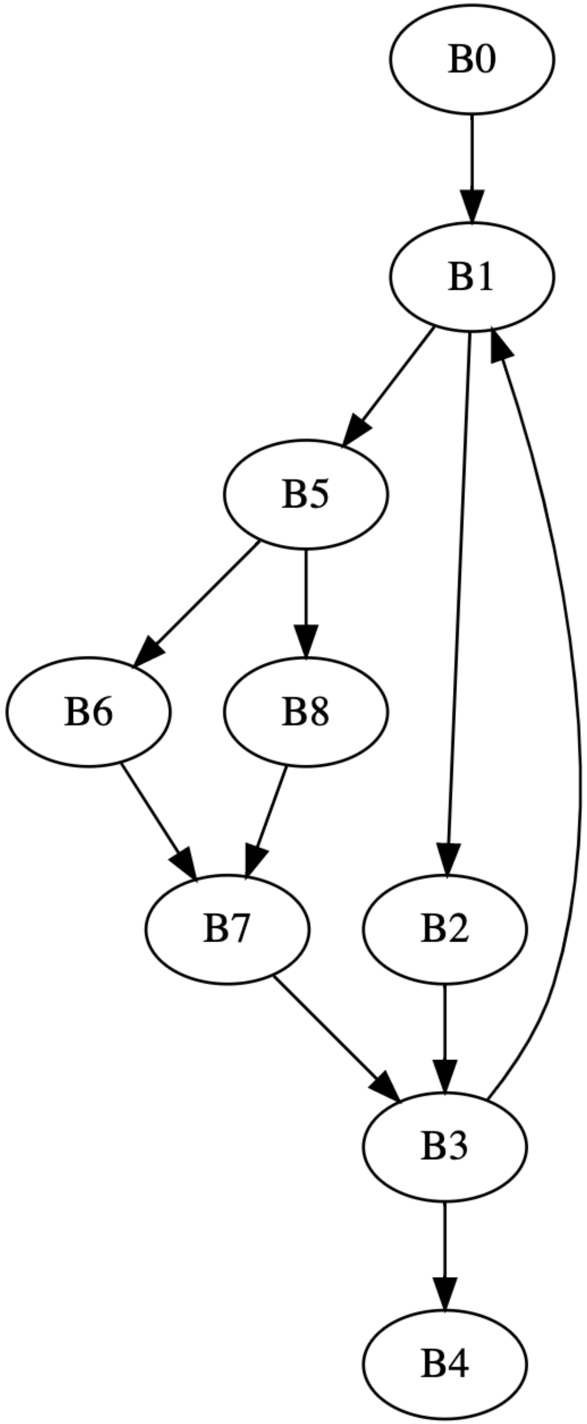
CSE211: Compiler Design

Oct. 18, 2021

- **Topic**: Flow analysis and Live variables
- **Questions**:
    - *How can we deal with arbitrary control flow graphs?*



Announcements

- Homework 1:
  - Due today (at 11:59 pm)
  - zip up files and submit on Canvas
    - one or two zip files, doesn't matter as long as I can easily get to the code!

- Homework 2:
  - Out now: specification is out. Code skeletons are released
  - 2 weeks to complete
    - Local Value Numbering
    - Live variable analysis (today)

Announcements

Next two classes:

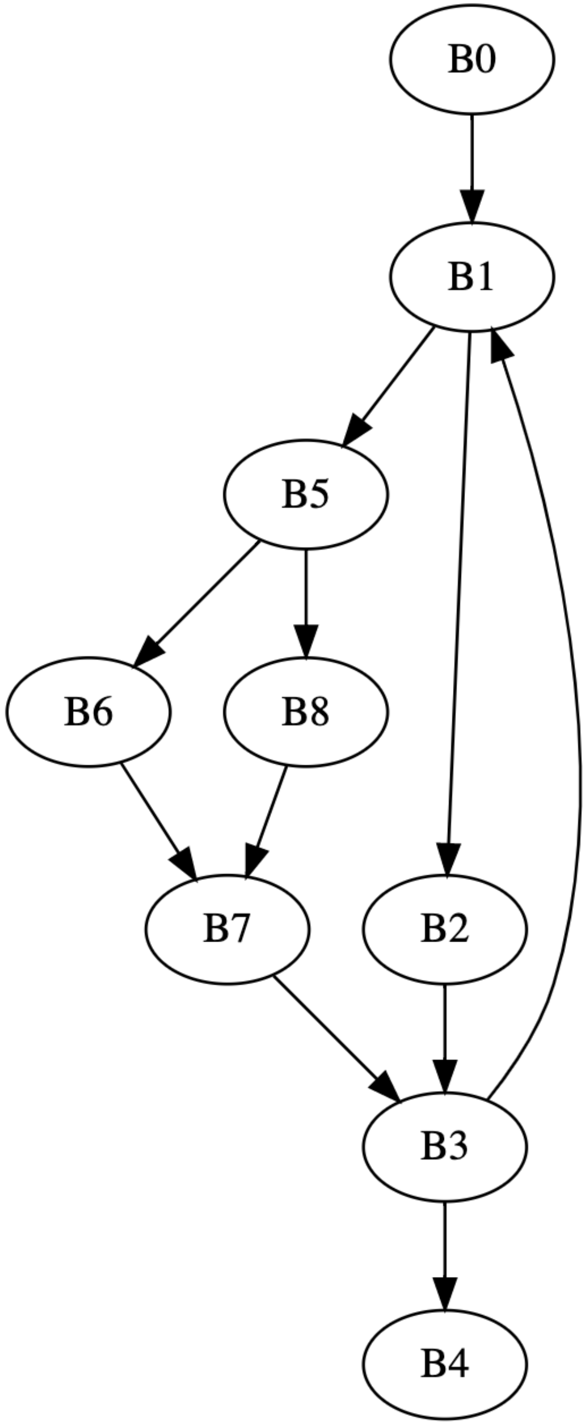
- Wednesday:
  - Will be canceled ☹ timing conflict that I miscalculated at the conference.
  - You can spend the time working on HW2
- Friday will be remote
  - I will give a live lecture (zoom link on canvas), Please try to attend, although I won’t be taking attendance
  - I will record the lecture and make it available online if you would prefer to attend asynchronously

CSE211: Compiler Design

Oct. 18, 2021

- **Topic**: global optimizations

- **Questions**:
  - *How can we deal with arbitrary control flow graphs?*



Review

- Local optimizations:
    - Examples?

Local optimizations: local value numbering

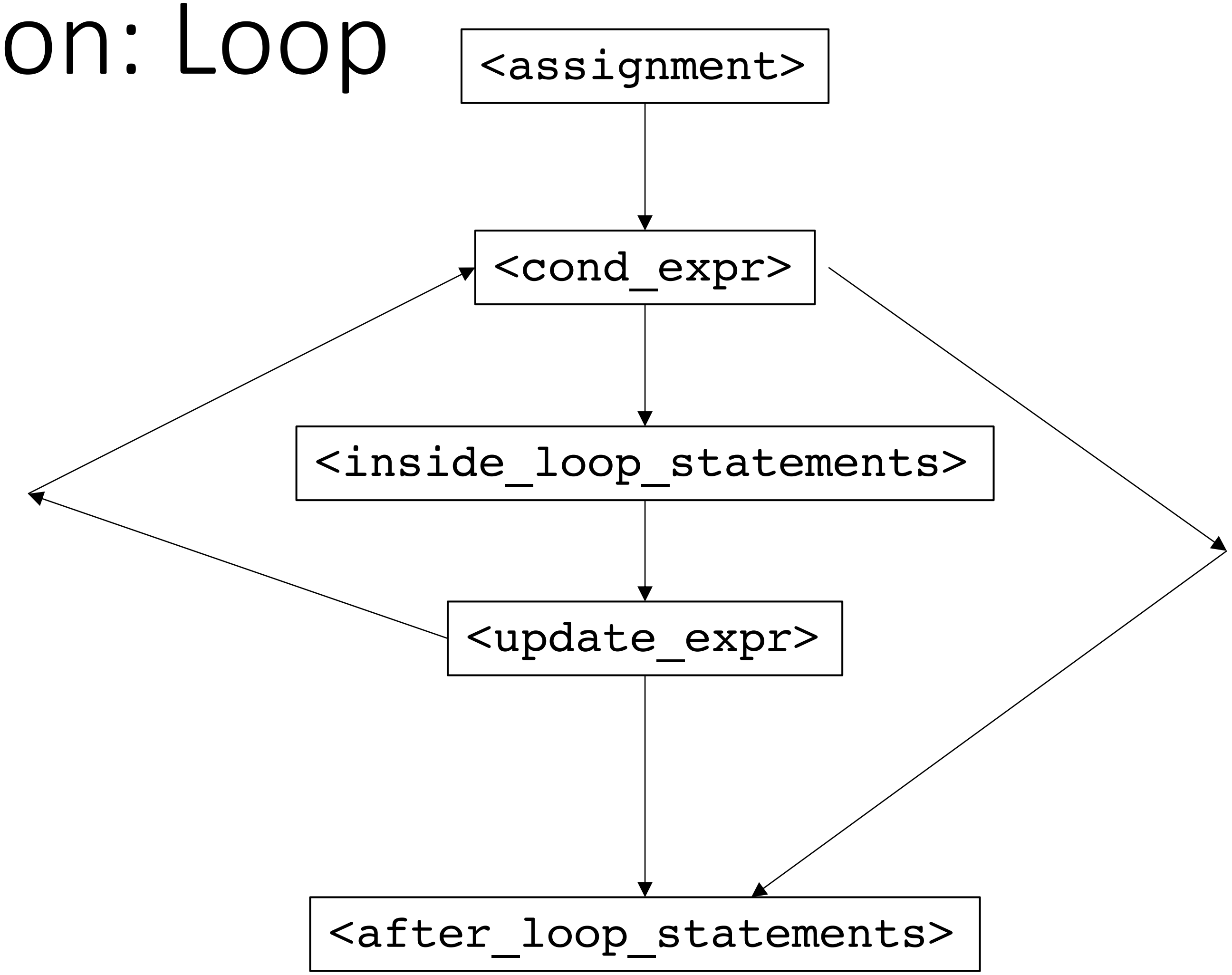
```
a2 = b0 + c1;
b4 = a2 - d3;
c5 = b4 + c1;
d6 = a2 - d3;
```

Review

- Regional optimizations:
    - Examples?

Regional optimization: Loop unrolling:

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



<assignment>

<cond_expr>

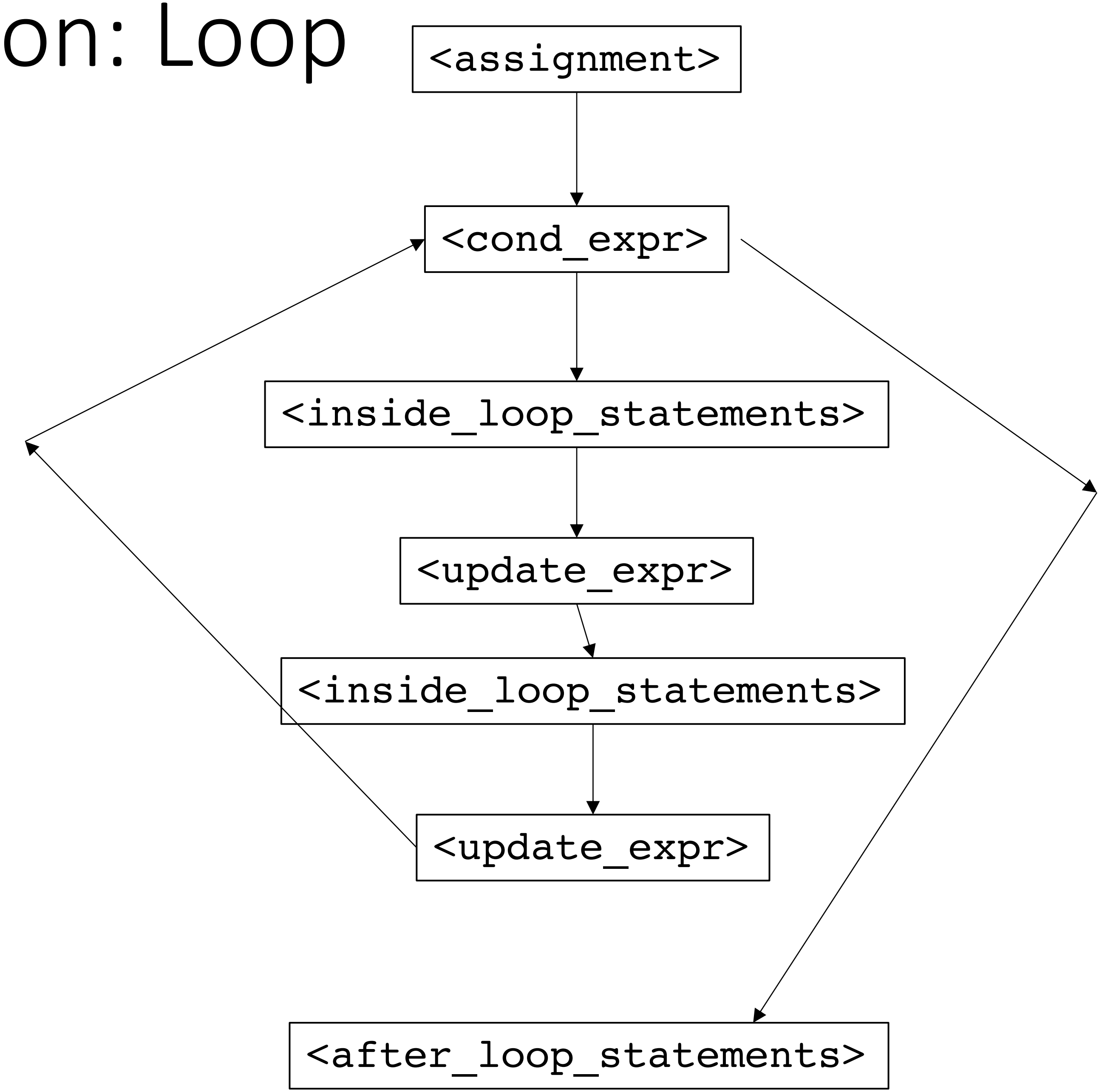
<inside_loop_statements>

<update_expr>

<after_loop_statements>

Regional optimization: Loop unrolling:

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



Review

- Global optimizations:
    - Examples?

