Dataflow Analysis
Lattices & Solvers

15-411/15-611 Compiler Design

Seth Copen Goldstein

February 26, 2026

Calculating Reaching Definitions

A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.

- Build up RD stmt by stmt
- Stmt s, “d: v <- x op y”, generates d
- Stmt s, “d: v <- x op y”, kills all other defs(v)

Or,

- Gen[s] = { d }
- Kill[s] = defs(v) – { d }

© 2019-20 Goldstein

Gen and kill for each stmt

Stmt		Gen	kill
1:	`n <- 10`	1	
2:	`older <- 0`	2	9
3:	`old <- 1`	3	10
4:	`result <- 0`	4	8
5:	`if n <= 1 goto 14`		
6:	`i <- 2`	6	11
7:	`if i > n goto 13`		
8:	`result <- old + older`	8	4
9:	`older <- old`	9	2
10:	`old <- result`	10	3
11:	`i <- i + 1`	11	6
12:	`JUMP 7`		
13:	`return result`		
14:	`return n`		

we determine the defs that reach a node by using:

- control flow information
- gen and kill info

© 2019-20 Goldstein

Computing in[n] and out[n]

- In[n]: the set of defs that reach the beginning of node n
- Out[n]: the set of defs that reach the end of node n

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

$$out[n] = gen[n] \cup (in[n] - kill[n])$$

- Initialize in[n]=out[n]={} for all n
- Solve iteratively

© 2019-20 Goldstein

Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

Code	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9		
3: old <- 1	3	10		
4: result <- 0	4	8		
5: if n <= 1 goto 14				
6: i <- 2	6	11		
7: if i > n goto 13				
8: result <- old + older	8	4		
9: older <- old	9	2		
10: old <- result	10	3		
11: i <- i + 1	11	6		
12: JUMP 7				
13: return result				
14: return n				

© 2019-20 Goldstein

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
`1: n <- 10`	1			1
`2: older <- 0`	2	9	1	1,2
`3: old <- 1`	3	10	1,2	1,2,3
`4: result <- 0`	4	8	1-3	1-4
`5: if n <= 1 goto 14`			1-4	1-4
`6: i <- 2`	6	11	1-4	1-4,6
`7: if i > n goto 13`			1-4,6	1-4,6
`8: result <- old + older`	8	4	1-4,6	1-3,6,8
`9: older <- old`	9	2	1-3,6,8	1,3,6,8,9
`10: old <- result`	10	3	1,3,6,8,9	1,6,8-10
`11: i <- i + 1`	11	6	1,6,8-10	1,8-11
`12: JUMP 7`			1,8-11	1,8-11
`13: return result`			1-4,6	1-4,6
`14: return n`			1-4	1-4

© 2019-20 Goldstein

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6	1-4,6
8: result <- old + older	8	4	1-4,6	1-3,6,8
9: older <- old	9	2	1-3,6,8	1,3,6,8,9
10: old <- result	10	3	1,3,6,8,9	1,6,8-10
11: i <- i + 1	11	6	1,6,8-10	1,8-11
12: JUMP 7			1,8-11	1,8-11
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

© 2019-20 Goldstein

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: n <- 10	1			1
2: older <- 0	2	9	1	1,2
3: old <- 1	3	10	1,2	1,2,3
4: result <- 0	4	8	1-3	1-4
5: if n <= 1 goto 14			1-4	1-4
6: i <- 2	6	11	1-4	1-4,6
7: if i > n goto 13			1-4,6	1-4,6
8: result <- old + older	8	4	1-4,6	1-3,6,8
9: older <- old	9	2	1-3,6,8	1,3,6,8,9
10: old <- result	10	3	1,3,6,8,9	1,6,8-10
11: i <- i + 1	11	6	1,6,8-10	1,8-11
12: JUMP 7			1,8-11	1,8-11
13: return result			1-4,6	1-4,6
14: return n			1-4	1-4

© 2019-20 Goldstein

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

$$in[n] = \bigcup_{p \in pred[n]} out[p] \quad out[n] = gen[n] \bigcup (in[n] - kill[n])$$

	Gen	kill	in	out
1: `n <- 10`	1			1
2: `older <- 0`	2	9	1	1,2
3: `old <- 1`	3	10	1,2	1,2,3
4: `result <- 0`	4	8	1-3	1-4
5: `if n <= 1 goto 14`			1-4	1-4
6: `i <- 2`	6	11	1-4	1-4,6
7: `if i > n goto 13`			1-4,6,8-11	1-4,6,8-11
8: `result <- old + older`	8	4	1-4,6,8-11	1-3,6,8-11
9: `older <- old`	9	2	1-3,6,8-11	1,3,6,8-11
10: `old <- result`	10	3	1,3,6,8-11	1,6,8-11
11: `i <- i + 1`	11	6	1,6,8-11	1,8-11
12: `JUMP 7`			1,8-11	1,8-11
13: `return result`			1-4,6,8-11	1-4,6,8-11
14: `return n`			1-4	1-4

© 2019-20 Goldstein

An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Combine the gen and kill sets into one per BB.

- Gen[BB]={2,3,4,5}
- Kill[BB]={1,8,11}

Code	Gen	kill
`1: i <- 1`	1	8,4
`2: j <- 2`	2	
`3: k <- 3 + i`	3	11
`4: i <- j`	4	1,8
`5: m <- i + k`	5	

© 2019-20 Goldstein

General Procedure

- Create Basic Blocks
- Create Control Flow Graph
- Summarize Basic Block into a single Gen set and a single Kill set
- Iterate over CFG until no changes
- create data flow facts for individual instructions as needed starting with in set of the basic block

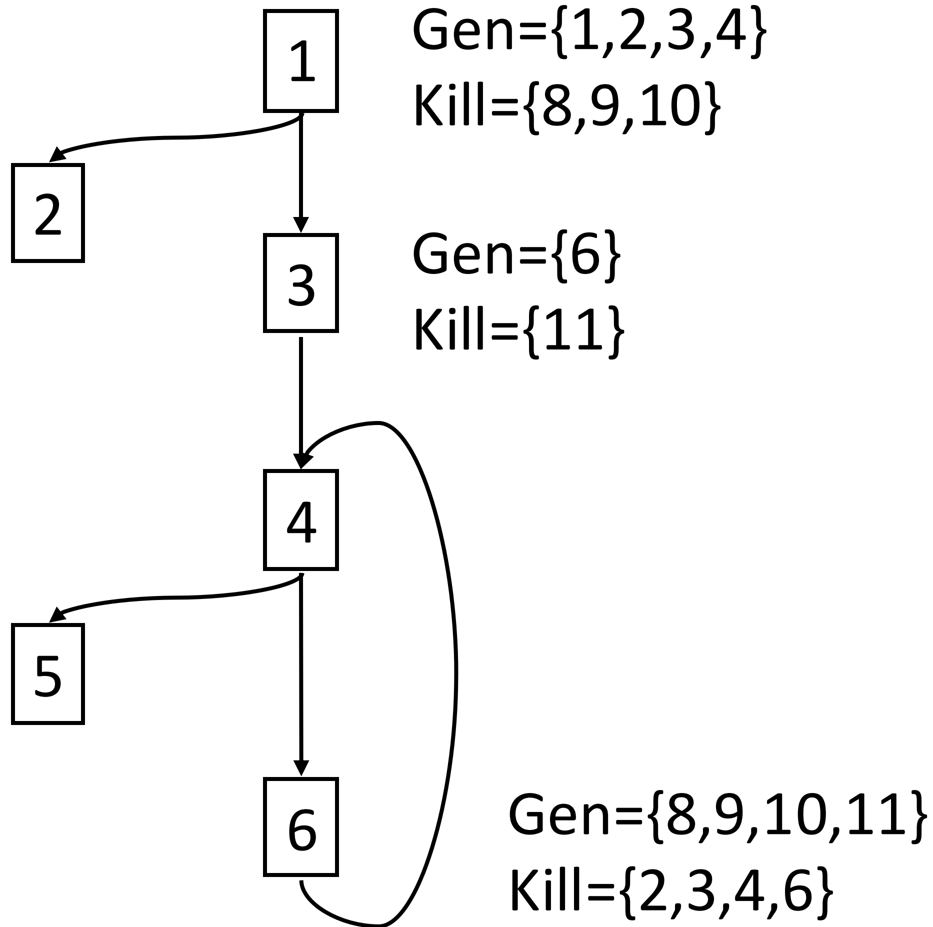
© 2019-20 Goldstein

BB sets

		Gen	kill		
	1: n <- 10	1			
	2: older <- 0	2	9		
1	3: old <- 1	3	10		
	4: result <- 0	4	8		
	5: if n <= 1 goto 14			1,2,3,4	8,9,10
3	6: i <- 2	6	11	6	11
4	7: if i > n goto 13				
	8: result <- old + older	8	4		
	9: older <- old	9	2		
6	10: old <- result	10	3		
	11: i <- i + 1	11	6		
	12: JUMP 7			8-11	2-4,6
5	13: return result				
2	14: return n				

© 2019-20 Goldstein

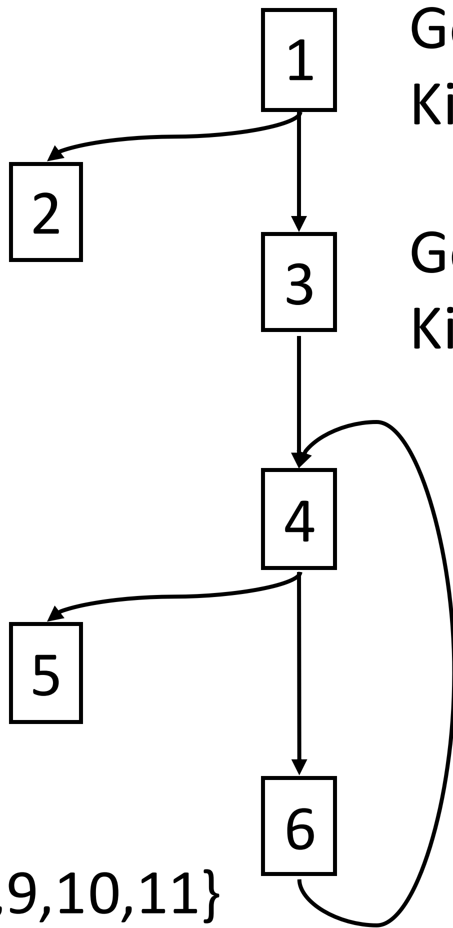
BB sets



In	out
	1,2,3,4

© 2019-20 Goldstein

BB sets



Gen={1,2,3,4}
Kill={8,9,10}

Gen={6}
Kill={11}

Gen={8,9,10,11}
Kill={2,3,4,6}

In	out
	1,2,3,4
1,2,3,4	1,2,3,4,6
1-4,6,8-11	1-4,6,8-11
1-4,6,8-11	1,8-11

© 2019-20 Goldstein

Forward Dataflow

- Reaching definitions is a forward dataflow problem:
  It propagates information from the predecessors of a node to the node
- Defined by:
  - Basic attributes: (gen and kill)
  - Transfer function: F_{bb} $out[n] = gen[n] \bigcup (in[n] - kill[n])$
  - Meet operator: union $in[n] = \bigcup_{p \in pred[n]} out[p]$
  - Set of values (a lattice, in this case powerset of program points)
  - Initial values for each node b
- Solve for fixed point solution

© 2019-20 Goldstein

How to implement?

- Values?
- Gen?
- Kill?
- F_{bb}?
- Init?
- Order to visit nodes?
- When are we done?
  - In fact, do we know we terminate?

© 2019-20 Goldstein

Implementing RD

- Values: bits in a bit vector
- Gen: 1 in each position generated, otherwise 0
- Kill: 0 in each position killed, otherwise 1
- F_{bb}: out[b] = gen[b] | (in[b] & kill[b])
- Init in[b]=out[b]=0

- When are we done?
- What order to visit nodes? Does it matter?

© 2019-20 Goldstein

RD Worklist algorithm

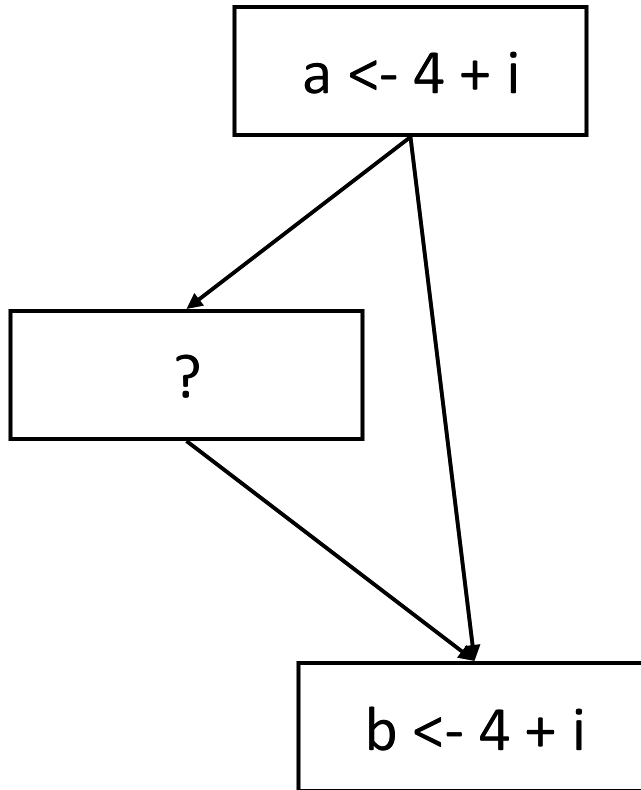
```
Initialize: in[B] = out[b] = ∅
Initialize: in[entry] = ∅
Work queue, W = all Blocks in topological order
while (|W|  != 0)  {
    remove b from W
    old = out[b]
    in[b] =  {over all pred(p) ∈ b} ∪ out[p]
    out[b] = gen[b] ∪ (in[b] - kill[b])
    if (old != out[b]) W = W ∪ succ(b)
}
```

So Far

	Union (may)	intersection (must)
Forward	Reaching defs	
Backward	Live variables	

© 2019-20 Goldstein

When can we do CSE?



© 2019-20 Goldstein

Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

```
a <- b+c
b <- a-d
c <- b+c
d <- a-d
```


© 2019-20 Goldstein

Available Expressions

- X+Y is "available" at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

```
a <- b+c

b <- a-d

c <- b+c  ⟵  b+c Not available, since b redefined

d <- a-d  ⟵  a-d is available
```


© 2019-20 Goldstein

Computing Available Expressions

- Forward or backward?
- Values?
- Lattice?
- gen[b] =
- kill[b] =
- out[b] =
- in[b] =
- initialization?

© 2019-20 Goldstein

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, then all exprs using x are killed.
- out[b] = (in[b] – kill[b]) $\cup$ gen[b]
- in[b] = what to do at a join point?
- initialization?

