Dataflow Analysis

Lecture 3: CPEN 400P

Karthik Pattabiraman, UBC

Outline

Goals of dataflow analysis

Dominators and Dominator sets: DFA-1

Live-Out variable analysis: DFA-2

Other problems using DFA

Generalized framework for DFA

Data-flow Analysis - 1

Definition

Data-flow analysis is a collection of techniques for *compile-time* reasoning about the *run-time* flow of values

- **We use the results of DFA to prove safety & identify opportunities**
  - **Not an end unto itself**
- **Almost always involves building a graph**
  - **Control-flow graph, call graph, or derivatives thereof**

Dataflow Analysis - 2

- **Usually formulated as a set of *simultaneous equations***
  - **Sets attached to nodes and edges**
  - **Often use sets with a lattice or semilattice structure**
- **Desired result is usually *meet over all paths* solution**
  - **“What is true on every path from the entry?”**
  - **“Can this happen on any path from the entry?”**

Outline

Goals of dataflow analysis

Dominators and Dominator sets: DFA-1

Live-Out variable analysis: DFA-2

Other problems using DFA

Generalized framework for DFA

Dominators

Definitions

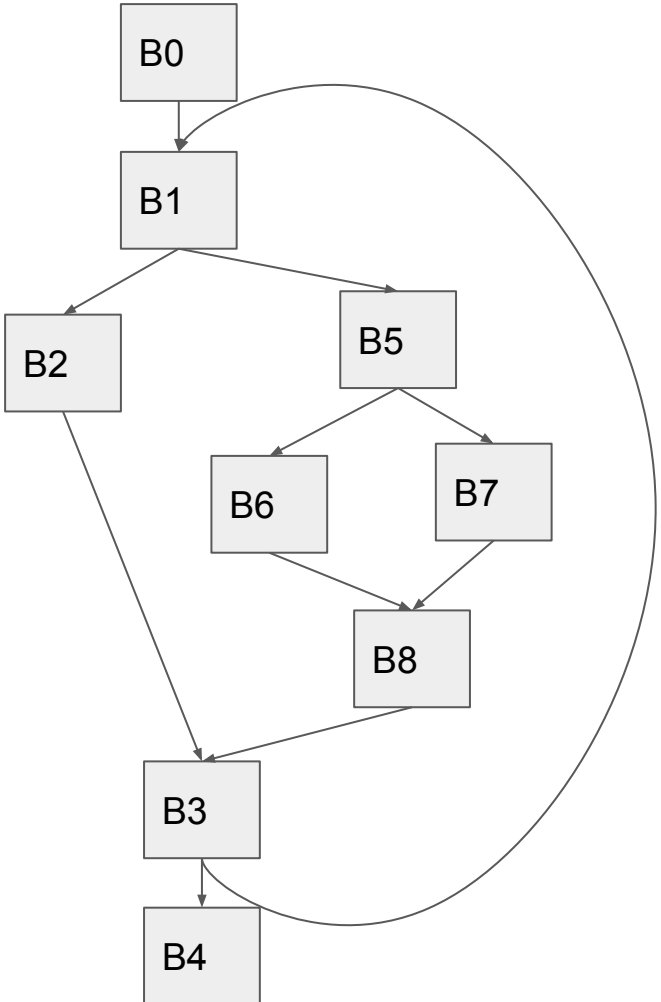
> ***x*** **dominates** ***y*** **if and only if every path from the entry of the control-flow graph to the node for** ***y*** **includes** ***x***

- **By definition, *x* dominates *x***
- **The first entry node of a procedure dominates every block in it**
- **We associate a DOM set with each node**
- **$|DOM(x)| \geq 1$**

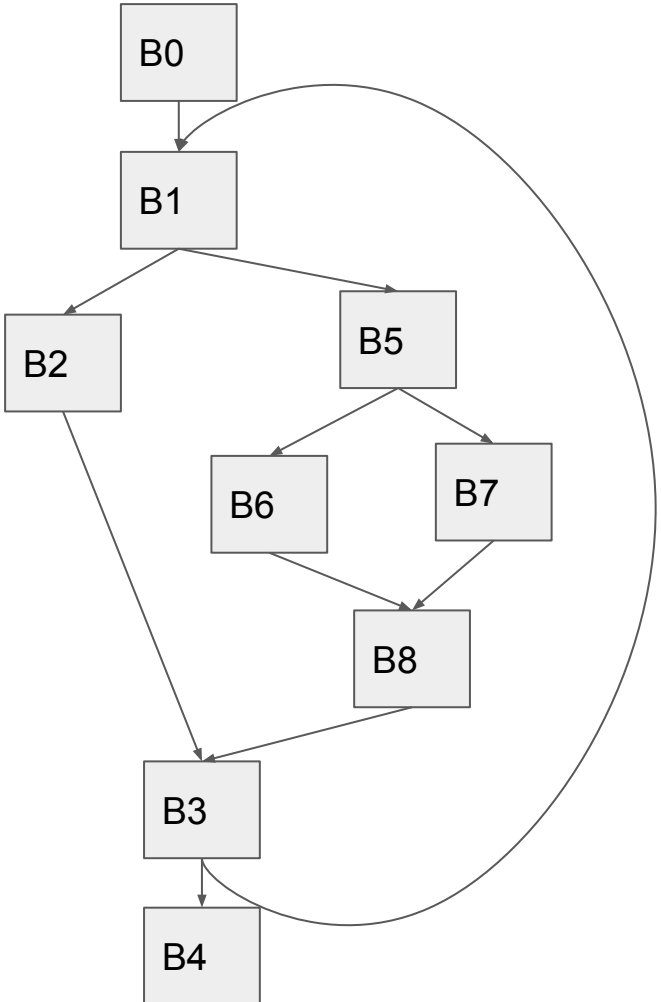
Uses of Dominators

Dominators have many uses in analysis & transformation

- **Finding loops**
- **Building SSA form**
- **Making code motion decisions**



Basic Block	Dominator Set
B0	
B1	
B2	
B3	
B4	
B5	
B6	
B7	
B8	



Basic Block	Dominator Set
B0	{ 0 }
B1	{ 0, 1 }
B2	{ 0, 1, 2 }
B3	{ 0, 1, 3 }
B4	{ 0, 1, 3, 4 }
B5	{ 0, 1, 5 }
B6	{ 0, 1, 5, 6 }
B7	{ 0, 1, 5, 7 }
B8	{ 0, 1, 5, 8 }

How does one compute Dominators using DFA ?

