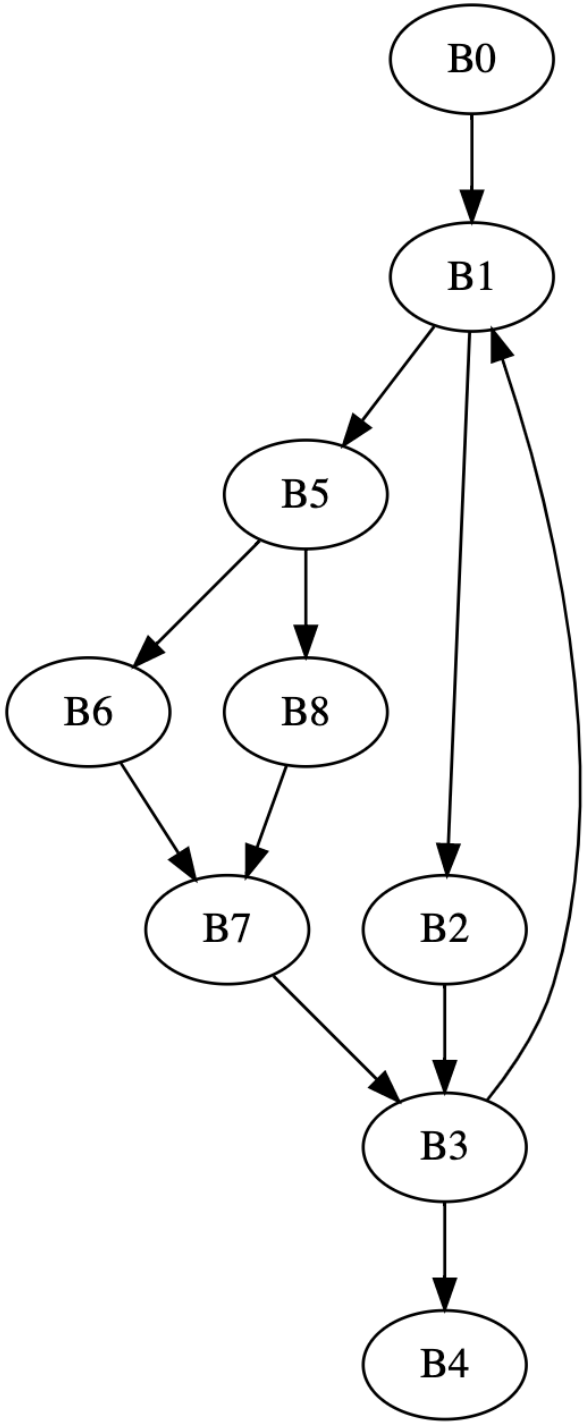
CSE211: Compiler Design

Oct. 13, 2022

- **Topic**: Regional optimizations, intro to global optimizations

- **Questions**:
  - What are some of your favorite compiler optimizations?
  - *Can we apply local value numbering to an entire program?*
  - *What are examples of having unlimited registers vs. having limited registers?*



Announcements

- Office hours tomorrow:
  - 3:30 - 5:30 PM

- Homework 1:
  - Due on Tuesday (at 11:59 pm)
  - Help will be sparse in evenings and weekends!

- Homework 2:
  - Aim is to release on Tuesday by midnight
  - 2 weeks to complete
    - Local Value Numbering
    - Live variable analysis

Announcements

Next week:

- Thursday I will be in Phoenix. I will provide a lecture recording.
- Office hours will be on Tuesday:
  - 3:30 - 5:30
- No more travel for the quarter

Thinking about part 2 of the homework

- Who has started?

- What has been your implementation design?
  - AST Types
  - parsing
  - derivatives
  - optimizations

- How to implement the option operator (?)

Review local value numbering

First step?

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

Review local value numbering

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

```
H = {
}
```

Review local value numbering

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

```
H = {
      “b0 + c1” : “a2”,
}
```

Review local value numbering

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

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
}
```

Review local value numbering

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

```
H = {
      “b0 + c1” : “a2”,
      “a2 - d3” : ”b4”,
}
```

Review local value numbering

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

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
}
```

mismatch due to numberings!

Review local value numbering

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

```
H = {
      “b0 + c1” : “a2”,
      “a2 - d3” : ”b4”,
      “b4 + c1” : “c5”,
}
```

Review local value numbering

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

```
H = {
      “b0 + c1” : “a2”,
      “a2 - d3” : ”b4”,
      “b4 + c1” : “c5”,
}
```

Review local value numbering

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

```
H = {
      "b0 + c1" : "a2",
      "a2 - d3" : "b4",
      "b4 + c1" : "c5",
}
```

match!

Other LVN considerations?

Other LVN considerations?

Can this block be optimized?

```
a = b + c;
f = a - d;
c = c + b;
d = d - a;
```

Other LVN considerations?

What about this one?

```
a = b + c;
a = a - d;
c = b + c;
d = a - d;
```

Local value numbering: Memory

- Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];
b[i] = x[j] + y[k];
```

is this transformation allowed?

```
a[i] = x[j] + y[k];
b[i] = a[i];
```

Local value numbering: Memory

- Consider a 3 address code that allows memory accesses

```
a[i] = x[j] + y[k];
b[i] = x[j] + y[k];
```

is this transformation allowed?
No!

only if the compiler can prove that `a` does not alias `x` and `y`

```
a[i] = x[j] + y[k];
b[i] = a[i];
```

In the worst case, every time a memory location is updated, the compiler must update the value for all pointers.

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair

```
(a[i],3) = (x[j],1) + (y[k],2);
b[i] = x[j] + y[k];
```

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair

  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

compiler analysis:

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],1) + (y[k],2);
```

in this case we do not have to update the number

compiler analysis:

can we trace `a,x,y` to

```
a = malloc(…);
x = malloc(…);
y = malloc(…);
```

// `a,x,y` are never overwritten

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (x[j],4) + (y[k],5);
```

in this case we do not have to update the number

`restrict a`

programmer annotations can also tell the compiler that no other pointer can access the memory pointed to by a

Local value numbering: Memory

- How to number:
  - Number each pointer/index pair
  - Any pointer/index pair that might alias must be incremented at each instruction

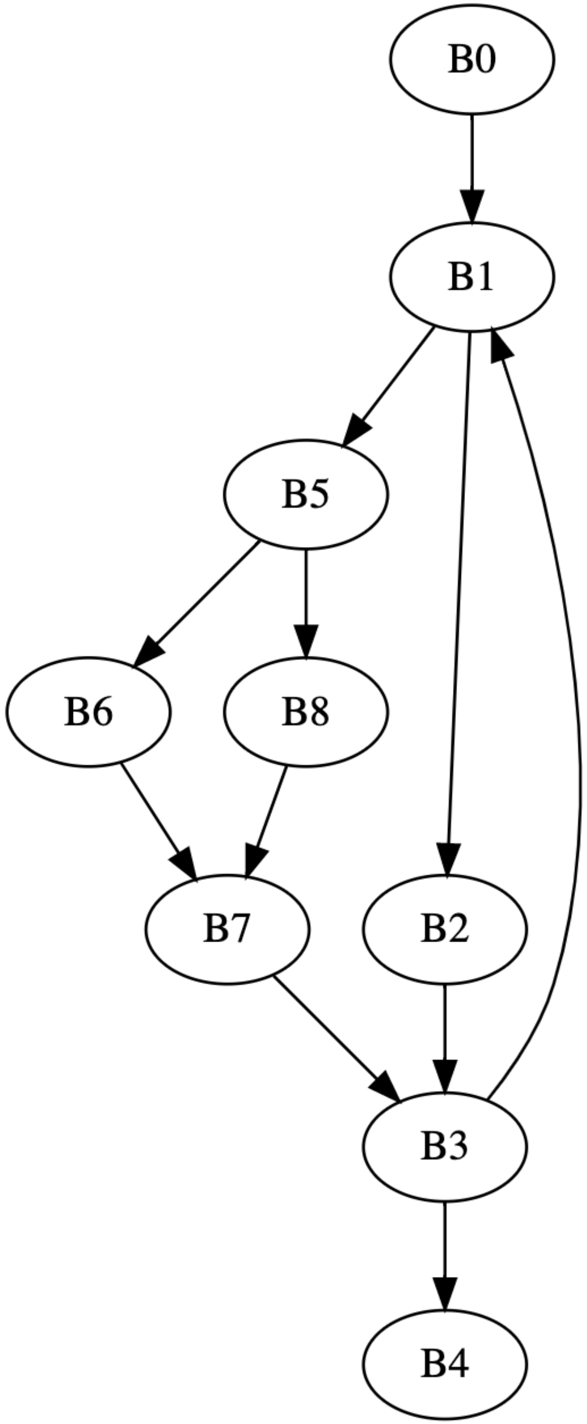
```
(a[i],3) = (x[j],1) + (y[k],2);
(b[i],6) = (a[i],3);
```

CSE211: Compiler Design

Oct. 13, 2022

- **Topic**: Regional optimizations, intro to global optimizations

- **Questions**:
  - What are some of your favorite compiler optimizations?
  - *Can we apply local value numbering to an entire program?*
  - *What are examples of having unlimited registers vs. having limited registers?*



Optimizing over wider regions

- Local value numbering operated over just one basic block.
- We want optimizations that operate over:
  - several basic blocks (regional)
  - across an entire procedure (global)
- For this, we need Control Flow Graphs

Control flow graphs

A graph where:

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

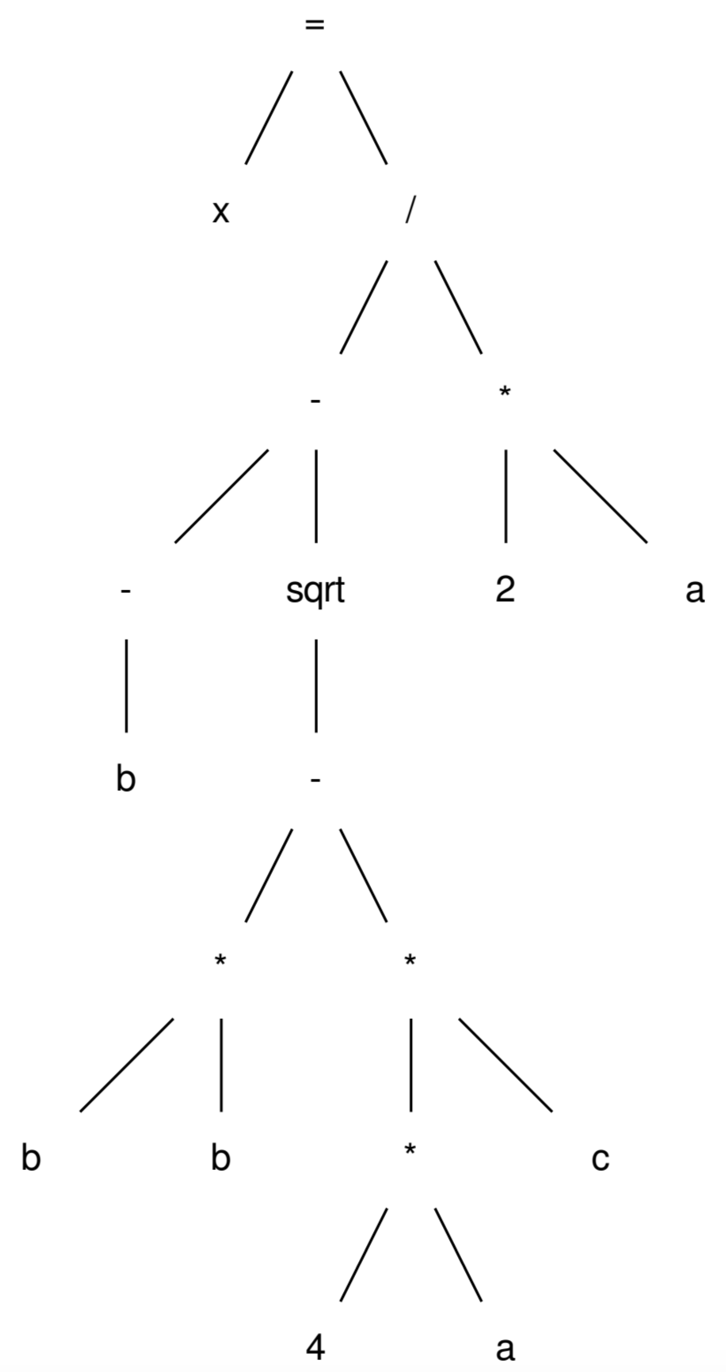
reminder, what is a basic block?
What is 3 address code?

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

if:
r2 = ...;
br end_if;

else:
r3 = ...;

end_if:
r4 = ...;
```

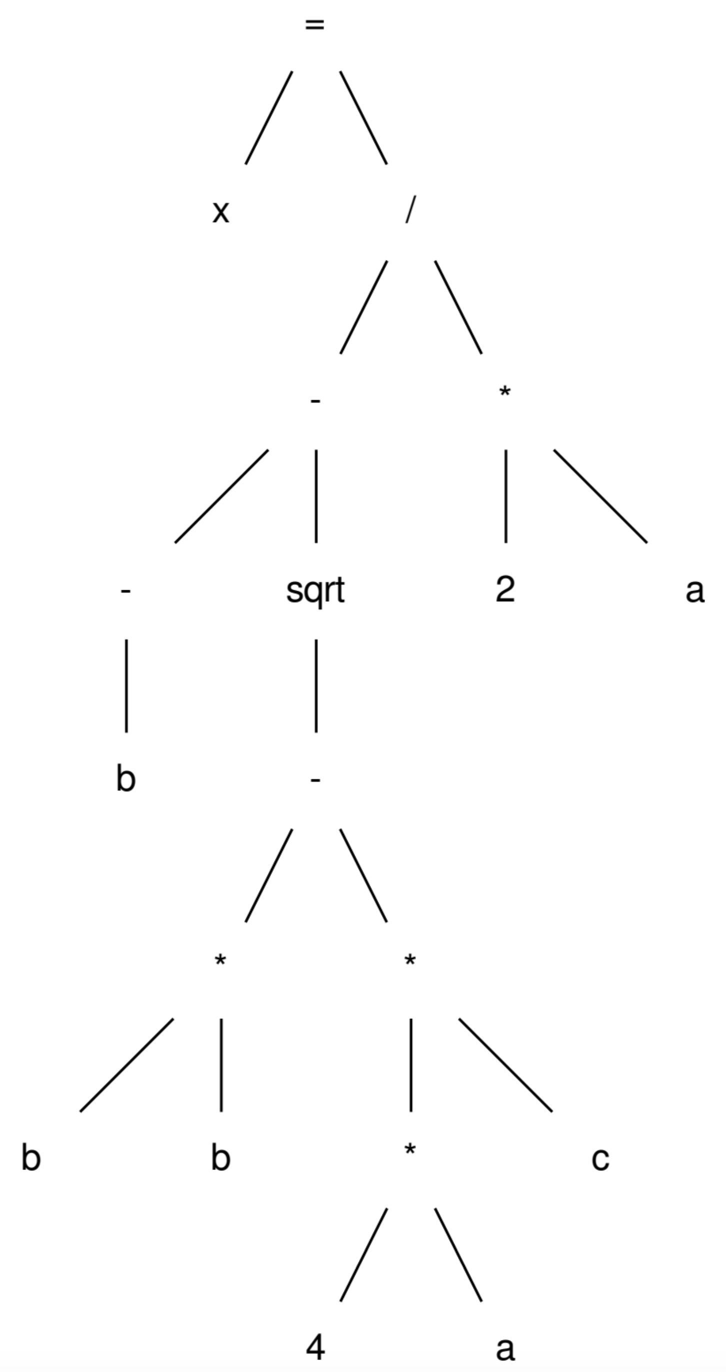
Review IRs:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```



```
r0 = neg(b);
r1 = b * b;
r2 = 4 * a;
r3 = r2 * c;
r4 = r1 – r3;
r5 = sqrt(r4);
r6 = r0 – r5;
r7 = 2 * a;
r8 = r6 / r7;
x  = r8;
```



What are some properties of 3 address code?

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;
```

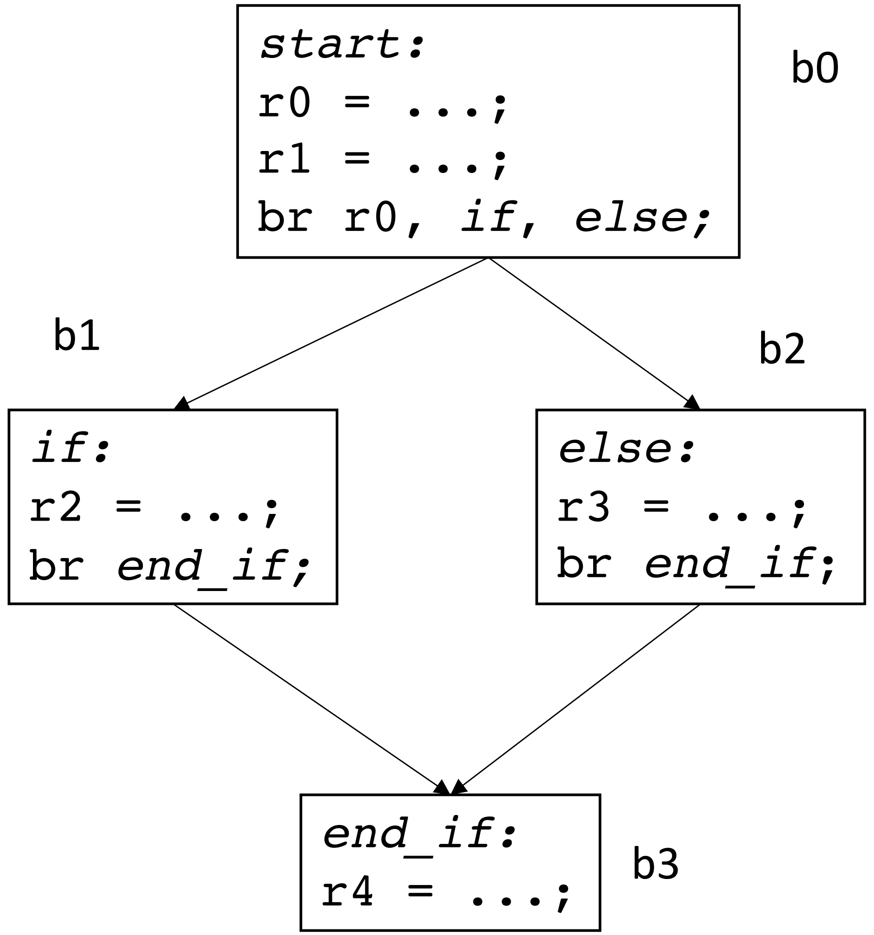
```
end_if:
r4 = ...;
```

Control flow graphs

A graph where:

- nodes are basic blocks

- edges mean that it is possible for one block to branch to another



Interesting CFGs

What are some you can think of?

Interesting CFGs

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

Regional optimizations

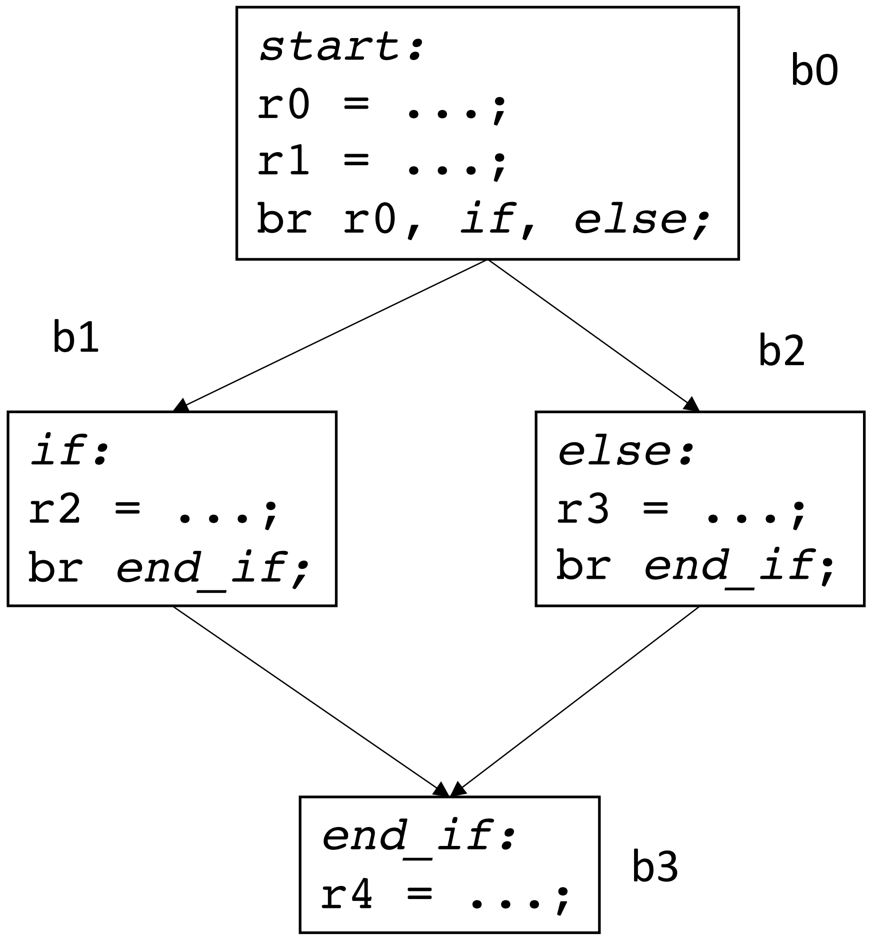
- Usually constrained to a "common" subset of the CFG:
- For example: if/else statements

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

if:
r2 = ...;
br end_if;

else:
r3 = ...;

end_if:
r4 = ...;
```

