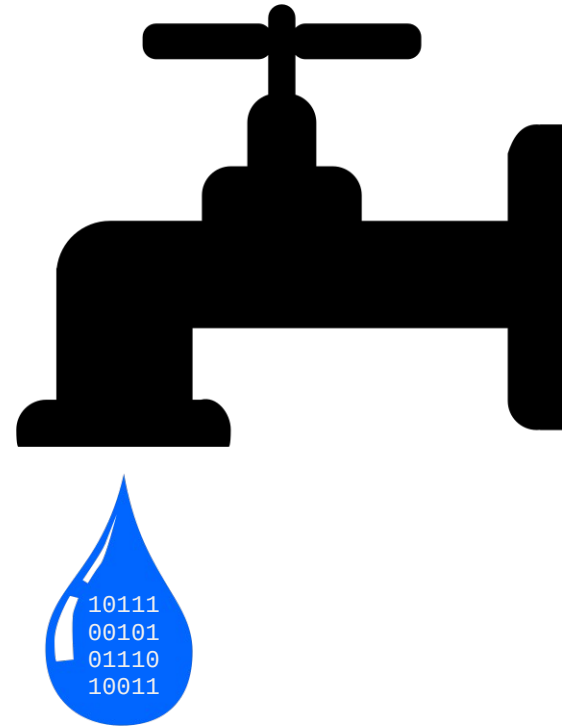


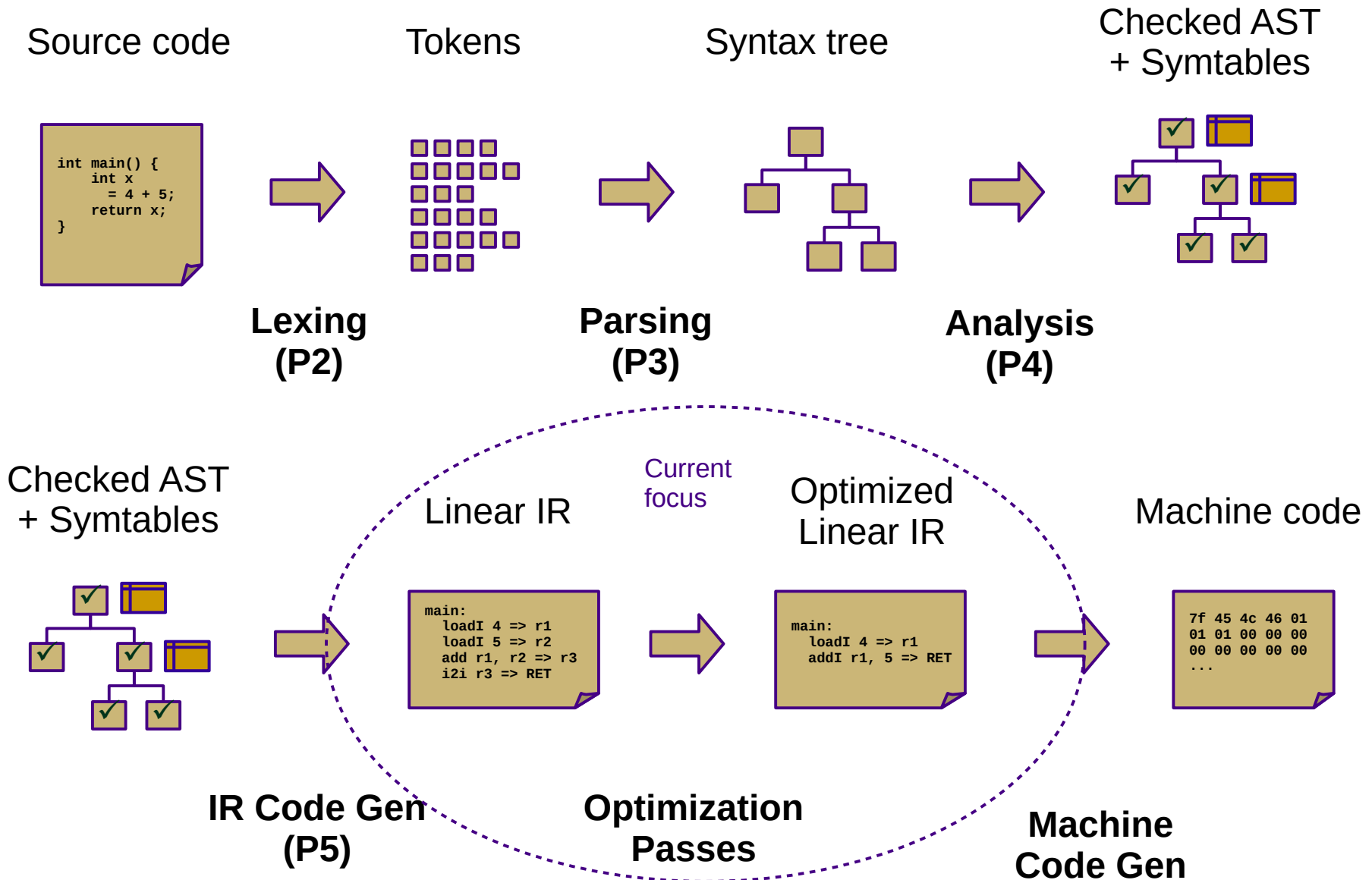
CS 432 Fall 2025

Mike Lam, Professor



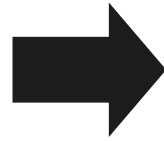
Data-Flow Analysis

Compilers

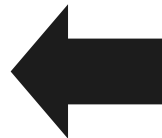


Optimization

```
int a;  
a = 0;  
while (a < 10) {  
    a = a + 1;  
}
```



```
loadI 0 => r1  
loadI 10 => r2  
l1:  
  cmp_LT r1, r2 => r4  
  cbr r4 => l2, l3  
l2:  
  addI r1, 1 => r1  
  jump l1  
l3:  
  storeAI r1 => [bp-4]
```



```
loadI 0 => r1  
storeAI r1 => [bp-4]  
l1:  
  loadAI [bp-4] => r2  
  loadI 10 => r3  
  cmp_LT r2, r3 => r4  
  cbr r4 => l2, l3  
l2:  
  loadAI [bp-4] => r5  
  loadI 1 => r6  
  add r5, r6 => r7  
  storeAI r7 => [bp-4]  
  jump l1  
l3:
```



```
loadI 10 => r1  
storeAI r1 => [bp-4]
```

Optimization is Hard

- **Problem:** it's hard to reason about all possible executions
 - Preconditions and inputs may differ
 - Optimizations should be correct and efficient in all cases
- Optimization tradeoff: **investment vs. payoff**
 - "Better than naïve" is fairly easy
 - "Optimal" is impossible
 - Real world: somewhere in between
 - Better speedups with more static analysis
 - Usually worth the added compile time
- Also: linear IRs (e.g., ILOC) don't expose control flow
 - This makes analysis and optimization difficult
 - Need to re-introduce some representation of possible control flow

