CSE211: Compiler Design

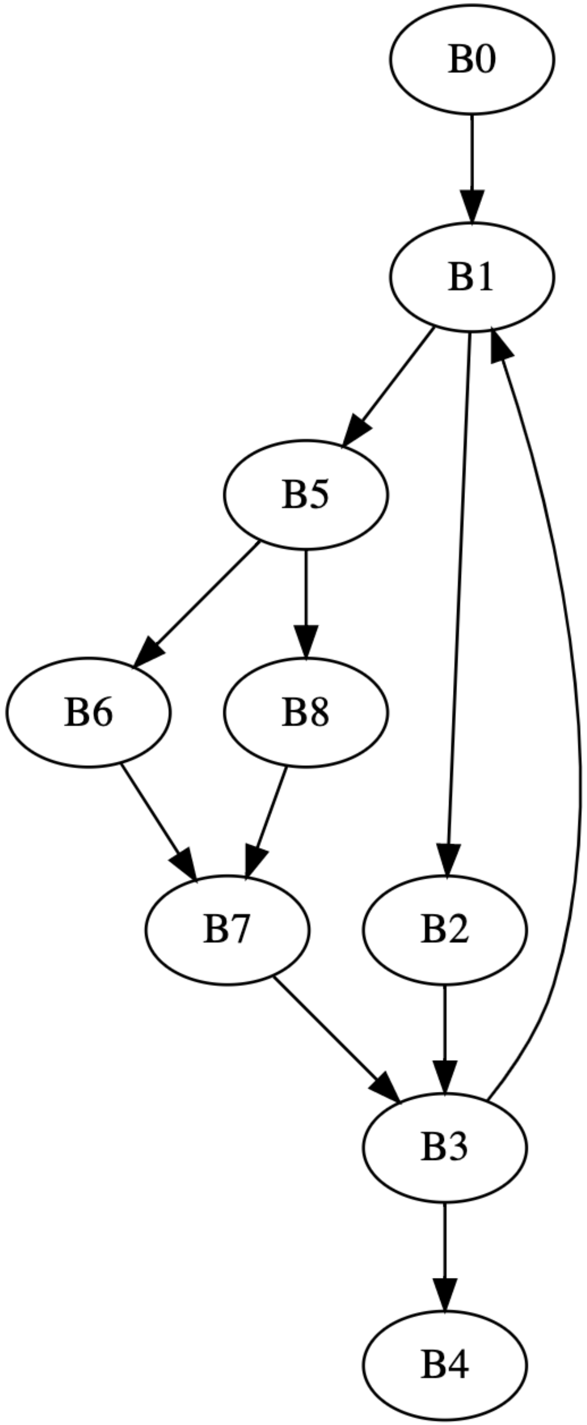
Oct. 27, 2020

- **Topic**: Data Flow Analysis Continued

- **Questions**:

  Questions/comments about homework 1?

  What are some interesting control flow constructs and how do they look in a CFG?



Announcements

- Homework 1 is due on Thursday!
- Office Hours are Wednesday from 3 - 4 PM.
- If you need help with homework 1, message me before hand with a brief summary of your question. I will use this to schedule and potentially make groups

Announcements

- According to the schedule: the last day of module 2.
- But we need to go over SSA form
- Schedule may get moved back a week. (I know people are excited for module 3!)

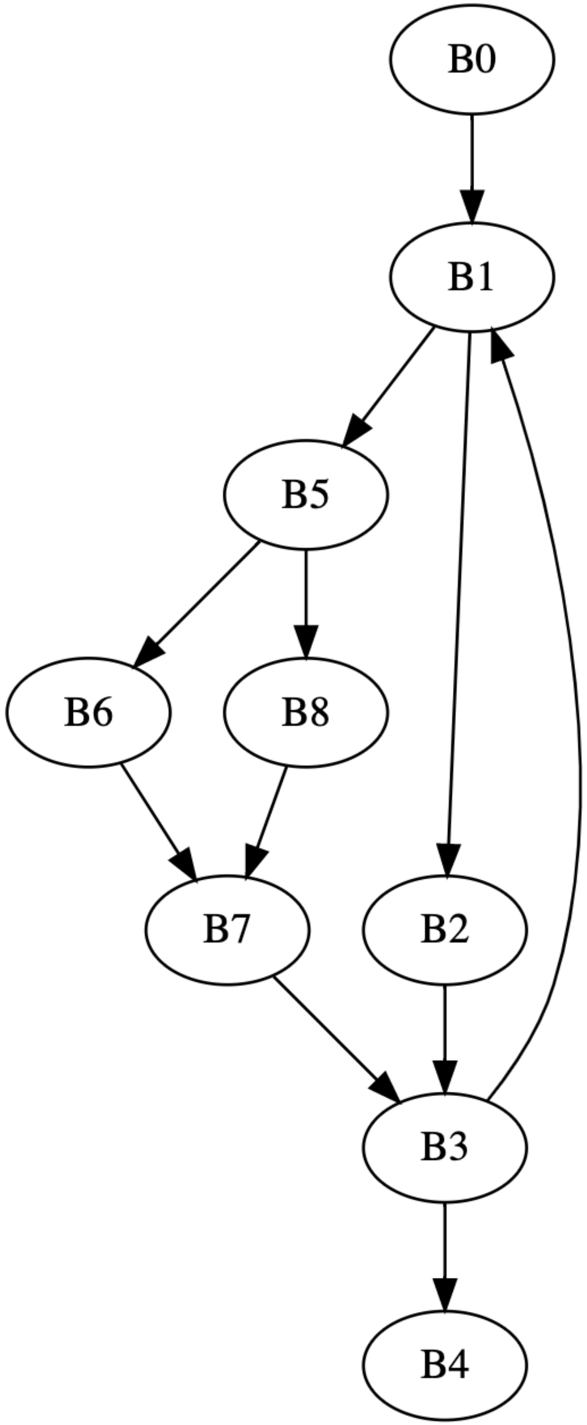
CSE211: Compiler Design

Oct. 27, 2020

- **Topic**: Data Flow Analysis Continued
- **Questions**:

  Questions/comments about homework 1?

  What are some interesting control flow constructs and how do they look in a CFG?



Control Flow Graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

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


if:
r2 = ...;
br end_if;


else:
r3 = ...;


