You are a lucky bug. I'm seeing that you'll be shipped with the next five releases.

CPEN 321

Validation & Verification, Basics of Analysis

Agenda

- **Logistics**
- Validation and Verification
- Basics of Analysis
- Symbolic Execution ← Next week

Following Milestones

- W4: Development team and the customer discuss the requirements.
- W5: M1 – Requirements (both customer and development teams).
- W6: M2 – Design (development team).
- W8: M3 – MVP (development team).
- W9: M4 – Code review (development teams).
- W10: M5 – Test plan and results (development team).
- W11: M6 – Refined specifications (both customer and development teams).
- W12: M7 – Customer acceptance test (customer team).

Expectations for MVP

1. Have both client and server up, running, and communicating
2. Have at least one **major** use case fully implemented
   - a major use case: includes some "risky" steps
   - e.g., integration with third-party services, complex algorithm, etc.

<u>Submission</u>:

- (The usual) one-page status report
- No formal deliverable should be attached to the report

<u>In the weekly meeting</u>:

- Bring a mobile device running the front-end app
- Be ready to connect (ssh) to the backend

Ask Your Questions!

Office Hours:

- Julia: Mon. 4:30-5:30pm, KAIS 4053 (or by appointment)
- Michael / Sahar: Mon. 12-1pm, KAIS 4095
- Harsha / Zeyad: Wed. 2-3pm, KAIS 4095

Quest Lecture

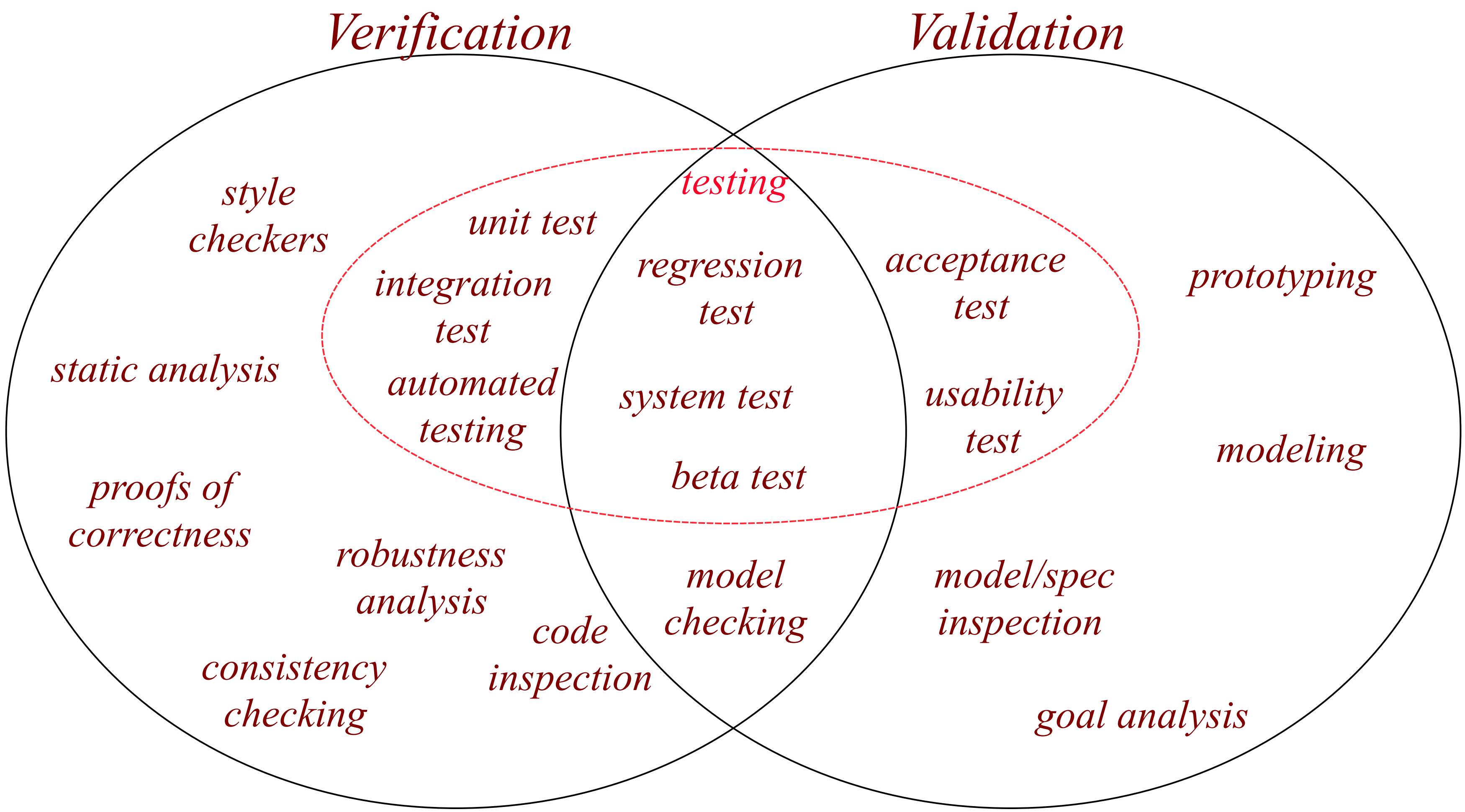
- Monday, October 29
- **Anthony Chu**, Microsoft

- "Containers for development and production"
  – How Docker containers can be used for local development, testing, and deployment to a production environment

Agenda

- Logistics
- **Validation and Verification**
- Basics of Analysis
- Symbolic Execution ← Next week

Validation and Verification



Validation and Verification

Validation:

Are we building the right system?

Does it address the customer needs?

Verification:

Are we building the system right?

Does the implementation meets the spec?

Program Analysis: Reasoning About Code

- The process of automatically analyzing the behavior of programs
- Examples?

Types of Analysis

- *Static* – without executing the program
  - Counting lines of code for a method
  - Checking that each method has a comment
  - Checking that each called method is defined
  - …
  - Tracking data flows
  - Symbolic execution
  - Model checking
  - …

- *Dynamic* – at runtime
  - Testing
  - Profiling
  - Monitoring
  - Debugging
  - Program slicing
  - Tracking data flows
  - …

Major Application Areas

- Program optimization:
  - improving the program's performance while reducing its resource usage
- Program correctness:
  - validation of a correctness, robustness, security, style checkers, code inspection

Why Program Analysis?

