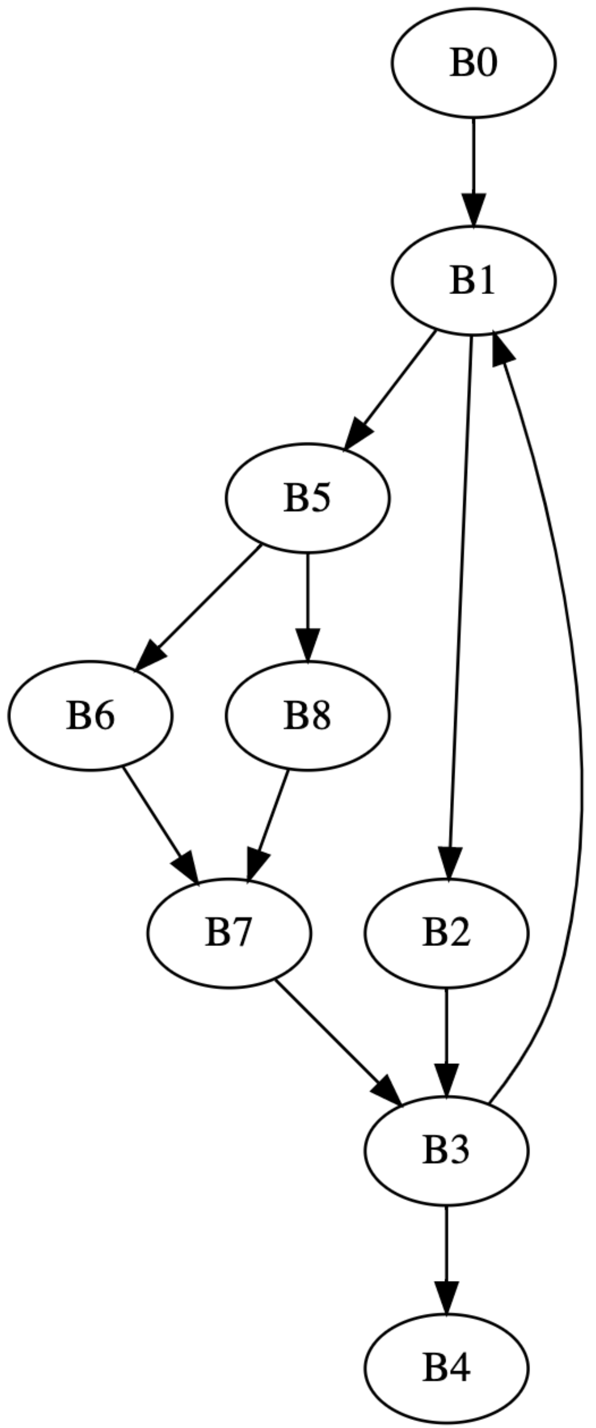
CSE211: Compiler Design

Oct. 25, 2023

- **Topic**: global optimizations

- **Questions**:
  - *What is a control flow graph?*



Announcements

- Homework 1 is due today
  - No extensions
  - Only one person needs to turn it in

- Homework 2 is released today
  - Part 1 is about local value numbering
    - You should have everything you need to do it
  - Part 2 is about live variable analysis
    - It is a global analysis that we will learn about

Announcements

- No office hours this week for me
  - Only (planned) disruption this quarter
  - Visit Rithik during his office hours
  - Ask questions on piazza

Announcements

- Paper review is due on Monday (by midnight)
- Midterm is on Monday
  - In person during class time
  - 10% of grade
  - 3 pages of notes

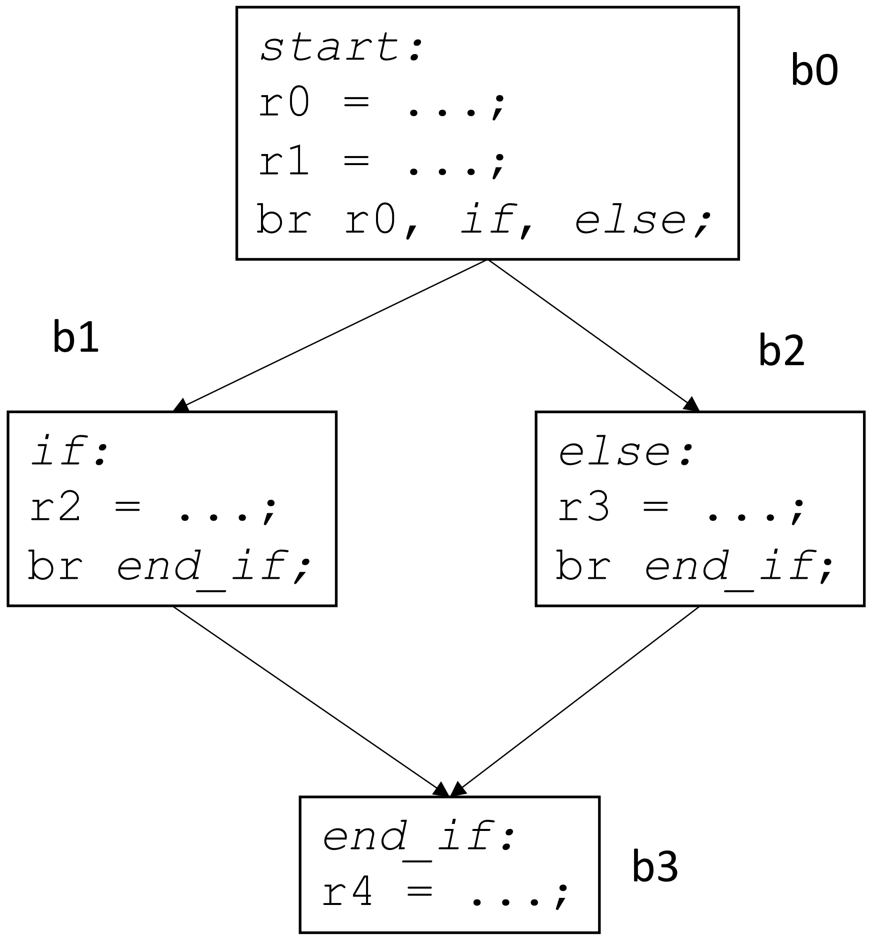
Review

- Regional optimizations:
  - Examples?

Regional optimization: Super local value numbering

- Usually constrained to a "common" subset of the CFG:
- For example: if/else statements

```
b0_H = {
        "..." : "r0",
        "..." : "r1",
}
```

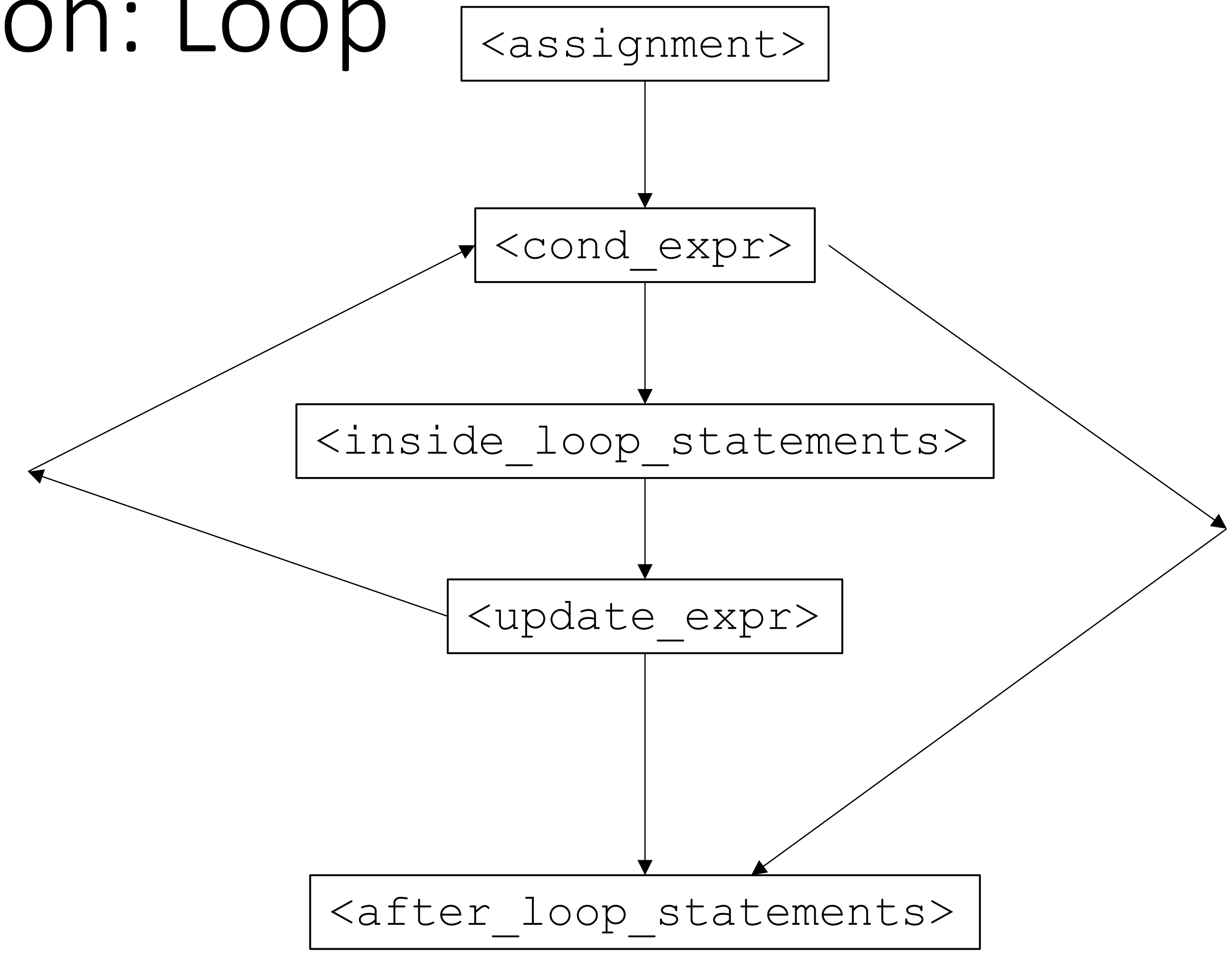


Do local value numbering, but start off with a non-empty hash table!

Which blocks can use which hash tables?

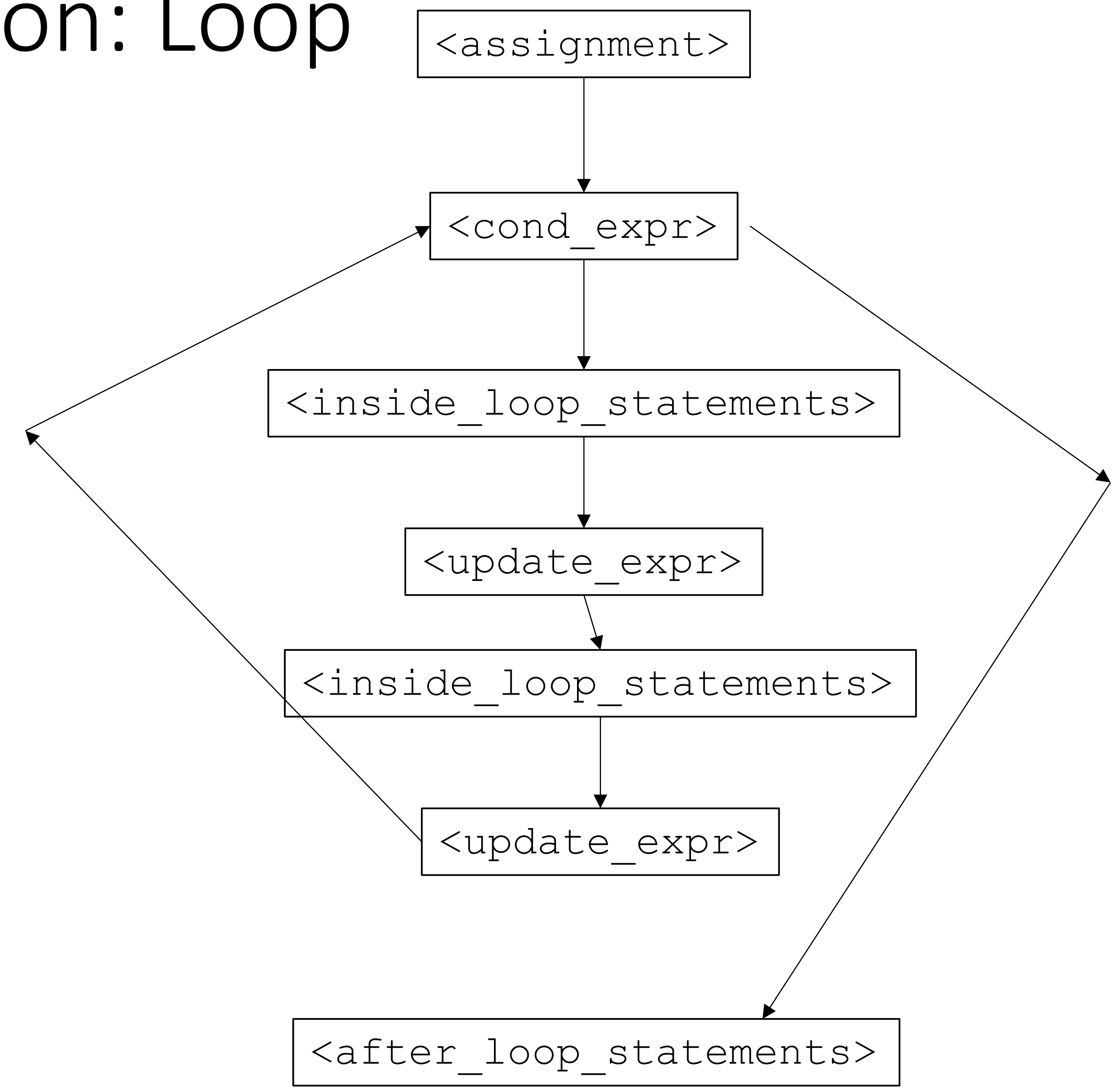
Regional optimization: Loop unrolling:

Assume we
know that the loop will
iterate an even number
of times:



Regional optimization: Loop unrolling:

Assume we
know that the loop will
iterate an even number
of times:



Regional Optimization: Code placement:

- Back to if/else

- Eventually we will straight line the code:

```
start:
r0 = ...;
r1 = ...;
br r0, if, else;
```

```
if:
r2 = ...;
br end_if;
```

```
else:
r3 = ...;
br end_if;
```

```
end_if:
r4 = ...;
br next_lbl
```

```
start:
r0 = ...;
r1 = ...;
br r0, if, else;
```

```
if:
r2 = ...;
```

```
end_if:
r4 = ...;
br next_lbl
```

```
else:
r3 = ...;
br end_if;
```

If we know that one branch is taken more often than the other...
say the branch is true most often

New material

Global optimizations

- Difference between regional:
  - handle arbitrary CFGs, cannot rely on structure!
  - Algorithms become more general
  - Potential for more optimizations!

