Lecture 5: White Box Approach for Test Case Designing

Kĩ thuật kiểm thử hộp trắng

IT4501

Bùi Thị Mai Anh (anhbtm@soict.hust.edu.vn)

Bộ môn Công nghệ phần mềm

Viện công nghệ thông tin và truyền thông

Contents

Black Box vs. White Box

- Black Box: functional testing
  - Unit test
  - Integration test
  - System test
  - Programmers & Test Engineers & Quality Assurance Engineers
- White Box: structural testing
  - Unit test
  - Integration test
  - Programmers & Test Engineers

- External/user view:
  – Check conformance with specification
- Abstraction from details:
  – Source code not needed
- Scales up:
  – Different techniques at different levels of granularity

- Internal/developer view:
  – Allows tester to be confident about test coverage
- Based on control or data flow:
  – Easier debugging
- Does not scale up:
  – Mostly applicable at unit and integration testing levels

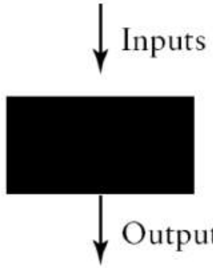
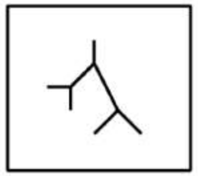
USE BOTH!

Effective Test Case Design

- Exhaustive testing (use of all possible inputs and conditions) is impractical
  - Must use a subset of all possible test cases
  - Must have high probability of detecting faults
- Need processes that help us selecting test cases
- Effective testing - detect more faults
  - Focus attention on specific types of faults
  - Know you are testing the right thing
- Efficient testing - detect faults with less effort
  - Avoid duplication
  - Systematic techniques are measurable and repeatable

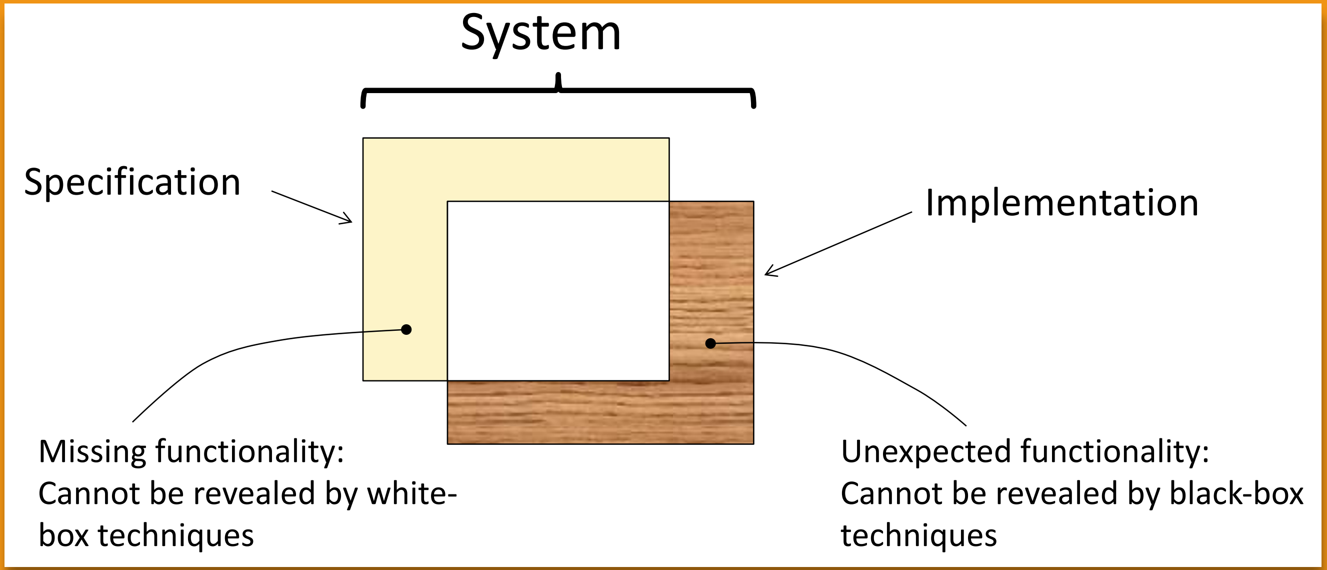
Basic Testing Strategies

- Motivation:
  - Effective Test Case Design
- Compare 2 approaches
  - Black Box
  - White Box

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

Black Box vs. White Box

- Black Box: functional testing
  - Unit test
  - Integration test
  - System test
  - Programmers & Test Engineers & Quality Assurance Engineers
- White Box: structural testing
  - Unit test
  - Integration test
  - Programmers & Test Engineers



Control Flow Testing
Kiểm thử luồng điều khiển

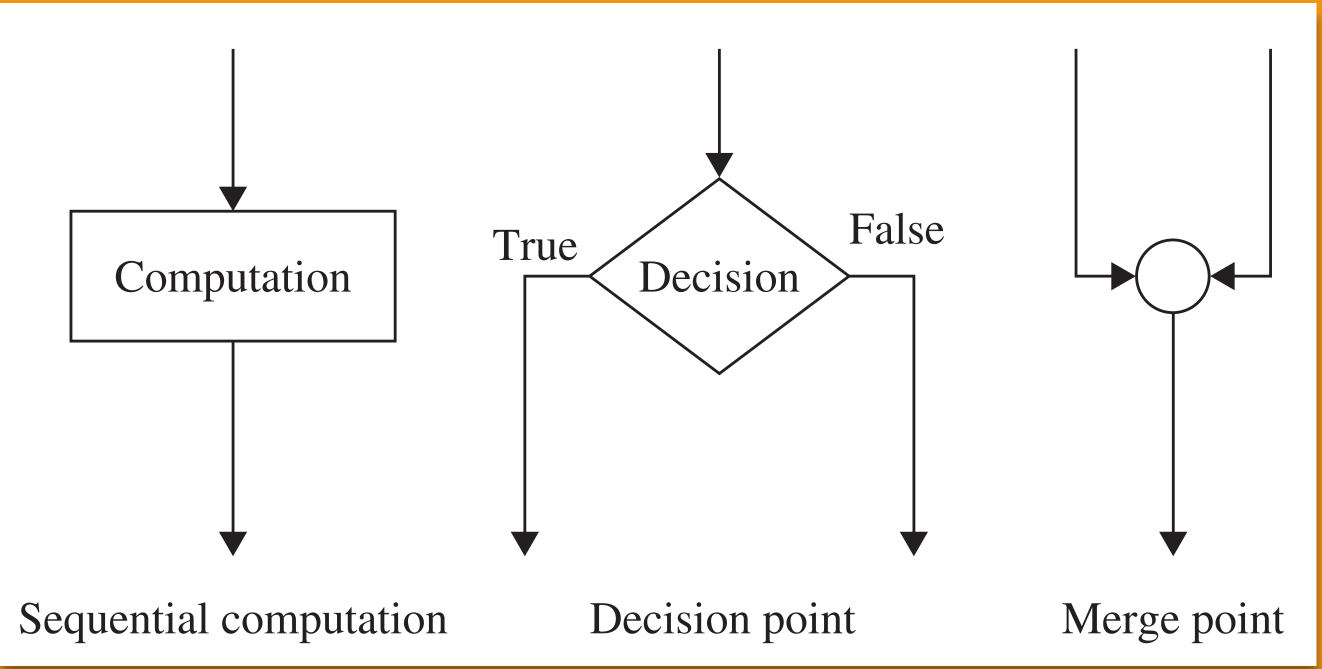
7

Basic terms

- Program unit: entry point to exit point
  - Procedures, Functions, Modules, Components
- 2 kinds of statement
  - Assigment statement
  - Conditional statement
- Program path
  - A sequence of statements from entry point to exit point of a program unit
  - A program unit may contains many program paths
  - An execution instance of the unit
- A given set of input data, the unit executes a different path

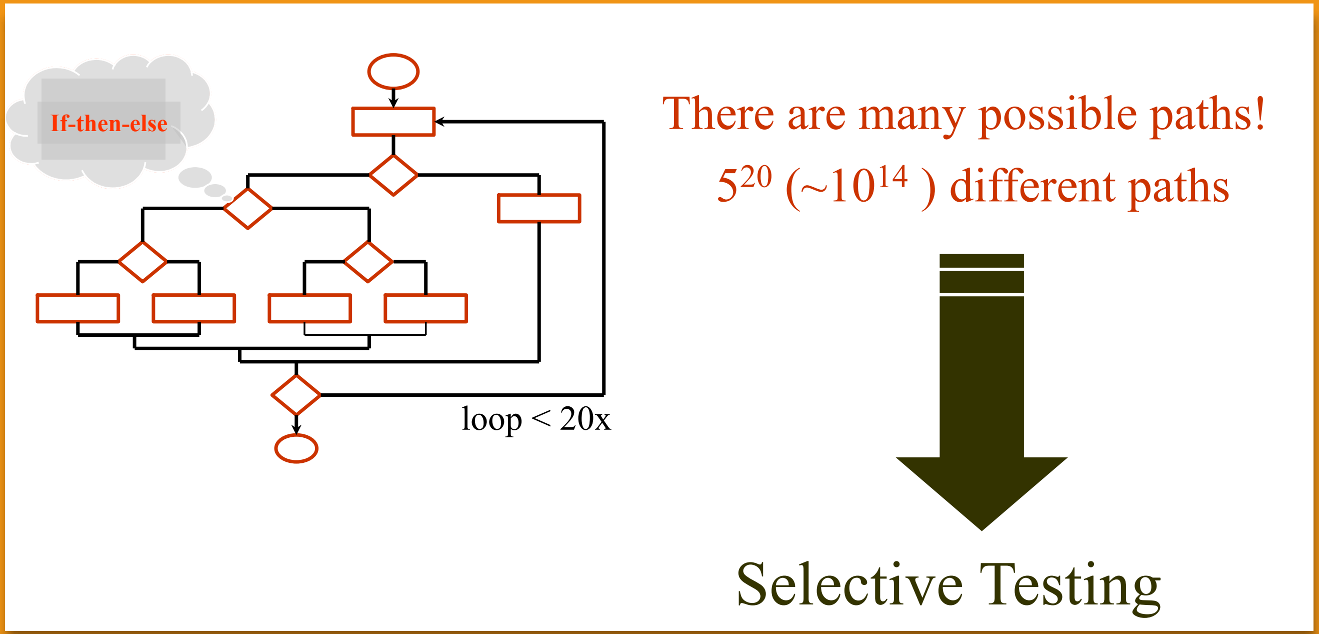
Control Flow Graph
Đồ thị luồng điều khiển

