



Ch 4: Program Comprehension

Part 2: Program Representations

Why Program Representations

- ❑ Original representations
 - Source code (cross languages).
 - Binaries (cross machines and platforms).
 - Source code / binaries + test cases.
- ❑ They are hard for machines to analyze.
- ❑ Software is translated into certain representations before analyses are applied.

Program Representations

- Before we can reason about programs, we must have a vocabulary and a model to analyze
- Difficult models:
 - Compiled binaries
 - Source code
- A **good** representation should make explicit the relationships you want to analyse

Comprehension Tasks

During program comprehension, we:

- Explore the program code non-linearly
- Derive a variety of views of the program code
 - To focus on particular aspects
 - To remove irrelevant details
- Formulate hypotheses and search for evidence
- Link program constructs to real world concepts
 - E.g. var SAL refers to the salary of an employee

Many of these involve repetitive tasks that are more quickly and more reliably performed by a software tool -

Types of Tool

- Software visualisation tools

 - Support browsing and exploration of the software

- Static analysis tools

 - Extract information from program code

- Dynamic analysis tools

 - Extract information from individual executions of the code

- Knowledge-based repositories

 - Store knowledge about the domain, and

 - Document the process of understanding

Program Representations

Static program representations

- Abstract syntax tree;
- Control flow graph;
- Program dependence graph;
- Call graph;

Dynamic program representations

- Control flow trace, address trace and value trace;
- Dynamic dependence graph;
- Whole execution trace;

Program Representations

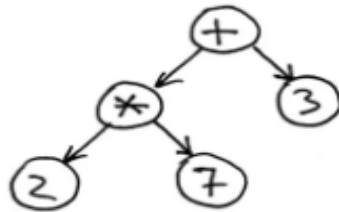
Core graph representations for analysis:

- 1) Abstract Syntax Trees
- 2) Control Flow Graphs
- 3) Program Dependence Graphs
- 4) Call Graphs

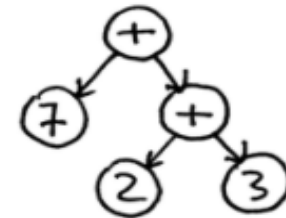
1) Abstract Syntax Trees

- Lifts the source into a canonical tree form

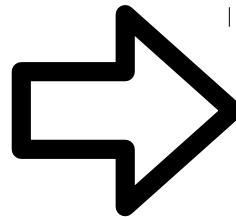
$2 * 7 + 3$



$7 + ((2 + 3))$



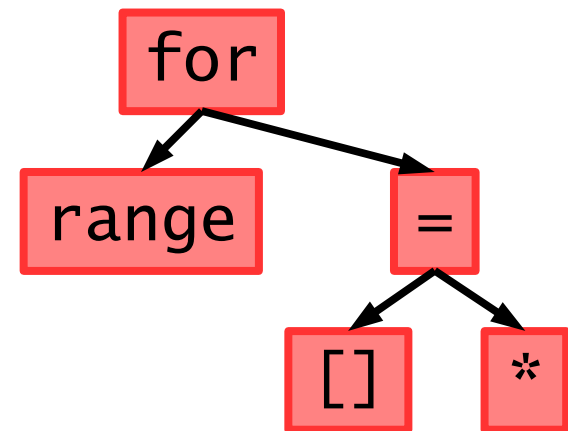
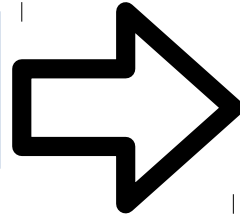
```
for i in range(5,10):  
    a[i] = i * 5
```



1) Abstract Syntax Trees

- Lifts the source into a canonical treeform
 - **Internal** nodes are operators, statements, etc.

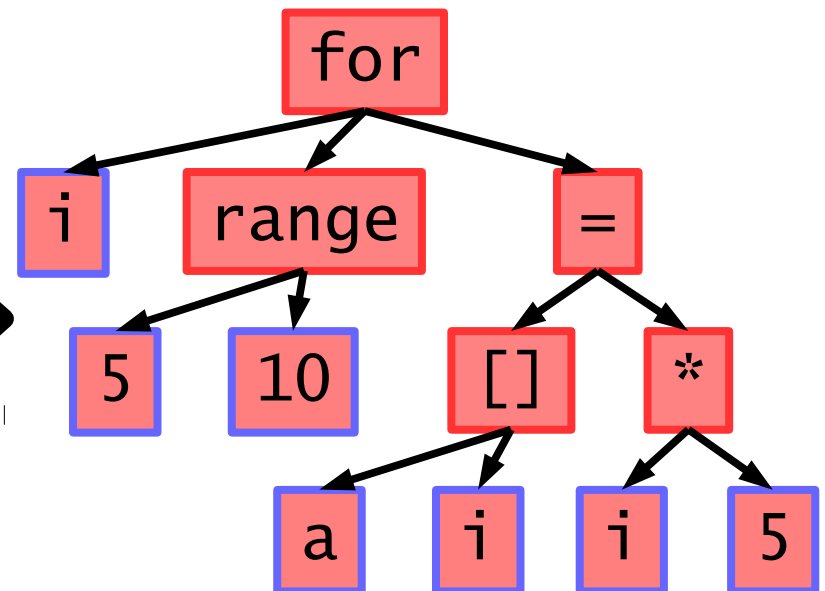
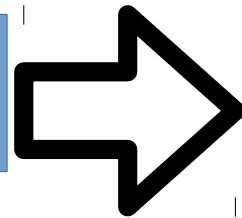
```
for i in range(5,10):  
    a[i] = i * 5
```



1) Abstract Syntax Trees

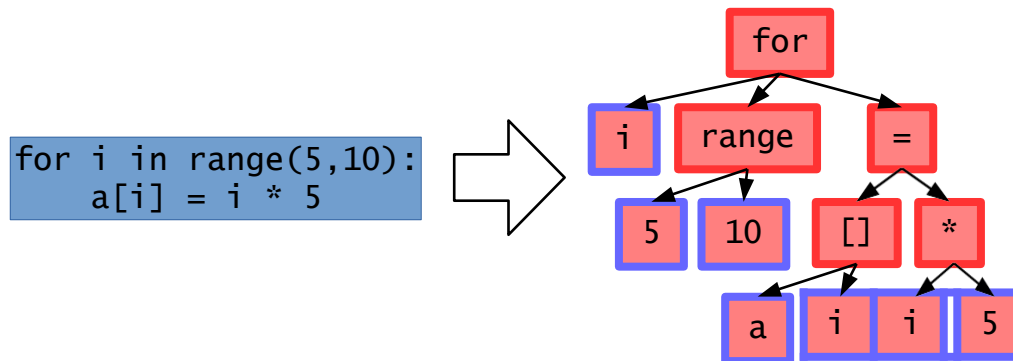
- Lifts the source into a canonical treeform
 - **Internal** nodes are operators, statements, etc.
 - **Leaves** are values, variables, operands

```
for i in range(5,10):  
    a[i] = i * 5
```



1) Abstract Syntax Trees

- Lifts the source into a canonical tree
- form Used for syntax analysis & transformation:

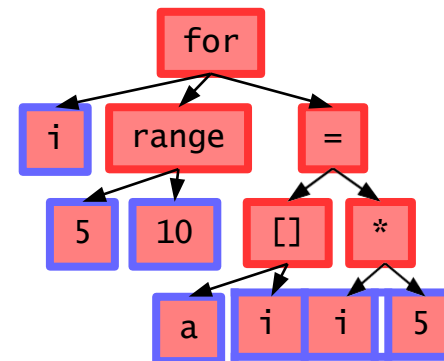
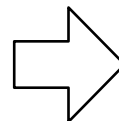


1) Abstract Syntax Trees

Used for syntax analysis & transformation:

- Simple bug patterns
- Style checking
- Refactoring

```
for i in range(5,10):  
    a[i] = i * 5
```

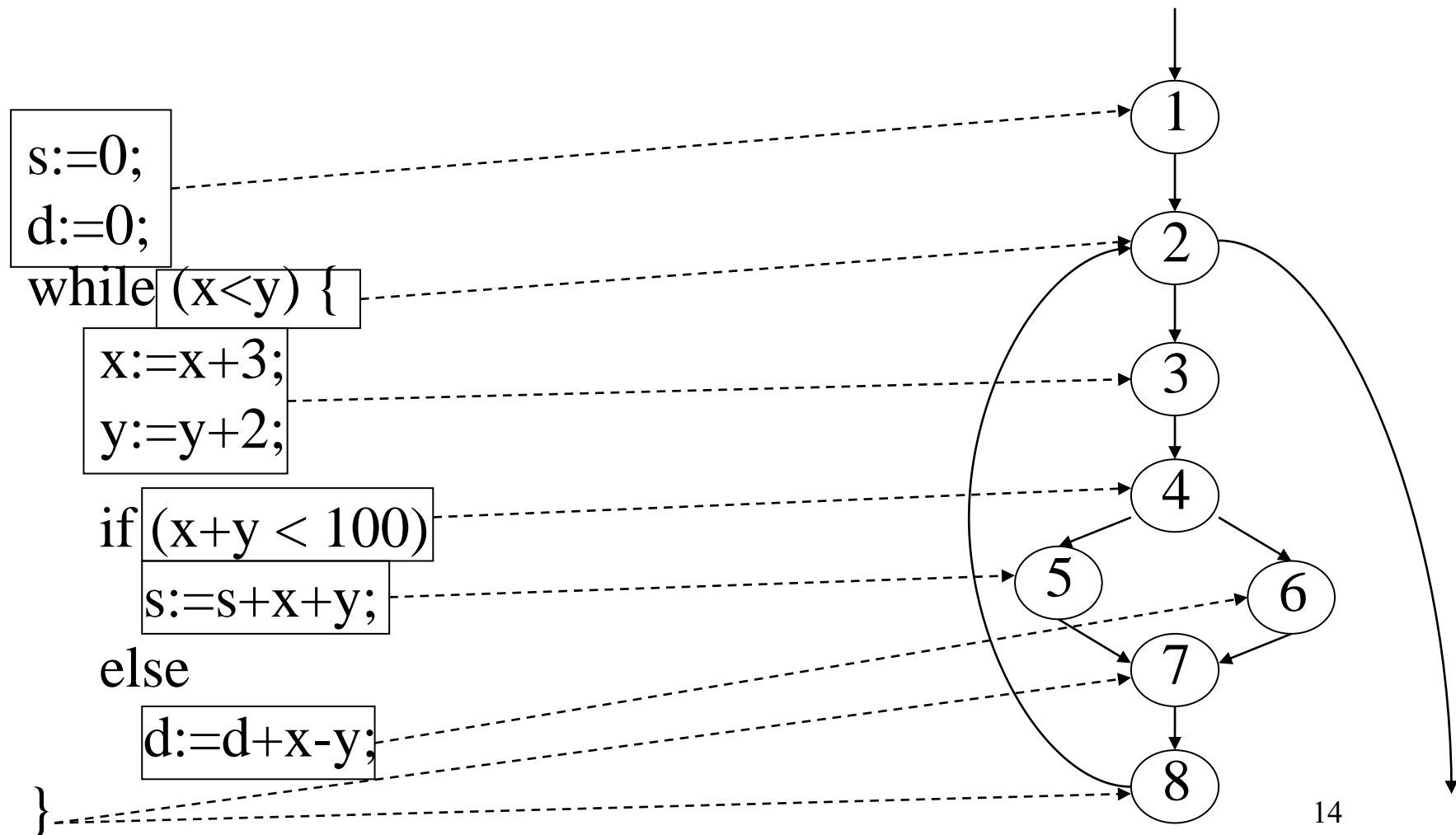


2) Control Flow Graph

The most commonly used program representation.
It is widely used in program analysis, malware analysis, testing.

A **control flow graph** (or flow graph) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N . We often write $G = (N, E)$ to denote a flow graph G with nodes given by N and edges by E .

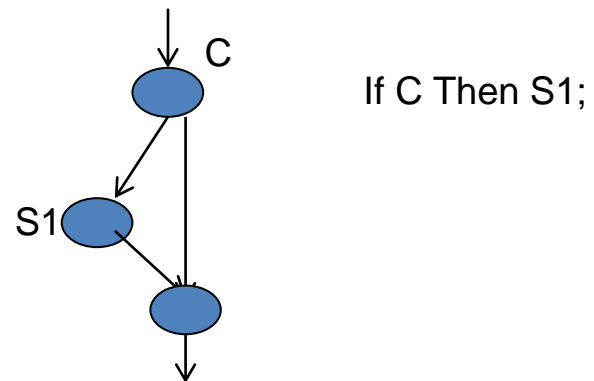
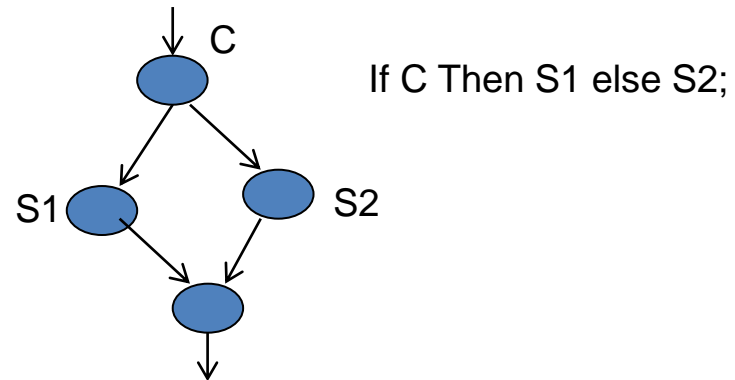
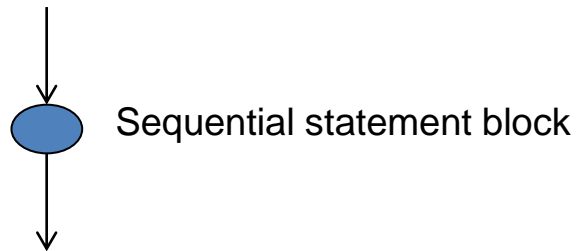
Example of a Control Flow Graph (CFG)



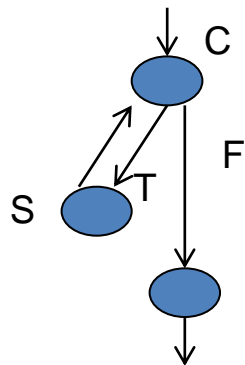
Flow Graph Notation

- ▣ A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- ▣ A node containing a simple conditional expression is referred to as a predicate node
 - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented **by a separate predicate node**
 - A predicate node has two edges leading out from it (True and False)
- ▣ An edge, or a link, is a an arrow representing flow of control in a specific direction
 - An edge must start and terminate at a node
 - An edge does not intersect or cross over another edge

