CSE211: Compiler Design

Oct. 22, 2021

- **Topic**: More flow analysis applications and intro to SSA

- **Questions**:
  - *Questions or comments about homework 1?*
  - *Questions or comments about homework 2?*

```
3:                                                ; preds = %1
  %4 = tail call i32 @_Z14first_functionv(), !dbg !19
  call void @llvm.dbg.value(metadata i32 %4, metadata !14, metadata
  br label %7, !dbg !21

5:                                                ; preds = %1
  %6 = tail call i32 @_Z15second_functionv(), !dbg !22
  call void @llvm.dbg.value(metadata i32 %6, metadata !14, metadata
  br label %7

7:                                                ; preds = %5, %3
  %8 = phi i32 [ %4, %3 ], [ %6, %5 ], !dbg !24
  call void @llvm.dbg.value(metadata i32 %8, metadata !14, metadata
  ret i32 %8, !dbg !25
}
```

Announcements

- Homework 2:
  - Due Nov. 1
  - Great questions on slack!
  - I’ll have office hours next thursday
- Back to in-person on Monday!

CSE211: Compiler Design

Oct. 22, 2021

- **Topic**: More flow analysis applications and intro to SSA

- **Questions**:
  - *Questions or comments about homework 1?*
  - *Questions or comments about homework 2?*

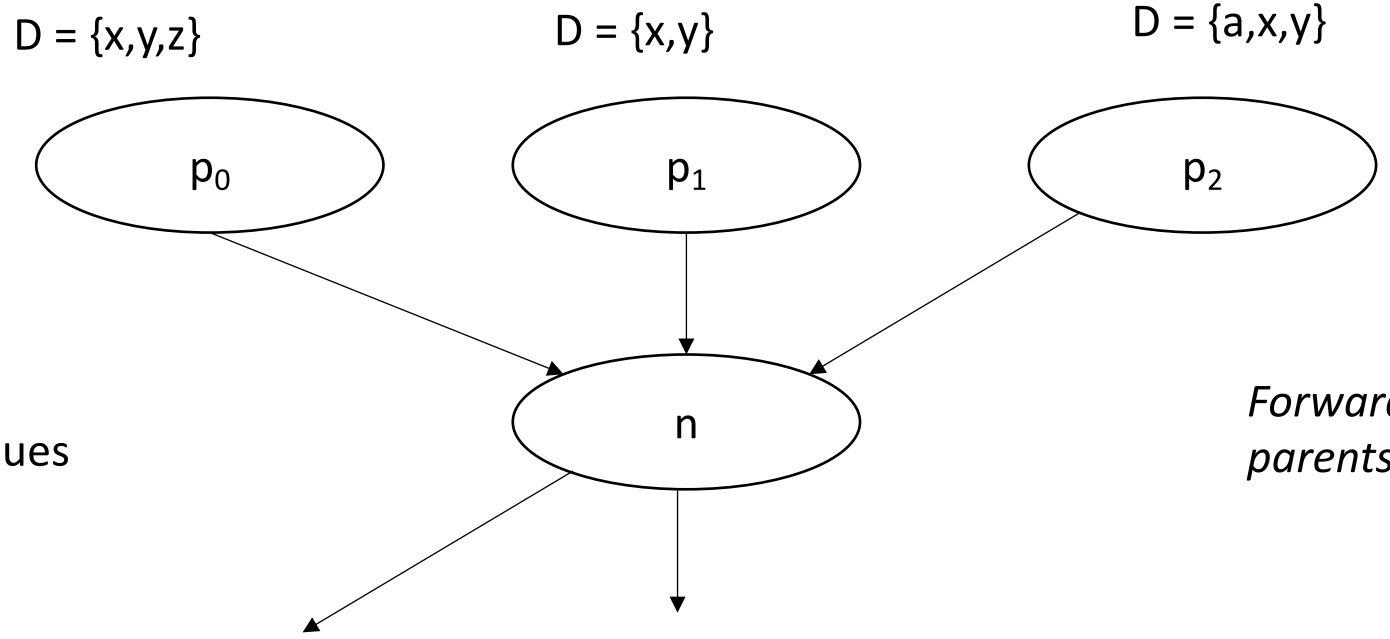
```
3:                                                  ; preds = %1
  %4 = tail call i32 @_Z14first_functionv(), !dbg !19
  call void @llvm.dbg.value(metadata i32 %4, metadata !14, metadata
  br label %7, !dbg !21

5:                                                  ; preds = %1
  %6 = tail call i32 @_Z15second_functionv(), !dbg !22
  call void @llvm.dbg.value(metadata i32 %6, metadata !14, metadata
  br label %7

7:                                                  ; preds = %5, %3
  %8 = phi i32 [ %4, %3 ], [ %6, %5 ], !dbg !24
  call void @llvm.dbg.value(metadata i32 %8, metadata !14, metadata
  ret i32 %8, !dbg !25
}
```

Global optimizations review: Dominance

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



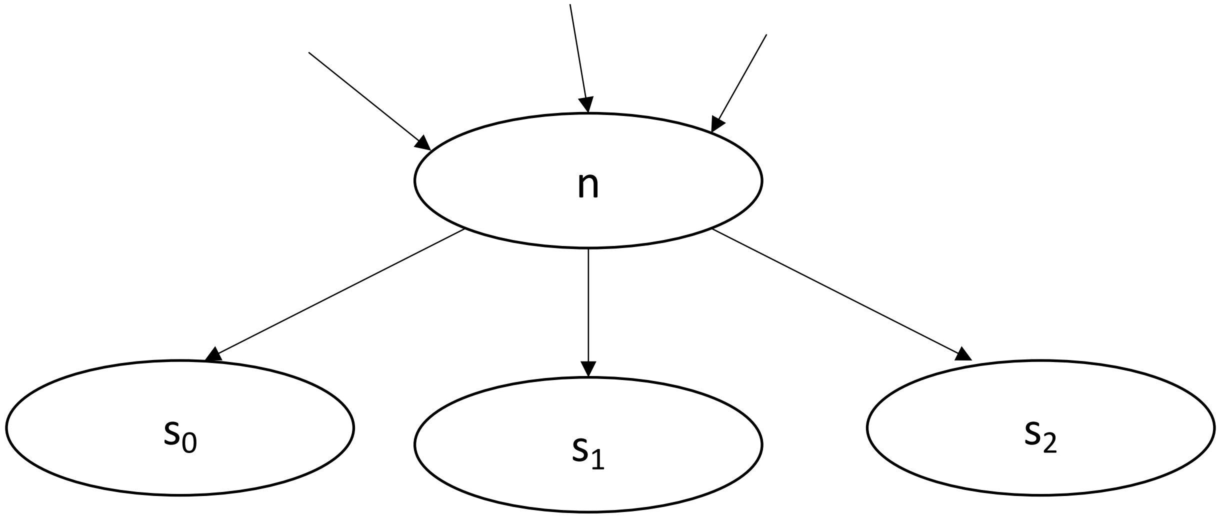
update:
intersection of parent values

Forward flow, as updates flow from parents to children.

