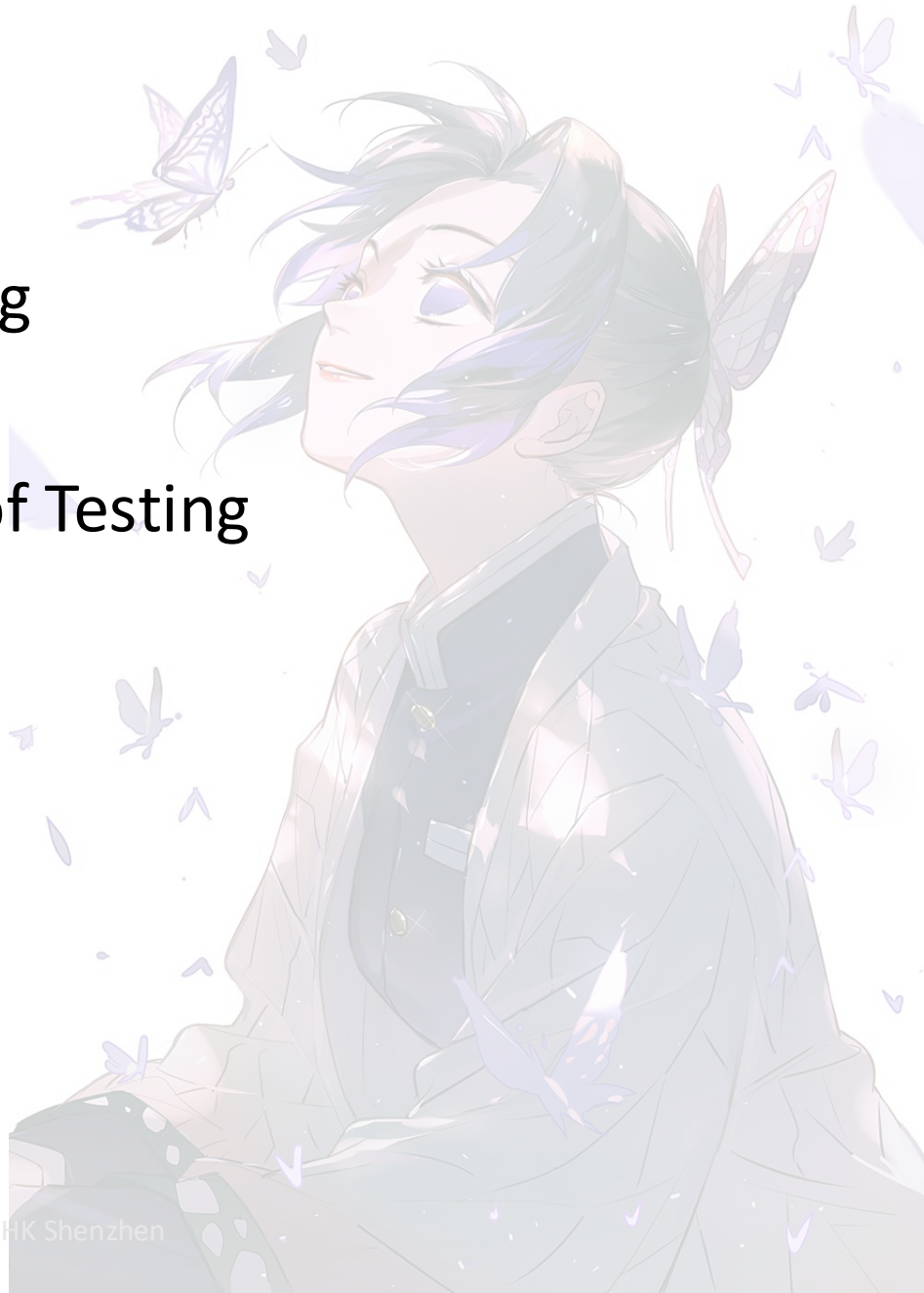
Software Engineering

Testing Basics

CUHK Shenzhen
Pinjia He

1. What is Software Testing
2. Why Software Testing
3. Stages and Categories of Testing
4. Coverage Metrics



What is Software Testing

Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do.

—— IBM

https://www.ibm.com/topics/software-testing

Find as many **bugs** as possible.

The Patriot Accident

- The Patriot missile air defense system tracks and intercepts incoming missiles
- On February 25, 1991, a Patriot system ignored an incoming Scud missile
- 28 soldiers died; 98 were injured

History of Software Testing

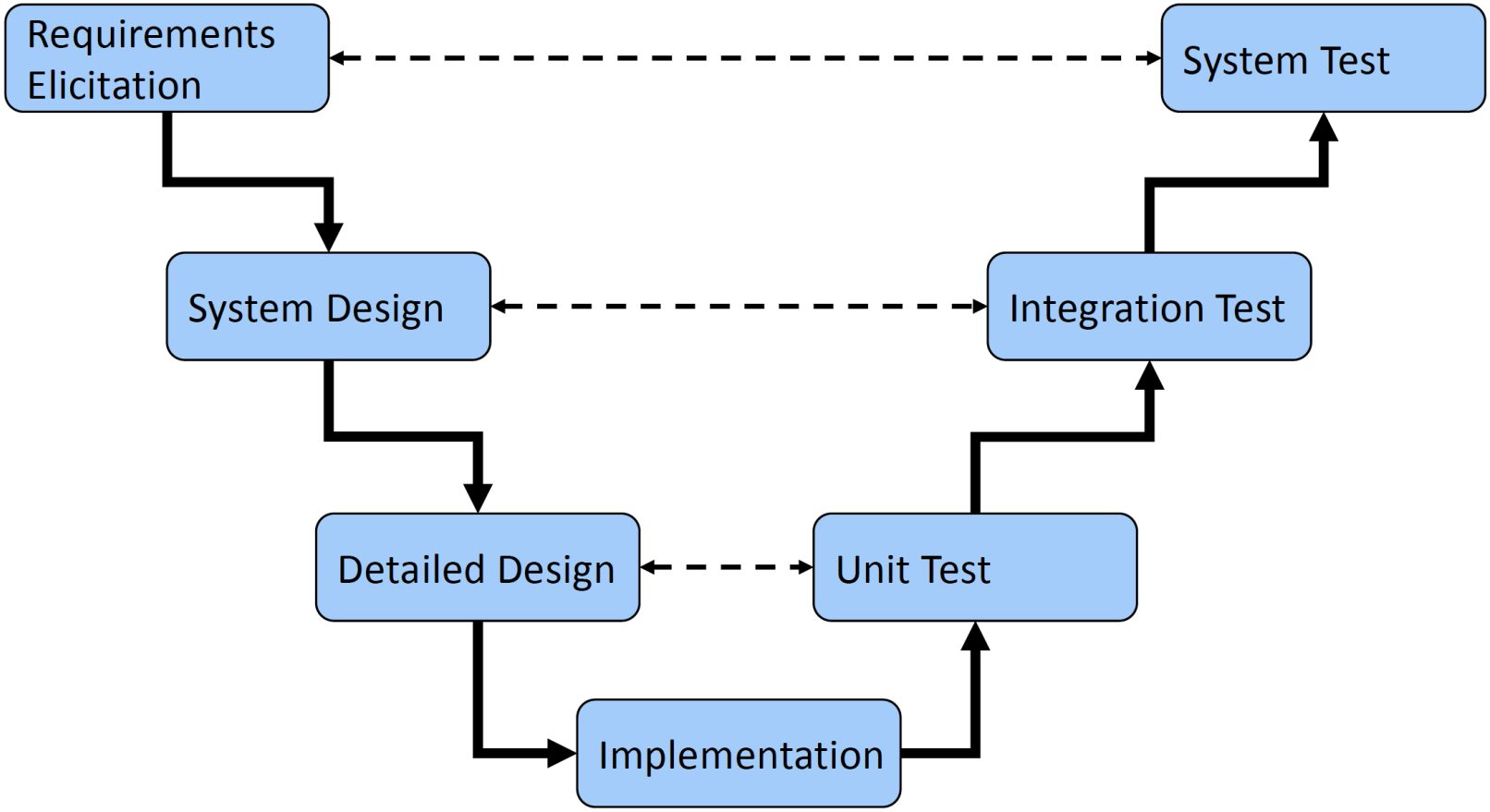
- Debugging era
- Demonstration era
- Destruction era
- Evaluation era
- Prevention era

Testing and Analysis

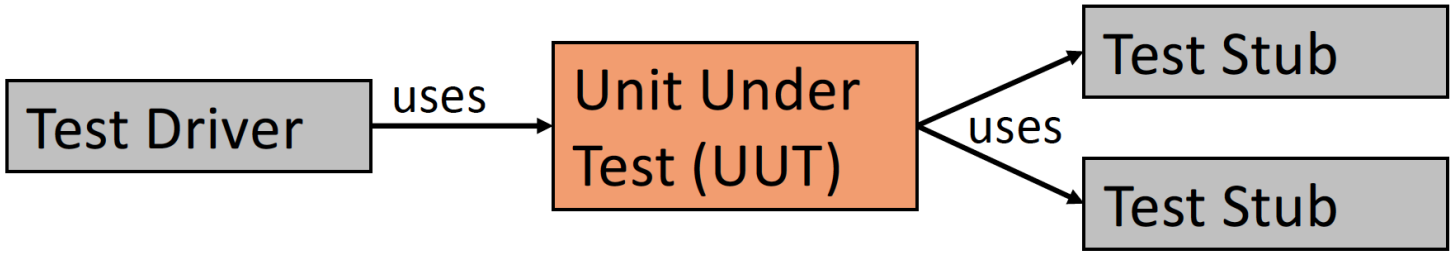
	manual	automatic
empirical	manual testing	automated testing
analytical	inspection	program analysis

https://andrewbegel.com/info461/readings/verification.html

Testing Stages



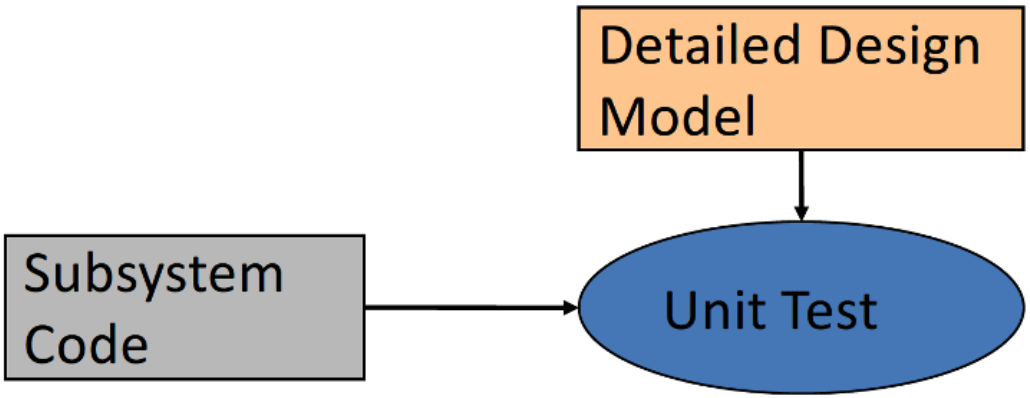
Creation of Test Harness



- Test driver
  - ✓Apply tests to UUT, including setup & clean-up
- Test stub
  - ✓Partial, temporary implementation of a component used by UUT
  - ✓Mock a missing component back fake data

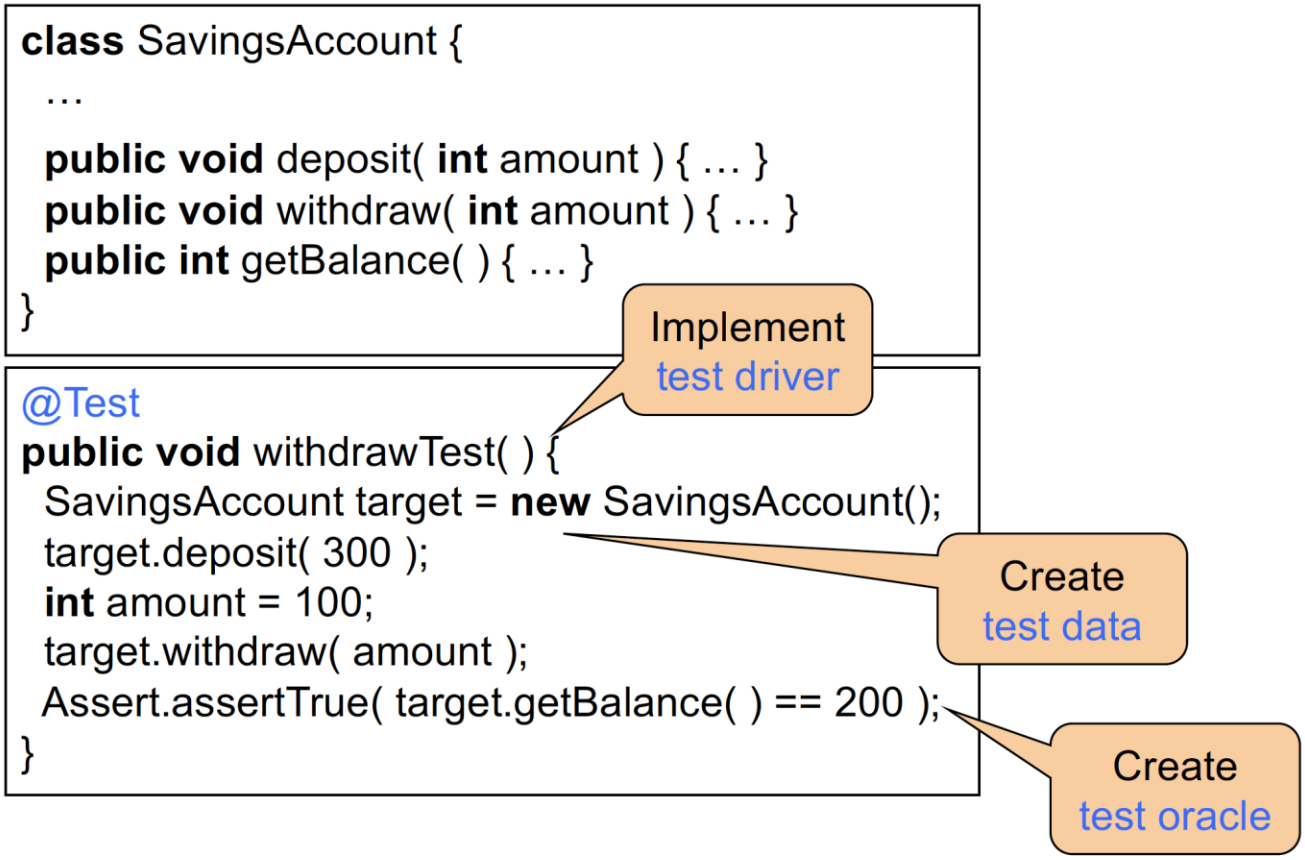
Unit Testing

- Testing individual units



- **Goal**: Confirm units are correct & meet intended functionalities

Unit Testing Example



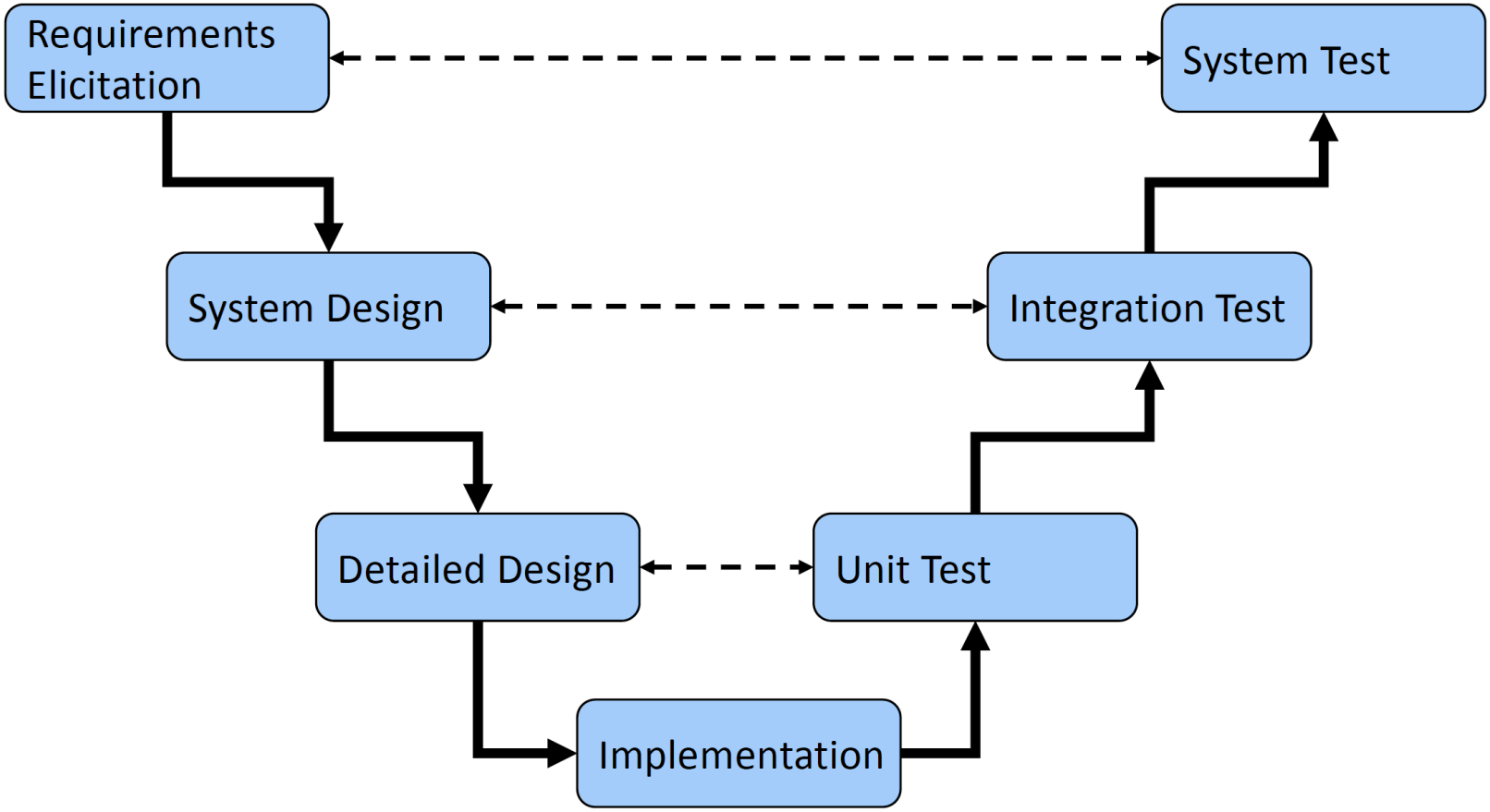
Test Execution

- Execute the tests

Eight Rules of Testing [M. Fowler]

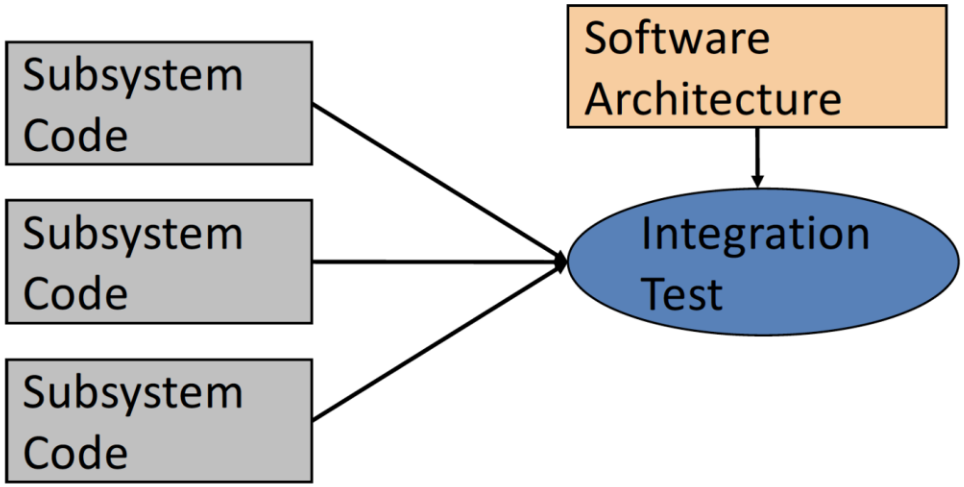
1. Make sure all tests are fully automatic and check their own results
2. A test suite is a powerful bug detector that reduces the time to find bugs
3. Run tests frequently – every test at least once a day
4. When you get a bug report, start by writing a unit test to expose the bug
5. Better to write and run incomplete tests than not run complete tests
6. Concentrate your tests on boundary conditions
7. Don't forget to test raised exceptions when things are expected to go wrong
8. Do not let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs

Testing Stages



Integration Testing

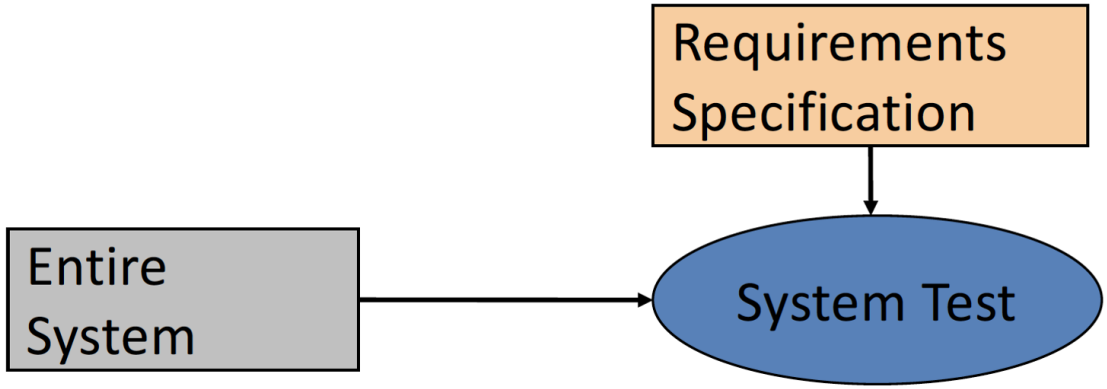
- Test groups of subsystem & eventually the entire system



- Goal: Test interfaces between subsystems

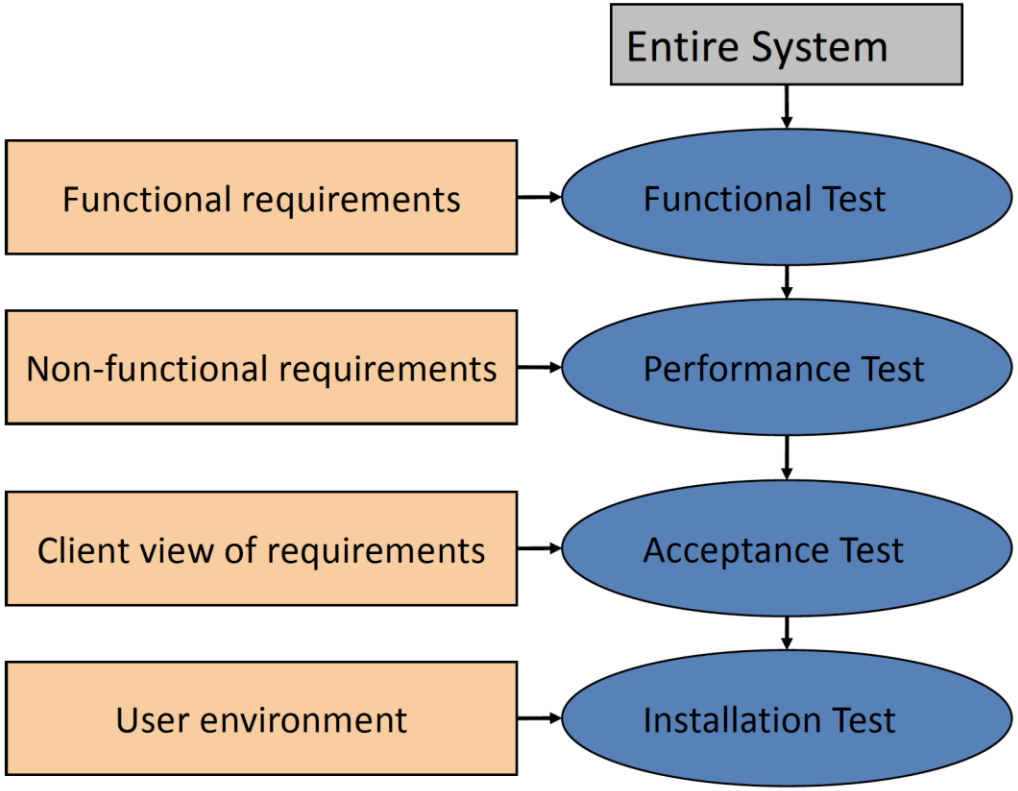
System Testing

- Test the entire system



- Goal: Determine if the system meets the requirements, both functional and non-functional

System Testing Stages



Functional Testing

- Goal: Test functionality of system
  - System is treated as a black box

- Test cases are design from requirements
  - Based on use cases
  - Alternative source: user manual

- Test cases describe
  - Input data
  - Flow of events
  - Results to check

Acceptance Testing

- **Goal:** Demonstrate system meets customer requirements
- Performed by the client, not by the developer

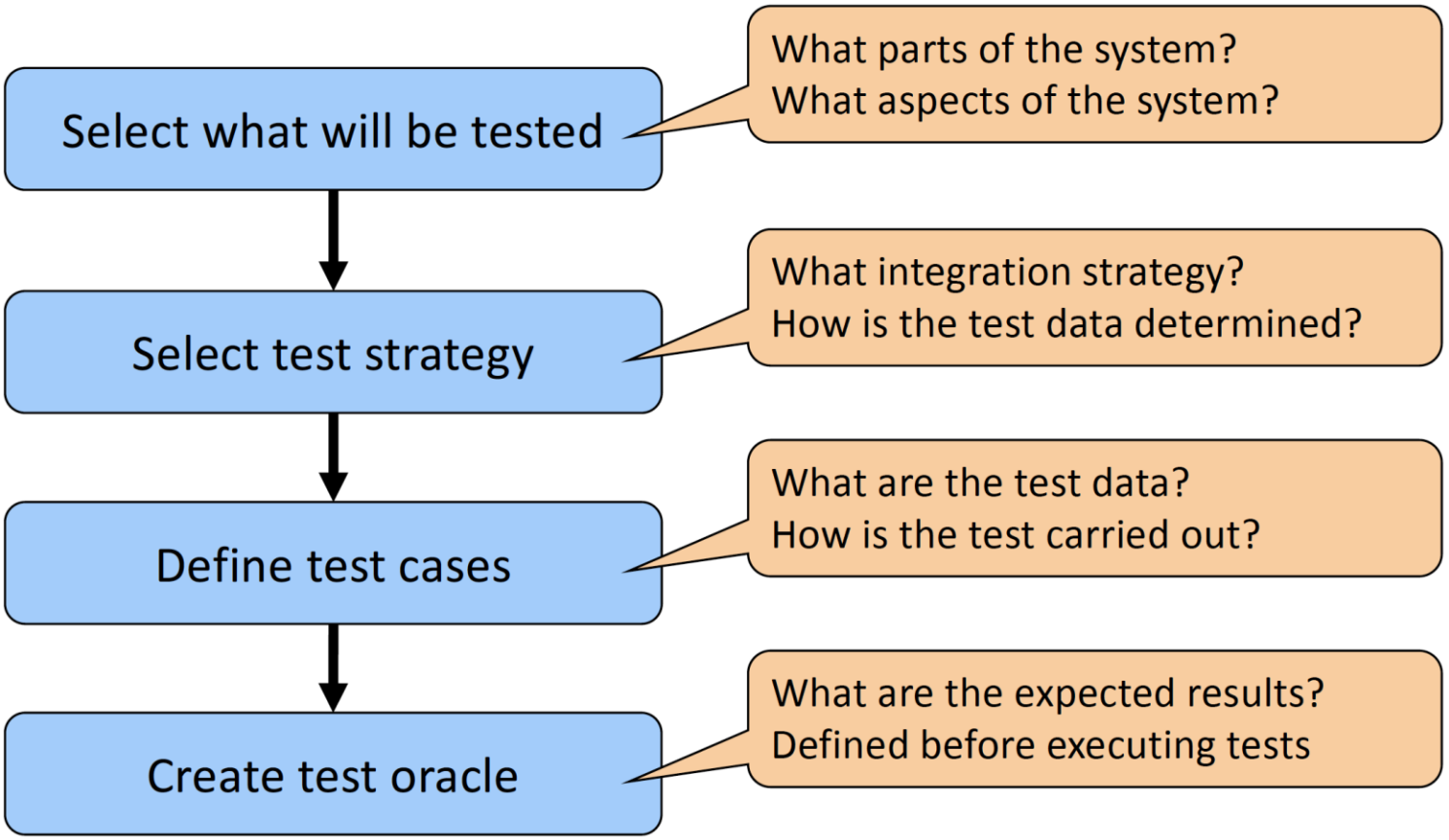
- Alpha test
  - Client uses the software at the developer's site
  - Software used in controlled setting --- developers ready to fix bugs

- Beta test
  - Conducted at client's site (developer not present)
  - Software gets a realistic workout in target environments

Independent Testing

- Hard for programmers to accept they made a mistake
  - Plus a vested interest in not finding mistakes
  - Often stick to the data that makes the program work
- Designing and programming are constructive tasks
  - Testers must seek to break the software
- Testing is done best by independent testers

