White Box Testing

Dr. Aprna Tripathi

Outline

- White Box Testing
  - Control Flow Testing
  - Data Flow Testing
    - Static Data Flow Testing
    - Dynamic Data Flow Testing
  - Coverage Testing
    - Method Coverage
    - Statement Coverage
    - Decision/Branch Coverage
    - Condition Coverage

Definition

- White-box testing is a verification technique software engineers can use to examine if their code works as expected.
- White box testing is a strategy in which testing is based on:
  - the internal paths,
  - structure, and
  - implementation of the software under test (SUT).
- White-box testing is also known as *structural testing*, *clear box testing*, and *glass box testing*
- Generally requires detailed programming skills

White Box Testing

- A software engineer can design test cases that:

(1) exercise independent paths within a module or unit;

(2) exercise logical decisions on both their true and false side;

(3) execute loops at their boundaries and within their operational bounds; and

(4) exercise internal data structures to ensure their validity

White Box Testing Techniques

- Control Flow Testing
- Data Flow Testing
- Coverage Testing

Control-Flow Testing

- **Control-flow testing** is a structural testing strategy that uses the program’s control flow as a model.
- Control-flow testing techniques are based on judiciously selecting a set of test paths through the program.
- The set of paths chosen is used to achieve a certain measure of testing thoroughness.
  - *E.g.*, pick enough paths to assure that every source statement is executed as least once.

Motivation

- Control-flow testing is most applicable to new software for unit testing.
- Control-flow testing assumptions:
  - specifications are correct
  - data is defined and accessed properly
  - there are no bugs other than those that affect control flow
- Structured and OO languages reduce the number of control-flow bugs.

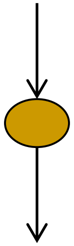
Control Flowgraphs

- The control flowgraph is a graphical representation of a program’s control structure.

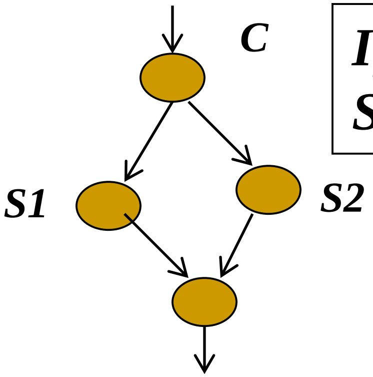
Flowgraphs Consist of Three Primitives

- A **decision** is a program point at which the control can diverge.
  - (*e.g.*, if and case statements).
- A **junction** is a program point where the control flow can merge.
  - (*e.g.*, end if, end loop, goto label)
- A **process block** is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.*, straight-line code).
  - A process has one entry and one exit.
  - A program does not jump into or out of a process.

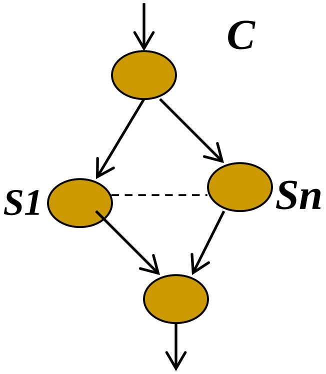
Control Flow Testing



Sequential statement block

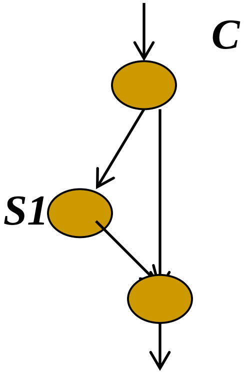


If C Then S1 else S2;



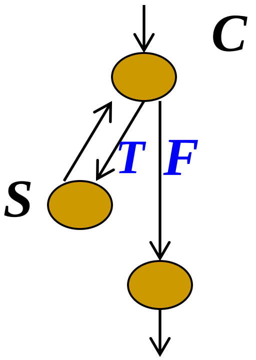
Case C of

L1: S1;
L2: S2;
…
Ln: Sn;
end;

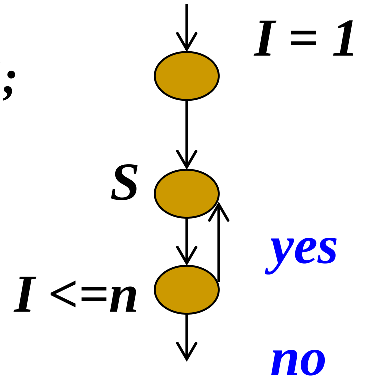


If C Then S1;

Control Flow Testing

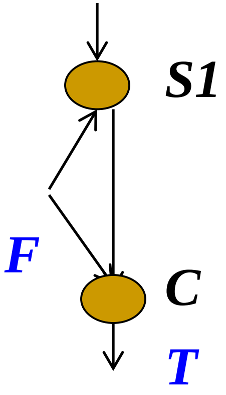


While C do S;



***For** loop:*

for I = 1 to n do S;

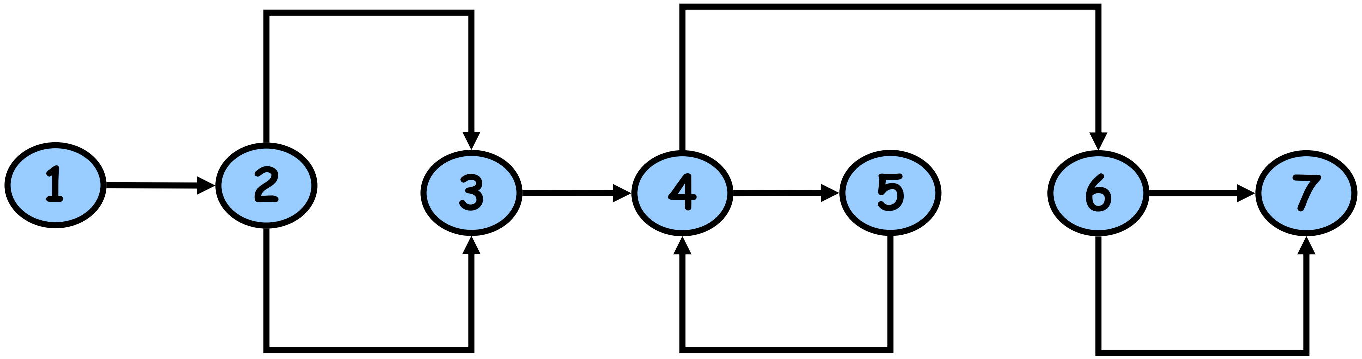


Do loop:

do S1 until C;

Exponentiation Algorithm

```
1    scanf("%d %d",&x, &y);
2    if (y < 0)
         pow = -y;
     else
         pow = y;
3    z = 1.0;
4    while (pow != 0) {
          z = z * x;
          pow = pow - 1;
5         }
6    if (y < 0)
          z = 1.0 / z;
7    printf ("%f",z);
```

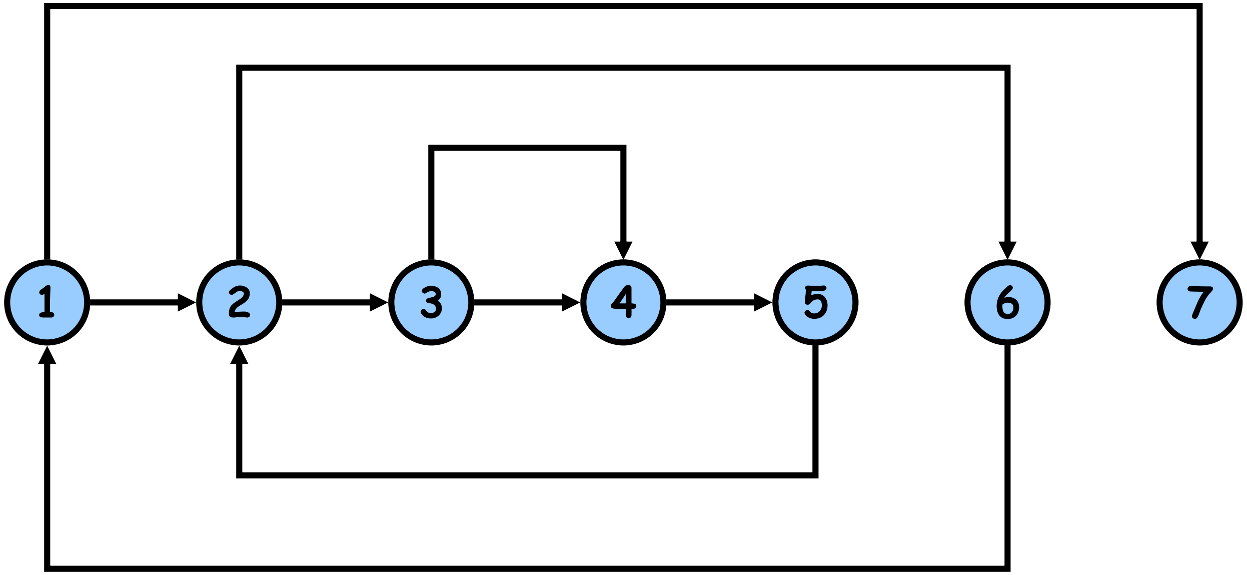


Bubble Sort Algorithm

```
for (j=1; j<N; j++)  {
    last = N - j + 1;
    for (k=1; k<last; k++)  {
        if (list[k] > list[k+1])  {
            temp = list[k];
            list[k] = list[k+1];
            list[k+1] = temp;
        }
    }
}
print(“Done\n”);
```

Bubble Sort Algorithm

```
1  for (j=1; j<N; j++)  {
        last = N - j + 1;
2       for (k=1; k<last; k++)  {
3             if (list[k] > list[k+1])  {
                   temp = list[k];
                   list[k] = list[k+1];
                   list[k+1] = temp;
4             }
5      }
6 }
7 print("Done\n");
```



The general white box testing process

- The SUT's implementation is analyzed.
- Paths through the SUT are identified.
- Inputs are chosen to cause the SUT to execute selected paths - path sensitization
- Expected results for those inputs are determined.
- The tests are run.
- Actual outputs are compared with the expected outputs.
- A determination is made as to the proper functioning of the SUT.

Applicability

- White box testing can be applied at all levels of system development—unit, integration, and system.

- White box testing is more than code testing—it is **path** testing.

- We can apply the same techniques to test paths between modules within subsystems, between subsystems within systems, and even between entire systems.

Disadvantages

- First, the number of execution paths may be so large then they cannot all be tested.

- Second, the test cases chosen may not detect data sensitivity errors.
  - For example: p=q/r; may execute correctly except when r=0.

- Third, white box testing assumes the control flow is correct (or very close to correct). Since the tests are based on the existing paths, nonexistent paths cannot be discovered through white box testing.

- Fourth, the tester must have the programming skills to understand and evaluate the software under test.

Advantages

The tester can be sure that every path through the software under test has been identified and tested.

Control Flow Testing

- Identifies the execution paths through a module of program code
- Creates and executes test cases to cover those paths.
  - Path: A sequence of statement execution that begins at an entry and ends at an exit.
  - *Basis path testing* is a means for ensuring that all independent paths through a code module have been tested
  - An *independent path* is any path through the code that introduces at least one new set of processing statements or a new condition.
  - Basis path testing provides a minimum, lower-bound on the number of test cases that need to be written.

Cyclomatic Complexity

Cyclomatic complexity is a software metric

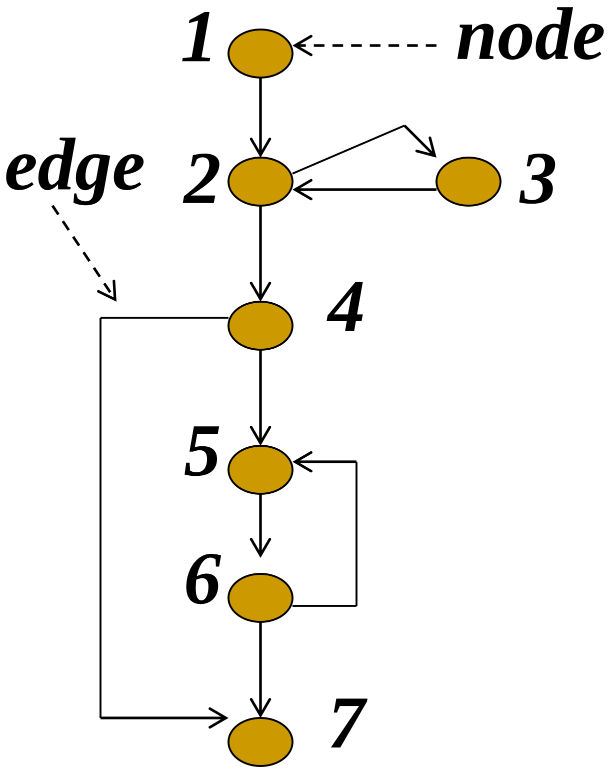
- the value computed for cyclomatic complexity defines the number of independent paths in a program.
- Cyclomatic complexity, V(G), for a flow graph G is defined as

  $V(G) = E - N + 2$

  where E is the number of flow graph edges and N is the number of flow graph nodes.
- Cyclomatic complexity, $V(G) = P + 1$

  where P is the number of predicate nodes contained in the flow graph G.

An Example



No. of edges = 9

No. of nodes = 7

No. of predicate nodes = 3

V(G) = 3 + 1 = 4

V(G) = 9 - 7 + 2 = 4

An Example(cont)

- Step 1 : Using the design or code as a foundation, draw a corresponding flow graph.
- Step 2: Determine the cyclomatic complexity of the resultant flow graph.
- Step 3: Determine a minimum basis set of linearly independent paths.

  For example,

  path 1: 1-2-4-5-6-7

  path 2: 1-2-4-7

  path 3: 1-2-3-2-4-5-6-7

  path 4: 1-2-4-5-6-5-6-7
- Step 4: Prepare test cases that will force execution of each path in the basis set.
- Step 5: Run the test cases and check their results

exhaustive testing drawbacks

- The number of paths could be huge and thus untestable within a reasonable amount of time.
  - Every decision doubles the number of paths and
  - Every loop multiplies the paths by the number of iterations through the loop.
- For example:

```
 for (i=1; i<=1000; i++)
   for (j=1; j<=1000; j++)
     for (k=1; k<=1000; k++)
       doSomethingWith(i,j,k);
```

executes doSomethingWith() one billion times (1000×1000×1000).

Each unique path deserves to be tested.

exhaustive testing drawbacks(Cont)

- Paths called for in the specification may simply be missing from the module. Any testing approach based on implemented paths will never find paths that were not implemented.

```
 if (a>0) doIsGreater();
 if (a==0) dolsEqual();
 // missing statement - if (a<0) dolsLess();
```

exhaustive testing drawbacks(Cont)

- Defects may exist in processing statements within the module even though the control flow itself is correct.

```
// actual (but incorrect) code
a=a+1;
// correct code
a=a-1;
```

- The module may execute correctly for almost all data values but fail for a few.

```
int blech (int a, int b) {
   return a/b;
 }
```

fails if **b** has the value 0 but executes correctly if **b** is not 0.

Data Flow Testing

Data Flow Testing

- Data flow testing is a powerful tool to detect improper use of data values due to coding errors.

```
main() {
            int x;
            if (x==42){ ...}
    }
```

Data Flow Testing

- Variables that contain data values have a defined life cycle. They are created, they are used, and they are killed (destroyed) - Scope

```
{                    // begin outer block
       int x;        // x is defined as an integer within this outer block
       …;            // x can be accessed here
       {              // begin inner block
         int y;               // y is defined within this inner block
...;                // both x and y can be accessed here
       }          // y is automatically destroyed at the end of this block       ...;
       …;           // x can still be accessed, but y is gone
}                   // x is automatically destroyed
```

Data Flow Testing

- Variables can be used
  - in computation
  - in conditionals

- Possibilities for the first occurrence of a variable through a program path
  - ~d the variable does not exist, then it is defined (d)
  - ~u the variable does not exist, then it is used (u)
  - ~k the variable does not exist, then it is killed or destroyed (k)

Data Flow Testing

- Examine time-sequenced pairs of defined (d), used (u), and killed (k):
    - dd - not invalid but suspicious. Probably a programming error.
    - du - perfectly correct. The normal case.
    - dk - not invalid but probably a programming error.
    - ud - acceptable.
    - uu - acceptable.
    - uk - acceptable.

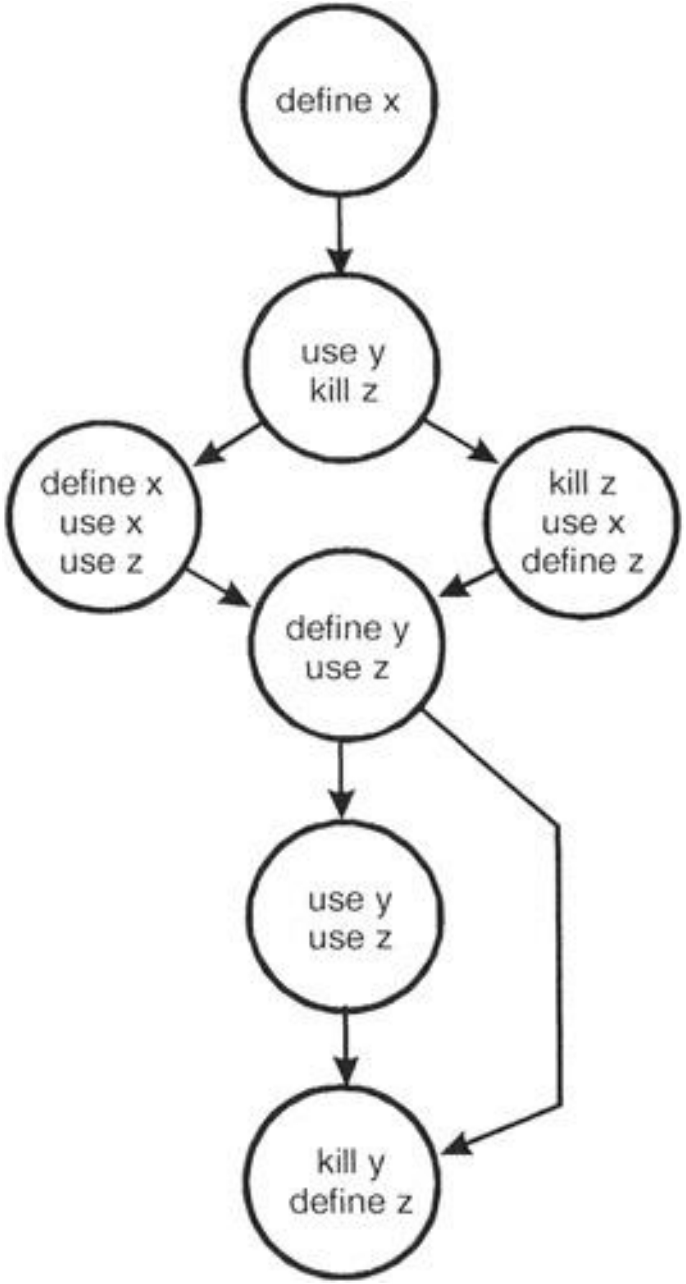
Data Flow Testing

kd - acceptable.

ku - a serious defect. Using a variable that does not exist or is undefined is always an error.

kk - probably a programming error.

