

# Software Security

## Program Analysis Techniques ( 1 )

Guosheng Xu,  
guoshengxu@bupt.edu.cn

# Outline

- **Techniques used in Security Analysis**
- **Basic Program Analysis**
  - Control flow analysis
  - Data flow analysis
- **Taint Analysis**
- **Symbolic execution**
- **Fuzzing**

# Techniques used in security analysis

**Vulnerability  
Detection**

**Malware  
Analysis**

**Code Obfuscation**

**De-obfuscation**

**Program Analysis: Data flow, Control flow**

**Symbolic Execution**

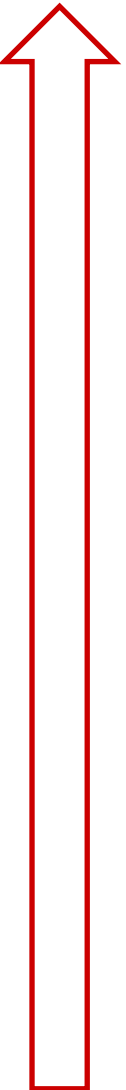
**Fuzzing**

**Taint Analysis**

**Dynamic Analysis,**

**Machine learning techniques, etc.**

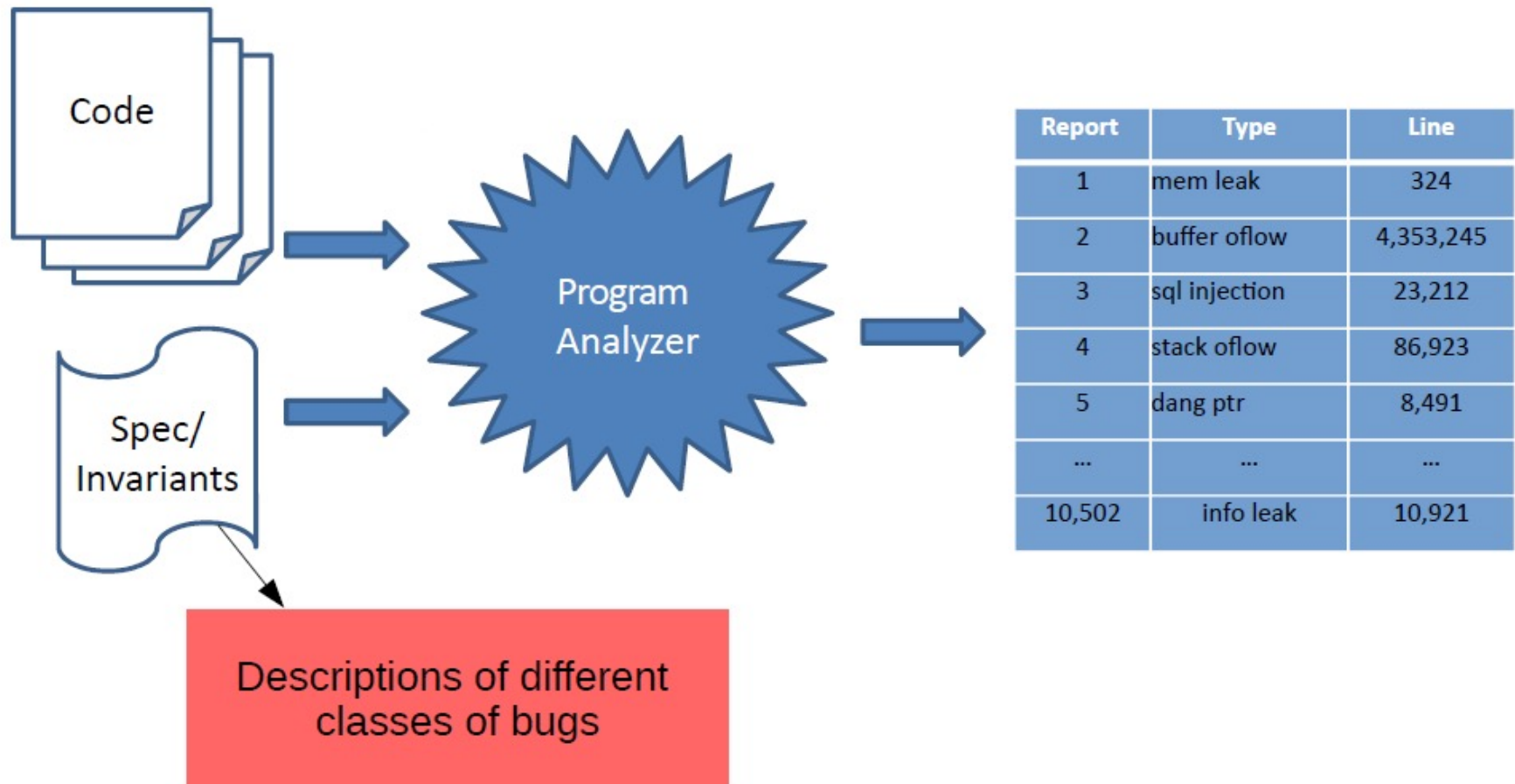
**Software, Apps, Smart Contracts, etc.**



# Take vulnerability detection as an example

- **How do we deal with security vulnerabilities?**
  - Automatically find and fix vulnerabilities
  - Techniques used in vulnerability detection

# Finds vulnerabilities with program analysis



# Automated vulnerability detection

## ■ Two Options

- Static analysis
  - Inspect code or run automated method to find errors or gain confidence about their absence
  - Try to aggregate the program behavior over a large number of paths without enumerating them explicitly
- Dynamic analysis
  - Run code, possibly under instrumented conditions, to see if there are likely problems
  - Enumerate paths but avoid redundant ones

**However, Finding paths from sources to sinks is not sufficient**


**Are those paths feasible for an attack?**

# Automated vulnerability detection

## ■ Main challenges

```
int main (int x, int y)
{
    if (2*y!=x)
        return -1;
    if (x>y+10)
        Return -1;
    ....
    ... /* buggy code */
}
```

What values of x and y will cause the program to reach here



1. Too many paths (may be infinite)
2. How will program analyzer find inputs that will reach different parts of code to be tested?



# An Example

```

1 void onCreate(Bundle savedInstanceState) {
2     Intent intent=getIntent();
3     String host = intent.getStringExtra("hostname");
4     String user = intent.getStringExtra("username");
5     String file = intent.getStringExtra("filename");
6     String url="http://www.example.com";
7     if (host.contains("example.com"))
8         url = "http://" + host + "/";
9     if (file.contains(".."))
10        file = file.replace("..", "");
11    String userId = getUserID(user);
12    if (userId != null)
13        textView.setText(user);
14    String b64File = toBase64(file);
15    String httpPar = toHttpParams(b64File,user_id);
16    ...
17    try {
18        DefaultHttpClient httpC = new DefaultHttpClient();
19        HttpGet get = new HttpGet(url+httPar);
20        ...
21        httpC.execute(get);
22    }
23    catch(IOException e) {
24        e.printStackTrace();
25    }
26 }
27 String toBase64(String p) {
28     if(p=null || p.equals(""))
29         p = "/data/data/com.example/defaultFile.pdf";
30     else
31         p = "/data/data/com.example/public/" + p;
32     byte[] bytes = InputStream.read(p);
33     String b = Base64Encoder.toString(bytes);
34     return b;
35 }

```

**组件暴露漏洞示例：这个例子真的存在并且能够被利用么？**

- 第一行onCreate会启动一个组件
- 该暴露组件中有很多敏感语句，可能能够被攻击者利用，例如URL请求、数据库查询语句等
- 第19行httpget:如果攻击者能够改变url的值，到一个他能够控制的网站，那么该组件就能够被利用，将敏感数据泄露给攻击者
- 第31行：通过网络传输的文件路径。如果攻击者能够修改p的值，那么任意文件能够被传输
- **为了达到这些目的，攻击者只能通过发送Intent到这个组件，来改变Sink语句中的值**

# An Example

```

1 void onCreate(Bundle savedInstanceState) {
2     Intent intent=getIntent();
3     String host = intent.getStringExtra("hostname");
4     String user = intent.getStringExtra("username");
5     String file = intent.getStringExtra("filename");
6     String url="http://www.example.com";
7     if (host.contains("example.com"))
8         url = "http://" + host + "/";
9     if (file.contains(".."))
10         file = file.replace("..", "");
11     String userId = getUserID(user);
12     if (userId != null)
13         textView.setText(user);
14     String b64File = toBase64(file);
15     String httpPar = toHttpParams(b64File,user_id);
16     ...
17     try {
18         DefaultHttpClient httpC = new DefaultHttpClient();
19         HttpGet get = new HttpGet(url+httpPar);
20         ...
21         httpC.execute(get);
22     }
23     catch(IOException e) {
24         e.printStackTrace();
25     }
26
27
28
29
30
31
32     byte[] bytes = inputStream.read(p);
33     String b = Base64Encoder.toString(bytes);
34     return b;
35 }

```

组件暴露漏洞示例：这个例子真的存在并且能够被利用么？

代码中的安全敏感操作

(1) line7-8 对URL的处理

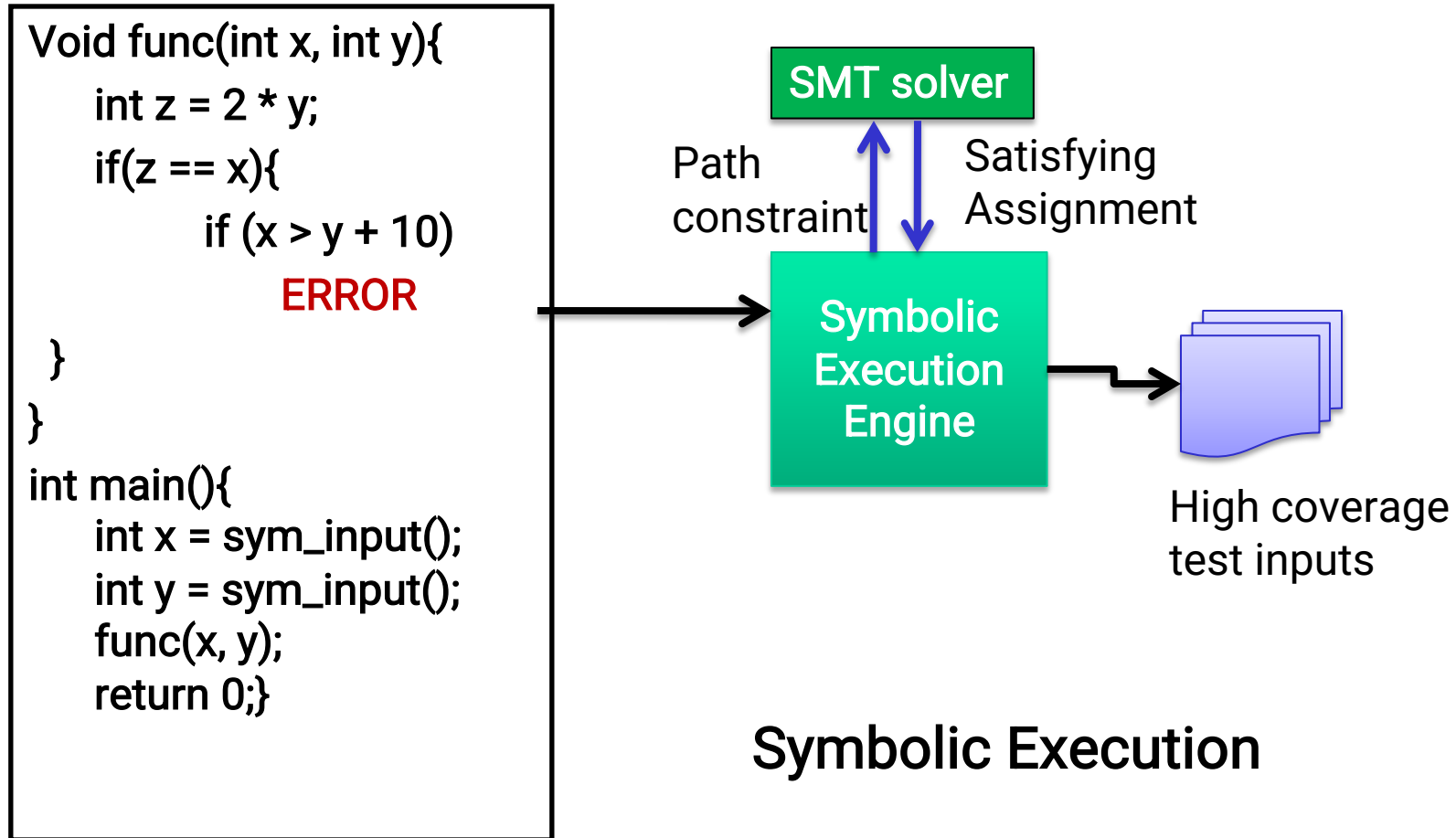
(2) line9-10对file的处理

(3) line11-13对GUI的处理

这些操作都会导致如前所提到的攻击失效，因此如此**明显的组件暴露漏洞并不存在**，并且不能够被利用。

因此，仅找到存在的攻击路径还不够，还需要对该路径上的操作进行分析，来判断攻击是否可行。

# Symbolic Execution



- For known (1-day) vulnerabilities, we could use program analysis techniques and symbolic execution techniques to identify.
- How to identify the 0-day vulnerabilities?

# Fuzzing Techniques

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)



# AI techniques used in Vulnerability detection

## ■ VulDeePecker: A Deep Learning-Based System for Vulnerability Detection , NDSS 2018

System	Dataset	FPR (%)	FNR (%)	TPR (%)	P (%)	F1 (%)
VulDeePecker vs. Other pattern-based vulnerability detection systems						
Flawfinder	BE-SEL	44.7	69.0	31.0	25.0	27.7
RATS	BE-SEL	42.2	78.9	21.1	19.4	20.2
Checkmarx	BE-SEL	43.1	41.1	58.9	39.6	47.3
VulDeePecker	BE-SEL	<b>5.7</b>	<b>7.0</b>	93.0	88.1	90.5
VulDeePecker vs. Code similarity-based vulnerability detection systems						
VUDDY	BE-SEL-NVD	0	95.1	4.9	100	9.3
VulPecker	BE-SEL-NVD	1.9	89.8	10.2	84.3	18.2
VulDeePecker	BE-SEL-NVD	<b>22.9</b>	<b>16.9</b>	83.1	78.6	80.8
VUDDY	BE-SEL-SARD	N/C	N/C	N/C	N/C	N/C
VulPecker	BE-SEL-SARD	N/C	N/C	N/C	N/C	N/C
VulDeePecker	BE-SEL-SARD	<b>3.4</b>	<b>5.1</b>	94.9	92.0	93.4

# Techniques used in security analysis

**Vulnerability  
Detection**

**Malware  
Analysis**

**Code Obfuscation**

**De-obfuscation**

**Program Analysis: Data flow, Control flow**

**Symbolic Execution**

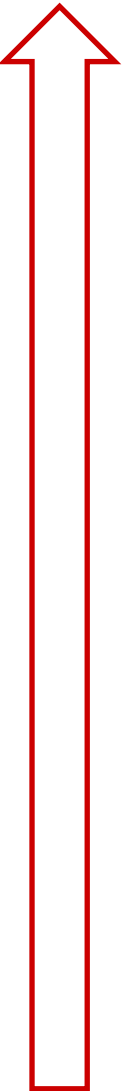
**Fuzzing**

**Taint Analysis**

**Dynamic Analysis,**

**Machine learning techniques, etc.**

**Software, Apps, Smart Contracts, etc.**

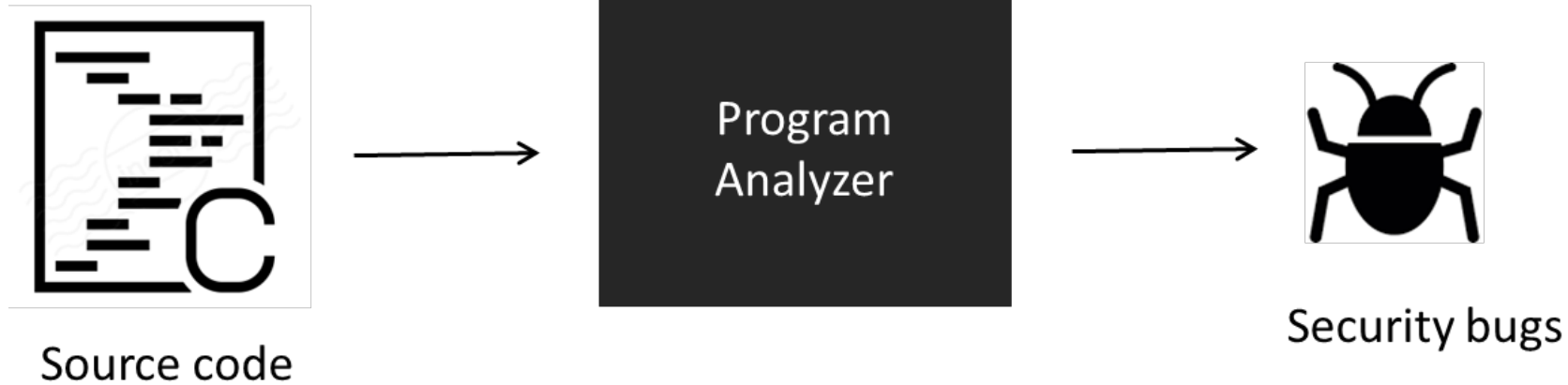


# **Basic Program Analysis**

## **-- Control Flow Analysis**



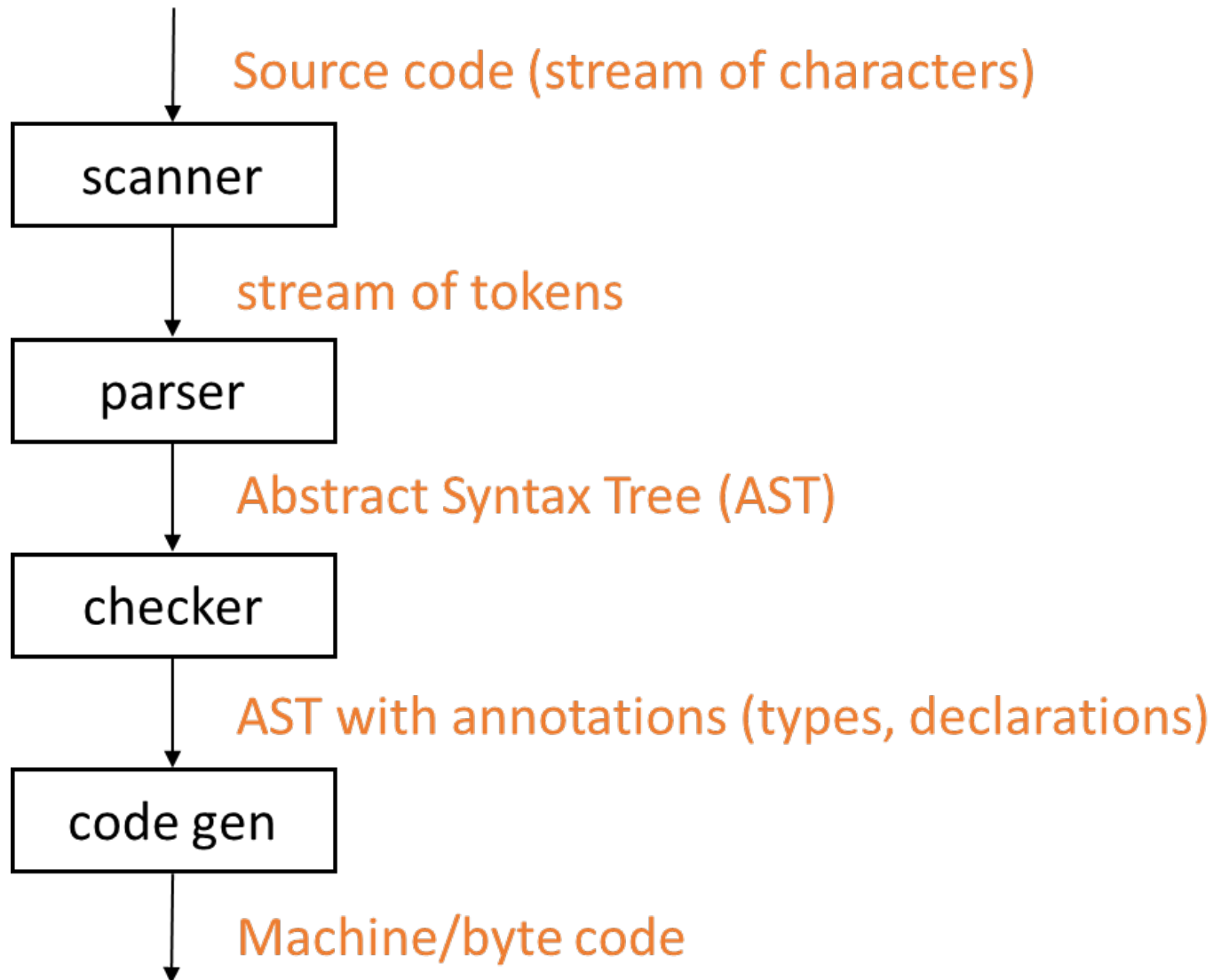
# Our Goal



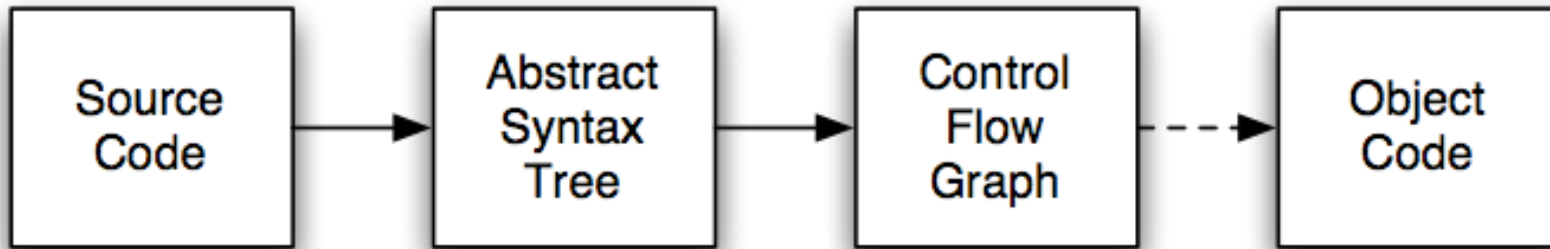
Program analyzer must be able to understand program properties (e.g., can a variable be NULL at a particular program point? )

