Data-Flow Analysis

Overview
Available Expressions Problem and Iterative Data-Flow Formulation

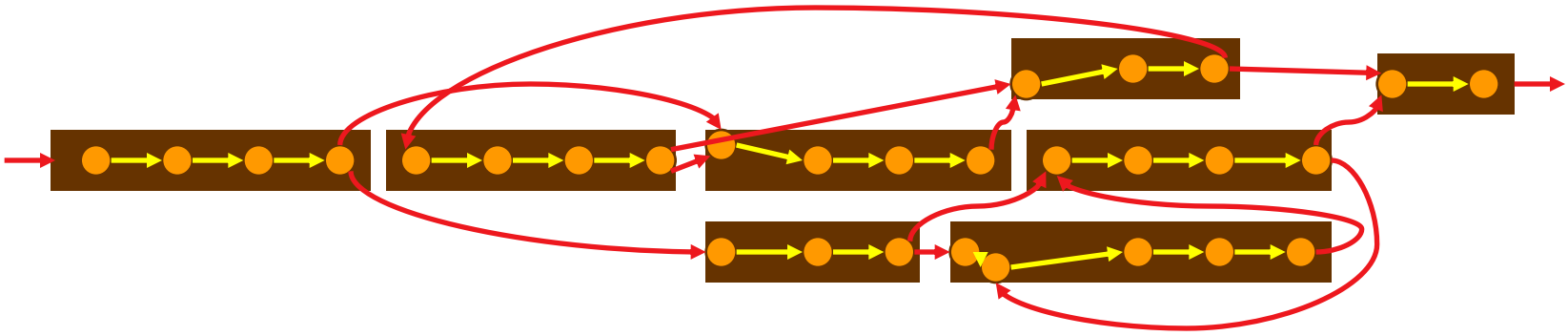
Pedro C. Diniz
pedi@fe.up.pt

Outline

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

Control Flow of a Program

- Forms a Graph



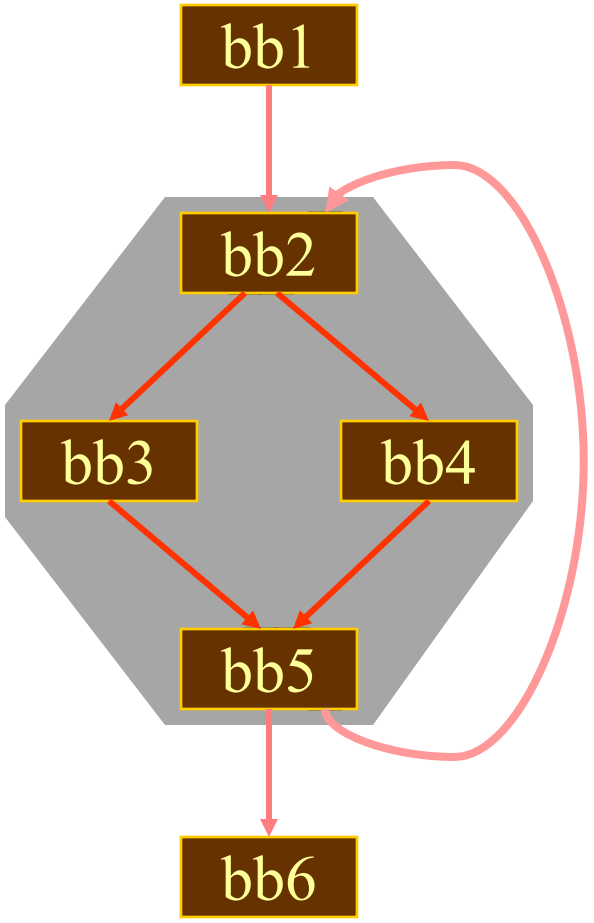
- A Very Large Graph
- Create Basic Blocks
- A Control-Flow Graph (CFG) connects basic blocks

Control Flow Graph (CFG)

- Control-Flow Graph G = <N, E>
- Nodes(N): Basic Blocks
- Edges(E): (x,y) ∈ E iff first instruction in the basic block ***y*** follows the last instruction in the basic block ***x***

Identifying Recursive Structures Loops

- Identify Back Edges
- Find the nodes and edges in the loop given by the back edge
- Other than the Back Edge
  - Incoming edges only to the basic block with the back edge head
  - one outgoing edge from the basic block with the tail of the back edge
- How do I find the Back Edges?

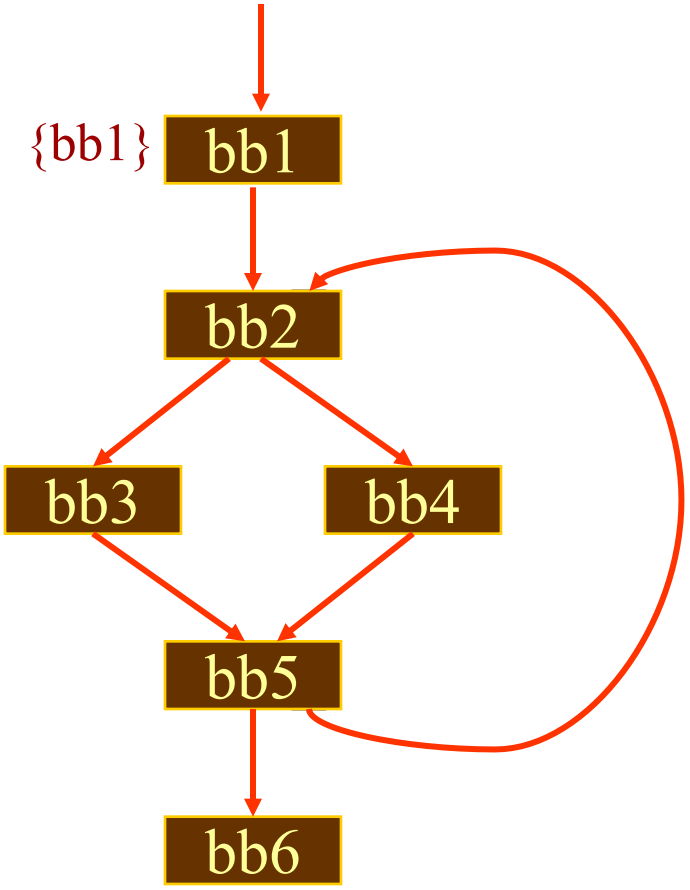


Computing Dominators

- Algorithm
  - Make dominator set of the entry node as itself
  - Make dominator set of the remainder nodes include all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node as the intersection of the dominator sets of the predecessor nodes and the current node
  - Repeat until no change

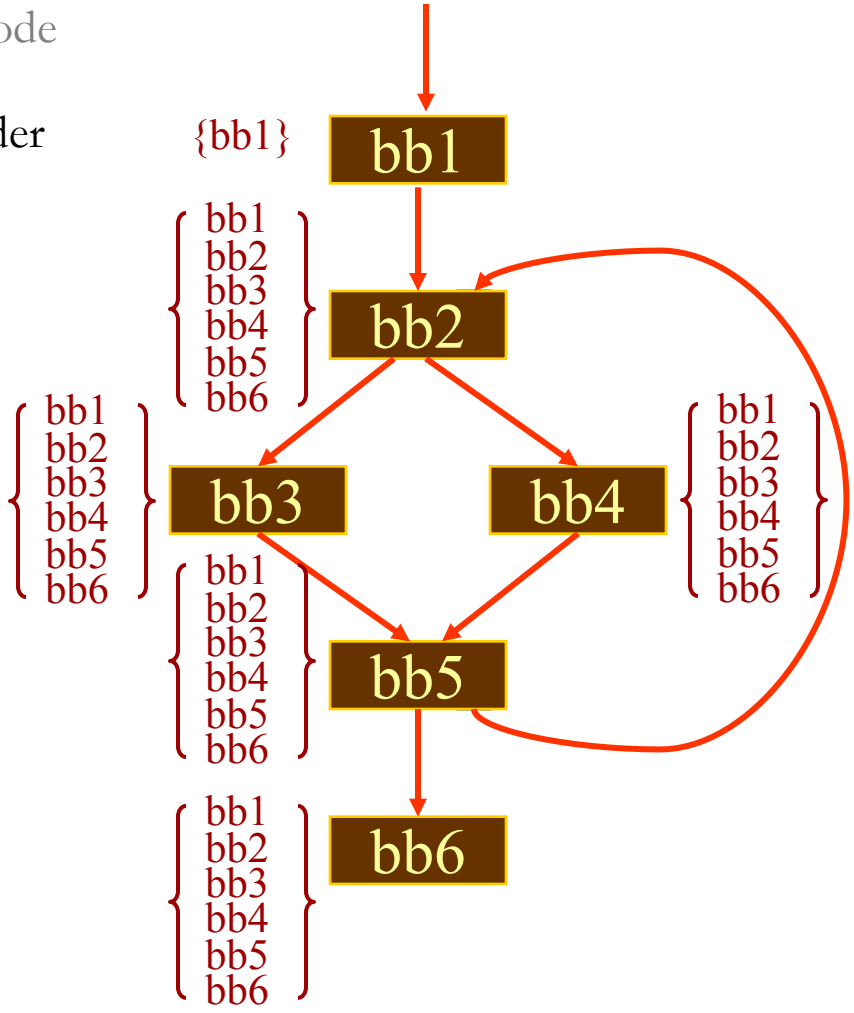
Computing Dominators

- Algorithm
  - Make dominator set of the entry node as itself
  - Make dominator set of the remainder nodes to include all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node as the intersection of the dominator sets of the predecessor nodes and the current node
  - Repeat until no change



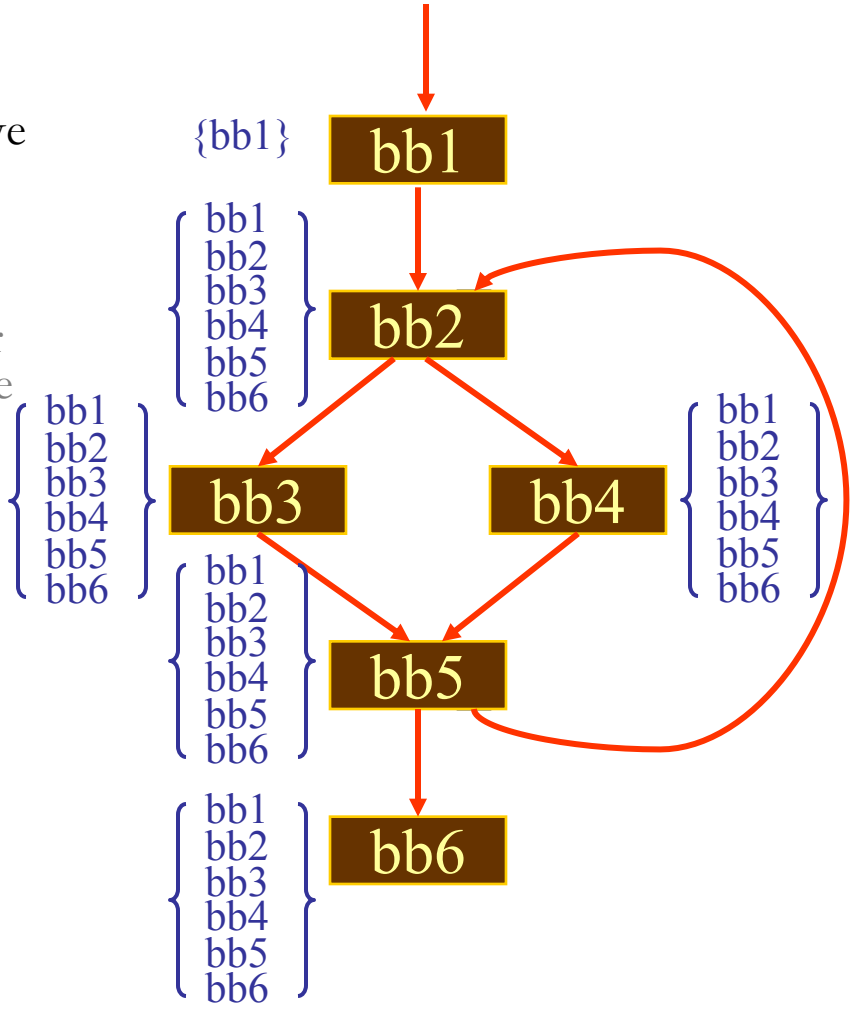
Computing Dominators

- Algorithm
  - Make dominator set of the entry node as itself
  - Make dominator set of the remainder nodes to include all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node as the intersection of the dominator sets of the predecessor nodes and the current node
  - Repeat until no change