- Development costs
  - generally measured in hundreds to thousands of dollars per delivered LOC
  - testing and analysis is usually 50% of this cost
- Maintenance costs
  - 2-3 times as much as development

Program understanding, validation, repair, etc.
rely on program analysis

Black-Box vs. White-Box Analysis

- *Black-box* – requires no knowledge of internal paths, code structures, or implementation of the software
  - Testing
  - Monitoring
  - …
- *White-box* – based on internal paths, code structures, and implementation of the software
  - Counting lines
  - Control and data flow analysis
  - Symbolic execution
  - Model checking
  - Testing
  - …

Black-Box vs. White-Box Analysis

- *Black-box* – requires no knowledge of internal paths, code structures, or implementation of the software
  - Testing
  - Monitoring
  - …
- ***White-box* – based on internal paths, code structures, and implementation of the software**
  - Counting lines
  - Control and data flow analysis
  - Symbolic execution
  - Model checking
  - Testing
  - …

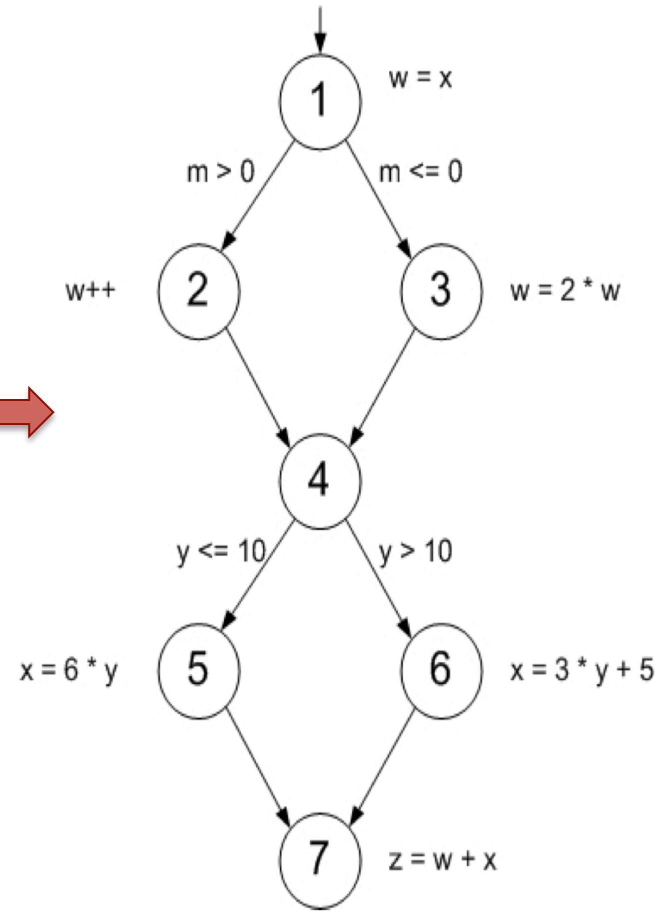
Agenda

- Logistics
- Validation and Verification
- **Basics of Analysis**
  - Control flow
  - Call graph
  - Data Flow Analysis
- Symbolic Execution ← Next week

Models

Models

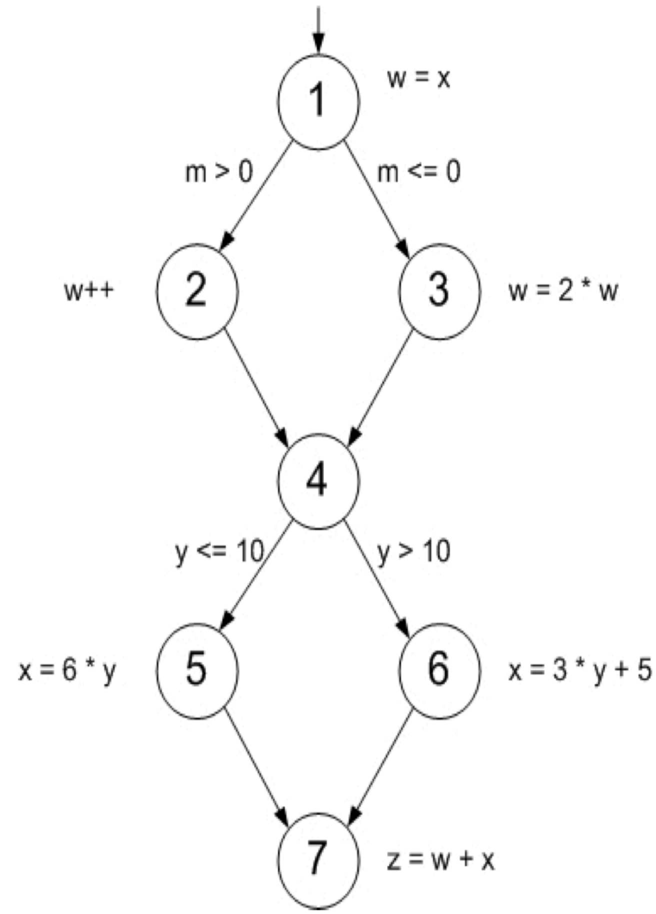
Models



Modeling Software

Graphs! E.g.,

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

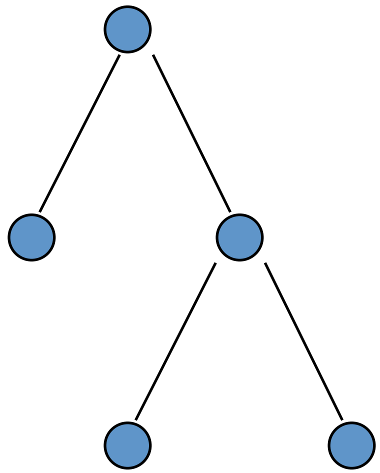


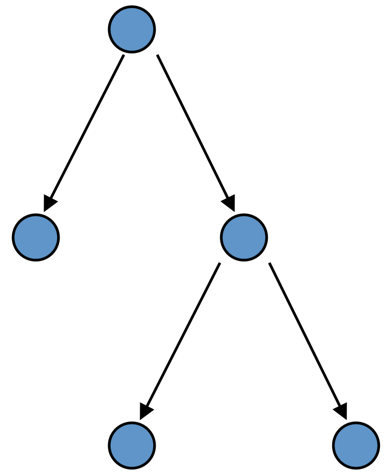
Graphs

- 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*

Graphs and Trees

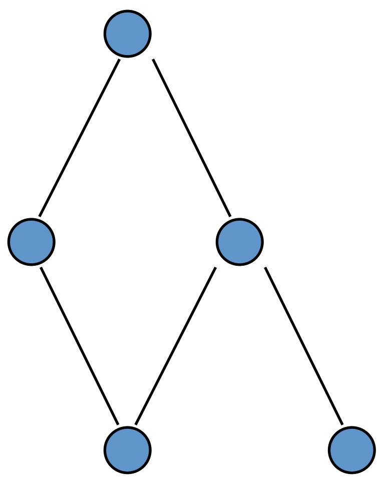
tree

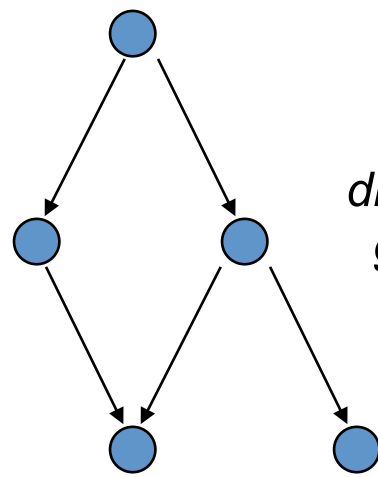




directed tree

cyclic undirected graph





directed acyclic graph (DAG)

