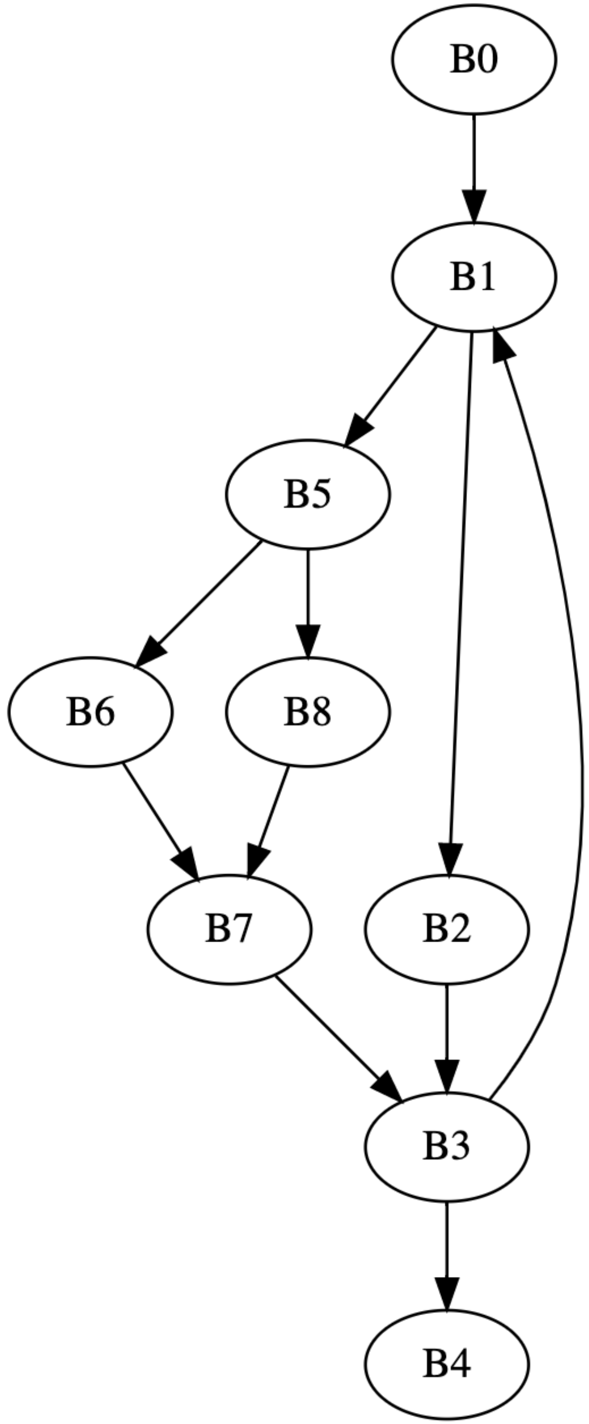
CSE211: Compiler Design

Oct. 27, 2023

- **Topic**: global optimizations continuted
- **Questions**:
  - *What is a fixed point iteration?*
  - *How can we speed up fixed point iteration algorithms?*



Announcements

- Homework 1 was due on Wednesday

- Homework 2 is out
  - Fixed the links
  - I noticed that I did give some scheduling flexibility

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

Announcements

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

Review

Global optimizations

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

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

First concept:

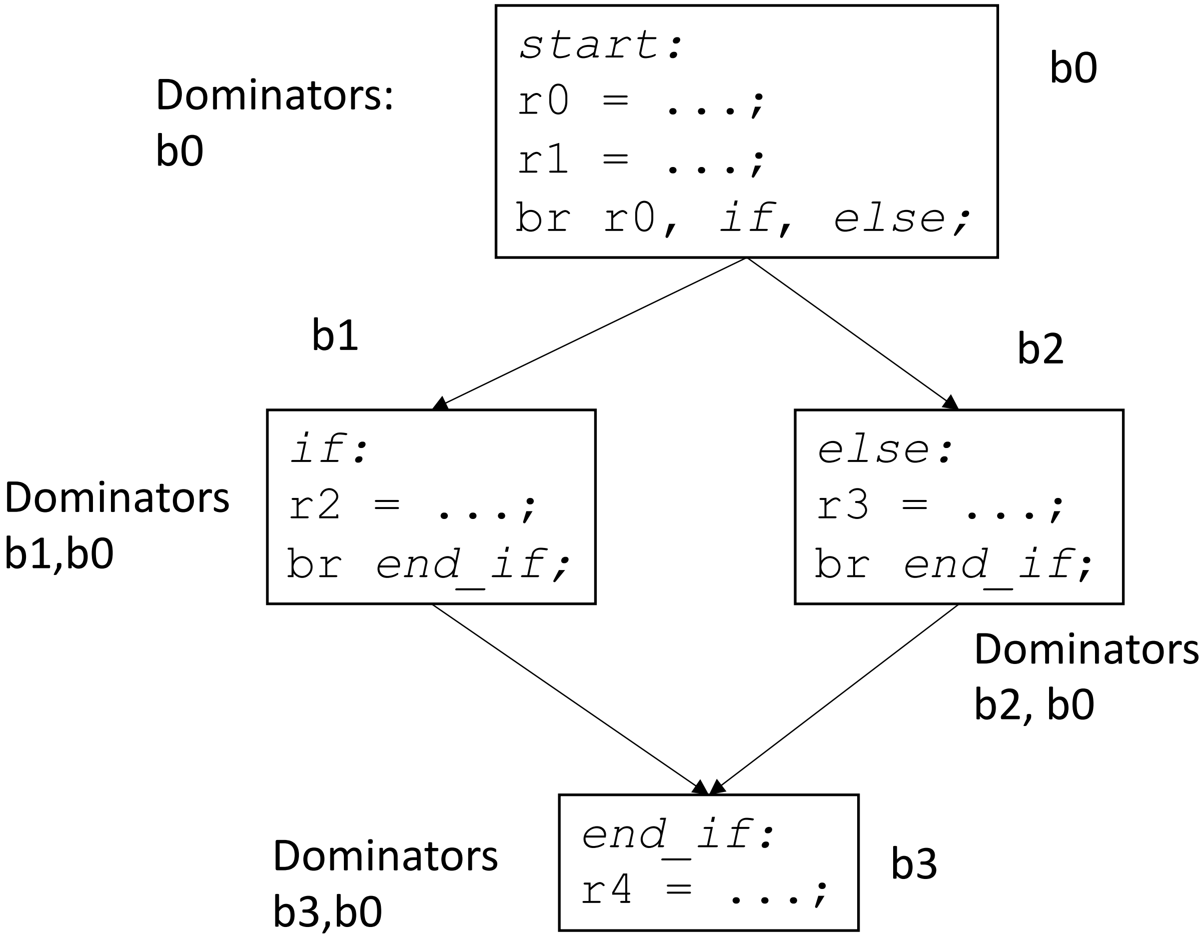
- Dominance in a CFG
- Builds up a framework for reasoning
- Building block for many algorithms
  - global local value numbering when unlimited registers
  - Conversion to SSA

Dominance

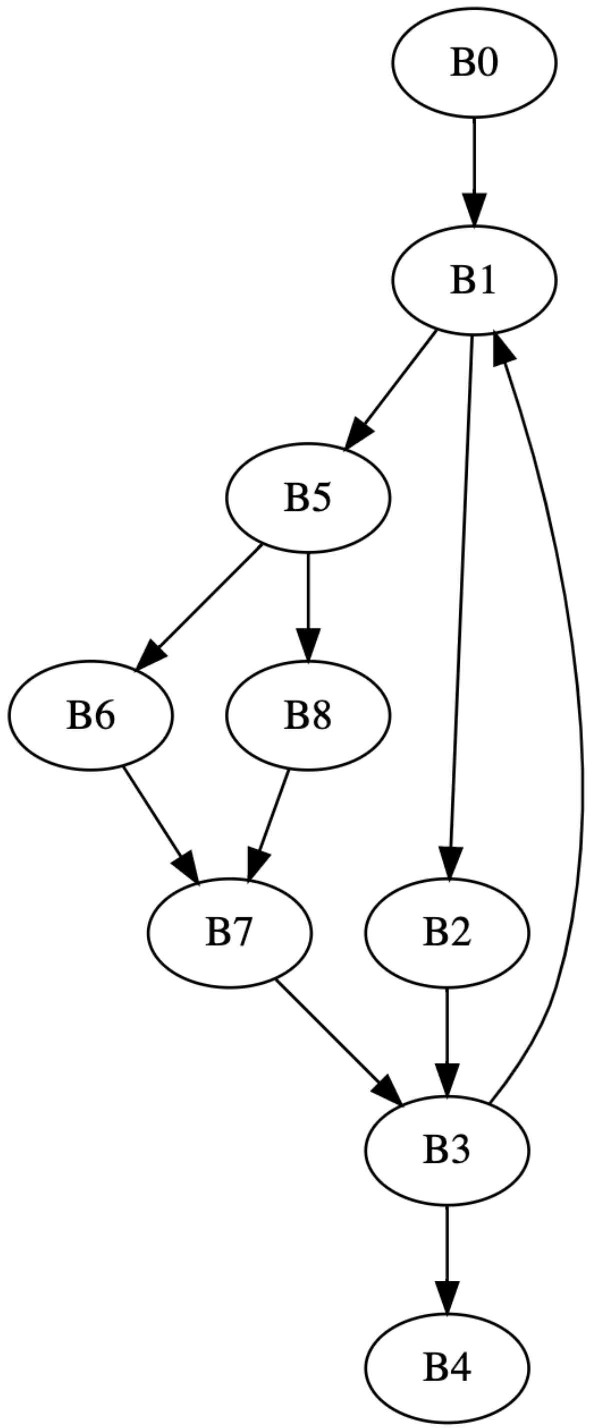
- a block b_x dominates block b_y if every path from the start to block b_y goes through b_x

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

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



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

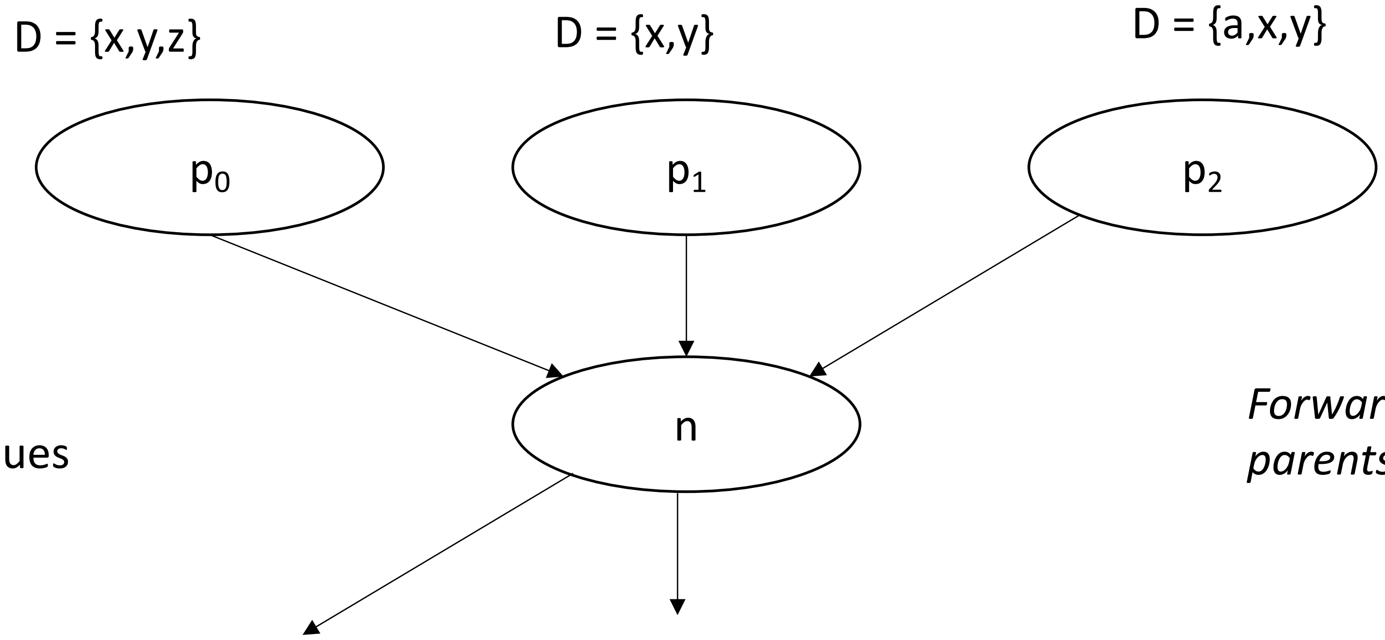
- Initial state, all nodes start with all other nodes are dominators:
  - *Dom(n) = N*
  - *Dom(start) = {start}*

iteratively compute:

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

Now lets think about dominance

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



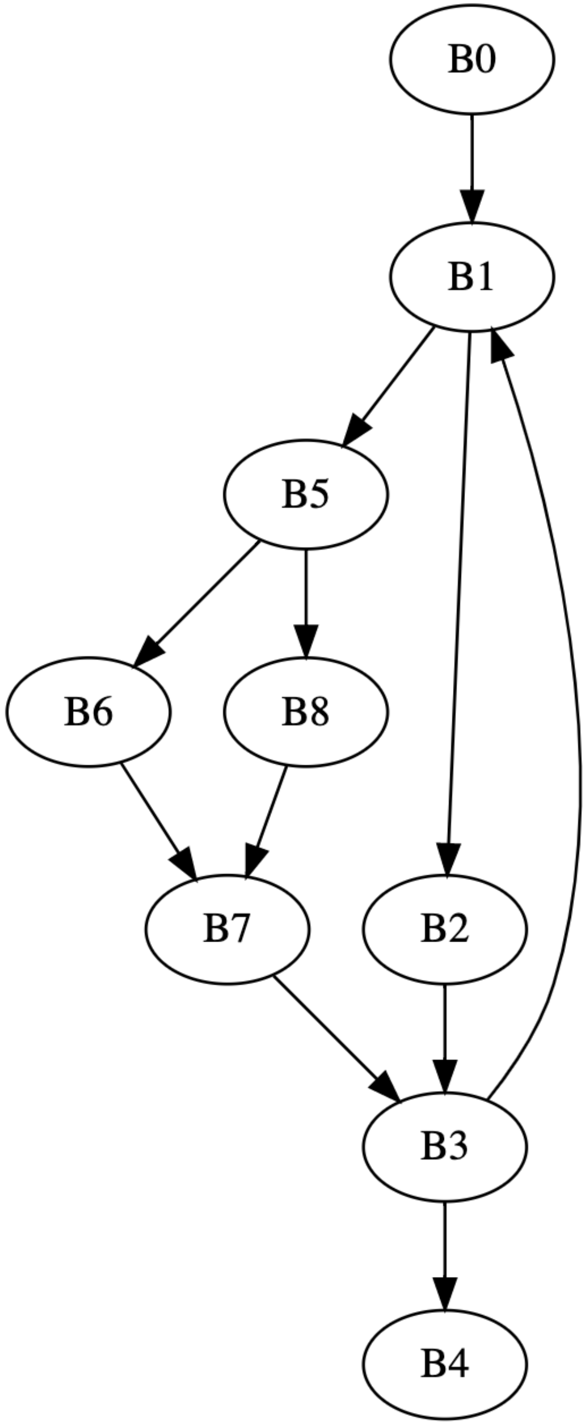
update:
intersection of parent values

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	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	...	



New Material

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

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

Another analysis: Live Variable Analysis

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

- examples:

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

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

Another analysis: Live Variable Analysis

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

- examples:

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

p

Live variables: x,w

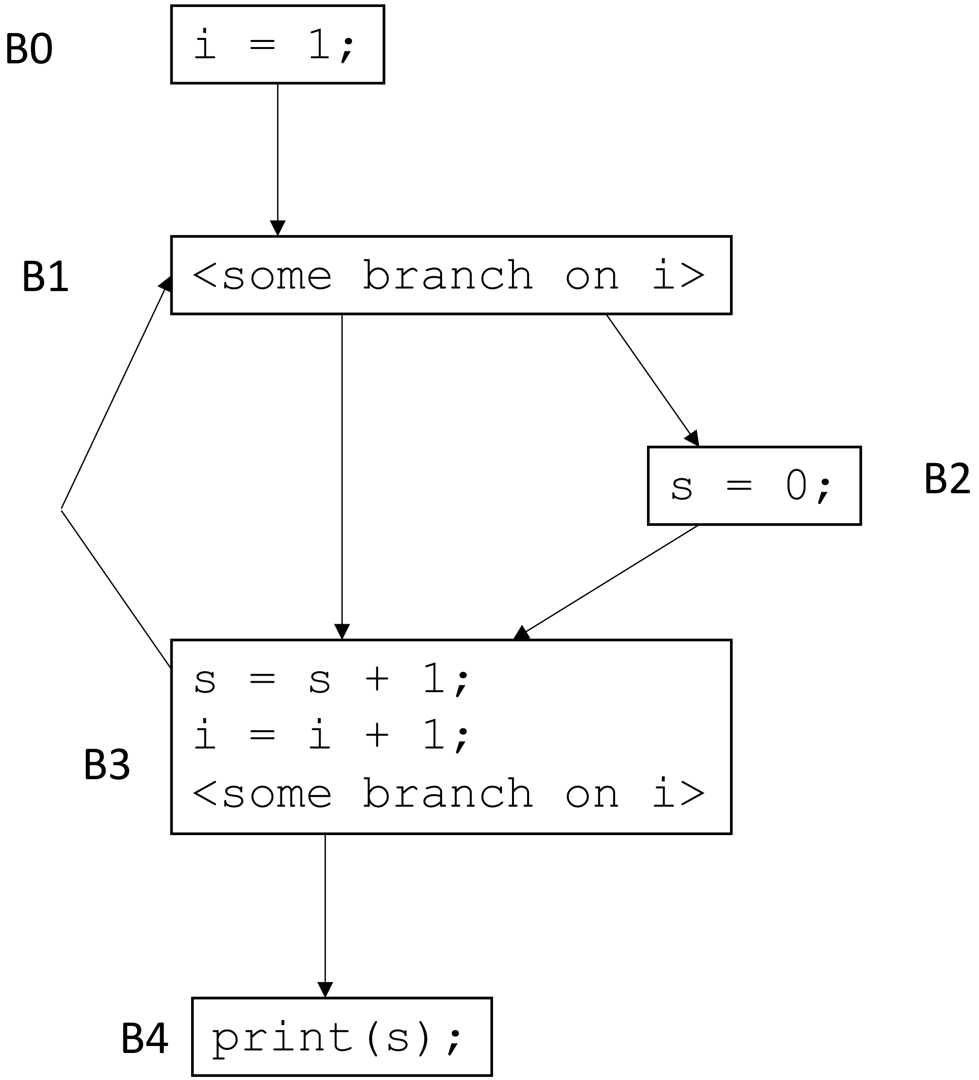
Accessing an uninitialized variable!

p

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

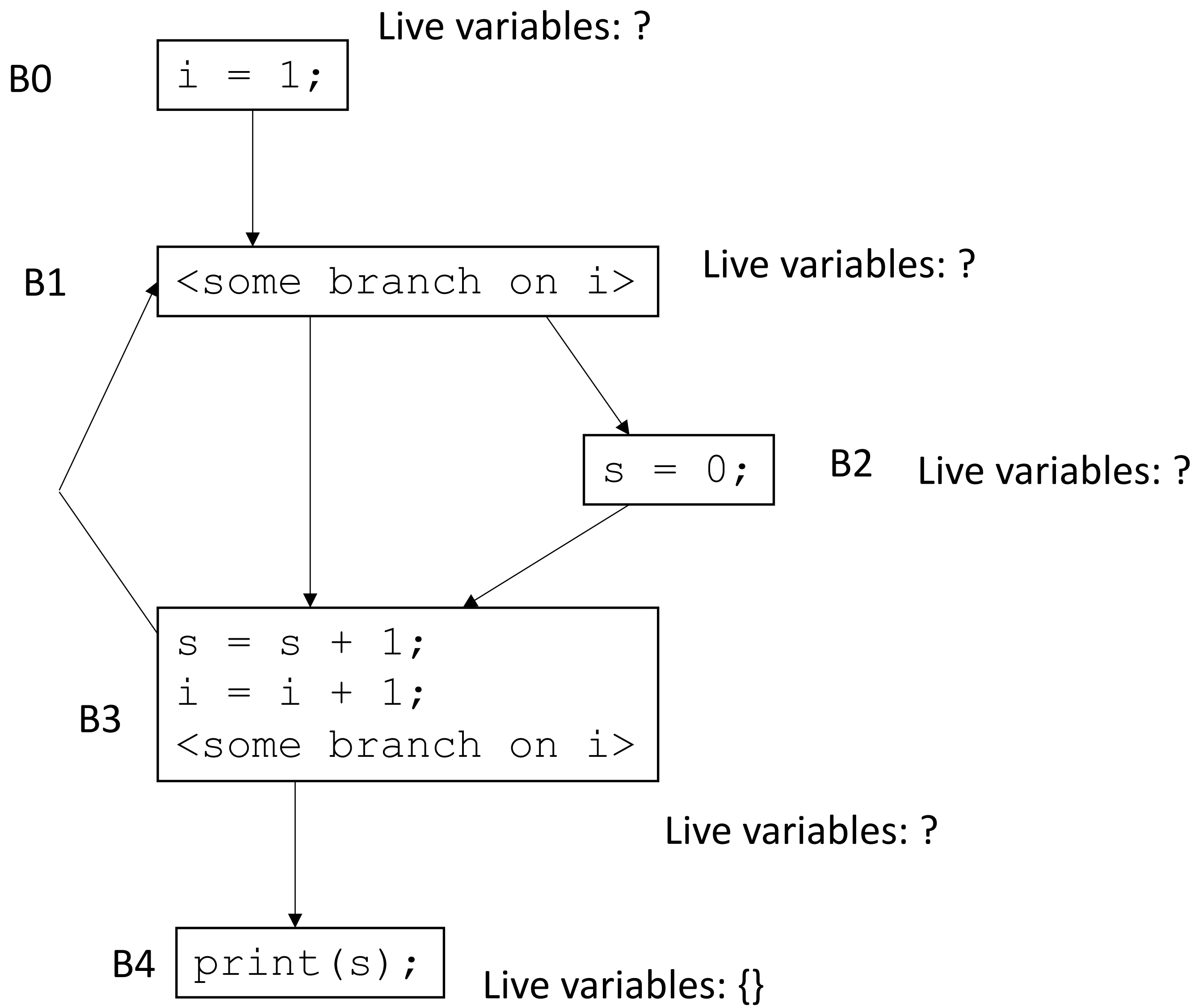
Live variables: w,z

Live variable analysis in the CFG:

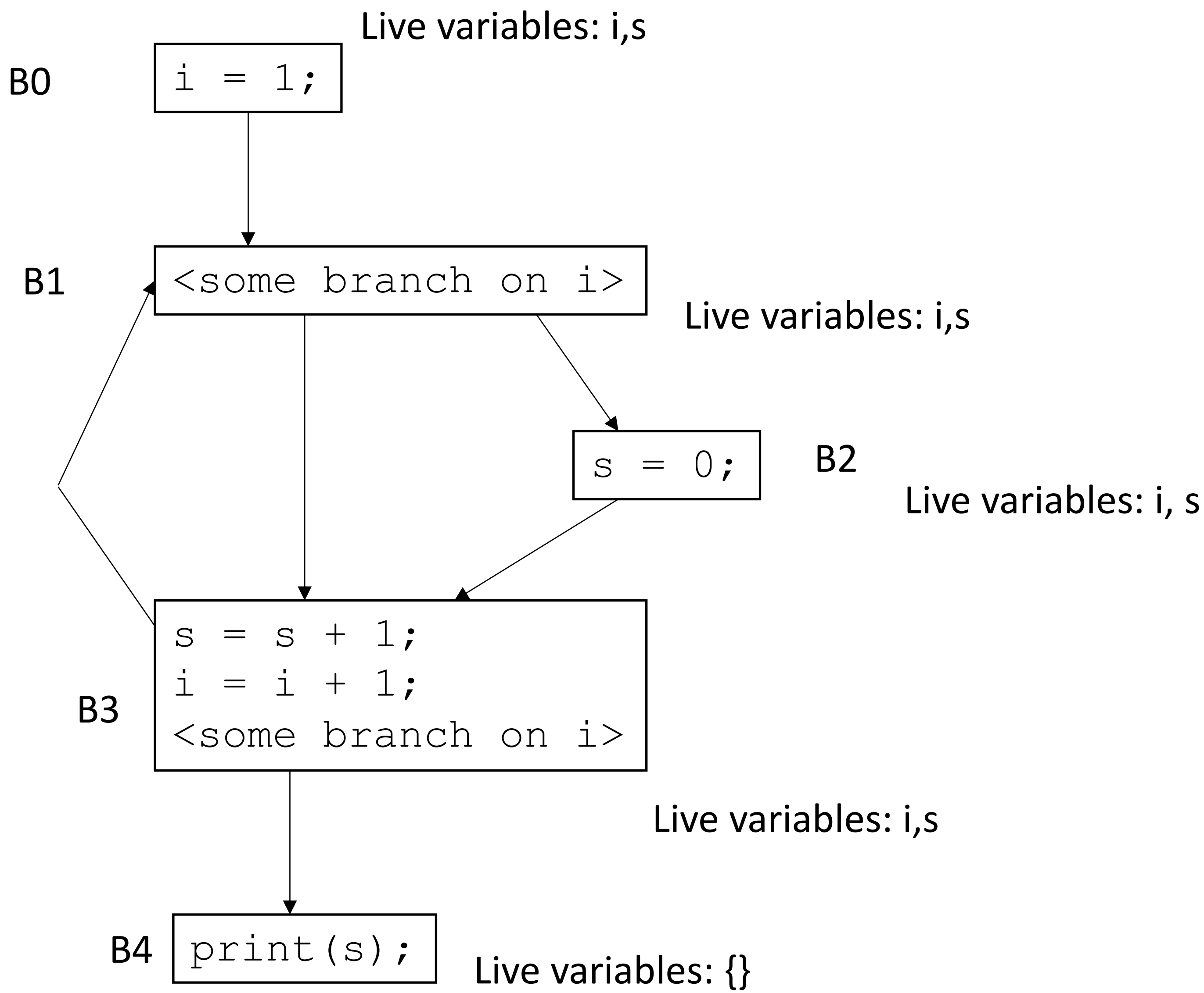


For each block B_x : we want to compute LiveOut:
The set of variables that are live at the end of B_x

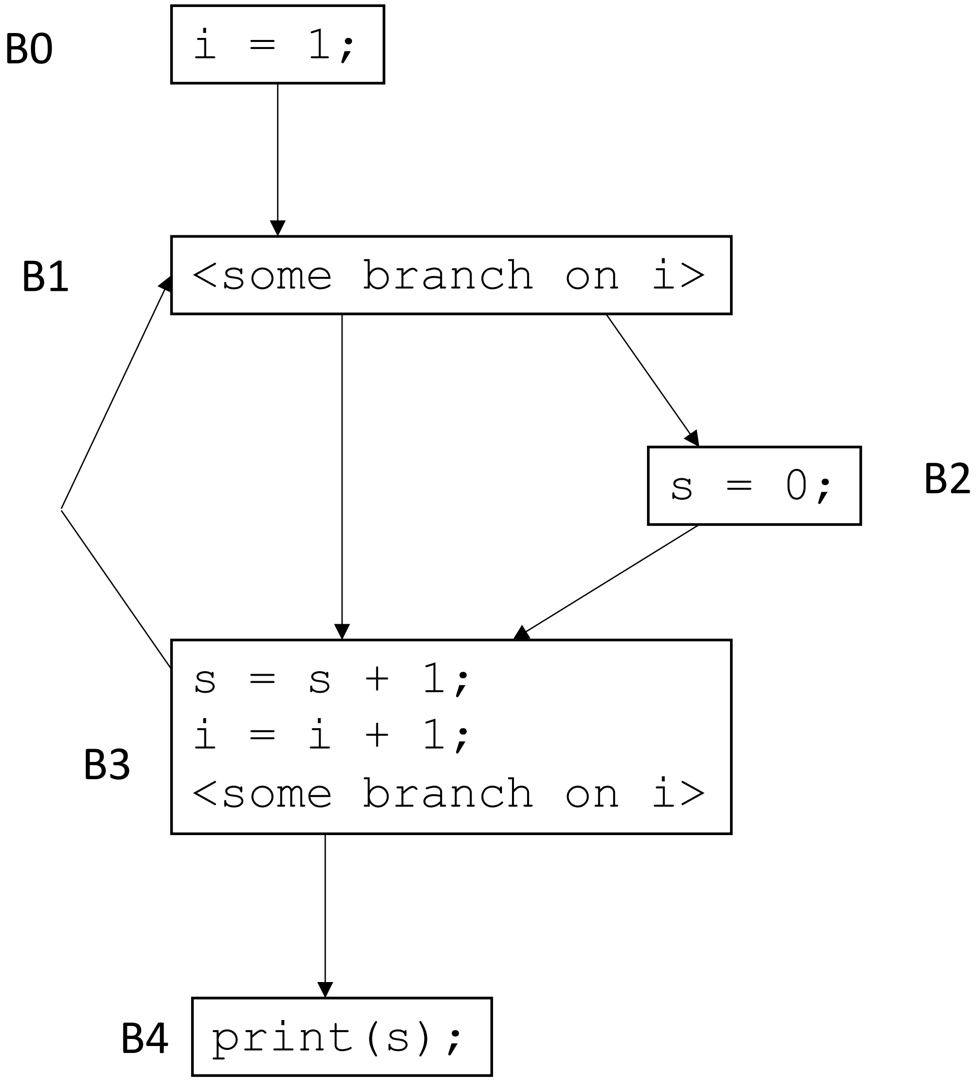
Live variable analysis in the CFG:



Live variable analysis in the CFG:



Live variable analysis in the CFG:



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

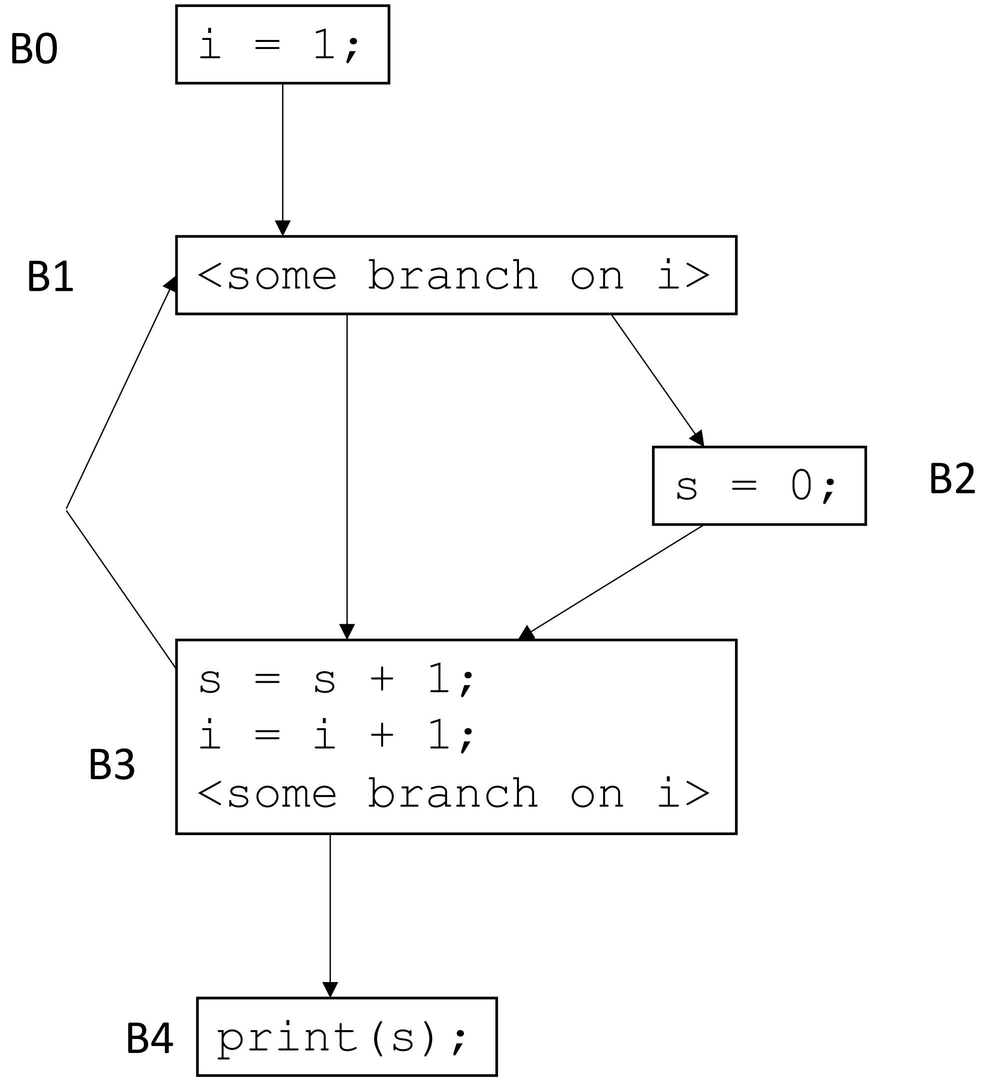
VarKill for block b is any variable in block b that gets overwritten

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

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

Block	VarKill	UEVar
B0		
B1		
B2		
B3		
B4		

Live variable analysis in the CFG:



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

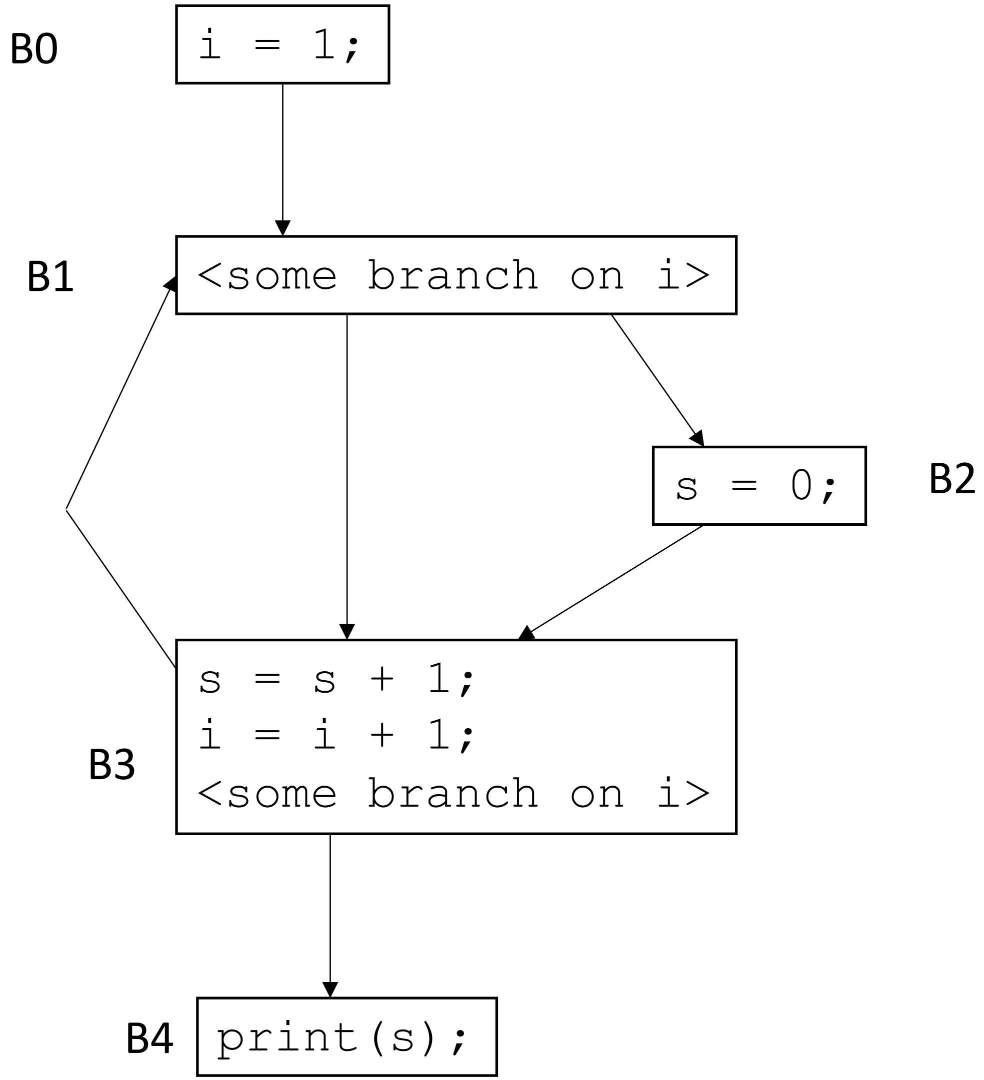
VarKill for block b is any variable in block b that gets overwritten

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

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

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

Live variable analysis in the CFG:



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

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

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

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

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

Live variable analysis in the CFG:

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

Live variable analysis in the CFG:

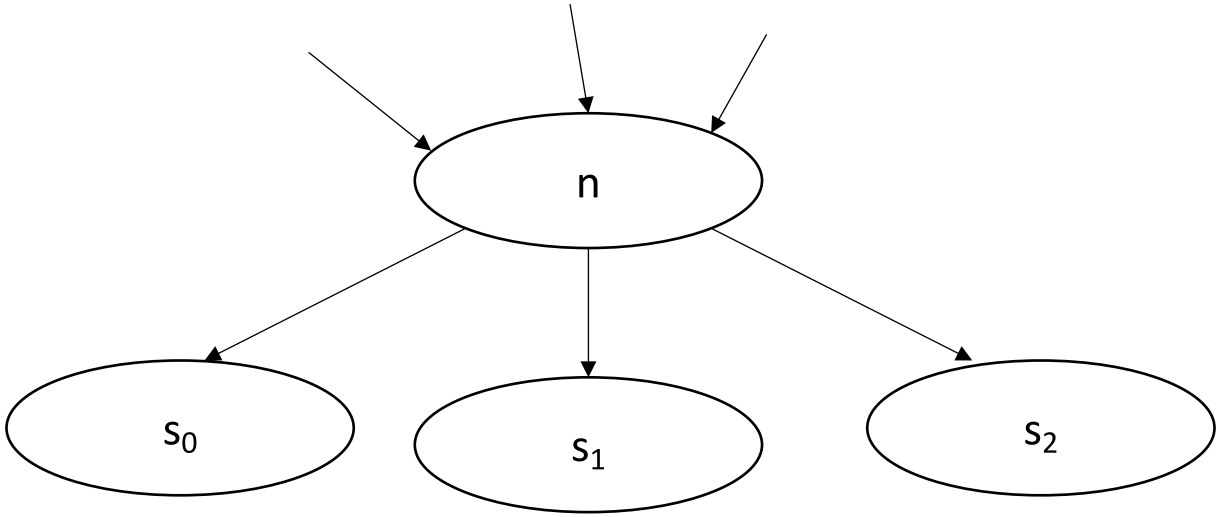
- Initial condition: LiveOut(n) = {} for all nodes
  - Ground truth, no variables are live at the exit of the program, i.e., end node n_{end} has LiveOut(n_{end})= {}

Now we can perform the iterative fixed-point computation:

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

Live variable analysis in the CFG:

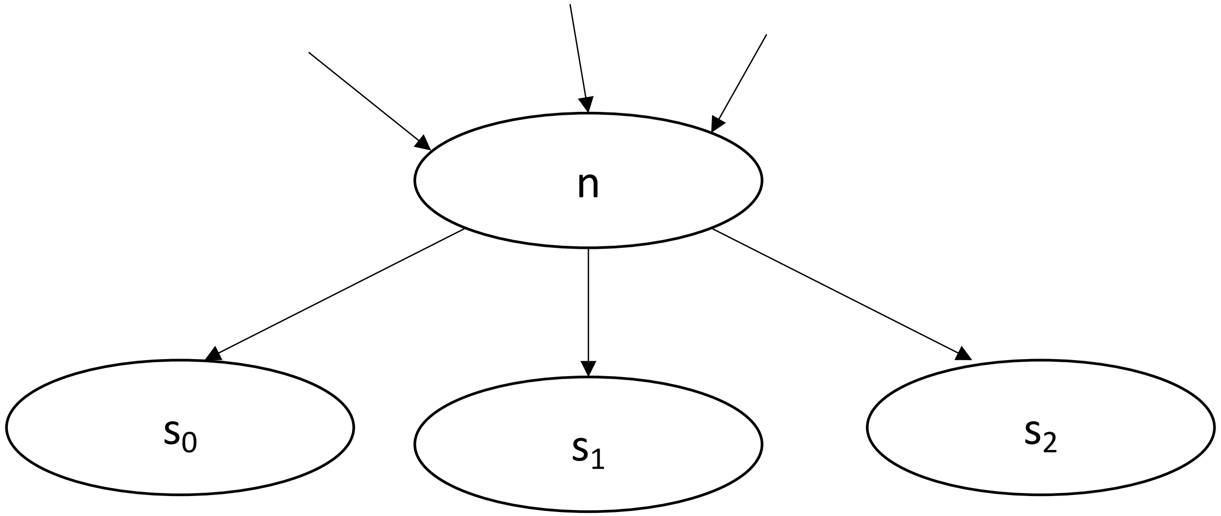
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\mathrm{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$



Backwards flow analysis because values flow from successors

Live variable analysis in the CFG:

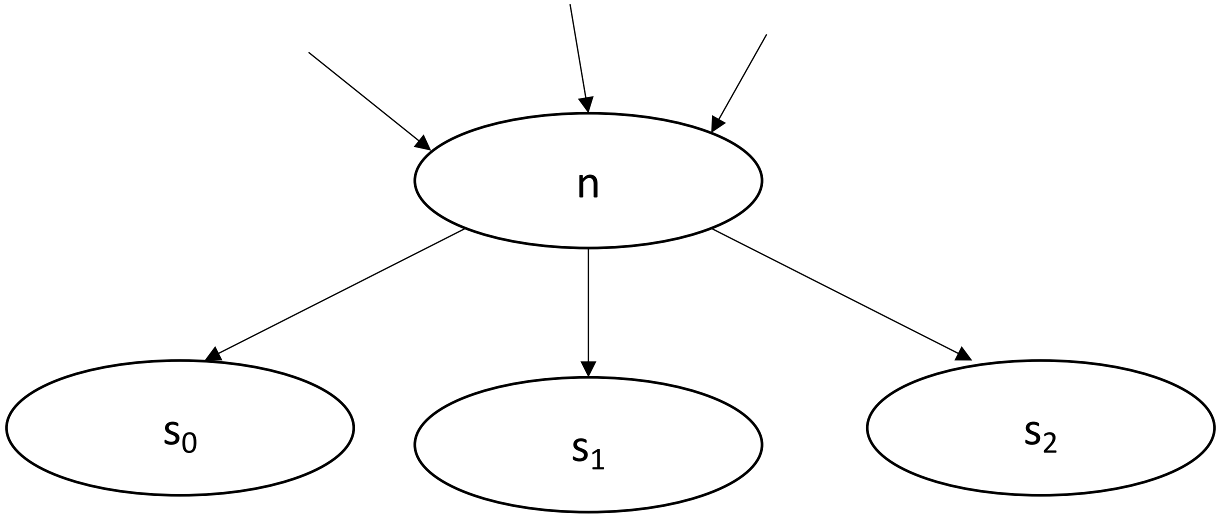
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$



any variable in UEVar(s)
is live at n

Live variable analysis in the CFG:

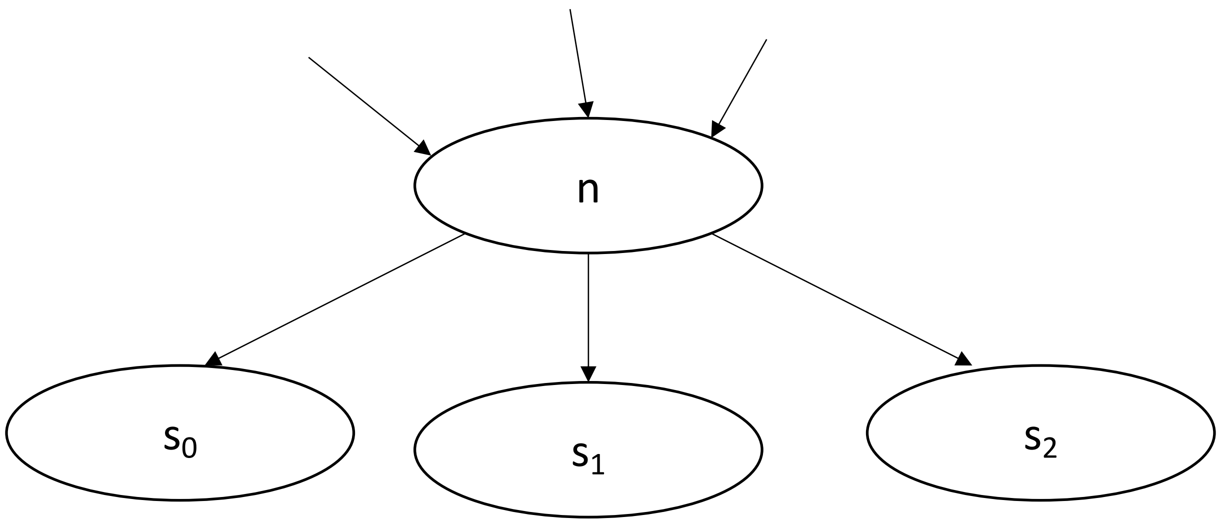
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\mathrm{LiveOut}(s) \cap \overline{VarKill(s)} \,) \right)$$



variables that are not overwritten in s

Live variable analysis in the CFG:

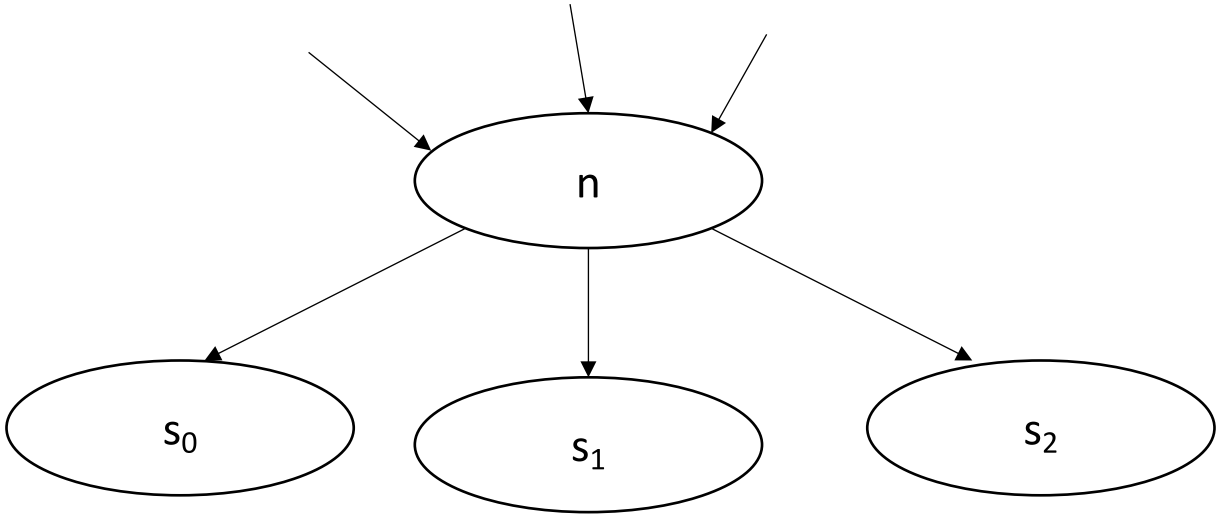
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\mathrm{LiveOut(s)} \cap \overline{VarKill(s)}\,\right)\right)$$



variables that are live
at the end of s

Live variable analysis in the CFG:

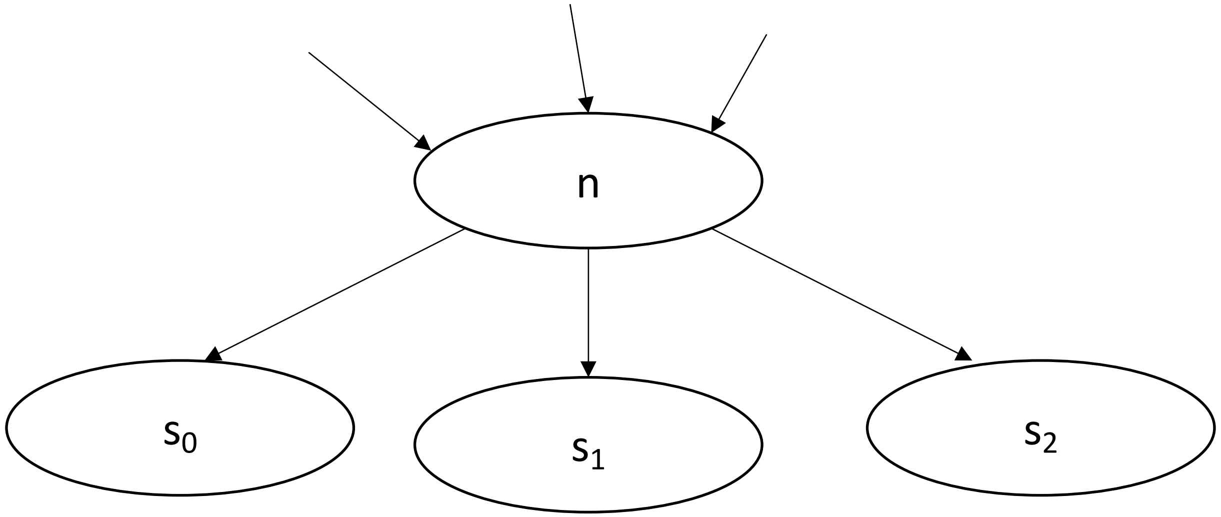
$$LiveOut(n) = \bigcup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(\text{LiveOut}(s) \cap \overline{VarKill(s)}\right)\right)$$



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

Live variable analysis in the CFG:

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



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*

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

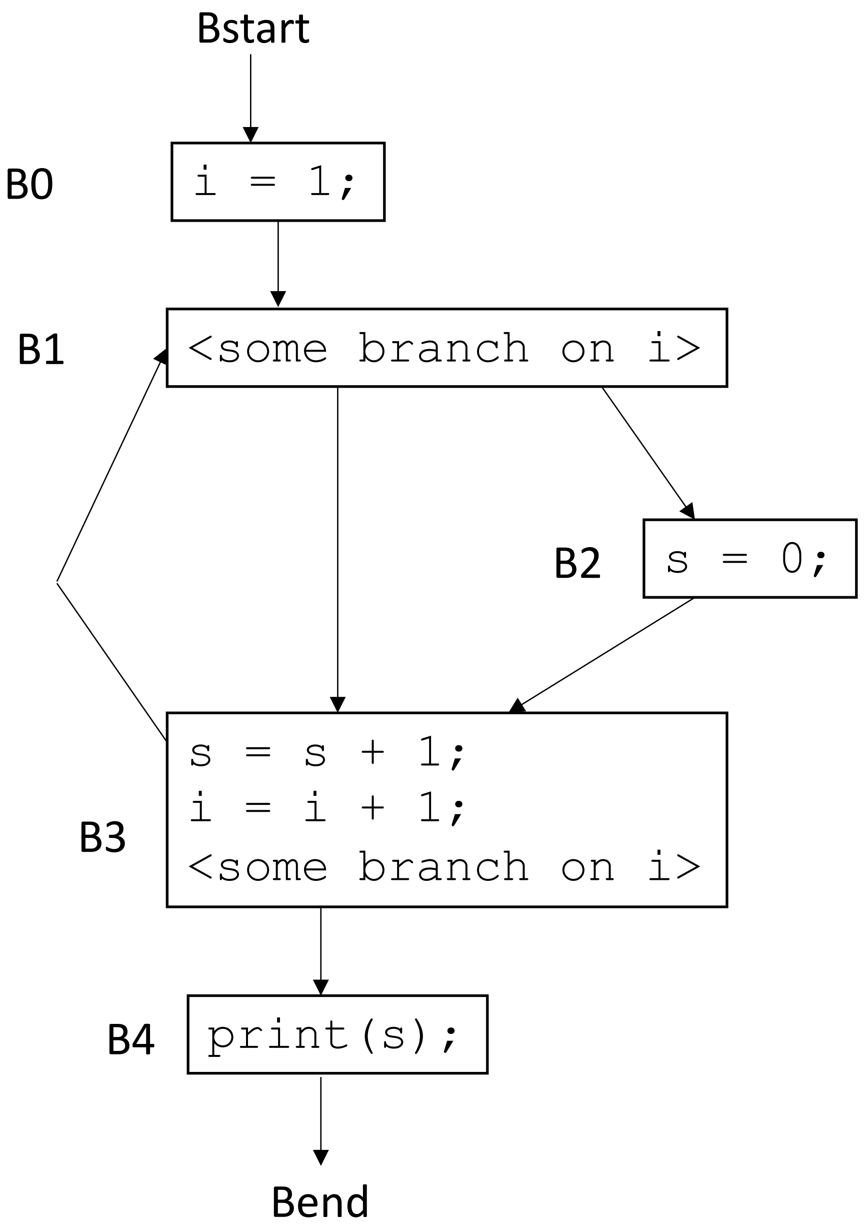
$$Dom(n) = \{n\} \cup (\bigcap_{p \text{ in } preds(n)} Dom(p))$$

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
- *Some vs. Every*

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

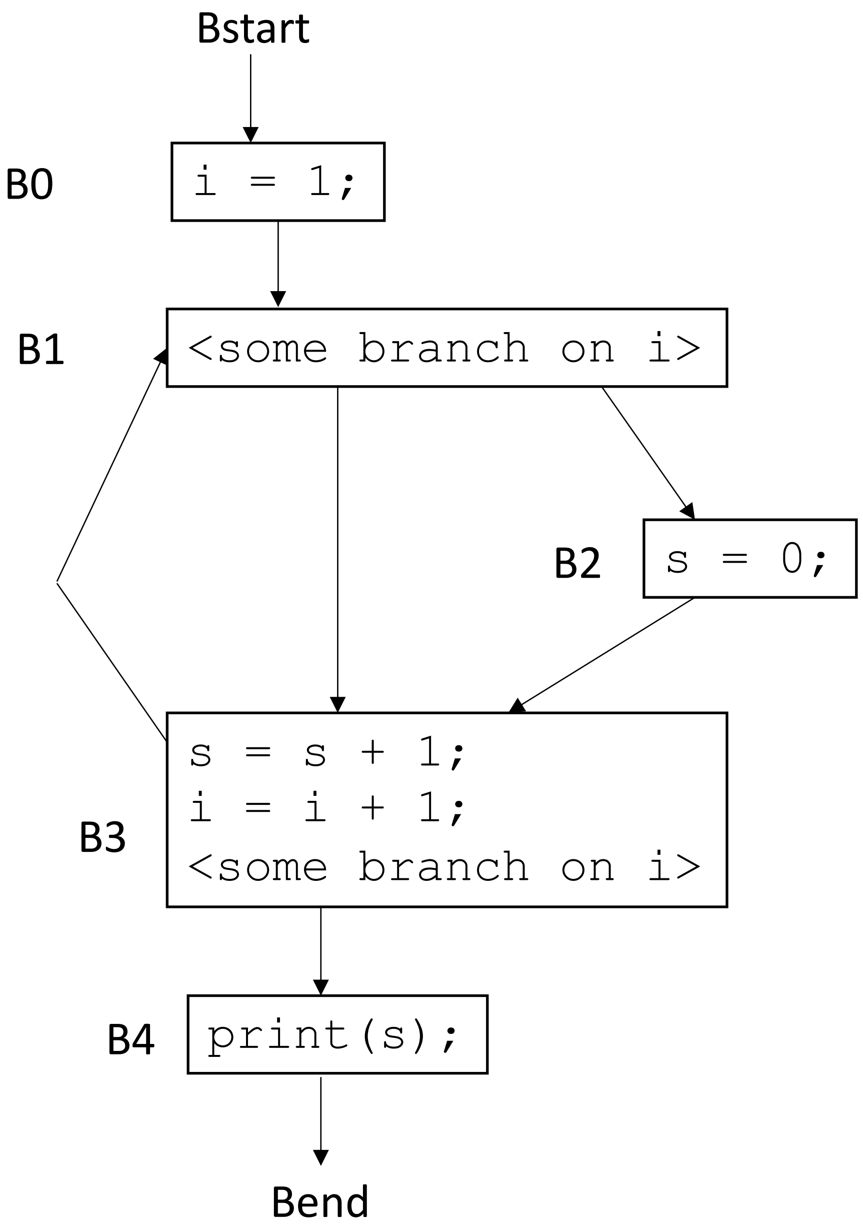
$$Dom(n) = \{n\} \cup \left(\bigcap_{p \text{ in } preds(n)} Dom(p) \right)$$