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

Global optimizations review: Live variable analysis

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



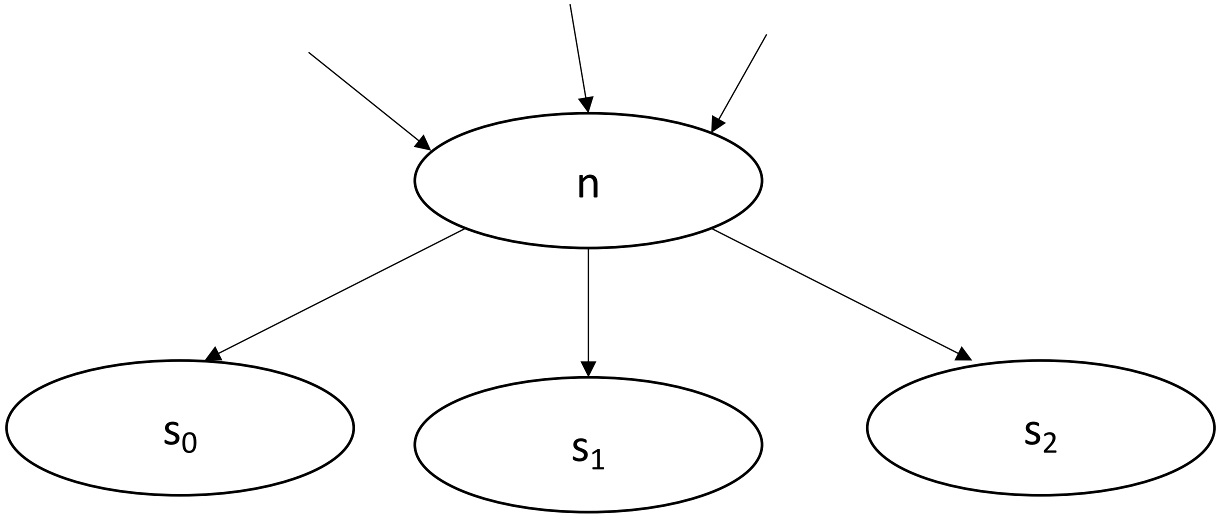
backwards flow

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

Global optimizations review: Live variable analysis

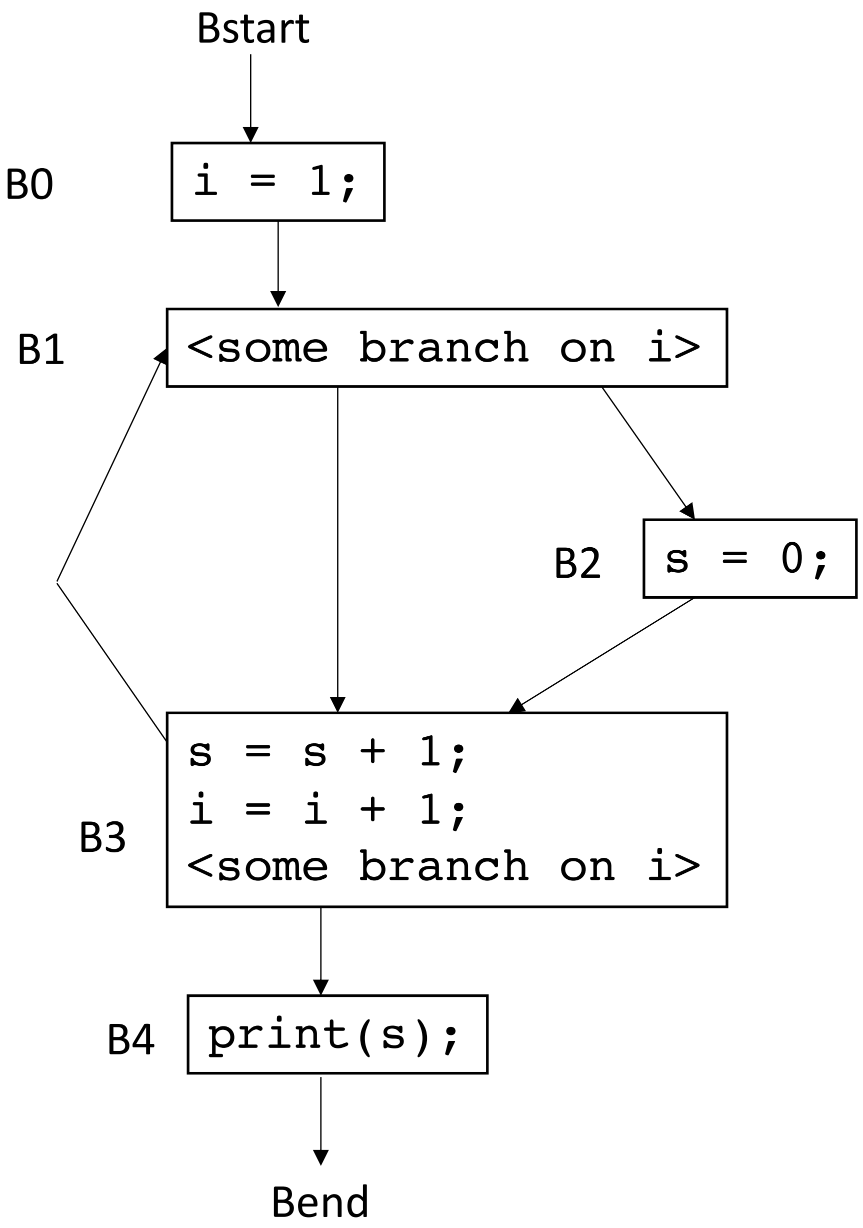
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup (\text{LiveOut}(s) \cap \overline{VarKill(s)}) \right)$$

What are the sets?



