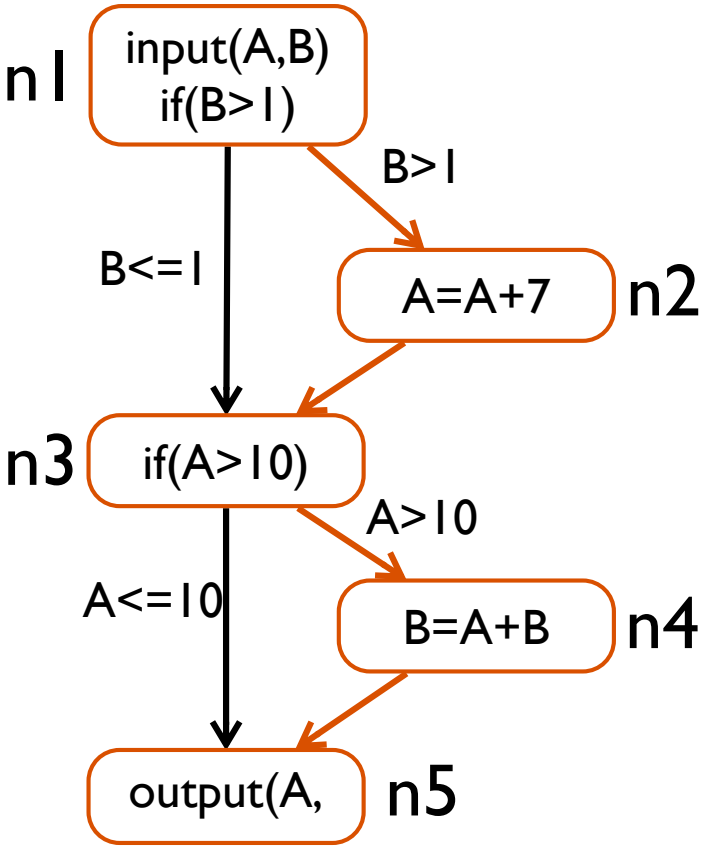
Def-clear paths subsumed by <1,2,3,4,5> for variable **A**

du-pair	path(s)
(1,2)	<1,2> ✔
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> ✔
(2,5)	<2,3,4,5> ✔
	<2,3,5>
(2,<3,4>)	<2,3,4> ✔
(2,<3,5>)	<2,3,5>



Def-clear paths subsumed by <1,2,3,4,5> for variable **B**

du-pair	path(s)
(1,4)	<1,2,3,4> ✓
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2> ✓
(1,<1,3>)	<1,3>
(4,5)	<4,5> ✓



Dataflow test coverage criteria

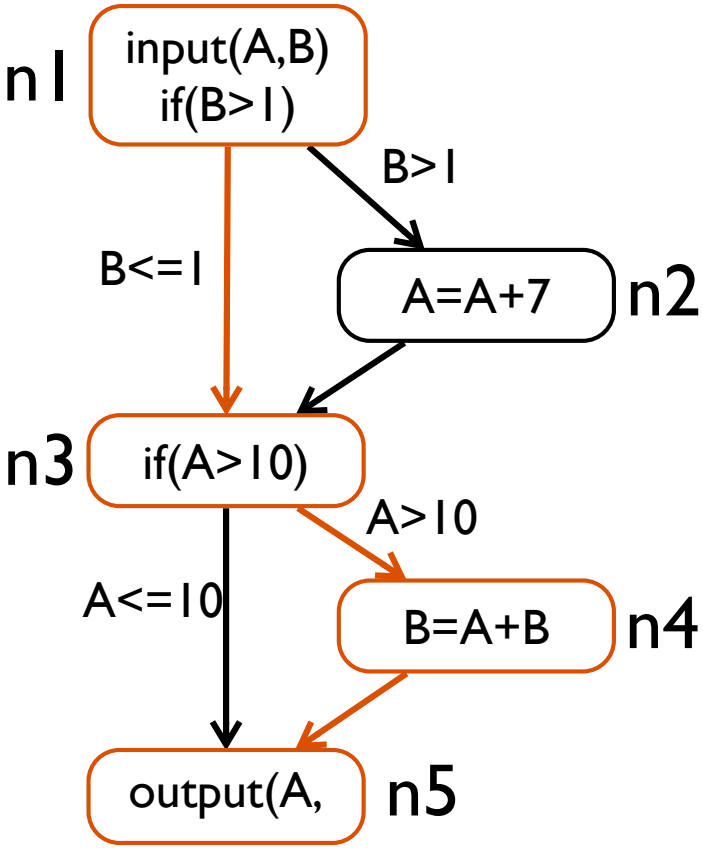
- Since <1,2,3,4,5> covers at least one def-clear path from every definition of **A** or **B** to at least one c-use or p-use of **A** or **B**, All-Defs coverage is achieved

Dataflow test coverage criteria

- **All-Uses**:
  - for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use and every p-use (including all outgoing edges of the predicate statement) of **v** must be covered
  - Requires that all du-pairs covered
- Consider additional test cases executing paths:
  - t2: <1,3,4,5>
  - t3: <1,2,3,5>
- Do all three test cases provide All-Uses coverage?