- Represent the graphical structure of a program unit
- A sequence of statements from entry point to exit point of the unit depicted using graph notions



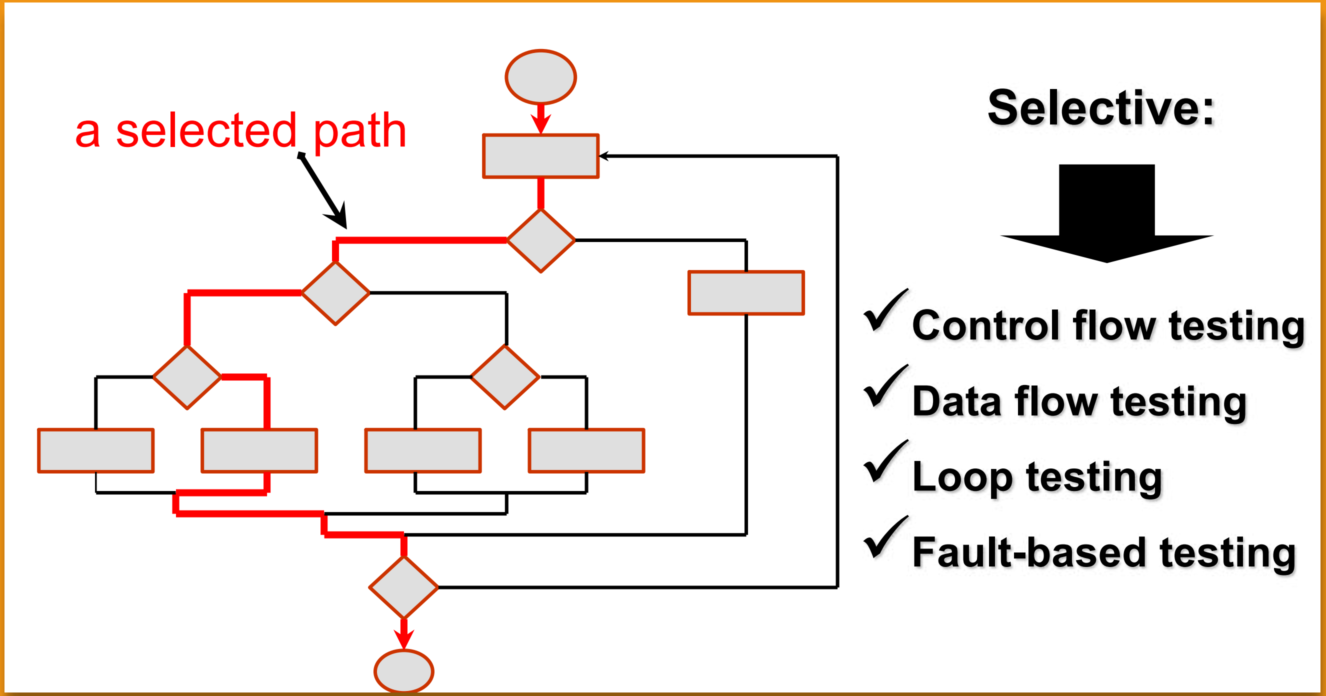
Control Flow Testing

- Main idea: select a few paths in a program unit and observe whether or not the selected paths produce the expected outcome
- Executing a few paths while trying to assess the behavior of the entire program unit



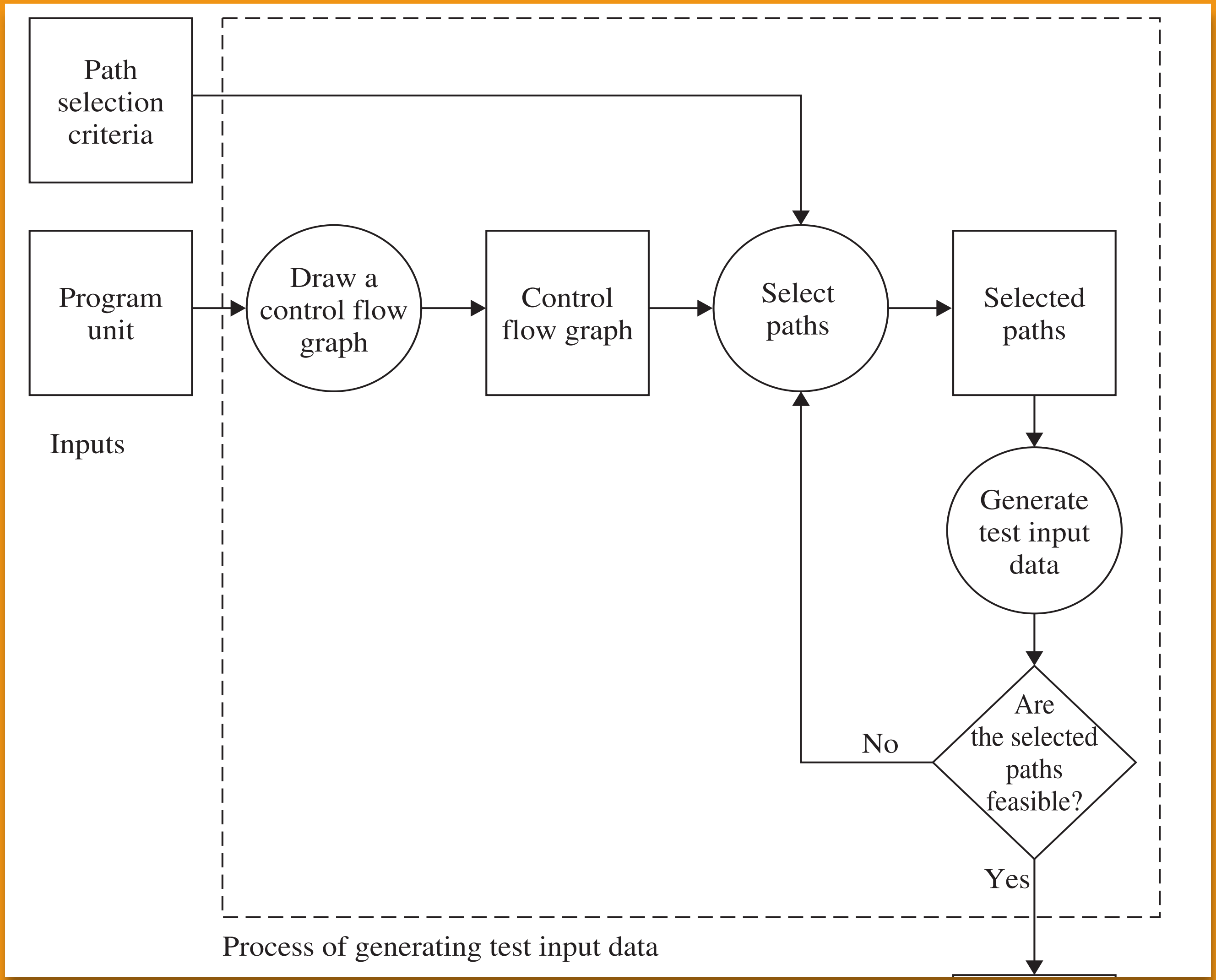
Select as few paths as possible

- Selecting relevant paths based on some criteria
  - All paths
  - Statement Coverage
  - Branch Coverage
  - Predicate Coverage
- Most applicable to new software for unit test



Outline of Control Flow Testing

- Inputs
  - Source code of unit
  - Path selection criteria
- Generate CFG: draw CFG from source code of the unit
- Selection of paths: selected paths to satisfy path selection criteria
- Generation of test input data



Path selection criteria

- Example:
  - Given the source code of the function `AccClient`
  - Draw the CFG

Life Insurance Example

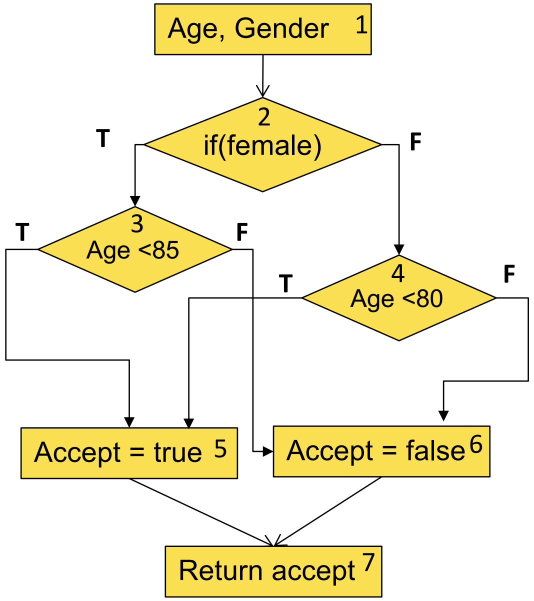
```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

Path selection criteria

- Example:
  - Given the source code of the function `AccClient`
  - Draw the CFG

Life Insurance Example

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

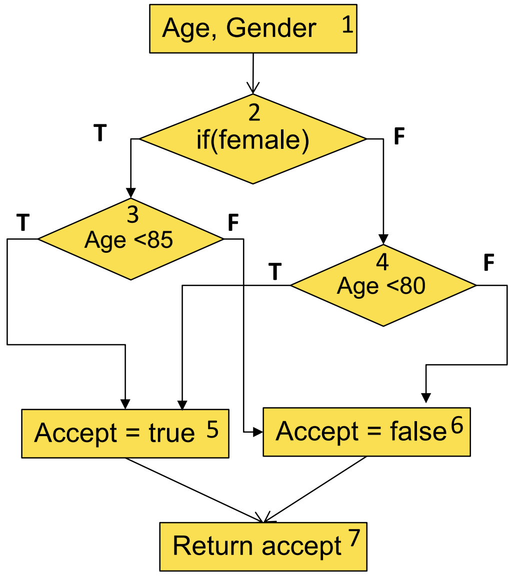


All path coverage criterion Tiêu chí lựa chọn tất cả các đường dẫn

- Objective: Design all possible test cases so that all paths of the program are executed
- 4 test cases satisfy the all path coverage criterion

All paths

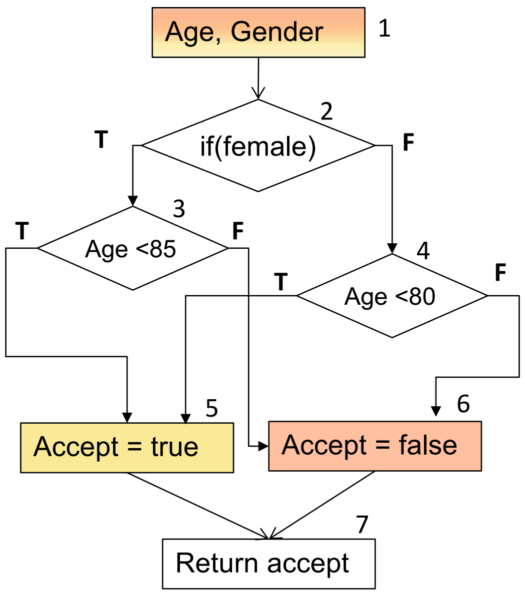
Female	Age < 85	Age < 80
Yes	Yes	Yes
Yes	Yes	No
~~Yes~~	~~No~~	~~Yes~~
Yes	No	No
No	Yes	Yes
No	Yes	No
~~No~~	~~No~~	~~Yes~~
No	No	No



Statement coverage criterion Tiêu chí bao phủ hết tập câu lệnh

- Main idea: Execute each statement at least once
- A possible concern may be:
  - dead code

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
    accept := age < 85;
   else
    accept := age < 80;
return accept
```



Disadvantages of statement coverage

- Statement coverage is the weakest, indicating the fewest number of test cases
- Bugs can easily occur in the cases that statement coverage cannot see
- The most significantly shortcoming of statement coverage is that it fails to measure whether you test simple If statements with a **false decision outcome**.