→ Must perform control and data flow analysis

# The Structure of a Compiler



# Compiler Overview



- **Abstract Syntax Tree : Source code parsed to produce AST**
- **Control Flow Graph: AST is transformed to CFG**
- **Data Flow Analysis: operates on CFG**

# Syntactic Analysis句法分析

- **Input: sequence of tokens from scanner**
- **Output: abstract syntax tree**
- **Actually,**
  - parser first builds a parse tree
  - AST is then built by translating the parse tree
  - parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack

解析树，很少显式构建；仅取决于解析器如何将内容推送到堆栈

# Example

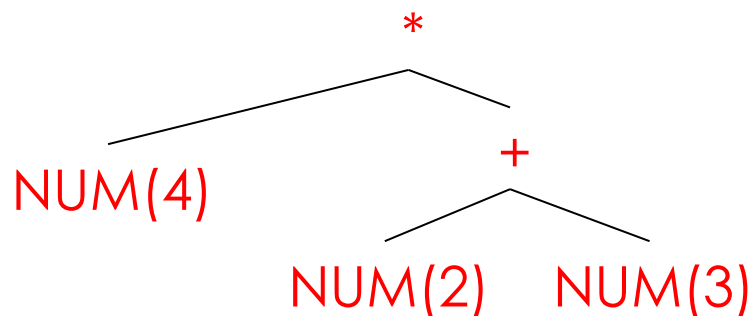
## ■ Source Code

4\*(2+3)

## ■ Parser input

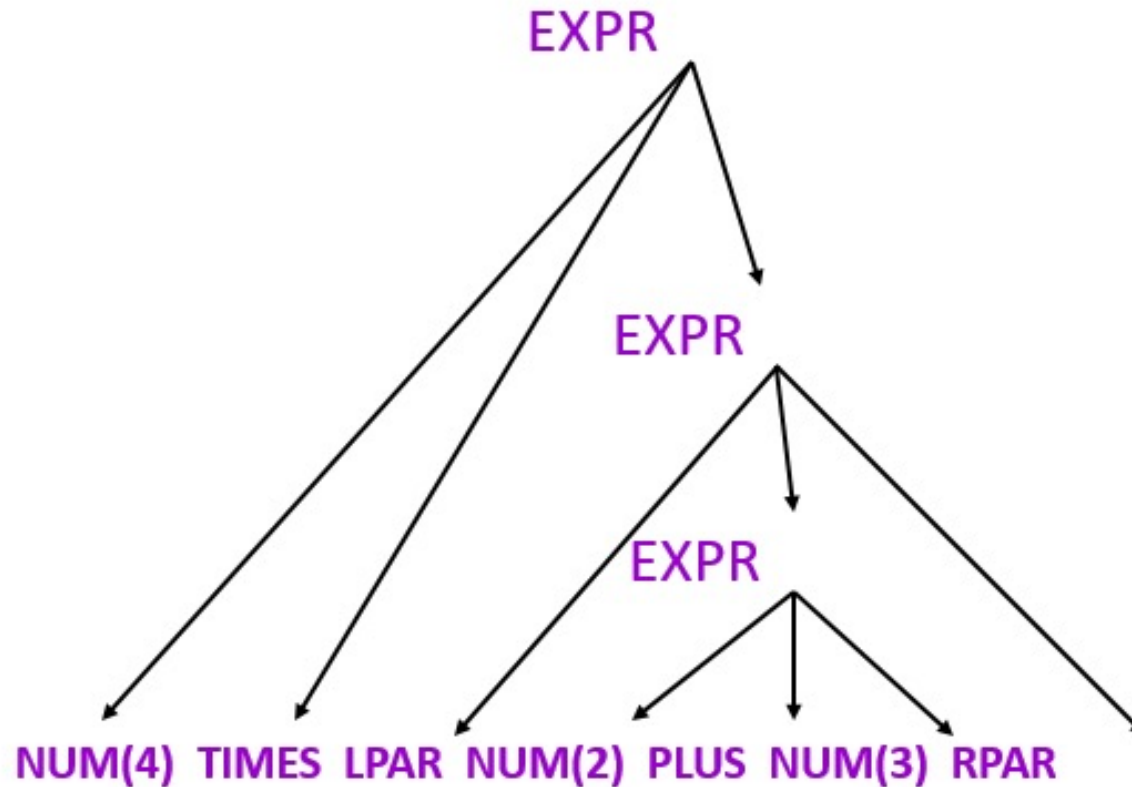
NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR

## ■ Parser output (AST):



在计算机科学中，**抽象语法树**（abstract syntax tree或者缩写为AST），或者**语法树**（syntax tree），是**源代码**的抽象语法结构的树状表现形式，这里特指编程语言的源代码。树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，**嵌套**括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于if-condition-then这样的条件跳转语句，可以使用带有两个分支的节点来表示。

# Parse tree for the example: $4*(2+3)$



leaves are tokens

# Another example

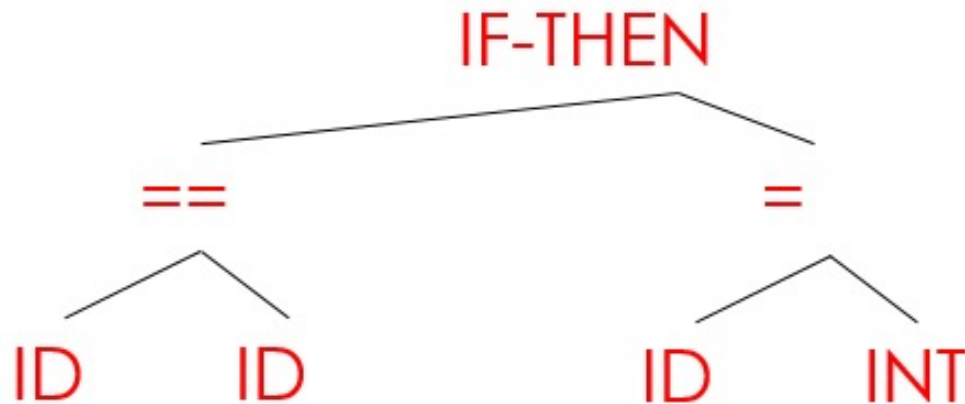
- Source Code

if (x == y) { a=1; }

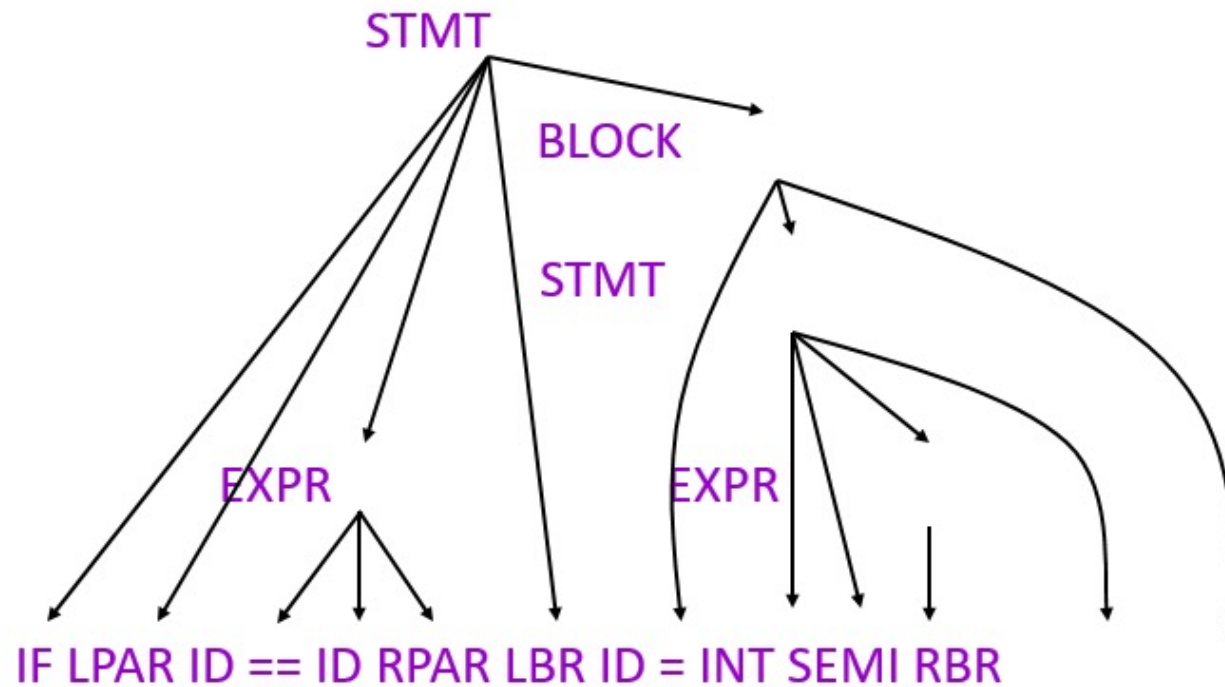
- Parser input

IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR

- Parser output (AST):



# Parse tree for example: if (x==y) {a=1;}



leaves are tokens



# Parse Tree

- Representation of grammars in a tree-like form.

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. ... Dragon Book

- Is a one-to-one mapping from the grammar to a tree-form.

# Abstract Syntax Tree (AST)

- Simplified syntactic representations of the source code, and they're most often expressed by the data structures of the language used for implementation

ASTs differ from parse trees because superficial distinctions of form, unimportant for translation, do not appear in syntax trees.. ... Dragon Book

AST与语法分析树有所不同，形式表面区别（对于翻译不重要）不会出现在语法树中

- Without showing the whole syntactic clutter, represents the parsed string in a structured way, discarding all information that may be important for parsing the string, but isn't needed for analyzing it.

# Disadvantages of ASTs

## ■ AST has many similar forms

- E.g., for, while, repeat...until
- E.g., if, ?:, switch

```
int x = 1 // what's the value of x ?
```

```
// AST traversal can give the answer, right?
```

```
What about int x; x = 1; or int x= 0; x += 1; ?
```

## ■ Expressions in AST may be complex, nested

- $(x * y) + (z > 5 ? 12 * z : z + 20)$

## ■ Want simpler representation for analysis

- ...at least, for dataflow analysis

# CONTROL FLOW GRAPH & ANALYSIS

# Representing Control Flow

Control flow is implicit in an AST

Use a **Control-flow graph (CFG)**

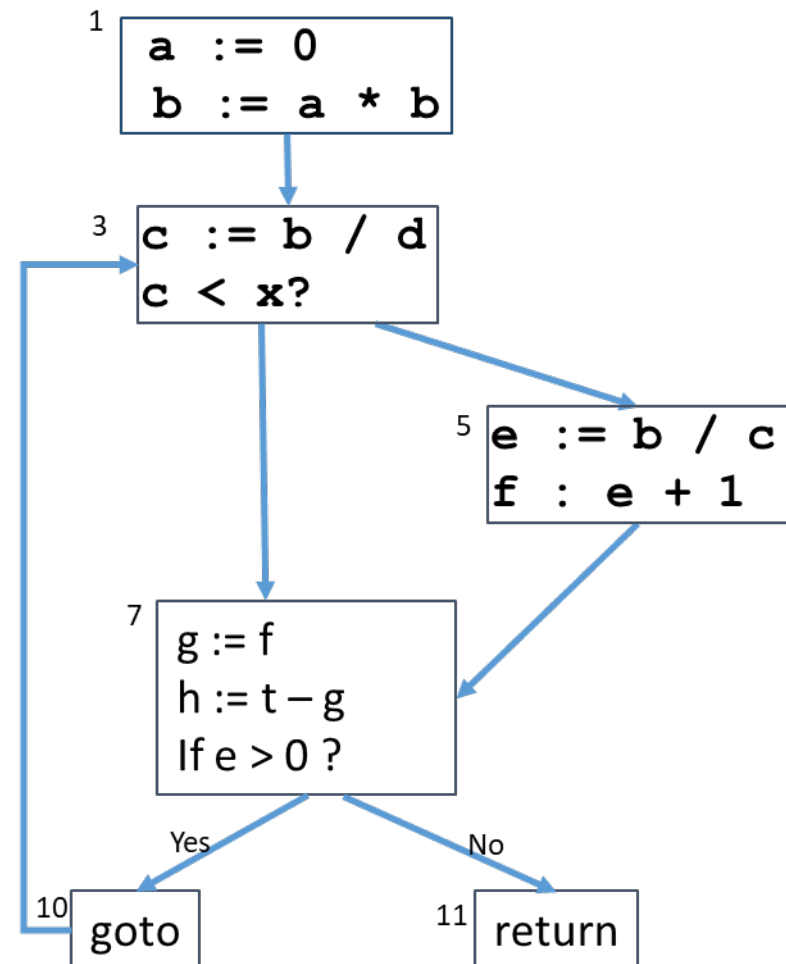
- Nodes represent statements
- Edges represent explicit flow of control

# What Is Control-Flow Analysis?

```

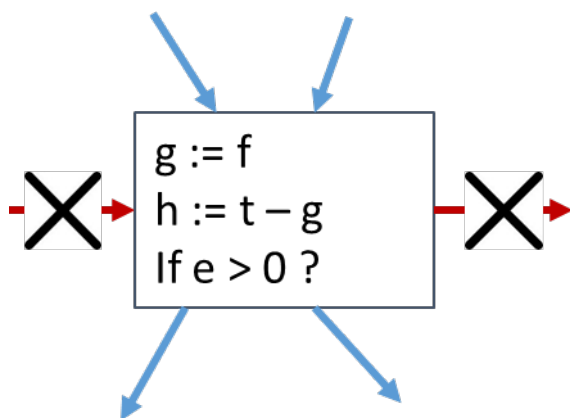
1      a := 0
2      b := a * b
3  L1:  c := b/d
4      if c < x goto L2  e :=
5      b / c
6      f := e + 1
7  L2:  g := f
8      h := t - g
9      if e > 0 goto L3
10 goto L1
11 L3:  return

```



# Basic Blocks

- A **basic block** is a sequence of straight line code that can be entered only at the beginning and exited only at the end



- Building Basic Blocks
- Identify leaders
  - The first instruction in a procedure,
  - The target of any branch,
  - An instruction immediately following a branch (implicit target)

1. 只有一个入口，表示程序中不会有其它任何地方能通过jump跳转类指令进入到此基本块中。
2. 只有一个出口，表示程序只有最后一条指令能导致进入到其它基本块去执行。

# Basic Block Example

```

1      a := 0
2      b := a * b
3  L1:  c := b/d
4      if c < x goto L2
5      e := b / c
6      f := e + 1
7  L2:  g := f
8      h := t - g
9      if e > 0 goto L3
10     goto L1
11  L3:  return

```

## Leaders?

– {1, 3, 5, 7, 10, 11}

## Blocks?

– {1, 2}

– {3, 4}

– {5, 6}

– {7, 8, 9}

– {10}

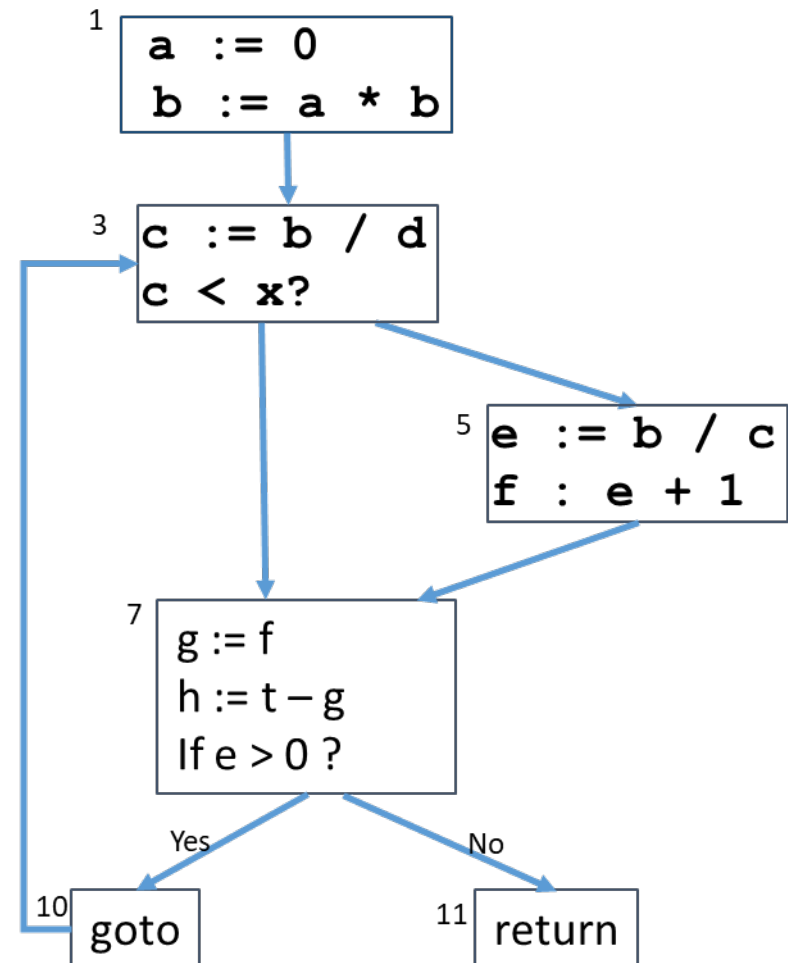
– {11}



# Building a CFG From Basic Block

## Construction

- Each CFG node represents a basic block
- There is an edge from node i to j if
  - Last statement of block i branches to the first statement of j, or
  - Block i does **not** end with an unconditional branch and is immediately followed in program order by block j (fall through)



