Data Flow Analysis

- Structure of Data Flow Analysis
- Reaching Definitions Analysis
- Liveness Analysis
- Homework: summarize Chap. 8.1

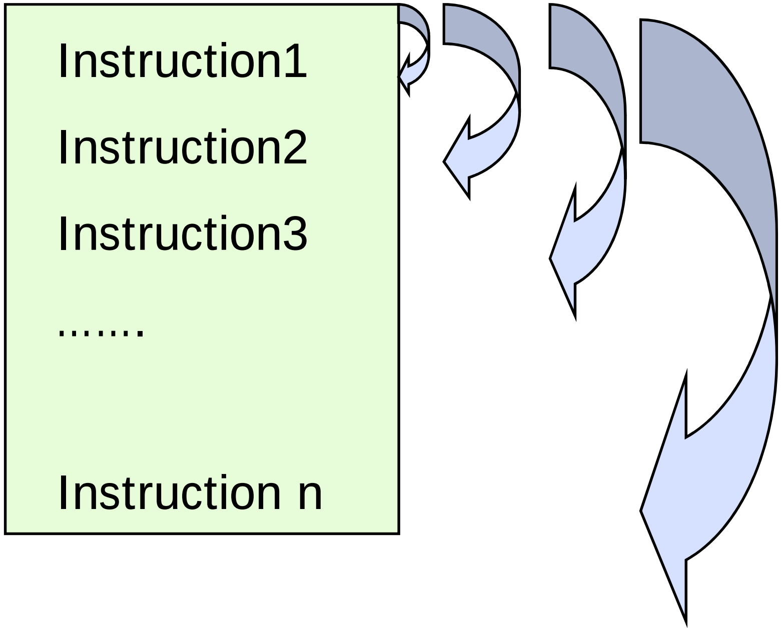
Data Flow Analysis

- Provide global information on how a given procedure (large code) manipulates its data
  - Different optimizations may require different info.
    - e.g., building live ranges, constant folding
  - Must not give incorrect information that causes incorrect optimizations
    - Should be conservative approximation if not precise
- We'll compute reaching definitions and live variables, which are most heavily-used info.

Local Data Flow Analysis

Performed within a basic block

- Analyze effect of each instruction
- Compose effects of instructions so that we can derive information from the beginning (or from the end) of a basic block to each instruction



Global Data Flow Analysis

Performed beyond basic blocks

- Analyze effect of each basic block
- Compose effects of basic blocks to derive information at basic block boundaries from the beginning (or from the end) of a procedure

From basic block boundaries, apply local analysis technique to get info. at each instructions

You can also do global analysis in the level of instructions, yet it would be more expensive

Effects of an Instruction

- For an instruction a = b + c
  - **Uses** variables b and c
  - **Kills** an old definition of a
  - **Generate** a new definition a

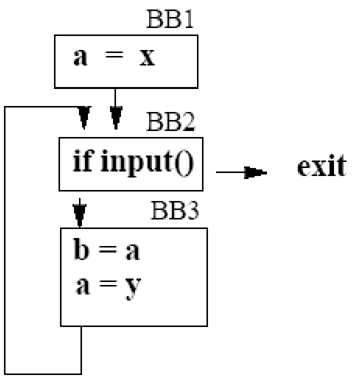
Effects of a Basic Block (BB)

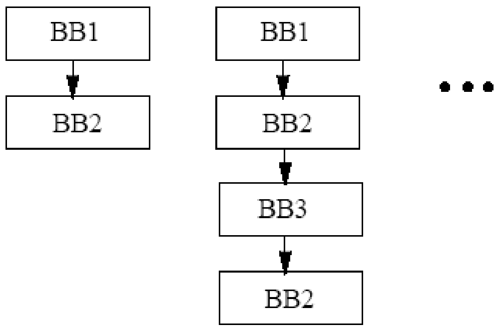
- **Compose effects of instructions**
  - A **locally exposed use** in BB is a use of a data item which is not preceded in the BB by a definition of the data item
  - Any definition of a data item in the BB **kills** all the definitions of the same data item reaching the BB
  - A **locally generated definition**: last definition of data item in BB

```
t1 = r1 + r1
r2 = t1
t2 = r2 + r1
r1 = t2
t3 = r1 * r1
r2 = t3
if r2 > 100 goto L1
```