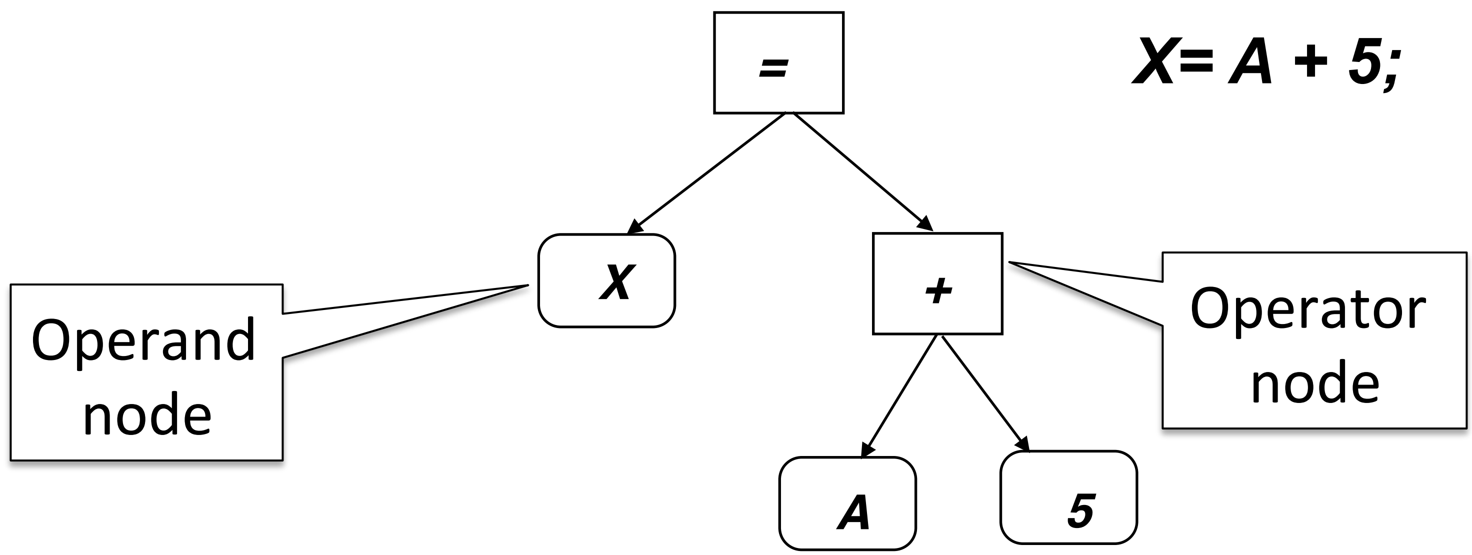
What Kind of Graphs are Used?

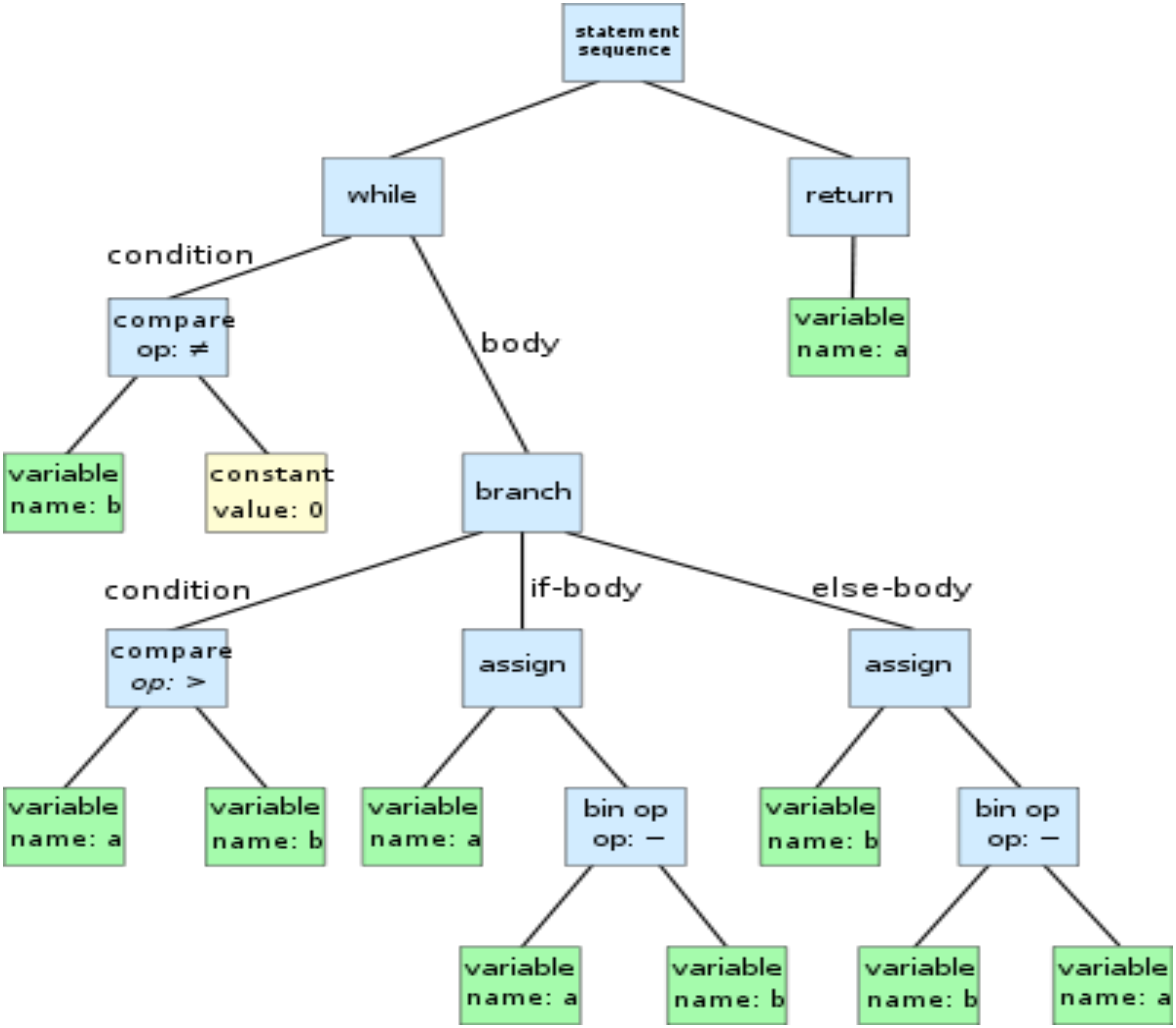
- **Sufficiently general**: general enough for practical use in the intended domain of application
- **Compact**: representable in a reasonably compact form
  - What is *reasonably compact* depends largely on how the model will be used
- **Predictive**: represent the modeled artifact well enough to distinguish between *good* and *bad* outcomes of analysis

No single model represents all characteristics well enough to be useful for all kinds of analysis

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




statement sequence
while
return
condition
compare
op: ≠
body
variable
name: a
variable
name: b
constant
value: 0
branch
condition
if-body
else-body
compare
op: >
assign
assign
variable
name: a
variable
name: b
variable
name: a
bin op
op: −
variable
name: b
bin op
op: −
variable
name: a
variable
name: b
variable
name: b
variable
name: a