© 2019-20 Goldstein

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- gen[b] = if b evals expr e and doesn't define variables used in e
- kill[b] = if b assigns to x, exprs(x) are killed
- out[b] = (in[b] – kill[b]) $\cup$ gen[b]
- in[b] = An expr is avail only if avail on ALL edges, so: in[b] = $\cap$ over all p$\in$ pred(b), out[p]
- Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail

© 2019-20 Goldstein

Constructing Gen & Kill

Stmt s	Gen	Kill
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	
M[a] <- b		
f(a, …)		{M[x] for all x}
t <- f(a,…)		

© 2019-20 Goldstein

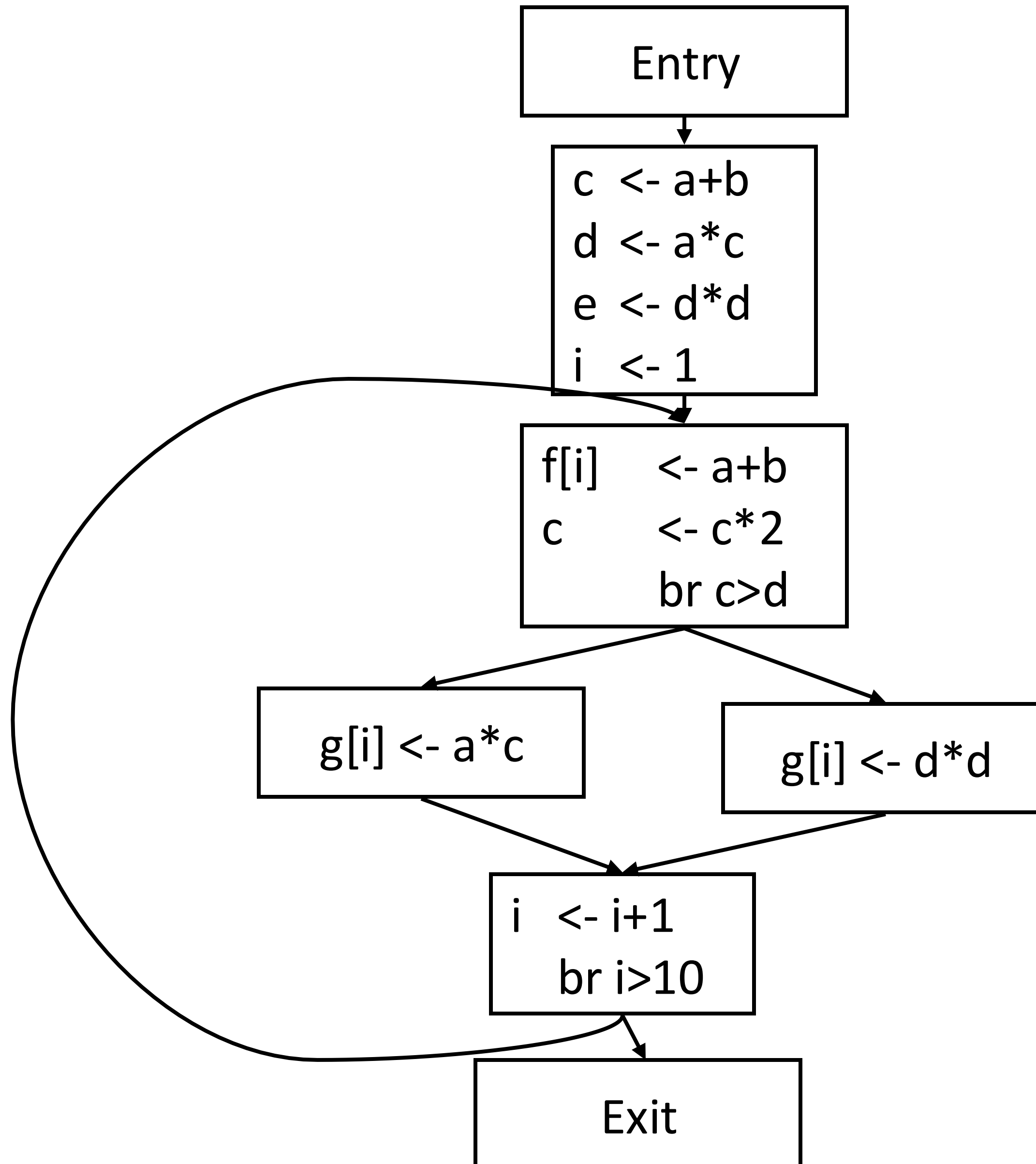
Constructing Gen & Kill

Stmt s	Gen	Kill
t <- x op y	{x op y}-kill[s]	{exprs containing t}
t <- M[a]	{M[a]}-kill[s]	{exprs containing t}
M[a] <- b	{}	{for all x, M[x]}
f(a, …)	{}	{for all x, M[x]}
t <- f(a,…)	{}	{exprs containing t for all x, M[x]}

© 2019-20 Goldstein

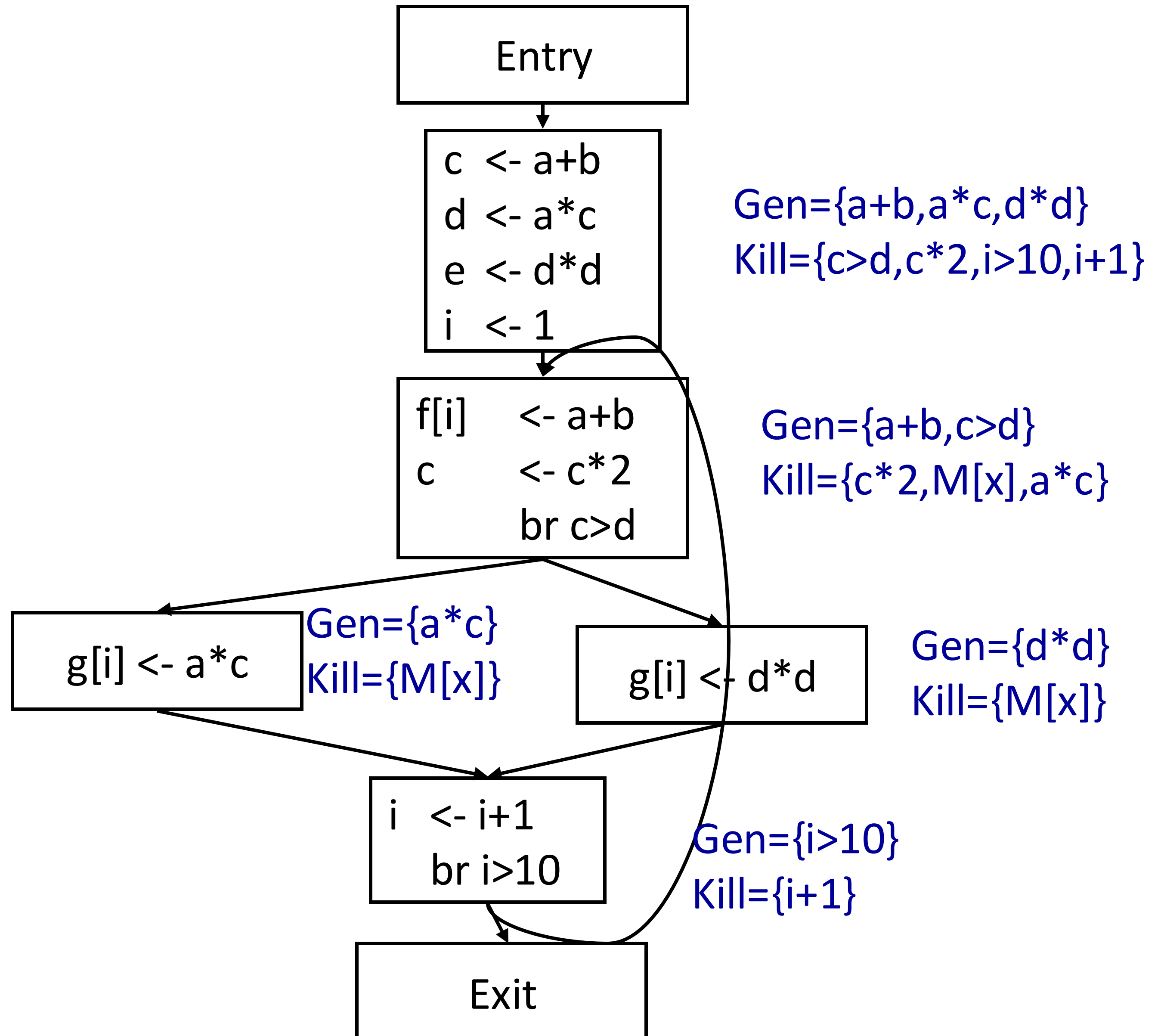
Example

$$out[b] = (in[b] - kill[b]) \cup gen[b]$$
$$in[b] = \cap \text{ over } p \in pred(b), out[p]$$