Composing Effects Across Basic Blocks

- Distinguish between Static Program vs. Dynamic Execution

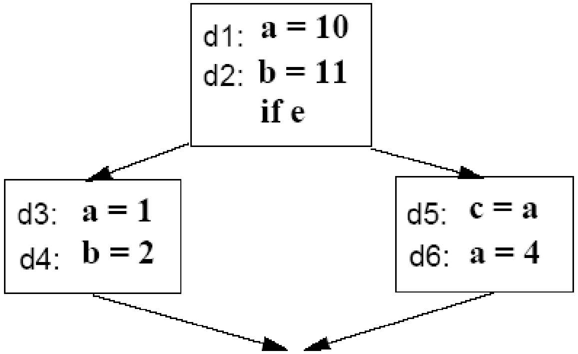




  - Statically: finite program
  - Dynamically: potentially infinite possible execution paths
    - We can, in theory, reason about each possible path as if all instructions executed are in one BB
- Data Flow Analysis
  - Associate with each **static** point in the program information true of the set of **dynamic** instances of that program point

Reaching Definitions

- A **definition** of a variable x is an instruction that assigns, or may assign (e.g., conditional execution), a value to x
  - A **definition** *d* reaches a point *p* if **there exists** a path from the point immediately following *d* to *p* such that *d* is not killed along that path



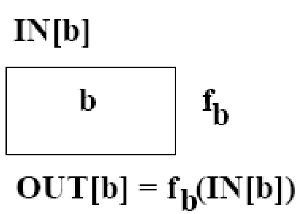
Analysis of Reaching Definitions

- Problem Statement
  - For each basic block *b*, determine if each definition in the procedure reaches *b*
    - We want such info. both at the beginning of *b* (called IN[b]) and at the end of *b* (called OUT[b])
- A Representation
  - IN[b], OUT[b]: a bit vector, one bit for each definition in the procedure
- A basic block b can affect the computation:
  - Can we compute OUT[b] from IN[b]? Or can we compute IN[b] from OUT[b]? Which one is correct?

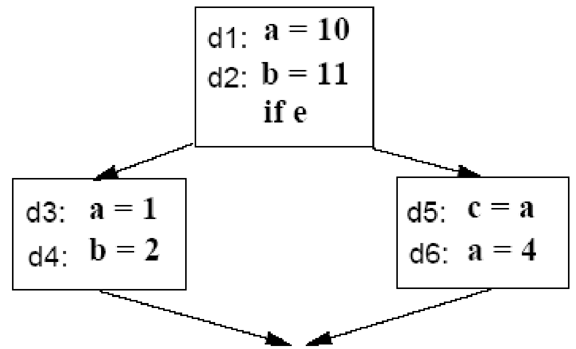
Effect of a Basic Block

- If IN[b] is given, we can compute OUT[b] (forward problem)

Schema



Example:



- A transfer function f_b for a basic block b:

$$OUT[b] = f_b(IN[b])$$

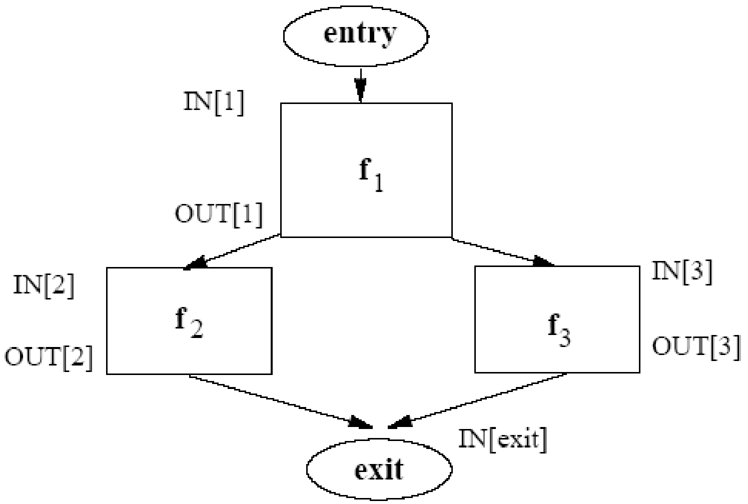
(incoming reaching definitions → outgoing reaching definitions)

