CSC D70: Compiler Optimization Dataflow Analysis

Prof. Gennady Pekhimenko

University of Toronto

Winter 2021

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons

Refreshing from Last Lecture

- Basic Block Formation

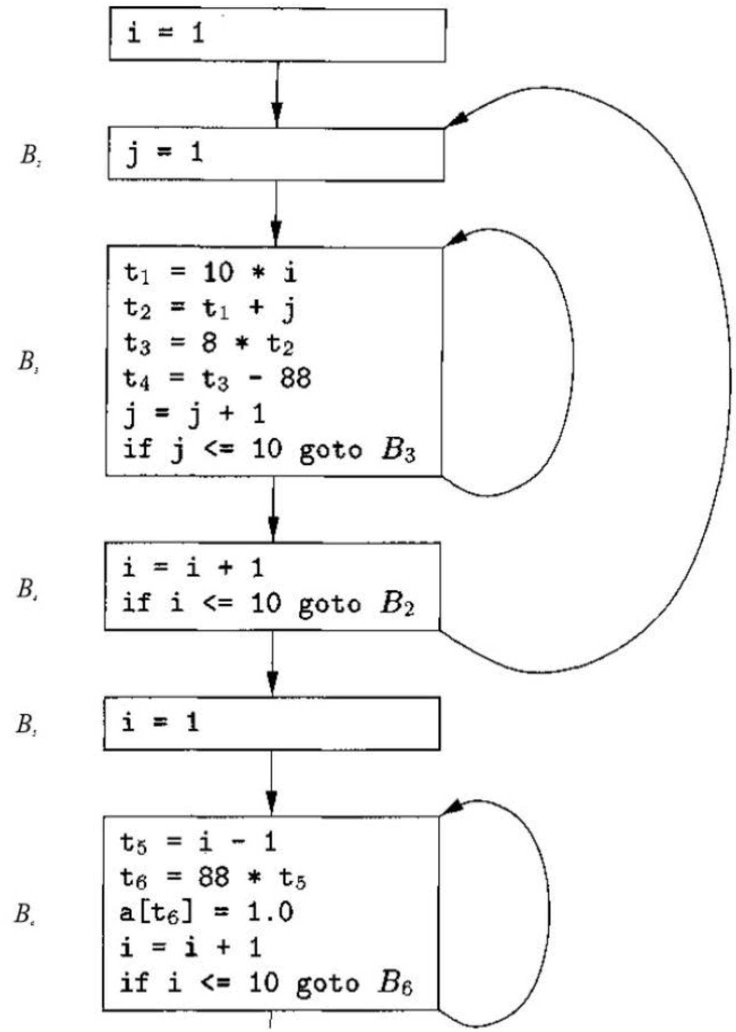
- Value Numbering

Partitioning into Basic Blocks

- Identify the leader of each basic block
  - First instruction
  - Any target of a jump
  - Any instruction immediately following a jump
- Basic block starts at leader & ends at instruction immediately before a leader (or the last instruction)

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

★ = Leader



ALSU pp. 529-531

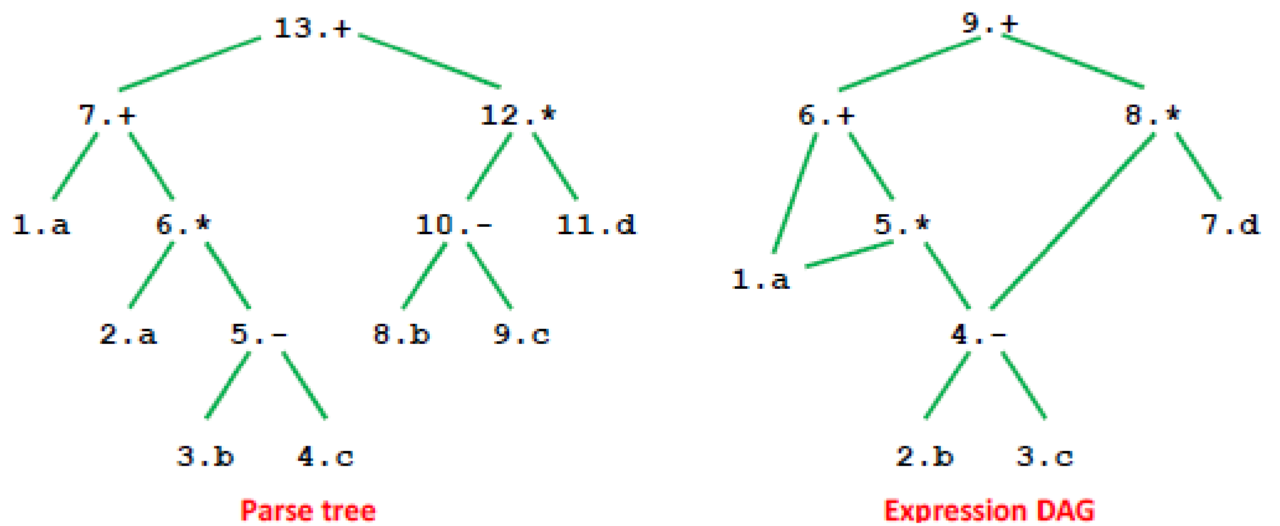
Graph Abstractions

Example 1:

- grammar (for bottom-up parsing):

E -> E + T | E – T | T, T -> T*F | F, F -> (E) | id

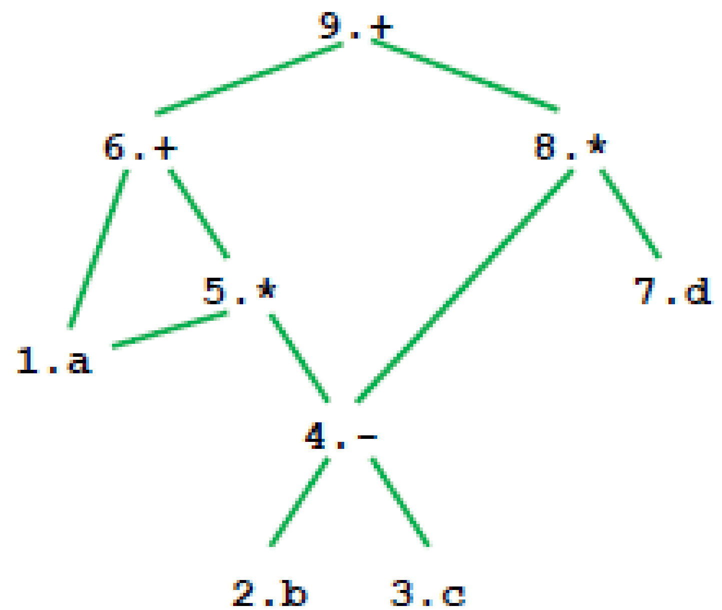
- expression: a+a*(b-c)+(b-c)*d



Graph Abstractions

Example 1: an expression

`a+a*(b-c)+(b-c)*d`



Optimized code:

t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

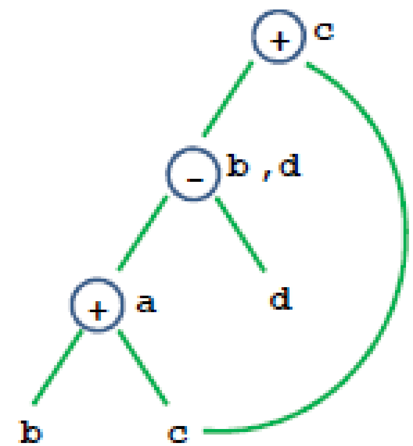
t5 = t3 + t4

How well do DAGs hold up across statements?

- **Example 2**

```
a = b+c;
b = a-d;
c = b+c;
d = a-d;
```

DAG – directed acyclic graph




```
Is this optimized code correct?
a = b+c;
d = a-d;
c = d+c;
```

Critique of DAGs

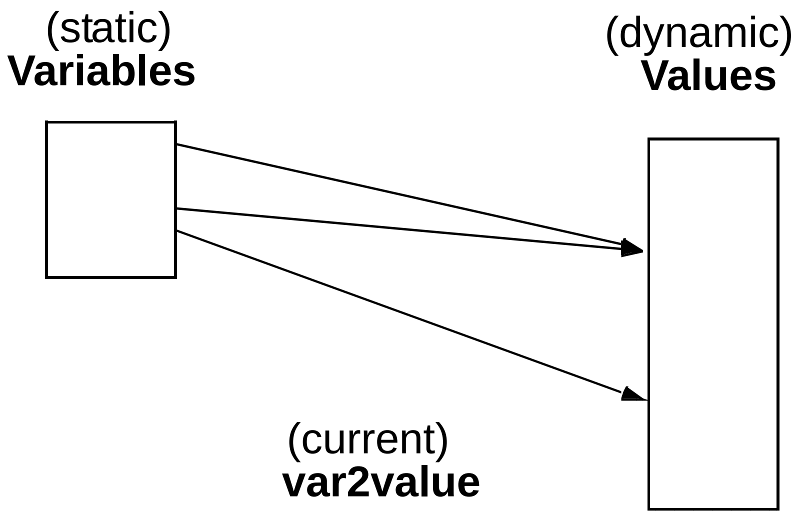
- **Cause of problems**
  - Assignment statements
  - Value of variable depends on TIME

- **How to fix problem?**
  - build graph in order of execution
  - attach variable name to latest value

- **Final graph created is not very interesting**
  - Key: variable->value mapping across time
  - loses appeal of abstraction

Value Numbering (VN)

- More explicit with respect to VALUES, and TIME



- **each value has its own “number”**
  - **common subexpression means same value number**
- var2value: current map of variable to value
  - used to determine the value number of current expression

r1 + r2 => var2value(r1)+var2value(r2)