end_if:
r4 = ...;
```

Control Flow Graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another

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

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

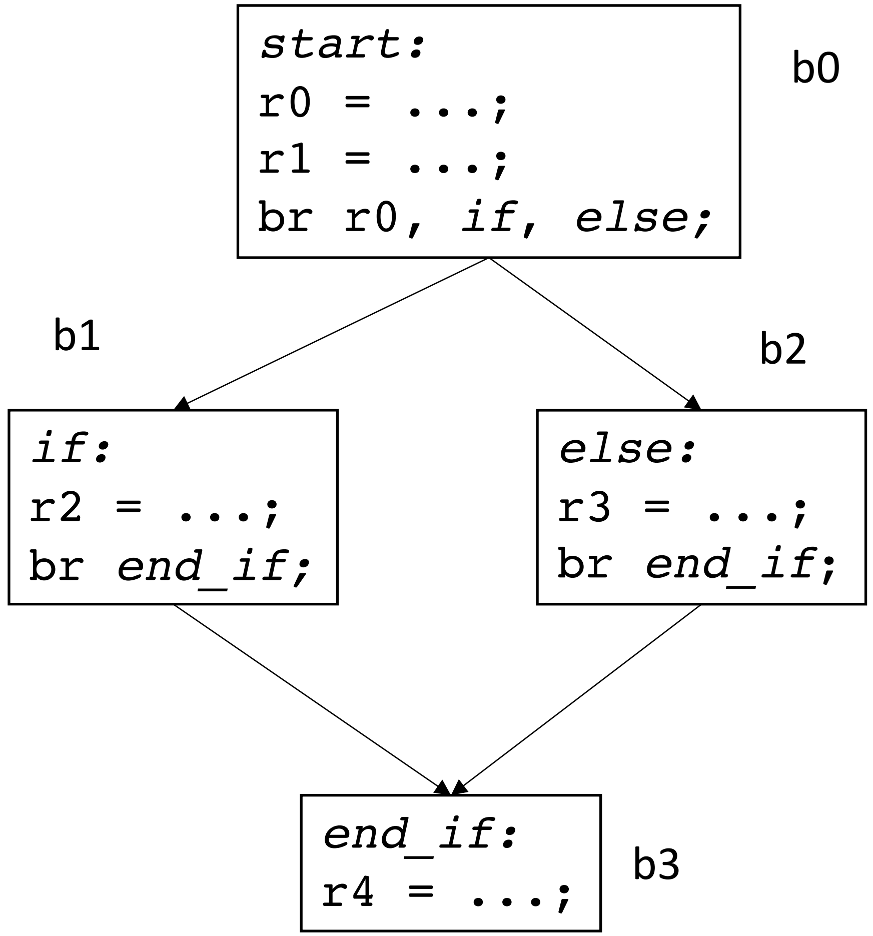
```
else:
r3 = ...;
br end_if;
```

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

Control Flow Graphs

A graph where:

- nodes are basic blocks
- edges mean that it is possible for one block to branch to another



b0

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

b1

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

b2

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

b3

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

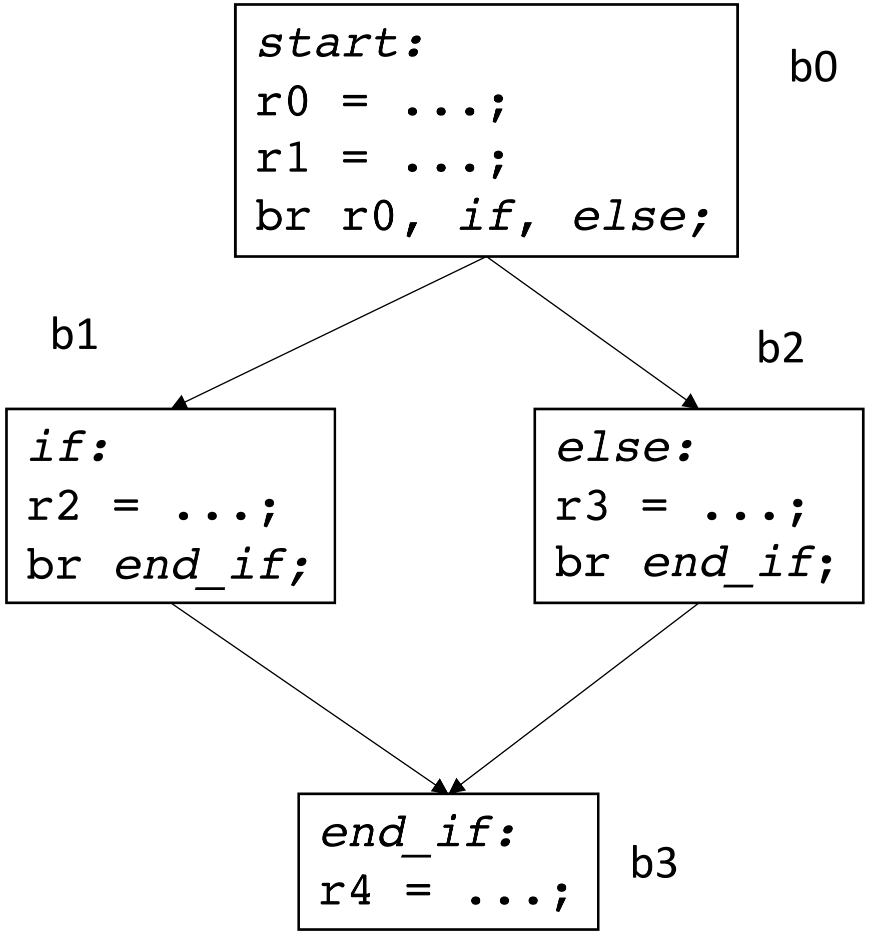
Interesting CFGs

interesting CFGs

- Exceptions
- Break in a loop
- Switch statement (consider break, no break)
- first class branches (or functions)

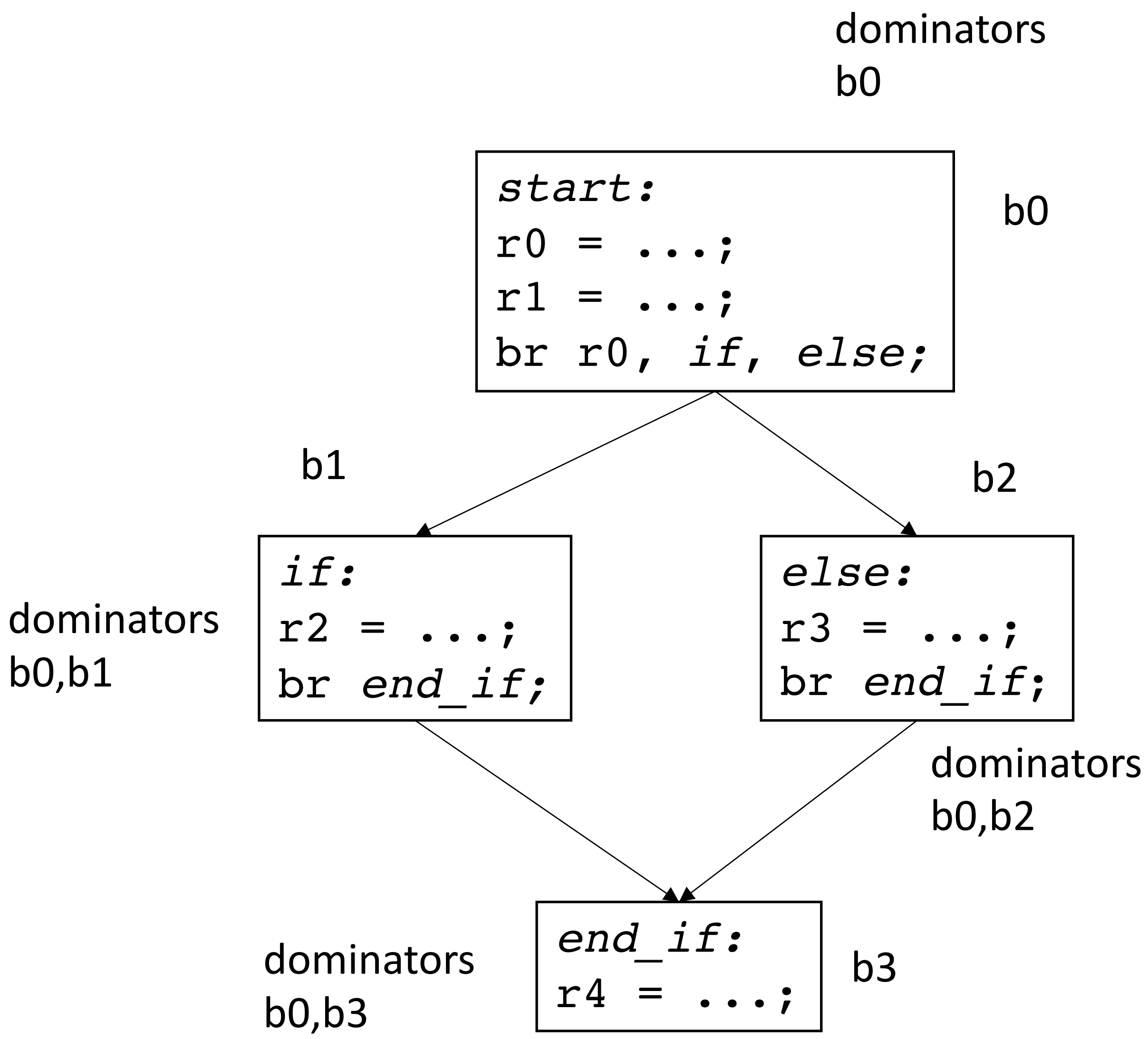
Dominance

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



Dominance

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



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

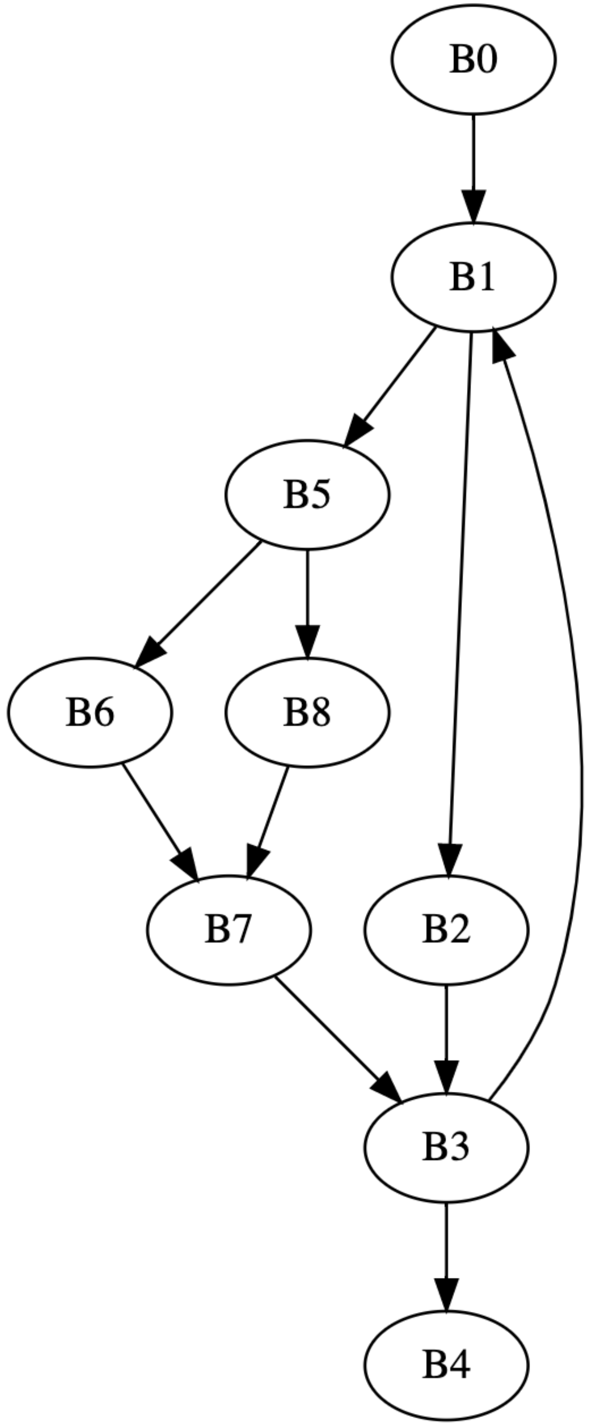
dominators
b0,b3

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

b3

a larger example from last lecture

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



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:

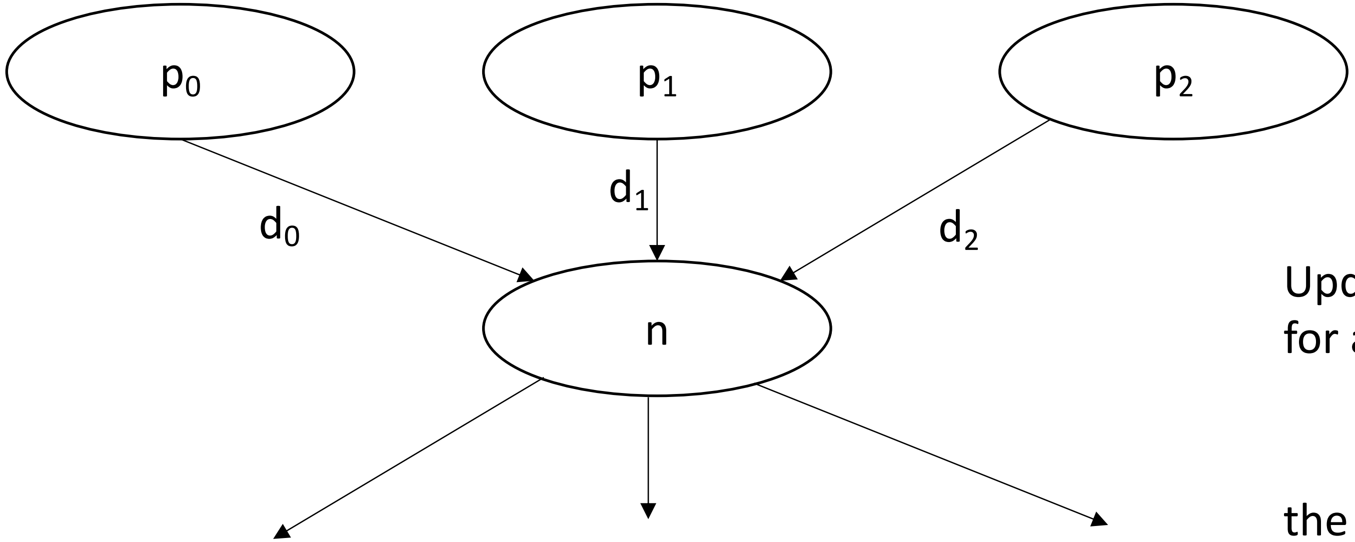
$$Dom(n) = \{n\} \cup \left(\bigcap_{\text{m in preds(n)}} Dom(m) \right)$$

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 graphs and 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

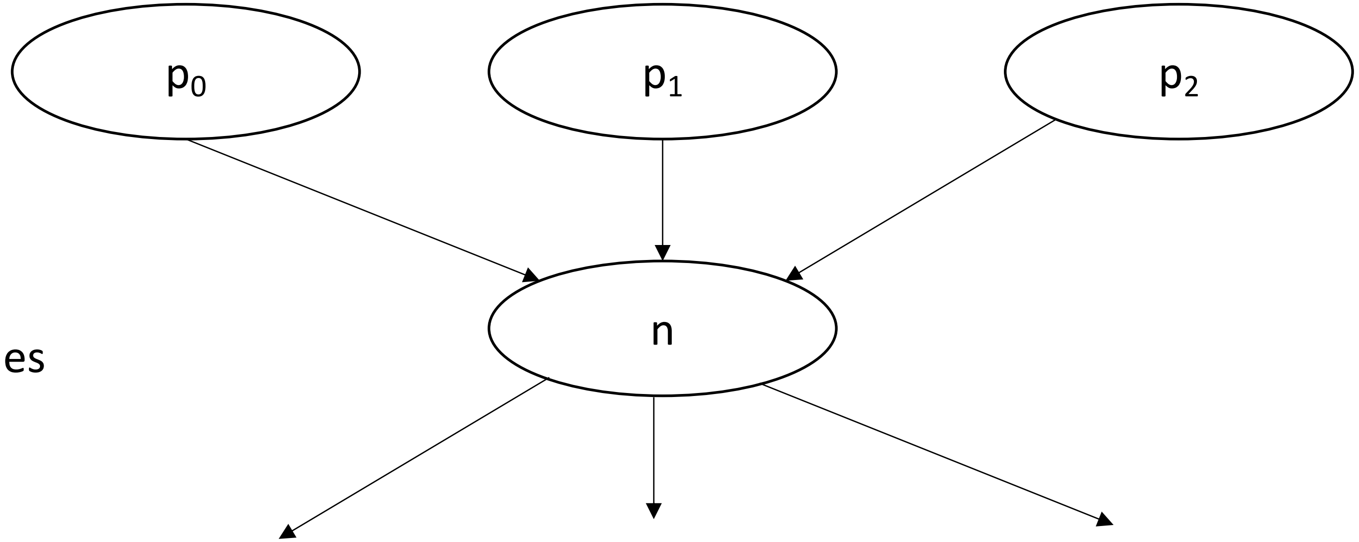


Update:
for all parents *p*: min(*p* + *d*)

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

Now lets think about dominance

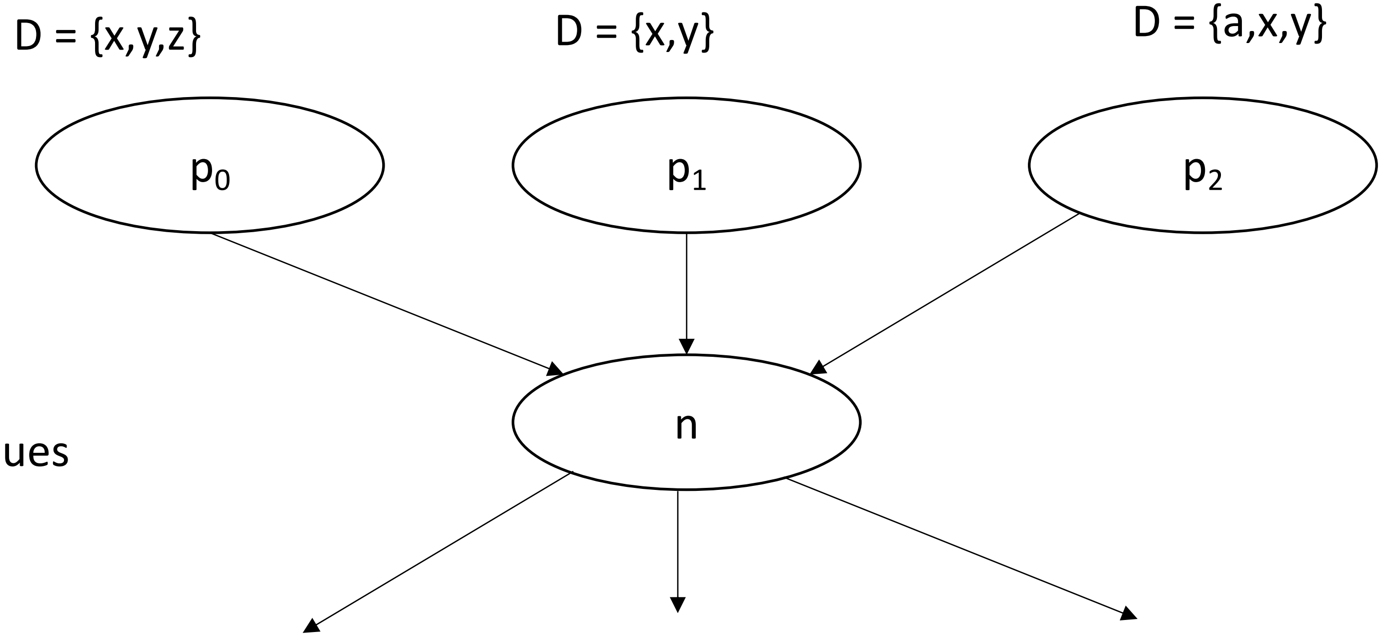
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

Now lets think about dominance

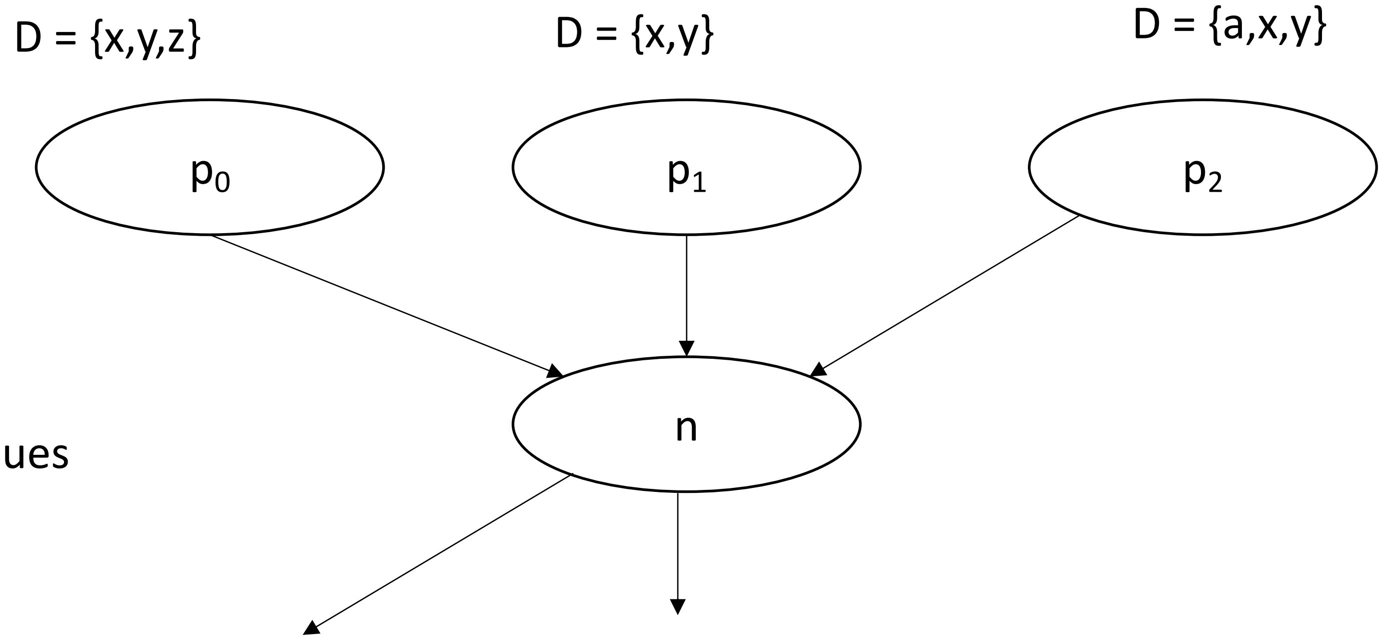
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



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

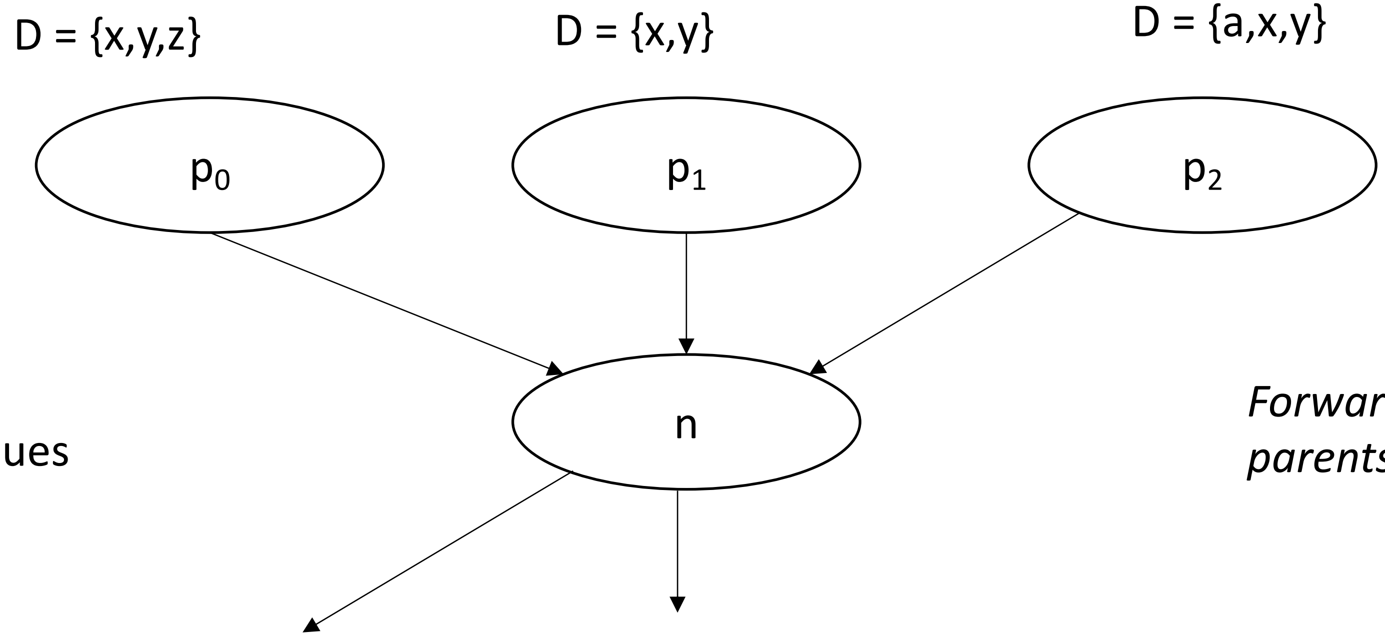


update:
intersection of parent values

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

Now lets think about dominance

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



