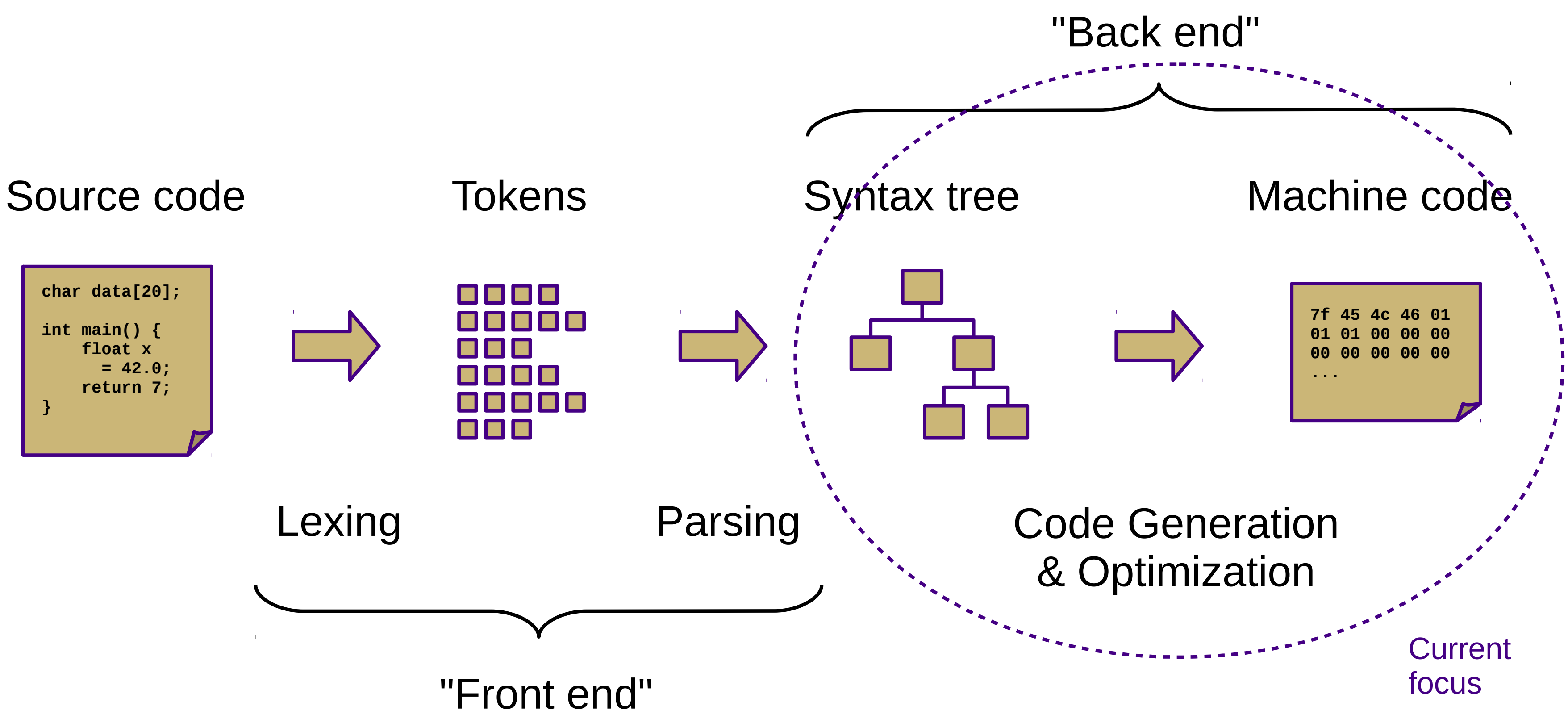
CS 432
Fall 2016

Mike Lam, Professor

Data-Flow Analysis

Compilers


"Back end"
Source code
Tokens
Syntax tree
Machine code
char data[20];
int main() {
float x
= 42.0;
return 7;
}
7f 45 4c 46 01
01 01 00 00 00
00 00 00 00 00
...
Lexing
Parsing
Code Generation
& Optimization
Current
focus
"Front end"

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
  - Consider this code:

```
int *p; cin >> p; *p = 42;
```