Think recursive algorithm over a set

What’s the starting node and it’s dominator ?

Every node dominates itself

If I know the dominators of all the predecessors of a node, how do I compute the dominator set of the node ?

Computing Dominators

- A node *n* dominates *m* iff *n* is on every path from n_0 to *m*
  - Every node dominates itself
  - *All of m's predecessors need to be dominated by n*

$$\text{DOM}(n_0) = \{ n_0 \}$$

$$\text{DOM}(n) = \{ n \} \cup (\cap_{p \in preds(n)} \text{DOM}(p))$$

Initially, DOM(n) = *N*, $\forall\ n \neq n_0$

Computing Dominators

- **These simultaneous set equations define a simple problem in data-flow analysis**
- **Equations have a unique fixed point solution**
- **An iterative fixed-point algorithm will solve them quickly**

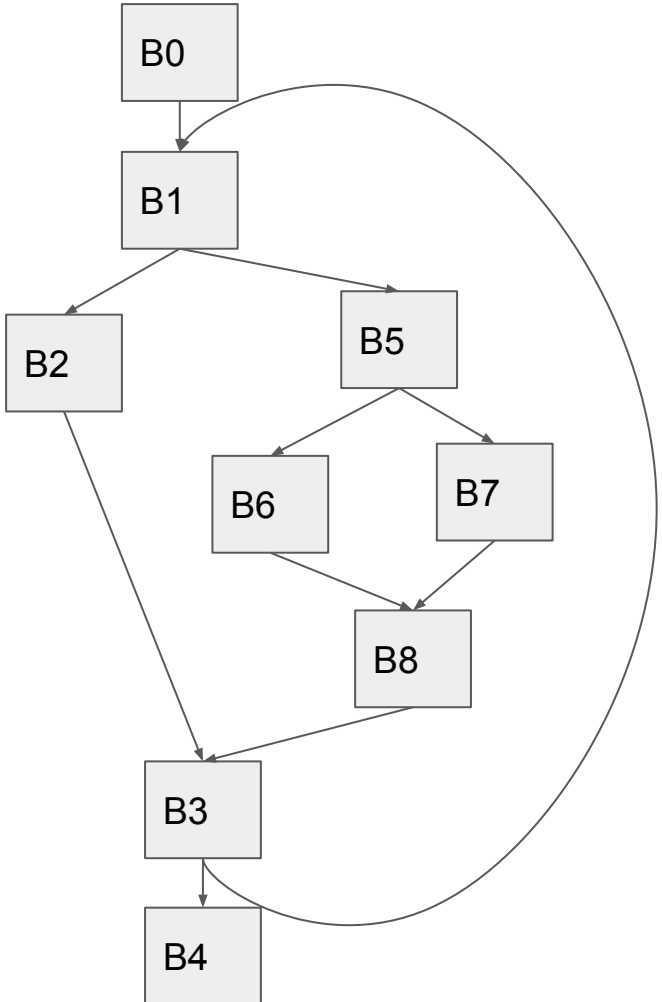
$$\text{DOM}(n_0) = \{ n_0 \}$$

$$\text{DOM}(n) = \{ n \} \cup (\cap_{p \in preds(n)} \text{DOM}(p))$$

Iterative fixed point algorithm for dominators

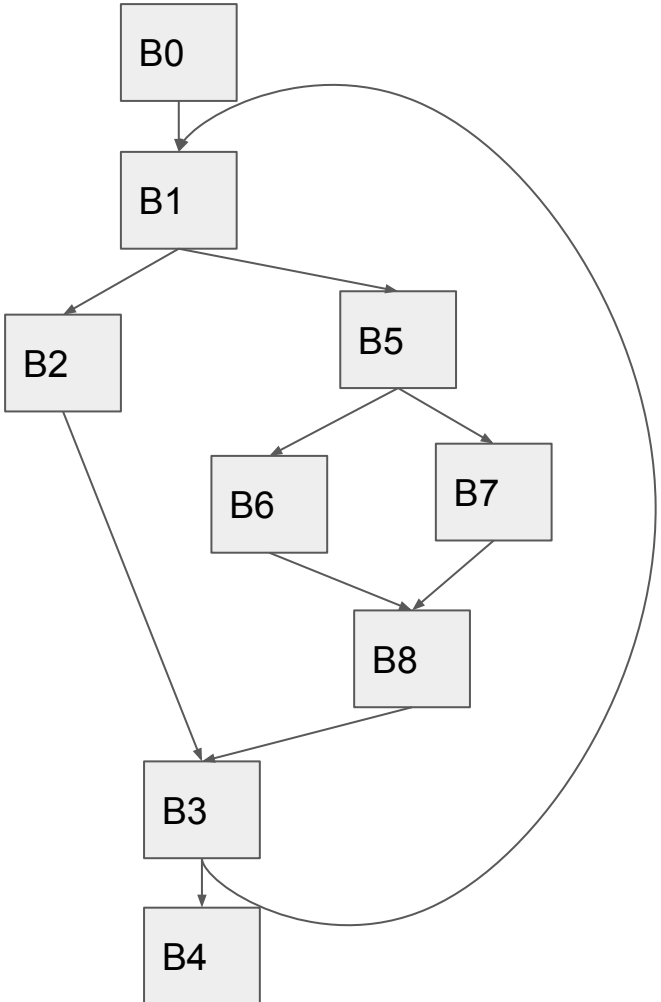
$DOM(b_0) \leftarrow \emptyset$

```
for i ← 1 to N
      DOM(b_i) ← { all nodes in graph }

change ← true
while (change)
      change ← false

      for i ← 0 to N
            TEMP ← { i } ∪ (∩_{x∈pred(b)} DOM(x))

            if DOM(b_i) ≠ TEMP then
                  change ← true
                  DOM(b_i) ← TEMP
```



