



中山大學  
SUN YAT-SEN UNIVERSITY

# Part II [Problem]

## 2. Test Adequacy (Whitebox)



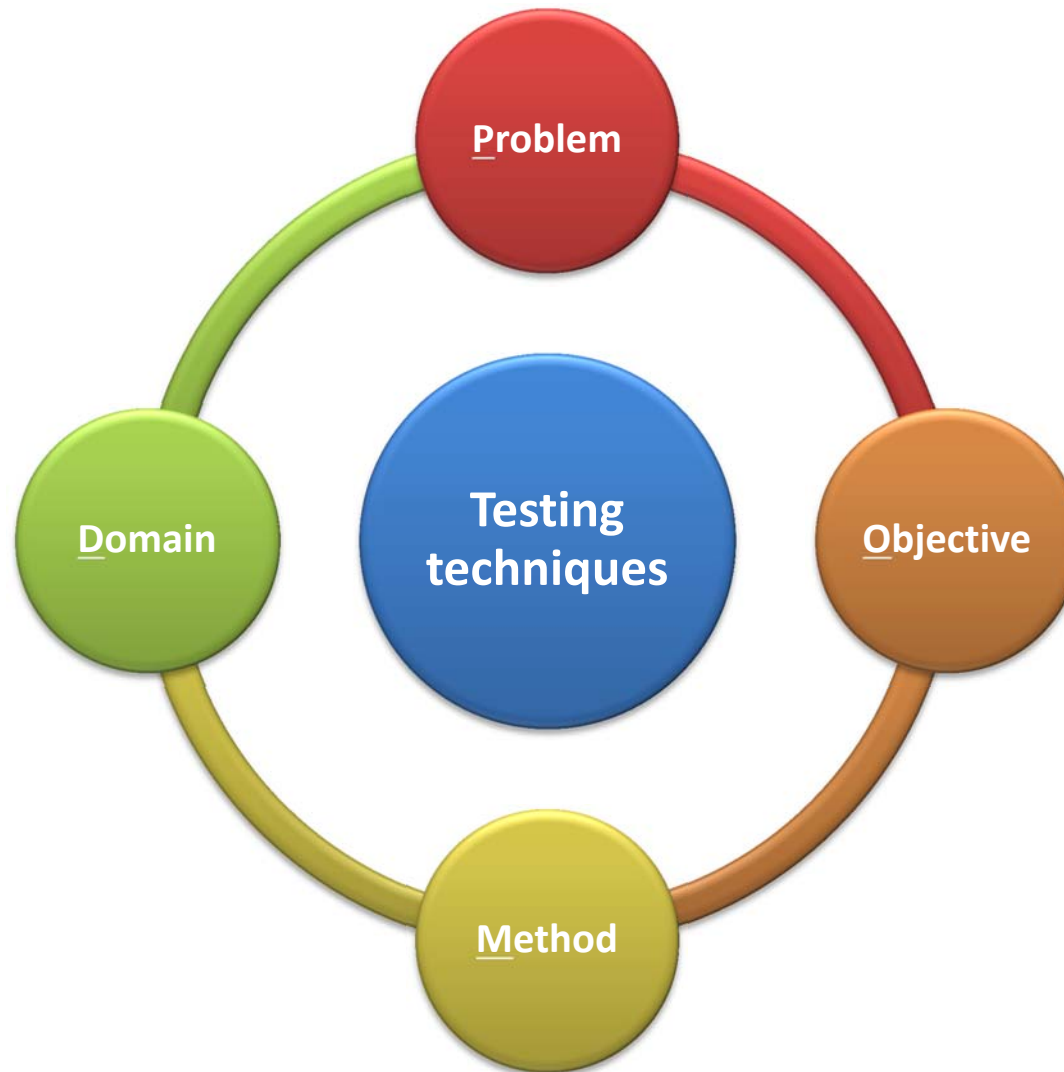
**SE-307 Software Testing Techniques**

<http://my.ss.sysu.edu.cn/wiki/display/SE307/Home>

Instructor: Dr. Wang Xinming, School of Software, Sun Yat-Sen University

# Review: PMOD

---



# Review: Problems

---

- Why testing is hard?
  - **Fundamental** problems
    - Test oracle
    - Test adequacy
    - Test generation
  - **Important** problems
    - Test process and plan
    - Test automation



# Review: Test Oracle

---

- Practical Test Oracle
  - Specification as test oracle
  - Reference implementation as test oracle
  - Program assertion as test oracle
  - Metamorphic property as test oracle
  - Execution profiling as test oracle
  - Heuristic and statistical test oracle
- They are not mutually exclusive!

# Fundamental problems in testing

---

- Test oracle
- Test adequacy
- Test generation

# The test adequacy problem

---

- What we would like:
  - A real way of measuring effective testing  
*If the system passes an adequate suite of test cases, then it must be correct (or dependable)*
- But that's impossible!
  - Adequacy of test suites, in the sense above, is provably undecidable.
- So we'll have to settle on weaker proxies for adequacy
  - Test adequacy criteria
- Problem statement: *Which adequacy criteria shall we use to achieve a good balance of schedule and quality?*

## 1. Design rules to highlight **inadequacy**

---

- Many design disciplines employ *design rules*
  - “Traces (on a chip, on a circuit board) must be at least 10mm wide and separated by at least 5mm”
  - Building code: Maximum span between beams in ceiling, floor, and walls; acceptable materials; wiring insulation; ...
- Design rules do not guarantee good designs
  - Good design depends on talented, creative, disciplined designers; design rules help them avoid or spot flaws
  - Test design is no different

## 2. Obligation of the tester

---

- Adequacy criterion = set of test obligations
- A test suite satisfies an adequacy criterion if every test obligation in the criterion is covered by at least one of the test cases in the test suite.

- Example:

*The statement coverage adequacy criterion is satisfied by test suite  $S$  for program  $P$  if each executable statement in  $P$  is executed by at least one test case in  $S$ .*



# Ways to understand test adequacy

## 3. Indicator to test progress

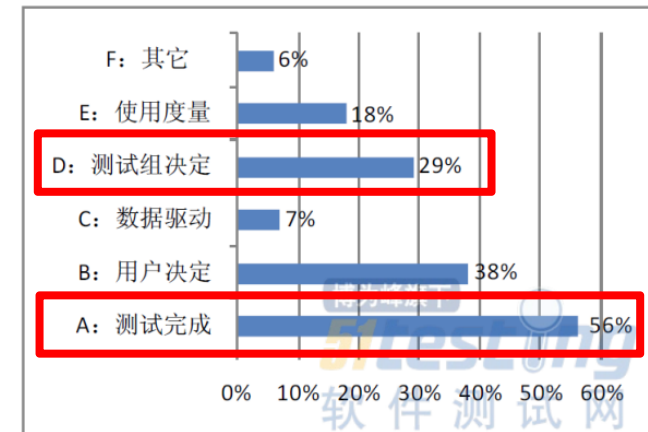
SE-307 Software Testing Techniques

- Progress measurement

- How far has we test the program?
- When can we release the software?

- Test suite construction

- Guidance in devising a thorough test suite
- Revealing missing tests: what might I have missed with this test suite?



2011 年调查中公司决定产品是否可以交付的因素

# Practical Test Adequacy Criteria

---

- Whitebox (白盒测试充分性标准)
  - Control-flow coverage (控制流覆盖)
  - Logic coverage (逻辑覆盖)
  - Data-flow coverage (数据流覆盖)
  - Interface coverage (接口覆盖)
  - Mutant coverage (变异覆盖)
- Blackbox (黑盒测试充分性标准)

# Control Flow & Data Flow Coverage

---

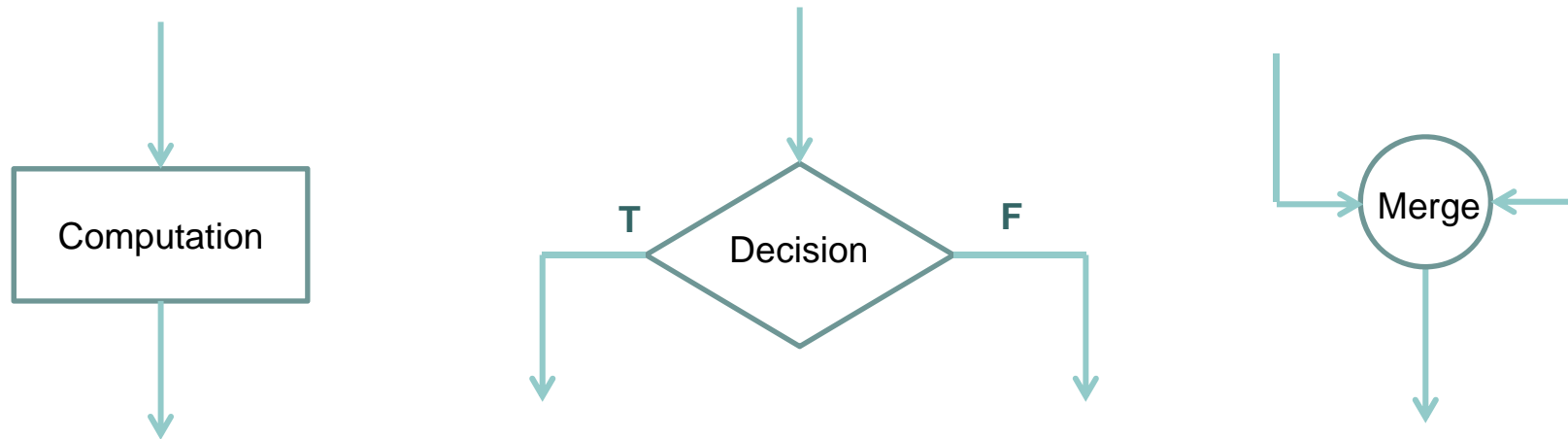
- **Control Flow Coverage** addresses the coverage of “execution” or “control” paths.
- **Data Flow Coverage** addresses the coverage of “interactions among data items.”

# Definition: Control Flow Graph

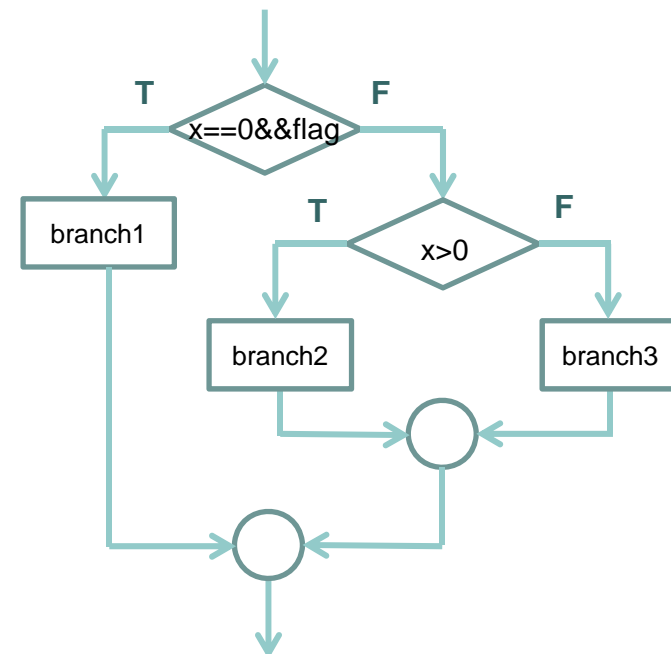
---

- **Control Flow Coverage** requires the conversion of the source code of *a function* to a **control flow graph (CFG)**
- **A CFG contains the following elements:**
  - **Node:** each node represents an unit of processing
    - **Entry/Exit node:** Entry node is where the function starts and Exit node is where the function returns.
    - **Decision node:** A node associated with multiple out-edges (e.g. binary predicate); also called a “branching” node.
    - **Merge node:** A node associated with multiple in-edges
    - **Computation node:** A node that is neither decision nor merge node.
  - **Edge:** each edge (line between nodes x and y) represents the “execute-after” relation between node x and node y.
    - **Out-edges/In-edges:** a directed edge that originates from a node is an out-edge; a directed link that ends up in a node is an in-edge.

# Control Flow Graph Representation



*if (x==0 && flag) { branch 1 }  
else if (x>0) { branch 2 }  
else { branch 3 }  
....*



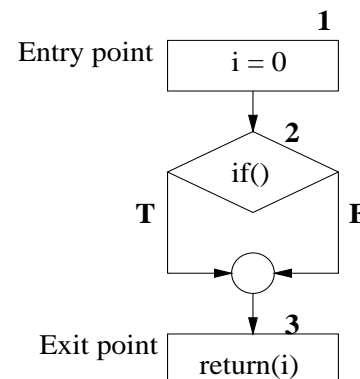
# Notes

- All decision nodes are required to be side-effect free.
  - Extract the computation out from the condition.

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */
```

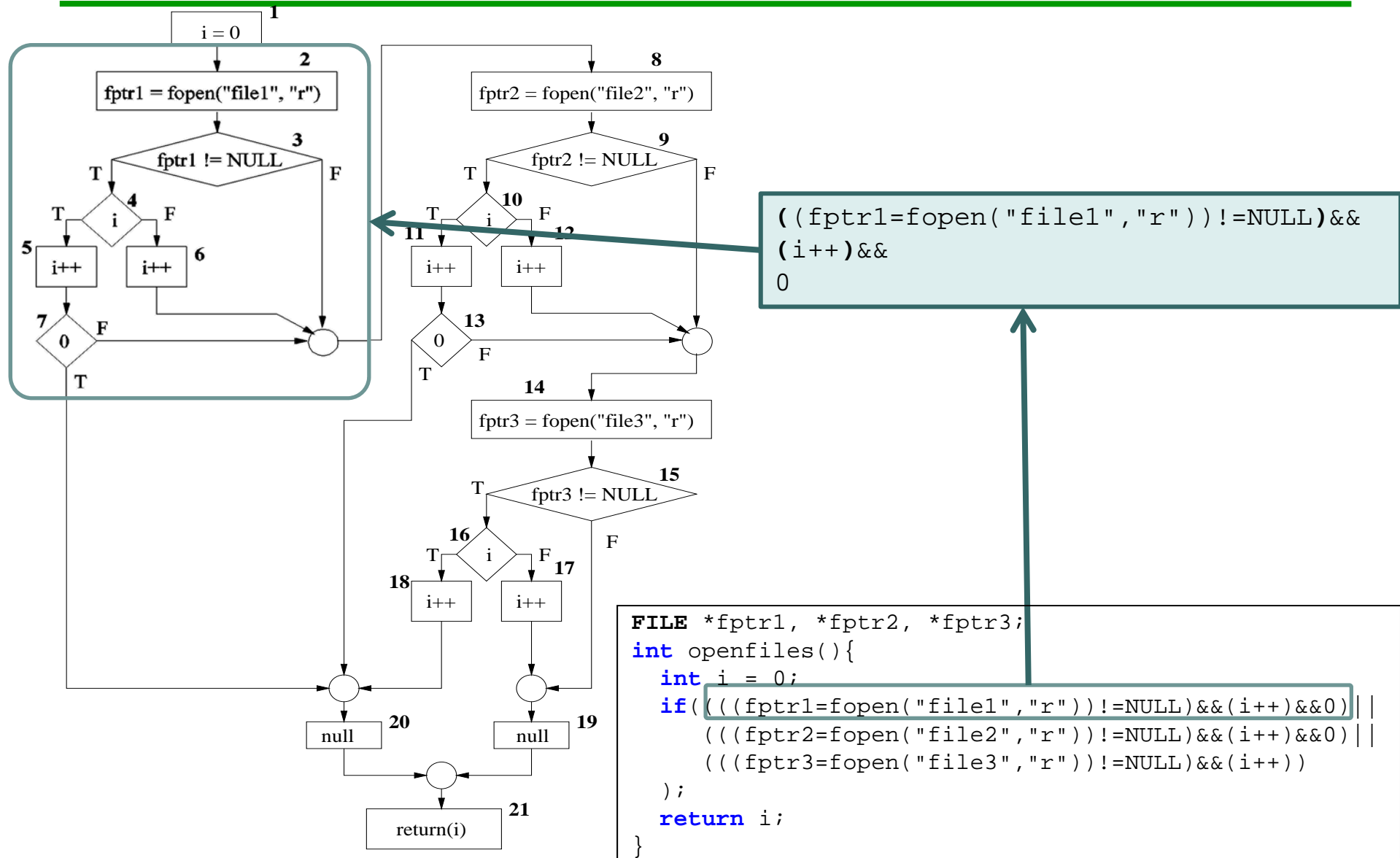
```
/* This function tries to open files "file1", "file2", and "file3"
   for read access, and returns the number of files successfully
   opened. The file pointers of the opened files are put in the
   global variables. */
```

