CMPT 745
Software Engineering

What are programs?

Nick Sumner
wsumner@sfu.ca

What is a program?

- A program ***communicates*** a set of ***instructions*** for performing as task

What is a program?

- A program ***communicates*** a set of ***instructions*** for performing as task
  - Do you always communicate in the same way?

What is a program?

- A program ***communicates*** a set of ***instructions*** for performing as task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?

What is a program?

- A program ***communicates*** a set of ***instructions*** for performing as task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?
- Programs communicate to different actors
  - Team mates
  - Compilers
  - Government entities

What is a program?

- A program ***communicates*** a set of ***instructions*** for performing as task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?
- Programs communicate to different actors
  - Team mates
  - Compilers
  - Government entities
- Different programs have different requirements
  - Performance over all
  - Security!

What is a program?

- A program ***communicates*** a set of ***instructions*** for performing as task
  - Do you always communicate in the same way?
  - Do you always have the same concerns for different ways of communicating?
- Programs communicate to different actors
  - Team mates
  - Compilers
  - Government entities
- Different programs have different requirements
  - Performance over all
  - Security!
- We cannot reason about programs in only one way

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

1001101
0101011
1101011
0001110

frob.exe

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

1001101
0101011
1101011
0001110

frob.exe

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
    - Often used in reverse engineering or security tasks

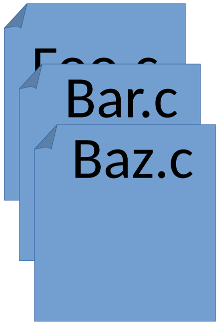
Program Representation

- 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
    - Often used in reverse engineering or security tasks

Why might binaries be good for security tasks?

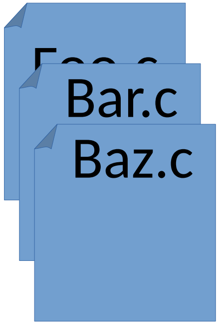
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze
- Difficult models:
  - Compiled binaries
  - Source code



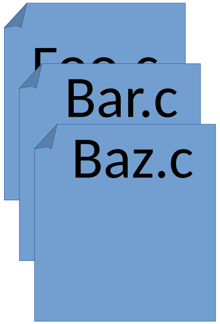
Program Representation

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



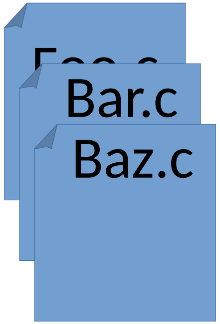
Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze
- Difficult models:
  - Compiled binaries
  - Source code
    - Very language specific
    - Relationships can be hard to extract



Program Representation

- Before we can reason about programs, we must have a vocabulary and a *model* to analyze
- Difficult models:
  - Compiled binaries
  - Source code
    - Very language specific
    - Relationships can be hard to extract
    - Often used when relating to comments or specs



Program Representation

- 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 analyze

Program Representation

Core graph representations for analysis:

1) Abstract Syntax Trees
2) Control Flow Graphs
3) Program Dependence Graphs
4) Call Graphs
5) Points-to Graphs
6) Emerging Representations for ML

1) Abstract Syntax Trees

- Lifts the source into a canonical tree form

1) Abstract Syntax Trees

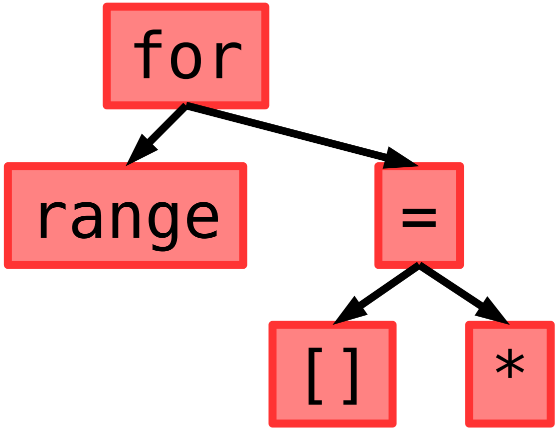
- Lifts the source into a canonical tree form

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

1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
  - Internal nodes are operators, statements, etc.

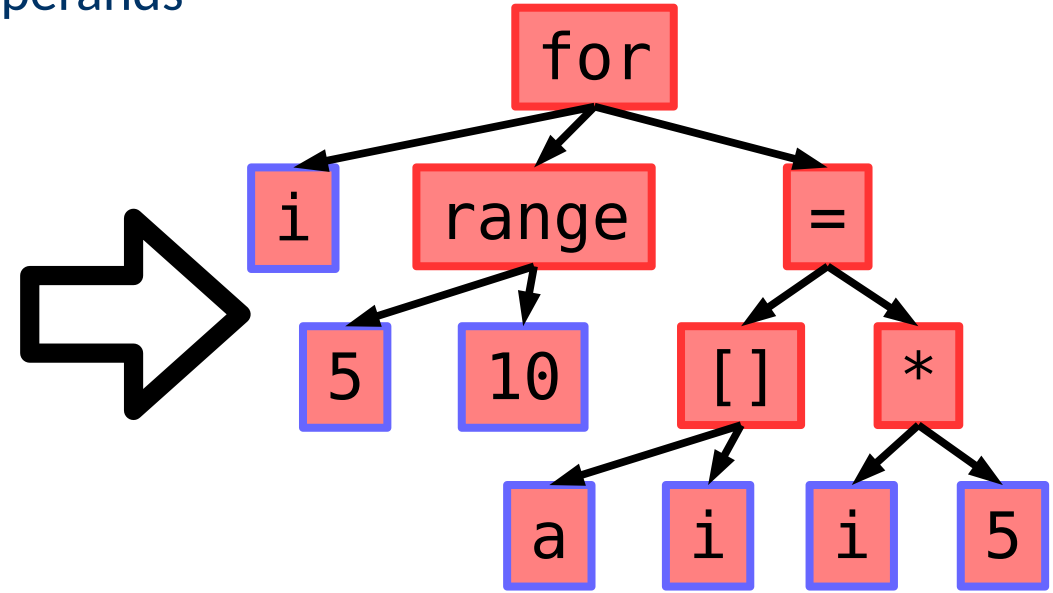
```
for i in range(5,10):
    a[i] = i * 5
```



1) Abstract Syntax Trees

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

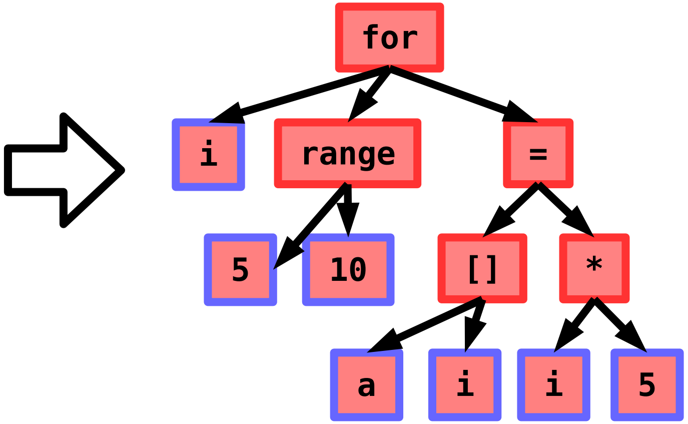
```
for i in range(5,10):
    a[i] = i * 5
```



1) Abstract Syntax Trees

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

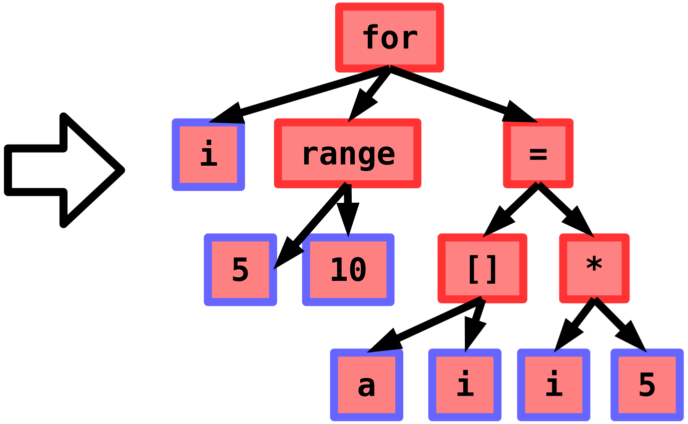
```
for i in range(5,10):
    a[i] = i * 5
```



1) Abstract Syntax Trees

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

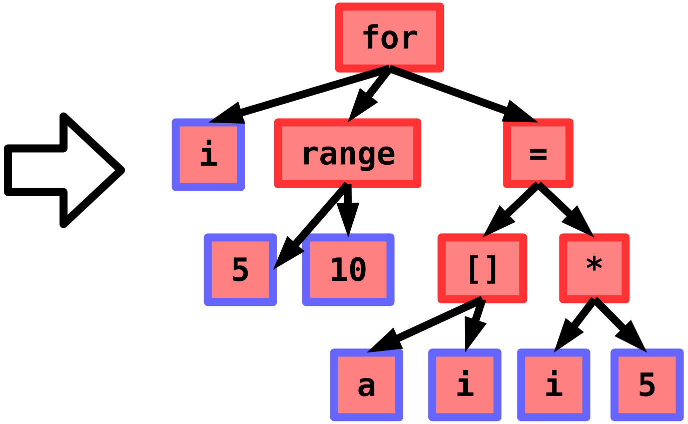
```
for i in range(5,10):
    a[i] = i * 5
```



1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:
  - Simple bug patterns
  - Style checking

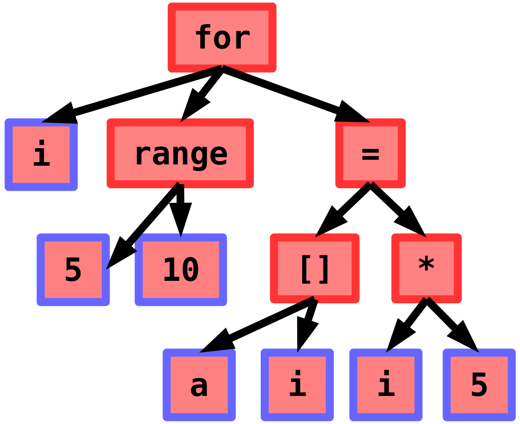
```
for i in range(5,10):
    a[i] = i * 5
```



1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:
  - Simple bug patterns
  - Style checking
  - Refactoring

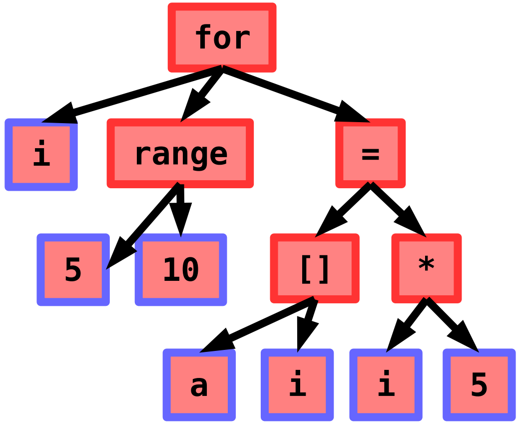
```
for i in range(5,10):
    a[i] = i * 5
```



1) Abstract Syntax Trees

- Lifts the source into a canonical tree form
- Used for syntax analysis & transformation:
  - Simple bug patterns
  - Style checking
  - Refactoring
  - Training prediction/completion models

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

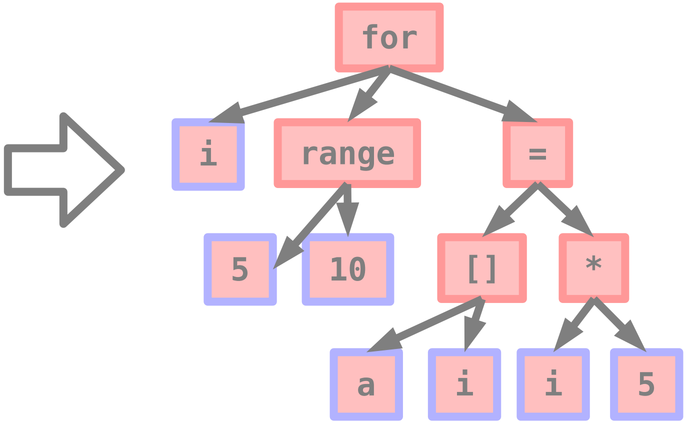


1) Abstract Syntax Trees

- Lifts the source into a canonical tree form

But:
1) The same program may still be spelled many ways

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

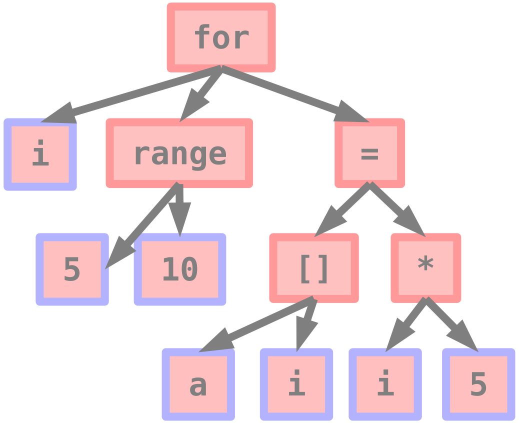


1) Abstract Syntax Trees

- Lifts the source into a canonical tree form

But:
1) The same program may still be spelled many ways
2) Some information is *implicit* rather than *explicit*

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



2) Control Flow Graphs

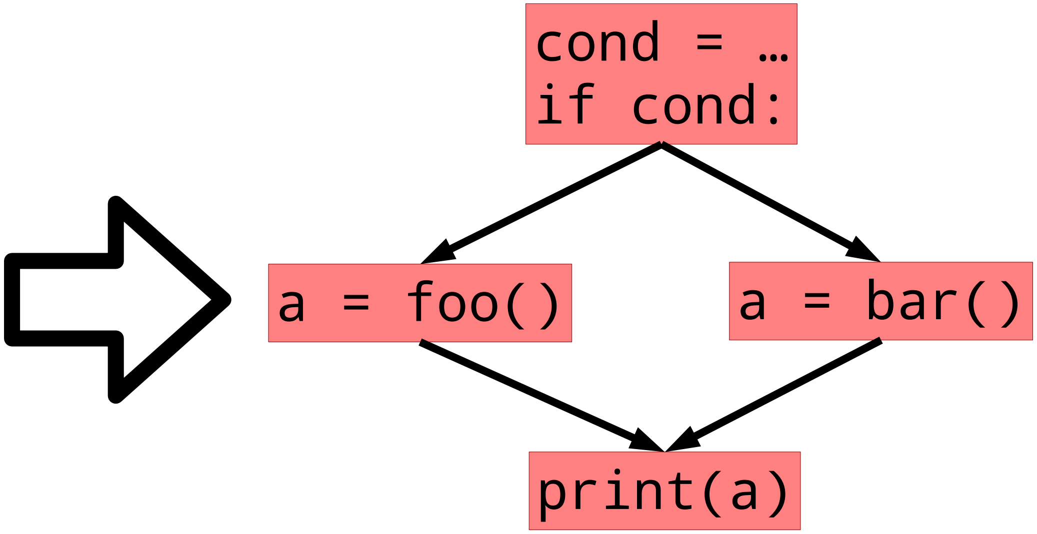
- Express the possible decisions and possible paths through a program

```
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

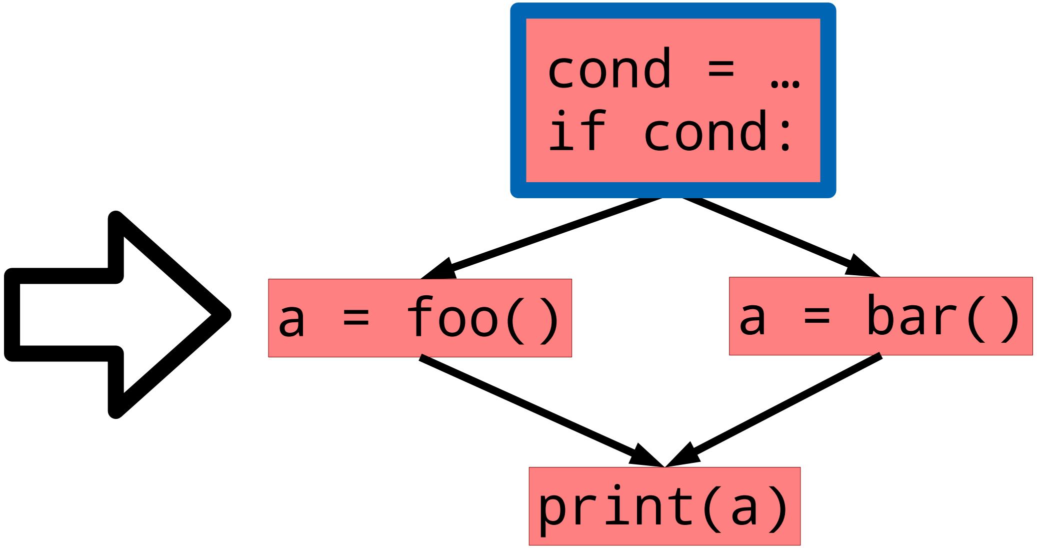
```
cond = input()
if cond:
  a = foo()
else:
  a = bar()
print(a)
```



2) Control Flow Graphs

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

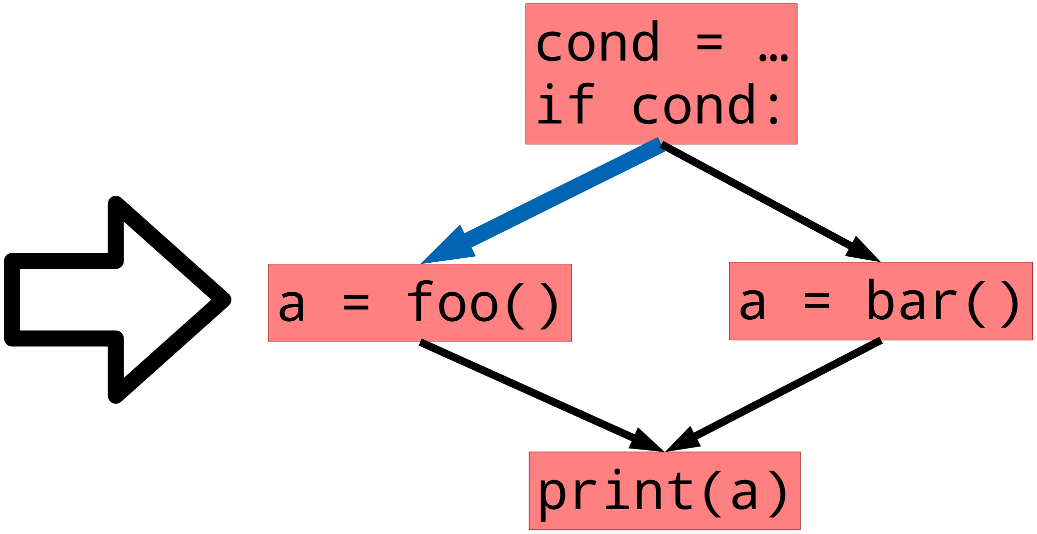
```
cond = input()
if cond:
  a = foo()
else:
  a = bar()
print(a)
```



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

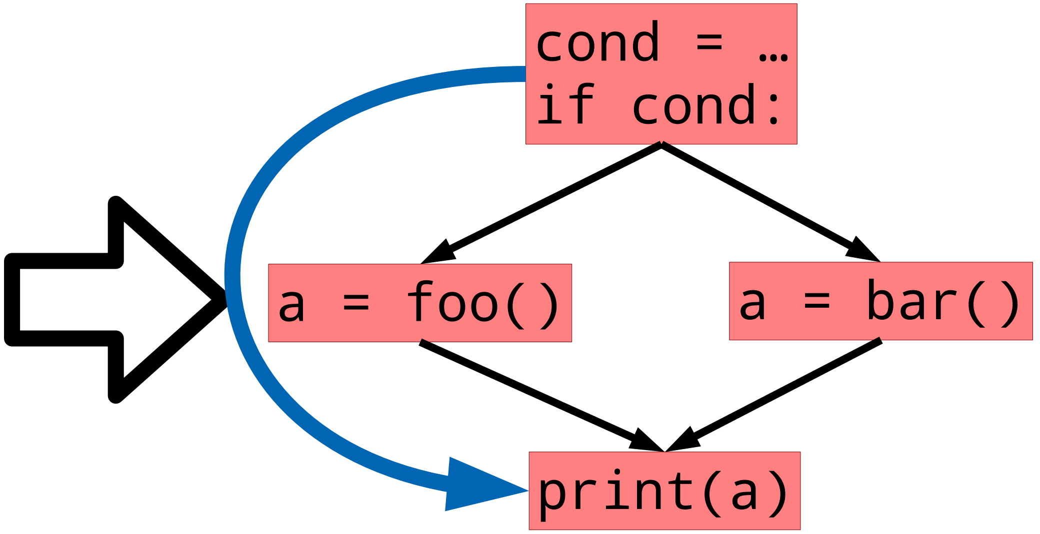
```
cond = input()
if cond:
  a = foo()
else:
  a = bar()
print(a)
```



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

2) Control Flow Graphs (CFGs)

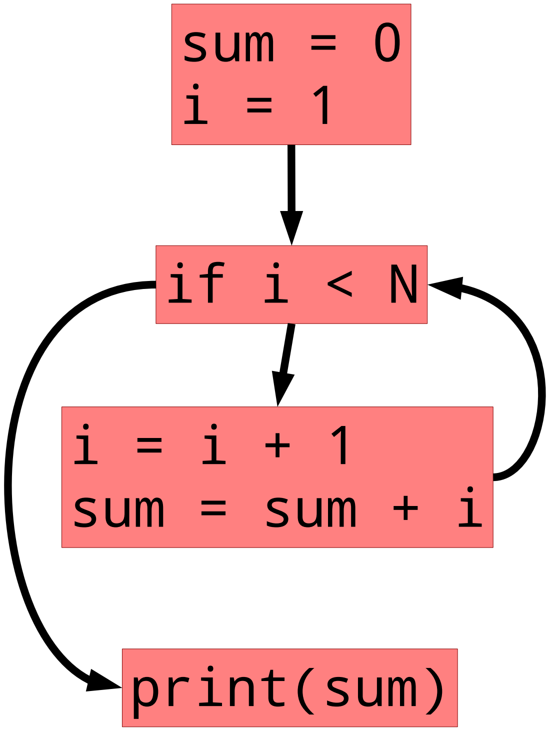
- Language specific features are often abstracted away

