Data Flow Criteria

Data Flow Testing -- Additional Criteria

Data flow testing

- **Static data flow**
  - Involves actual program execution (similar to CFG) but paths are identified based on **data flow testing criteria.**
- **Dynamic data**
  - Identify potential defects (**anomalies)**
  - Analyze source code
  - Do not execute code (dead code)

Data Flow Anomaly

There are different possible two letter combinations for 'd', 'k' and 'u'. Some re bugs, some are suspicious and some are okay

- dd: Probably harmless, but suspicious. Why define the object twice without an intervening usage?
- dk: Probably a bug,. Why define the object without using it?
- du: The normal case. The object is defined and then used.
- kd: Normal situation. An object is killed and then redefined.
- kk: Harmless but probably buggy. Did you want to be sure it was really killed?
- ku: A bug. The object does not exist.
- ud: Usually not a bug because the language permits reassignment at almost any time.
- uu: Normal situation

https://csestudyzone.blogspot.com/2015/06/data-flow-anomalies-in-data-flow-testing.html

Limitation of Static Anomaly Detection

- **Arrays:** reference to array is done at the level of the element in the array while defining or killing the array is done at the object level
- **Dead variable**: There no general solution to prove that the variable is dead. Only at a given point in time that is possible
- Dynamic subroutine or function names
- Concurrency
- Interrupts and incidents

Data Flow Anomaly Examples

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

Type: **dd**: Defined and then defined again

Why this happened?

Four interpretations of Example 1

The first statement is redundant.
The first statement has a fault -- the intended one might be: w = f1(x).
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).

Def and Use Definition

- Occurrences of variables
- **A *definition (def)*** is a location where a value for a variable is stored into memory (assignment, input, etc.).
    - X= 0;
  - **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;
    - Predicate use (p-use)
      - Example: if (y > 100) { …}

Data flow testing criteria

- Data flow testing criteria use the fact that values are carried from defs to uses. We call these *du-pairs* (they are also known as *definition-use*, *def-use*, and *du* associations in the testing literature).
- The idea of data flow criteria is to exercise du-pairs in variousways

Example :Def and Use

Given the following code statements :

1. X = y+3;
2. X= 10;
3. ….
4. …..
5. If (x < y) {

Write all definitions and uses for the three statements?

Def(1)= {x}   use(1) ={y}

Def(2) = {x}

use(5) = {x,y}

Example :Def and Use

1.	x= y+3	def(1)= {x}	use(1) = {y}
2.	y=10;	def(2) = {y}	
3.	…		
4.	…		
5.	X = x2+y	def(5)= {x}	use(5) = {x,y}
6.	If (x>y) {		use(6) = {x,y}

- For the path [1 2 3 4] the definition of x has not change → we call it definition clear path (dc)
- For the path [1 2 3 4 5 6] the definition of x changed (redefined) → we do not have a clear path

Example :Def and Use

1.
2. If (x<1) { use(2) = {x}
3. y = z+1 def(3)={y} use(2) = {z} → c-used
4. }

Def Occurrences

def may occur for variable x in the following situations:

1. x appears on the left side of an assignment statement
2. x is an actual parameter in a call site and its value is changed within the method
3. x is a formal parameter of a method (an implicit def when the method begins execution)
4. x is an input to the program

Use Occurrences

A use may occur for variable `x` in the following situations:

1. `x` appears on the right side of an assignment statement
2. `x` appears in a conditional test (note that such a test is always associated with at least two edges)
3. `x` is an actual parameter to a method
4. `x` is an output of the program
5. `x` is an output of a method in a `return` statement or returned as a parameter

def-clear

- An important concept when discussing data flow criteria is that a def of a variable may or may not reach a particular use. The most obvious reason that a def of a variable *v* at location *li* (a location could be a node or an edge) will not reach a use at location *l j* is because no path goes from *li* to *l j*
- A more subtle reason is that the variable's value may be changed by another def before it reaches the use

- No location between *li* and *l j* changes the value.

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.