Aside: Verifying Returns in P3

- It is tempting to try to verify that functions end with a return statement in P3, but this is not possible with a naive approach
- Consider cases like this:

```
def int foo(bool x)
{
    // other code here

    if (x) {
        return 5;
    } else {
        return 10;
    }
}
```

This is **guaranteed** to be **safe** (every path has a return statement) but requires non-trivial, non-local static analysis to verify (i.e., can't just check the last statement in the function)

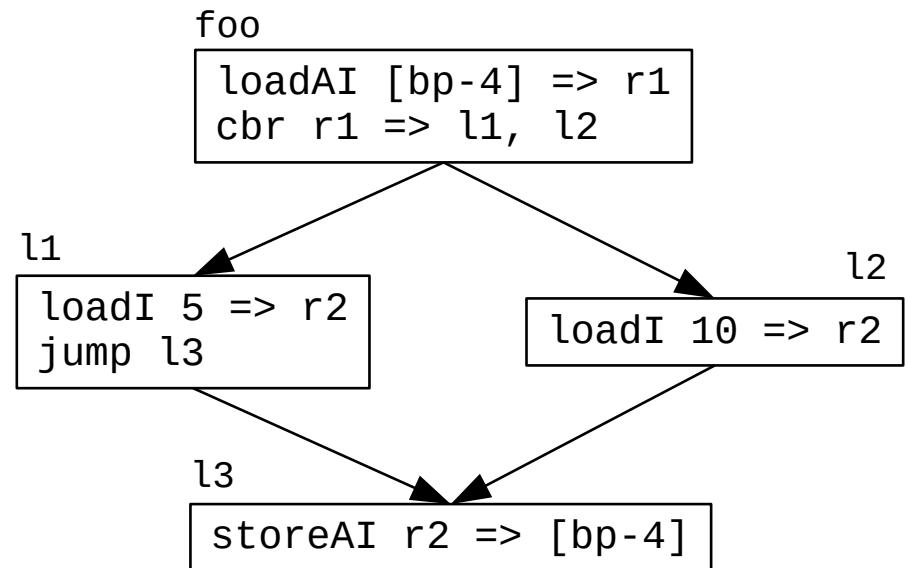
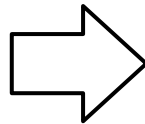
Control-Flow Graphs

- **Basic blocks**
 - "Maximal-length sequence of branch-free code"
 - "Atomic" sequences (instructions that always execute together)
- **Control-flow graph (CFG)** *(note overloaded acronym)*
 - Nodes/vertices for basic blocks
 - Edges for control transfer
 - Branch/jump instructions (explicit) or fallthrough (implicit)
 - p is a **predecessor** of q if there is a path from p to q
 - p is an **immediate** predecessor if there is an edge directly from p to q
 - Blocks without a predecessor are considered **entry** blocks
 - q is a **successor** of p if there is a path from p to q
 - q is an **immediate** successor if there is an edge directly from p to q
 - Blocks without a successor are considered **exit** blocks