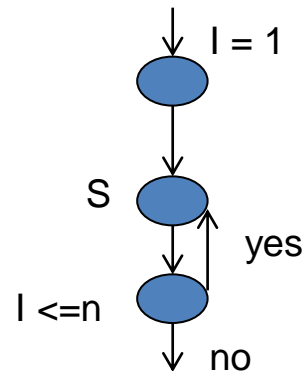
Program Flow Graph (Control Flow Diagram)



Program Flow Graph (Control Flow Diagram)

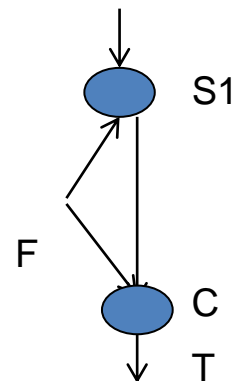


While C do S;



For loop:

for I = 1 to n do S;



Do loop:

do S1 until C;

2) Control Flow Graphs

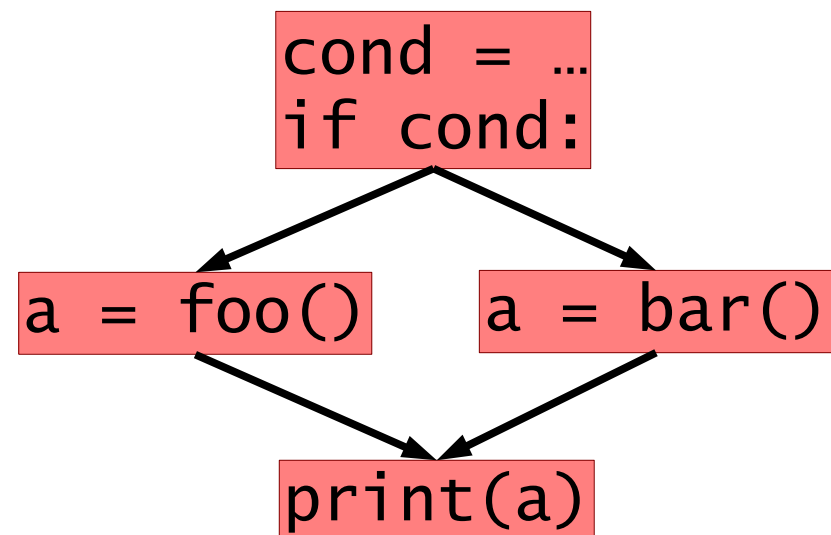
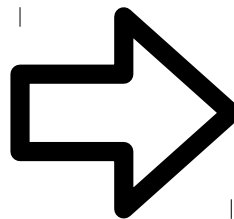
- Express the possible decisions and possible paths through a program

```
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```

2) Control Flow Graphs

- Express the possible decisions and possible paths through a program

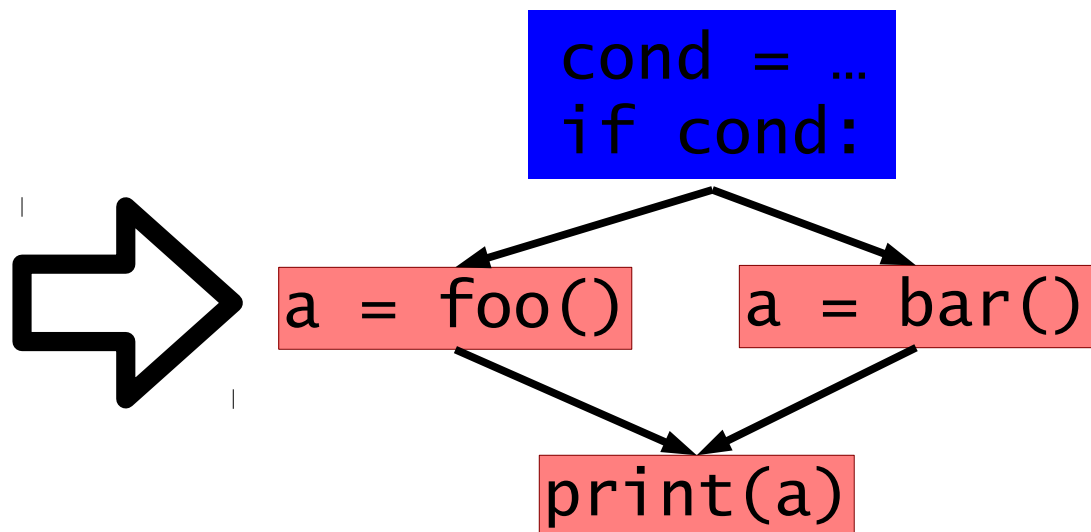
```
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```



2) Control Flow Graphs

- Express the possible decisions and possible paths through a program
 - Basic Blocks** (Nodes) are straight line code

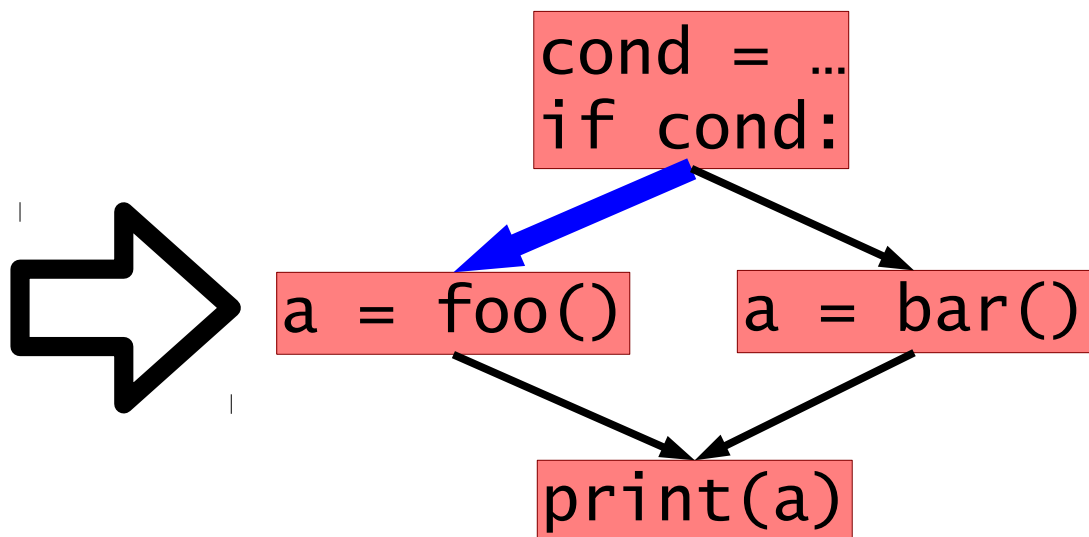
```
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```



2) Control Flow Graphs

- Express the possible decisions and possible paths through a program
 - *Basic Blocks* (Nodes) are straight line code
 - *Edges* show how decisions can lead to different basic blocks

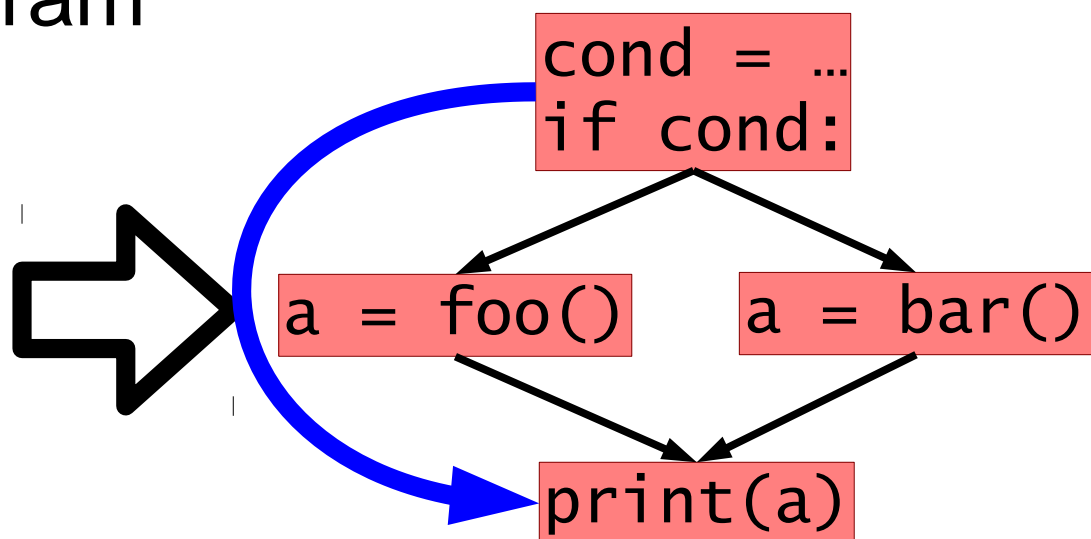
```
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```



2) Control Flow Graphs

- Express the possible decisions and possible paths through a program
 - *Basic Blocks* (Nodes) are straight line code
 - *Edges* show how decisions can lead to different basic blocks
 - *Paths* through the graph are potential paths through the program

```
cond = input()
if cond:
    a = foo()
else:
    a = bar()
print(a)
```