Regional optimizations

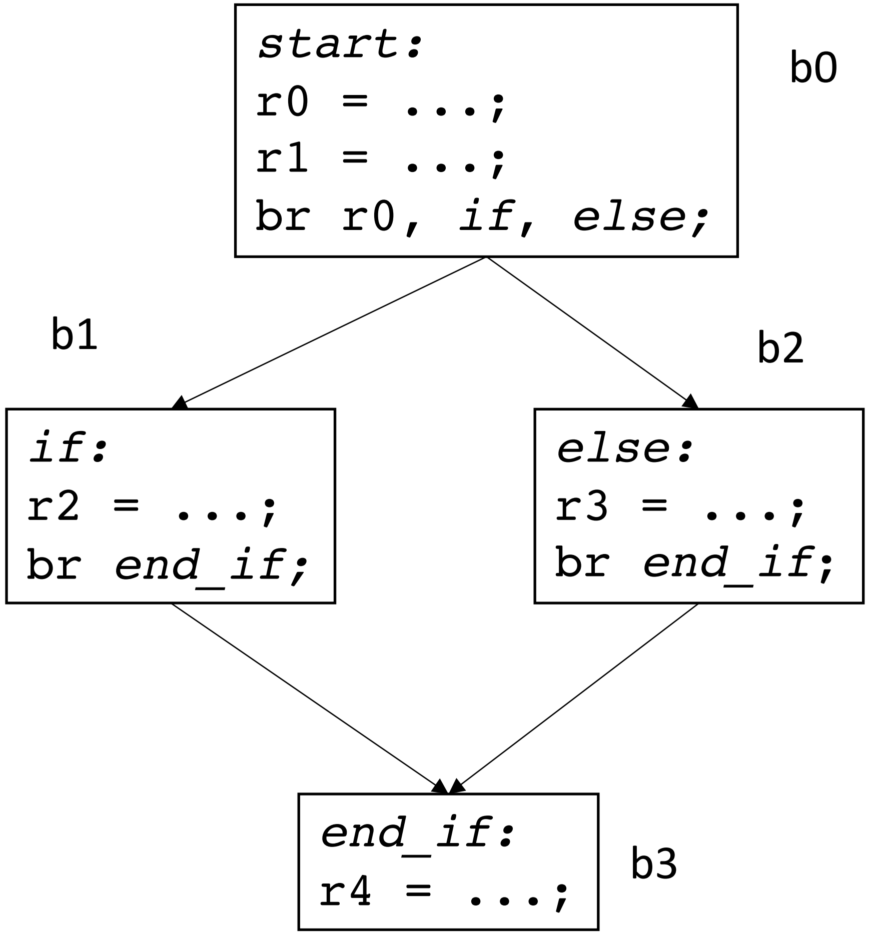
- Usually constrained to a "common" subset of the CFG:

- For example: if/else statements



Super local value numbering

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



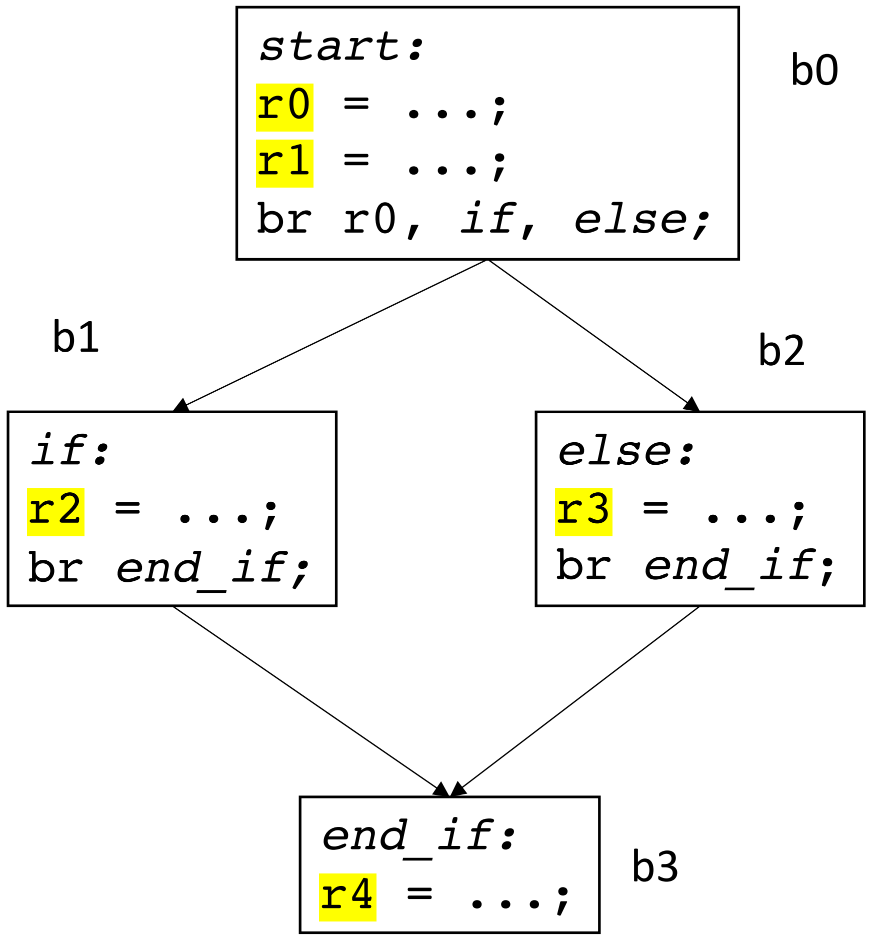
What are the implications of doing local value numbering in each of the basic blocks?

Super local value numbering

- Usually constrained to a "common" subset of the CFG:

- For example: if/else statements

Global counter would need to be kept across blocks when numbering



What are the implications of doing local value numbering in each of the basic blocks?

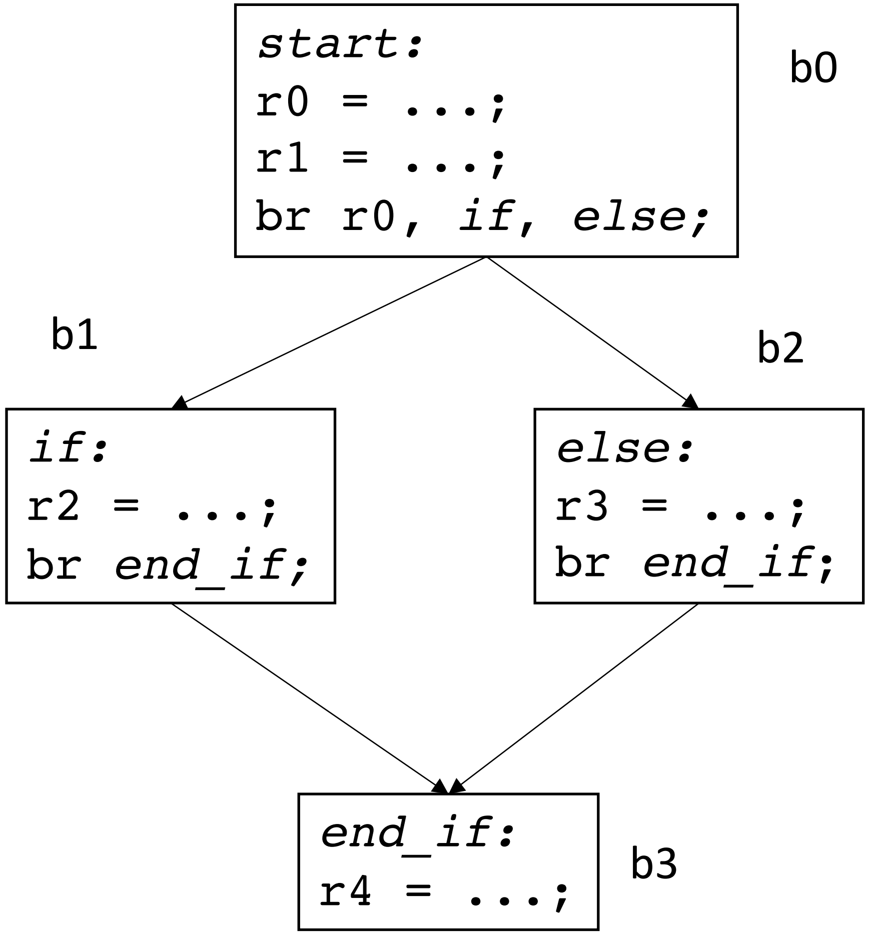
Super local value numbering

- Usually constrained to a "common" subset of the CFG:

- For example: if/else statements

breadth first traversal, creating hash tables for each block

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

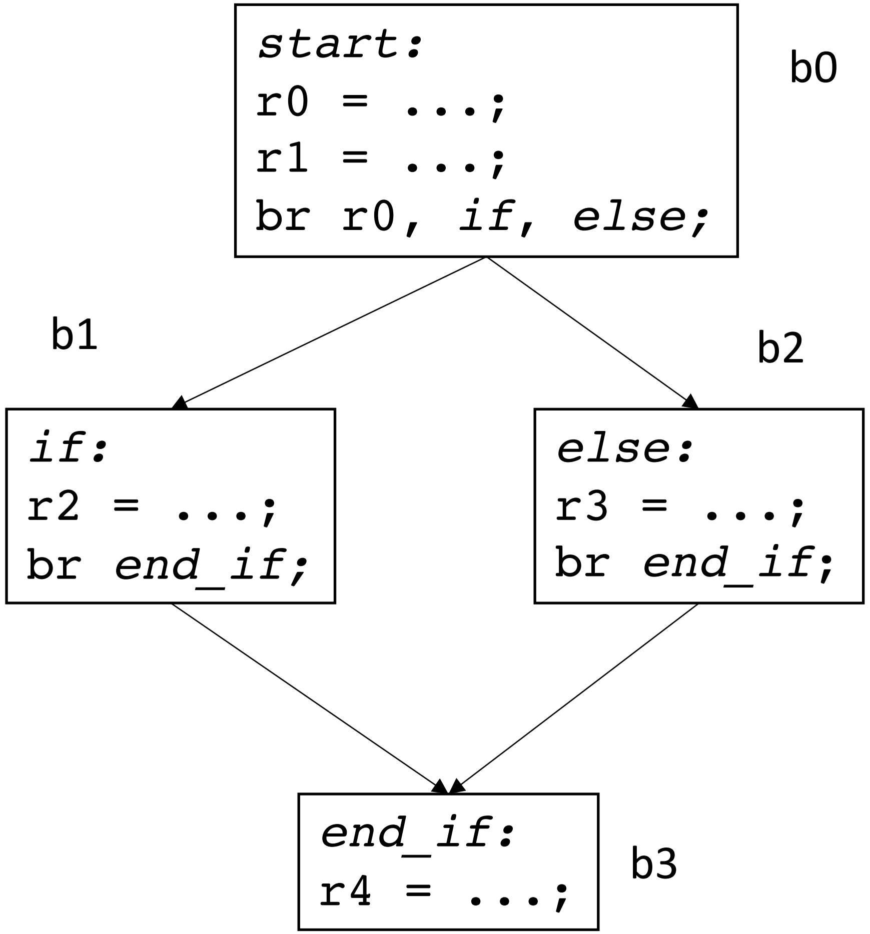


What are the implications of doing local value numbering in each of the basic blocks?

Super local value numbering

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

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



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

Which blocks can use which hash tables?

Super local value numbering

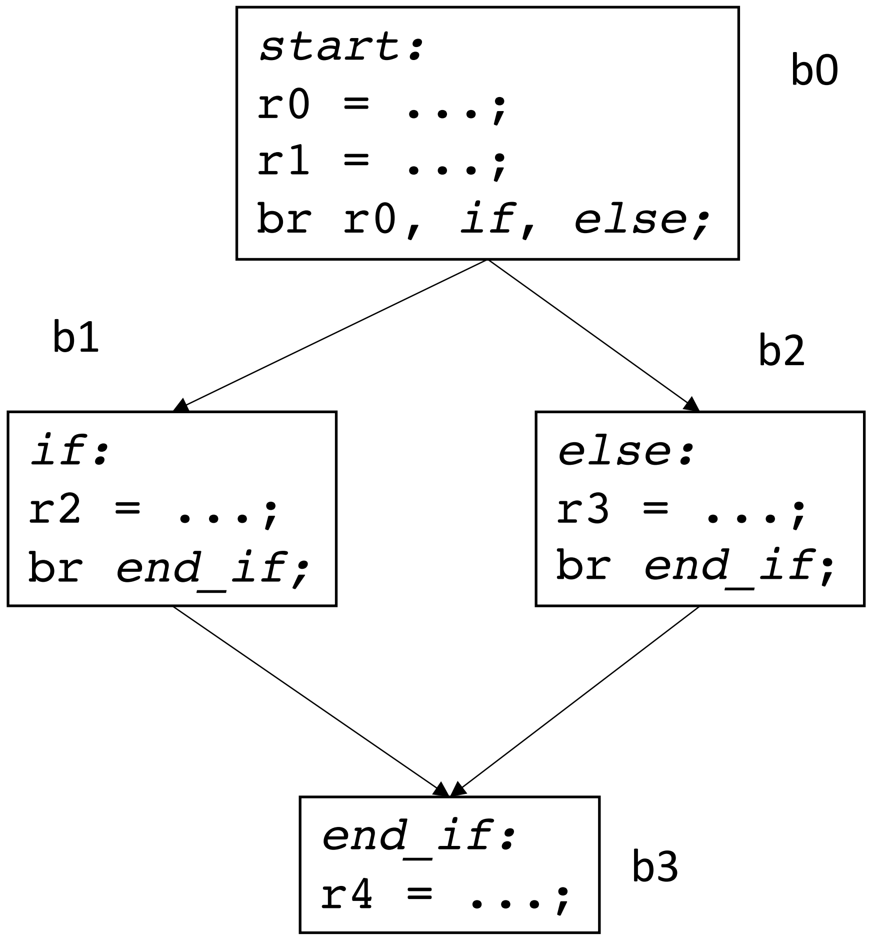
- Usually constrained to a “common” subset of the CFG:

- For example: if/else statements

Is it possible to re-write so that b3 can use expressions from b1 or b2?

breadth first traversal, creating hash tables for each block

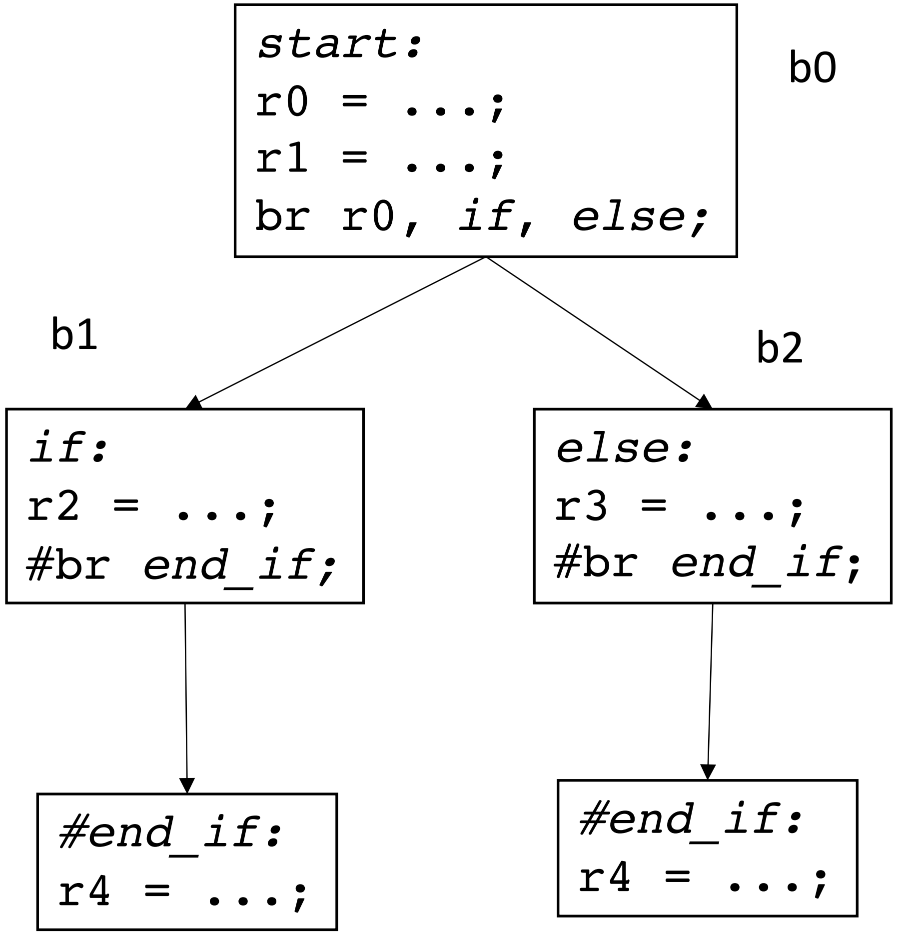
```
b0_H = {
        "..." : "r0",
        "..." : "r1",
}
```



Super local value numbering

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

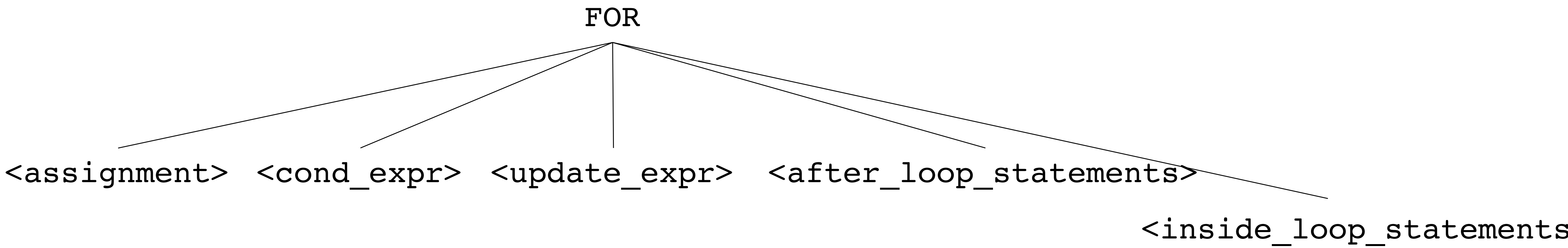
```
b0_H = {
        "..." : "r0",
        "..." : "r1",
}
```

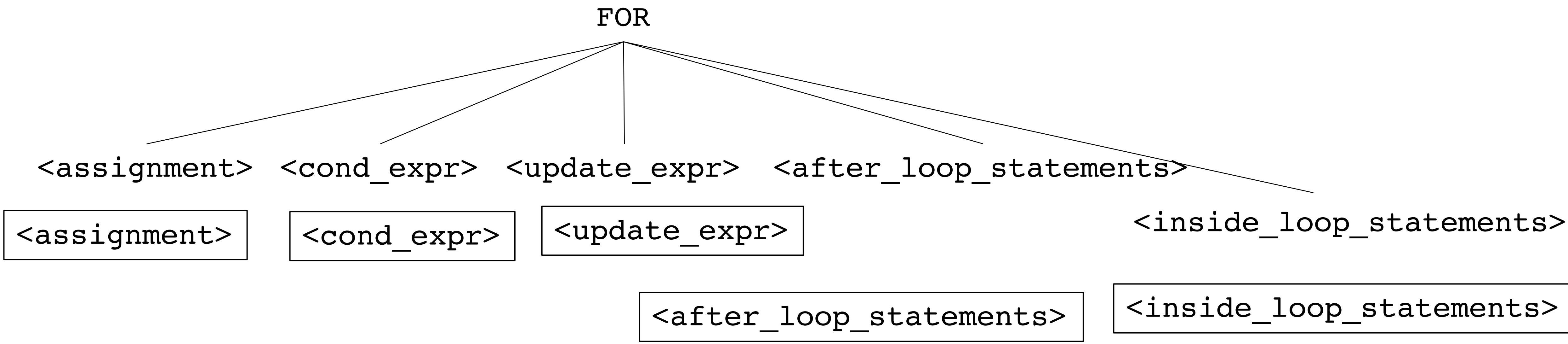


Is it possible to re-write so that b3 can use expressions from b1 and b2? Duplicate blocks and merge!

Pros? Cons?