Testing Steps



Testing Strategies

- **Exhaustive testing**: Check UUT for all possible inputs
  - Not feasible, even for trivial programs
- **Random testing**: Select test data randomly & uniformly
- **Functional testing**: Use requirements to decide test cases
- **Structural testing**: Use design knowledge about system structure, algorithms, data structures to determine test cases that exercise a large portion of the code

Testing Strategies

Functional testing

- Goal: Cover all requirements
- Black-box test
- Suitable for all test stages

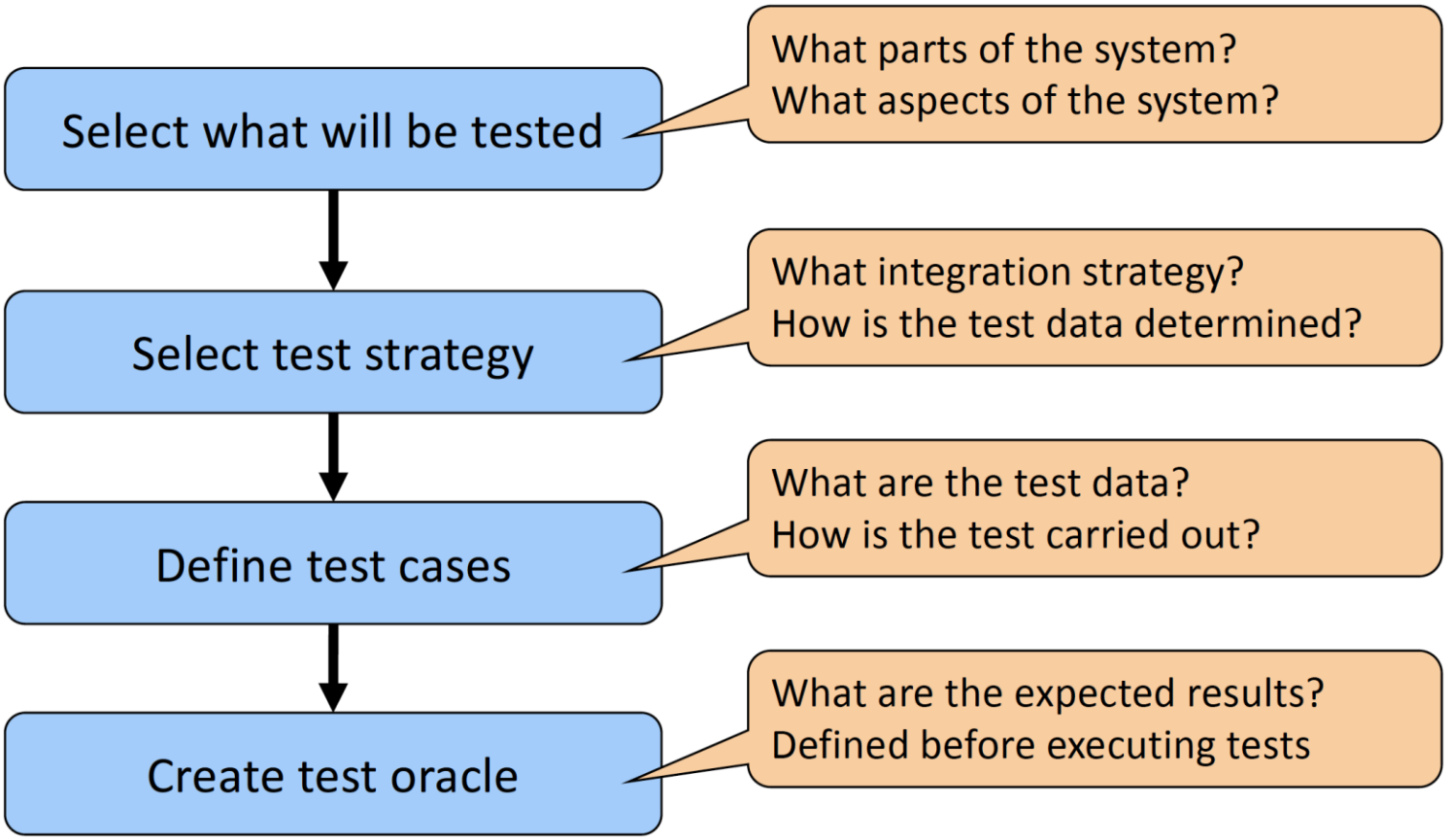
Structural testing

- Goal: Cover all the code
- White-box test
- Suitable for unit testing

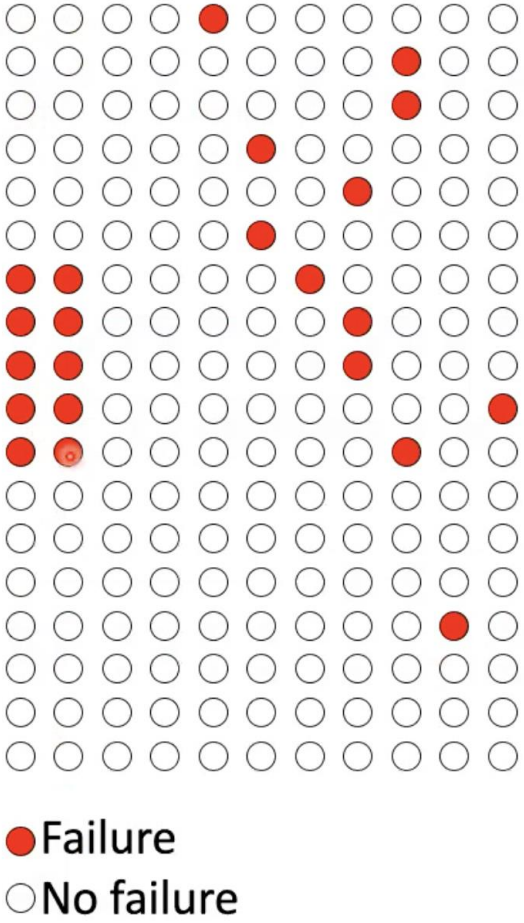
Random testing

- Goal: Cover corner cases
- Black-box test
- Suitable for all test stages

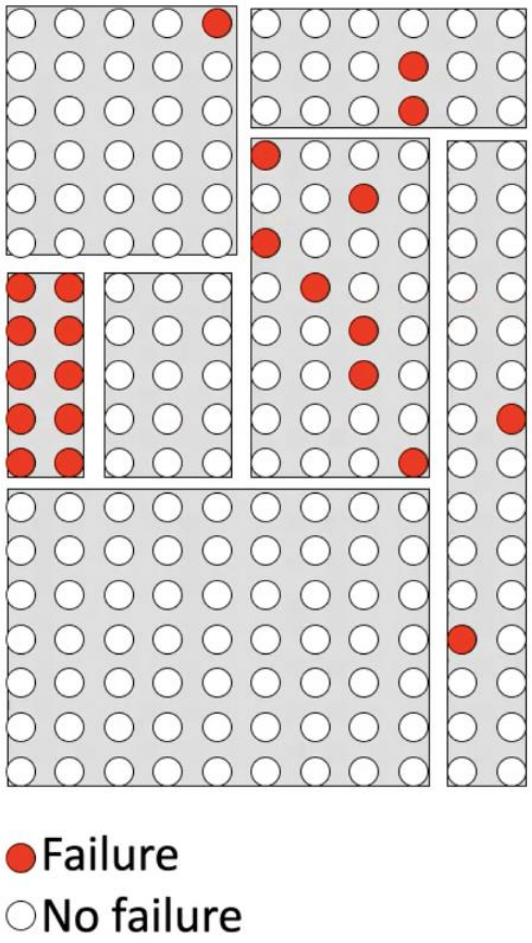
Testing Steps



Finding representative inputs

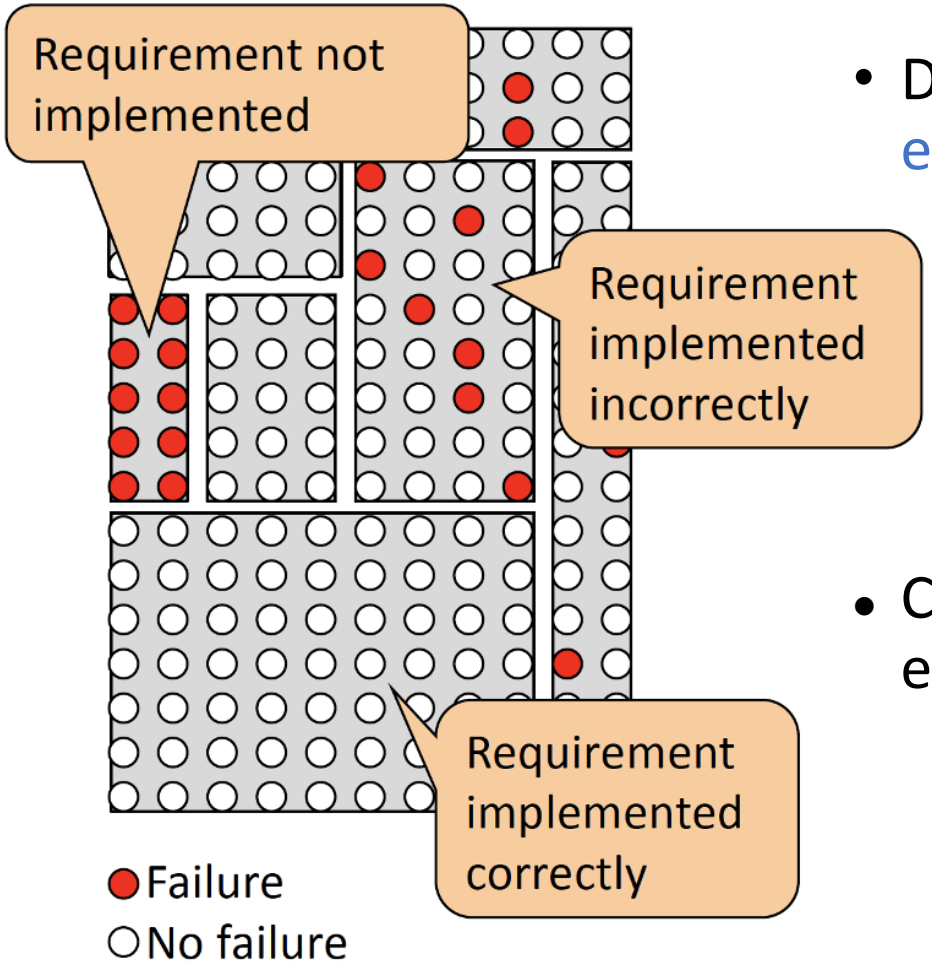


Finding representative inputs



- Divide inputs into equivalence classes
  - Each possible input belongs to one equivalence class
  - Goal: some classes have higher failure density
- Choose tests for each equivalence class

Finding representative inputs



- Divide inputs into equivalence classes
  - Each possible input belongs to one equivalence class
  - Goal: some classes have higher failure density
- Choose tests for each equivalence class

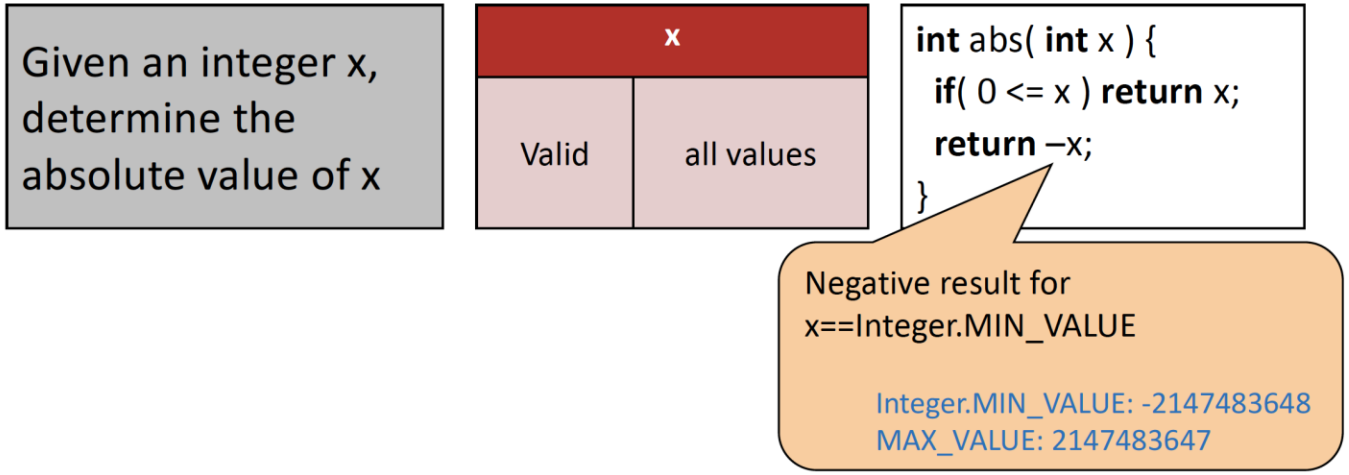
Selecting representative values

- Once partitioned the inputs, need to select concrete values for the tests for each equivalence class

- Input from a range of valid values
  - Below, within, and above the range

- Input from a discrete set of values
  - Valid and invalid discrete values
  - Instances of each subclass

Boundary Testing



- Many errors occur at boundaries of the input domain
  - Overflows
  - Comparisons (‘<’ instead of ‘<=’, etc.)
  - Missing emptiness checks (e.g., collections)
  - Wrong number of iterations

Combinatorial Testing

- Many input values: equiv. classes + boundary testing
- Testing all combinations leads to combinatorial explosion
- Reduce test cases to make effort feasible
  - Random selection
  - Semantic constraints
  - Combinatorial selection

Semantic Constraints

Given three values, a, b, c, compute all solutions of the equation $ax^2 + bx + c = 0$;
report an error if all three values are zero

	Two solutions	One solution	No solution
Linear equation		$a = 0$ and $b \neq 0$	$a = 0$, $b = 0$, and $c \neq 0$
(Truly) quadratic equation	$a \neq 0$ and $b^2-4ac > 0$	$a \neq 0$ and $b^2-4ac = 0$	$a \neq 0$ and $b^2-4ac < 0$
Invalid input	$a = 0$, $b = 0$, $c = 0$		

Pairwise Combinatorial Testing

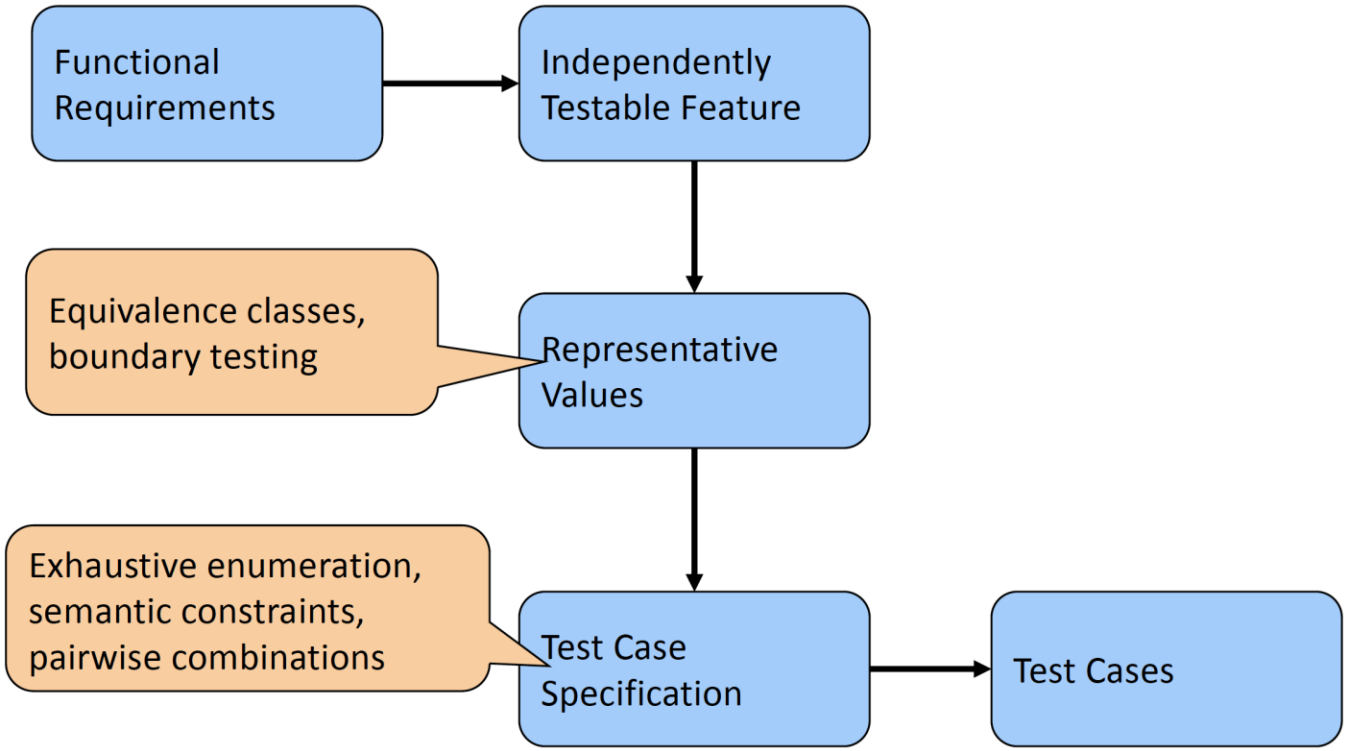
- Focus on possible combinations of pairs of inputs

- Example: Consider a method with 4 Boolean parameters
  - Combinatorial testing requires $2^4 = 16$ test cases
  - Pairwise-combinations testing requires 5 test cases:

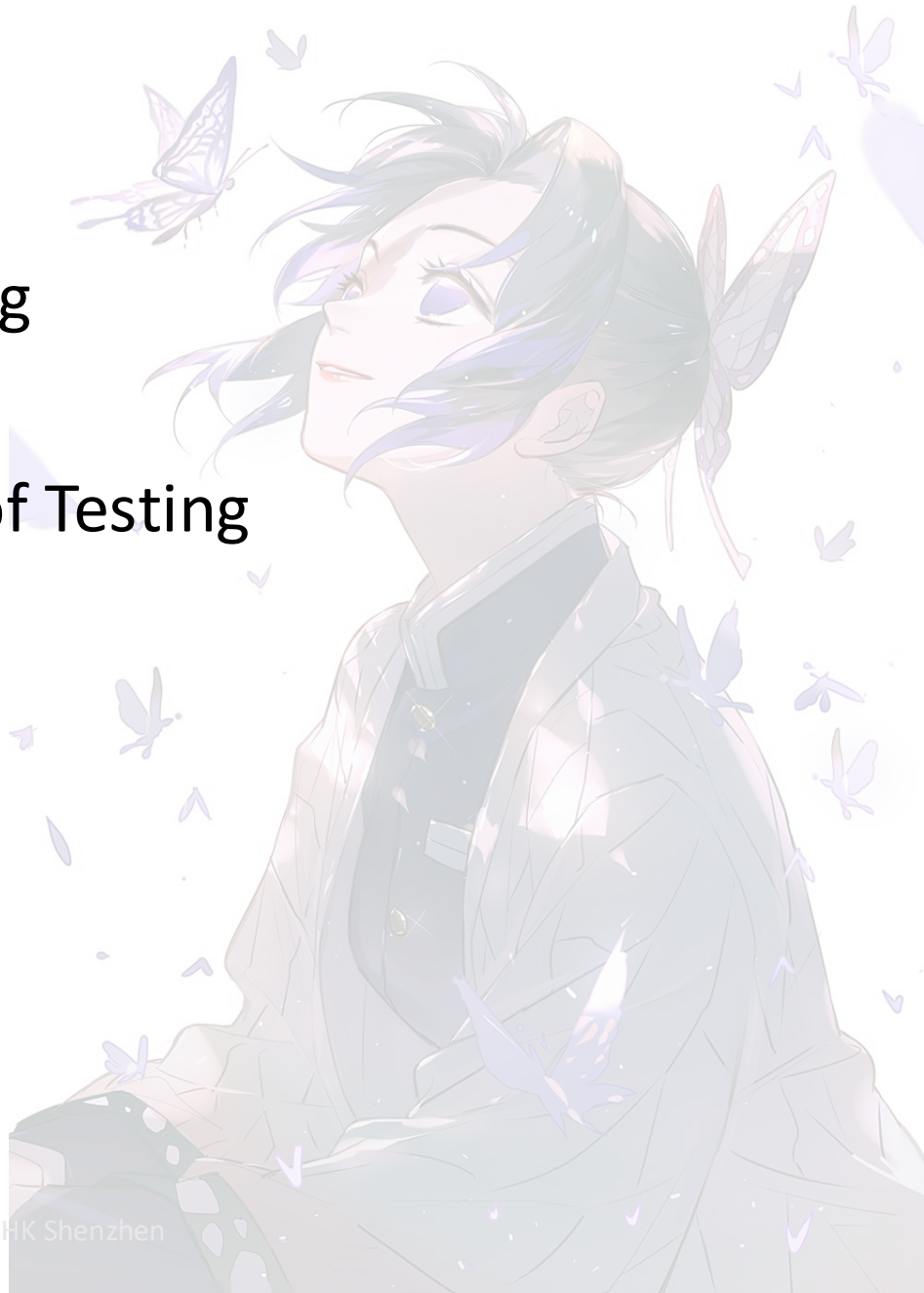
    TTTT, TFFF, FTFF, FFTF, FFFT

- Can be generalized to k-tuples (k-way testing)
- For n parameters with d values per parameter, the number of test cases grow logarithmically in n and quadratic in d
- Results hold for large n & d, and for all k in k-way testing

Functional Testing: Summary

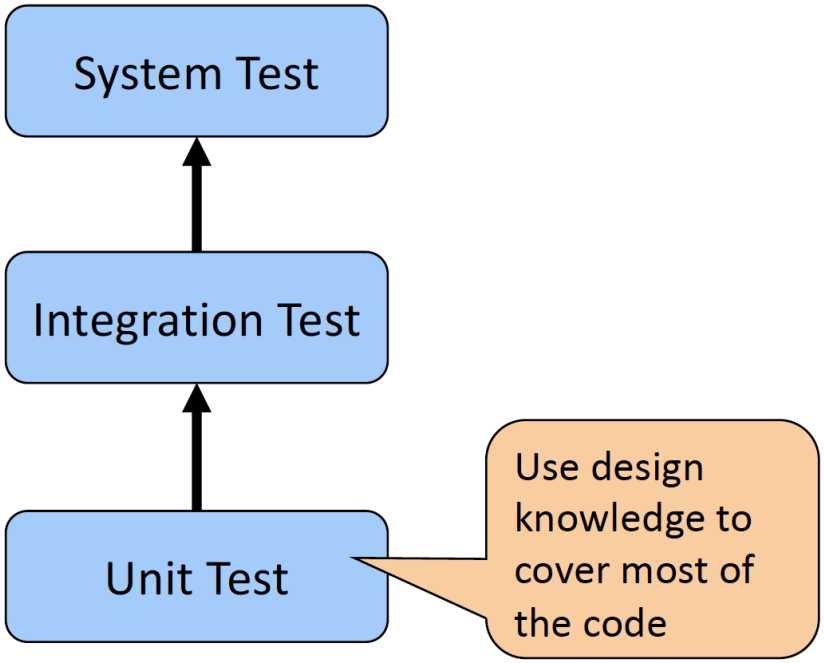


1. What is Software Testing
2. Why Software Testing
3. Stages and Categories of Testing
4. Coverage Metrics



Structural Testing

- White-box test a unit to cover a large portion of its code



Basic Blocks

- Basic block: a sequence of statements having
  - One entry point: no code within is a jump target
  - One exit point: only the last instruction exits the block

  - If the first instruction is executed, the rest are also executed

Basic Blocks: Example

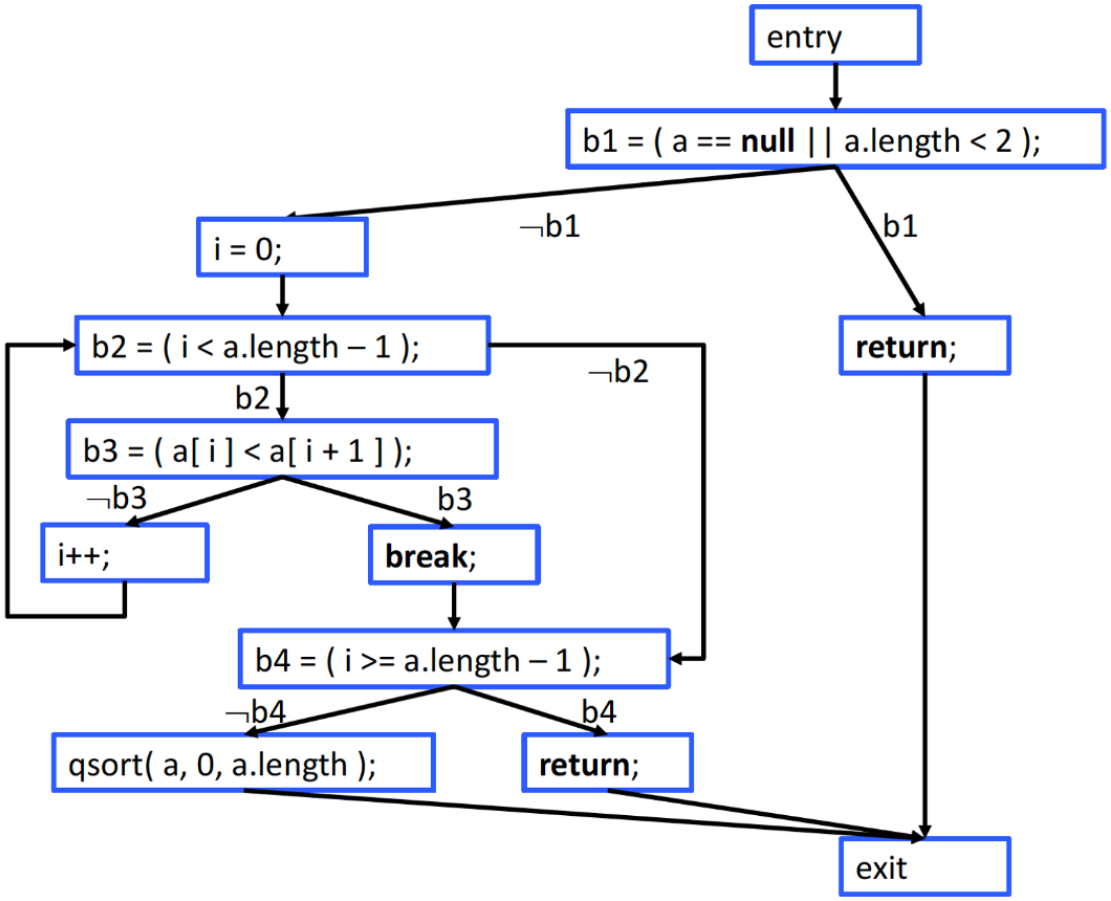
```
public void sort( int[ ] a ) {
  if( a == null || a.length < 2 )
    return;
  int i;
  for( i = 0; i < a.length – 1; i++ ) {
    if( a[ i ] < a[ i + 1 ] )
      break;
  }
  if( i >= a.length – 1 )
    return;
  qsort( a, 0, a.length );
}
```

Intraprocedural Control Flow Graphs

- An intraprocedural CFG of a proc. p is a graph (N, E) where
  - Nodes
    - Basic blocks in p
    - Designated entry and exit blocks
  - Edges
    - **(a, b)**: an edge from a to b if there is direct control flow from a to b
    - **(entry, a)**: a is the first basic block of p
    - **(b, exit)**: for each b ending with a (possibly implicit) return statement