2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

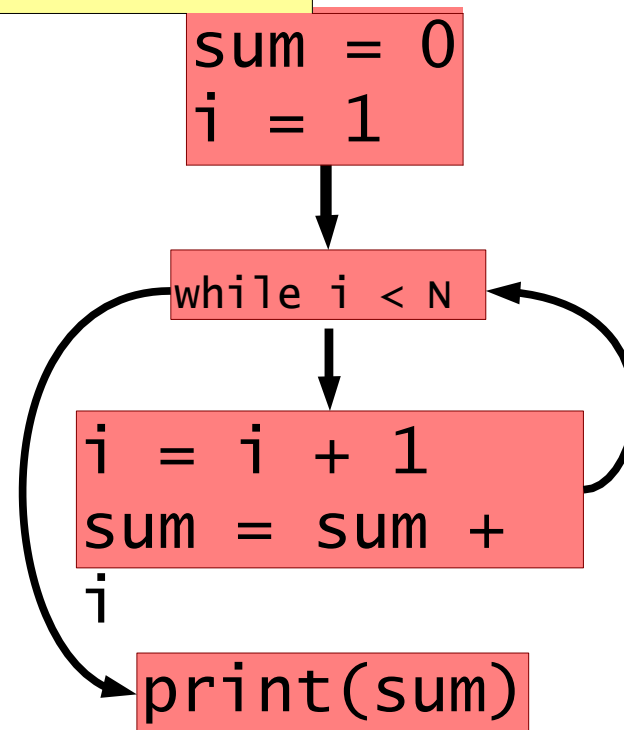
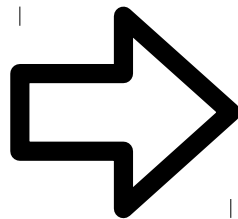
```
sum  = 0
i    = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```

2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

Example

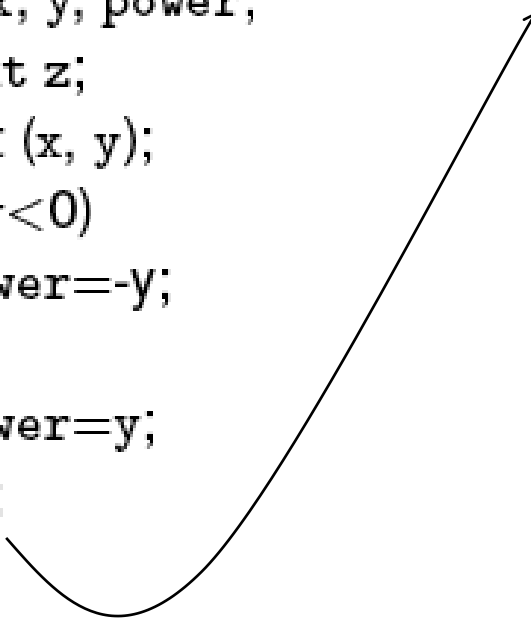
```
sum = 0
i = 1
while i < N:
    i = i + 1
    sum = sum + i
print(sum)
```



2) Control Flow Graphs (CFGs)

Example

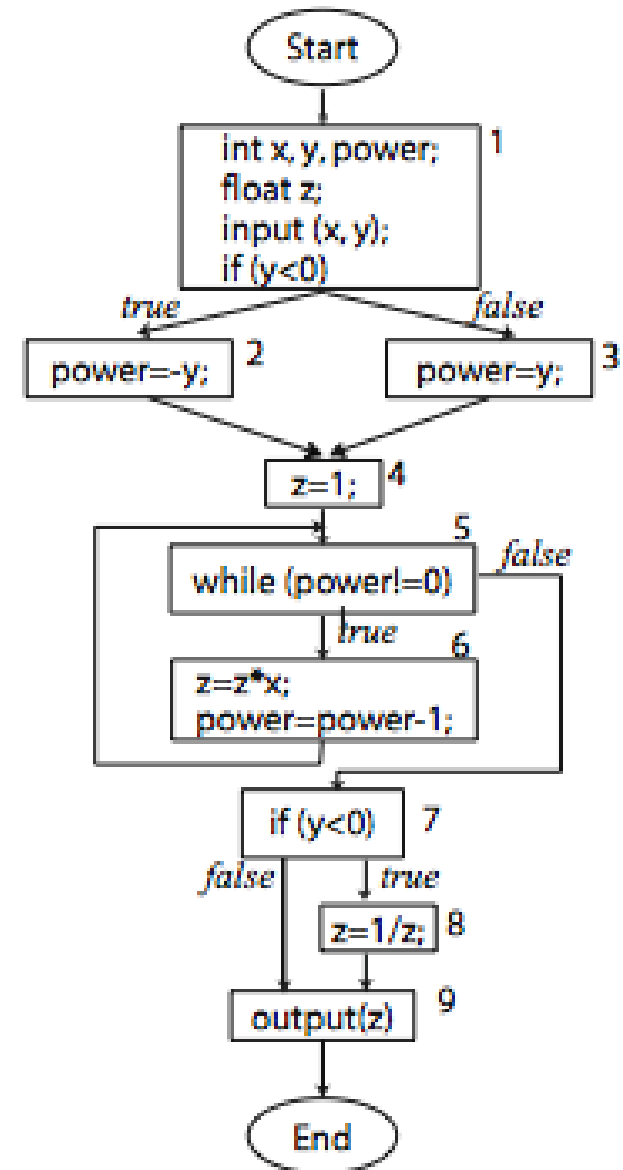
```
1  begin
2    int x, y, power;
3    float z;
4    input (x, y);
5    if (y<0)
6      power=-y;
7    else
8      power=y;
9    z=1;
10   while (power!=0){
11     z=z*x;
12     power=power-1;
13   }
14   if (y<0)
15     z=1/z;
16   output(z);
17 end
```



CFG Example

$N = \{\text{Start}, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{End}\}$

$E = \{(\text{Start}, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, \text{End})\}$

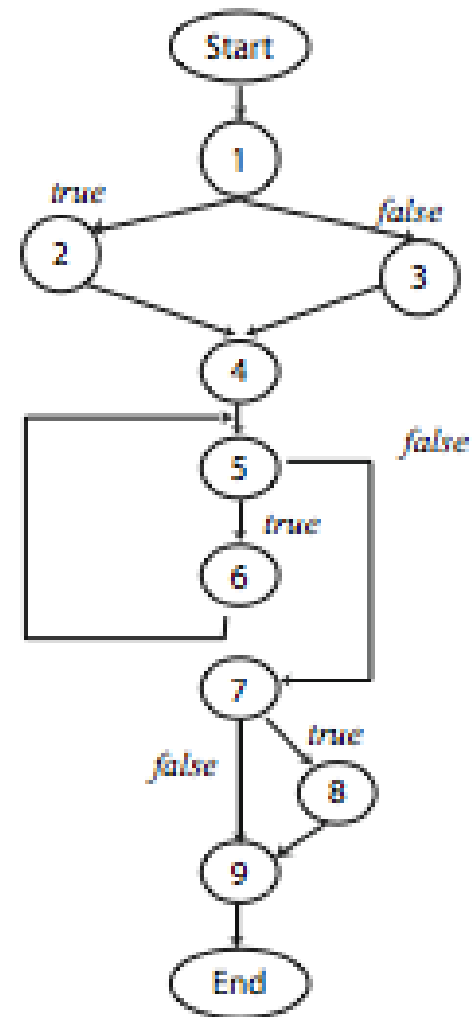


CFG Example

Same CFG with statements removed.

$N = \{\text{Start}, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{End}\}$

$E = \{(\text{Start}, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, \text{End})\}$



3) Program Dependence Graph (PDG)

- A **Program Dependence Graph** captures how instructions can influence each other

Instruction X depends on Y if Y can influence X

- Nodes are instructions
- An edge $Y \rightarrow X$ shows that Y influences X

2 main types of influence:

- Data dependence
- Control dependence – influence through decisions

3) Program Dependence Graph (PDG)

Data dependence:

Where did these values come from?

Control dependence:

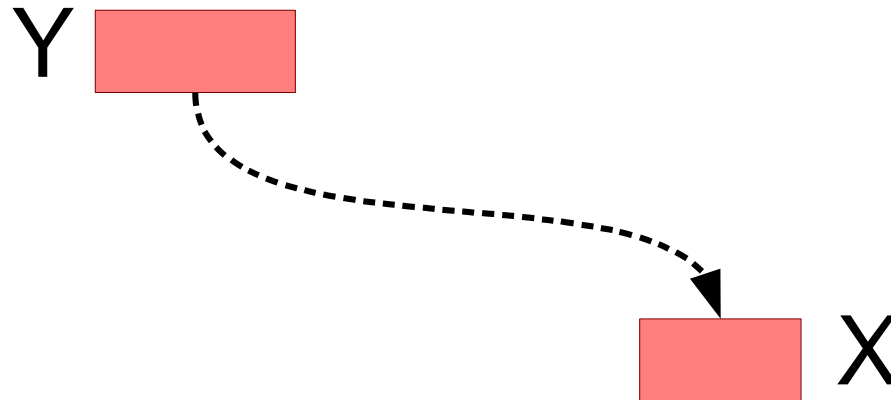
Which statement controls whether this statement executes?

- Nodes: as in the CFG
- Edges: unlabelled, from entry/branching points to controlled blocks

Data Dependence

X data depends on Y if

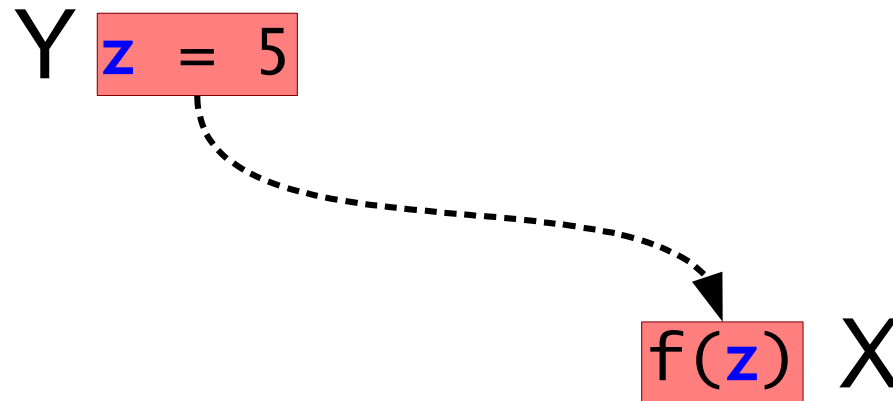
- There exists a path from Y to X in the CFG



Data Dependence

X data depends on Y if

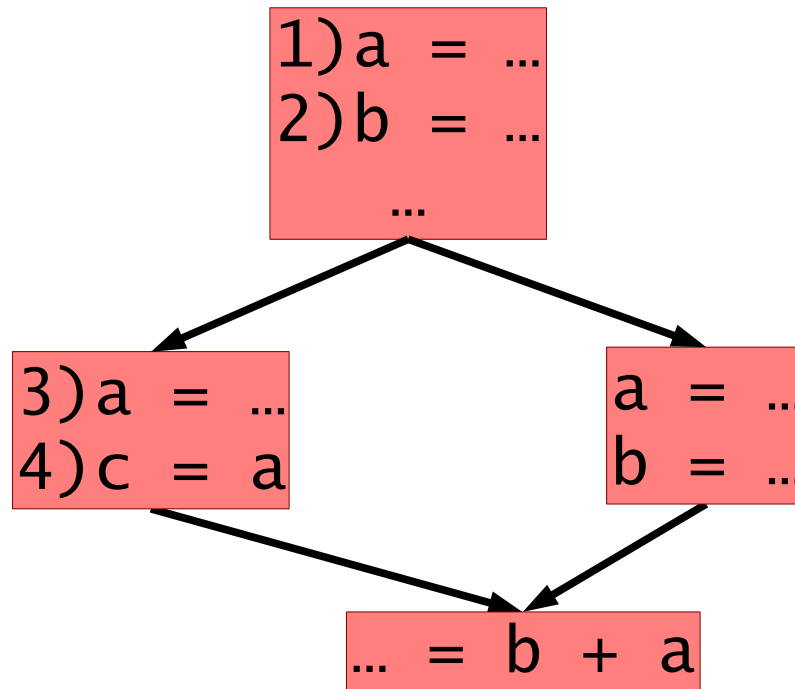
- There exists a path from Y to X in the C F G
 - Variable/value definition at Y is used at X



Data Dependence

X data depends on Y if

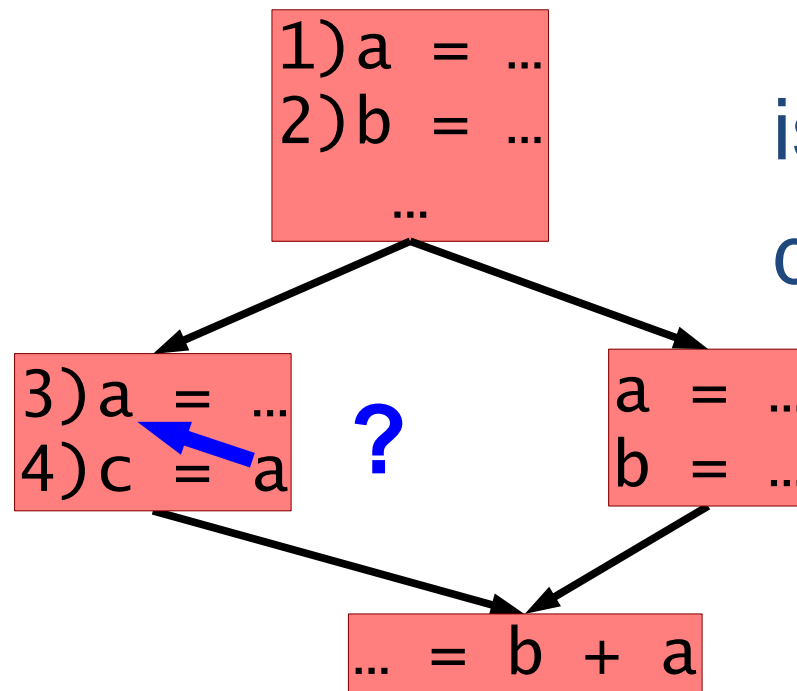
- There exists a path from Y to X in the C F G
 - Variable/value definition at Y is used at X



Data Dependence

X data depends on Y if

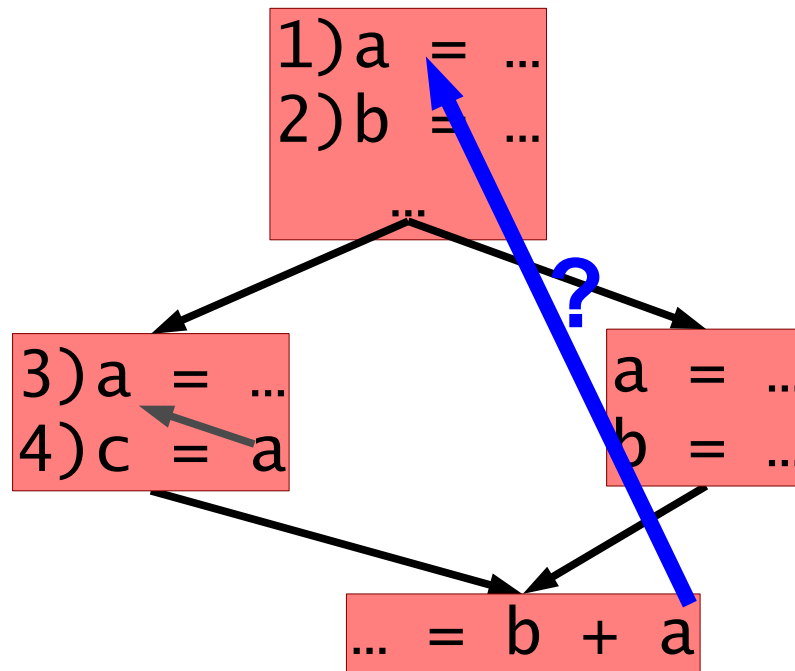
- There exists a path from Y to X in the C F G
- Variable/value definition at Y is used at X



is this valid data
dependence?

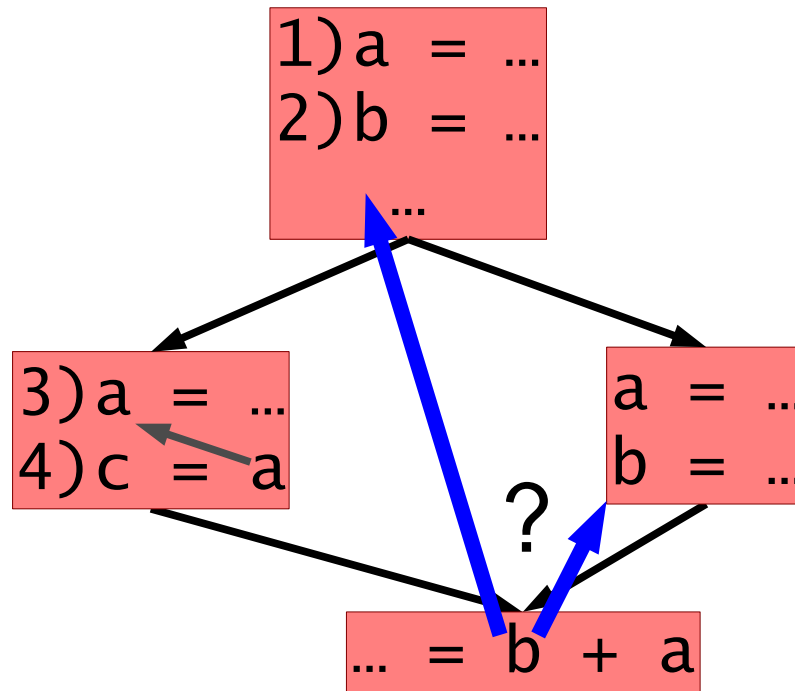
Data Dependence

- How about, is this valid data dependence?



Data Dependence

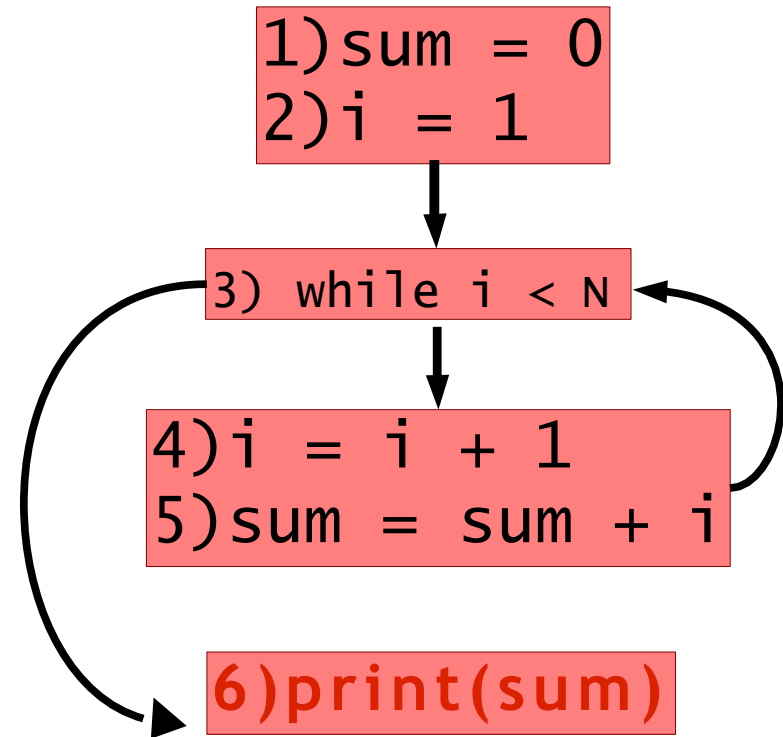
- How about, is this valid data dependence?



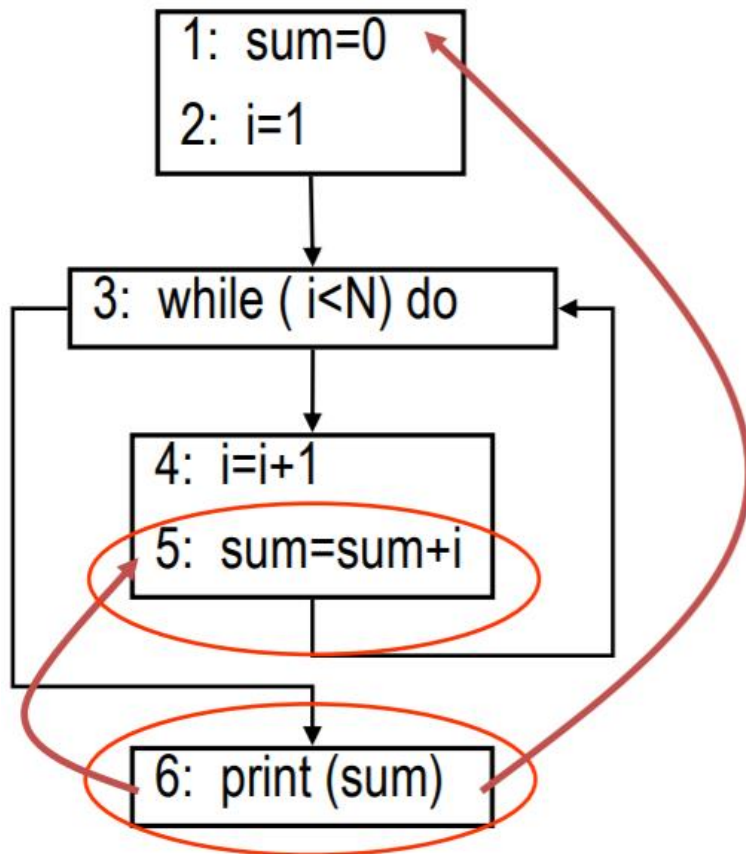
Data Dependence -Example

Determine which statements influence which other statements? In terms of data control