```
int openfiles(){
    int i = 0;
    if( ((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && 0) ||
        ((( fptr2 = fopen("file2", "r")) != NULL) && (i++) && 0) ||
        ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
    );
    return i;
}
```



Decision node has side-effect!

# Side-Effect Free CFG



# Control Flow Coverage

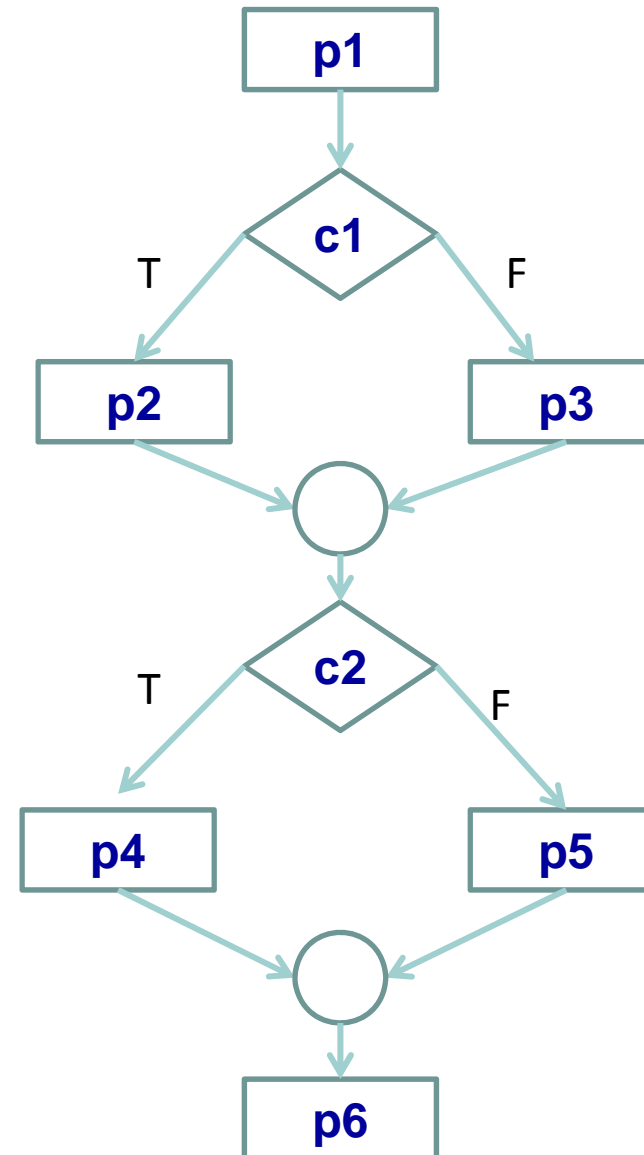
---

- The basic idea behind **control flow coverage** is to define the test adequacy criteria as graph elements that we design test data to “cover”.
  - How to generate the test data to cover these elements belongs to another fundamental problem: **test generation**.
- Types of control flow coverage that can be defined as elements CFG:
  - Statement coverage（语句覆盖）
  - Basic block coverage（基本块覆盖）
  - Branch coverage（分支覆盖）
  - Loop coverage（循环覆盖）
  - All path coverage（全路径覆盖）
  - Basis path coverage（基本路径覆盖）



# Statement coverage (语句覆盖)

- Statement coverage concerns the coverage of every *executable* statements.
  - Cover all decision and computation nodes in CFG.
  - Insensitive to the logical operators (|| and &&) in the decision node.
  - Does not consider *declarative* or *unreachable* statements.
  - A variant: *line coverage* (行覆盖)
- We can cover all the statements with only 2 paths out of a total of 4 paths:
  - Path1: p1-c1-p2-c2-p5-p6
  - Path2: p1-c1-p3-c2-p4-p6
- The weakest among all control flow coverage criteria



# Basic Block Coverage (基本块覆盖)

---

- 100% statement coverage  $\Leftrightarrow$  100% basic block coverage
  - Then why bother introducing another coverage criteria?

- Problem of statement coverage:

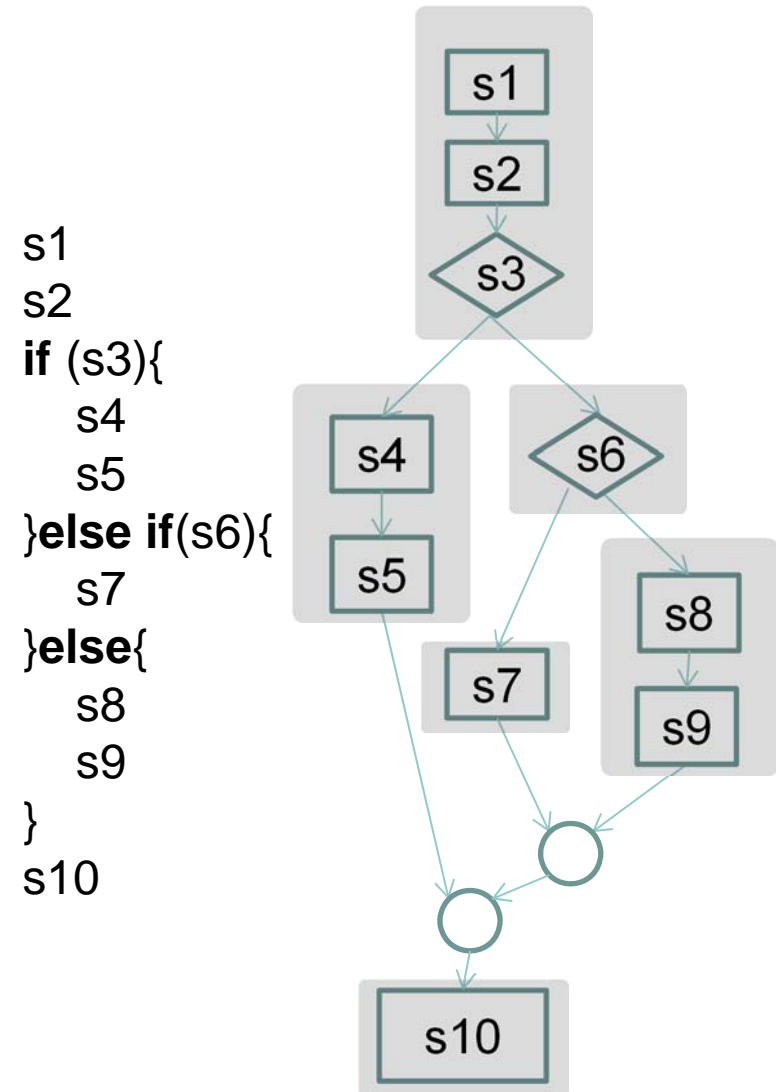
```
if (...) {  
    one statement  
} else {  
    ninety nine statements  
}
```

If we have tested one path, then the test process is either 1% (too pessimistic) or 99% (too optimistic).

- Another problem: instrumentation cost
- Basic block: a better unit to measure coverage
  - Much more commonly used in the industry than statement coverage

# Definition: Basic Block

- A **basic block** is a maximal sequence of statements having the following properties:
  - (单入口) It can only be entered through the first statement;
  - (单出口) It can only exit through the last statement;
  - (顺序执行) Whenever the first statement is executed, the remaining statements are executed in the given sequential order.



# Identify Basic Blocks in CFG

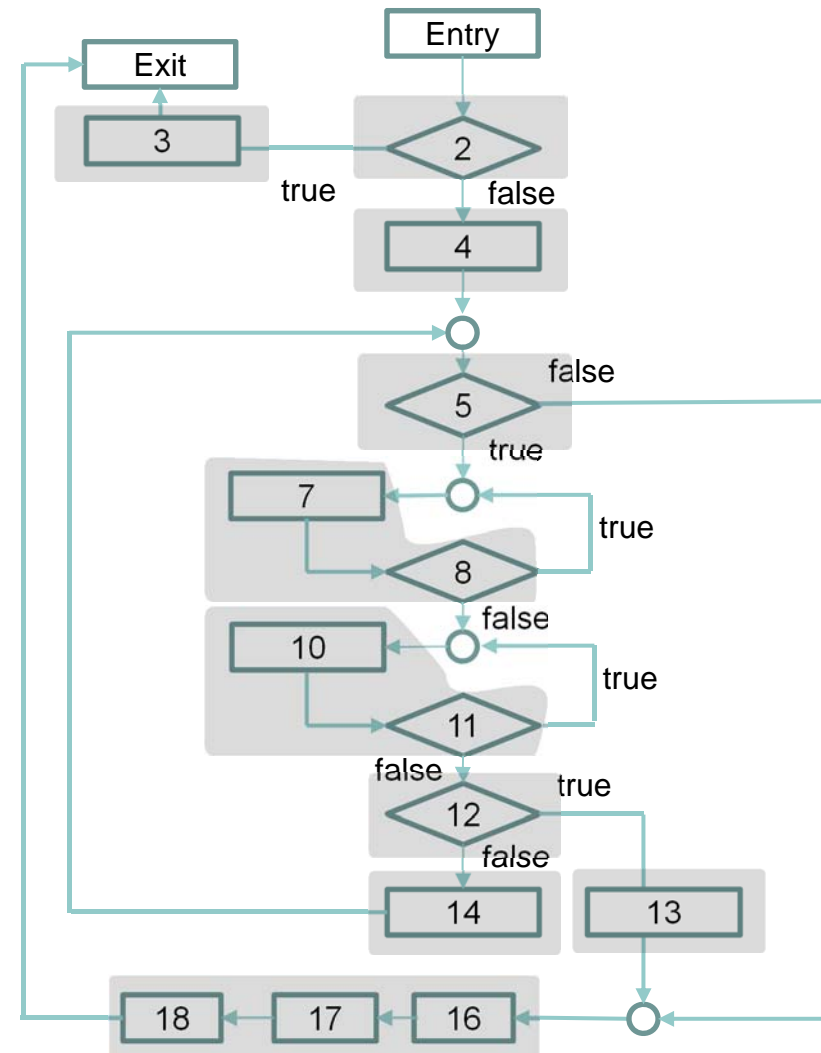
---

- Determine the *block entry*, the first node in a basic block
  - The first node in the CFG is a block entry;
  - Any node that is directly reached by a *decision node* is a block entry;
  - Any node that is directly reached by a *merge node* is a block entry.
- For each block entry, its basic block contains all the following nodes up to, but not including, the next entry node or the exit node of the CFG.

# Quiz 1: CFG & Basic Block Coverage

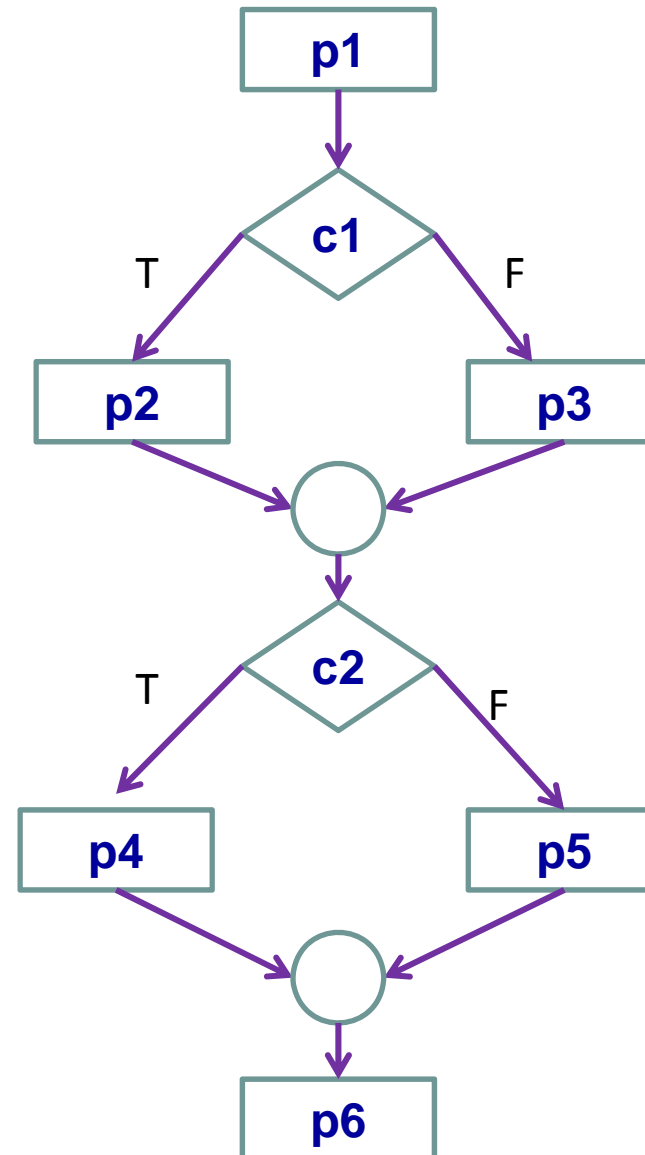
- Draw the CFG for the following code and identify basic blocks

```
// External input-output array: int a[]
void quicksort( int m, int n ) {
1:   int i, j, v, x;
2:   if ( n <= m )
3:       return;
4:   i = m-1; j = n; v = a[n];
5:   while(1) {
6:       do
7:           i=i+1;
8:       while( a[i] < v );
9:       do
10:          j=j-1;
11:      while( a[j] > v );
12:      if ( i >= j )
13:          break;
14:      x = a[i]; a[i] = a[j]; a[j] = x;
15:  }
16:  x = a[i]; a[i] = a[n]; a[n] = x;
17:  quicksort( m, j );
18:  quicksort( i+1, n );
}
```



# Branch Coverage (分支覆盖)

- Branch coverage concern the coverage of all *feasible* edges coming out of the decision control.
  - Cover all edges in CFG.
  - Also known as *decision coverage*
  - Again, insensitive to the logical operators (|| and &&) in the decision node.
- Note that we can cover all the branches with only 2 paths out of a total of 4 paths:
  - Path1: p1-c1-p2-c2-p5-p6
  - Path2: p1-c1-p3-c2-p4-p6
- In general, we can cover all branches with less than or equal to the total number of paths.



# Notes

- Branch coverage for exception handling code

```

1:  try {
2:      in.open("testfile.txt")
3:      str = in.readline()
4:      if(str == null)
5:          throw new IOException();
6:      in.close();
7:  } catch (IOException e) {
8:      system.out.println("hello!");
9:  } finally{
10:     system.out.println("hello!");
11:  }
12:  ...

```

How many test cases do we need to cover every edge (branch coverage)?

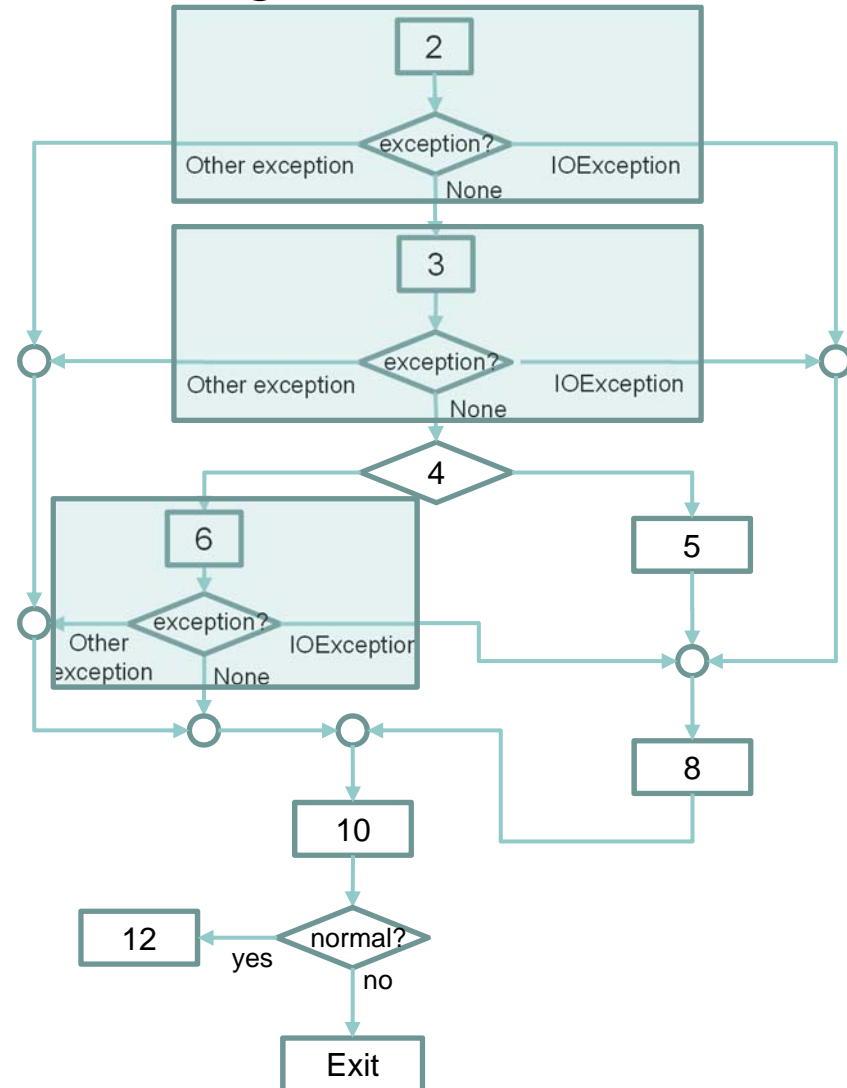
**"Happy" path:** 1,2,3,4,6,10,12

**Exception at 2:** 1,2,8,10,12 and 1,2,10

**Exception at 3:** 1,2,3,8,10,12 and 1,2,3,10

**Throw at 5:** 1,2,3,4,5,8,10

**Exception at 6:** 1,2,3,4,6,8,10,12 and 1,2,3,4,6,10

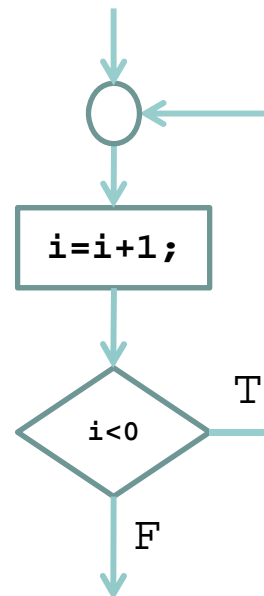


# Branch Coverage vs. Statement Coverage

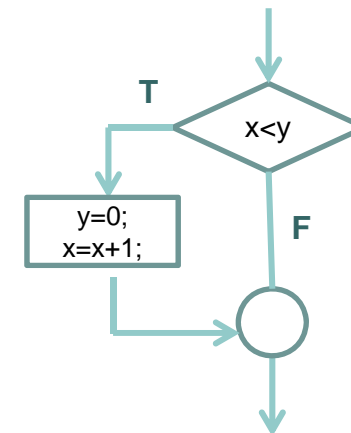
- What are their relation?
- Branch coverage *subsumes* statement coverage
  - 100% branch coverage  $\Rightarrow$  100% statement coverage
  - 100% statement coverage  $\neq$  100% branch coverage

- Example:

```
do{
    i=i+1;
}while(i<0);
```



```
if (x < y){
    y = 0;
    x = x + 1;
}
```

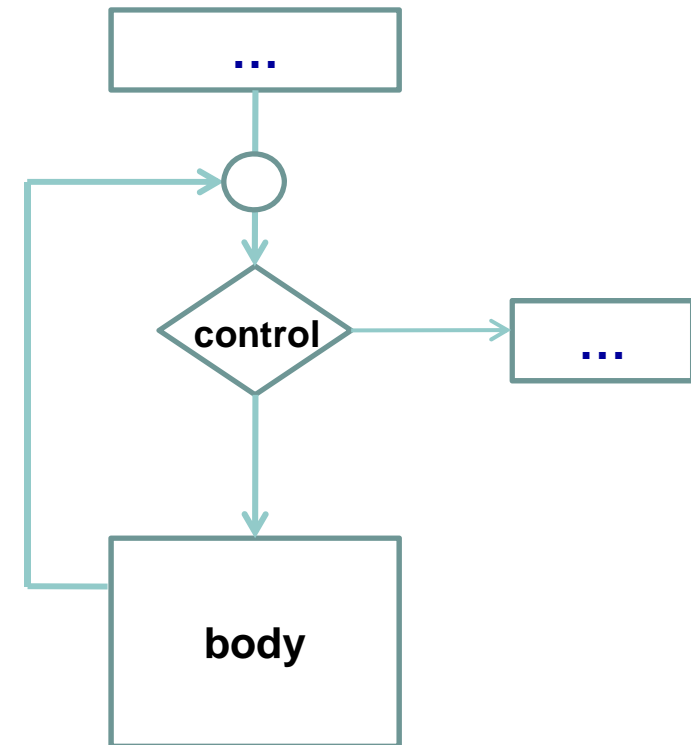


- Any other example?



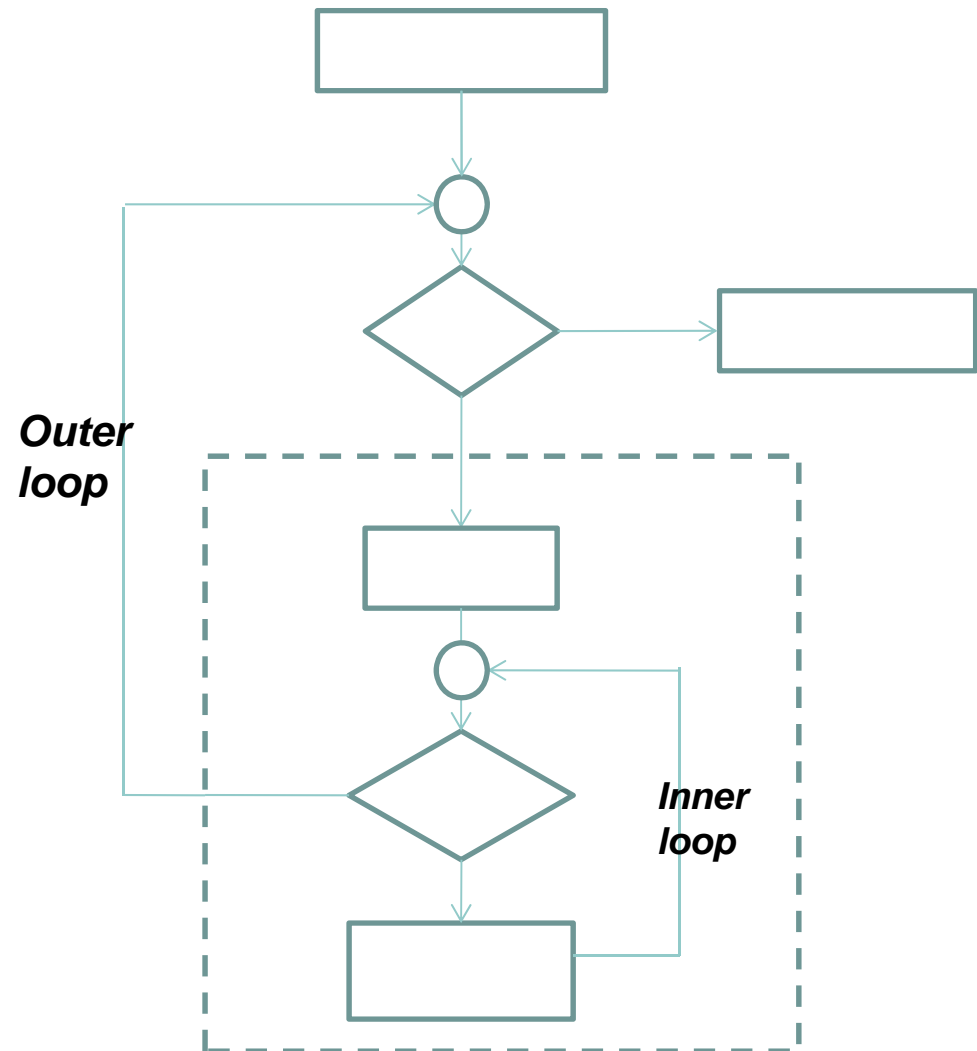
# Loop Coverage (循环覆盖)

- A loop has 2 main components:
  - **Loop body**: statements that is executed repeatedly
  - **Control statement**: determine whether if the loop body should be executed and thus serve as entry and exit criteria.
- Loop coverage concern the repetition count of the loop body execution, including the following 5 situations, if they are feasible:
  - Bypass the loop (never enter the loop)
  - Enter the loop once
  - Enter the loop more than once
  - Enter the loop max-1 times
  - Enter the loop max times



# The Case of Nested Loop

- If we use the 5 possible test cases for a loop structure as a guideline, then 2 nested loops would need  $4 \times 5 = 20$  test cases.
- In general, this would still be big for multiple nested loops:  $O(5^n)$  for  $n$ -nested loops.



# Quiz 2: Loop Coverage

---

Find test cases that achieve 100% loop coverage for the following code:

```
i = strlen(fname) - 1;  
while (i > 0 && fname[i] != '.') {i--;
```

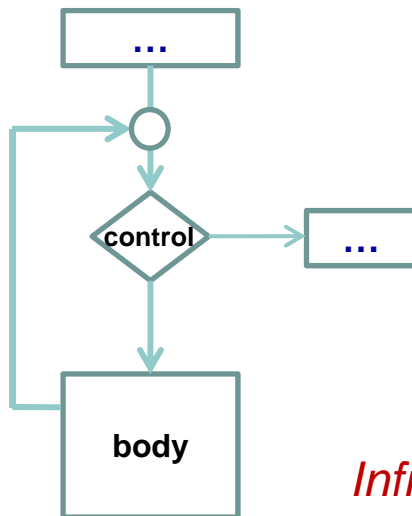
To do “loop coverage”, list some of the possible test cases:

- (0 times) → *fname contains: “report1.”*
- (1 time) → *fname contains: “report1.t”*
- (more than 1 time) → *fname contains: “report1.txt”*
- (max – 1 times) → *fname contains: “a.txt”*
- (max times) → *fname contains: “report1”*

**Note:** max loop count here is the *length of the string minus 1*

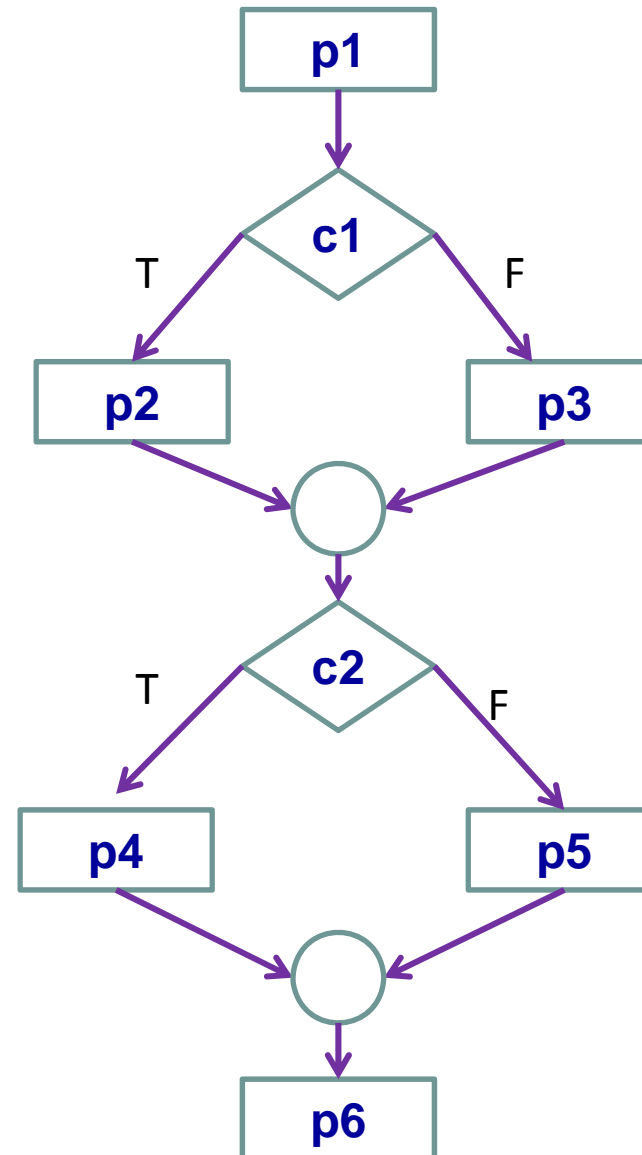
# All Path Coverage (全路径覆盖)

- All path coverage concerns the coverage of every feasible path in the CFG
  - The example: four paths
  - Ignore infeasible paths.
- Problem:
  - How to handle loop?



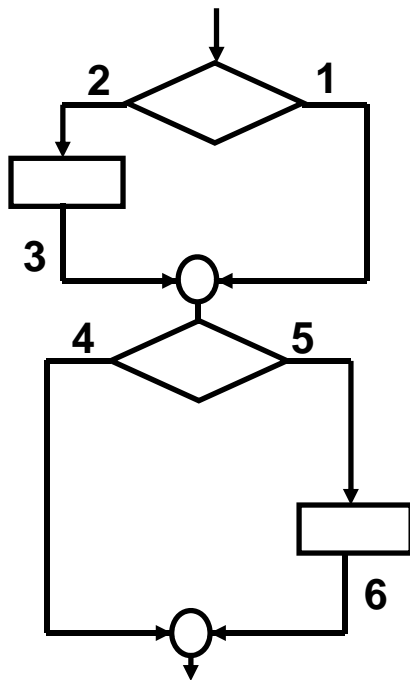
That is why we need to introduce the concepts of *linearly independent paths*, *basis path set*, and *basis paths coverage*.

*Infinite number of paths!*



# Linearly Independent Paths

- A set of paths is said to be *linearly independent*, if none of them can be constructed as a “linear combination” of the other paths.
- For example:



path1, path2 and path3 are linearly independent

	①	②	③	④	⑤	⑥
path1	1				1	1
path2	1			1		
path3		1	1	1		
path4		1	1		1	1

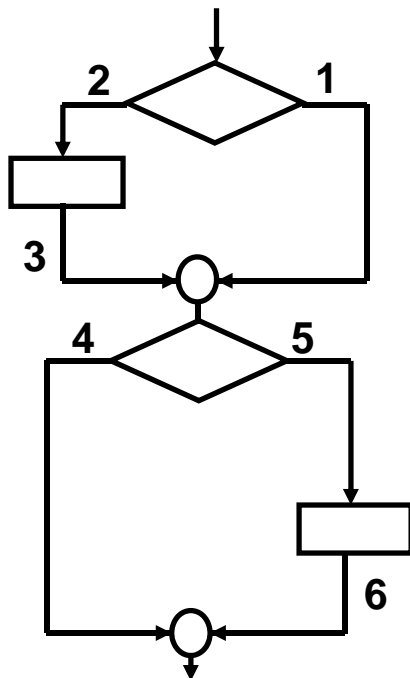
path1, path2, path3, path4 are not linearly independent, because  

$$\begin{aligned} \text{path 4} &= \text{path3} + \text{path1} - \text{path2} = (0,1,1,1,0,0) + (1,0,0,0,1,1) - (1,0,0,1,0,0) \\ &= (1,1,1,1,1,1) - (1,0,0,1,0,0) \\ &= (0,1,1,0,1,1) \end{aligned}$$

here ‘-’ means sub-string deletion, ‘+’ means sub-string insertion

# Basis Path Set

- A set of linearly independent paths is said to be a *basis path set for a CFG*, if any path in the CFG can be constructed as a linear combination of these paths.

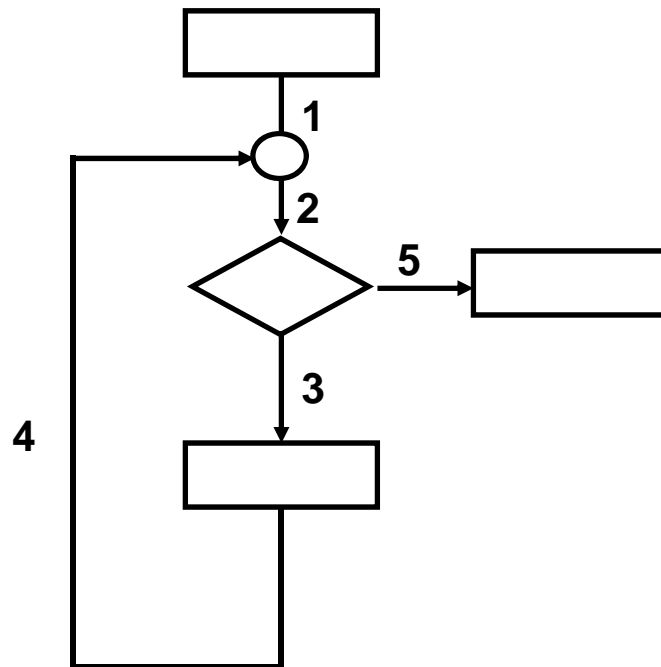


	①	②	③	④	⑤	⑥
path1	1				1	1
path2	1			1		
path3		1	1	1		
path4		1	1		1	1

Although path1 and path3 are linearly independent, they are NOT a basis path set for this CFG, because no linear combination of path1 and path3 can get path4.

# Basis path coverage

- Concern the coverage of basis path set
  - Typically a much smaller subset (always finite) of all possible paths (can be infinite).
  - Basic idea:** it might be sufficient to test the basis path set only instead of all possible paths, since the other paths can be constructed as their linear combination.



Total number of paths may be “infinite” (very large) because of the loop

Basis path set contains two paths only:

path1 : 1,2,5

path2 : 1,2,3,4,2,5

Consider path3: 1,2,3,4,2,3,4,2,5

path2 - path1 = 3,4,2

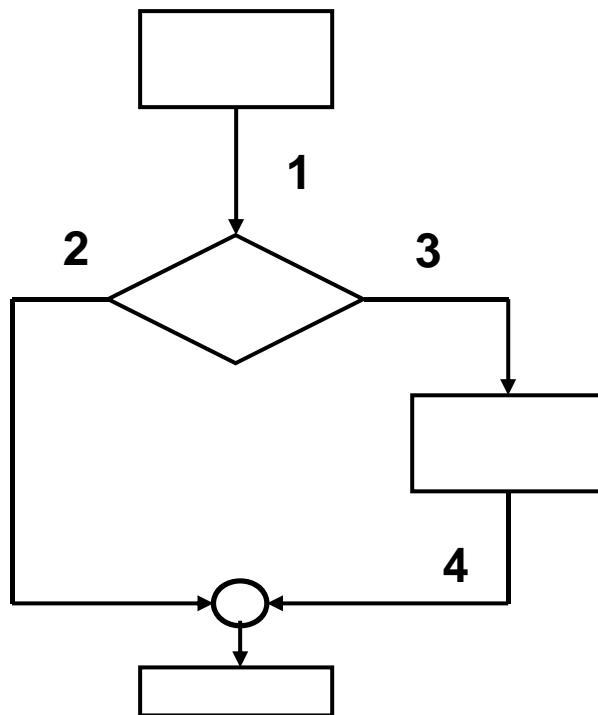
path2 + **3,4,2** = 1,2,3,4,2,**3,4,2**,5

Therefore

path3 = path2 + (path2 - path1)

# Relation with Cyclomatic Complexity

- **McCabe's Cyclomatic complexity number** is defined as the size of basis path set in the CFG.
  - There are 2 ways to get the Cyclomatic complexity number from a CFG.
    - # of binary decisions + 1
    - # of edges - # of nodes + 2



The basis path set contains two paths:

Path1 : 1,2

Path2 : 1,3,4

McCabe's Cyclomatic Number:

a) # of binary decisions + 1 = 1 + 1 = 2

b) # of edges - # of nodes + 2 = 4 - 4 + 2 = 2

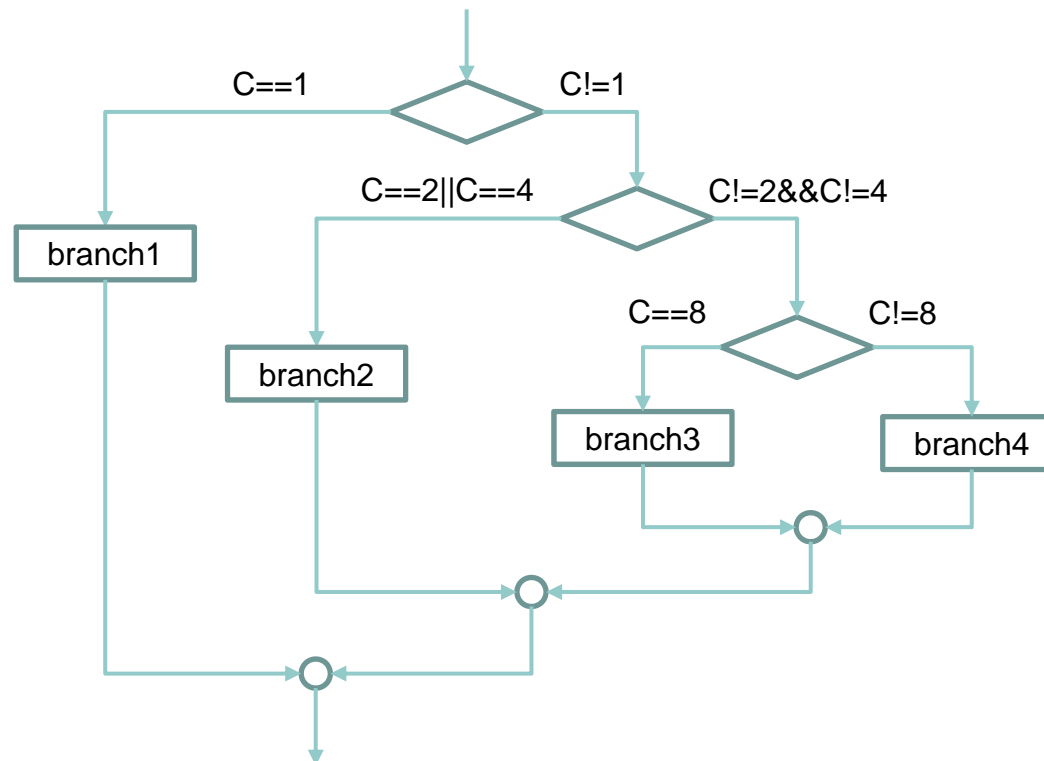
McCabe, "A complexity Measure," *IEEE Transactions on Software Engineering*, Dec. 1976



# Notes 1

- A switch statement with  $n$  branches has  $n-1$  binary decision node.