```
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

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

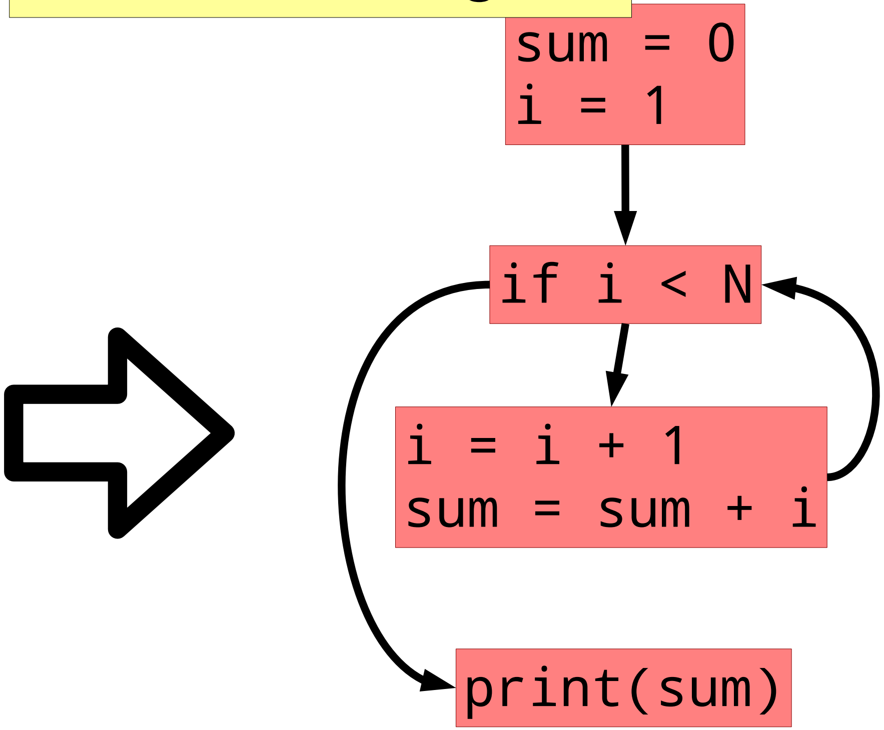


2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

The 'while' is gone

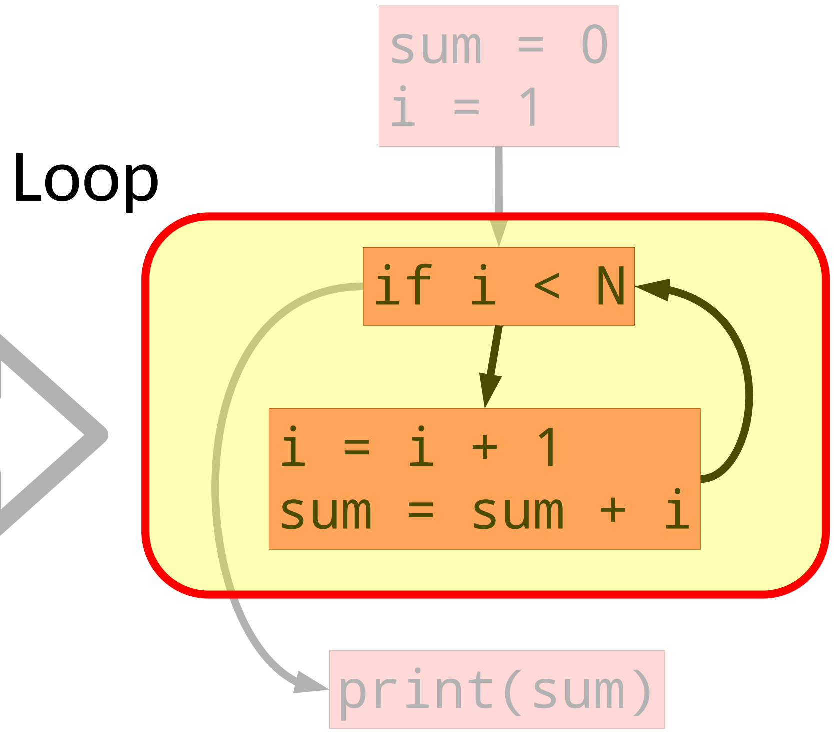
```
sum = 0
i = 1
while i < N:
  i = i + 1
  sum = sum + i
print(sum)
```



2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

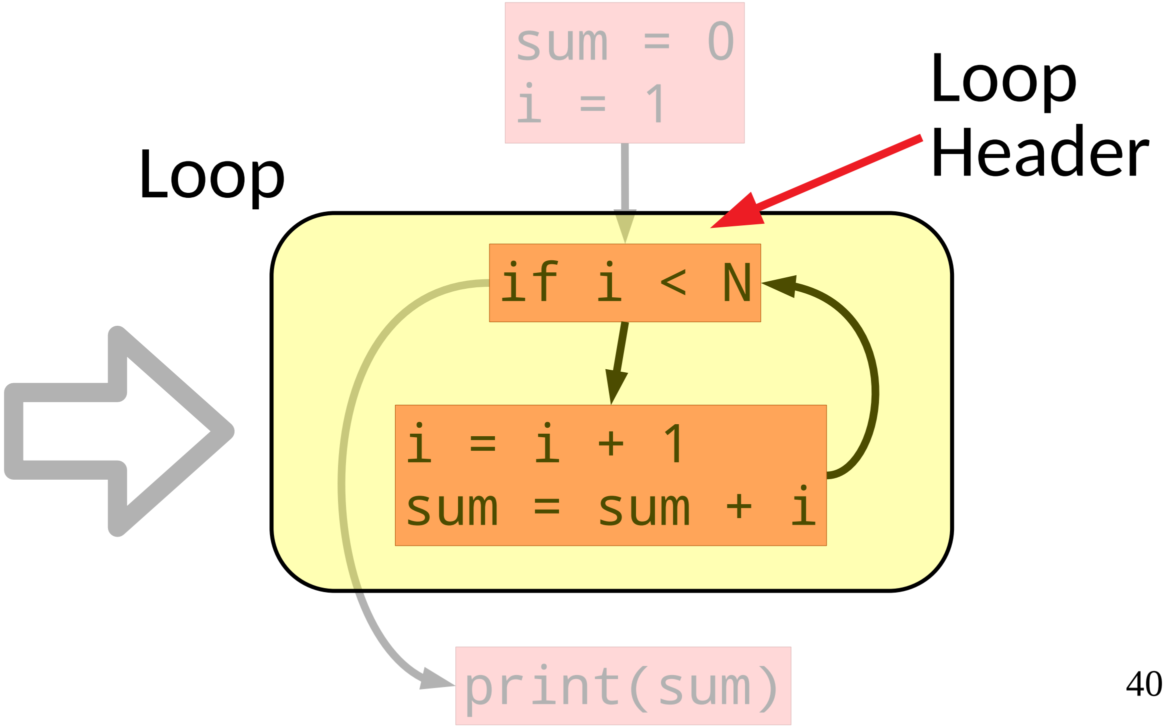
```
sum = 0
i = 1
while i < N:
   i = i + 1
   sum = sum + i
print(sum)
```



2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

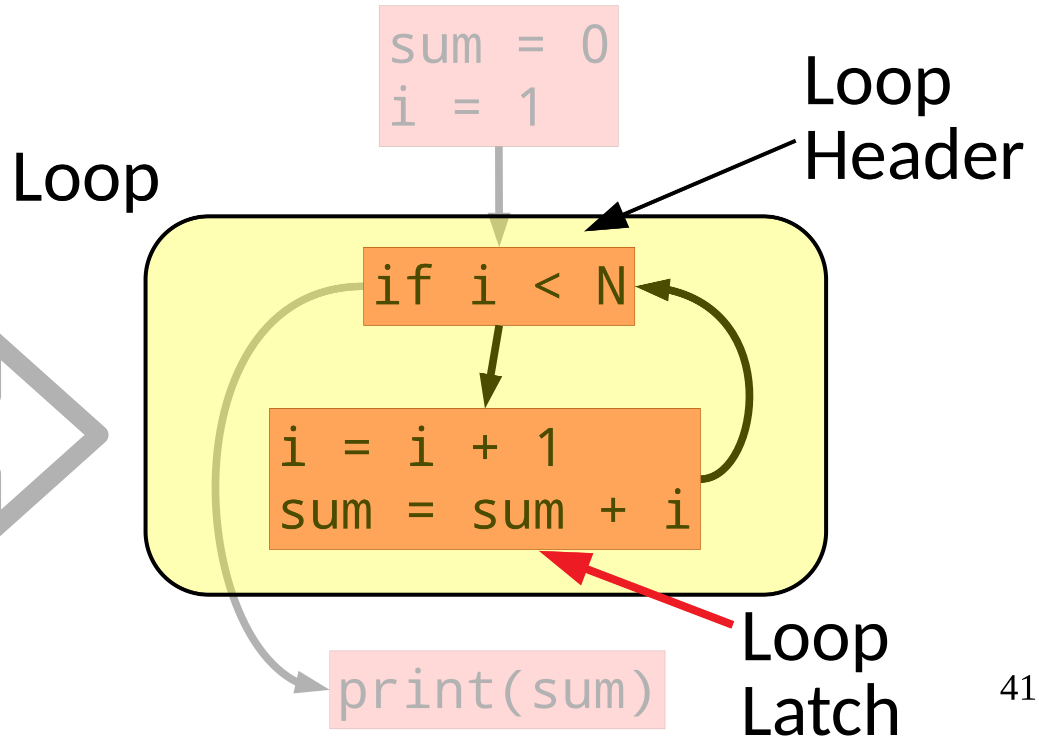
```
sum = 0
i = 1
while i < N:
   i = i + 1
   sum = sum + i
print(sum)
```



2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

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

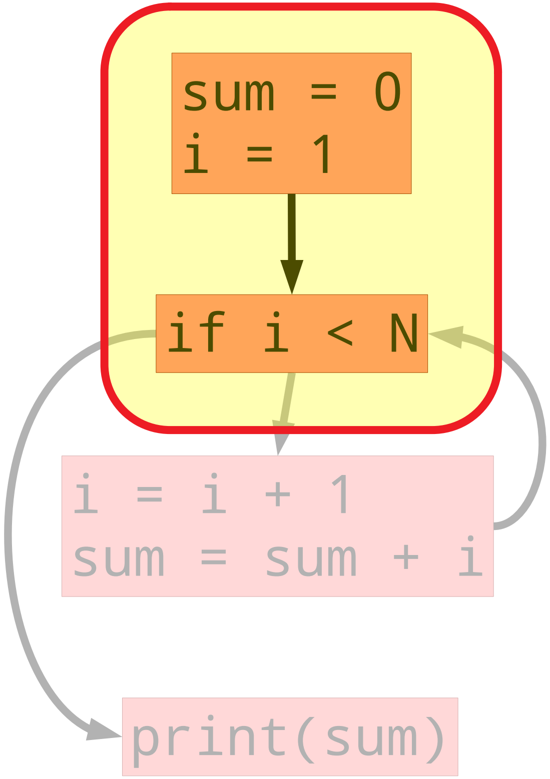


2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

Why is the 'if' in a separate block?

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

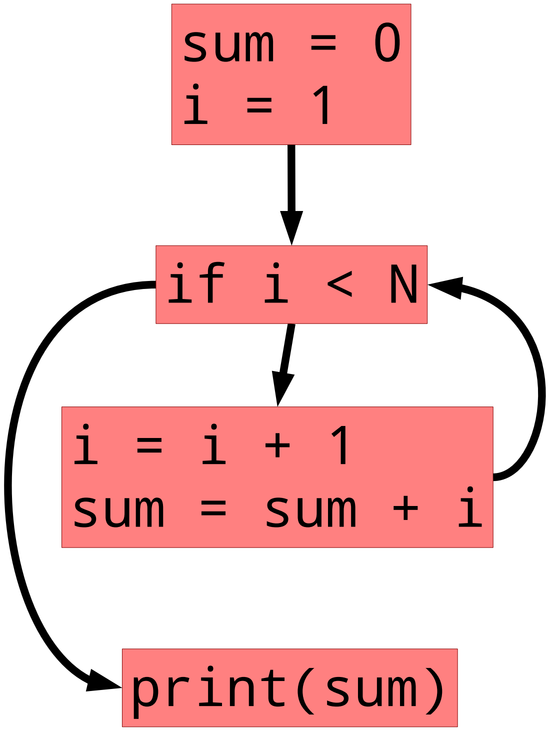


2) Control Flow Graphs (CFGs)

- Language specific features are often abstracted away

```
sum = 0
i = 1

print(sum)
```

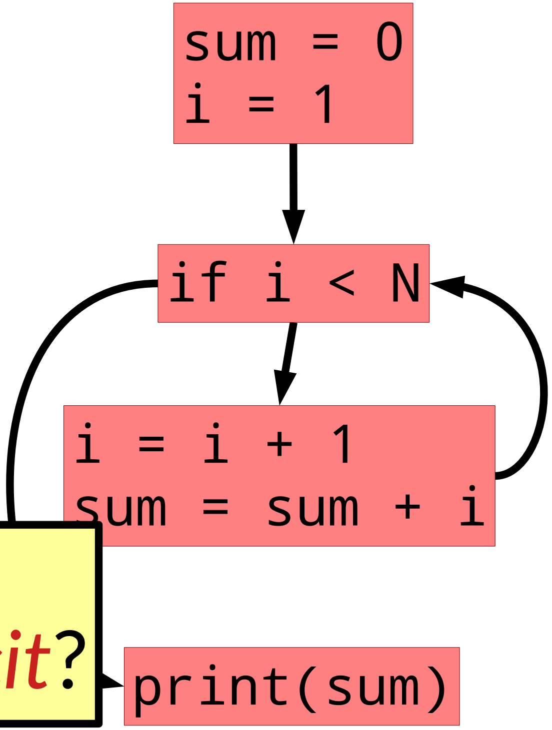
What would the CFG of the equivalent 'for' look like?