Branch coverage criterion
Tiêu chí bao phủ nhánh

- Also called Decision Coverage
- A branch is an outgoing edge from a node
  - A rectangle node has at most one out going branch
  - All diamond nodes have 2 outgoint branches
- A decision element in a program may be one of
  - If – then
  - Switch – case
  - Loop
- Main idea: selecting paths such that every branch is included in at least one path

Example

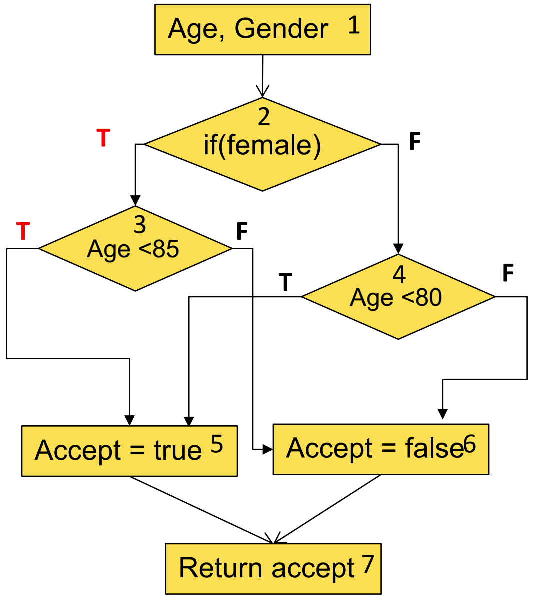
- We test the path
  - 1-2(T)-3(T)-5-7

Branch Coverage /1

AccClient(83, female)->accept

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

true



Example

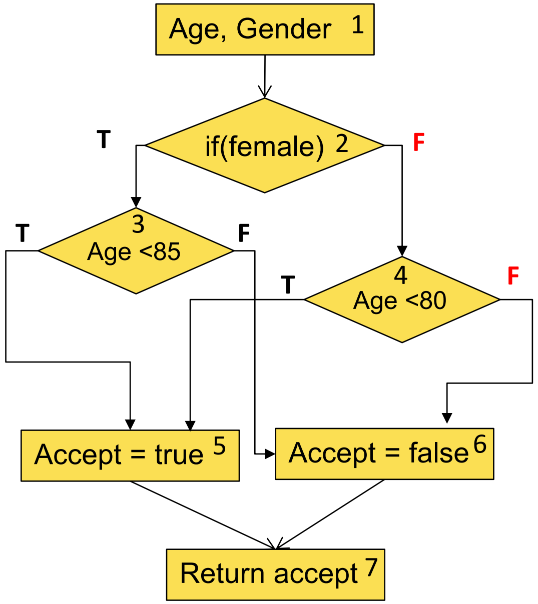
- We test the path
  - 1-2(F)-4(F)-6-7

Branch Coverage /2

AccClient(83, male)
->reject

```
bool AccClient(agetype
 age; gndrtype gender)
bool accept
 if(gender=female)
   accept := age < 85;
  else
   accept := age < 80;
return accept
```

false



Example

- We test the path
  - 1-2(F)-4(T)-5-7

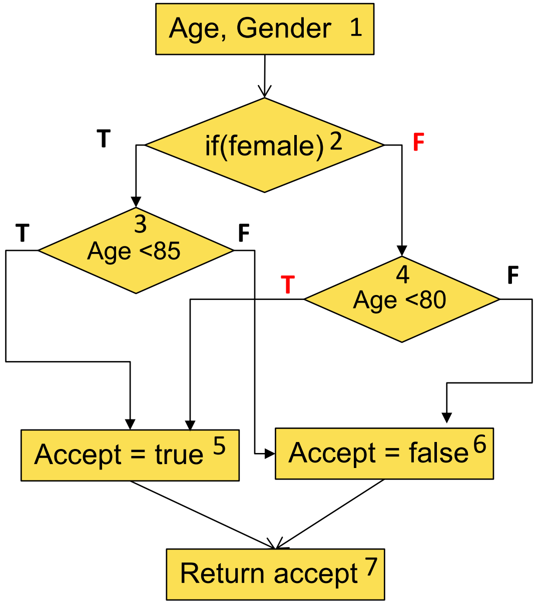
AccClient(78, male)->accept

Branch Coverage /3

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
     accept := age < 85;
   else
     accept := age < 80;
return accept
```

true

true



Example

- We test the path
  - 1-2(T)-3(F)-6-7

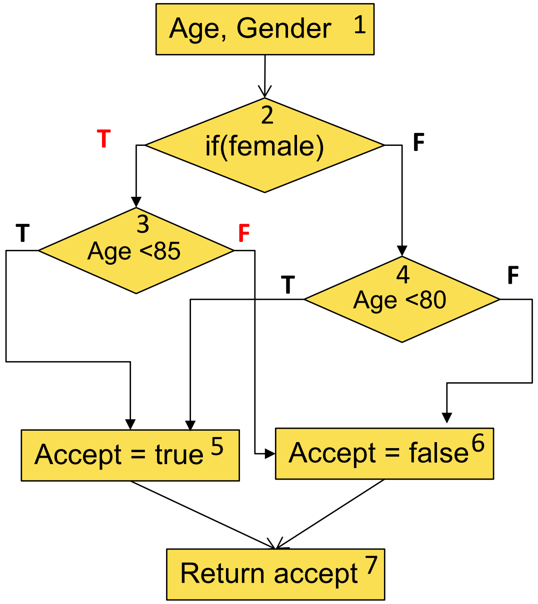
Branch Coverage /4

AccClient(88,
female) ->reject

```
bool AccClient(agetype
  age; gndrtype gender)
bool accept
  if(gender=female)
    accept := age < 85;
   else
    accept := age < 80;
return accept
```

false

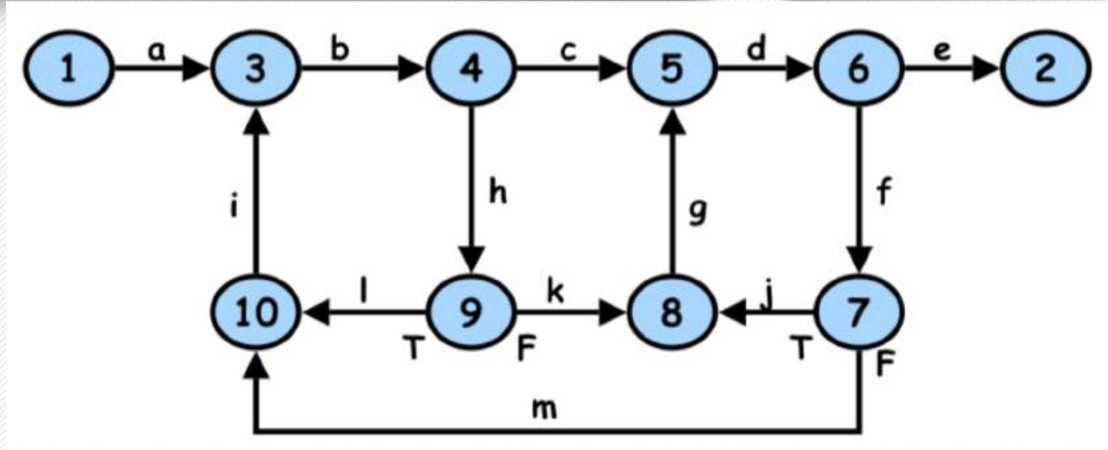
false



Comparing 3 criteria

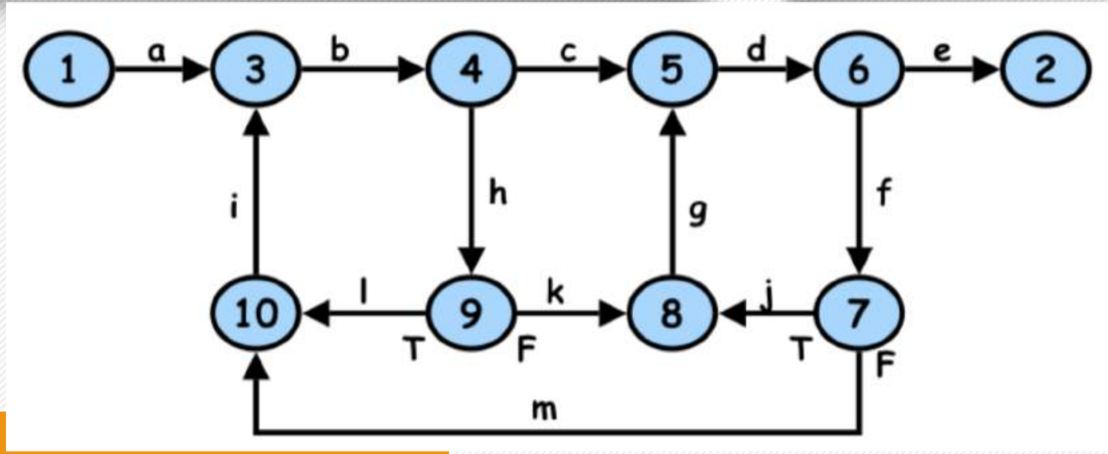
- (1) All path coverage: assure 100% paths executed
- (2) Statement coverage: pick enough paths to assure that every source statement is executed at least once
- (3) Branch coverage: assure that every branch has been exercised at least once under some test
- (1) implies (3), (3) implies (2)
- These 3 criteria are also called as **Path Testing Techniques**

Example of statement and branch coverage



Example of statement and branch coverage

- Question 1:
  - Does every decisions have a T and F values?
  - Yes ➔ Implies branch coverage
- Question 2:
  - Is every link covered at least once?
  - Yes ➔ Implies statement coverage



Path	Decisions				Process Link												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abcde	T	T			x	x	x	x	x								
abhkgde	F	T		F	x	x		x	x		x	x			x		
abhlibcde	F	T		T	x	x	x	x	x			x	x			x	
abcdfjgde	T	F	T		x	x	x	x	x	x	x			x			
abcdfmibcde	T	F	F		x	x	x	x		x			x				x

Example: Exponential Function