Algorithm

```
Data structure:
     VALUES = Table of
        expression     //[OP, valnum1, valnum2}
        var            //name of variable currently holding expression

For each instruction (dst = src1 OP src2) in execution order

  valnum1 = var2value(src1); valnum2 = var2value(src2);

  IF [OP, valnum1, valnum2] is in VALUES
     v = the index of expression
     Replace instruction with CPY dst = VALUES[v].var
  ELSE
     Add
        expression = [OP, valnum1, valnum2]
        var        = dst
     to VALUES
     v = index of new entry; tv is new temporary for v
     Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                               CPY dst = tv;

  set_var2value (dst, v)
```

More Details

- **What are the initial values of the variables?**
  – values at beginning of the basic block
- **Possible implementations:**
  – Initialization: create “initial values” for all variables
  – Or dynamically create them as they are used
- **Implementation of VALUES and var2value: hash tables**

Example

```
Assign: a->r1,b->r2,c->r3,d->r4
a = b+c;     ADD t1 = r2,r3
             CPY r1 = t1
b = a-d;     SUB t2 = r1,r4
             CPY r2 = t2
c = b+c;     ADD t3 = r2,r3
             CPY r3 = t3
d = a-d;     SUB t4 = r1,r4
             CPY r4 = t4
```

Conclusions

- **Comparisons of two abstractions**
  - DAGs
  - Value numbering
- **Value numbering**
  - VALUE: distinguish between variables and VALUES
  - TIME
    - Interpretation of instructions in order of execution
    - Keep dynamic state information

VN Example

```
Assign: a->r1,b->r2,c->r3,d->r4
a = b+c;      ADD t1 = r2,r3
              CPY r1 = t1   //(a = t1)
b = a-d;      SUB t2 = r1,r4
              CPY r2 = t2   //(b = t2)
c = b+c;      ADD t3 = r2,r3
              CPY r3 = t3   //(c = t3)
d = a-d;      CPY r~~2~~4 = t2
```

Outline

1. Structure of data flow analysis

2. Example 1: Reaching definition analysis

3. Example 2: Liveness analysis

4. Generalization

What is Data Flow Analysis?

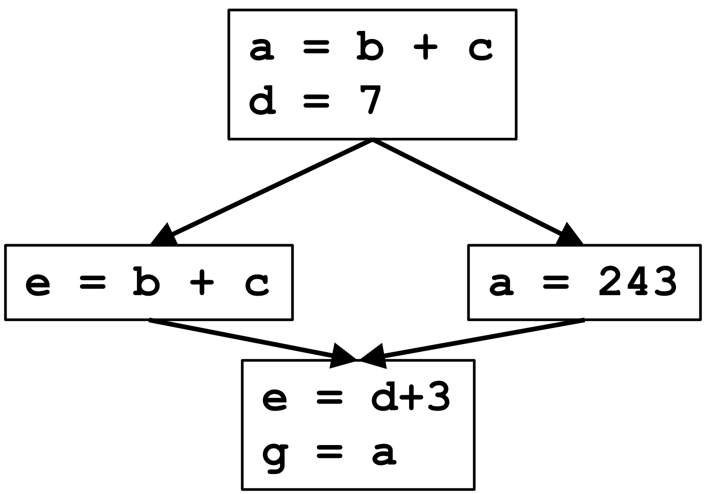
- **Local analysis (e.g., value numbering)**
  - analyze effect of each instruction
  - compose effects of instructions to derive information from beginning of basic block to each instruction

- **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

What is Data Flow Analysis? (2)

- **Data flow analysis:**
  - Flow-sensitive: sensitive to the control flow in a function
  - intraprocedural analysis
- **Examples of optimizations:**
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

What is Data Flow Analysis? (3)



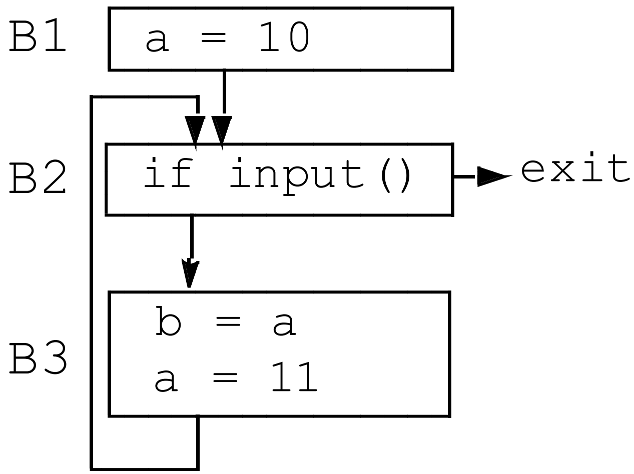
For each variable x determine:

Value of x?

Which “definition” defines x?

Is the definition still meaningful (live)?

Static Program vs. Dynamic Execution



- **Statically**: Finite program
- **Dynamically**: Can have infinitely many possible execution paths
- **Data flow analysis abstraction:**
  - For each point in the program:
    combines information of all the instances of the same program point.
- **Example of a data flow question:**
  - Which definition defines the value used in statement "b = a"?

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.

Effects of a Basic Block

A **locally available definition** = last definition of data item in b.b.

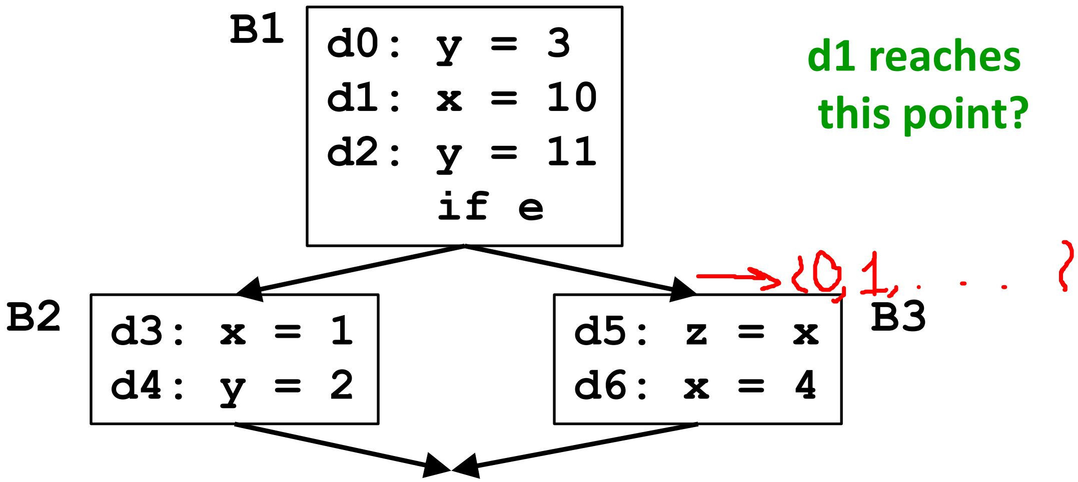
```
t1 = r1+r2
r2 = t1
t2 = r2+r1
r1 = t2
t3 = r1*r1
r2 = t3
if r2>100 goto L1
```