Loop unrolling:

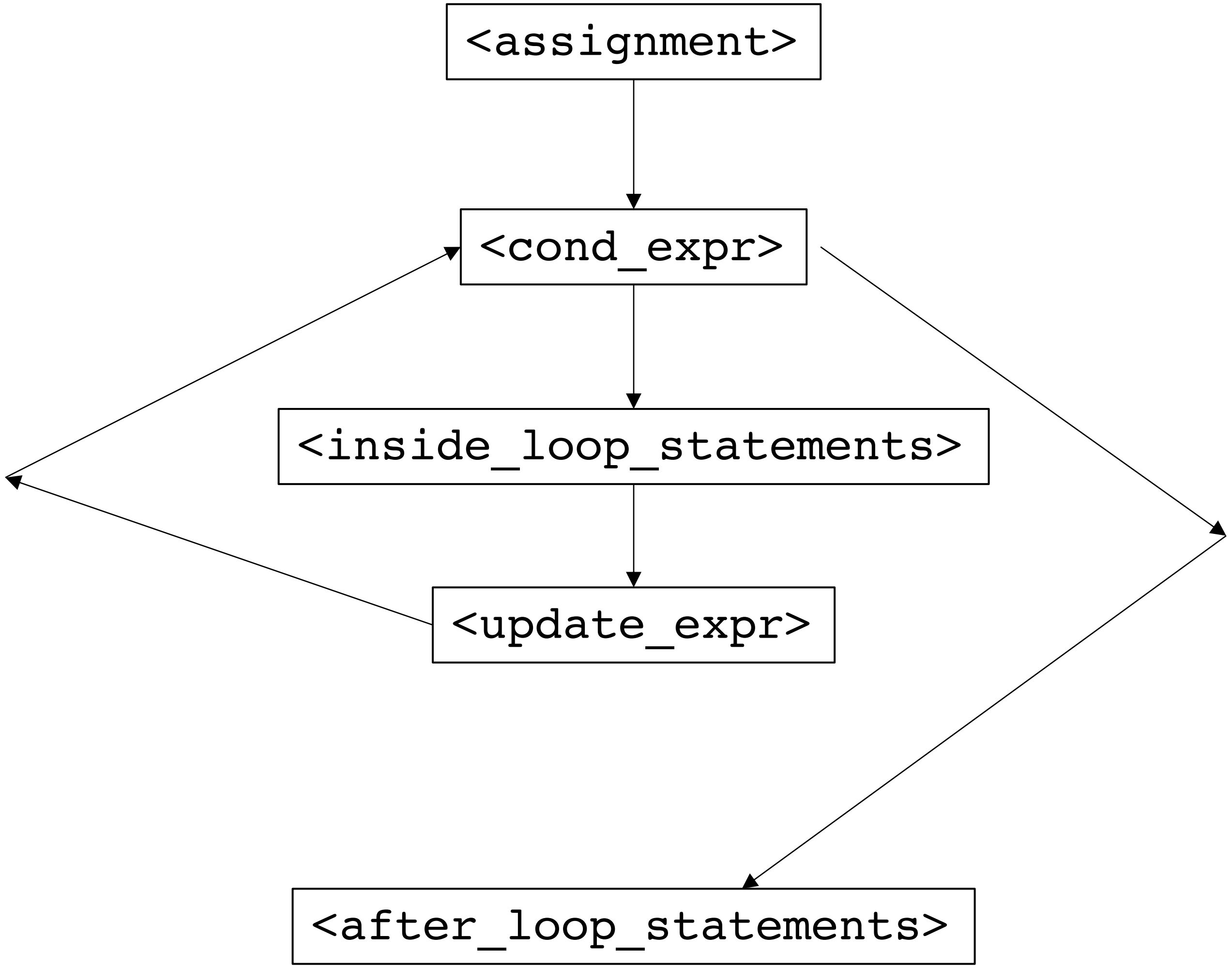




If all of these are basic blocks then the CFG looks like:

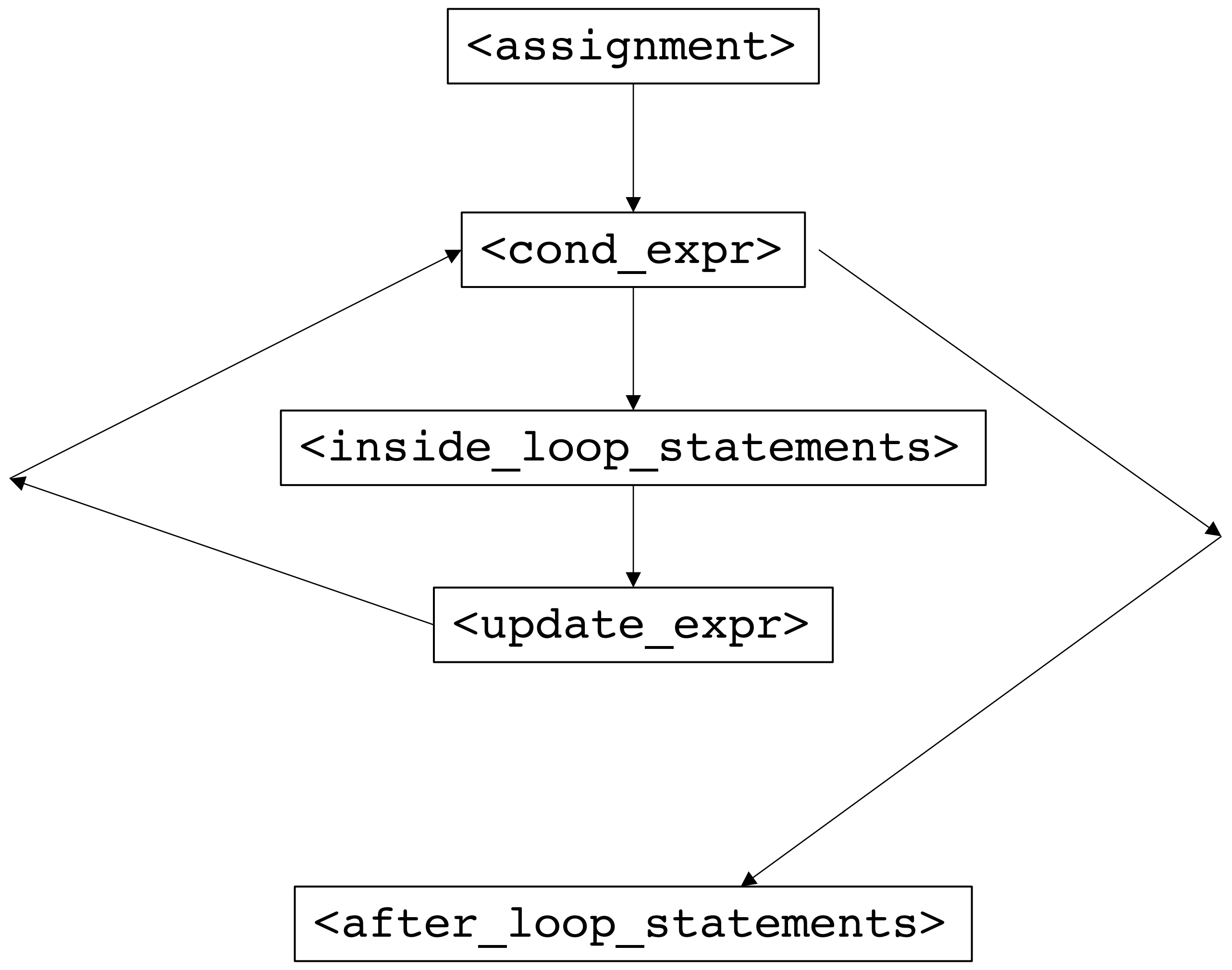
Loop unrolling:

What could change this CFG?

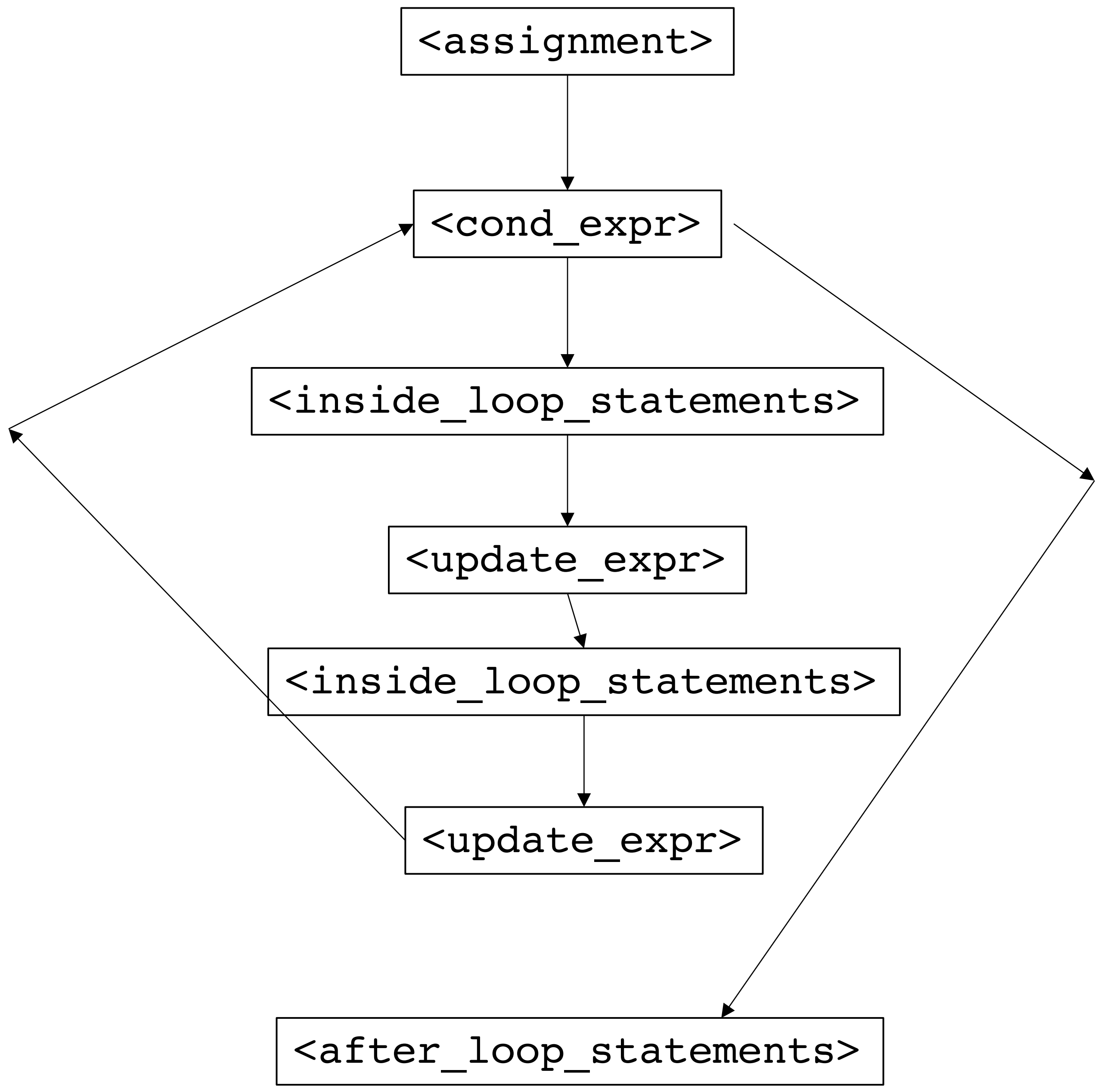


Loop unrolling:

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



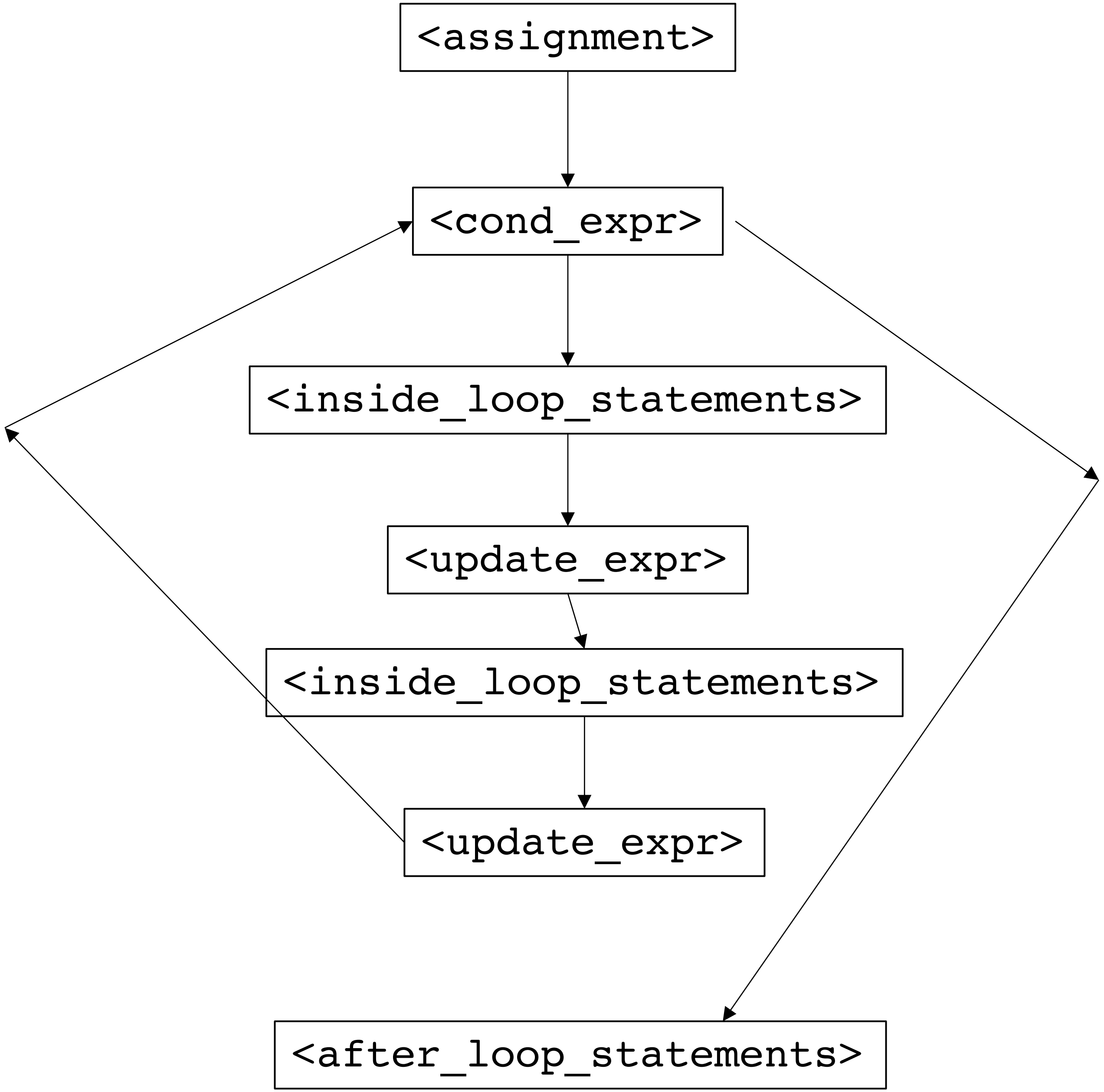
Loop unrolling:



Loop unrolling:

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

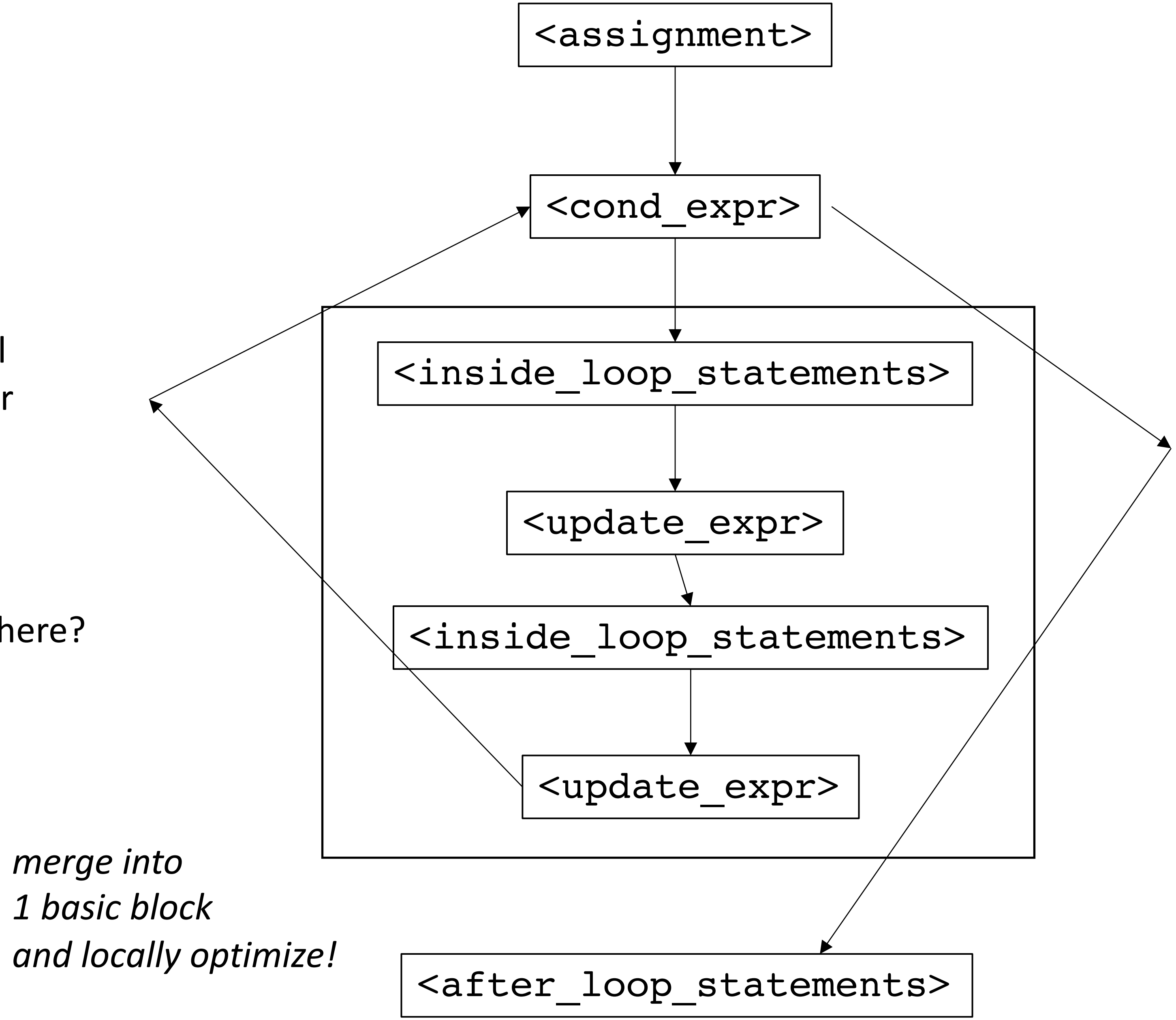
What have we saved here?



Loop unrolling:

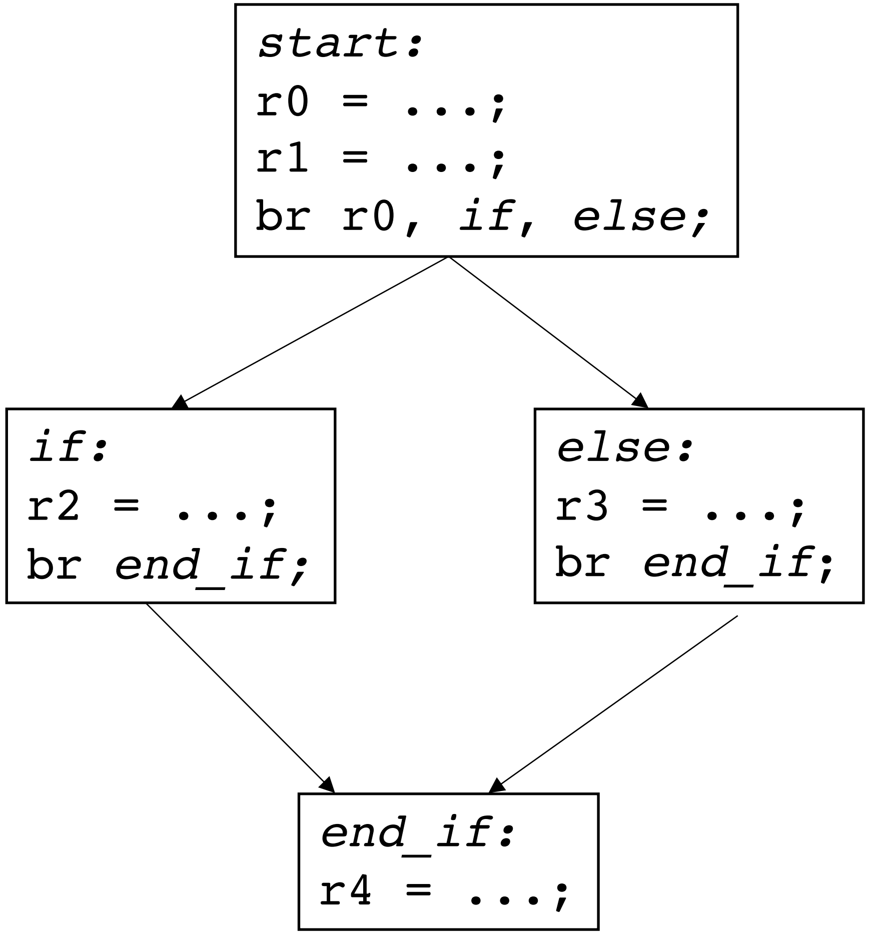
Assume we
know that the loop will
iterate an even number
of times:

What have we saved here?



