# SEENG3320/6320: Software Verification and Validation

School of Electrical Engineering and Computing

Semester 1, 2020

# Data Flow Coverage

# Motivation

```
begin
    int x,y; float z;
    input(x,y);
    z=0;
    if(x!=0)
      z=z+y;
    else z=z-y;
    if(y!=0)
      z=z/x;
    else z=z*x;
    output(z):
end
```

Any bug in this program?

# Motivation

```
1 begin
2       int x,y; float z;
3       input(x,y);
4       z=0;
5       if(x!=0)
6         z=z+y;
7       else z=z-y;
8       if(y!=0)  //should be y!=0 and x!=0
9         z=z/x;
10      else z=z*x;
11      output(z):
12 end
```

| Test case | x | y | z |
|---|---|---|---|
| t1 | 0 | 0 | 0.0 |
| t2 | 1 | 1 | 1.0 |

- 100% statement and decision coverage
- Still cannot discover the bug at line 8

# Motivation

```
begin
    int x,y; float z;
    input(x,y);
    z=0;
    if(x!=0)
      z=z+y;
    else z=z-y;
    if(y!=0)  //should be y!=0 and x!=0
      z=z/x;
    else z=z*x;
    output(z):
end
```

| Test Case | x | y | z |
|---|---|---|---|
| t1 | 0 | 0 | 0.0 |
| t2 | 1 | 1 | 1.0 |
| t3 | 0 | 1 | NaN |
| t4 | 1 | 0 | 1.0 |

# 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

# 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:

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

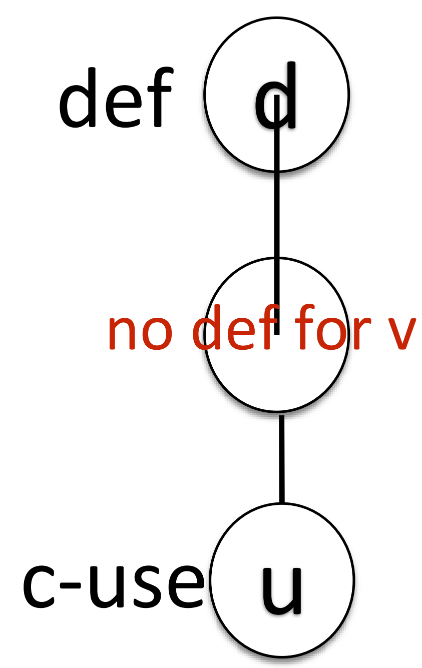
There is a p-use of x and a c-use of y

# Variable use

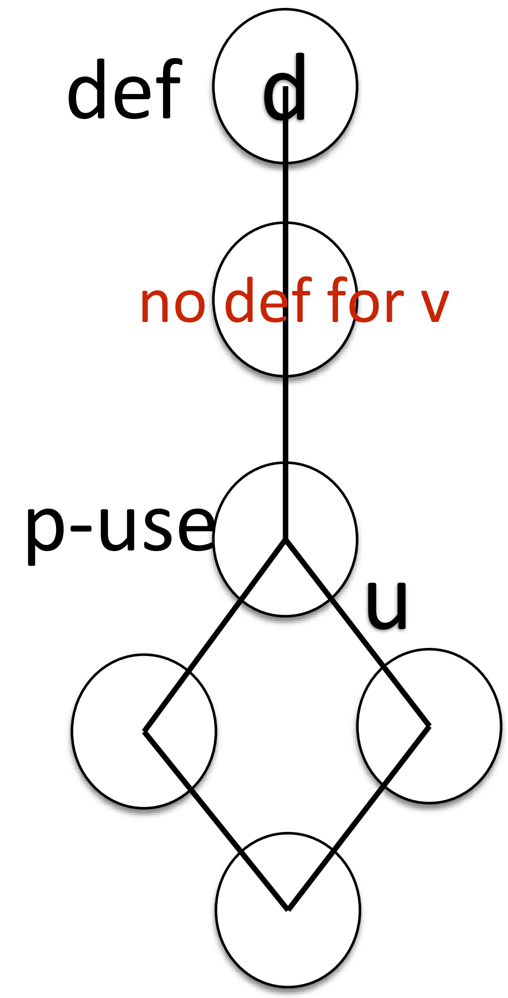
- A variable can also be used and then re-defined in a single statement when it appears:
  - on both sides of an assignment statement
    - *e.g.,* **y** = **y** + x
  - as an call-by-reference parameter in a subroutine call
    - *e.g.,* increment( &**y** )

