Principles of Program Analysis

Julia Rubin

January 22, 2020

Agenda

- Basics of Analysis
  - Control flow
  - Call graph
  - Data Flow Analysis
  - Symbolic Execution
- Presentation Tips

Program Analysis: Reasoning About Code

- The process of automatically analyzing the behavior of programs

- 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

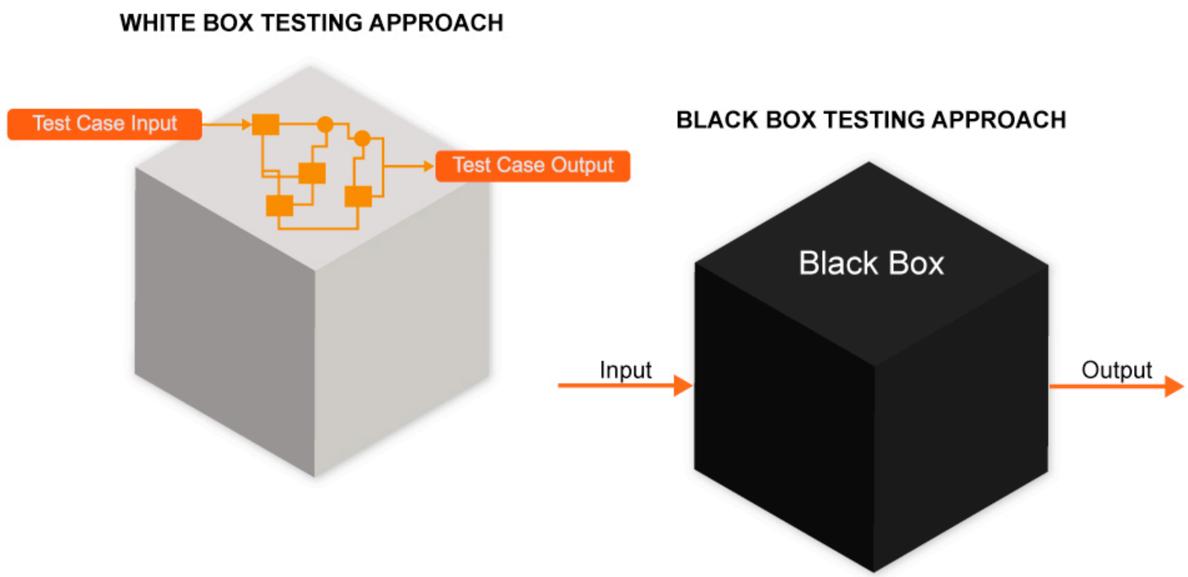
<table>
<tr><th></th><th>Static Analysis
(without running the code)</th><th>Dynamic Analysis
(at runtime)</th></tr>
<tr><td rowspan="2">White Box
(code internals)</td><td>– Checking that each method has a comment
– Style checks
– Symbolic execution
– Model checking</td><td>– Unit testing
– Mock testing
– Debugging</td></tr>
<tr><td colspan="2">– Counting lines of code for a method
– Checking that each called method is defined
– Null dereference (security checks)
– Tracking data flows ...</td></tr>
<tr><td>Black Box
(input-output)</td><td></td><td>– Integration testing
– User acceptance testing
– Profiling
– Monitoring</td></tr>
</table>

Types of Analysis

<table>
<tr><th></th><th>Static Analysis
(without running the code)</th><th>Dynamic Analysis
(at runtime)</th></tr>
<tr><td rowspan="2">White Box
(code internals)</td><td>– Checking that each method has a comment
– Style checks
– Symbolic execution
– Model checking</td><td>– Unit testing
– Mock testing
– Debugging</td></tr>
<tr><td colspan="2">– Counting lines of code for a method
– Checking that each called method is defined
– Null dereference (security checks)
– Tracking data flows ...</td></tr>
<tr><td>Black Box
(input-output)</td><td></td><td>– Integration testing
– User acceptance testing
– Profiling
– Monitoring</td></tr>
</table>

White-Box Analysis

Based on internal paths, code structures, and implementation of the software

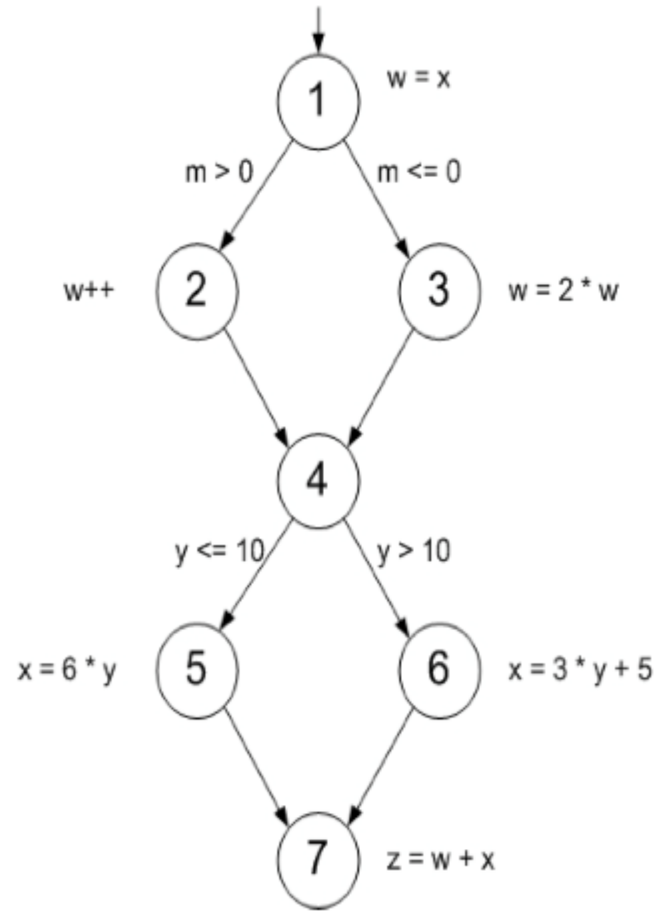


Analysis Primitives

Modeling Software

Graphs! E.g.,

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

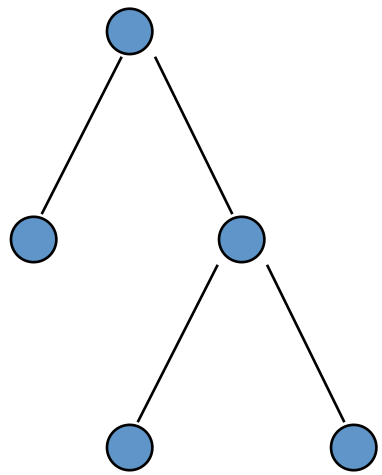


Graphs

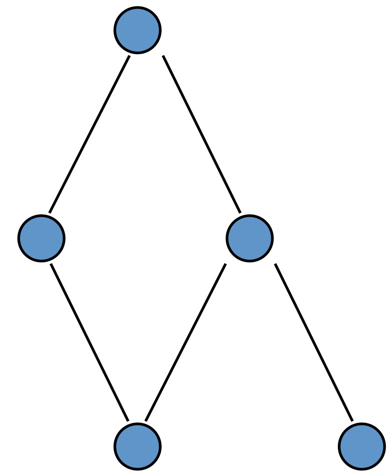
- A graph, G = (N, E), is an ordered pair consisting of
  - a set of nodes N
  - a set of edges E = $\{(n_i, n_j)\}$
  - if the pairs in E are ordered, then G is called a *directed graph*
  - if not, it is called an *undirected graph*

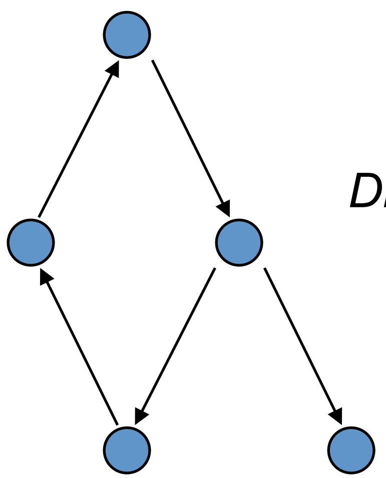
Graphs and Trees

tree
(acyclic graph)