```
1)sum = 0
2)i = 1
3)while i < N:
4)    i = i + 1
5)    sum = sum + i
6)print(sum)
```



Data Dependence



5 is not data dependent on 2,
since *i* is re-defined at 4.

5 is data dependent on 1, and 4.

6 is data dependent on 5, and 1.

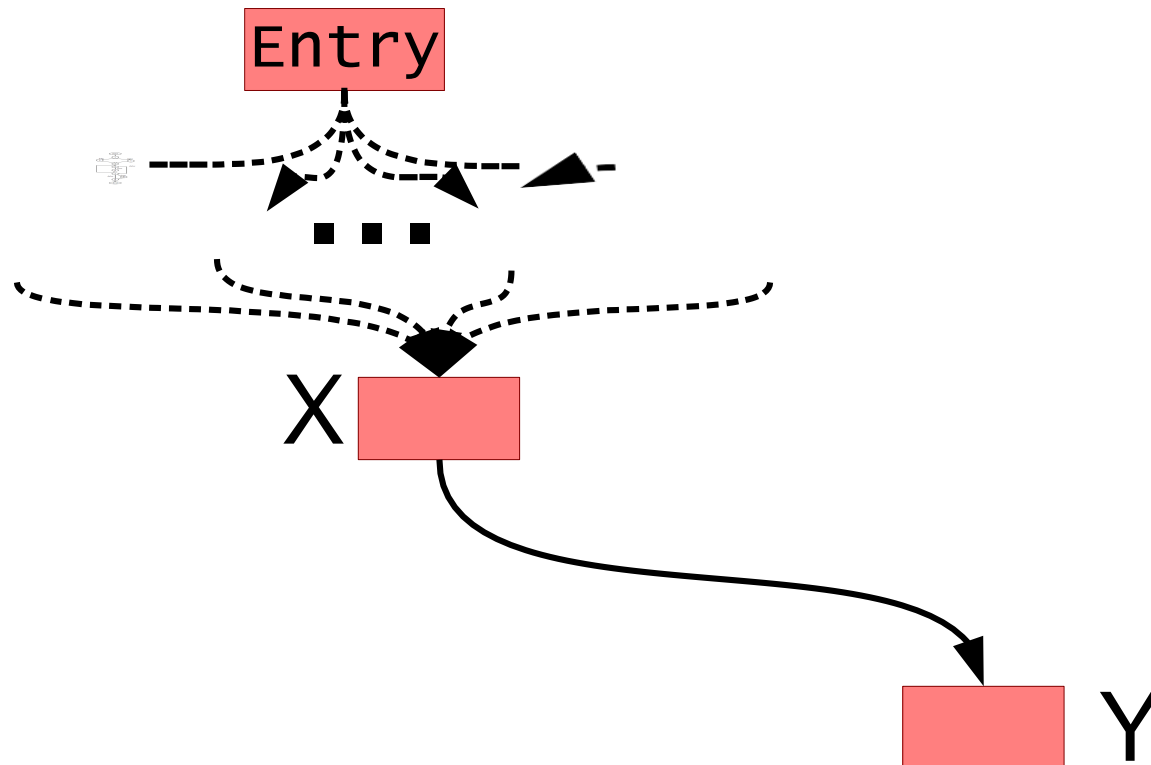
Control Dependence

- Recall:
- Control dependence captures how decisions influence program behaviour.
- We need a way of capturing this via graphs....

Control Dependence

Preliminary: X **dominates** Y if

- every path from the **entry node to Y** passes X
 - strict, normal, & immediate dominance

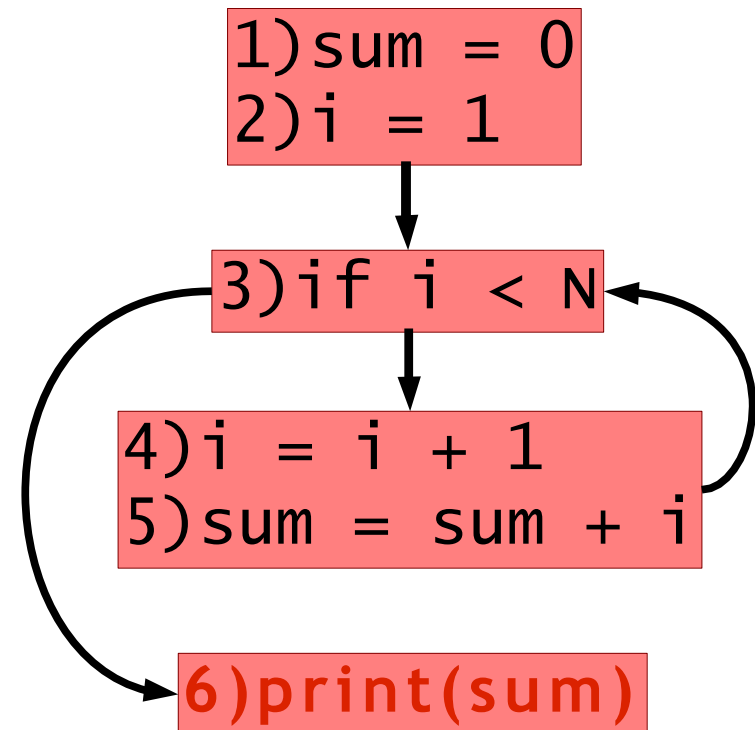


Control Dependence

X dominates Y if all possible program paths from START to Y have to pass X.

```
1)sum = 0
2)i = 1
3)while i < N:
4)    i = i + 1
5)    sum = sum + i
6)print(sum)
```

DOM(6)= ?



Control Dependence

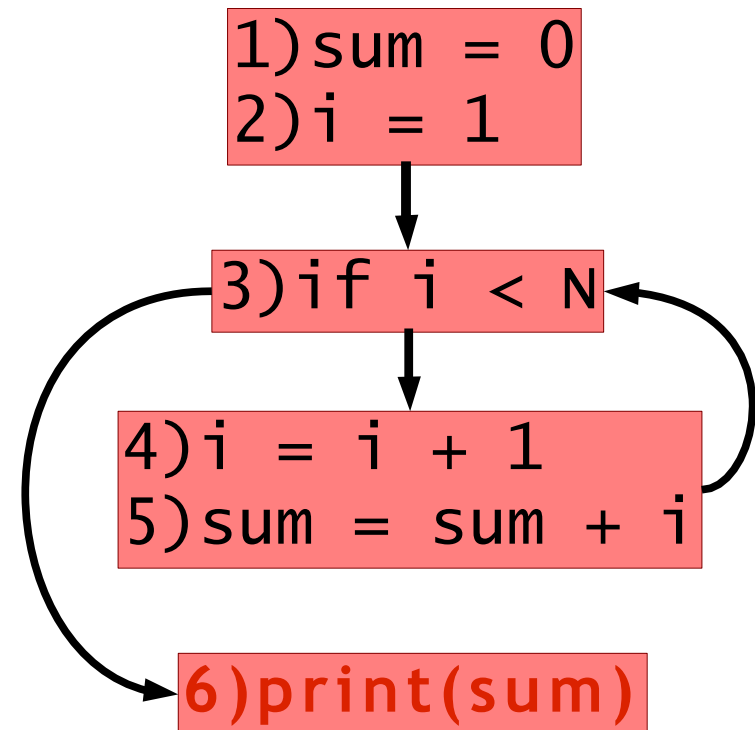
Preliminary: X dominates Y if

- every path from the entry node to Y passes X
 - strict, normal, & immediate dominance

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

$DOM(6) = \{1, 3, 6\}$

what is dominator of 6



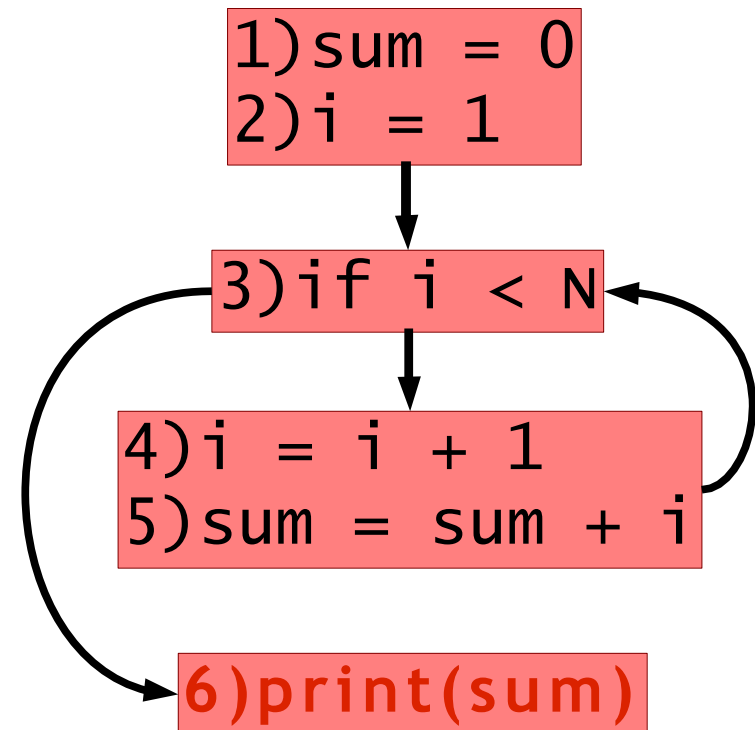
Note that a basic block is identified by the first statement in the block.

Control Dependence

- X strictly dominates Y if X dominates Y and $X \neq Y$

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

SDOM(6) = ?



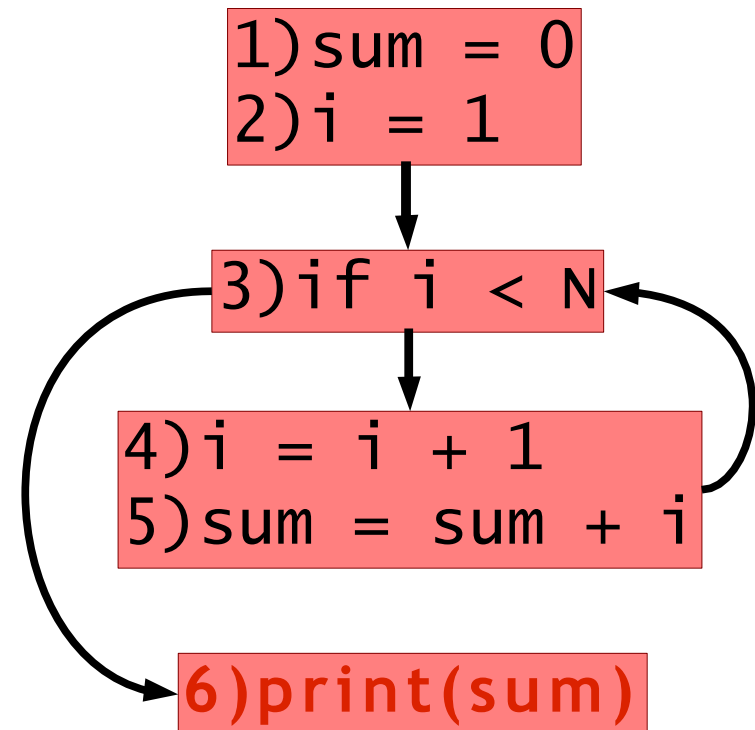
what is strictly dominator of 6

Control Dependence

- X strictly dominates Y if X dominates Y and $X \neq Y$