© 2019-20 Goldstein

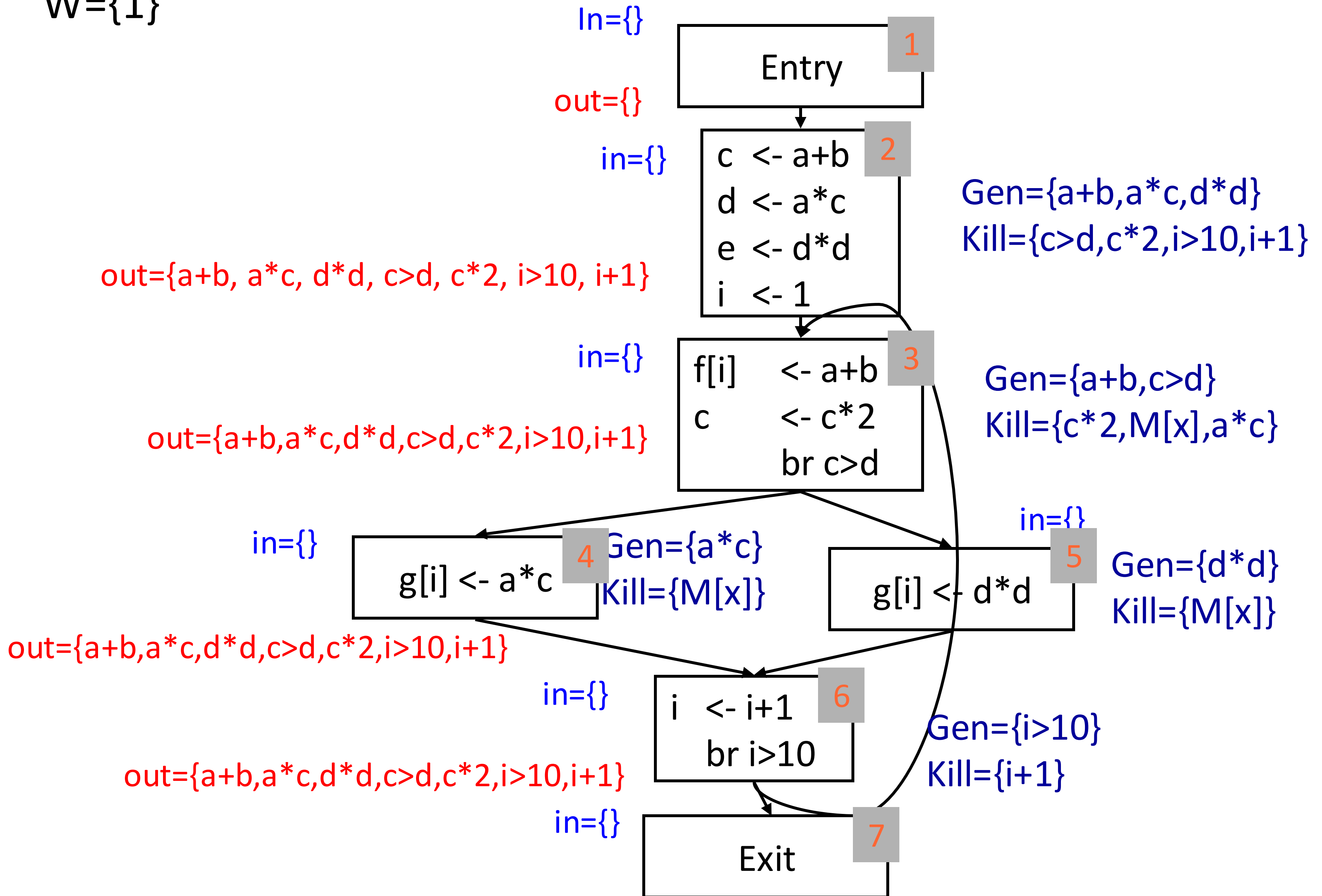
Example



© 2019-20 Goldstein

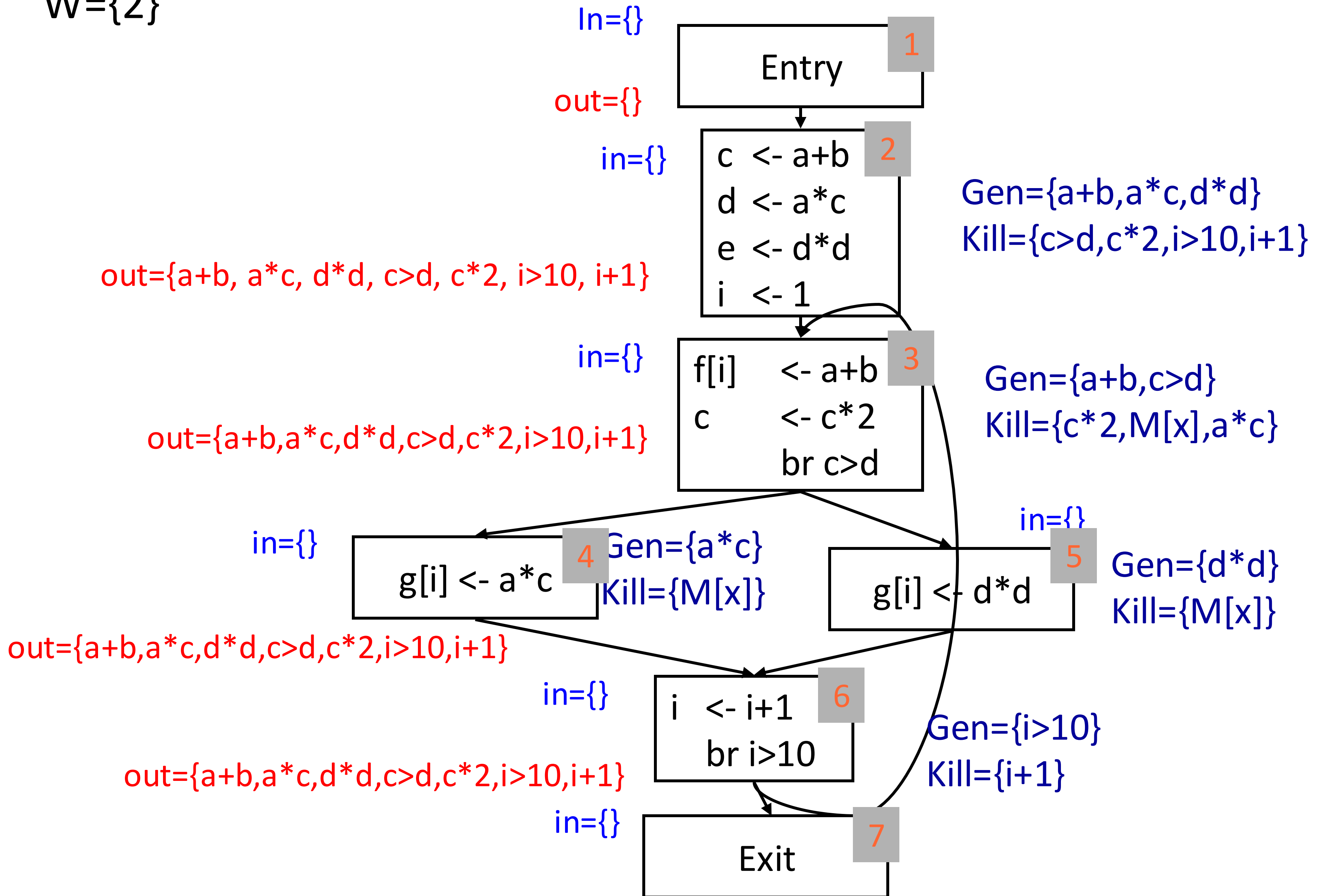
Example

W={1}



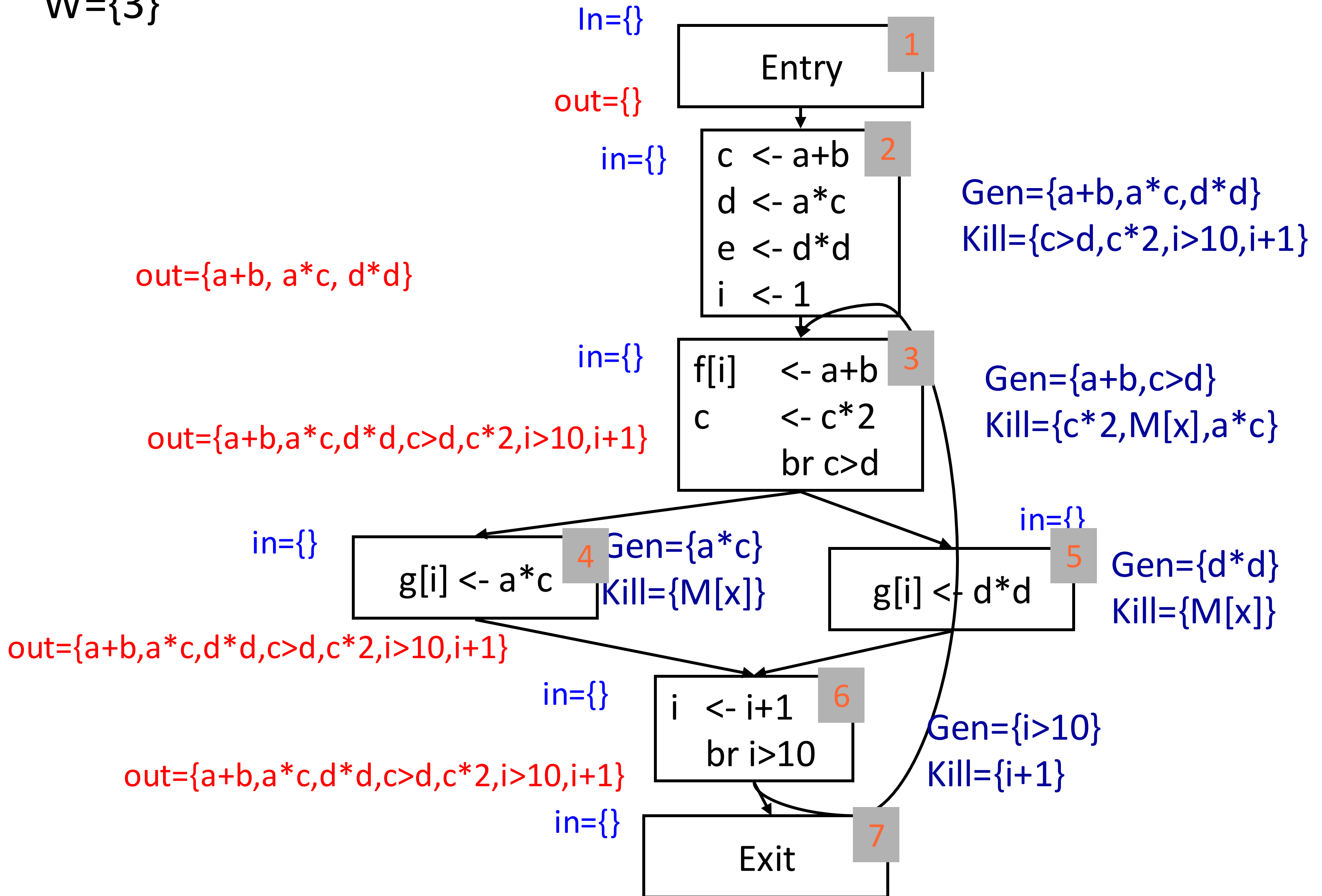
© 2019-20 Goldstein

Example



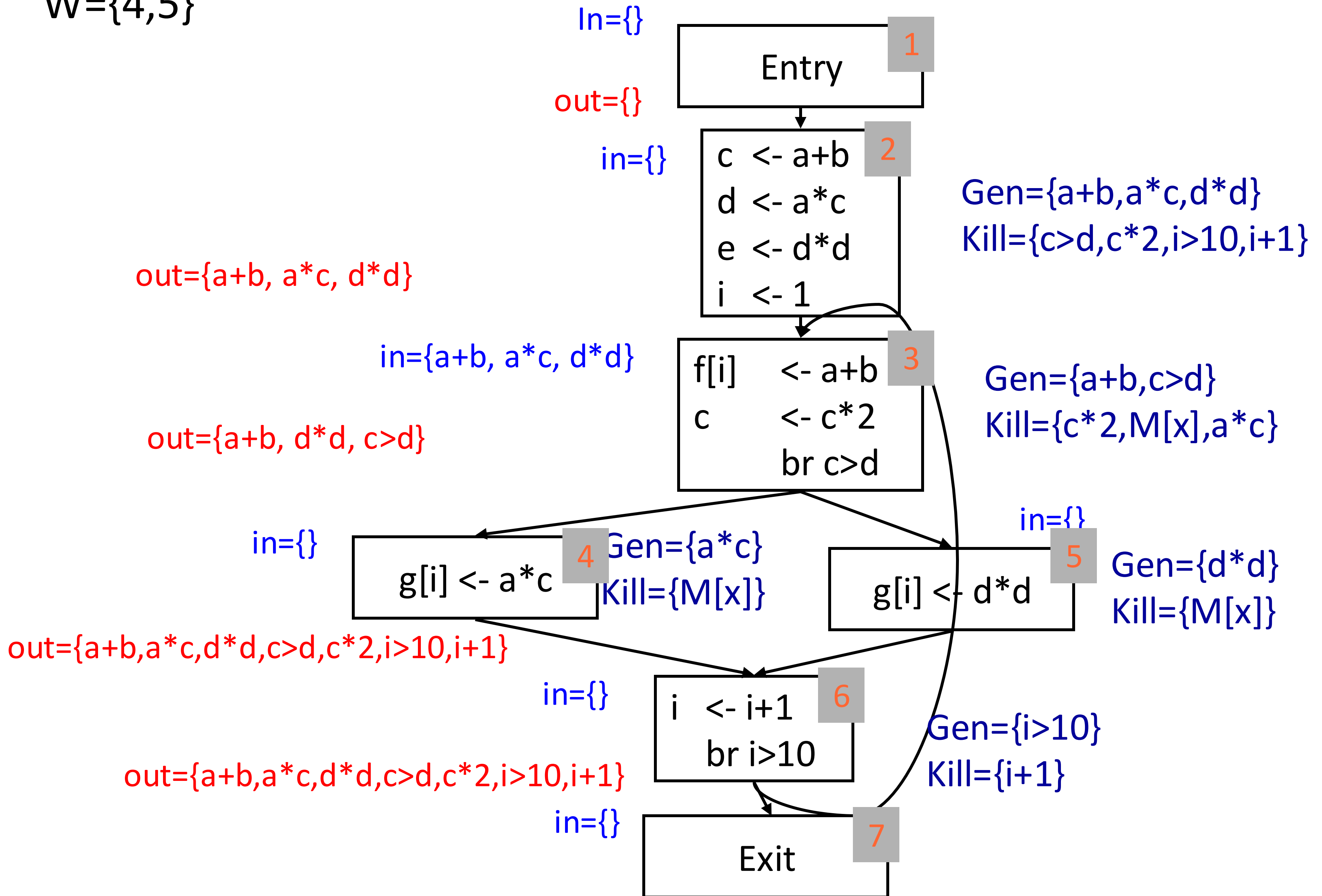
© 2019-20 Goldstein

Example



© 2019-20 Goldstein

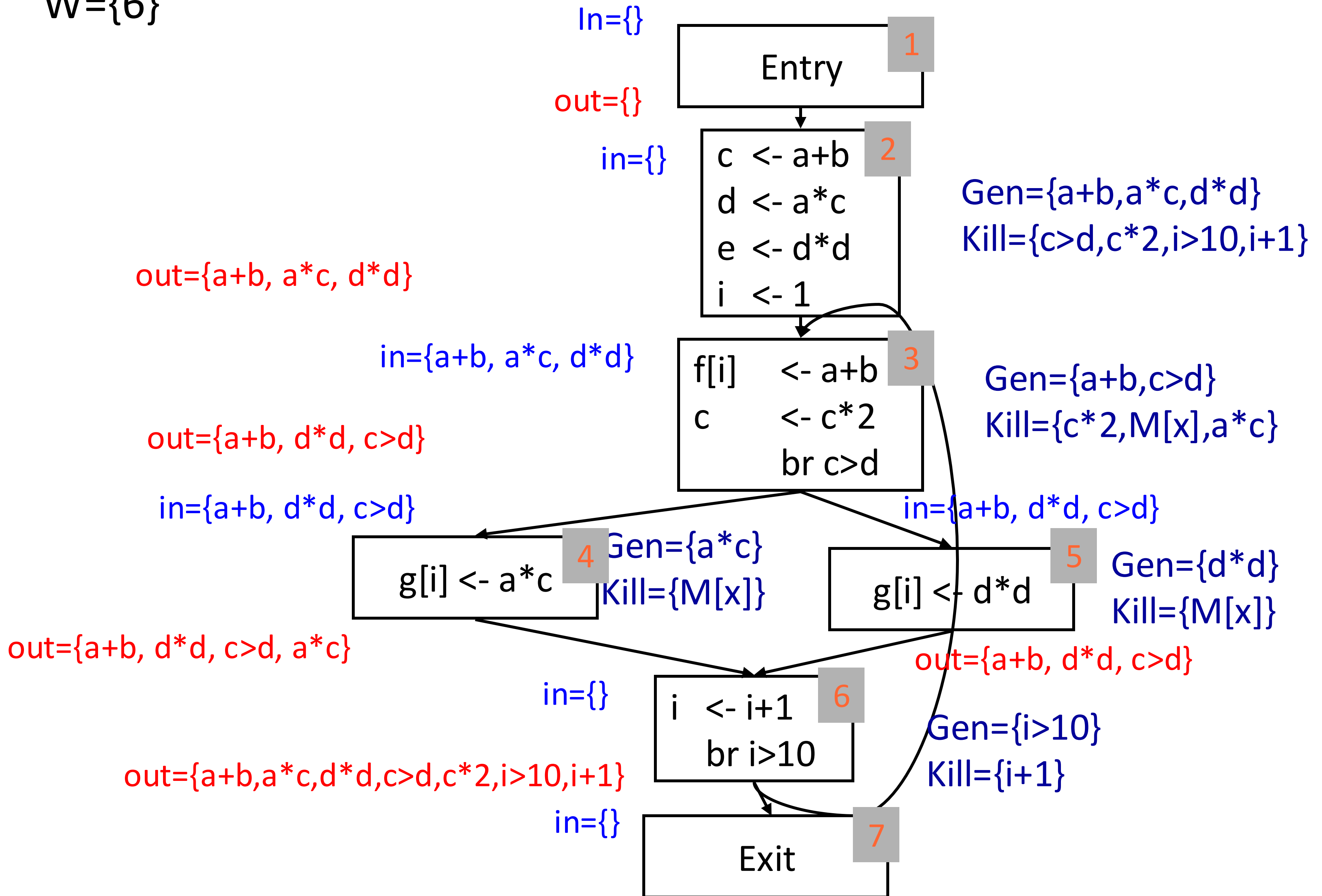
Example



© 2019-20 Goldstein

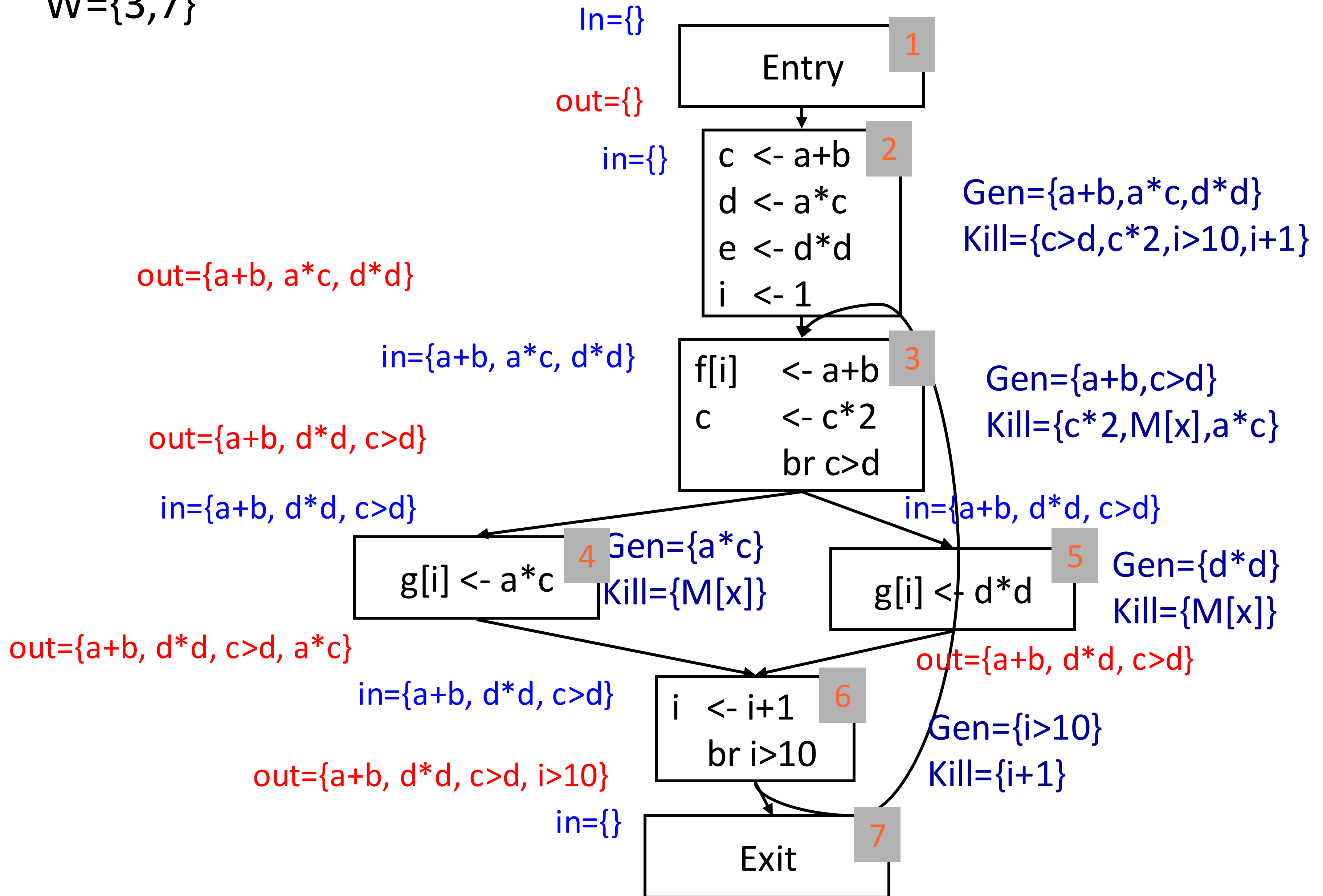
Example

W={6}



© 2019-20 Goldstein

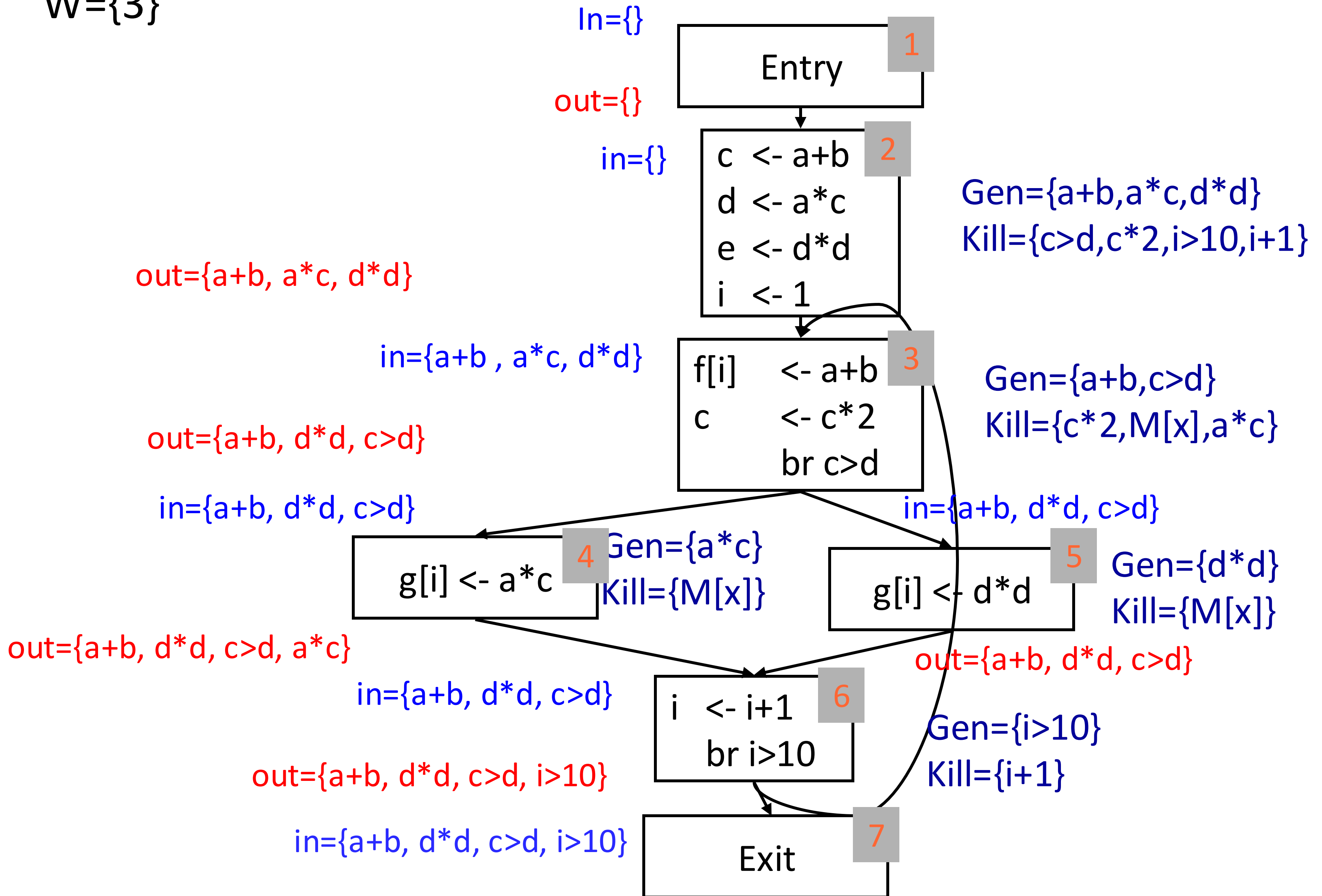
Example