Example **static Data Flow Testing**

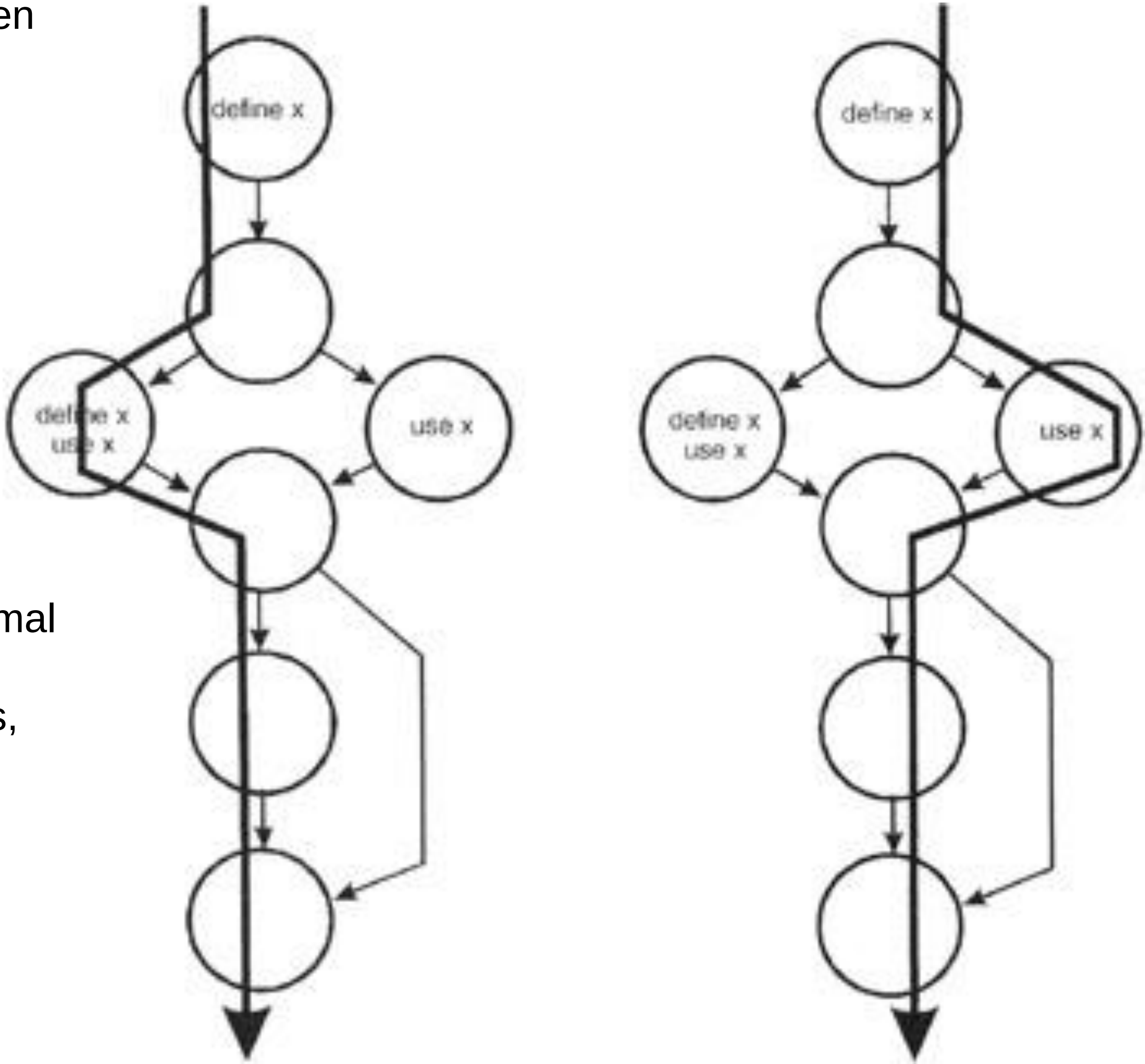


Example(cont)

- For each variable within the module we will examine define-use-kill patterns along the control flow paths

Example(cont)

Consider variable x as we traverse the left and then the right path



~define correct, the normal case

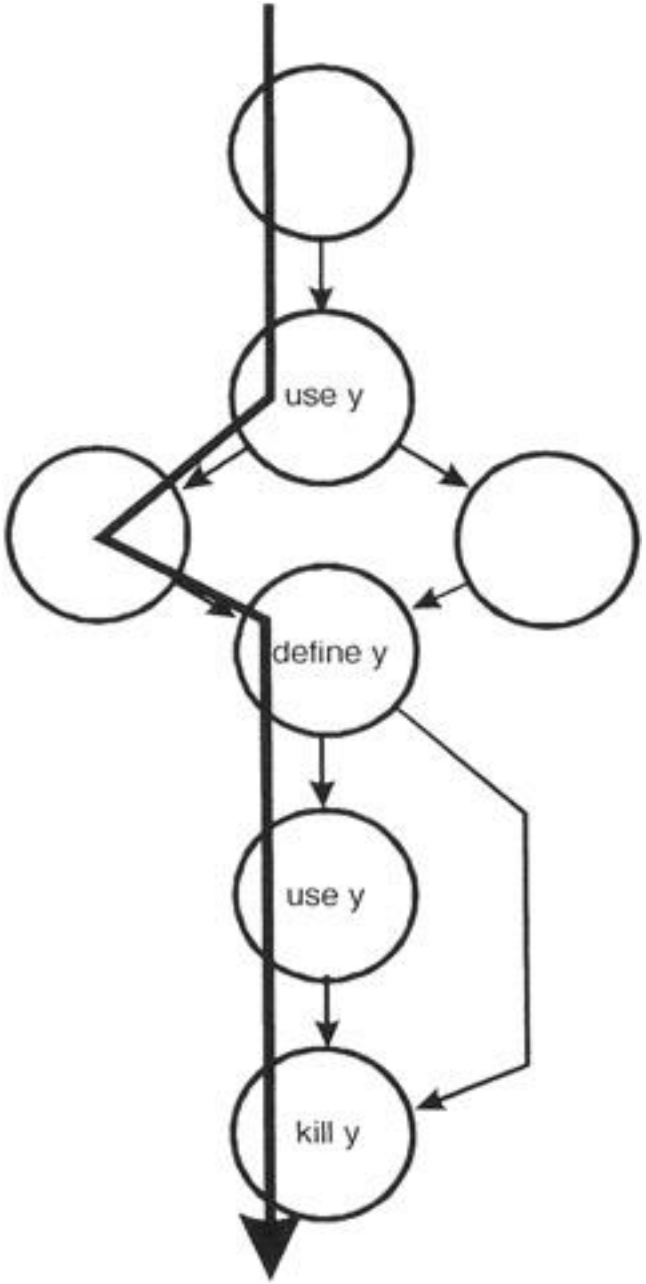
define-define suspicious, perhaps a programming error

define-use correct, the normal case

Example(cont)

Consider variable y

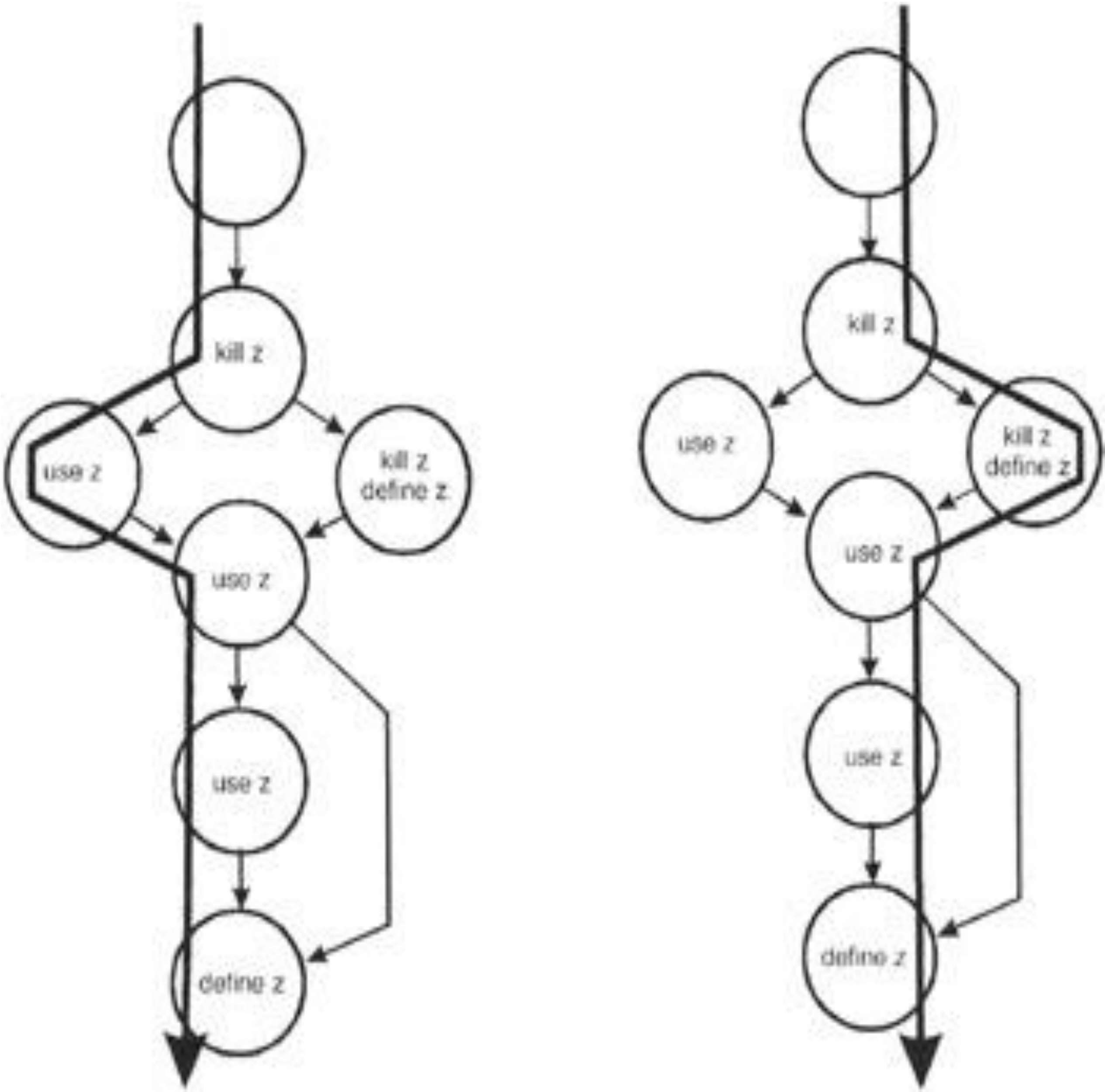
~use	major blunder
use-define	acceptable
define-use	correct, the normal case
use-kill	acceptable



Example(cont)

Consider variable z

~kill	programming error
kill-use	major blunder
use-use	correct, the normal case
use-define	acceptable
kill-kill	probably a programming error
kill-define	acceptable
define-use	correct, the normal case



Example(cont)

problems

x: define-define

y: ~use

y: define-kill

z: ~kill

z: kill-use

z: kill-kill

Static Data Flow Testing

- Static testing cannot find all errors

Examples:

Arrays are collections of data elements that share the same name and type. For example

```
int test[100];          //defines an array named test
                        // consisting of 100 integer elements,
                // named test[0], test[1], etc.
```

Arrays are defined and destroyed as a unit but specific elements of the array are used individually.

Static analysis cannot determine whether the define-use-kill rules have been followed properly unless each element is considered individually.

Static Data Flow Testing (cont)

- In complex control flows it is possible that a certain path can never be executed.

  In this case an improper define-use-kill combination might exist but will never be executed and so is not truly improper.

Dynamic Data Flow Testing

- Data flow testing is based on a module's control flow, it assumes that the control flow is basically correct.
- The data flow testing process is to choose enough test cases so that:
  - Every "define" is traced to each of its "uses"
  - Every "use" is traced from its corresponding "define"
  - To do this,
    - enumerate the paths through the module.
    - Begin at the module's entry point, take the leftmost path through the module to its exit.
    - Return to the beginning and vary the first branching condition. Follow that path to the exit.
    - Repeat until all the paths are listed.
    - For every variable, create at least one test case to cover every define-use pair.
      - How do we choose the values? Using ……..

The General Idea

- A program unit accepts inputs, performs computations, assigns new values to variables, and returns results.
- One can visualize of “flow” of data values from one statement to another.
- A data value produced in one statement is expected to be used later.
  - Example
    - Obtain a file pointer ……. use it later.
  - If the later use is never verified, we do not know if the earlier assignment is acceptable.

The General Idea

- Two motivations of data flow testing
  - The memory location for a variable is accessed in a "desirable" way.
  - Verify the correctness of data values "defined" (i.e. generated) – observe that all the "uses" of the value produce the desired results.
- Idea: A programmer can perform a number of tests on data values.
  - These tests are collectively known as data flow testing.

The General Idea

- Data flow testing can be performed at two conceptual levels.
  - Static data flow testing
  - Dynamic data flow testing
- Static data flow testing
  - Identify potential defects, commonly known as **data flow anomaly.**
  - Analyze source code.
  - Do not execute code.

The General Idea

- Dynamic data flow testing
  - Involves actual program execution.
  - Bears similarity with control flow testing.
    - Identify paths to execute them.
    - Paths are identified based on **data flow testing criteria**.

Data Flow Anomaly

- Anomaly: It is an abnormal way of doing something.
  - Example 1: The second definition of x overrides the first.

    ```
    x = f1(y);
    x = f2(z);
    ```

- Three types of abnormal situations with using variable.
  - Type 1: Defined and then defined again
  - Type 2: Undefined but referenced
  - Type 3: Defined but not referenced

Data Flow Anomaly

- Type 1: Defined and then defined again (Example 1 above)
    - Four interpretations of Example 1
        - The first statement is redundant.
        - The first statement has a fault -- the intended one might be: w = f1(y).
        - The second statement has a fault – the intended one might be: v = f2(z).
        - There is a missing statement in between the two: v = f3(x).
    - Note: It is for the programmer to make the desired interpretation.

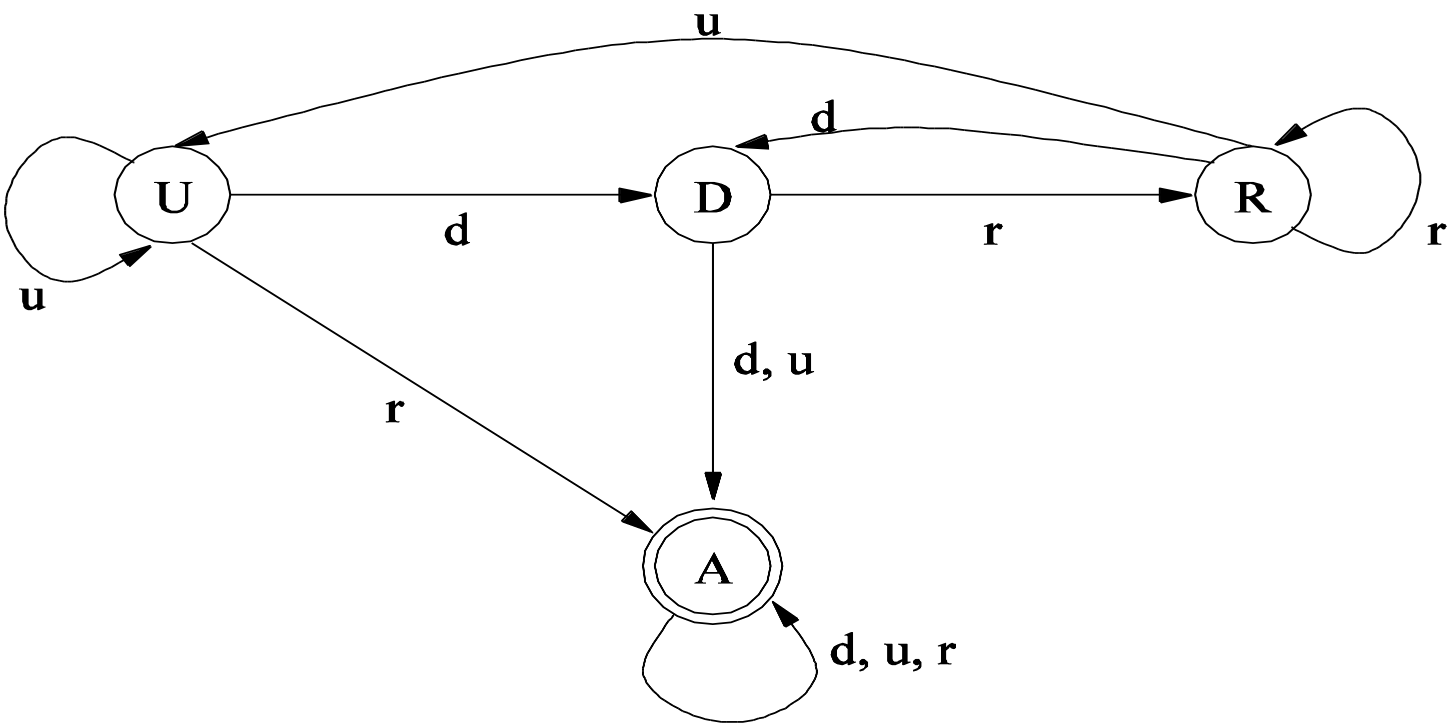
Data Flow Anomaly

- Type 2: Undefined but referenced
  - Example: x = x – y – w; /* w has not been defined by the programmer. */
  - Two interpretations
    - The programmer made a mistake in using w.
    - The programmer wants to use the compiler assigned value of w.
- Type 3: Defined but not referenced
  - Example: Consider x = f(x, y). If x is not used subsequently, we have a Type 3 anomaly.

Data Flow Anomaly

- The concept of **a state-transition diagram** is used to **model a program variable** to identify data flow anomaly.
- Components of the state-transition diagrams
  - The states
    - U: Undefined
    - D: Defined but not referenced
    - R: Defined and referenced
    - A: Abnormal
  - The actions
    - *d*: define the variable
    - *r*: reference (or, read) the variable
    - *u*: undefine the variable

Data Flow Anomaly



Legends:

Figure 5.2: State transition diagram of a program variable [10] (©[1979] IEEE).

States

U: Undefined

D: Defined but not referenced

R: Defined and referenced

A: Abnormal

Actions

d: Define

r: Reference

u: Undefine

Data Flow Anomaly

- Obvious question: What is the relationship between the **Type 1, Type 2,** and **Type 3** anomalies and Figure 5.2?
- The three types of anomalies (Type 1, Type 2, and Type 3) are found in the diagram in the form of **action sequences**:
  - Type 1: *dd*
  - Type 2: *ur*
  - Type 3: *du*

Data Flow Anomaly

- Detection of data flow anomaly via program instrumentation
  - Program instrumentation: Insert new code to monitor the states of variables.
  - If the state sequence contains *dd*, *ur*, or *du* sequence, a data flow anomaly is said to occur.
- Bottom line: What to do after detecting a data flow anomaly?
  - Investigate the cause of the anomaly.
  - To fix an anomaly, write new code or modify the existing code.

Overview of Dynamic Data Flow Testing

- A programmer manipulates/uses variables in several ways, i.e.: Initialization, assignment, using in a computation, using in a condition
  - One should not feel confident that a variable has been **assigned the correct value**, if no test causes the execution of a **path** from the point of assignment to a point where the value is **used**.
    - Assignment of correct value means whether or not a value has been correctly generated.
    - Use of a variable means
      - If new values of the same variable or other variables are generated.
      - If the variable is used in a conditional statement to alter the flow of control.
- The above motivation indicates that **certain kinds of paths** are executed in data flow testing.

Overview of Dynamic Data Flow Testing

- Data flow testing is outlined as follows:
  - Draw a data flow graph from a program.
  - Select one or more data flow testing criteria.
  - Identify paths in the data flow graph satisfying the selection criteria.
  - Derive path predicate expressions from the selected paths
  - Solve the path predicate expressions to derive test inputs.

