Lecture 4

Introduction to Data Flow Analysis

I Structure of data flow analysis

II Example 1: Reaching definition analysis

III Example 2: Liveness analysis

IV Generalization

Reference: Chapter 8, 8.1-4

Data Flow Analysis

- Local analysis (e.g. value numbering)
  - analyze effect of each instruction
  - compose effects of instructions to derive information from beginning of basic block to each instruction
- Data flow analysis
  - analyze effect of each basic block
  - compose effects of basic blocks to derive information at basic block boundaries
  - (from basic block boundaries, apply local technique to generate information on instructions)

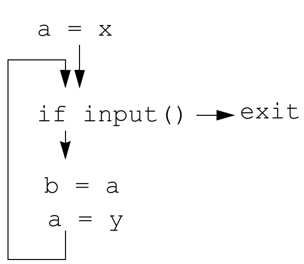
Effects of a basic block

- Effect of a statement: a = b+c
  - **Use**s variables (b, c)
  - **Kill**s an old definition (old definition of a)
  - new **definition** (a)
- Compose effects of statements -> Effect of a basic block
  - A **locally exposed use** in a b.b. is a use of a data item which is not preceded in the b.b. by a definition of the data item
  - any definition of a data item in the basic block **kills** all definitions of the same data item reaching the basic block.
  - A **locally available definition** = last definition of data item in b.b.

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


Across Basic Blocks

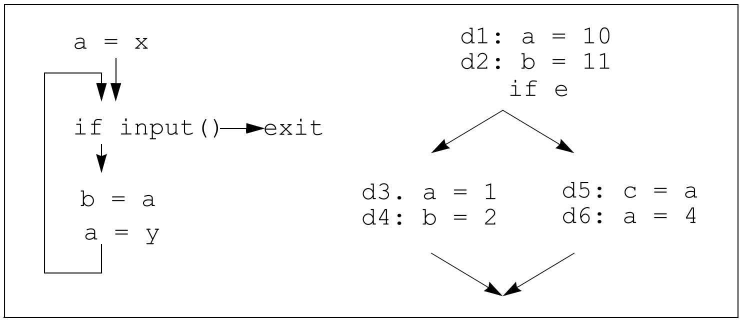
- **Static program vs. dynamic execution**



- **Statically:** Finite program
  Dynamically: Potentially infinite possible execution paths
- Can reason about each possible path
  as if all instructions executed are in one basic block
- Data flow analysis:
  Associate with each **static** point in the program
  information true of
  the set of **dynamic** instances of that program point

II. Reaching Definitions

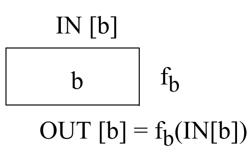
- A **definition** of a variable *x* is a statement that assigns, or may assign, a value to *x*.
- A **definition** *d* **reaches** a point *p* if **there exists a** path from the point immediately following *d* to *p* such that *d* is not killed along that path.



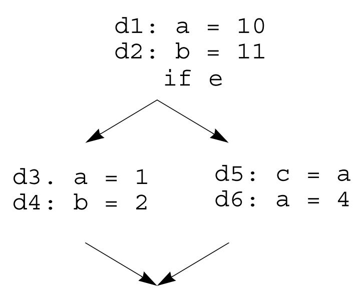
- Problem statement
  - For each basic block b, determine if each definition in the program reaches b
- A representation:
  - IN[b], OUT[B]: a bit vector, one bit for each definition

Describing Effects of the Nodes (basic blocks)

Schema

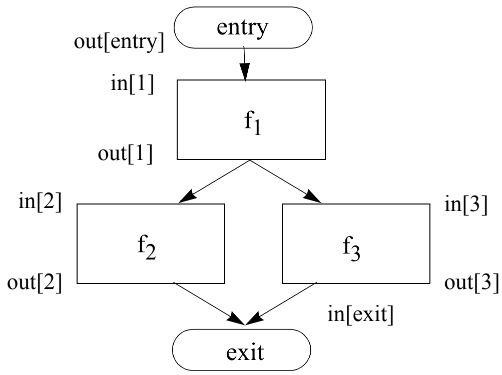


Example



- **a transfer function** f_b of a basic block b:
  $OUT[b] = f_b(IN[b])$
  incoming reaching definitions -> outgoing reaching definitions
- A basic block b
  - **generate** definitions: Gen[b], set of locally available definitions in b
  - **propagate** definitions: in[b] - Kill[b], where Kill[b]=set of defs (in rest of program) killed by defs in b
- **out[b] = Gen[b] U (in(b)-Kill[b])**

Effects of the Edges (acyclic)



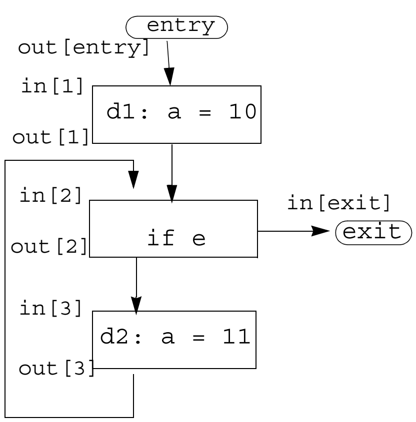
f	Gen	Kill
1	{1,2}	{3,4,6}
2	{3,4}	{1,2,6}
3	{5,6}	{1,3}

- out[b] = f_b(in[b])
- Join node: a node with multiple predecessors
- **meet** operator:

  in[b] = out[p_1] U out[p_2] U ... U out[p_n], where

  p_1, ..., p_n are all predecessors of b

Cyclic Graphs



- Equations still hold
  - out[b] = f_b(in[b])
  - in[b] = out[p_1] U out[p_2] U ... U out[p_n], p_1, ..., p_n pred.
- Solve for fixed point solution

