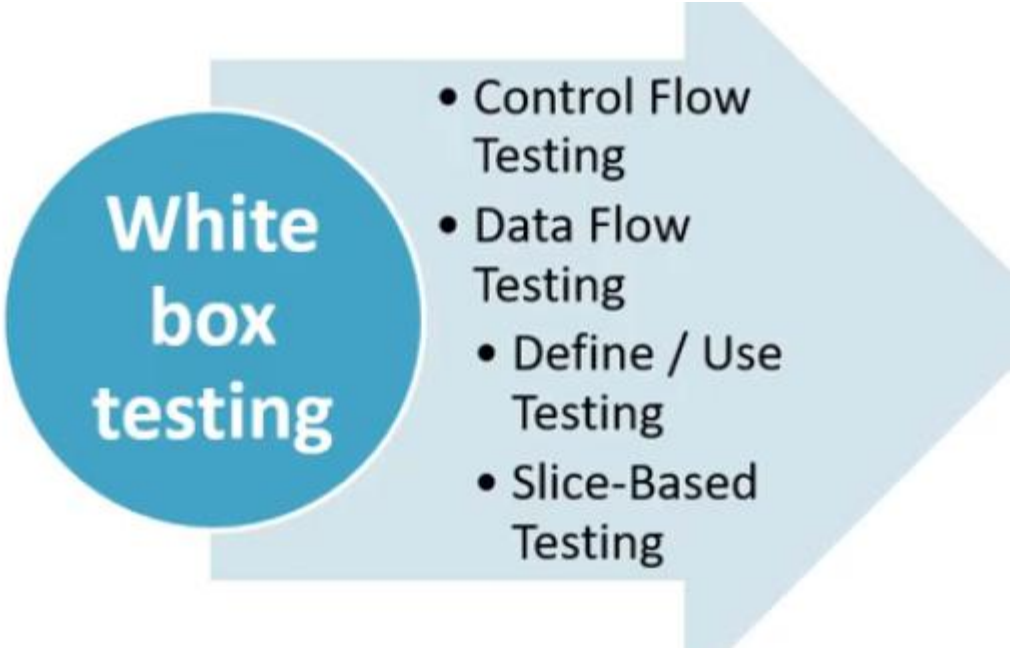
# White-Box Testing

Structural Testing , Coverage Testing

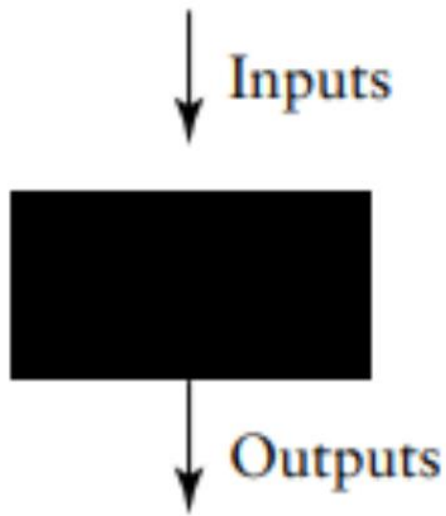
all (100%) of the statements in the module are executed at least once



# Testing: Facts and Myths

- Fact: Testing can be used to find errors in software, cannot be used to show that a software is correct.
- Fact: Testing cannot be replaced by software reviews, inspections, quality audits etc.
- Fact: Testing cannot be fully automated, needs human intervention.
- Myth: It is wrong to say that "My code is correct and doesn't need to be tested".

***The two basic testing strategies.***

| Test Strategy | Tester's View | Knowledge Sources | Methods |
|---|---|---|---|
| Black box |   | Requirements document<br>Specifications<br>Domain knowledge<br>Defect analysis data | Equivalence class partitioning<br>Boundary value analysis<br>State transition testing<br>Cause and effect graphing<br>Error guessing |
| White box |  | High-level design<br>Detailed design<br>Control flow graphs<br>Cyclomatic complexity | Statement testing<br>Branch testing<br>Path testing<br>Data flow testing<br>Mutation testing<br>Loop testing |

The goal for white box testing is to ensure that the internal components of a program are working properly. A common focus is on structural elements such as statements and branches.

## Test Adequacy Criteria

A program is said to be adequately tested with respect to a given criterion if all of the target structural elements have been exercised according to the selected criterion.

(i) helping testers to select properties of a program to focus on during test;
(ii) helping testers to select a test data set for a program based on the selected properties;
(iii) supporting testers with the development of quantitative objectives for testing;
(iv) indicating to testers whether or not testing can be stopped for that program.

**A test data set is statement, or branch, adequate if a test set *T* for program *P* causes all the statements, or branches, to be executed respectively.**

(i) exercising program paths from entry to exit,

(ii) execution of specific path segments derived from data flow combinations such as definitions and uses of variables

the planned degree of coverage may be less than 100% possibly due to the following:

- The nature of the unit

  —Some statements/branches may not be reachable.

  —The unit may be simple, and not mission, or safety, critical, and so complete coverage is thought to be unnecessary.

- The lack of resources

  —The time set aside for testing is not adequate to achieve 100% coverage.

  —There are not enough trained testers to achieve complete coverage for all of the units.

  —There is a lack of tools to support complete coverage.

- Other project-related issues such as timing, scheduling, and marketing constraints

## Coverage and Control Flow Graphs

logical elements

flow of control in a unit of code.

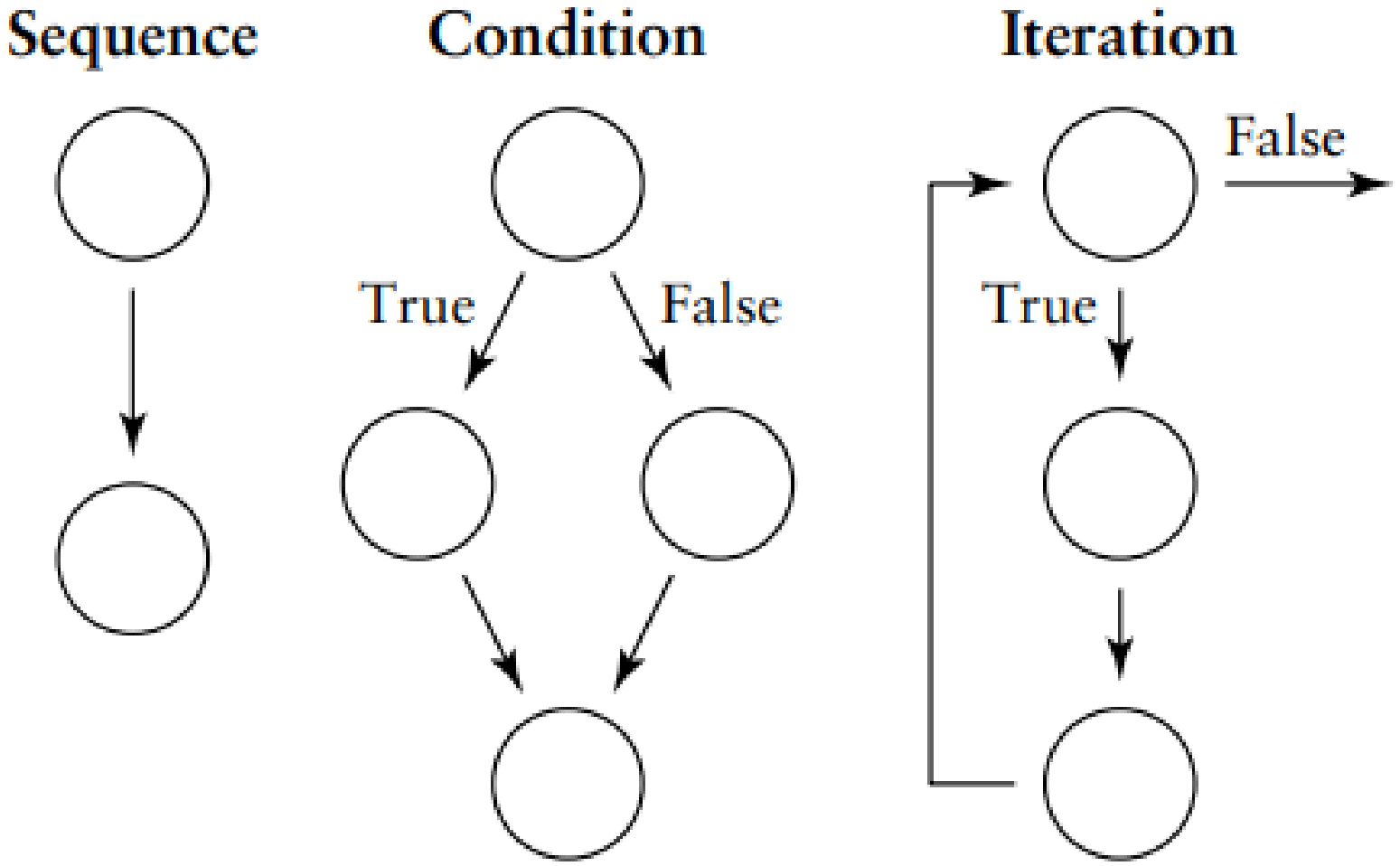
(i) program statements;

(ii) decisions/branches (these influence the program flow of con

(iii) conditions (expressions that evaluate to true/false, and do no
any other true/false-valued expressions);

(iv) combinations of decisions and conditions;

(v) paths (node sequences in flow graphs).

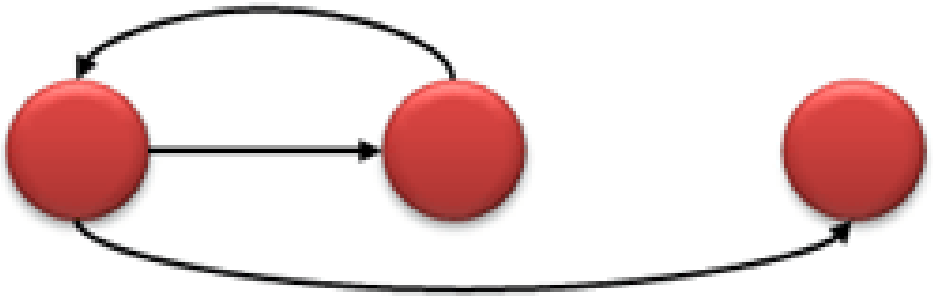



**Each decision element in the code (if-then, case, loop) executes with all possible outcomes at least once**

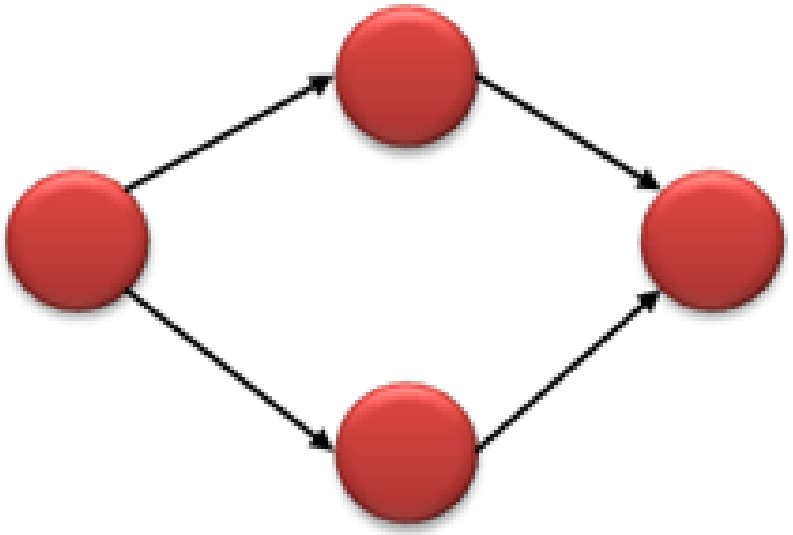
*Representation of program primes.*

Sequence

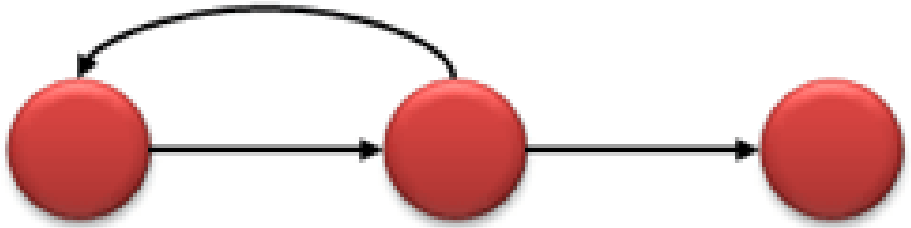
While



If-then-else



Until



Calculating cyclomatic complexity and test cases design are

Cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the Control Flow Graph of the program

- Construction of graph with nodes and edges from code.
- Identification of independent paths.
- Cyclomatic Complexity Calculation
- Design of Test Cases

$$M = E - N + 2P$$

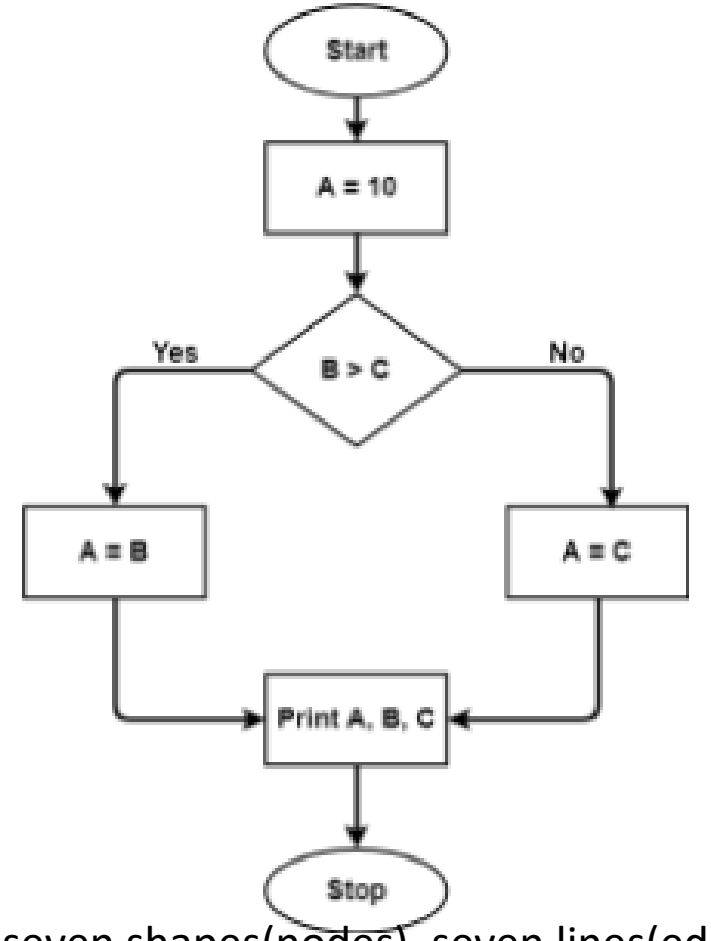
*where,*

*E = the number of edges in the control flow graph*

*N = the number of nodes in the control flow graph*

*P = the number of connected components*

```
A = 10
    IF B > C THEN
        A = B
    ELSE
        A = C
    ENDIF
Print A
Print B
Print C
```




The graph shows seven shapes(nodes), seven lines(edges), hence cyclomatic complexity is 7-7+2 = 2

# How to Calculate Cyclomatic Complexity

**Mathematical representation:**

Mathematically, it is set of independent paths through the graph diagram. The Code complexity of the program can be defined using the formula –

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

**Where,**

E – Number of edges

N – Number of Nodes

```
V (G) = P + 1
```

# Steps to be followed:

The following steps should be followed for computing Cyclomatic complexity and test cases design.