Locally exposed uses? **r1**

Kills any definitions? **Any other definition of t2**

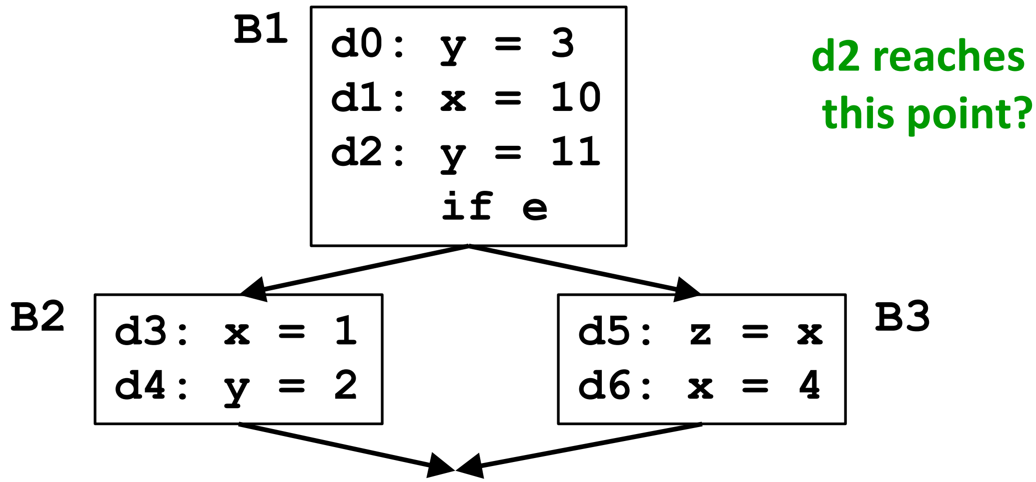
Locally avail. definition? **t2**

Reaching Definitions



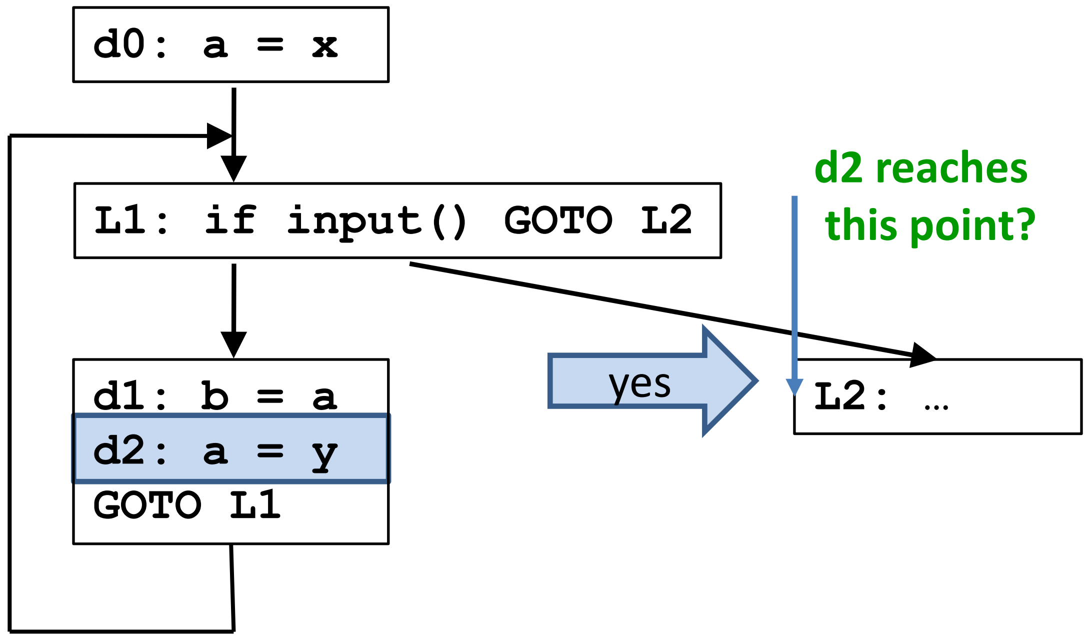
- Every assignment is a **definition**
- A **definition** *d* **reaches** a point *p* if **there exists** path from the point immediately following *d* to *p* such that *d* is not killed (overwritten) along that path.
- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point, vector-length = #defs

Reaching Definitions (2)

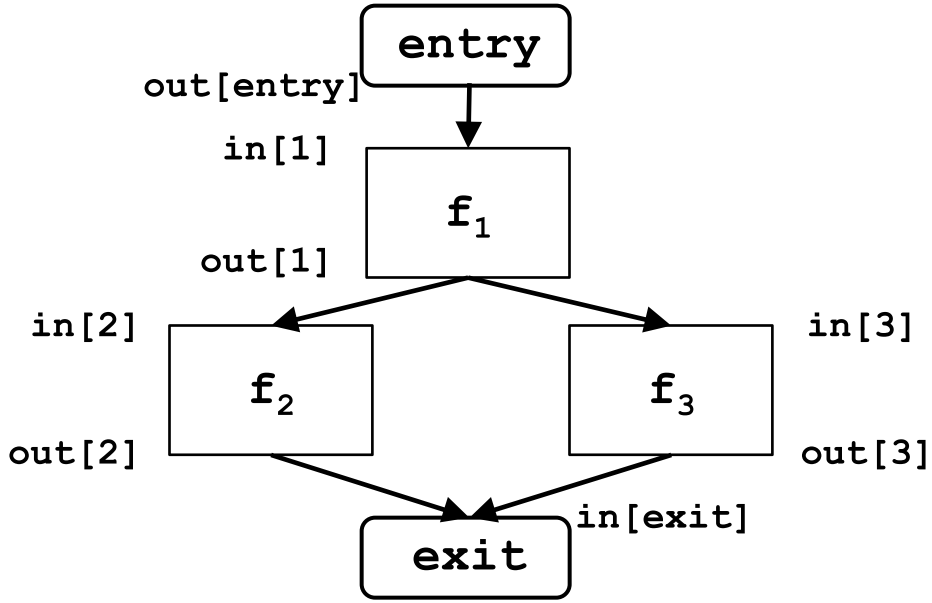


- Every assignment is a **definition**
- A **definition** *d* **reaches** a point *p*
  if **there exists** path from the point immediately following *d* to *p*
  such that *d* is not killed (overwritten) along that path.
- Problem statement
  - For each point in the program, determine if each definition in the program reaches the point
  - A bit vector per program point, vector-length = #defs

Reaching Definitions (3)

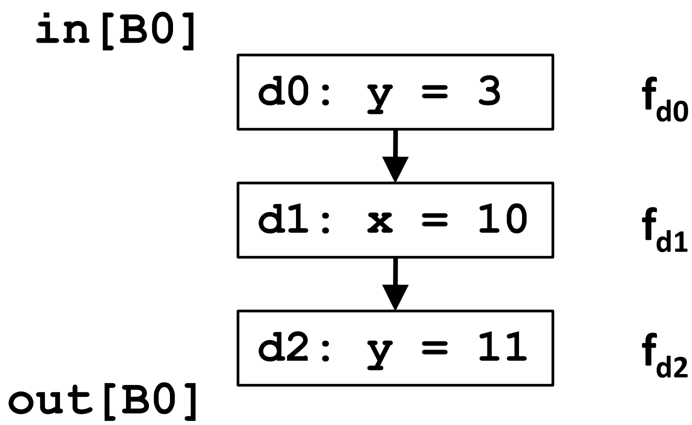


Data Flow Analysis Schema



- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between in[b] and out[b] for all basic blocks b
  - Effect of code in basic block:
    - Transfer function f_b relates in[b] and out[b], for same b
  - Effect of flow of control:
    - relates out[b_1], in[b_2] if b_1 and b_2 are adjacent
- Find a solution to the equations

Effects of a Statement



- f_s : A transfer function of a statement
  - abstracts the execution with respect to the problem of interest
- For a statement s (d: x = y + z)
  out[s] = f_s(in[s]) = Gen[s] U (in[s]-Kill[s])
  - **Gen[s]**: definitions generated: Gen[s] = {d}
  - **Propagated** definitions: in[s] - Kill[s],
    where **Kill[s]**=set of all other defs to x in the rest of program

Effects of a Basic Block

```
in[B0]
d0: y = 3      f_d0
d1: x = 10     f_d1
d2: y = 11     f_d2
out[B0]
```

$f_B = f_{d2} \cdot f_{d1} \cdot f_{d1}$

- Transfer function of a statement s:
  - $out[s] = f_s(in[s]) = Gen[s] \cup (in[s]-Kill[s])$
- Transfer function of a basic block B:
  - Composition of transfer functions of statements in B
- $out[B] = f_B(in[B]) = f_{d2}f_{d1}f_{d0}(in[B])$

$= Gen[d_2] \cup (Gen[d_1] \cup (Gen[d_0] \cup (in[B]-Kill[d_0]))-Kill[d_1]))\ -Kill[d_2]$

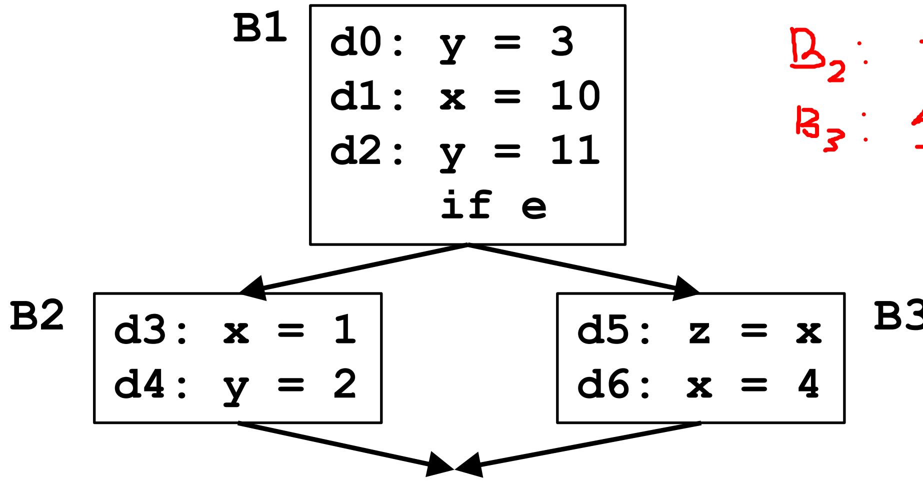
$= Gen[d_2] \cup (Gen[d_1] \cup (Gen[d_0] - Kill[d_1]) - Kill[d_2]) \cup$
$in[B] - (Kill[d_0] \cup Kill[d_1] \cup Kill[d_2])$

$= Gen[B] \cup (in[B] - Kill[B])$

- Gen[B]: locally exposed definitions (available at end of bb)
- Kill[B]: set of definitions killed by B

Example

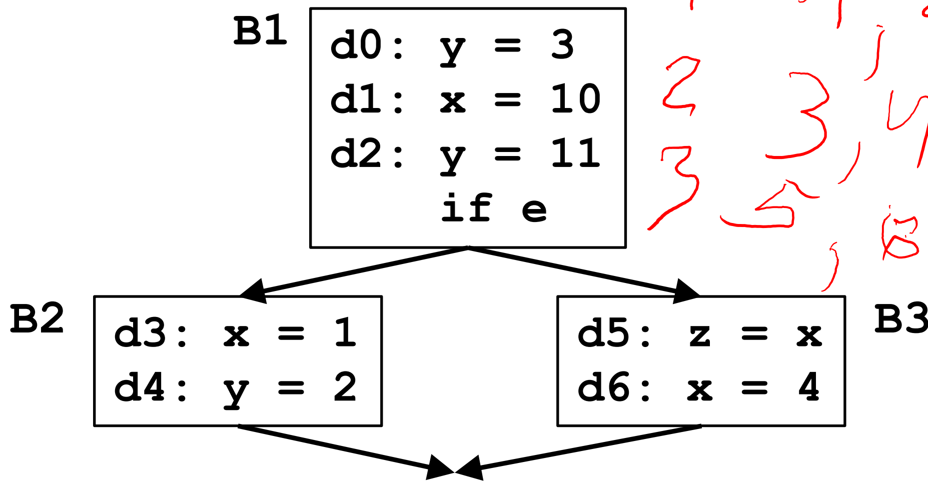
	Gen	Kill
B_1:	1, 2	0, 3, 4, 6, 2
B_2:	3 4	0 1 2 6
B_3:	5 6	1 3



