

# CS/CE/SE 6367

## Software Testing, Validation and Verification

Lecture 4  
Code Coverage (II)



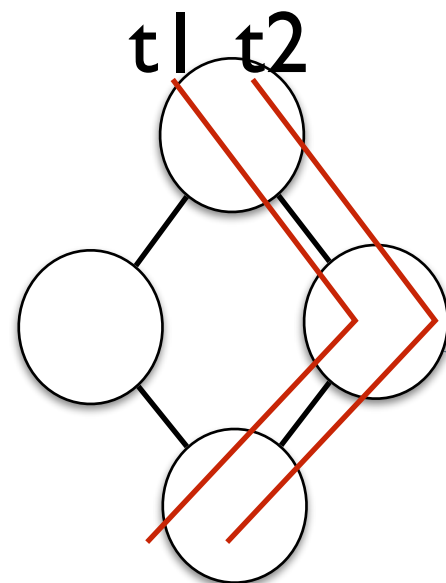
# Last Class

- Code coverage
  - Control-flow coverage
    - Statement coverage
    - Branch coverage
    - Path coverage
  - Coverage Collection Tools
    - EcEmma

# This Class

- 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



Are t1 and t2 identical?

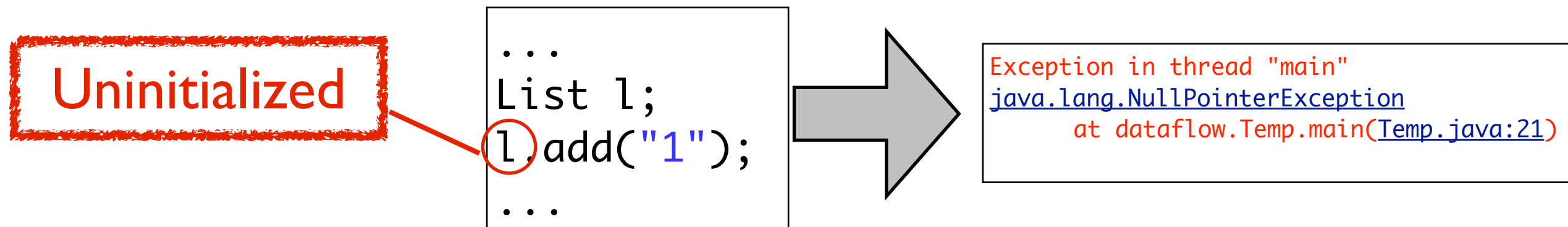
although the paths are the same, different tests may have different variable values defined/used

Control-flow graph

- 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 inaccessible (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.*,  $y = \mathbf{x} + 17$
  - as an call-by-value parameter in a subroutine or function call
    - *e.g.*,  $y = \text{sqrt}(\mathbf{x})$
  - in the predicate of a branch statement
    - *e.g.*,  $\text{if } (\mathbf{x} > 0) \{ \dots \}$

# 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:

```
if ( x > 0 ) {  
    print(y);  
}
```

There is a p-use of **x** and a c-use of **y**

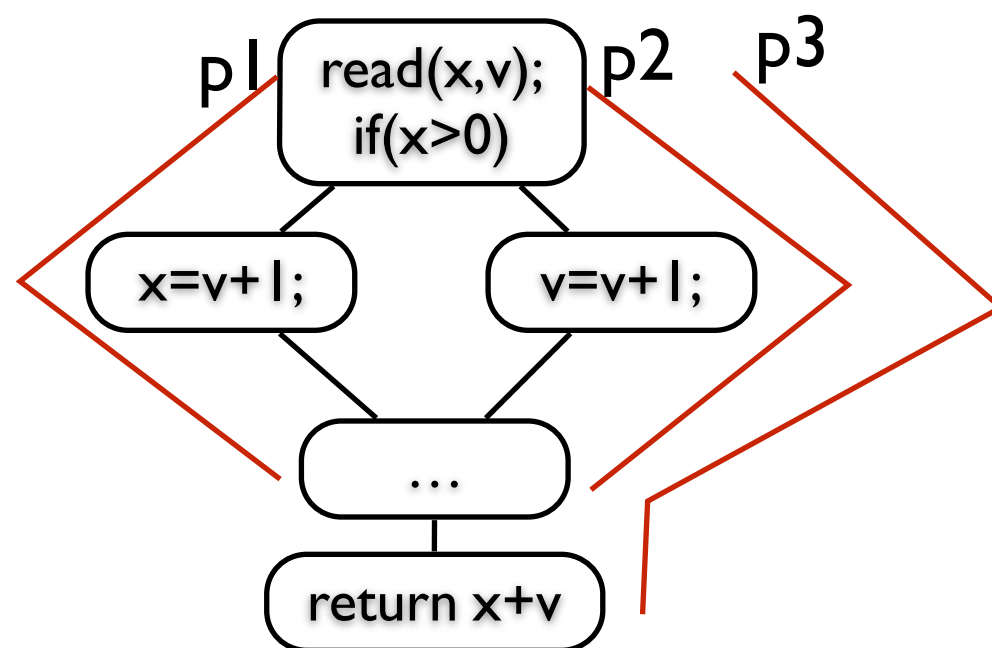


# Variable Use

- 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



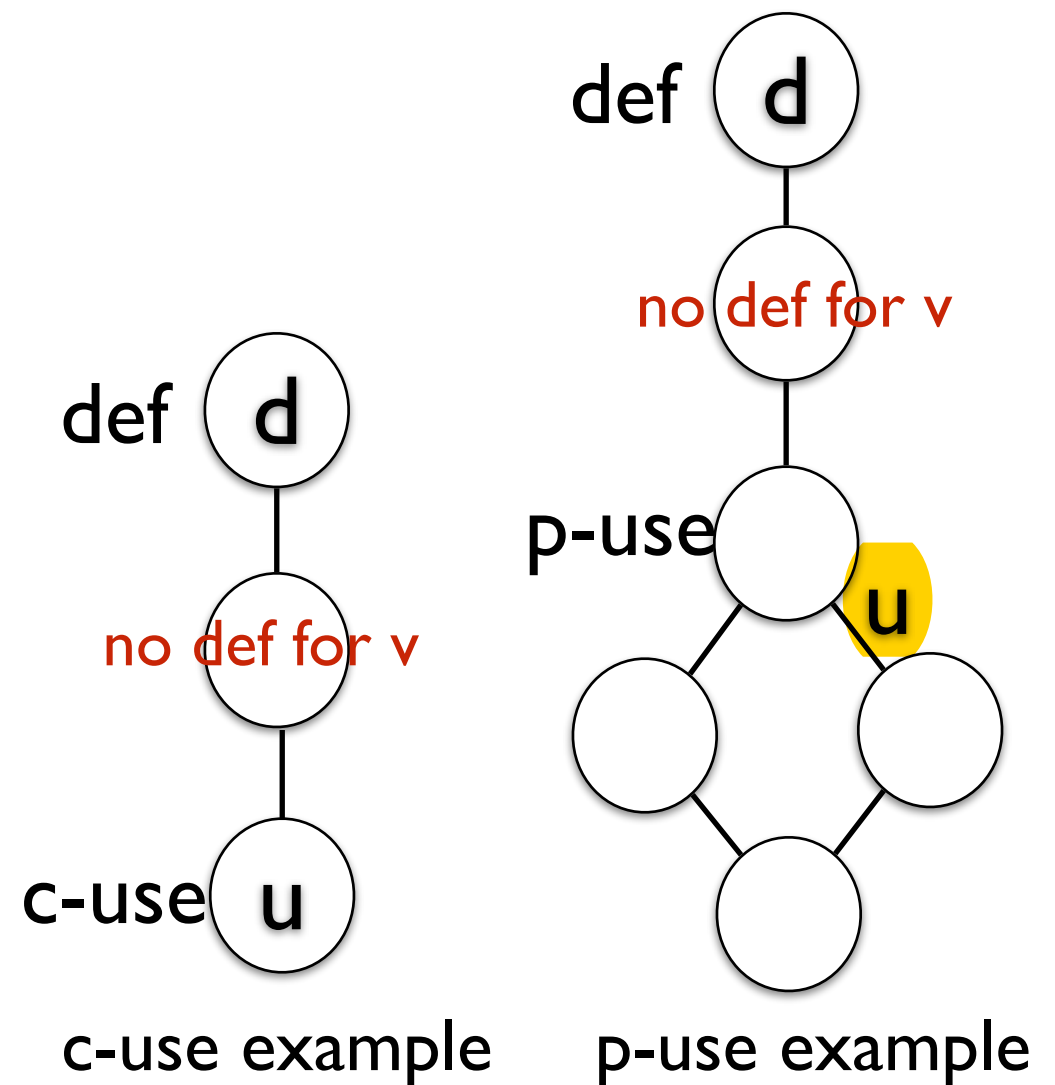
Control-flow graph

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

p1, p2 are not complete, while p3 is

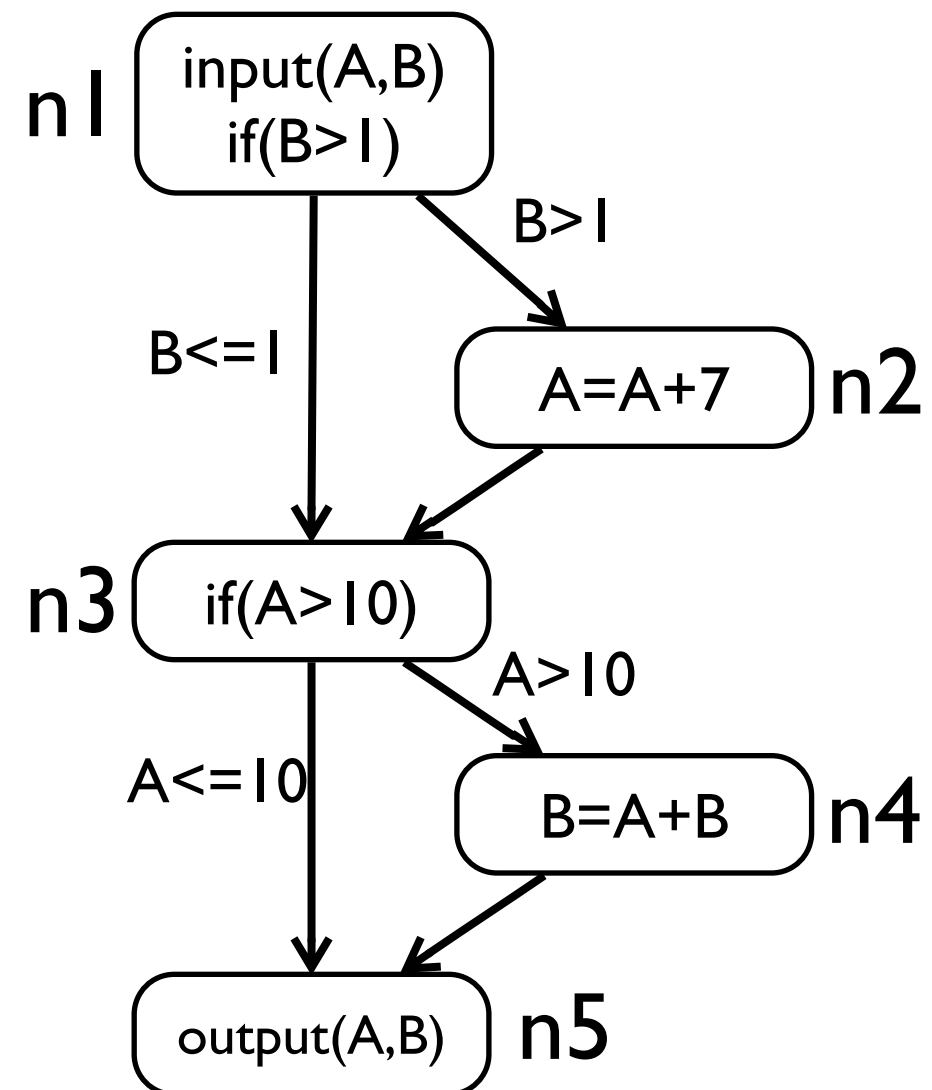
# Definition-Use Pair (DU-Pair)