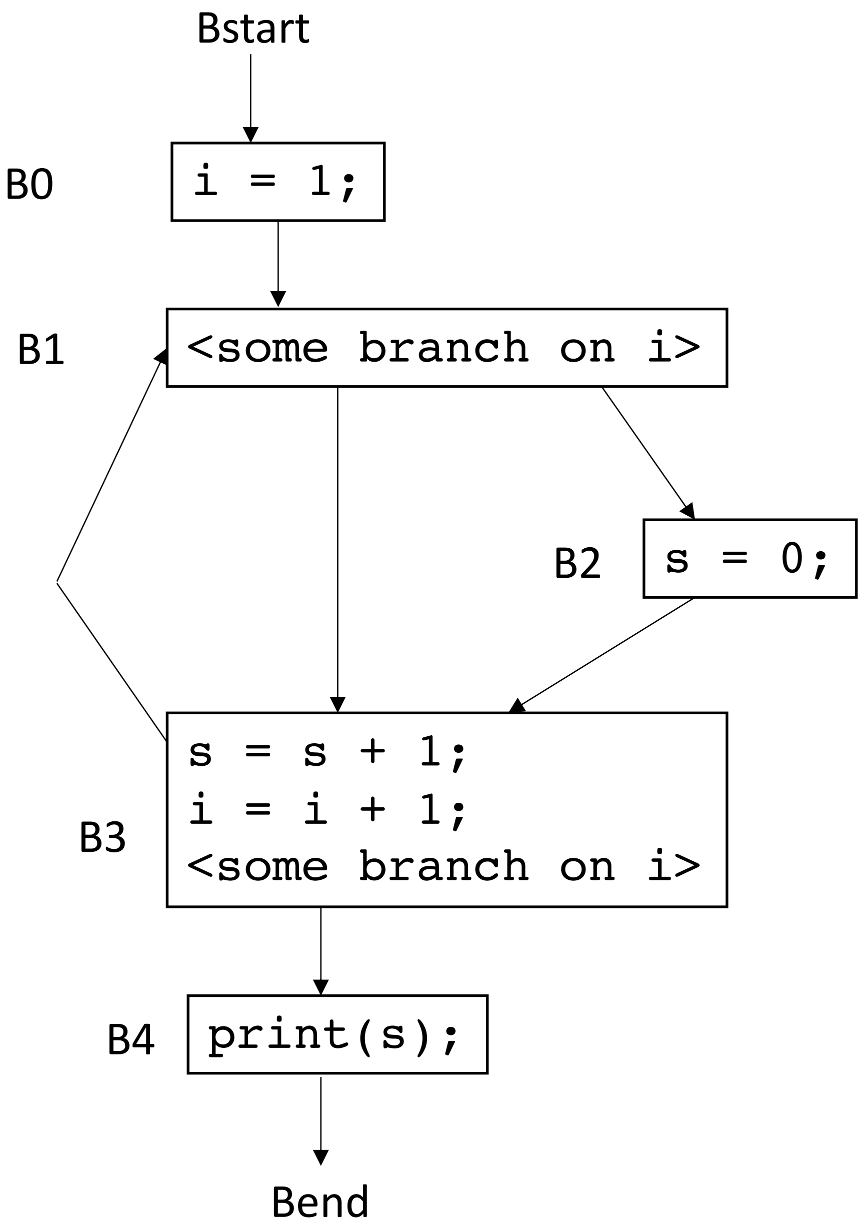
backwards flow

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



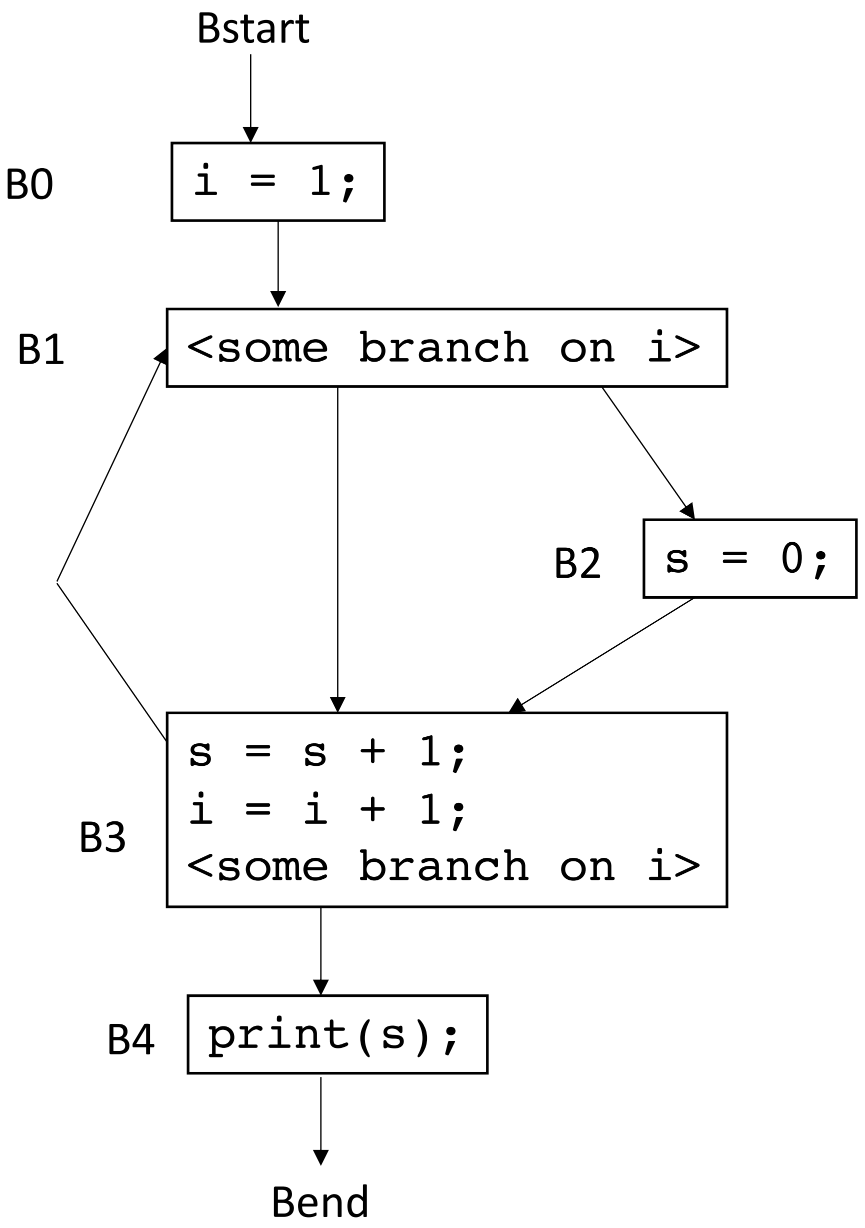
$$LiveOut(n) = \cup_{s \text{ in } succ(n)} \left(UEVar(s) \cup \left(LiveOut(s) \cap \overline{VarKill(s)} \right) \right)$$

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



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

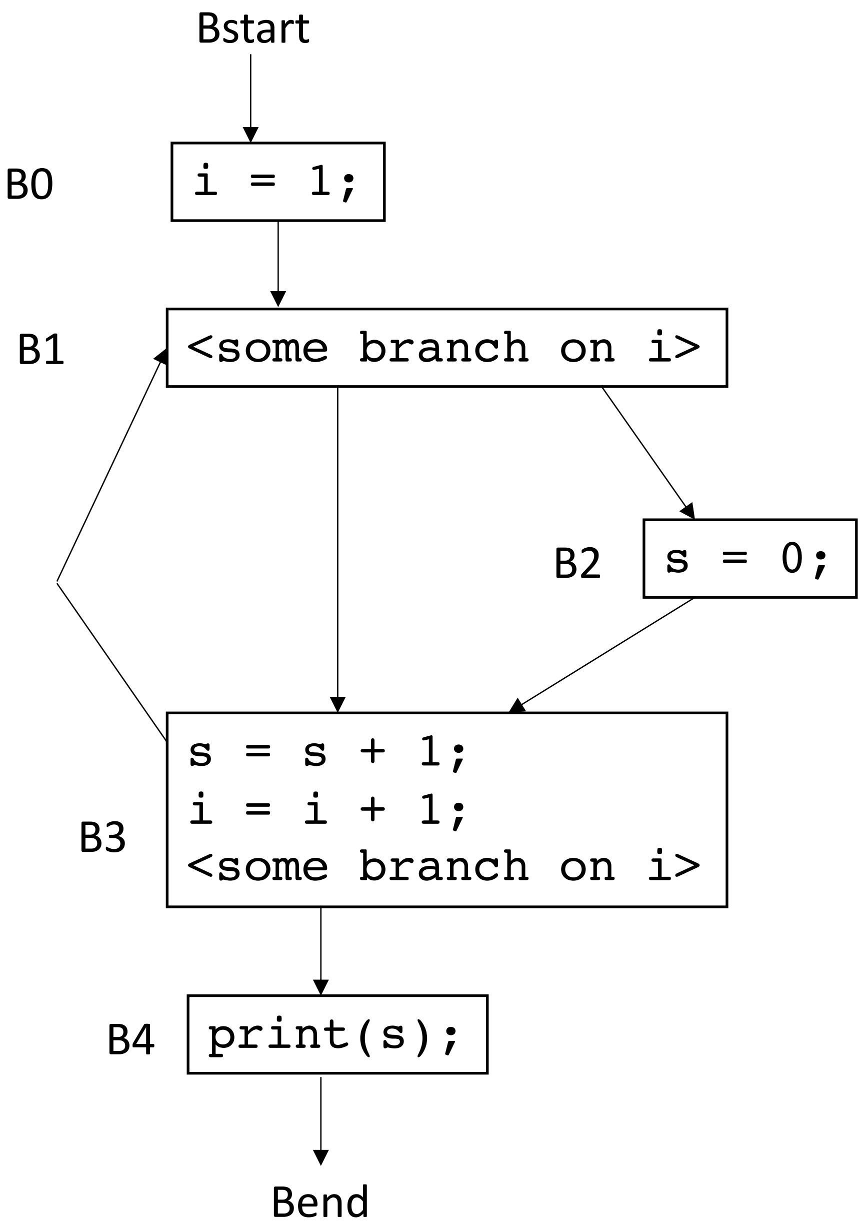
Block	VarKill	UEVar	~VarKill	LiveOut I_0
Bstart	{}	{}		
B0	i	{}		
B1	{}	i		
B2	s	{}		
B3	i,s	i,s		
B4	{}	s		
Bend	{}	{}		



Now we can perform the iterative fixed point computation:

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

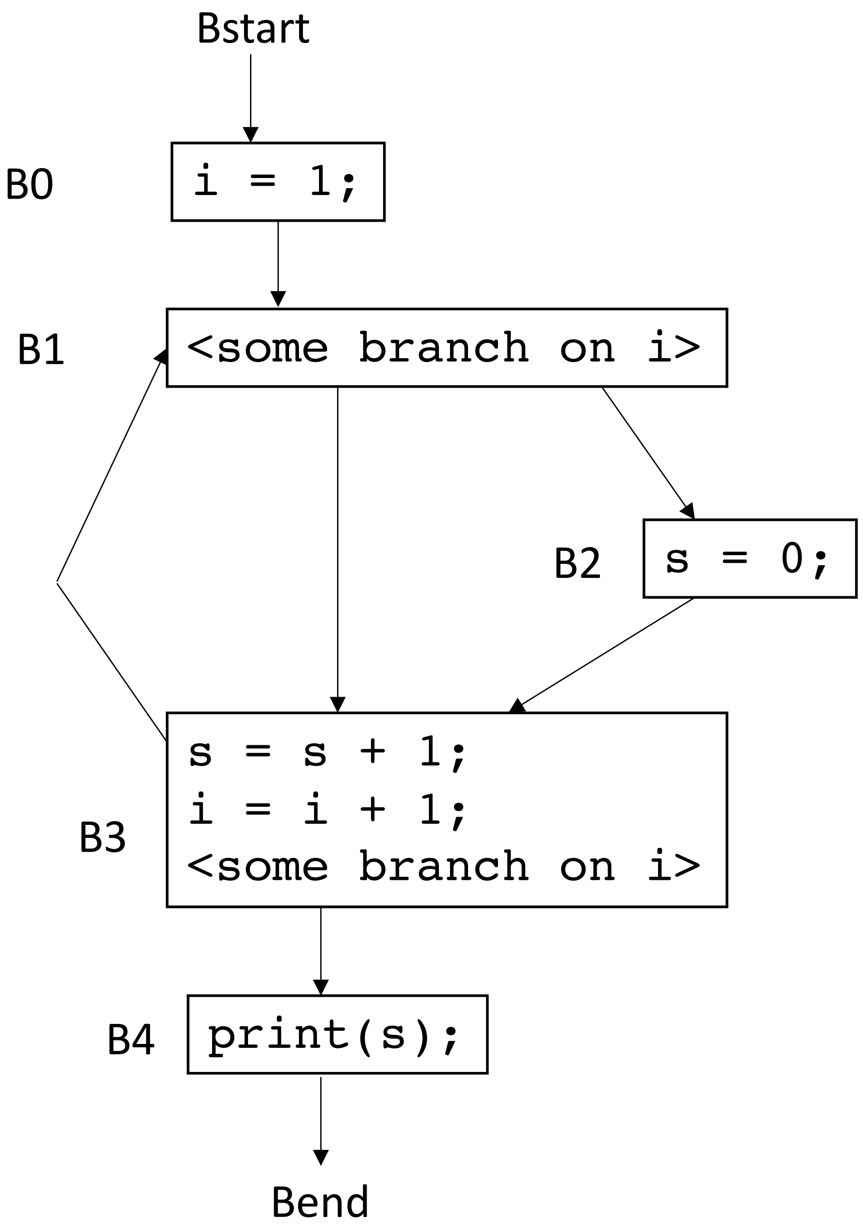
Block	VarKill	UEVar	~VarKill	LiveOut I_0
Bstart	{}	{}	i,s	{}
B0	i	{}	s	{}
B1	{}	i	i,s	{}
B2	s	{}	i	{}
B3	i,s	i,s	{}	{}
B4	{}	s	i,s	{}
Bend	{}	{}	i,s	{}



Now we can perform the iterative fixed point computation:

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

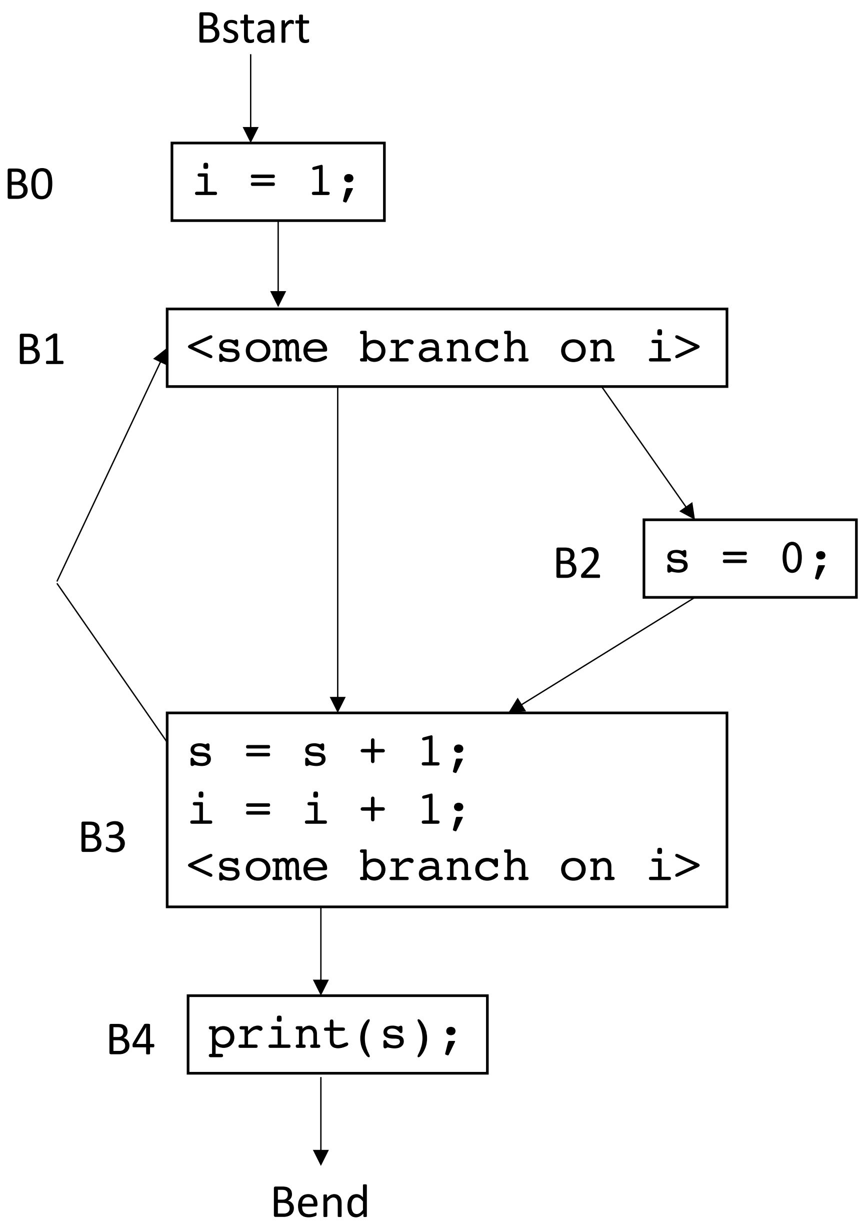
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1
Bstart	{}	{}	i,s	{}	{}
B0	i	{}	s	{}	i
B1	{}	i	i,s	{}	i,s
B2	s	{}	i	{}	i,s
B3	i,s	i,s	{}	{}	i,s
B4	{}	s	i,s	{}	{}
Bend	{}	{}	i,s	{}	{}



Now we can perform the iterative fixed point computation:

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

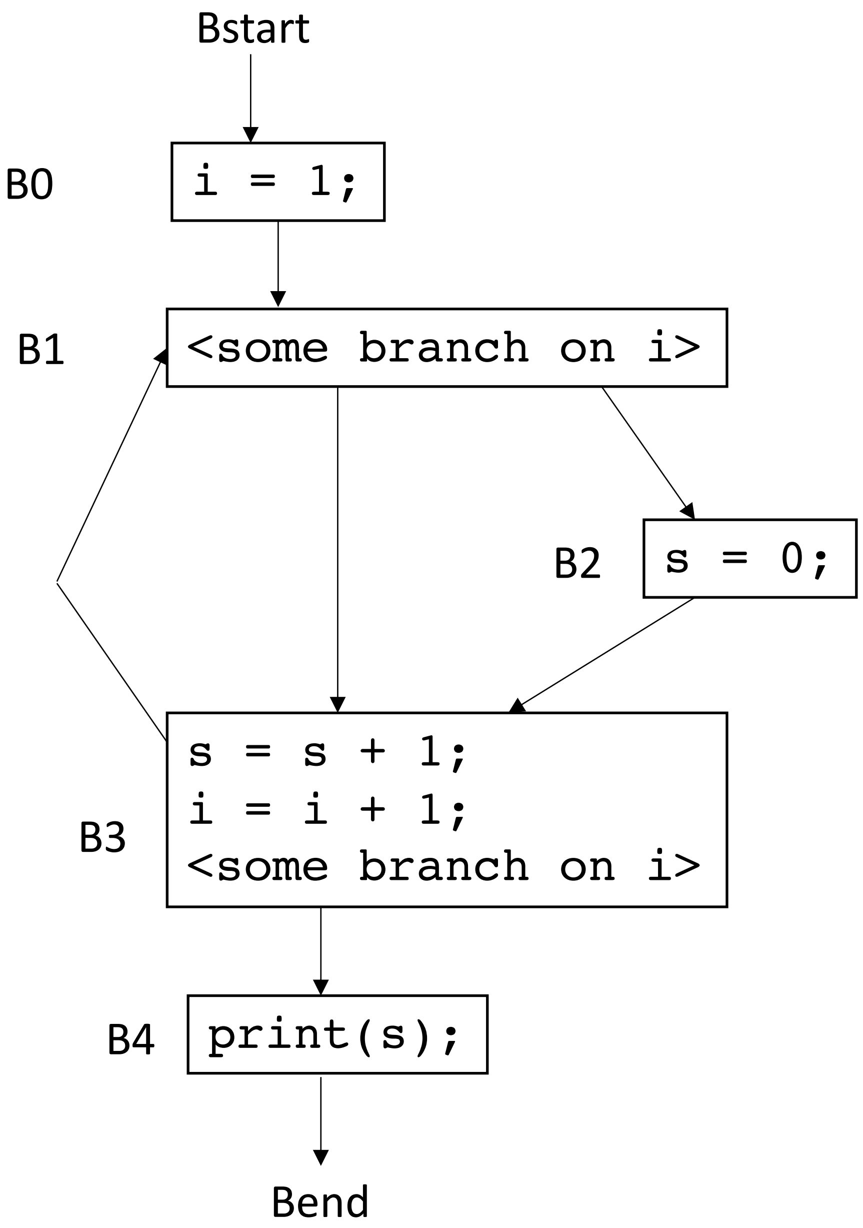
Block	VarKill	UEVar	~VarKill	LiveOut l_0	LiveOut l_1
Bstart	{}	{}	i,s	{}	{}
B0	i	{}	s	{}	i
B1	{}	i	i,s	{}	i,s
B2	s	{}	i	{}	i,s
B3	i,s	i,s	{}	{}	i,s
B4	{}	s	i,s	{}	{}
Bend	{}	{}	i,s	{}	{}



