CMPT 880
Special Topics:

Program Analysis & Reliability

Nick Sumner - Spring 2014

Much adapted from Xiangyu Zhang, Antony Hosking, Sorin Lerner,
Jonathan Aldrich, Sam Blackshear

Today

- Administrivia
- Dive right in!
  - Overview
  - Program Representations
  - Slicing
  - Basic Static Analysis
  - LLVM Basics

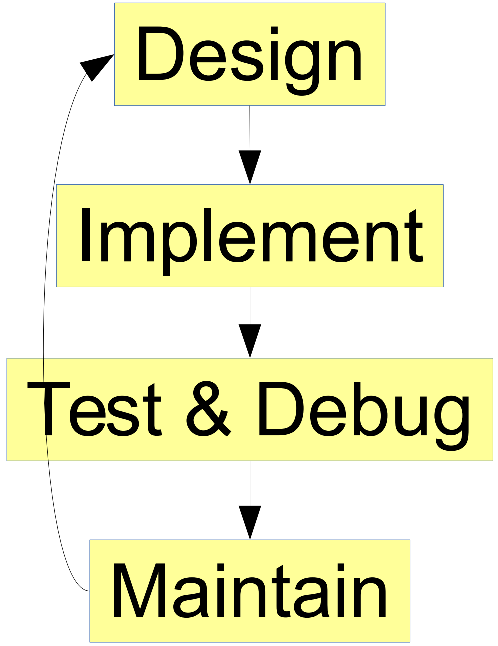
Time permitting

Course Website

- www.cs.sfu.ca/~wsumner/teaching/880-13/
  - Schedule
  - Policies
  - Assignments
  - Paper Suggestions

Why are you here?

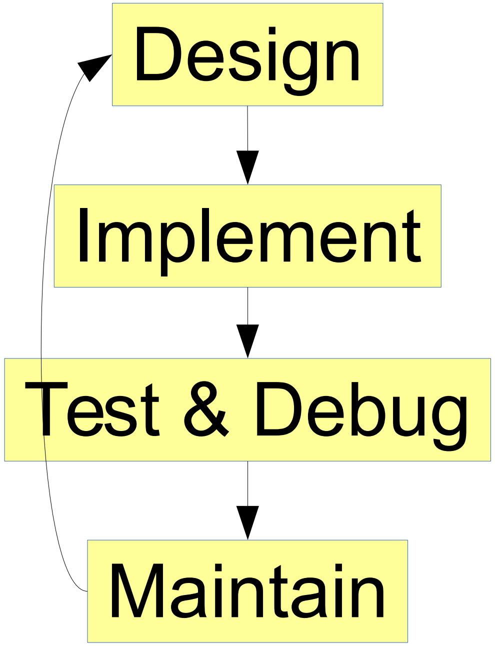
- Programs are big, complex, and difficult to reason about.



Why are you here?

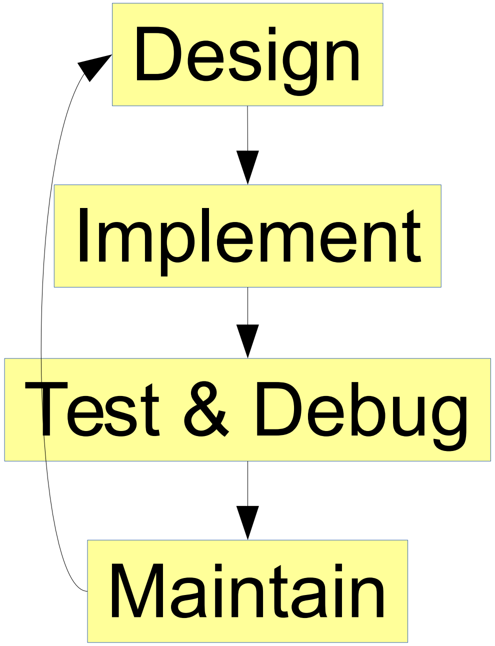
- Programs are big, complex, and difficult to reason about.

Are there more efficient designs?



Why are you here?

- Programs are big, complex, and difficult to reason about.

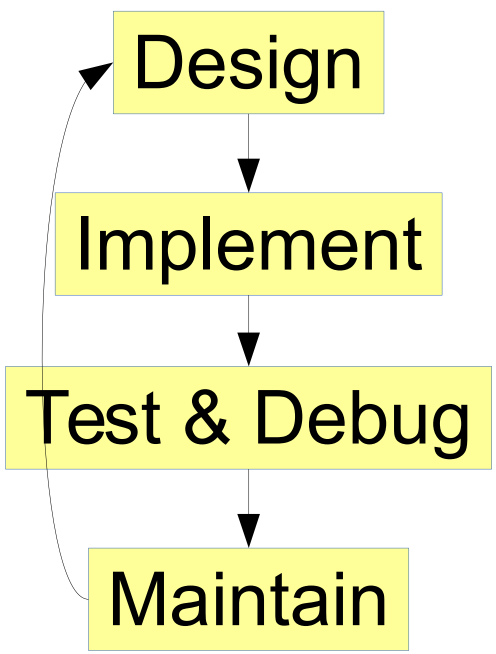


Are there more efficient designs?

What is the cause of a bug?

Why are you here?

- Programs are big, complex, and difficult to reason about.



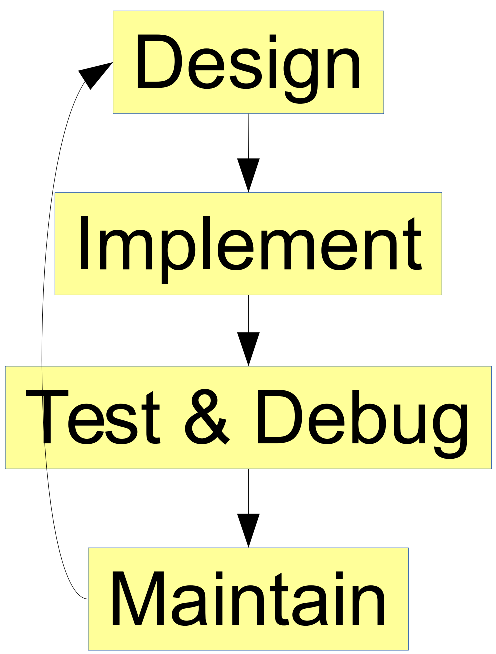
Are there more efficient designs?

What is the cause of a bug?

How do I find new bugs?

Why are you here?

- Programs are big, complex, and difficult to reason about.



Are there more efficient designs?

What is the cause of a bug?

How do I find new bugs?

How do I find security vulnerabilities?
Can I protect against them?

Why are you here?

- Programs are big, complex, and difficult to reason about.
  - Billions in lost profits and savings
  - Human casualties
  - Very tired grad students

Why are you here?

- Programs are big, complex, and difficult to reason about.
  - Billions in lost profits and savings
  - Human casualties
  - Very tired grad students

People are bad at tedious, subtle tasks, but computers are great at them!

Goal

- Learn how difficult tasks in development can be pushed onto computers.

Goal

- Learn how difficult tasks in development can be pushed onto computers.
  - Survey of *program analysis* techniques & papers

Goal

- Learn how difficult tasks in development can be pushed onto computers.
  - Survey of *program analysis* techniques & papers
    - Profiling

(Speed, Potential Concurrency, Memory, ...)

Goal

- Learn how difficult tasks in development can be pushed onto computers.
  - Survey of *program analysis* techniques & papers
    - Profiling
    - Testing

More effective tests. Bridge testing & verification

Goal

- Learn how difficult tasks in development can be pushed onto computers.
  - Survey of *program analysis* techniques & papers
    - Profiling
    - Testing
    - Debugging

      Explaining or locating the causes of bugs

Goal

- Learn how difficult tasks in development can be pushed onto computers.
  - Survey of *program analysis* techniques & papers
    - Profiling
    - Testing
    - Debugging
    - Concurrency

How to explain race conditions?

Atomicity violations?

How to find 'Heisenbugs'?

Goal

- Learn how difficult tasks in development can be pushed onto computers.
  - Survey of *program analysis* techniques & papers
    - Profiling
    - Testing
    - Debugging
    - Concurrency
    - Security

How to find vulnerabilities before attackers.

(...or as attackers)

Structure

- First few weeks (2-3) are review & background
  - I present.
  - You think about papers you'd like to present

Structure

- First few weeks (2-3) are review & background
  - I present.
  - You think about papers you'd like to present
- Reading foundational & new papers
  - 2 student presentations & paper discussions per week
  - Brief critique on weeks you don't present

Structure

- First few weeks (2-3) are review & background
  - I present.
  - You think about papers you'd like to present
- Reading foundational & new papers
  - 2 student presentations & paper discussions per week
  - Brief critique on weeks you don't present
- 2 small projects to introduce LLVM
- Course projects presented at end.

Presentations

- Guidelines on website
- 2 Goals
  - Help reinforce the material for the class
  - Lead an interesting discussion to examine the trade offs of each technique. (I'll be helping.)

Presentations

- Guidelines on website
- 2 Goals
  - Help reinforce the material for the class
  - Lead an interesting discussion to examine the trade offs of each technique. (I'll be helping.)
- Show how the technique behaves in the best case
- Show or lead discussion on where it might behave poorly

Critiques

- Guidelines on website
- 1-2 page response to 1 paper each week that you do not present.
- Primarily meant to prepare you for the discussion on the paper that week.

Term Projects

- Groups of 1 or 2.
- 1 page proposals due March 3.
- Find something that interests (or irritates) you and go after it!
  - Maybe look at how these techniques can help your existing research

What Could We Look At?

- Surviving Failures

What Could We Look At?

- Surviving Failures
- Plagiarism Detection

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation
- Automated Regression Testing

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation
- Automated Regression Testing
- Program Guided Fuzz Testing

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation
- Automated Regression Testing
- Program Guided Fuzz Testing
- Data Race Explanation

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation
- Automated Regression Testing
- Program Guided Fuzz Testing
- Data Race Explanation
- Battery Use Profiling

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation
- Automated Regression Testing
- Program Guided Fuzz Testing
- Data Race Explanation
- Battery Use Profiling
- Mobile Privilege Protection/Reduction

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identifying Information Leaks
- Automated Debugging
- Automated Test Generation
- Automated Regression Testing
- Program Guided Fuzz Testing
- Data Race Explanation
- Battery Use Profiling
- Mobile Privilege Protection/Reduction
- Reproducing Remote Bugs
- ...

What Could We Look At?

- Surviving Failures
- Plagiarism Detection
- Malware Detection
- Identif
- Autom
- Autom
- Automated Regression Testing
- Program Guided Fuzz Testing
- Data Race Explanation

- Battery Use Profiling
- Mobile Privilege Protection/Reduction
- ng Remote

- I have planned out a survey, but we can customize it for interest
- The last few weeks will be chosen by your interests already

Program Representations

Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze

Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze
- Difficult models:
  - Compiled binaries
    - Difficult to even separate code from data in general
  - Source code
    - Very language specific

Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze
- Difficult models:
  - Compiled binaries
    - Difficult to even separate code from data in general
  - Source code
    - Very language specific
- Need something better

Program Representation

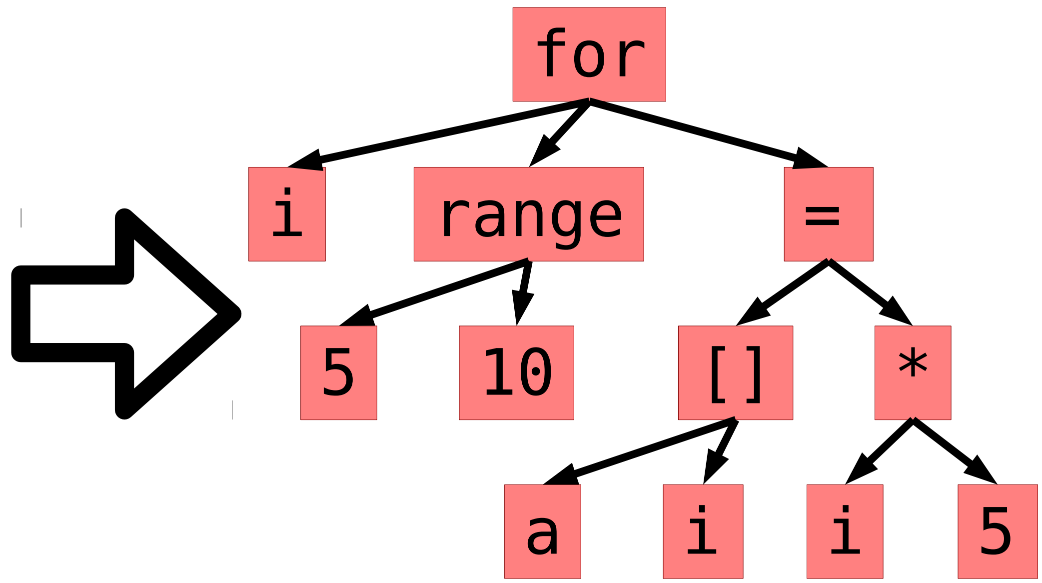
Core Representations for Analysis:

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

1) Abstract Syntax Trees

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

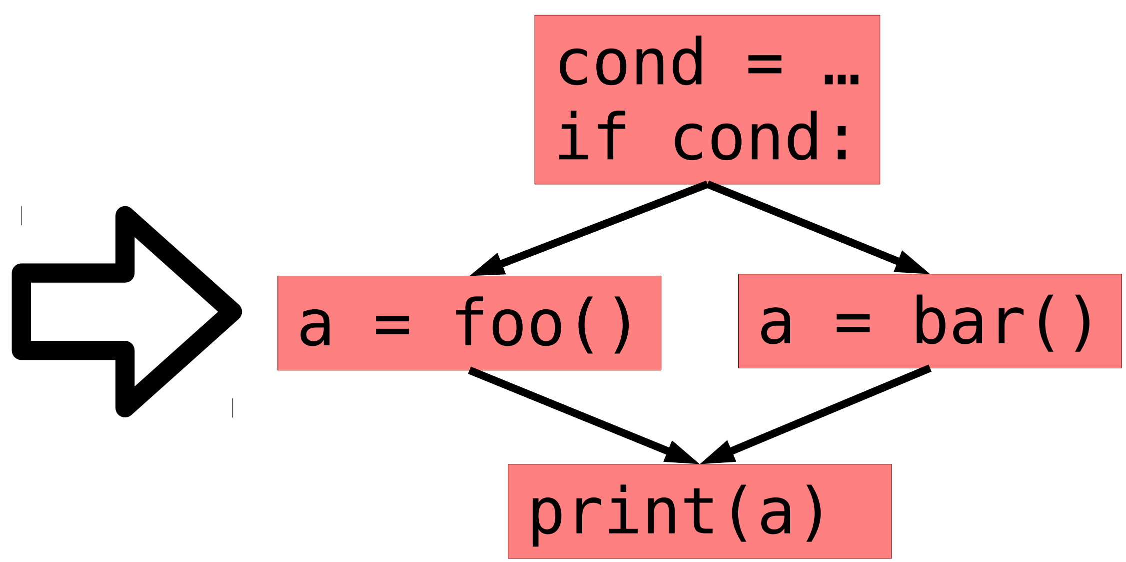
```
for i in range(5,10):
    a[i] = i * 5
```