© 2019-20 Goldstein

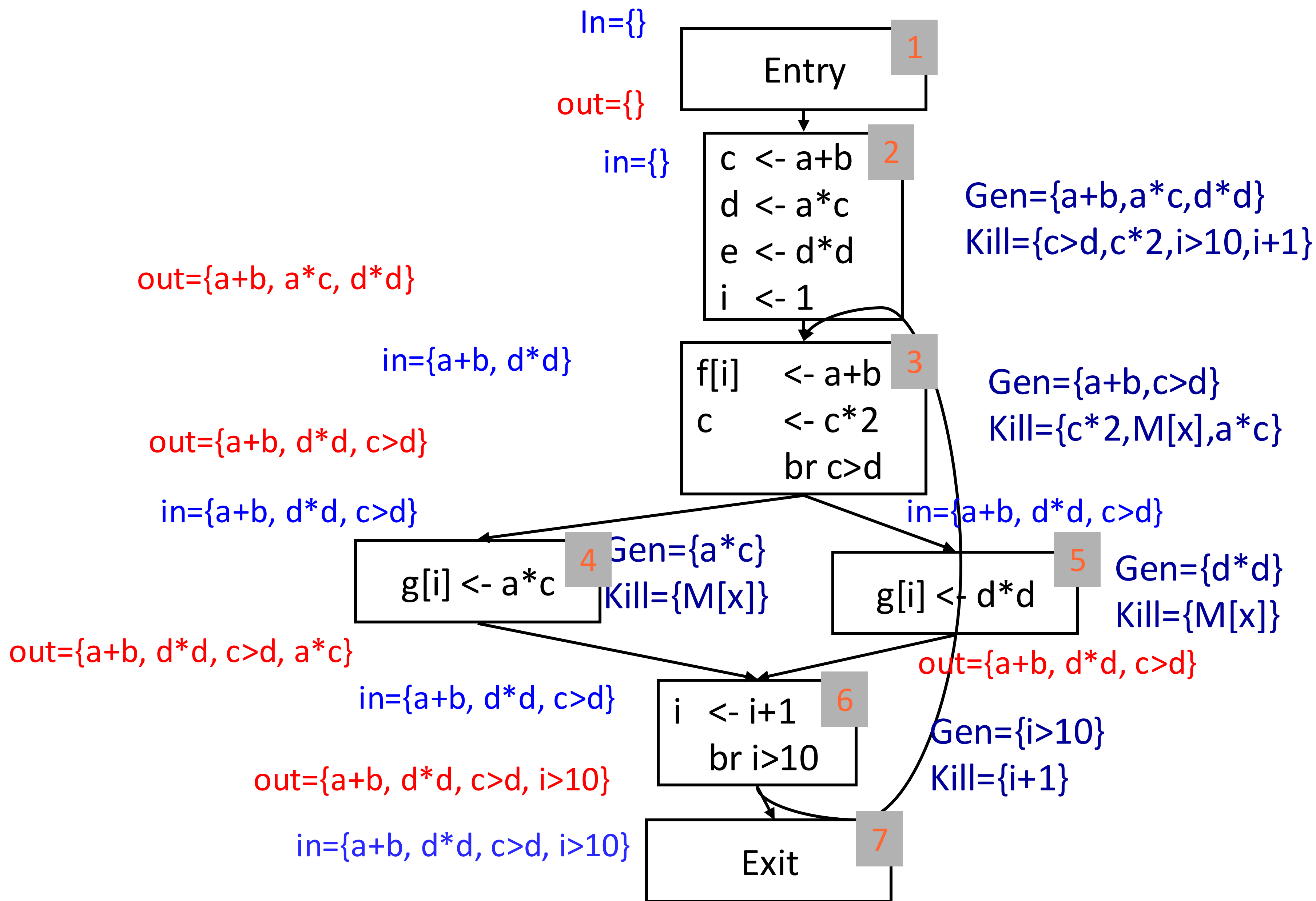
Example

W={3}



© 2019-20 Goldstein

Example



© 2019-20 Goldstein

CSE

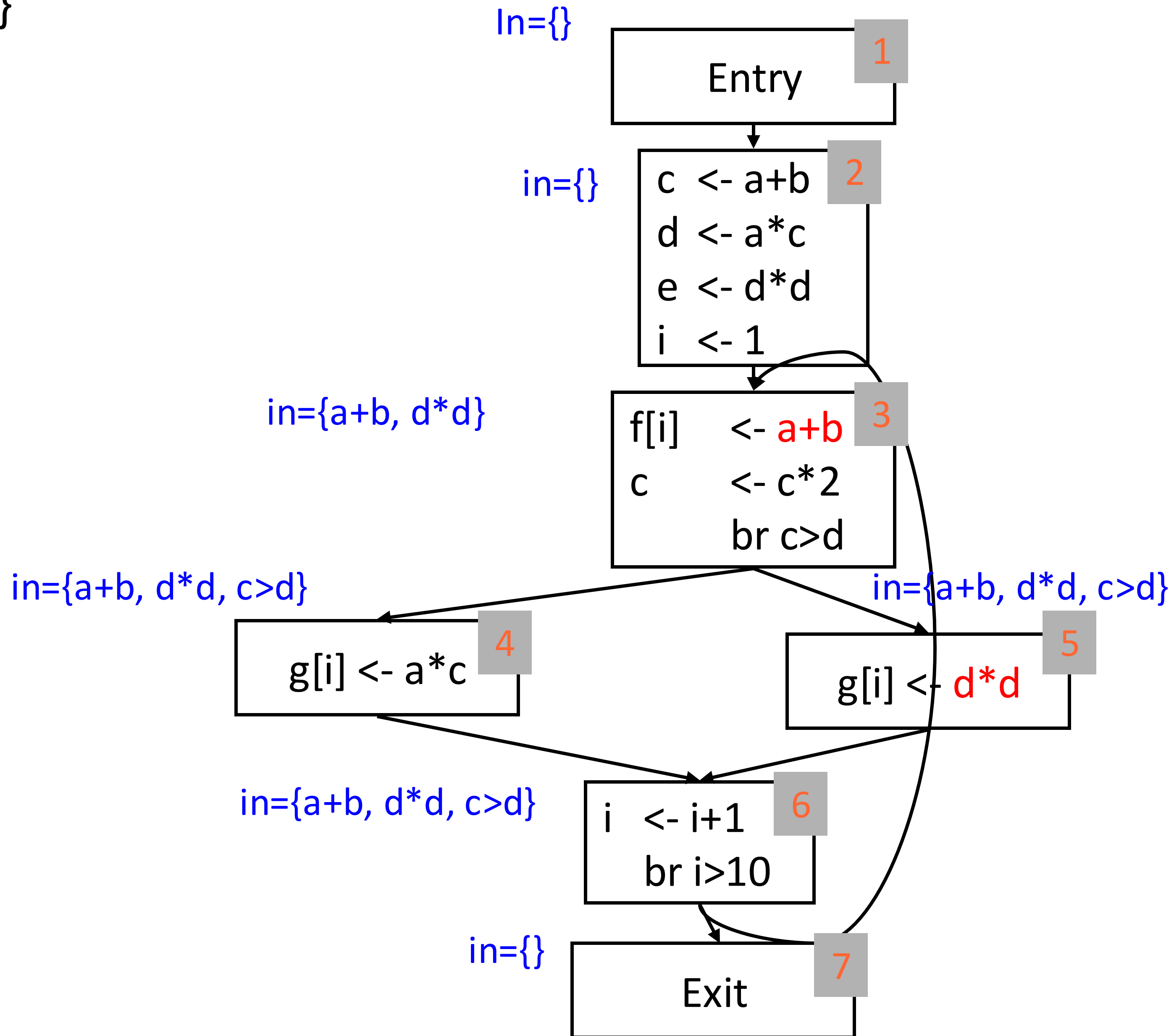
- Calculate Available expressions
- For every stmt in program

```
If expression, x op y, is available {
    Compute reaching expressions for "x op y"
        at this stmt
    foreach stmt in RE of the form t <- x op y
            rewrite at:   t' <- x op y
                          t <- t'
    }
    replace "x op y" in stmt with t'
}
```