Now we can perform the iterative fixed point computation:

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

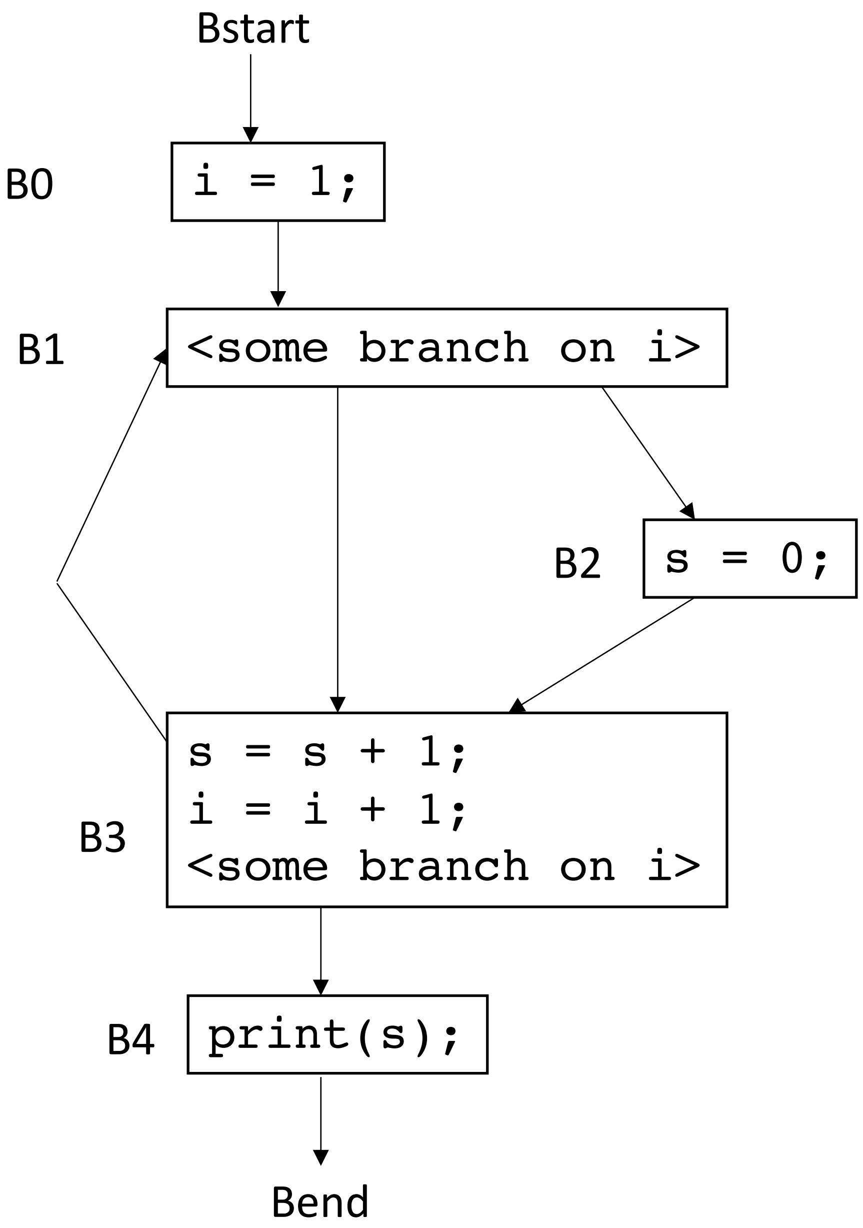
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	
B0	i	{}	s	{}	i	
B1	{}	i	i,s	{}	i,s	
B2	s	{}	i	{}	i,s	
B3	i,s	i,s	{}	{}	i,s	
B4	{}	s	i,s	{}	{}	
Bend	{}	{}	i,s	{}	{}	



Now we can perform the iterative fixed point computation:

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

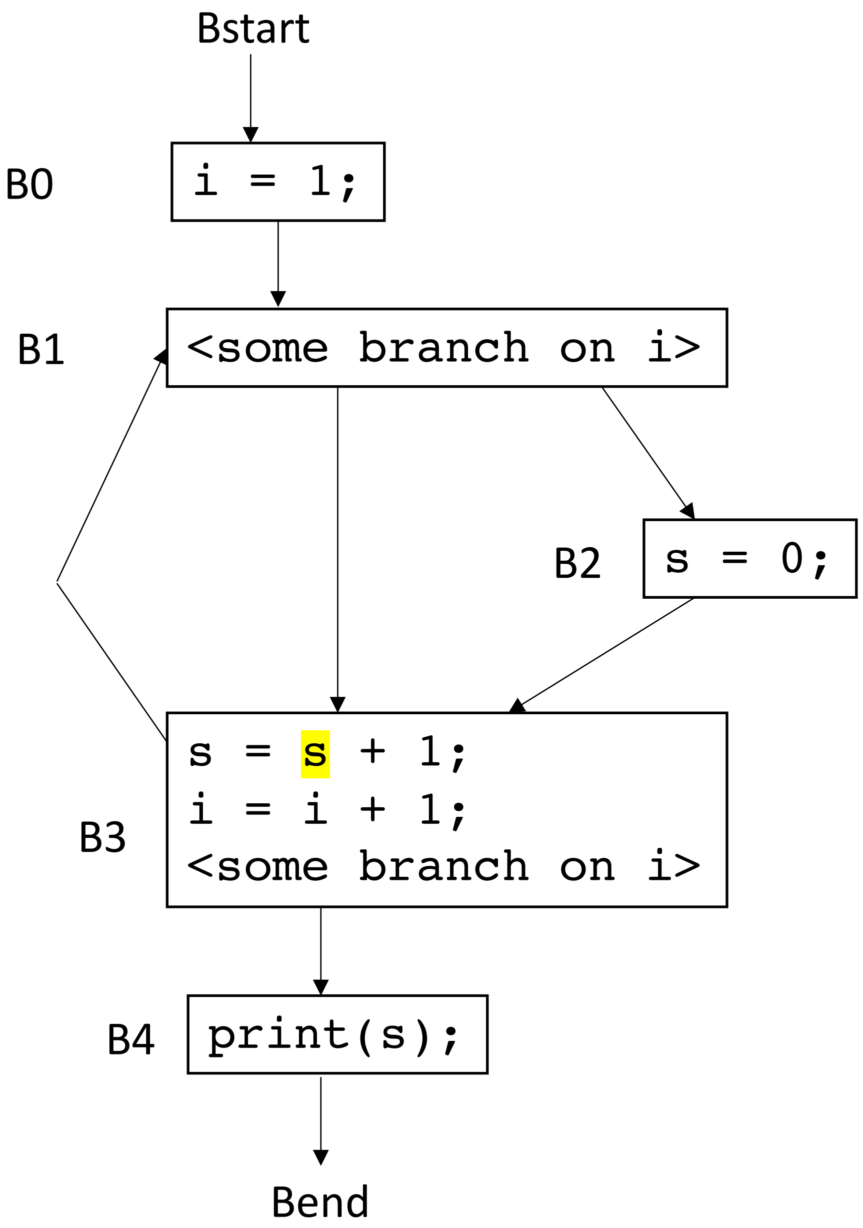
Block	VarKill	UEVar	~VarKill	LiveOut I_0	LiveOut I_1	LiveOut I_2
Bstart	{}	{}	i,s	{}	{}	{}
B0	i	{}	s	{}	i	i,s
B1	{}	i	i,s	{}	i,s	i,s
B2	s	{}	i	{}	i,s	i,s
B3	i,s	i,s	{}	{}	i,s	i,s
B4	{}	s	i,s	{}	{}	{}
Bend	{}	{}	i,s	{}	{}	{}