**Step 1** – Construction of graph with nodes and edges from the code

**Step 2** – Identification of independent paths

**Step 3** – Cyclomatic Complexity Calculation

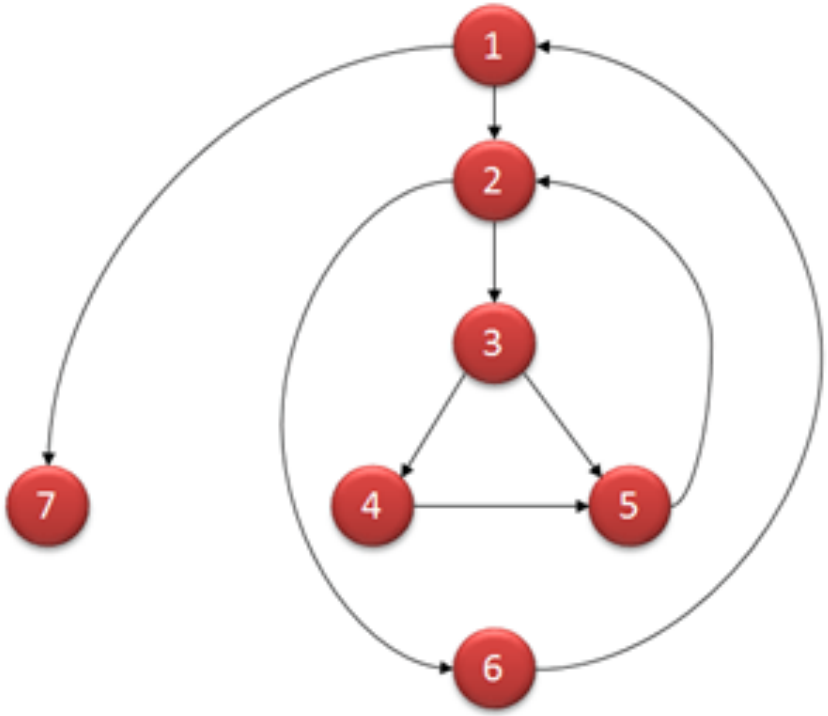
**Step 4** – Design of Test Cases

Once the basic set is formed, TEST CASES should be written to execute all the paths.

This metric is useful because of properties of Cyclomatic complexity (M) –

1. M can be number of test cases to achieve branch coverage (Upper Bound)
2. M can be number of paths through the graphs. (Lower Bound)

```
i = 0;
n=4; //N-Number of nodes present i

while (i<n-1) do
j = i + 1;

while (j<n) do

if A[i]<A[j] then
swap(A[i], A[j]);

end do;
j=j+1;

end do;
```




**Computing mathematically,**

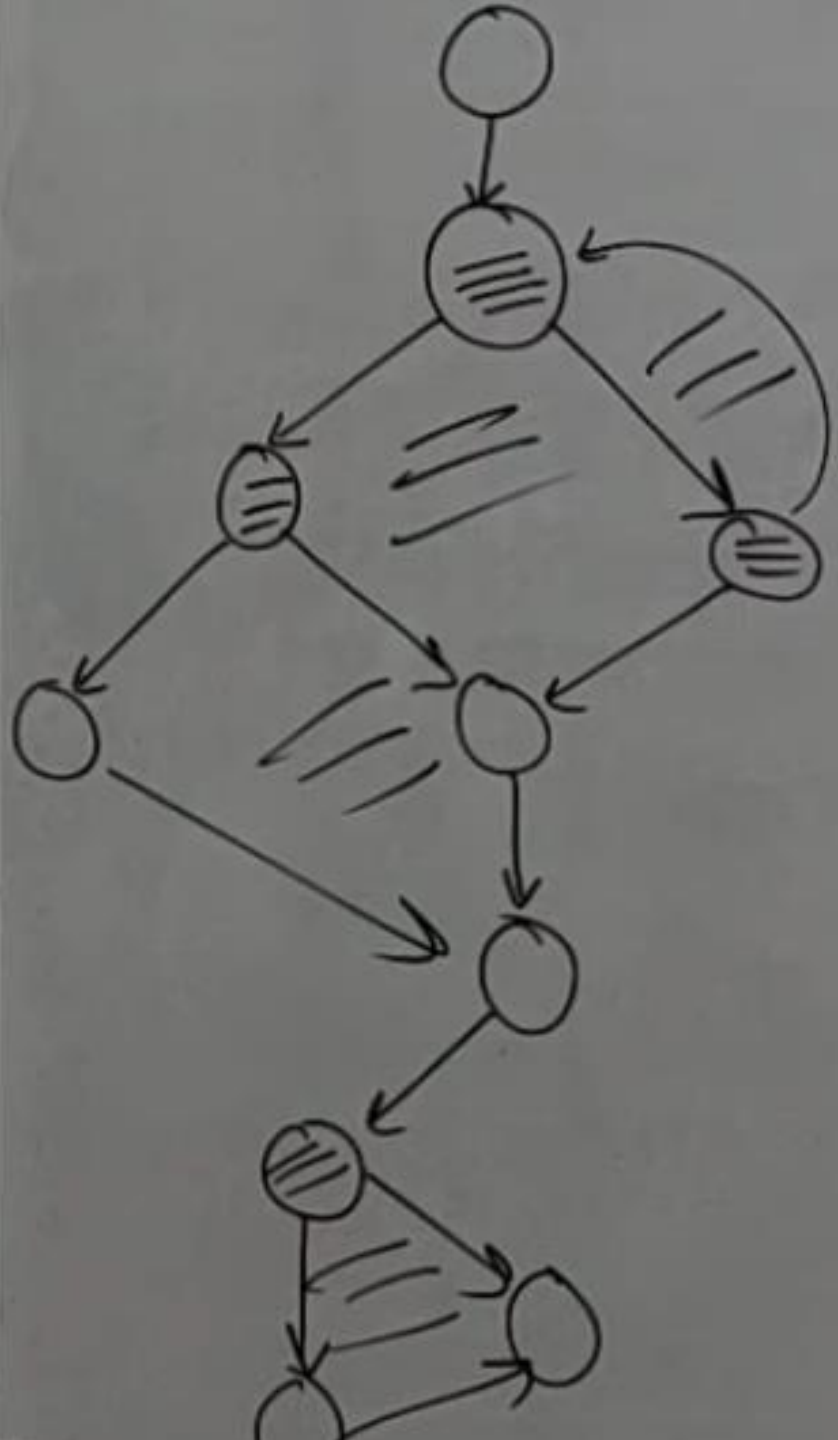
- V(G) = 9 – 7 + 2 = 4
- V(G) = 3 + 1 = 4 (Condition nodes are 1,2 and 3 nodes)
- Basis Set – A set of possible execution path of a program
- 1, 7
- 1, 2, 6, 1, 7
- 1, 2, 3, 4, 5, 2, 6, 1, 7
- 1, 2, 3, 5, 2, 6, 1, 7

number test cases, will be equivalent to the cyclomatic complexity of the program.

## Properties of Cyclomatic complexity:

1. V (G) is the maximum number of independent paths in the gr
2. V (G) >=1
3. G will have one path if V (G) = 1
4. Minimize complexity to 10

| Cyclomatic Complexity | Meaning |
| --- | --- |
| 1 – 10 | • Structured and Well Written Code<br>• High Testability<br>• Less Cost and Effort |
| 10 – 20 | • Complex Code<br>• Medium Testability<br>• Medium Cost and Effort |
| 20 – 40 | • Very Complex Code<br>• Low Testability<br>• High Cost and Effort |
| > 40 | • Highly Complex Code<br>• Not at all Testable<br>• Very High Cost and Effort |



$E - N + 2$

$13 - 10 + 2$

$= 5$ ✓

$V(G) = \pi + 1$

predicate node

$4 + 1 = 5$ ✓

$V(G) = \text{Closed path} + 1$

$= 4 + 1 = 5$

Calculate cyclomatic complexity for the given code-

```
1.  IF A = 354
2.     THEN IF B > C
3.          THEN A = B
4.          ELSE A = C
5.     END IF
6.  END IF
7.  PRINT A
```

## Method-01:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 2 + 1

= 3

## Method-02:

Cyclomatic Complexity

= E – N + 2

= 8 – 7 + 2

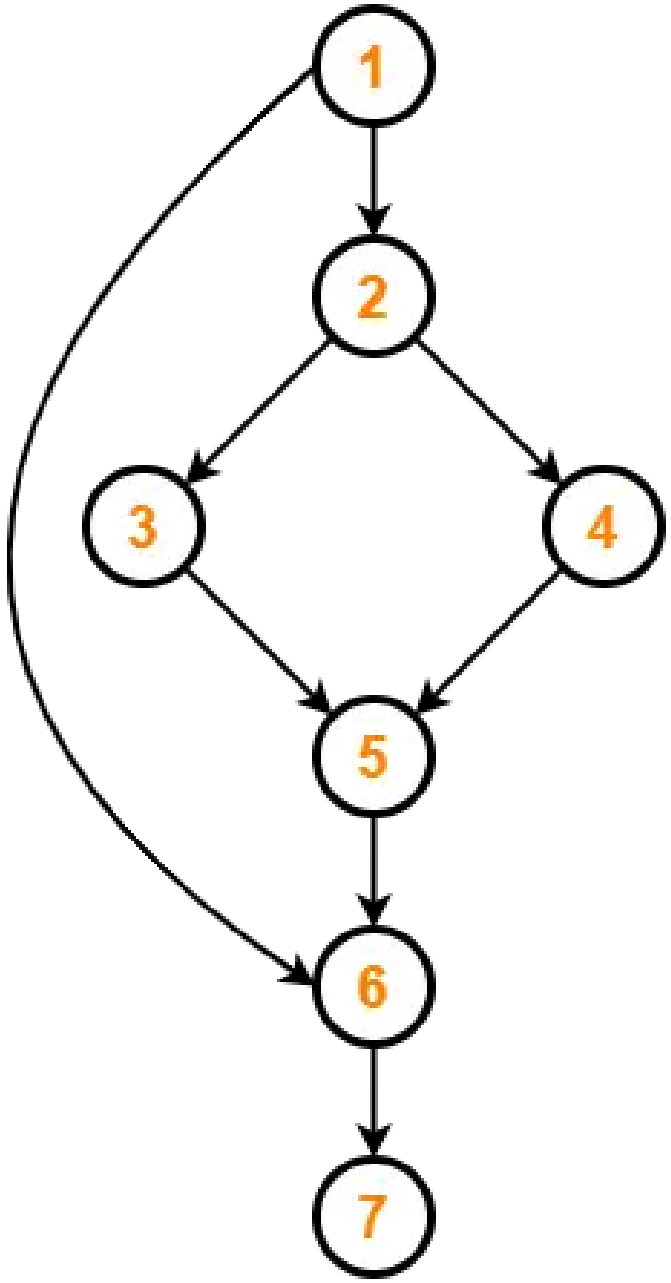
= 3

## Method-03:

Cyclomatic Complexity

= P + 1

= 2 + 1



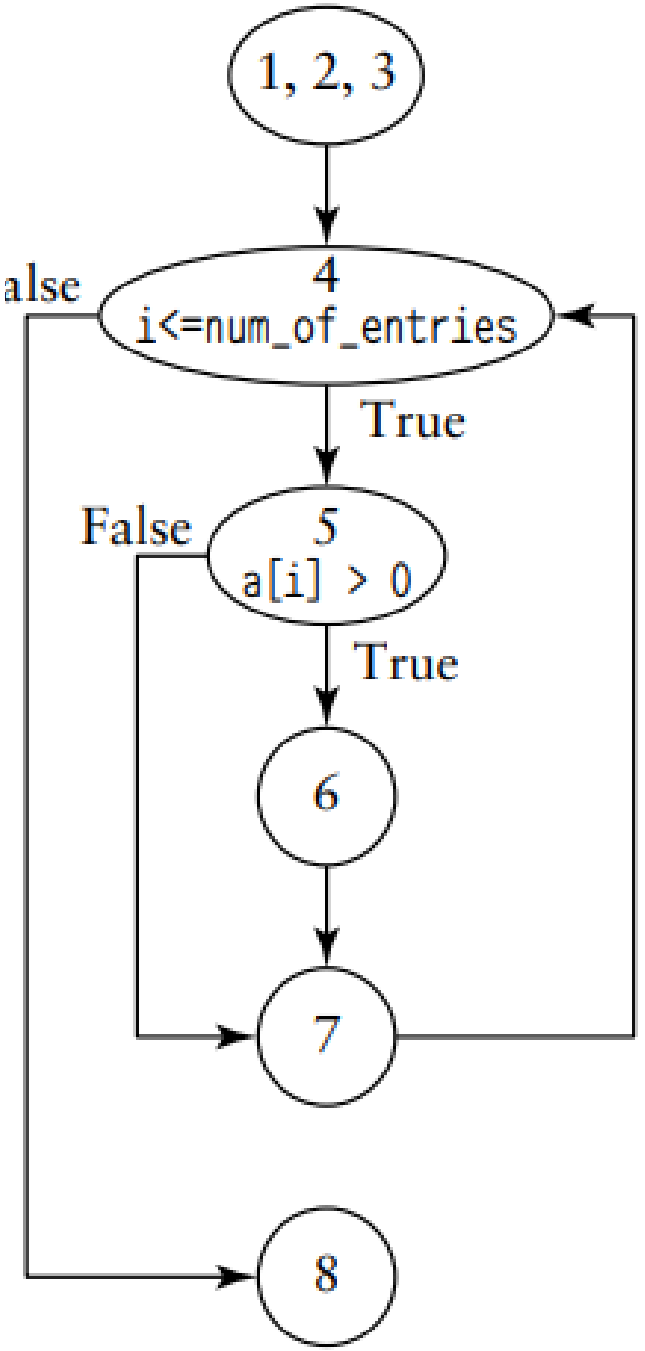

Control Flow Graph

# Covering Code Logic

*/* pos_sum finds the sum of all positive numbers (greater than zero) stored in an integer array a. Input parameters are num_of_entries, an integer, and a, an array of integers with num_of_entries elements. The output parameter is the integer sume */*

```
1.  pos_sum(a, num_of_entries, sum)
2.      sum = 0
3.      inti = 1
4.      while (i <= num_of_entries)
5.          if a[i] >0
6.              sum = sum + a[i]
            endif
7.          i = i + 1
        end while
8.  end pos_sum
```