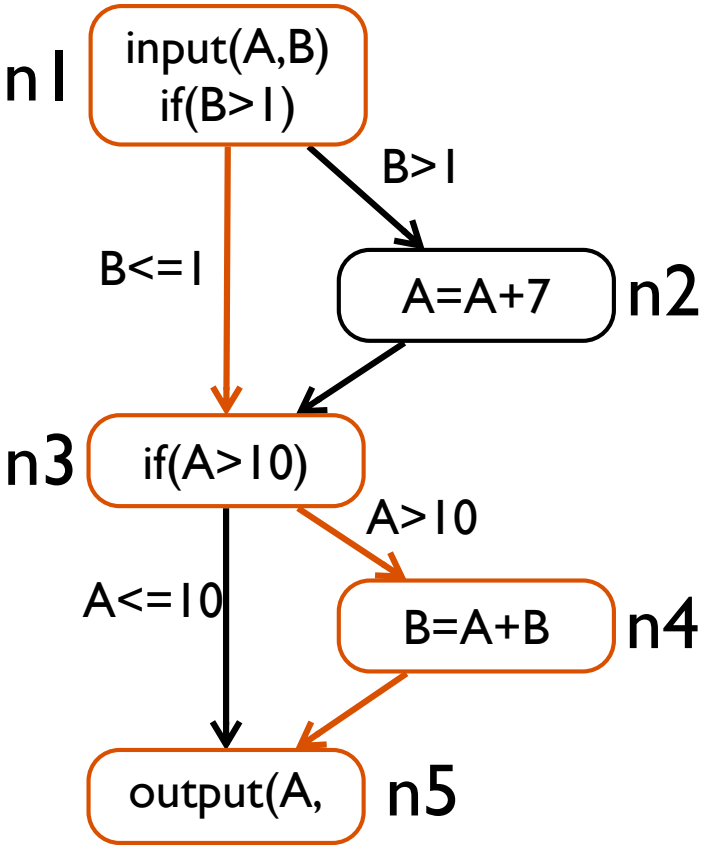
Def-clear paths subsumed by <1,3,4,5> for variable **A**

du-pair	path(s)
(1,2)	<1,2> ✔
(1,4)	<1,3,4> ✔
(1,5)	<1,3,4,5> ✔
	<1,3,5>
(1,<3,4>)	<1,3,4> ✔
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> ✔
(2,5)	<2,3,4,5> ✔
	<2,3,5>
(2,<3,4>)	<2,3,4> ✔
(2,<3,5>)	<2,3,5>



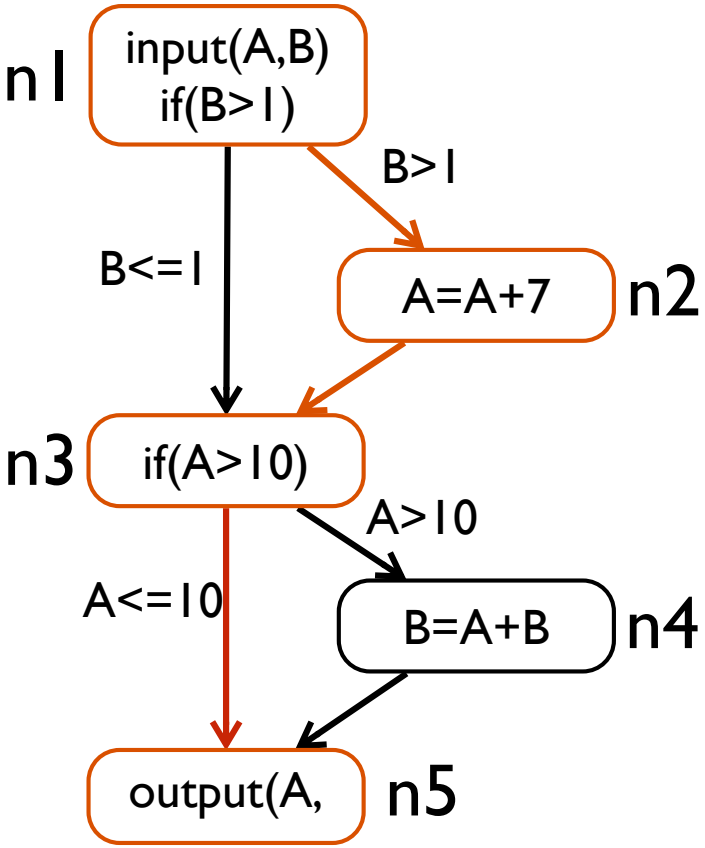
Def-clear paths subsumed by <1,3,4,5> for variable **B**

du-pair	path(s)
(1,4)	<1,2,3,4> ✓
	<1,3,4> ✓
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2> ✓
(1,<1,3>)	<1,3> ✓
(4,5)	<4,5> ✓ ✓



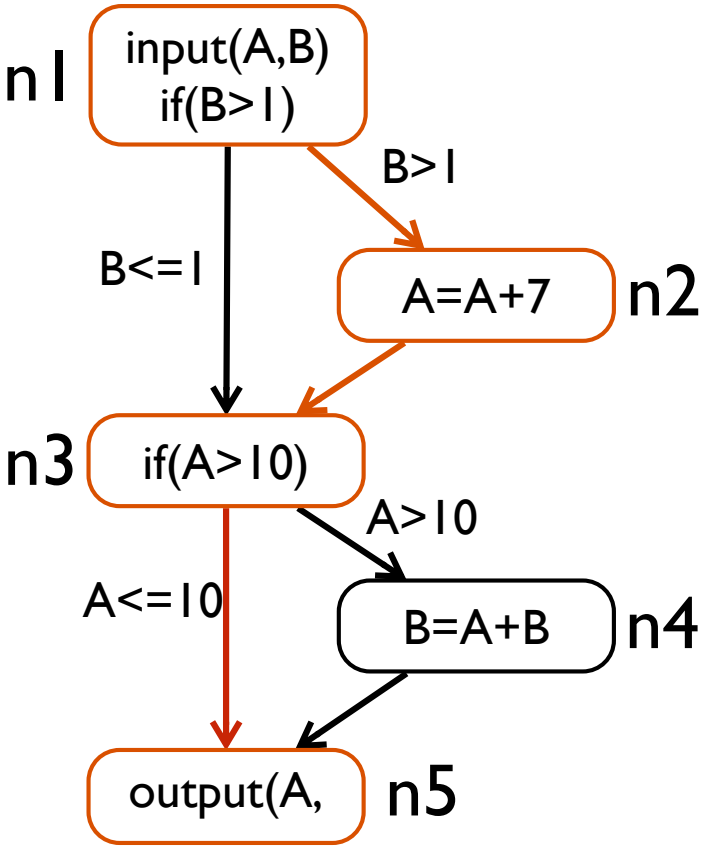
Def-clear paths subsumed by <1,2,3,5> for variable **A**