Describing the Effect of a Basic Block

- A basic block *b*
  - **generates** definitions: GEN[*b*], set of locally generated definitions in *b*
  - **propagate** definitions: IN[*b*] – KILL[*b*], where KILL[*b*] is set of definitions in rest of program killed by definitions in *b*

$$\text{OUT}[b] = \text{GEN}[b] \cup (\text{IN}[b] - \text{KILL}[b])$$

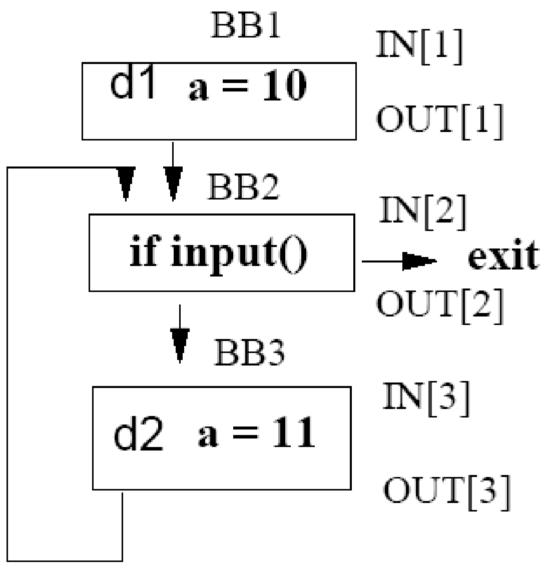
Effect of Edges (in Acyclic Graphs)



	GEN	KILL
1	{1,2}	{3,4,6}
2	{3,4}	{1,2,6}
3	{5,6}	{1,3}

- We know OUT[b] can be computed from IN[b]. Now can we compute IN from predecessor(s)' OUT? Just propagate if it is a single predecessor
- Join node (a node with multiple predecessors) complicates computation
- We need a **meet** operator at join nodes. What should it be?
- For reaching definitions, it is a union operator
  - $IN[b] = OUT[p_1] \cup OUT[p_2] \cup \ldots OUT[p_n]$, where p_1, p_2, ...p_n are all predecessors of b

Effect of Edges (in Cyclic Graphs)



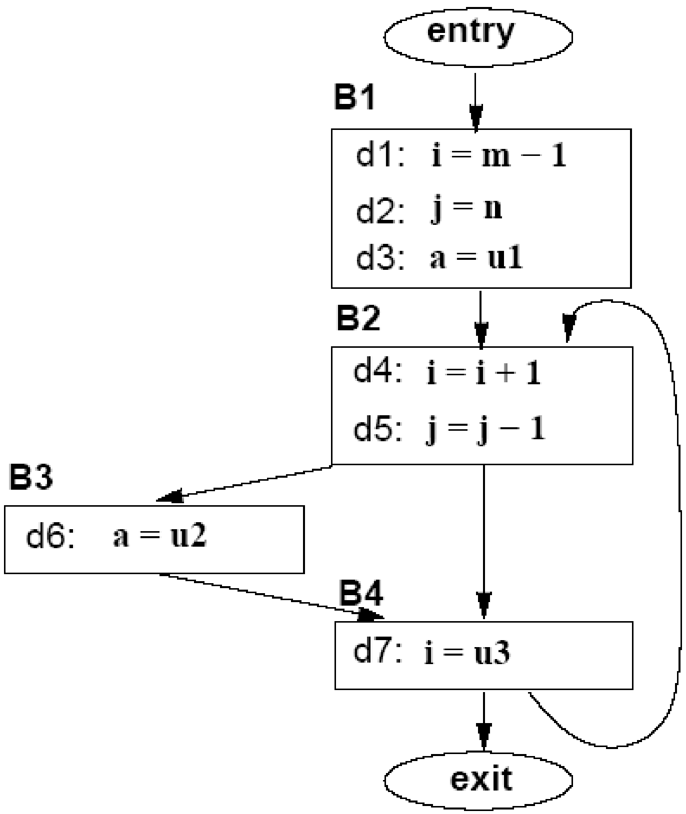
	GEN	KILL
1	d1	d2
2		
3	d2	d1

- Previous equations still hold
  - $OUT[b] = f_b(IN[b])$
  - $IN[b] = OUT[p_1] \cup OUT[p_2] ... \cup OUT[p_n]$
- Any solution that meets the equation: fixed point solution
- For a more precise solution, we need repeated computation using the equations, e.g., using a worklist algorithm