*Code sample with branch and loop.*




*A control flow graph*

# **Paths:** control flow graph $(G)$

# McCabe's Cyclomatic Complexity $V(G)$

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

**E is the number of edges in the control flow graph and**
**N is the number of nodes**

$E = 7, N = 6$
$V(G) = 7 - 6 + 2 = 3$

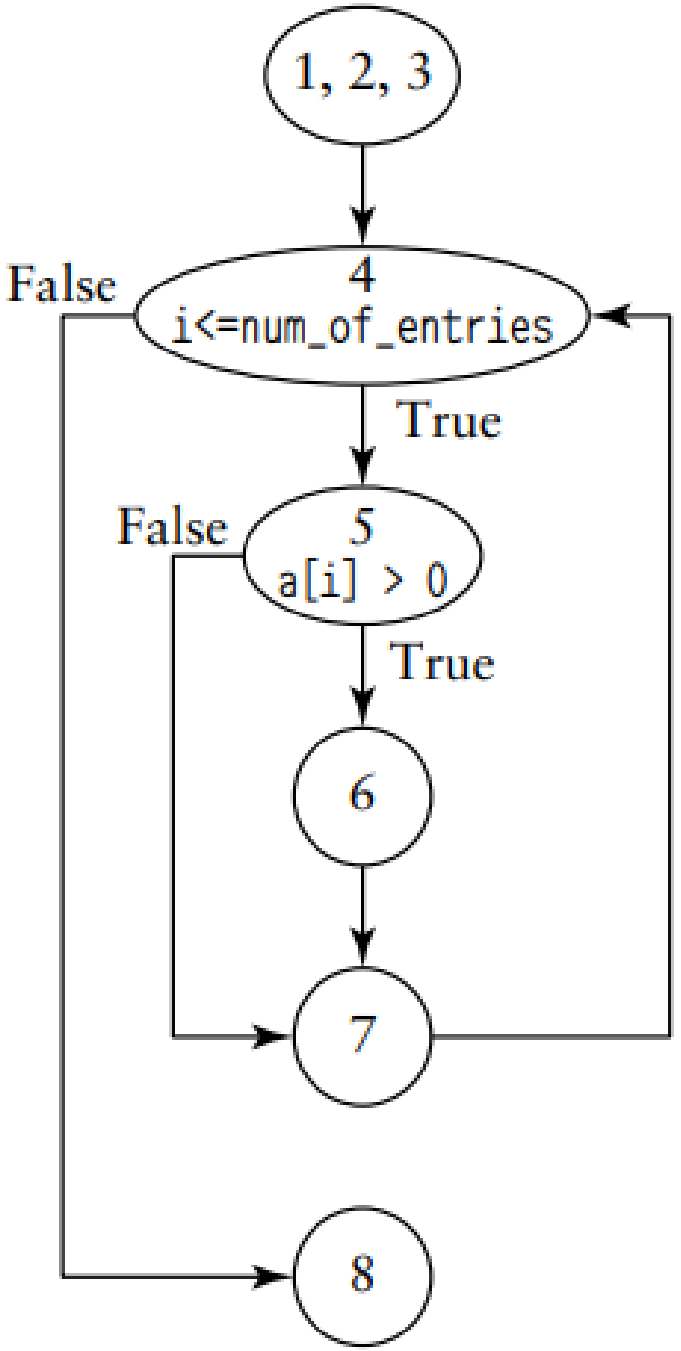
basis set
(i) 1-2-3-4-8
(ii) 1-2-3-4-5-6-7-4-8
(iii) 1-2-3-4-5-7-4-8

**It provide an approximation of the number of test cases needed for branch coverage in a module of structured code**

**Cyclomatic complexity is a measure of the number of so-called "independent" paths in the graph**

**A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.**

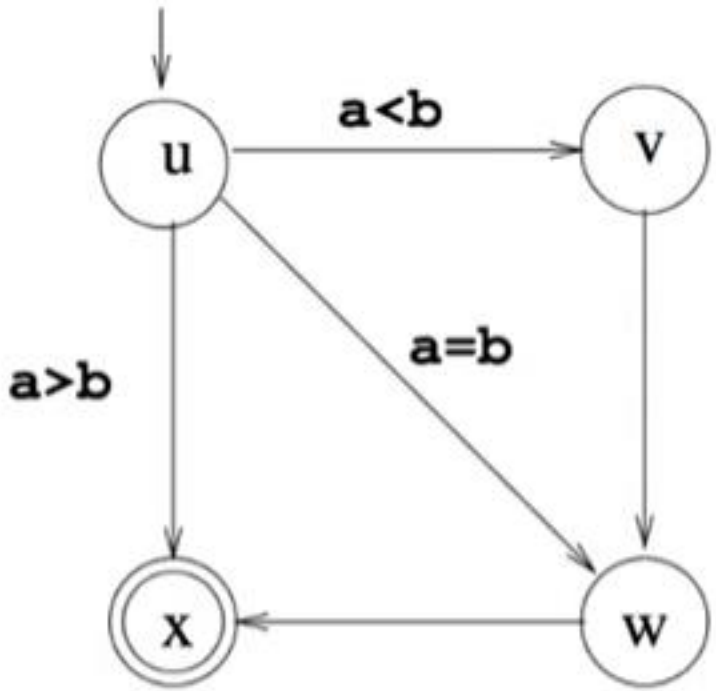
1-2-3-4-8



- A test path is a path that starts in an initial vertex and ends in a final vertex.

  Note: Initial and final vertices capture the beginning and ending of paths, respectively, in the corresponding graph.
- Test paths represent execution of test cases.
  - Some test paths can be executed by many test cases (Feasible paths).
  - Some test paths cannot be executed by any test case (Infeasible paths).

- When a test case $t$ executes a path, we call it the test path executed by $t$, denoted by $path(t)$.
- Similarly, the set of test paths executed by a set of test cases $T$ is denoted by $path(T)$.




Test case input: (a=0,b=1) Test path: u, v, x, x

Test case input:(a=1, b=1) Test path:u, w, x

Test case input:(a=3,b=1) Test path:u,x

- Test requirement describes properties of test paths.
- Test Criterion are rules that define test requirements.
- Satisfaction: Given a set $TR$ of test requirements for a criterion $C$, a set of tests $T$ satisfies $C$ on a graph iff for every test requirement in $t \in TR$, there is a test path in path(T) that meets the test requirement $t$.
- For example, the set of test cases below in the graph satisfy branch coverage at the node $u$ in the graph.

# Two different coverage criteria on graphs

We will consider two different coverage criteria for designing test cases based on graphs:

- Structural Coverage Criteria: Defined on a graph just in terms of vertices and edges.
- Data Flow Coverage Criteria: Requires a graph to be annotated with references to variables and defines criteria requirements based on the annotations.

*A test case for the code in Figure that satisfies the decision coverage criterion.*

100% decision coverage

| Decision or branch | Value of variable i | Value of predicate | Test case: Value of a, num_of_entries |
|---|---|---|---|
| | | | a=1, −45,3<br>num_of_entries = 3 |
| while | 1 | True | |
| | 4 | False | |
| if | 1 | True | |
| | 2 | False | |

If ($x$ < MIN and $y$ > MAX and (not INT Z))

(i) $x$ < MIN, (ii) $y$ > MAX, and (iii) not INT Z

| Condition 1 | Condition 2 |
|---|---|
| True | True |
| True | False |
| False | True |
| False | False |

```
if(age <65 and married == true)
        do X
        do Y ……..
else
        do Z
```

Condition 1: Age less than 65

Condition 2: Married is true

**Test cases for simple decision coverage**

| Value for age | Value for married | Decision outcome (compound predicate as a whole) | Test case ID |
|---|---|---|---|
| 30 | True | True | 1 |
| 75 | True | False | 2 |

```
if(age <65 and married = = true)
        do X
        do Y .....
else
        do Z
```

**Test cases for condition coverage**

| Value for age | Value for married | Condition 1 outcome | Condition 2 outcome | Test case ID |
|---|---|---|---|---|
| 75 | True | False | True | 2 |
| 30 | False | True | False | 3 |

A combination of test cases 1, 2, and 3 would satisfy this criterion.

**Test cases for decision condition coverage**

| Value for age | Value for married | Condition 1 outcome | Condition 2 outcome | Decision outcome (compound predicate as a whole) | Test case ID |
|---|---|---|---|---|---|
| 30 | True | True | True | True | 1 |
| 75 | True | False | True | False | 2 |
| 30 | False | True | False | False | 3 |

It is thestrongest program-based testing criterion, and it calls for complete path coverage; that is, every path (as distinguished from independent paths) in a module must be exercised by the test set at least once

**Advantages of Cyclomatic Complexity:.**

- It can be used as a quality metric, gives relative complexity of various designs.
- It is able to compute faster than the Halstead's metrics.
- It is used to measure the minimum effort and best areas of concentration for testing.
- It is able to guide the testing process.
- It is easy to apply.

**Disadvantages of Cyclomatic Complexity:**

- It is the measure of the program's control complexity and not the data complexity.
- In this, nested conditional structures are harder to understand than non-nestec structures.
- In case of simple comparisons and decision structures, it may give a misleading figure.

## Examples of tools are

- OCLint – Static code analyzer for C and Related Languages
- Reflector Add In – Code metrics for .NET assemblies
- GMetrics – Find metrics in Java related applications

$V(G) = E - N + 2 \times P$ where $E \rightarrow$ Edge

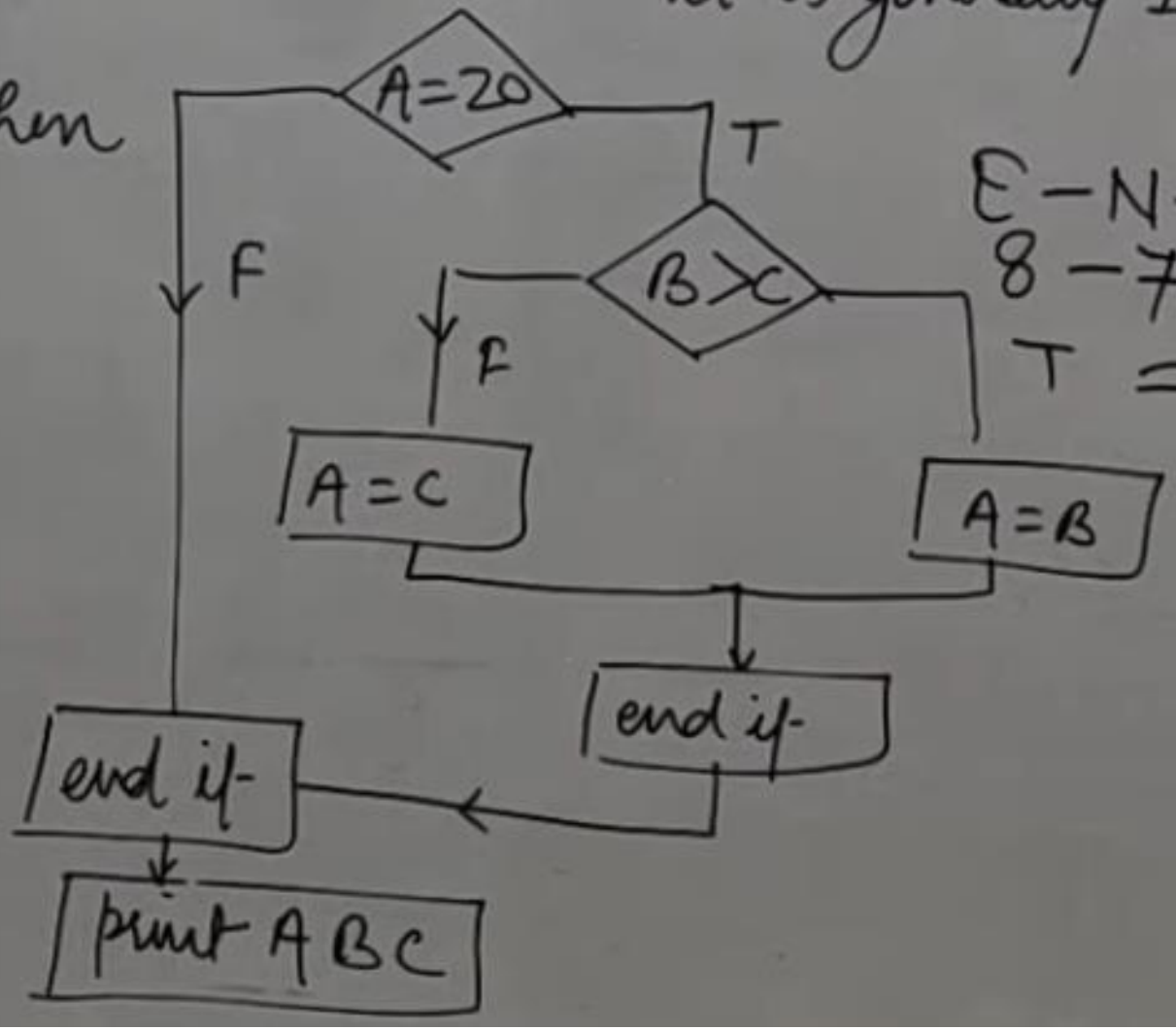
$N \rightarrow$ node

$P \rightarrow$ Connected component

it is generally 1 (for a single prog)

**Example**

```
If A=20 then
    if B>C then
        A=B
    else
        A=C
    end if
end if
print A
print B
print C
```




$E - N + 2 \times 1$

$8 - 7 + 2$

$T = 3$

Calculate cyclomatic complexity for the given code-

```
1.   { int i, j, k;
2.     for (i=0 ; i<=N ; i++)
3.     p[i] = 1;
4.     for (i=2 ; i<=N ; i++)
5.     {
6.        k = p[i]; j=1;
7.        while (a[p[j-1]] > a[k] {
8.            p[j] = p[j-1];
9.            j--;
10.    }
11.       p[j]=k;
12.  }
```