```
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);
```

Example: Bubble sort


```
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");
```

Limitations of path testing techniques

- Path Testing is applicable to new unit
- Limitations
  - Interface mismatches and mistakes are not taken
  - Not all initialization mistakes are caught by path testing
  - Specification mistakes are not caught

Predicate Coverage
Tiêu chí bao phủ điều kiện

29

Basic concepts

- Predicate: are expresions that can be evaluated to a boolean value (true or false)
- Predicate may contain:
  - **Boolean** variables
  - **Non-boolean** variables that are compared with the relational operators $\{\geq, \leq, =, \neq, <, >\}$
  - **Boolean function** calls
- The internal structure is created by **logical operators:**
  - $\neg, \rightarrow, \leftrightarrow, \wedge, \vee, \oplus$
- **A clause:** is a predicate that does not contain any of the logic operator. Example: $(a=b) \vee C \wedge p(x)$ contains 3 clauses

How to create a Path Predicate Expression?

- Write down the predicates for the decisions you meet along a path
- The result is a set of path predicate expressions
- All of these expressions must be satisfied to achieve a selected path

```
X1,X2,X3,X4,X5,X6
 if (X5 > 0 || X6 < 0)              /* predicates A,B */
 ...
 if(X1 + 3 * X2 + 17 >= 0)          /* predicate C */
 ...
 if(X3 == 17)                       /* predicate D */
 ...
 if(X4 - X1 >= 14 * X2)             /* predicate E */
```

Predicate Faults
Các lỗi điều kiện có thể xảy ra

- An incorrect Boolean operator is used
- An incorrect Boolean variable is used
- Missing or extra Boolean variables
- An incorrect relational operator is used
- **Parentheses** are used incorrectly

Predicate Coverage

- To ensure all predicates in the source code are implemented correctly
- Main idea:
  - For each predicate p, test cases must assure that p evaluates to true at least once and p evaluates to false at least once

Example: $p = ((a > b) \vee C) \wedge p(x))$

	a	b	C	p(x)
1	5	4	true	true
2	5	6	false	false

"decision coverage" in literature

Example

$$p = ((a < b) \lor D) \land (m >= n*o)$$

Clause Coverage

- Main idea: For each individual condition (clause) c, c should evaluate to true and c should evaluate to false at least once.

Example: $((a > b) \vee C) \wedge p(x))$

	a	b	C	p(x)
1	5	4	true	true
2	5	6	false	false

"condition coverage" in literature

Example

$$P = ((a < b) \vee D) \wedge (m >= n*o)$$

Predicate vs. Clause Coverage

- Does Predicate coverage subsume clause coverage?
- Does clause coverage subsume predicate coverage?
- Naturally, we want to test both the predicate and individual clauses

Example: $p = a \vee b$

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Predicate and Clause Coverage

- PC does not **fully** exercise all the clauses
  - For example: p = a v b. In most programming languages, at run time, when a evaluates to True, p evaluates to True and **b does not exercise**.
- CC does not always ensure PC
- This is, we can satisfy CC without causing the prediate to be both true and false
- This is definitively not what we want
  - We need to come up with another approach
- Inversely, with PC, it is not always that all individual clauses evaluate to both true and false

Combinatorial Coverage
Bao phủ kết hợp

- Main idea: For each predicate p, test cases should ensure that the clauses in p should evaluate to each possible combination of truth values

Example: $(a \vee b) \wedge c$

CoC requires every possible combination

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Example 1

- Write all the clauses and the Combination of Clauses (CoC) of the given predicate $P = ((a > b) \vee C) \wedge p(x)$

What is the problem with combinatorial coverage?

- Combinatorial coverage is very expensive if we have multiple clauses in the predicate
- For each decision point with N conditions, we need 2^N test cases

Motivation

- Predicate should evaluate to both true and false in test cases
- Each individual condition (clause) is tested independently of each other
- Each individual clause should affect the predicate
  - i.e., changing the value of an individual clause, the value of predicate is also changed

Active clause

- Major clause: the clause which is being focused upon
- Minor clause: all other clauses in the predicate
- Determination: clause c in p determines p

A clause c_i in predicate p, called the major clause, determines p if and only if the values of the remaining minor clauses c_j are such that changing c_i changes the value of p

<u>$P = A \vee B$</u>

if B = true, P is always true.

so if B = false, A determines P.

if A = false, B determines P.

<u>$P = A \wedge B$</u>

if B = false, P is always false.

so if B = true, A determines P.

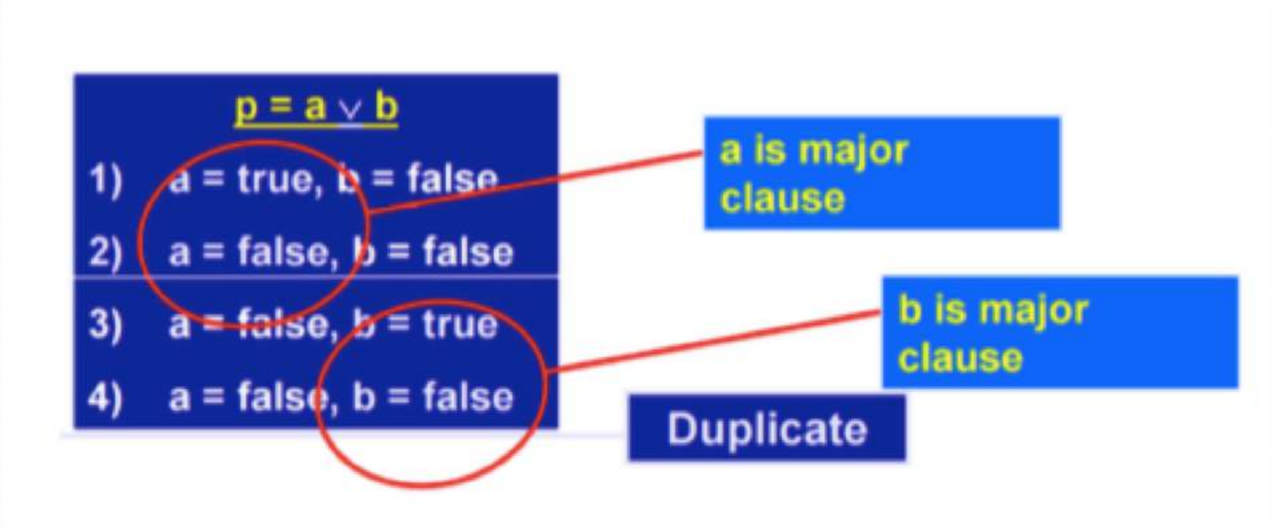
if A = true, B determines P.

Example: P = a Λ (b V c)

- Identify, when considering a as active clause, which values should be assigned to b and c?
- Answer:
  - a is active clause, the values of b and c are those such that changing the value of a changes the value of P.
  - So, (b v c) = 1, the value of P is determined by the value of a. We can assign b = 1, c = 1
- Test case T = {(a=T, b=T), (a=F,b=F)} satisfy GACC, but not PC, all both cases, P = T

Active Clause Coverage (ACC)

- For each predicate **p** and each major clause **c** of **p**, choose minor clauses so that **c** determines **p**. 2 test requirements for assuring ACC: for each **c**: **c** evaluates to **true** and **c** evaluates to **false**
- **With a decision point consists of N conditions, only N+1 test cases are required. Why?**

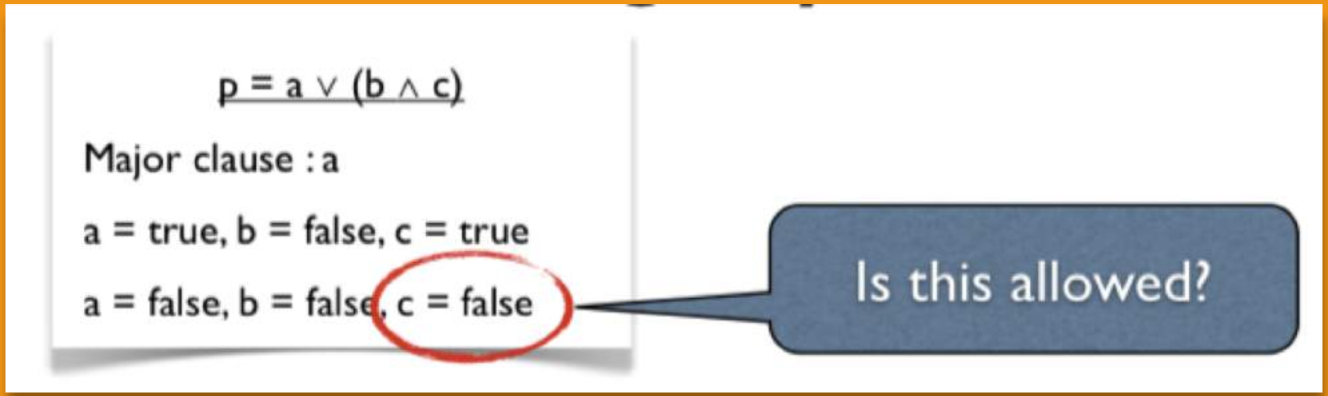


Example