© 2019-20 Goldstein

Find x op y available

W={3}



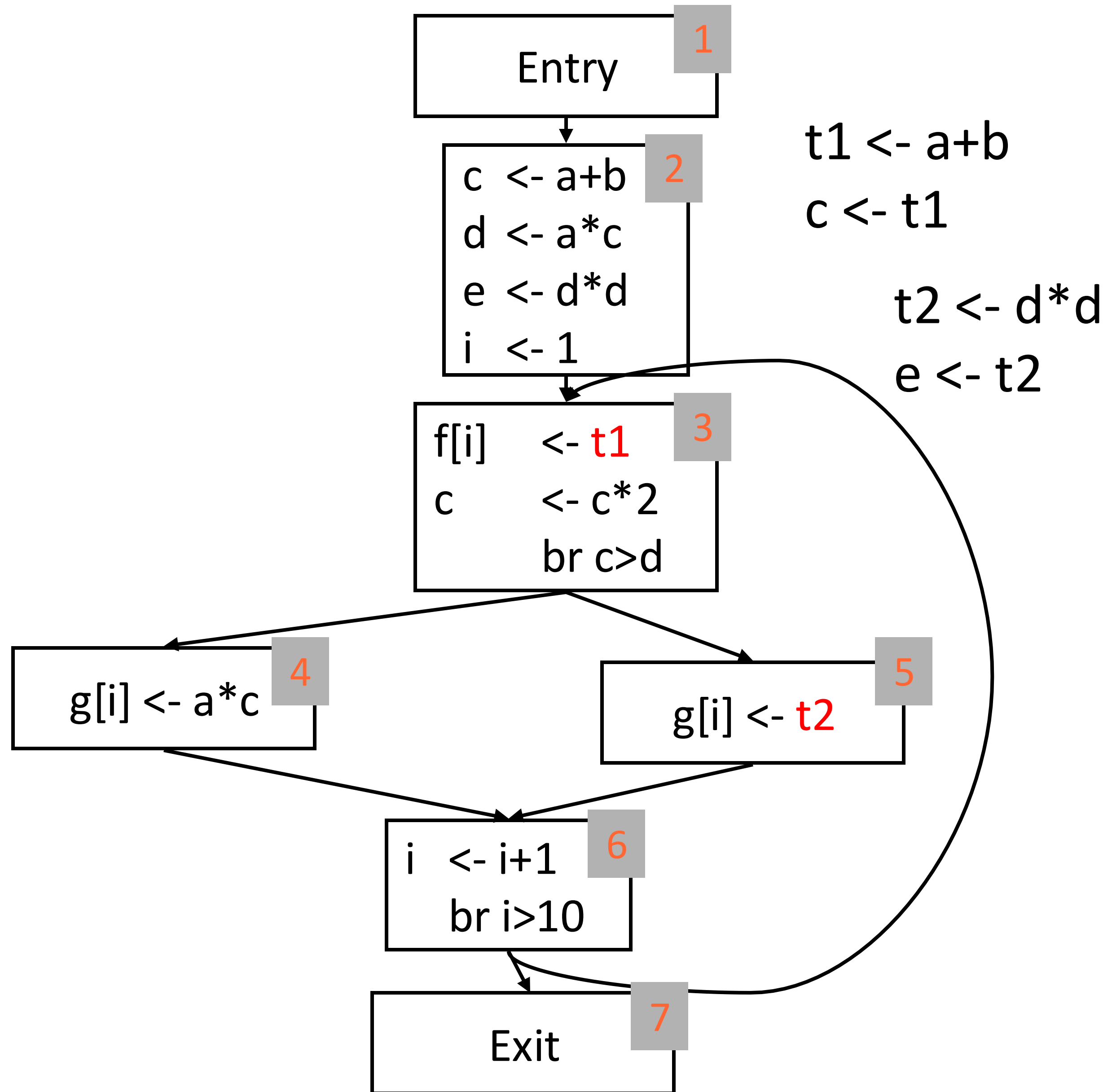
© 2019-20 Goldstein

Calculating Reaching Expressions

- Could be dataflow problem, but not needed enough, so …
- To find RE for “x op y” at stmt S
  - traverse cfg backward from S until
    - reach t <- x + y (& put into RE)
    - reach definition of x or y

© 2019-20 Goldstein

Example



© 2019-20 Goldstein

Dataflow Summary

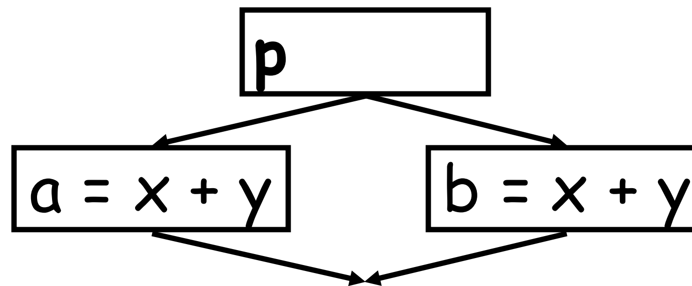
	Union (may)	intersection (must)
Forward	Reaching defs	Available exprs
Backward	Live variables	very busy exprs

Later in course we look at bidirectional dataflow

© 2019-20 Goldstein

Very Busy Expressions

- A Backward, Must data flow analysis
- An expression e is *very busy at point p* if On every path from p, e is evaluated before the value of e is changed
- Optimization
  - Can hoist very busy expression computation



© 2019-20 Goldstein

Forward Must Data Flow Algorithm

Out(s) = Gen(s) for all statements s

W = {all statements}

Repeat

  Take s from W

  $In(s) = \bigcap_{s' \in pred(s)} Out(s')$

  $Temp = Gen(s) \cup (In(s) - Kill(s))$

  If (temp != Out(s)) {

    Out(s) = temp

    $W = W \cup succ(s)$

  }

Until $W = \varnothing$

© 2019-20 Goldstein

Forward May Data Flow Algorithm

```
Out(s) = Gen(s) for all statements s
W = {all statements}
Repeat
        Take s from W
        In(s) = ∪ s' ∈ pred(s) Out(s')
        Temp = Gen(s) ∪ (In(s) – Kill(s))
        If (temp != Out(s)) {
                Out(s) = temp
                W = W ∪ succ(s)
        }
Until W = ∅
```


© 2019-20 Goldstein

Backward May Data Flow Algorithm

In(s) = Gen(s) for all statements s

W = {all statements} (worklist)

Repeat

    Take s from W

    Out(s) = $\cup_{s' \in succ(s)}$ In(s')

    Temp = Gen(s) $\cup$ (Out(s) – Kill(s))

    If (temp != In(s)) {

        In(s) = temp

        W = W $\cup$ pred(s)

    }

Until W = $\varnothing$

© 2019-20 Goldstein

Backward Must Data Flow Algorithm

In(s) = Gen(s) for all statements s

W = {all statements} (worklist)

Repeat

    Take s from W

    Out(s) = $\bigcap_{s' \in succ(s)}$ In(s')

    Temp = Gen(s) $\cup$ (Out(s) – Kill(s))

    If (temp != In(s)) {

        In(s) = temp

        W = W $\cup$ pred(s)

    }

Until W = $\varnothing$

© 2019-20 Goldstein

Dataflow Analysis

- A framework for proving facts about program
  - Reasons about lots of little facts
  - Little or no interaction between facts
  - Based on all paths through program
- Solve with iterative solver:
  - How do we know it terminates?
  - How do we know whether solution is precise? (or even correct?)

Data Flow Equations

- Let s be a statement
  - Succ(s) = {immediate successors of s}
  - Pred(s) = {immediate predecessors of s}
  - In(s) program point just before executing s
  - Out(s) program point just after executing s
- Transfer functions (for forward, must):

$$In(s) = \bigcap_{s' \in \mathrm{pred}(s)} Out(s')$$

$$Out(s) = Gen(s) \cup (In(s) - Kill(s))$$

- Gen(s) set of facts made true by s
- Kill(s) set of facts invalidated by s

Worklist algorithm (forward)

```
Initialize: in[B] = out[b] = Universe
Initialize: in[entry] = ∅
Work queue, W = all Blocks in topological order
while (|W|  != 0)  {
    remove b from W
    temp = out[b]
    compute In[b]
    compute Out[b]
    if (temp != out[b]) W = W ∪ succ(b)
}
```

Some Unidirectional Dataflow Analysis

	Union (may)	intersection (must)
Forward	Reaching definitions	Available expressions
Backward	Live variables	very busy expressions

Available Expressions

- X+Y is “available” at statement S if
  - x+y is computed along every path from the start to S AND
  - neither x nor y is modified after the last evaluation of x+y

a <- b+c

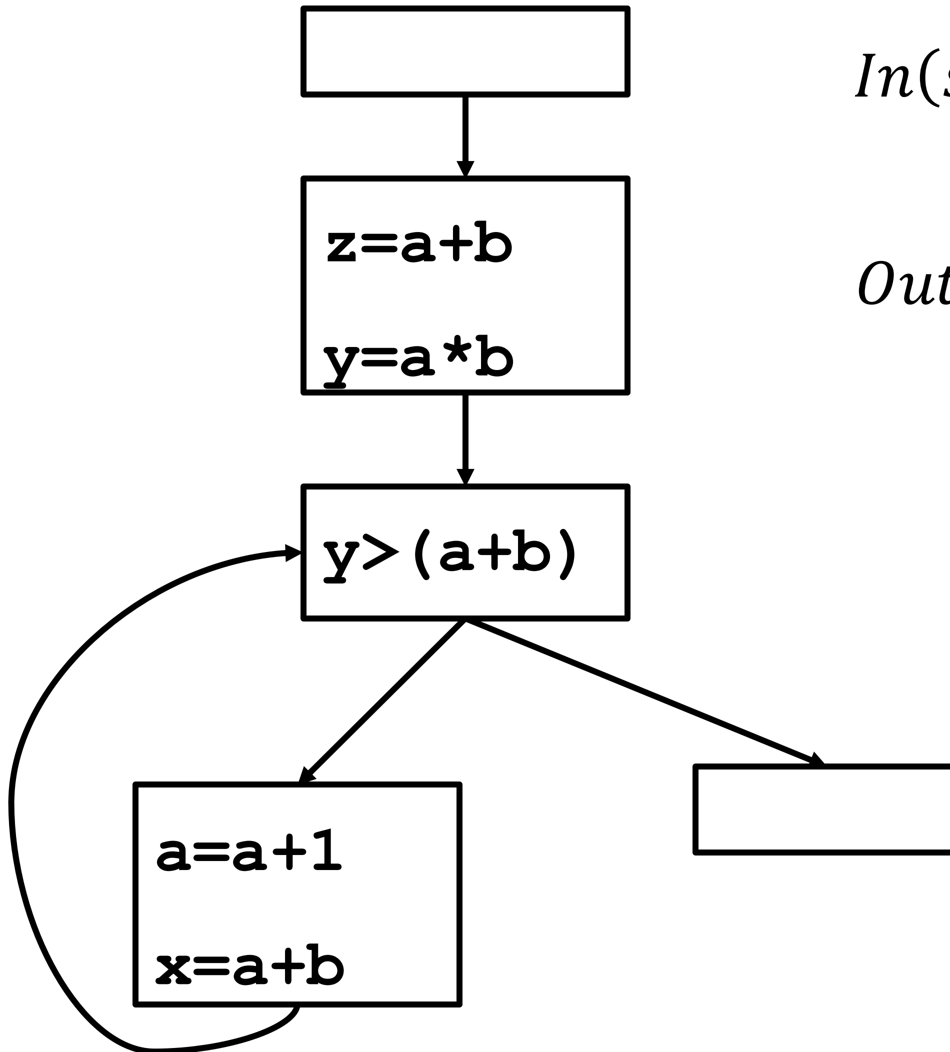
b <- a-d

c <- b+c ⟵ b+c Not available, since b redefine

d <- a-d ⟵ a-d is available

© 2019-20 Goldstein

Available Expressions



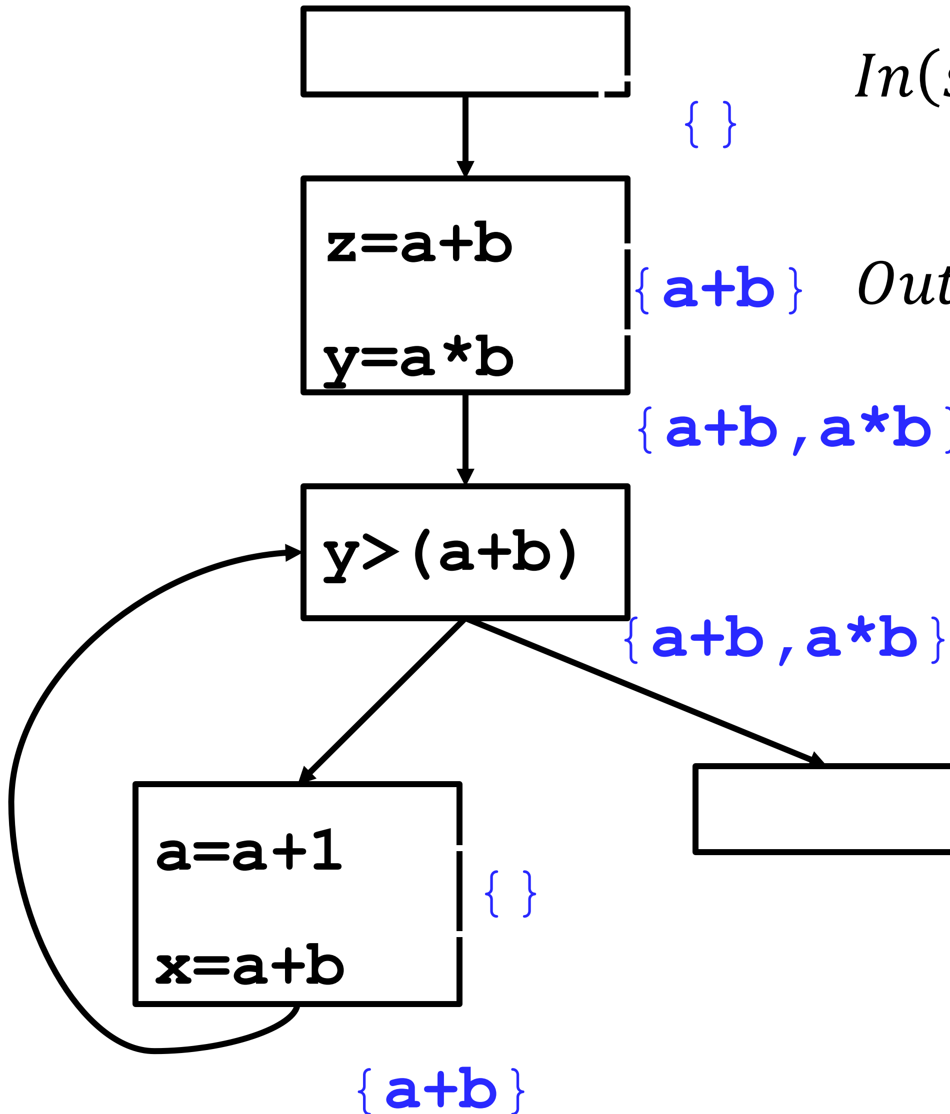
$$In(s) = \bigcap_{s' \in \text{pred}(s)} Out(s')$$

