Principles of Program Analysis

Julia Rubin

Program Analysis: Reasoning About Code

- The process of automatically analyzing the behavior of programs
  - Input: the code of the program
  - Output: code or interesting facts about the code

- Examples?

Major Application Areas

- Program correctness:
  - code inspection, style checkers, security threats, validation of correctness, robustness
- Program optimization:
  - improving the program's performance while reducing its resource usage
- Program understanding, validation, and repair:
  - explaining code, identifying and automatically fixing error

Types of Analysis

	Static Analysis (without running the code)	Dynamic Analysis (at runtime)
White Box (code internals)		– Unit testing – Mock testing – Debugging
Black Box (input-output)		– Integration testing – User acceptance testing – Profiling – Monitoring

Types of Analysis

	Static Analysis (without running the code)	Dynamic Analysis (at runtime)
White Box (code internals)	– Style checks and linters	– Unit testing – Mock testing – Debugging
Black Box (input-output)		– Integration testing – User acceptance testing – Profiling – Monitoring

– Security checks
– *Program understanding*
– *Testing and debugging aid*
– …

Why Program Analysis?

- To reduce development costs
  - measured in hundreds to thousands of dollars per delivered LOC
    - before Generative AI
  - validation and verification is usually 50% of this cost
    - Became even more major with Generative AI
- Maintenance costs
  - 2-3 times as much as development

Top Software Companies Invest in Proper Static Analysis

COMMUNICATIONS OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARC | PRACTICE | CAREER | ARCHIVE | VIDEOS

Home / Magazine Archive / April 2018 (Vol. 61, No. 4) / Lessons from Building Static Analysis Tools at Google / Full Text

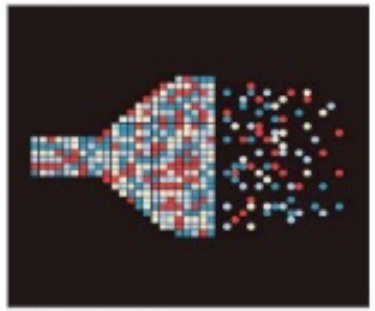
CONTRIBUTED ARTICLES

Lessons from Building Static Analysis Tools at Google

By Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, Ciera Jaspan
Communications of the ACM, April 2018, Vol. 61 No. 4, Pages 58-66
10.1145/3188720
Comments

SIGN IN for Full Access

VIEW AS: SHARE:



Credit: Igor Kisselev

Software bugs cost developers and soft
great deal of time and money. For exar
bug in a widely used SSL implementati
caused it to accept invalid SSL certifics
related to date formatting caused a lar
outage.[23] Such bugs are often staticall
are, in fact, obvious upon reading the
documentation yet still make it into pr

Back to Top

Key Insights

- Static analysis authors should focus on the developer and listen to their feedback.
- Careful developer workflow integration

SIGN IN

COMMUNICATIONS OF THE ACM

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARC | PRACTICE | CAREER | ARCHIVE | VIDEOS

Home / Magazine Archive / August 2019 (Vol. 62, No. 8) / Scaling Static Analyses at Facebook / Full Text

CONTRIBUTED ARTICLES

Scaling Static Analyses at Facebook

By Dino Distefano, Manuel Fähndrich, Francesco Logozzo, Peter W. O'Hearn
Communications of the ACM, August 2019, Vol. 62 No. 8, Pages 62-70
10.1145/3338112
Comments

VIEW AS: SHARE:

Static analysis tools are programs that examine, and attempt to draw conclusions about, the source of other programs without running them. At Facebook, we have been investing in advanced static analysis tools that employ reasoning techniques similar to those from program verification. The tools we describe in this article (Infer and Zoncolan) target issues related to crashes and to the security of our services, they perform sometimes complex reasoning spanning many procedures or files, and they are integrated into engineering workflows in a way that attempts to bring value while minimizing friction.

Back to Top

SIGN IN for Full Access

User Name

Password

» Forgot Password?

» Create an ACM Web Account

SIGN IN

ARTICLE CONTENTS:

Introduction
Key Insights
Context for Static Analysis at Facebook
Software Development at Facebook
Moving Fast with Infer

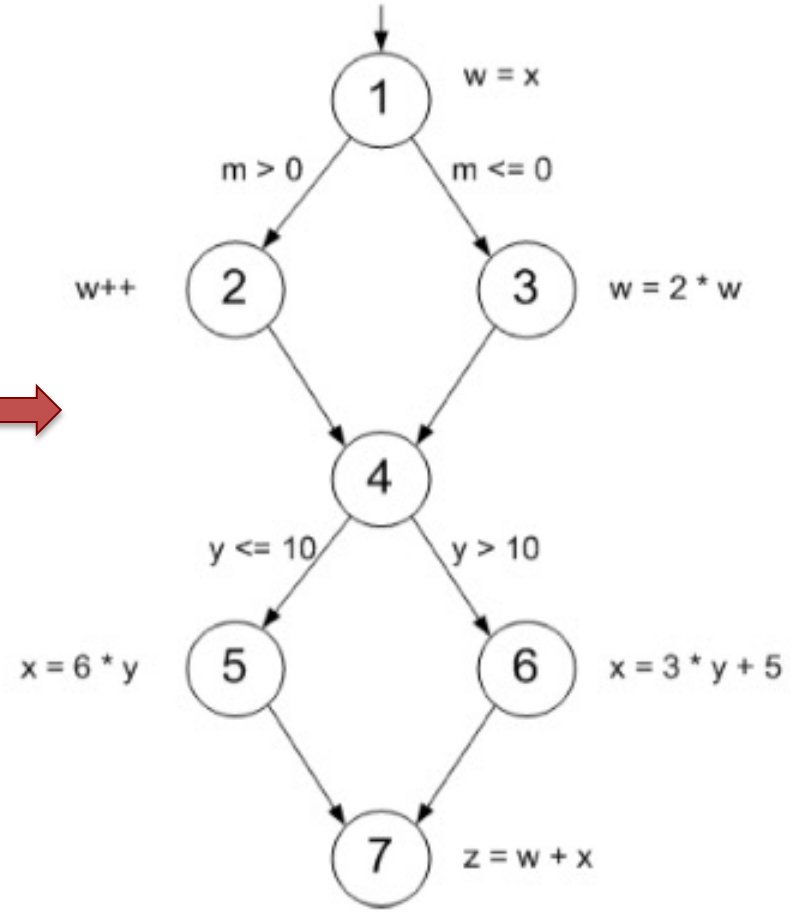
Agenda

- Basics of Analysis
  - Control flow
  - Call graph
  - Data Flow Analysis
  - Symbolic Execution (next class)

Models

Models

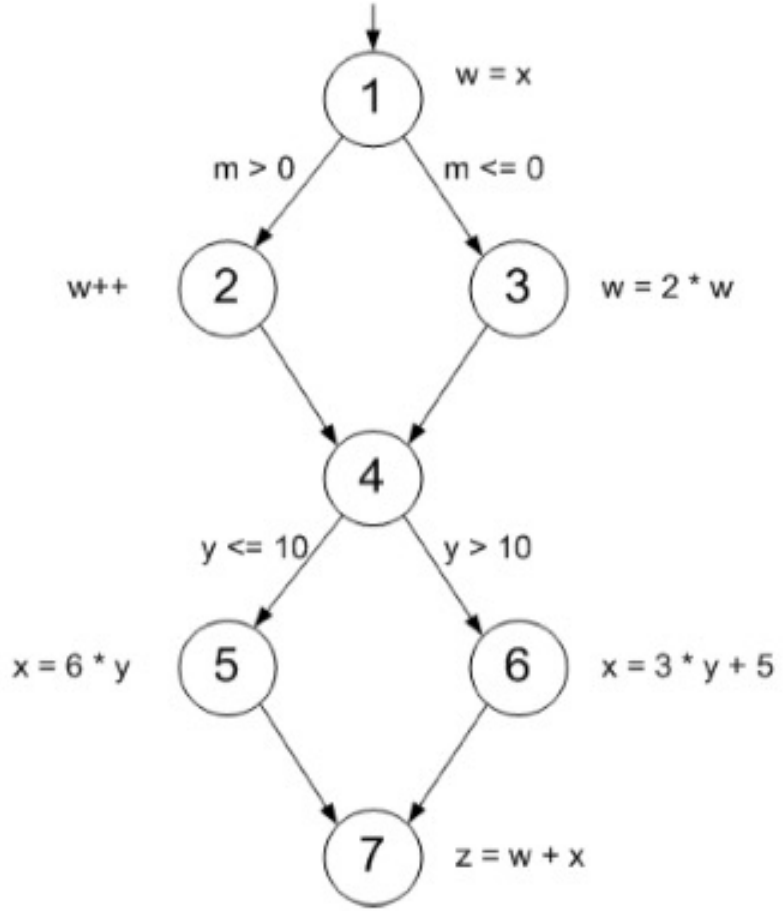
Models



Modeling Software

Graphs! E.g.,

- abstract syntax graphs
- control flow graphs
- call graphs
- reachability graphs
- …

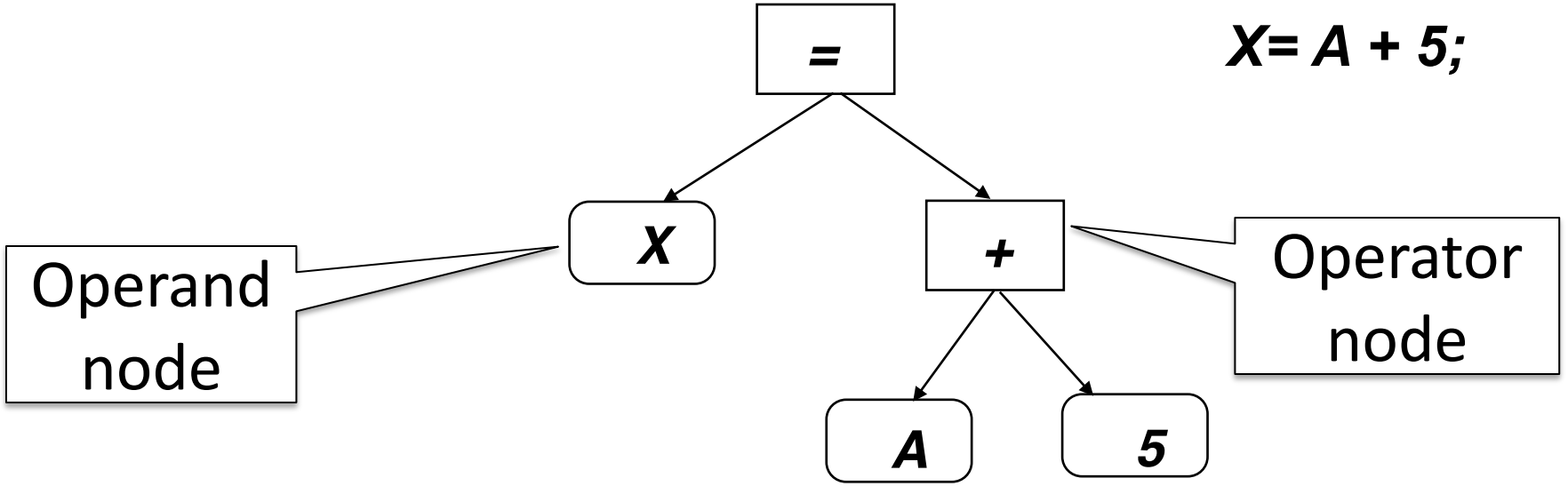


Graphs

- A graph, G = (N, E), is an ordered pair consisting of
  - a set of nodes N
  - a set of edges E = {(n_i, n_j)}
  - if the pairs in E are ordered, then G is called a *directed graph*
  - if not, it is called an *undirected graph*

Abstract Syntax Tree (AST)

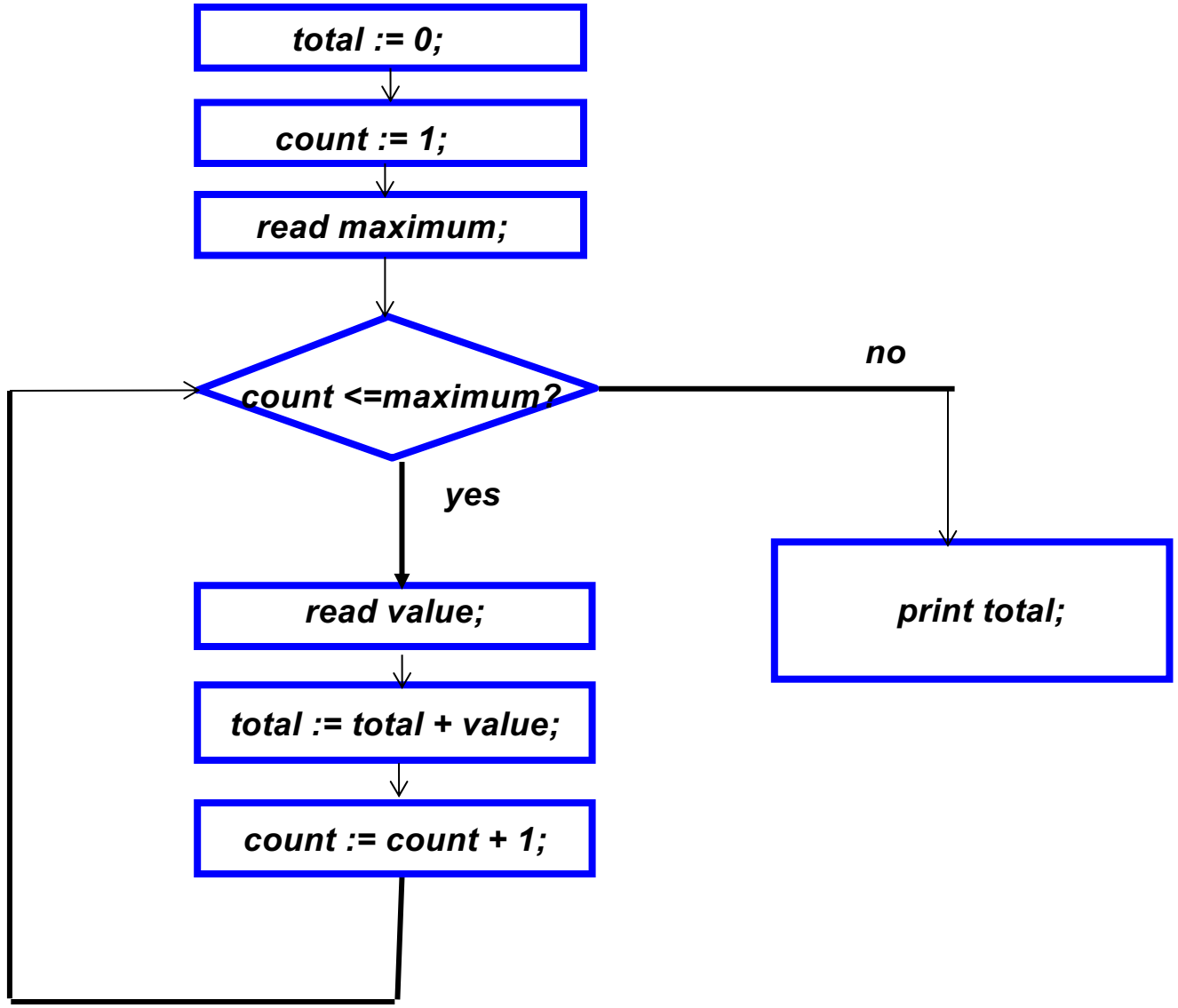
- A common form for representing expressions and program statements
- Two kinds of nodes: operator and operands
  - operator applied to N operands
- Each node denotes a construct occurring in the source code




```
while (b!=0) {
   if (a > b) {
       a = a – b;
    }
   else {
     b  = b – a;
   }
}
return a;
```

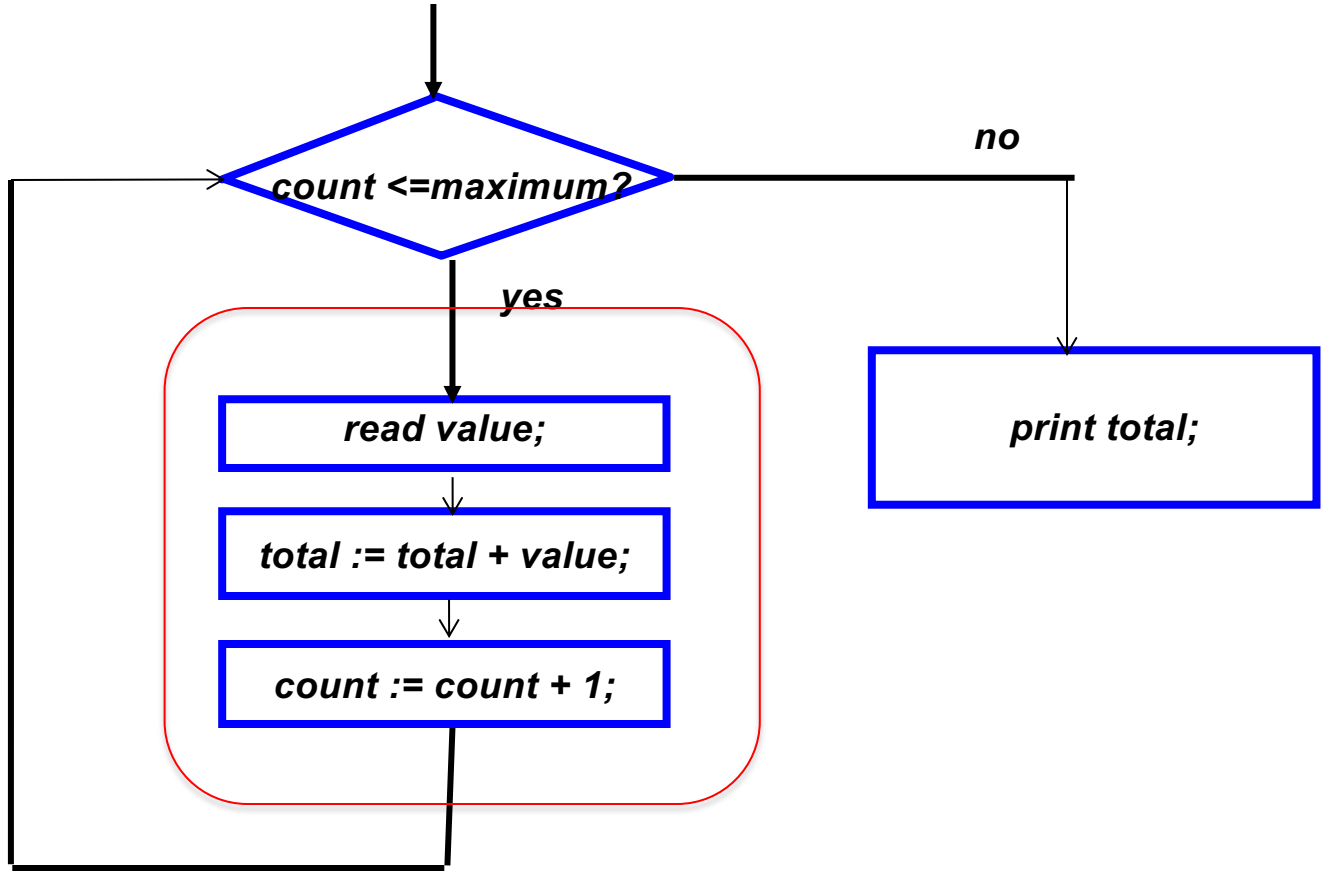
Control Flow Graph (CFG) – Example

```
total, value, count, maximum : int;

total := 0;

count := 1;

read maximum;

while (count <= maximum) do

        read value;

        total := total + value;

        count: = count + 1;

endwhile;

print total;
```