Control Flow Graph (CFG) – Example

total, value, count, maximum : int;

total := 0;

count := 1;

read maximum;

while (count <= maximum) do

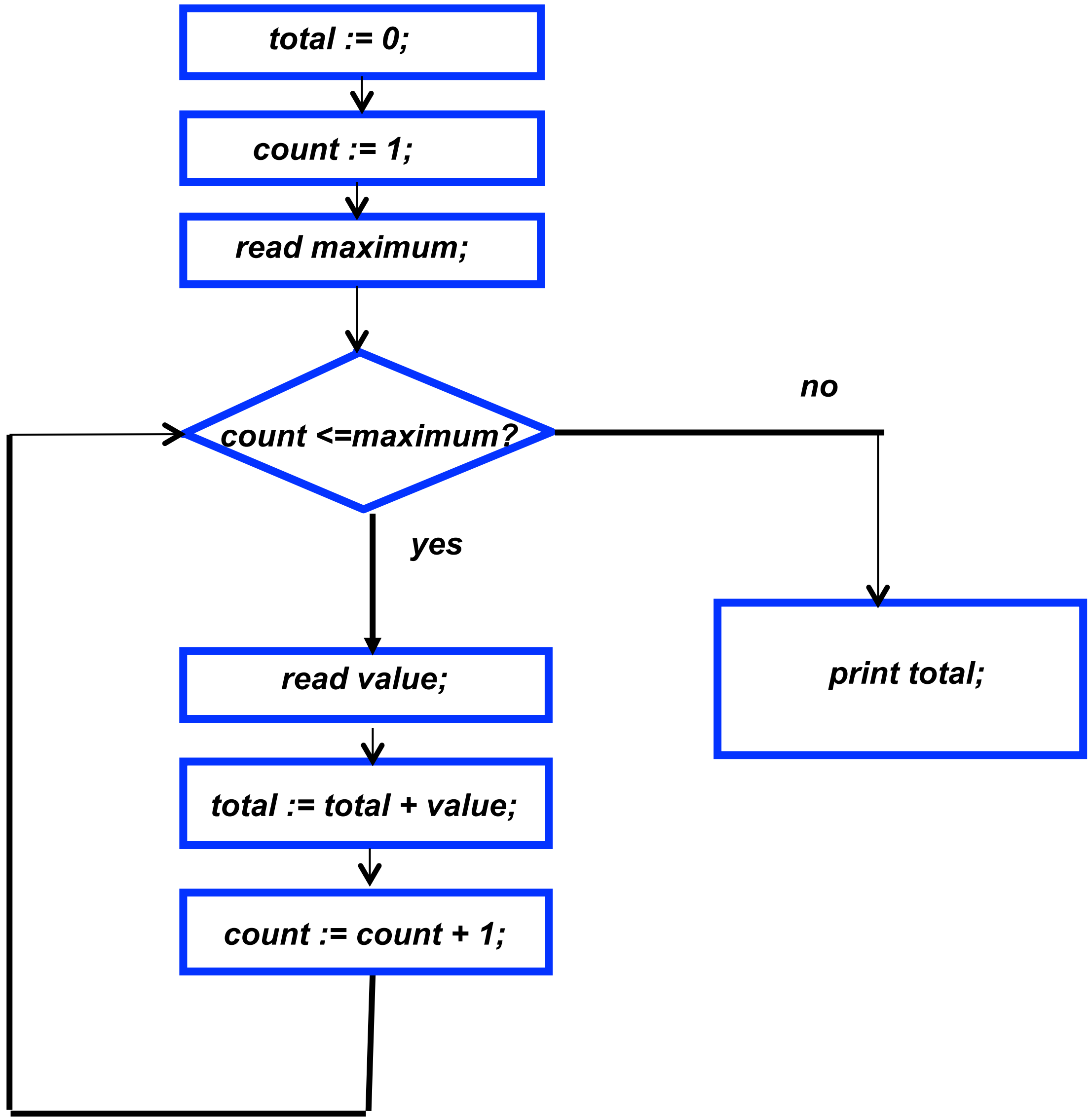
read value;

total := total + value;

count: = count + 1;

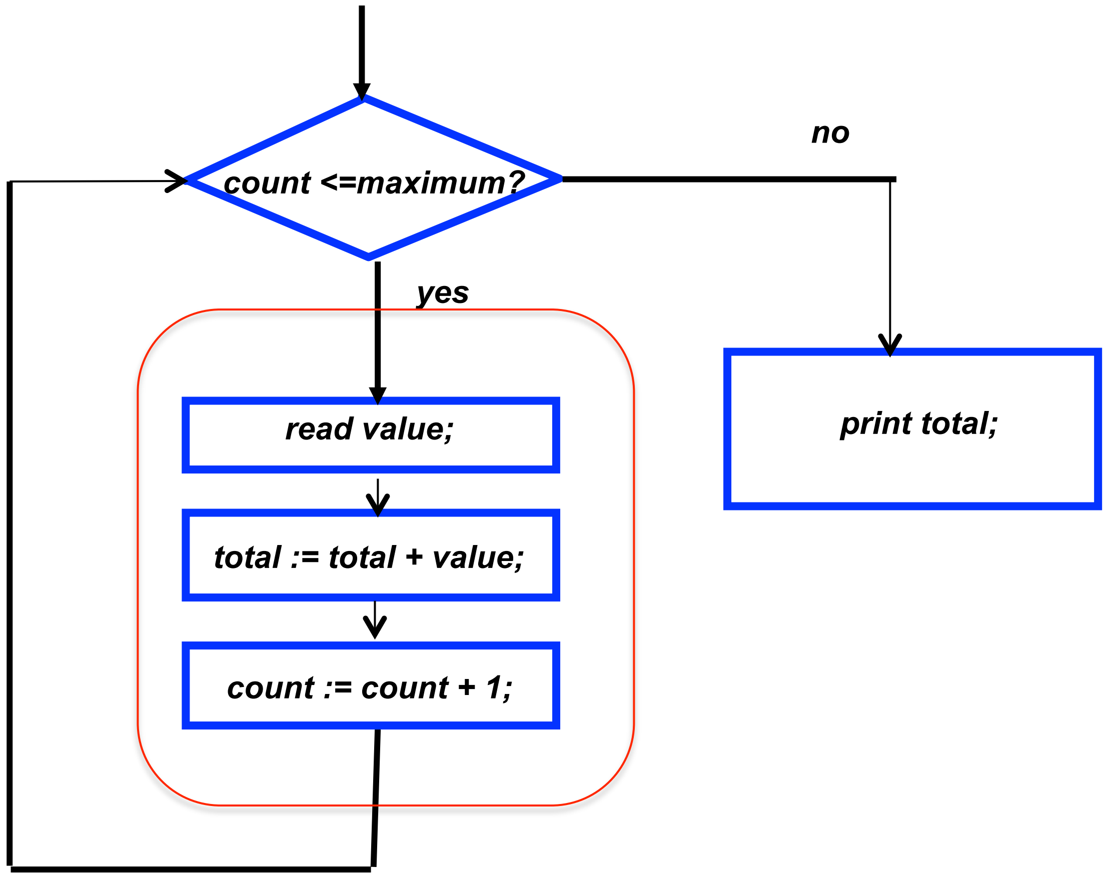
endwhile;