# DATA FLOW ANALYSIS

# Data flow analysis

- Derives information about the dynamic behavior of a program by only examining the static code
- Intra-procedural analysis
- Flow-sensitive: sensitive to the control flow in a function
- Examples
  - Reaching Definitions
  - Available Expressions
  - Live Variables

# Reaching Definitions ( 到达定值 )

## ■ Concept of definition and use

- $a = x + y$

- is a definition of  $a$

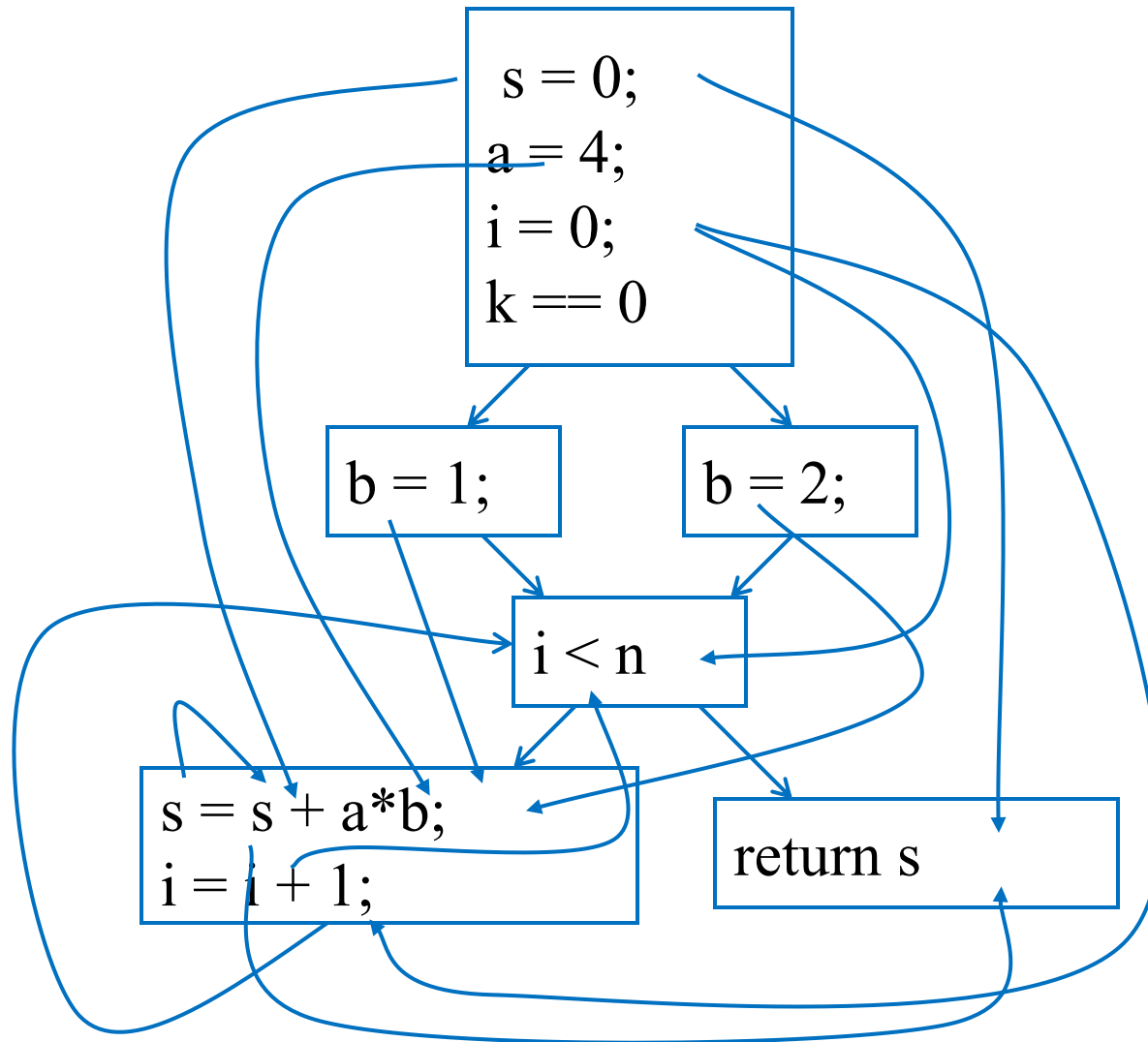
- is a use of  $x$  and  $y$

## ■ A definition reaches a use if

- value written by definition

- may** be read by use

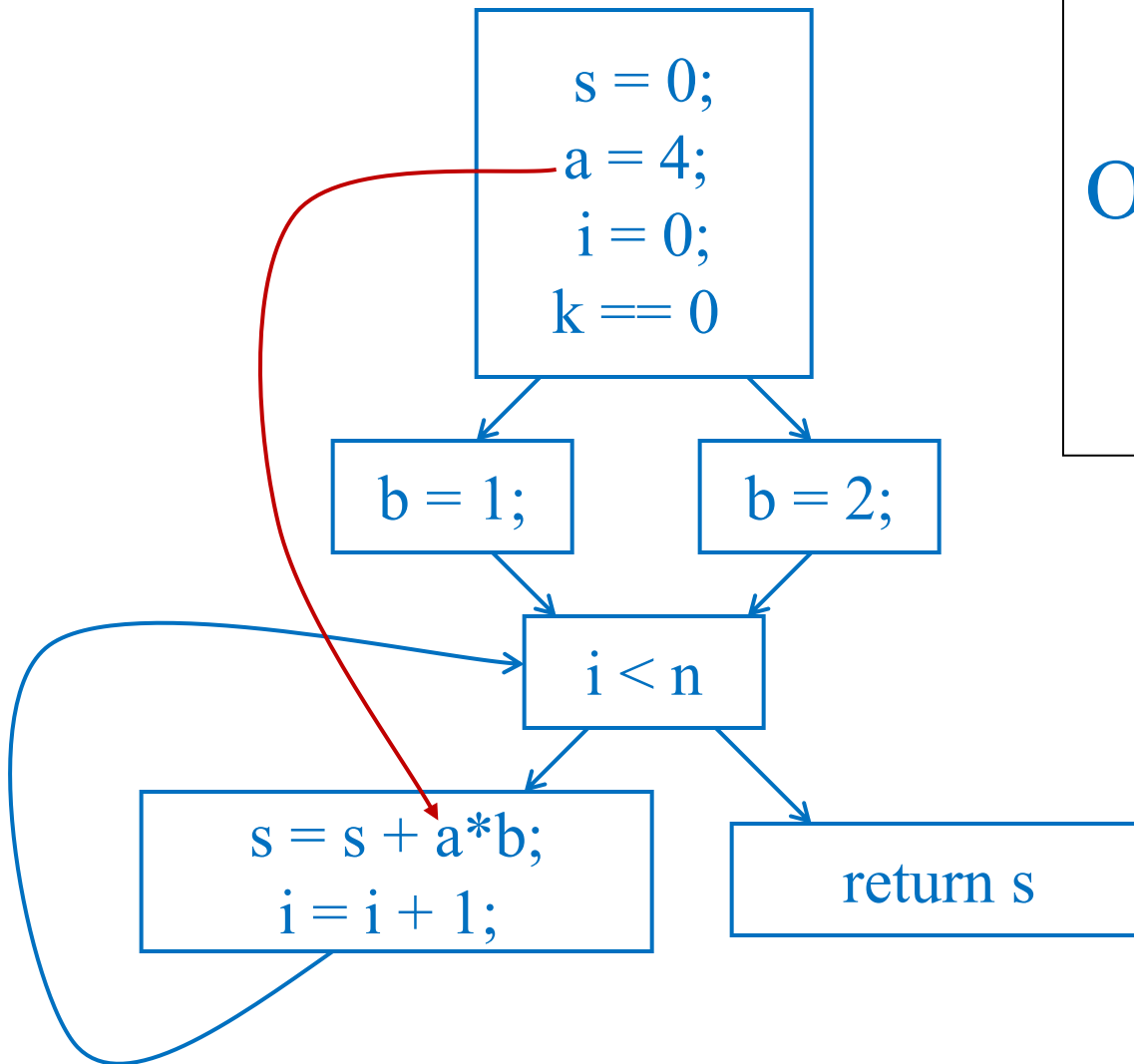
# Reaching Definitions



# Reaching Definitions and Constant Propagation/常量传播

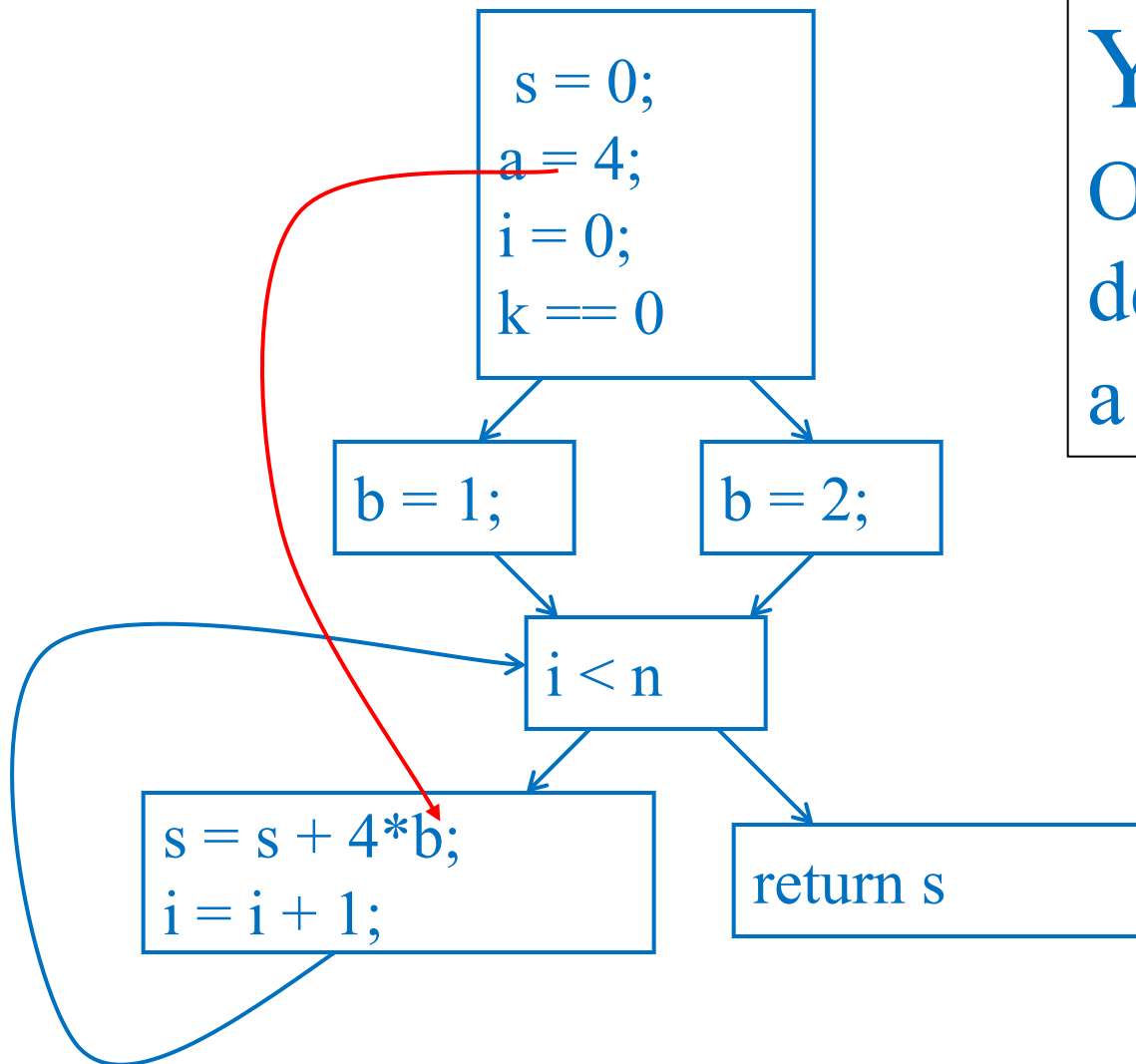
- **Is a use of a variable a constant?**
  - Check all reaching definitions
  - If all assign variable to same constant
  - Then use is in fact a constant
- **Can replace variable with constant**

# Is a Constant in $s = s + a * b$ ?



**Yes!**  
On all reaching  
definitions  
 $a = 4$

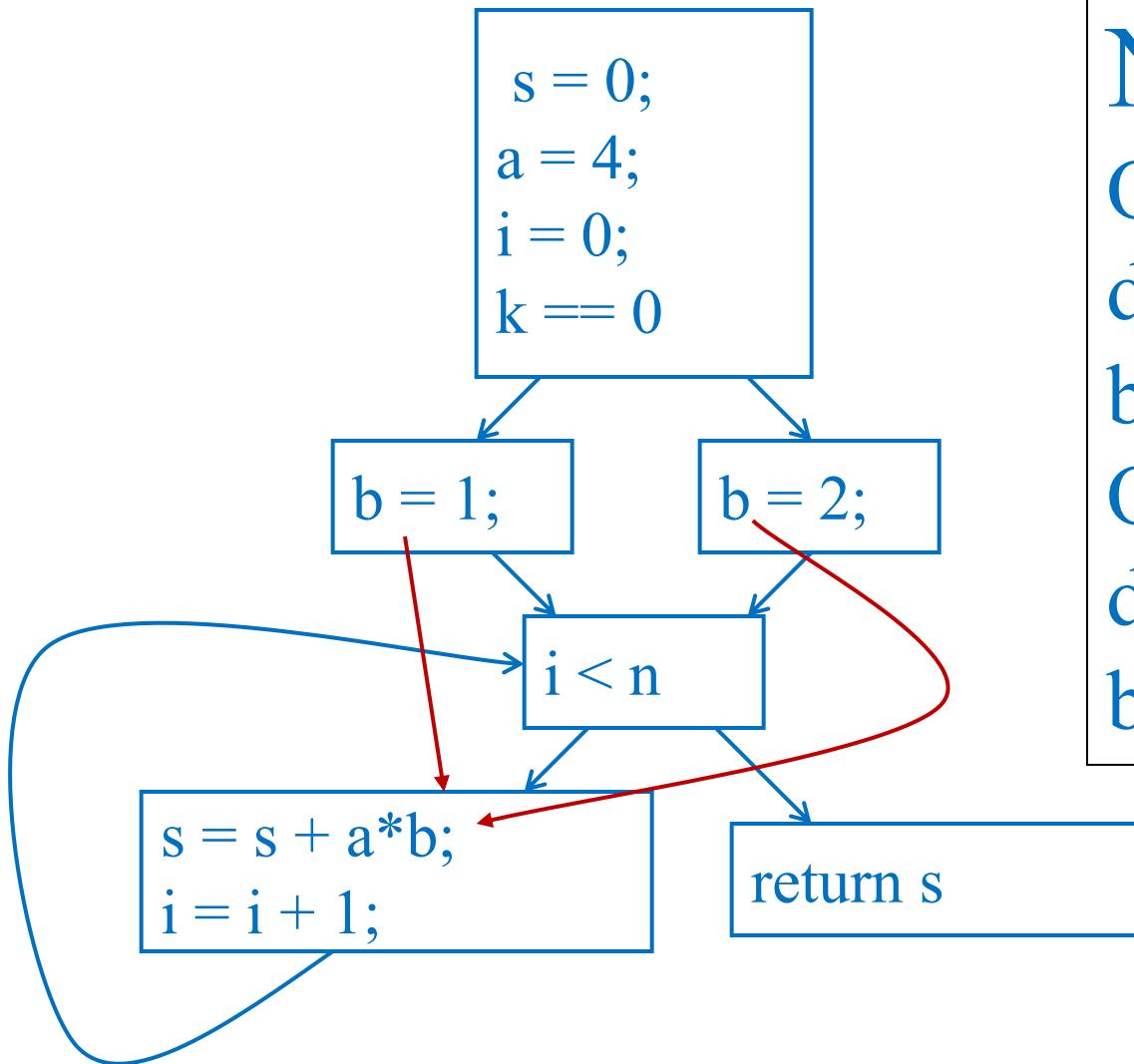
# Constant Propagation Transform



**Yes!**  
On all reaching  
definitions  
`a = 4`



# Is $b$ Constant in $s = s + a * b$ ?



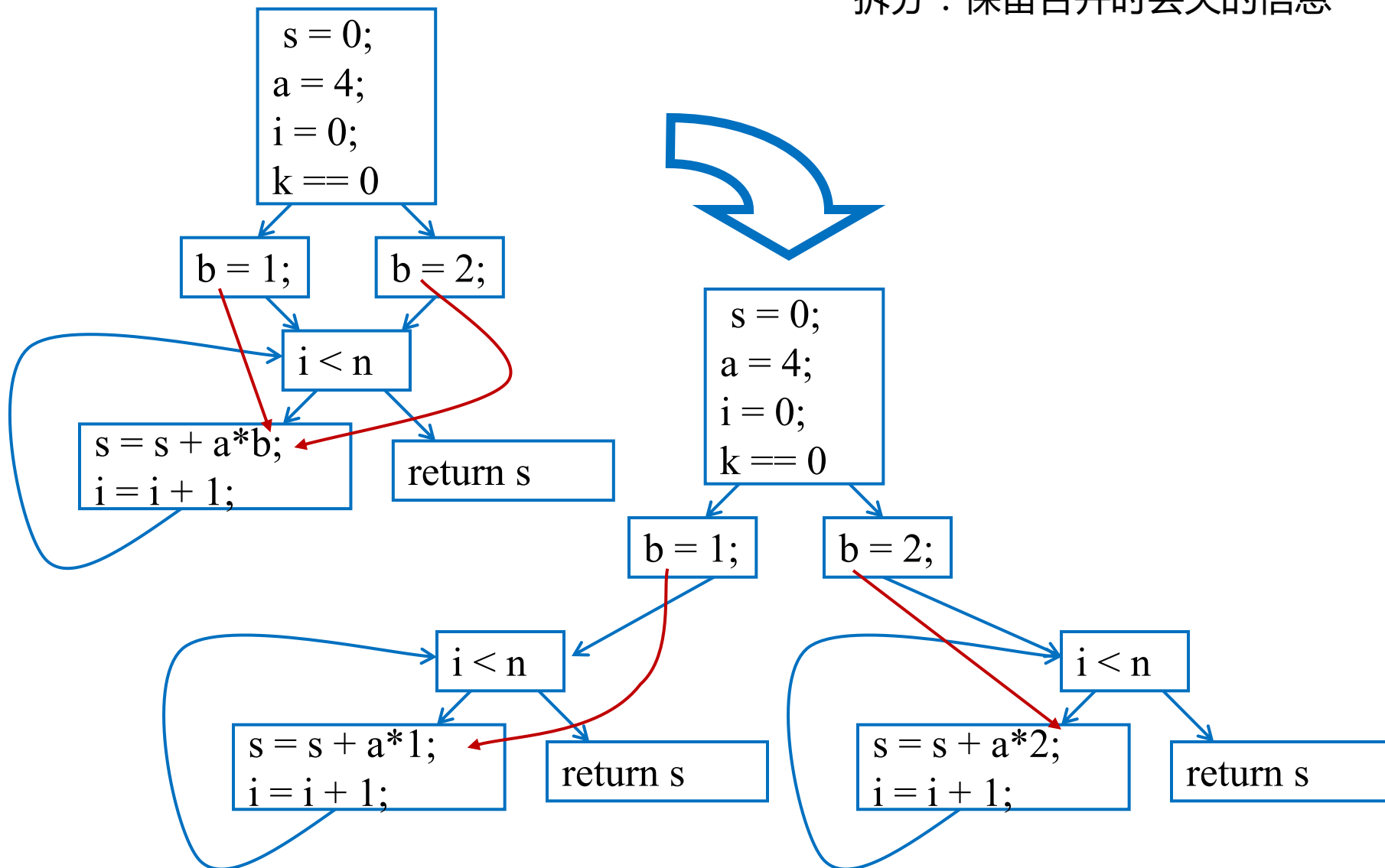
No!