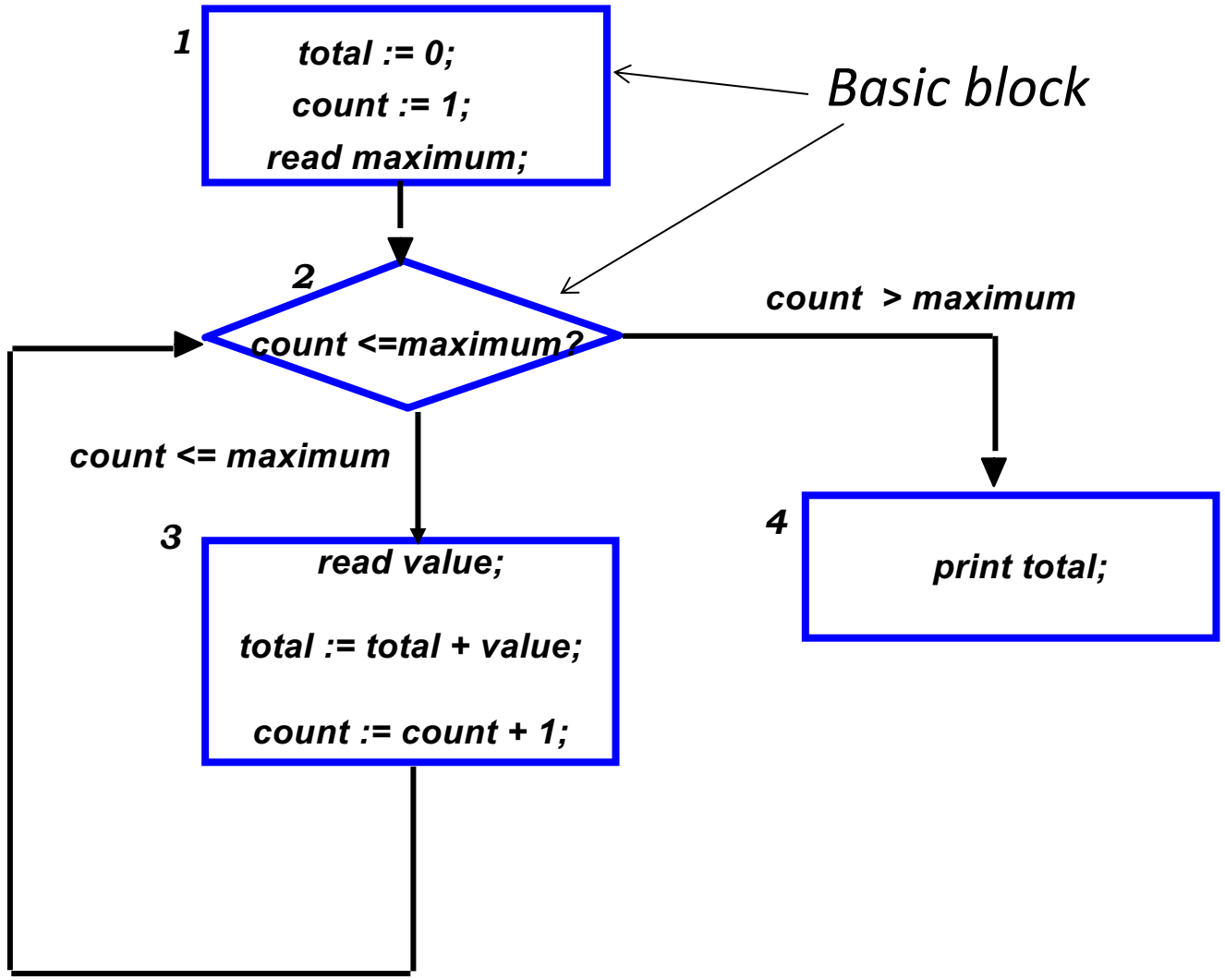
Basic Block

- Maximal program region with a **single entry** and **single exit** point



Control Flow Graph (CFG) – Example

```
total, value, count, maximum : int;
total := 0;
count := 1;
read maximum;
while (count <= maximum) do
        read value;
        total := total + value;
        count: = count + 1;
endwhile;
print total;
```



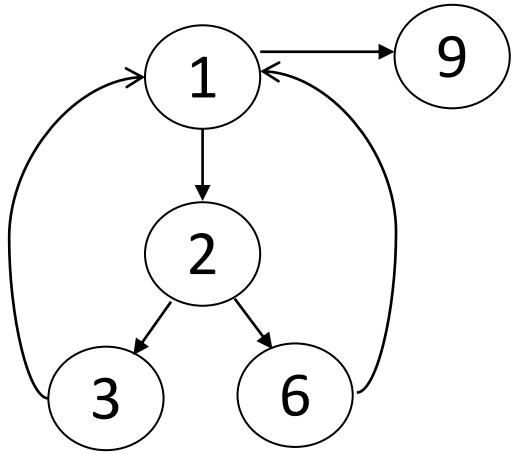
Control Flow Graph (CFG) – Definition

- Nodes N: statements or basic blocks
- Directed edges E: *potential* transfer of control from the end of one region directly to the beginning of another
  - E = { (n_i, n_j) | syntactically, the execution of n_j follows the execution of n_i}
- Intra-procedural (within a method)

Path

- A control-flow induced sequence of statement

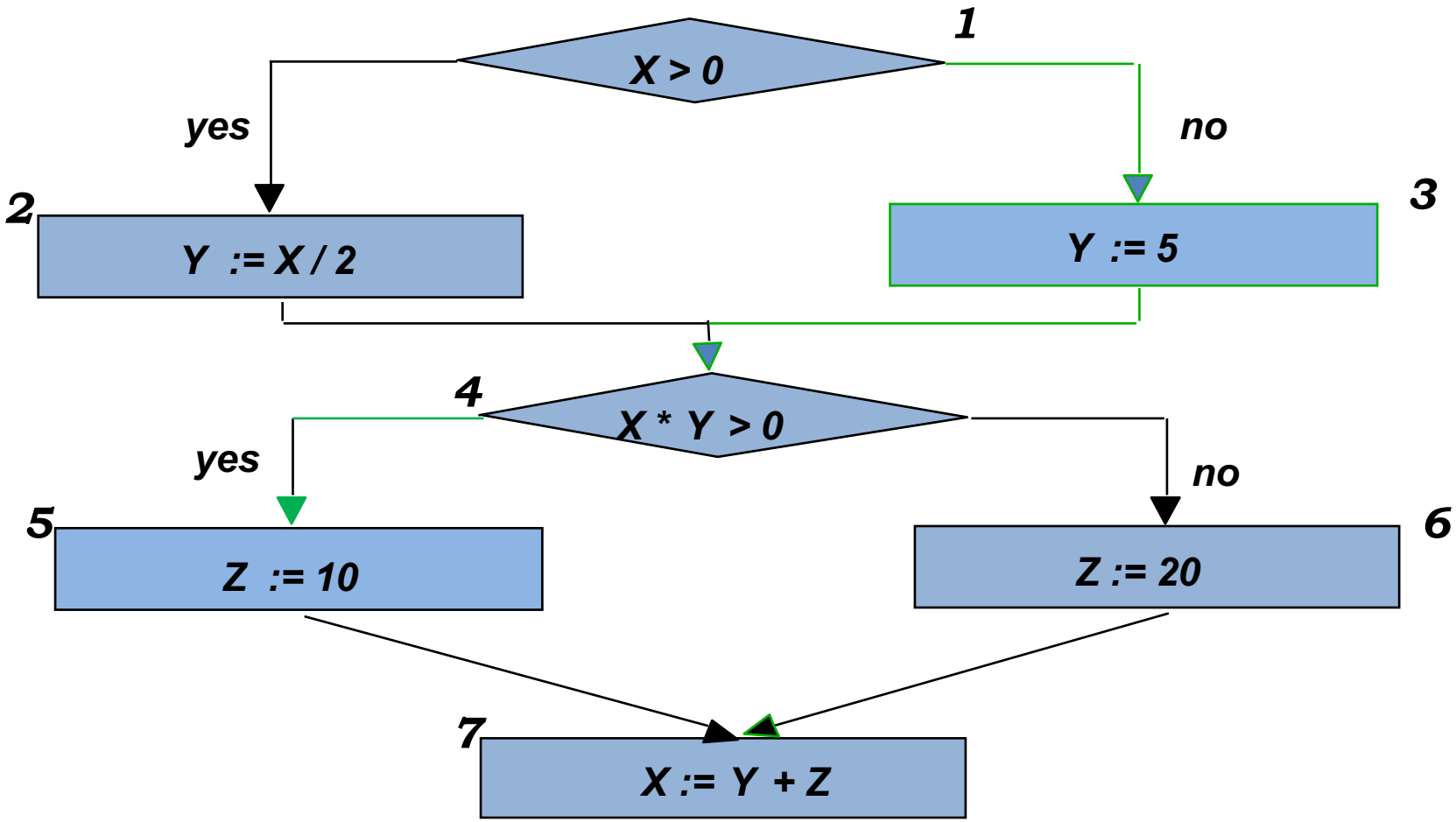
```
1 while (b!=0) {
2   if (a > b) {
3       a = a – b;
4   }
5   else {
6     b  = b – a;
7   }
8 }
9 return a;
```



1 → 2 → 3 → 1→ 9
1 → 2 → 3 → 1→ 2 → 3 → 1→ 9
…
1→ 2 → 6 → 1→ 9
…
1 → 2 → 3 → 1 → 2 → 6 → 1→ 9
…

1→ 2 → 3 → 1→ 2 → 6 → ?

Infeasible paths



Dead and Unreachable Code

unreachable code

Never executed

```
X:= X + 1;
Goto loop;
Y = Y + 5;
```

dead code

'Executed', but irrelevant

```
X = X + 1;
X = 7;
X = X + Y;
```

Benefits of CFG

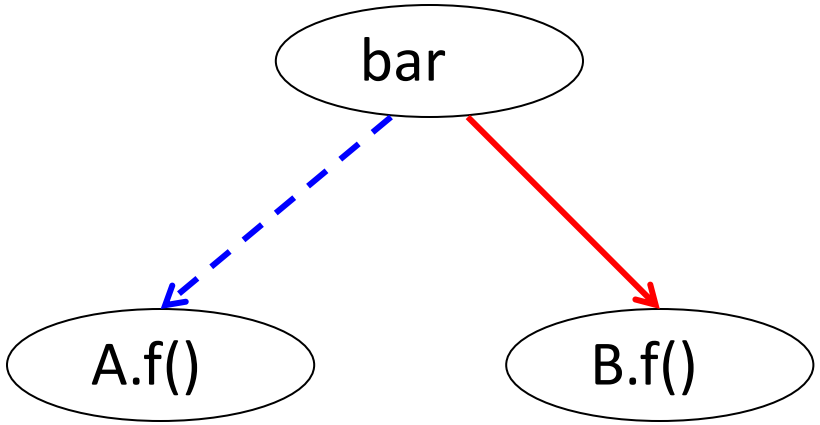
- Probably the most commonly used representation
- Basis for many types of automated analysis
  - Graphical representations of interesting programs are too complex for direct human understanding
- Basis for various transformations
  - Compiler optimizations

Call Graphs (Interprocedural CFG)

- Between functions (not within)
- Nodes represent procedures
  - Java methods
  - C functions
  - …
- Edges represent potential *calls* relation

Call Graph With Method Overriding

```
class A {
  void f();
}
class B extends A {
  void f();
}

bar() {
  B b = new B();
  A a = b;
  a.f();
}
```



Question: which edges are in the call graph?

A: Blue dotted edge

B: Red solid edge

C: Both

D: None

Call Graphs … Not That Simple

- Creating the exact (static) call graph is an **undecidable** problem

```
class A {
  void f();
}
class B extends A {
  void f();
}

bar(A a) {
  a.f();
}
```

(Rice's theorem)

"All non-trivial, semantic properties of programs are undecidable".

- A *semantic property* is about the program's behavior (for instance, does the program terminate for all inputs)
  - Unlike a syntactic property (for instance, does the program contain an if-then-else statement).
- A property *is non-trivial* if it is neither true nor false for every computable function

Call Graphs … Not That Simple

- Creating the exact (static) call graph is an **undecidable** problem
- Computing call graphs requires
  - Point-to analysis (i.e., analysis of types)
  - Exceptions
  - …
- Multiple existing heuristic algorithms
  - Various degrees of precision / scalability

```
class A {
  void f();
}
class B extends A {
  void f();
}

bar(A a) {
  a.f();
}
```

Summary So Far

- Control Flow Graph:
  - Nodes: either program statements or basic blocks
  - Edges: potential flow of control between statements / blocks
- Call Graph:
  - Nodes: methods
  - Edges: potential flow of control between methods (method calls)

Data Flow Analysis

- A technique for gathering information about the propagation of data values in the program

Variable Definition and Uses (DU)

- Variable **definition:** the variable is assigned a value
  - Variable declaration (often the special value “uninitialized”)
  - Variable initialization
  - Assignment
  - Values received by a parameter, e.g., foo(23);
  - Value increments

```
int x
int x = 5
x = 5
foo(int x)
x++
```

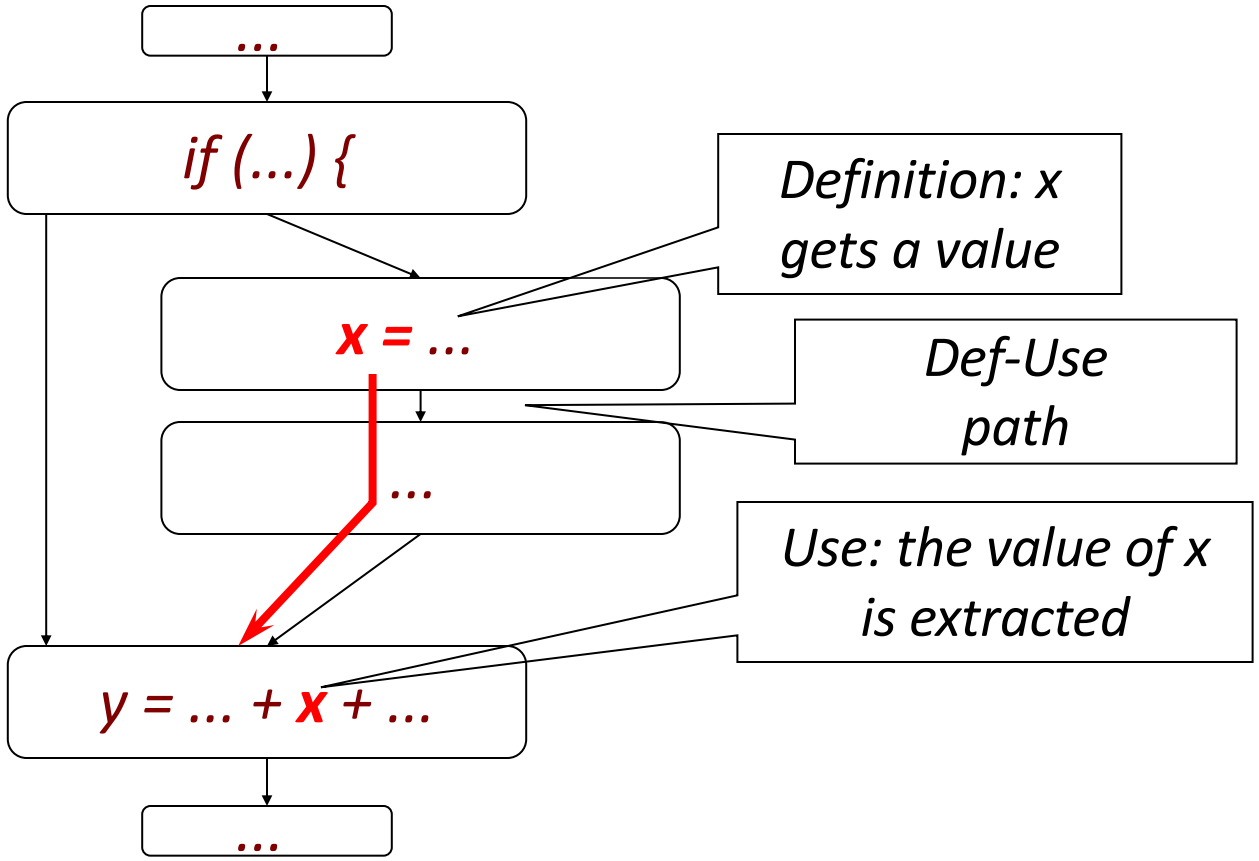
- Variable **use:** the variable’s value is actually used
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

```
y = x
if (x>0)
foo(x)
return x
x++
```

Def-Use Path

```
...
if (...) {
   x = ... ;
...
}
y = ... + x + ... ;
```

Not a control-flow path!