# 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**
  - 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



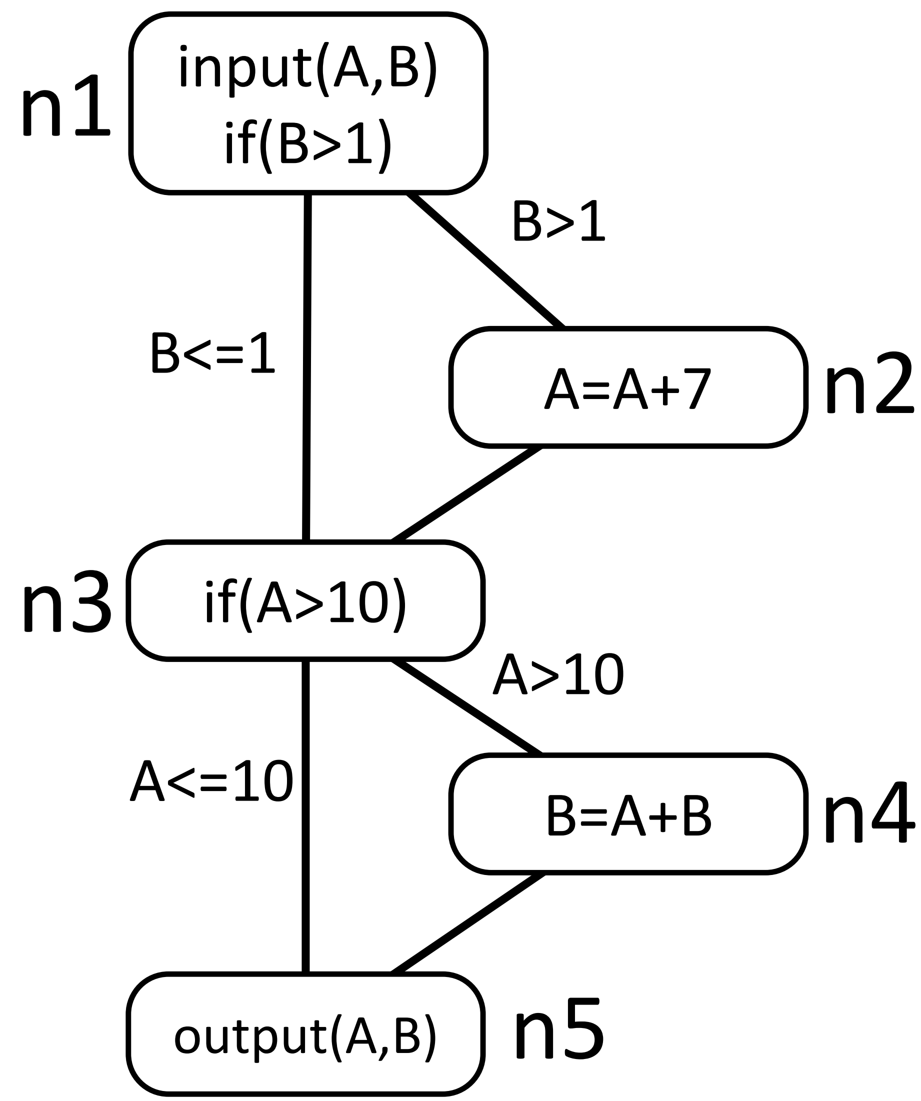

c-use example




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)
```

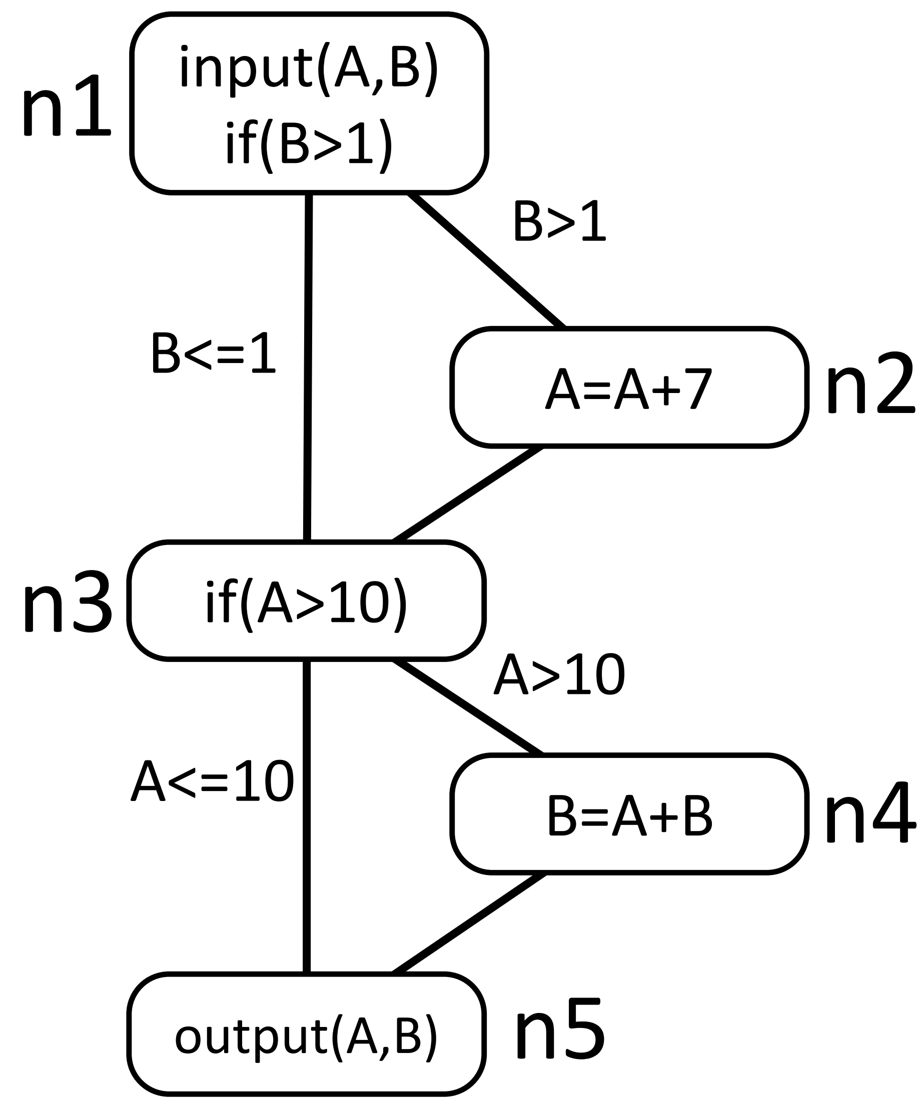


# Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to *basic block* coverage or *node coverage*
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage <-> 100% statement coverage
  - A test case that improves one will improve the other

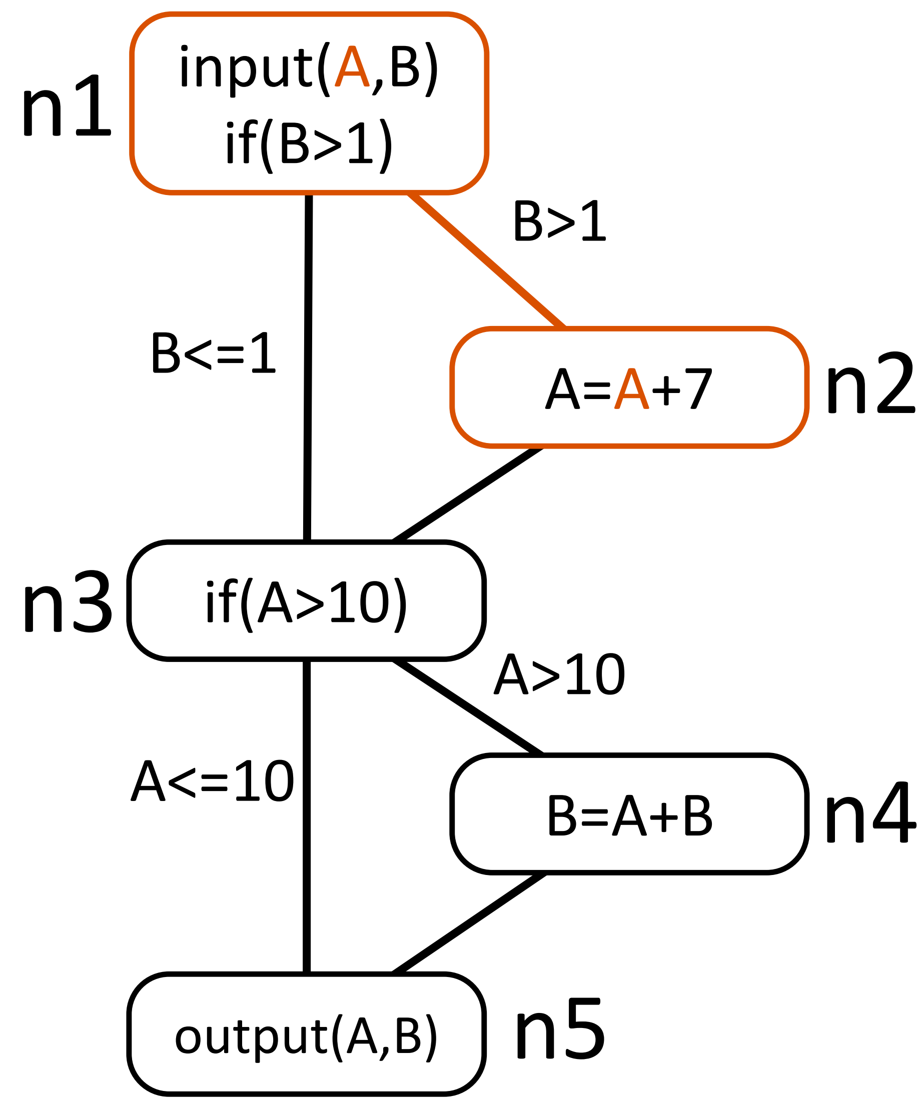
# Identifying du-pairs – variable **A**

| 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> |



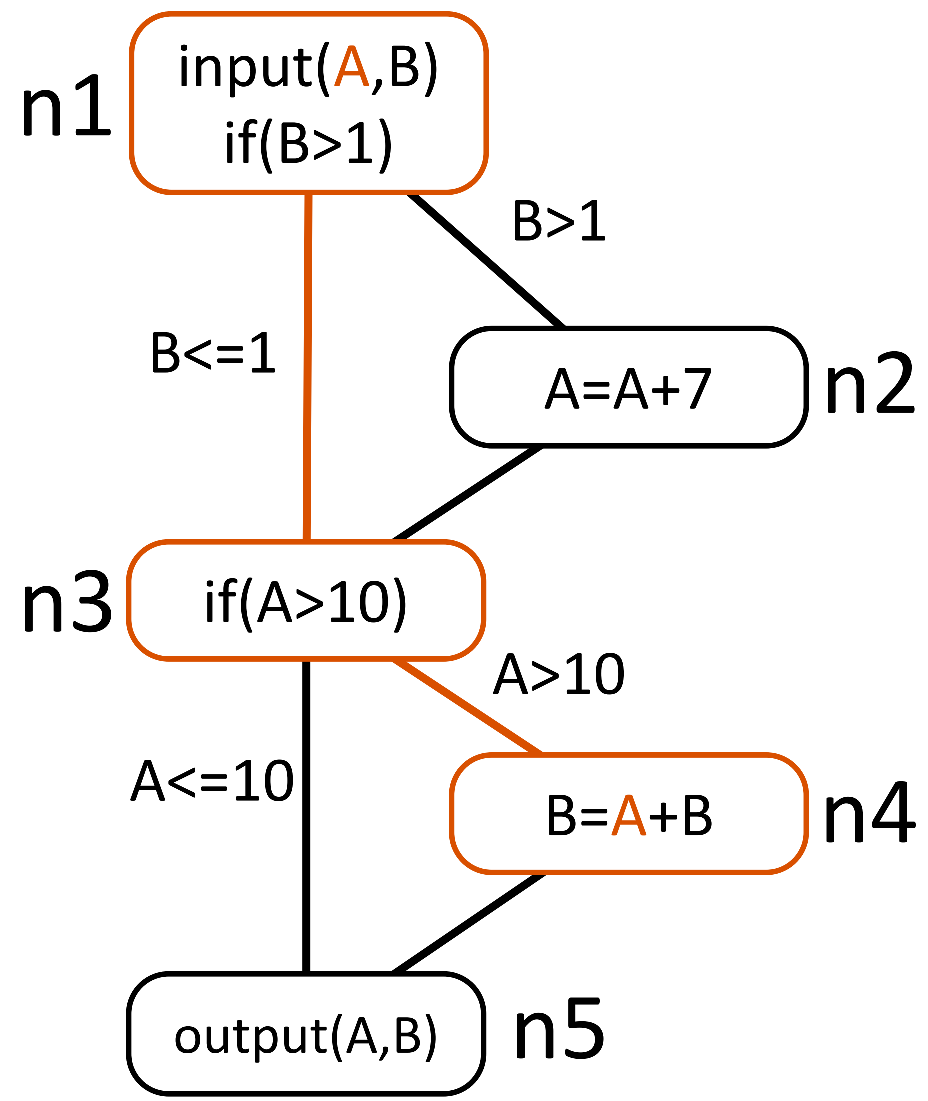
# Identifying du-pairs – variable **A**

| 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> |



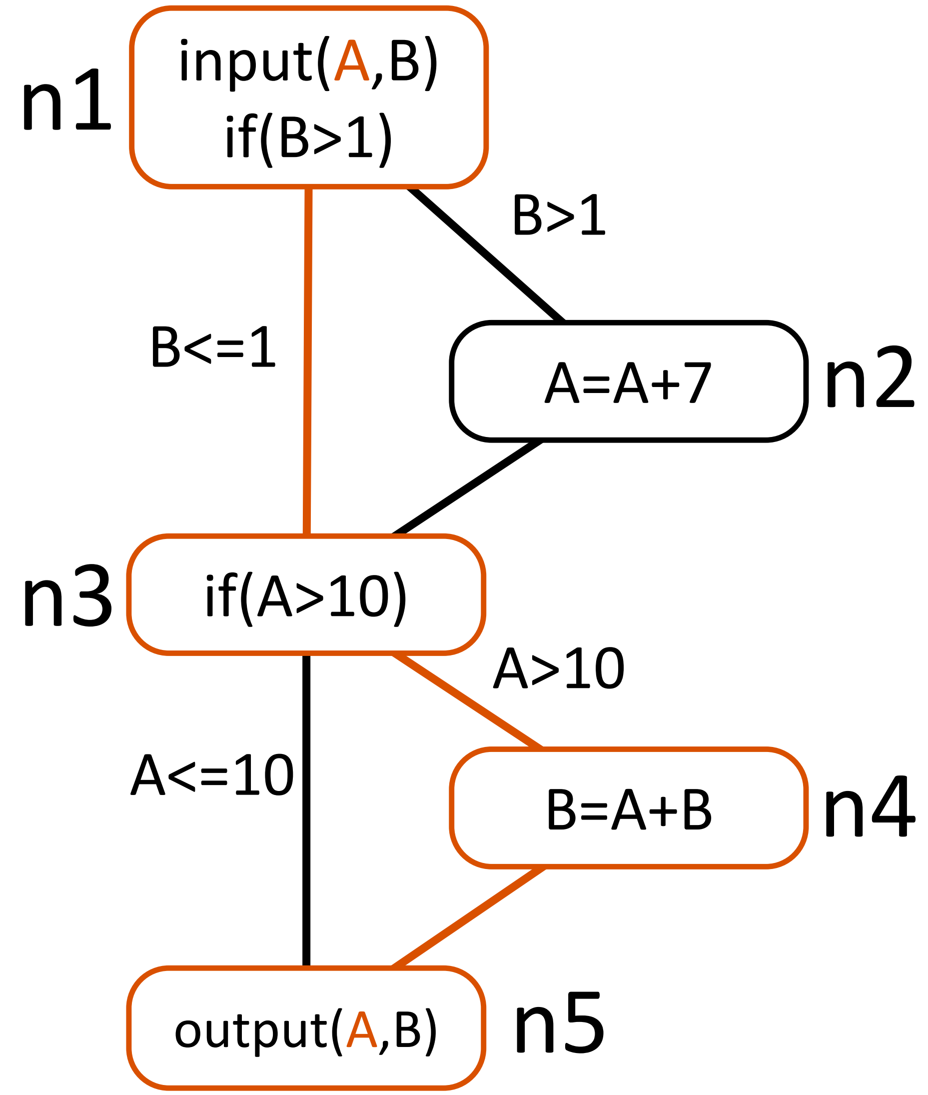
# Identifying du-pairs – variable **A**

| 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> |



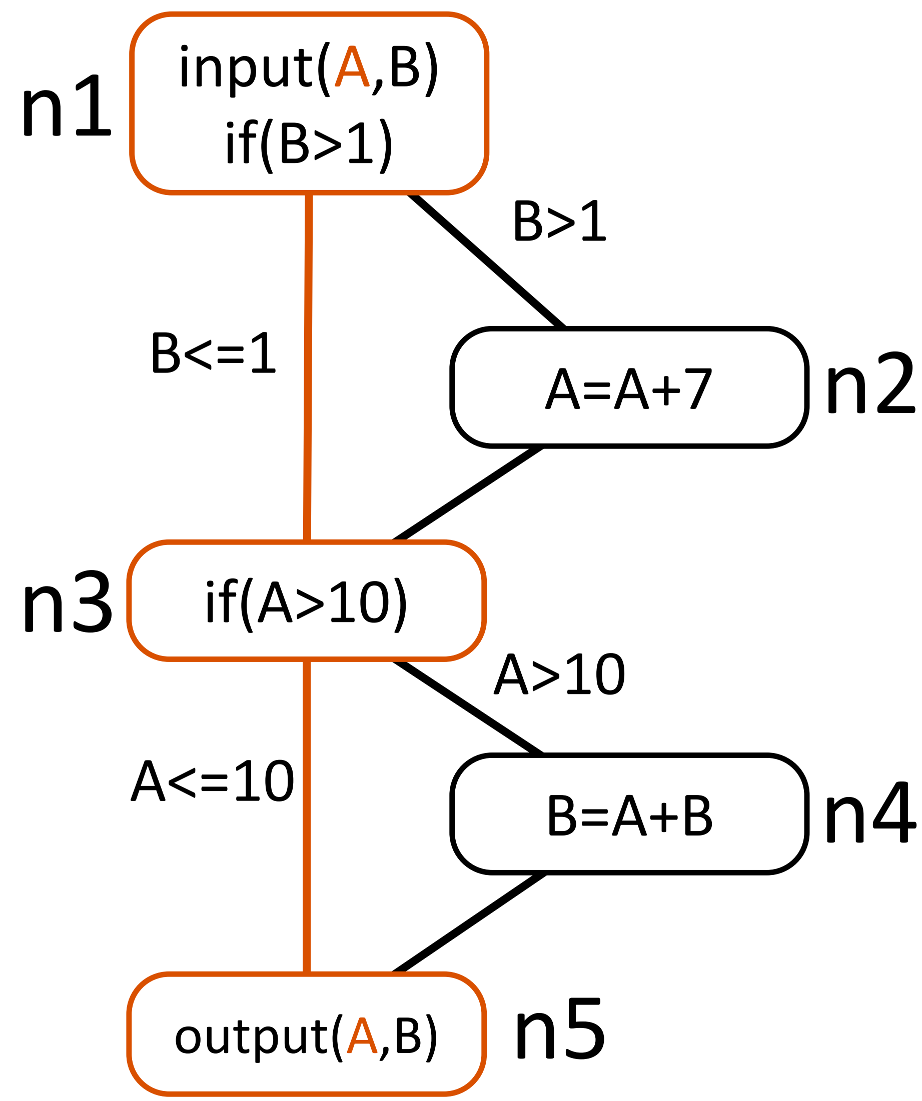
# Identifying du-pairs – variable **A**

| 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> |