2) Control Flow Graphs

- Express the possible decisions and possible paths through a program

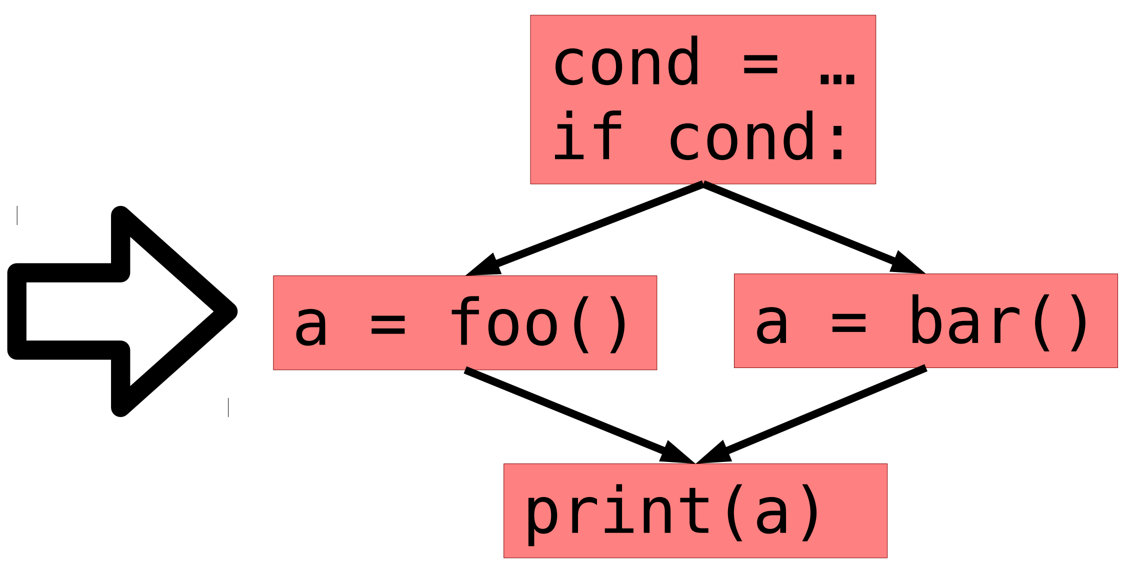
```
cond = input()
if cond:
  a = foo()
else:
  a = bar()
print(a)
```



2) Control Flow Graphs

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

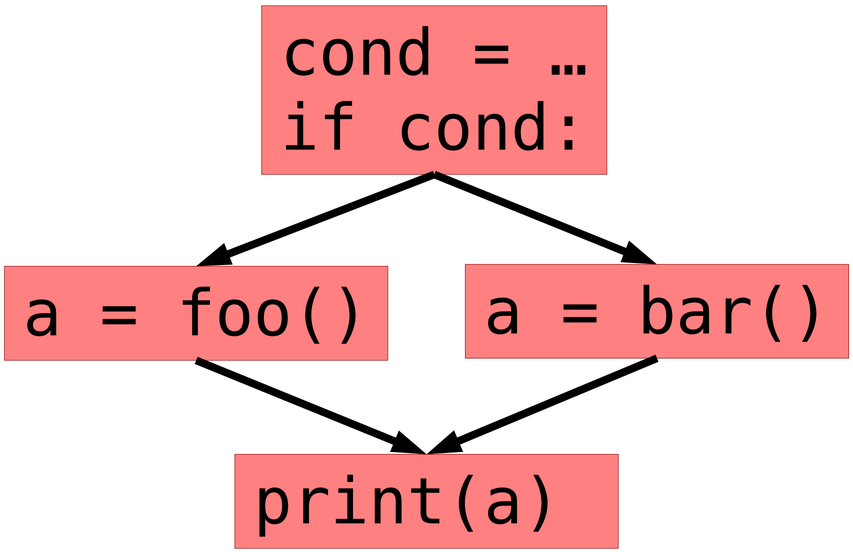
```
cond = input()
if cond:
  a = foo()
else:
  a = bar()
print(a)
```



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

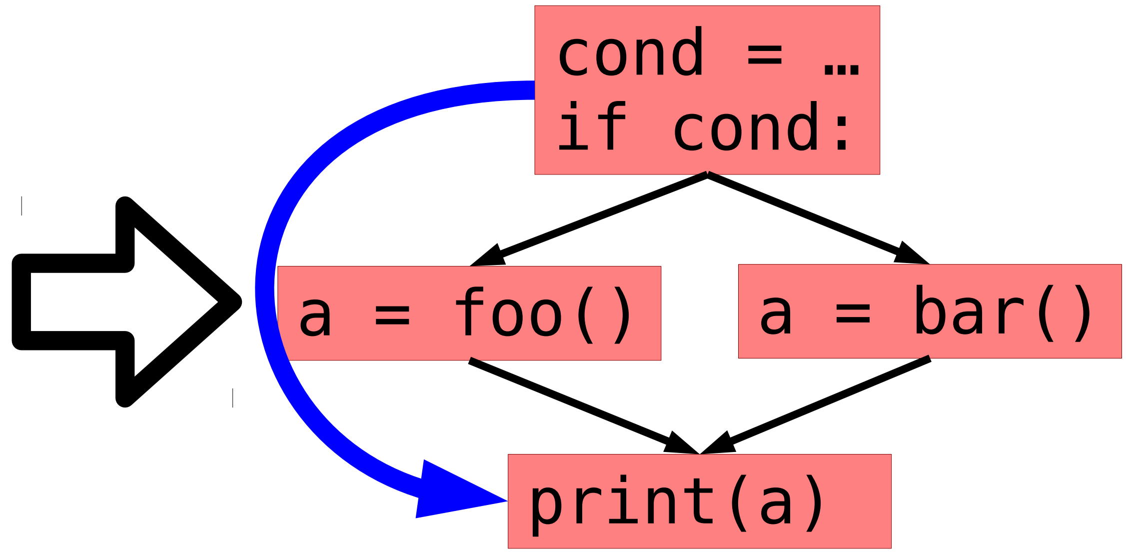
```
cond = input()
if cond:
  a = foo()
else:
  a = bar()
print(a)
```



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)
```

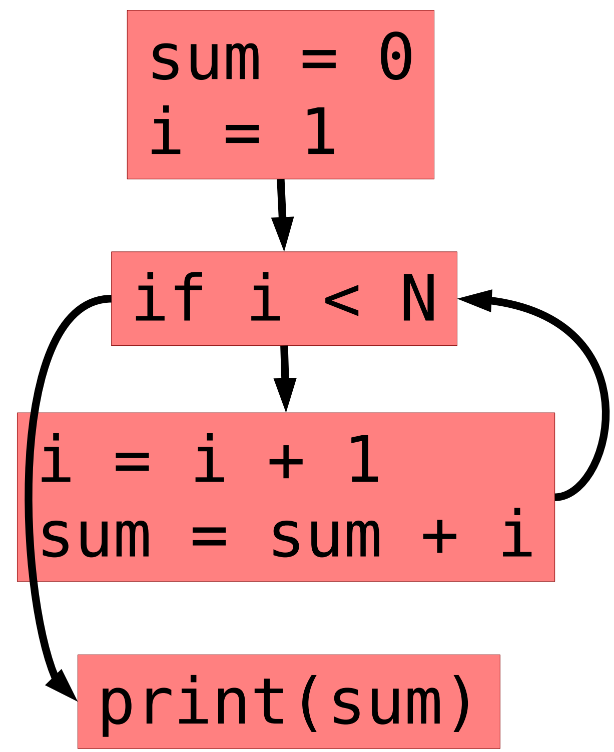


2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

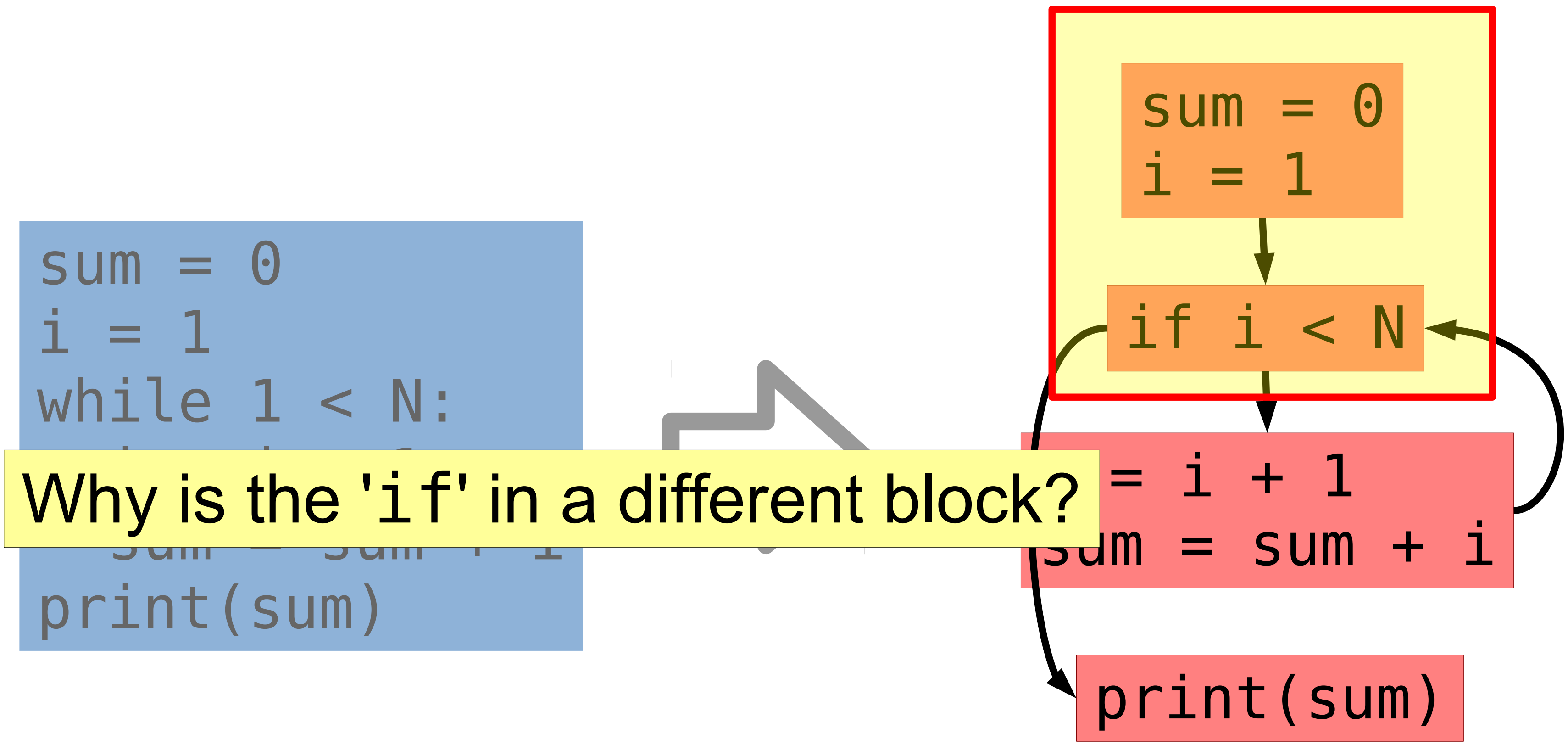
The 'while' is gone

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



2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away



3)Program Dependence Graph (PDG)

- Instruction X depends on Y if Y *can influence* X
  - Nodes are instructions
  - An edge Y→X shows that Y influences X

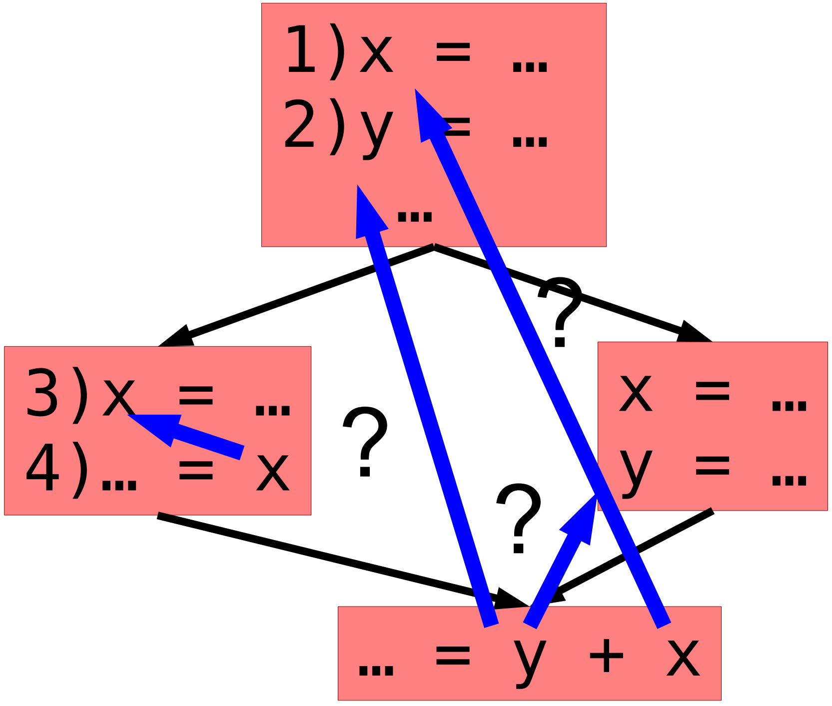
3)Program Dependence Graph (PDG)

- Instruction X depends on Y if Y *can influence* X
  - Nodes are instructions
  - An edge Y→X shows that Y influences X
- 2 main types of influence:
  - Data dependence
  - Control dependence

Data Dependence

X data depends on Y if

- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X



Control Dependence

Preliminary: X dominates Y if

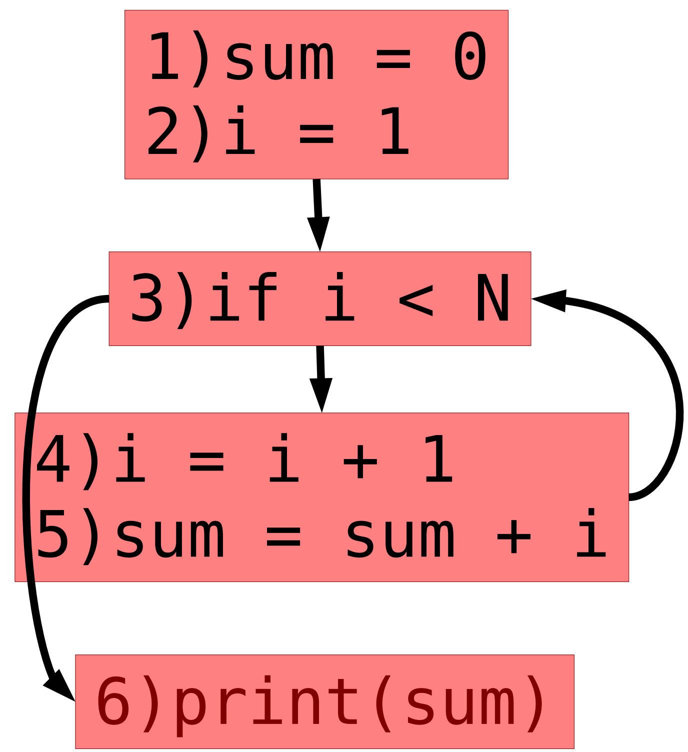
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance

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 1 < N:
4)  i = i + 1
5)  sum = sum + i
6)print(sum)
```