du-pair	path(s)
(1,2)	<1,2> ✓ ✓
(1,4)	<1,3,4> ✓
(1,5)	<1,3,4,5> ✓
	<1,3,5>
(1,<3,4>)	<1,3,4> ✓
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> ✓
(2,5)	<2,3,4,5> ✓
	<2,3,5> ✓
(2,<3,4>)	<2,3,4> ✓
(2,<3,5>)	<2,3,5> ✓



Def-clear paths subsumed by <1,2,3,5> for variable **B**

du-pair	path(s)
(1,4)	<1,2,3,4> ✓
	<1,3,4> ✓
(1,5)	<1,2,3,5> ✓
	<1,3,5>
(1,<1,2>)	<1,2> ✓ ✓
(1,<1,3>)	<1,3> ✓
(4,5)	<4,5> ✓ ✓

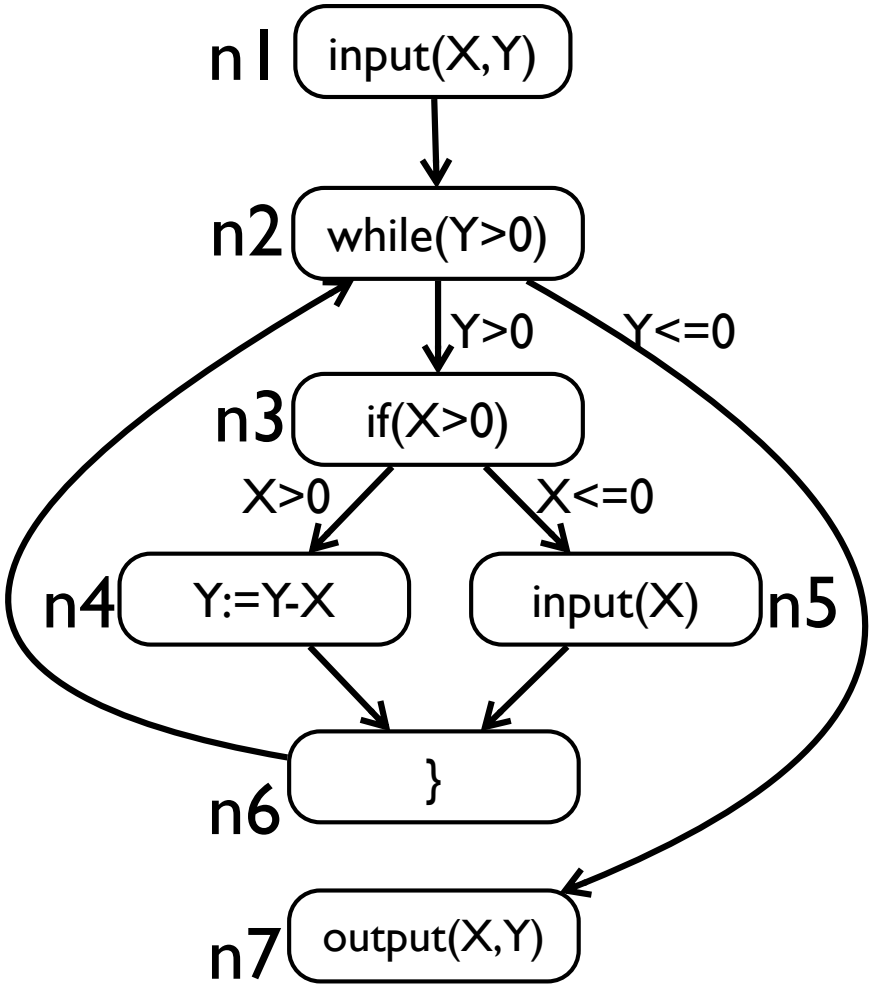


Dataflow test coverage criteria

- None of the three test cases covers the du-pair (1,<3,5>) for variable **A**,
- All-Uses Coverage is not achieved

