软件分析

南京大学
计算机科学与技术系
程序设计语言与
静态分析研究组
李樾
谭添

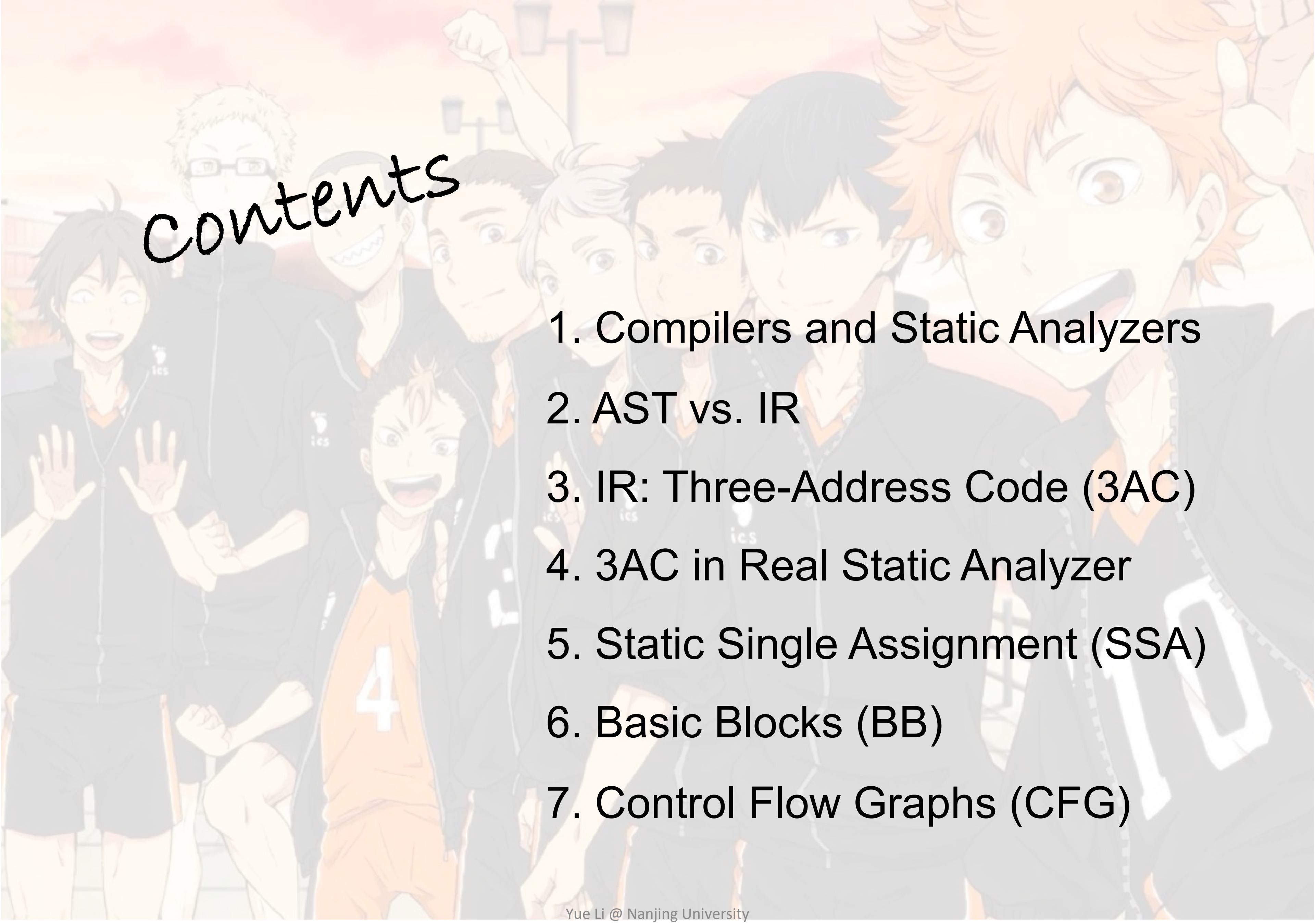
Static Program Analysis

Intermediate Representation

Nanjing University

Yue Li

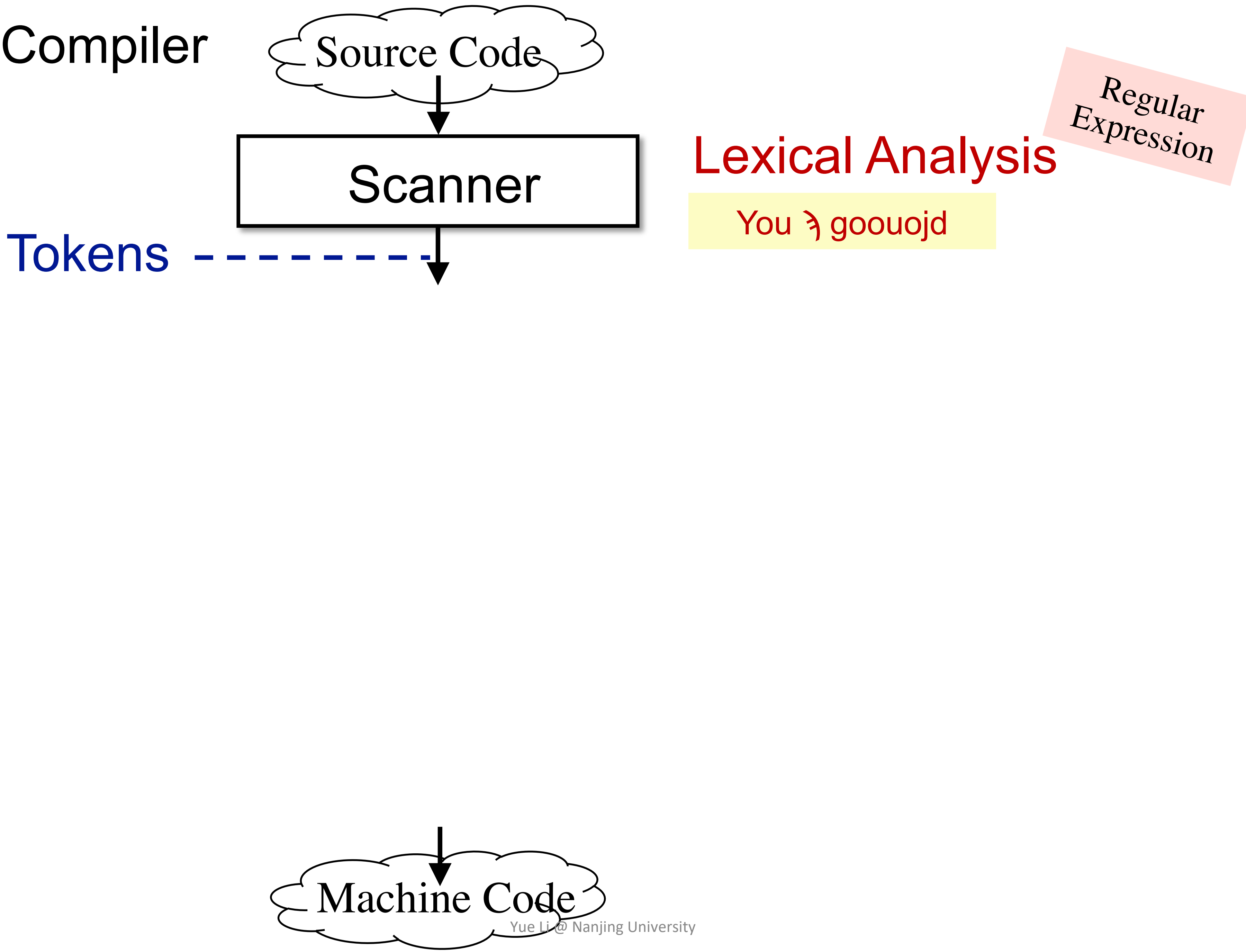
Fall 2021



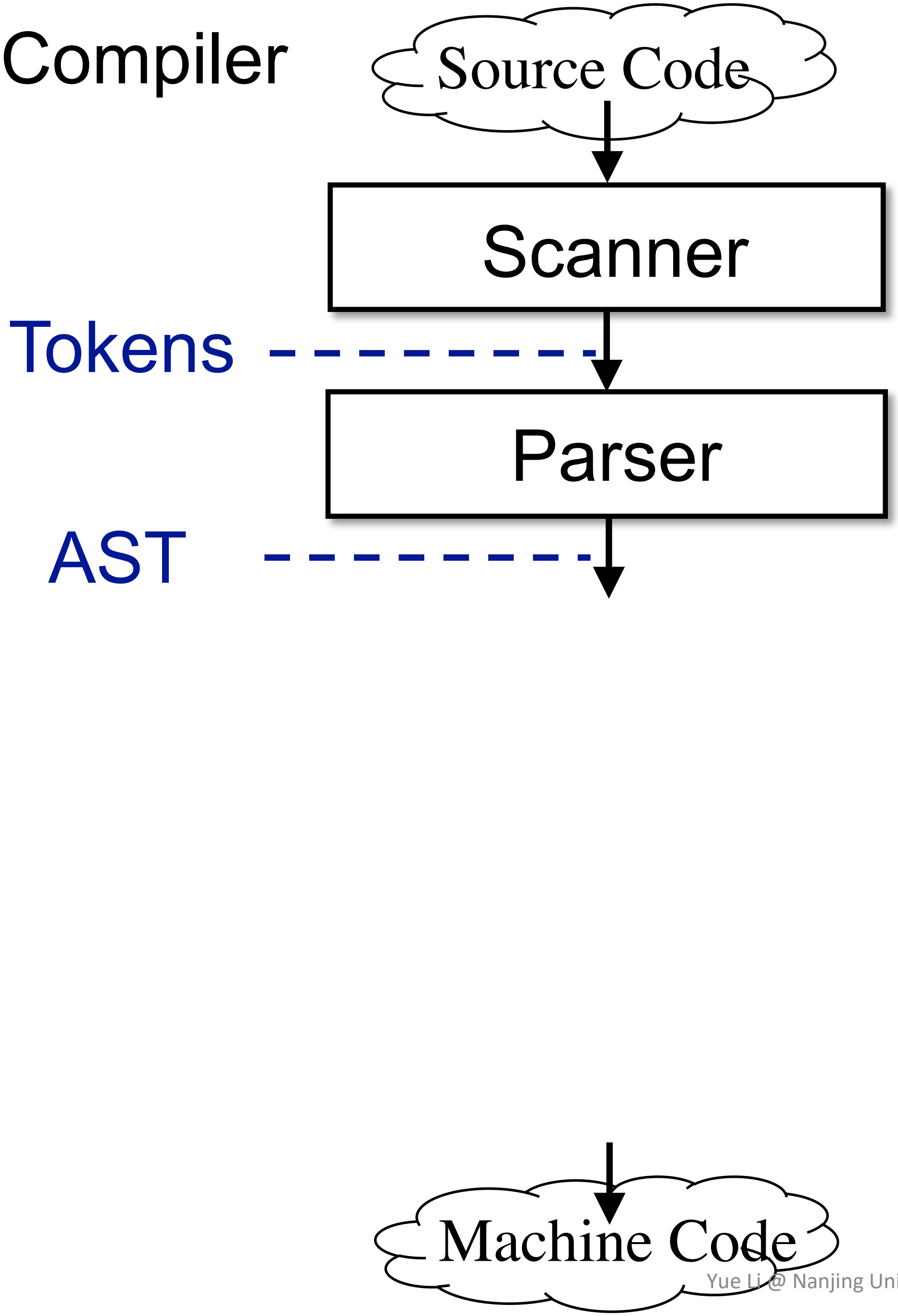
Contents

Compiler

Compiler


Source Code
Scanner
Tokens
Lexical Analysis
You goouojd
Regular Expression
Machine Code