```
public static void printHonorRollStatus(double cummulativeGPA,
        double termGPA, int creditsCompleted, boolean fullTimeStatus) {
    // Determine if the student is on the deans list.
    if ((creditsCompleted > 30) && (cummulativeGPA > 3.20)
            && (fullTimeStatus == true) && (termGPA > 2.0)) {
        System.out.println("You are on the dean's list.");
    } else if ((creditsCompleted > 30) && (cummulativeGPA > 3.70)
            && (fullTimeStatus == true) && (termGPA > 2.0)) {
        System.out.println("You are on the high honors dean's list.");
    } else if ((creditsCompleted > 30) && (cummulativeGPA > 2.0)
            && (fullTimeStatus == true) && (termGPA > 3.2)) {
        System.out.println("You are on the honor list.");
    } else {
```

Resolving the Ambiguity problem
Giải quyết vấn đề nhập nhằng

- With the example **p = a v (b ∧ c), a is active clause**
- How we can choose the value of b and c? c should be different from b or not?



General Active Clause Coverage (GACC)

- For each clause c, choose minor clauses' values such that c determines p
- Clause c has to evaluate to both true and false
- Minor clauses don't need to be the same when clause c is true as well as when c is false.
- $c_j(c_i\text{=true})=c_j\ (c_i = \text{false})$ for all c_j OR $c_j(c_i\text{=true})!=c_j\ (c_i = \text{false})$ for all c_j

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Does GACC subsume predicate coverage? NOT always, ex: P = A <-> B

GACC and subsumption of CC/PC

- By definition: GACC subsumes CC, that is, satisfying GACC leads to satisfy CC, but not PC
- Example: P = a <->b.
  - Test case T = {(a=T, b=T), (a=F,b=F)} satisfy GACC, but not PC, all both cases, P = T
  - Test case T = {(a=T, b=F), (a=F,b=T)} satisfy GACC too, but not PC either, all both cases, P = F

Correlated Active Clause Coverage (CACC)

- For each clause c_i in P, choose minor clauses c_j such that c_i determines the predicate P
- Clause c_i has to evaluate to true or false in test cases
- The values chosen for the minor clauses c_j must cause P to be true for one value of the major clause c_i and false for the other value of c_i
- $P(c_i = true) != P(c_i = false)$
- Minor clauses don't need to be the same. That is:
    - $c_j(c_i{=}true){=}c_j\,(c_i = false)$ for all c_j **OR** $c_j(c_i{=}true){!=}c_j\,(c_i = false)$ for all c_j

Example

- When a is active clause, which rows can be chosen to satisfy CACC?
  - When a is active, values of b and c are chosen such that P(a=true) != P(a=false)
  - Any of rows 1, 2, 3 and any of row 5,6,7 - one of total nine pairs satisfies CACC

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

CACC and subsumption of GACC/CC/PC

- By definition, CACC subsumes GACC thus CC and also PC
- Example: P = a<->b
  - Which test cases satisfy PC?
  - Which test cases satisfy CACC?

Example: $P = a \wedge (b \leftrightarrow c)$

- Which test cases satisfy GACC?
- Which test cases satisfy CACC?

t	a	b	c	$p = a \wedge (b \leftrightarrow c)$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Restricted Active Clause Coverage (RACC)

- For each predicate P and each active clause c_i in P, choose minor clause c_j so that c_i determines P. c_i evaluates to both true and false in test cases
- Predicate P evaluates to true in one case of major clause and false in the other (that is CACC)
- The values choosen for minor clauses c_j must be the same when c_i evaluates to true as when c_i evaluates to false.
- That is, $c_j(c_i\text{=true})=c_j\ (c_i = \text{false})$ for all c_j
- This is the stricter version of CACC

RACC and subsumptions of CACC/GACC/PC/CC

- By definition, RACC subsumes CACC, thus subsumes GACC, CC and PC
- RACC often leads to infeasible test requirements
- Example: Consider the predicate P = ((a > b) && (b < c)) || (c>a) = (X && Y)|| Z
  - When Z is active clause, we choose X = 1, Y = 0. Infeasible test case: Z = 0, X = 1, Y = 0

Logical Operators in Source Code

- & | ~ ^ correspond also to logical operators.
- What is the difference between (a&&b) and (a&b)? (a||b) and (a|b)?

logical expression	Java expression
$a \wedge b$	a && b
$a \vee b$	a || b
$\neg a$	! a
$a \rightarrow b$	a ? b : true;
$a \leftrightarrow b$	a == b
$a \oplus b$	a != b

Code transformation


```
if (a&&b)
  S1;
else
  S2;
```

```
if (a){
  if (b)
    S1;
  else
    S2;
} else
  S2;
```

Exercise 1

- Identify test cases for
  - PC
  - CC
  - CoC
  - GACC
  - CACC
  - RACC

```
public static boolean isLeapYear(int y) {
  return y % 400 == 0 || (y % 4 == 0 && y % 100 != 0);
}
```

```
a: y % 400 == 0     b: y % 4 == 0    c: y % 100 != 0
p: a ∨ (b ∧ c)
```

Exercise 2

- Determine predicates and clauses in the source code
- Identify reachable predicates of the source code
- Identify test cases that satisfy PC, CC and CoC
- Are there infeasible requirements ?

```
public static int daysInMonth(int m, int y) {
  if (m <= 0 || m > 12)
    throw new IllegalArgumentException("Invalid month: " + m);
  if (m == 2) {
    if (y % 400 == 0 || (y % 4 == 0 && y % 100 != 0))
      return 29;
    else
      return 28;
  }
  if (m <= 7) {
    if (m % 2 == 1)
      return 31;
    return 30;
  }
  if (m % 2 == 0)
    return 31;
  return 30;
}
```

- Predicates and clauses
  - **p1:** c1 || c2 ; **c1:** m <= 0, **c2:** m > 12
  - **p2:** c3; **c3:** m == 2;
  - **p3:** **c4** || (**c5** && **c6**) ; **c4:** y % 400 == 0, **c5:** y % 4 == 0, **c6:** y % 100 != 0;
  - **p4:** c7; **c7:** m <= 7;
  - **p5:** c8; **c8:** m % 2 == 1;
  - **p6:** c9; **c9:** m % 2 == 0

Exercise 2

- Determine predicates and clauses in the source code
- Identify reachable predicates of the source code
- Identify test cases that satisfy PC, CC and CoC
- Are there infeasible requirements ?

```java
public static int daysInMonth(int m, int y) {
  if (m <= 0 || m > 12) // p1
    throw new IllegalArgumentException("Invalid month: " + m);
  if (m == 2) { // p2
    if (y % 400 == 0 || (y % 4 == 0 && y % 100 != 0)) // p3
      return 29;
    else
      return 28;
  }
  if (m <= 7) { // p4
    if (m % 2 == 1) // p5
      return 31;
    return 30;
  }
  if (m % 2 == 0) // p6
    return 31;
  return 30;
}
```

- Reachability predicates - **r(p)**
  - **r(p1)** = true (always reached)
  - **r(p2)** = ¬ p1 = m >= 0 ∧ m < 12
  - **r(p3)** = r(p2) ∧ p2 = (m > 0 ∧ m < 12 ∧ m == 2) = (m == 2)
  - **r(p4)** = r(p2) ∧ ¬ p2
  - **r(p5)** = r(p4) ∧ p4 = (m > 0 ∧ m < 12 ∧ m != 2) ∧ m <= 7
  - **r(p6)** = r(p4) ∧ ¬ p4 = (m > 0 ∧ m < 12 ∧ m != 2) ∧ m > 7

Exercise 3