Now we can perform the iterative fixed point computation:

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

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



Now we can perform the iterative fixed point computation:

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

Block	VarKill	UEVar	~VarKill	LiveOut I_0	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

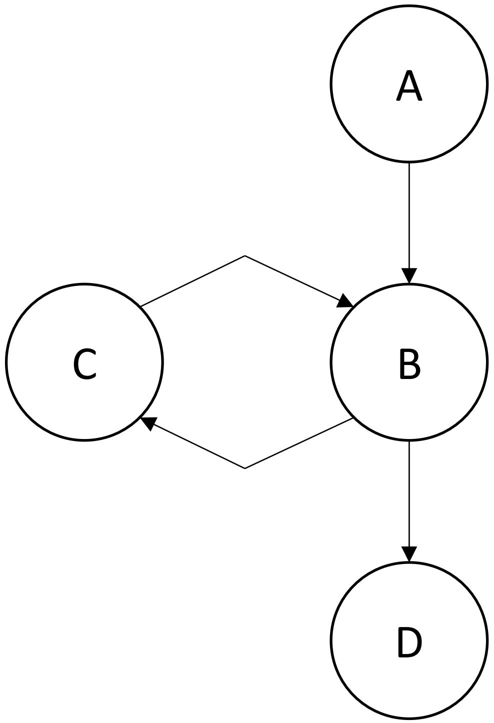
- Reverse post-order was good for forward flow:
  - Parents are computed before their children

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

- Different from post order?

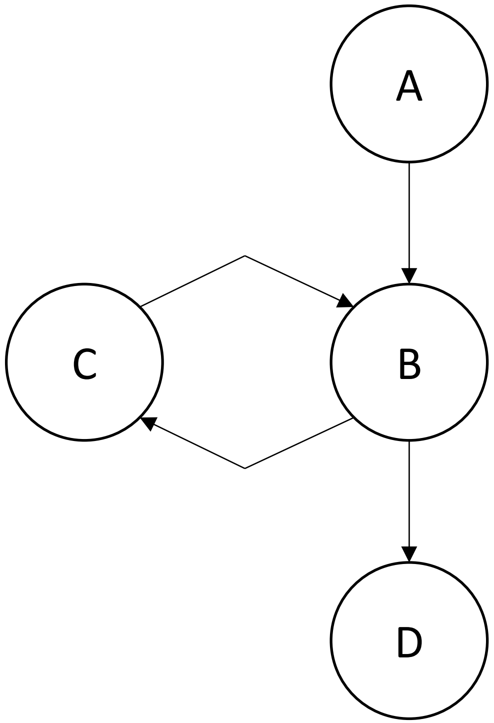
Example

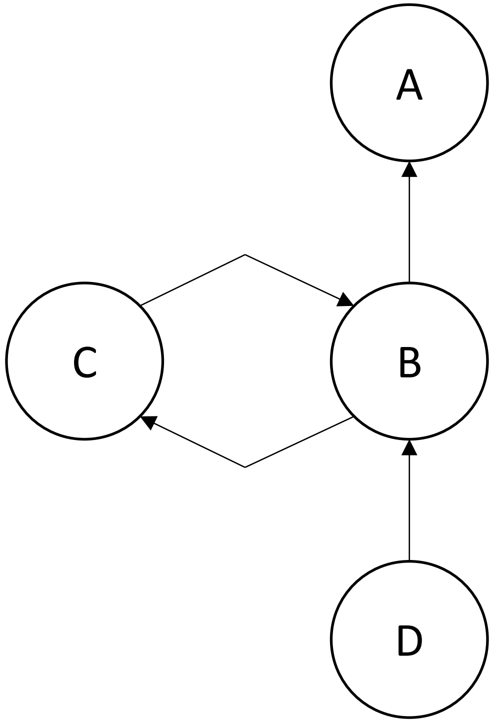
post order: D, C, B, A



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



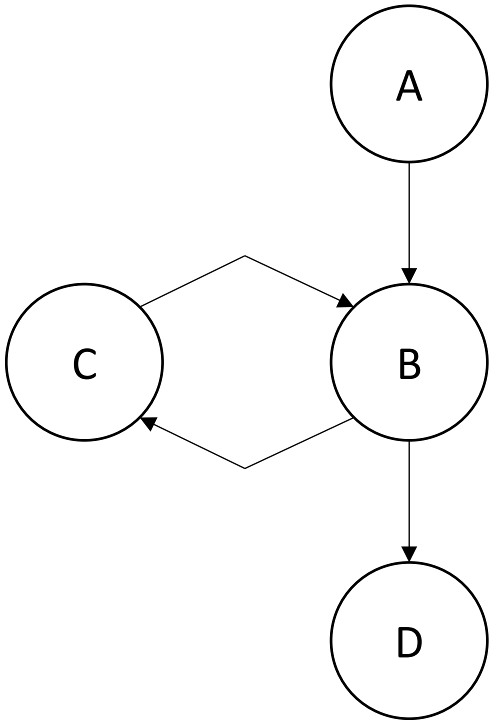


reverse CFG

post order: D, C, B, A

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

Example

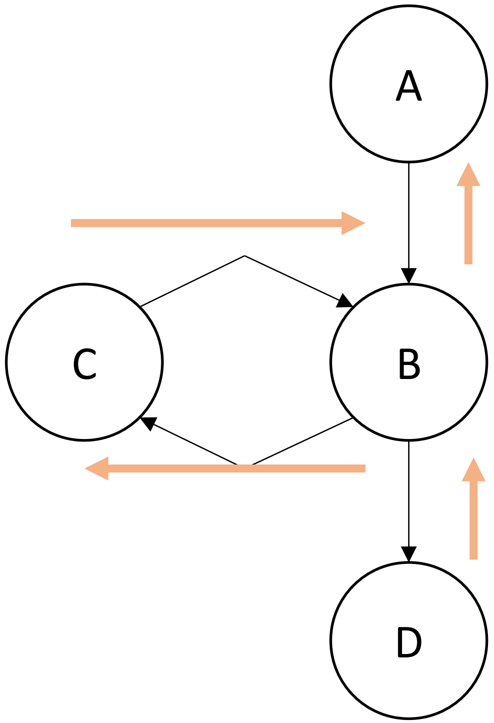


post order: D, C, B, A

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

Show PyCFG example from homework

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

Live variable limitations

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

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

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

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

Live variable limitations

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

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

UEVar (upward exposed variable) for block b is any variable in b that is read before being overwritten.

Consider:

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

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

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

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;
```

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

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;
```

VarKill *also needs to know about aliasing*

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

Demo

- Godbolt demo

Sound vs. Complete

- Sound: Any property the analysis says is true, is true. However, there may be false positives
- Complete: Any error the analysis reports is actually an error. The analysis cannot prove a property though.