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)
```



What information is *explicit*?
What information is still *implicit*?

3)Program Dependence Graph (PDG)

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

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

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→X shows that Y influences X

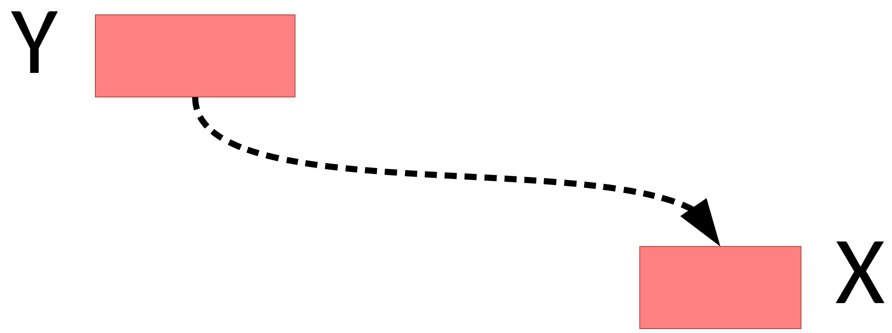
3)Program Dependence Graph (PDG)

- A ***Program Dependence Graph*** captures how instructions can influence each other
- Instruction X depends on Y if Y *can influence* X
- 2 main types of influence:
  - Data dependence
  - Control dependence

Data Dependence

X data depends on Y if

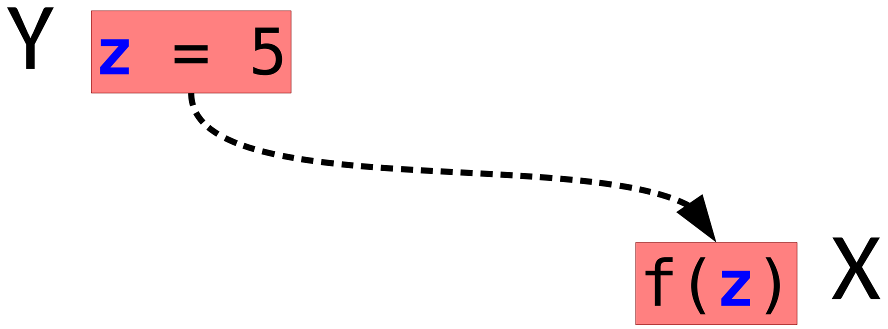
- There exists a path from Y to X in the CFG



Data Dependence

X data depends on Y if

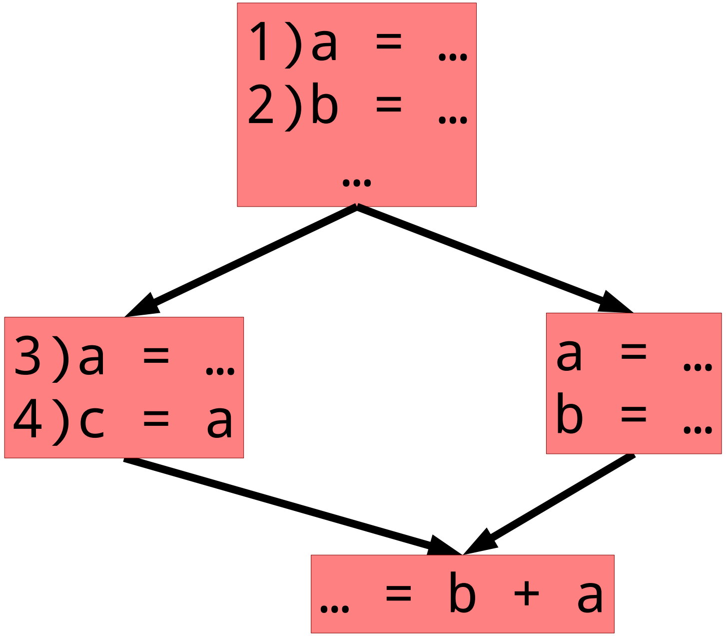
- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X



Data Dependence

X data depends on Y if

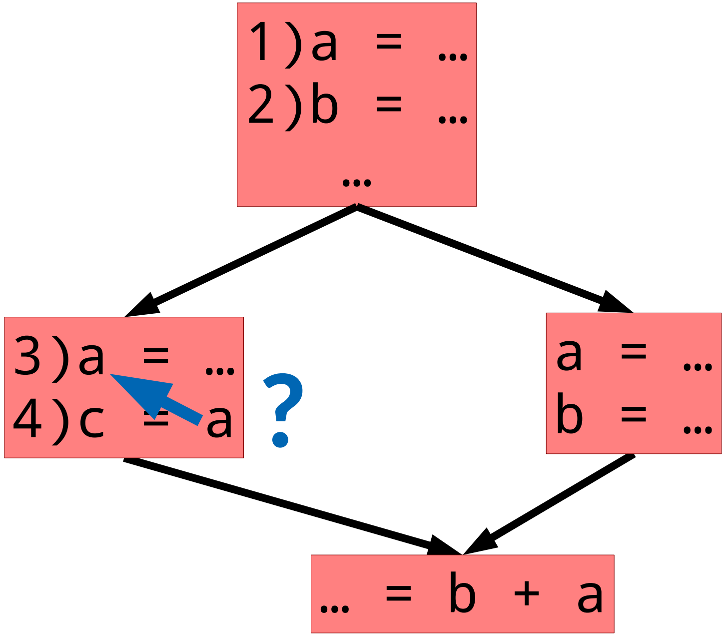
- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X



Data Dependence

X data depends on Y if

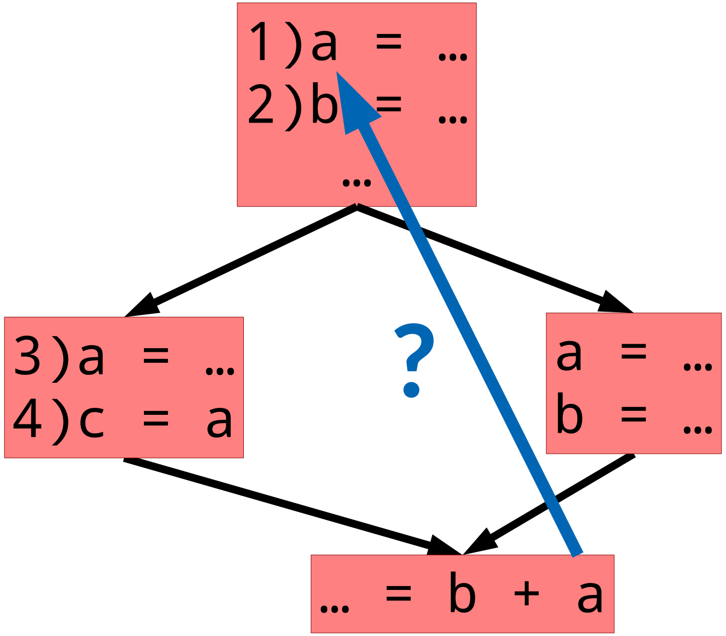
- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X



Data Dependence

X data depends on Y if

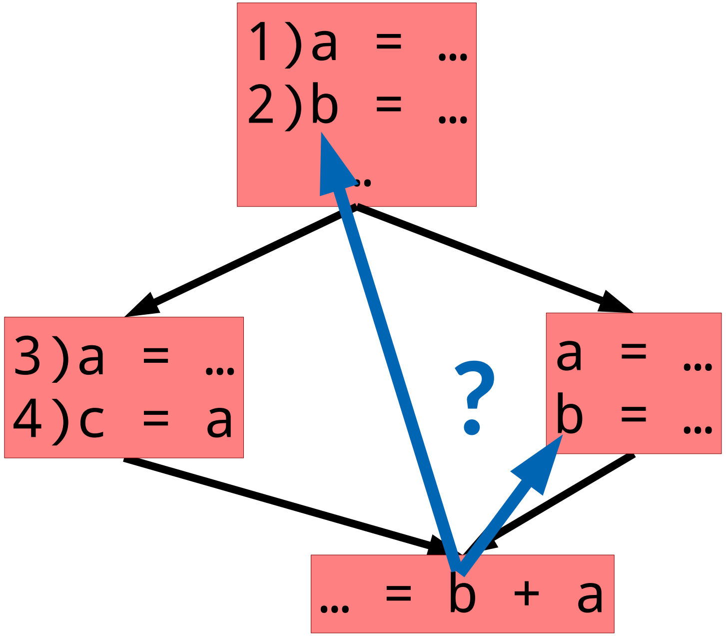
- There exists a path from Y to X in the CFG
- A variable/value definition at Y is used at X



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

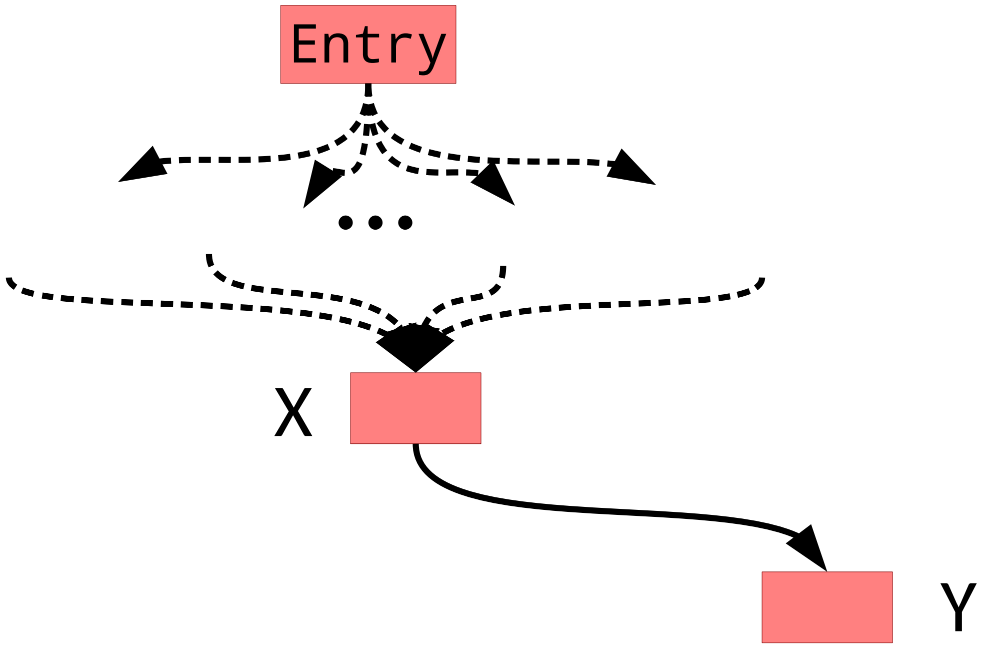
Preliminary: X dominates Y if

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

Control Dependence

Preliminary: X dominates Y if

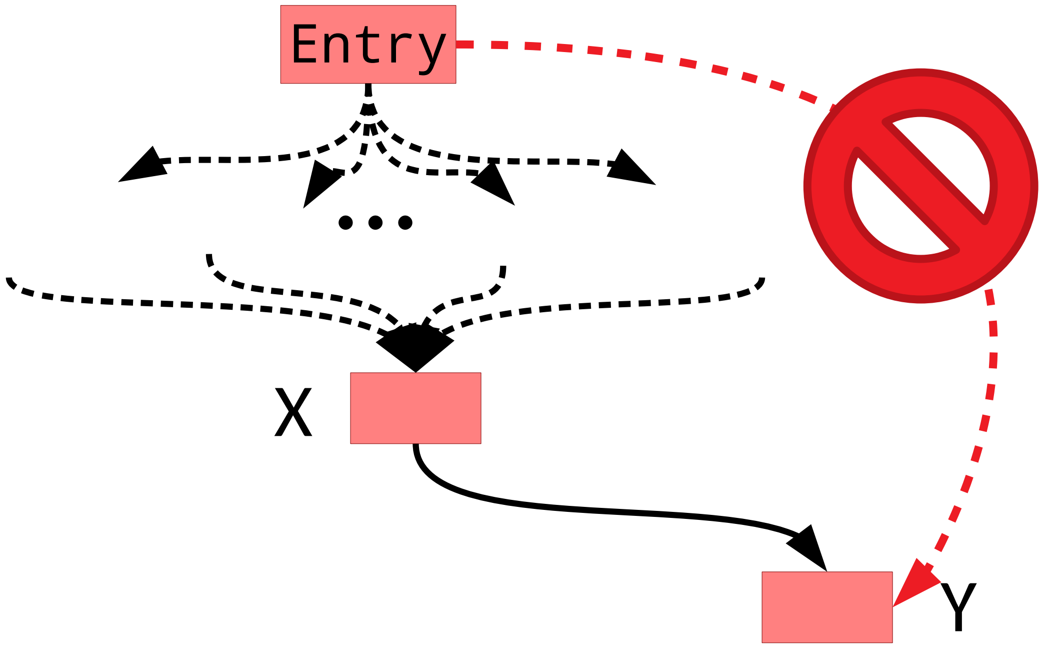
- every path from the entry node to Y passes X
  - strict, normal, & immediate dominance



Control Dependence

Preliminary: X dominates Y if

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

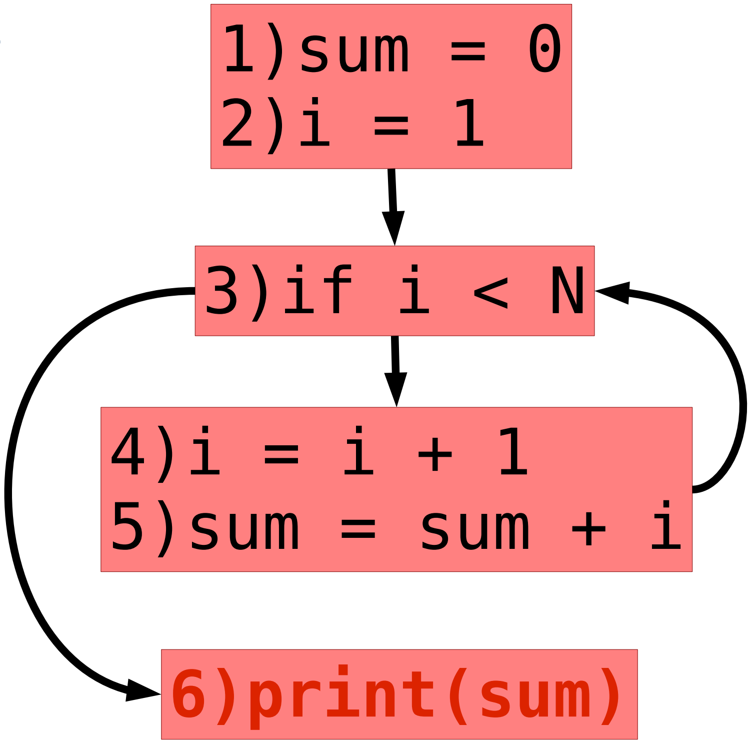


Control Dependence

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



DOM(6)= ?

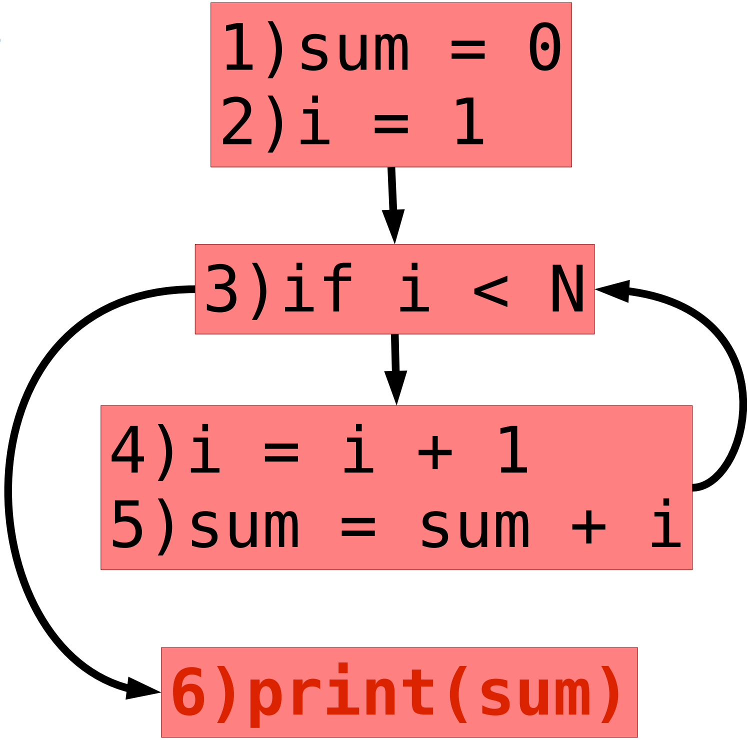
IDOM(6)= ?

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)
```



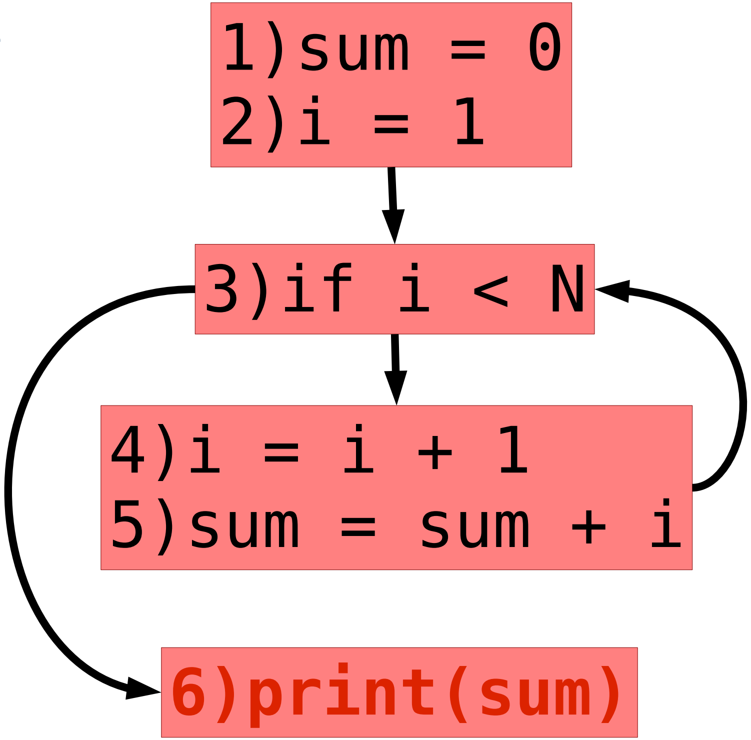
DOM(6)={1,2,3,6} IDOM(6)= ?

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

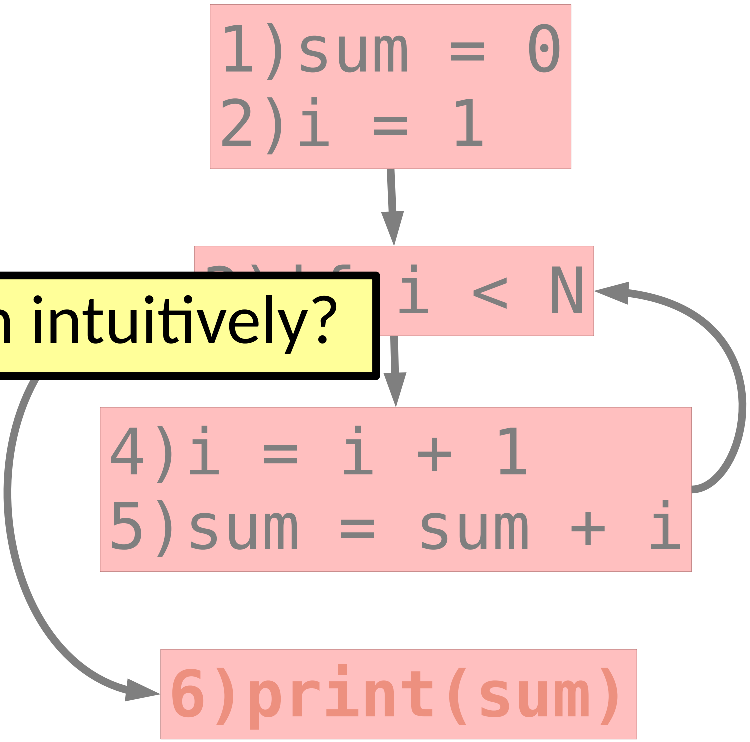
Control Dependence

Preliminary: X dominates Y if

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

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

What does this mean intuitively?



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

Control Dependence

Preliminary: X ***post*** dominates Y if

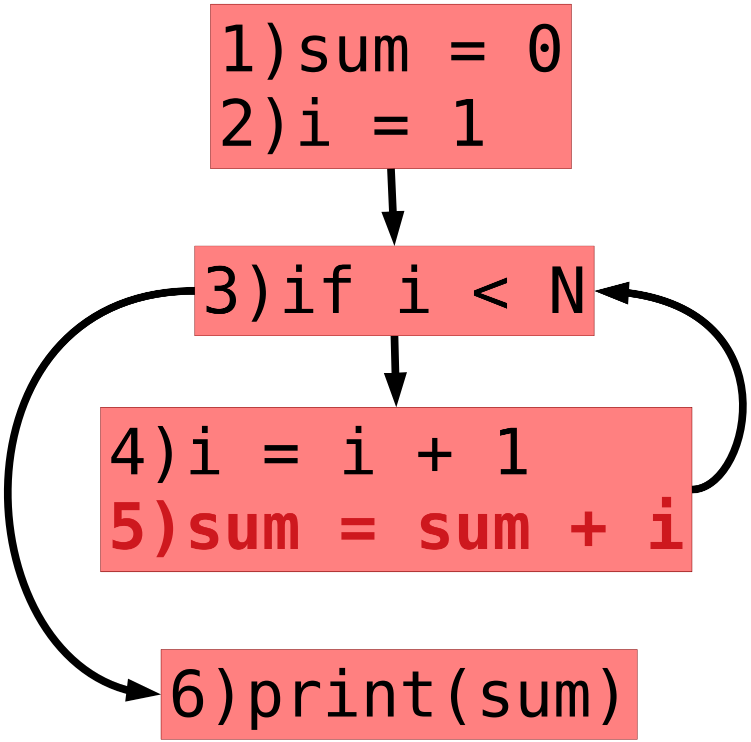
- every path from the ***Y to exit*** passes X
  - strict, normal, & immediate dominance

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



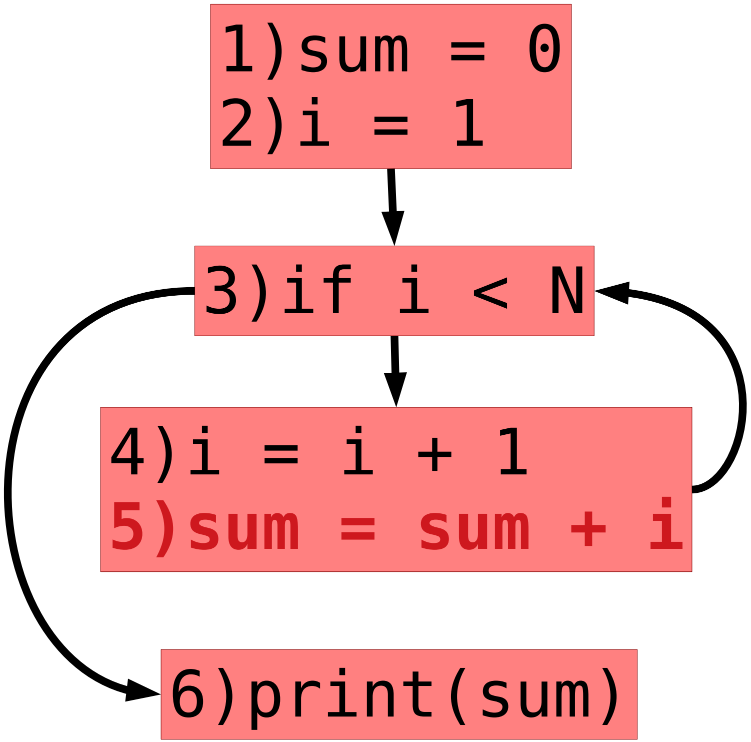
PDOM(5)= **?**   IPDOM(5)= **?**

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



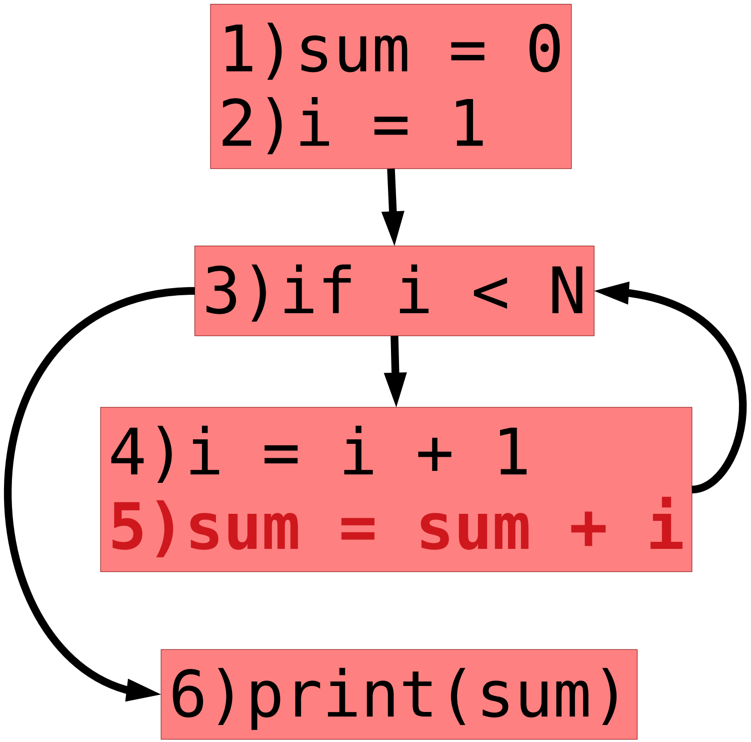
PDOM(5)={3,5,6}   IPDOM(5)= ?