Compiler



Lexical Analysis

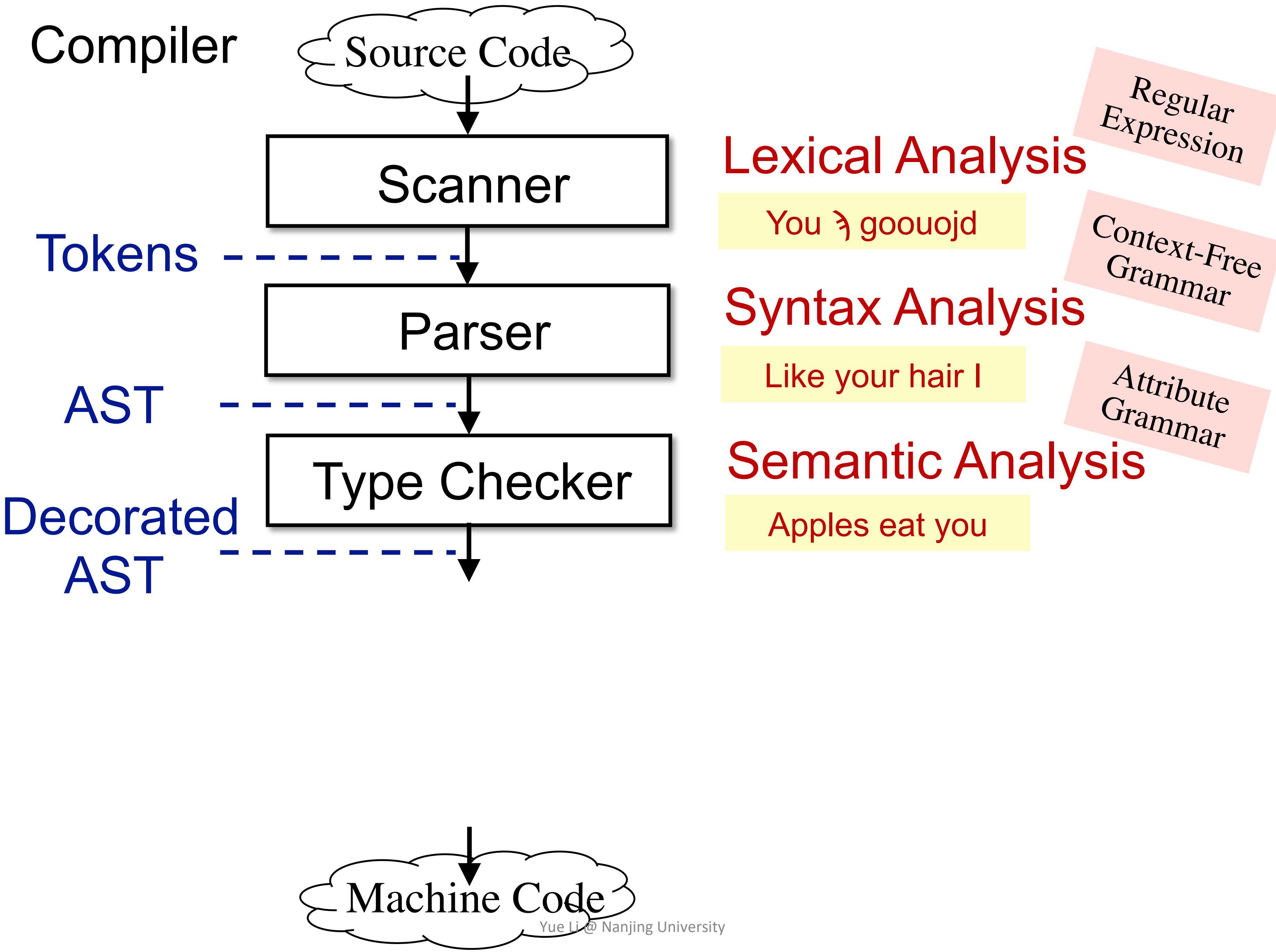
You goouojd

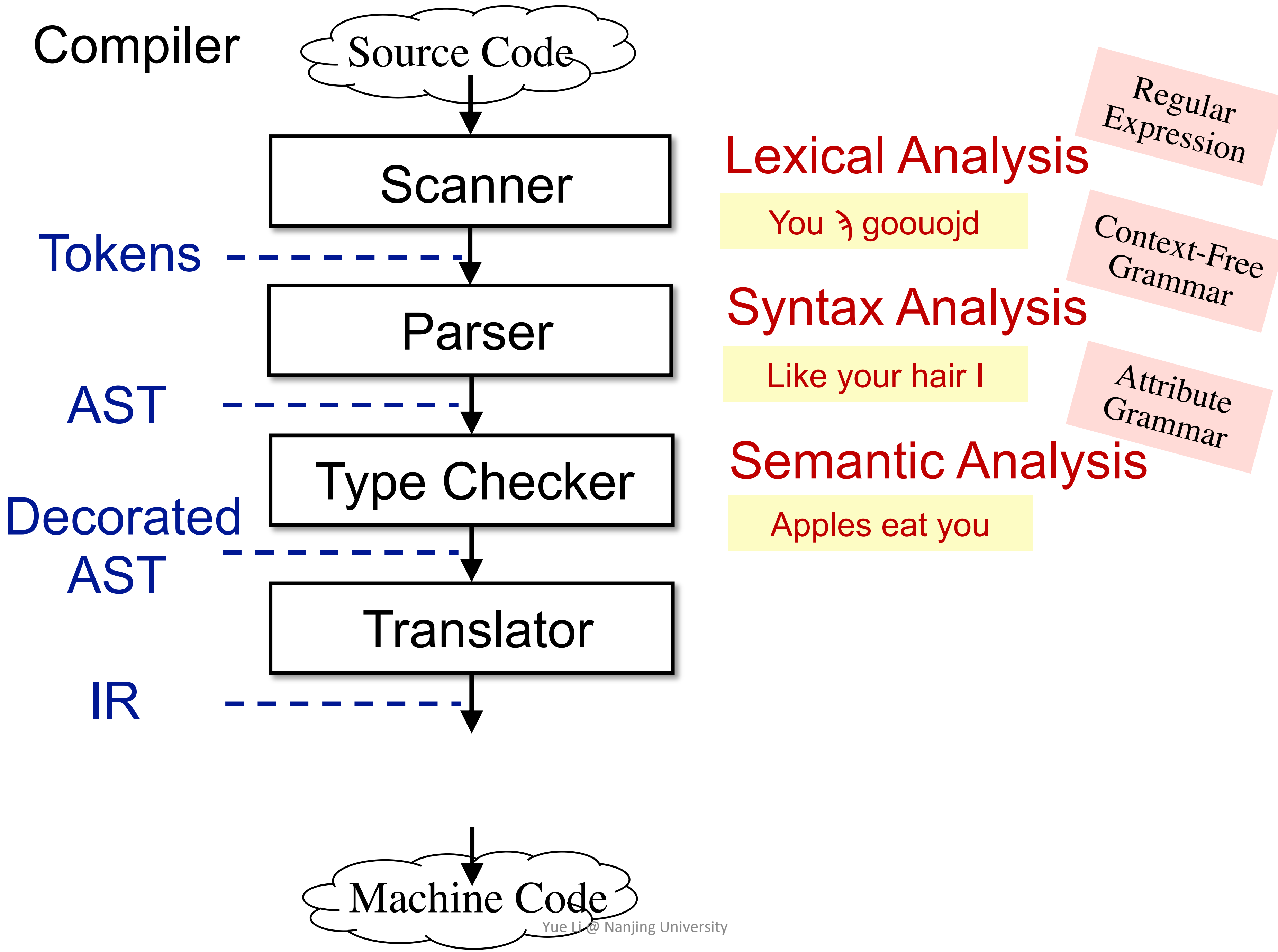
Syntax Analysis

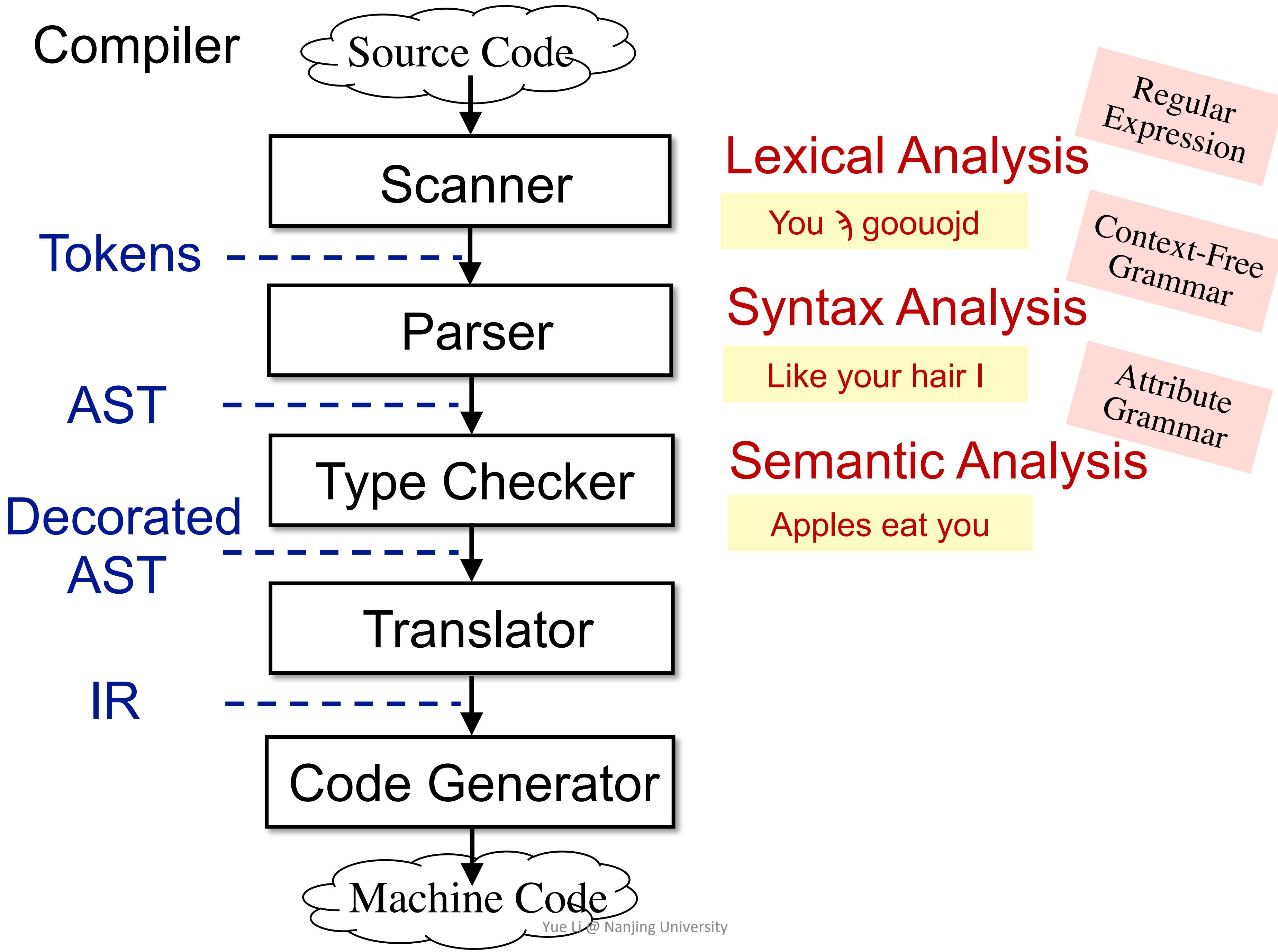
Like your hair I

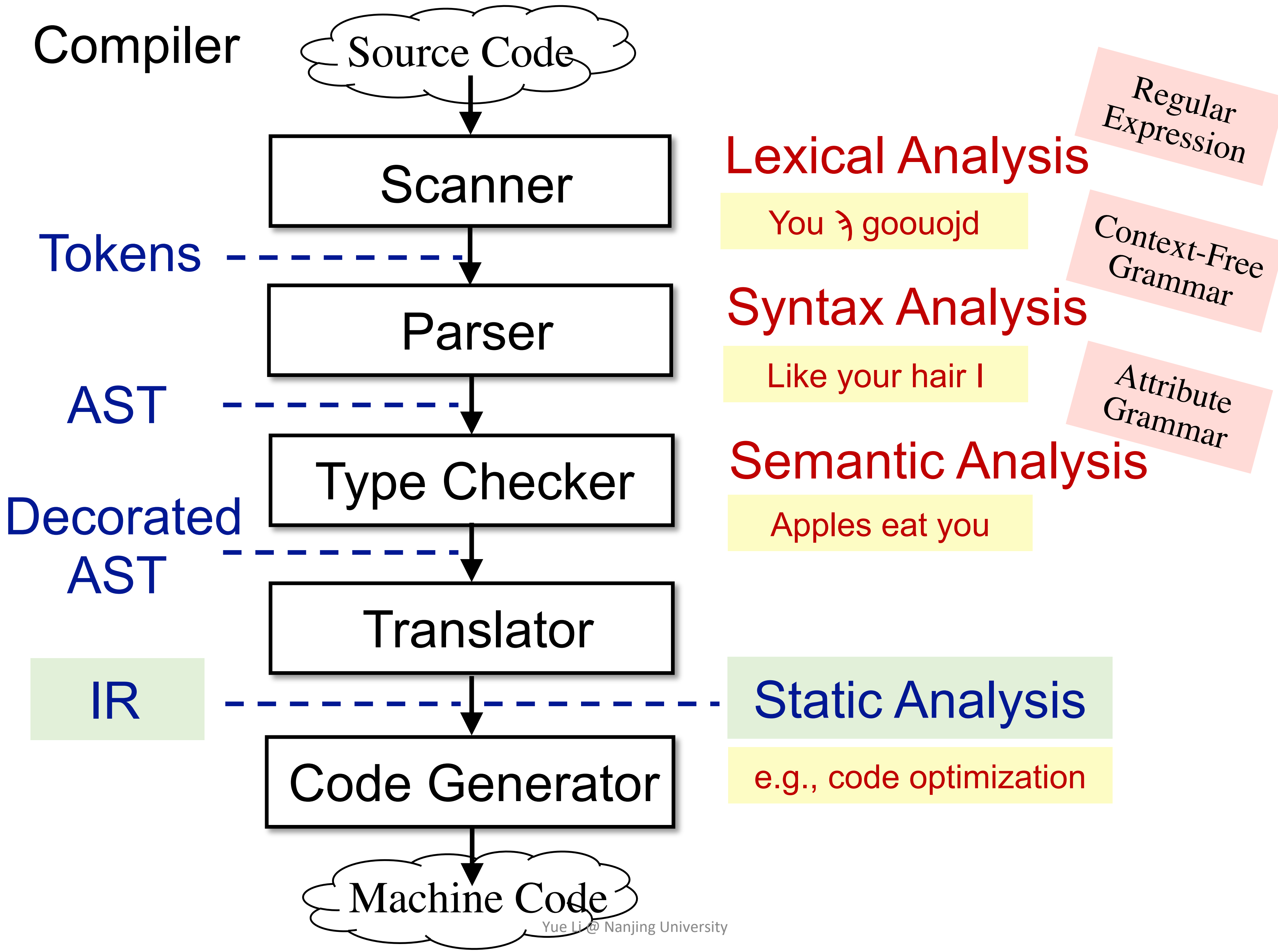
Regular Expression

Context-Free Grammar


Compiler
Source Code
Scanner
Tokens
Parser
AST
Type Checker
Decorated AST
Machine Code
Lexical Analysis
You goouojd
Syntax Analysis
Like your hair I
Semantic Analysis
Apples eat you
Regular Expression
Context-Free Grammar
Attribute Grammar

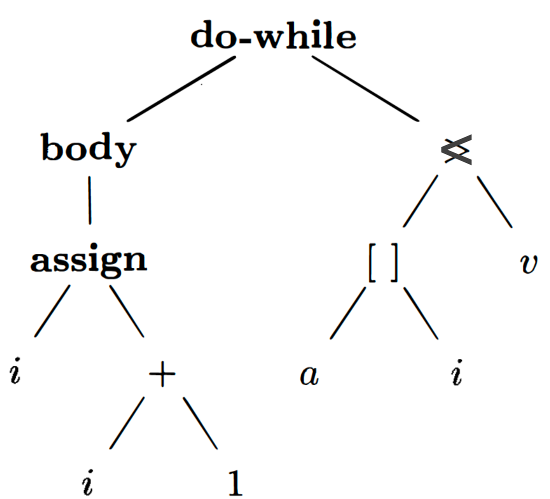

Compiler
Source Code
Scanner
Tokens
Parser
AST
Type Checker
Decorated AST
Translator
IR
Machine Code
Lexical Analysis
You goouojd
Syntax Analysis
Like your hair I
Semantic Analysis
Apples eat you
Regular Expression
Context-Free Grammar
Attribute Grammar
Yue Li @ Nanjing University