- 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**



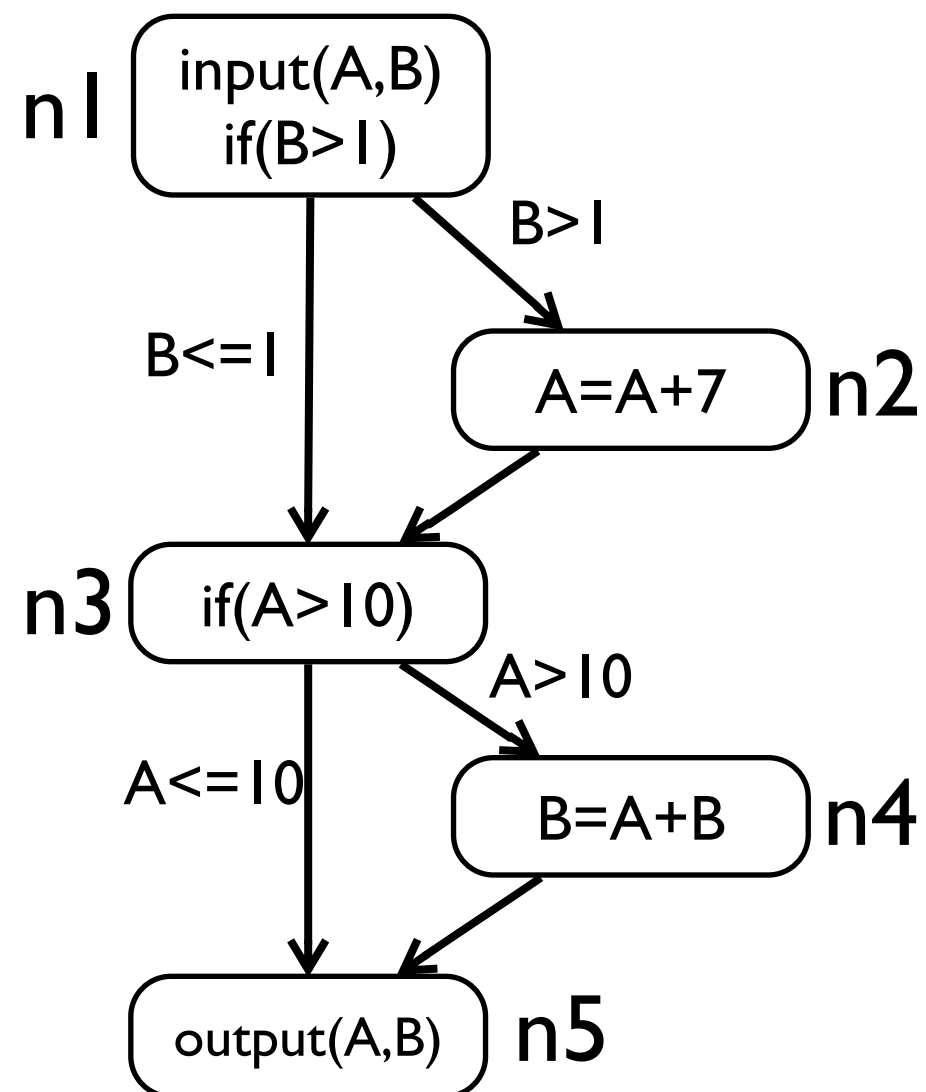
# DU-Pair: Example 1

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



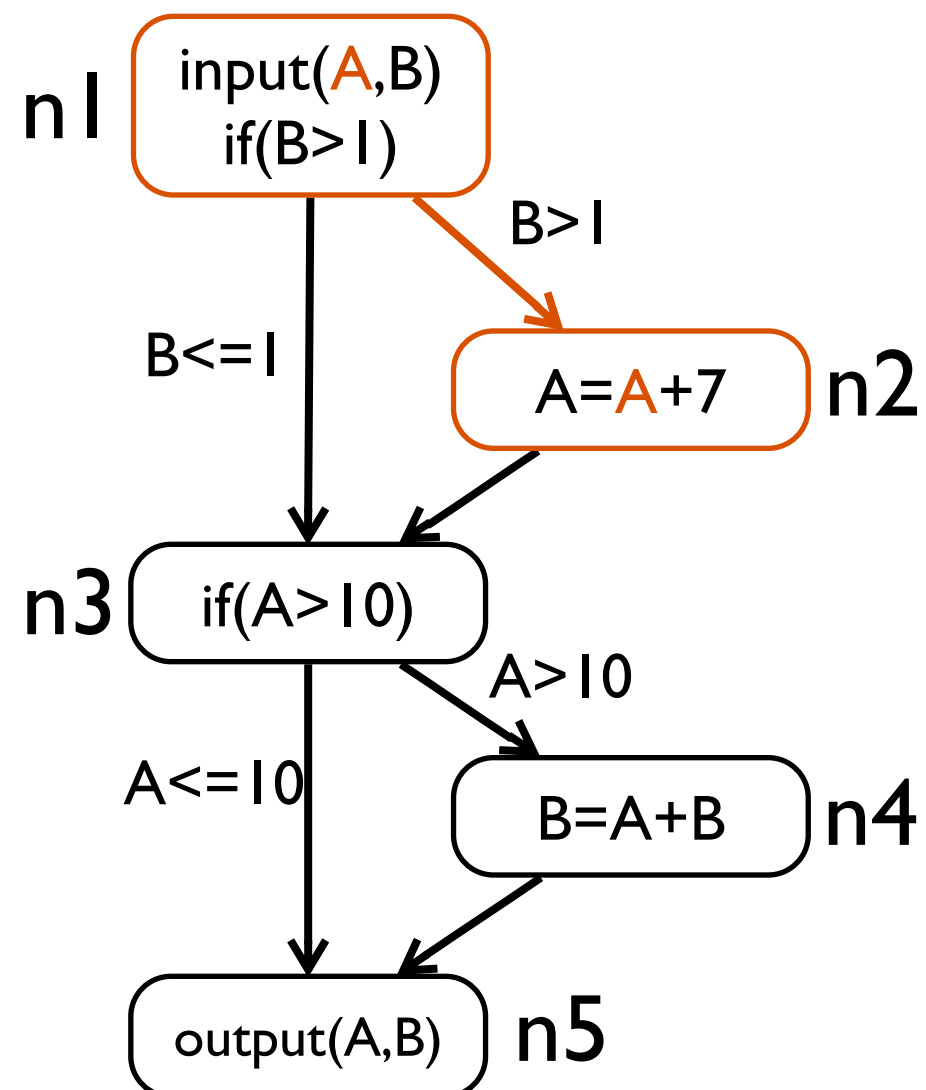
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(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>



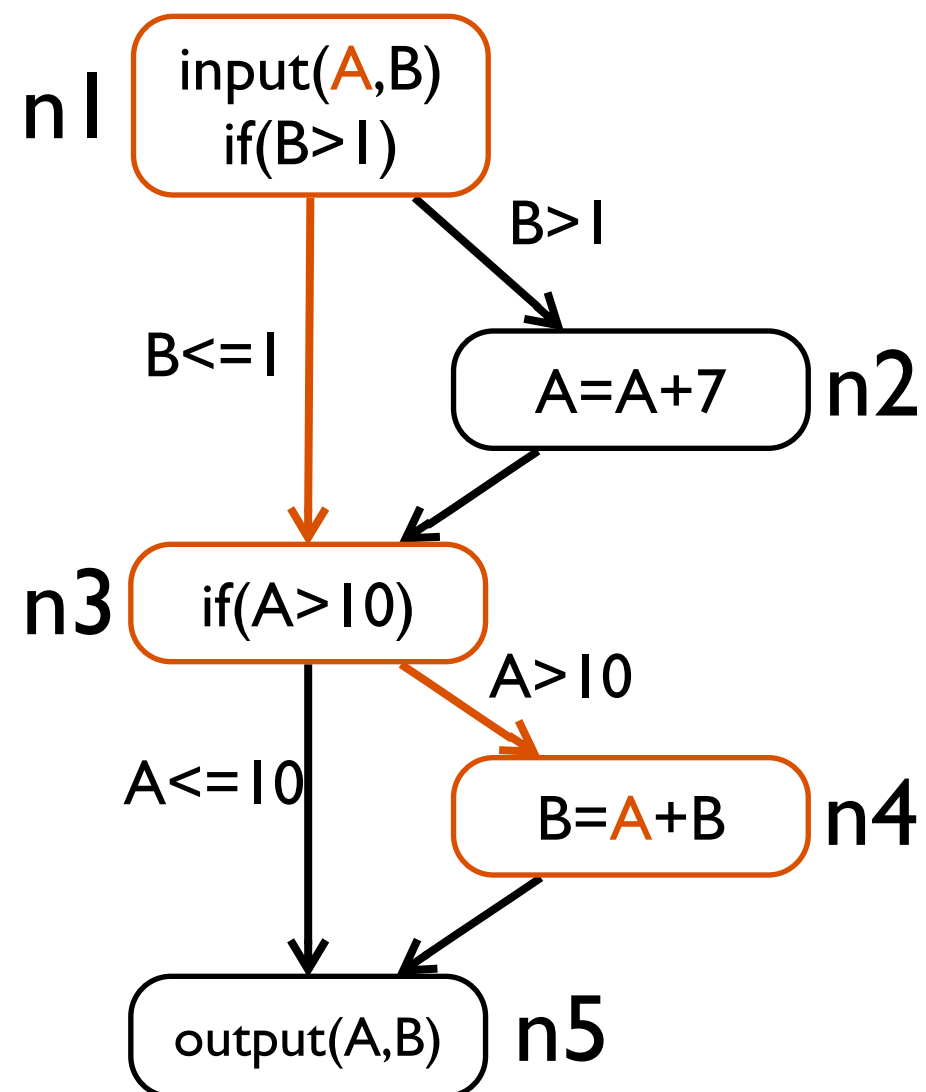
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
<b>(1,2)</b>	<b>&lt;1,2&gt;</b>
(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>



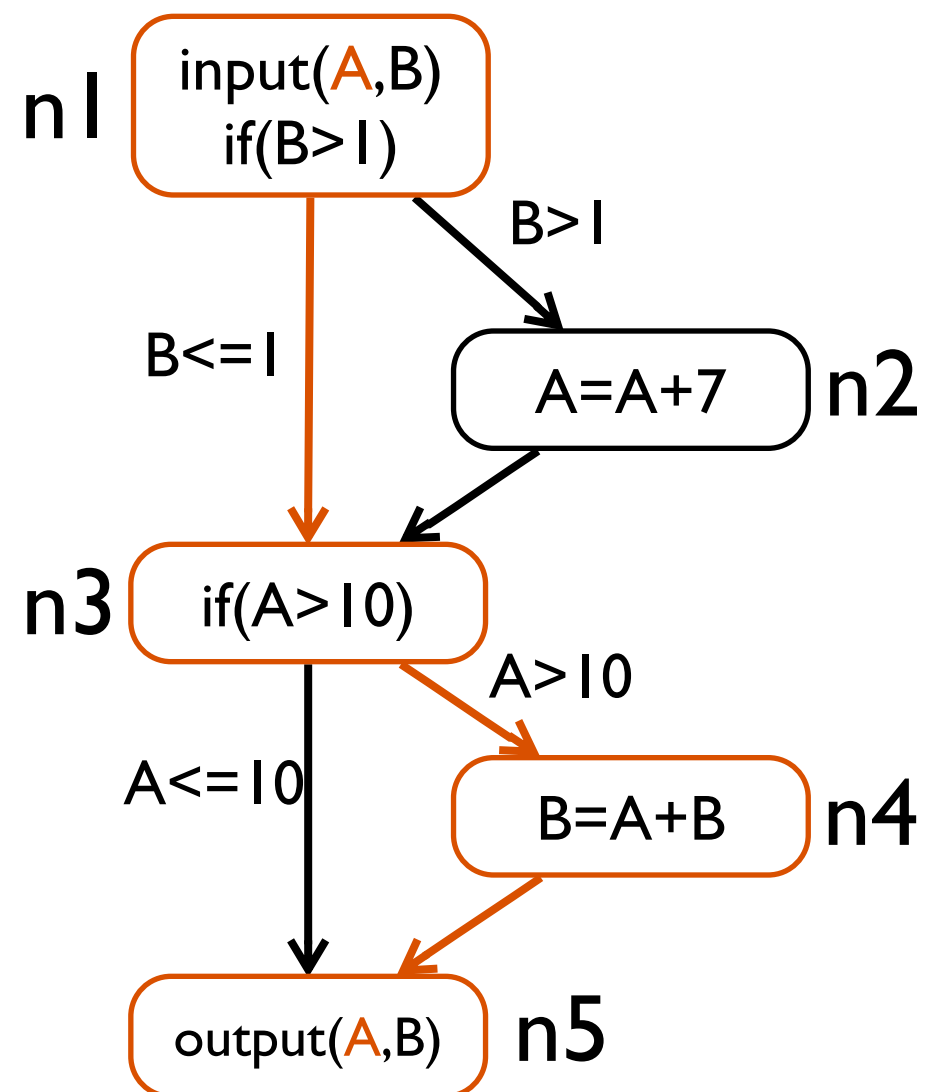
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2>
<b>(1,4)</b>	<b>&lt;1,3,4&gt;</b>
(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**

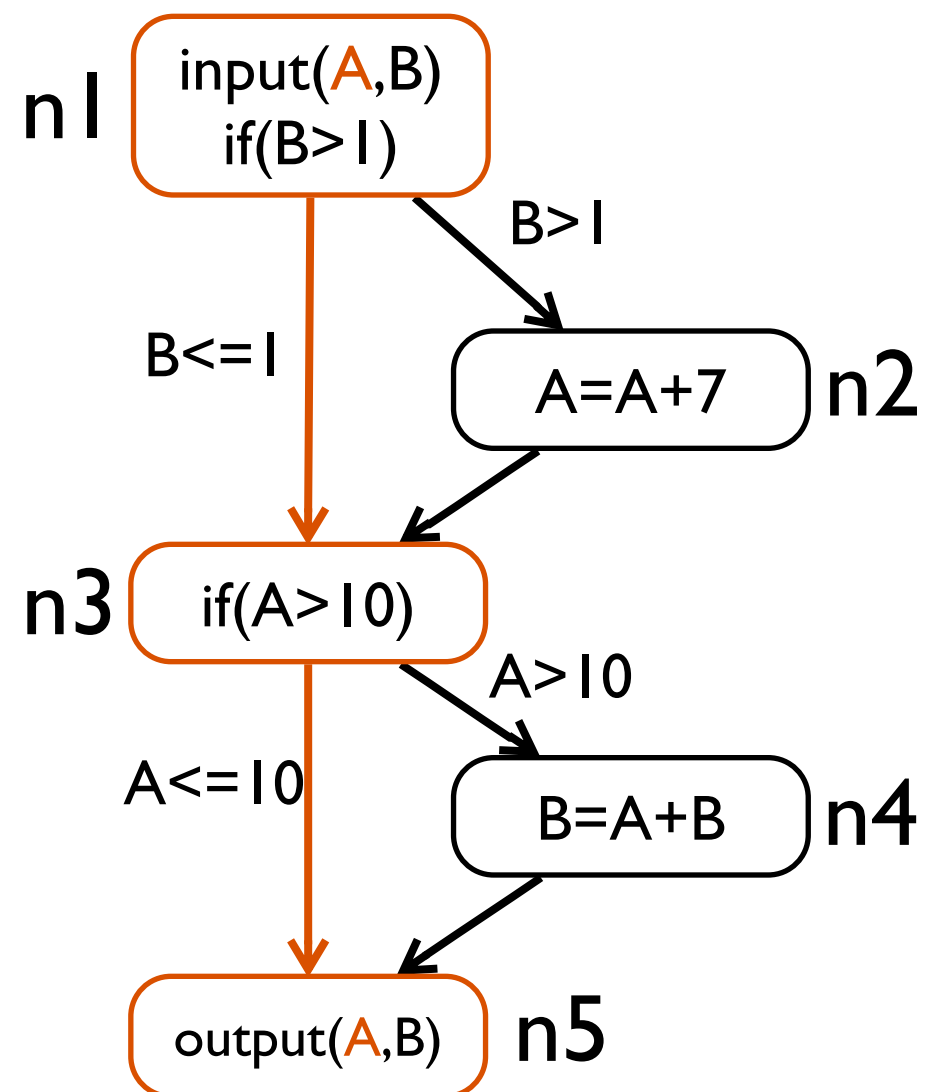
<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2>
(1,4)	<1,3,4>
<b>(1,5)</b>	<b>&lt;1,3,4,5&gt;</b>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>





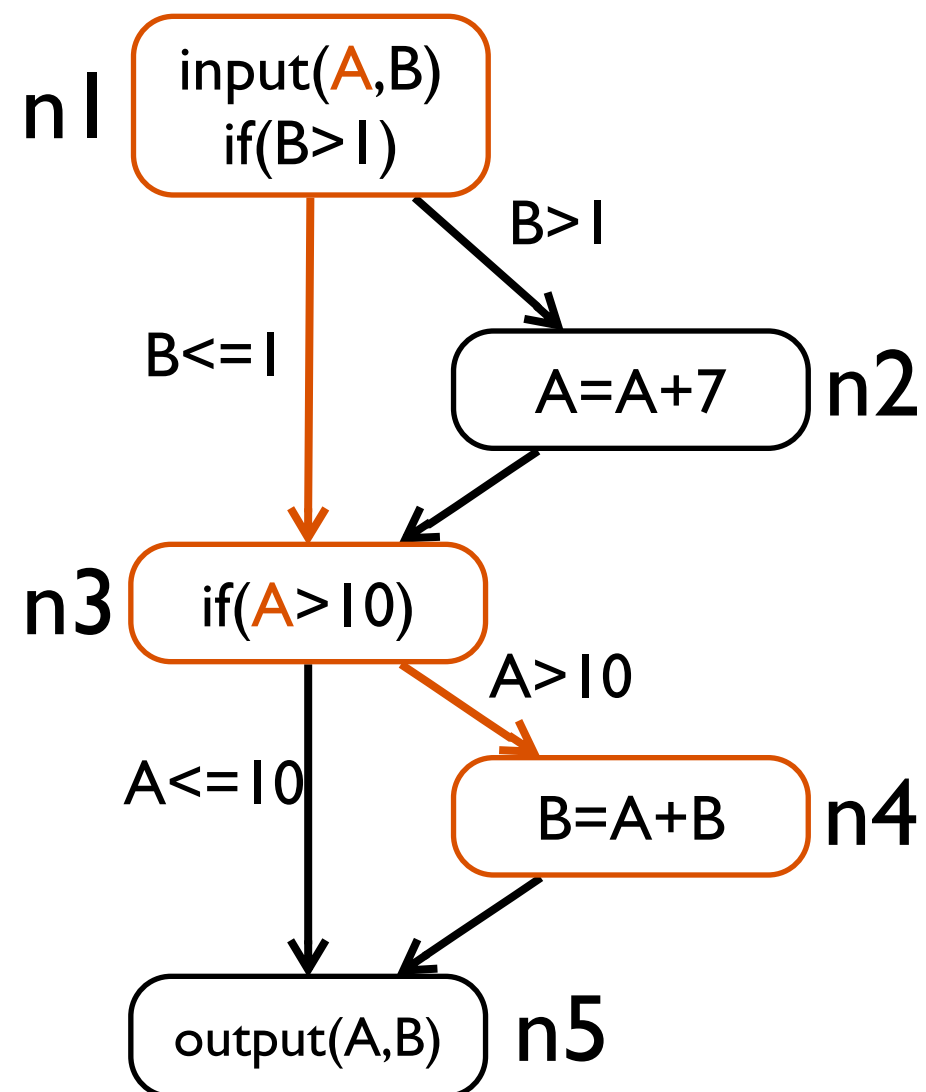
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2>
(1,4)	<1,3,4>
<b>(1,5)</b>	<1,3,4,5>
	<b>&lt;1,3,5&gt;</b>
(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>