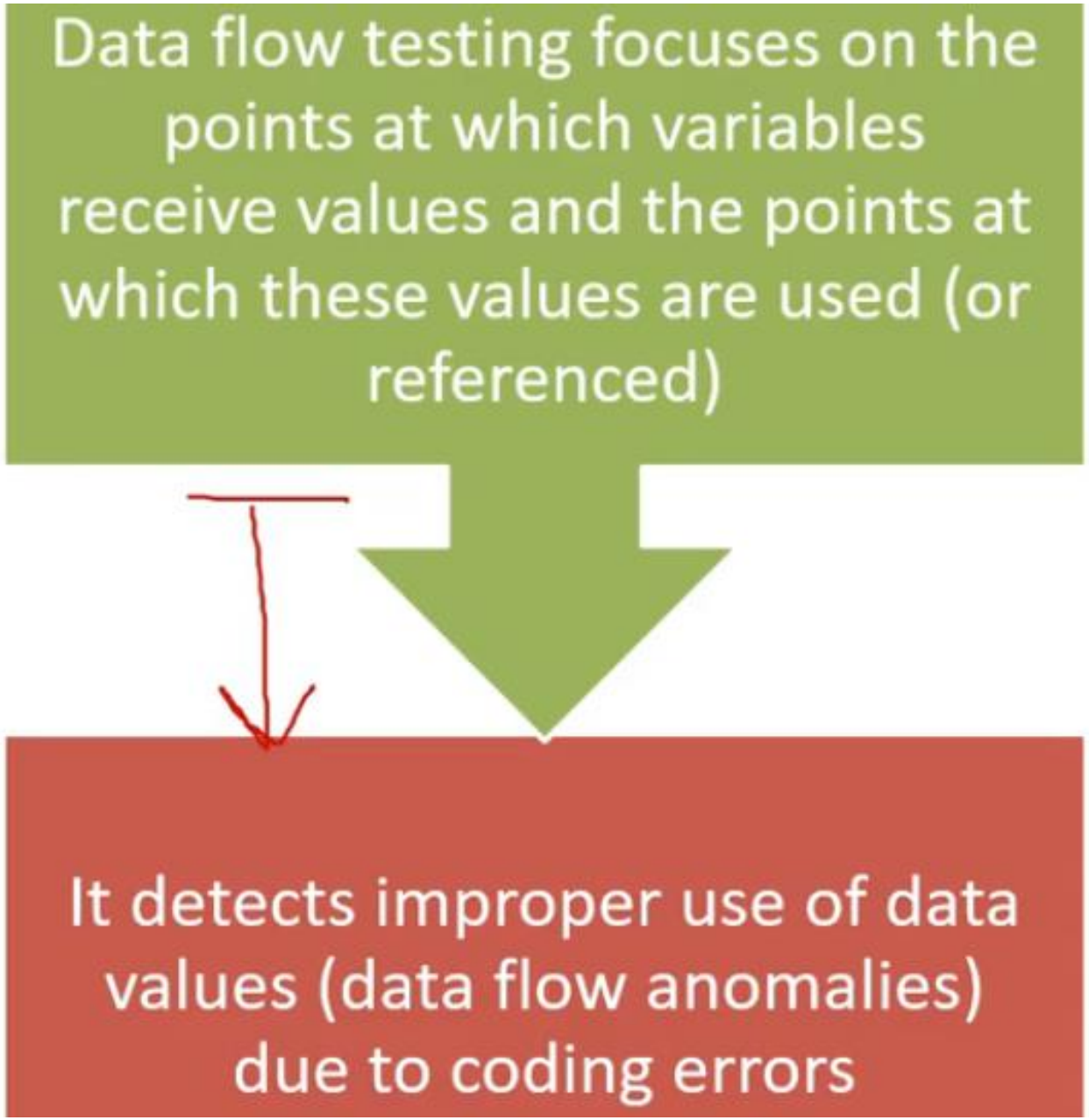
Calculate cyclomatic complexity for the given code-

```
1.  begin int x, y, power;
2.        float z;
3.        input(x, y);
4.        if(y<0)
5.        power = -y;
6.        else power = y;
7.        z=1;
8.        while(power!=0)
9.        {     z=z*x;
10.             power=power-1;
11.       } if(y<0)
12.       z=1/z;
13.       output(z);
14.       end
```

# Graphs with data

- Graph models of programs can be tested adequately by including values of variables (data values) as a part of the model.
- Data values are created at some point in the program and used later. They can be created and used several times.
- We deal with definition and use of data values.
- We will define coverage criteria that track a definition(s) of a variable with respect to its use(s).

# Data Flow Testing



# Data Flow Testing - Static Variants

- Early data flow analyses often centered on a set of faults that are known as define/reference anomalies
  - A variable that is defined but never used (referenced)
  - A variable that is used but never defined
  - A variable that is defined twice before it is used

**Data Flow Testing** is a type of structural testing. It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program. It has nothing to do with data flow diagrams.
It is concerned with:

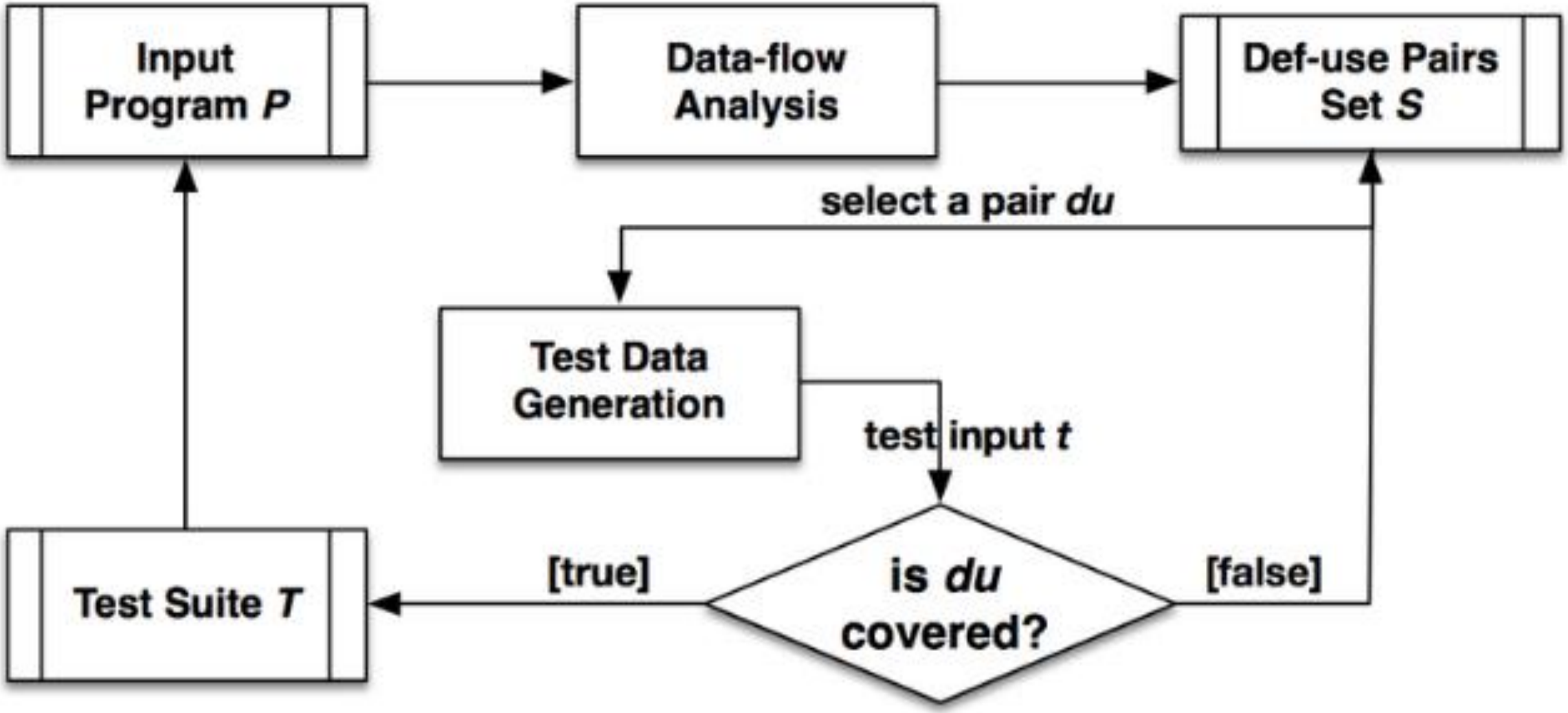
- Statements where variables receive values,
- Statements where these values are used or referenced.

To illustrate the approach of data flow testing, assume that each statement in the program assigned a unique statement number. For a statement number S-

```
DEF(S) = {X | statement S contains the definition of X}
USE(S) = {X | statement S contains the use of X}
```

- A definition (def) is a location where a value of a variable is stored into memory.
  - It could be through an input, an assignment statement etc.
- A use is a location where a value of a variable is is accessed.
  - It could be assigned to another variable, be a part of an if, while or other conditions etc.
- Values are *carried* from their defs to uses. We call these du-pairs or def-use pairs.
- A du-pair is a pair of locations $(l_i, l_j)$ such that a variable $v$ is defined at $l_i$ and used at $l_j$.

# Data Flow Testing



Data Flow Testing uses the control flow graph to find the situations that can interrupt the flow of the program.

Reference or define anomalies in the flow of the data are detected at the time of associations between values and variables.

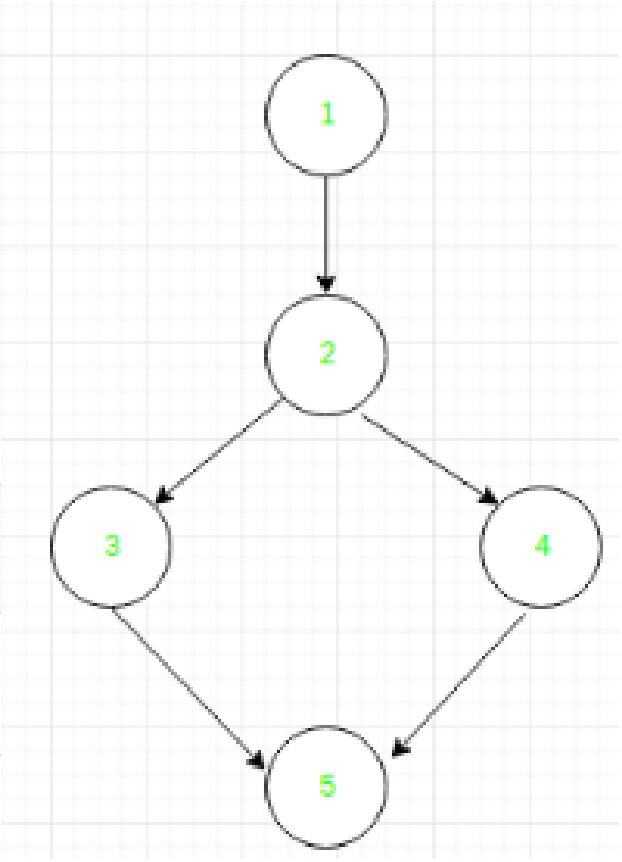
anomalies

- A variable is defined but not used or referenced,
- A variable is used but never defined,
- A variable is defined twice before it is used

```
1. read x, y;
2. if(x>y)
3. a = x+1
else
4. a = y-1
5. print a;
```

**Define/use of variables of above example:**

| Variable | Defined at node | Used at node |
|---|---|---|
| x | 1 | 2, 3 |
| y | 1 | 2, 4 |
| a | 3, 4 | 5 |

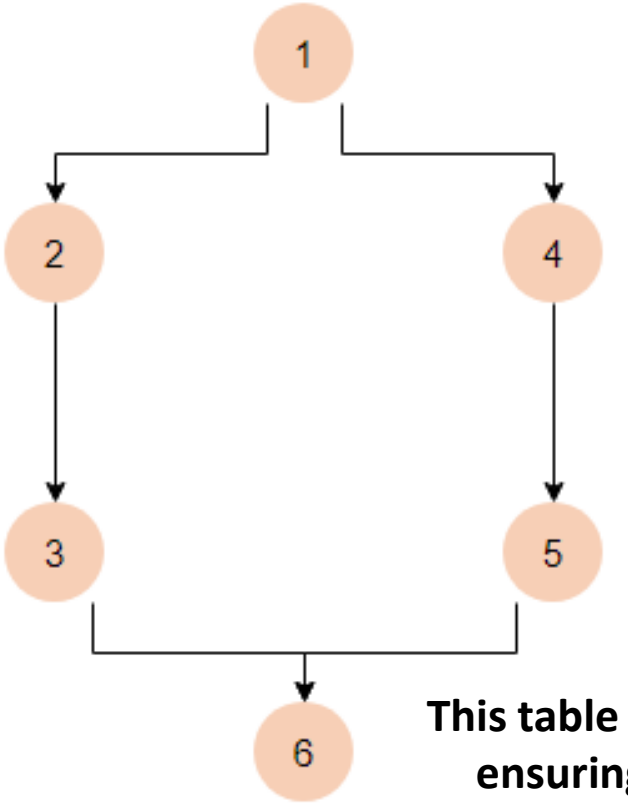


# Types of data flow testing

There are two types of data flow testing:

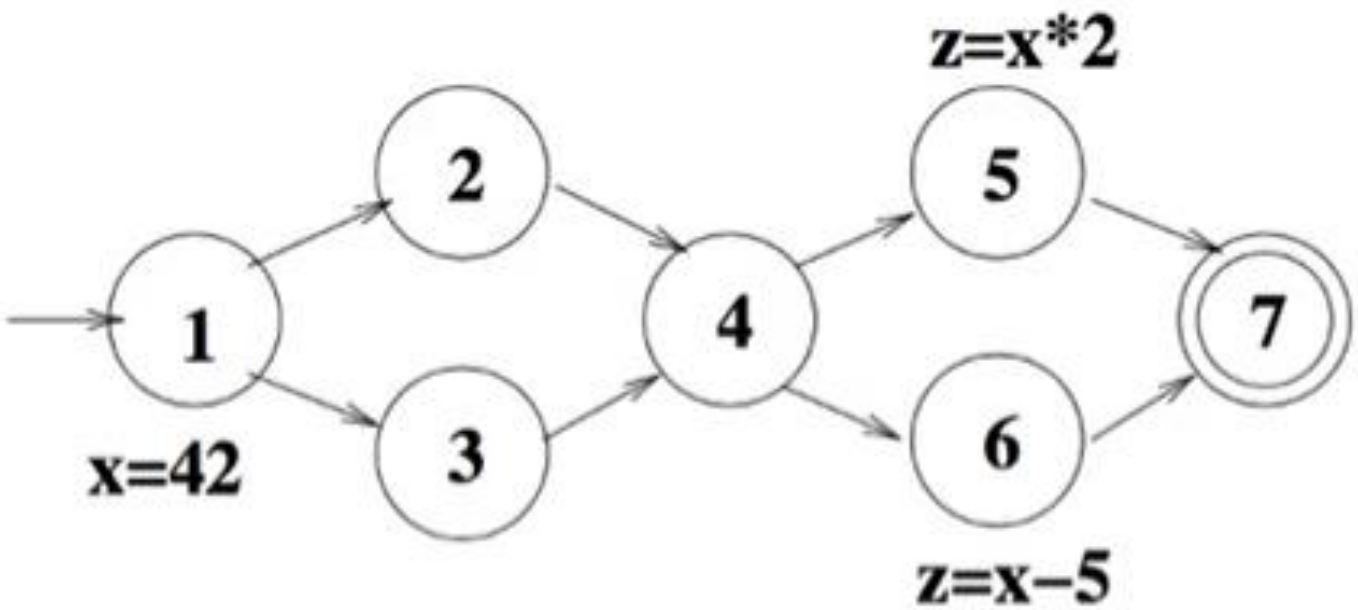
- **Static data flow testing:** The declaration, usage, and deletion of the variables are examined without executing the code. A control flow graph is helpful in this.
- **Dynamic data flow testing:** The variables and data flow are examined with the execution of the code.

```
1. input(x)
2. if(x>5)
3.      z = x + 10
4. else
5.      z = x - 5
6. print("Value of Z: ", z)
```




| Variable Name | Defined At | Used At |
|---|---|---|
| x | 1 | 2 |
| z | 3, 5 | 6 |

**This table to ensure that no anomaly occurs in the code by ensuring multiple tests. E.g., each variable is declared before it is used**




- Defs:
  - $def(1) = \{x\}$,
  - $def(5) =$
    $def(6) = \{z\}$.
- Uses:
  - $use(5) = \{x\}$,
  - $use(6) = \{x\}$.