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



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

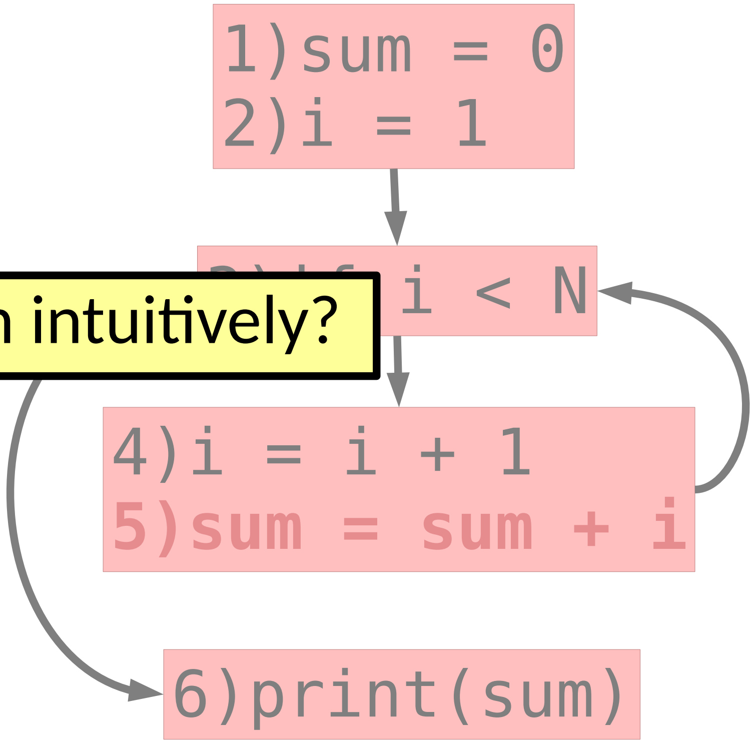
Control Dependence

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

What does this mean intuitively?



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

Control Dependence (Finally)

Y is control dependent on X iff

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 (Finally)

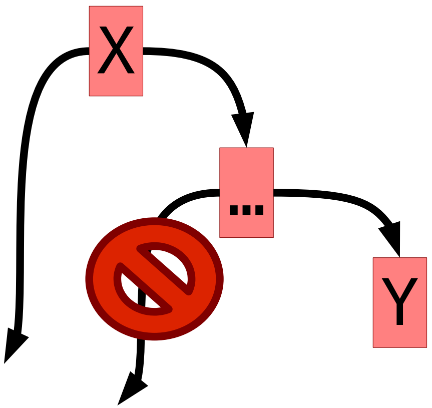
Y is control dependent on X iff

- 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



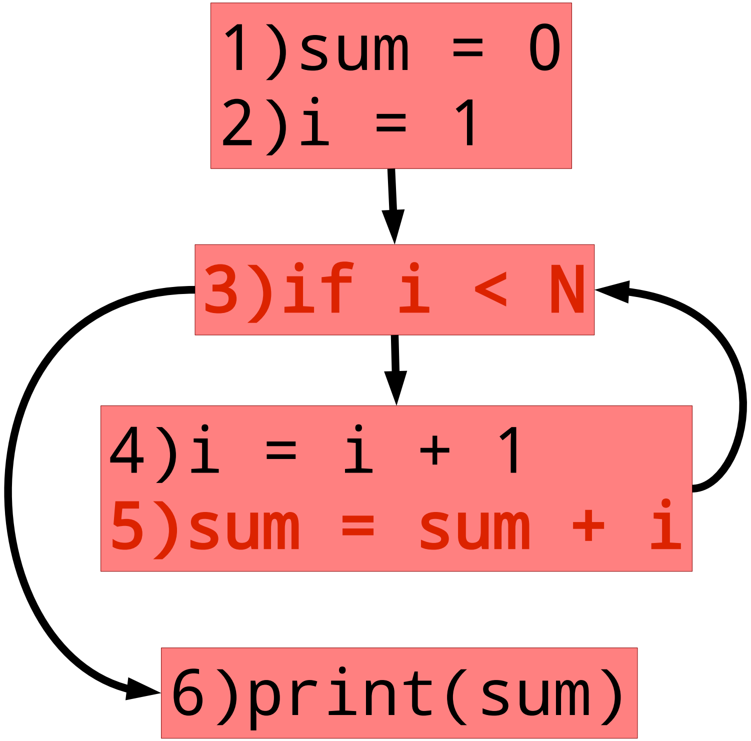
Control Dependence

- 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

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

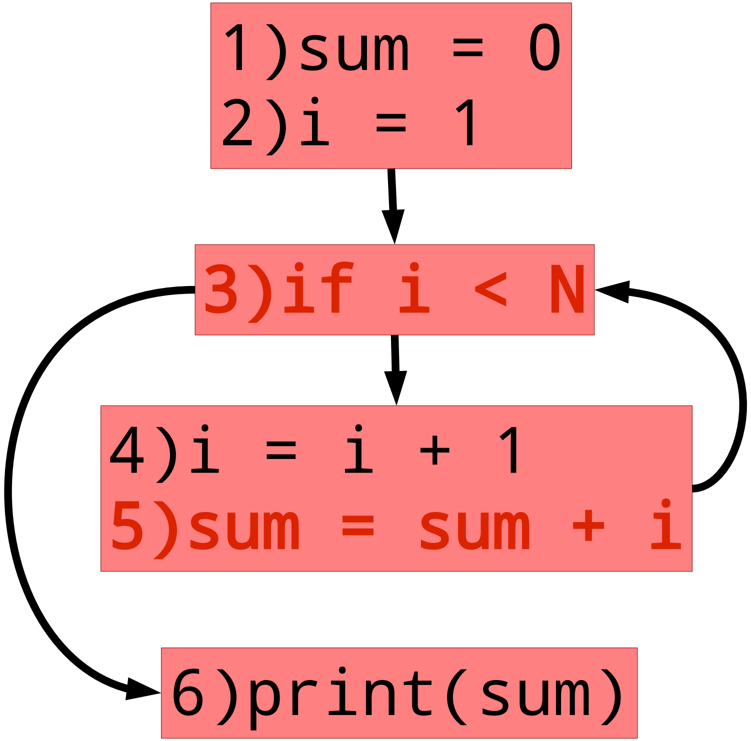
```
1)sum = 0
2)i = 1
3)while i < N:
4)   i = i + 1
5)   sum = sum + i
6)print(sum)
```



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)? CD(3)

Control Dependence

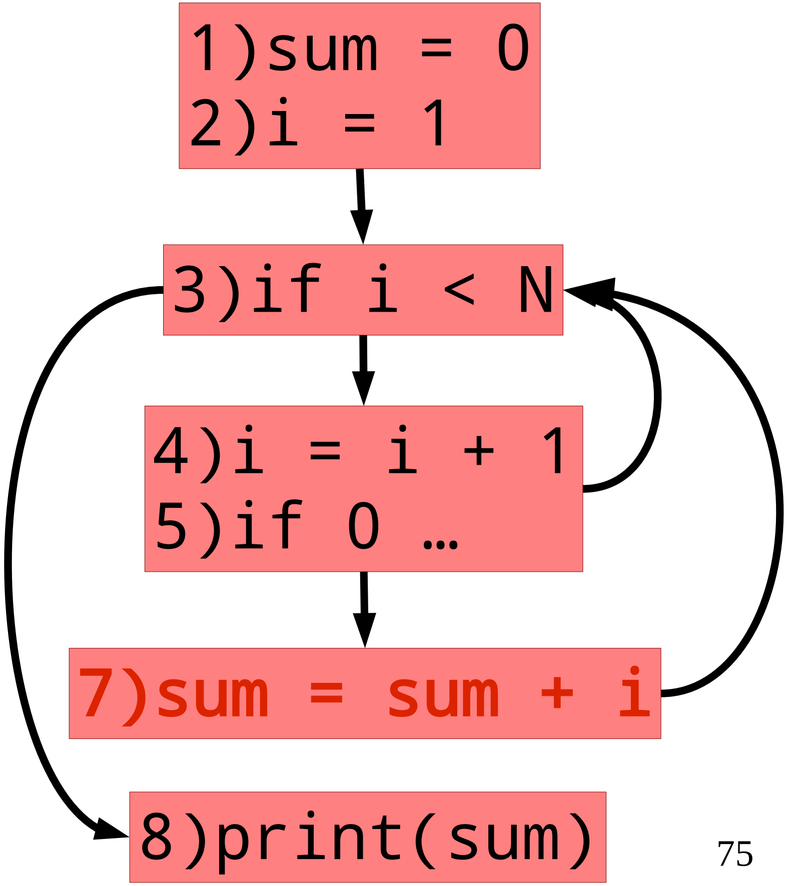
- There exists a path from X to Y s.t. Y post dominates every node between X and Y.
- Y does not strictly post dominate X

```
sum = 0
i = 1
while i < N:
  i = i + 1
  if 0 == i%2:
    continue
  sum = sum + i
print(sum)
```

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)  if 0 == i%2:
6)    continue
7)  sum = sum + i
8)print(sum)
```

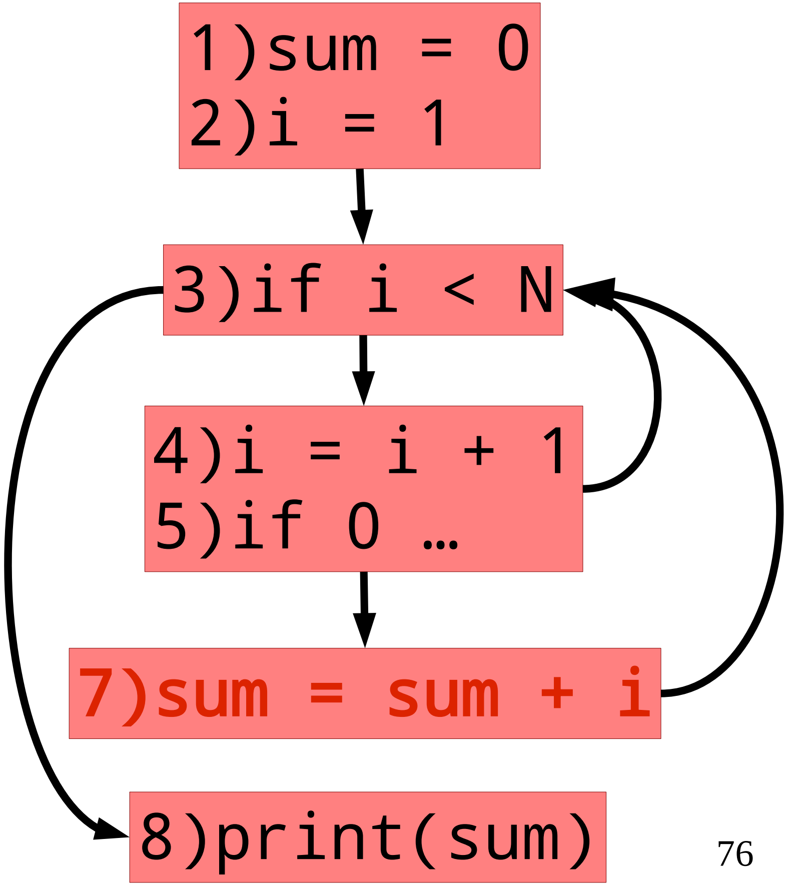


Control Dependence

- 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)  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

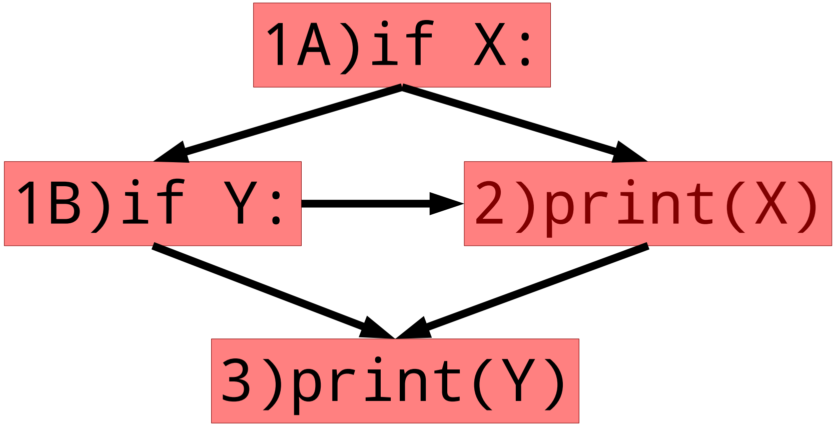
```
1)if X or Y:
2)  print(X)
3)print(Y)
```

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*

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?

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?

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?
- …

3)Program Dependence Graph(PDG)

The PDG is the combination of

- The control dependence graph
- The data dependence graph

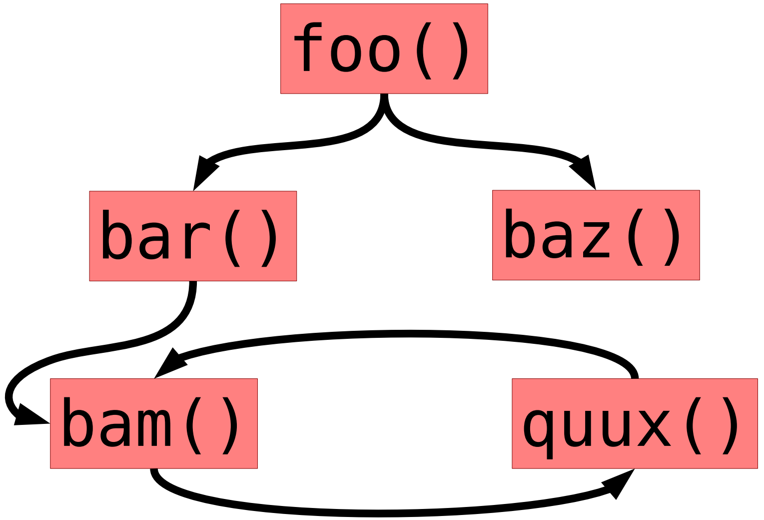
Recall: Edges identify *potential influence*

- **Debuggin**
- **Security:**
- **Testing:** How can I reach a statement?
- …

Can you see *challenges* that may arise when using the PDG in practice?

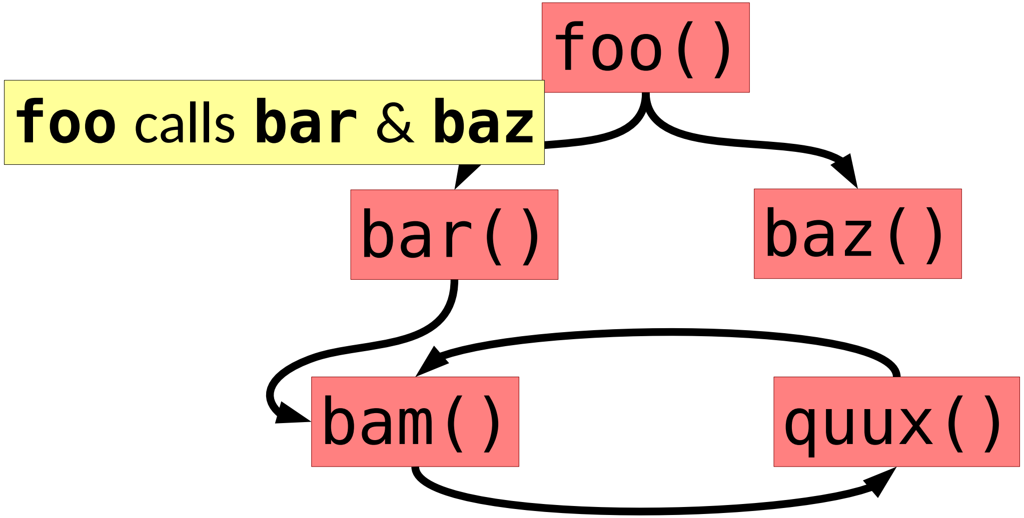
4) Call Graph (Multigraph)

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



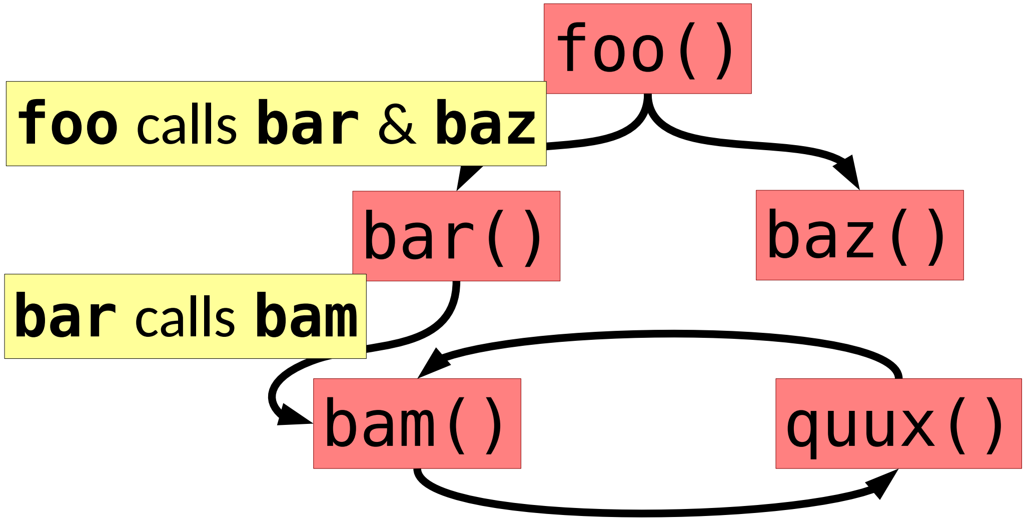
4) Call Graph (Multigraph)

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



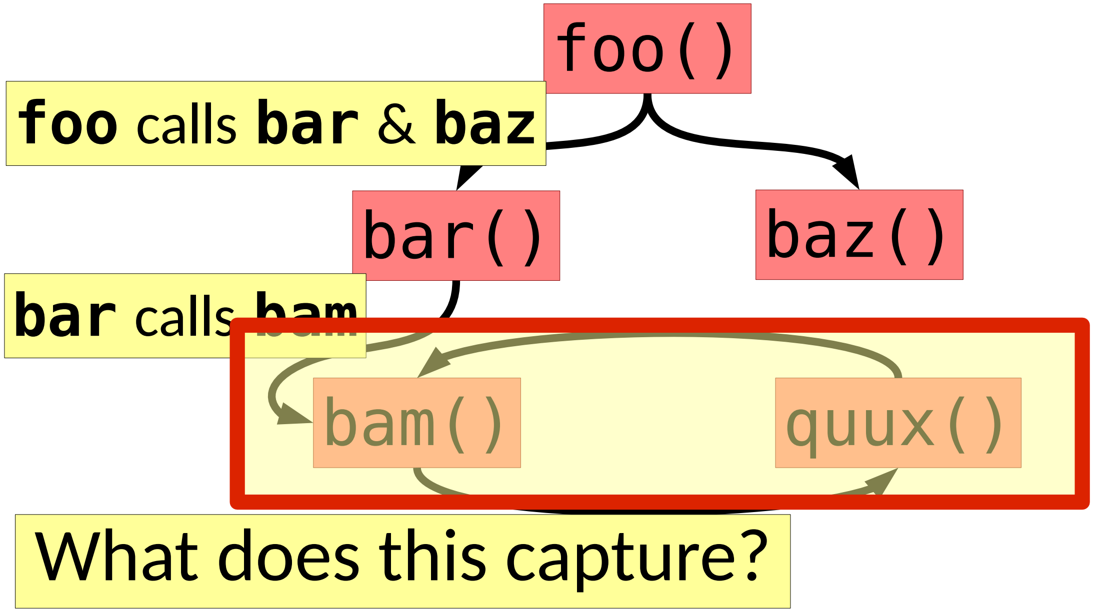
4) Call Graph (Multigraph)

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



4) Call Graph (Multigraph)

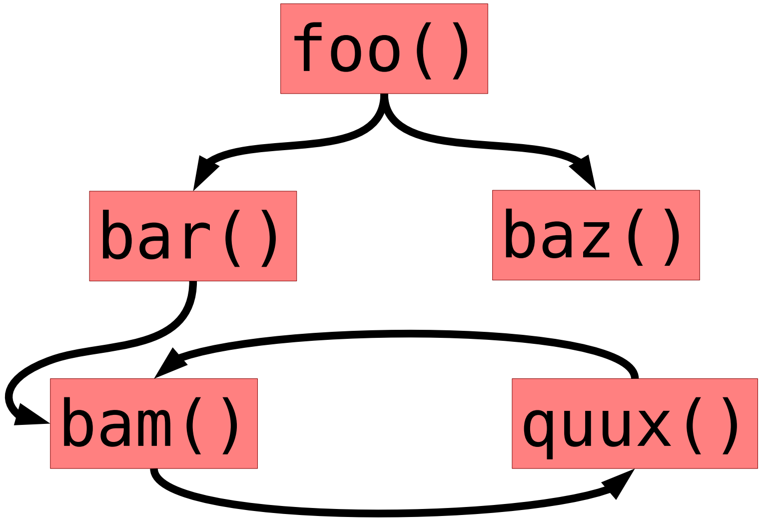
- Captures the composition of a program
  - Nodes are functions
  - Edges show possible calls



4) Call Graph (Multigraph)

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

How should we handle function pointers?



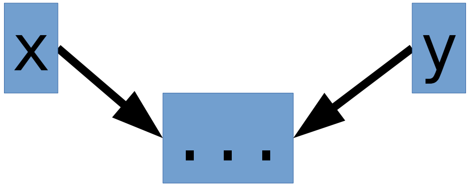
5) Points-to Graphs

Pointers / indirection create two difficult problems:

5) Points-to Graphs

Pointers / indirection create two difficult problems:

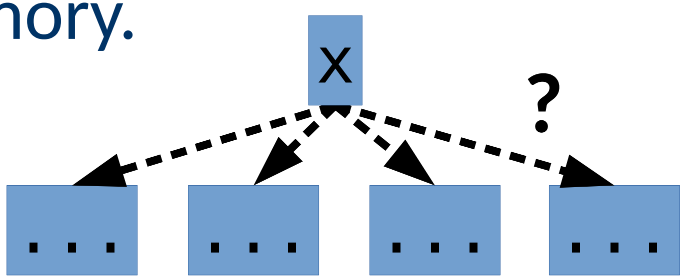
- Aliasing
  - Multiple variables may denote the same memory location



5) Points-to Graphs

Pointers / indirection create two difficult problems:

- Aliasing
  - Multiple variables may denote the same memory location
- Ambiguity
  - One variable may potentially denote several different targets in memory.



5) Points-to Graphs

Pointers / indirection create two difficult problems:

- Aliasing
  - Multiple variables may denote the same memory location
- Ambiguity
  - One variable may potentially denote several different targets in memory.