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


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

AST vs. IR

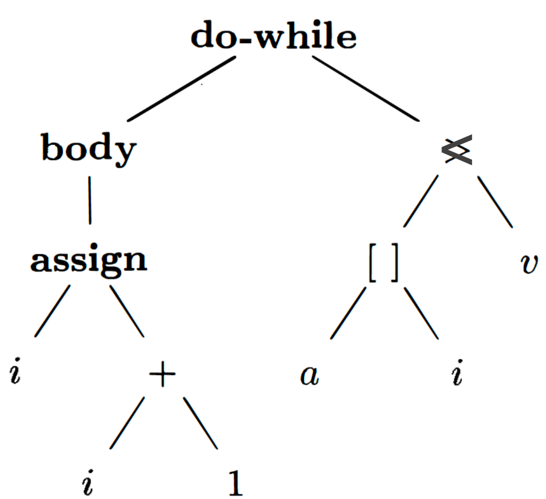
AST




```
"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

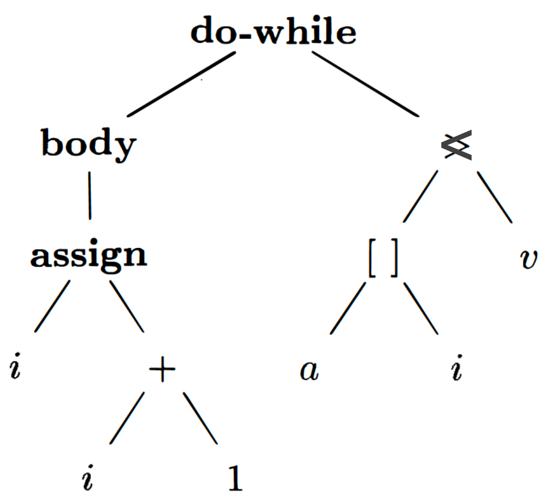
- high-level and closed to grammar structure
- usually language dependent
- suitable for fast type checking
- lack of control flow information

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

`“do i=i+1; while(a[i]<v);”`

AST



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.

t2 = a + b + 3 ⇨ t1 = a + b
t2 = t1 + 3

Intermediate Representation (IR)

- 3-Address Code (3AC)

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

t2 = a + b + 3 → t1 = a + b
t2 = t1 + 3

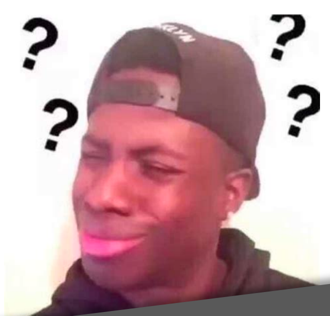
Intermediate Representation (IR)

- 3-Address Code (3AC)

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

t2 = a + b + 3 ⇨ t1 = a + b
t2 = t1 + 3

Each 3AC contains at most 3 addresses



Why called 3-address?

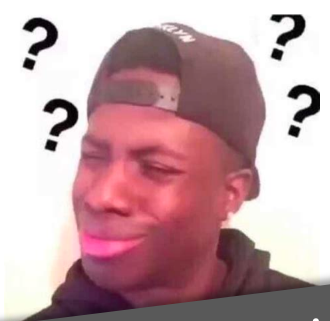
Intermediate Representation (IR)

- 3-Address Code (3AC)

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

t2 = a + b + 3 ⇨ t1 = a + b
t2 = t1 + 3

Each 3AC contains at most 3 addresses



Why called 3-address?

Address can be one of the following:

- Name: a, b
- Constant: 3
- Compiler-generated temporary: t1, t2

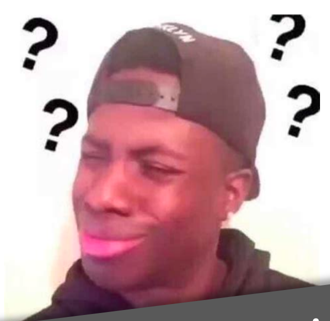
Intermediate Representation (IR)

- 3-Address Code (3AC)

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

t2 = a + b + 3 ⇨ t1 = a + b
t2 = t1 + 3

Each 3AC contains at most 3 addresses



Why called 3-address?

Address can be one of the following:

- Name: a, b
- Constant: 3
- Compiler-generated temporary: t1, t2

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

Do-While Loop

```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

Method Call

Method Call

```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)

Method Call

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");
    }
}
```

3AC(jimple)

```
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;
}
```

Class

```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

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

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

    }
}
```

Java Src

Static Single Assignment (SSA)

Optional material

- All assignments in SSA are to variables with distinct names
  - Give each definition a fresh name
  - Propagate fresh name to subsequent uses
  - Every variable has exactly one definition

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

3AC

$$
\begin{aligned}
p_1 &= a + b \\
q_1 &= p_1 - c \\
p_2 &= q_1 * d \\
p_3 &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}
$$

SSA

Static Single Assignment (SSA)