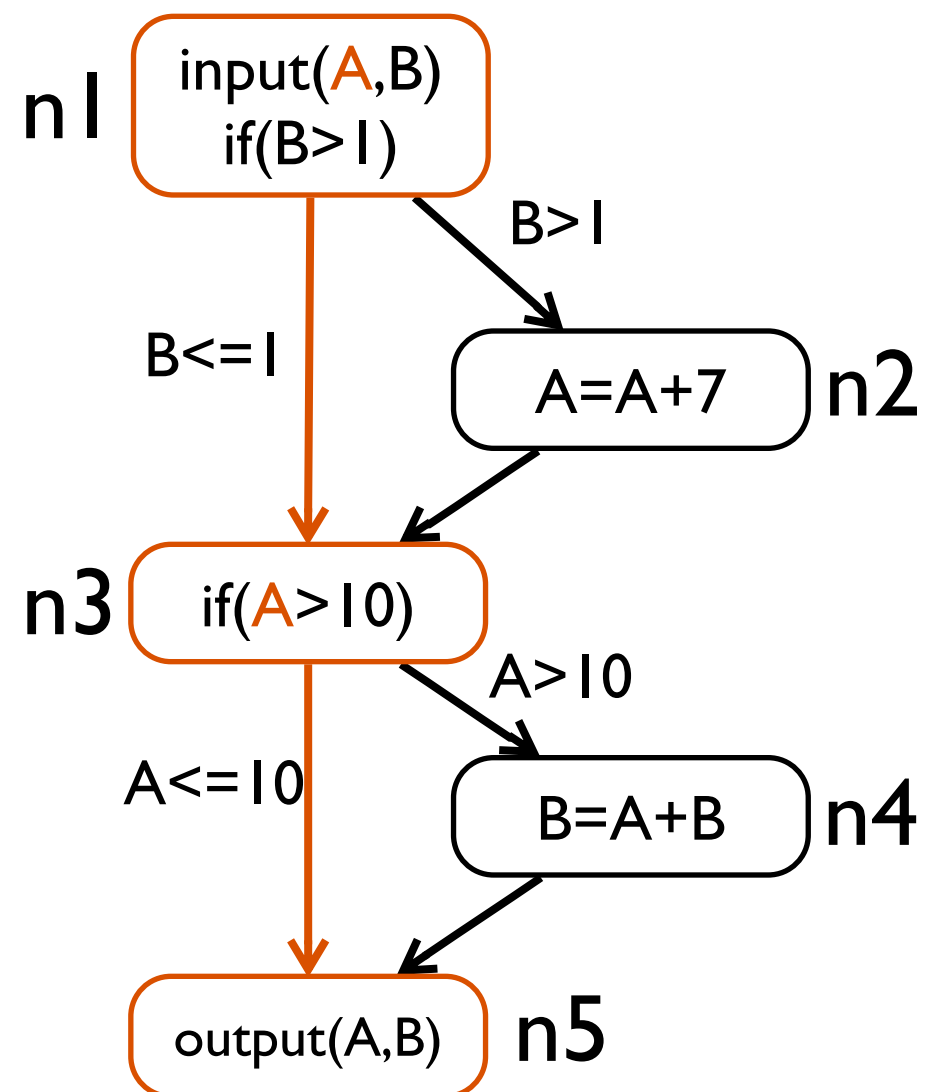
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
<b>(1,&lt;3,4&gt;)</b>	<b>&lt;1,3,4&gt;</b>
(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>



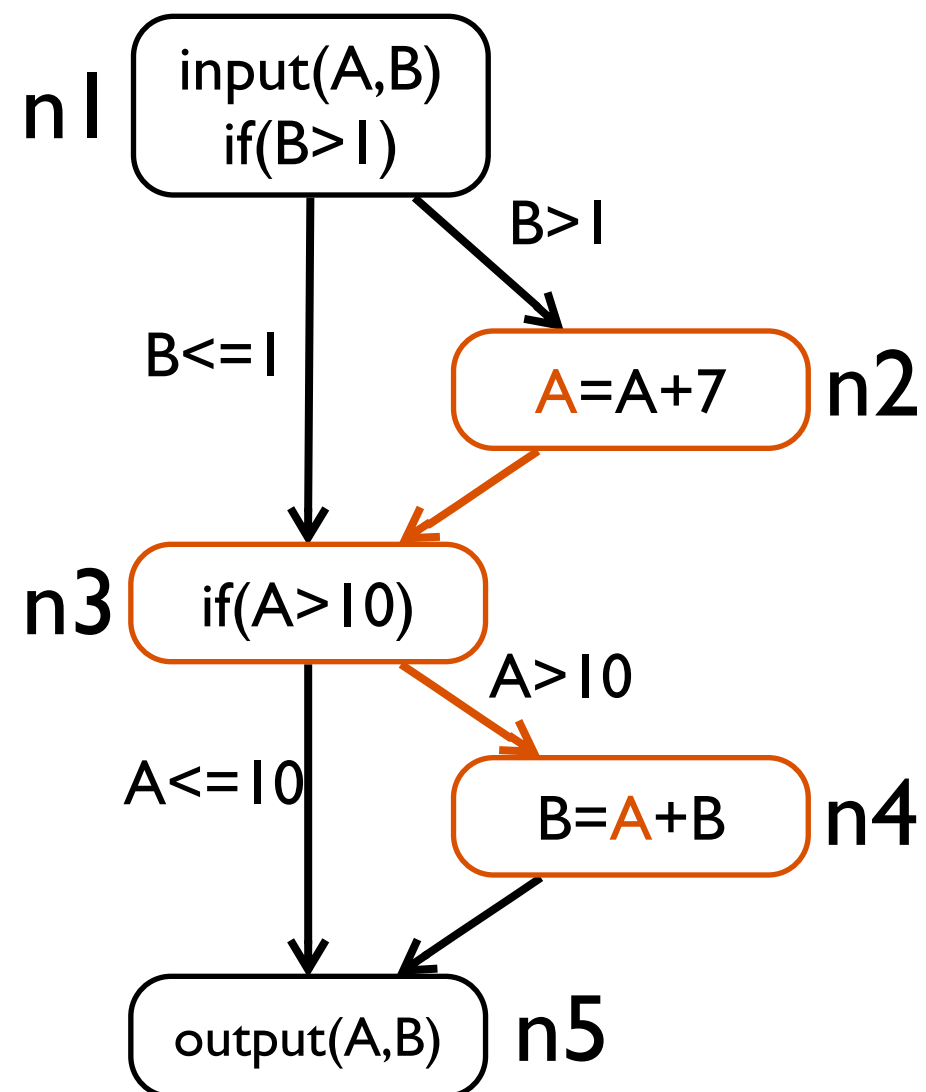
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
<b>(1,&lt;3,5&gt;)</b>	<b>&lt;1,3,5&gt;</b>
(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>



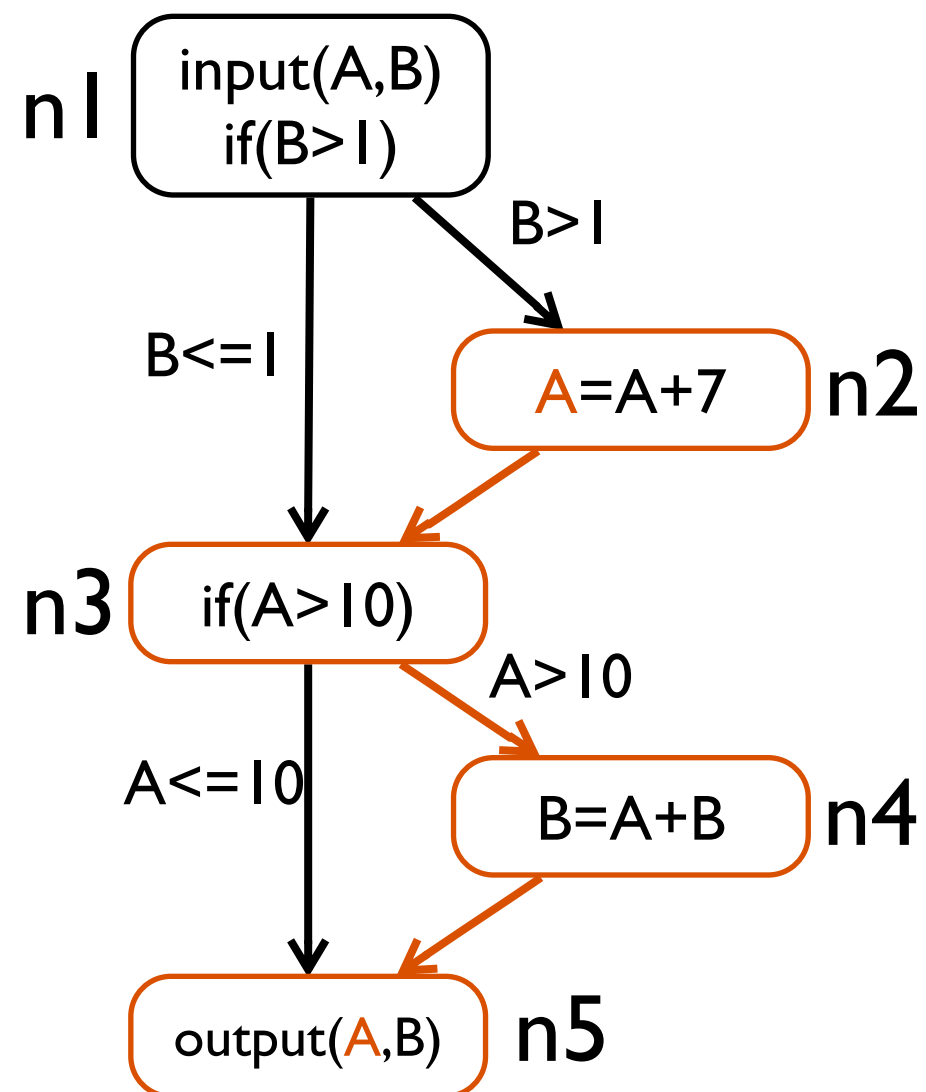
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(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>
<b>(2,4)</b>	<b>&lt;2,3,4&gt;</b>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



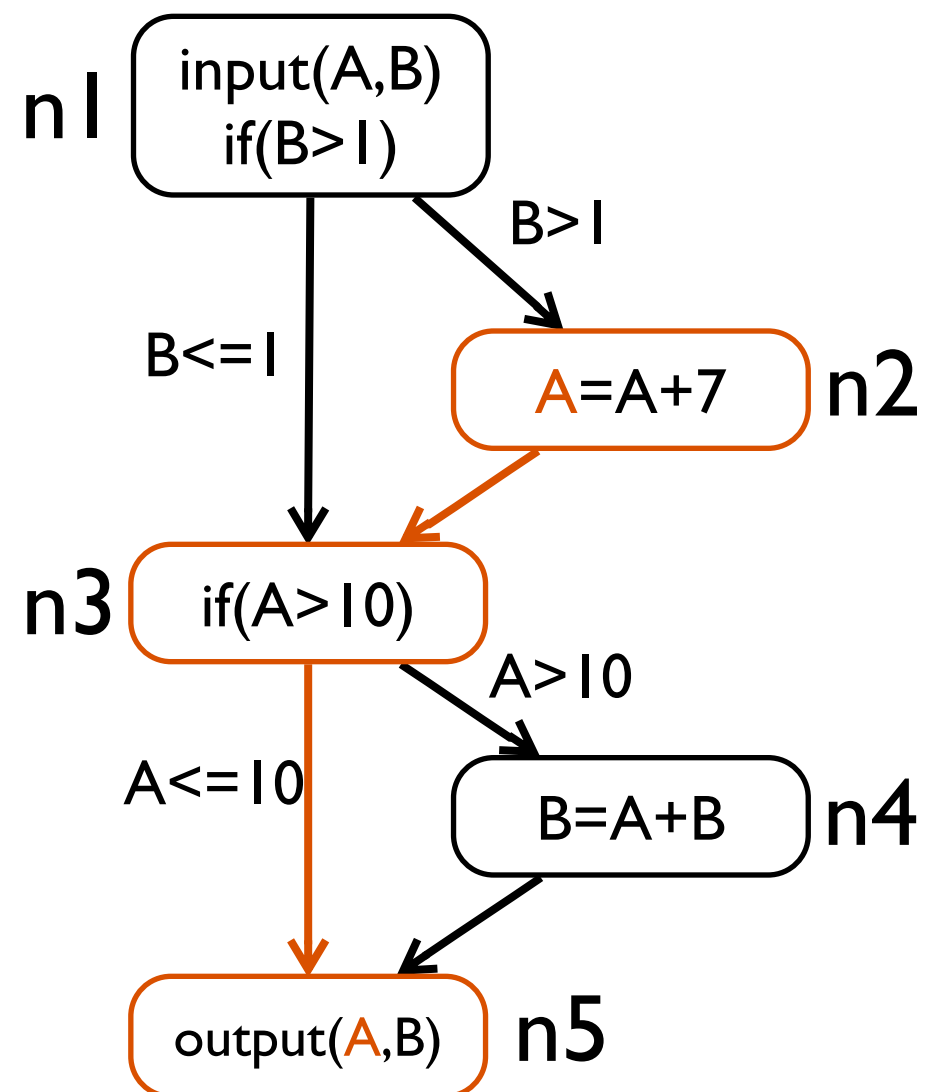
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(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>
<b>(2,5)</b>	<b>&lt;2,3,4,5&gt;</b>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