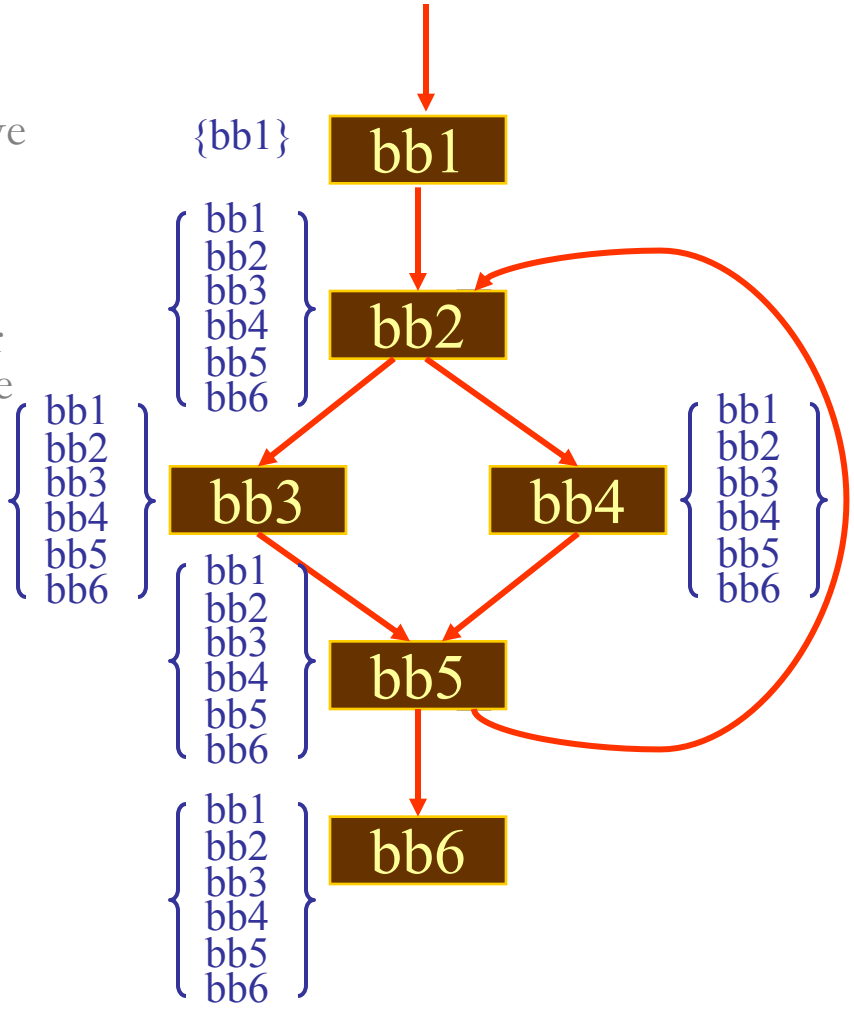
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



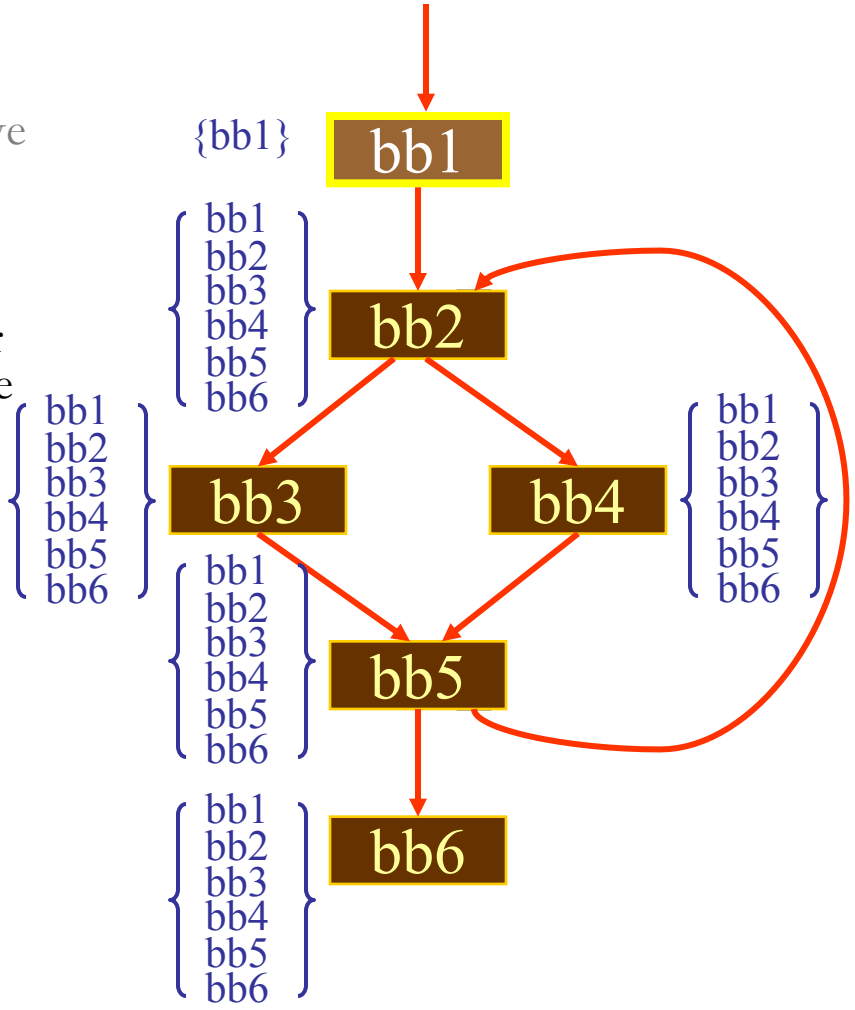
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



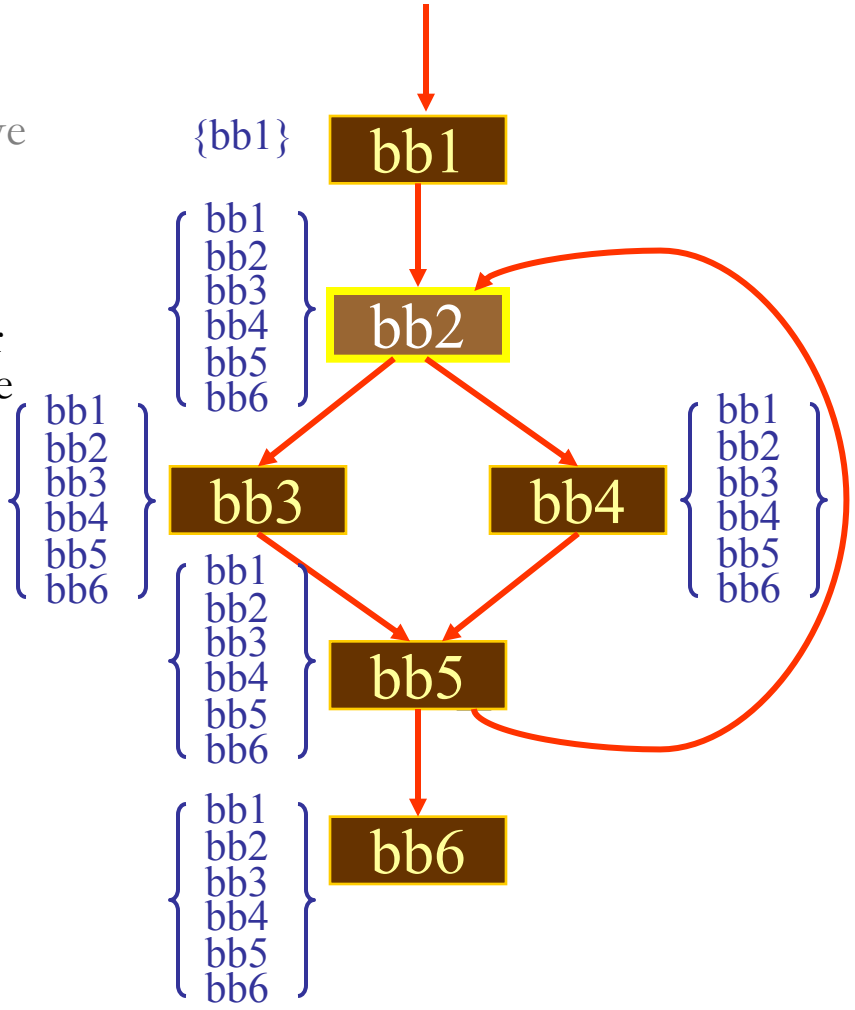
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



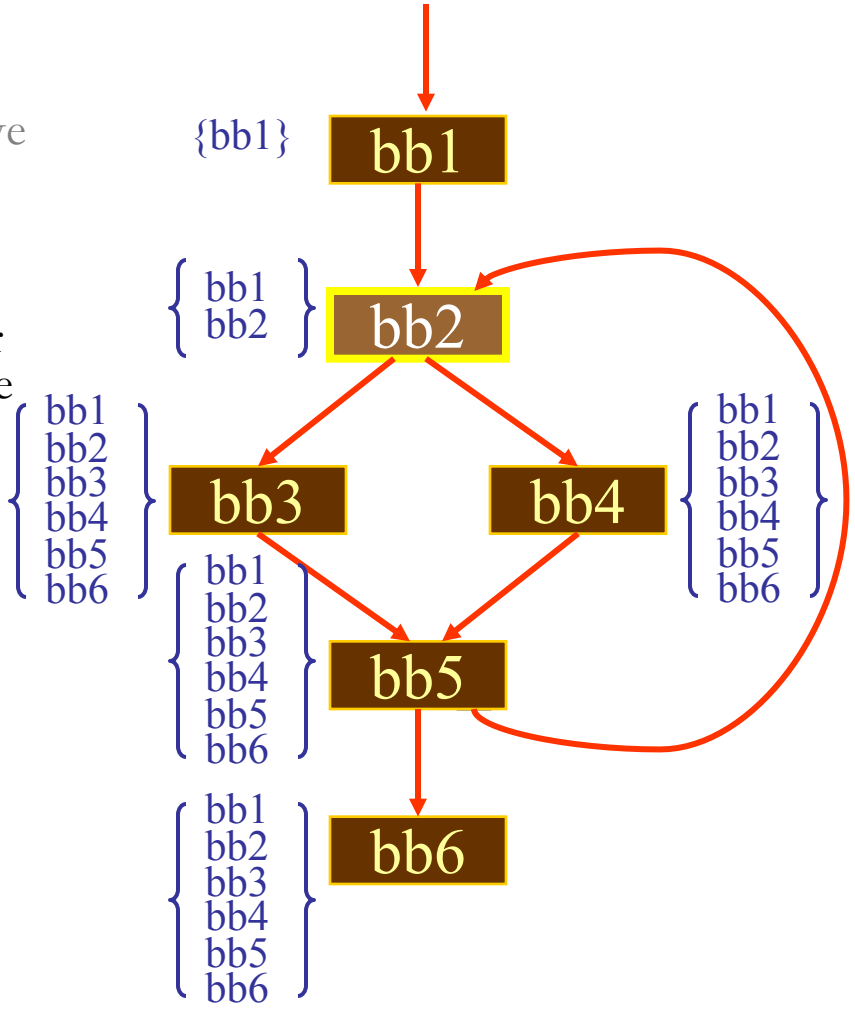
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



