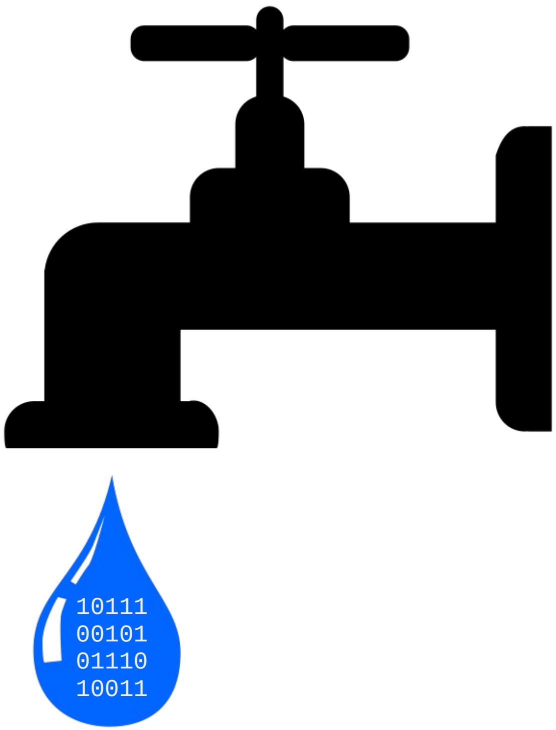
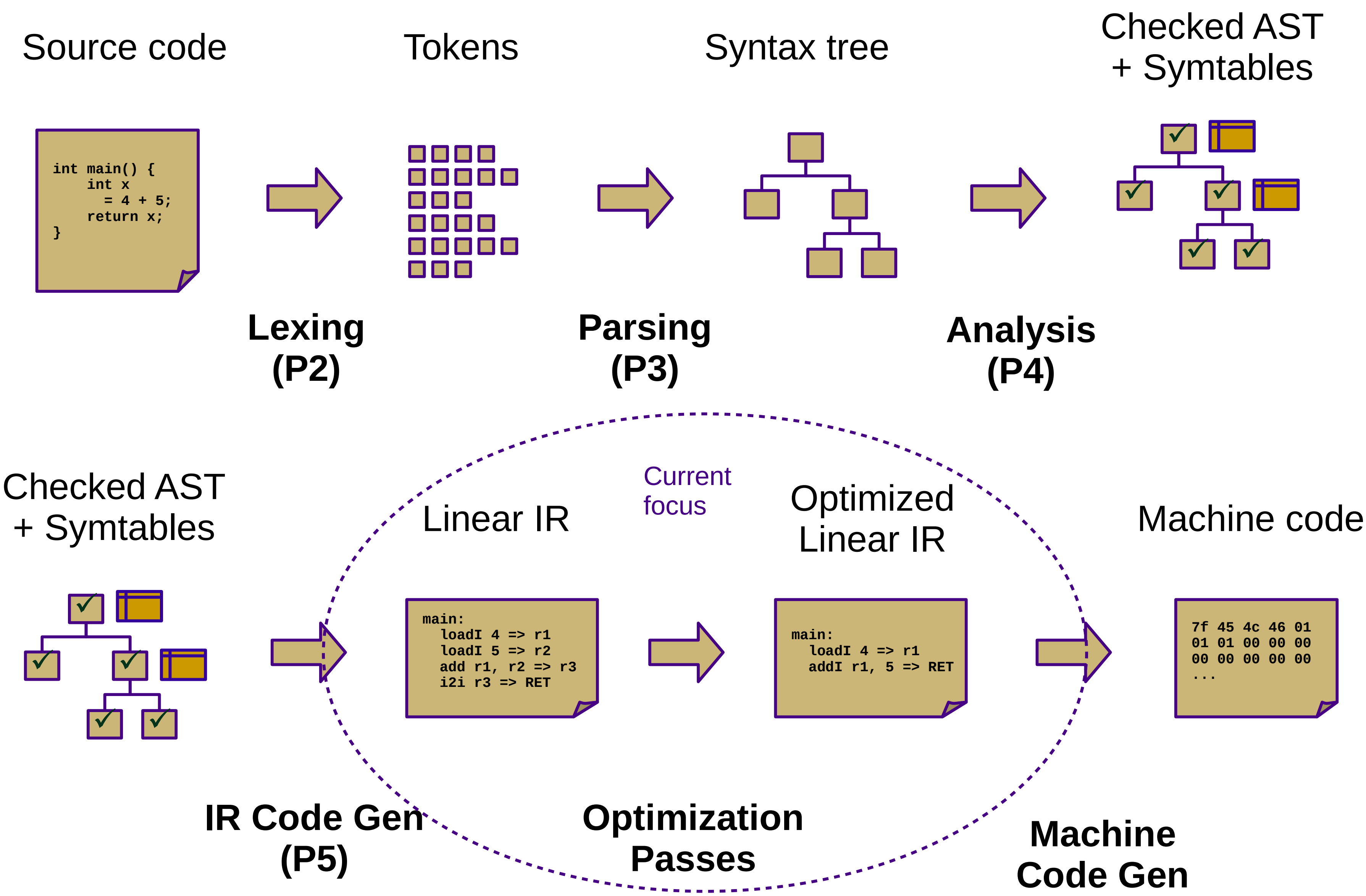
CS 432
Fall 2022

