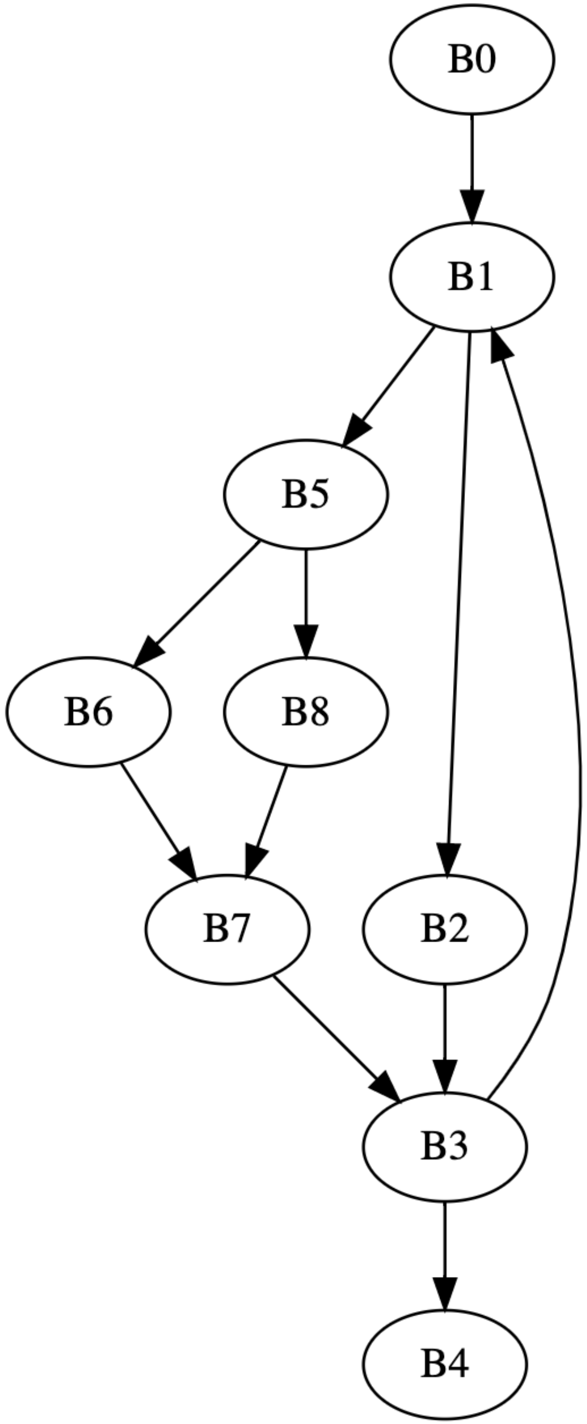
CSE211: Compiler Design

Oct. 18, 2022

- **Topic:** global optimizations
- **Questions**: how can we reason about arbitrary CFGs?



Announcements

- Office hours Today:
  - 3:30 - 5:30 PM
  - Moved to remote
- Homework 1:
  - Due today (at 11:59 pm)
  - Likely won’t be available for help after office hours
- Homework 2:
  - I will try to release it today
    - Possible for delays
  - I will send out the pair programming sign up sheet
  - Try to find a partner by the end of the week so that you can start!

Announcements

- Thursday will be asynchronous
    - Plan on asynchronous
    - Unless I’m sick, then we’ll do remote, like today. I’ll let you know ASAP
    - Thanks for being patient with the uncertainty!

Announcements

- Mark your attendance for today after you watch the recording (or if you are attending live)
    - Please try to keep on top of this.
    - We have put attendance in up until now. Let us know within 1 week if there are any issues.

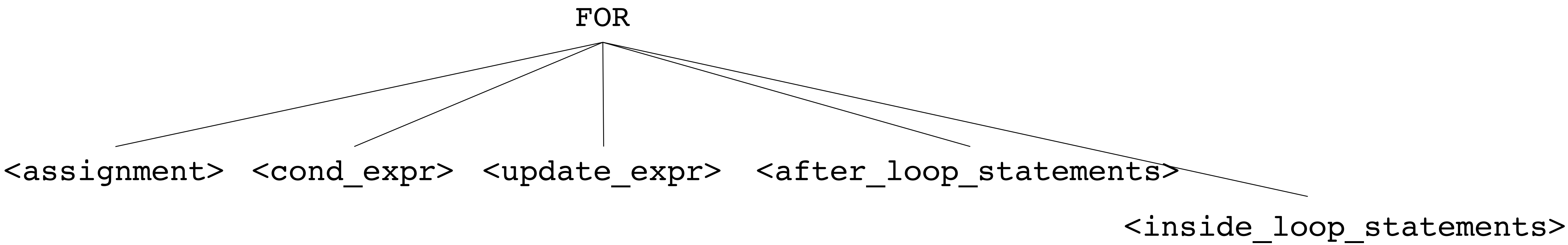
- Only mark Oct. 20 attendance after you watch the lectures.

Guest lecture confirmed

- Felix Klock
  - Principle Engineer at AWS
  - Big contributor to Rust
  - wants to tell us about work on incremental compilation

- Nov. 29 (Felix will be remote, but we will stream his lecture in the classroom)

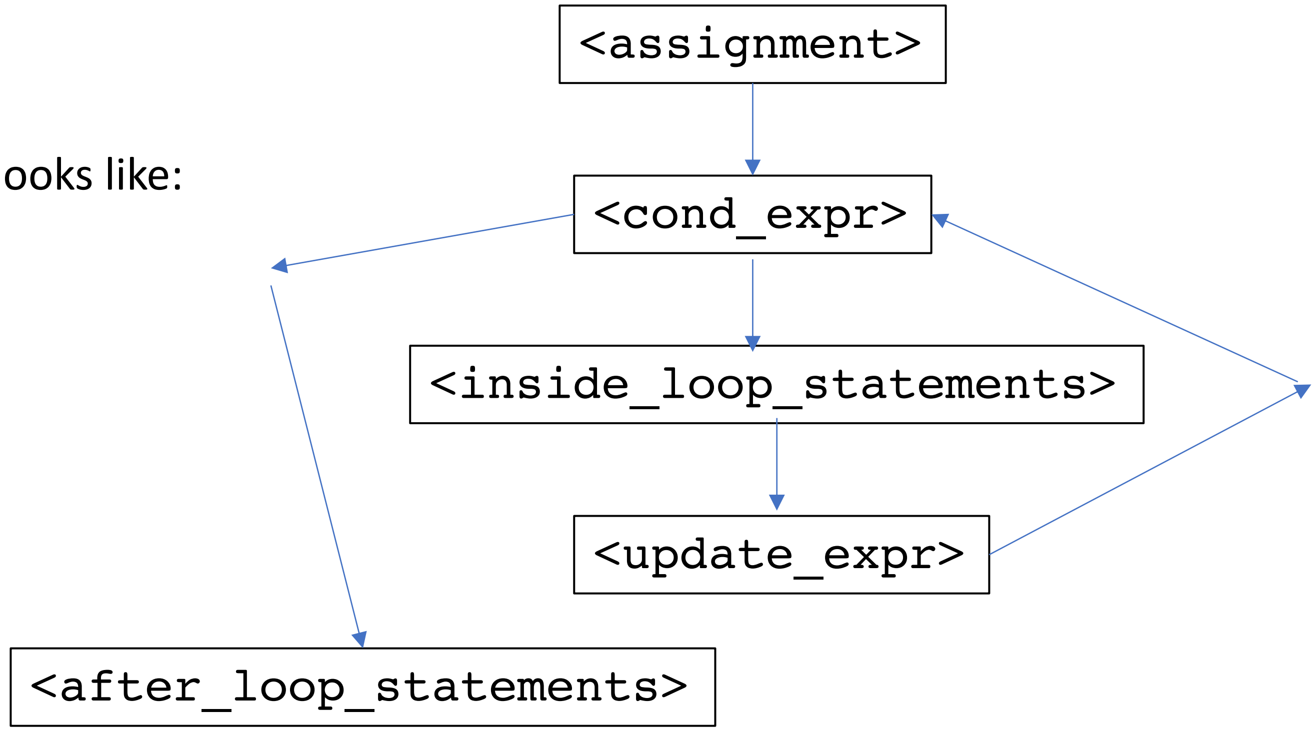
Review regional optimizations



FOR

<assignment> <cond_expr> <update_expr> <after_loop_statements> <inside_loop_statements>

If all of these are basic blocks then the CFG looks like:



<assignment>

<cond_expr>

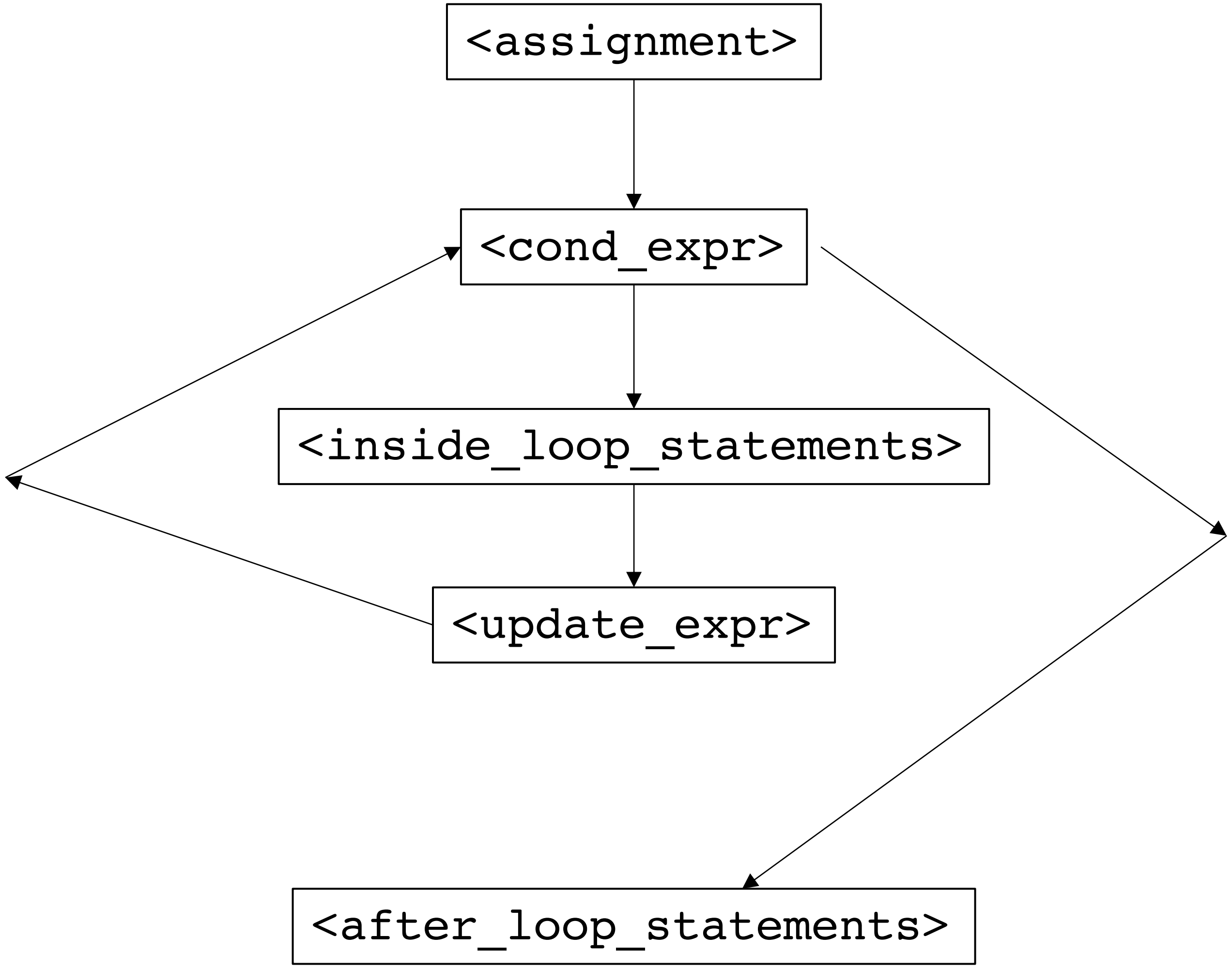
<inside_loop_statements>

<update_expr>

<after_loop_statements>

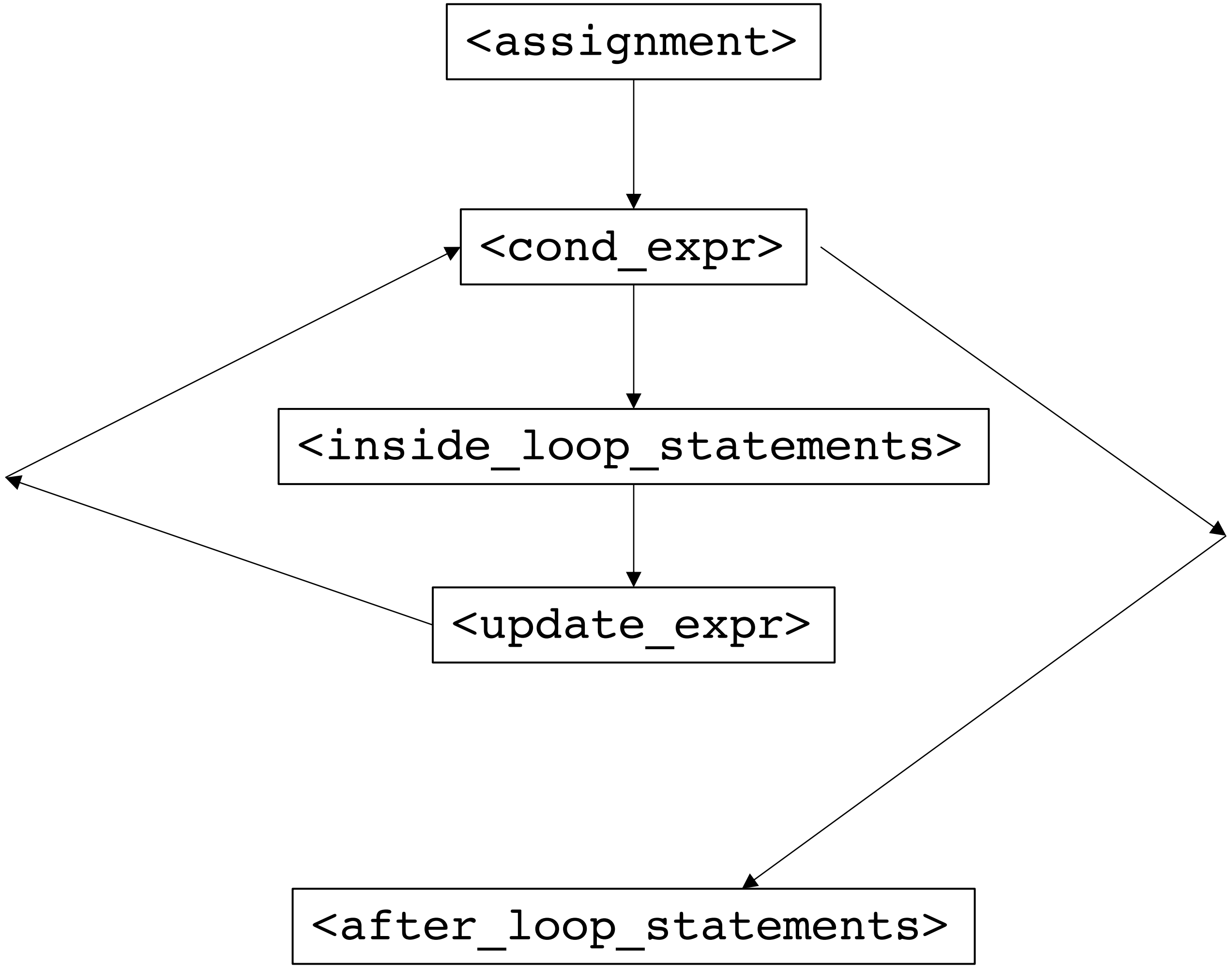
Loop unrolling:

What could change this CFG?

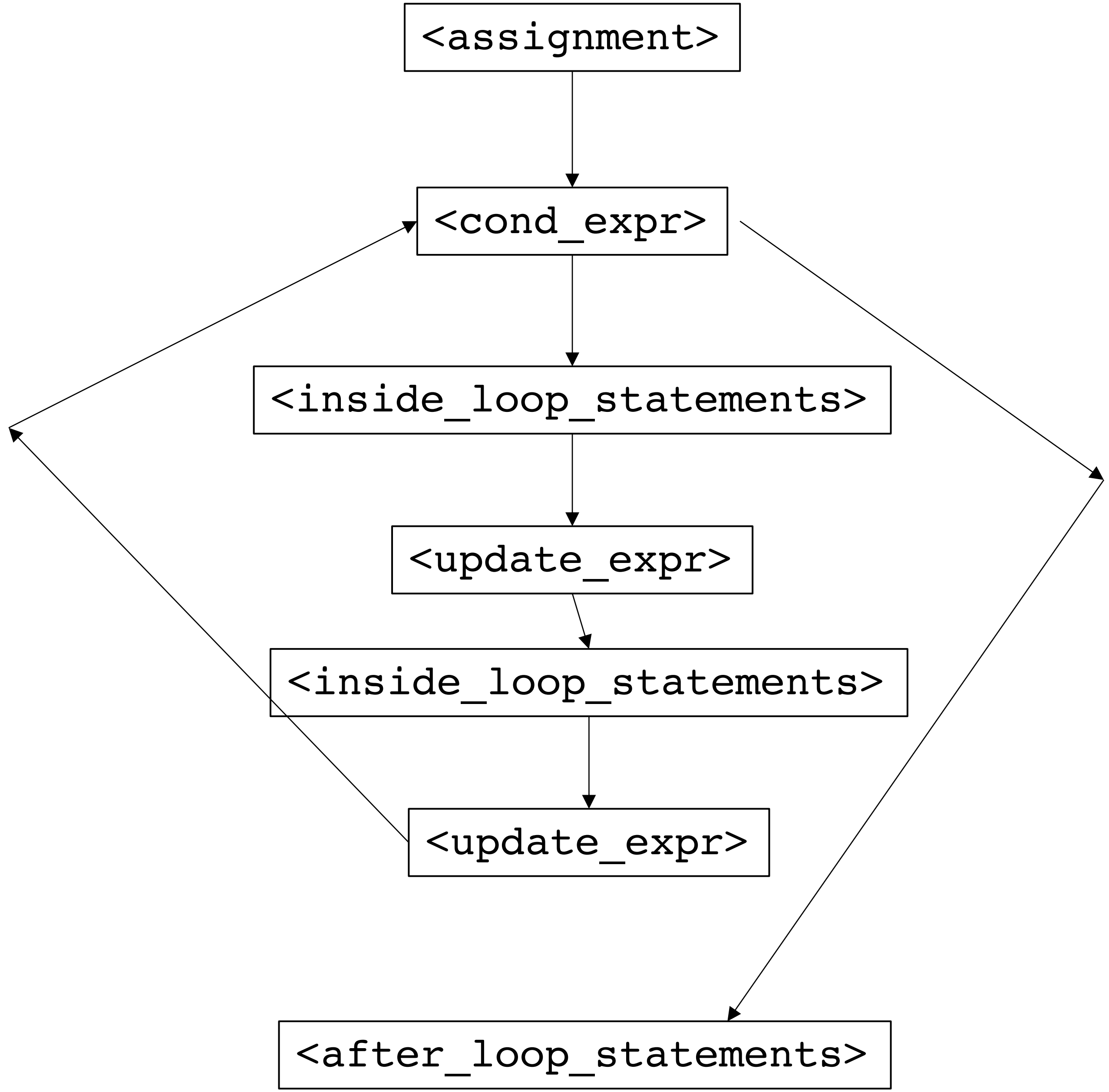


Loop unrolling:

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



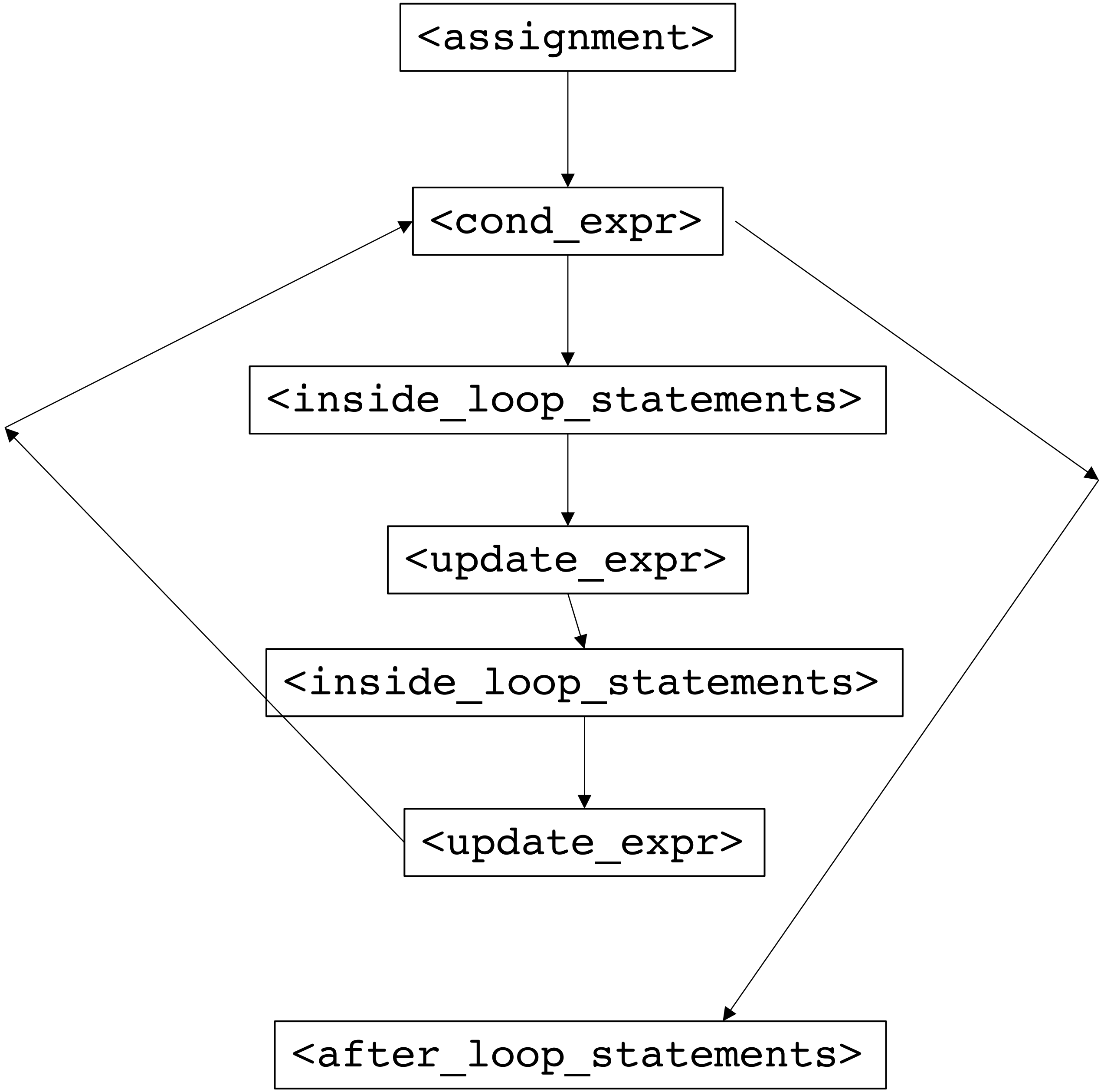
Loop unrolling:


<assignment>
<cond_expr>
<inside_loop_statements>
<update_expr>
<inside_loop_statements>
<update_expr>
<after_loop_statements>

Loop unrolling:

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

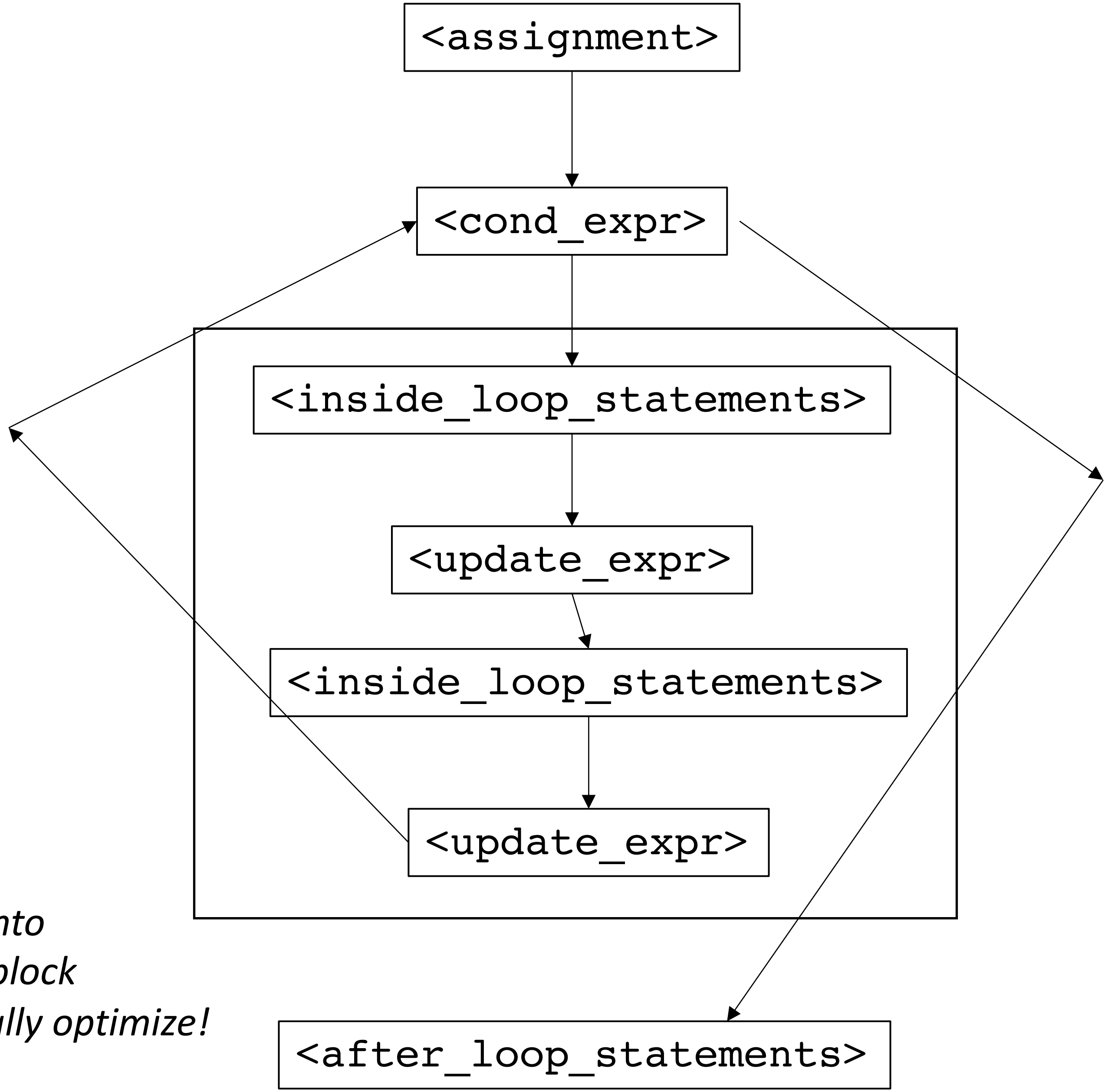
What have we saved here?



Loop unrolling:

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

What have we saved here?



`<assignment>`

`<cond_expr>`

`<inside_loop_statements>`

`<update_expr>`

`<inside_loop_statements>`

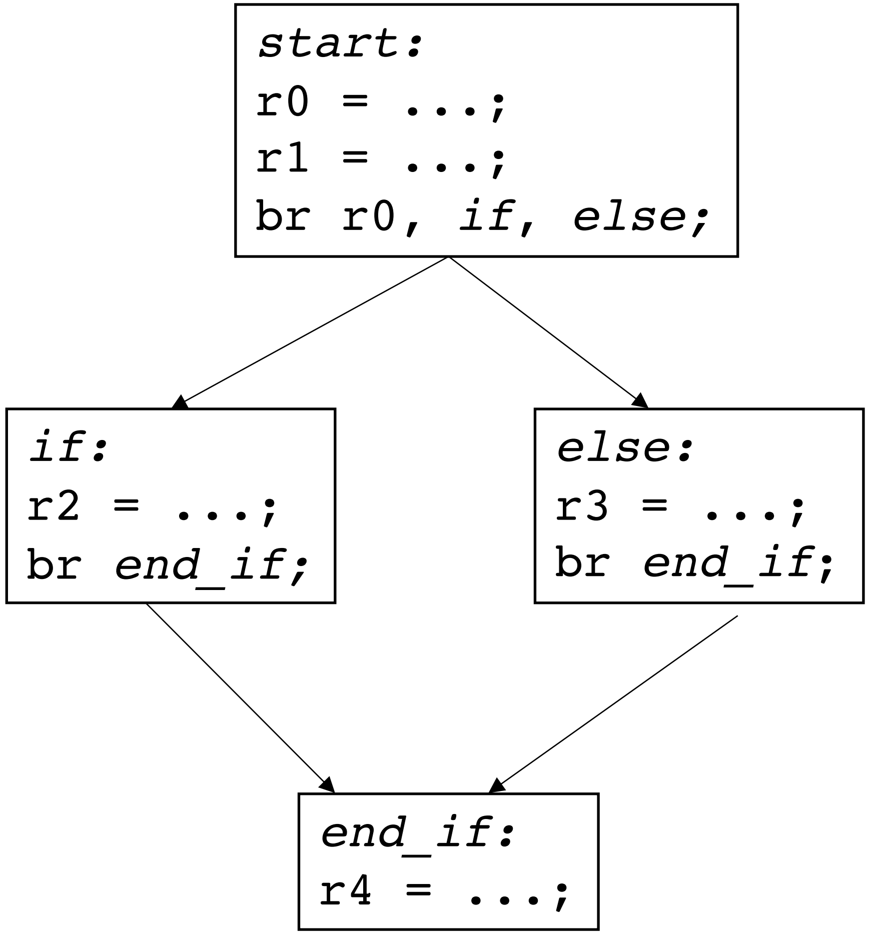
`<update_expr>`

`<after_loop_statements>`

merge into
1 basic block
and locally optimize!

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 = ...;
```

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
```

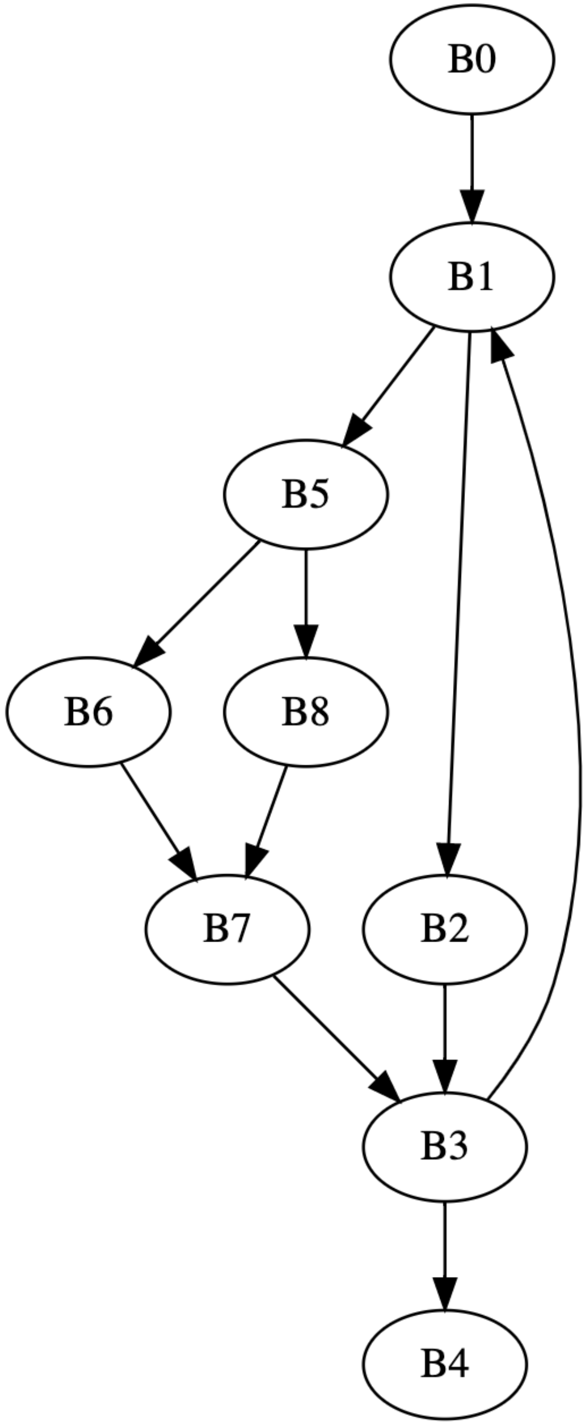
```
else:
r3 = ...;
br end_if;
```

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

CSE211: Compiler Design

Oct. 18, 2022

- **Topic:** global optimizations
- **Questions**: how can we reason about arbitrary CFGs?

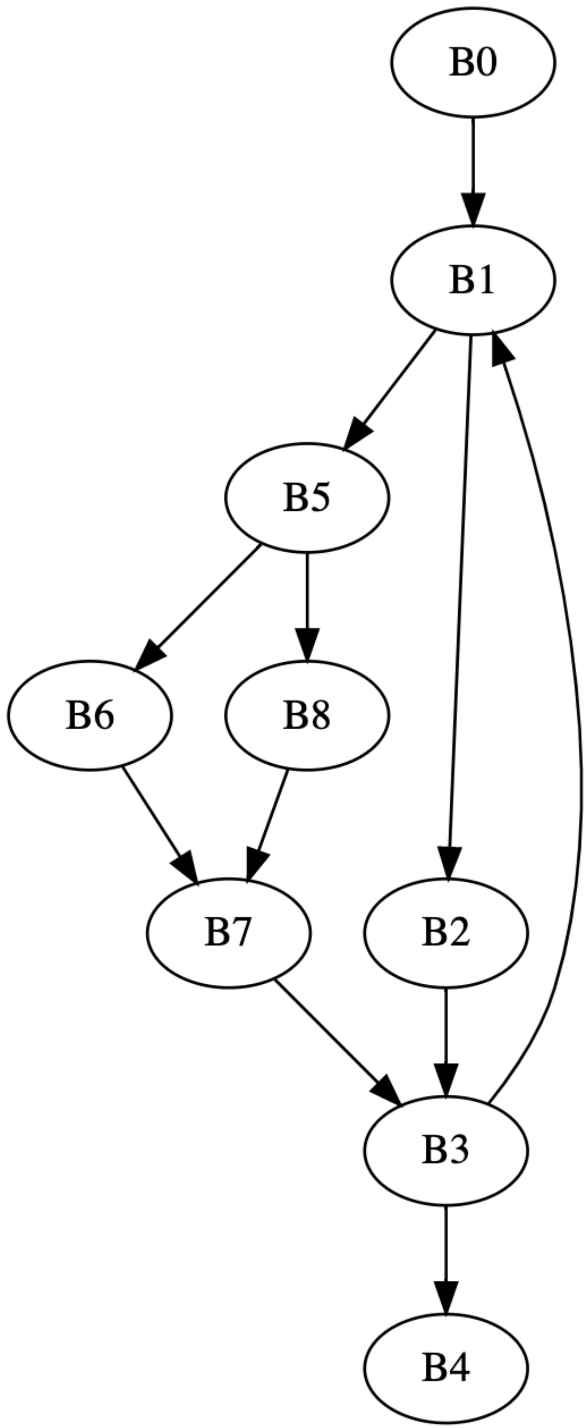


CSE211: Compiler Design

Oct. 18, 2022

- **Topic:** global optimizations
- **Questions**: how can we reason about arbitrary CFGs?

Quick note: Global vs. Regional vs. Local - What do they all mean?



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

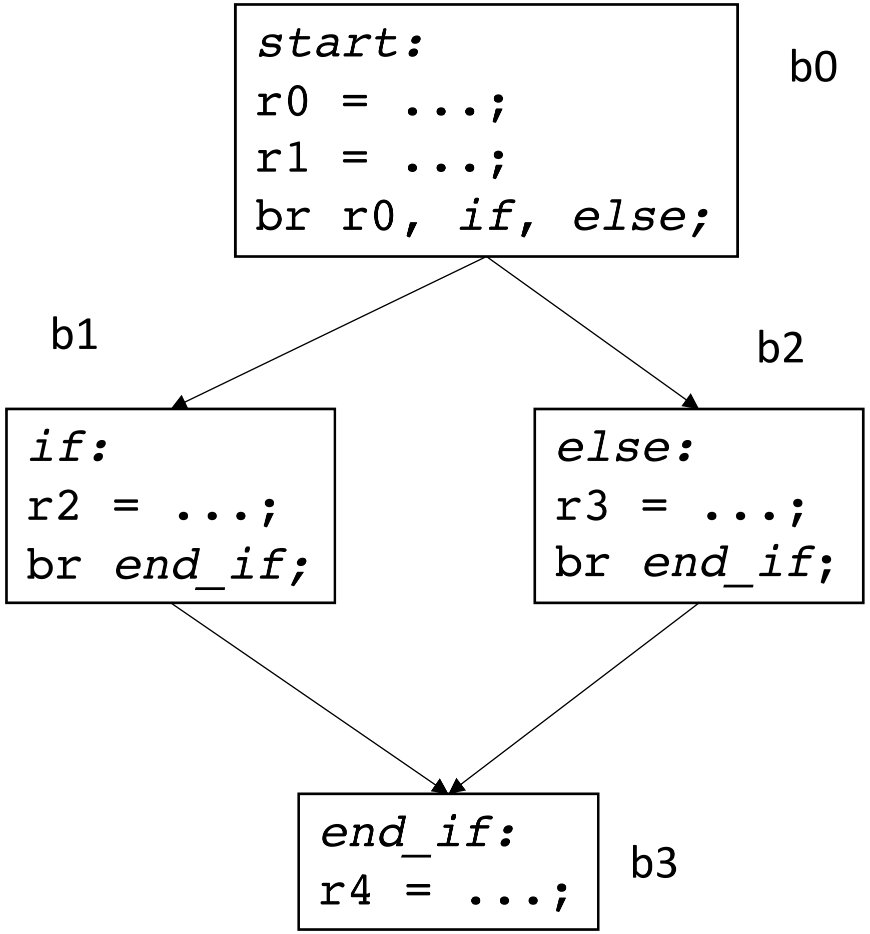
First concept:

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

Dominance

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

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



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

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

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

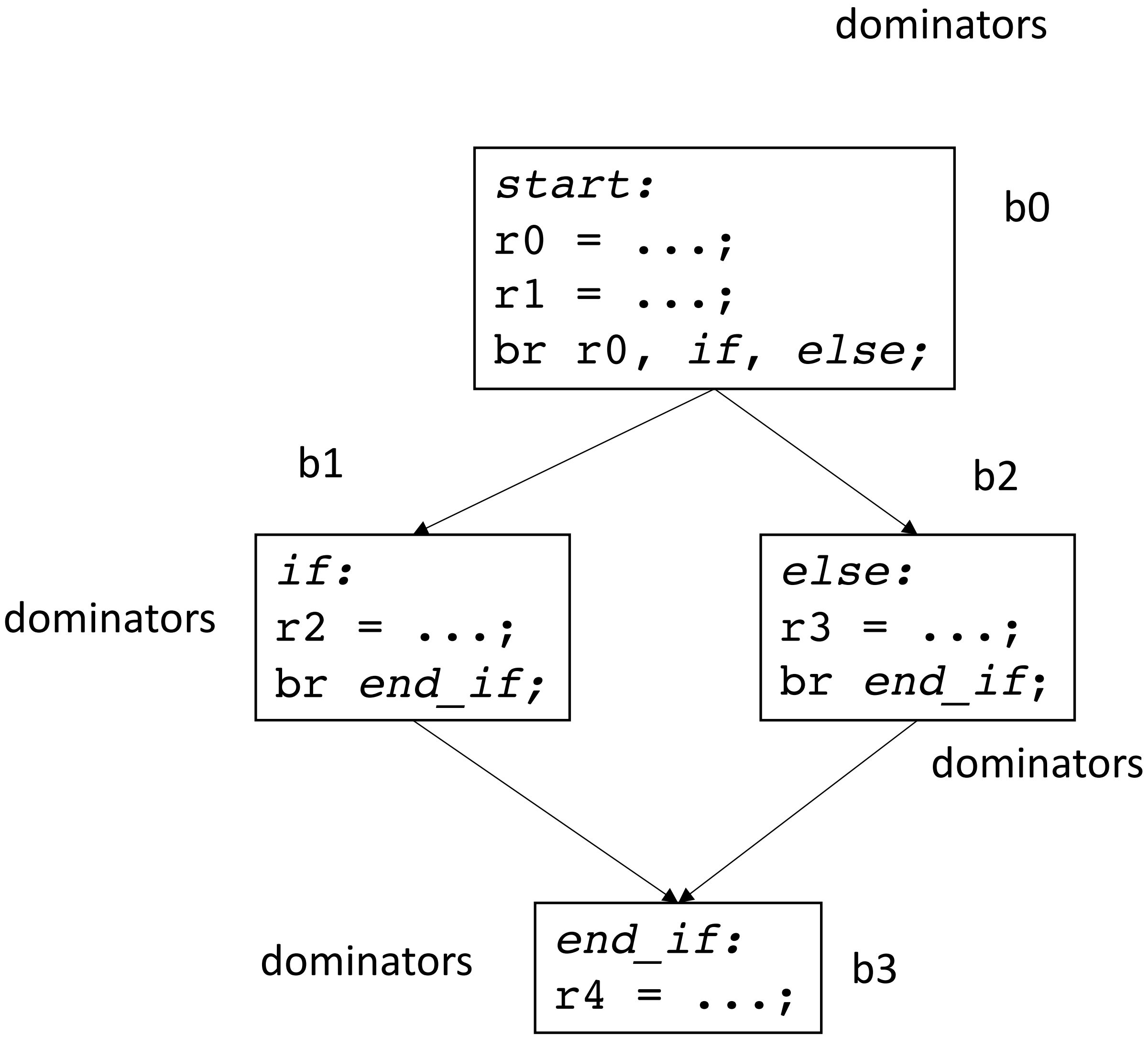
```
end_if:
r4 = ...;
```

Dominance

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

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

- Can we use this notion to extend local value numbering?