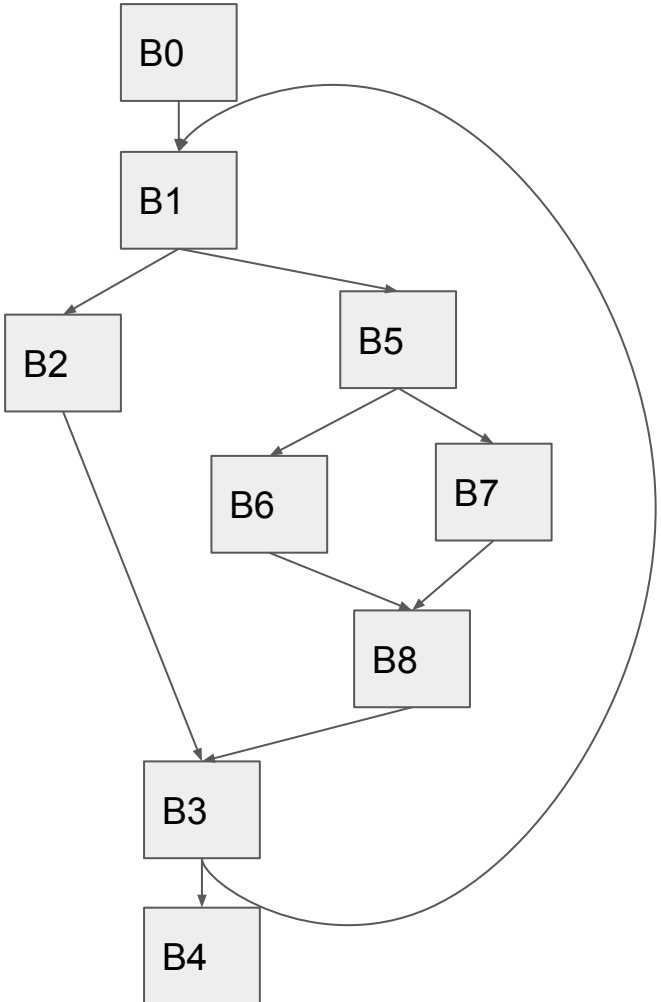
Initial

Basic Block	**Dominator Set**
B0	{ 0 }
B1	N
B2	N
B3	N
B4	N
B5	N
B6	N
B7	N
B8	N



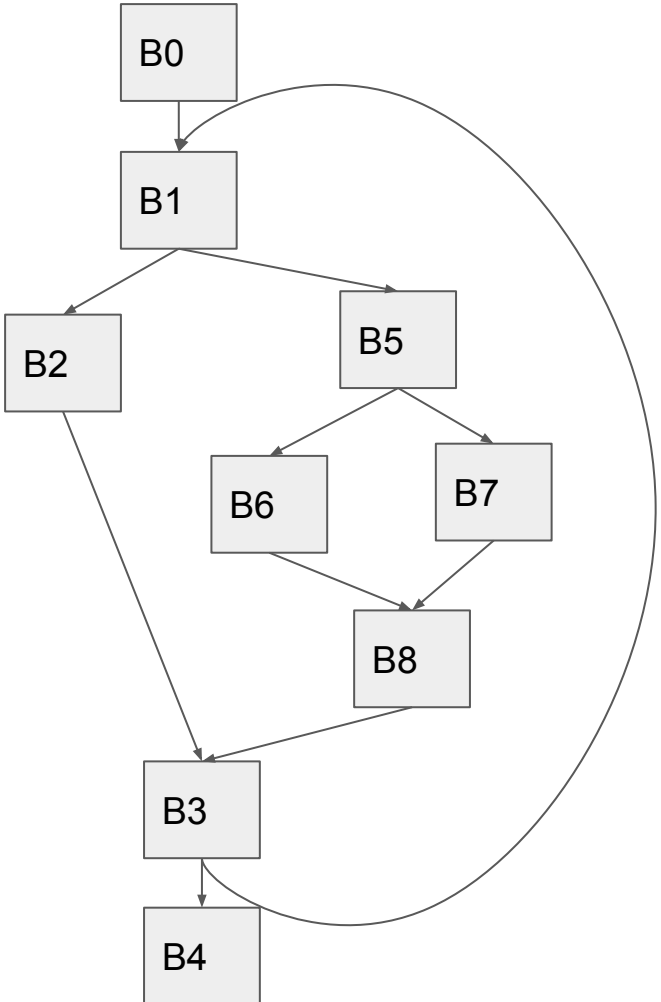
Iteration 1

Basic Block	**Dominator Set**
B0	{ 0 }
B1	{ 0, 1 }
B2	{ 0, 1, 2 }
B3	{ 0, 1, 2, 3 }
B4	{ 0, 1, 2, 3, 4 }
B5	{ 0, 1, 5 }
B6	{ 0, 1, 5, 6 }
B7	{ 0, 1, 5, 7 }
B8	{ 0, 1, 5, 8 }



Iteration 2

Basic Block	**Dominator Set**
B0	{ 0 }
B1	{ 0, 1 }
B2	{ 0, 1, 2 }
B3	{ 0, 1, 3 }
B4	{ 0, 1, 3, 4 }
B5	{ 0, 1, 5 }
B6	{ 0, 1, 5, 6 }
B7	{ 0, 1, 5, 7 }
B8	{ 0, 1, 5, 8 }



Iteration 3 (No change)

Basic Block	**Dominator Set**
B0	{ 0 }
B1	{ 0, 1 }
B2	{ 0, 1, 2 }
B3	{ 0, 1, 3 }
B4	{ 0, 1, 3, 4 }
B5	{ 0, 1, 5 }
B6	{ 0, 1, 5, 6 }
B7	{ 0, 1, 5, 7 }
B8	{ 0, 1, 5, 8 }

Aside on Data-Flow Analysis

Termination

- **The DOM sets are initialized to the (finite) set of nodes**
- **The DOM sets shrink monotonically**
- **The algorithm reaches a *fixed point* where they stop changing**

Correctness

- **We <u>can</u> prove that the fixed point solution is also the MOP**
- **Beyond the scope of the class, though we'll explain intuition later**

Efficiency

- **The round-robin algorithm is ***<u>not</u>*** particularly efficient**
- **Order in which we visit nodes is important for efficient solutions**

Class Activity

What would happen in the above example if we went backwards in this order (see below). How many iterations would it take to converge ? Why ?