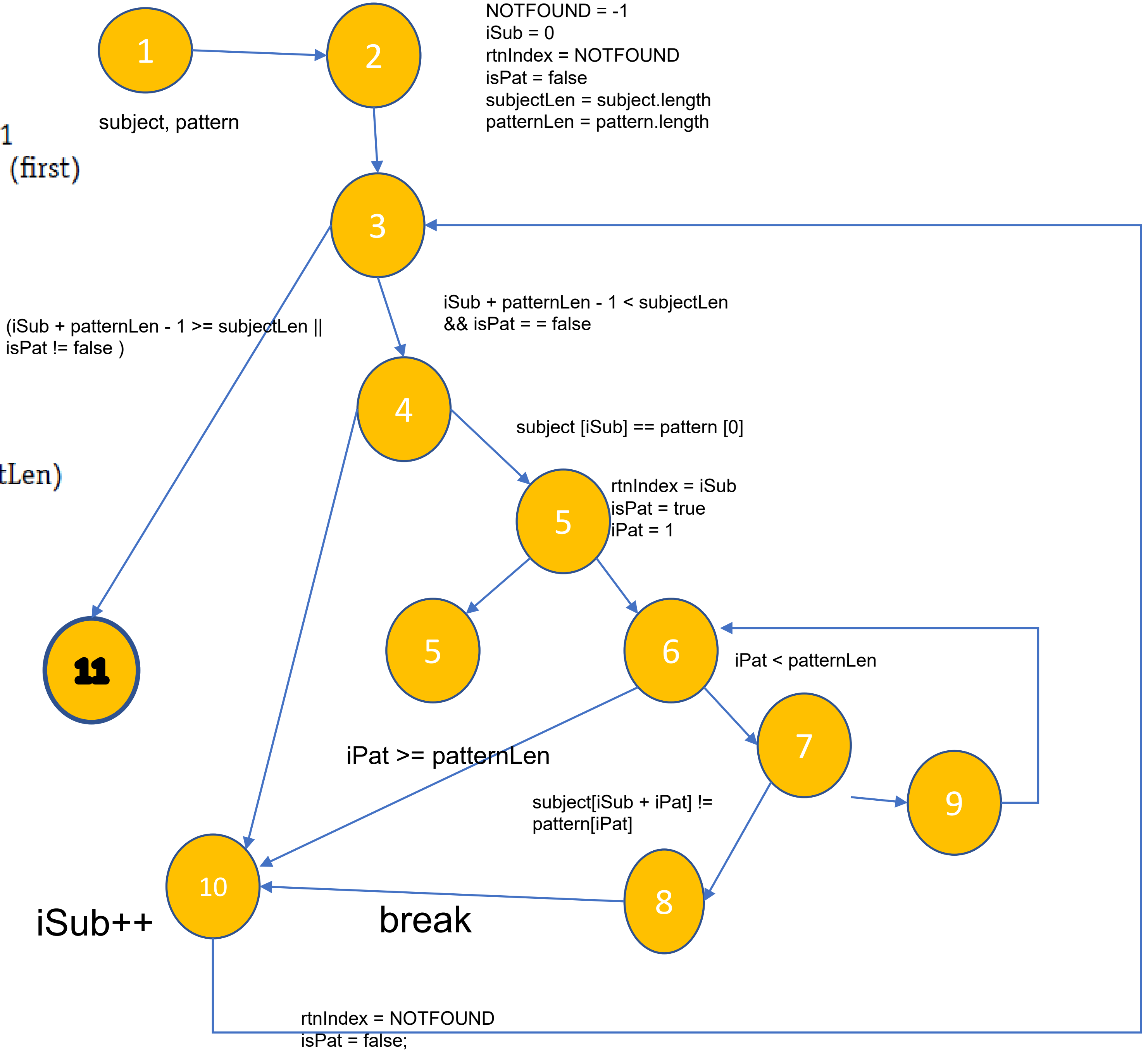
Motivation for Data Flow Graph

- 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.
- 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.

```
public int pat (char[] subject, char[] pattern)
{
// Post: if pattern is not a substring of subject, return -1
//       else return (zero-based) index where the pattern (first)
//       starts in subject

  final int NOTFOUND = -1;
  int  iSub = 0, rtnIndex = NOTFOUND;
  boolean isPat  = false;
  int subjectLen = subject.length;
  int patternLen = pattern.length;

  while (isPat == false && iSub + patternLen - 1 < subjectLen)
  {
   if (subject [iSub] == pattern [0])
   {
    rtnIndex = iSub; // Starting at zero
    isPat = true;
    for (int iPat = 1; iPat < patternLen; iPat ++)
    {
     if (subject[iSub + iPat] != pattern[iPat])
     {
      rtnIndex = NOTFOUND;
      isPat = false;
      break;  // out of for loop
     }
    }
   }
   iSub ++;
  }
  return (rtnIndex);
```