```
switch(C){  
  case 1:  
    branch1;  
    break;  
  
  case 2:  
  case 4:  
    branch2;  
    break;  
  
  case 8:  
    branch3;  
    break;  
  
  default:  
    branch4;  
};
```



Cyclomatic number: 4

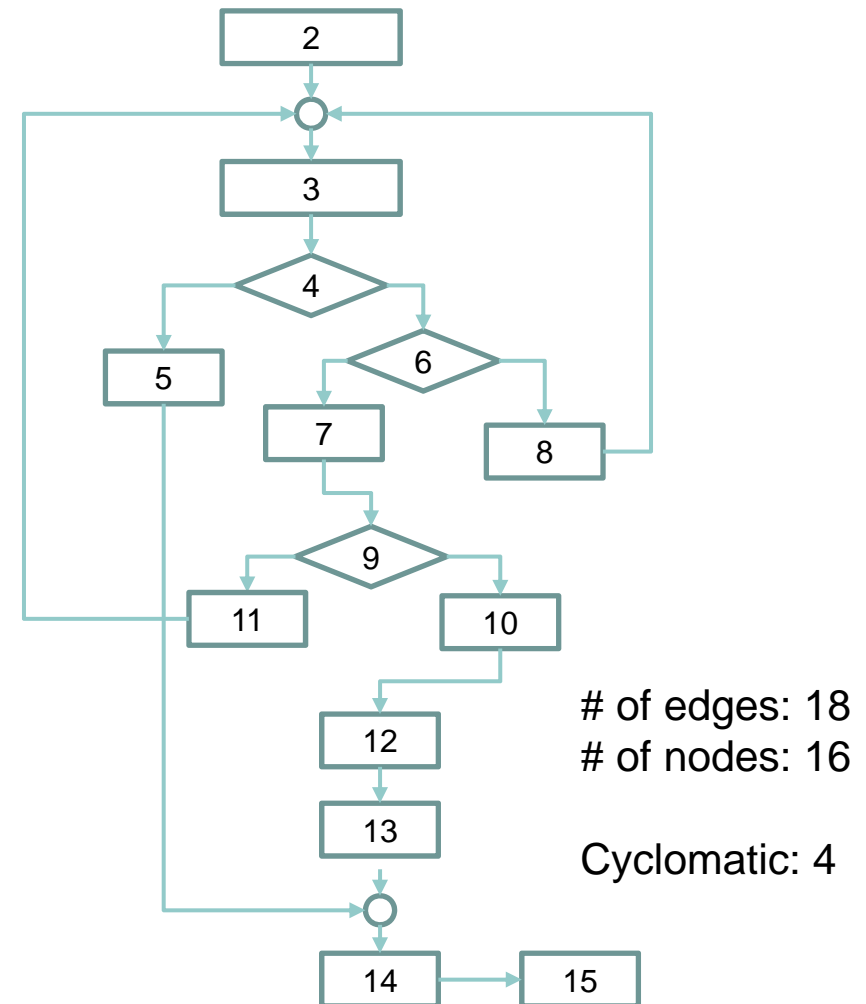
# Notes 2

- The two rules are still valid when there is goto statement or exception/throw.

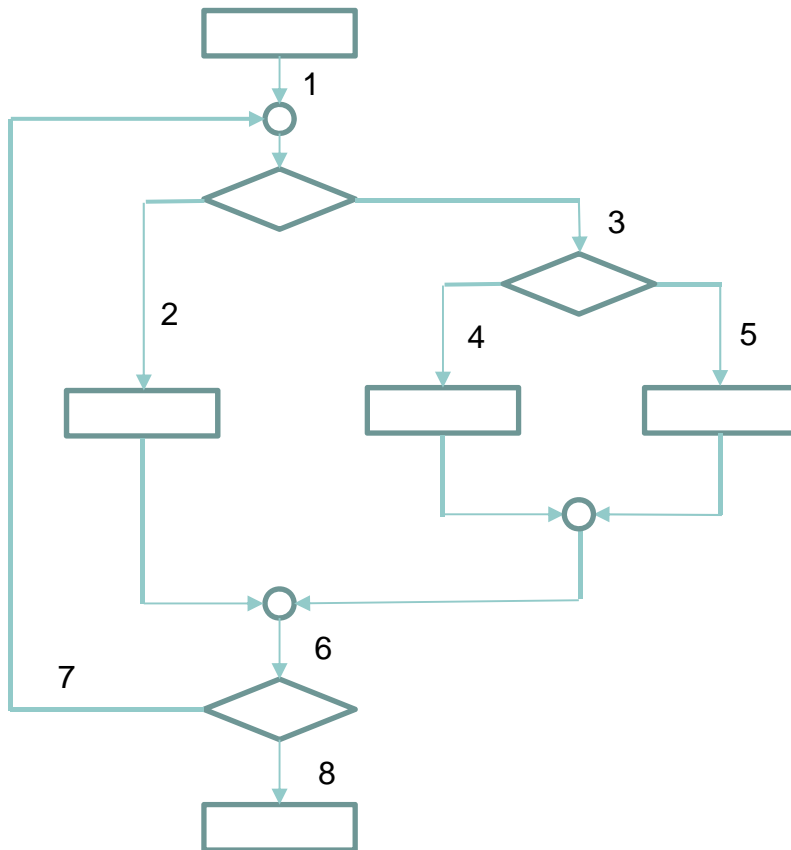
```

1  int functionY(){
2      int x = 0, y = 19;
3  A: x++;
4      if (x > 999)
5          goto D;
6      if (x % 11 == 0)
7          goto B;
8      else goto A;
9  B: if (x % y == 0)
10         goto C;
11     else goto A;
12 C: printf("%d\n", x);
13     goto A;
14 D: printf("End of list\n");
15     return 0;
16 }

```



# Quiz 3: Basis Path Coverage



- Cyclomatic:  $E+N-2 = 4$

- The basis path set

Path 1: 1,3,4,6,8

Path 2: 1,3,5,6,8

Path 3: 1,2,6,8

Path 4: 1,2,6,7,2,6,8

- Consider other paths

Path 5: 1,3,4,6,7,2,6,8

Path4 - Path3 = 6,7,2

Path1 + 6,7,2 = 1,3,4,**6,7,2**,6,8 = path5

Path 6: 1,3,4,6,7,3,5,6,8

Path4 - Path3 = 2,6,7

Path2 + 2,6,7 = 1,**2,6,7**,3,5,6,8

1,2,6,7,3,5,6,8 - path3 = 6,7,3,5

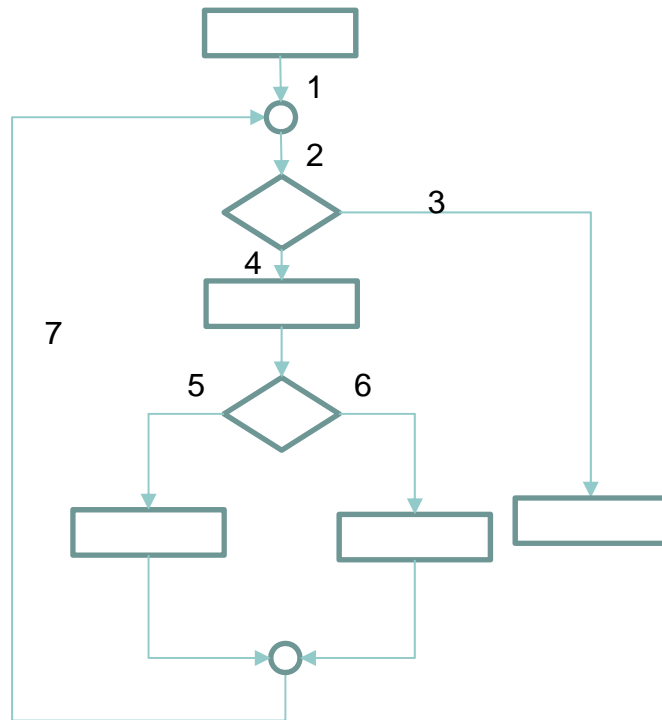
Path1+6,7,3,5 = 1,3,4,**6,7,3,5**,6,8 = path6

# Quiz 3: Basis Path Coverage

```

int i=20;
while (i<10) {
    System.out.printf("i is %d", i);
    if (i%2 == 0){
        System.out.println("even");
    } else {
        System.out.println("odd");
    }
}

```



- The size of basis path set (Cyclomatic number)?

- $E - N + 2 = 10 - 9 = 3$

- **Example:**

Path1 = 1,2,3

Path2 = 1,2,4,5,7,2,3

Path3 = 1,2,4,6,7,2,3

Consider Path5: 1,2,4,5,7,2,4,6,7,2,3

Path3 - Path1 = 4,5,7,2

Path2 - Path1 = 4,6,7,2

Path1 + **4,5,7,2** + **4,6,7,2** =  
 1,2,**4,5,7,2**,**4,6,7,2**,3 = Path5

Therefore:

Path 5 = (Path3-Path1)+(Path2-Path1) + Path1

# Review: Control flow coverage

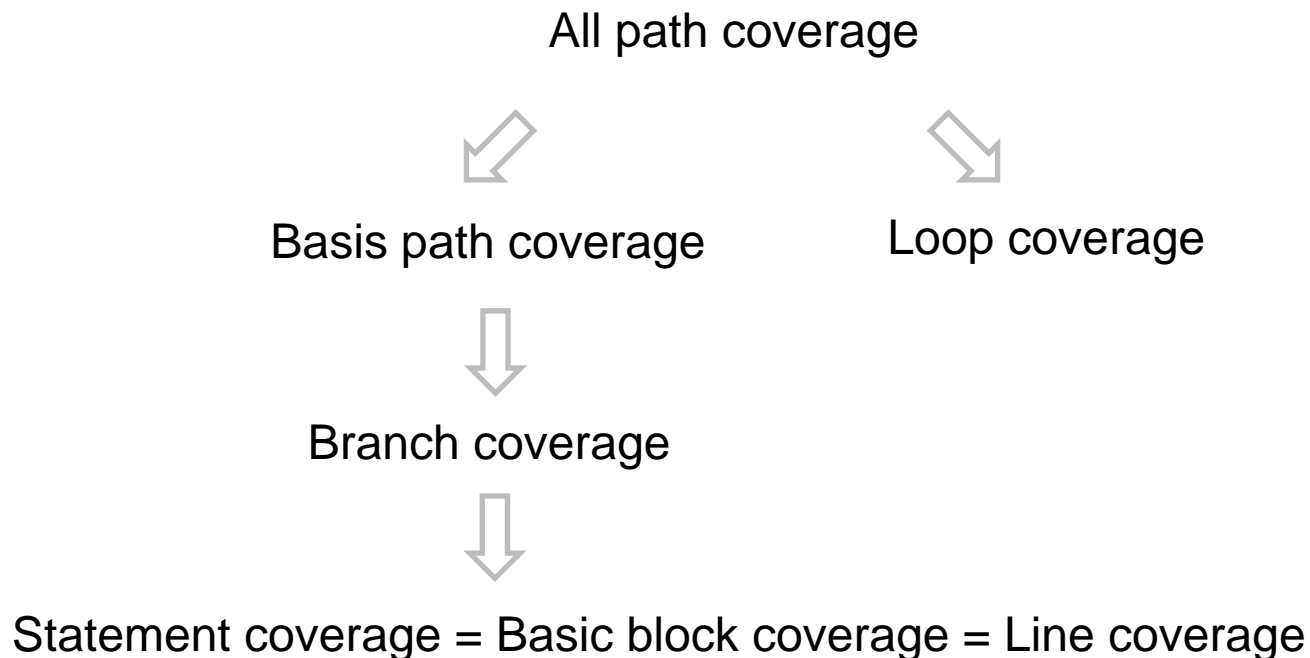
---

- What we have just learnt:
  - Statement coverage (语句覆盖)
  - Line coverage (行覆盖)
  - Basic block coverage (基本块覆盖)
  - Branch coverage (分支覆盖)
  - Loop coverage (循环覆盖)
  - All path coverage (全路径覆盖)
  - Basis path coverage (基本路径覆盖)
- What are their relation?

# Subsumption Relation

---

- We say that coverage criterion A subsumes coverage criterion B if test suite that satisfies A must at the same time satisfies B.



# Wisdom of Using Coverage Criteria

---

- Never seek to improve coverage *just for the sake of increasing coverage*
  - Well, unless it's a command from-on-high
- Coverage is not the goal
  - Finding failures that expose faults is the goal
  - No amount of coverage will prove that the program cannot fail
  - Measuring coverage can still be a useful indicator of progress toward a thorough test suite, of trouble spots requiring more attention.

# Tool: gcov for C/C++

---

## Supported coverage types: *Line coverage*

```
-: 0:Source:wordcount.c
-: 0:Graph:wordcount.gcn0
-: 0:Data:wordcount.gcda
-: 0:Runs:1
-: 0:Programs:1
-: 1:#include <stdio.h>
-: 2:#define TRUE 1
-: 3:#define FALSE 0
1: 4:main() {
1: 5:  int nl=0, nw=0, nc=0;
1: 6:  int inword=FALSE;
-: 7:  char c;
-: 8:
1: 9:  c=getchar();
4: 10: while (c != EOF) {
2: 11:     ++nc;
2: 12:     if (c=='\n')
2: 13:         {++nl;}
4: 14:     if (c==' ' || c=='\n' || c=='\t')
2: 15:         {inword=FALSE; }
#####16:     else if (!inword)
#####17:         {inword=TRUE; ++nw; }
2: 18:     c=getchar();
-: 19: }
1: 20: printf("%d lines, %d words, %d chars\n",
-: 21:         nl, nw, nc);
-: 22: }
```



# Tool: CodeCover for Java

<http://codecover.org>

Supported coverage types: *statement*, *branch*, *loop*, *condition*

Will be used in assignment #2

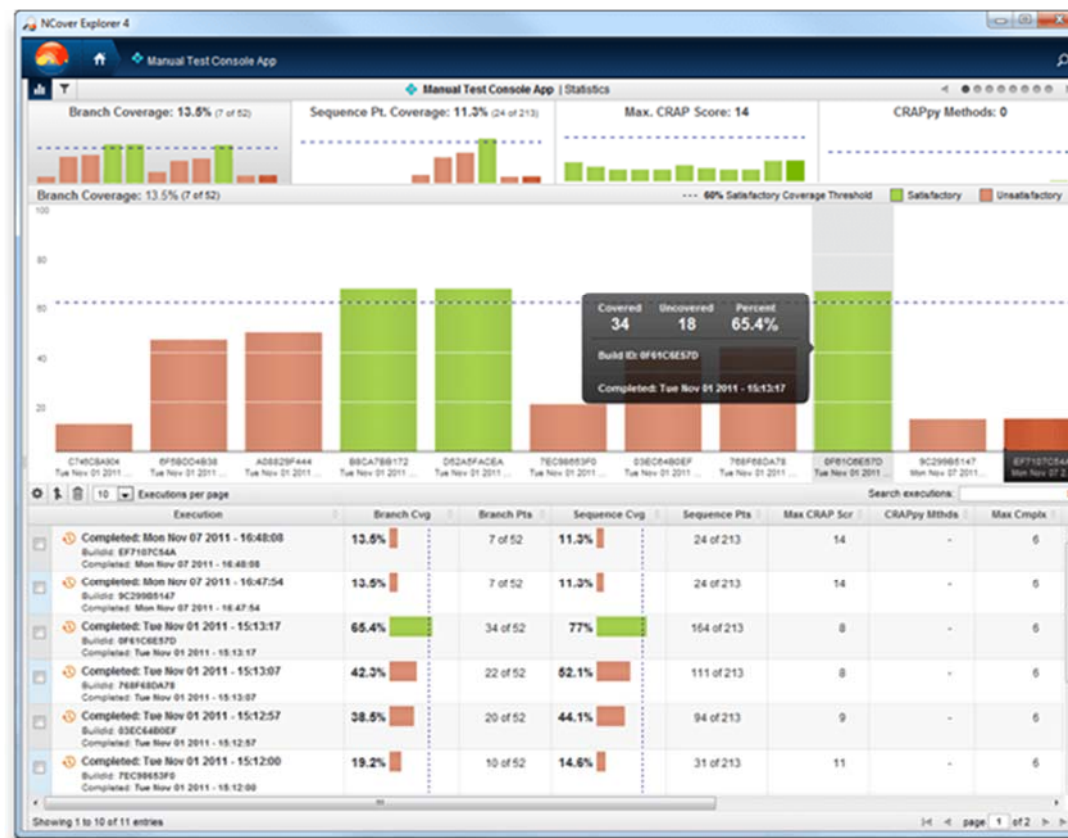
The screenshot shows the Eclipse IDE with the 'Input.java' file open. The code includes a catch block for exceptions and a try block for reading a file. Below the code editor, the 'Coverage' tab is active, displaying a table of coverage data for various classes and methods. The table is highlighted with a red rectangle.

Name	Statement	Branch	Loop	Term	?-Operator	Synchronized
Input	0.0 %	-	-	-	-	-
Input	33.3 %	50.0 %	-	-	-	-
input2String	71.4 %	75.0 %	33.3 %	100.0 %	-	-
Parser	86.0 %	60.7 %	29.6 %	60.4 %	-	-
PITCH_ATTRIBUTES	100.0 %	75.0 %	33.3 %	83.3 %	-	-
PITCH_LENGTH	100.0 %	-	-	-	-	-
PITCH_WIDTH	100.0 %	-	-	-	-	-
Parser	100.0 %	-	-	-	-	-

# Tool: NCover for C#

<http://codecover.org/>

Supported coverage types: *statement, branch, method, path*



# Practical Coverage Criteria

---

- Whitebox (白盒测试充分性标准)
  - Control-flow coverage (控制流覆盖)
  - Logic coverage (逻辑覆盖)
  - Data-flow coverage (数据流覆盖)
  - Interface coverage (接口覆盖)
  - Mutant coverage (变异覆盖)
- Blackbox (黑盒测试充分性标准)

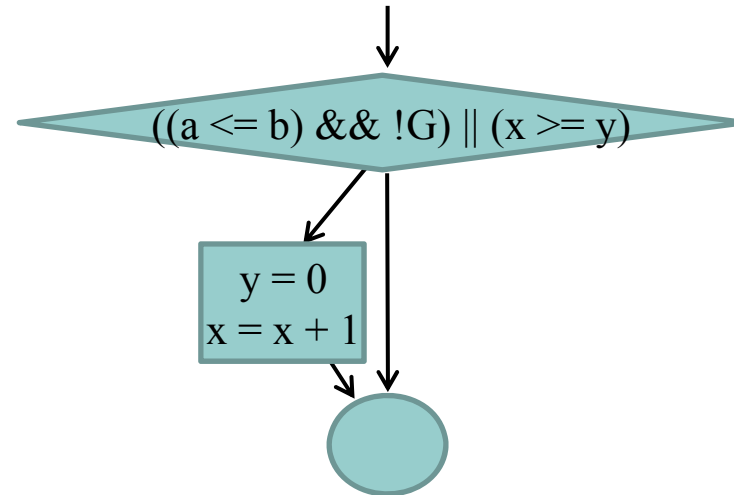
# Logic Coverage: Motivation

What if, instead of:

```
if (x < y)
{
    y = 0;
    x = x + 1;
}
```

we have:

```
if (((a>b) || G)) && (x < y))
{
    y = 0;
    x = x + 1;
}
```



Now, branch coverage will guarantee that we cover all the edges, but does not guarantee we will do so for all the *different logical reasons*

We want to test the logic of the guard of the if statement

# Types of logic coverage

---

- Decision coverage (i.e. branch coverage) 判定覆盖（也就是分支覆盖）
- Condition coverage 条件覆盖
- Short-circuiting condition coverage 短路求值语义下的条件覆盖
- Condition/decision coverage (C/D) 条件判定覆盖
- Multiple-condition coverage 条件组合覆盖
- Modified condition/decision coverage (MC/DC) 修改的条件判定覆盖

# Definitions: Condition and Decision

---

- **Decision**

- Branching expression of the if/while/for statements

- **Condition**

- A Boolean expression containing no Boolean operators (||, &&, !).
  - E.g., a>b
- If the same expression appears more than once in a decision, each occurrence is considered a distinct condition.