Data Flow Graph

- Occurrences of variables
  - Definition: A variable gets a new value.
    - i = x; /* The variable i gets a new value. */
  - Un-definition or kill: This occurs if the value and the location become unbound.
  - Use: This occurs when the value is fetched from the memory location of the variable. There are **two forms** of uses of a variable.
    - Computation use (c-use)
      - Example: x = 2*y; /* y has been used to compute a value of x. */
    - Predicate use (p-use)
      - Example: if (y > 100) { …} /* y has been used in a condition. */

Data Flow Graph

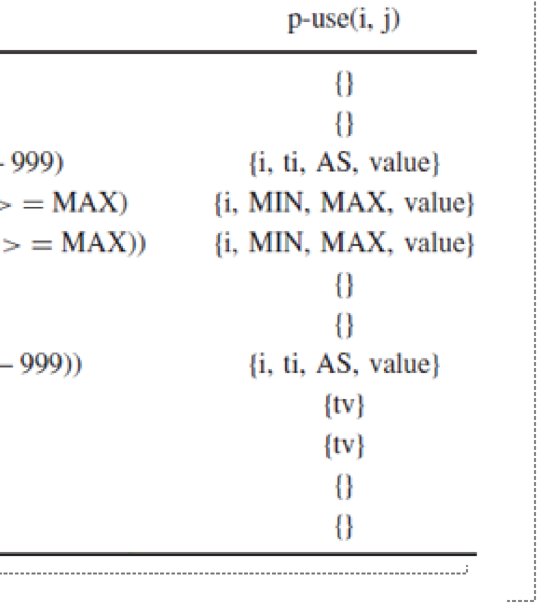
- A data flow graph is a directed graph constructed as follows.
  - A sequence of **definitions** and **c-uses** is associated with each **node** of the graph.
  - A set of **p-uses** is associated with each **edge** of the graph.
  - The entry node has a definition of each edge parameter and each nonlocal variable used in the program.
  - The exit node has an undefinition of each local variable.

Example code: ReturnAverage()

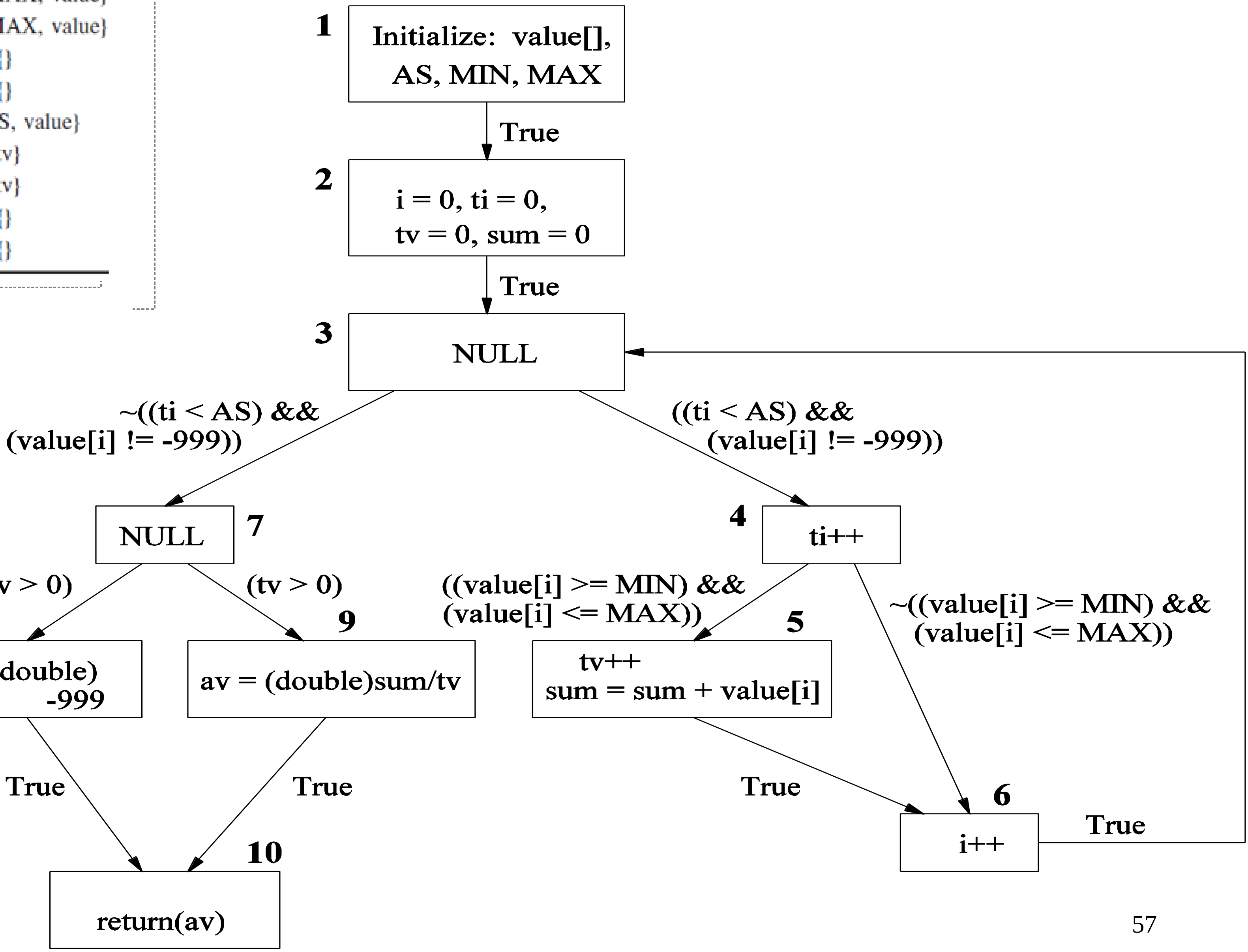
```
public static double ReturnAverage(int value[],  int AS, int MIN, int MAX){
   /* Function: ReturnAverage  Computes the  average of all  those  numbers in
    the  input array  in the  positive  range  [MIN, MAX]. The  maximum size  of the
    array is AS. But, the  array size could be smaller than AS in which case the end
    of input is represented by -999. */

        int i, ti, tv, sum;
        double av;
        i = 0; ti = 0; tv = 0; sum = 0;
        while (ti < AS && value[i] != -999) {
           ti++;
           if (value[i] >= MIN && value[i] <= MAX) {
              tv++;
              sum = sum + value[i];
           }
           i++;
        }
        if (tv > 0)
           av = (double)sum/tv;
        else
           av = (double) -999;
        return (av);     }
```

v Graph



	p-use(i, j)
	{}
	{}
- 999)	{i, ti, AS, value}
> = MAX)	{i, MIN, MAX, value}
> = MAX))	{i, MIN, MAX, value}
	{}
	{}
– 999))	{i, ti, AS, value}
	{tv}
	{tv}
	{}
	{}


1
Initialize: value[], AS, MIN, MAX
True
2
i = 0, ti = 0, tv = 0, sum = 0
True
3
NULL
~((ti < AS) && (value[i] != -999))
((ti < AS) && (value[i] != -999))
7
NULL
4
ti++
~(tv > 0)
(tv > 0)
((value[i] >= MIN) && (value[i] <= MAX))
~((value[i] >= MIN) && (value[i] <= MAX))
8
av = (double) -999
9
av = (double)sum/tv
5
tv++
sum = sum + value[i]
True
True
True
6
i++
True
10
return(av)

Data Flow Graph

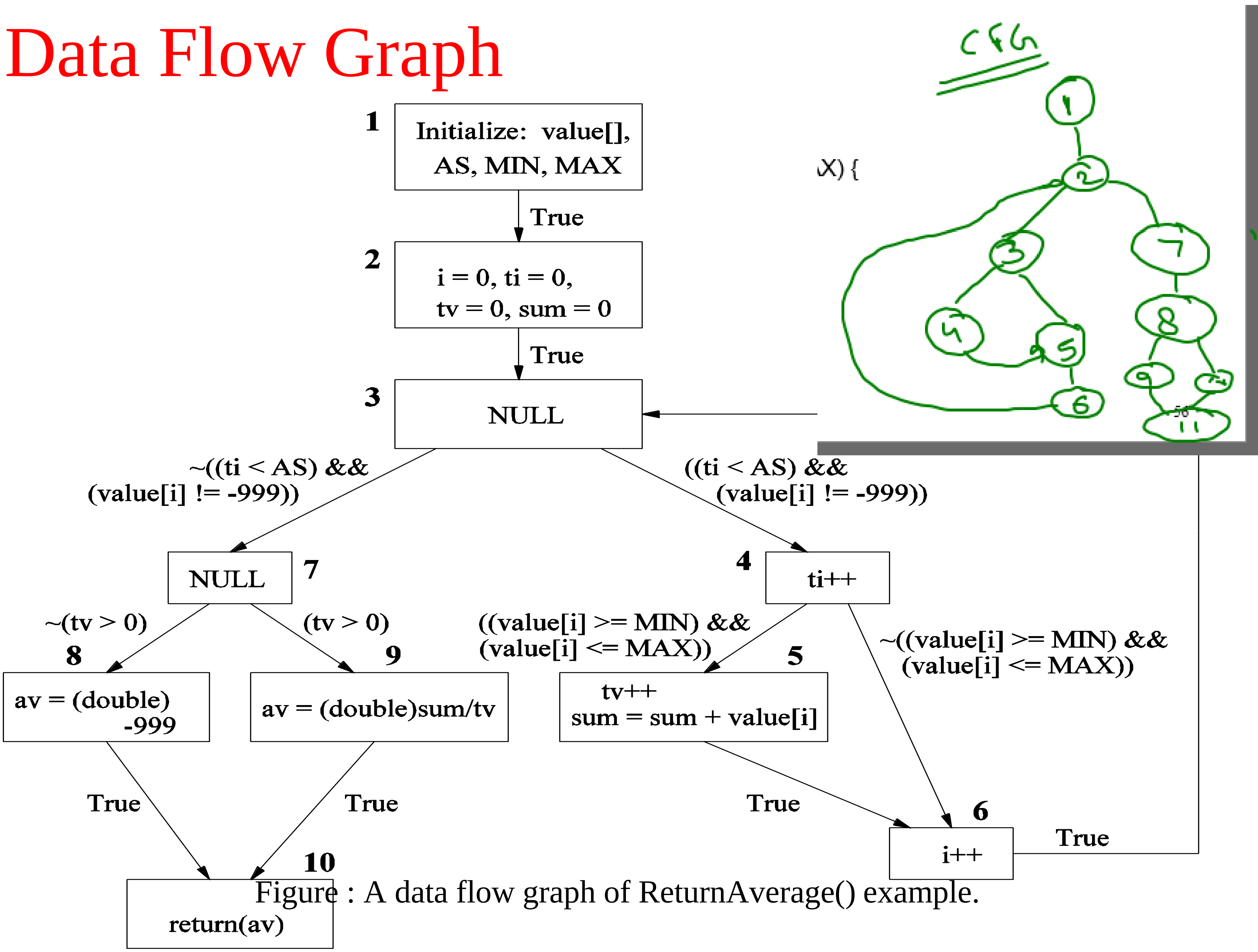


Figure : A data flow graph of ReturnAverage() example.

Data Flow Graph

- A data flow graph is a directed graph constructed as follows.
  - A sequence of **definitions** and **c-uses** is associated with each **node** of the graph.
  - A set of **p-uses** is associated with each **edge** of the graph.
  - The entry node has a definition of each edge parameter and each nonlocal variable used in the program.
  - The exit node has an undefinition of each local variable.

Data Flow Terms

- **Global c-use**: A c-use of a variable x in node i is said to be a global c-use if x has been defined before in a node other than node i.
  - Example: The c-use of variable tv in node 9 is a global c-use.

Data Flow Terms

- **Definition clear path**: A path ($i - n_1 - \cdots n_m - j$), $m \geq 0$, is called a definition clear path (def-clear path) with respect to variable x

  from node i to node j, and

  from node i to edge (n_m, j),

  if x has been neither defined nor undefined in nodes $n_1 - \cdots n_m$.

  - Example: (2 – 3 – 4 – 6 – 3 – 4 – 6 – 3 – 4 – 5) is a def-clear path w.r.t. tv .
  - Example: (2 – 3 – 4 – 5) and (2 – 3 – 4 – 6) are def-clear paths w.r.t. variable tv from node 2 to 5 and from node 2 to 6, respectively.

Data Flow Terms

- **Global definition**: A node i has a global definition of variable x if node i has a definition of x and there is a def-clear path w.r.t. x from node i to some

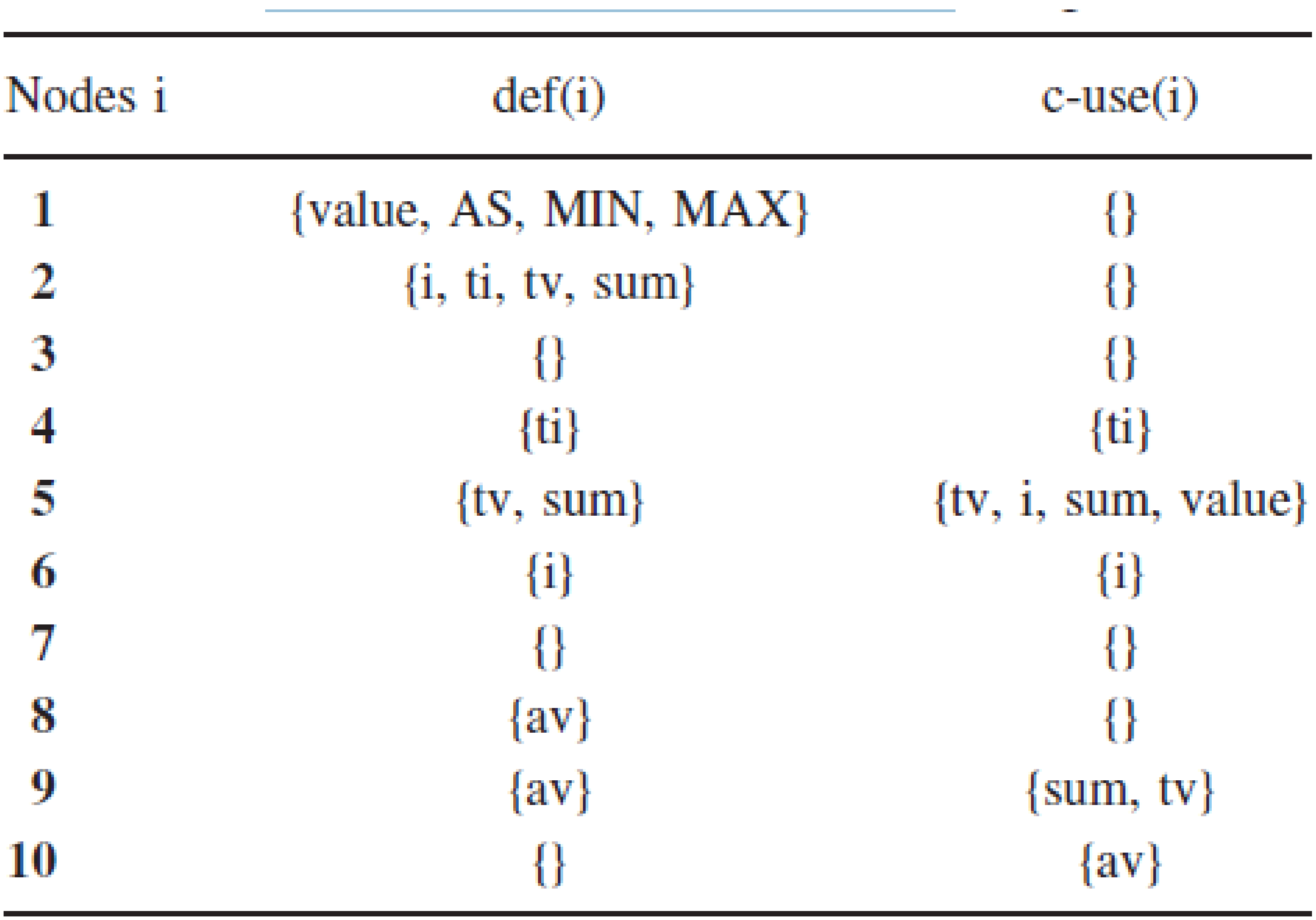
  node containing a global c-use, or

  edge containing a p-use of variable x.
- **Simple path**: A simple path is a path in which all nodes, except possibly the first and the last, are distinct.
  - Example: Paths (2 – 3 – 4 – 5) and (3 – 4 – 6 – 3) are simple paths.
- **Loop-free paths**: A loop-free path is a path in which all nodes are distinct.
- **Complete path**: A complete path is a path from the entry node to the exit node.

Data Flow Terms

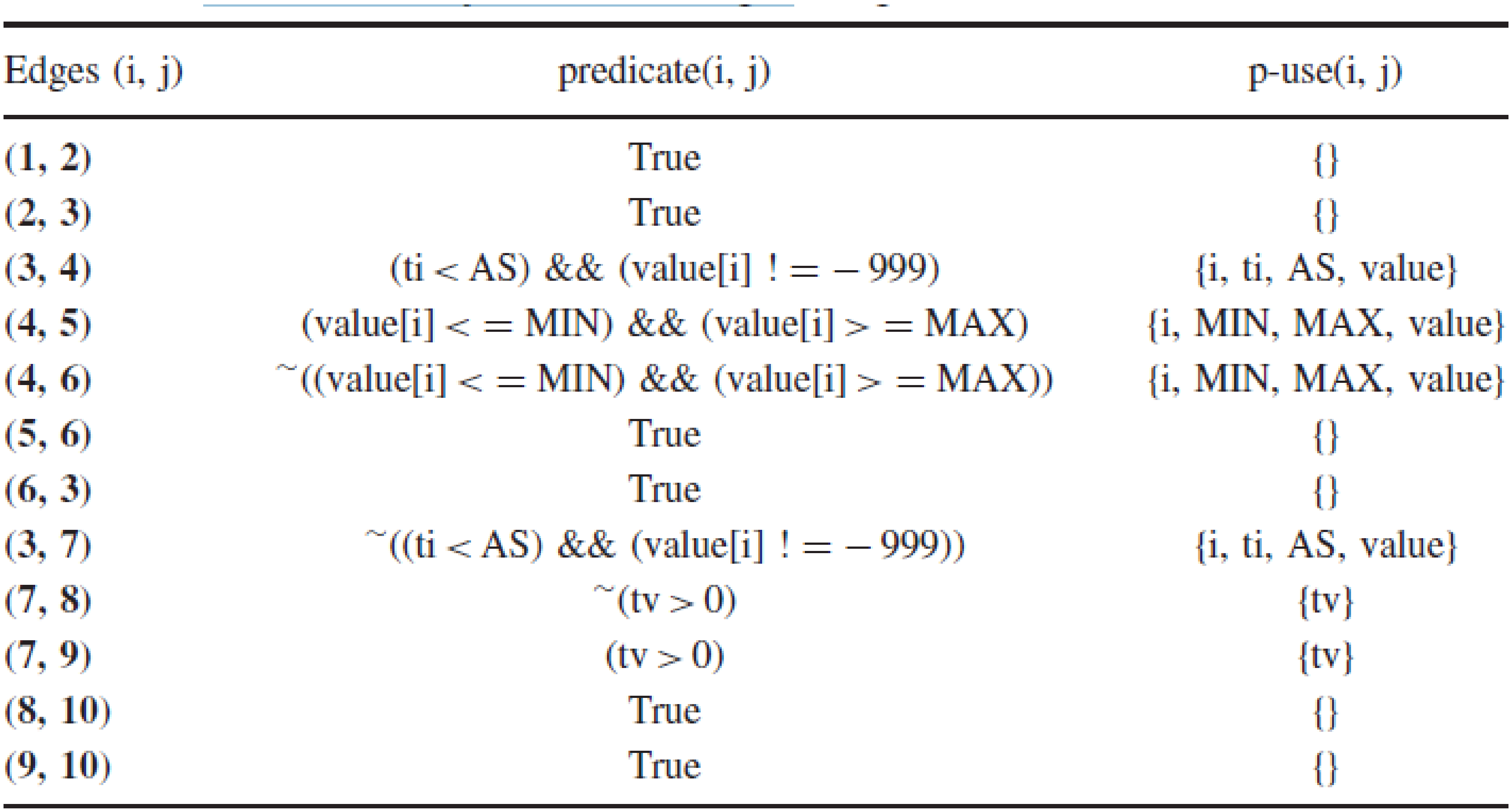
- **Du-path**: A path ($n_1 – n_2 – … – n_j – n_k$) is a du-path path w.r.t. variable x if node n_1 has a global definition of x and either
  - node n_k has a global c-use of x and ($n_1 – n_2 – … – n_j – n_k$) is a def-clear simple path w.r.t. x, or
  - Edge (n_j, n_k) has a p-use of x and ($n_1 – n_2 – … – n_j – n_k$) is a def-clear, loop-free path w.r.t. x.
- Example: Considering the global definition and global c-use of variable tv in nodes 2 and 5, respectively, (2 – 3 – 4 – 5) is a du-path.
- Example: Considering the global definition and p-use of variable tv in nodes 2 and on edge (7, 9), respectively, (2 – 3 – 7 – 9) is a du-path.