Block	Initial	Iteration 1	Iteration 2	Iteration 3
B0	{ 0 }			
B1	N			
B5	N			
B8	N			
B6	N			
B7	N			
B2	N			
B3	N			
B4	N			

Class Activity

What would happen in the above example if we went backwards in this order (see below). How many iterations would it take to converge ? Why ?

Block	Initial	Iteration 1	Iteration 2	Iteration 3
B0	{ 0 }	{ 0 }	{ 0 }	{ 0 }
B1	N	{ 0, 1 }	{ 0, 1 }	{ 0, 1 }
B5	N	{ 0, 1, 5 }	{ 0, 1, 5 }	{ 0, 1, 5 }
B8	N	N	{ 0, 1, 5, 8 }	{ 0, 1, 5, 8 }
B6	N	{ 0, 1, 5, 6 }	{ 0, 1, 5, 6 }	{ 0, 1, 5, 6 }
B7	N	{ 0, 1, 5, 7 }	{ 0, 1, 5, 7 }	{ 0, 1, 5, 7 }
B2	N	{ 0, 1, 2 }	{ 0, 1, 2 }	{ 0, 1, 2 }
B3	N	{ 0, 1, 2, 3 }	{ 0, 1, 3 }	{ 0, 1, 3 }
B4	N	{ 0, 1, 2, 3, 4 }	{ 0, 1, 3, 4 }	{ 0, 1, 3, 4 }

Outline

Goals of dataflow analysis

Dominators and Dominator sets: DFA-1

Live-Out variable analysis: DFA-2

Other problems using DFA

Generalized framework for DFA

Live-Out Problem

For each node (basic block) in the CFG, the set of variables that are *live* on exit from the block, i.e., variable is used before being redefined.

Important for compilers to find

1. Variables that are uninitialized (later)
2. Variables for which registers need to be allocated
3. Remove useless or dead store instructions

Formal Defn.: A variable *v* is live at point *p* if and only if there exists a path in the CFG from *p* to a use of *v* along which *v* is not redefined (i.e., overwritten)

Two Sets: UEVar and VarKill

UEVar(B): Set of Upper-Exposed Variables in Basic Block B

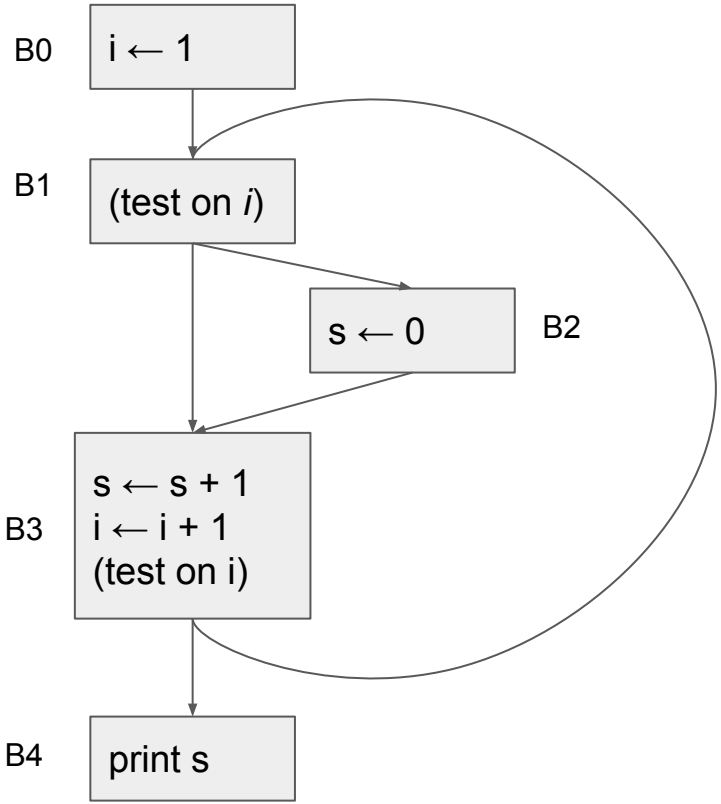
- Variables that are used in basic block B *before* they are (re)defined in B
- These are variables that need to have been defined before reaching B