Control-Flow Graphs

- Conversion: linear IR to CFG
 - Find **leaders** (initial instruction of a basic block) and build blocks
 - In ILOC, every call or jump target is a leader
 - Add edges between blocks based on jumps/branches and fall-through
 - Complicated by indirect jumps (none in our ILOC!)

```
foo:  
  loadAI [bp-4] => r1  
  cbr r1 => l1, l2  
l1:  
  loadI 5 => r2  
  jump l3  
l2:  
  loadI 10 => r2  
l3:  
  storeAI r2 => [bp-4]
```



Static CFG Analysis

- Single block analysis is easy, and trees are too
- General CFGs are harder
 - Which branch of a conditional will execute?
 - How many times will a loop execute?
- How do we handle this?
 - One method: iterative **data-flow analysis**
 - Simulate all possible paths through a region of code
 - “**Meet-over-all-paths**” conservative solution
 - **Meet operator** combines information across paths

Semilattices

- In general, a **semilattice** is a set of values L , special values \top (**top**) and \perp (**bottom**), and a **meet operator** \wedge such that
 - $a \geq b$ iff $a \wedge b = b$
 - $a > b$ iff $a \geq b$ and $a \neq b$
 - $a \wedge \top = a$ for all $a \in L$
 - $a \wedge \perp = \perp$ for all $a \in L$
- Partial ordering
 - Monotonic

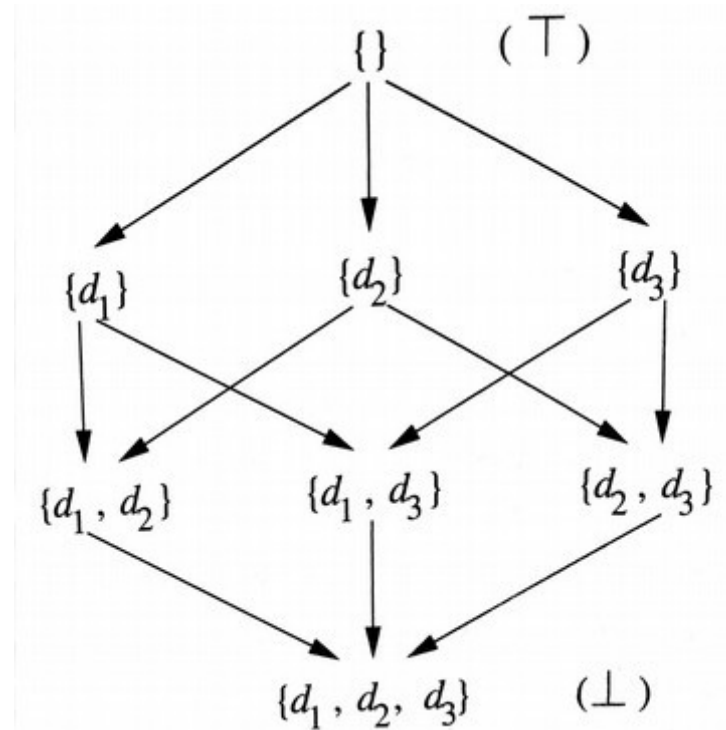
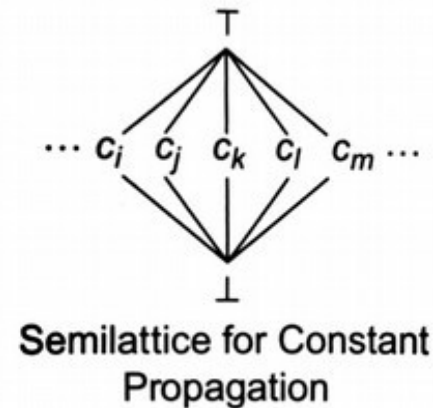


Figure 9.22 from Dragon book: semilattice of definitions using \cup (set union) as the meet operation