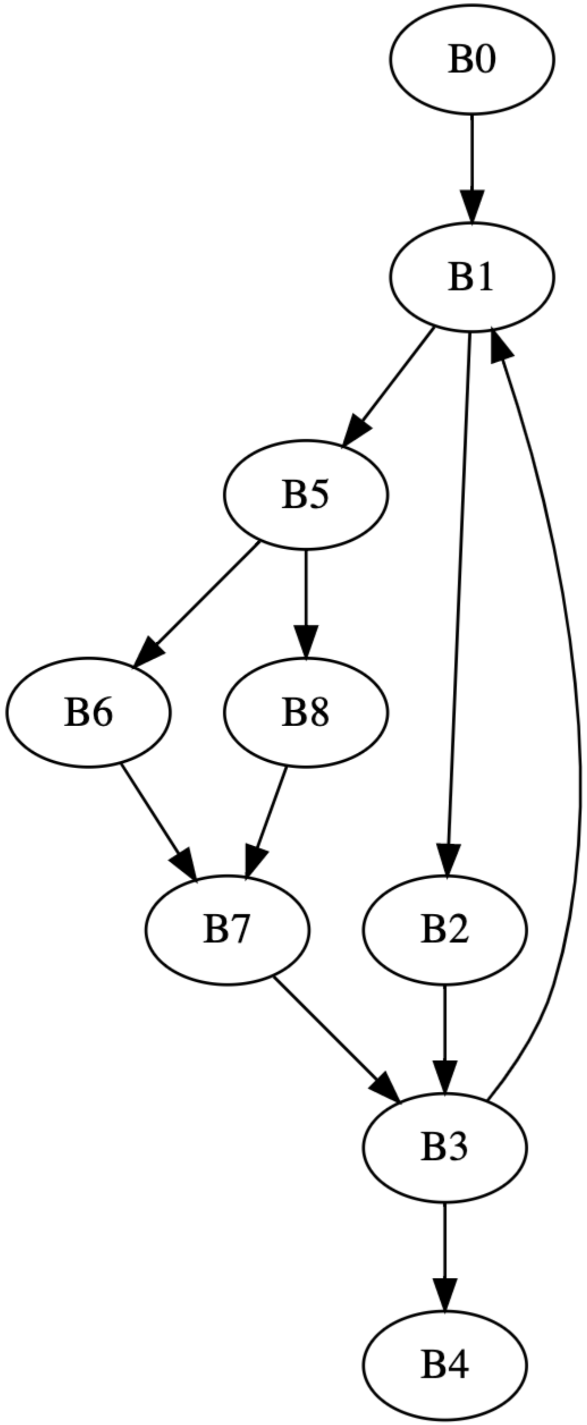
update:
intersection of parent values

Forward flow, as updates flow from parents to children.

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

How can we optimize the algorithm?

Node	Initial	I1	I2	I3
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	…	…

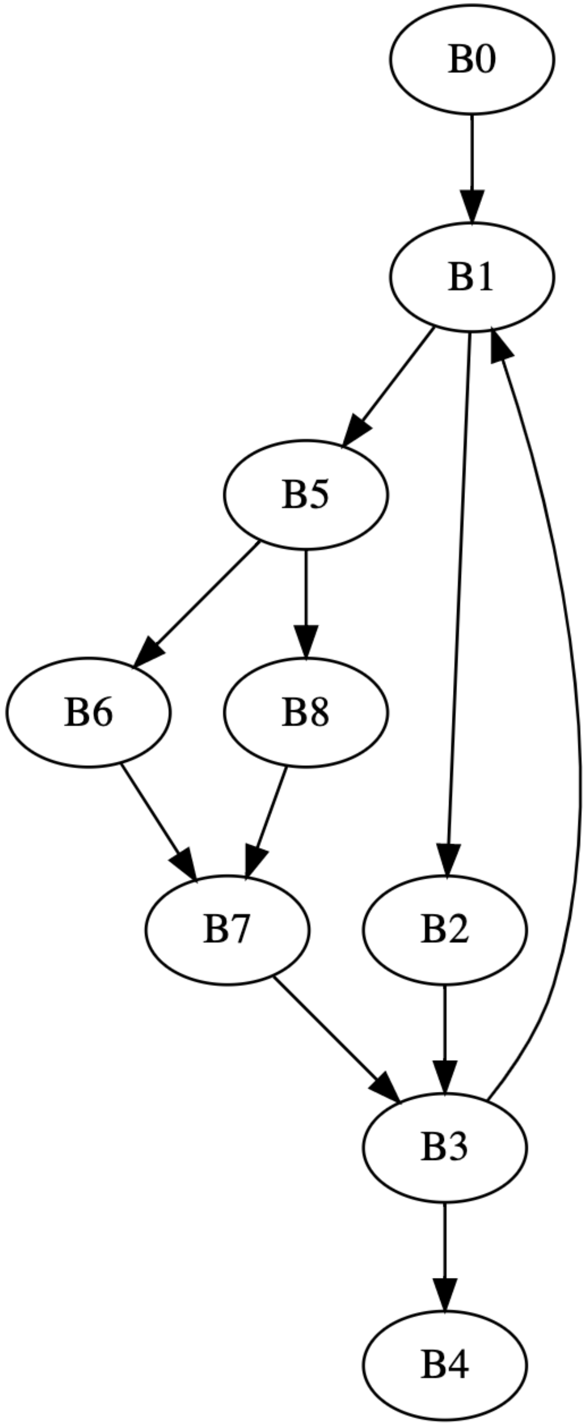


How can we optimize the algorithm?

Node	Initial	I1	I2	I3
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	…	…

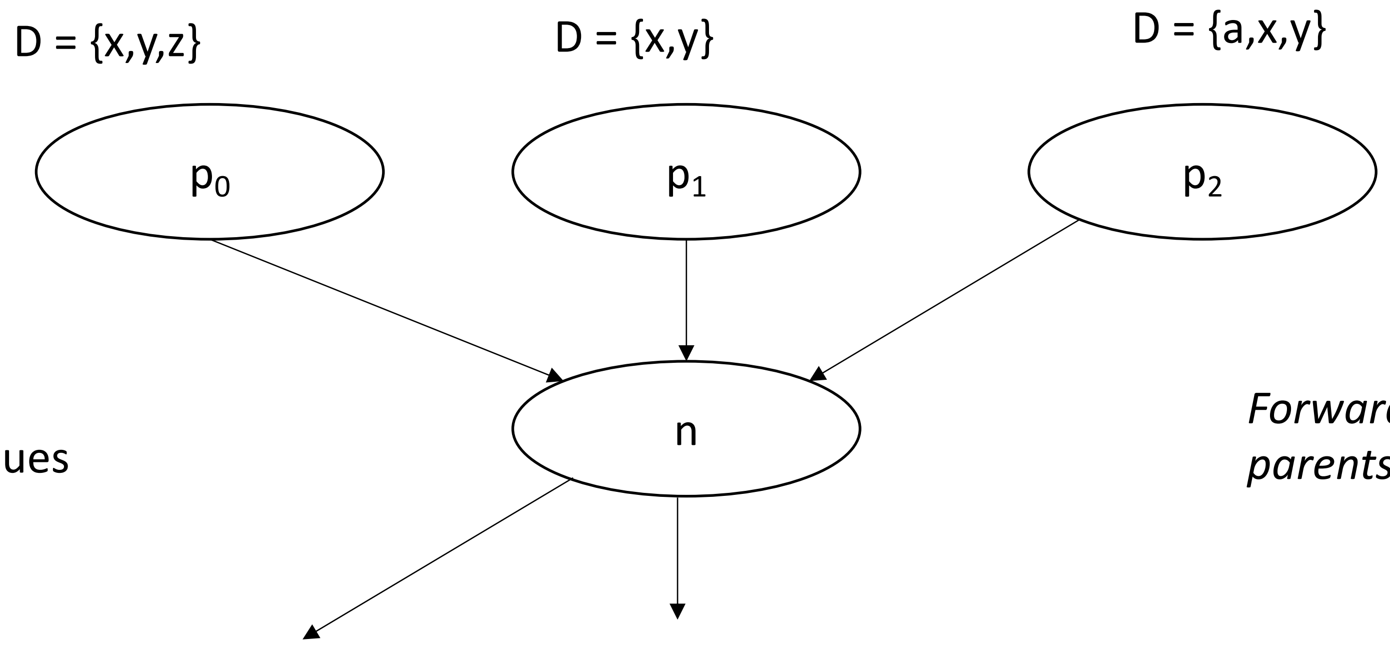
This can be any order...

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

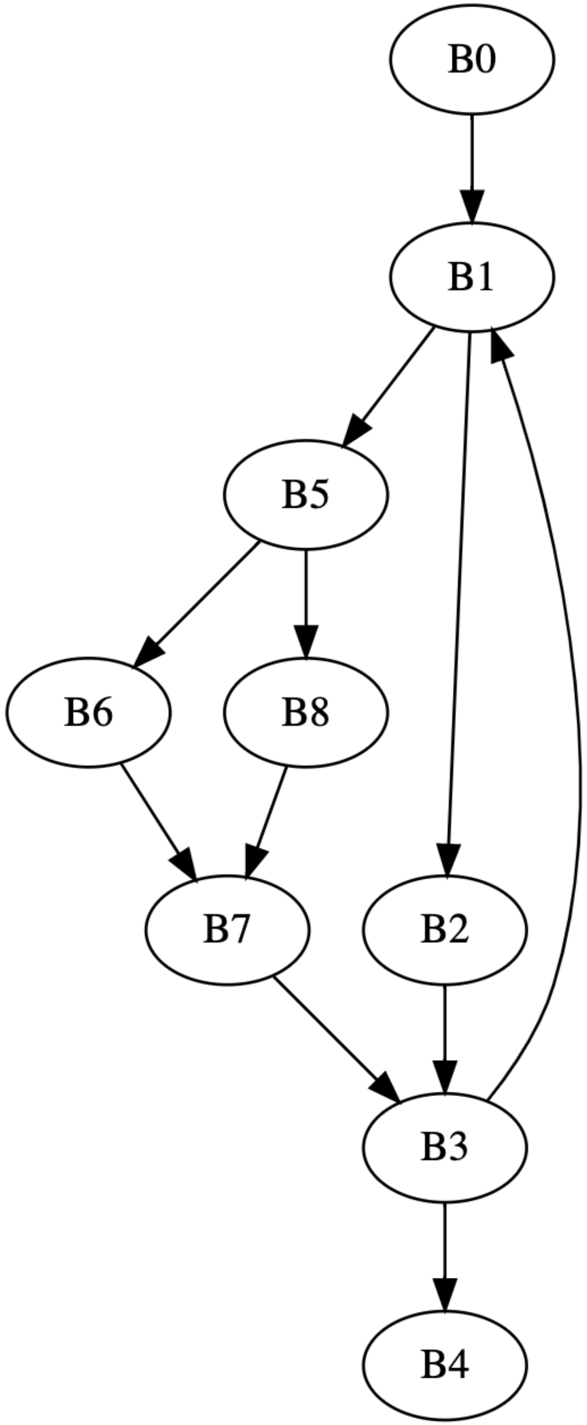
Forward flow, as updates flow from parents to children.

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

How can we optimize the algorithm?

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

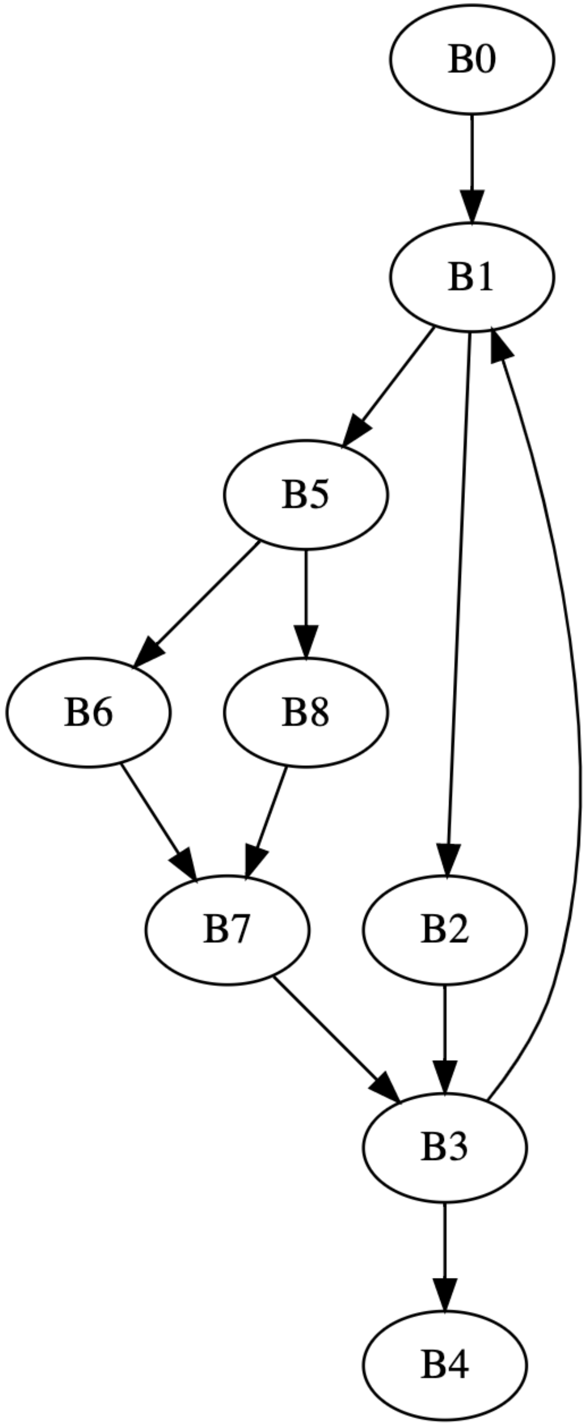
Reverse
post-order (rpo),
where parents are visited
first



How can we optimize the algorithm?

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

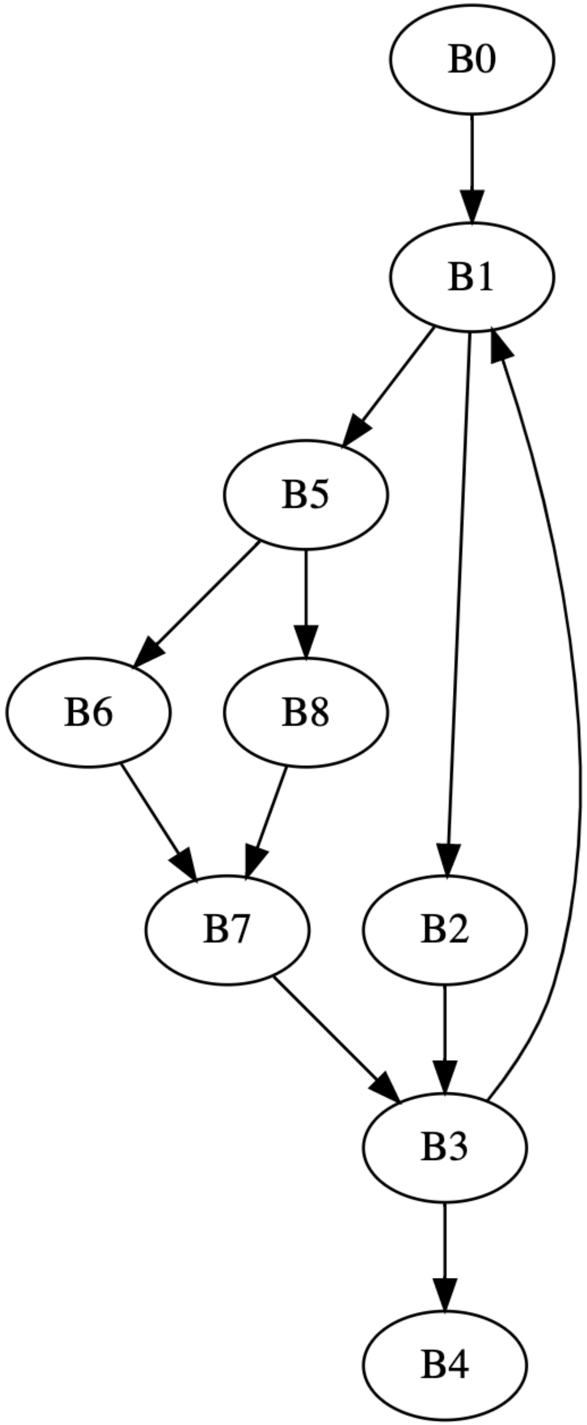
Reverse
post-order (rpo),
where parents are visited
first



How can we optimize the algorithm?

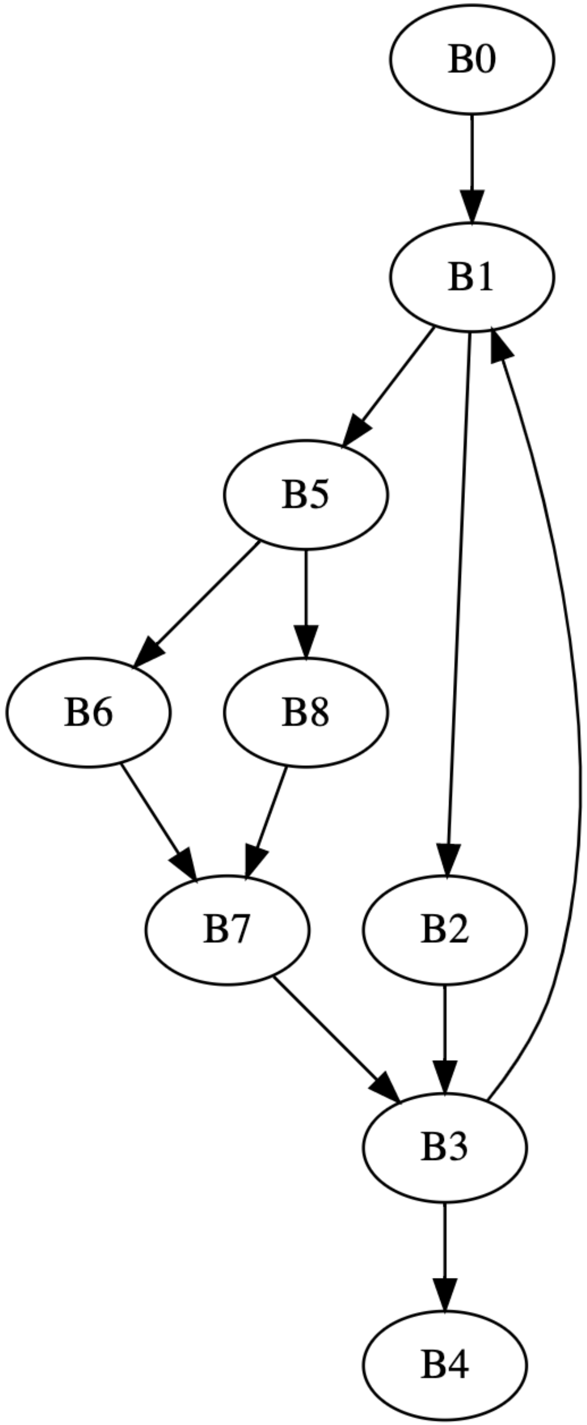
Node	New Order
B0	B0
B1	B1
B2	B2
B3	B5
B4	B6
B5	B8
B6	B7
B7	B3
B8	B4

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



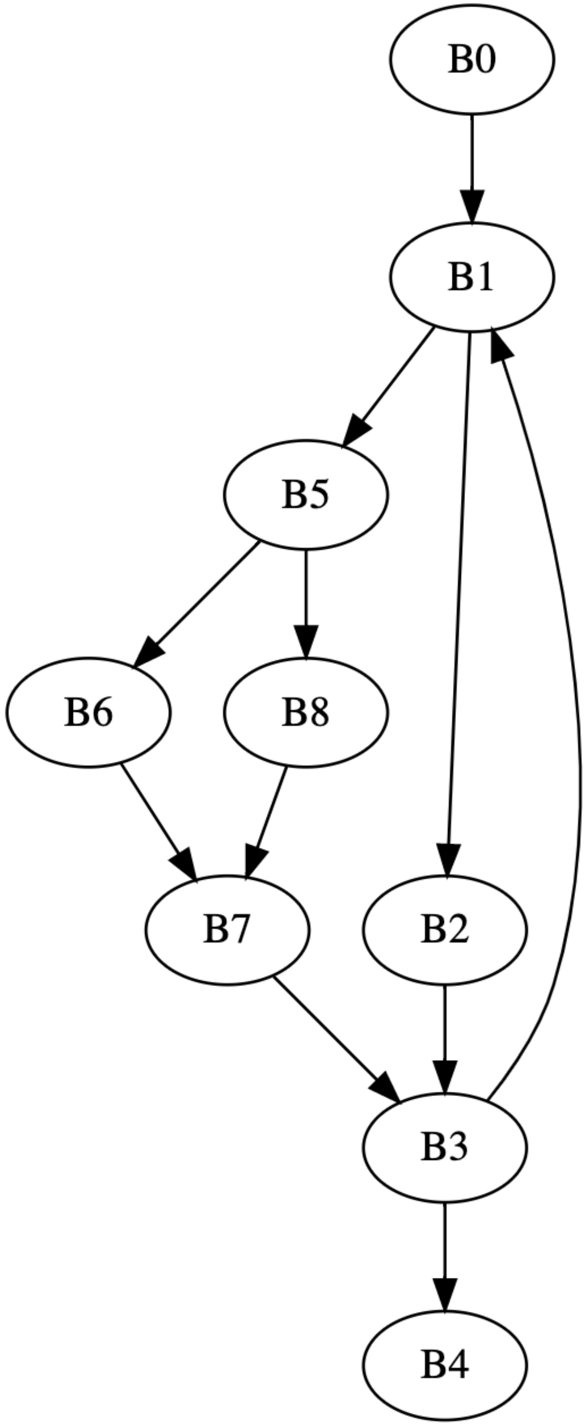
How can we optimize the algorithm?

Node	Initial	I1		
B0	B0			
B1	*N*			
B2	*N*			
B5	*N*			
B6	*N*			
B8	*N*			
B7	*N*			
B3	*N*			
B4	*N*			



How can we optimize the algorithm?

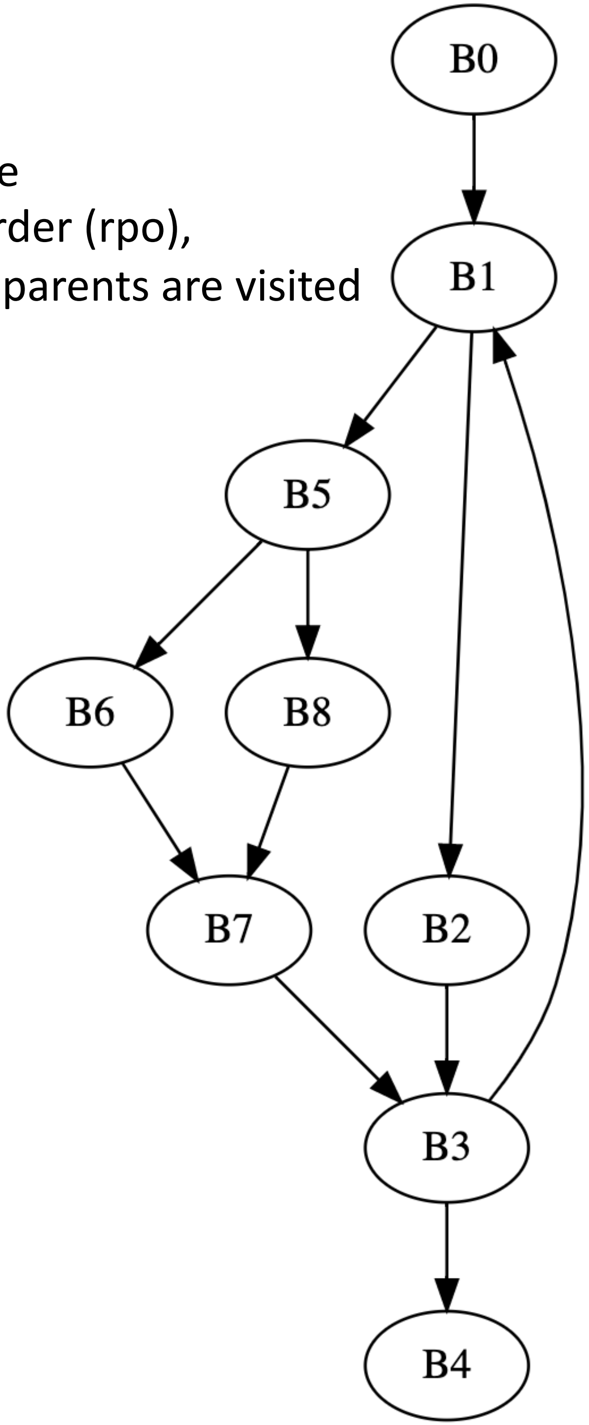
Node	Initial	I1		
B0	B0	B0		
B1	*N*	B0,B1		
B2	*N*	B0,B1,B2		
B5	*N*	B0,B1,B5		
B6	*N*	B0,B1,B5,B6		
B8	*N*	B0,B1,B5,B8		
B7	*N*	B0,B1,B5,B7		
B3	*N*	B0,B1,B3		
B4	*N*	B0,B1,B4		



How can we optimize the algorithm?

Node	Initial	I1	I2	
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		

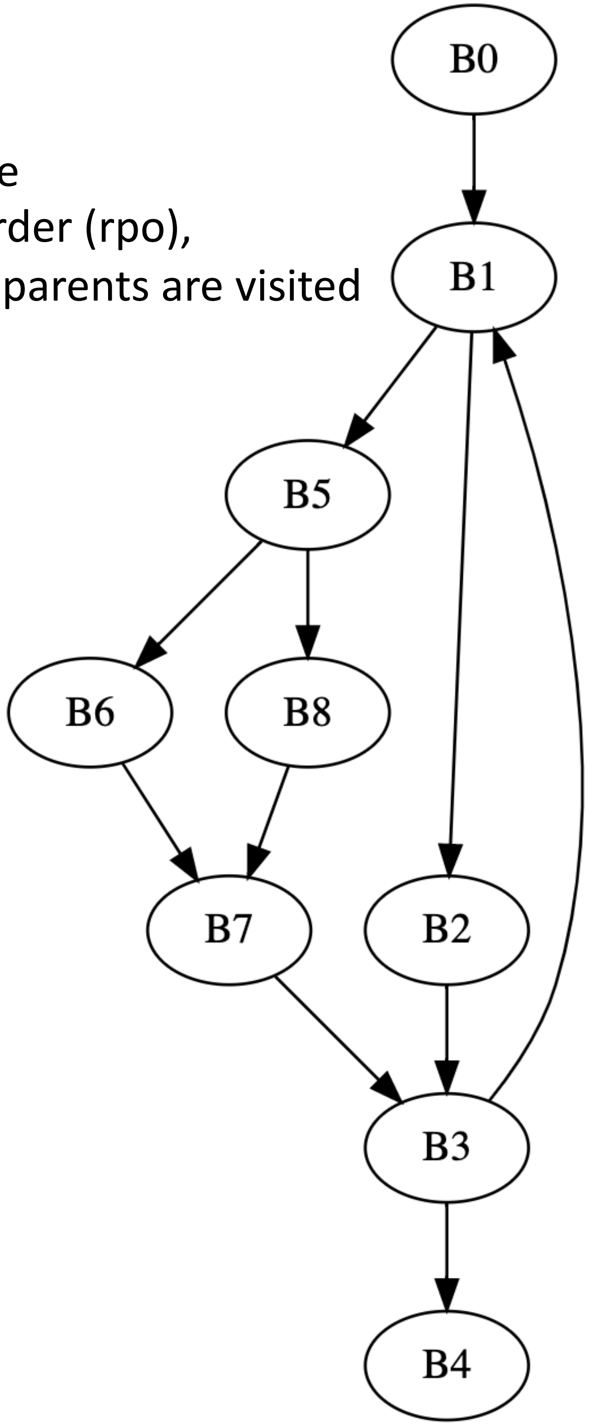
Reverse
post-order (rpo),
where parents are visited
first



How can we optimize the algorithm?

Node	Initial	I1	I2	
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	...	

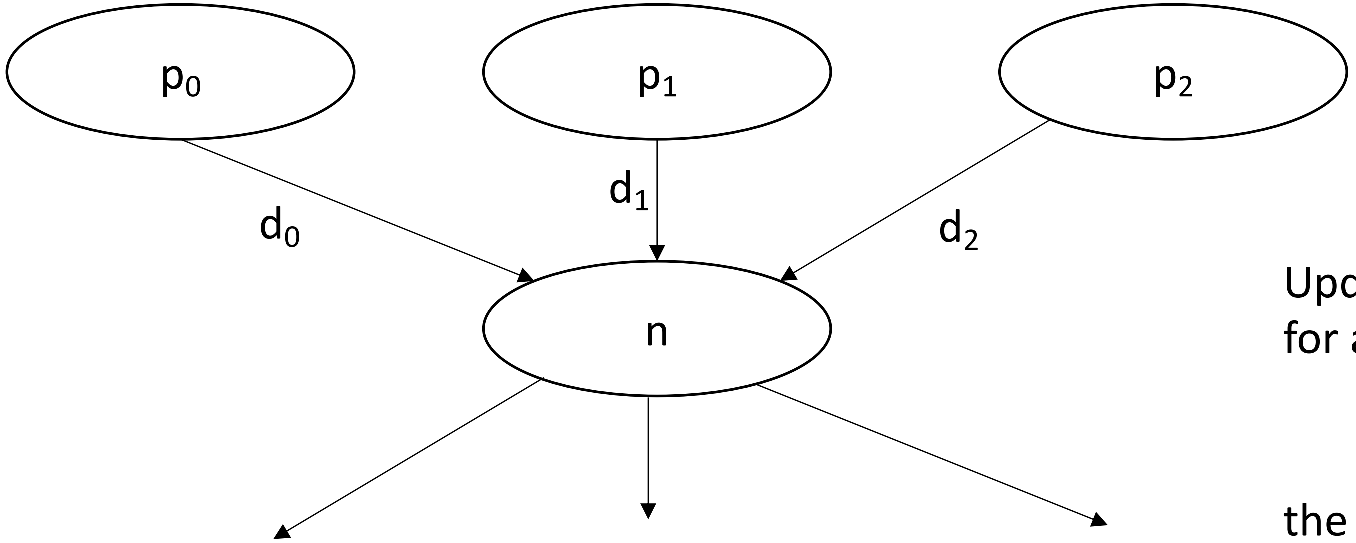
Reverse post-order (rpo), where parents are visited first



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:

```
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

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

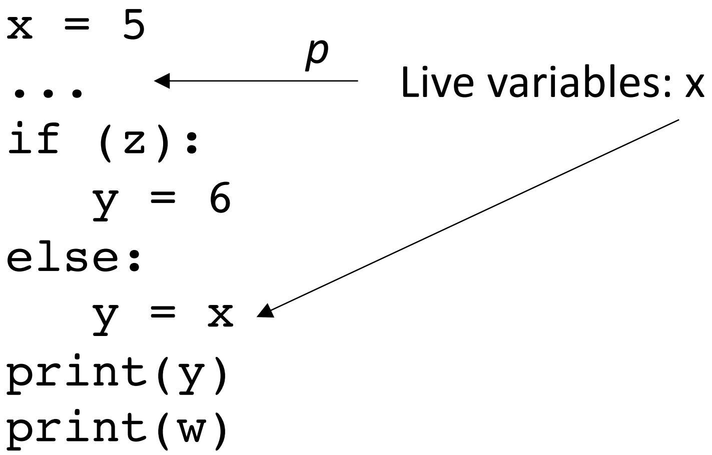
Live variables: x

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

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

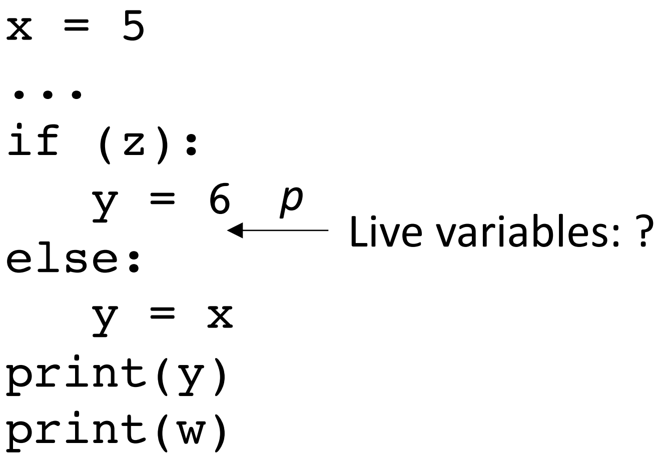
- examples:



```
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)
```

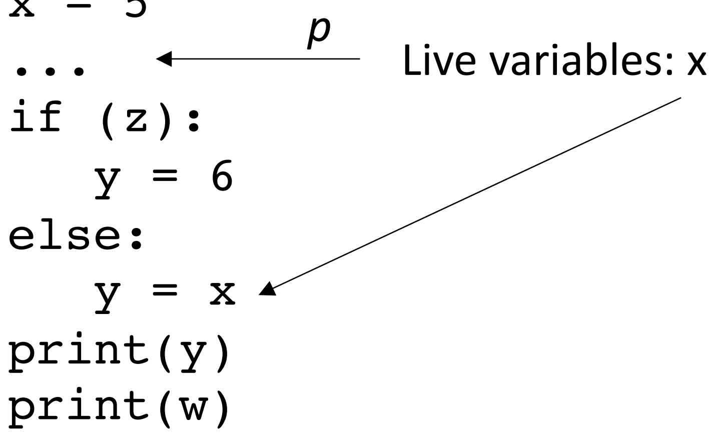
p

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

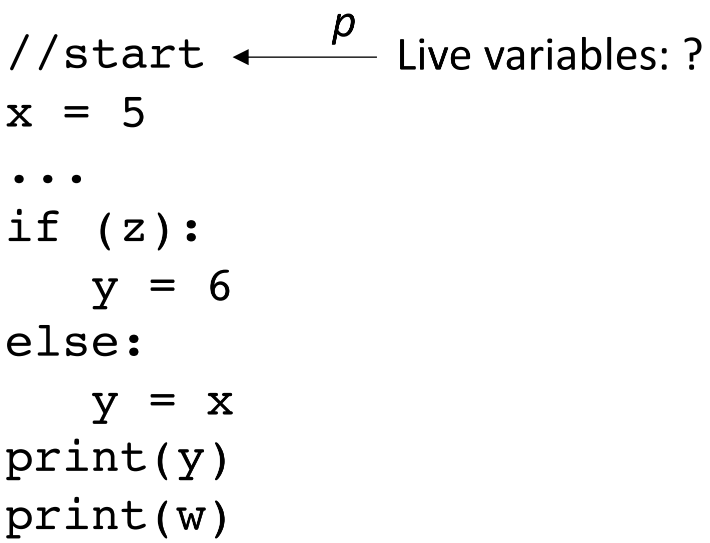
- examples:



```
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

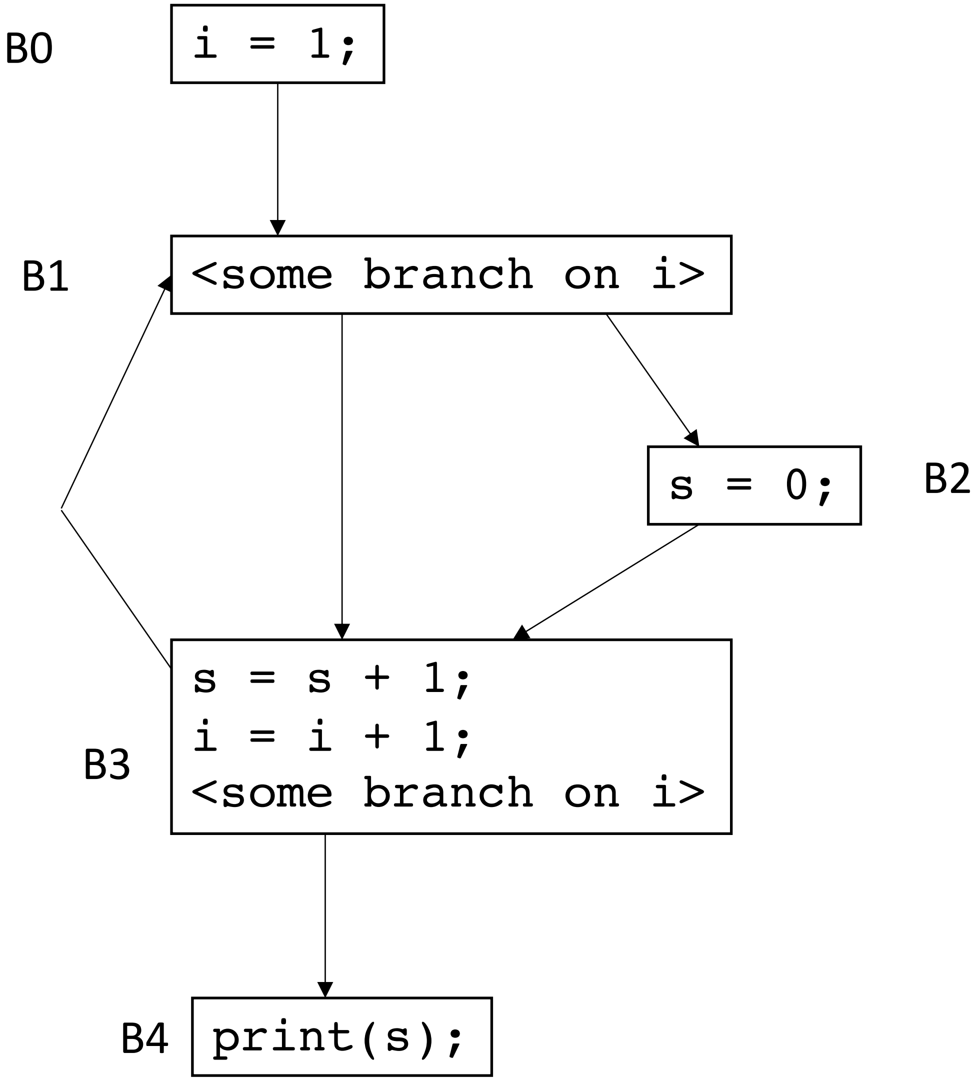
Accessing an uninitialized variable!

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

p

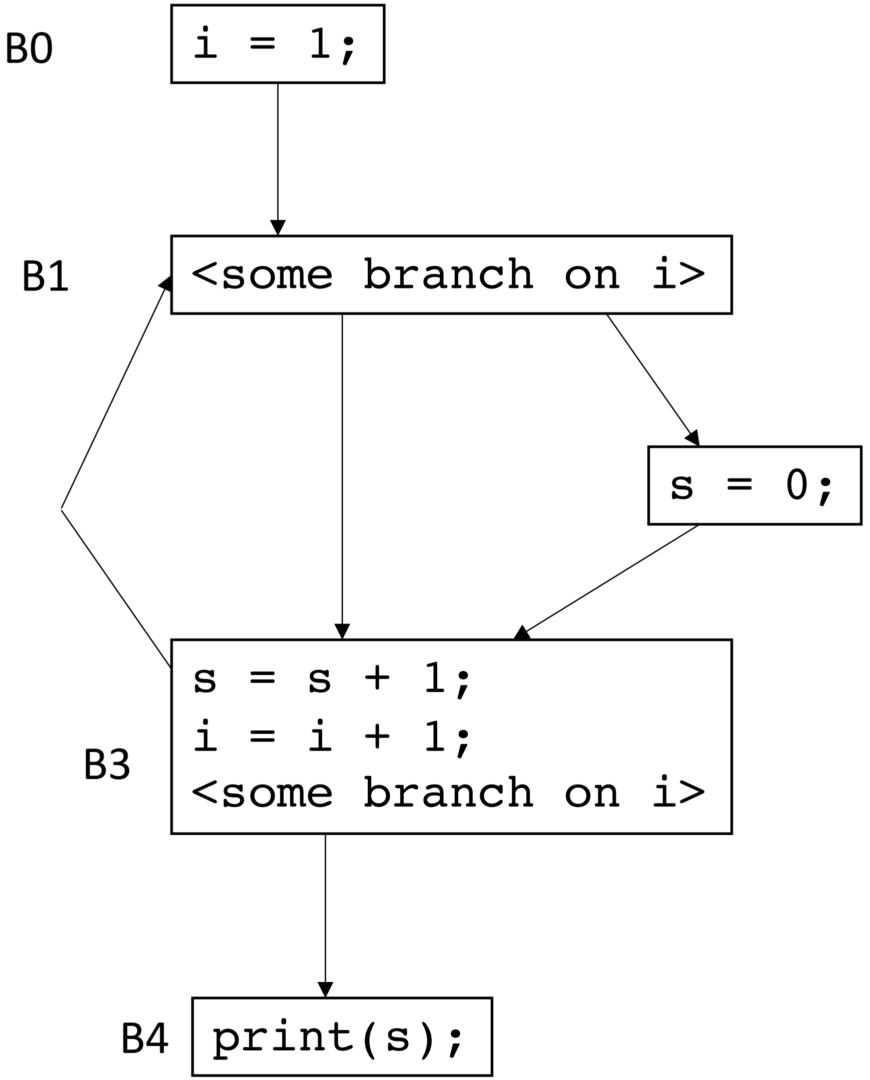
Live variables: w

Live variable analysis in the CFG:



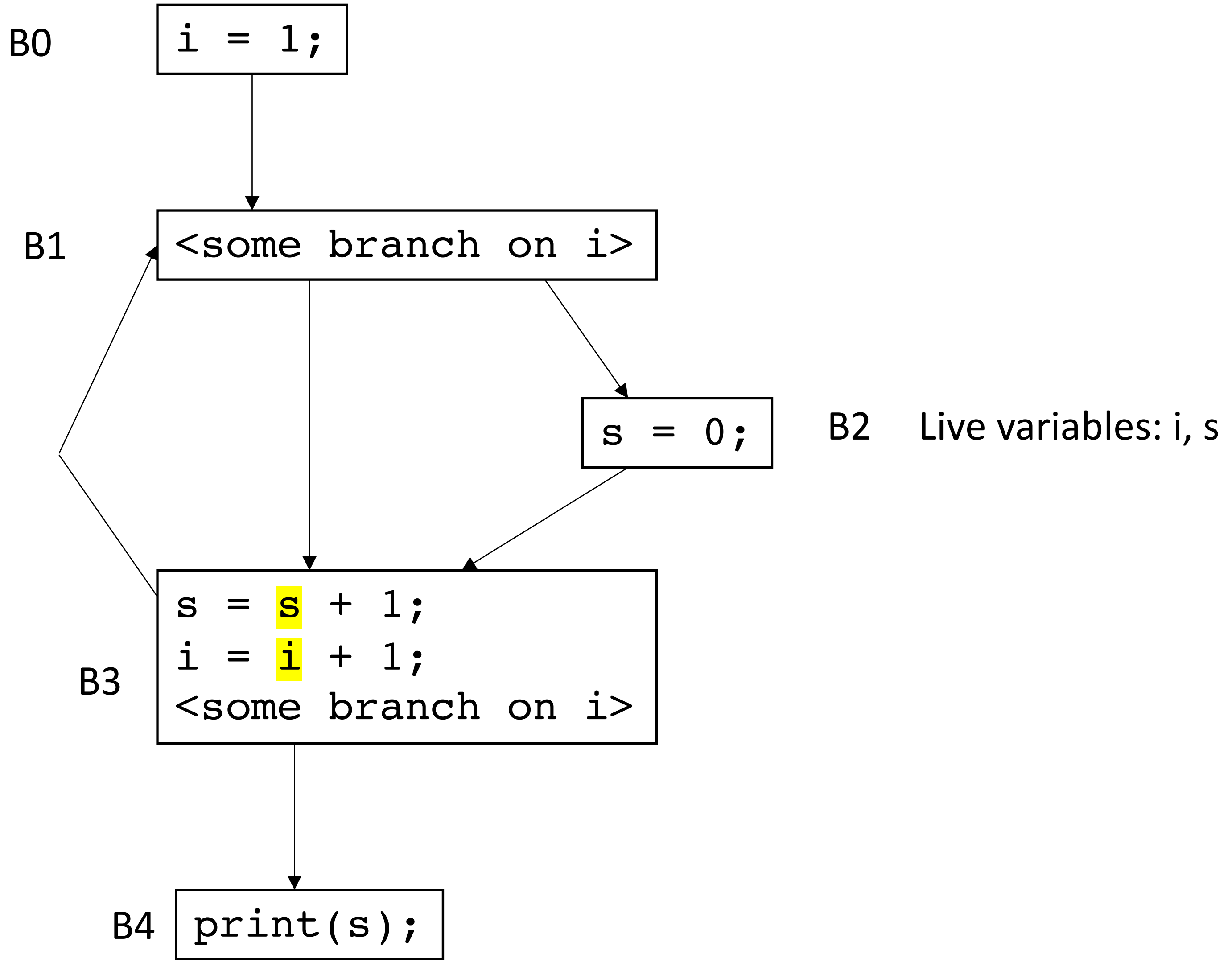
For each block B_x : we want to compute LiveOut:
The set of variables that are live at the end of B_x

Live variable analysis in the CFG:

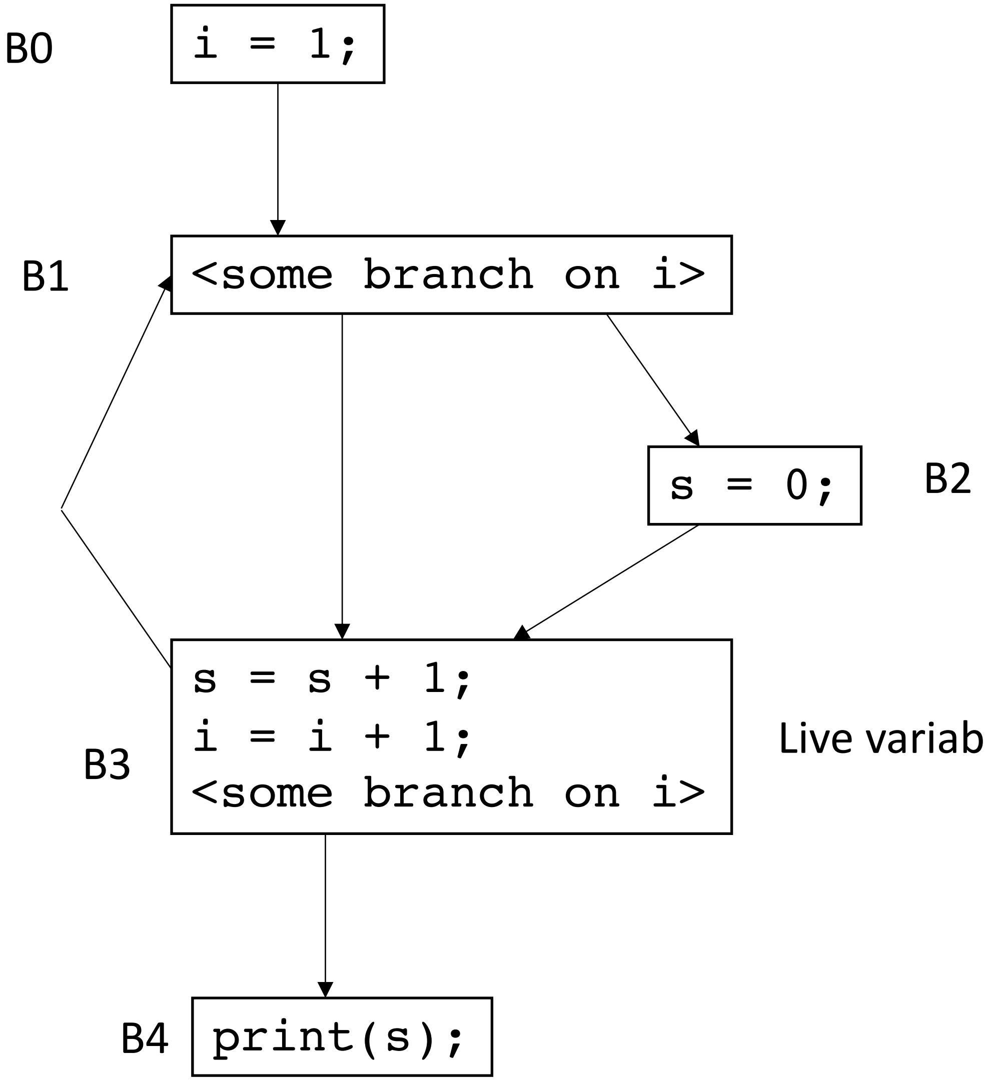


Live variables: i, s

Live variable analysis in the CFG:



Live variable analysis in the CFG:



```
i = 1;
```

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

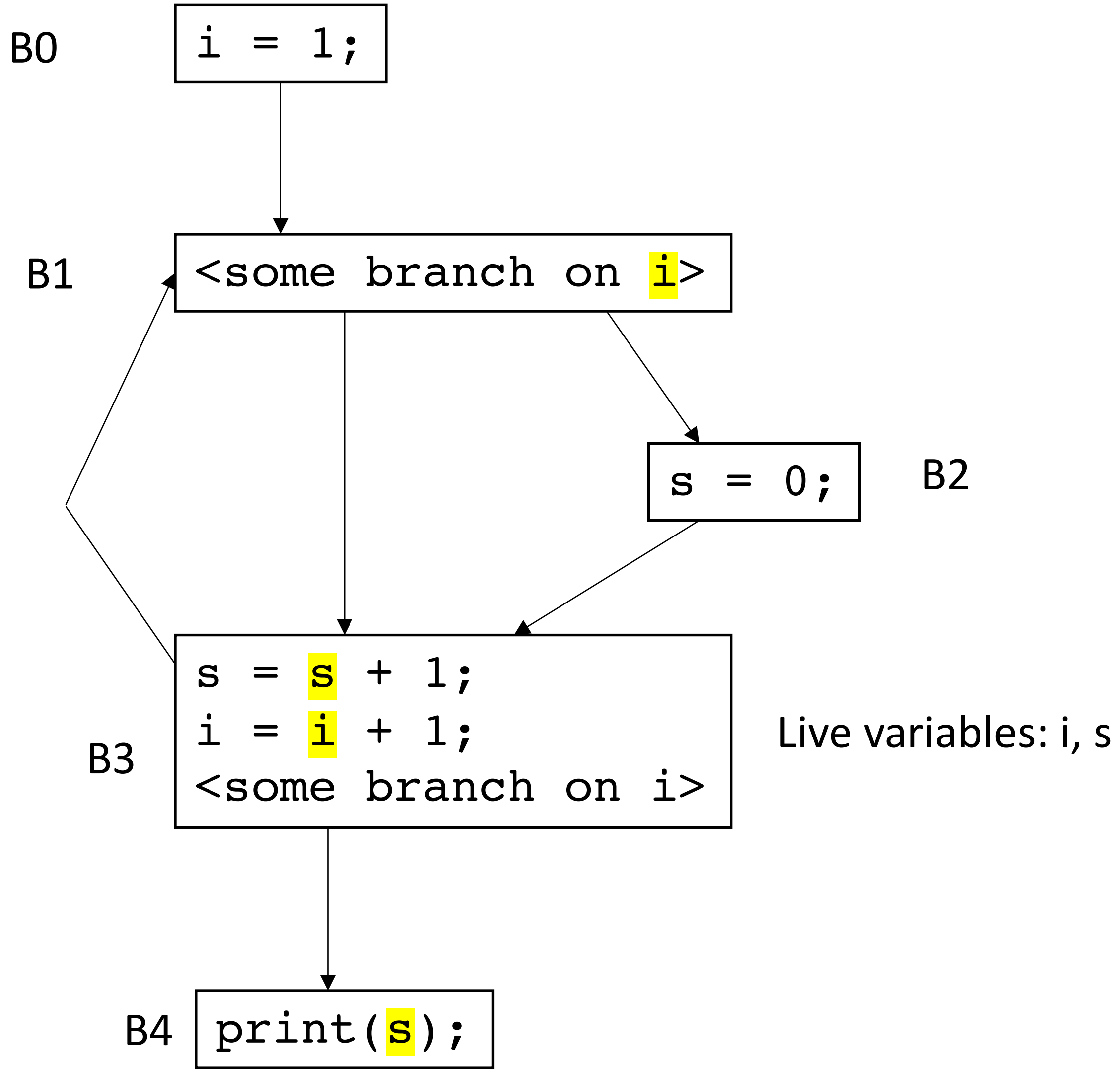
```
s = 0;
```

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

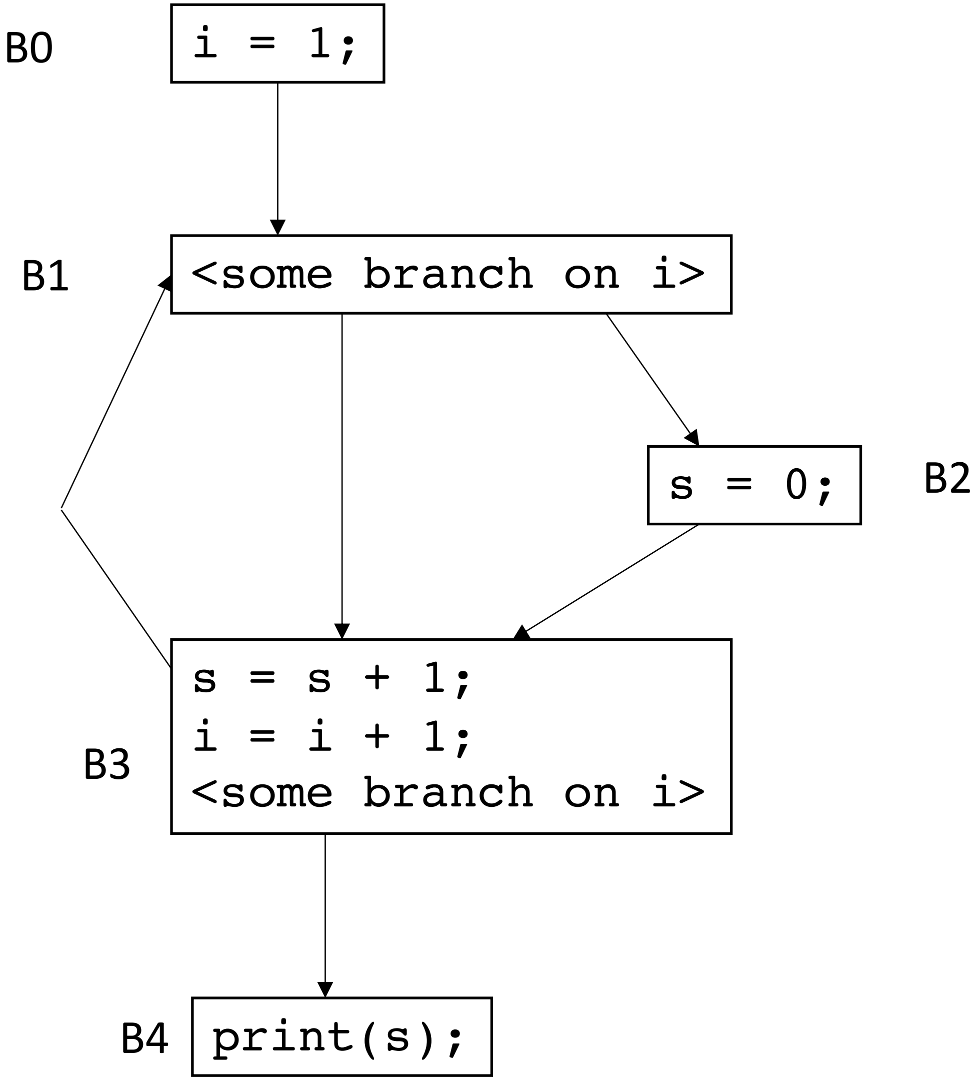
```
print(s);
```

Live variables: i, s

Live variable analysis in the CFG:



Live variable analysis in the CFG:



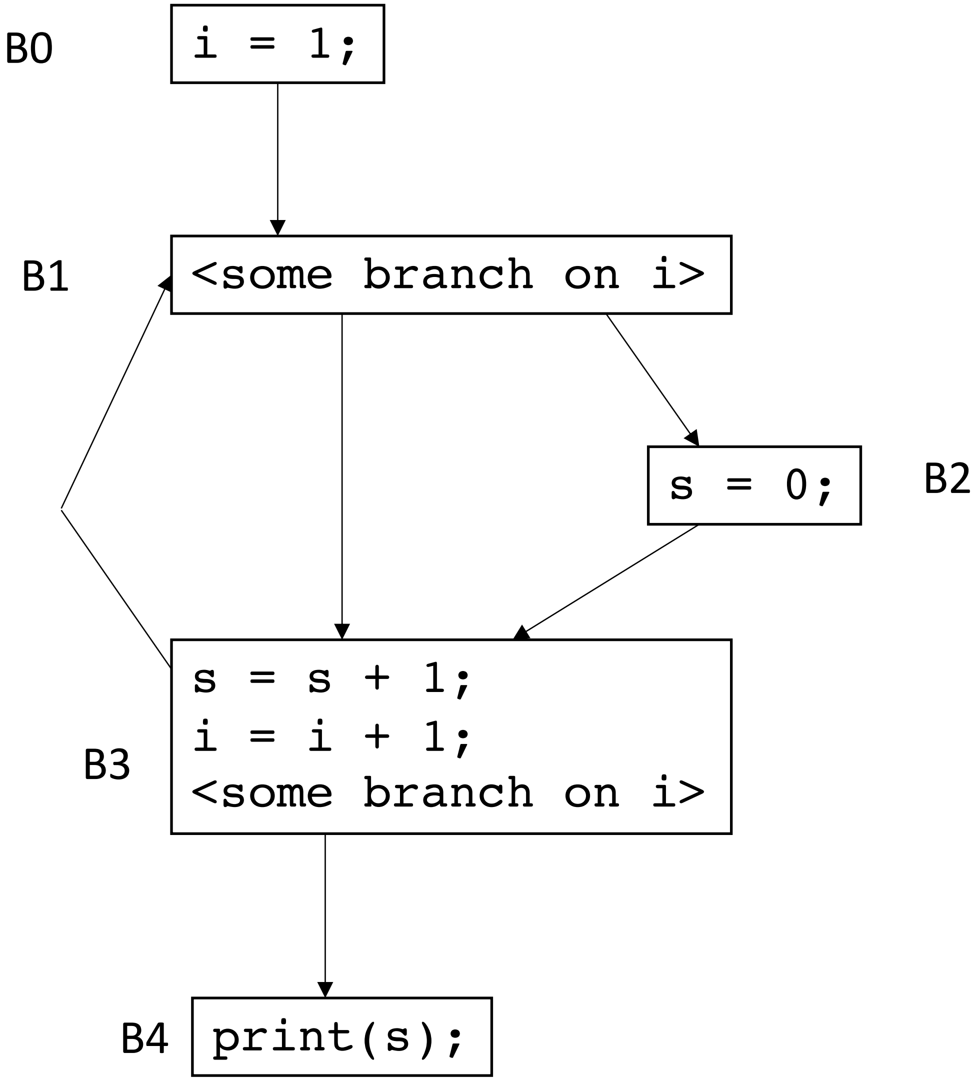
To compute the LiveOut sets, we need two initial sets:

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 read before being overwritten

Block	VarKill	UEVar
B0		
B1		
B2		
B3		
B4		

Live variable analysis in the CFG:



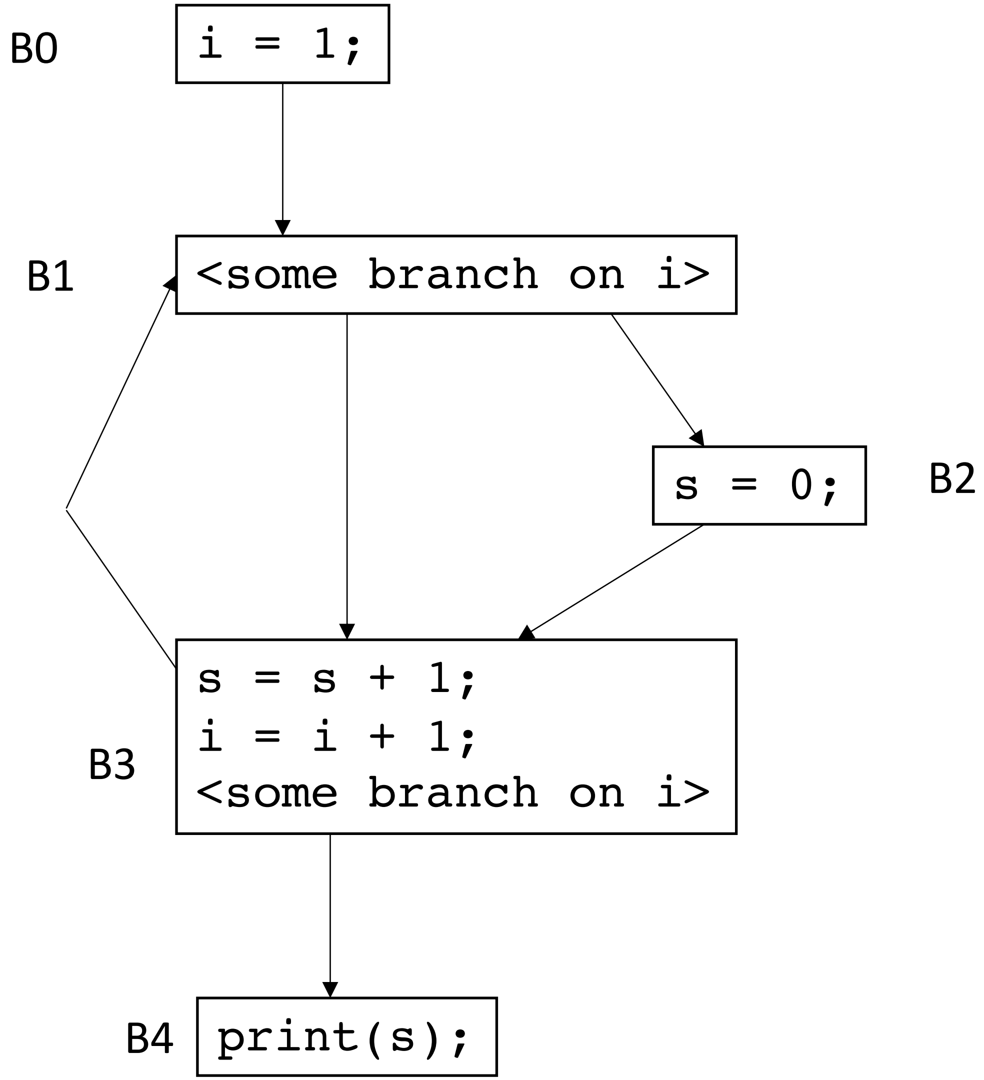
To compute the LiveOut sets, we need two initial sets:

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 read before being overwritten

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 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:

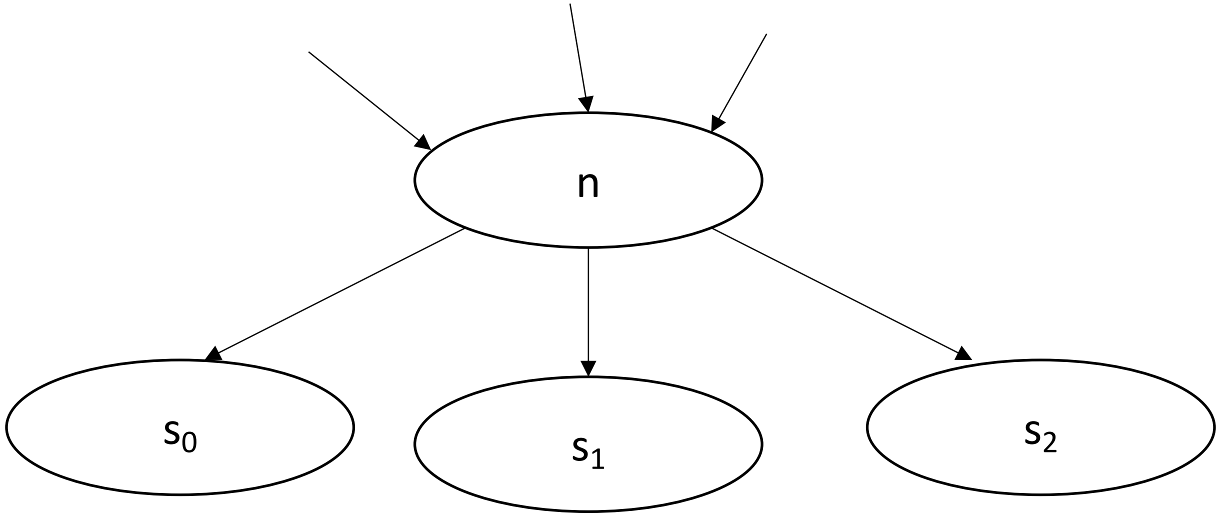
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed point computation:

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

Live variable analysis in the CFG:

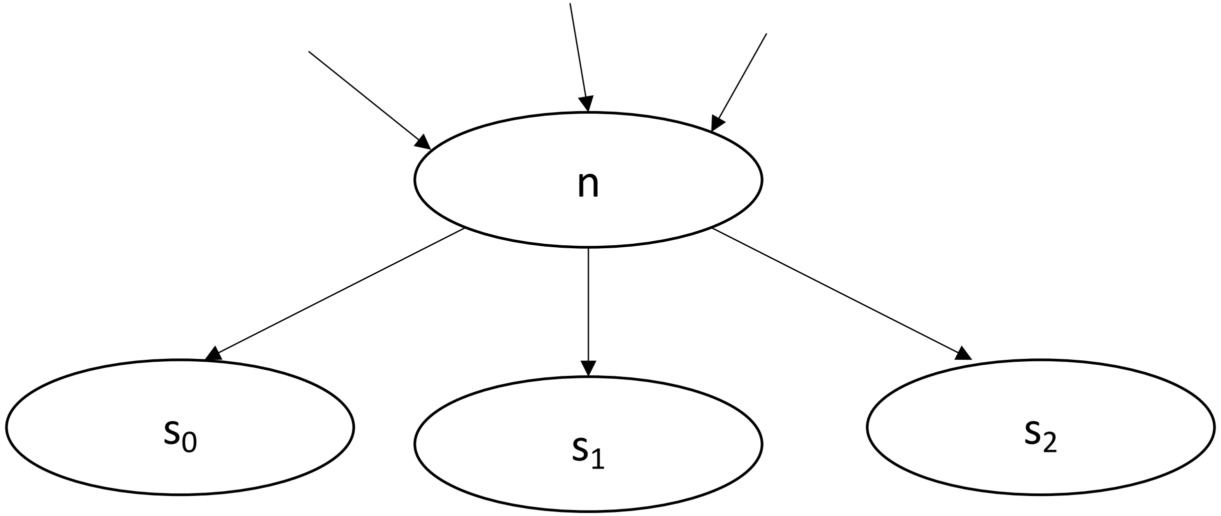
$$\mathit{LiveOut}(n) = \cup_{s \text{ in } succ(n)} \left(\mathit{UEVar}(s) \cup (\text{LiveOut}(s) \cap \overline{\mathit{VarKill}(s)} \right))$$



Backwards flow analysis because values flow from successors

Live variable analysis in the CFG:

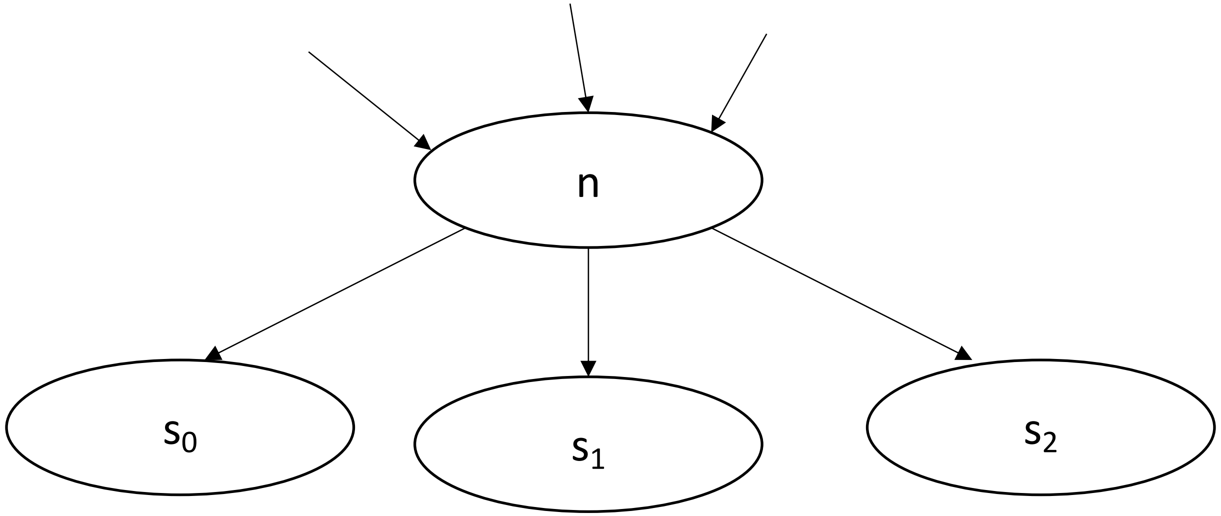
$$\mathit{LiveOut(n)} = \cup_{\text{s in succ(n)}} \left(\mathit{UEVar(s)} \cup (\text{LiveOut(s)} \cap \overline{\mathit{VarKill(s)}}\,) \right)$$



any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

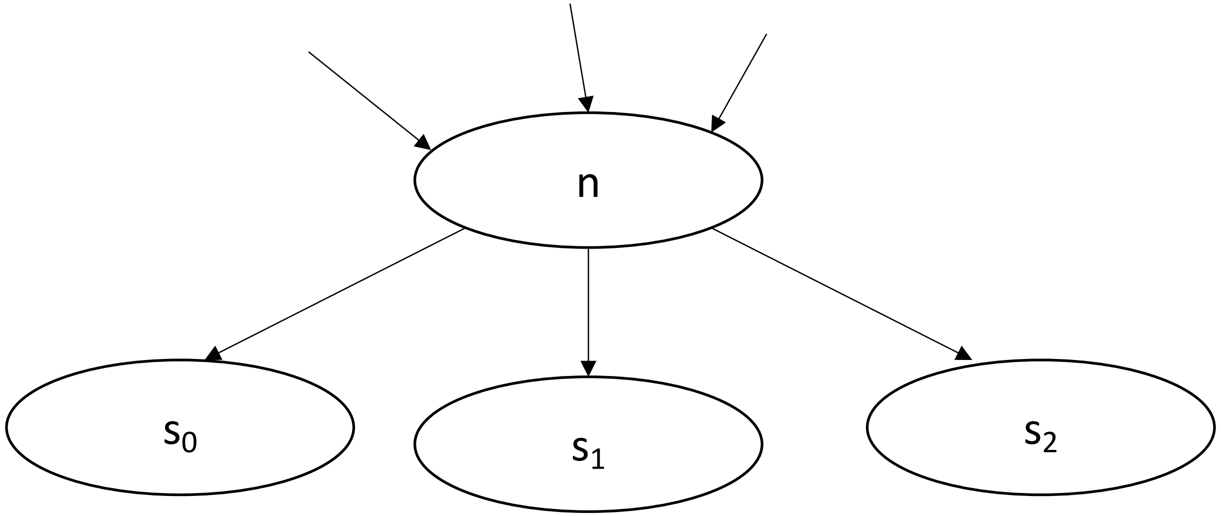
$$\mathit{LiveOut(n)} = \cup_{\text{s in succ(n)}} \; (\; \mathit{UEVar(s)} \cup (\text{LiveOut(s)} \cap \overline{\mathit{VarKill(s)}}\;))$$



variables that are not overwritten in s

Live variable analysis in the CFG:

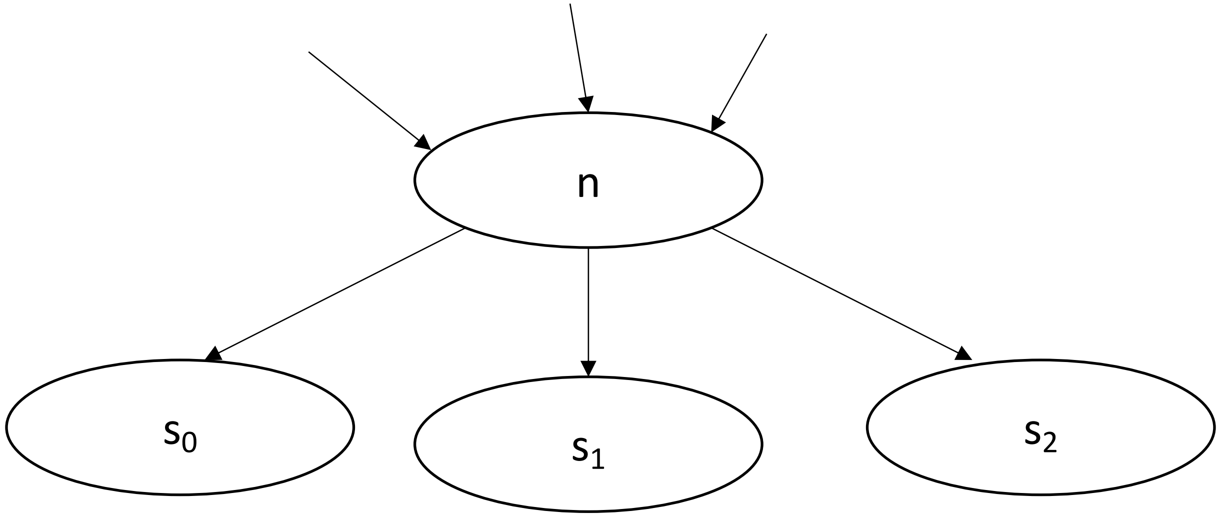
$$\mathit{LiveOut(n)} = \cup_{\text{s in succ(n)}} \left(\mathit{UEVar(s)} \cup \left(\text{LiveOut(s)} \cap \overline{\mathit{VarKill(s)}}\,\right)\right)$$



variables that are live
at the end of s

Live variable analysis in the CFG:

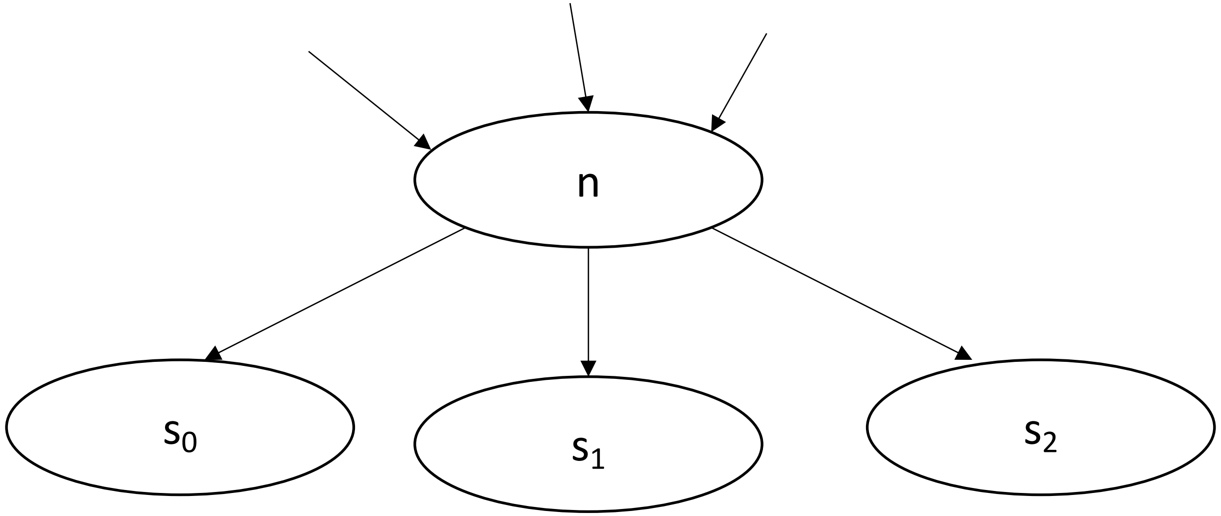
$$\mathit{LiveOut(n)} = \cup_{\text{s in succ(n)}} \; (\; \mathit{UEVar(s)} \cup (\text{LiveOut(s)} \cap \overline{\mathit{VarKill(s)}}\;))$$



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

Live variable analysis in the CFG:

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



LiveOut is a union
rather than an intersection

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

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

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

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

Consider the language we use for each:

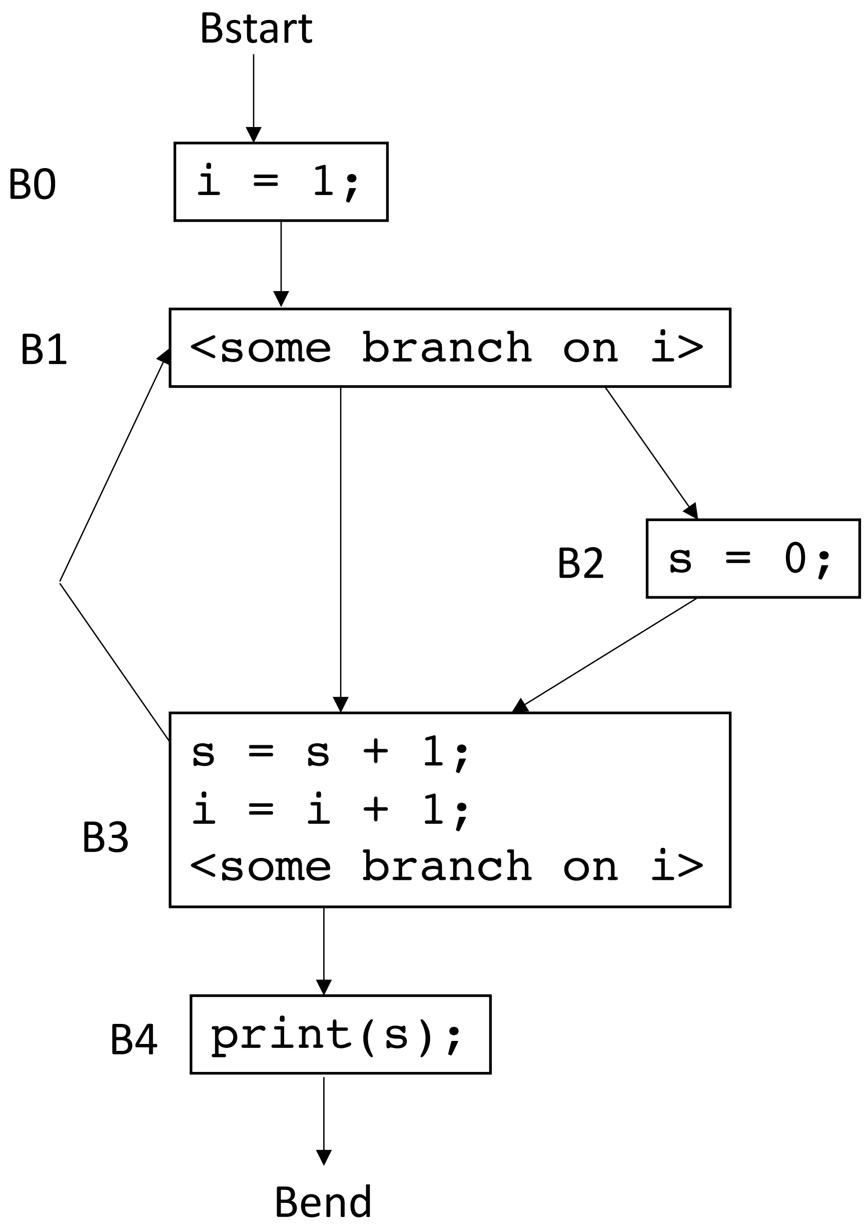