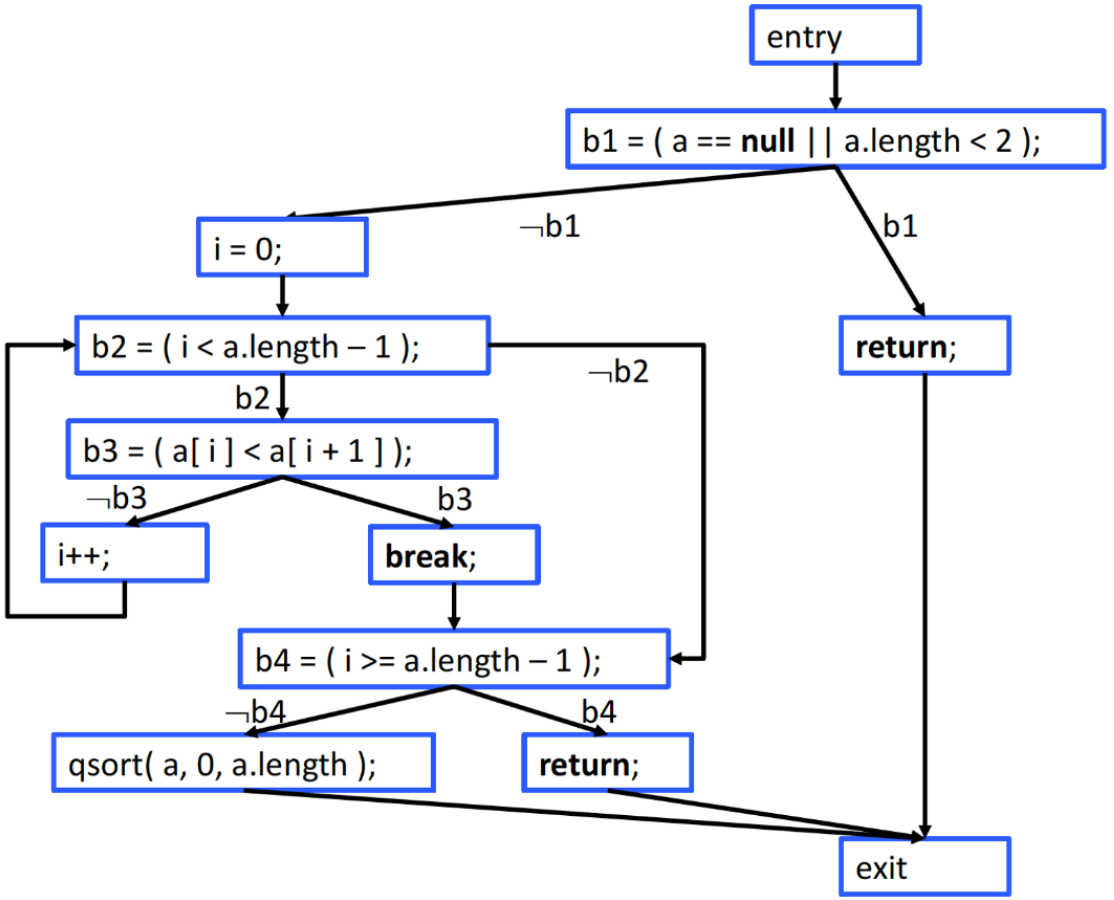
Control Flow Graphs: Example

```
public void sort( int[ ] a ) {
  if( a == null || a.length < 2 )
    return;
  int i;
  for( i = 0; i < a.length – 1; i++ ) {
    if( a[ i ] < a[ i + 1 ] )
      break;
  }
  if( i >= a.length – 1 )
    return;
  qsort( a, 0, a.length );
}
```



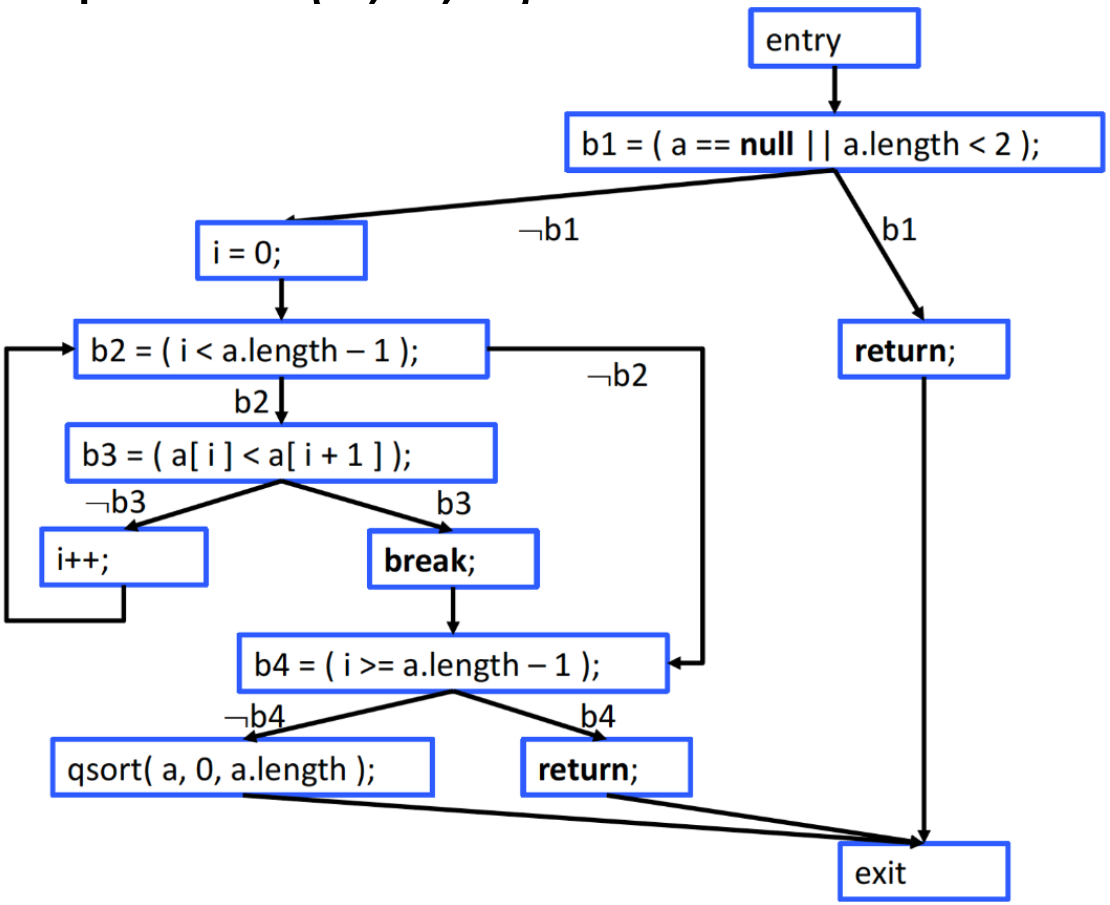
Test Coverage

- The CFG serves as an adequacy criterion for tests
- The more parts executed, the higher chance to find bugs
- "parts" can be
  - Nodes
  - Edges
  - Paths
  - etc.



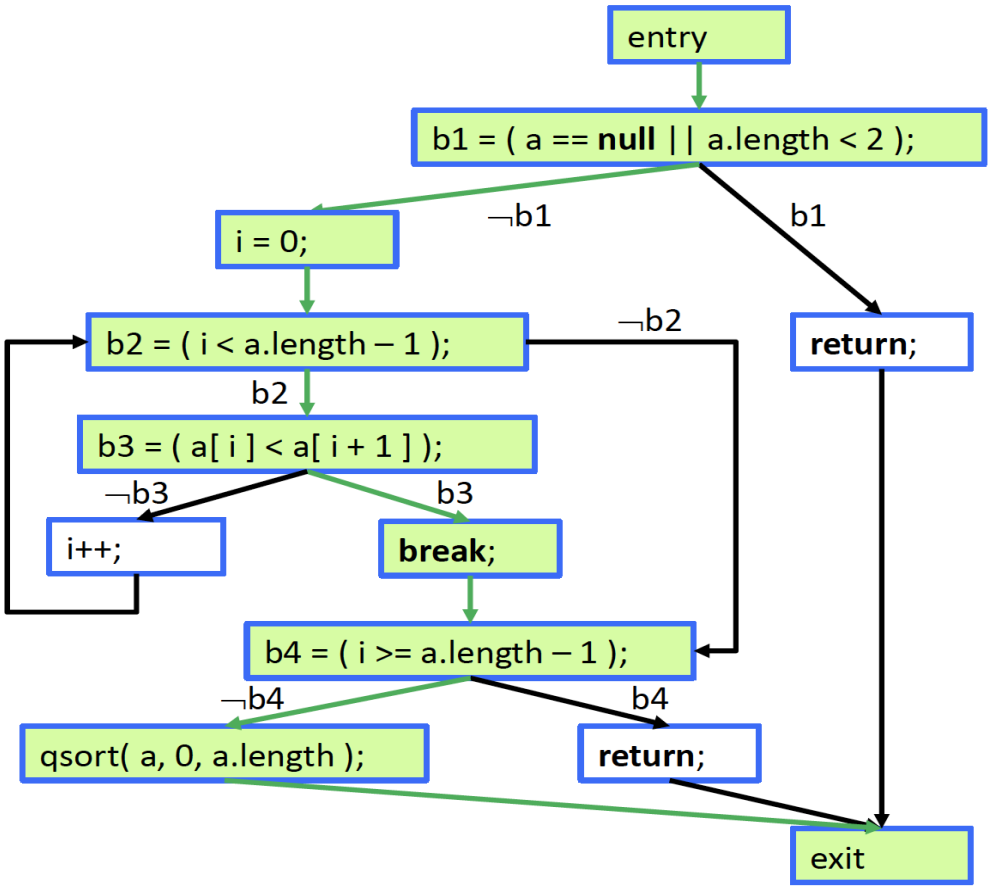
Test Coverage

- Consider input a = {3, 7, 5}



Test Coverage

- Consider input a = {3, 7, 5}



Statement Coverage

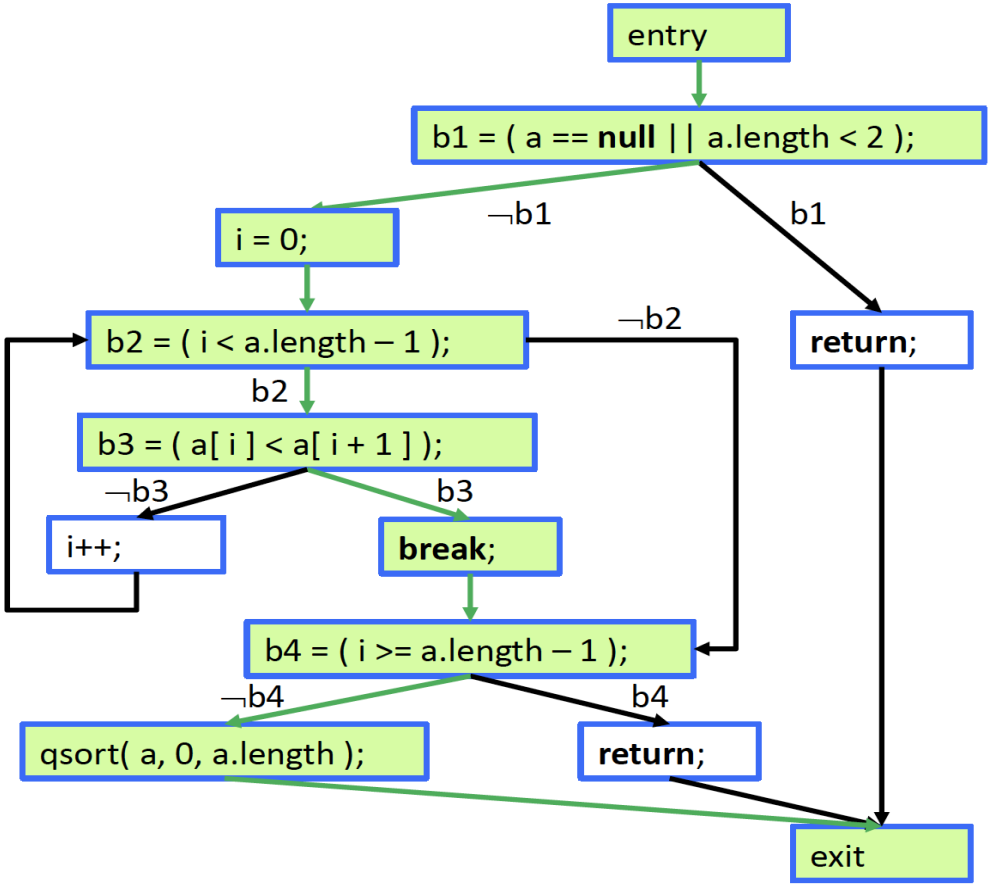
- Judge a test suite's quality by how much the CFG is covered

- Idea: Can detect a bug in a statement only by executing it
  - Can also be defined on basic blocks

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}}$$

Statement Coverage: Example

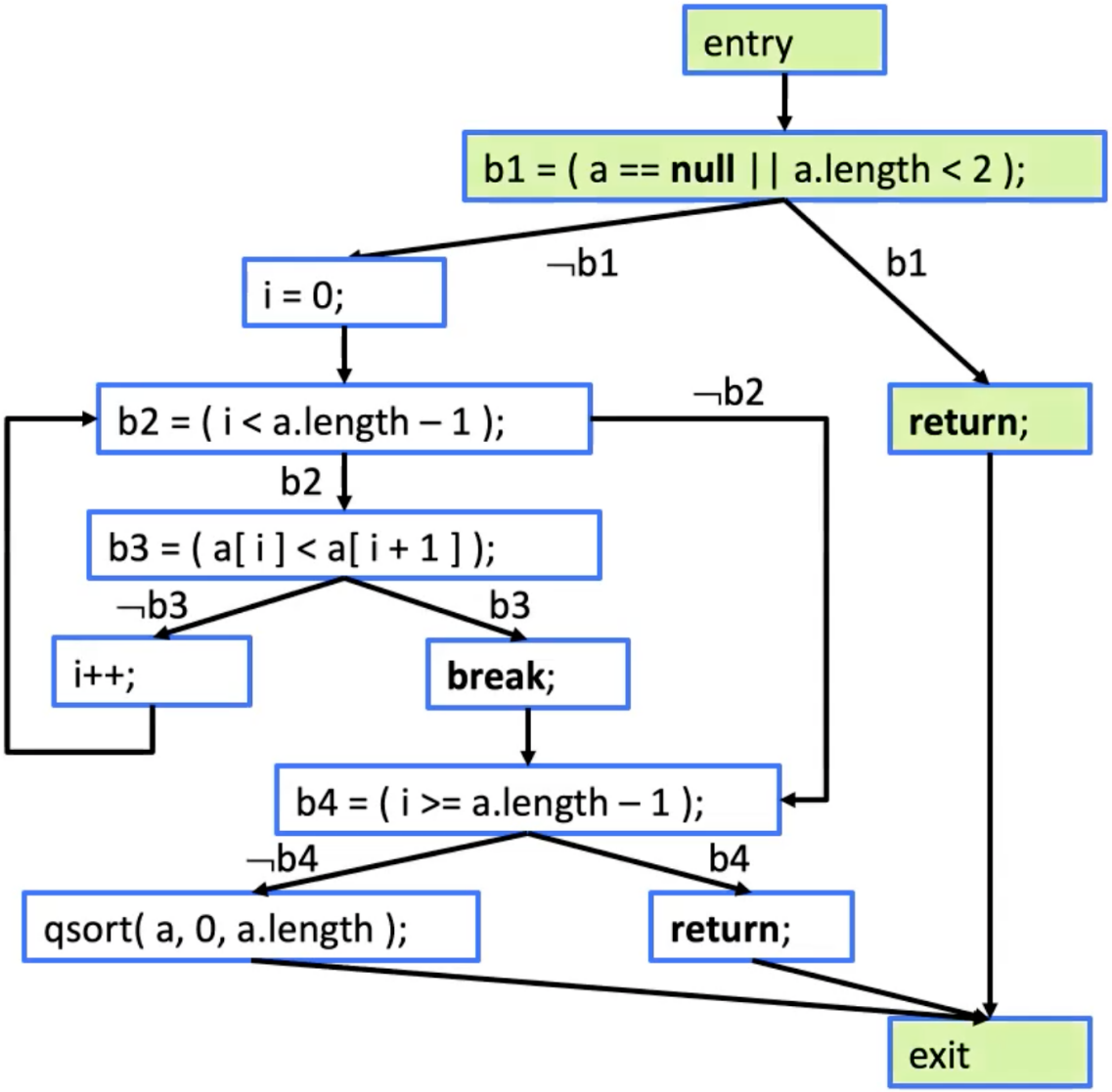
- Consider input a = {3, 7, 5}
- It executes 7/10 basic blocks
- Statement coverage: 70%
- How to achieve 100% statement coverage?



Statement Coverage: Example (cont.)

- We can achieve 100% statement coverage with three test cases
  - a = { 1 }
  - a = { 5, 7 }
  - a = { 7, 5 }

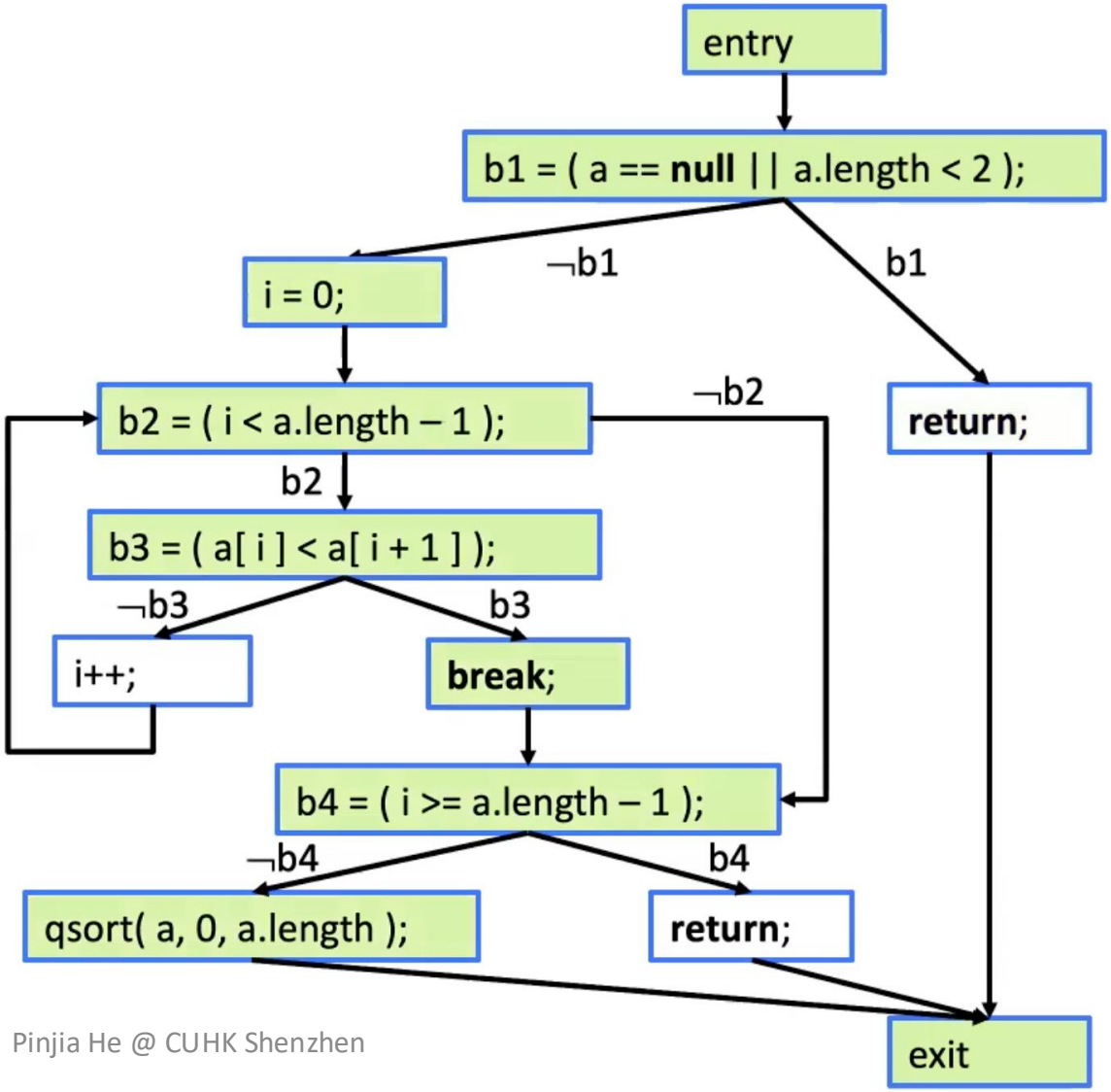
- The last detects the bug



Statement Coverage: Example (cont.)

- We can achieve 100% statement coverage with three test cases
  - a = { 1 }
  - a = { 5, 7 }
  - a = { 7, 5 }

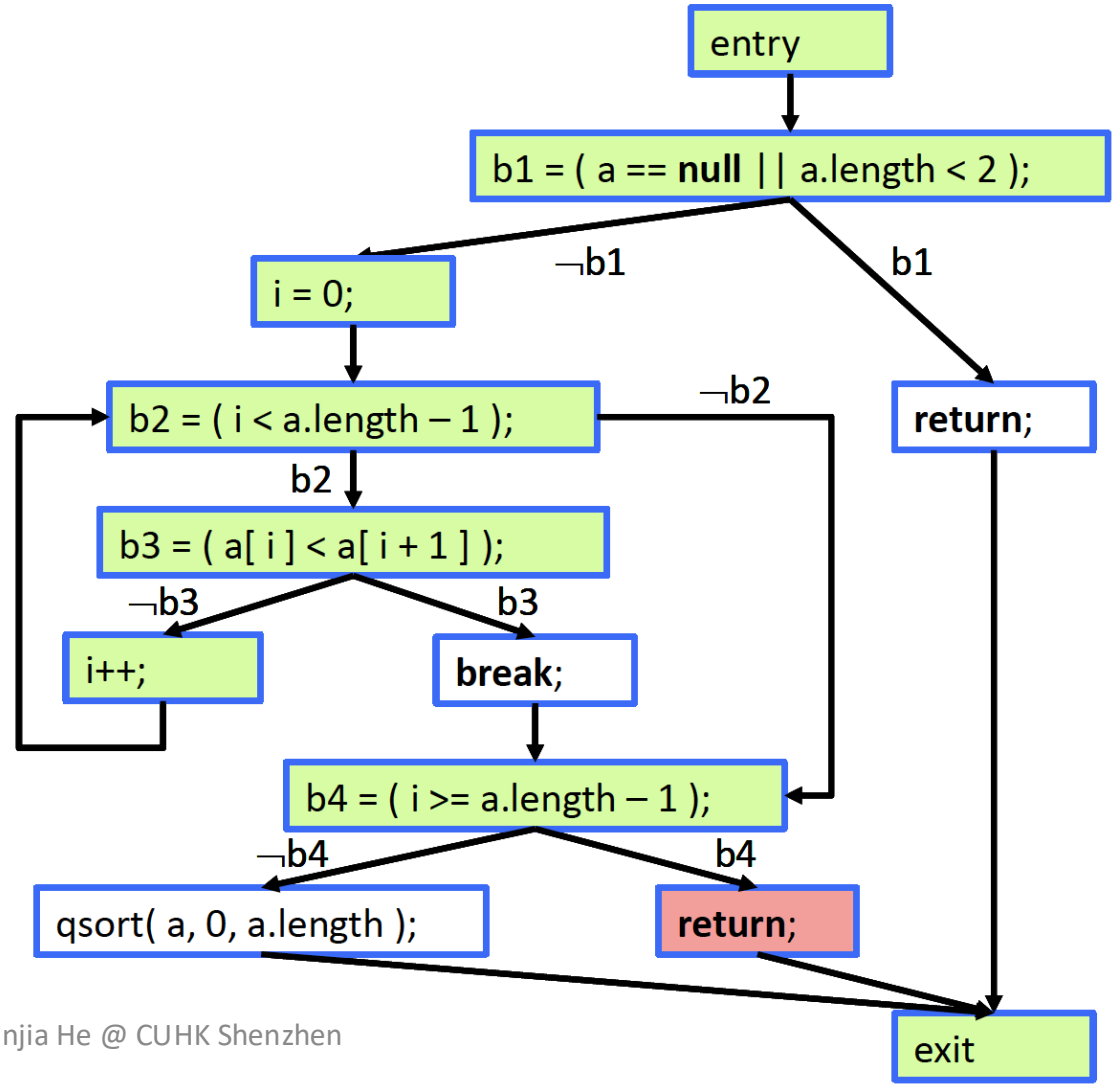
- The last detects the bug



Statement Coverage: Example (cont.)

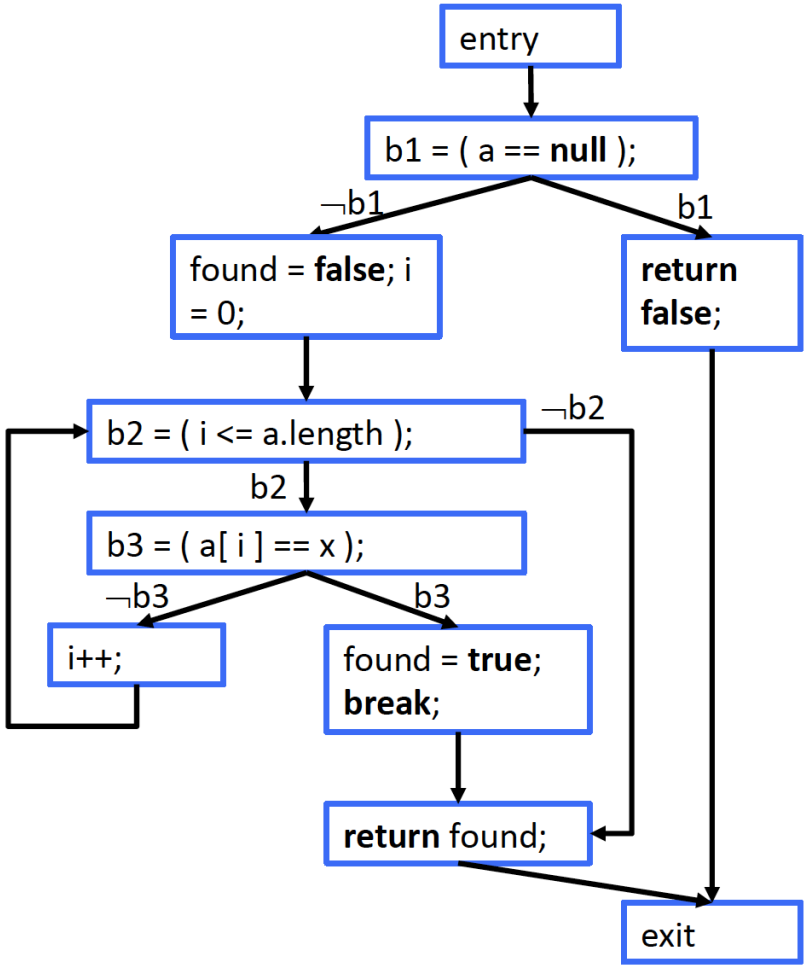
- We can achieve 100% statement coverage with three test cases
  - a = { 1 }
  - a = { 5, 7 }
  - a = { 7, 5 }

- The last detects the bug



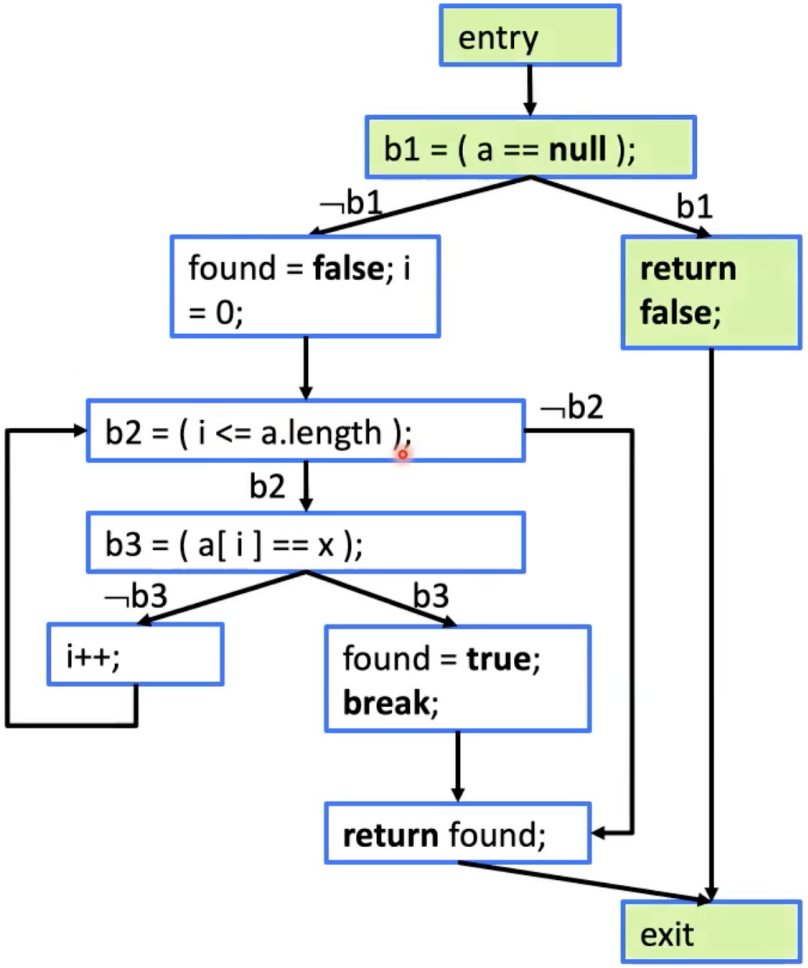
Statement Coverage: Discussion

```
boolean contains( int[ ] a, int x ) {
  if( a == null )  return false;
  boolean found = false;
  for( int i = 0; i <= a.length; i++ ) {
    if( a[ i ] == x ) {
      found = true;
      break;
    }
  }
  return found;
}
```