Constant propagation

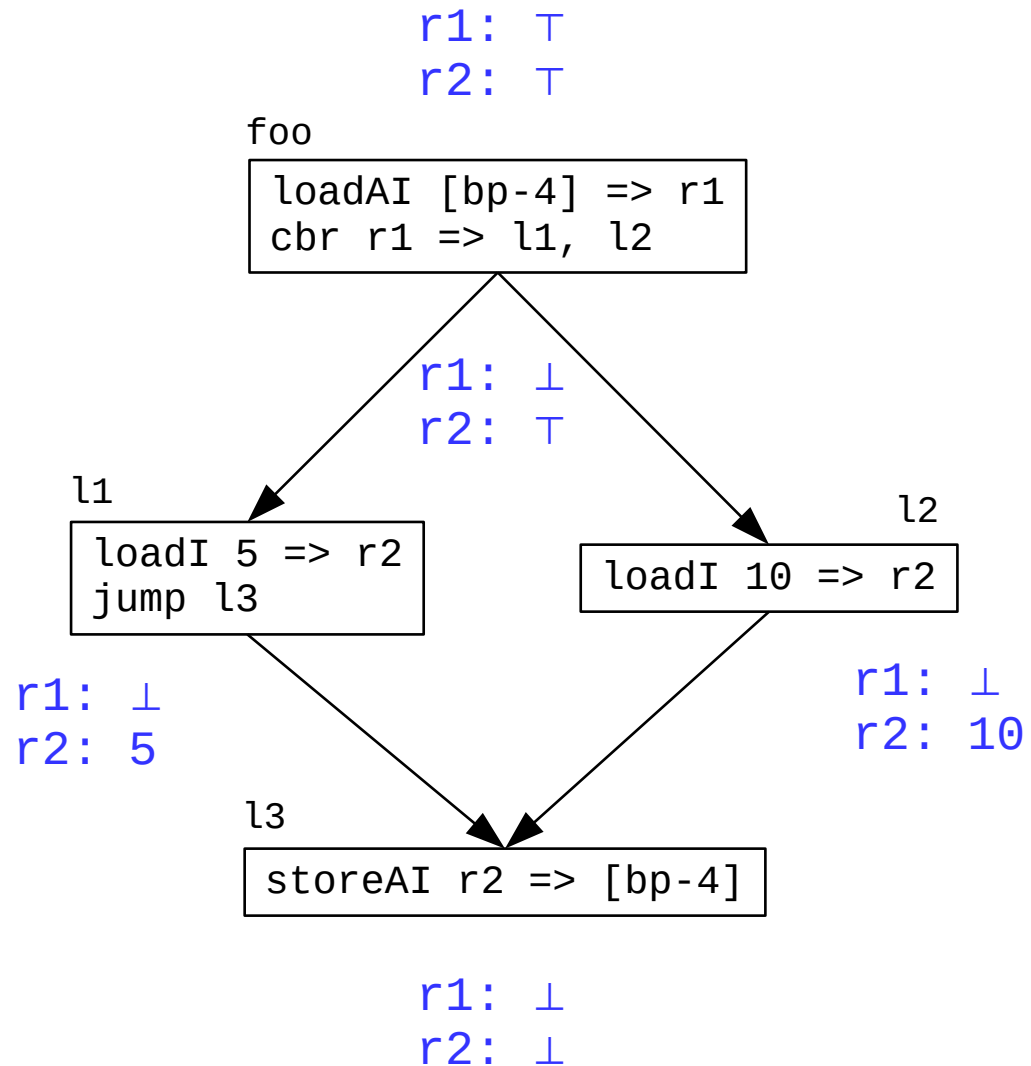
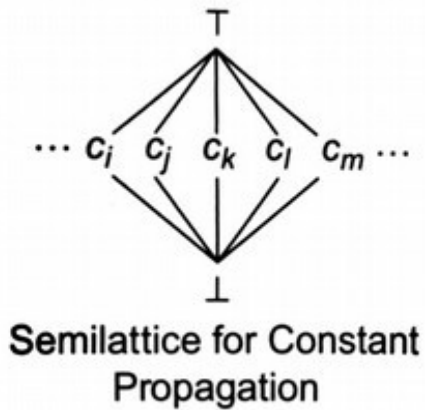
- For **sparse simple constant propagation** (SSCP), the lattice is very shallow

- $c_i \wedge \top = c_i$ for all c_i
- $c_i \wedge \perp = \perp$ for all c_i
- $c_i \wedge c_j = c_i$ if $c_i = c_j$
- $c_i \wedge c_j = \perp$ if $c_i \neq c_j$



- Basically: each SSA value is either unknown (\top), a known constant (c_i), or it is a variable (\perp)
 - Initialize to unknown (\top) for all SSA values
 - Interpret operations over lattice values (always lowering)
 - Propagate information until convergence