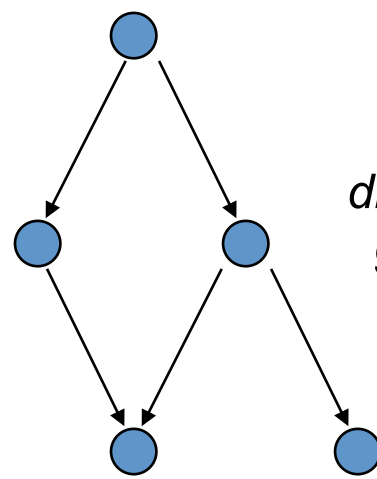


cyclic undirected
graph





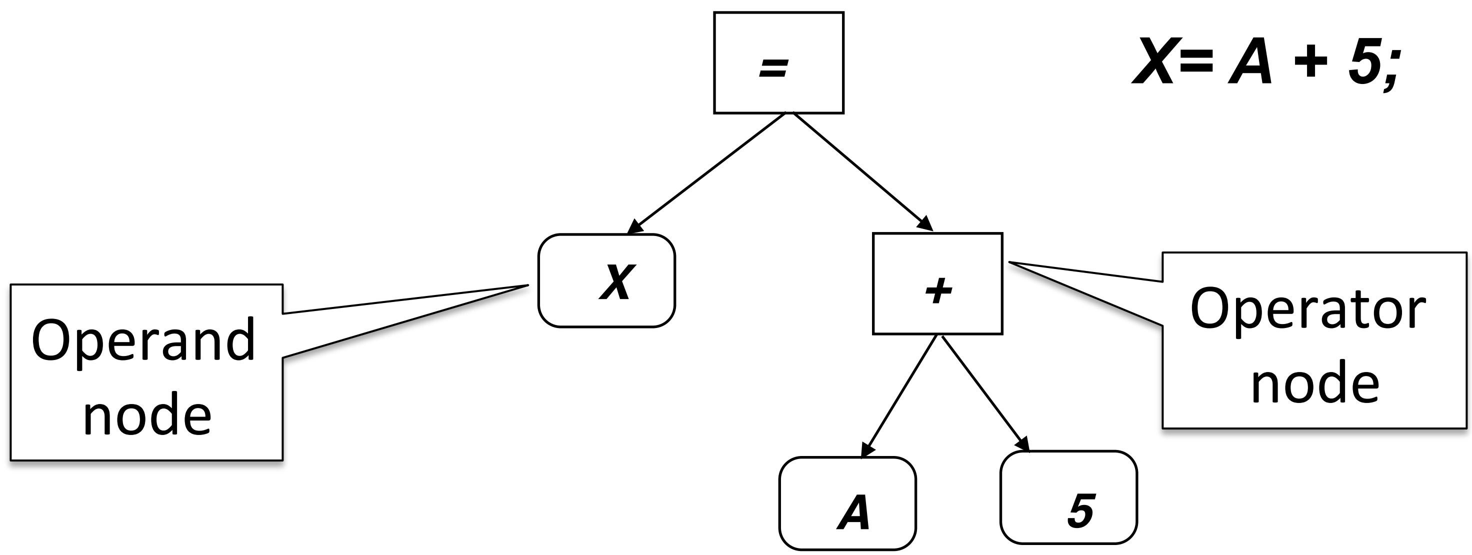
Directed cyclic
graph

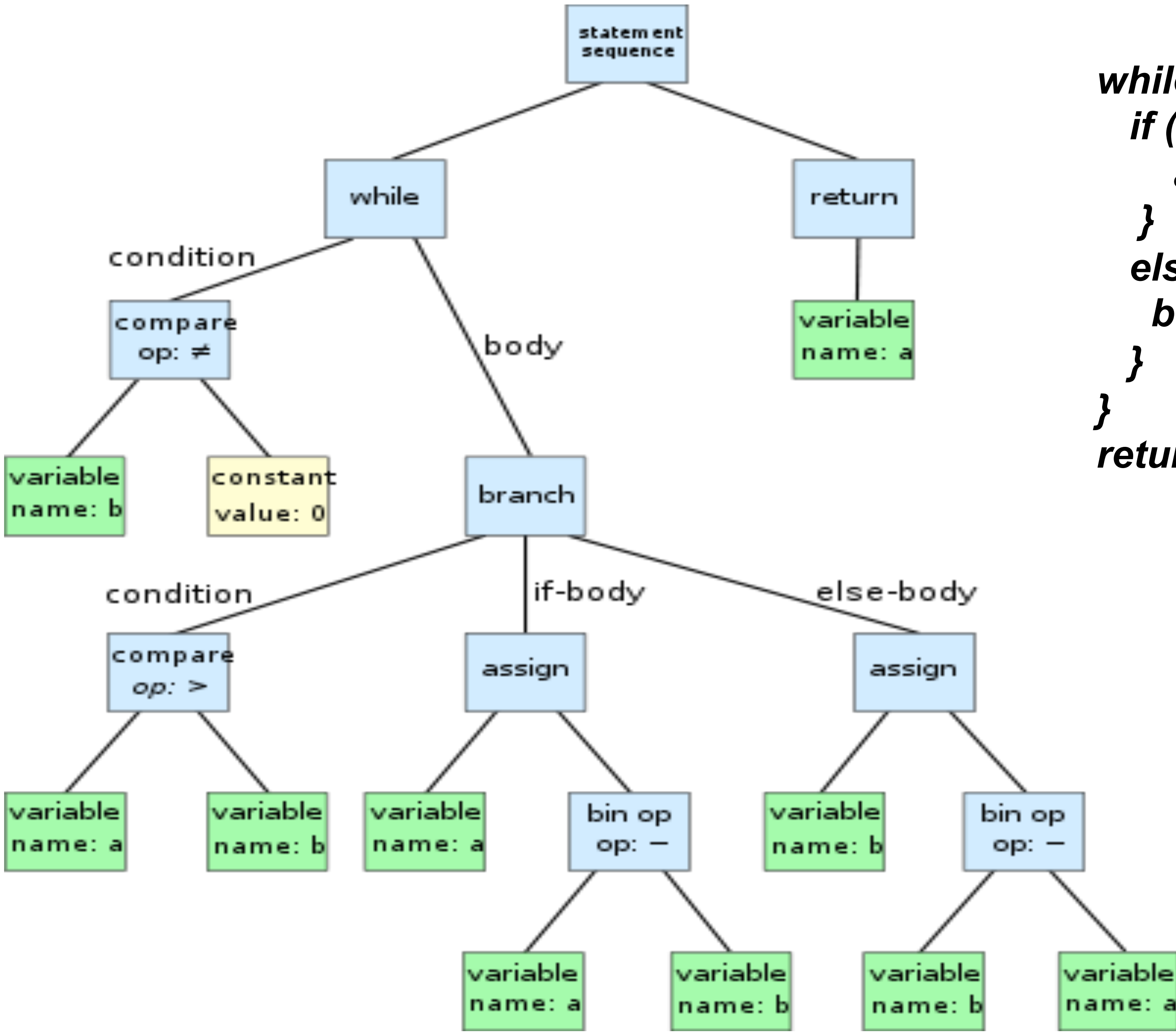


directed acyclic
graph (DAG)

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

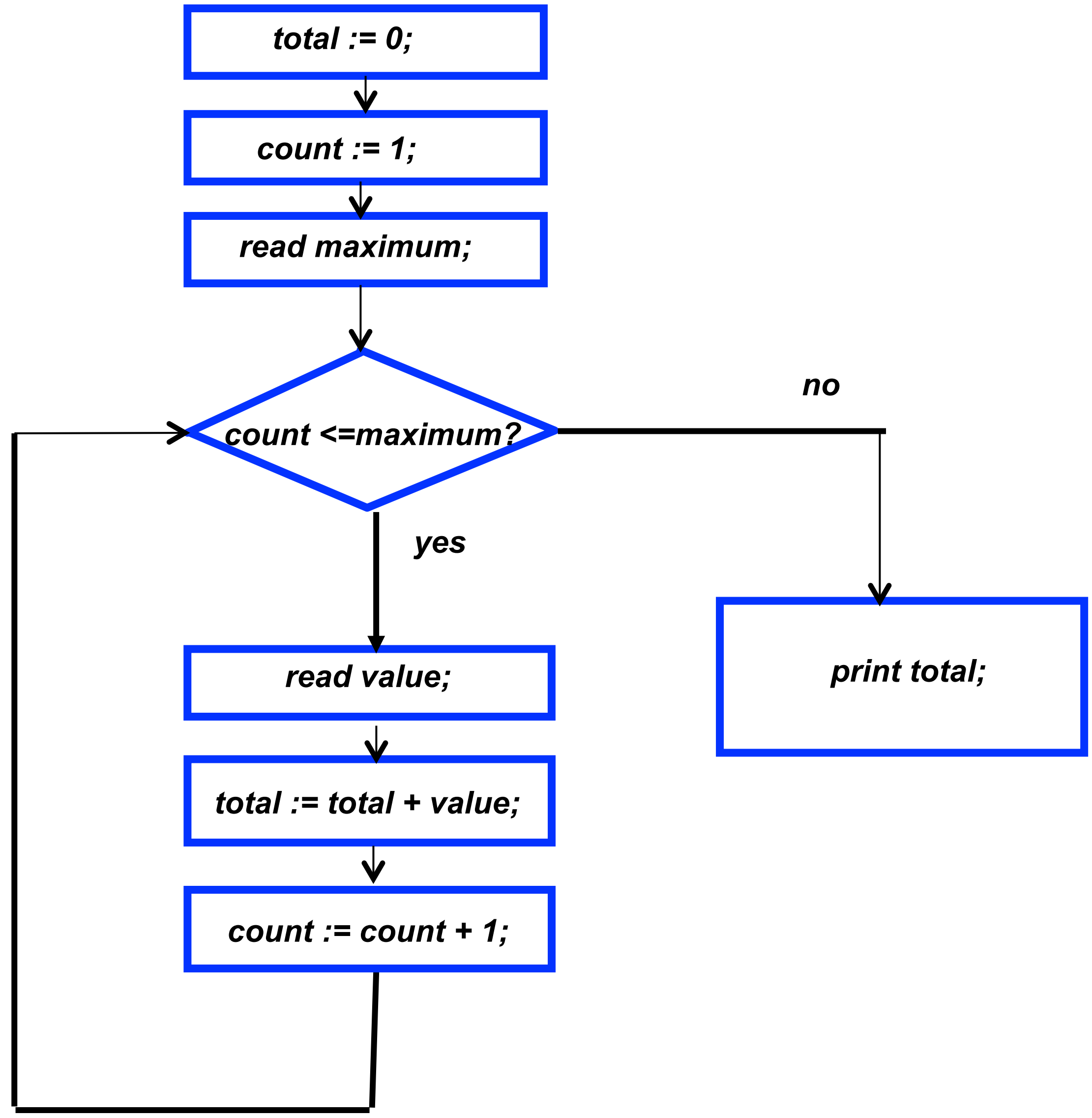
    read value;

    total := total + value;

    count: = count + 1;

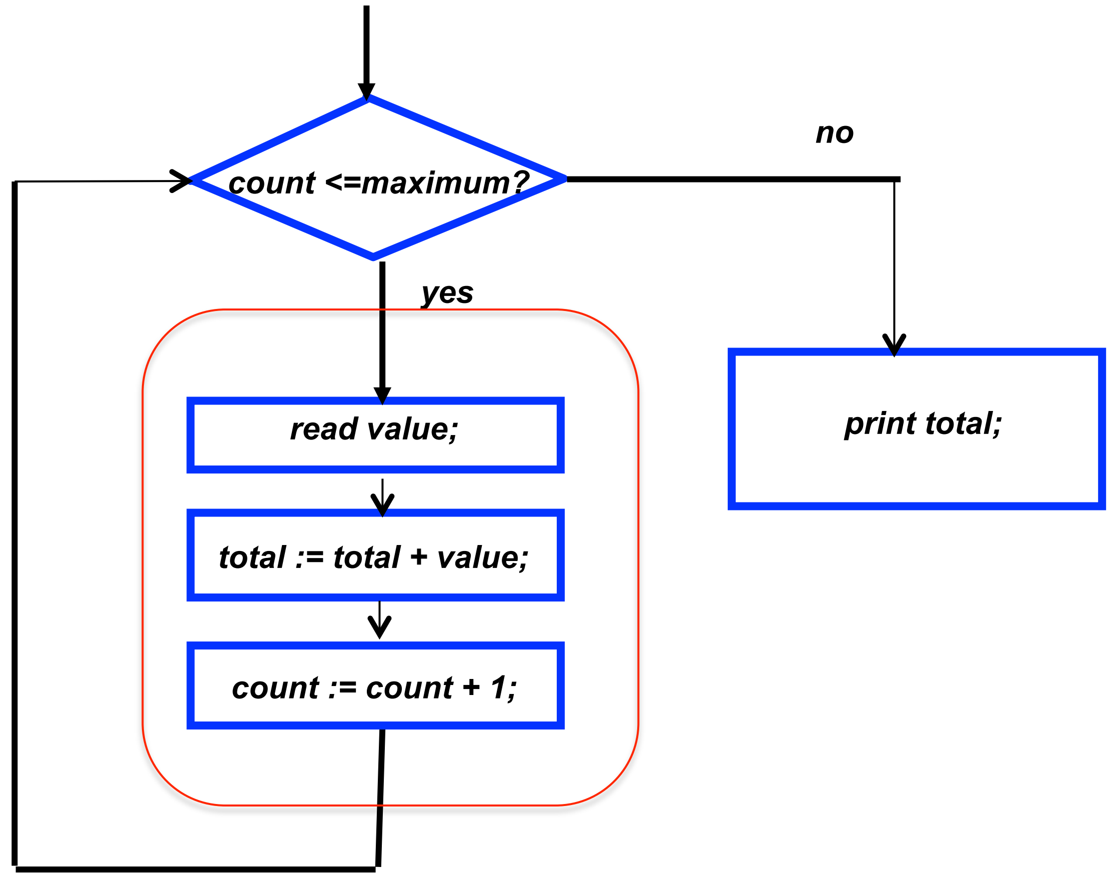
endwhile;

print total;