One reaching  
definition with  
 $b = 1$

One reaching  
definition with  
 $b = 2$

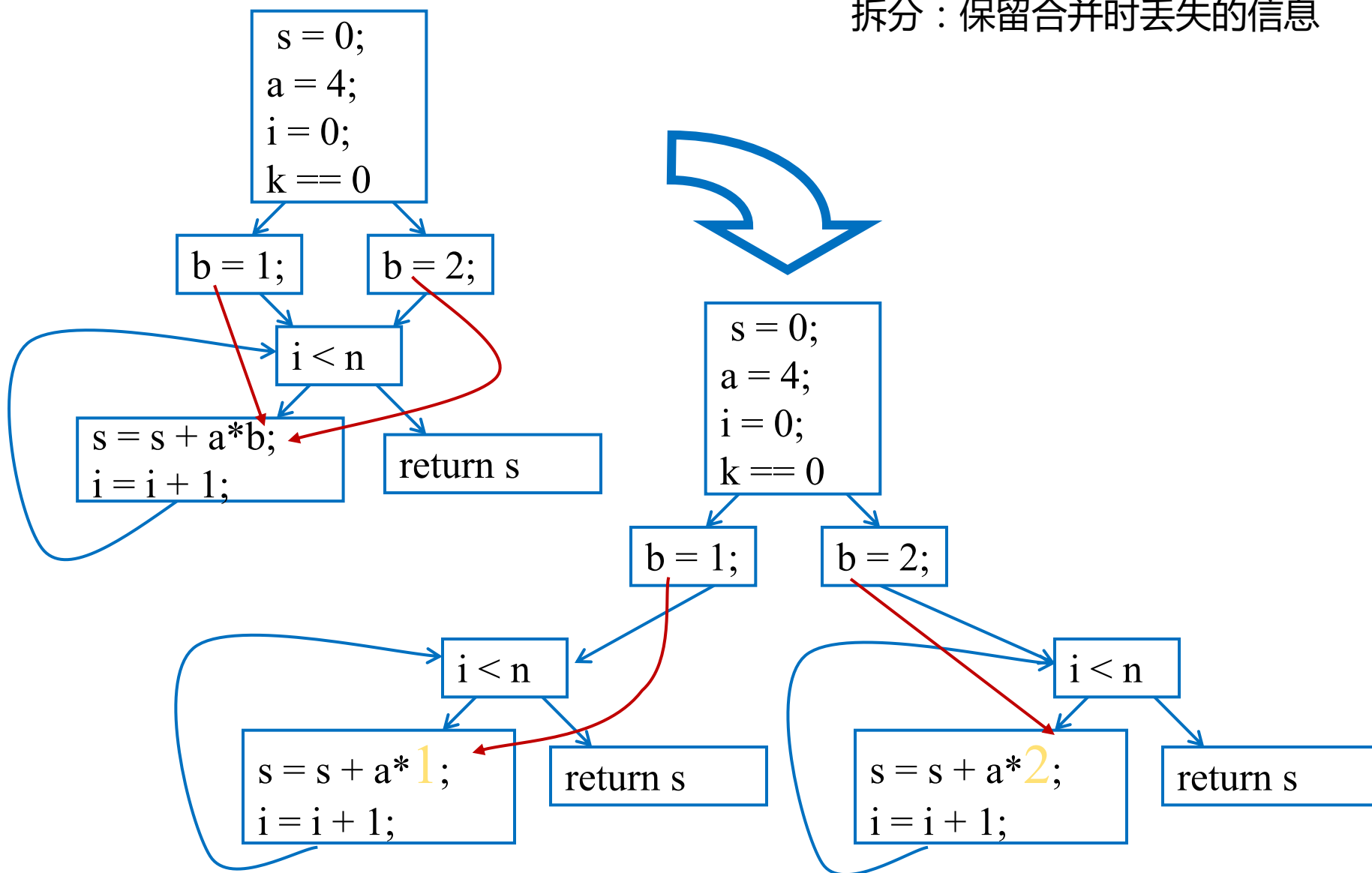
# Splitting: Preserves Information Lost At Merges

拆分：保留合并时丢失的信息



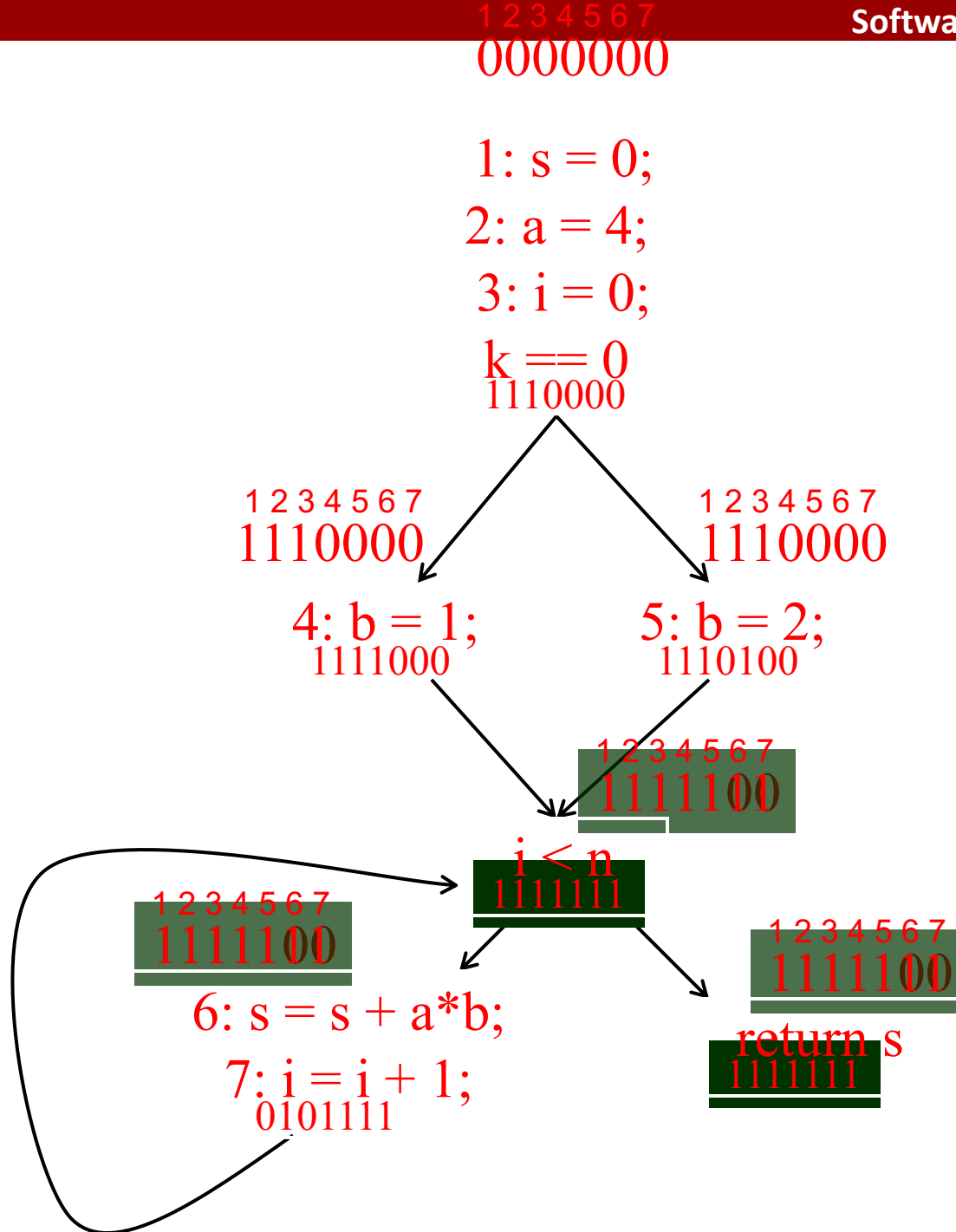
# Splitting: Preserves Information Lost At Merges

拆分：保留合并时丢失的信息



# Computing Reaching Definitions

- **Compute with sets of definitions**
  - represent sets using bit vectors
  - each definition has a position in bit vector
- **At each basic block, compute**
  - definitions that reach start of block
  - definitions that reach end of block
- **Do computation by simulating execution of program until reach fixed point**



# Data-Flow Analysis Schema ( 数据流分析模式 )

- **Data-flow value**: at every program point
- **Domain**: The set of possible data-flow values for this application
- **IN[S] and OUT[S]**: the data-flow values before and after each statement's
- **Data-flow problem**: find a **solution** to a set of constraints on the IN [s] 's and OUT[s] 's, for all statements s.
  - based on the semantics of the statements ("transfer functions" )
  - based on the flow of control.

# Constraints

- **Transfer function**: relationship between the data-flow values before and after a statement.
  - Forward:  $OUT[s] = f_s(IN[s])$
  - Backward:  $IN[s] = f_s(OUT[s])$
- **Within a basic block  $(s_1, s_2, \dots, s_n)$** 
  - $IN[s_{i+1}] = OUT[s_i]$ , for all  $i = 1, 2, \dots, n-1$

# Data-Flow Schemas on Basic Blocks

- Each basic block  $B (s_1, s_2, \dots, s_n)$  has
  - IN – data-flow values immediately before a block
  - OUT – data-flow values immediately after a block
  - $IN[B] = IN[S_1]$
  - $OUT[B] = OUT[S_n]$
  - $OUT[B] = f_B (IN[B] )$ 
    - Where  $f_B = fs_n \circ \dots \circ fs_2 \circ fs_1$



# Between Blocks

## ■ Forward analysis

- (eg: Reaching definitions)
- $IN[B] = \bigcup_{P \text{ a predecessor of } B} OUT[P]$

## ■ Backward analysis

- (eg: live variables)/实时变量
- $IN[B] = f_B(OUT[B])$
- $OUT[B] = \bigcup_{S \text{ a successor of } B} IN[S].$

# Formalizing Reaching Definitions

## ■ Each basic block has

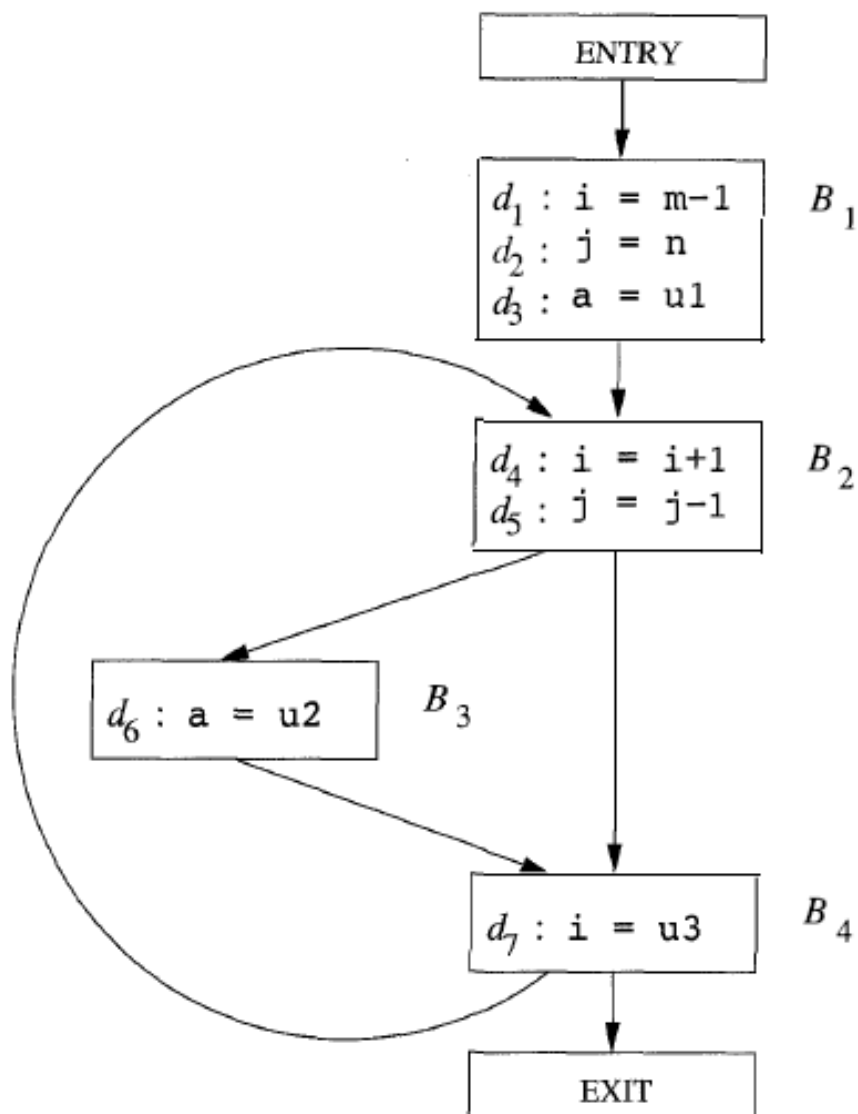
- IN - set of definitions that reach beginning of block/可到b开始
- OUT - set of definitions that reach end of block/可到b结尾
- GEN - set of definitions generated in block/在b中产生，可出
- KILL - set of definitions killed in block/到b中，但出不了b

■ **GEN[s = s + a\*b; i = i + 1;] = 0000011**

■ **KILL[s = s + a\*b; i = i + 1;] = 1010000**

■ **Compiler scans each basic block to derive GEN and KILL sets**

# Example



$$gen_{B_1} = \{ d_1, d_2, d_3 \}$$

$$kill_{B_1} = \{ \quad, \quad \}$$

$$gen_{B_2} = \{ d_4, d_5 \}$$

$$kill_{B_2} = \{ d_1, d_2, d_7 \}$$

$$gen_{B_3} = \{ d_6 \}$$

$$kill_{B_3} = \{ d_3 \}$$

$$gen_{B_4} = \{ d_7 \}$$

$$kill_{B_4} = \{ d_1, d_4 \}$$

# Dataflow Equations

- $IN[b] = OUT[b_1] \cup \dots \cup OUT[b_n]$ 
  - where  $b_1, \dots, b_n$  are predecessors of  $b$  in CFG
- $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- $IN[entry] = 00000000$
- Result: system of equations

# Solving Equations

- Use fixed point algorithm
- Initialize with solution of  $OUT[b] = 0000000$
- Repeatedly apply equations
  - $IN[b] = OUT[b_1] \cup \dots \cup OUT[b_n]$
  - $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- Until reach fixed point
- Until equation application has no further effect
- Use a worklist to track which equation applications may have a further effect

# Reaching Definitions Algorithm

```
for all nodes n in N
    OUT[n] = emptyset; // OUT[n] = GEN[n];
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = emptyset;
    for all nodes p in predecessors(n)
        IN[n] = IN[n] U OUT[p];

    OUT[n] = GEN[n] U (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed U { s };
```

# Questions

## ■ Does the algorithm halt?

- yes, because transfer function is monotonic/单调
- if increase IN, increase OUT
- in limit, all bits are 1

## ■ If bit is 0, does the corresponding definition ever reach basic block?

## ■ If bit is 1, does the corresponding definition always reach the basic block?

# Data flow analysis

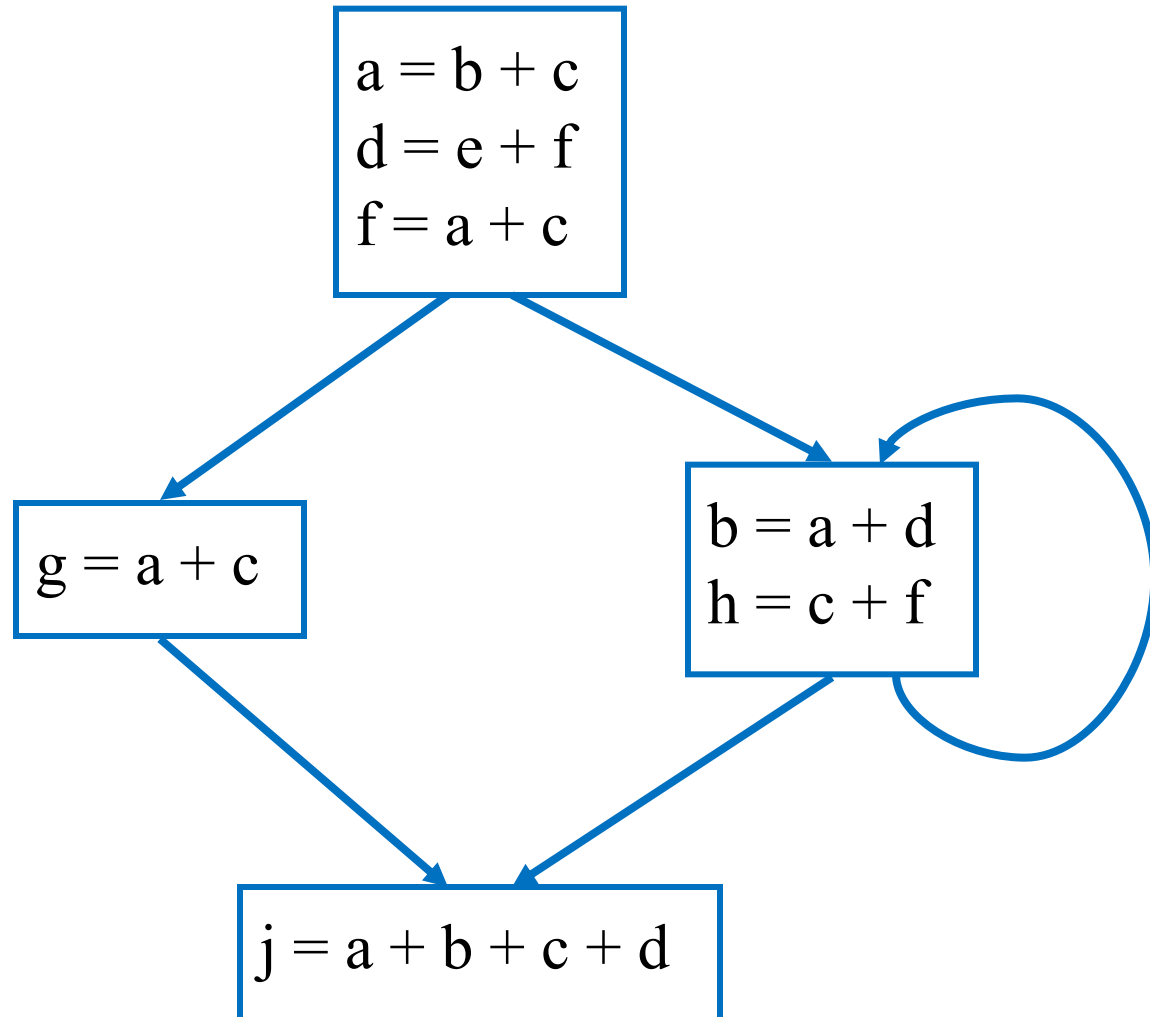
- Reaching Definitions
- Available Expressions可用表达式
- Live Variables