Constant propagation example



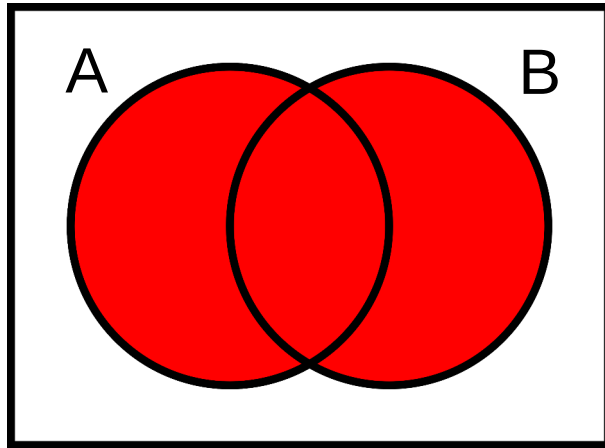
Data-Flow Analysis

- Define **properties** of interest for basic blocks
 - Usually **sets** of blocks, variables, definitions, etc.
- Define a **formula** for how properties change within a block
 - $F(B)$ is based on $F(A)$ where A is an **immediate** predecessor or successor of B
 - This is basically the *meet* operator for a particular problem
- Specify **initial information** for all blocks
 - Entry/exit blocks usually have special initial values
- Run an **iterative update** algorithm to propagate changes
 - Keep running until the properties converge for all basic blocks
- Key concept: **finite descending chain property**
 - Properties must be monotonically increasing or decreasing
 - Otherwise, termination is not guaranteed

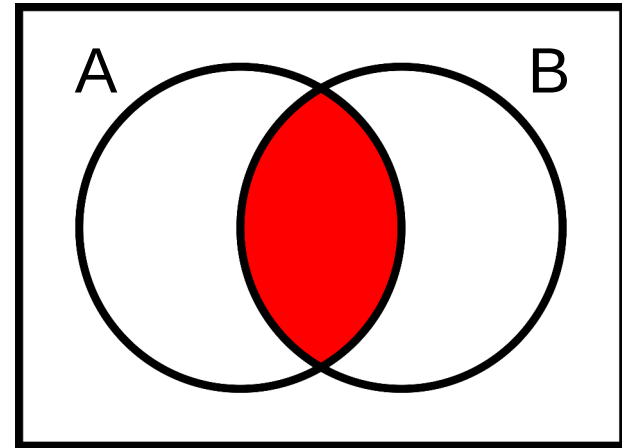
Data-Flow Analysis

- This kind of algorithm is called **fixed-point iteration**
 - It runs until it converges to a “fixed point”
- **Forward** vs. **backward** data-flow analysis
 - Forward: along graph edges (based on predecessors)
 - Backward: reverse of forward (based on successors)
- Particular data-flow analyses:
 - **Constant propagation**
 - **Dominance**
 - **Liveness**
 - Available expressions
 - Reaching definitions
 - Anticipable expressions

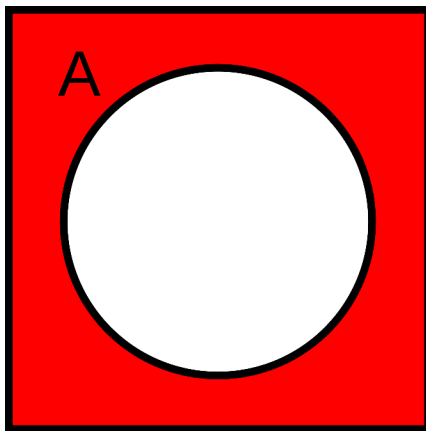
Review: Set Theory



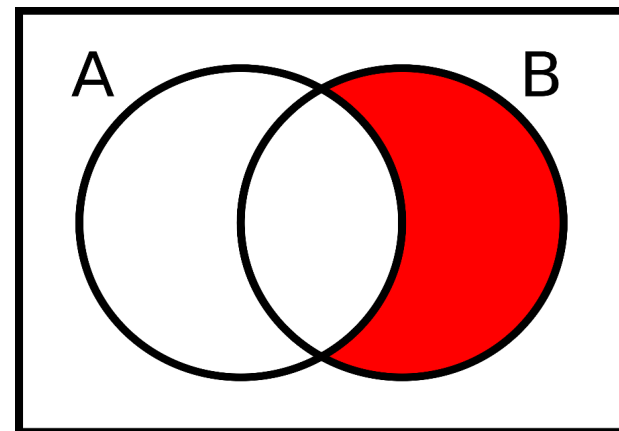
$$A \cup B$$



$$A \cap B$$



$$\bar{A}$$



$$B \cap \bar{A} = B - A$$

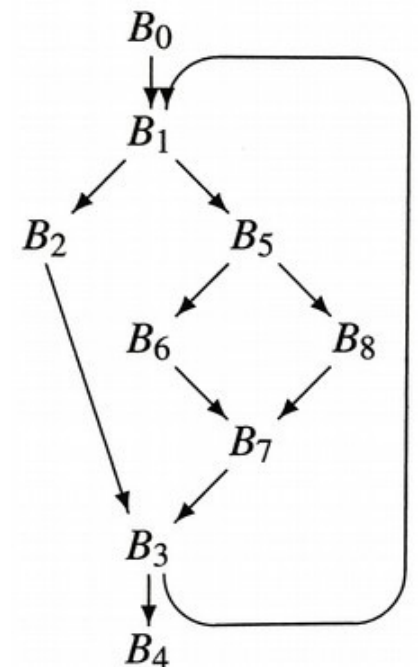
Dominance

- Block A **dominates** block B if A is on every path from the entry to B
 - Block A **immediately** dominates block B if there are no blocks between them
 - Block B **postdominates** block A if B is on every path from A to an exit
 - Every block both dominates and postdominates itself
- Simple dataflow analysis formulation
 - $preds(b)$ is the set of blocks that are immediate predecessors of block b
 - $Dom(b)$ is the set of blocks that dominate block b
 - intersection of Dom for all immediate predecessors (p in $preds(b)$)
 - $PostDom(b)$ is the set of blocks that postdominate block b
 - (similar definition using immediate successors)