- All assignments in SSA are to variables with distinct names
  - Give each definition a fresh name
  - Propagate fresh name to subsequent uses
  - Every variable has exactly one definition


```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

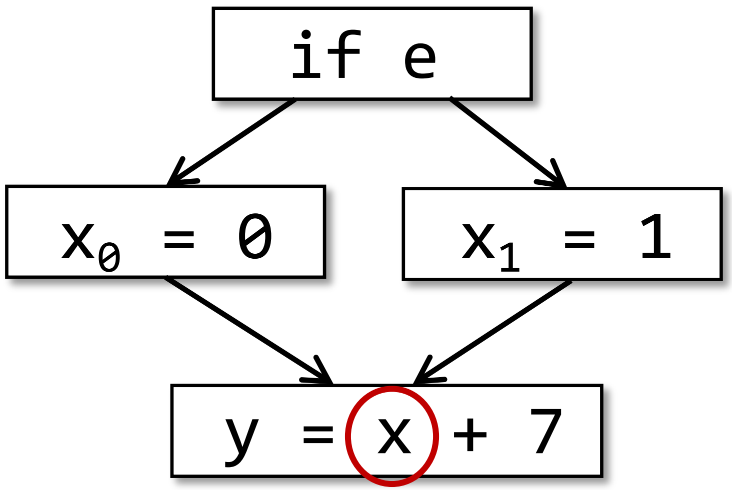
3AC

$$
\begin{aligned}
p_1 &= a + b \\
q_1 &= p_1 - c \\
p_2 &= q_1 * d \\
p_3 &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}
$$

SSA

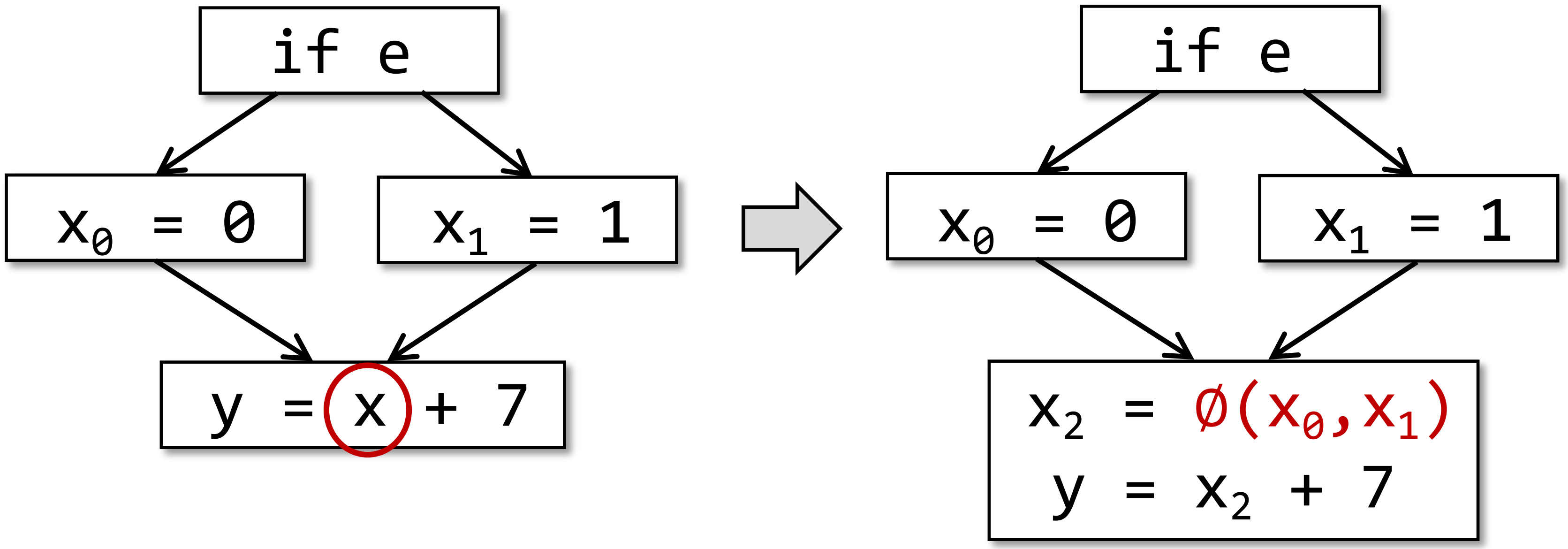
Static Single Assignment (SSA)

- What if a variable use is at control flow merges?



Static Single Assignment (SSA)

- What if a variable use is at control flow merges?



- A special merge operator, Ø (called phi-function), is introduced to select the values at merge nodes
- $Ø(x_0,x_1)$ has the value x_0 if the control flow passes through the true part of the conditional and the value x_1 otherwise

Why SSA?

Why not SSA?

Why SSA?

- Flow information is indirectly incorporated into the unique variable names

May help deliver some simpler analyses, e.g., flow-insensitive analysis gains partial precision of flow-sensitive analysis via SSA

- Define-and-Use pairs are explicit

Enable more effective data facts storage and propagation in some on-demand tasks

Some optimization tasks perform better on SSA (e.g., conditional constant propagation, global value numbering)

Why not SSA?

Why SSA?

- Flow information is indirectly incorporated into the unique variable names

May help deliver some simpler analyses, e.g., flow-insensitive analysis gains partial precision of flow-sensitive analysis via SSA

- Define-and-Use pairs are explicit

Enable more effective data facts storage and propagation in some on-demand tasks

Some optimization tasks perform better on SSA (e.g., conditional constant propagation, global value numbering)

Why not SSA?

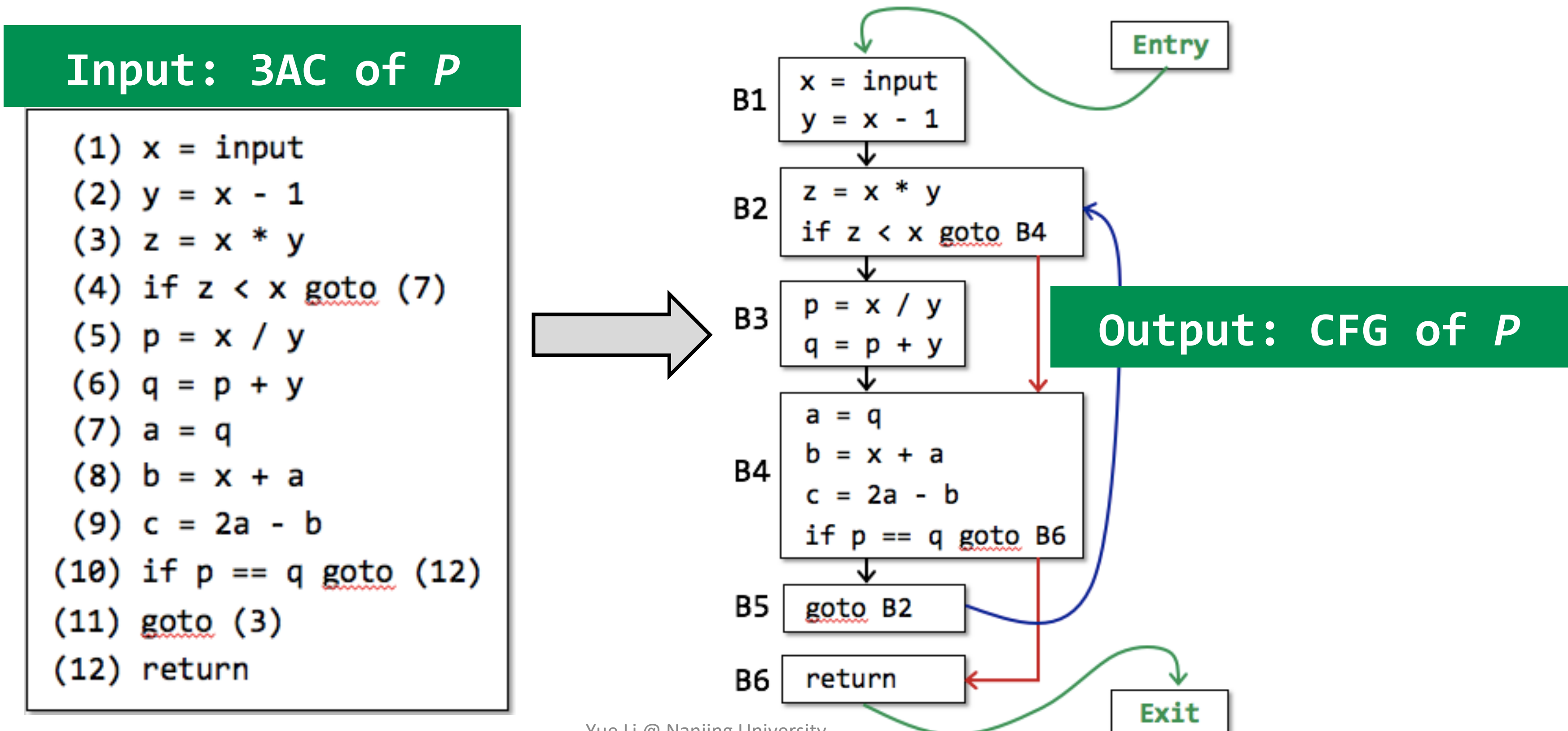
- SSA may introduce too many variables and phi-functions
- May introduce inefficiency problem when translating to machine code (due to copy operations)

Control Flow Analysis

- Usually refer to building Control Flow Graph (CFG)

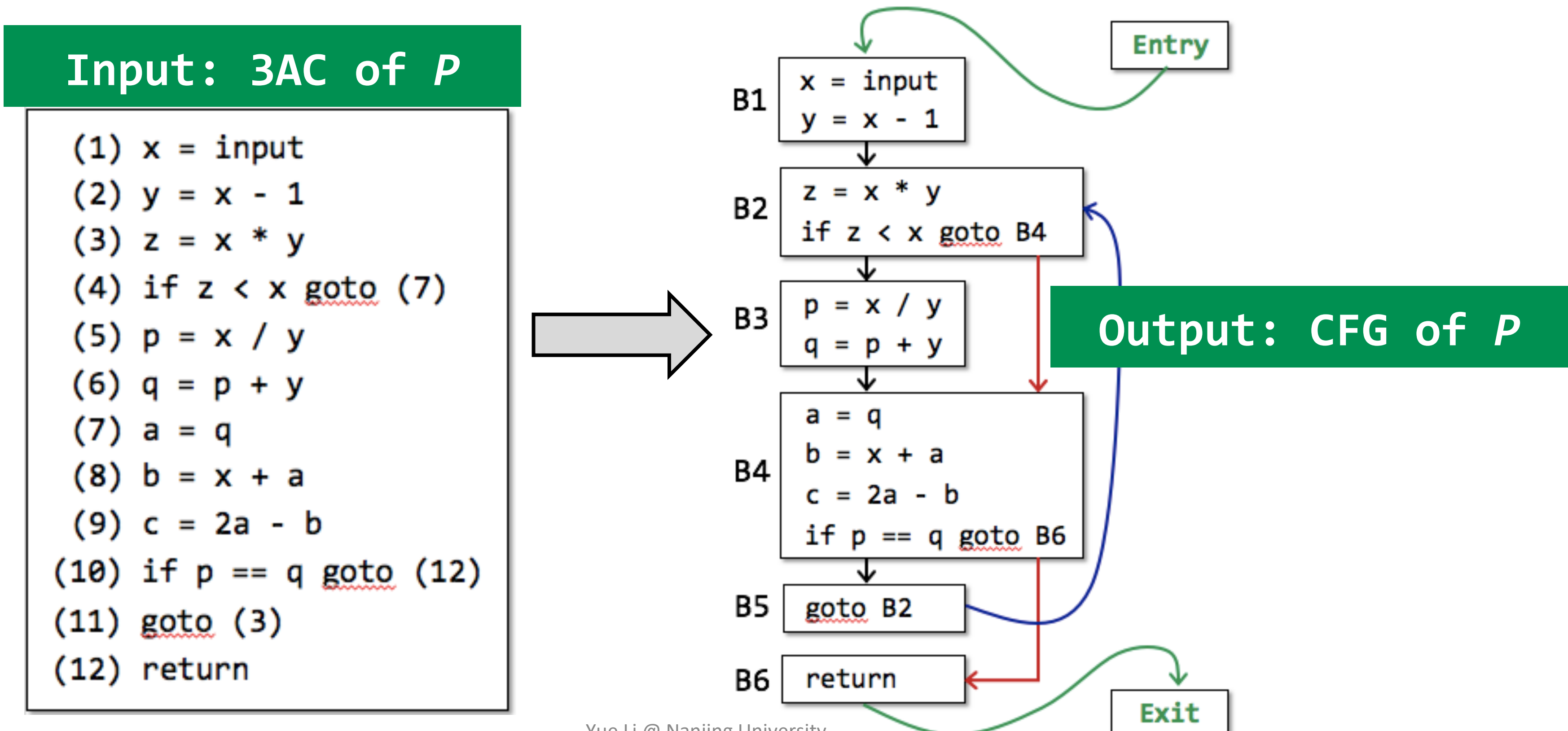
Control Flow Analysis

- Usually refer to building Control Flow Graph (CFG)



Control Flow Analysis

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

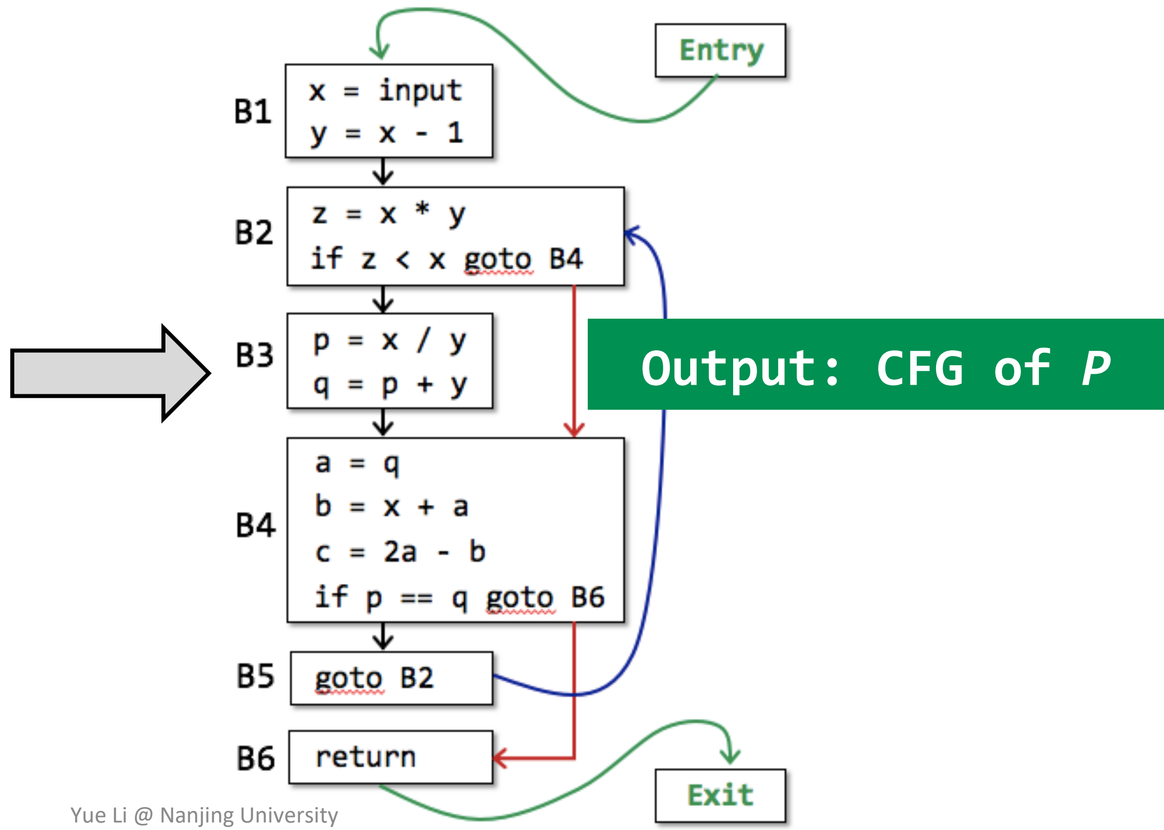


Control Flow Analysis

- 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)

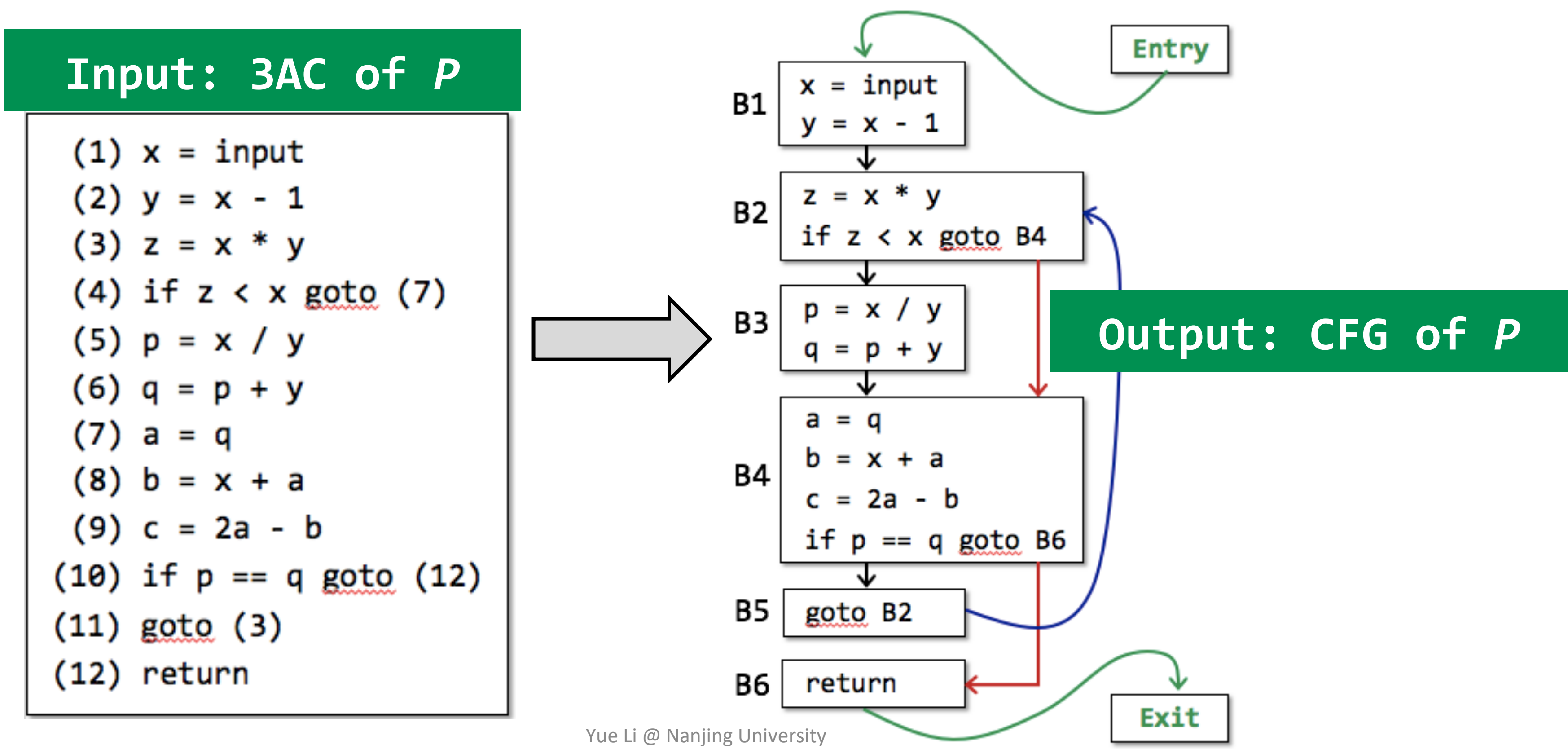
Input: 3AC of *P*

```
(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
```



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)



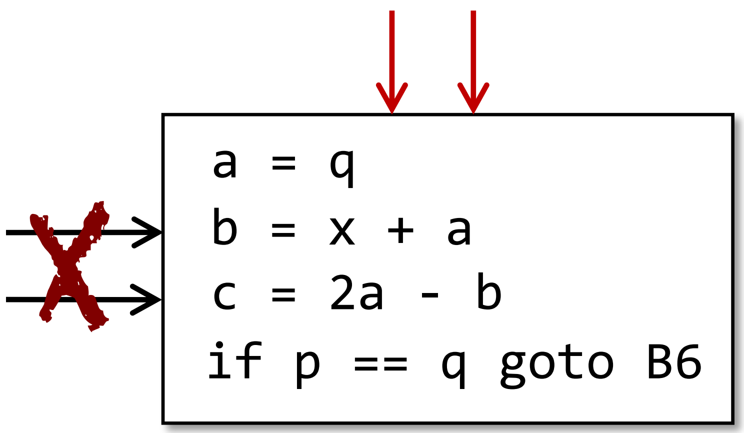
Basic Blocks (BB)

- 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
```

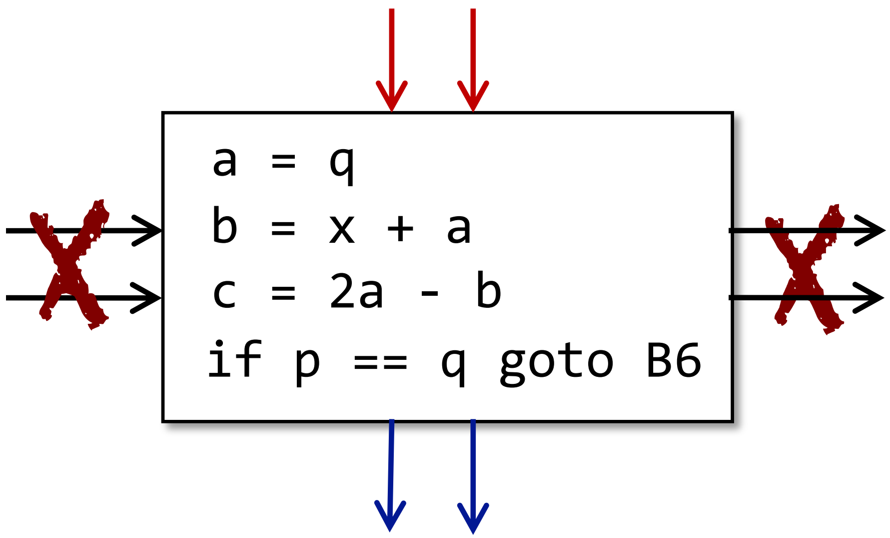
Basic Blocks (BB)

- 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*



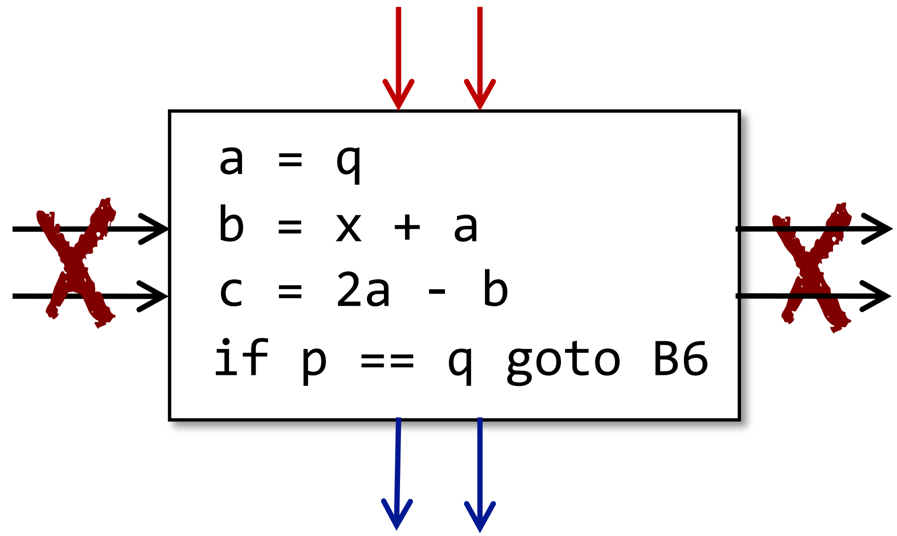
Basic Blocks (BB)