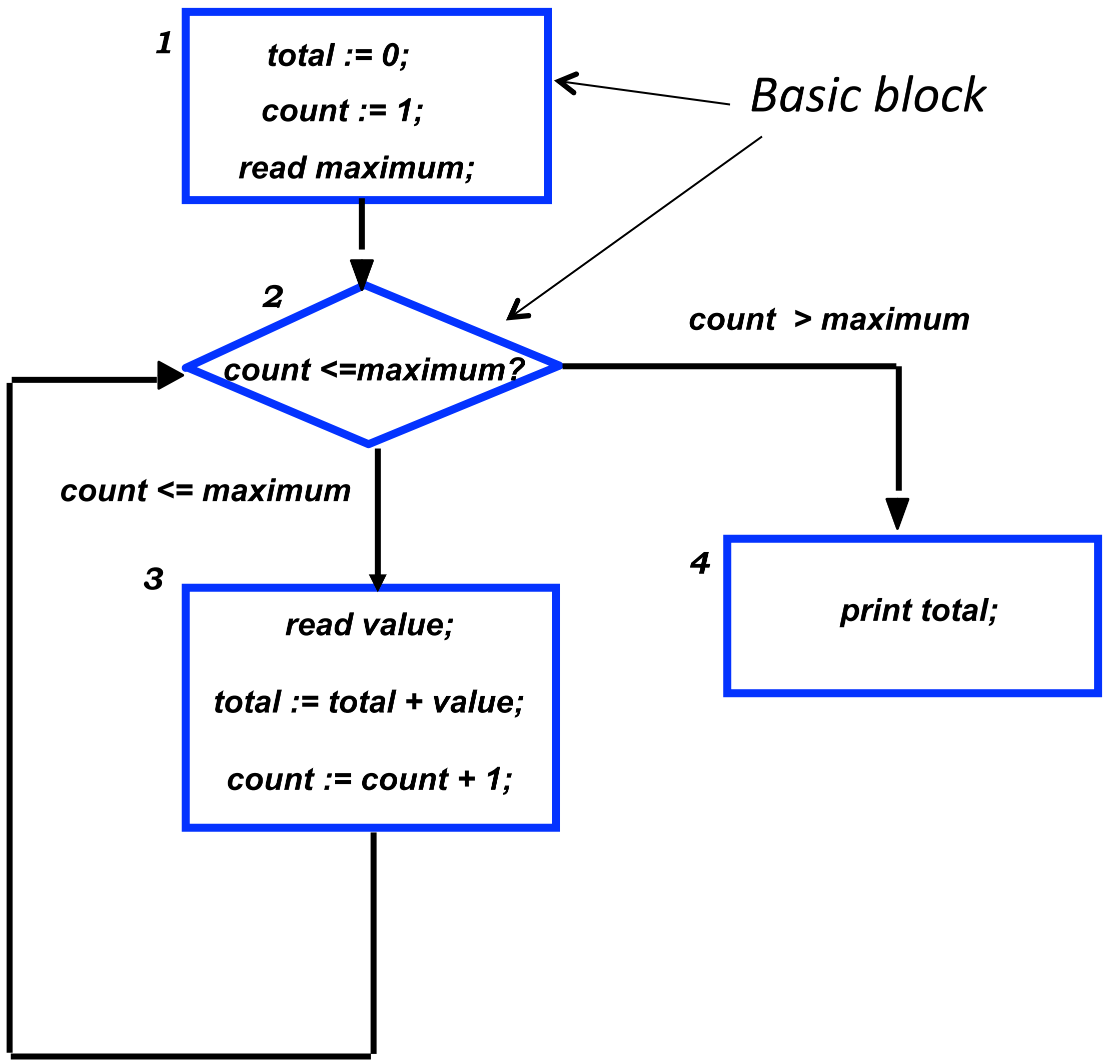
Basic Block

- 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;
```

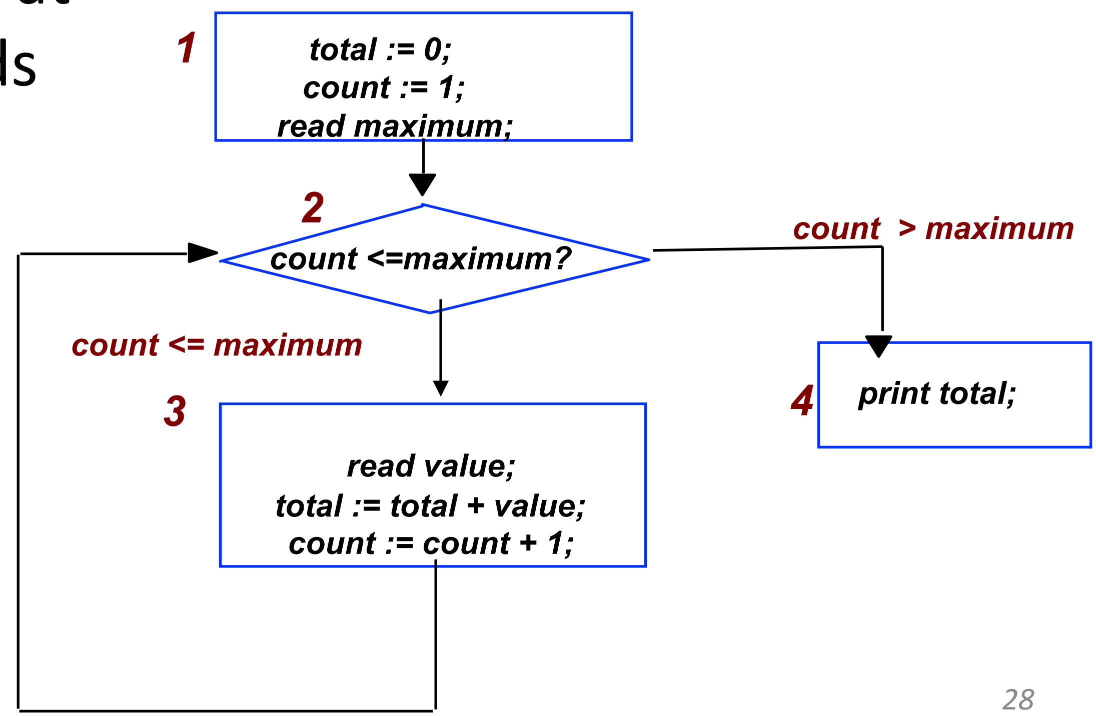


Control Flow Graph (CFG) – Definition

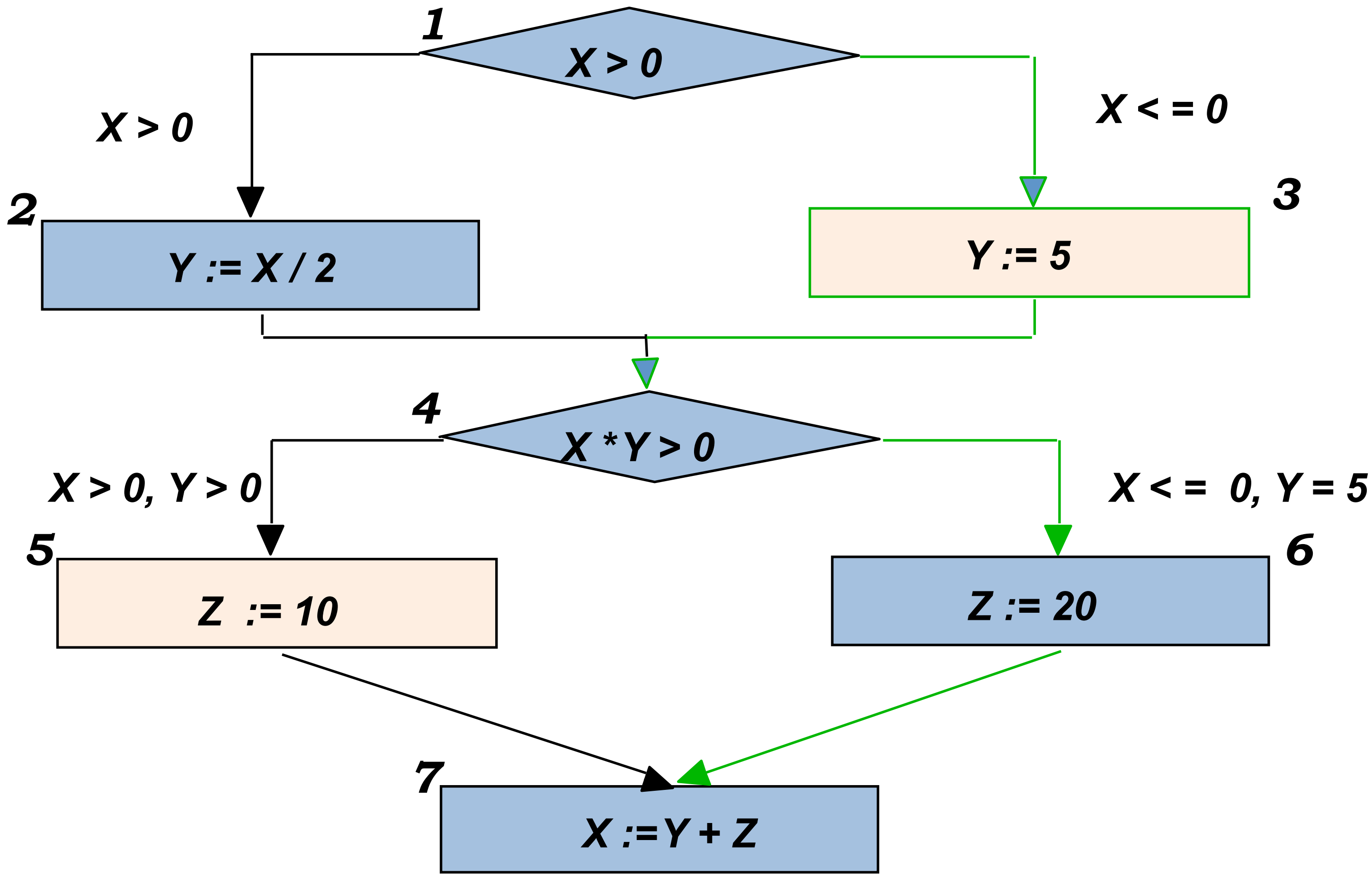
- Nodes N: statements or (more often) basic blocks
- Directed edges E: *potential* transfer of control from the end of one region directly to the beginning of another
  - $E = \{ (n_i, n_j) \mid$ syntactically, the execution of n_j follows the execution of $n_i \}$
- Intraprocedural (within a method)

CFG Paths

- A subpath through a control flow graph: a sequence of nodes $n_k,\ldots,n_m$, such that for each n_i, $k \leq i < m$, (n_i, n_{i+1}) is an edge in the graph,
  - e.g., 2, 3, 2, 3, 2, 4
- a complete path starts at the start node and ends at the final node
  - e.g., 1, 2, 3, 2, 4



Infeasible paths



CFG ***overestimates*** *the executable behavior*

Dead and Unreachable Code

unreachable code

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

Never executed

dead code

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

‘Executed’, but irrelevant

CFG - Recap

- A directed graph where
  - Each node represents a statement or a basic block
  - Edges represent control flow
- Intraprocedural (within a method)
- Over-approximate possible flows

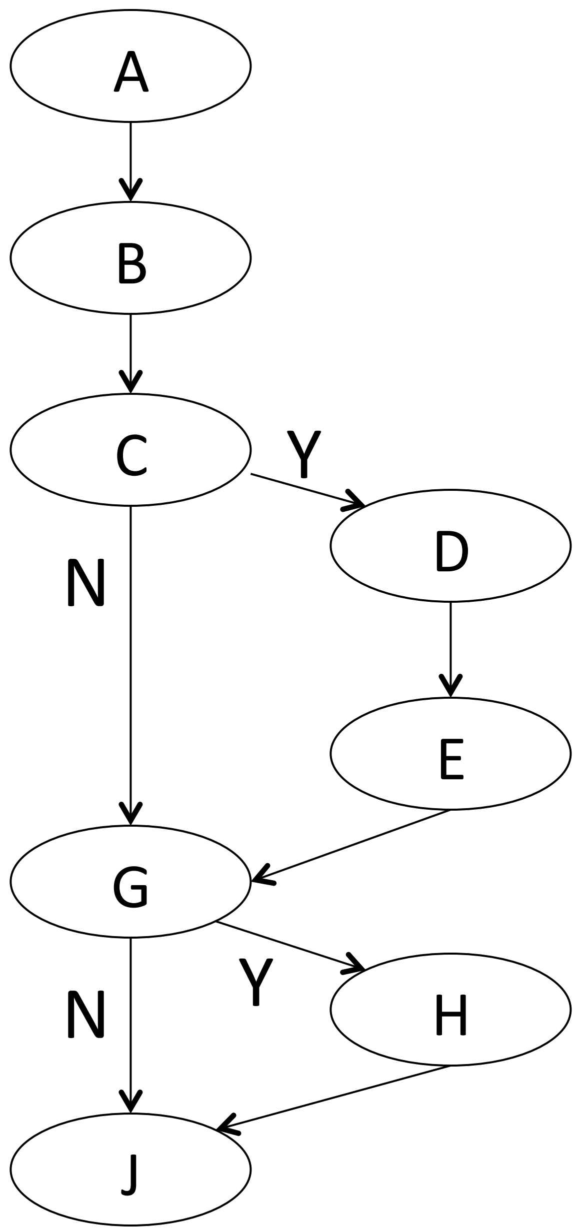
Benefits of CFG

- 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
  - S/W analysis

Exercise

- Draw control flow graph for this method