DOM(6)={1,2,3,6}   IDOM(6)=3

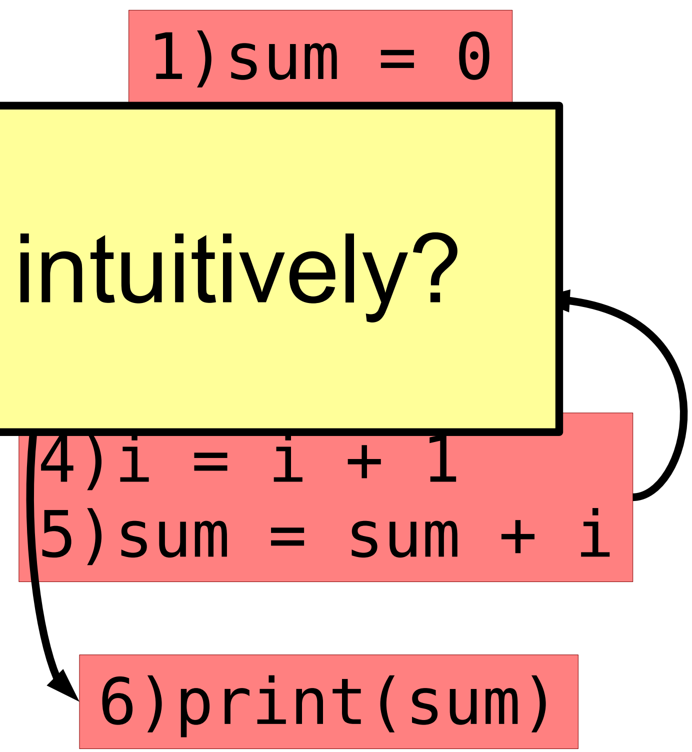
Control Dependence

Preliminary: X dominates Y if

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

What does this mean intuitively?

```
1)
2)
3)
4)   i = i + 1
5)   sum = sum + i
6)print(sum)
```



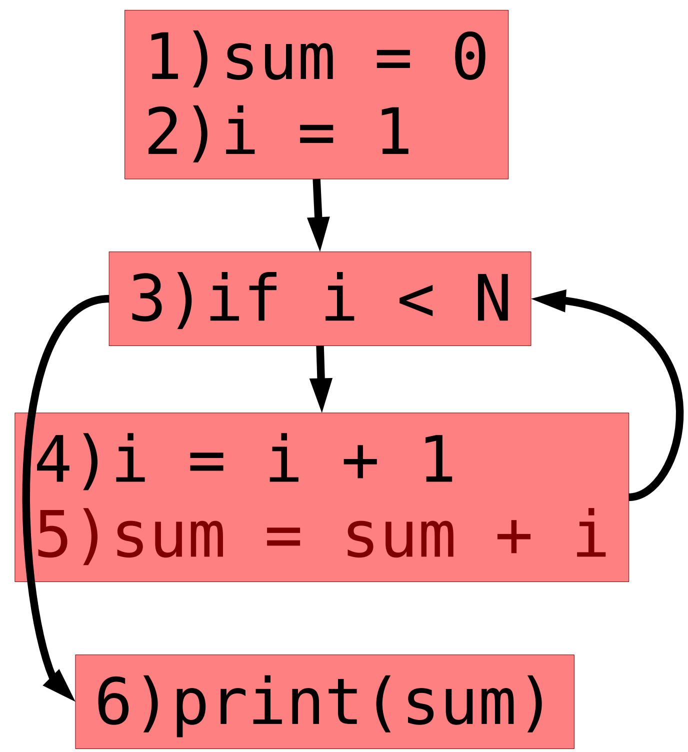
DOM(6)={1,2,3,6}   IDOM(6)=3

Control Dependence

Preliminary: X post dominates Y if

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

```
1)sum = 0
2)i = 1
3)while 1 < 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 iff

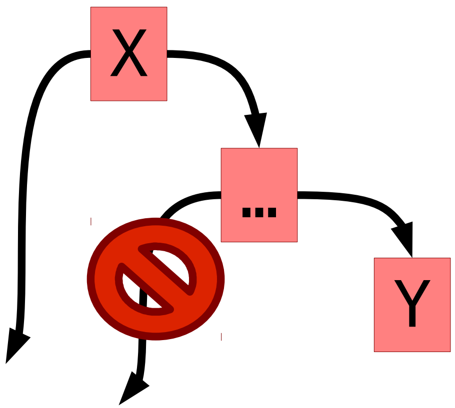
- Definition 1:

  X directly decides whether Y executes

Control Dependence (Finally)

Y is control dependent on X iff

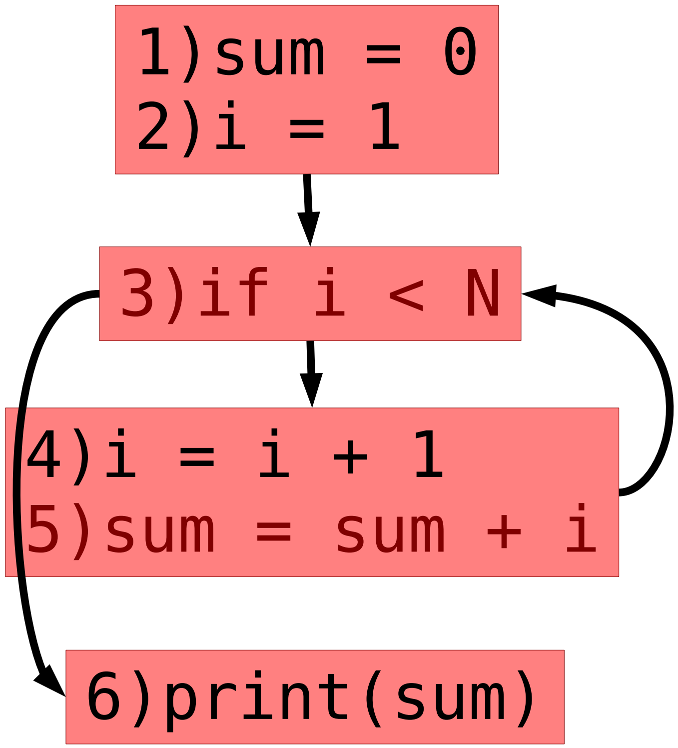
- Definition 1:
  X directly decides whether Y executes
- Definition 2:
  - There exists a path from X to Y s.t. Y post dominates every node between X and Y.
  - Y does not strictly post dominate X



Control Dependence

- 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 1 < N:
4)  i = i + 1
5)  sum = sum + i
6)print(sum)
```



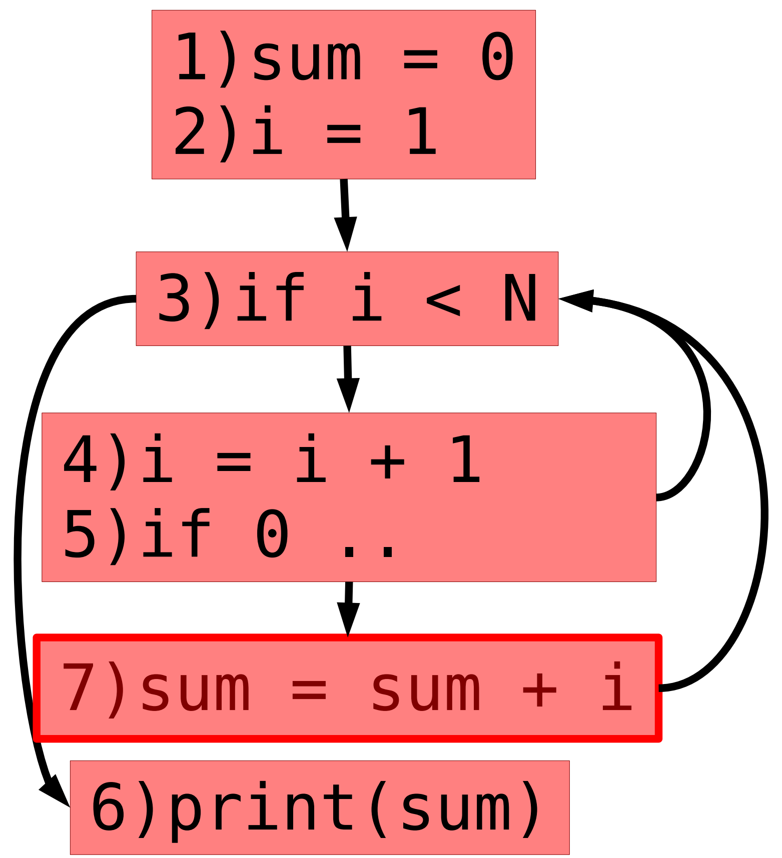
What is CD(5)? CD(3)

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 1 < N:
4)  i = i + 1
5)  if 0 == i%2:
6)    continue
7)  sum = sum + i
8)print(sum)
```

What is CD(7)?



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

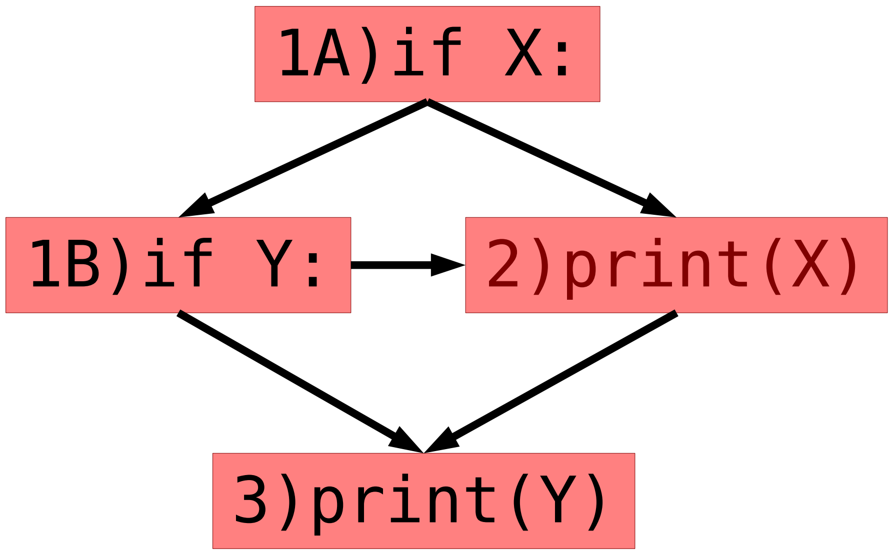
```
1)if X or Y:
2)  print(X)
3)print(Y)
```

What is CD(2)?

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)if X or Y:
2)  print(X)
3)print(Y)
```



What is CD(2)?

3)Program Dependence Graph(PDG)

The PDG is the combination of

- The control dependence graph
- The data dependence graph

3)Program Dependence Graph(PDG)

The PDG is the combination of

- The control dependence graph
- The data dependence graph

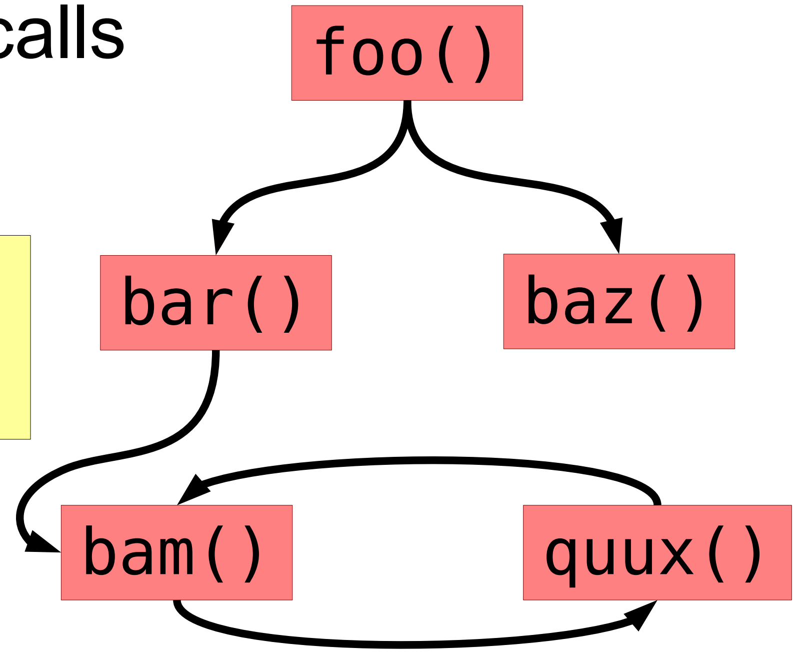
Recall: Edges identify *potential influence*

- 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

How should we handle function pointers?