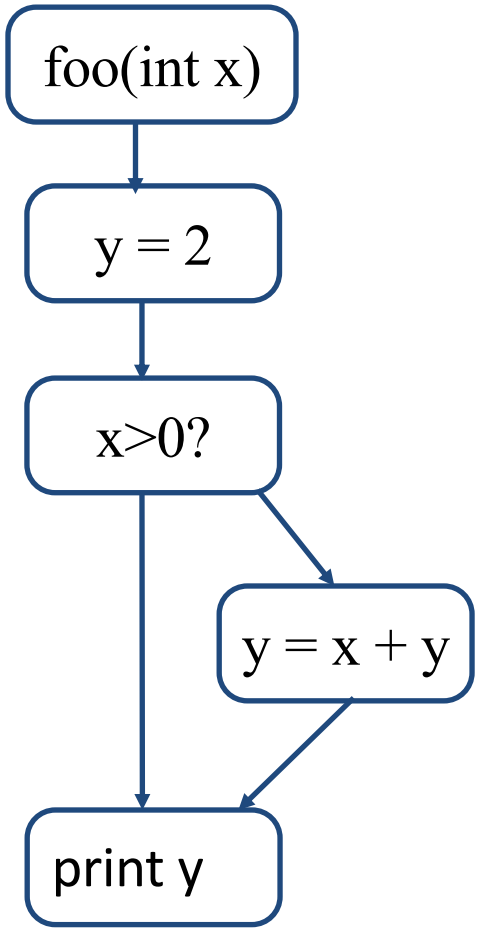


Data Dependence Graph

- Nodes: program statements
- Edges: def-use (du) pairs, labeled with the variable name

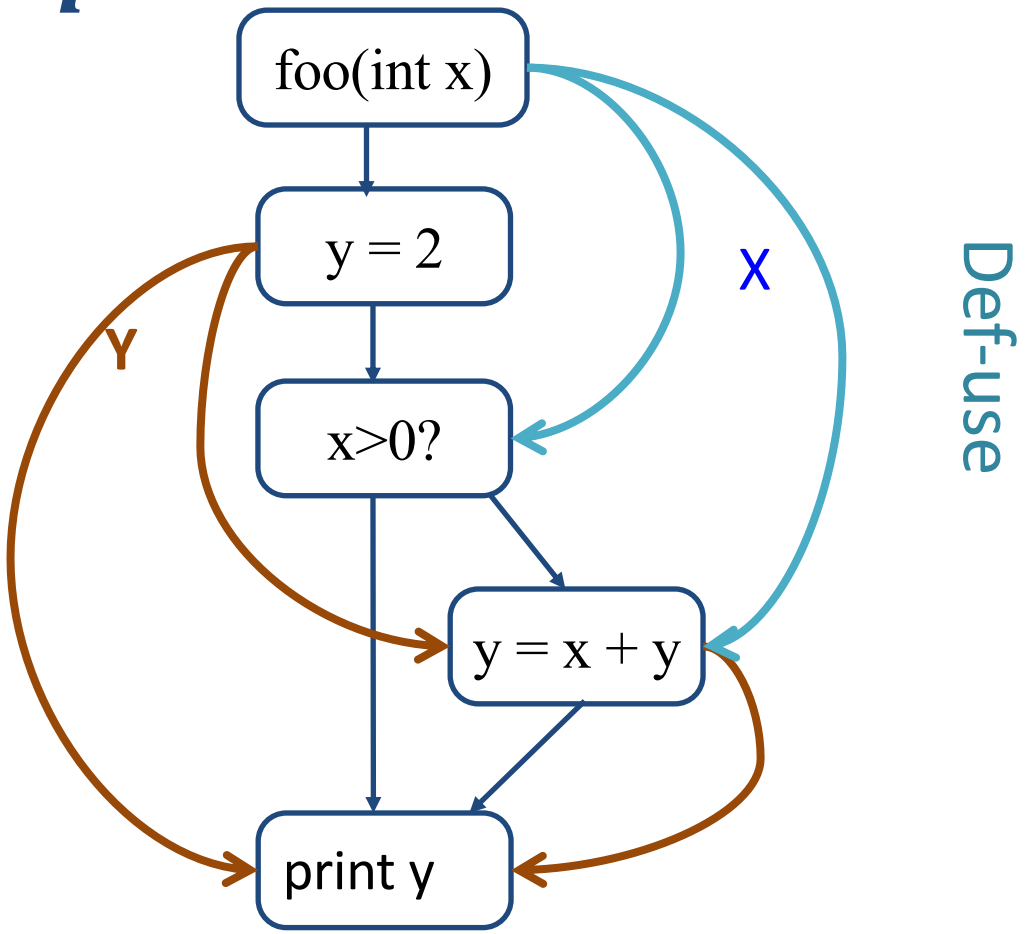
Example

```
foo(int x) {
  y = 2;
  if(x > 0)
      y = x + y;
  endif;
  print y;
}
```



Example

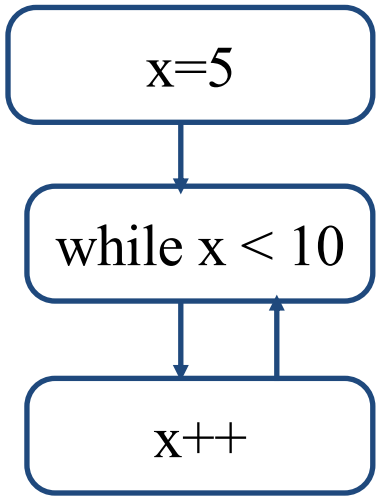
```
foo(int x) {
  y = 2;
  if(x > 0)
      y = x + y;
  endif;
  print y;
}
```



What can be printed in the last statement?

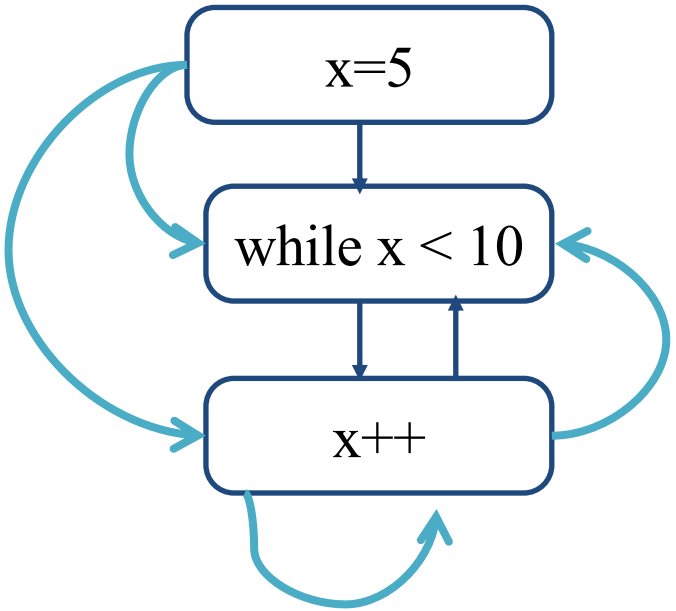
What about loops?

```
x=5;
while (x< 10)
{
  x++;
}
```



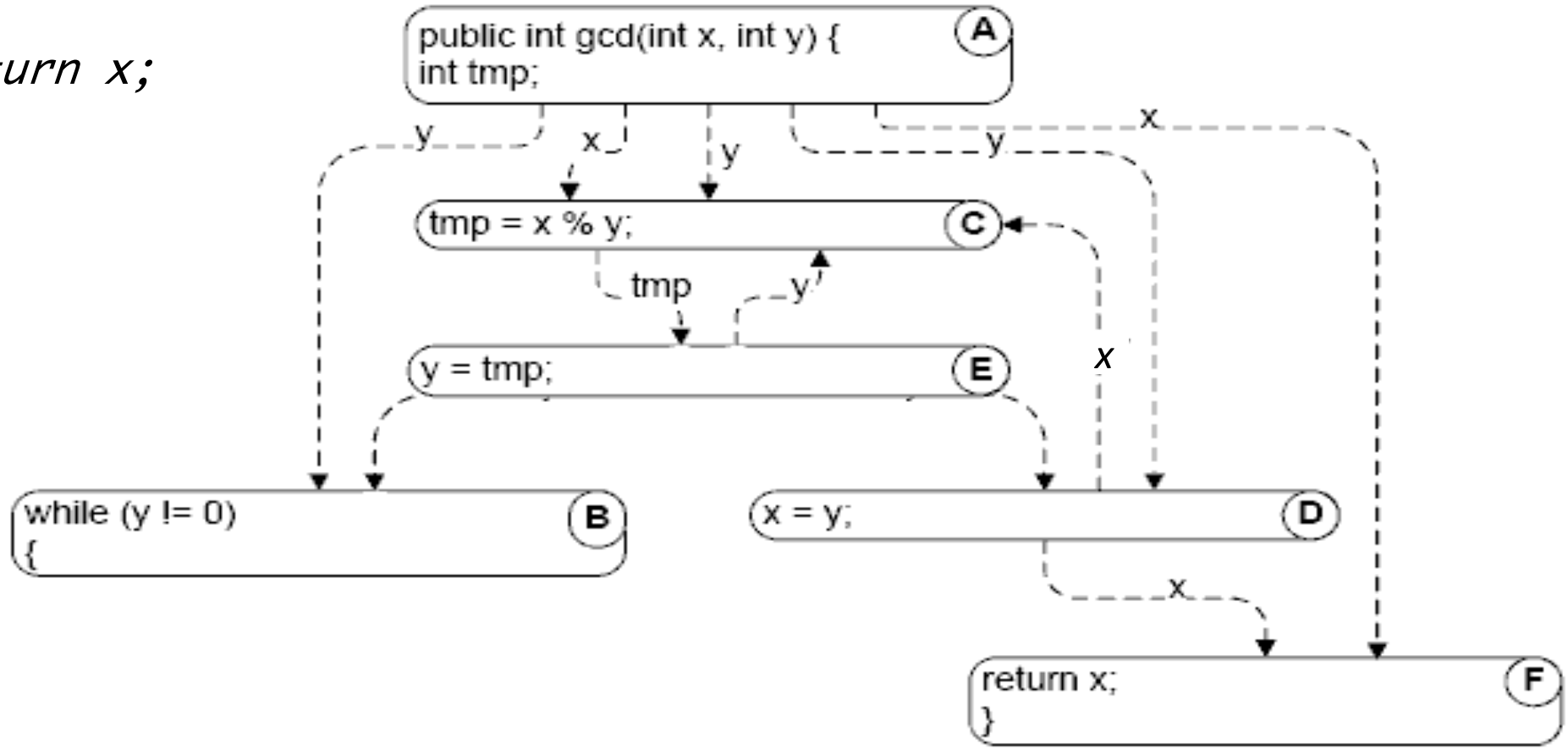
What about loops?

```
x=5;
while (x< 10)
{
  x++;
}
```



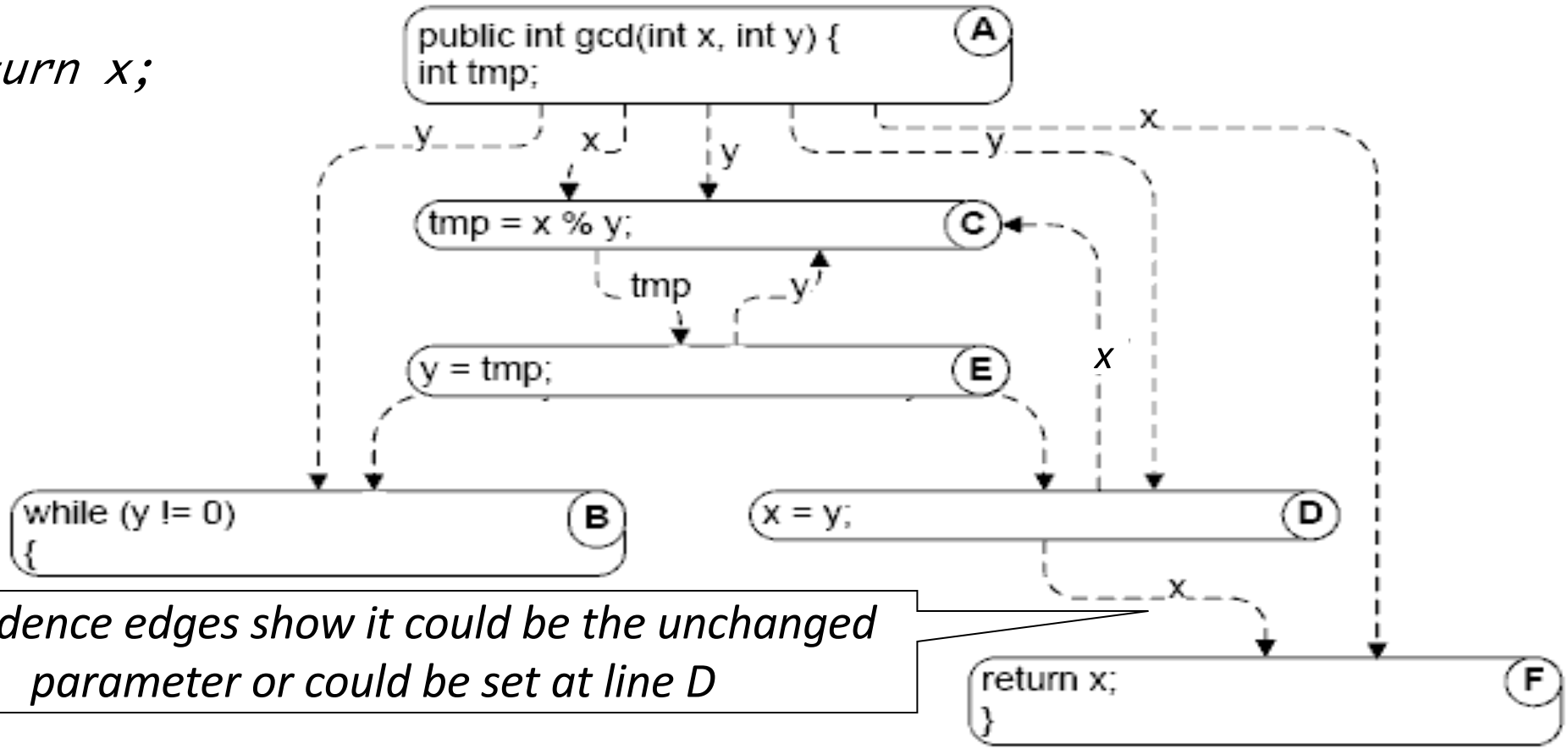
```
A: public int gcd(int x, int y) {
     int tmp;
B: while (y != 0) {
C:   tmp = x % y;
D:     x = y;
E:     y = tmp;

   }
F:  return x;
}
```

Control flow edges are omitted in this example



```
A: public int gcd(int x, int y) {
     int tmp;
B: while (y != 0) {
C:   tmp = x % y;
D:     x = y;
E:     y = tmp;

   }
F:   return x;
}
```

"where could the value returned in line F come from?"



Dependence edges show it could be the unchanged parameter or could be set at line D

Data Flow Analysis – How Used

- Compilers and optimization, e.g.,
  - determine if a definition is dead and can be removed
  - determine if a variable always has a constant value

- Security analysis, e.g.,
  - determine if a sensitive value reaches a sensitive sink **(taint analysis**)
    - Information leakages
    - SQL injection

- …

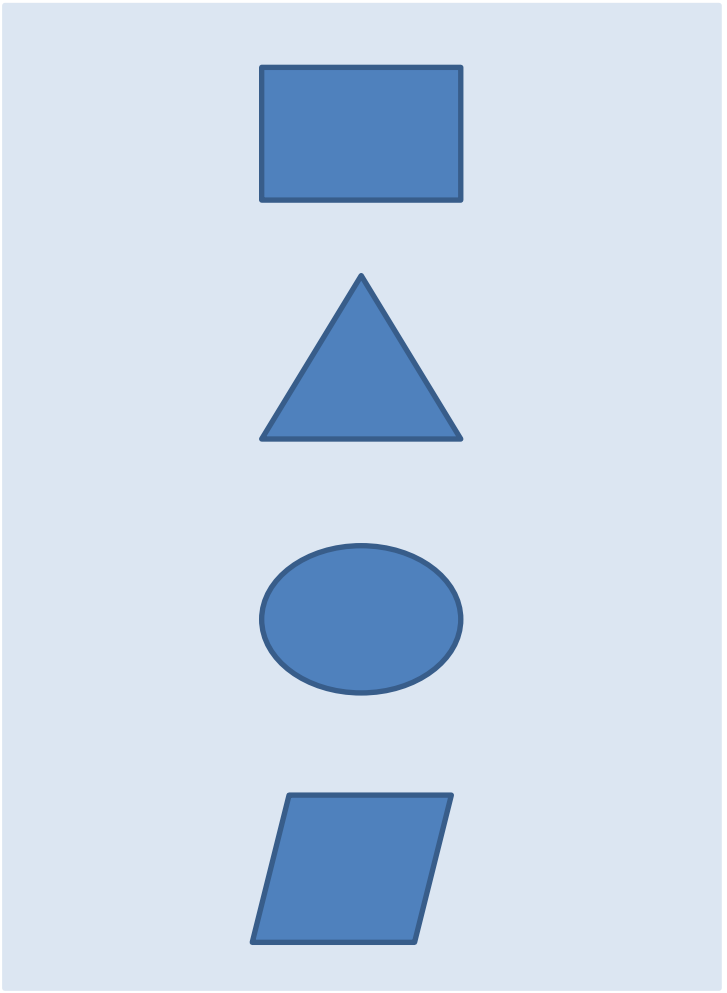
Taint Analysis: A Type of Data Flow Analysis

Def-use chains:

```
x = 7;
y = x + 2;
z = y / 5;
```

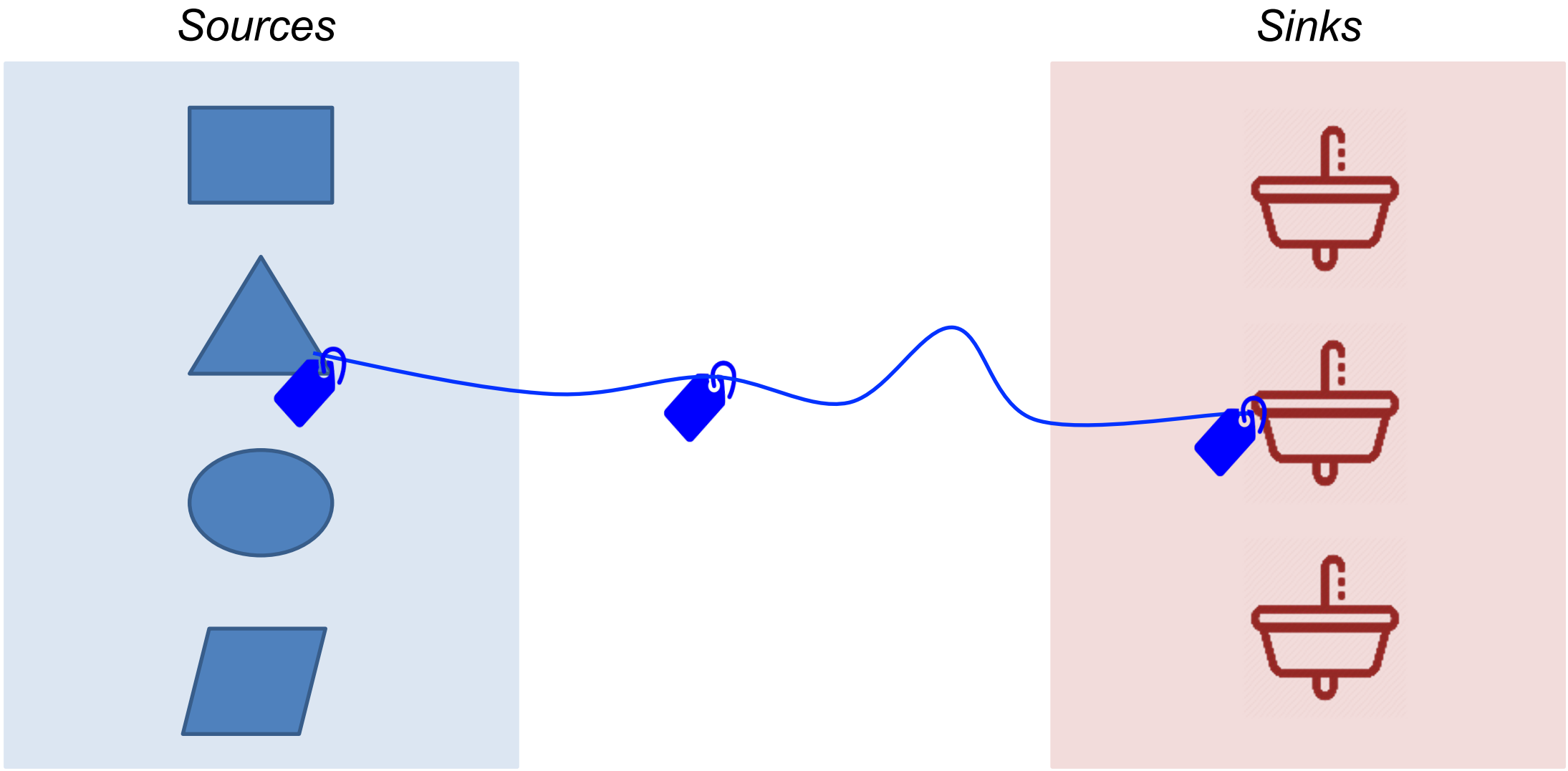
Taint Analysis: A Type of Data Flow Analysis