```
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:   }
```



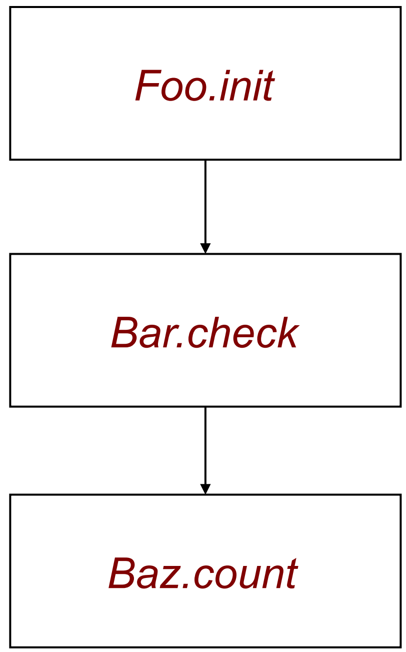

```
A:  void f(int x) {
B:      int y = x;
C:      if (x ≥ 10) {
D:          x = x – 10;
E:          y++;
F:      }
G:      if (x ≥ 5) {
H:          x++;
I:      }
J:      print(x,y);
K:  }
```

Call Graphs (Interprocedural CFG)

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

Example

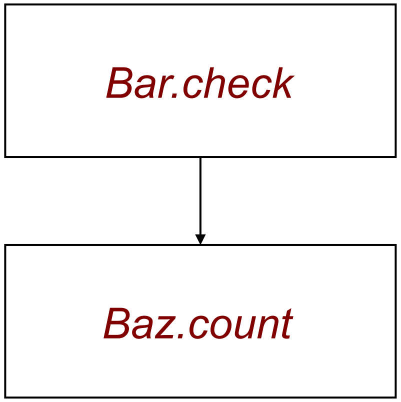
```
public class Foo {
   void init() {
     new Bar()).check();
   }
}

public class Bar {
   void check() {
      count();
    }
}

class Baz {
   void static count() {
      //do stuff
    }
}
```



Example

```java
public class Foo {
   void init() {
     new Bar()).check();
   }
}

public class Bar {
   void check() {
      count();
    }
}

class Baz {
   void static count() {
      //do stuff
    }
}
```

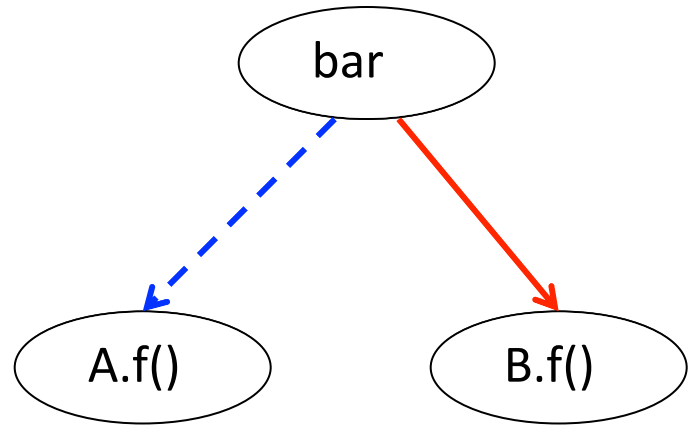
```java
public static void main(String args[]) {
        (new Bar()).check();
}
```



Call graph ***overestimates*** *the executable behavior*

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 degree of precision / scalability

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

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

CFG and Call Graph – Precision

- Flow Sensitivity
- Context Sensitivity

Flow Sensitivity

- Flow-sensitive: analysis captures the sequential order of execution of statements
- Flow-insensitive: analysis only concerned with what statements are present in the program, not with the order or the reachability of statements.
- Precise vs. expensive

```
void someMethod(int y)
{
    a();
    b();
}
```

b() is called after a()

```
void someMethod(int y)
{
    int x1 = 2 * y;
    int x2 = x_1 + 1;
}
```

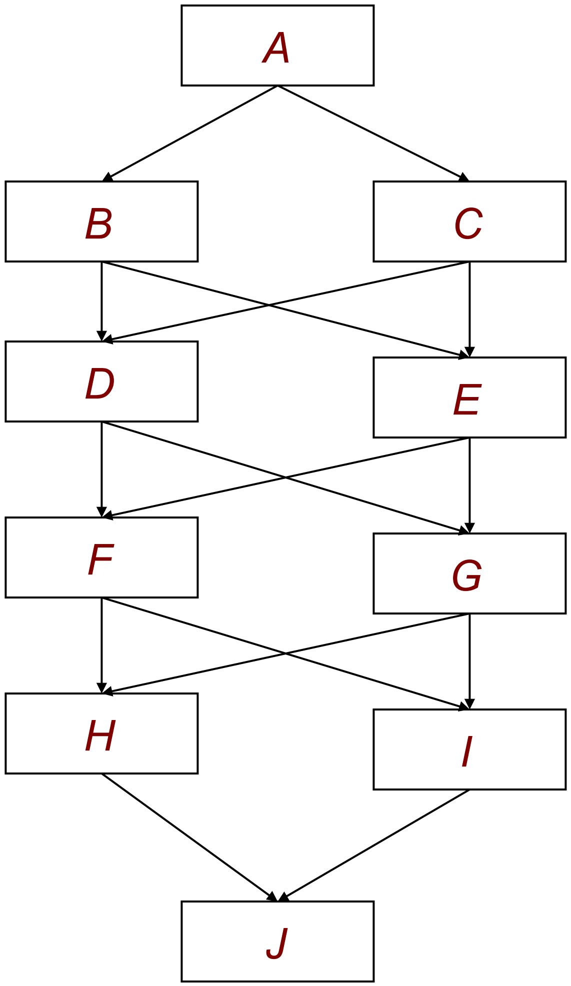
x1 is even, x2 is odd

Context Sensitivity

- For inter-procedural analyses
- Analyze a method separately for different calling contexts
  - call site sensitivity: call from different statements
  - object (a.k.a. allocation site) sensitivity: call for different receivers (objects on which the method is called)
- Precise vs. expensive

```
Class F {
   int data;
   void foo();
}
F a = new F(1);
F b = new F(2);
a.foo();
b.foo();
```

Context Sensitive - Expensive



1 context A

2 contexts AB AC

4 contexts ABD ABE ACD ACE

8 contexts …

16 calling contexts …

Static vs. Dynamic CFG / Call Graph

- Static:
  - Expensive analysis
  - Over-approximate the behaviors (if feasible)
  - Sometimes misses flows

- Dynamic
  - Expensive instrumentation (if feasible)
  - Accurate for the detected flows
  - Clearly under-approximates

Agenda

- Basics of Analysis
  - Control flow
  - Call graph
  - **Data Flow Analysis**
  - Symbolic Execution

- Presentation Tips

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

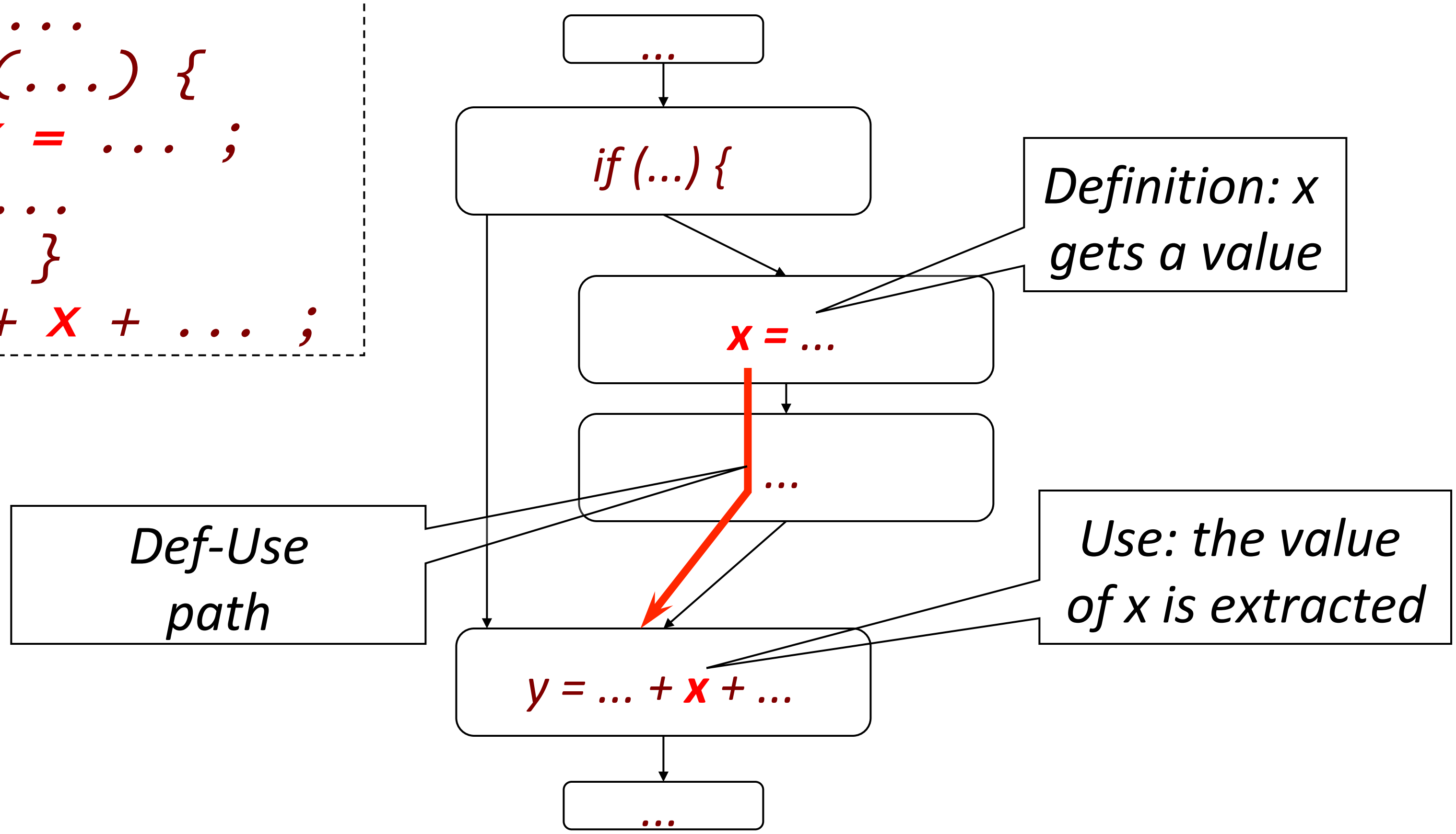
int x
int x = 5
x = 5
foo(int x)
x++

- 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 + . . . ;
```