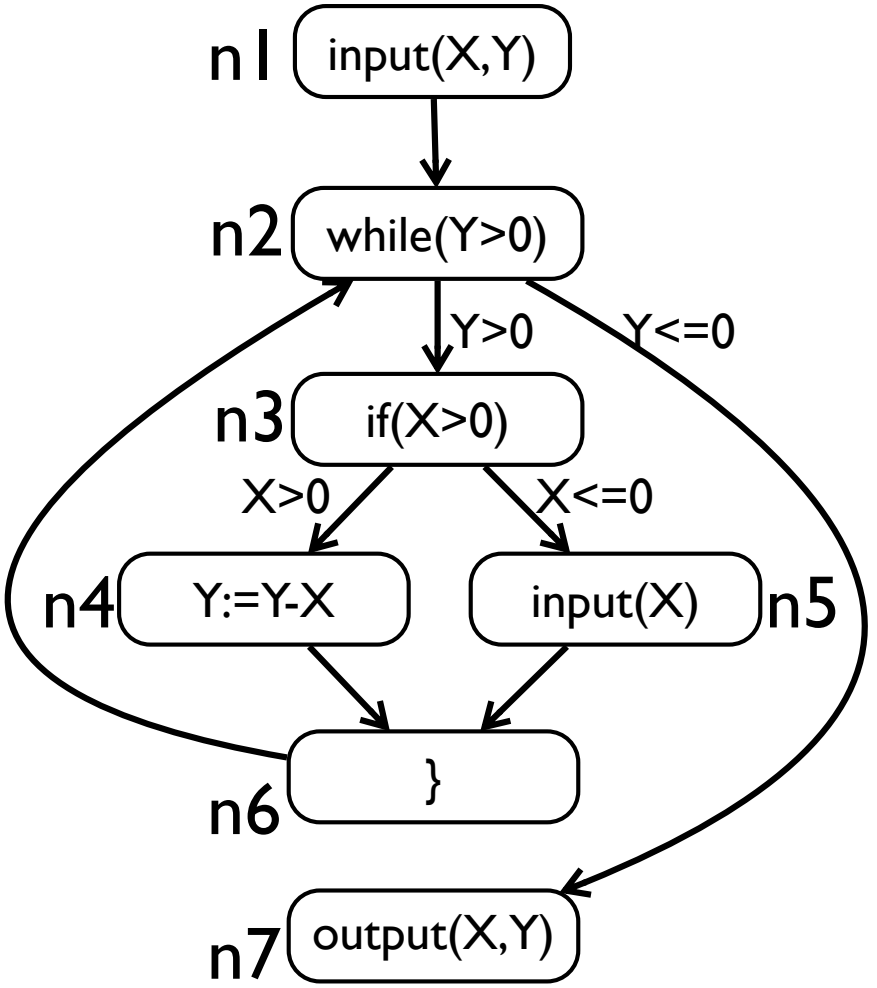
Du-pair: more complicated example

```
1. input(X,Y)
2. while (Y>0) {
3.     if (X>0)
4.         Y := Y-X
       else
5.         input(X)
6.     }
7. output(X,Y)
```



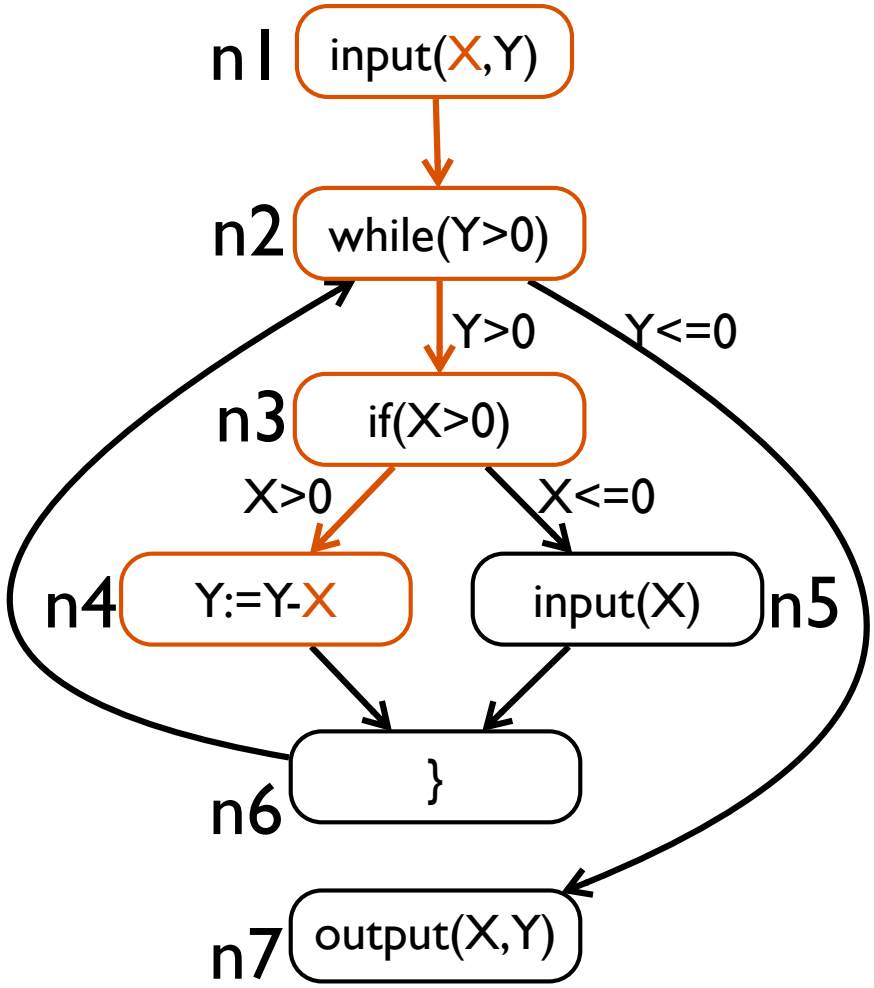
airs – variable **X**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