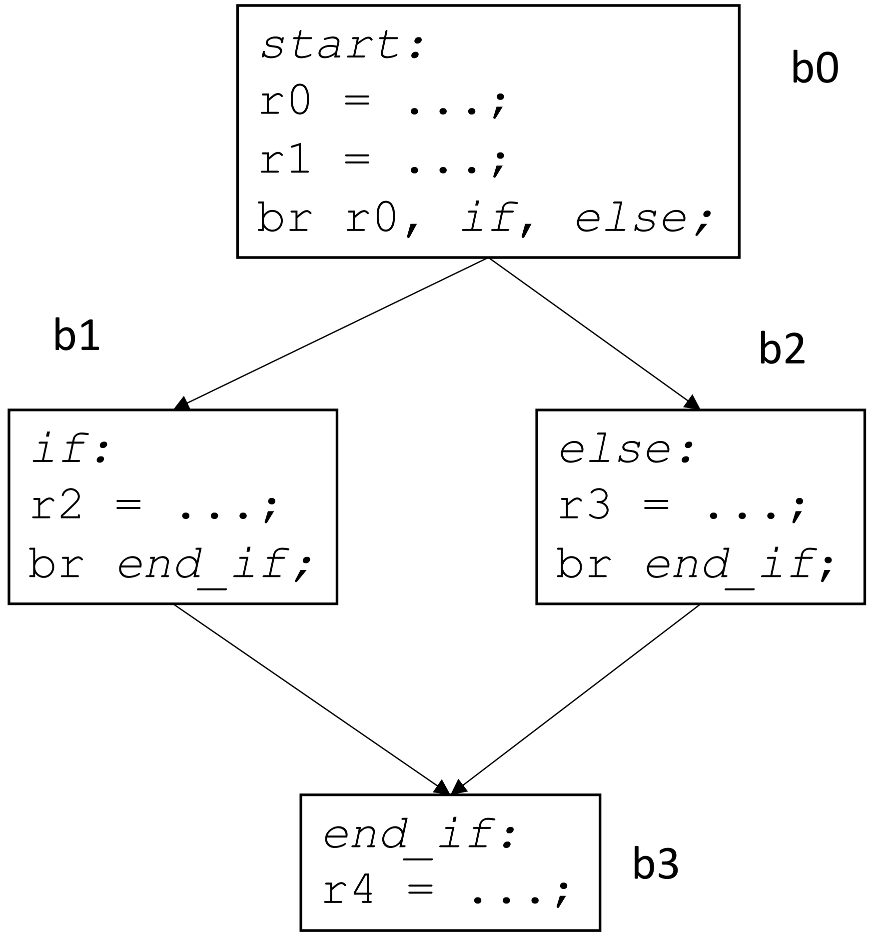
- Highly suggest reading for this part of the class
  - Chapter 9 of EAC

First concept:

- Dominance in a CFG
- Builds up a framework for reasoning
- Building block for many algorithms
  - global local value numbering when unlimited registers
  - Conversion to SSA

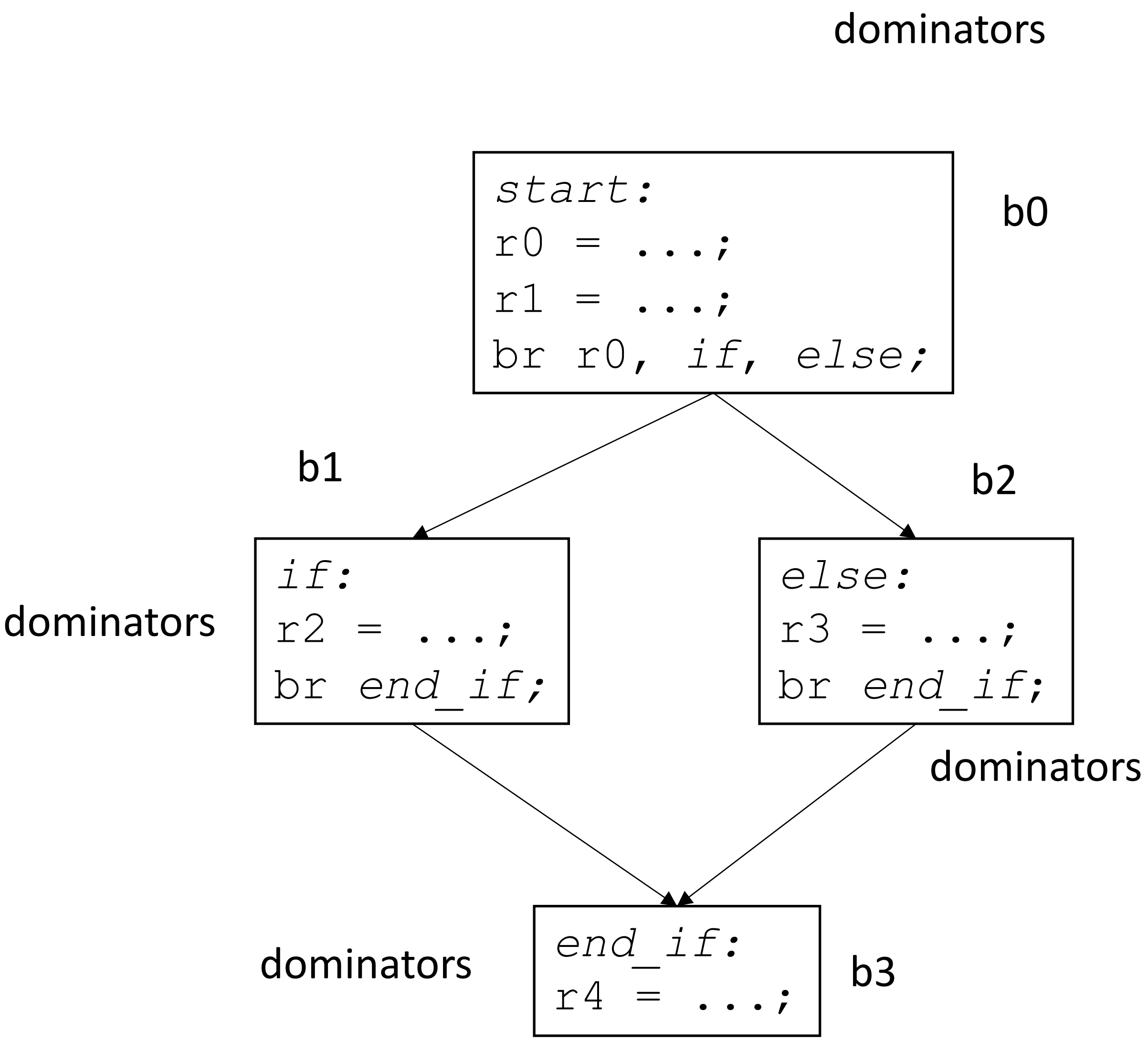
Dominance

- a block b_x dominates block b_y if every path from the start to block b_y goes through b_x
- definition:
  - domination (includes itself)
  - strict domination (does not include itself)

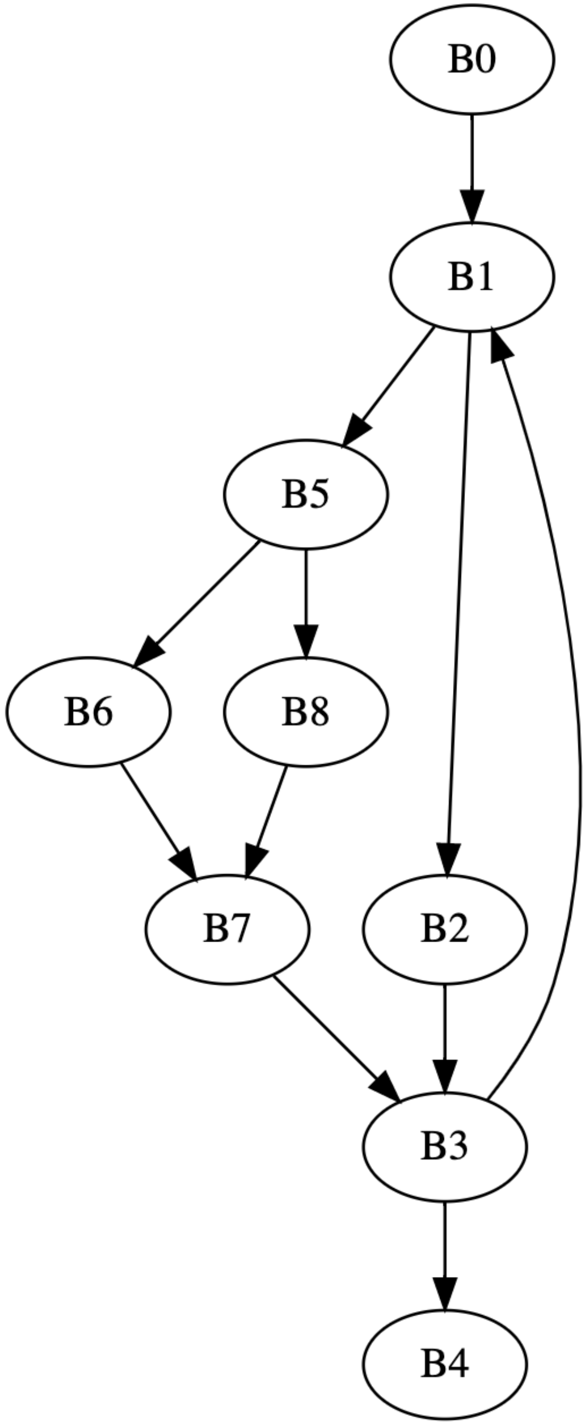


Dominance

- a block b_x dominates block b_y if every path from the start to block b_y goes through b_x
- definition:
  - domination (includes itself)
  - strict domination (does not include itself)
- Can we use this notion to extend local value numbering?

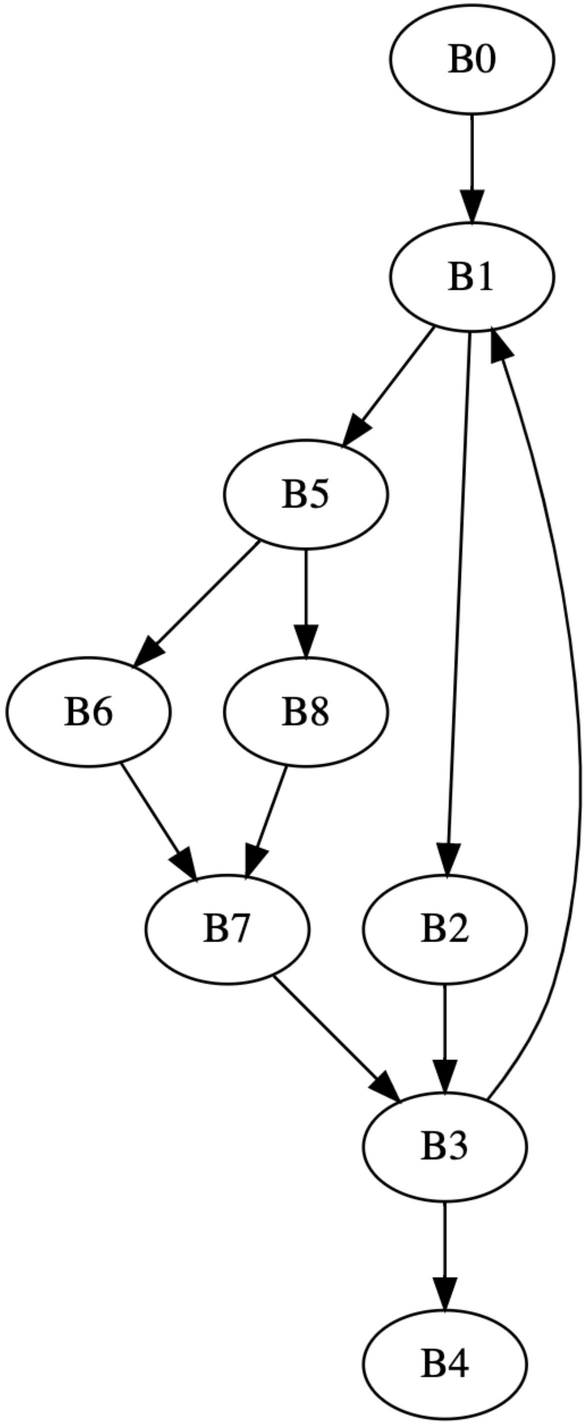


Node	Dominators
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	



Node	Dominators
B0	B0
B1	B0, B1
B2	B0, B1, B2
B3	B0, B1, B3
B4	B0, B1, B3, B4
B5	B0, B1, B5
B6	B0, B1, B5, B6
B7	B0, B1, B5, B7
B8	B0, B1, B5, B8

Concept introduced in 1959, algorithm not not given until 10 years later



Computing dominance

- Iterative fixed-point algorithm

- Initial state, all nodes start with all other nodes are dominators:
  - *Dom(n) = N*
  - *Dom(start) = {start}*

iteratively compute:

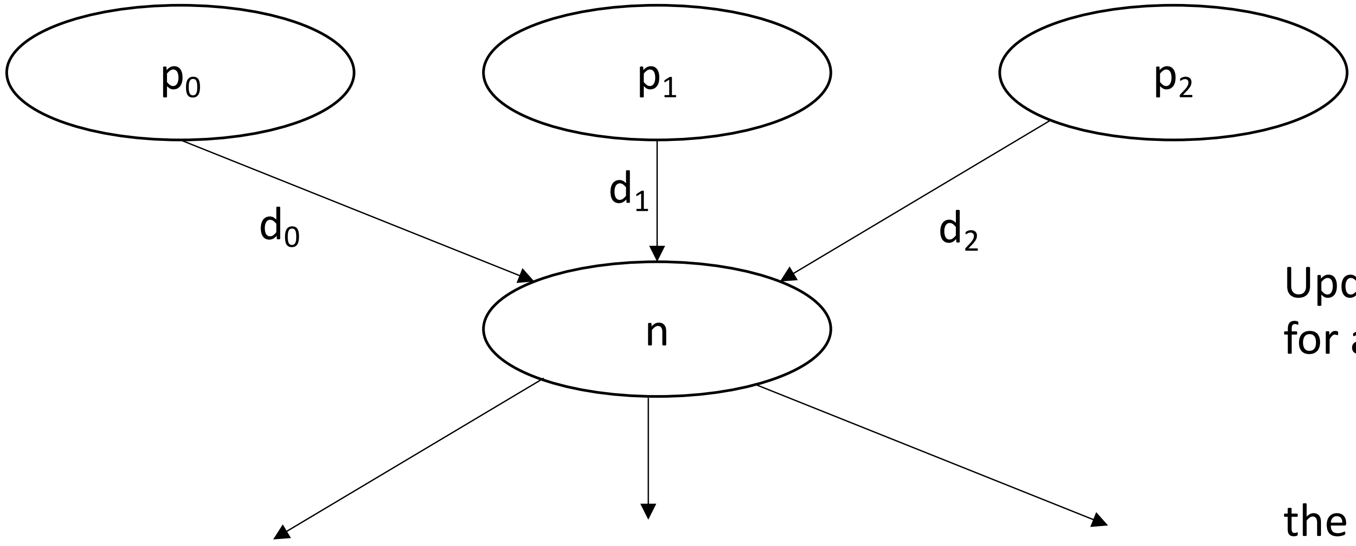
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

Building intuition behind the math

- This algorithm is vertex centric
  - local computations consider only a target node and its immediate neighbors
- At least one node is instantiated with ground truth:
  - starting node dominator is itself
- Information flows through the graph as nodes are updated

For example: Bellman Ford Shortest path

- Root node is initialized to 0
- Every node determines new distances based on incoming distances.
- When distances stop updating, the algorithm is converged

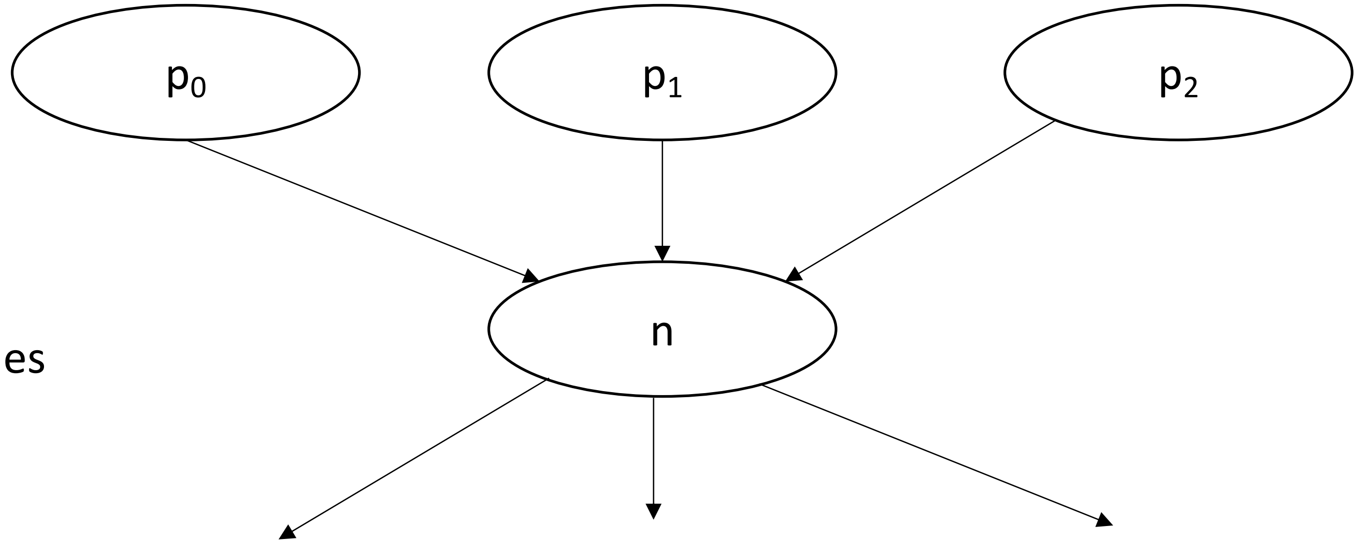


Update:
for all parents *p*: min(*p* + *d*)

the next iteration, another parent
may have found a shorter path.

Now lets think about dominance

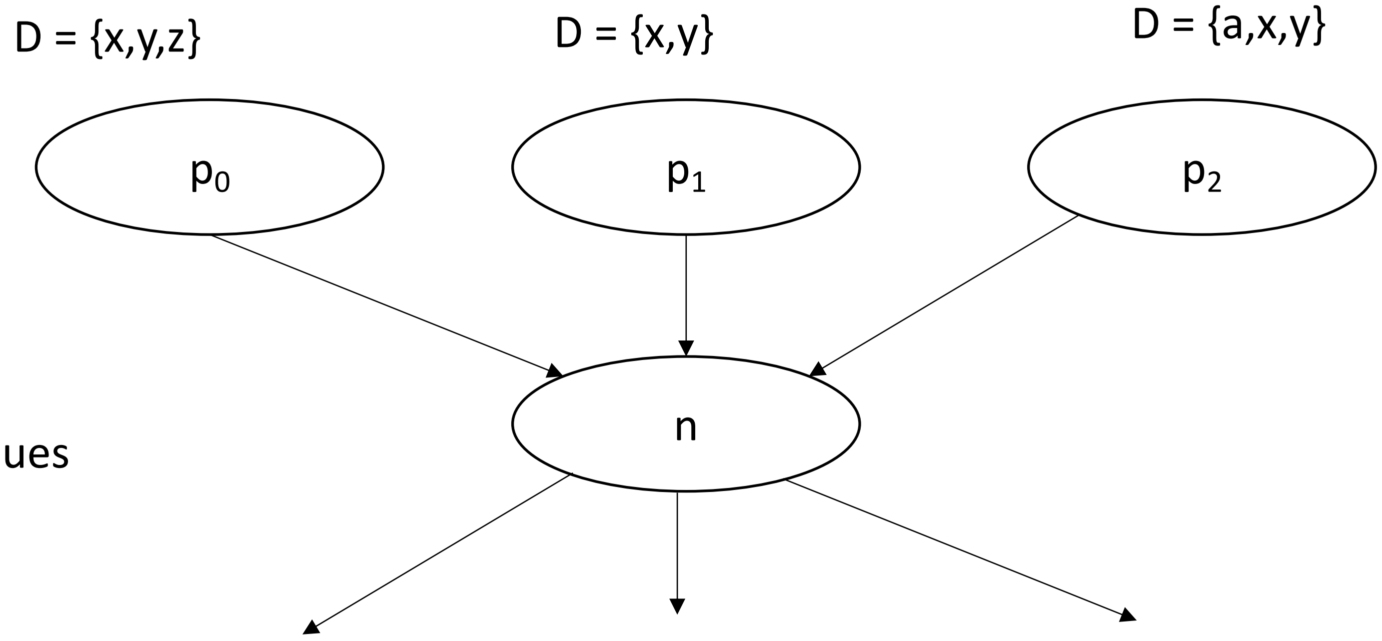
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

Now lets think about dominance

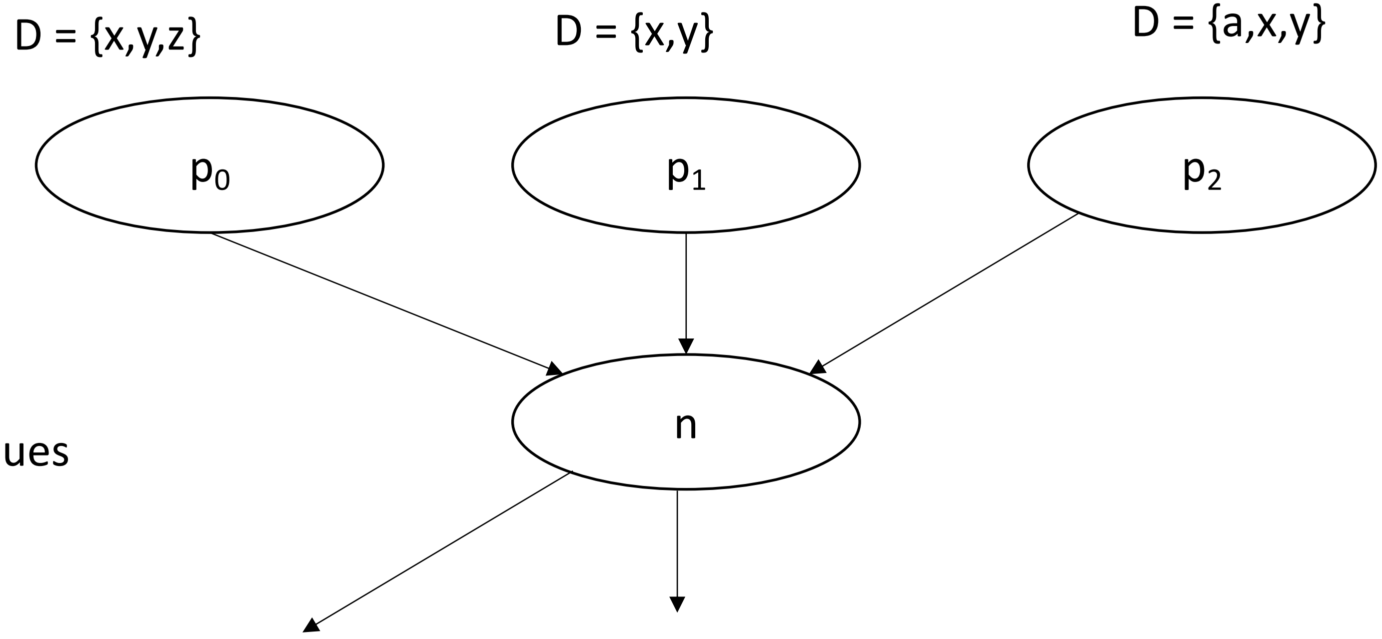
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

Now lets think about dominance

- Root node is initialized to itself
- Every node determines new dominators based on parent dominators

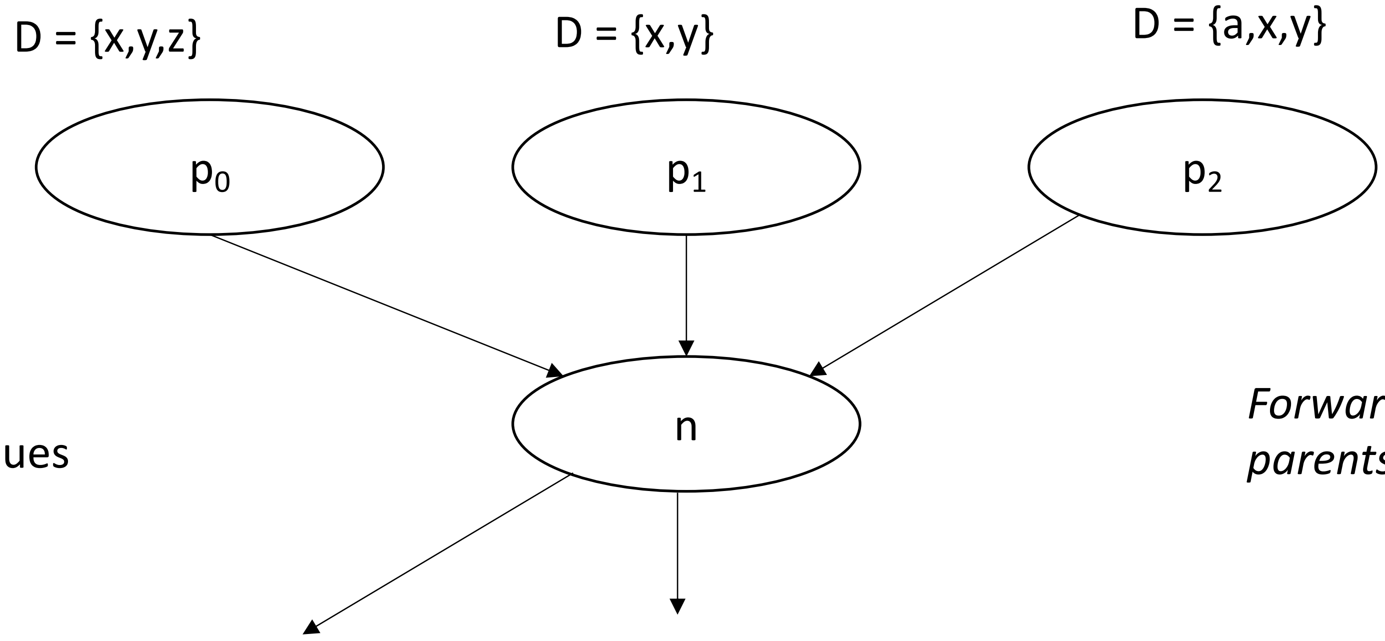


update:
intersection of parent values

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Now lets think about dominance

- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



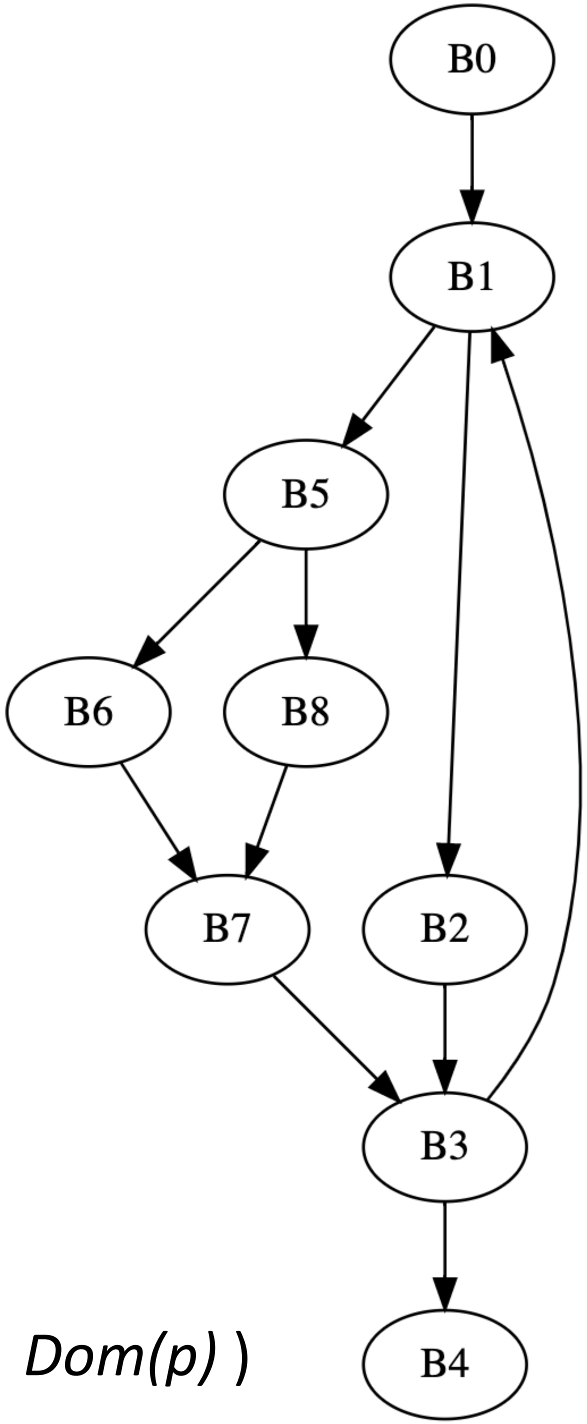
update:
intersection of parent values

Forward flow, as updates flow from parents to children.

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Lets try it

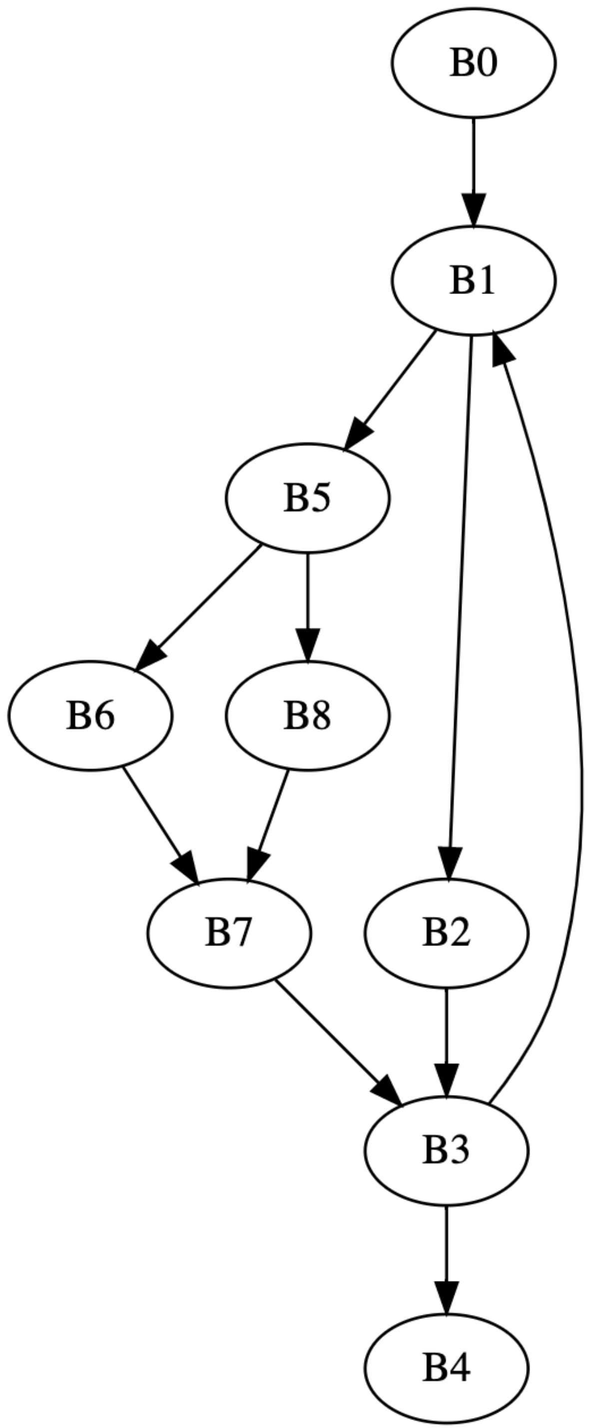
Node	Initial	Iteration 1
B0	B0	
B1	*N*	
B2	*N*	
B3	*N*	
B4	*N*	
B5	*N*	
B6	*N*	
B7	*N*	
B8	*N*	



$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in preds}(n)} Dom(p) \right)$$

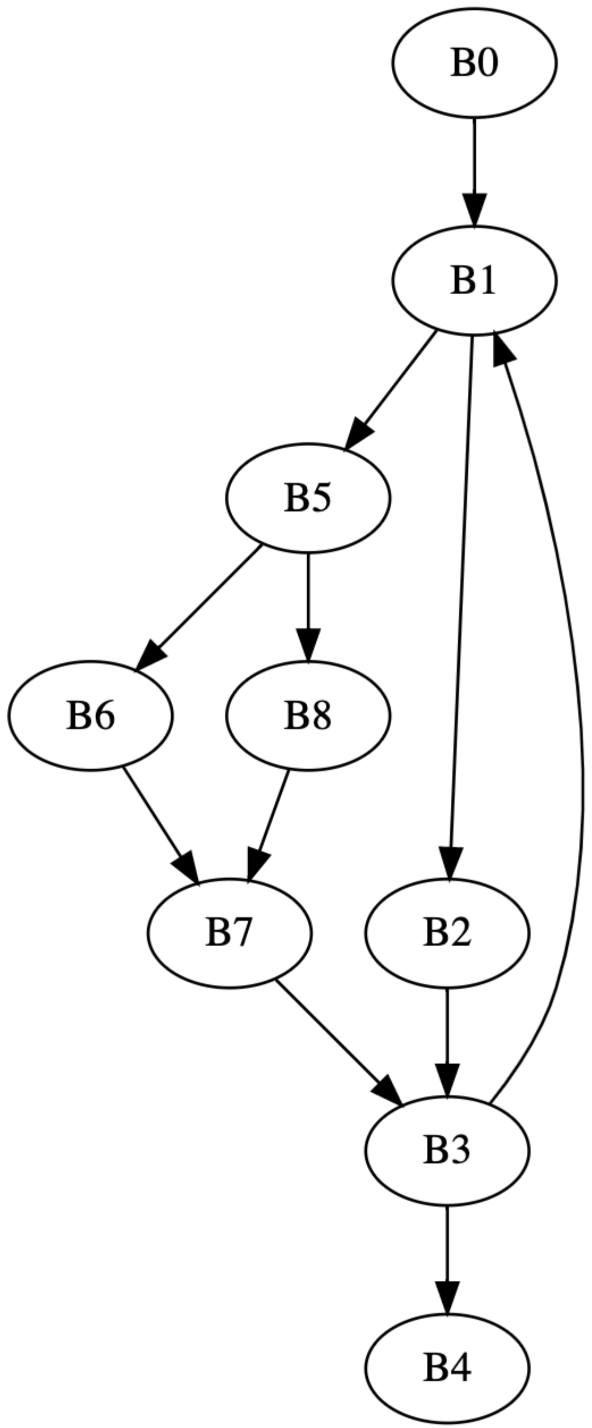
Lets try it

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0		
B1	*N*	B0,B1		
B2	*N*	B0,B1,B2		
B3	*N*	B0,B1,B2,B3		
B4	*N*	B0,B1,B2,B3,B4		
B5	*N*	B0,B1,B5		
B6	*N*	B0,B1,B5,B6		
B7	*N*	B0,B1,B5,B6,B7		
B8	*N*	B0,B1,B5,B8		



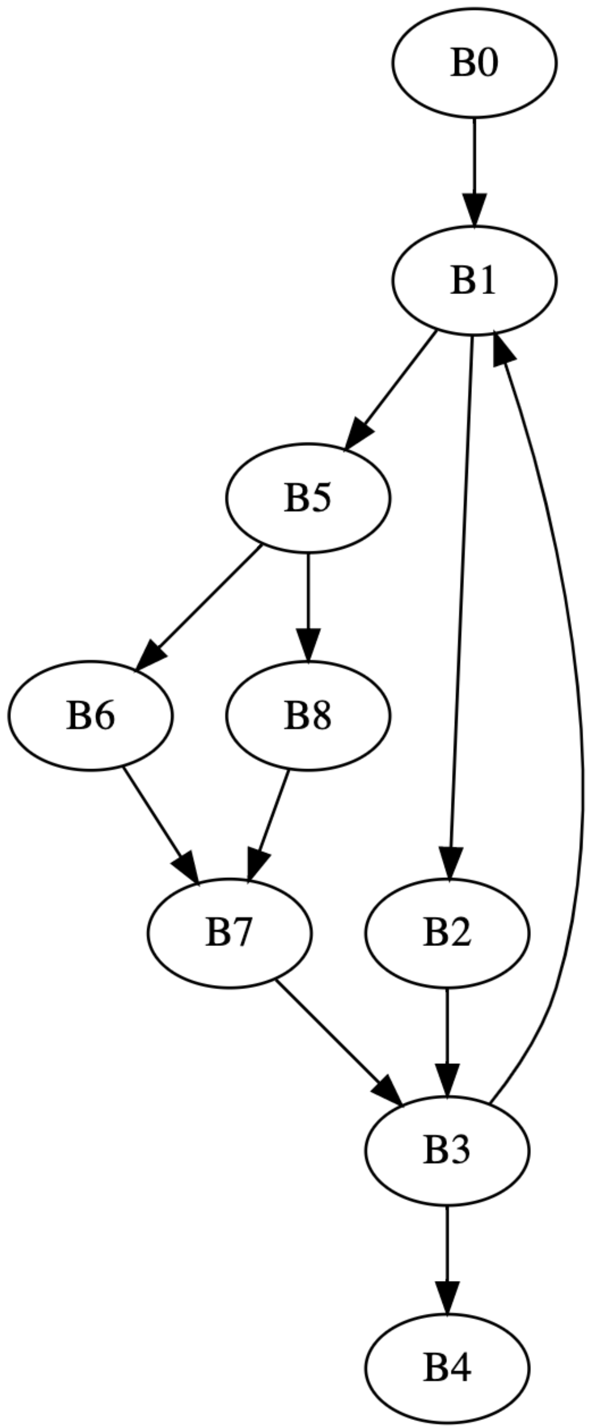
Lets try it

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	…	
B1	*N*	B0,B1	…	
B2	*N*	B0,B1,B2	…	
B3	*N*	B0,B1,B2,B3	B0,B1,B3	
B4	*N*	B0,B1,B2,B3,B4	B0,B1,B3,B4	
B5	*N*	B0,B1,B5	…	
B6	*N*	B0,B1,B5,B6	…	
B7	*N*	B0,B1,B5,B6,B7	B0,B1,B5,B7	
B8	*N*	B0,B1,B5,B8	…	



How can we optimize the algorithm?

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	…	…
B1	*N*	B0,B1	…	…
B2	*N*	B0,B1,B2	…	…
B3	*N*	B0,B1,B2,B3	B0,B1,B3	…
B4	*N*	B0,B1,B2,B3,B4	B0,B1,B3,B4	…
B5	*N*	B0,B1,B5	…	…
B6	*N*	B0,B1,B5,B6	…	…
B7	*N*	B0,B1,B5,B6,B7	B0,B1,B5,B7	…
B8	*N*	B0,B1,B5,B8	…	…

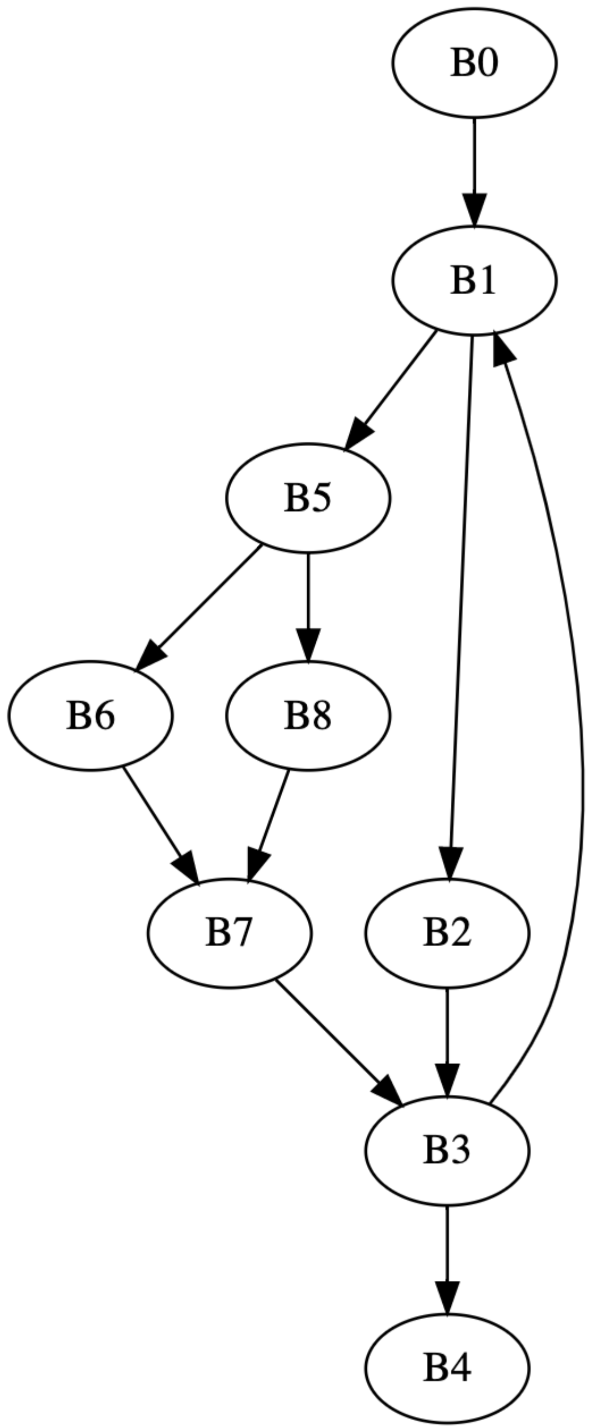


How can we optimize the algorithm?