# Making associations

**associations between two kinds of statements:**

- **Where variables are defined**
- **Where those variables are used**

An association is made with this format:

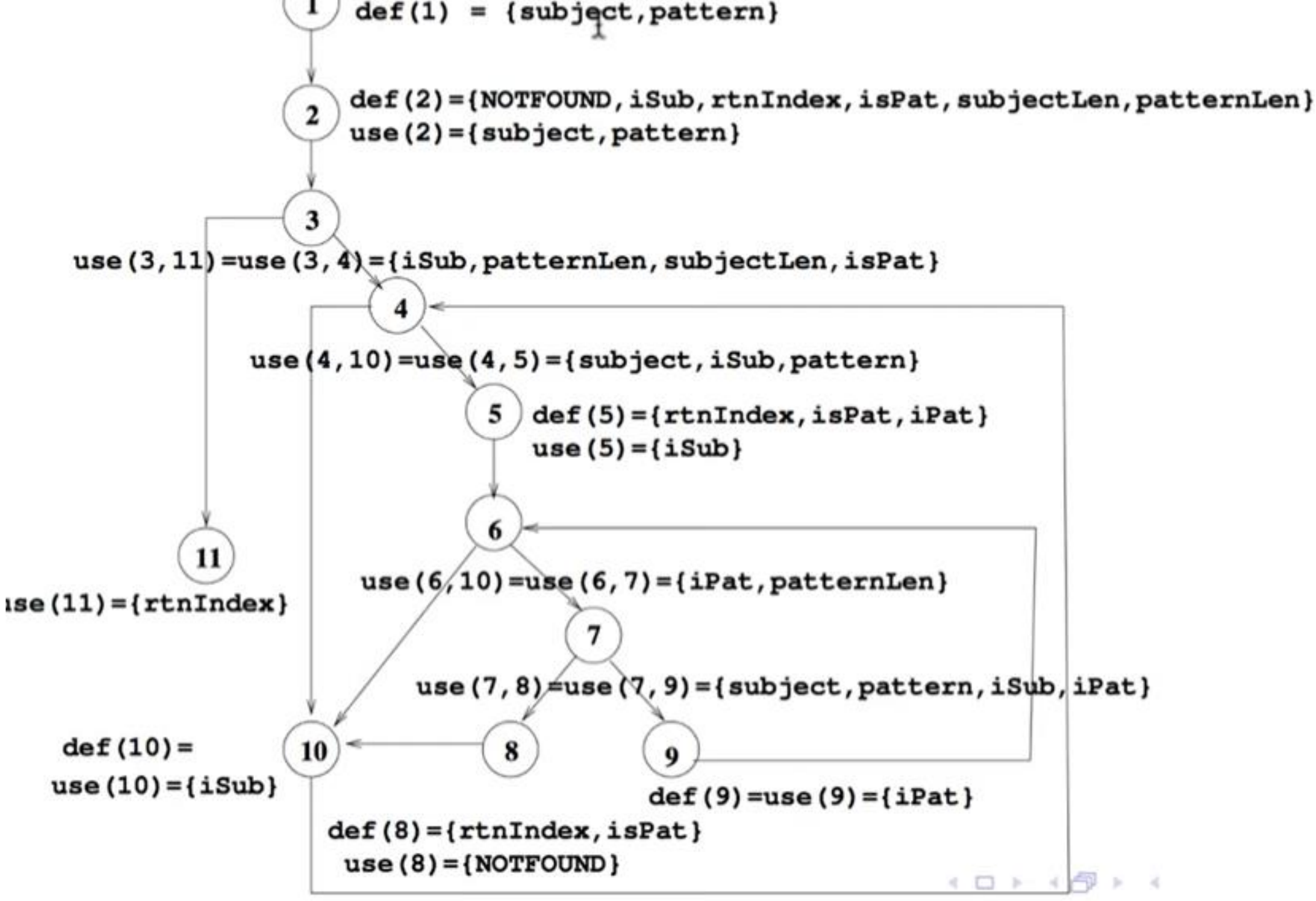
**(line number where the variable is declared, line number where the variable is used, name of the variable)**

**(L1_DEF, L2_USED, V)**

```
1. input(x)
2. if(x>5)
3.       z = x + 10
4. else
5.       z = x - 5
6. print("Value of Z: ", z)
```

- (1, (2,t), x): for the `true` case of IF statement in line 2
- (1, (2,f), x): for the `false` case of IF statement in line 2
- (1, 3, x): variable `x` is being used in line 3 to define the value of `z`
- (1, 5, x): variable `x` is being used in line 5 to define the value of `z`
- (3, 6, z): variable `z` is being used in line 6, which is defined in line 3
- (5, 6, z): variable `z` is being used in line 6, which is defined in line 5

```
public class PatternIndex
{  public static void main (String[] argv)
   {       if (argv.length != 2)
           { System.out.println
             ("java PatternIndex Subject Pattern");
             return; }
           String subject = argv[0];
           String pattern = argv[1];
           int n = 0;
           if ((n = patternIndex(subject, pattern))==-1)
             System.out.println
             ("Pattern is not a substring of the subject");
           else
             System.out.println
             ("Pattern string begins at character " + n);
   }
```


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

Now, there are two types of uses of a variable:

- **predicate use**: the use of a variable is called p-use. Its value is used to decide the flow of the program, e.g., line 2.
- **computational use**: the use of a variable is called c-use when its value is used compute another variable or the output, e.g., line 3.

After the associations are made, these associations can be divided into these groups:

- All definitions coverage
- All p-use coverage
- All c-use coverage
- All p-use, some c-use coverage
- All c-use, some p-use coverage
- All uses coverage

- **the tester makes test cases and examines each point.**
- **The statements and variables which are found to be extra are removed from the code.**

In testing literature, there are two notions of uses available.

- If $v$ is used in a computational or output statement, the use is referred to as a computation use (or c-use), and the pair is denoted as $dcu(l_i, l_j, v)$, where $v$ is defined at $l_i$ and used at $l_j$.
- If $v$ is used in a conditional statement, its use is called as a predicate use (or p-use).
- For conditional use, two def-use pairs appear:
  - $dpu(l_i, (l_j, l_t), v)$ and $dpu(l_i, (l_j, l_f), v)$, where $v$ is defined at $l_i$, used at $l_j$, but has two opposite flow directions $(l_j, l_t)$ and $(l_j, l_f)$.
  - The former denotes the true edge of the conditional statement in which $v$ is used; the latter the false edge.
- We will not distinguish between the above two uses in this course.

- A def of a variable may or may not reach a particular use.
  - A def of a variable $v$ at location $l_i$ will not reach use of $v$ at location $l_j$ if there is no path from $l_i$ to $l_j$.
  - The value of $v$ could be changed by another def before it reaches an use.
- 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.
- 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 $l_j$.
- Note: All the path definitions above are parameterized with respect to a variable $v$.

# Def-use paths

- A du-path with respect to a variable $v$ is a simple path that is def-clear from a def of $v$ to a use of $v$.
  - du-paths are parameterized by $v$.
  - They need to be simple paths.
  - There may be intervening uses on the path.
- $du(n_i, n_j, v)$: The set of du-paths from $n_i$ to $n_j$ for variable $v$.
- $du(n_i, v)$: The set of du-paths that start at $n_i$ for variable $v$.

# Grouping du-paths:

- Grouping according to definitions: Consider all du-paths with respect to a given variable defined in a given node.
- The def-path set $du(n_i, v)$ is the set of du-paths with respect to variable $v$ that start at node $n_i$.
- For large programs, the number of def-path sets can be large.
- We *do not* group du-paths by uses.

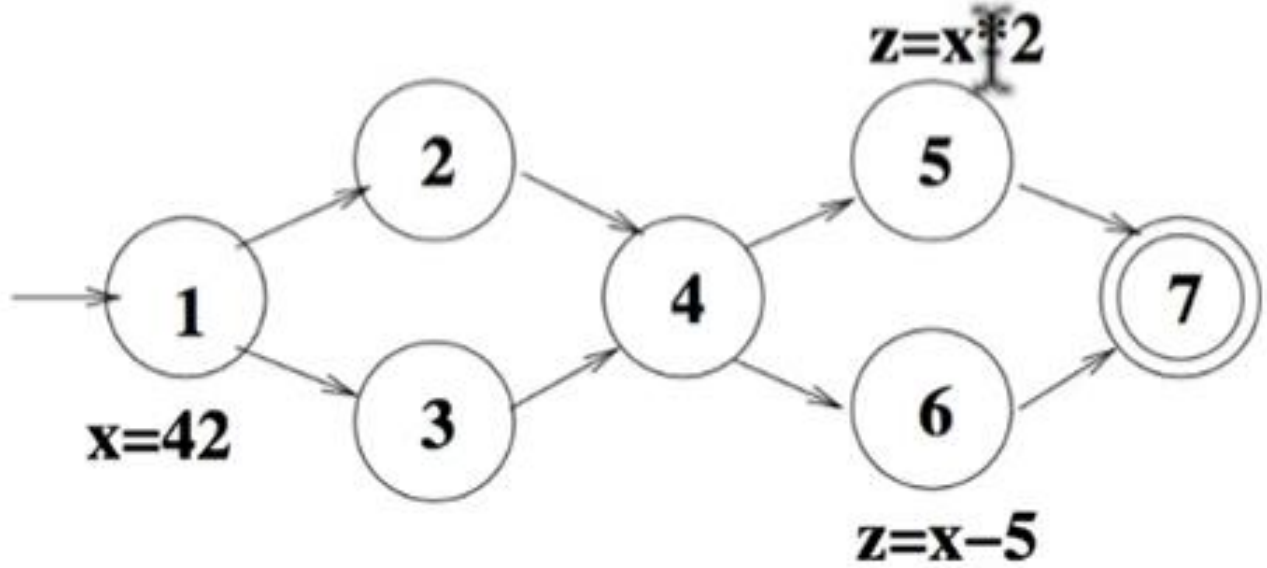
- Grouping du-paths as per definitions and uses allows definitions to flow to uses.
- A def-pair set, $du(n_i, n_j, v)$ is the set of du-paths with respect to variable $v$ that start at node $n_i$ and end at node $n_j$.
- A def-pair set collects together all the (simple) ways to get from a given definition to a given use.
- A def-pair set for a def at node $n_i$ is the union of all the def-path sets for that def. $du(n_i, v) = \cup_{n_j} du(n_i, n_j, v)$.

- All-Defs Coverage: For each def-path set $S = du(n, v)$, TR contains at least one path $d$ in $S$.
- All-Uses Coverage: For each def-pair set $S = du(n_i, n_j, v)$, TR contains at least one path $d$ in $S$.
  Since a def-pair set $du(n_i, n_j, v)$ represents all def-clear simple paths from a def of $v$ at $n_i$ to a use of $v$ at $n_j$, all-uses requires us to tour at least one path for every def-use pair.
- All-du-Paths Coverage: For each def-pair set $S = du(n_i, n_j, v)$, TR contains every path $d$ in $S$.