Mike Lam, Professor



Data-Flow Analysis

Compilers


Source code
Tokens
Syntax tree
Checked AST
+ Symtables
int main() {
int x
= 4 + 5;
return x;
}
Lexing
(P2)
Parsing
(P3)
Analysis
(P4)
Checked AST
+ Symtables
Linear IR
Current
focus
Optimized
Linear IR
Machine code
main:
loadI 4 => r1
loadI 5 => r2
add r1, r2 => r3
i2i r3 => RET
main:
loadI 4 => r1
addI r1, 5 => RET
7f 45 4c 46 01
01 01 00 00 00
00 00 00 00 00
...
IR Code Gen
(P5)
Optimization
Passes
Machine
Code Gen

Optimization

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

```
  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 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 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 explicitly expose control flow
  - This makes analysis and optimization difficult

Control-Flow Graphs

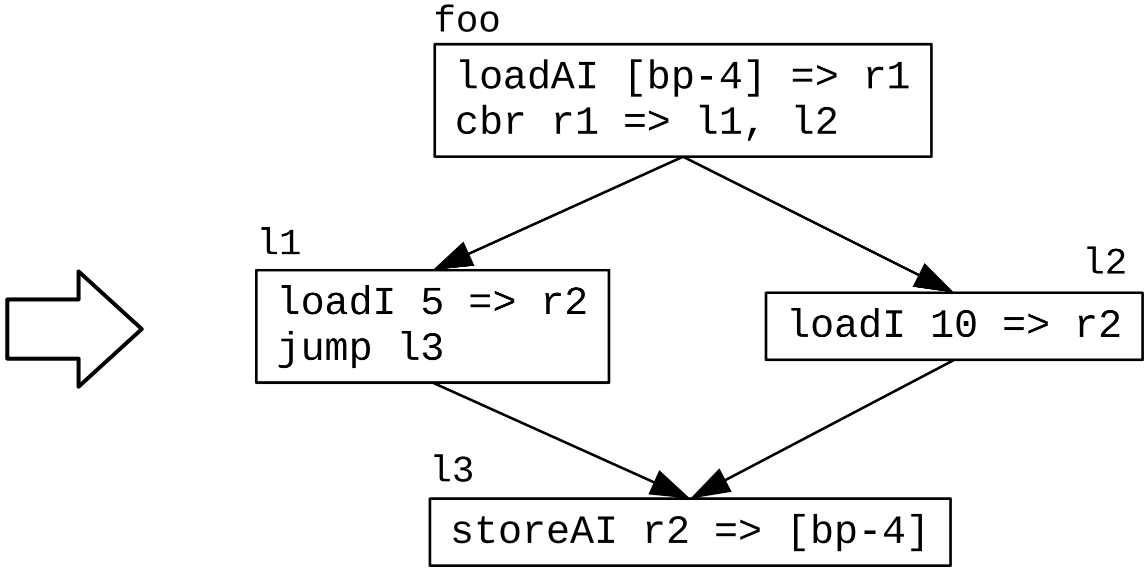
- Basic blocks
  - "Maximal-length sequence of branch-free code"
  - "Atomic" sequences (instructions that always execute together)

- Control-flow graph (CFG)
  - 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
    - 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

Control-Flow Graphs

- Conversion: linear IR to CFG