```
public static TClass triangleType(int a, int b, int c) {
    if (a <= 0 || b <= 0 || c <= 0)  /* p1 */
      return INVALID;
    if (a >= b + c || b >= a+c || c >= a+b) /* p2 */
        return INVALID;
    int count = 0;
    if (a == b) /* p3 */
      count++;
    if (a == c) /* p4 */
      count++;
    if (b == c) /* p5 */
      count++;
    if (count == 0) /* p6 */
      return SCALENE;
    if (count == 1) /* p7 */
      return ISOSCELES;
    return EQUILATERAL;
  }
```

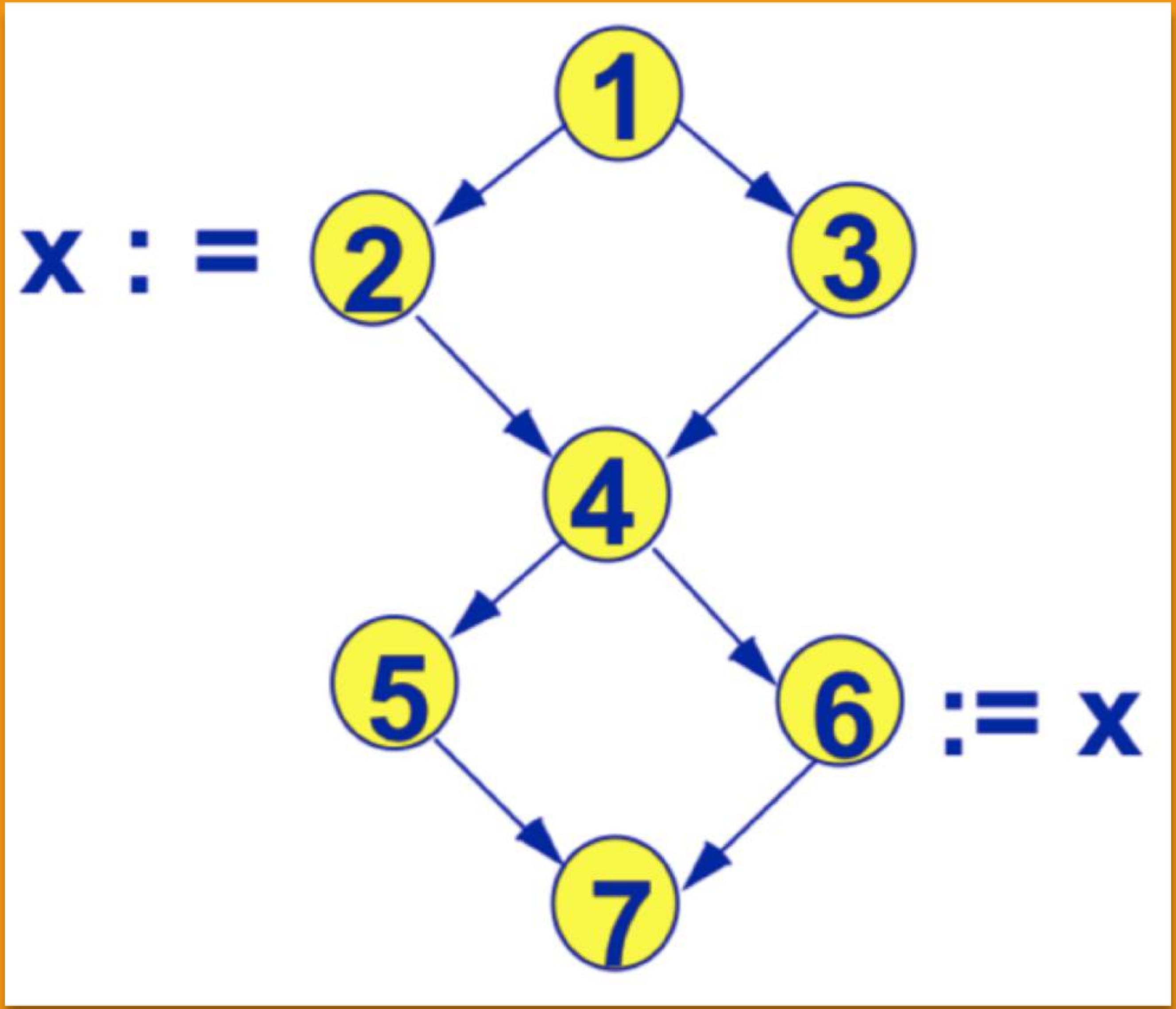
- Identify:
  - 1) the reachability predicates;
  - 2) TR(CC) and TR(PC)
  - 3) test cases that satisfy a) PC b) CC
  - 4) determination predicates for the clauses of p1 and p2
  - 5) TR(CACC)
  - 6) test cases that satisfy a) CACC b) RACC [are there infeasible requirements?]

Data Flow Testing
Kiểm thử luồng dữ liệu

62

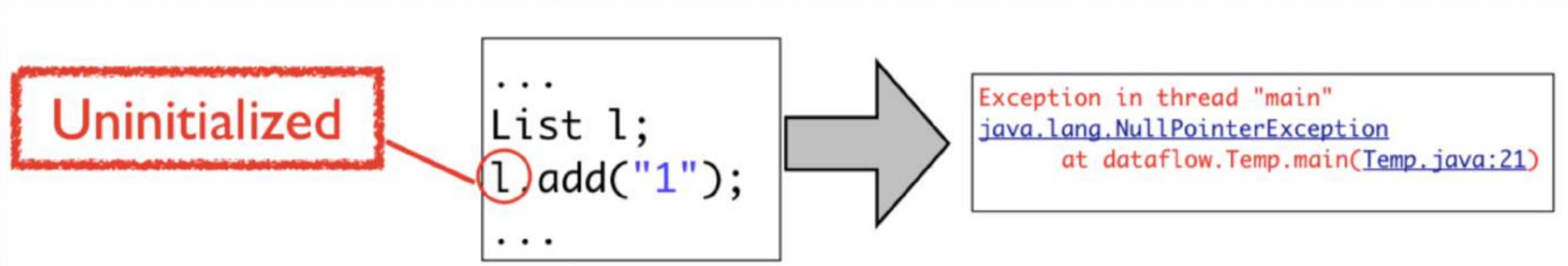
Motivation

- S2: define x, assign to x a value
- S6: use x to assign to other variable
- If Branch Coverage
  - 1-2-4-5-7
  - 1-3-4-6-7
- We cannot explore simultaneously the relationship between definition of x in statement 2 and the use of x in statement 6



Motivation

- GOAL: Try to ensure that values are computed and used correctly
- Consider: how data gets accessed and modified in the system and how it can get corrupted
- Common access-related bugs:
  - Using an undefined or uninitialized variable
  - Deallocating or reinitializing a variable before it is constructed, initialized or used
  - Deleting a collection object leaving its members unaccessible



Variable Definition
Định nghĩa biến

- A program variable is **DEFINED** whenever its value is modified
  - on the left hand side of an assigment statement, e.g., y = 20
  - 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
Sử dụng biến

- A program variable is **USED** whenever its value is read:
  - On the right hand side of an assignment statement, e.g., y = x + 17, x is used, y is defined
  - 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 code below, there is a p-use of x and a c-use of y

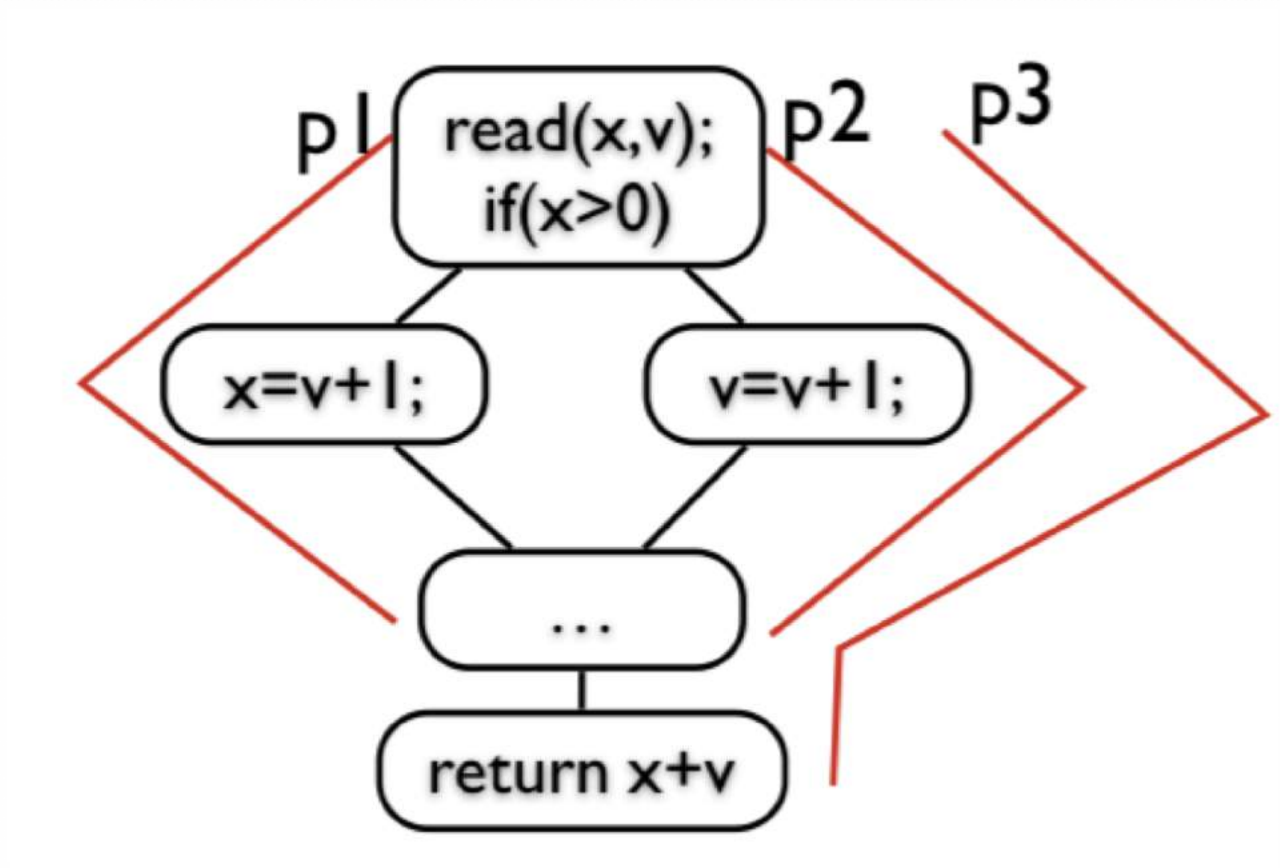
```
if (x > 0) {
    print(y);
}
```

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)**

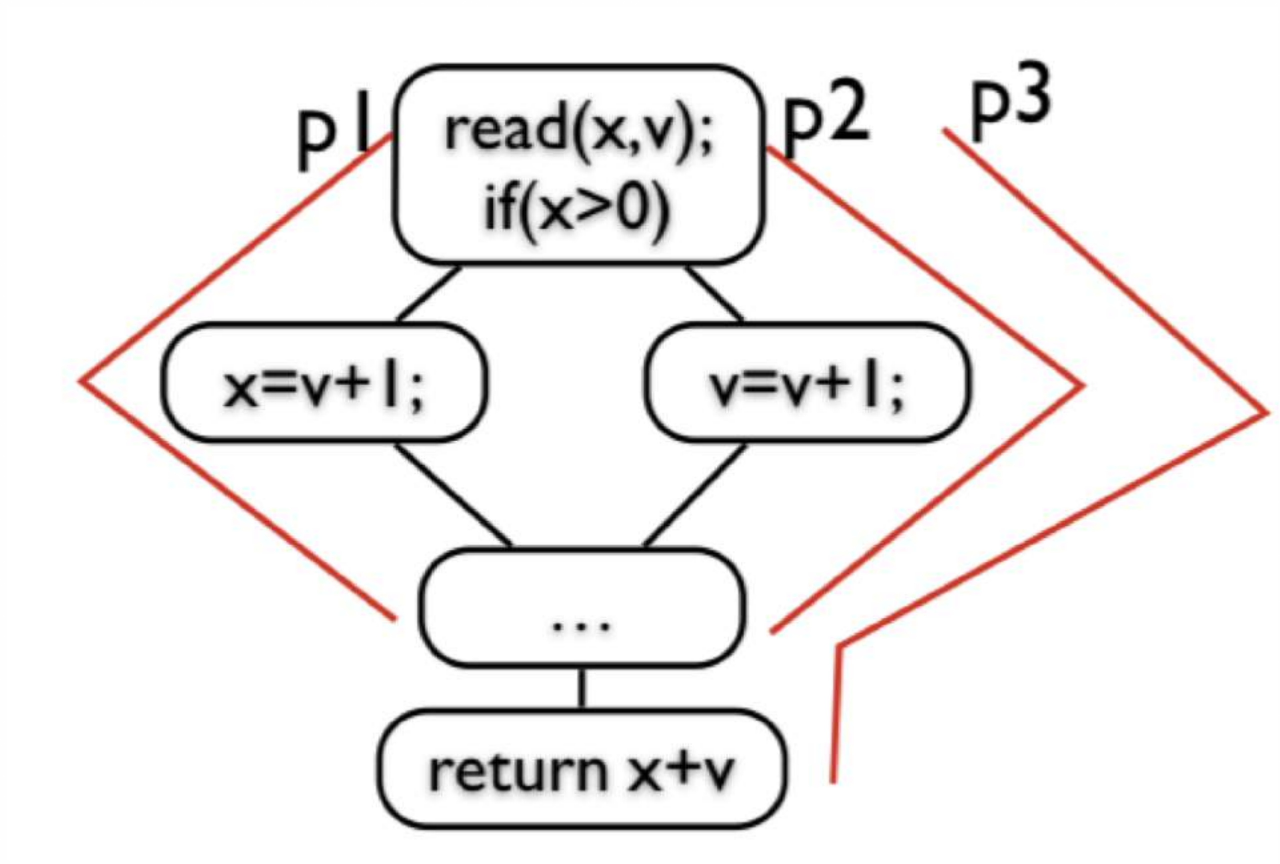
Terminology
Các thuật ngữ khác