$$Out(s) = \left(In(s) \cup Gen(s)\right) - Kill(s)$$

For x= a⊕b:
- Gen = {a⊕b}
- Kill = {All expressions using x}

Initialize all but entry to universe of expressions

Available Expressions



$$In(s) = \bigcap_{s' \in \text{pred}(s)} Out(s')$$

$$Out(s) = \big(In(s) \cup Gen(s)\big) - Kill(s)$$

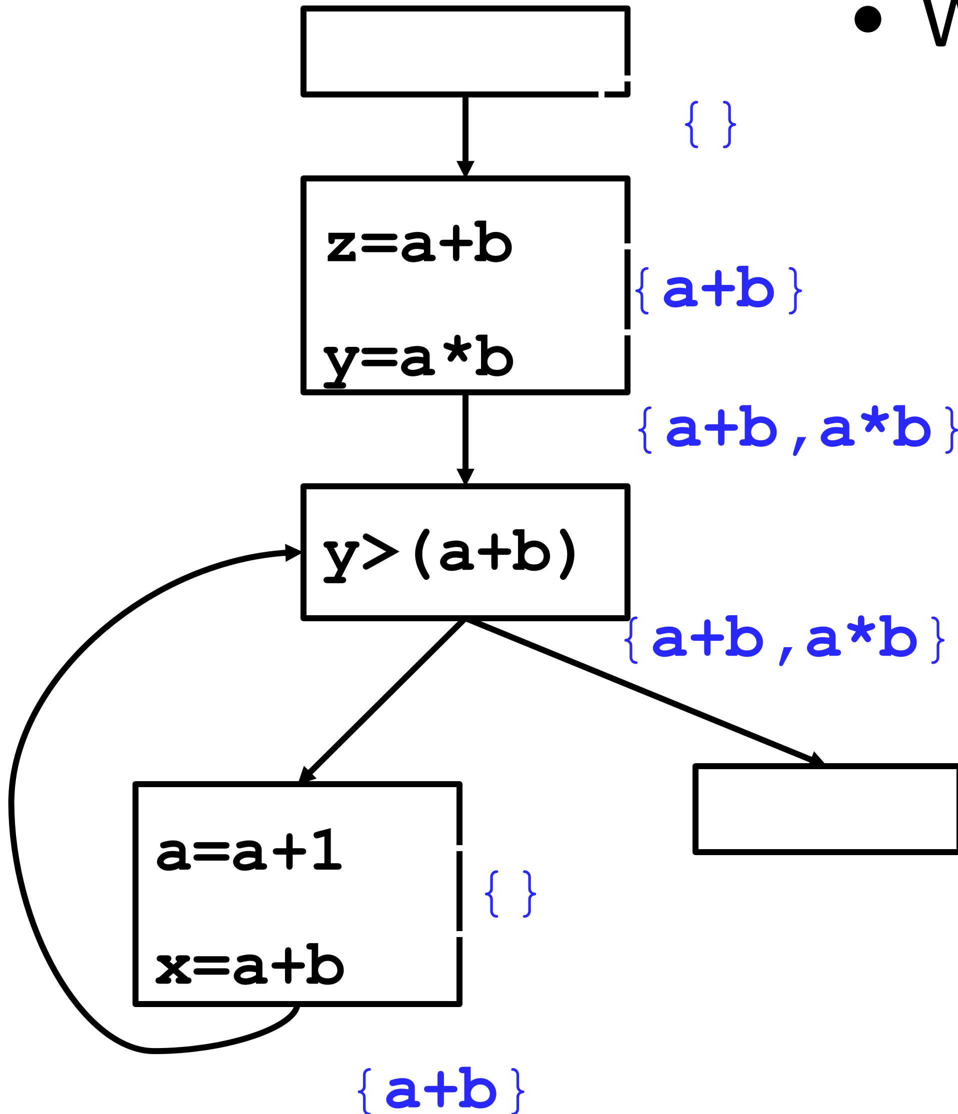
For x= a⊕b:

Gen = {a⊕b}

Kill = {All expressions using x}

Initialize all but entry to universe of expressions

Available Expressions

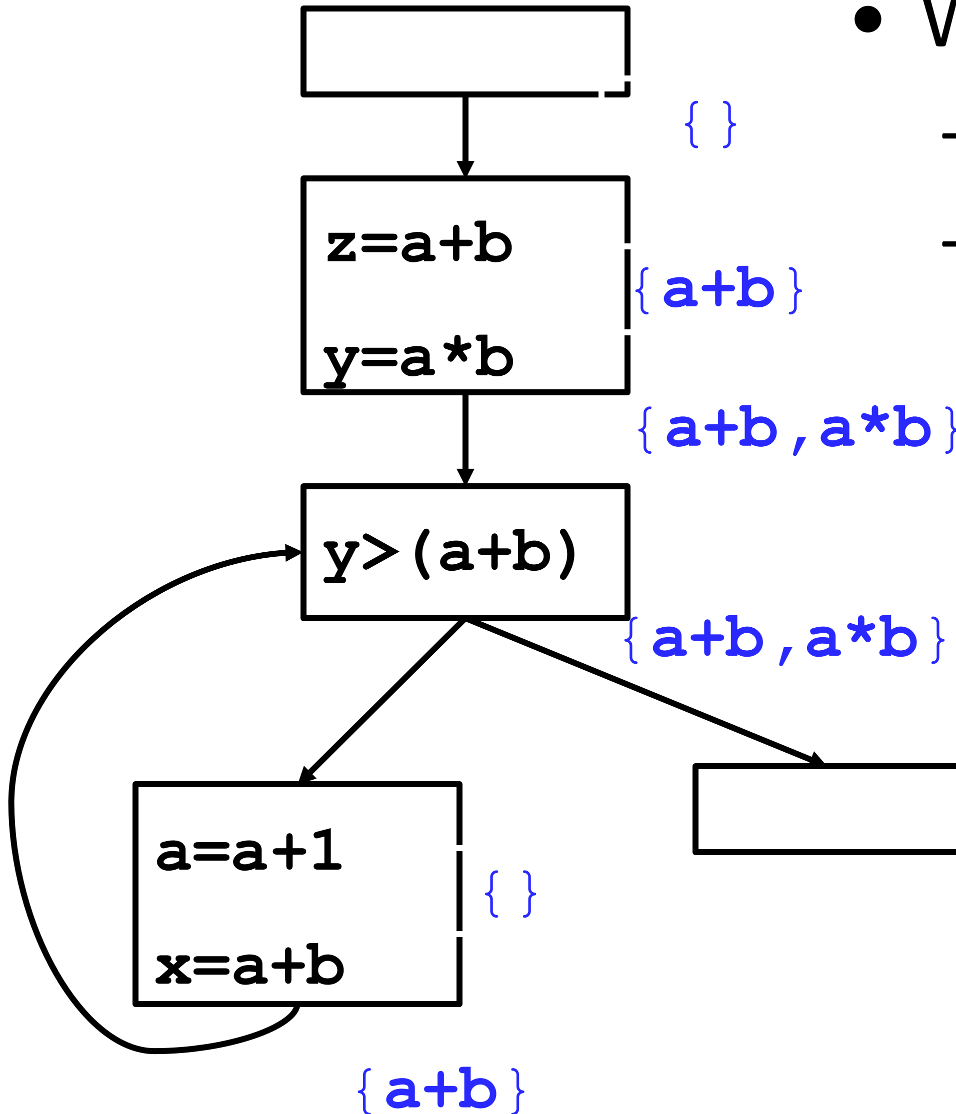


- Why Does this terminate?

$$In(s) = \bigcap_{s' \in \mathrm{pred}(s)} Out(s')$$

$$Out(s) = \left(In(s) \cup Gen(s)\right) - Kill(s)$$

Available Expressions



- Why Does this terminate?
  - In(s) never grows
  - Out(s) never grows

$$In(s) = \bigcap_{s' \in \mathrm{pred}(s)} Out(s')$$

$$Out(s) = \big(In(s) \cup Gen(s)\big) - Kill(s)$$

Liveness as a dataflow problem

- This is a backwards analysis
  - A variable is live out if used by a successor
  - Gen: For a use: indicate it is live coming into s
  - Kill: Defining a variable v in s makes it dead before s (unless s uses v to define v)
  - Lattice is just live (top) and dead (bottom)
- Values are variables
- In[n] = variables live before n

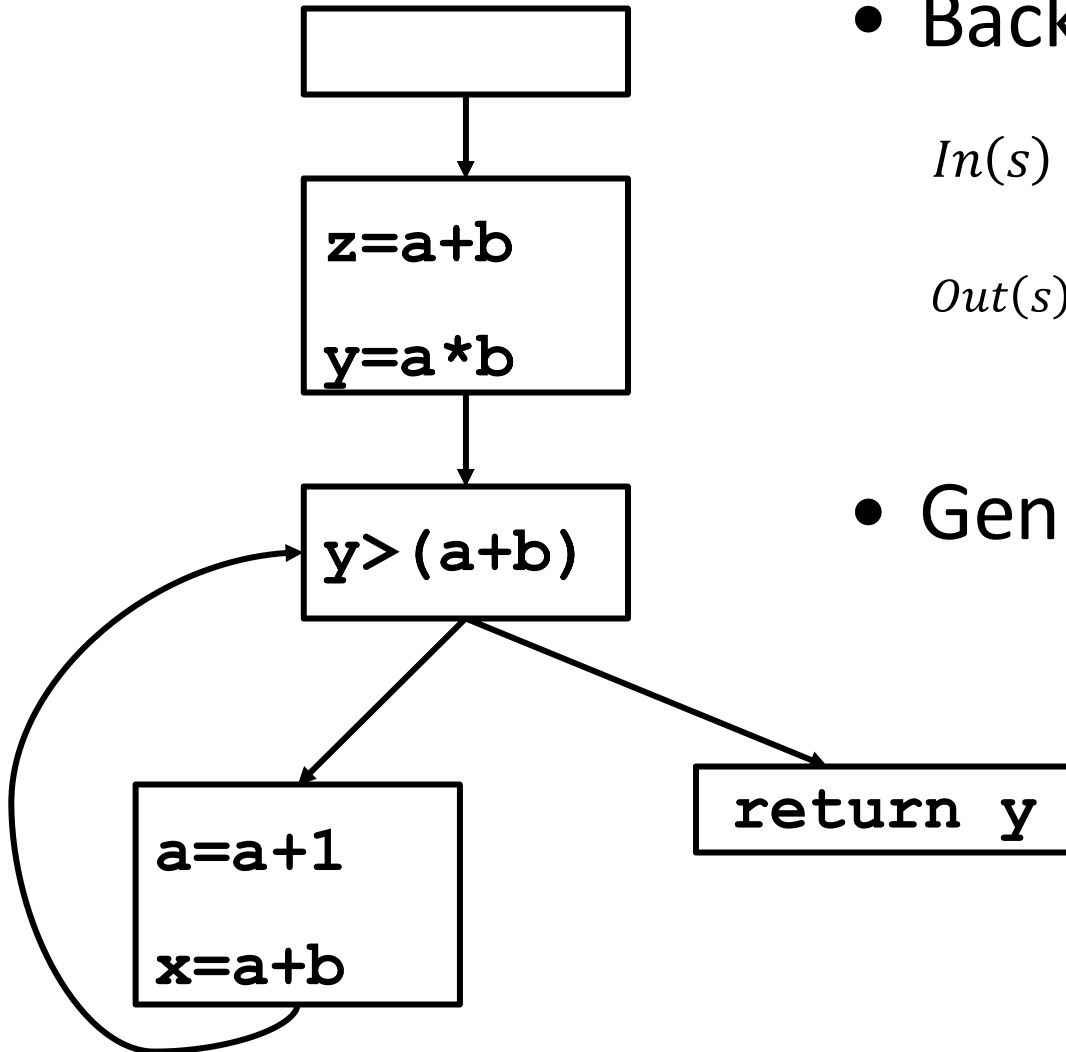
  $$= (out[n] - kill[n]) \cup gen[n]$$

- Out[n] = variables live after n

  $$= \bigcup_{s \in succ(n)} In[s]$$

© 2019-20 Goldstein

Liveness



- Backward, May

$$In(s) = \big(Out(s) - kill(s)\big) \cup Gen(s)$$

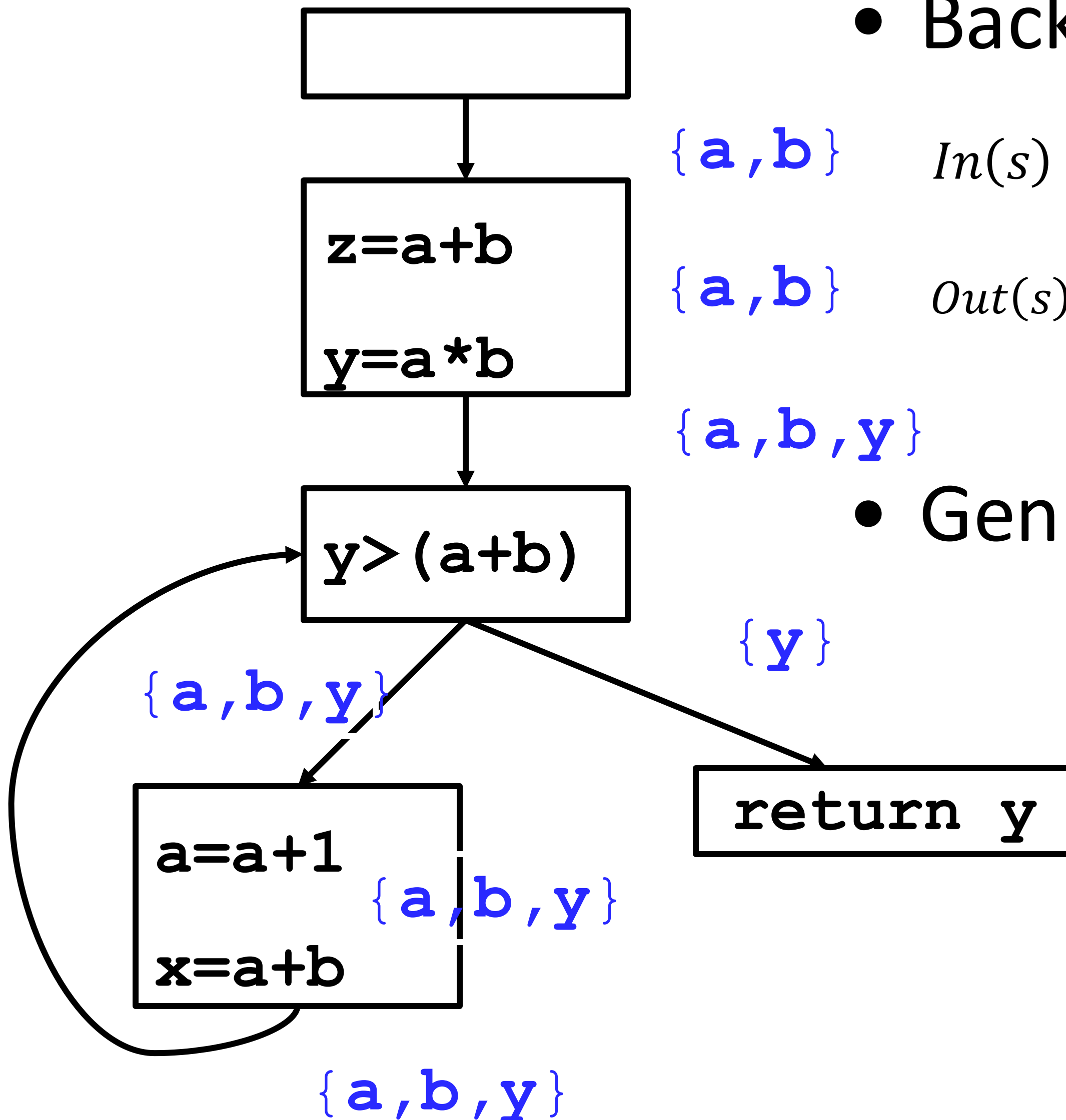
$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