  - Find leaders (initial instruction of a basic block) and build blocks
    - Every call or jump target is a leader
  - Add edges between blocks based on jumps/branches and fallthrough
  - 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) and ⊥ (bottom), and a meet operator ^ such that
  - a ≥ b iff a ^ b = b
  - a > b iff a ≥ b and a ≠ b
  - a ^ ⊤ = a for all a ∈ L
  - a ^ ⊥ = ⊥ for all a ∈ L
- Partial ordering
  - Monotonic

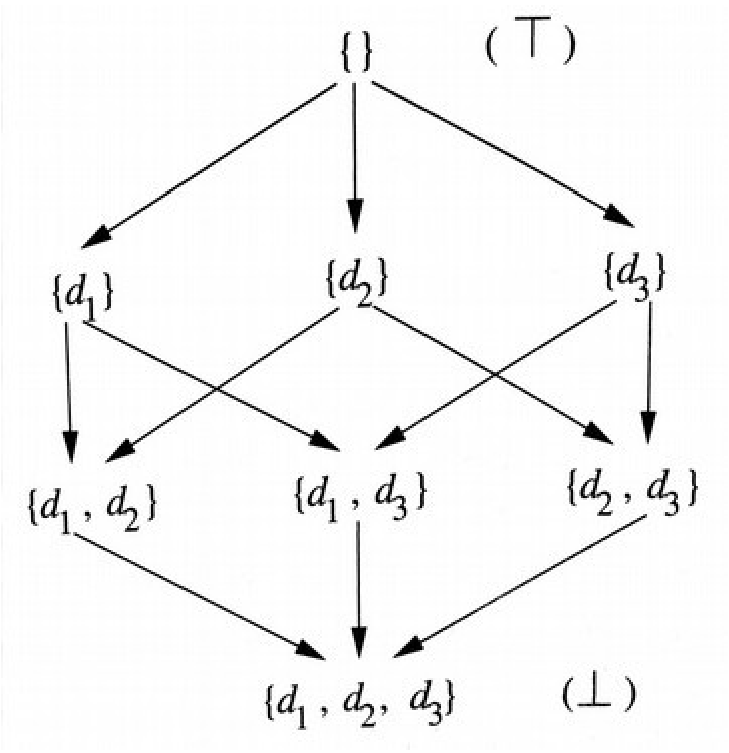
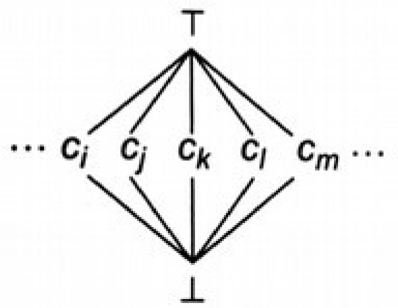


Figure 9.22 from Dragon book: semilattice of definitions using U (set union) as the meet operation

Constant propagation

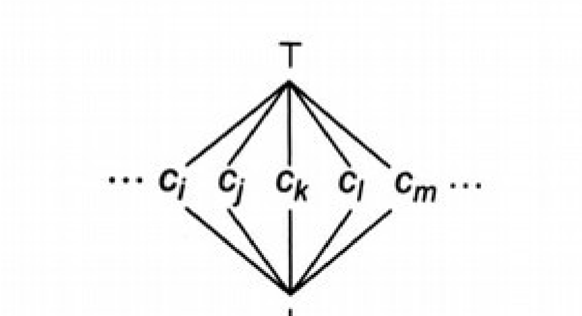
- For sparse simple constant propagation (SSCP), the lattice is very shallow
  - $c_i \wedge \top = c_i$ for all c_i
  - $c_i \wedge \bot = \bot$ for all c_i
  - $c_i \wedge c_j = c_i$ if $c_i = c_j$
  - $c_i \wedge c_j = \bot$ if $c_i \neq c_j$



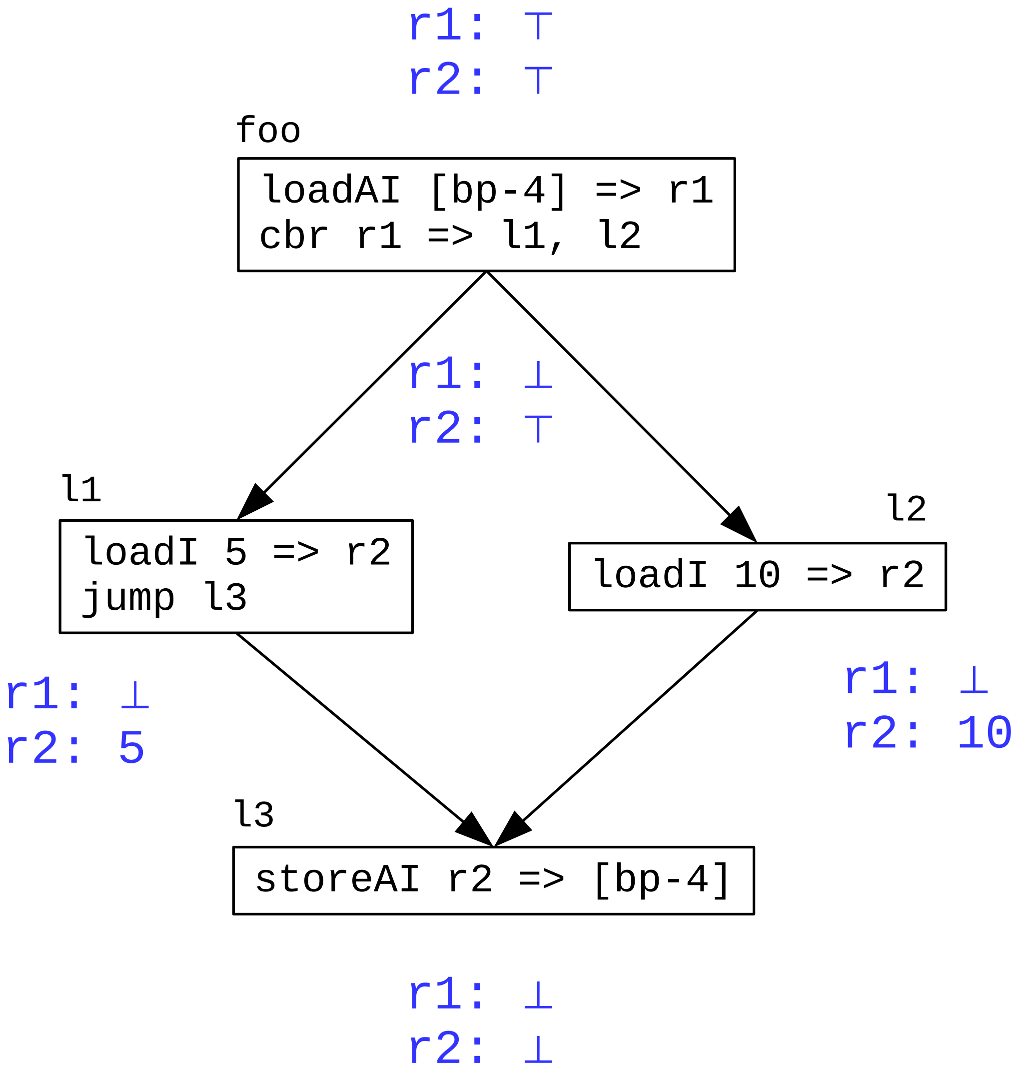
Semilattice for Constant Propagation

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

Constant propagation example



Semilattice for Constant Propagation



Data-Flow Analysis