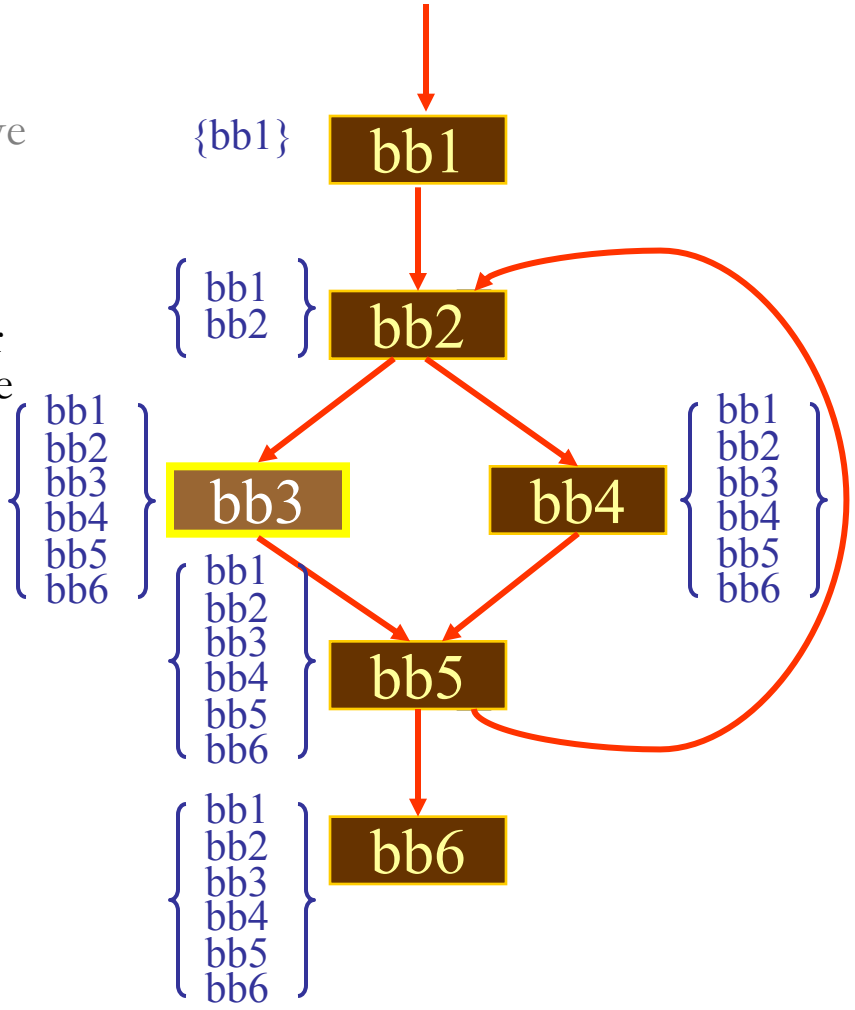
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



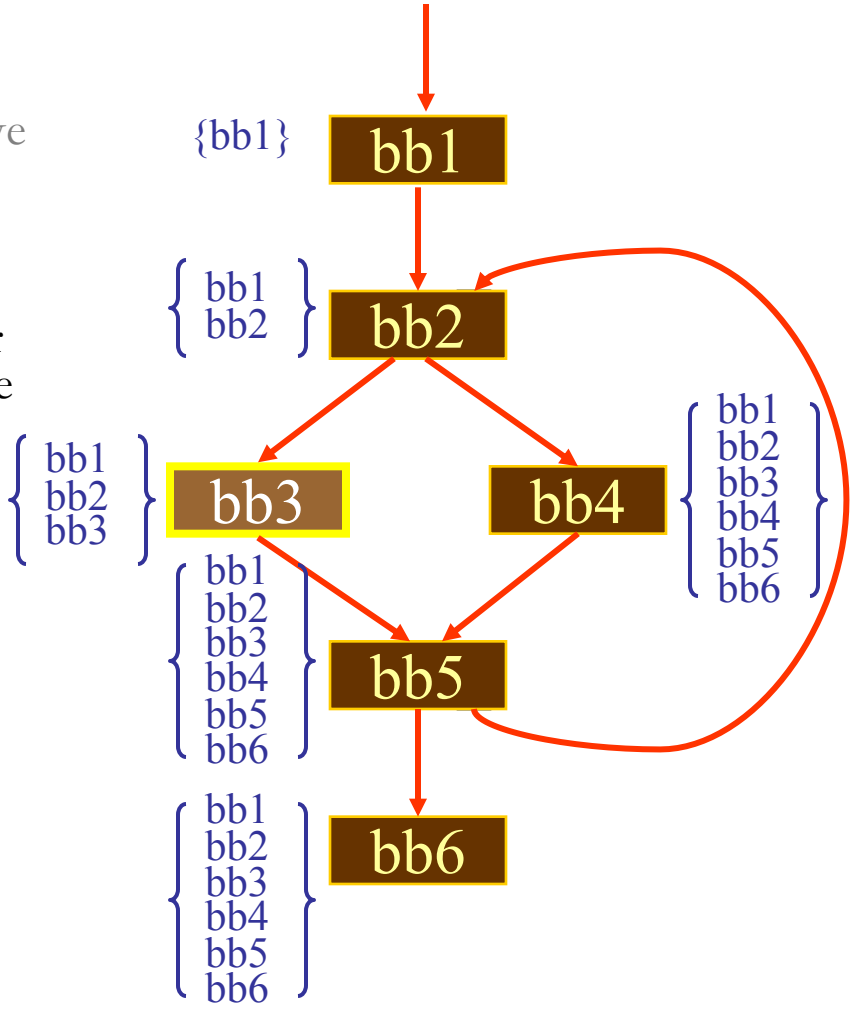
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



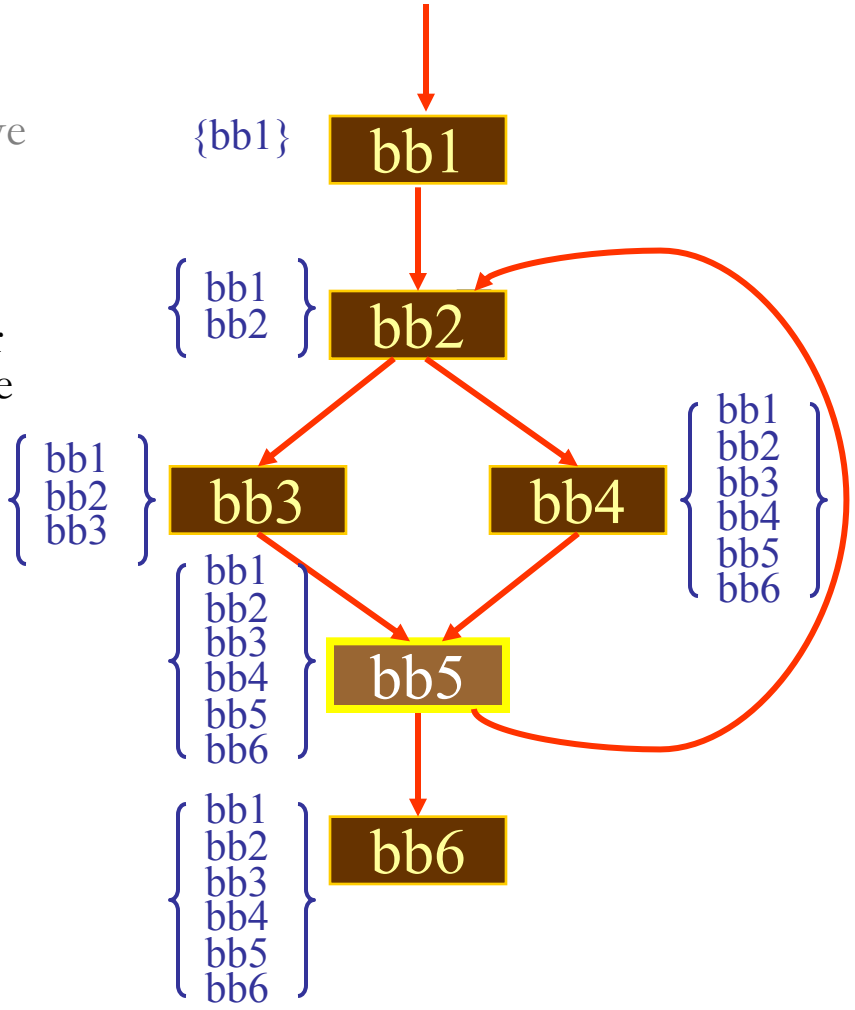
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



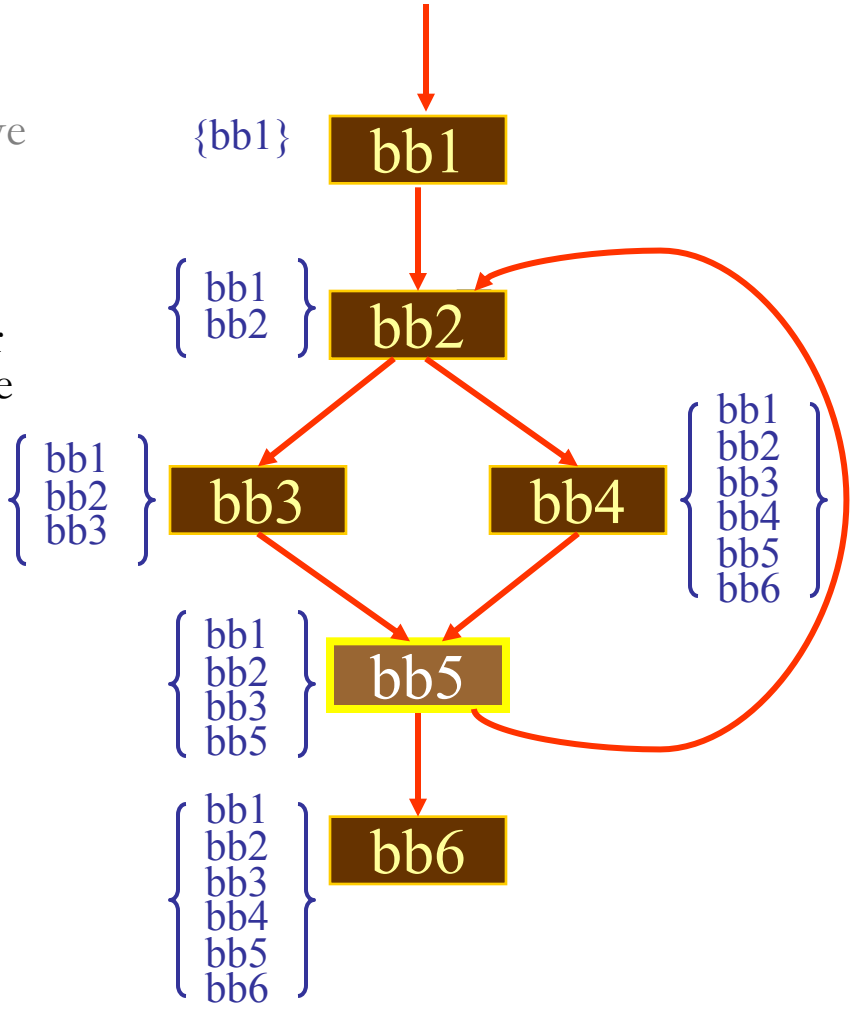
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



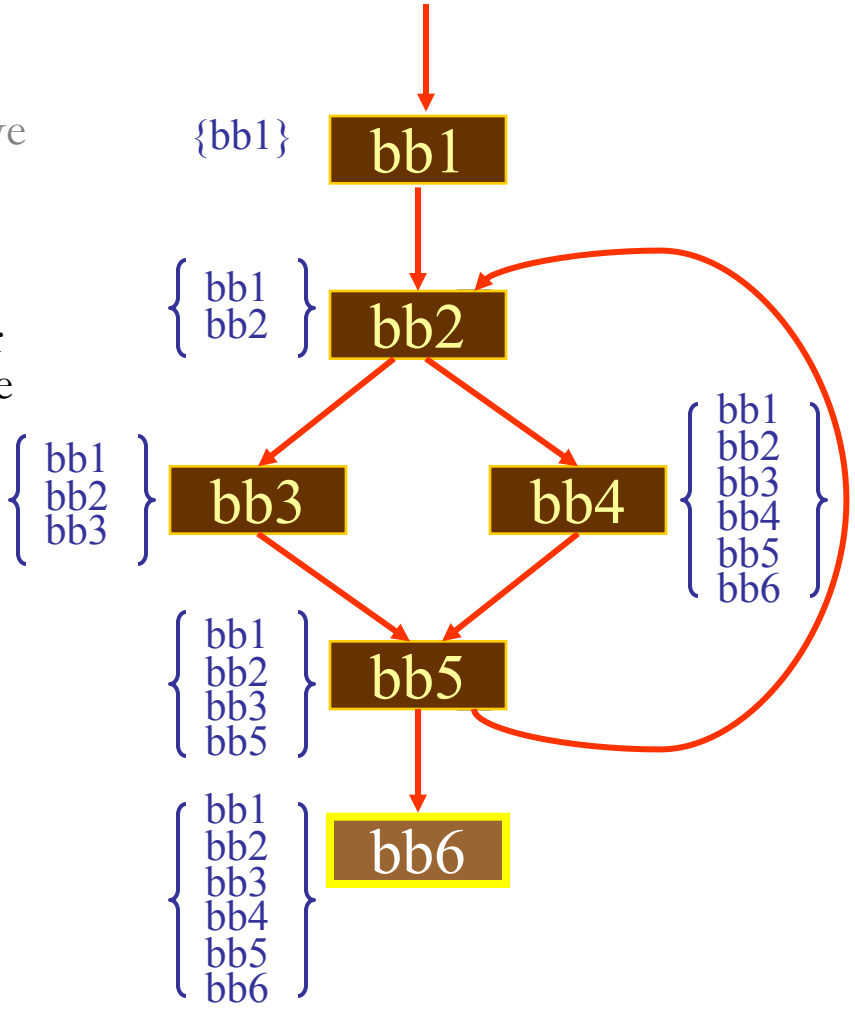
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