- 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
  - **generates** definitions: Gen[b],
    – set of locally available definitions in b
  - **kills** 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])**

Example

Gen Kill



1   1, 2   B1   0, 2, 3, 4, 6

2   3, 4   B2   0, 1, 2, 6

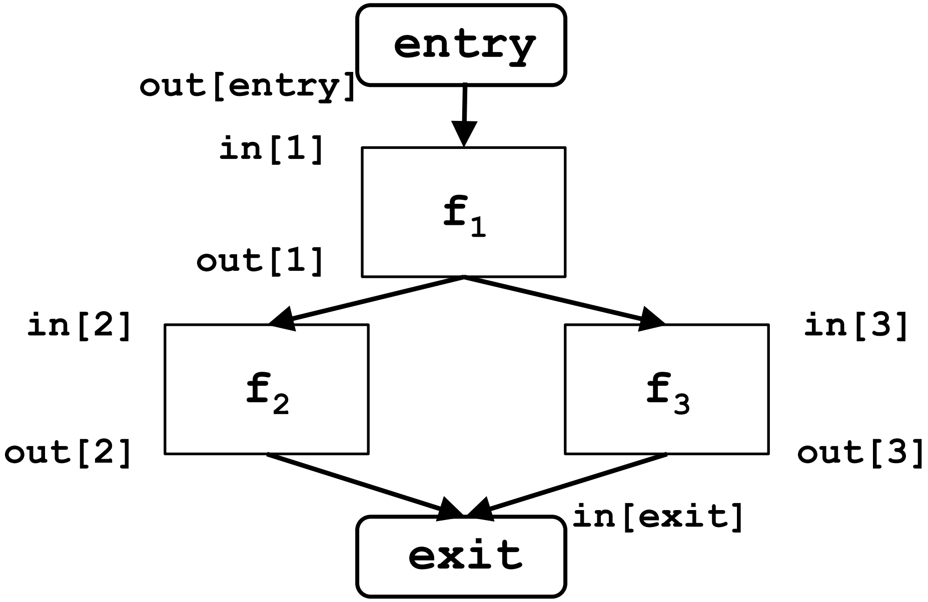
3   5, 6   B3   1, 3

- 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
  - **generates** definitions: Gen[b],
    - set of locally available definitions in b
  - **kills** 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)

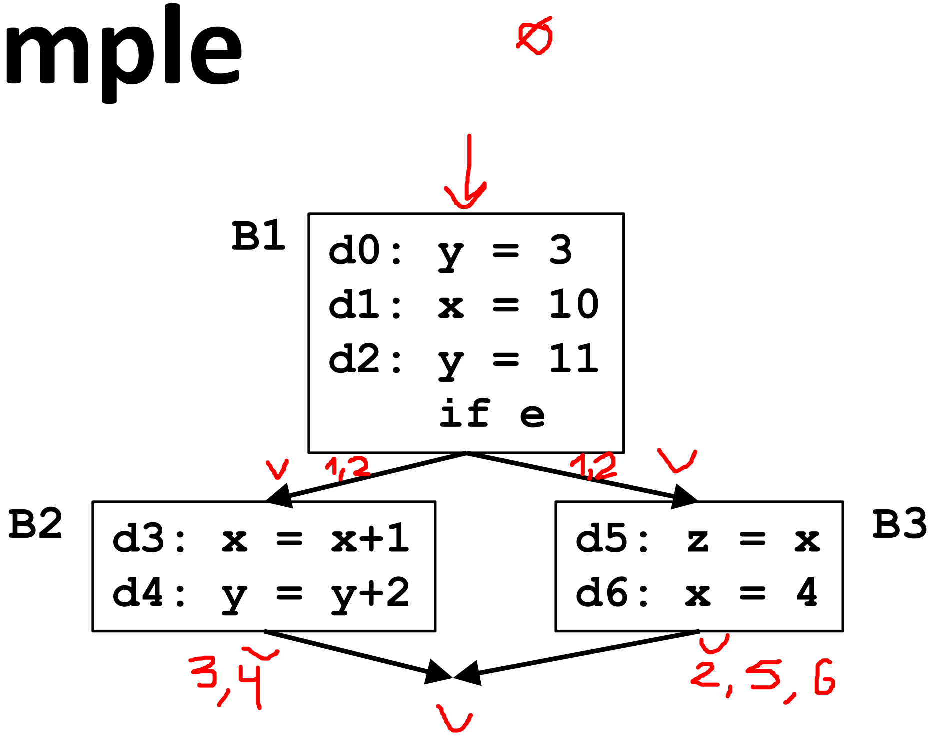


- out[b] = f_b(in[b])
- Join node: a node with multiple predecessors
- **meet** operator:

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

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

Example



f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

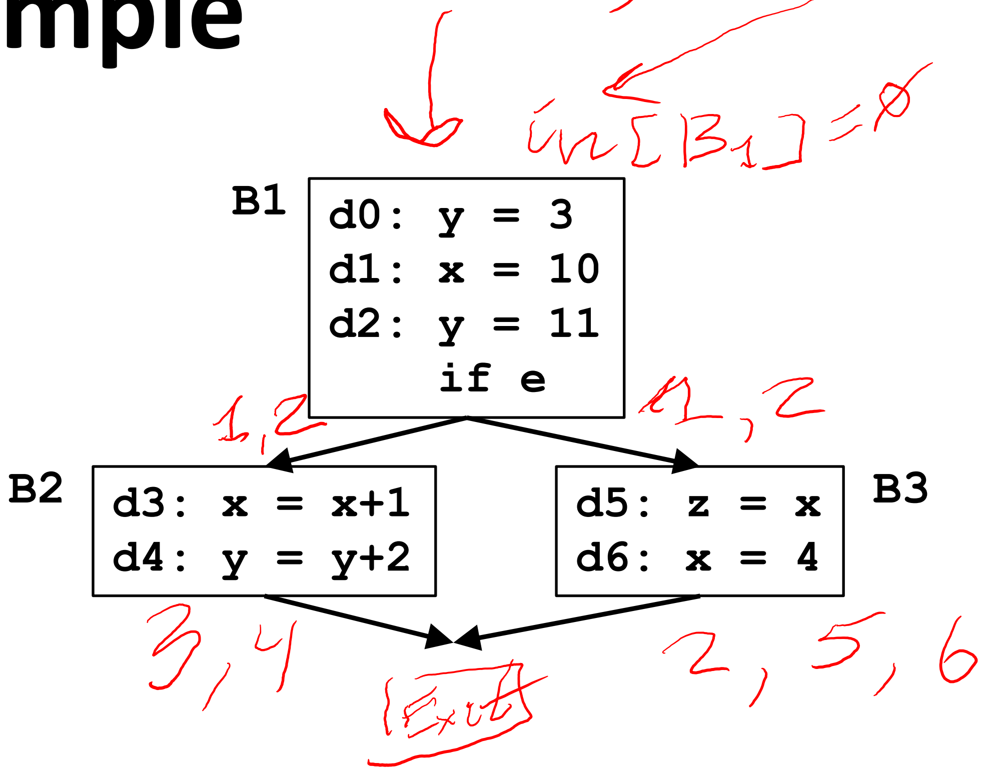
- out[b] = f_b(in[b])
- Join node: a node with multiple predecessors
- **meet** operator:

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

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

Example

Entry = out[E] = ∅



f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

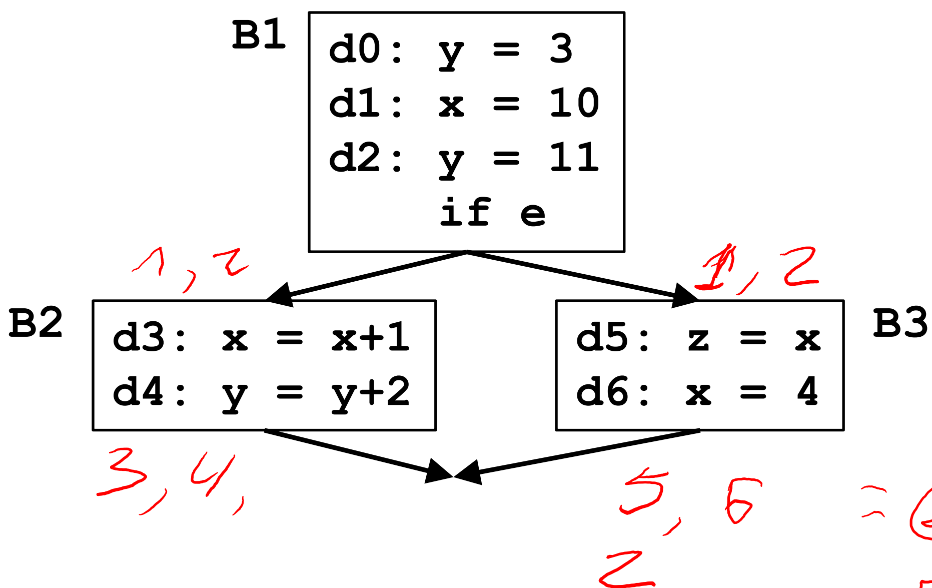
- out[b] = f_b(in[b])
- Join node: a node with multiple predecessors
- **meet** operator:

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

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

Example

E out[E] = ∅
in[B_1] = ∅



f	Gen	Kill
1	{1,2}	{0,2,3,4,6}
2	{3,4}	{0,1,2,6}
3	{5,6}	{1,3}

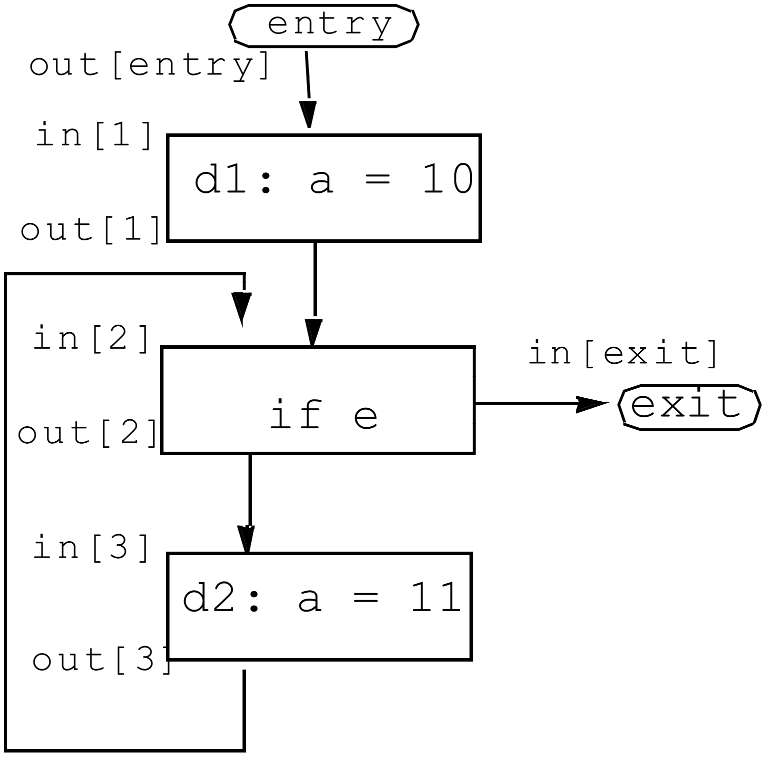
out[B_1] = Gen[B1] ∪ [in[B1] - Kill[B_1]]

- 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.
- Find: fixed point solution

Reaching Definitions: Iterative Algorithm

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

// Boundary condition
   out[Entry] = ∅

// Initialization for iterative algorithm
   For each basic block B other than Entry
      out[B] = ∅

// iterate
   While (Changes to any out[] occur) {
      For each basic block B other than Entry {
         in[B] = ∪ (out[p]), for all predecessors p of B
         out[B] = f_B(in[B])    // out[B]=gen[B]∪(in[B]-kill[B])
      }
```

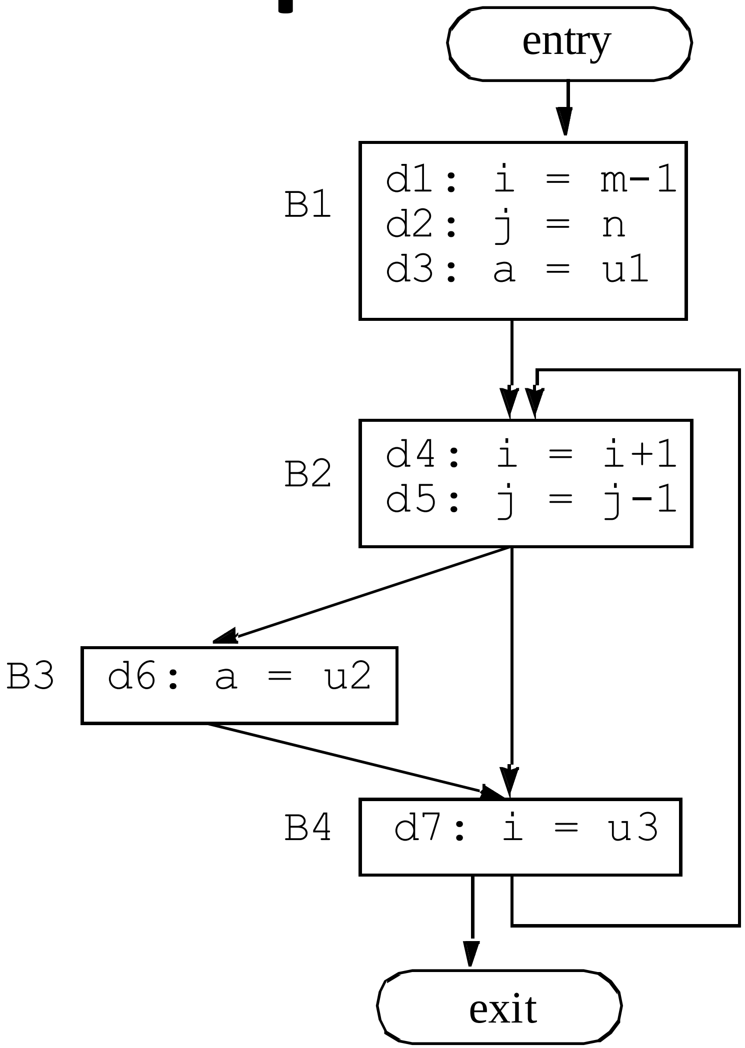
Reaching Definitions: Worklist Algorithm

```
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 ChangedNodes
        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

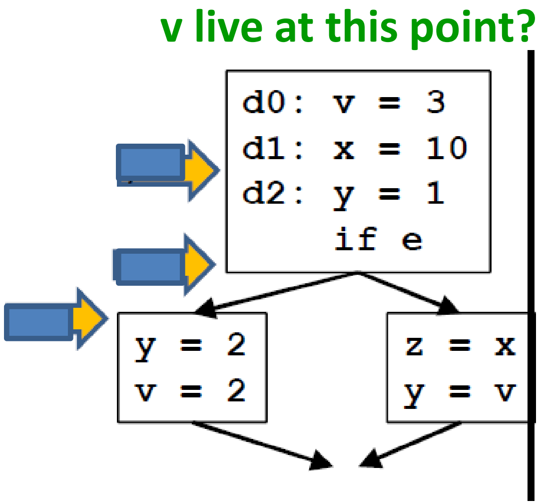


	First Pass	Second Pass
IN[B1]	000 00 0 0	000 00 0 0
OUT[B1]	111 00 0 0	111 00 0 0
IN[B2]	111 00 0 0	111 01 1 1
OUT[B2]	001 11 0 0	001 11 1 0
IN[B3]	001 11 0 0	001 11 1 0
OUT[B3]	000 11 1 0	000 11 1 0
IN[B4]	001 11 1 0	001 11 1 0
OUT[B4]	001 01 1 1	001 01 1 1
IN[exit]	001 01 1 1	001 01 1 1

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 …
```

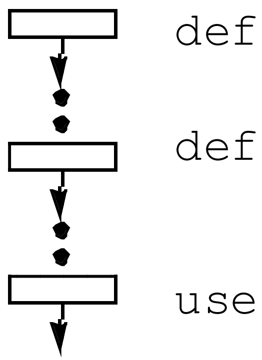


- **Problem statement**
  - For each basic block
    - determine if each variable is live in each basic block
  - Size of bit vector: one bit for each variable

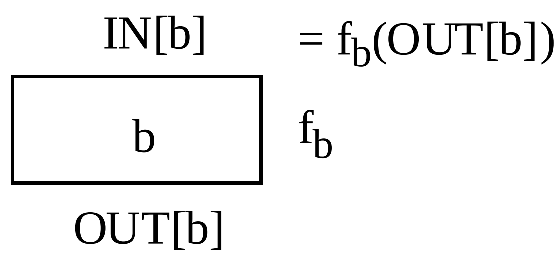
Transfer Function

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

an execution path



control flow



example

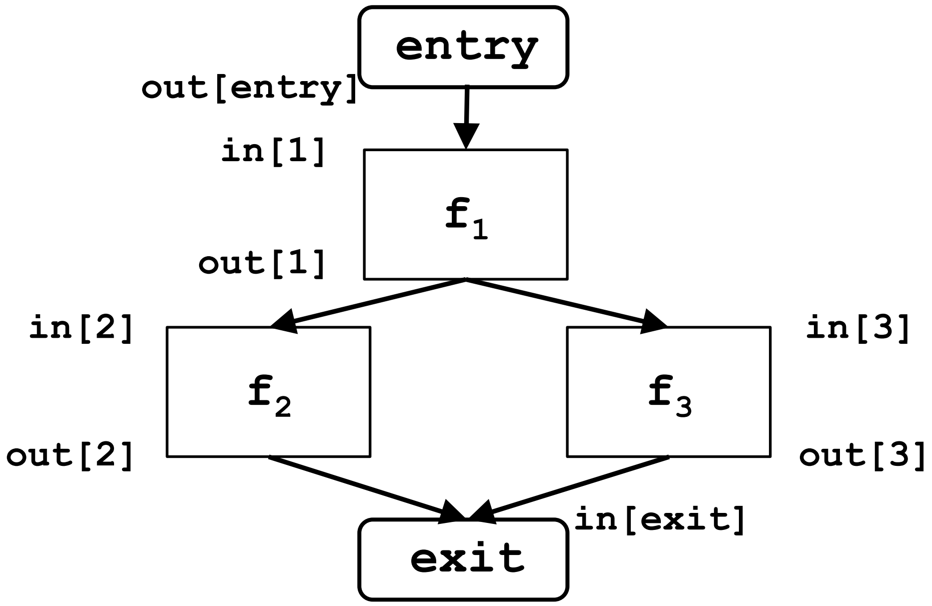
```
d3: a = 1
d4: b = 1