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	...	...
B1	*N*	B0,B1	...	...
B2	*N*	B0,B1,B2	...	...
B3	*N*	B0,B1,B2,B3	B0,B1,B3	...
B4	*N*	B0,B1,B2,B3,B4	B0,B1,B3,B4	...
B5	*N*	B0,B1,B5	...	...
B6	*N*	B0,B1,B5,B6	...	...
B7	*N*	B0,B1,B5,B6,B7	B0,B1,B5,B7	...
B8	*N*	B0,B1,B5,B8	...	...

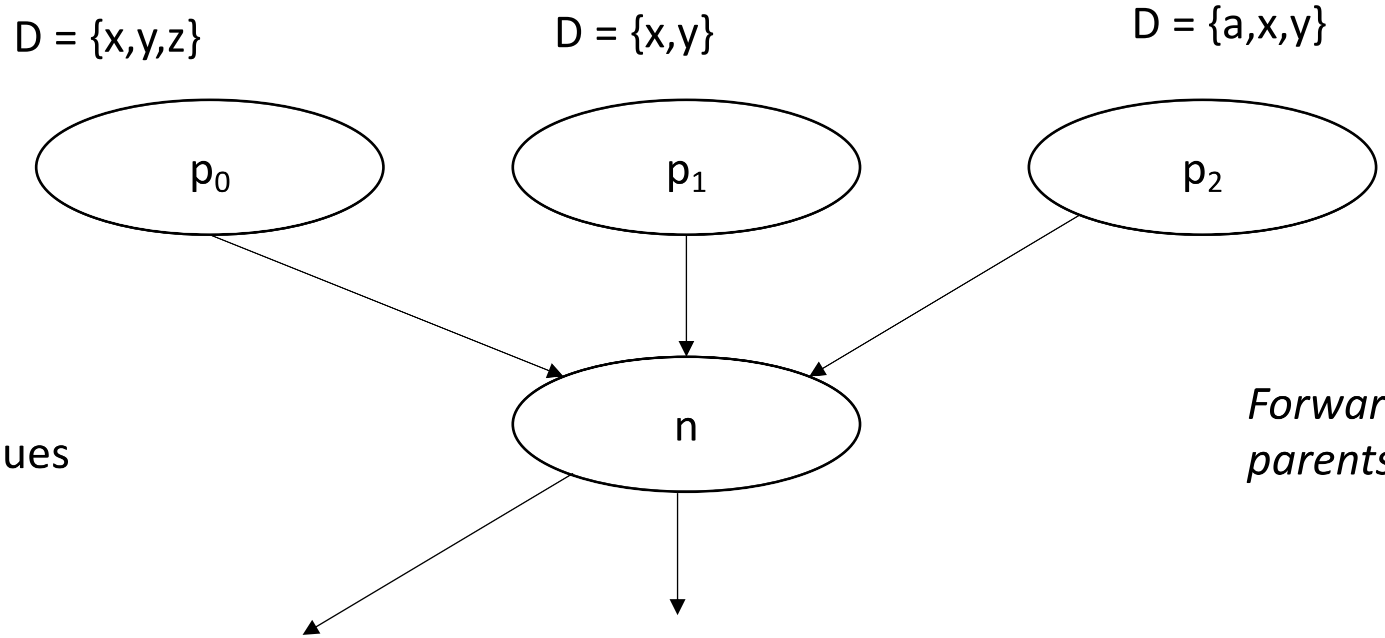
This can be any order...

How can we optimize the order?



Given this intuition, what ordering would be best?

- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

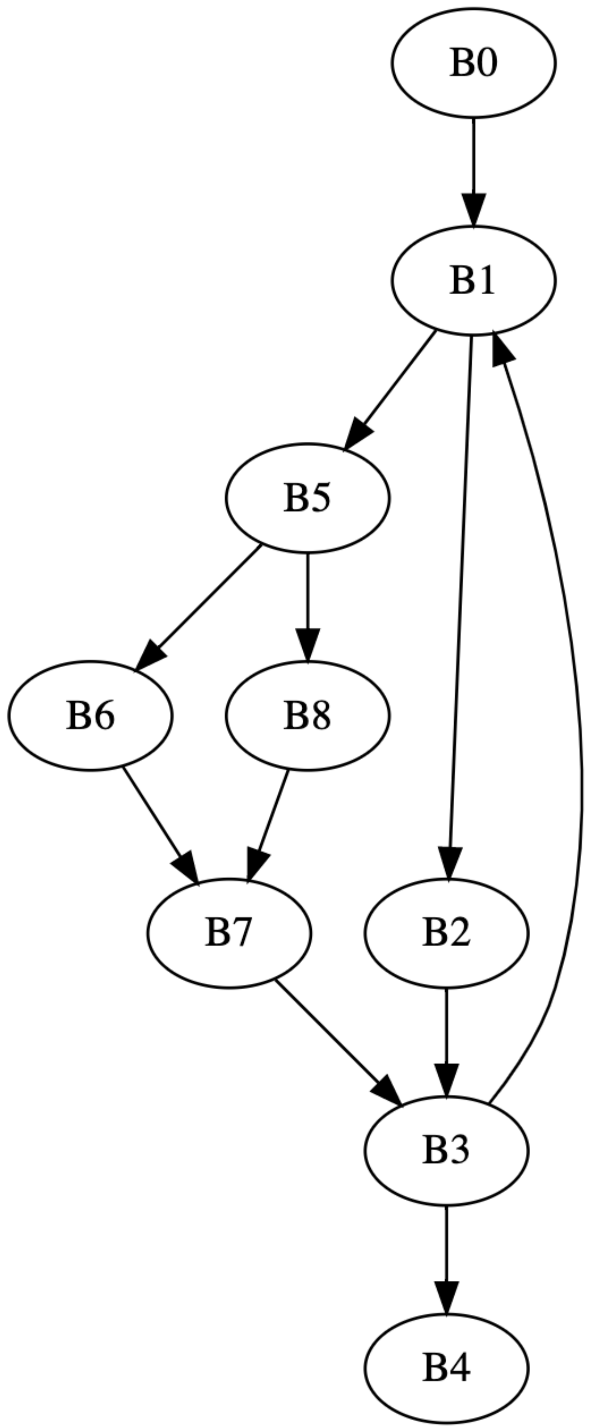
Forward flow, as updates flow from parents to children.

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

How can we optimize the algorithm?

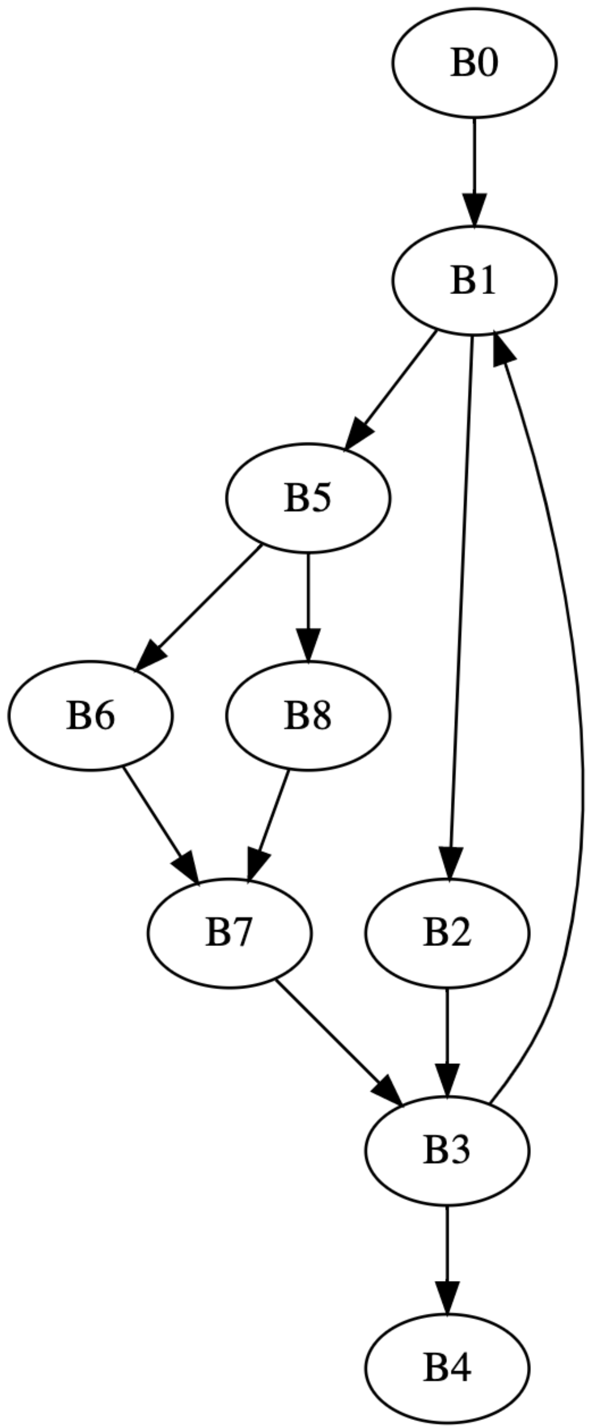
Node	New Order
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	

Reverse
post-order (rpo),
where parents are visited
first



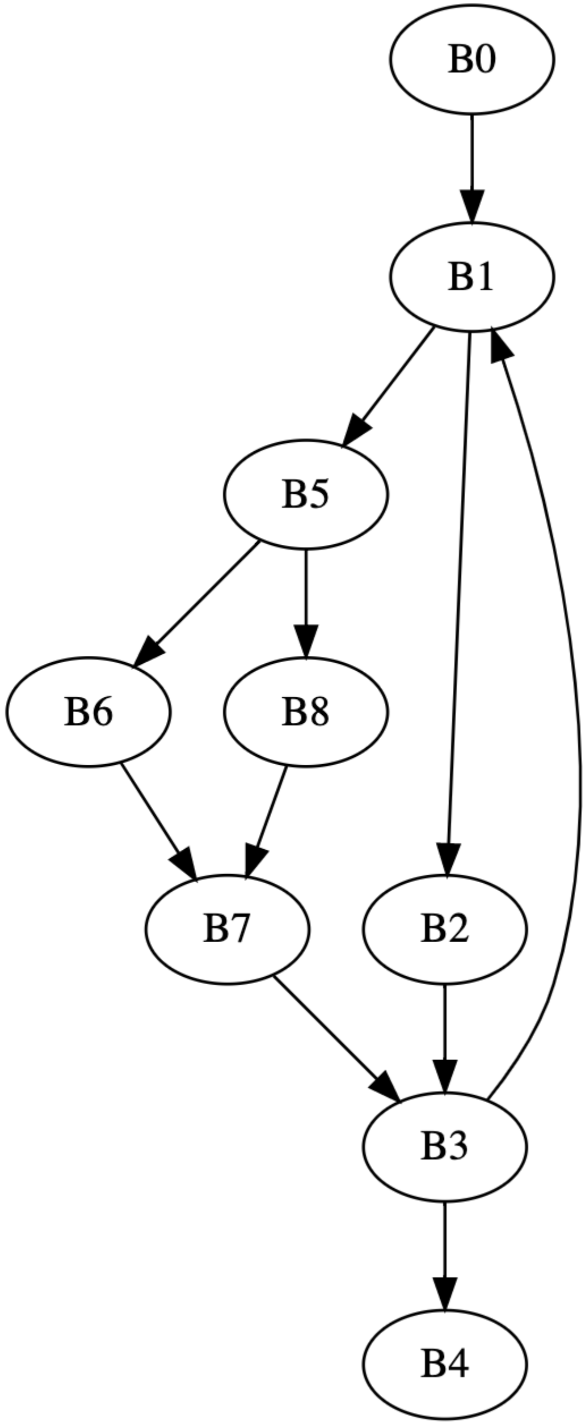
How can we optimize the algorithm?