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

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

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

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

Live variable limitations

Imprecision can come from CFG construction:

consider:

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

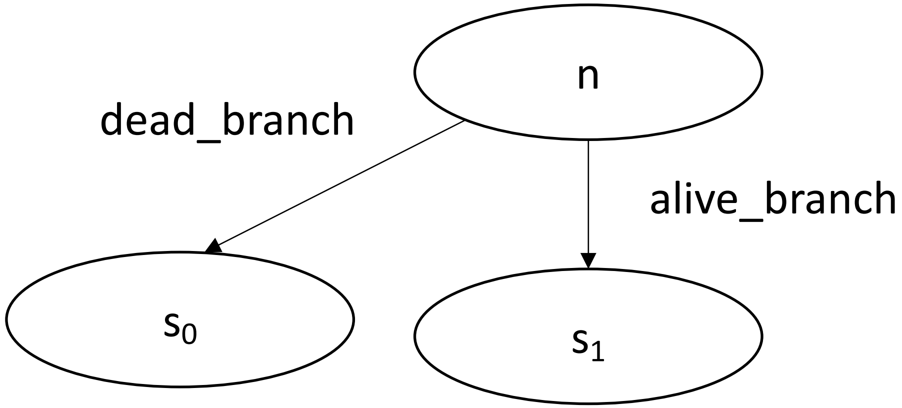
Live variable limitations

Imprecision can come from CFG construction:

consider:

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

could come from arguments, etc.



Live variable limitations

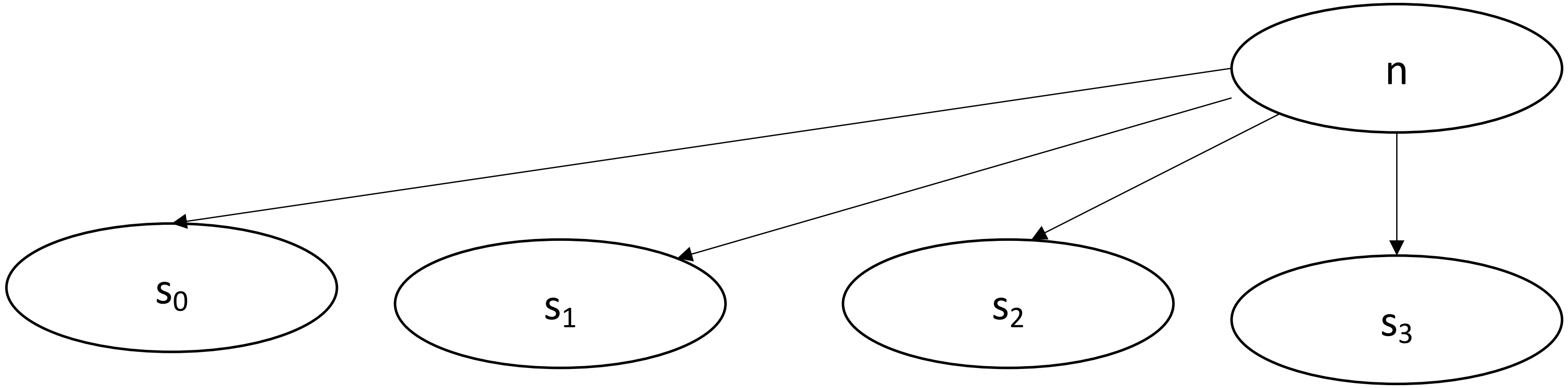
Imprecision can come from CFG construction:

consider first class labels (or functions):

```
br label_reg
```

where label_reg is a register that contains a register

need to branch to all possible basic blocks!