```
1)sum = 0
2)i = 1
3)while i < N:
4)    i = i + 1
5)    sum = sum + i
6)print(sum)
```

$SDOM(6) = \{ 1, 3 \}$

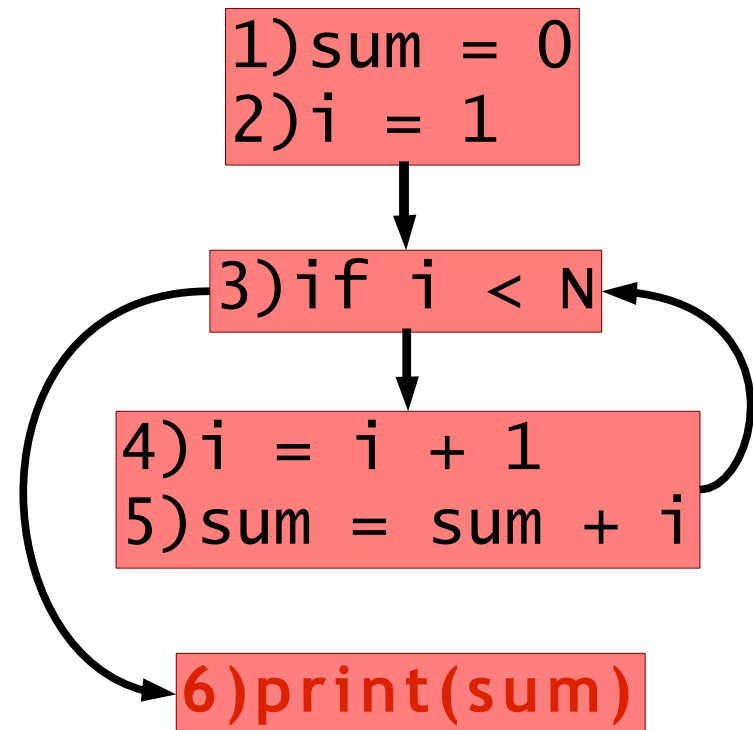


Control Dependence

X is the immediate dominator of Y if X is the last dominator of Y along a path from Start to Y.

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

IDOM(6)=?

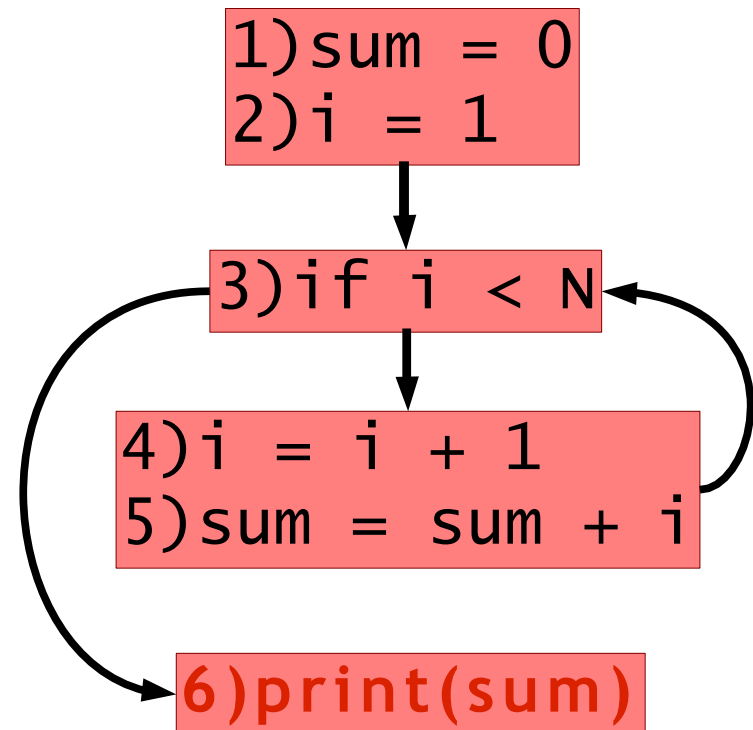


Control Dependence

X is the immediate dominator of Y if X is the last dominator of Y along a path from Start to Y.

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

IDOM(6) = {3}



Control Dependence

Preliminary: X *post* dominates Y if

- every path from the *Y to exit* passes X
 - strict, normal, & immediate dominance

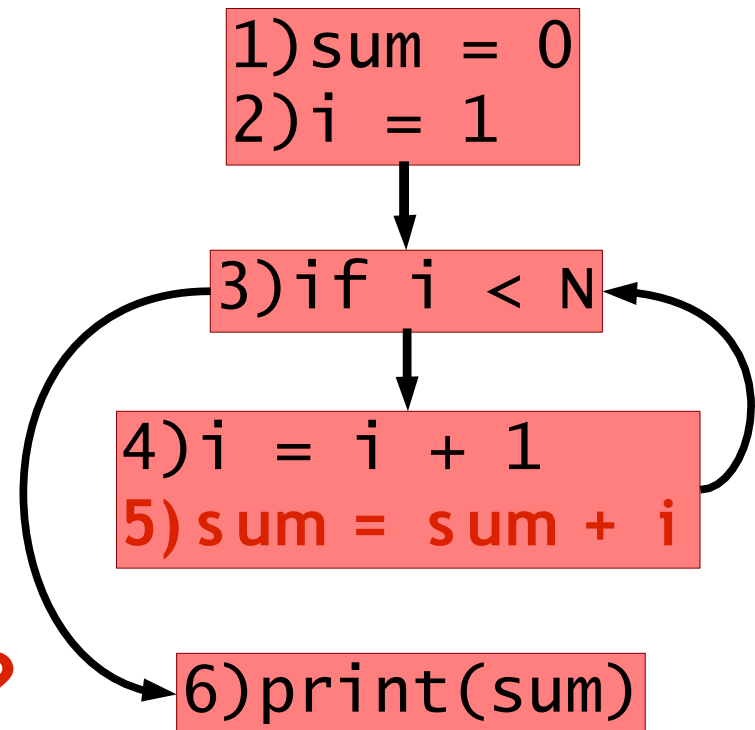
Control Dependence

X post-dominates Y if every possible program path from Y to End has to pass X. – Strict post-dominator, immediate post-dominance

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

PDOM(5)= ?

IPDOM(5)= ?

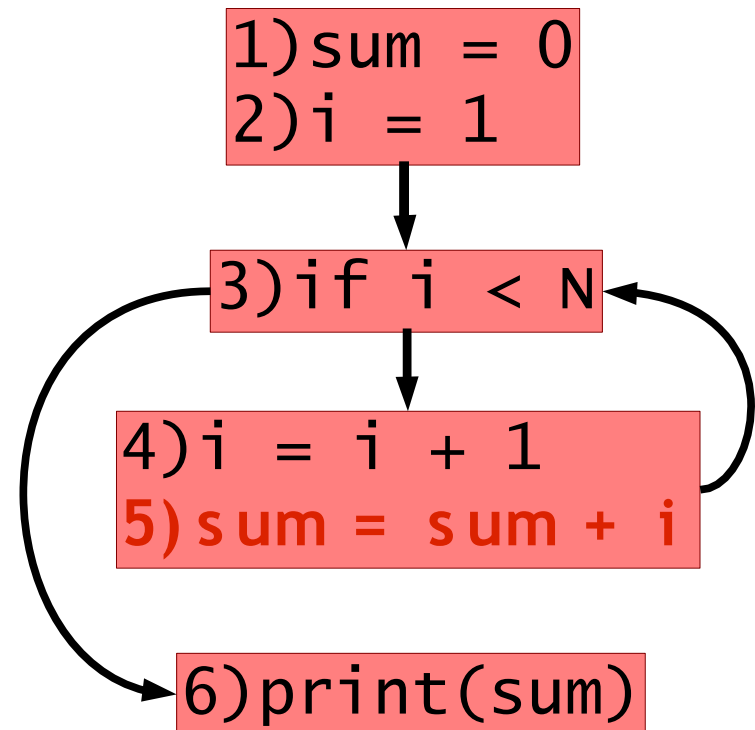


Control Dependence

X post-dominates Y if every possible program path from Y to End has to pass X. – Strict post-dominator, immediate post-dominance

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

$PDOM(5) = \{3, 5, 6\}$ $IPDOM(5) = 3$



Control Dependence (Finally)

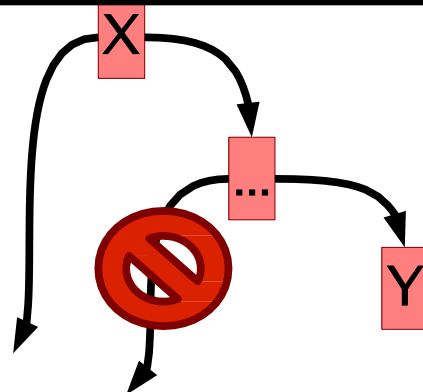
Y is **control dependent** on X if

- Defnition 1:

X directly decides whether Y executes

- Defnition 2:

- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X



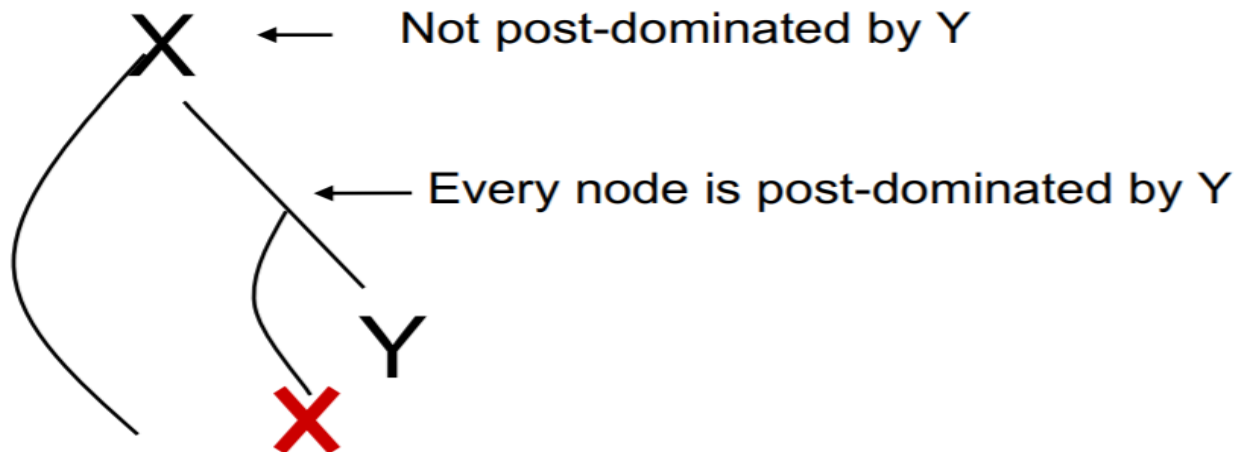
Control Dependence

Intuitively, Y is control-dependent on X iff X directly determines whether Y executes (statements inside one branch of a predicate are usually control dependent on the predicate)

X is not strictly post-dominated by Y \Rightarrow There is a path from X to End that does not pass Y or $X==Y$

there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

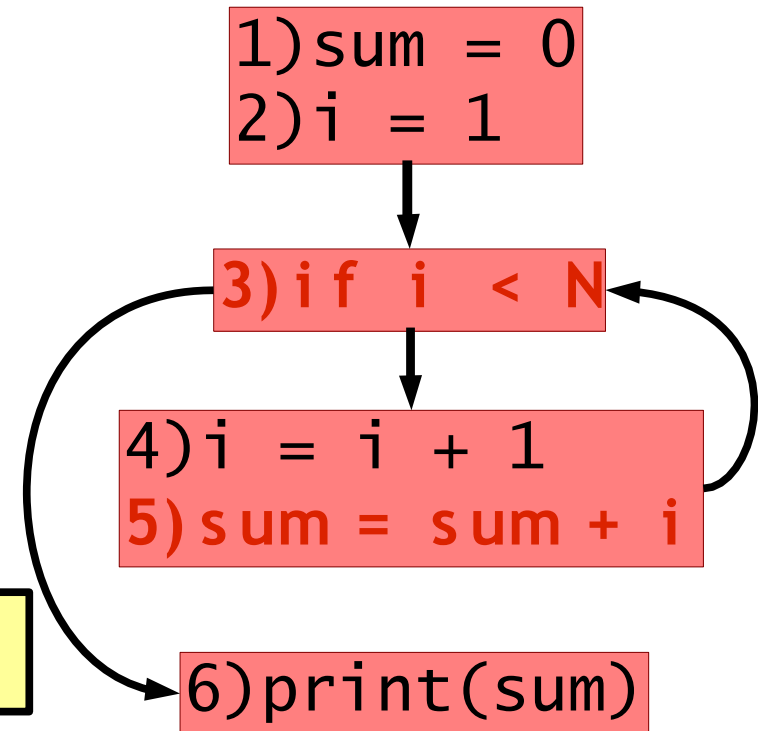
\Rightarrow No such paths for nodes in a path between X and Y.



Control Dependence

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

What is CD(5)? CD(3)

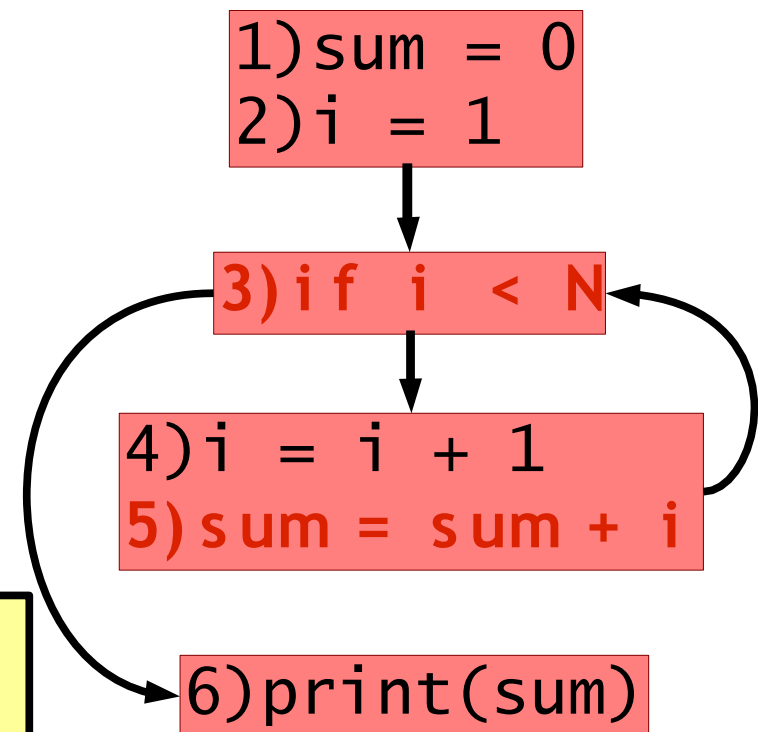


Control Dependence

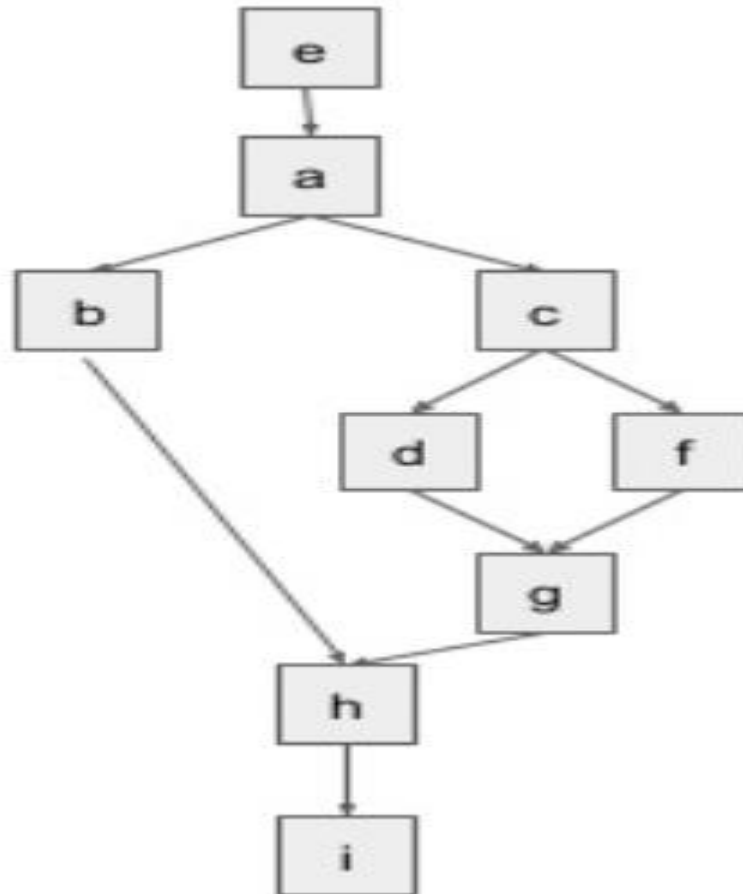
- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