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0			
B1	*N*			
B2	*N*			
B5	*N*			
B6	*N*			
B8	*N*			
B7	*N*			
B3	*N*			
B4	*N*			



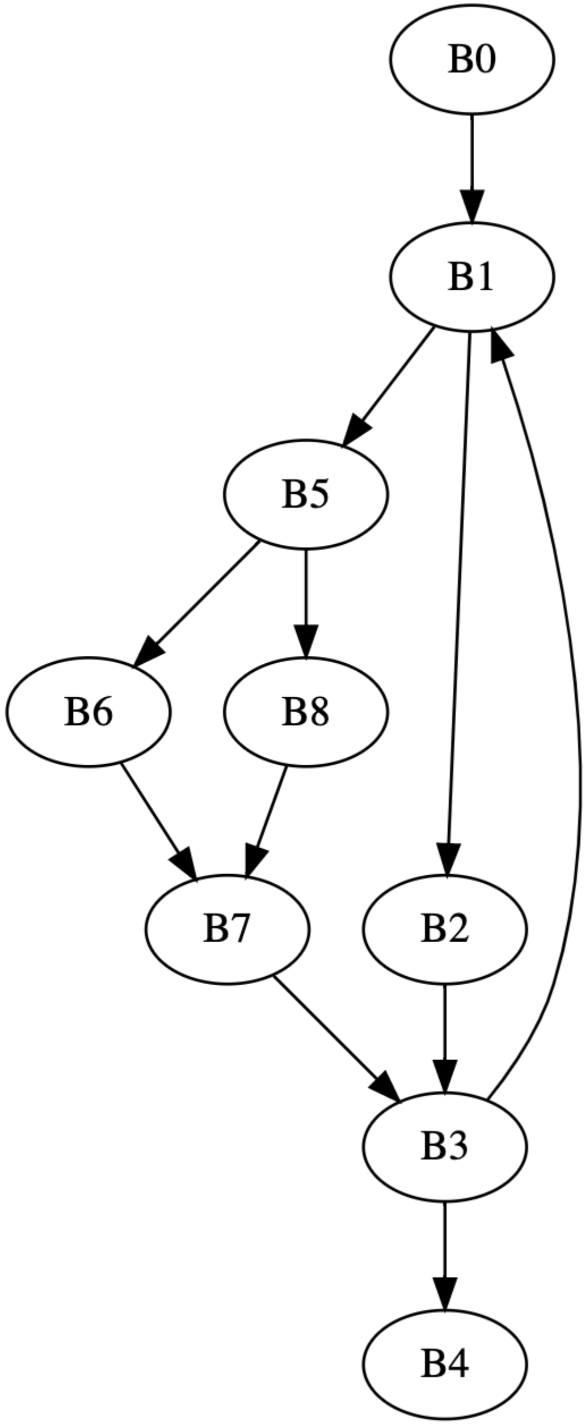
How can we optimize the algorithm?

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0		
B1	*N*	B0,B1		
B2	*N*	B0,B1,B2		
B5	*N*	B0,B1,B5		
B6	*N*	B0,B1,B5,B6		
B8	*N*	B0,B1,B5,B8		
B7	*N*	B0,B1,B5,B7		
B3	*N*	B0,B1,B3		
B4	*N*	B0,B1,B4		



How can we optimize the algorithm?

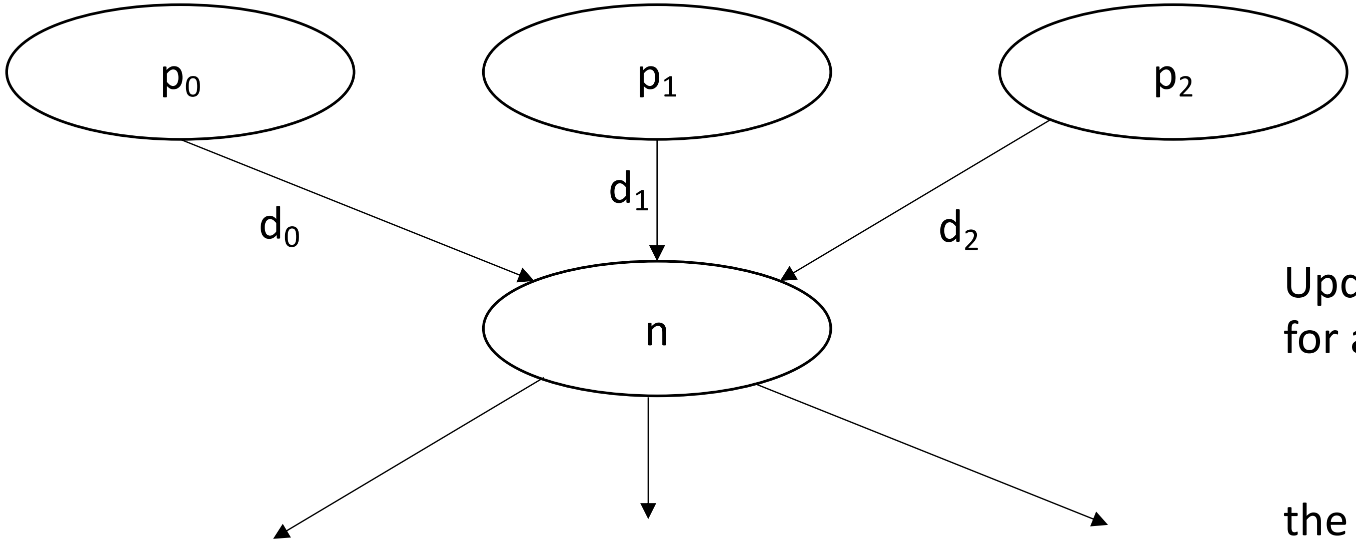
Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	...	
B1	*N*	B0,B1	...	
B2	*N*	B0,B1,B2	...	
B5	*N*	B0,B1,B5	...	
B6	*N*	B0,B1,B5,B6	...	
B8	*N*	B0,B1,B5,B8	...	
B7	*N*	B0,B1,B5,B7	...	
B3	*N*	B0,B1,B3	...	
B4	*N*	B0,B1,B4	...	



A quick aside about graph algorithms:

- Does node ordering matter in SSSP?
- Yes! Dijkstra's algorithm uses a priority queue
- Prioritize nodes with the lowest value

Traversal order in graph algorithms is a big research area!



Update:
for all parents *p*: min(*p* + *d*)

the next iteration, another parent may have found a shorter path.

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

p ← Live variables: z, w

```
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

p Live variables: ?

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
...          ←  p   Live variables: x,w
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

```
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
...          <- p   Live variables: x,w
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

```
x = 5
...
if (z):
   y = 6     <- p   Live variables: ?
else:
   y = x
print(y)
print(w)
```

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

p

Live variables: x,w

```
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

p

Live variables: y,w

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
...          <-- p   Live variables: x,w
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

```
//start      <-- p   Live variables: ?
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
...          <---- p    Live variables: x,w
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

```
//start      <---- p    Live variables: w,z
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

Another analysis: Live Variable Analysis

- A variable *v* is live at some point *p* in the program if there exists a path from *p* to some use of *v* where *v* has not been redefined

- examples:

```
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

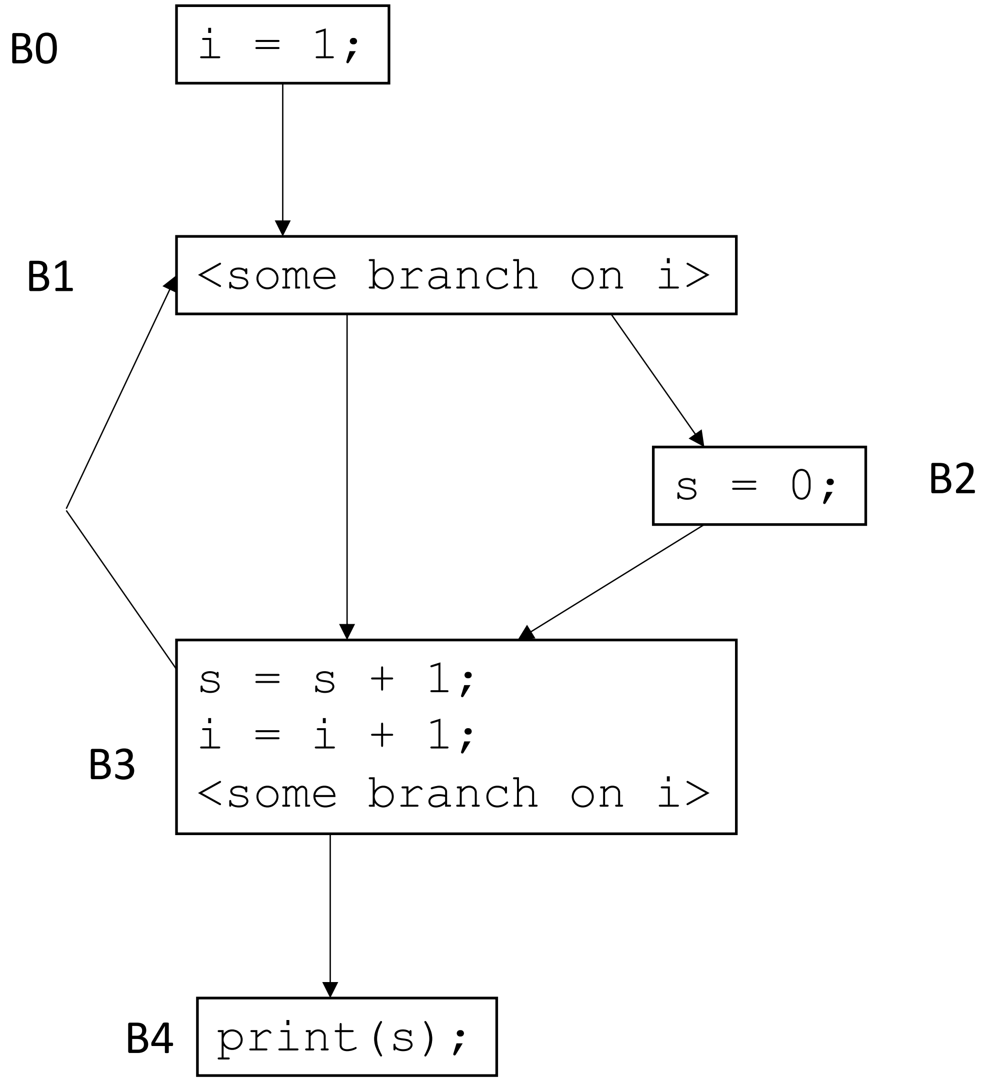
p Live variables: x,w

Accessing an uninitialized variable!

```
//start
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

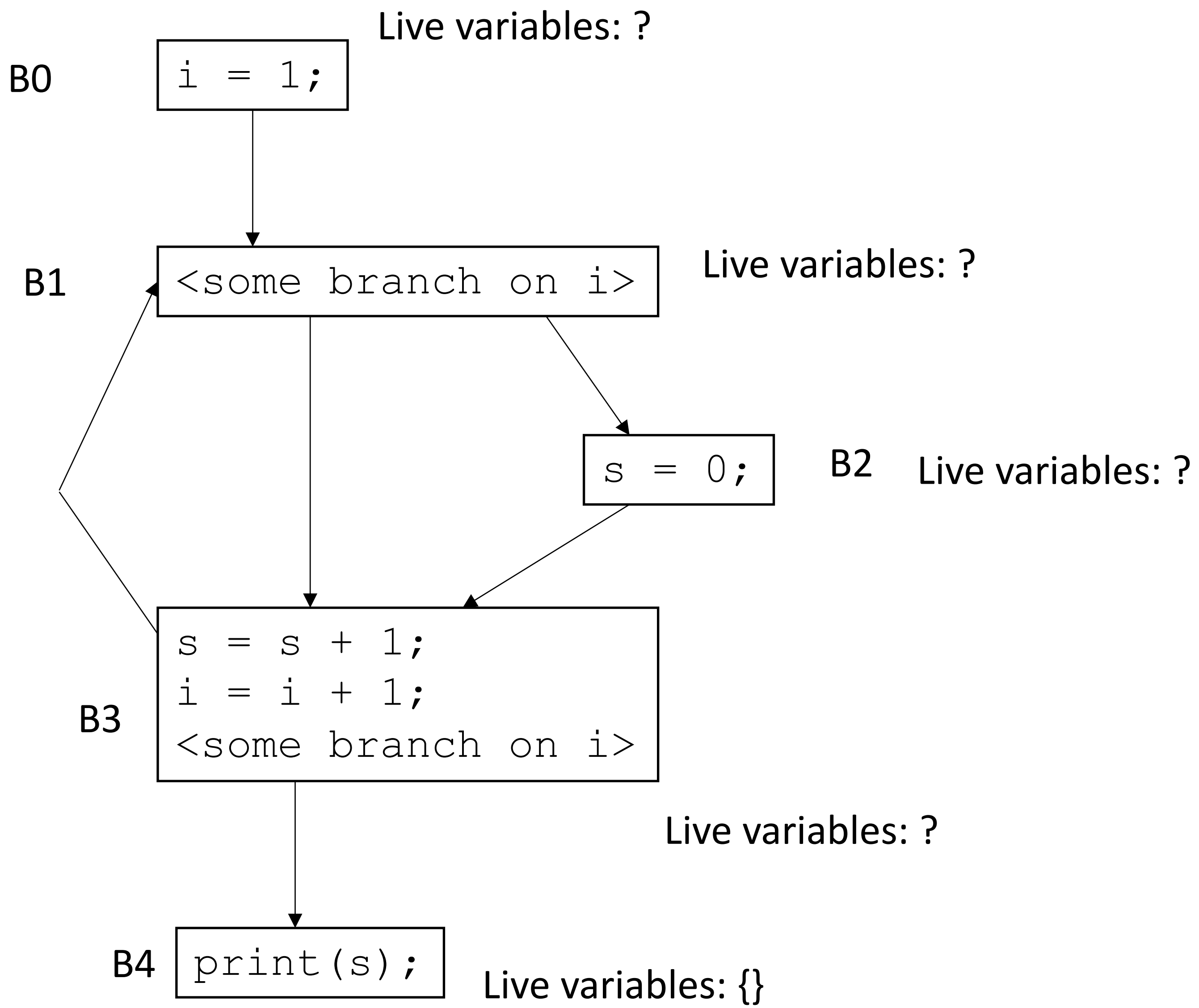
p Live variables: w,z

Live variable analysis in the CFG:

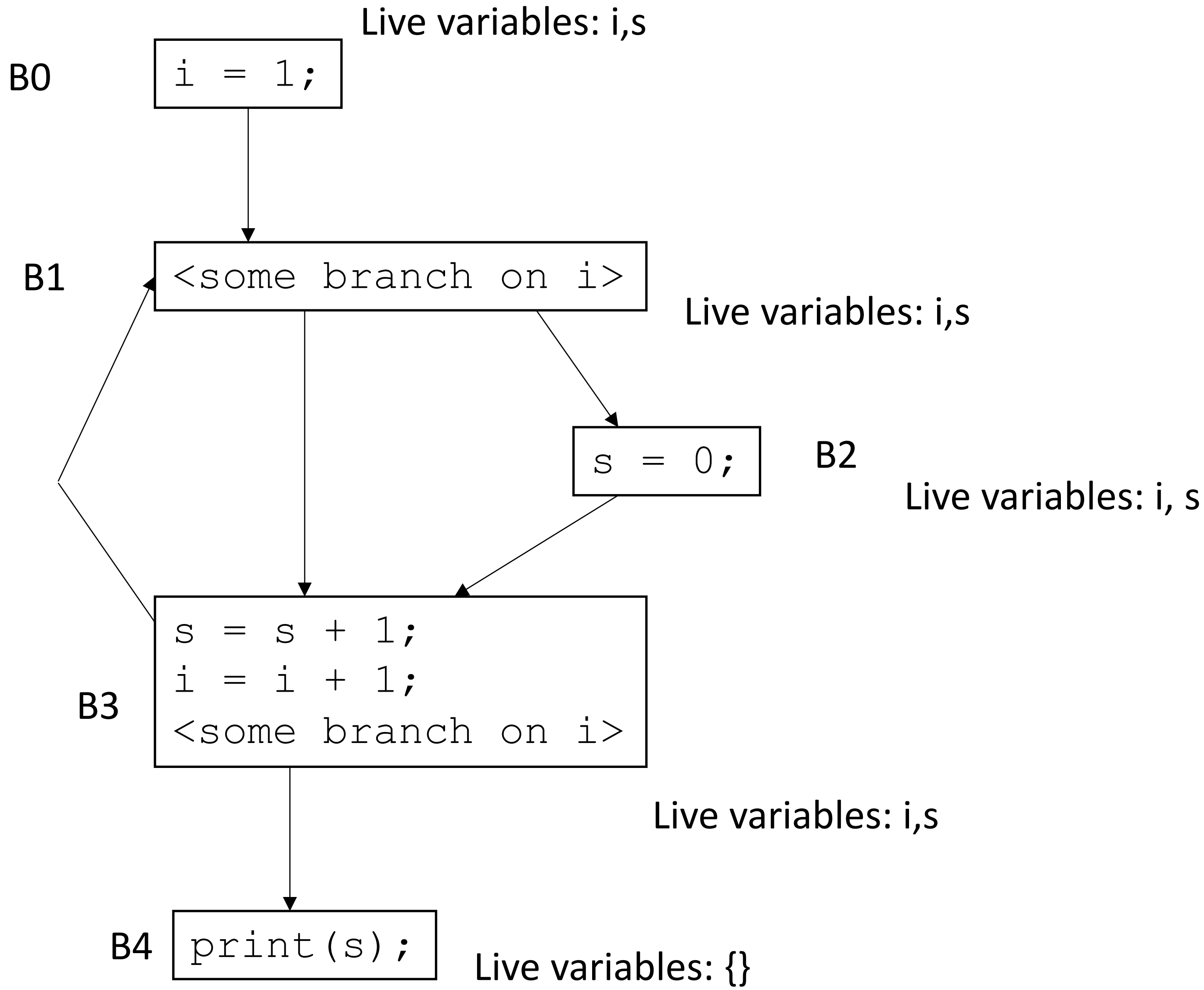


For each block B_x : we want to compute LiveOut: The set of variables that are live at the end of B_x

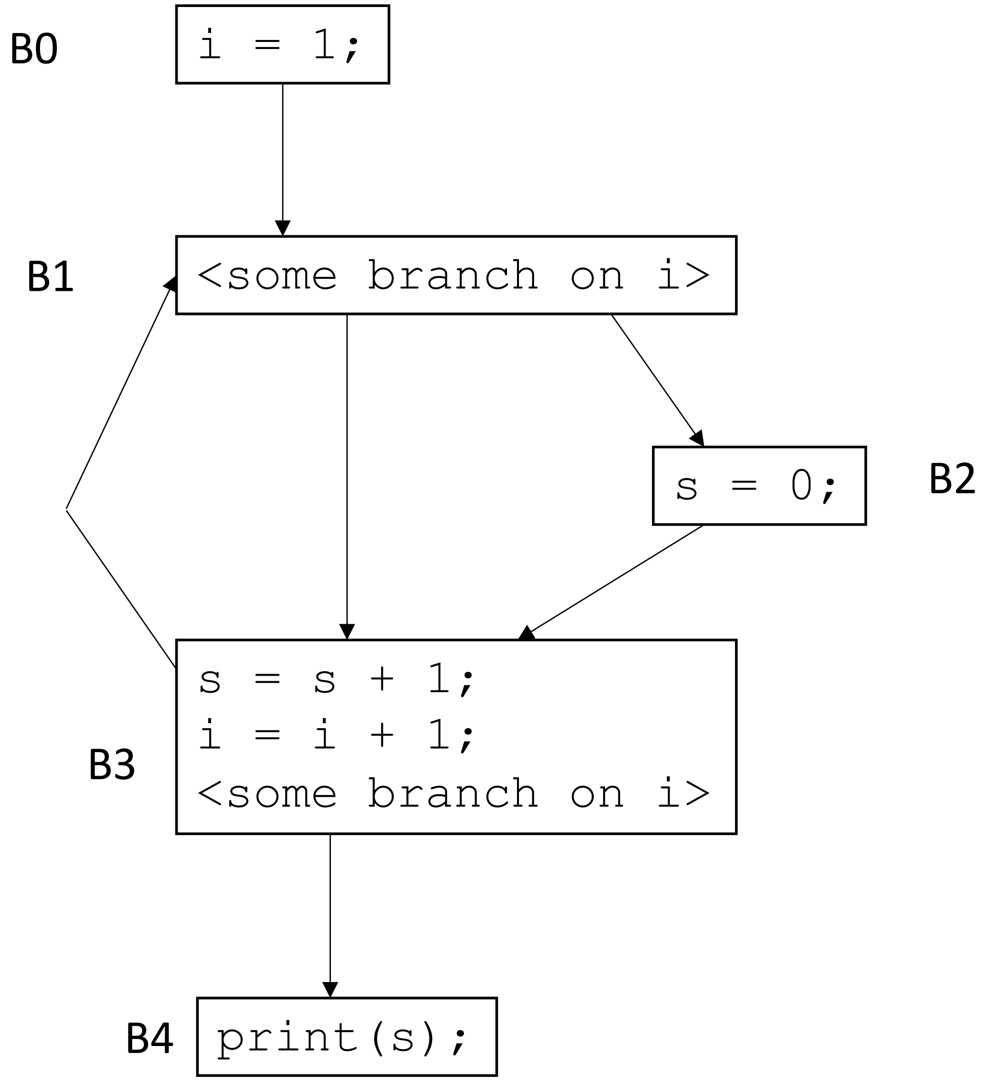
Live variable analysis in the CFG:



Live variable analysis in the CFG:



Live variable analysis in the CFG:



To compute the LiveOut sets, we need two initial sets:

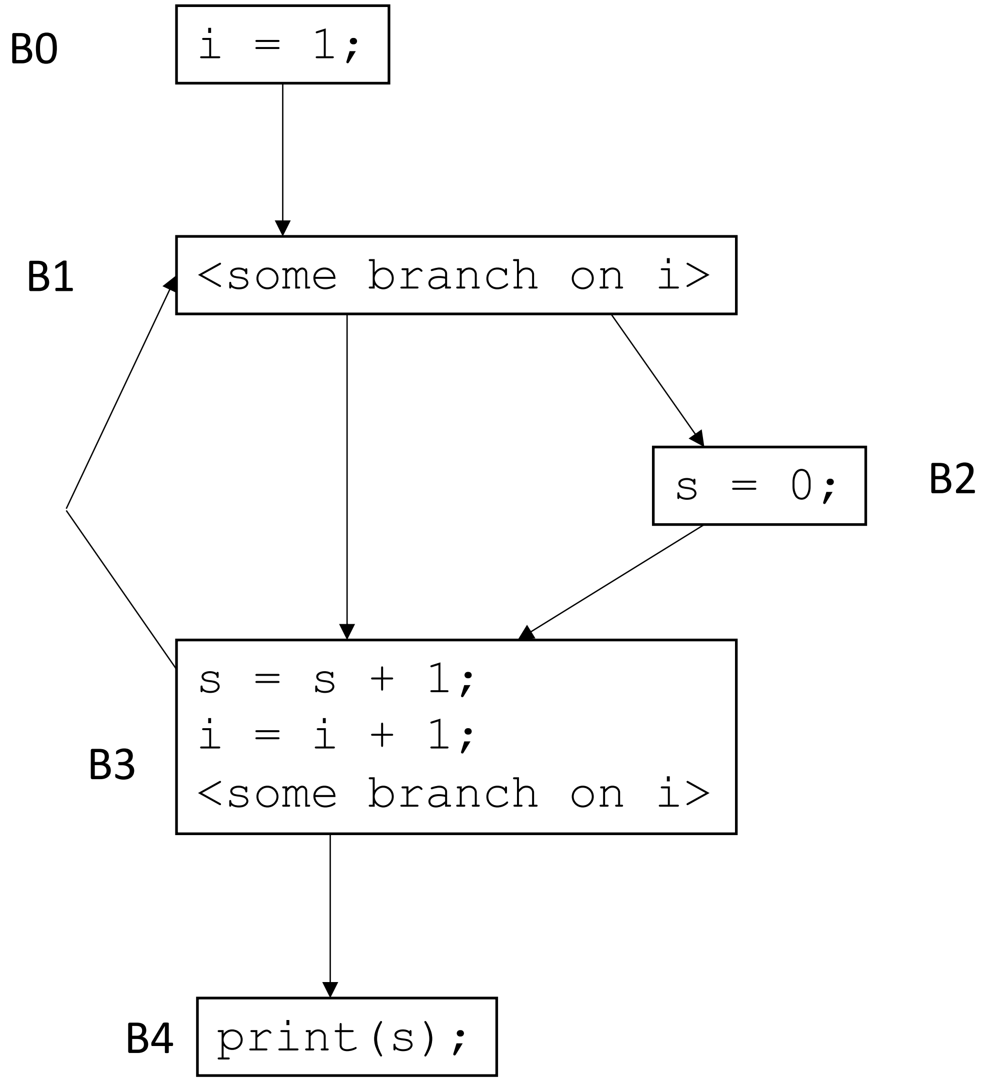
VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b
is any variable in b that is satisfies these two conditions

- it is not written to and it is read
- it is read before it is written to

Block	VarKill	UEVar
B0		
B1		
B2		
B3		
B4		

Live variable analysis in the CFG:



To compute the LiveOut sets, we need two initial sets:

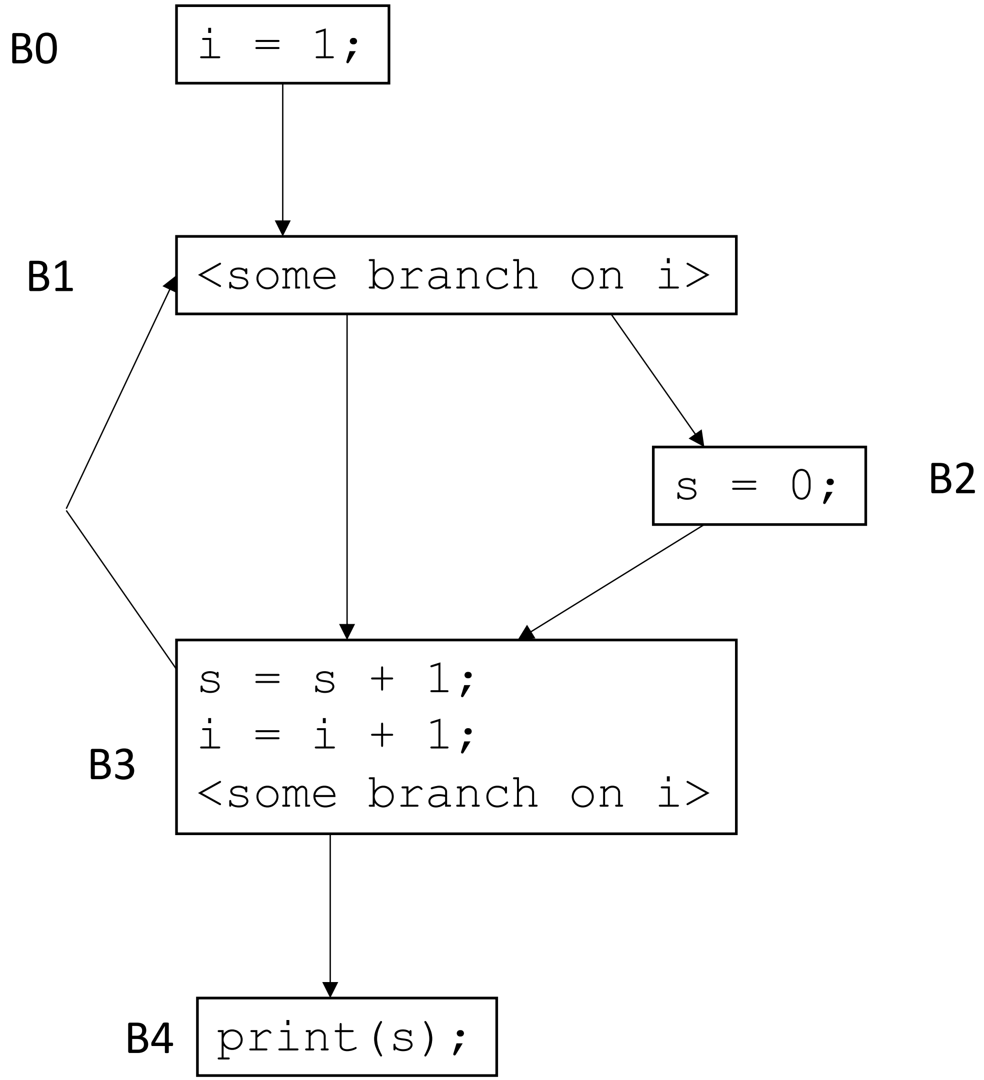
VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b
is any variable in b that is satisfies these two conditions

- it is not written to and it is read
- it is read before it is written to

Block	VarKill	UEVar
B0	i	
B1	{}	
B2	s	
B3	s,i	
B4	{}	

Live variable analysis in the CFG:



To compute the LiveOut sets, we need two initial sets:

VarKill for block b is any variable in block b that gets overwritten

UEVar (upward exposed variable) for block b
is any variable in b that is satisfies these two conditions

- it is not written to and it is read
- it is read before it is written to

Block	VarKill	UEVar
B0	i	{}
B1	{}	i
B2	s	{}
B3	s,i	s,i
B4	{}	s

Live variable analysis in the CFG:

- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Live variable analysis in the CFG:

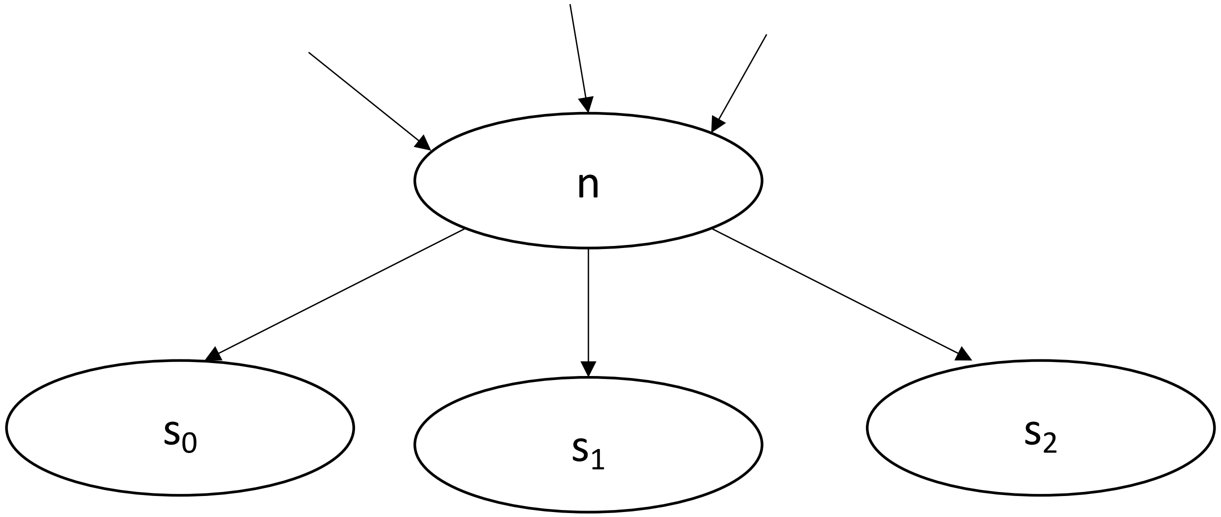