Sources



Sinks

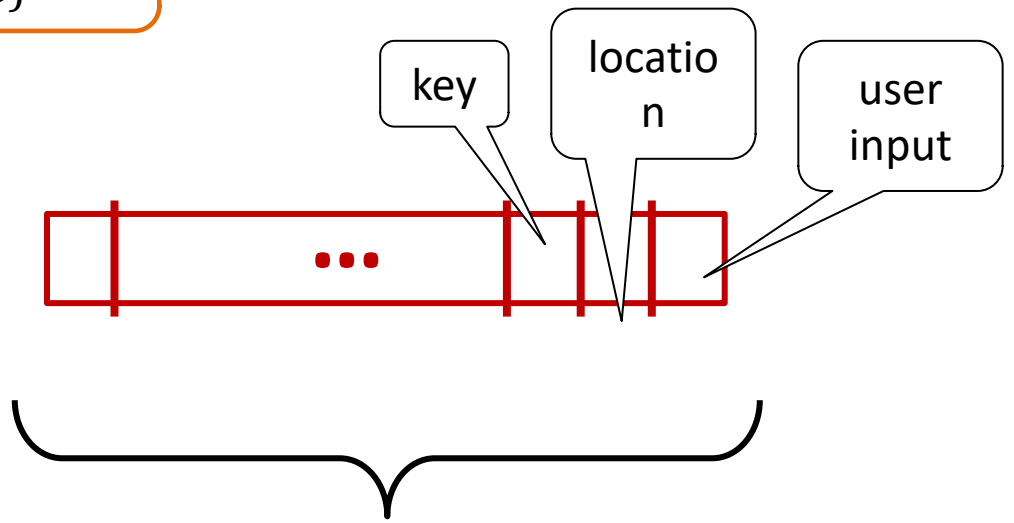
Taint Analysis: A Type of Data Flow Analysis



Dynamic Taint Analysis

```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    int input1T = 0b0001;
 5     String input2 = getUserInput();
 5T    int input2T = 0b0001;
 6     Location loc = getLocation();
 6T    int locT = 0b0010;
 7     String size = input1.getLength();
 7T    int sizeT = input1T;
 8     String lang = loc.getLanguage();
 8T    int langT = locT;
 9     String data = size + lang;
 9T    int dataT = sizeT | langT;
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   if (dataT != 0){report(data, dataT);}
23     sendToInternet(data);                    //sink
24 }}
```

Taint marking
(int per variable)

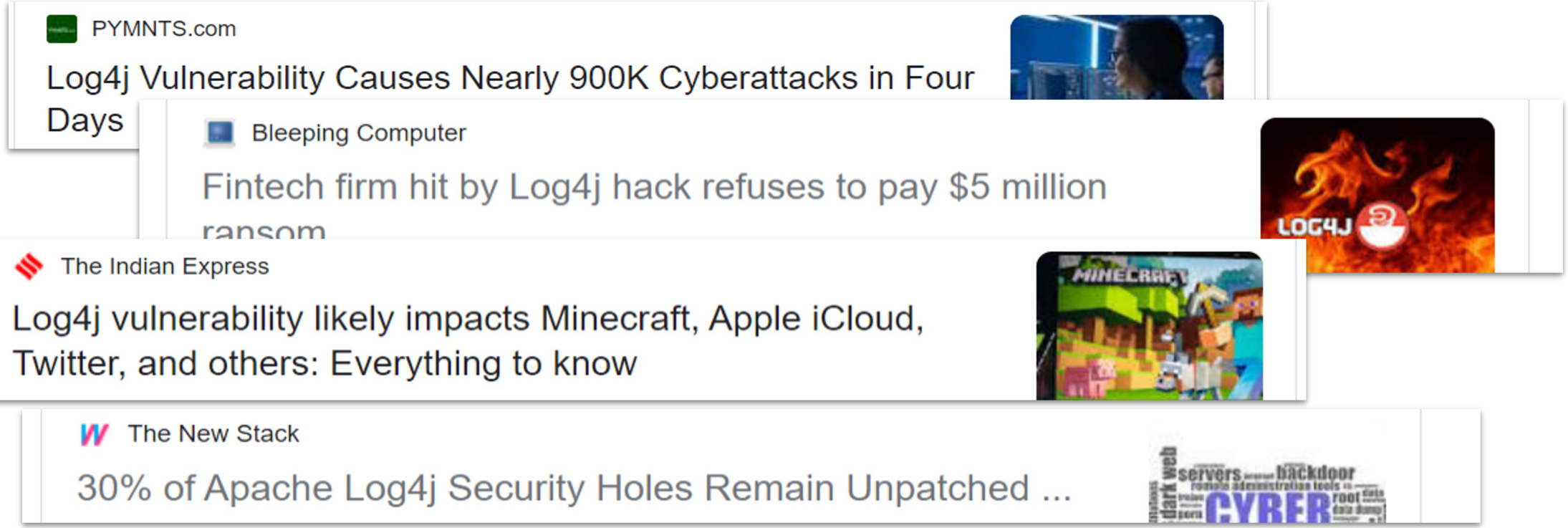


32 or 64 bits,
one bit per taint,
(hopefully) kept in a register

[3] Khaled Ahmed, Yingying Wang, Mieszko Lis, and Julia Rubin. *ViaLin: Path-Aware Dynamic Taint Analysis for Android*. FSE 2023.

Sources and Sinks in Security Domain

- Sensitive user data → Internet
- Data controlled by an attacker → database query (SQL injection)
- Data controlled by an attacker → JNDI API (Log4Shell vulnerability, end of 2021)
- …

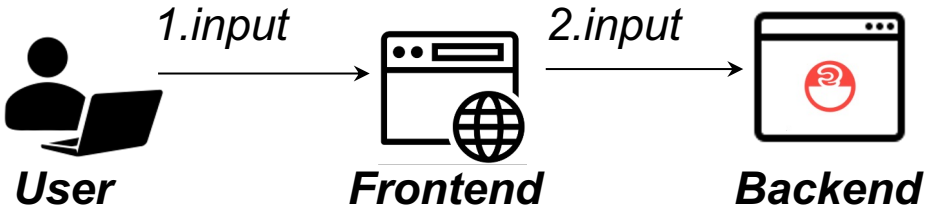


The Essence of Log4Shell, CVE-2021-44228

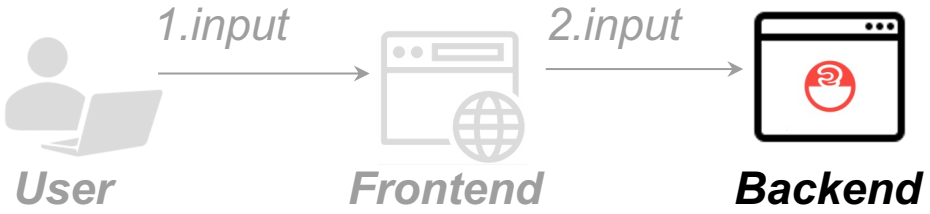
- User input is passed to JNDI method `InitialContext.lookup(String)` (which results in downloading and executing a malicious payload)

- “Classical” taint analysis problem…

Benign use case



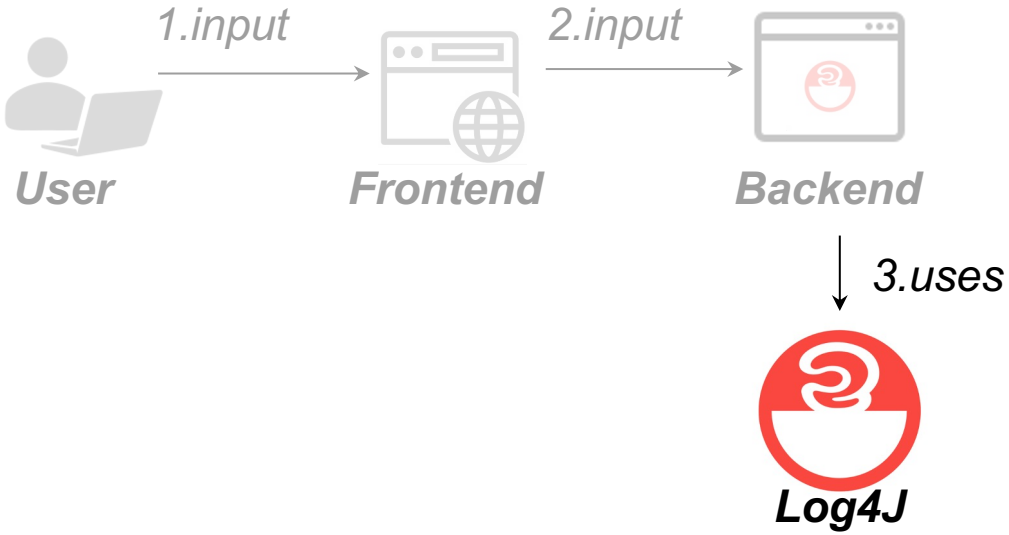
Benign use case



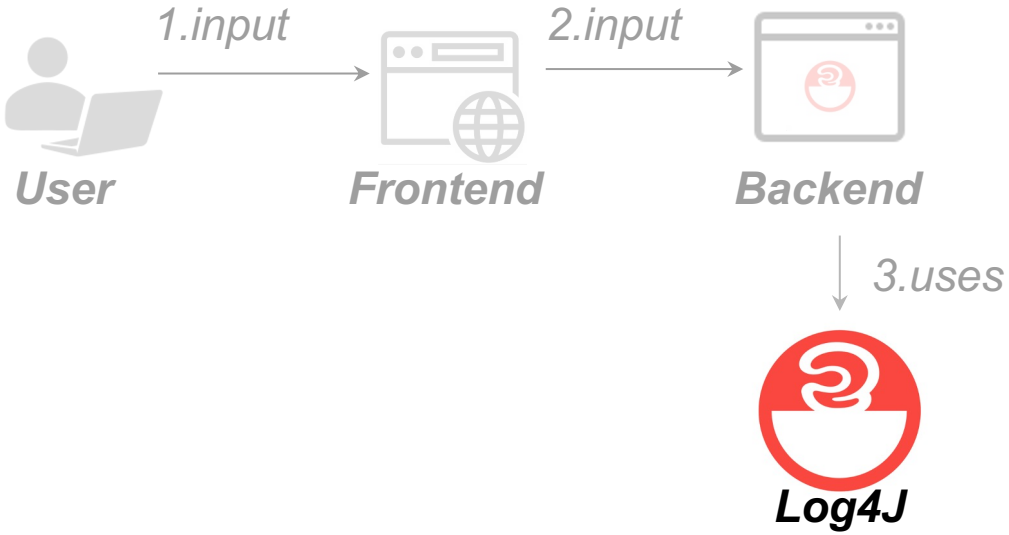
(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){
3.    logger.info(input); // log4j
4.    ...
5.}}
```

Benign use case



Benign use case



- popular logging library for Java

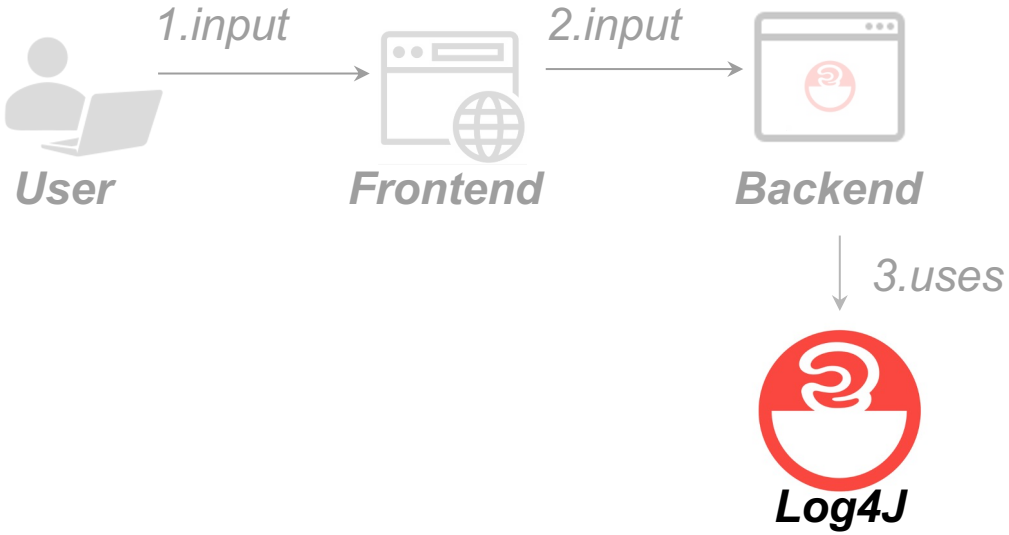
Example:

```
1. logger.info("Hello, world!");
```

Console output:

```
Hello, world!

...
```

Benign use case



- popular logging library for Java
- replace strings in format **"${prefix:name}"** with actual values, during runtime

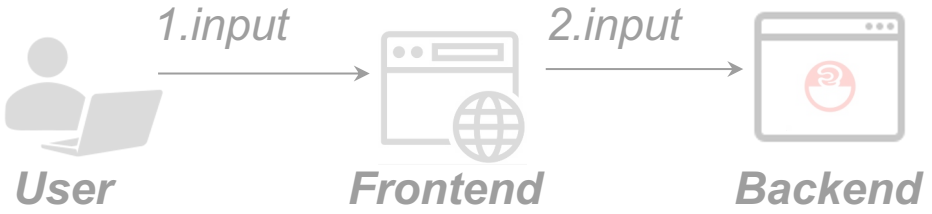
Example:

```
1. logger.info("Hello, world!");
2. logger.info("${java:runtime}");
3. logger.info("${docker:containerId}");
4. logger.info("${jndi:ldap://ldap_server/obj}");
5. ...
```

Console output:

```
Hello, world!
OpenJDK Runtime Environment (build 1.8.0_302-b08) from Temurin
...
```

Benign use case



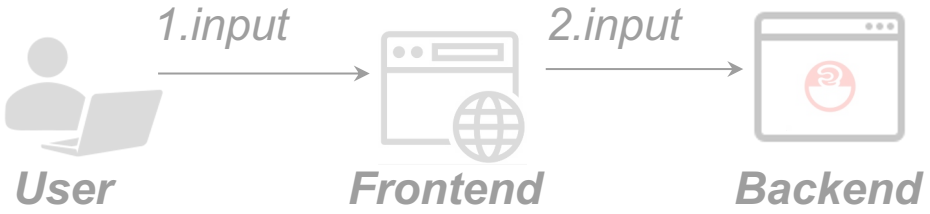
(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){
3.    logger.info(input); // log4j
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3);
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```

Benign use case



(simplified) Backend app code:

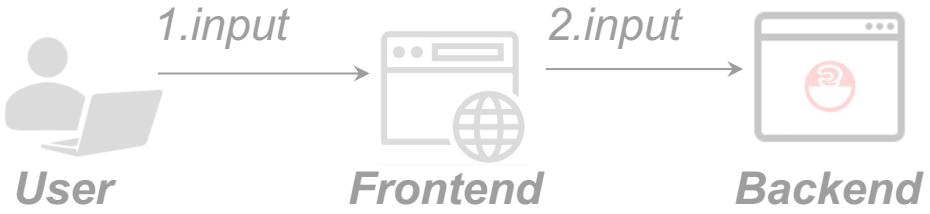
```
1.public class App{
2.  public static void search(String input){
3.    logger.info(input); // log4j
4.    ...
5.}}
```

Check if string in format *"${xxx}"*

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.      int sInd = 2;
4.      int eInd = str.length()-1;
5.      String str2 = str.substring(sInd, eInd);
6.      if (str2.startsWith("java:")) {...}
7.      else if (str2.startsWith("docker:")) {...}
8.      else if (str2.startsWith("jndi:")) {
9.         String str3 = str2.substring(5);
10.        Context ctx = new InitialContext();
11.        Object obj = ctx.lookup(str3);
12.        ...
13.     }else {...}
14.  }else {...}
15.}
```

Benign use case



(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){
3.    logger.info(input); // log4j
4.    ...
5.}}
```

3.uses

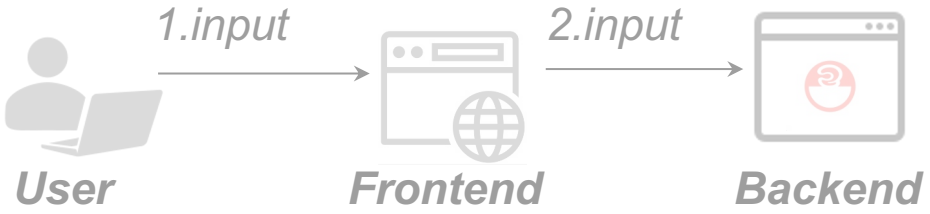
Log4J

Strip away
"${" and "}"

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3);
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```

Benign use case



3.uses

Log4J

Match prefix

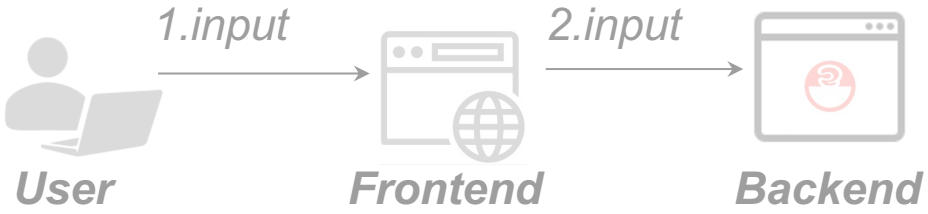
(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){
3.    logger.info(input); // log4j
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.         String str3 = str2.substring(5);
10.        Context ctx = new InitialContext();
11.        Object obj = ctx.lookup(str3);
12.        ...
13.    }else {...}
14.  }else {...}
15.}
```

Benign use case



(simplified) Backend app code:

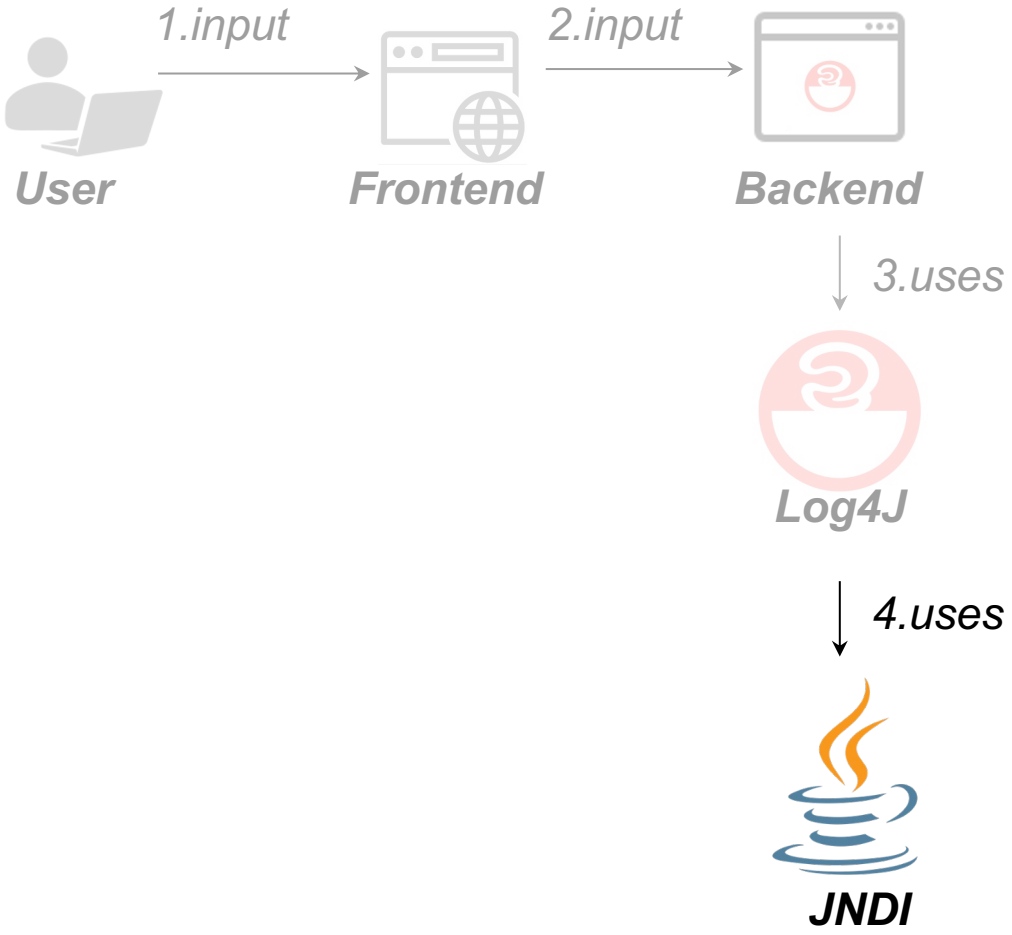
```
1.public class App{
2.  public static void search(String input){
3.    logger.info(input); // log4j
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.         String str3 = str2.substring(5);
10.        Context ctx = new InitialContext();
11.        Object obj = ctx.lookup(str3);
12.        ...
13.    }else {...}
14.  }else {...}
15.}
```