Def() and c-use() Sets of Nodes for Return_Average



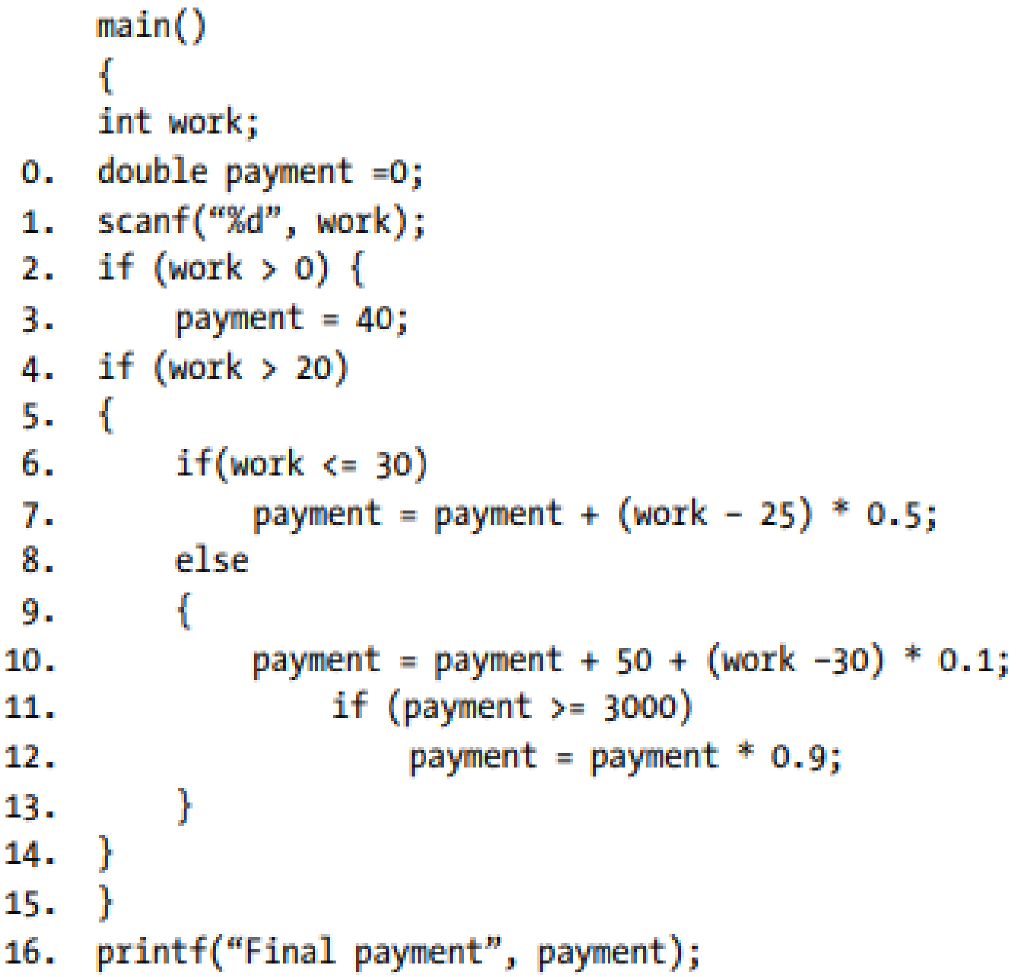
Nodes i	def(i)	c-use(i)
1	{value, AS, MIN, MAX}	{}
2	{i, ti, tv, sum}	{}
3	{}	{}
4	{ti}	{ti}
5	{tv, sum}	{tv, i, sum, value}
6	{i}	{i}
7	{}	{}
8	{av}	{}
9	{av}	{sum, tv}
10	{}	{av}

Predicates and p-use() Set of Edges For Return_Average



Edges (i, j)	predicate(i, j)	p-use(i, j)
(1, 2)	True	{}
(2, 3)	True	{}
(3, 4)	(ti < AS) && (value[i] ! = – 999)	{i, ti, AS, value}
(4, 5)	(value[i] < = MIN) && (value[i] > = MAX)	{i, MIN, MAX, value}
(4, 6)	~((value[i] < = MIN) && (value[i] > = MAX))	{i, MIN, MAX, value}
(5, 6)	True	{}
(6, 3)	True	{}
(3, 7)	~((ti < AS) && (value[i] ! = – 999))	{i, ti, AS, value}
(7, 8)	~(tv > 0)	{tv}
(7, 9)	(tv > 0)	{tv}
(8, 10)	True	{}
(9, 10)	True	{}

Exercise- Design the DFG and find def() c-use() and p-use() set

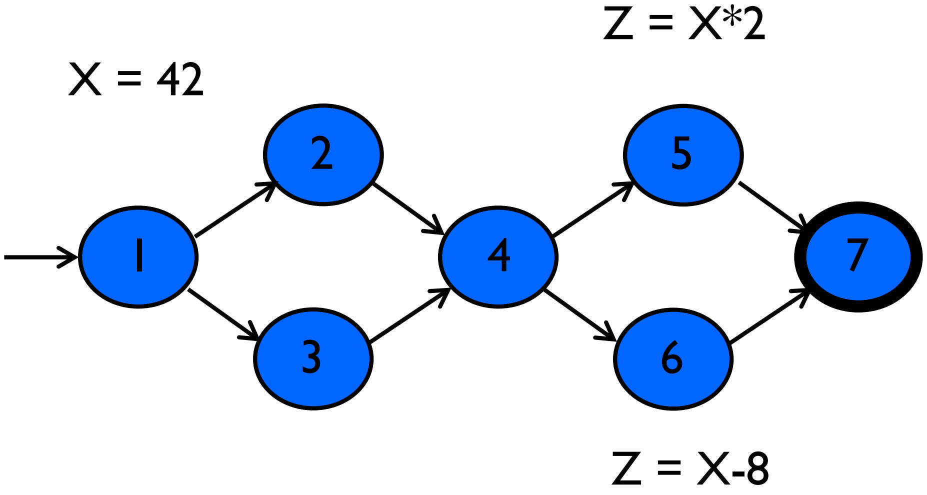


```
     main()
     {
     int work;
0.   double payment =0;
1.   scanf(“%d”, work);
2.   if (work > 0) {
3.         payment = 40;
4.   if (work > 20)
5.   {
6.         if(work <= 30)
7.               payment = payment + (work - 25) * 0.5;
8.         else
9.         {
10.              payment = payment + 50 + (work -30) * 0.1;
11.                    if (payment >= 3000)
12.                          payment = payment * 0.9;
13.        }
14.  }
15.  }
16.  printf(“Final payment”, payment);
```

Data Flow Criteria

Goal: Try to ensure that values are computed and used correctly

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



X = 42

Z = X*2

Z = X-8

Defs: def (1) = {X}

def (5) = {Z}

def (6) = {Z}

Uses: use (5) = {X}

use (6) = {X}

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

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) such that a variable *v* is defined at l_i and used at l_j

- Def-clear : A path from l_i to l_j is *def-clear* with respect to variable *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 with respect to *v* from a def of *v* to a use of *v*
- du (n_i, n_j, *v*) – the set of du-paths from n_i to n_j
- du (n_i, *v*) – the set of du-paths that start at n_i

Touring DU-Paths

- A test path *p* *du-tours* subpath *d* with respect to *v* if *p* tours *d* and the subpath taken is def-clear with respect to *v*

- Sidetrips can be used, just as with previous touring

- Three criteria
  - Use every def
  - Get to every use
  - Follow all du-paths

Data Flow Test Criteria

- First, we make sure every def reaches a use

All-defs coverage (ADC) : For each set of du-paths $S = du(n, v)$, TR contains at least one path d in S.

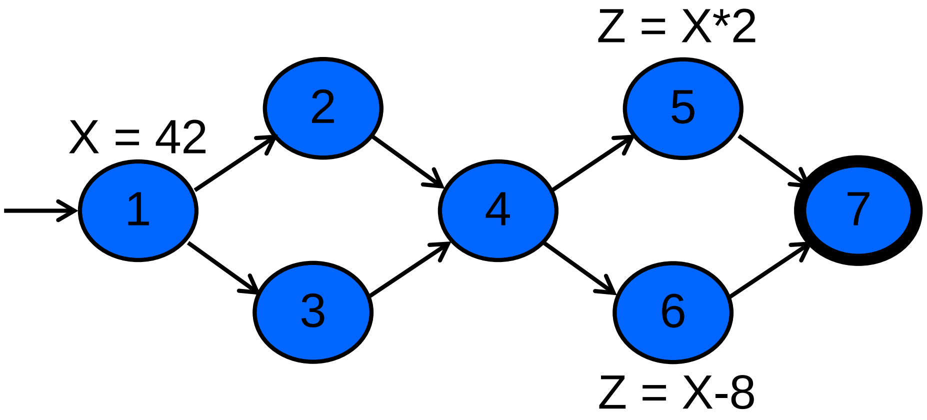
- Then we make sure that every def reaches all possible uses

All-uses coverage (AUC) : For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.

- Finally, we cover all the paths between defs and uses

All-du-paths coverage (ADUPC) : For each set $S = du(ni, nj, v)$, TR contains every path d in S.

Data Flow Testing Example



All-defs for *X*
[1, 2, 4, 5]

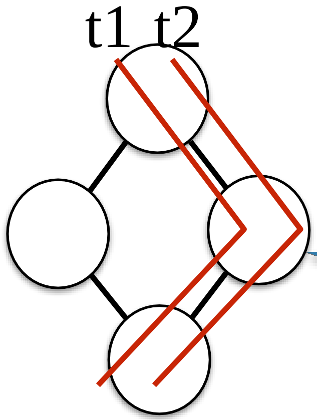
All-uses for *X*
[1, 2, 4, 5]
[1, 2, 4, 6]

All-du-paths for *X*
[1, 2, 4, 5]
[1, 3, 4, 5]
[1, 2, 4, 6]
[1, 3, 4, 6]

Data Flow Testing Criteria

- Seven data flow testing criteria
  - All-defs
  - All-c-uses
  - All-p-uses
  - All-p-uses/some-c-uses
  - All-c-uses/some-p-uses
  - All-uses
  - All-du-paths

Motivation



Control-flow graph

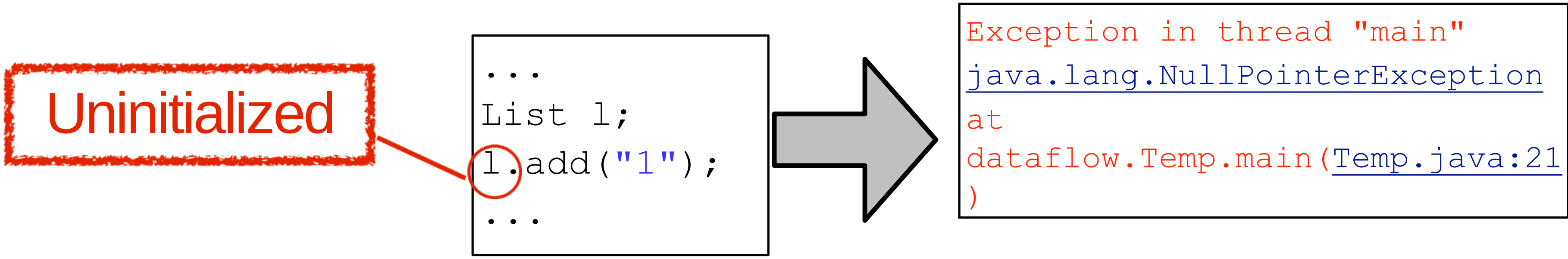
Are t1 and t2 identical?

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)



Uninitialized

```
...
List l;
l.add("1");
...
```

```
Exception in thread "main"
java.lang.NullPointerException
at
dataflow.Temp.main(Temp.java:21
)
```

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

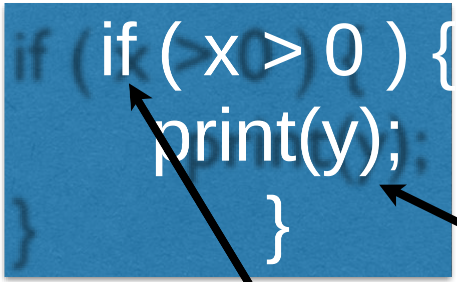
  - *e.g.,*   y = sqrt(**x**)

  •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

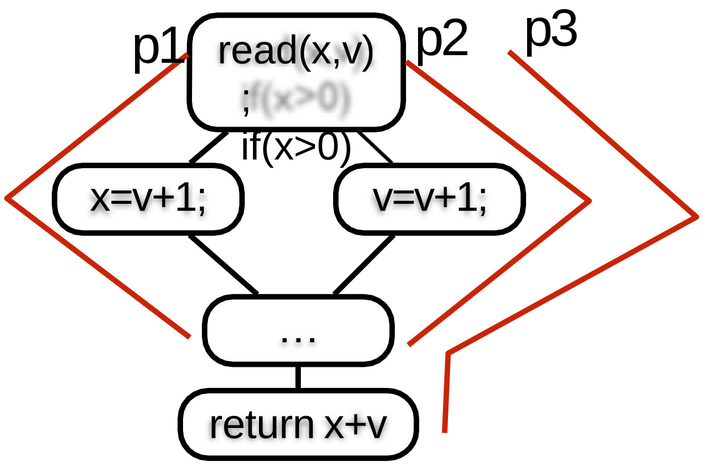
- *e.g.,* **y** = **y** + x

•as an call-by-reference parameter in a subroutine call

- *e.g.,* increment(&**y**)

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



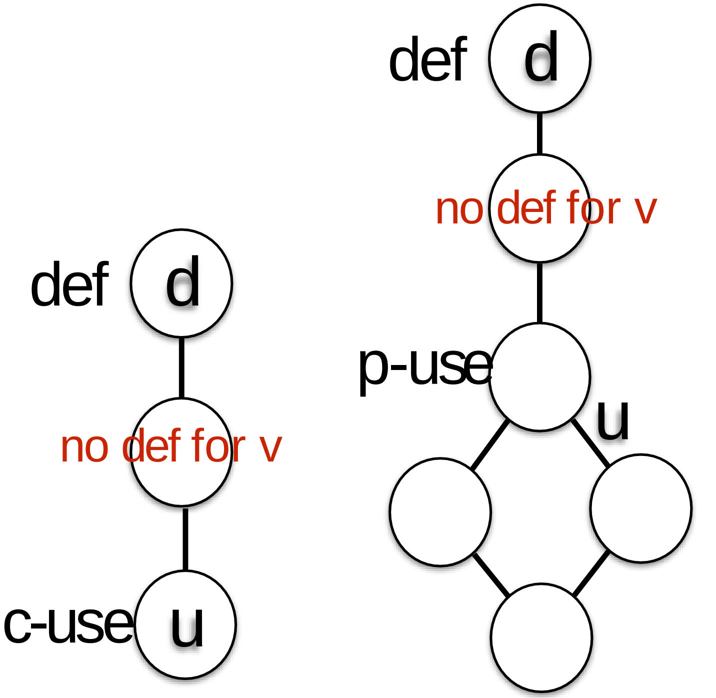
Control-flow graph

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

p1, p2 are not complete, while p3 is

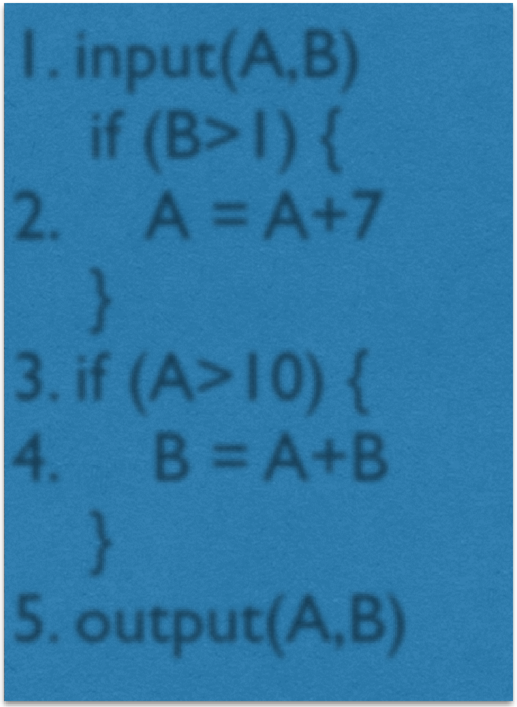
Definition-Use Pair (DU-Pair)

- 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**

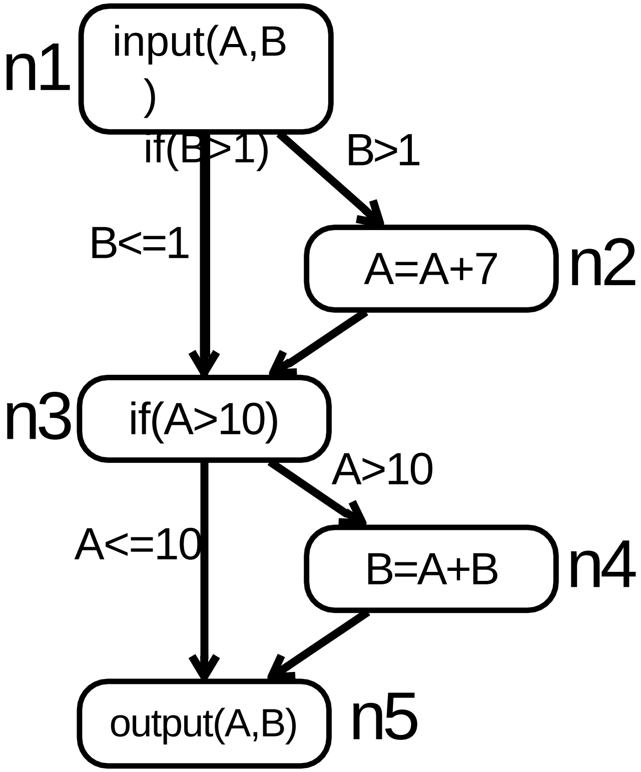


c-use example p-use example

DU-Pair: Example 1

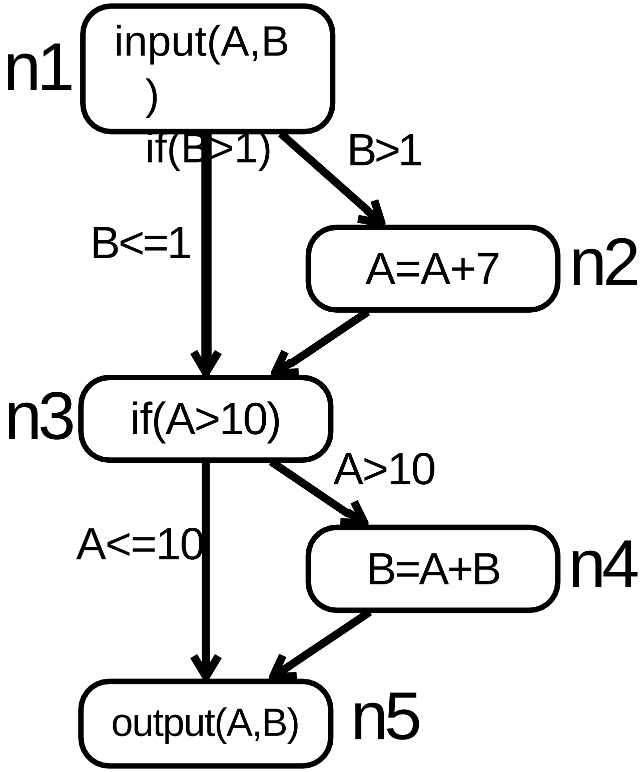


```
1. input(A,B)
   if (B>1) {
2.     A = A+7
   }
3. if (A>10) {
4.     B = A+B
   }
5. output(A,B)
```