```
1) sum = 0
2) i = 1
3) while i < N:
4)     i = i + 1
5)     sum = sum + i
6) print(sum)
```

What is $CD(5) = \{3\}$
 $CD(3) = \{3\}$



Example



DOM (d) = ?

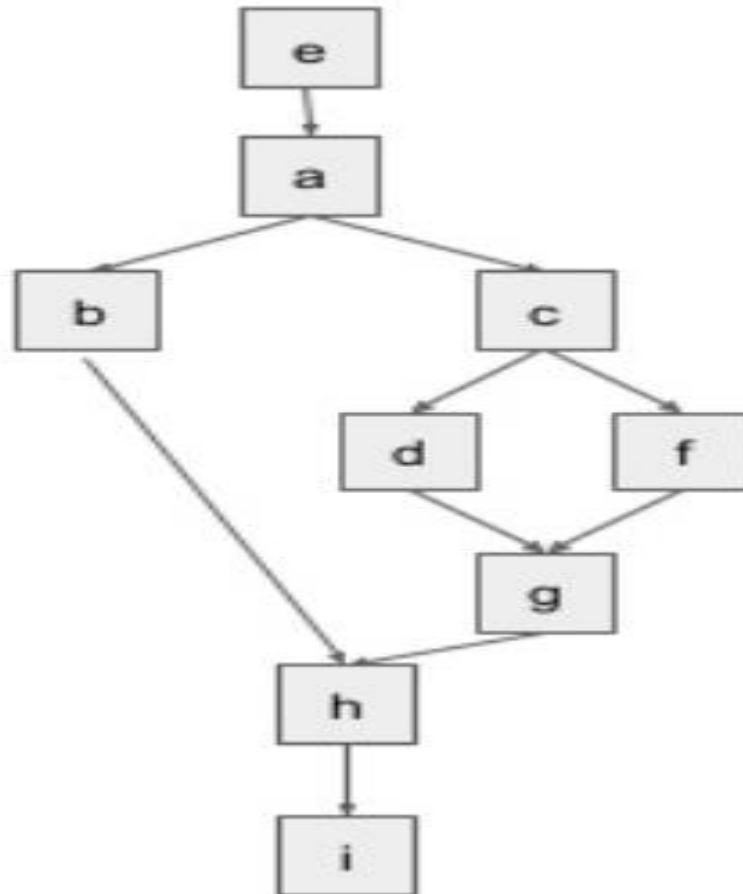
IDOM (d) or (g) = ?

SDOM (d) = ?

PDOM(b) = ?

CD(b) = ?

Example



$\text{DOM}(d) = \{e, a, c, d\}$

$\text{IDOM}(d) \text{ or } (g) = \{c\}$

$\text{SDOM}(d) = \{e, a, c\}$

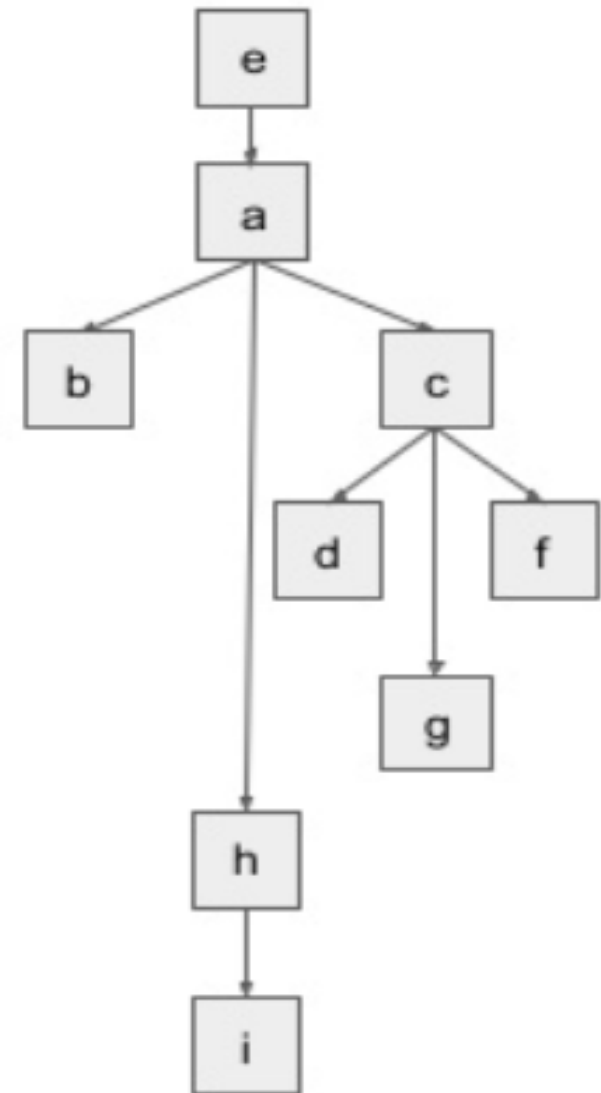
$\text{PDOM}(b) = \{h, i\}$

$\text{CD}(b) = \{a\}$

Example

Dominator Tree

- Node
 - Every node of the flow graph
- Edge
 - If n is in $IDOM(m)$, then the dominator tree has an edge from n to m .



3) Program Dependence Graph(PDG)

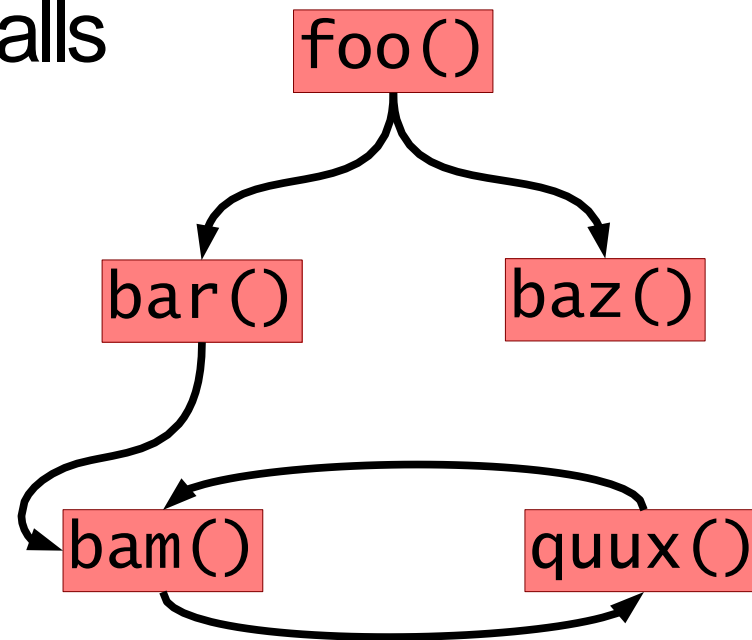
Debugging: What may have caused a bug?

Security: Can sensitive information leak?

Testing: How can I reach a statement? ...

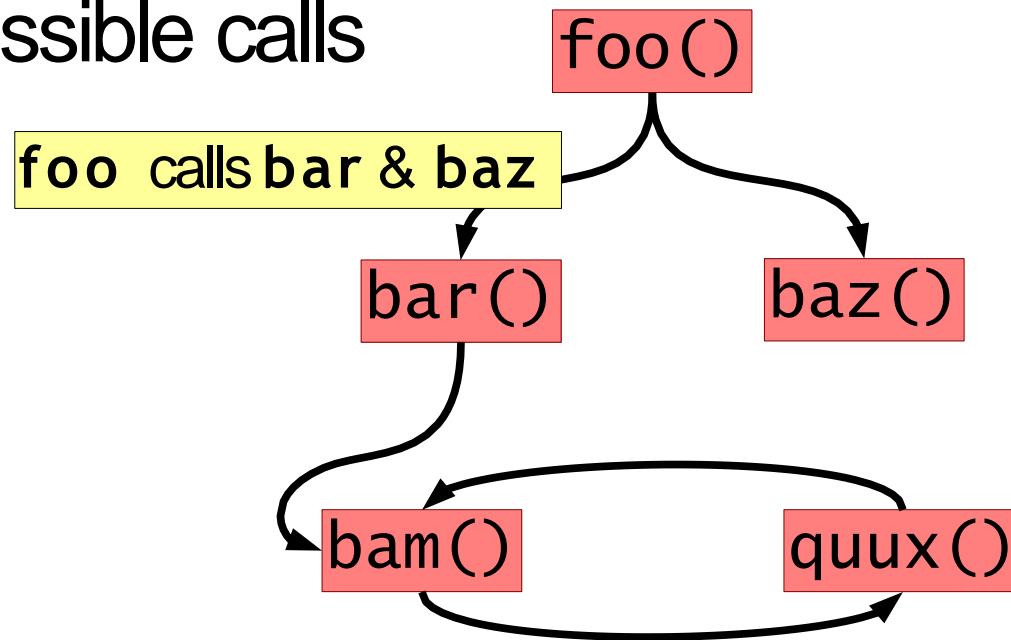
4) Call Graph (Multigraph)

- Captures the composition of a program
 - Nodes are functions
 - Edges show possible calls



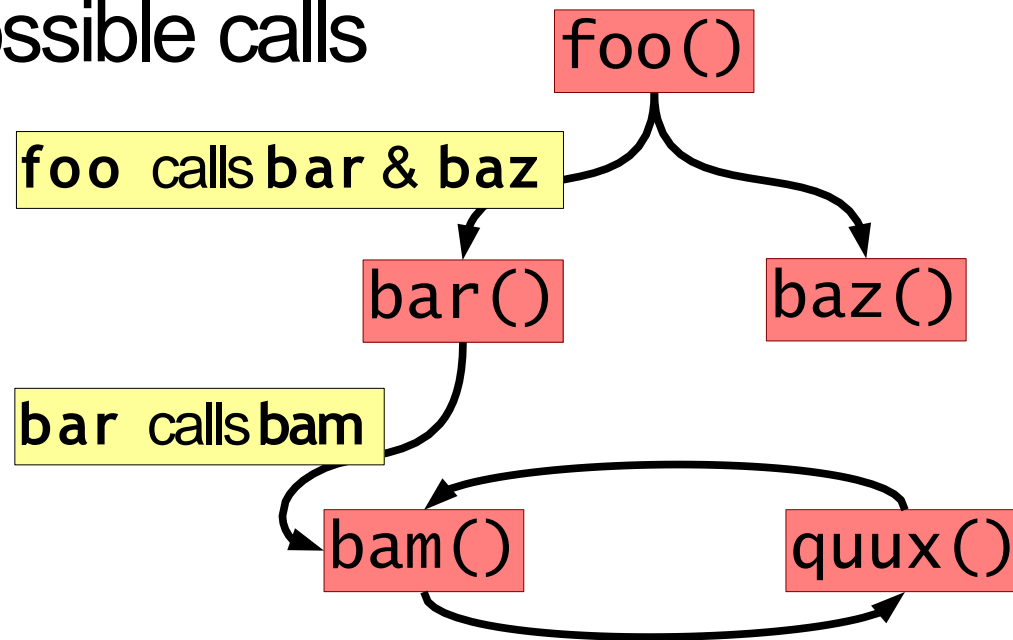
4) Call Graph (Multigraph)

- Captures the composition of a program
 - Nodes are functions
 - Edges show possible calls



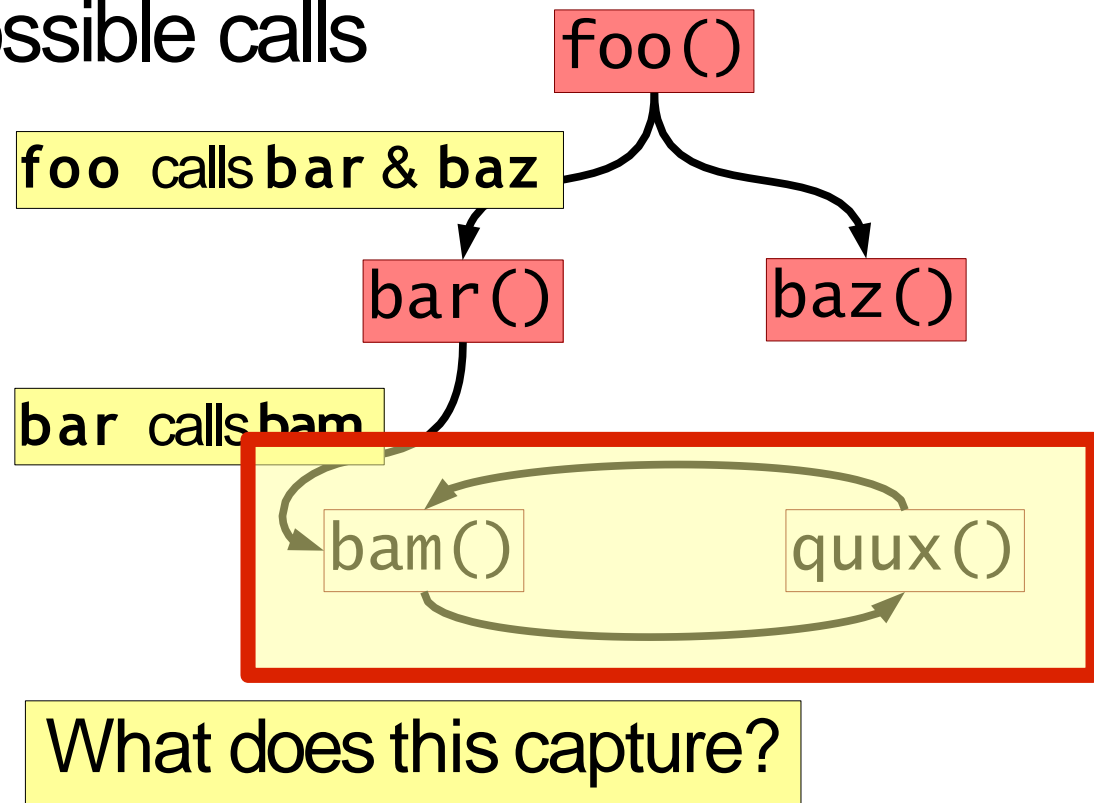