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

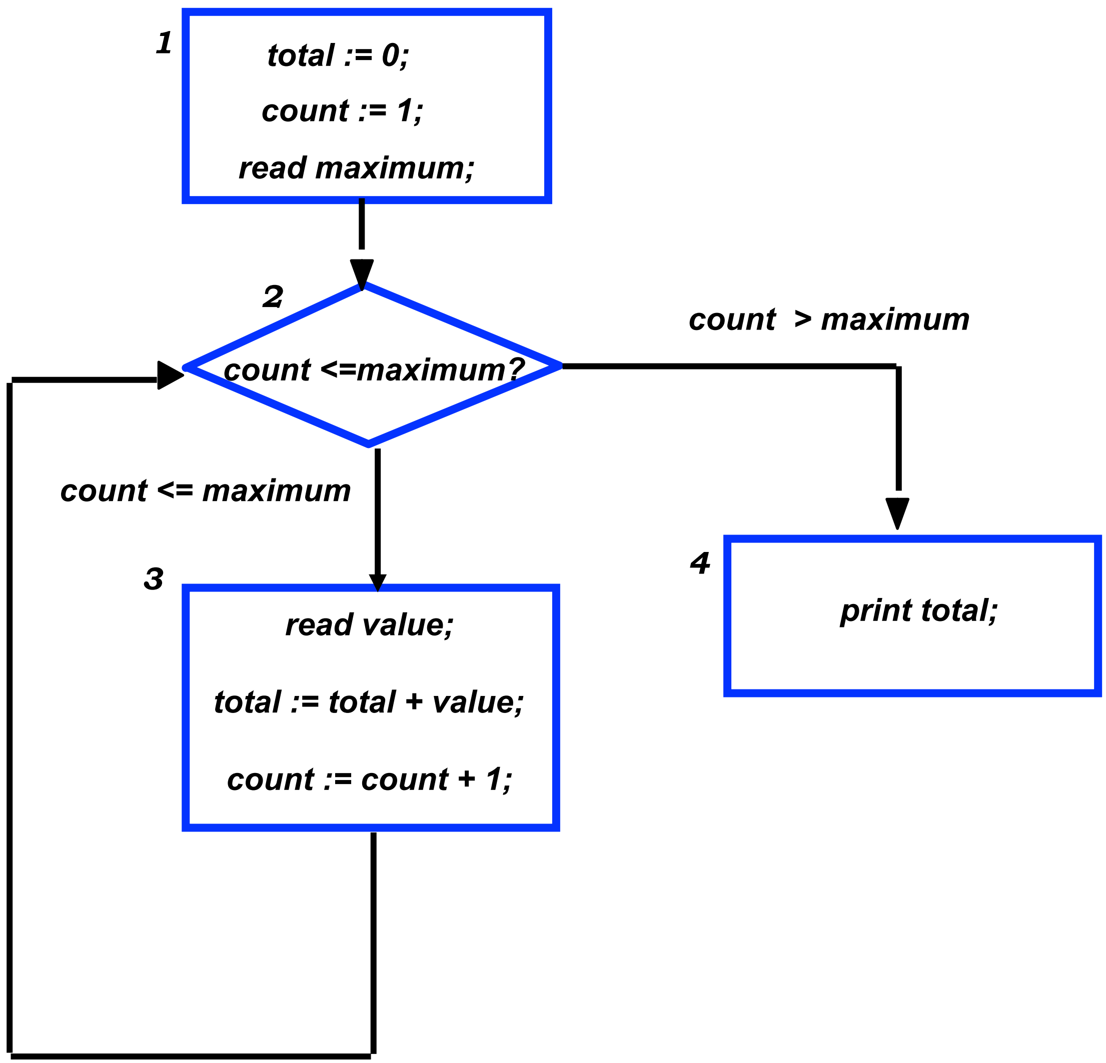
read value;

total := total + value;

count: = count + 1;

endwhile;

print total;