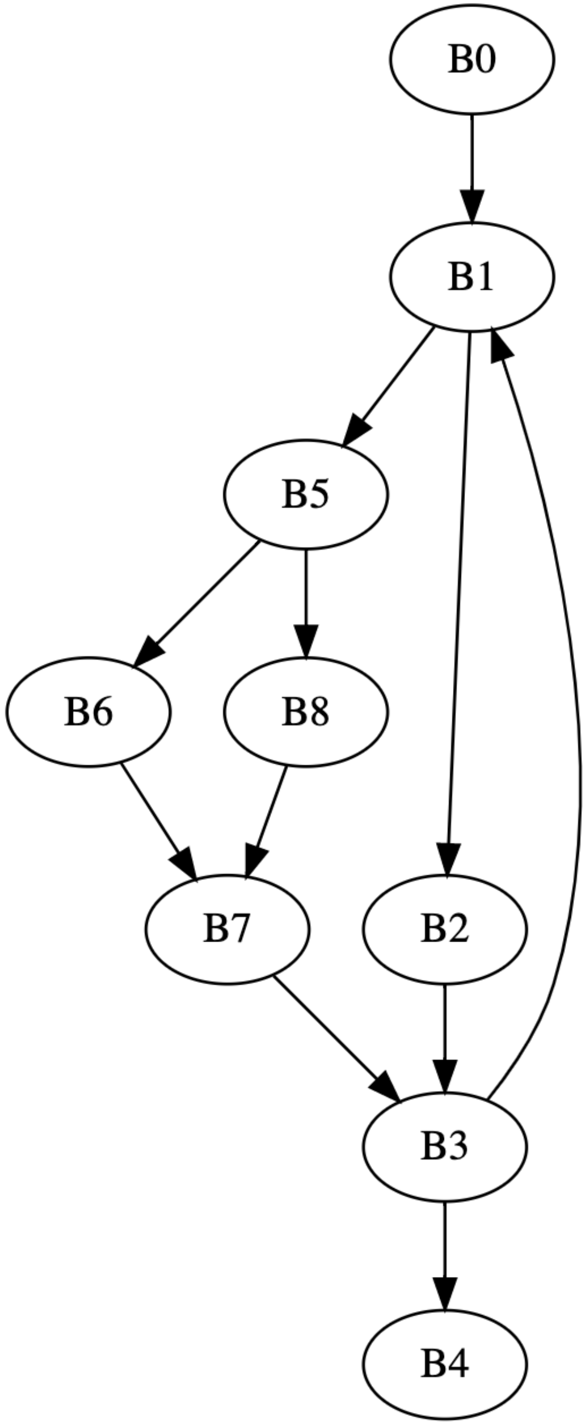
```
start:
r0 = ...;
r1 = ...;
br r0, if, else;
```

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

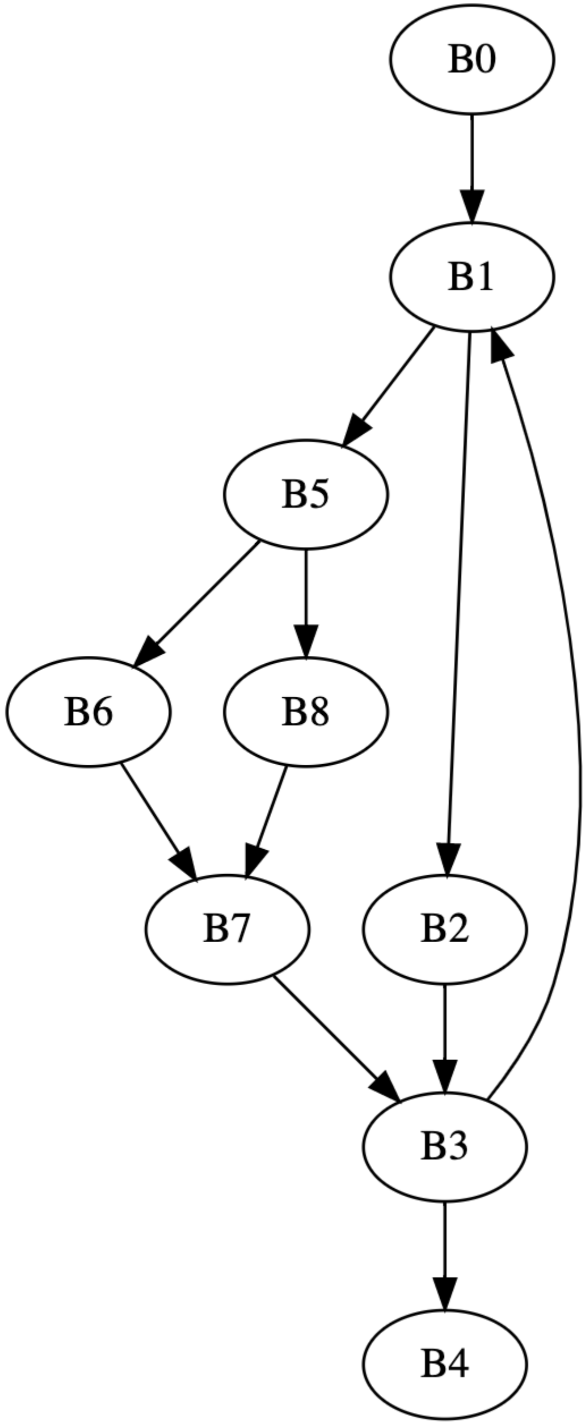
```
else:
r3 = ...;
br end_if;
```

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

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:

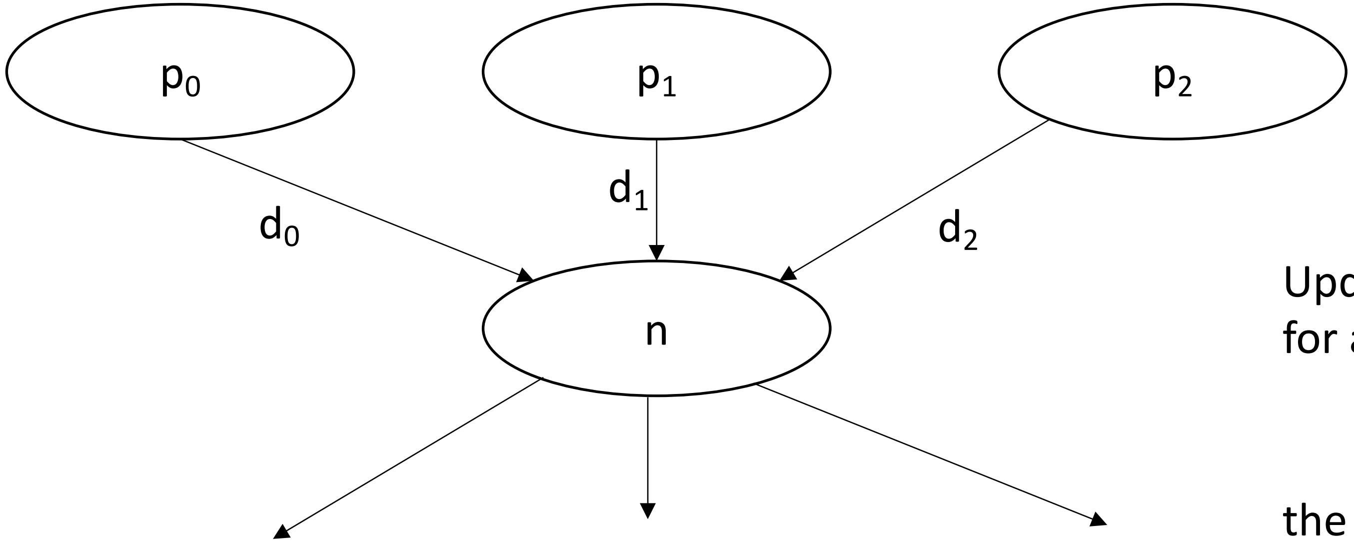
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in } preds(n)} Dom(m) \right)$$

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

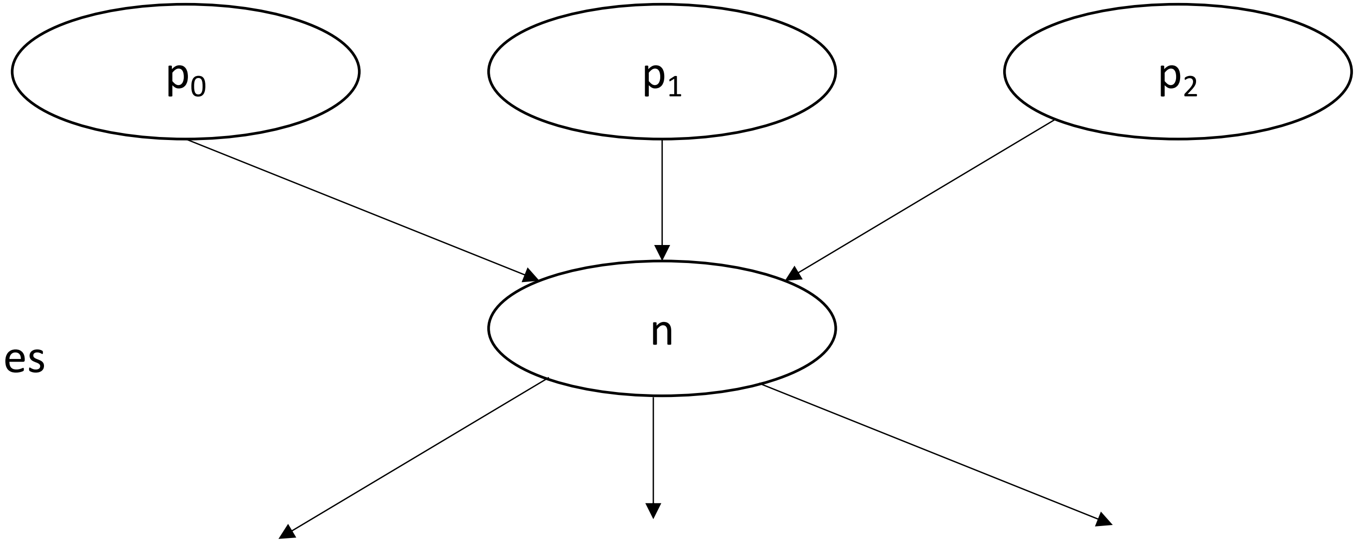


Update:
for all parents *p*: min(*p* + *d*)

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

Now lets think about dominance

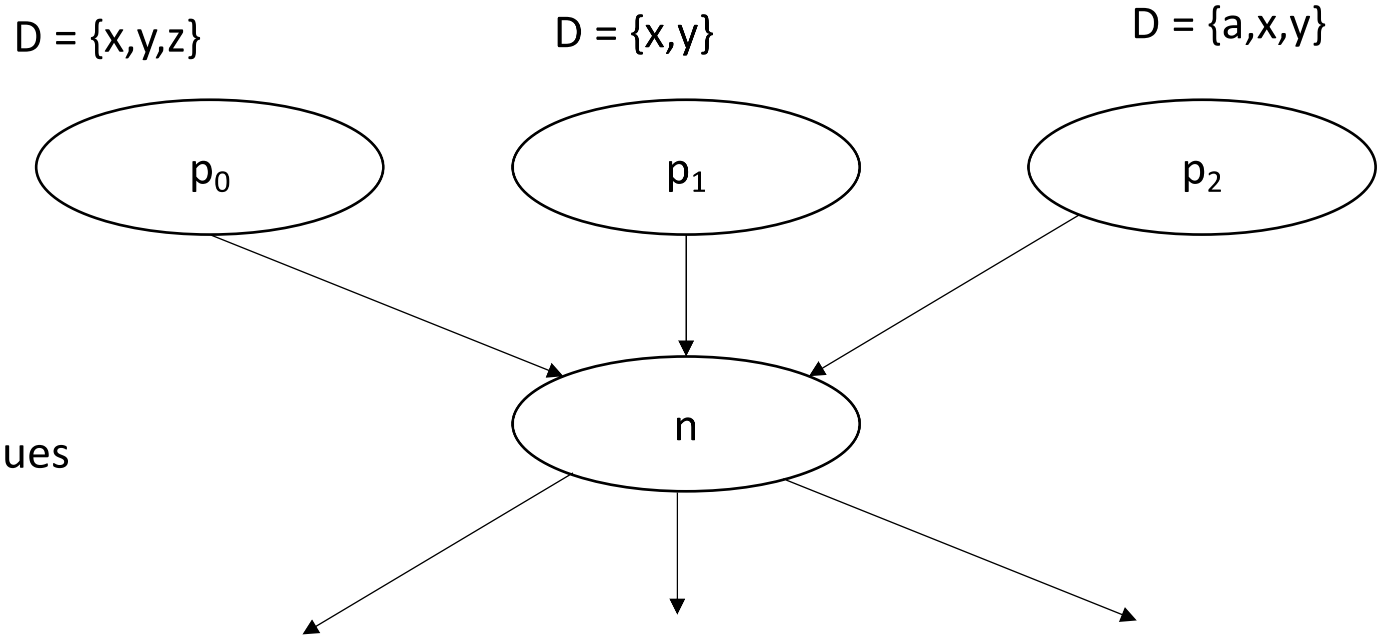
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