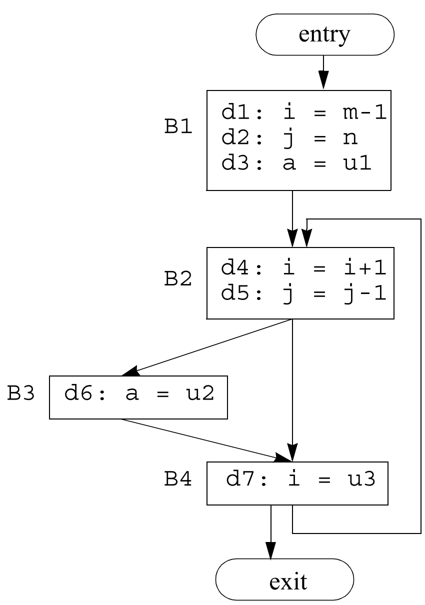
Reaching Definitions: Worklist Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
     out[Entry] = ∅      // can set out[Entry] to special def
                         // if reaching then undefined use
     For all nodes i
         out[i] = ∅          // can optimize by out[i]=gen[i]
     ChangedNodes = N

// iterate
     While ChangedNodes ≠ ∅ {
        Remove i from Changed Nodes
        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 ChangedNodes
        }
     }
```


Example



III. Live Variable Analysis

- **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**.
- **Motivation**
  - e.g. register allocation

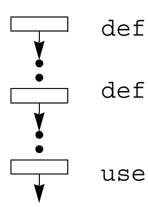
```
for i = 0 TO n
   .. i ..
...
for i = 0 to n
   .. i ..
```

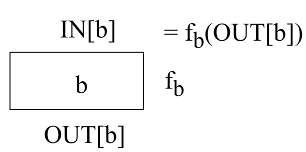
- **Problem statement**
  - For each basic block
    - determine if each variable is live in each basic block
  - Size of bit vector: one bit for each variable

Effects of a Basic Block (Transfer Function)

- **Observation:Trace uses backwards to the definitions**

an execution path | control flow | example



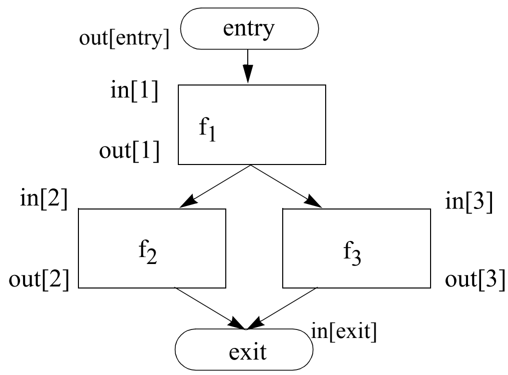



```
d3: a = 1
d4: b = 1

d5: c = a
d6: a = 4
```

- **A basic block b can**
  - generate live variables:
    Use[b], set of locally exposed uses in b
  - propagate incoming live variables: OUT[b] - Def[b],
    where Def[b]= set of variables defined in b.b.
- **transfer function** for block b:
  in[b] = Use[b] U (out(b)-Def[b])

Flow Graph



f	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] U in[s_2] U ... U in[s_n], where

  s_1, ..., s_n are all successors of b

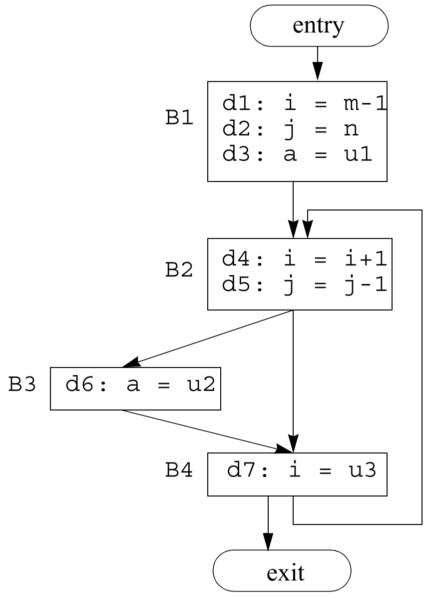
Live Variable: Worklist Algorithm

```
input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
     in[Exit] = ∅                  //local variables
     For all nodes i
         in[i] = ∅                 //can optimize by in[i]=use[i]
     ChangedNodes = N

// iterate
     While ChangedNodes ≠ ∅ {
        Remove i from Changed Nodes
        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 ChangedNodes
        }
     }
```

Example



IV. Framework

	Reaching Definitions	Live Variables
Domain	Sets of definitions	Sets of variables
Transfer function $f_b(x)$ Generate U Propagate		
direction of function	forward: out[b] = f_b(in[b])	backward: in[b] = f_b(out[b])
Generate	Gen_b (Gen_b: definitions in b)	Use_b (Use_b: var. used in b)
Propagate	in[b]-$Kill_b$ ($Kill_b$: killed defs)	out[b]-Def_b (Def_b:var defined)
Merge operation	U (in[b]=U out[predecessors])	U (out[b]= U in[successors])
Initialization	out[entry] = $\emptyset$	in[exit] = $\emptyset$
	out[b] = $\emptyset$	in[b] = $\emptyset$

Questions

- **Correctness**
  - equations are satisfied, if the program terminates.
- **Precision: how good is the answer?**
  - is the answer ONLY a union of all possible executions?
- **Convergence: will the analysis terminate?**
  - or, will there always be some nodes that change?
- **Speed: how fast is the convergence?**
  - how many times will we visit each node?