- 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*



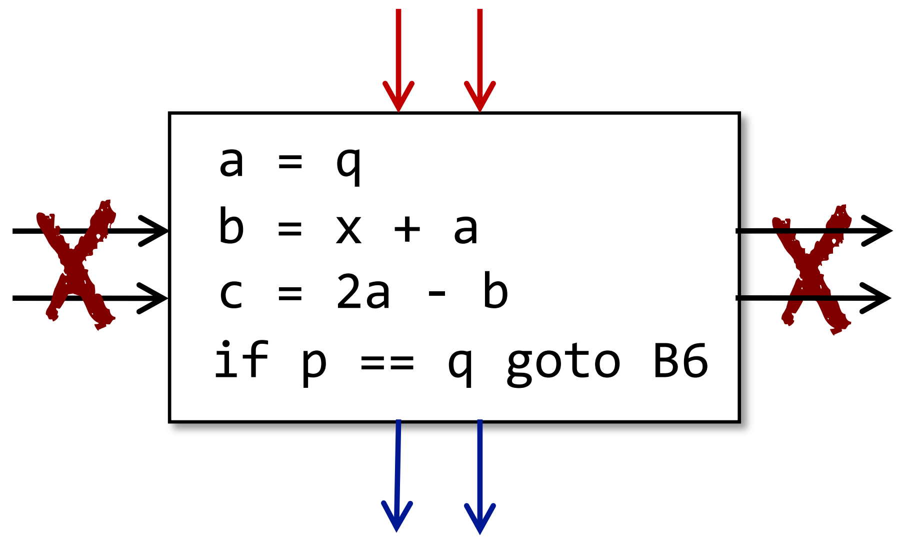
```
 (1) x = input
 (2) y = x - 1
 (3) z = x * y
 (4) if z < x goto (7)
 (5) p = x / y
 (6) q = p + y
 (7) a = q
 (8) b = x + a
 (9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

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



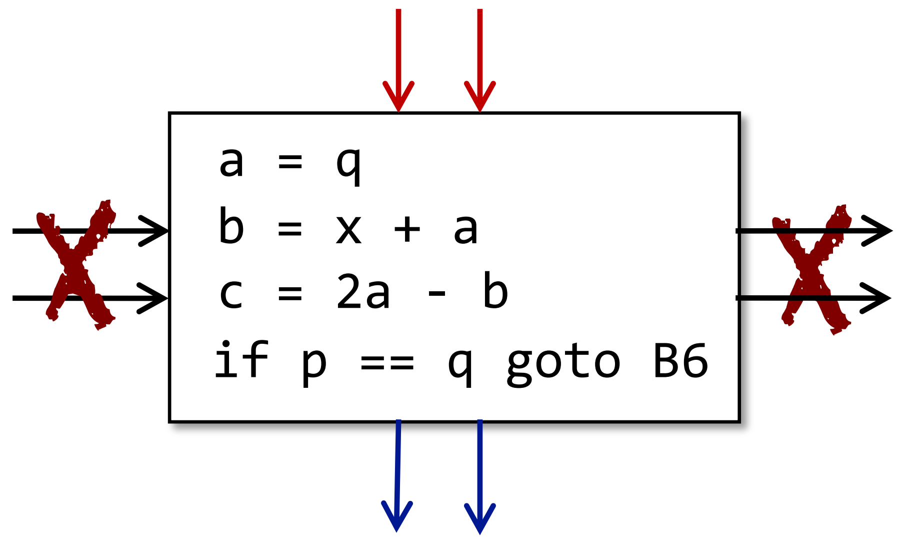
```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

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



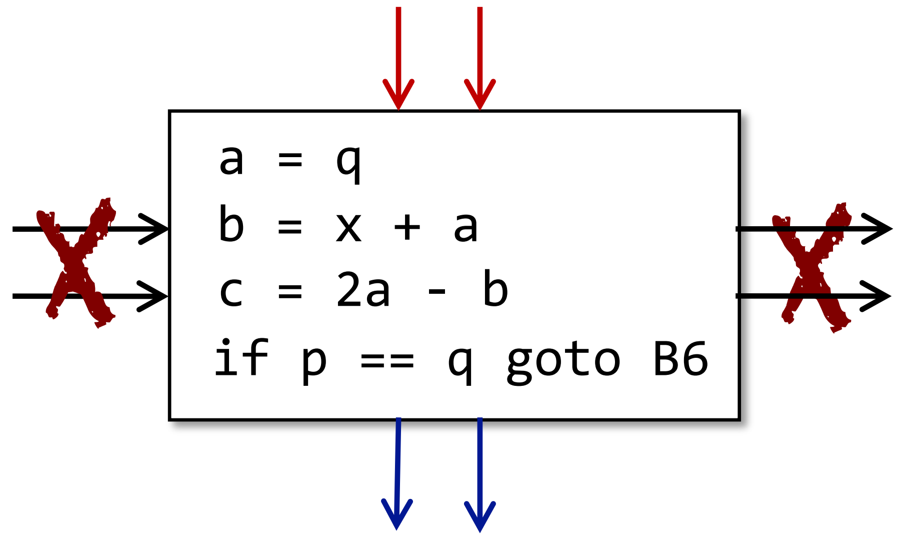
```
(1)  x = input
(2)  y = x - 1
(3)  z = x * y
(4)  if z < x goto (7)
(5)  p = x / y
(6)  q = p + y
(7)  a = q
(8)  b = x + a
(9)  c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

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



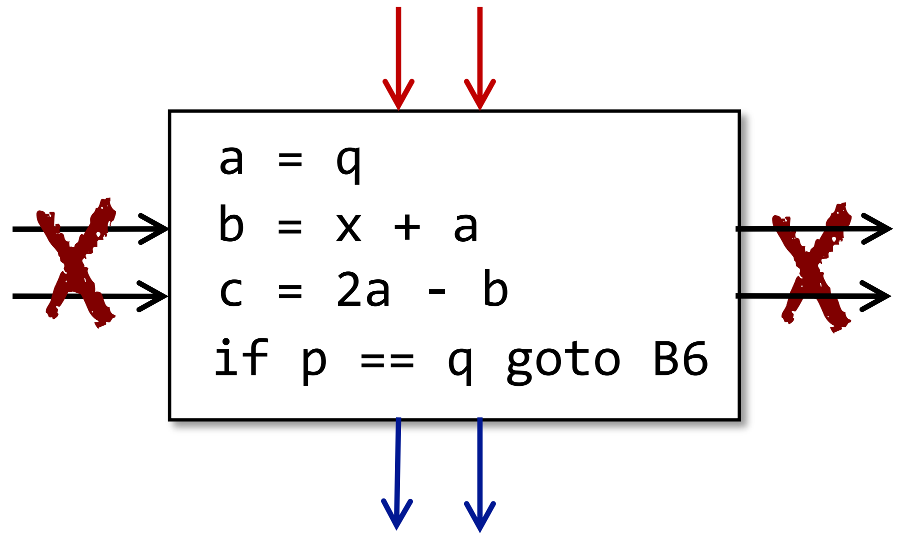
```
(1) x = input
(2) y = x - 1
(3) z = x * y
(4) if z < x goto (7)
(5) p = x / y
(6) q = p + y
(7) a = q
(8) b = x + a
(9) c = 2a - b
(10) if p == q goto (12)
(11) goto (3)
(12) return
```

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



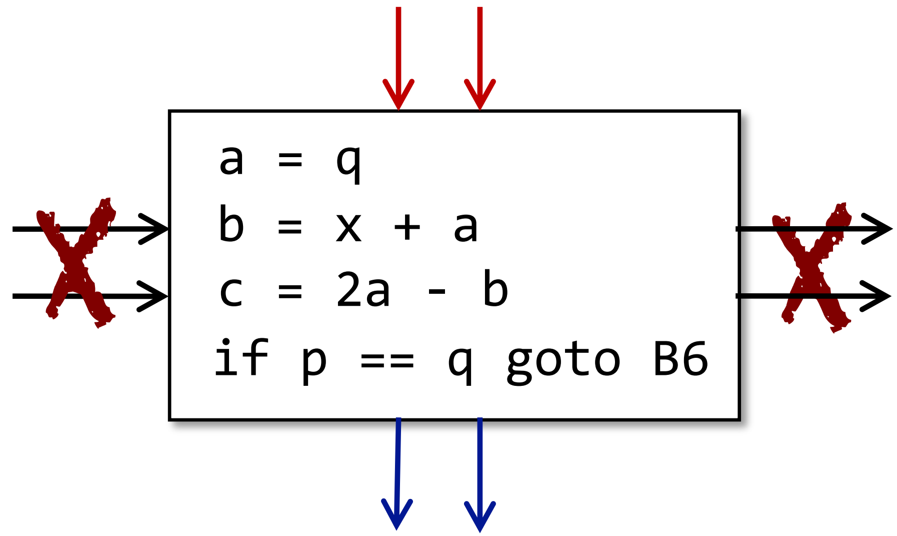
Now try to design the algorithm to build BBs by yourself!

```
(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
```



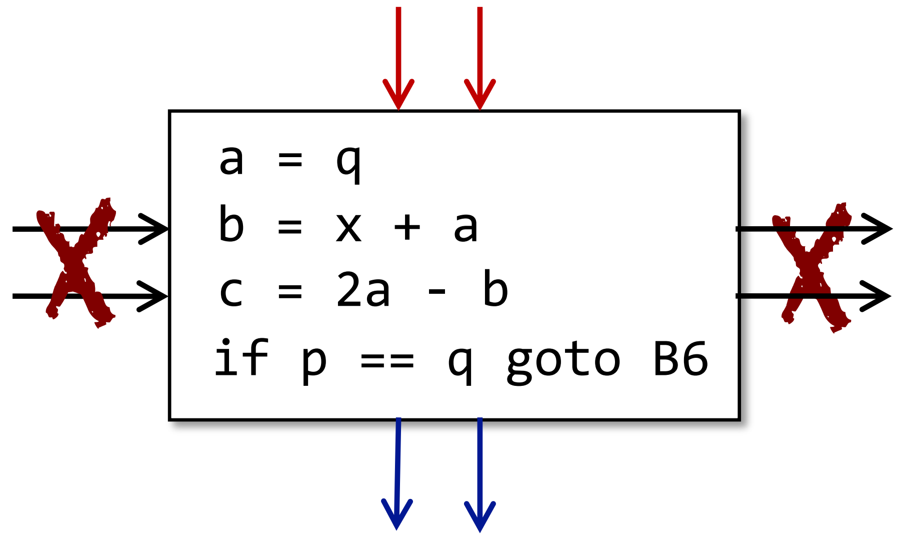
Now try to design the algorithm to build BBs by yourself!

```
(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
```



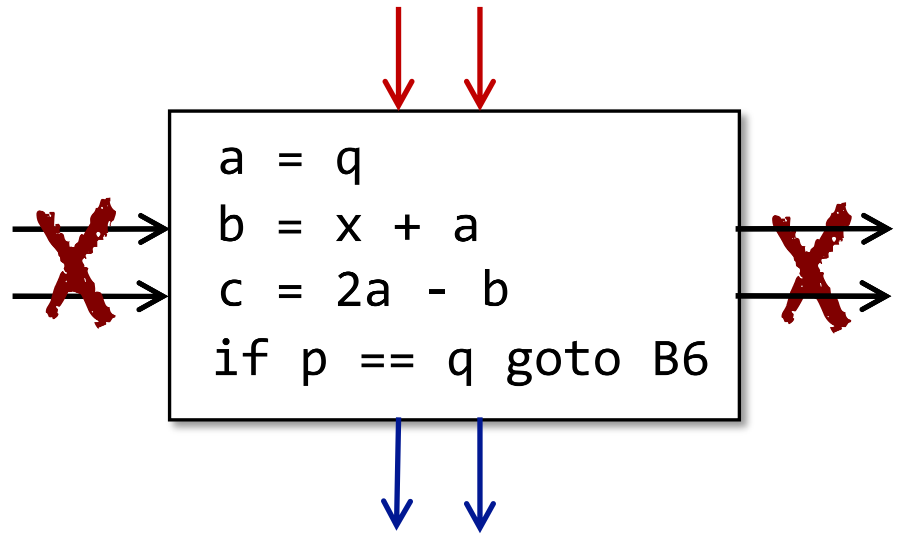
Now try to design the algorithm to build BBs by yourself!

```
(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
```



```
(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
```

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
```

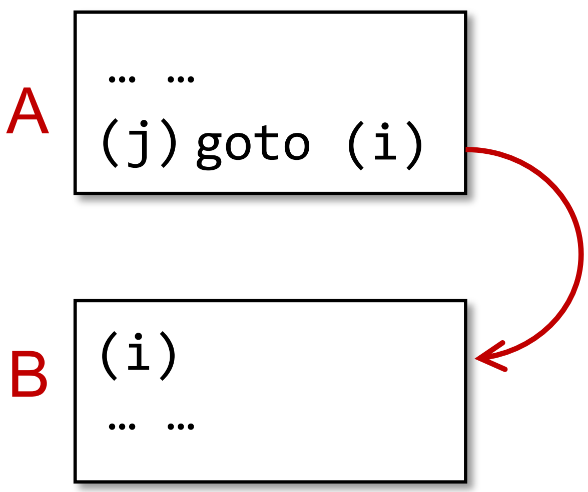
How to build CFG on top of BBs?