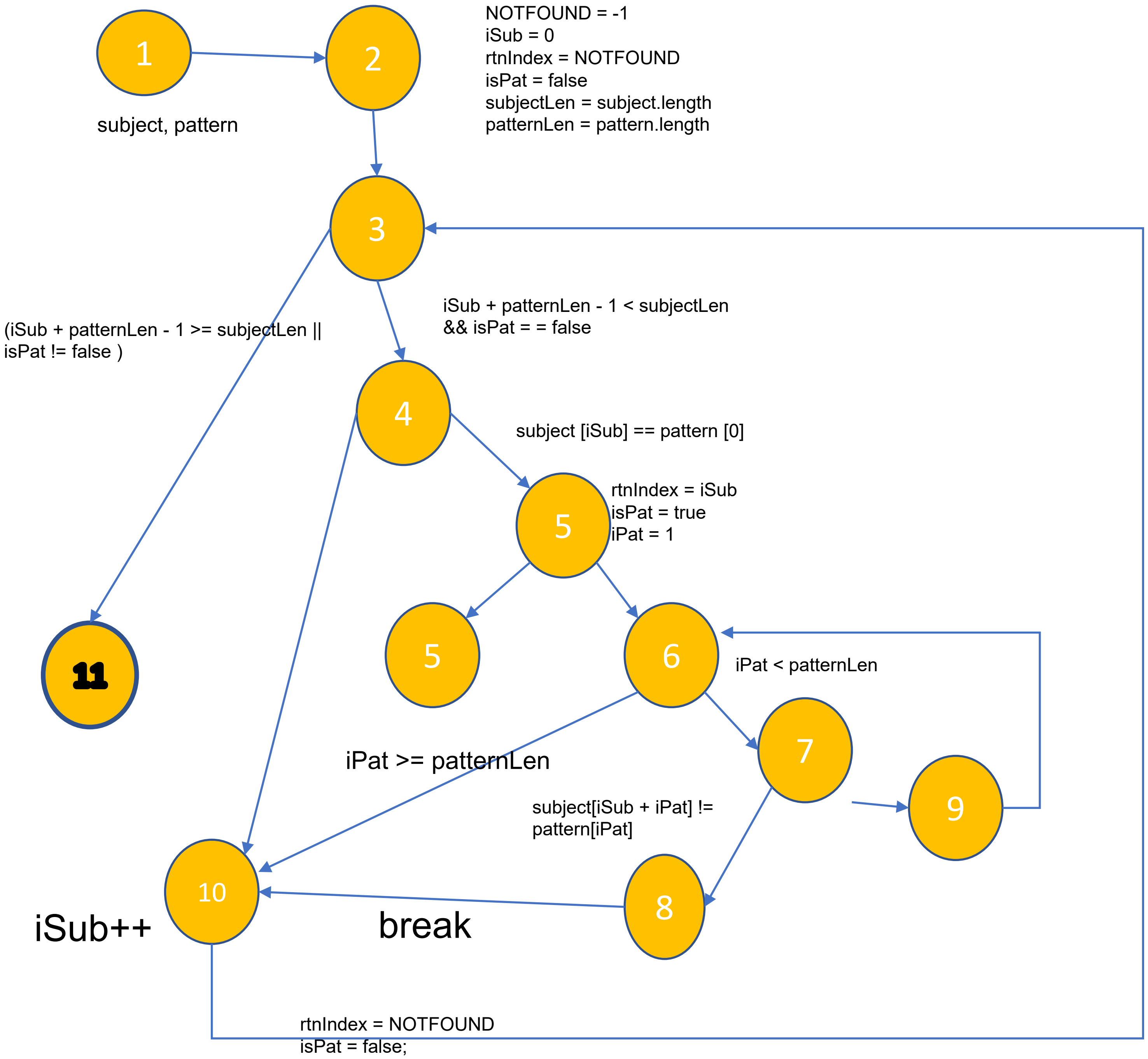
def(1) = { subject, pattern }

def(2) = {NOTFOUND, iSub, rtnIndex,isPat, subjectLen, atternLen }

use(2) = {subject , pattern }

use (3,11) = use (3,4) = { iSub, patternLen, subjectLen, isPat }

use(4,10) = use(4,5) = { subject, iSub, pattern }

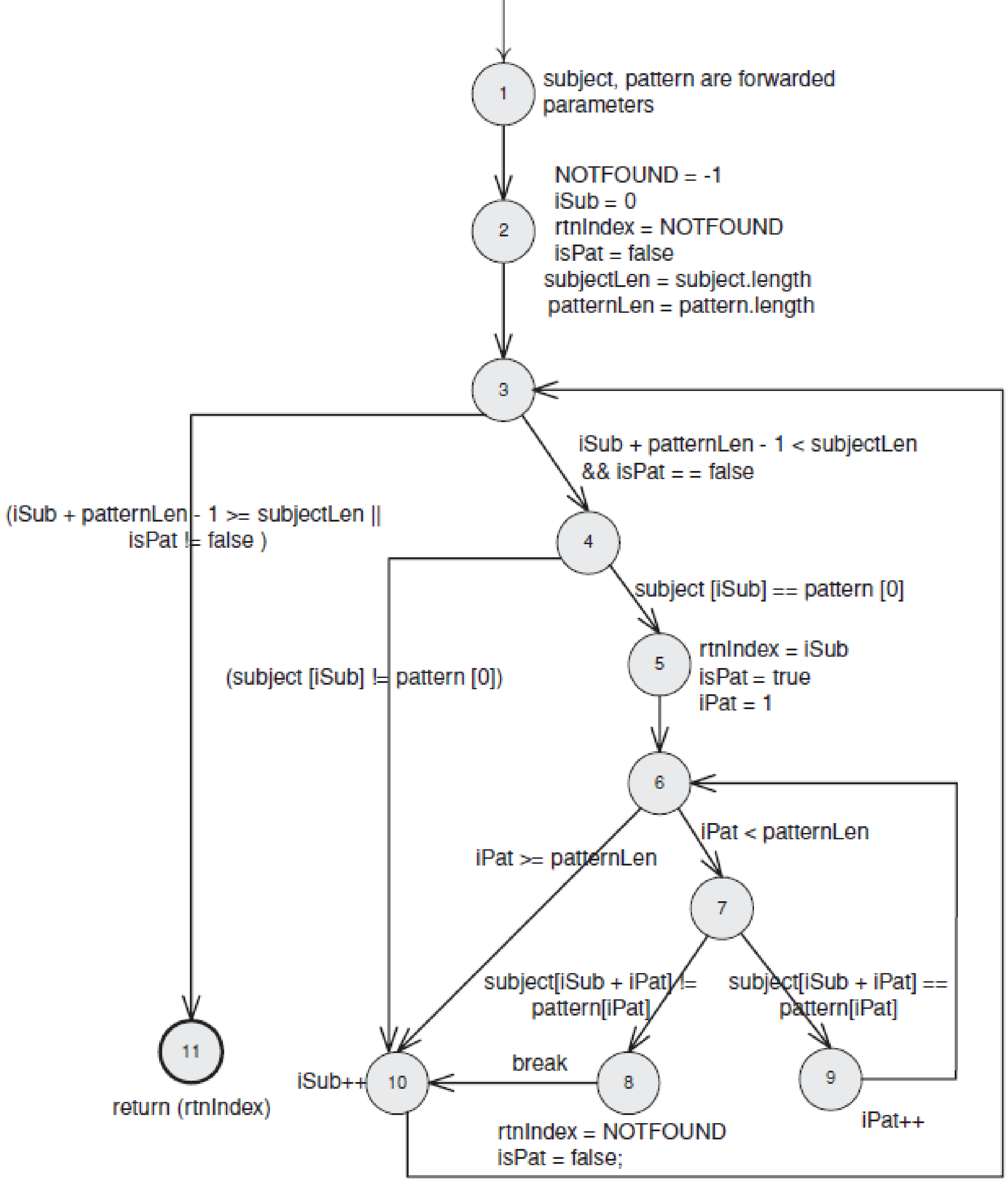


Data Flow Example

```java
public int pat (char[] subject, char[] pattern)
{
// Post: if pattern is not a substring of subject, return -1
//       else return (zero-based) index where the pattern (first)
//       starts in subject

  final int NOTFOUND = -1;
  int  iSub = 0, rtnIndex = NOTFOUND;
  boolean isPat  = false;
  int subjectLen = subject.length;
  int patternLen = pattern.length;

  while (isPat == false && iSub + patternLen - 1 < subjectLen)
  {
    if (subject [iSub] == pattern [0])
    {
      rtnIndex = iSub; // Starting at zero
      isPat = true;
      for (int iPat = 1; iPat < patternLen; iPat ++)
      {
        if (subject[iSub + iPat] != pattern[iPat])
        {
          rtnIndex = NOTFOUND;
          isPat = false;
          break;  // out of for loop
        }
      }
    }
    iSub ++;
  }
  return (rtnIndex);
}
```



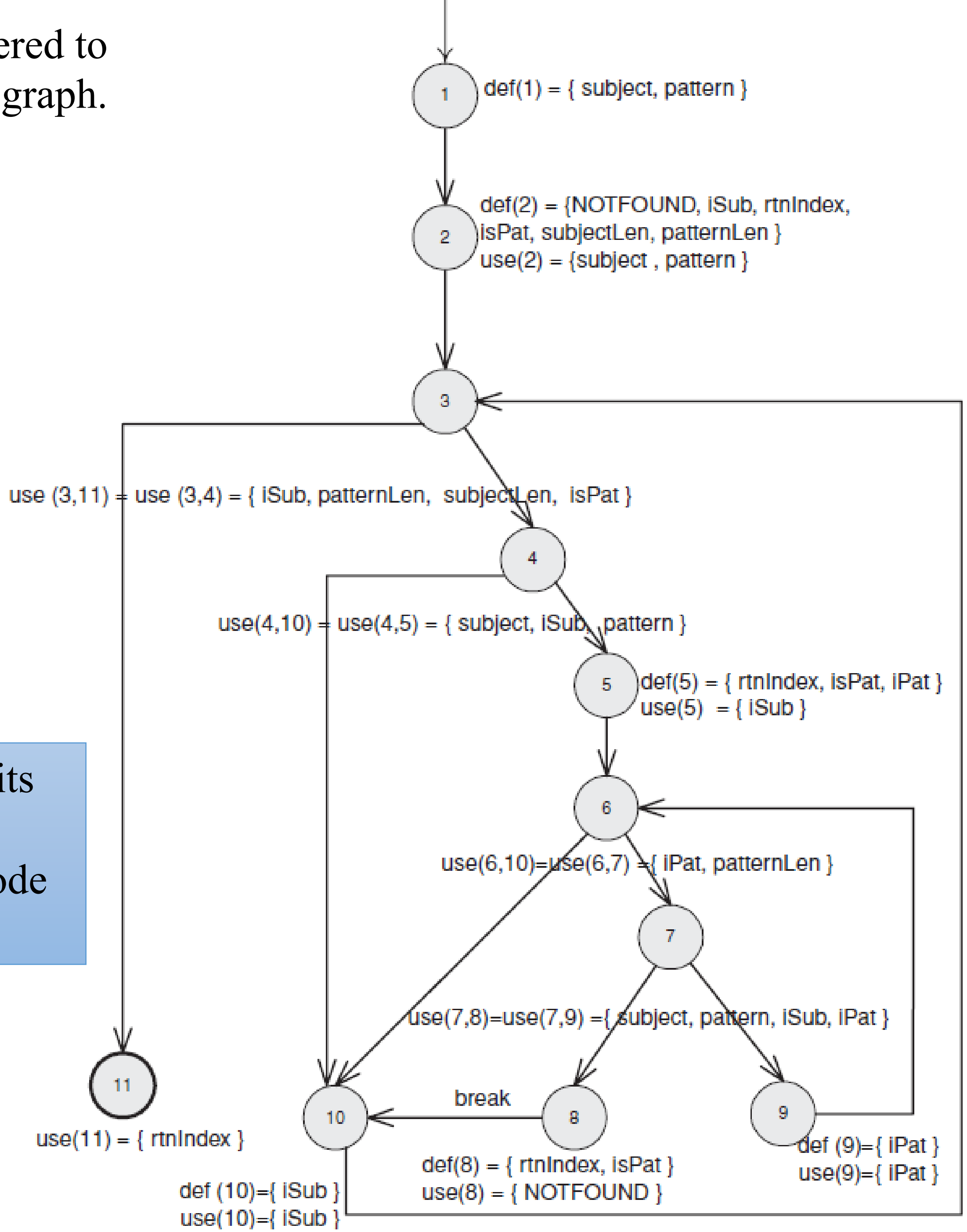
Note that the parameters (*subject* and *pattern*) are considered to be *explicitly defined* by the first node in the graph.

def set of node 1 is *def*(1) = {*subject, pattern*}.

if subject[*i Sub*] == *pattern*[0]) result in uses of each of the associated variables for both edges in the decision. That is, *use*(4, 10) ≡ *use*(4,5) ≡ {*subject*, *i Sub*, *pattern*}.