VarKill (B): Set of variables that are defined (i.e., killed) in basic block B

- Anytime a variable is (re)defined, it’s previous value is killed

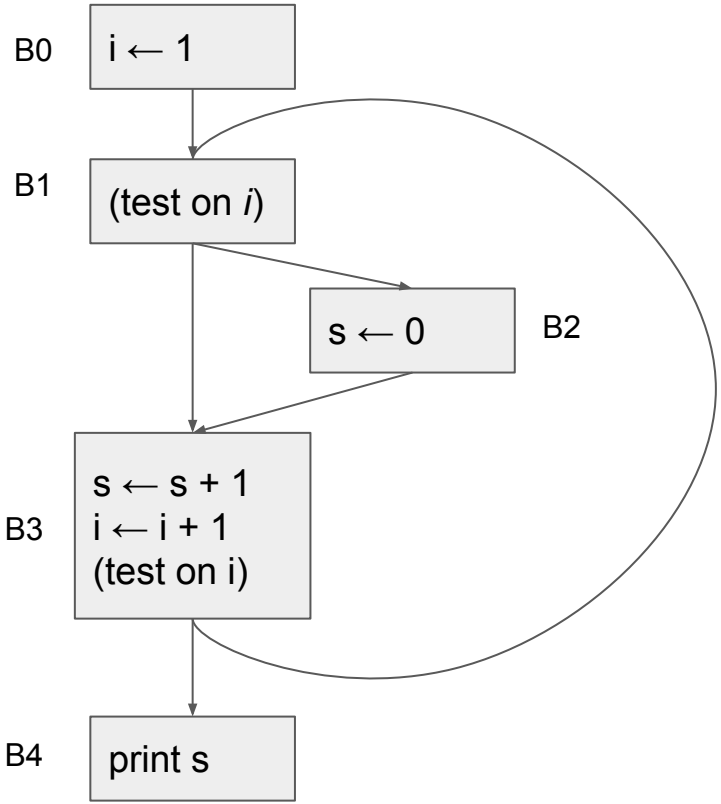
We’ll define LiveOut(B) in a recursive way using the above two sets

Can you find the UEVar and VarKill sets below ?



Block	**UEVar**	**VarKill**
B0		
B1		
B2		
B3		
B4		

Can you find the UEVar and VarKill sets below ?



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

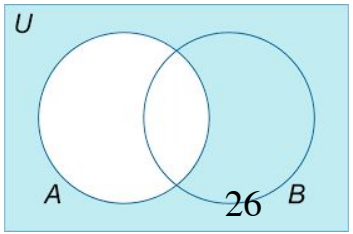
Recursive Formulation of LiveOut

How will you define LiveOut(B) in terms of UEVar and VarKill ?

- Defined in terms of succ(B). Why ?
- A variable is LiveOut in a block B if it's used in *any successor block of B* before being defined (UEVar (succ(B))
- A variable is LiveOut in a block B if it is LiveOut in the successor block
- *Unless* it is defined in the successor block *(*VarKill (succ(B))

NOTE: Subtracting a set is the same as intersecting with the complement (i.e., the set of elements that are not in the set), written as $\overline{S}$

- B - A → B intersection (A complement)



Dataflow Equations for LiveOut

We write it using two equations: LiveIn and LiveOut for convenience

- LiveIn is the set of variables that are live on *entry* to a basic block
- LiveOut is the set of variables that are live on *exit* from a basic block

LiveOut of a block is the Union of the LiveIn sets of its successor block

Initialization

$$LIVEOUT(n) = \emptyset, \ \forall n$$

Fixed-point equations

$$LIVEOUT(b) = \bigcup_{s \in succ(b)} LIVEIN(s)$$

$$LIVEIN(b) = UEVAR(b) \cup (LIVEOUT(b) \cap \overline{VARKILL(b)})$$

Comparison between Dominators and LiveOut

Criteria	**Dominators**	**LiveOut**
Type of Analysis	Forward data-flow (i.e., predecessors)	Backward data-flow (i.e., successors)
Meet Over Paths (MOP)	Intersection	Union
Needs other sets	No	Yes
Algorithm for solving	**Iterative**	**Iterative**

Algorithm for Liveout (from the EaC book)

```
// assume block b has k operations
// of form "x ← y op z"
for each block b
   Init(b)

Init(b)
   UEVar(b) ← ∅
   VarKill(b) ← ∅
   for i ← 1 to k
      if y ∉ VarKill(b)
         then add y to UEVar(b)
      if z ∉ VarKill(b)
         then add z to UEVar(b)
      add x to VarKill(b)
```

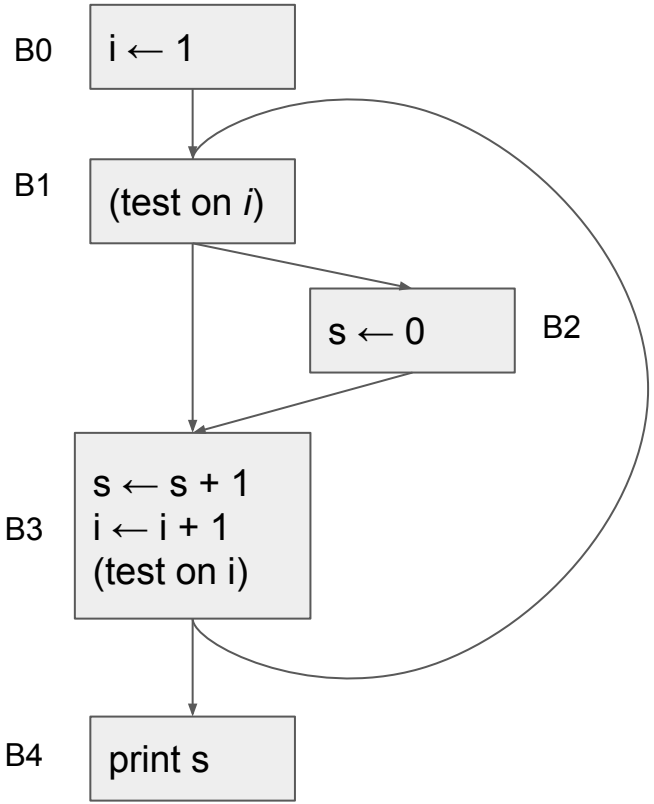
(a) Gathering Initial Information

```
// assume CFG has N blocks
// numbered 0 to N-1
for i ← 0 to N-1
   LiveOut(i) ← ∅

changed ← true
while (changed)
   changed ← false
   for i ← 0 to N-1
      recompute LiveOut(i)
      if LiveOut(i) changed then
         changed ← true
```