- Gen:

For x= a⊕b:
  Gen = {a,b}
  Kill = {x}

Initialize all to empty set

Liveness



- Backward, May

$$In(s) = \big(Out(s) - kill(s)\big) \cup Gen(s)$$

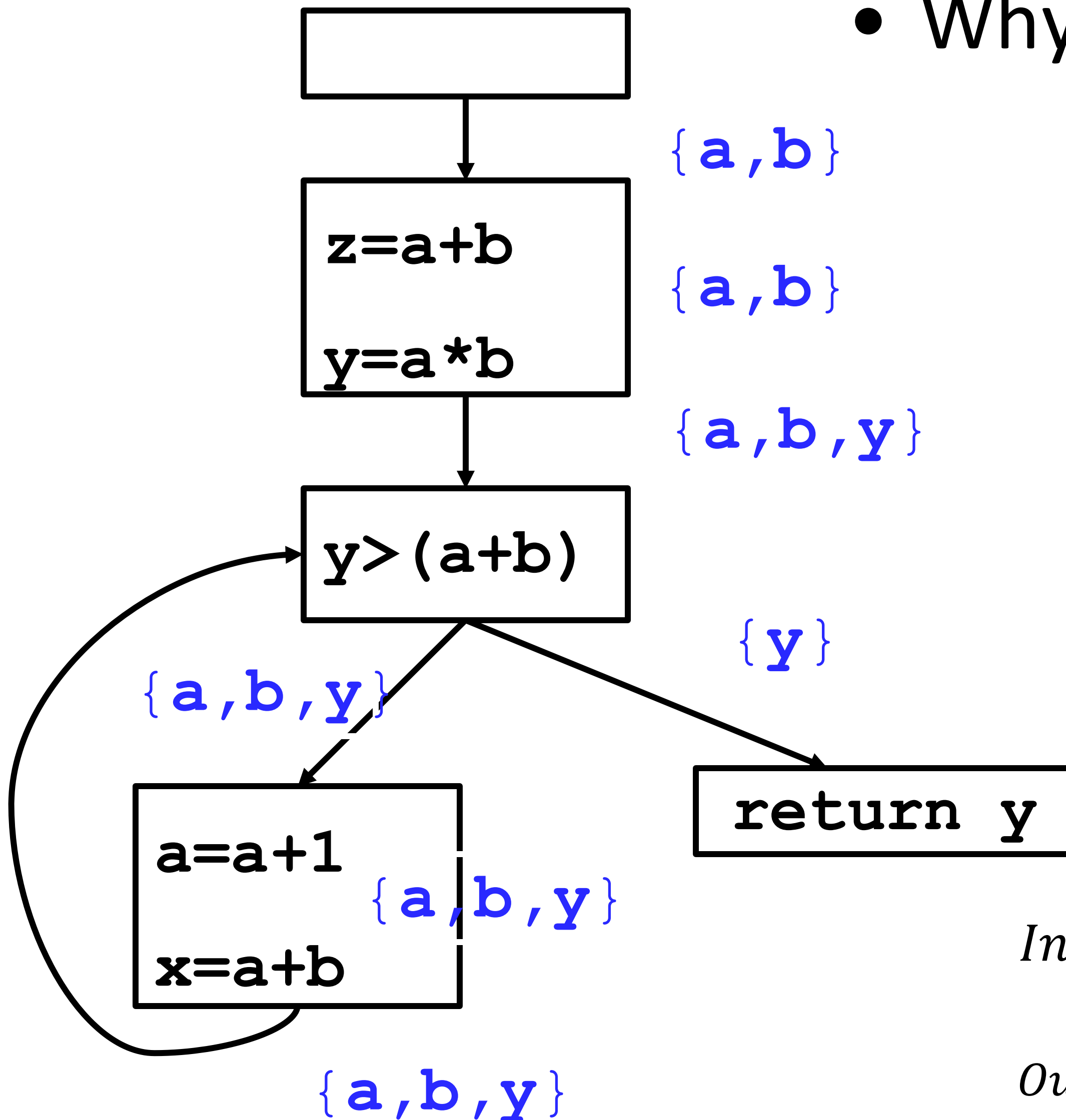
$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

- Gen:

For x= a⊕b:
Gen = {a,b}
Kill = {x}

Initialize all to empty set

Liveness

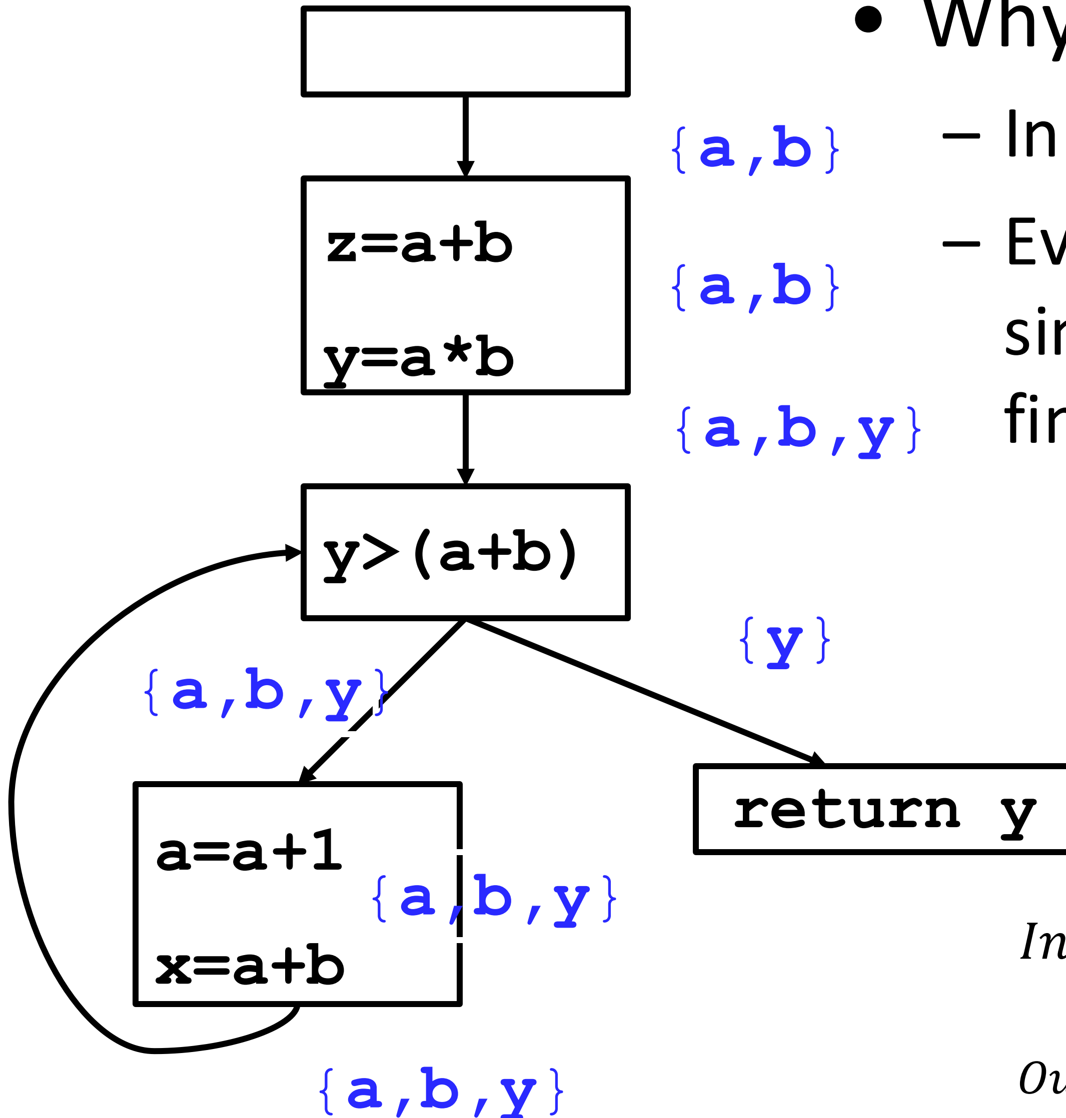


- Why does this terminate?

$$In(s) = \left(Out(s) - kill(s)\right) \cup Gen(s)$$

$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

Liveness



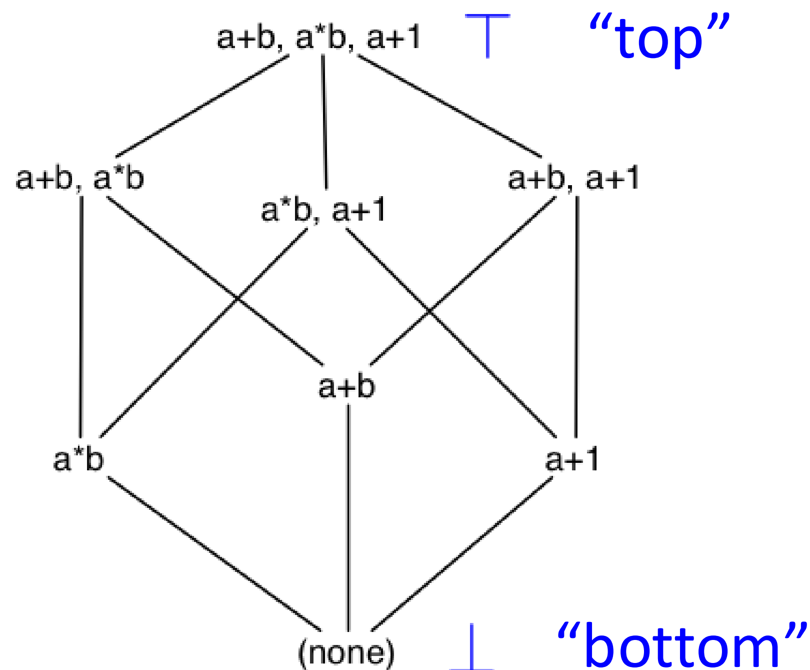
- Why does this terminate?
  - In(s) & Out(s) never shrink
  - Eventually reach fixed point since number of variables is finite.

$$In(s) = \big(Out(s) - kill(s)\big) \cup Gen(s)$$

$$Out(s) = \bigcup_{s' \in \mathrm{succ}(s)} In(s')$$

Data Flow Facts and lattices

- Typically, data flow facts form a lattice
- Example, Available expressions



Lattices

- All our dataflow analyses map program points to elements of a *lattice*.
- A *complete lattice* L = (S, ≤, ∨, ∧, ⊥, T) is formed by:
  - A set S
  - A partial order ≤ between elements of S.
  - A least element ⊥
  - A greatest element T
  - A join operator ∨
  - A meet operator ∧

Least Upper Bound & Join

- If $L = (S, \leq, \vee, \wedge, \perp, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  least upper bound of $\{e_1, e_2\} \equiv e_{lub} = (e_2 \vee e_1) \in S$

Least Upper Bound & Join

- If $L = (S, \leq, \vee, \wedge, \perp, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  least upper bound of $\{e_1, e_2\} \equiv e_{lub} = (e_2 \vee e_1) \in S$
- $\vee$ is the "join" operator
- e_{lub}, the least upper bound, has the properties:
  - $e_1 \leq e_{lub}$ and $e_2 \leq e_{lub}$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e_{lub} \leq e'$

Least Upper Bound & Join

- If $L = (S, \leq, \vee, \wedge, \bot, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  least upper bound of $\{e_1, e_2\} \equiv e_{lub} = (e_2 \vee e_1) \in S$
- $\vee$ is the "join" operator
- e_{lub}, the least upper bound, has the properties:
  - $e_1 \leq e_{lub}$ and $e_2 \leq e_{lub}$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e_{lub} \leq e'$
- least upper bound of $S' \subseteq S$, is pairwise lub of all elements of S'
- For L to be a lattice, for all $S' \subseteq S$, $lub(S') \in S$

Greatest Lower Bound & Meet

- If L = (S, $\leq$, $\vee$, $\wedge$, $\bot$, T) is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  greatest lower bound of $\{e_1, e_2\} \equiv e_{glb} = (e_2 \wedge e_1) \in S$
- $\wedge$ is the "meet" operator
- e_{glb}, the greatest lower bound, has the properties:
  - $e_{glb} \leq e_1$ and $e_{glb} \leq e_2$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e' \leq e_{glb}$

Greatest Lower Bound & Meet

- If $L = (S, \leq, \vee, \wedge, \bot, T)$ is a complete lattice, and $e_1 \in S$ and $e_2 \in S$, then
  greatest lower bound of $\{e_1, e_2\} \equiv e_{glb} = (e_2 \wedge e_1) \in S$
- $\wedge$ is the "meet" operator
- e_{glb}, the greatest lower bound, has the properties:
  - $e_{glb} \leq e_1$ and $e_{glb} \leq e_2$
  - For all $e' \in S$, if $e_1 \leq e'$ and $e_2 \leq e'$, then $e' \leq e_{glb}$
- greatest lower bound of $S' \subseteq S$, is pairwise glb of all elements of S'
- For L to be a lattice, for all $S' \subseteq S$, $glb(S') \in S$

Properties of join (and meet)

- Join is idempotent: $x \vee x = x$
- Join is commutative: $y \vee x = x \vee y$
- Join is associative: $x \vee (y \vee z) = (x \vee y) \vee z$
- Join has a multiplicative one:

  for all $x \in S$, $(\bot \vee x) = x$
- Join has a multiplicative zero:

  for all $x \in S$, $(T \vee x) = T$

Properties of join (and meet)

- Join is idempotent: $x \vee x = x$
- Join is commutative: $y \vee x = x \vee y$
- Join is associative: $x \vee (y \vee z) = (x \vee y) \vee z$
- Join has a multiplicative one:

  for all $x \in S$, $(\bot \vee x) = x$
- Join has a multiplicative zero:

  for all $x \in S$, $(T \vee x) = T$

- Similarly for meet, but:
  - multiplicative one is T, i.e., for all $x \in S$, $(T \wedge x) = x$
  - multiplicative zero is ⊥, i.e., for all $x \in S$, $(\bot \wedge x) = \bot$

Semilattices

- Notice the dataflow analysis we looked at have either the join or meet operator, e.g.,
  - available expressions uses meet: $\wedge$ is intersection
  - liveness uses join: $\vee$ is union
- If only one of meet or join are defined, we call it a semilattice.