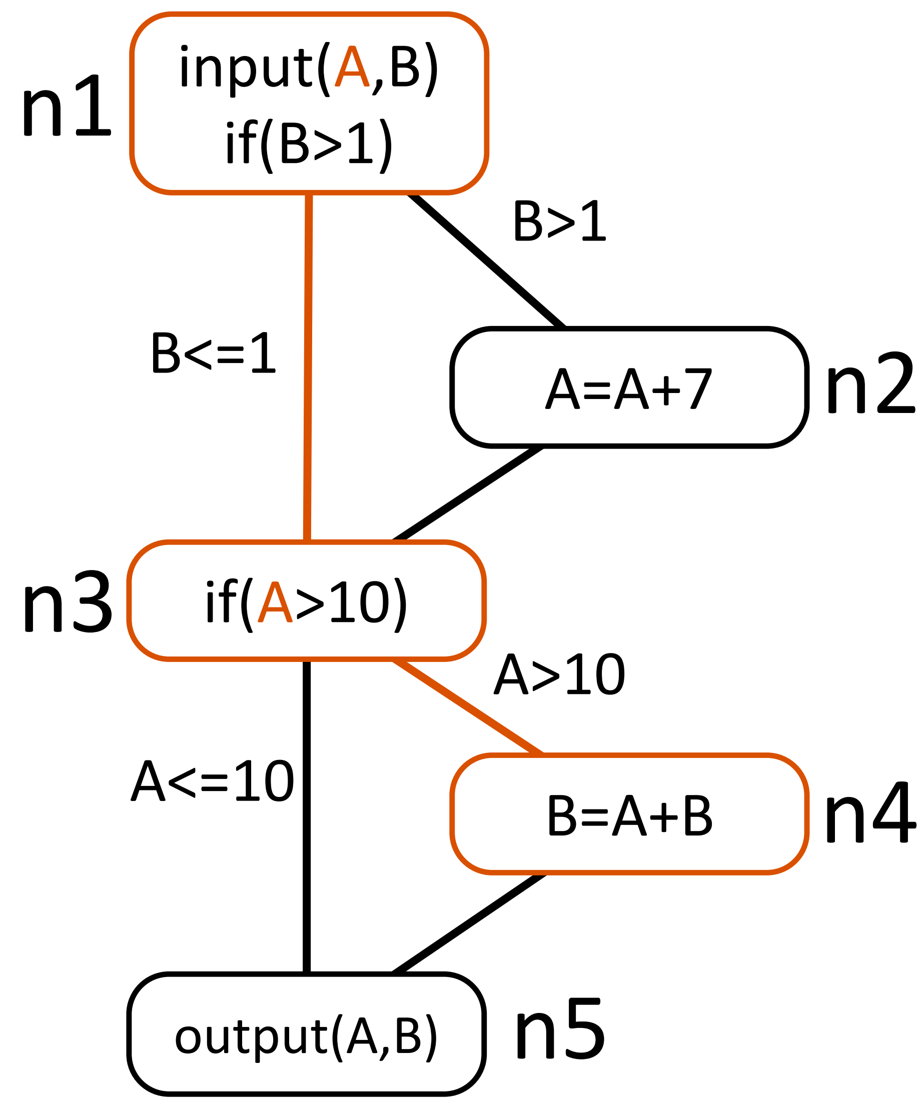
# Identifying du-pairs – variable **A**

| 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> |



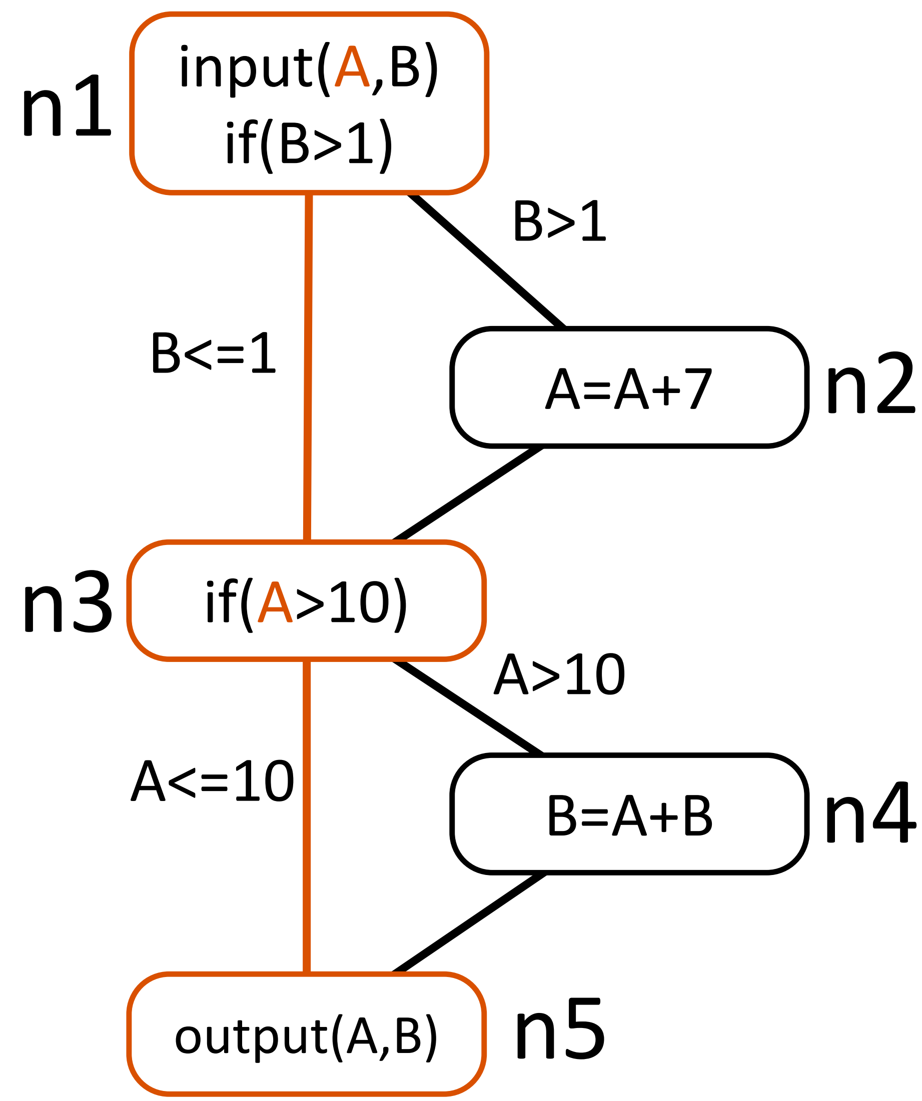
# Identifying du-pairs – variable **A**

| **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> |



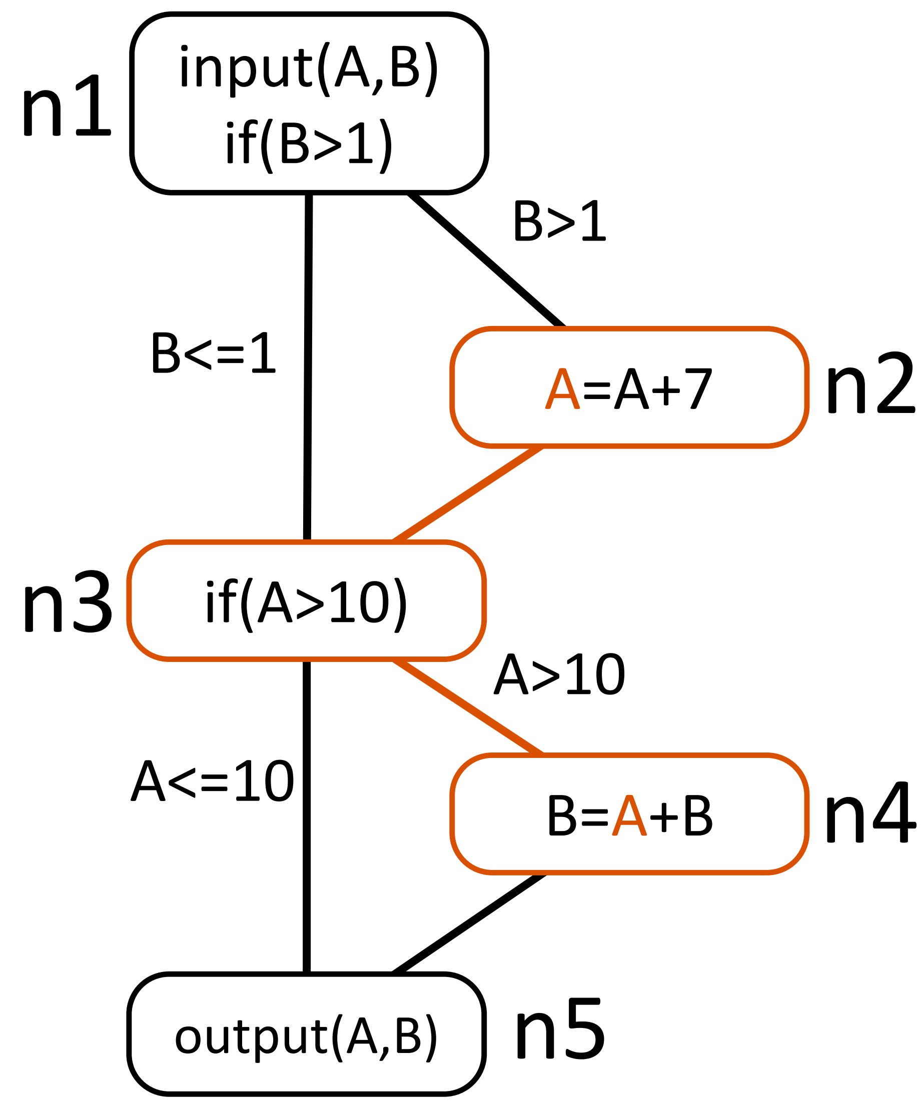
# Identifying du-pairs – variable **A**

| 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> |



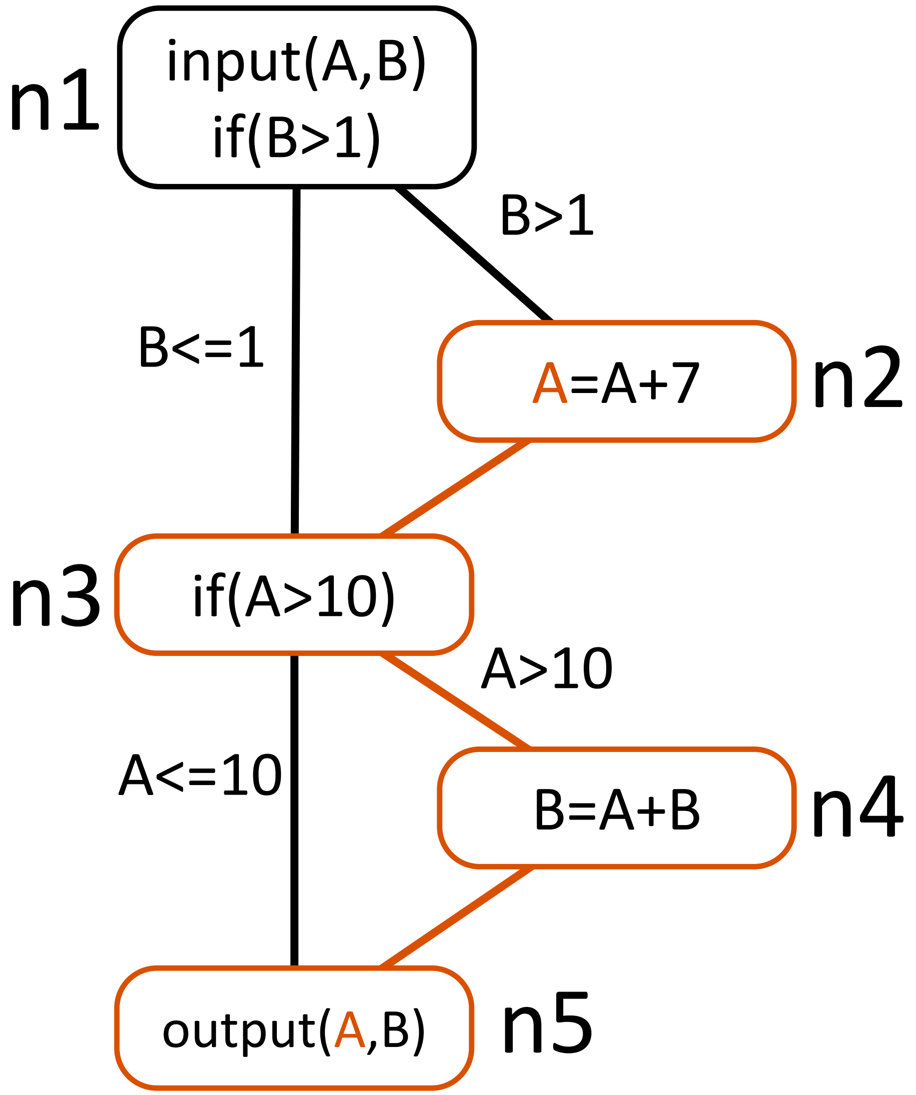
# Identifying du-pairs – variable **A**

| 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> |



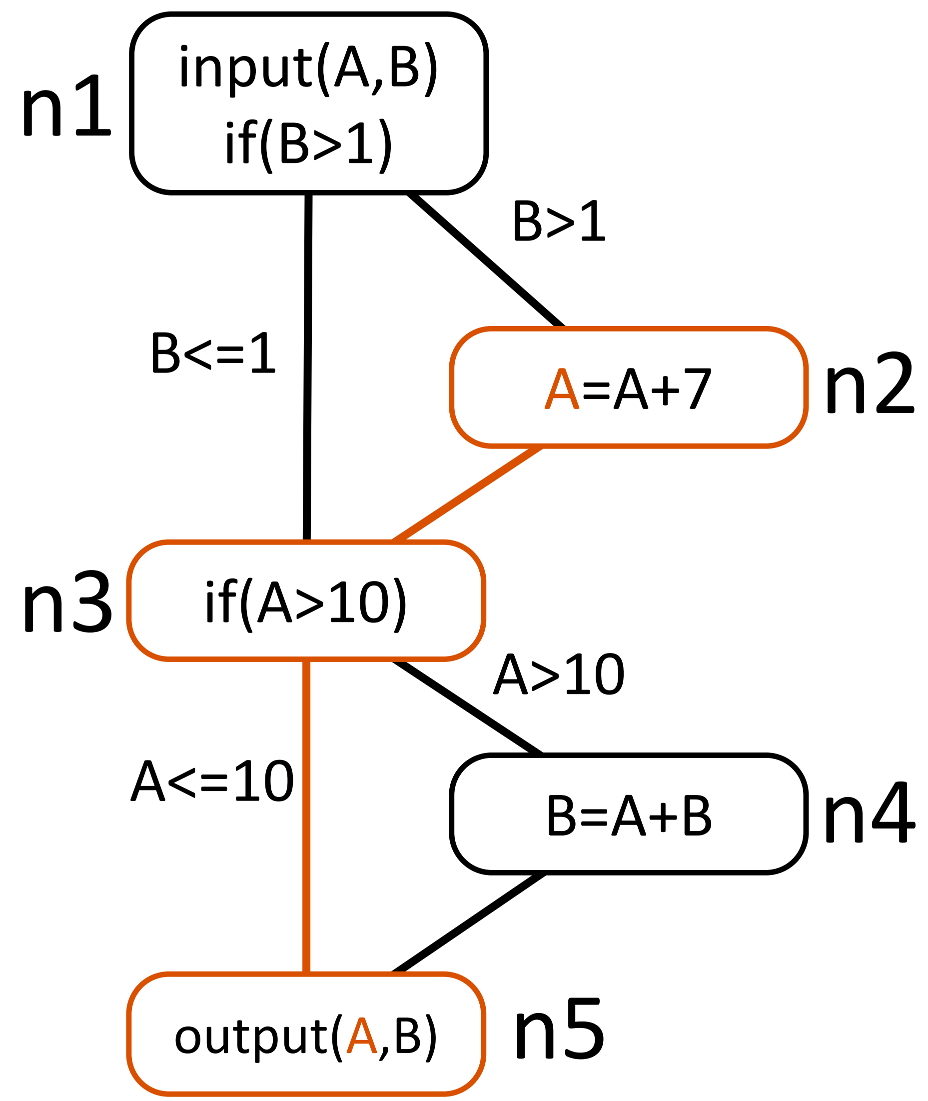
# Identifying du-pairs – variable **A**

| 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> |



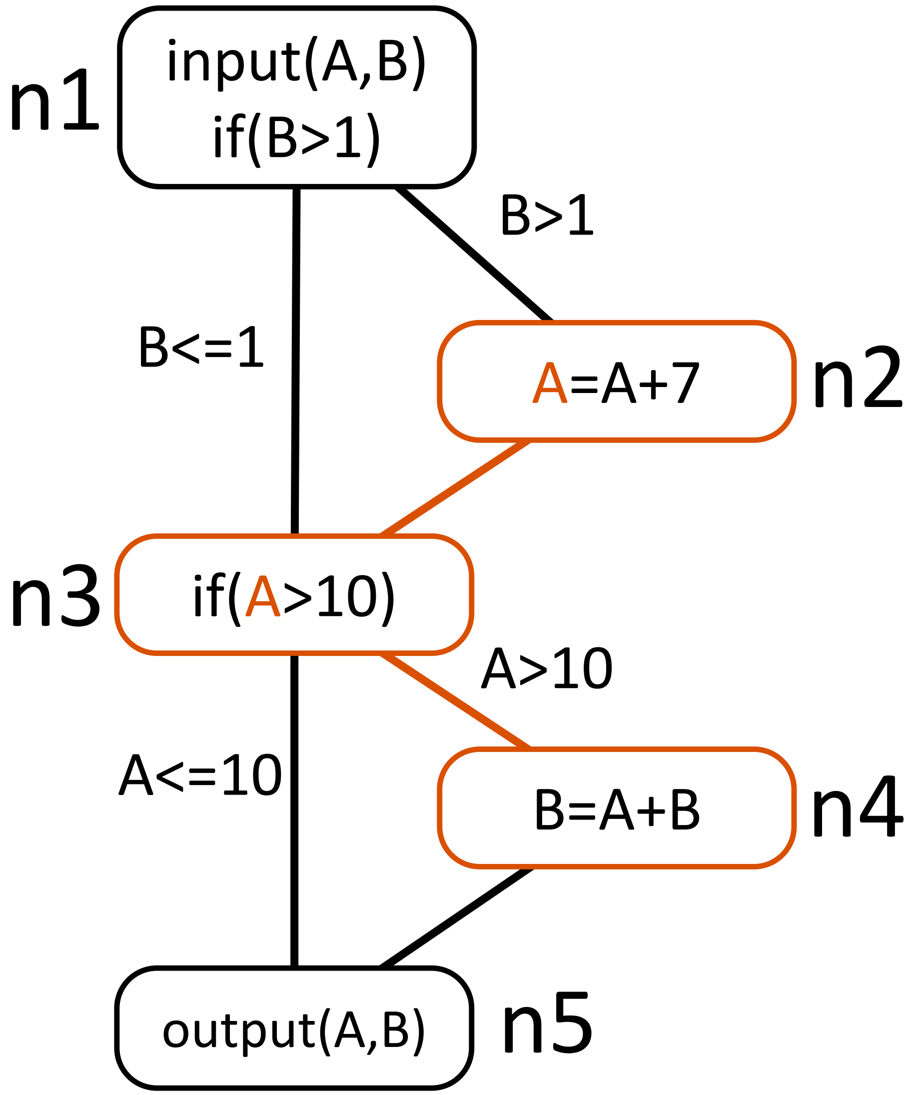
# Identifying du-pairs – variable **A**

| 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> |



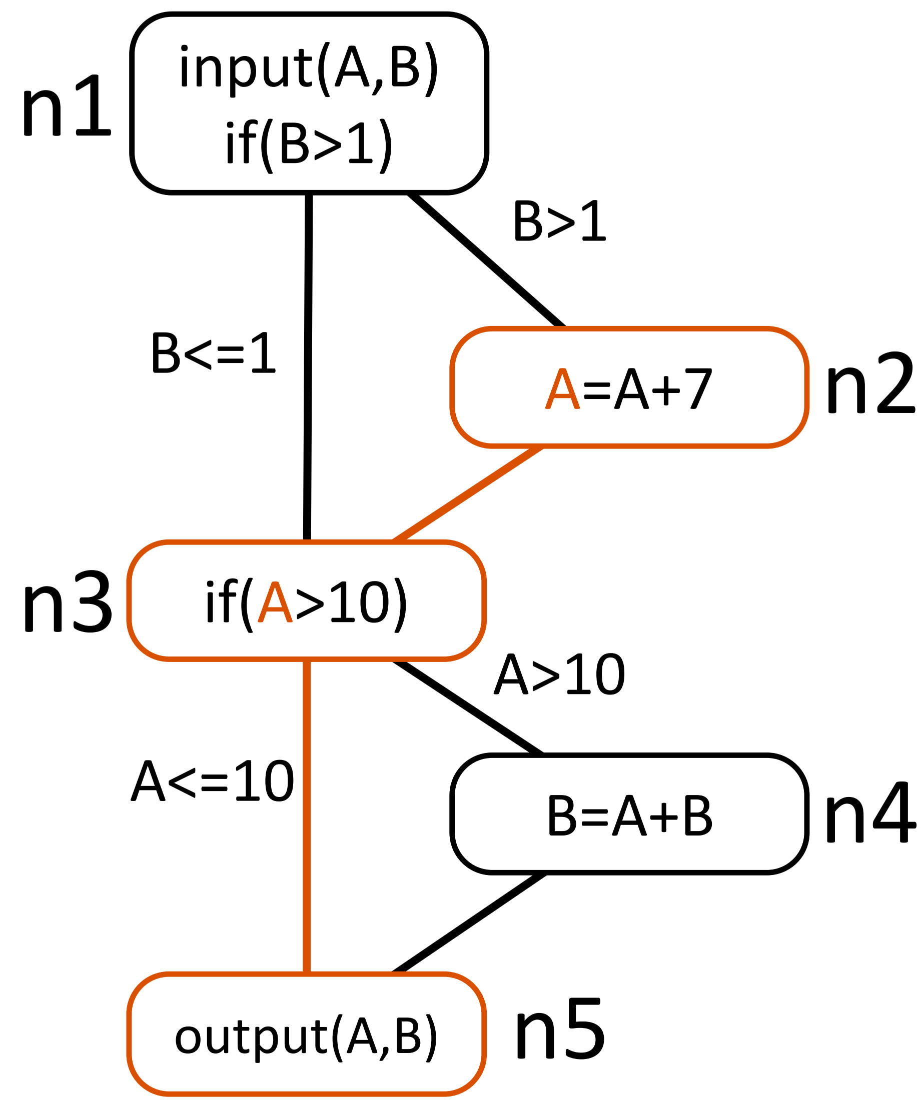
# Identifying du-pairs – variable **A**

| 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> |



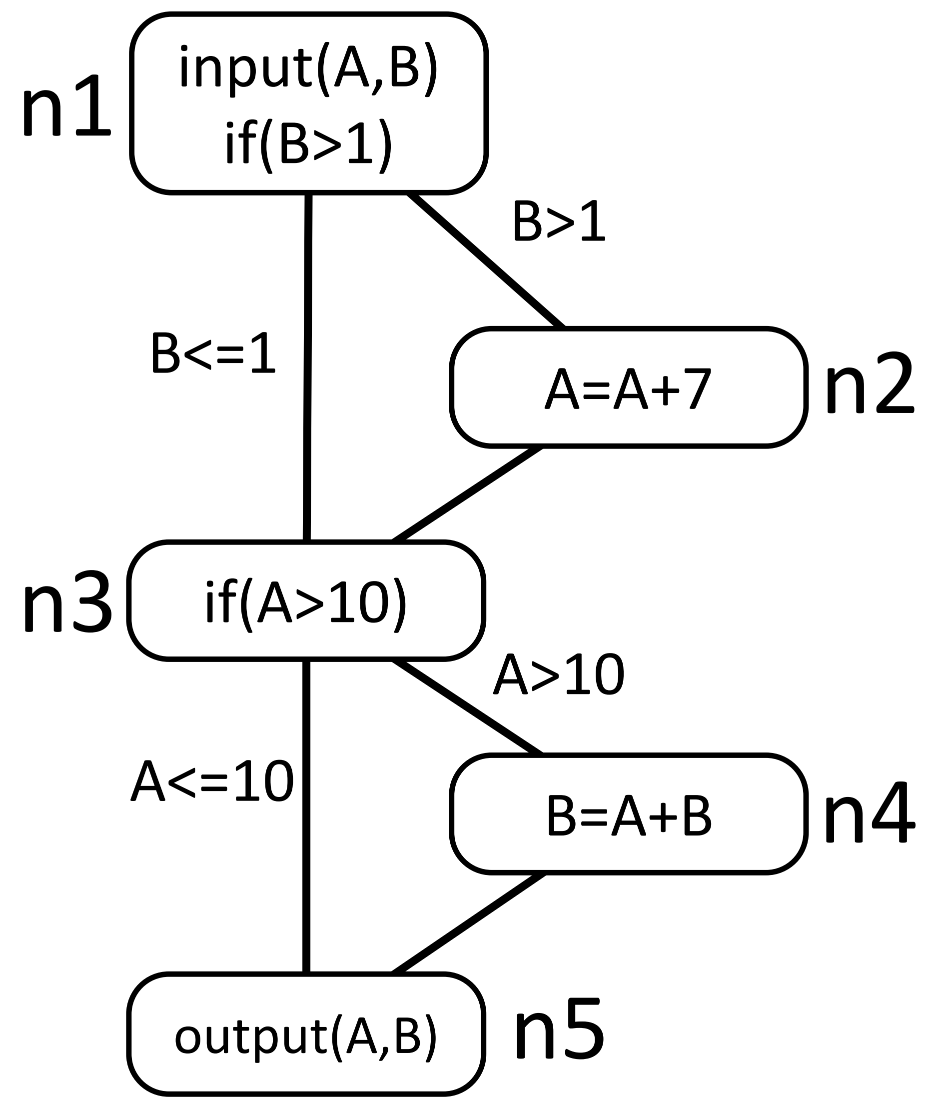