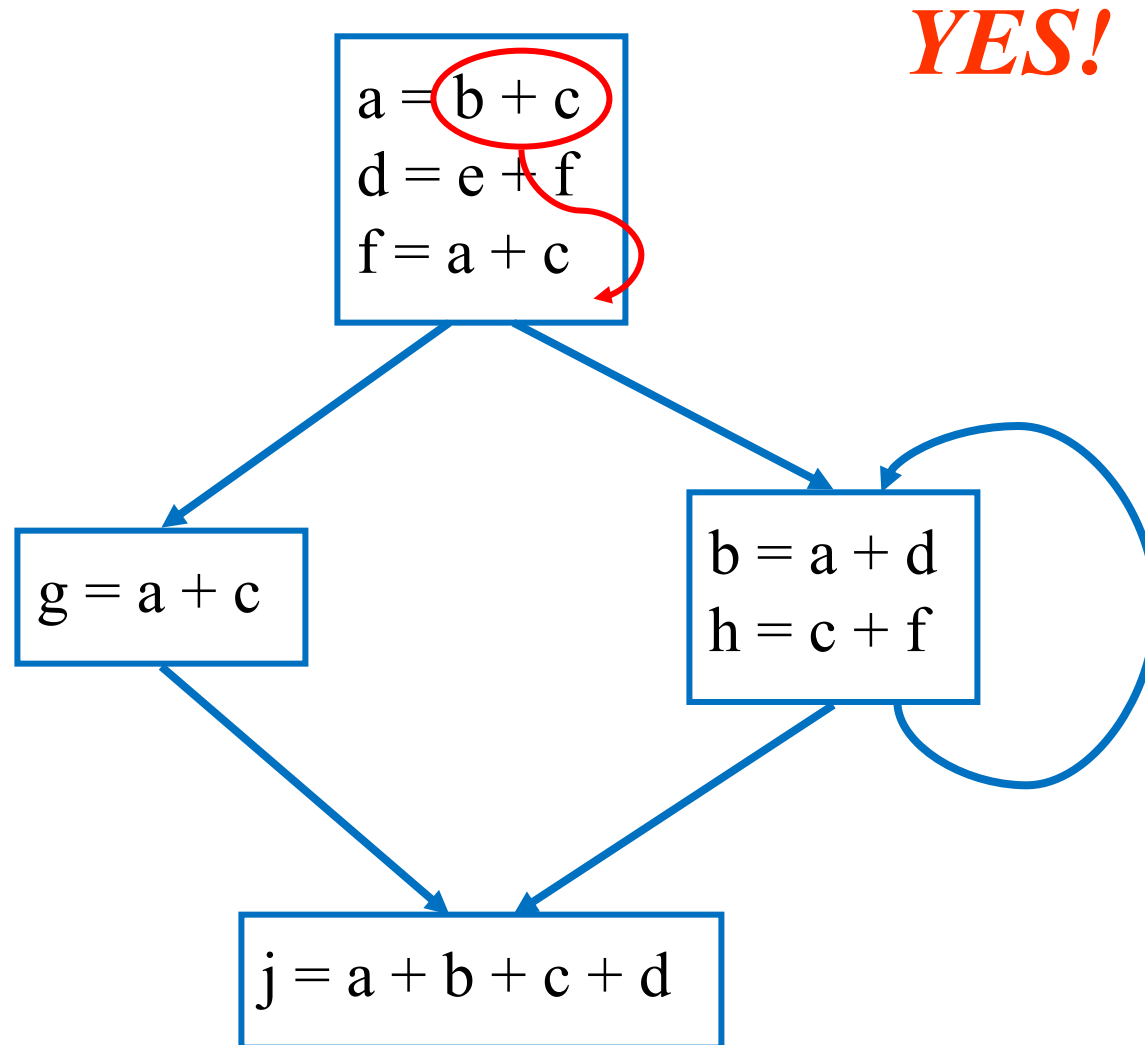
# Available Expressions (可用表达式)

- **An expression  $x+y$  is available at a point  $p$  if**
  - every path from the initial node to  $p$  must evaluate  $x+y$  before reaching  $p$ ,
  - and there are no assignments to  $x$  or  $y$  after the evaluation but before  $p$ .
- 
- **If expression is available at use, no need to reevaluate it**

# Example: Available Expression

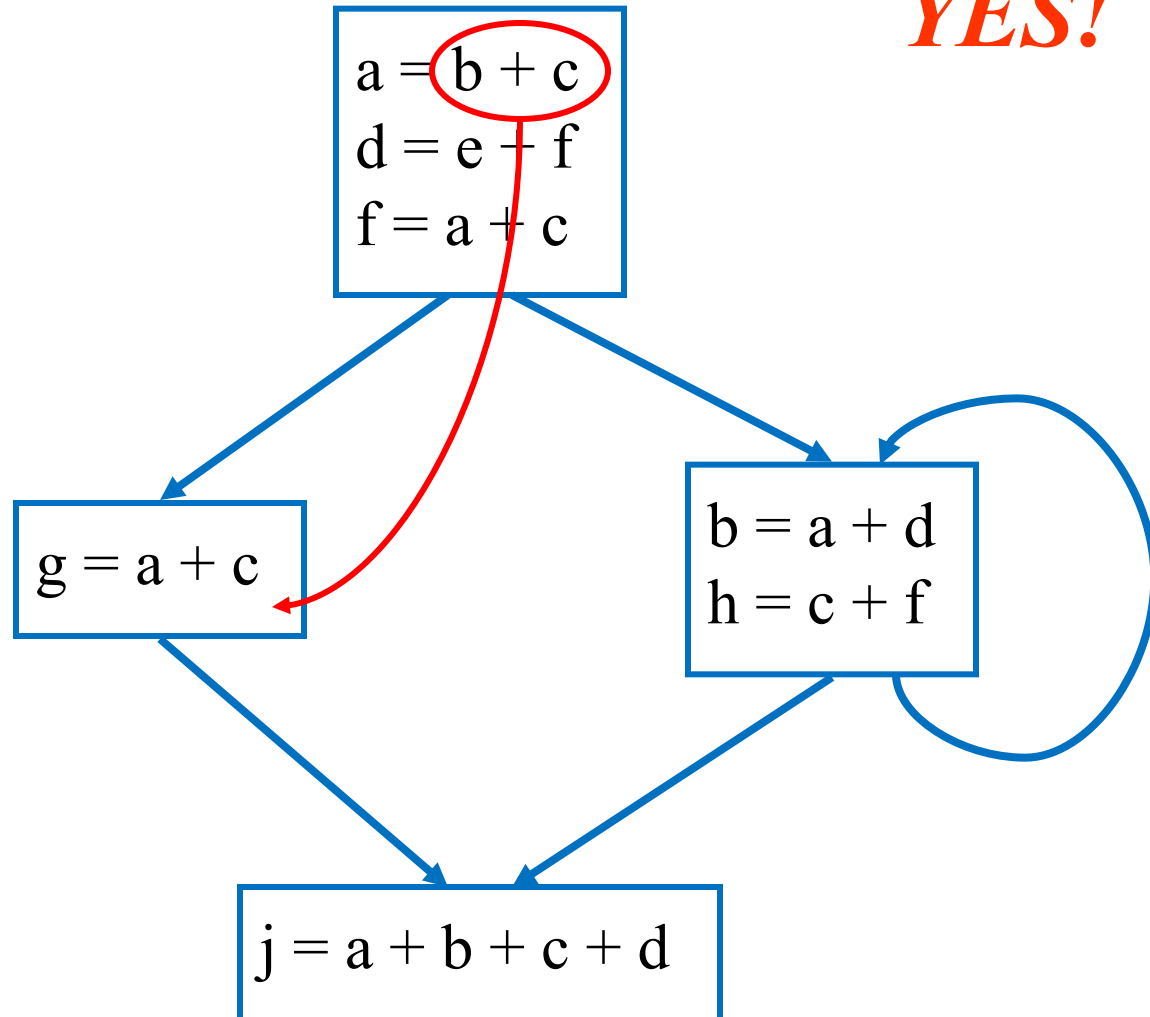


# Is the Expression Available?



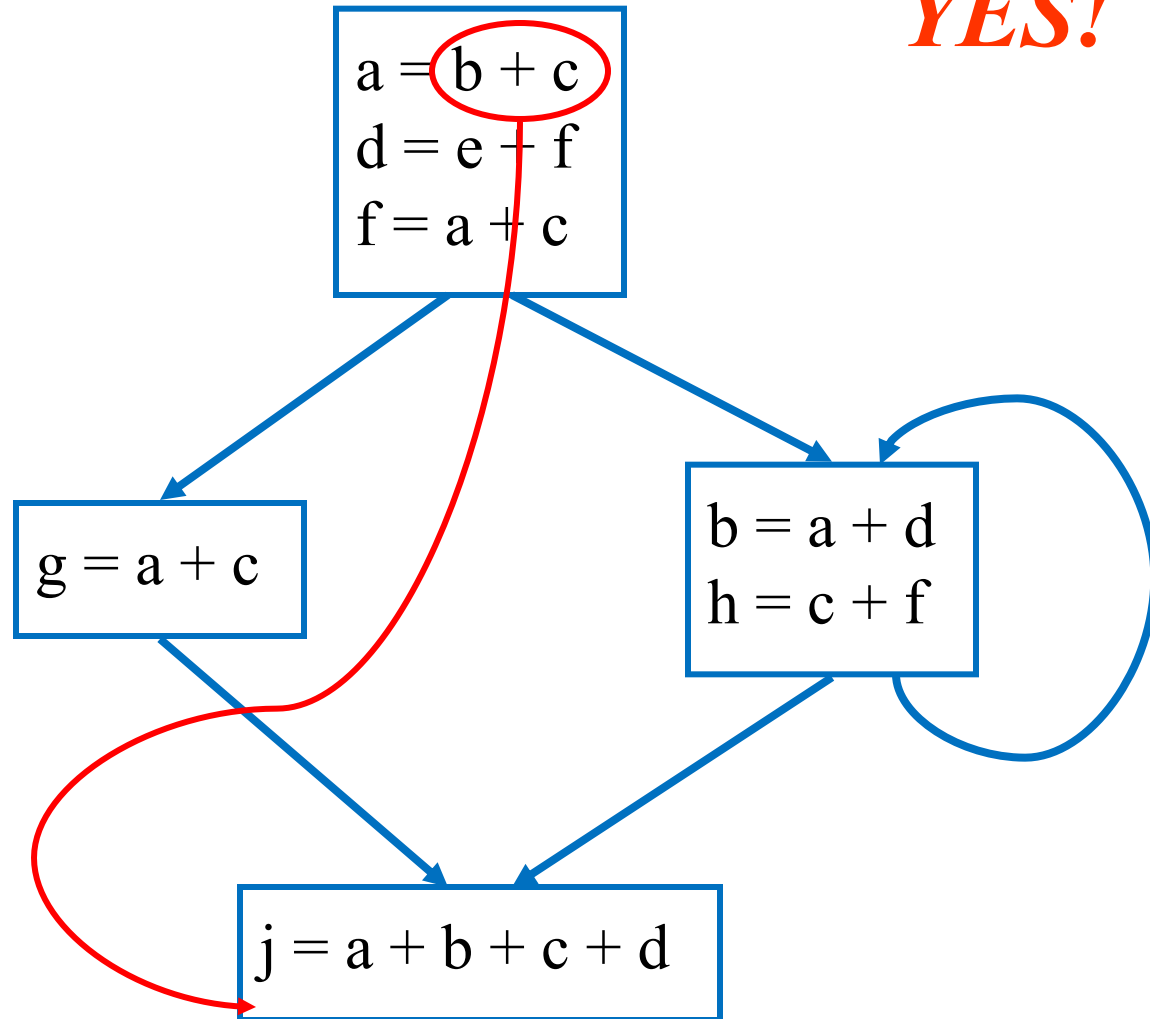
# Is the Expression Available?