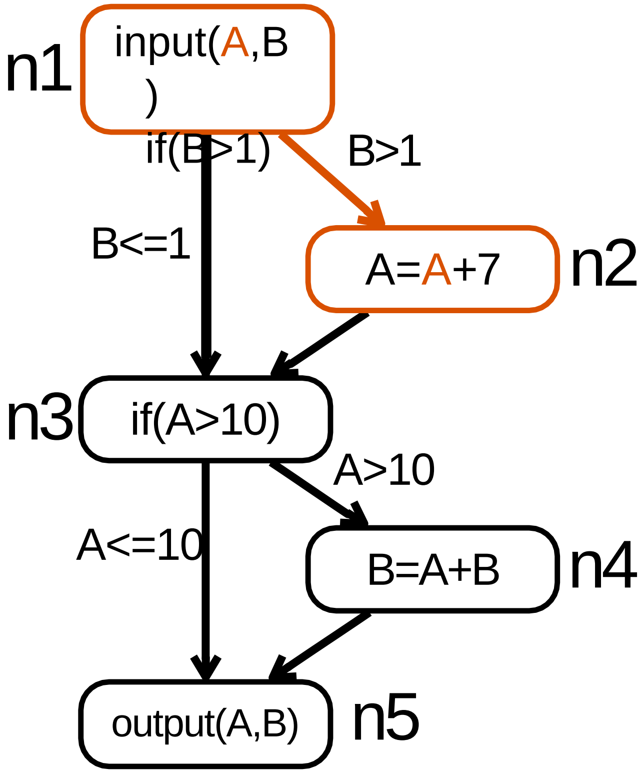
Identifying DU-Pairs – Variable **A**

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>



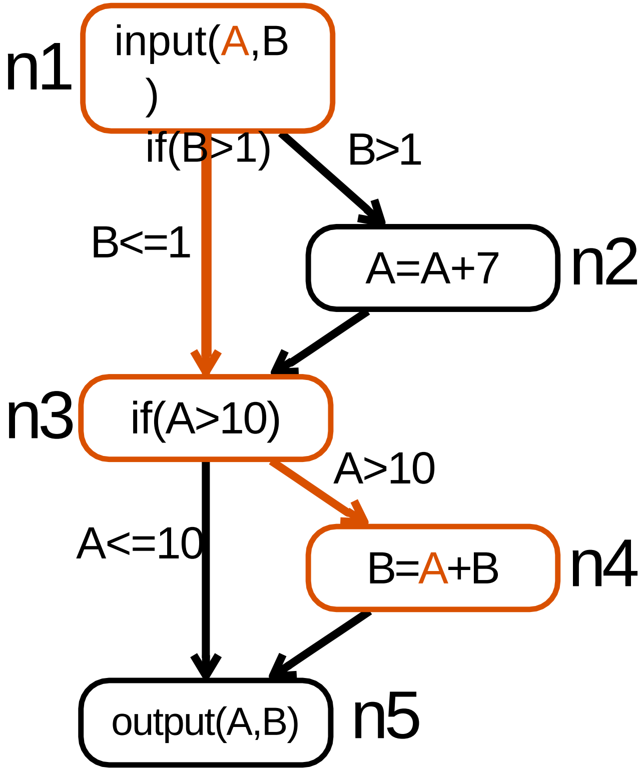
Identifying DU-Pairs – Variable **A**

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>



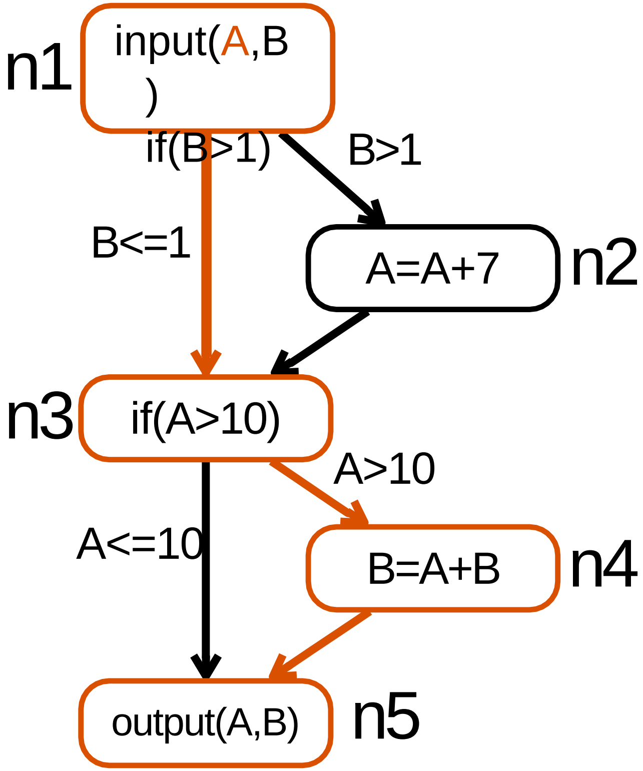
Identifying DU-Pairs – Variable **A**

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>



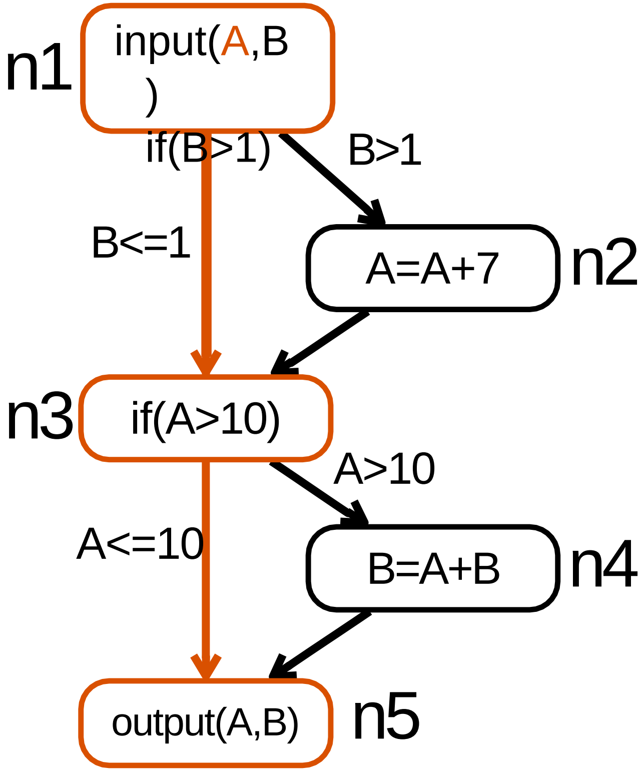
Identifying DU-Pairs – Variable **A**

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>



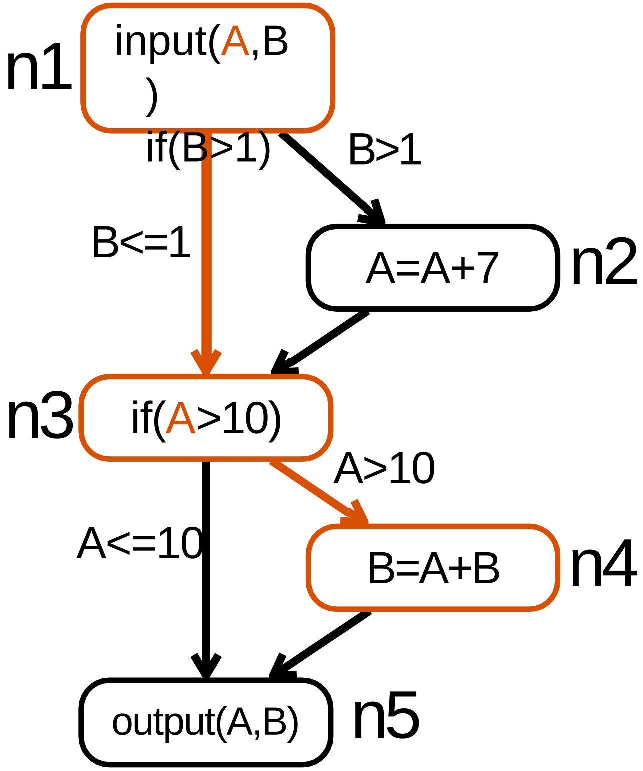
Identifying DU-Pairs – Variable **A**

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>



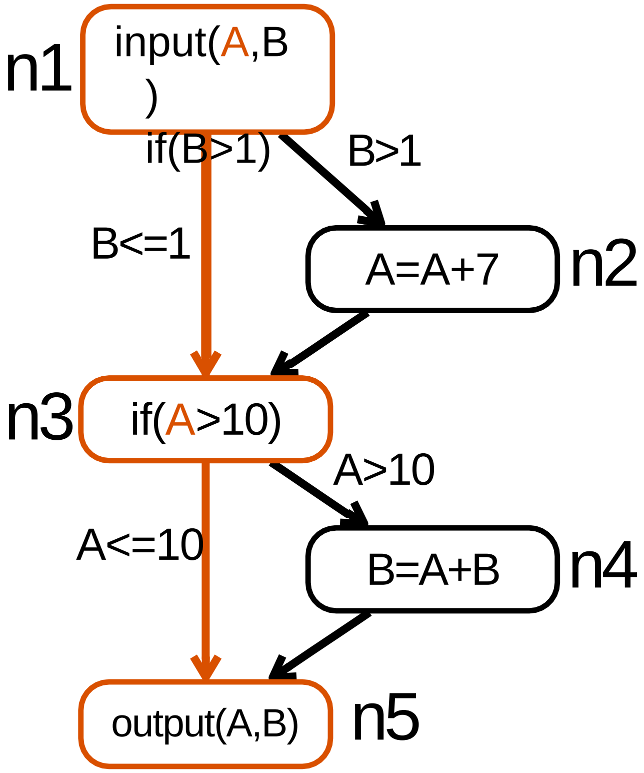
Identifying DU-Pairs – Variable **A**

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>



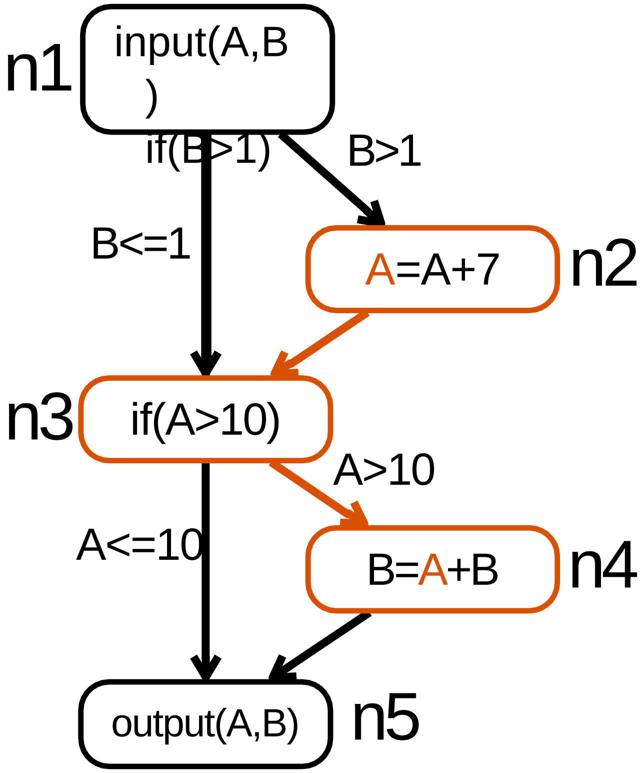
Identifying DU-Pairs – Variable **A**

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>



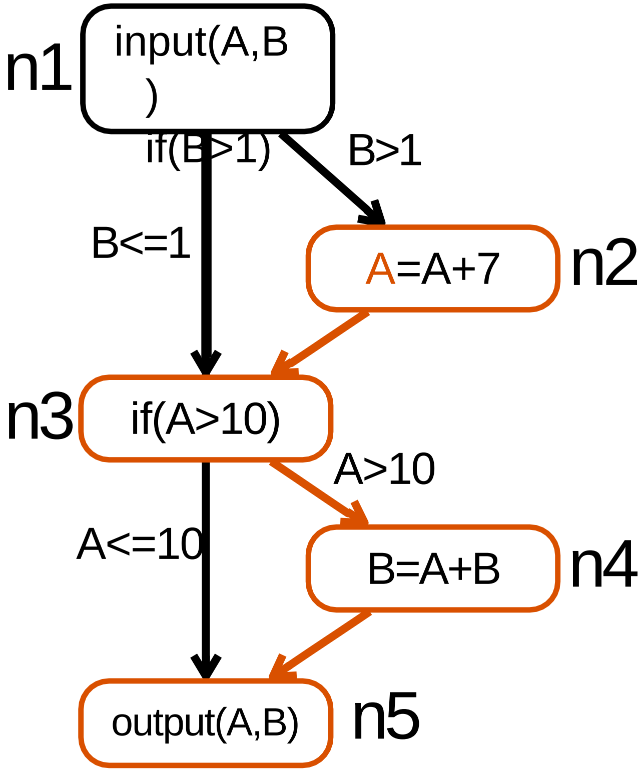
Identifying DU-Pairs – Variable **A**

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>



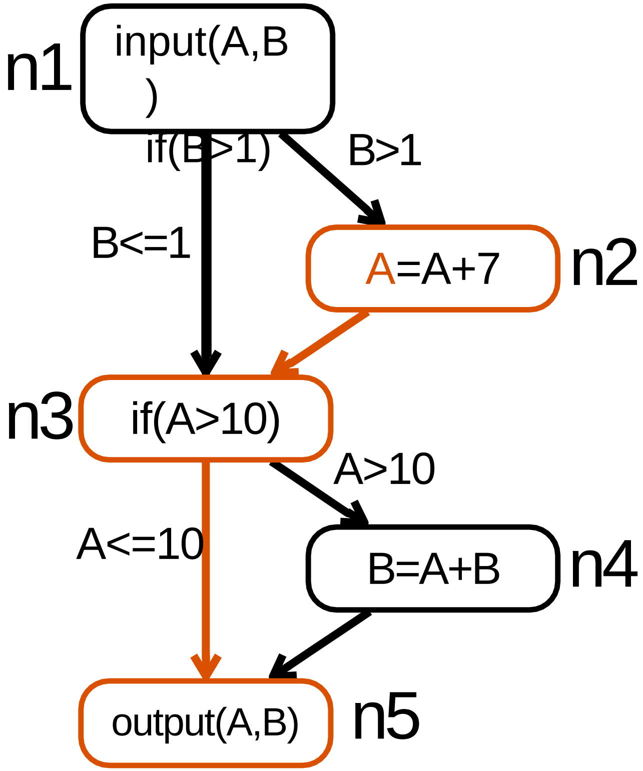
Identifying DU-Pairs – Variable **A**

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>



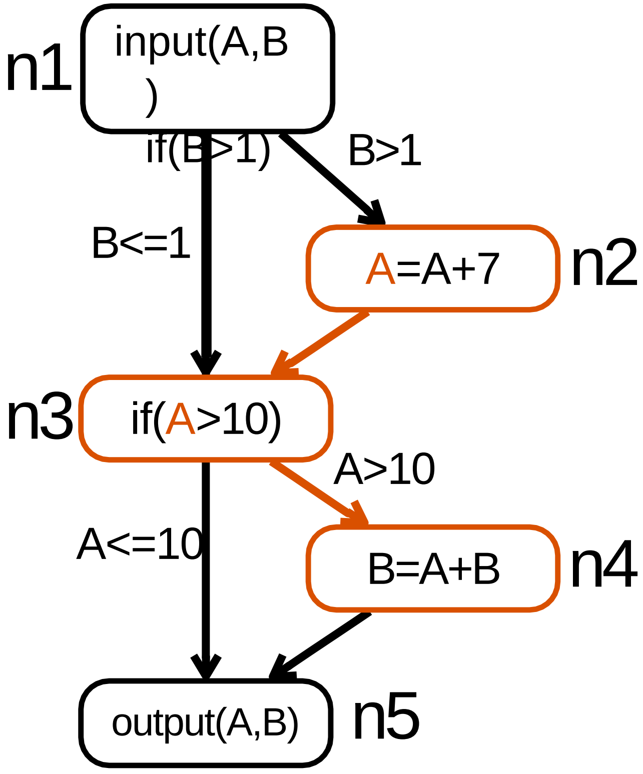
Identifying DU-Pairs – Variable **A**

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>



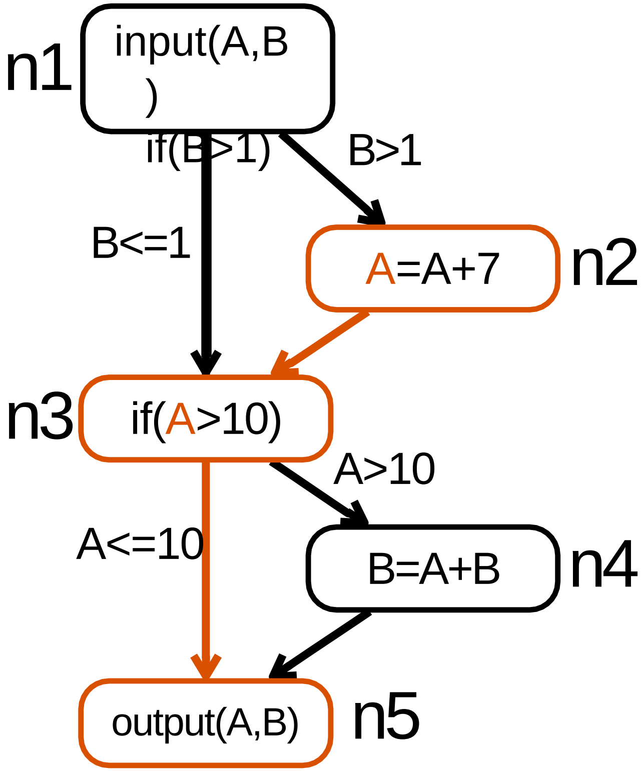
Identifying DU-Pairs – Variable **A**

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>



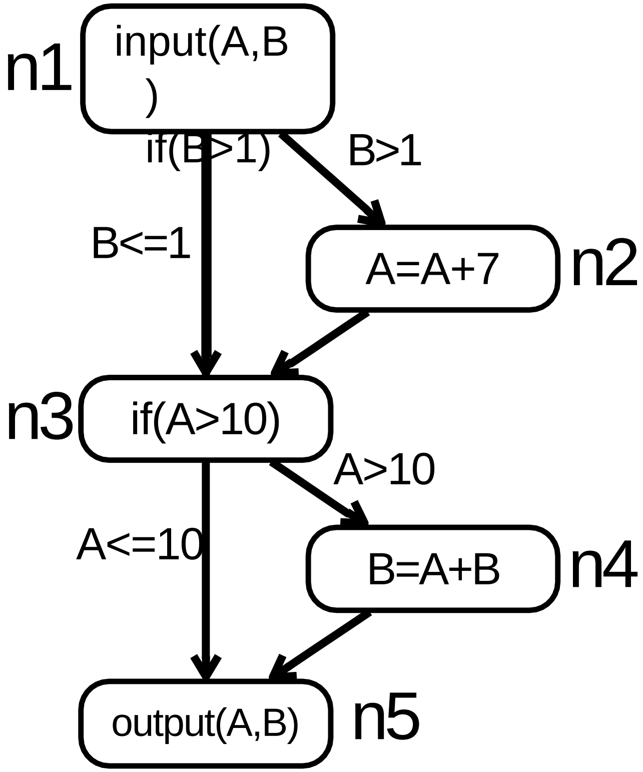
Identifying DU-Pairs – Variable **A**

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>



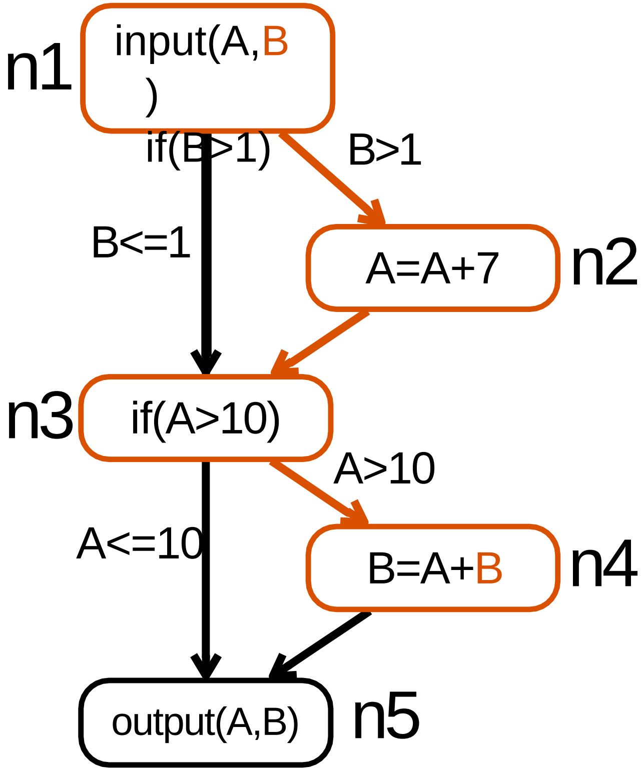
Identifying DU-Pairs – Variable **B**

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**

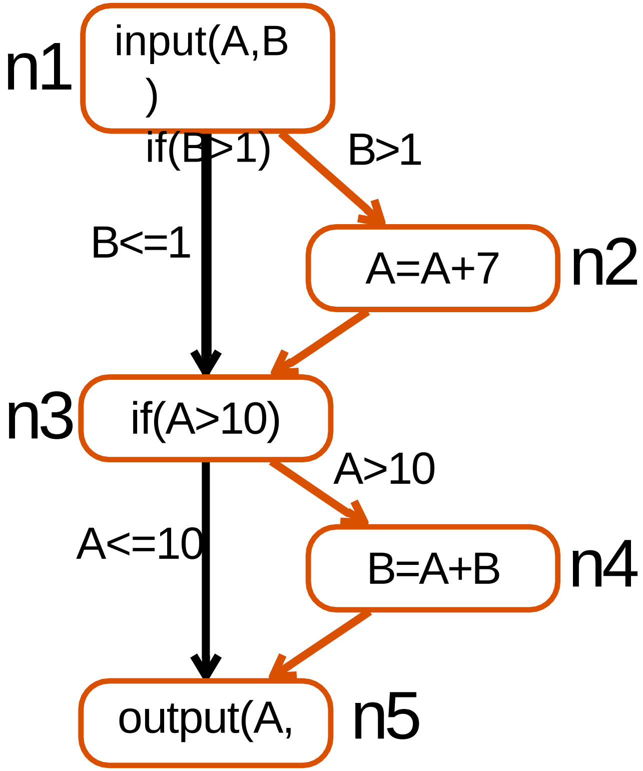