Use JNDI lookup

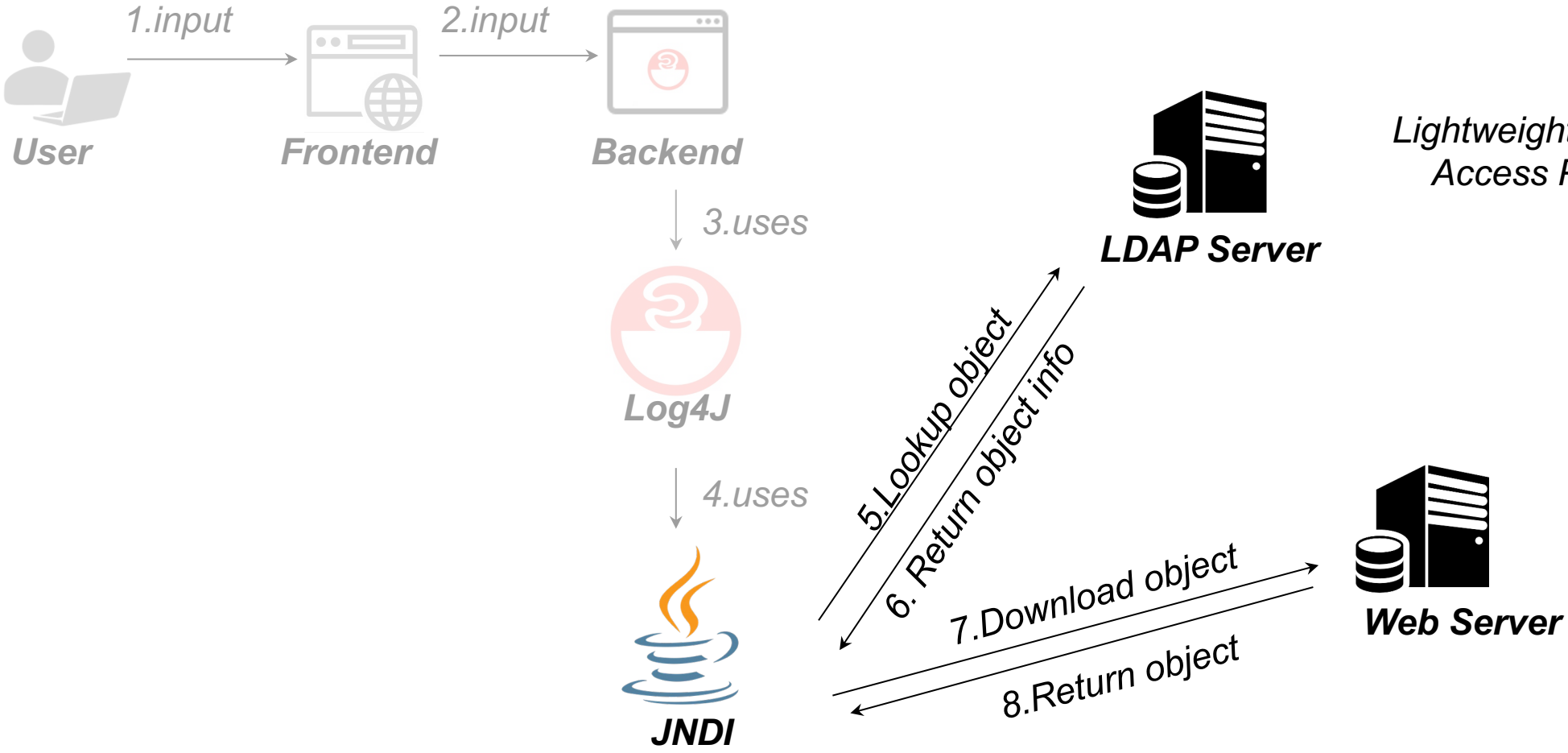
Benign use case




```
Context ctx = new InitialContext();
Object obj = ctx.lookup(name);
```

- JNDI = Java Naming and Directory Interface
- Looks up objects (classes, databases) by name

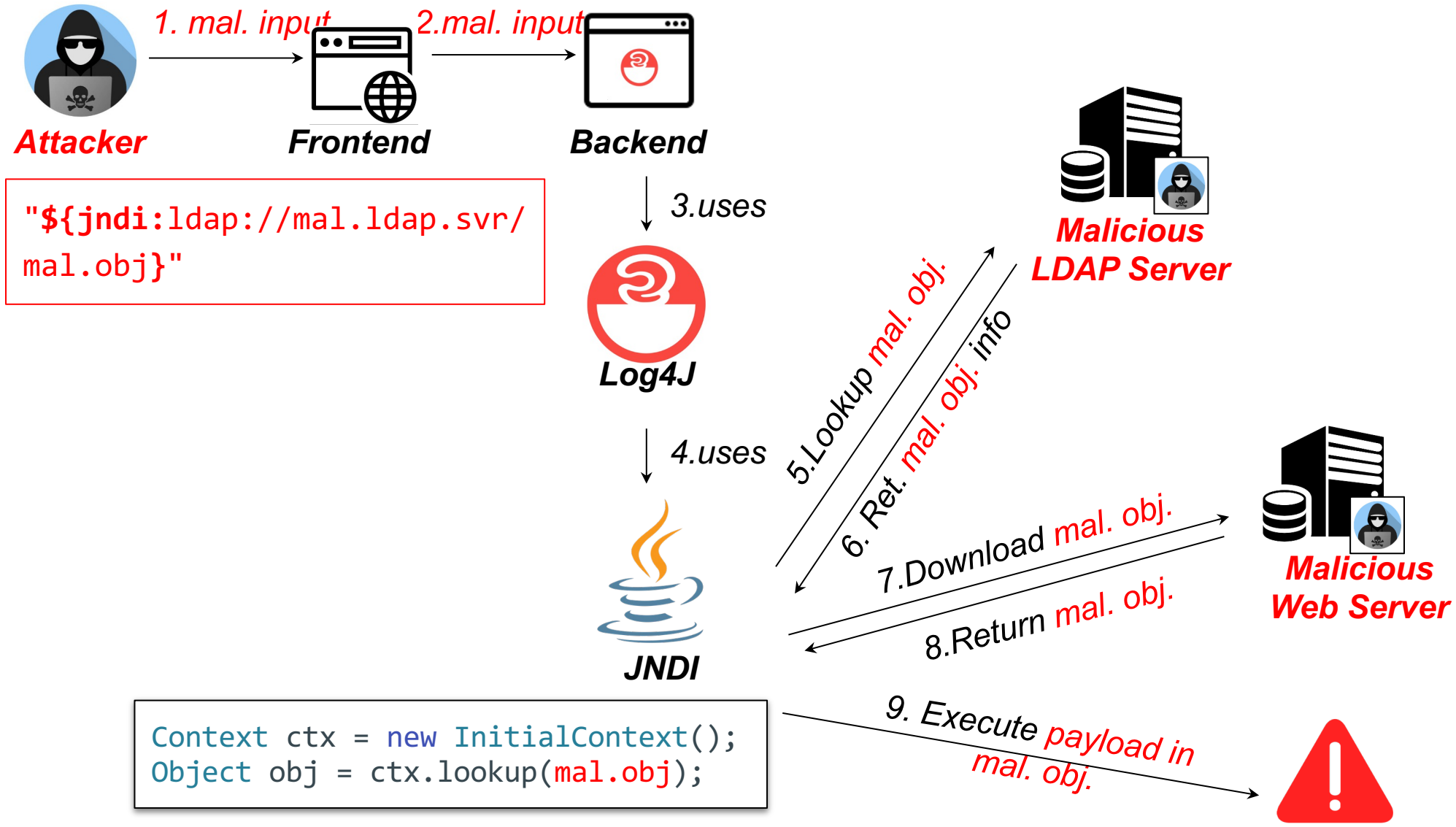
Benign use case

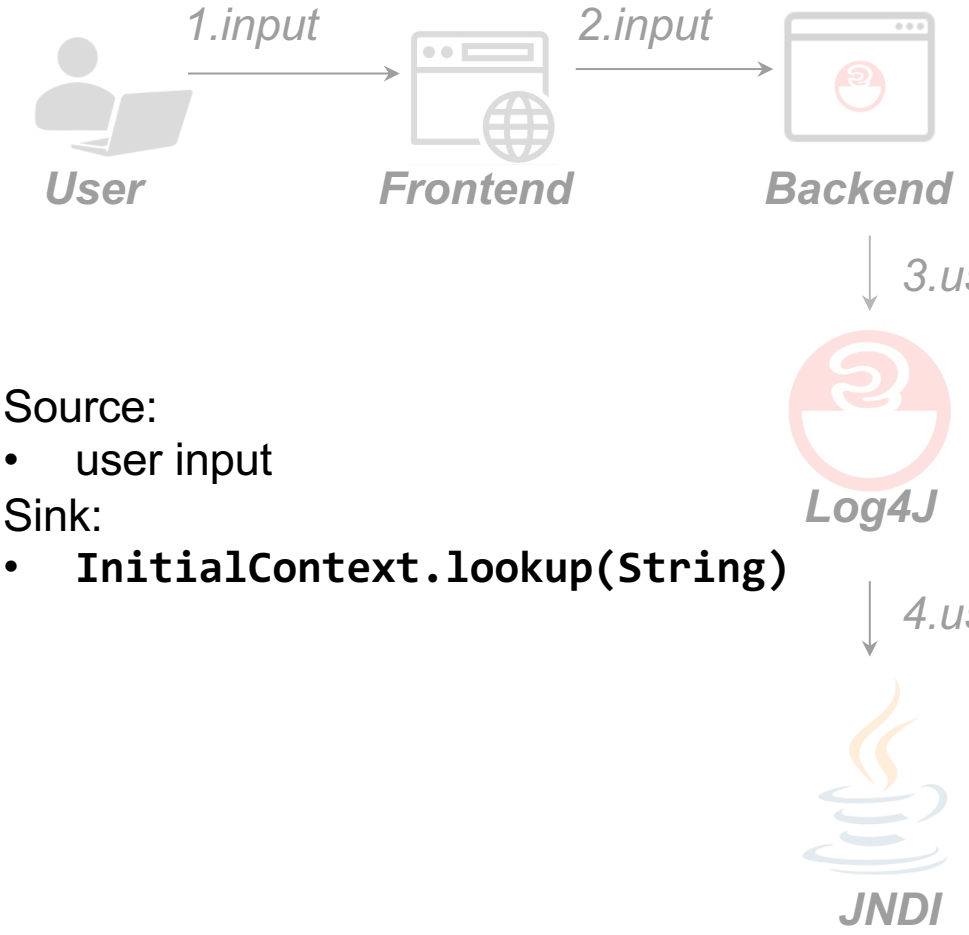



```
Context ctx = new InitialContext();
Object obj = ctx.lookup(name);
```

Exploit by attacker

Log4J would attempt to resolve malicious string via JNDI, allowing attackers to load and run arbitrary code.





Source:

- user input

Sink:

- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```

Source:
- user input

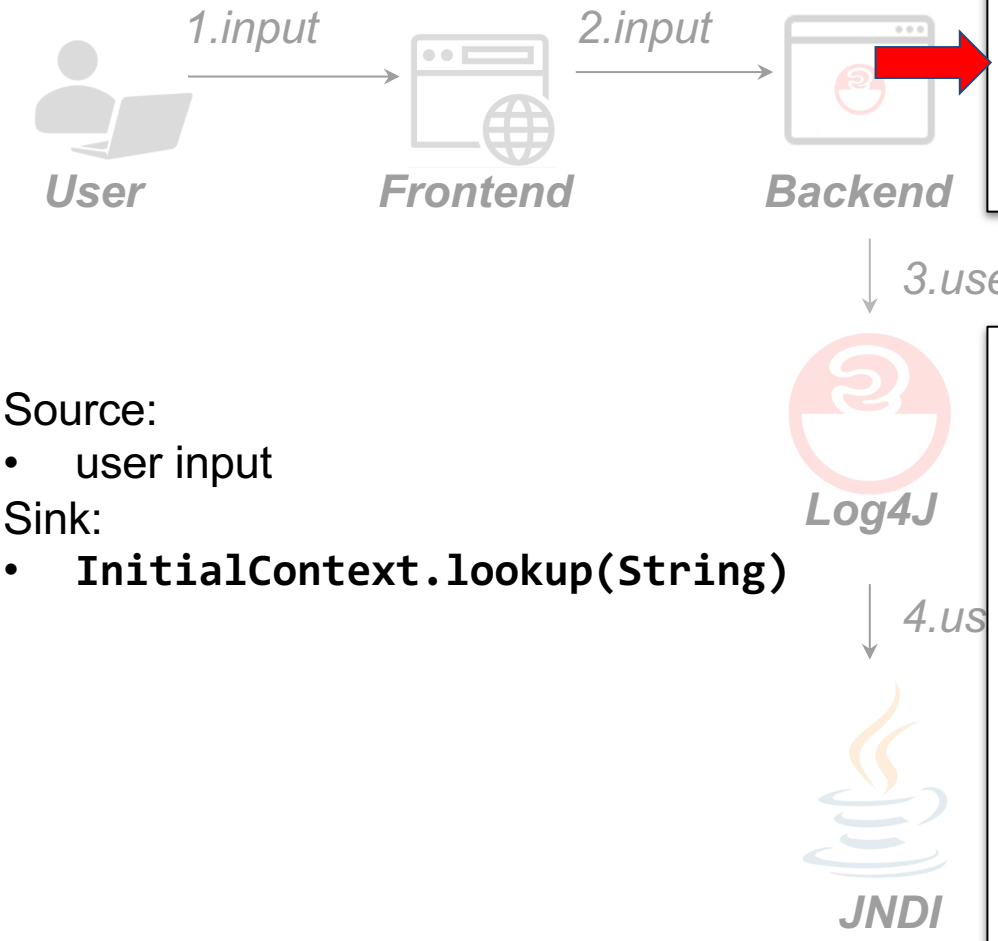
Sink:
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

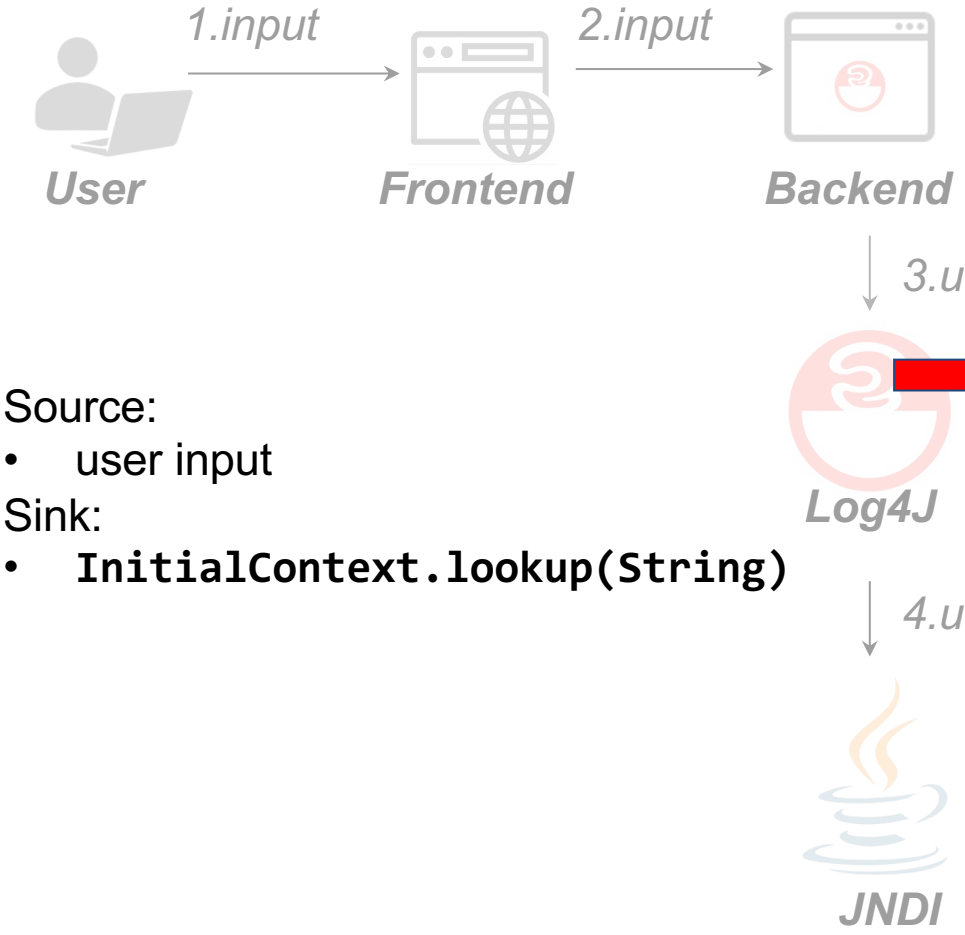
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

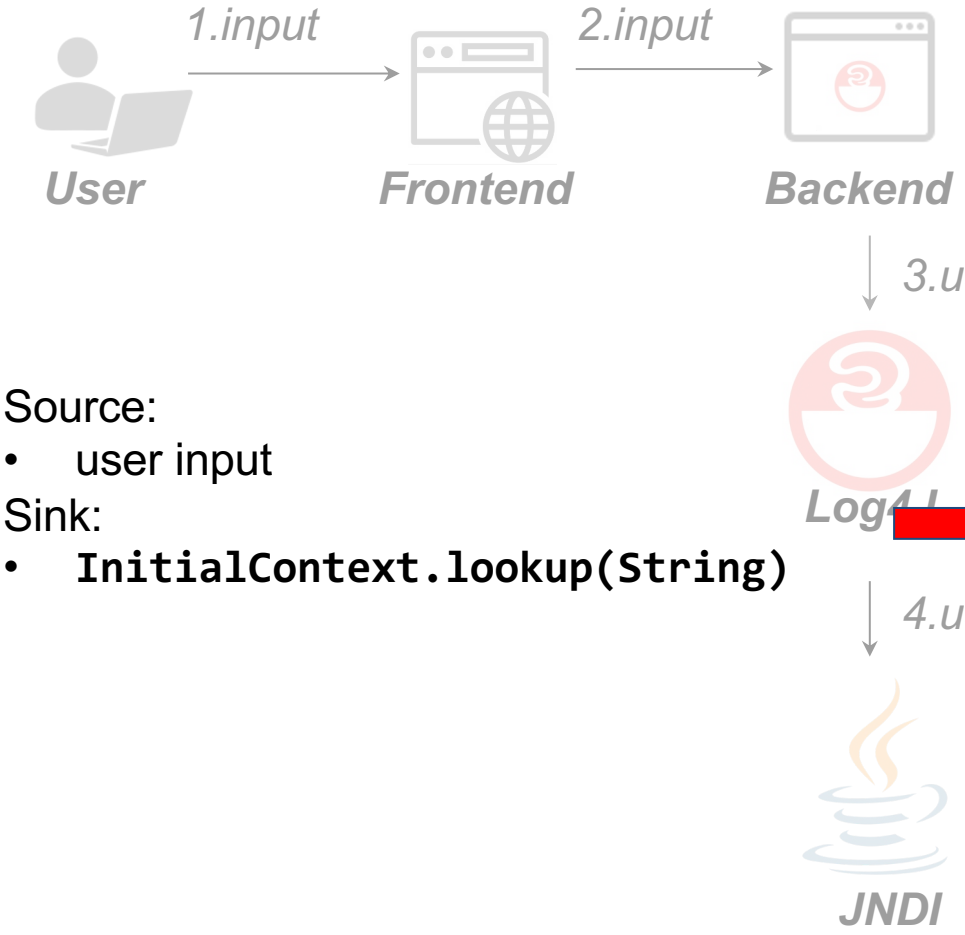
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