***YES!***

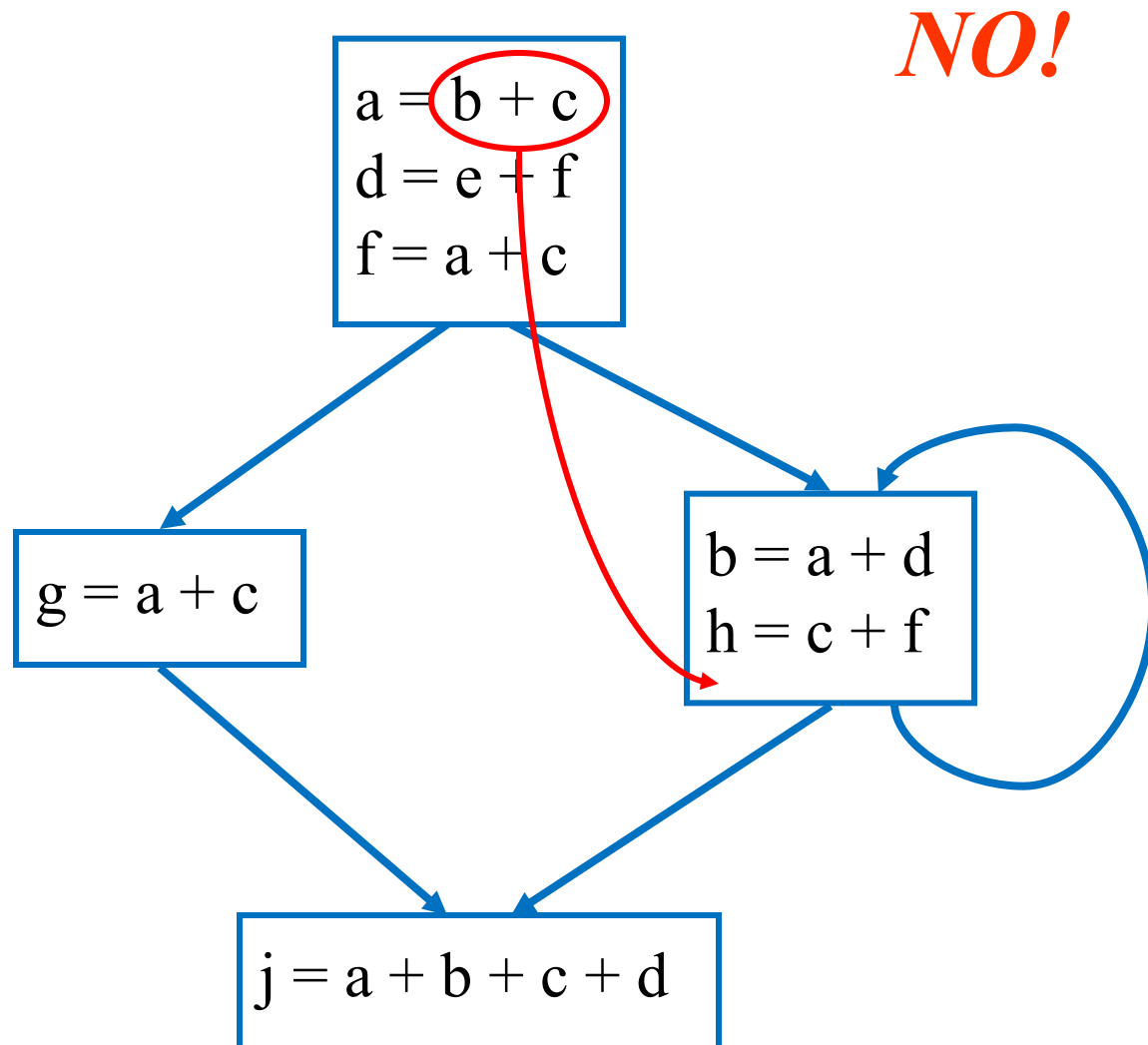


# Is the Expression Available?

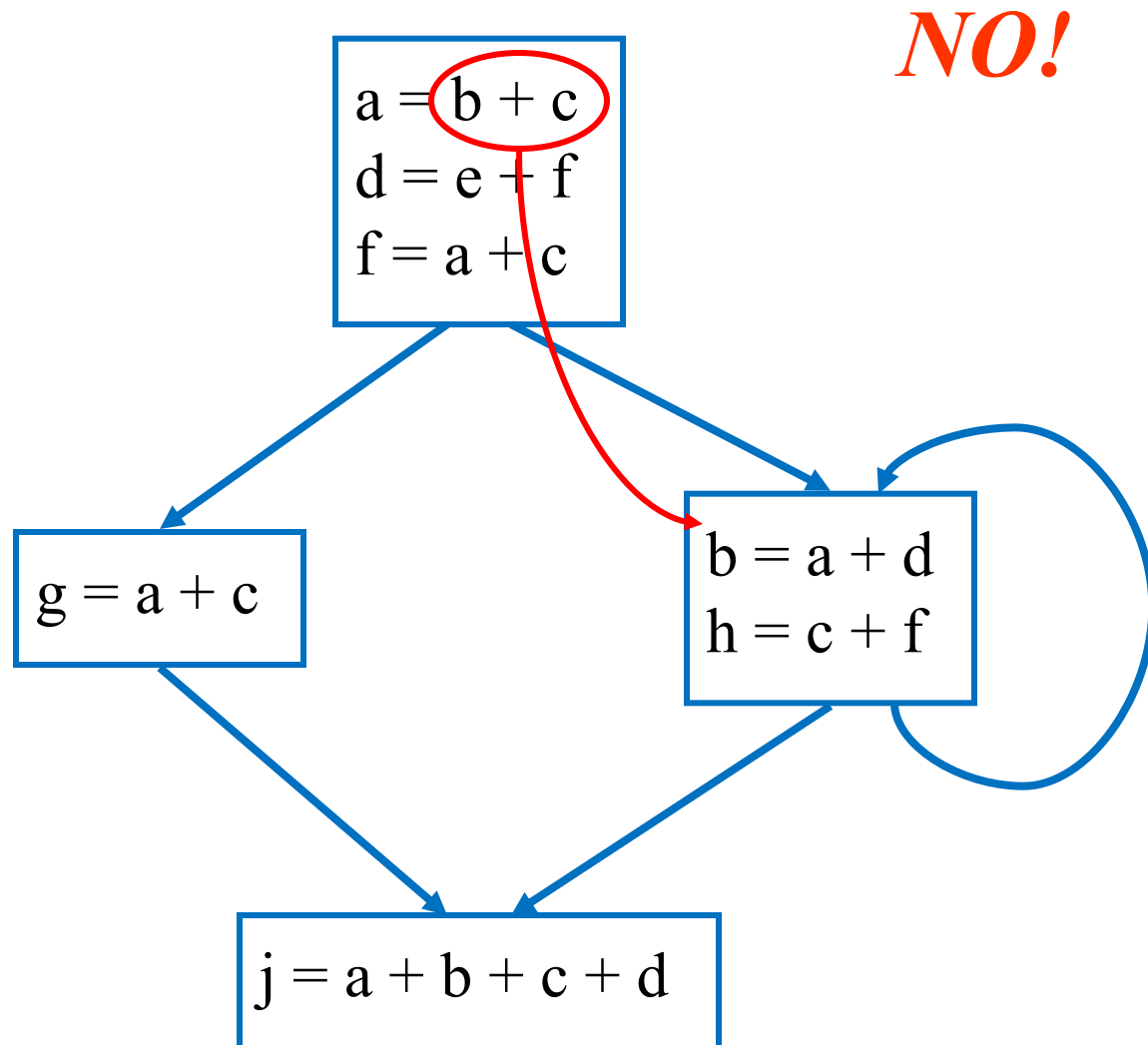
***YES!***



# Is the Expression Available?

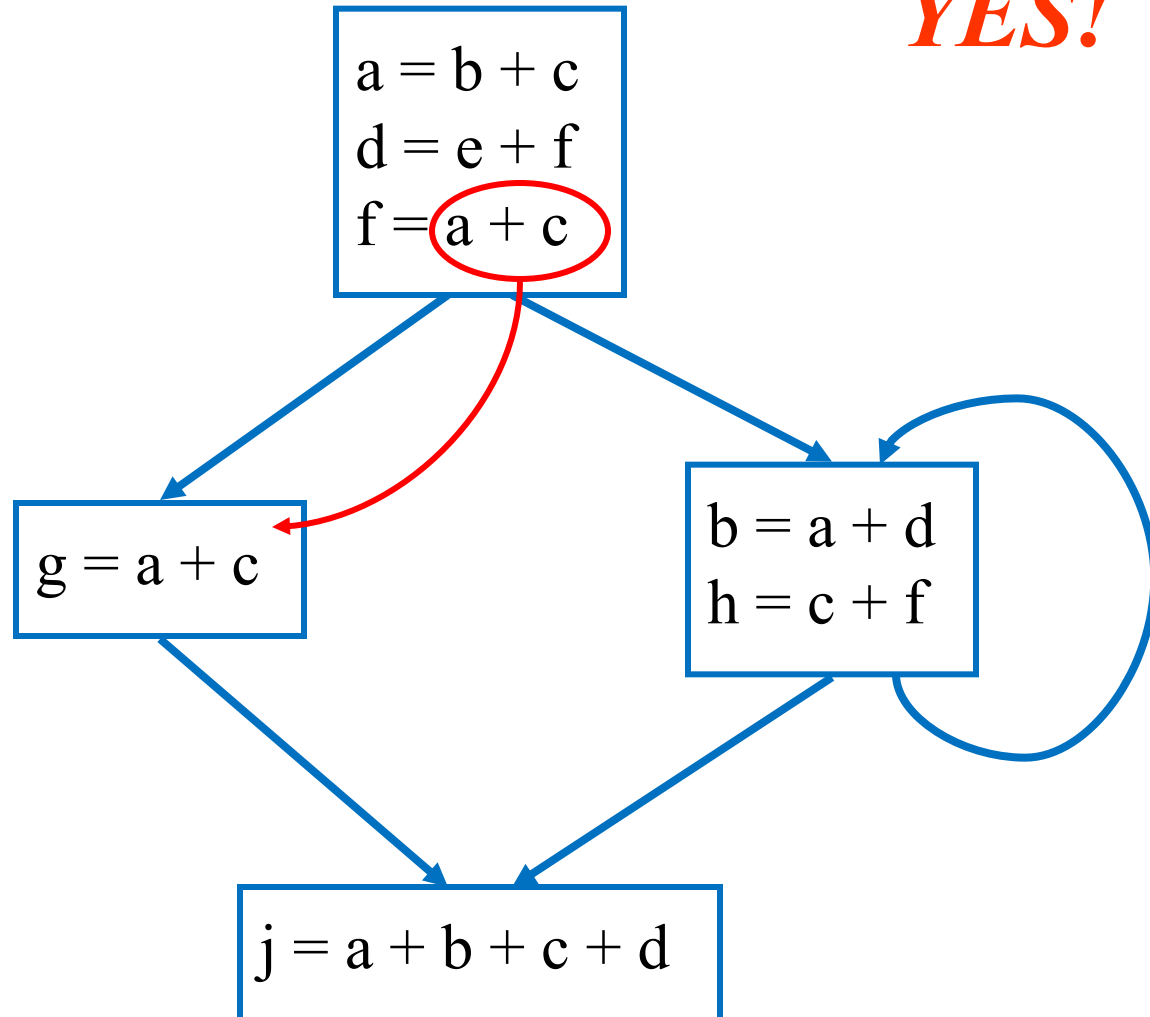


# Is the Expression Available?



# Is the Expression Available?

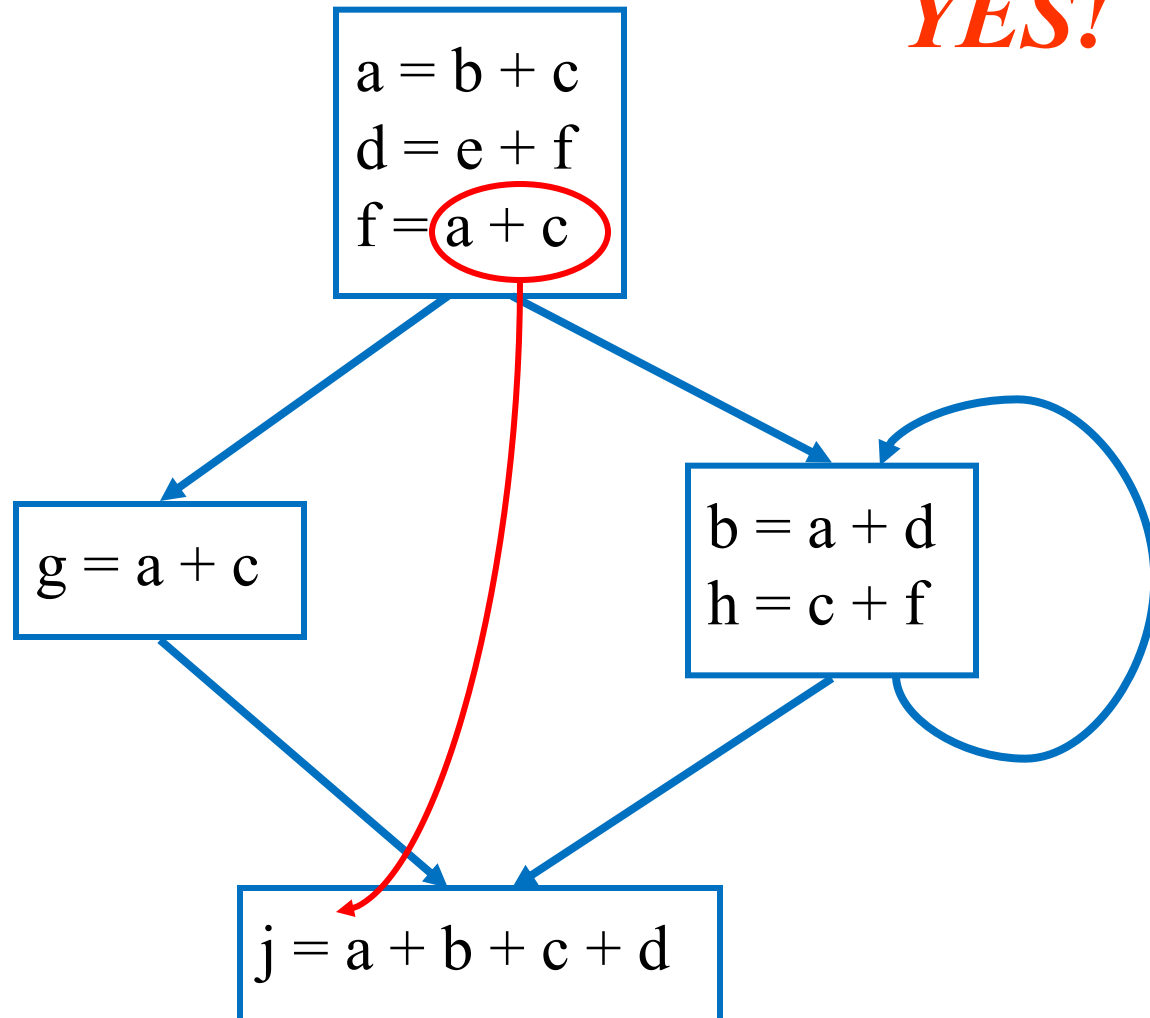
***YES!***



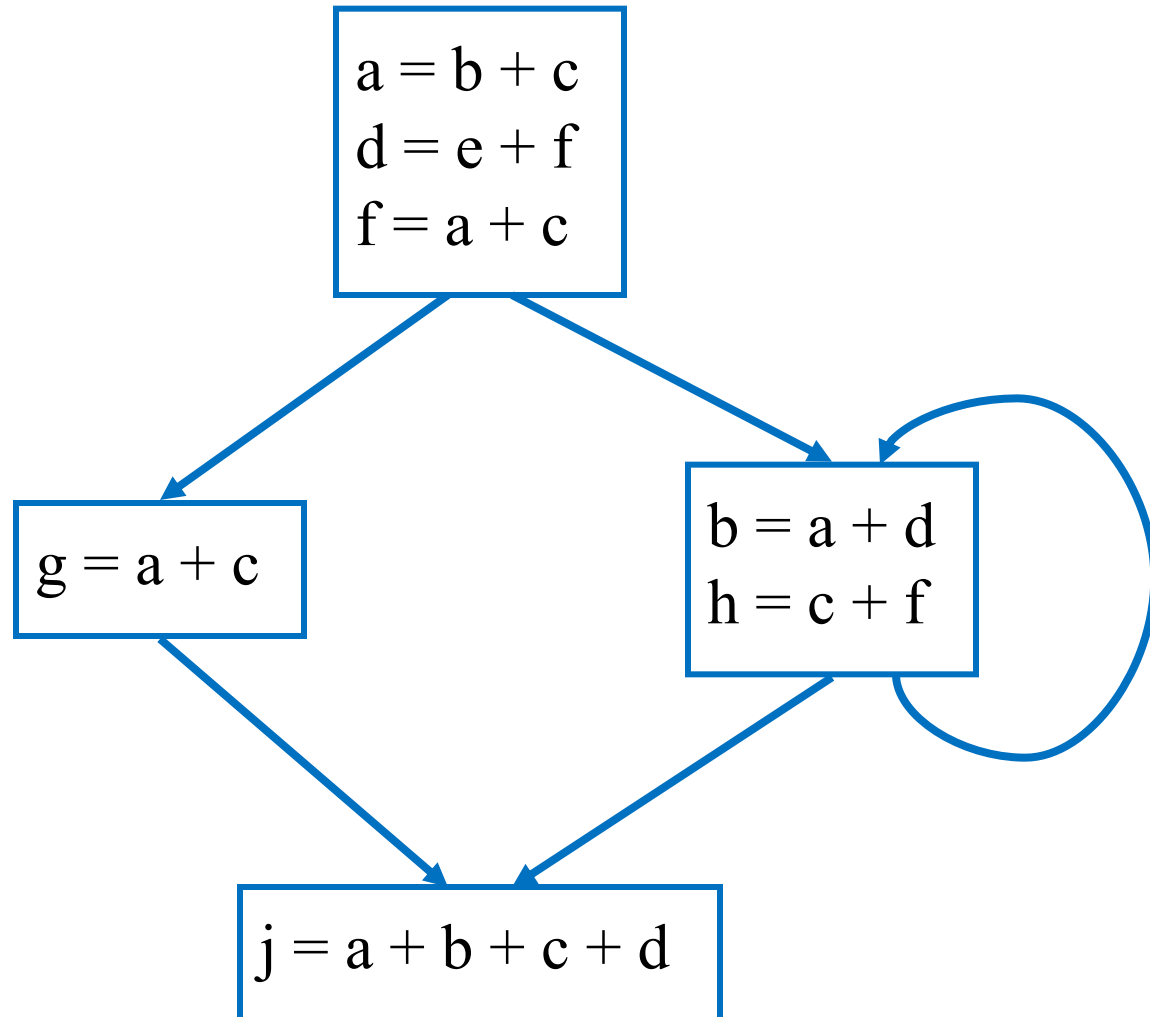


# Is the Expression Available?

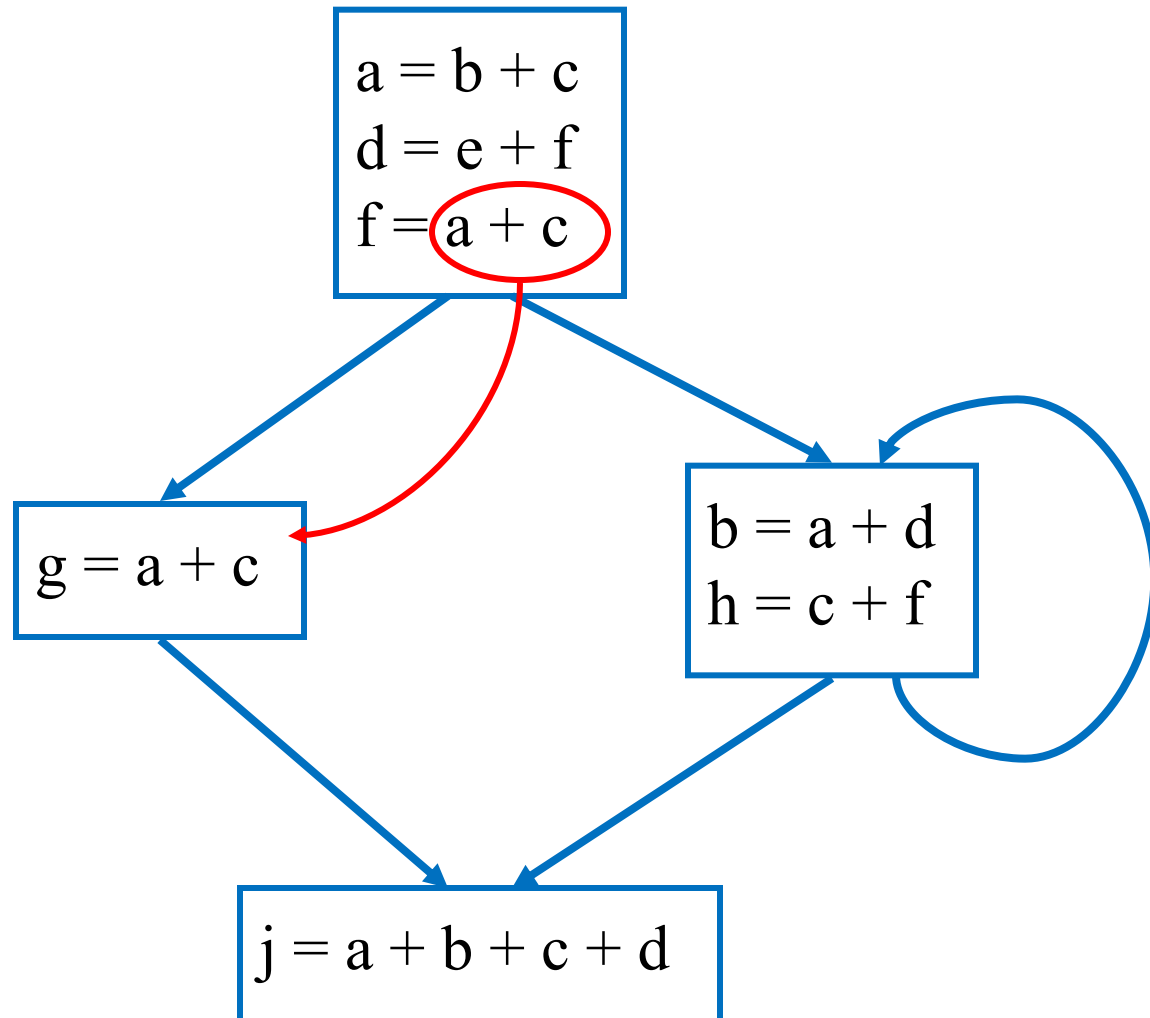
***YES!***



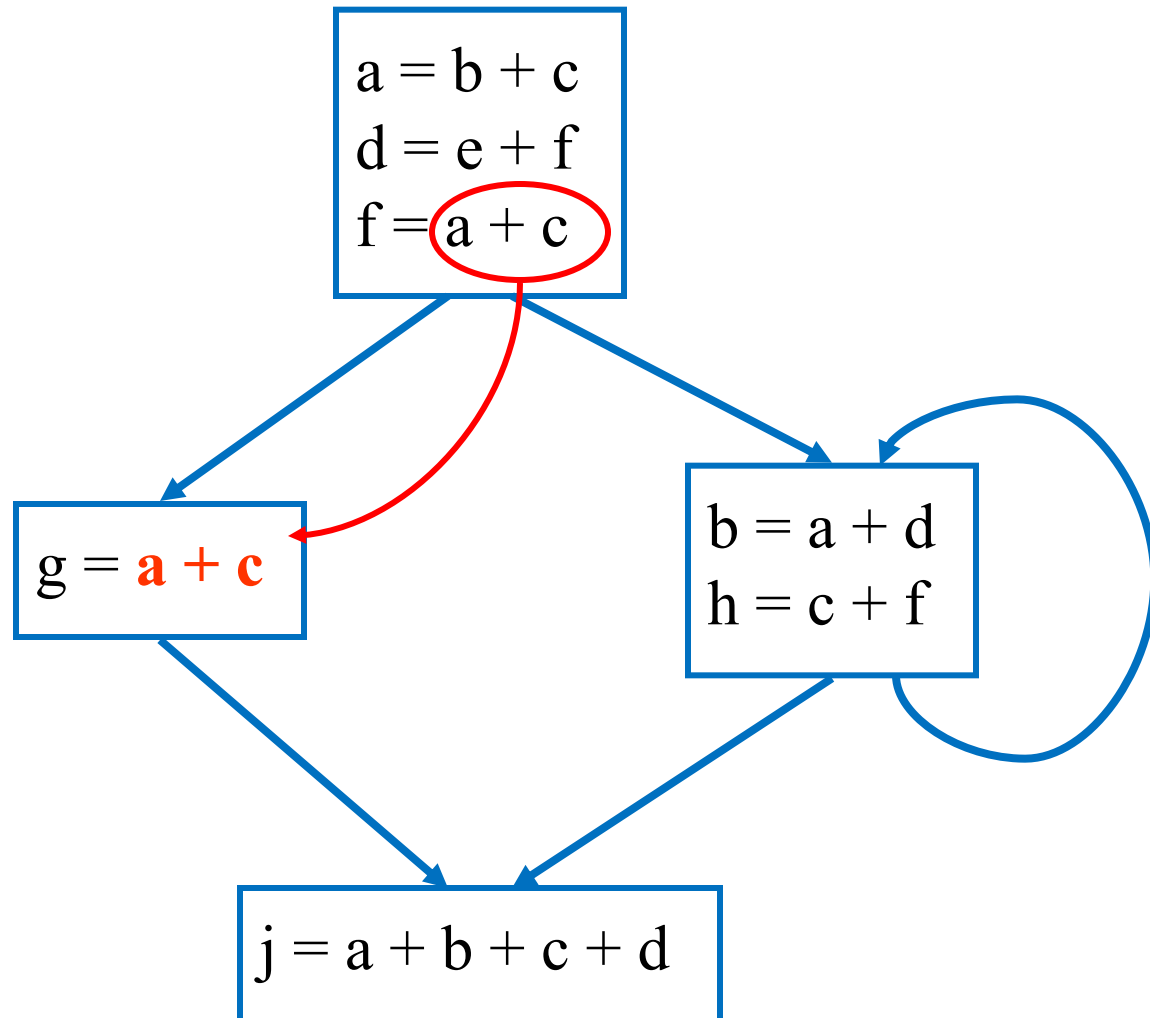
# Use of Available Expressions



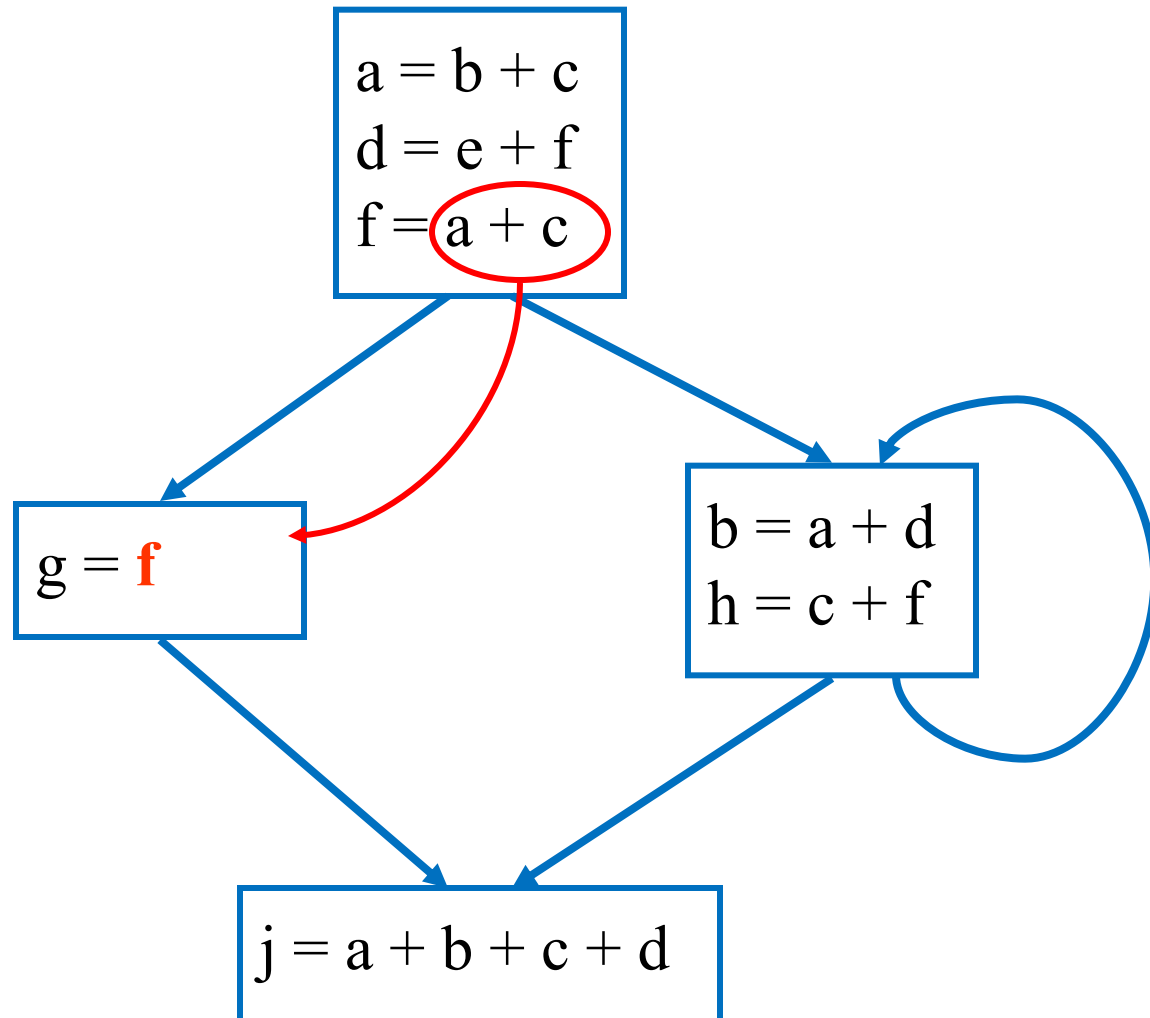
# Use of Available Expressions



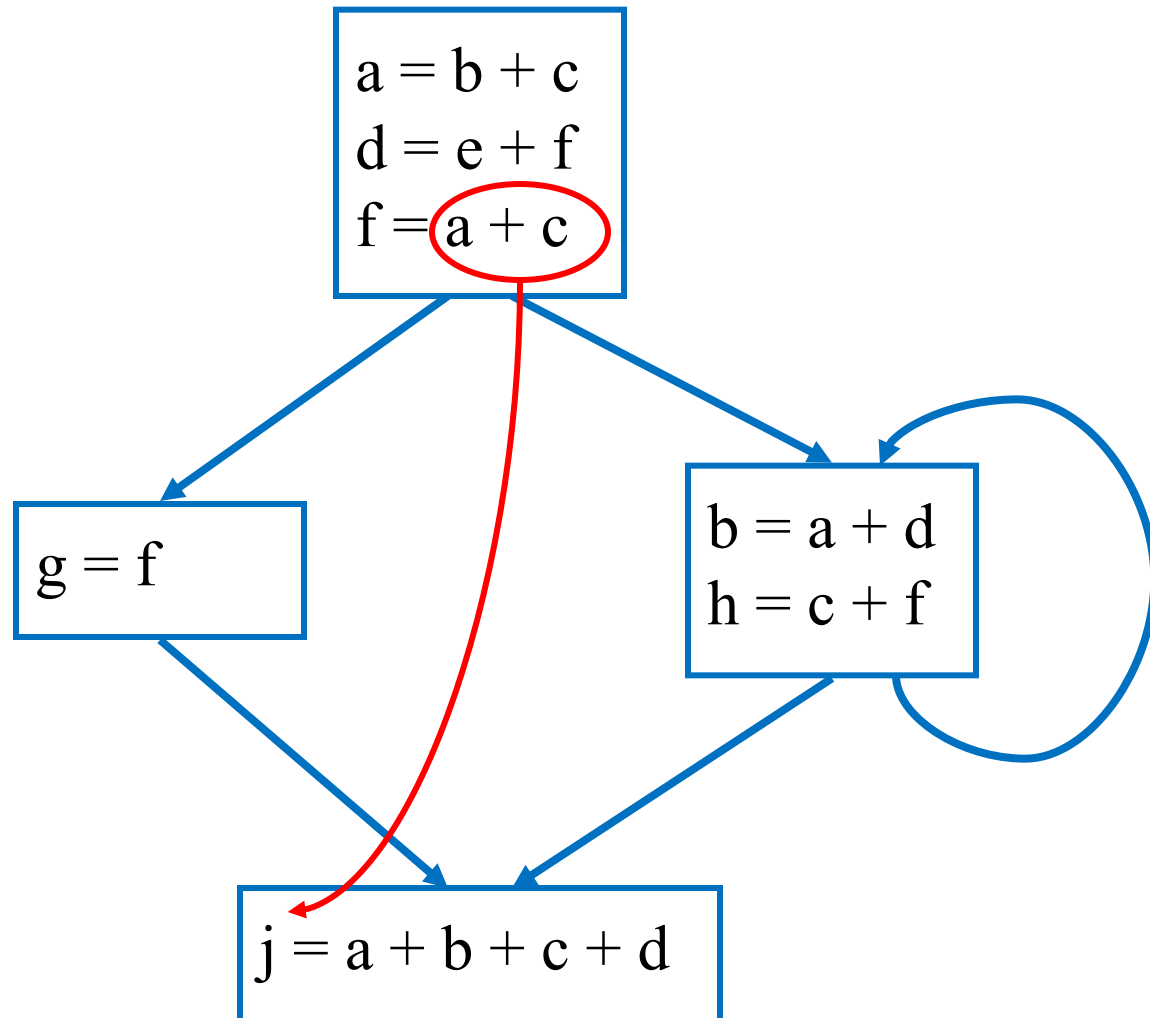
# Use of Available Expressions



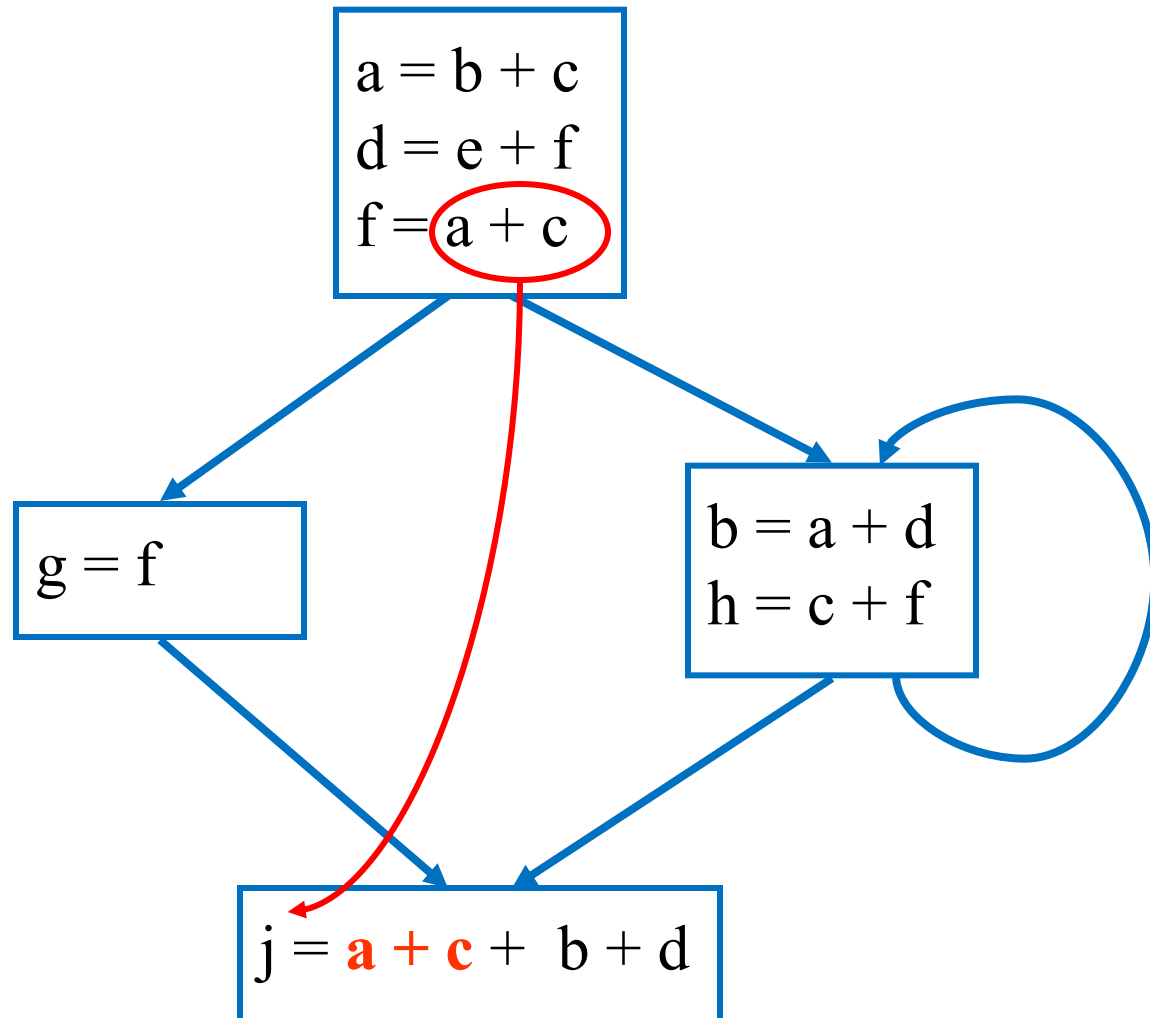
# Use of Available Expressions



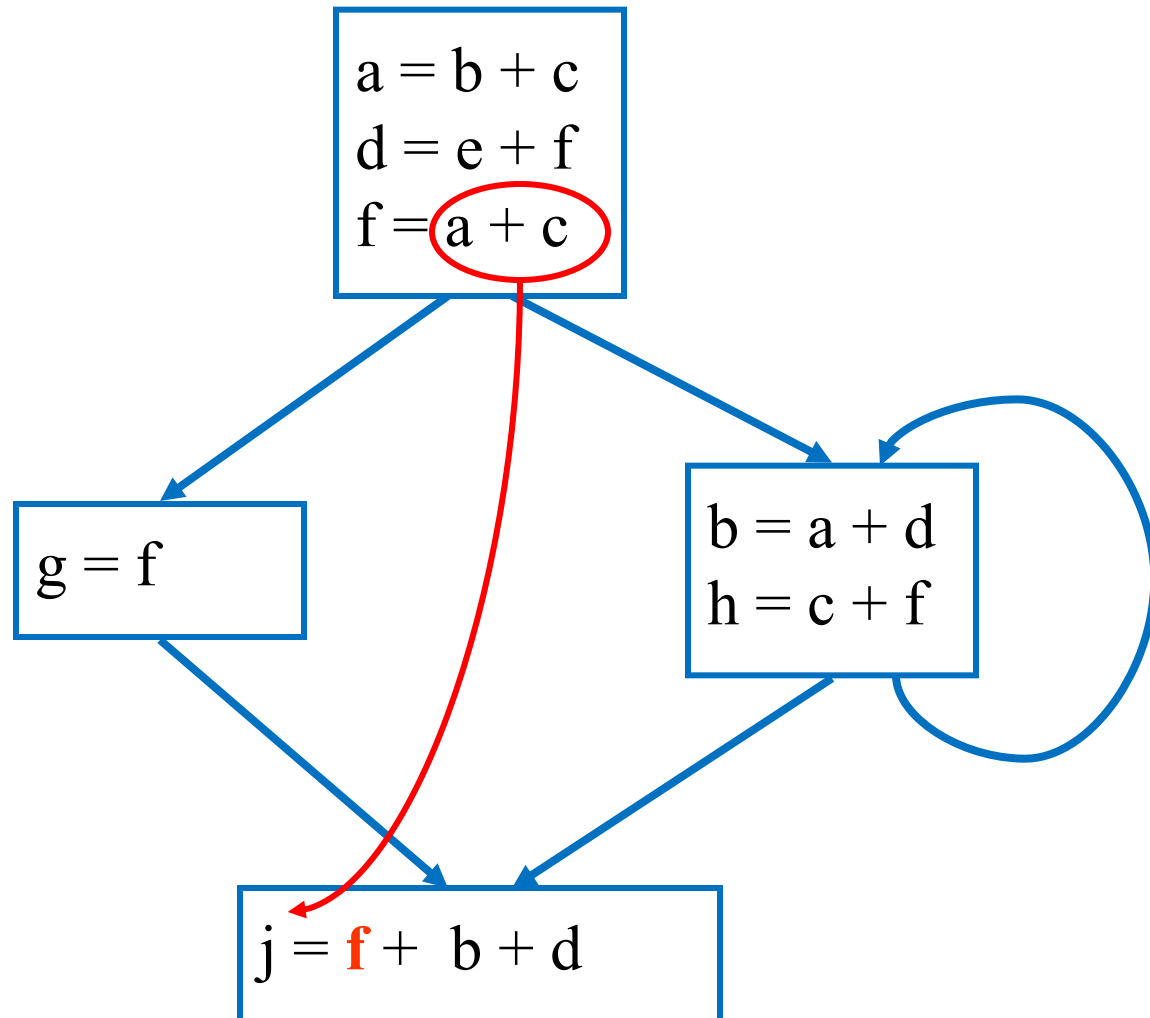
# Use of Available Expressions



# Use of Available Expressions

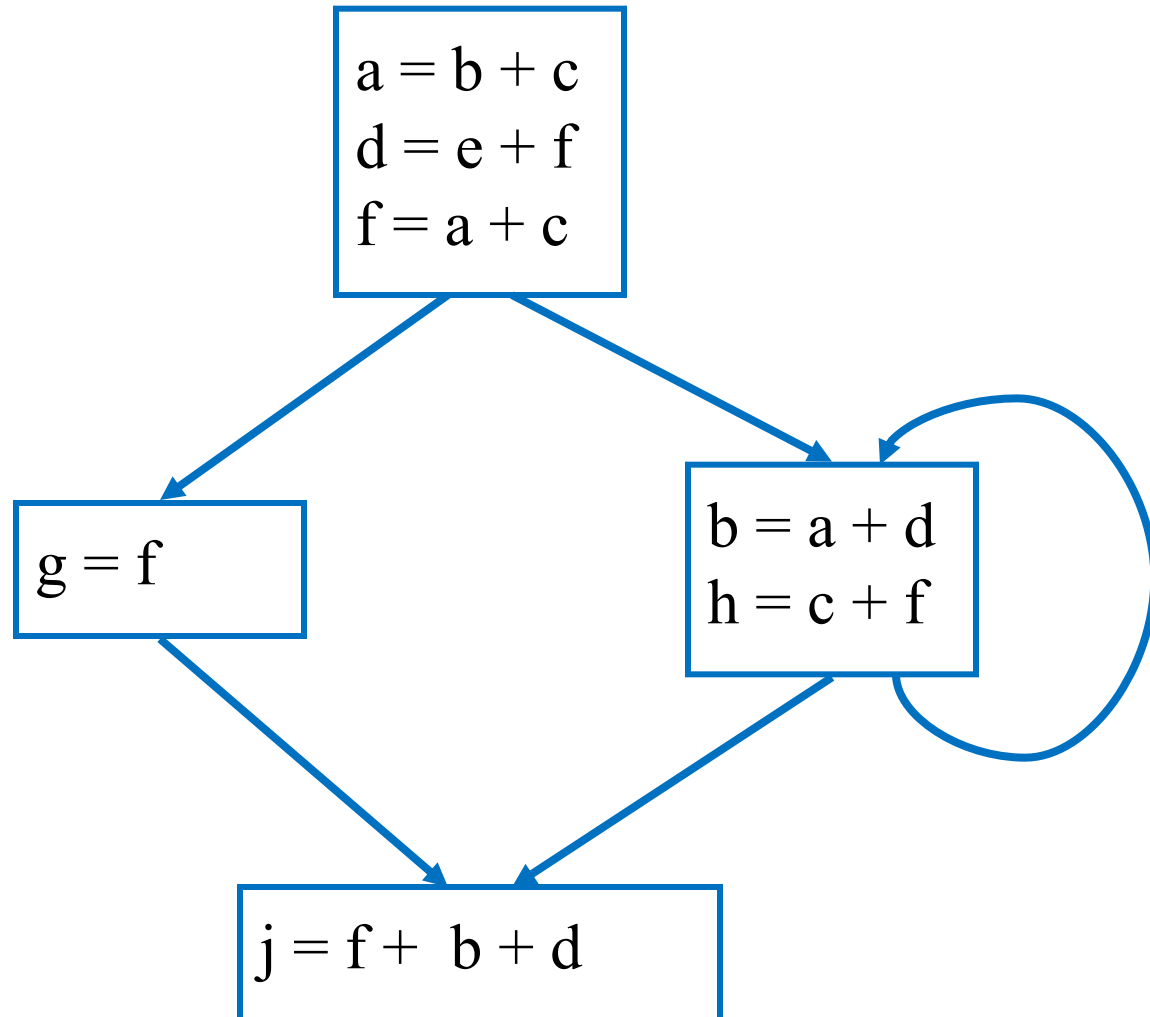


# Use of Available Expressions





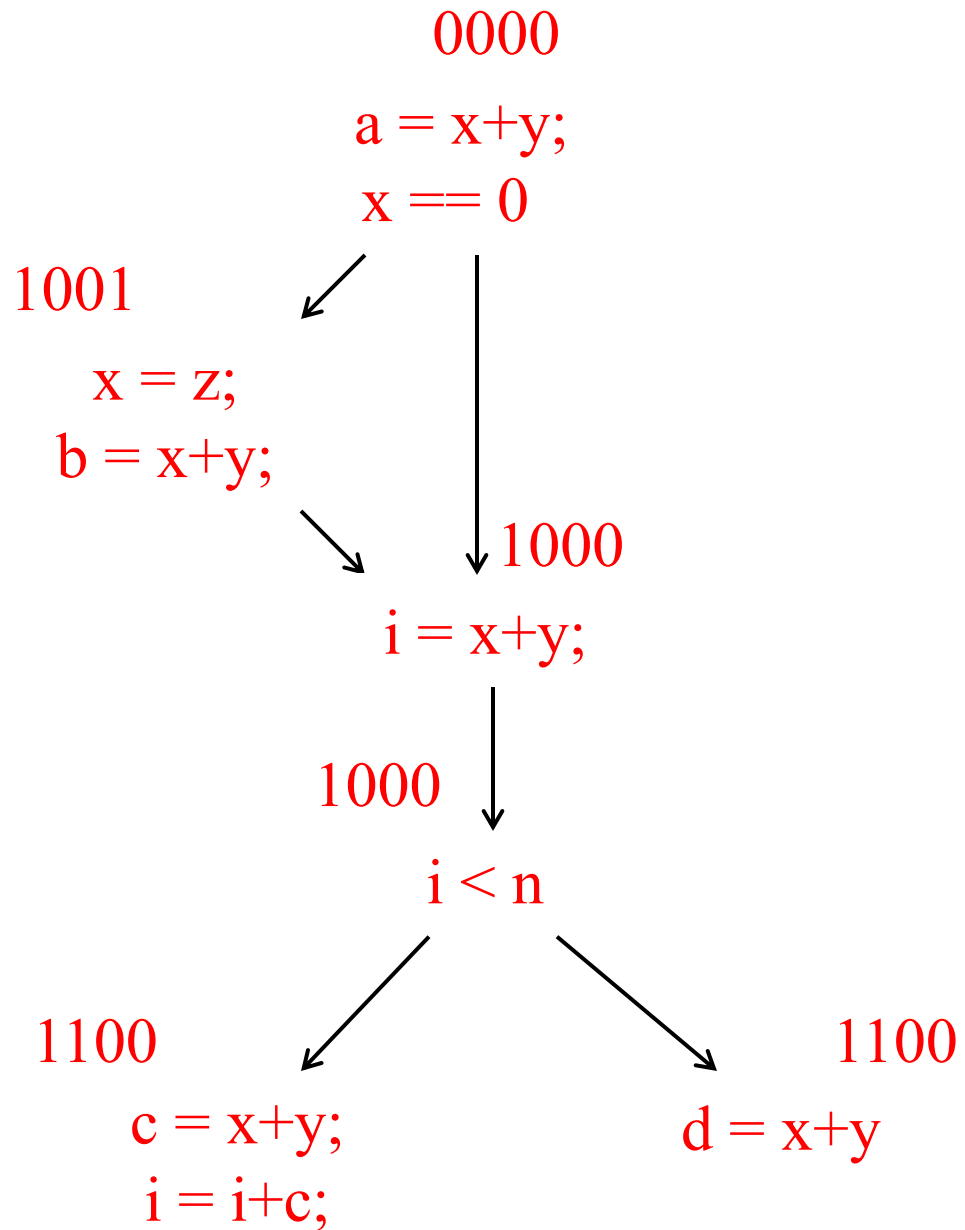
# Use of Available Expressions



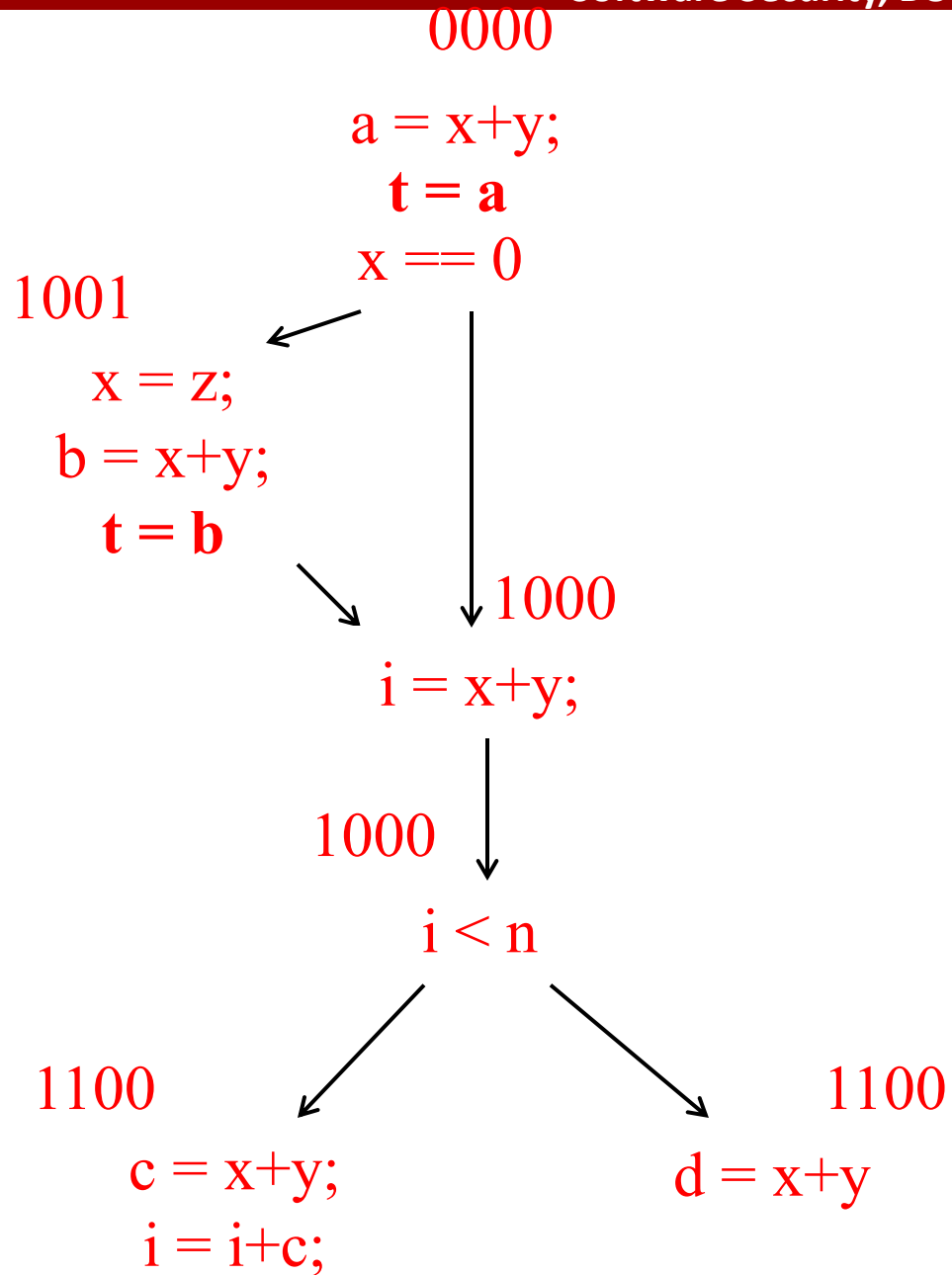
# Computing Available Expressions

- Represent sets of expressions using bit vectors
- Each expression corresponds to a bit
- Run dataflow algorithm **similar to** reaching definitions
- **Big difference**
  - definition reaches a basic block if it comes from **ANY** predecessor in CFG
  - expression is available at a basic block only if it is available from **ALL** predecessors in CFG

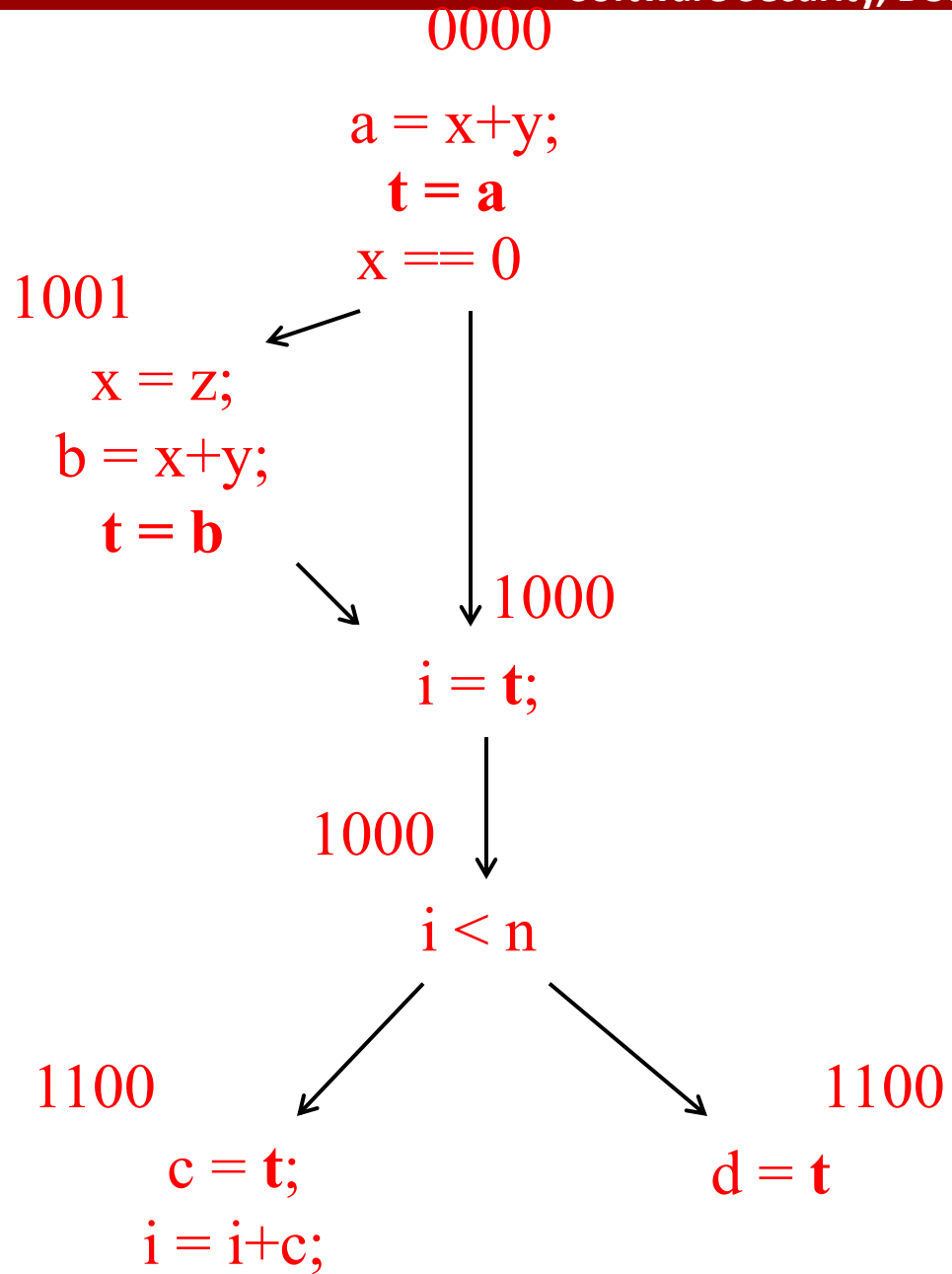
## Expressions

1:  $x+y$ 2:  $i < n$ 3:  $i+c$ 4:  $x==0$ 

## Expressions

1:  $x+y$ 2:  $i < n$ 3:  $i+c$ 4:  $x==0$ 

## Expressions

1:  $x+y$ 2:  $i < n$ 3:  $i+c$ 4:  $x == 0$ 

# Formalizing Analysis

## ■ Each basic block has

- IN - set of expressions available at start of block
- OUT - set of expressions available at end of block
- GEN - set of expressions computed in block (and not killed later)
- KILL - set of expressions killed in in block (and not re-computed later)

■  $\text{GEN}[x = z; b = x+y] = 1000$

■  $\text{KILL}[x = z; b = x+y] = 0001$

■ Compiler scans each basic block to derive GEN and KILL sets

# Dataflow Equations

- $IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$ 
  - where  $b_1, \dots, b_n$  are predecessors of  $b$  in CFG
- $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- $IN[entry] = 0000$
- Result: system of equations

# Solving Equations

- Use fixed point algorithm
- $IN[entry] = 0000$
- Initialize  $OUT[b] = 1111$
- Repeatedly apply equations
  - $IN[b] = OUT[b_1] \cap \dots \cap OUT[b_n]$
  - $OUT[b] = (IN[b] - KILL[b]) \cup GEN[b]$
- Use a worklist algorithm to reach fixed point



# Available Expressions Algorithm

```
for all nodes n in N
    OUT[n] = E;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry }; // N = all nodes in graph