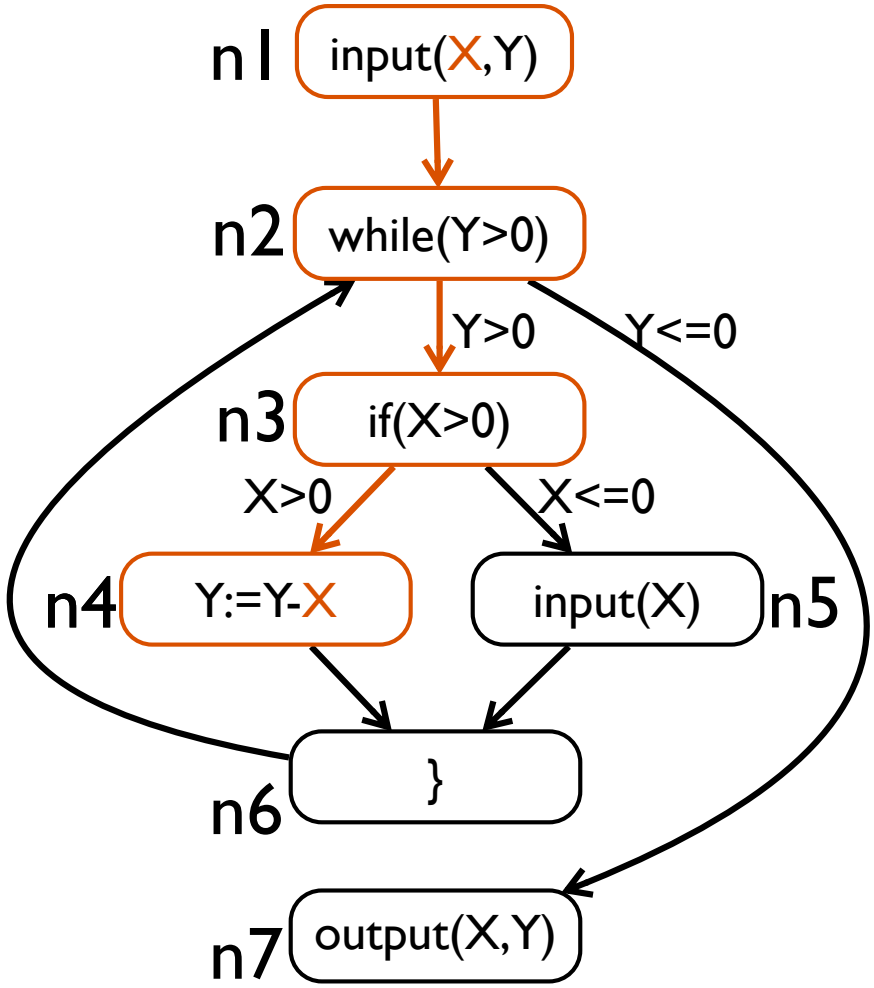
airs – variable **X**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



airs – variable **X**

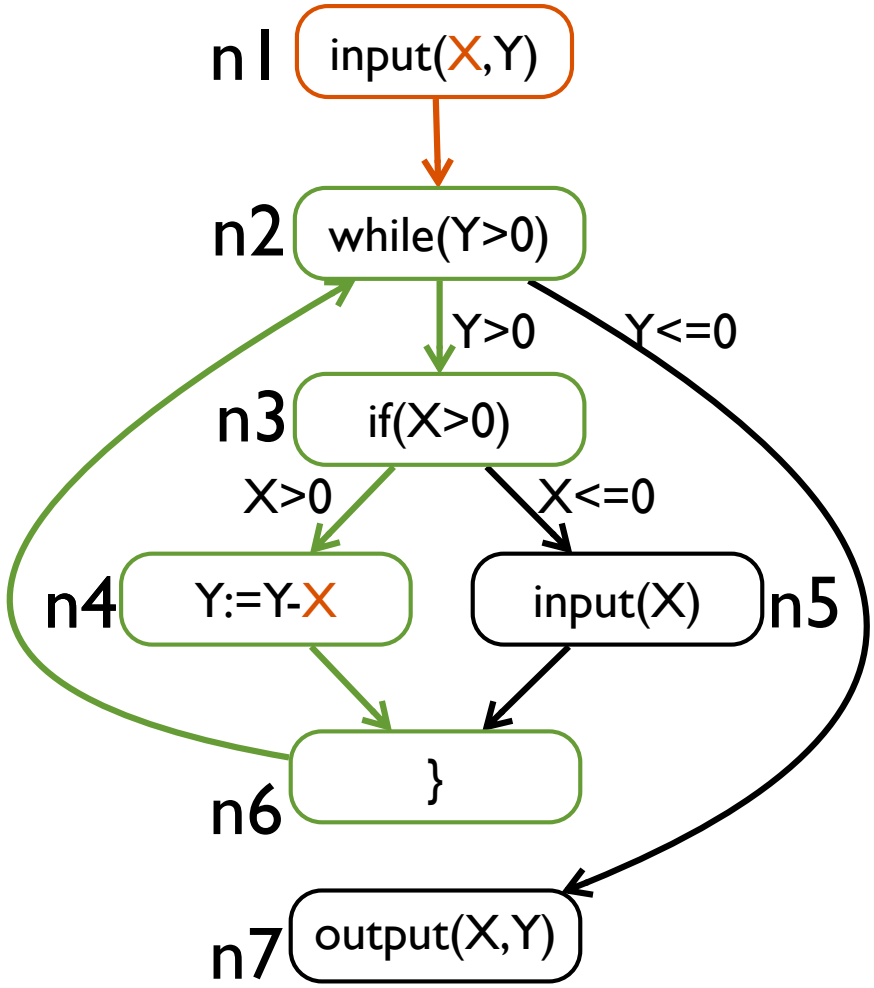
du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