Initial conditions: $Dom(\mathbf{entry}) = \{\mathbf{entry}\}$

$\forall b \neq \mathbf{entry}, Dom(b) = \{\mathbf{all blocks}\}$

Updates: $Dom(b) = \{b\} \cup \bigcap_{p \in preds(b)} Dom(p)$



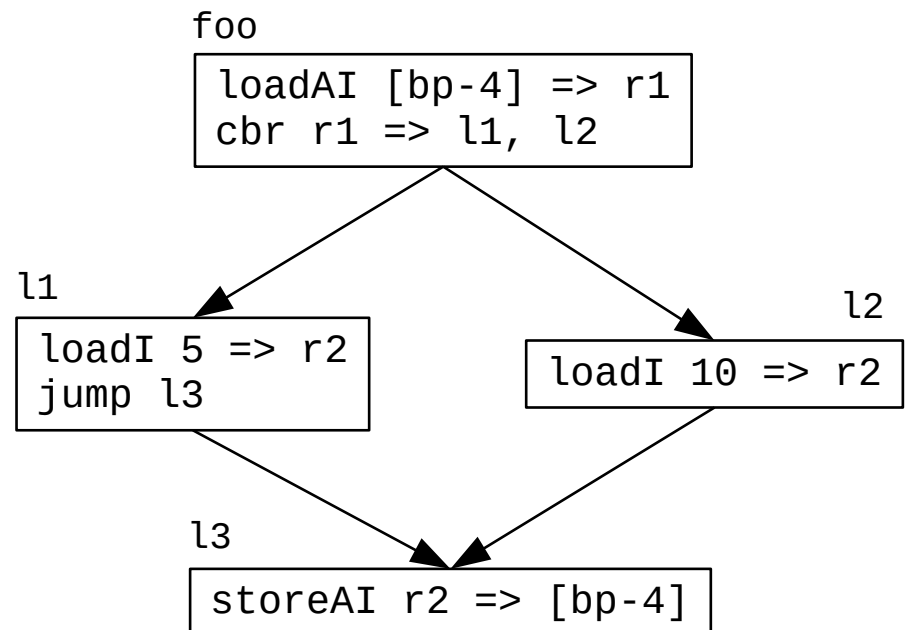
Dominance example

Initial conditions: $Dom(\mathbf{entry}) = \{\mathbf{entry}\}$

$\forall b \neq \mathbf{entry}, Dom(b) = \{\mathbf{all\ blocks}\}$

Updates: $Dom(b) = \{b\} \cup \bigcap_{p \in preds(b)} Dom(p)$

$Dom(\mathbf{foo}) = \{\mathbf{foo}\}$
 $Dom(\mathbf{l1}) = \{\mathbf{foo}, \mathbf{l1}\}$
 $Dom(\mathbf{l2}) = \{\mathbf{foo}, \mathbf{l2}\}$
 $Dom(\mathbf{l3}) = \{\mathbf{foo}, \mathbf{l3}\}$



Liveness

- Variable or register v is **live** at point p if there is a path from p to a use of v with no intervening assignment to v
 - Useful for finding uninitialized variables (live at function entry)
 - Useful for optimization (remove unused assignments)
 - Useful for register allocation (keep live vars in registers)
- Initial information: *UEVar* and *VarKill*
 - *UEVar*(B): variables/registers read from in B before any corresponding write in B
 - (“upwards exposed” variables/registers)
 - *VarKill*(B): variables/registers that are written to (“killed”) in B
- Textbook notation note: $X \cap \overline{Y} = X - Y$

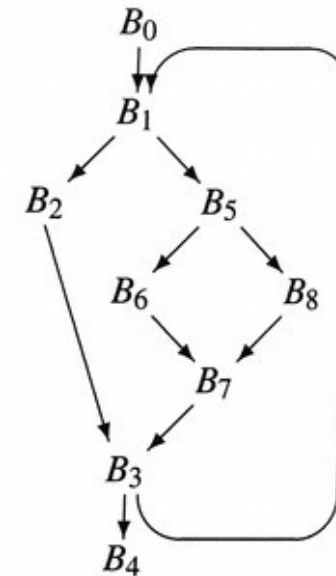
Initial conditions: $\forall b, \text{LiveOut}(b) = \emptyset$