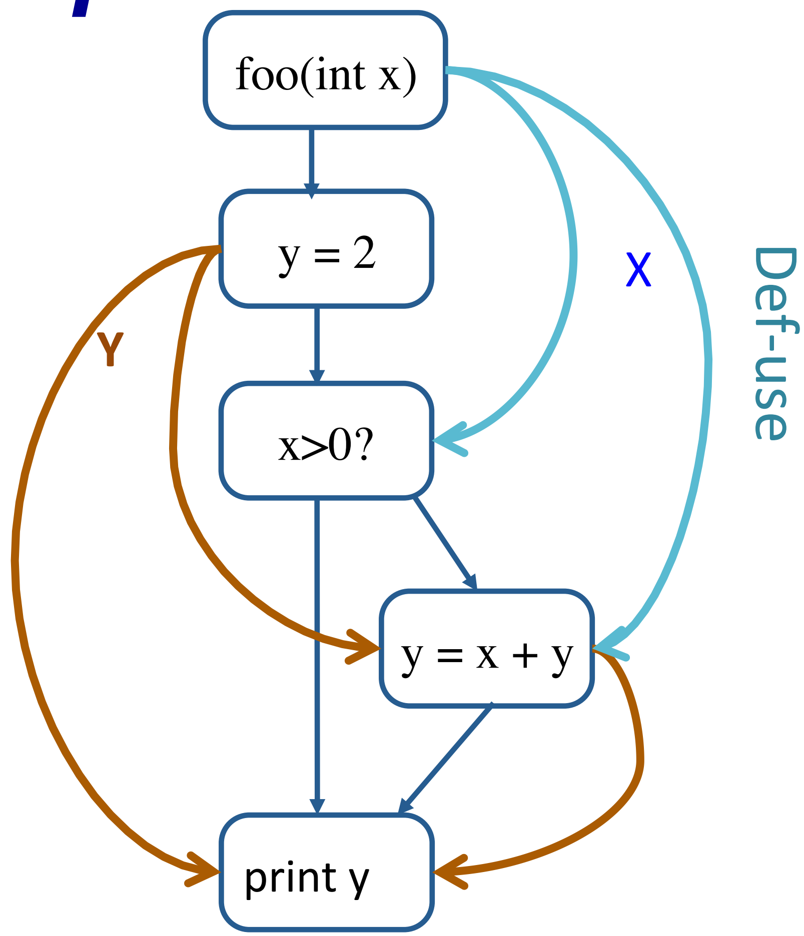
Data Dependence Graph

- Nodes: program statements

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

Example

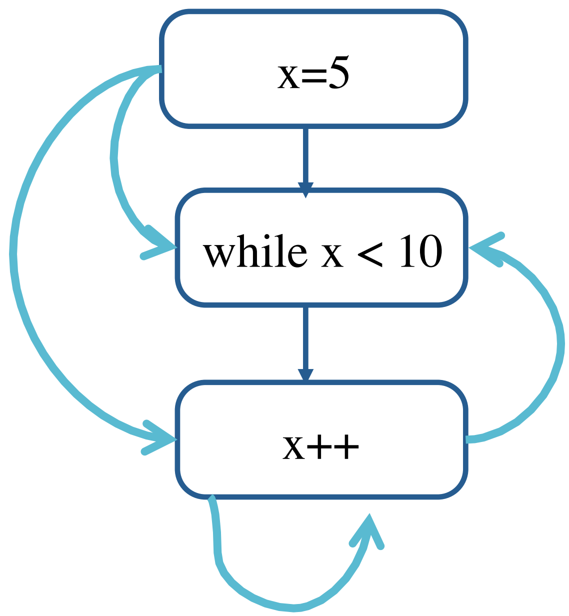
```
foo(int x) {
  y = 2;
  if(x > 0)
      y = x + y;
  endif;
  print y;
}
```



What can be printed in the last statement?

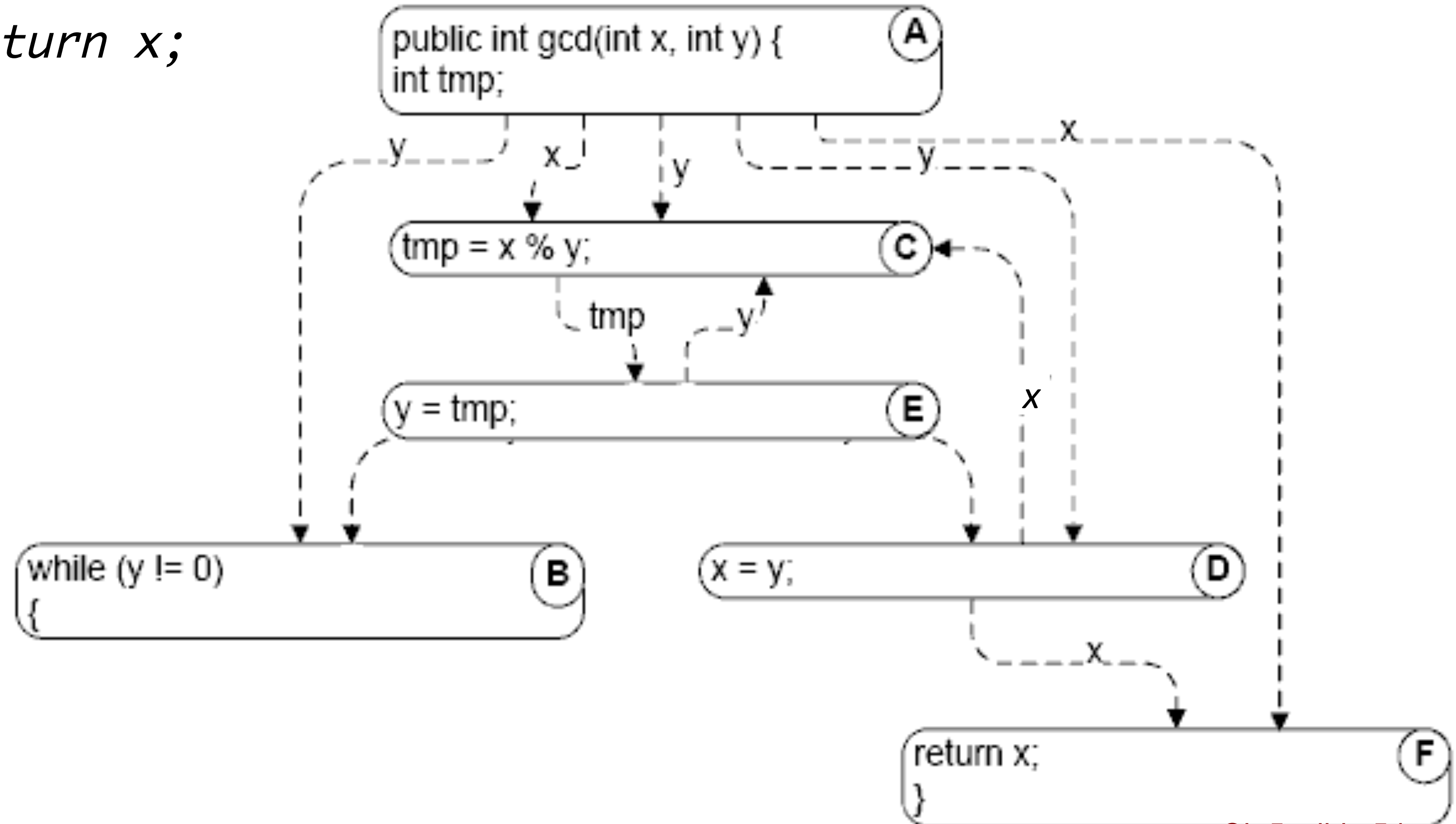
What about loops?