  •for every program variable **v**, at least one def-clear path from every definition of **v** to at least one c-use or one p-use of **v** must be covered

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?

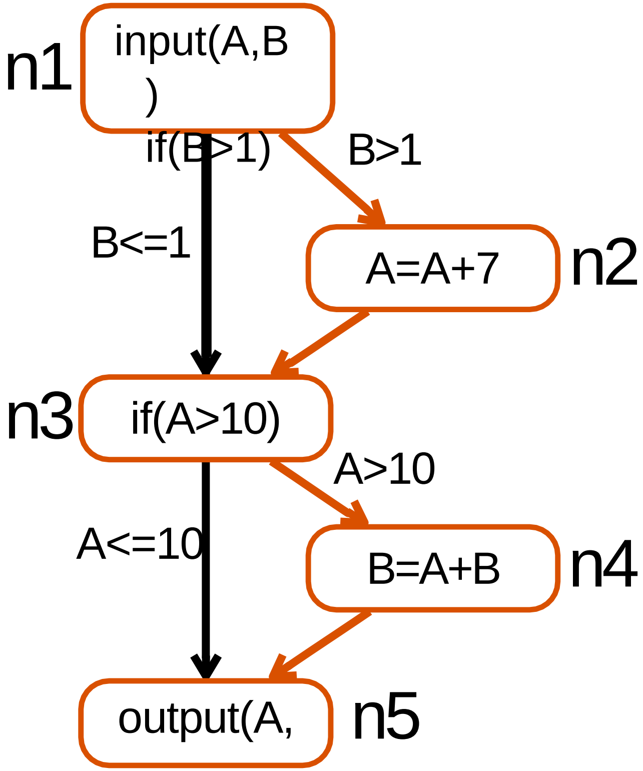
Def-Clear Paths subsumed by <1,2,3,4,5> for Variable **A**

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



Dataflow Test Coverage Criteria

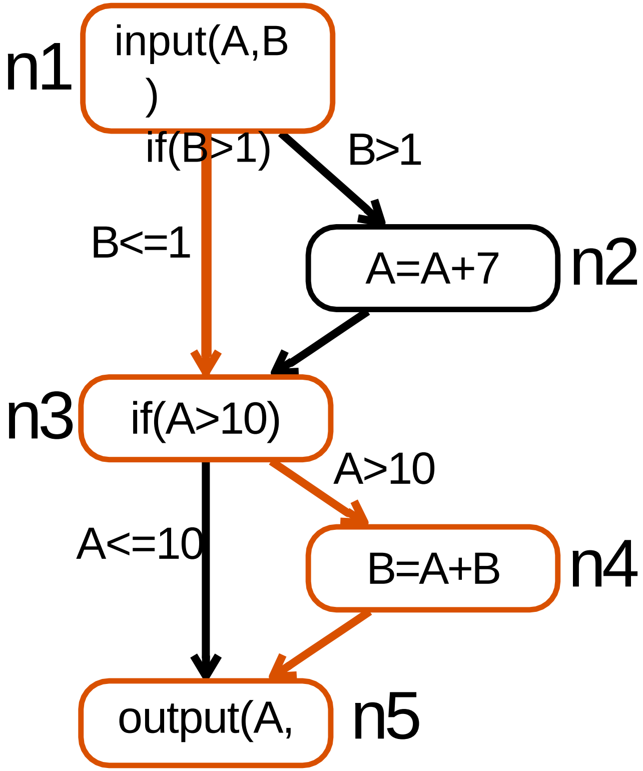
- Since <1,2,3,4,5> covers at least one def-clear path from every definition of **A** or **B** to at least one c-use or p-use of **A** or **B**, All-Defs coverage is achieved

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?

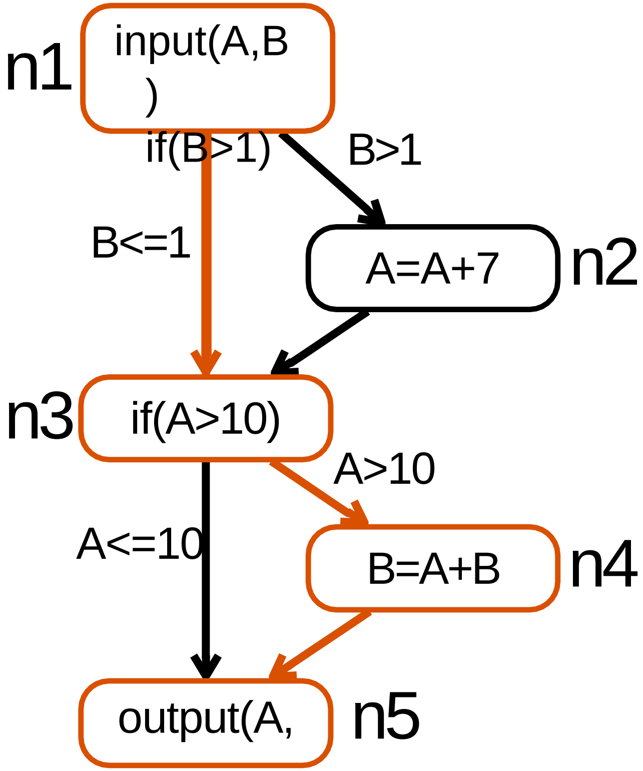
Def-Clear Paths Subsumed by <1,3,4,5> for Variable **A**

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



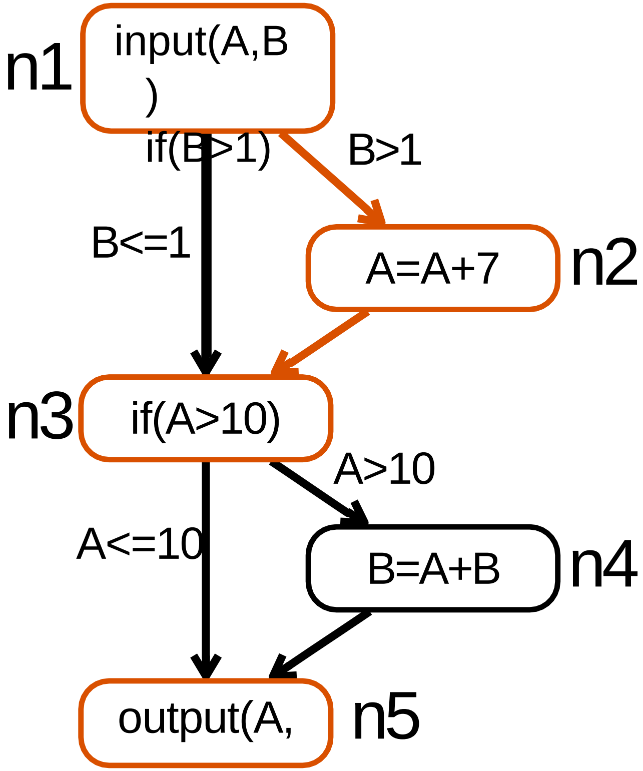
Def-Clear Paths Subsumed by <1,3,4,5> for Variable **B**

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



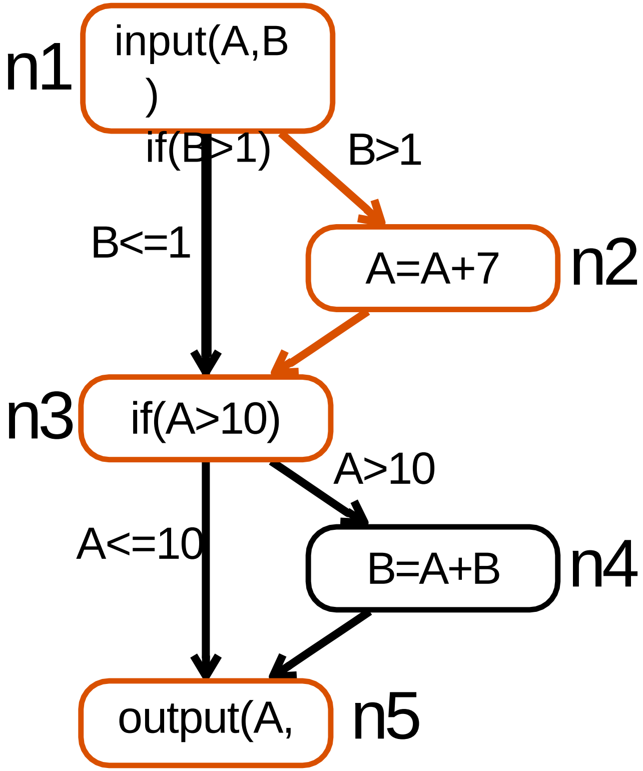
Def-Clear Paths Subsumed by <1,2,3,5> for Variable **A**

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



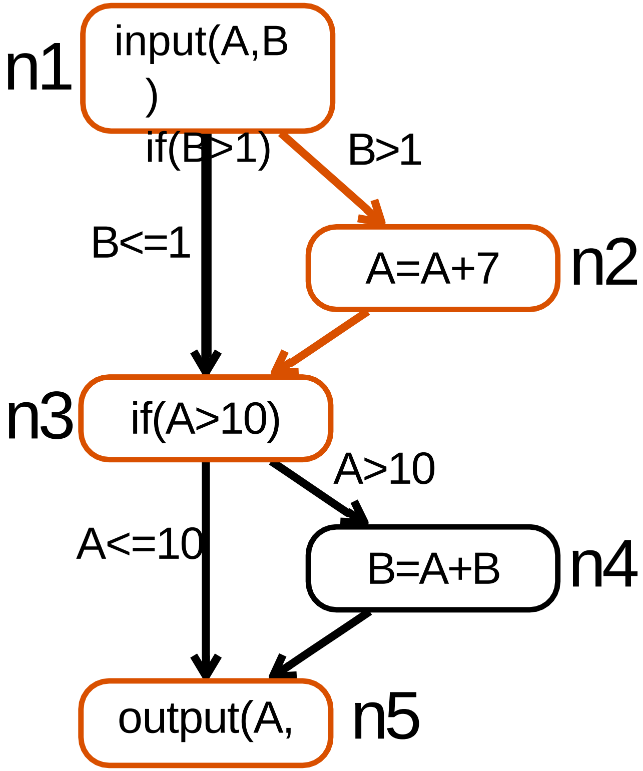
Def-Clear Paths Subsumed by <1,2,3,5> for Variable **A**

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



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

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



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

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 <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.

NOTE!

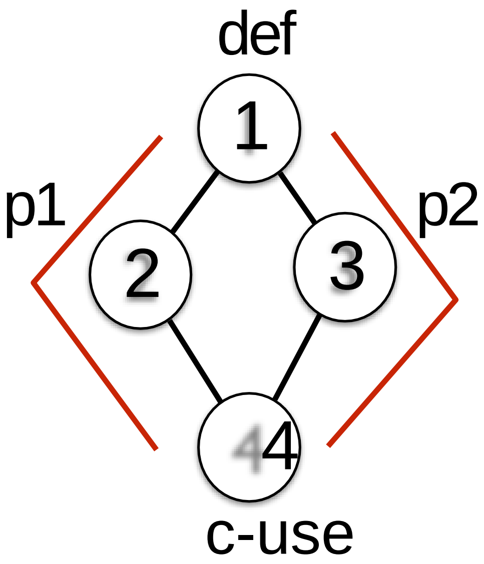
Identifying DU-Paths

du-pair	path(s)	du-path?
(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 **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**:

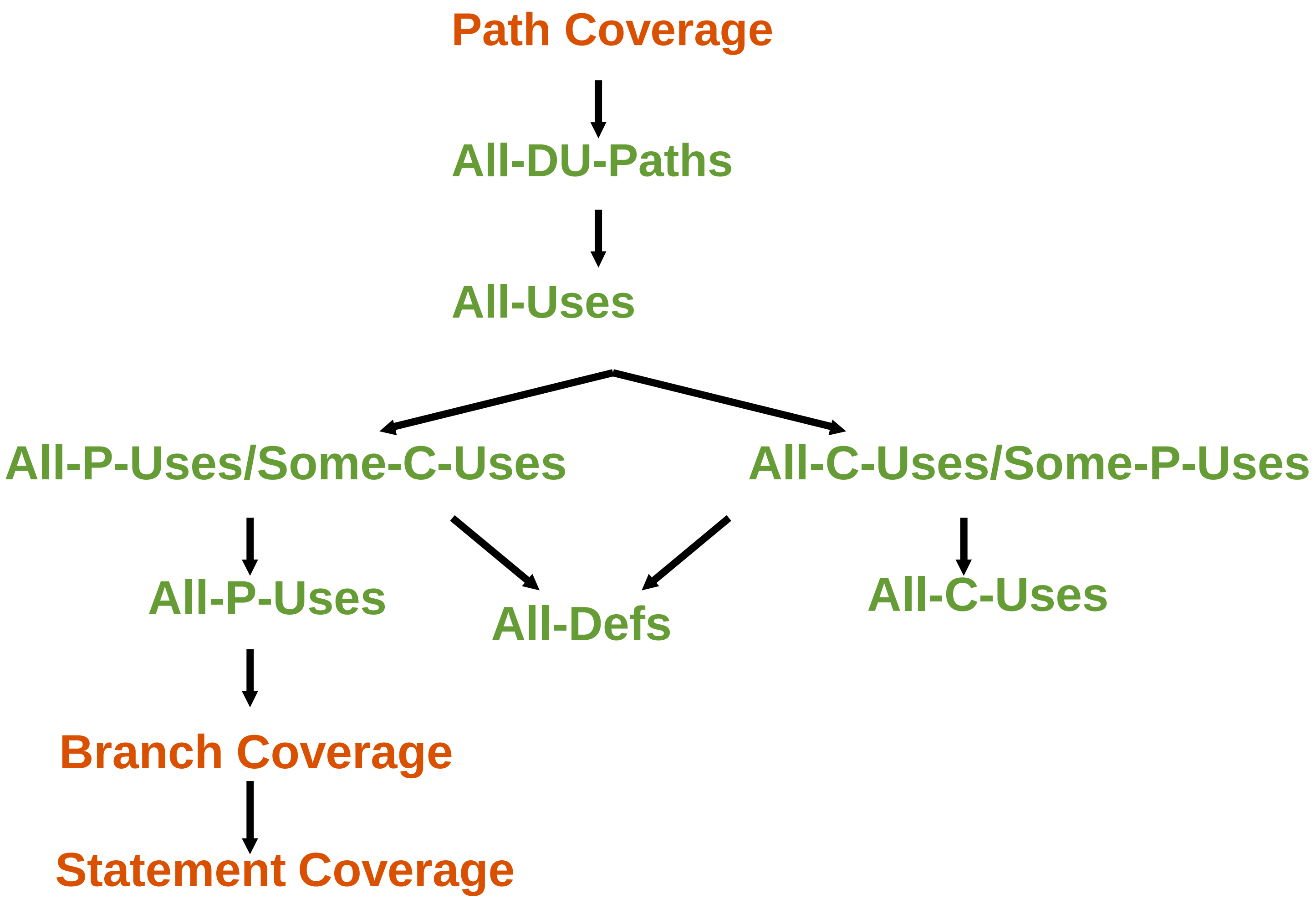
  for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use of **v** must be covered