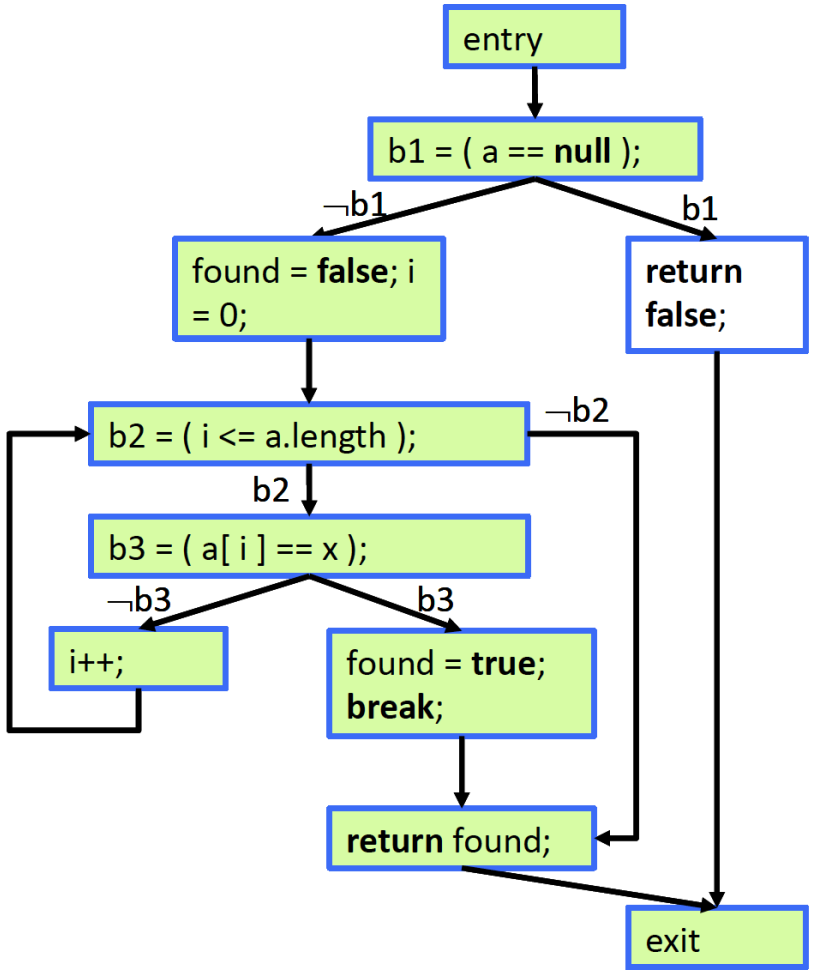
Statement Coverage: Discussion

- Achieve 100% statement coverage with 2 tests
  - a = null
  - a = { 1, 2 }, x = 2



Statement Coverage: Discussion

- Achieve 100% statement coverage with 2 tests
  - a = null
  - a = { 1, 2 }, x = 2
- The tests do not detect the bug
- More thorough testing is needed



Branch Coverage

- **Idea:** test all possible branches in the control flow
- Edge (m,n) is a branch iff there is edge (m, n') with $n \neq n'$

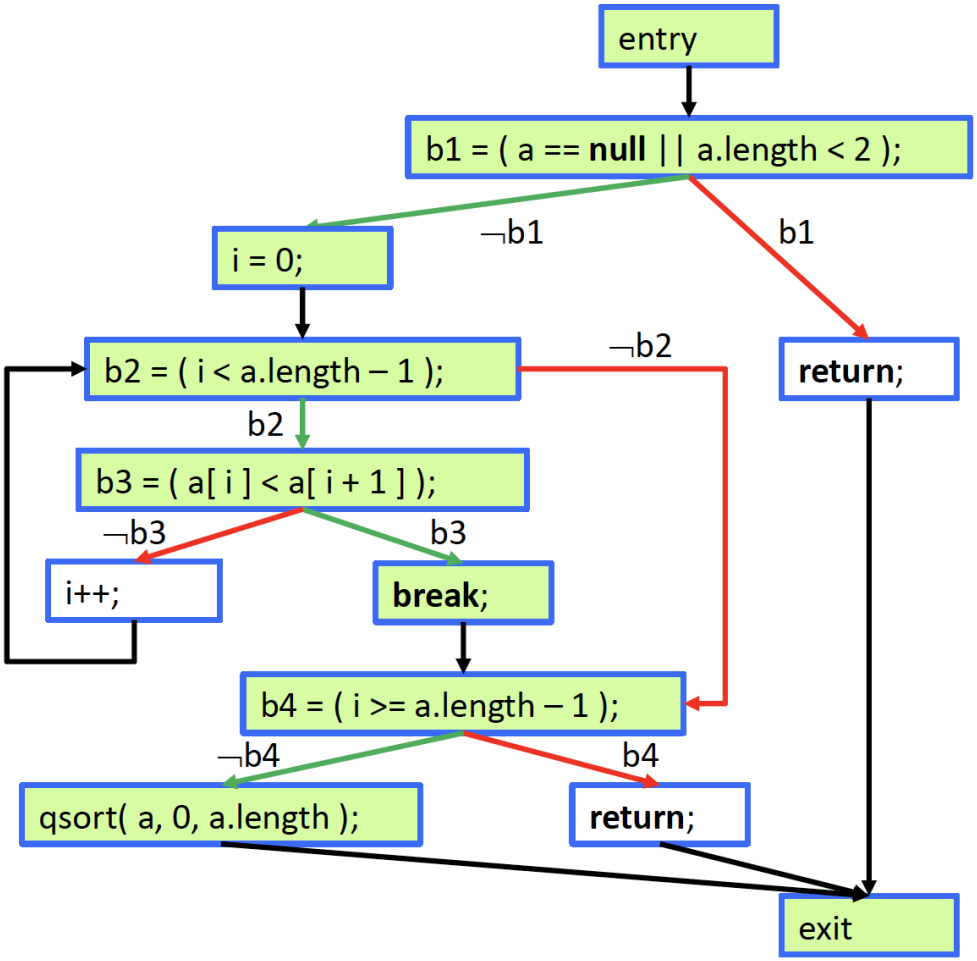
$$\text{Branch Coverage} = \frac{\text{Number of executed branches}}{\text{Total number of branches}}$$

- Branch coverage is 100% if code has no branches

- Most widely-used adequacy criterion in industry
- Branch coverage is more thorough than statement coverage
  - Q: 100% branch coverage => 100% statement coverage ?
  - Q: ">=n% branch cov." => ">=n% statement cov." ?

Branch Coverage: Example 1

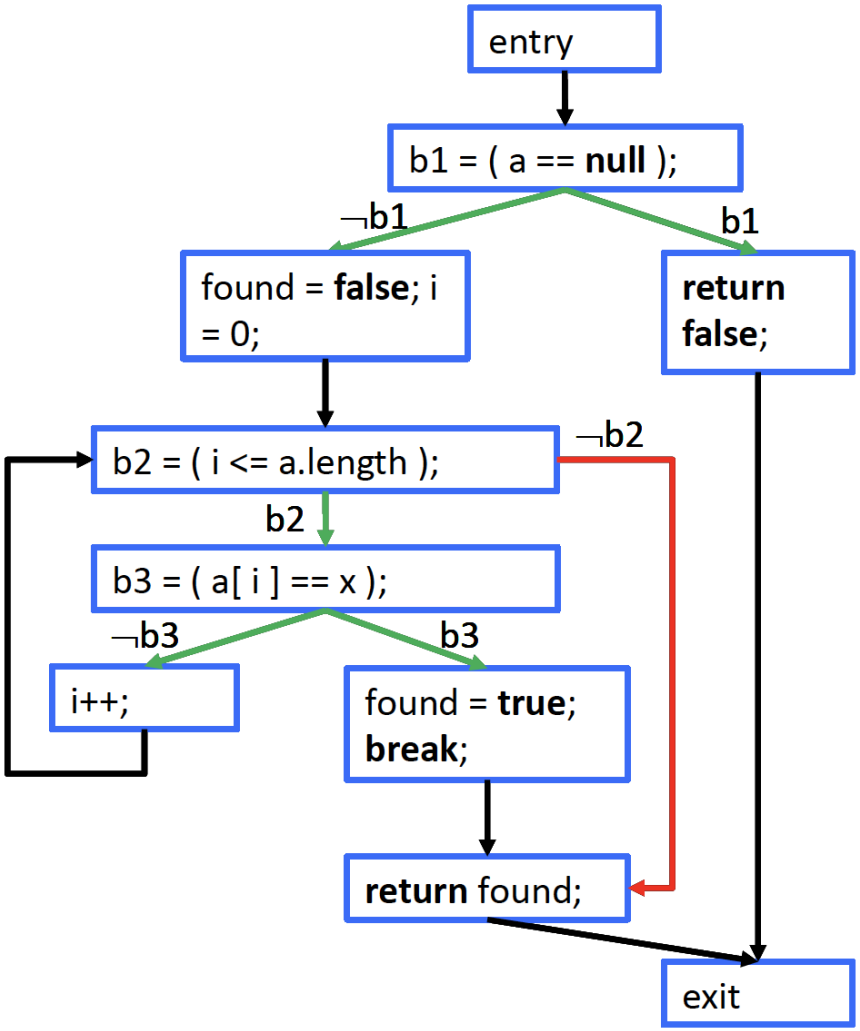
- Consider input a = {3, 7, 5}
- It executes 4/8 branches
- Branch coverage 50%
- What tests to include for 100% branch coverage



Branch Coverage: Example 2

- The two tests:
  - a = null
  - a = {1,2}, x = 2

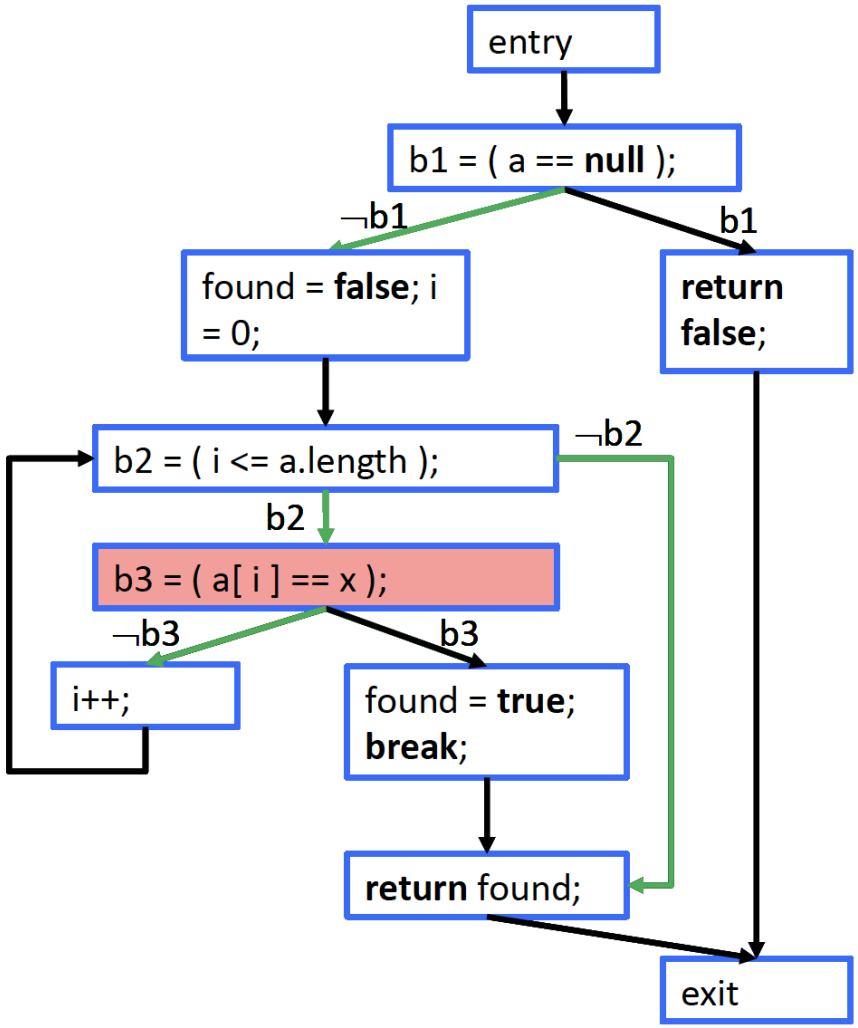
  execute 5/6 branches
- Branch coverage: 83%
- How to achieve 100% branch coverage?



Branch Coverage: Example 2

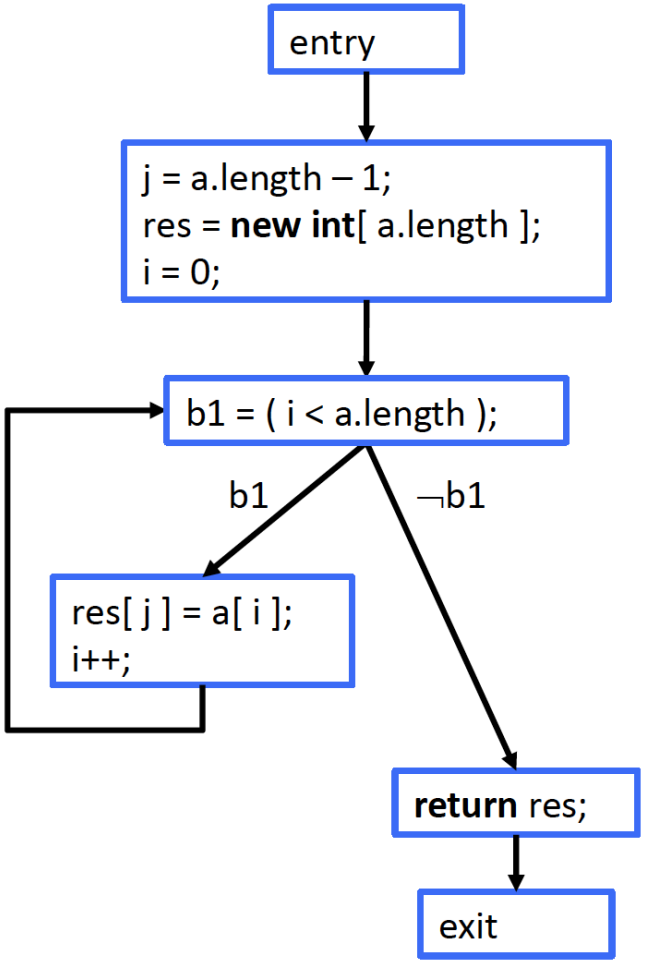
- To achieve 100% branch coverage requires a test that runs the loop to the end
  - a = null
  - a = {1}, x = 1
  - a = {1}, x = 3

- The last one detects the bug



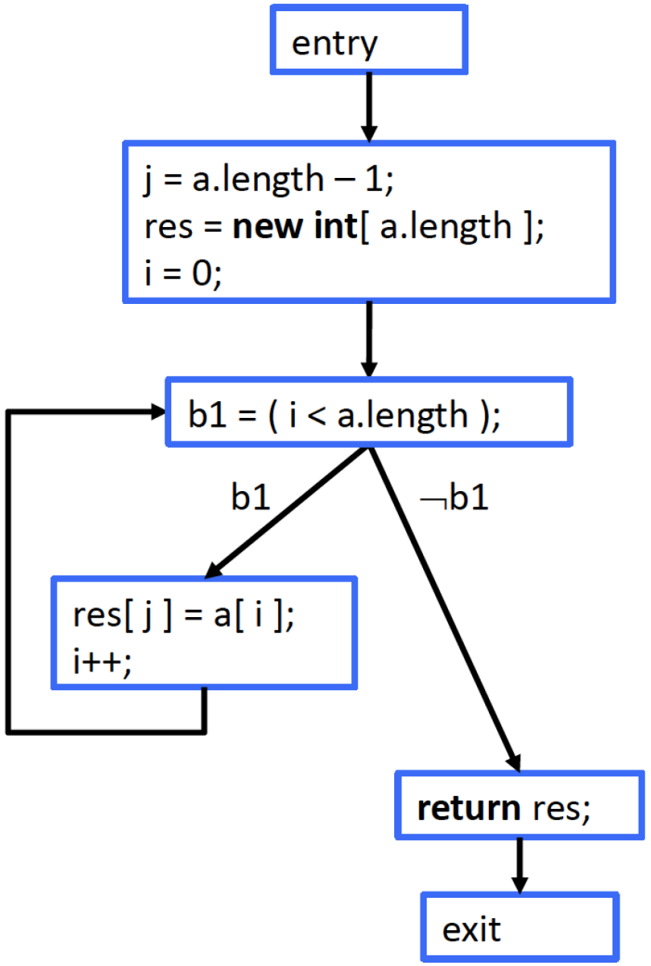
Branch Coverage: Discussion

```
int[ ] reverse( int[ ] a ) {
  int j = a.length – 1;
  int[ ] res = new int[ a.length ];
  for( int i = 0; i < a.length; i++ ) {
    res[ j ] = a[ i ];
  }
  return res;
}
```



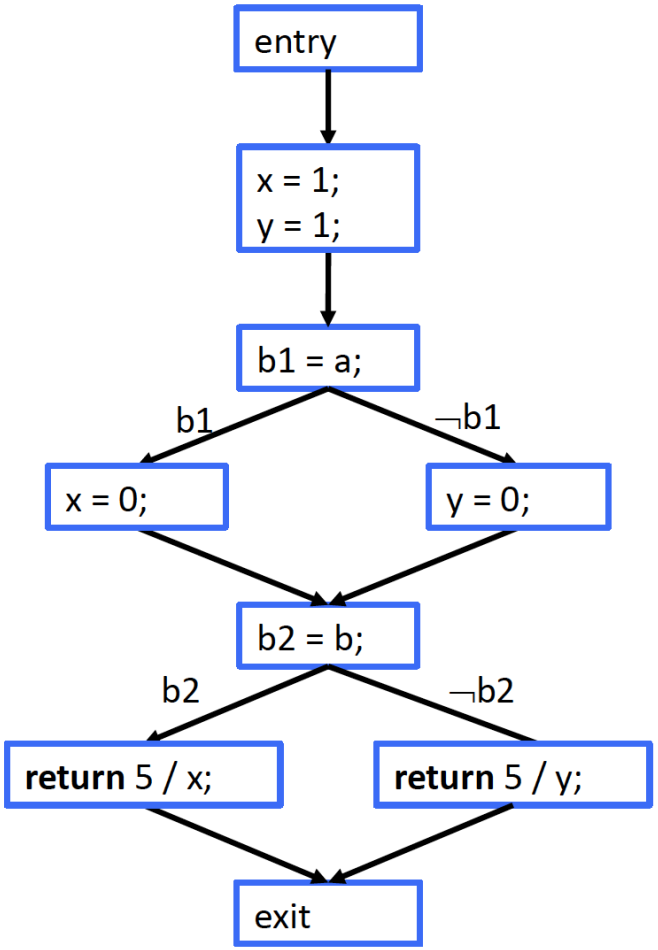
Branch Coverage: Discussion

- Can obtain 100% branch coverage with 1 test
  - a = {1}
- The test doesn't detect the bug
- More tests are needed



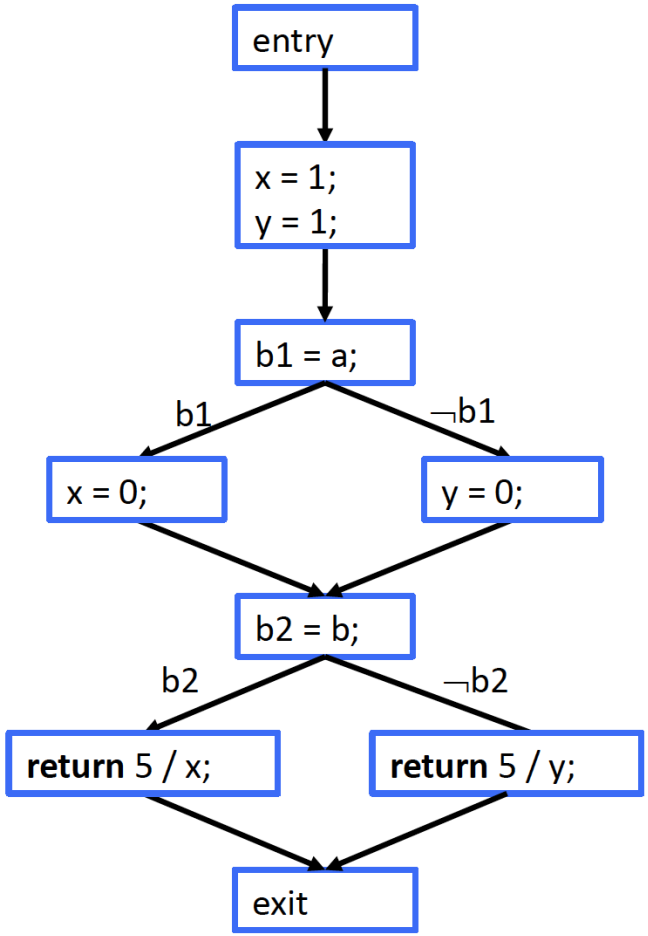
Branch Coverage: Discussion

```
int foo( boolean a, boolean b ) {
  int x = 1;
  int y = 1;
  if( a )
    x = 0;
  else
    y = 0;
  if( b )
    return 5 / x;
  else
    return 5 / y;
}
```



Branch Coverage: Discussion

- We can achieve 100% branch coverage with two tests
  - a = **true**, b = **false**
  - a = **false**, b = **true**
- The tests do not detect the bug
- More thorough testing is needed



Path Coverage

- **Idea**: test all possible paths through the CFG
- A path is a sequence of nodes $n_1, \ldots n_k$ such that
  - $n_1 = entry$
  - $n_k = exit$
  - There is an edge (n_i, n_{i+1}) in the CFG

$$\text{Path Coverage} = \frac{\text{Number of executed paths}}{\text{Total number of paths}}$$

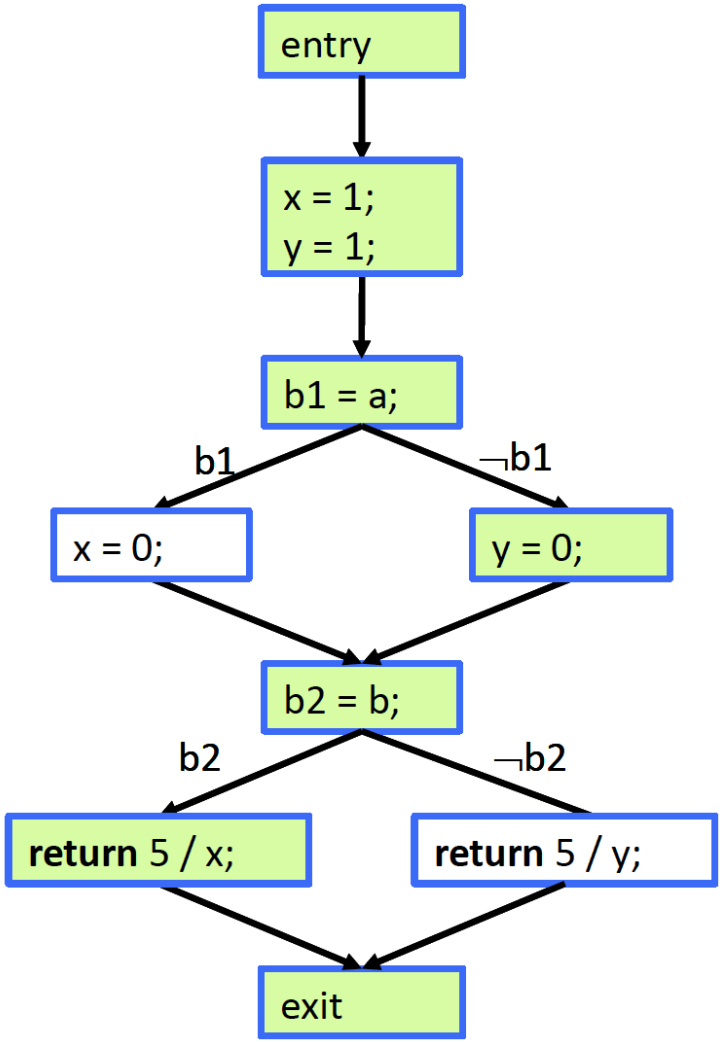
- Path coverage is more thorough than statement and branch coverage
  - 100% path coverage => 100% complete statement and branch coverage
  - But, “>=n% path coverage” does not generally imply “>=n% statement coverage” or “>=n% branch coverage”
- Complete path coverage is not feasible for input-dependent loops
  - Unbounded number of paths

Path Coverage: Example 1

- The two tests
  - a = **true**, b = **false**
  - a = **false**, b = **true**

  execute 2/4 paths

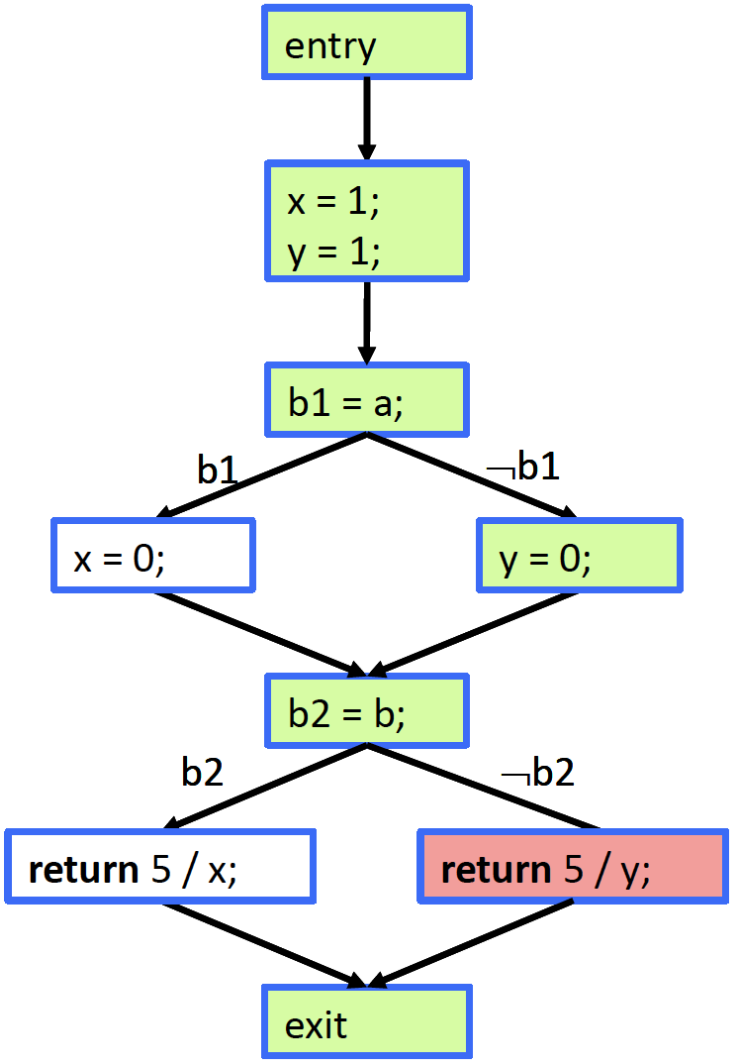
- Path coverage: 50%



Path Coverage: Example 1

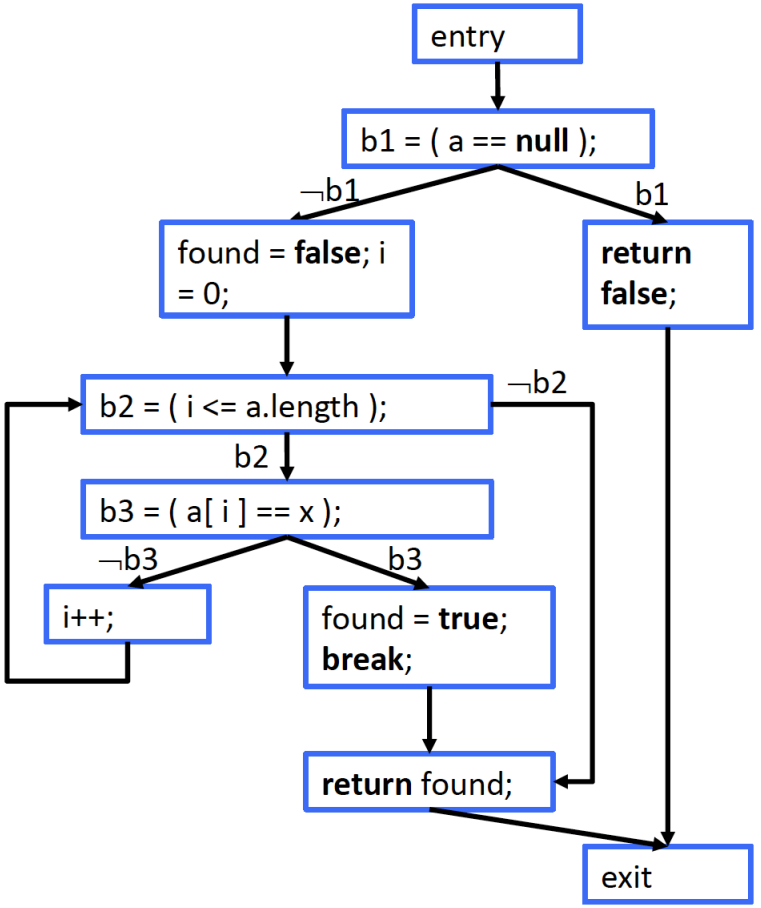
- We can achieve 100% path coverage with four tests
  - a = **true**, b = **false**
  - a = **false**, b = **true**
  - a = **true**, b = **true**
  - a = **false**, b = **false**