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



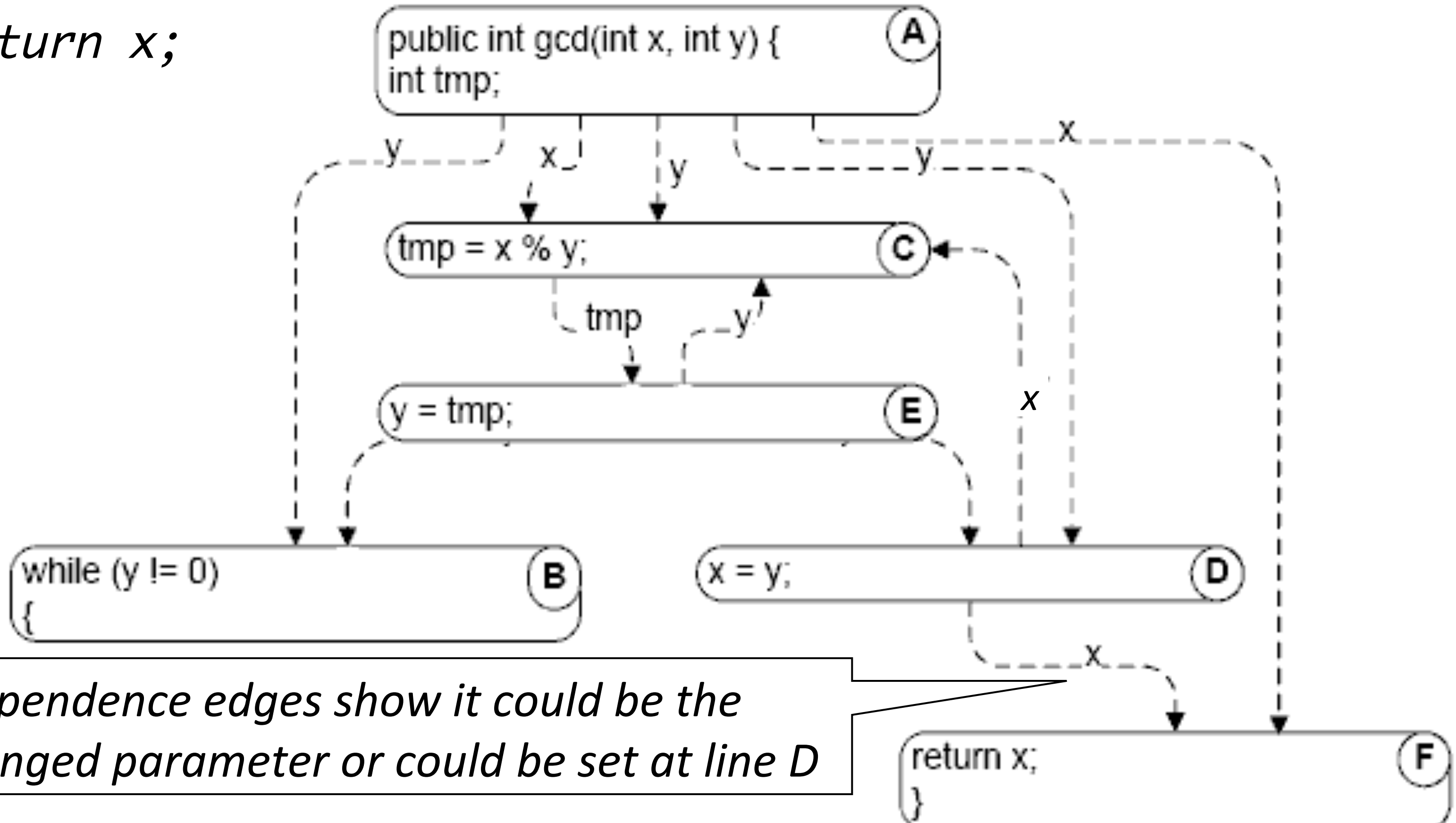
```
A: public int gcd(int x, int y) {
   int tmp;
B: while (y != 0) {
C:   tmp = x % y;
D:     x = y;
E:     y = tmp;

  }
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

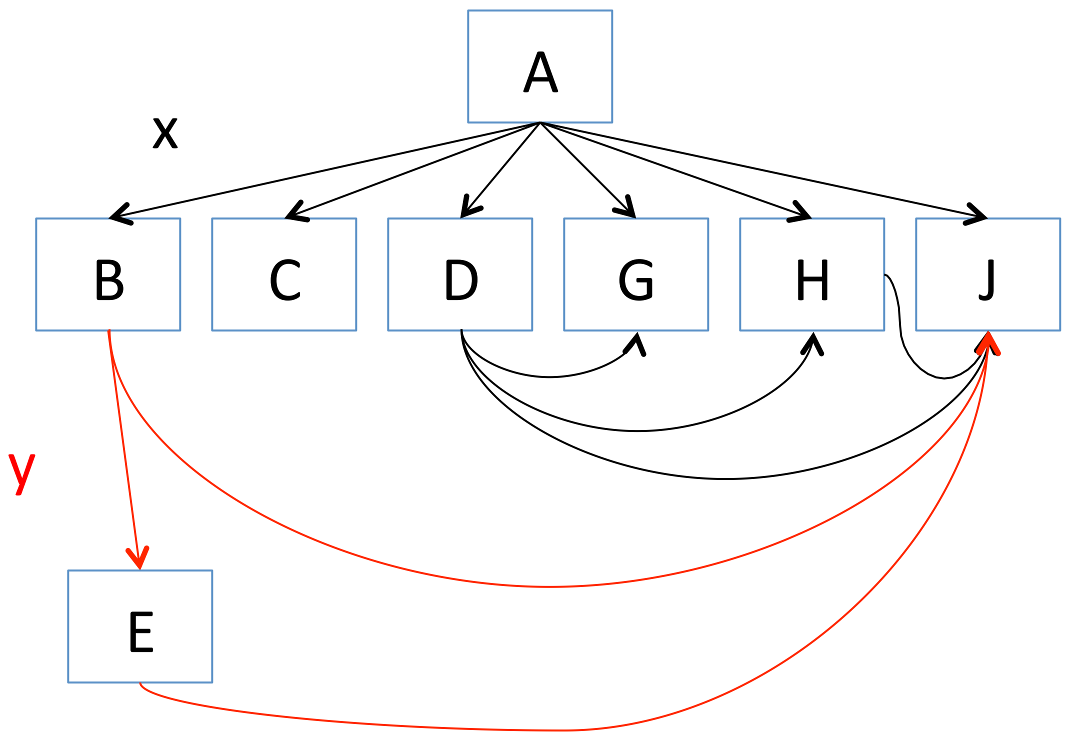
Data Flow Analysis – How Used

- 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)
- …

Exercise

- Draw data flow graph for this method

```
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:  }
```

Agenda

- Basics of Analysis
  - Control flow
  - Call graph
  - Data Flow Analysis
  - **Symbolic Execution**

- Presentation Tips

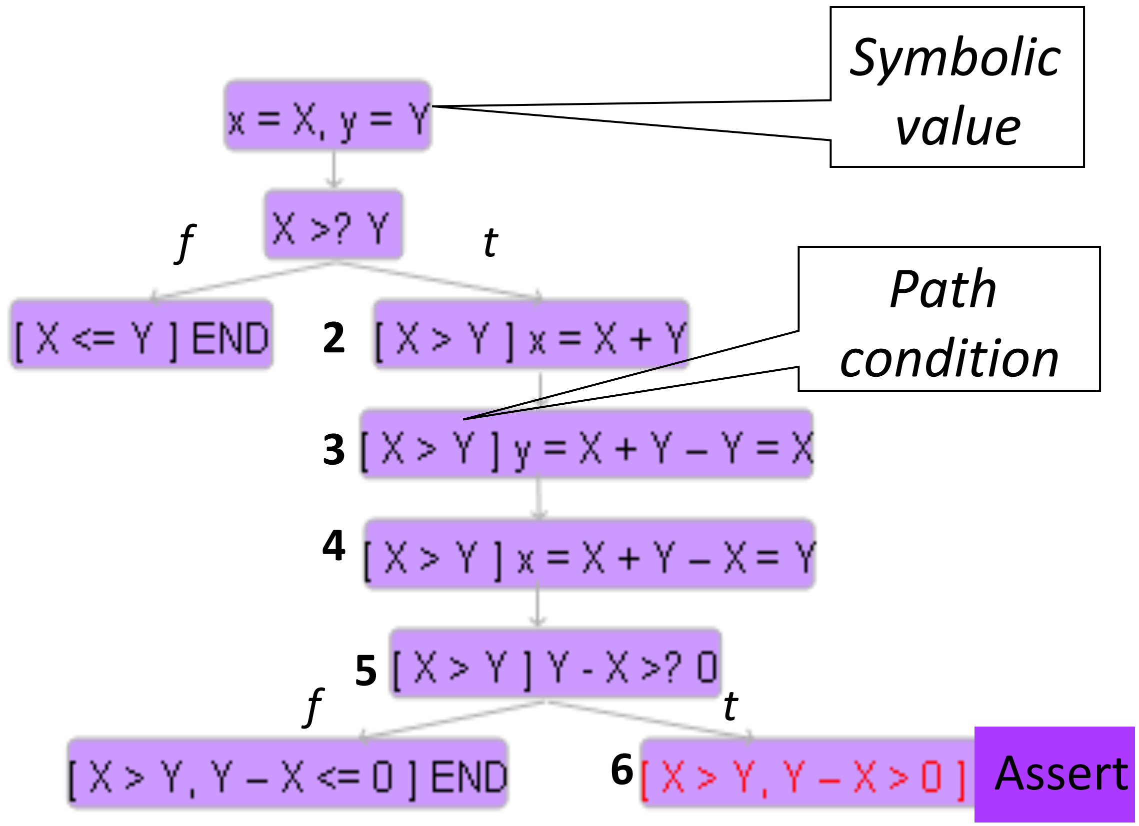
Motivating Question

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

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);
   }
```

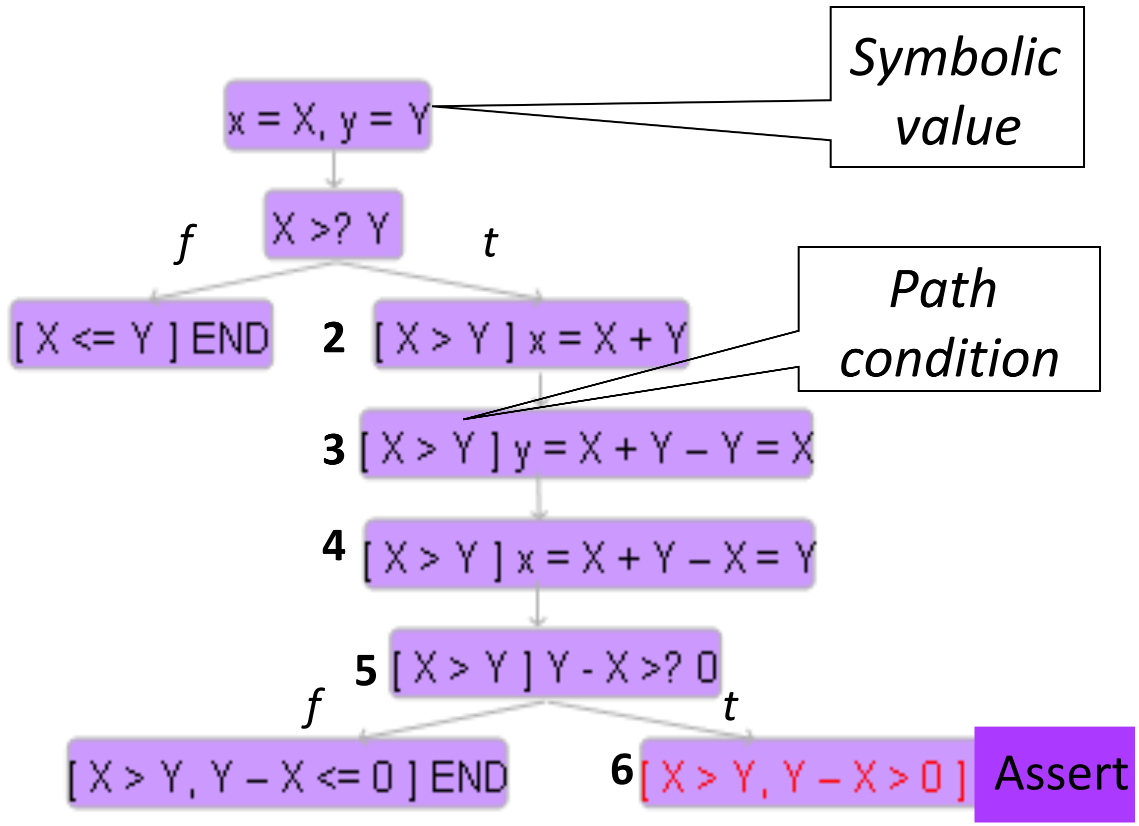


Is the assert (line 6) reachable?