```
x.lock()
…
y.unlock()
```

5) Points-to Graphs

Pointers / indirection create two difficult problems:

- Aliasing
  - Multiple variables may denote the same memory location
- Ambiguity
  - One variable may potentially denote several different targets in memory.

```
x.lock()
…
y.unlock()
```

```
x = password
…
broadcast(y)
```

5) Points-to Graphs

Points-to graphs capture this points-to relation

5) Points-to Graphs

Points-to graphs capture this points-to relation

- The relation (p,x) where p MAY/MUST point to x

5) Points-to Graphs

Points-to graphs capture this points-to relation

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

5) Points-to Graphs

Points-to graphs capture this points-to relation

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

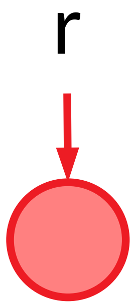
```
1) r = C()
2) p.f = r
3) t = C()
4) if …:
5)   q = p
6) r.f = t
```

5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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
```

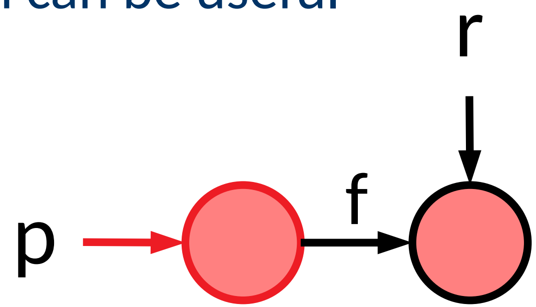


5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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
```

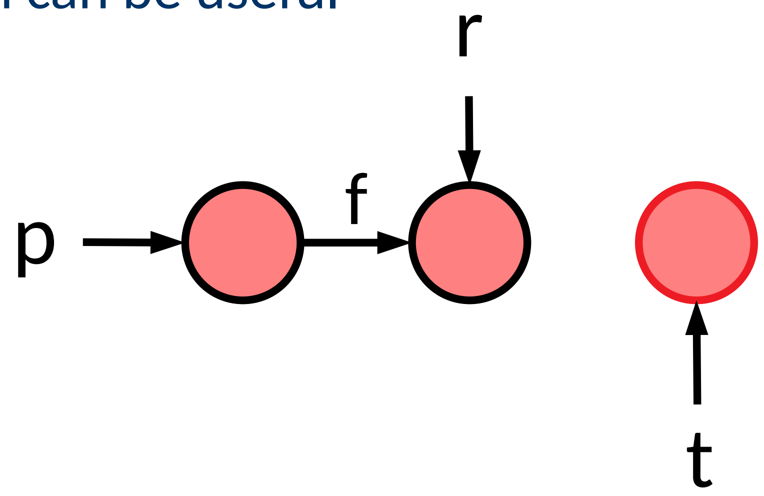


5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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
```

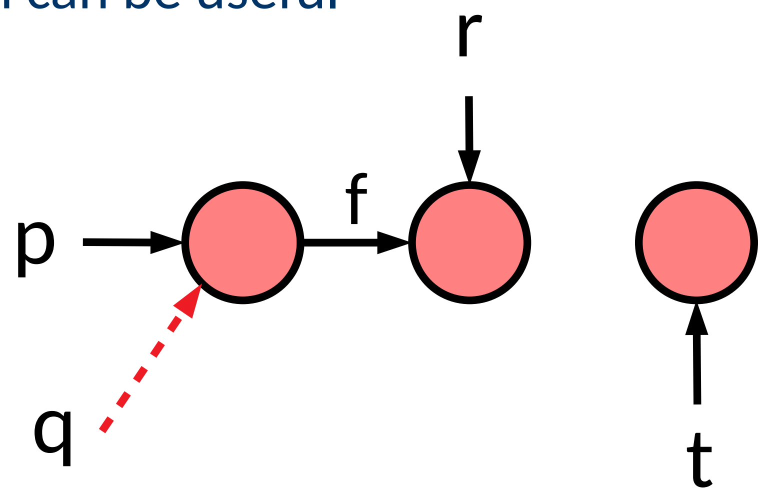


5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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
```

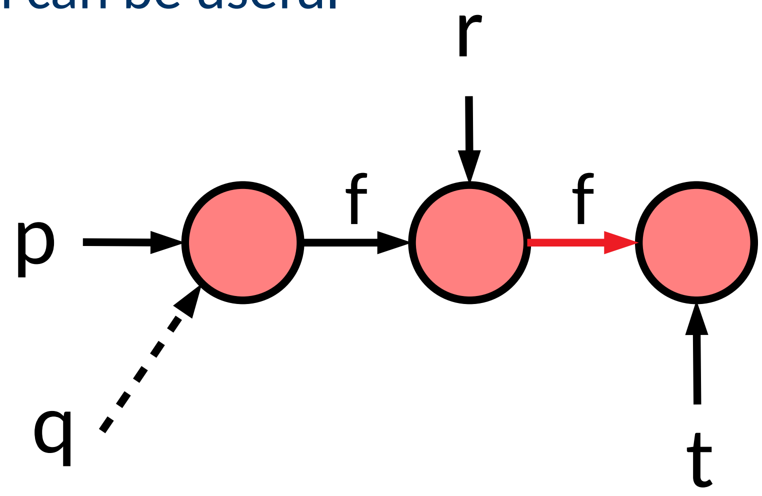


5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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
```