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



D = {x,y,z}

D = {x,y}

D = {a,x,y}

p_0

p_1

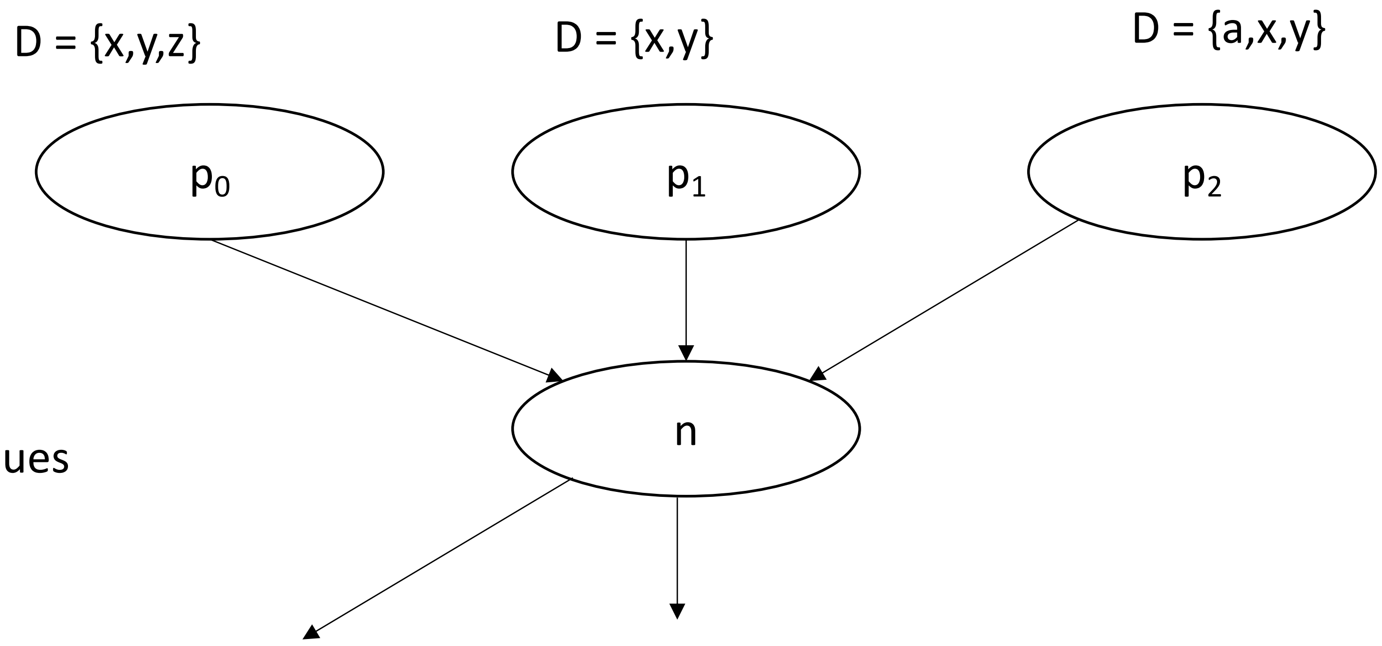
p_2

n

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

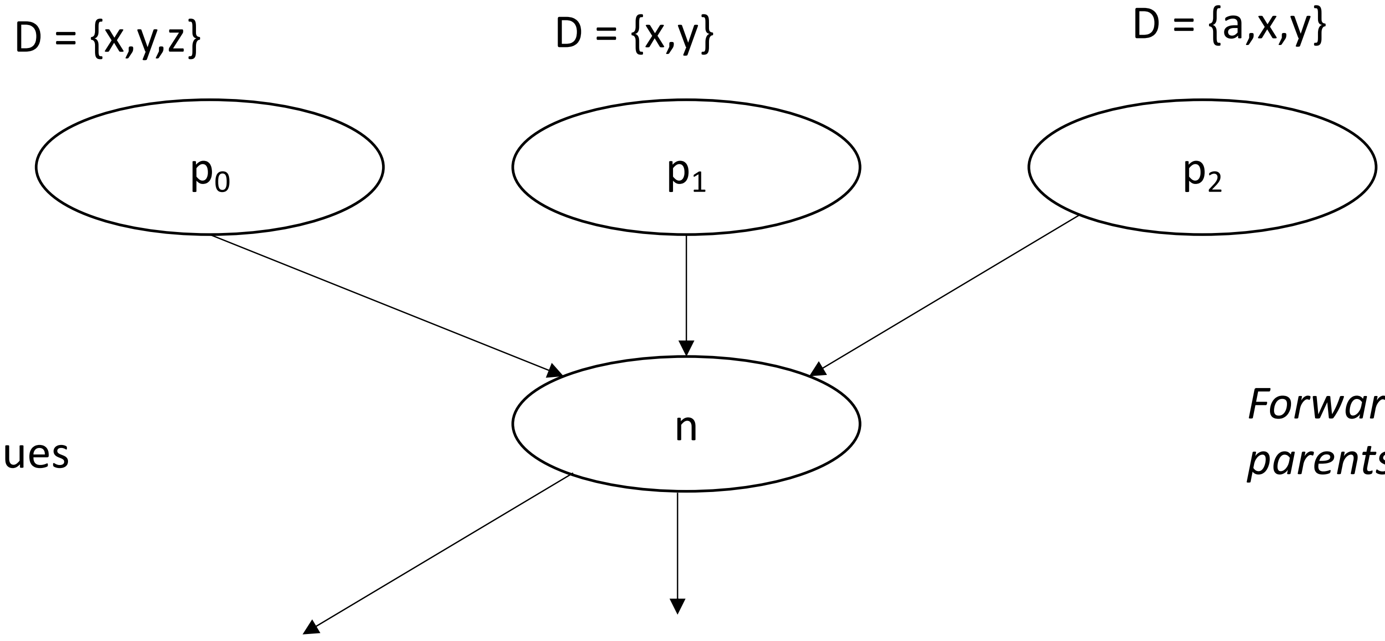


update:
intersection of parent values

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

Now lets think about dominance

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



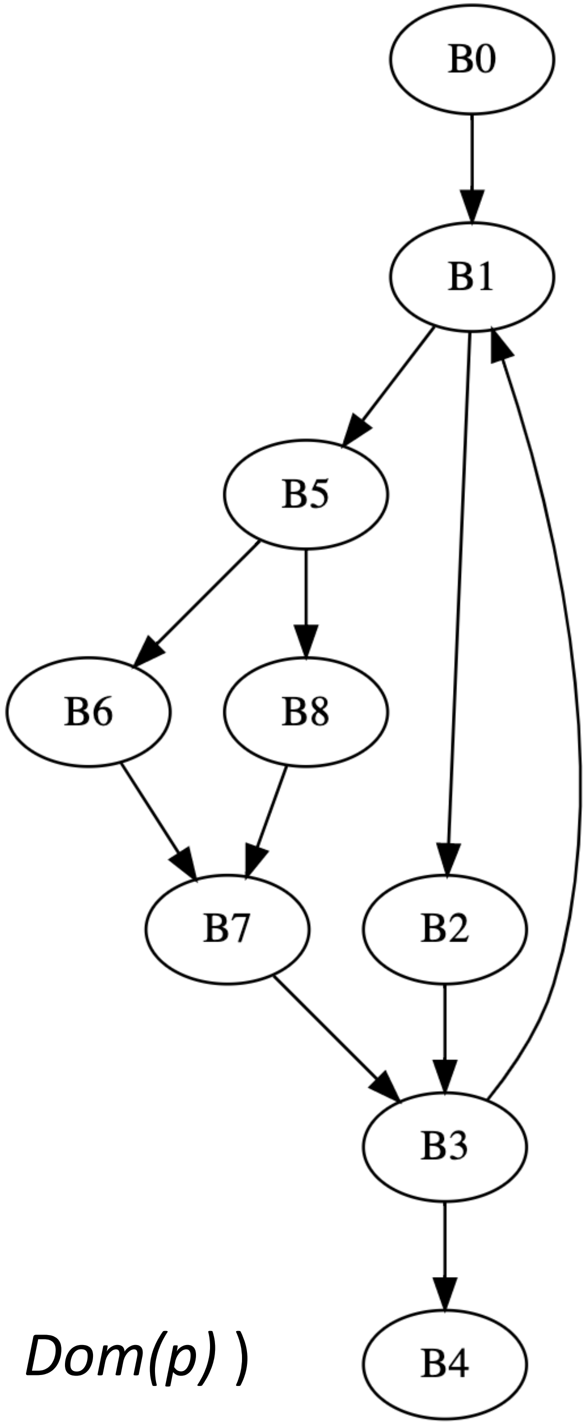
update:
intersection of parent values

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

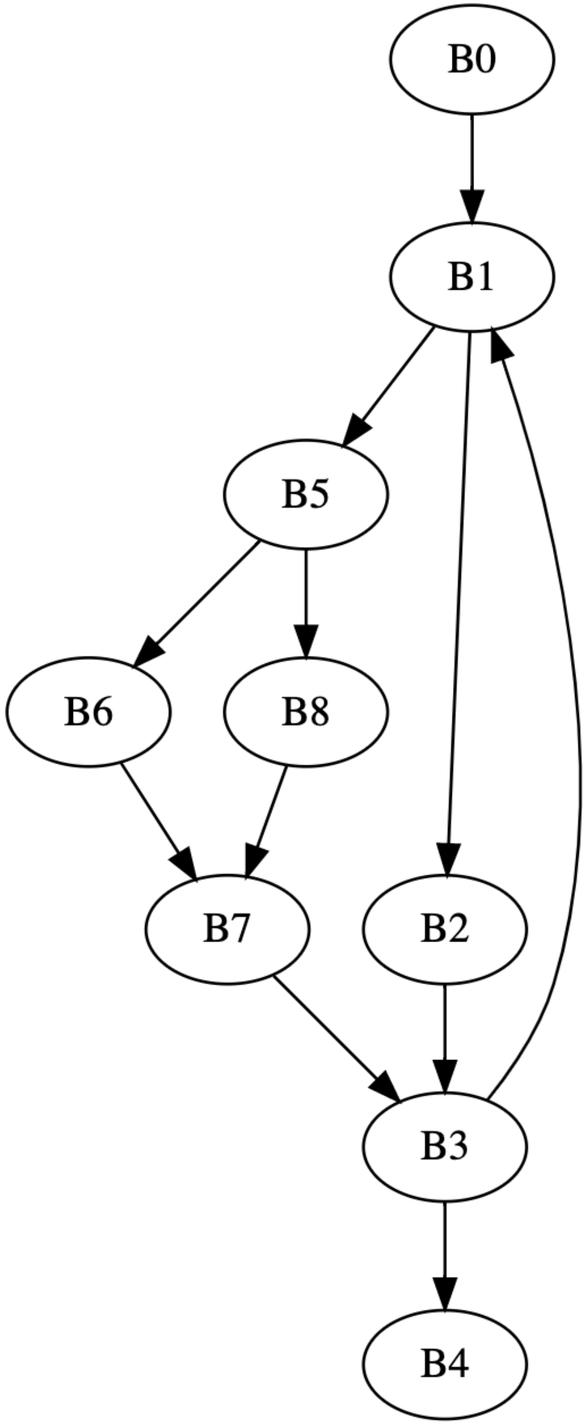
Node	Initial	Iteration 1
B0	B0	
B1	*N*	
B2	*N*	
B3	*N*	
B4	*N*	
B5	*N*	
B6	*N*	
B7	*N*	
B8	*N*	



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

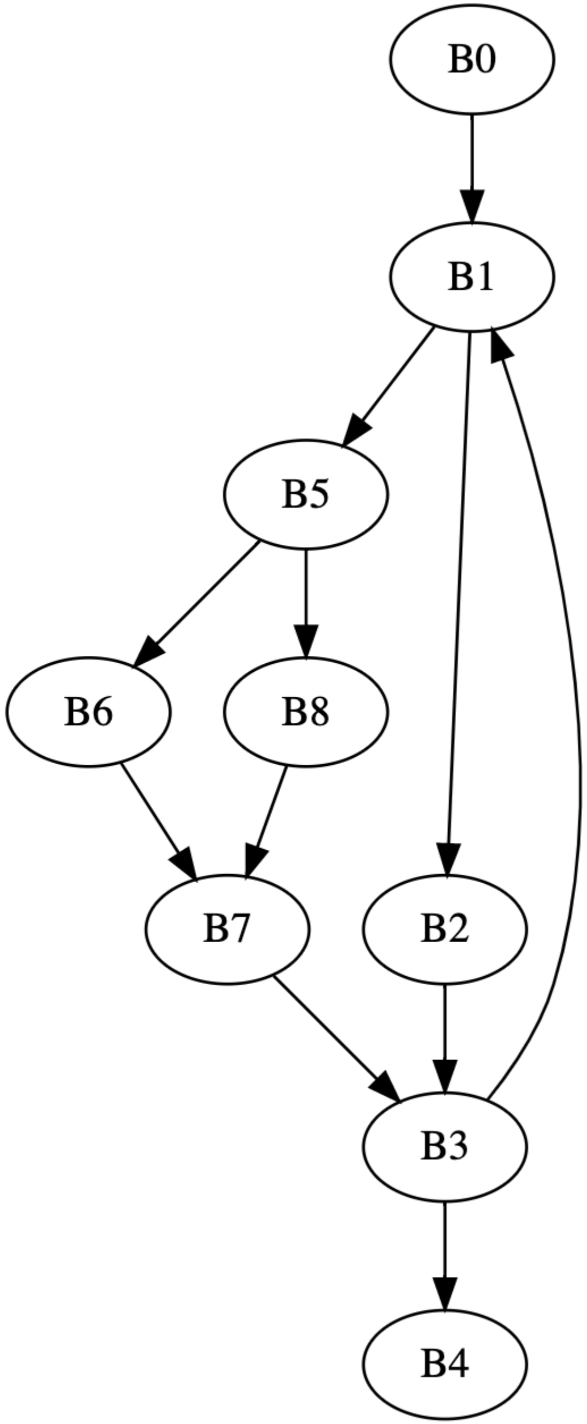
Lets try it

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		



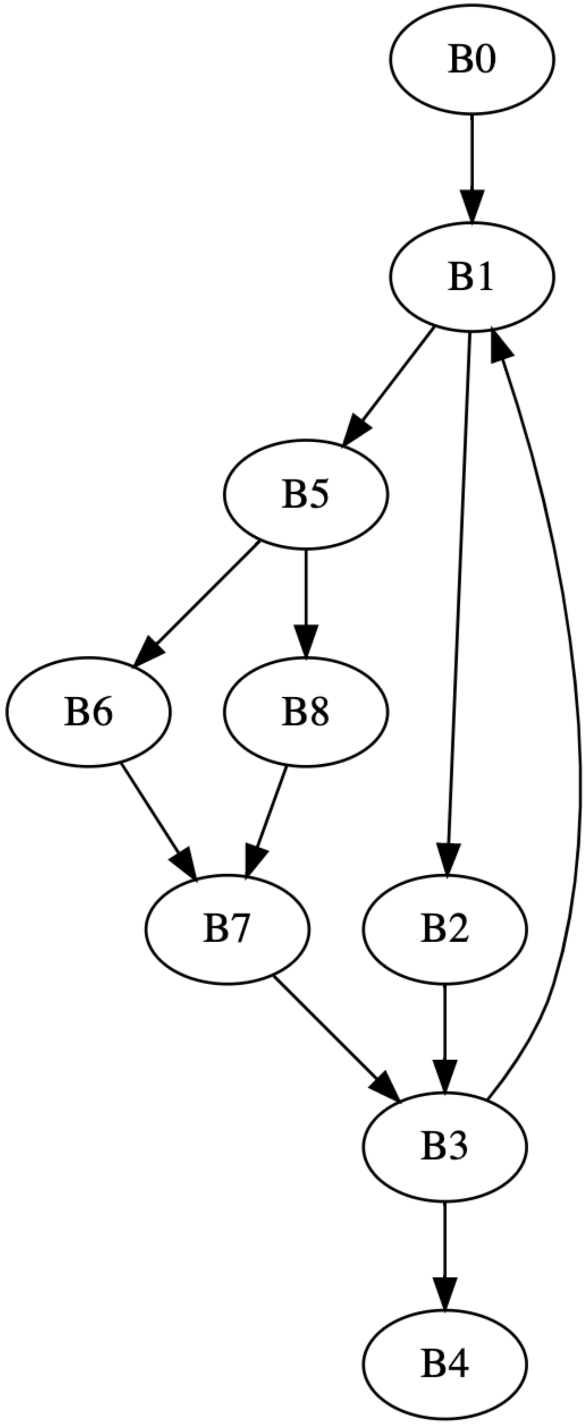
Lets try it

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	…	…

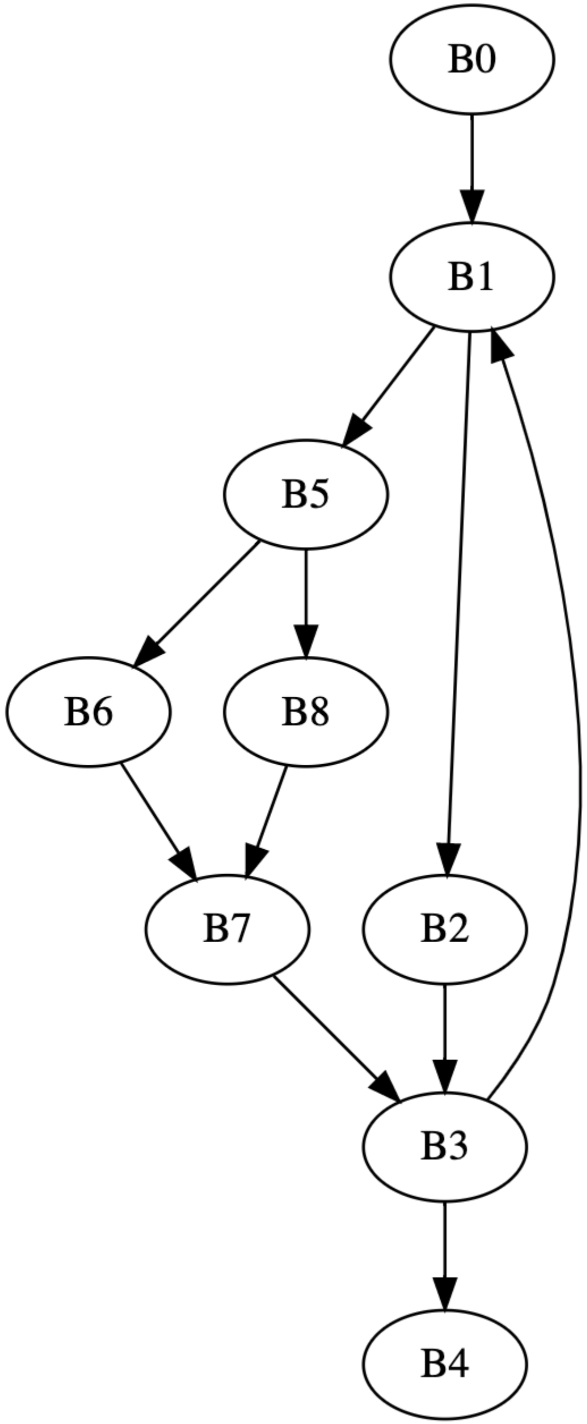


How can we optimize the algorithm?

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	…	…

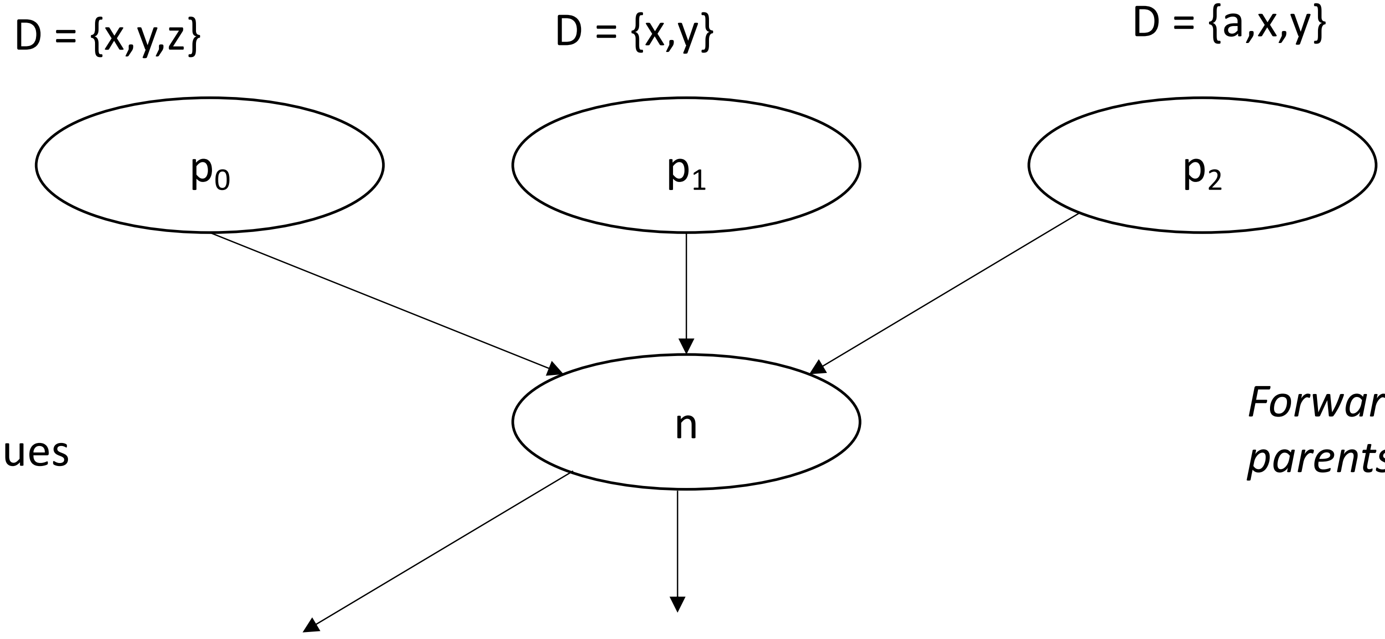
This can be any order...

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

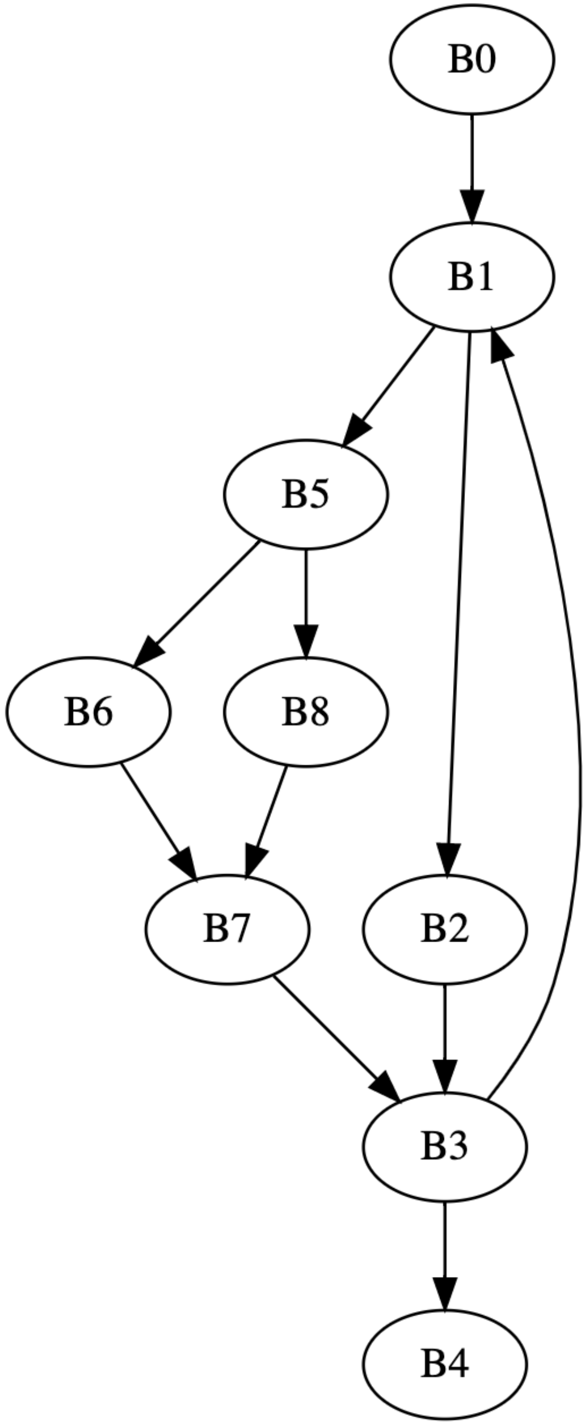
Forward flow, as updates flow from parents to children.

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

How can we optimize the algorithm?

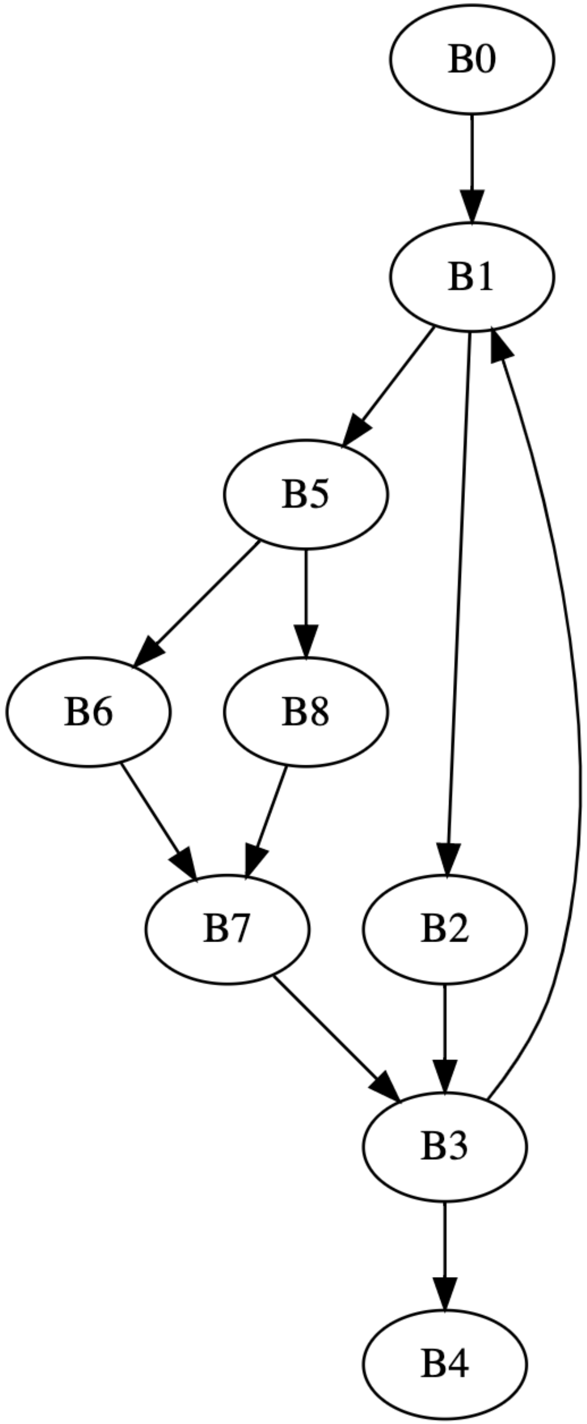
Node	New Order
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	

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



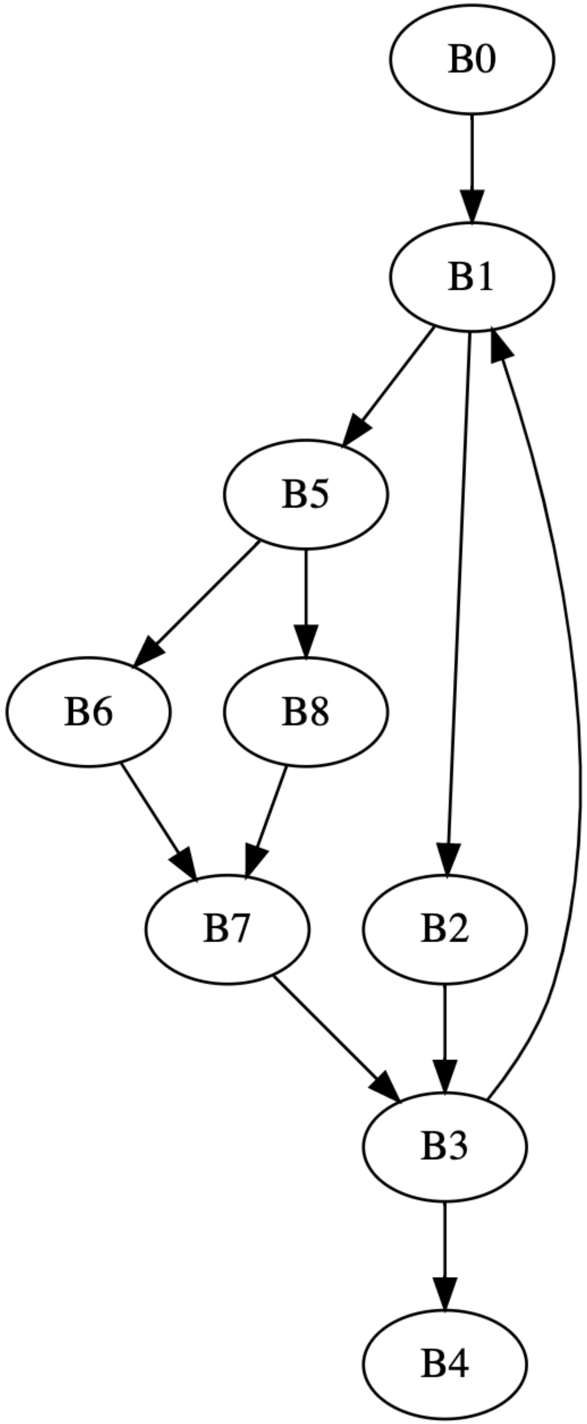
How can we optimize the algorithm?