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);
   }
```

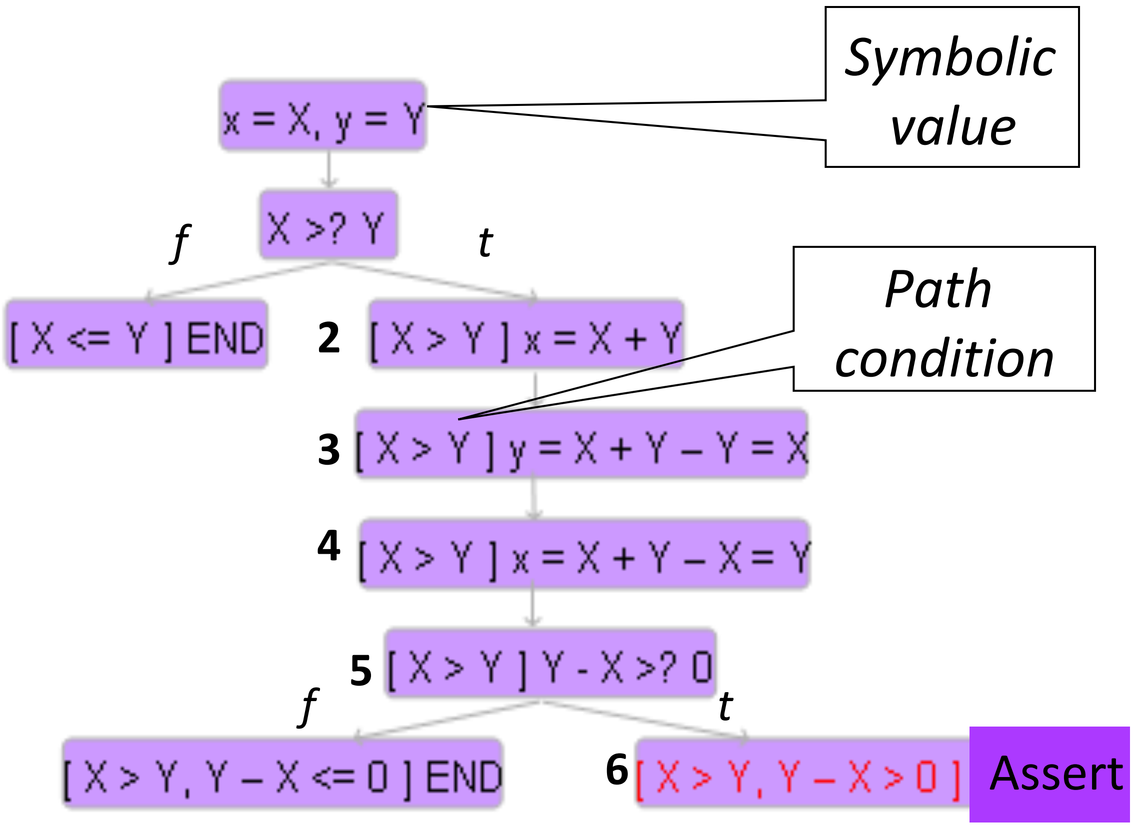


Is the assert (line 6) reachable?

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);
   }
```

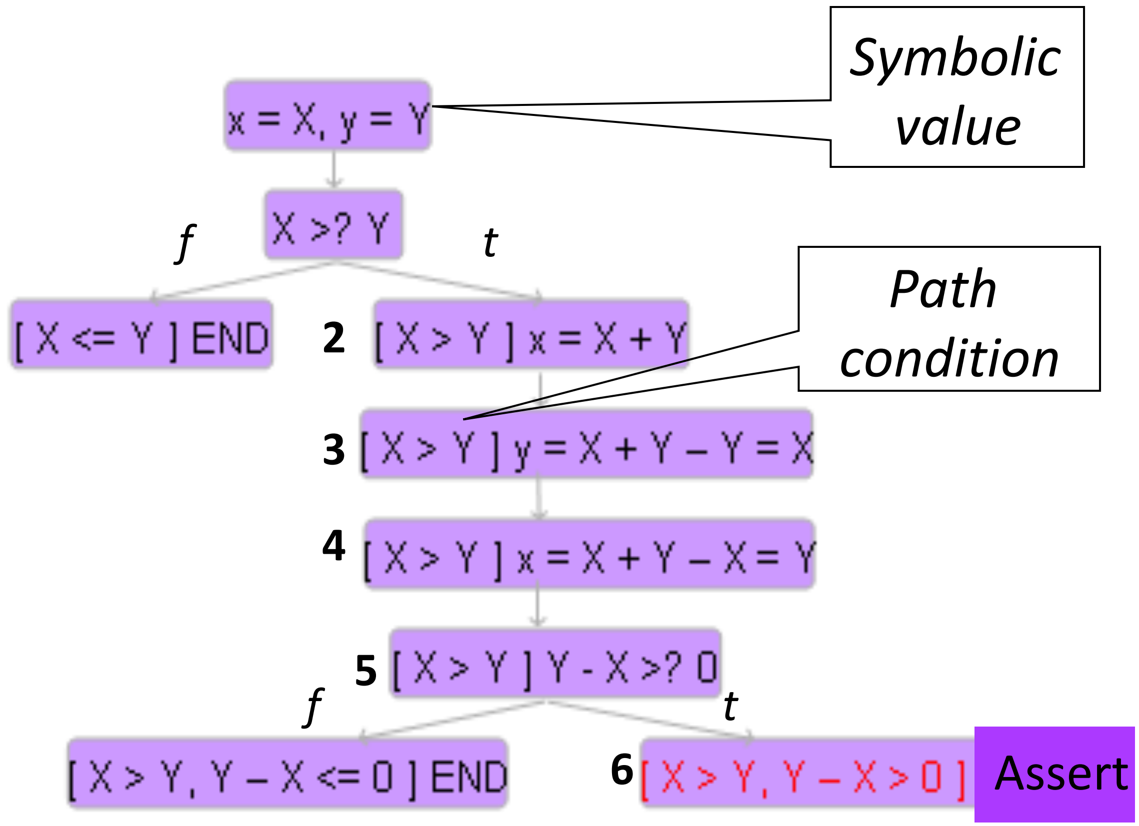


Is the assert (line 6) reachable?

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);
   }
```

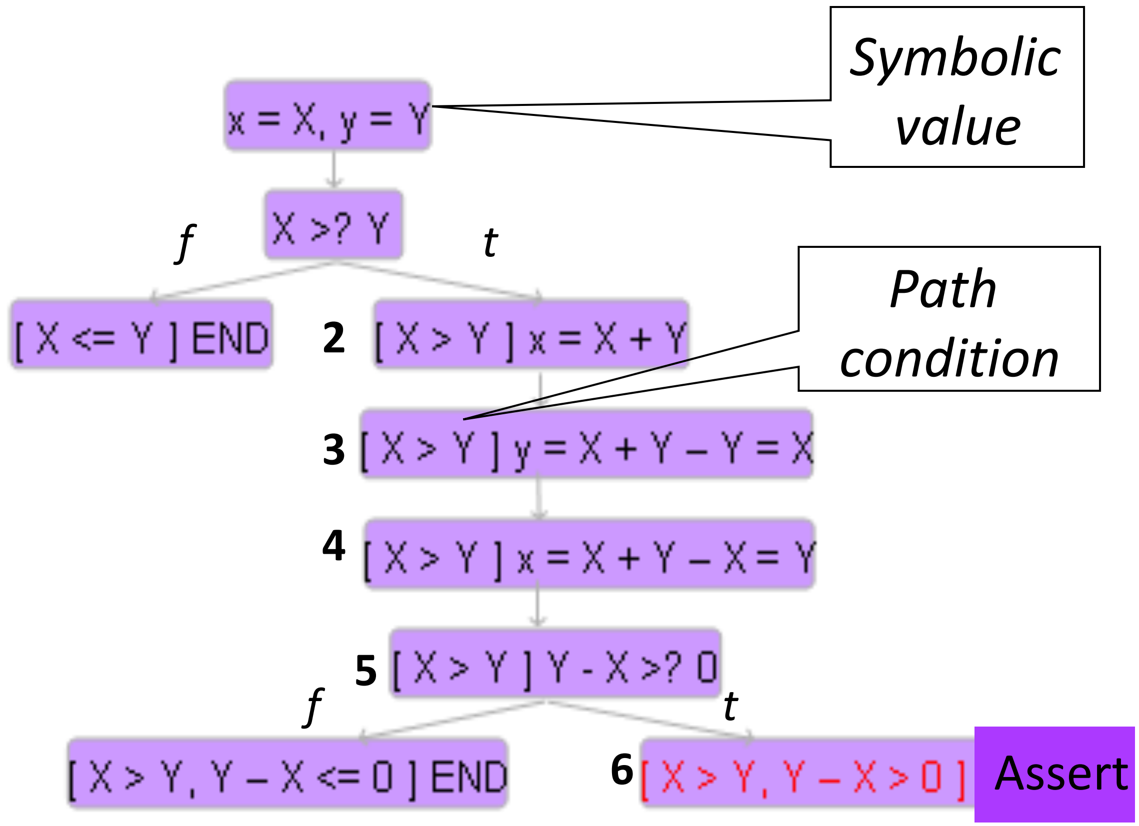


Is the assert (line 6) reachable?

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);
   }
```



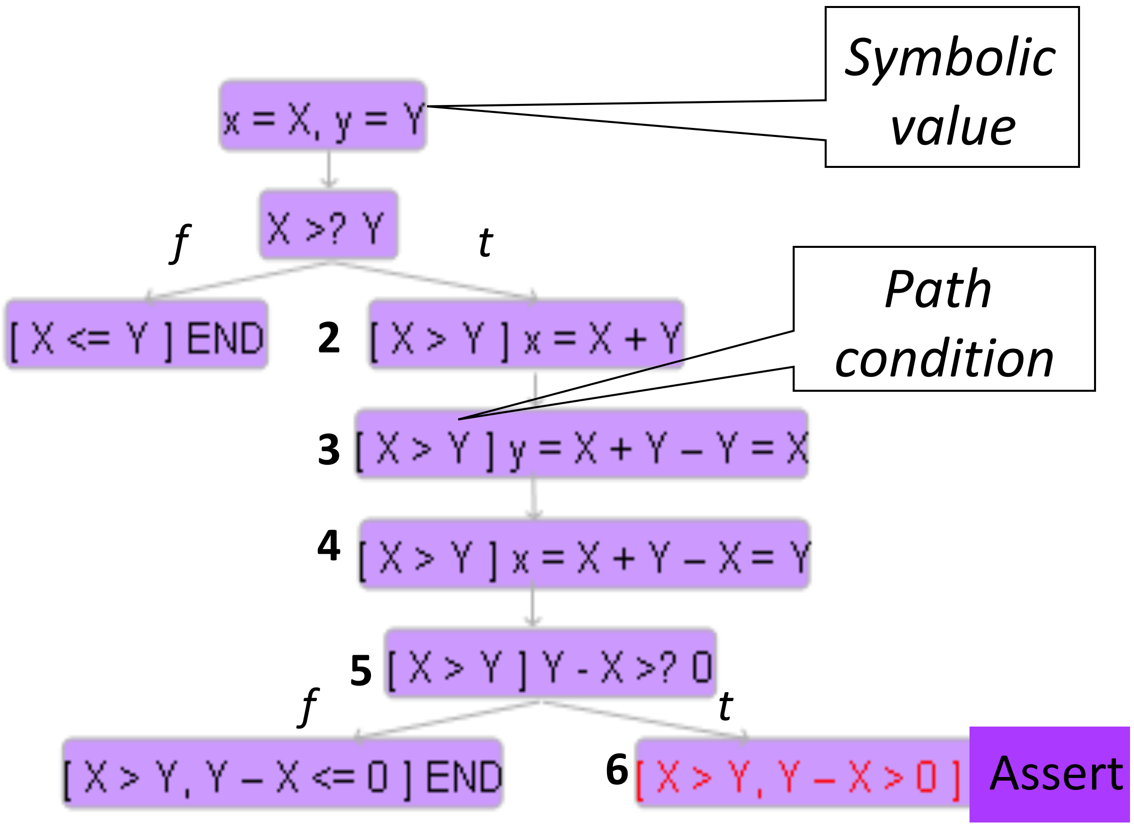
Is the assert (line 6) reachable?