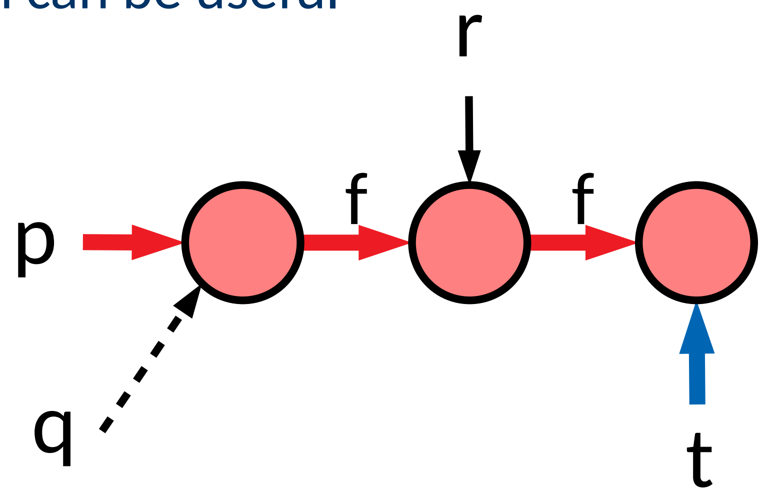


5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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
```



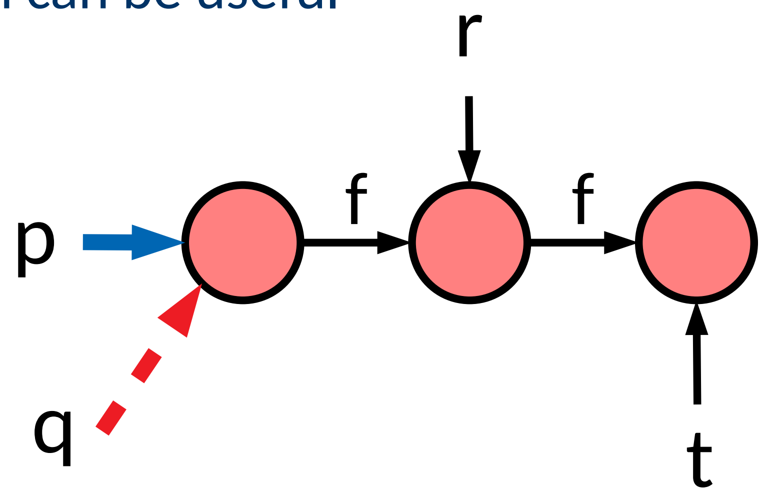
p.f.f MUST ALIAS t

5) Points-to Graphs

Points-to graphs capture this points-to relation

- 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

6) Emerging Representations for ML

- Machine learning is seen as a value driver for many tasks,
  but using it effectively to reason about software is still challenging

6) Emerging Representations for ML

- Machine learning is seen as a value driver for many tasks, but using it effectively to reason about software is still challenging
- Trying simple models should always be considered first e.g. simple feed forward networks can work better [Yedida 2021]

6) Emerging Representations for ML

- Machine learning is seen as a value driver for many tasks, but using it effectively to reason about software is still challenging
- Trying simple models should always be considered first
  - Bug fix & close time estimation [Yedida 2021]
  - Project planning & analytics [Krishna 2020]
  - Recognizing actionable compiler warnings [Yang 2020]

6) Emerging Representations for ML

- Machine learning is seen as a value driver for many tasks,
  but using it effectively to reason about software is still challenging
- Trying simple models should always be considered first
- Observe:
  Many engineering tasks require *discrete* & *symbolic* reasoning.
  - ML is classically better on non symbolic problems.

6) Emerging Representations for ML

- Machine learning is seen as a value driver for many tasks, but using it effectively to reason about software is still challenging
- Trying simple models should always be considered first
- Observe:
  Many engineering tasks require discrete & symbolic reasoning.
  - ML is classically better on non symbolic problems.
  - Bridging the gap is an area of open research (neurosymbolic, ...)

6) Emerging Representations for ML

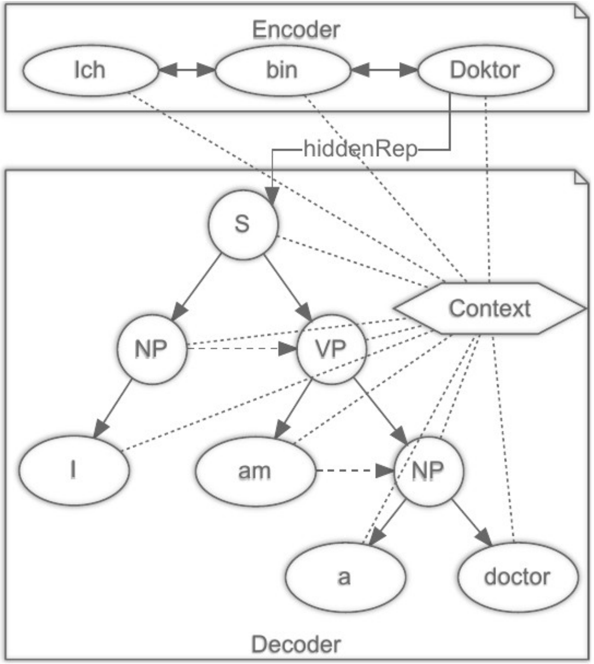
- Machine learning is seen as a value driver for many tasks,
  but using it effectively to reason about software is still challenging
- Trying simple models should always be considered first
- Observe:
  Many engineering tasks require discrete & symbolic reasoning.
  - ML is classically better on non symbolic problems.
  - Bridging the gap is an area of open research (neurosymbolic, ...)
  - Solutions that do not require *a priori* implementation are desirable

6) Emerging Representations for ML

- Machine learning is seen as a value driver for many tasks,
  but using it effectively to reason about software is still challenging
- Trying simple models should always be considered first
- Observe:
  Many engineering tasks require discrete & symbolic reasoning.
  - ML is classically better on non symbolic problems.
  - Bridging the gap is an area of open research (neurosymbolic, ...)
  - Solutions that do not require *a priori* implementation are desirable
- But different models & pipelines arise to aid in
  reasoning about software

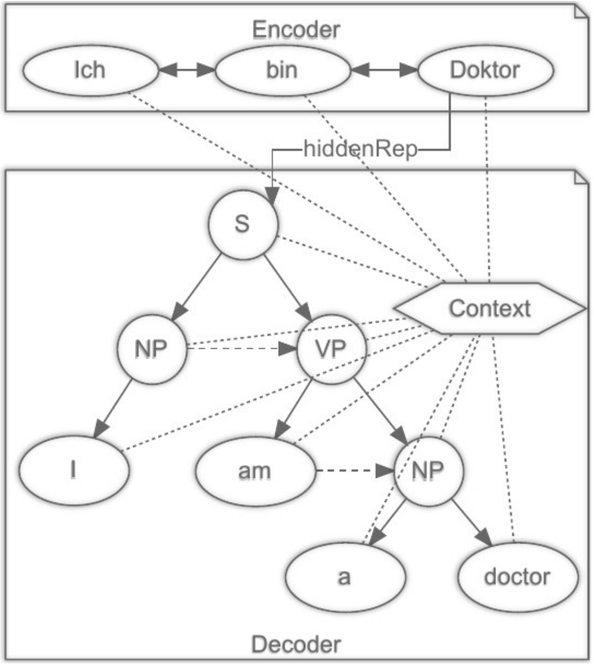
6) Emerging Representations for ML

Seq2DRNN
Encoder-Decoders
[Alvarez-Melis 2017, Gu 2019]

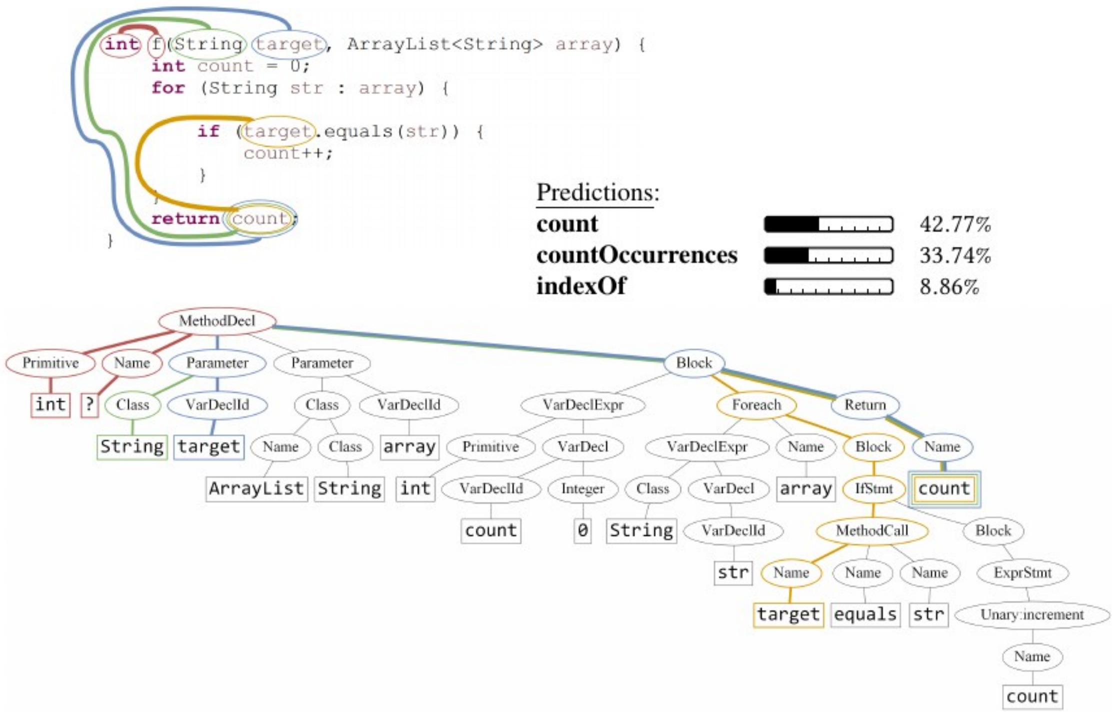


6) Emerging Representations for ML

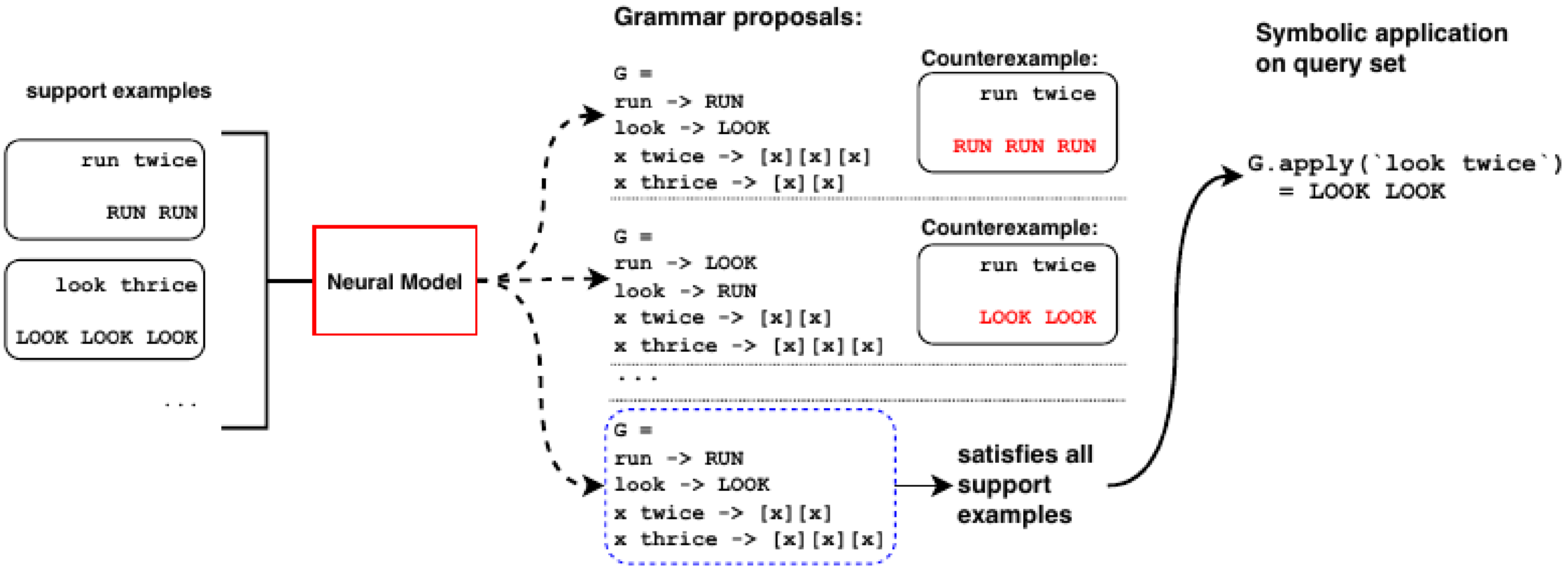
Seq2DRNN
Encoder-Decoders
[Alvarez-Melis 2017, Gu 2019]



code2vec
[Alon 2018]



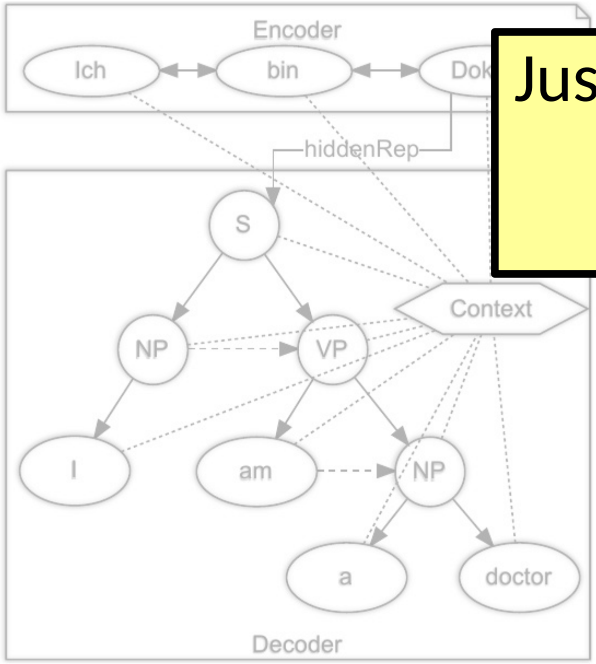
6) Emerging Representations for ML



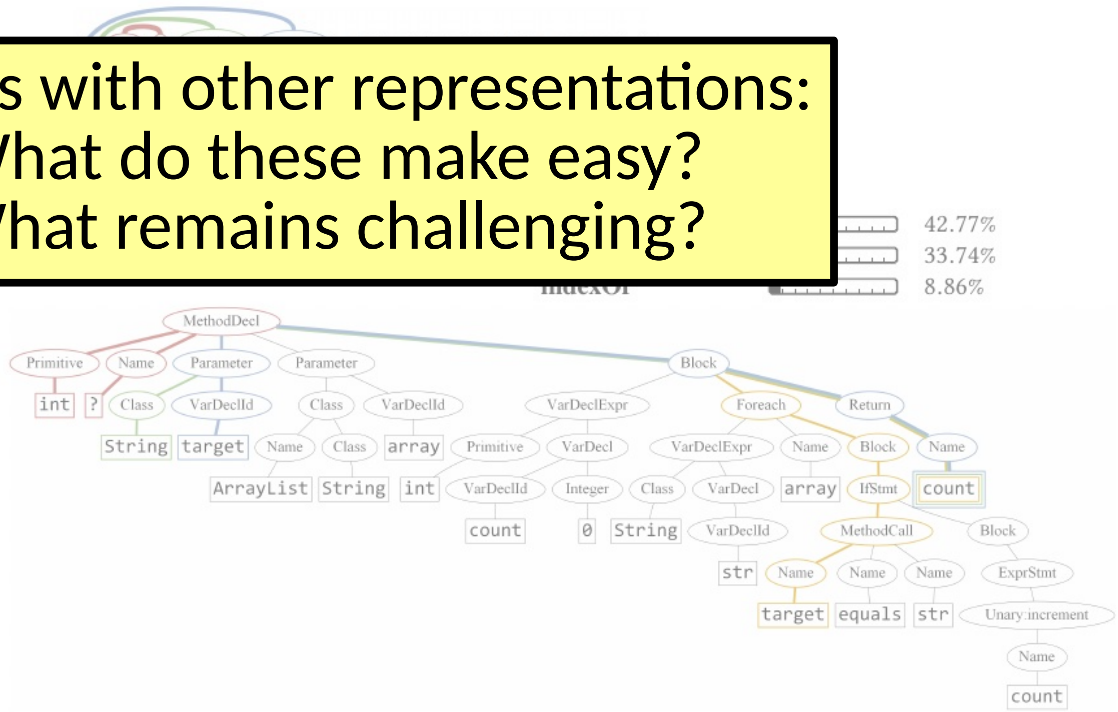
neurosymbolic models for synthesis
[Nye 2020]

6) Emerging Representations for ML

Seq2DRNN
Encoder-Decoders
[Alvarez-Melis 2017, Gu 2019]



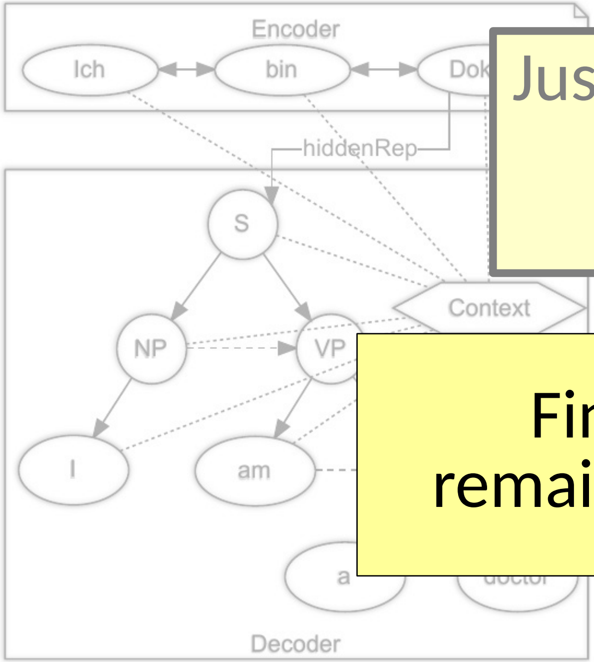
code2vec
[Alon 2018]



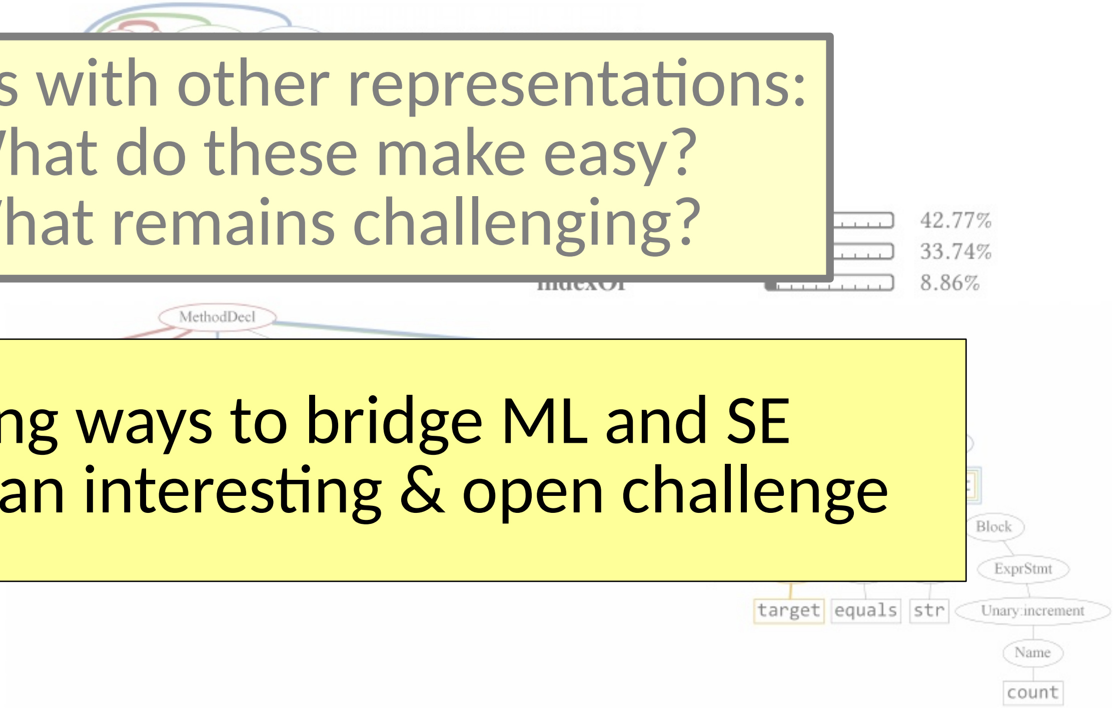
Just as with other representations:
What do these make easy?
What remains challenging?

6) Emerging Representations for ML

Seq2DRNN
Encoder-Decoders
[Alvarez-Melis 2017, Gu 2019]



code2vec
[Alon 2018]



Just as with other representations:
What do these make easy?
What remains challenging?

Finding ways to bridge ML and SE
remains an interesting & open challenge

Representing Program Executions

Execution Representations

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

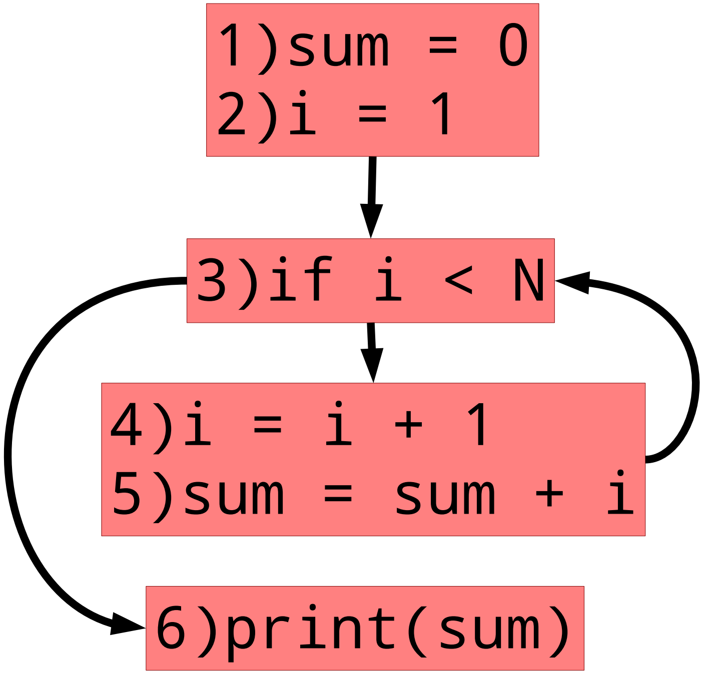
Execution Representations

- ***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

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

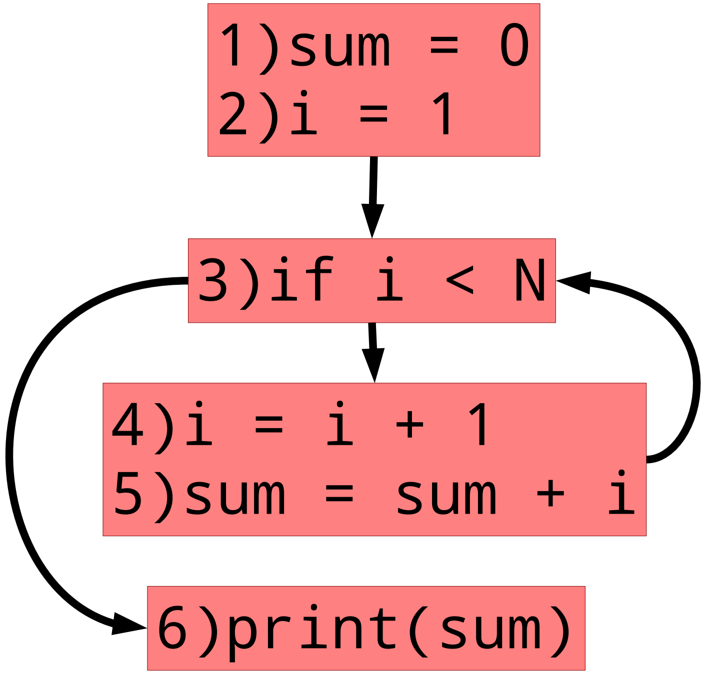


$1_1\ 2_1\ 3_1\ 4_1\ 5_1\ 3_2\ 4_2\ 5_2\ 3_3\ 6_1$

All Equivalent

```
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



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

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

} All Equivalent

```
1)sum = 0
2)i = 1
```

```
3)if i < N
```

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

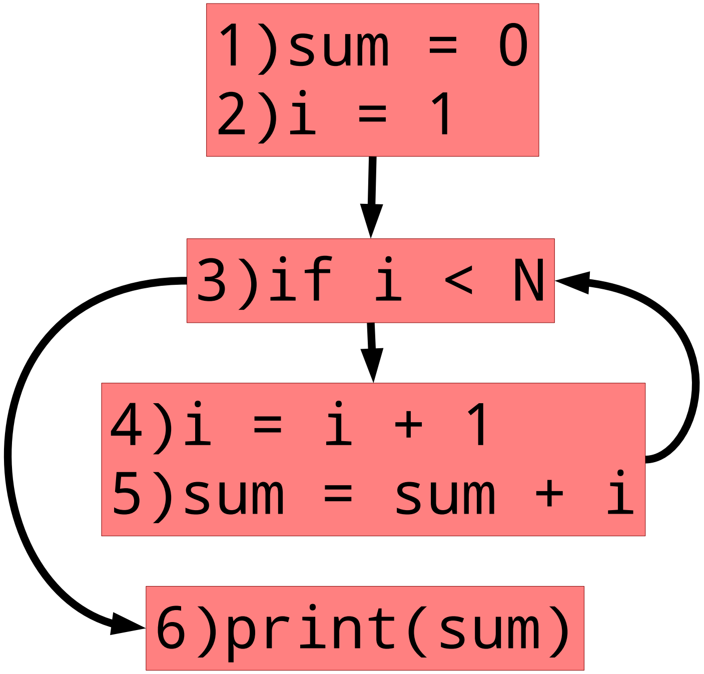
```
3)if i < N
```

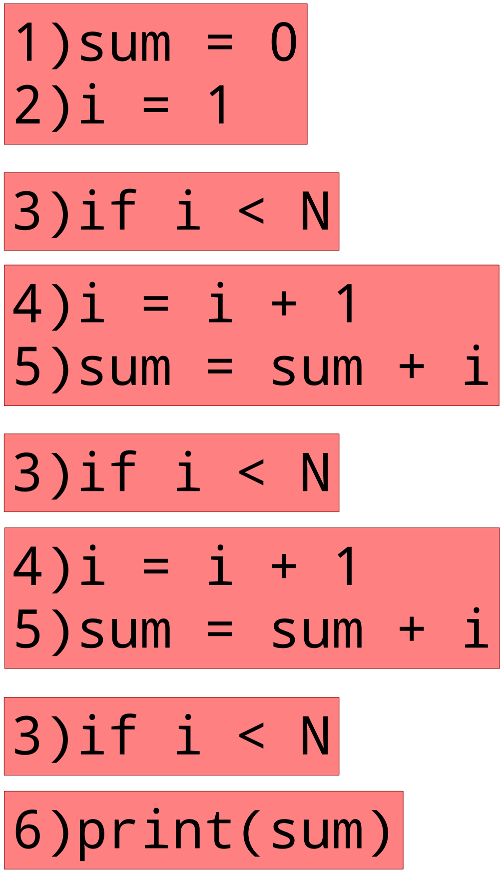
```
4)i = i + 1
5)sum = sum + i
```

```
3)if i < N
```

```
6)print(sum)
```

Control Flow Trace





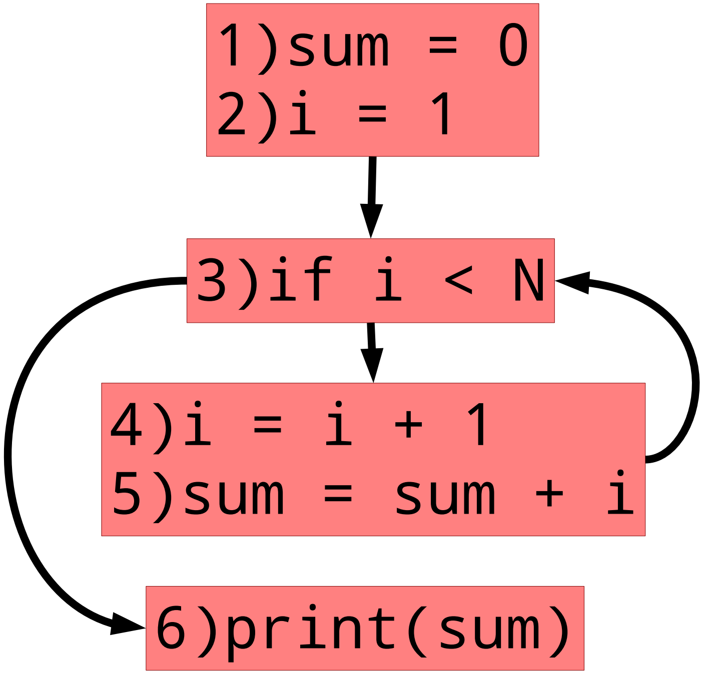
$1_1\ 2_1\ 3_1\ 4_1\ 5_1\ 3_2\ 4_2\ 5_2\ 3_3\ 6_1$