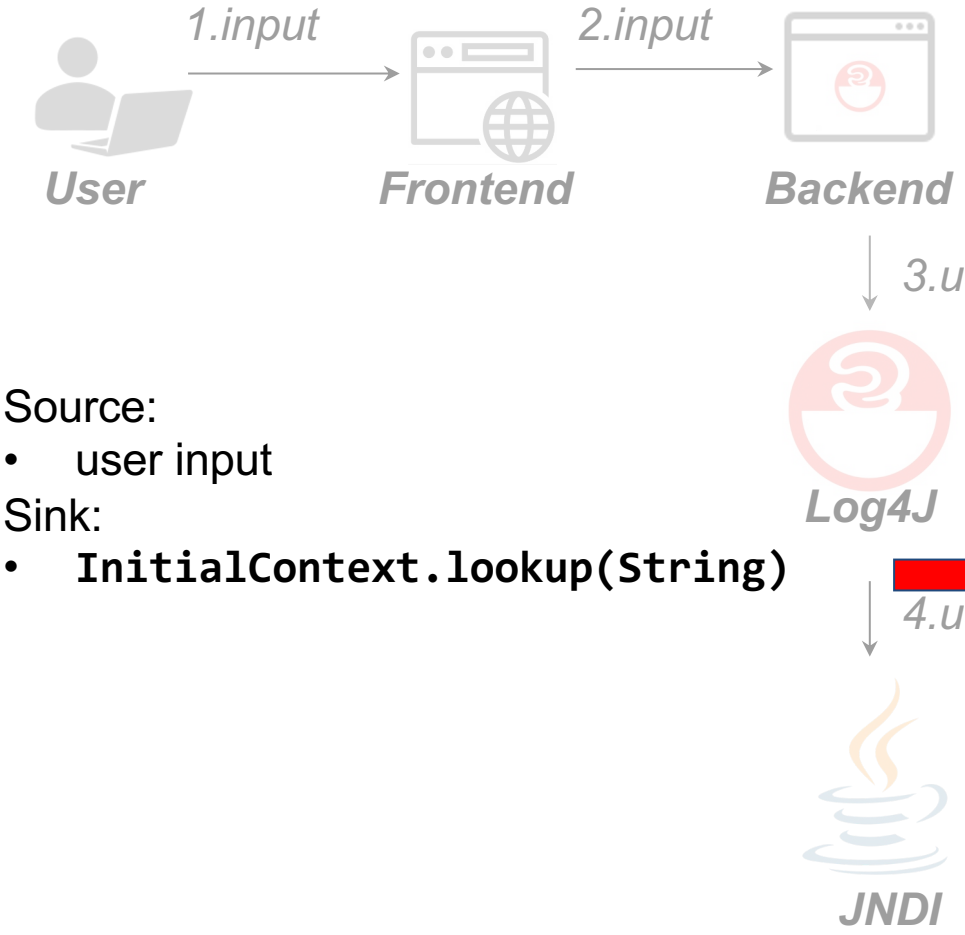
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

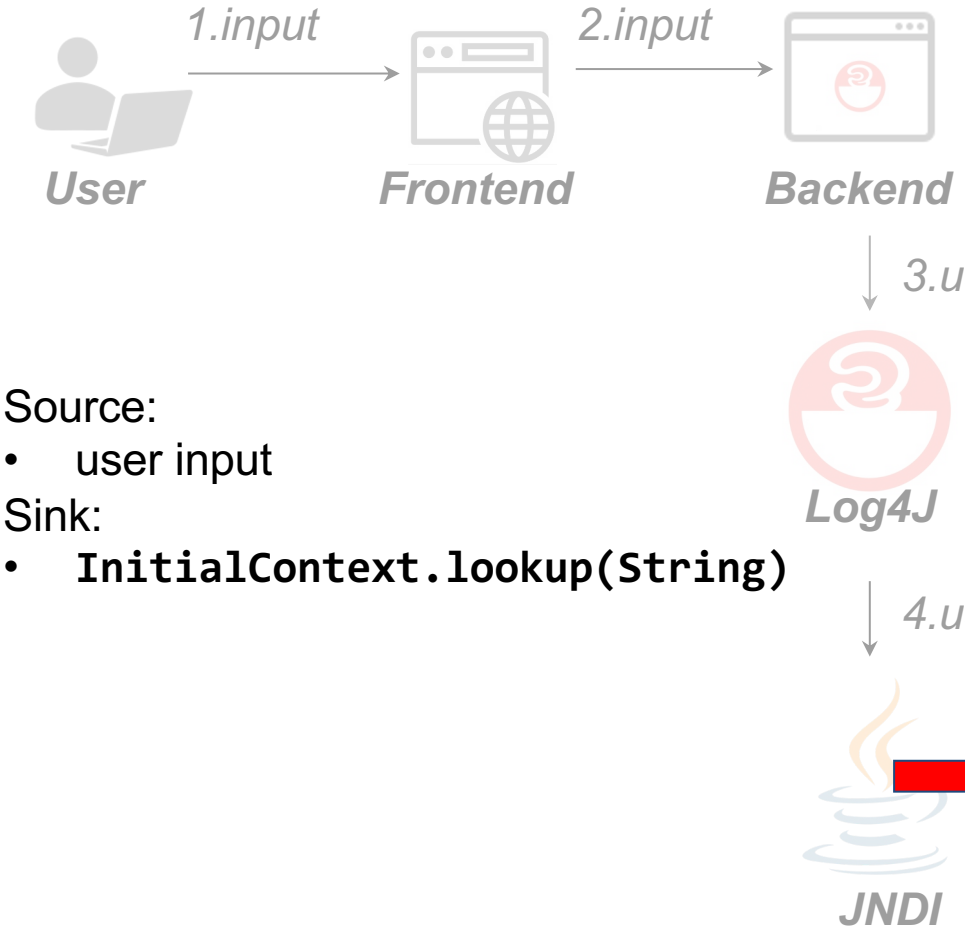
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

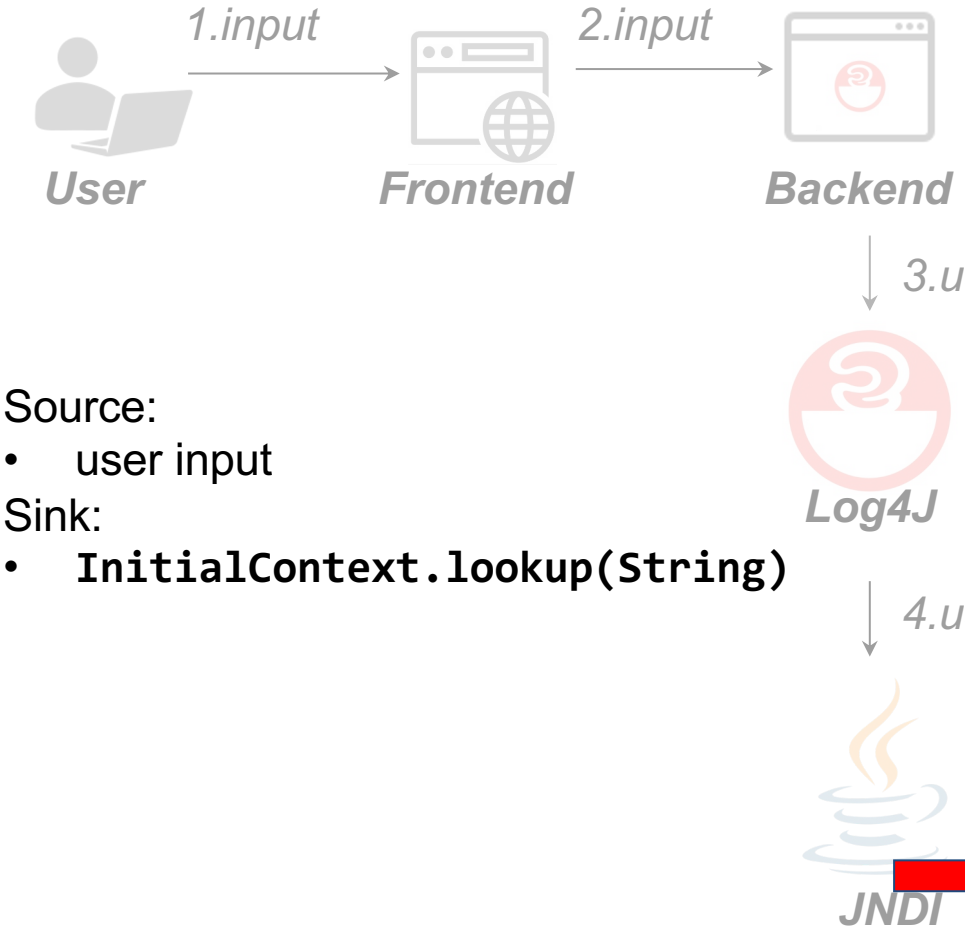
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

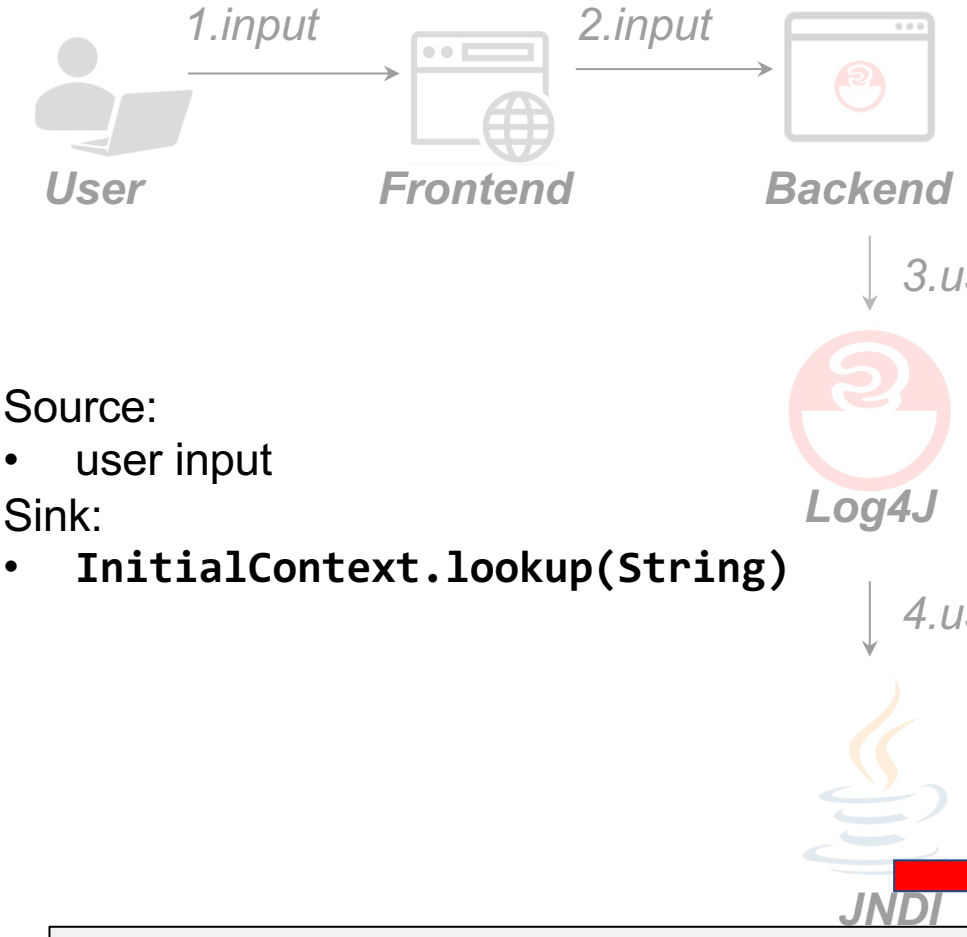
- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
12.       ...
13.    }else {...}
14.  }else {...}
15.}
```



Source:

- user input

Sink:

- **`InitialContext.lookup(String)`**

(simplified) Backend app code:

```
1.public class App{
2.  public static void search(String input){ // src
3.    logger.info(input);
4.    ...
5.}}
```

(simplified) Log4J code:

```
1. public void info(String str){
2.   if(str.startsWith("${") && str.endsWith("}")){
3.     int sInd = 2;
4.     int eInd = str.length()-1;
5.     String str2 = str.substring(sInd, eInd);
6.     if (str2.startsWith("java:")) {...}
7.     else if (str2.startsWith("docker:")) {...}
8.     else if (str2.startsWith("jndi:")) {
9.        String str3 = str2.substring(5);
10.       Context ctx = new InitialContext();
11.       Object obj = ctx.lookup(str3); // sink
```

Potential vulnerability!

*User input in **`search(String)` → `ctx.lookup(String)`***

Program Analysis: Recap So Far

- Control Flow Graph:
  - Nodes: either program statements or basic blocks
  - Edges: potential flow of control between statements / blocks
- Call Graph:
  - Nodes: methods
  - Edges: potential flow of control between methods (method calls)
- Data Flow Graph:
  - Nodes: program statements
  - Edges: def-use relationships, each labeled with the variable name (over possible control paths)

Agenda

- Basics of Analysis
  - Control flow
  - Call graph
  - Data flow
- Symbolic Execution (and its use in testing)

Symbolic Execution (and its Use in Testing)

Testing is a dynamic verification of the behavior of a program

- on a finite set of test cases
- **suitably selected from the usually infinite executions domain**
- against the specified expected behavior (oracle)

Question*: When to stop testing?*
(the domain of test cases is infinite)

Bill Sempf
@sempf

QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv.

Systematic Testing

In black-box testing:

- *Test cases come from requirements / user stories.*

In white-box testing:

- *Inspect the code / coverage criteria to see if you missed cases*

Measuring Test Suite Quality with Coverage

- Various kinds of coverage
  - **Statement**: is every statement run by some test case?
  - **Branch**: is every direction of an if or while statement (true or false) taken by some test case?
  - **Path**: is every path through the program taken by some test case?

Statement Coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

- Coverage:

  $\frac{\text{\# executed statements}}{\text{\# statements}}$

Statement Coverage

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
void testFoo() {
  foo(1);
}
```

- Coverage:

  $$\frac{\text{\# executed statements}}{\text{\# statements}}$$

Branch Coverage

- Every path going out of a node executed at least once
  - Coverage: percentage of edges hit.
- *To achieve 100%, each predicate must be both true and false*

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

Branch Coverage

- Every path going out of a node executed at least once
  - Coverage: percentage of edges hit.
- *To achieve 100%, each predicate must be both true and false*

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

```
void testFoo() {
  foo(1);
  foo(10);
}
```

Path Coverage

Each CFG path must be executed at least once

Coverage:
 # executed path
 # paths

```
void foo (z) {
  var x = 10;
  if (z++ < x) {
    x=+ z;
  }
}
```

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

What is the minimal number of test cases to achieve full branch coverage?

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

What is the minimal number of test cases to achieve full branch coverage?

Two, for example:

1. ***cond1: true, cond2: true***
2. ***cond1: false, cond2: false***

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

What about path coverage?

Branch vs. Path Coverage

```
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

What about path coverage?

Four:

1. ***cond1: true, cond2: true***
2. ***cond1: false, cond2: true***
3. ***cond1: true, cond2: false***
4. ***cond1: false, cond2: false***

What about infeasible paths?

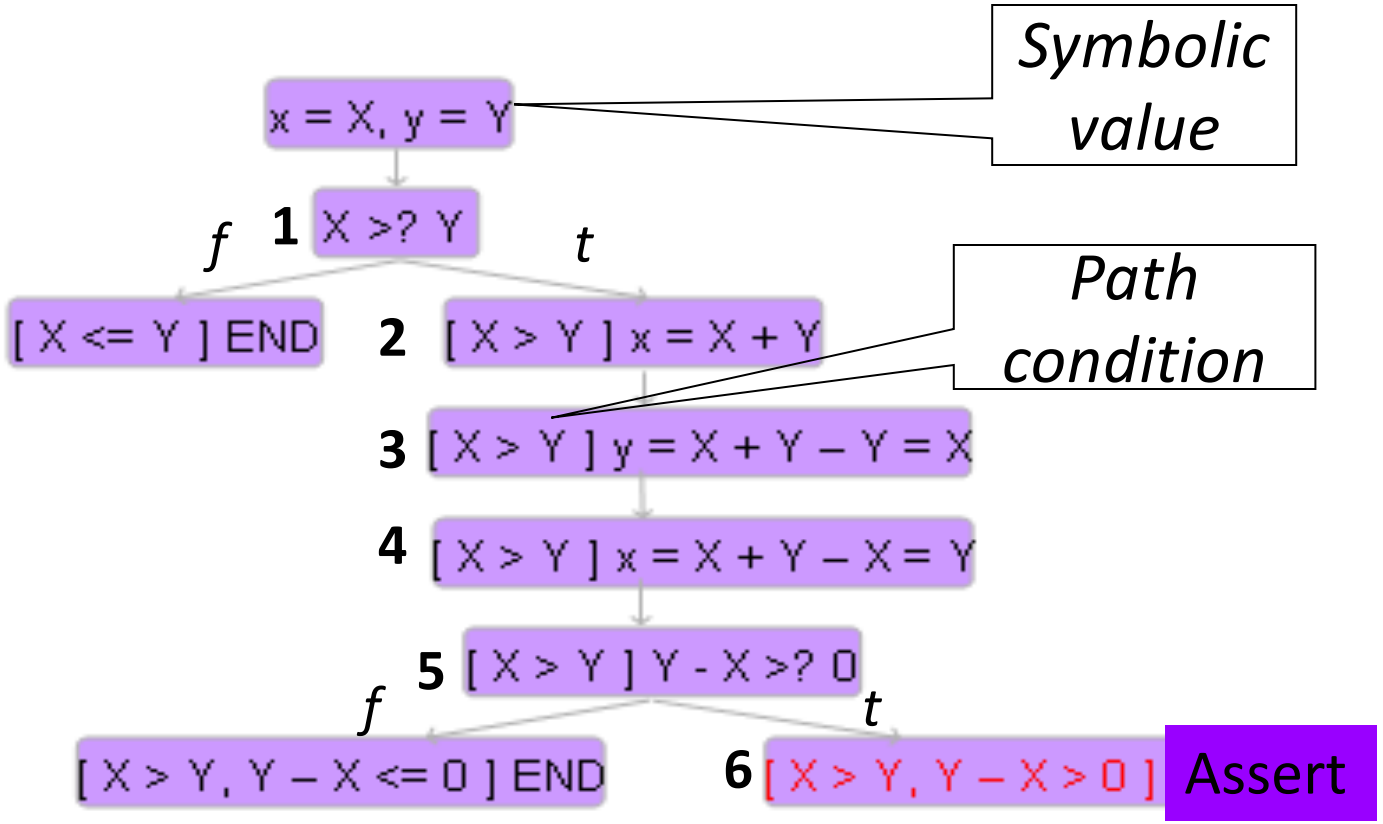
```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false); //error
7: }
```

How many feasible paths are in this code?

Is line 6 reachable?

Symbolic Execution by Example

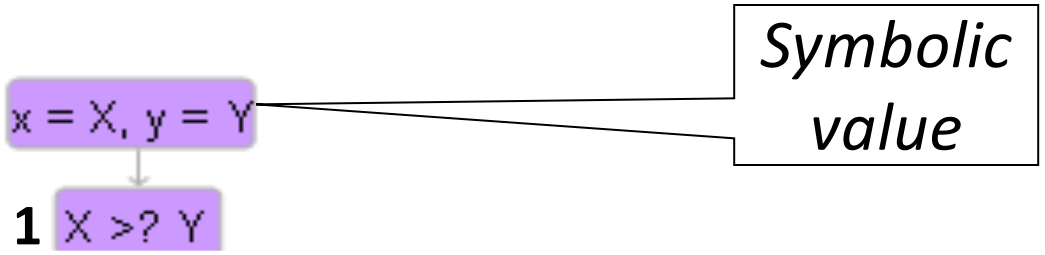
```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```



Is the assert (line 6) reachable?

