Data-Flow Analysis

Jaejin Lee
School of Computer Science and Engineering
Seoul National University
http://aces.snu.ac.kr

Basic Blocks

- A sequence of statements that is always entered at the beginning and exited at the end without halt or possibility of branching except at the end
- Two consecutive instructions are in the same basic block iff the execution of the first instruction guarantees the execution of the next instruction
- For intermediate representation, such as three-address statements

```
    read m
    f0 = 0
    f1 = 1
    if m <= 1 goto L3
----------------------
    i = 2
----------------------
L1: if i <= m goto L2
----------------------
    return f2
----------------------
L2: f2 = f0 + f1
    f0 = f1
    f1 = f2
    i = i + 1
    goto L1
----------------------
L3: return m
```

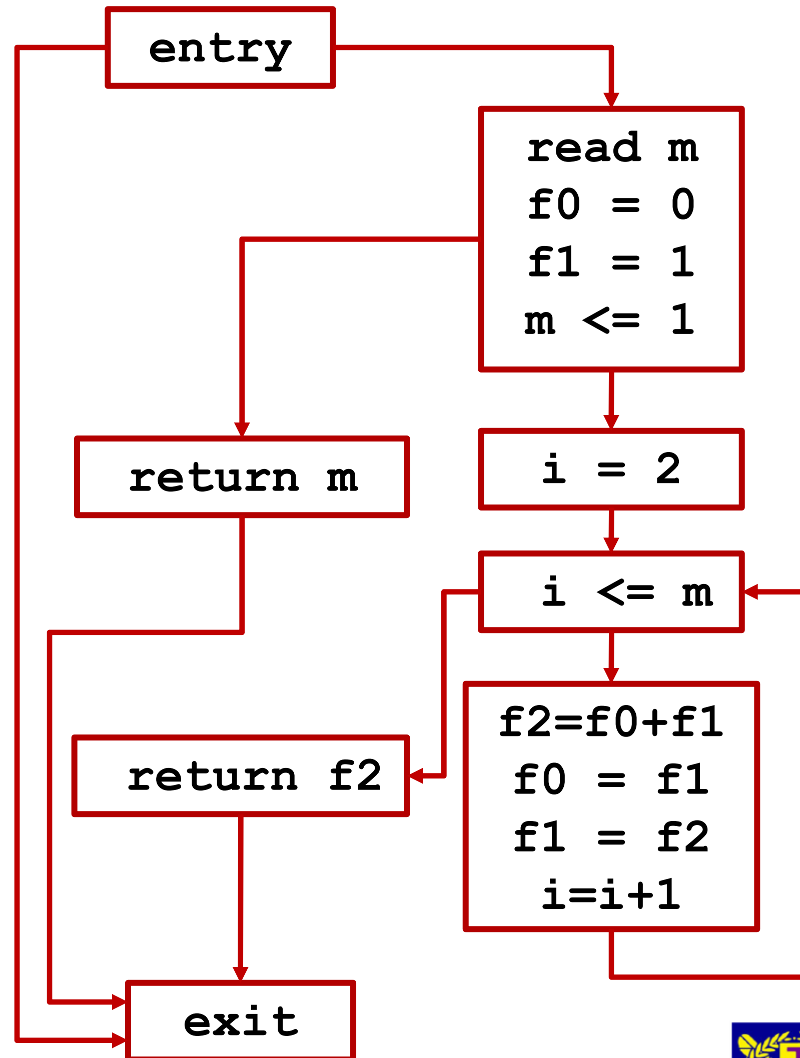
Finding Basic Blocks

1. The first instruction is a leader
2. Any instruction that is the target of a conditional or unconditional jump is a leader
3. Any instruciton that is immediately follows a conditional or unconditional jump is a leader

- For each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program

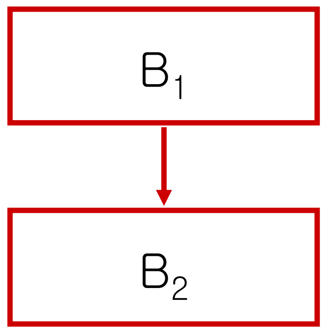
Control-Flow Graphs

- Flow-of-control information
- Directed graph
- CFG = (V, E, Entry, Exit)
- V = the set of basic blocks U {Entry, Exit}
  - Entry is the unique program entry
  - Exit is the unique program exit
- Edges in E represent potential flow of control
  - There is a directed edge from B_1 to B_2 if B_2 can immediately follow B_1 in some execution sequence
  - An edge from Entry to Exit



Edges in CFGs

- There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2, or
- B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump
- B_1 – predecessor of B_2
- B_2 – successor of B_1



Code Optimizations

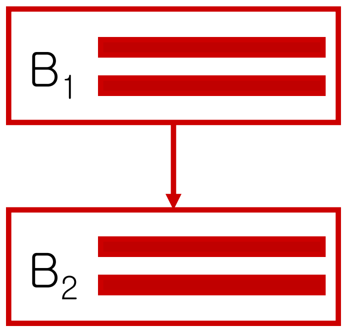
- Local code optimization – code improvement with in a basic block
- Global code optimization – improvements take into account what happens across basic blocks
  - Most are based on data-flow anlaysis
- A compiler optimization must preserve the semantics of the original program
  - Common-subexpression elimination
  - Copy propagation
  - Dead-code elimination
  - Constant folding
  - …

Data-Flow Analysis

- Derives information about the flow of data along program execution paths
- Analyzes the behavior of a program by considering all the possible sequence of program points (paths) through a flow graph that the program execution can take

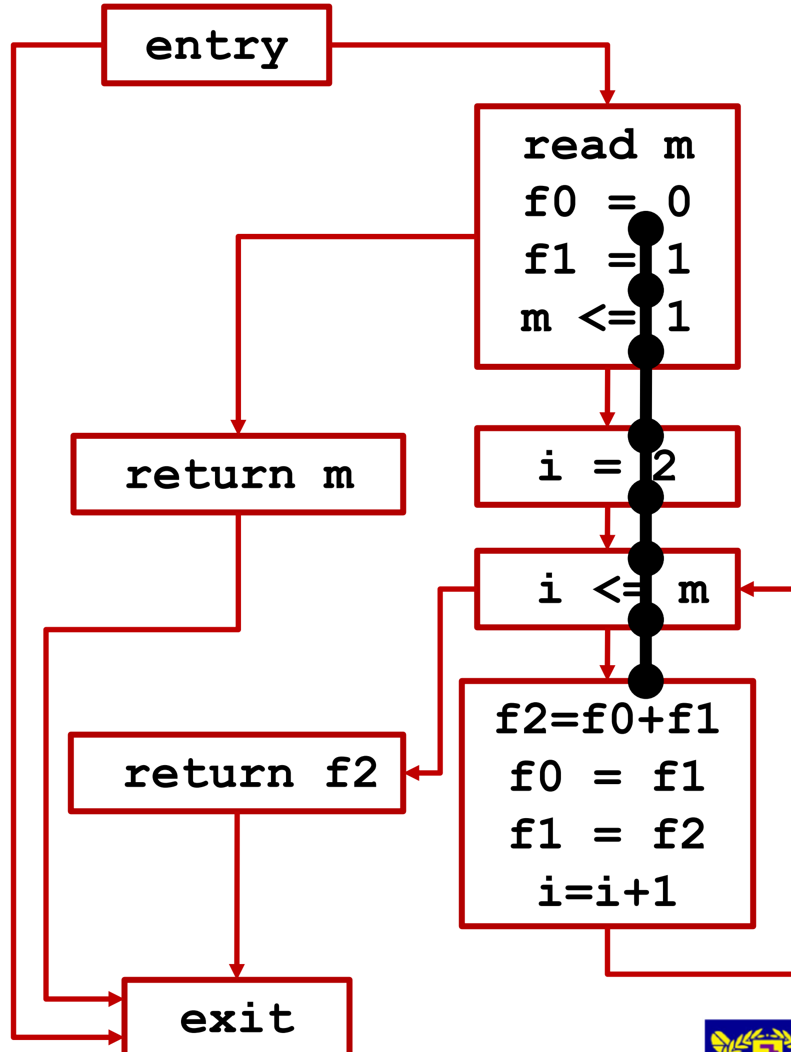
Data-Flow Analysis (contd.)

- Within one basic block, the program point after a statement is the same as the program point before the next statement
- If there is an edge form block B_1 to block B_2, then the program point after the last statement of B_1 may be followed immediately by the program point before the first statement of B_2



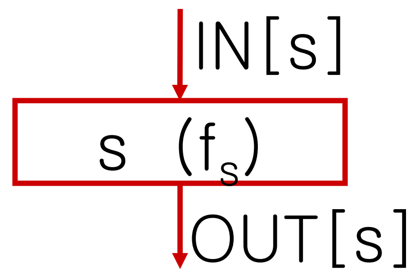
Execution Paths

- From point p_1 to p_n
  - A sequence of points p_1, p_2, ⋯, p_n such that for each i = 1, 2, ⋯, n – 1, either
  - p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  - p_i is the end of some block and p_{i+1} is the beginning of a successor block



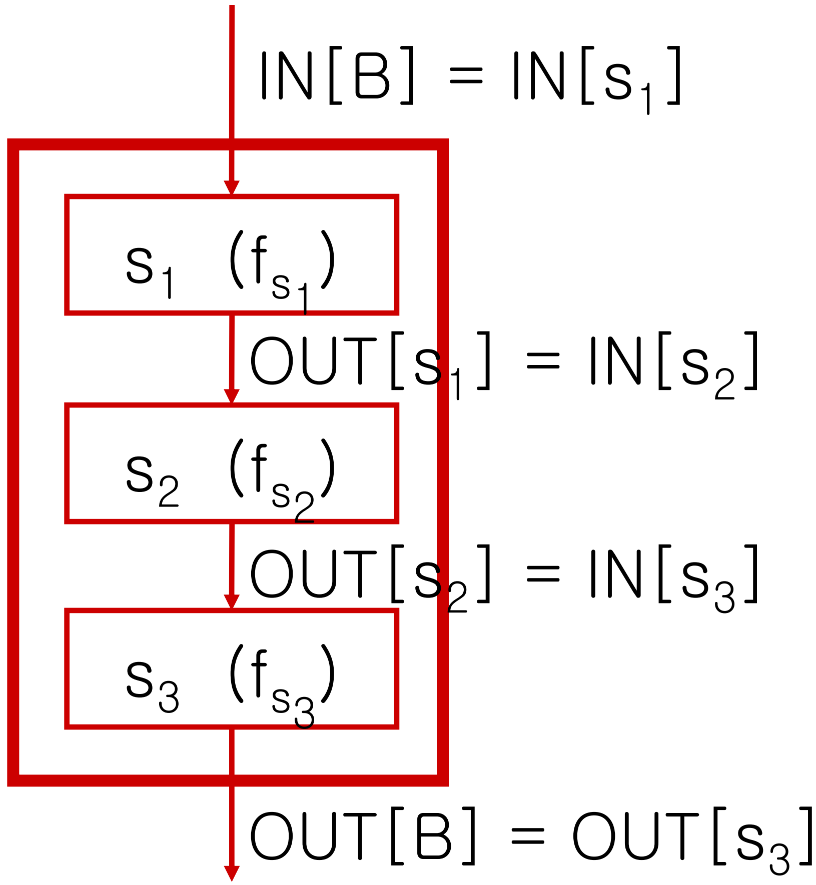
Transfer Functions

- Data-flow values before and after each statement s
  - IN[s] and OUT[s]
- Data-flow problem – to find a solution to a set of constraints on the IN[s]'s and OUT[s]'s for all statements s
- Transfer function – the relationship between the data-flow values before and after the statement
  - Forward-flow problem – OUT[s] = f_s(IN[s])
  - Backward-flow problem – IN[s] = f_s(OUT[s])



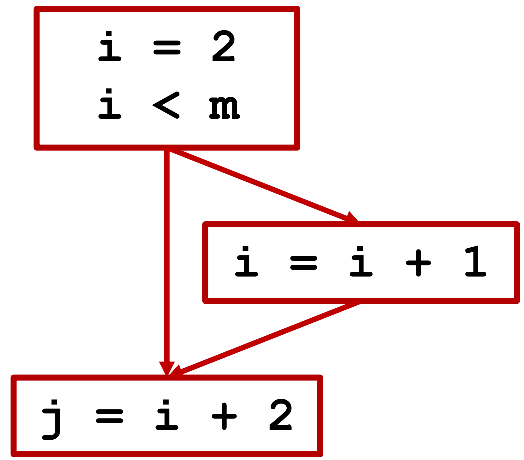
Extension to Basic Blocks

- Suppose block B consists of statements s_1, ⋯, and s_n in that order
  - $IN[B] = IN[s_1]$
  - $OUT[B] = OUT[s_n]$
- Forward-flow problem
  - $f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1}$
  - $OUT[B] = f_B(IN[B])$
- Backward-flow problem
  - $f_B = f_{s_1} \circ \cdots \circ f_{s_{n-1}} \circ f_{s_n}$
  - $IN[B] = f_B(OUT[B])$