Code placement:

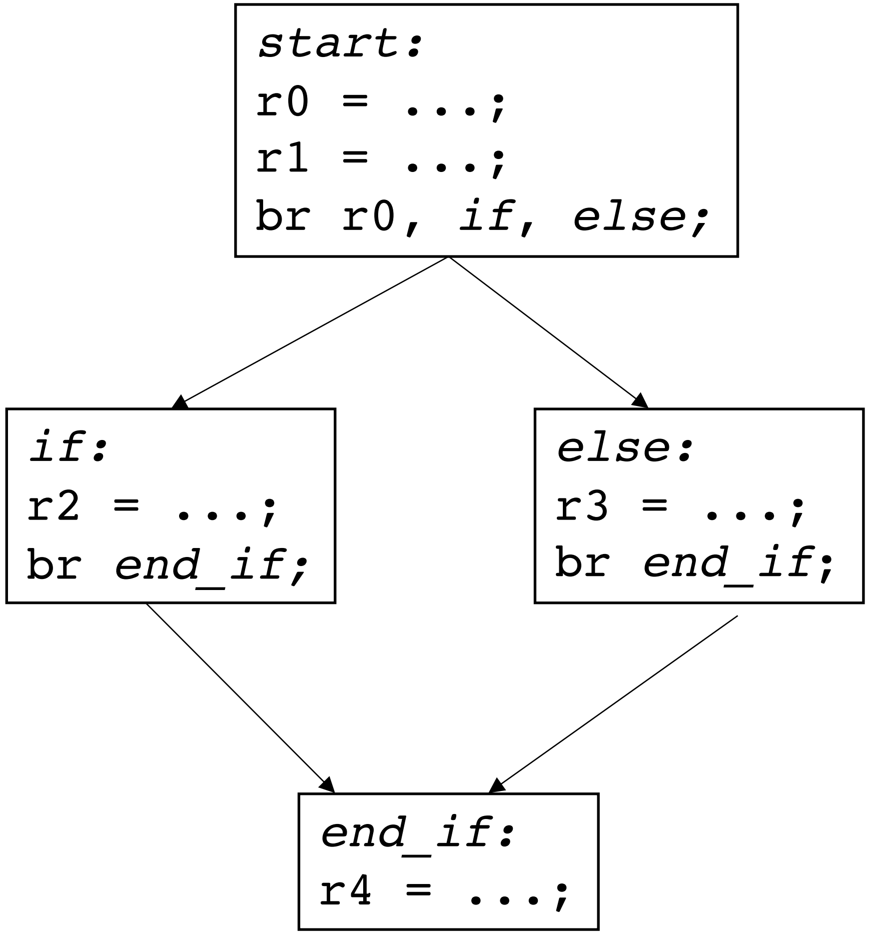
- Back to if/else



Code placement:

- Back to if/else

- Eventually we will straight line the code:



Code placement:

- Back to if/else
- Eventually we will straight line the code:

one option, what else?

```
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:

one option, what else?

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

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

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

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

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

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

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

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

Performance impact between the two?

Code placement:

- Back to if/else

- Eventually we will straight line the code:

one option, what else?

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

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

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

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

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

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

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

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

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

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 = ...;
```