4) Call Graph (Multigraph)

- Captures the composition of a program
 - Nodes are functions
 - Edges show possible calls



4) Call Graph (Multigraph)

- Captures the composition of a program
 - Nodes are functions
 - Edges show possible calls



A **cycle** in the graph indicates recursive procedure calls

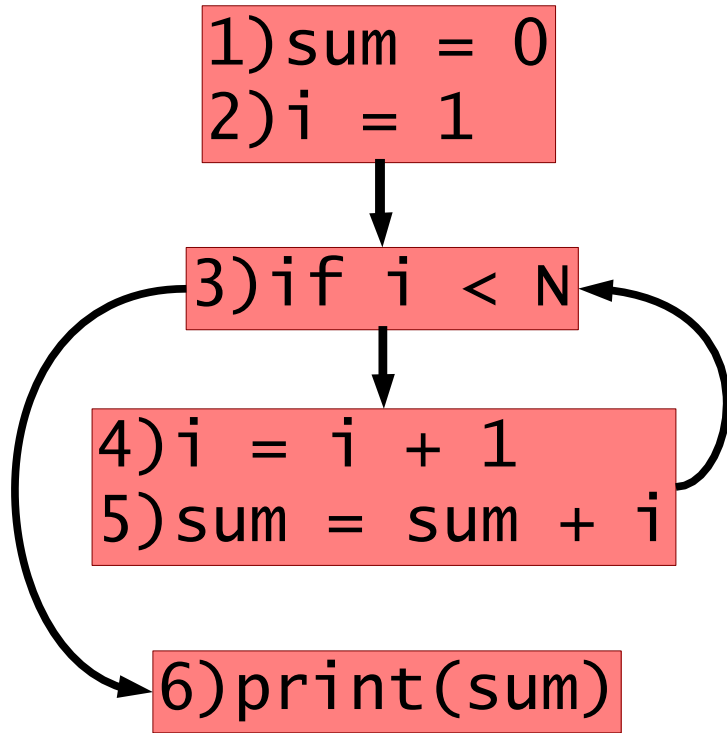
Execution Representations

- Program representations are static
 - All possible program behaviours at once
 - Usually projected onto the CFG

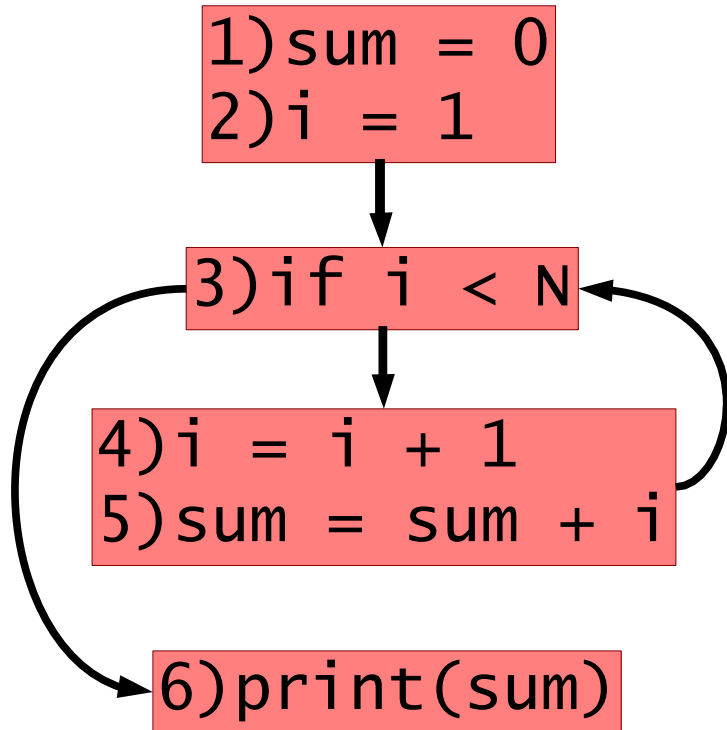
Execution Representations

- Program representations are **static**
 - All possible program behaviours at once
 - Usually projected onto the CFG
- Execution representations are **dynamic**
 - Only the behaviour of a single real execution
 - Multiple instances of an instruction occur multiple times

Control Flow Trace

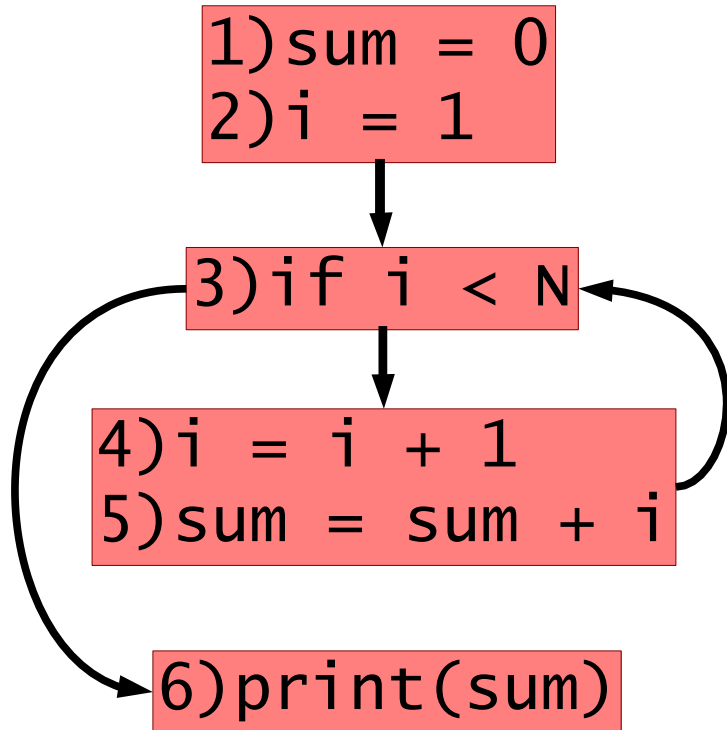


Control Flow Trace



1) sum = 0
2) i = 1
3) if i < N
4) i = i + 1
5) sum = sum + i
3) if i < N
4) i = i + 1
5) sum = sum + i
3) if i < N
6) print(sum)

Control Flow Trace



1₁ 2₁ 3₁ 4₁ 5₁ 3₂ 4₂ 5₂ 3₃ 6₁

1₁ 3₁ 4₁ 3₂ 4₂ 3₃ 6₁

TTF

1) sum = 0
2) i = 1

3) if i < N

4) i = i + 1
5) sum = sum + i

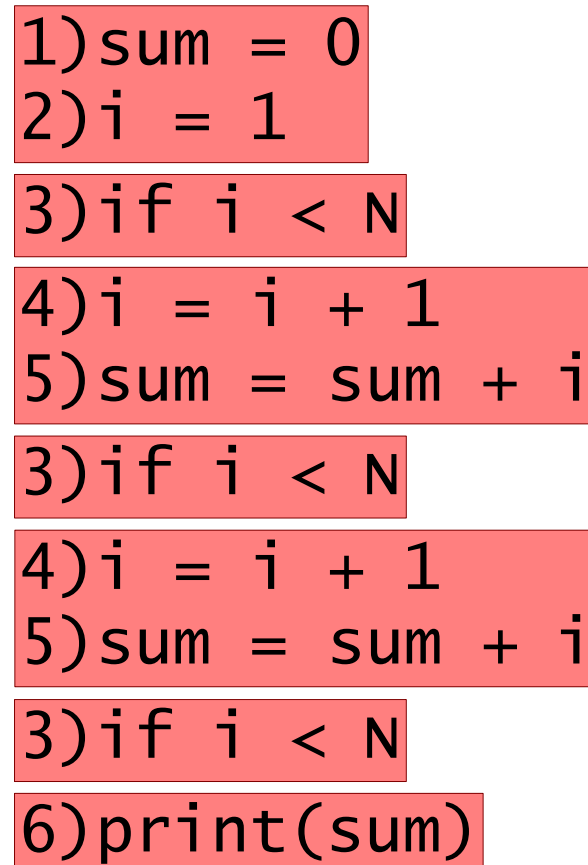