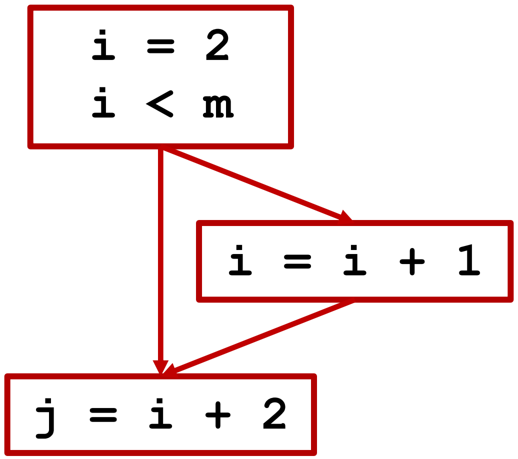
Data Flow Problem #1: Reaching Definitions

- Which definitions of a variable may reach each use of the variable in a procedure?
- A definition of a variable x is a statement that assigns, or may assign, a value to x
  - A statement defines a variable x if it may assign x a value
    - conservative



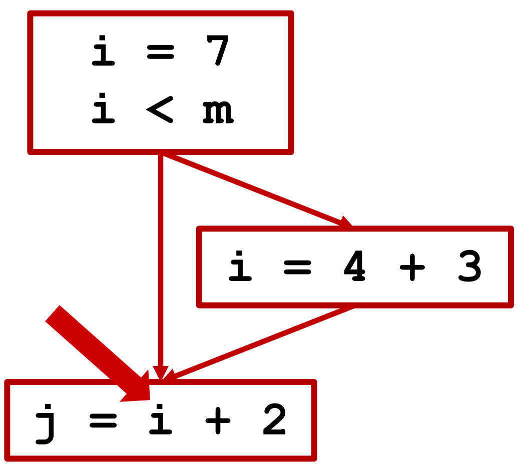
Reaching Definitions (contd.)

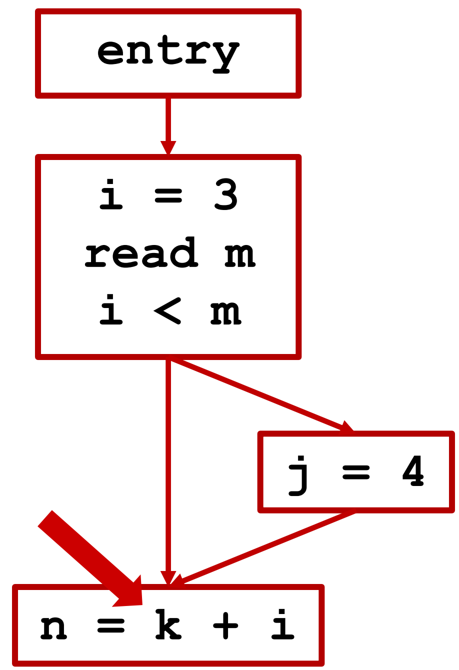
- A definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not killed along that path
- A definition of a variable x is killed if there is any other definition of x anywhere along the path



Usages of Reaching Definitions

- Is x a constant at point p?
- Is x undefined at point p?





Effects of a Statement

d: u = v + w

- Generates a definition d of variable u
- Kills all the other definitions in the program that define variable u
- Transfer function

$$f_d(x) = gen_d \cup (x - kill_d)$$

  - gen_d – the set of definitions generated by the statement
  - $kill_d$ – the set of all other definitions of u in the program

d0: z = 7
d4: y = 4 + 8

d1: x = a + b
d2: y = x + 3
d3: x = x + 4

$gen_{d1} = \{d1\}$, $kill_{d1} = \{d3\}$
$gen_{d2} = \{d2\}$, $kill_{d2} = \{d4\}$
$gen_{d3} = \{d3\}$, $kill_{d3} = \{d1\}$

Effects of a Basic Block

- Compose effects of statements

$$f_B(x) = f_n(\cdots f_2(f_1(x))\cdots)$$

$$= gen_n \cup ((gen_{n-1} \cup ((\cdots (gen_1 \cup (x - kill_1)) \cdots) - kill_{n-1})) - kill_n)$$

in[B]={d0,d4}

Gen[B]={d2,d3}

Kill[B]={d4}

d1: x=a+b
d2: y=x+3
d3: x=x+4

out[B]={d0,d2,d3}

d0: z=7
d4: y=4+8

$gen_{d1} = \{ d1 \}$, $kill_{d1} = \{d3\}$
$gen_{d2} = \{ d2 \}$, $kill_{d2} = \{d4\}$
$gen_{d3} = \{ d3 \}$, $kill_{d3} = \{d1\}$

Effects of a Basic Block (contd.)

$$f_B(x) = Gen[B] \cup (x - Kill[B])$$
$$Kill[B] = kill_1 \cup kill_2 \cup \cdots \cup kill_n$$
$$Gen[B] = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \cdots (gen_1 - kill_2 - kill_3 - \cdots - kill_n)$$

- Gen[B] – contains all the definitions inside the block that are visible immediately after the block (*downwards exposed*)
- Kill[B] – the union of all the definitions killed by the individual statements

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

Gen[B]={d2,d3}

Kill[B]={d4}

in[B]={d0,d4}

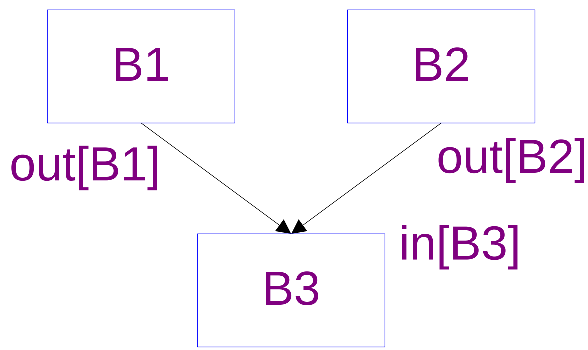
d1: x=a+b
d2: y=x+3
d3: x=x+4

out[B]={d0,d2,d3}

d0: z=7
d4: y=4+8

Effects of Control Flow

- Deal with incoming information from different predecessors of a basic block B
- $in[B] = \cup_{p \in pred[B]} out[P]$



Solving Reaching Definitions Problem

- Data flows forwards
- Create data flow equations and solve them for all basic blocks in the CFG

$$Out[B] = Gen[B] \cup (in[B] - Kill[B])$$
$$in[B] = \cup_{p \in pred[B]} out[P]$$

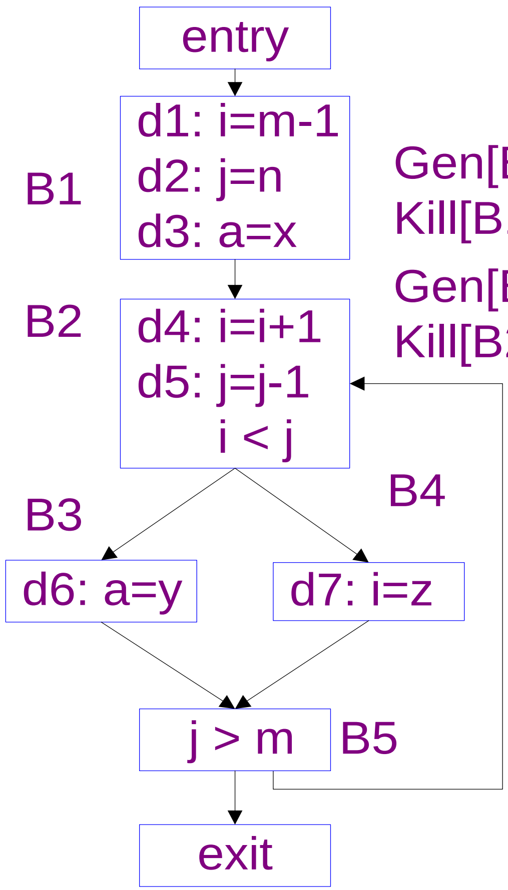
- Use iterative algorithm to solve the equations
- Use bit vectors to represent sets (not necessarily)
  - One bit for each definition
  - $\cap$ becomes bitwise and
  - $\cup$ becomes bitwise or

Iterative Algorithm

- Repeatedly visit all the nodes and update in and out
  - excluding unreachable nodes

```
out[entry] = ∅;
for ( each block B other than entry ) {
    out[B] = ∅; // or out[B] = Gen[B]
}
while (changes to any out occur) {
    for ( each block B other than entry ) {
        in[B] = ∪ pred. P of B out[P];
        out[B] = Gen[B] ∪ (in[B] – Kill[B]);
    }
}
```

Reaching Definitions Example



Gen[B1] = {d1,d2,d3} = 1110000
Kill[B1] = {d4,d5,d6,d7} = 0001111

Gen[B2] = {d4,d5} = 0001100
Kill[B2] = {d1,d2,d7} = 1100001

Gen[B3] = {d6} = 0000010
Kill[B3] = {d3} = 0010000

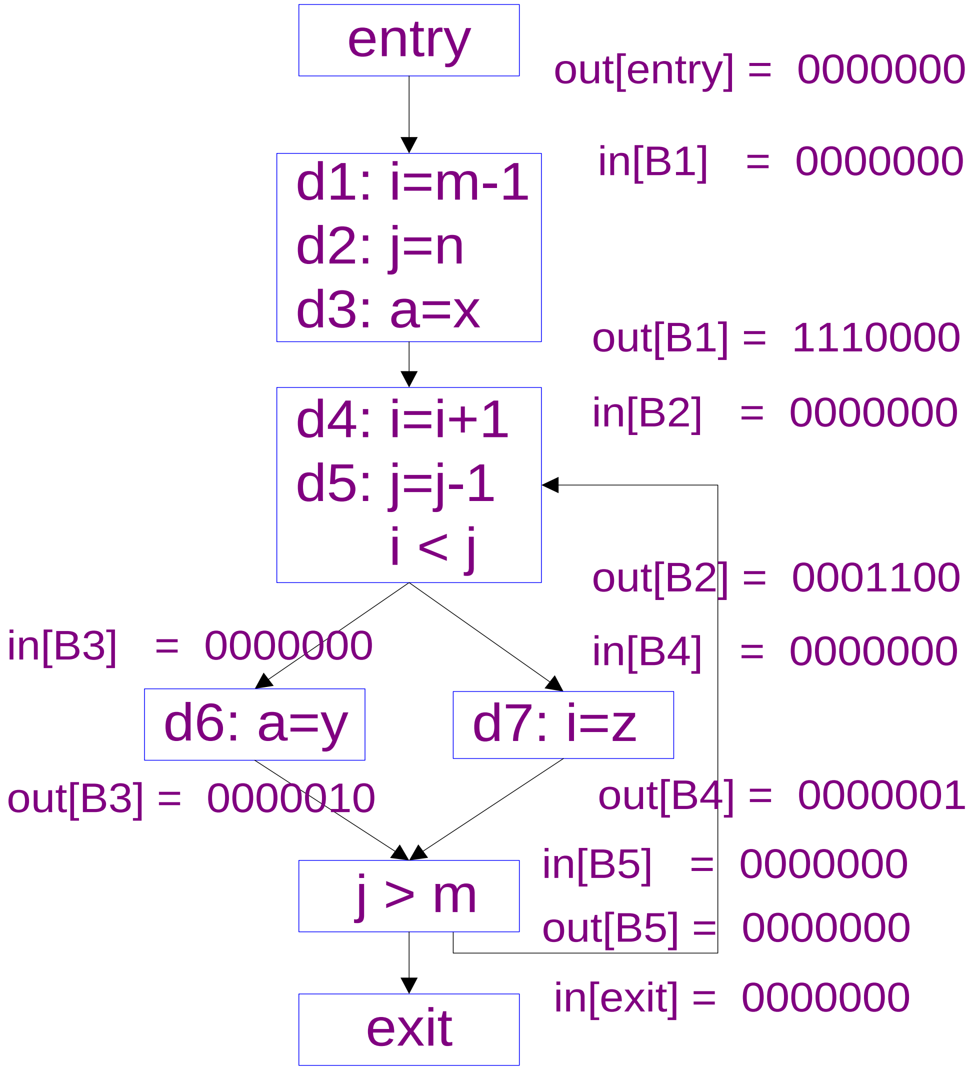
Gen[B4] = {d7} = 0000001
Kill[B4] = {d1,d4} = 1001000

Gen[B5] = { } = 0000000
Kill[B4] = { } = 0000000

out[B1]=Gen[B1] U (in[B1]-Kill[B1])
out[B2]=Gen[B2] U (in[B2]-Kill[B2])
out[B3]=Gen[B3] U (in[B3]-Kill[B3])
out[B4]=Gen[B4] U (in[B4]-Kill[B4])
out[B5]=Gen[B5] U (in[B5]-Kill[B5])

in[B1]=out[entry]
in[B2]=out[B1] U out[B5]
in[B3]=out[B2]
in[B4]=out[B2]
in[B5]=out[B1] U out[B5]

Reaching Definitions Example (contd.)