- The two new tests detect the bug



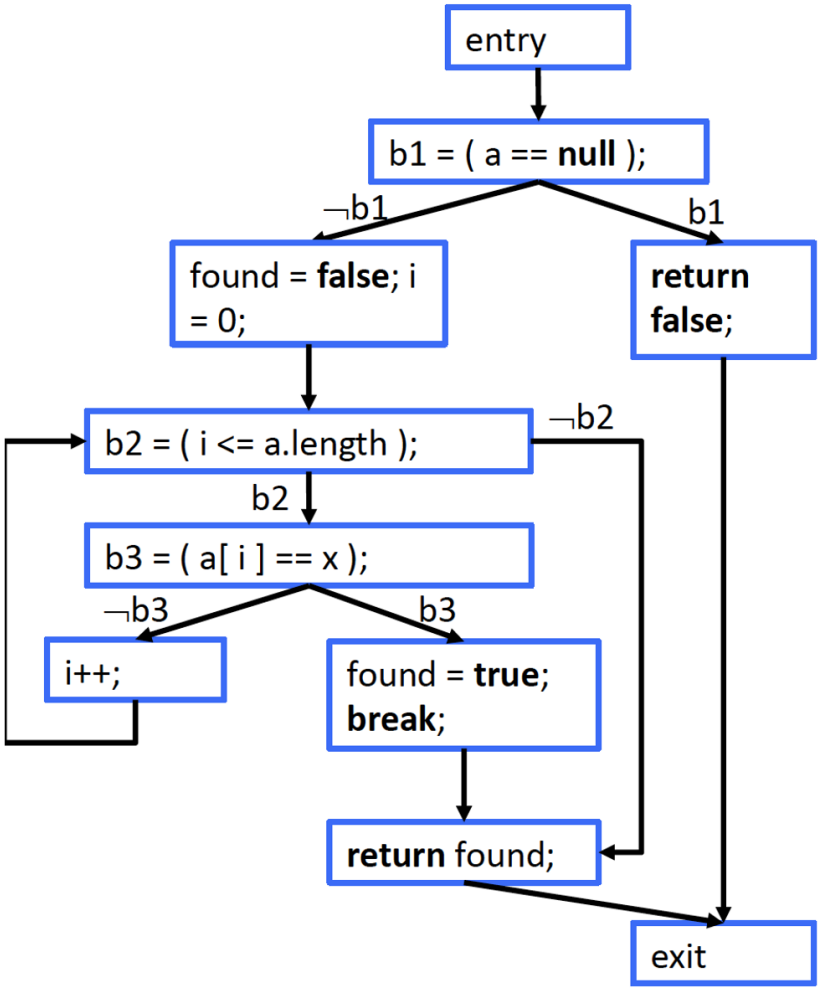
Path Coverage: Example 2

```
boolean contains( int[ ] a, int x ) {
  if( a == null )  return false;
  boolean found = false;
  for( int i = 0; i <= a.length; i++ ) {
    if( a[ i ] == x ) {
      found = true;
      break;
    }
  }
  return found;
}
```



Path Coverage: Example 2

- Number of loop iterations is unknown statically
- Arbitrarily many tests are needed for complete path coverage



Loop Coverage

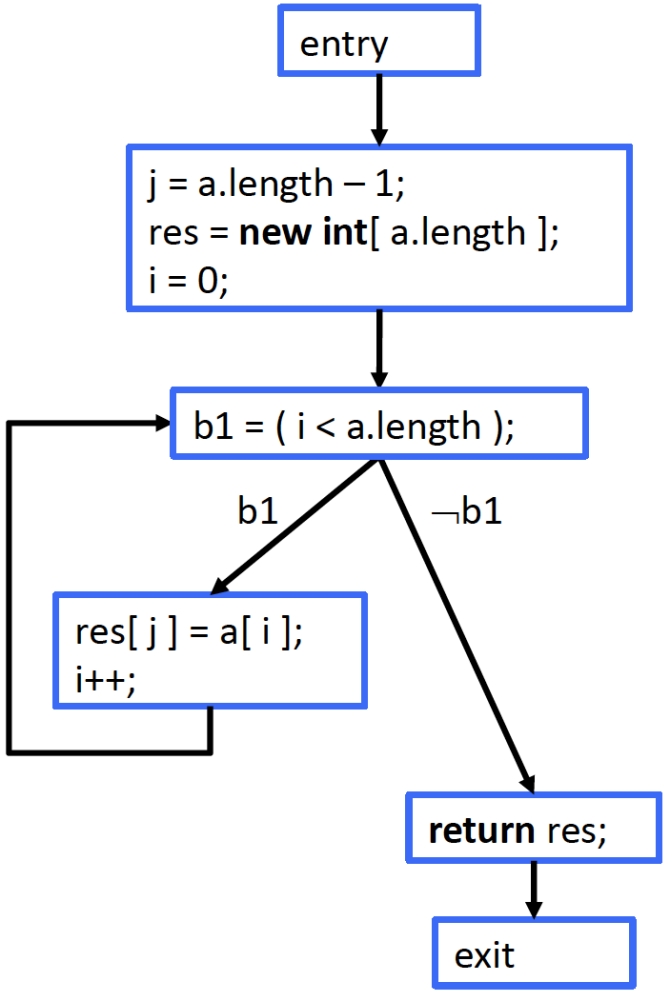
- Idea: for each loop, test 0,1,>=2 (consecutive) iterations

$$\text{Loop Coverage} = \frac{\text{Number of executed loops with 0, 1, and more than 1 iterations}}{\text{Total number of loops} * 3}$$

Loop Coverage: Example

- We can achieve 100% loop coverage with 3 tests
  - a = {}
  - a = {1}
  - a = {1,2}

- The last detects the bug



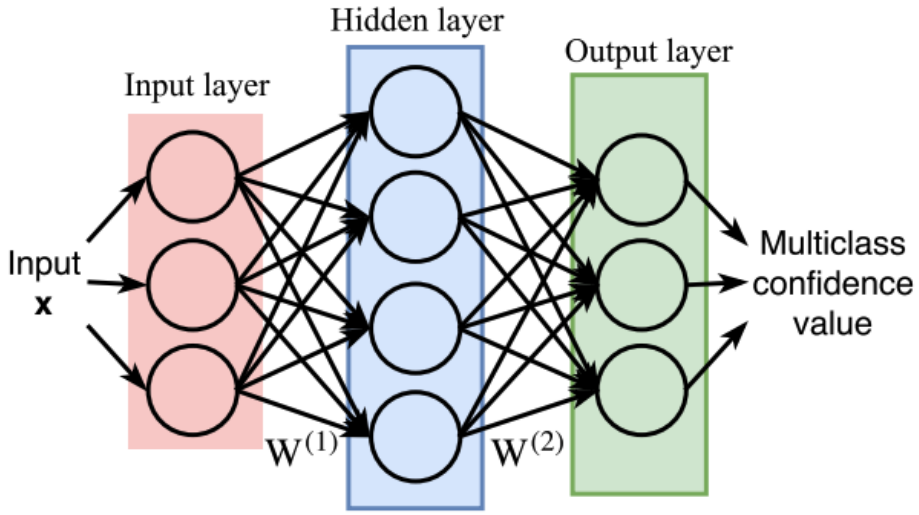
Many Others

- Condition coverage
- Multiple condition coverage
- Decision coverage
- Condition/decision coverage
- Modified condition/decision coverage

…

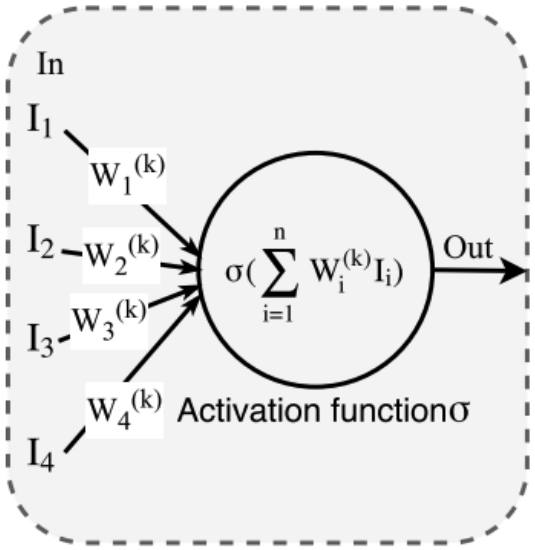
Coverage Metrics for DL

- How about deep neural networks?



Function modeled by DNN:

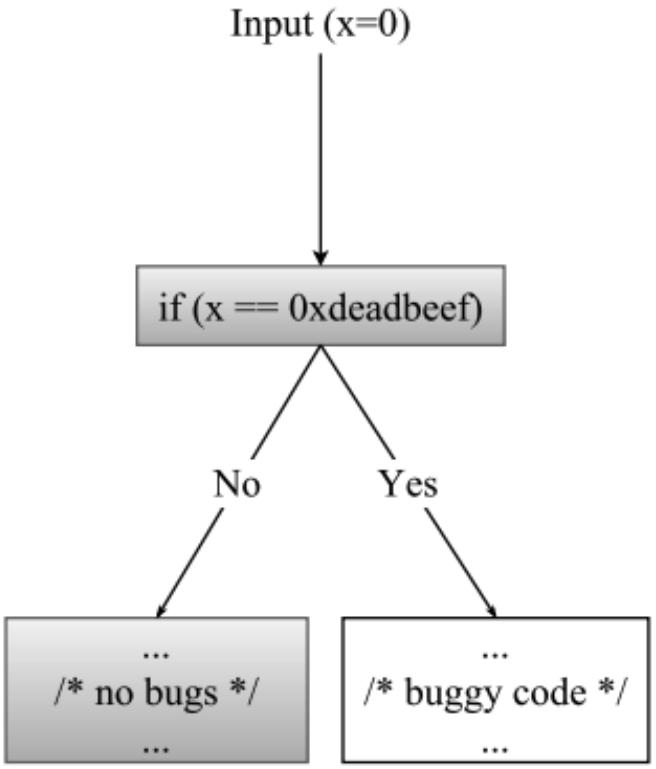
$$f(x) = \sigma(W^{(2)} \cdot \sigma(W^{(1)} \cdot x))$$

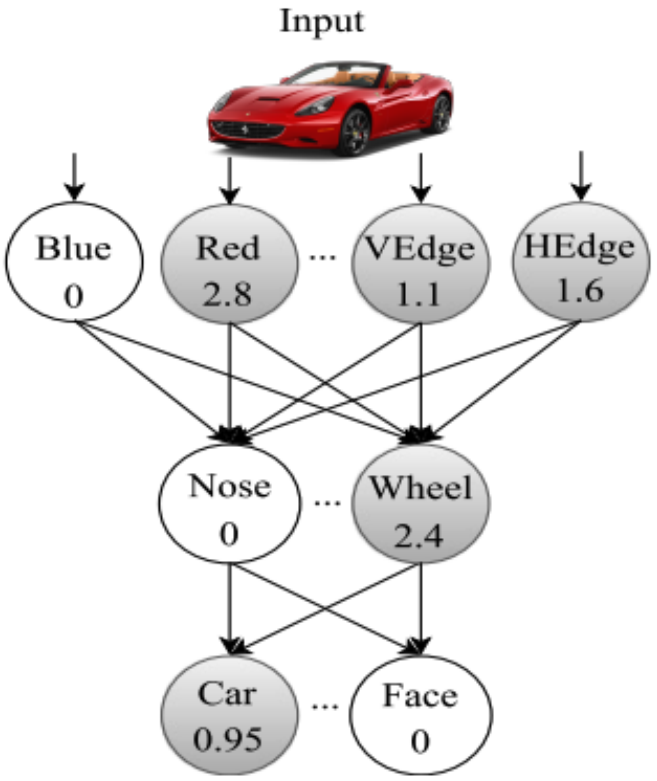


Individual neurons in layer k

[SOSP17] Pei et al. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems"

Coverage Metrics for DL





[SOSP17] Pei et al. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems"

Neuron Coverage

- How many neurons were activated

$$NCov(T, \boldsymbol{x}) = \frac{|\{n|\forall \boldsymbol{x} \in T, out(n, \boldsymbol{x}) > t\}|}{|N|}$$

- N: all neurons
- T: test inputs
- out: function returns the output value of neuron
- t: threshold

Many others

- K-multisection neuron coverage
- Neuron boundary coverage
- Strong neuron activation coverage
- Top-k neuron patterns

…

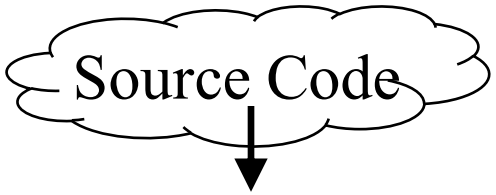
Useful?

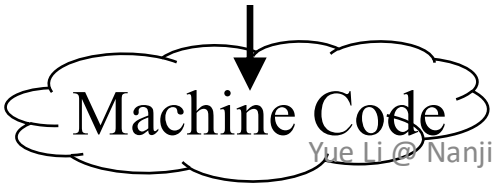
- [ESEC/FSE20] Is neuron coverage a meaningful measure for testing deep neural networks?

- [ESEC/FSE20] Correlations between deep neural network model coverage criteria and model quality

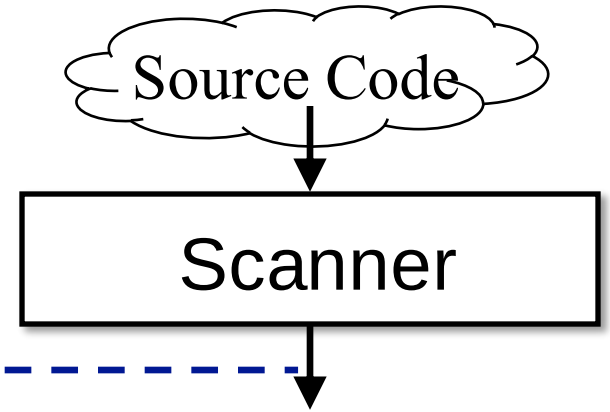
…

Compiler





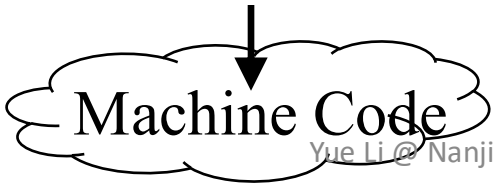
Compiler



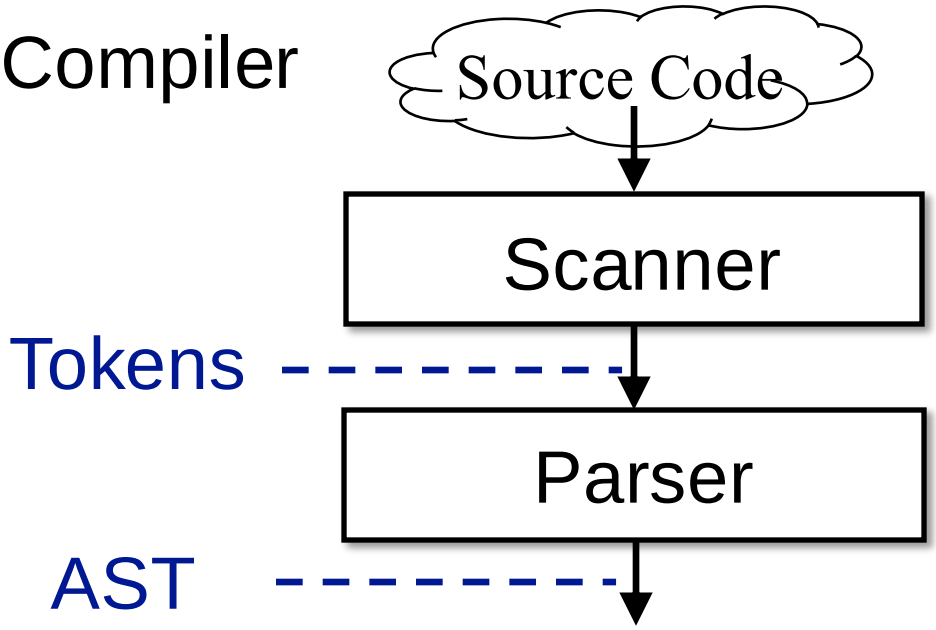
Regular Expression

Lexical Analysis

You ३ goouojd



Compiler



Lexical Analysis

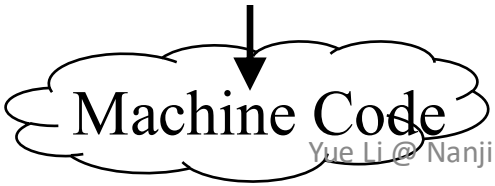
Regular Expression

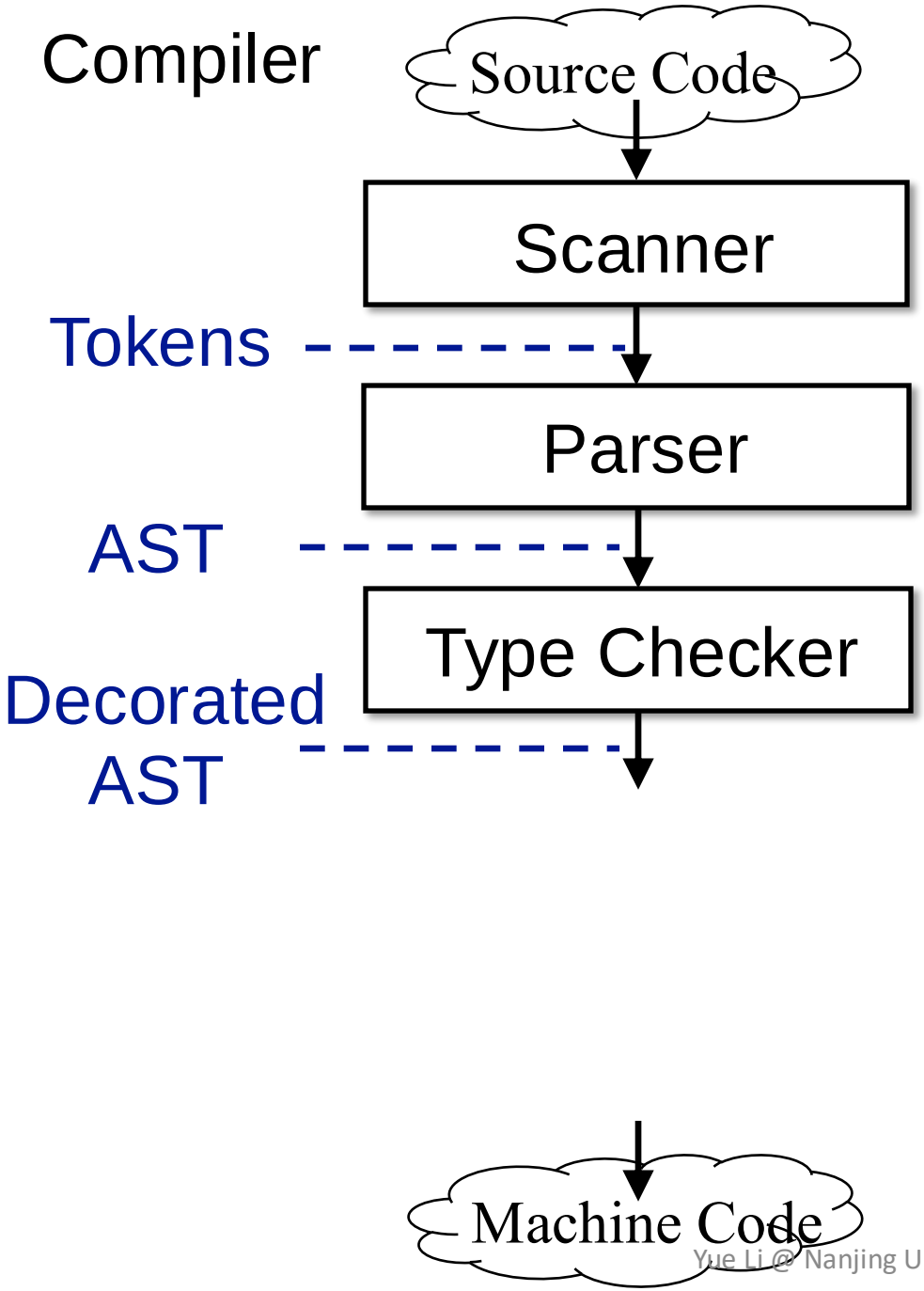
You ३ goouojd

Syntax Analysis

Context-Free Grammar

Like your hair I





Lexical Analysis

You ∋ goouojd

Regular Expression

Syntax Analysis

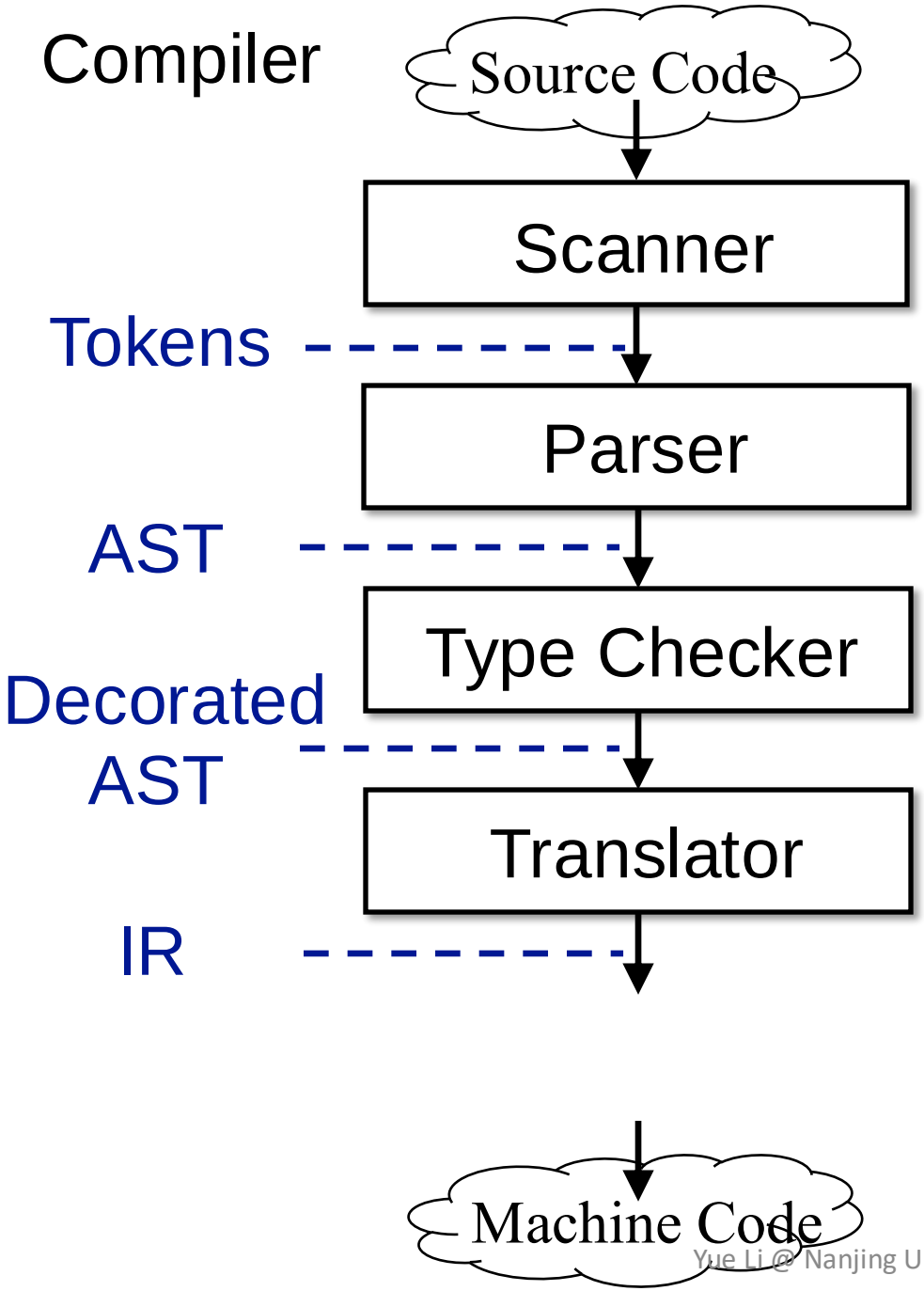
Like your hair I

Context-Free Grammar

Semantic Analysis

Apples eat you

Attribute Grammar



Lexical Analysis

You ∋ goouojd

Regular Expression

Syntax Analysis

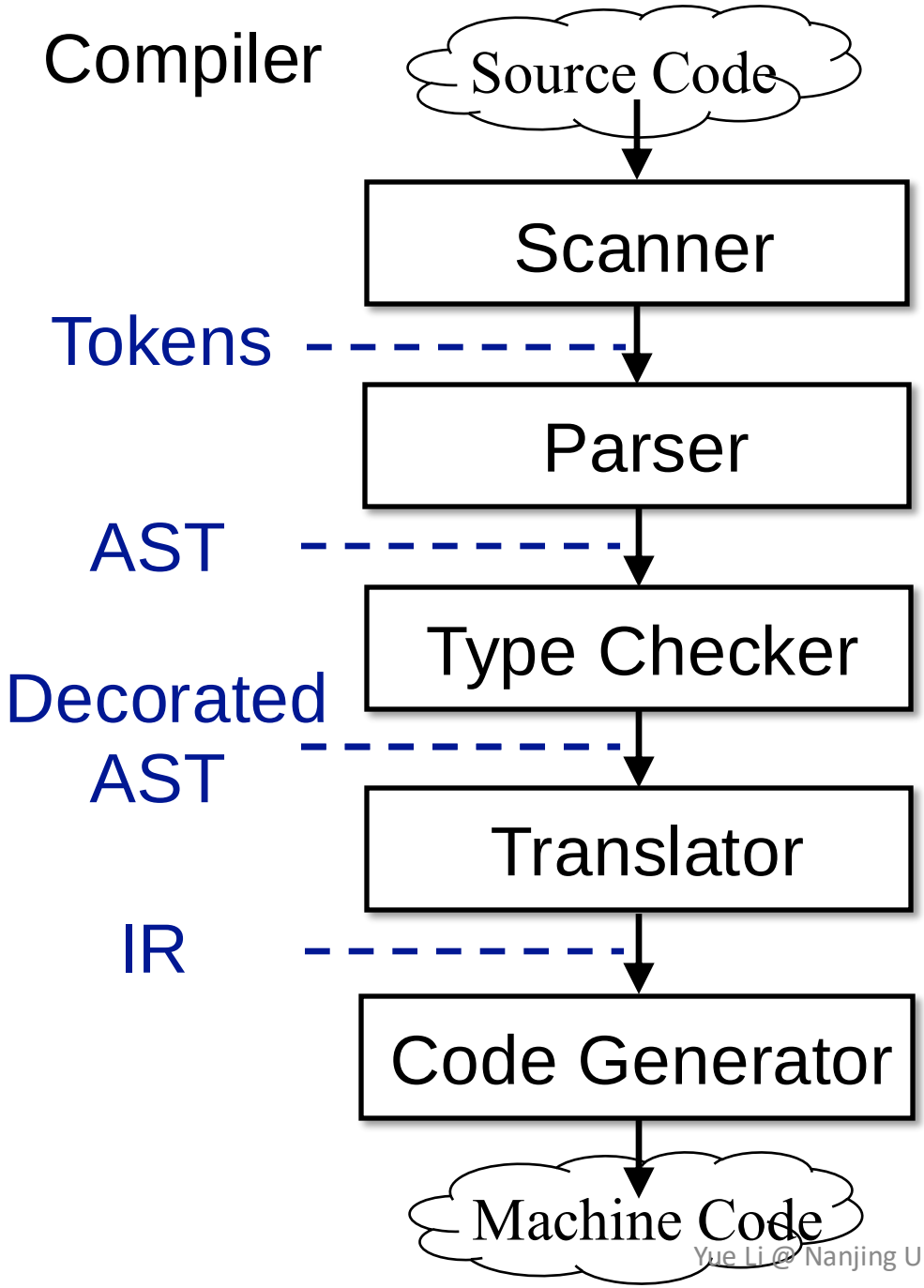
Like your hair I

Context-Free Grammar

Semantic Analysis

Apples eat you

Attribute Grammar



Lexical Analysis

You ∋ goouojd

Regular Expression

Syntax Analysis

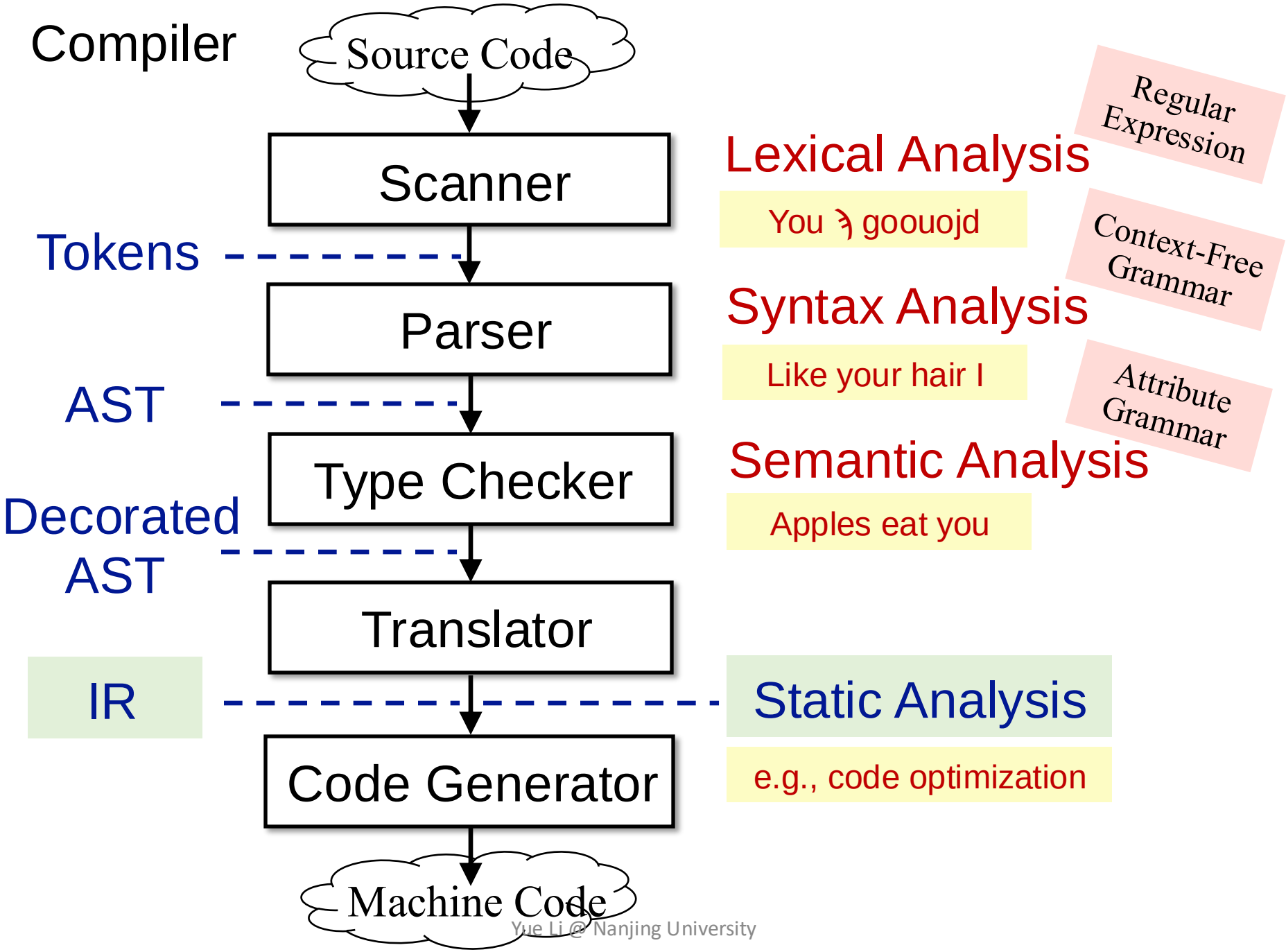
Like your hair I

Context-Free Grammar

Semantic Analysis

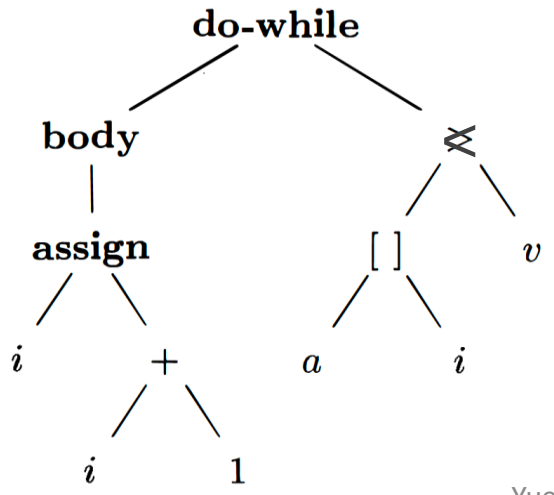
Apples eat you

Attribute Grammar


Compiler
Source Code
Scanner
Tokens
Parser
AST
Type Checker
Decorated AST
Translator
IR
Code Generator
Machine Code
Lexical Analysis
You ∋ goouojd
Regular Expression
Syntax Analysis
Like your hair I
Context-Free Grammar
Semantic Analysis
Apples eat you
Attribute Grammar
Static Analysis
e.g., code optimization