# Identifying du-pairs – variable **A**

| 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> |



# Dataflow test coverage criteria

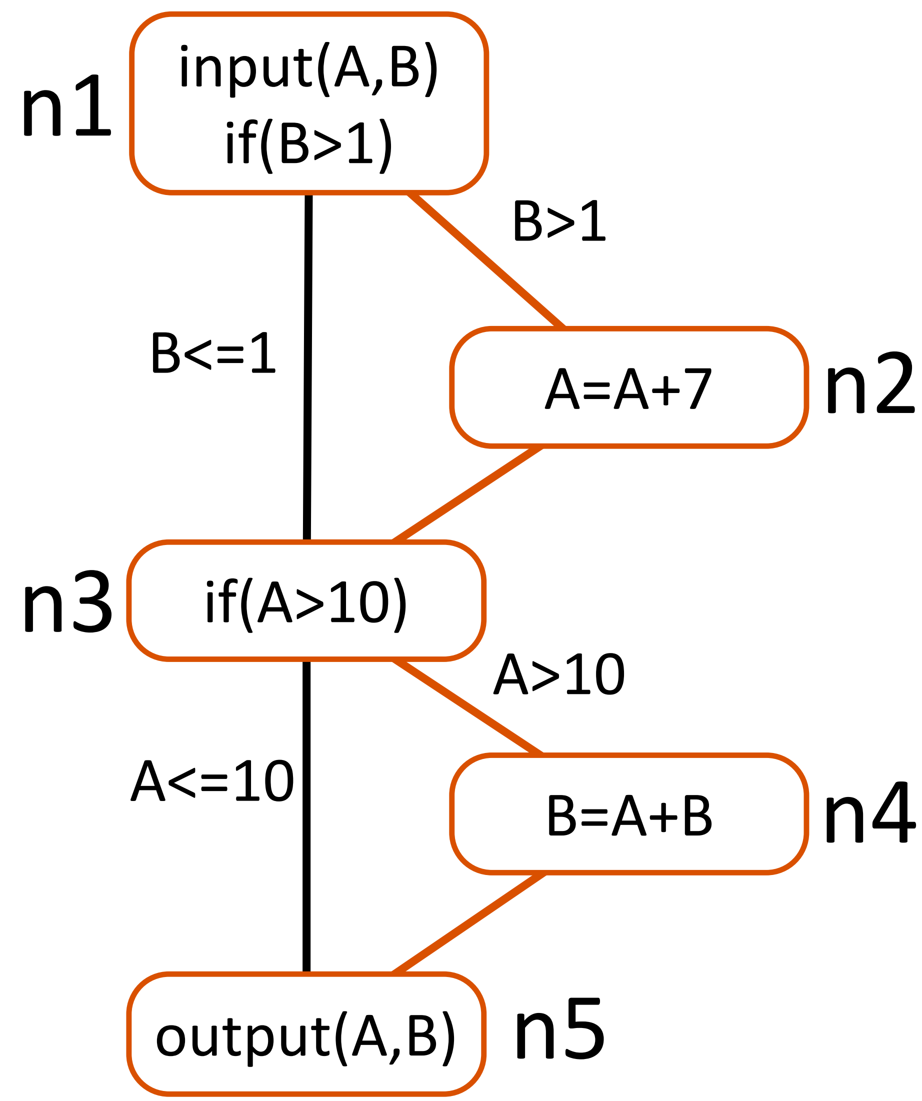
- **All-Defs**
  - for every program variable **v**, at least one def-clear path from every definition of **v** to at least one c-use or one p-use of **v** must be covered

# 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?

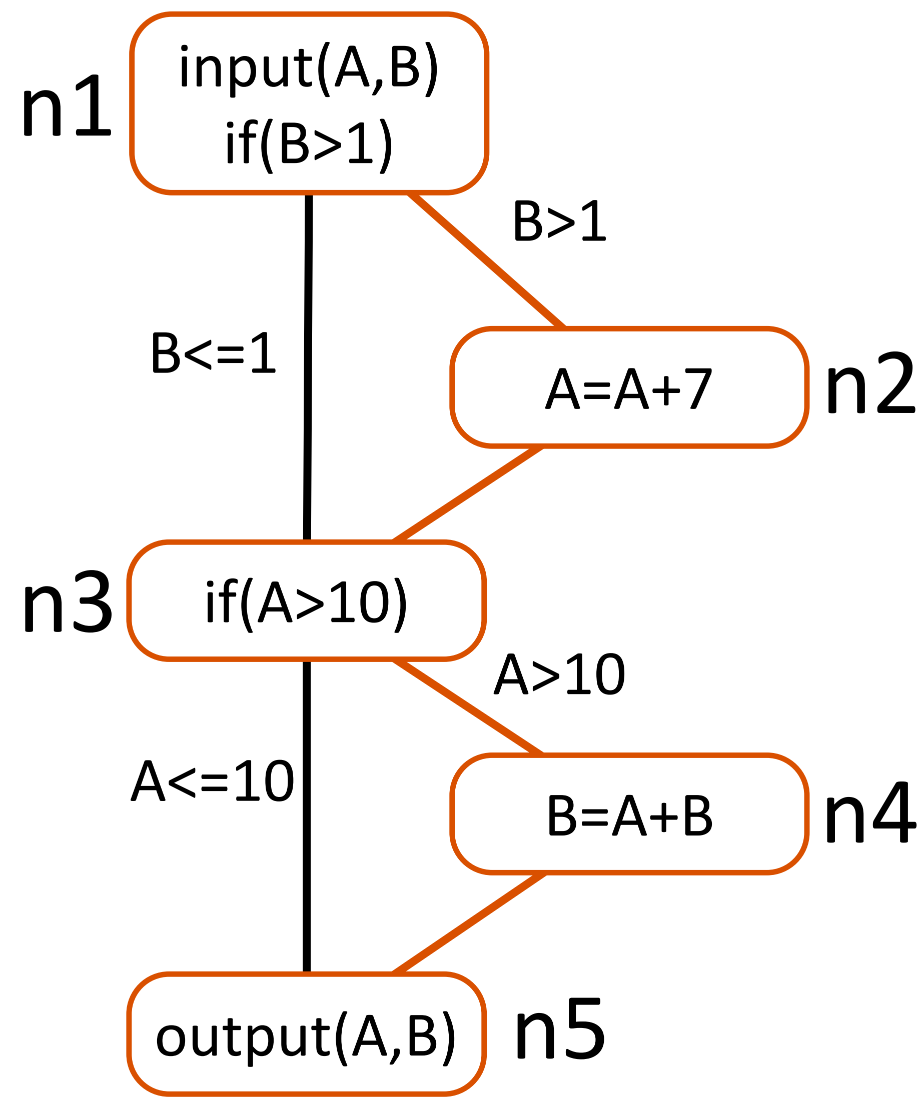
# Def-clear paths subsumed by <1,2,3,4,5> for variable **A**

| 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

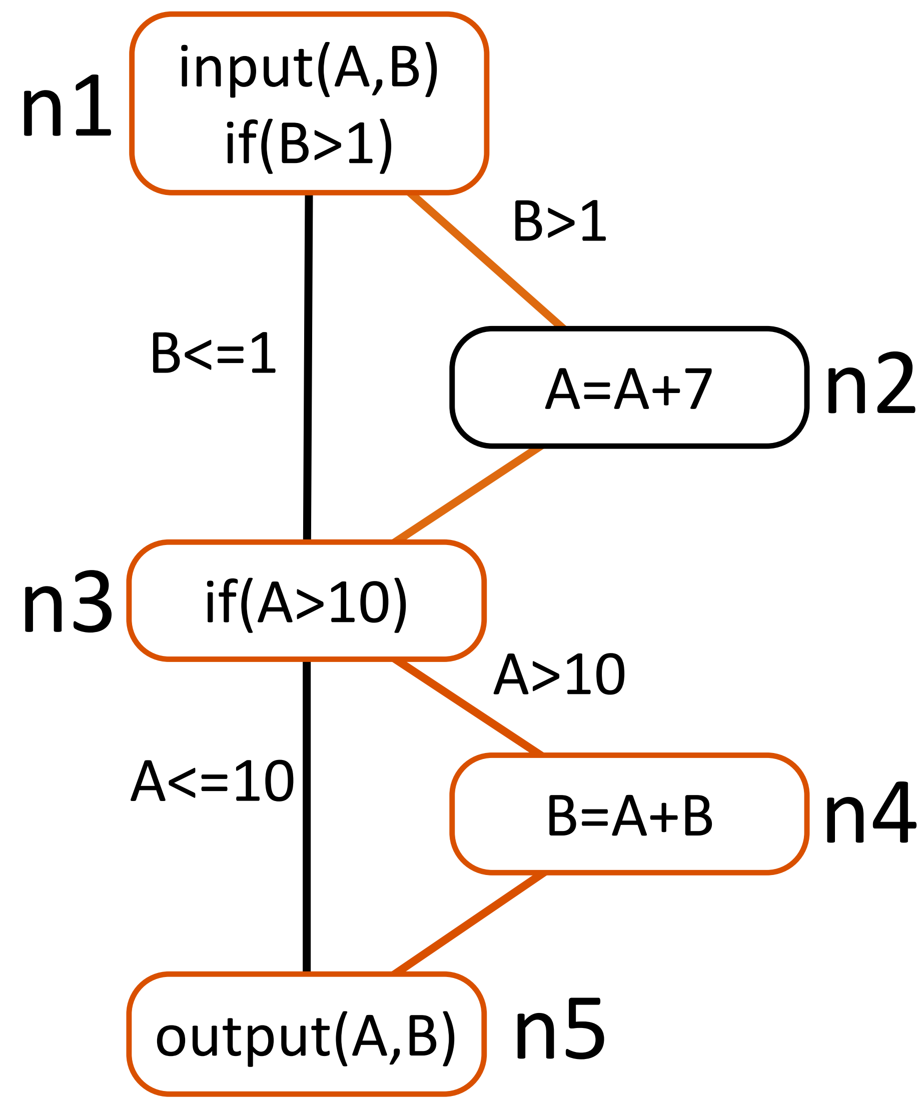
- **All-P-Uses**:
  for every program variable **v**, at least one def-clear path from every definition of **v** to every p-use of **v** must be covered
- **All-C-Uses**:
  for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use of **v** must be covered

# 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 the test cases executing paths:
  - t1: <1,2,3,4,5>
  - t2: <1,3,4,5>
  - t3: <1,2,3,5>
- Do all three test cases provide All-Uses coverage?

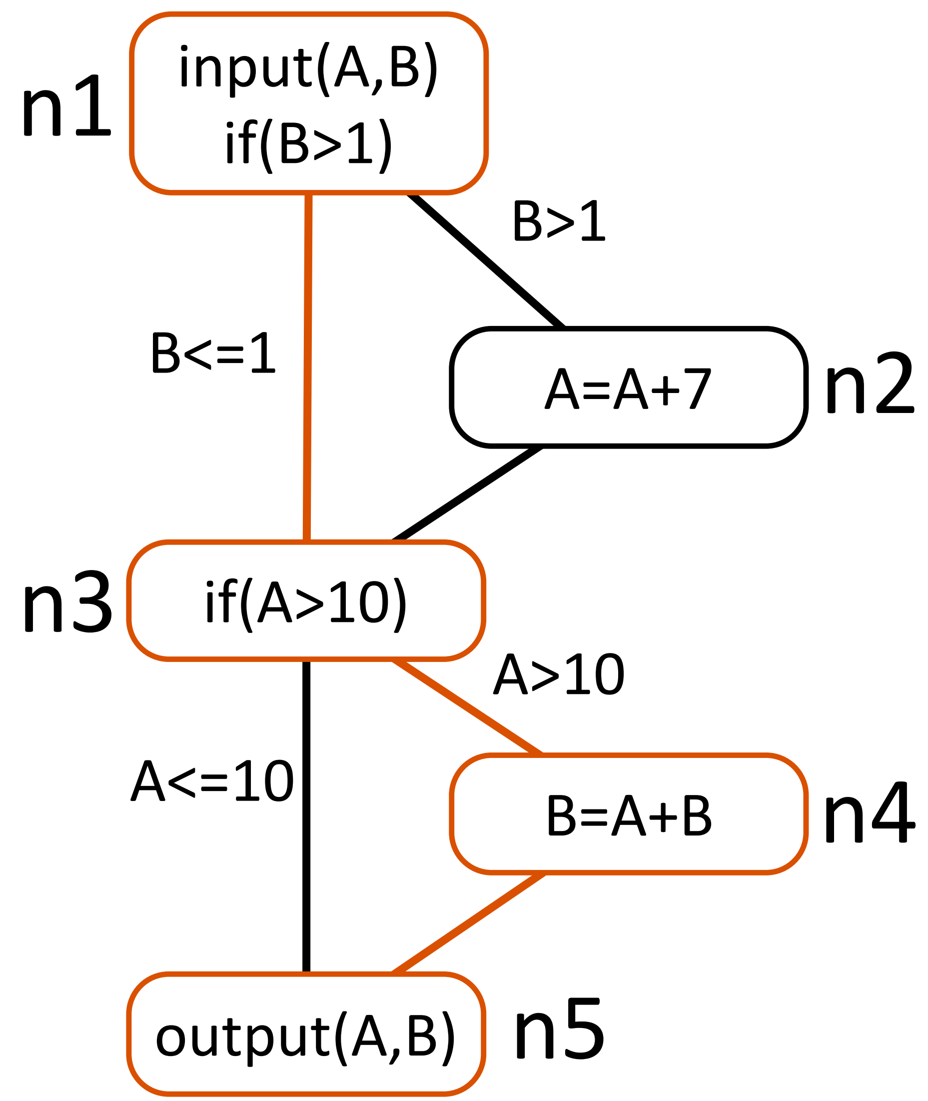