```
if (((a>b) || G)) && (a>b))  
{  
    y = 0;  
    x = x + 1;  
}
```

# Condition Coverage

- Condition coverage concerns the coverage of each condition taking both `true` and `false`.
  - Does not consider constant condition, such as `(true)` and `(x==x)`.
  - **Does not consider short-circuiting.**
    - Apply to languages without short-circuiting, e.g. Visual Basic
  - Condition coverage does not subsume decision coverage.
  - What is the number of test cases to achieve 100% condition coverage?
    - Always 2

```
if ( (A || B) && C ) { ... }  
else { ... }
```

Test suite that satisfy condition coverage:

A=true, B=false, C=false  
A=false, B=true, C=true

```
if(A && B) { ... }  
else { ... }
```

Test suite that satisfy condition coverage:

A=true, B=false  
A=false, B=true

Problem: branch coverage not achieved

# Condition/Decision Coverage

---

- Simply condition coverage + decision coverage
  - For each condition, the test suite covers both `true` and `false`.
  - For the whole decision, the test suite covers both `true` and `false`.

```
if ( (A || B) && C ) { ... }  
else { ... }
```

Test suite that satisfy **both** condition coverage **and** condition/decision coverage.

A=true, B=false, C=false  
A=false, B=true, C=true

```
if(A && B) { ... }  
else { ... }
```

Test suite that satisfy condition coverage, **but not** condition/decision coverage.

A=true, B=false  
A=false, B=true



# Short-circuiting Condition Coverage

- The same with condition coverage, except that we now require each condition *being actually evaluated* to `true` and `false`.
  - Apply to languages with short-circuiting, e.g. C, C++, Java
  - Short-circuiting condition coverage subsumes decision coverage.
    - Why? Think about it.

```
if ( (A || B) && C ) { ... }  
else { ... }
```

Test Cases for **Condition Coverage**:

A=true, B=false, C=false  
A=false, B=true, C=true

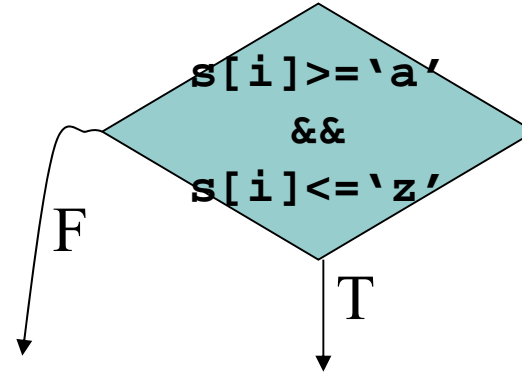
```
if ( (A || B) && C ) { ... }  
else { ... }
```

Test Cases for **Short-circuiting Condition Coverage**:

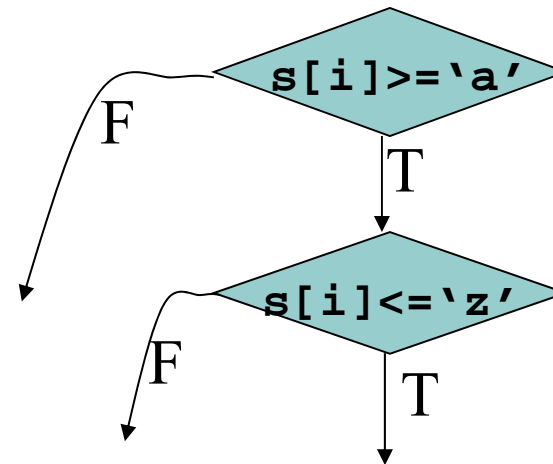
A=true, B=not-eval, C=false  
A=false, B=true, C=true  
A=false, B=false, C=not-eval

# Finding Short-Circuiting Condition Adequate Test Cases

- Transform the complex decision ....

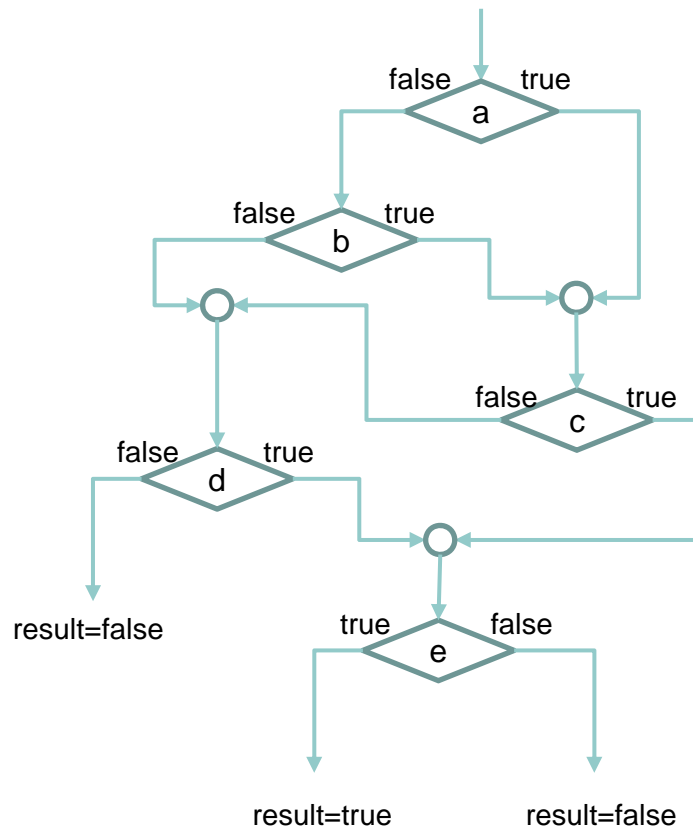


- into condition graph:



- Checking short-circuiting condition coverage => checking whether all edges in the condition graph are covered

# Example



Three test cases to satisfy short-circuiting condition coverage (cover all edges):

a=false, b=false, c=-, d=false, e=-  
 a=false, b=true, c=false, d=true, e=true  
 a=true, b=-, c=true, d=-, e=false

Notes:

- 1) it is weaker than MC/DC (need at least 6).
- 2) it is stronger than condition coverage (need 2).

Questions:

What is the **minimal** number of test cases and how to find them?

Transform it into the minimal flow problem and use the Ford-Fulkerson algorithm to solve it. Read:

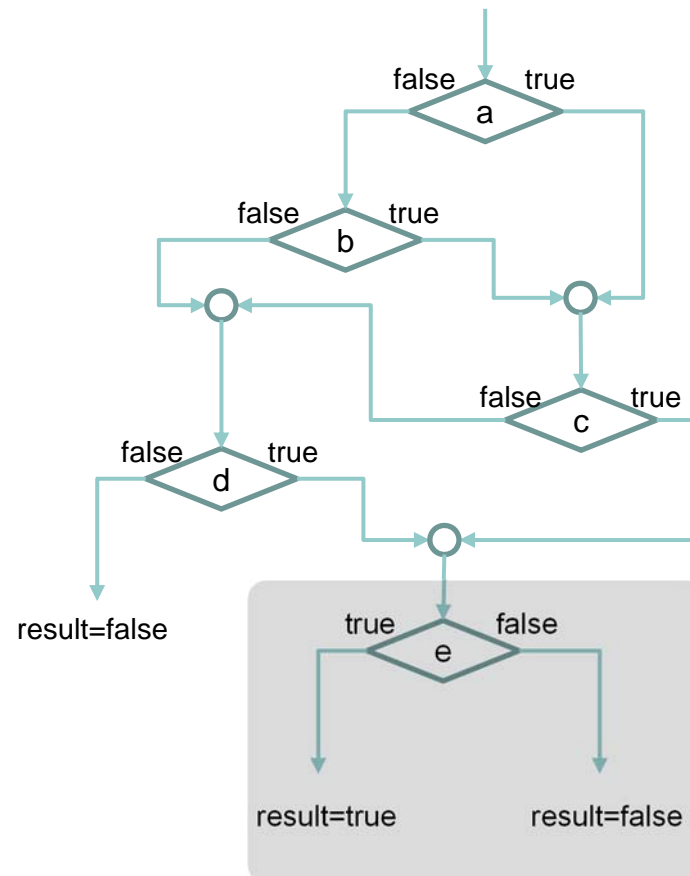
[http://bioinformatics.oxfordjournals.org/content/suppl/2012/04/29/bts258.DC1/supplemental\\_material\\_v1\\_1.pdf](http://bioinformatics.oxfordjournals.org/content/suppl/2012/04/29/bts258.DC1/supplemental_material_v1_1.pdf)

Why short-circuiting condition coverage **subsumes** decision coverage?

$( (a || b) \ \&\& \ c \ || \ d ) \ \&\& \ e$

# Why short-circuiting condition coverage subsumes decision coverage?

- There must be a condition that is evaluated last in the graph.
  - If it is evaluated, it must not have been short-circuited.
  - And it must be evaluated to `true` and `false`.



# Problem of Condition Coverage

---

```
if ( (A || B) && C ) { ...}  
else {...}
```

Test Cases for **Short-circuiting Condition Coverage**:

Test case1: A=true, B=not-eval, C=true

**Test case2: A=false, B=true, C=false**

Test case3: A=false, B=false, C=not-eval

- Do we really test the correctness of B?
  - Consider test case2, even if the evaluation result of B is `false`, the decision outcome is still the same, because C is `false`.
  - In this case, the evaluation result `false` of B *does not determine the decision outcome*. Thus this test case cannot reveal any fault in B.

That is the reason why we need to introduce MC/DC

# Modified Condition/Decision Coverage

---

- MC/DC requires that for each condition, both its evaluation `true` and `false` are shown to determine the outcome of the decision.
- That is, the outcome of a decision changes as a result of changing a single condition.
- For each condition *C*, select two test cases such that the truth values of all the conditions, except for *C*, are the same, and the decision outcome evaluates to be `true` for one and `false` for the other.

# Example of MC/ DC Coverage

With these values for  $G$  and  $(x < y)$ ,  $(a > b)$  determines the value of the predicate

With these values for  $(a > b)$  and  $(x < y)$ ,  $G$  determines the value of the predicate

With these values for  $(a > b)$  and  $G$ ,  $(x < y)$  determines the value of the predicate

$((a > b) \text{ or } G) \text{ and } (x < y)$

1	T	F	T	T	
2	F	F	T	F	
3	F	T	T	T	
4	F	F	T	F	
5	T	T	T	T	
6	T	T	F	F	

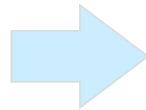
duplicate

# Finding MC/DC Adequate Test Cases

**if (A && B) then...**

- (1) create truth table for conditions.
- (2) Extend truth table so that it indicated which test cases can be used to satisfy MC/DC for each condition.

A B	result
T T	T
T F	F
F T	F
F F	F



number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		



# Creating Test Cases Cont'd

---

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

- MC/DC for **A**:
  - Take 1 + 3
- MC/DC for **B**:
  - Take 1 + 2
- Resulting test cases are
  - 1 + 2 + 3
  - (T , T) + (T , F) + (F , T)

# More Advanced Example

**if (A && (B || C)) then...**

number	ABC	result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

Note: We want to determine the MINIMAL set of test cases

Here:

{2,3,4,6}

{2,3,4,7}

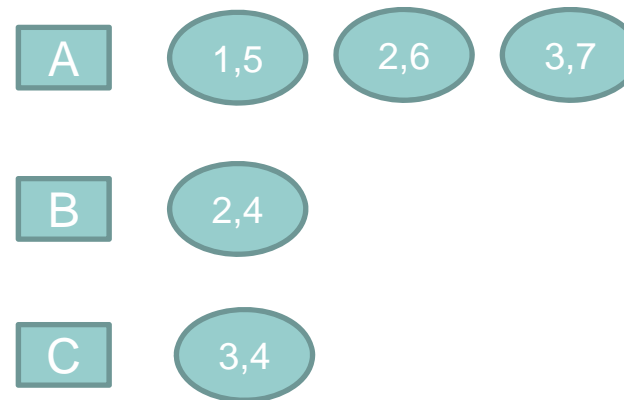
Non-minimal set is:

{1,2,3,4,5}

# Finding MC/DC Adequate Test Cases

- An special (and easy) case of the classic **Boolean satisfiability problem**

number	ABC	result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			



$$A = (x1 \wedge x5) \vee (x2 \wedge x6) \vee (x3 \wedge x7)$$

$$B = (x2 \wedge x4)$$

$$C = (x3 \wedge x4)$$

$$A \wedge B \wedge C =$$

$$(x1 \wedge x2 \wedge x3 \wedge x4 \wedge x5) \vee \\ (x2 \wedge x3 \wedge x4 \wedge x6) \vee \\ (x2 \wedge x3 \wedge x4 \wedge x7)$$

Make as few as possible variables among  $x1, x2, \dots, x7$  to be true while at the same time make  $A \wedge B \wedge C$  to be true.

# Coupled Conditions

---

For a condition such as:

$$(x > 0 \ \&\& \ z < 0) \ || \ (y > 0 \ \&\& \ x > 0)$$

, it is not possible to keep the first occurrence of  $x > 0$  fixed while varying the value of its second occurrence of  $x > 0$ .

Here the first occurrence of A is said to be coupled to its second occurrence.

B is coupled to A when changing B will also affect A

Another example:  $x > 0 \ \&\& \ x \geq 0$

The requirement of “being fixed” can be dropped on the second occurrence when finding MC/DC for the first occurrence.

# Quiz 4: MC/DC coverage

---

- Find MC/DC adequate test suite for the following decision:
  - $( (a \parallel b) \ \&\& \ c \parallel d ) \ \&\& \ e$

a	b	c	d	e	result
<u>T</u>	F	<u>T</u>	F	<u>T</u>	T
F	<u>T</u>	T	F	T	T
T	F	F	<u>T</u>	T	T
T	F	T	F	<u>F</u>	F
T	F	<u>F</u>	<u>F</u>	T	F
<u>F</u>	<u>F</u>	T	F	T	F

# Comments on MC/DC

- Linear growth in required tests:
  - For an expression with  $N$  uncoupled conditions, MC/DC can be satisfied by a minimum of  $N + 1$  test cases and a maximum of  $2N$  test cases.
- Stronger adequacy criteria:
  - MC/DC ensures a much higher level of coverage than branch or condition coverage.
  - Federal Aviation Administration's requirement that test suites be MC/DC adequate.
  - In the avionics domain, complex Boolean expressions are much more common.

$N$	1	2	3	4	5	6-10	11-15	> 15
Software Tools	446	72	9	0	0	0	0	0
Booch Components	9048	402	52	0	0	0	0	0
EFIS avionics display	1343	182	38	16	18	11	3	0

Fig. 4 Complexity of expressions in representative domains

# Multiple Condition Coverage

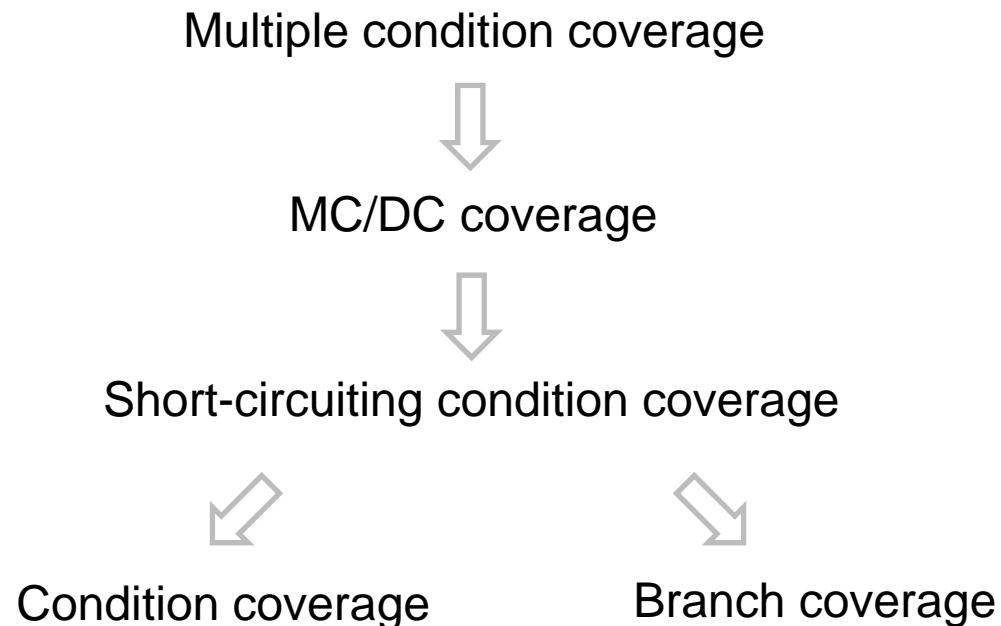
---

- Cover *all possible combinations* of conditions.
  - Some combinations are not possible because coupled conditions. e.g.  $(x > 0 \ \&\& \ x > 0)$
  - If a decision D has  $k$  uncoupled conditions, the total number of combinations  $2^k$
  - Not practical for expression with a large amount of conditions.

# Subsumption Relation of Logic Coverage

---

- We say that coverage criterion A subsumes coverage criterion B if test suite that satisfies A must at the same time satisfies B.





# Practical Coverage Criteria

---

- Whitebox (白盒测试充分性标准)
  - Control-flow coverage (控制流覆盖)
  - Logic coverage (逻辑覆盖)
  - Data-flow coverage (数据流覆盖)
  - Interface coverage (接口覆盖)
  - Mutant coverage (变异覆盖)
- Blackbox (黑盒测试充分性标准)

# What is *dataflow*?

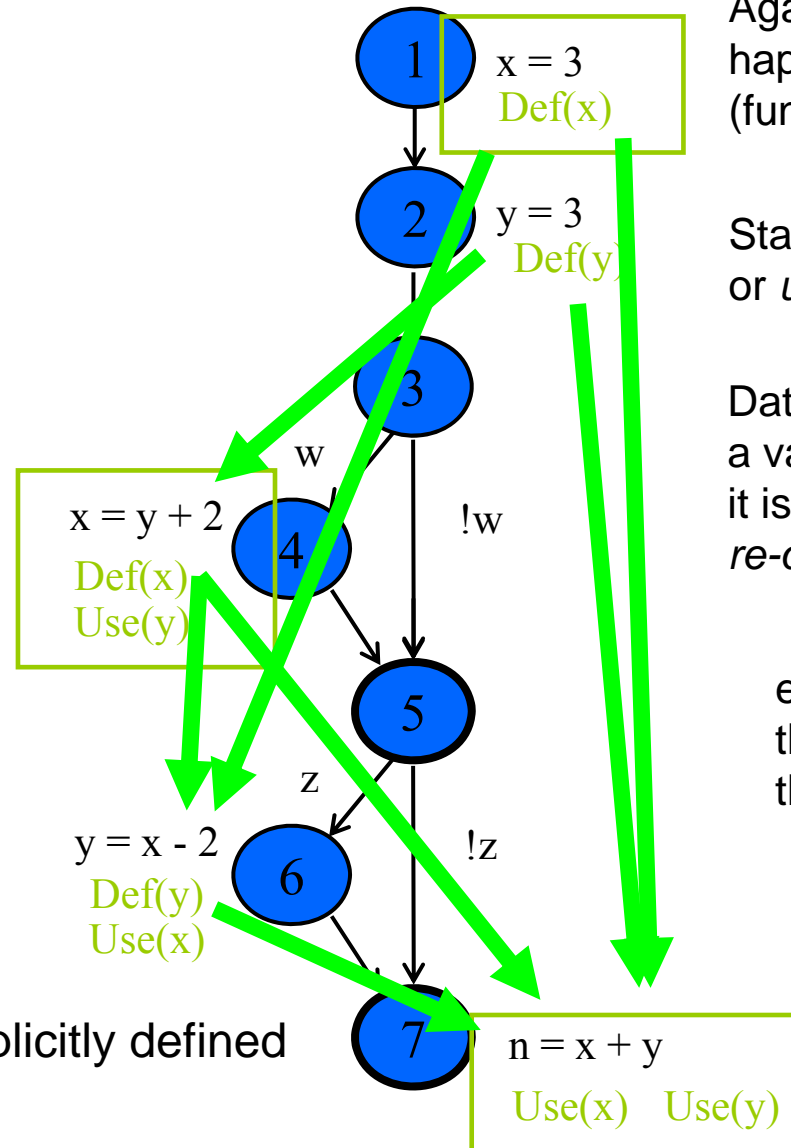
```