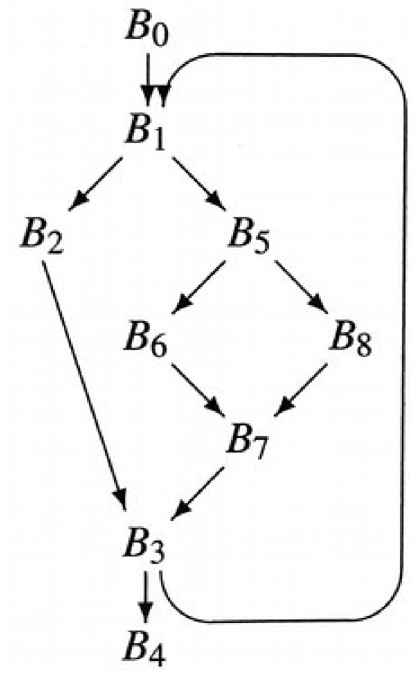
- Define properties of interest for basic blocks
  - Usually **sets** of blocks, variables, definitions, etc.
- Define a formula for how those properties change within a block
  - F(B) is based on F(A) where A is a predecessor or successor of B
  - This is basically the *meet* operator for a particular 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

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 predecessors of block b
  - *Dom*(b) is the set of blocks that dominate block b
    - intersection of *Dom* for all immediate predecessors
  - *PostDom*(b) is the set of blocks that postdominate block b
    - (similar definition using *succs*(b))



$$\textit{Initial conditions}: \quad Dom(\textbf{\textit{entry}}) = \{\textbf{\textit{entry}}\}$$
$$\forall\, b \neq \textbf{\textit{entry}},\quad Dom(b) = \{\textbf{\textit{all blocks}}\}$$

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

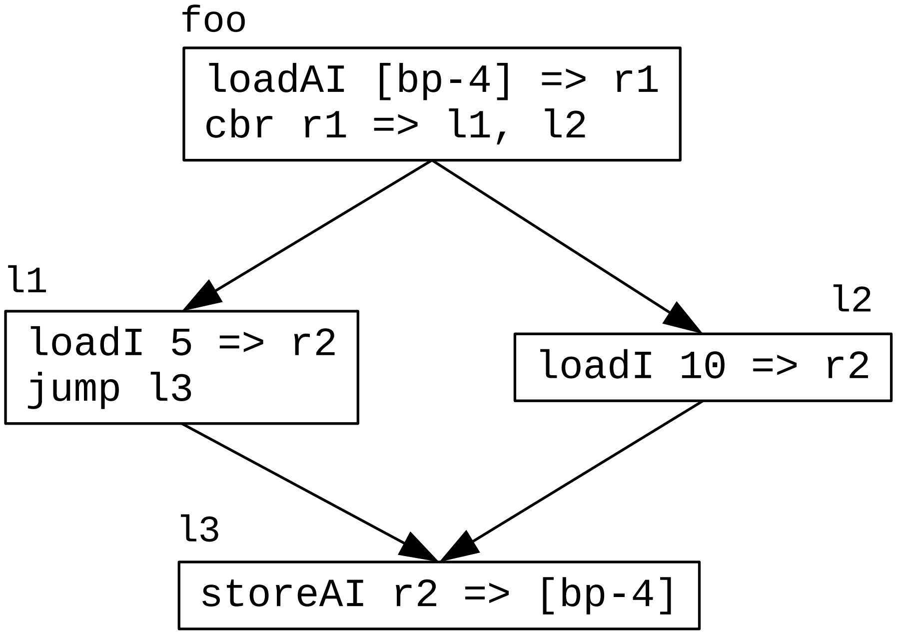
Dominance example

$$\textit{Initial conditions}: \quad Dom(\textbf{entry}) = \{\textbf{entry}\}$$

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

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