  - 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

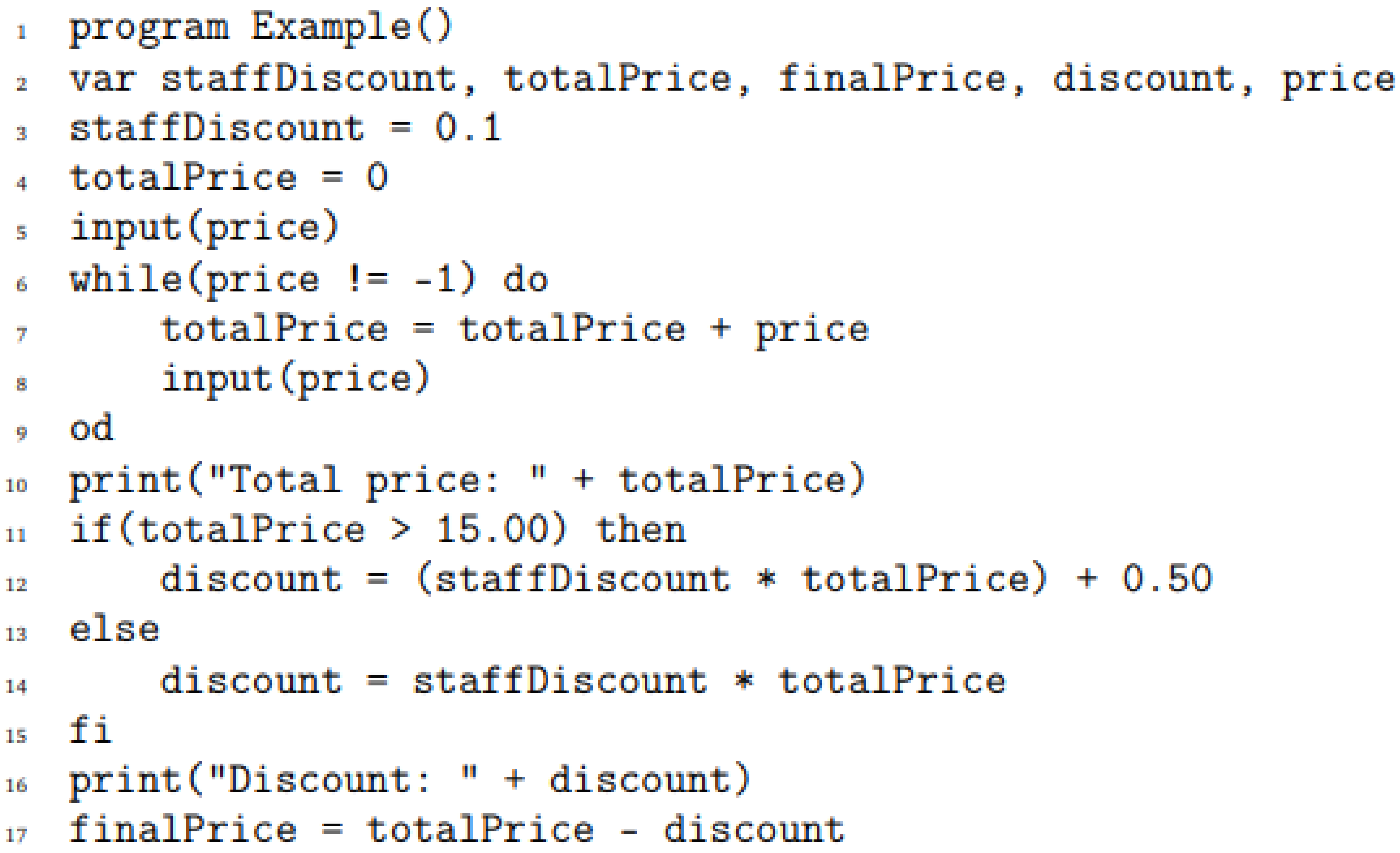
Branch Coverage

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

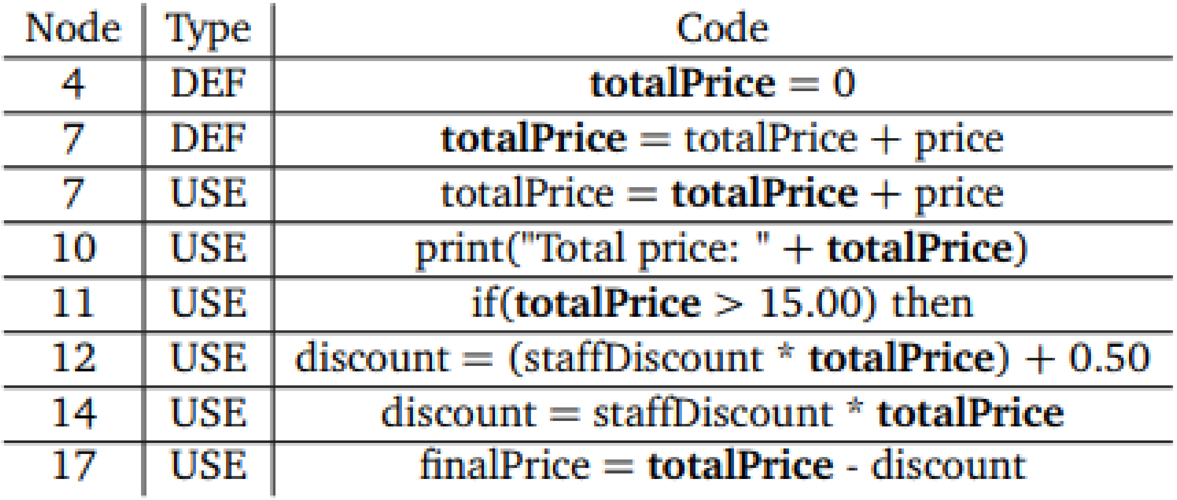
Example:1



```
program Example()
var staffDiscount, totalPrice, finalPrice, discount, price
staffDiscount = 0.1
totalPrice = 0
input(price)
while(price != -1) do
    totalPrice = totalPrice + price
    input(price)
od
print("Total price: " + totalPrice)
if(totalPrice > 15.00) then
    discount = (staffDiscount * totalPrice) + 0.50
else
    discount = staffDiscount * totalPrice
fi
print("Discount: " + discount)
finalPrice = totalPrice - discount
```

- Identify the def and ref nodes (p-use or c-use) with reference to Total_Price Variable
- Identify the du path w.r.t Total_Price
- Identify all def clear paths for all the variables

Solution



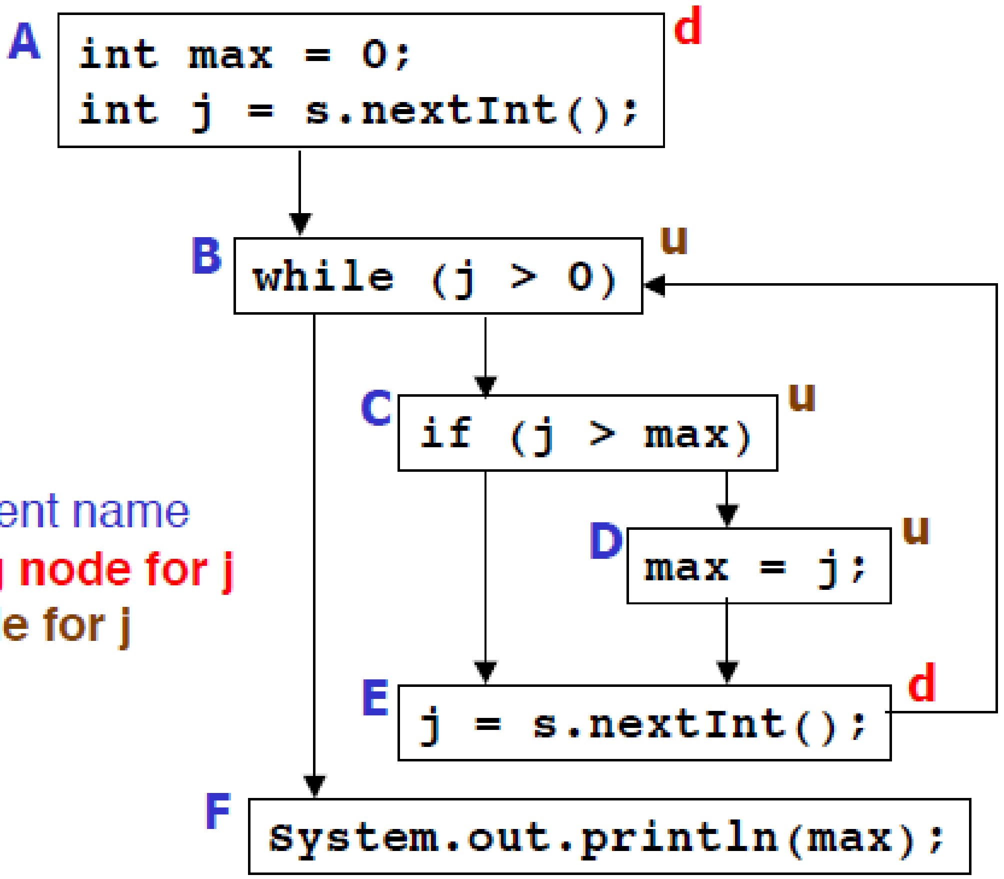
Node	Type	Code
4	DEF	**totalPrice** = 0
7	DEF	**totalPrice** = totalPrice + price
7	USE	totalPrice = **totalPrice** + price
10	USE	print("Total price: " + **totalPrice**)
11	USE	if(**totalPrice** > 15.00) then
12	USE	discount = (staffDiscount * **totalPrice**) + 0.50
14	USE	discount = staffDiscount * **totalPrice**
17	USE	finalPrice = **totalPrice** - discount

- Du path w.r.t Price
- 5-6
- 8-9-6

Example:2

```
int max = 0;
int j = s.nextInt();
while (j > 0)
if (j > max) {
max = j;
}
j = s.nextInt();
}
System.out.println(max);
```

- Identify the Def and Use node for variable j?



```
int max = 0;
int j = s.nextInt();
```

```
while (j > 0)
```

```
if (j > max)
```

```
max = j;
```

```
j = s.nextInt();
```

```
System.out.println(max);
```

Legend

A..F Segment name

d defining node for j

u use node for j

```
miles_per_gallon ( miles, gallons, price : INTEGER )
if gallons = 0 then // Watch for division by zero!!
    Print(“You have “ + gallons + “gallons of gas”)
else if miles/gallons > 25
  then print( “Excellent car. Your mpg is “ + miles/gallon)
    else print( “You must be going broke. Your mpg is “ + miles/gallon + “ cost “ + gallons * price)
fi
end
```

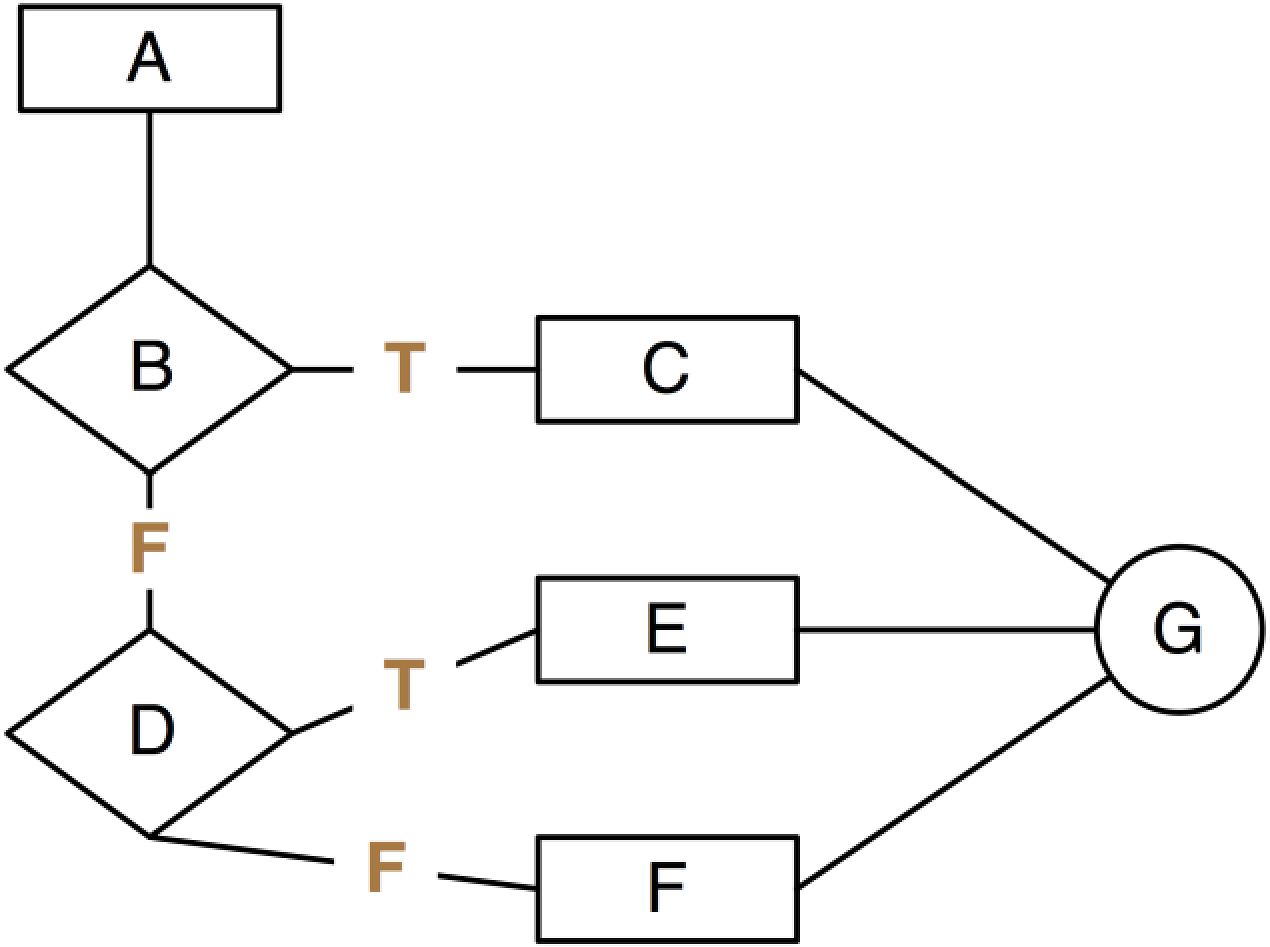
- Identify def, c-use and p-use for each variable along with node?

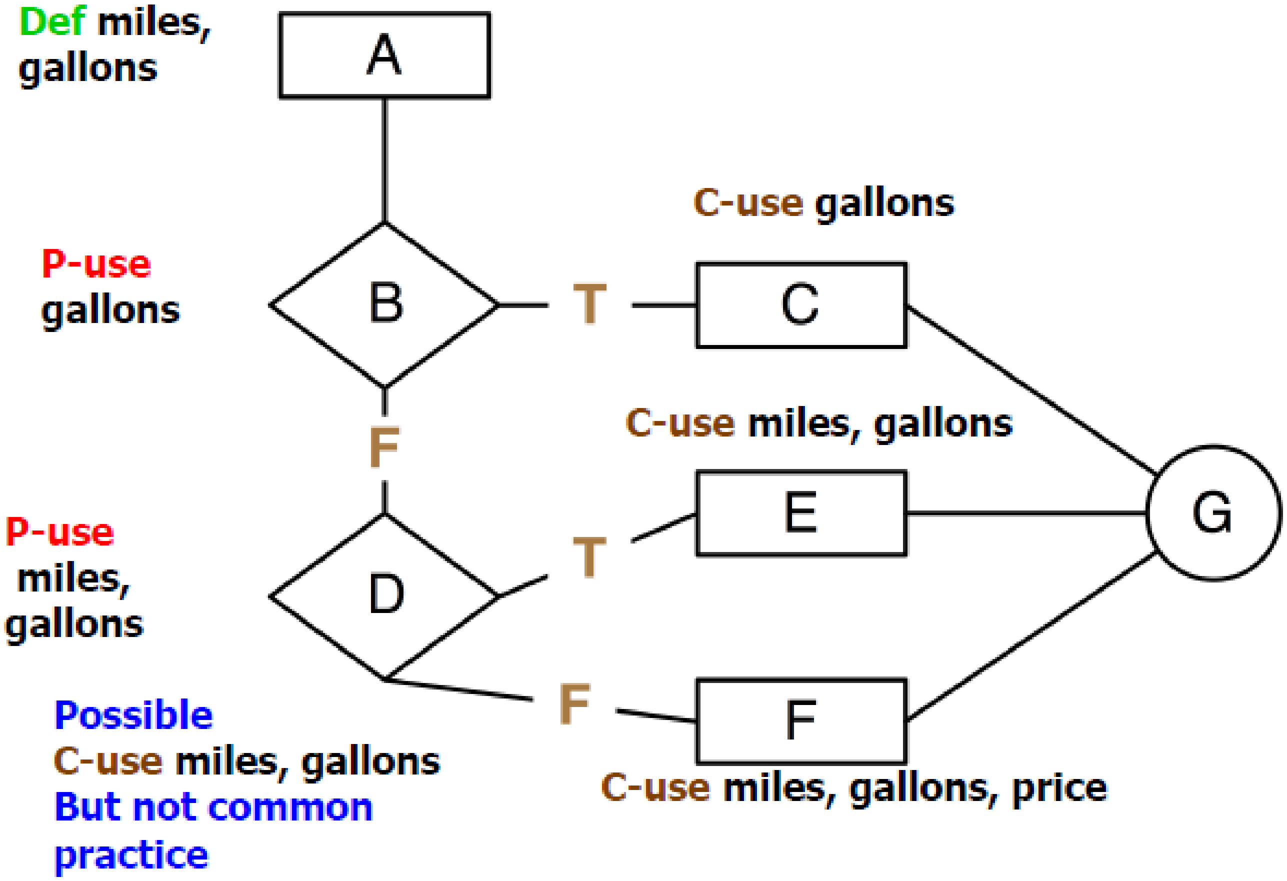
gasguzzler (miles, gallons, price : INTEGER)	A
if gallons = 0 then	B

// Watch for division by zero!!
Print(“You have “ + gallons + “gallons of gas”)
	C

else if miles/gallons > 25	D
then print(“Excellent car. Your mpg is “ + miles/gallon)	E
else print(“You must be going broke. Your mpg is “ + miles/gallon + “ cost “ + gallons * price)	F

fi
end
	G


A
B
T
C
F
E
G
T
D
F
F



Def miles, gallons

P-use gallons

C-use gallons

C-use miles, gallons

P-use miles, gallons

Possible C-use miles, gallons But not common practice

C-use miles, gallons, price

- All-Defs
- Each definition of each variable for at least one use of the definition
  - **A B D**
- All-C-uses
- At least one path of each variable to each c-use of the Definition
  - **A B D E**
  - **A B D F**
  - **A B D**

- All-P-uses/Some-C-uses
- At least one path of each variable definition to each puse of the variable. If any variable definitions are not covered by p-use, then use c-use
  - **A B D**
- All-uses
- At least one path of each variable definition to each puse and each c-use of the definition
  - **A B D A**
  - **A B D E**
  - **A B D F**