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*			



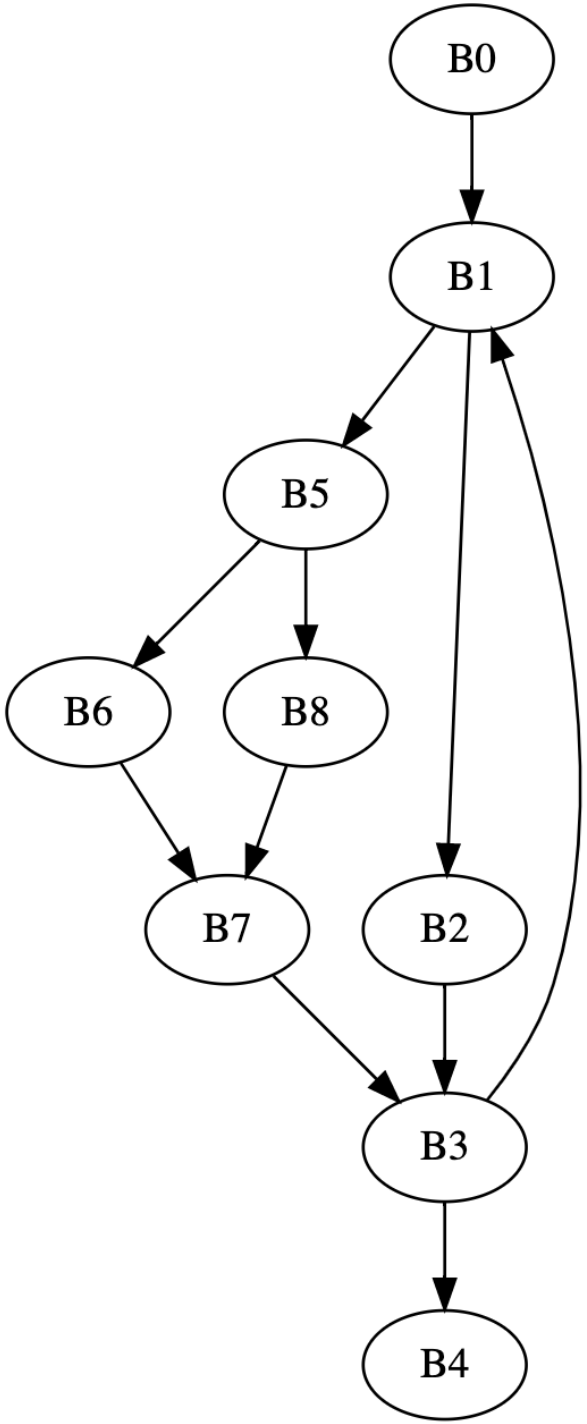
How can we optimize the algorithm?

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?

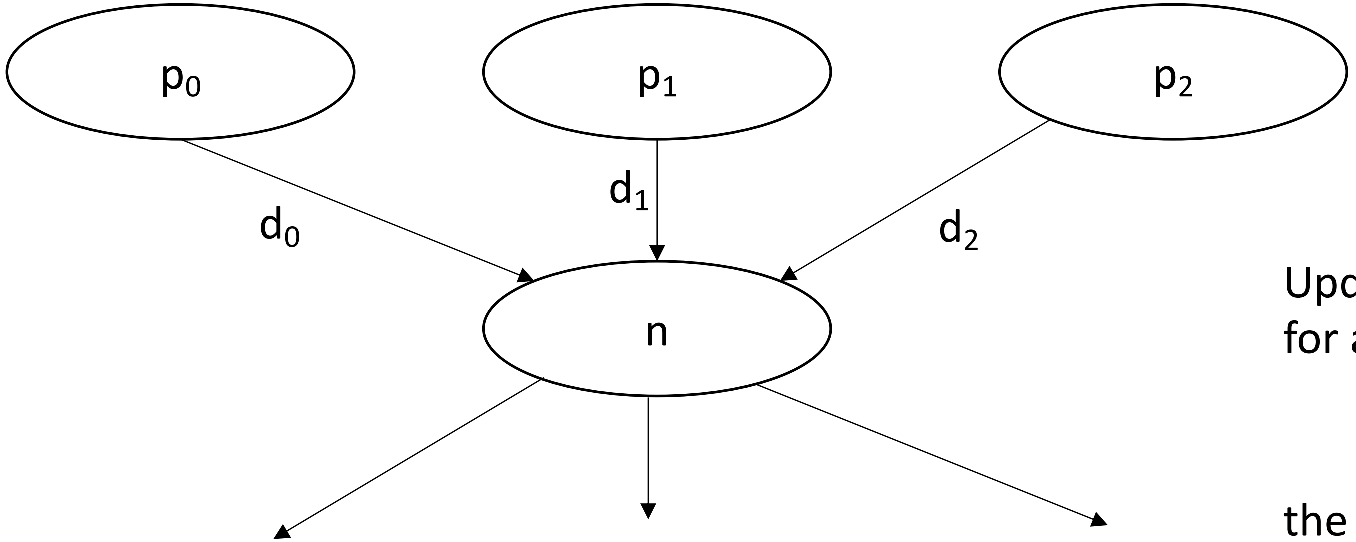
Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	...	
B1	*N*	B0,B1	...	
B2	*N*	B0,B1,B2	...	
B5	*N*	B0,B1,B5	...	
B6	*N*	B0,B1,B5,B6	...	
B8	*N*	B0,B1,B5,B8	...	
B7	*N*	B0,B1,B5,B7	...	
B3	*N*	B0,B1,B3	...	
B4	*N*	B0,B1,B4	...	



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: ?

```
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
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)
```

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:

```
//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:

```
//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

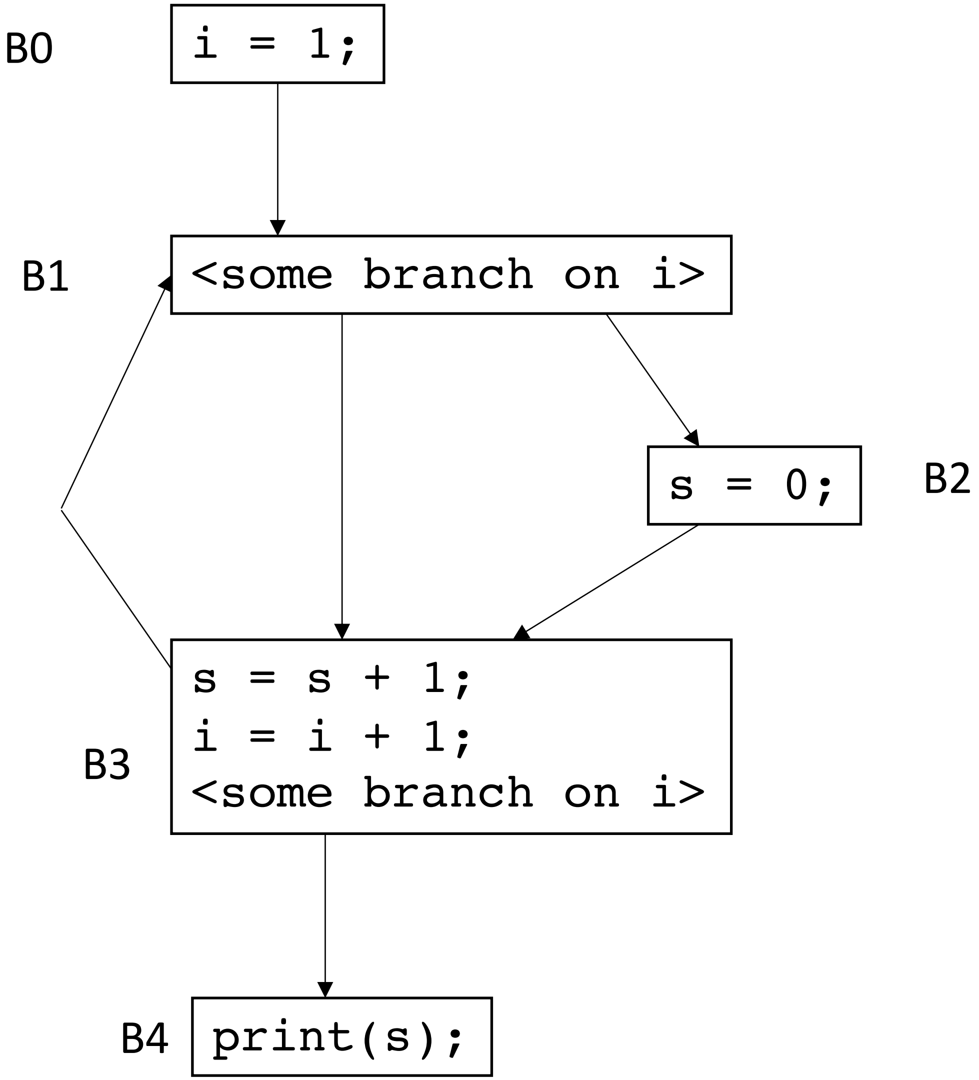
- examples:

Accessing an uninitialized variable!

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

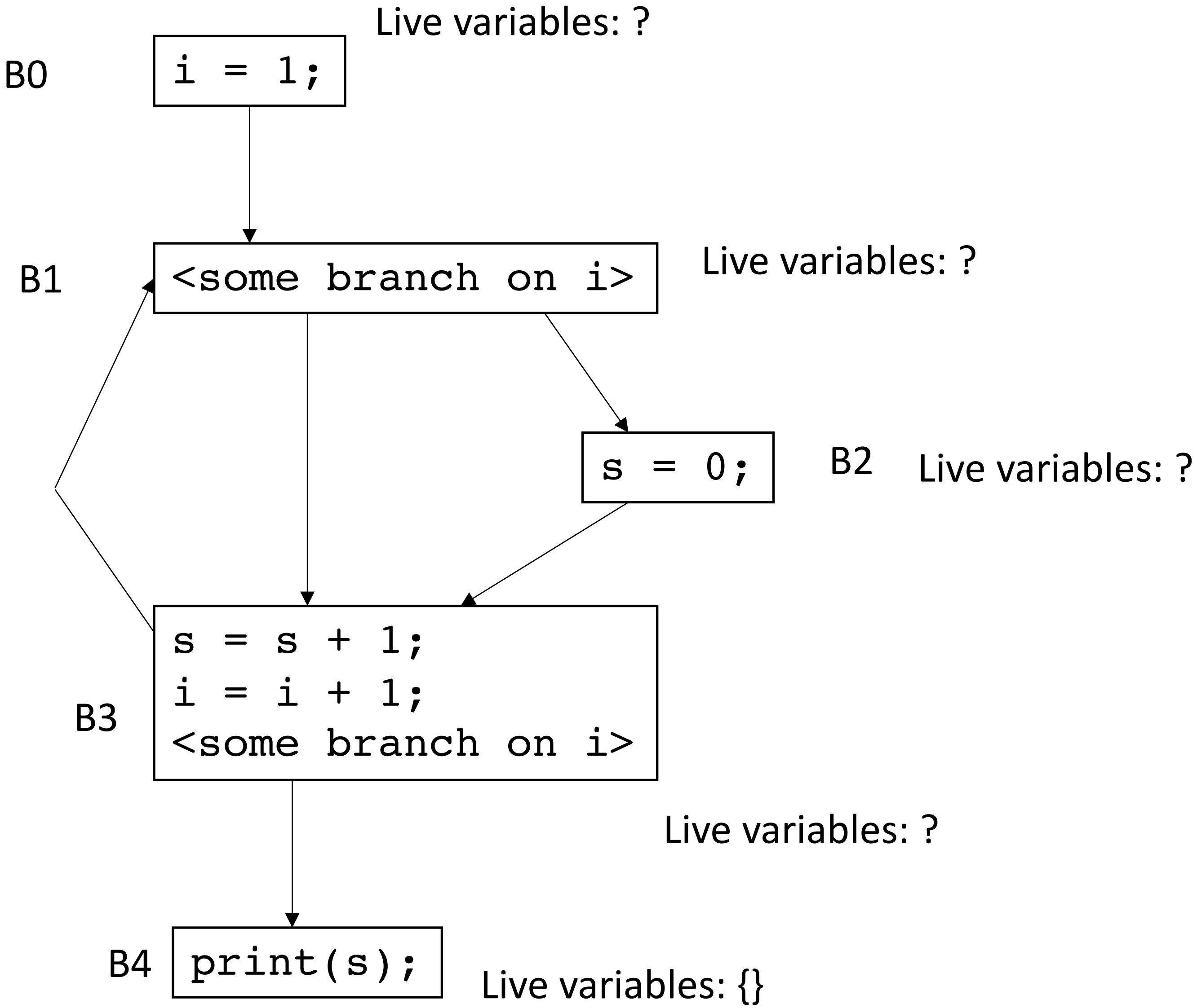
p ← Live variables: w

Live variable analysis in the CFG:

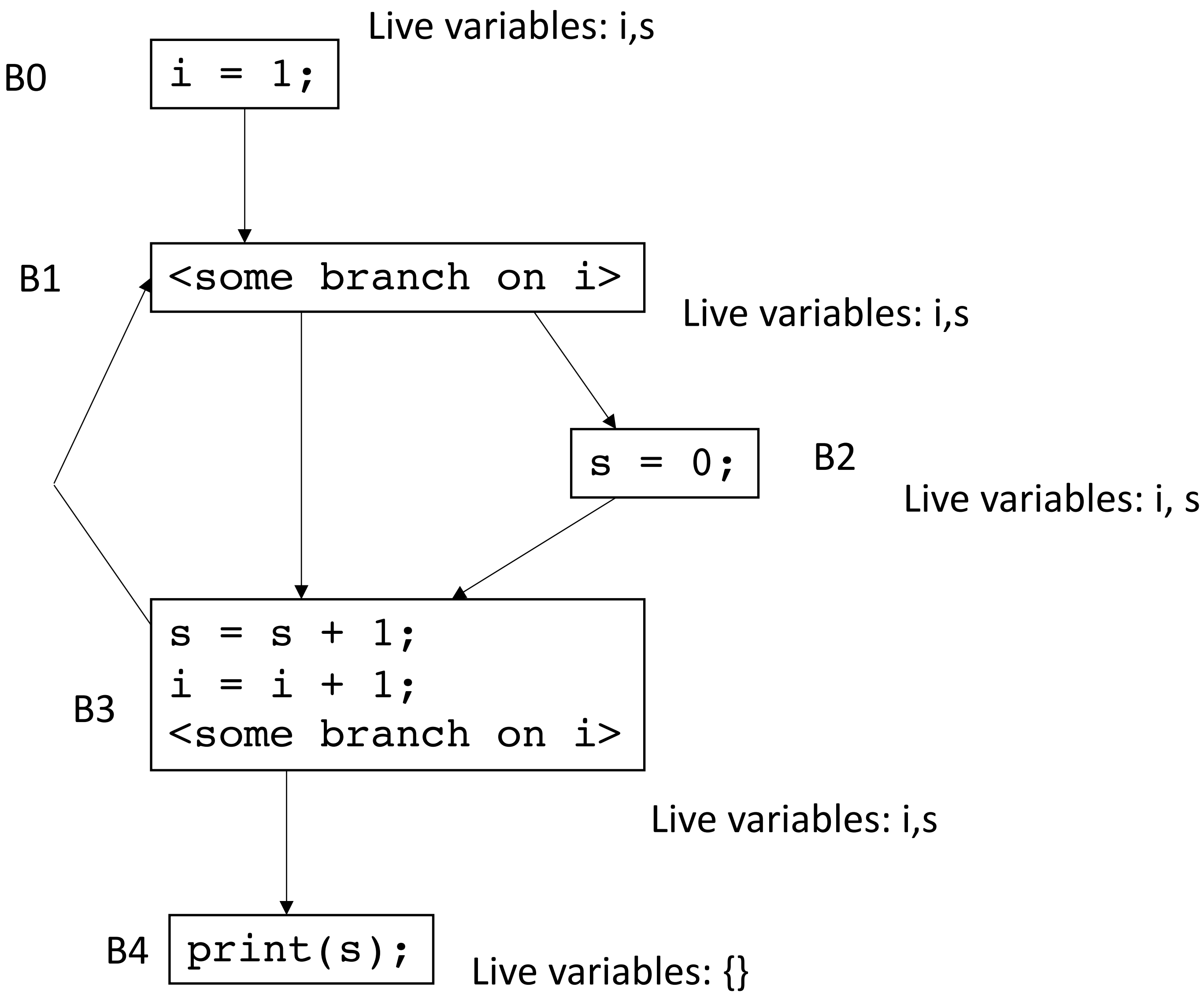


For each block B_x : we want to compute LiveOut:
The set of variables that are live at the end of B_x

Live variable analysis in the CFG:



Live variable analysis in the CFG:



Live variables: i,s

B0
```
i = 1;
```

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

Live variables: i,s

B2
```
s = 0;
```

Live variables: i, s

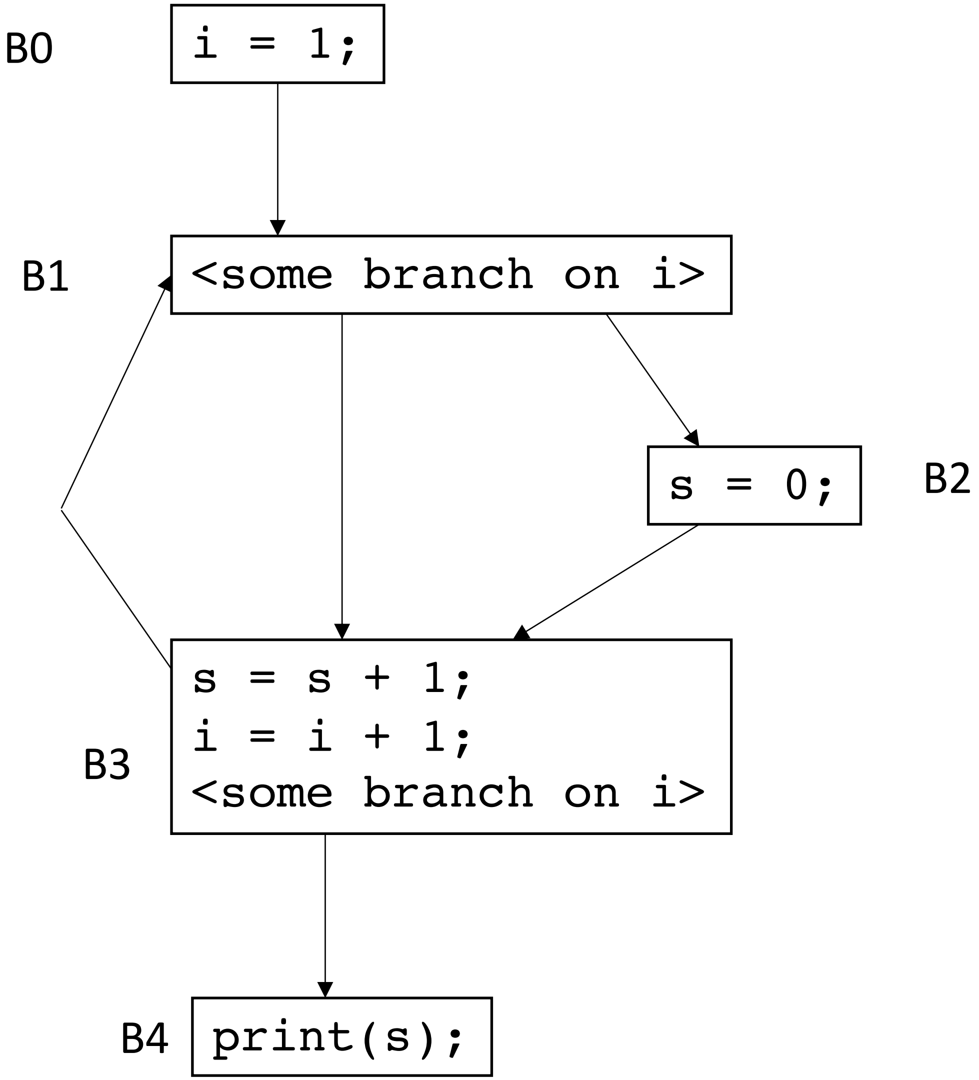
B3
```
s = s + 1;
i = i + 1;
<some branch on i>
```