x = 3;
y = 3;

if (w) {
  x = y + 2;
}

if (z) {
  y = x - 2;
}

n = x + y
  
```



Again, we only care about what happens in one single code unit (function or method).

Statements can *defined* or *used* variables

Data flow occurs from where a variable is *defined* to where it is *used*, without any intervening *re-definitions*

e.g., in this path: 1 2 3 5 6 7, the definition of  $x$  at 1 reaches the use of  $x$  at 7.

In this path: 1 2 3 **4** 5 6 7 the definition of  $x$  at 1 fail to reach the use of  $x$  at 7 because of the redefinition at 4

The dataflow is implicitly defined by the CFG

# Operations on Variables

---

- **Definition:** A value is written into a variable
  - **Example:** `x=0; def(x)`
  - **Example:** `*y=1; def(*y)`
  - **Example:** `scanf("%d %d", &x, &y); def(x), def(y)`
  - **Example:** `int x[10]; def(x), def(x[0]), ...def(x[9])`
- **Use:** The value of the variable is read
  - **Example:** `y=x+1; use(x)`
  - **Example:** `*y=1; use(y)`
  - **Example:** `k=*y+1; use(y), use(*y)`
- **Undefinition:** A variable is not longer accessible
  - **Example:** `free(p); undef(*p)`
- A statement can involve more than one operations

# Quiz 5: Def and Use

---

- How many defs and uses in each line?

```
int* p = new int[10];
```

```
p[5] = 0;
```

```
p[5] = p[5]+1;
```

```
int* p = new int[10]; //def(p), def(p[0])... def(p[9])
```

```
p[5] = 0; // use(p), def(p[5])
```

```
p[5] = p[5]+1; // use(p), use(p[5]), def(p[5])
```

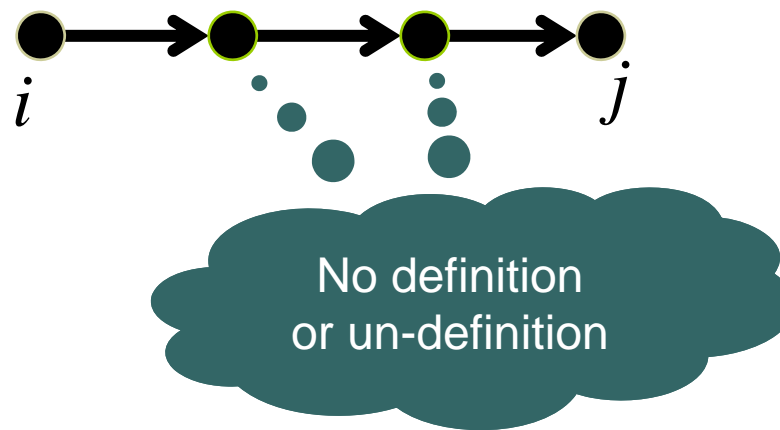
# Two Types of use

---

- **Computation use (c-use):** The value of the variable directly affects a computation or is part of an output
  - **Example 1:** `y = x + 1;`      c-use(x)
  - **Example 2:** `printf("%d",x);`      c-use(x)
- **Predicate use (p-use):** The value of the variable directly affects a control flow (and may indirectly affect a computation)
  - **Example 1:** `if (x == 0) { y = 1; }`      p-use(x)
  - **Example 2:** `switch(c) {case 1:... default:...}`      p-use(c)
  - **Example 3:** `if(A[i+1]>0)return;`      p-use(A), p-use(i),  
p-use(A[i+1])

# Def-Clear Path

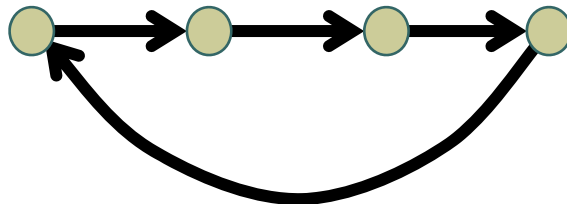
- ◆ A *definition clear path with respect to  $x$*  (def-clear path wrt  $x$ ) is a path  $(i, n_1, n_2, \dots, n_m, j)$ , where  $m \geq 0$ , containing no  $\text{def}(x)$  or  $\text{undef}(x)$  in nodes  $n_1, n_2, \dots, n_m$ 
  - Node  $i$  and  $j$  might or might not contain  $\text{def}(x)$  or  $\text{undef}(x)$ .



# Simple Path

---

- A ***simple path*** is one in which all nodes, except possibly the first and last, are distinct.
- **Example:**



# Loop-Free Path

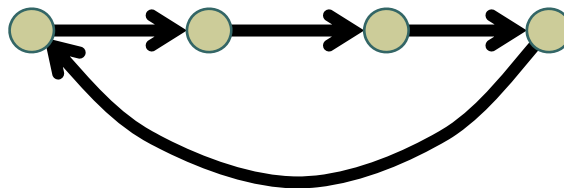
---

- A **loop-free path** is a simple path in which all nodes are distinct.
- **Example:**

Loop free path:



Not a loop free path:



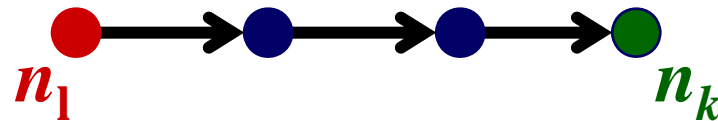


# Du-Path

- ◆ A path  $(n_1, n_2, \dots, n_k)$  is a **du-path for variable  $x$**  if  $n_1$  define  $x$  and
  - **either**  $n_k$  has a **c-use** of  $x$  and  $(n_1, n_2, \dots, n_k)$  is a def-clear **simple path** with respect to  $x$

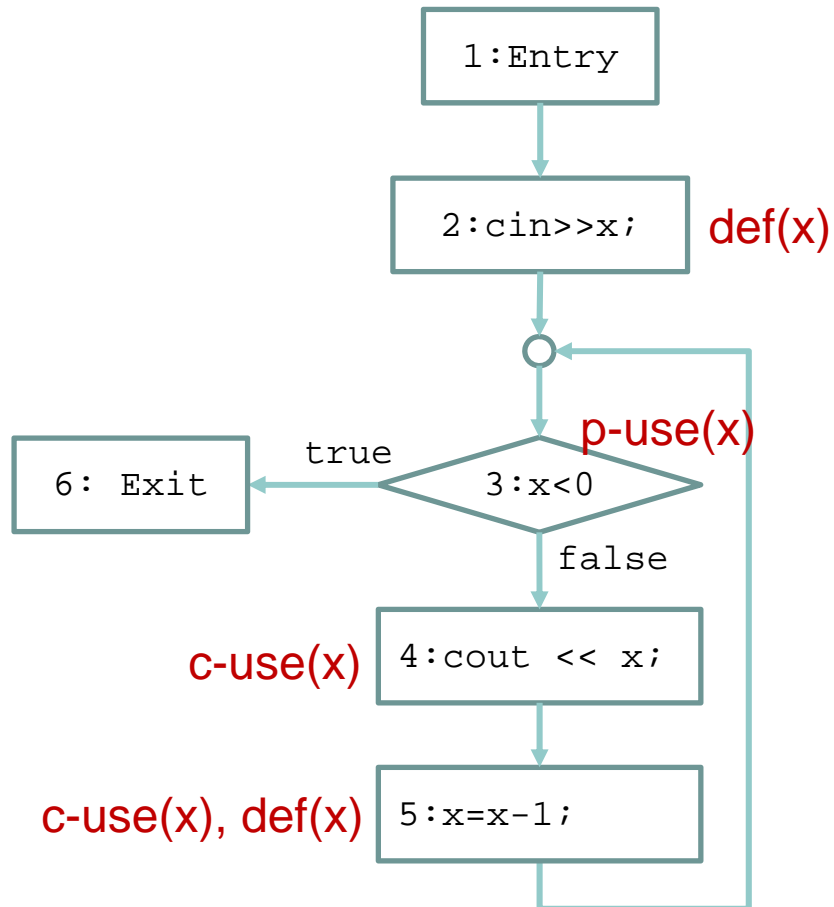


- **or**  $n_k$  has a **p-use** of  $x$  and  $(n_1, n_2, \dots, n_k)$  is a def-clear **loop-free path** with respect to  $x$ .



**Any CFG can have only finite number of du-paths. Why?**

# Example



## du-path for x:

2, 3  
 2, 3, 4  
 2, 3, 4, 5  
 5, 3  
 5, 3, 4  
**5, 3, 4, 5**

## Not du-path for x:

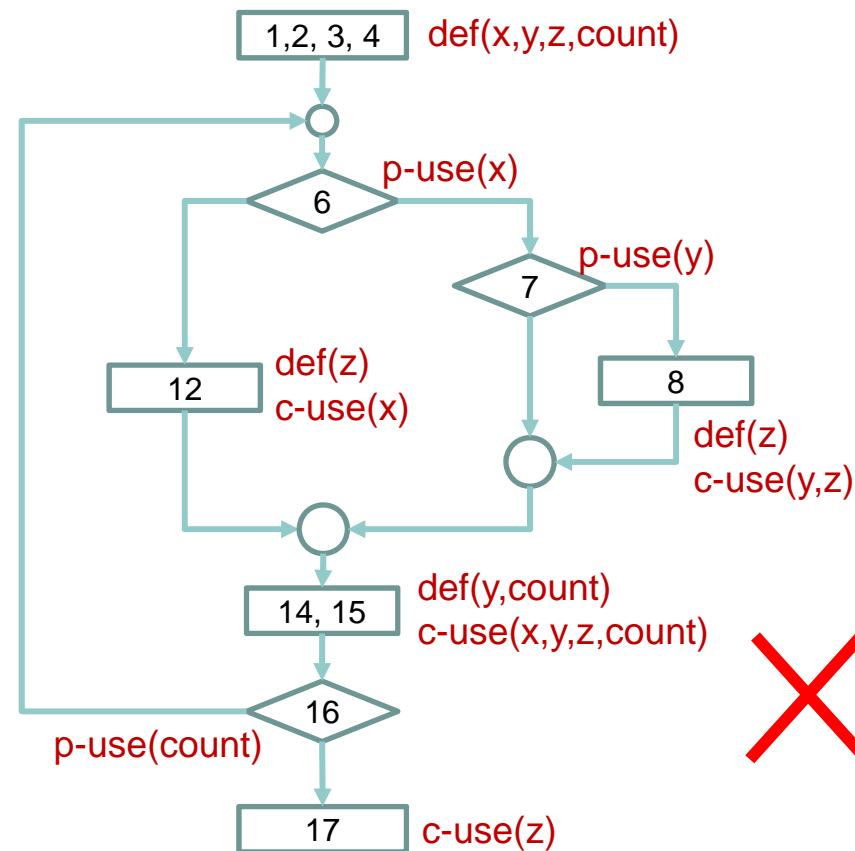
4, 5  
 3, 4, 5, 3  
 2, 3, 4, 5, 3  
 5, 3, 4, 5, 3

# Quiz 6: c-use, p-use, def, and du-path