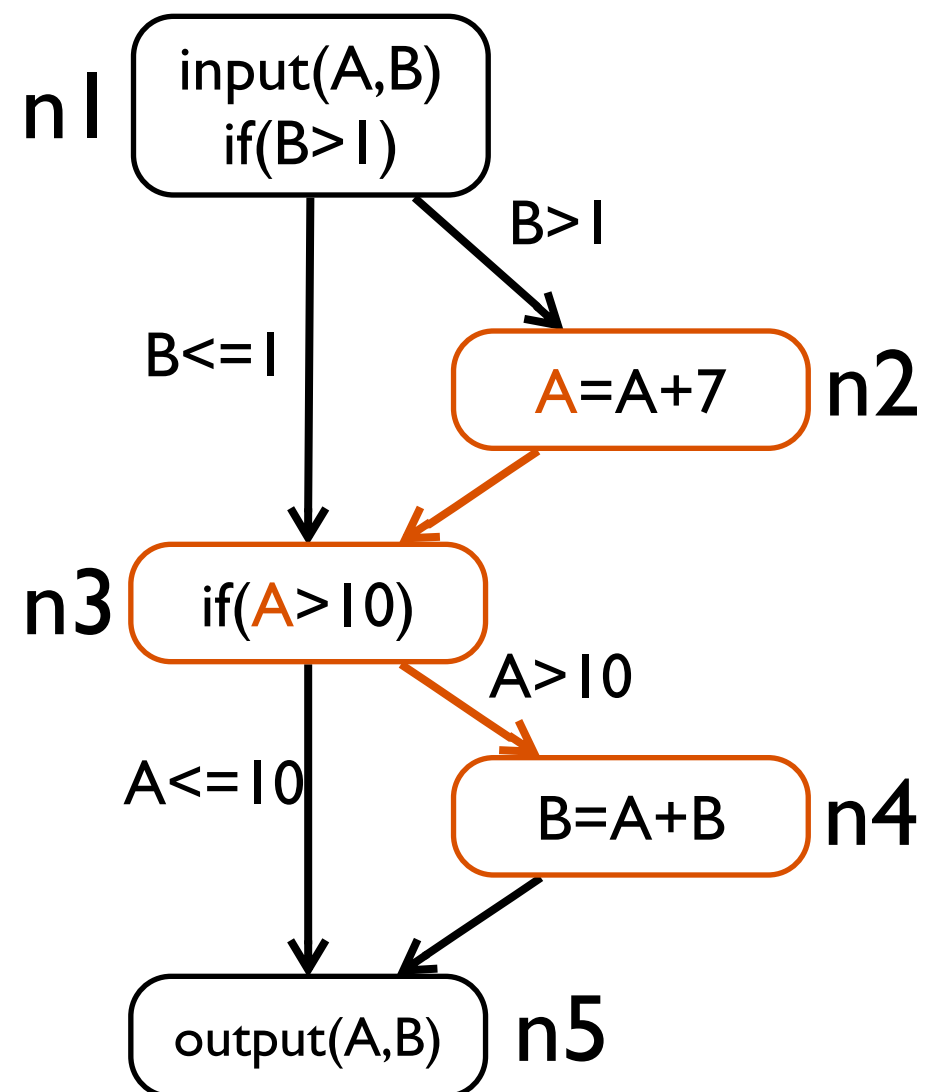
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(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>
<b>(2,5)</b>	<2,3,4,5>
	<b>&lt;2,3,5&gt;</b>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



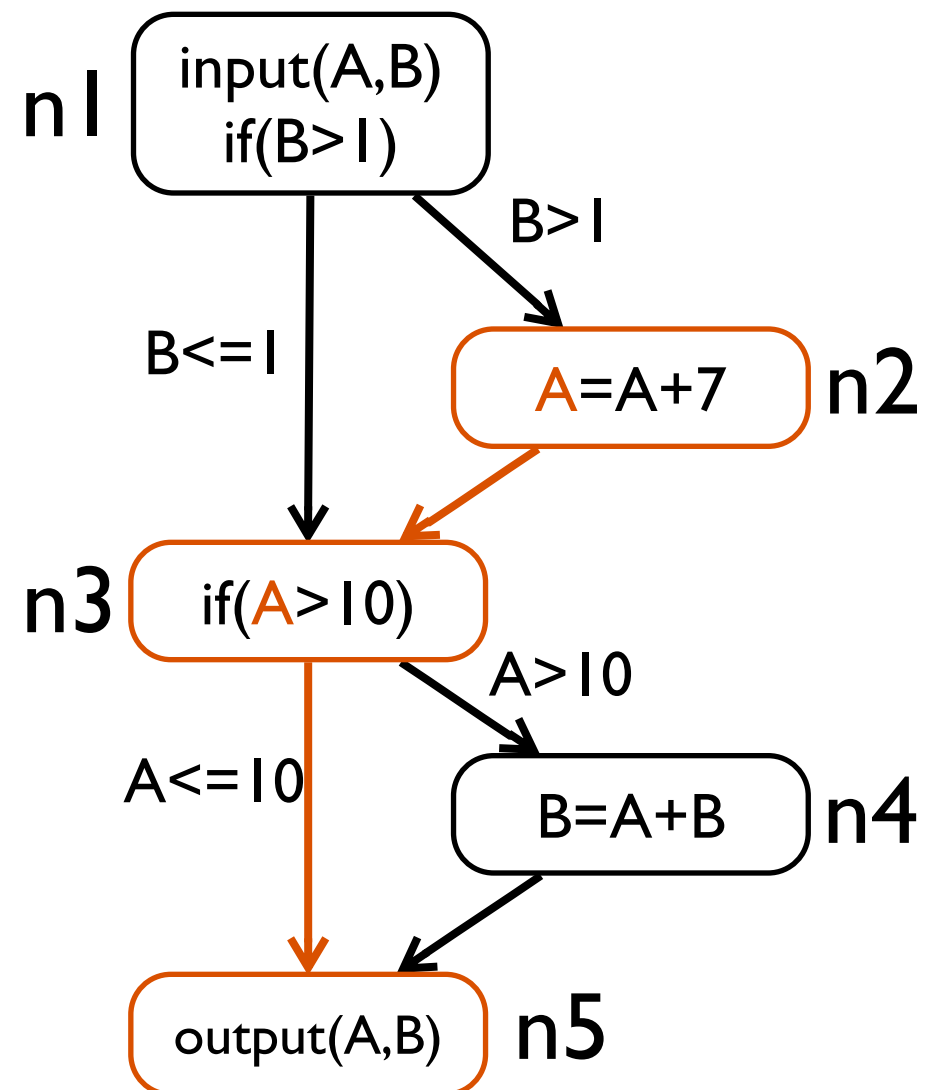
# Identifying DU-Pairs – Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(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>
<b>(2,&lt;3,4&gt;)</b>	<b>&lt;2,3,4&gt;</b>
(2,<3,5>)	<2,3,5>



# Identifying DU-Pairs – Variable **A**

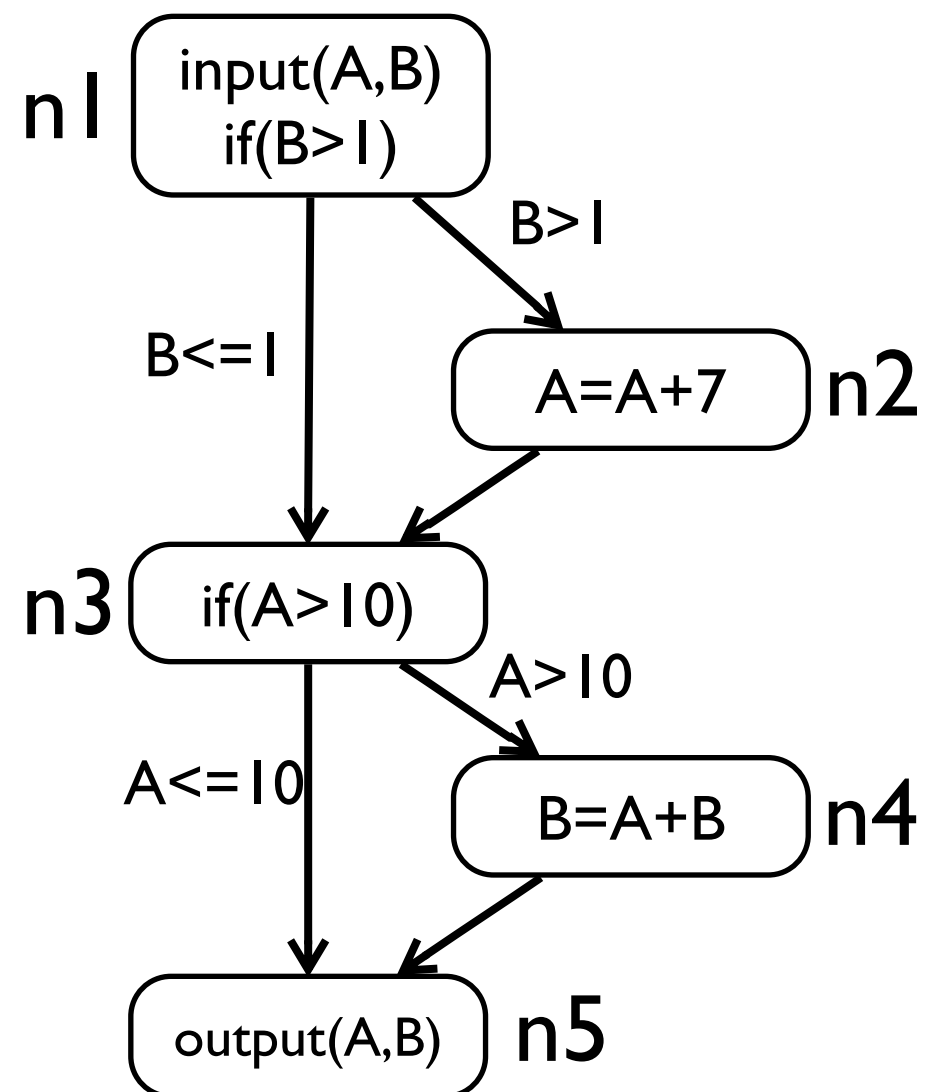
<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
<b>(2,&lt;3,5&gt;)</b>	<b>&lt;2,3,5&gt;</b>





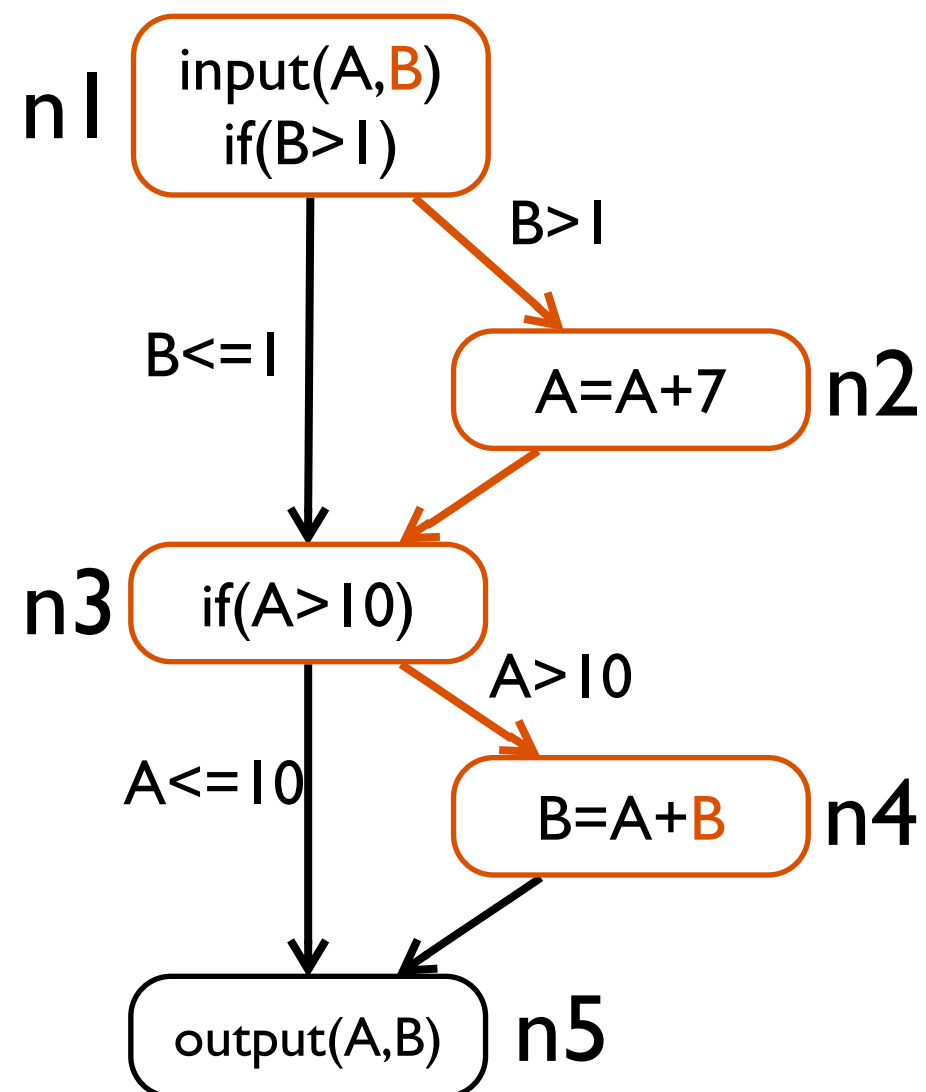
# Identifying DU-Pairs – Variable **B**

<u>du-pair</u>	<u>path(s)</u>
(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**

<u>du-pair</u>	<u>path(s)</u>
<b>(1,4)</b>	<b>&lt;1,2,3,4&gt;</b>
	<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

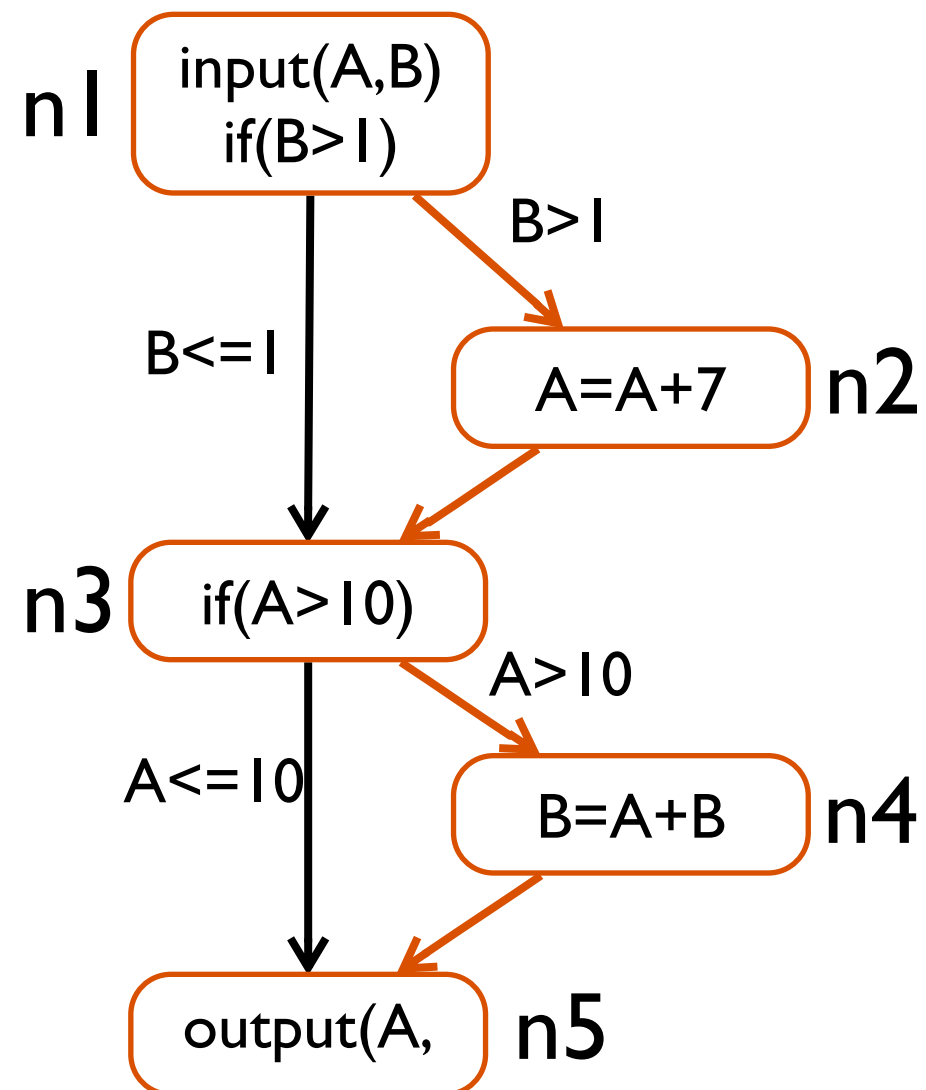
- **All-Defs**
  - for every program variable  $\mathbf{v}$ , at least one def-clear path from every definition of  $\mathbf{v}$  to at least one **c-use** or one **p-use** of  $\mathbf{v}$  must be covered