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

How many branches here

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
```

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

How many branches here

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 = ...;
br end_if;
```

```
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

Global optimizations

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

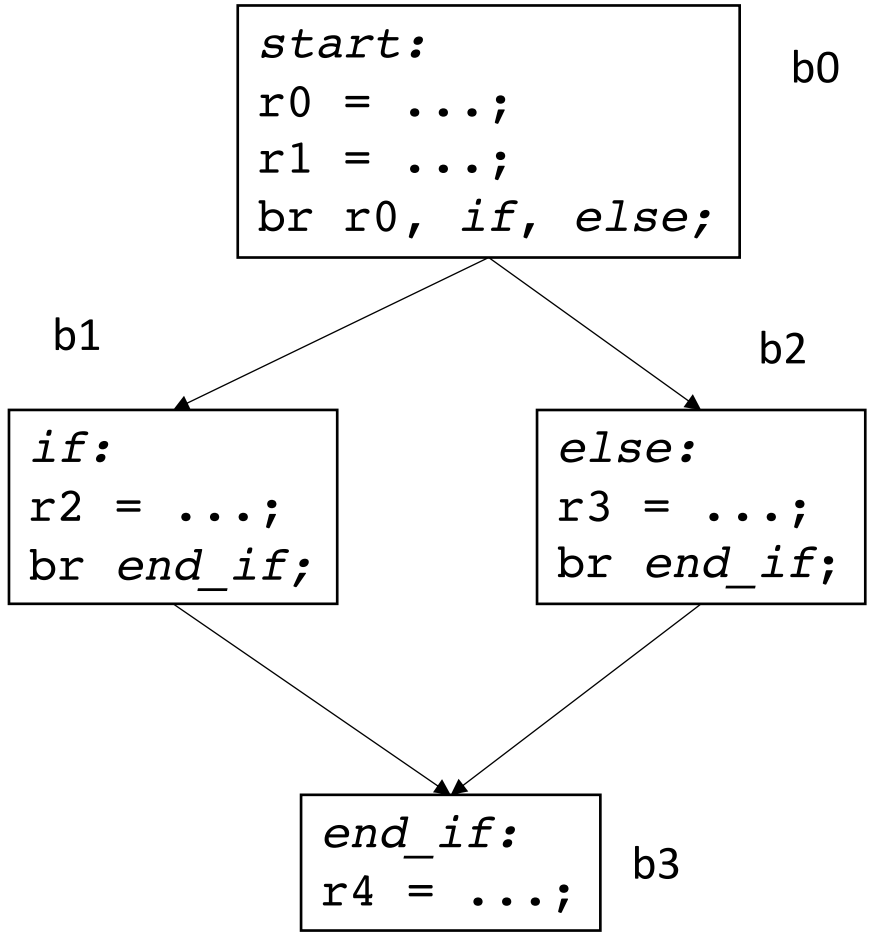
- Highly suggest reading for this part of the class
  - Chapter 9 of EAC

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_x goes through b_y
- definition:
  - domination (includes itself)
  - strict domination (does not include itself)

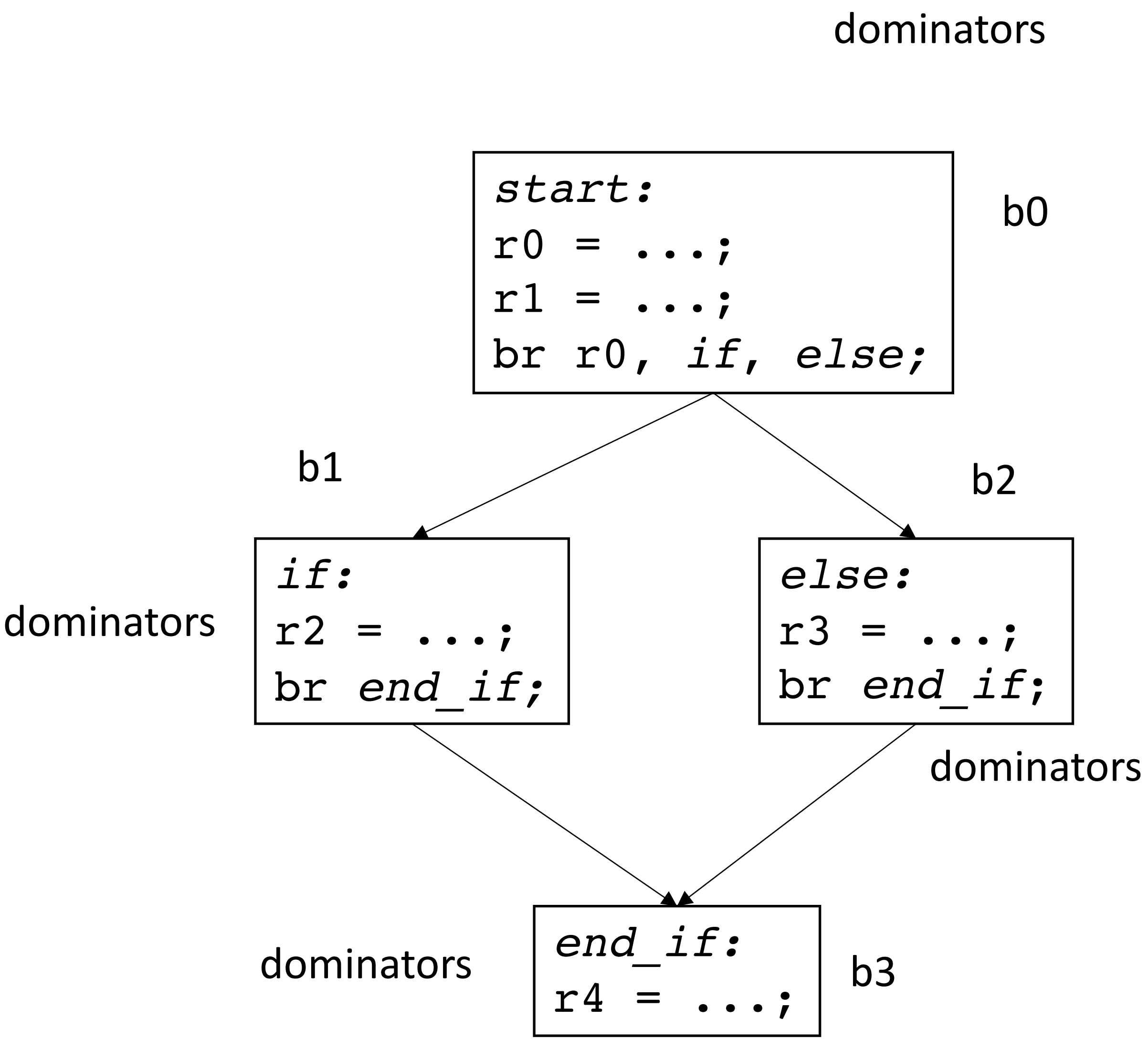


Dominance

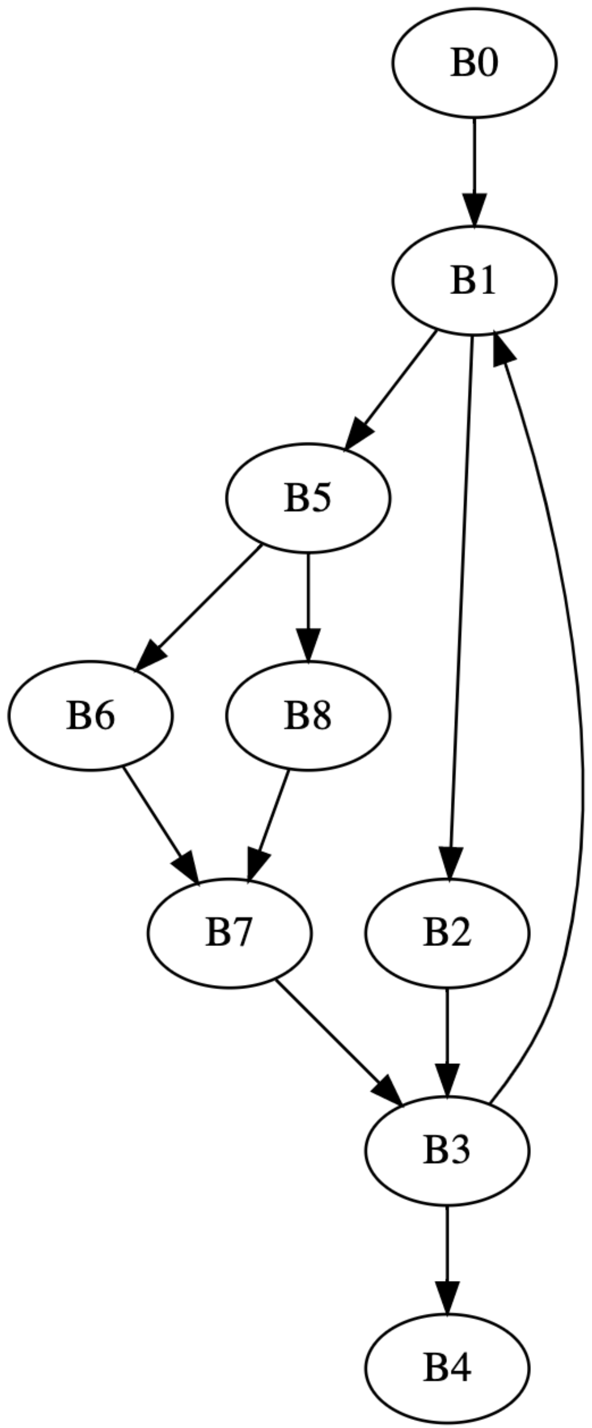
- a block b_x dominates block b_y iff every path from the start to block b_x goes through b_y

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

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

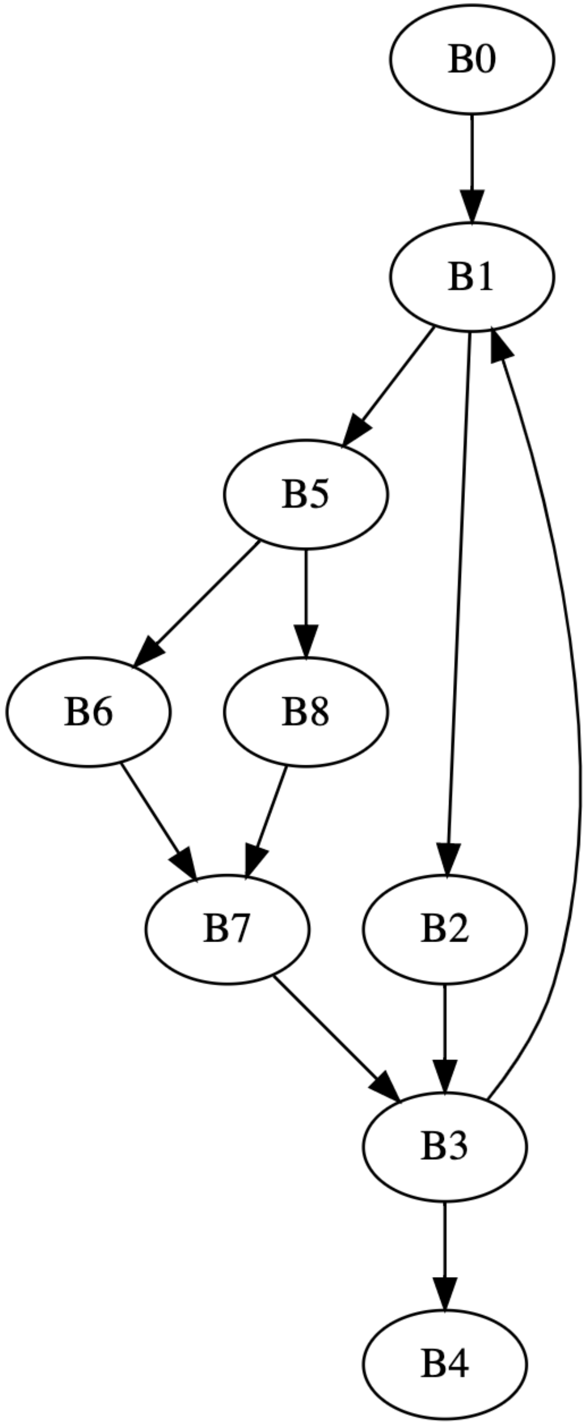


Node	Dominators
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	



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

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



Computing dominance

- Iterative fixed point algorithm

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

iteratively compute:

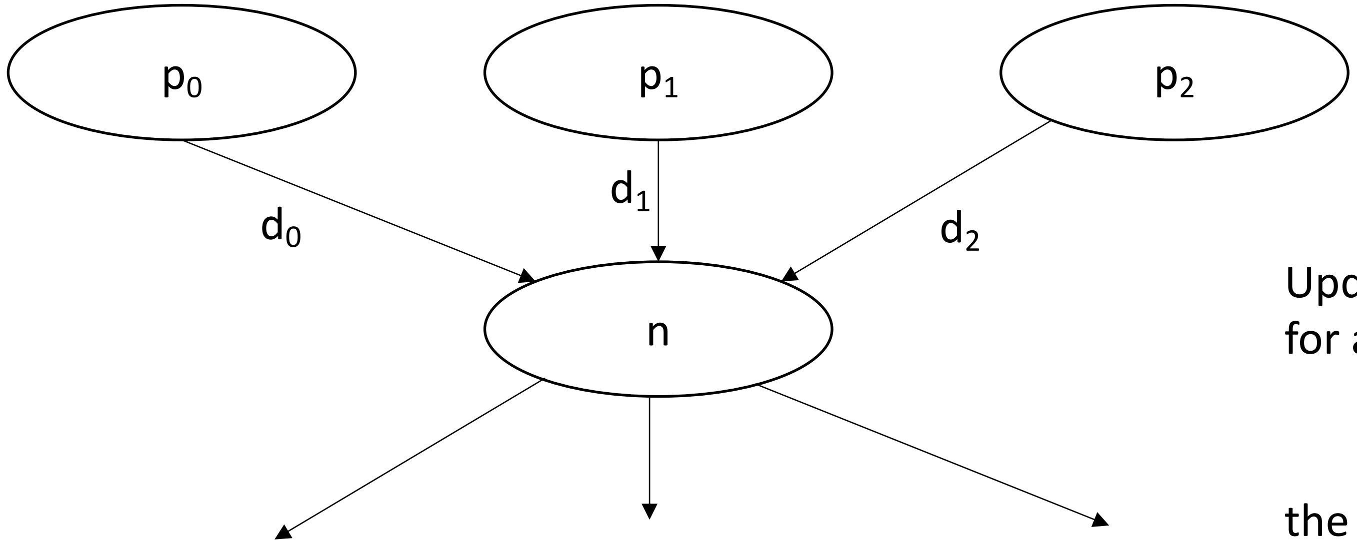
$$Dom(n) = \{n\} \cup \left(\bigcap_{m \text{ in preds}(n)} Dom(m) \right)$$

Building intuition behind the math

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

For example: Bellman Ford Shortest path

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

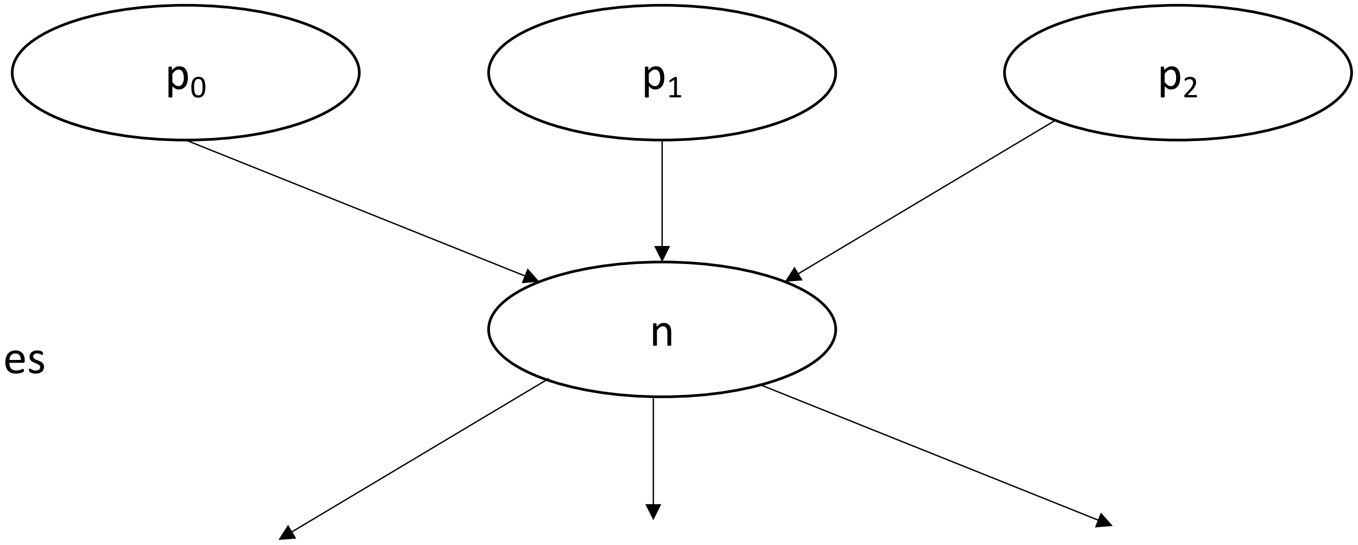


Update:
for all parents *p*: min(*p* + *d*)

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

Now lets think about dominance

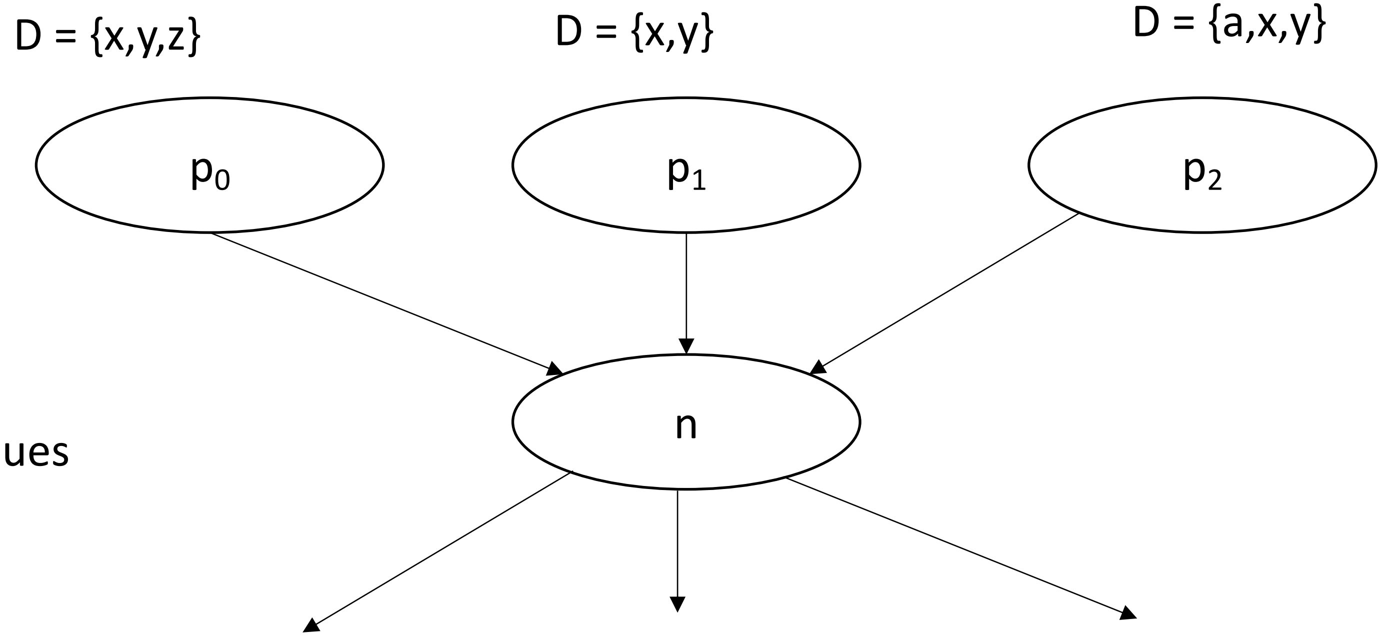
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

Now lets think about dominance

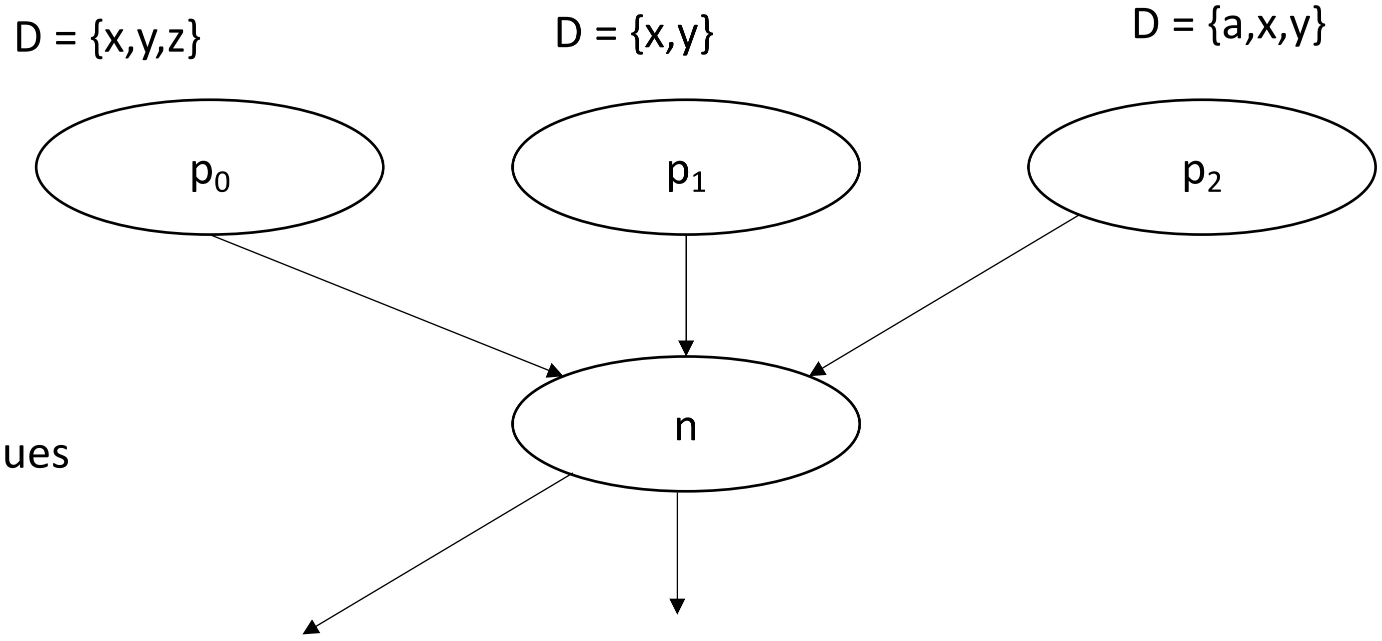
- Root node is initialized to itself
- Every node determines new dominators based on parent dominators



update:
intersection of parent values

Now lets think about dominance

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

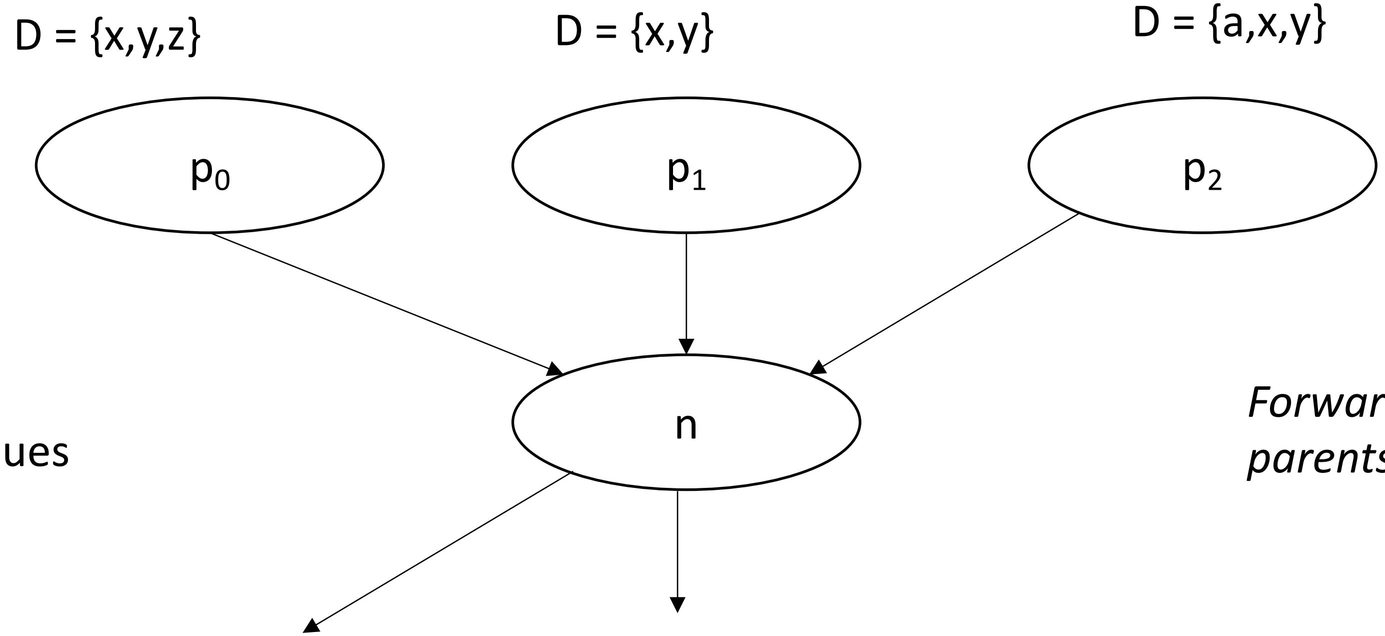


update:
intersection of parent values

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

Now lets think about dominance

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



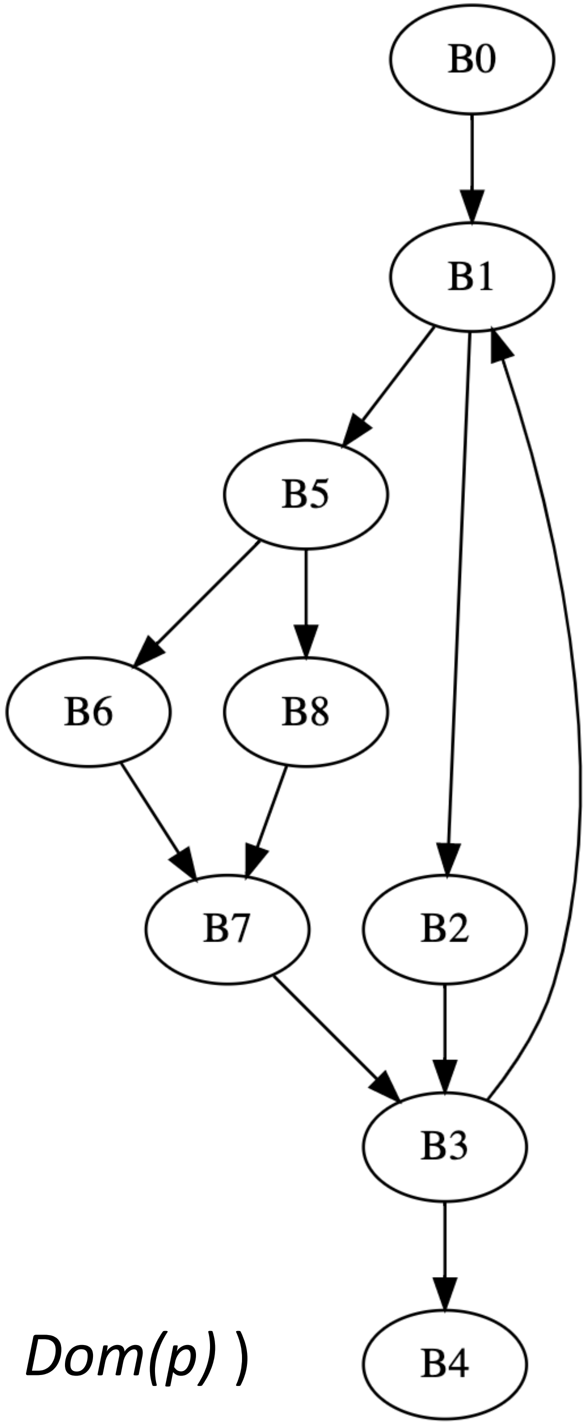
update:
intersection of parent values

Forward flow, as updates flow from parents to children.

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

Lets try it

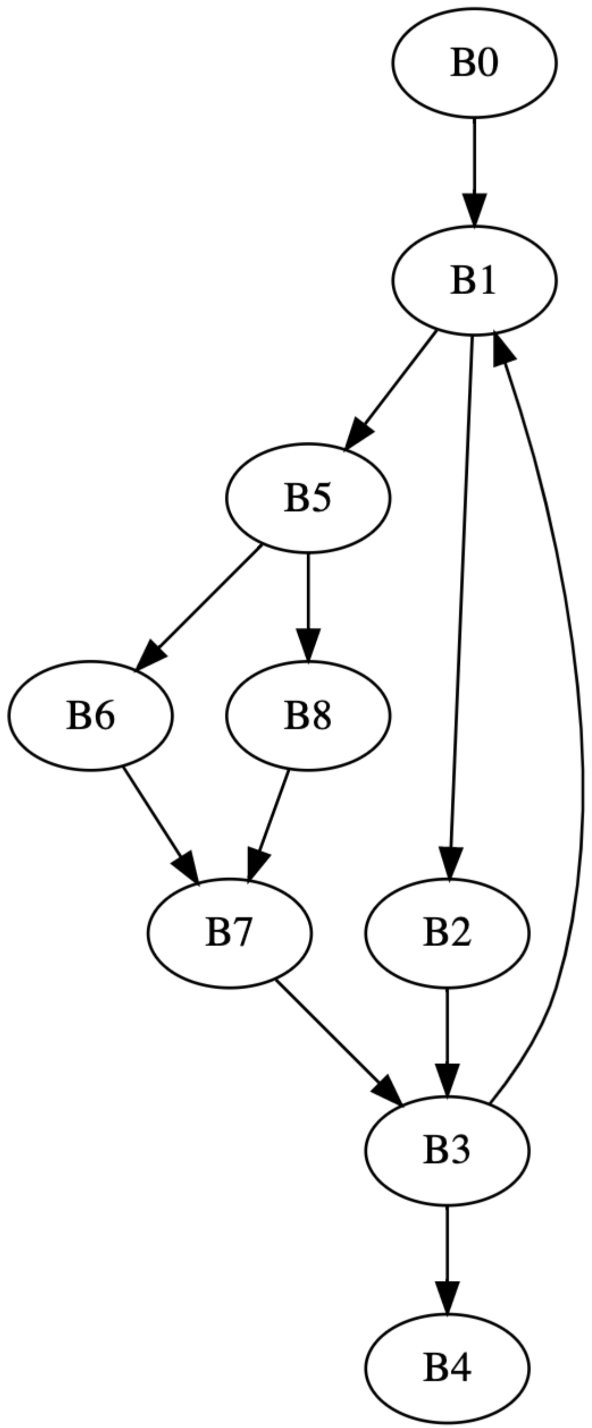
Node	Initial	Iteration 1
B0	B0	
B1	*N*	
B2	*N*	
B3	*N*	
B4	*N*	
B5	*N*	
B6	*N*	
B7	*N*	
B8	*N*	



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

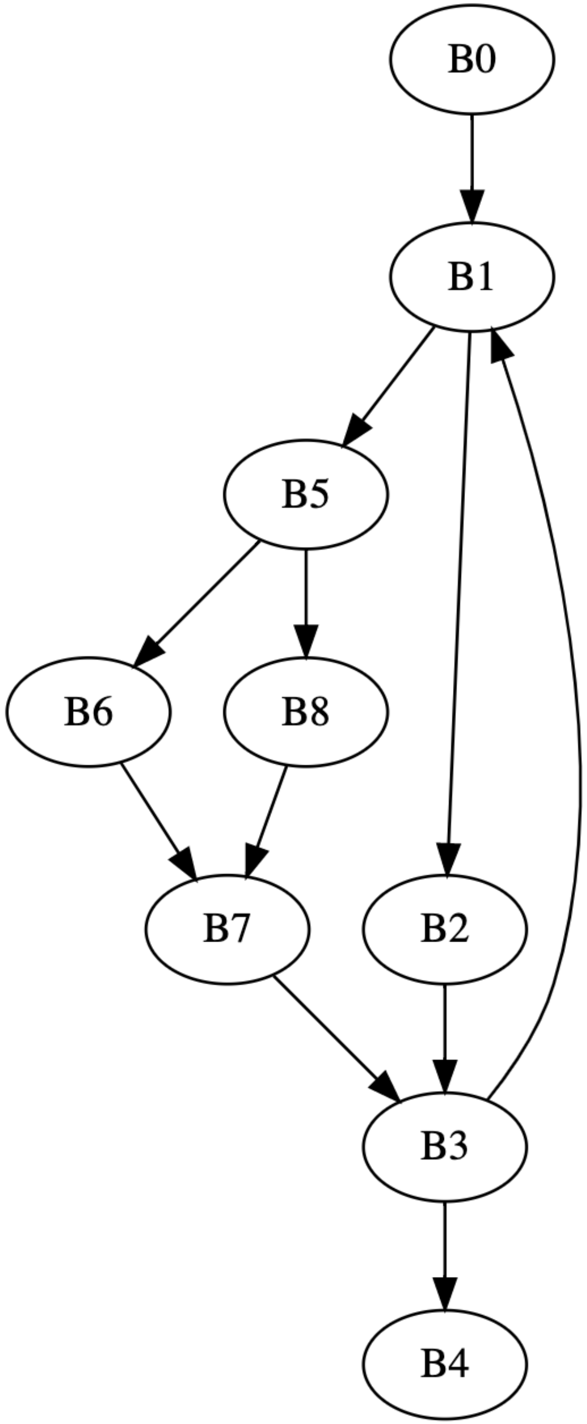
Lets try it

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



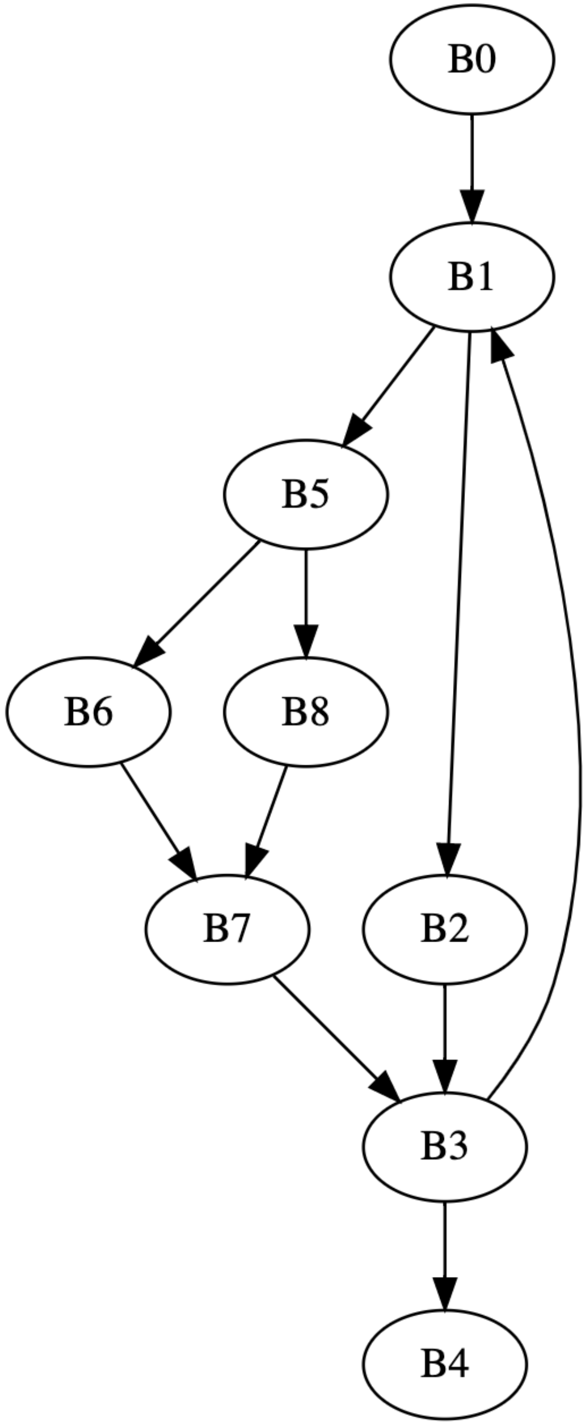
Lets try it