Global optimizations

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

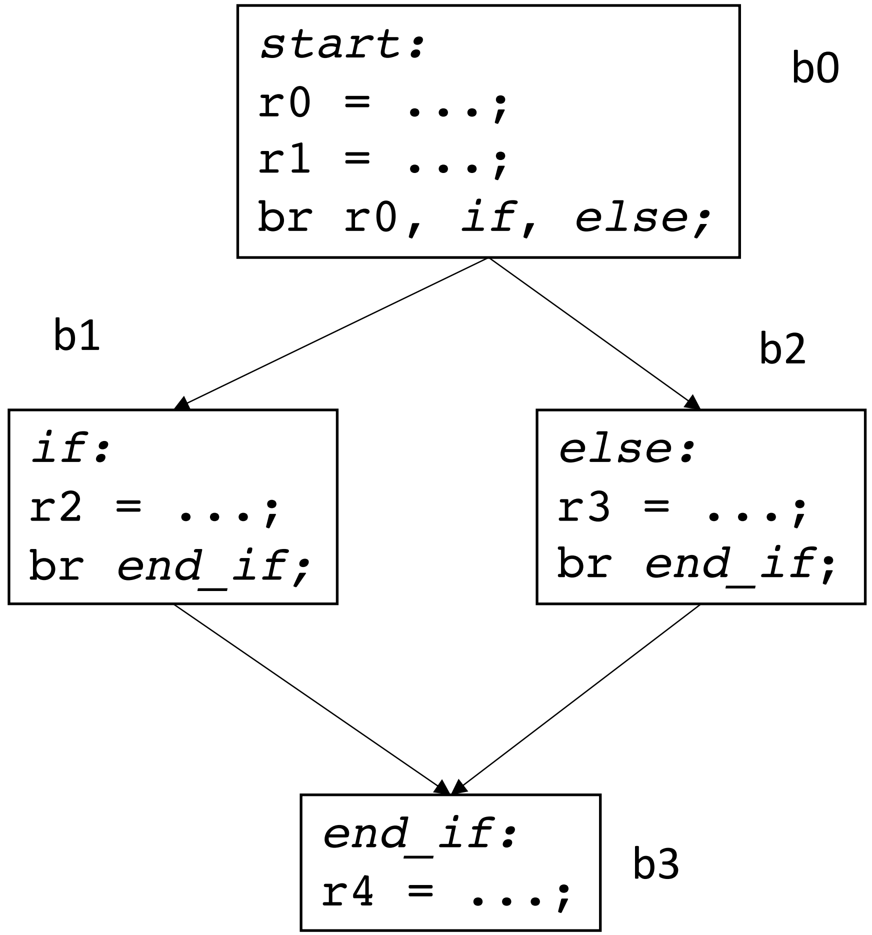
- Highly suggest reading for this part of the class
  - Chapter 9 of EAC

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



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

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

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

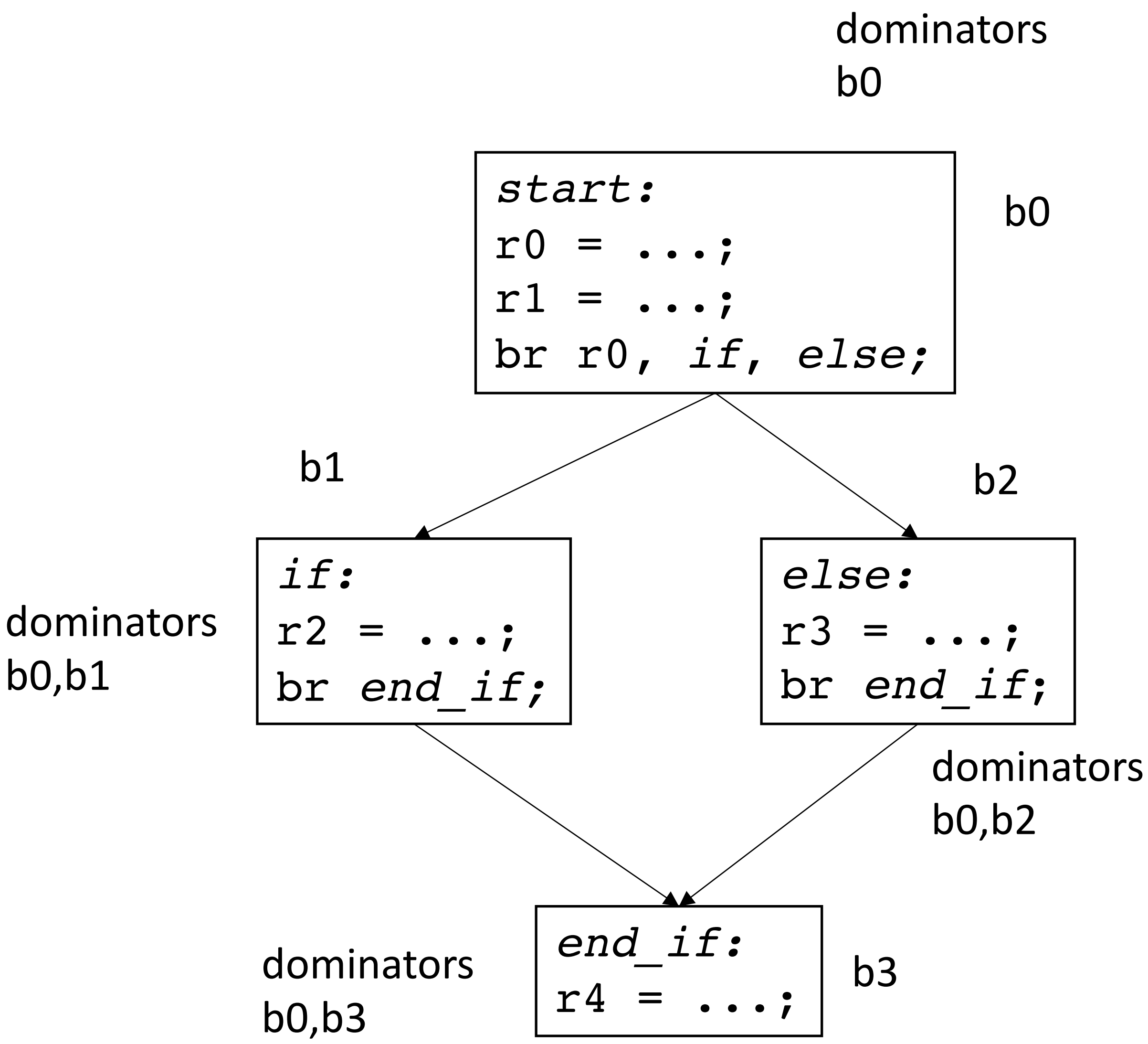
```
end_if:
r4 = ...;
```

Dominance

- a block b_x dominates block b_y iff every path from the start to block b_x goes through b_y

- definition:
  - domination (includes itself)
  - strict domination (does not include itself)

- Can we apply this to local value numbering?



dominators
b0

b0

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

b1

b2

dominators
b0,b1

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

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

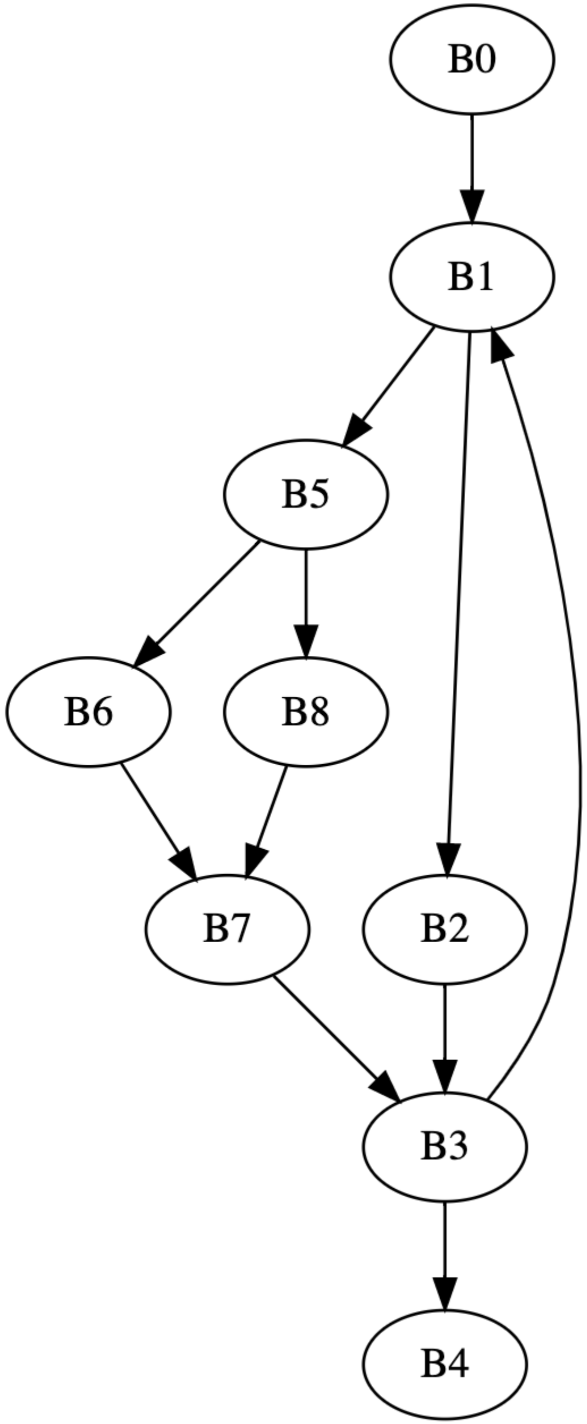
dominators
b0,b2

dominators
b0,b3

```
end_if:
r4 = ...;
```

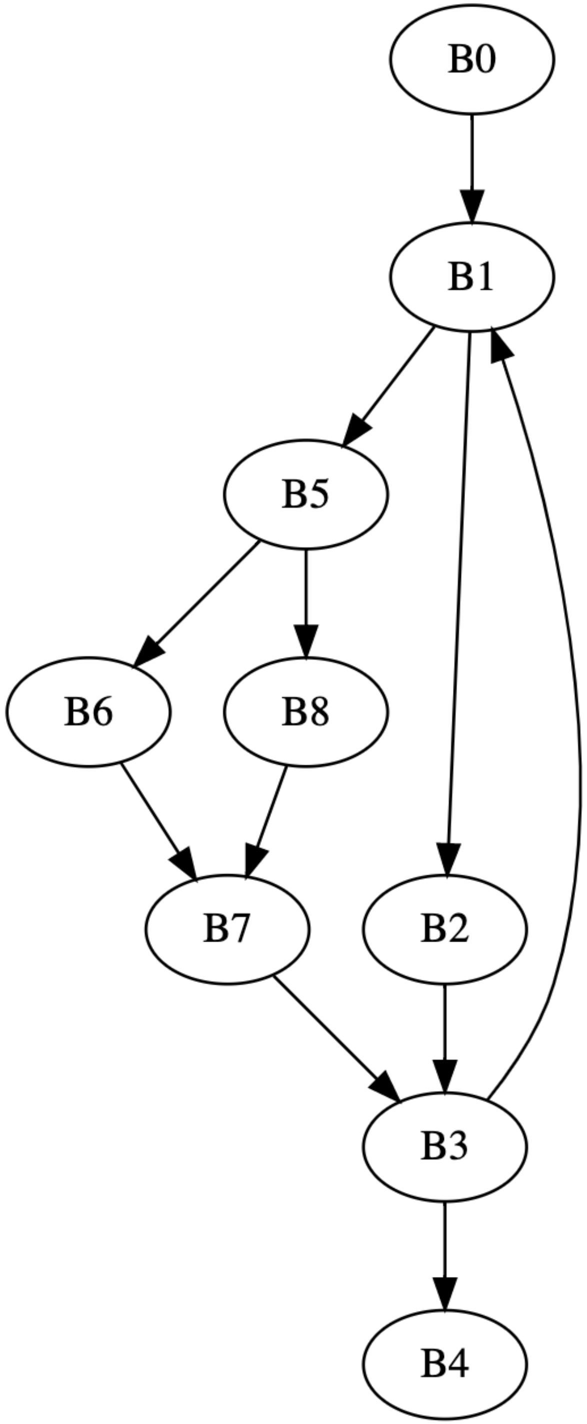
b3

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



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

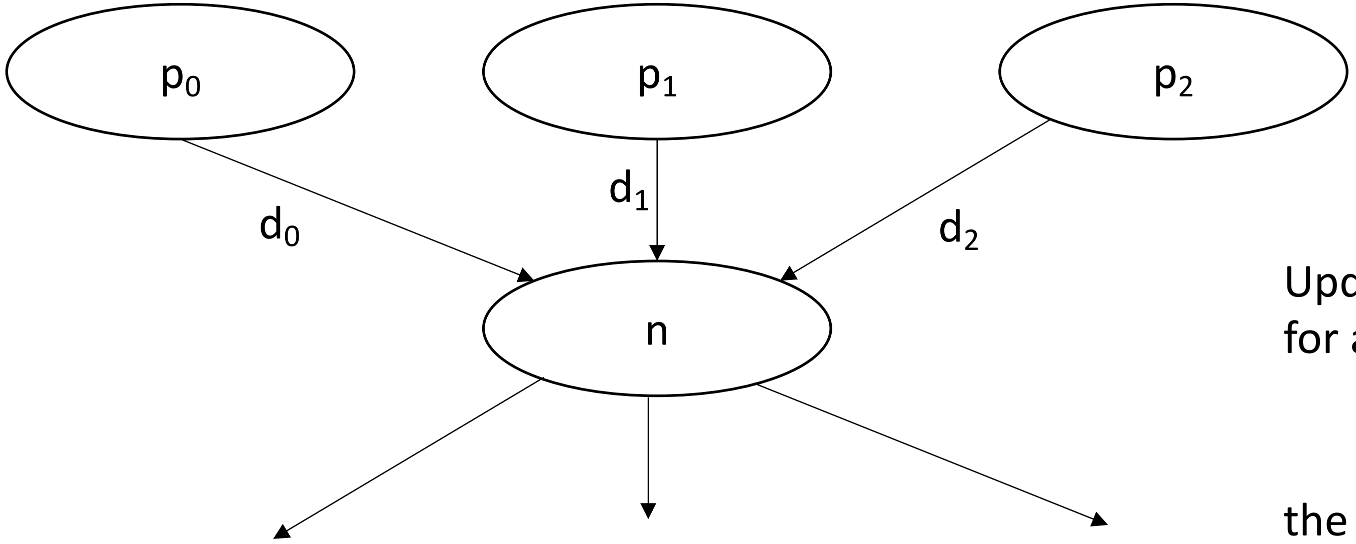
- This algorithm is vertex centric
    - local computations consider only a target node and its immediate neighbors

- 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

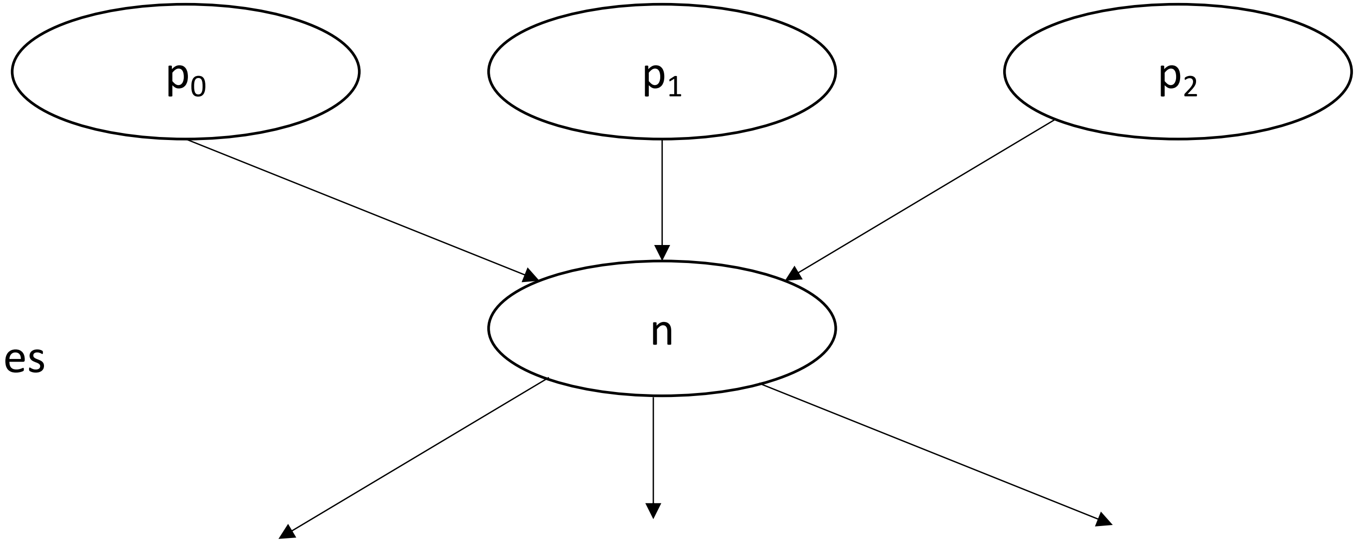


Update:
for all parents *p*: min(*p* + *d*)