Gen[B1] = 1110000
Kill[B1] = 0001111

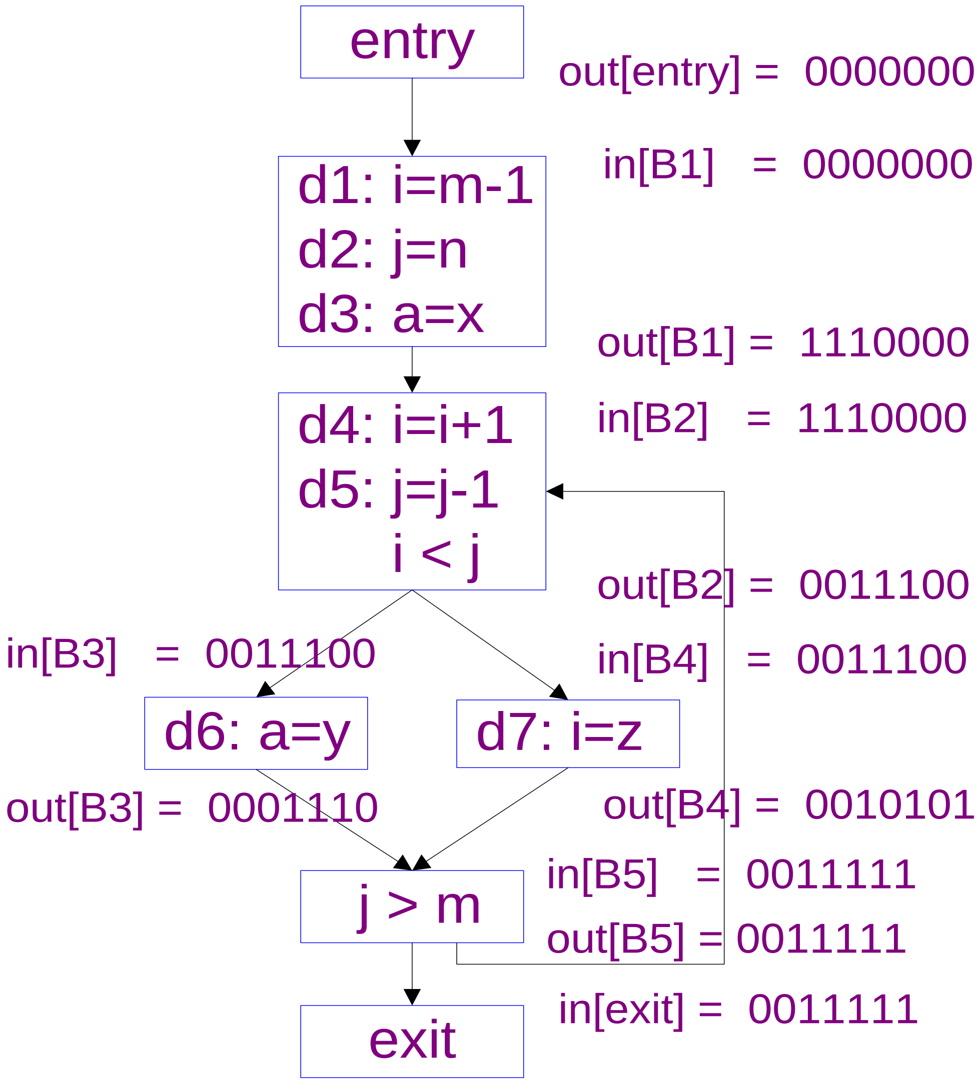
Gen[B2] = 0001100
Kill[B2] = 1100001

Gen[B3] = 0000010
Kill[B3] = 0010000

Gen[B4] = 0000001
Kill[B4] = 1001000

Gen[B5] = 0000000
Kill[B4] = 0000000

Reaching Definitions Example (contd.)



Gen[B1] = 1110000
Kill[B1] = 0001111

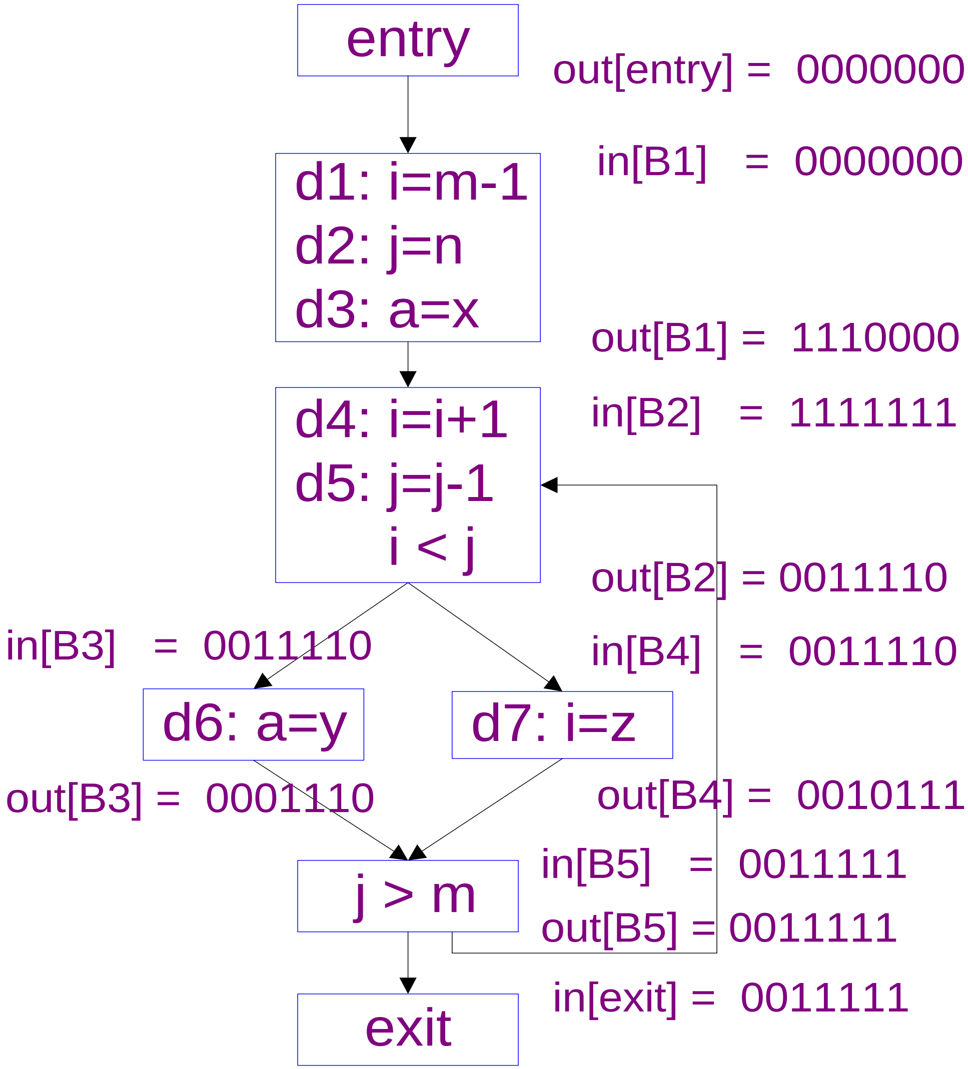
Gen[B2] = 0001100
Kill[B2] = 1100001

Gen[B3] = 0000010
Kill[B3] = 0010000

Gen[B4] = 0000001
Kill[B4] = 1001000

Gen[B5] = 0000000
Kill[B4] = 0000000

Reaching Definitions Example (contd.)



Gen[B1] = 1110000
Kill[B1] = 0001111

Gen[B2] = 0001100
Kill[B2] = 1100001

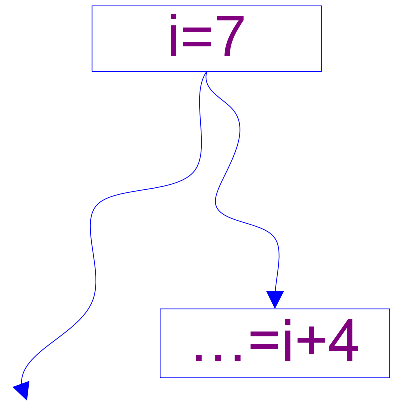
Gen[B3] = 0000010
Kill[B3] = 0010000

Gen[B4] = 0000001
Kill[B4] = 1001000

Gen[B5] = 0000000
Kill[B4] = 0000000

Data Flow Problem #2: Live Variable Analysis

- A variable x is live at a point p if the value of x at p could be used along some path in the flow graph starting at p
- Used in
  - Register allocation
  - Code motion in loops
  - Elimination of useless assignments (dead code elimination)
- Data flows backwards



Transfer Functions of a Basic Block

- Def[B]: the set of variables definitely assigned values in B
- Use[B]: the set of variables whose values may be used in B prior to any definition of the variable
  - Uses not covered by the definitions in B
- in[B] is a function of out[B]

in[B]={a,b,z}

Use[B]={a,b}

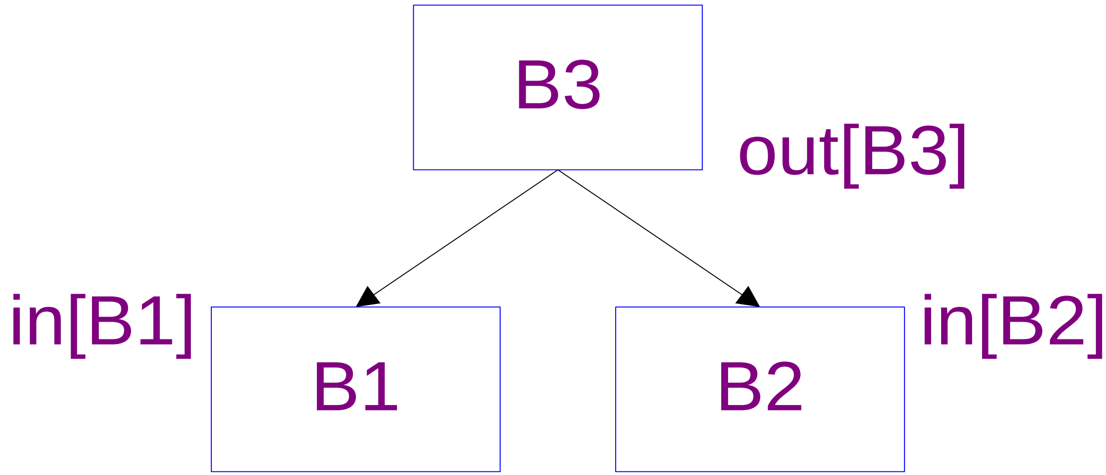
Def[B]={x,y}

```
x=a+b
y=x+3
x=x+4
```

out[B]={x,z}

$$in[B] = f_B(out[B]) = Use[B] \cup (out[B] - Def[B])$$

Effects of Control Flow



out[B] = in[P1] U in[P2] U … U in[Pn]

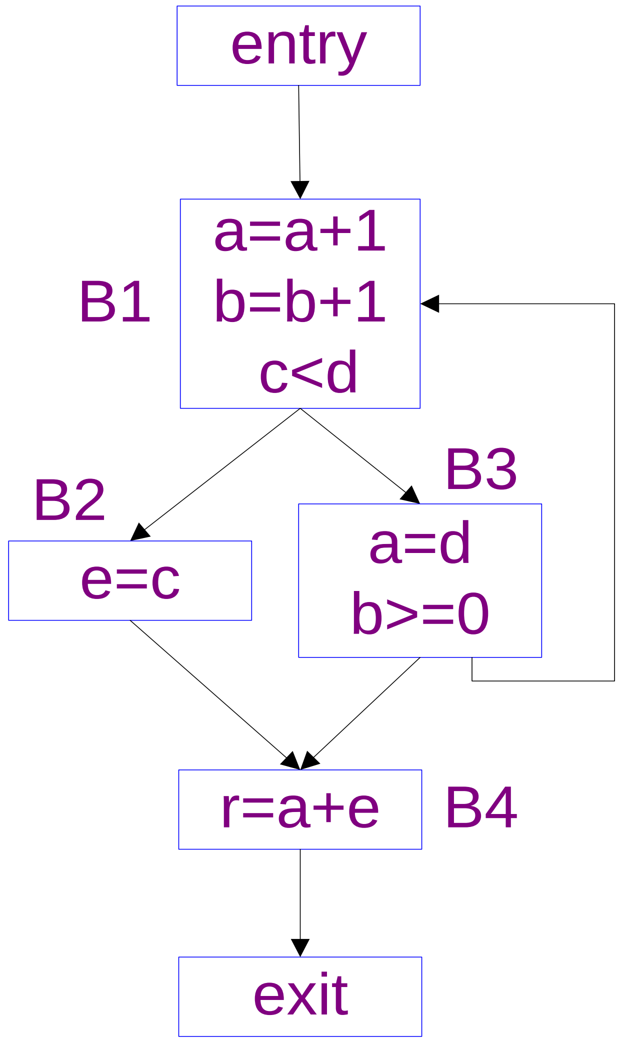
P1,P2, …, Pn are successors of B

Iterative Solution for Live Variable Analysis

- Repeatedly visit all the nodes and update in and out

in[exit] = ∅
for each block B other than exit **do**
  in [B] = ∅ // or in[B] = Use[B]
enddo
while changes to any in occur **do**
  for each block B other than exit **do**
    out[B] = $\cup_{\text{succ. P of B}}$ in[P]
    in[B] = Use[B] ∪ (out[B] – Def[B])
  enddo
enddo