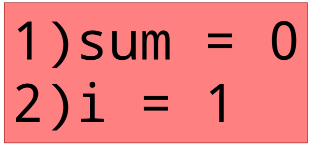
$1_1\ 3_1\ 4_1\ 3_2\ 4_2\ 3_3\ 6_1$

TTF

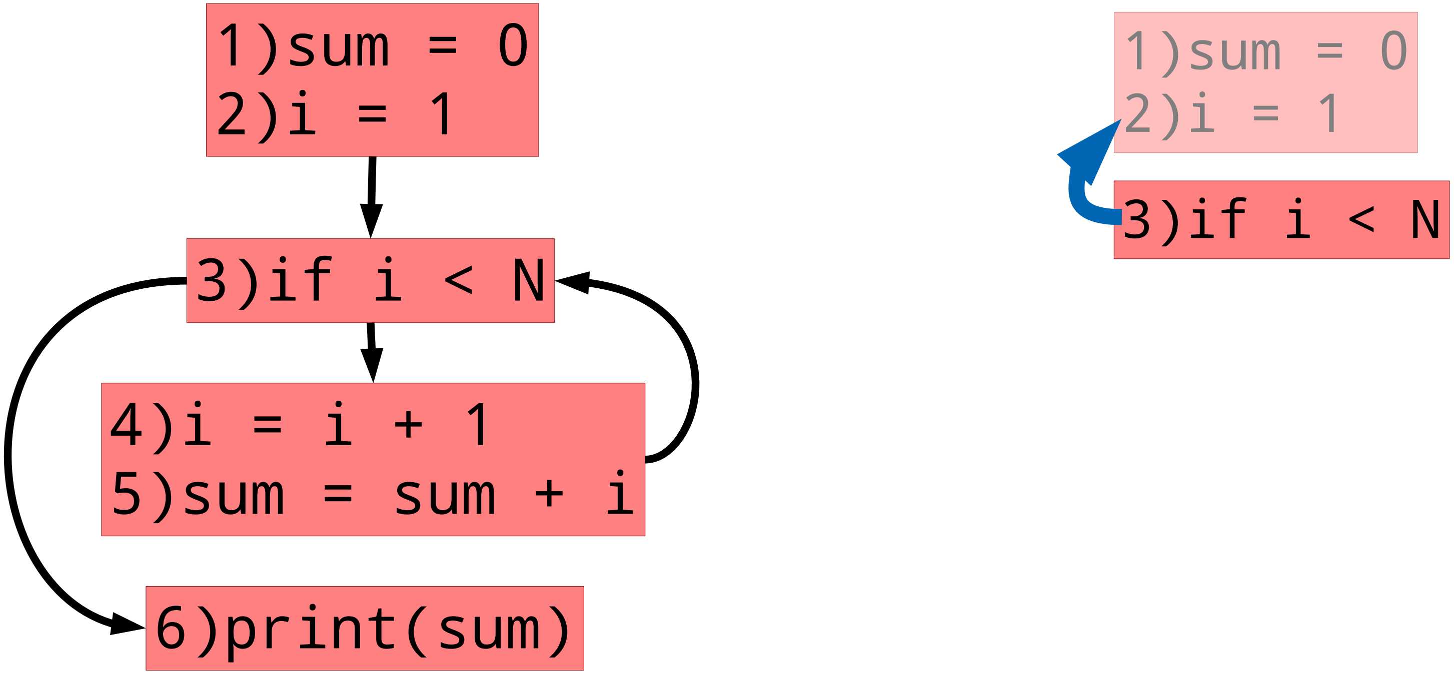
} All Equivalent

Dynamic Dependence Graph

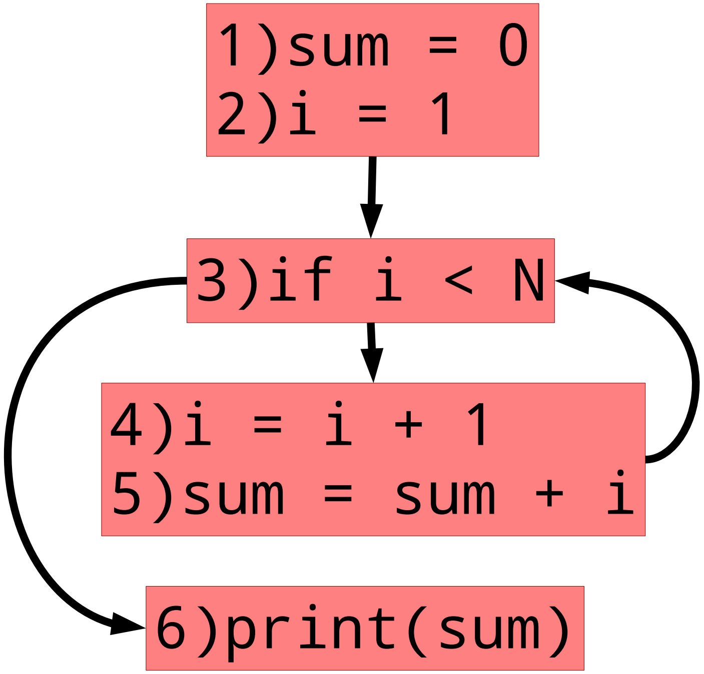


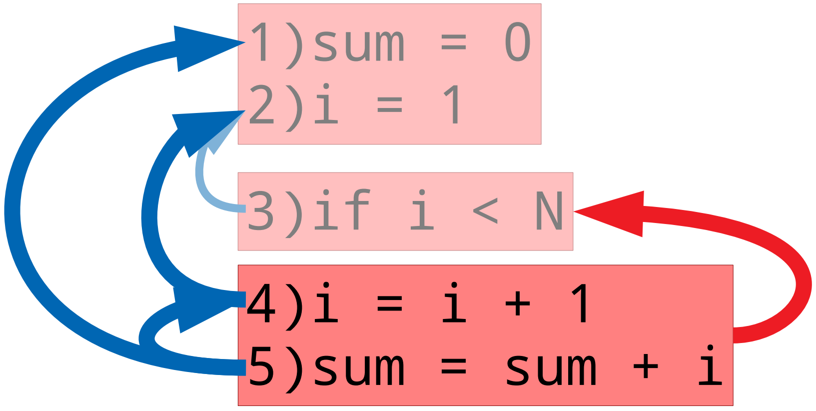


Dynamic Dependence Graph

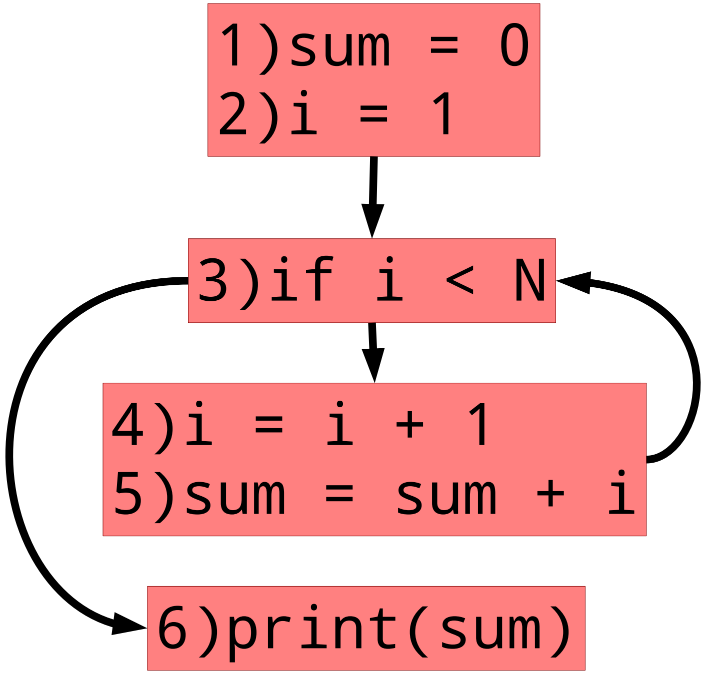


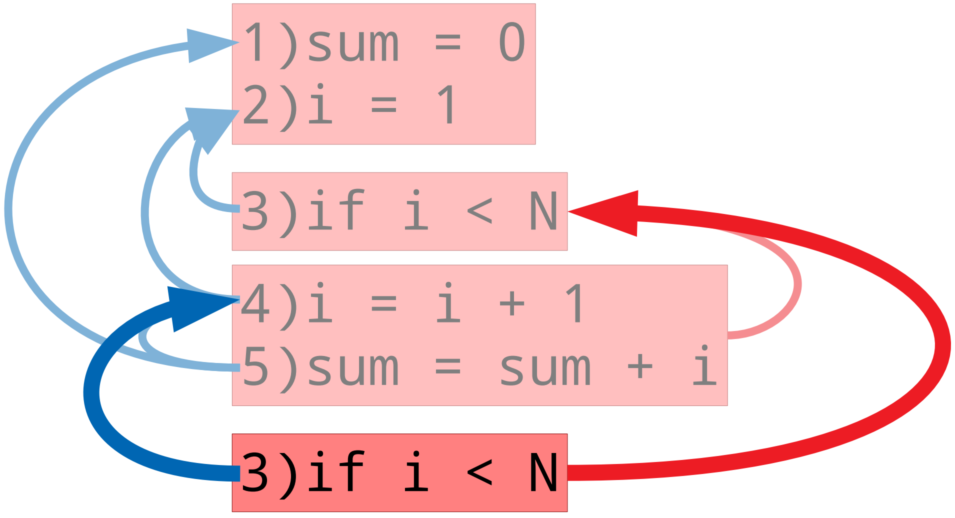
Dynamic Dependence Graph



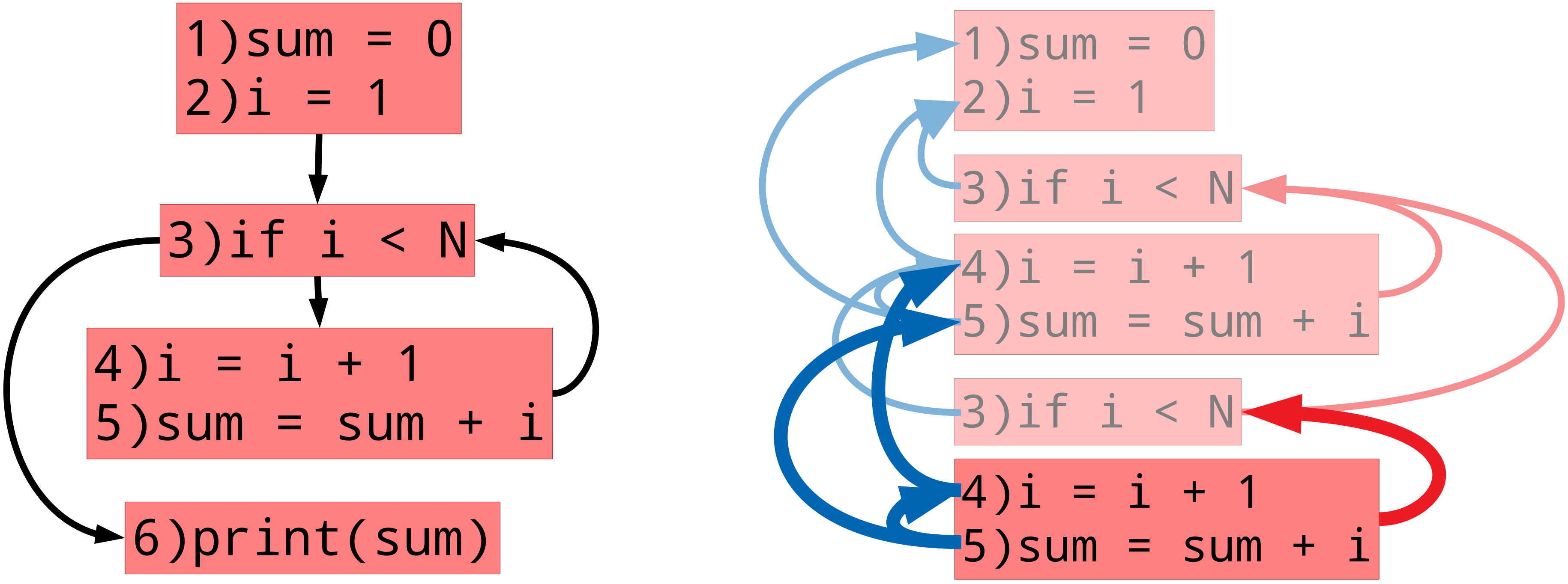


Dynamic Dependence Graph

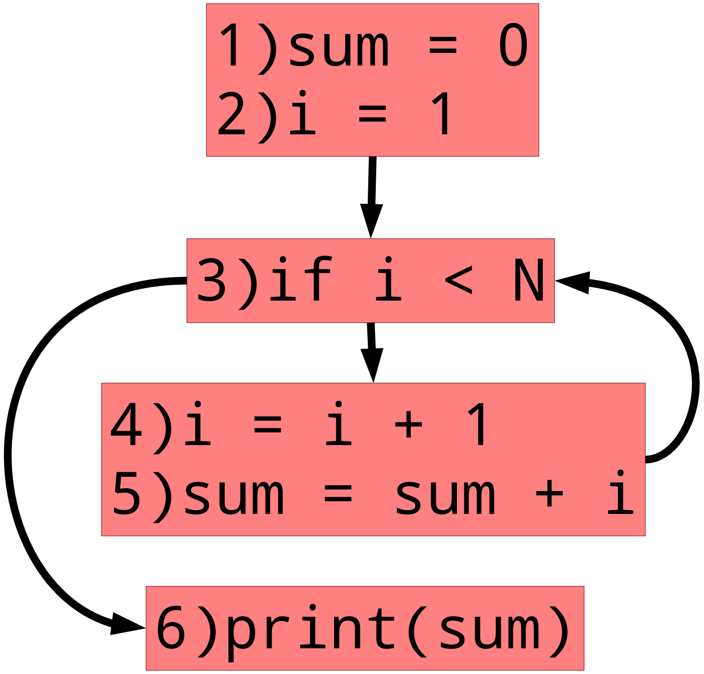


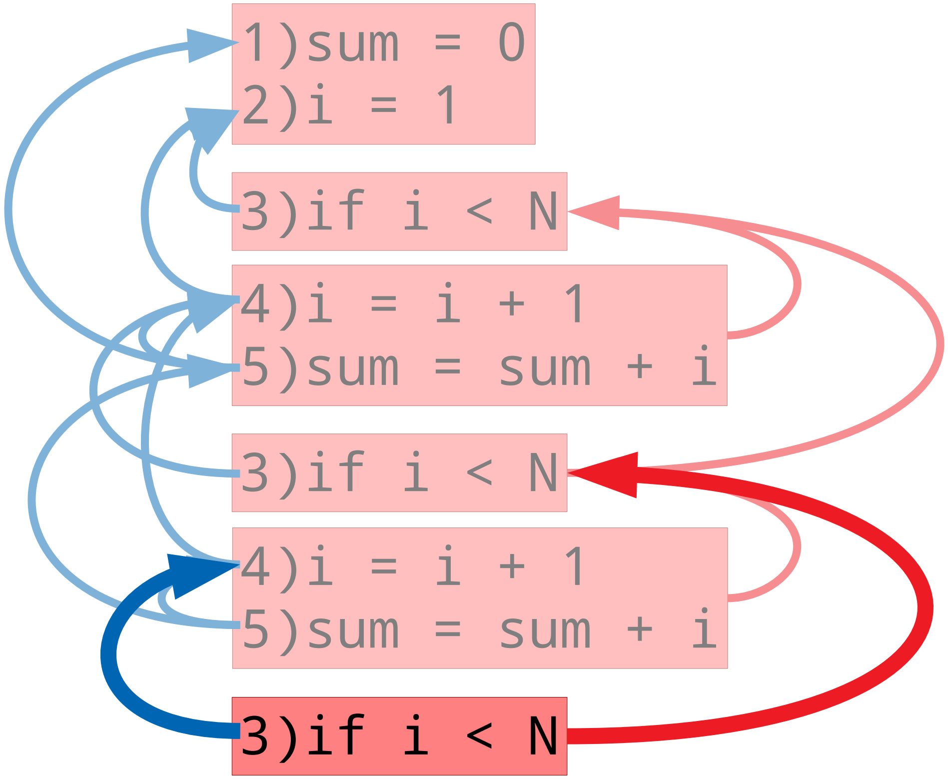


Dynamic Dependence Graph

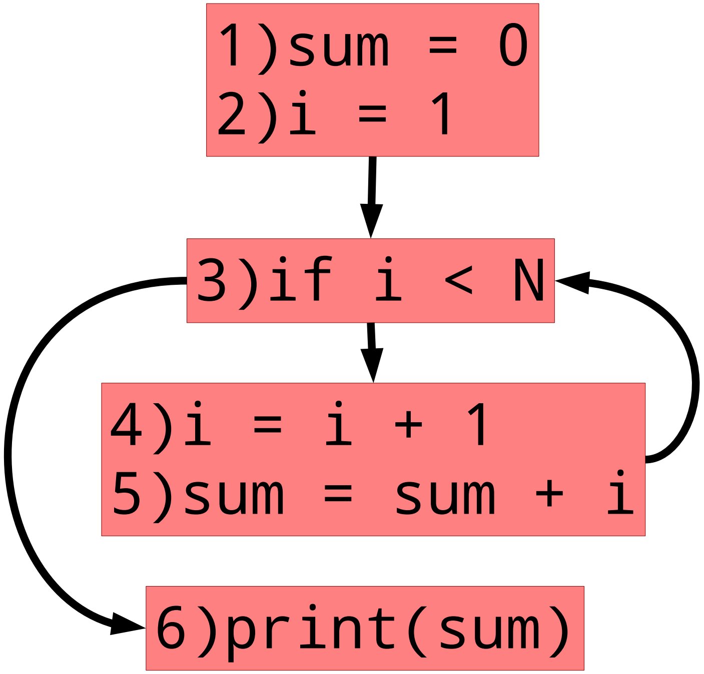


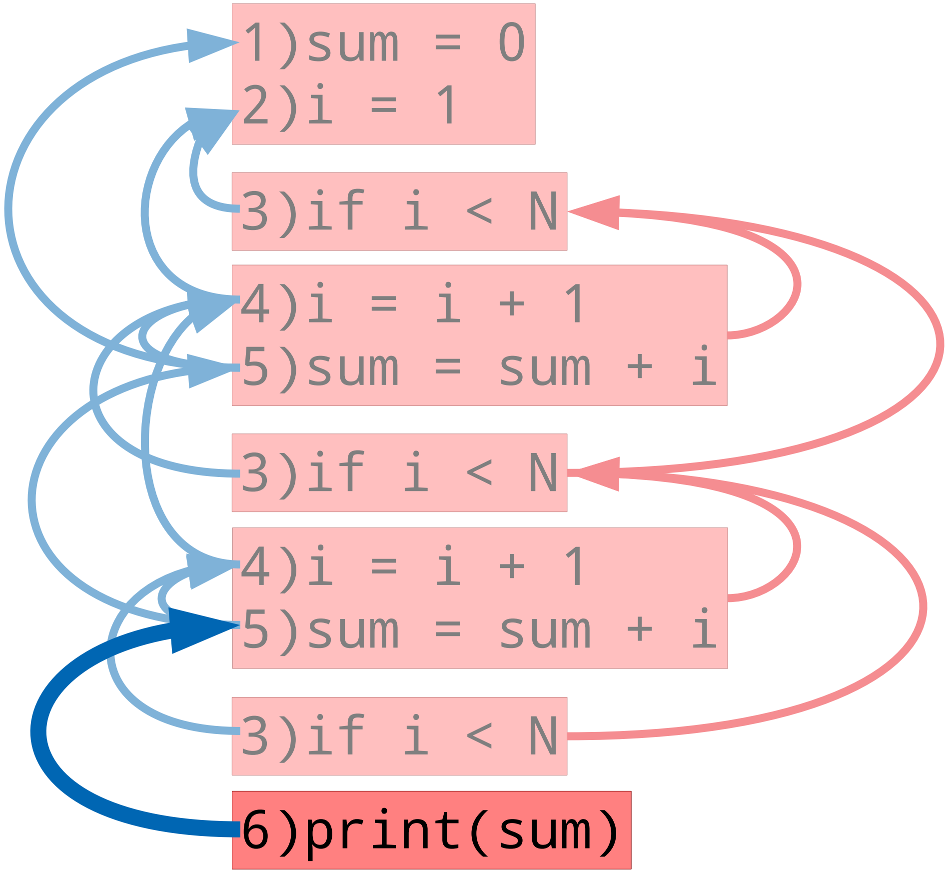
Dynamic Dependence Graph



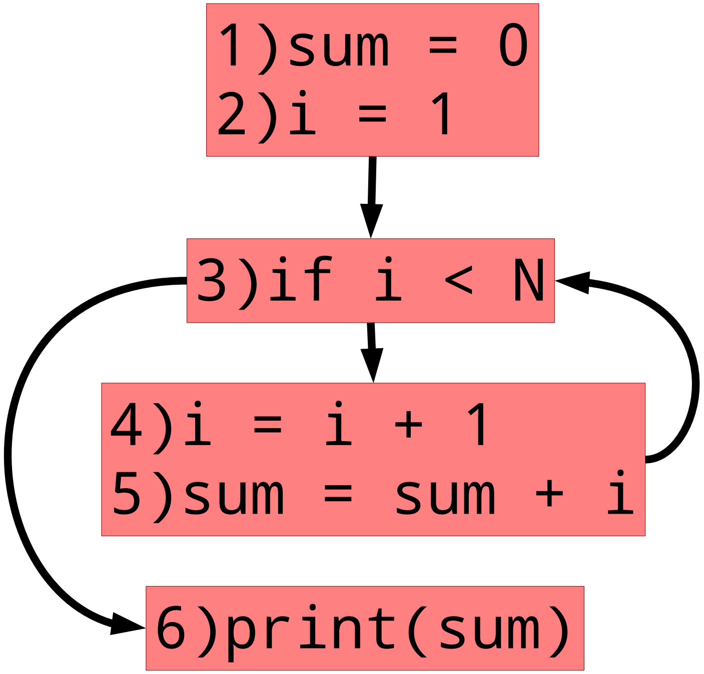


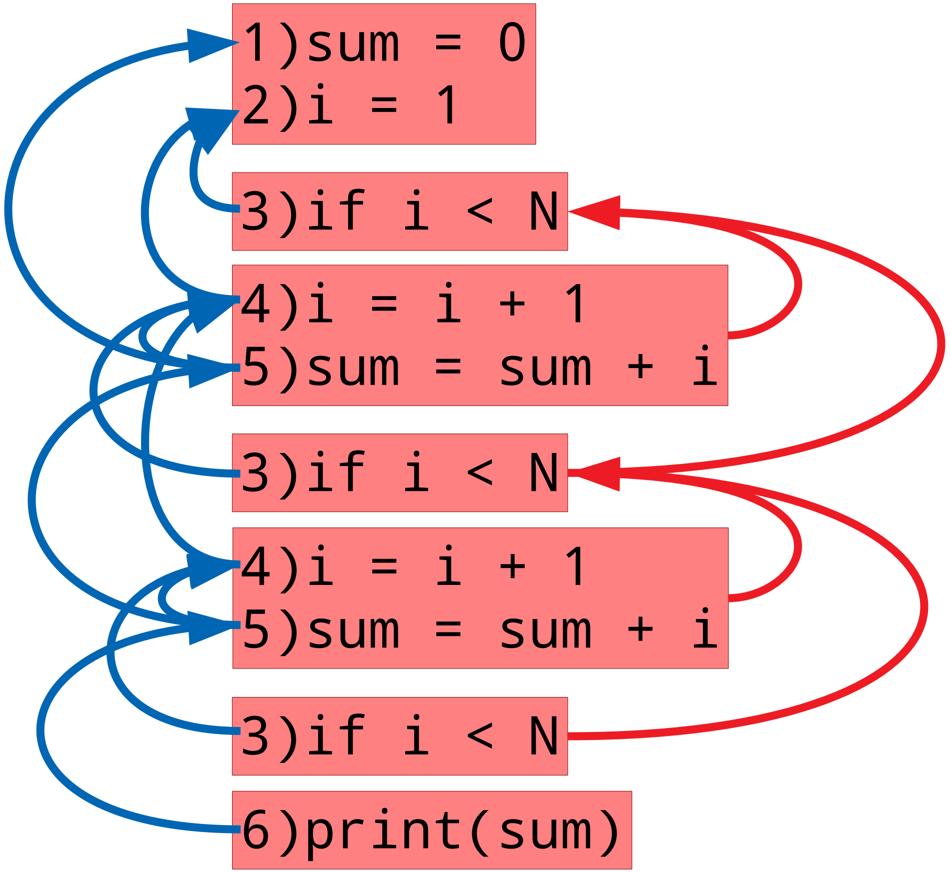
Dynamic Dependence Graph



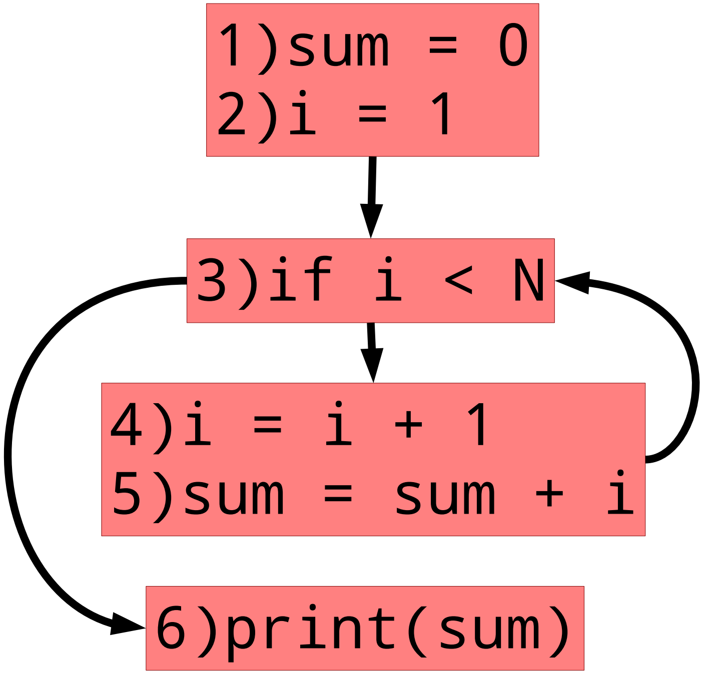


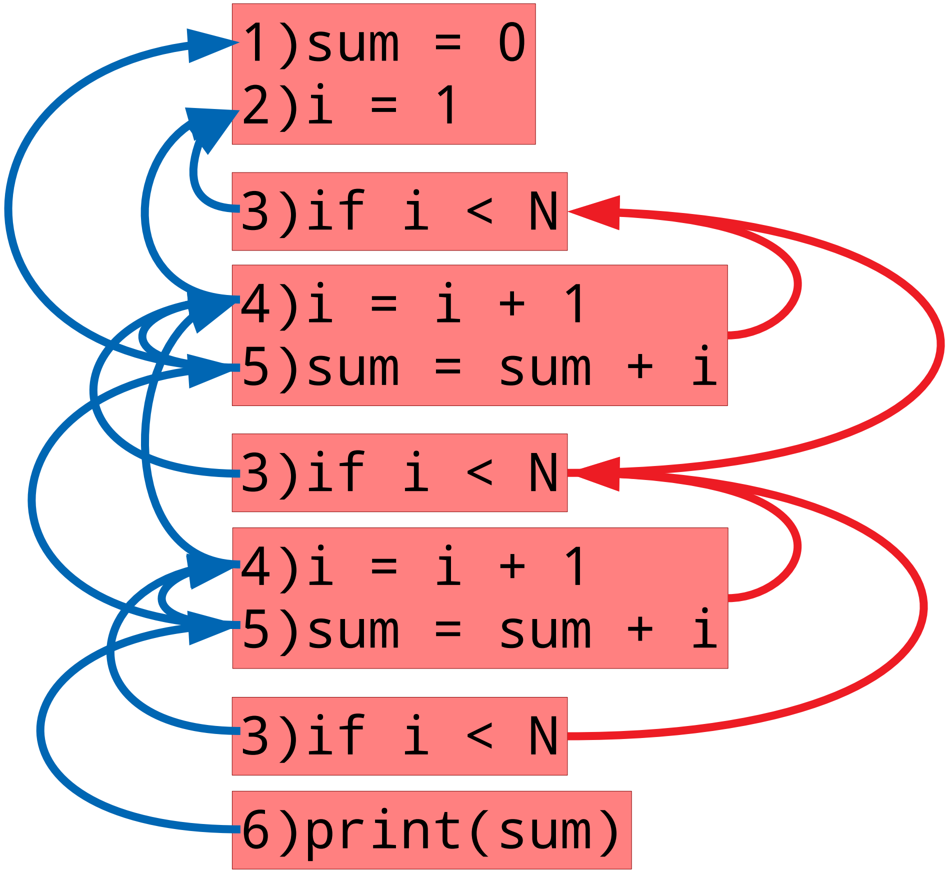
Dynamic Dependence Graph





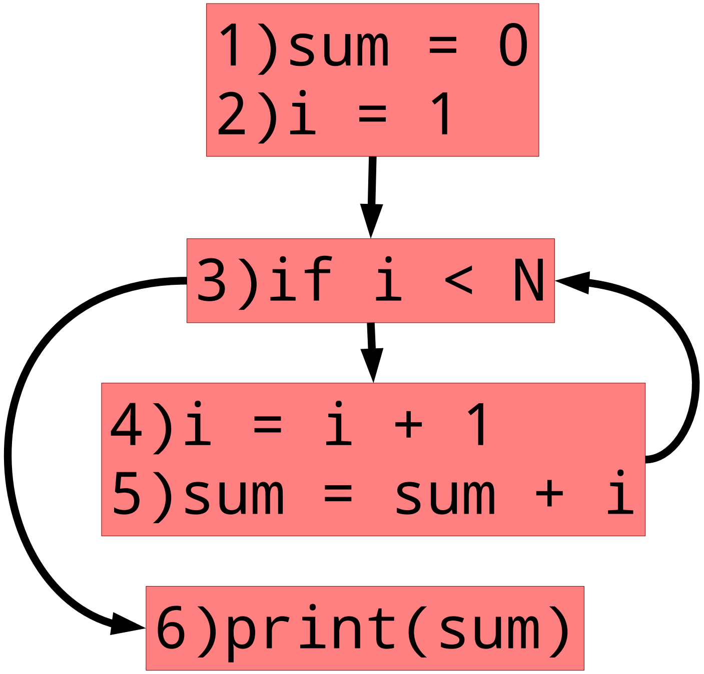
Dynamic Dependence Graph

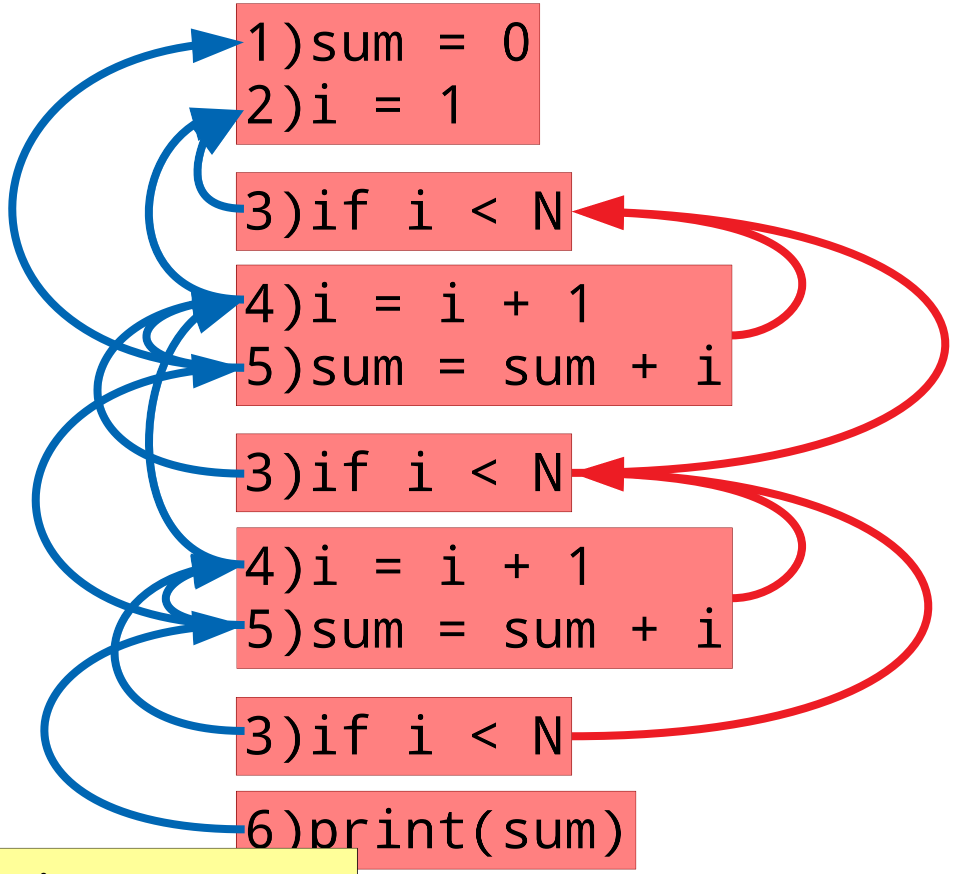




Notably a *bit* difficult for people to wade through.

Dynamic Dependence Graph

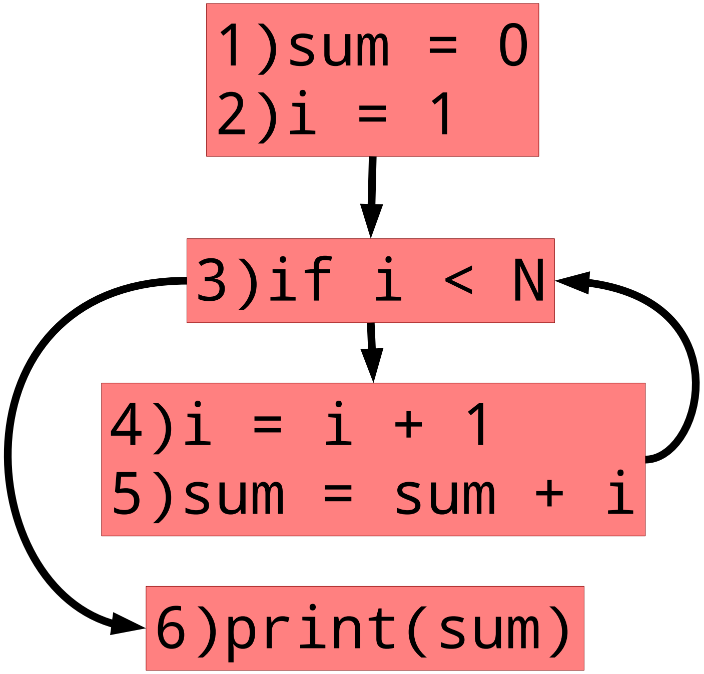


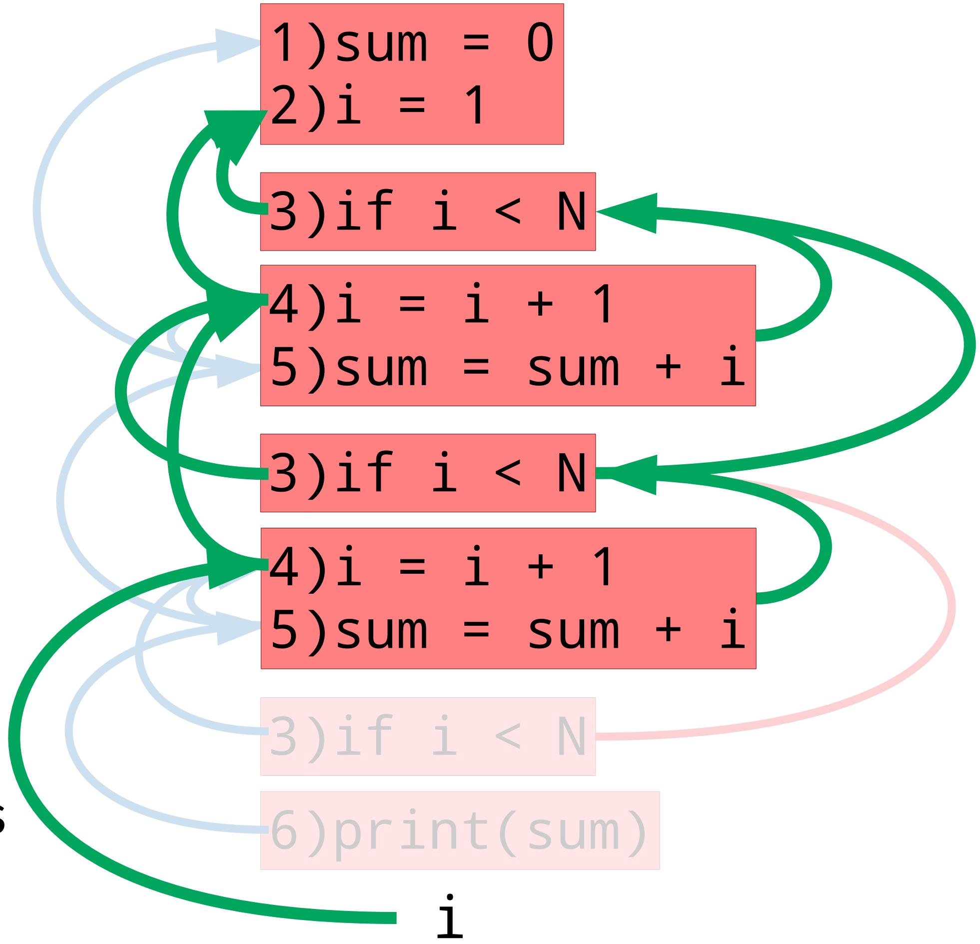


Notably a *bit* difficult for people to wade through.

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

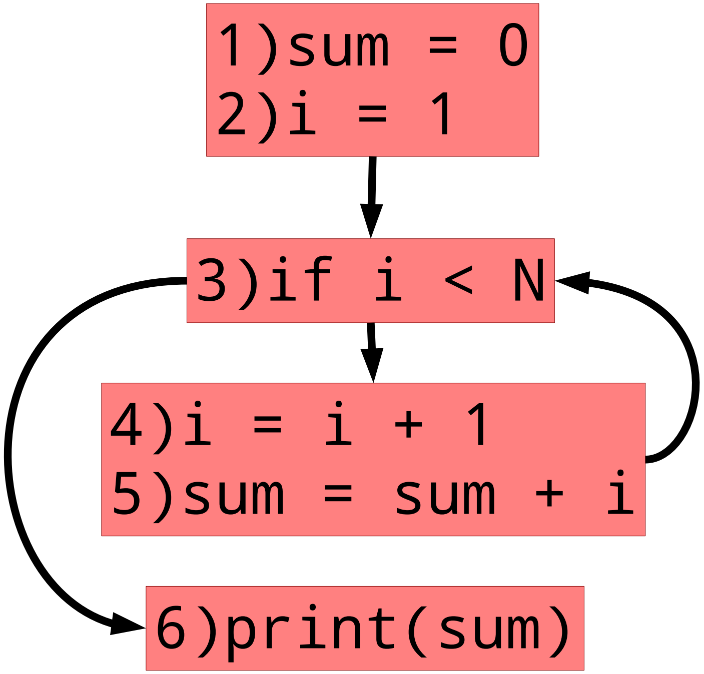