- **Dominance** of node b_x contains b_y if:
  - **every** path from the start to b_x goes through b_y

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

- *Some vs. Every*

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

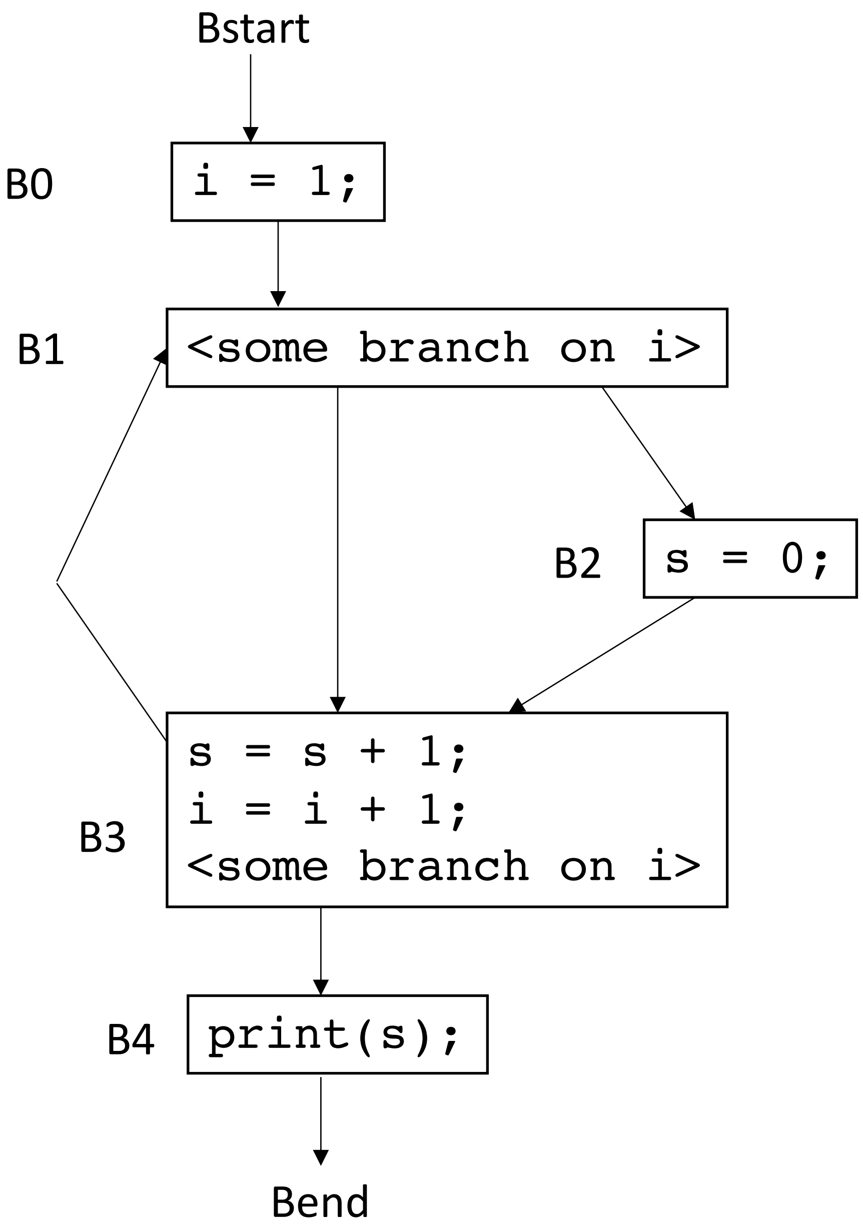
$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in preds(n)}} Dom(p) \right)$$



Now we can perform the iterative fixed point computation:

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

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



Bstart

B0 `i = 1;`

B1 `<some branch on i>`

B2 `s = 0;`

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

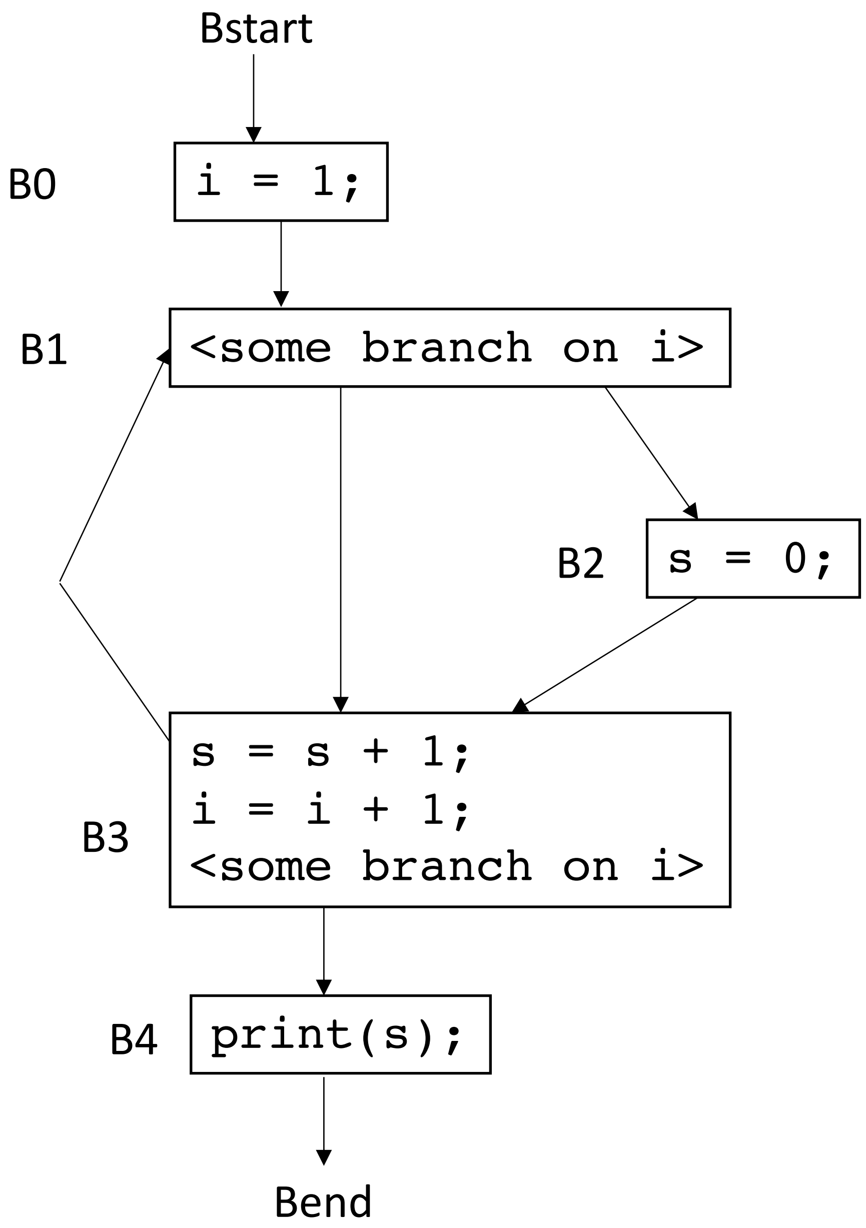
B4 `print(s);`

Bend

Now we can perform the iterative fixed point computation:

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

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



Bstart

B0
```
i = 1;
```

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

B2
```
s = 0;
```

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

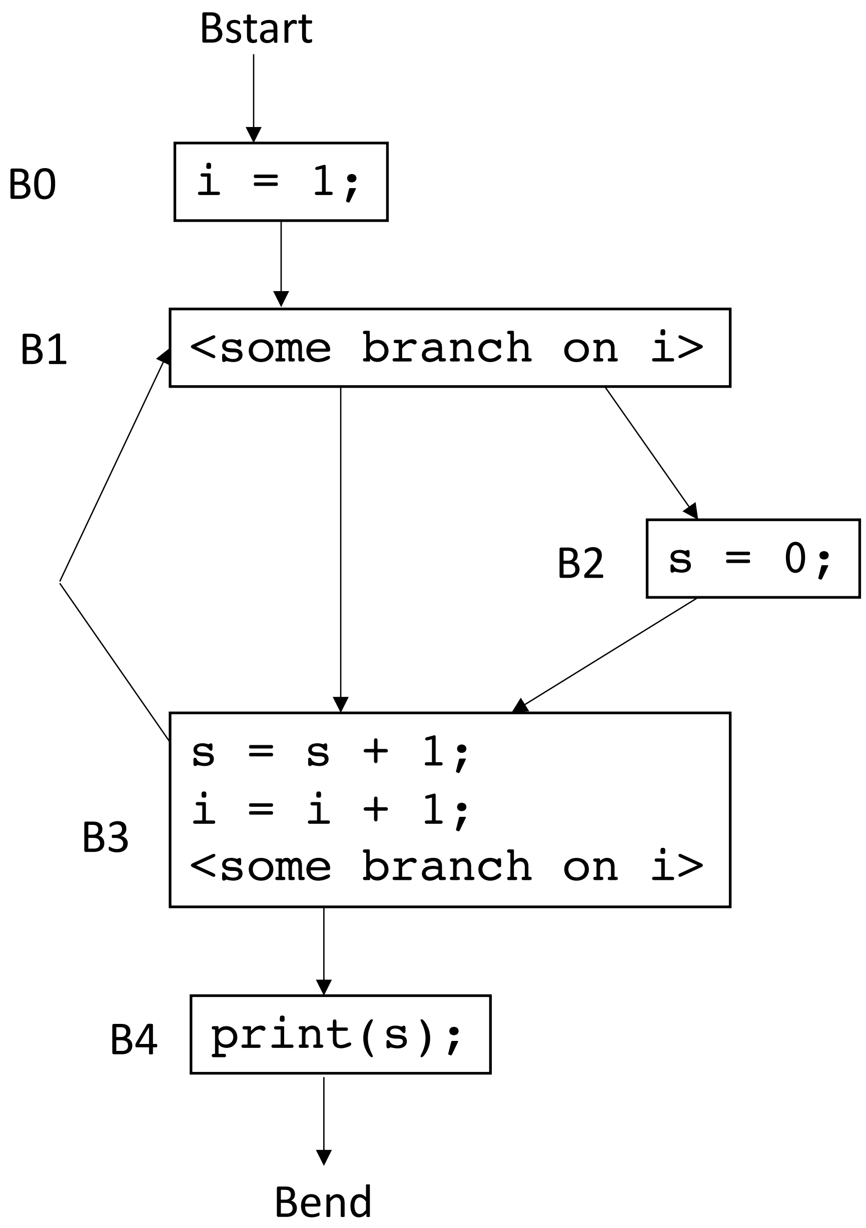
B4
```
print(s);
```

Bend

Now we can perform the iterative fixed point computation:

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

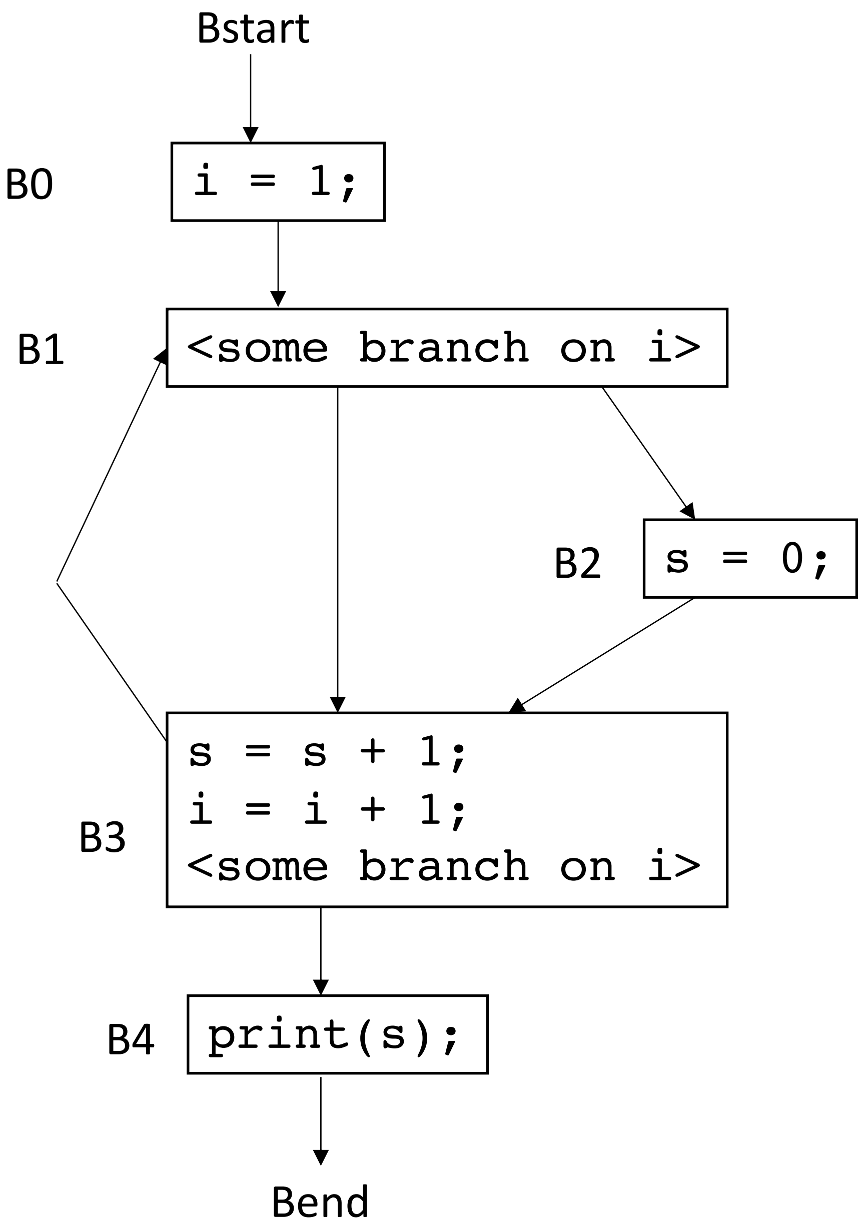
Block	VarKill	UEVar	LiveOut l_0	LiveOut l_1
Bstart	{}	{}	{}	{}
B0	i	{}	{}	i
B1	{}	i	{}	s,i
B2	s	{}	{}	s,i
B3	s,i	s,i	{}	s,i
B4	{}	s	{}	{}
Bend	{}	{}	{}	{}



Now we can perform the iterative fixed point computation:

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

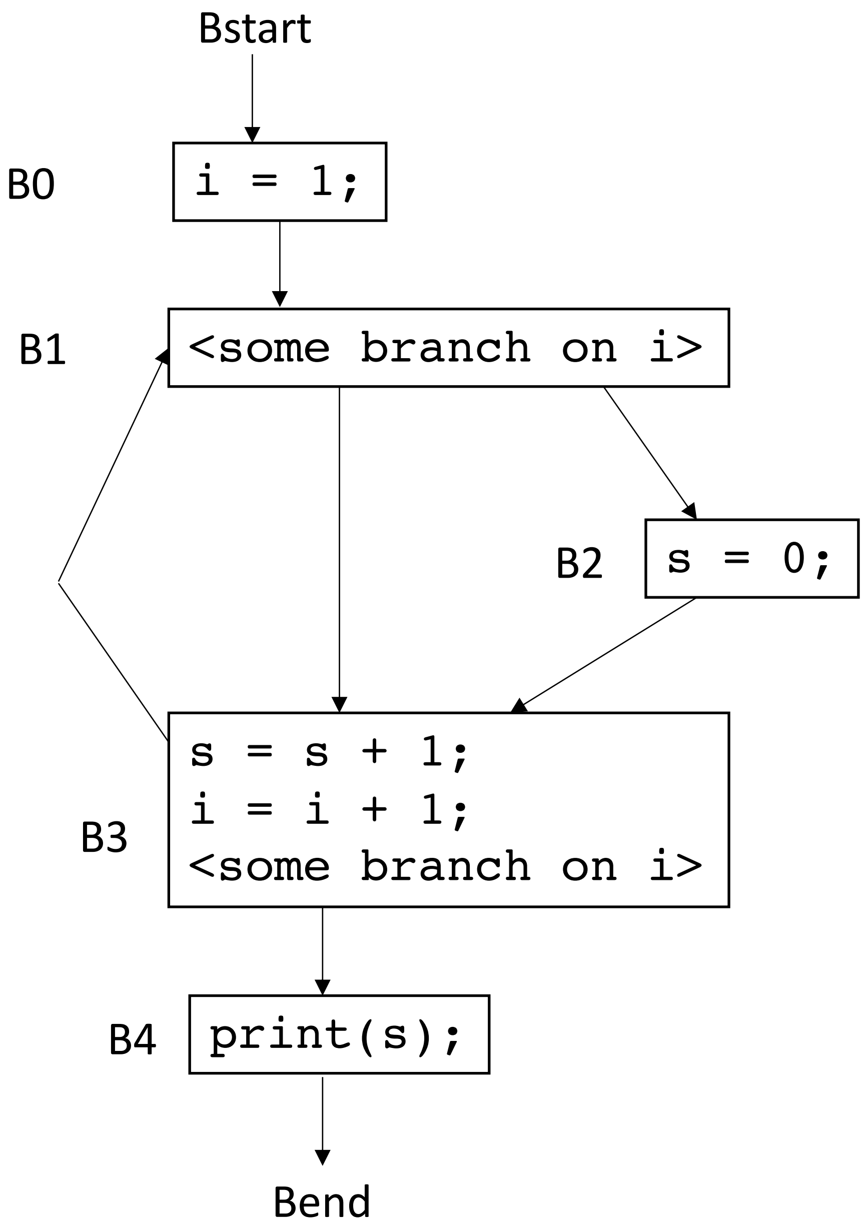
Block	VarKill	UEVar	LiveOut l_0	LiveOut l_1	LiveOut l_2
Bstart	{}	{}	{}	{}	
B0	i	{}	{}	i	
B1	{}	i	{}	s,i	
B2	s	{}	{}	s,i	
B3	s,i	s,i	{}	s,i	
B4	{}	s	{}	{}	
Bend	{}	{}	{}	{}	



Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	LiveOut l_0	LiveOut l_1	LiveOut l_2
Bstart	{}	{}	{}	{}	{}
B0	i	{}	{}	i	s,i
B1	{}	i	{}	s,i	s,i
B2	s	{}	{}	s,i	s,i
B3	s,i	s,i	{}	s,i	s,i
B4	{}	s	{}	{}	{}
Bend	{}	{}	{}	{}	{}



Bstart

B0
```
i = 1;
```

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

B2
```
s = 0;
```

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

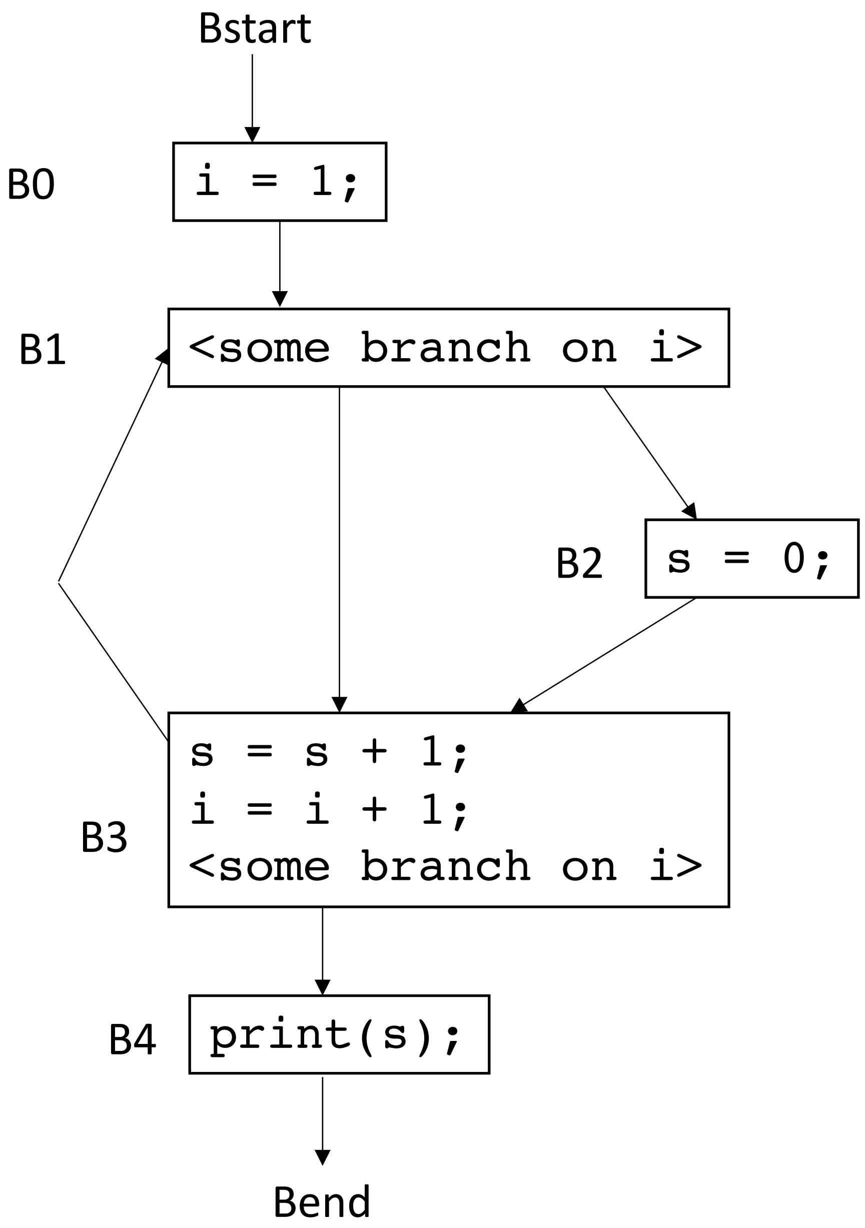
B4
```
print(s);
```

Bend

Now we can perform the iterative fixed point computation:

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

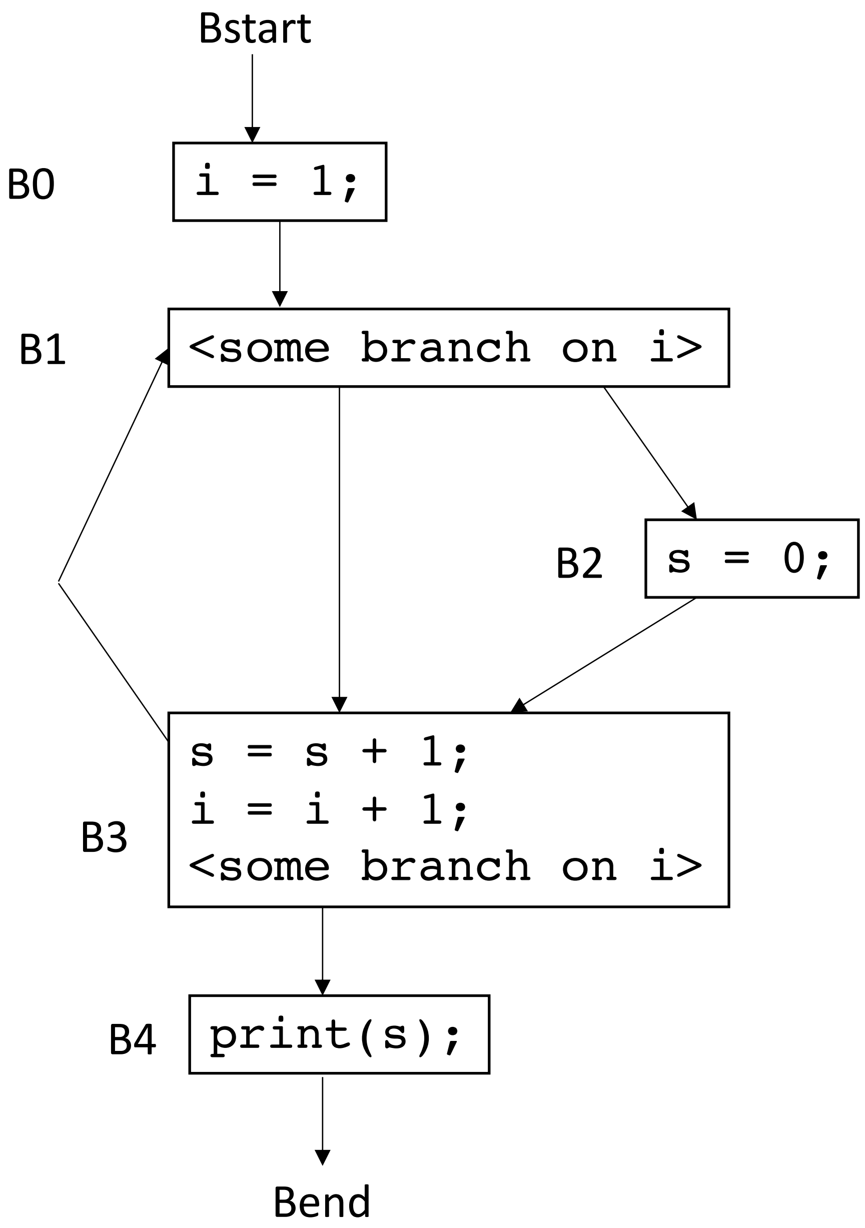
Block	VarKill	UEVar	LiveOut I_0	LiveOut I_1	LiveOut I_2	LiveOut I_3
Bstart	{}	{}	{}	{}	{}	
B0	i	{}	{}	i	s,i	
B1	{}	i	{}	s,i	s,i	
B2	s	{}	{}	s,i	s,i	
B3	s,i	s,i	{}	s,i	s,i	
B4	{}	s	{}	{}	{}	
Bend	{}	{}	{}	{}	{}	



Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	LiveOut l_0	LiveOut l_1	LiveOut l_2	LiveOut l_3
Bstart	{}	{}	{}	{}	{}	s
B0	i	{}	{}	i	s,i	s,i
B1	{}	i	{}	s,i	s,i	s,i
B2	s	{}	{}	s,i	s,i	s,i
B3	s,i	s,i	{}	s,i	s,i	s,i
B4	{}	s	{}	{}	{}	{}
Bend	{}	{}	{}	{}	{}	{}