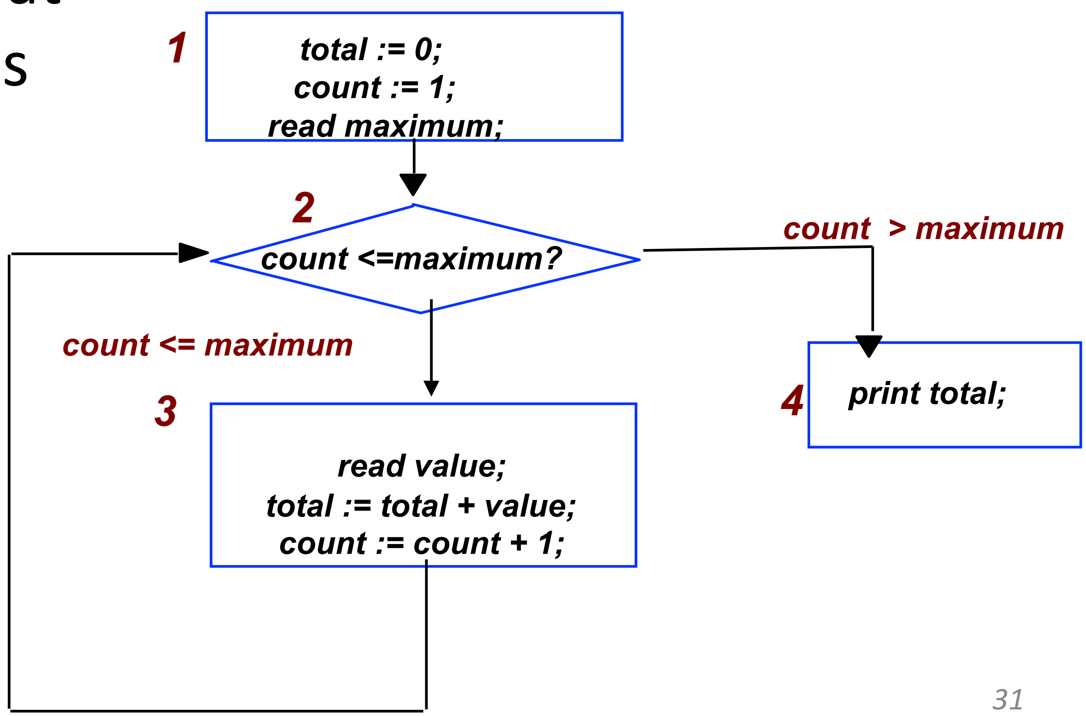


Control Flow Graph (CFG) – Definition

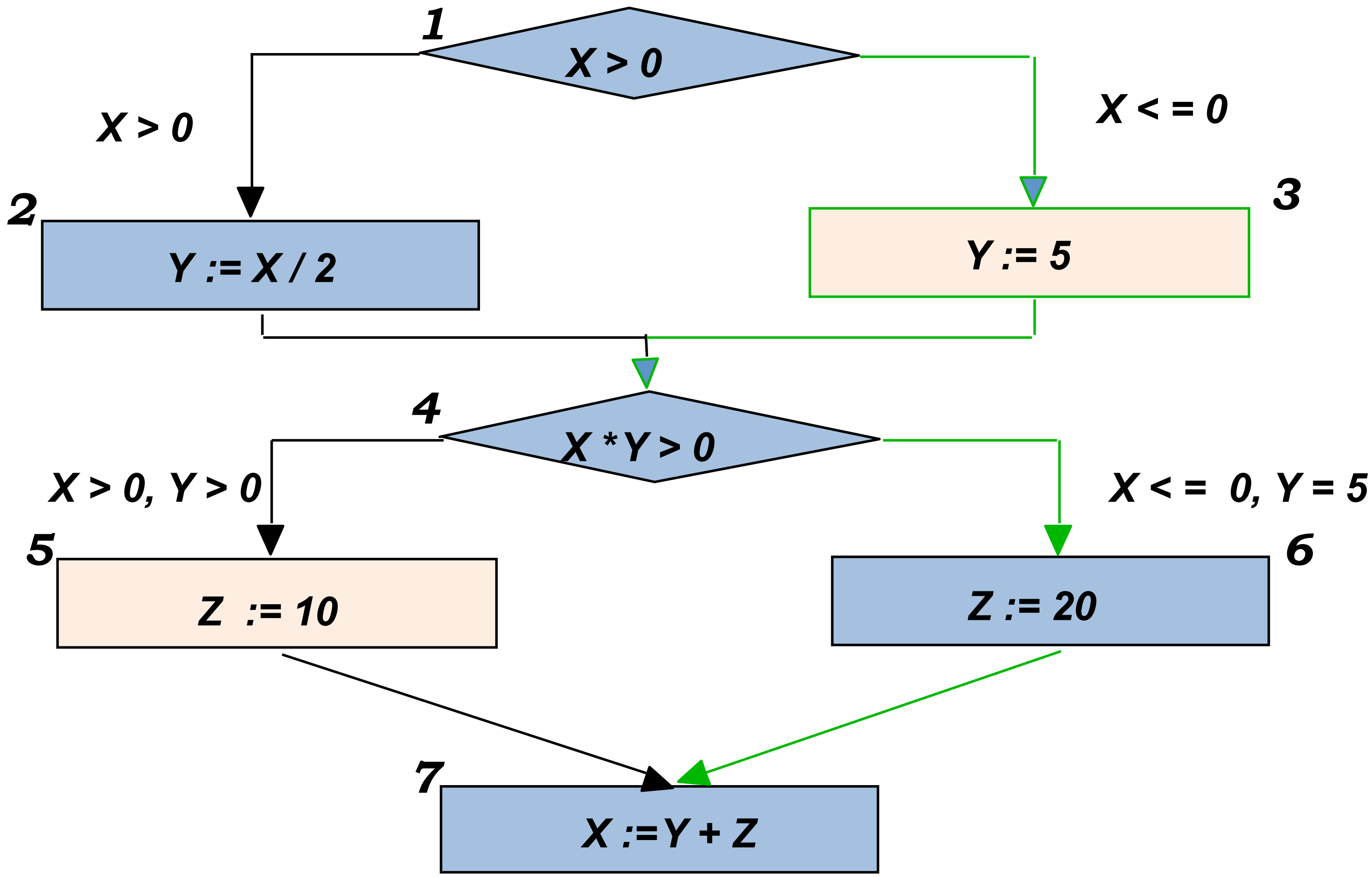
- Nodes N: statements or (more often) basic blocks
- Directed edges E: *potential* transfer of control from the end of one region directly to the beginning of another
  - E = { (n_i, n_j) | syntactically, the execution of n_j follows the execution of n_i}
- Intraprocedural (within a method)

CFG Paths

- A subpath through a control flow graph: a sequence of nodes $n_k,\ldots,n_m$, such that for each n_i, $k \le i < m$, (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

Benefits of CFG

- Probably the most commonly used representation
  - Numerous variants
- 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

Call Graphs (Interprocedural CFG)

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

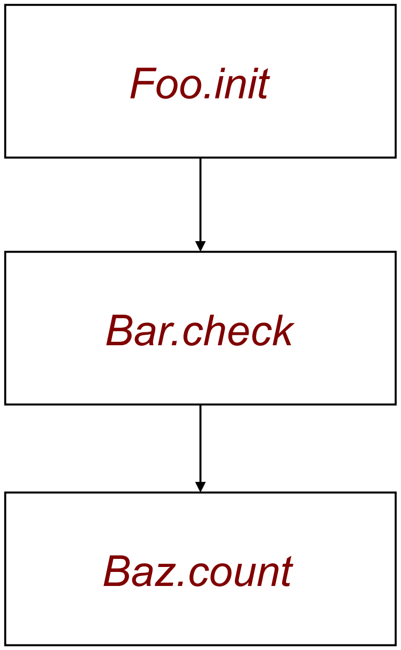
Example

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

public class Bar {

   void check() {
      count();
    }
}

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



Example

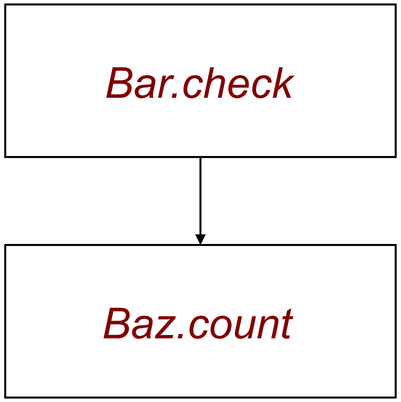
```
public class Foo {
   void init() {
     new Bar()).check();
   }
}

public class Bar {

   void check() {
      count();
    }
}

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

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



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

Call Graphs … Not That Simple

- Creating the exact (static) call graph is an **undecidable** problem
  - Computing call graphs require point-to analysis (a.k.a. pointer analysis or alias analysis)
  - Exceptions
  - …
- Multiple existing heuristic algorithms
  - Various degree of precision / scalability