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

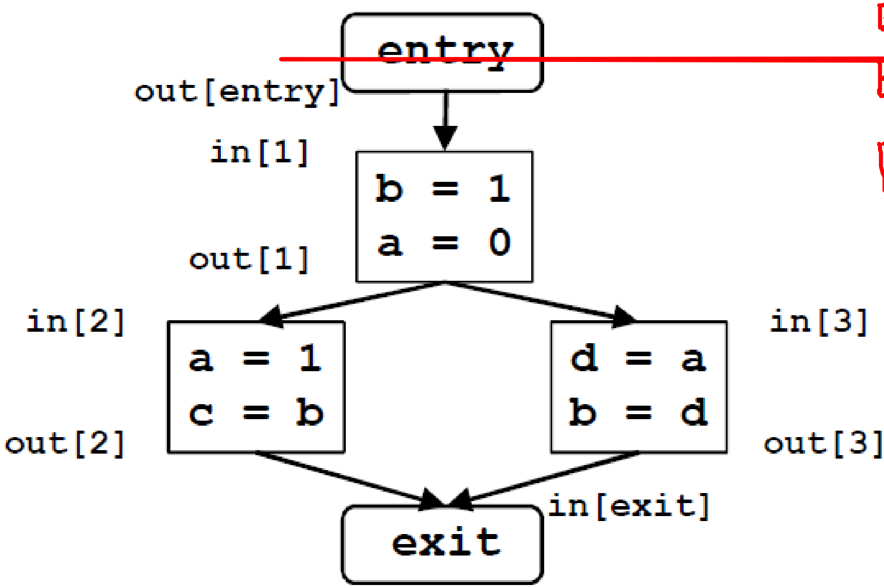


- in[b] = f_b(out[b])
- Join node: a node with multiple successors
- **meet** operator:

  out[b] = in[s_1] U in[s_2] U ... U in[s_n], where

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

Flow Graph (2)



Use Def

B_1: ∅
B_2: b
B_3: a

f	Use	Def
1	{}	{a,b}
2	{b}	{a,c}
3	{a}	{b,d}

- 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

Liveness: Iterative Algorithm

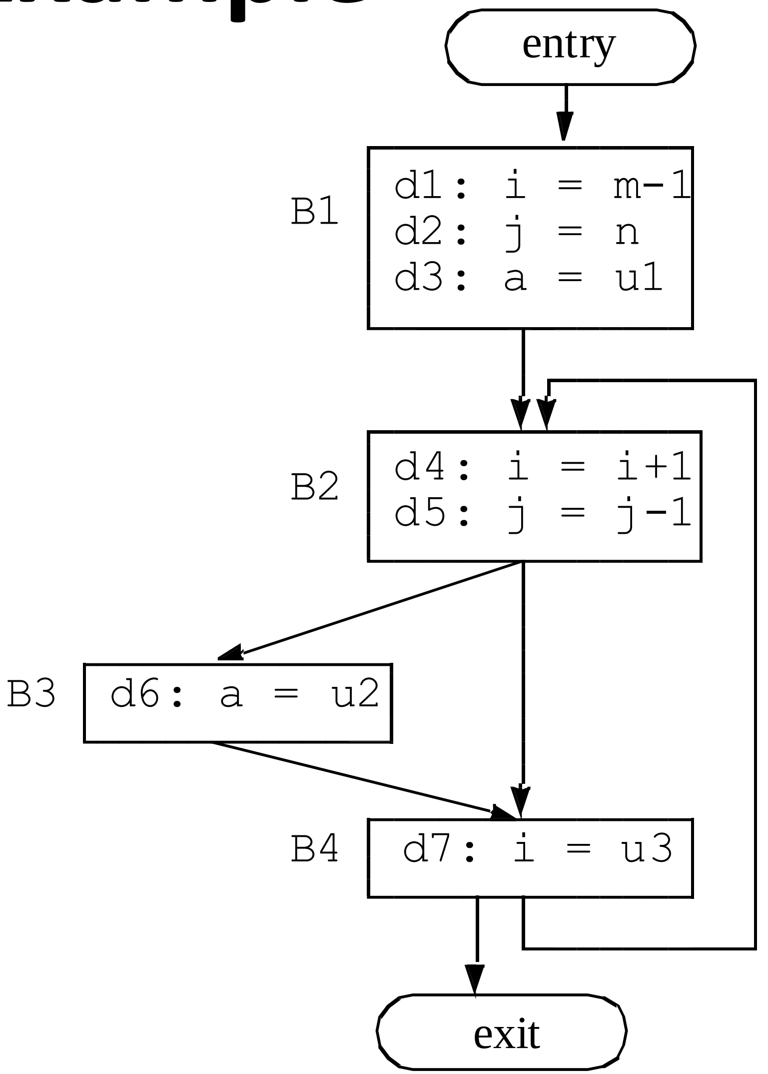
```
input: control flow graph CFG = (N, E, Entry, Exit)

// Boundary condition
   in[Exit] = ∅

// Initialization for iterative algorithm
   For each basic block B other than Exit
      in[B] = ∅

// iterate
   While (Changes to any in[] occur) {
      For each basic block B other than Exit {
         out[B] = ∪ (in[s]), for all successors s of B
         in[B] = f_B(out[B])    // in[B]=Use[B]∪(out[B]-Def[B])
      }
```

Example



	First Pass	Second Pass
OUT[entry]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
IN[B1]	{m,n,u1,u2,u3}	{m,n,u1,u2,u3}
OUT[B1]	{i,j,u2,u3}	{i,j,u2,u3}
IN[B2]	{i,j,u2,u3}	{i,j,u2,u3}
OUT[B2]	{u2,u3}	{j,u2,u3}
IN[B3]	{u2,u3}	{j,u2,u3}
OUT[B3]	{u3}	{j,u2,u3}
IN[B4]	{u3}	{j,u2,u3}
OUT[B4]	{}	{i,j,u2,u3}

Framework

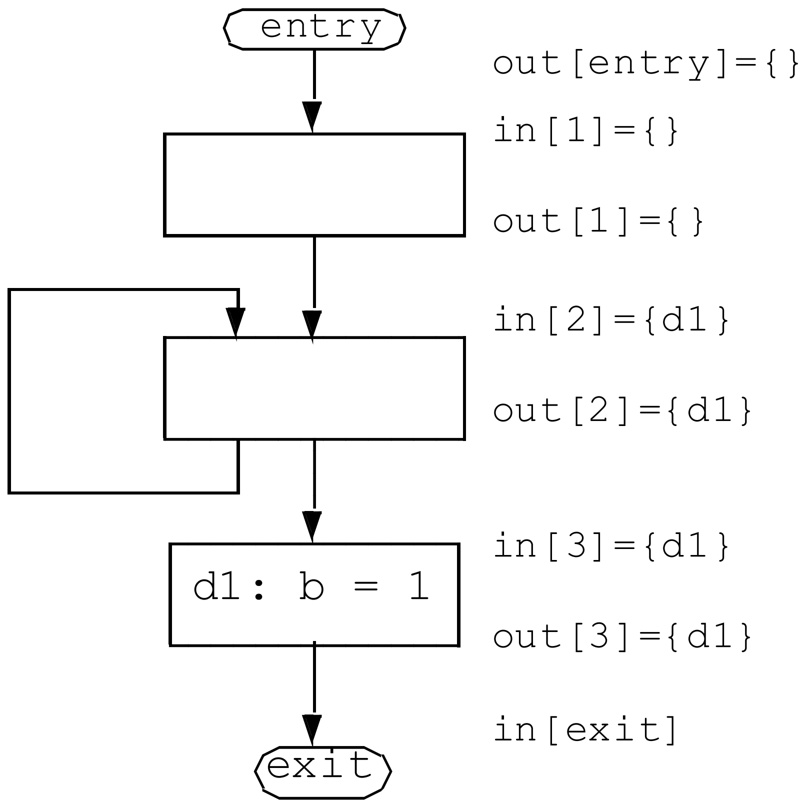
	Reaching Definitions	**Live Variables**
Domain	Sets of definitions	Sets of variables
Direction	forward: $out[b] = f_b(in[b])$ $in[b] = \wedge\ out[pred(b)]$	backward: $in[b] = f_b(out[b])$ $out[b] = \wedge\ in[succ(b)]$
Transfer function	$f_b(x) = Gen_b \cup (x - Kill_b)$	$f_b(x) = Use_b \cup (x - Def_b)$
Meet Operation ($\wedge$)	$\cup$	$\cup$
Boundary Condition	$out[entry] = \emptyset$	$in[exit] = \emptyset$
Initial interior points	$out[b] = \emptyset$	$in[b] = \emptyset$

Other examples (e.g., Available expressions), defined in ALSU 9.2.6

Thought Problem 1. “Must-Reach” Definitions

- **A definition D (a = b+c) <u>must</u> reach point P iff**
  - D appears at least once along on all paths leading to P
  - a is not redefined along any path after last appearance of D and before P
- **How do we formulate the data flow algorithm for this problem?**

Thought Problem 2: A legal solution to (May) Reaching Def?



- Will the worklist algorithm generate this answer?

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?