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

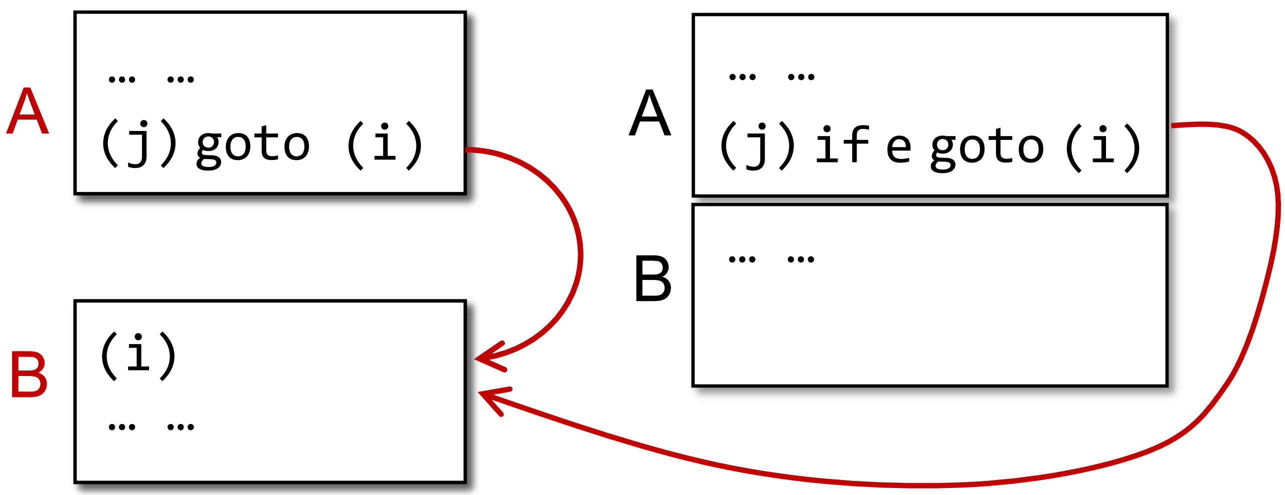
Control Flow Graph (CFG)

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



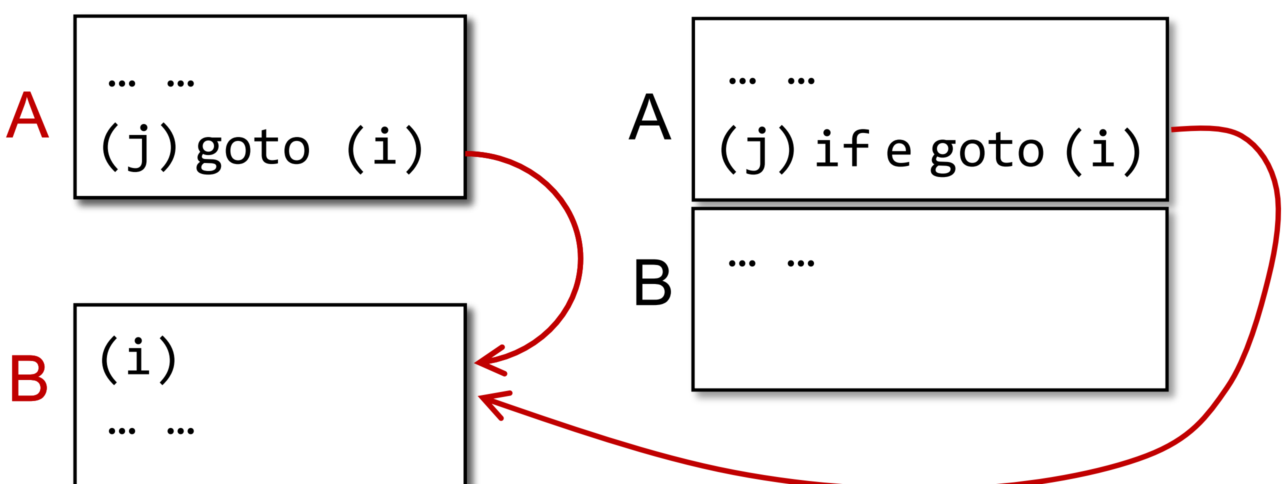
Control Flow Graph (CFG)

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



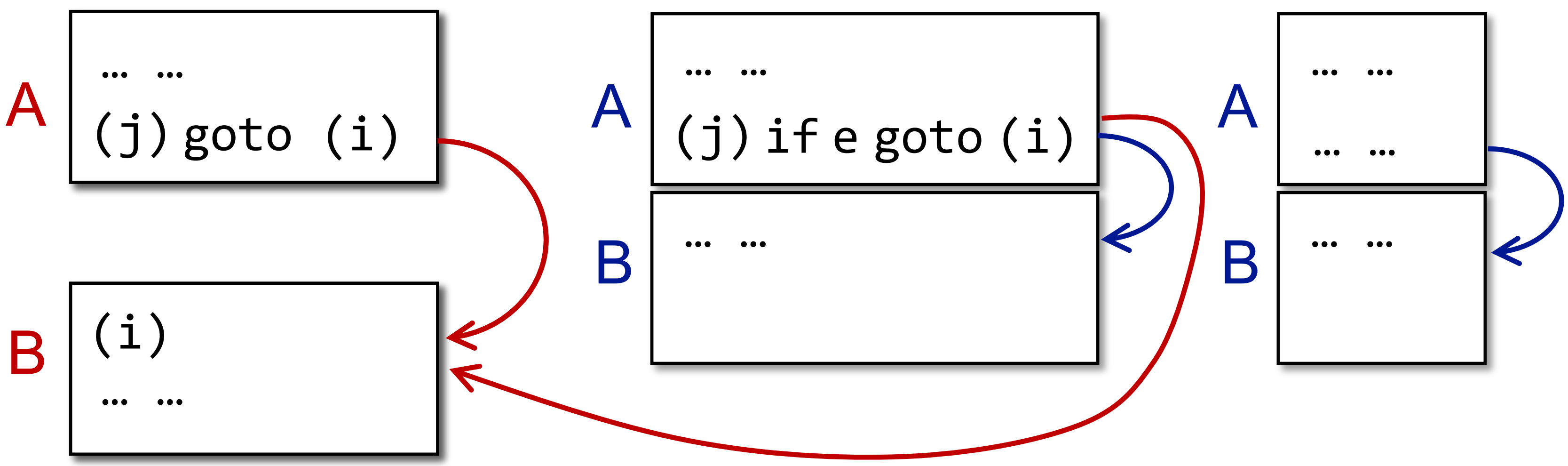
Control Flow Graph (CFG)

- 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



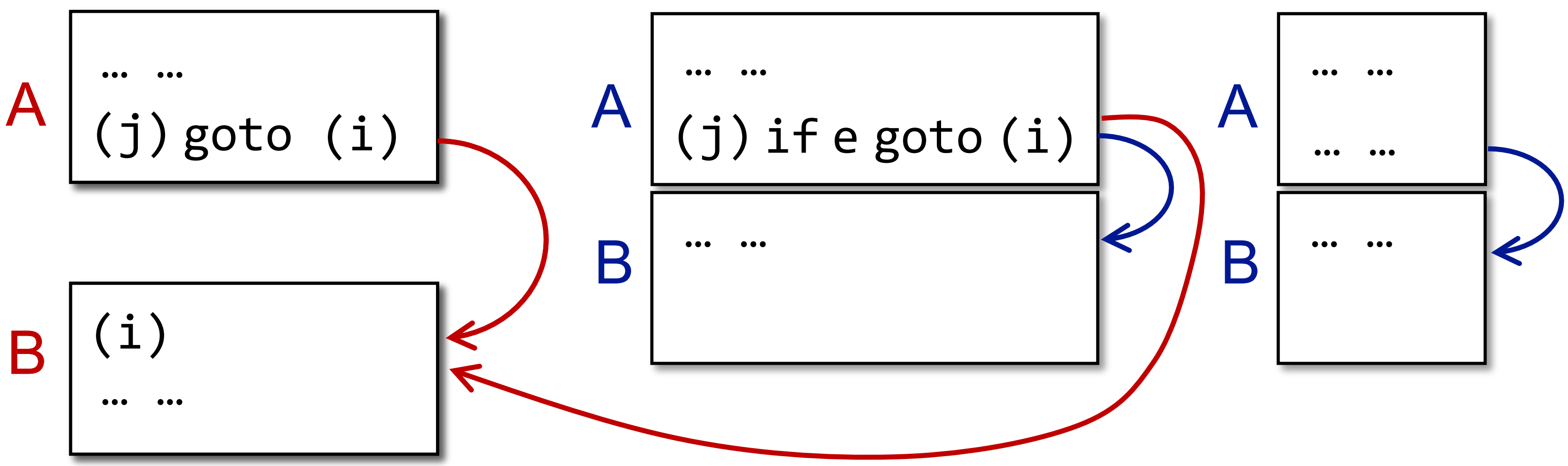
Control Flow Graph (CFG)

- 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



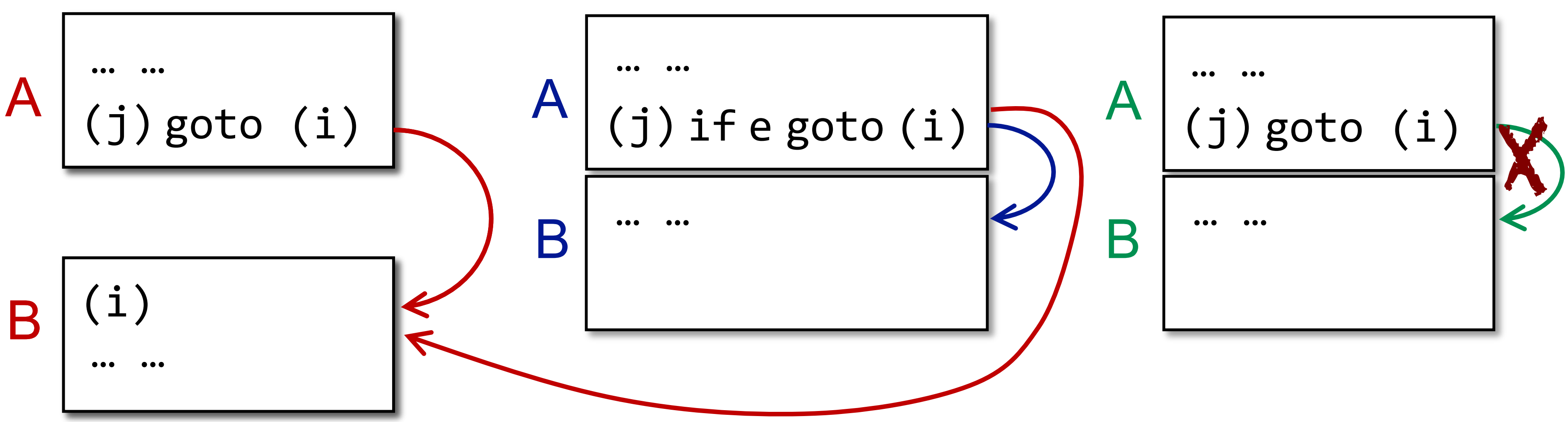
Control Flow Graph (CFG)

- 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



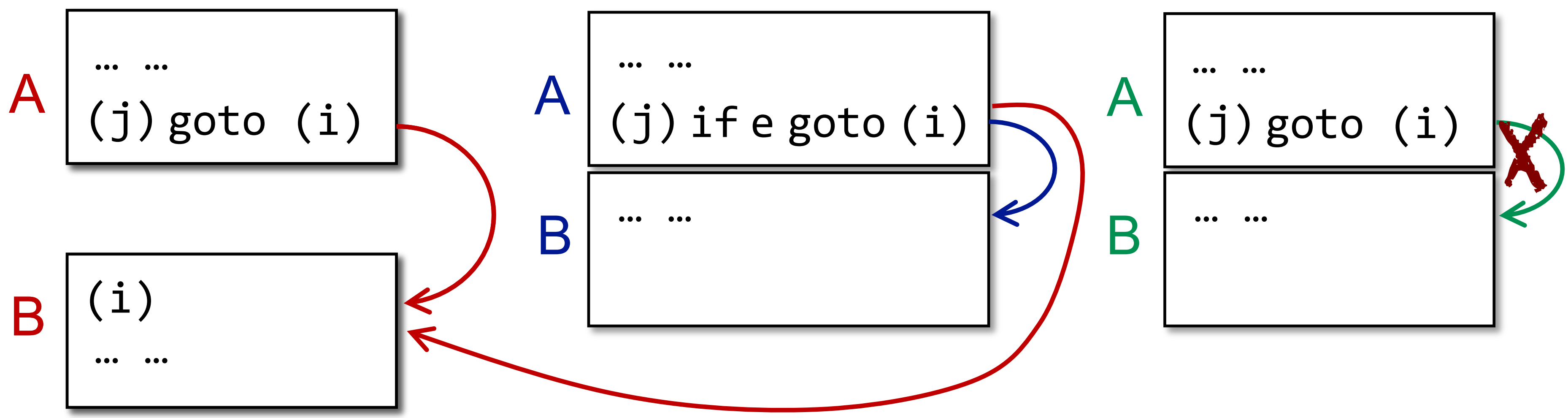
Control Flow Graph (CFG)

- 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



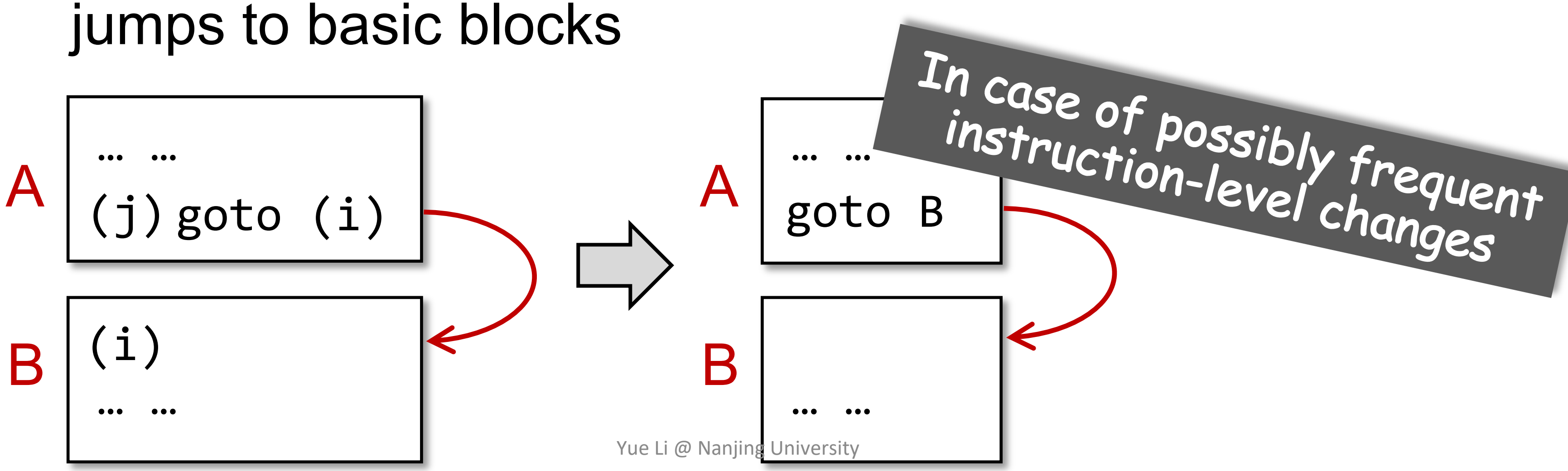
Control Flow Graph (CFG)