# 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?

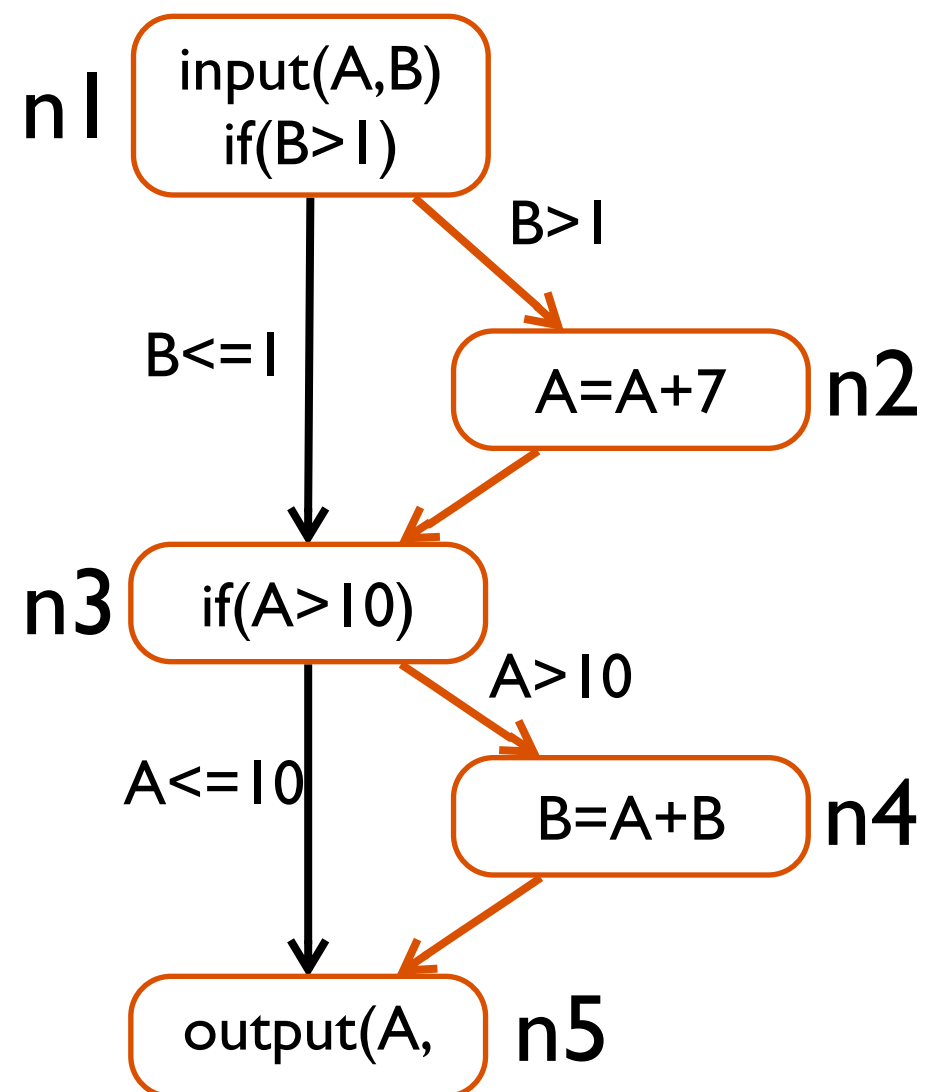
# Def-Clear Paths subsumed by <1,2,3,4,5> for Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2> <b>a</b>
(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> <b>a</b>
(2,5)	<2,3,4,5> <b>a</b>
	<2,3,5>
(2,<3,4>)	<2,3,4> <b>a</b>
(2,<3,5>)	<2,3,5>



# Def-Clear Paths Subsumed by <1,2,3,4,5> for Variable **B**

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4> <sub>a</sub>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2> <sub>a</sub>
(1,<1,3>)	<1,3>
(4,5)	<4,5> <sub>a</sub>



# Dataflow Test Coverage Criteria

- Since  $\langle 1, 2, 3, 4, 5 \rangle$  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

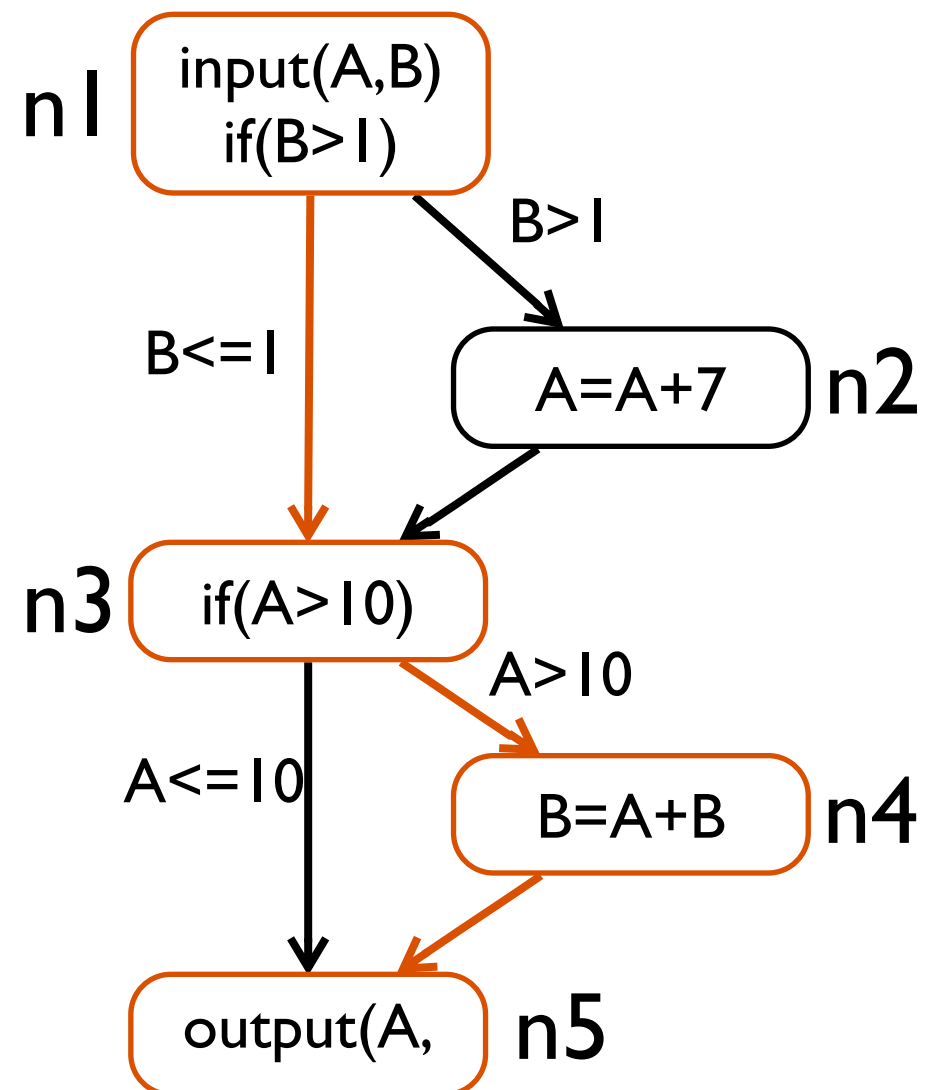
# Dataflow Test Coverage Criteria

- **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?



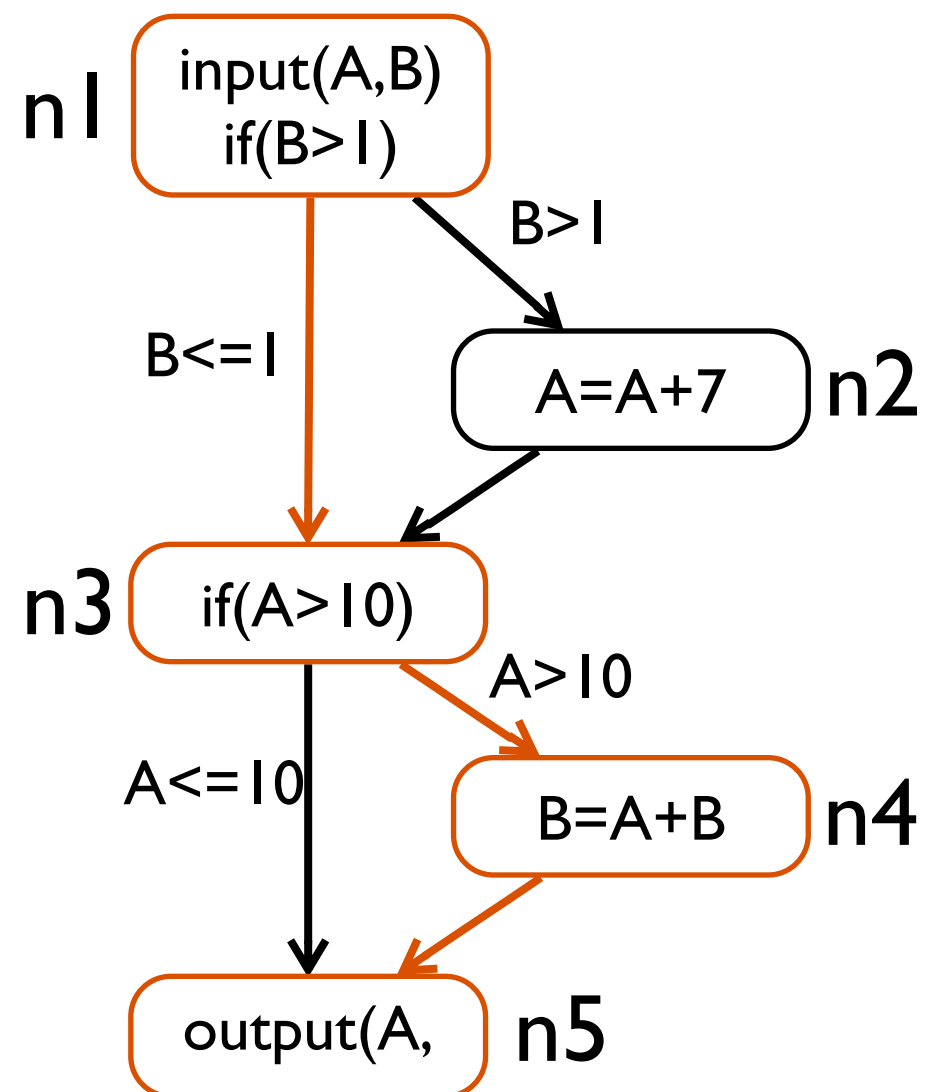
# Def-Clear Paths Subsumed by <1,3,4,5> for Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2> <b>a</b>
(1,4)	<1,3,4> <b>a</b>
(1,5)	<1,3,4,5> <b>a</b>
	<1,3,5>
(1,<3,4>)	<1,3,4> <b>a</b>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> <b>a</b>
(2,5)	<2,3,4,5> <b>a</b>
	<2,3,5>
(2,<3,4>)	<2,3,4> <b>a</b>
(2,<3,5>)	<2,3,5>



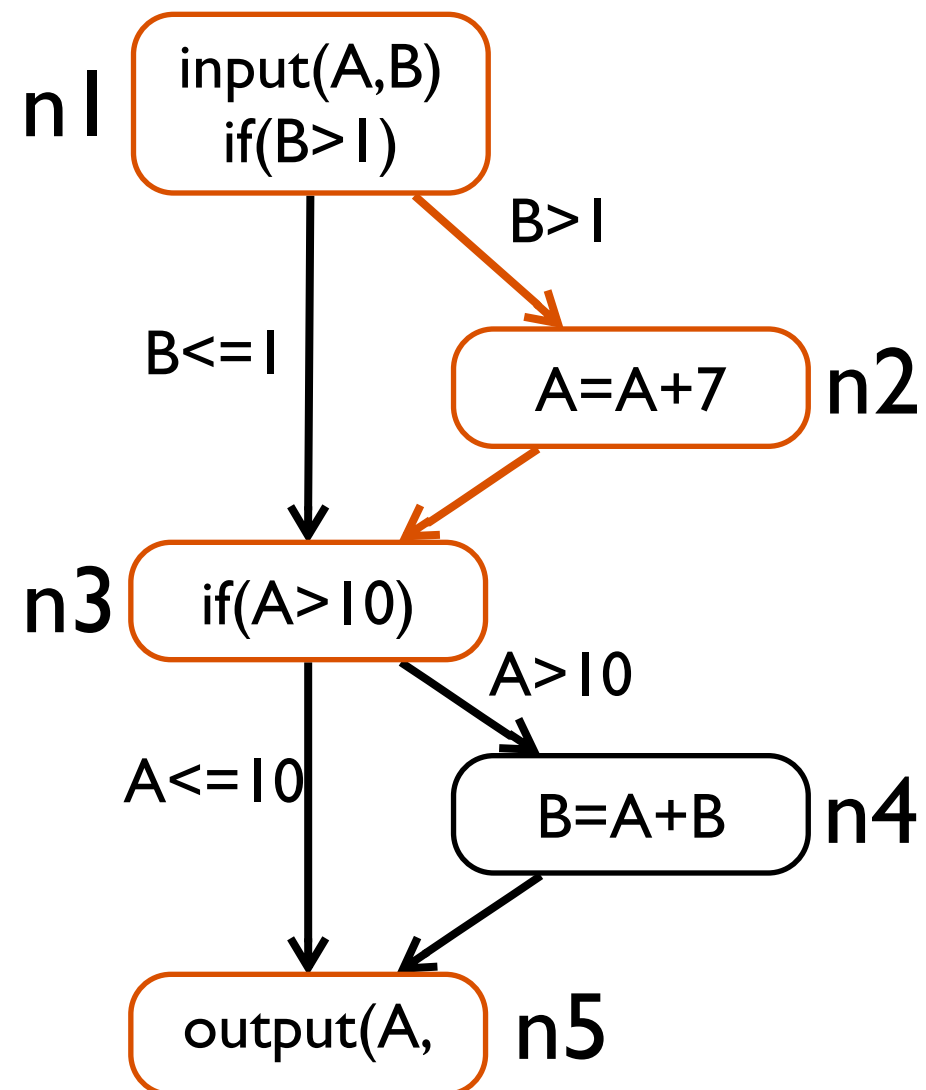
# Def-Clear Paths Subsumed by <1,3,4,5> for Variable **B**

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4> <sub>a</sub>
	<1,3,4> <sub>a</sub>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2> <sub>a</sub>
(1,<1,3>)	<1,3> <sub>a</sub>
(4,5)	<4,5> <sub>a a</sub>