Condition for 6:

X>Y & Y-X > 0

Symbolic Execution by Example

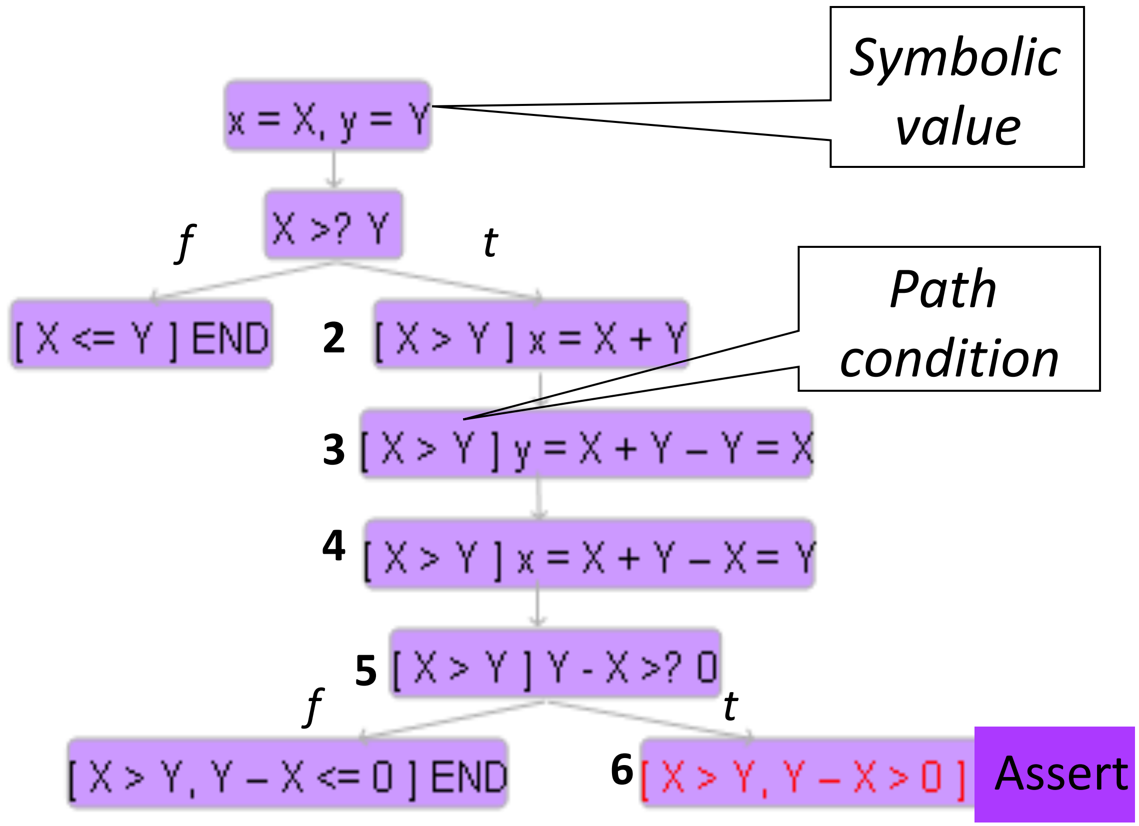
```
1: if(x>y) {
2:     x = x + y;
3:     y = x - y;
4:     x = x - y;
5:     if (x - y > 0)
6:         assert(false);
   }
```



Is the assert (line 6) reachable? NO!

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);
   }
```



Two equivalence classes (a) X<=Y (b) X>Y

Symbolic Execution

- Static Analysis
- Tracking **symbolic** rather than **actual** values
- Builds **predicates** that characterize
  - Conditions for executing paths
  - Effects of the execution on **program state**
- Is used to reason about all the **inputs** that take the same path through a program

Symbolic Path Constraints

- Theorem prover (**constraint solver**) determines if an answer exists and the branch can be taken
  - Popular constraint solvers: Z3, CVC, lp solver
  - Undecidable problem in theory

- Each path in the tree represents an equivalence class of inputs

- When paths terminate, symbolic execution computes concrete values for each path by **solving** the path constraints
  - These values can be thought of as concrete path representatives
  - E.g., test cases that can help developers reproduce bugs

Applications of Symbolic Execution

- Guiding the test input generation to cover all branches
- Identifying infeasible program paths
- Security testing
- Clone detection (equivalence checking)
- …

Limitations of Symbolic Execution

- Expensive
  - Executing all feasible program paths is exponential in the number of branches
  - Does not scale to large programs
- Problems with function calls
- Problem with handling loops
  - often *unroll* them up to a certain depth rather than dealing with termination or loop invariants
- Expensive to reason about *expressions*
  - Although modern SMT solvers help!

Difference Between Symbolic Execution and Data Flow Analysis?

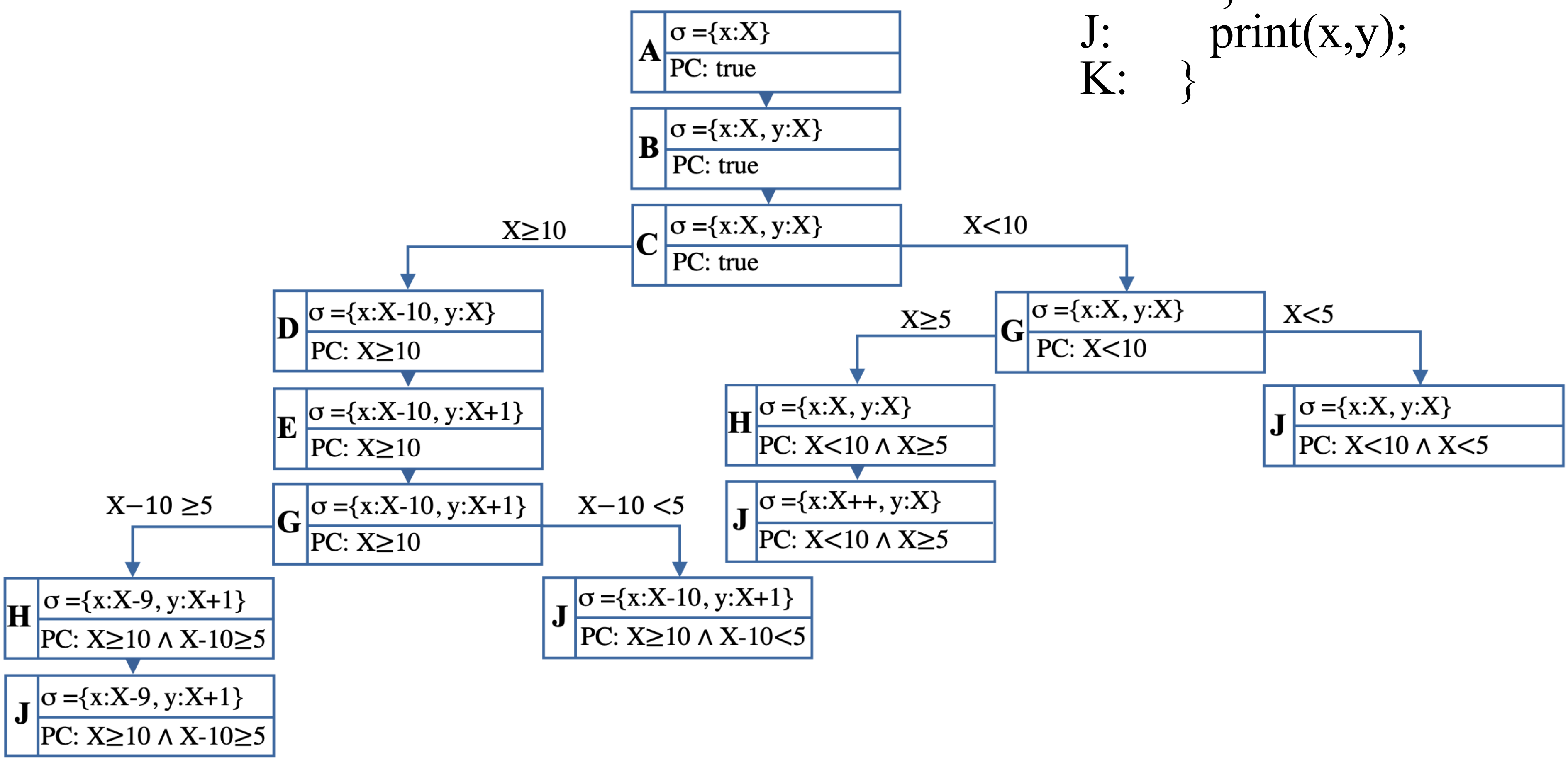
- Different purpose
  - Symbolic execution: reason about path feasibility
  - Data flow analysis: check which definitions are live, which values are constant, etc.
- Different abstraction
  - Symbolic execution: symbolic computational paths
  - Data flow analysis: concrete def-use
- Accuracy
  - Symbolic execution: only feasible paths considered
  - Data flow analysis: all paths considered
- …

Exercise

- Execute symbolically

```
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:   }
```



Summary: "Holy-Grail" of Analysis

- ✓ Useful
- ✓ Accurate
- ✓ Scalable

Questions:

- What is the right model for the task?
- What is the right analysis for the task?

Summary: Static vs. Dynamic Analysis

Dynamic Analysis

- Draw inferences from a sample
- Scale well
- Precise for the analyzed samples
- Miss info
- Can require expensive instrumentation

Static Analysis

- Try to be conservative, i.e., never declare a property to be valid if it is not
- Over-estimate actual behavior
- Often sacrifice precision for scalability

In reality – both have limitations.

Choose the right tool for the task!