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

Now lets think about dominance

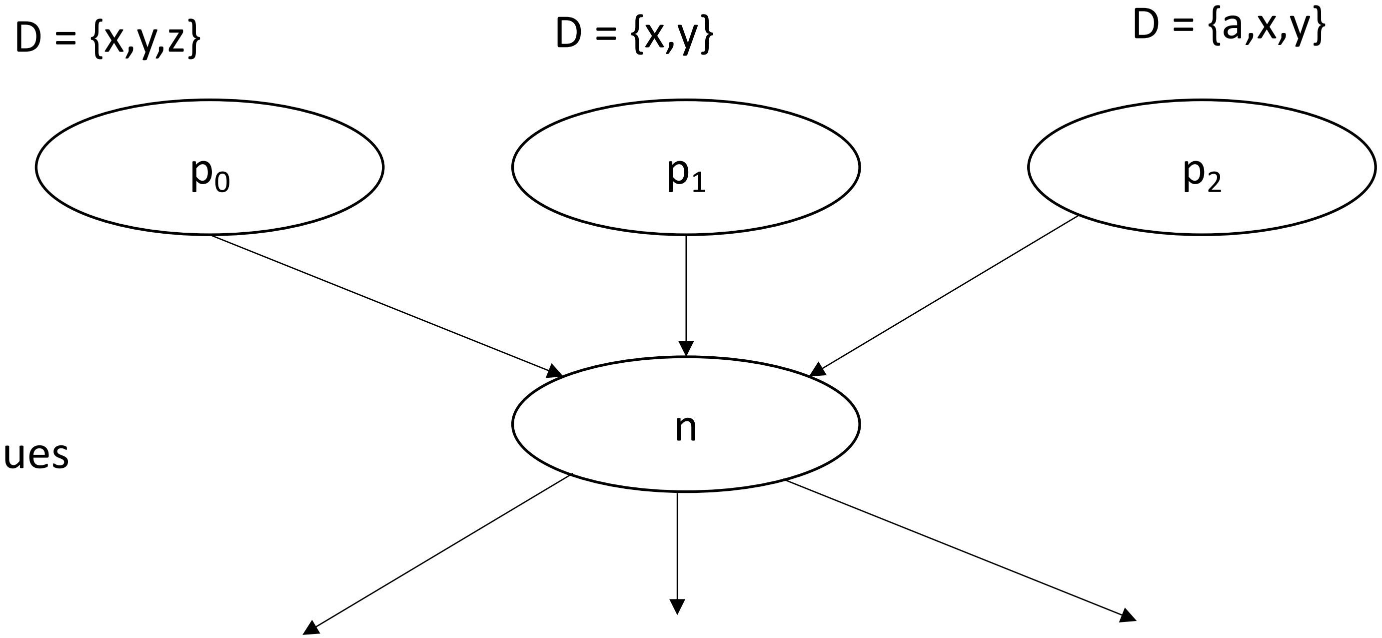
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

Now lets think about dominance

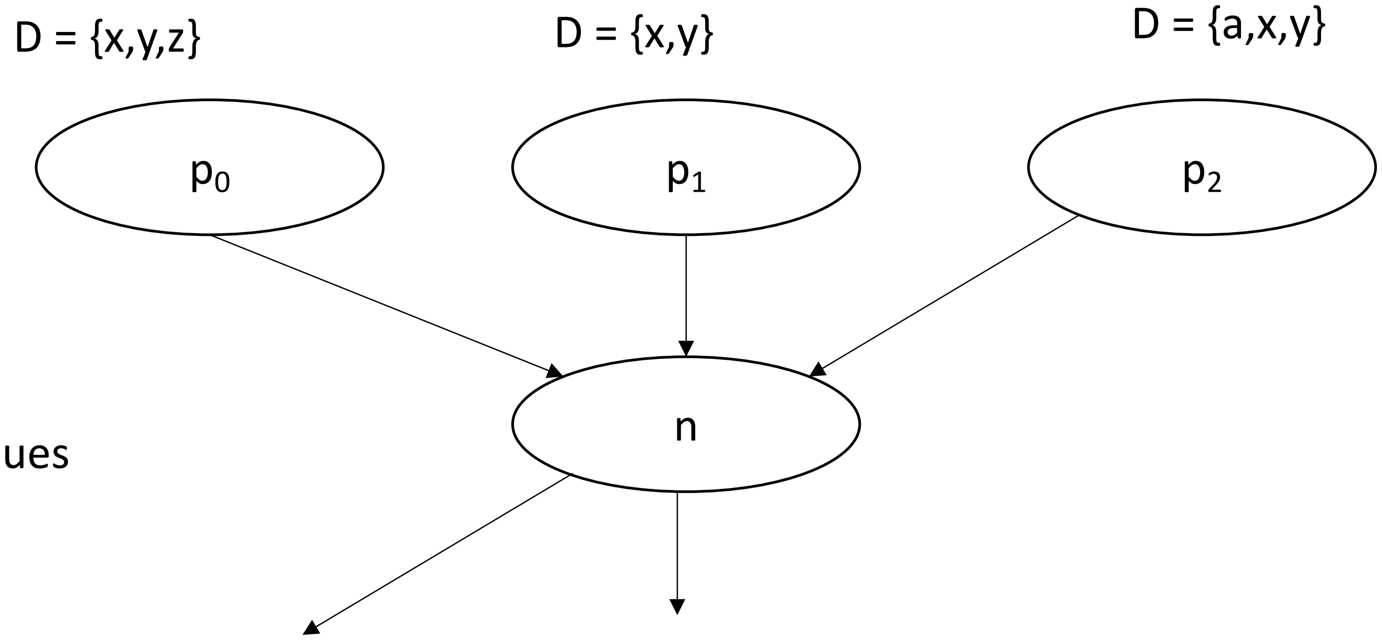
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



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



D = {x,y,z}

D = {x,y}

D = {a,x,y}

p_0

p_1

p_2

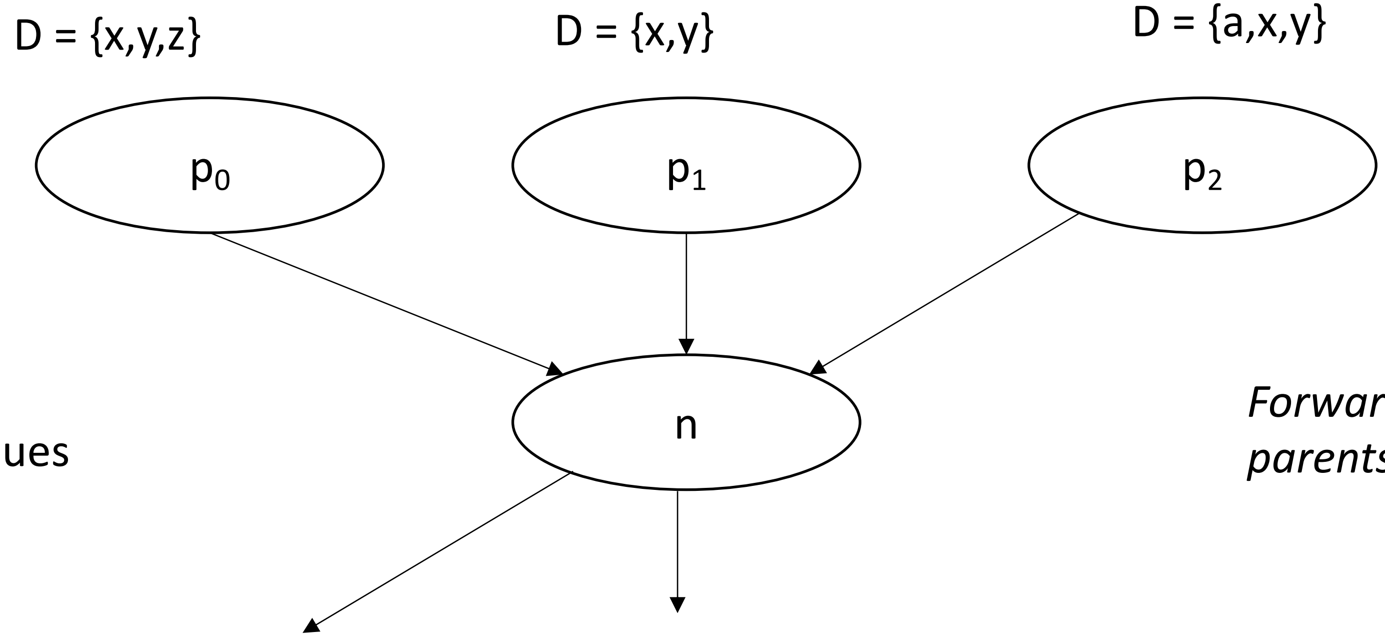
n

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



D = {x,y,z}

D = {x,y}

D = {a,x,y}

p_0

p_1

p_2

n

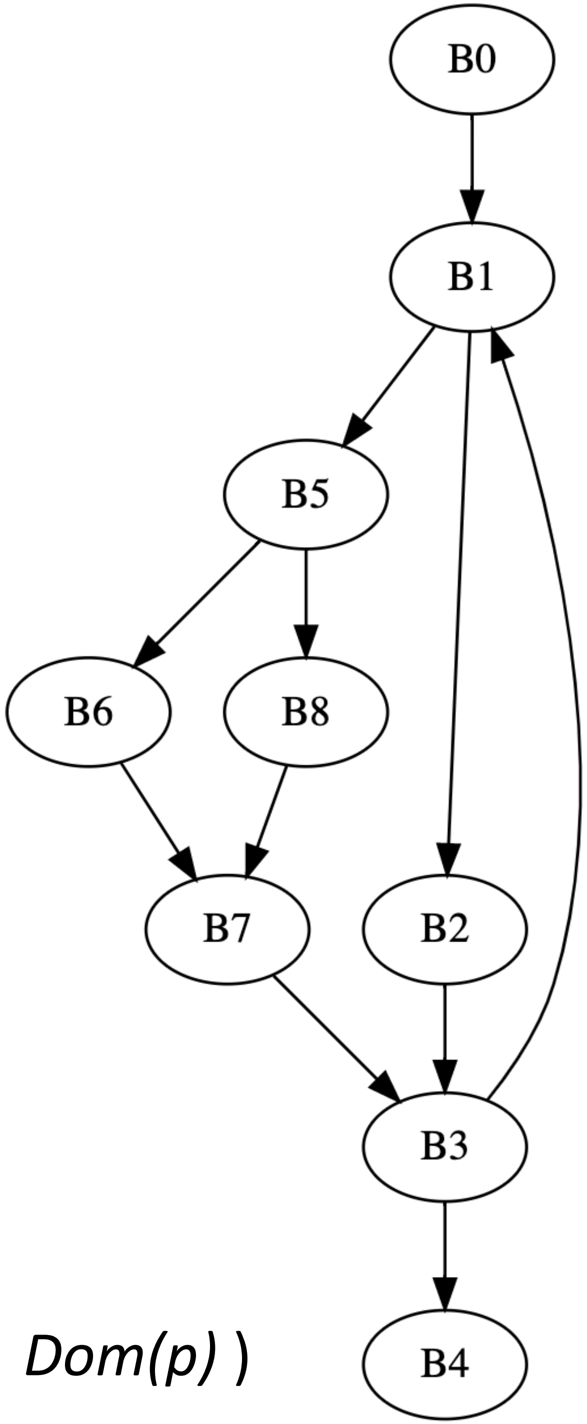
update:
intersection of parent values

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

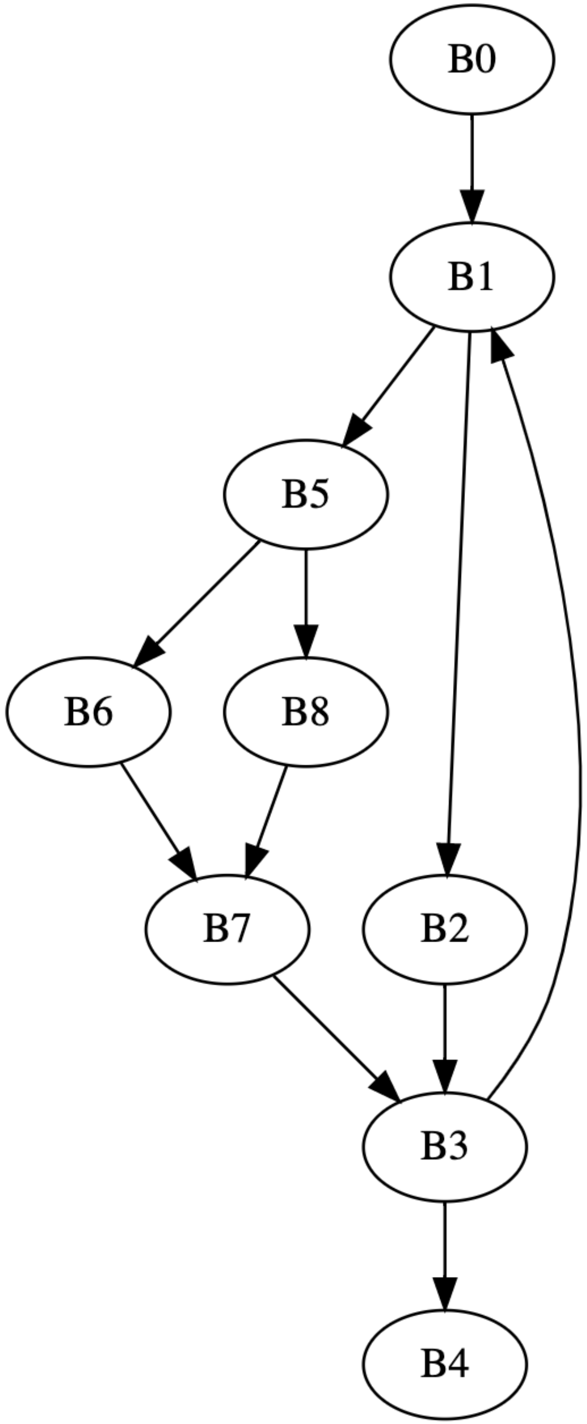
Node	Initial	Iteration 1
B0	B0	…
B1	*N*	B0, B1
B2	*N*	B0, B1, B2
B3	*N*	B0, B1, B2, B3
B4	*N*	B0, B1, B2, B3, B4
B5	*N*	B0, B1, B5
B6	*N*	B0, B1, B5, B6
B7	*N*	B0, B1, B5, B6, B7
B8	*N*	B0, B1, B5, B8



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

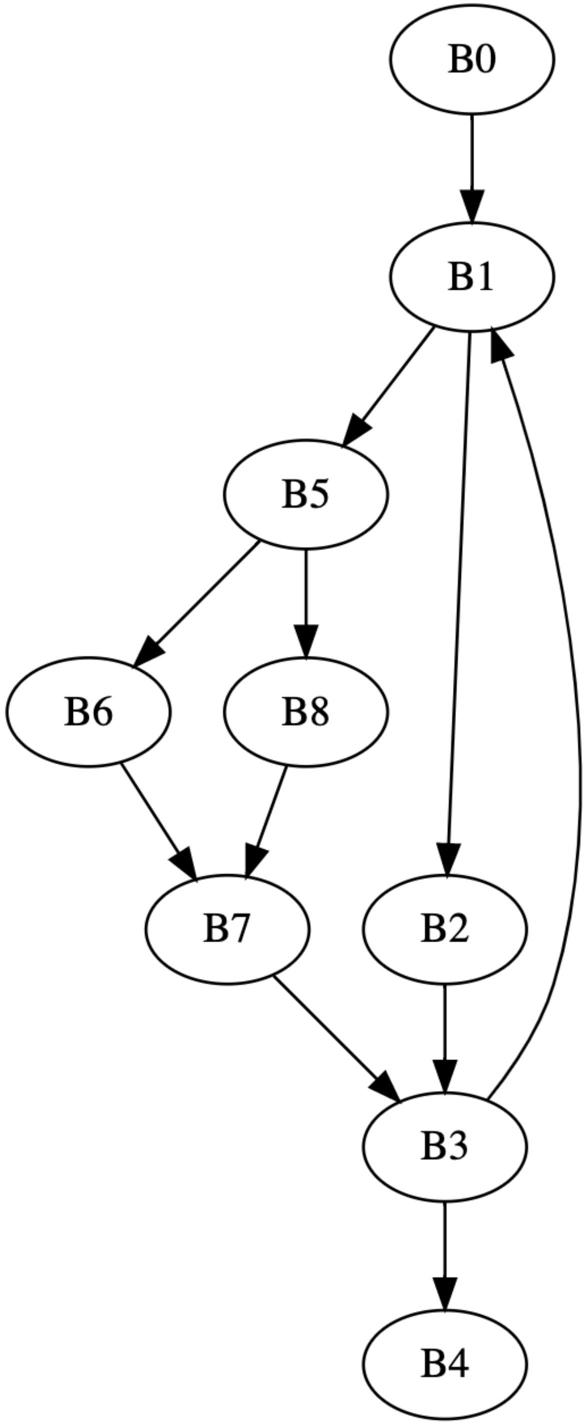
Lets try it

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	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	...	...