The parameter *subject* is used at node 2 (with a reference to its *length* attribute) and at edges (4, 5), (4, 10), (7, 8), and (7, 9), thus du-paths exist from node 1 to node 2 and from node 1 to each of those four edges.

iPat ++, which is equivalent to *iPat* = *iPat*+1.



Du-path

- A *du-path* with respect to a variable *v* is a simple path that is def-clear with respect to *v* from a node *ni* for which *v* is in *def*(*ni*) to a node *nj* for which *v* is in *use*(*nj*).
- Note that a du-path is always associated with a specific variable *v*, a du-path always has to be simple, and there may be intervening uses on the path
- **The test criteria for data flow will be defined as sets of du-paths**. This makes the criteria quite simple, but first we need to categorize the du-paths into several *groups*

Data Flow Coverage Criteria

- Data flow coverage criteria will be defined as sets of du-paths
- Du-paths will check for definitions of variable reaching their uses

1. Grouping du-paths as per definitions
2. Grouping du-paths as per definitions and uses

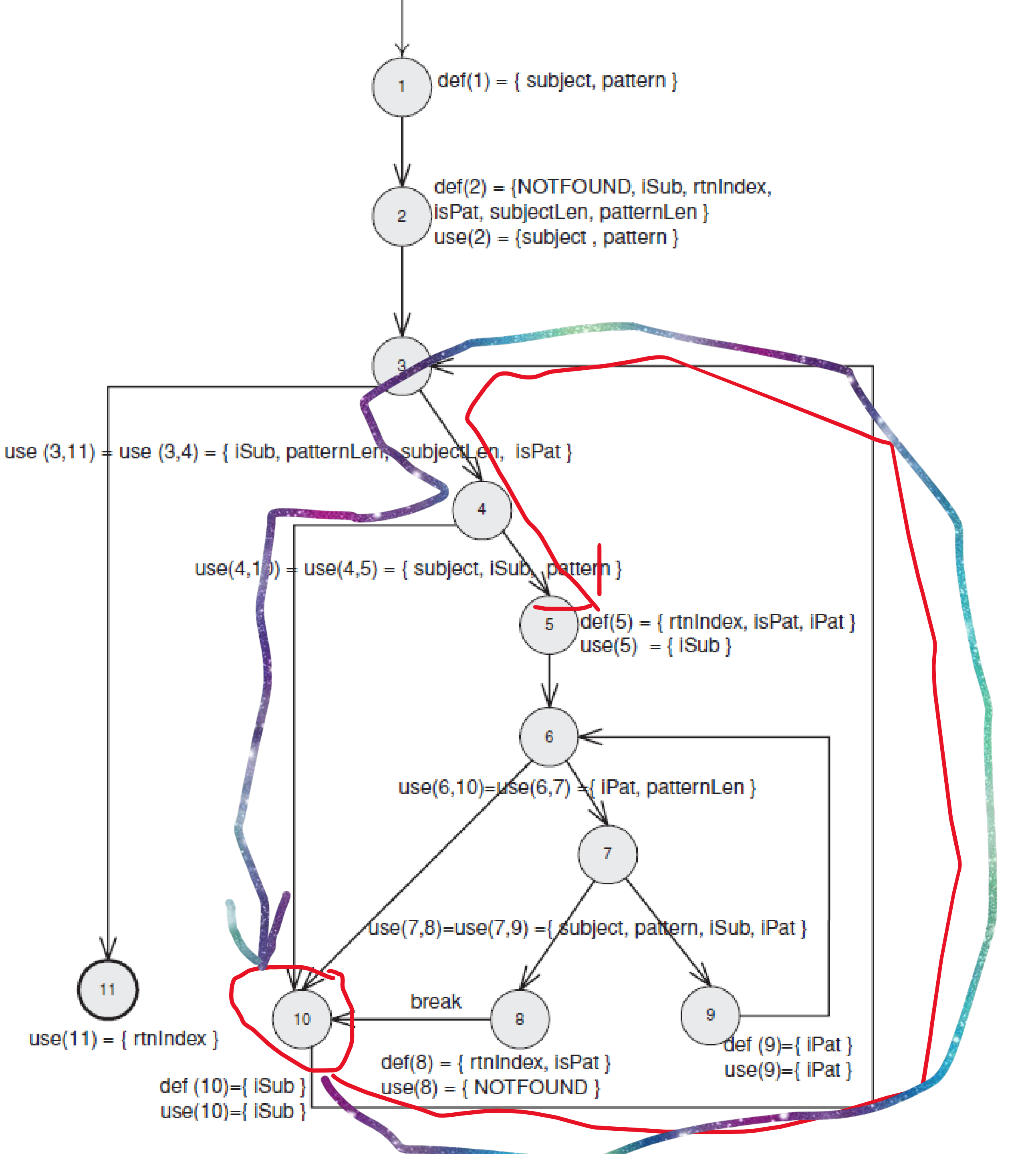
Grouping du-paths as per definitions

- Consider all du-paths with respect to a given variable in a given node
- The def-path set du(ni,v) is the set of du-paths with respect to variable v that start at node ni
- We do not group du-paths by uses

The first grouping of du-paths is according to definitions. Specifically, consider all of the du-paths with respect to a given variable defined in a given node

Let the *def-path* set *du*(*ni*, *v*) be the set of du-paths with respect to variable *v* that start at node *ni*

- The def-path set for the use of *isub* at node 10 is:
- *du*(10, *i Sub*) = {[10, 3, 4], [10, 3, 4, 5], [10, 3, 4, 5, 6, 7, 8], [10, 3, 4, 5, 6, 7, 9], [10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10], [10, 3, 11]}
- This def-path set can be broken up into the following def-pair sets:
- *du*(10, 4, *iSub*) = *is*{[10, 3, 4]}
- *du*(10, 5, *iSub*) = {[10, 3, 4, 5]}
- *du*(10, 8, *iSub*) = {[10, 3, 4, 5, 6, 7, 8]}
- *du*(10, 9, *iSub*) = {[10, 3, 4, 5, 6, 7, 9]}
- *du*(10, 10, *iSub*) = {[10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10]}
- *du*(10, 11, *iSub*) = {[10, 3, 11]}



Defs and uses at each node in the CFG for TestPat

node	def	use
1	{subject, pattern}	
2	{NOTFOUND, isPat, iSub, rtnIndex, subjectLen, patternLen}	{subject, pattern}
3		
4		
5	{rtnIndex, isPat, iPat}	{iSub}
6		
7		
8	{rtnIndex, isPat}	{NOTFOUND}
9	{iPat}	{iPat}
10	{iSub}	{iSub}
11		{rtnIndex}

du-paths for each variable in TestPat

edge	use
(1, 2)	
(2, 3)	
(3, 4)	{iSub, patternLen, subjectLen, isPat}
(3, 11)	{iSub, patternLen, subjectLen, isPat}
(4, 5)	{subject, iSub, pattern}
(4, 10)	{subject, iSub, pattern}
(5, 6)	
(6, 7)	{iPat, patternLen}
(6, 10)	{iPat, patternLen}
(7, 8)	{subject, iSub, iPat, pattern}
(7, 9)	{subject, iSub, iPat, pattern}
(8, 10)	
(9, 6)	
(10, 3)	