Foundations of Data Flow Analysis

1. **Meet operator**
2. **Transfer functions**
3. **Correctness, Precision, Convergence**
4. **Efficiency**

- Reference: ALSU pp. 613-631
- Background: Hecht and Ullman, Kildall, Allen and Cocke[76]
- Marlowe & Ryder, Properties of data flow frameworks: a unified model. Rutgers tech report, Apr. 1988

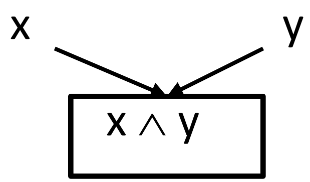
A Unified Framework

- **Data flow problems are defined by**
  - Domain of values: V
  - Meet operator ($V \wedge V \rightarrow V$), initial value
  - A set of transfer functions ($V \rightarrow V$)

- **Usefulness of unified framework**
  - To answer questions such as correctness, precision, convergence, speed of convergence for a family of problems
    - If meet operators and transfer functions have properties X, then we know Y about the above.
  - Reuse code

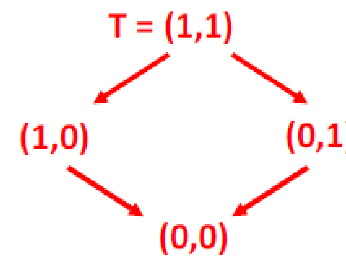
Meet Operator

- **Properties of the meet operator**
  - commutative: $x \wedge y = y \wedge x$



  - idempotent: $x \wedge x = x$
  - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
  - there is a Top element T such that $x \wedge T = x$

Partial Order

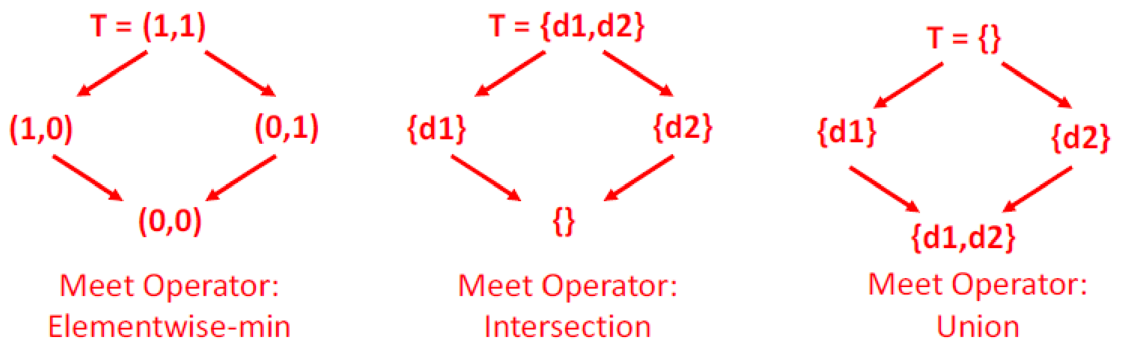


Meet Operator:
Elementwise-min

- **Meet operator defines a partial ordering on values**
  - $x \leq y$ if and only if $x \wedge y = x$ (y -> x in diagram)
    - Transitivity: if $x \leq y$ and $y \leq z$ then $x \leq z$
    - Antisymmetry: if $x \leq y$ and $y \leq x$ then $x = y$
    - Reflexivity: $x \leq x$

Partial Order

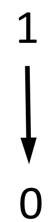
- Example: let **V** = {x | such that x $\subseteq$ { $\mathbf{d_1}$, $\mathbf{d_2}$}}, $\wedge$ = $\cap$



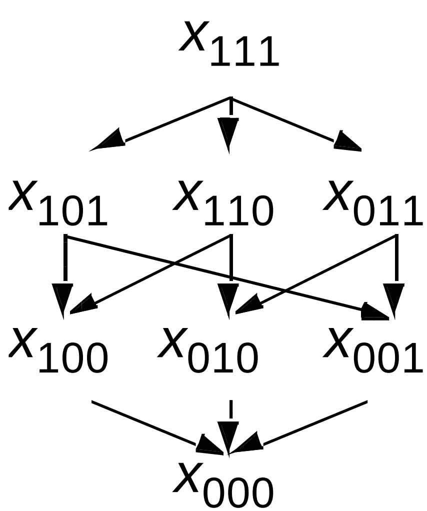
- Top and Bottom elements
  - Top T such that: $x \wedge T = x$
  - Bottom $\perp$ such that: $x \wedge \perp = \perp$
- Values and meet operator in a data flow problem define a semi-lattice:
  – there exists a T, but not necessarily a $\perp$.
- x, y are ordered: $x \leq y$ then $x \wedge y = x$ (y -> x in diagram)
- what if x and y are not ordered?
  - $x \wedge y \leq x$, $x \wedge y \leq y$, and if $w \leq x$, $w \leq y$, then $w \leq x \wedge y$

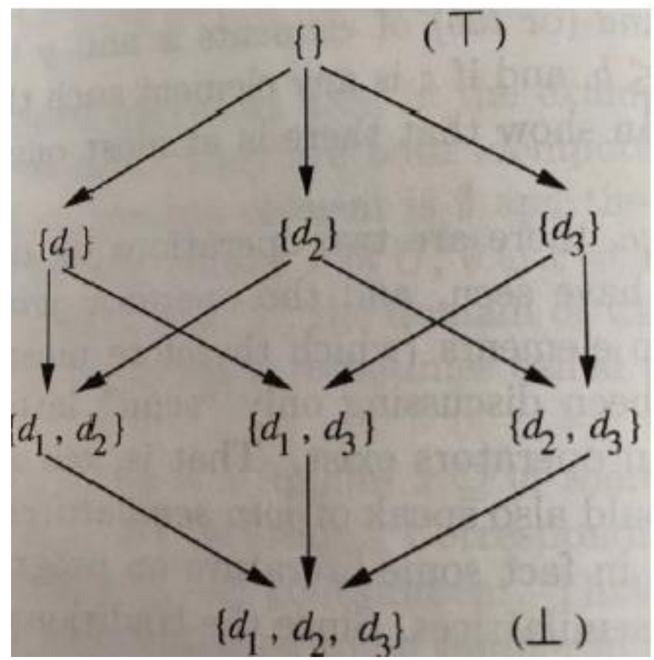
One vs. All Variables/Definitions

- **Lattice for each variable: e.g. intersection**



- **Lattice for three variables:**





Descending Chain

- **Definition**
  - The **height** of a lattice is the largest number of **> relations** that will fit in a descending chain.

    $x_0 > x_1 > x_2 > \ldots$

- **Height of values in reaching definitions?**

  Height n – number of definitions

- **Important property: finite descending chain**
- **Can an infinite lattice have a finite descending chain?**

  yes

- **Example: Constant Propagation/Folding**
  - To determine if a variable is a constant
- **Data values**
  - undef, ... -1, 0, 1, 2, ..., not-a-constant

Transfer Functions

- **Basic Properties** f**: V $\rightarrow$ V**
  - Has an identity function
    - There exists an f such that f (x) = x, for all x.
  - Closed under composition
    - if $f_1, f_2 \in F$, then $f_1 \cdot f_2 \in F$

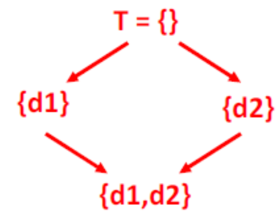
Monotonicity

- A framework (F, V, $\wedge$) is monotone if and only if
  - $x \leq y$ implies $f(x) \leq f(y)$
  - i.e. a "smaller or equal" input to the same function will always give a "smaller or equal" output
- Equivalently, a framework (F, V, $\wedge$) is monotone if and only if
  - $f(x \wedge y) \leq f(x) \wedge f(y)$
  - i.e. merge input, then apply *f* is **small than or equal to** apply the transfer function individually and then merge the result

Example

- **Reaching definitions: f(x) = Gen ∪ (x - Kill), ∧ = ∪**
  - Definition 1:
    - $x_1 \leq x_2$, Gen ∪ (x_1 - Kill) ≤ Gen ∪ (x_2 - Kill)
  - Definition 2:
    - (Gen ∪ (x_1 - Kill)) ∪ (Gen ∪ (x_2 - Kill))

      = (Gen ∪ (($x_1 \cup x_2$) - Kill))



Meet Operator: Union

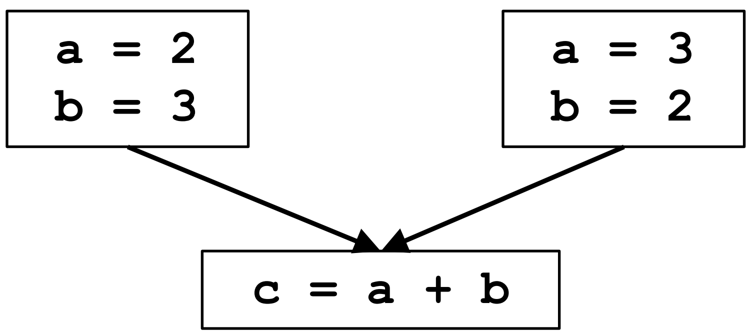
$[x_1 \leq x_2 \text{ iff } x_2 \rightarrow x_1]$

- **Note: Monotone framework does not mean that f(x) ≤ x**
    - e.g., reaching definition for two definitions in program
    - suppose: f_x: $Gen_x = \{d_1, d_2\}$; $Kill_x = \{\}$

- **If input(second iteration) ≤ input(first iteration)**
    - result(second iteration) ≤ result(first iteration)

Distributivity

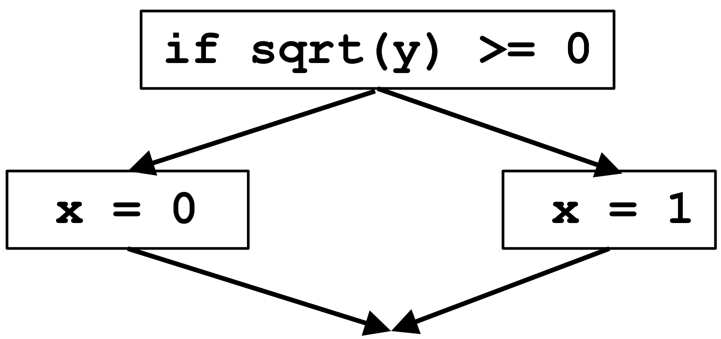
- A framework ($F, V, \wedge$) is **distributive** if and only if
  - $f(x \wedge y) = f(x) \wedge f(y)$
  - i.e. merge input, then apply f is **equal to** apply the transfer function individually then merge result
- Example: Constant Propagation is NOT distributive



Data Flow Analysis

- **Definition**
  - Let $f_1, \ldots, f_m : \in F$, where f_i is the transfer function for node i
    - $f_p = f_{n_k} \cdot \ldots \cdot f_{n_1}$, where p is a path through nodes $n_1, \ldots, n_k$
    - f_p = identify function, if p is an empty path
- **Ideal data flow answer:**
  - For each node n:

    $\wedge f_{p_i}(\mathrm{T})$, for all possibly executed paths p_i reaching n.