- *Definition-Clear-Path*
  - *A path is <u>definition-clear</u> ("**def-clear**") with respect to a variable **v** if it has no variable re-definition of **v** on this path*
  - p1 is def-clear path of v while p2 is not



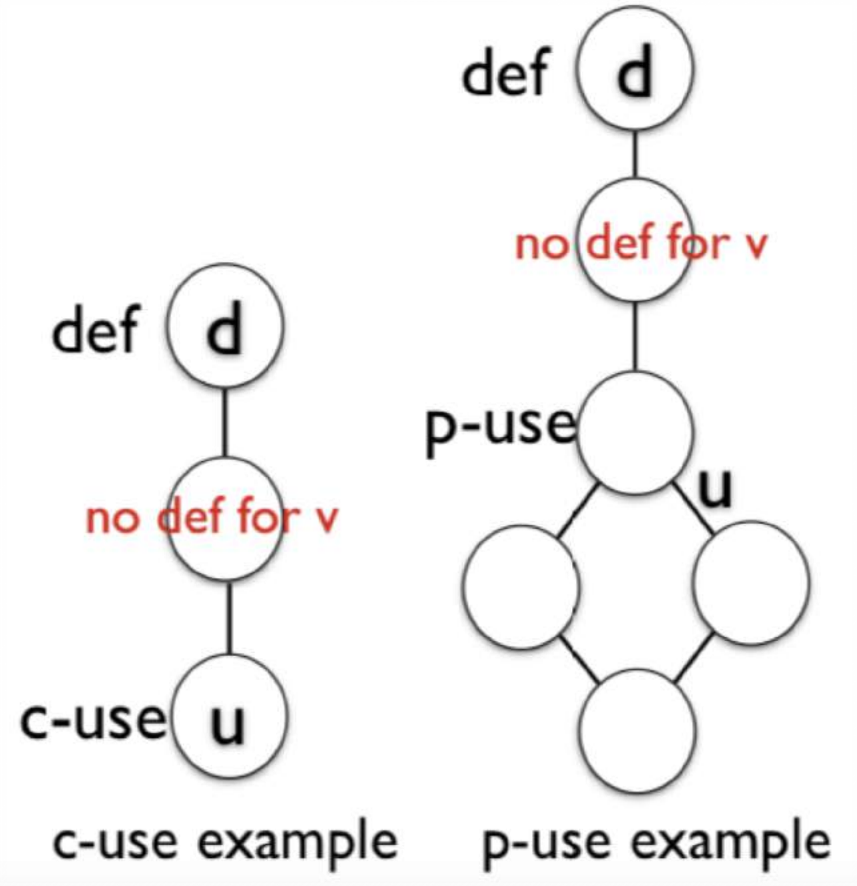
Terminology
Các thuật ngữ khác

- *Complete-Path*
  - *A path is <u>complete</u> if the initial node of this path is a entry node and the final node of this path is an exit node*
  - p1, p2 are not complete while p3 is



Definition-Use Pair (DU-pair)
Cặp định nghĩa – sử dụng

- 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 out-going edge of the predicate statement
  - There is a ***def-clear*** path with respect to ***v*** from ***d*** to ***u***



Example 2


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

Data Flow Test Coverage Criteria
Các tiêu chí kiểm thử bao phủ luồng dữ liệu

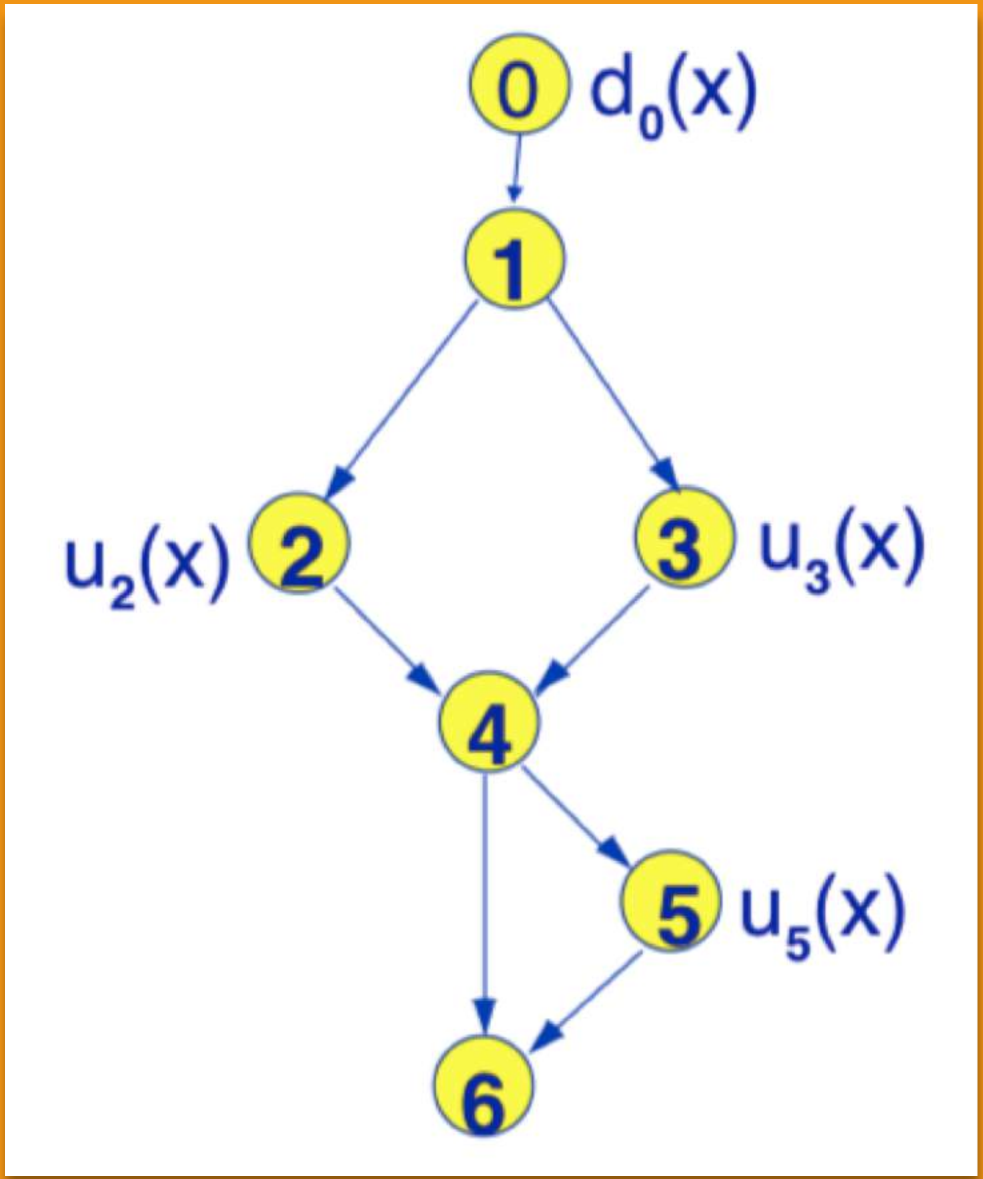
- *All-defs coverage* – Bao phủ tất cả các điểm định nghĩa dữ liệu trên đồ thị
- *All-uses coverage* – Bao phủ tất cả các điểm sử dụng dữ liệu trên đồ thị
- *All-DU-Paths coverage* – Bao phủ tất cả các đường dẫn DU trên đồ thị
- *All-p-uses/Some-c-uses coverage* – Bao phủ tất cả các điểm sử dụng dữ liệu trong các câu lệnh rẽ nhánh và một vài điểm sử dụng dữ liệu
- *All-c-uses/Some-p-uses coverage* – Bao phủ tất cả các điểm sử dụng dữ liệu và một vài điểm sử dụng dữ liệu trong các câu lệnh rẽ nhánh
- *All-p-uses coverage* – Bao phủ tất cả các loại sử dụng dữ liệu trong điều kiện
- *All-c-uses coverage* – Bao phủ tất cả các loại sử dụng thông thường của dữ liệu

All-Defs Coverage
Bao phủ tất cả các điểm định nghĩa

- For every program variable ***v***, at least <u>one</u> ***def-clear path*** from every definition of ***v*** to at least one ***c-use*** or one ***p-use*** of ***v*** must be covered

Example 1

- All Defs requires
  - $d_0(x)$ to a use
- Satisfactory Path:
  - <0-1-2-4-6>



Example 2

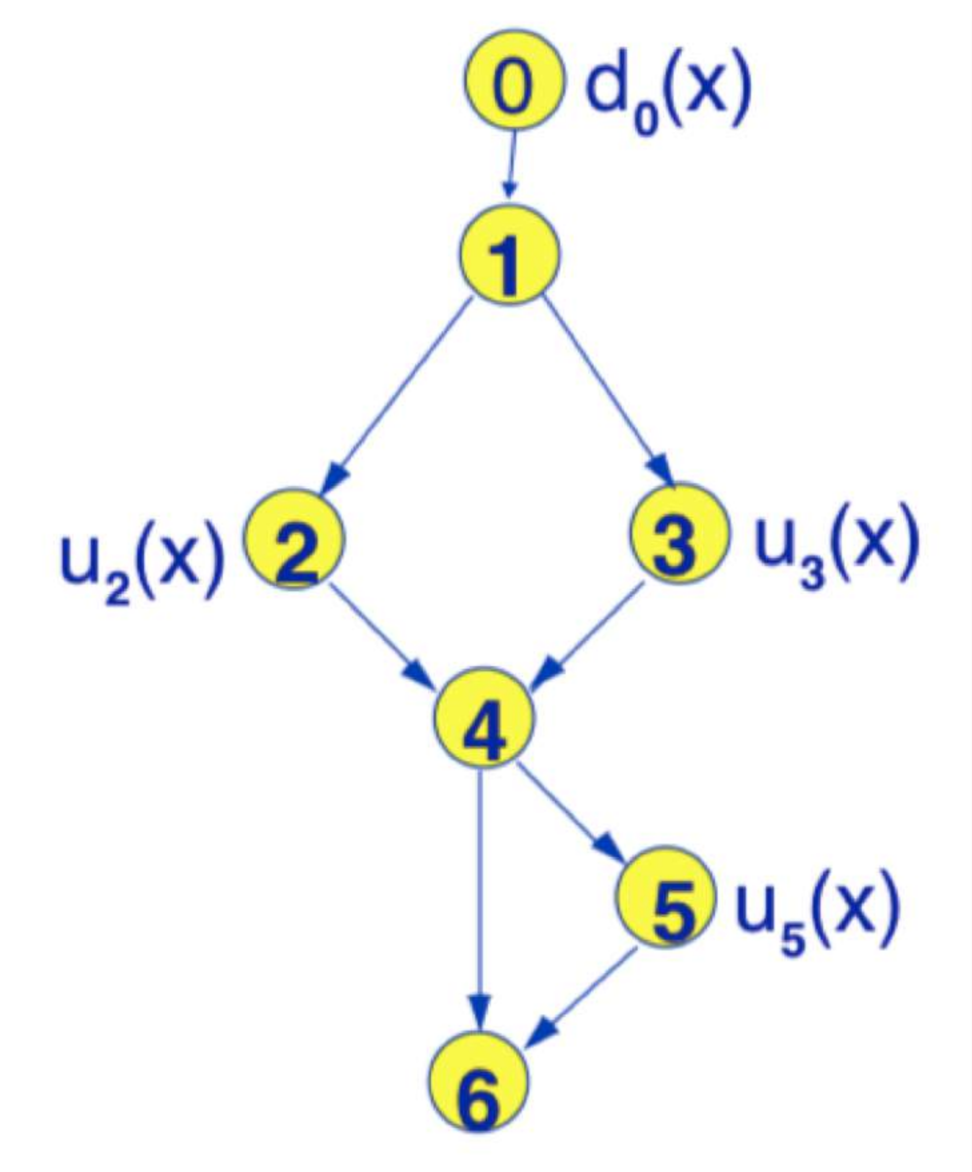
- Identify all test cases to satisfy All-defs
- Two variables: A and B
- Consider a test case executing path
  - t1: <1,2,3,4,5>
- t1 satisfy All-defs or not?

All-Uses Coverage
Bao phủ tất cả các điểm sử dụng

- 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

Example 1

- All-Uses requires