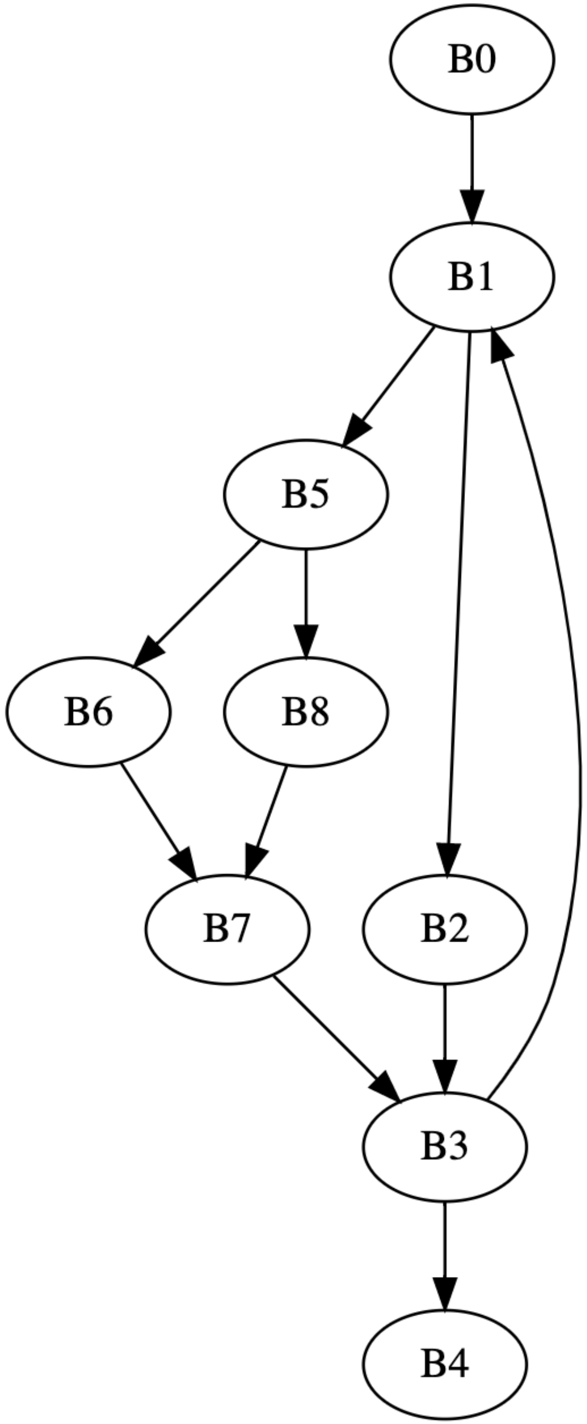


How can we optimize the algorithm?

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	…	…

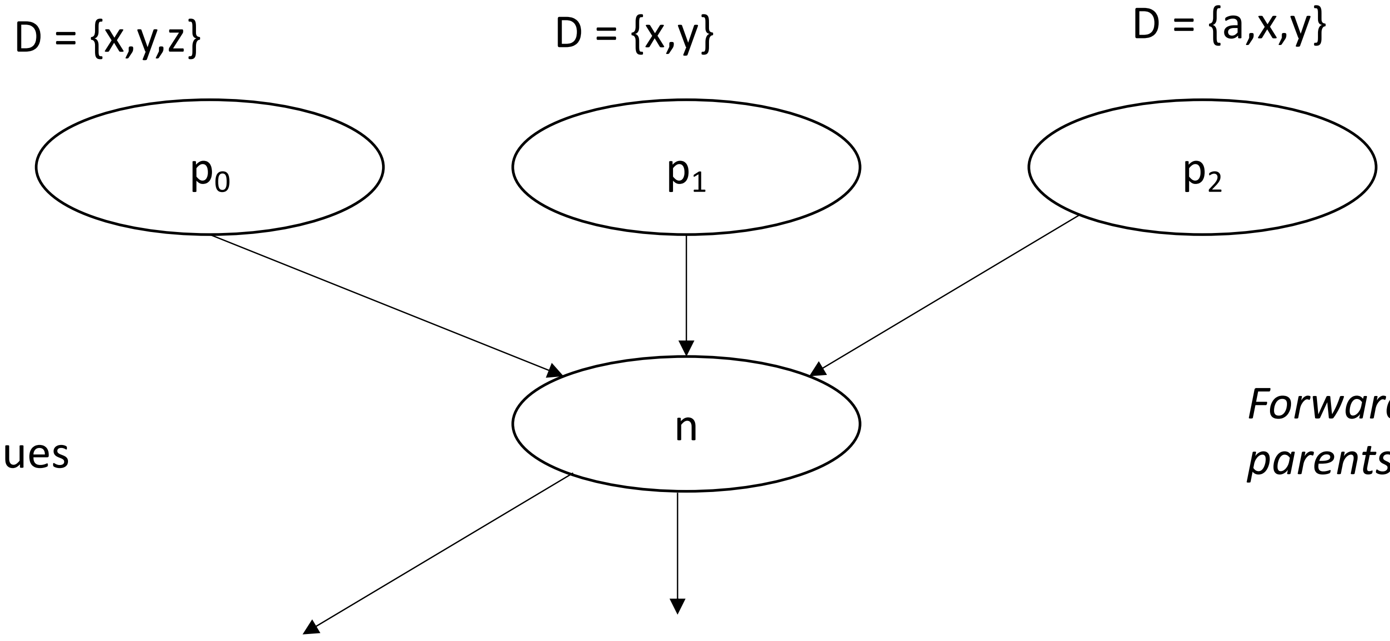
This can be any order...

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

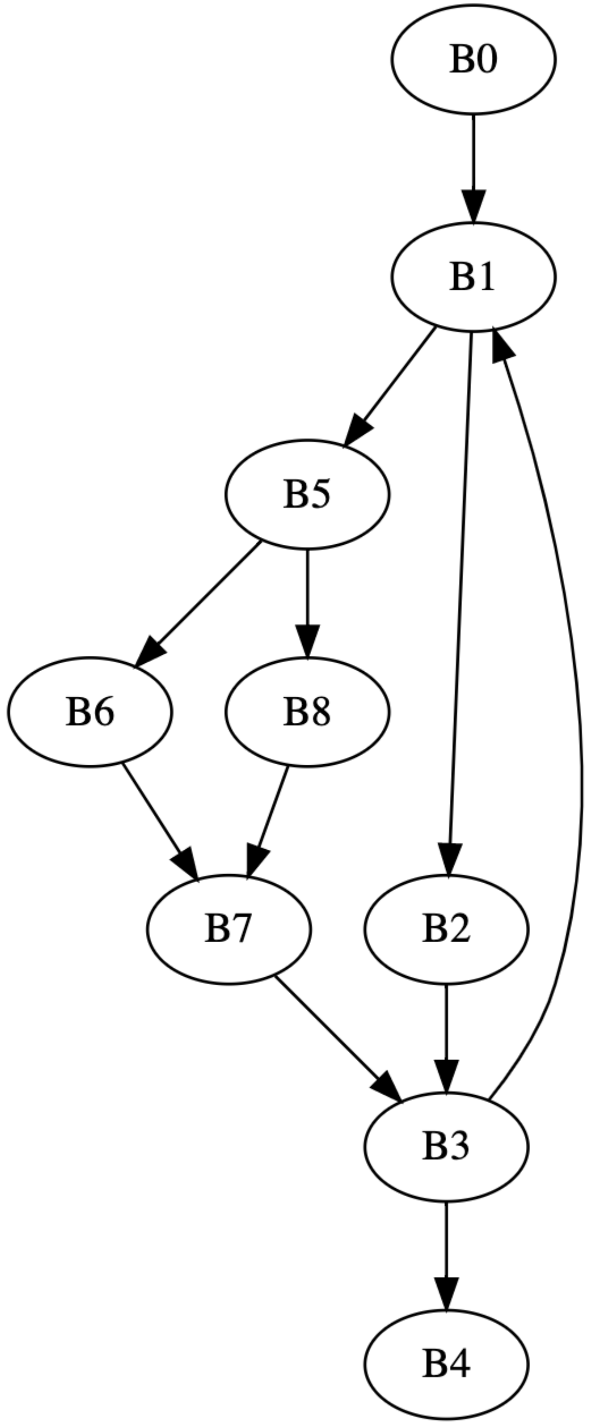
Forward flow, as updates flow from parents to children.

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

How can we optimize the algorithm?

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

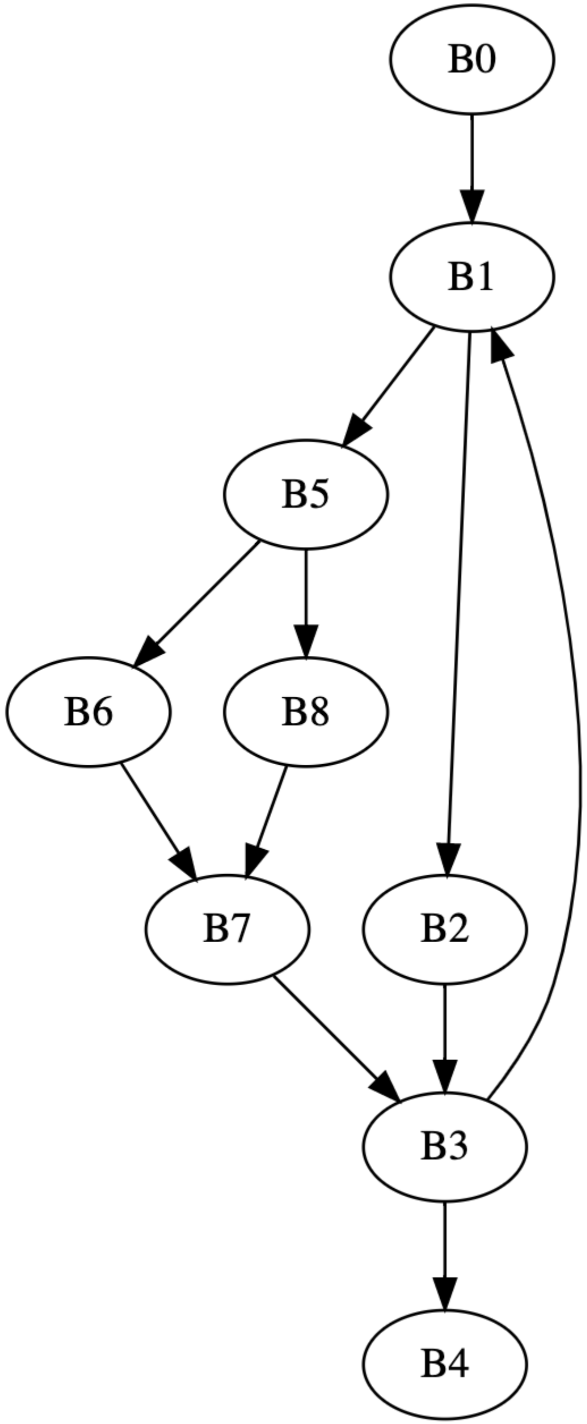
Reverse
post-order (rpo),
where parents are visited
first



How can we optimize the algorithm?

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

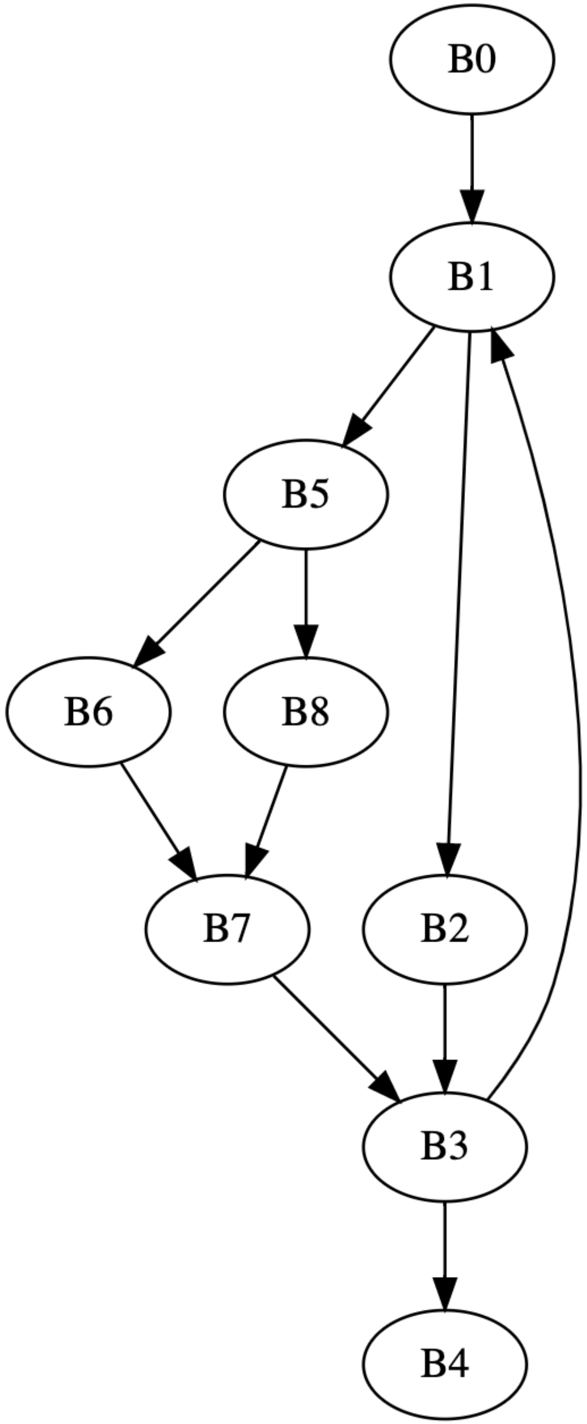
Reverse
post-order (rpo),
where parents are visited
first



How can we optimize the algorithm?