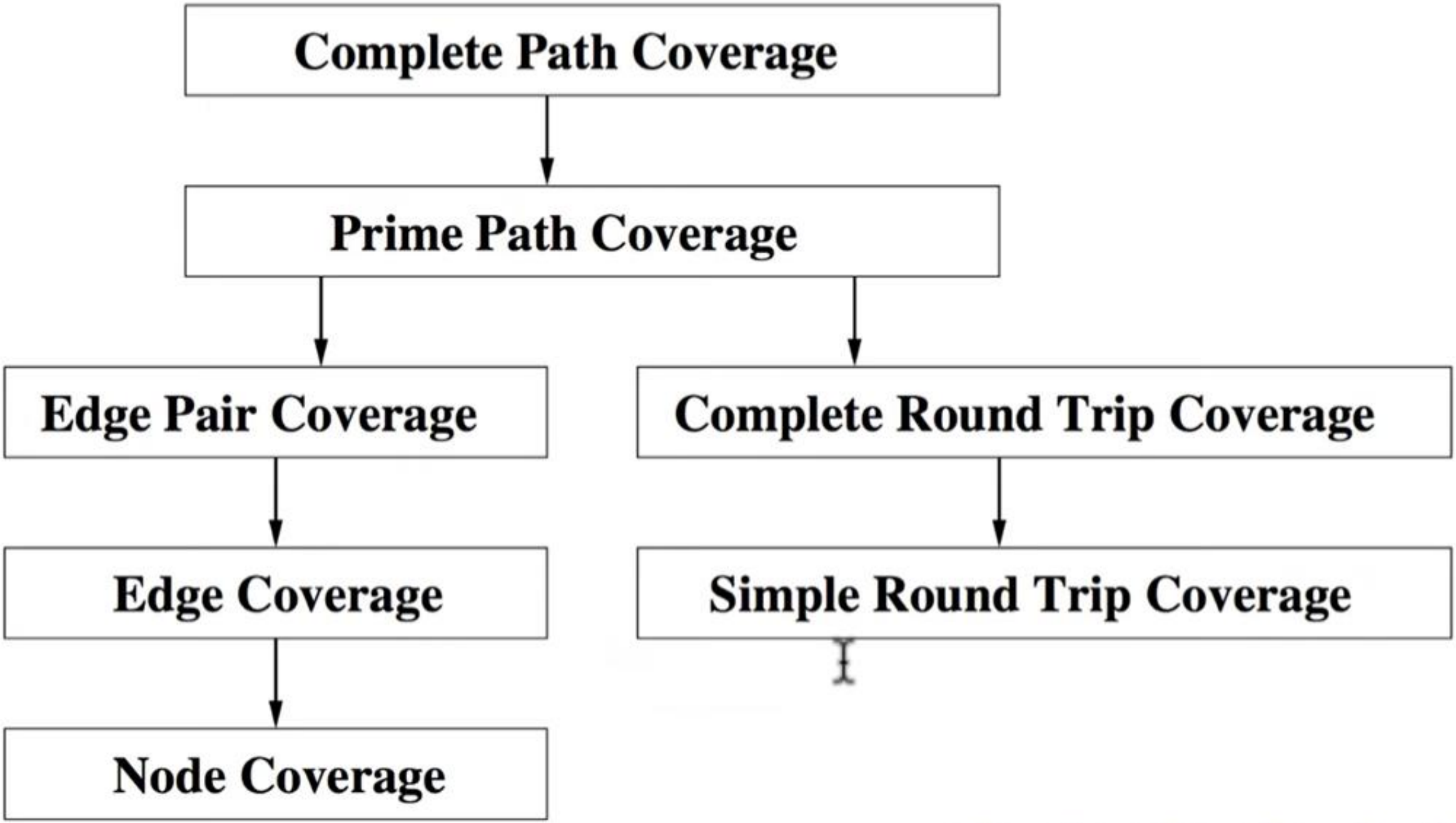
There are three common data flow criteria:

- TR: Each def reaches *at least one use*.
- TR: Each def reaches *all possible uses*.
- TR: Each def reaches all possible uses through *all possible du-paths*.

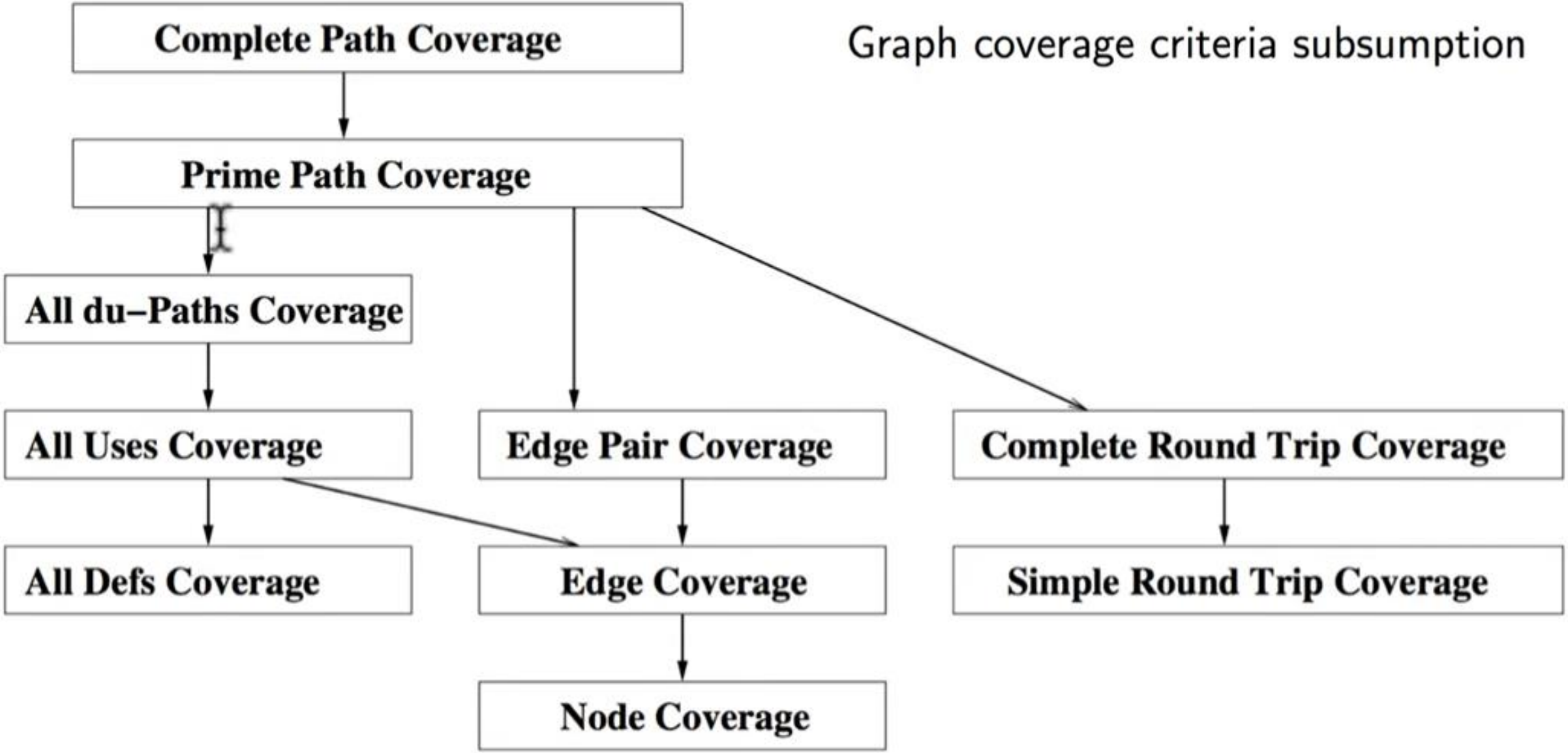
To get test paths to satisfy these criteria, we can assume best effort touring, i.e., side trips are allowed as long as they are def-clear.




- All defs for X: Test path is [1.2.4.6.7]
- All uses for X: Test paths are [1,2,4.5,7] and [1,2,4,6,7].
- All du-paths for X: Test paths are [1,2,4,5,7], [1,3,4,5,7], [1,2,4,6,7] and [1,3,4,6,7].


Complete Path Coverage
Prime Path Coverage
Edge Pair Coverage
Complete Round Trip Coverage
Edge Coverage
Simple Round Trip Coverage
Node Coverage

Graph coverage criteria subsumption



**We say a variable is defined in a statement when its value is assigned or changed.**

```
Y = 26 * X
Read (Y)
```

the variable *Y* is defined, that is, it is assigned a new value. In data flow notation this is indicated as a *def* for the variable *Y*.

**We say a variable is used in a statement when its value is utilized in a statement. The value of the variable is not changed.**

```
Y = 26 * X
```

the variable *X* is used. Specifically it has a *c-use*. In the statement

```
if (X > 98)
Y = max
```

*X* has a predicate or *p-use*.

several data-flow based test adequacy criteria

**All def**

**All p-uses**

**All c-uses/some p-uses**

**All p-uses/some c-uses**

**All uses**

**All def-use paths**

| | | |
|---|---|---|
| 1 | sum = 0 | *sum, def* |
| 2 | read (n), | *n, def* |
| 3 | i = 1 | *i, def* |
| 4 | while (i <= n) | *i, n p-sue* |
| 5 | read (number) | *number, def* |
| 6. | sum = sum + number | *sum, def, sum, number, c-use* |
| 7 | i = i + 1 | *i, def, c-use* |
| 8 | end while | |
| 9 | print (sum) | *sum, c-use* |

*Sample code with data flow*

The strongest of these criteria is all def-use paths. This includes all p- and c-uses.

Table for n

| pair id | def | use |
|---|---|---|
| 1 | 2 | 4 |

Table for number

| pair id | def | use |
|---|---|---|
| 1 | 5 | 6 |

Table for sum

| pair id | def | use |
|---|---|---|
| 1 | 1 | 6 |
| 2 | 1 | 9 |
| 3 | 6 | 6 |
| 4 | 6 | 9 |

Table for i

| pair id | def | use |
|---|---|---|
| 1 | 3 | 4 |
| 2 | 3 | 7 |
| 3 | 7 | 7 |
| 4 | 7 | 4 |

```
1   sum = 0                          sum, def
2   read (n),                        n, def
3   i = 1                            i, def
4   while (i <= n)                   i, n p-sue
5         read (number)              number, def
6.        sum = sum + number         sum, def, sum, number, c-us
7         i = i + 1                  i, def, c-use
8   end while
9   print (sum)                      sum, c-use
```

**Test data set 1:** $n = 0$

**Test data set 2:** $n = 5$, number = 1,2,3,4,5

# Basic blocks, defs and uses

- Identify the basic blocks
- Identify all the definitions
  - Where variables get their values
- Identify all the uses
  - Where variables are used
  - Indicate p-uses (in predicates)
  - Indicate c-uses (in computations)
- Draw path from each definition to each use that might get data from that definition

```
1.  s = 0;
2.  i = 1;
3.  while (i <= n)
    {
4.      s + = i;
5.      i ++
    }
6.  cout << s;
7.  cout << i;
8.  cout << n;
```

- Identify the basic blocks
- Identify all the definitions

**Basic block:** a part of code that executes without branching

  - Where variables are used
  - Indicate p-uses (in predicates)
  - Indicate c-uses (in computations)
- Draw path from each definition to each use that might get data from that definition

```
1.  s = 0;
2.  i = 1;
3.  while (i <= n)
    {
4.      s + = i;
5.      i ++
    }
6.  cout << s;
7.  cout << i;
8.  cout << n;
```

- Identify the basic blocks
- Identify all the definitions
  - Where variables get their values
- Identify all the uses
  - Where variables are used
  - Indicate p-uses (in predicates)
  - Indicate c-uses (in computations)
- Draw path from each definition to each use that might get data from that definition

```
1.  s = 0;          Def(s)
2.  i = 1;          Def(i)
3.  while (i <= n)
    {
4.      s + = i;    Def(s)
5.      i ++
    }
6.  cout << s;
7.  cout << i;
8.  cout << n;
```

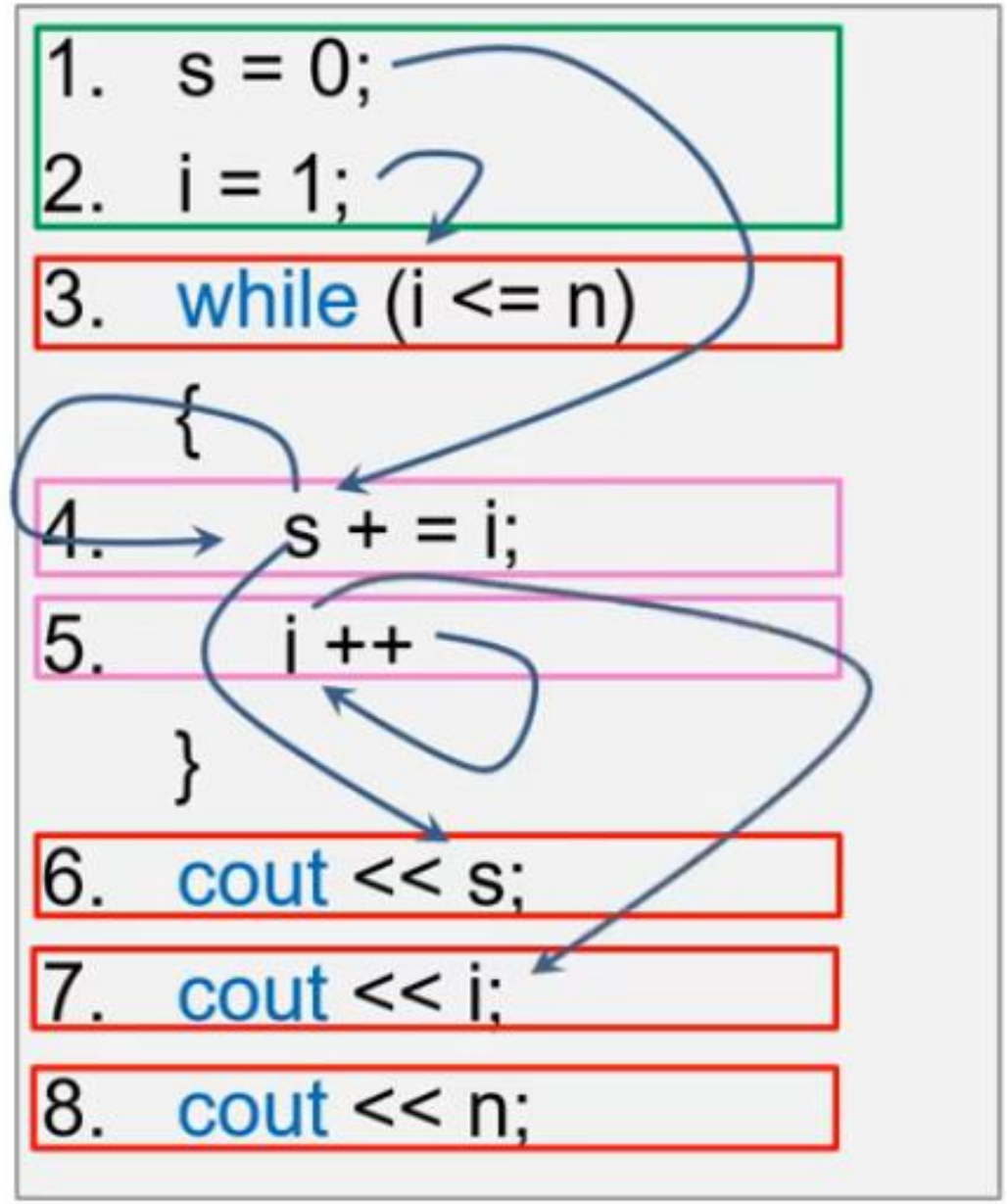
**p-use:** a path starts from a definition of the variable and ends in a statement in which it appears inside a certain predicate

- Identify all the uses
  - Where variables are used
  - Indicate p-uses (in predicates)
  - Indicate c-uses (in computations)

**c-use:** a path starts from a definition of the variable and ends at a statement in which

```
1.  s = 0;
2.  i = 1;                          p-use(i),
3.  while (i <= n)                  p-use(n)
    {
4.      s + = i;                    c-use(i)
5.      i ++                        c-use(i)
    }
6.  cout << s;                      c-use(s)
7.  cout << i;                      c-use(i)
8.  cout << n;                      c-use(n)
```

- Identify the basic blocks
- Identify all the definitions
  - Where variables get their values
- Identify all the uses
  - Where variables are used
  - Indicate p-uses (in predicates)
  - Indicate c-uses (in computations)
- Draw path from each definition to each use that might get data from that definition



- Given a program (P) written in an imperative programming language, its **program graph (G)** is a **directed graph** in which nodes (N) are **statement** fragments (basic block), and **edges** (E) represent **flow of control**. In addition it details the **definition, use** and **destruction** of each of the module's variable.