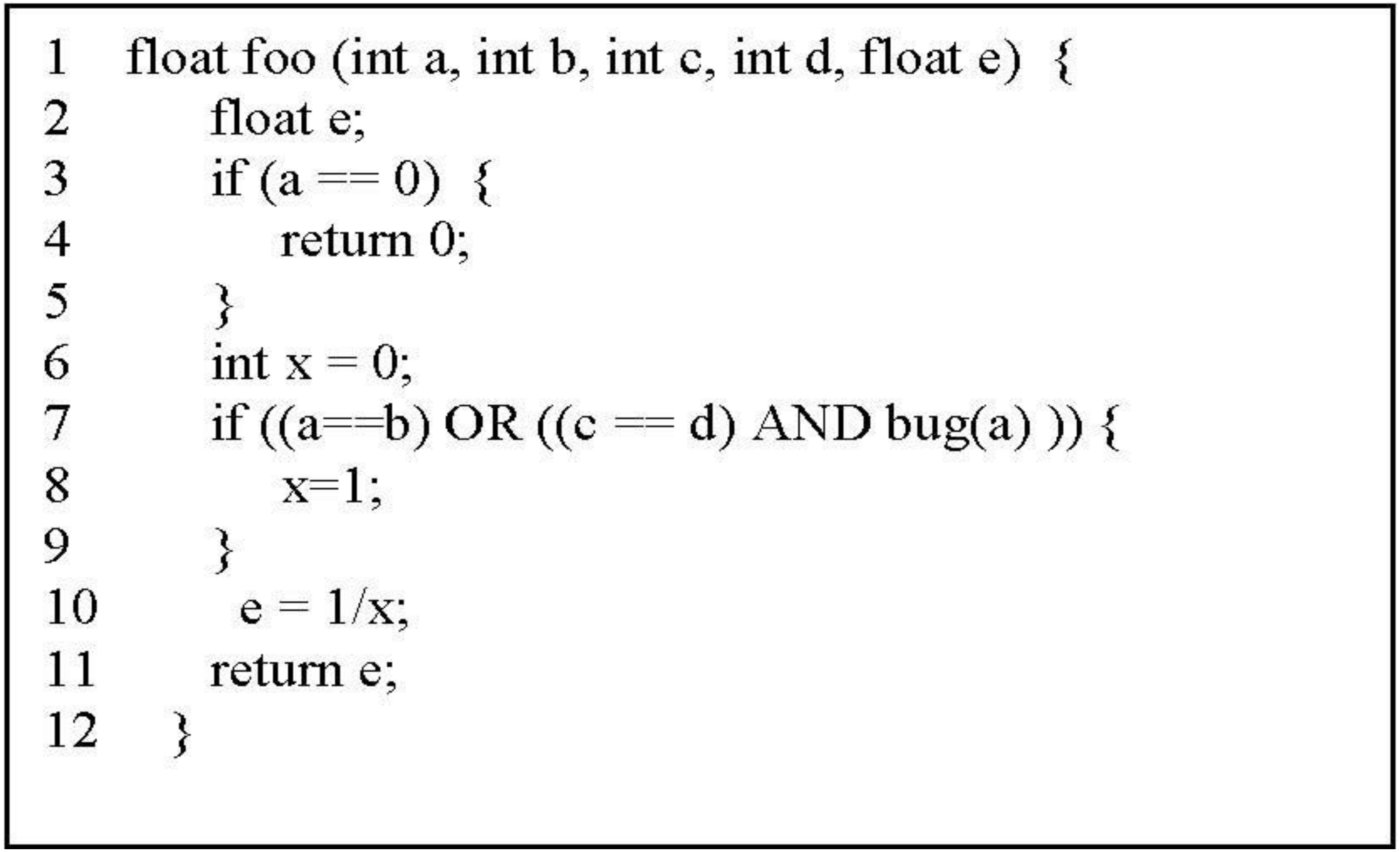
- All-P-uses
- At last one path of each variable to each p-use of the definition
- **A B D**
- All-C-uses/Some-P-uses
- At least one path of each variable definition to each cuse of the variable. If any variable definitions are not covered use p-use
  - **A B D E**
  - **A B D F**
  - **A B D**

- All-Defs
- **Each definition of each variable for at least one use of the definition**
    - **A B**
- All-C-uses
- **At least one path of each variable to each c-use of the definition**
    - **A B C**
    - **A B D E**
    - **A B D F**
    - **A B D**

Coverage Testing

- The adequacy of the test cases is often measured with a metric called coverage.
- *Coverage* is a measure of the completeness of the set of test cases.
- We write methods to ensure they are testable – most simply by having the method return a value.
- In a test case we predetermine the “answer” that is returned when the method is called with certain parameters so that our testing returns that predetermined value.

Example: **Sample Code for Coverage Analysis**



```
float foo (int a, int b, int c, int d, float e)  {
      float e;
      if (a == 0)  {
           return 0;
      }
      int x = 0;
      if ((a==b) OR ((c == d) AND bug(a) )) {
           x=1;
      }
        e = 1/x;
      return e;
   }
```

Method Coverage

- *Method coverage* is a measure of the percentage of methods that have been called by your test cases.
- Tests should call 100% of the system methods. \
- We need to ensure you have 100% method coverage.
- In the example we attain 100% method coverage by calling the foo method.
- Test Case 1: call the method **foo(0, 0, 0, 0, 0.)**, expected return value of 0.
- Through this one call we attain 100% method coverage.
  - This is assuming that bug(a) is not a method!!!!!
  - If bug(a) is considered a method, then we have achieved 50% method coverage
    - Is there a single test case that can generate 100% method coverage in this case?

Statement Coverage

- *Statement coverage* is a measure of the percentage of program statements that are run when your tests are executed.
- The objective should be to achieve 100% statement coverage through your testing.
- Identify the cyclomatic number and executing this minimum set of test cases will make this statement coverage achievable.
- In Test Case 1, we executed the program statements on lines 1-5 out of 12 lines of code - a 42% (5/12) statement coverage from Test Case 1.

Test Case 1: call the method **foo(0, 0, 0, 0, 0.)**, expected return value of 0.

Example

```
read (x); read (y);
if x > 0 then
        write ("1");
else
        write ("2");
end if;
if y > 0 then
        write ("3");
else
        write ("4");
end if;
```

{<x = 2, y = 3>, <x = - 13, y = 51>, <x = 97, y = 17>, <x = - 1, y = - 1>} covers all statements

{<x = - 13, y = 51>, <x = 2, y = - 3>} is minimal

Weakness of the criterion

```
if x < 0 then
        x := -x;
end if;
z := x;
```

{<x=-3} covers all statements

it does not exercise the case when x is positive and the then branch is not entered

Statement Coverage(Cont.)

- To attain 100% statement coverage, one should execute an additional test case.
- Test Case 2: the method call **foo(1, 1, 1, 1, 1.)**, expected return value of 1.
- This executes the program statements on lines 6-12 - a 100% statement coverage.

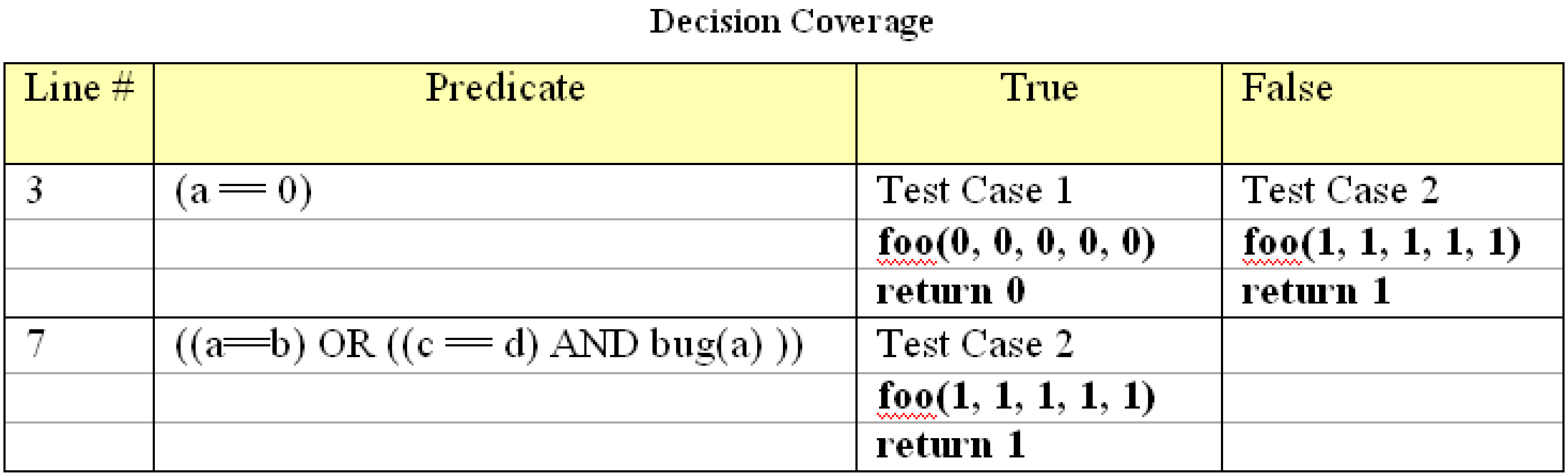
Decision/Branch Coverage

- *Decision or branch coverage* is a measure of how many of the Boolean expressions of the program have been evaluated as both true and false in the testing.
- The example program has two decision points – one on line 3 and the other on line 7.

```
3 if (a == 0) {
7 if ((a==b) OR ((c == d) AND bug(a) )) {
```

Decision/Branch Coverage(Cont.)

- For decision/branch coverage, evaluate an entire Boolean expression as one true-or-false predicate even if it contains multiple logical-and or logical-or operators.
- We need to ensure that each of these predicates (compound or single) is tested as both true and false.



Decision Coverage

Line #	Predicate	True	False
3	(a == 0)	Test Case 1	Test Case 2
		foo(0, 0, 0, 0, 0)	**foo(1, 1, 1, 1, 1)**
		return 0	**return 1**
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2	
		foo(1, 1, 1, 1, 1)	
		return 1	

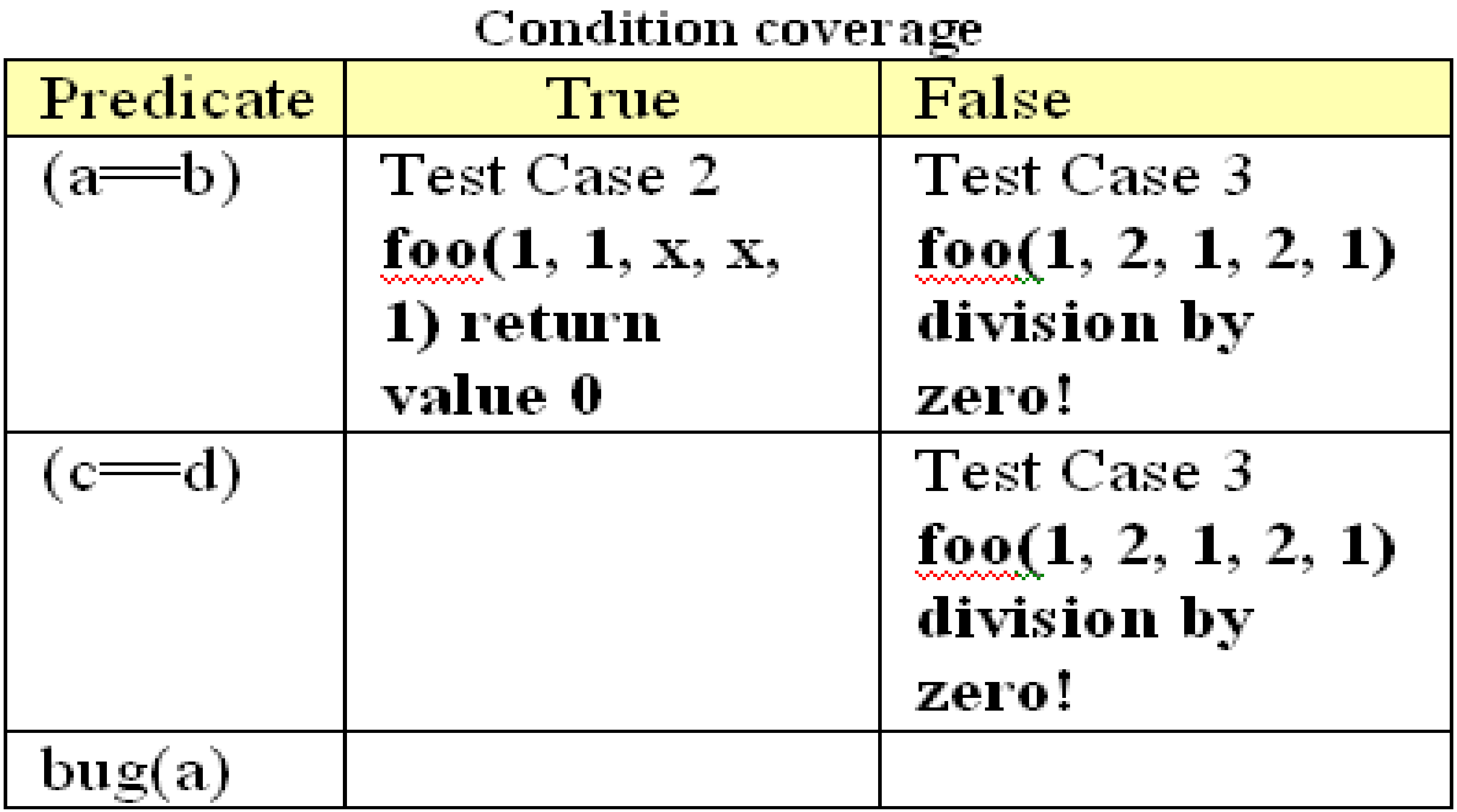
Decision/Branch Coverage(Cont.)

- Three of the four necessary conditions - 75% branch coverage.
- We add Test Case 3 : **foo(1, 2, 1, 2, 1)** to bring us to 100% branch coverage(making the Boolean expression False).
- Expected return value?
- The objective is to achieve 100% branch coverage in your testing.
- In large systems only 75%-85% is practical.
- Only 50% branch coverage is practical in very large systems of 10 million source lines of code or more.

Condition Coverage

- *Condition coverage* reports the true or false outcome of *each* Boolean sub-expression of a compound predicate.
- In line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a).
- Condition coverage measures the outcome of each of these sub-expressions independently of each other.
- With condition coverage, you ensure that each of these sub-expressions has independently been tested as both true and false.

Condition Coverage(cont.)



Condition coverage

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, x, x, 1) return value 0**	Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!**
(c==d)		Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!**
bug(a)		

Condition Coverage(cont.)

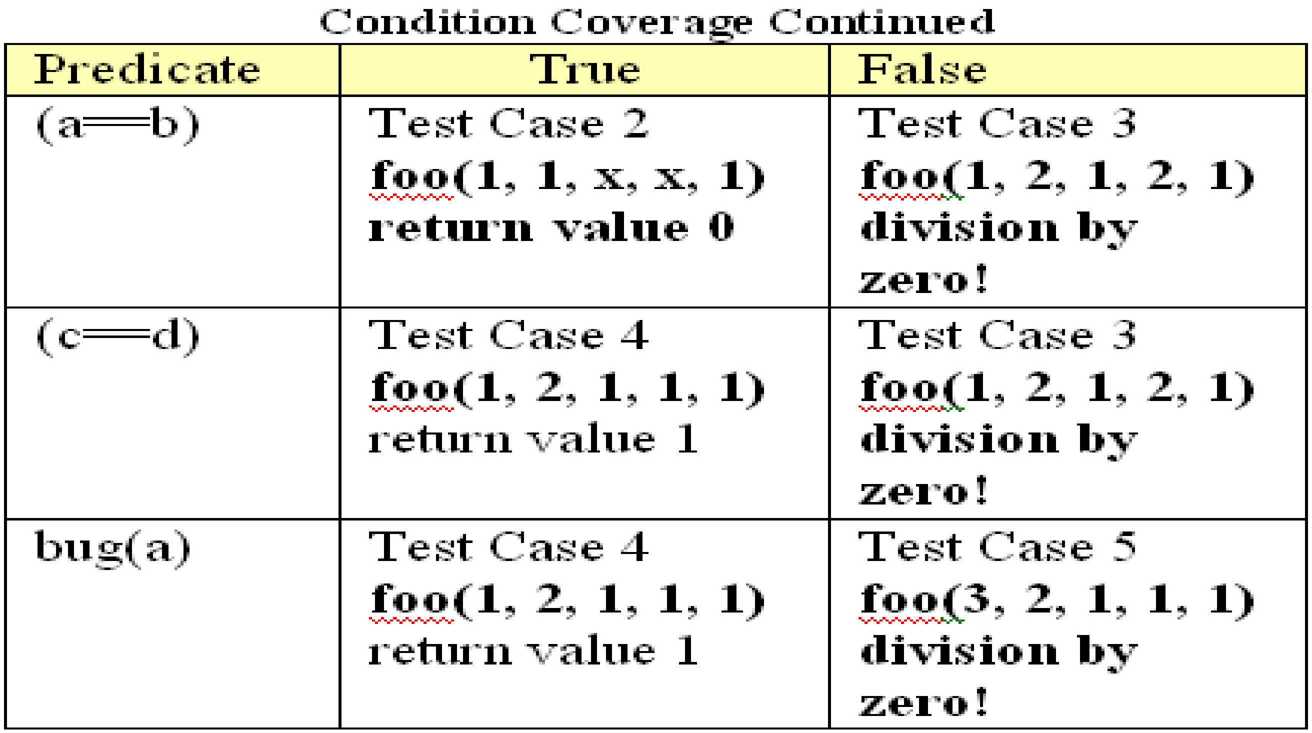
- Condition coverage of the table is only 50%.
- The true condition (c==d) has never been tested.
- short-circuit Boolean has prevented the method bug(int) from ever being executed.
- Suppose bug(int) returns a value of true when passed a value of a=1 and returns a false value if fed any integer greater than 1.

Condition Coverage(cont.)

- Test Case 4 address test (c==d) as true: **foo(1, 2, 1, 1, 1)**, expected return value 1.
- When we run the test case, the function bug(a) actually returns false, which causes our actual return value (division by zero) to not match our expected return value.
- This allows us to detect an error in the bug method. Without the addition of condition coverage, this error would not have been revealed.
- To finalize our condition coverage, we must force bug(a) to be false.

Condition Coverage(cont.)

- Test Case 5, **foo(3, 2, 1, 1, 1)**, expected return value “division by zero error”.
- The condition coverage thus far is shown in the Table.



Condition Coverage Continued

Predicate	True	False

(a==b)	Test Case 2
	foo(1, 1, x, x, 1)
	return value 0
		Test Case 3
		foo(1, 2, 1, 2, 1)
		division by zero!

(c==d)	Test Case 4
	foo(1, 2, 1, 1, 1)
	return value 1
		Test Case 3
		foo(1, 2, 1, 2, 1)
		division by zero!

bug(a)	Test Case 4
	foo(1, 2, 1, 1, 1)
	return value 1
		Test Case 5
		foo(3, 2, 1, 1, 1)
		division by zero!

Condition Coverage(cont.)

- There are no industry standard objectives for condition coverage, but we suggest that you keep condition coverage in mind as you develop your test cases.

- Our condition coverage revealed that some additional test cases were needed.

Condition-coverage criterion

- Select a test set T such that every edge of P’s control flow is traversed and all possible values of the constituents of compound conditions are exercised at least once
  - it is finer than edge coverage

Weakness

```
if x ≠ 0 then
        y := 5;
else
        z := z - x;
end if;
if z > 1 then
        z := z / x;
else
        z := 0;
end if;
```

{<x = 0, z = 1>, <x = 1, z = 3>} causes the execution of all edges, but fails to expose the risk of a division by zero

Summary

- A common programming mistake is referencing the value of a variable without first assigning a value to it.
- A data flow graph shows the processing flow through a module. In addition, it details the definition, use, and destruction of each of the module's variables.
- Enumerate the paths through the module. Then, for every variable, create at least one test case to cover every define-use pair.
- *Coverage* is a measure of the completeness of the set of test cases
- *Method coverage* is a measure of the percentage of methods that have been called by your test cases

Summary(cont.)

- *Statement coverage* is a measure of the percentage of program statements that are run when your tests are executed
- The objective should be to achieve 100% method and statement coverage through your testing
- *Decision or branch coverage* is a measure of how many of the Boolean expressions of the program have been evaluated as both true and false in the testing
- *Condition coverage* reports the true or false outcome of *each* Boolean sub-expression of a compound predicate