5) Points-to Graphs

Aliasing:

- Multiple variables may denote the same memory location

Multiple Targets:

- One variable may potentially denote several different targets in memory.

5) Points-to Graphs

Aliasing:

- Multiple variables may denote the same memory location

Multiple Targets:

- One variable may potentially denote several different targets in memory.

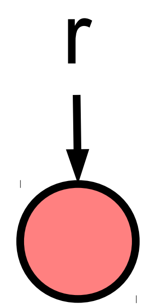
```
x.lock()
…
y.unlock()
```

```
x = password
…
broadcast(y)
```

5) Points-to Graphs

- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

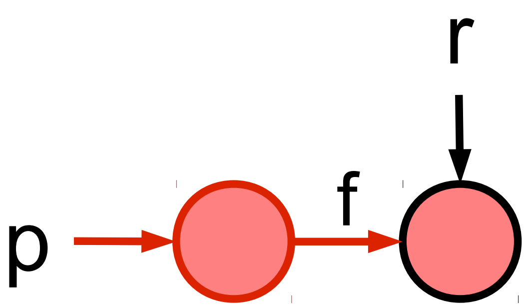
```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)   q = p
6) r.f = t
```



5) Points-to Graphs

- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

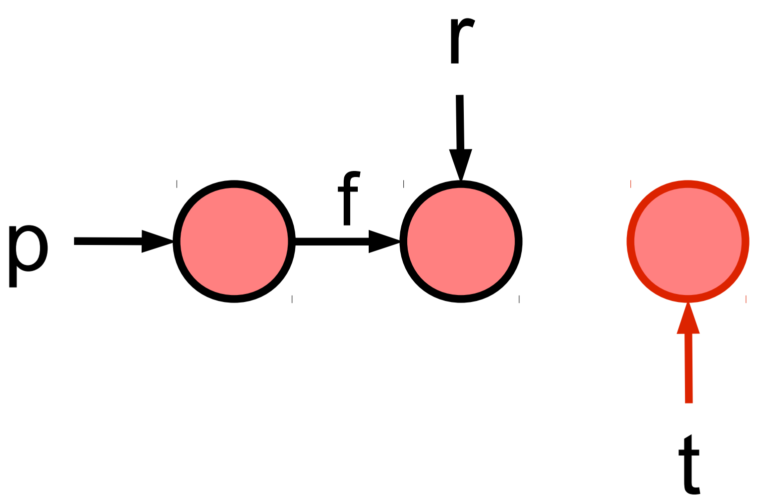
```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)   q = p
6) r.f = t
```



5) Points-to Graphs

- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

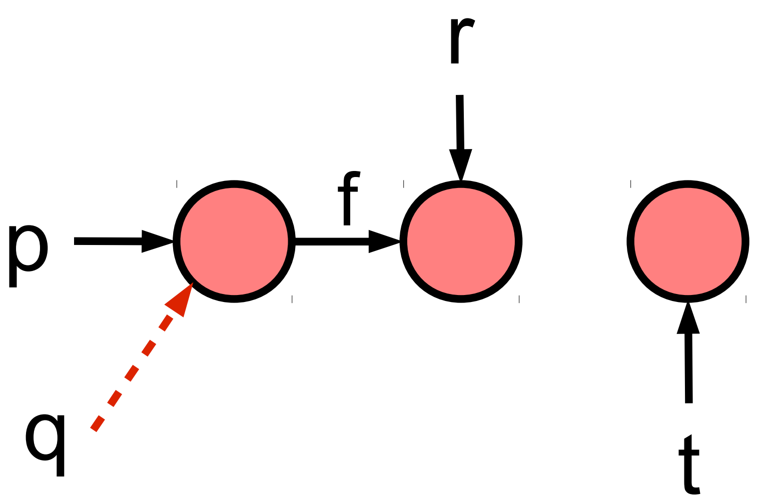
```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)   q = p
6) r.f = t
```



5) Points-to Graphs

- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

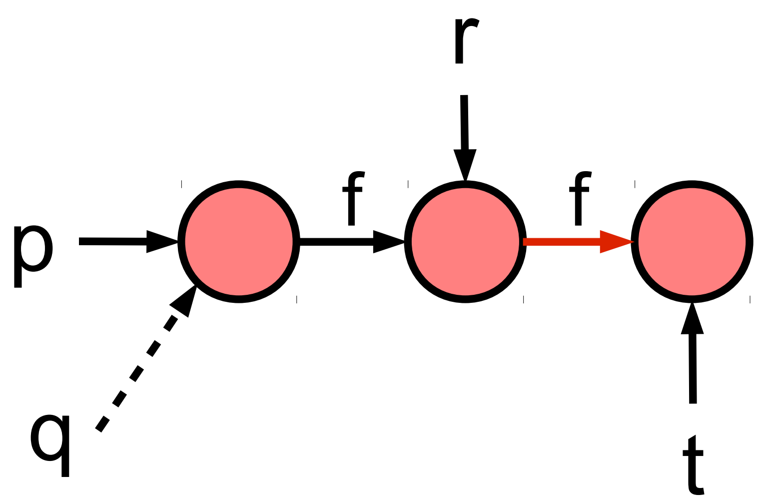
```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)    q = p
6) r.f = t
```



5) Points-to Graphs

- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

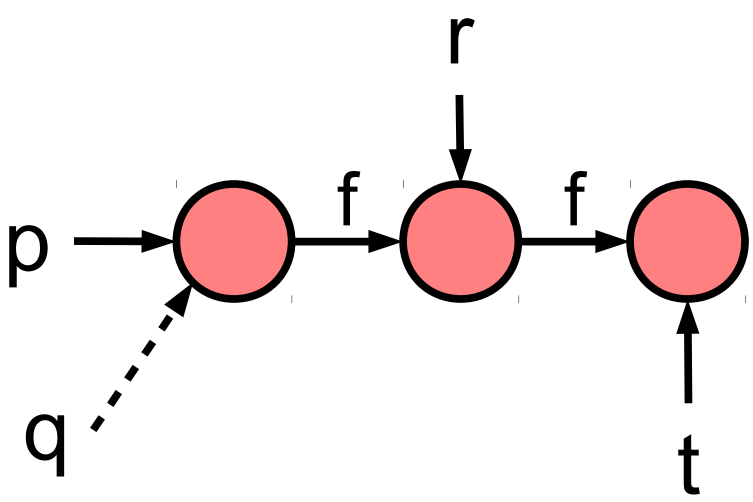
```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)   q = p
6) r.f = t
```



5) Points-to Graphs

- The relation (p,x) where p MAY/MUST point to x
  - Both MAY and MUST information can be useful

```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)   q = p
6) r.f = t
```



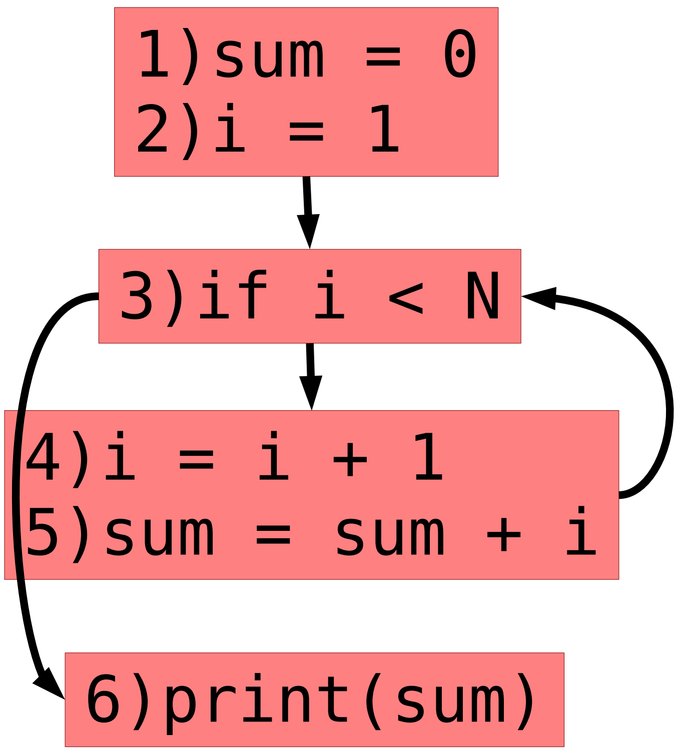
p.f.f MUST ALIAS t
q MAY ALIAS p

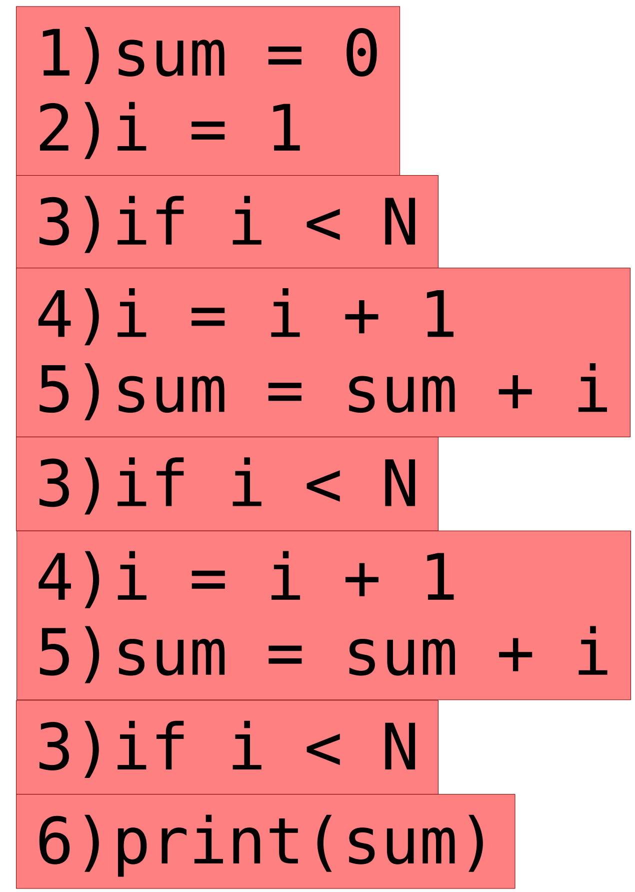
Execution Representations

- Program Representations are *static*
  - All possible program behaviors at once
  - Usually projected onto the CFG

- Execution Representations are *dynamic*
  - Only the behavior of a single real execution
  - Multiple instances of an instruction occur multiple times

Control Flow Trace





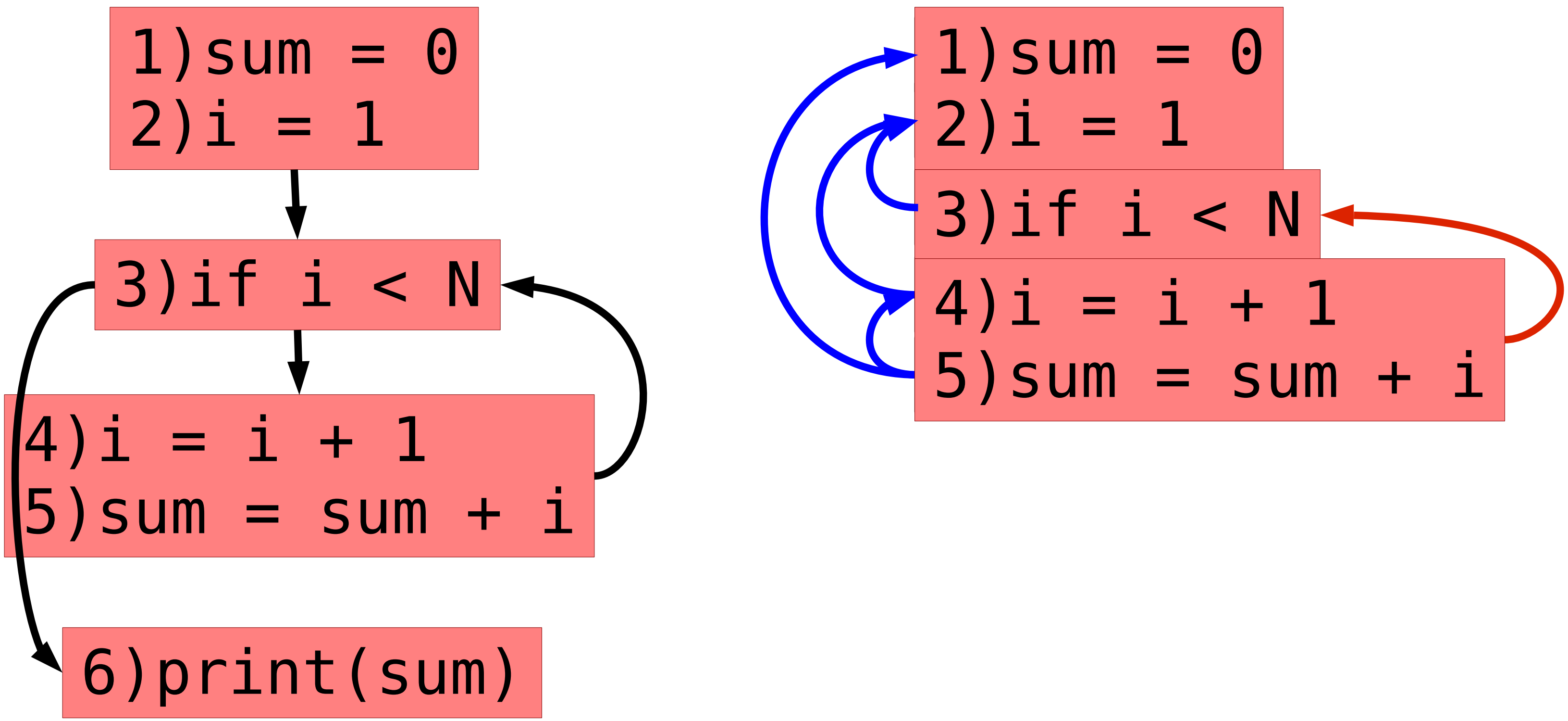
$1_1\ 2_1\ 3_1\ 4_1\ 5_1\ 3_2\ 4_2\ 5_2\ 3_3\ 6_1$