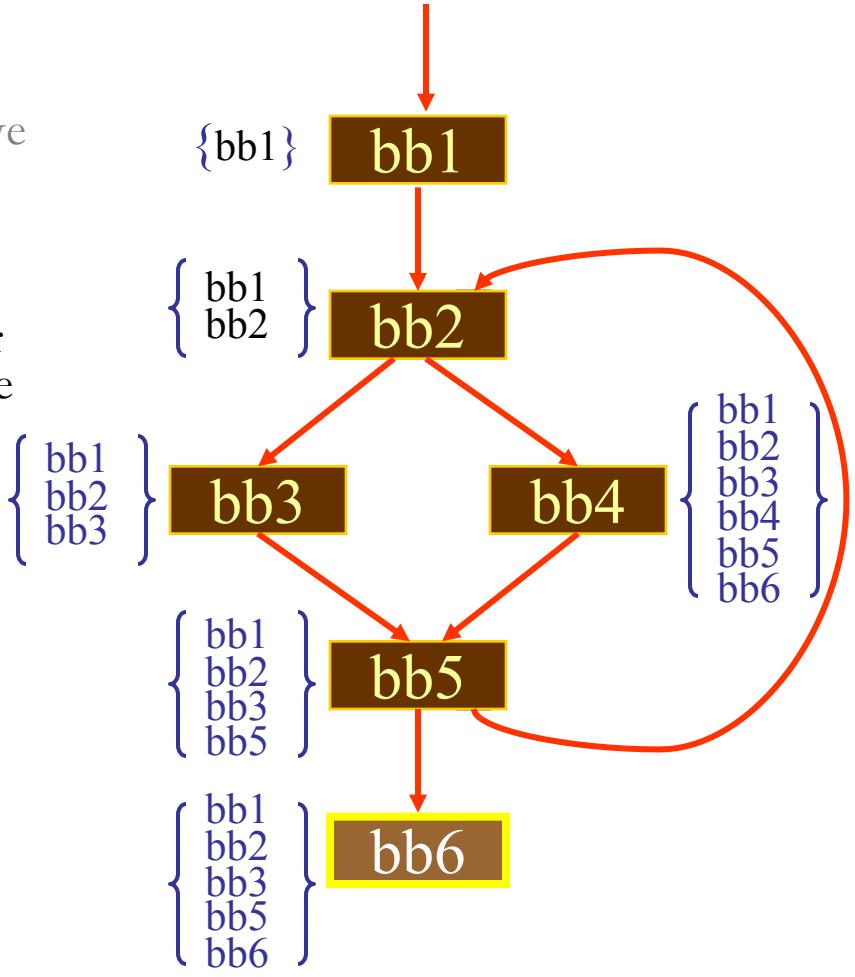
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



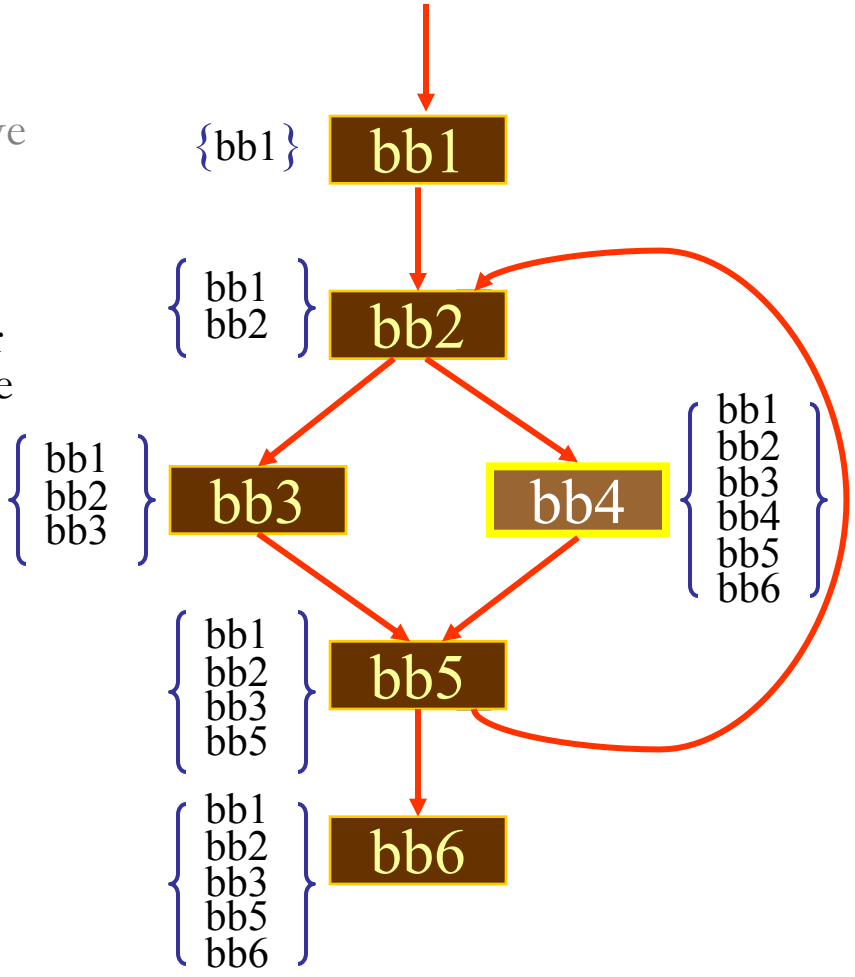
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



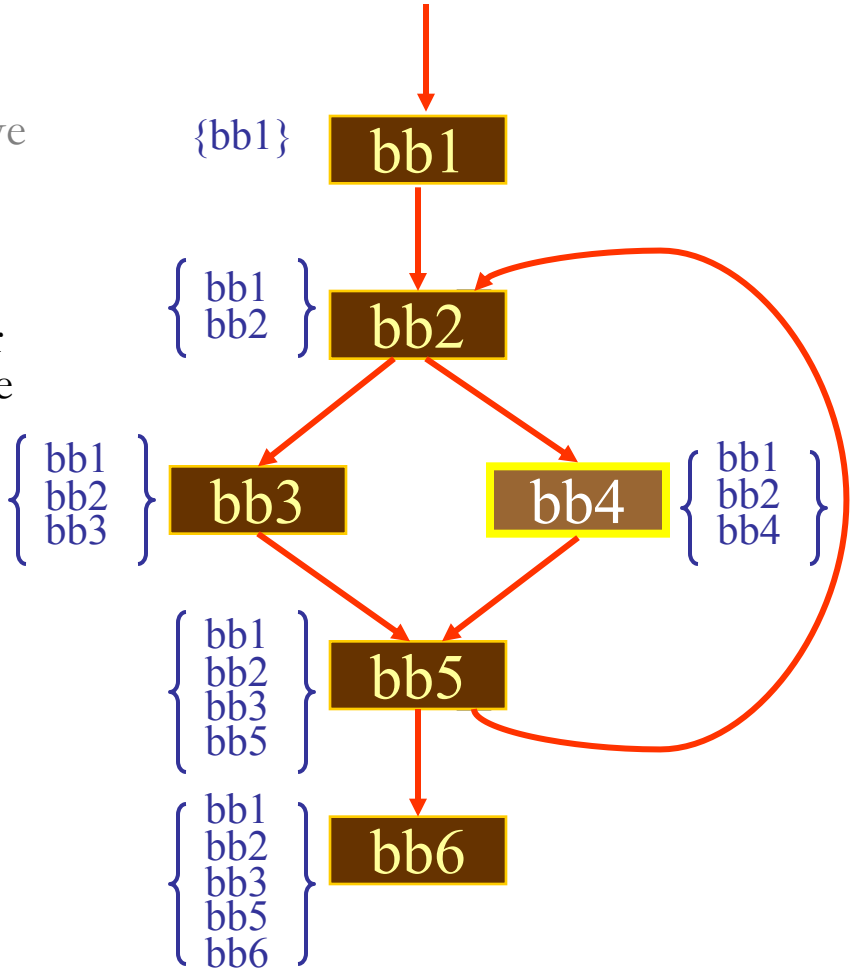
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



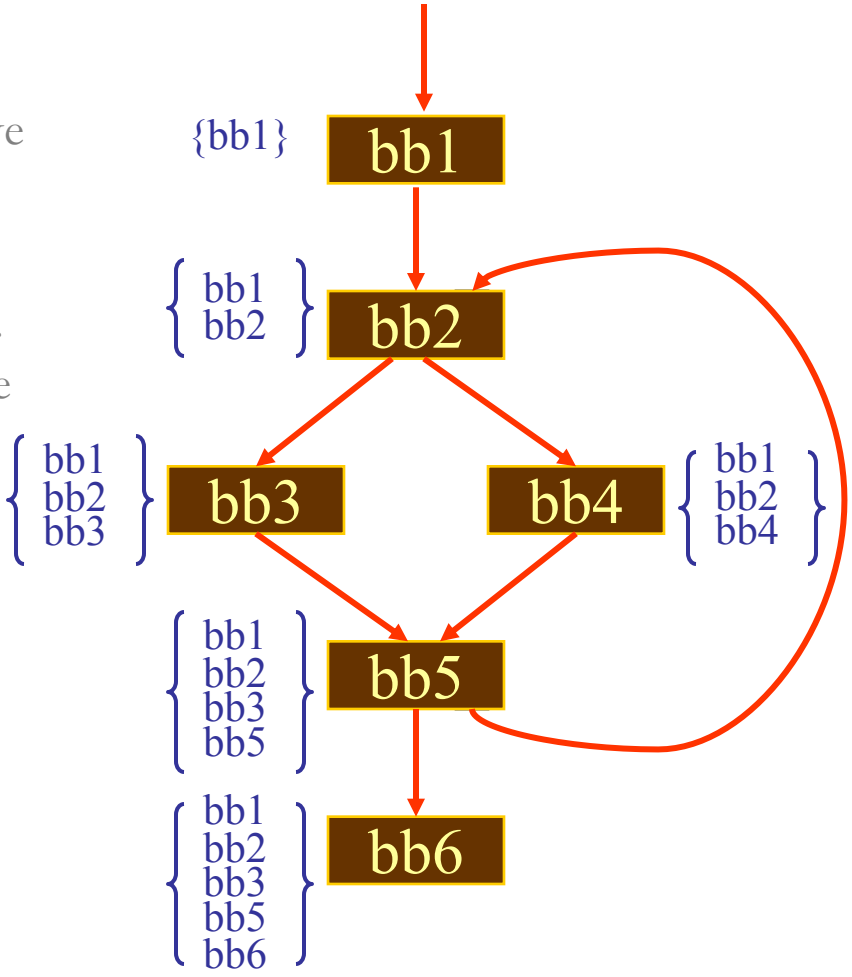
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



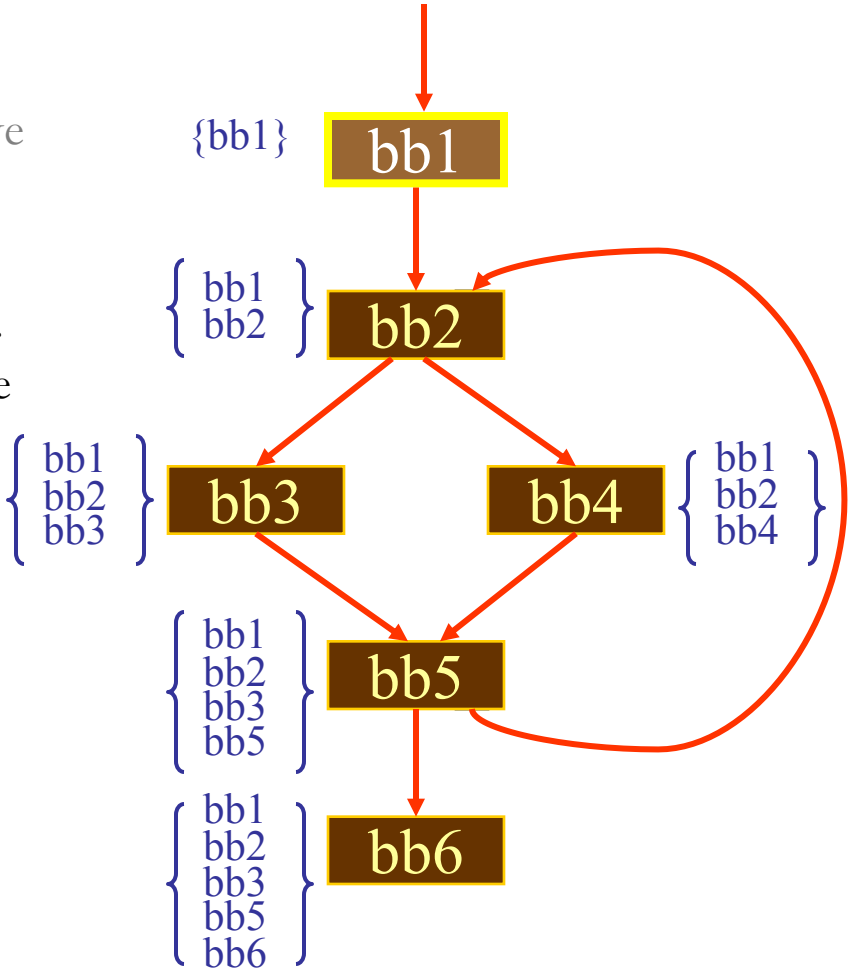
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