(b) Solving the Equations

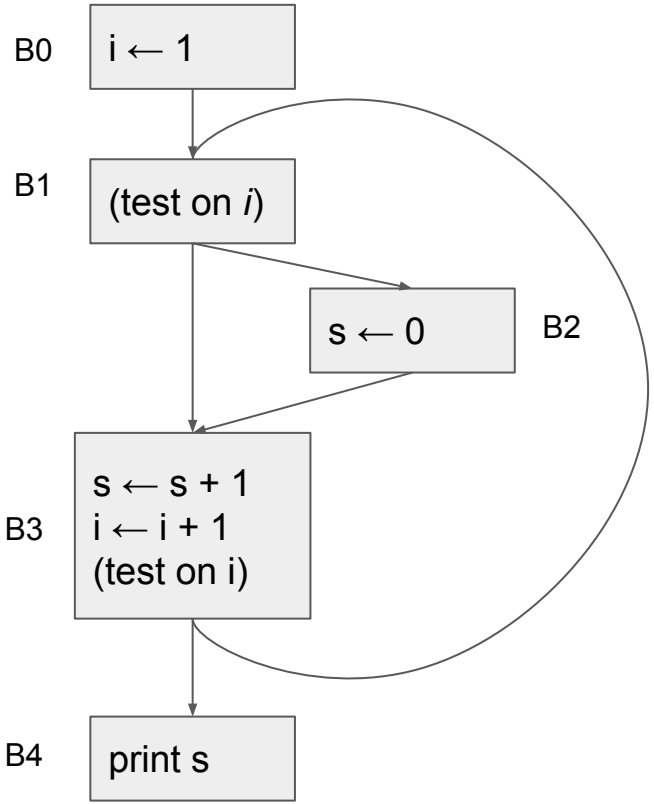
Eac Book: Figure 8.14

Example: Initial



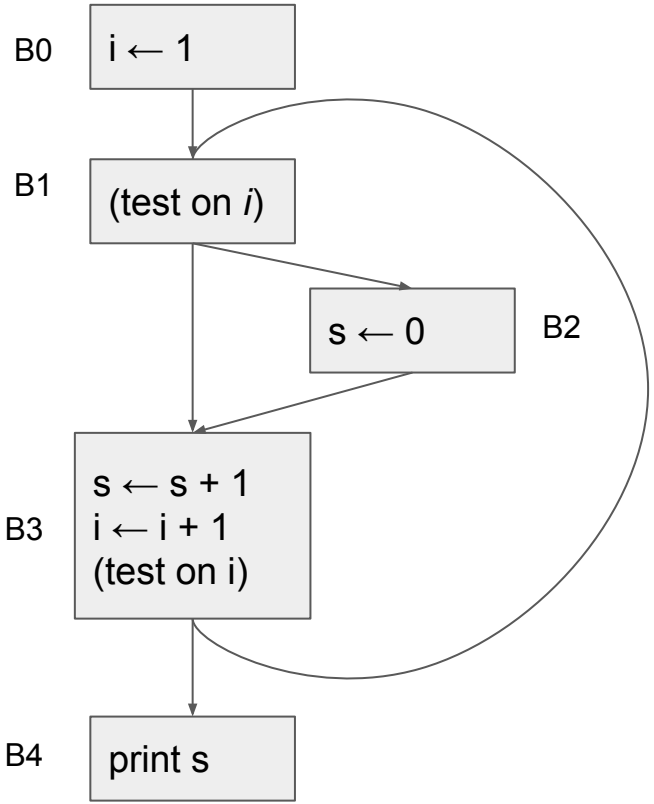
Block	UEVar	VarKill	LiveOut
B0	{ }	{ i }	{ }
B1	{ i }	{ }	{ }
B2	{ }	{ s }	{ }
B3	{ s, i }	{ s, i }	{ }
B4	{ s }	{ }	{ }

Example: Iteration-1



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

Example: Iteration-2 (Converged)



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

Use of LiveOut: Uninitialized Variables

How will you use LiveOut to compute the set of uninitialized variables in a procedure ?

- All variables that are in the LiveOut set of the initial block are undefined by definition
- However, this approach may yield false-positives (i.e., falsely identify variables as undefined). Can you think of why ?
- This is a common problem with static analysis (over-approximation)

Challenges in identifying Uninitialized Variables

Changes via pointers:

```
int* p = &x;

*p = 0;

…

x = x + 1;
```

Is x uninitialized above ?

- Use of pointer analysis can resolve this (also known as alias analysis)
- General problem is undecidable !

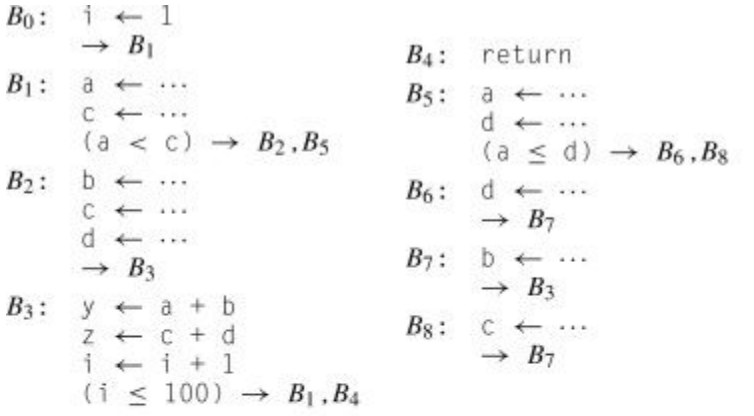
Infeasible Paths:

```
void foo(int n) {  // Assume n >= 0
        int i = 1;
        int s;              // s is uninitialized here
        while  (i <= n) {
                if (i==1) s = 0;
                s = s + i; // Is s uninitialized here?
        }
```

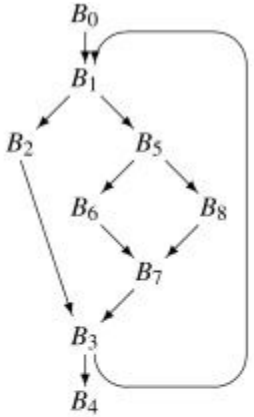
Difficult to resolve in general case

- Can be resolved in this case via loop specialization
- Need symbolic execution techniques (later)