```
i = 1;
<some branch on i>
s = 0;
s = s + 1;
i = i + 1;
<some branch on i>
print(s);
```

Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	LiveOut l_0	LiveOut l_1	LiveOut l_2	LiveOut l_3
Bstart	{}	{}	{}	{}	{}	s
B0	i	{}	{}	i	s,i	s,i
B1	{}	i	{}	s,i	s,i	s,i
B2	s	{}	{}	s,i	s,i	s,i
B3	s,i	s,i	{}	s,i	s,i	s,i
B4	{}	s	{}	{}	{}	{}
Bend	{}	{}	{}	{}	{}	{}

Node ordering for backwards flow

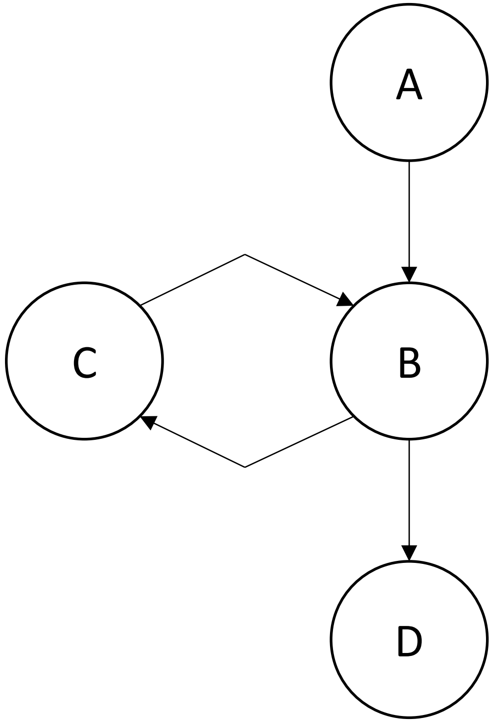
- Reverse post-order was good for forward flow:
  - Parents are computed before their children

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

- Different from post order?

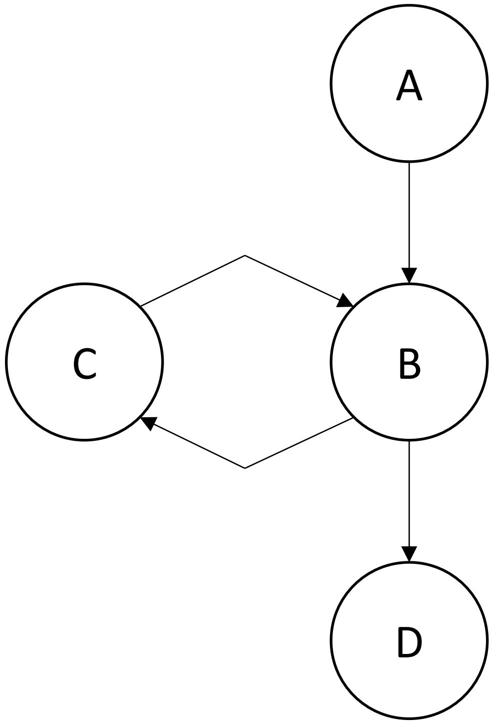
Example

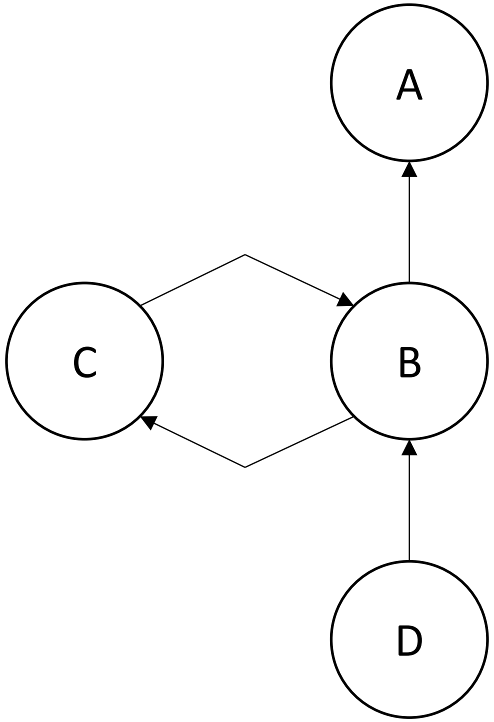
post order: D, C, B, A



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



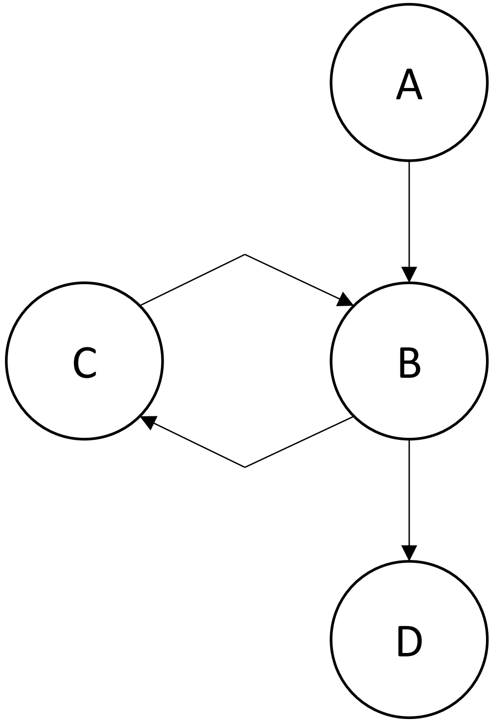


reverse CFG

post order: D, C, B, A

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

Example

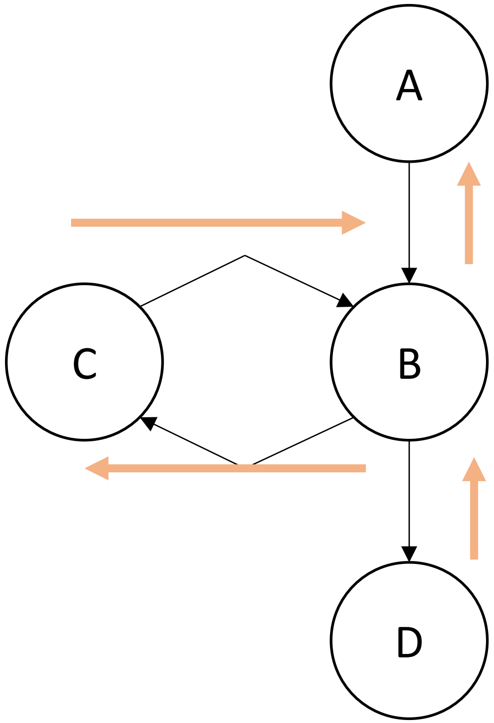


post order: D, C, B, A

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



updates in backwards flow

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

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*

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

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;
```