Grouping du-paths as per definitions and uses

- A def-pair set du(ni,nj,v) is the set of du-paths with respect to variable v that start at node ni and end at node nj
- A def-per set collect together all the simple ways to get from a given definition to a given use
- A def-pair for a def at node ni is the union of all the def-path set for the def. du(ni,v)= Unj du(ni,nj,v)

Definitions

CRITERION **2.9 All-Defs Coverage (ADC):** *For each def-path set $S = du(n, v)$, TR contains at least one path d in S.*

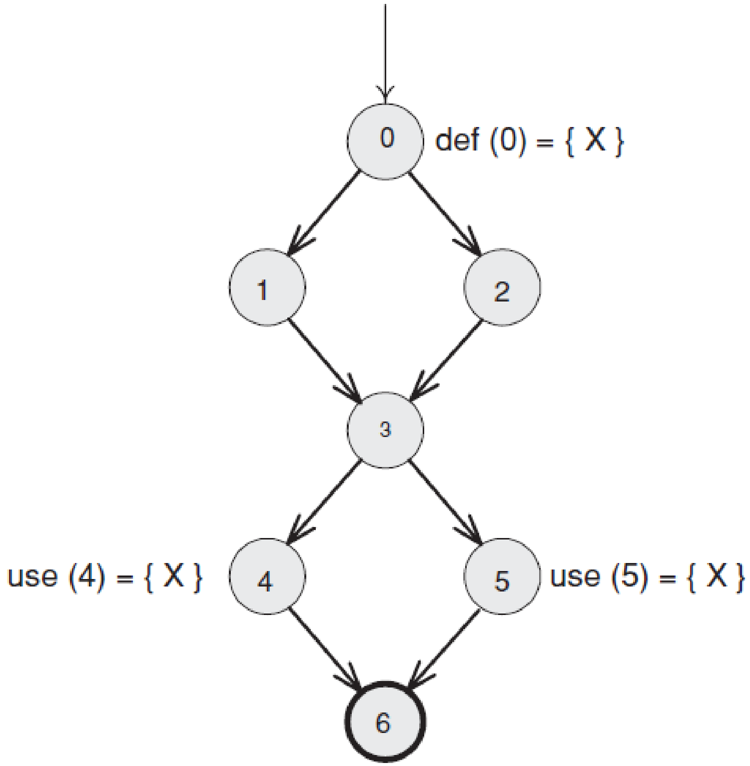
CRITERION **2.10 All-Uses Coverage (AUC):** *For each def-pair set $S = du(n_i, n_j, v)$, TR contains at least one path d in S.*

CRITERION **2.11 All-du-Paths Coverage (ADUPC):** *For each def-pair set $S = du(n_i, n_j, v)$, TR contains every path d in S.*

All-defs
0-1-3-4

All-uses
0-1-3-4
0-1-3-5

All-du-paths
0-1-3-4
0-1-3-5
0-2-3-4
0-2-3-5



ADC: [0 1 3 4] or [0 2 3 5]

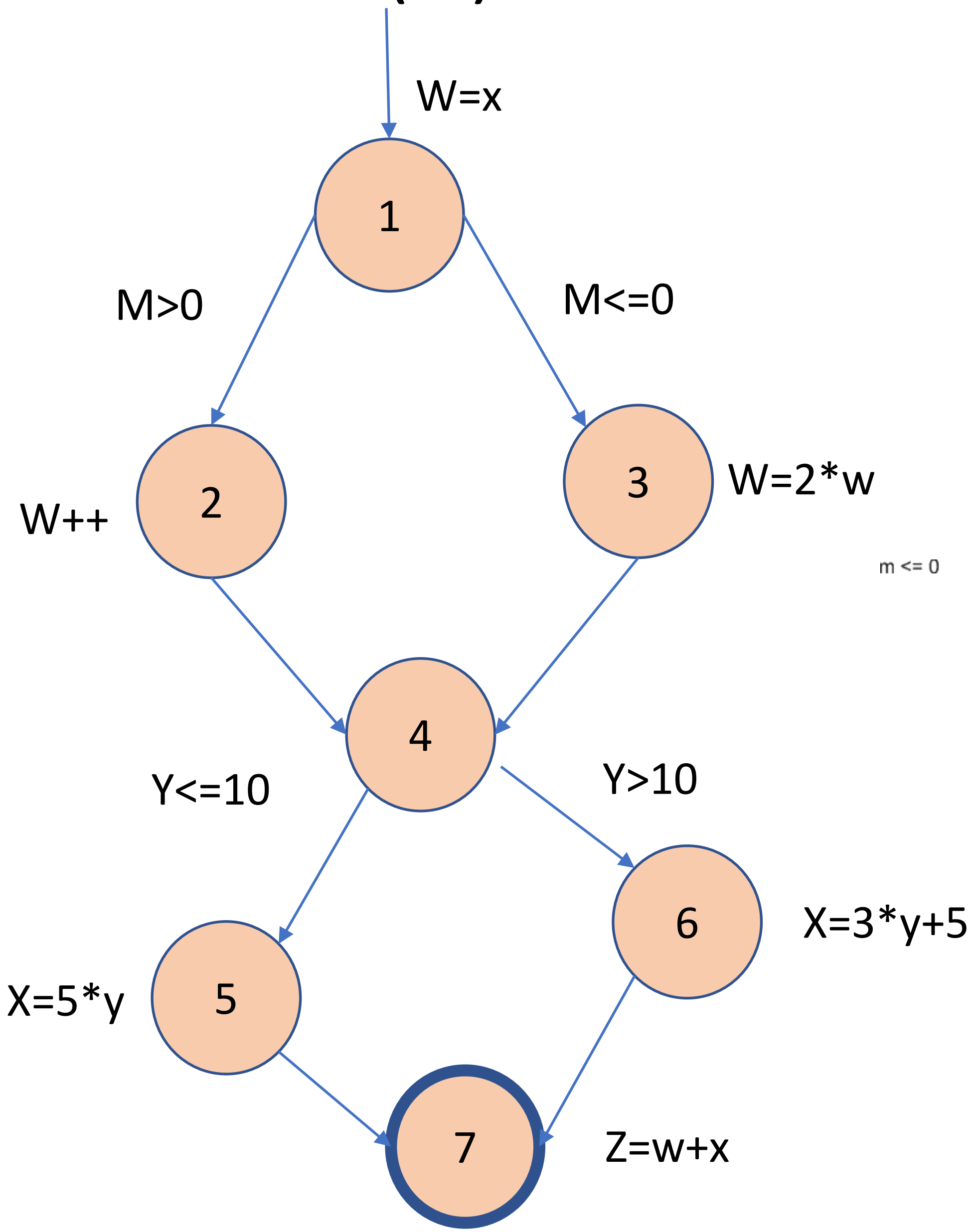
Exercise

Use the following program fragment for questions a-e below.