while (Changed != emptyset)
    choose a node n in Changed;
    Changed = Changed - { n };

    IN[n] = E; // E is set of all expressions
    for all nodes p in predecessors(n)
        IN[n] = IN[n]  $\cap$  OUT[p];

    OUT[n] = GEN[n]  $\cup$  (IN[n] - KILL[n]);

    if (OUT[n] changed)
        for all nodes s in successors(n)
            Changed = Changed  $\cup$  { s };
```

# Questions

- Does algorithm always halt?
- If expression is available in some execution, is it always marked as available in analysis?
- If expression is not available in some execution, can it be marked as available in analysis?

# General Correctness

- **Concept in actual program execution**
  - Reaching definition: definition  $D$ , execution  $E$  at program point  $P$
  - Available expression: expression  $X$ , execution  $E$  at program point  $P$
- **Analysis reasons about all possible executions**
- **For all executions  $E$  at program point  $P$ ,**
  - if a definition  $D$  reaches  $P$  in  $E$
  - then  $D$  is in the set of reaching definitions at  $P$  from analysis
- **Other way around**
  - if  $D$  is not in the set of reaching definitions at  $P$  from analysis
  - then  $D$  never reaches  $P$  in any execution  $E$
- **For all executions  $E$  at program point  $P$ ,**
  - if an expression  $X$  is in set of available expressions at  $P$  from analysis
  - then  $X$  is available in  $E$  at  $P$

# Duality In Two Algorithms

两种算法的对偶

## ■ Reaching definitions

- Confluence operation is **set union**
- $OUT[b]$  initialized to empty set

## ■ Available expressions

- Confluence operation is **set intersection**
- $OUT[b]$  initialized to set of available expressions

## ■ General framework for dataflow algorithms.

## ■ Build parameterized dataflow analyzer once, use for all dataflow problems

# Data flow analysis

- Reaching Definitions
- Available Expressions
- Live Variables

# Live Variable Analysis ( 活跃变量分析 )

- **A variable  $v$  is live at point  $p$  if**
  - $v$  is used along some path starting at  $p$ , and
  - no definition of  $v$  along the path before the use.
- **When is a variable  $v$  dead at point  $p$ ?**
  - No use of  $v$  on any path from  $p$  to exit node, or
  - If all paths from  $p$  redefine  $v$  before using  $v$ .

# What Use is Liveness Information?

## ■ Register allocation.

- If a variable is dead, can reassign its register

## ■ Dead code elimination.

- Eliminate assignments to variables not read later.
- But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
- Can eliminate other dead assignments.
- Handle by making all externally visible variables live on exit from CFG

活跃变量分析的用途之一是为基本块进行寄存器分配。在一个值被计算并且保存到一个寄存器中后，它很可能会在基本快中使用。

如果它在基本块得结尾处是死的，就不必在结尾处保存这个值。

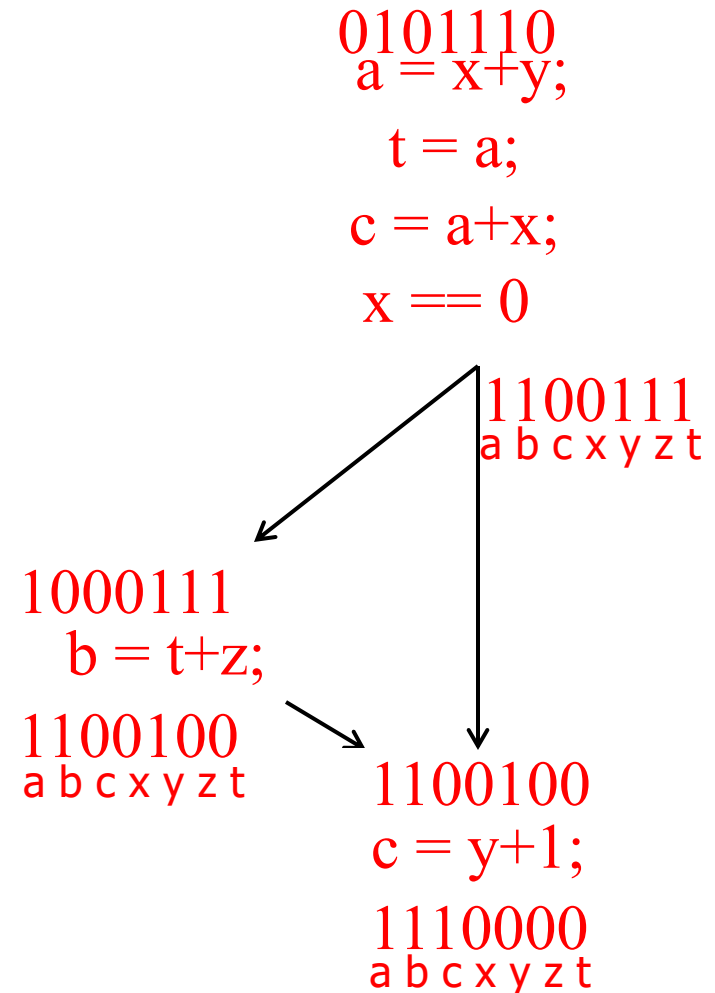