# Def-clear paths subsumed by <1,2,3,4,5> for variable **A**

| 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> |



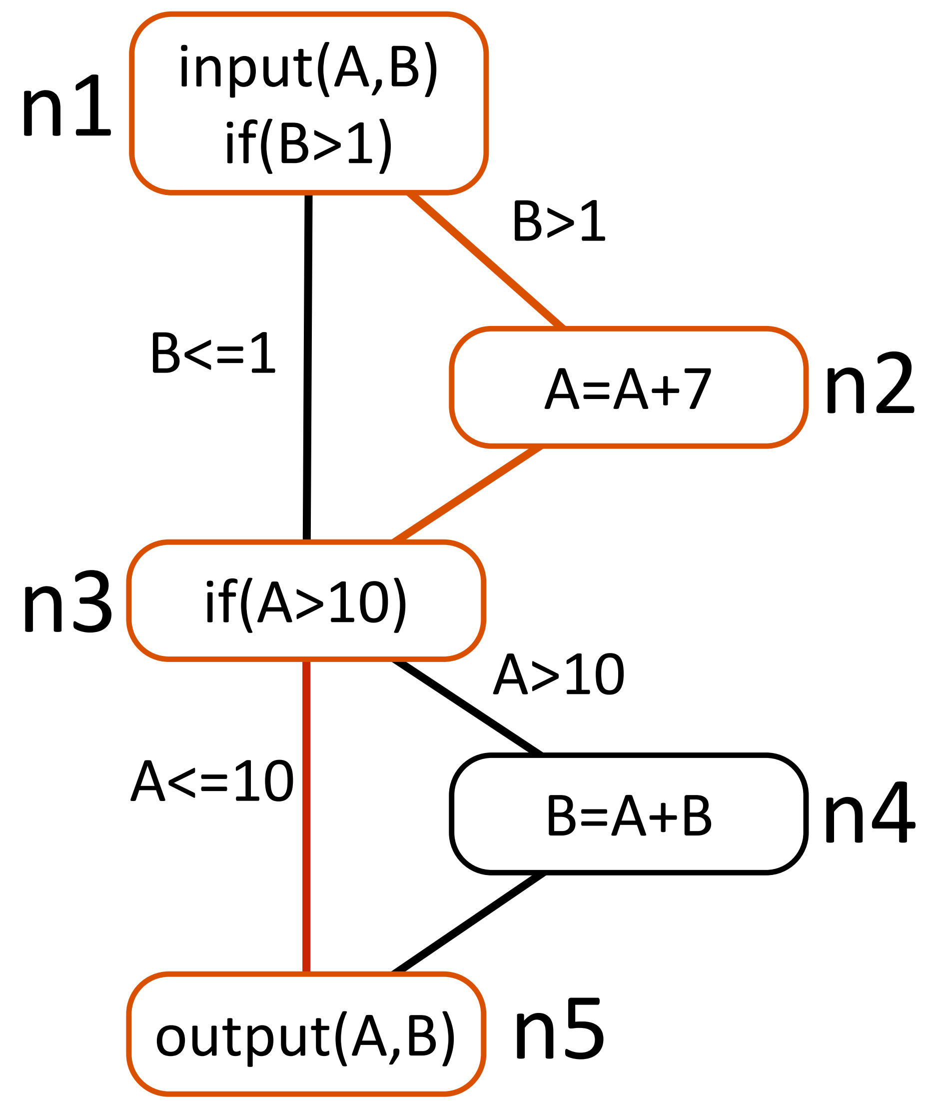
# Def-clear paths subsumed by <1,3,4,5> for variable **A**

| 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> |



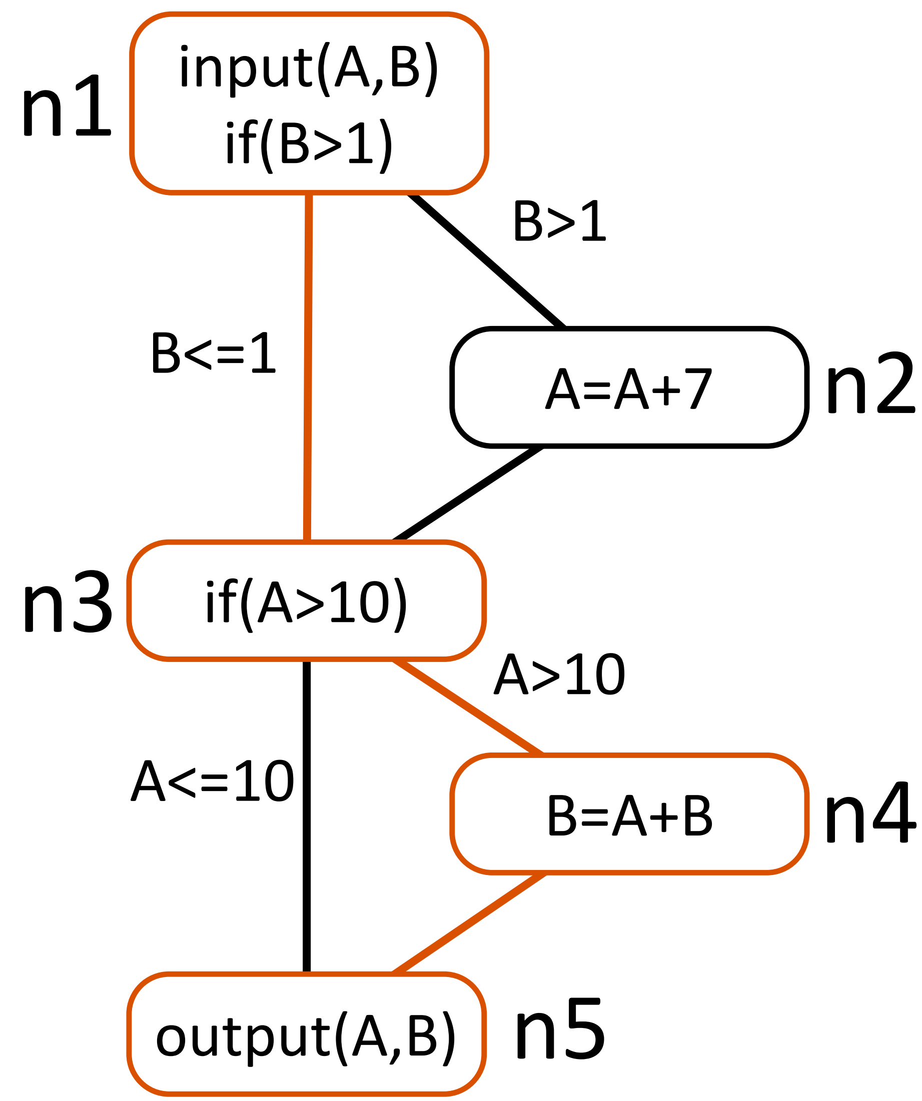
# Def-clear paths subsumed by <1,2,3,5> for variable **A**

| 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> ✓ |



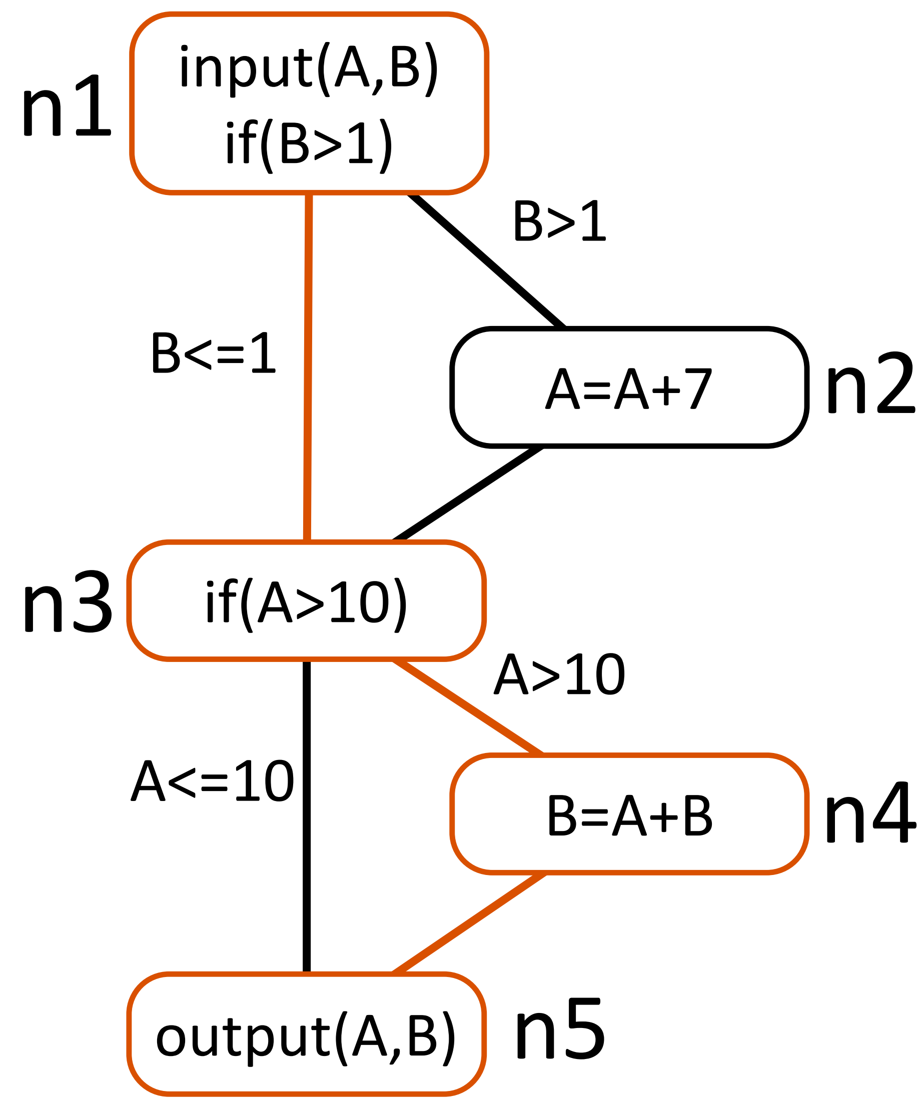
# Def-clear paths subsumed by <1,2,3,4,5> for variable **B**

| 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> ✓ |



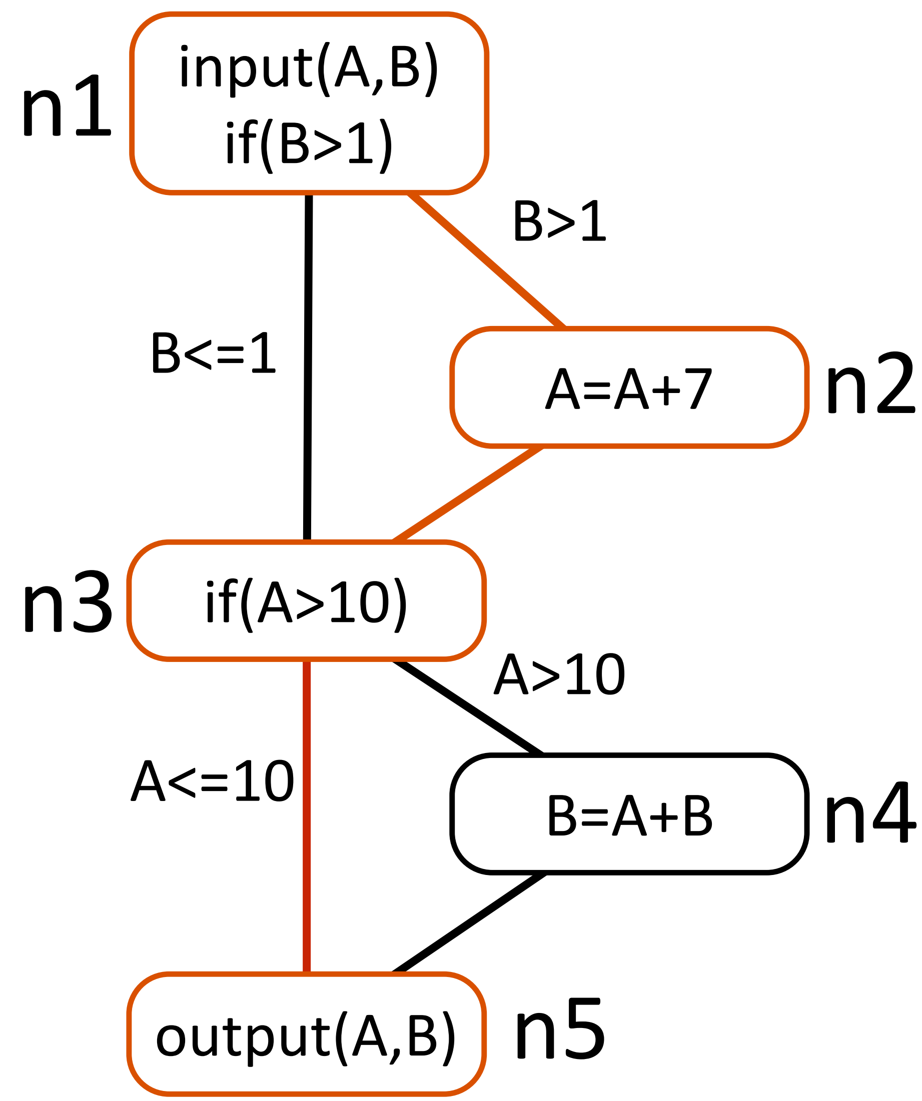
# Def-clear paths subsumed by <1,3,4,5> for variable **B**

| 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> ✓ ✓ |



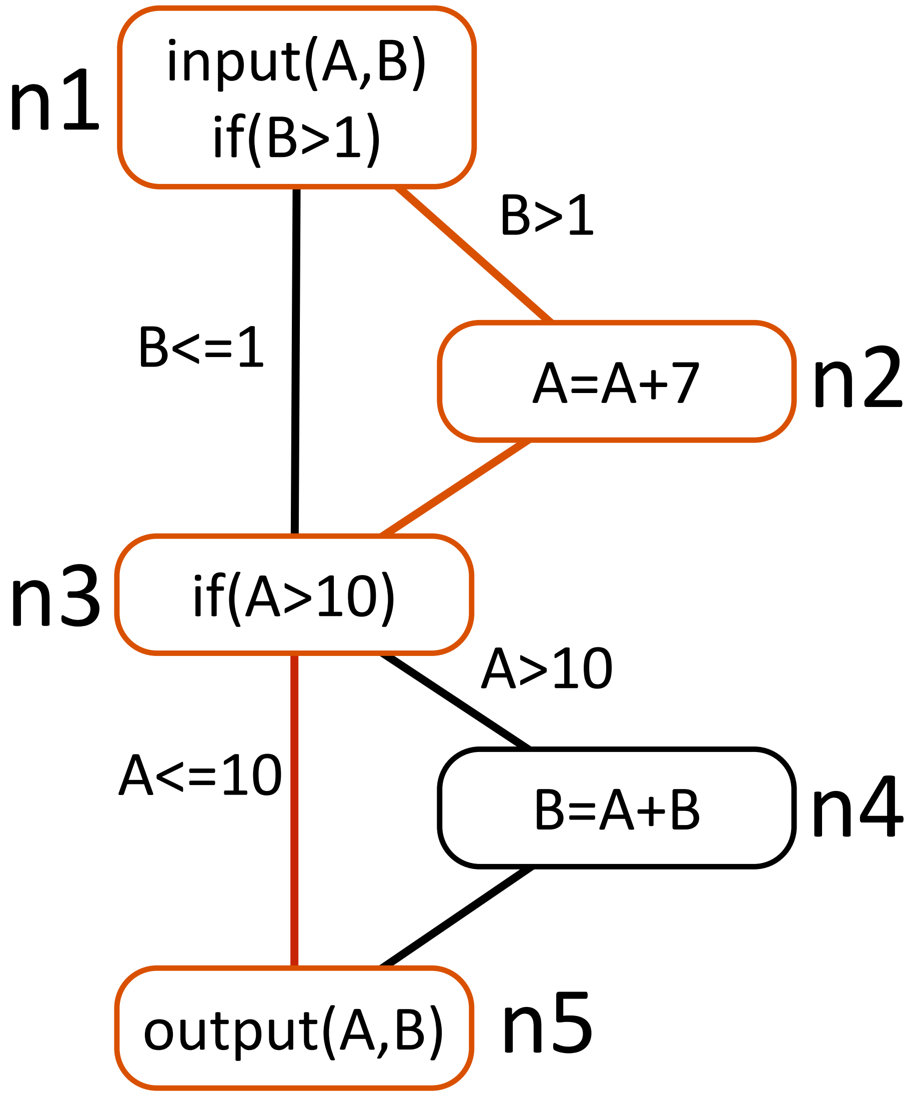
# Def-clear paths subsumed by <1,2,3,5> for variable **A**

| 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> ✓ ✓ |

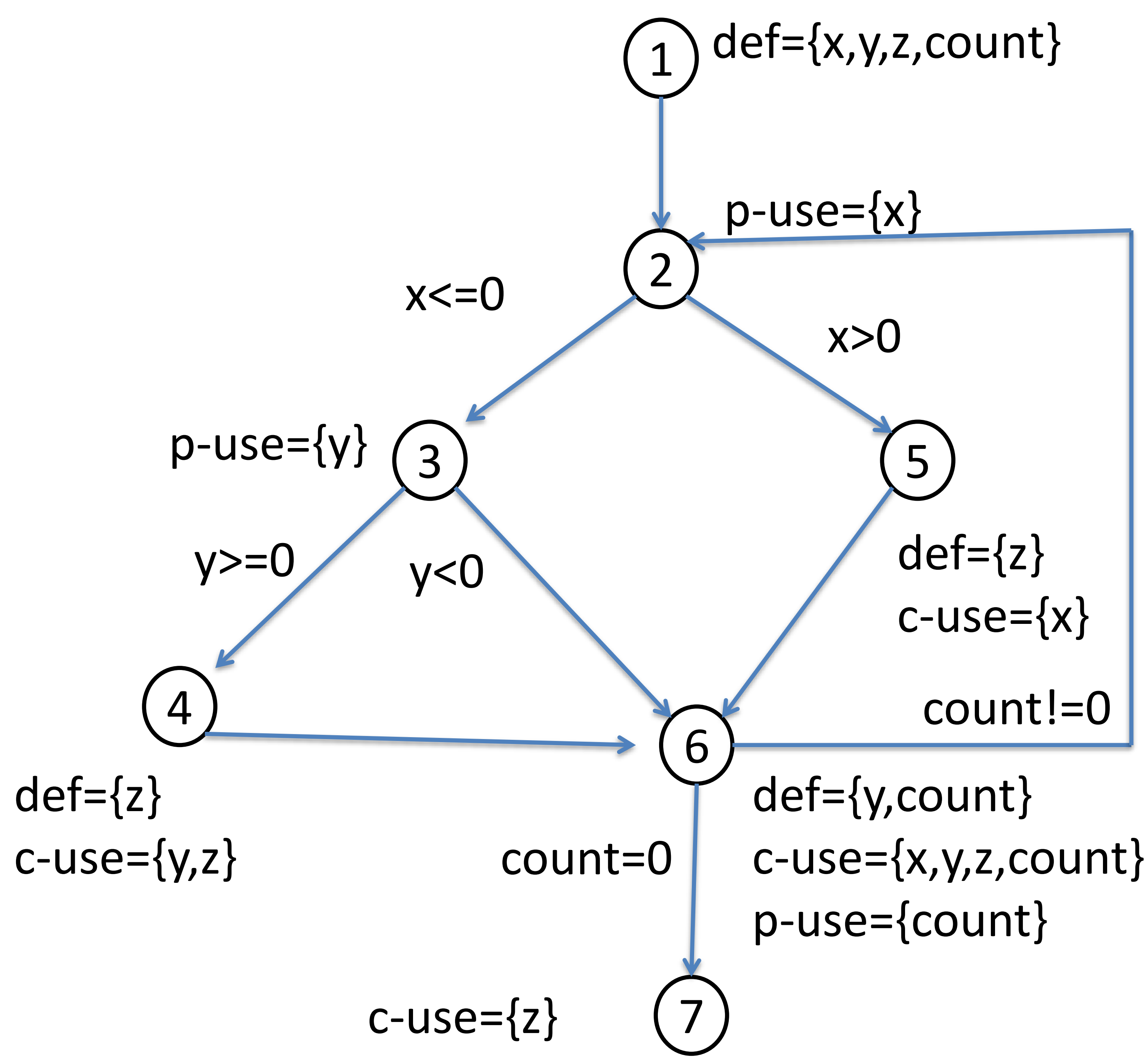


# Dataflow test coverage criteria

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

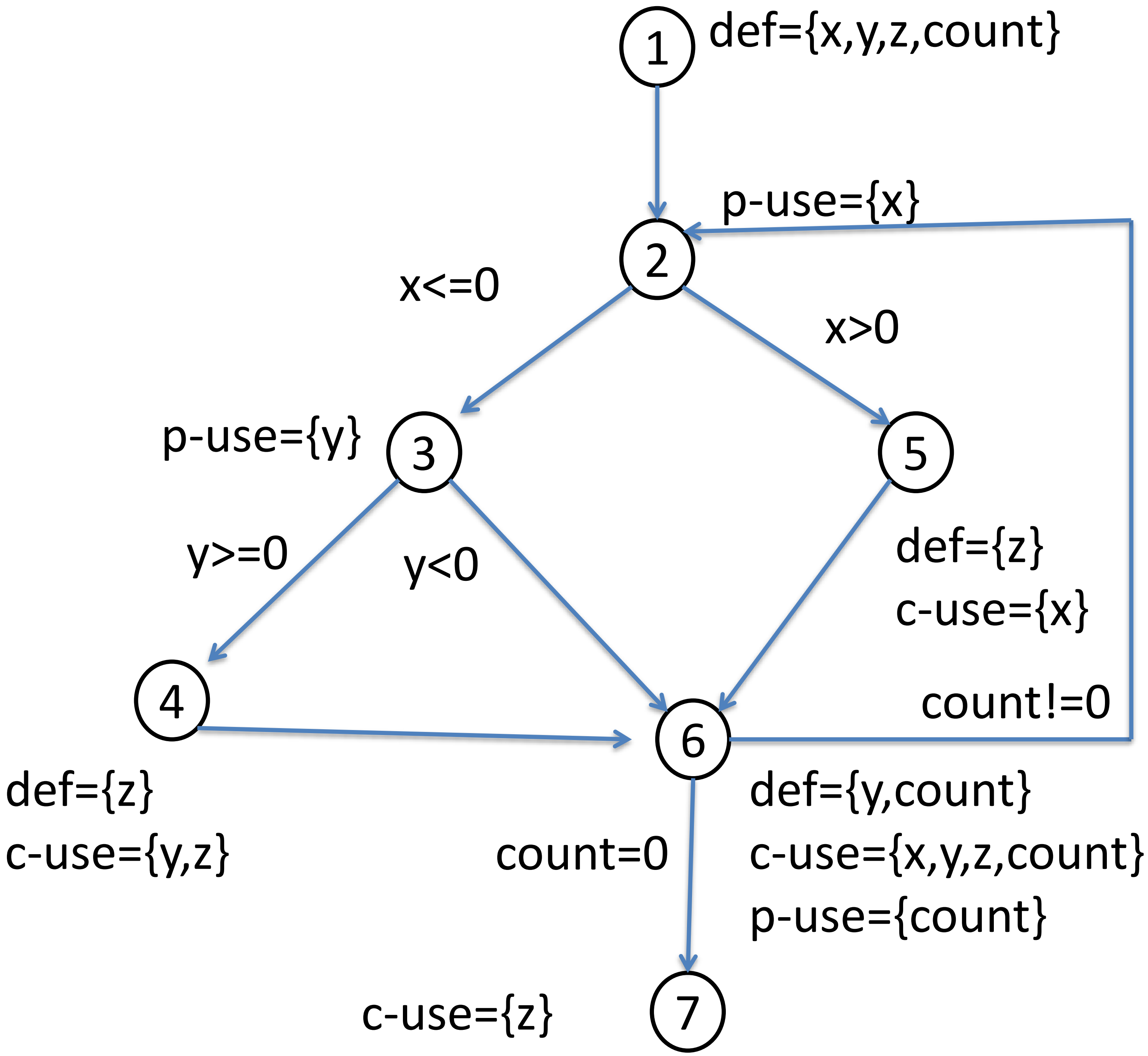
# Another Example