Updates: $\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{UEVar}(s) \cup (\text{LiveOut}(s) - \text{VarKill}(s))$

Liveness example

B_0 : $i \leftarrow 1$
 $\rightarrow B_1$
 B_1 : $a \leftarrow \dots$
 $c \leftarrow \dots$
 $(a < c) \rightarrow B_2, B_5$
 B_2 : $b \leftarrow \dots$
 $c \leftarrow \dots$
 $d \leftarrow \dots$
 $\rightarrow B_3$
 B_3 : $y \leftarrow a + b$
 $z \leftarrow c + d$
 $i \leftarrow i + 1$
 $(i \leq 100) \rightarrow B_1, B_4$

B_4 : return
 B_5 : $a \leftarrow \dots$
 $d \leftarrow \dots$
 $(a \leq d) \rightarrow B_6, B_8$
 B_6 : $d \leftarrow \dots$
 $\rightarrow B_7$
 B_7 : $b \leftarrow \dots$
 $\rightarrow B_3$
 B_8 : $c \leftarrow \dots$
 $\rightarrow B_7$



(a) Code for the Basic Blocks

(b) Control-Flow Graph

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
UEVAR	\emptyset	\emptyset	\emptyset	$\{a, b, c, d, i\}$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
VARKILL	$\{i\}$	$\{a, c\}$	$\{b, c, d\}$	$\{y, z, i\}$	\emptyset	$\{a, d\}$	$\{d\}$	$\{b\}$	$\{c\}$

(c) Initial Information

$$\forall b, \text{LiveOut}(b) = \emptyset \quad \text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{UEVar}(s) \cup (\text{LiveOut}(s) - \text{VarKill}(s))$$

Alternative definition

- Define **LiveIn** as well as **LiveOut**
 - Two formulas for each basic block
 - Makes things a bit simpler to reason about
 - Separates change *within* block from change *between* blocks

$$\forall b, \text{LiveIn}(b) = \emptyset, \text{LiveOut}(b) = \emptyset$$

$$\text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b))$$

$$\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s)$$

Liveness example

$\forall b, \text{LiveIn}(b) = \emptyset, \text{LiveOut}(b) = \emptyset$

$\text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b))$

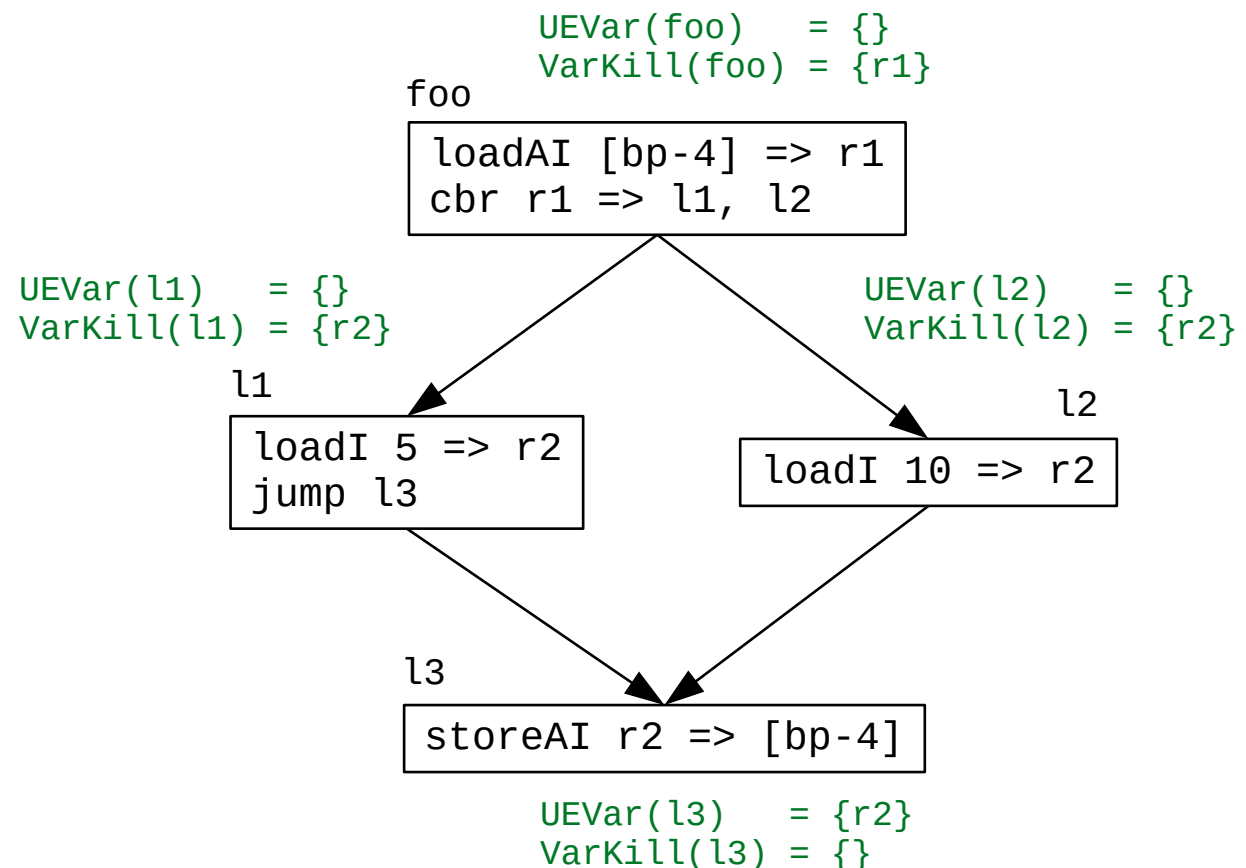
$\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s)$