# Conceptual Idea of Analysis/分析的概念

- Simulate execution
- But start from exit and go backwards in CFG
- Compute liveness information from end to beginning of basic blocks



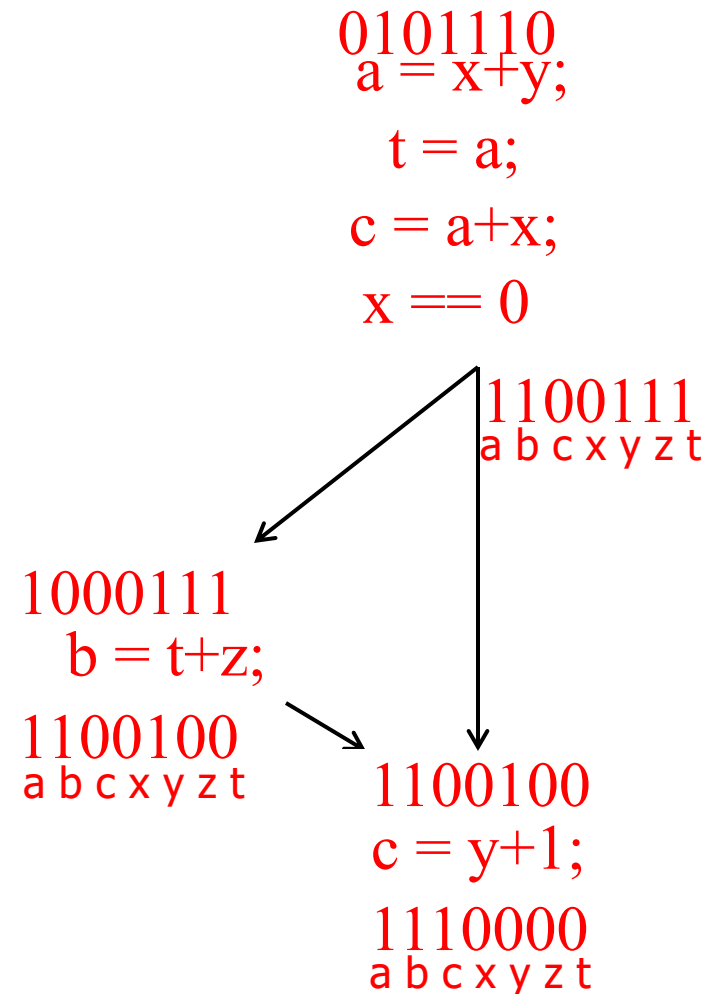
# Liveness Example

- Assume a,b,c visible outside method
- So are live on exit
- Assume x,y,z,t not visible
- Represent Liveness Using Bit Vector
  - order is abcxyz t



# Dead Code Elimination

- Assume a,b,c visible outside method
- So are live on exit
- Assume x,y,z,t not visible
- Represent Liveness Using Bit Vector
  - order is abcxyz t



# Formalizing Analysis

- **Each basic block has**
  - IN - set of variables live at start of block
  - OUT - set of variables live at end of block
  - USE - set of variables with upwards exposed uses in block (use prior to definition)
  - DEF - set of variables defined in block prior to use
- **$\text{USE}[x = z; x = x+1;] = \{ z \}$  (x not in USE)**
- **$\text{DEF}[x = z; x = x+1; y = 1;] = \{x, y\}$**
- **Compiler scans each basic block to derive USE and DEF sets**

# Algorithm

```
for all nodes  $n$  in  $N - \{ \text{Exit} \}$ 
     $\text{IN}[n] = \text{emptyset};$ 
 $\text{OUT}[\text{Exit}] = \text{emptyset};$ 
 $\text{IN}[\text{Exit}] = \text{use}[\text{Exit}];$ 
 $\text{Changed} = N - \{ \text{Exit} \};$ 

while ( $\text{Changed} \neq \text{emptyset}$ )
    choose a node  $n$  in  $\text{Changed}$ ;
     $\text{Changed} = \text{Changed} - \{ n \};$ 

     $\text{OUT}[n] = \text{emptyset};$ 
    for all nodes  $s$  in  $\text{successors}(n)$ 
         $\text{OUT}[n] = \text{OUT}[n] \cup \text{IN}[p];$ 

     $\text{IN}[n] = \text{use}[n] \cup (\text{out}[n] - \text{def}[n]);$ 

    if ( $\text{IN}[n]$  changed)
        for all nodes  $p$  in  $\text{predecessors}(n)$ 
             $\text{Changed} = \text{Changed} \cup \{ p \};$ 
```

# Similar to Other Dataflow Algorithms

- **Backward analysis, not forward**
- **Still have transfer functions**
- **Still have confluence operators**
- **Can generalize framework to work for both forwards and backwards analyses**

# Comparison

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$OUT[ENTRY] = \emptyset$	$IN[EXIT] = \emptyset$	$OUT[ENTRY] = \emptyset$
Meet ( $\wedge$ )	$\cup$	$\cup$	$\cap$
Equations	$OUT[B] = f_B(IN[B])$ $IN[B] =$ $\bigwedge_{P, pred(B)} OUT[P]$	$IN[B] = f_B(OUT[B])$ $OUT[B] =$ $\bigwedge_{S, succ(B)} IN[S]$	$OUT[B] = f_B(IN[B])$ $IN[B] =$ $\bigwedge_{P, pred(B)} OUT[P]$
Initialize	$OUT[B] = \emptyset$	$IN[B] = \emptyset$	$OUT[B] = U$

# Summarize

## ■ Dataflow Analysis

- Control flow graph
- $IN[b]$ ,  $OUT[b]$ , transfer functions, join points

## ■ Paired analyses and transformations

- Reaching definitions/constant propagation
- Available expressions/common sub-expression elimination
- Live-variable analysis/Dead code elimination