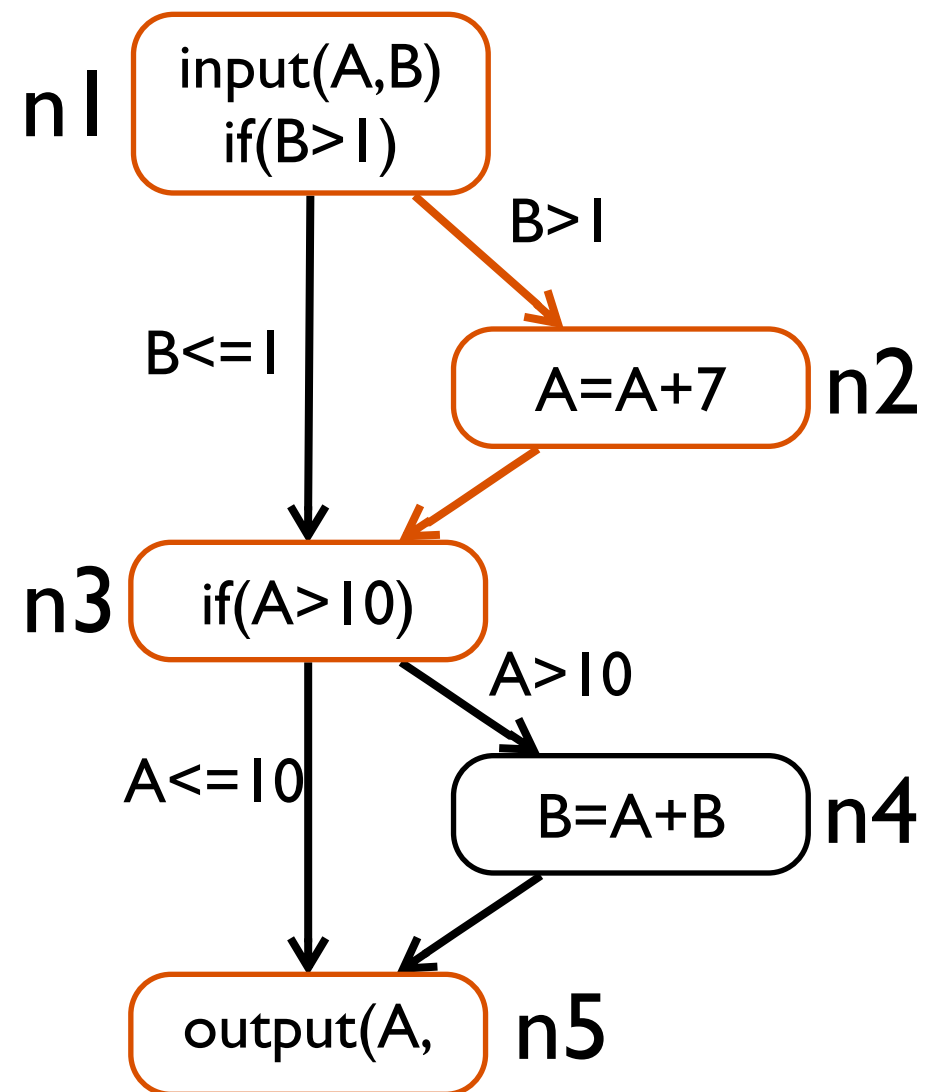
# Def-Clear Paths Subsumed by <1,2,3,5> for Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2> <b>a</b> <b>a</b>
(1,4)	<1,3,4> <b>a</b>
(1,5)	<1,3,4,5> <b>a</b>
	<1,3,5>
(1,<3,4>)	<1,3,4> <b>a</b>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> <b>a</b>
(2,5)	<2,3,4,5> <b>a</b>
	<2,3,5> <b>a</b>
(2,<3,4>)	<2,3,4> <b>a</b>
(2,<3,5>)	<2,3,5> <b>a</b>



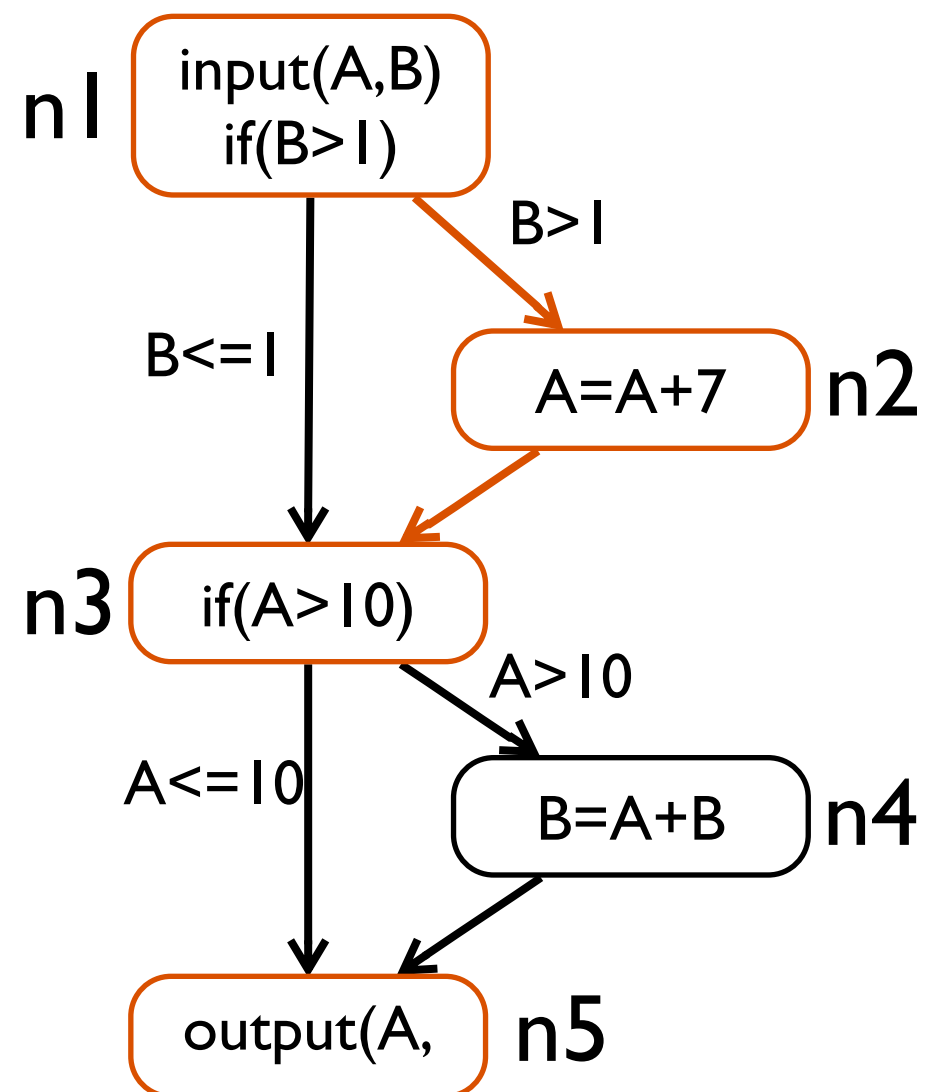
# Def-Clear Paths Subsumed by <1,2,3,5> for Variable **A**

<u>du-pair</u>	<u>path(s)</u>
(1,2)	<1,2> <b>a</b> a
(1,4)	<1,3,4> <b>a</b>
(1,5)	<1,3,4,5> <b>a</b>
	<1,3,5>
(1,<3,4>)	<1,3,4> <b>a</b>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4> <b>a</b>
(2,5)	<2,3,4,5> <b>a</b>
	<2,3,5> <b>a</b>
(2,<3,4>)	<2,3,4> <b>a</b>
(2,<3,5>)	<2,3,5> <b>a</b>



# Def-Clear Paths Subsumed by <1,2,3,5> for Variable **B**

<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4> <sub>a</sub>
	<1,3,4> <sub>a</sub>
(1,5)	<1,2,3,5> <sub>a</sub>
	<1,3,5>
(1,<1,2>)	<1,2> <sub>a</sub> a
(1,<1,3>)	<1,3> <sub>a</sub>
(4,5)	<4,5> <sub>a</sub> a



# Dataflow Test Coverage Criteria

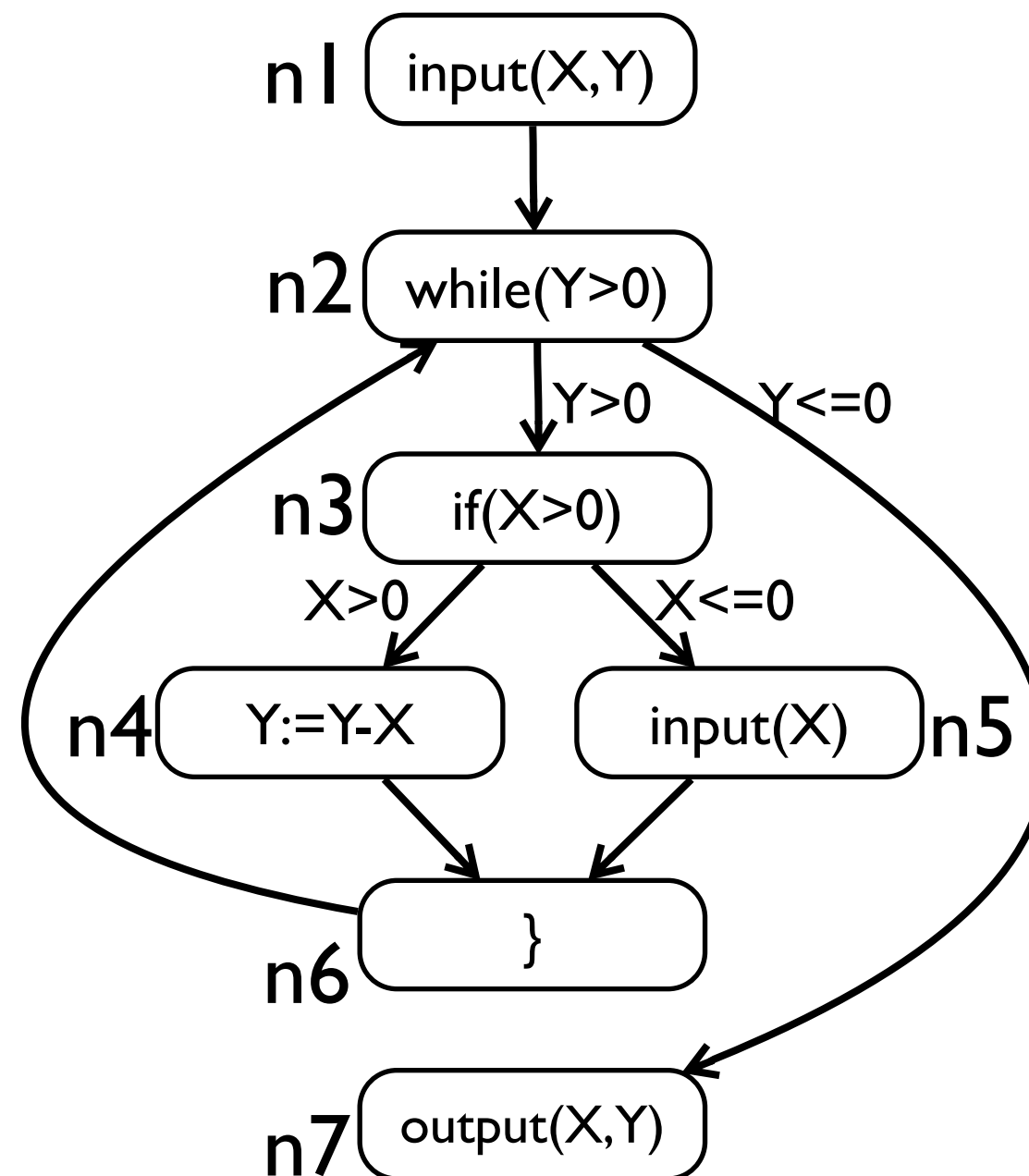
- None of the three test cases covers the du-pair  $(1, \langle 3, 5 \rangle)$  for variable **A**,
- **All-Uses** Coverage is not achieved

# DU-Pair: More Complicated Example

```

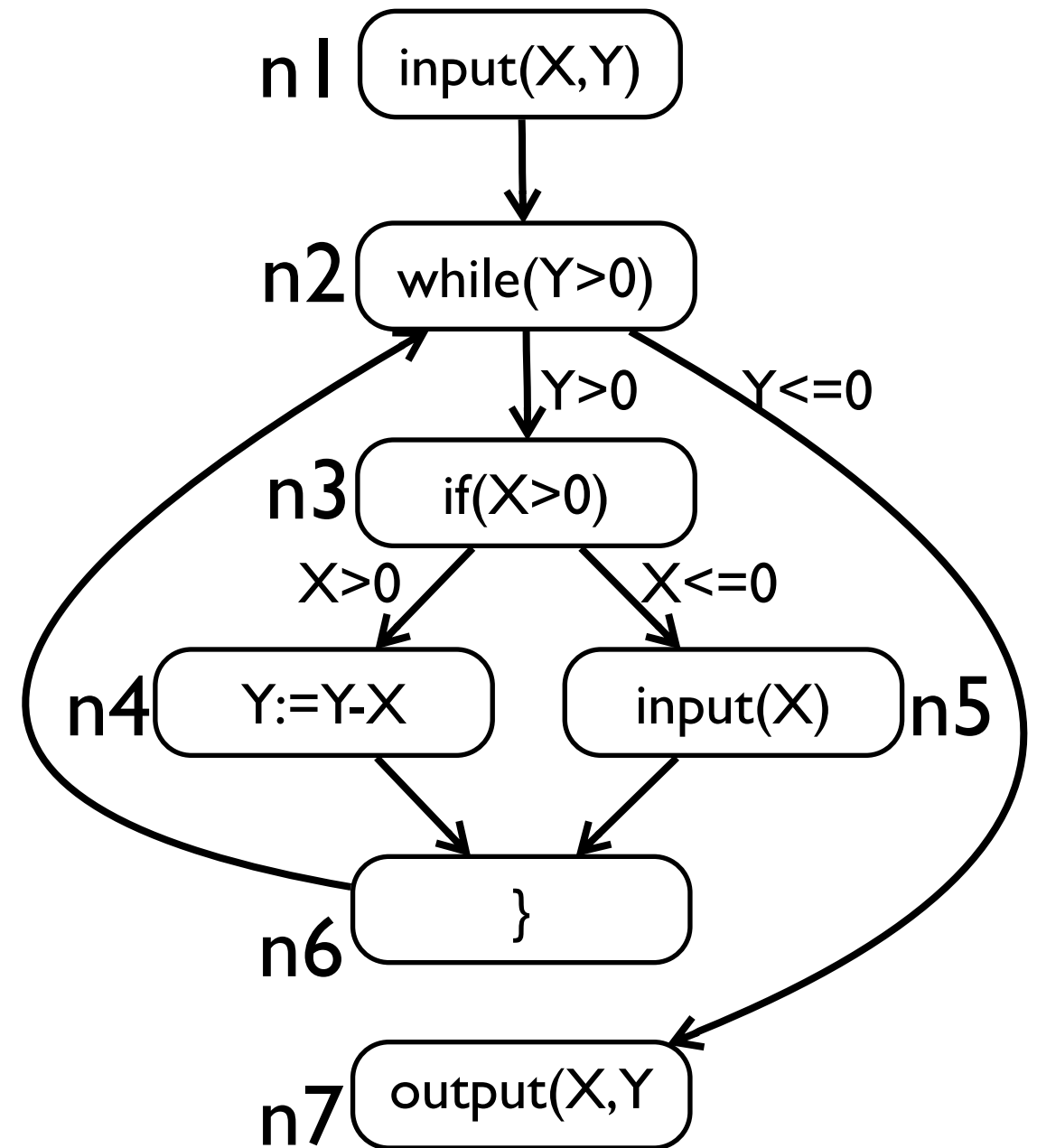
1. input(X,Y)
2. while (Y>0) {
3.   if (X>0)
4.     Y := Y-X
5.   else
6.     input(X)
7. }
8. output(X,Y)

```



<u>du-pair</u>	<u>path(s)</u>
(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>