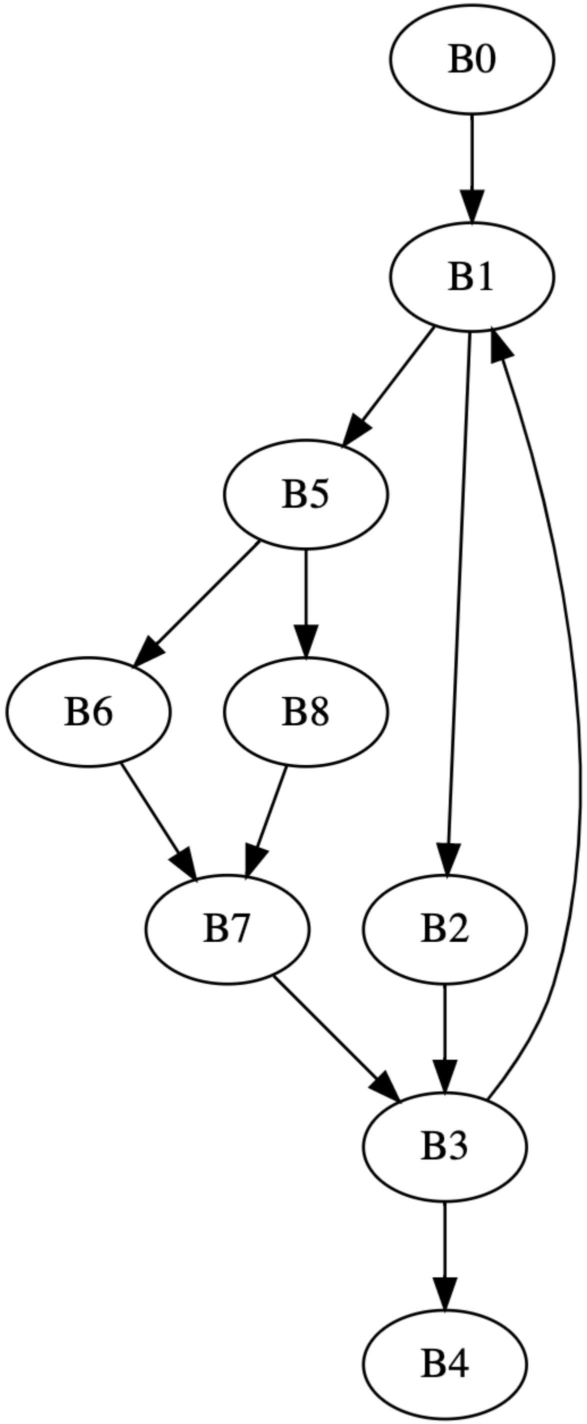
Node	New Order
B0	B0
B1	B1
B2	B2
B3	B5
B4	B6
B5	B8
B6	B7
B7	B3
B8	B4

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



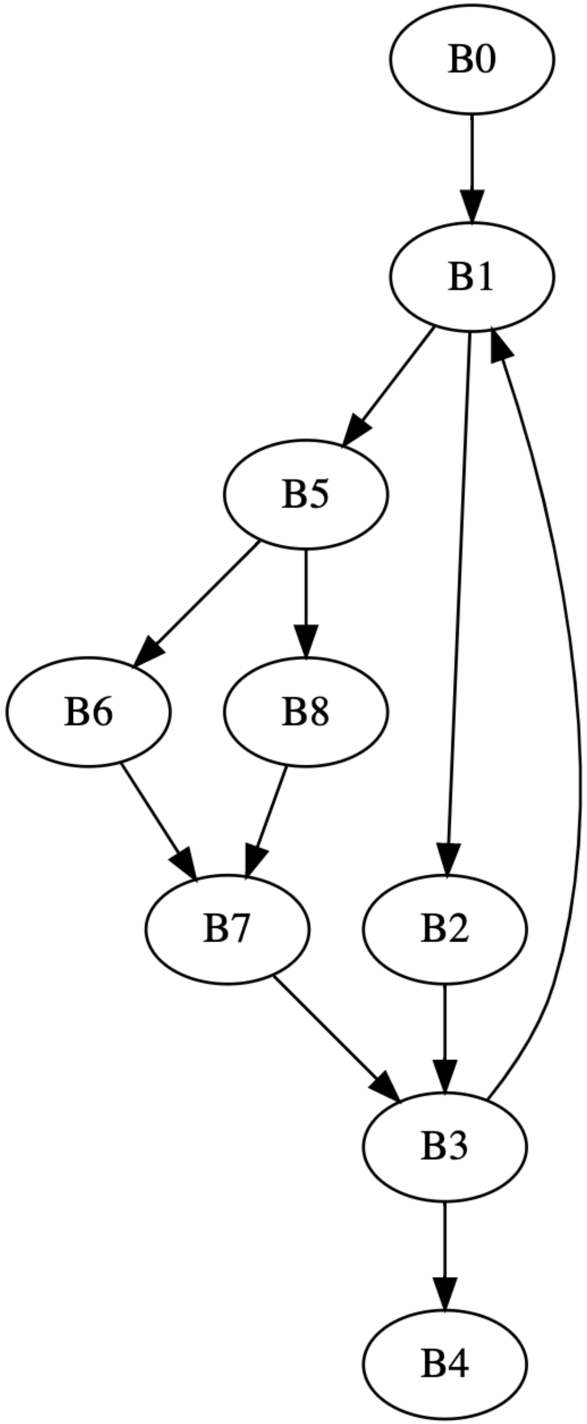
How can we optimize the algorithm?

Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	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,B3		



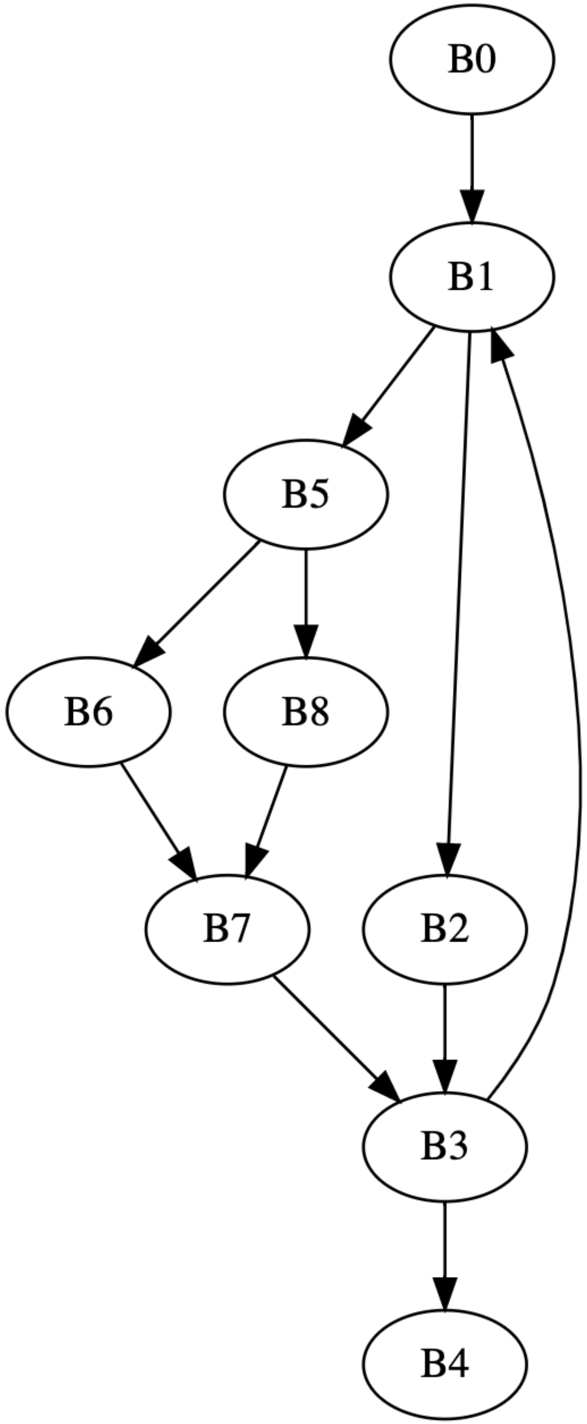
How can we optimize the algorithm?

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		



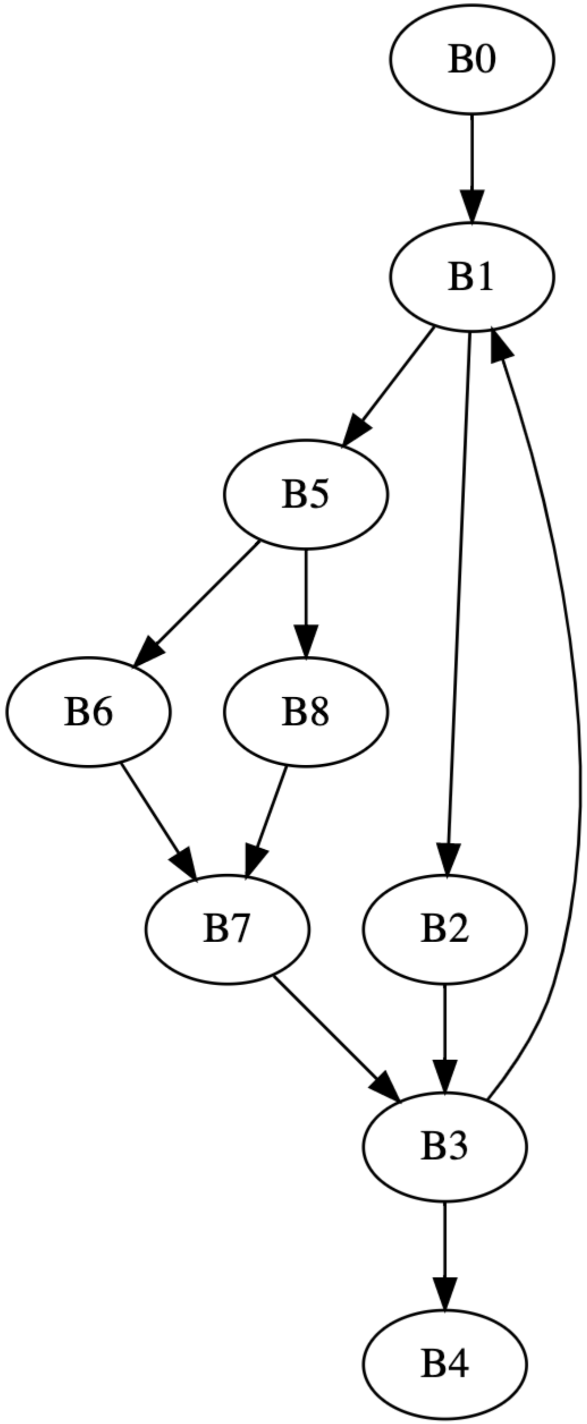
How can we optimize the algorithm?

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		



How can we optimize the algorithm?

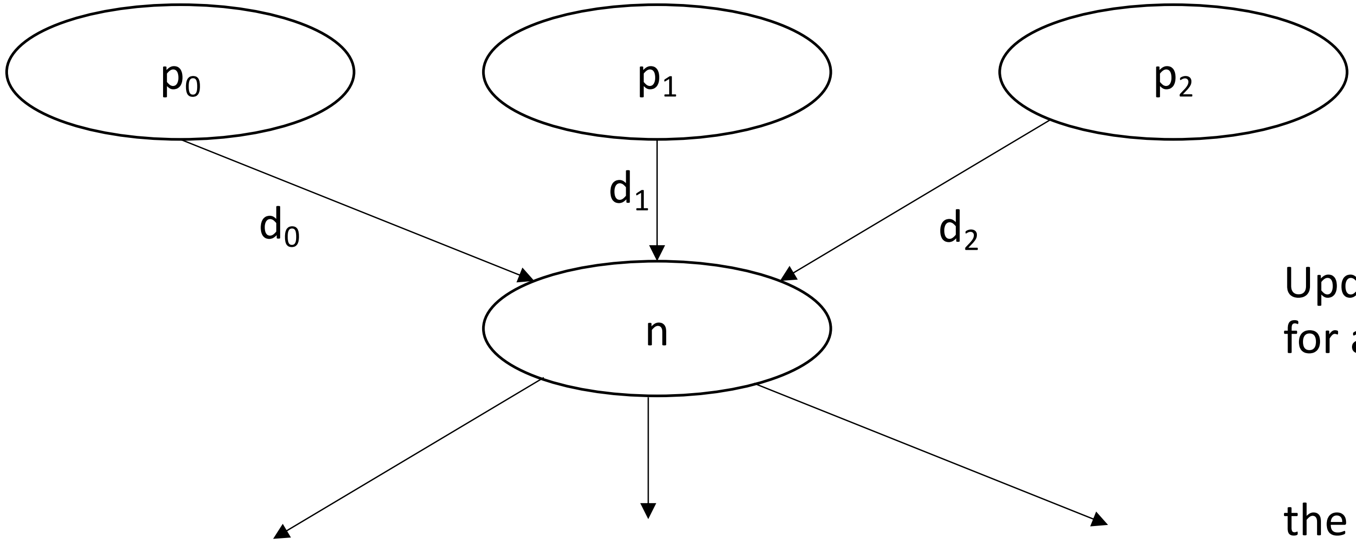
Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	...	
B1	*N*	B0,B1	...	
B2	*N*	B0,B1,B2	...	
B5	*N*	B0,B1,B5	...	
B6	*N*	B0,B1,B5,B6	...	
B8	*N*	B0,B1,B5,B8	...	
B7	*N*	B0,B1,B5,B7	...	
B3	*N*	B0,B1,B3	...	
B4	*N*	B0,B1,B4	...	



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

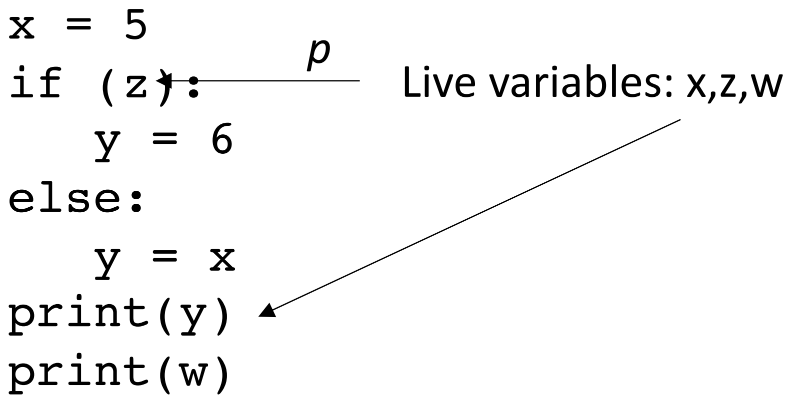
Live variables: z, w

```
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)
```

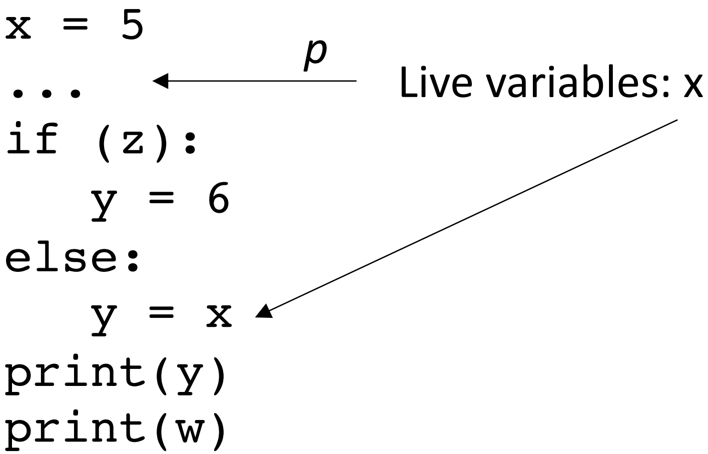
p

Live variables: x,z,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:



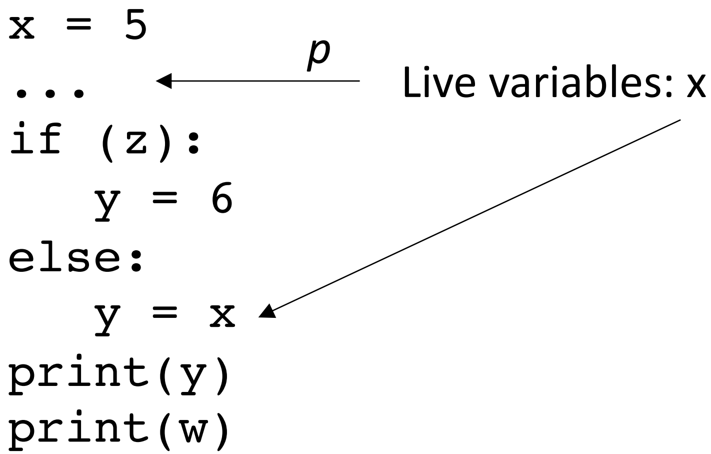
```
x = 5
...
if (z):
   y = 6
else:
   y = x
print(y)
print(w)
```

```
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

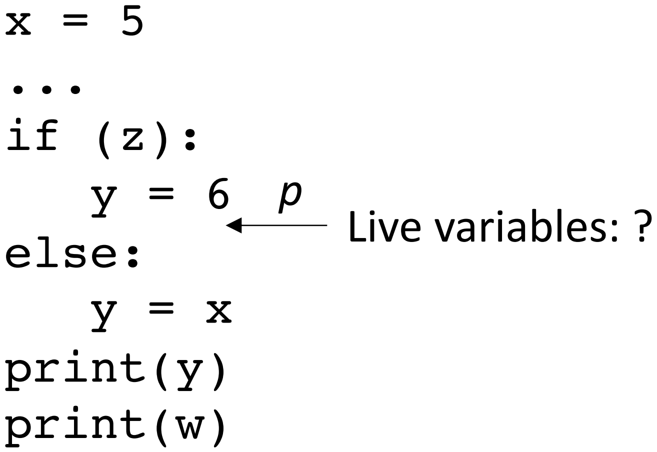
- examples:



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

p

Live variables: x



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

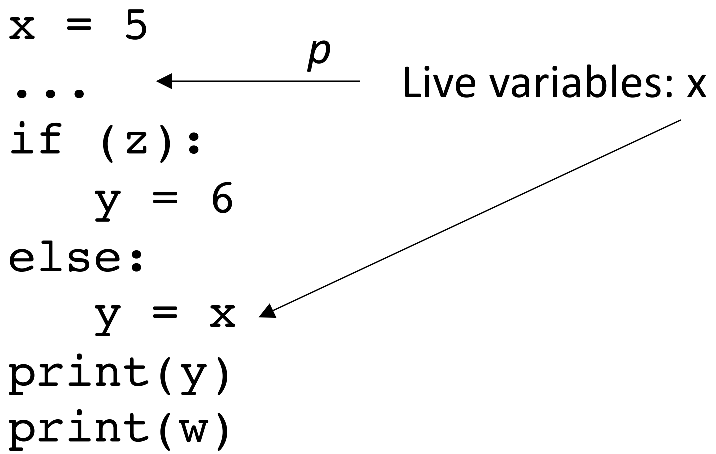
p

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

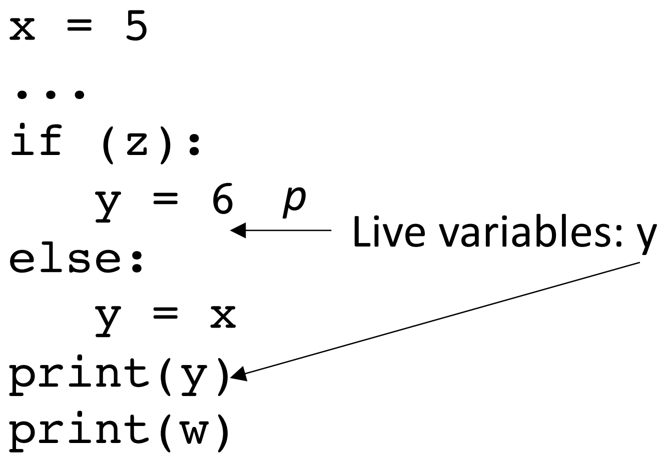
- examples:



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

p

Live variables: x



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

p

Live variables: y

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

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

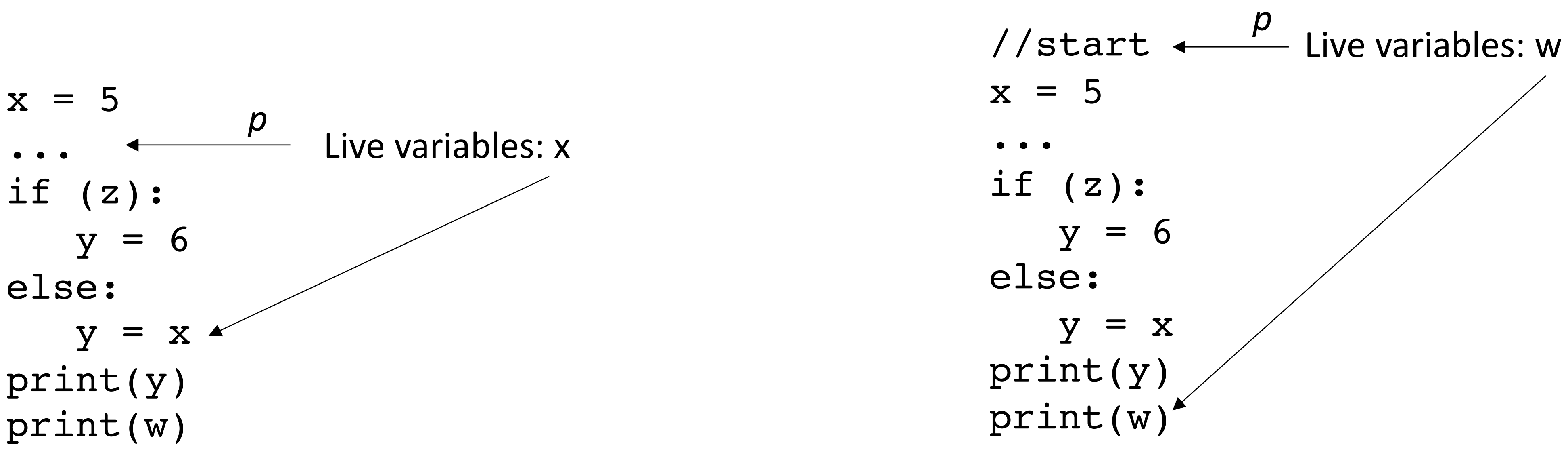
p

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

p

Live variables: x

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

p

Live variables: 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

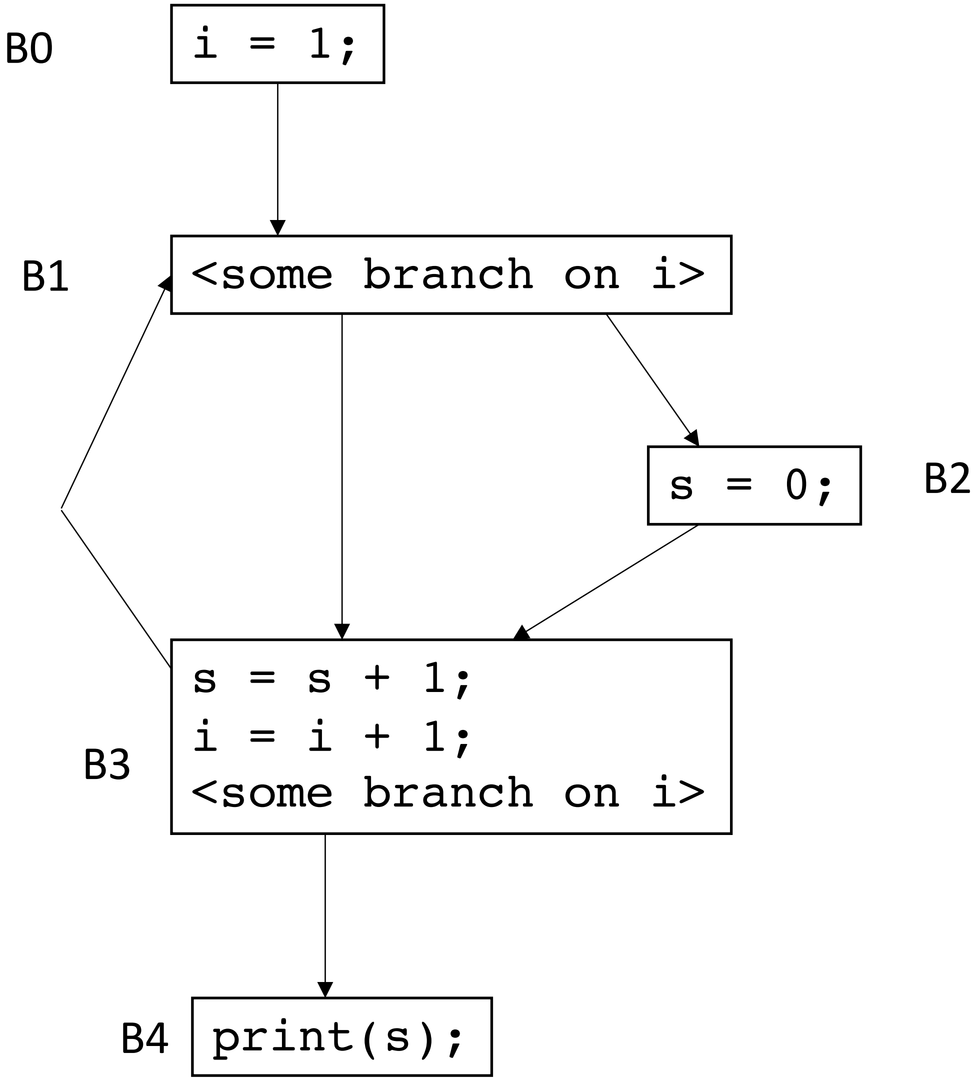
Accessing an uninitialized variable!

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

p

Live variables: w

Live variable analysis in the CFG:



B0
```
i = 1;
```

B1
```
<some branch on i>
```

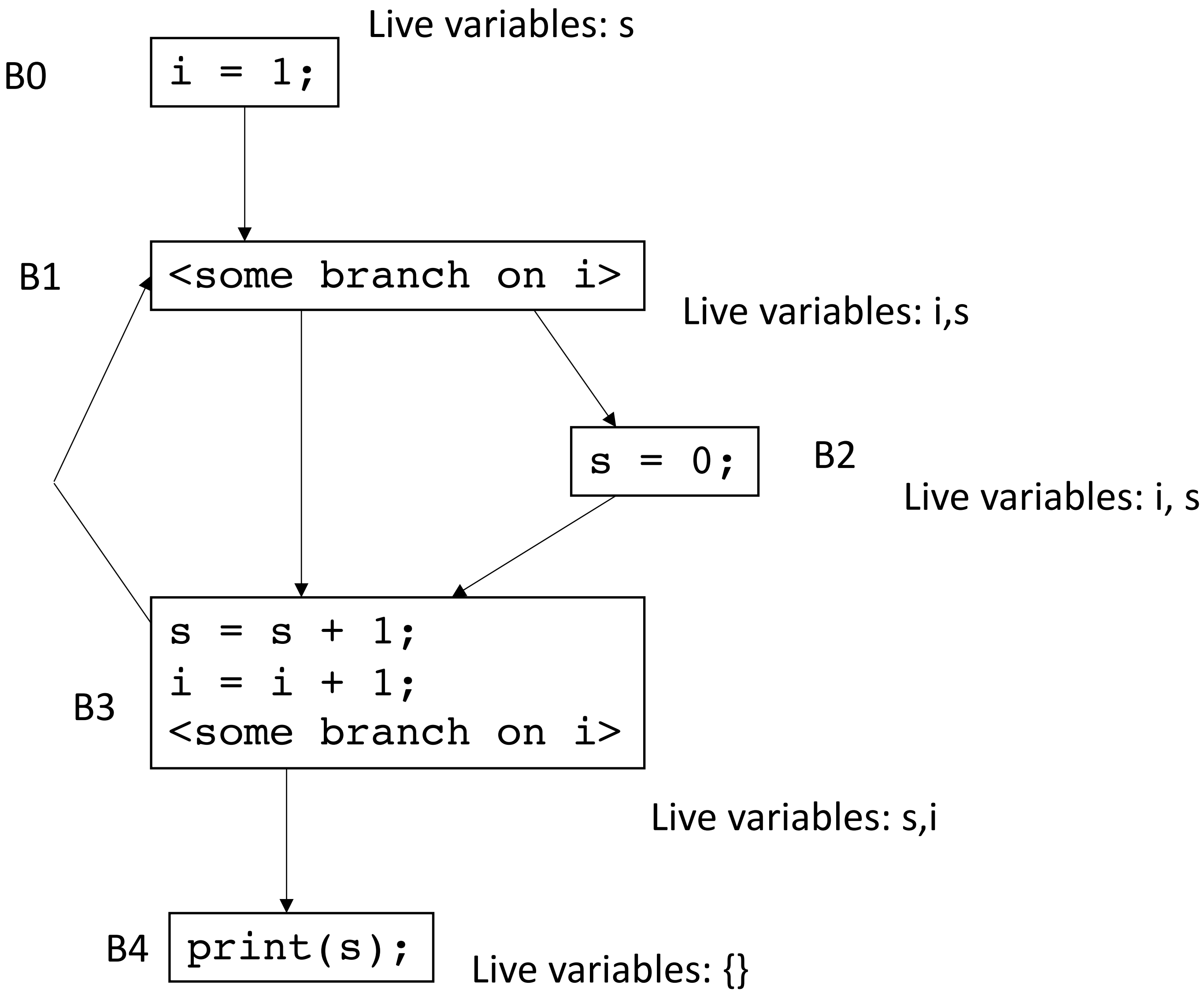
B2
```
s = 0;
```

B3
```
s = s + 1;
i = i + 1;
<some branch on i>
```

B4
```
print(s);
```

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 variables: s

B0

```
i = 1;
```

B1

```
<some branch on i>
```

Live variables: i,s

```
s = 0;
```

B2

Live variables: i, s

B3

```
s = s + 1;
i = i + 1;
<some branch on i>
```

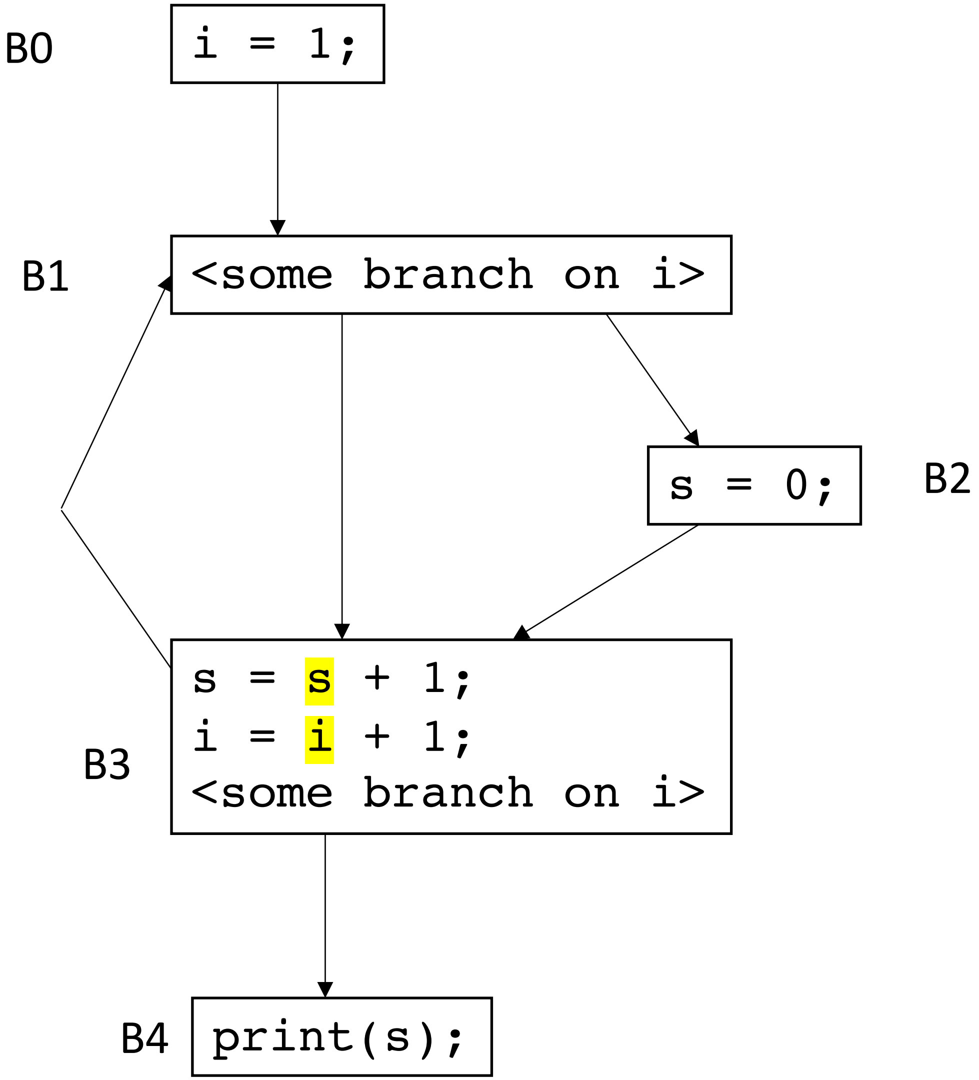
Live variables: s,i

B4