The Data Flow Framework

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

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

Available Expressions

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

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

Available Expressions

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

Forward Flow

Available Expressions

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

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

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

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

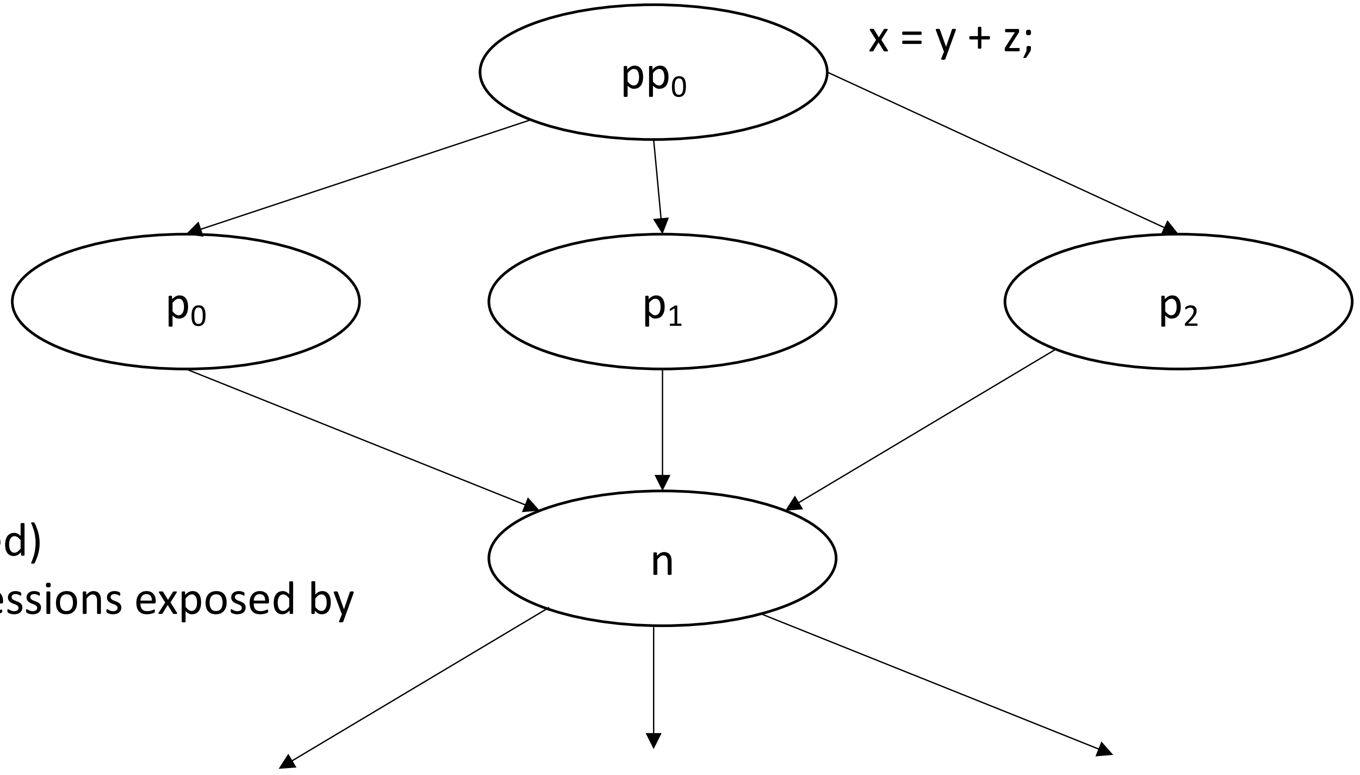
Available Expressions

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

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

Available Expressions

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



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

Available Expressions

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

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

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

”must” analysis

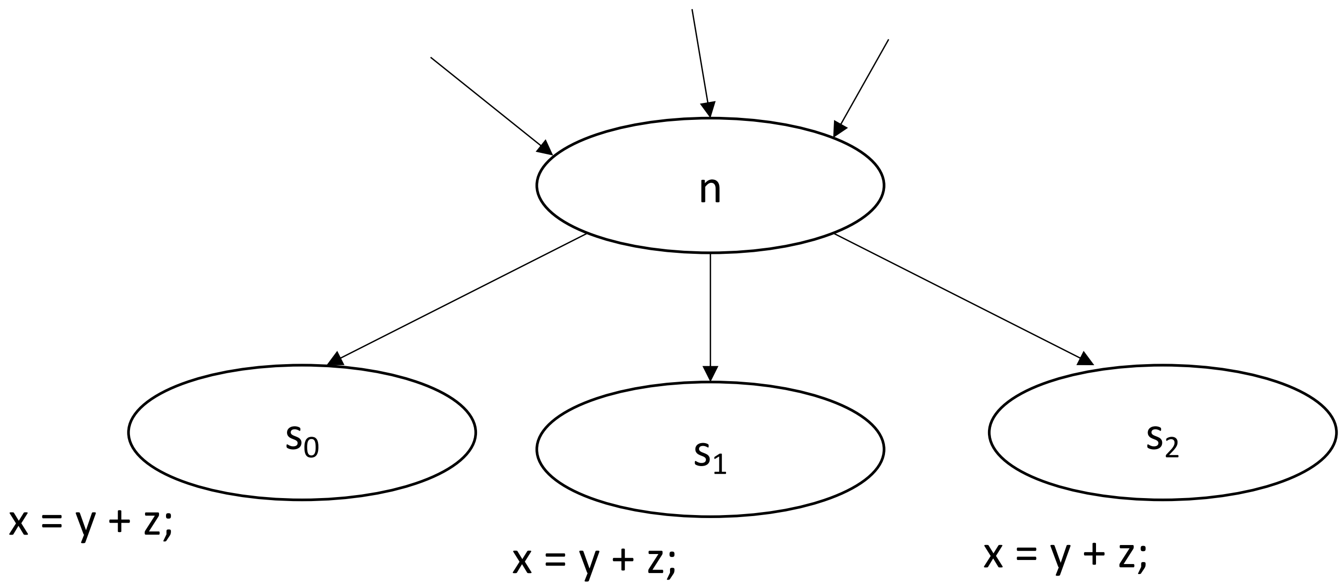
Anticipable Expressions

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

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

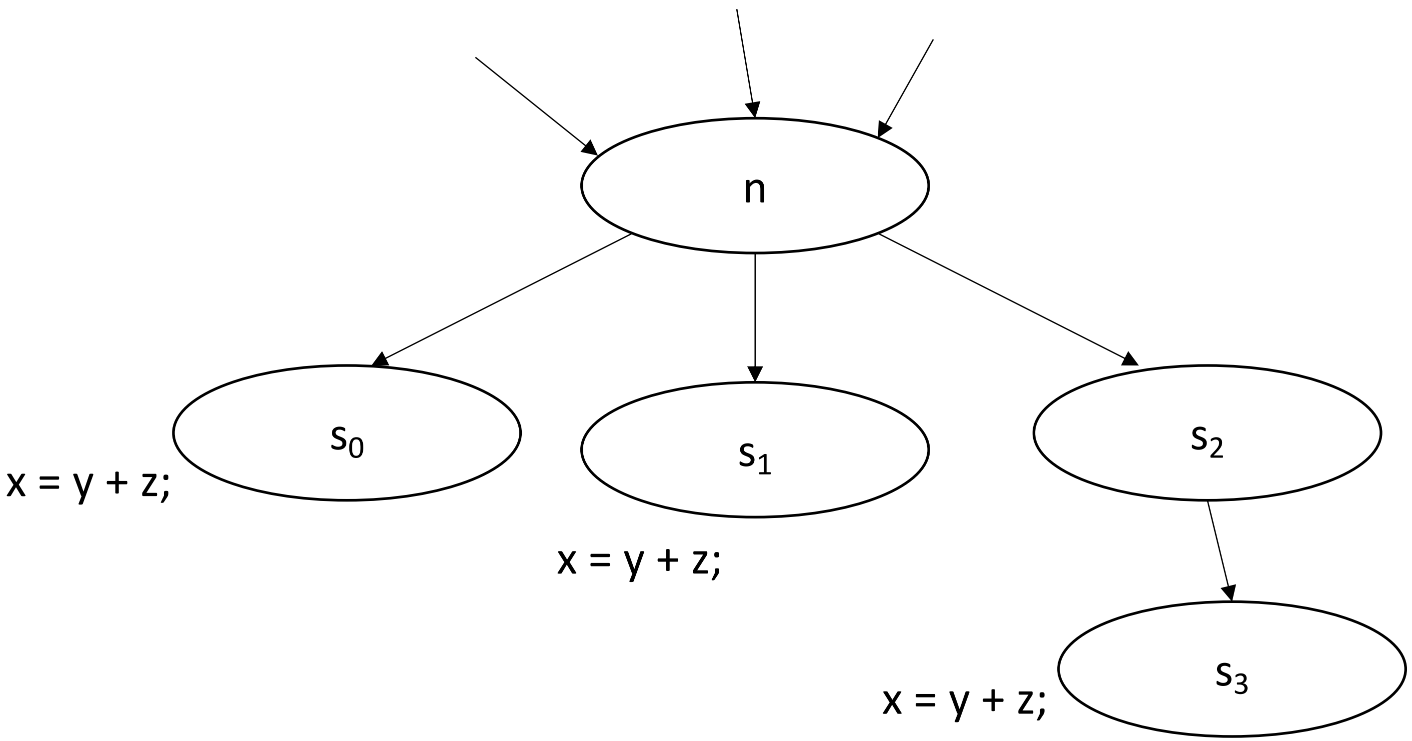
Anticipable Expressions

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



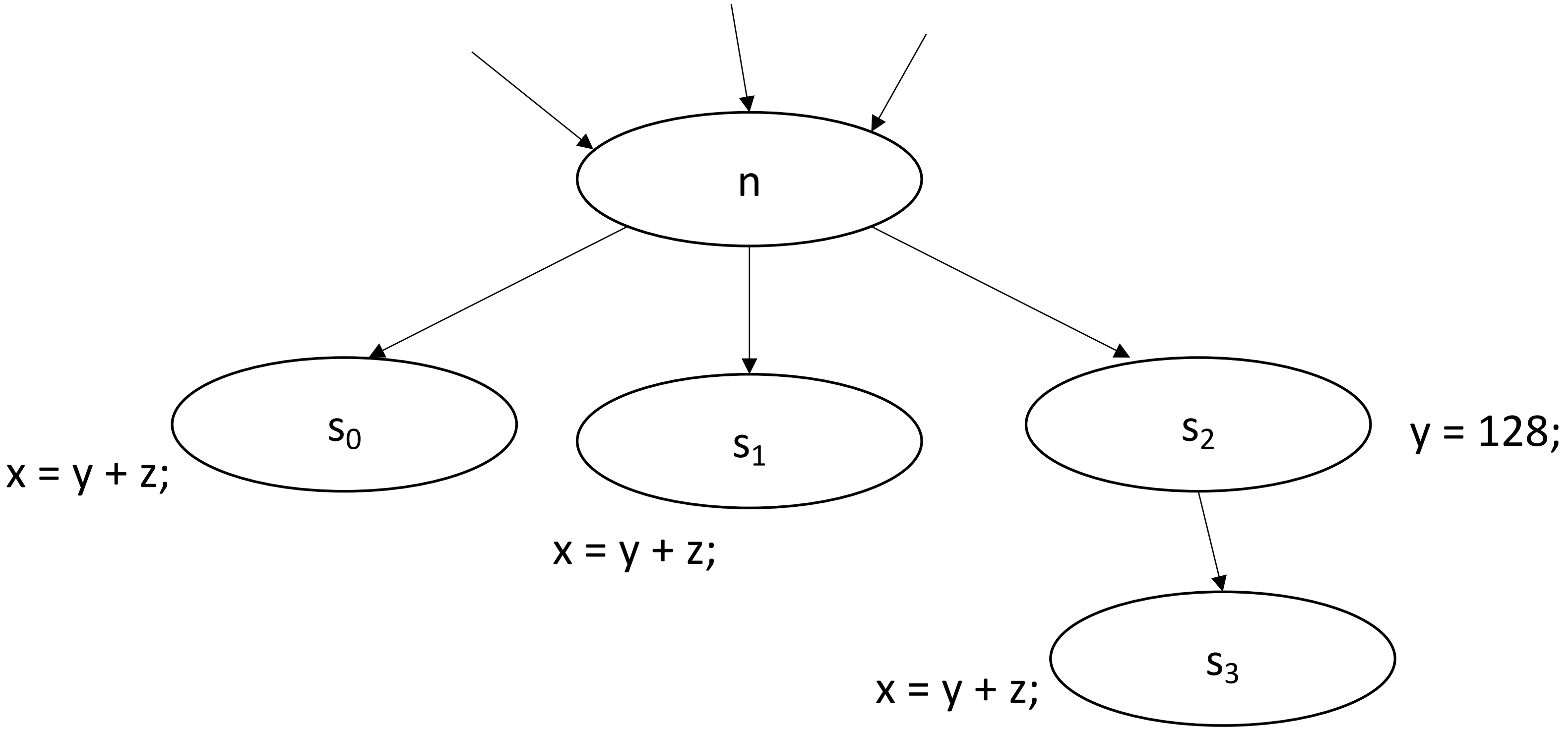
Anticipable Expressions

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



Anticipable Expressions

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



Anticipable Expressions

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

***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

See you in-person on Monday

- More optimal SSA construction
- Have a good weekend!