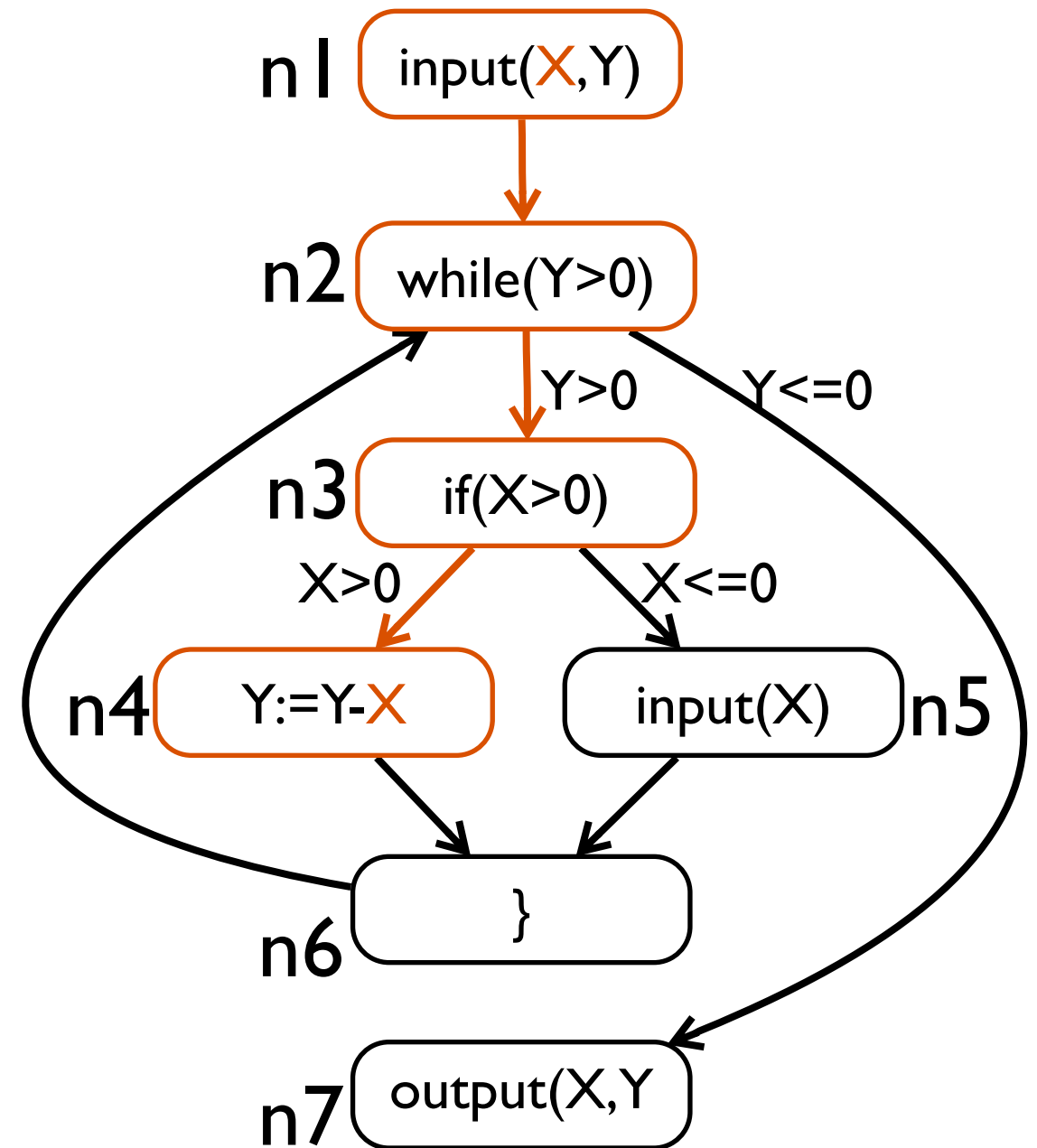
airs – Variable **X**





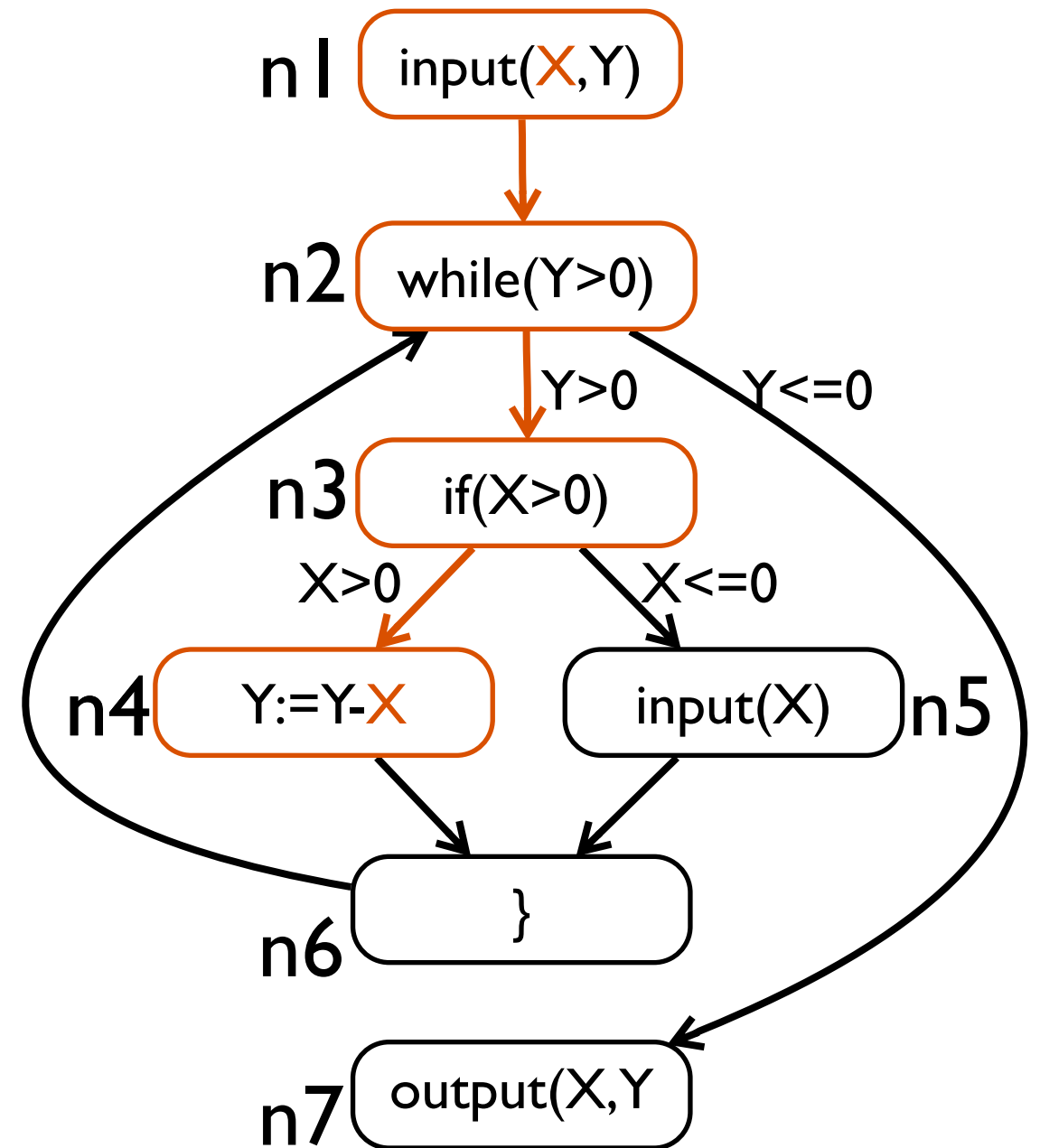
<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>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

airs – Variable **X**



<u>du-pair</u>	<u>path(s)</u>
(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>

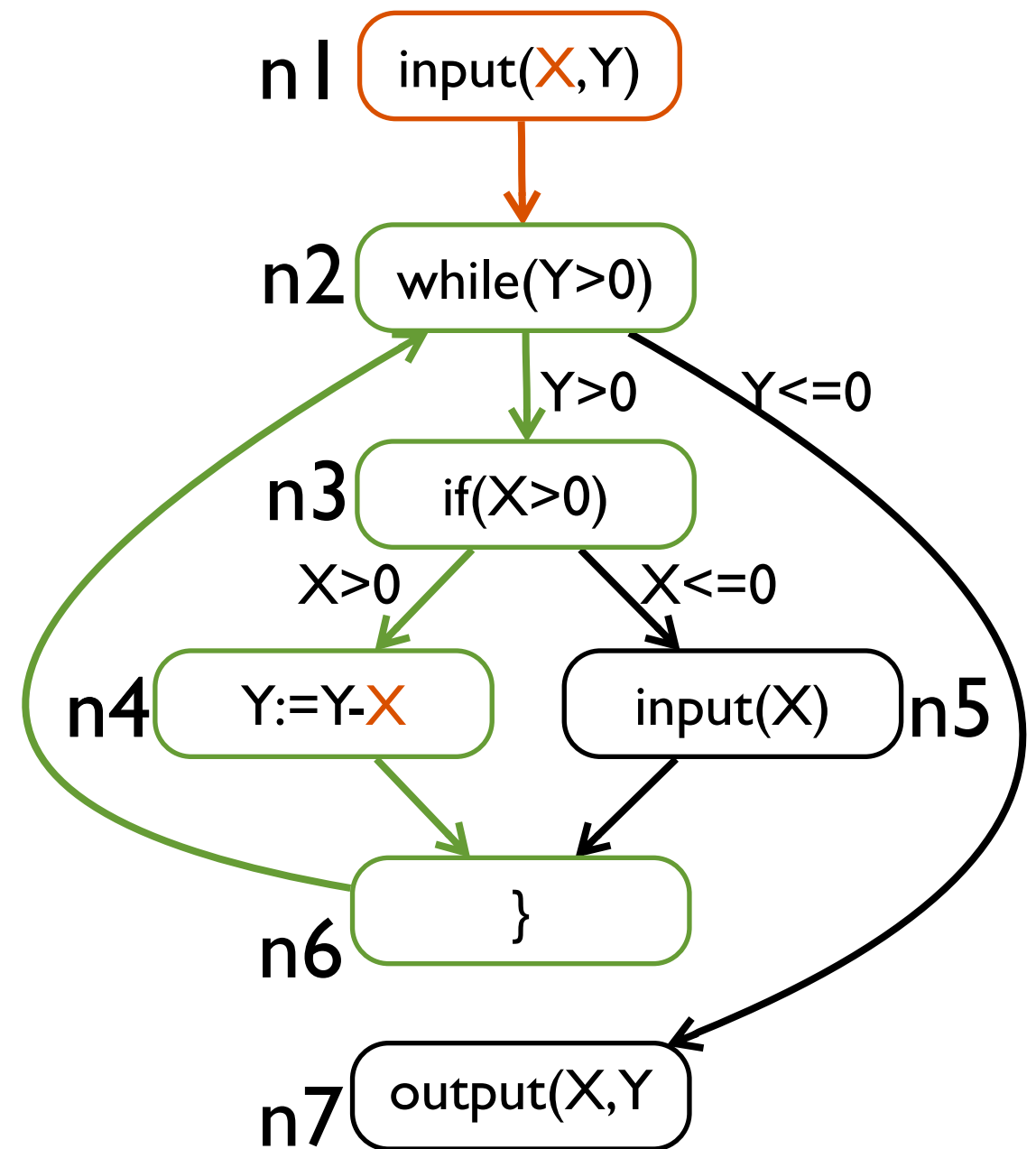
pairs – Variable **X**



(a)\* means “a” will be repeated for  $\geq 1$  times

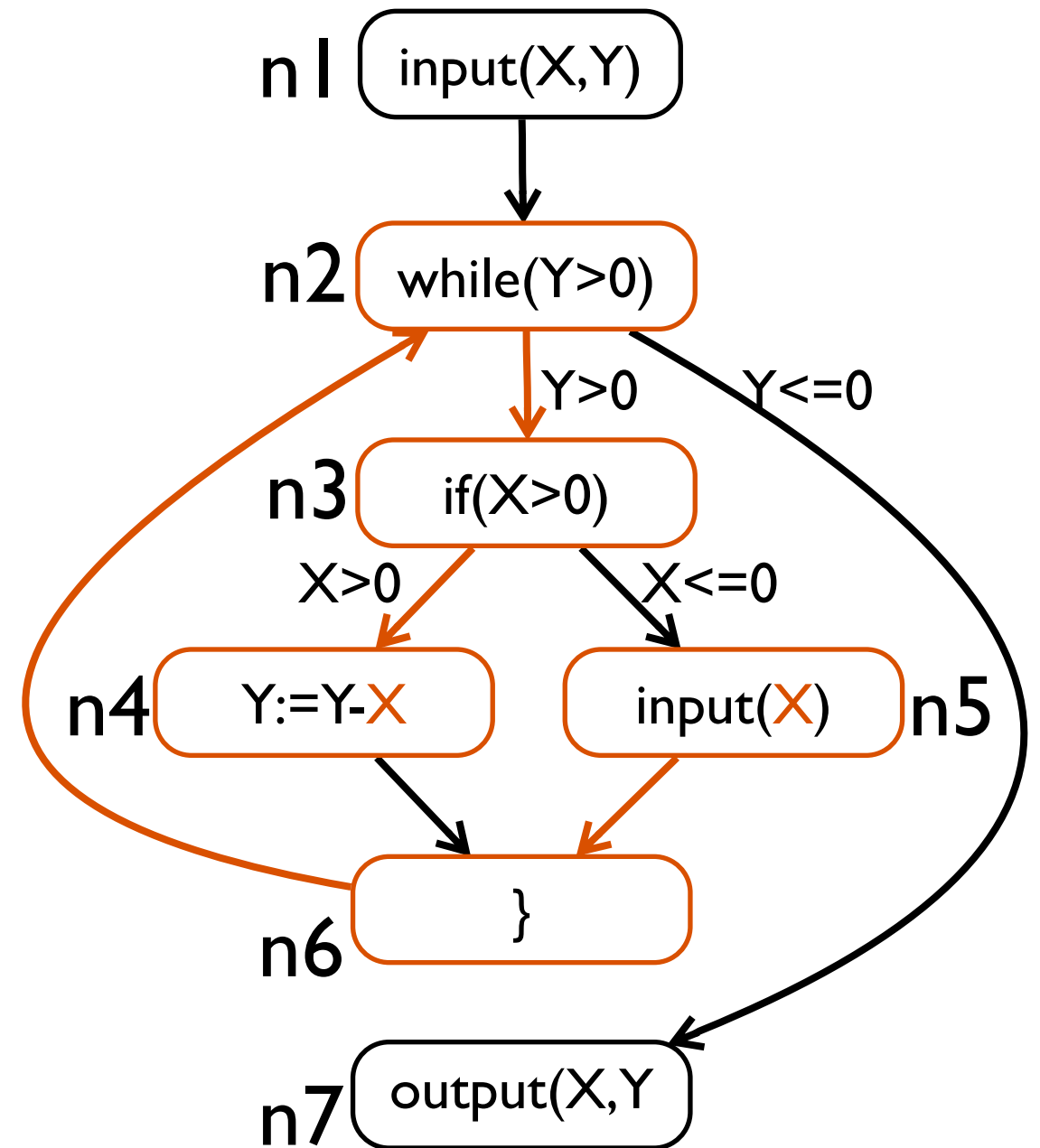
<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
(5,7)	<5,6,2,7>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>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

pairs – Variable **X**



<u>du-pair</u>	<u>path(s)</u>
(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)*>
<b>(5,4)</b>	<b>&lt;5,6,2,3,4&gt;</b>
	<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>