```
Dom(foo) = {foo}
Dom(l1)  = {foo, l1}
Dom(l2)  = {foo, l2}
Dom(l3)  = {foo, l3}
```



Liveness

- Variable *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 used in B before any redefinition in B
    - (“upwards exposed” variables)
  - *VarKill*(B): variables that are defined (“killed”) in B
- Textbook notation note: $X \cap \overline{Y} = X - Y$

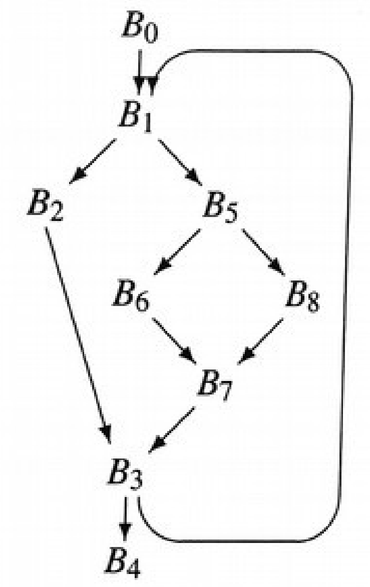
$$\textit{Initial conditions}: \ \forall b, \ \ LiveOut(b) = \emptyset$$

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

Liveness example

```
B0:  i ← 1
     → B1
B1:  a ← ···
     c ← ···
     (a < c) → B2,B5
B2:  b ← ···
     c ← ···
     d ← ···
     → B3
B3:  y ← a + b
     z ← c + d
     i ← i + 1
     (i ≤ 100) → B1,B4
B4:  return
B5:  a ← ···
     d ← ···
     (a ≤ d) → B6,B8
B6:  d ← ···
     → B7
B7:  b ← ···
     → B3
B8:  c ← ···
     → B7
```

(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	Ø	Ø	Ø	{a,b,c,d,i}	Ø	Ø	Ø	Ø	Ø
VARKILL	{i}	{a,c}	{b,c,d}	{y,z,i}	Ø	{a,d}	{d}	{b}	{c}

(c) Initial Information

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

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

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

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

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