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e., end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed-point computation:

$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$

Live variable analysis in the CFG:

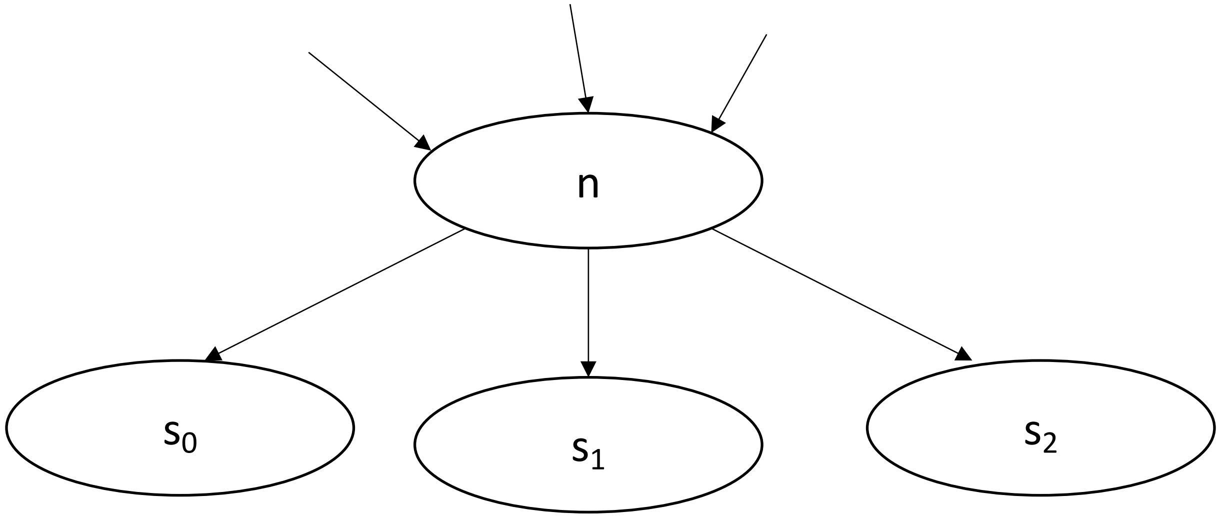
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$



Backwards flow analysis because values flow from successors

Live variable analysis in the CFG:

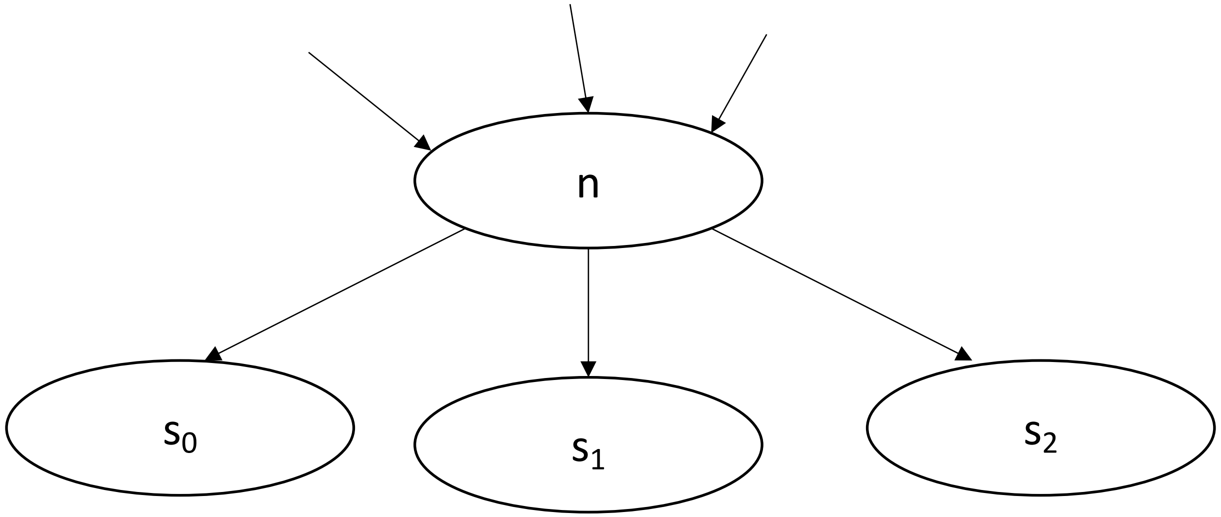
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

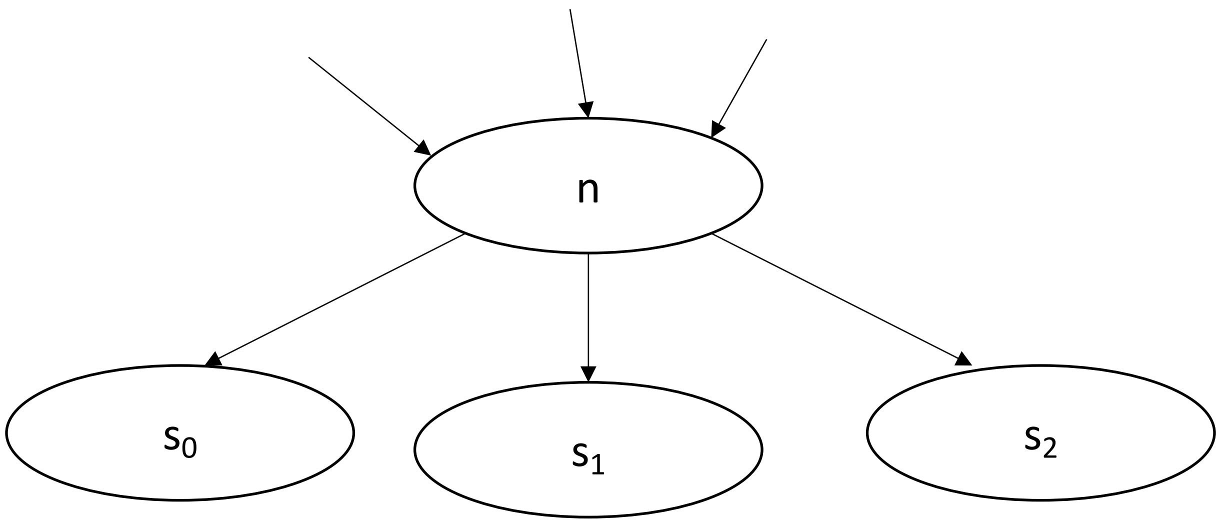
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$



variables that are not overwritten in s

Live variable analysis in the CFG:

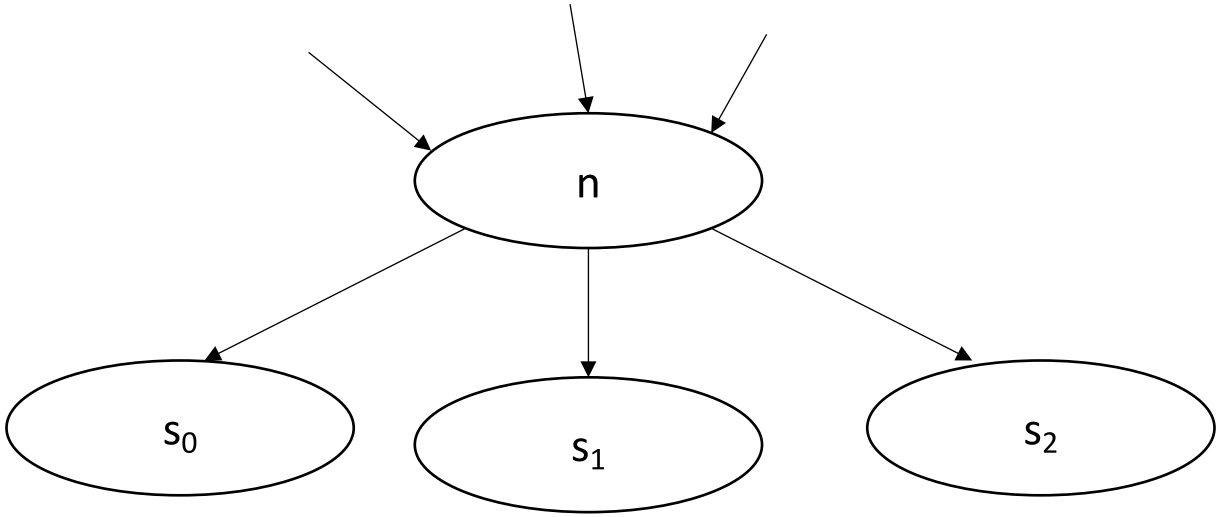
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$



variables that are live at the end of s

Live variable analysis in the CFG:

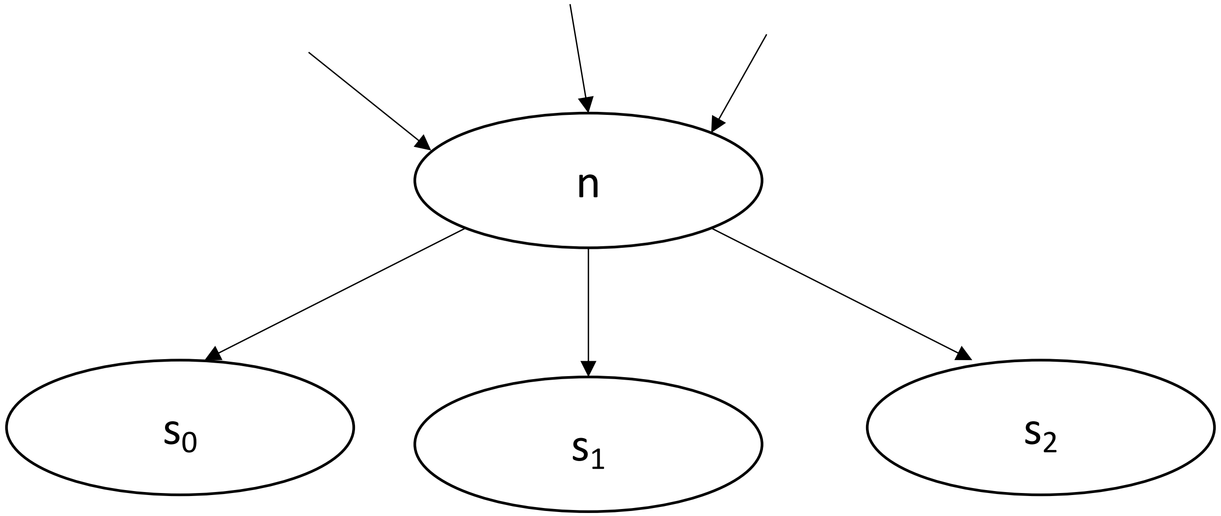
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)} \right) \right)$$



variables that are live at the end of s, and not overwritten by s

Live variable analysis in the CFG:

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$



LiveOut is a union
rather than an intersection

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Consider the language we use for each:

- **Dominance** of node b_x contains b_y if:
  - every path from the start to b_x goes through b_y

- **LiveOut** of node b_x contains variable y if:
  - some path from b_x contains a usage of y

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Consider the language we use for each:

- **Dominance** of node b_x contains b_y if:
  - **every** path from the start to b_x goes through b_y
- **LiveOut** of node b_x contains variable y if:
  - **some** path from b_x contains a usage of y
- *Some vs. Every*

$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$

$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$

Next time:

- More global analysis!