Live Variable Analysis Example



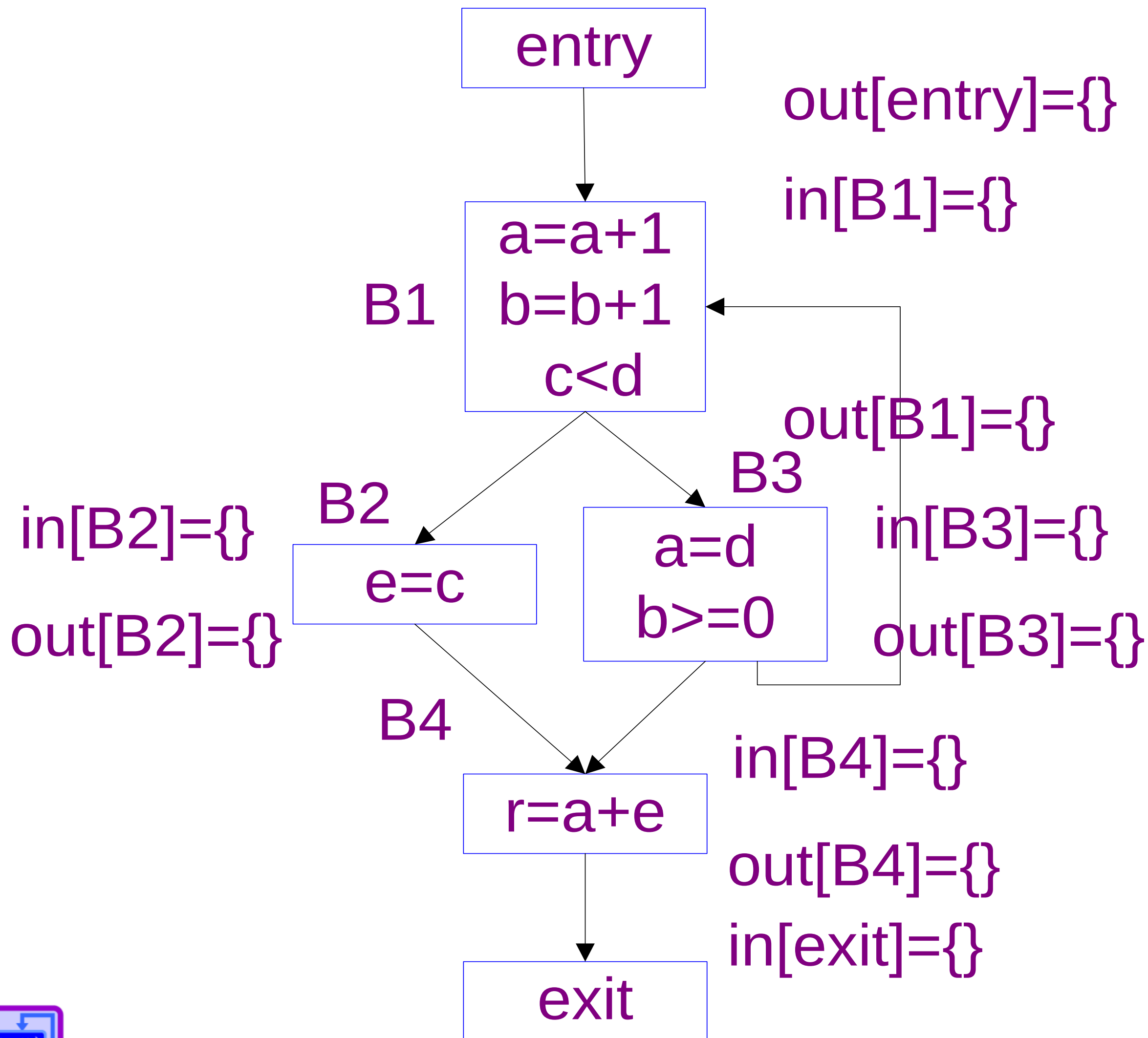
Def[B1] = {a,b}
Use[B1] = {a,b,c,d}

Def[B2] = {e}
Use[B2] = {c}

Def[B3] = {a}
Use[B3] = {b,d}

Def[B4] = {r}
Use[B4] = {a,e}

Live Variable Analysis Example (contd.)



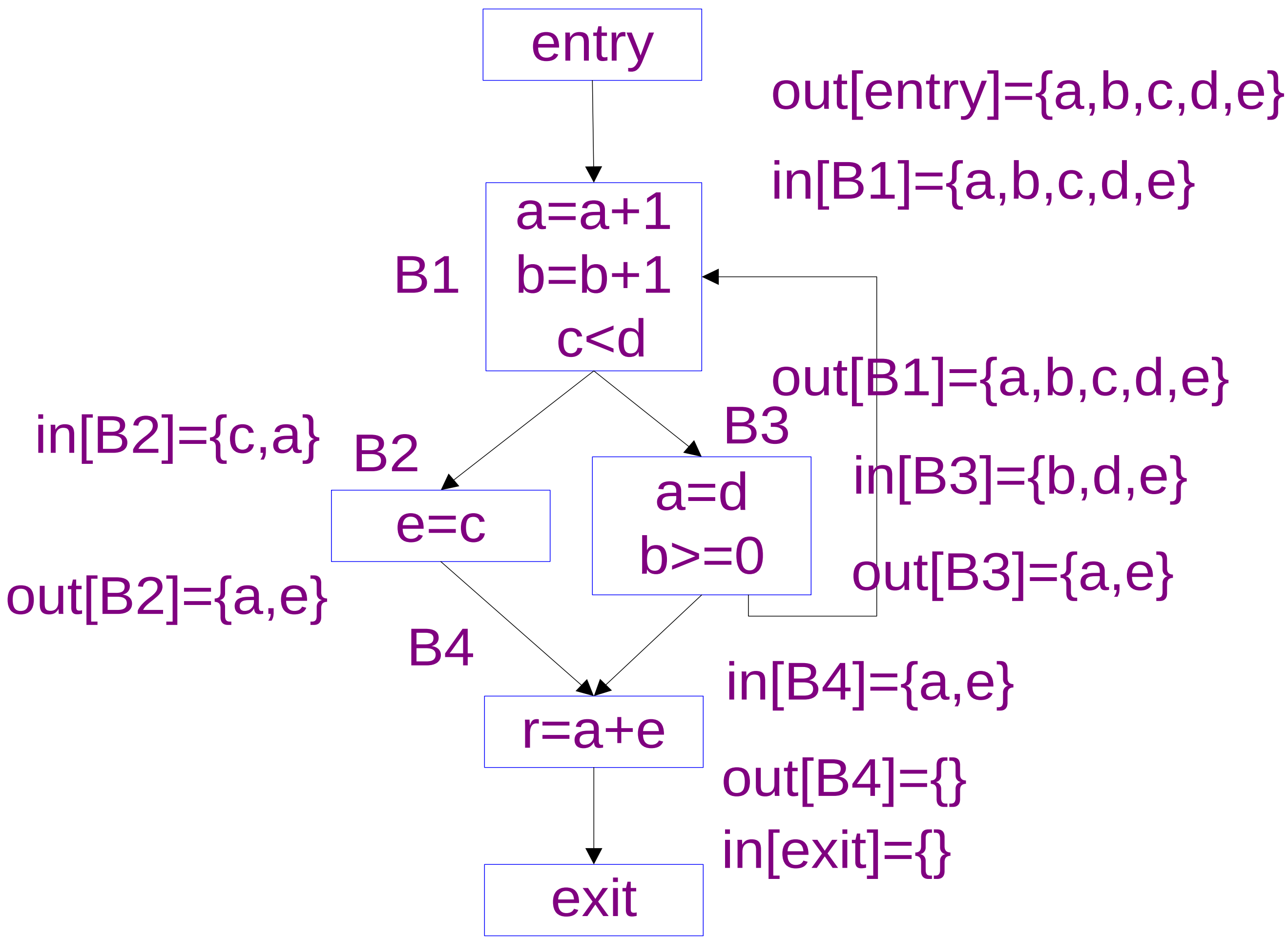
Def[B1] = {a,b}
Use[B1] = {a,b,c,d}

Def[B2] = {e}
Use[B2] = {c}

Def[B3] = {a}
Use[B3] = {b,d}

Def[B4] = {r}
Use[B4] = {a,e}

Live Variable Analysis Example (contd.)



Def[B1] = {a,b}
Use[B1] = {a,b,c,d}

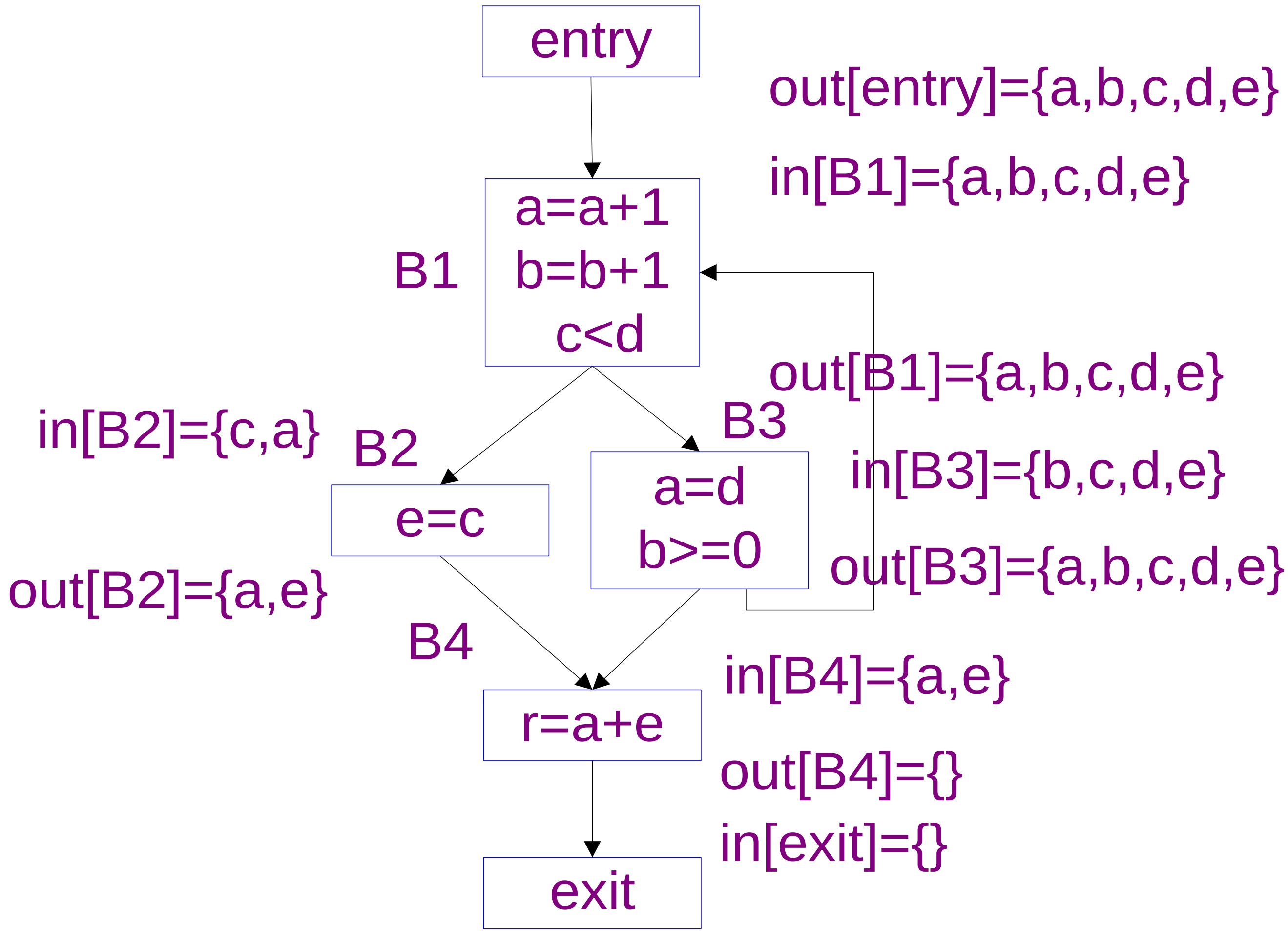
Def[B2] = {e}
Use[B2] = {c}

Def[B3] = {a}
Use[B3] = {b,d}

Def[B4] = {r}
Use[B4] = {a,e}

In the order of B4, B3, B2, B1

Live Variable Analysis Example (contd.)