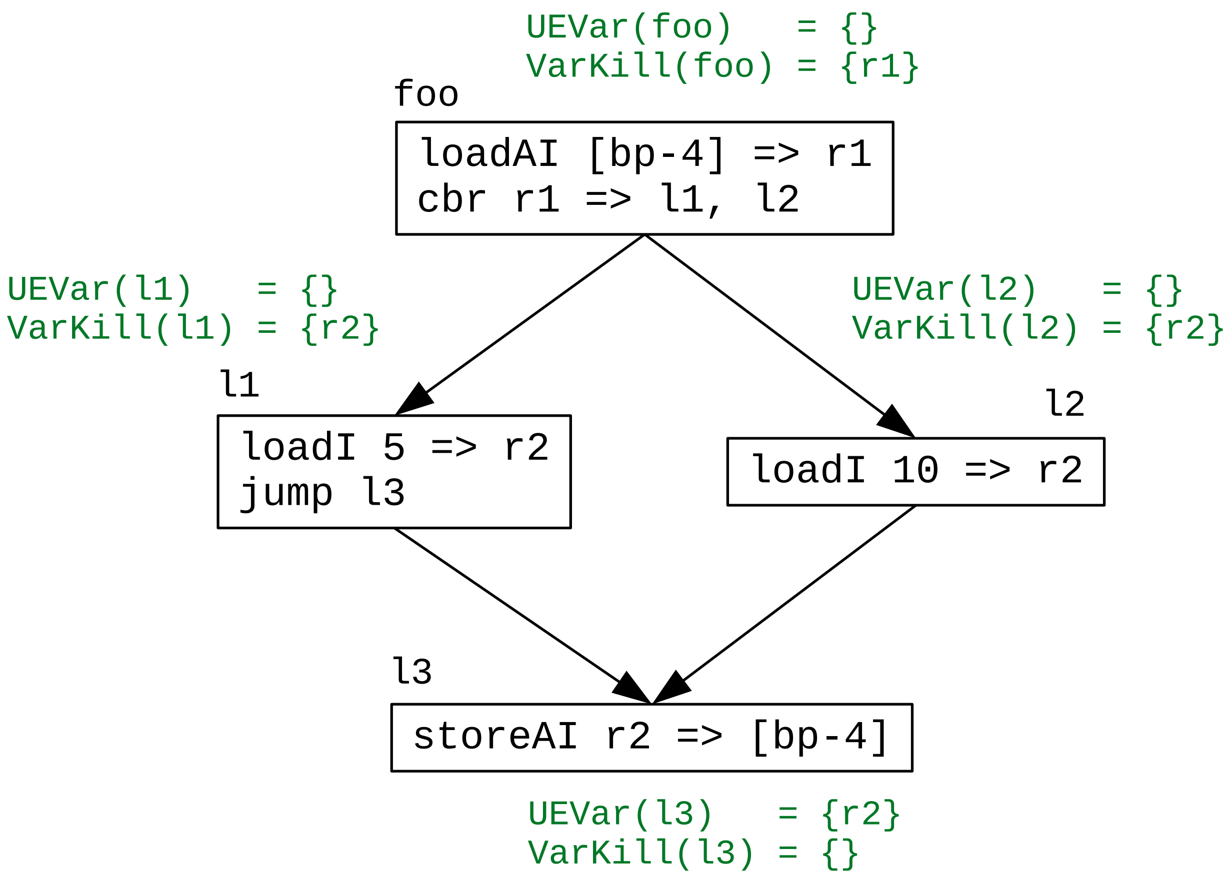
Liveness example

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

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

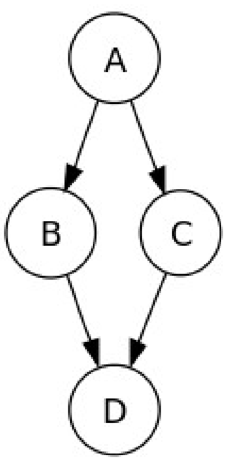
$$LiveOut(b) = \bigcup_{s \in succs(b)} LiveIn(s)$$

```
LiveIn (foo) = {}
LiveOut(foo) = {}

LiveIn (l1)  = {}
LiveOut(l1)  = {r2}

LiveIn (l2)  = {}
LiveOut(l2)  = {r2}

LiveIn (l3)  = {r2}
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 that block
  - 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 *postorderings*:

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

Valid *preorderings*:

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

Valid *reverse postorderings*:

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

Summary

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

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

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

Dominance

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

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

Liveness
(EAC version)

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

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

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

Liveness
(Dragon version)