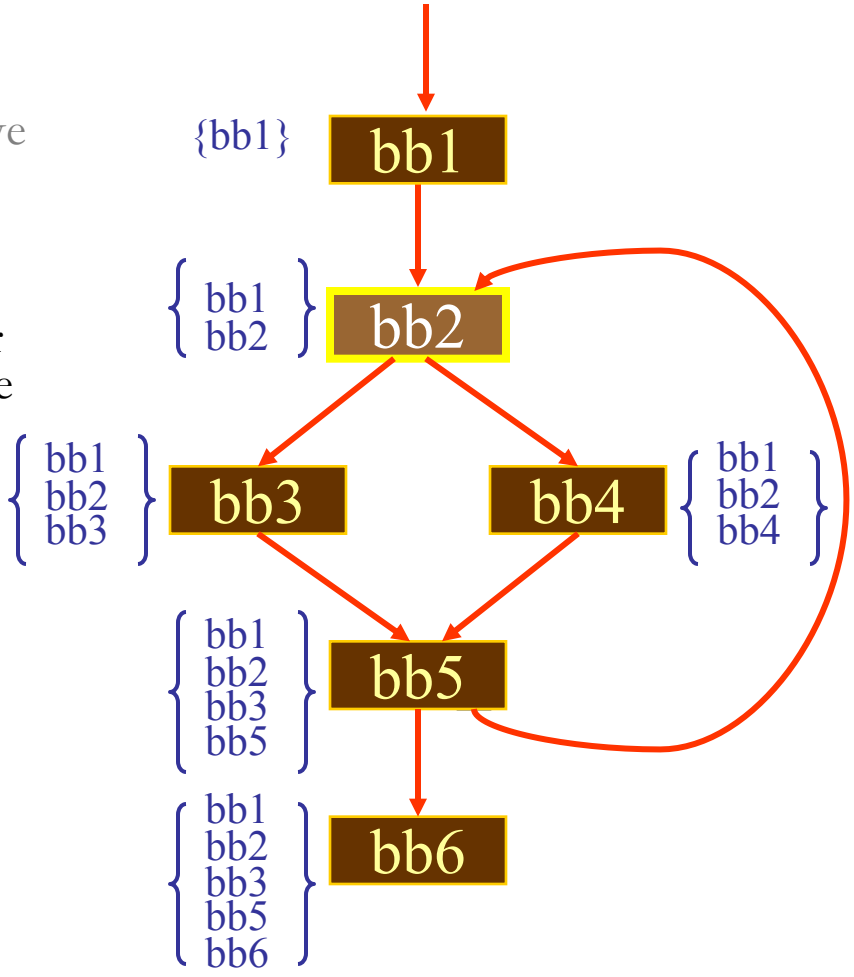
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



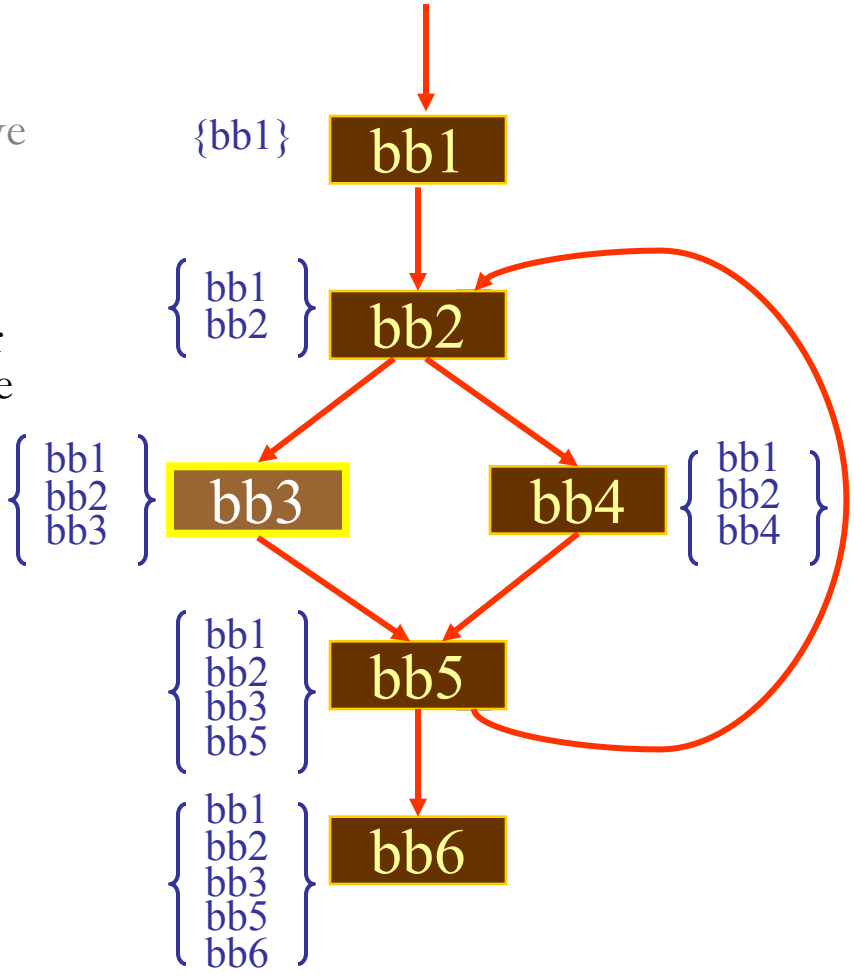
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



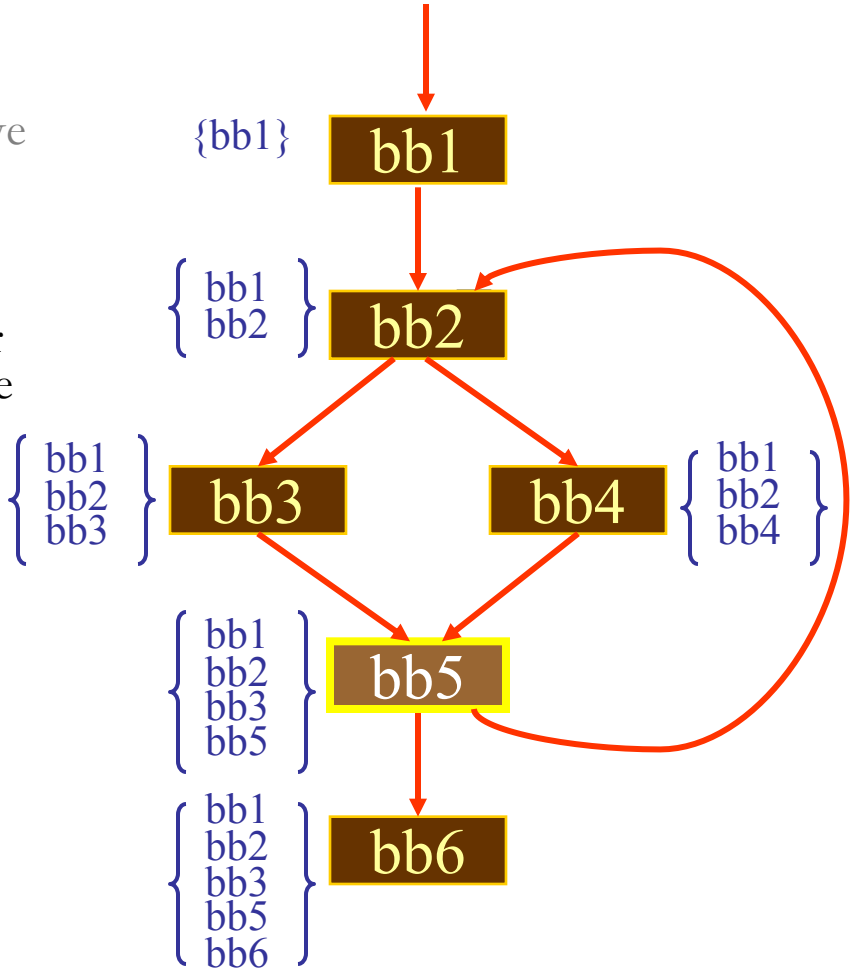
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



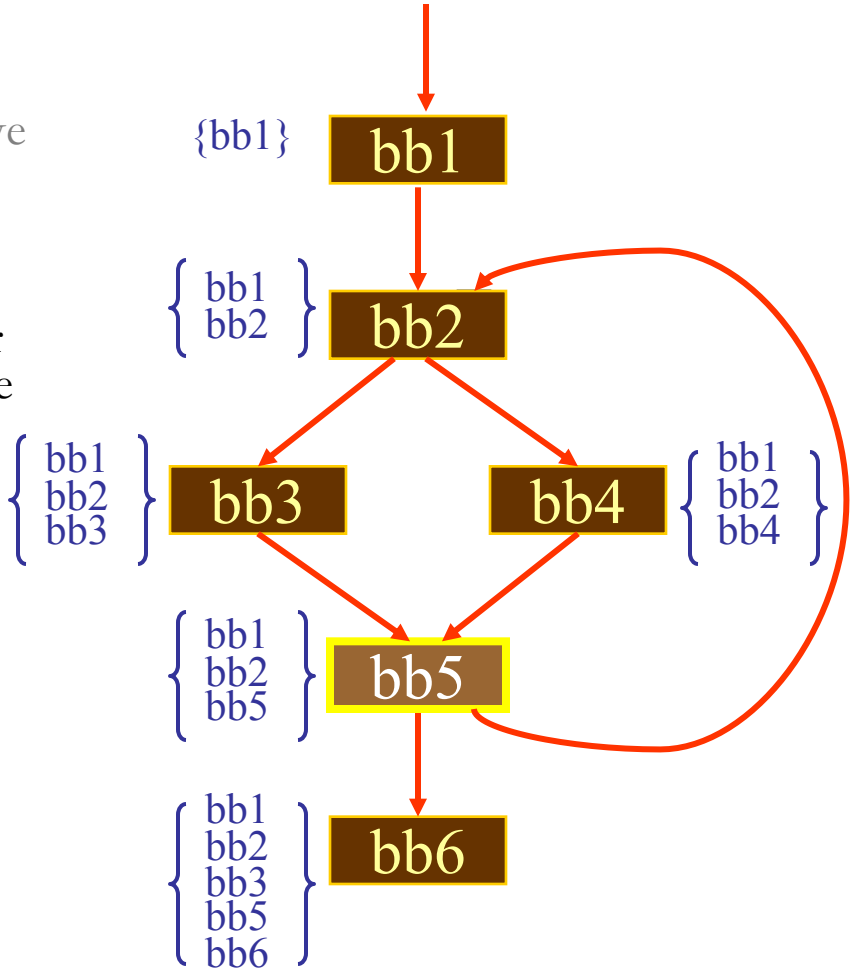
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