Def[B1] = {a,b}
Use[B1] = {a,b,c,d}

Def[B2] = {e}
Use[B2] = {c}

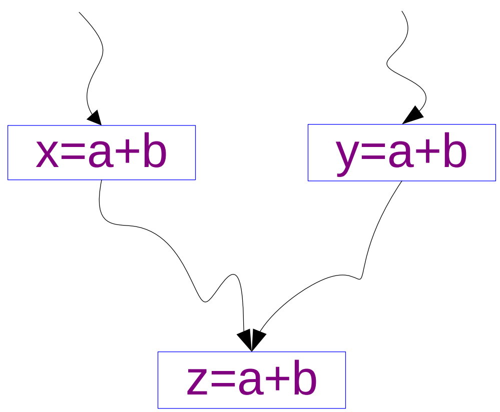
Def[B3] = {a}
Use[B3] = {b,d}

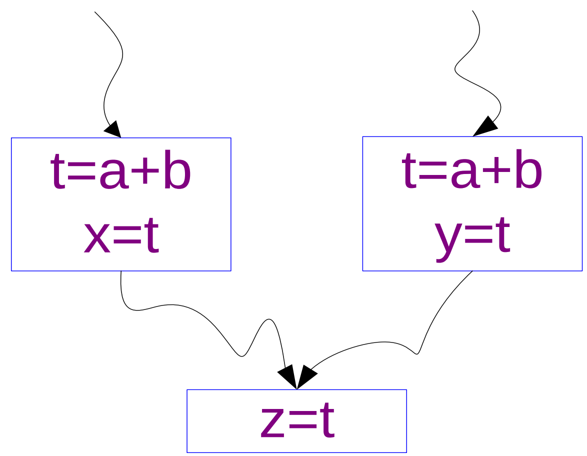
Def[B4] = {r}
Use[B4] = {a,e}

In the order of B4, B3, B2, B1

Data Flow Problem #3: Available Expressions

- An expression x op y is available at a point p if every path from the entry node to p evaluates x op y, and after the last such evaluation prior to reaching p, there are no subsequent assignments to x or y
- Used in common subexpression elimination





Kill[B] and Gen[B]

- Kill[B] – A block kills expression x op y if it assigns (or may assign) x or y and does not subsequently recompute x op y
- Gen[B] – A block generates expression x op y if it definitely evaluates x op y and does not subsequently define x or y

$$\text{in}[B] = \cap_{\text{pred. } P \text{ of } B}\ \text{out}[P]$$

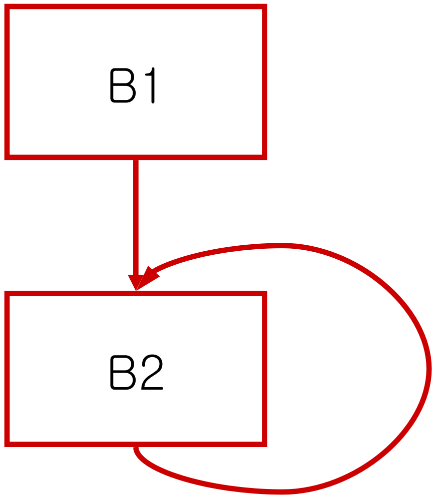
$$\text{out}[B]=\text{Gen}[B] \cup (\text{in}[B]-\text{Kill}[B])$$

Iterative Solution for Available Expressions

- Repeatedly visit all the nodes and update in and out

out[entry] = ∅
for each block B other than entry **do**
  out [B] = U
enddo
while changes to any out occur **do**
  for each block B other than entry **do**
    in[B] = $\cap_{\text{pred. P of B}}$ out[P]
    out[B] = Gen[B] ∪ (in[B] – Kill[B])
  enddo
enddo

$Out[B] = \varnothing$ is Too Restrictive



$in[B2] = out[B1] \cap out\ [B2]$
$out[B2] = Gen[B2] \cup (in[B2] - Kill[B2])$

$in^{j+1}[B2] = out[B1] \cap out^{j}[B2]$
$out^{j+1}[B2] = Gen[B2] \cup (in^{j+1}[B2] - Kill[B2])$

$out^{0}[B2] = \varnothing$
$in^{1}[B2] = out[B1] \cap out^{0}[B2] = \varnothing$

$out^{0}[B2] = U$
$in^{1}[B2] = out[B1] \cap out^{0}[B2] = out[B1]$

Data-Flow Analysis Summary

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer Function	Gen[B] (x – Kill[B])	Use[B] (x – Def[B])	Gen[B] (x – Kill[B])
Boundary	out[entry] = $\varnothing$	int[exit] = $\varnothing$	out[entry] = $\varnothing$
Meet($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	out[B] = f_B(in[B]) in[B] = $\wedge$ $_{\text{pred. P of B}}$ out[P]	in[B] = f_B(out[B]) out[B] = $\wedge$ $_{\text{succ. P of B}}$ in[P]	out[B] = f_B(in[B]) in[B] = $\wedge$ $_{\text{pred. P of B}}$ out[P]
Initialize	out[B] = $\varnothing$	in[B] = $\varnothing$	out[B] = U

Foundations of Data-Flow Analysis

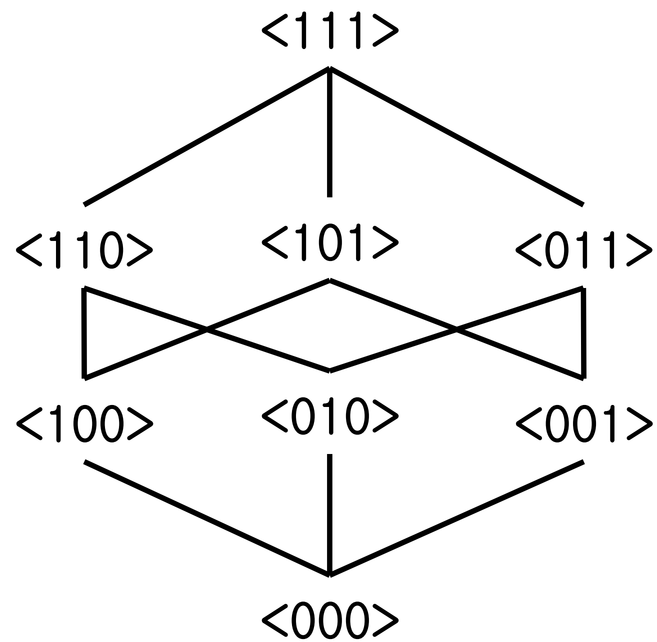
- Under what circumstances is the iterative algorithm used in data-flow analysis correct?
- How precise is the solution obtained by the iterative algorithm?
- Will the iterative algorithm converge?
- What is the meaning of the solution to the equations?

Data-Flow Analysis Framework (D, V, $\wedge$,F)

- A direction of the data flow D
  - Forward or backward
- A semilattice, which includes a domain of values V and a meet operator $\wedge$
- A family F of transfer functions from V to V
  - Must include functions suitable for the boundary conditions, which are constant transfer functions for the entry and exit

Partial Order

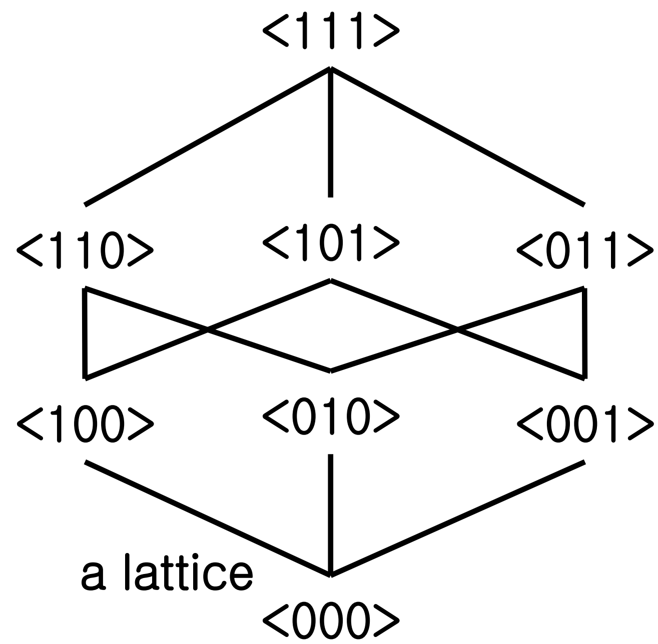
- A binary relation ≤ over a set V is a partial order is if for all x, y, and z in V,
  - Reflexive: $x \leq x$
  - Antisymmetric: $x \leq y$ and $y \leq x \rightarrow x = y$
  - Transitive: $x \leq y$ and $y \leq z \rightarrow x \leq z$
- A set V with a partial order ≤ is called a partially ordered set (poset) (V, ≤)
- $x < y$ iff $(x \leq y)$ and $x \neq y$

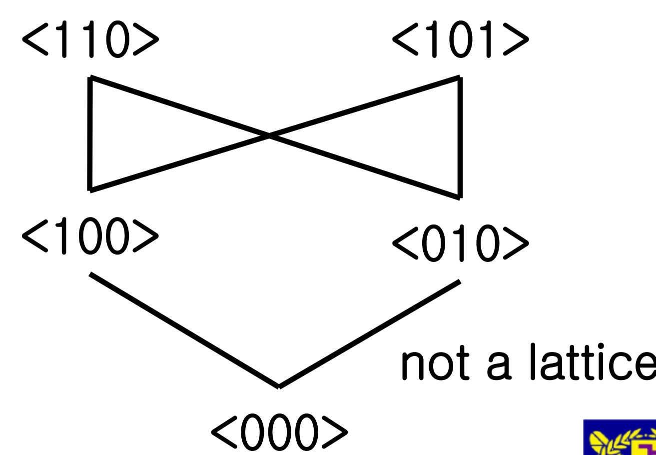


bit vector = poset $(2^{\{1,2,3\}}, \subseteq)$

Semilattices

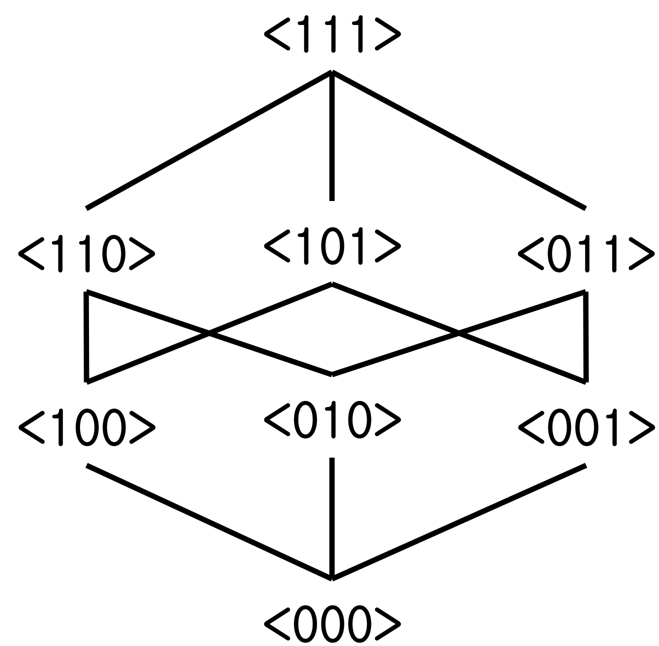
- A semmilattice (V, ∧) is a set V and a binary meet operator ∧ such that for all x, y, and z in V
  - idempotent: $x \wedge x = x$
  - commutative: $x \wedge y = y \wedge x$
  - associative: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
  - has a top element, denoted T, such that for all x in V, $T \wedge x = x$
  - optionally, has a bottom element, denoted ⊥, such that for all x in V, $\perp \wedge x = \perp$





Partial Order for a Semilattice (V, ∧)

- Define a partial order ≤ for a semilattice (V, ∧)
  - For all x and y in V, x ≤ y iff x ∧ y = x
  - ≤ is reflexive, antisymmetric, and transitive



Greatest Lower Bounds

- Suppose (V, $\wedge$) is a semilattice
- A greatest lower bound (glb) of x and y in V is an element g such that,
  - $g \leq x$
  - $g \leq y$, and
  - If z is any element such that $z \leq x$ and $z \leq y$, then $z \leq g$
- There is at most one such element g if it exist