```
begin
    float x,y,z=0.0;
    int count;
    input(x,y,count);
    do{
      if(x≤0) {
        if(y≥0)  {
          z=y*z+1;
        }
      }
      else{
       z=1/x;
      }
      y=x*y+z;
      count=count-1;}
    while(count>0)
    output(z);
end
```

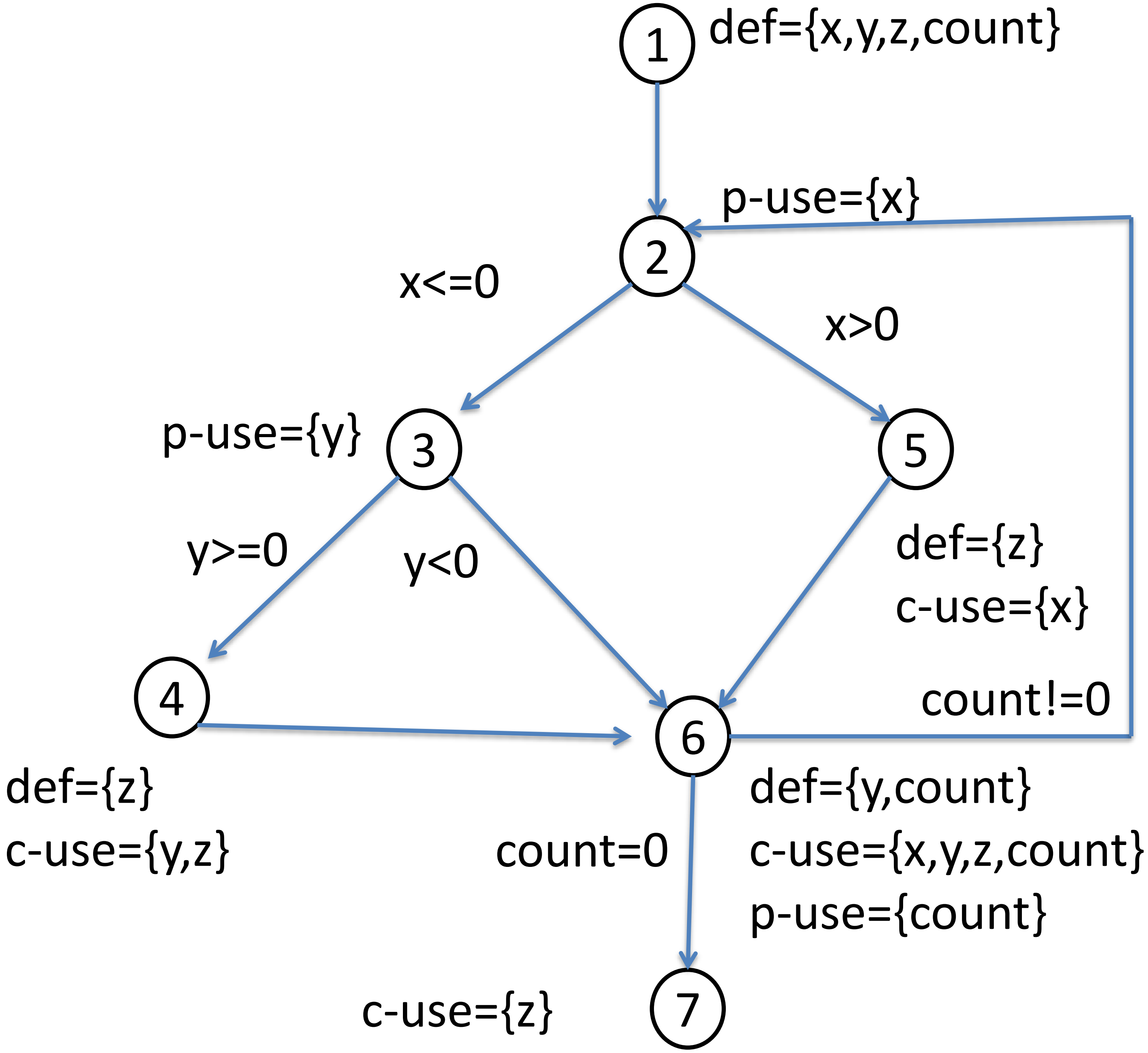


# Some Notations

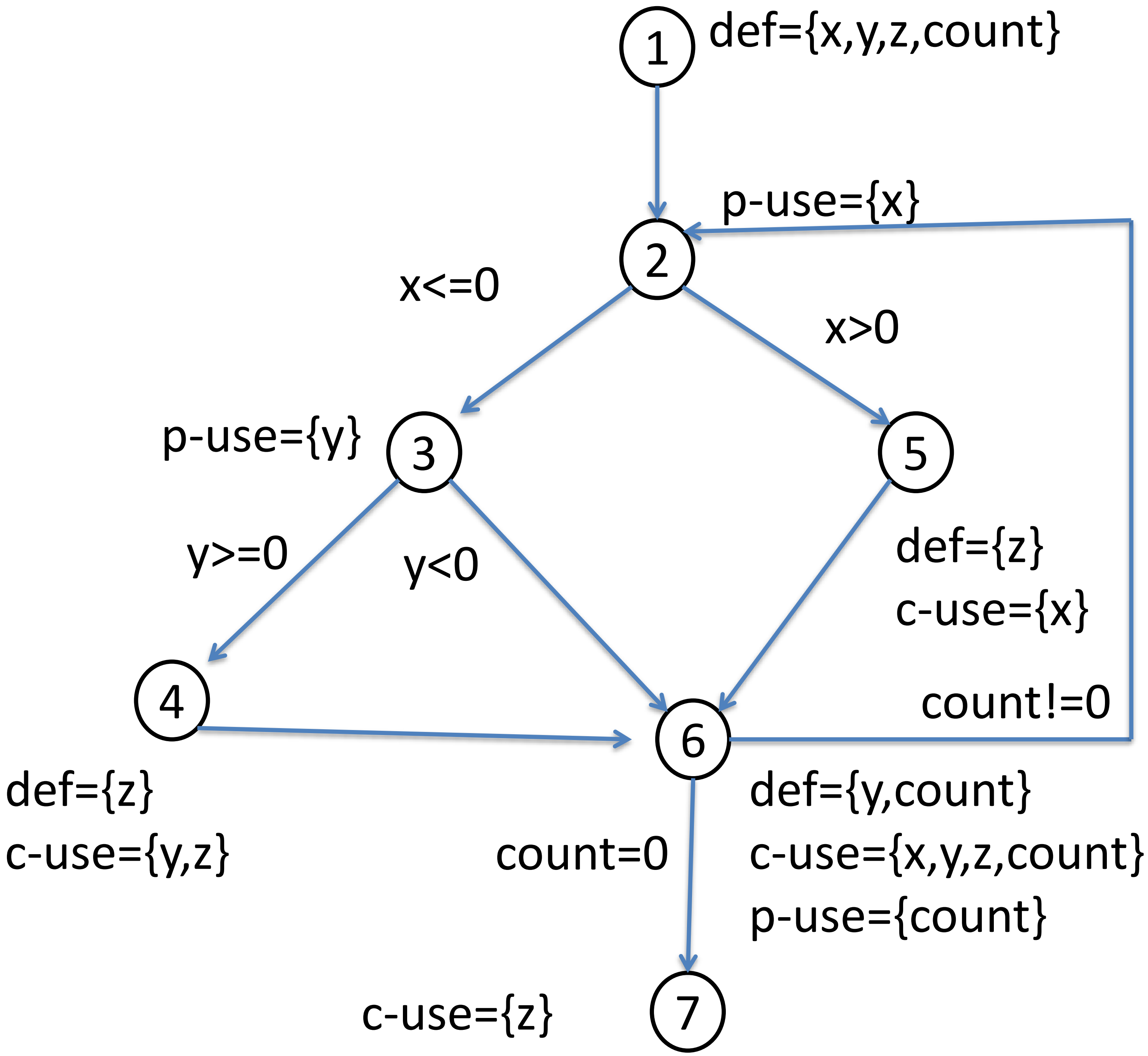
- $d_i(x)$：The definition of variable x in node i
- $u_i(x)$：The use of variable x in node i
- dcu ($d_i(x)$)：The set of c-uses with respect to $d_i(x)$.
  - dcu($d_i(x)$) is also denoted as *dcu(x,i)*
- dpu ($d_i(x)$)：The set of p-uses with respect to $d_i(x)$.
  - dpu($d_i(x)$) is also denoted as *dpu(x,i)*




| Variable v | Note n | dcu(v,n) | dpu(v,n) |
|---|---|---|---|
| x | 1 | {5,6} | {(2,3),(2,5)} |
| y | 1 | {4,6} | {(3,4),(3,6)} |
| … | … | … | … |




- **dcu(y,1)={4,6}， dcu(z,1)={4,6,7}**
- **dcu(z,1) subsumes dcu(y,1)**




$$CU = \sum_{i=1}^{n} \sum_{j=1}^{d_i} |dcu(v_i, n_j)|$$

$$PU = \sum_{i=1}^{n} \sum_{j=1}^{d_i} |dpu(v_i, n_j)|$$

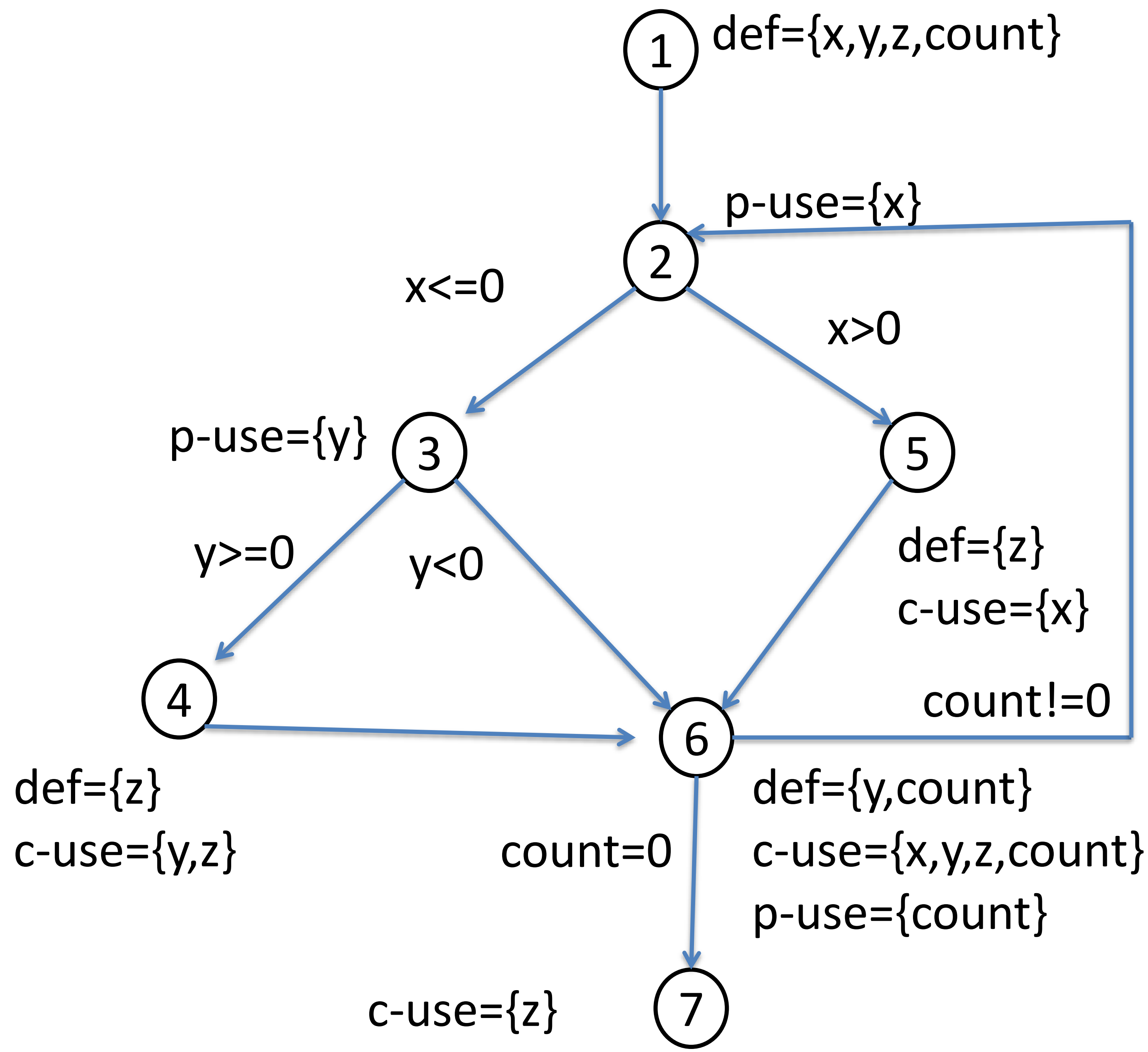
- **n=4,d1=1,d2=2,d3=3,d4=2**
- **The size of All-C-Uses (CU): 17**
- **The size of All-P-Uses (PU): 10**