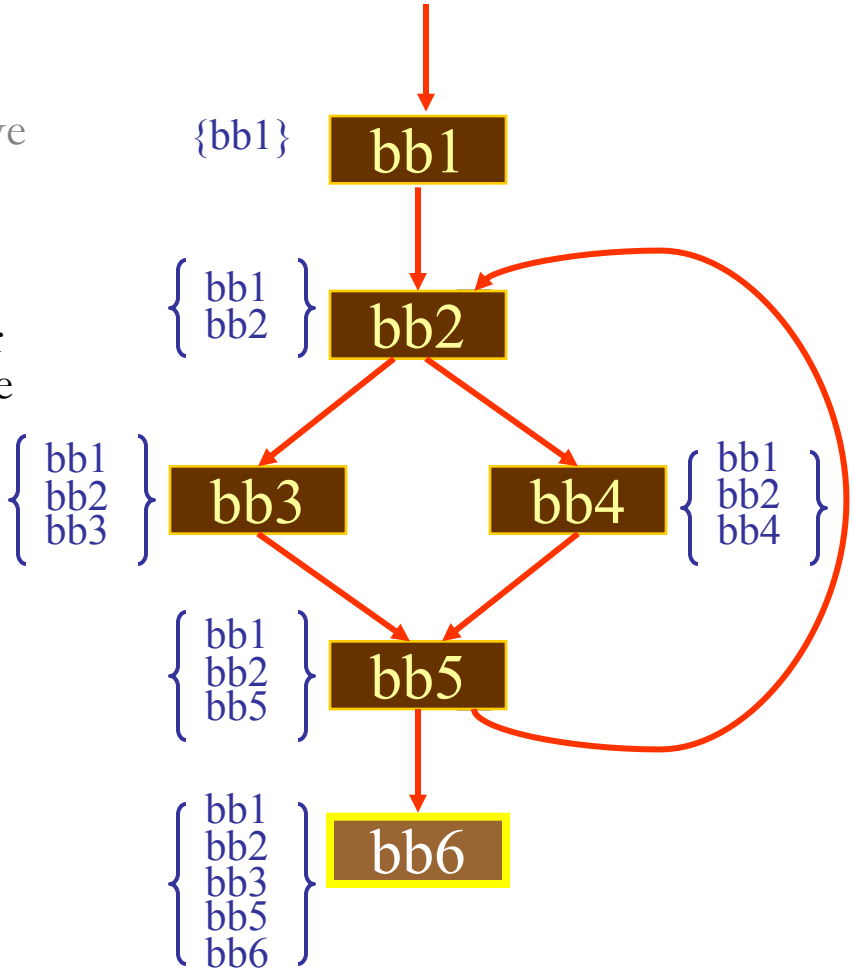
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



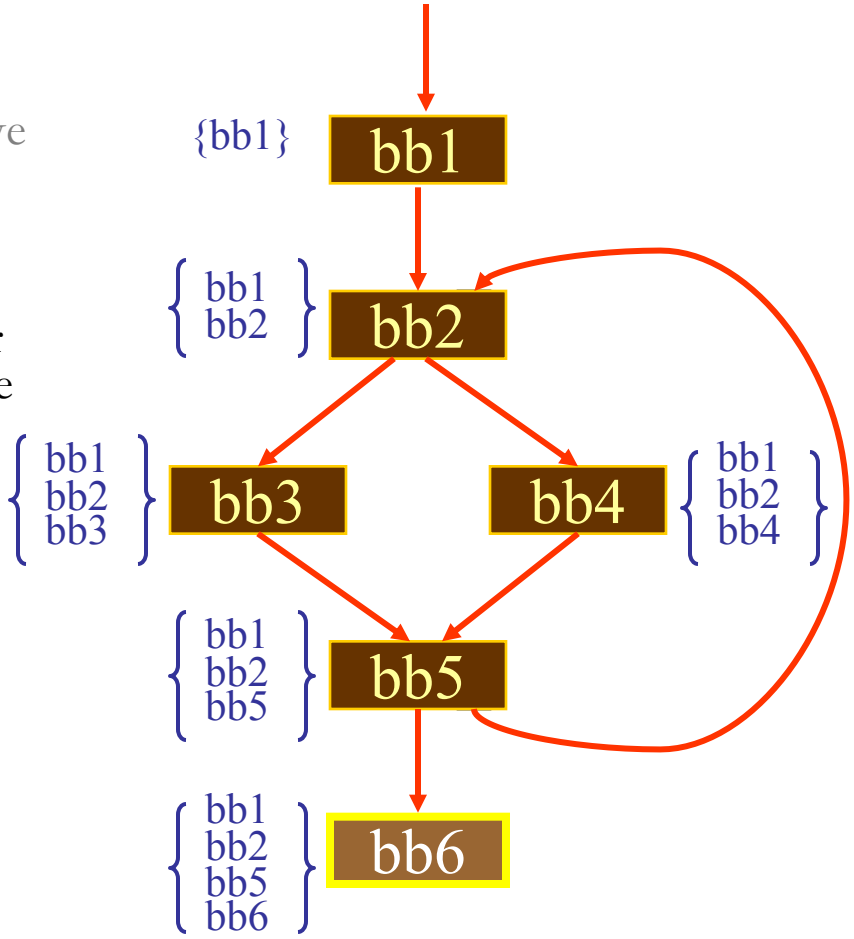
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



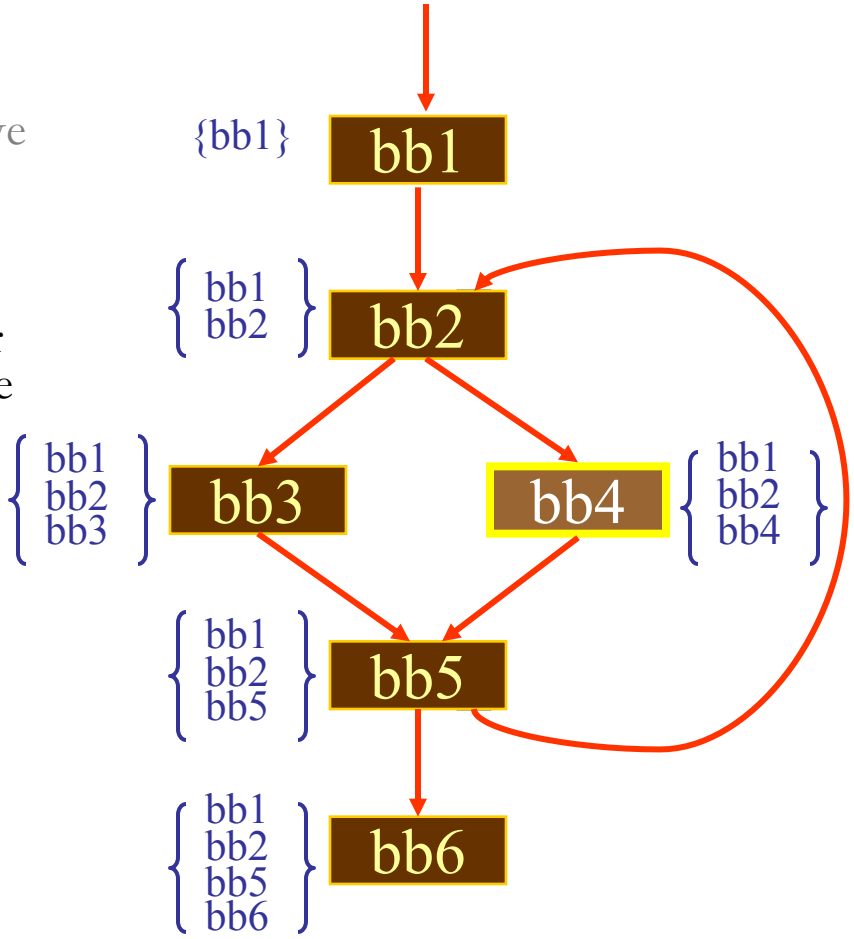
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



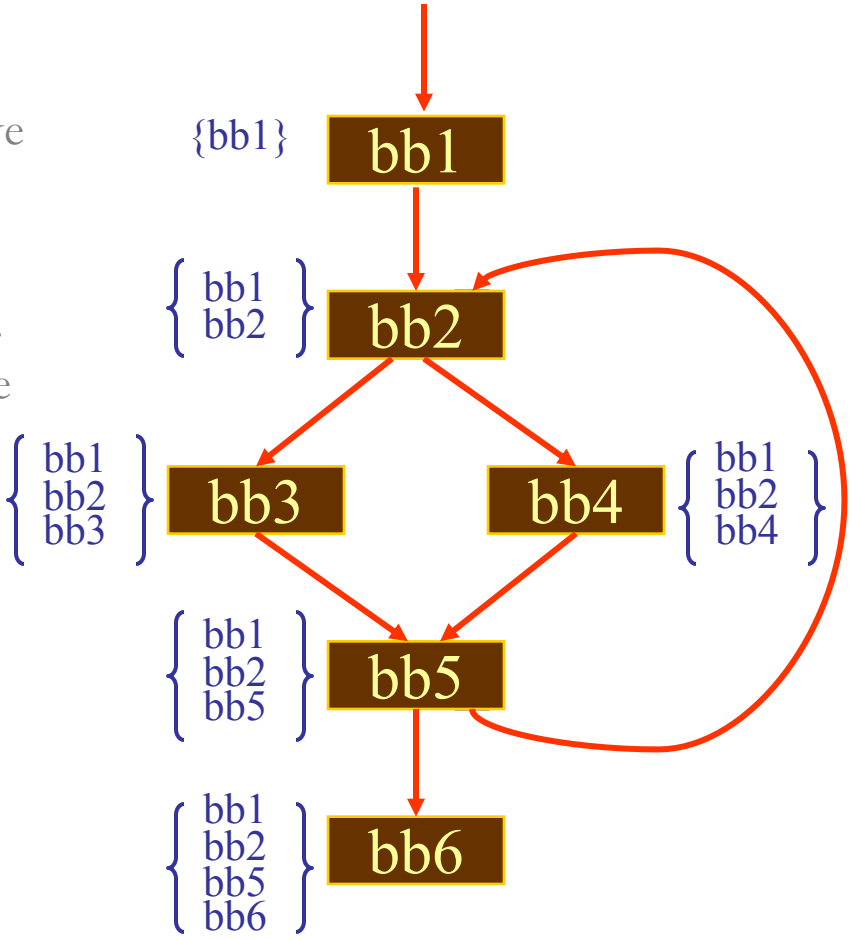
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



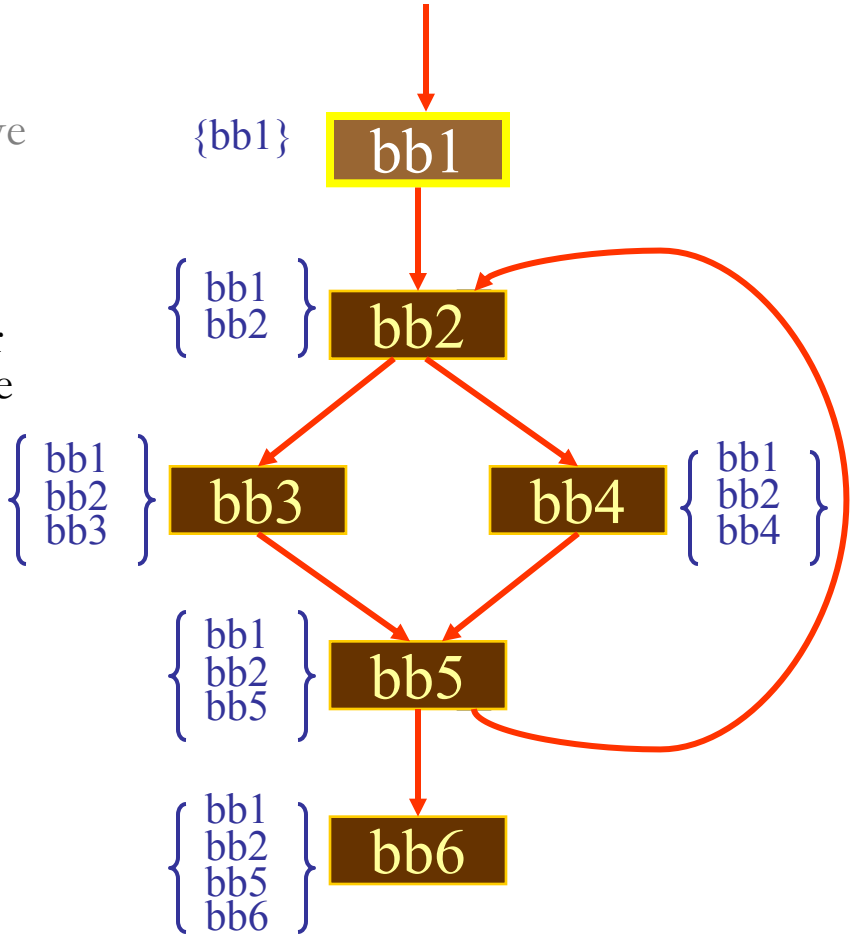
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