$1_1\ 3_1\ 4_1\ 3_2\ 4_2\ 3_3\ 6_1$

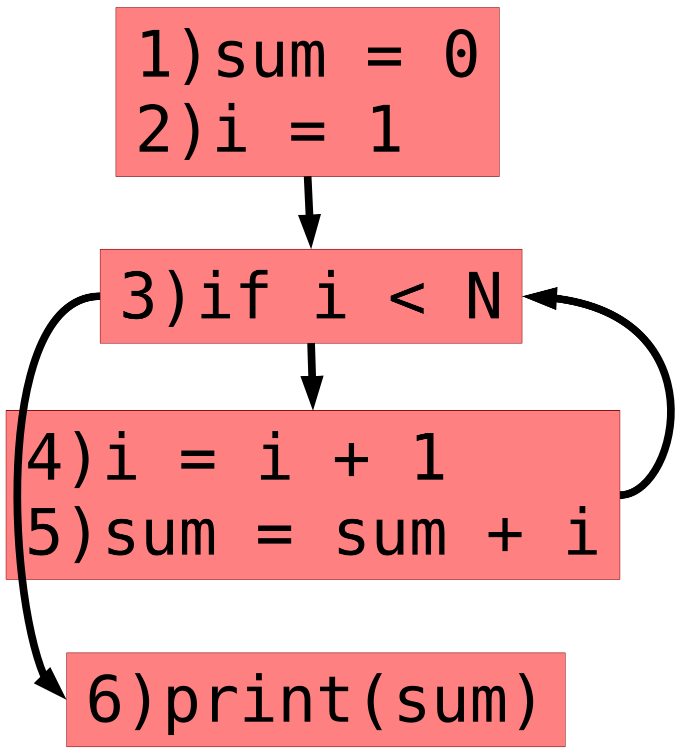
TTF

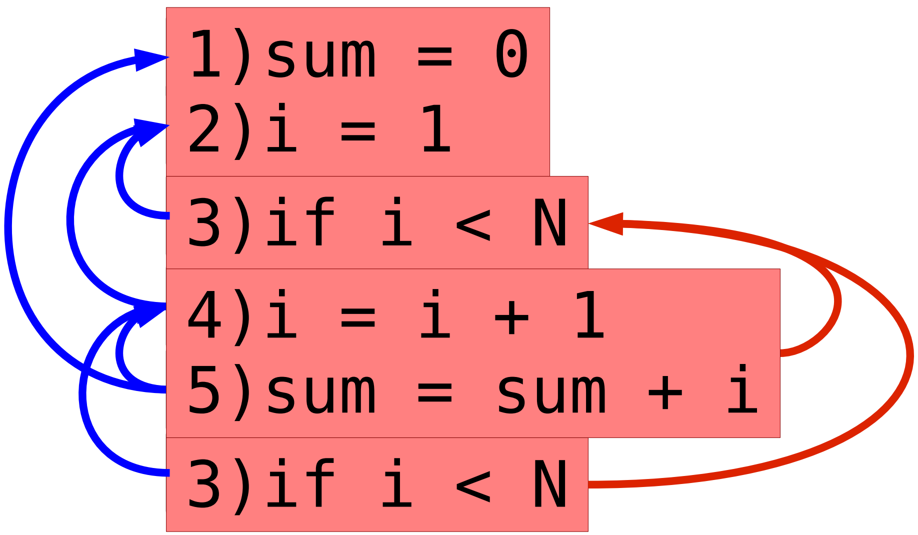
} All Equivalent

Dynamic Dependence Graph

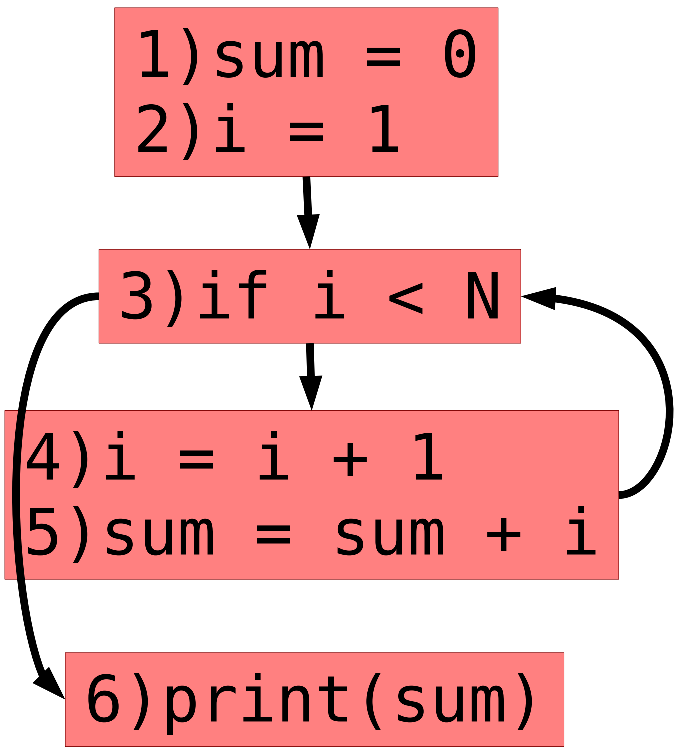


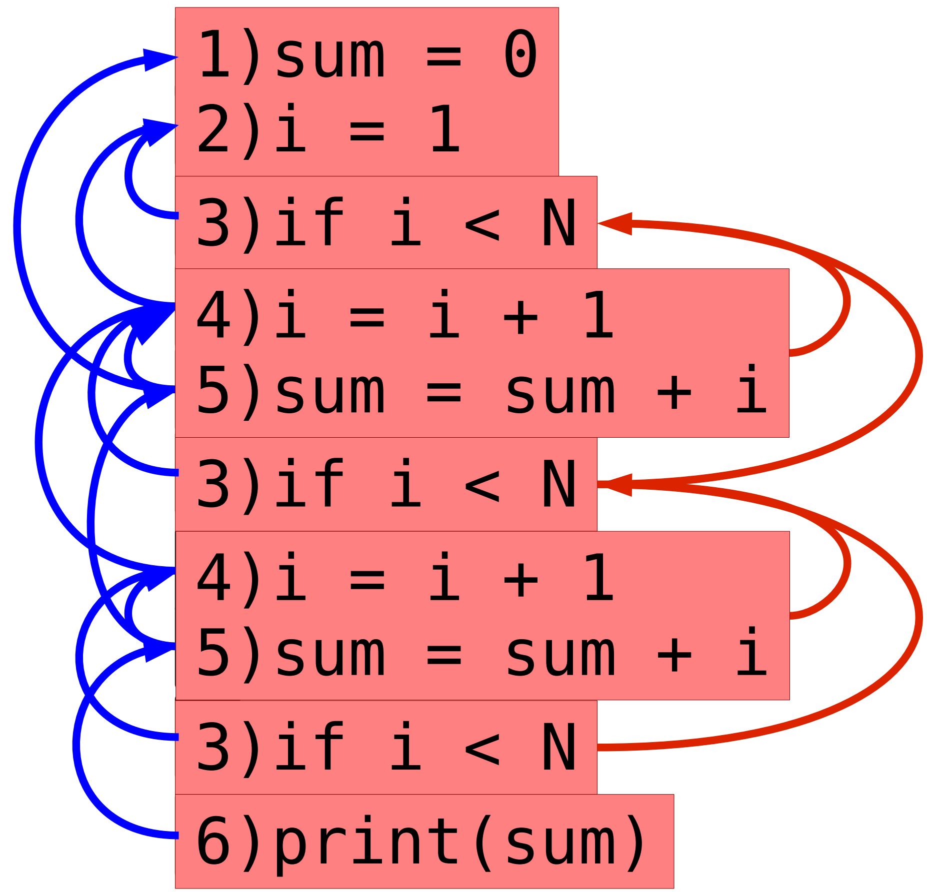
Dynamic Dependence Graph



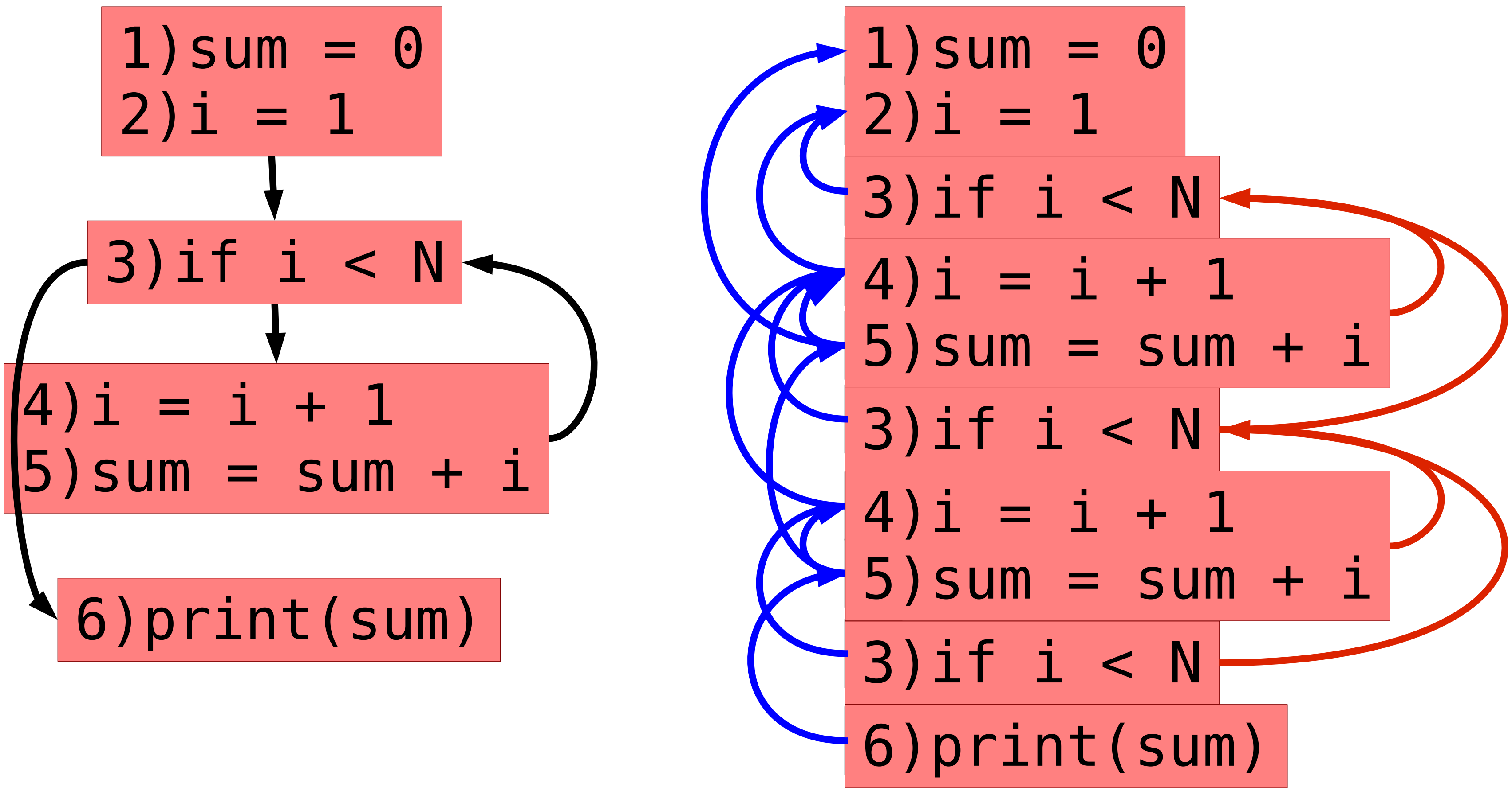


Dynamic Dependence Graph





Dynamic Dependence Graph



Notably a bit difficult for a human to wade through.

Program Representations

Given these models, we can start to discuss interesting transformations and analyses on real programs.

Such as...

Slicing

Program Slicing

- The *slice* of a value v at a statement s is:
  - the set of statements involved in computing v's value at s. [Weiser 82]

Program Slicing

- The *slice* of a value v at a statement s is:
  - the set of statements involved in computing v's value at s. [Weiser 82]

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

Program Slicing

- The *slice* of a value v at a statement s is:
  - the set of statements involved in computing v's value at s. [Weiser 82]

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

Program Slicing

- The *slice* of a value v at a statement s is:
  - the set of statements involved in computing v's value at s. [Weiser 82]

- The statements that may influence v...
  - Data dependence
  - Control dependence
  - Compute using the PDG!

Program Slicing Uses

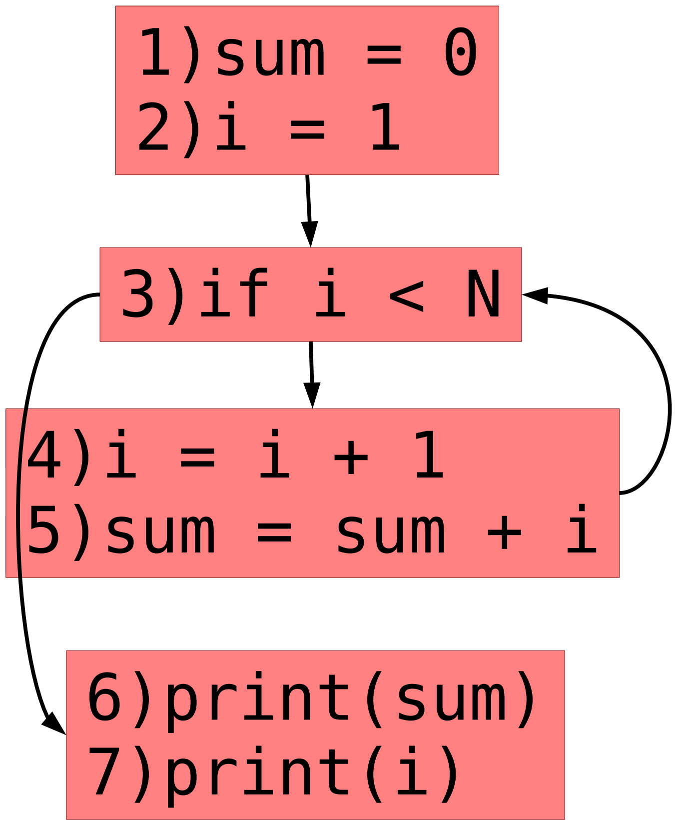
- Debugging
- Testing
- Reverse Engineering
- Optimization
- Design Profiling
- Malware analysis
- ...