Class Activity: Compute LiveOut for the following code



(a) Code for the Basic Blocks



(b) Control-Flow Graph

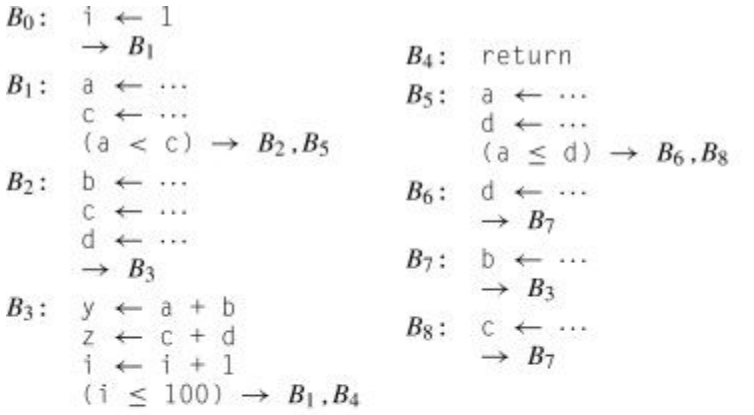
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
UEVAR	∅	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	∅
VARKILL	{i}	{a,c}	{b,c,d}	{y,z,i}	∅	{a,d}	{d}	{b}	{c}

(c) Initial Information

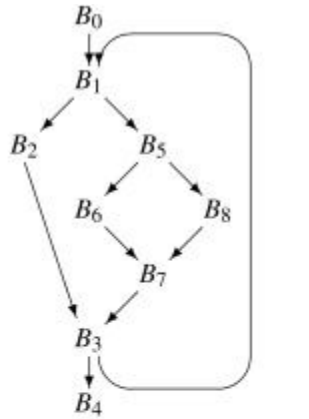
Eac Book: Figure 9.2

B0	{ }
B1	{ }
B2	{ }
B3	{ }
B4	{ }
B5	{ }
B6	{ }
B7	{ }
B8	{ }

Class Activity Solution: Iteration 1



(a) Code for the Basic Blocks



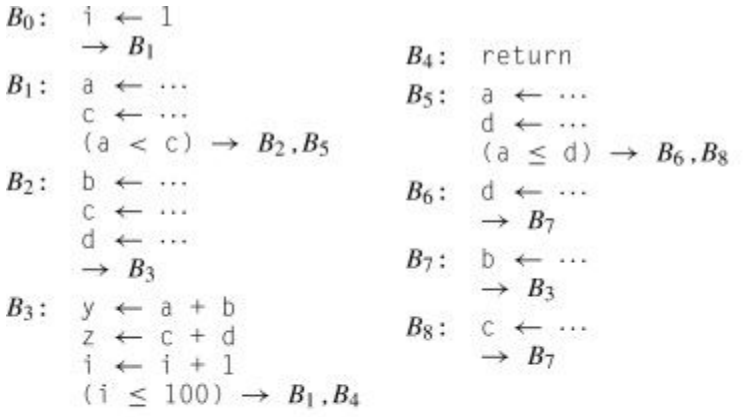
(b) Control-Flow Graph

	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}

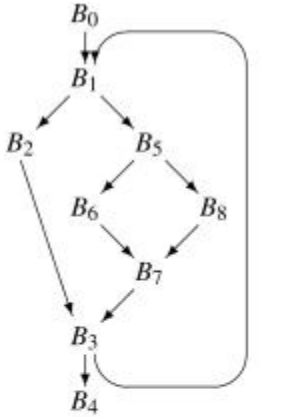
(c) Initial Information

B0	{ }
B1	{ }
B2	{ a, b, c, d, i }
B3	{ }
B4	{ }
B5	{ }
B6	{ }
B7	{ a, b, c, d, i }
B8	{ a, c, d, i }

Class Activity Solution: Iteration 2



(a) Code for the Basic Blocks



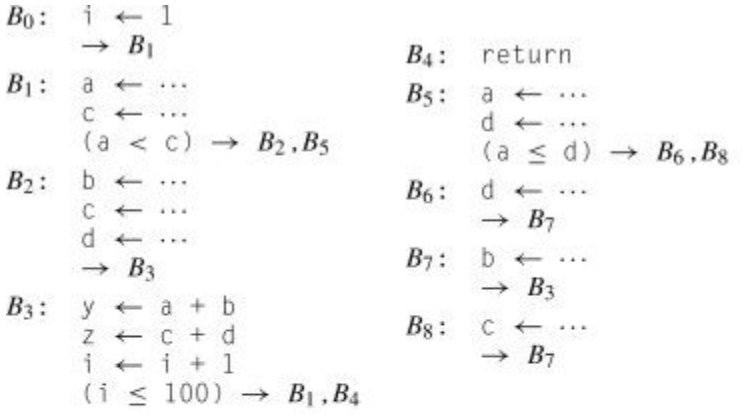
(b) Control-Flow Graph

	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}

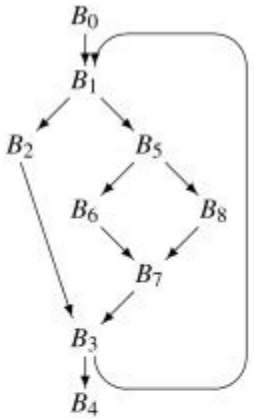
(c) Initial Information

B0	{ }
B1	{ a, i }
B2	{ a, b, c, d, i }
B3	{ i }
B4	{ }
B5	{ a, d, i }
B6	{ a, c, d, i }
B7	{ a, b, c, d, i }
B8	{ a, c, d, i }