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



How can we optimize the algorithm?

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

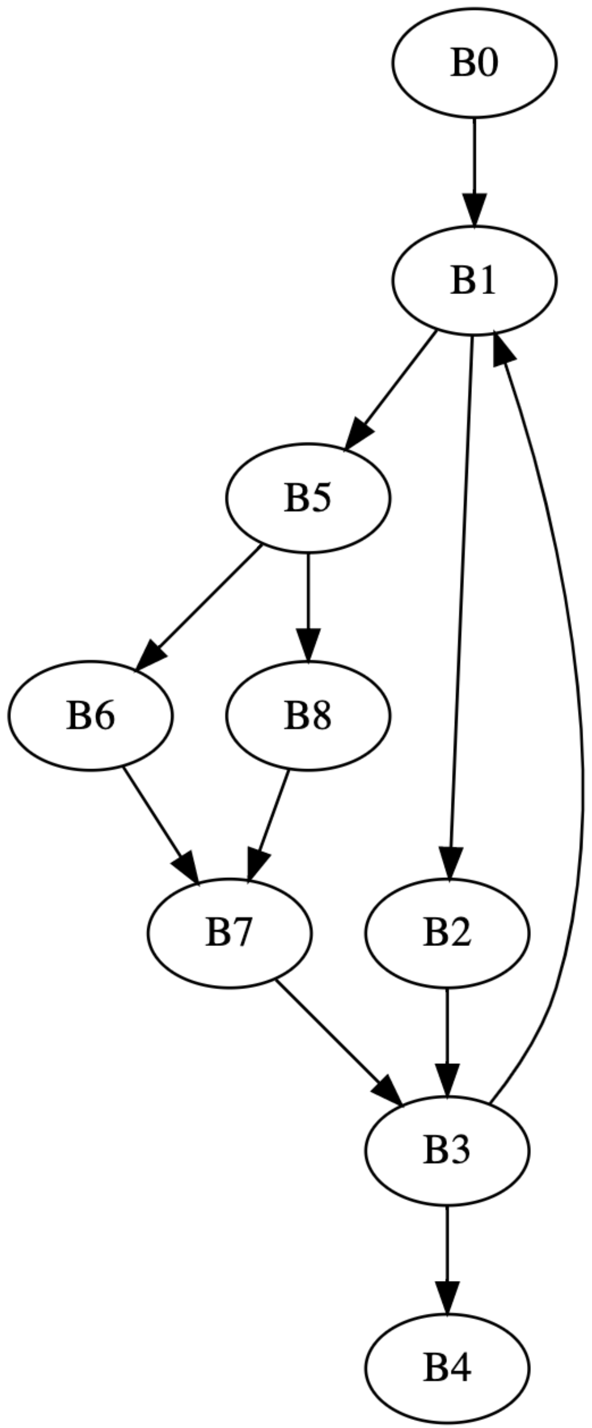


How can we optimize the algorithm?

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

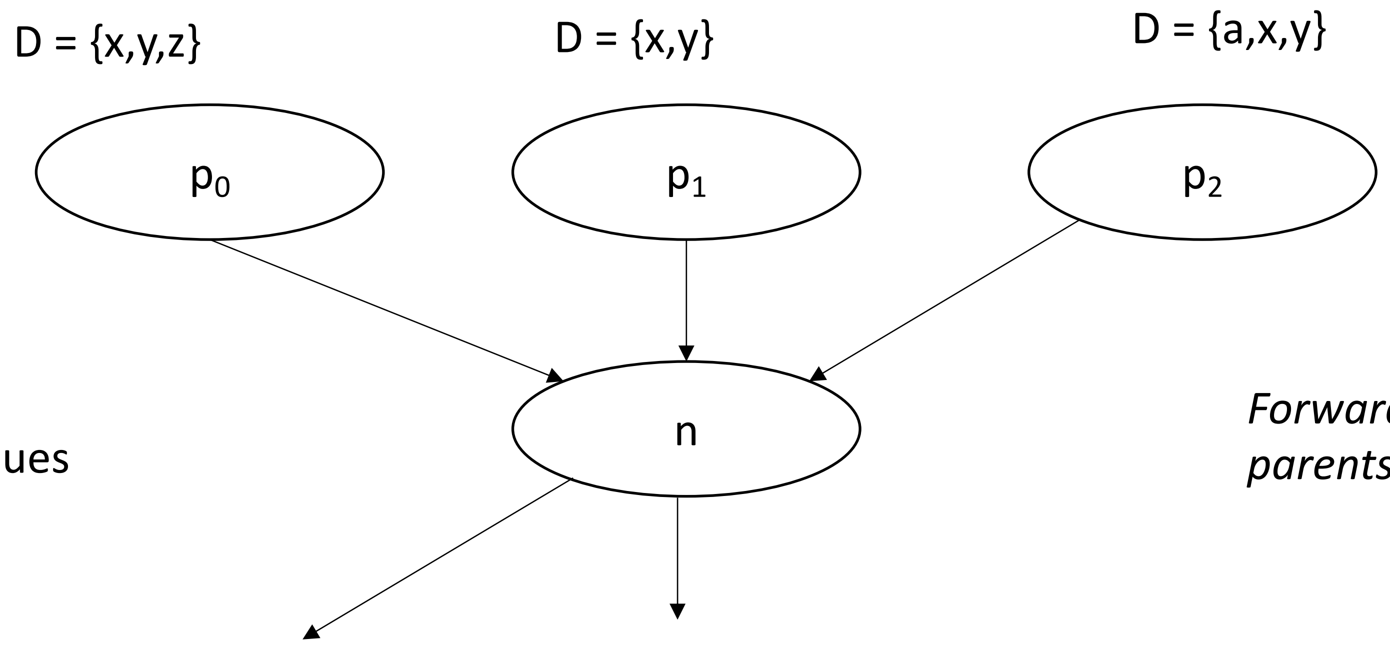
This can be any order...

How can we optimize the order?



Given this intuition, what ordering would be best?

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



update:
intersection of parent values

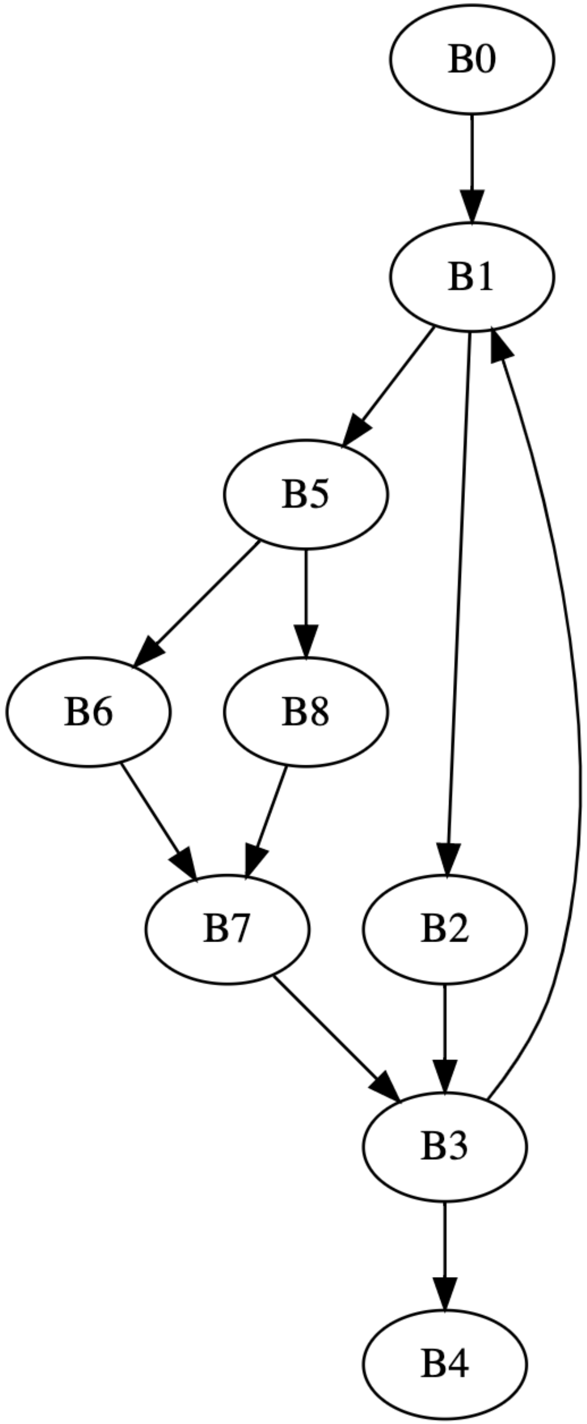
Forward flow, as updates flow from parents to children.

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

How can we optimize the algorithm?

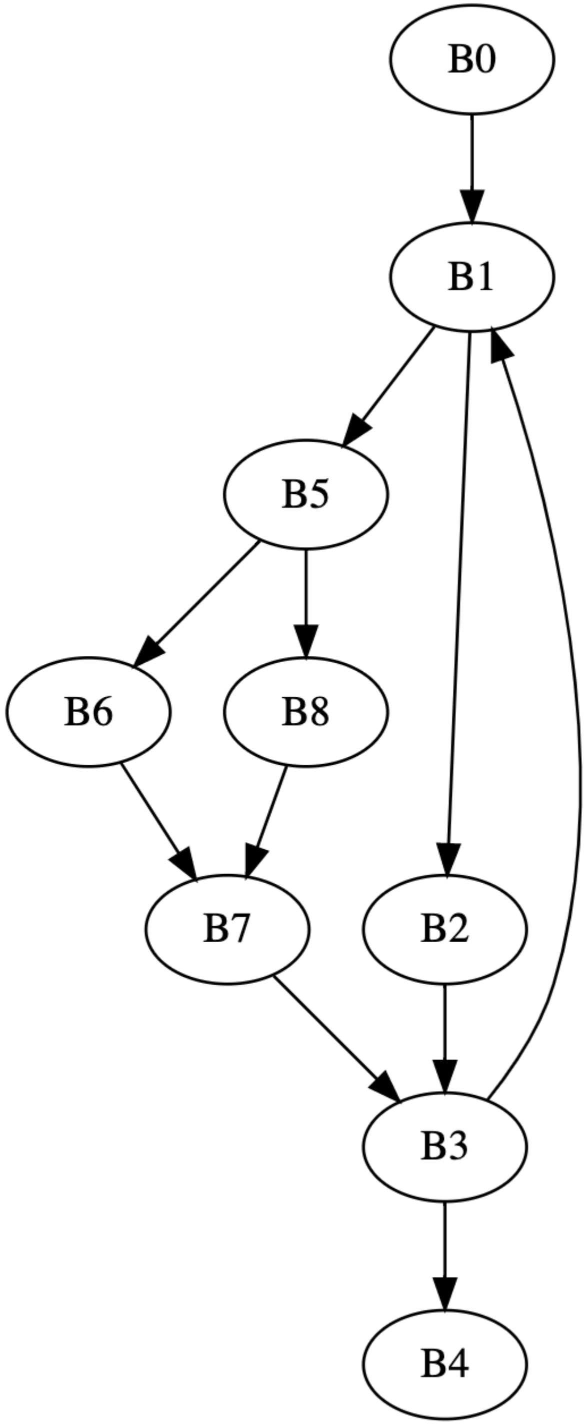
Node	New Order
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	

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



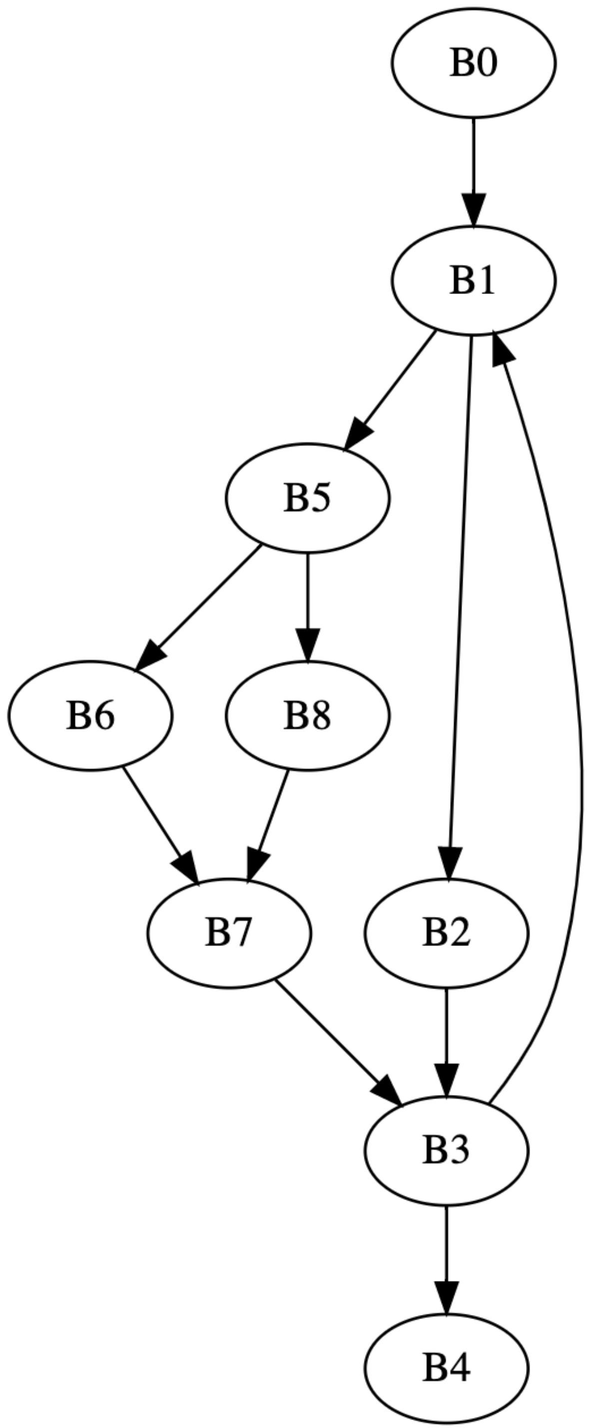
How can we optimize the algorithm?

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



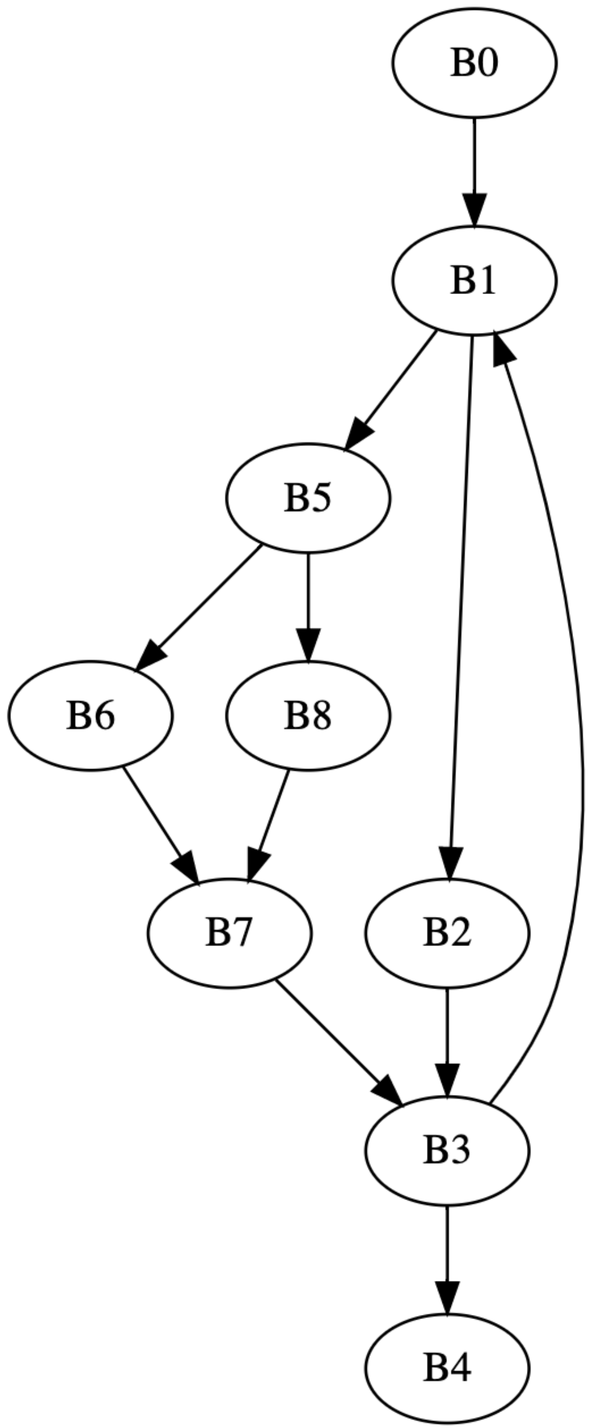
How can we optimize the algorithm?

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



How can we optimize the algorithm?

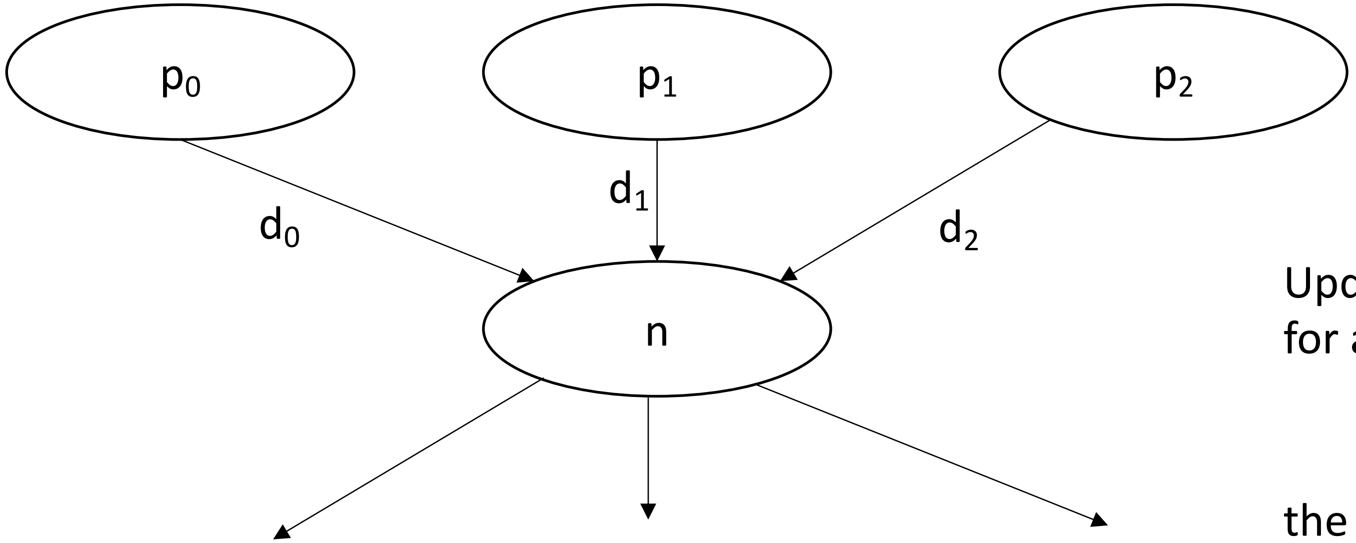
Node	Initial	Iteration 1	Iteration 2	Iteration 3
B0	B0	B0	...	
B1	*N*	B0,B1	...	
B2	*N*	B0,B1,B2	...	
B5	*N*	B0,B1,B5	...	
B6	*N*	B0,B1,B5,B6	...	
B8	*N*	B0,B1,B5,B8	...	
B7	*N*	B0,B1,B5,B7	...	
B3	*N*	B0,B1,B3	...	
B4	*N*	B0,B1,B4	...	



A quick aside about graph algorithms:

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

Traversal order in graph algorithms is a big research area!



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

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

Another analysis: Live Variable Analysis

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

- examples:

Another analysis: Live Variable Analysis

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

- examples:

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

Another analysis: Live Variable Analysis

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

- examples:

p ← Live variables: z, w

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

Another analysis: Live Variable Analysis

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

- examples:

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

p Live variables: ?

Another analysis: Live Variable Analysis

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

- examples:

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

$\xleftarrow{p}$ Live variables: x,w

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

Another analysis: Live Variable Analysis

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

- examples:

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

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

Another analysis: Live Variable Analysis

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

- examples:

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

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