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```

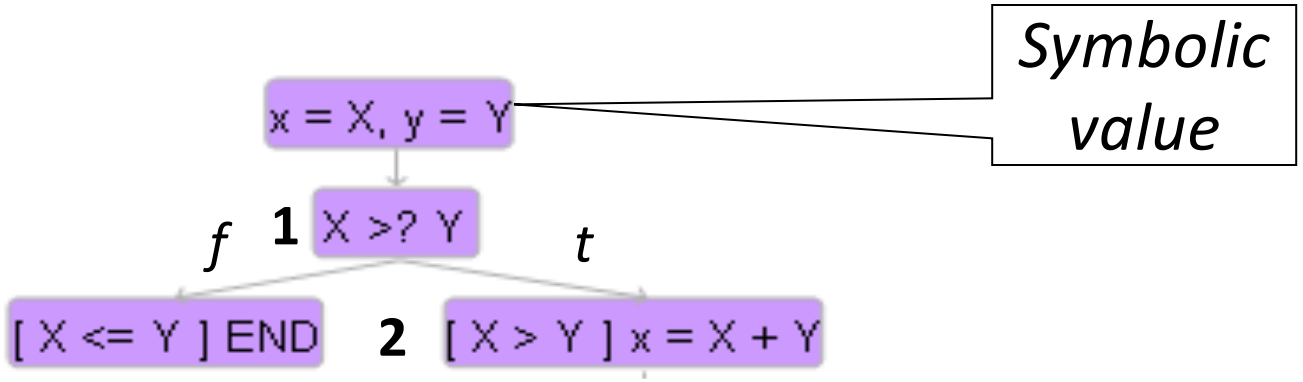


Is the assert (line 6) reachable?

was: x = X, y = Y
1: X > Y

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```



Best practice when done manually: write all values in each tree node, to avoid confusion (both x and y, like below)!

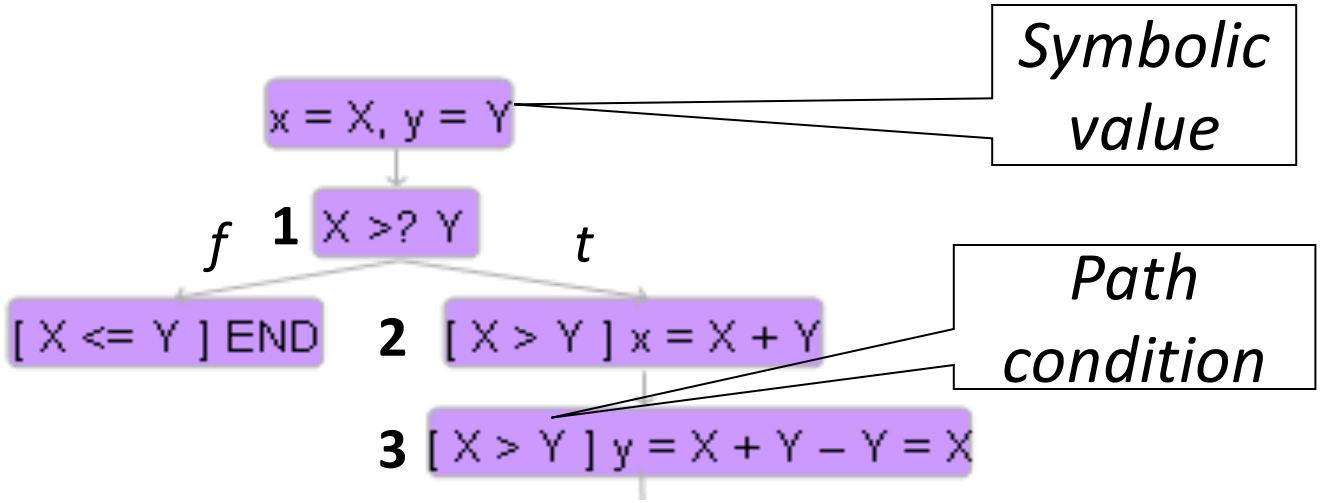
Is the assert (line 6) reachable?

was: x = X, y = Y

2: x = X+Y, y=Y

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```



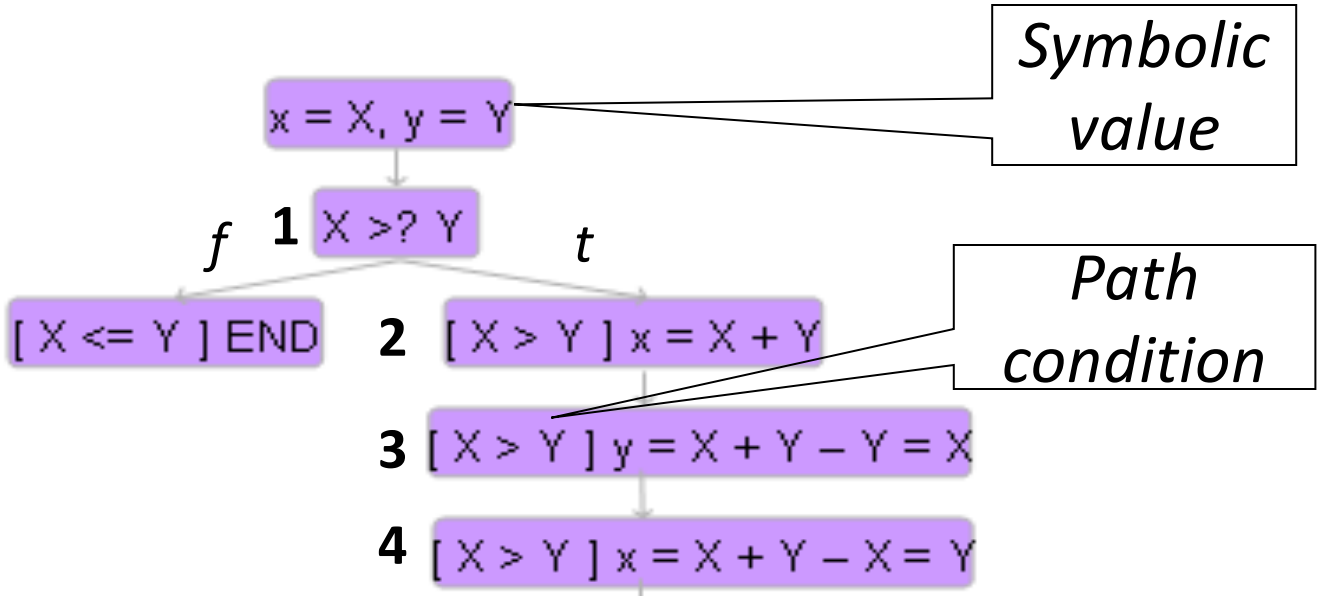
Is the assert (line 6) reachable?

Was: x = X+Y, y=Y

3: x = X+Y, y=X

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```



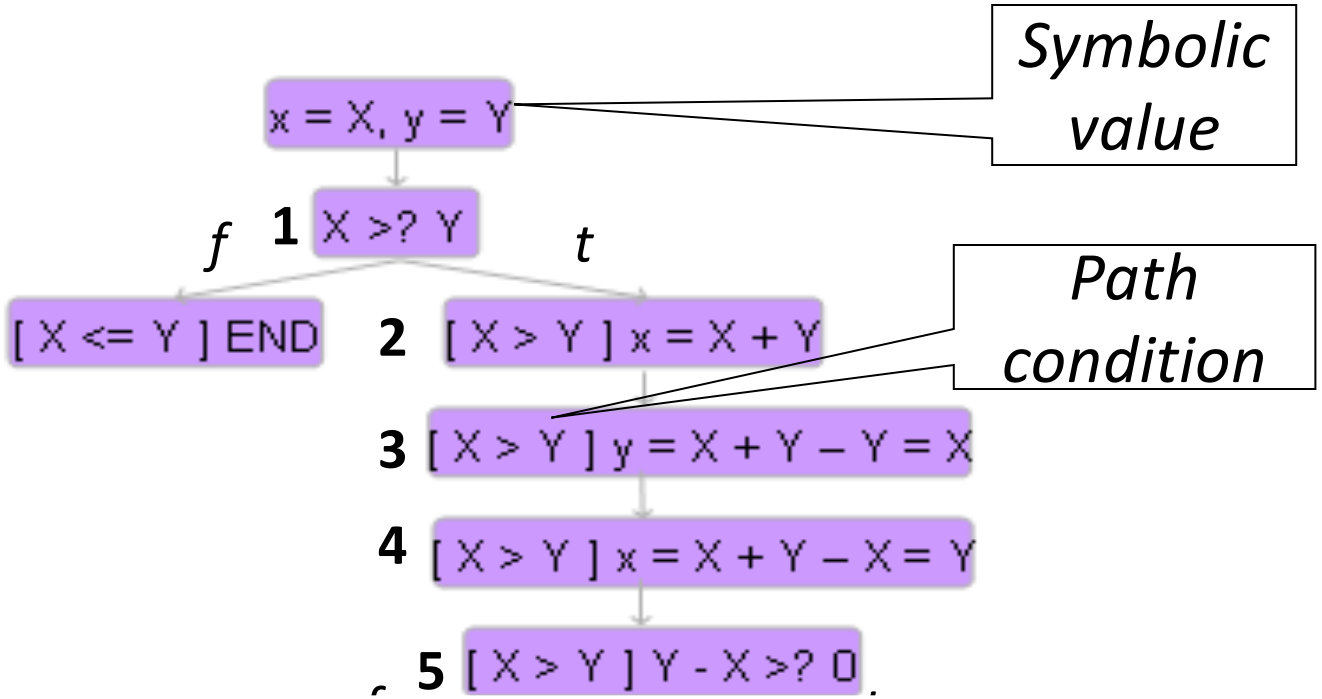
Is the assert (line 6) reachable?

Was: x = X+Y, y=X

4: x = Y, y=X

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```

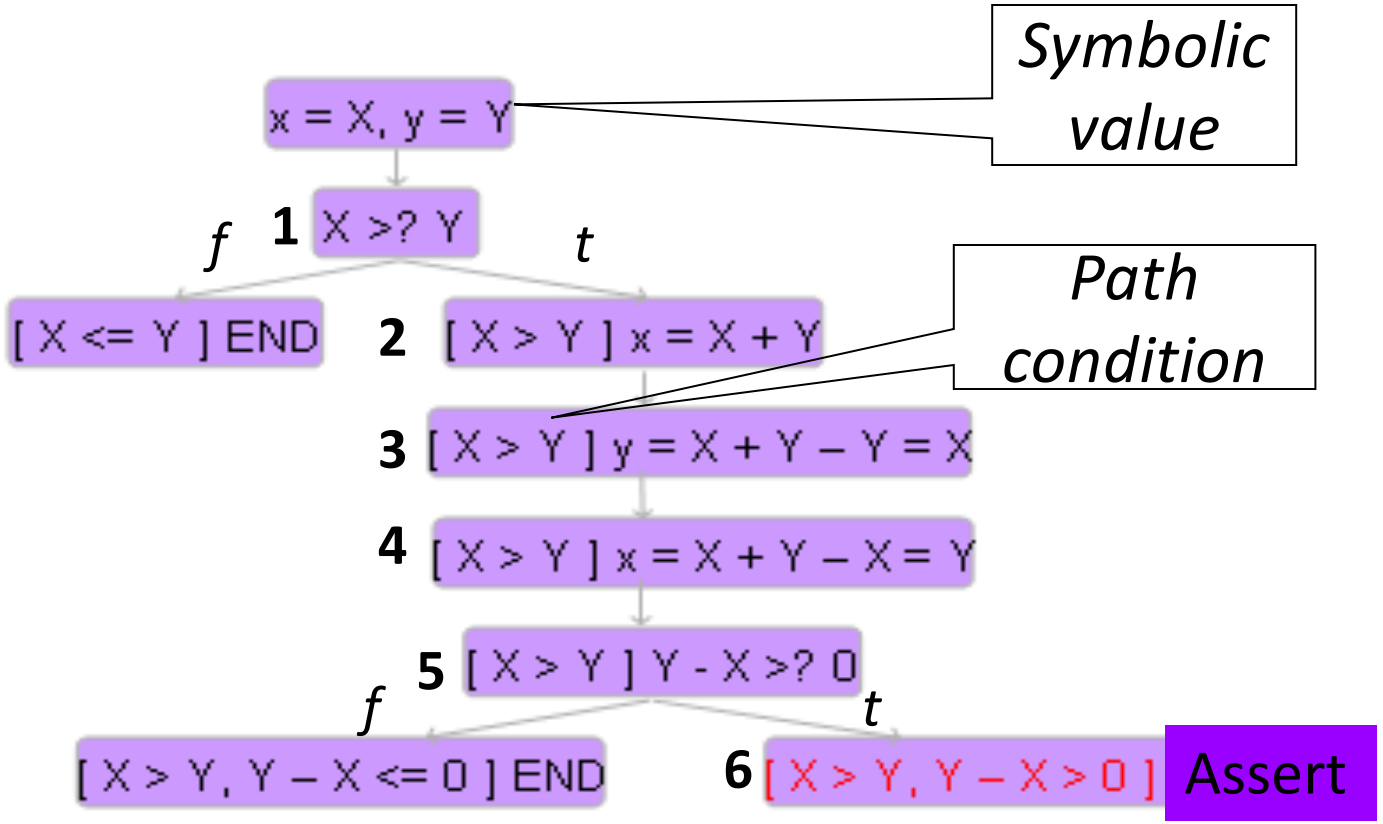


Is the assert (line 6) reachable?

5: Y-X > 0?

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
7: }
```

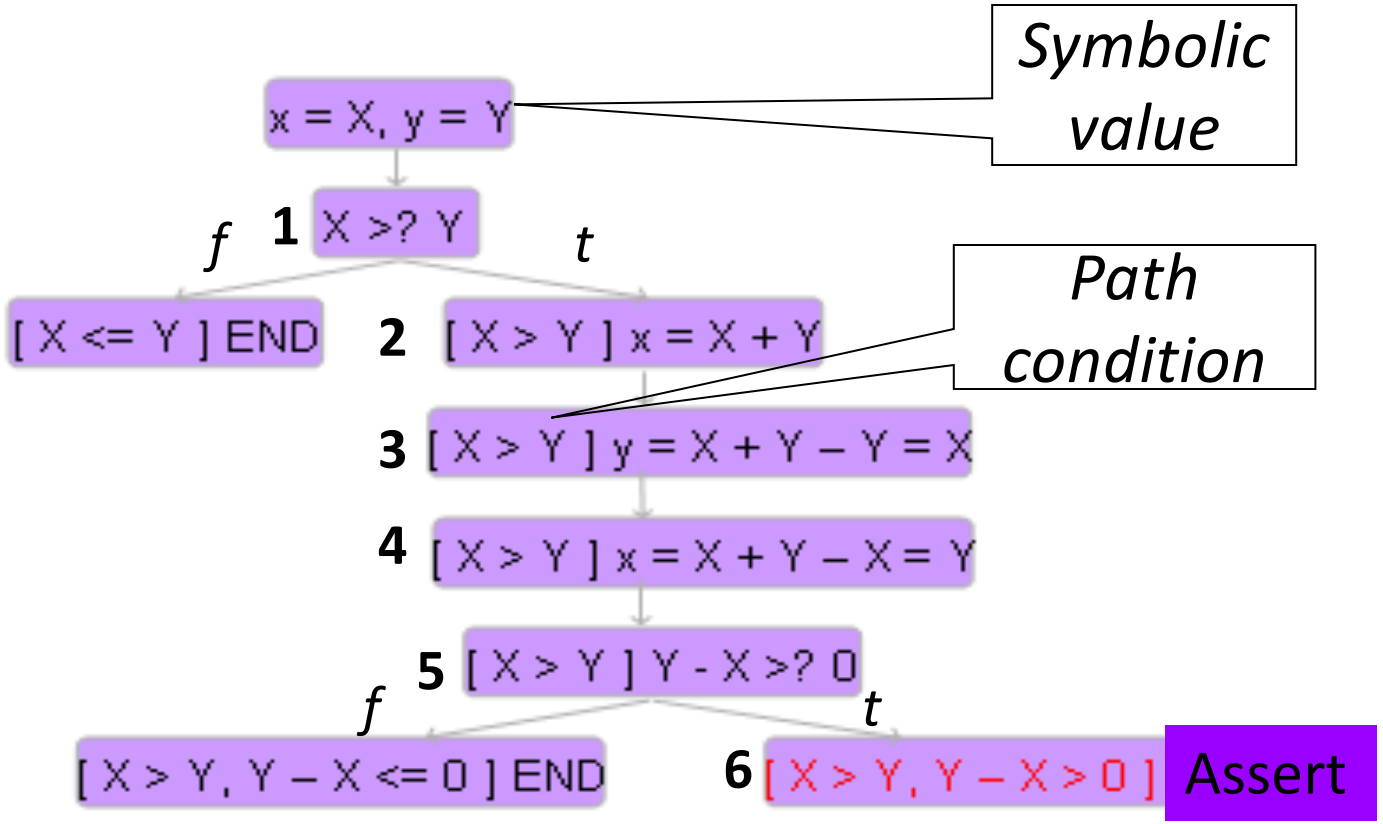


Is the assert (line 6) reachable?

Condition for 6: X>Y & Y-X > 0

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:       assert(false);
7: }
```