Live variables: i,s

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

Live variables: {}

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);
```

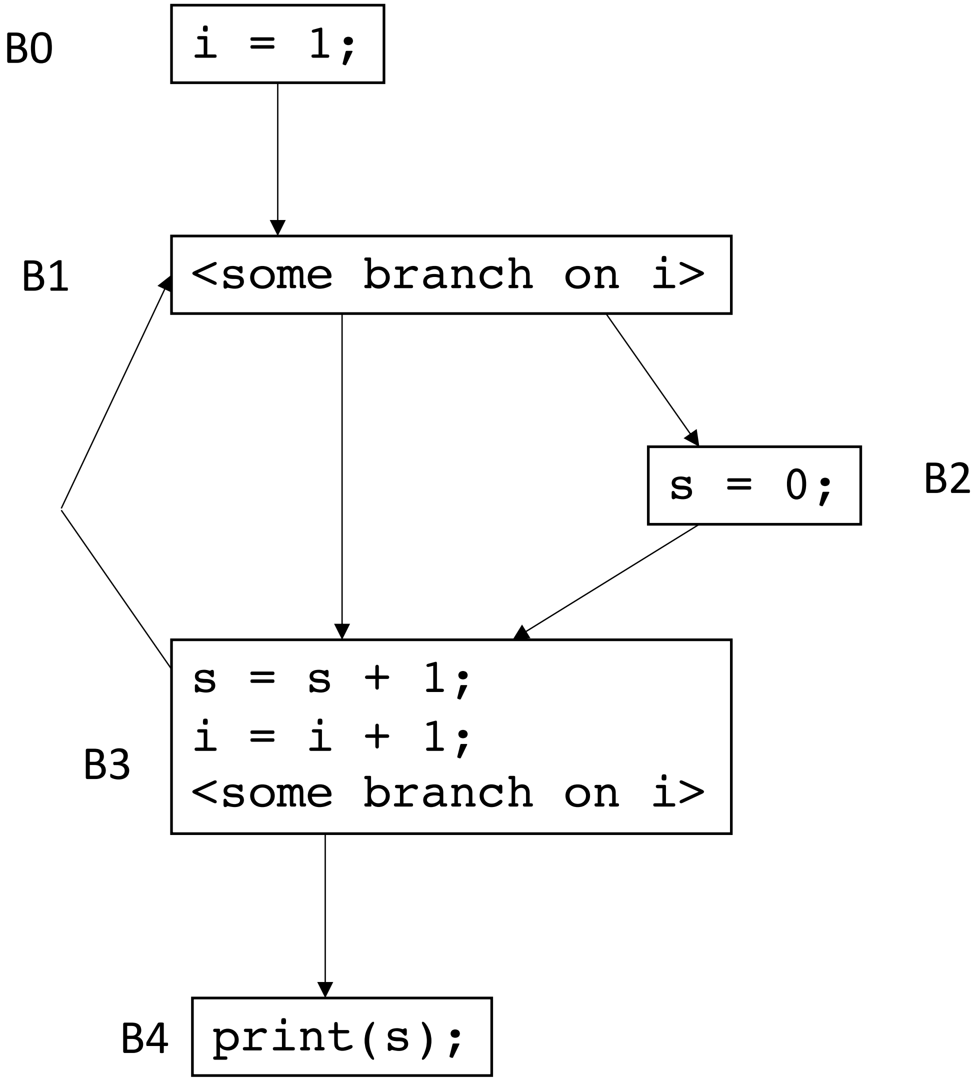
To compute the LiveOut sets, we need two initial sets:

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 satisfies these two conditions
- it is read and it is not written to
- 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:

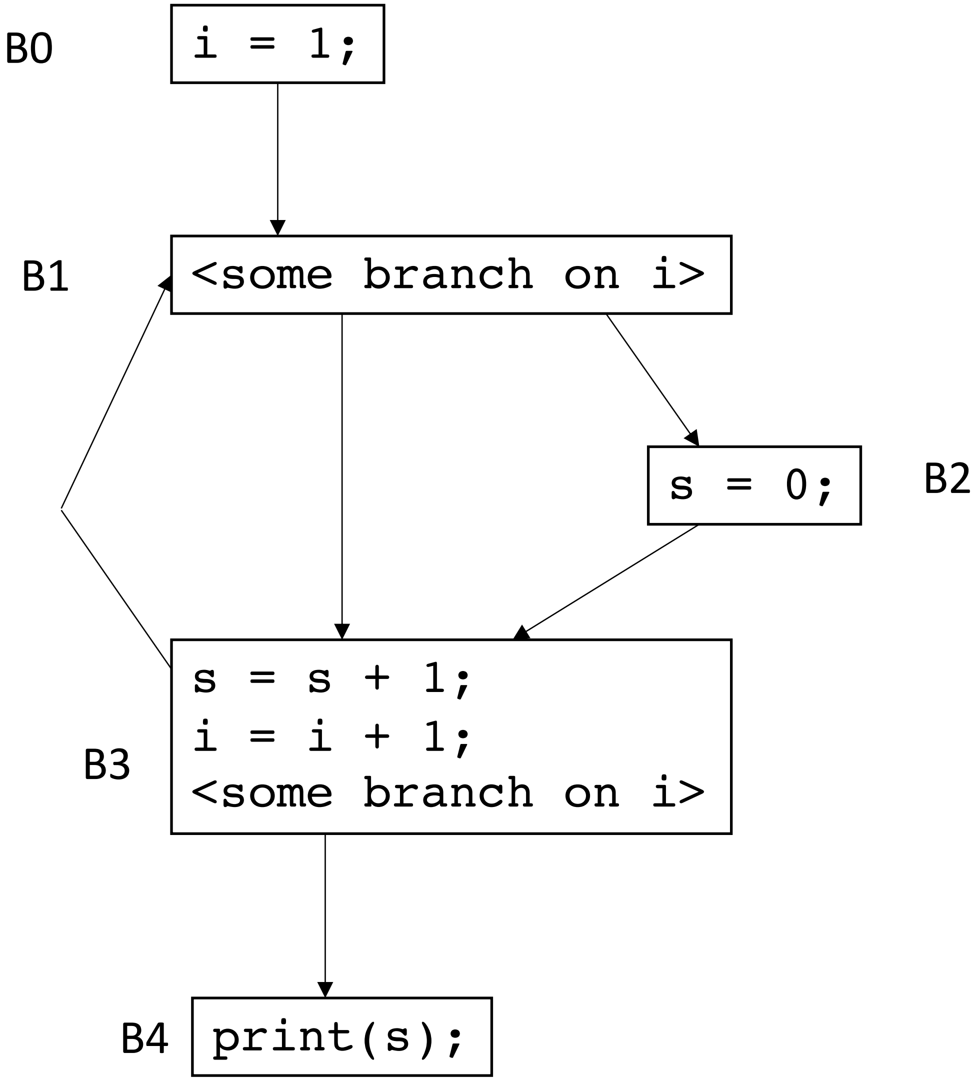
VarKill for block b is any variable in block b that gets overwritten

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

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

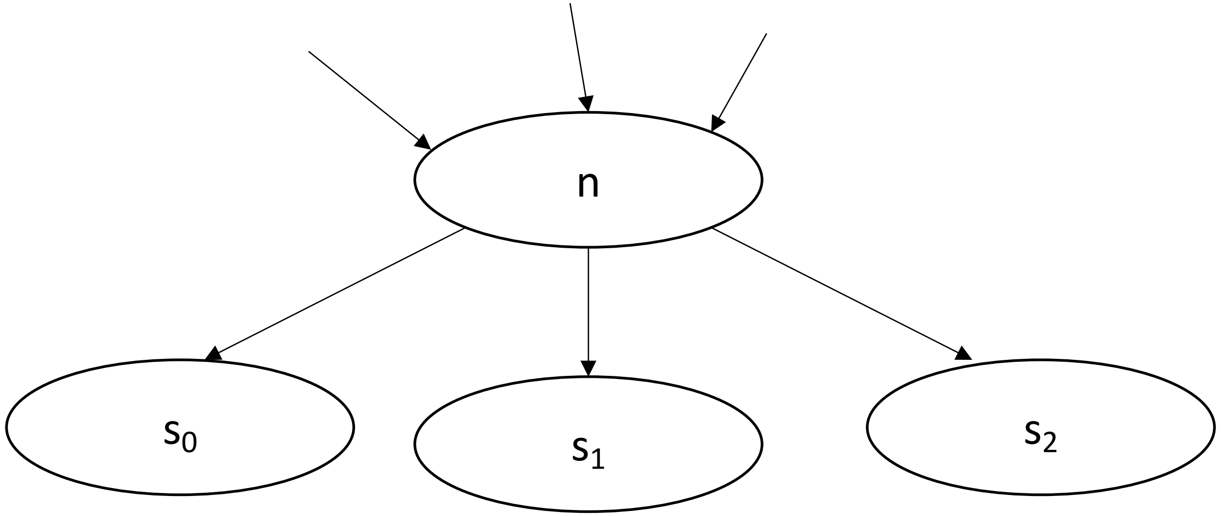
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed point computation:

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

Live variable analysis in the CFG:

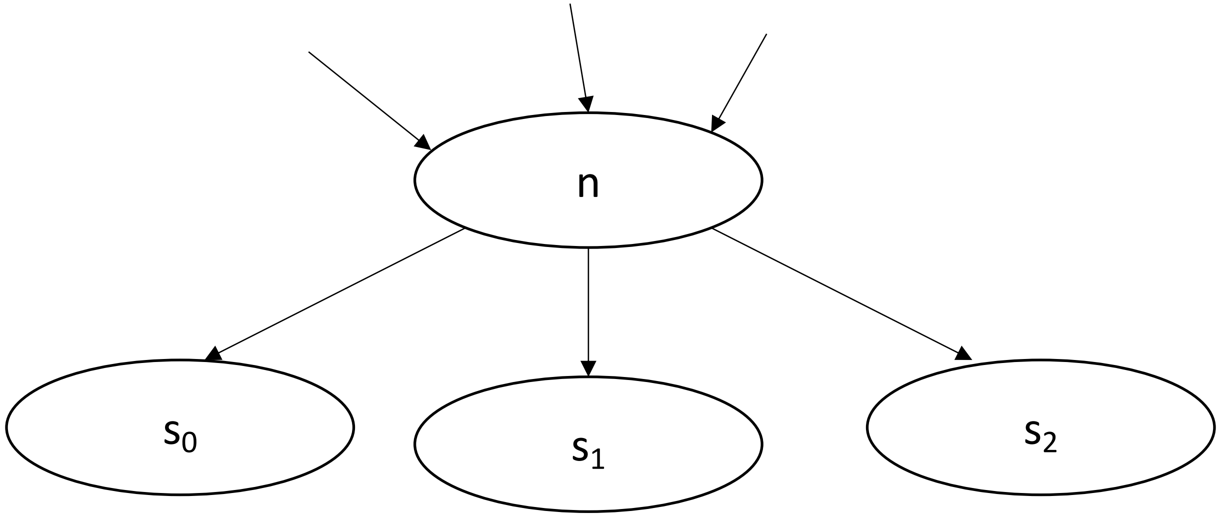
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



Backwards flow analysis because values flow from successors

Live variable analysis in the CFG:

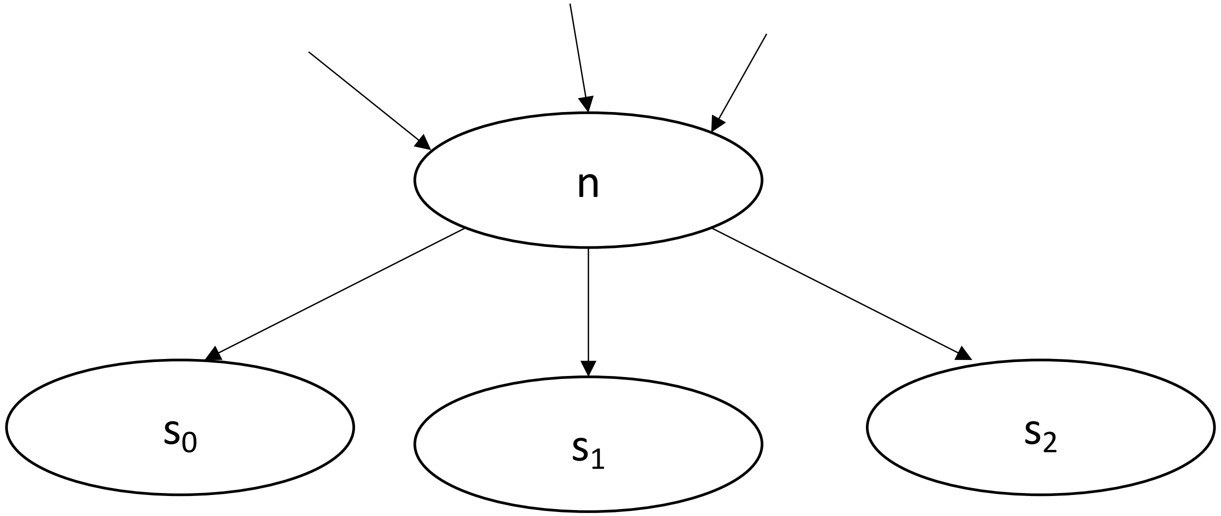
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)} \right) \right)$$



any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

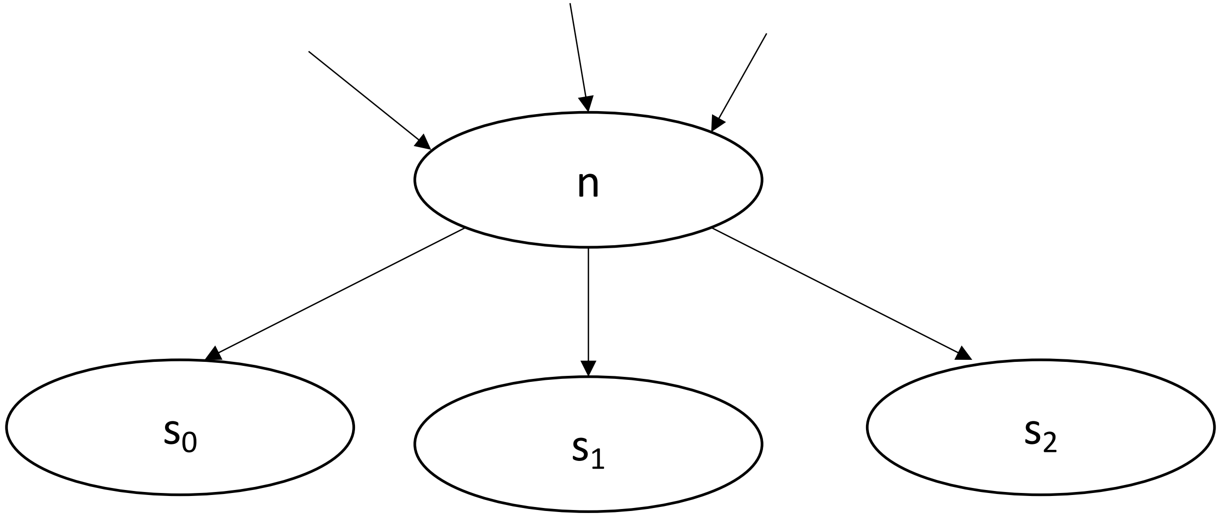
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



variables that are not overwritten in s

Live variable analysis in the CFG:

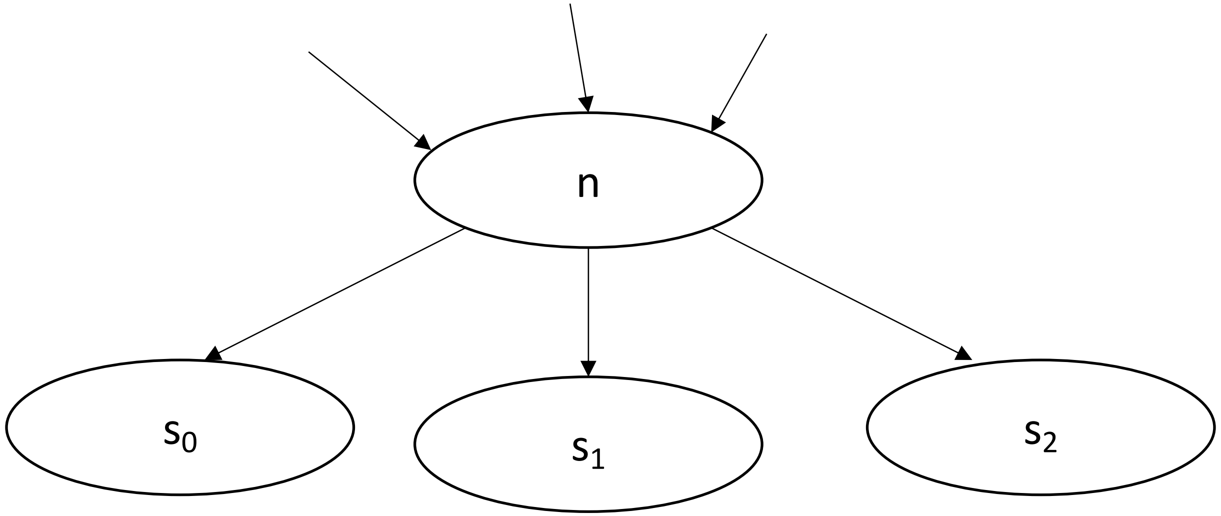
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\,\right)\right)$$



variables that are live
at the end of s

Live variable analysis in the CFG:

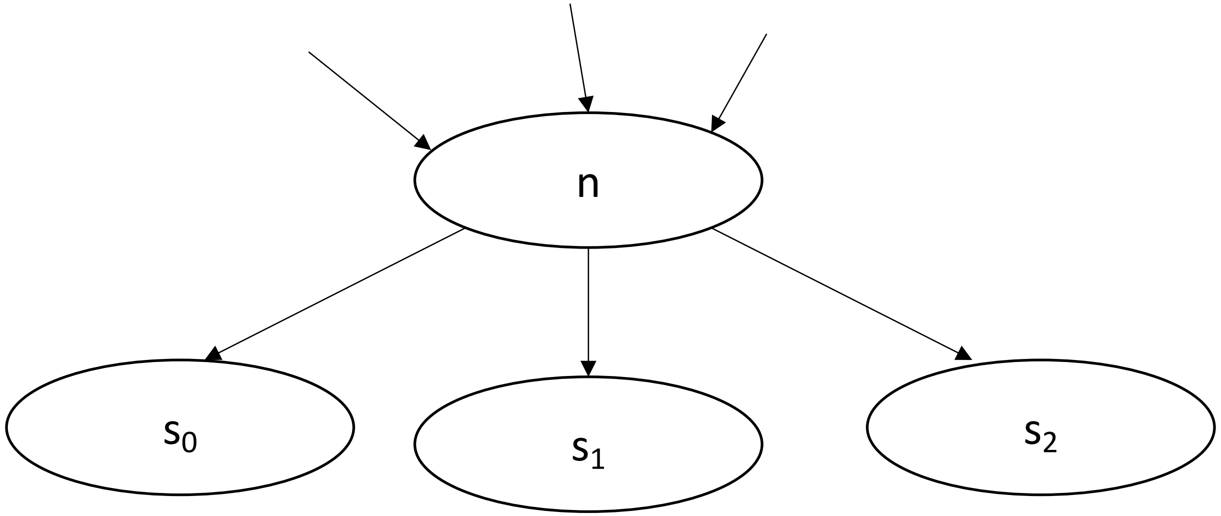
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$



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

Live variable analysis in the CFG:

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



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:

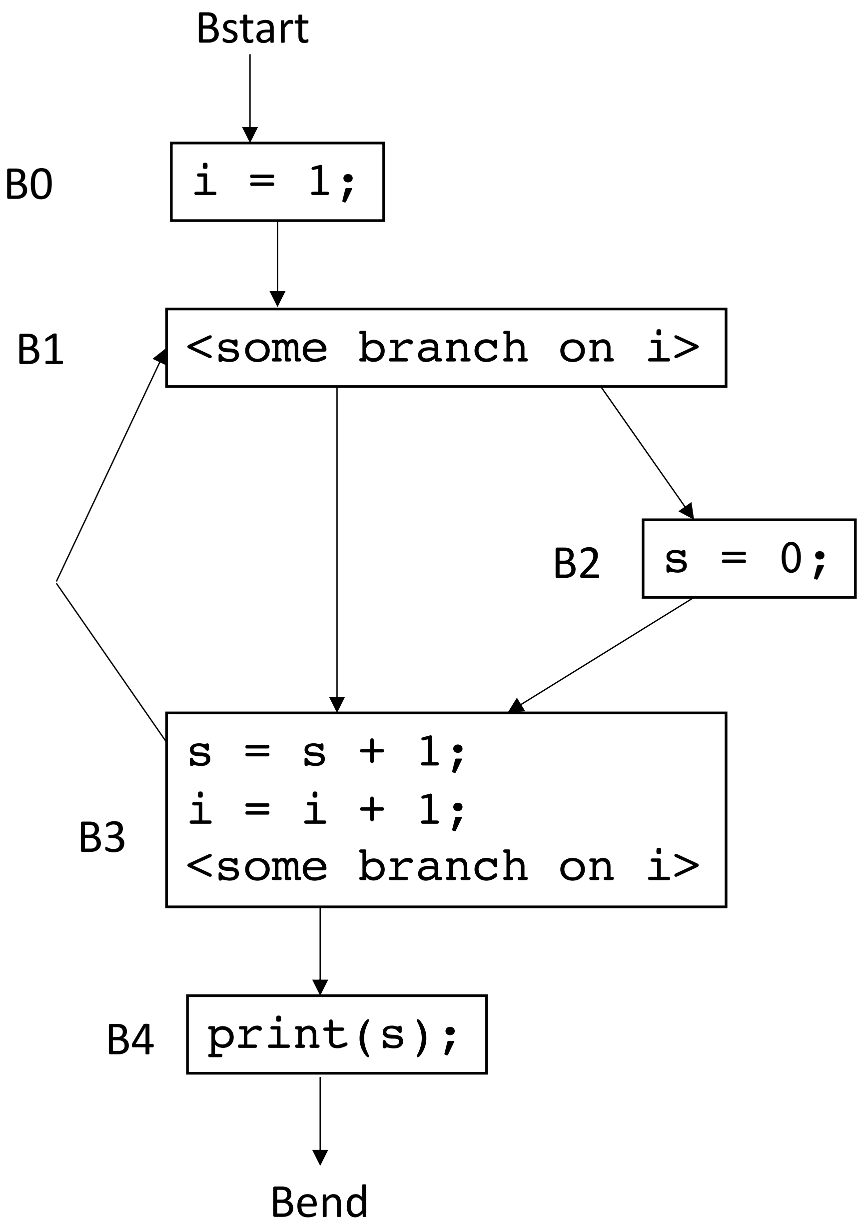
- **Dominance** of node b_x contains b_y if:
    - **every** path from the start to b_x goes through b_y

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

- *Some vs. Every*

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

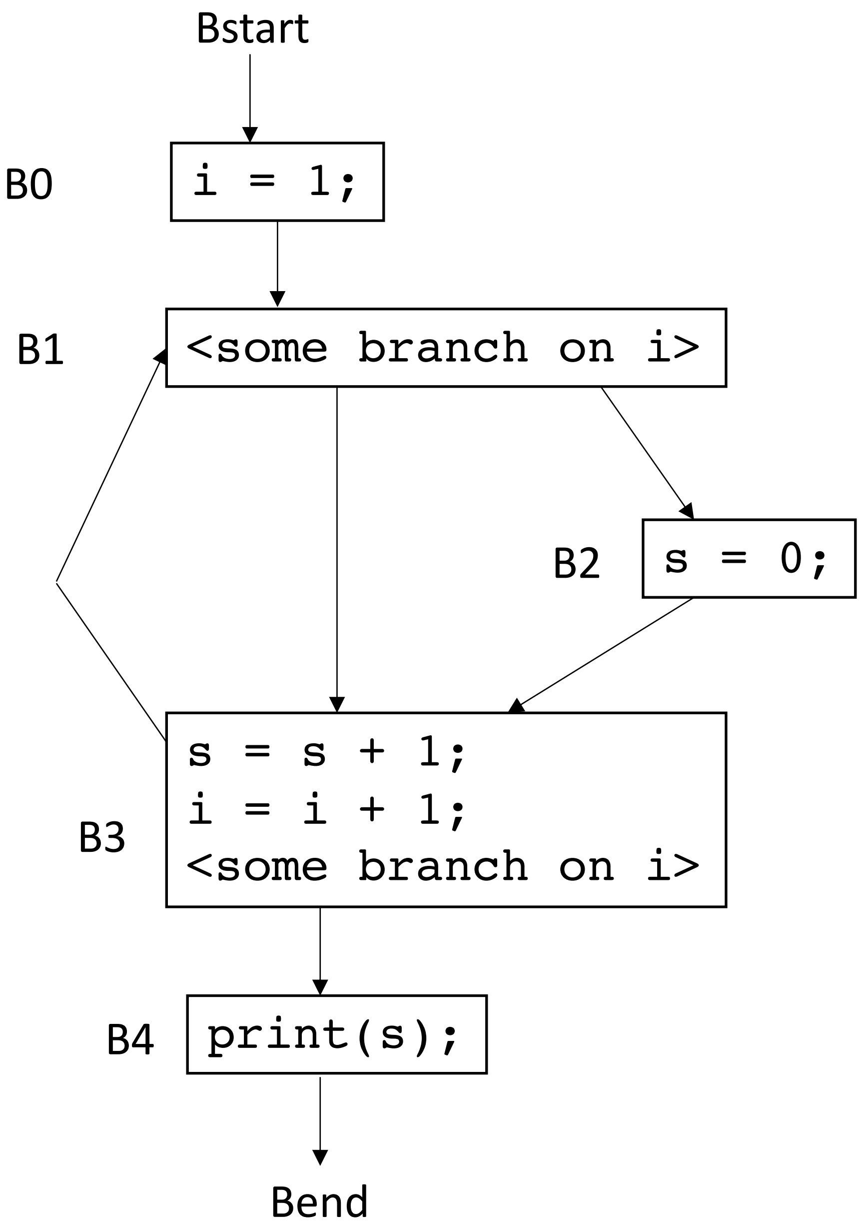
$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in preds(n)}} Dom(p) \right)$$



Now we can perform the iterative fixed point computation:

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

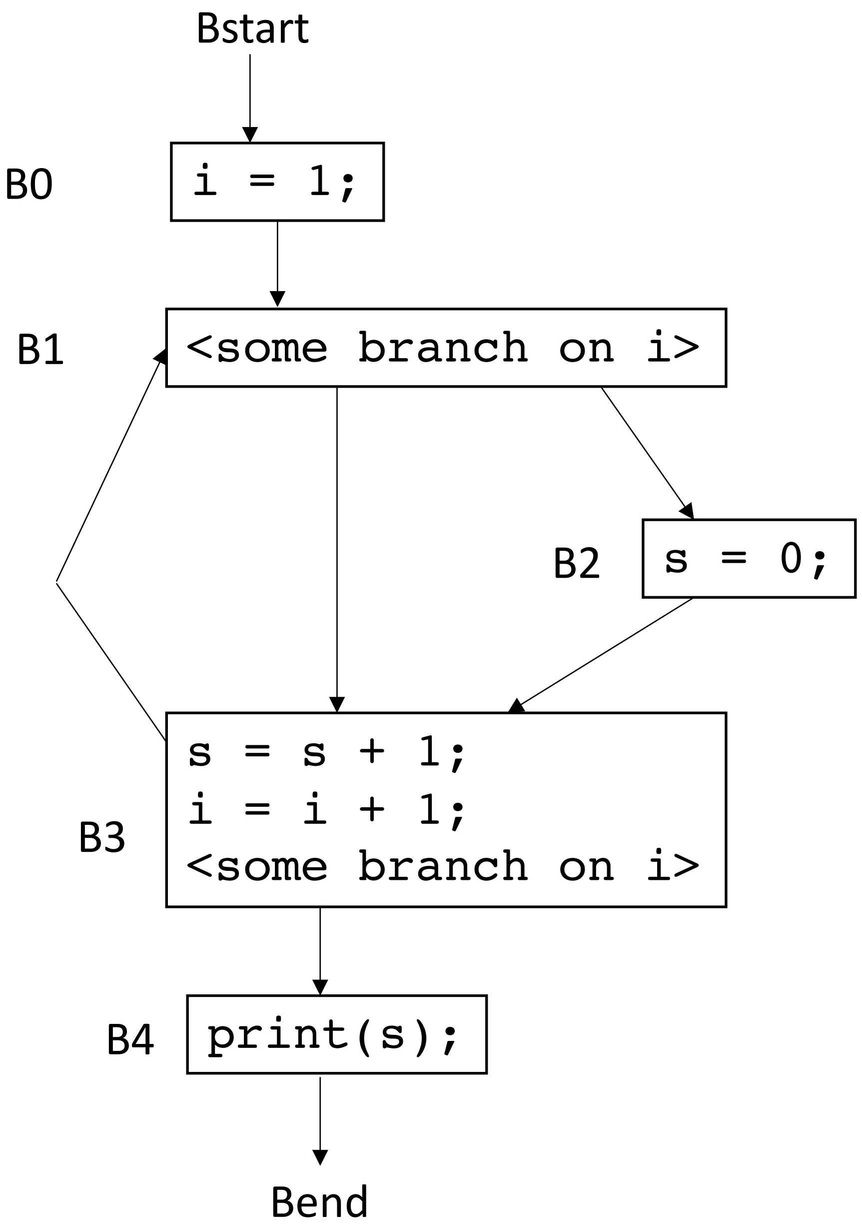
Block	VarKill	UEVar	~VarKill	LiveOut I_0
Bstart	{}	{}	i,s	
B0	i	{}	s	
B1	{}	i	i,s	
B2	s	{}	i	
B3	i,s	i,s	{}	
B4	{}	s	i,s	
Bend	{}	{}	i,s	



Now we can perform the iterative fixed point computation:

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

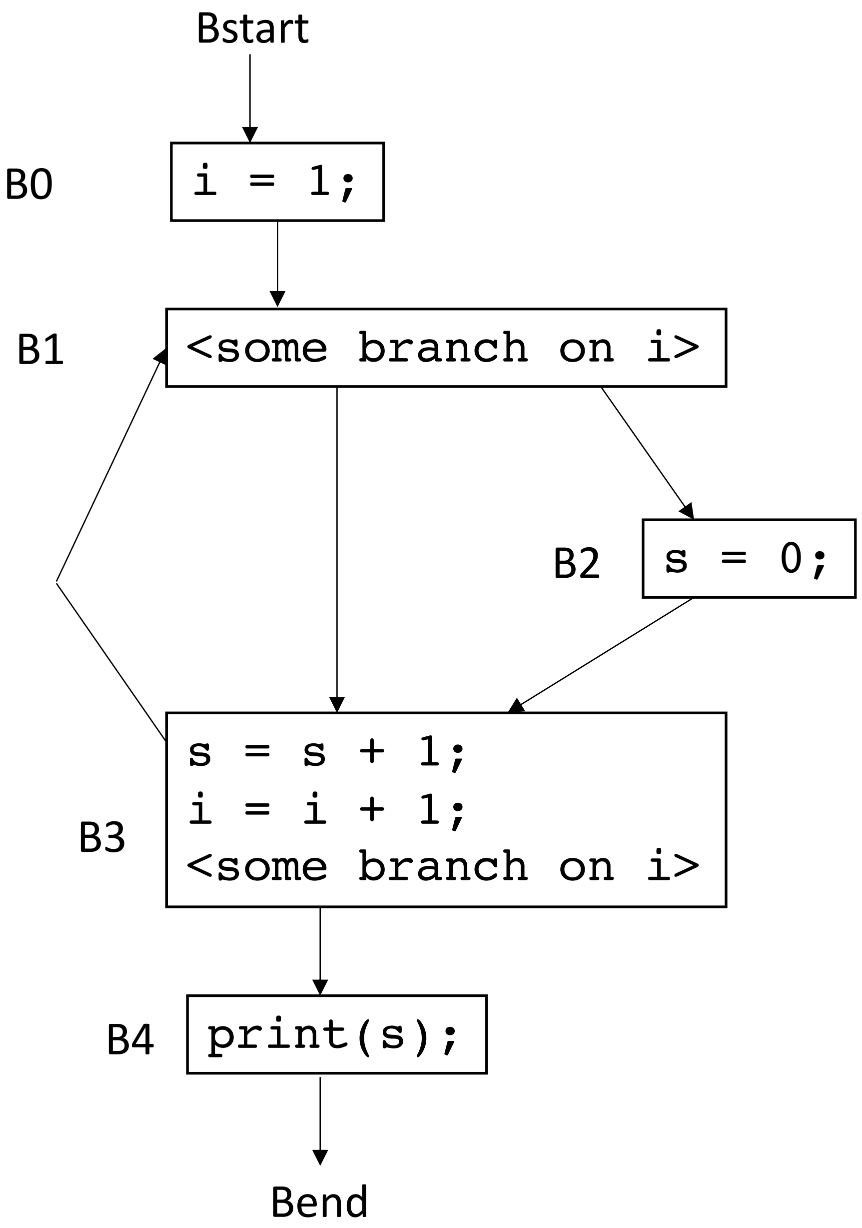
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1
Bstart	{}	{}	i,s	{}	
B0	i	{}	s	{}	
B1	{}	i	i,s	{}	
B2	s	{}	i	{}	
B3	i,s	i,s	{}	{}	
B4	{}	s	i,s	{}	
Bend	{}	{}	i,s	{}	



Now we can perform the iterative fixed point computation:

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

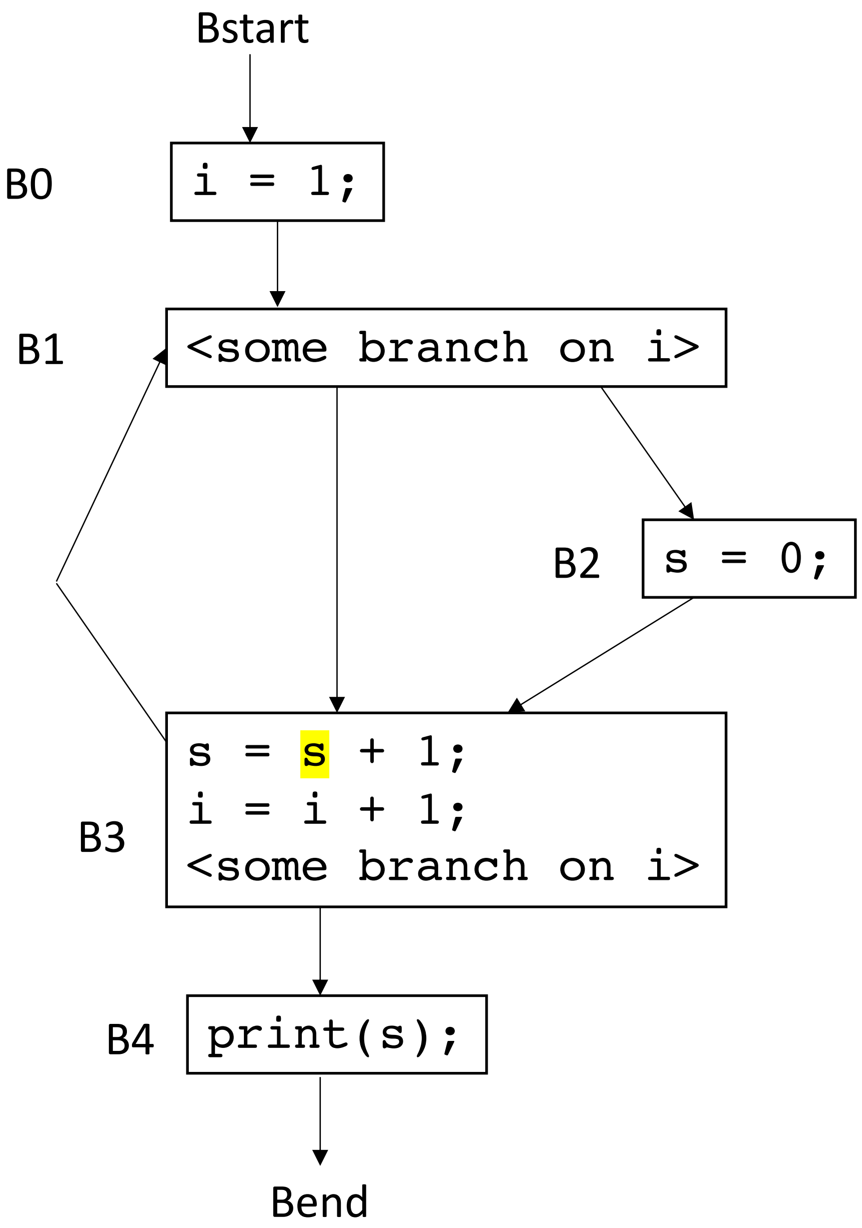
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	
B0	i	{}	s	{}	i	
B1	{}	i	i,s	{}	i,s	
B2	s	{}	i	{}	i,s	
B3	i,s	i,s	{}	{}	i,s	
B4	{}	s	i,s	{}	{}	
Bend	{}	{}	i,s	{}	{}	



Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2	.. I_3
Bstart	{}	{}	i,s	{}	{}	{}	
B0	i	{}	s	{}	i	i,s	
B1	{}	i	i,s	{}	i,s	i,s	
B2	s	{}	i	{}	i,s	i,s	
B3	i,s	i,s	{}	{}	i,s	i,s	
B4	{}	s	i,s	{}	{}	{}	
Bend	{}	{}	i,s	{}	{}	{}	



Bstart

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);
```