How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward

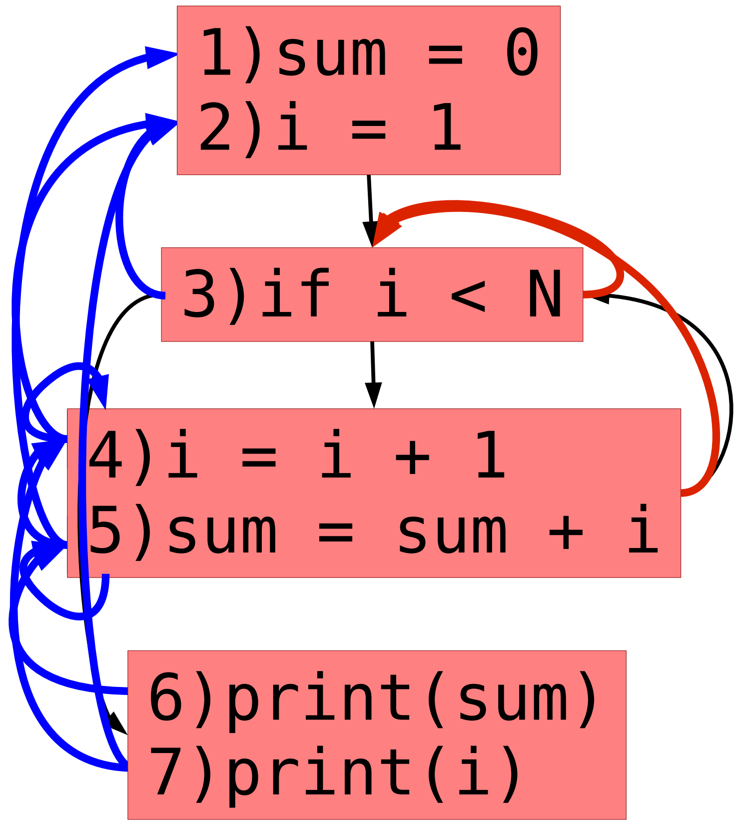
How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward



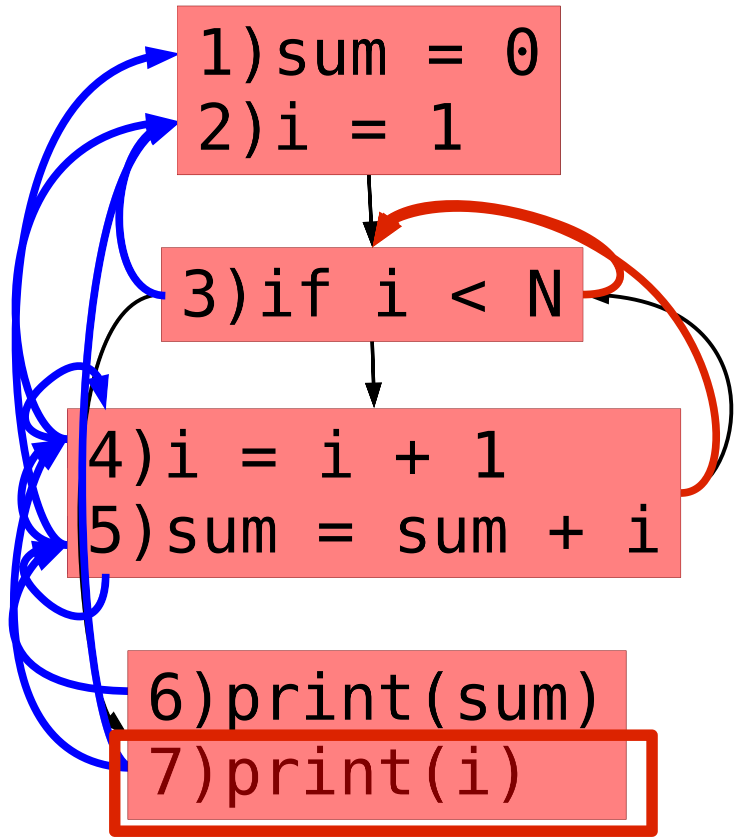
How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward



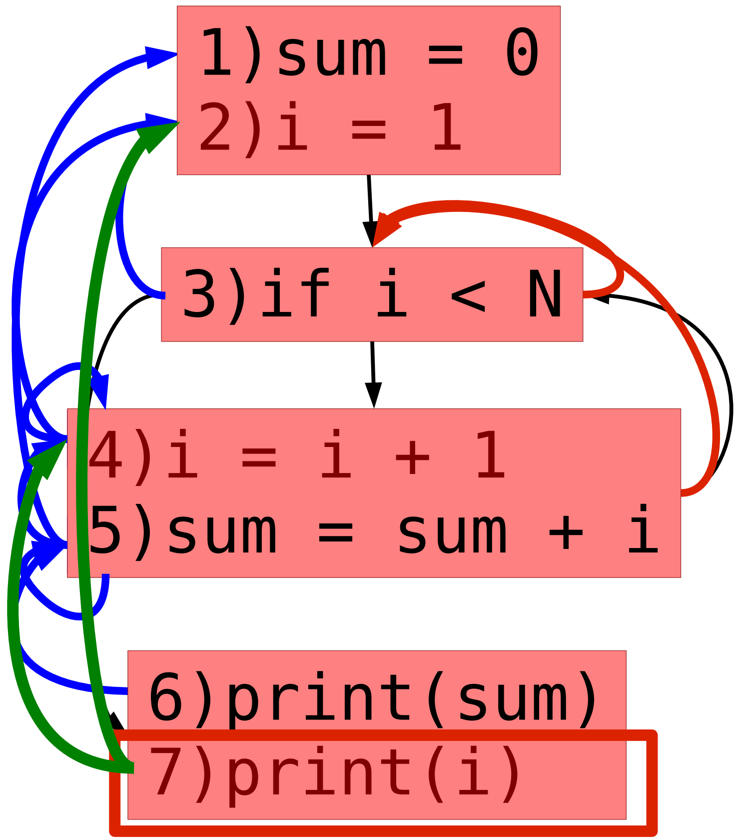
How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward



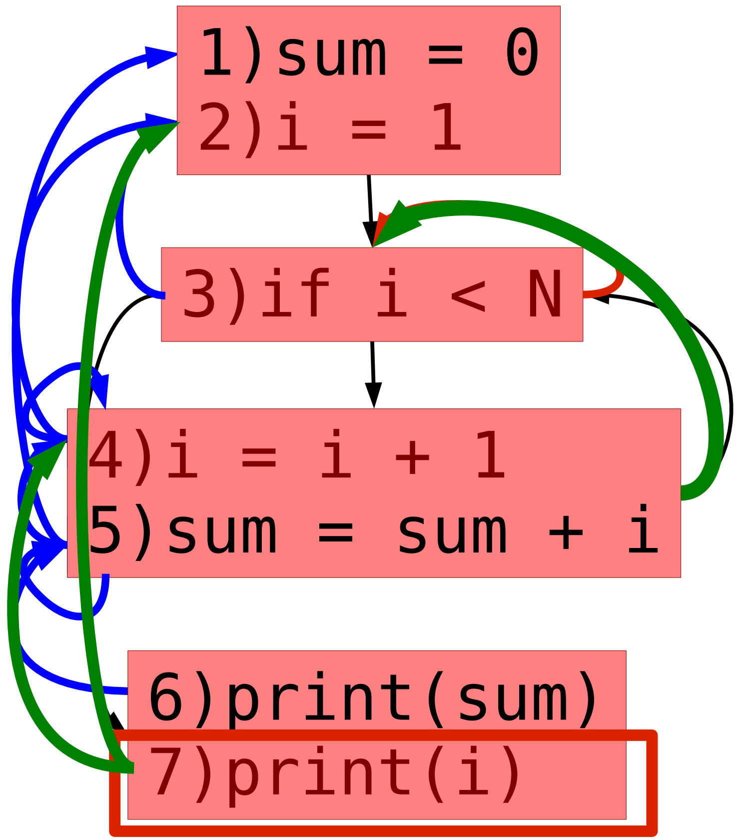
How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward



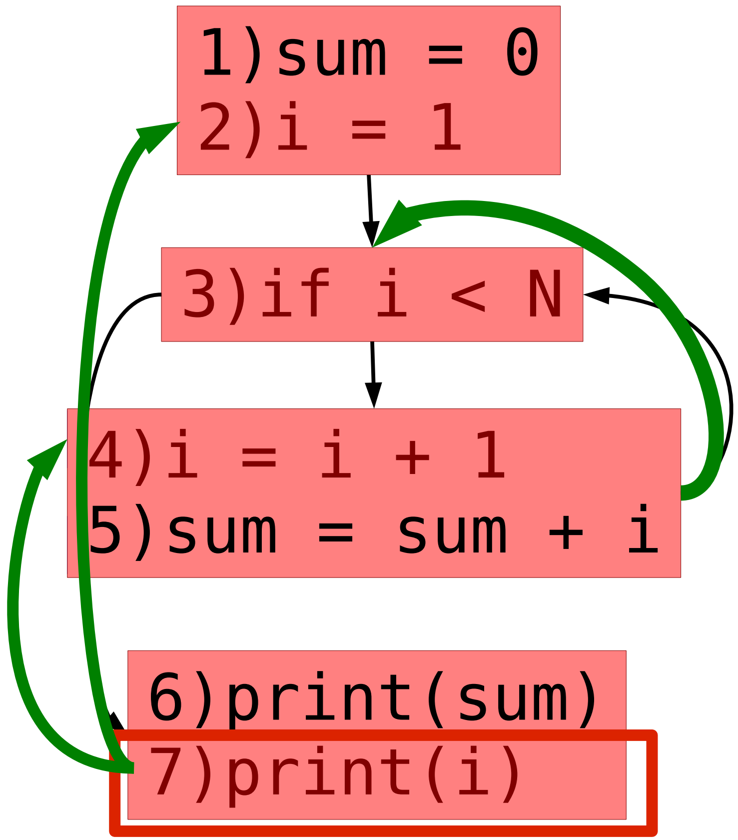
How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward



How to Slice?

- Transitive closure of edges in the PDG
  - Start from v and just follow edges backward



Very Configurable

- Static vs. Dynamic (PDG vs. DDG)
- Backward vs. Forward
- Executable vs. Nonexecutable

What do forward and backward *mean*?

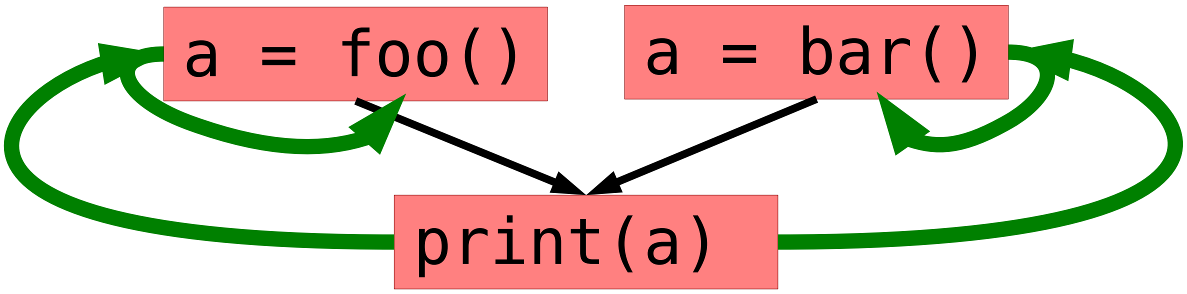
Why might a slice not be executable?

Strengths of Static Slicing

- Considers all possible executions
  - Necessary for conservative analyses
  - (“Might I leak secret information?”)
- Fast to compute
- Space efficient