```
//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)
```

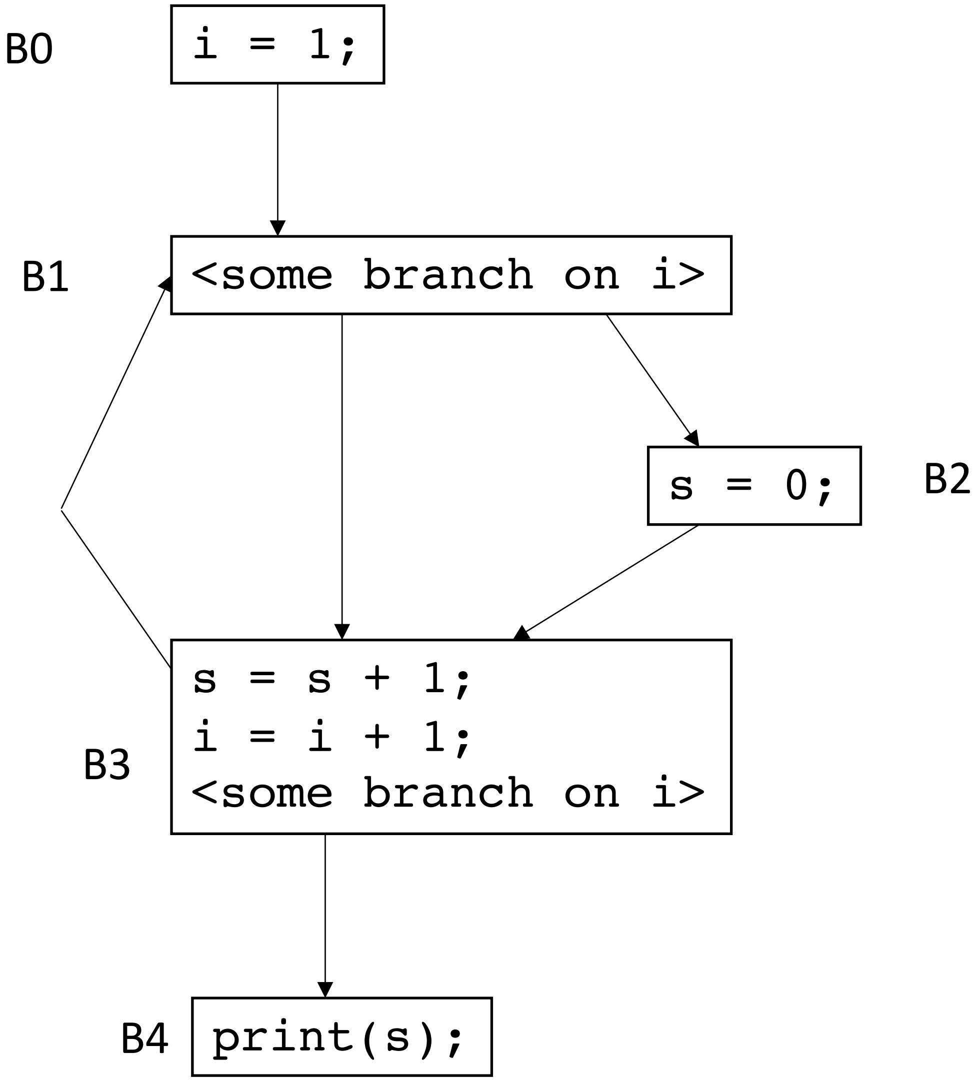
p Live variables: x

Accessing an uninitialized variable!

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

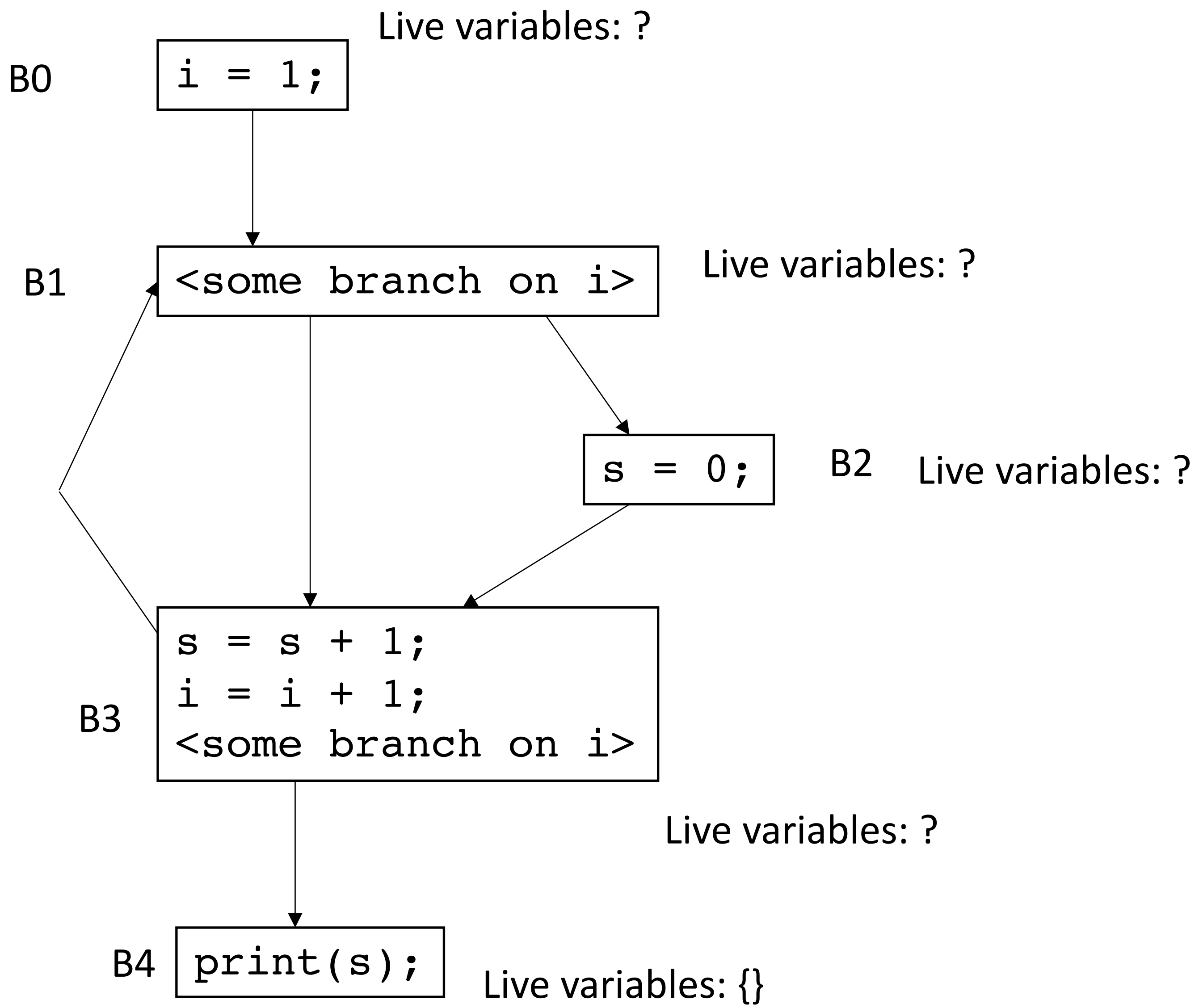
p Live variables: w

Live variable analysis in the CFG:

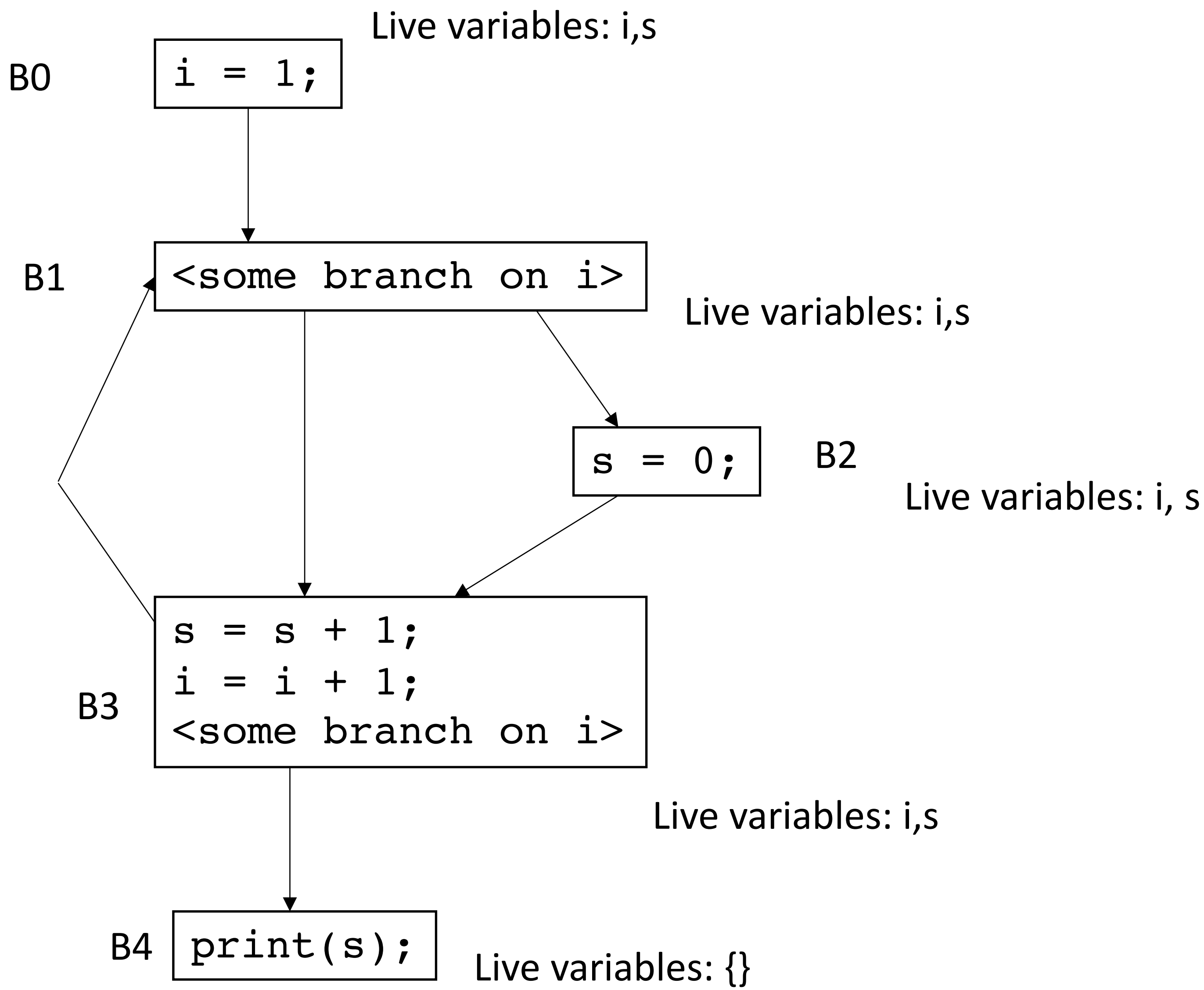


For each block B_x : we want to compute LiveOut:
The set of variables that are live at the end of B_x

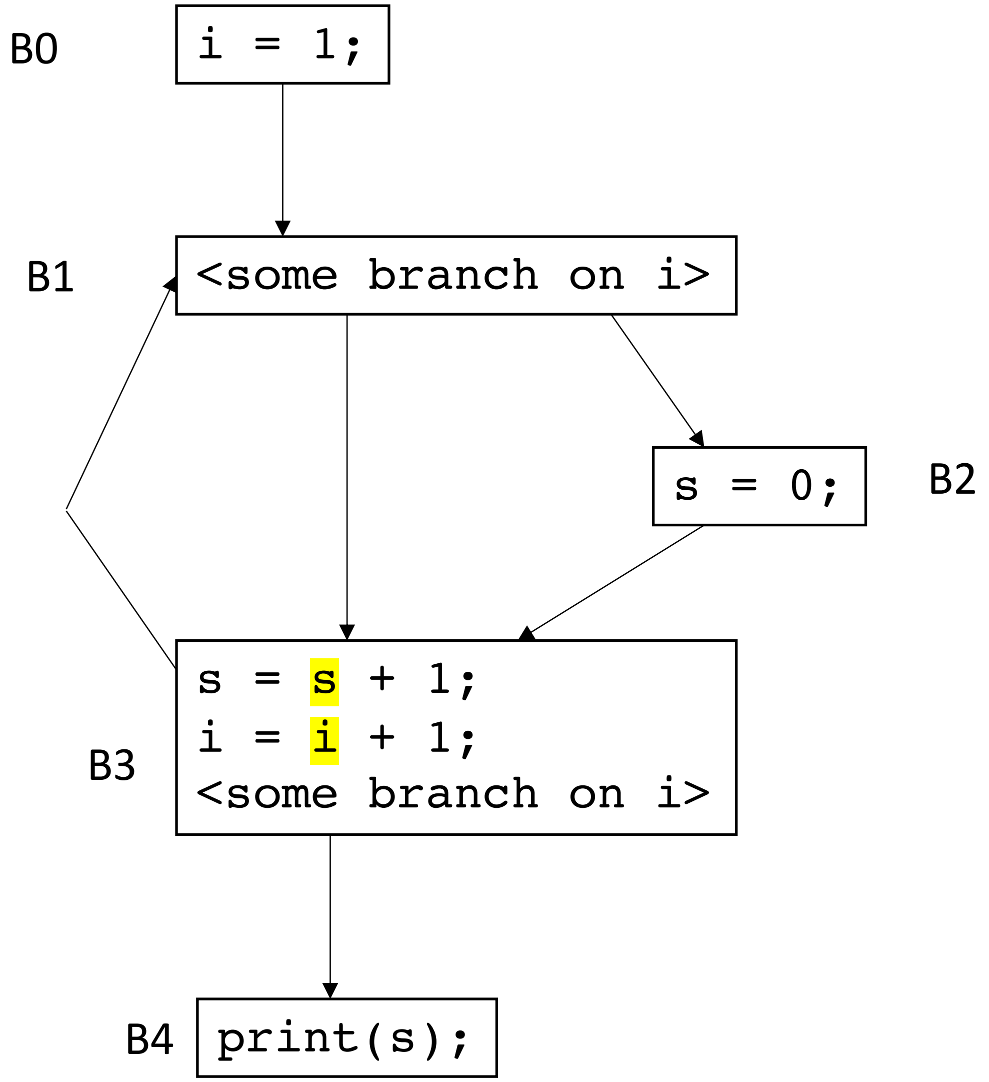
Live variable analysis in the CFG:



Live variable analysis in the CFG:

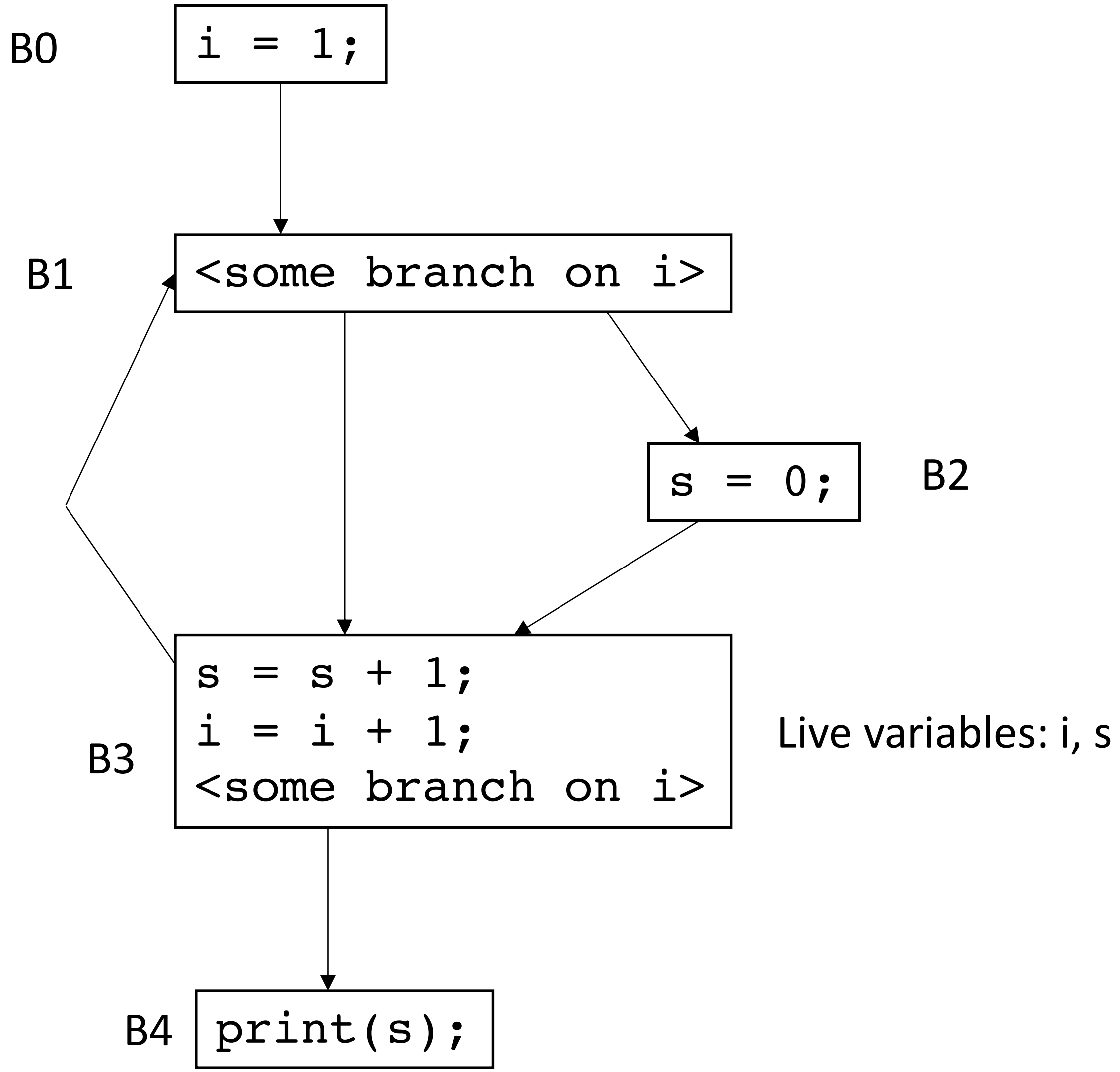


Live variable analysis in the CFG:

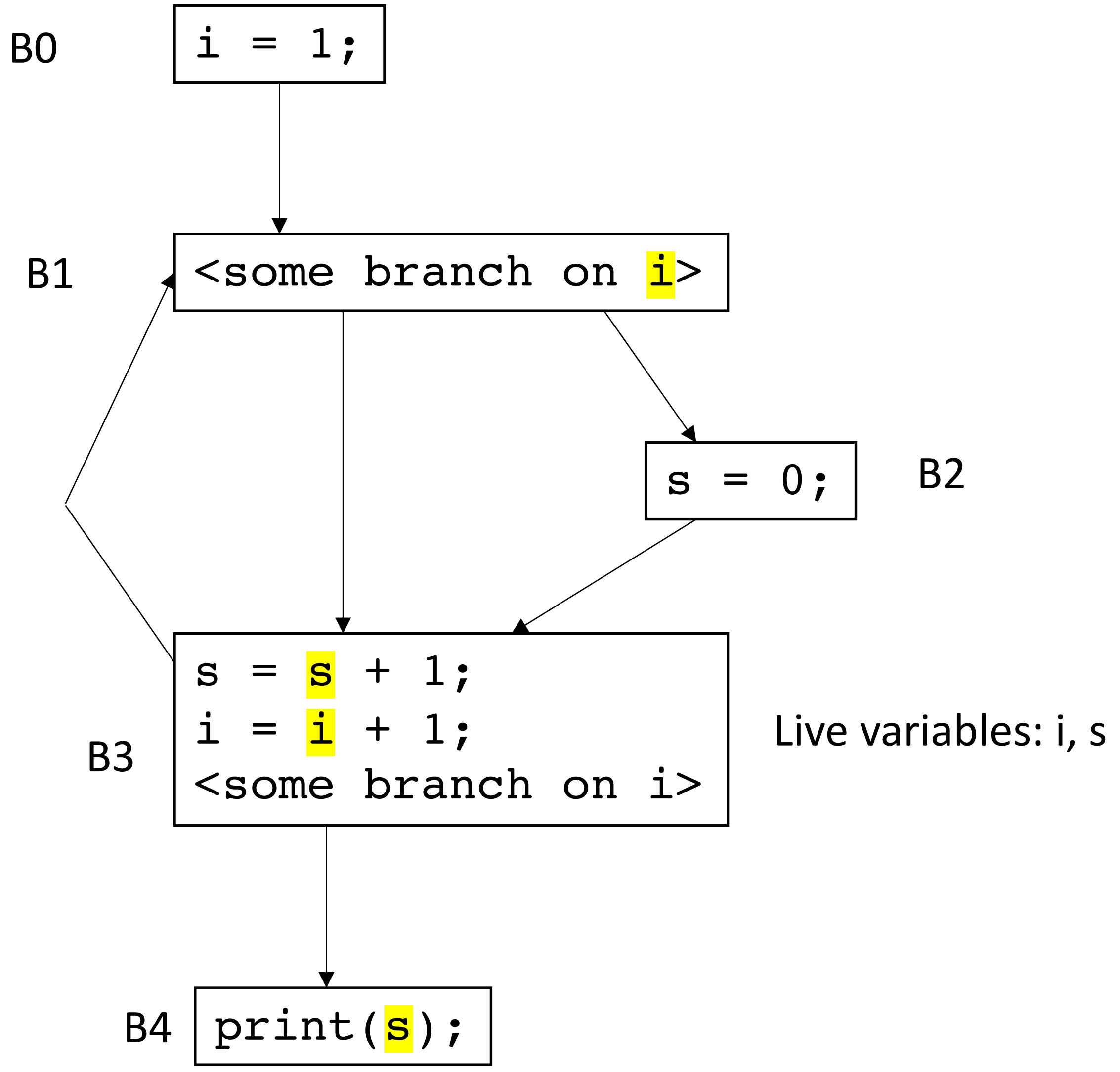


Live variables: i, s

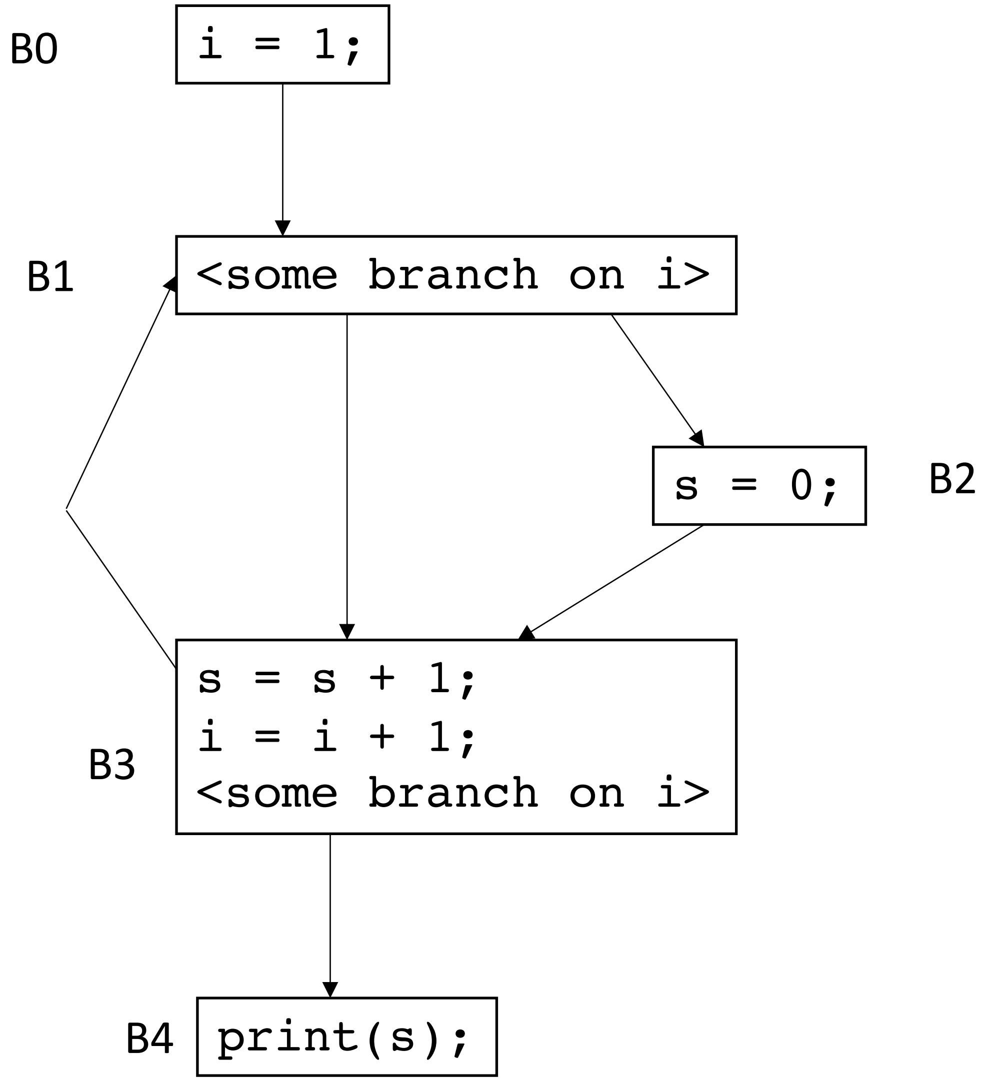
Live variable analysis in the CFG:



Live variable analysis in the CFG:



Live variable analysis in the CFG:



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

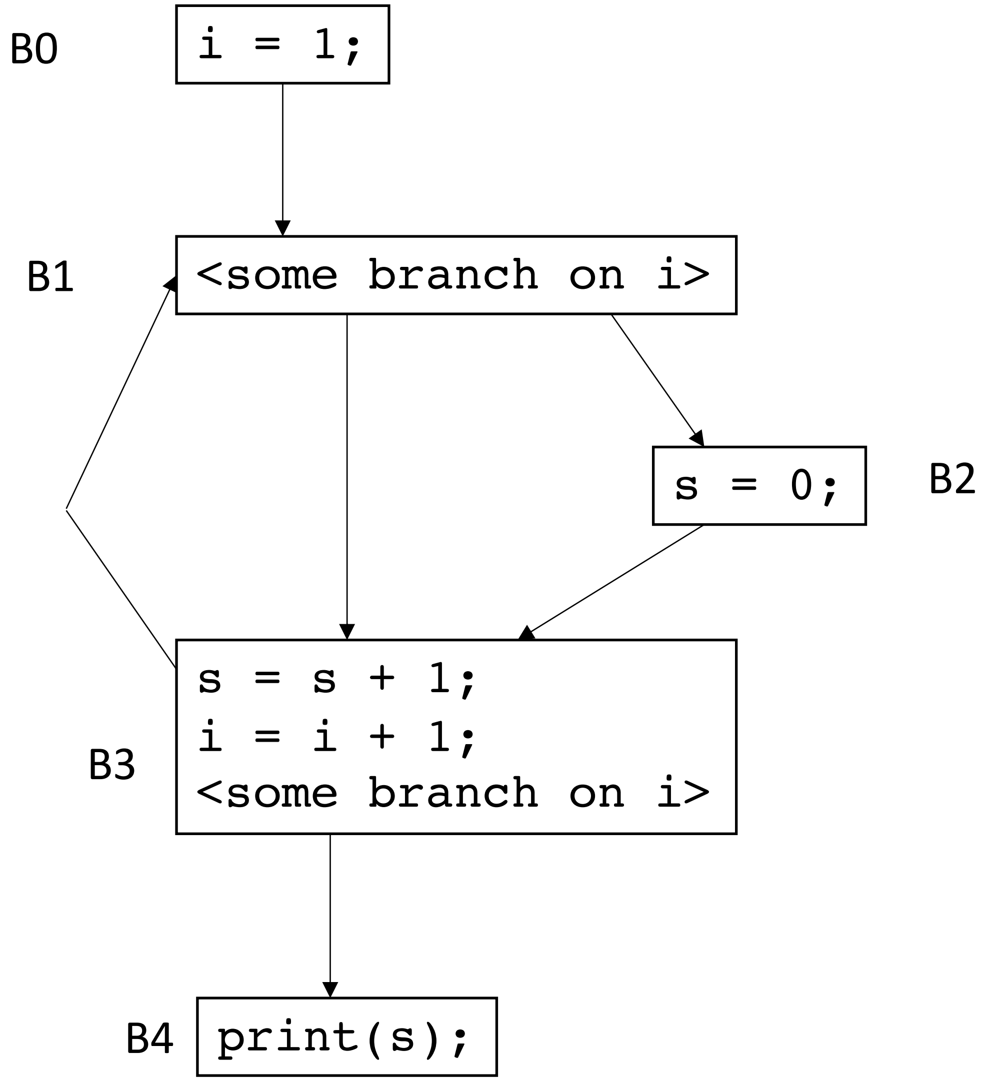
VarKill for block b is any variable in block b that gets overwritten

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

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

Block	VarKill	UEVar
B0		
B1		
B2		
B3		
B4		

Live variable analysis in the CFG:



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

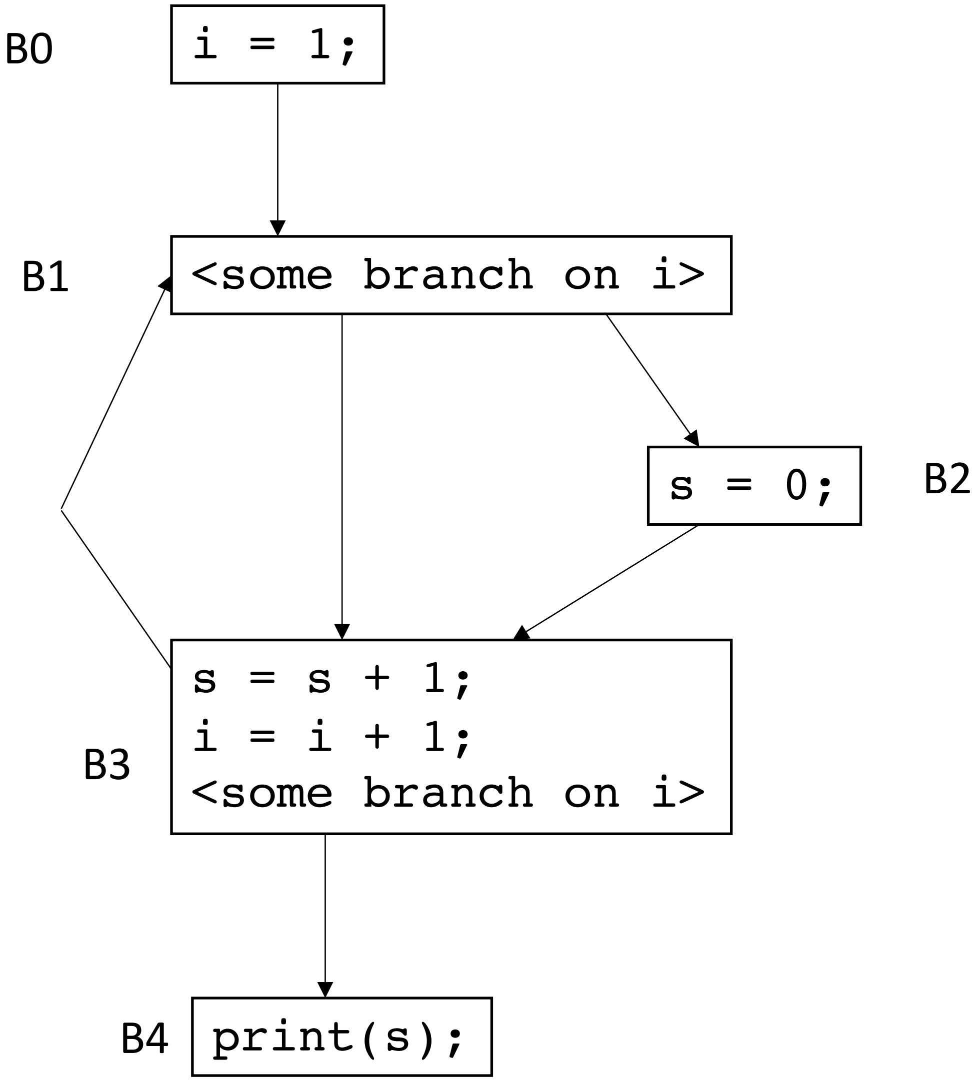
VarKill for block b is any variable in block b that gets overwritten

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

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

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

Live variable analysis in the CFG:



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

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

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

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

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

Live variable analysis in the CFG:

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

Live variable analysis in the CFG:

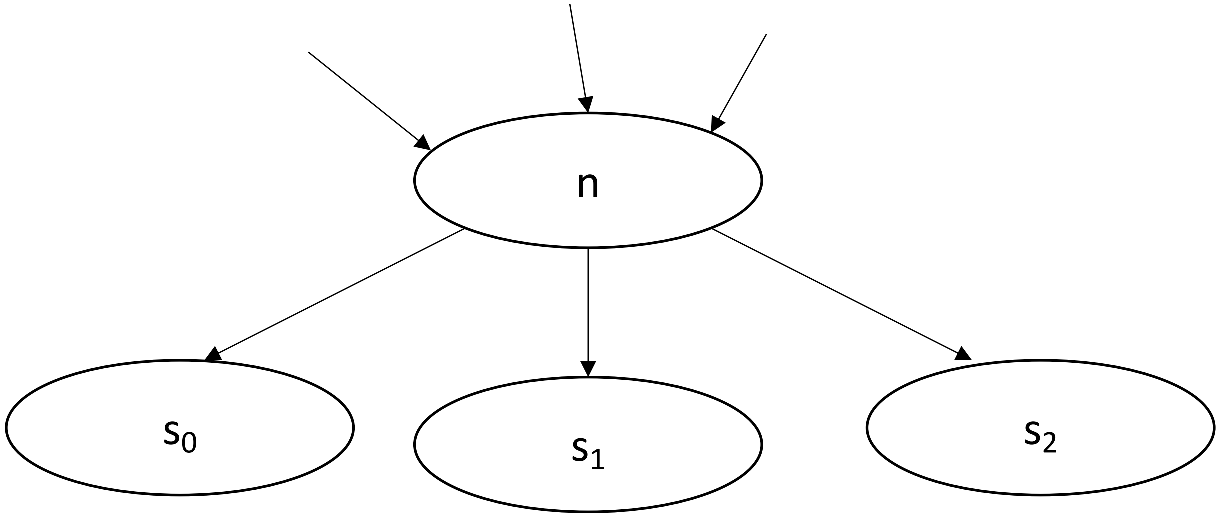