  - a: $d_o(x)$ to a u2(x)
  - b: $d_o(x)$ to $u_3(x)$
  - c: $d_o(x)$ to $u_5(x)$
- Satisfactory Path:
  - <0-1-2-4-5-6> satisfies a, c
  - <0-1-3-4-6> satisfies b



Example 2

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

Example 3

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

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

Example: Simple and Loop-Free Paths

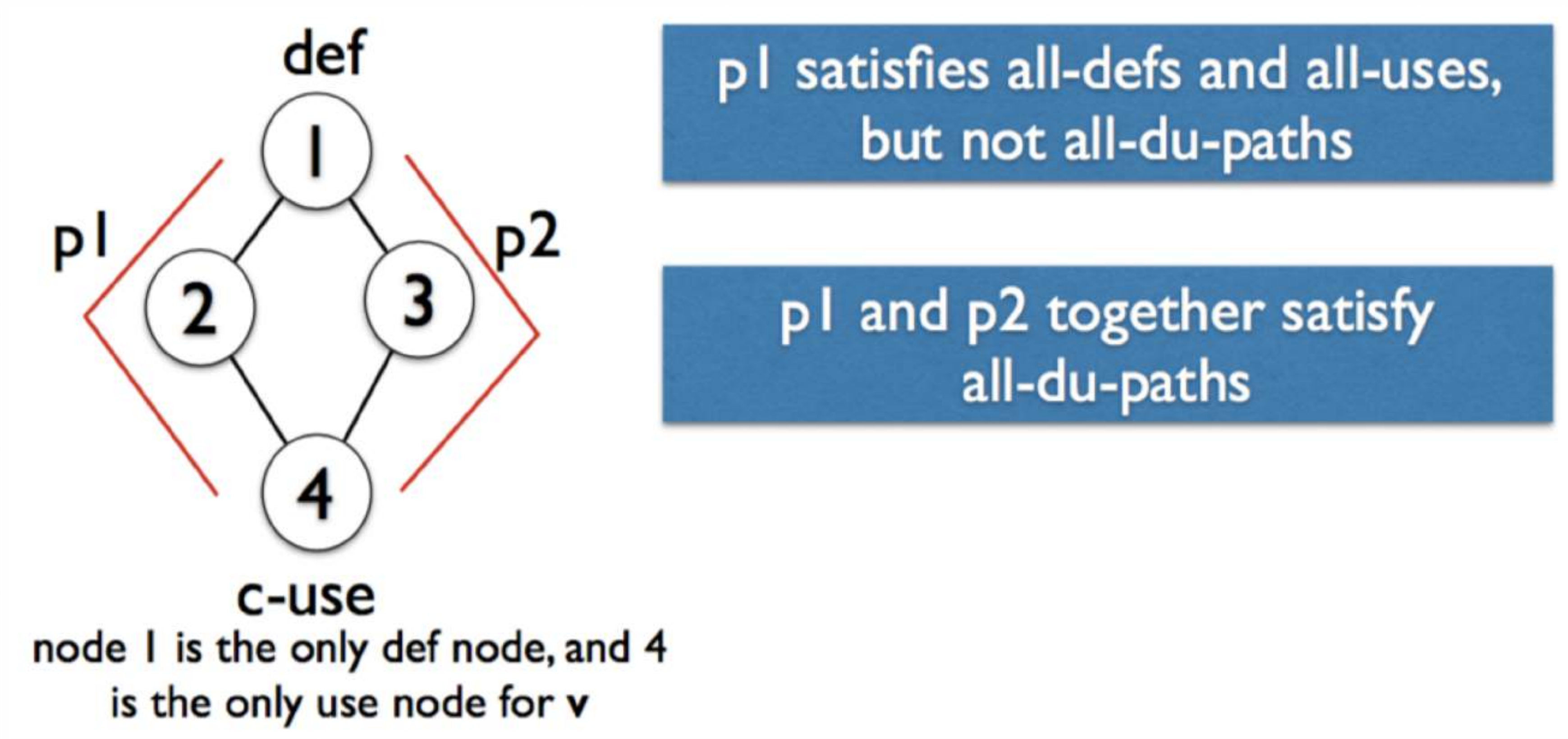
path	Simple?	Loop-free?
<1,3,4,2>		
<1,2,3,2>		
<1,2,3,1,2>		
<1,2,3,2,4>		

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

All DU-Paths Coverage

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



Example 2: All DU-paths coverage

- Identify all DU-Paths for variable A and B

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

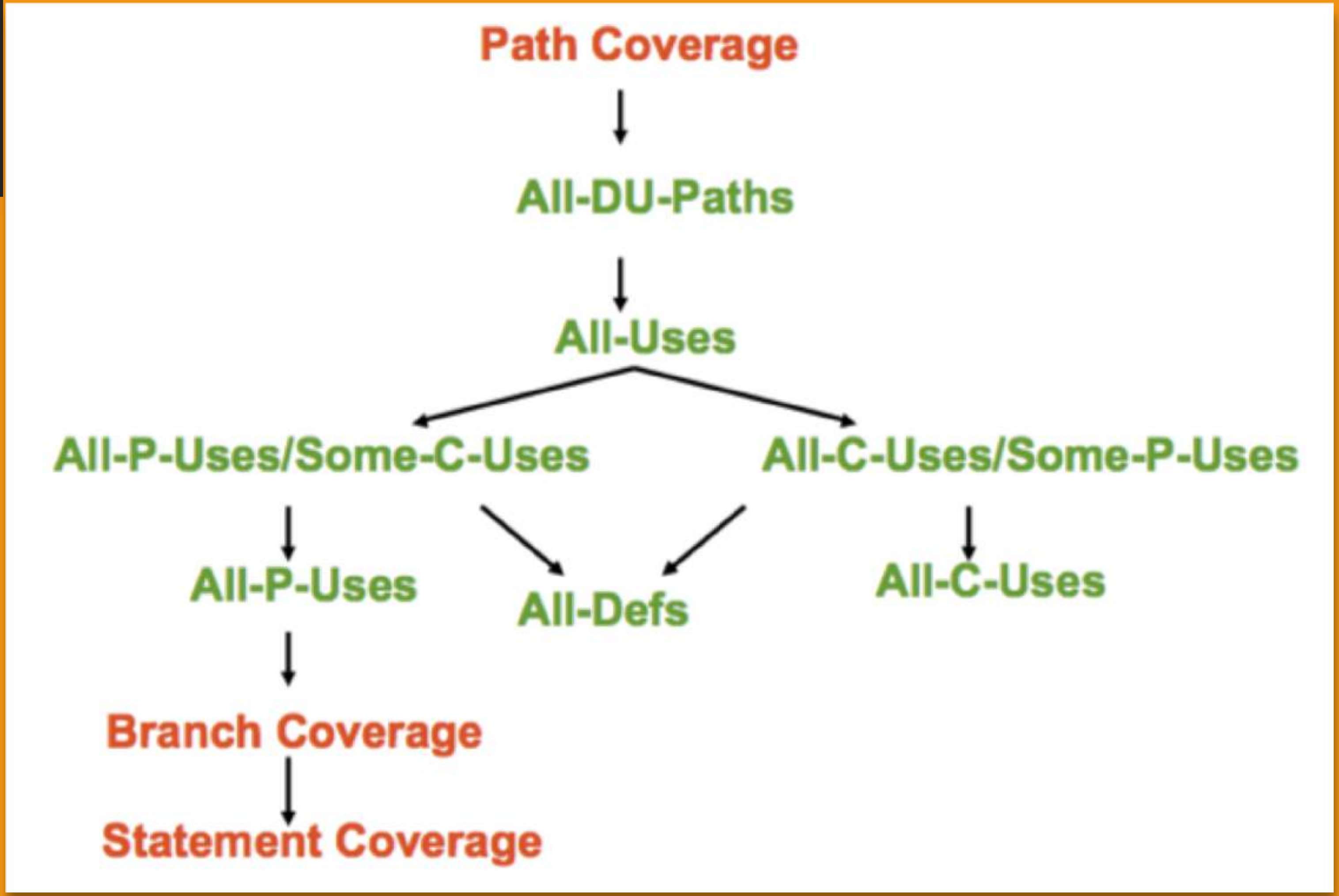
Example 3: All DU-paths coverage

- Identify all DU-Paths for variable X and Y

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

Data Flow Testing Summary

- Relationship between Data Flow Testing and Path Testing
- Path testing with **Branch Coverage** and Data-flow testing with **All-uses** is a very good combination in practice



Exercise 1

- Draw a data flow graph for the ***binsearch*** function
- Find a set of complete paths satisfying all-defs criterion with respect to variable ***mid***
- Do the same for variable ***high***

```java
public class BinarySearch {
    public int binsearch(int x, int[] V, int n) {
        int low, high, mid;
        low = 0;
        high = n - 1;
        while (low <= high) {
            mid = (low + high) / 2;
            if (x < V[mid])
                high = mid - 1;
            else if (x > V[mid])
                low = mid + 1;
            else
                return mid;
        }
        return -1;
    }
}
```

Exercise 2

- Using **All-DU paths** to test the **pow** function
- Using **All-Uses** combining with Branch Coverage to test this function

```java
public class Pow {
    public void pow(int x, int y) {
        float z;
        int p;
        if (y < 0)
            p = 0 - y;
        else p = y;
        z = 1.0f;
        while (p!=0)
        {
            z = z * x;
            p = p - 1;
        }
        if (y < 0)
            z = 1.0f/z;
        System.out.println(z);
    }
}
```