Glb and Meet Operation

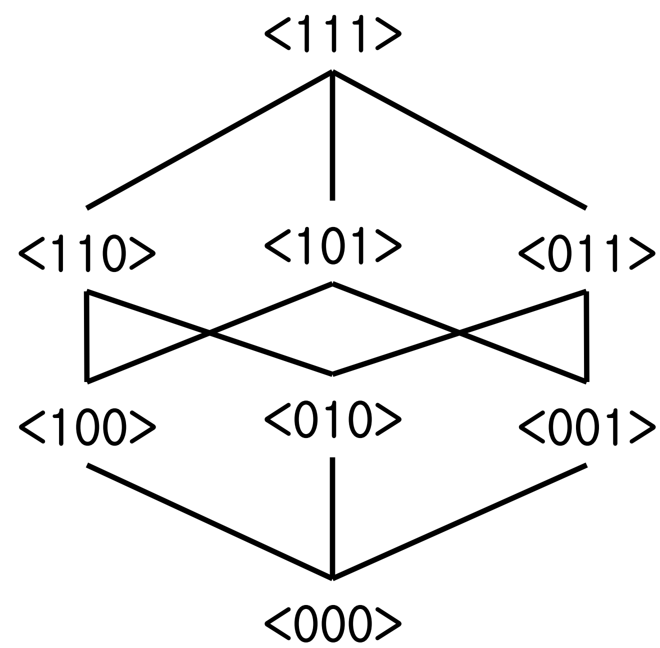
- The meet of x and y is their only glb
  - Let $g = x \wedge y$
  - $g \leq x$ because $(x \wedge y) \wedge x = x \wedge y$
  - $g \wedge x = ((x \wedge y) \wedge x) = (x \wedge (y \wedge x)) = (x \wedge (x \wedge y)) = ((x \wedge x) \wedge y) = (x \wedge y) = g$
  - Similarly, $g \leq y$
  - Suppose z is any element such that $z \leq x$ and $z \leq y$. $z \leq g$ and therefore z cannot be a glb of x and y unless $z = g$
    - $z \wedge g = (z \wedge (x \wedge y)) = ((z \wedge x) \wedge y)$
    - Since $z \leq y$, we know $z \wedge y = z$, and therefore $z \wedge g = z$
    - Proven $z \leq g$ and conclude $g = x \wedge y$ is the only glb of x and y

Product of Two Semilattices (A, $\wedge_A$) and (B, $\wedge_B$)

- domain – $D = A \times B$
- meet – if $(a,b) \in D$ and $(a',b') \in D$, $(a, b) \wedge (a', b') = (a \wedge_A a', b \wedge_B b')$
- partial order – $(a, b) \le (a', b')$ iff $a \le_A a'$ and $b \le_B b'$
- When $a \wedge_A a' = a$ and $b \wedge_B b' = b$, then $(a \wedge_A a', b \wedge_B b') = (a, b)$
  - $a \wedge_A a' = a \leftrightarrow a \le_A a'$
  - $b \wedge_B b' = b \leftrightarrow b \le_B b'$
- The product is an associative operation
  - Meet – $(a_1, a_2, \cdots, a_k) \wedge (b_1, b_2, \cdots, b_k) = (a_1 \wedge_1 b_1, a_2 \wedge_2 b_2, \cdots, a_k \wedge_k b_k)$
  - Partial order – $(a_1, a_2, \cdots, a_k) \le (b_1, b_2, \cdots, b_k)$ iff $a_i \le_i b_i$, for all i

Chains

- An ascending chain in a poset $(V, \leq)$ is a sequence where $x_1 < x_2 < \cdots < x_n$ and $x_i \in V$ for all i
- The height of a semilattice is the largest number of < relations in any ascending chain
  - Reaching definitions semilattice for a program with n definition is n
- A lattice consisting of a finite set of values has a finite height
- A lattice consisting of an infinite set of values may have a finite height



Bit vector
$\subseteq$

Transfer Functions

- The family of transfer functions
  - F: V → V
  - F has an identity function I, such that I(x) = x for all x in V
  - F is closed under composition
    - For any two functions f and g in F, the function h defined by h(x) = g(f(x)) is in F
- Reaching definitions
  - There is an identity function where Gen[B] and Kill[B] are both the empty set
  - $f_{B1}(x) = Gen[B1] \cup (x - Kill[B1])$
  - $f_{B2}(x) = Gen[B2] \cup (x - Kill[B2])$
  - $f_{B2}(f_{B1}(x)) = Gen[B2] \cup ((Gen[B1] \cup (x - Kill[B1])) - Kill[B2])$
    $= (Gen[B2] \cup (Gen[B1] - Kill[B2])) \cup (x - (Kill[B1] \cup Kill[B2]))$

Monotone Frameworks

- A data-flow framework $(D, F, V, \wedge)$ is monotone if for all x and y in V and f in F, $x \leq y$ implies $f(x) \leq f(y)$
- Equivalently,
  - for all x and y in V and f in F, $f(x \wedge y) \leq f(x) \wedge f(y)$
  - Proof
    - Assume $x \leq y$ implies $f(x) \leq f(y)$
      - $x \wedge y \leq x$ and $x \wedge y \leq y$
      - $f(x \wedge y) \leq f(x)$ and $f(x \wedge y) \leq f(y)$
      - Since $f(x) \wedge f(y)$ is glb of $f(x)$ and $f(y)$, $f(x \wedge y) \leq f(x) \wedge f(y)$
    - Assume $f(x \wedge y) \leq f(x) \wedge f(y)$ and suppose $x \leq y$
      - $f(x) \leq f(x) \wedge f(y)$ since $x \wedge y = x$
      - Since $f(x) \wedge f(y)$ is glb of $f(x)$ and $f(y)$, we know $f(x) \wedge f(y) \leq f(y)$
      - $f(x) \leq f(x) \wedge f(y) \leq f(y)$

Distributive Frameworks

- A data-flow framework (D, F, V, $\wedge$) is distributive if for all x and y in V and f in F, $f(x \wedge y) = f(x) \wedge f(y)$
- Distributivity implies monotonicity, but not vice versa
  - If a = b, then $a \wedge b = a$, so $a \leq b$
- Reaching Definitions
  - $Gen[B] \cup ((x \cup y) - Kill[B])$
    $= (Gen[B] \cup (x - Kill[B])) \cup (Gen[B] \cup (y - Kill[B]))$
  - $(x \cup y) - Kill[B] = (x - Kill[B]) \cup (y - Kill[B])$

The Iterative Algorithm for General Frameworks

- A data-flow graph, with entry and exit nodes
- A direction of the data-flow D
- A set of values V
- A meet operator $\wedge$
- A set of transfer functions F for basic blocks
- A constant value v_{entry} and v_{exit} in V representing the boundary condition for forward and backward frameworks, respectively

The Iterative Algorithm for General Frameworks (contd.)

```
out[entry] = v_entry ;
for ( each block B other than entry ) out[B] = T;
while (changes to any out occur)
    for (each block B other than entry) {
        in[B] = ∧ pred. P of B out[P];
        out[B] = f_B(in[B]);
    }
```

```
in[exit] = v_exit ;
for ( each block B other than entry ) in[B] = T;
while (changes to any in occur)
    for (each block B other than exit) {
        out[B] = ∧ succ. P of B in[P];
        in[B] = f_B(out[B]);
    }
```

Properties of the Iterative Algorithm

1. If the iterative algorithm converges, the result is a solution to the data-flow equations
   - If the equations are not satisfied by the time the while-loop ends, then there will be at least one change to an OUT (in the forward case) or IN (in the backward case)
   - Loop once more

Properties of the Iterative Algorithm (contd.)

2. If the framework is monotone, then the solution found is the maximum fixedpoint (MFP) of the data-flow equations
   - A maximum fixedpoint is a solution with the property that in any other solution, the values of IN[B] and OUT[B] are ≤ the corresponding values of the MFP

Properties of the Iterative Algorithm (contd.)

- Proof (forward case)
  - Basis
    - IN[B] and OUT[B] for all blocks B $\neq$ entry are initialized with T
    - After the first iteration the value of IN[B] and OUT[B] is not greater than the initialized value
  - Induction
    - Assume that after the k^{th} iteration, the values are all not greater than those after $(k-1)^{th}$ iteration
    - $IN[B] = \wedge_{\text{pred. P of B}}\ out[P]$
      - $OUT^k[P] \leq OUT^{k-1}[P] \rightarrow IN^{k+1}[B] \leq IN^k[B]$
    - $OUT[B] = f_B(IN[B])$
      - $IN^{k+1}[B] \leq IN^k[B] \rightarrow OUT^{k+1}[P] \leq OUT^k[P]$ (by monotonicity)

Properties of the Iterative Algorithm (contd.)

- The values taken by IN[B] and OUT[B] for any B can only decrease as the algorithm iterates
- Every change observed for values of IN[B] and OUT[B] is necessary to satisfy the equations
- If the iterative algorithm terminates, the result must have values that are at least as great as the corresponding values in any other solutions
  - The meet operators return the glb of their inputs
  - The transfer functions return the only solution that is consistent with the block itself and its given input

Properties of the Iterative Algorithm (contd.)

3. If the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge
   - The values of each IN[B] and OUT[B] decrease with each change, and the algorithm stops if at some round nothing changes
   - The algorithm converges after a number of rounds no greater than (the height) × (the number of basic blocks)

The Ideal Solution

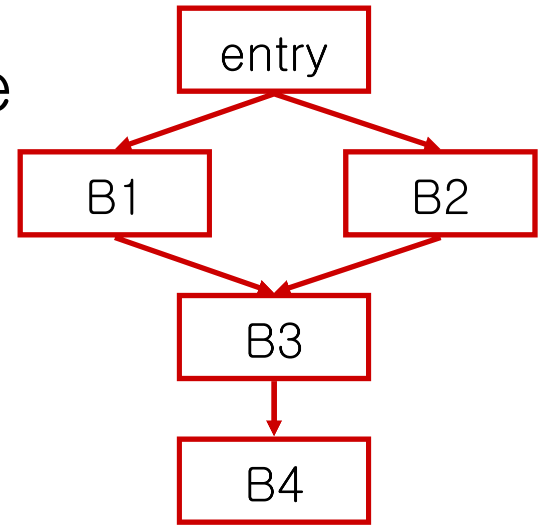
- Let $P = entry \rightarrow B_1 \rightarrow B_2 \rightarrow \cdots \rightarrow B_k$ be a path in G
  - $f_P(x) = f_k(f_{k-1}(\cdots(f_1(x))\cdots))$
- $IDEAL[B] = \wedge_{\text{P a possible execution path from entry to B}} f_P[v_{entry}]$
  - Any answer that is greater than IDEAL is incorrect
  - Any value smaller than or equal to the ideal is conservative, i.e., safe

The Meet-Over-Paths Solution

- $MOP[B] = \wedge_{P \text{ a path from entry to B}} f_P[v_{entry}]$
  - A super set of all the paths that are possibly executed
  - $MOP[B] \leq IDEAL[B]$
- Computing MOP is undecidable
  - There is no algorithm that can compute MOP for an arbitrary instance of monotone framework
- There is no efficient way to tell exactly which paths are real and which are not
  - Accept the MOP solution as the closest feasible solution

MOP vs. MFP

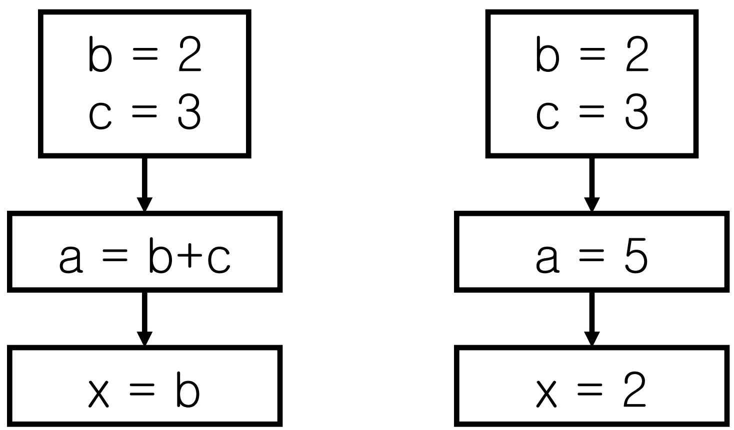
- If all the functions are distributive,
  - MFP solution = the MOP solution
- If the transfer functions are all monotone but not necessarily distributive, the iterative algorithm produces the MFP solution but not necessarily the MOP solution



$$MOP[B4] = ((f_{B3} \circ f_{B1}) \wedge (f_{B3} \circ f_{B2}))(V_{entry})$$
$$MFP[B4] = f_{B3}((f_{B1}(V_{entry}) \wedge f_{B2}(V_{entry})))$$

Constant Propagation (CONST)