Partial Order

- A partial order is a pair (S, ≤) such that:
  - ≤ $\subseteq$ S×S
  - ≤ is reflexive, i.e.,

    x ≤ x
  - ≤ is anti-symmetric, i.e.,

    x ≤ y and y ≤ x implies x=y
  - ≤ is transitive, i.e.,

    x ≤ y and x ≤ z implies x ≤z

Partial Order, $\vee$, $\wedge$, and Semi-Lattice

- Join, least upper bound, on a semi-lattice defines a partial order:

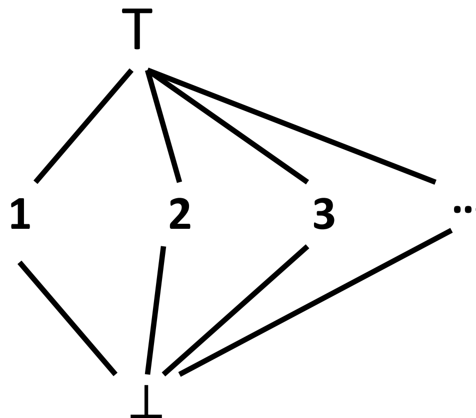
  $$x \leq y \text{ iff } x \vee y = y$$

- Meet, greatest lower bound, on a semi-lattice defines a partial order:

  $$x \leq y \text{ iff } x \wedge y = x$$

Useful Lattices

- $(2^S, \subseteq)$ forms a lattice for any set S.
  - 2^S is the power set of S (set of all subsets)
- If $(S, \leq)$ is a lattice, so is $(S, \geq)$
  - i.e., lattices can be flipped
- A lattice for constant propagation

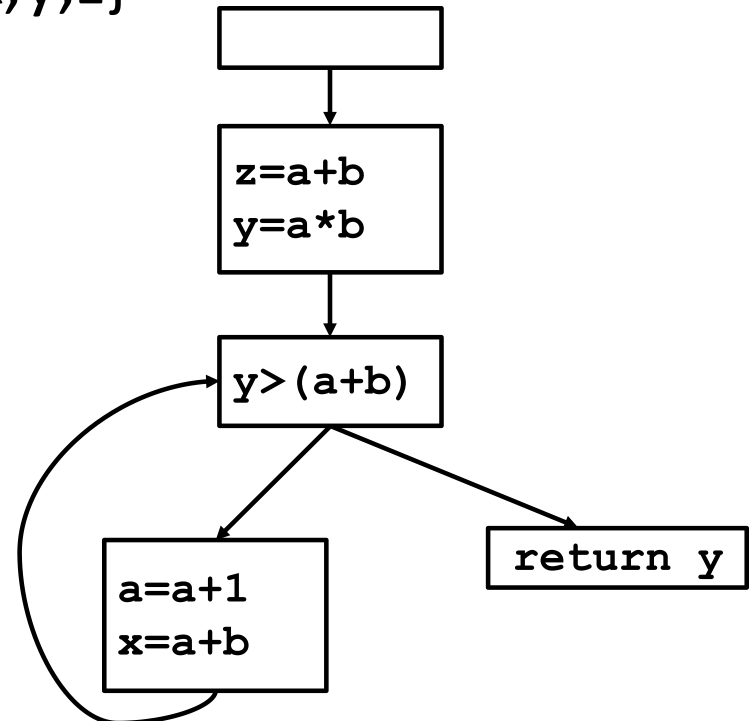


Semilattice of Liveness

- $L=(2\text{^}\{a,b,x,y,z\}, \subseteq, \cup, \{\}, \{a,b,x,y,z\})$
  - Only define Join, $\cup$
  - Least Element, $\perp$, $\{\}$
  - Greatest Element, T, $\{a,b,x,y,z\}$
  - $x \leq y$ means $x \subseteq y$

- more generally,

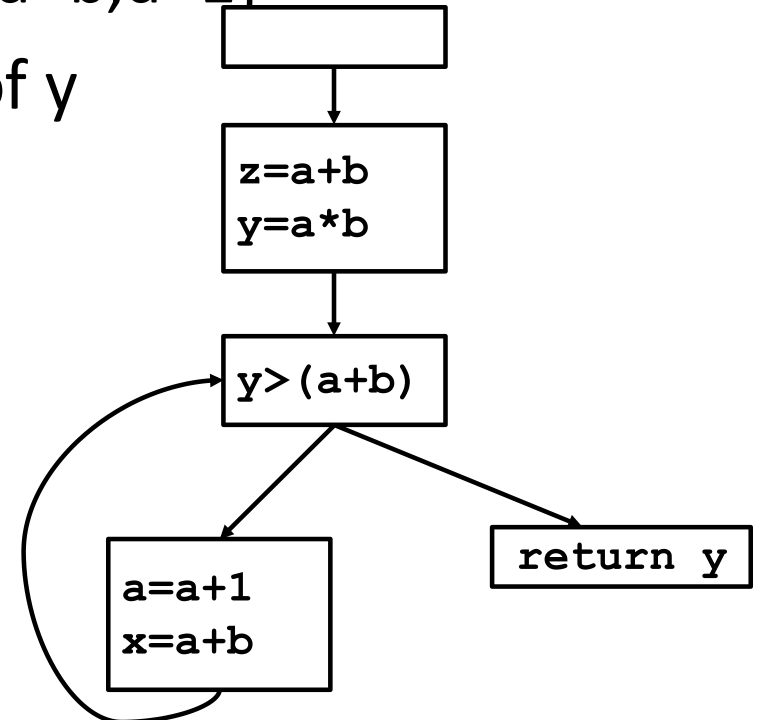
  $L=(2^S, \subseteq, \cup, \{\}, S)$



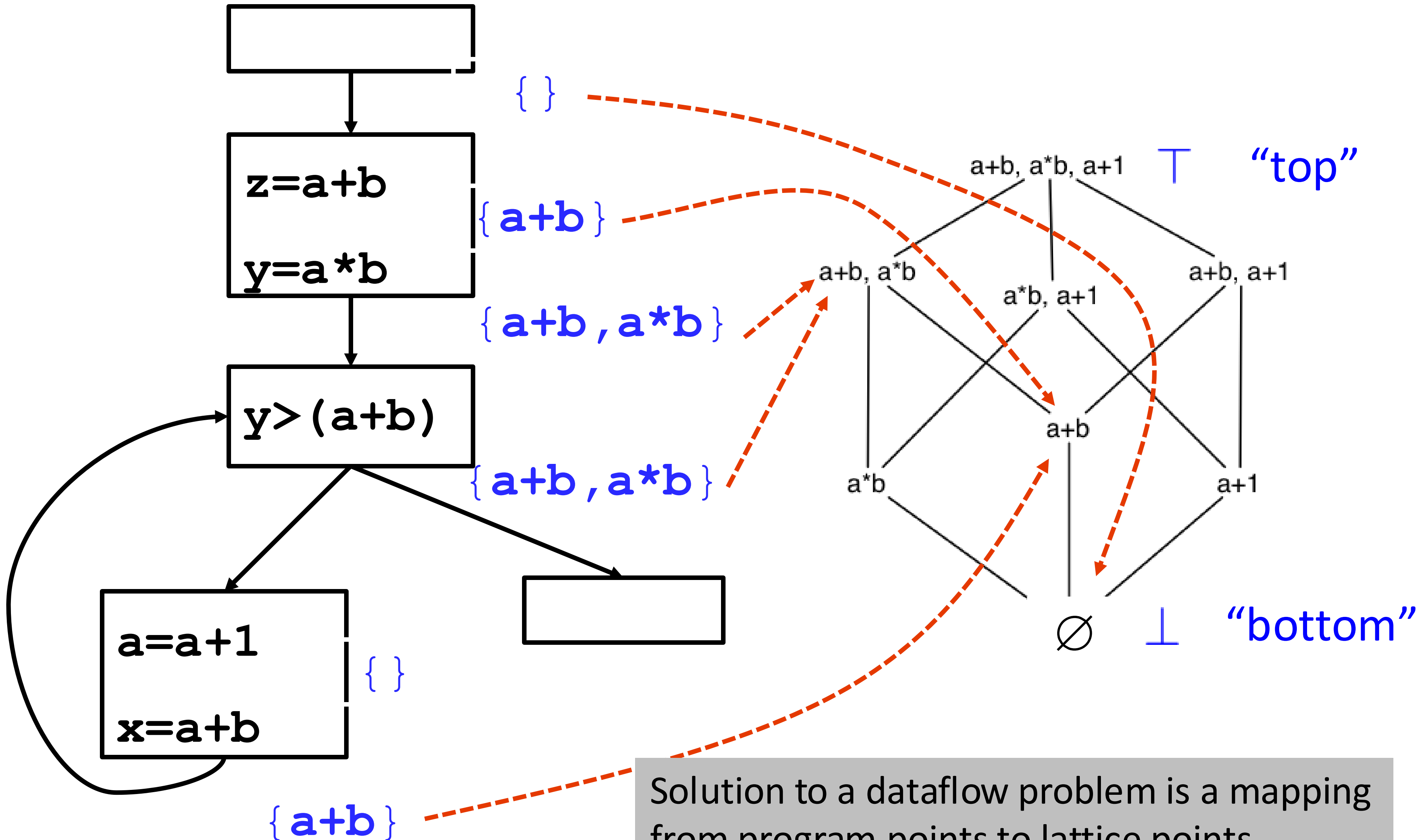
Semilattice of Available Expressions

- $L=(\{a+b,a*b,a+1\},\supseteq,\cap,\{\}, \{a+b,a*b,a+1\})$
  - Only define Meet, $\cap$
  - Least Element, $\perp$, {}
  - Greatest Element, T, {a+b,a*b,a+1}
  - x ≤ y means x is superset of y
- In general:

  $L=(2^S, \supseteq,\cap, \{\},S)$



Available Expressions



Solution to a dataflow problem is a mapping from program points to lattice points

Monotonicity & Termination

- A function f on a partial order is monotonic if

  $x \leq y$ implies $f(x) \leq f(y)$
- We call f a transfer function

Monotonicity for Available Expressions

- A function f on a partial order is monotonic if

  $x \leq y$ implies $f(x) \leq f(y)$

For x= a⊕b:

Gen = {a⊕b}

Kill = {All expressions using x}

$$In(s) = \bigcap_{s' \in \text{pred}(s)} Out(s)$$

$$Out(s) = Gen(s) \cup \left(In(s) - Kill(s)\right)$$

$$Out(s) = f_s\left(\bigcap_{s' \in \text{pred}(s)} Out(s')\right)$$

Termination

- Algorithm terminates because:
  - The lattice has finite height
  - The operations to compute In and Out are monotonic
  - On every iteration either:
    - W gets smaller, or
    - out(s) decreases for some s, i.e., we move down lattice

```
Initialize: in[s] = out[s] = Universe
Initialize: in[entry] = ∅
Work queue, W = all Blocks
while (|W| != 0)  {
    remove s from W
    temp = out[s]
    compute In[s]
    compute Out[s]
    if (temp != out[s]) W = W ∪ succ(s)
}
```

Fixpoints

- We always start with Top
  - Every expression is available,
    no definitions reach this point
  - Most optimistic assumption
  - Strongest possible hypothesis
    (i.e., true of fewest number of states)
- Revise as we encounter contradictions
  - Always move down in the lattice (with meet)
- Result: A greatest fixpoint

Distributive Data Flow Problems

- By monotonicity, we also have

$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- A function f is distributive if

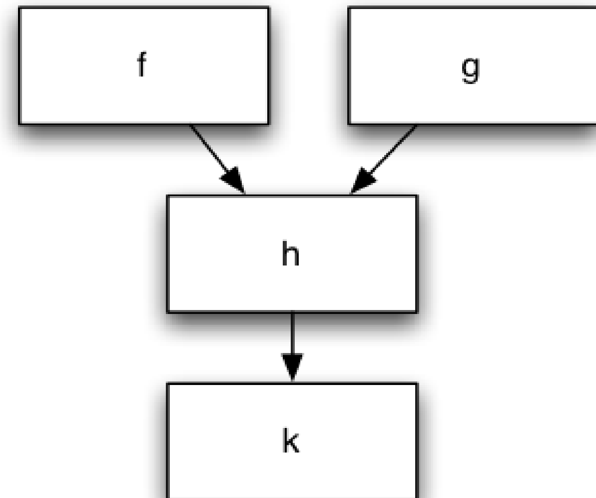
$$f(x \sqcap y) = f(x) \sqcap f(y)$$

Does meet over all paths == greatest lower bound?

Benefit of Distributivity

- Joins lose no information

$$k(h(f(\top) \sqcap g(\top))) =$$
$$k(h(f(\top)) \sqcap h(g(\top))) =$$
$$k(h(f(\top))) \sqcap k(h(g(\top)))$$



Accuracy of Data Flow Analysis

- Ideally, we would like to compute the meet over all paths (MOP) solution:
  - Let f_s be the transfer function for statement s
  - If p is a path $\{s_1, ..., s_n\}$, let $f_p = f_n;...;f_1$
  - Let path(s) be the set of paths from the entry to s

$$\mathrm{MOP}(s) = \sqcap_{p \in \mathrm{path}(s)} f_p(\top)$$

- If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution

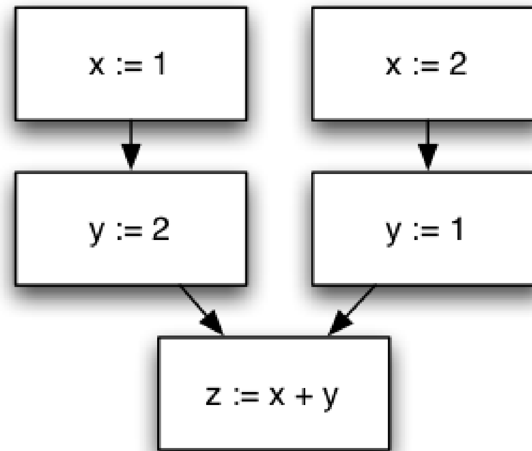
What Problems are Distributive?

- Analyses of *how* the program computes
  - Live variables
  - Available expressions
  - Reaching definitions
  - Very busy expressions

- All Gen/Kill problems are distributive

A Non-Distributive Example

- Constant propagation



- In general, analysis of *what* the program computes is not distributive

Flow-Sensitivity

- Data flow analysis is *flow-sensitive*
  - The order of statements is taken into account
  - i.e., we keep track of facts per program point

- Alternative: *Flow-insensitive* analysis
  - Analysis the same regardless of statement order
  - Standard example: types

Terminology Review

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Distributive vs. Non-distributive