- (a) Draw a control ow graph for this program fragment. Use the node numbers given above.
- (b) Which nodes have defs for variable w?
- (c) Which nodes have uses for variable w?
- (d) Are there any du-paths with respect to variable w from node 1 to node 7? If not, explain why not. If any exist, show one.
- (e) Enumerate all of the du-paths for variables w and x.

```
w = x;              // node 1
if (m > 0)
{
   w++;             // node 2
}
else
{
   w=2*w;           // node 3
}
// node 4 (no executable statement)
if (y <= 10)
{
   x = 5*y;         // node 5
}
else
{
   x = 3*y+5;       // node 6
}
z = w + x;          // node 7
```

Solution for (a): Draw a control ow graph

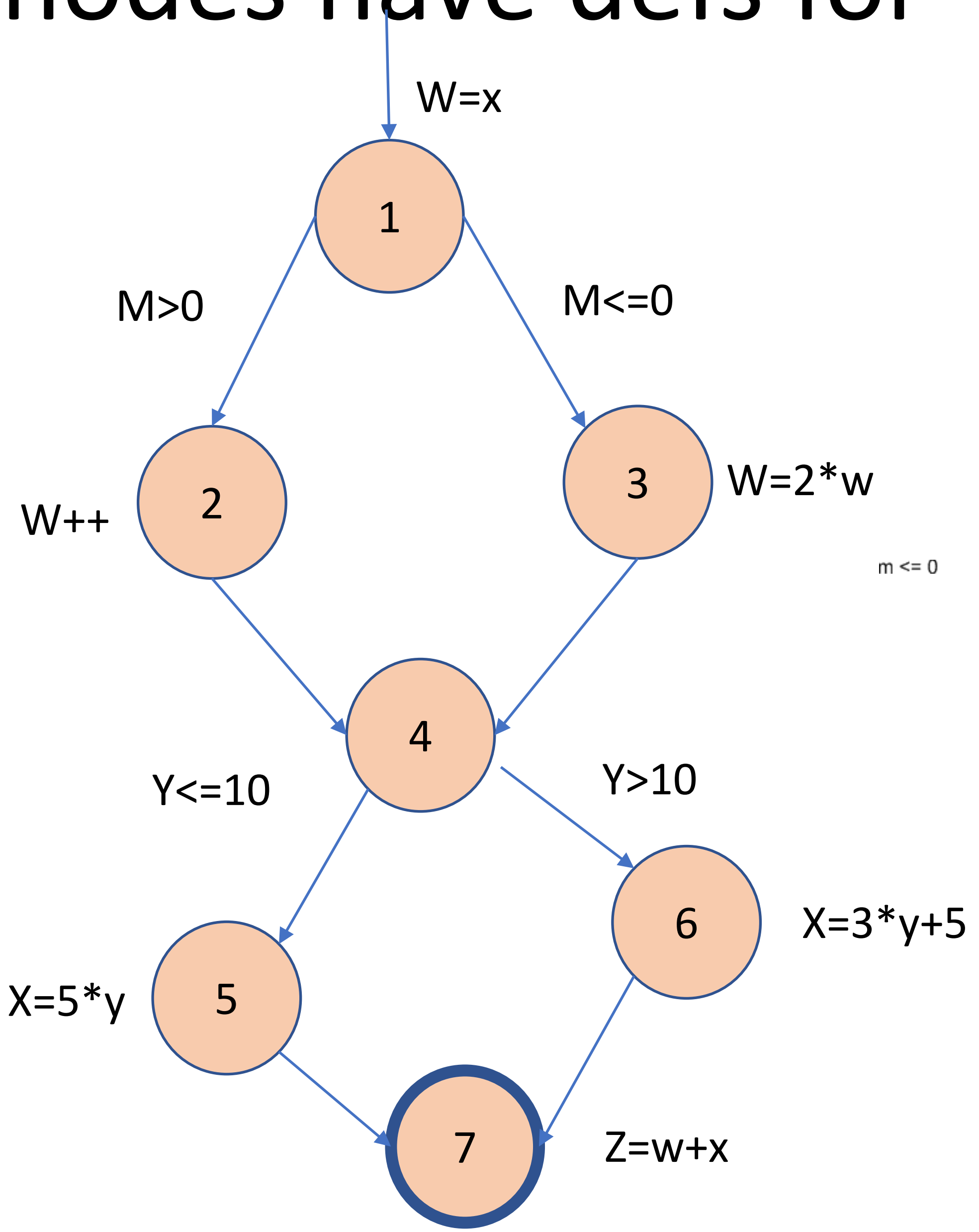



```
w = x;              // node 1
if (m > 0)
{
   w++;             // node 2
}
else
{
   w=2*w;           // node 3
}
// node 4 (no executable statement)
if (y <= 10)
{
   x = 5*y;         // node 5
}
else
{
   x = 3*y+5;       // node 6
}
z = w + x;          // node 7
```

Solution for: **(b) Which nodes have defs for variable w?**

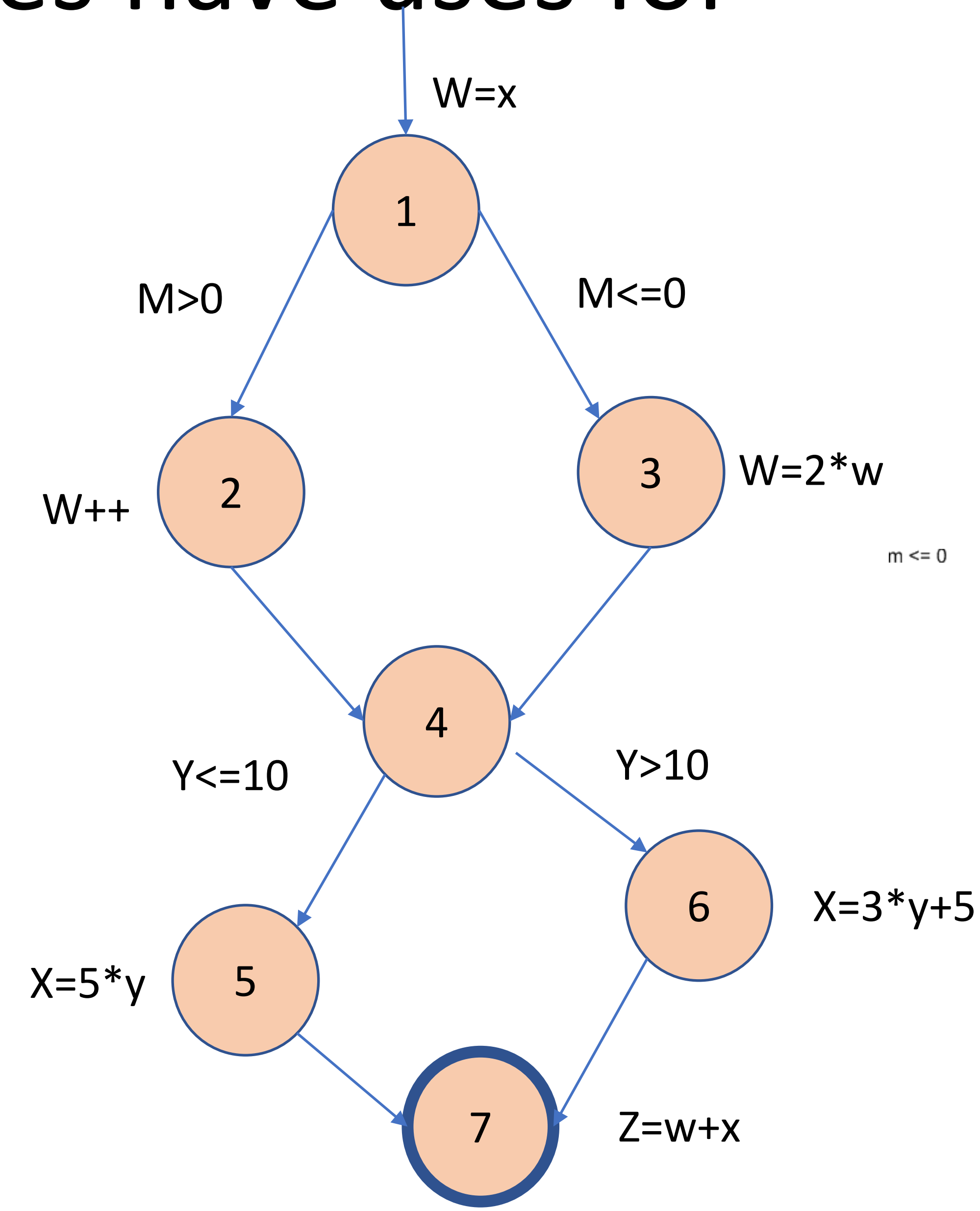
- Def(1): {w}
- Def(3) :{w}
- Def(2) : {w}