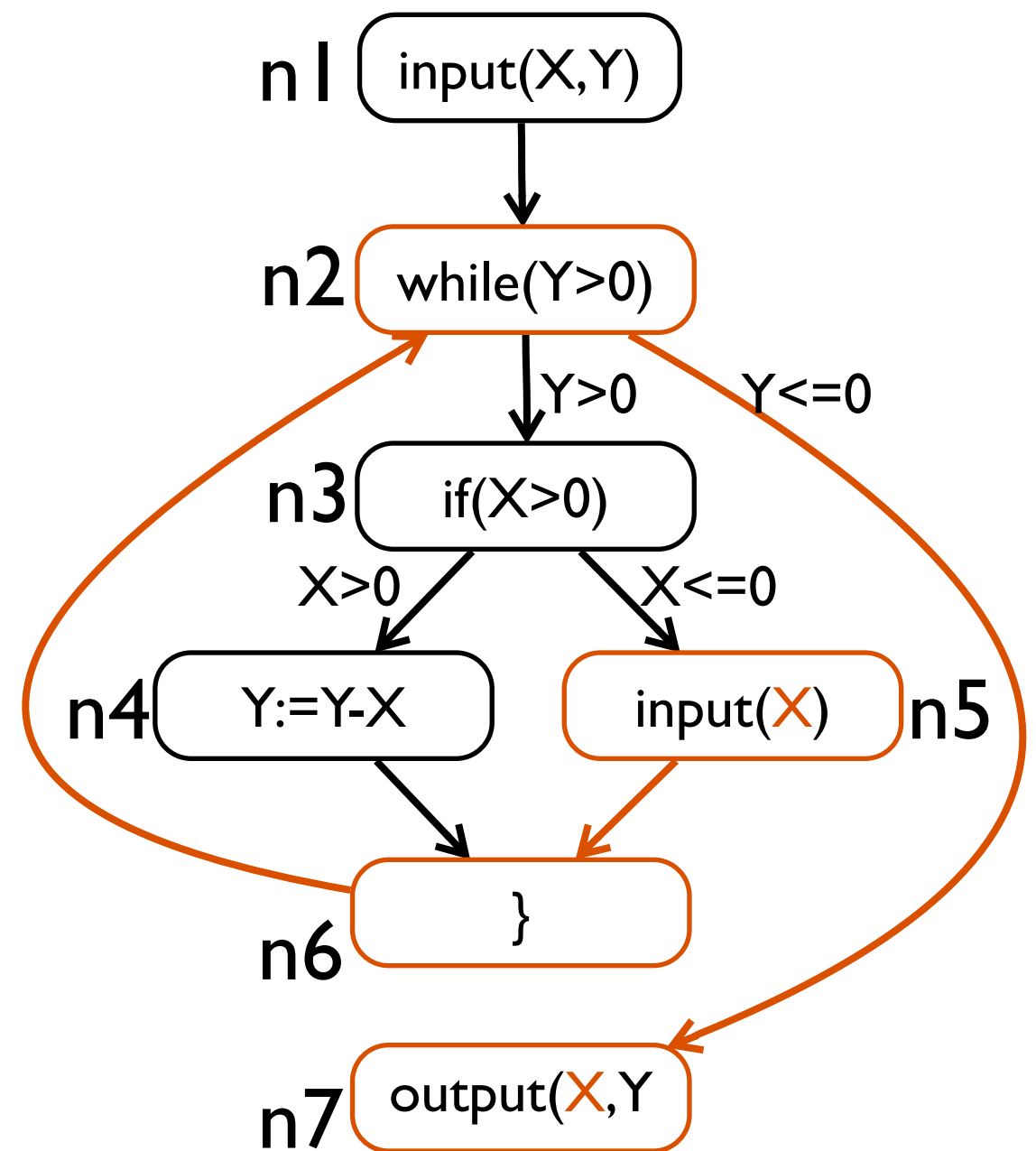
airs – Variable **X**



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

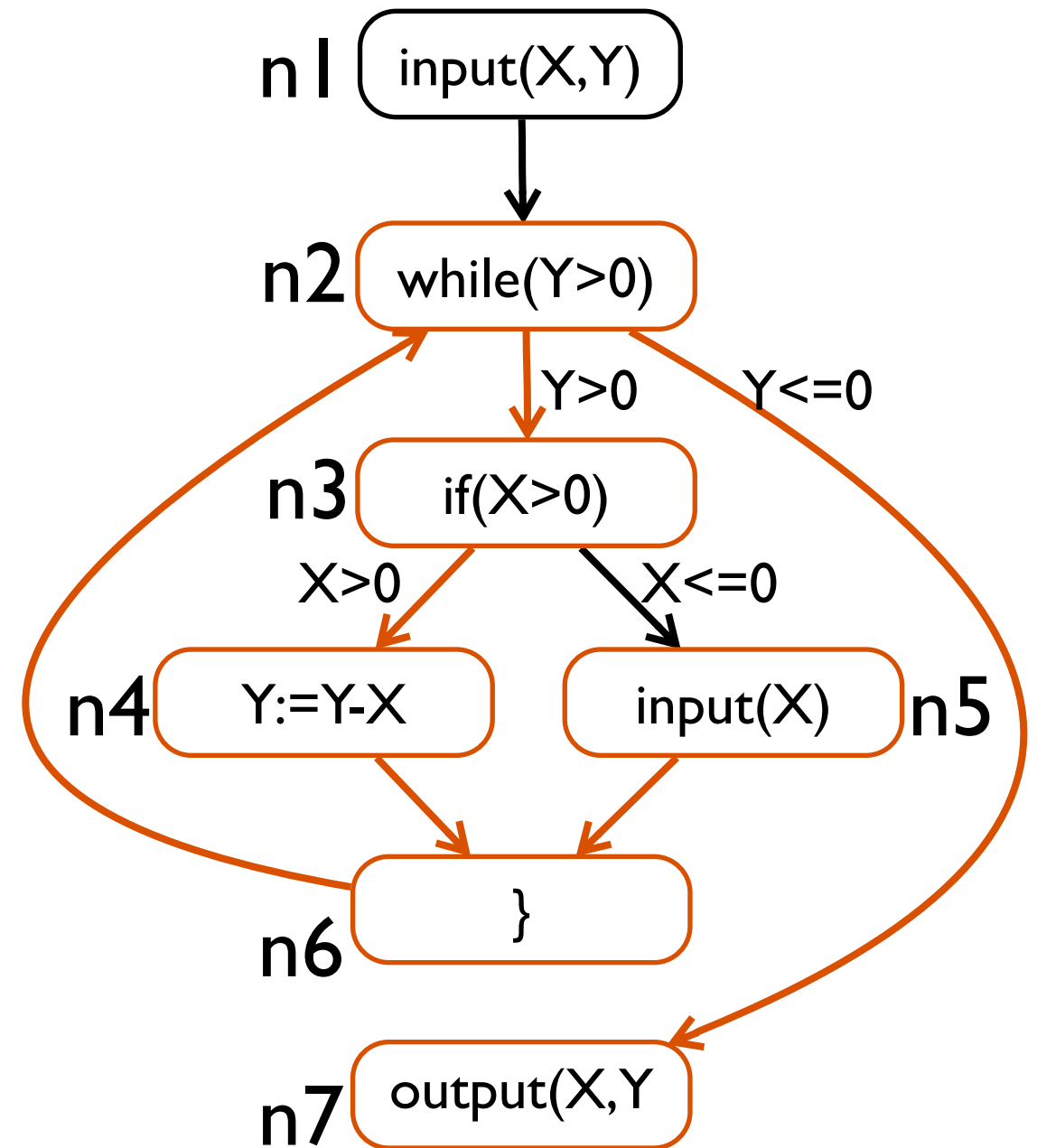
<u>du-pair</u>	<u>path(s)</u>
(1,4)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,7)	<1,2,7>
	<1,2,(3,4,6, 2)*,7>
(1,<3,4>)	<1,2,3,4>
	<1,2,3,4,(6,2,3,4)*>
(1,<3,5>)	<1,2,3,5>
	<1,2,3,5,(6,2,3,5)*>
(5,4)	<5,6,2,3,4>
	<5,6,2,3,4,(6,2,3,4)*>
<b>(5,7)</b>	<b>&lt;5,6,2,7&gt;</b>
	<5,6,2,(3,4,6,2)*,7>
(5,<3,4>)	<5,6,2,3,4>
	<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>

**Infeasible!**



<u>du-pair</u>	<u>path(s)</u>
(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)*>
<b>(5,7)</b>	<5,6,2,7>
	<b>&lt;5,6,2,(3,4,6,2)*,7&gt;</b>
(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>

pairs – Variable **X**



# 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

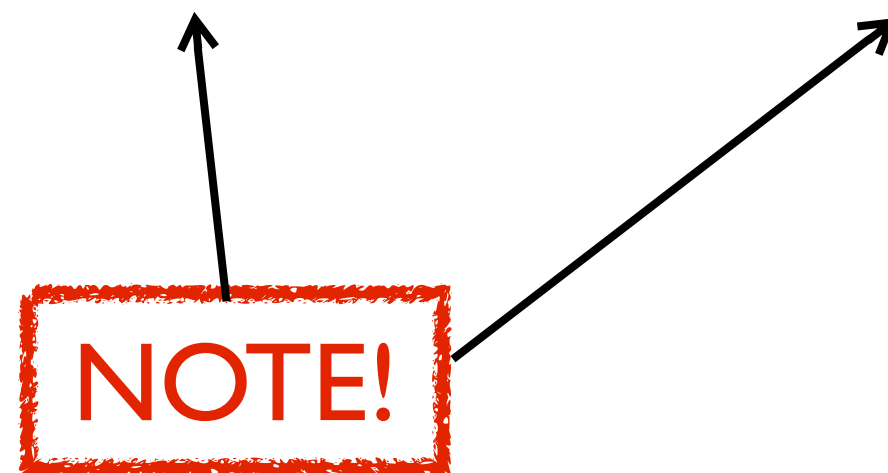
# Simple and Loop-Free Paths

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



# DU-Path

- A path  $\langle n_1, n_2, \dots, n_j, n_k \rangle$  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  $\langle n_1, n_2, \dots, n_j, n_k \rangle$  is a def-clear **simple** path, or
  - there is a **p-use** of **v** at edge  $\langle n_j, n_k \rangle$  and  $\langle n_1, n_2, \dots, n_j \rangle$  is a def-clear **loop-free** path.

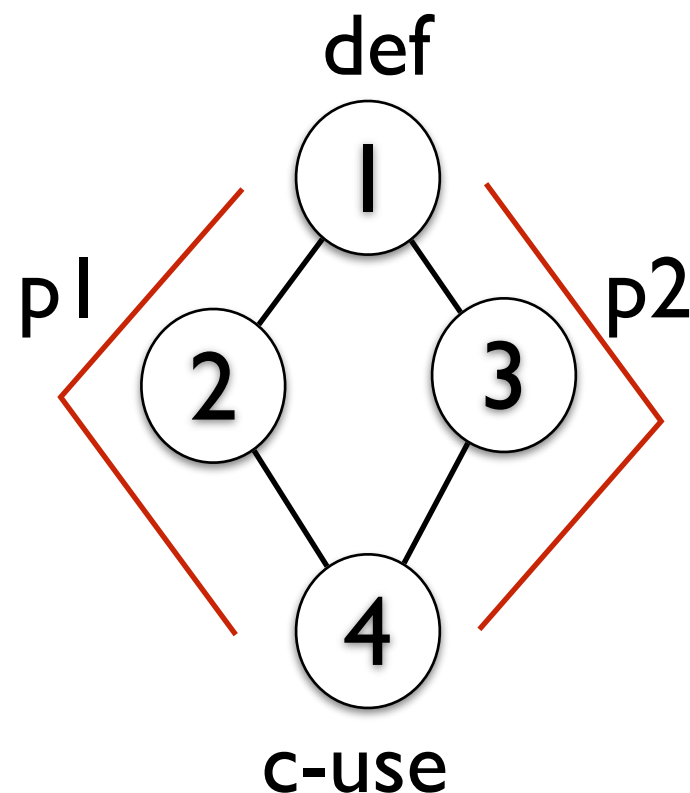


# Identifying DU-Paths

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

# Another Dataflow Test Coverage Criterion

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



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

p1 and p2 together satisfy  
all-du-paths

node 1 is the only def node, and 4  
is the only use node for  $\mathbf{v}$

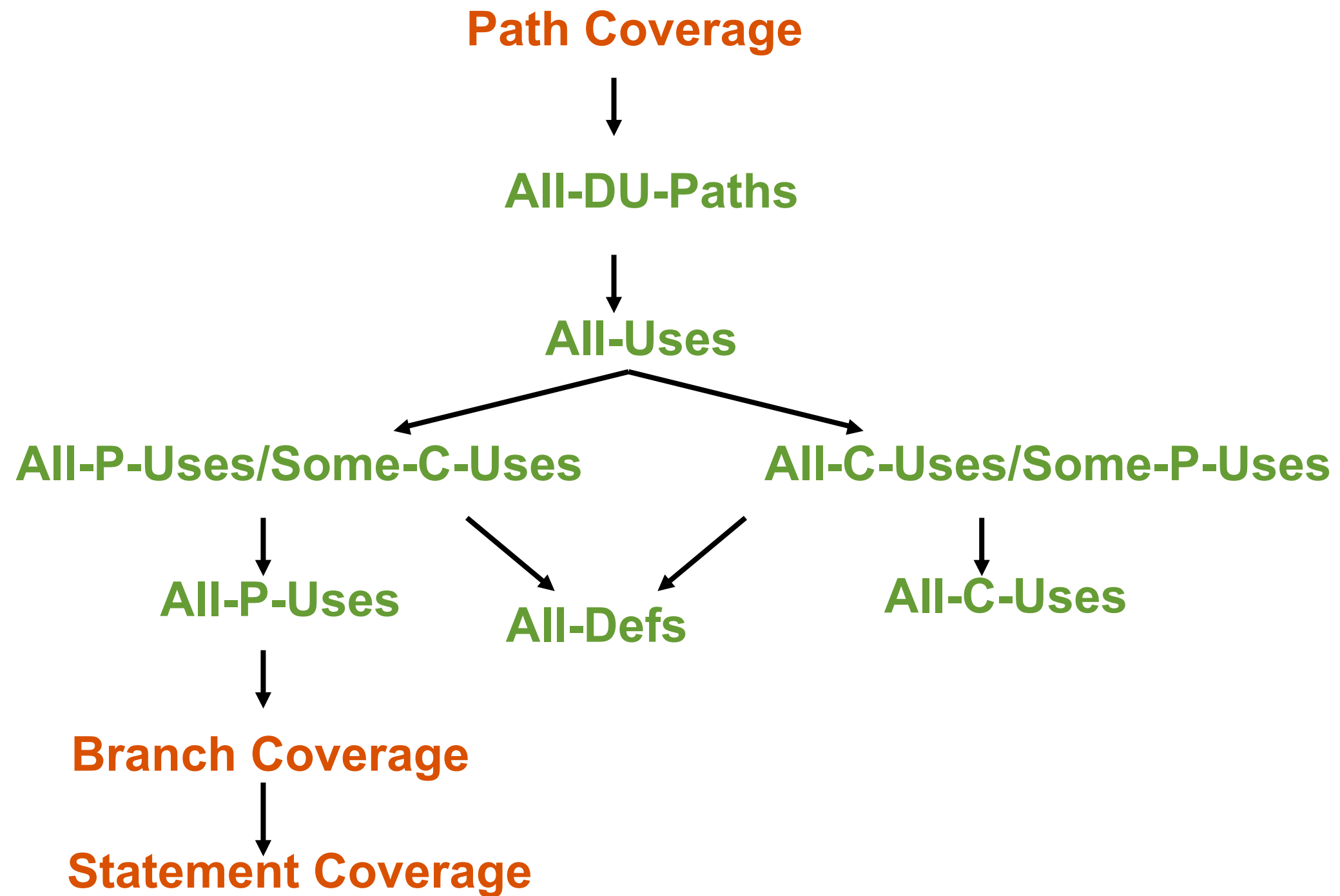
# More Dataflow Test Coverage Criteria

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

# More Dataflow Test Coverage Criteria (2)

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

# Summary



# 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

Thanks