Class Activity Solution: Iteration 3



(a) Code for the Basic Blocks



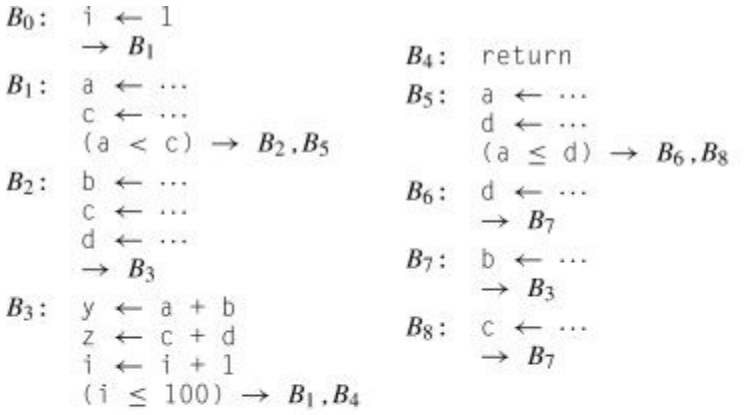
(b) Control-Flow Graph

	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}

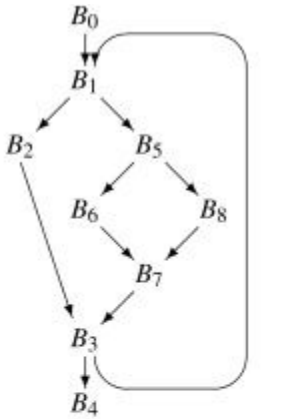
(c) Initial Information

B0	{ i }
B1	{ a, i }
B2	{ a, b, c, d, i }
B3	{ i }
B4	{ }
B5	{ a, c, d, i }
B6	{ a, c, d, i }
B7	{ a, b, c, d, i }
B8	{ a, c, d, i }

Class Activity Solution: Iteration 4 (Converged)



(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

B0	{ i }
B1	{ a, c, i }
B2	{ a, b, c, d, i }
B3	{ i }
B4	{ }
B5	{ a, c, d, i }
B6	{ a, c, d, i }
B7	{ a, b, c, d, i }
B8	{ a, c, d, i }

Outline

Goals of dataflow analysis

Dominators and Dominator sets: DFA-1

Live-Out variable analysis: DFA-2

Other problems using DFA

Generalized framework for DFA

Other Dataflow problems

Many other problems can be expressed as data-flow equations. For example,

1. Available Expressions

2. Reaching Definitions

3. Anticipable Expressions (also known as Very Busy Expressions)

Available Expressions

An expression is *available* on entry to *b* if, along every path from the entry node to *b*, it has been computed AND none of its constituent subexpressions has been subsequently modified.

$$AVAIL(b) = \cap_{x \in pred(b)} (DEEXPR(x) \cup (AVAIL(x) \cap \overline{EXPRKILL(x)}))$$

- ***EXPRKILL(b)* is the set of expression killed in *b*, and**
- ***DEEXPR(b)* is the set of expressions defined in *b* and not subsequently killed in *b* (Downward Exposed Expressions)**

Initial conditions: $AVAIL(n_0) = \emptyset$, $AVAIL(n)$ = { all expressions }

Available expressions is a *forward* data-flow problem (why?)

Using Available Expressions

AVAIL* is the basis for *global common subexpression elimination

- If $x+y \in AVAIL(b)$ then the compiler can replace any upwards exposed occurrence of $x+y$ in b with a reference to the previously computed value
- Standard caveats occur about preserving prior computations
  - Hash $x+y$ and assign temporary names; copy all $x+y$ to name
  - Walk backward through code to find minimal set of copies
- Resulting algorithm is similar to value numbering
  - Based on textual identity of expressions, not value identity
  - Recognizes redundancy carried along loop-closing branch

Reaching Definitions

Find the set of definitions (i.e., variables' values) that reach a program point 'p'

A definition *d reaches* operation *i* if and only if *i* reads the value *v*, and there exists a path from *d* to *v* that does not define *i*

Initially, for all n, **Reaches(n) = Ø**

Reaches(n) = $\bigcup_{m \in preds(n)}$ **(DEDEF(m) U (Reaches(m) ∩ $\overline{\text{DEFKILL(m)}}$)**

DEDEF(m): Downward exposed definitions (i.e., defined in m and not redefined)

DEFKILL(m): If it defines a name *v,* and m contains definition that also defines *v*

Anticipable Expressions

An expression *e* is anticipable or very busy if on exit from block *b*, if and only if (1) *every* path that leaves *b* evaluates and subsequently uses *e*, AND (2) evaluating *e* at the end of *b* would produce the same result as the first evaluation along paths.

Initial conditions: $ANTOUT(n_f) = \emptyset$, $ANTOUT(n)$ = { all expressions }

$$ANTOUT(n) = \bigcap_{m \in succ(n)} (UEEXPR(m) \cup (ANTOUT(m) \cap \overline{EXPRKILL(m)}))$$

UEEXPR(m): Upper-Exposed Expressions - Used in 'm' before being killed

EXPRKILL(m): Set of expressions that are defined in m (same as in AVAIL)

Outline

Goals of dataflow analysis

Dominators and Dominator sets: DFA-1

Live-Out variable analysis: DFA-2

Other problems using DFA

Generalized framework for DFA

Why identify common features ?

Many kinds of data-flow problems - each comes with unique characteristics

But many characteristics are common (MOP operator, backward Vs forward etc.)

Can be combined into a common framework for code reuse etc. - developer only needs to specify the unique parts of the problem in a standardized way

Heuristics to efficiently solve generalized data-flow problems matching certain templates (e.g., traverse basic blocks in reverse post-order for backward dataflow)

Cleaner way to understand and prove correctness of algorithms (not covered)

Common features of data-flow analysis

1. Type of data-flow analysis
    a. Backward (successors)
    b. Forward (predecessors)

2. Meet over Paths (MoP) Operator
    a. Union (“May”)
    b. Intersection (“Must”)

Dataflow Analysis can be classified in this common way

Classification (Direction/MOP)	**May**	**Must**
Forward	Reaching Definitions	Available Expressions
Backward	Live Variables	Anticipable Expressions

What about Dominators ? Where does it fit in the above table ?

Outline

Goals of dataflow analysis

Dominators and Dominator sets: DFA-1

Live-Out variable analysis: DFA-2

Other problems using DFA

Generalized framework for DFA