Now we can perform the iterative fixed point computation:

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

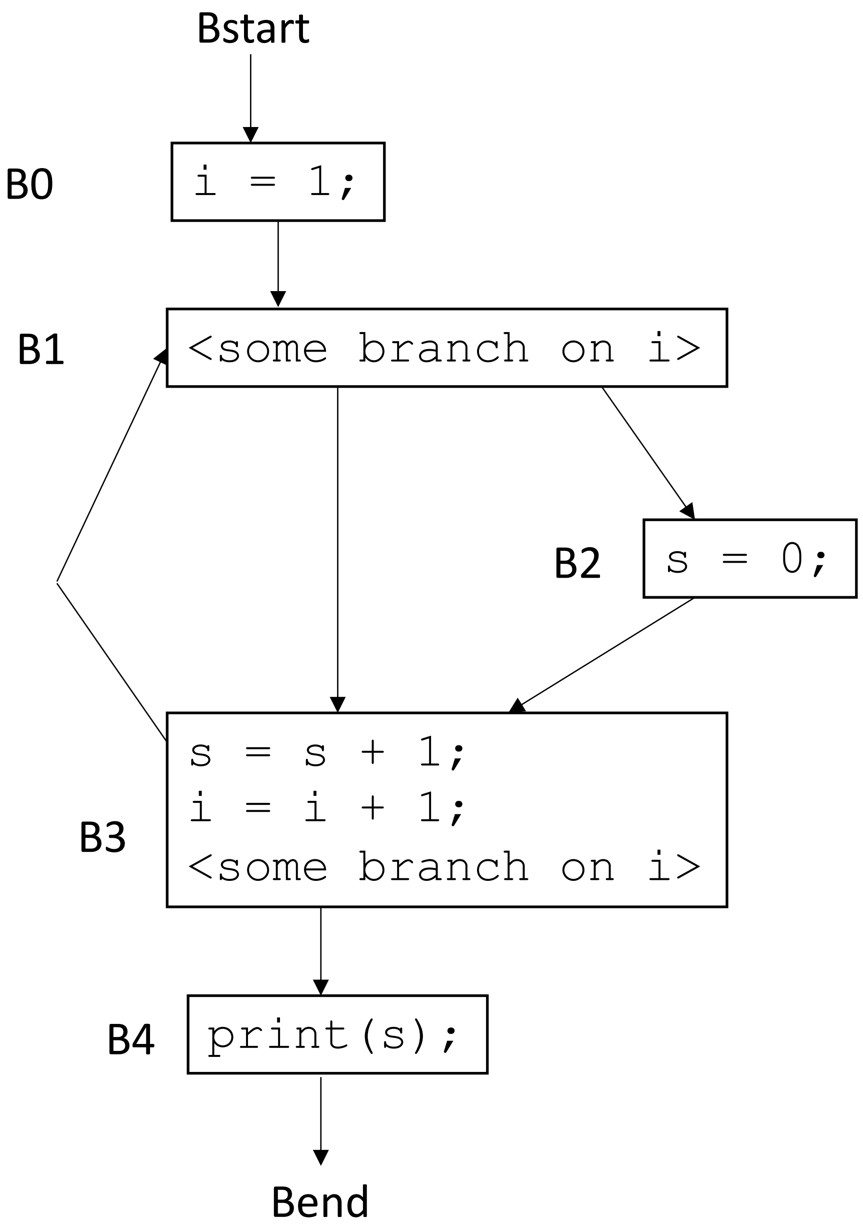
Block	VarKill	UEVar	~VarKill	LiveOut I_0
Bstart	{}	{}	i,s	{}
B0	i	{}	s	{}
B1	{}	i	i,s	{}
B2	s	{}	i	{}
B3	i,s	i,s	{}	{}
B4	{}	s	i,s	{}
Bend	{}	{}	i,s	{}



Now we can perform the iterative fixed point computation:

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

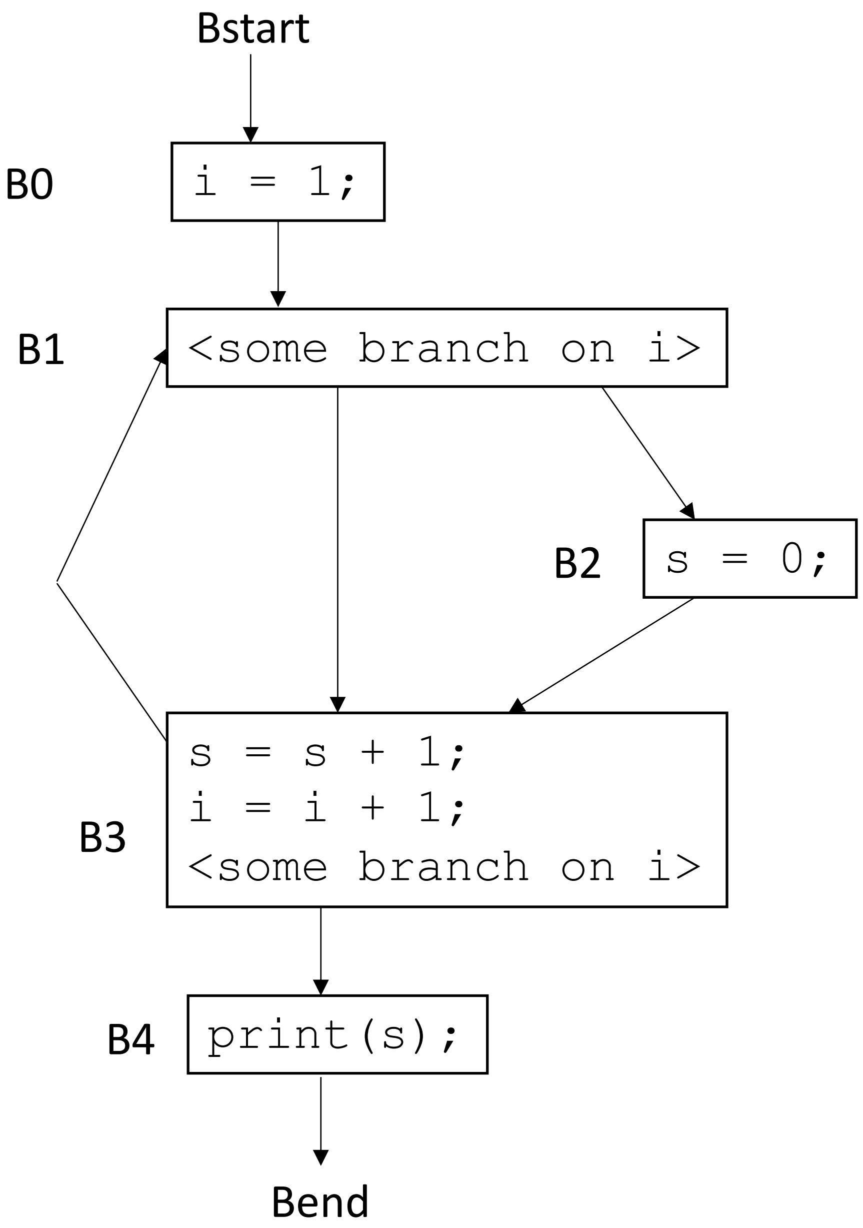
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1
Bstart	{}	{}	i,s	{}	
B0	i	{}	s	{}	
B1	{}	i	i,s	{}	
B2	s	{}	i	{}	
B3	i,s	i,s	{}	{}	
B4	{}	s	i,s	{}	
Bend	{}	{}	i,s	{}	



Now we can perform the iterative fixed point computation:

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

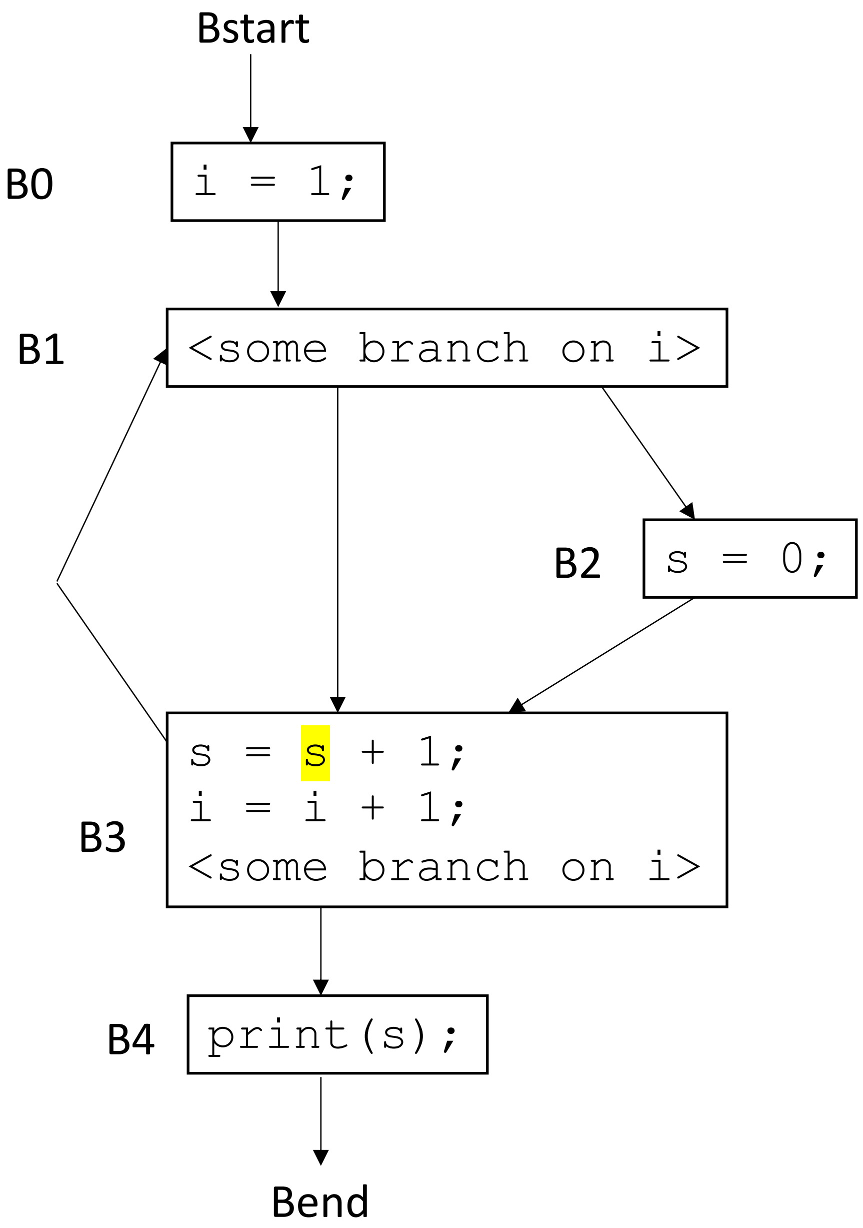
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	
B0	i	{}	s	{}	i	
B1	{}	i	i,s	{}	i,s	
B2	s	{}	i	{}	i,s	
B3	i,s	i,s	{}	{}	i,s	
B4	{}	s	i,s	{}	{}	
Bend	{}	{}	i,s	{}	{}	



Now we can perform the iterative fixed point computation:

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

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	{}	{}	{}	



Now we can perform the iterative fixed point computation:

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

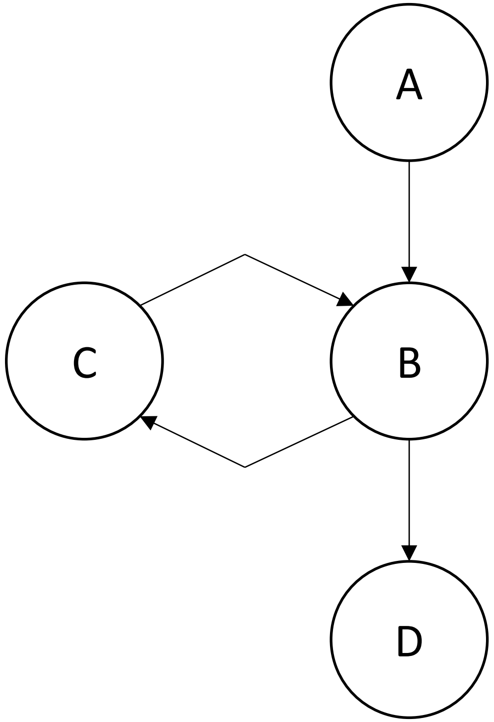
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2	.. I_3
Bstart	{}	{}	i,s	{}	{}	{}	s
B0	i	{}	s	{}	i	i,s	i,s
B1	{}	i	i,s	{}	i,s	i,s	i,s
B2	s	{}	i	{}	i,s	i,s	i,s
B3	i,s	i,s	{}	{}	i,s	i,s	i,s
B4	{}	s	i,s	{}	{}	{}	{}
Bend	{}	{}	i,s	{}	{}	{}	{}

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?