Bend

Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2	.. I_3
Bstart	{}	{}	i,s	{}	{}	{}	s
B0	i	{}	s	{}	i	i,s	i,s
B1	{}	i	i,s	{}	i,s	i,s	i,s
B2	s	{}	i	{}	i,s	i,s	i,s
B3	i,s	i,s	{}	{}	i,s	i,s	i,s
B4	{}	s	i,s	{}	{}	{}	{}
Bend	{}	{}	i,s	{}	{}	{}	{}

Node ordering for backwards flow

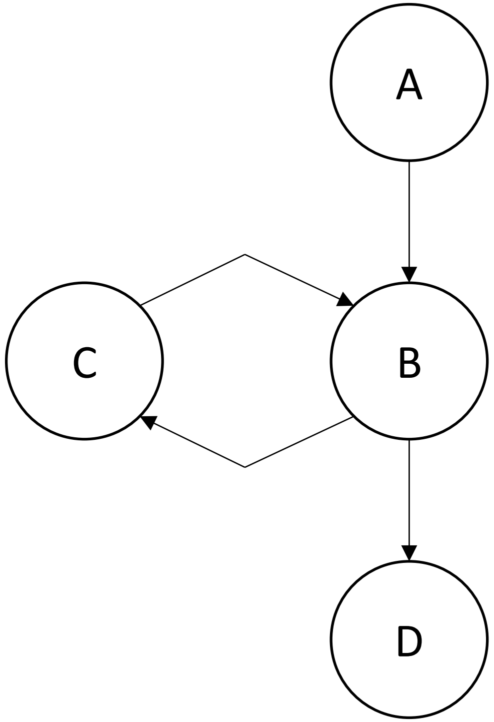
- Reverse post-order was good for forward flow:
    - Parents are computed before their children

- For backwards flow: use reverse post-order of the reverse CFG
    - Reverse the CFG
    - perform a reverse post-order

- Different from post order?

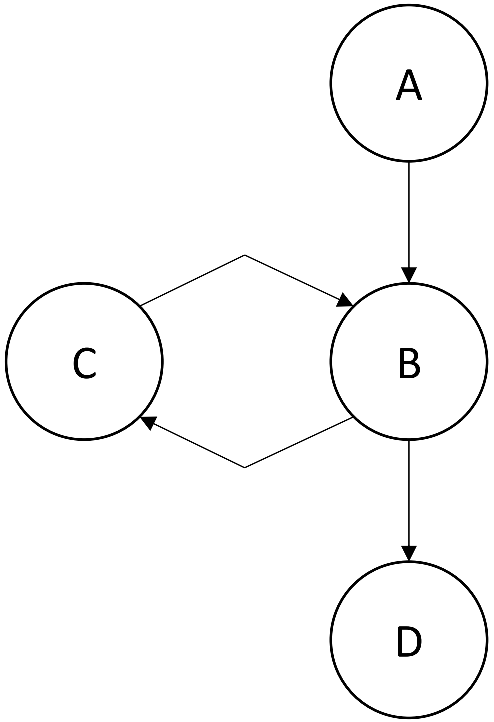
Example

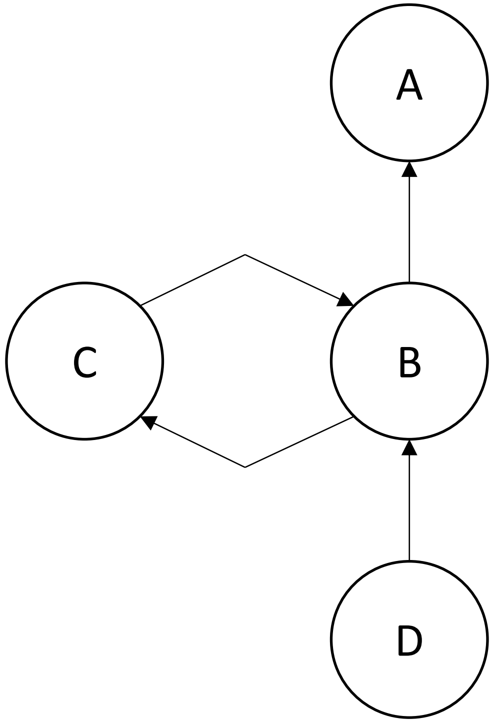
post order: D, C, B, A



acks: thanks to this blog post for the example!
https://eli.thegreenplace.net/2015/directed-graph-traversal-orderings-and-applications-to-data-flow-analysis/

Example



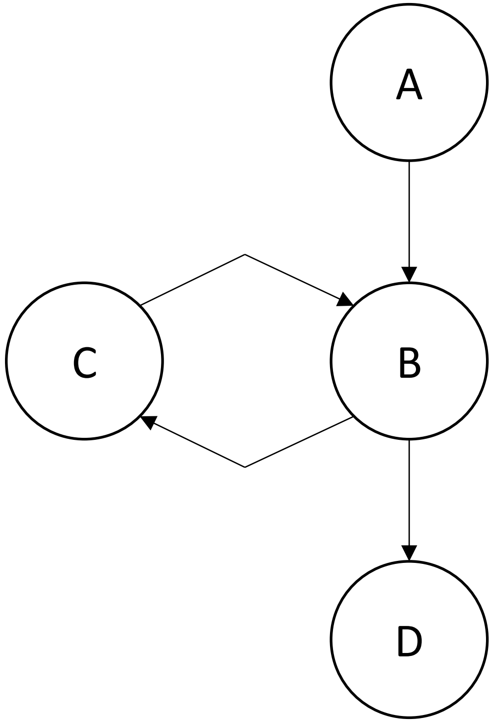


reverse CFG

post order: D, C, B, A

rpo on reverse CFG: D, B, C, A

Example



post order: D, C, B, A

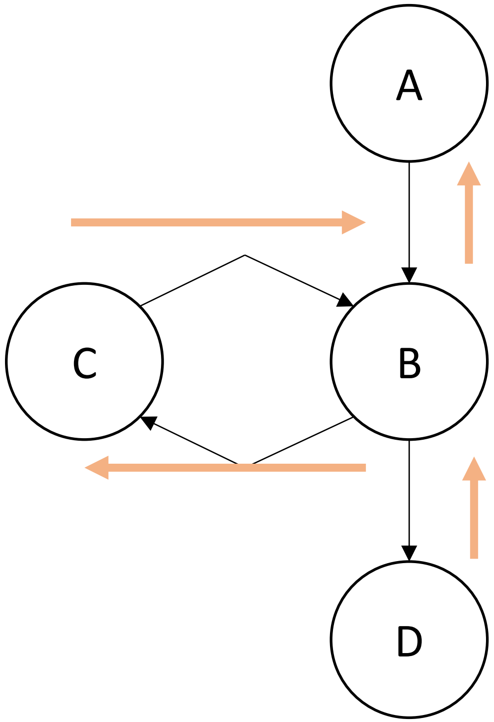
rpo on reverse CFG: D, B, C, A

rpo on reverse CFG computes B before C, thus, C can see updated information from B

Example

post order: D, C, B, A

rpo on reverse CFG: D, B, C, A



updates in backwards flow

rpo on reverse CFG computes B before C, thus, C can see updated information from B

Show PyCFG example from homework

- run the `print_dot.py` command on some test cases to see the output

Live variable limitations

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.