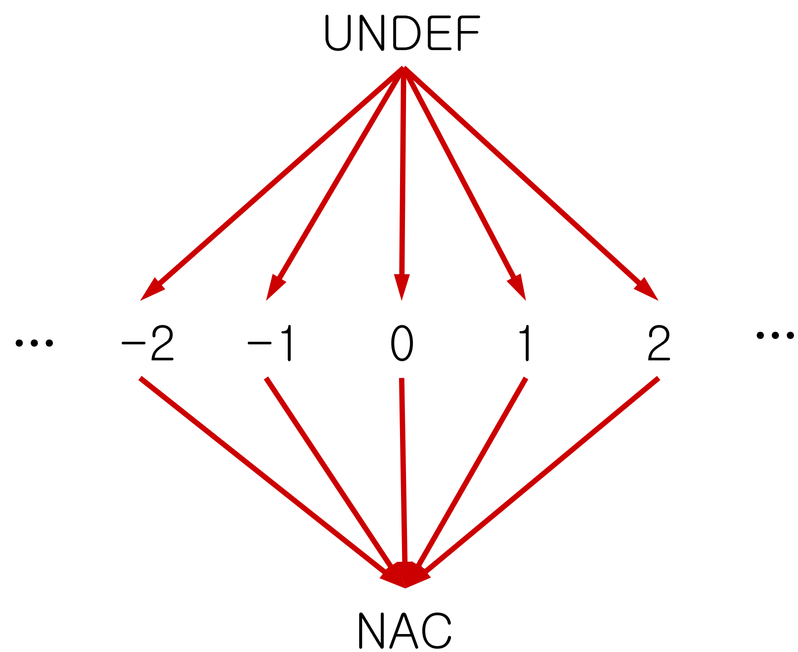
- For each program point, whether or not a variable has a constant value whenever execution reaches that point
  - For constant folding or constant propagation
    - Replaces expressions that evaluate to the same constant every time they are executed, by that constant



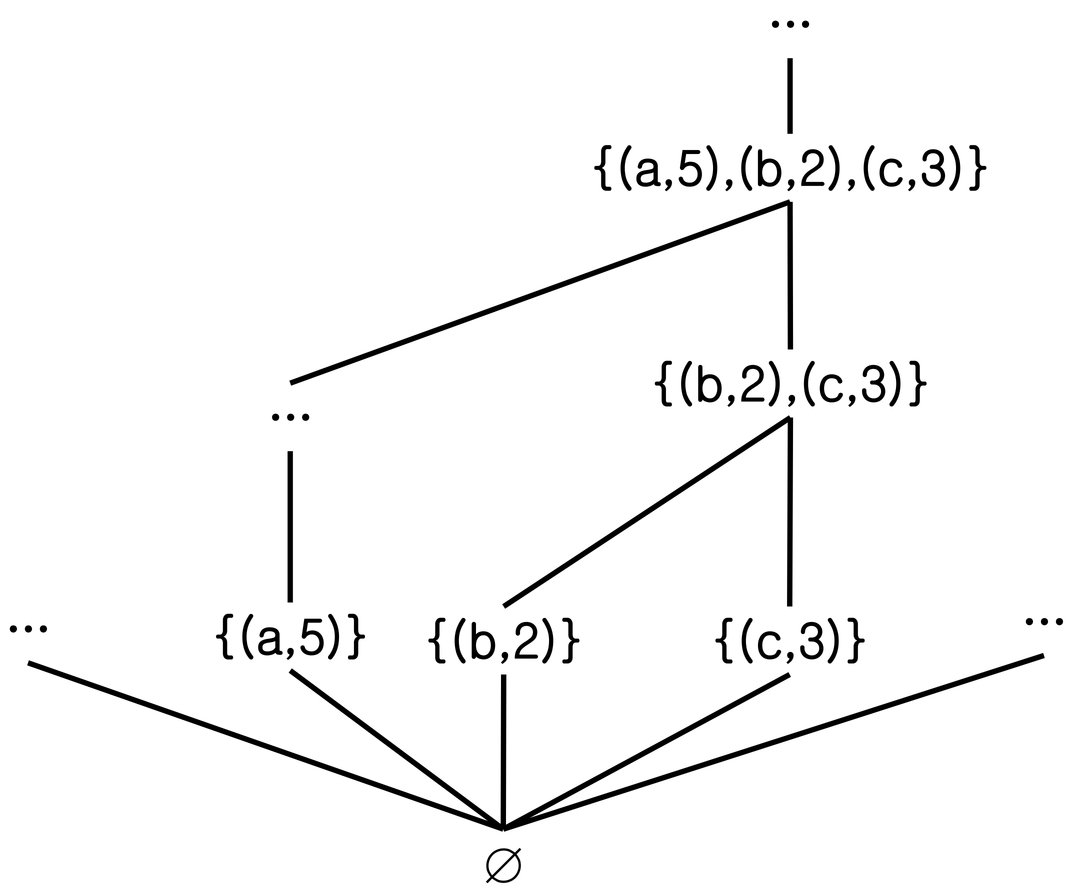
The Lattice for CONST

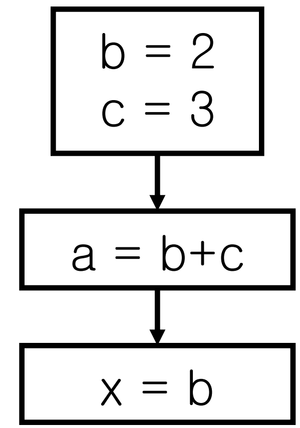
- $(L, \wedge)$ is a product lattice and $L \subseteq 2^{V \times (C \cup \{UNDEF,\ NAC\})}$, where V is the infinite set of variables and C is the set of constant values
  - NAC – not a constant
  - UNDEF – undefined
  - $\wedge = \cap$
  - L is the set of functions from V to C
    - Sets of (variable, constant value) pairs
    - Bit-vector is inappropriate
  - $f \in L$ is the information about variables that we may assume at certain points of a flow graph
    - $(v, c) \in f$
      - variable v has a constant value c

The Lattice for a Single Variable



The Lattice for CONST

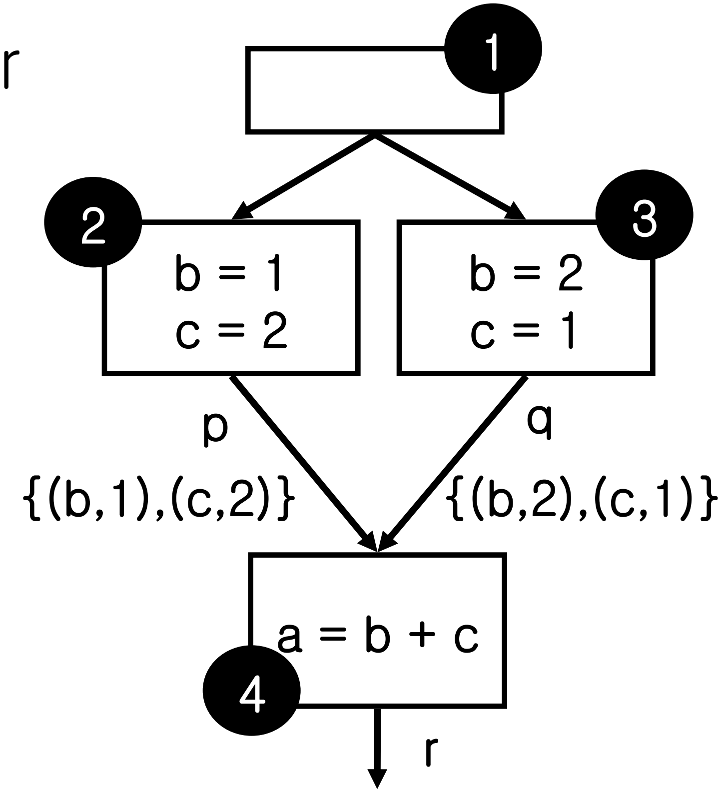




Transfer Functions for CONST

- Let f_s be the transfer function of statement s, and let m and m' represent data-flow values such that m' = f_s(m)
  - If s is not an assignment statement, then f_s is the identity function
  - If s is an assignment to variable x, then m'(v) = m(v), for all variables v ≠ x
    - If RHS of s is a constant c, then m'(x) = c
    - If RHS is of the form y • z, then
      - m'(x) = m(y) • m(z) if m(y) and m(z) are constant values
      - m'(x) = NAC if either m(y) or m(z) is NAC
      - m'(x) = UNDEF otherwise
    - If RHS is any other expression (e.g. a function call or assignments through a pointer), then m'(x) = NAC

CONST is not Distributive



- Data flow formulation for the point r
  - p = {(b,1), (c,2)}
  - q = {(b,2), (c,1)}
  - $p \cap q = \varnothing$
  - Therefore, $f_4(p \cap q) = f_4(\varnothing) = \varnothing$
- MOP formulation for the point r
  - $f_4(p)=f_4(\{(b,1),(c,2)\})=$ $\{(a,3),(b,1),(c,2)\}$
  - $f_4(q)=f_4(\{(b,2),(c,1)\})=$ $\{(a,3),(b,2),(c,1)\}$
  - $f_4(p) \cap f_4(q) = \{(a,3)\}$

$f_4(p \cap q) \neq f_4(p) \cap f_4(q)$

CONST is a Monotone Framework (contd.)

- Need to show for all x, y∈*L* and for all functions of the form $f_{A=B\bullet C}$ or $f_{A=r}$,
  - $f_{A=B\bullet C}(x \cap y) \subseteq f_{A=B\bullet C}(x) \cap f_{A=B\bullet C}(y)$, and
  - $f_{A=r}(x \cap y) \subseteq f_{A=r}(x) \cap f_{A=r}(y)$

$f_{A=B\bullet C}(x \cap y) \subseteq f_{A=B\bullet C}(x) \cap f_{A=B\bullet C}(y)$

- For all $X \in V - \{A\}$, if $(X,r) \in f_{A=B\bullet C}(x \cap y)$ then $(X,r) \in x$ and $(X,r) \in y$. Thus, $(X,r) \in f_{A=B\bullet C}(x)$ and $(X,r) \in f_{A=B\bullet C}(y)$
- If $(A,r) \in f_{A=B\bullet C}(x \cap y)$, then $\{(B,r_1),(C,r_2)\}$ is a subset of both x and y, for some r_1 and r_2 such that $r = r_1 \bullet r_2$. This implies $(A,r) \in f_{A=B\bullet C}(x)$ and $(A,r) \in f_{A=B\bullet C}(y)$
- If A is undefined in $f_{A=B\bullet C}(x \cap y)$
  - One of (B, b) and (C, c) is not in $x \cap y$. One of x and y can not have both of (B, b) and (C, c). This means that A is undefined one of $f_{A=B\bullet C}(x)$ and $f_{A=B\bullet C}(y)$. Thus, A is undefined in $f_{A=B\bullet C}(x) \cap f_{A=B\bullet C}(y)$

$f_{A=r}(x \cap y) \subseteq f_{A=r}(x) \cap f_{A=r}(y)$

- For all $X \in V - \{A\}$, if $(X,r) \in f_{A=r}(x \cap y)$ then $(X,r) \in x$ and $(X,r) \in y$. Thus, $(X,r) \in f_{A=r}(x)$ and $(X,r) \in f_{A=r}(y)$
- $(A,r) \in f_{A=r}(x \cap y)$, $(A,r) \in f_{A=r}(x)$, and $(A,r) \in f_{A=r}(y)$ are always true

Speed of Convergence of Iterative Data-Flow Algorithms

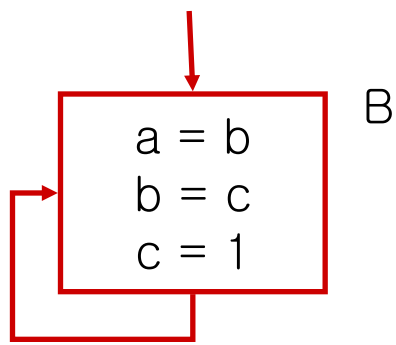
- The maximum number of iterations
  - The height of the lattice × the number of nodes
- Whether all events of significance at a node will be propagated to that node along some acyclic path?
- If all useful information propagates along acyclic paths, tailor the order in which we visit nodes in the iterative algorithm
  - Relatively few passes
  - Depth-first order or the reverse of depth-first order

Speed of Convergence of Iterative Data-Flow Algorithms (contd.)

- Reaching definitions
  - If a definition d is in IN[B], then there is some acyclic path from the block containing d to B s.t. d is in the IN's and OUT's all along that path
- Available expressions
  - If an expression x + y is not available at the entrance to B, then there is some acyclic path that demonstrates that either the path is from the entry node and includes no statement that kills or generates x + y, or the path is from a block that kills x + y and along the path there is no subsequent generation of x + y
- Live variables
  - If x is live on exit from B, then there is an acyclic path from B to a use of x, along which there are no definitions of x
- Paths with cycles add nothing for these analyses
  - Remove cycles and find a shorter path

Speed of Convergence of Iterative Data-Flow Algorithms (contd.)