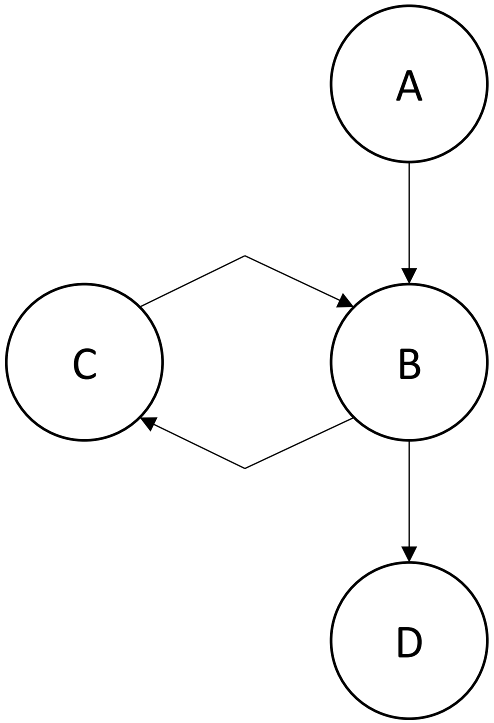
Example

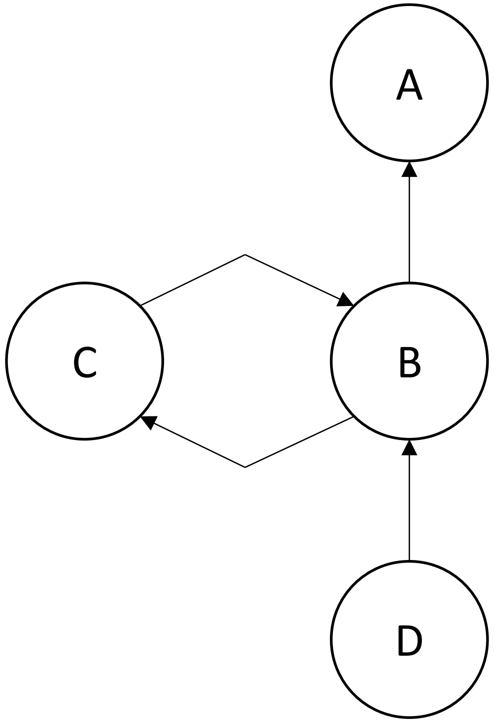
post order: D, C, B, A



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



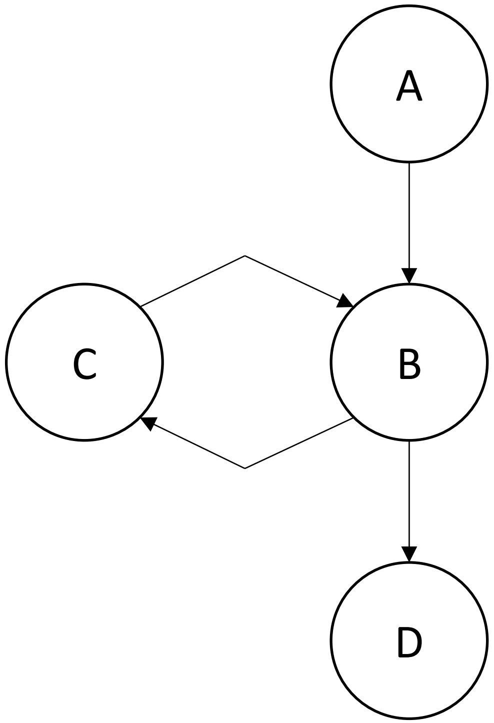


reverse CFG

post order: D, C, B, A

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

Example

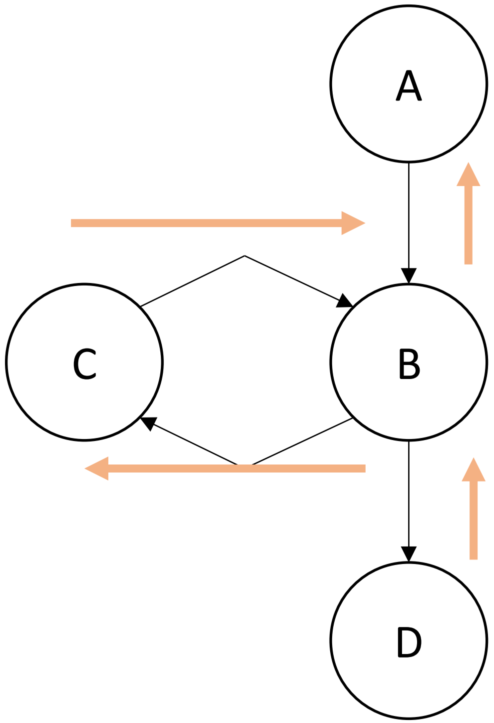


post order: D, C, B, A

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



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*

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

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;
```

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

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*

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

Live variable limitations

Imprecision can come from CFG construction:

consider:

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

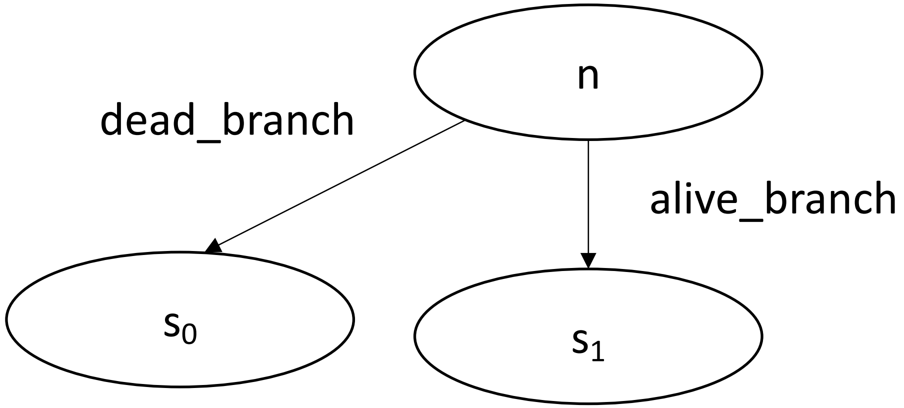
Live variable limitations

Imprecision can come from CFG construction:

consider:

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

could come from arguments, etc.



Live variable limitations

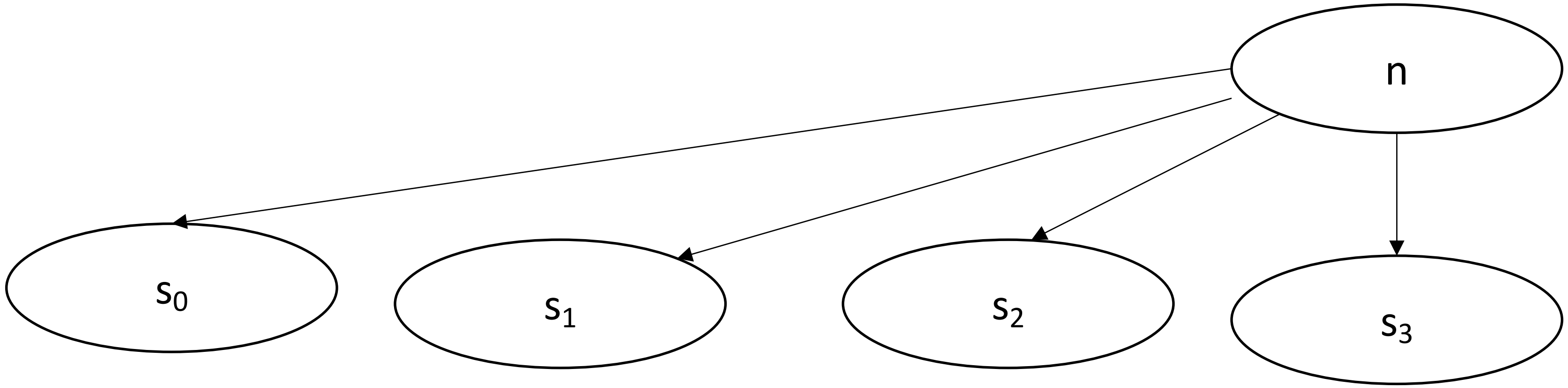
Imprecision can come from CFG construction:

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

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