Dynamic Dependence Graph

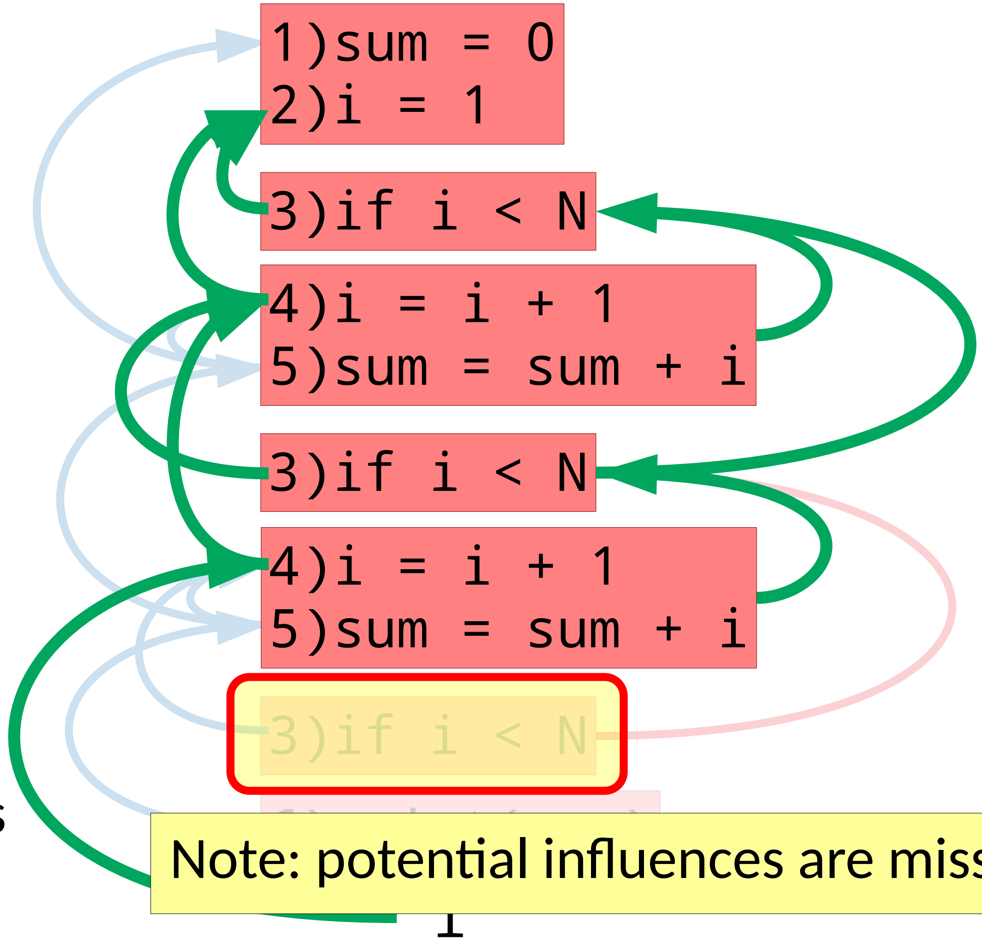




Slicing (static or dynamic) computes a transitive closure of dependences

Dynamic Dependence Graph





Slicing (static or dynamic) computes a transitive closure of dependences

Note: potential influences are missed!

Dynamic Dependence Graphs

Capture a notion of ***observed*** influence

Dynamic Dependence Graphs

Capture a notion of ***observed*** influence

- **Debugging:** What caused a bug?

Dynamic Dependence Graphs

Capture a notion of ***observed*** influence

- **Debugging:** What caused a bug?
- **Security:** How did sensitive information leak?

Dynamic Dependence Graphs

Capture a notion of ***observed*** influence

- **Debugging:** What caused a bug?
- **Security:** How did sensitive information leak?
- **Testing:** What tests need to be run based on a change?
- …

Dynamic Dependence Graphs

Capture a notion of ***observed*** influence

- **Debugging:** What caused a bug?
- **Security:** How did sensitive information leak?
- **Testing:** What tests need to be run based on a change?
- …

Prioritizing, pruning, & bundling information is often critical when applying slicing

Summary

- Different tasks may benefit from representing programs in different ways
- Thinking of the right representation for the task you have is important