```
begin
    float x,y,z=0.0;
    int count;
    input(x,y,count);
    do{
      if(x≤0) {
        if(y≥0)  {
          z=y*z+1;
        }
      }
      else{
       z=1/x;
      }
      y=x*y+z;
      count=count-1;}
    while(count>0)
    output(z);
end
```

**Test Case t: <x=5,y=-1, count=1>**

**Path：1-2-5-6-7**

- **C-uses (achieved by this test case): 6**
- **All-C-Uses (CU): 17**
- **C-use Coverage: 6/17<1**

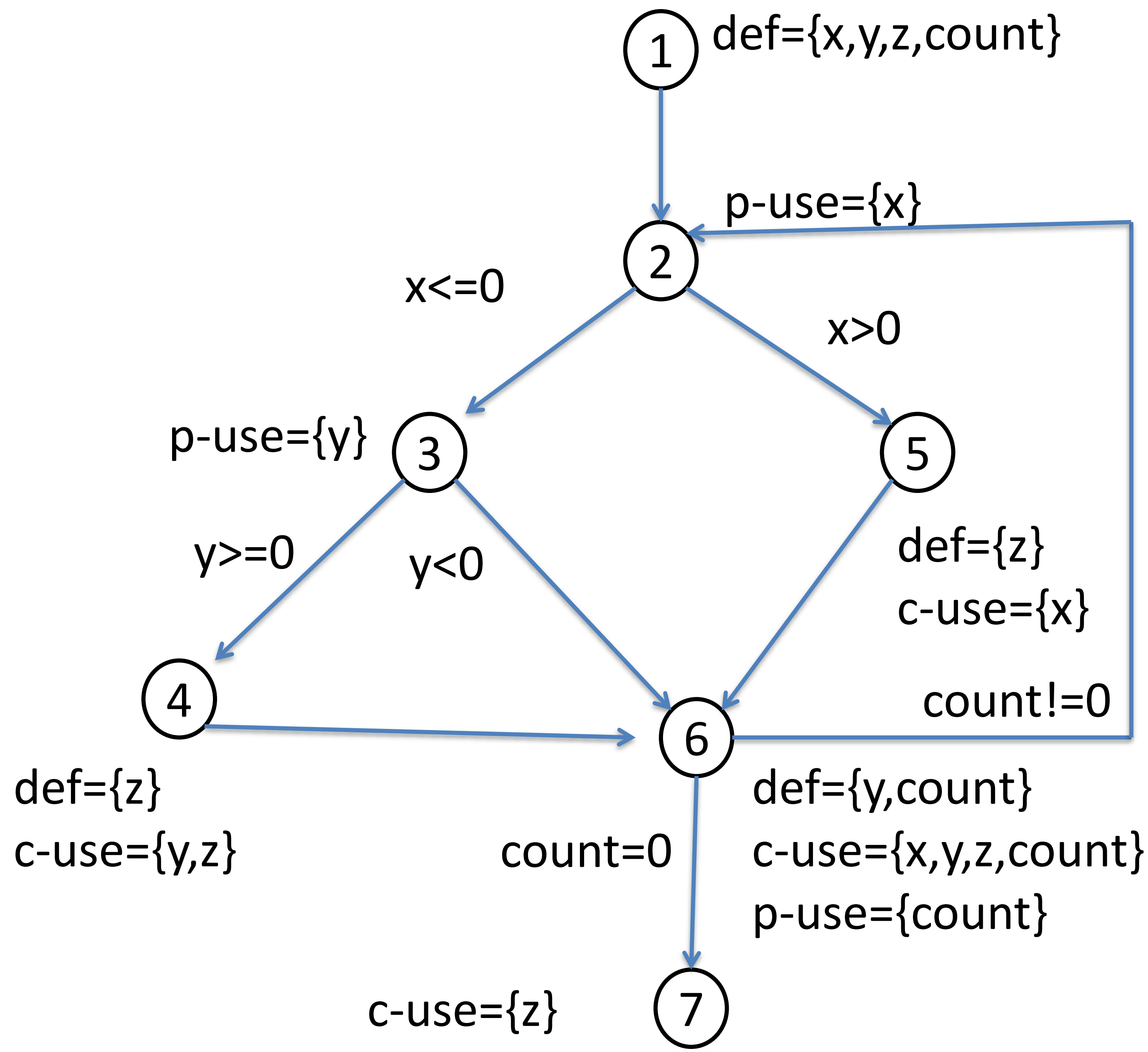