# What is Data Flow Testing

- White-box testing or source code based testing
- Variables' Definition and Usage
- What is a Variable?
- one of structured software testing methodologies
- Define Use Testing & Program Slices

# Data Flow Testing

- It is one of the white-box testing techniques.
- Dataflow Testing focuses on two points:
    1) In which statement the variables are defined.
    2) In which statement the variables are used.
- It designs the test cases that cover control flow paths around variable definitions and their uses in the modules.

# Example

30 20 10

1. read a, b, c;
2. if(a>b)
3. x = a+1
4. print x;

else

5. x = b-1
6. print z;

| | Define | Use |
|---|---|---|
| a | 1 | 2, 3 |
| b | 1 | 2, 5 |
| c | 1 | NA |
| x | 3, 5 | 4 |
| z | NA | 6 |

# Advantages of Data Flow Testing

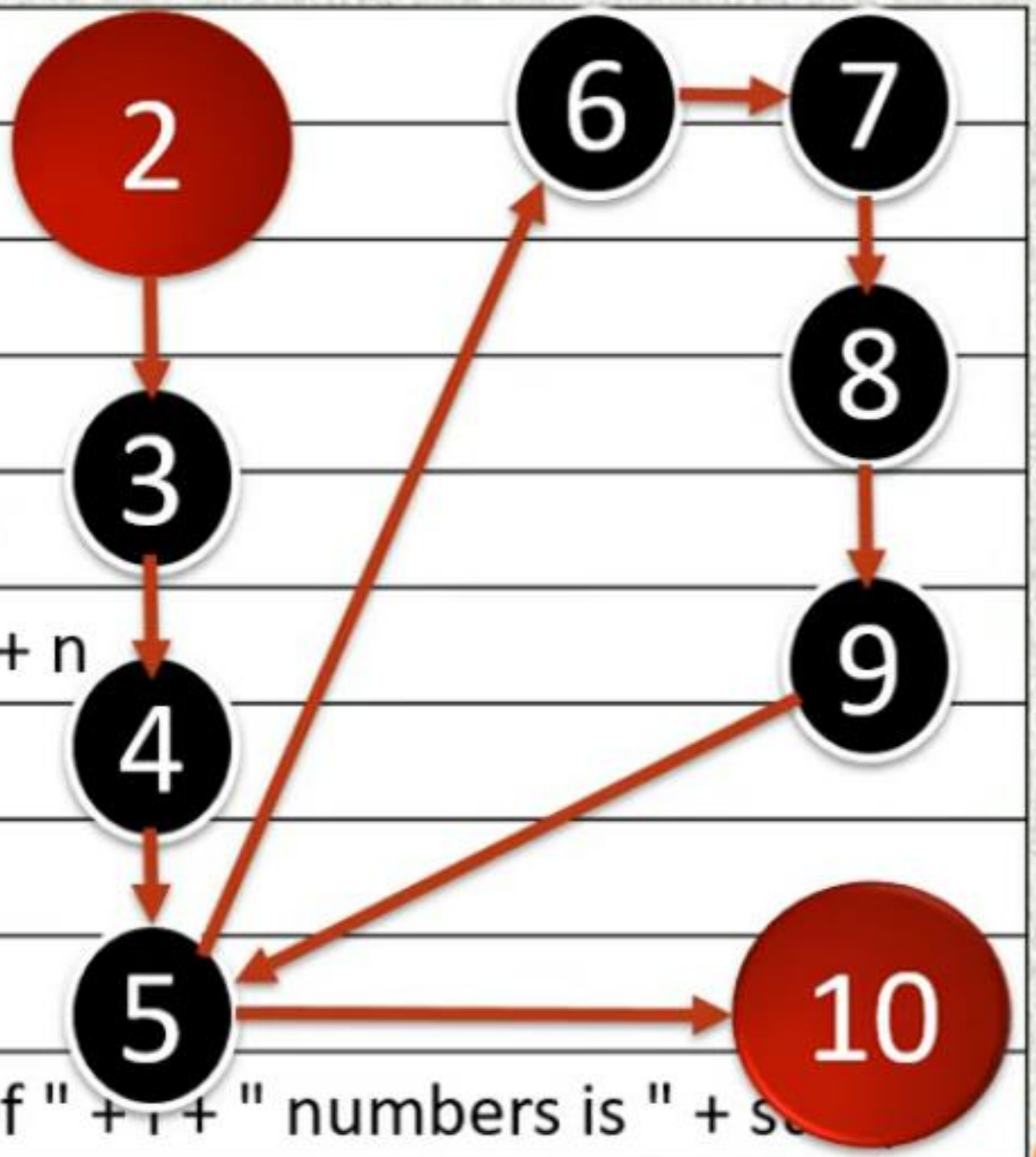
- A variable that is declared but never used within the program.
- A variable that is used but never declared.
- A variable that is defined multiple times before it is used.
- Deallocating a variable before it is used.

# DATA FLOW TESTING EXAMPLE

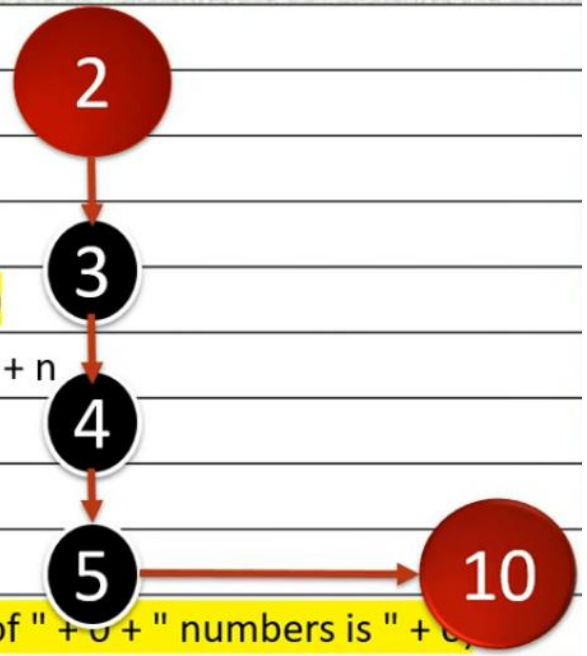
| | |
|---|---|
| Step 1: | var i, n, sum |
| Step 2: | i = 0 |
| Step 3: | sum = 0 |
| Step 4: | input (n) |
| Step 5: | While (n != 0) |
| Step 6: | sum = sum + n |
| Step 7: | i = i + 1 |
| Step 8: | input (n) |
| Step 9: | End While |
| Step 10: | print ("Total of " + i + " numbers is " + sum) |

# DATA FLOW TESTING EXAMPLE

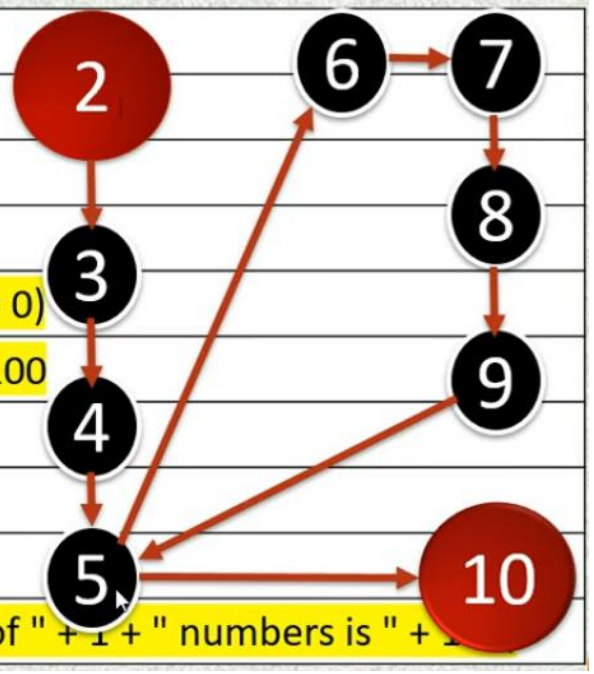
| | |
|---|---|
| Step 1: | var i, n, sum |
| Step 2: | i = 0 |
| Step 3: | sum = 0 |
| Step 4: | input (n) |
| Step 5: | While (n != 0) |
| Step 6: | sum = sum + n |
| Step 7: | i = i + 1 |
| Step 8: | input (n) |
| Step 9: | End While |
| Step 10: | print ("Total of " + i + " numbers is " + s |



| | |
|---|---|
| Step 1: | var i, n, sum |
| Step 2: | i = 0 |
| Step 3: | sum = 0 |
| Step 4: | input (0) |
| Step 5: | While (0 != 0) |
| Step 6: | sum = sum + n |
| Step 7: | i = i + 1 |
| Step 8: | input (n) |
| Step 9: | End While |
| Step 10: | print ("Total of " + 0 + " numbers is " + 0) |




| | |
|---|---|
| Step 1: | var i, n, sum |
| Step 2: | i = 0 |
| Step 3: | sum = 0 |
| Step 4: | input (100) |
| Step 5: | While (100 != 0) |
| Step 6: | sum = 0 + 100 |
| Step 7: | i = 0 + 1 |
| Step 8: | input (0) |
| Step 9: | End While |
| Step 10: | print ("Total of " + 1 + " numbers is " + 1) |

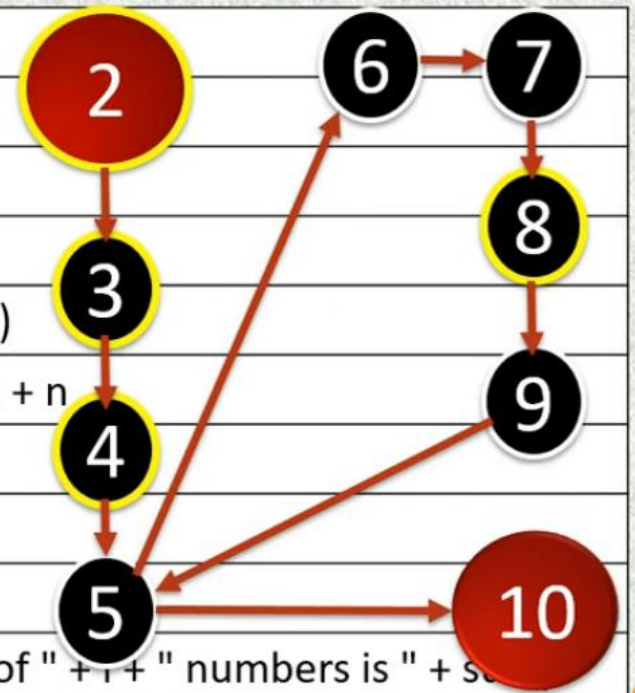



# DEFINE USE TESTING

- One of Data Flow Testing techniques
- Uses Paths related to Variables in CFG
- Variable is either defined or used

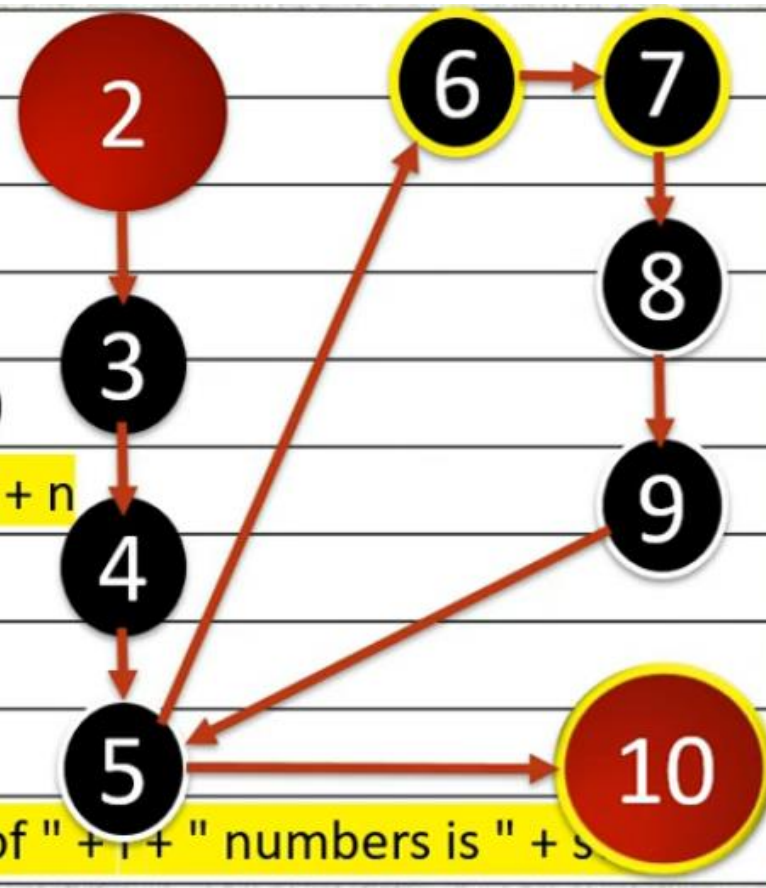
# DEFINE NODE

| | |
|---|---|
| Step 1: | var i, n, sum |
| Step 2: | i = 0 |
| Step 3: | sum = 0 |
| Step 4: | input (n) |
| Step 5: | While (n != 0) |
| Step 6: | sum = sum + n |
| Step 7: | i = i + 1 |
| Step 8: | input (n) |
| Step 9: | End While |
| Step 10: | print ("Total of " + i + " numbers is " + s |




# USE NODE

| | |
|---|---|
| Step 1: | var i, n, sum |
| Step 2: | i = 0 |
| Step 3: | sum = 0 |
| Step 4: | input (n) |
| Step 5: | While (n != 0) |
| Step 6: | sum = sum + n |
| Step 7: | i = i + 1 |
| Step 8: | input (n) |
| Step 9: | End While |
| Step 10: | print ("Total of " + i + " numbers is " + s |



# USE TESTING

- One of Data Flow Testing techniques
- Uses Paths related to Variables in CFG
- Variable is either defined or used
- Define node
- Use node
  - P-use (e.g. n!=0)
  - C-use (e.g. for variable n: sum = sum + n)
    - (O-use, L-use or I-use)

## Define Use Nodes for variable i

| | | |
|---|---|---|
| Step 1: | var i, n, sum | |
| Step 2: | i = 0 | DEF |
| Step 3: | sum = 0 | |
| Step 4: | input (n) | |
| Step 5: | While (n != 0) | |
| Step 6: | sum = sum + n | |
| Step 7: | i = i + 1 | DEF, USE |
| Step 8: | input (n) | |
| Step 9: | End While | |
| Step 10: | print ("Total of " + i + " numbers is " + sum) | USE |

## Define Use Nodes for variable n

| | | |
|---|---|---|
| Step 1: | var i, n, sum | |
| Step 2: | i = 0 | |
| Step 3: | sum = 0 | |
| Step 4: | input (n) | DEF |
| Step 5: | While (n != 0) | USE |
| Step 6: | sum = sum + n | USE |
| Step 7: | i = i + 1 | |
| Step 8: | input (n) | DEF |
| Step 9: | End While | |
| Step 10: | print ("Total of " + i + " numbers is " + sum) | |

## Define Use Nodes for variable sum

| | | |
|---|---|---|
| Step 1: | var i, n, sum | |
| Step 2: | i = 0 | |
| Step 3: | sum = 0 | DEF |
| Step 4: | input (n) | |
| Step 5: | While (n != 0) | |
| Step 6: | sum = sum + n | DEF, USE |
| Step 7: | i = i + 1 | |
| Step 8: | input (n) | |
| Step 9: | End While | |
| Step 10: | print ("Total of " + i + " numbers is " + sum) | USE |

# What are Data Flow Anomalies?

Data Flow Anomalies are identified while performing while box testing or Static Testing.