```
class A {
  void f();
}
class B extends A {
  void f();
}
B b = new B();
A a = b;
a.f();
```

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

Data Flow Analysis

Intuition:

- Statements interact through **data flow**
- Value computed in one statement is used in another

Definition

- **Data flow analysis** is a technique for gathering information about the possible set of values calculated at various points in a computer program.
- Usually annotates CFG

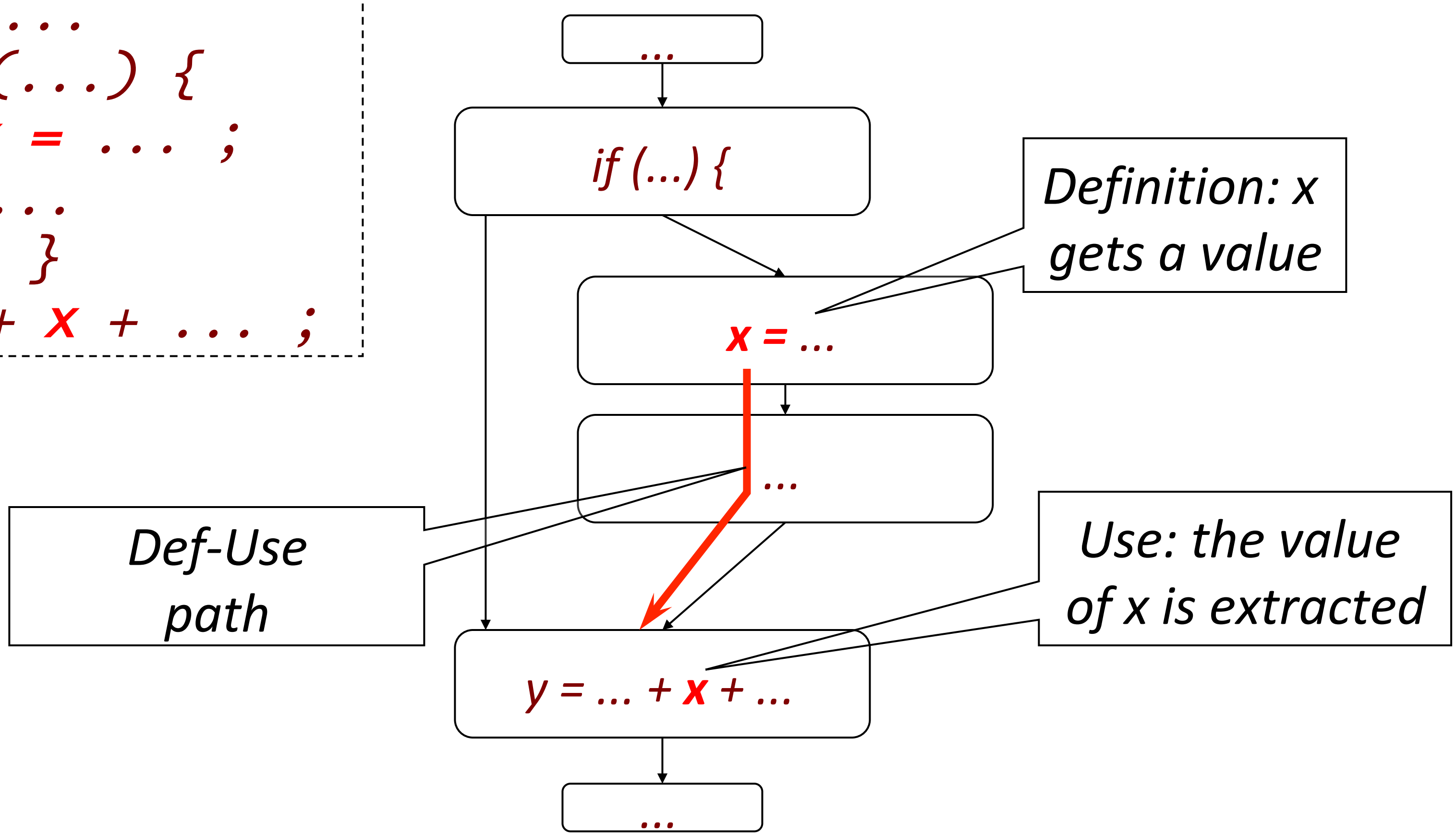
Variable Definition and Uses (DU)

- 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, e.g., x++;

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

Def-Use Path

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

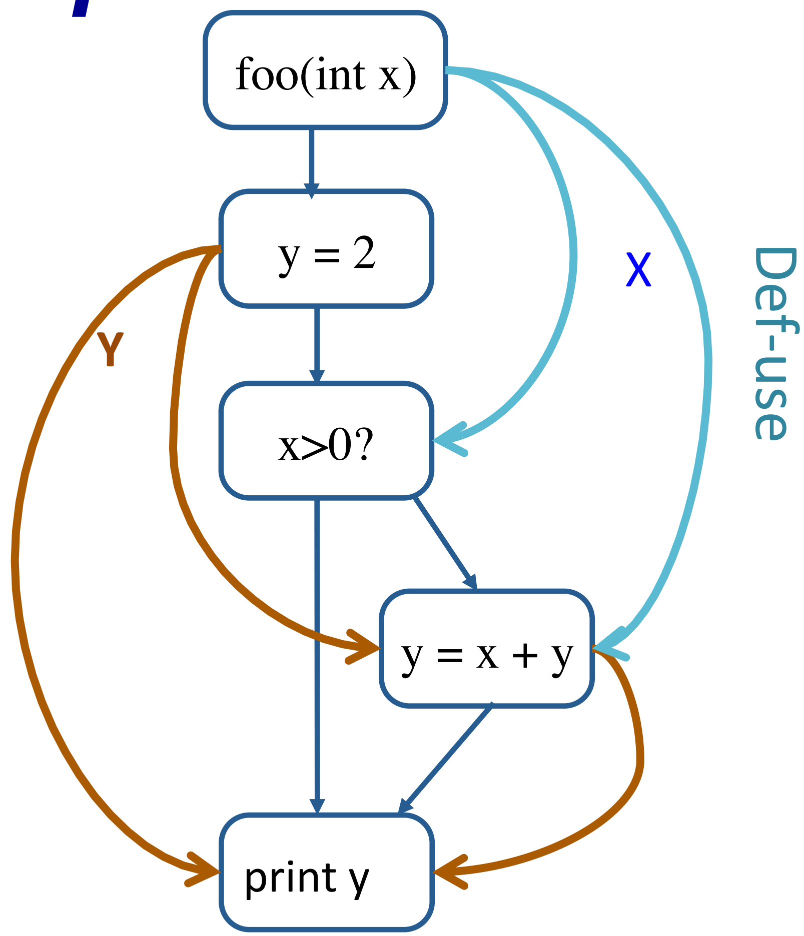


Data Dependence Graph

- Nodes: as in the control flow graph (usually statements rather than basic blocks)
- Edges: def-use (du) pairs, labeled with the variable name

Example

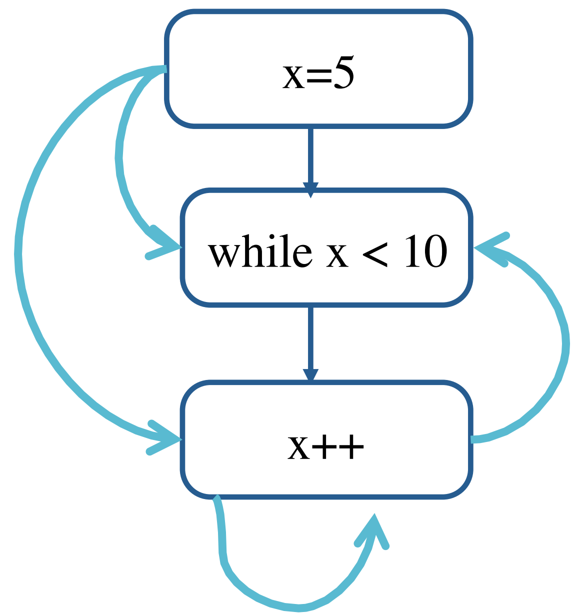
```
foo(int x) {
  y = 2;
  if(x > 0)
      y = x + y;
  endif;
  print y;
}
```



How can the last statement be reached??

What about loops?

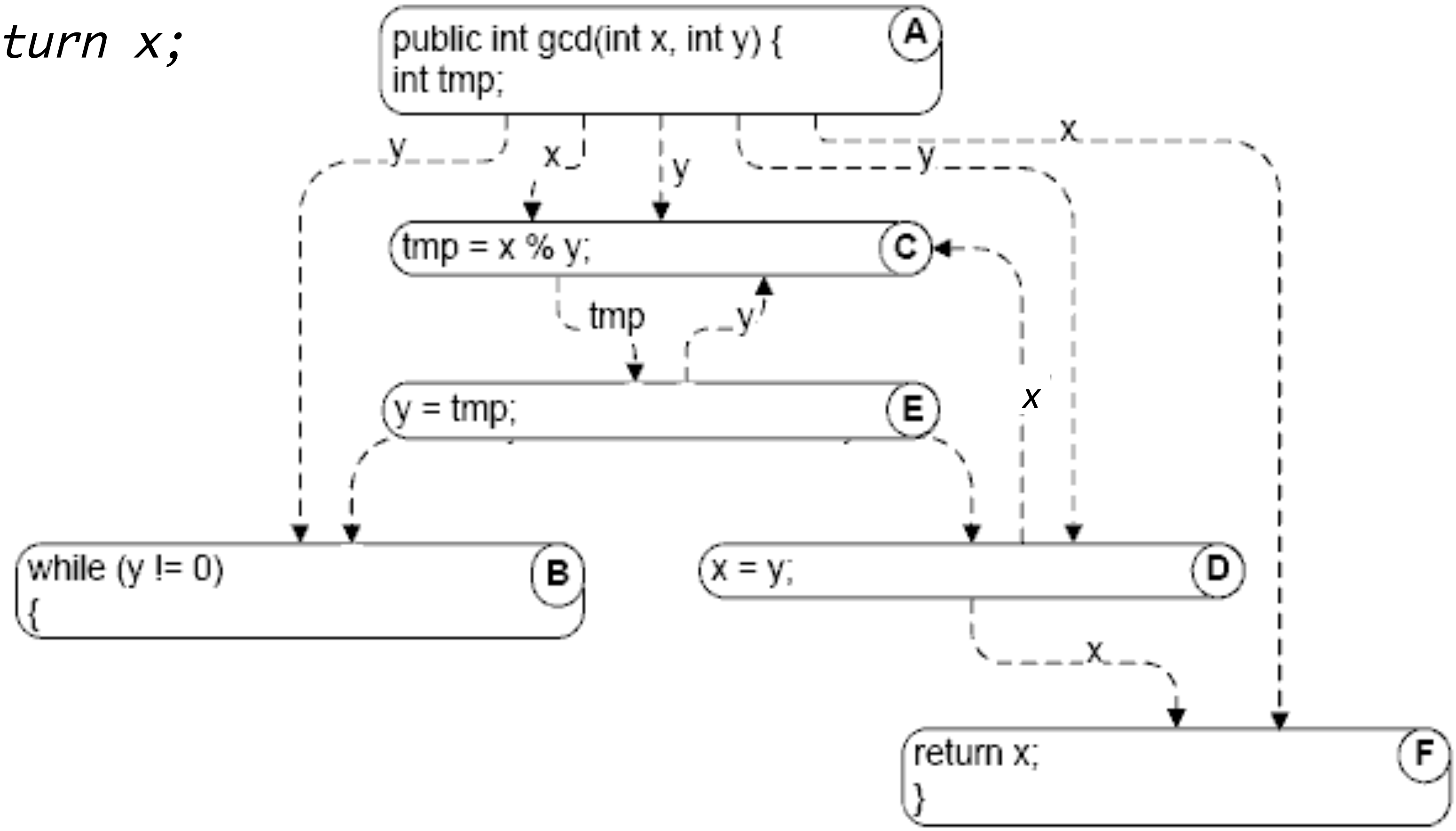
```
x=5;
while (x< 10)
{
  x++;
}
```