```
begin
    float x,y,z=0.0;
    int count;
    input(x,y,count);
    do{
      if(x≤0) {
        if(y≥0)  {
          z=y*z+1;
        }
      }
      else{
       z=1/x;
      }
      y=x*y+z;
      count=count-1;}
    while(count>0)
    output(z);
end
```

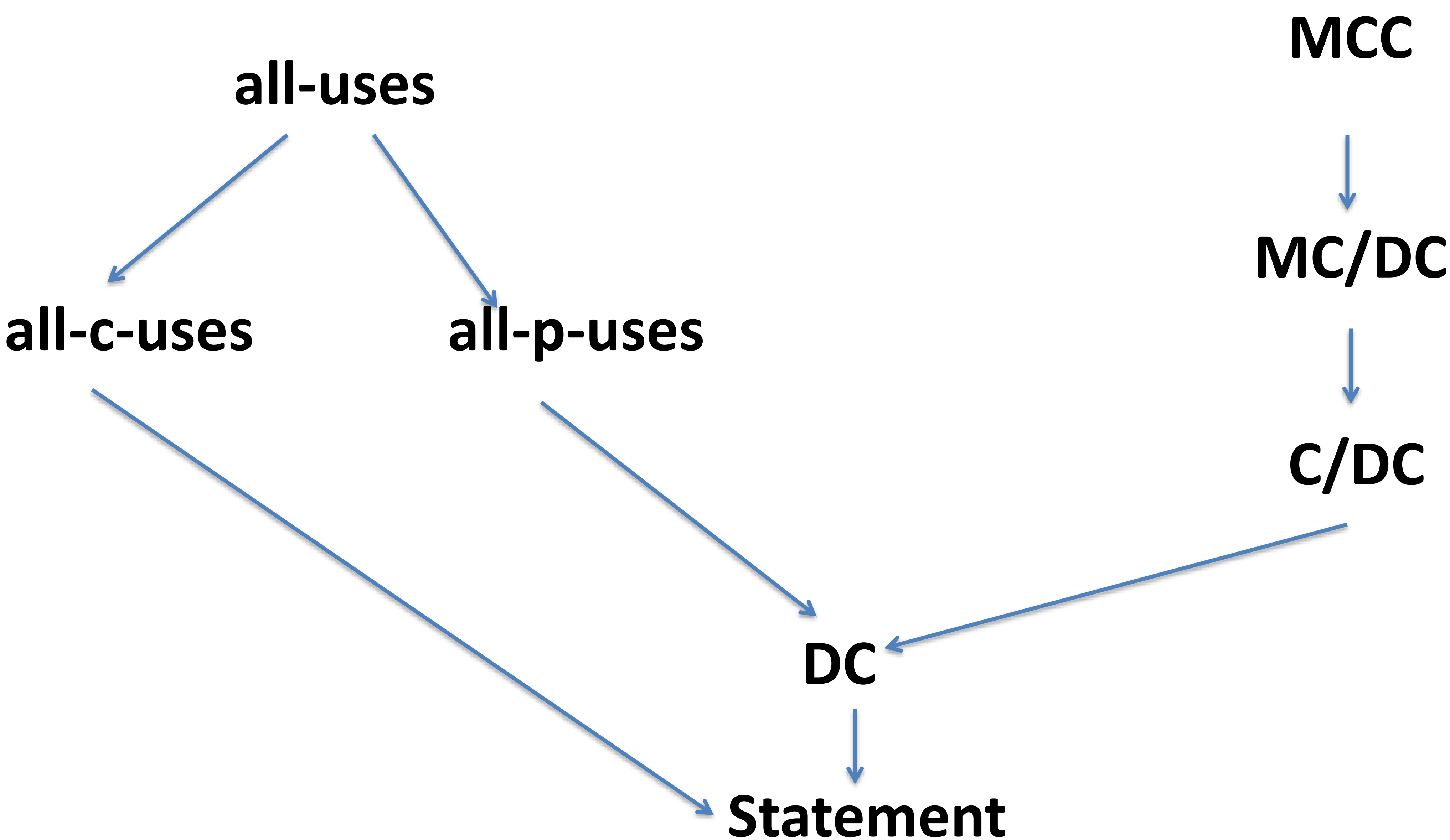
- **T:{t1:<x=5,y=-1,count=1>,t2:<x=-2,y=-1,count=3>}**

  **t1 path： 1-2-5-6-7**

  **t2 path： 1-2-3-$6_1$-2-3-4-$6_2$-2-3-$6_3$-7**
- **P-use coverage: 100%**



# Relationships among some of the coverage criteria



# Quiz

```
begin
    int x,y; float z;
    input(x,y);
    z=0;
    if(x!=0)
      z=z+y;
    else z=z-y;
    if(y!=0)  //should be y!=0 and x!=0
      z=z/x;
    else z=z*x;
    output(z):
end
```

- How to achieve All-Uses coverage?

| Test case | x | y | z |
|---|---|---|---|
| t1 | 0 | 0 | 0.0 |
| t2 | 1 | 1 | 1.0 |
| .. | .. | .. | .. |

# References and Acknowledgement


- 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.
- 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
- Lecture Materials from Lingming Zhang and Dan Hao.
- A. Mathur, Foundations of Software Testing (2nd Edition), ISBN: 978-8131794760

# Thanks!


Testing
Software
development
design
articles
management
test
quality
code
web
use
risk
good
usability
People
Project
information
article
Agile
system
check