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

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*

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

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.

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

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
```

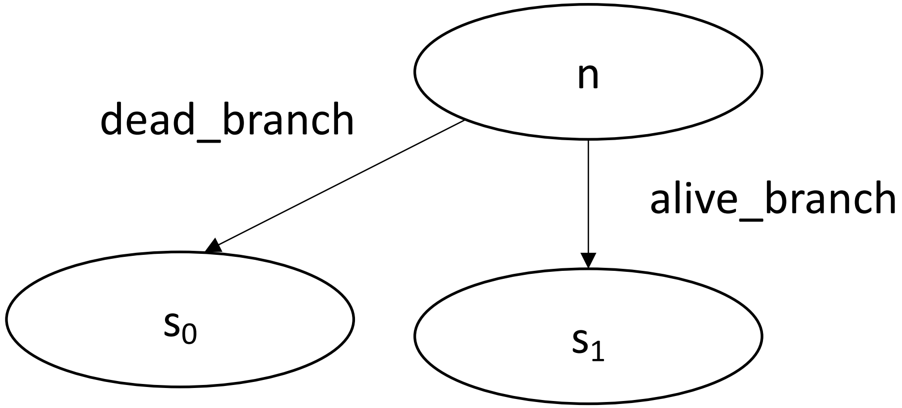
Live variable limitations

Imprecision can come from CFG construction:

consider:

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

could come from arguments, etc.



Live variable limitations

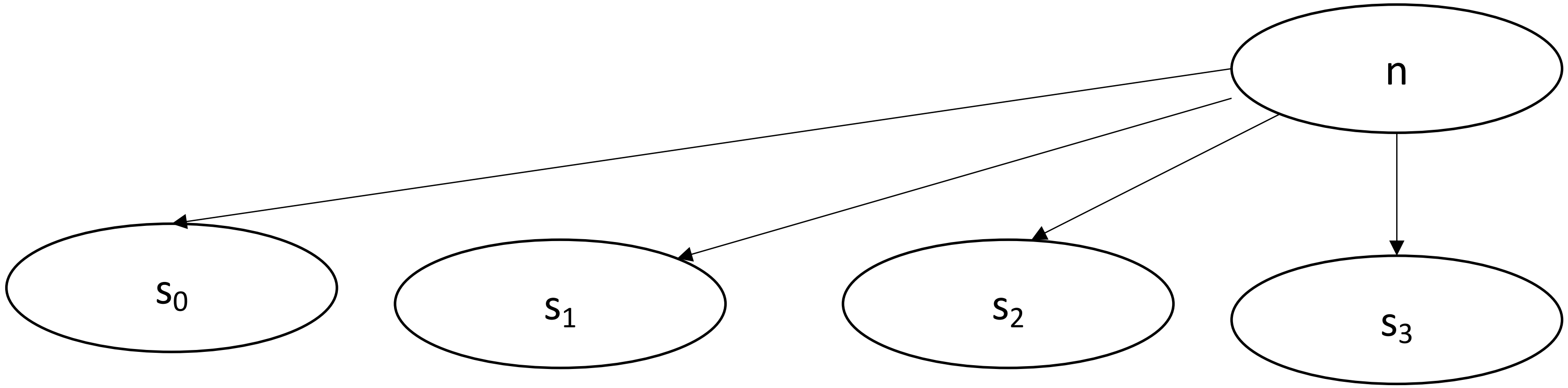
Imprecision can come from CFG construction:

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

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

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

Available Expressions

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

An expression e is “available” at 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

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

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

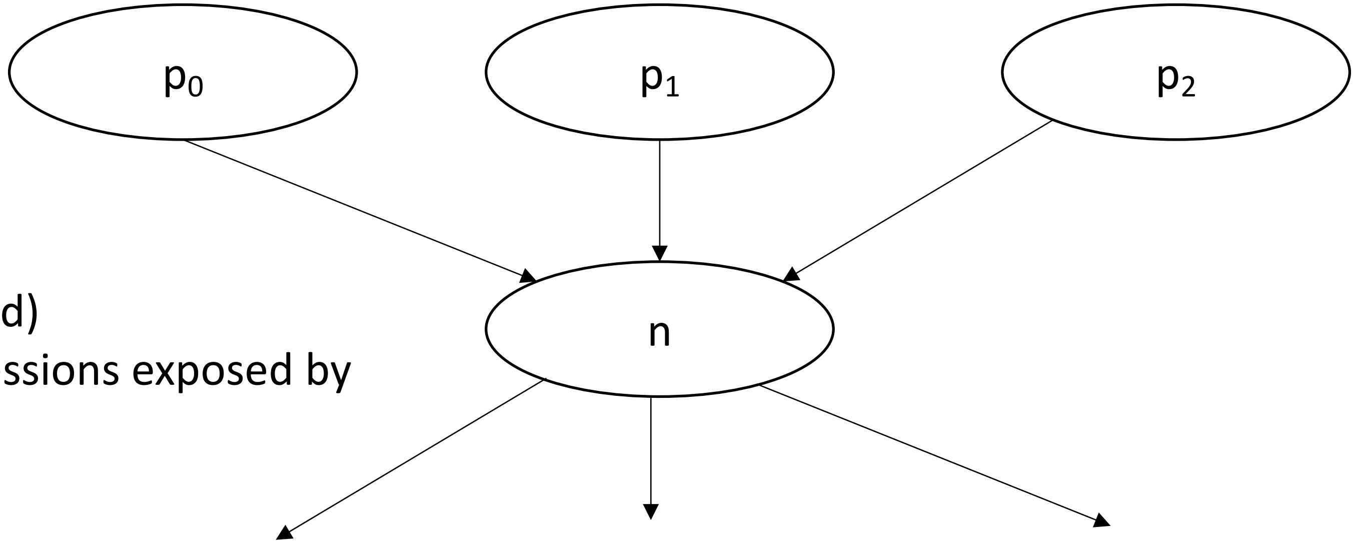
Available Expressions

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

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

Available Expressions

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



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

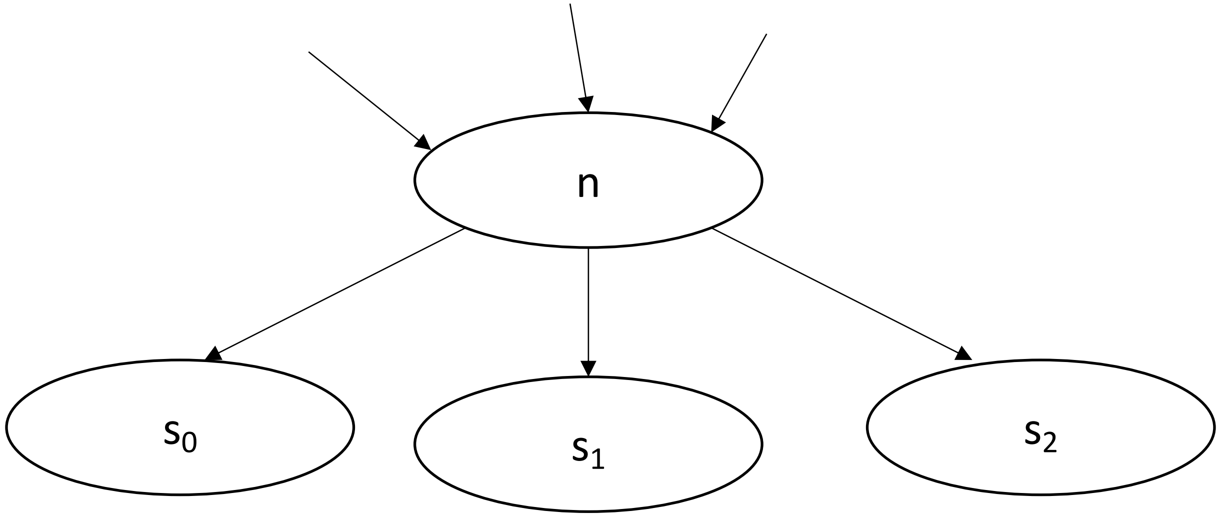
Anticipable Expressions

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

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

Anticipable Expressions

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



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*

Reaching Definitions

- Read about this in 9.2.4

- trace variable usages in block b to possible definitions

- can be used in alias analysis

Next Lecture

- SSA form and homework