- **But determining all possibly executed paths is undecidable**

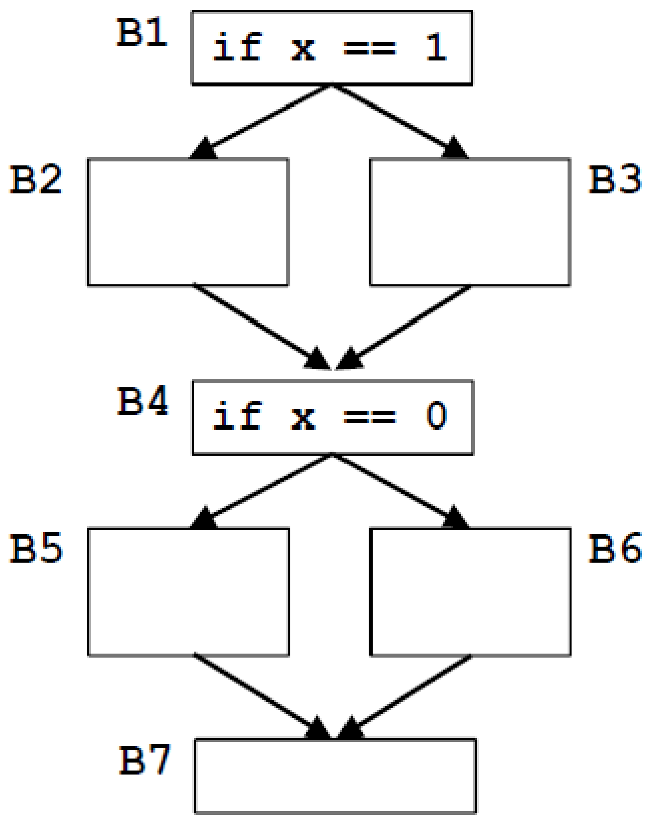
Meet-Over-Paths (MOP)

- **Error in the conservative direction**
- **Meet-Over-Paths (MOP):**
  - For each node *n:*

    $\text{MOP}(n) = \wedge f_{p_i}(\text{T})$, for all paths p_i reaching n

  - a path exists as long there is an edge in the code
  - consider more paths than necessary
  - MOP = Perfect-Solution $\wedge$ Solution-to-Unexecuted-Paths
  - MOP ≤ Perfect-Solution
  - Potentially more constrained, solution is small
    - hence *conservative*
  - It is not **safe** to be > Perfect-Solution!
- **Desirable solution: as close to MOP as possible**

MOP Example



Assume: B2 & B3 do not update x

Ideal: Considers only 2 paths
B1-B2-B4-B6-B7 (i.e., x=1)
B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths
B1-B2-B4-B5-B7
B1-B3-B4-B6-B7

Solving Data Flow Equations

- **Example: Reaching definitions**
  - out[entry] = {}
  - Values = {subsets of definitions}
  - Meet operator: $\cup$
    - in[b] = $\cup$ out[*p*], for all predecessors *p* of b
  - Transfer functions: out[b] = $gen_b \cup (in[b] - kill_b)$
- **Any solution satisfying equations = Fixed Point Solution (FP)**
- **Iterative algorithm**
  - initializes out[b] to {}
  - if converges, then it computes Maximum Fixed Point (MFP):
    - MFP is the largest of all solutions to equations
- **Properties:**
  - FP ≤ MFP ≤ MOP ≤ Perfect-solution
  - FP, MFP are safe
  - in(b) ≤ MOP(b)

Partial Correctness of Algorithm

- **If data flow framework is monotone, then if the algorithm converges, IN[b] ≤ MOP[b]**
- **Proof: Induction on path lengths**
  - Define IN[entry] = OUT[entry] and transfer function of entry = Identity function
  - Base case: path of length 0
    - Proper initialization of IN[entry]
  - If true for path of length k, $p_k = (n_1, ..., n_k)$, then true for path of length k+1: $p_{k+1} = (n_1, ..., n_{k+1})$
    - Assume: $IN[n_k] \le f_{n_{k-1}}(f_{n_{k-2}}(\ldots f_{n_1}(IN[entry])))$
    - $IN[n_{k+1}] = OUT[n_k] \wedge \ldots$

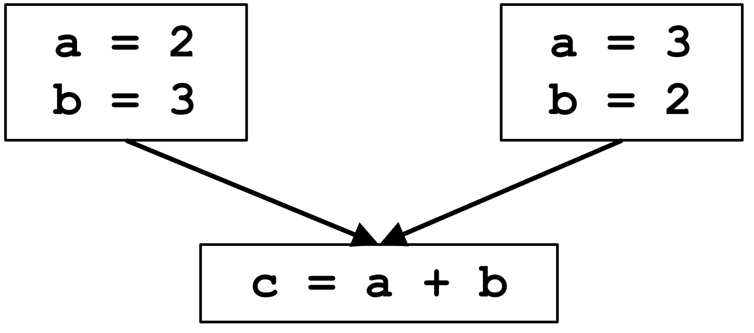
      $\le OUT[n_k]$

      $\le f_{n_k}(IN[n_k])$

      $\le f_{n_{k-1}}(f_{n_{k-2}}(\ldots f_{n_1}(IN[entry])))$

Precision

- **If data flow framework is distributive,then if the algorithm converges, IN[b] = MOP[b]**



- Monotone but not distributive: behaves as if there are additional paths

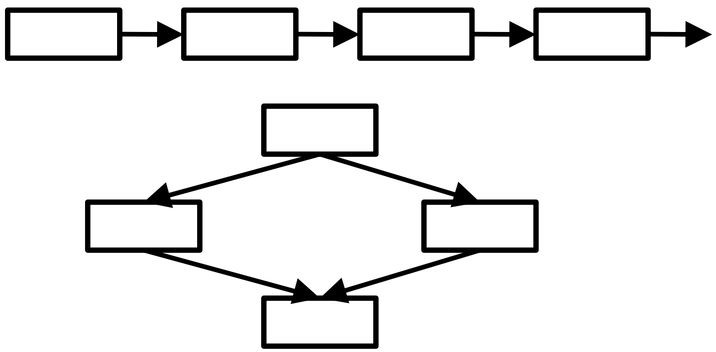
Additional Property to Guarantee Convergence

- **Data flow framework (monotone) converges if there is a finite descending chain**

- For each variable IN[b], OUT[b], consider the sequence of values set to each variable across iterations:
  - if sequence for in[b] is monotonically decreasing
    - sequence for out[b] is monotonically decreasing
      - (out[b] initialized to T)
  - if sequence for out[b] is monotonically decreasing
    - sequence of in[b] is monotonically decreasing

Speed of Convergence

- Speed of convergence depends on order of node visits



- Reverse “direction” for backward flow problems

Reverse Postorder

- **Step 1: depth-first post order**

```
main() {
    count = 1;
    Visit(root);
}
Visit(n) {
    for each successor s that has not been visited
        Visit(s);
    PostOrder(n) = count;
    count = count+1;
}
```

- **Step 2: reverse order**

```
For each node i
    rPostOrder = NumNodes - PostOrder(i)
```

Depth-First Iterative Algorithm (forward)

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

/* Initialize */
    out[entry] = init_value
    For all nodes i
       out[i] = ⊤
    Change = True

/* iterate */
    While Change {
       Change = False
       For each node i in rPostOrder {
          in[i] = ∧(out[p]), for all predecessors p of i
          oldout = out[i]
          out[i] = f_i(in[i])
          if oldout ≠ out[i]
             Change = True
       }
    }
```

Speed of Convergence

- **If cycles do not add information**
  - information can flow in one pass down a series of nodes of increasing order number:
    - e.g., 1 -> 4 -> 5 -> 7 -> 2 -> 4 ...
  - passes determined by number of back edges in the path
    - essentially the nesting depth of the graph
  - Number of iterations = number of back edges in any acyclic path + 2
    - (2 are necessary even if there are no cycles)

- **What is the depth?**
  – corresponds to depth of intervals for “reducible” graphs
  – in real programs: average of 2.75

A Check List for Data Flow Problems

- **Semi-lattice**
  - set of values
  - meet operator
  - top, bottom
  - finite descending chain?
- **Transfer functions**
  - function of each basic block
  - monotone
  - distributive?
- **Algorithm**
  - initialization step (entry/exit, other nodes)
  - visit order: rPostOrder
  - depth of the graph

Conclusions

- Dataflow analysis examples
  - Reaching definitions
  - Live variables

- Dataflow formation definition
  - Meet operator
  - Transfer functions
  - Correctness, Precision, Convergence
  - Efficiency

CSC D70: Compiler Optimization Dataflow Analysis

Prof. Gennady Pekhimenko
University of Toronto

Winter 2021

The content of this lecture is adapted from the lectures of Todd Mowry and Phillip Gibbons