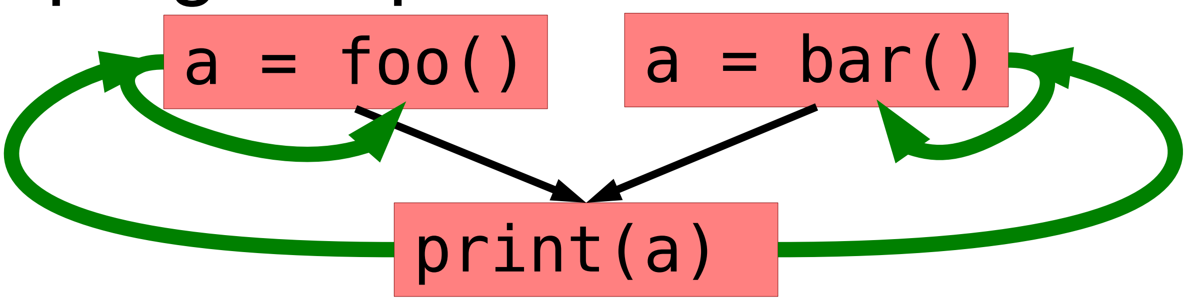
Issues with Static Slicing

- Multiple program paths

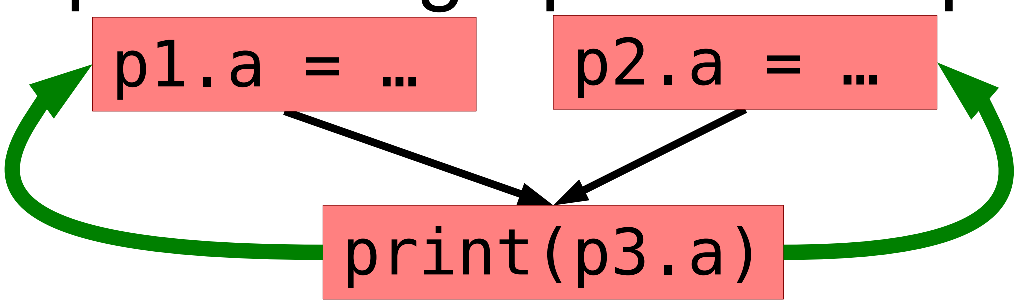


Issues with Static Slicing

- Multiple program paths

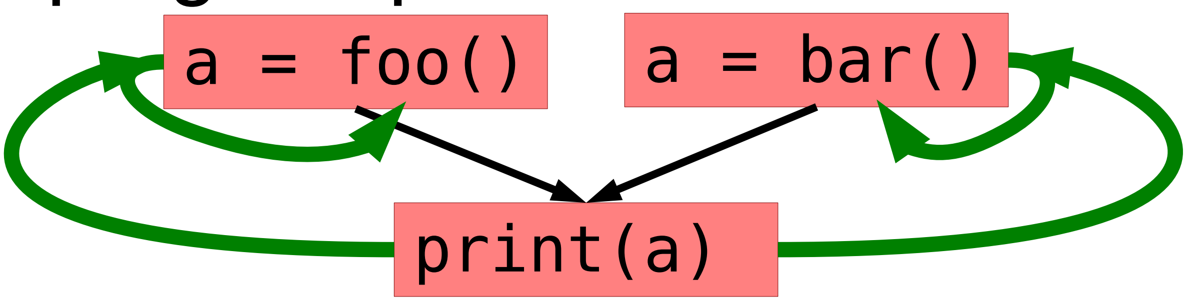


- Pointers – points-to graphs are imprecise

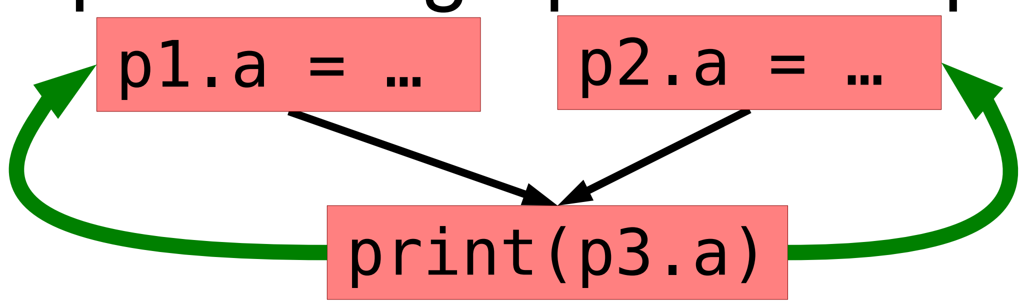


Issues with Static Slicing

- Multiple program paths



- Pointers – points-to graphs are imprecise



- Function pointers – must consider all possible call targets

Strengths of Dynamic Slicing

- Precisely considers a single execution (DDG)
  - "Did I ..."
- No imprecision from aliasing or multiple paths
  - Why?
- Cover fewer static program statements

Issues with Dynamic Slicing

- Capturing a trace and computing a DDG is expensive
  - (GB sized trace files)
- Slow to compute
  - Churn a great deal of memory
- Very many statement instances and dynamic dependences to examine
- Misses alternative histories
  - What would have happened if … ?

Coping with Scale

Both types of slicing benefit from techniques that *prune* or *focus* slices on just what is *interesting*

Coping with Scale

Both types of slicing benefit from techniques that *prune* or *focus* slices on just what is *interesting*

- *Thin Slicing*- Focus on propagating v, ignoring data structures [PLDI07]
- *Chopping*- Combine forward & backward info [ASE05]
- *Confidence Analysis*- Instructions used to compute correct values less likely to be buggy [PLDI06]
- *Guided Browsers*- Zoom in on demand [ICSE06]
- Much more...

Static Analysis

Static Analysis

Static analyses consider all possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?
  - Does my program terminate?

Static Analysis

Static analyses consider all possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?
  - Does my program terminate?

But wait? Isn't that impossible?

Static Analysis

Static analyses consider all possible behaviors of a program without running it.

- Look for a property of interest
  - Do I dereference NULL pointers?
  - Do I leak memory?
  - Do I violate a protocol specification?
  - Is this file open?
  - Does my program terminate?

But wait? Isn't that impossible?

- Only if answers must be perfect.

Static Analysis

Overapproximate or underapproximate the problem, and try to solve this simpler version.

Static Analysis

Overapproximate or underapproximate the problem, and try to solve this simpler version.

- Sound analyses
  - Overapproximate
  - Guaranteed to find violations of property
  - May raise false alarms

Static Analysis

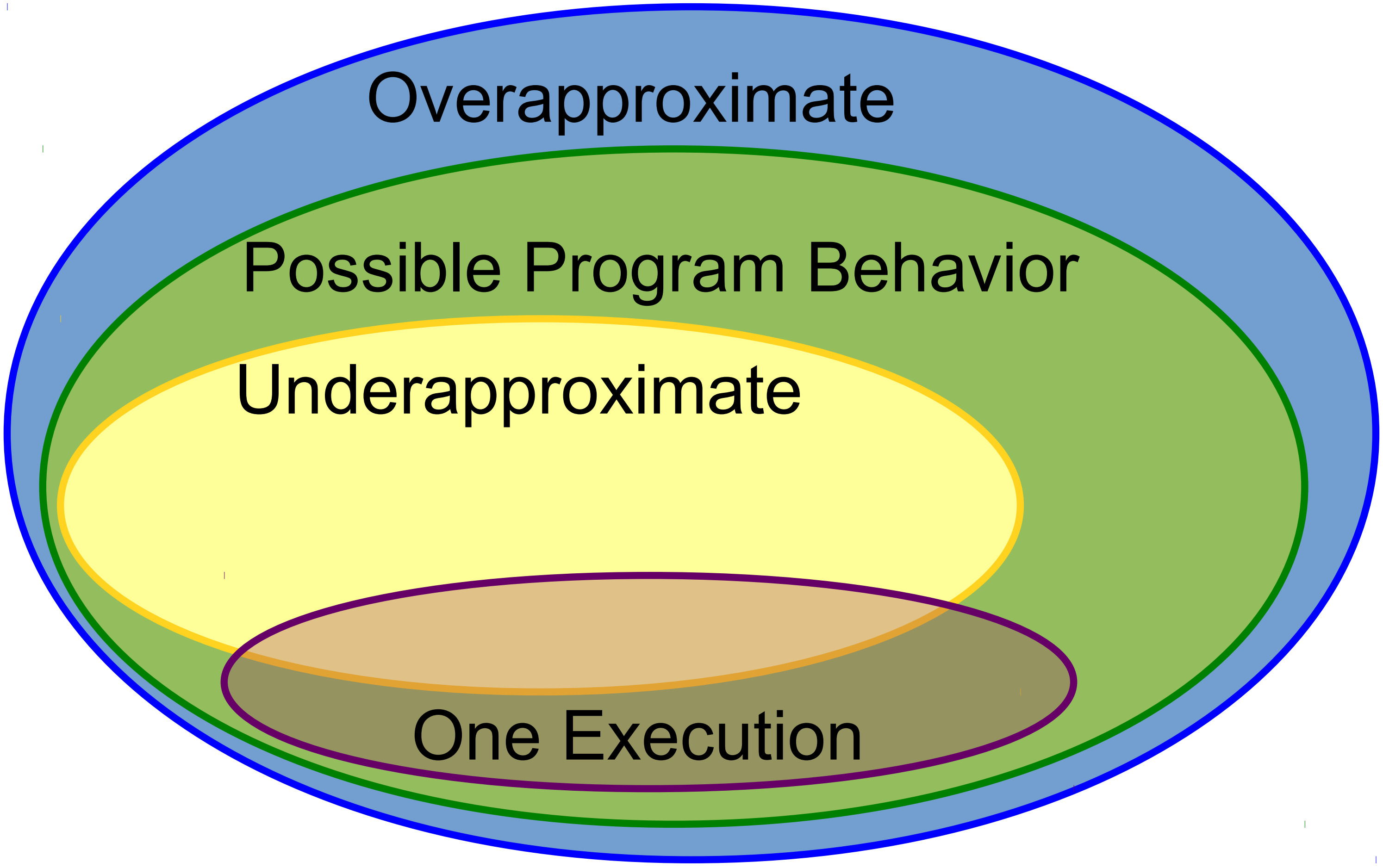
Overapproximate or underapproximate the problem, and try to solve this simpler version.

- Sound analyses
  - Overapproximate
  - Guaranteed to find violations of property
  - May raise false alarms
- Comlete analyses
  - Underapproximate
  - Reported violations are real
  - May miss violations

Striking the right balance is key to a useful analysis

Static Analysis

Modeled program behaviors



A Simple Example

Q: Is a particular number ever negative?

- Might be an offset into invalid memory!

Approximate the problem

A Simple Example

Q: Is a particular number ever negative?

- Might be an offset into invalid memory!

Approximate the problem

- Concrete domain: integers
- Abstract domain: {-,0,+} $\cup$ {$\top$,$\bot$}

A Simple Example

Q: Is a particular number ever negative?

– Might be an offset into invalid memory!

Approximate the problem

- Concrete domain: integers
- Abstract domain: {-,0,+} $\cup$ {$\top$,$\bot$}

concrete(x) = 5 → abstract(x) = +
concrete(y) = -3 → abstract(y) = -
concrete(z) = 0 → abstract(z) = 0

Combines sets of the concrete domain

A Simple Example

- Transfer Functions show how to evaluate this approximated program:
  - **+** + **+** → **+**
  - **-** + **-** → **-**
  - **0** + **0** → **0**
  - **0** + **-** → **-**
  - …
  - **+** + **-** → ⊤(unknown)
  - **…** / **0** → ⊥(undefined)

A Simple Example

- Transfer Functions show how to evaluate this approximated program:
  - **+** + **+** → **+**
  - **-** + **-** → **-**
  - **0** + **0** → **0**
  - **0** + **-** → **-**
  - …
  - **+** + **-** → ⊤(unknown)
  - **…** / **0** → ⊥(undefined)

- Can be subtle.
  - The above is not sound or complete. Why?

A Simple Example

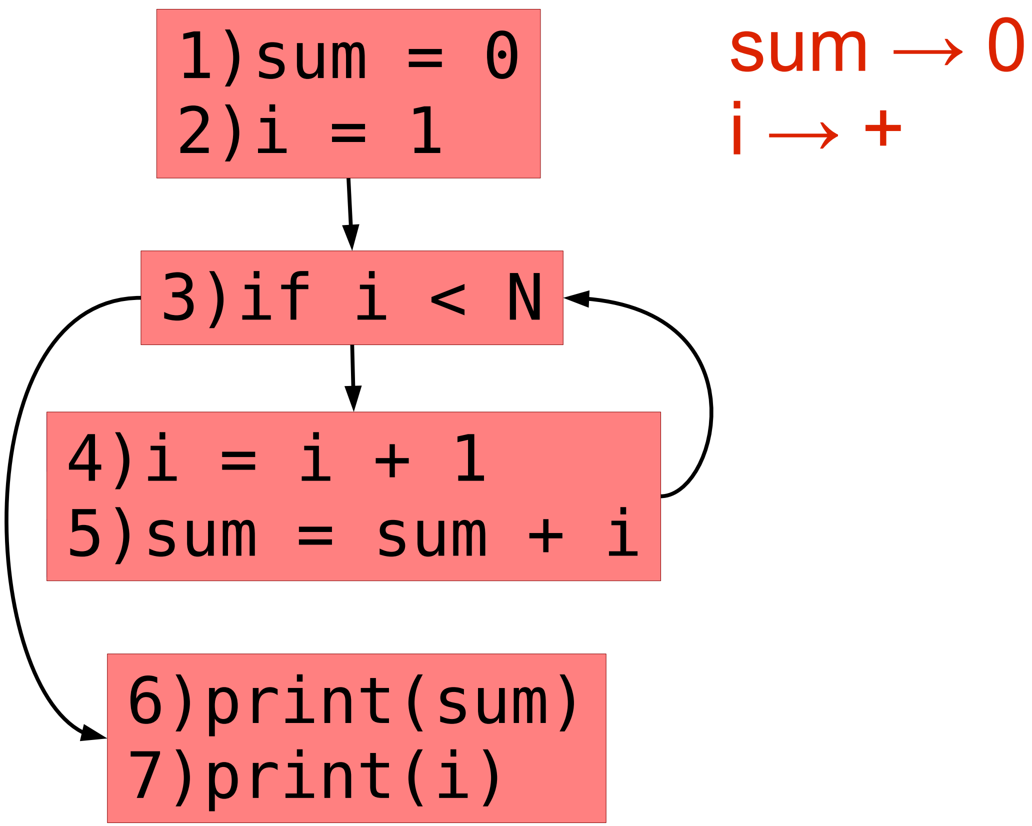
- Transfer Functions show how to evaluate this approximated program:
  - **+** + **+** → **+**
  - **-** + **-** → **-**
  - **0** + **0** → **0**
  - **0** + **-** → **-**
  - …
  - **+** + **-** → ⊤(unkno
  - **…** / **0** → ⊥(unde

Consider a divide by 0 analysis.
What are:
True Positives
False Positives
True Negatives
False Negatives

- Can be subtle.
  - The above is not sound or complete. Why?

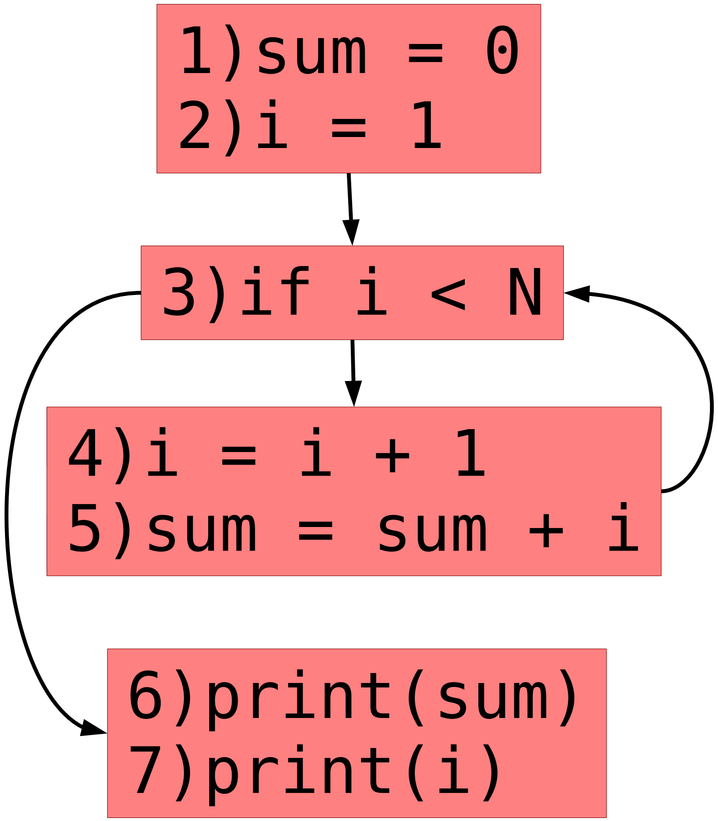
Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.



Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.

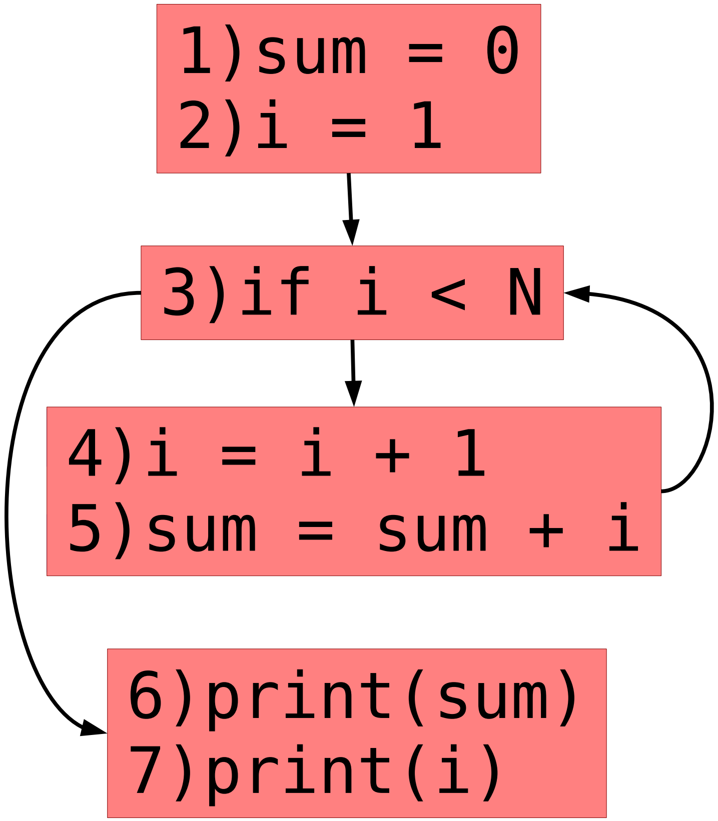


sum → 0
i → +

sum → 0
i → +

Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.



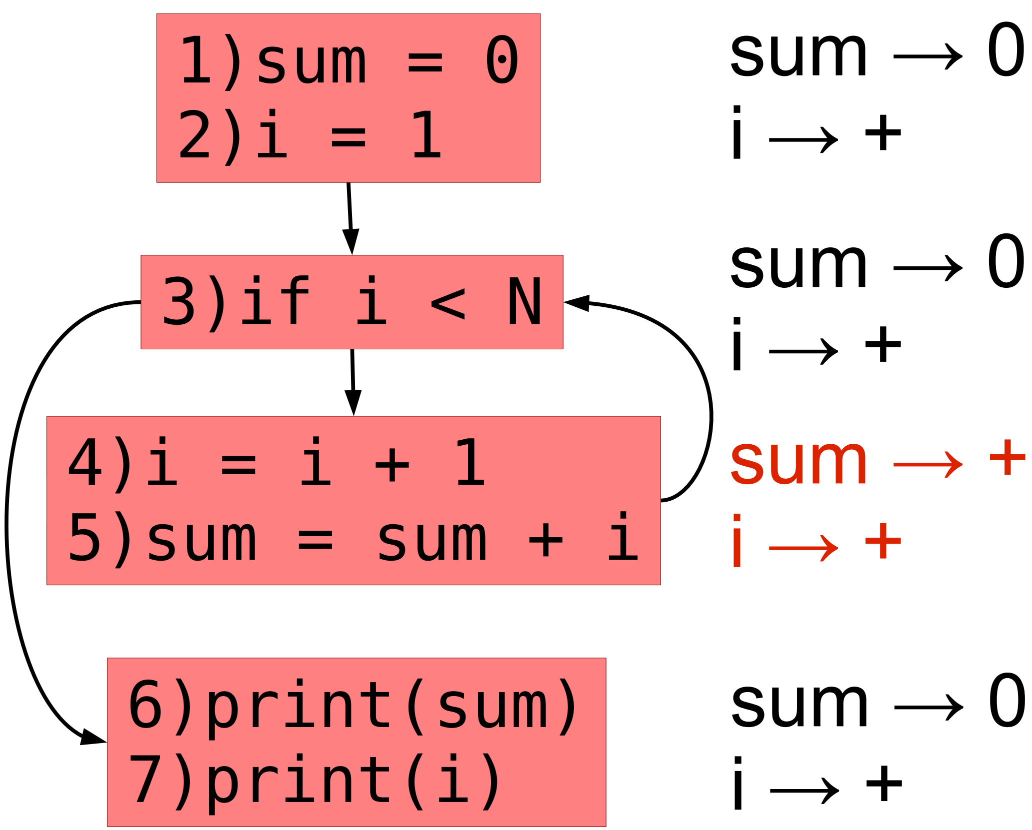
sum → 0
i → +

sum → 0
i → +

sum → 0
i → +

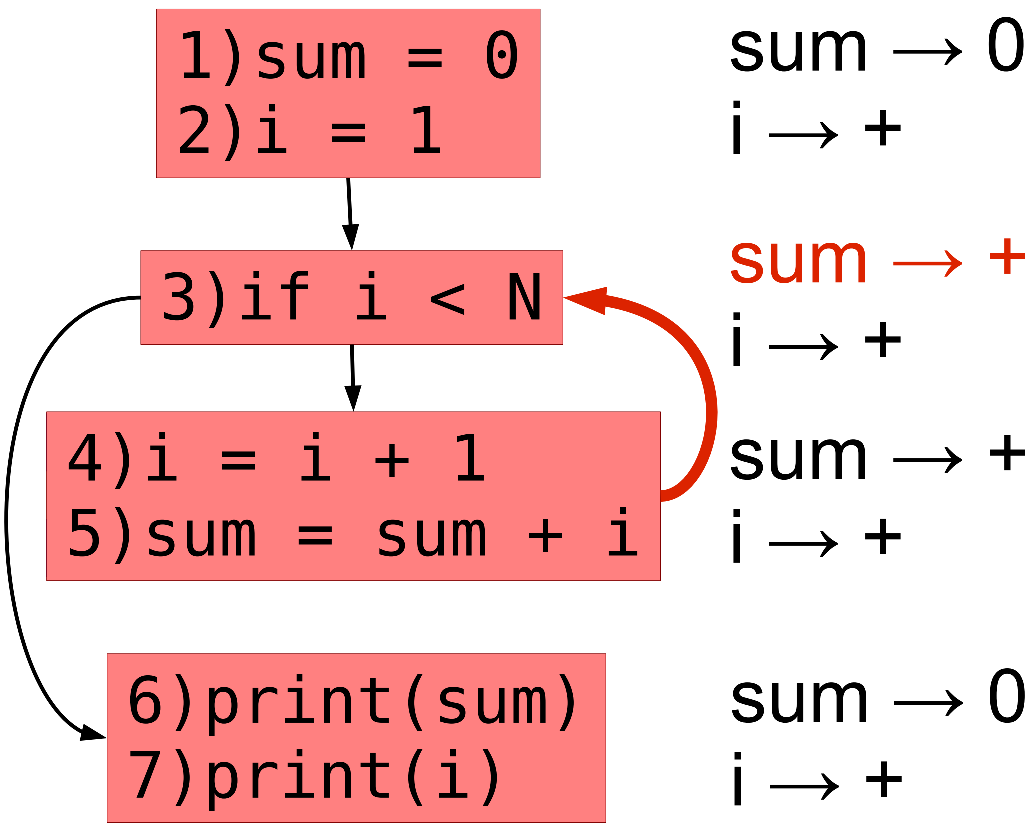
Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.



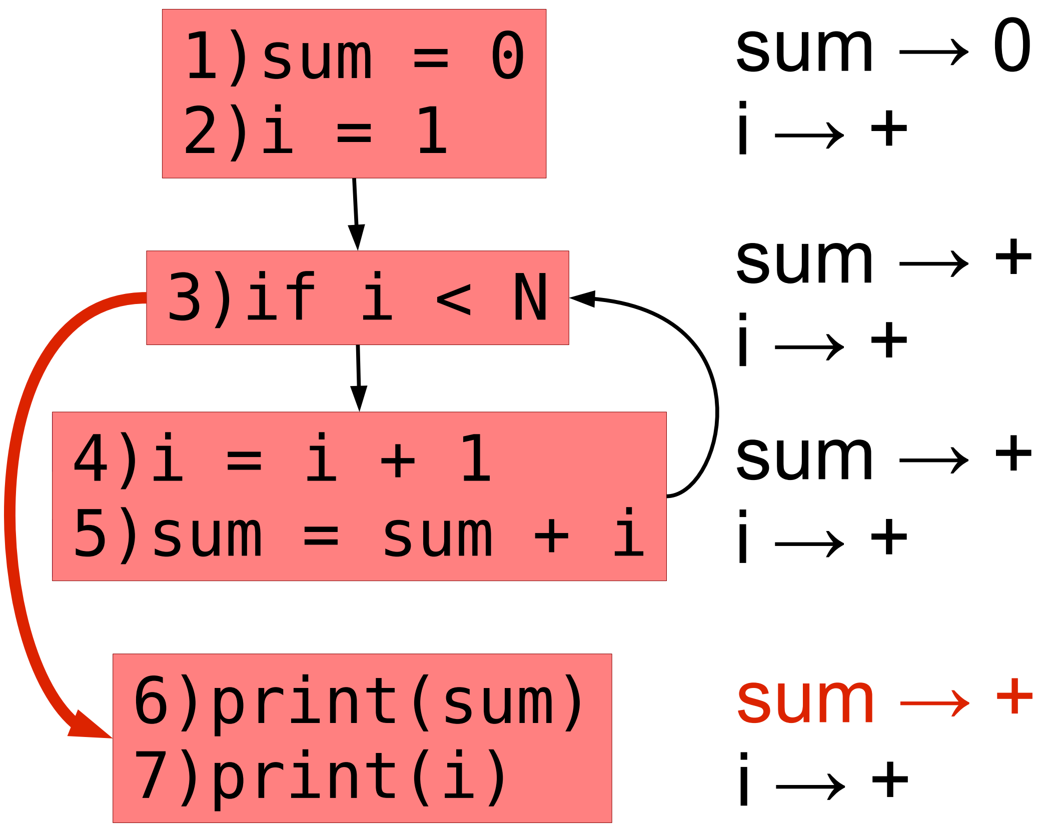
Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.



Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.



Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.
  - Continue until we reach a fixed point
  - (No more changes)
  - Proper ordering can improve the efficiency.

Data Flow Analysis

- Now model the abstract program state and propagate through the CFG.
  - Continue until we reach a fixed point
  - (No more changes)
  - Proper ordering can improve the efficiency.

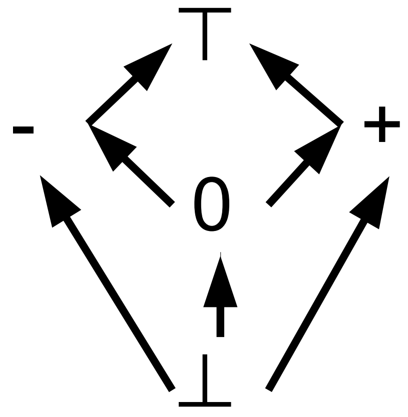
Will it always terminate?

Data Flow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function

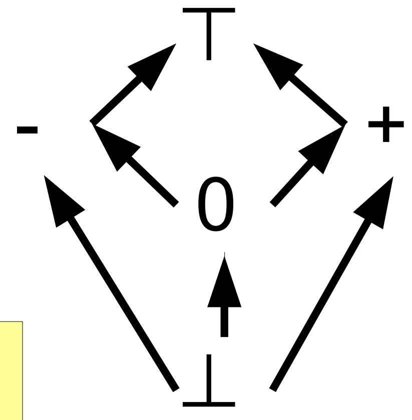
Data Flow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework
  - $\{-,0,+\} \cup \{\top,\bot\}$
  - They define a partial order
  - Abstract state can only move ***up*** lattice



Data Flow Analysis

- Guarantee termination by carefully choosing
  - The abstract domain
  - The transfer function
- For basic analyses, use a monotone framework
  - {-,0,+} ∪ {⊤,⊥}
  - They define a partial order
  - Abstract state can only move ***up*** lattice



Why is this enough?

Data Flow Analysis

- Note: need to model program state at each statement
- Proper ordering & a work list algorithm improves the efficiency

Static Analysis

- We've already seen a few static analyses:
  - Call graph construction
  - Points-to graph construction (What are MAY/MUST?)
  - Static slicing

Static Analysis

- We've already seen a few static analyses:
  - Call graph construction
  - Points-to graph construction (What are MAY/MUST?)
  - Static slicing
- The choices for approximation are why these analyses are imprecise.

Flow Insensitive Analysis

- Saw *flow sensitive* analysis
  - Modeling state at each statement is expensive
  - Scales to functions and small components
  - Usually not beyond 1000s of lines

Flow Insensitive Analysis

- Saw *flow sensitive* analysis
  - Modeling state at each statement is expensive
  - Scales to functions and small components
  - Usually not beyond 1000s of lines
- *Flow insensitive* analyses aggregate into a global state
  - Better scalability
  - Less precision
  - "Does this function modify global variable X?"

Context Sensitive Analyses

- Program behavior may be dependent on the call stack / calling context.
  - "If bar() is called by foo(), then it is exception free."
  - Can enable more precise *interprocedural* analyses

Static Analysis

- We'll cover this further as necessary during the semester

Project 1 & LLVM

Next Week:
Dynamic Analysis,
Profiling,
Testing,
Concurrency
Security