(a)* means "a" will be repeated for >=1 times

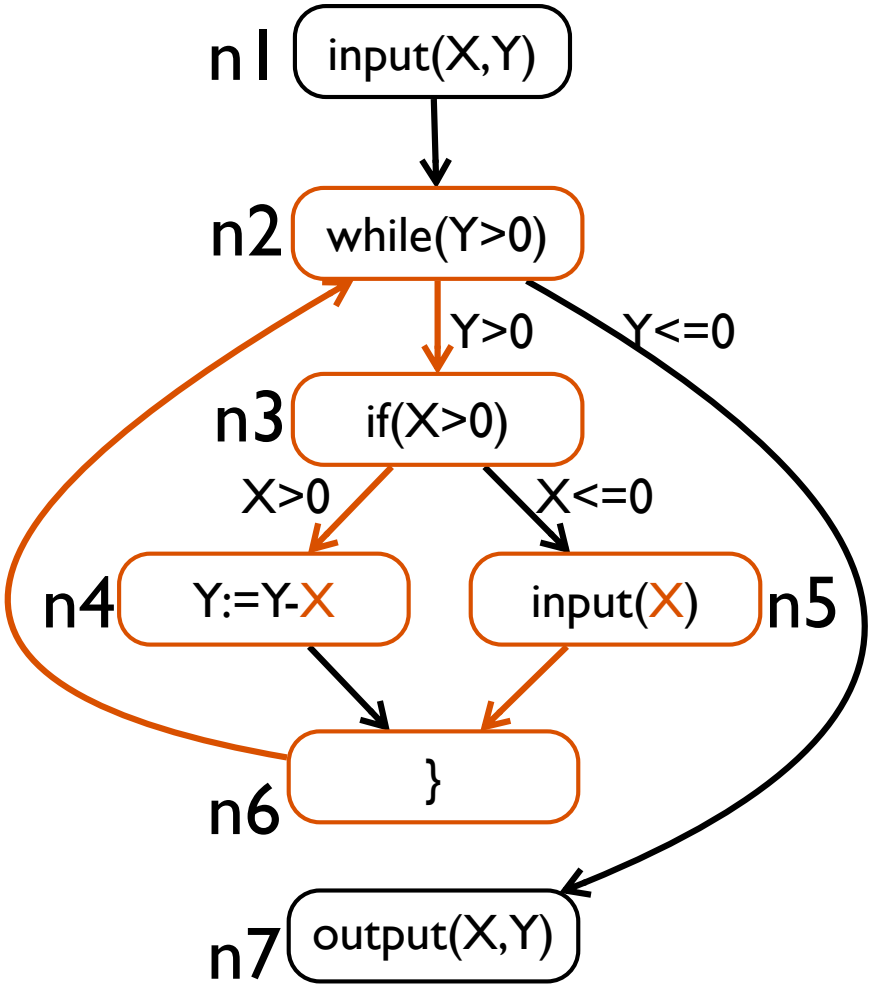
airs – variable **X**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



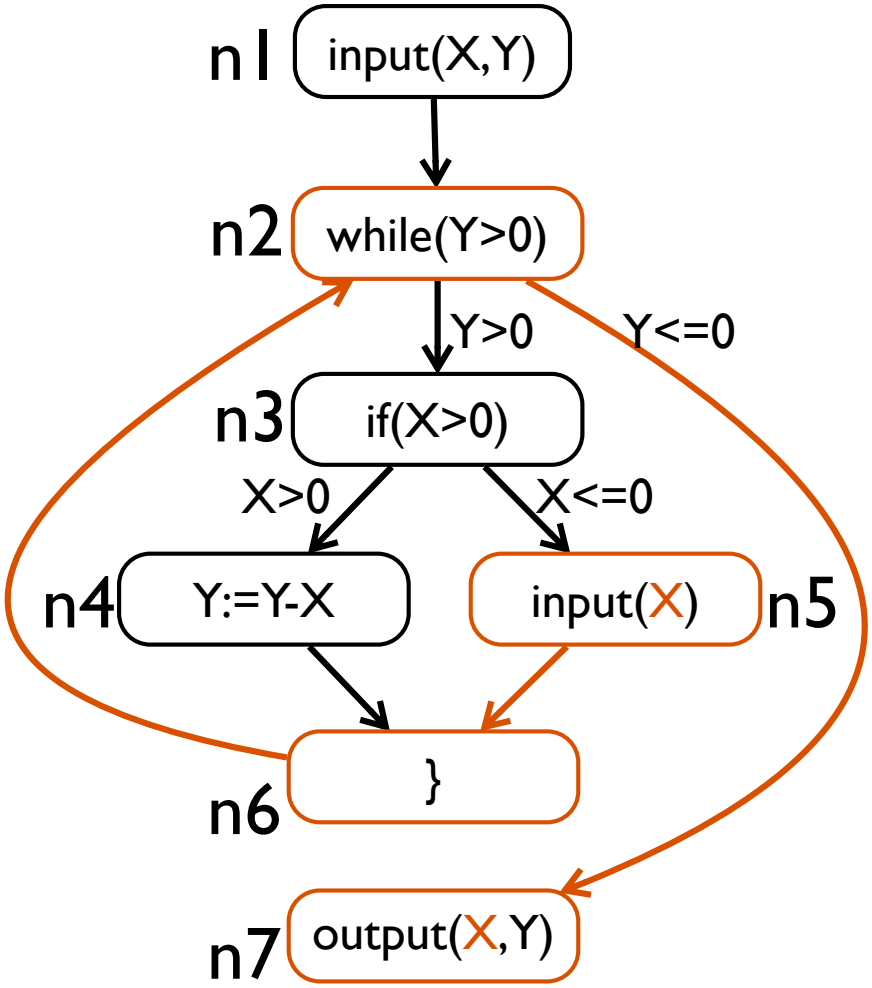
airs – variable **X**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	**<5,6,2,3,4>**
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