- Constant propagation
  - The first time B is visited, c is found to be constant 1, but both a and b are undefined
  - The second time, b and c are found to be constant 1
  - The third time, a is found to be constant 1



Spanning Trees

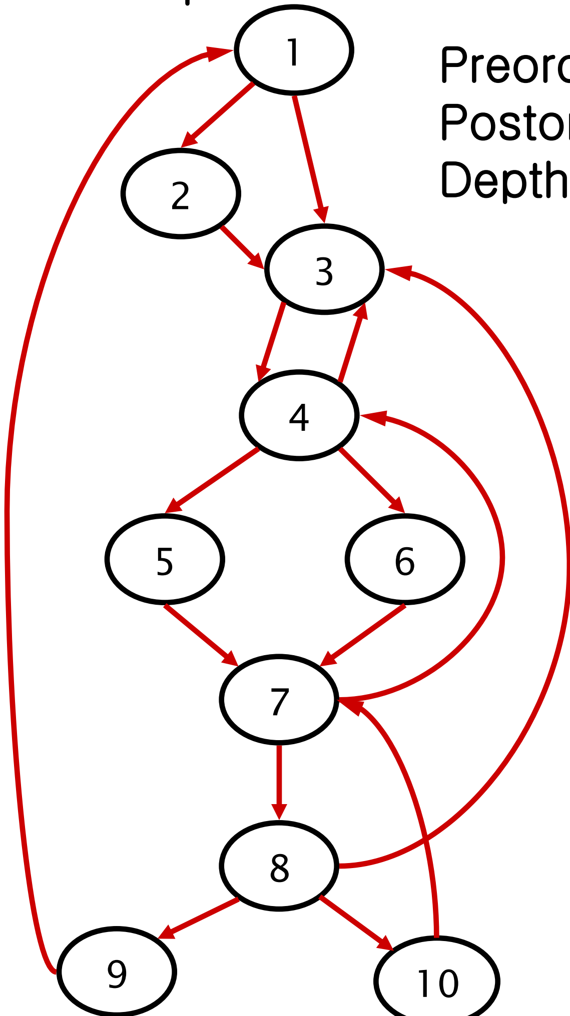
- A spanning tree of a graph is just a subgraph that contains all the nodes and is a tree
- A graph may have many spanning trees

Graph Search

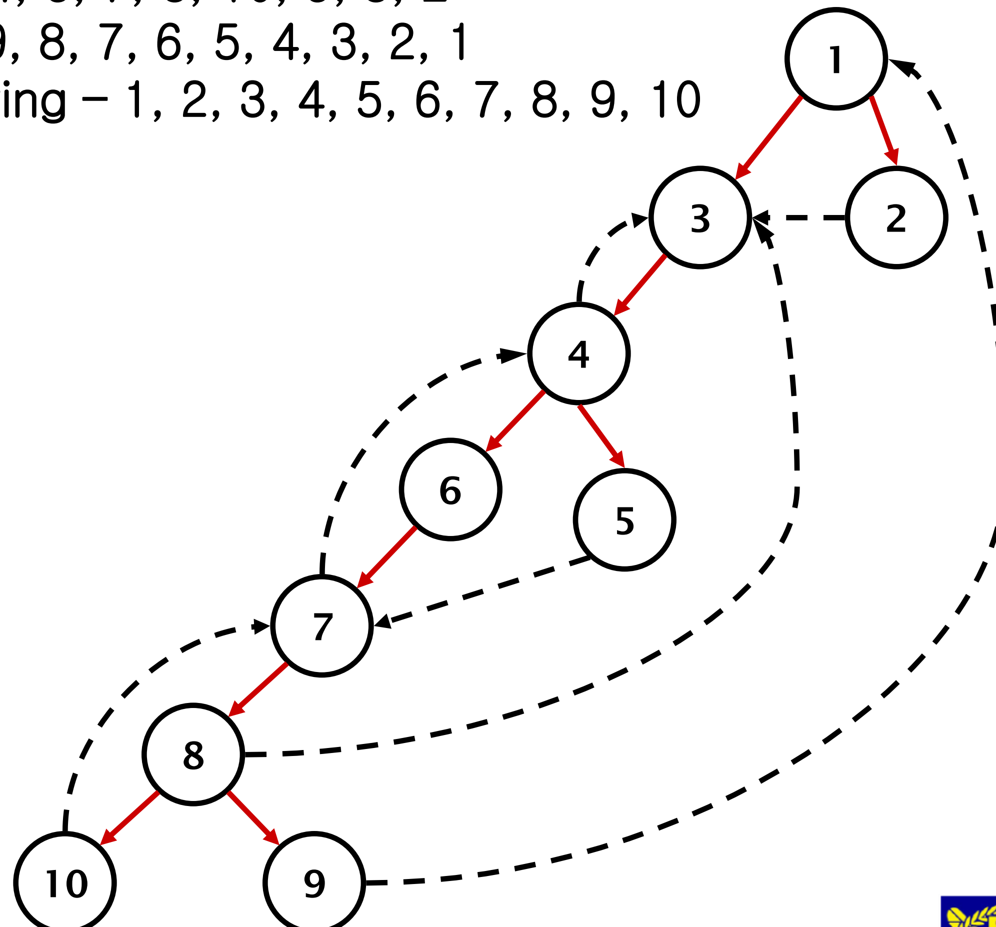
- Depth-first search
  - Visits all the nodes in the graph once, by starting at the entry node and visiting the nodes as far away from the entry node as quickly possible
- Depth-first spanning tree
  - The route of the search in a depth-first search
- Tree traversal
  - Preorder – visits a node before visiting any of its children, which it then visits recursively in left-to-right order
  - Postorder – visits a node's children, recursively in left-to-right order, before visiting the node itself
- Depth-first ordering
  - The reverse of a postorder traversal
  - Visit a node, traverse its rightmost child, the child to its left, and so on

Depth-First Spanning Tree

Depth first traversal – 1 , 3, 4, 6, 7, 8,10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1



Preorder – 1, 3, 4, 6, 7, 8, 10, 9, 5, 2
Postorder – 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
Depth-first ordering – 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

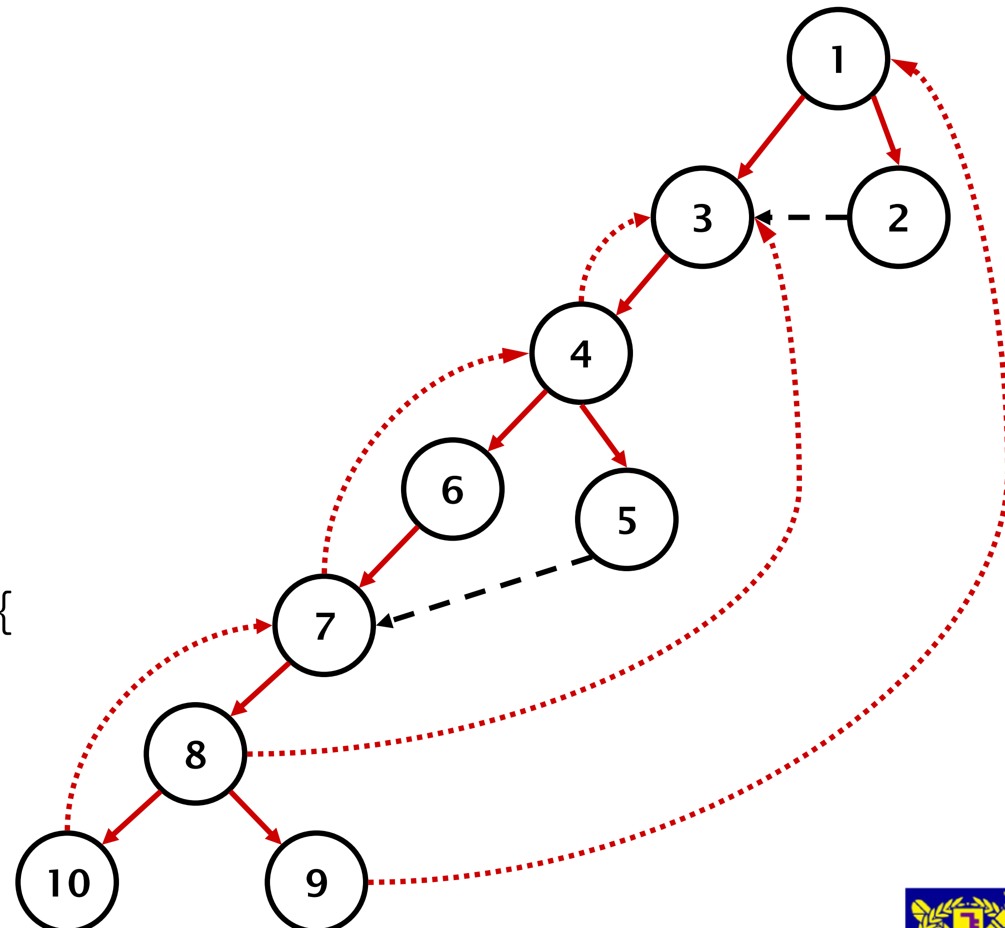


Depth-First Ordering

- m → n is a retreating edge iff dfn[m] ≥ dfn[n]
  - Go from a node m to an ancestor of m in the tree (possibly to m itself)
  - 4→3, 7→4, 10→7, 9→1, 8→3

```
void search(n) {
    mark n “visited”;
    for ( each successor s of n ) {
       if ( s is “unvisited” ) {
          add edge n→s to T;
          search(s);
       }
    }
    dfn[n] = c
    c = c – 1;
}
```

```
main() {
      T = {};
      for ( each node n of G ) {
          mark n “unvisited”;
      }
      c = the # of nodes in G;
      search(entry);
}
```



Speed of Convergence of Iterative Data-Flow Algorithms (contd.)

- A definition d propagates in a path, 3 → 5 →19 → 35 → 16 → 23 → 45 → 4 → 10 → 17 (the depth first numbers of basic blocks)
  - The first round, OUT[3] → IN[5] → OUT[5] → IN[19] → OUT[19] → IN[35] → OUT[35]
    - IN[16] has been already computed
  - The second round, OUT[35] → IN[16] → OUT[16] → IN[23] → OUT[23] → IN[45] → OUT[45]
    - IN[4] has been already computed
  - The third round, OUT[45] → IN[4] → OUT[4] → IN[10] → OUT[10] → IN[17] → OUT[17]
- After three passes d reaches block 17

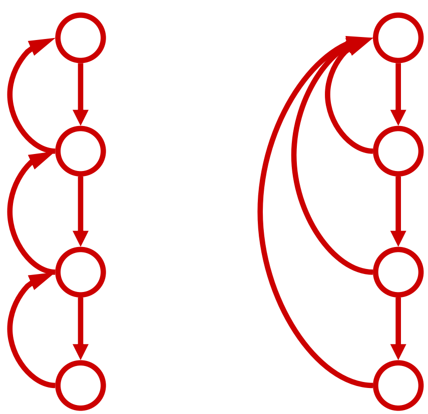
```
out[entry] = v_entry ;
for ( each block B other than entry ) out[B] = T;
while (changes to any out occur)
    for (each block B other than entry, in depth-first order) {
        in[B] = ∧ pred. P of B out[P];
        out[B] = f_B(in[B]);
    }
```

Speed of Convergence of Iterative Data-Flow Algorithms (contd.)

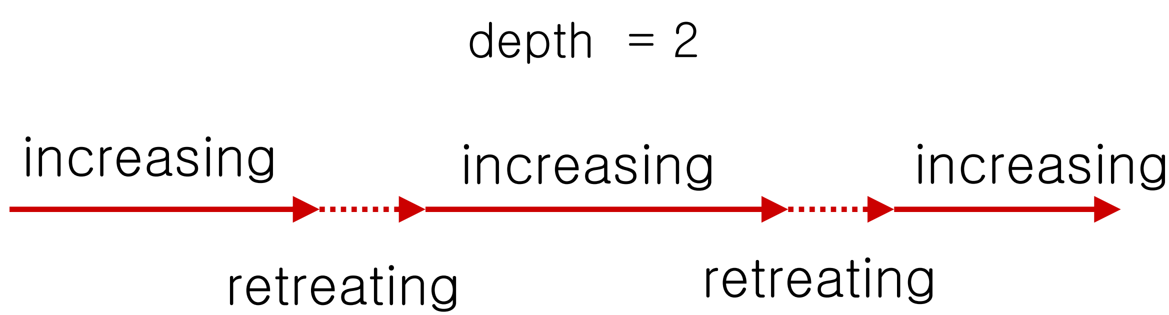
- For reaching definitions,
  - To propagate any reaching definition along any acyclic path is no more than *one* greater than the number of retreating edges
  - One more pass to detect that a fixed point is reached
  - *Two plus the depth of the flow graph*
- Typical flow graphs
  - Average depth around 2.75
  - D. E. Knuth, "An empirical study of FORTRAN programs," Software – Practice and Experience 1:2 (1971), pp. 105-133
- Backward-flow problems – use the reverse of the depth-first order
- The depth+2 bound works for any monotone framework
  - As long as information only needs to propagate along acyclic paths

Depth of a Flow Graph

- The depth of a flow graph is the largest number of retreating edges along any acyclic path in the flow graph
- Normal control-flow constructs produce reducible flow graphs with the number of back edges (i.e., retreating edges) at most the nesting depth of loops
  - Nesting depth tends to be small



depth = 3 depth = 1



References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. “Compilers” (second edition), Addison Wesley, 2006
- Matthew S. Hecht. “Flow Analysis of Computer Programs”, Elsevier Science Ltd., 1977