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e. end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed point computation:

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

Live variable analysis in the CFG:

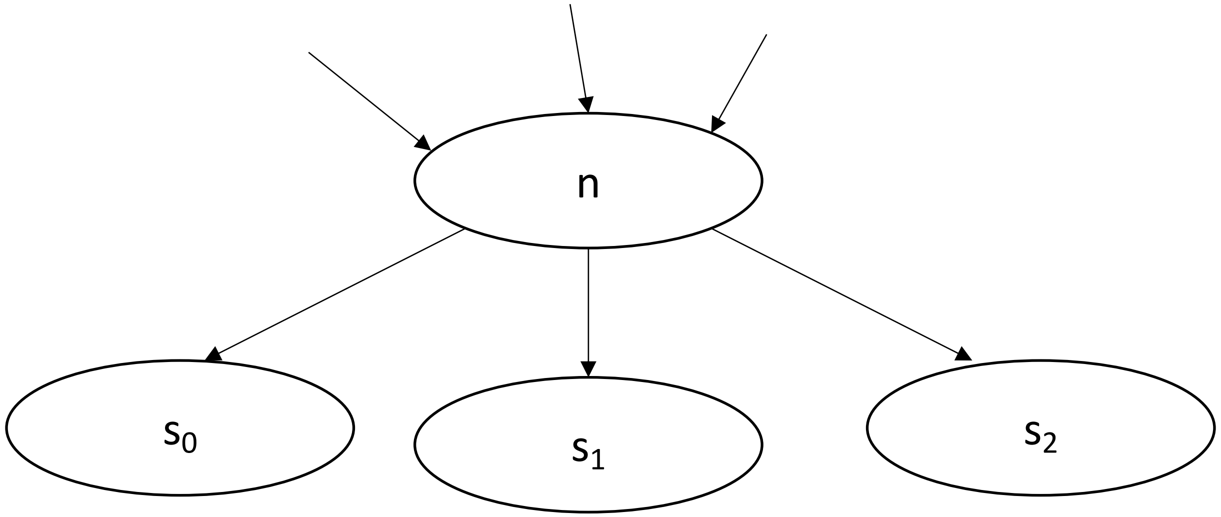
$$\mathit{LiveOut}(n) = \cup_{s \text{ in } succ(n)} \left(\mathit{UEVar}(s) \cup (\mathrm{LiveOut}(s) \cap \overline{\mathit{VarKill}(s)}) \right)$$



Backwards flow analysis because values flow from successors

Live variable analysis in the CFG:

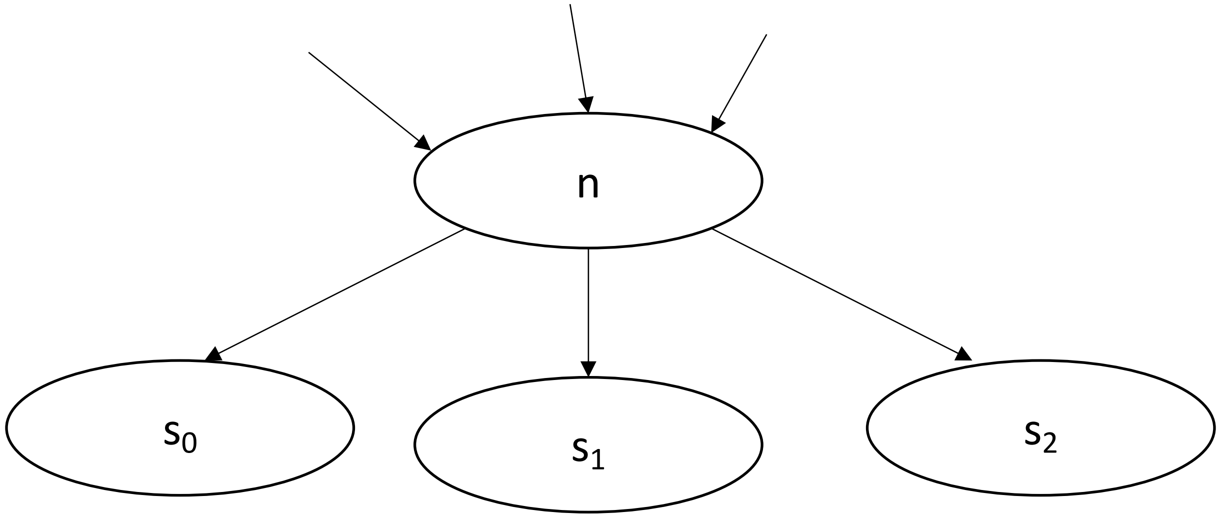
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



any variable in UEVar(s) is live at n

Live variable analysis in the CFG:

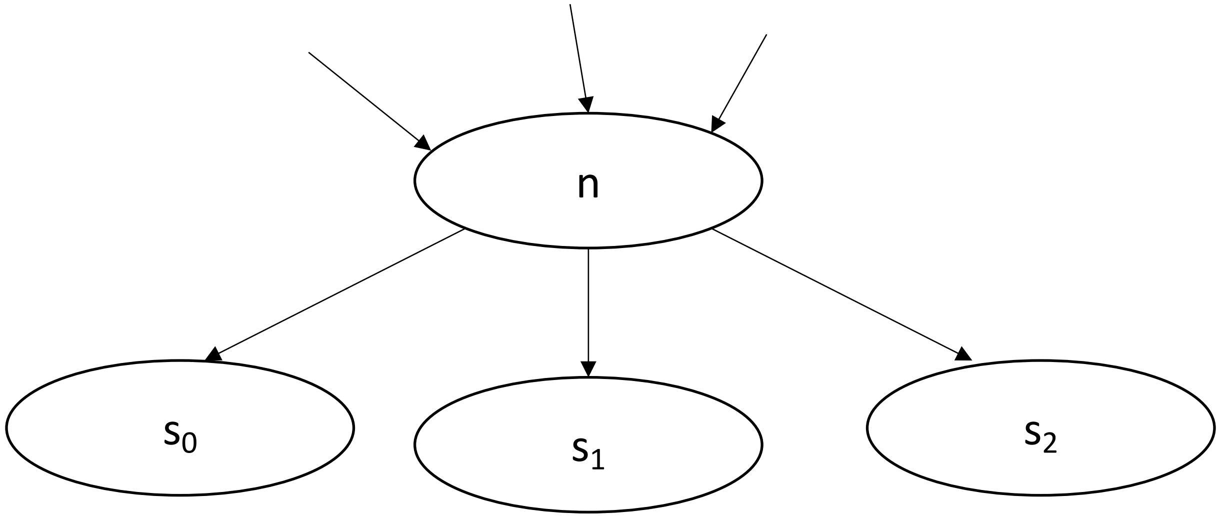
$$\mathit{LiveOut(n)} = \bigcup_{\text{s in succ(n)}} (\mathit{UEVar(s)} \cup (\text{LiveOut(s)} \cap \overline{\mathit{VarKill(s)}}))$$



variables that are not overwritten in s

Live variable analysis in the CFG:

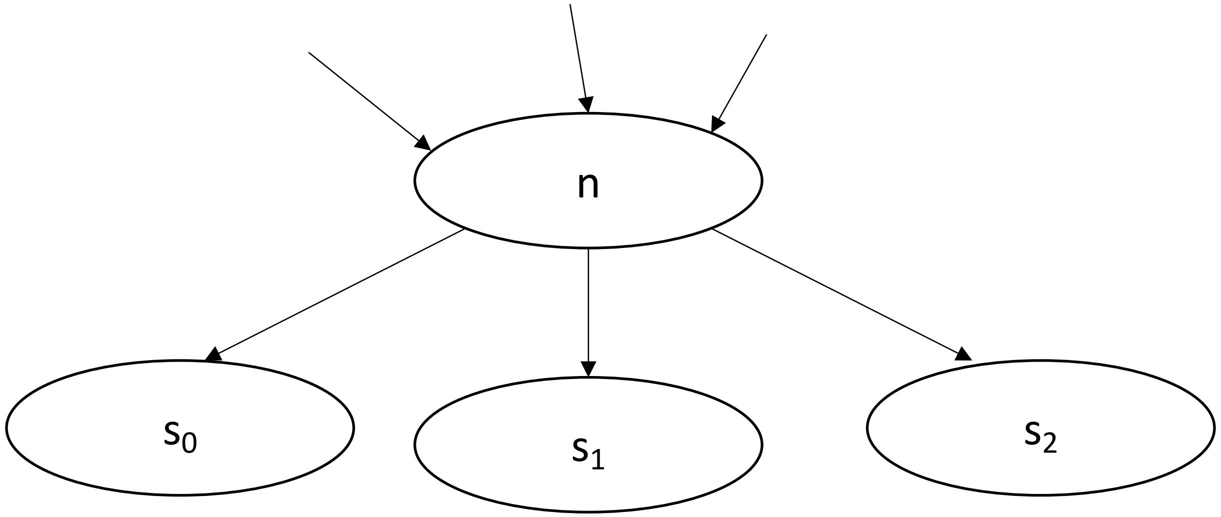
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)}\right)\right)$$



variables that are live at the end of s

Live variable analysis in the CFG:

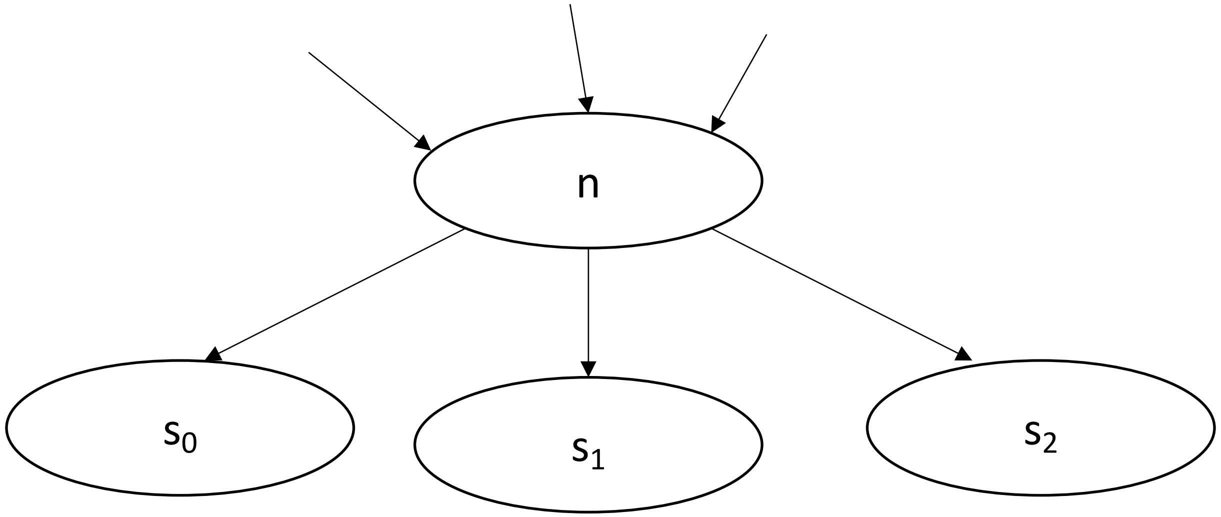
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}\,\right)\right)$$



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

Live variable analysis in the CFG:

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



LiveOut is a union
rather than an intersection

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

Consider the language we use for each:

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

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

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

Consider the language we use for each:

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

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

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

Have a nice weekend!

- We will discuss other flow algorithms
- Remember, homework 1 is due on Tuesday