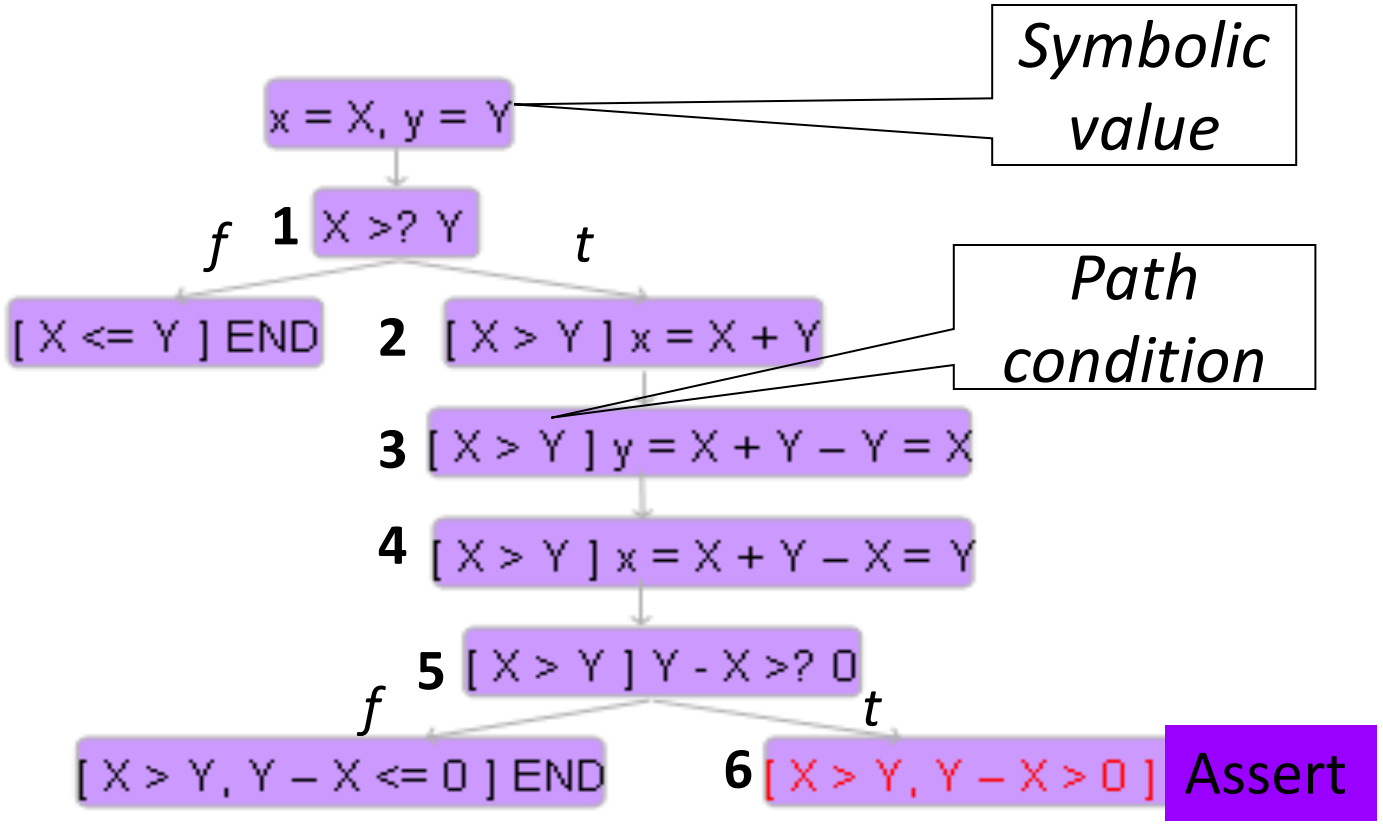


Three paths:

(1) X<=Y (2) X>Y (3) X>Y, Y> X

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:       assert(false);
7: }
```



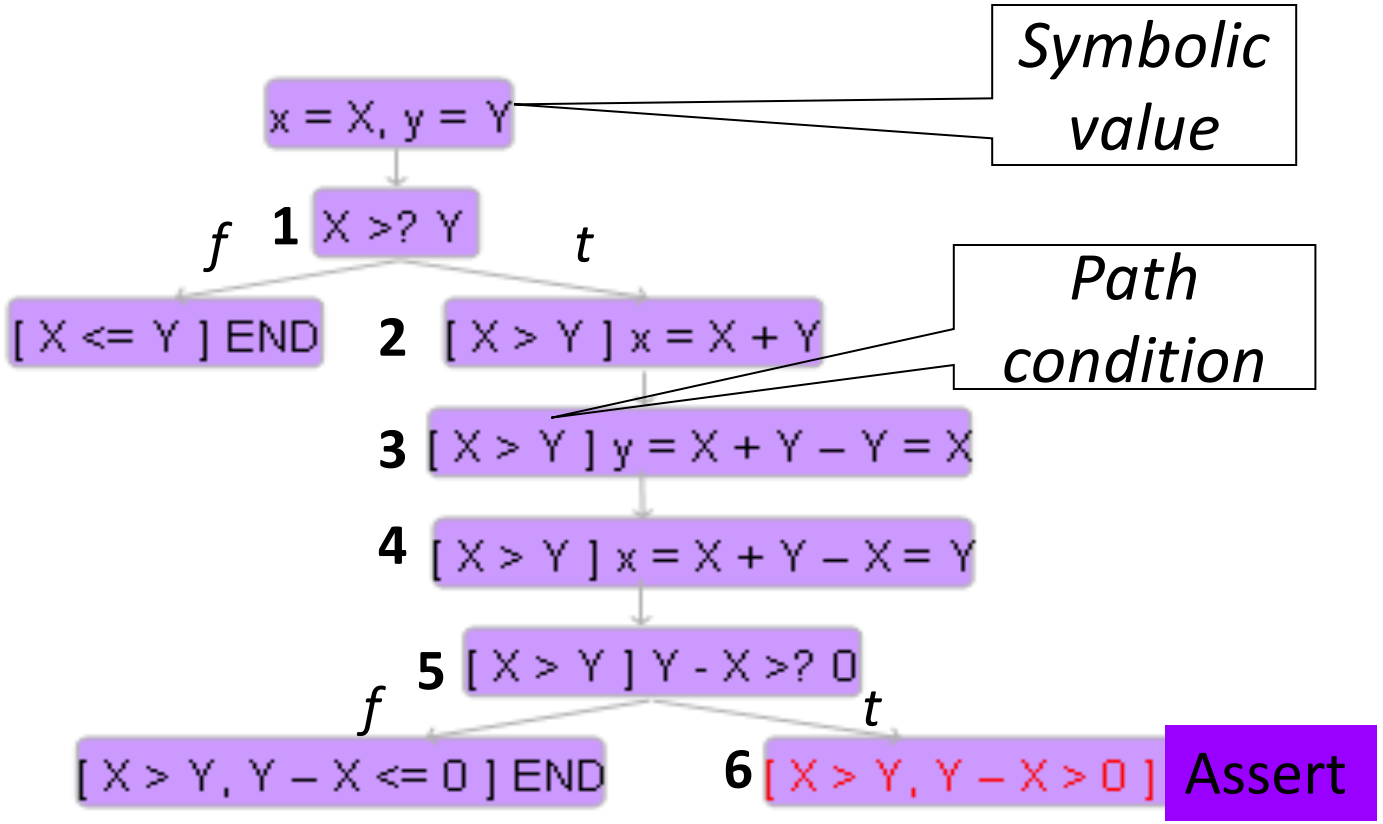
Three paths:

(1) X<=Y (2) X>Y — feasible

(3) X>Y, Y> X — infeasible

Symbolic Execution by Example

```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:       assert(false);
7: }
```



Is the assert (line 6) reachable?

NO!

Applications of Symbolic Execution

- Guiding the test input generation to cover all branches
- Identifying infeasible program paths
- Security testing
- …

Limitations of Symbolic Execution

- Expensive
  - Executing all feasible program paths is exponential in the number of branches
  - Does not scale to large programs
- Problem with handling loops
  - often *unroll* them up to a certain depth rather than dealing with termination or loop invariants
- (SAT / SMT solvers for conditions…)

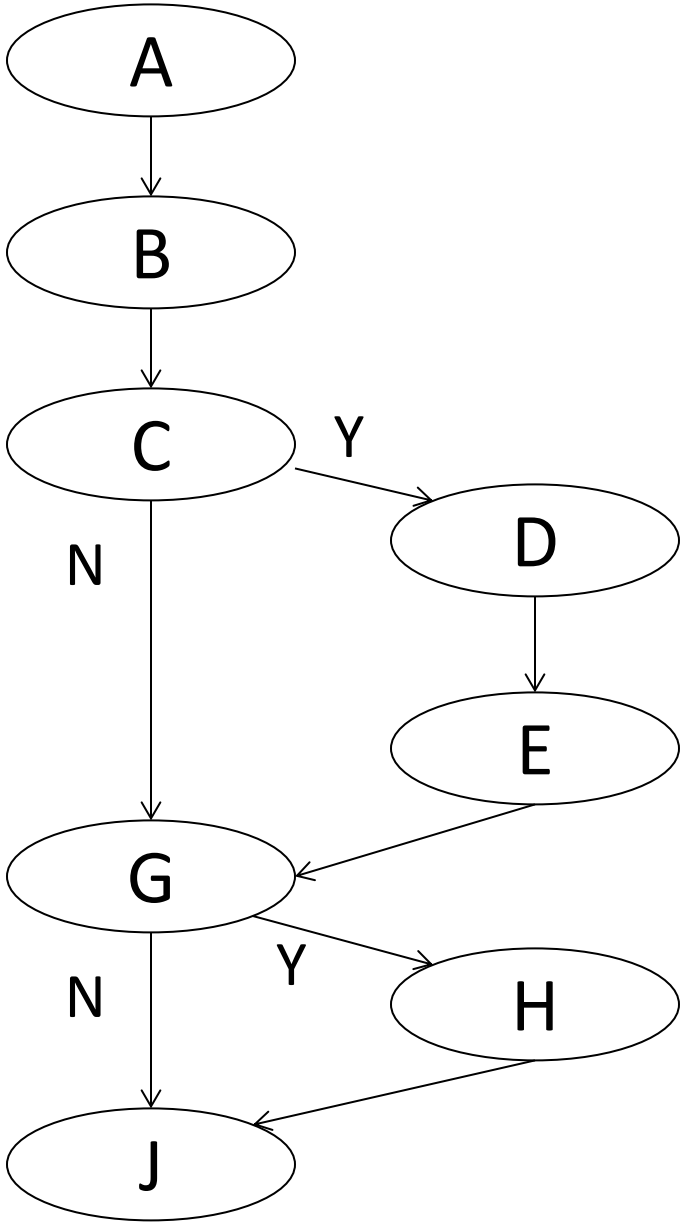
Exercise – 1/2

```
A:   void f(int x) {
B:       int y = x;
C:       if (x ≥ 10) {
D:           x = x – 10;
E:           y++;
F:       }
G:       if (x ≥ 5) {
H:           x++;
I:       }
J:       print(x,y);
K:   }
```

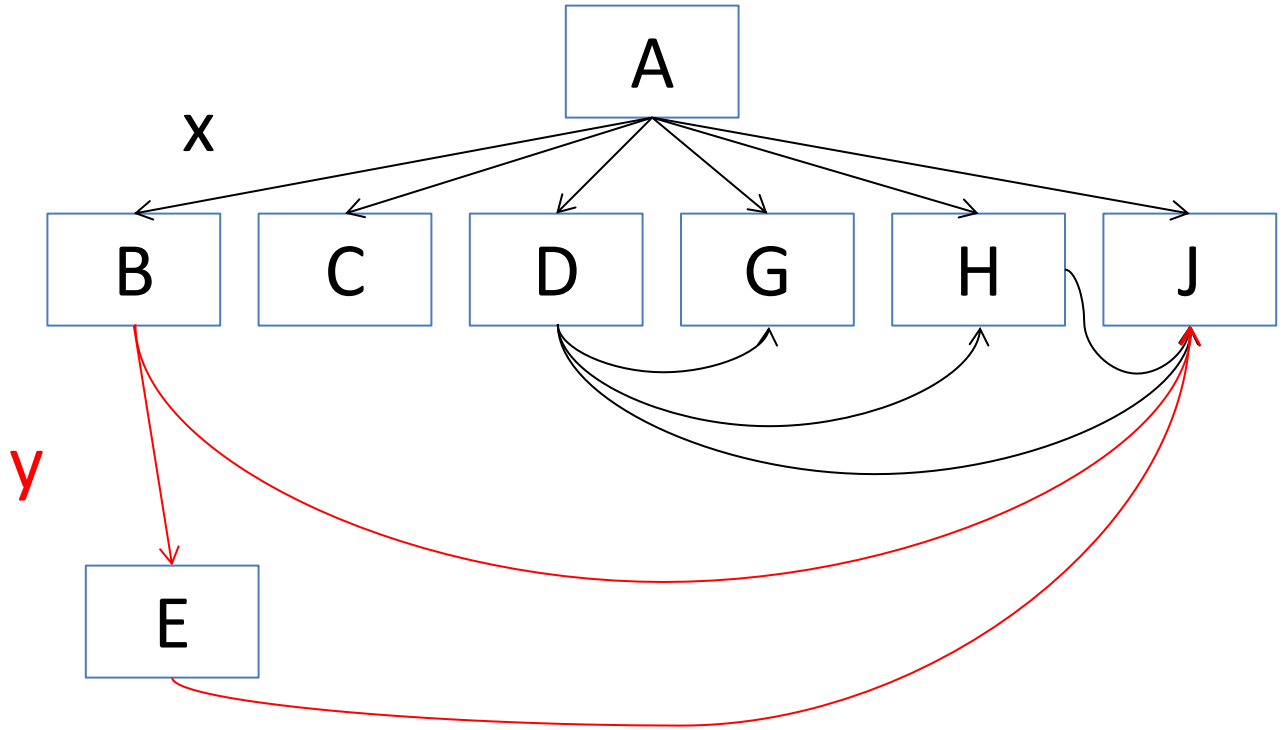
1. Draw the Control Flow Diagram (CFG) for this function (nodes are statements).
2. Draw the Data Flow Diagram of this function. **Do not** include control flow edges, just data flow

- How many control flow edges are in your graph in #1?
- How many data flow edges are in your graph in #2 for the variable x?
- How many data flow edges are in your graph in #2 for the variable y?

```
A:    void f(int x) {
B:        int y = x;
C:        if (x ≥ 10) {
D:            x = x – 10;
E:            y++;
F:        }
G:        if (x ≥ 5) {
H:            x++;
I:        }
J:      print(x,y);
K:    }
```



```
A:    void f(int x) {
B:        int y = x;
C:        if (x ≥ 10) {
D:            x = x – 10;
E:            y++;
F:        }
G:        if (x ≥ 5) {
H:            x++;
I:        }
J:        print(x,y);
K:    }
```



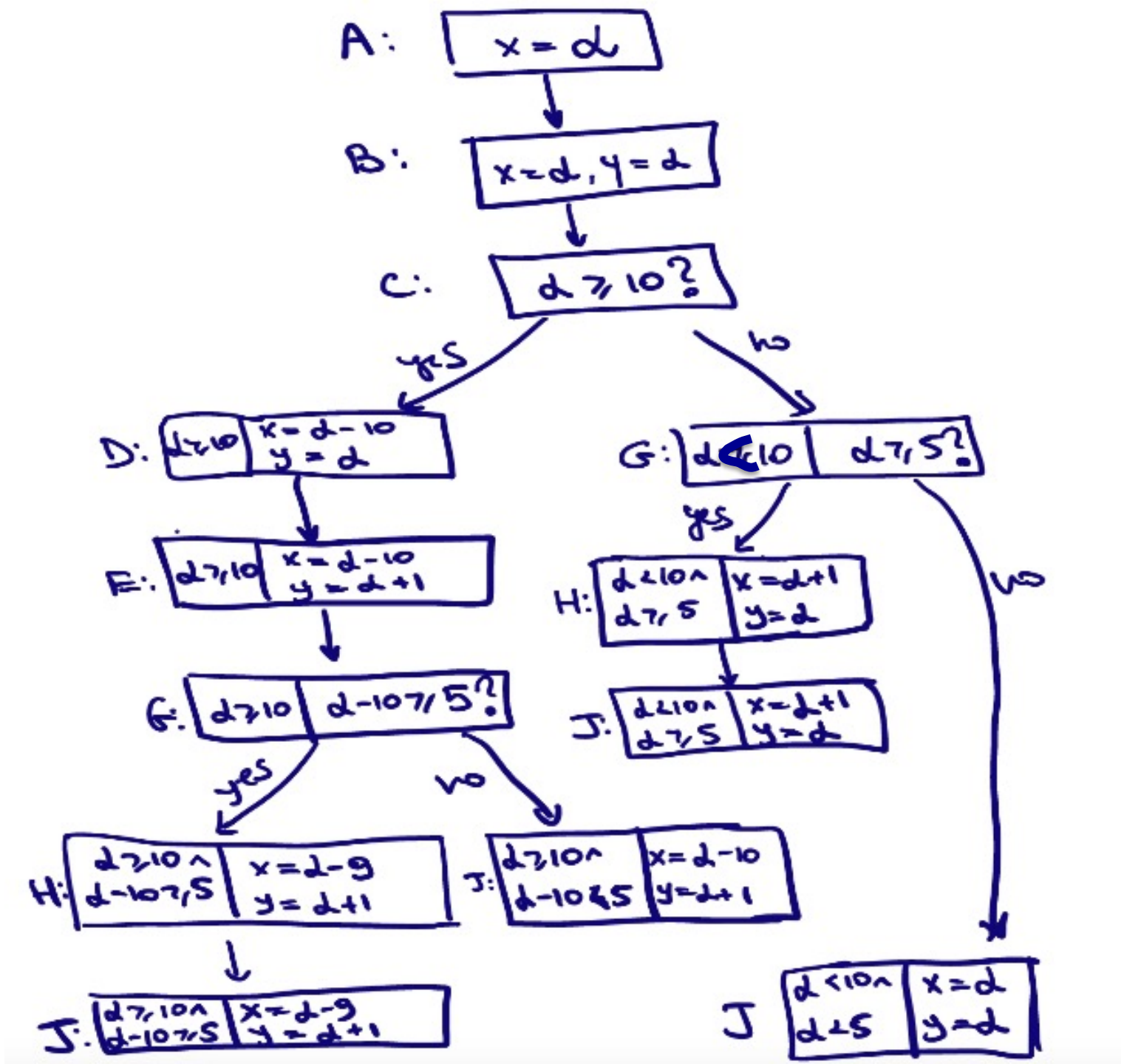
Exercise – 2/2

```
A:   void f(int x) {
B:       int y = x;
C:       if (x ≥ 10) {
D:           x = x – 10;
E:           y++;
F:       }
G:       if (x ≥ 5) {
H:           x++;
I:       }
J:       print(x,y);
K:   }
```

Execute the function symbolically, given a symbolic variable α

- How many feasible + unfeasible paths does this program have in total?
- What is the path condition for each of these paths?
- How many feasible paths does this program have?

```
A:   void f(int x) {
B:       int y = x;
C:       if (x ≥ 10) {
D:           x = x – 10;
E:           y++;
F:       }
G:       if (x ≥ 5) {
H:           x++;
I:       }
J:     print(x,y);
K:   }
```



Home Exercise

```
A:   void f(int x, y) {
B:       if (x>0) {
C:           y = x/2;
D:       }
E:       else {
F:           y = 5;
G:       }
H:       if (x*y>0) {
I:           z = 10;
J:       }
K:       else {
L:           z = 20;
M:       }
N:       x = y + z;
```

Execute the function symbolically, given a symbolic variables α and β

- How many feasible + unfeasible paths does this program have in total?
- What is the path condition for each of these paths?
- How many feasible paths does this program have?