AST vs. IR

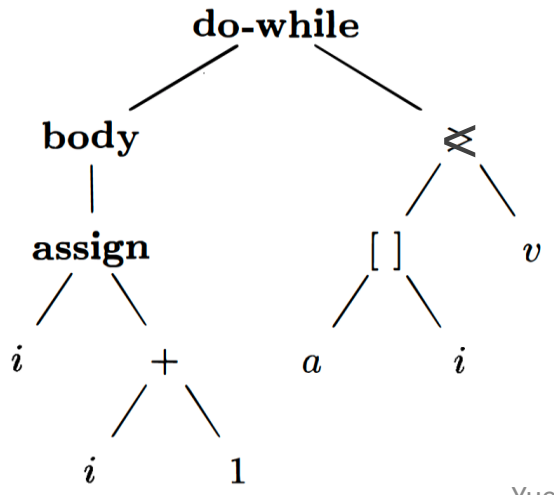
AST




```
“do i = i + 1; while ( a[i] < v);”
```

AST vs. IR

AST




```
"do i = i + 1; while ( a[i] < v);"
```

IR

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

("3-address" form)

AST vs. IR

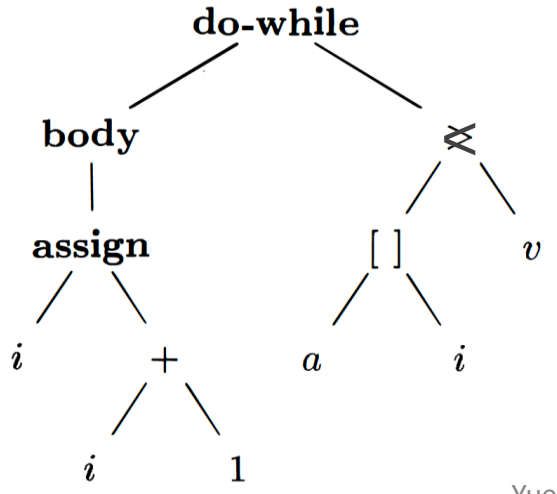
AST

- high-level and closed to grammar structure
- usually language dependent
- suitable for fast type checking
- lack of control flow information

IR

- low-level and closed to machine code
- usually language independent
- compact and uniform
- contains control flow information
- usually considered as the basis for static analysis

AST



"do i = i + 1; while (a[i] < v);"

IR

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

("3-address" form)

Intermediate Representation (IR)

- 3-Address Code (3AC)

  There is at most one operator on the right side of an instruction.

c = a + b + 3 ⇨

```
t1 = a + b
c  = t1 + 3
```

Intermediate Representation (IR)

- 3-Address Code (3AC)

  There is at most one operator on the right side of an instruction.

c = a + b + 3 ⇨

t1 = a + b
c = t1 + 3

Intermediate Representation (IR)

- 3-Address Code (3AC)

  There is at most one operator on the right side of an instruction.

c = a + b + 3 ⇨

t1 = a + b
c = t1 + 3

Each 3AC contains at most 3 addresses

Why called 3-address?

Intermediate Representation (IR)

- 3-Address Code (3AC)

  There is at most one operator on the right side of an instruction.

c = a + b + 3 ⇨

t1 = a + b
c = t1 + 3

Each 3AC contains at most 3 addresses

Address can be one of the following:

- Name: a, b, c
- Constant: 3
- Compiler-generated temporary: t1

Intermediate Representation (IR)

- 3-Address Code (3AC)

  There is at most one operator on the right side of an instruction.

c = a + b + 3 ⇨

t1 = a + b
c = t1 + 3

Each 3AC contains at most 3 addresses

Why called 3-address?

Address can be one of the following:

- Name: a, b, c
- Constant: 3
- Compiler-generated temporary: t1

Each type of instructions has its own 3AC form

Some Common 3AC Forms

- x = y *bop* z
- x = *uop* y
- x = y
- goto L
- if x goto L
- if x *rop* y goto L

x, y, z: addresses
bop: binary arithmetic or logical operation
uop: unary operation (minus, negation, casting)
L: a label to represent a program location
rop: relational operator (>, <, ==, >=, <=, etc.)
goto L: unconditional jump
if … goto L: conditional jump

Some Common 3AC Forms

- x = y *bop* z
- x = *uop* y
- x = y
- goto L
- if x goto L
- if x *rop* y goto L

x, y, z: addresses
bop: binary arithmetic or logical operation
uop: unary operation (minus, negation, casting)
L: a label to represent a program location
rop: relational operator (>, <, ==, >=, <=, etc.)
goto L: unconditional jump
if … goto L: conditional jump

Let's see some more real-world complicated forms

Soot and Its IR: Jimple

- Soot

  Most popular static analysis framework for Java

  https://github.com/Sable/soot

  https://github.com/Sable/soot/wiki/Tutorials

  Soot's IR is Jimple: typed 3-address code

```java
package nju.sa.examples;
public class DoWhileLoop3AC {
    public static void main(String[] args) {
        int[] arr = new int[10];
        int i = 0;
        do {
            i = i + 1;
        } while (arr[i] < 10);
    }
}
```

Java Src

```java
package nju.sa.examples;
public class DoWhileLoop3AC {
    public static void main(String[] args) {
        int[] arr = new int[10];
        int i = 0;
        do {
            i = i + 1;
        } while (arr[i] < 10);
    }
}
```

Java Src

Do-While Loop

```
public static void main(java.lang.String[])
{
    java.lang.String[] r0;
    int[] r1;
    int $i0, i1;

    r0 := @parameter0: java.lang.String[];

    r1 = newarray (int)[10];

    i1 = 0;

 label1:
    i1 = i1 + 1;

    $i0 = r1[i1];

    if $i0 < 10 goto label1;

    return;
}
```

3AC(jimple)

```java
package nju.sa.examples;
public class MethodCall3AC {

    String foo(String para1, String para2) {
        return para1 + " " + para2;
    }

    public static void main(String[] args) {
        MethodCall3AC mc = new MethodCall3AC();
        String result = mc.foo("hello", "world");
    }
}
```

Java Src

```java
package nju.sa.examples;
public class MethodCall3AC {

    String foo(String para1, String para2) {
        return para1 + " " + para2;
    }

    public static void main(String[] args) {
        MethodCall3AC mc = new MethodCall3AC();
        String result = mc.foo("hello", "world");
    }
}
```

Java Src


```
java.lang.String foo(java.lang.String, java.lang.String)
{
    nju.sa.examples.MethodCall3AC r0;
    java.lang.String r1, r2, $r7;
    java.lang.StringBuilder $r3, $r4, $r5, $r6;

    r0 := @this: nju.sa.examples.MethodCall3AC;

    r1 := @parameter0: java.lang.String;

    r2 := @parameter1: java.lang.String;

    $r3 = new java.lang.StringBuilder;

    specialinvoke $r3.<java.lang.StringBuilder: void <init>()>();

    $r4 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(r1);

    $r5 = virtualinvoke $r4.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(" ");

    $r6 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(r2);

    $r7 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.String toString()>();

    return $r7;
}
```

3AC (jimple)

```java
package nju.sa.examples;
public class MethodCall3AC {

    String foo(String para1, String para2) {
        return para1 + " " + para2;
    }

    public static void main(String[] args) {
        MethodCall3AC mc = new MethodCall3AC();
        String result = mc.foo("hello", "world");
    }
}
```

Java Src


```
public static void main(java.lang.String[])
{
    java.lang.String[] r0;
    nju.sa.examples.MethodCall3AC $r3;

    r0 := @parameter0: java.lang.String[];

    $r3 = new nju.sa.examples.MethodCall3AC;

    specialinvoke $r3.<nju.sa.examples.MethodCall3AC: void <init>()>();

    virtualinvoke $r3.<nju.sa.examples.MethodCall3AC:
                    java.lang.String foo(java.lang.String,java.lang.String)>("hello", "world");

    return;
}
```

3AC(jimple)


```java
package nju.sa.examples;
public class Class3AC {

    public static final double pi = 3.14;
    public static void main(String[] args) {

    }
}
```

Java Src

```
public class nju.sa.examples.Class3AC extends java.lang.Object
{
    public static final double pi;

    public void <init>()
    {
        nju.sa.examples.Class3AC r0;

        r0 := @this: nju.sa.examples.Class3AC;

        specialinvoke r0.<java.lang.Object: void <init>()>();

        return;
    }

    public static void main(java.lang.String[])
    {
        java.lang.String[] r0;

        r0 := @parameter0: java.lang.String[];

        return;
    }

    public static void <clinit>()
    {
        <nju.sa.examples.Class3AC: double pi> = 3.14;

        return;
    }
}
```

3AC(jimple)

Class

```
package nju.sa.examples;
public class Class3AC {

    public static final double pi = 3.14;
    public static void main(String[] args) {

    }
}
```

Java Src

Yue LI @ Nanjing University

Control Flow Analysis

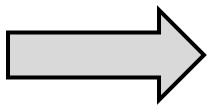
- Usually refer to building Control Flow Graph (CFG)

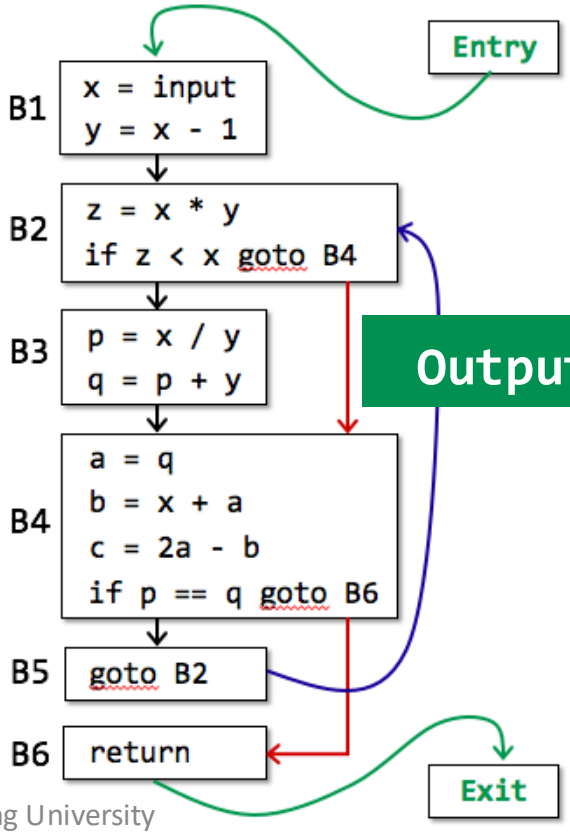
Control Flow Analysis

- Usually refer to building Control Flow Graph (CFG)

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```





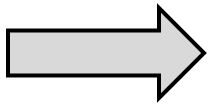
Output: CFG of *P*

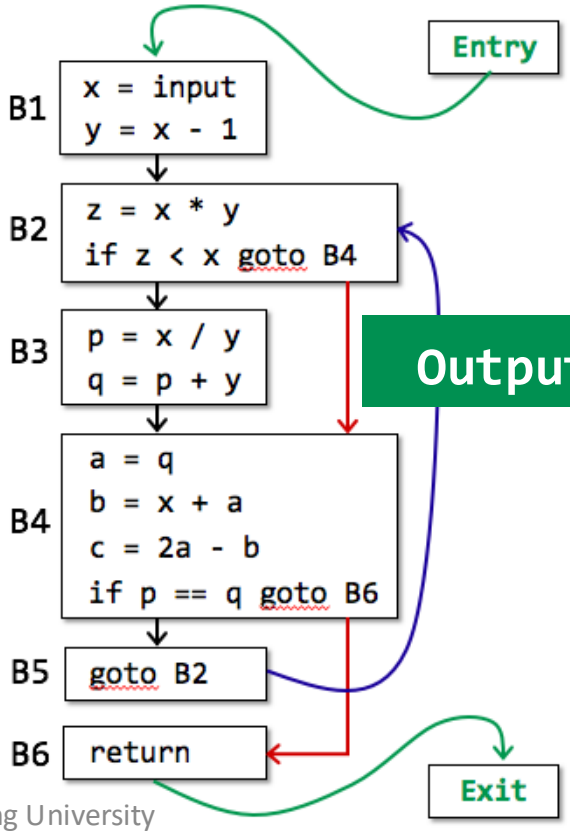
Control Flow Analysis

- Usually refer to building Control Flow Graph (CFG)
- CFG serves as the basic structure for static analysis

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```





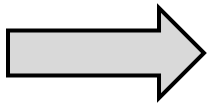
Output: CFG of *P*

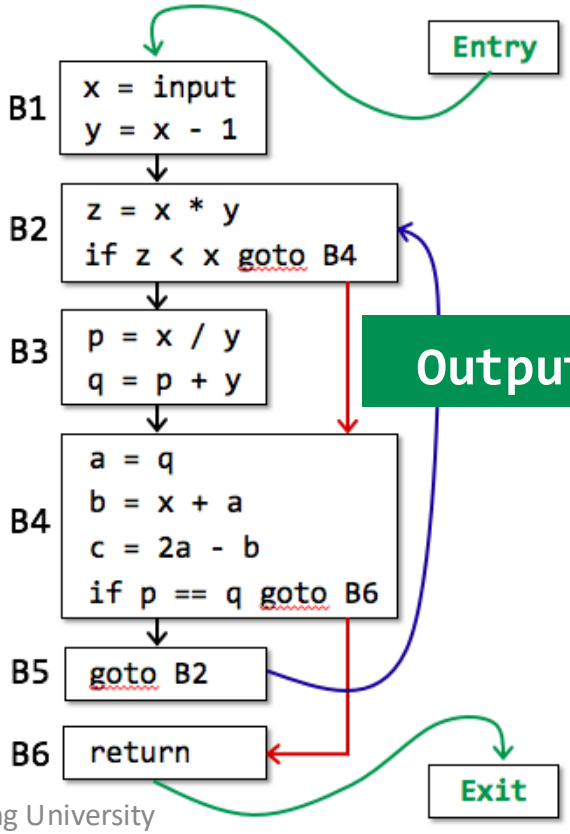
Control Flow Analysis

- Usually refer to building Control Flow Graph (CFG)
- CFG serves as the basic structure for static analysis
- The node in CFG can be an individual 3-address instruction, or (usually) a Basic Block (BB)

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```





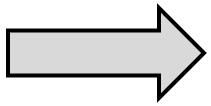
Output: CFG of *P*

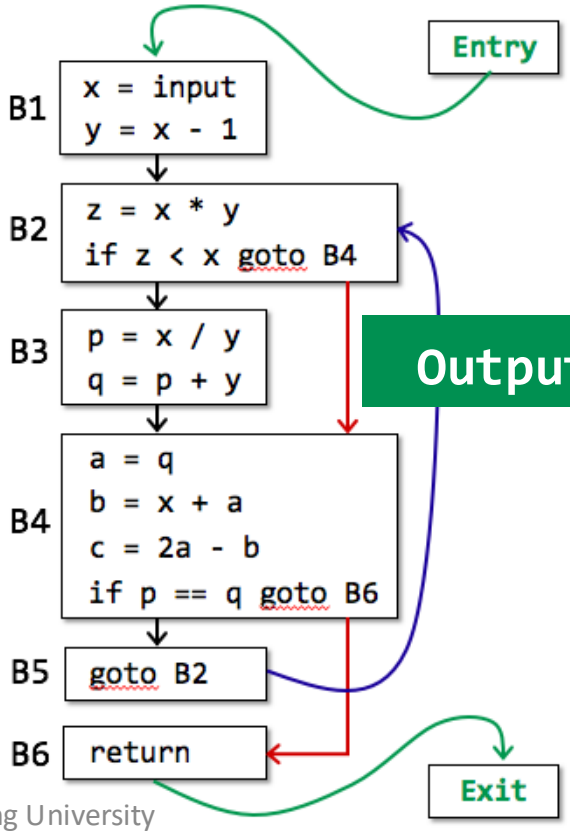
Control Flow Analysis

- Usually refer to building Control Flow Graph (CFG)
- CFG serves as the basic structure for static analysis
- The node in CFG can be an individual 3-address instruction, or (usually) a Basic Block (BB)

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```





Output: CFG of *P*

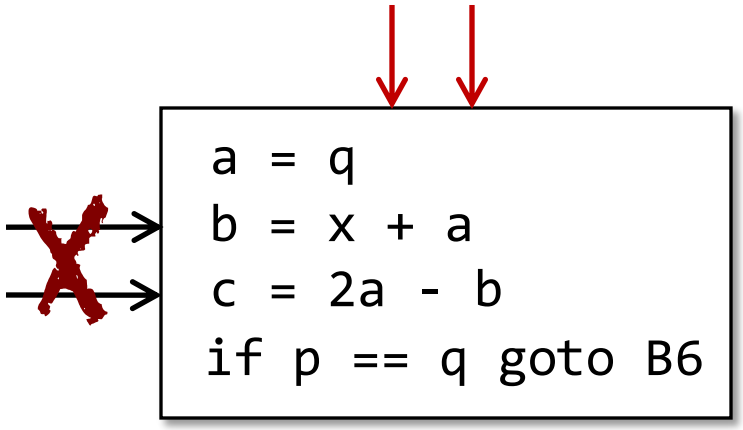
Basic Blocks (BB)

- Basic blocks (BB) are maximal sequences of consecutive three-address instructions with the properties that

```
a = q
b = x + a
c = 2a - b
if p == q goto B6
```

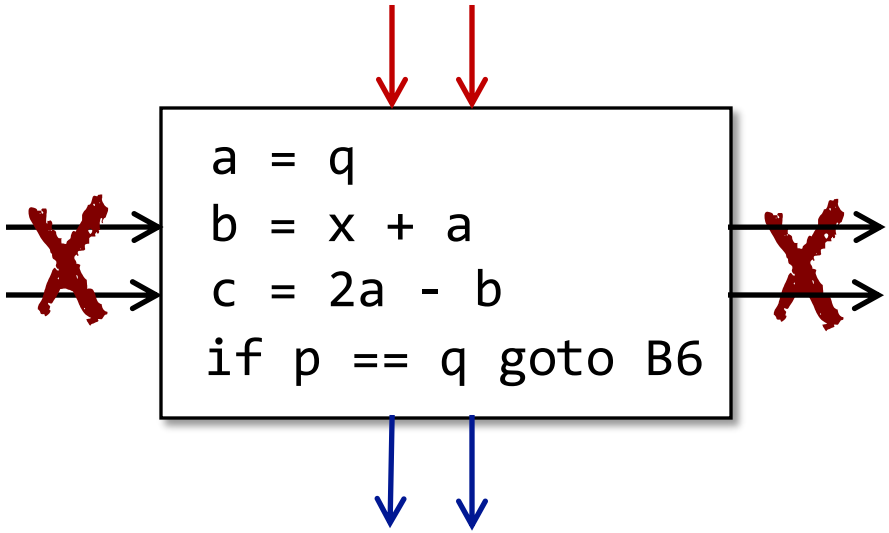
Basic Blocks (BB)

- Basic blocks (BB) are maximal sequences of consecutive three-address instructions with the properties that
  - It can be entered only at the beginning, i.e., *the first instruction in the block*



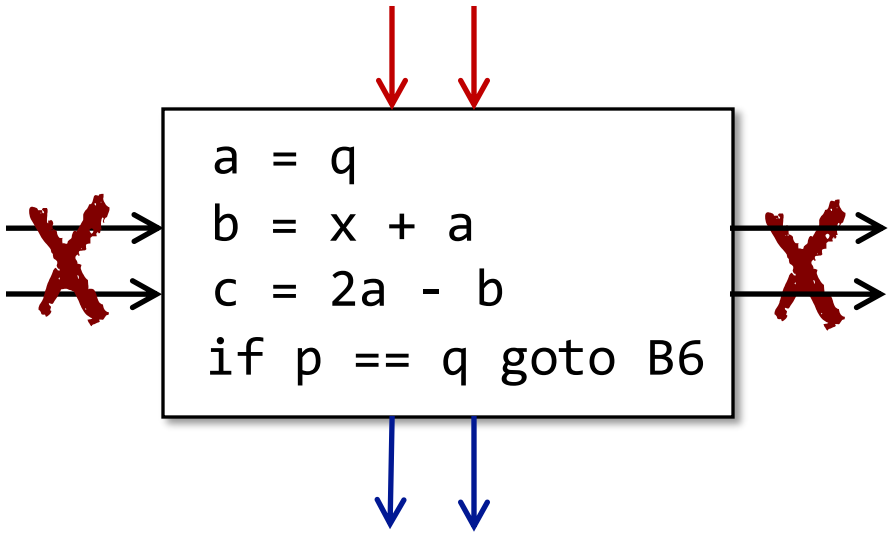
Basic Blocks (BB)

- Basic blocks (BB) are maximal sequences of consecutive three-address instructions with the properties that
  - It can be entered only at the beginning, i.e., *the first instruction in the block*
  - It can be exited only at the end, i.e., *the last instruction in the block*



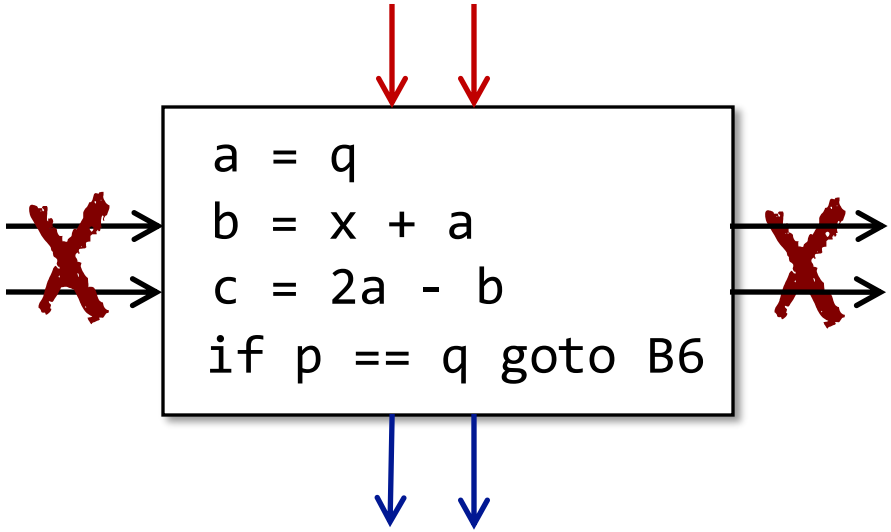
Now try to design the algorithm to build BBs by yourself!

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```



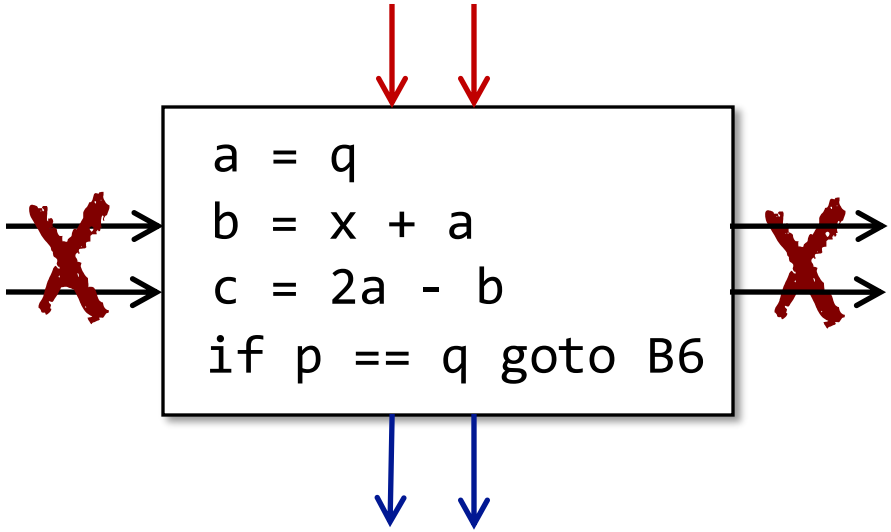
```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Now try to design the algorithm to build BBs by yourself!



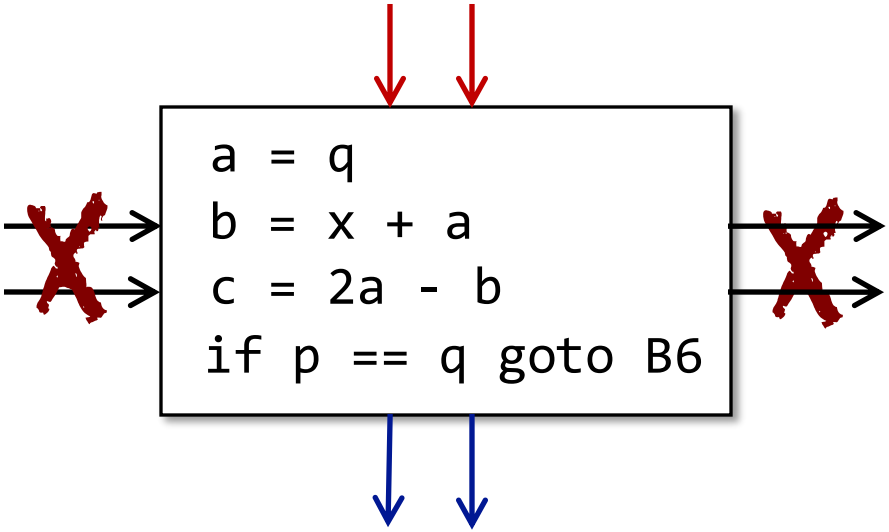
```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Now try to design the algorithm to build BBs by yourself!