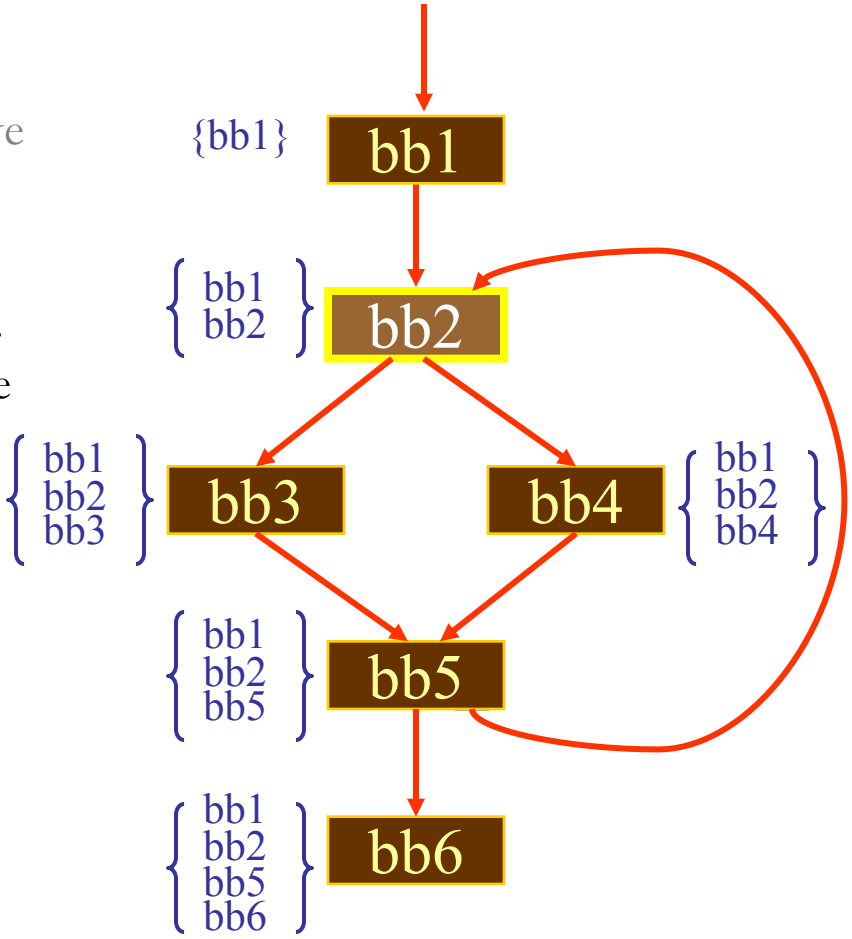
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



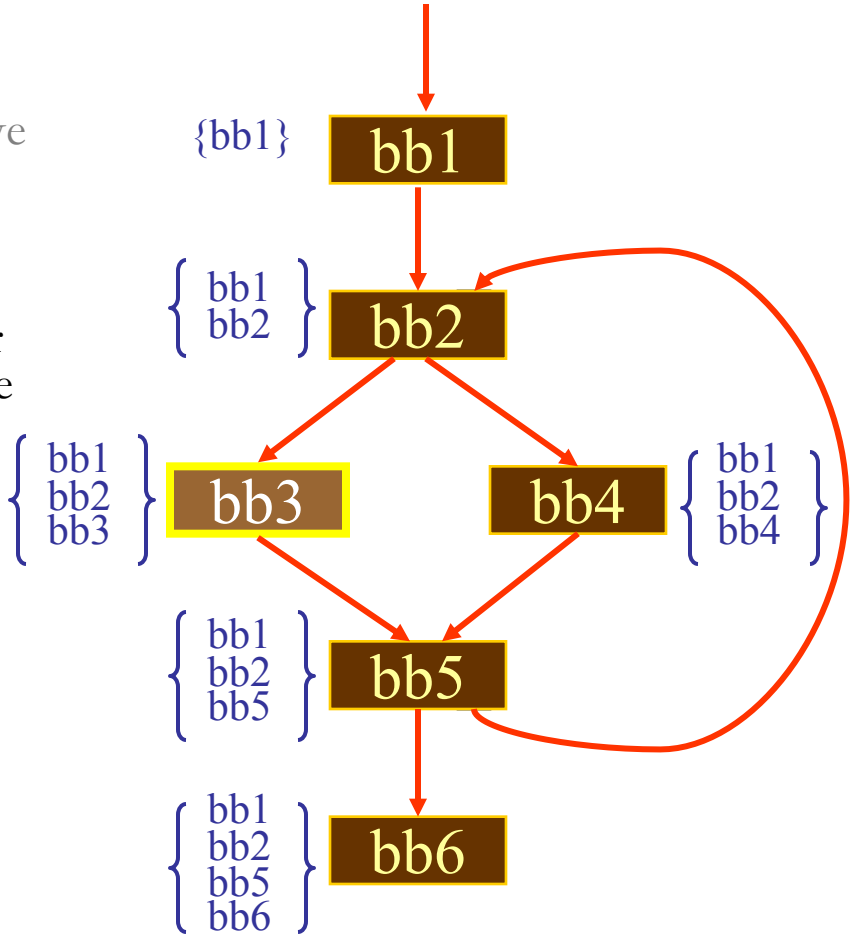
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



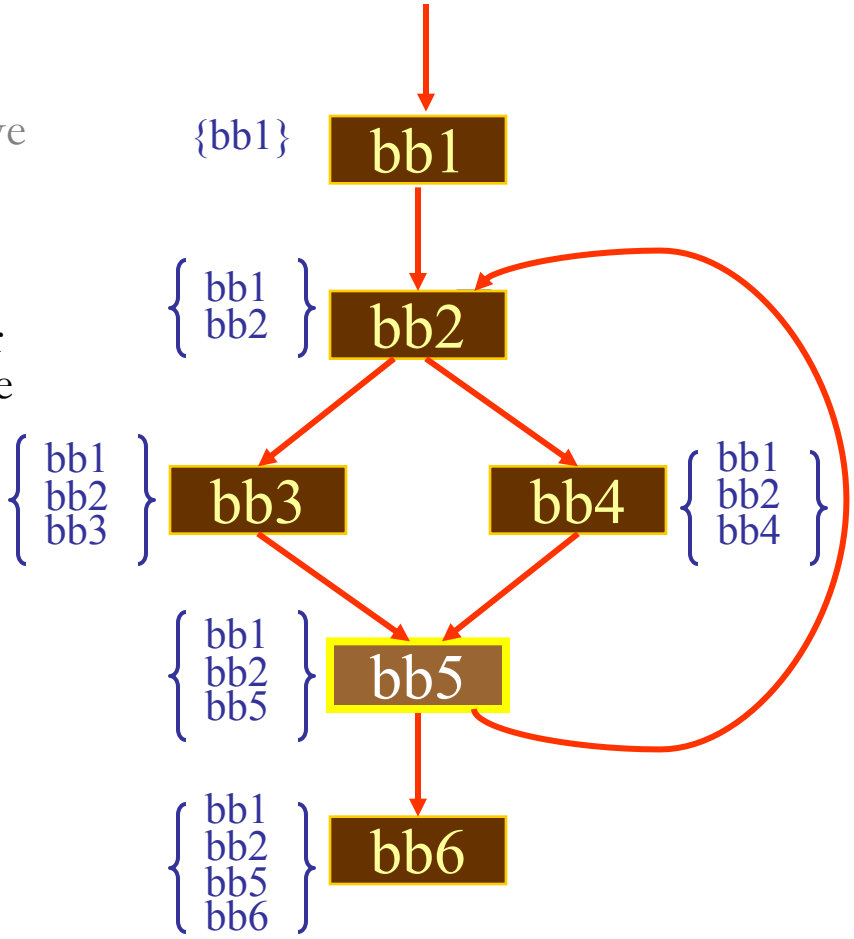
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



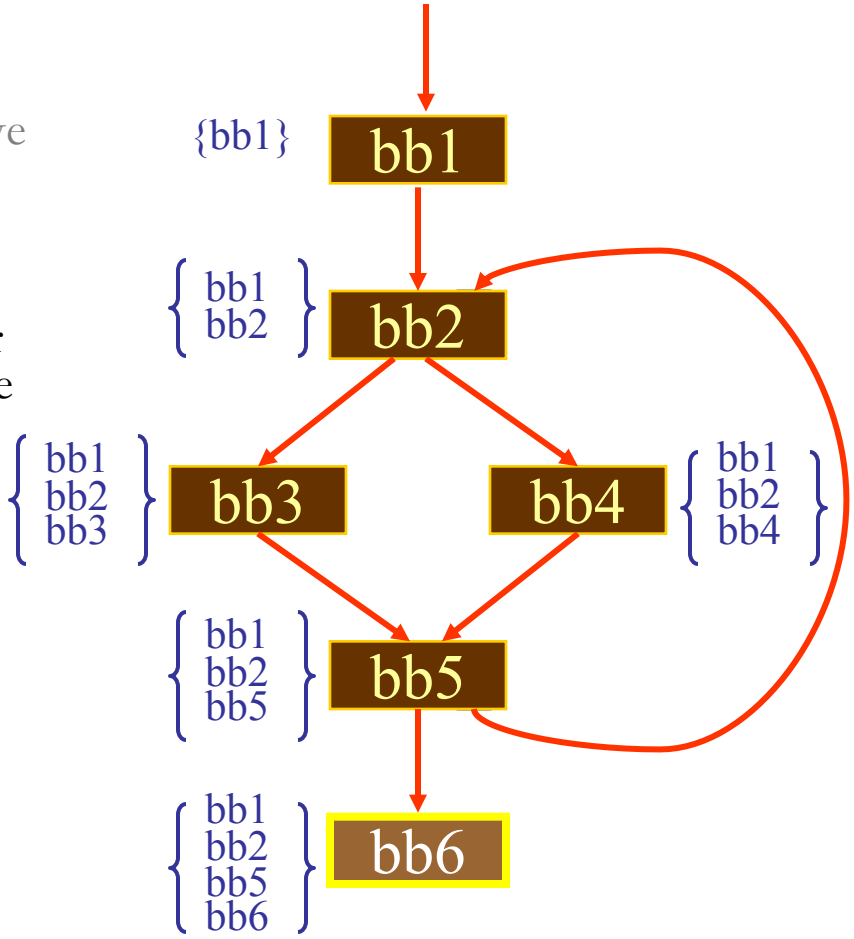
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



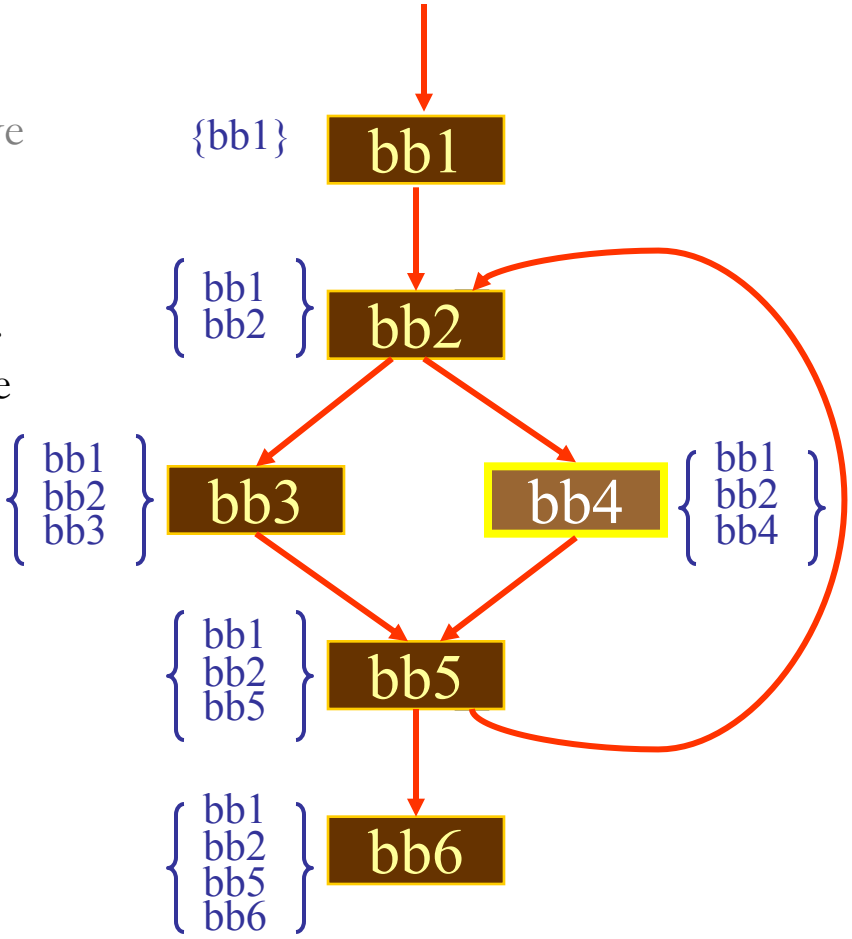
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