Reaching Definitions: Worklist Algorithm

```
Input: Control Flow Graph CFG = (N, E, Entry, Exit)
/* Initialize           */
    OUT[Entry] = { }
    for all nodes I
        OUT[i] = { }
    ChangeNodes = N
/* Iterate              */
    while ChangeNodes != { } {
        remove i from ChangeNodes
        IN[i] = U(OUT[p]), for all predecessors p of i
        oldout = OUT[i]
        OUT[i] = f_i(IN[i]) /* OUT[i] = GEN[i] U (IN[i] – KILL[i]) */
        if (oldout != OUT[i]) {
            for all successors s of i
            add s to ChangeNodes
        }
    }
```

Example



	GEN	KILL
B1	{1, 2, 3}	{4, 5, 6, 7}
B2	{4, 5}	{1, 2, 7}
B3	{6}	{3}
B4	{7}	{1, 4}

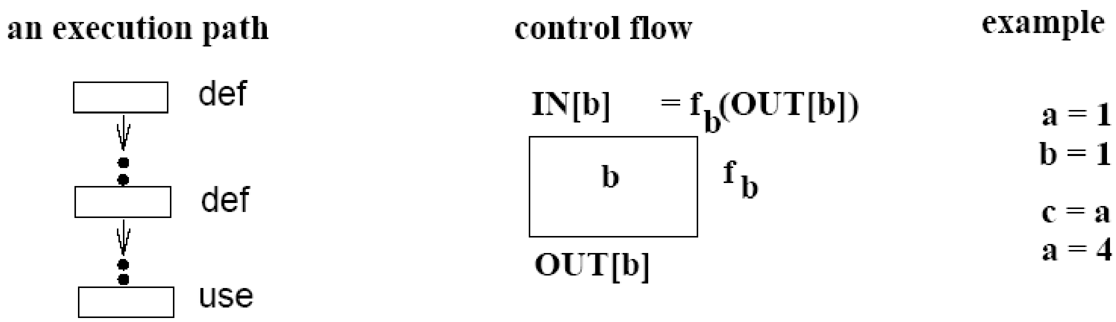
	IN	OUT
entry	{ }	{ }
B1	{ }	{1,2,3}
B2	{1,2,3}=>{1,2,3,5,6}	{3,4,5}=>{3,4,5,6}
B3	{3,4,5}=>{3,4,5,6}	{4,5,6}=>{4,5,6}
B4	{3,4,5,6}	{3,5,6,7}

Analysis of Live Variables

- Definition
  - A variable *v* is live at point *p* if the value of *v* is used along some path in the flow graph starting at *p*
  - Otherwise, the variable is dead
- Problem
  - For each BB, determine if each variable is live in each BB boundary (also called IN[b], OUT[b])
  - Size of bit vector: one bit for each variable

Effects of a Basic Block

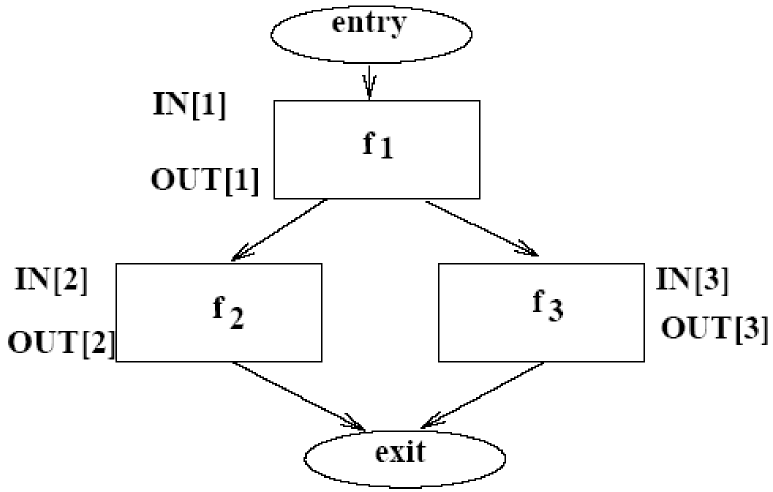
- If OUT[b] is given, we can compute IN[b] (backward problem)