So nodes 1,2,3 have def for variable w



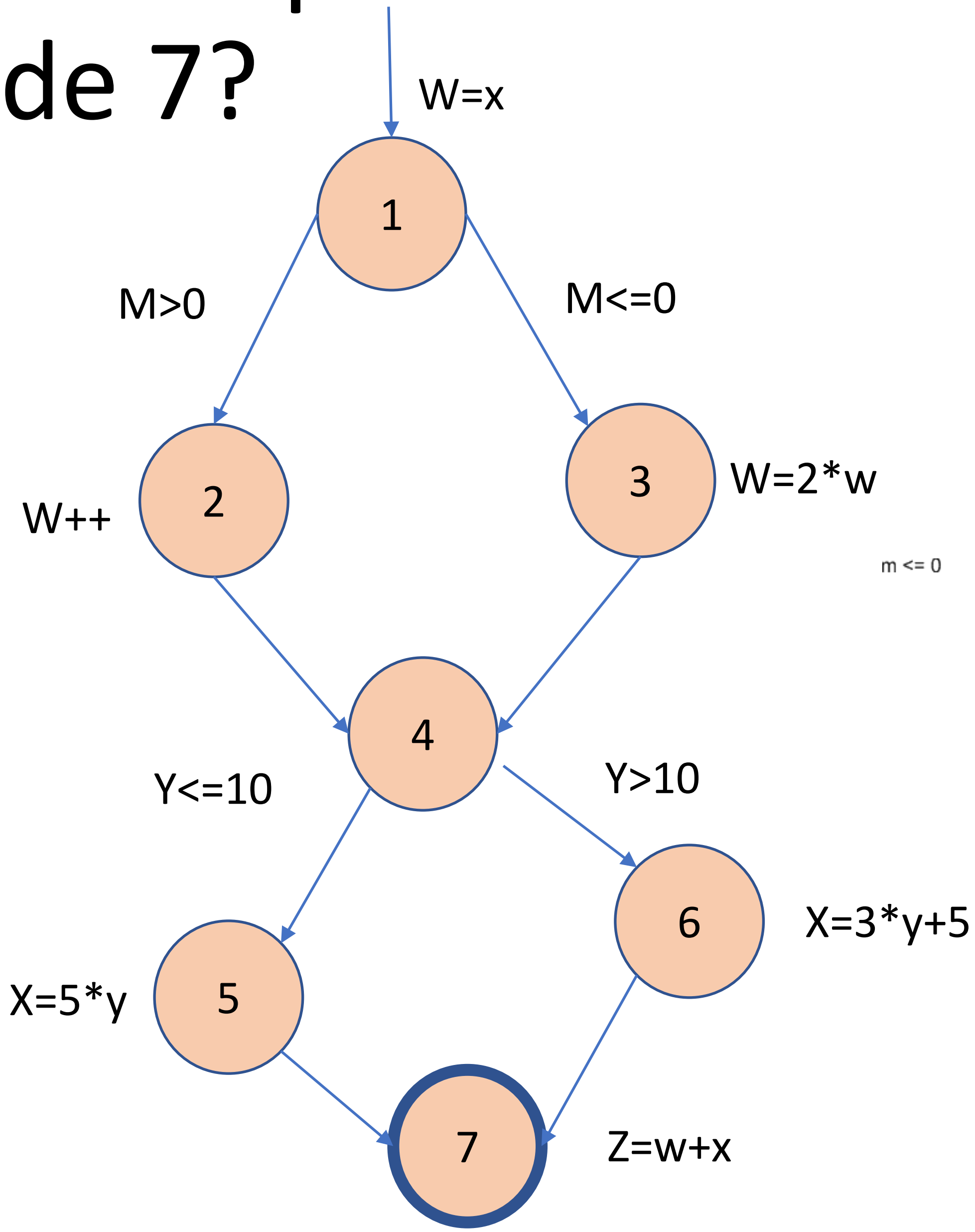
Solution for:(c) Which nodes have uses for variable w?