```

1  begin
2    float x, y, z=0.0;
3    int count;
4    input (x, y, count);
5    do {
6      if (x≤0) {
7        if (y≥0) {
8          z=y*z+1;
9        }
10     }
11     else{
12       z=1/x;
13     }
14     y=x*y+z
15     count=count-1
16     while (count>0)
17     output (z);
18  end

```



**du-path for x:**

1,2,3,4,6  
 1,2,3,4,6,12  
 1,2,3,4,6,12,14  
 1,2,3,4,6,7,14  
 1,2,3,4,6,7,8,14  
 1,2,3,4,6,12,14,15,16,17  
 1,2,3,4,6,7,14,15,16,17  
 1,2,3,4,6,7,8,14,15,16,17



Is the following path a du-path for x ?

1,2,3,4,6,12,14,15,16,6

Mark c-use, p-use, and def on nodes in the CFG  
Then find **all** du-paths for x.

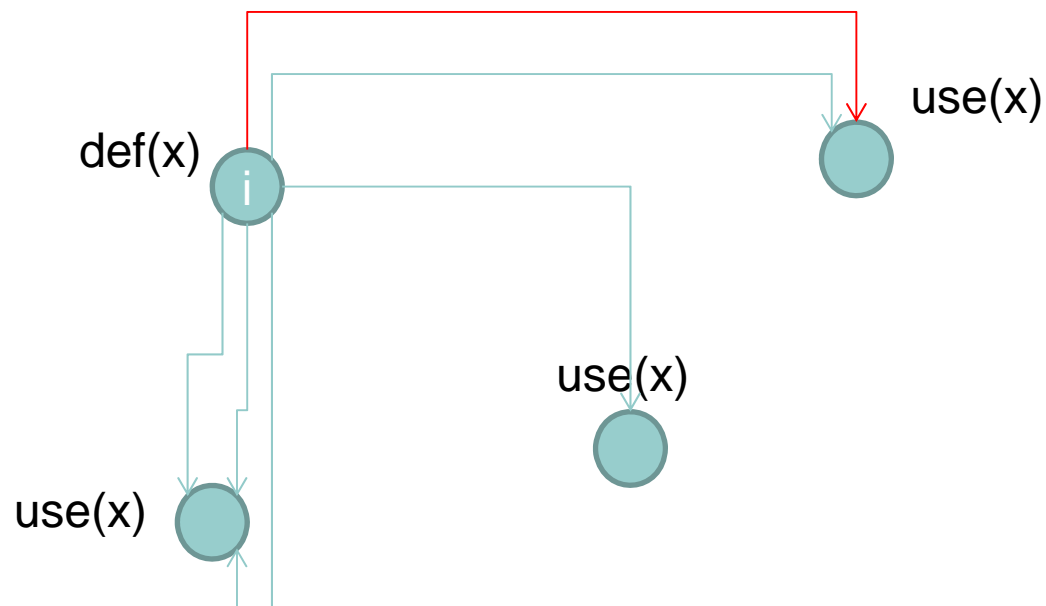
# Dataflow Coverage

---

- Concerns different set of du-paths that we need to cover:
  - All-defs
  - All-uses
  - All-du-paths
  
  - All-p-uses
  - All-c-uses
  - All-p-used/some-c-uses
  - All-c-uses/some-p-uses

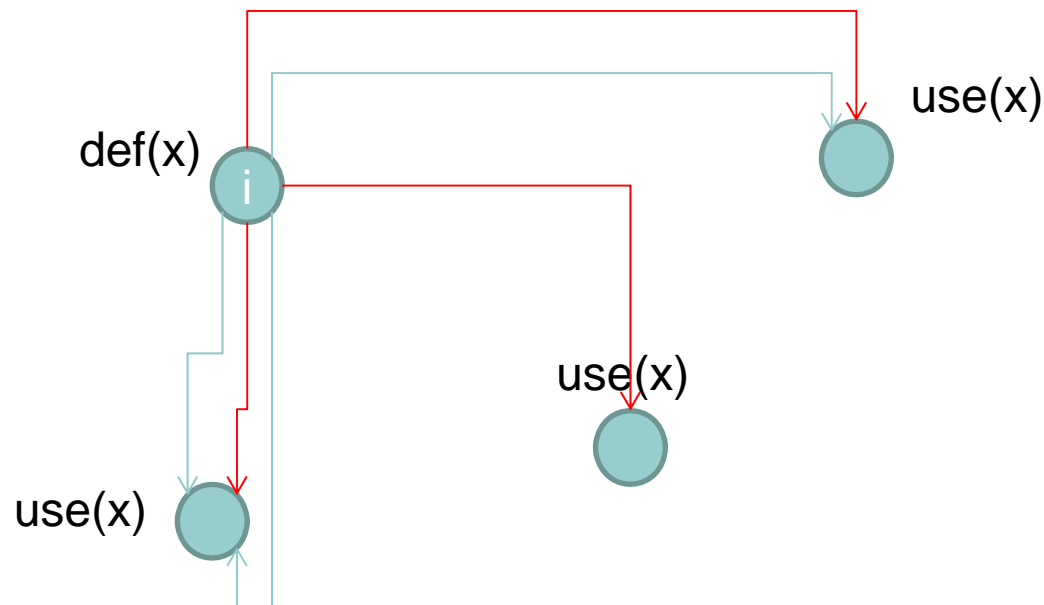
# All-Defs Coverage

- For each node  $i$  that contains a definition to variable  $x$ , the **all-defs coverage** requires that the test suite covers **at least one** feasible du-path of  $x$  that starts from  $i$ .
  - If none of such du-paths for  $x$  are feasible or there is no du-paths for  $x$  at all (e.g. define but never use), we don't need to cover anything.



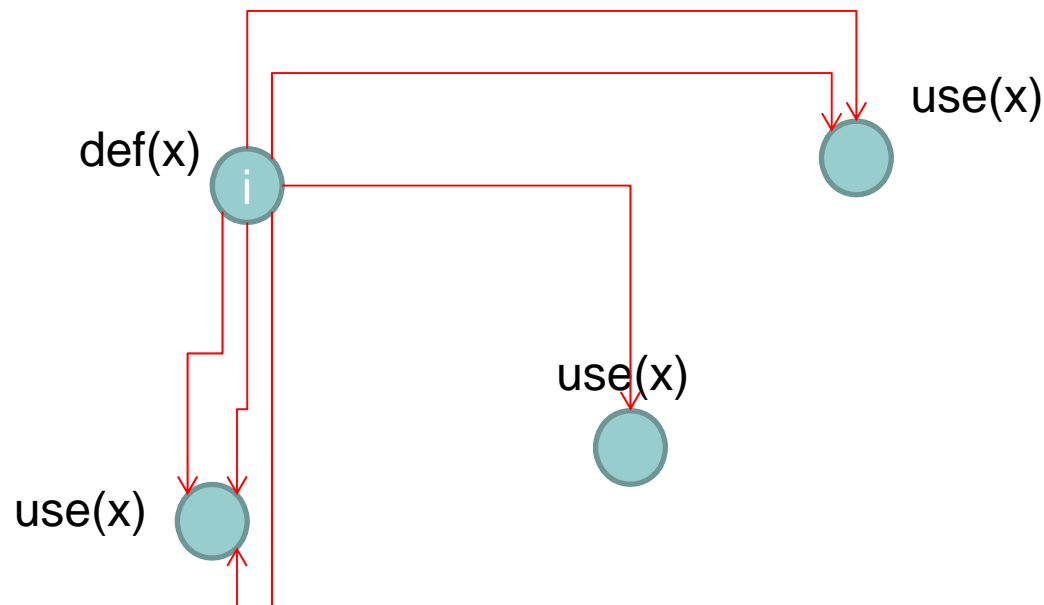
# All-Uses Coverage

- For each node  $i$  in the CFG that contains a definition to variable  $x$ , **all-uses coverage** requires that for **every** use of  $x$  in the CFG, the test suite covers **at least one** feasible du-path of  $x$  that starts from  $i$  and ends at that use of  $x$ .
  - Again, if none of the du-paths of  $x$  that starts from  $i$  to that use of  $x$  are feasible, we don't need to cover anything.



# All-Du-Paths Coverage

- For each node  $i$  in the CFG that contains a definition to variable  $x$ , the **all-du-paths coverage** requires that for **every** use of  $x$  in the CFG, the test suite covers **all** feasible du-path of  $x$  that starts from  $i$  and ends at that use of  $x$ .



# More Data Flow Coverage

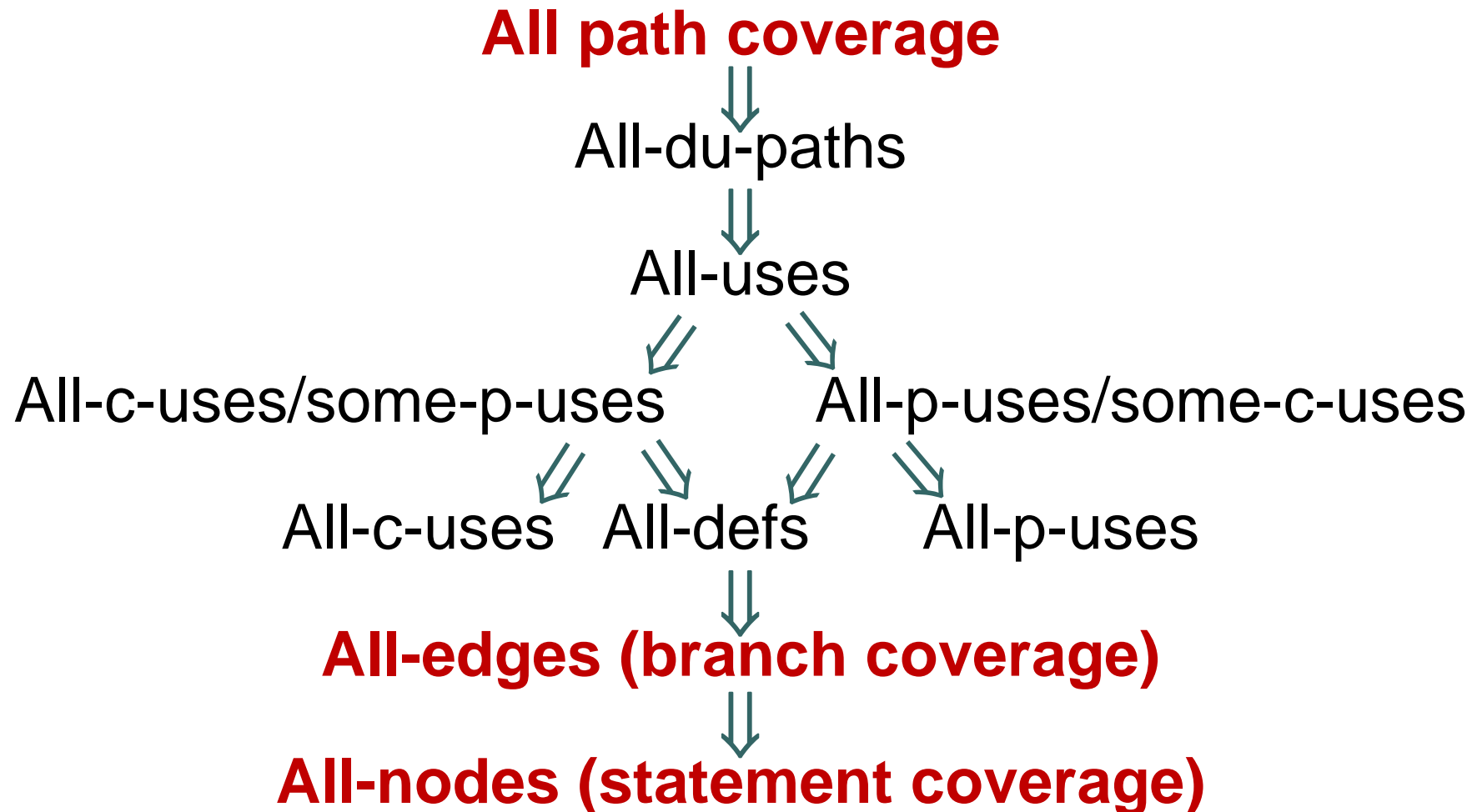
---

- The *all-p-uses*, *all-c-uses*, *all-p-used/some-c-uses* and *all-c-uses/some-p-uses* coverage place emphasis on either c-uses or p-uses.
  - **Example:** *all-c-uses/some-p-uses* – requires that for every c-use of x in the CFG, the test suite covers at least one feasible du-path of x that starts from i and ends at that use of x. Besides, it requires that the test suite covers at least one feasible du-path of x that start from i and ends at any p-use of x, if such p-uses exist.



# Subsumption Relationships

---



# Complexity in Dataflow Coverage

---

- **Function call**

- Some statements might call other functions. For example:

```
1: x = 0;
2: foo(x);
3: if(x == 0){...}
```

Is  $1 \rightarrow 2 \rightarrow 3$  a du-path for  $x$ ?

- It depends on what happens in `foo`.

- Case 1: `void foo(int x){...}`
    - $1 \rightarrow 2 \rightarrow 3$  is a feasible du-path for  $x$ .
  - Case 2: `void foo(int& x){x=1;return;}`
    - $1 \rightarrow 2 \rightarrow 3$  is **not** a feasible du-path for  $x$ .
  - Case 3 (yes): `void foo(int& x){if(x>0)x++;return;}`
    - $1 \rightarrow 2 \rightarrow 3$  is still a feasible du-path for  $x$ .

- **The key issue:** for some statements, the set of defs and uses might change in different executions.

- When deciding whether a du-path is feasible, we only need one of them to satisfy the requirements.

# Complexity in Dataflow Coverage

- Data structure
  - Need to treat each structure or array *element* independently.
  - Ditto class/object.
- Problem:
  - A statement can refer to different array elements in different execution. For example:

```
1: A[i] = 0;  
2: A[j] = 1;  
3: if(A[k] == 0){...}
```

Is  $1 \rightarrow 2 \rightarrow 3$  a feasible du-path for A[i]?

It is not if i and k are always different, or i and j are always the same.

But this is difficult to know with static analysis.

General guideline: prefer over-estimation. Always assume it as feasible.

# A Similar Problem

---

- The use of pointer

```
1: *p = k+1;
```

```
...
```

```
2: if( *q == 0 );
```

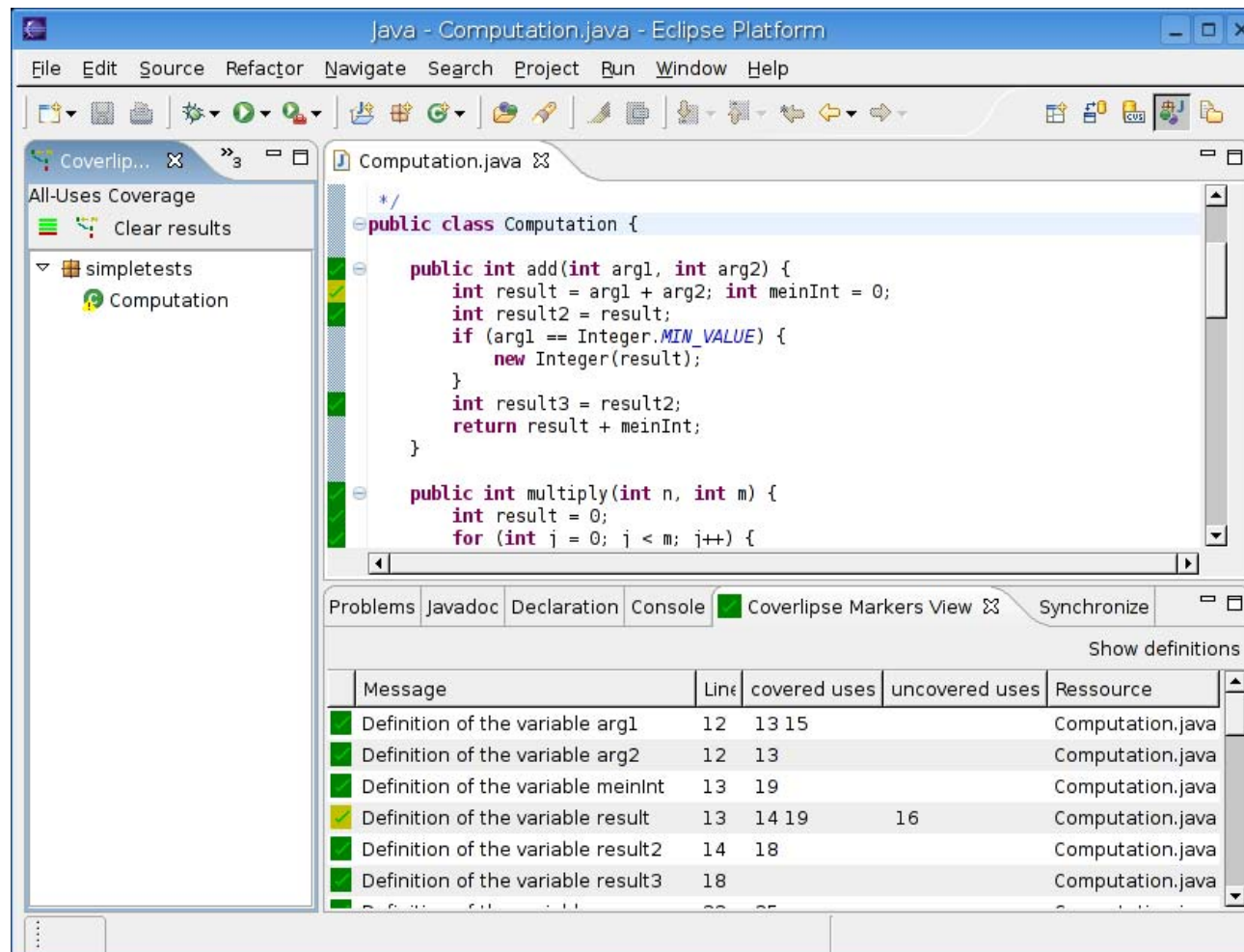
Do p and q point to the same memory location?

- In this case, the problem is more severe:
  - Are we going to assume all `int*` pointers in the source code can point to the same memory location? This can result in millions of du-paths!
- The problem can be partially addressed by **pointer alias analysis**.
  - Pointer alias analysis tells us, for each code location, which pair of pointers can point the same memory location, which pair cannot.
  - One of the fundamental techniques in program analysis course.

# Tool: Coverlipse

<http://coverlipse.sourceforge.net/>

- Support all-use coverage



# Practical Coverage Criteria

---

- Whitebox (白盒测试充分性标准)
  - Control-flow coverage (控制流覆盖)
  - Logic coverage (逻辑覆盖)
  - Data-flow coverage (数据流覆盖)
  - Interface coverage (接口覆盖)
  - Mutant coverage (变异覆盖)
- Blackbox (黑盒测试充分性标准)

# What we have learnt

---

- Code coverage for *one single code unit* (function or method)
  - Control-flow coverage
  - Data-flow coverage
  - Logic coverage
- How about coverage for *interaction between code units*?
  - Unit testing  $\Rightarrow$  Integration testing
  - Occurs at the *interface* (输入输出接口) of the code units. Thus the name “**interface coverage**”

# Interface Coverage

---

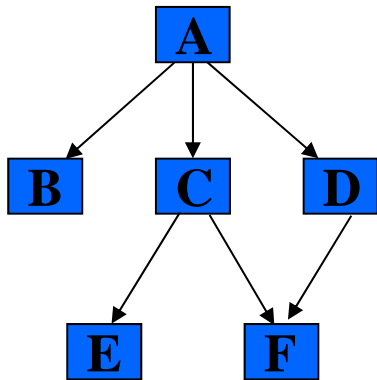
- **Function/call coverage**（方法/调用覆盖）
- Inter-procedural dataflow coverage（过程间数据流覆盖）
- Object-oriented coverage（面向对象覆盖）



# Function/Call Coverage

- **Call Graph**

- Nodes : Functions (or methods)
- Edges : Calls to functions (*call-site*)
- Compare it with control-flow graph (CFG)



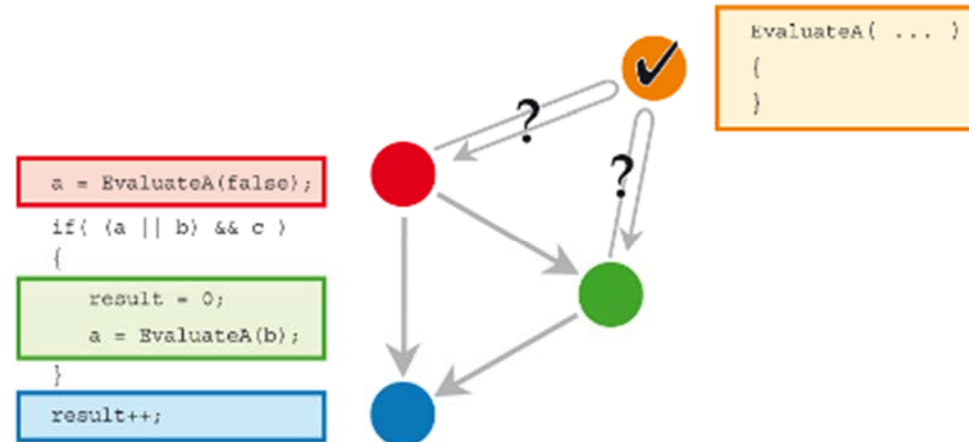
**Example call graph**

**Function coverage:** cover every function at least once (every node in the call graph)

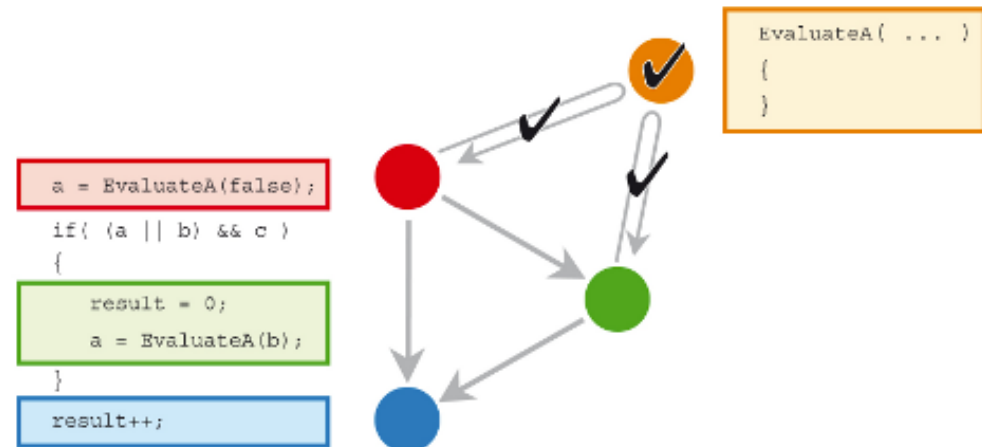
**Call coverage:** cover every call-site at least once (every edge in the call graph)

# Example

*Function coverage*



*Call coverage*



# Complexity in Function/Call Coverage

---

- Function overloading (函数重载)
- Function pointer (函数指针)
- Polymorphism (多态)

# Overloading

---

- **[Static binding issue]** Invoke which overloading function at the call-site?

```
int square(int x) { return x*x; }
```

```
double square(double y) { return y*y; }
```

```
float square(float z) { return z*z; }
```

```
int foo(char c){  
    ...  
    int result = square(c) // Call which function?  
    ...  
}
```

# Function Pointer

---

- **[Dynamic binding issue]** Cover every possible function instance at the function-pointer call-site.

```
int (*funcPointer) (int, char, int);
```

```
int firstExample ( int a, char b, int c){
    printf(" Welcome to the first example");
    return a+b+c;
}
```

```
int secondExample ( int a, char b, int c){
    printf(" Welcome to the second example");
    return a*b*c;
}
```

```
int foo(funcPointer func){
    ...
    int answer= func(7, 'A' , 2 );
    ...
}
```

Need to cover both two instants:  
 func==firstExample or  
 func==secondExample

Problem:  
 How to know the possible instants?

Solution:  
 Conservative enumeration.

# Polymorphism

---


- **[Dynamic binding issue]** Cover every possible method instance at the virtual-method call-site.