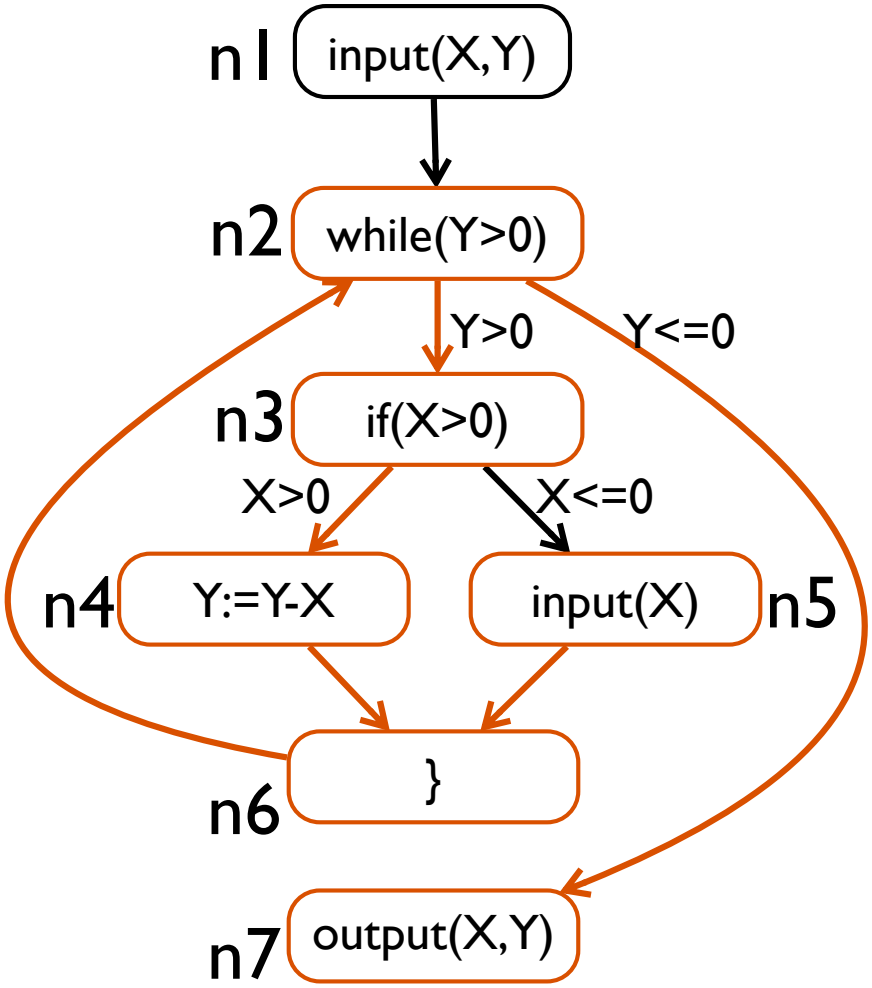
Note that the definition of a du-pair does not require the existence of a feasible def-clear path from **d** to **u**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	**<5,6,2,7>** Infeasible!
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



airs – variable **X**

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,<3,5>)	<5,6,2,3,5>
	<5,6,2,(3,4,6,2)*,3,5>



More dataflow terms and definitions

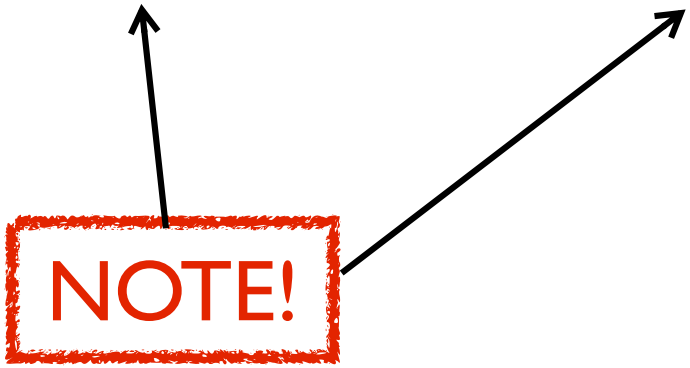
- A path (either partial or complete) is **simple**
  if all edges within the path are distinct, *i.e.,* different
- A path is **loop-free** if all nodes within the path are distinct,
  i.e., different

Simple and loop-free paths

path	Simple?	Loop-free?
<1,3,4,2>	✔	✔
<1,2,3,2>	✔	
<1,2,3,1,2>		
<1,2,3,2,4>	✔	

Du-path

- A path <n1,n2,...,nj,nk> is a **du-path** with respect to a variable **v,** if **v** is defined at node **n1** and either:
  - there is a c-use of **v** at node **nk** and <n1,n2,...,nj,nk> is a def-clear simple path, or
  - there is a p-use of **v** at edge <nj,nk> and <n1,n2,...nj> is a def-clear loop-free path.

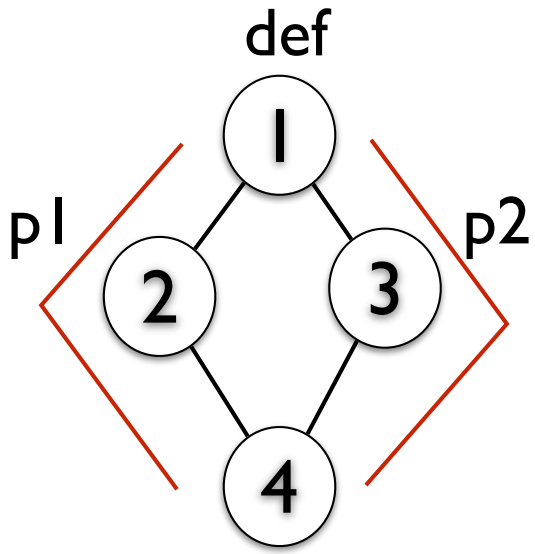


Identifying du-paths

du-pair	path(s)	du-path?
(5,4)	<5,6,2,3,4>	✓
	<5,6,2,3,4,(6,2,3,4)*>	
(5,7)	<5,6,2,7>	✓
	<5,6,2,(3,4,6,2)*,7>	
(5,<3,4>)	<5,6,2,3,4>	✓
	<5,6,2,3,4,(6,2,3,4)*>	
(5,<3,5>)	<5,6,2,3,5>	✓
	<5,6,2,(3,4,6,2)*,3,5>	