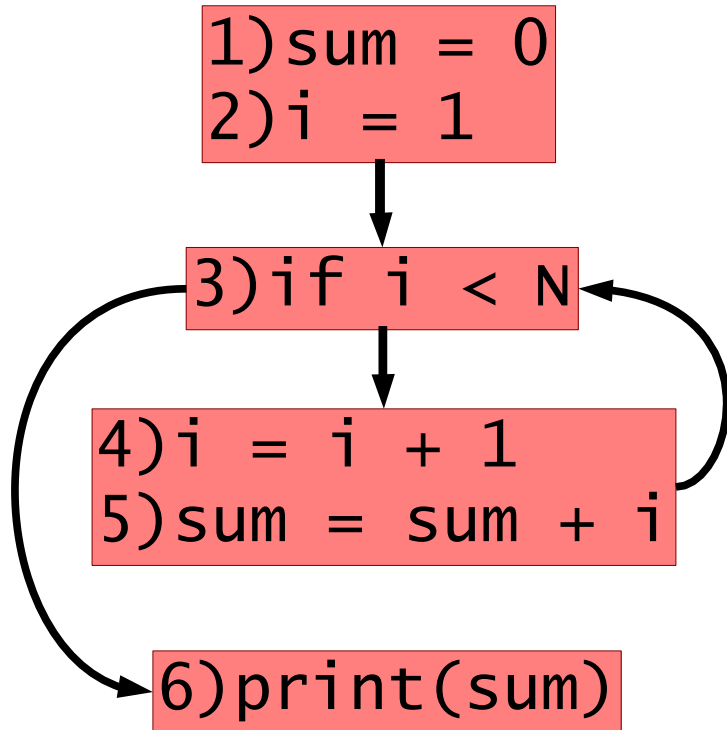
3) if i < N

4) i = i + 1
5) sum = sum + i

3) if i < N

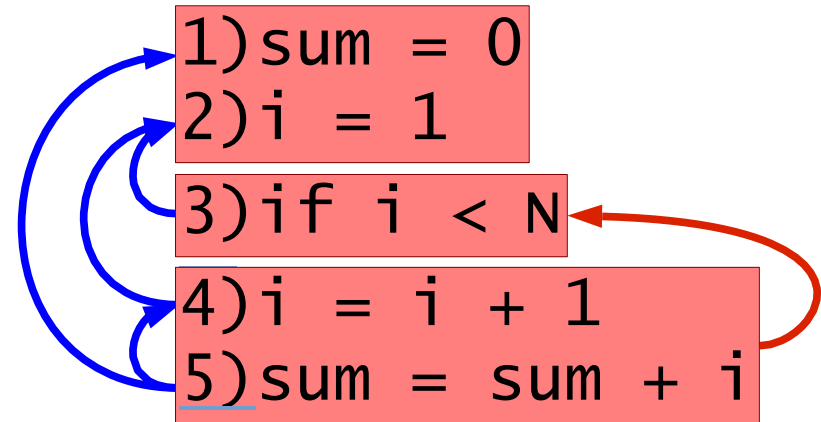
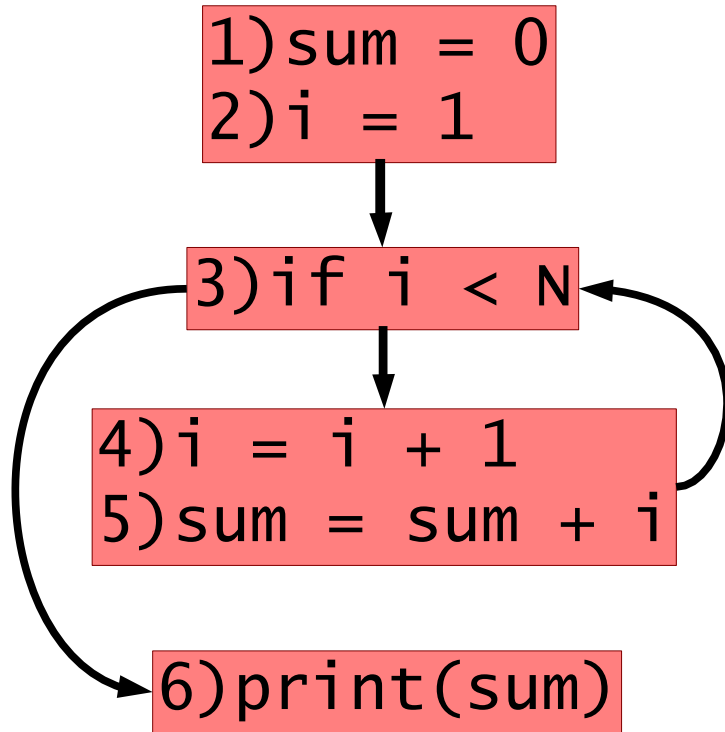
6) print(sum)

Control Flow Trace

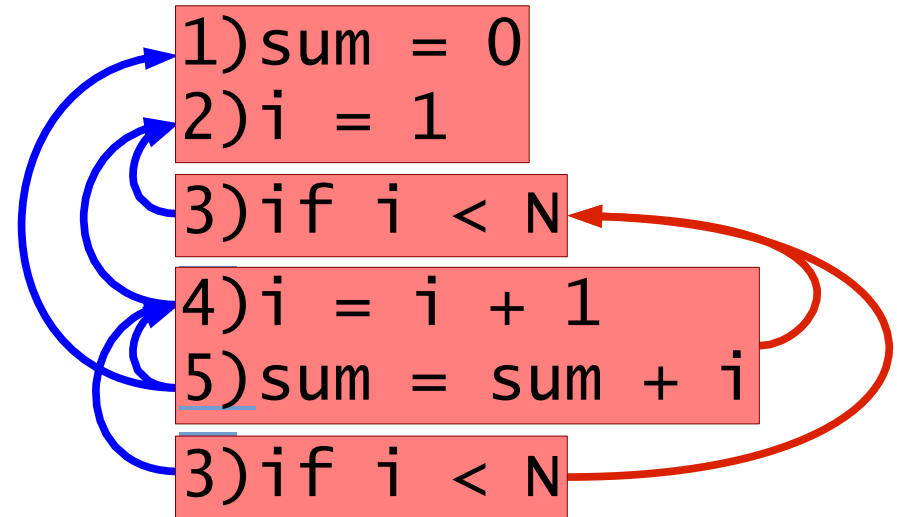
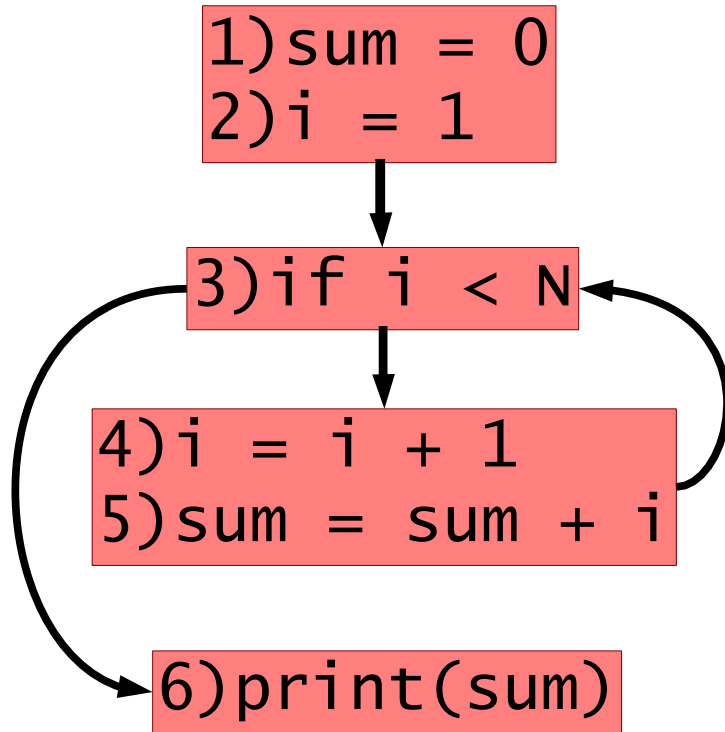


$$\left. \begin{array}{l} 1_1 \ 2_1 \ 3_1 \ 4_1 \ 5_1 \ 3_2 \ 4_2 \ 5_2 \ 3_3 \ 6_1 \\ 1_1 \ 3_1 \ 4_1 \ 3_2 \ 4_2 \ 3_3 \ 6_1 \\ \text{TTF} \end{array} \right\} \text{All Equivalent}$$

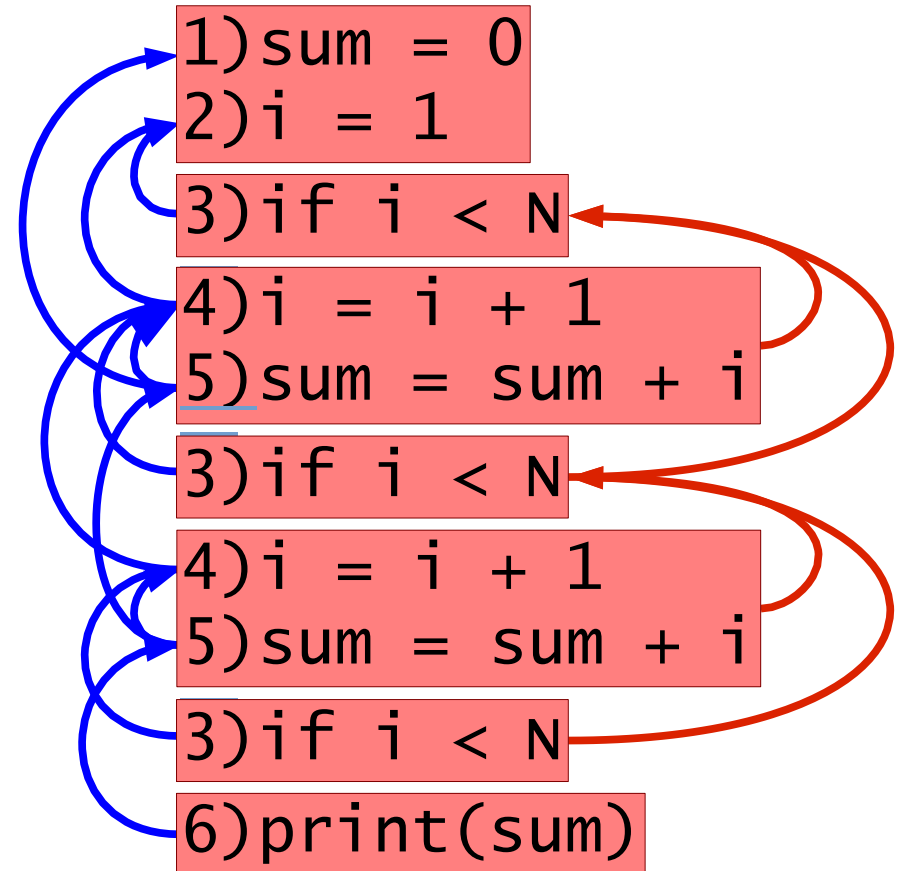
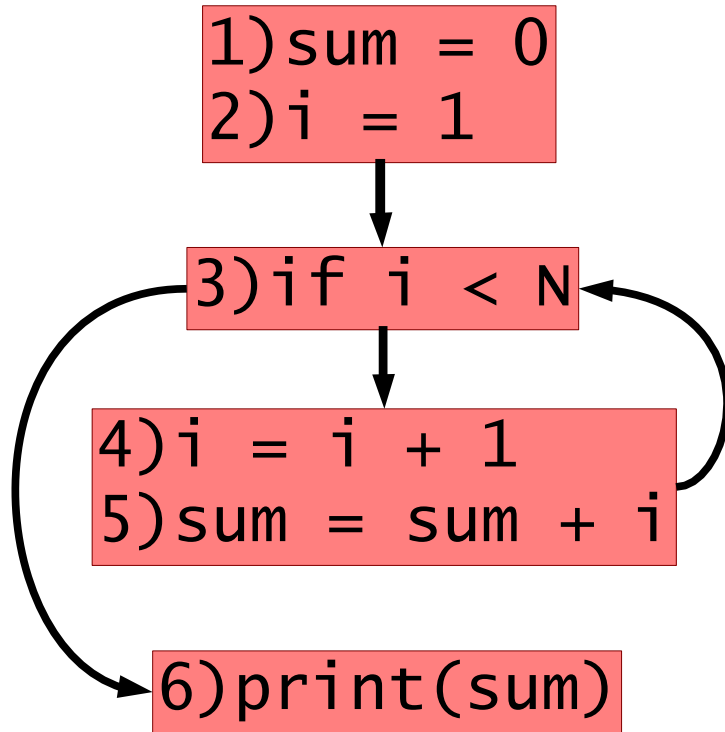
Dynamic Dependence Graph



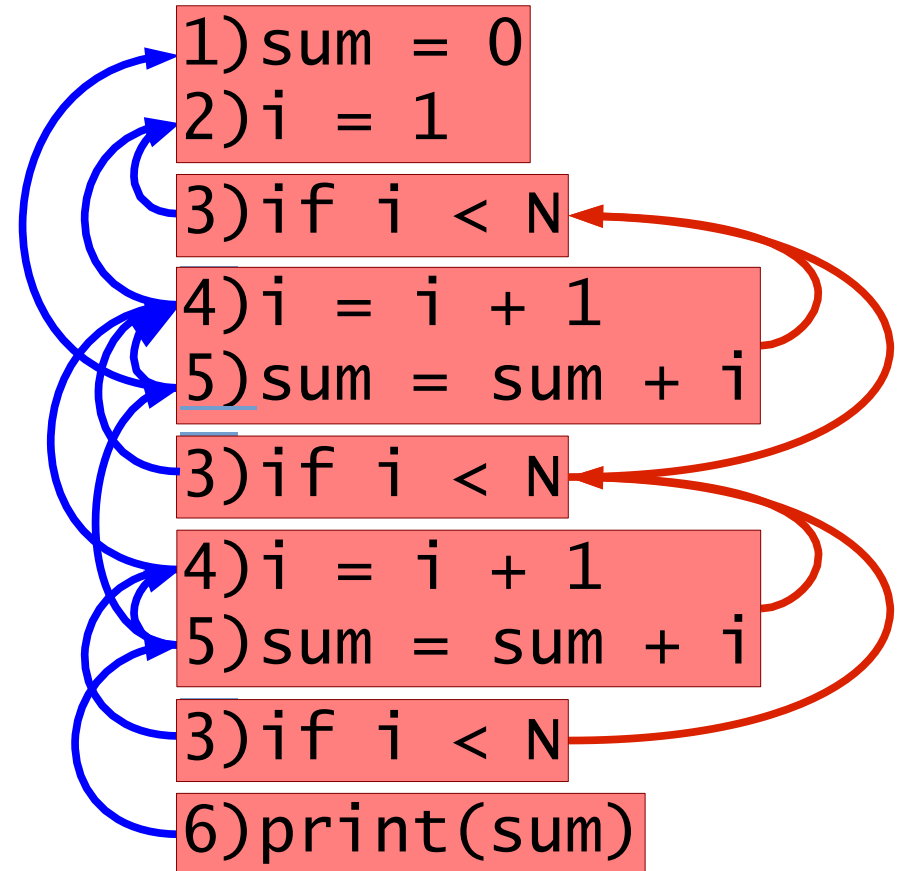
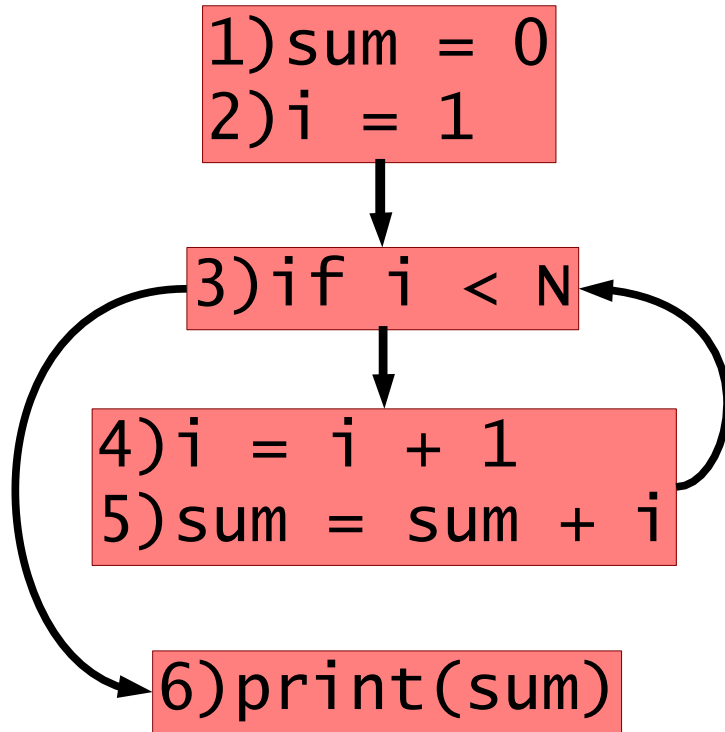
Dynamic Dependence Graph



Dynamic Dependence Graph

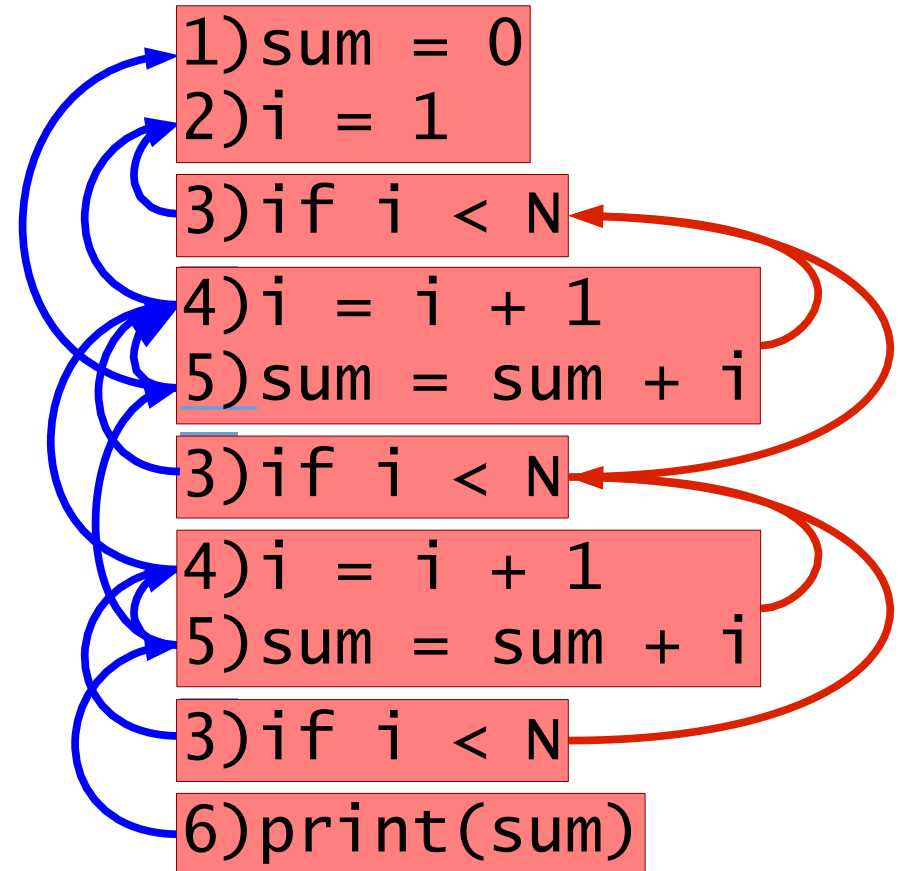
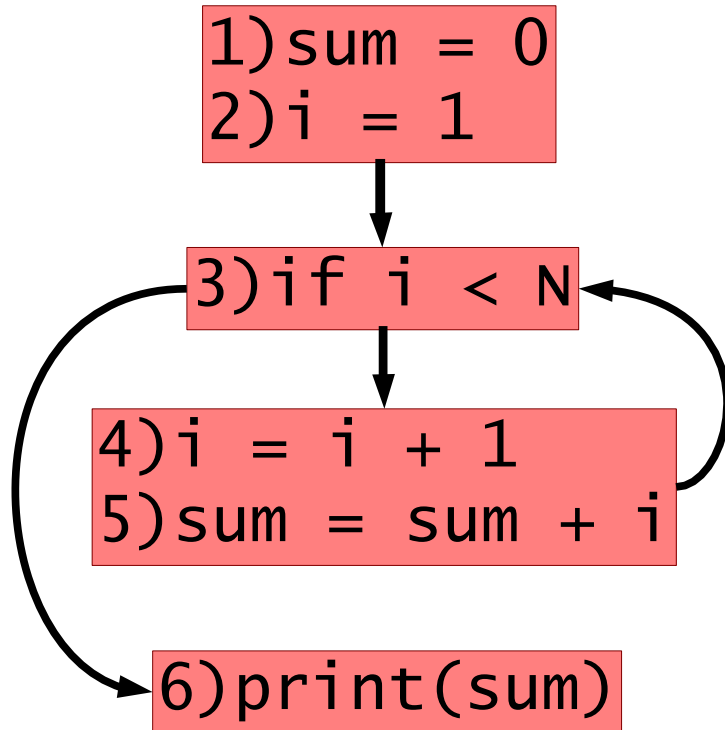


Dynamic Dependence Graph



Notably a bit difficult for a human to wade through.

Dynamic Dependence Graph



Notably a *bit* difficult for a human to wade through.

If only we could focus on the parts that interest us...

Execution Representations

Given these models, we can start to discuss interesting transformations and analysis of real programs.

Such as... slicing