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

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

Forward Flow

Available Expressions

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

intersection implies “must” analysis

Available Expressions

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

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

Available Expressions

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

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

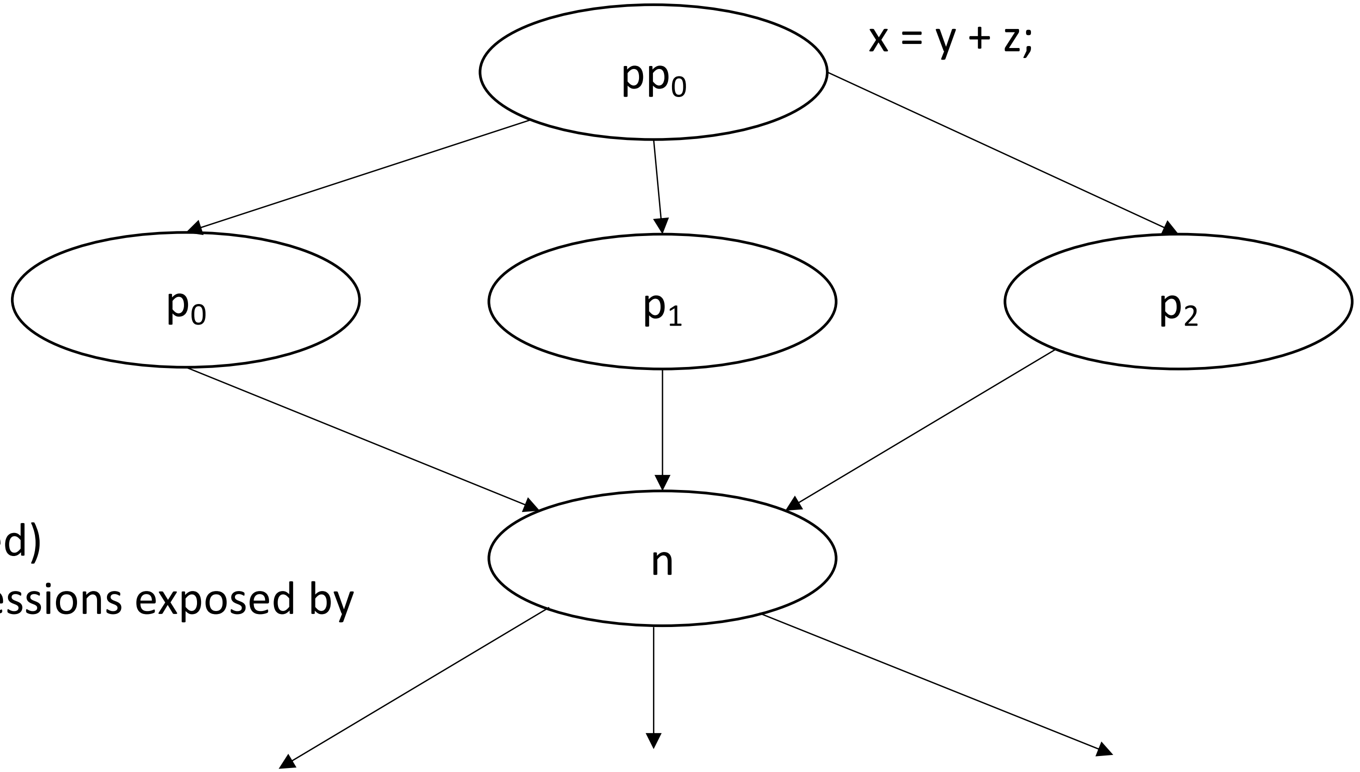
Available Expressions

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

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

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

***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

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

”must” analysis

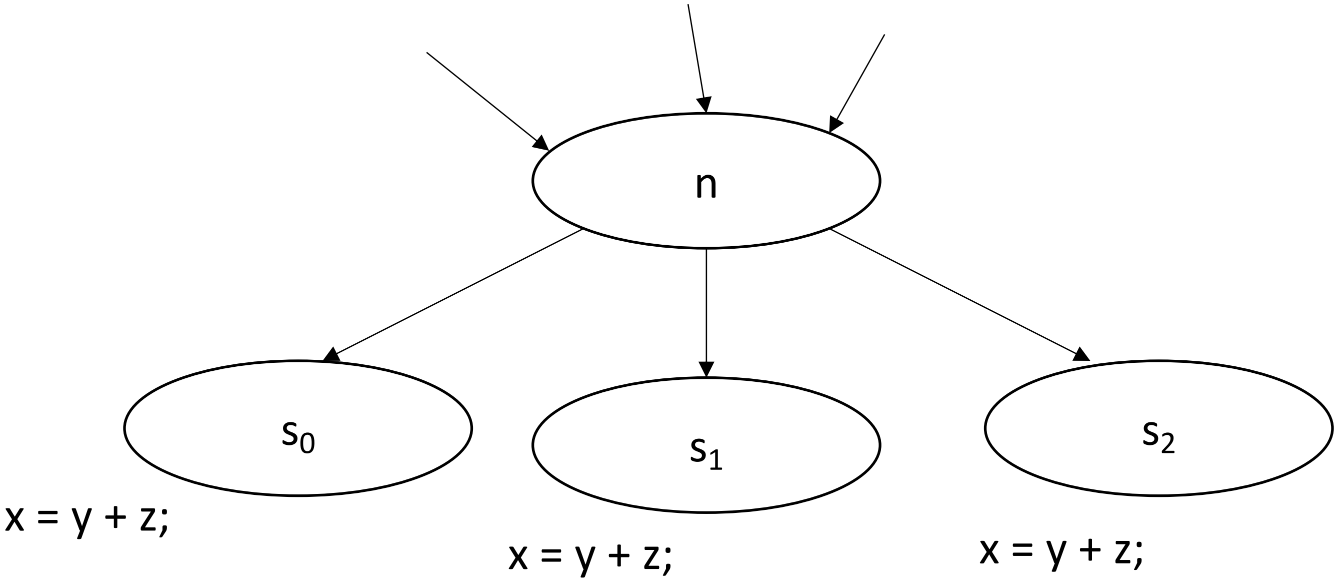
Anticipable Expressions

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

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

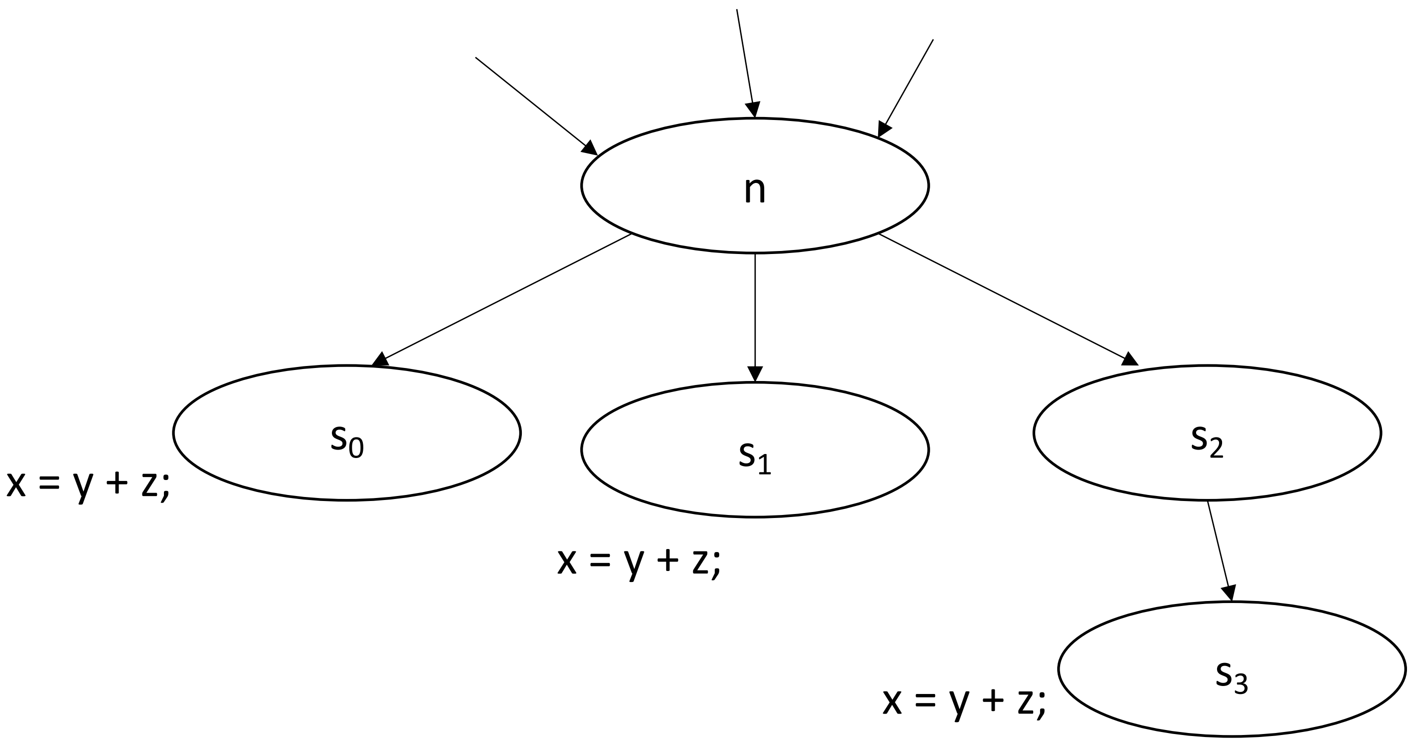
Anticipable Expressions

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



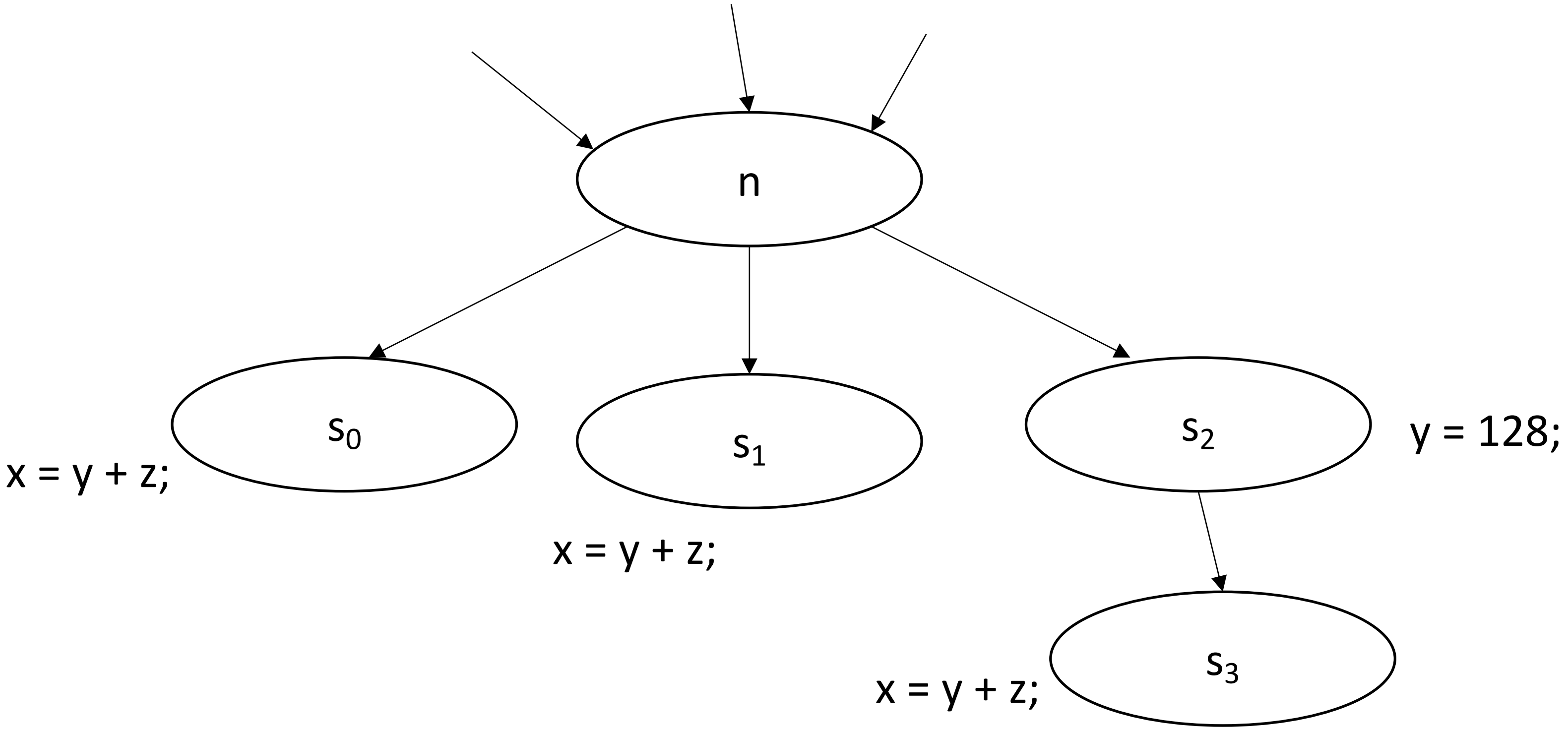
Anticipable Expressions

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



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

Next time:

- More global analysis!