- Nodes 2,3,7 have uses for variable w



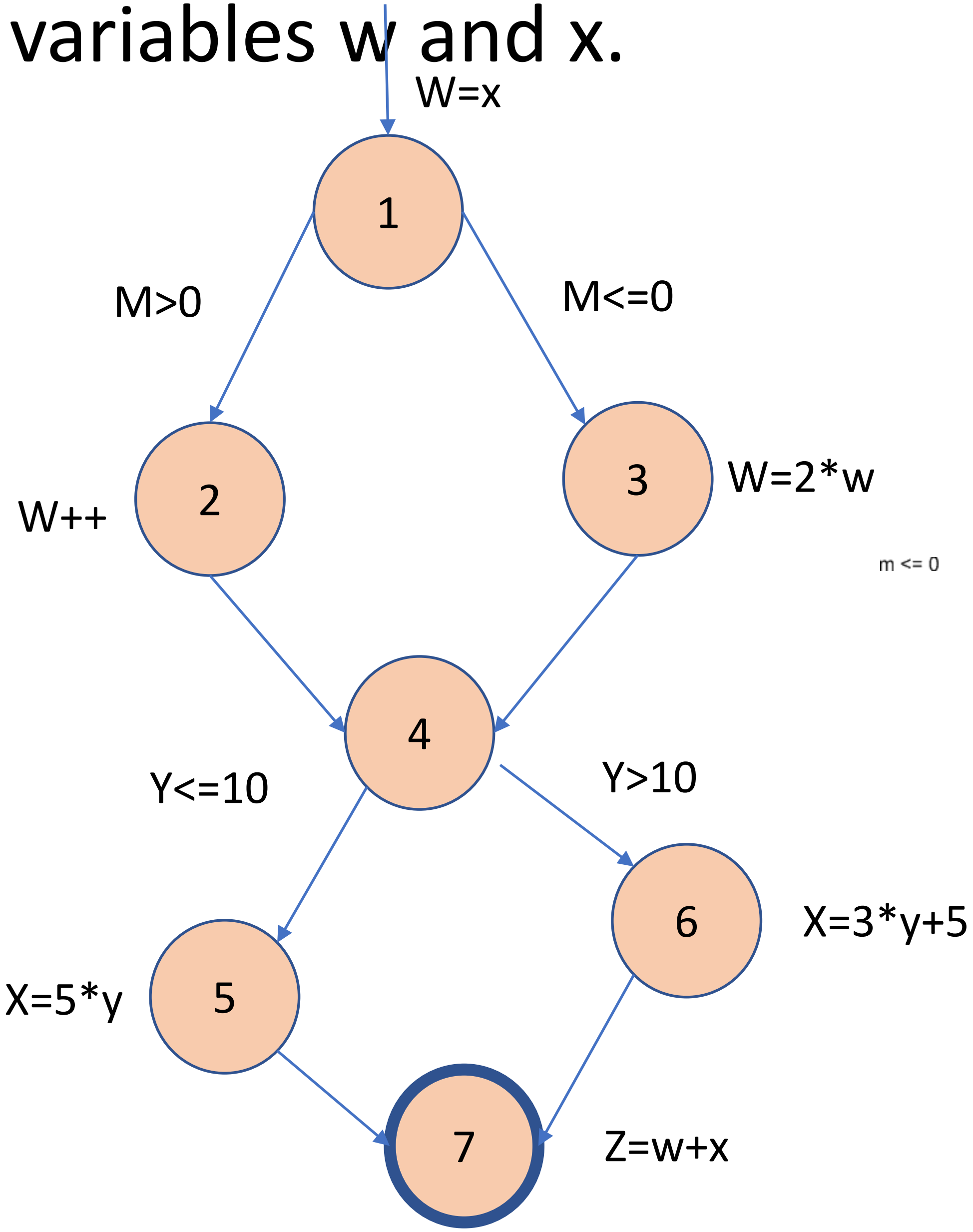
(d) Are there any du-paths with respect to variable w from node 1 to node 7?

- Node 1 have def
- Node 7 has use but in node 2 we have redefine so there is no clear path from 1 to 7



(e) Enumerate all of the du-paths for variables w and x.

- The du-path should be clear
- For variable w
  - [1 2] [1 3]
  - [2 4 5 7] [2 4 6 7]
  - [3 4 5 7] [3 4 6 7]
- For variable x
  - [5 7] [6 7]



Test Coverage Metrics -Recap

There are many different ways to measure test coverage, but some common metrics include:

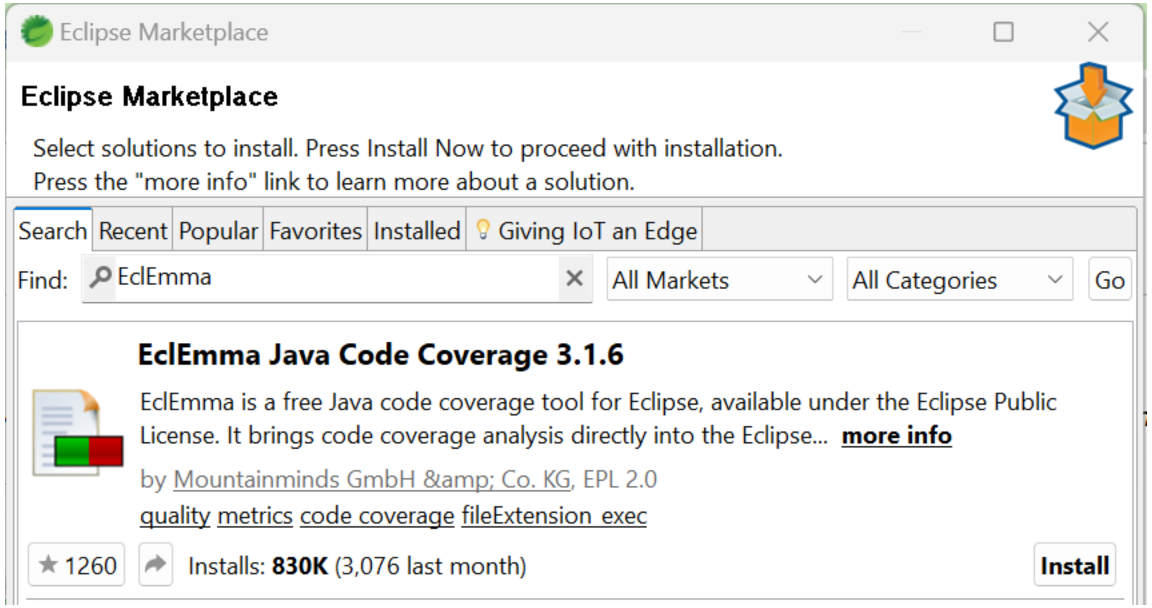
- **Lines of code covered:** This metric simply measures the number of lines of code that are covered by tests. This is a good starting point, but it doesn't give the whole picture, since some lines of code are more important than others.
- **Blocks covered:** This metric measures the number of blocks of code (e.g., if-statements, for-loops, etc.) that are covered by tests. This is a more granular metric than lines of code, but it still doesn't give the whole picture.
- **Functions/methods covered:** This metric measures the number of functions or methods that are covered by tests. This is a more granular metric than blocks, but it still doesn't give the whole picture.
- **Statements covered:** This metric measures the number of statements that are covered by tests. This is the most granular metric, but it still doesn't give the whole picture.
- **Conditions covered:** This metric measures the number of conditions (e.g., if-statements, boolean expressions, etc.) that are covered by tests.

Code Coverage Tools

- JaCoCo

  https://www.eclemma.org/jacoco/

  Download the plugin into your STS or Eclipse . Go to help→narketplace then search for *EclEmma*



Example

```java
class CurrencyNameTest {

    @Test
    void test() {
        BorwaserTest coverage = new BorwaserTest();

        String currency1= coverage.getCurrencyName("USD");
        assertEquals("American Dollar",currency1);

    }

}
```

```java
public String getCurrencyName(String code) {

        switch(code) {
        case "USD":
            return "American Dollar";
        case "JOD" :
            return "Jordanian Dinar";
        default:
            return "Wrong Currency";
        }

}
```

Testing Results

Coverage

Element	Coverage	Covered Instructions	Missed Instr...	Total Instructions
selenium	86.2 %	25	4	29
src	86.2 %	25	4	29
chrome	86.2 %	25	4	29
CaseCoverageTest.java	71.4 %	10	4	14
CaseCoverageTest	71.4 %	10	4	14
getCurrencyName(String)	63.6 %	7	4	11
CurrencyNameTest.java	100.0 %	15	0	15

Refine the test cases to achieve 100% coverage

Reading Material

- **Control Flow Analysis for Java Methods**
  - https://www.jacoco.org/jacoco/trunk/doc/flow.html

- This article is part of the course material