Data flow anomalies are represented using two characters based on the sequence of actions. They are defined (d), killed (k), and used (u). There are nine possible combinations based on these 3 sequence of actions which are dd, dk, du, kd, kk, ku, ud, uk, uu.

| Combination | Description | Anomaly possibilities |
|---|---|---|
| dd | Defined the data objects twice | Harmless but suspicious |
| dk | Defined the data object but killed it without using it. | Bad Programming Practice |
| du | Defined the data object and using it | NOT an Anomaly |
| kd | Killed the Data Object and redefined | NOT an Anomaly |
| kk | Killed the Data Object and killed it again | Bad Programming Practice |
| ku | Killed the Data Object and then used | Defect |
| ud | Used the Data Object and redefined | NOT an Anomaly |
| uk | Used the Data Object and Killed | NOT an Anomaly |
| uu | Used the Data Object and used it again | NOT an Anomaly |

# ANOMALY DETECTION

- Anomaly Detection is divided into two types
- Static Anomaly Detection
- Dynamic Anomaly Detection
  - Static analysis is analysis done on source code without actually executing it.
  - For example: source code syntax error detection is the static analysis result.
  - Dynamic analysis is done on intermediate values that result warning is the dynamic result.

# Limitations of static Anomaly Detection

- **Dead Variables:** Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.
- **Arrays:** Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array.

**Recoverable Anomalies and Alternate State Graphs:**

**Concurrency, Interrupts, System Issues:**

# DATA FLOW ANOMALY STATE GRAPH

- Data flow anomaly model prescribes that an object can be in one of four distinct states:
- **K** :- undefined, previously killed, doesnot exist
- **D** :- defined but not yet used for anything
- **U** :- has been used for computation or in predicate
- **A** :- anomalous

# DATA FLOW ANOMALY STATE GRAPH

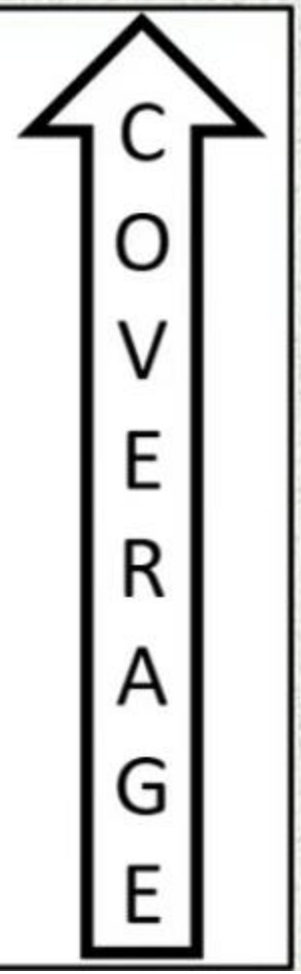
- These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.
- **Unforgiving Data - Flow Anomaly Flow Graph:** Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.
- Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible.

# Define Use Testing

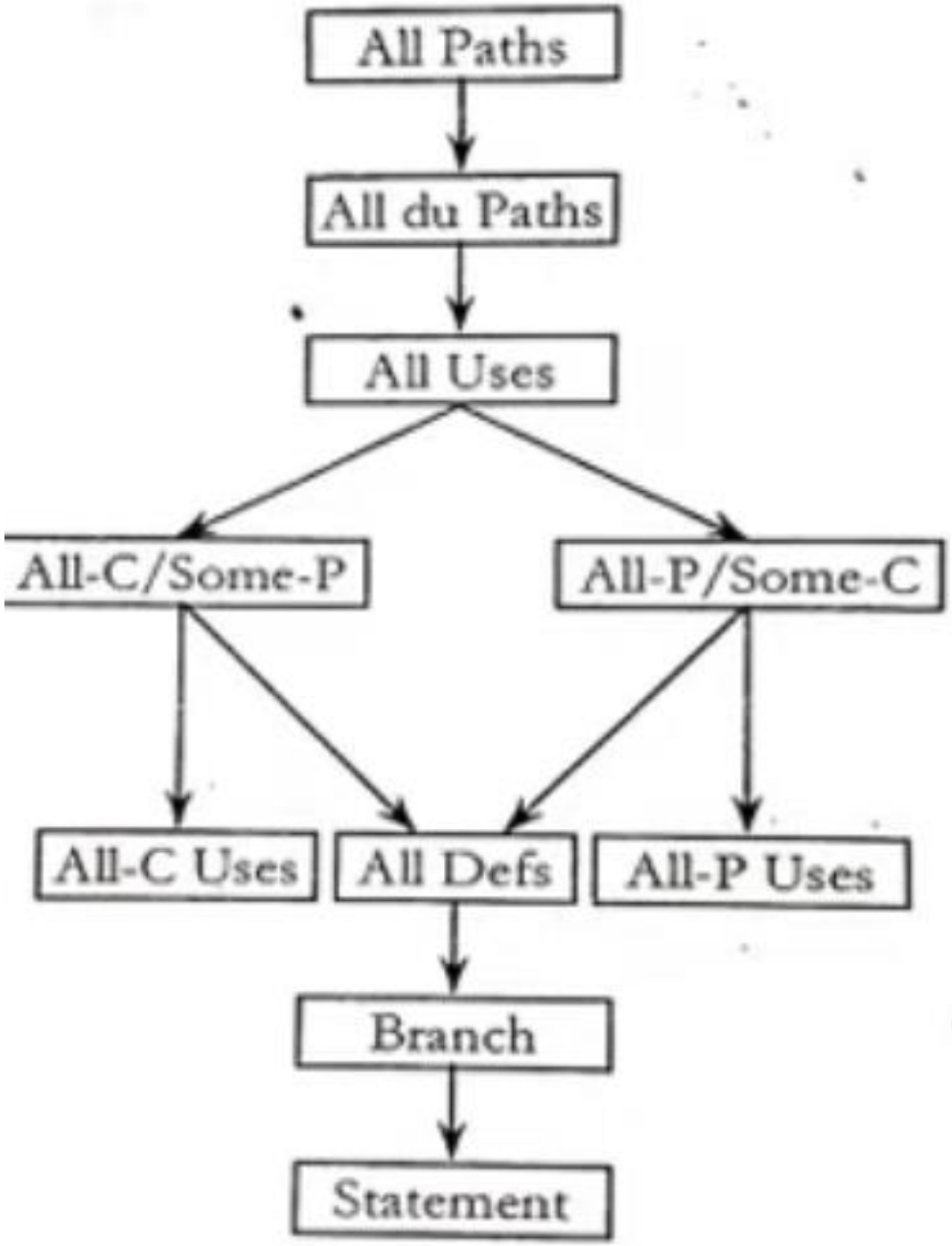
- Select a set of paths and test them to find anomalies (bugs)
- All-Paths (too many!), All-Edges and All-Nodes
- DU Path (Define Use Path)
- DC Path (Define Clear Path)

# All-DU-Paths

- All-Uses
  - All-C-Uses/Some-P-Uses
  - All-P-Uses/Some-C-Uses
    - All-Defs
    - All-P-Uses



# Ordering the Strategies



**DATA FLOW ANOMALIES:**

- An anomaly is denoted by a two-character sequence of actions.
- For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.
- What is an anomaly is depend on the application.
- There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.
    1. **dd** :- probably harmless but suspicious. Why define the object twice without an intervening usage?
    2. **dk** :- probably a bug. Why define the object without using it?
    3. **du** :- the normal case. The object is defined and then used.
    4. **kd** :- normal situation. An object is killed and then redefined.
    5. **kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
    6. **ku** :- a bug. the object doesnot exist.
    7. **ud** :- usually not a bug because the language permits reassignment at almost any time.
    8. **uk** :- normal situation.
    9. **uu** :- normal situation.

# Slice Based Testing in simple terms

- It divides the program into different slices and tests that slice which can majorly affect the entire software.
- The idea of slices is to separate a program into components that have some useful meaning
- Slicing criterion for a program specifies a windows for observing its behavior
  - Where a window,
    - Specified as a statement and a set of variables
    - Allows the observations of values of the specified variables before execution of particular statement
    - Ex: Value at line 25

# Slices for Variable i

| | | | |
|---|---|---|---|
| Step 1: | var i, n, sum | | |
| Step 2: | i = 0 | S (i, 2) | {2} |
| Step 3: | sum = 0 | | |
| Step 4: | input (n) | | |
| Step 5: | While (n != 0) | | |
| Step 6: | sum = sum + n | | |
| Step 7: | i = i + 1 | S (i, 7) | {2, 5, 7, 9} |
| Step 8: | input (n) | | |
| Step 9: | End While | | |
| Step 10: | print ("Total of " + i + " numbers... | S (i,10) | {2, 5, 7, 9} |

# Slices for Variable n

| | | | |
|---|---|---|---|
| Step 1: | var i, n, sum | | |
| Step 2: | i = 0 | | |
| Step 3: | sum = 0 | | |
| Step 4: | input (n) | S (n, 4) | {4} |
| Step 5: | While (n != 0) | S (n, 5) | {4, 5, 8, 9} |
| Step 6: | sum = sum + n | S (n, 6) | {4, 5, 8, 9} |
| Step 7: | i = i + 1 | | |
| Step 8: | input (n) | S (n, 8) | {8} |
| Step 9: | End While | | |
| Step 10: | print ("Total of " + i + " numbers... | | |

# Slices for Variable sum

| | | | |
|---|---|---|---|
| Step 1: | var i, n, sum | | |
| Step 2: | i = 0 | | |
| Step 3: | sum = 0 | S (sum, 3) | {3} |
| Step 4: | input (n) | | |
| Step 5: | While (n != 0) | | |
| Step 6: | sum = sum + n | S (sum, 6) | {3, 4, 5, 6, 8, 9} |
| Step 7: | i = i + 1 | | |
| Step 8: | input (n) | | |
| Step 9: | End While | | |
| Step 10: | print ("Total of " + i + " numbers... | S (sum, 10) | {3, 4, 5, 6, 8, 9} |

# DU PATHS FOR VARIABLE I

| | | | | |
|---|---|---|---|---|
| Step 1: | var i, n, sum | | | |
| Step 2: | i = 0 | DEF | S (i, 2) | {2} |
| Step 3: | sum = 0 | | | |
| Step 4: | input (n) | | | |
| Step 5: | While (n != 0) | | | |
| Step 6: | sum = sum + n | | | |
| Step 7: | i = i + 1 | DEF, USE | S (i, 7) | {2, 5, 7, 9} |
| Step 8: | input (n) | | | |
| Step 9: | End While | | | |
| Step 10: | print ("Total of " + i + " numbers... | USE | S (i,10) | {2, 5, 7, 9} |

<2, 7>
<2, 7, 10>
<7, 9, 10>
<7, 5, 7, 9, 10>

# DU PATHS FOR VARIABLE N

| | | | | |
|---|---|---|---|---|
| Step 1: | var i, n, s | | | |
| Step 2: | i = 0 | | | |
| Step 3: | sum = 0 | | | |
| Step 4: | input (n) | DEF | S (n, 4) | {4} |
| Step 5: | While (n != 0) | USE | S (n, 5) | {4, 5, 8, 9} |
| Step 6: | sum = sum + n | USE | S (n, 6) | {4, 5, 8, 9} |
| Step 7: | i = i + 1 | | | |
| Step 8: | input (n) | DEF | S (n, 8) | {8} |
| Step 9: | End While | | | |
| Step 10: | print ("Total of " + i + " numbers... | | | |

<4, 5>
<4, 5, 6>
<8, 9, 5>
<8, 9, 5, 6>

# DU PATHS FOR VARIABLE SUM

| | | | | |
|---|---|---|---|---|
| Step 1: | var i, n, sum | | | |
| Step 2: | i = 0 | | | |
| Step 3: | sum = 0 | DEF | | |
| Step 4: | input (n) | | | |
| Step 5: | While (n != 0) | | | |
| Step 6: | sum = sum + n | DEF, USE | S (sum, 6) | {3, 4, 5, 6, 8, 9} |
| Step 7: | i = i + 1 | | | |
| Step 8: | input (n) | | | |
| Step 9: | End While | | | |
| Step 10: | print ("Total of " + i + "... | USE | S (sum, 10) | {3, 4, 5, 6, 8, 9} |

<3, 6>
<3, 4, 5, 6, 8, 9, 5, 10>
<6>
<6, 8, 9, 5, 10>

Design, develop, code and run the program in any suitable language to solve the commission problem. Analyze it from the perspective of dataflow testing, derive different test cases, execute these test cases and discuss the test results.

## Loop Testing

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely;

(ii) one iteration of the loop;

(iii) two iterations of the loop;

(iv) $k$ iterations of the loop where $k < n$;

(v) $n - 1$ iterations of the loop;

(vi) $n + 1$ iterations of the loop (if possible).