```
print(s);
```

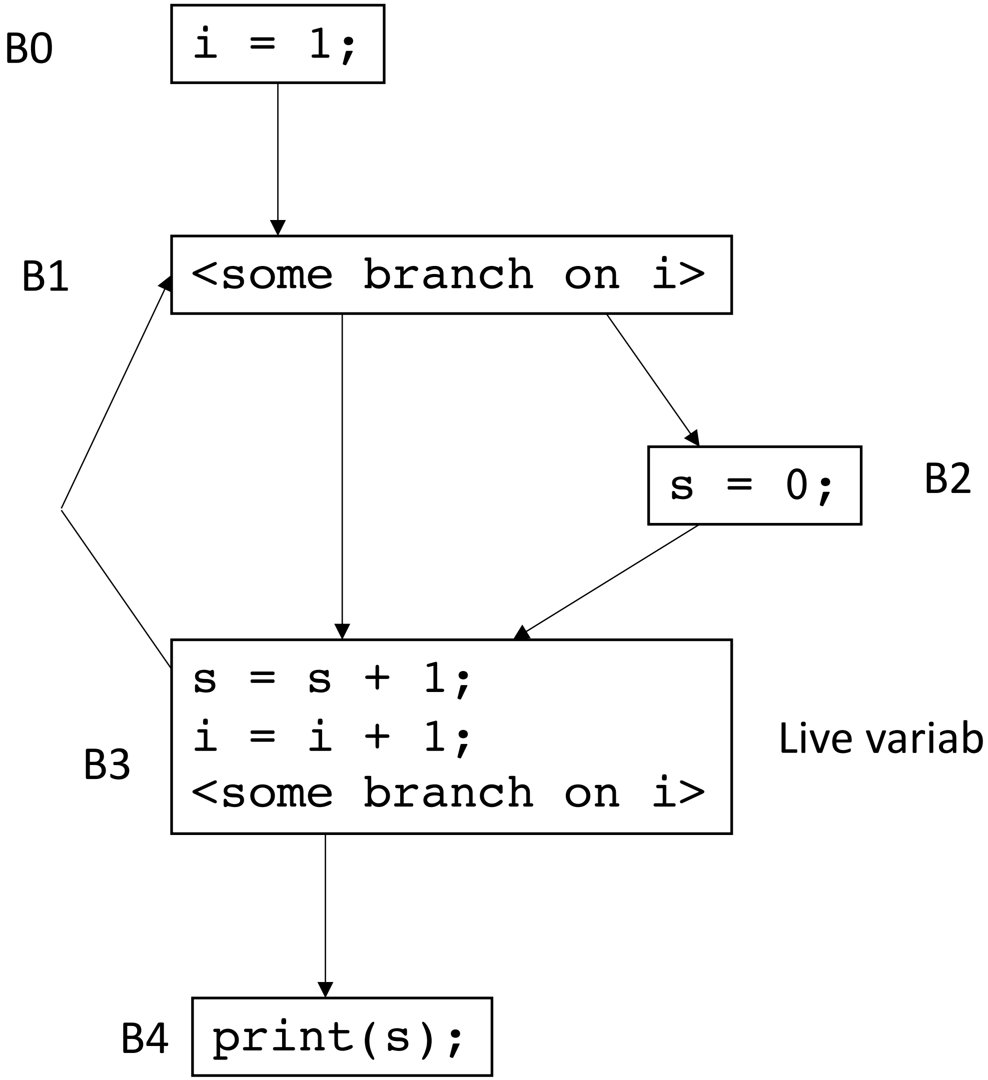
Live variables: {}

Live variable analysis in the CFG:



Live variables: i, s

Live variable analysis in the CFG:



```
i = 1;
```

```
<some branch on i>
```

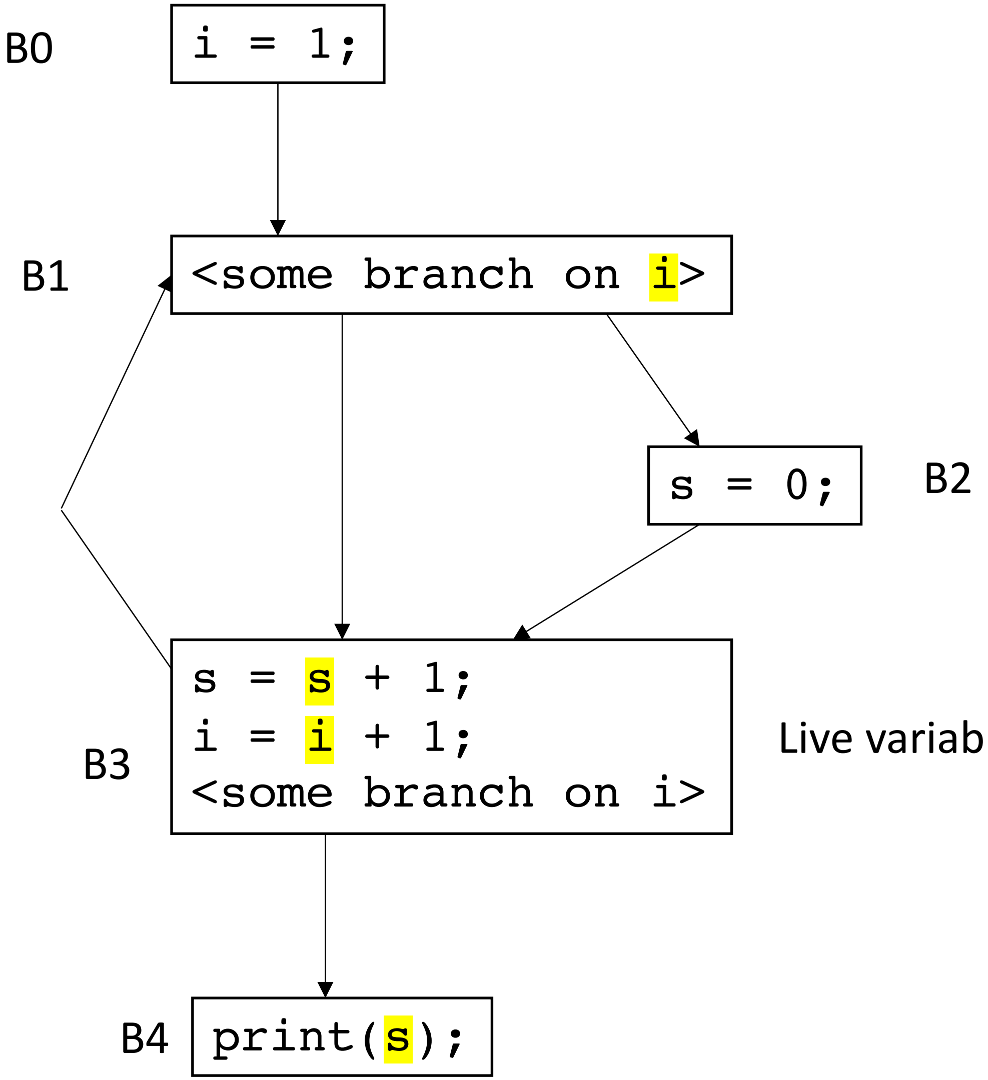
```
s = 0;
```

```
s = s + 1;
i = i + 1;
<some branch on i>
```

```
print(s);
```

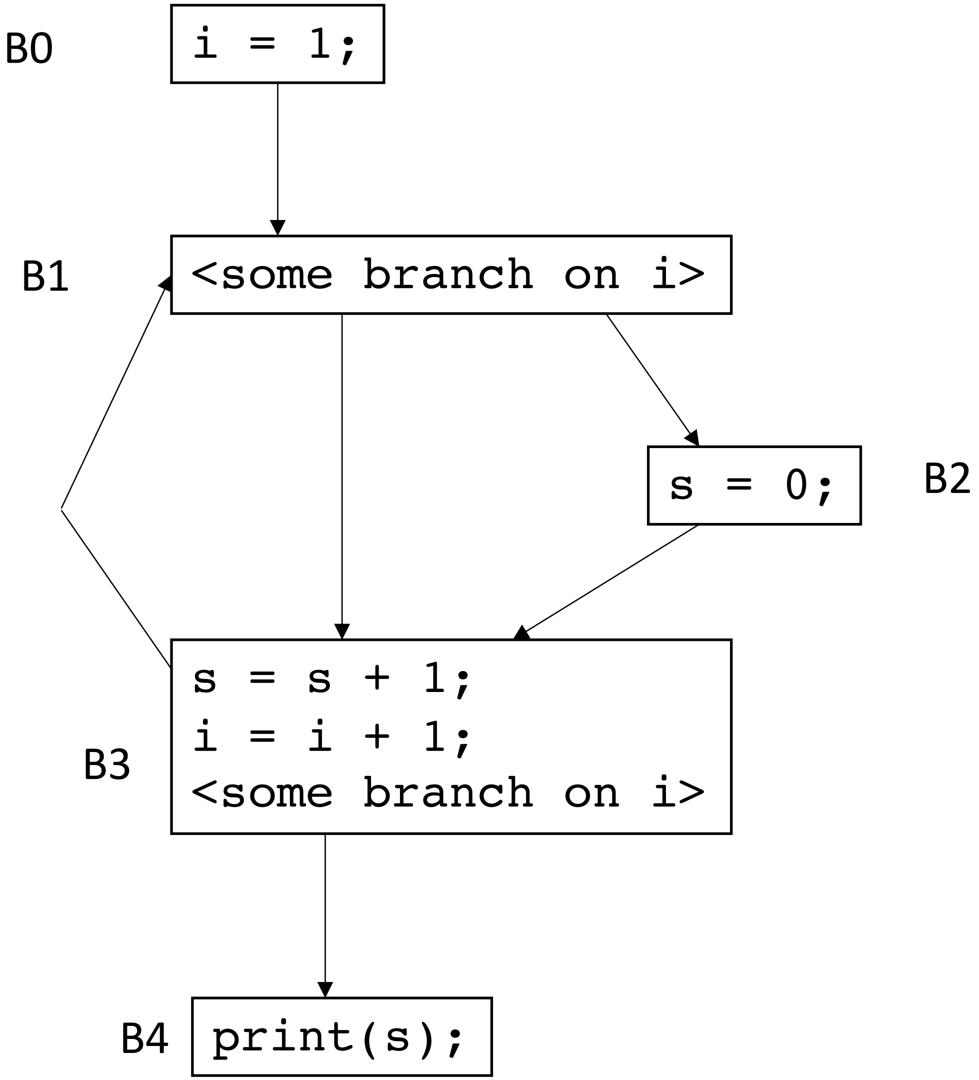
Live variables: i, s

Live variable analysis in the CFG:



Live variables: i, s

Live variable analysis in the CFG:



B0 `i = 1;`

B1 `<some branch on i>`

B2 `s = 0;`

B3
```
s = s + 1;
i = i + 1;
<some branch on i>
```

B4 `print(s);`

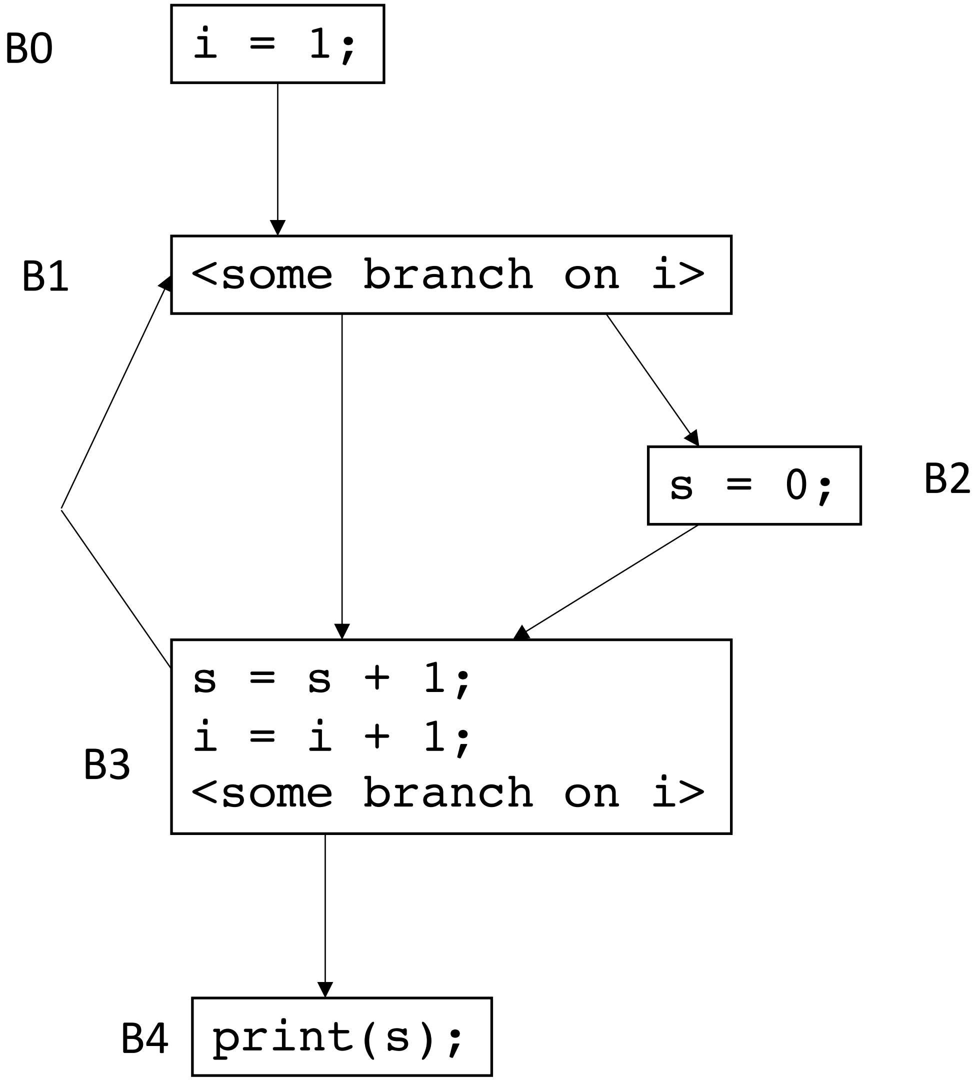
To compute the LiveOut sets, we need two initial sets:

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 read before being overwritten

Block	VarKill	UEVar
B0		
B1		
B2		
B3		
B4		

Live variable analysis in the CFG:



```
i = 1;
```

```
<some branch on i>
```

```
s = 0;
```

```
s = s + 1;
i = i + 1;
<some branch on i>
```

```
print(s);
```

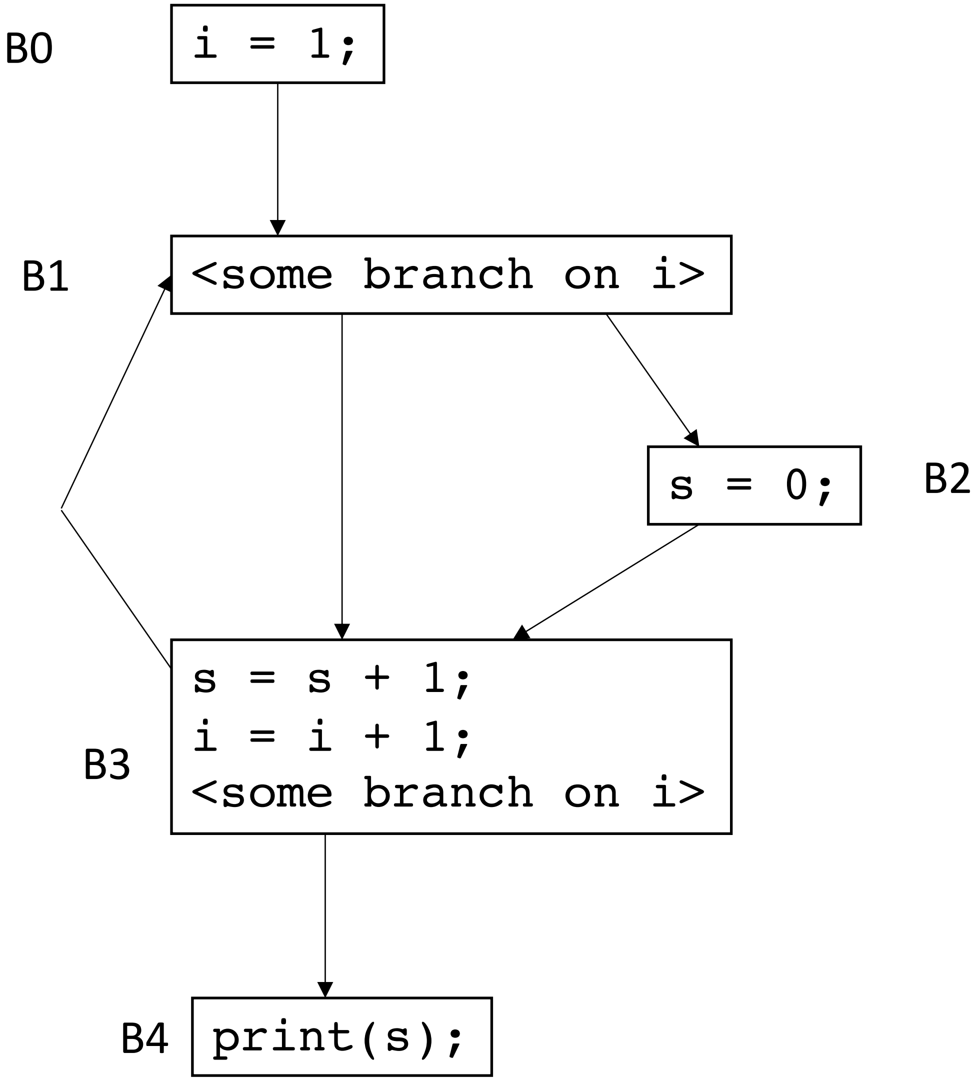
To compute the LiveOut sets, we need two initial sets:

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 read before being overwritten

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

Live variable analysis in the CFG:



B0
```
i = 1;
```

B1
```
<some branch on i>
```

B2
```
s = 0;
```

B3
```
s = s + 1;
i = i + 1;
<some branch on i>
```

B4
```
print(s);
```

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 read before being overwritten

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:

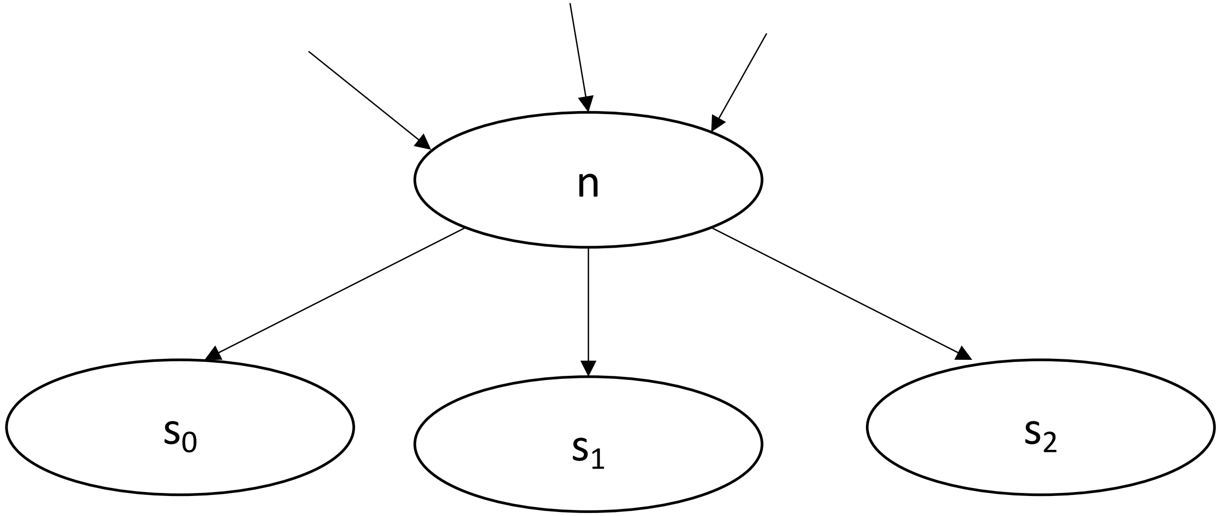
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed point computation:

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

Live variable analysis in the CFG:

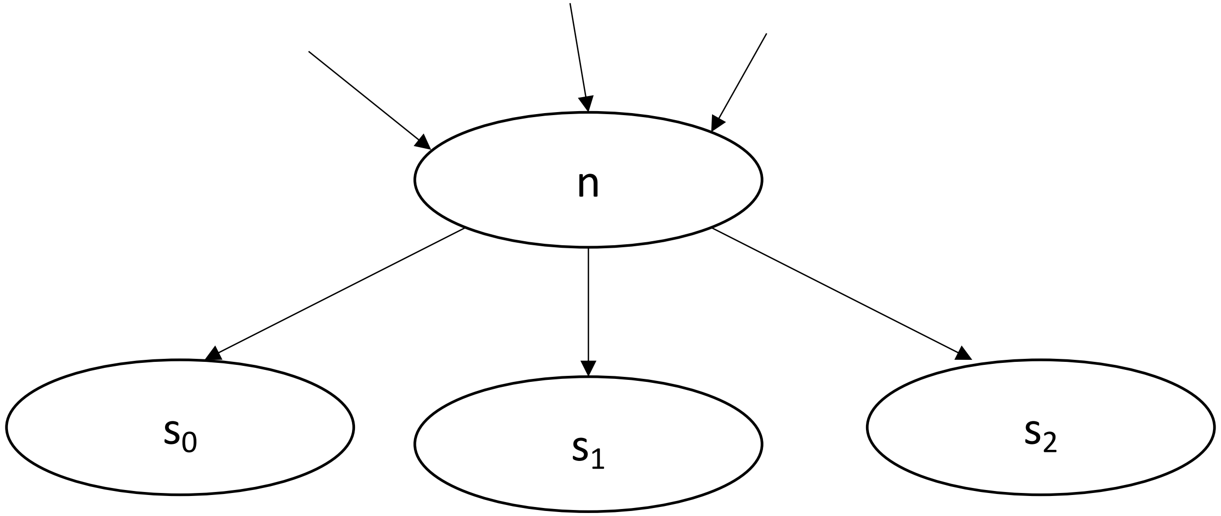
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



Backwards flow analysis because values flow from successors

Live variable analysis in the CFG:

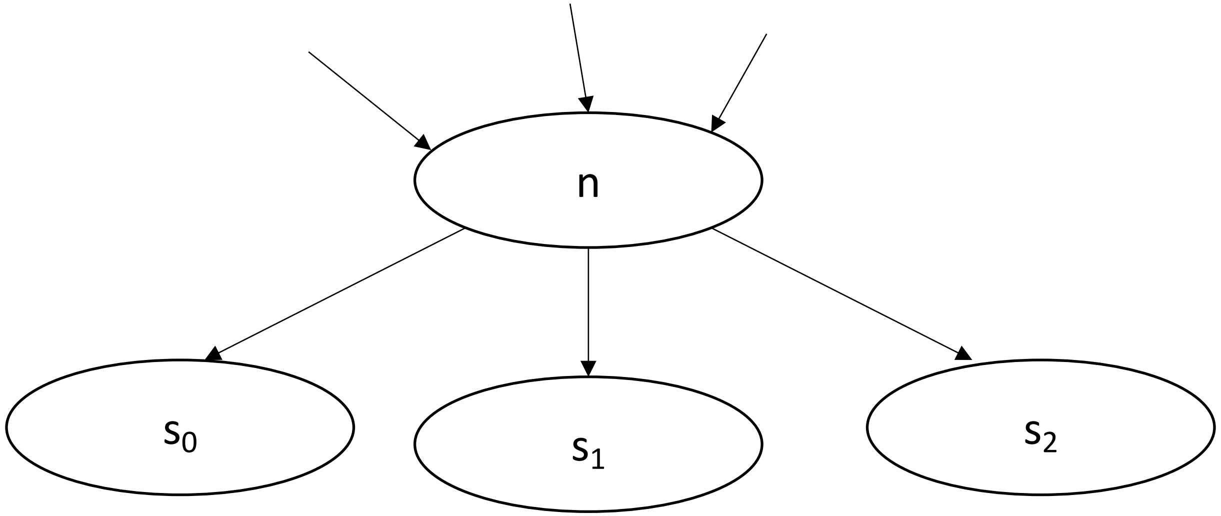
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

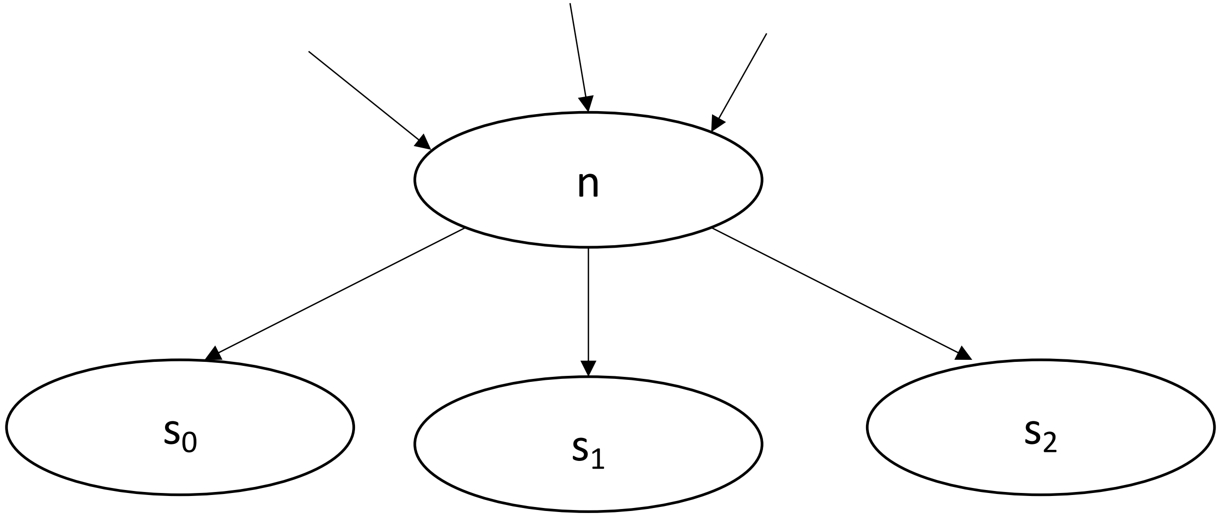
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



variables that are not overwritten in s

Live variable analysis in the CFG:

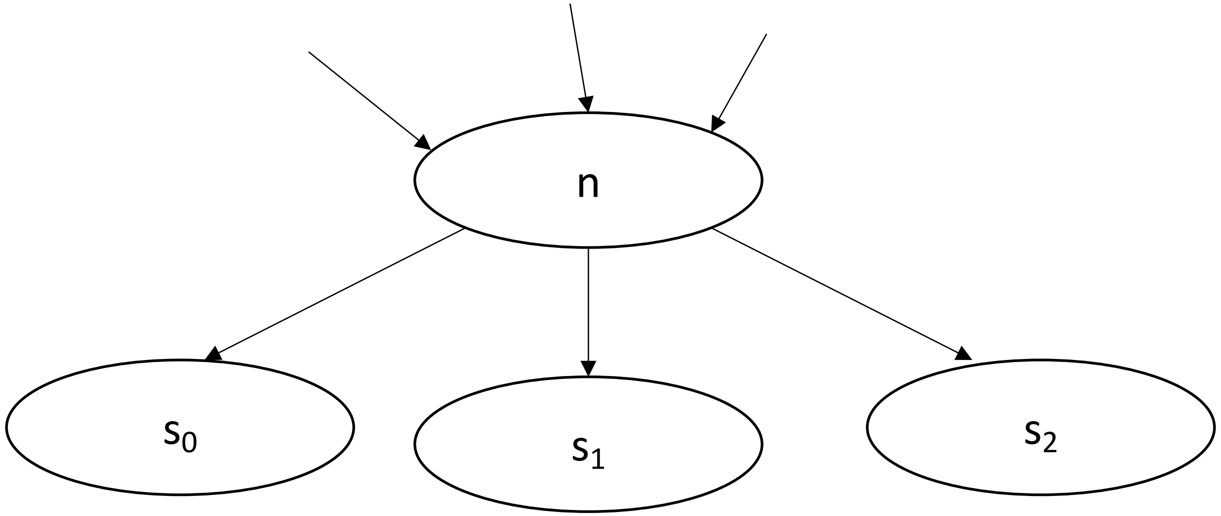
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$



variables that are live
at the end of s

Live variable analysis in the CFG:

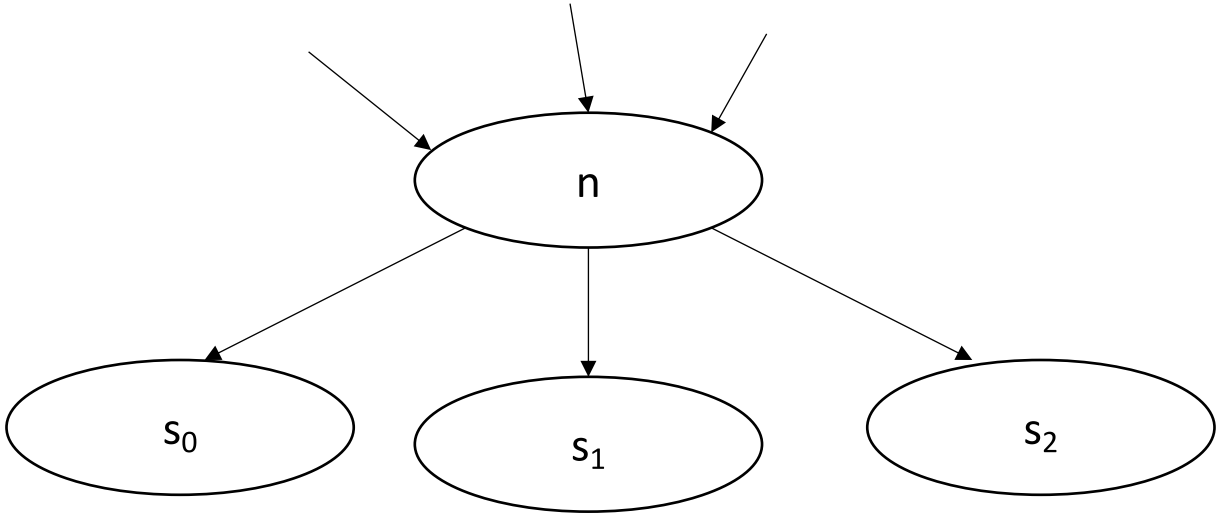
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$



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

Live variable analysis in the CFG:

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



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

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

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

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*

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

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

See you virtually on Friday

- We will discuss other flow algorithms

- Start talking about SSA construction

- Remember: no class on Wednesday! Get started on HW2!