- 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

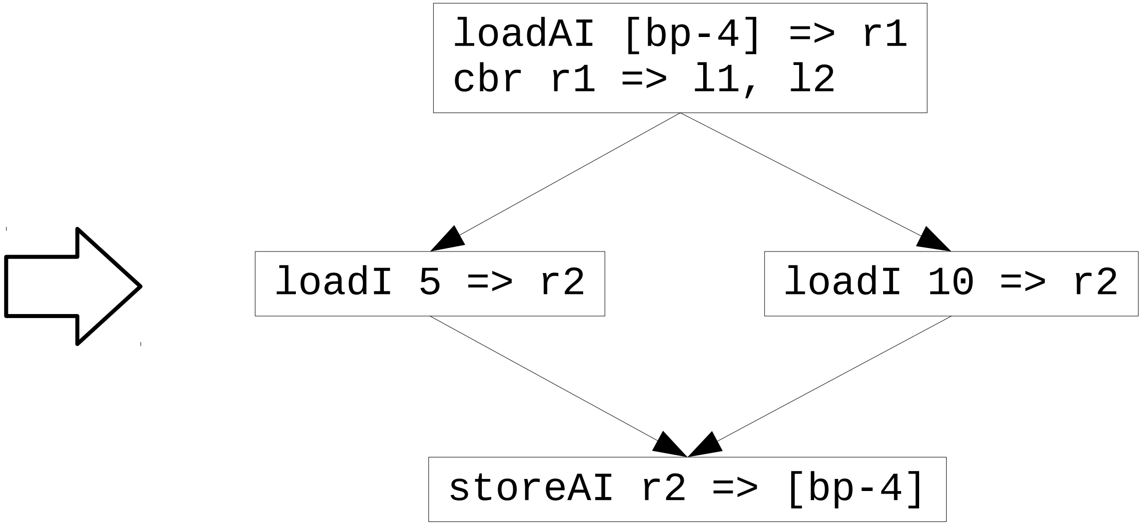
Control-Flow Graphs

- Linear IRs (e.g., ILOC) don't easily expose control flow
  - This makes analysis and optimization difficult
- Basic blocks
  - "Maximal-length sequence of branch-free code"
  - "Atomic" code sequences
  - Instructions that always execute together
- Control-flow graph (CFG)
  - Nodes/vertices for basic blocks
  - Edges for control transfer
    - Branches (explicit) or fallthrough (implicit)
    - p is a predecessor of q if there is a path from p to q
    - q is a successor of p if there is a path 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 branches and fallthrough
  - Complicated by jump-to-address instructions

```
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
- Trees are also relatively easy
  - No path merges or loops
- 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

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
- Gather initial information to help calculate property changes
  - Helper functions g(B) that can be used in F(B)
- Run an iterative update algorithm to propagate changes
  - Keep running until the properties converge for all basic blocks
  - More efficient w/ reverse postorder traversal: visit predecessors first
- Key concept: finite descending chain property
  - Properties must be monotonically increasing or decreasing
  - Otherwise, termination is not guaranteed

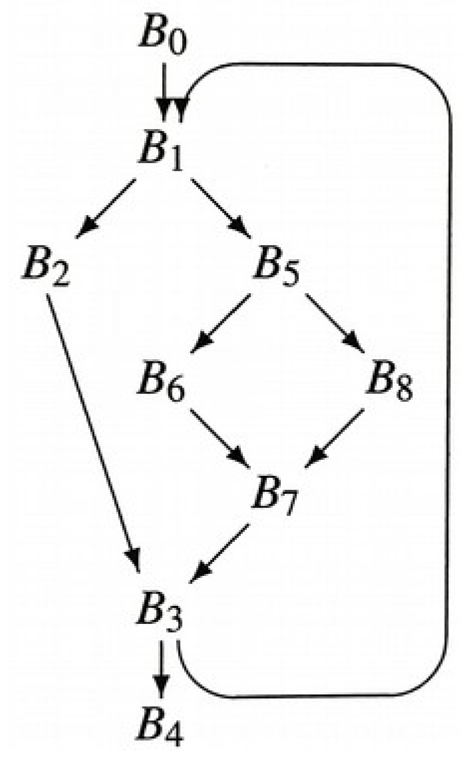
Data-Flow Analysis

- This kind of algorithm is called a fixed-point algorithm
  - 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)
- Types of data-flow analysis
  - Dominance
  - Liveness
  - Available expressions
  - Reaching definitions
  - Anticipable expressions

Dominance

- Block A dominates block B if A lies on every path from the entry block to B
  - Conversely, B postdominates block A if B lies on every path from A to any exit

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



Liveness

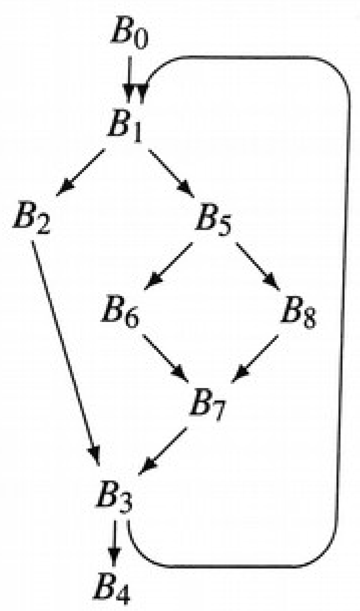
- Variable *v* is live at point *p* if there is a path from *p* to a use of *v* with no intervening assignment to *v*
  - Useful for finding uninitialized variables (live at function entry)
  - Useful for optimization (remove unused assignments)
  - Useful for register allocation (keep live vars in registers)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

$$LiveOut(n) = \bigcup_{m \in succs(n)} \left(UEVar(m) \cup \left(LiveOut(m) - VarKill(m) \right) \right)$$

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	∅	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	∅
VARKILL	{i}	{a,c}	{b,c,d}	{y,z,i}	∅	{a,d}	{d}	{b}	{c}

(c) Initial Information

$$LiveOut(n) = \bigcup_{m \in succs(n)} \left(UEVar(m) \cup \left(LiveOut(m) - VarKill(m) \right) \right)$$

Alternative definition

- Define LiveIn as well as LiveOut
  - Two formulas for each basic block
  - Makes things a bit simpler to reason about

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

$$LiveOut(n) = \bigcup_{m \in succs(n)} [LiveIn(m)]$$