```
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;
}
```

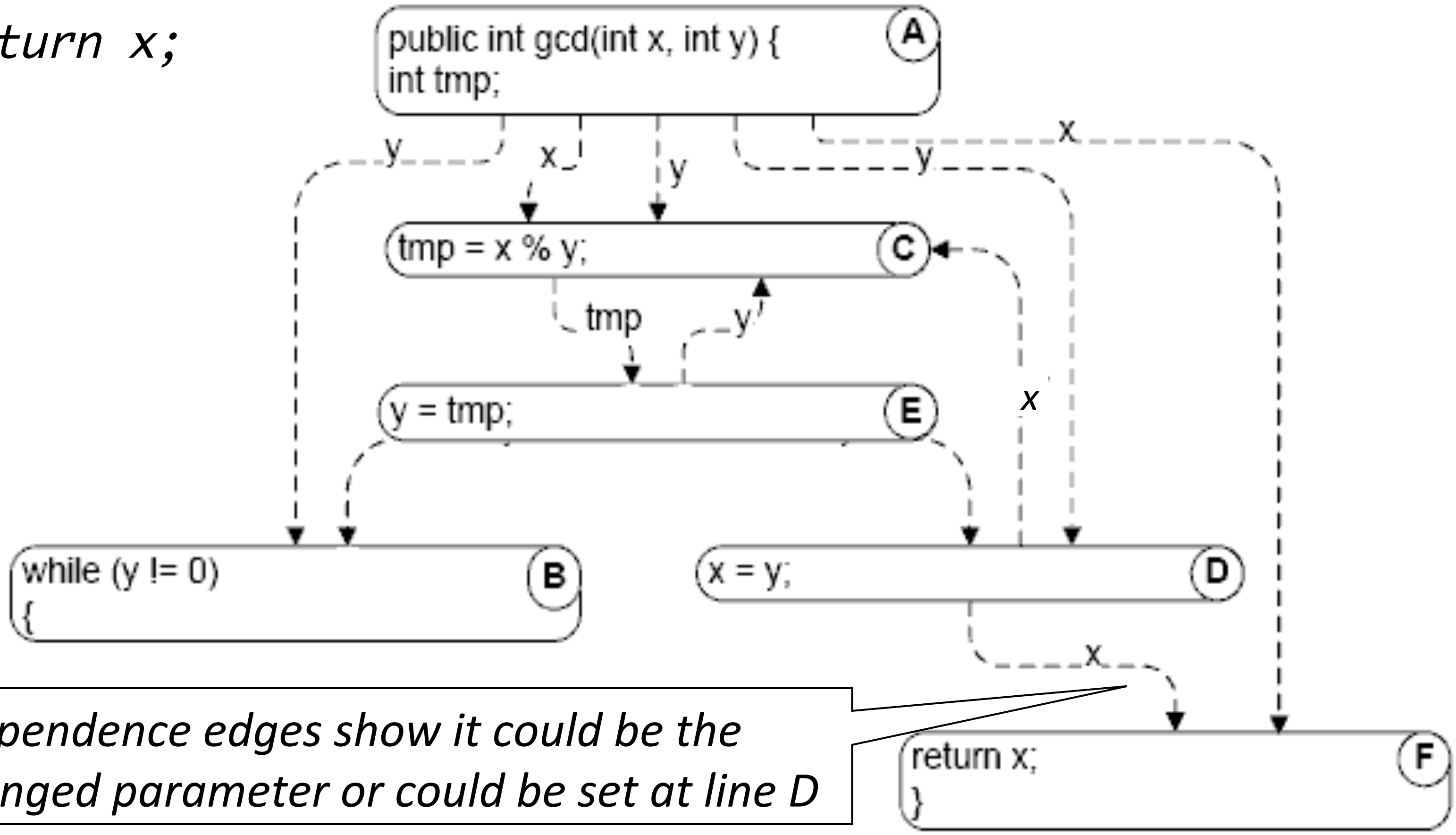
Control flow edges are omitted in this example



```
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

- Program analysis, e.g.,
  - determine if sensitive value reaches sensitive sink (taint analysis)

Exercise:
Draw a Control and Data Flow Diagrams

```
A:   void f(int x, int y, int z) {
B:     if (x>0) {
C:        x = -2;
D:     }
E:     if (y < 0) {
F:        y = 1;
G:     }
H:     else {
I:       y = -1;
J:    }
K:     z = x+y;
L:     assert(x+y+z ≥ 0)
M:   }
```