- A basic block *b* can
  - Generate live variables:
    USE[*b*], set of locally exposed uses (variables) in *b*
  - Propagate incoming live variables:
    OUT[*b*] – DEF[*b*], where DEF[*b*] is set of variables defined in b

- **Transfer functions** for a block *b*
  - IN[*b*] = USE[*b*] ∪ (OUT[*b*] – DEF[*b*])

Effect of Edges (in Acyclic Graphs)



	USE	DEF
1	{e}	{a,b}
2	{}	{a,b}
3	{a}	{a,c}

- $IN[b] = f_b(OUT[b])$
- Join Node: a node with multiple successors
- Meet operator:

  $OUT[b] = IN[s_1] \cup IN[s_2] \cup \ldots \cup IN[s_n]$

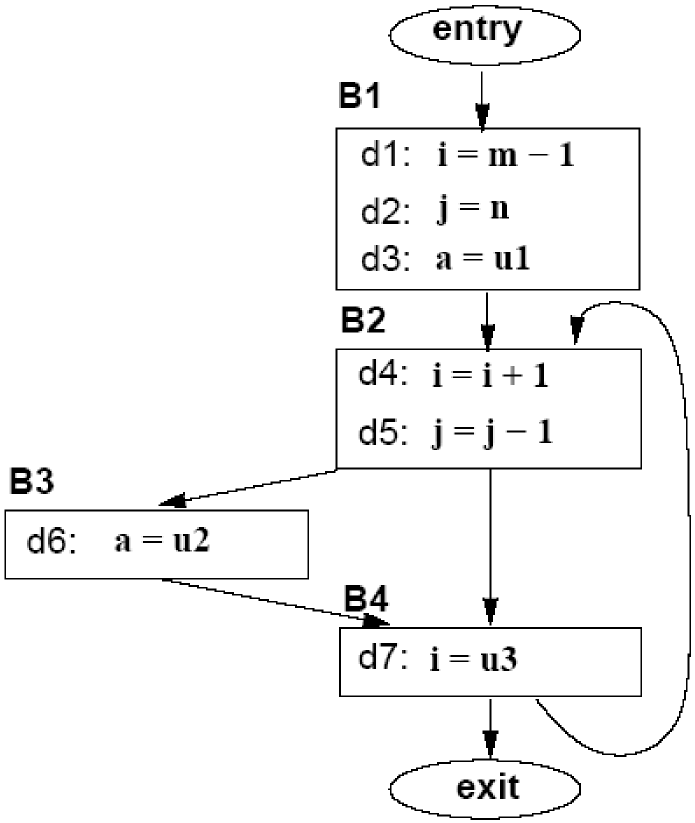
  $s_1, s_2, \ldots s_n$ are all successors of b

Effects of edges in cyclic graphs are similar as in reaching definitions

Live Variable: Worklist Algorithm

```
Input: Control Flow Graph CFG = (N, E, Entry, Exit)
/* Initialize           */
    IN[Exit] = { }
    for all nodes I
        IN[i] = { }
    ChangeNodes = N
/* Iterate              */
    while ChangeNodes != { } {
        remove i from ChangeNodes
        OUT[i] = U(IN[s]), for all successors s of i
        oldin = IN[i]
        IN[i] = f_i(OUT[i]) /* IN[i] = USE[i] U (OUT[i] – DEF[i]) */
        if (oldin != IN[i]) {
            for all predecessors p of i
            add p to ChangeNodes
        }
    }
```

Example



	GEN	DEF
B1		
B2		
B3		
B4		

	OUT	IN
entry		
B1		
B2		
B3		
B4		

Framework: Summary

	Reaching Definitions	Live Variables
Domain	Sets of Definitions	Sets of Variables
Transfer Function $f_b(x)$ Generate ∪ Propagate	GEN[b] ∪ (x – KILL[b])	USE[b] ∪ (x – DEF[b])
Direction of Function	Forward: out[b] = fb(in[b])	Backward: in[b] = fb(out[b])
Generate	GEN[b] (definitions in b)	USE[b] (vars used in b)
Propagate	IN[b] – KILL[b]	OUT[b] – DEF[b]
Merge Operation	IN[b] = U(OUT[pred])	OUT[b] = U(IN[succ])
Initialization	OUT[b] = {}	in[b] = {}
Boundary Condition	OUT[entry] = {}	IN[exit] = {}