- 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

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

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

B1
```
x = input
y = x - 1
```

B2
```
z = x * y
if z < x goto B4
```

B3
```
p = x / y
q = p + y
```

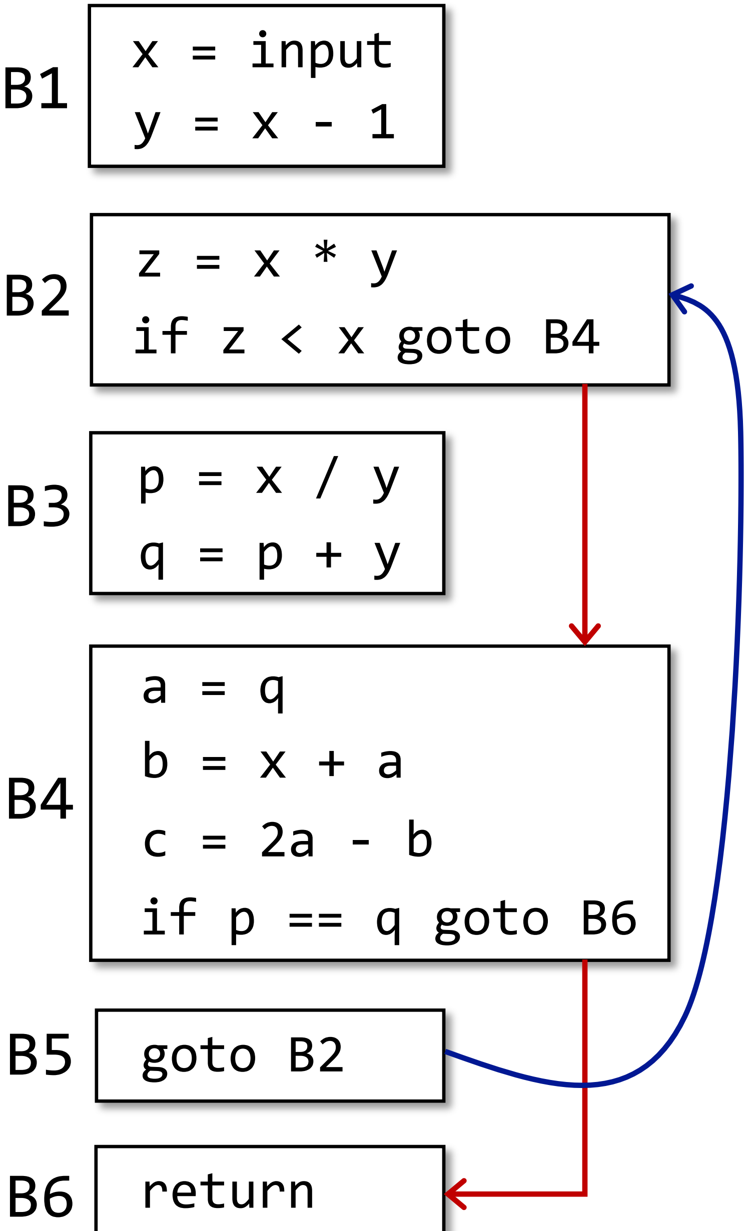
B4
```
a = q
b = x + a
c = 2a - b
if p == q goto B6
```

B5
```
goto B2
```

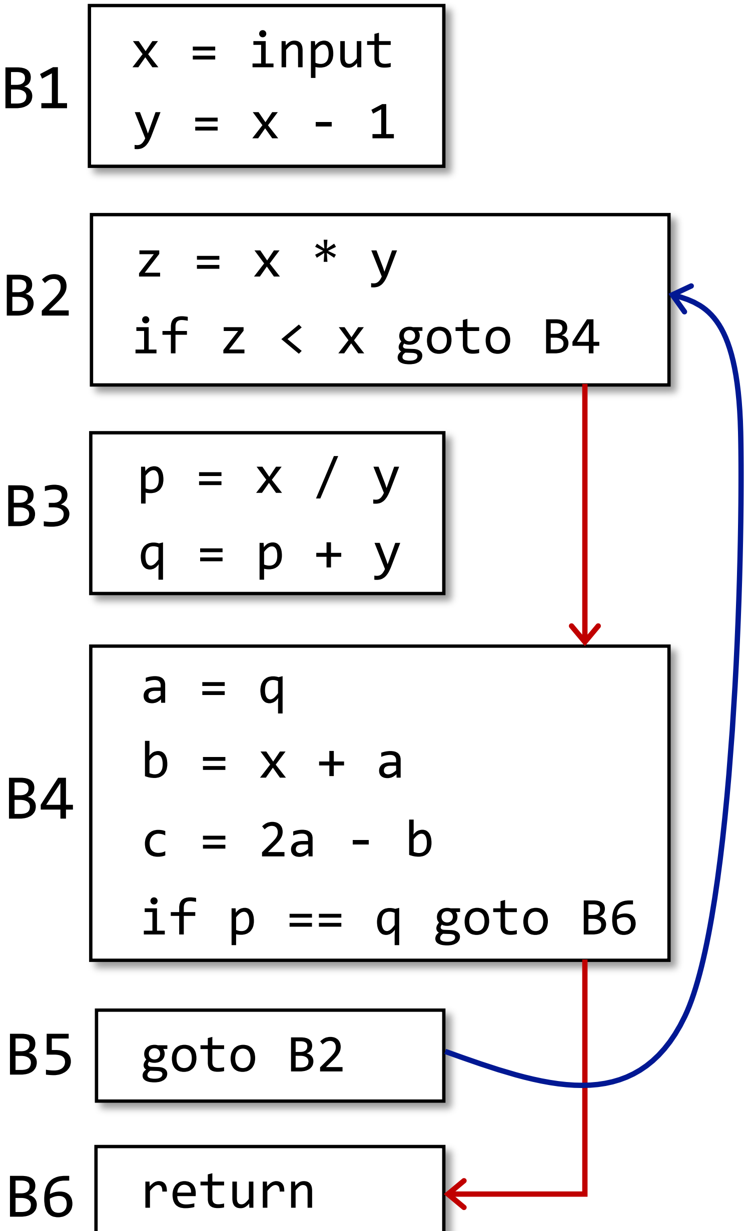
B6
```
return
```

Add edges in CFG

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

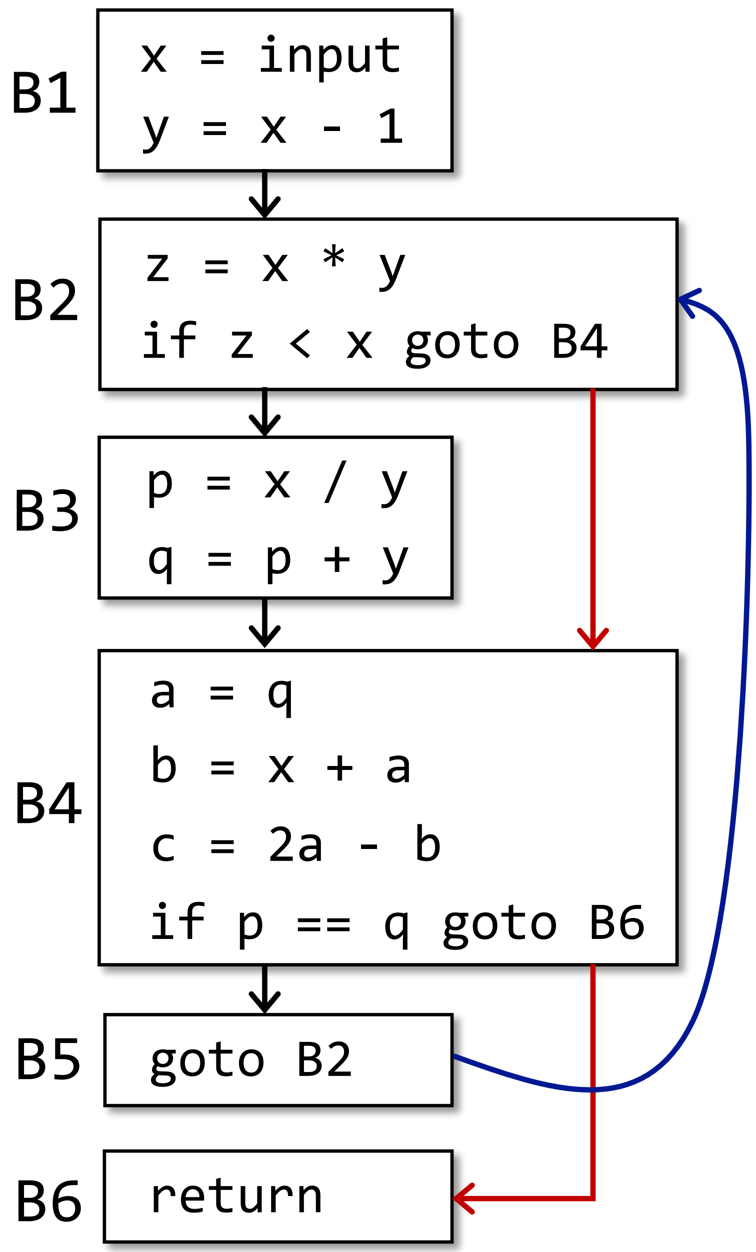


Add edges in CFG



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

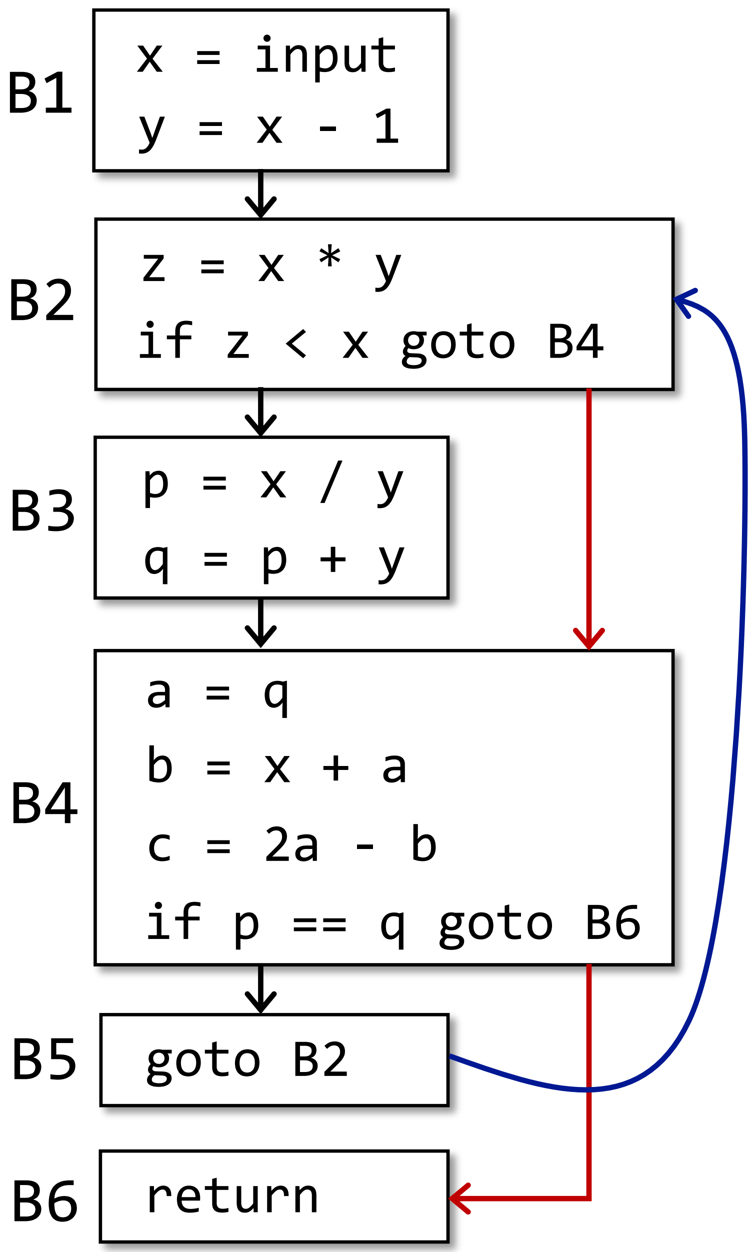
B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump



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

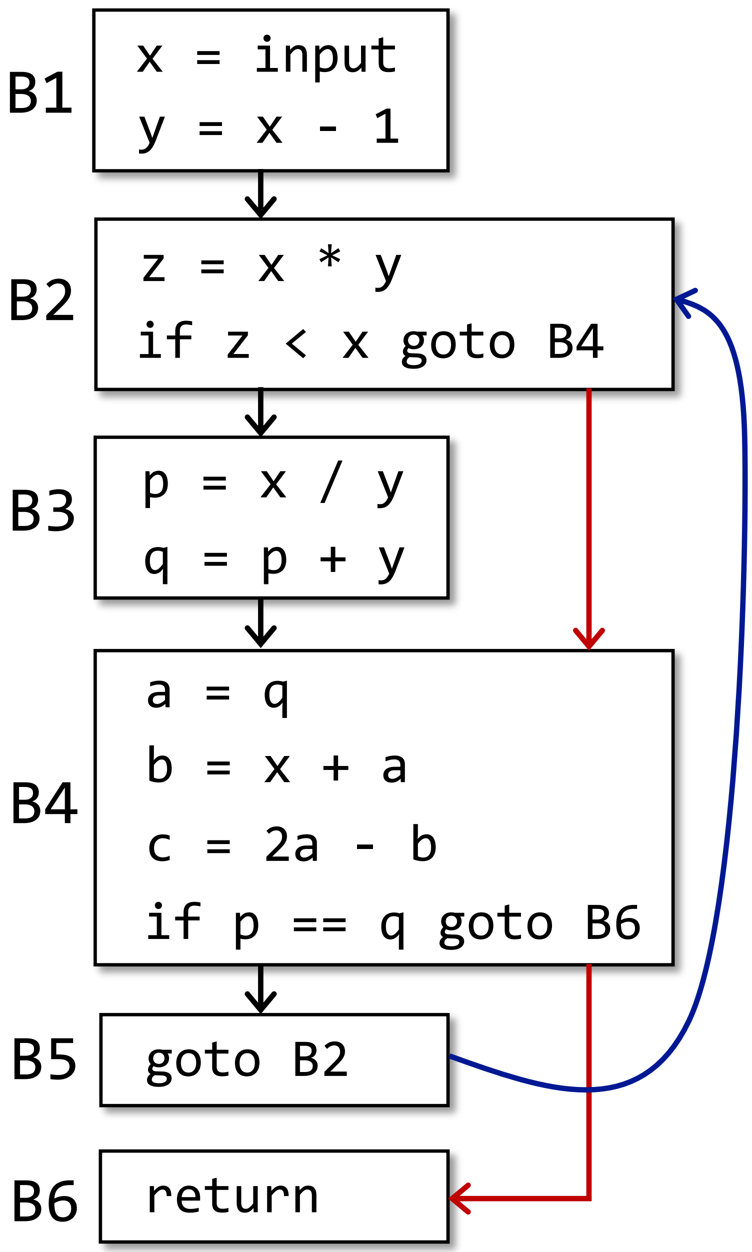


Add edges in CFG

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

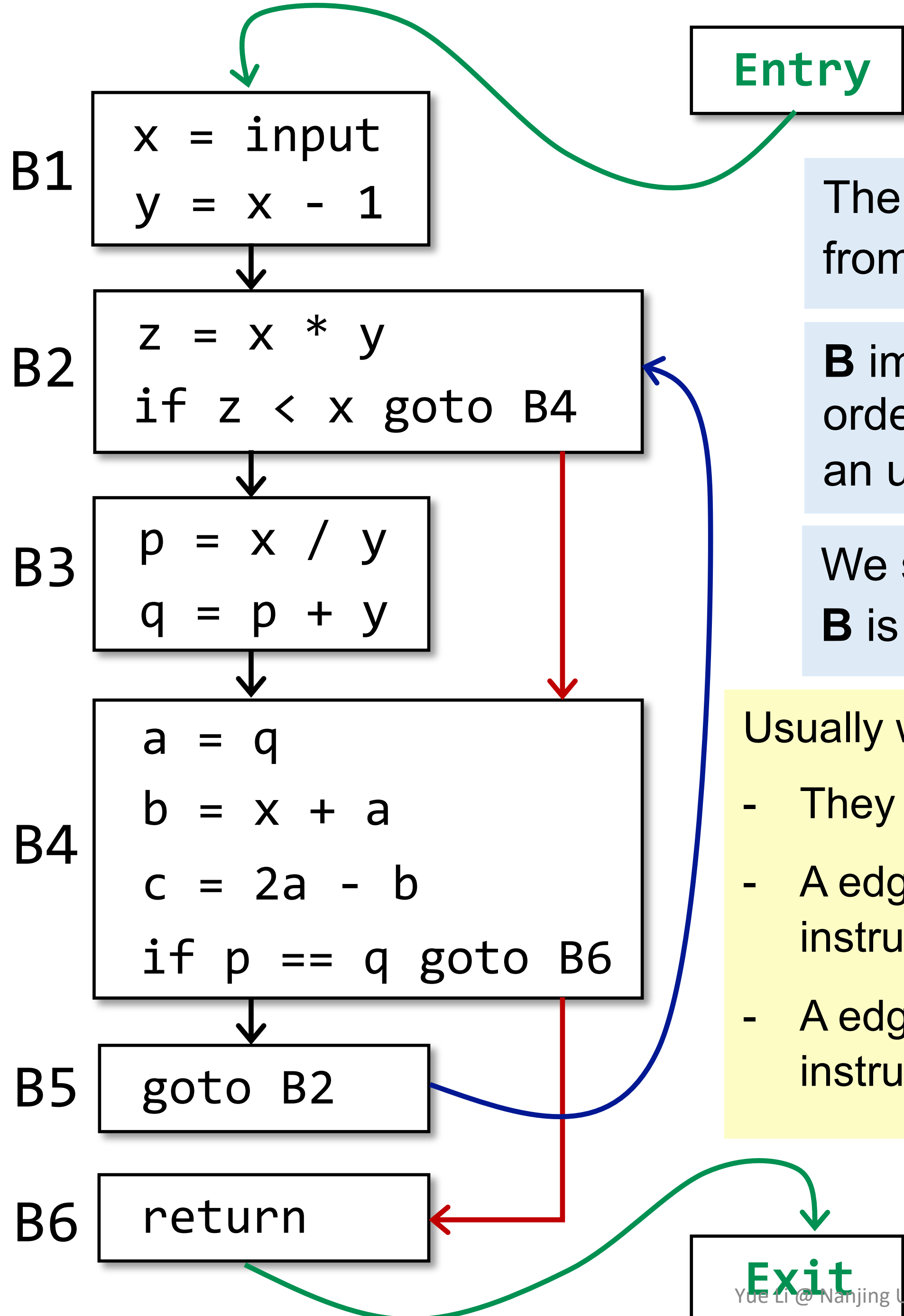
There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

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
- A edge from Entry to the BB containing the first instruction of IR
- A edge to Exit from any BB containing an instruction that could be the last instruction of IR



Add edges in CFG

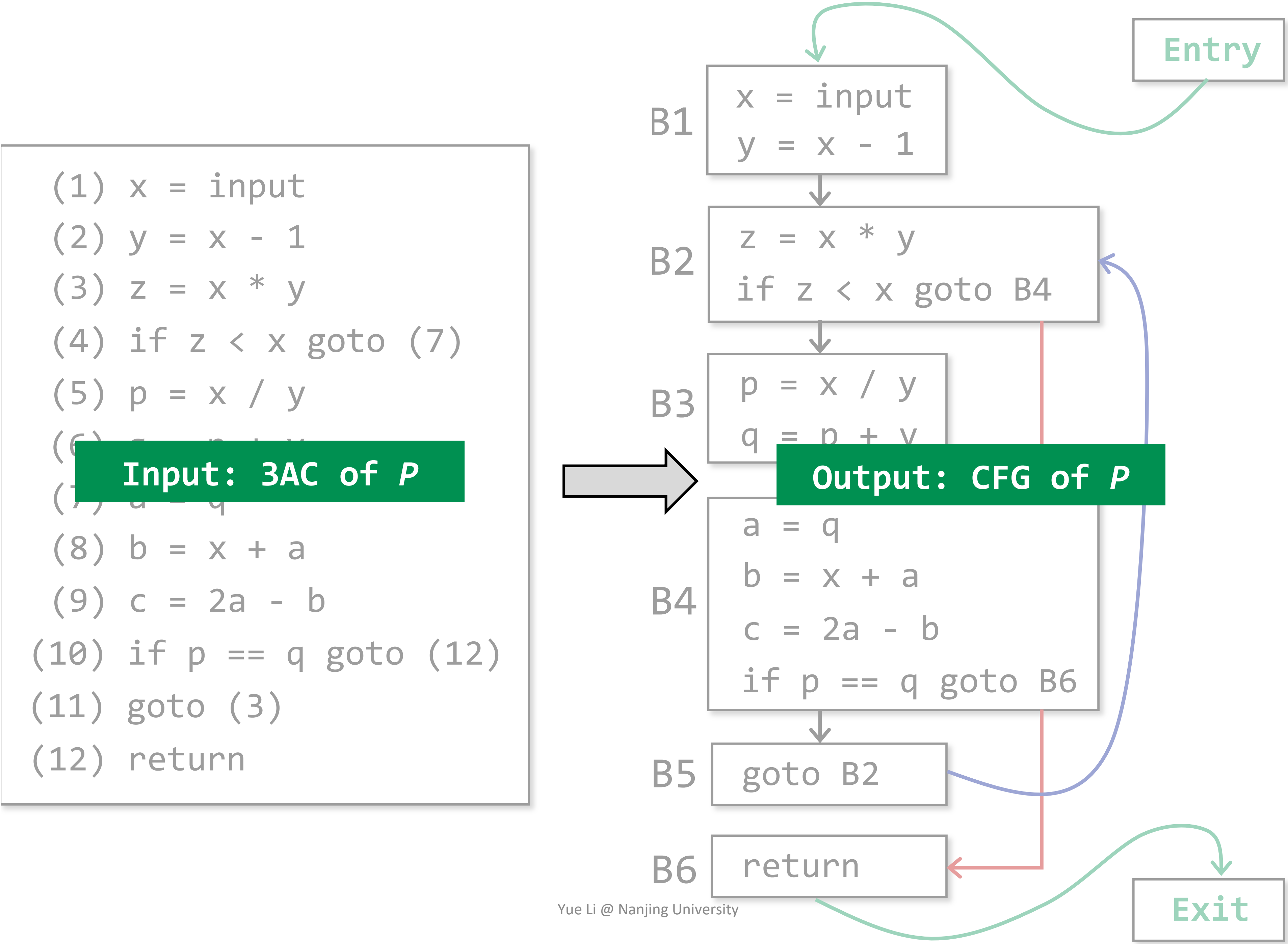
There is a conditional or unconditional jump from the end of **A** to the beginning of **B**

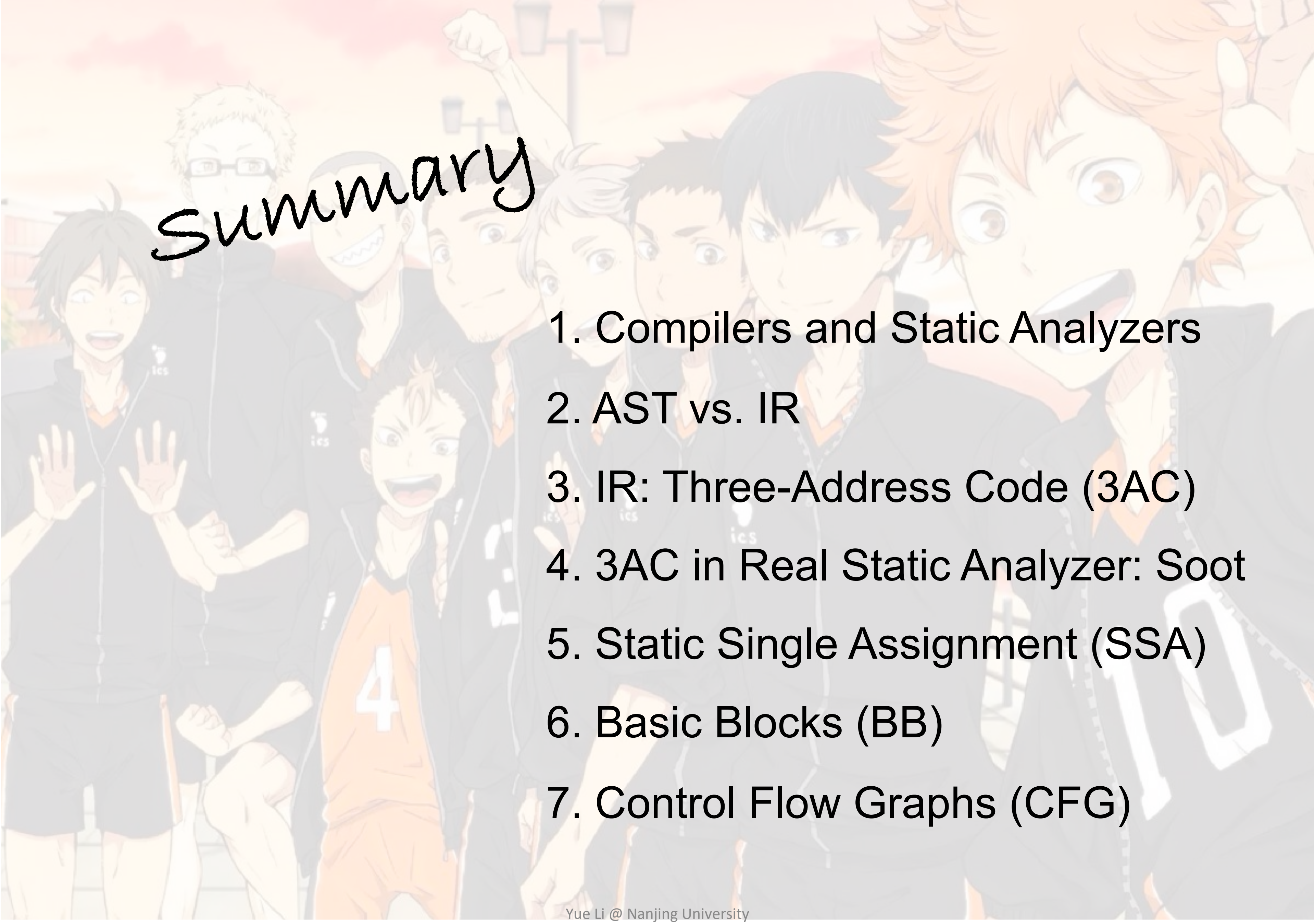
B immediately follows **A** in the original order of instructions and **A** does not end in an unconditional jump

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
- A edge from Entry to the BB containing the first instruction of IR
- A 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



Summary

1. Compilers and Static Analyzers
2. AST vs. IR
3. IR: Three-Address Code (3AC)
4. 3AC in Real Static Analyzer: Soot
5. Static Single Assignment (SSA)
6. Basic Blocks (BB)
7. Control Flow Graphs (CFG)

The X You Need To Understand in This Lecture

- The relation between compilers and static analyzers
- Understand 3AC and its common forms
- How to build basic blocks on top of IR
- How to construct control flow graphs on top of BBs?