$\text{LiveIn}(\text{foo}) = \{\}$
 $\text{LiveOut}(\text{foo}) = \{\}$

$\text{LiveIn}(l1) = \{\}$
 $\text{LiveOut}(l1) = \{r2\}$

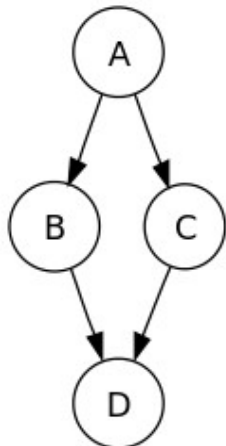
$\text{LiveIn}(l2) = \{\}$
 $\text{LiveOut}(l2) = \{r2\}$

$\text{LiveIn}(l3) = \{r2\}$
 $\text{LiveOut}(l3) = \{\}$



Block orderings

- Forwards dataflow analyses converge faster with **reverse postorder** processing of CFG blocks
 - Visit as many of a block's predecessors as possible before visiting it
 - Strict reversal of normal postorder traversal
 - Similar to concept of topological sorting on DAGs
 - NOT EQUIVALENT to preorder traversal!
 - Backwards analyses should use reverse postorder on reverse CFG



Depth-first search:

A, B, D, B, A, C, A (left first)
A, C, D, C, A, B, A (right first)

Valid *preorderings*:

A, B, D, C (left first)
A, C, D, B (right first)

Valid *postorderings*:

D, B, C, A (left first)
D, C, B, A (right first)

Valid *reverse postorderings*:

A, C, B, D
A, B, C, D

Summary

$$\text{Dom}(\mathbf{entry}) = \{\mathbf{entry}\}$$

$$\forall b \neq \mathbf{entry}, \text{Dom}(b) = \{\mathbf{all blocks}\}$$

Dominance

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

$$\forall b, \text{LiveOut}(b) = \emptyset$$

$$\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{UEVar}(s) \cup (\text{LiveOut}(s) - \text{VarKill}(s))$$

**Liveness
(EAC version)**

$$\forall b, \text{LiveIn}(b) = \emptyset, \text{LiveOut}(b) = \emptyset$$

$$\text{LiveIn}(b) = \text{UEVar}(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b))$$

**Liveness
(Dragon version)**

$$\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s)$$

Summary (example)

$$\text{Dom}(\mathbf{entry}) = \{ \mathbf{entry} \}$$

$$\forall b \neq \mathbf{entry}, \text{Dom}(b) = \{ \mathbf{all blocks} \}$$

Dominance

$$\text{Dom}(b) = \{ b \} \cup \bigcup_{p \in \text{preds}(b)} \text{Dom}(p)$$

$UEVar(B)$: variables read in B before any corresponding write in B

(“upwards exposed” variables)

$VarKill(B)$: variables that are written to (“killed”) in B

$$\forall b, \text{LiveIn}(b) = \emptyset, \text{LiveOut}(b) = \emptyset$$

$$\text{LiveIn}(b) = UEVar(b) \cup (\text{LiveOut}(b) - \text{VarKill}(b))$$

Liveness
(Dragon version)

$$\text{LiveOut}(b) = \bigcup_{s \in \text{succs}(b)} \text{LiveIn}(s)$$

Summary (lab)

$$Dom(\mathbf{entry}) = \{\mathbf{entry}\}$$

$$\forall b \neq \mathbf{entry}, Dom(b) = \{\mathbf{all blocks}\}$$

Dominance

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

$UEVar(B)$: registers read in B before any corresponding write in B

(“upwards exposed” registers)

$VarKill(B)$: registers that are written to (“killed”) in B

$$\forall b, LiveIn(b) = \emptyset, LiveOut(b) = \emptyset$$

$$LiveIn(b) = UEVar(b) \cup (LiveOut(b) - VarKill(b))$$

Liveness
(Dragon version)

$$LiveOut(b) = \bigcup_{s \in succs(b)} LiveIn(s)$$