```
abstract class Animal {  
    String talk();  
}  
  
class Cat extends Animal {  
    String talk() { return "Meow!"; }  
}
```

```
class Dog extends Animal {  
    String talk() { return "Woof!"; }  
}
```

```
void write(Animal a) {  
    System.out.println(a.talk());  
}
```

Need to cover both two instants:  
Cat.talk() or Dog.talk()



# Integration Coverage

---

- Function/call coverage（方法/调用覆盖）
- **Inter-procedural dataflow coverage（过程间数据流覆盖）**
- Object-oriented coverage（面向对象覆盖）

# Inter-procedural Data Flow

---

- *Data flow across units* are more complicated than *data flow within units*
  - When values are passed, they “change names”
  - Occur in many different ways (e.g. parameters, global variables, heap content, external files)
- Some definitions:
  - **Caller**: A unit that invokes another unit
  - **Callee**: The unit that is called
  - **Call-site**: Statement or node where the call appears
  - **Actual parameter** (实参) : Expression in the caller
  - **Formal parameter** (形参) : Variable in the callee



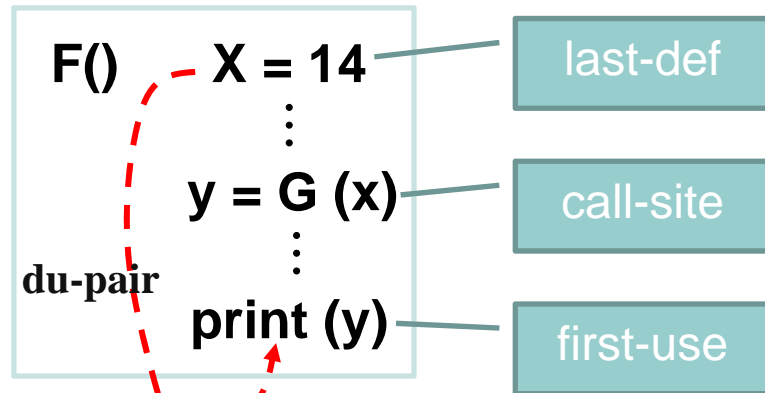
# Last-def and First-use

---

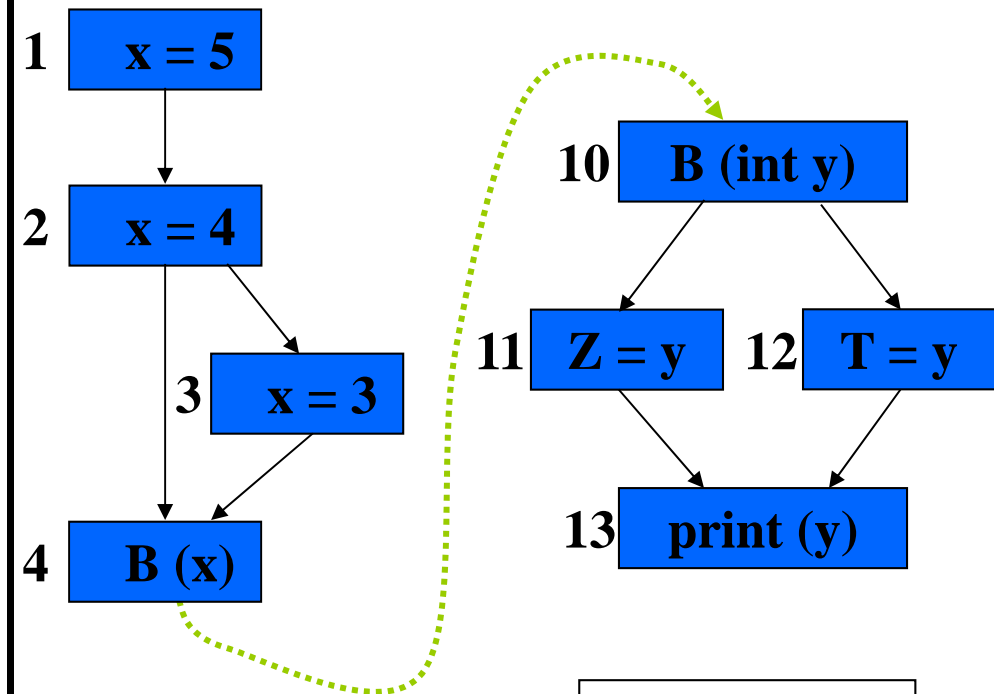
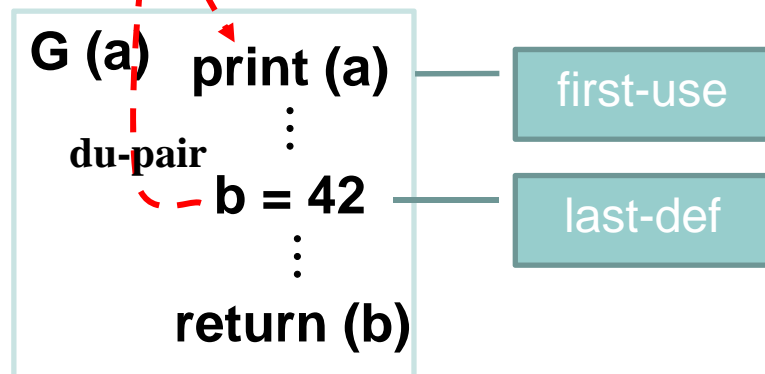
- We further restrict our focus to the *last definitions* of variables before calls/returns and the *first uses* of variables after calls/returns.
- **Last-def** : The definition that has at least one def-clear path from this definition to the call-site.
- **First-use**: The use that has at least one def-clear and use-clear path from the call-site to this use.

# Example

## Caller



## Callee



**Last Defs**  
2, 3

**First Uses**  
11, 12

# Inter-procedural Dataflow Coverage

---

- Similar to intra-procedural data flow coverage, except that:
  - we now look into the code of the callee; and
  - we only consider du-paths that start from last-defs and ends at first-uses.

# Inter-procedural Dataflow Coverage

---

- For each node  $i$  that contains a last-def to variable  $x$ , the ***all-inter-procedural-def coverage*** requires that the test suite covers at least one feasible du-path of  $x$  that starts from  $i$  and ends at any of the first-use of  $x$  in the immediate caller/callee.
- For each node  $i$  that contains a last-def to variable  $x$ , ***all-inter-procedural-use coverage*** requires that for every first-use of  $x$  in the immediate caller/callee, the test suite covers at least one feasible du-path of  $x$  that starts from  $i$  and ends at that first-use of  $x$ .

# Quiz 7: Inter-procedural Dataflow

```

int main(int argc, char**argv){
1:   int X, Y, Z
2:   int Ok, R1, R2;
3:   if ( argc == 3 ){
4:       X = atoi(argv[0]);
5:       Y = atoi(argv[1]); last-defs(X,Y,Z)
6:       Z = atoi(argv[2]);
7:   }else{
8:       X = 10;
9:       Y = 9;  last-defs(X,Y,Z)
10:      Z = 12;
11:  };
12:
13:  OK = 1; last-defs(OK)
14:  root (X, Y, Z, R1, R2, OK);
15:  if(OK)printf("%d %d", R1, R2); first-use(OK, R1, R2)
16:  else printf("No solution.");
}

void root(double A, double B, double C,
          double& Root1, double& Root2,
          int& result){
1:   double D;
2:   D = B*B-4.0*A*C; first-use(A,B,C)
3:   if(Result && D < 0.0){ first-use(Result)
4:       Result = 0; last-def(Result)
5:       return;
6:   } last-defs(Root1,Root2)
7:   Root1 = (-B + sqrt(D)) / (2.0*A);
8:   Root2 = (-B - sqrt(D)) / (2.0*A);
9:   Result = 1;
10:  return; last-def(Result)
}

```

What are the last-defs and first-uses?

What are the inter-procedural du-paths between them?

main.5, main.6, main.13, main.14, root.2 (for main.Y)

main.9, main.13, main.14, root.2(for main.Y)

root.7, root.8, root.9, root.10, main.14, main.15 (for main.R1)

# Def-Use Coverage on smart\_programmer

---

1. intra-procedural all-use coverage:  
     e.g. root.c:2 -> root.c:7  
     e.g. main.c:4 -> main.c:14
2. inter-procedural all-use coverage through pointer or global variable only.  
     e.g. root.c:7 -> main.c:15

```

int main(int argc, char**argv){
1:   int X, Y, Z
2:   int Ok, R1, R2;
3:   if ( argc == 3 ){
4:       X = atoi(argv[0]);
5:       Y = atoi(argv[1]);
6:       Z = atoi(argv[2]);
7:   }else{
8:       X = 10;
9:       Y = 9;
10:      Z = 12;
11:  };
12:
13:  OK = 1;
14:  root (X, Y, Z, R1, R2, OK);
15:  if(OK)printf("%d %d", R1, R2);
16:  else printf("No solution.");
}

```

```

void root(double A, double B, double C,
          double& Root1, double& Root2,
          int& result){
1:   double D;
2:   D = B*B-4.0*A*C;
3:   if(Result && D < 0.0){
4:       Result = 0;
5:       return;
6:   }
7:   Root1 = (-B + sqrt(D)) / (2.0*A);
8:   Root2 = (-B - sqrt(D)) / (2.0*A);
9:   Result = 1;
10:  return;
}

```

# Integration Coverage

---

- Function/call coverage（方法/调用覆盖）
- Inter-procedural dataflow coverage（过程间数据流覆盖）
- Object-oriented coverage（面向对象覆盖）

# OO Designs and OO Coverage

---

- Object-oriented Design: Emphasis on modularity
  - Design units: *classes, interfaces, methods*, etc.
  - Complexity in the connections between design units
  - Connections are dependency relations, also called *couplings*
- **Object-oriented coverage**: cover connections between design units
  - OO coverage is defined using graphs that model the connections between design units.



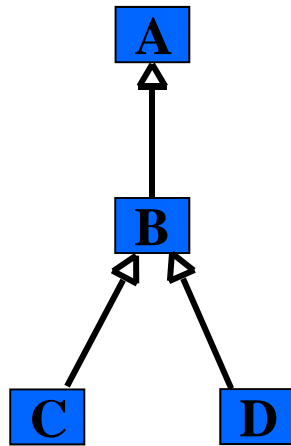
# Typical OO aspects

---

- Inheritance: method overriding, variables shadowing, constructors overriding works differently
- Polymorphism
  - a required object of type T can be instantiated by a subclass at the runtime
  - overloading
- In addition to object level members we also have class level members (the '*static*').

# Call Graph is Insufficient for OO

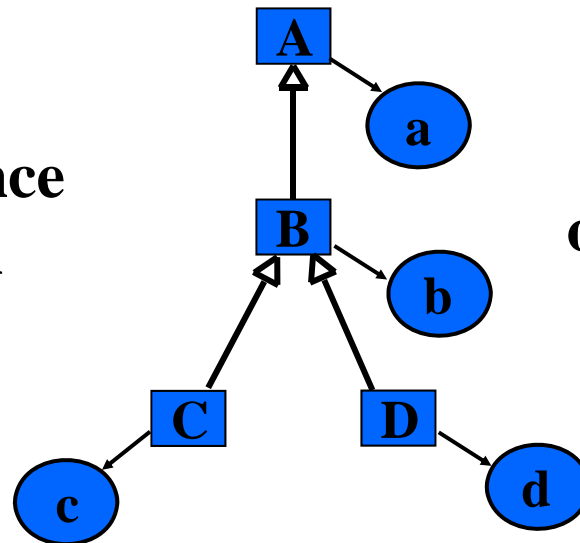
- Example: Inheritance & Polymorphism



**Example inheritance  
hierarchy graph**

**Classes are not executable, so  
this graph is not directly testable**

**We need objects**



**objects**

**What is coverage  
on this graph ?**

# Coverage on Inheritance Graph

---

- Create an object for each class and apply call coverage.

OO Method Coverage: For each method, call it with objects of all possible classes.

OO Call Coverage: For each method, call it with objects of all possible classes at all possible call-sites.

# OO Data Flow

---

- The defs and uses could be in the same class, or different classes
- Researchers have applied data flow testing to the direct coupling OO situation
  - Has not been used in practice
  - No tools available

# Practical Coverage Criteria

---

- Whitebox (白盒测试充分性标准)
  - Control-flow coverage (控制流覆盖)
  - Logic coverage (逻辑覆盖)
  - Data-flow coverage (数据流覆盖)
  - Interface coverage (接口覆盖)
  - Mutation coverage (变异覆盖)
- Blackbox (黑盒测试充分性标准)

# Use Mutants to Measure Coverage: the Idea



- Suppose we have a big bowl of marbles. How can we estimate their number?
  - We don't want to count them one by one.
  - Suppose we have another bag of 100 marbles with unique color (say black).
  - What if we mix them?

# Use Mutants to Measure Coverage: the Idea



- We mix 100 black marbles into the bowl
  - Stir well ...
- We draw out 100 marbles at random
  - 20 of them are black
- How many marbles do we have in the bowl originally?

# Mutation Coverage

---

- Now, instead of a bowl of marbles, we have a program **P** with a unknown number of real bugs. And we have a test suite T.
- We **add** 100 artificial bugs:
  - Assume they are exactly like real bugs in every way.
  - We make 100 copies of P, each with one of the 100 artificial bugs.
  - We call these copies as *mutants*.
- We run the test cases in T on the 100 mutants...
  - 20 of mutants are *killed*. That is, at least one of the test cases can detect the difference between **P** and the mutant (e.g. different output results).
  - The other 80 mutants are not killed.
- What can we infer about our test suite?
  - Mutant coverage: 20%



# How to kill a mutant?

```
P(x) {
    if (x<0)  x++ ;
    return x;
}
```

```
P(x) {
    if (x≤0)  x++ ;
    return x;
}
```

```
test() {
    r = P(0) ;
    assert (r≥0)
}
```

- (*reachability*) you need a test-case whose execution passes the mutation.
- You need a test-case that actually observes the effect of the mutation. But note there is a difference in:
  - (*infection*) “incorrect” state, right after the mutation.
  - (*propagation*) the infection propagates to *incorrect* P’s output. In principle, a test-case can only kill if there is propagation.

# Formal Definition

- **Mutant:** A program with a seeded fault
- **Mutation coverage** =  $D/N$ 
  - D = Number of killed mutants
  - N = Total number of mutants
- **Mutation testing:** use mutant coverage as the test adequacy criteria to design the test suite.

<b>Program P:</b>  <pre>cin&gt;&gt;y&gt;&gt;z; x=y+z; cout&lt;&lt;x;</pre>	<b>Mutant 1:</b>  <pre>cin&gt;&gt;y&gt;&gt;z; x=y*<b>z</b>; cout&lt;&lt;x;</pre>	<b>Mutant 2:</b>  <pre>cin&gt;&gt;y&gt;&gt;z; x=y-<b>z</b>; cout&lt;&lt;x;</pre>	<b>Mutant 3:</b>  <pre>cin&gt;&gt;y&gt;&gt;z; x=y+<b>y</b>; cout&lt;&lt;x;</pre>	<b>test suite:</b> 1: y=2, z=2 2: y=3, z=3  kill mutant 1 kill mutant 2 does not kill mutant 3
--	--	--	--	--

mutant coverage: 66.7%

# Why Mutation Coverage?

---

- Mutation coverage is **the strongest coverage criterion**.
  - It subsumes all the coverage criteria we have learnt.
- Mutation coverage subsumes statement coverage
  - For every statement, create a *mutant that deletes this statement*.
    - If the test suite satisfies mutant coverage, it must satisfy statement coverage, because it cannot kill a mutant if it does not cover the deleted statement.
- Mutation coverage subsumes branch coverage
  - For every conditional-statement, create a *mutant that changes the branch condition to true* and another *mutant that changes it to false*.
    - The test suite cannot kill the former if it does not cover the `false` branch.
- Mutation coverage subsumes loop coverage
  - `for(int i=0; i<=k; i++) {...} ⇒ for(int i=0; i<k; i++) {...}` if the test suite does not cover the **maximum loop count**, it cannot kill this mutant.
  - `for(int i=0; i<=k; i++) {...} ⇒ if(k<0)k=0; for(int i=0; i<=k; i++) {...}` if the test suite does not cover **zero loop count**, it cannot kill this mutant.
- Mutation coverage subsumes all path coverage
  - For every path, extract its path condition C. Then create the mutant by inserting `if(C)delete null;` at the function entry.

# Example

---

- Consider the following function `foo` that is required to return the sum of two integers `x` and `y`. Clearly `foo` is incorrect.

```
int foo(int x, y) {  
    return (x-y); // shall be x+y;  
}
```

- Now suppose that `foo` has been tested using two test cases: `<x=1, y=0>` and `<x=-1, y=0>`
  - These two test cases are adequate with respect to all the coverage criteria we have learnt.
  - But they still fail to reveal the bug.

# Example

- Suppose that the following three mutants are generated from `foo`.

```
int foo(int x, y){
    return (x-y); // shall be x+y;
}
```

M1:

```
int foo(int x, y){
    return (x+y);
}
```

M2:

```
int foo(int x, y){
    return (x-0);
}
```

M3:

```
int foo(int x, y){
    return (0+y);
}
```

Execute the mutants on the two test cases:  $\langle x=1, y=0 \rangle$  and  $\langle x=-1, y=0 \rangle$

Test (t)	<b>foo(t)</b>	M1(t)	M2(t)	M3(t)
t1	1	1	1	0
t2	-1	-1	-1	0
		Live	Live	<b>Killed</b>

# Example

---

The low mutant coverage reveals the inadequacy of our test suite and guide us to seek additional test cases.

Why our test cases cannot distinguish

```
int foo(int x, y){  
    return (x-y); // shall be x+y;  
}
```

from

```
int foo(int x, y){  
    return (x-0); // shall be x+y;  
}
```

?

Reason: we always set y to be 0. Need to test other value of y.

# Generation of Mutants

- **Mutation operators:** creates mutants by making simple changes in the program under test.

Mutant operator	In P	In mutant
Variable replacement	$z = x * y + 1;$	$x = x * y + 1;$ $z = x * x + 1;$
Relational operator replacement	if ( $x < y$ )	if( $x > y$ ) if( $x \leq y$ )
Off-by-1	$z = x * y + 1;$	$z = x * (y + 1) + 1;$ $z = (x + 1) * y + 1;$
Replacement by 0	$z = x * y + 1;$	$z = 0 * y + 1;$ $z = 0;$
Arithmetic operator replacement	$z = x * y + 1;$	$z = x * y - 1;$ $z = x + y - 1;$

# Variants of Mutant Coverage

---

- **Strong mutation coverage**
  - Requires that the test case produces different output results for the program  $P$  and its mutant.
  - That is, we require reachability, infection, and propagation.
- **Weak mutation coverage**
  - Only requires that the test case triggers internal state differences.
  - That is, we require reachability, infection, but not propagation.



# Complexity in Mutation Testing

---

- Scalability
  - Too many mutants.
  - “*Selective mutation operators*”.
- Equivalent mutant
  - A mutant that is semantically equivalent to the original program

```
while...  
...  
i++  
if (i==5)  
    break;
```

```
while...  
...  
i++  
if (i>=5)  
    break;
```

# Tools

---

- C: proteum
  - <http://www.labes.icmc.usp.br/proteum>
- Java: MuJava
  - <http://cs.gmu.edu/~offutt/mujava/>
- Java: Jester
  - <http://jester.sourceforge.net/>

# Thank you!