Another dataflow test coverage criterion

- **All-DU-Paths**:
  - for every program variable **v**, every du-path from every definition of **v** to every c-use and every p-use of **v** must be covered



node 1 is the only def node, and 4 is the only use node for **v**

p1 satisfies all-defs and all-uses, but not all-du-paths

p1 and p2 together satisfy all-du-paths

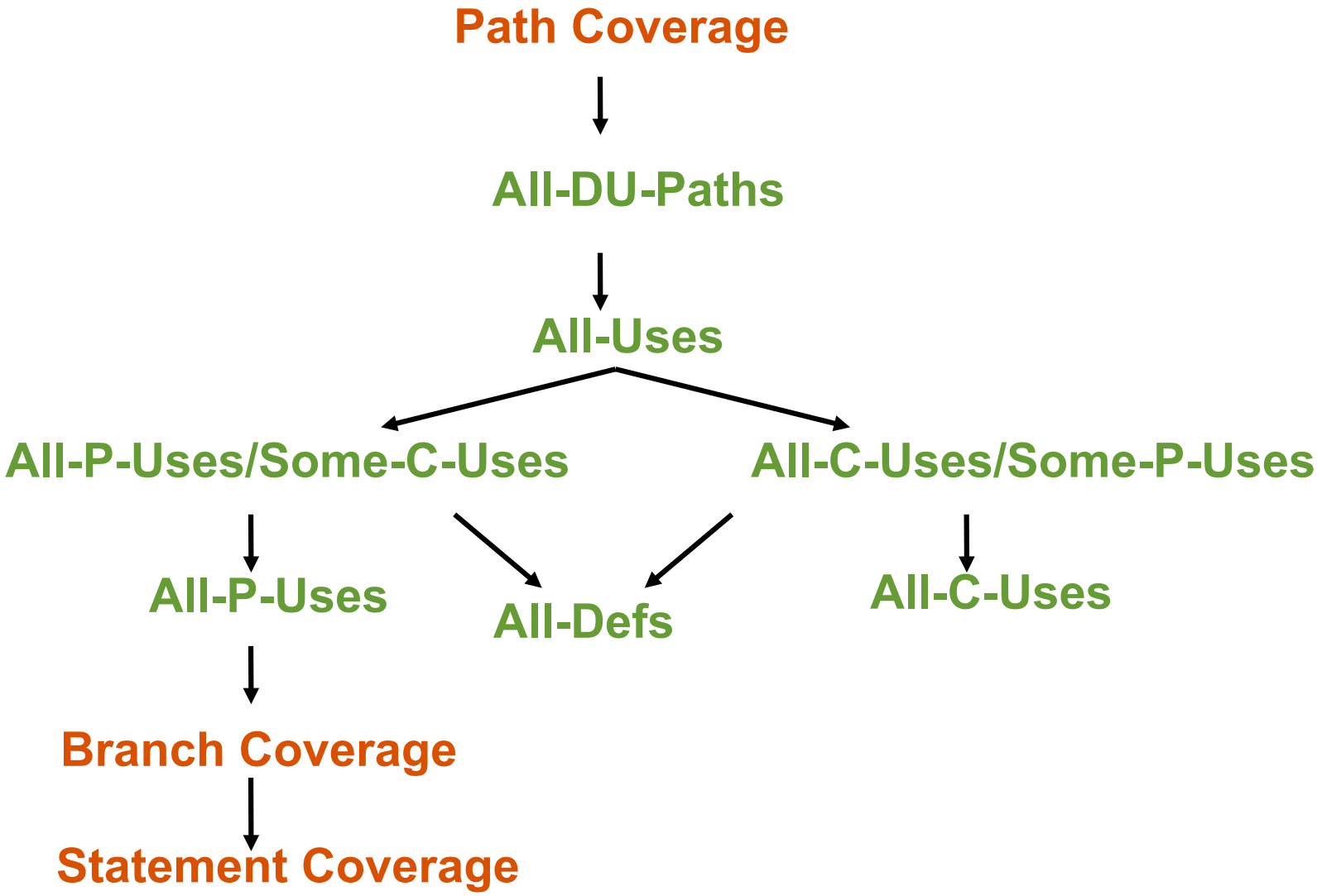
More dataflow test coverage criteria

- **All-P-Uses/Some-C-Uses**:
  for every program variable **v**, at least one def-clear path from every definition of **v** to every p-use of **v** must be covered
  - If no p-use of **v** is available, at least one def-clear path to a c-use of **v** must be covered
- **All-C-Uses/Some-P-Uses**:
  for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use of **v** must be covered
  - If no c-use of **v** is available, at least one def-clear path to a p-use of **v** must be covered

More dataflow test coverage criteria (2)

- **All-P-Uses**:
  for every program variable **v**, at least one def-clear path from every definition of **v** to every p-use of **v** must be covered
- **All-C-Uses**:
  for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use of **v** must be covered

Summary


Path Coverage
All-DU-Paths
All-Uses
All-P-Uses/Some-C-Uses
All-C-Uses/Some-P-Uses
All-P-Uses
All-Defs
All-C-Uses
Branch Coverage
Statement Coverage

Suggested readings

- Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, 11(4), April 1985, pp. 367-375. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1702019
- P. Frankl and E. Weyuker. An Applicable Family of Data Flow Testing Criteria. IEE Transaction on software eng., vol.14, no.10, October 1988.
- E. Weyuker. The evaluation of Program-based software test data adequacy criteria. Communication of the ACM, vol.31, no.6, June 1988.
- Software Testing: A Craftsman's Approach.2nd CRC publication, 2002