Consider:

```
s = a[x] + 1;
```

Live variable limitations

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.

Consider:

```
s = a[x] + 1;
```

UEVar *needs to assume a[x] is any memory location that it cannot prove non-aliasing*

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

Live variable limitations

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.

Consider:

```
a[x] = s + 1;
```

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

Live variable limitations

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.

Consider:

```
a[x] = s + 1;
```

***VarKill** also needs to know about aliasing*

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

Demo

- Godbolt demo

Sound vs. Complete

- Sound: Any property the analysis says is true, is true. However, there may be false positives

- Complete: Any error the analysis reports is actually an error. The analysis cannot prove a property though.

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

How to instantiate the UEVar and VarKill for sound/complete analysis w.r.t. memory?

`a[x] = s + 1;`

`s = a[x] + 1;`

Live variable limitations

Imprecision can come from CFG construction:

consider:

```
br 1 < 0, dead_branch, alive_branch
```

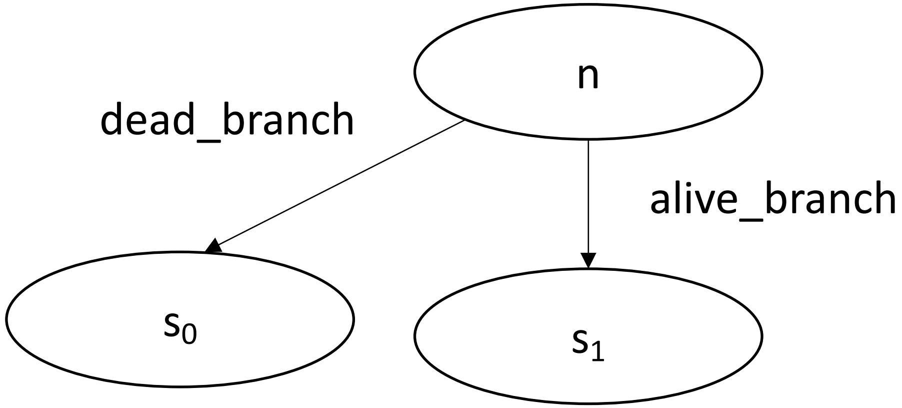
Live variable limitations

Imprecision can come from CFG construction:

consider:

```
br 1 < 0, dead_branch, alive_branch
```

could come from arguments, etc.



Live variable limitations

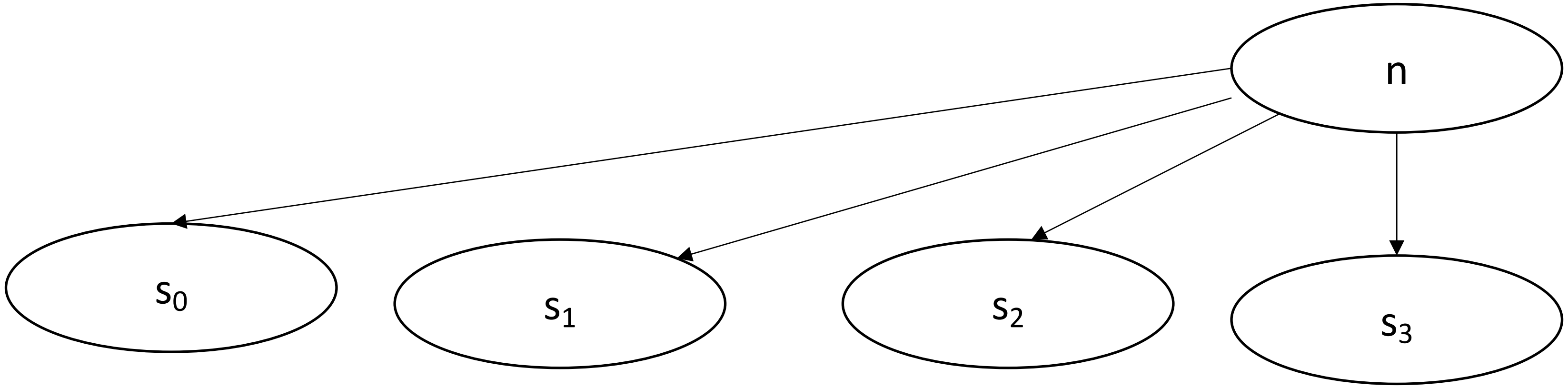
Imprecision can come from CFG construction:

consider first class labels (or functions):

```
br label_reg
```

where label_reg is a register that contains a register

need to branch to all possible basic blocks!



The Data Flow Framework

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

$$f(x) = Op_{v\ in\ (succ\ |\ preds)}\ c_0(v)\ op_1\ (f(v)\ op_2\ c_2(v))$$

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

An expression e is "available" at the beginning of a basic block b_x if for all paths to b_x, e is evaluated and none of its arguments are overwritten

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

Forward Flow

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

intersection implies "must" analysis

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

DEExpr(p) is all Downward Exposed Expressions in p. That is expressions that are evaluated AND operands are not redefined

Available Expressions

$$\mathit{AvailExpr}(n) = \bigcap_{p \text{ in } preds} \mathit{DEExpr}(p) \cup (\mathit{AvailExpr}(p) \cap \overline{\mathit{ExprKill}(p)})$$

AvailExpr(p) is any expression that is available at p

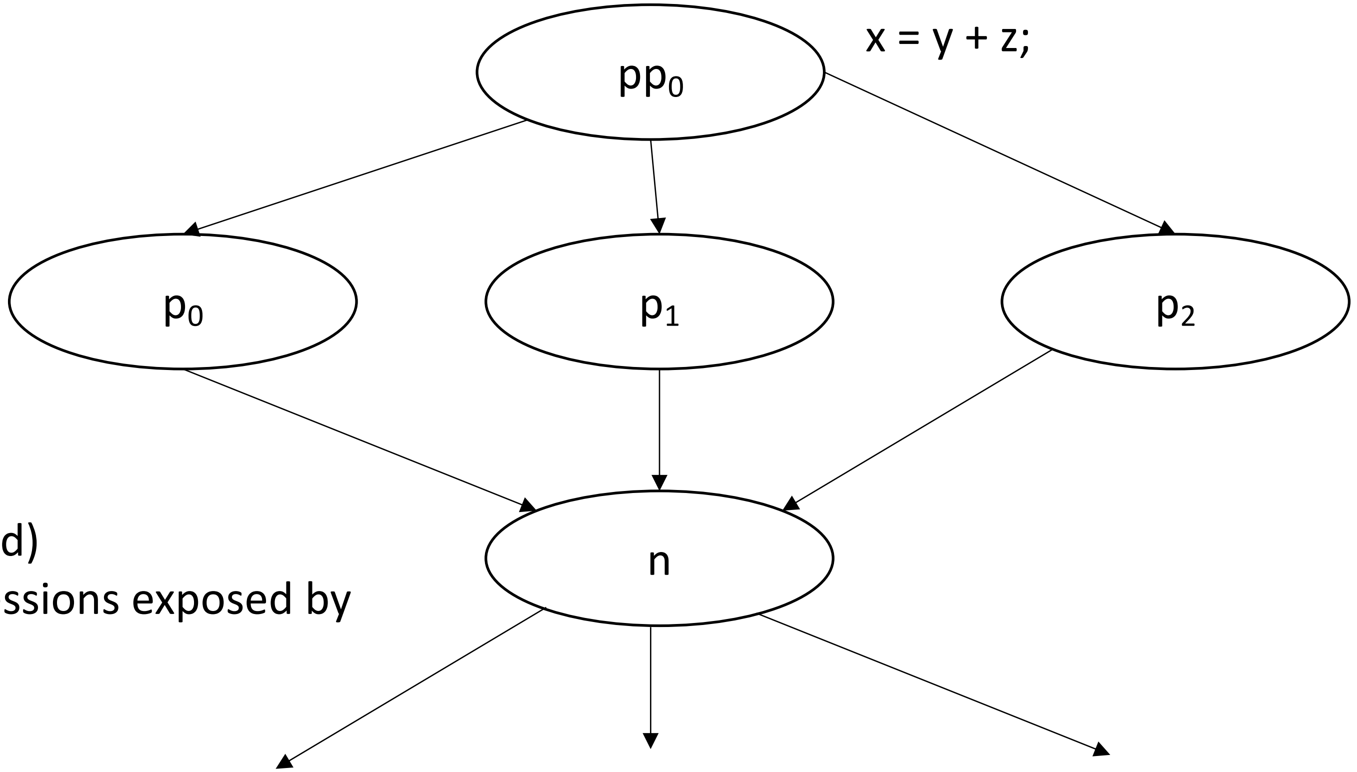
Available Expressions

$$AvailExpr(n)= \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

ExprKill(p) is any expression that p killed, i.e. if one or more of its operands is redefined in p

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$



Any expression
that is available (and not killed)
the parents, along with expressions exposed by
all the parents.

Available Expressions

$$AvailExpr(n) = \bigcap_{p\ in\ preds} DEExpr(p) \cup (AvailExpr(p) \cap \overline{ExprKill(p)})$$

***Application**: you can add availExpr(n) to local optimizations in n, e.g. local value numbering*

Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

An expression e is "anticipable" at a basic block b_x if for all paths that leave b_x, e is evaluated

Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

Backwards flow

Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

”must” analysis

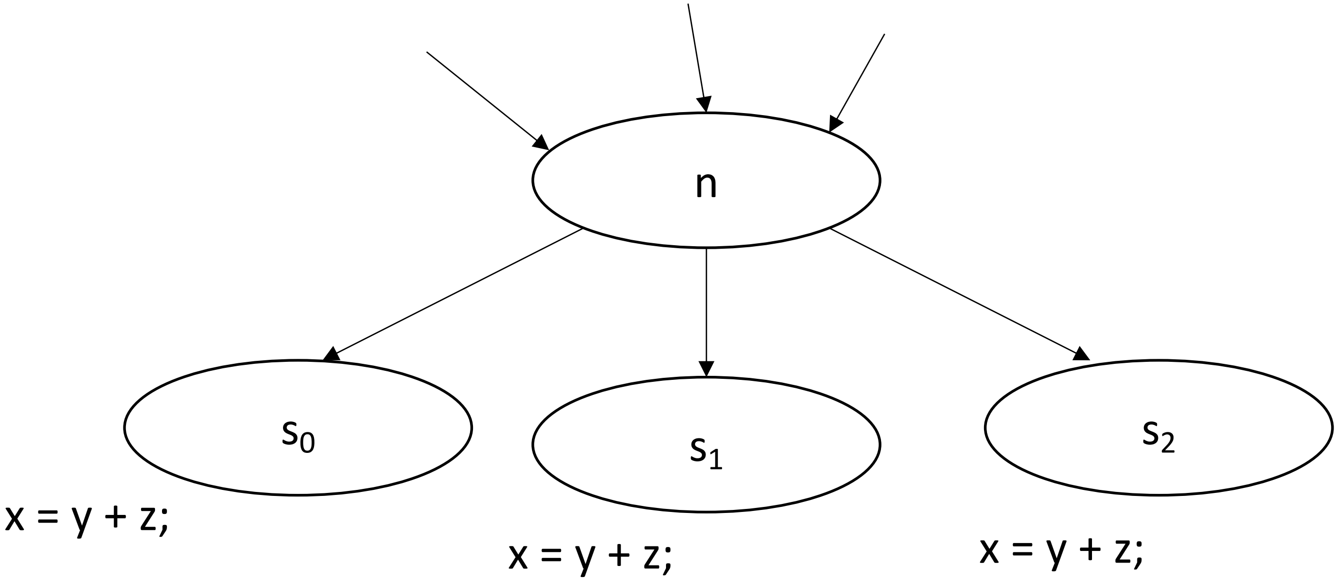
Anticipable Expressions

$$AntOut(n) = \bigcap_{s \in succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

UEExpr(p) is all Upward Exposed Expressions in p. That is expressions that are computed in p before operands are overwritten.

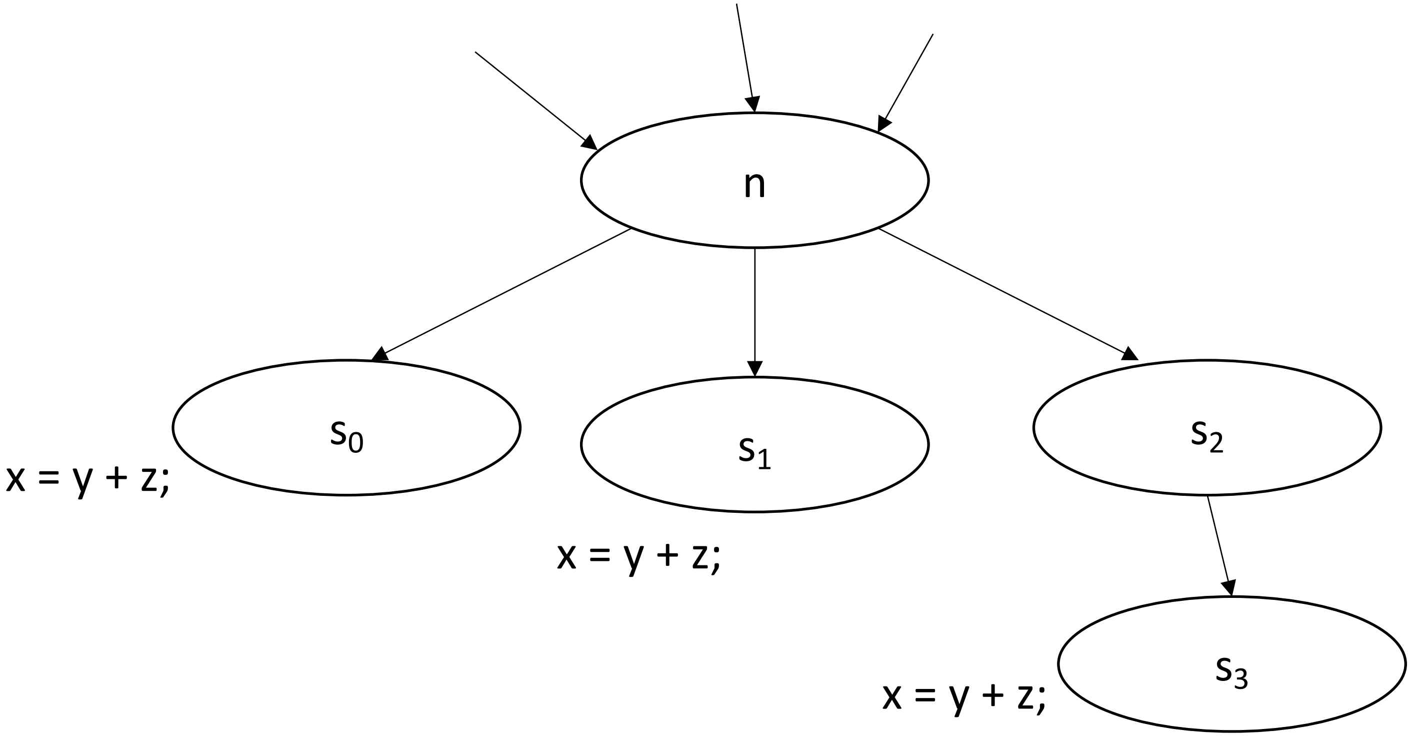
Anticipable Expressions

$$AntOut(n) = \bigcap_{s \text{ in } succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$



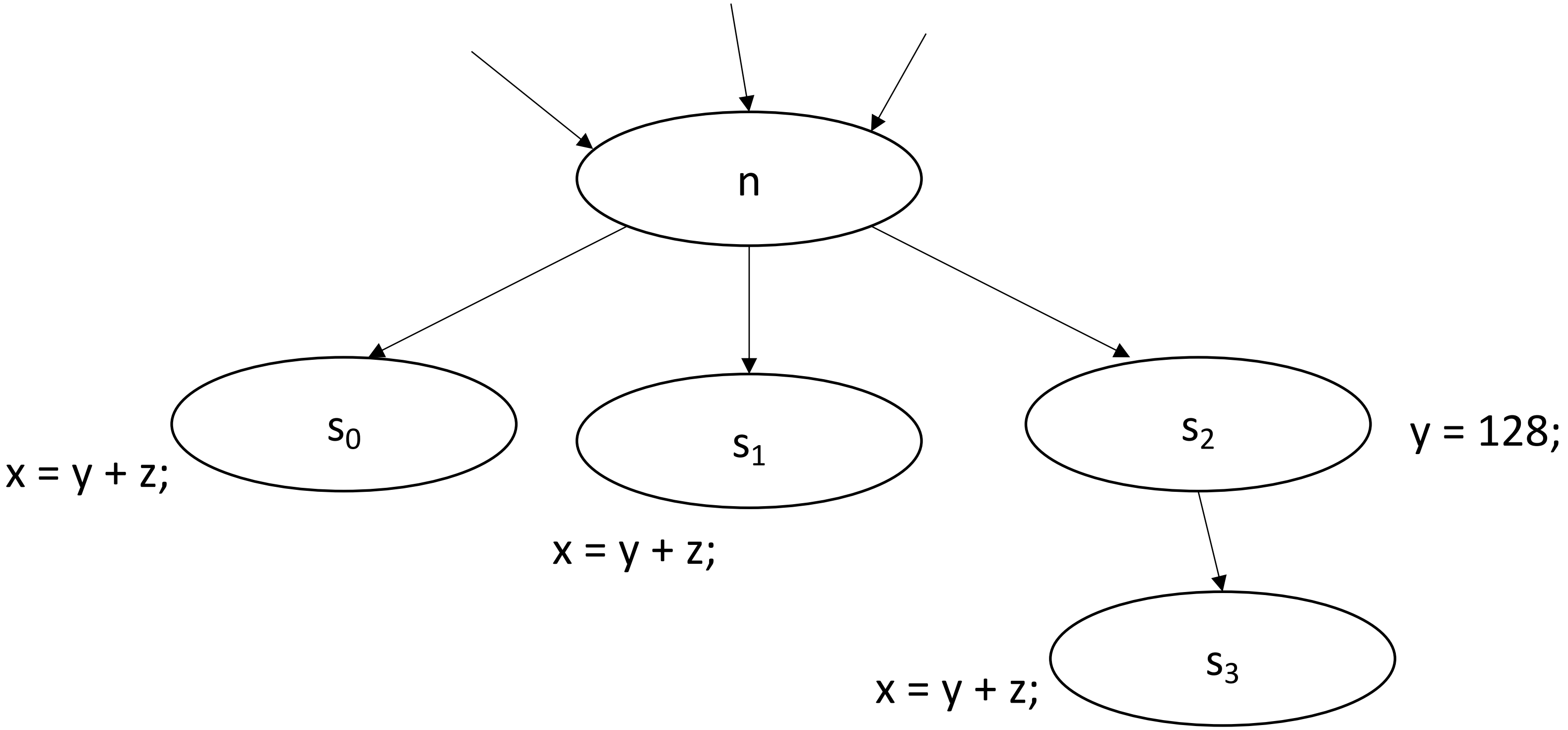
Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$



Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$



Anticipable Expressions

$$AntOut(n) = \bigcap_{s\ in\ succ} UEExpr(s) \cup (AntOut(s) \cap \overline{ExprKill(s)})$$

***Application**: you can hoist AntOut expressions to compute as early as possible*

potentially try to reduce code size: -Oz

More flow algorithms:

Check out chapter 9 in EAC: Several more algorithms.

“Reaching definitions” have applications in memory analysis

Have a nice weekend!

- See you in office hours or in a week!