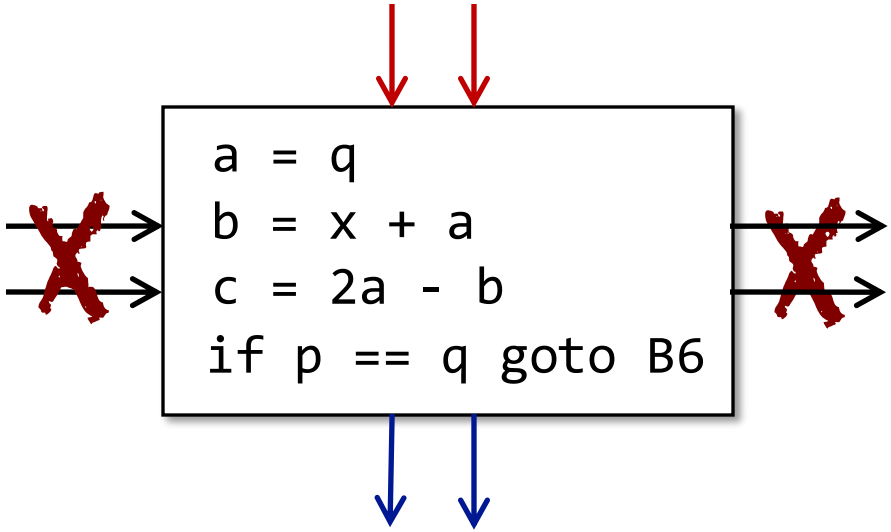
```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Now try to design the algorithm to build BBs by yourself!



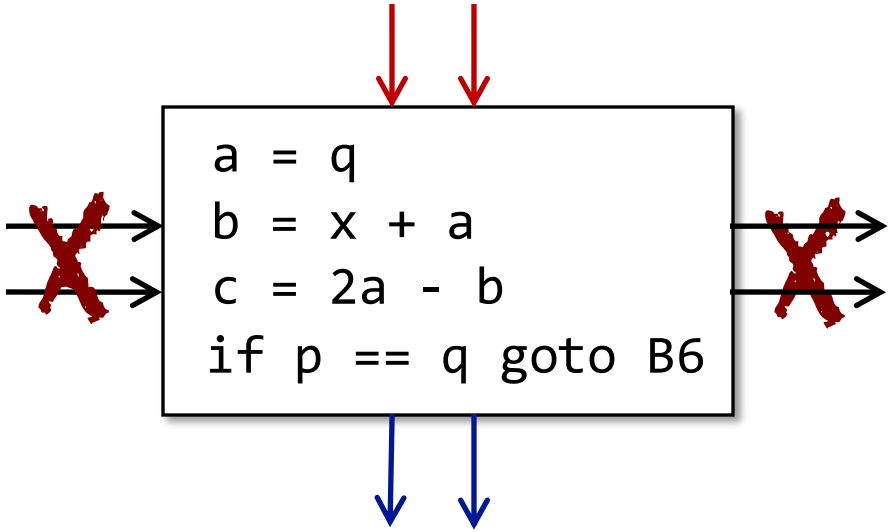
```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Now try to design the algorithm to build BBs by yourself!



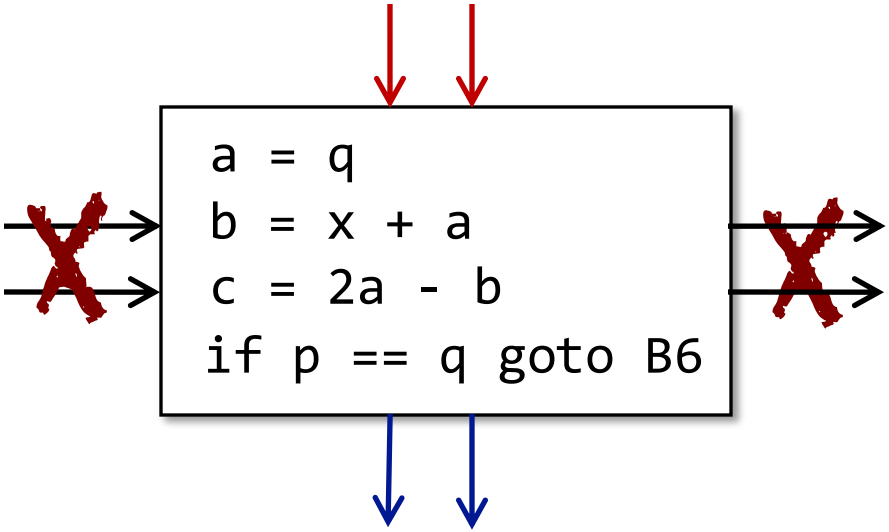
```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Now try to design the algorithm to build BBs by yourself!



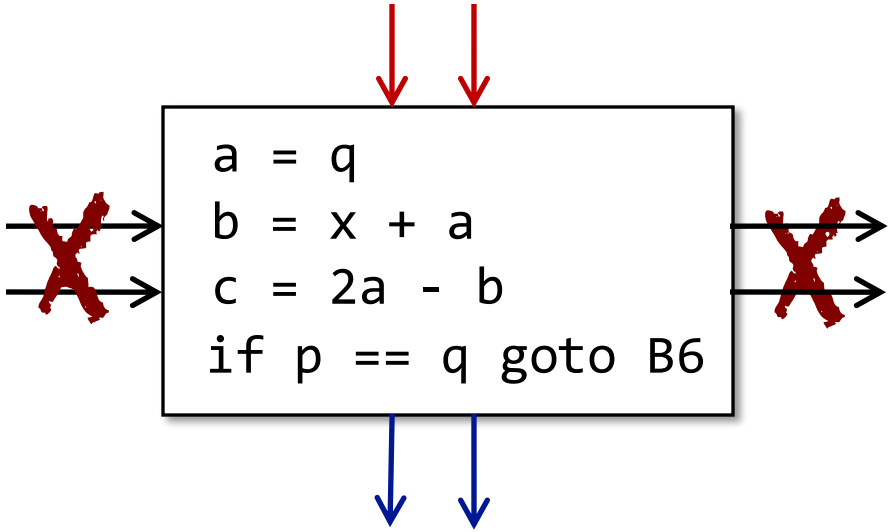
Now try to design the algorithm to build BBs by yourself!

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```



```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Now try to design the algorithm to build BBs by yourself!



How to build Basic Blocks?

INPUT: A sequence of three-address instructions of *P*

OUTPUT: A list of basic blocks of *P*

METHOD: (1) Determine the leaders in *P*

- The first instruction in *P* is a leader
- Any target instruction of a conditional or unconditional jump is a leader
- Any instruction that immediately follows a conditional or unconditional jump is a leader

(2) Build BBs for P

- A BB consists of a leader and all its subsequent instructions until the next leader

How to build Basic Blocks?

INPUT: A sequence of three-address instructions of *P*

OUTPUT: A list of basic blocks of *P*

METHOD: (1) Determine the leaders in *P*

- The first instruction in *P* is a leader
- Any target instruction of a conditional or unconditional jump is a leader
- Any instruction that immediately follows a conditional or unconditional jump is a leader

(2) Build BBs for P

- A BB consists of a leader and all its subsequent instructions until the next leader

Input: 3AC of *P*

Output: BBs of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- The first instruction in *P* is a leader

Input: 3AC of *P*

```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- The first instruction in *P* is a leader

Input: 3AC of *P*

```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)

- Any target instruction of a conditional or unconditional jump is a leader

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)

- Any target instruction of a conditional or unconditional jump is a leader

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)

- Any instruction that immediately follows a conditional or unconditional jump is a leader

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)

- Any instruction that immediately follows a conditional or unconditional jump is a leader

Input: 3AC of *P*

```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)
- (5),(11),(12)

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)
- (5),(11),(12)

Leaders: (1), (3), (5), (7), (11), (12)

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)
- (5),(11),(12)

Leaders: (1), (3), (5), (7), (11), (12)

(2) Build BBs for P

- A BB consists of a leader and all its subsequent instructions until the next leader

Input: 3AC of *P*

```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)
- (5),(11),(12)

Leaders: (1), (3), (5), (7), (11), (12)

(2) Build BBs for P

- A BB consists of a leader and all its subsequent instructions until the next leader

- B1 {(1)}
- B2 {(3)}
- B3 {(5)}
- B4 {(7)}
- B5 {(11)}
- B6 {(12)}

Input: 3AC of *P*

```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

(1) Determine the leaders in *P*

- (1)
- (3),(7),(12)
- (5),(11),(12)

Leaders: (1), (3), (5), (7), (11), (12)

(2) Build BBs for P

- A BB consists of a leader and all its subsequent instructions until the next leader

- B1 {(1),(2)}
- B2 {(3),(4)}
- B3 {(5),(6)}
- B4 {(7),(8),(9),(10)}
- B5 {(11)}
- B6 {(12)}

Input: 3AC of *P*

```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

- B1 {(1),(2)}
- B2 {(3),(4)}
- B3 {(5),(6)}
- B4 {(7),(8),(9),(10)}
- B5 {(11)}
- B6 {(12)}

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

Output: BBs of *P*

B1
```
(1) x = input
(2) y = x - 1
```

B2
```
(3) z = x * y
(4) if z < x goto (7)
```

B3
```
(5) p = x / y
(6) q = p + y
```

B4
```
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
```

B5
```
(11) goto (3)
```

B6
```
(12) return
```

Input: 3AC of *P*

```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

How to build CFG on top of BBs?

Output: BBs of *P*

B1
```
(1) x = input
(2) y = x - 1
```

B2
```
(3) z = x * y
(4) if z < x goto (7)
```

B3
```
(5) p = x / y
(6) q = p + y
```

B4
```
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
```

B5
```
(11) goto (3)
```

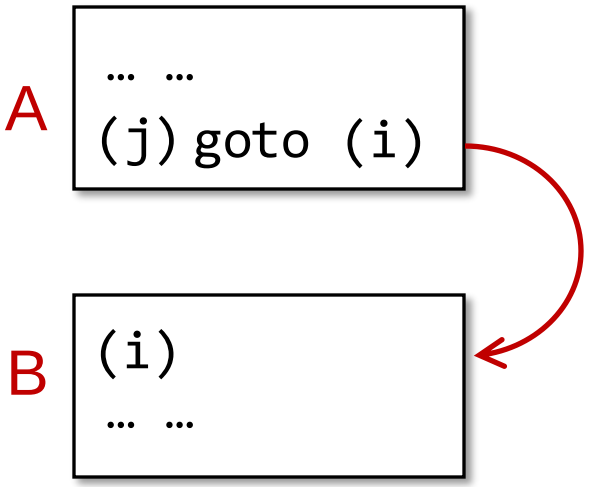
B6
```
(12) return
```

Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if

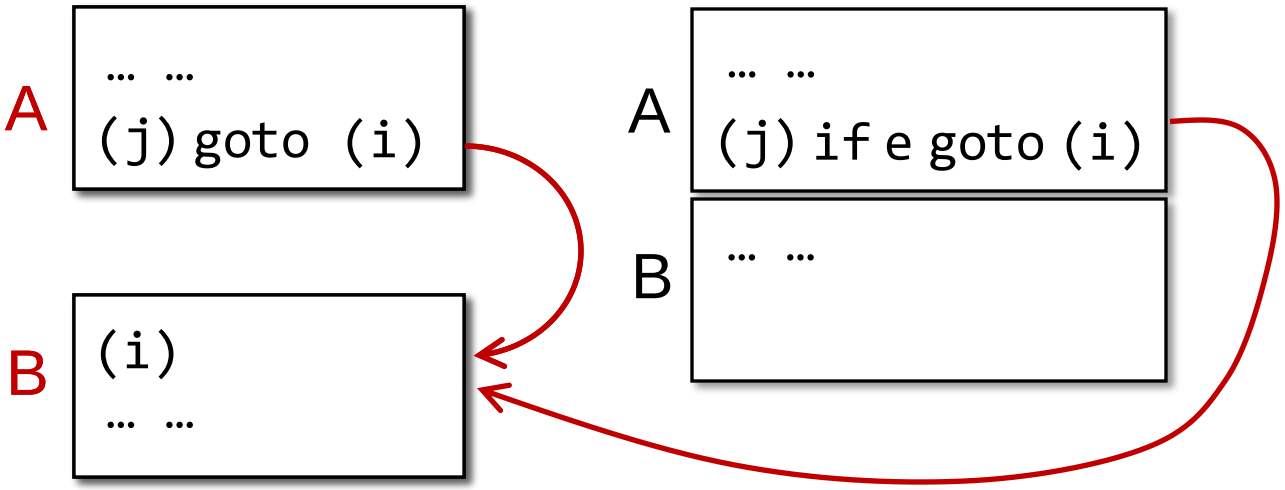
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if



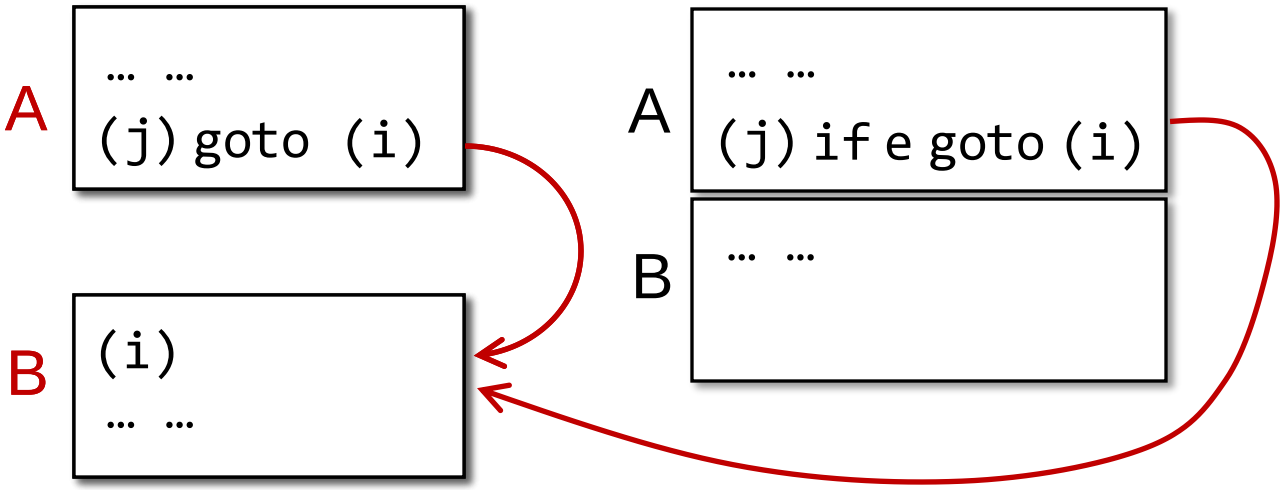
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if



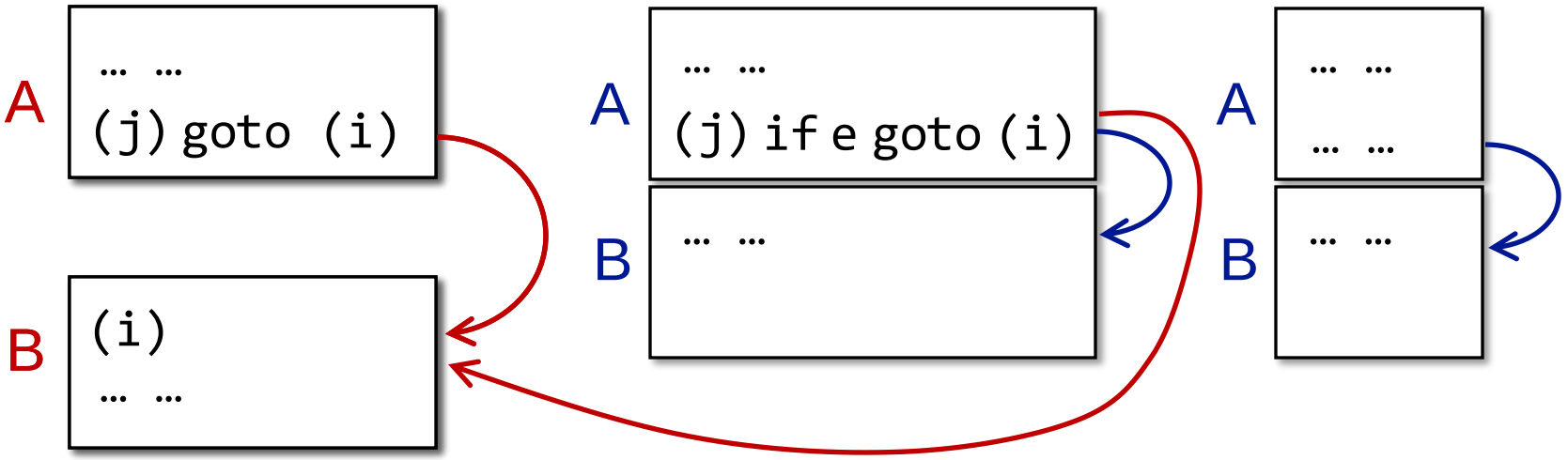
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if
  - There is a conditional or unconditional jump from the end of A to the beginning of B



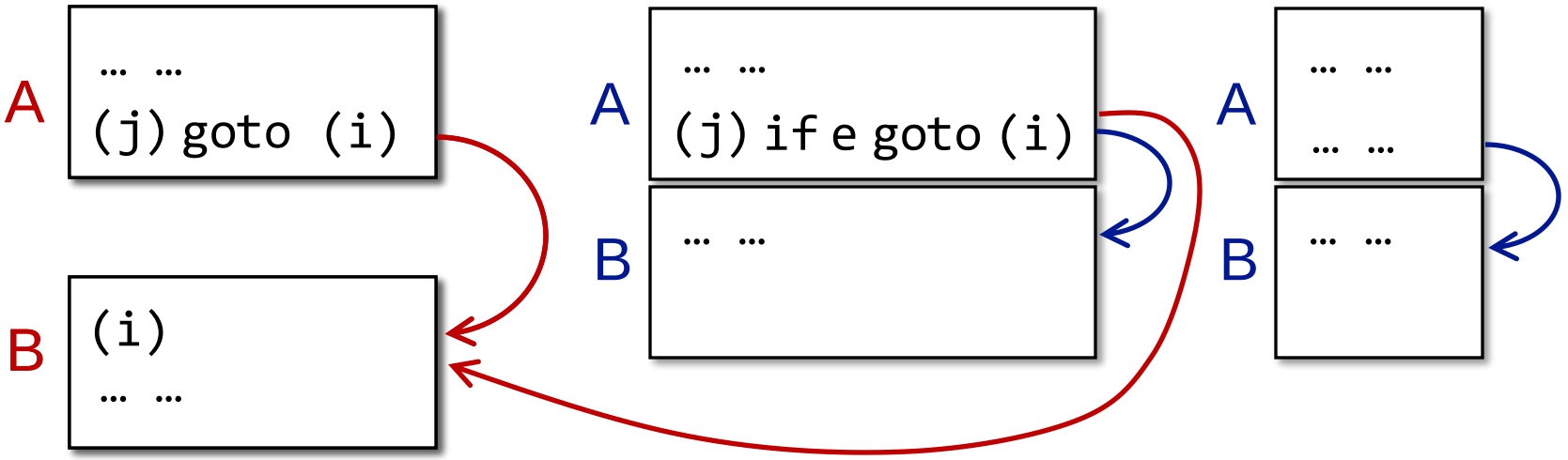
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if
  - There is a conditional or unconditional jump from the end of A to the beginning of B



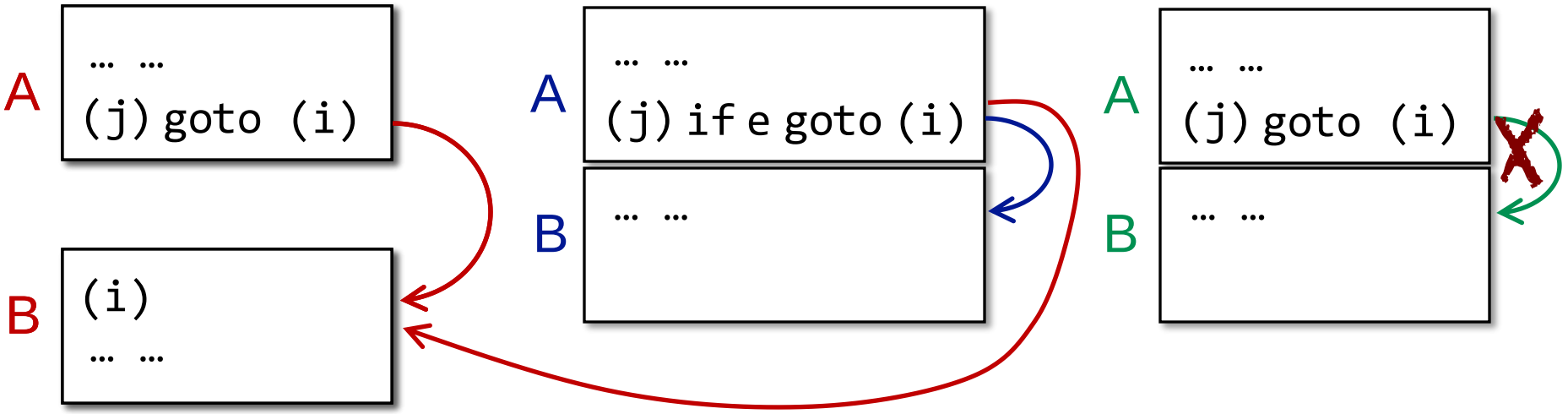
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if
  - There is a conditional or unconditional jump from the end of A to the beginning of B
  - B immediately follows A in the original order of instructions



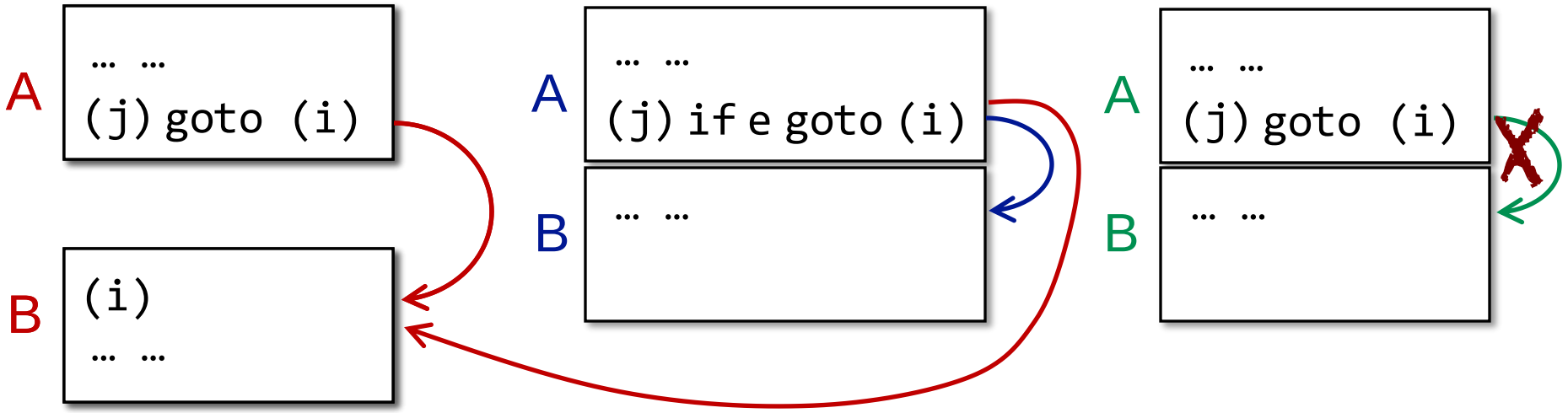
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if
  - There is a conditional or unconditional jump from the end of A to the beginning of B
  - B immediately follows A in the original order of instructions



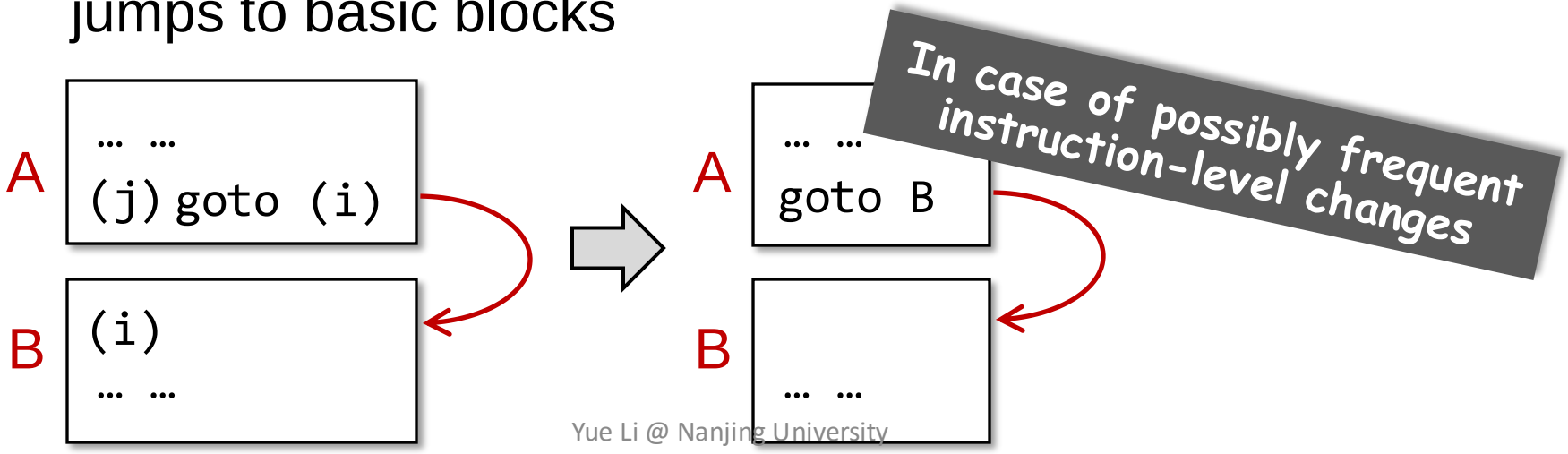
Control Flow Graph (CFG)

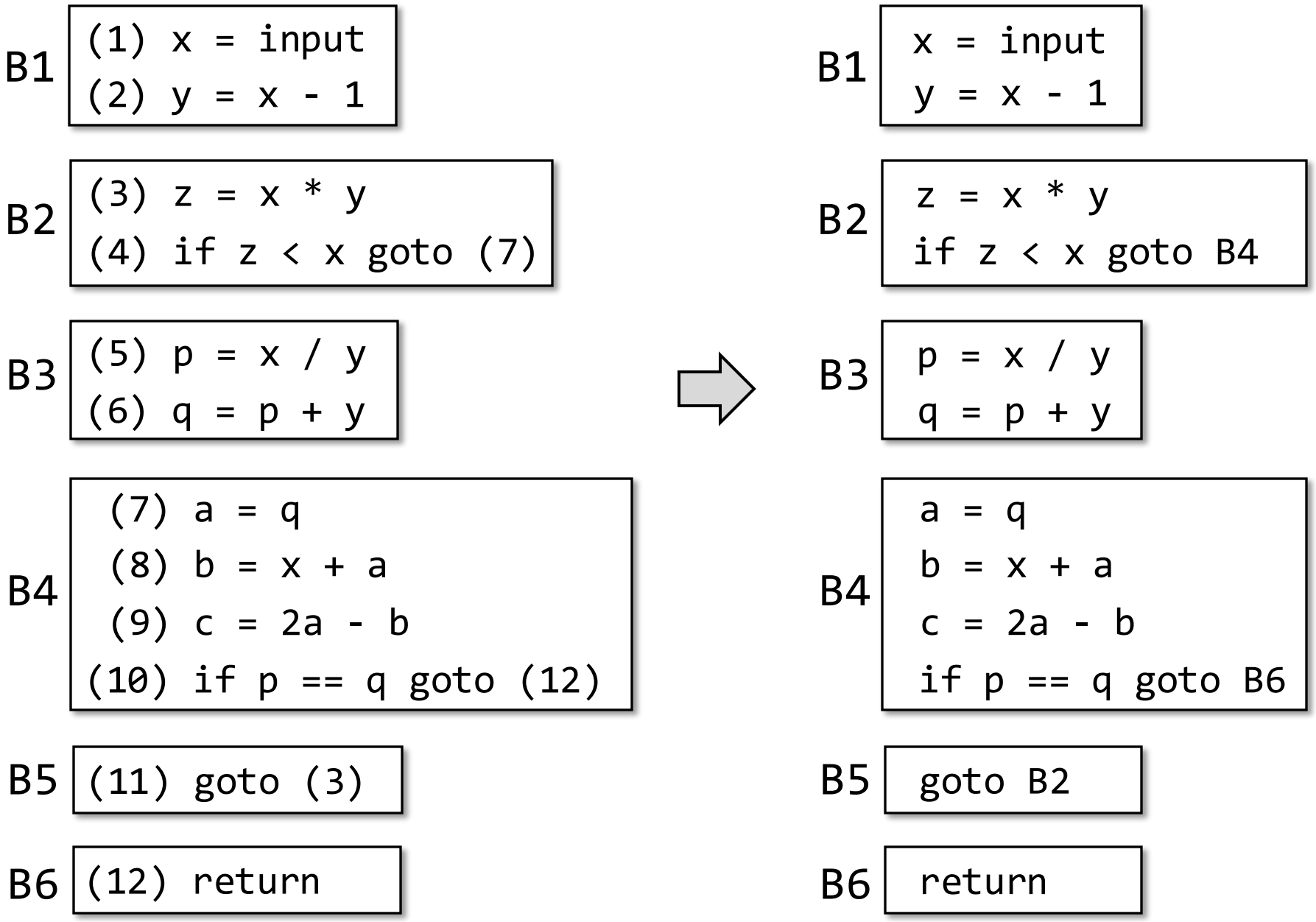
- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if
  - There is a conditional or unconditional jump from the end of A to the beginning of B
  - B immediately follows A in the original order of instructions and A does not end in an unconditional jump



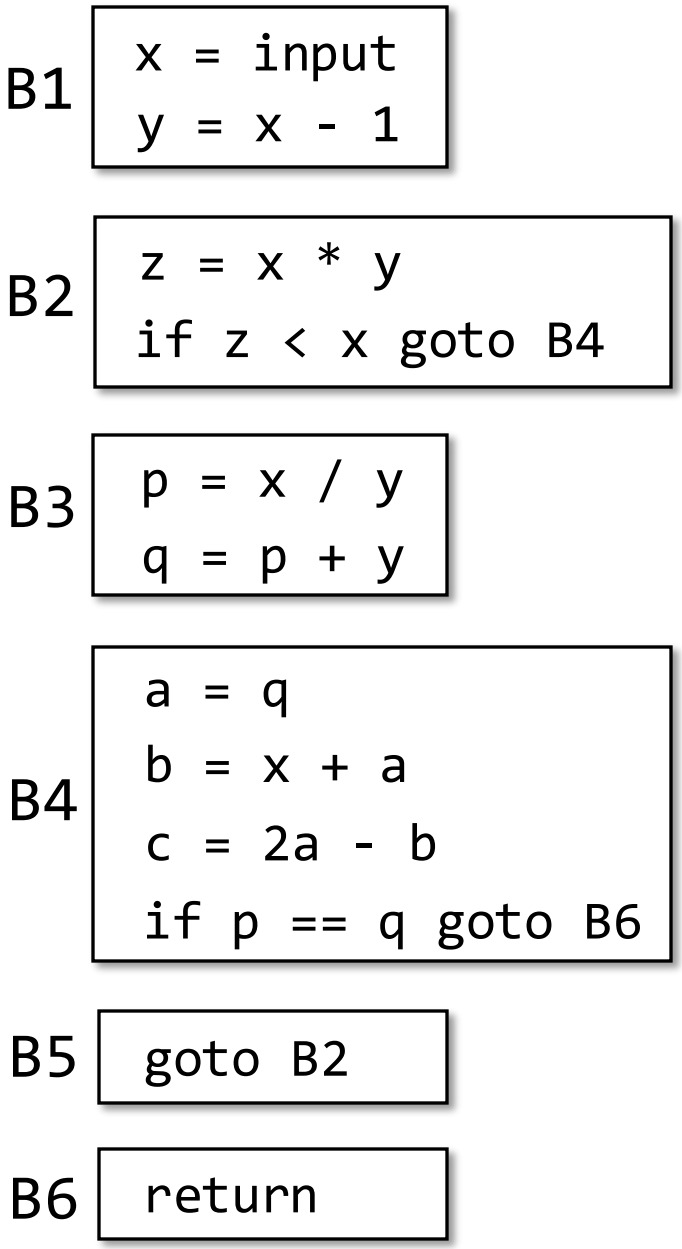
Control Flow Graph (CFG)

- The nodes of CFG are basic blocks
- There is an edge from block A to block B if and only if
  - There is a conditional or unconditional jump from the end of A to the beginning of B
  - B immediately follows A in the original order of instructions and A does not end in an unconditional jump
- It is normal to replace the jumps to instruction labels by jumps to basic blocks



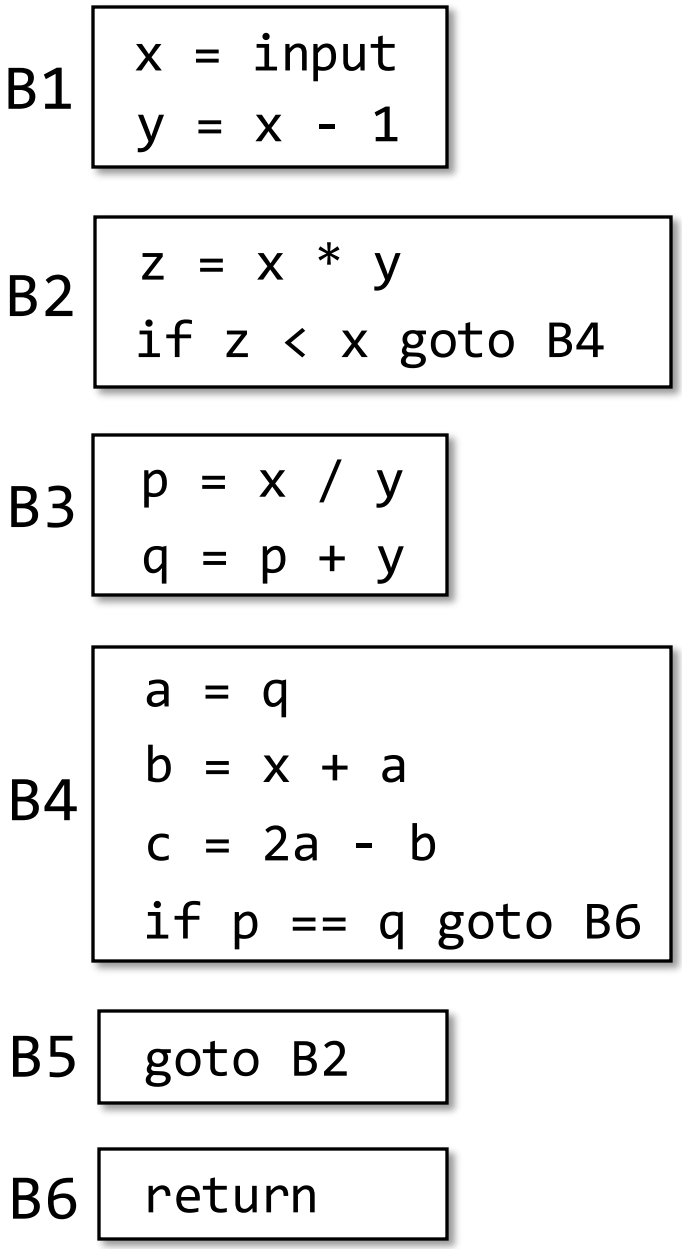

B1
(1) x = input
(2) y = x - 1
B2
(3) z = x * y
(4) if z < x goto (7)
B3
(5) p = x / y
(6) q = p + y
B4
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
B5
(11) goto (3)
B6
(12) return
B1
x = input
y = x - 1
B2
z = x * y
if z < x goto B4
B3
p = x / y
q = p + y
B4
a = q
b = x + a
c = 2a - b
if p == q goto B6
B5
goto B2
B6
return

Add edges in CFG



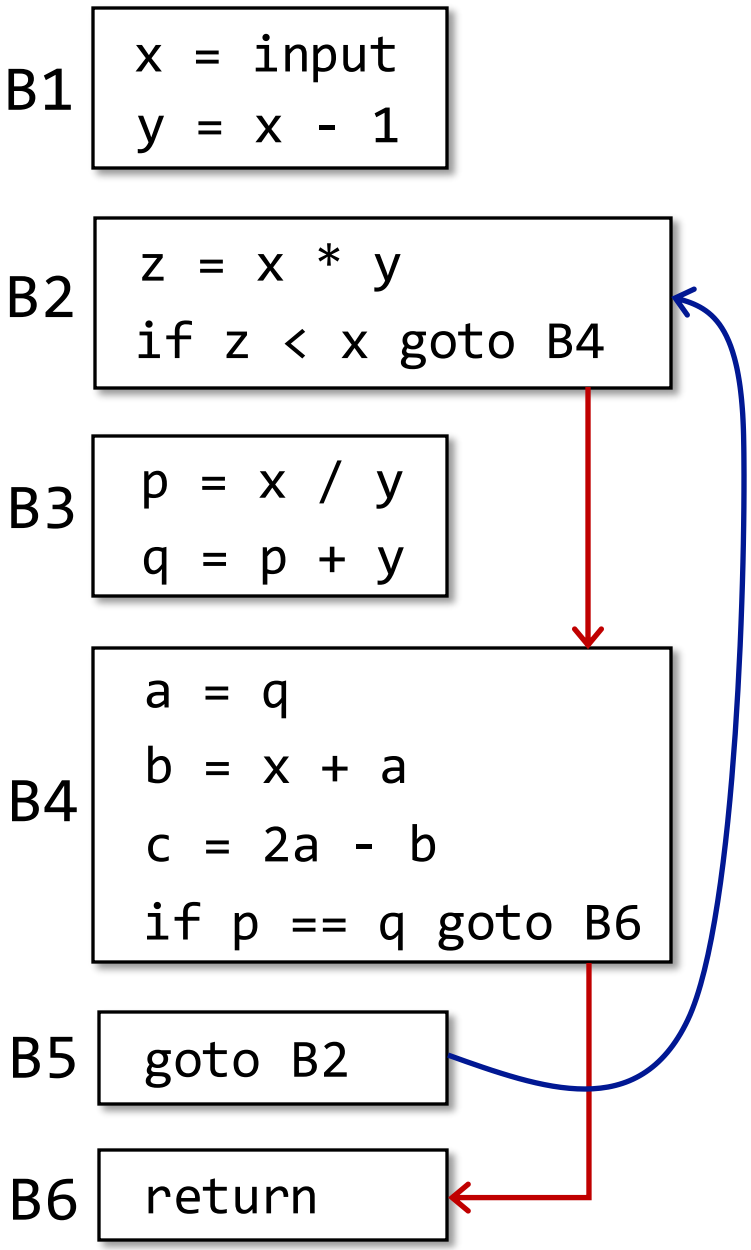
Add edges in CFG

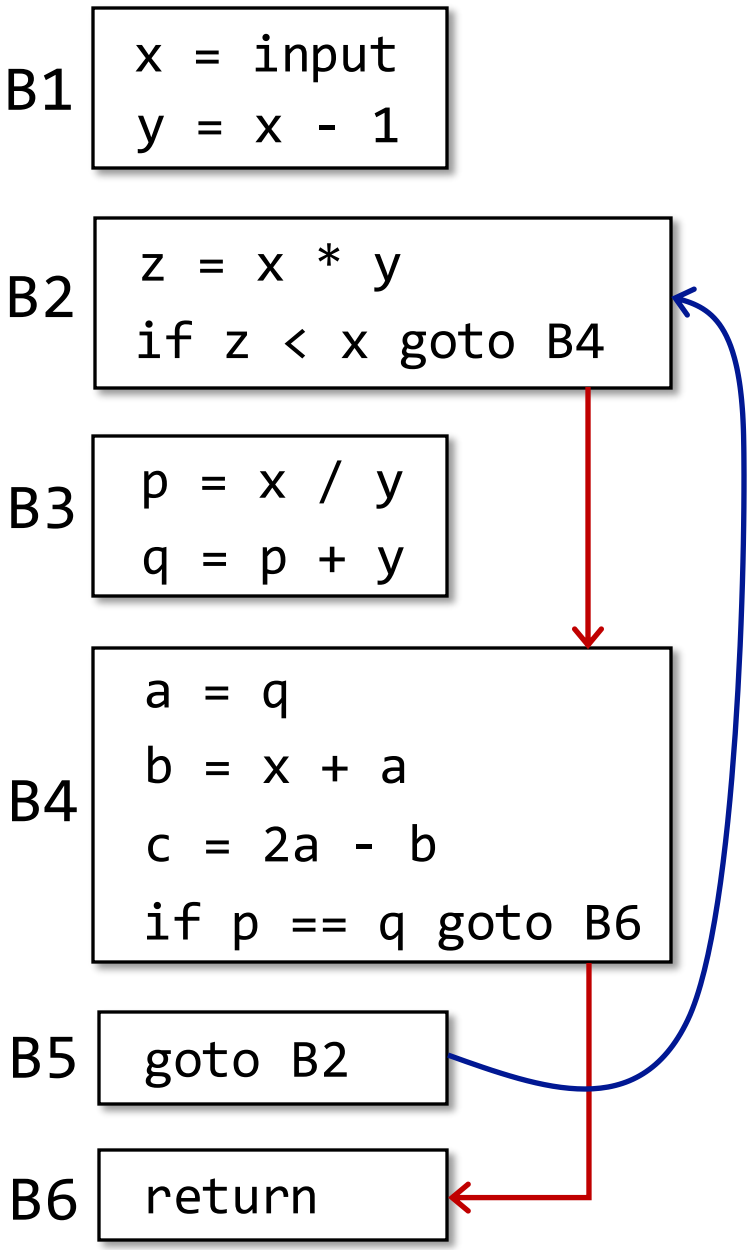
There is a conditional or unconditional jump from the end of **A** to the beginning of **B**



Add edges in CFG

There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

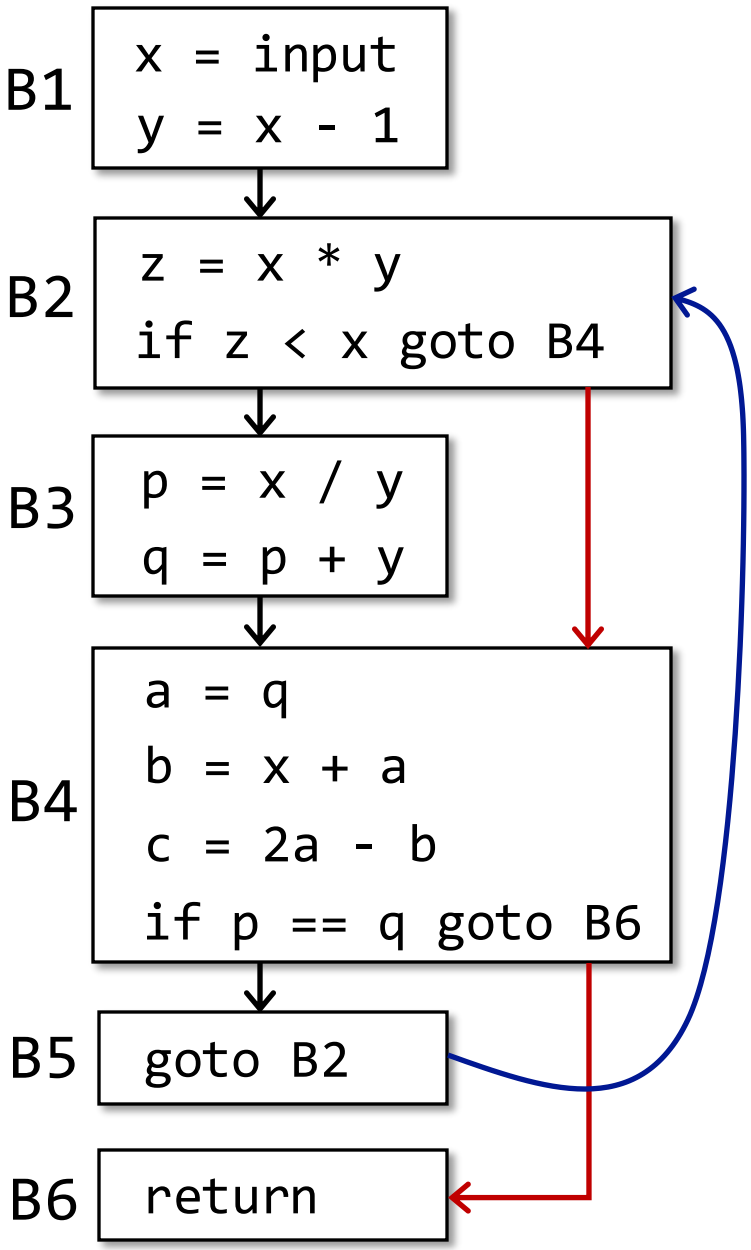




Add edges in CFG

There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

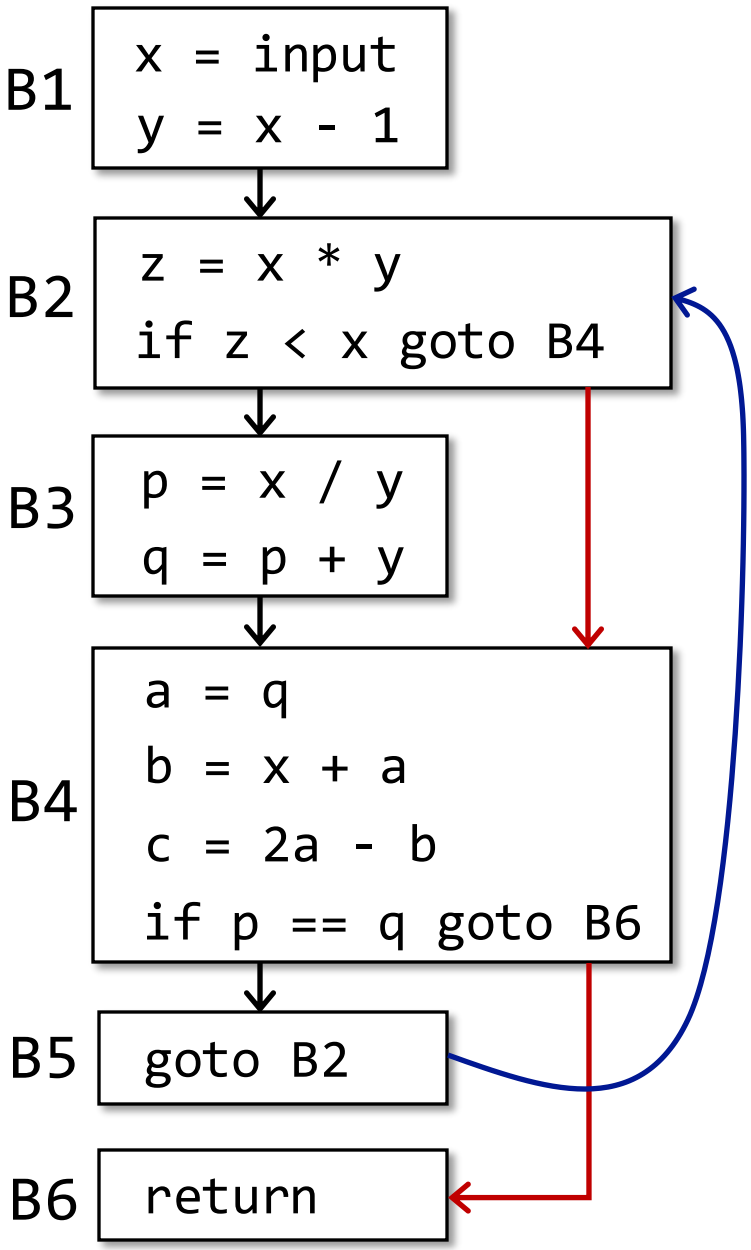
B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump



Add edges in CFG

There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump

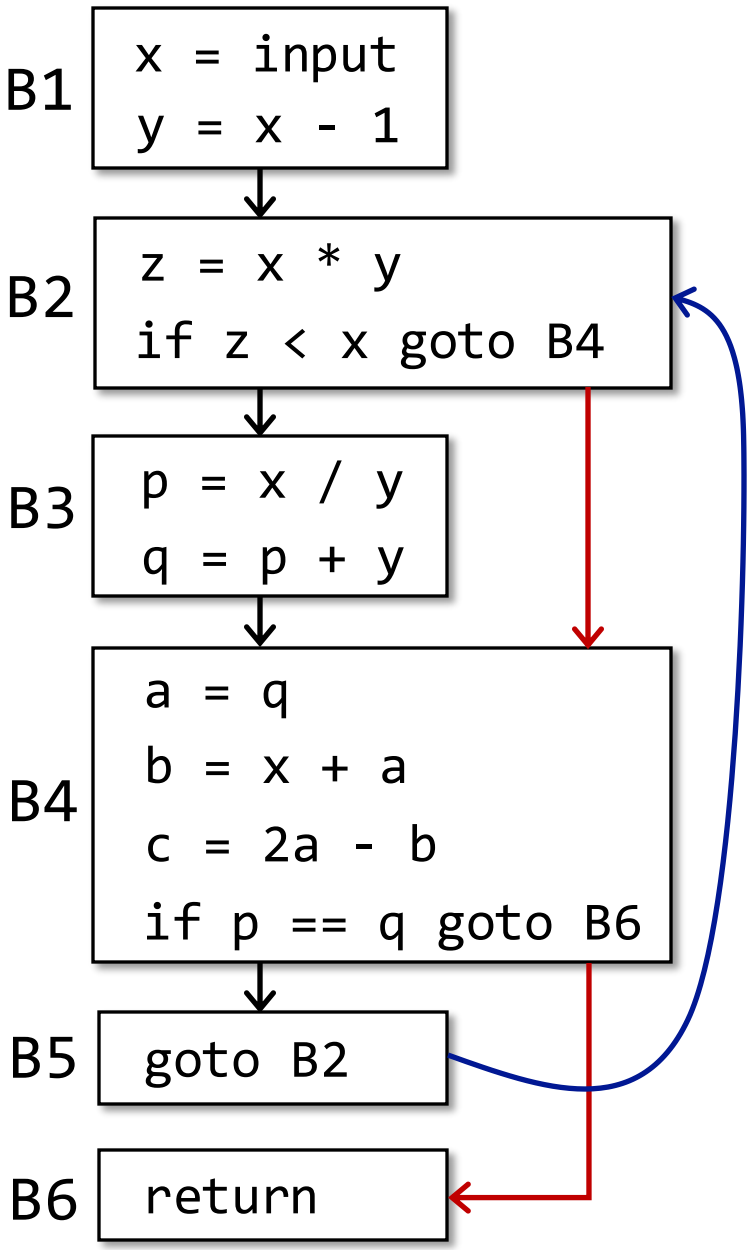


Add edges in CFG

There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump

We say that **A** is a **predecessor** of **B**, and **B** is a **successor** of **A**



Add edges in CFG

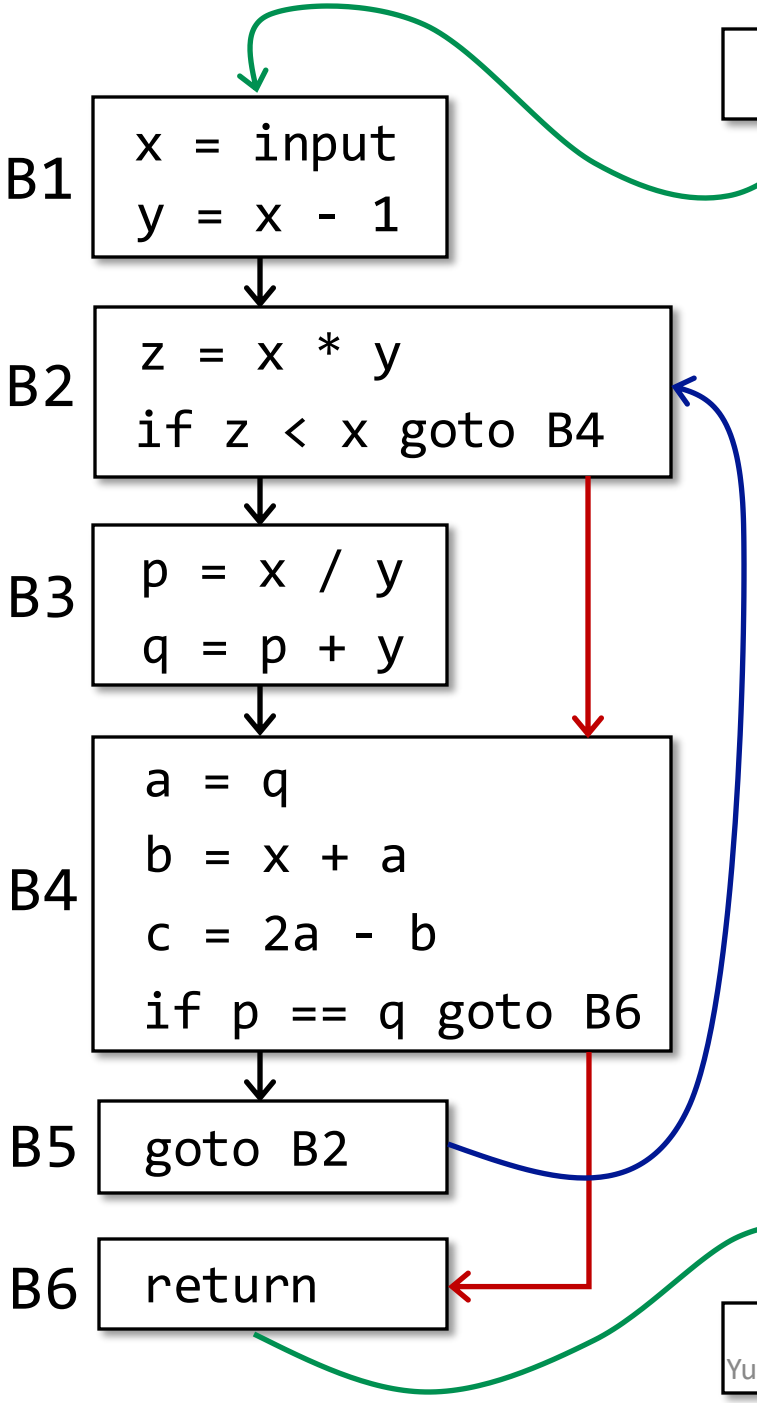
There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump

We say that **A** is a **predecessor** of **B**, and **B** is a **successor** of **A**

Usually we add two nodes, **Entry** and **Exit**.

- They do not correspond to executable IR
- An edge from Entry to the BB containing the first instruction of IR
- An edge to Exit from any BB containing an instruction that could be the last instruction of IR



Add edges in CFG

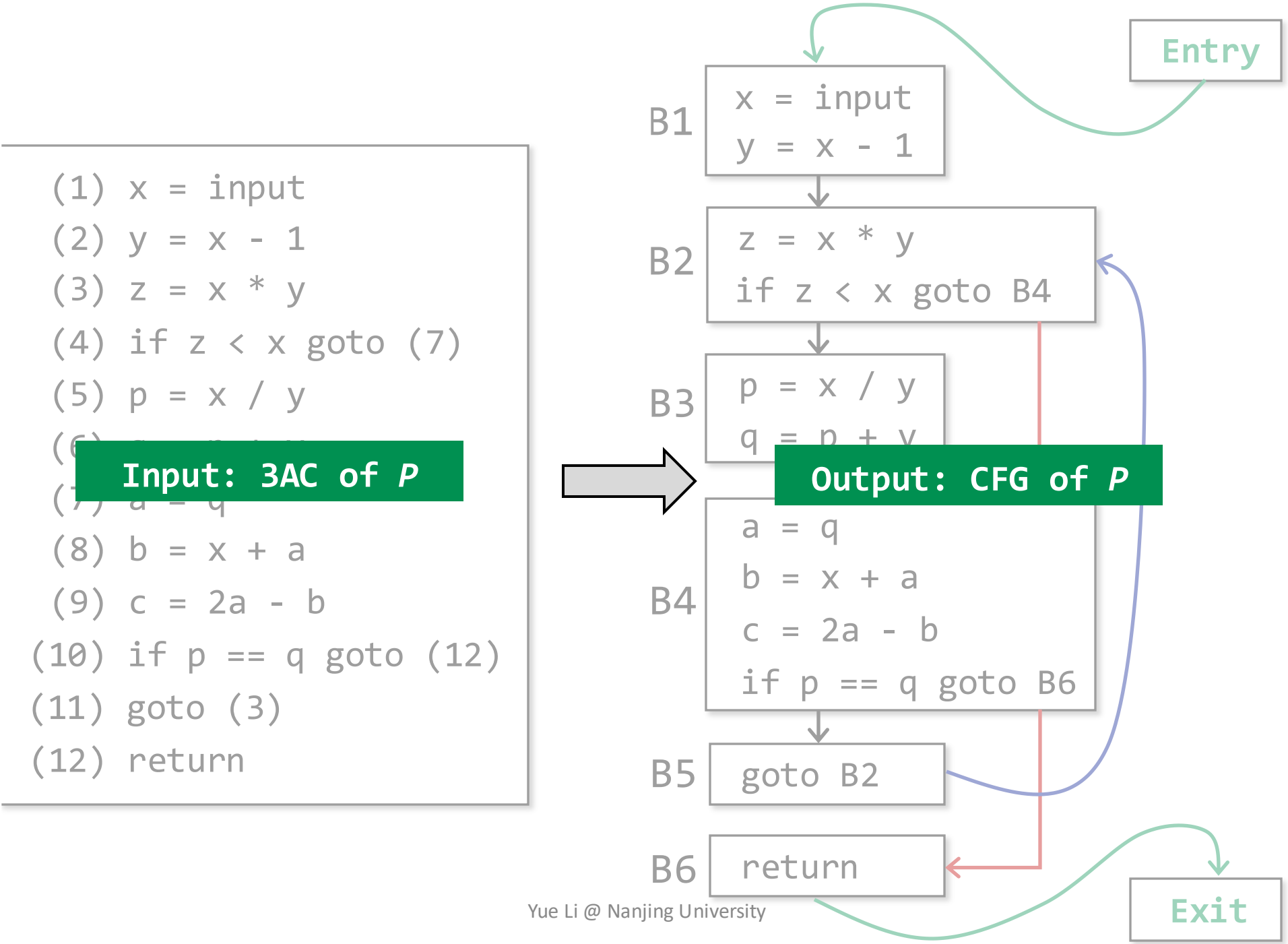
There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump

We say that **A** is a **predecessor** of **B**, and **B** is a **successor** of **A**

Usually we add two nodes, **Entry** and **Exit**.

- They do not correspond to executable IR
- An edge from Entry to the BB containing the first instruction of IR
- An edge to Exit from any BB containing an instruction that could be the last instruction of IR


(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
Input: 3AC of P
Output: CFG of P
Entry
B1
x = input
y = x - 1
B2
z = x * y
if z < x goto B4
B3
p = x / y
B4
a = q
b = x + a
c = 2a - b
if p == q goto B6
B5
goto B2
B6
return
Exit

Recommended Reading