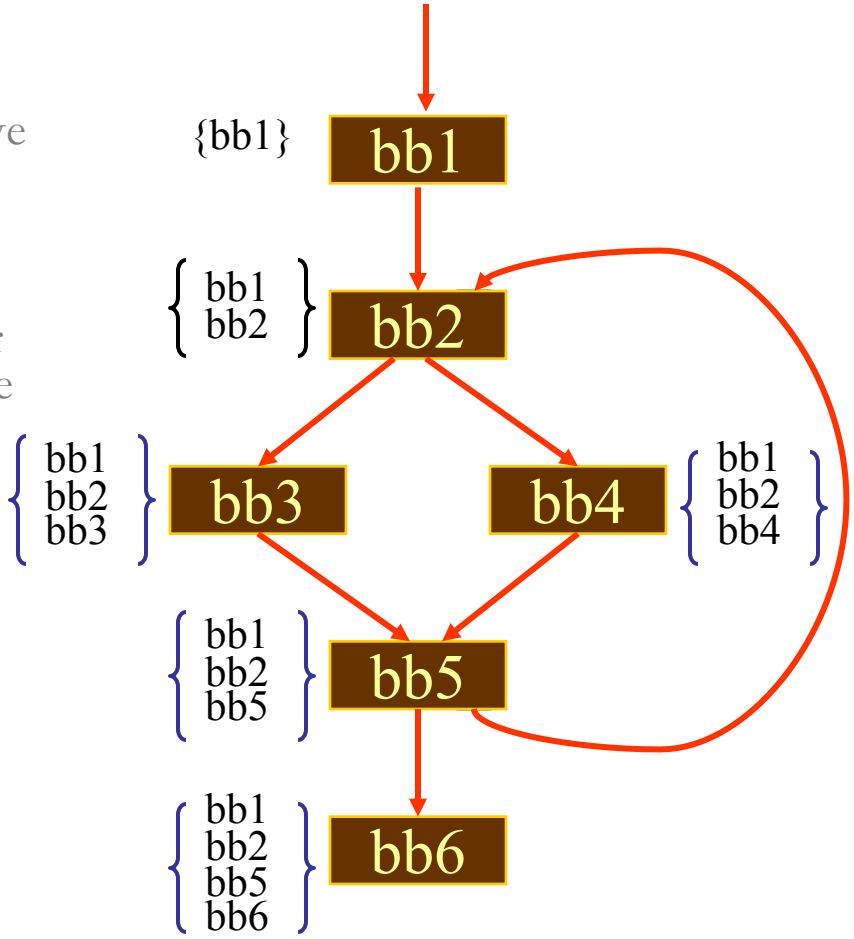
Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



Computing Dominators

- Algorithm
  - Make dominator set of the entry node has itself
  - Make dominator set of the rest have all the nodes
  - Visit the nodes in any order
  - Make dominator set of the current node intersection of the dominator sets of the predecessor nodes + the current node
  - Repeat until no change



Computing Dominators

- What we just witness was an Iterative Data-Flow Analysis Algorithm in Action
  - Initialize all the nodes to a given value
  - Visit nodes in some order
  - Calculate the node's value
  - Repeat until no value changes (fixed-point computation)

Data-Flow Analysis

A collection of techniques for compile-time reasoning about the runtime flow of values in a program

- Local Analysis
  - Analyze the "effect" of each Instruction in each Basic Block
  - Compose "effects" of instructions to derive information from beginning of basic block to each instruction
- Data-Flow Analysis
  - Iteratively propagate basic block information over the control-flow graph until no changes
  - Calculate the final value(s) at the beginning/end of the Basic Block
- Local Propagation
  - Propagate the information from the beginning/end of the Basic Block to each instruction

Outline

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

Example: Available Expression

- An Expression is *Available* at point *p* if and only if
  - All paths of execution reaching the current point passes through the point where the expression was defined
  - No variable used in the expression was modified between the definition point and the current point *p*

Example: Available Expression

- An Expression is *Available* at point *p* if and only if
  - All paths of execution reaching the current point passes through the point where the expression was defined
  - No variable used in the expression was modified between the definition point and the current point *p*

- In other words: Expression is still *current* at p

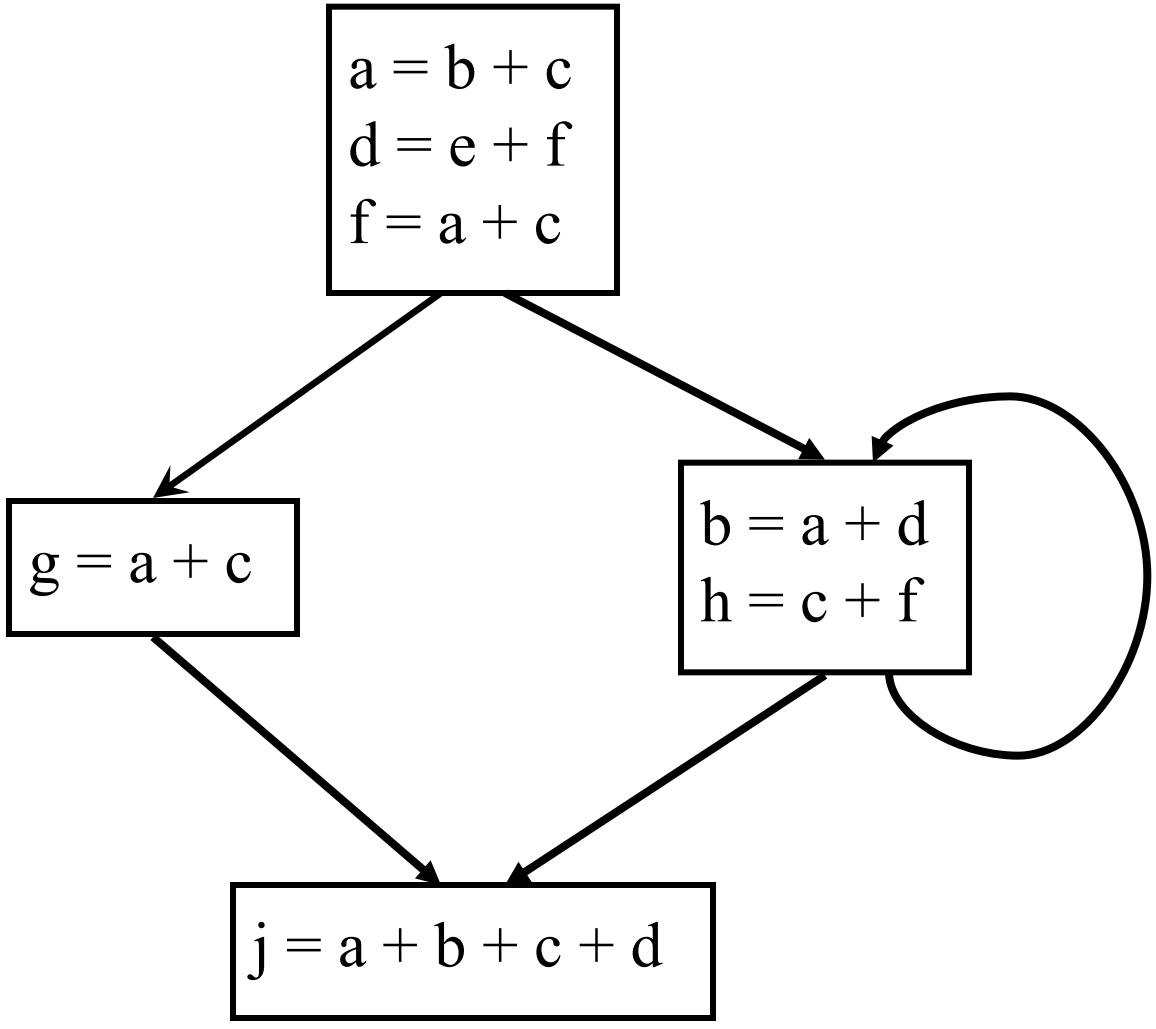
Example: Available Expression

- An Expression is *Available* at point *p* if and only if
  - All paths of execution reaching the current point passes through the point where the expression was defined
  - No variable used in the expression was modified between the definition point and the current point *p*
- In other words: Expression is still *current* at p
- Why is this a Data-Flow Problem?

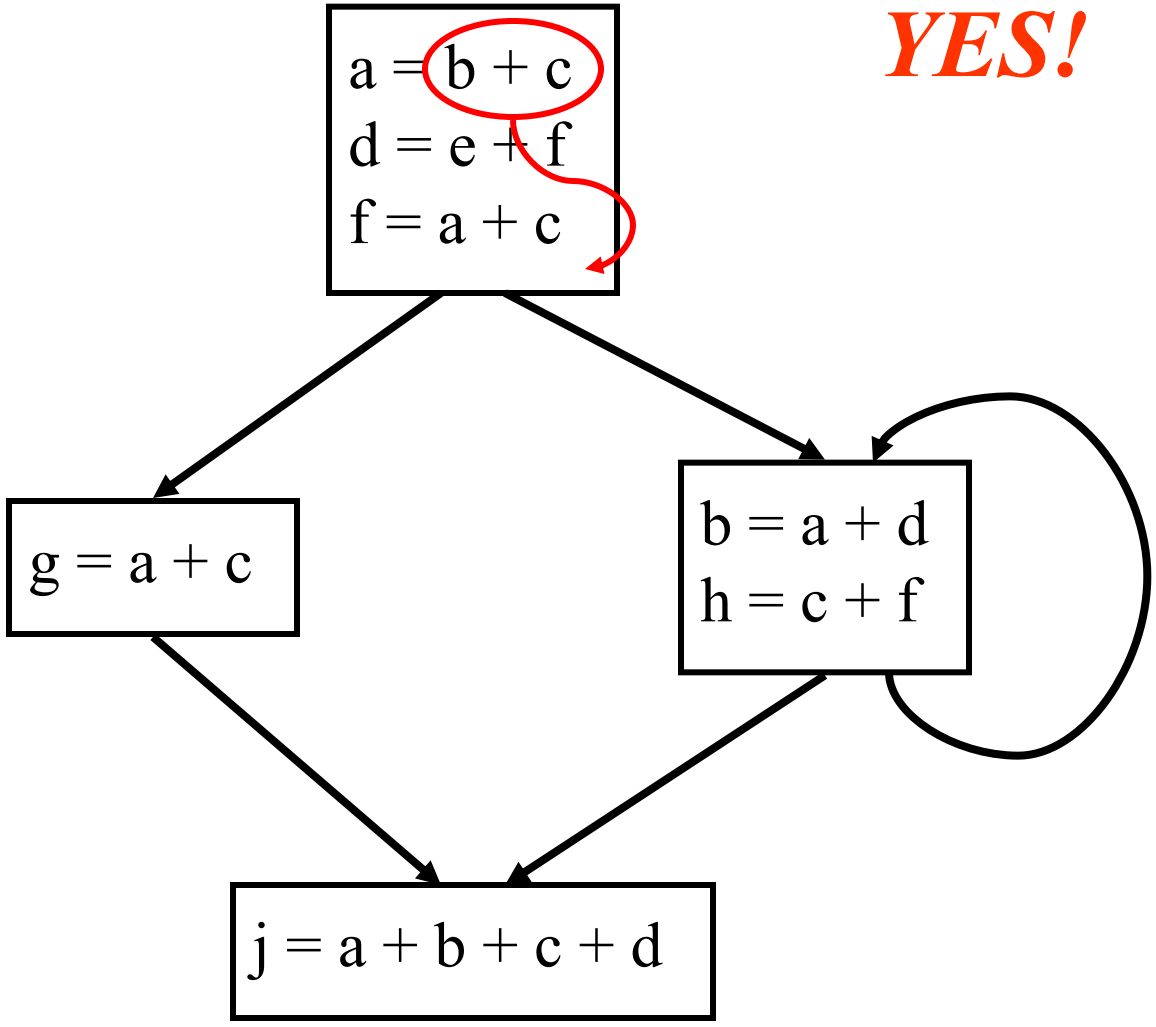
Example: Available Expression

- An Expression is *Available* at point *p* if and only if
  - All paths of execution reaching the current point passes through the point where the expression was defined
  - No variable used in the expression was modified between the definition point and the current point *p*
- In other words: Expression is still *current* at p
- Why is this a Data-Flow Problem?
  - We have to "know" a property about the program's execution that depends on the control-flow of the program!
  - All-Paths or At-Least-One-Path Issue.

Example: Available Expression

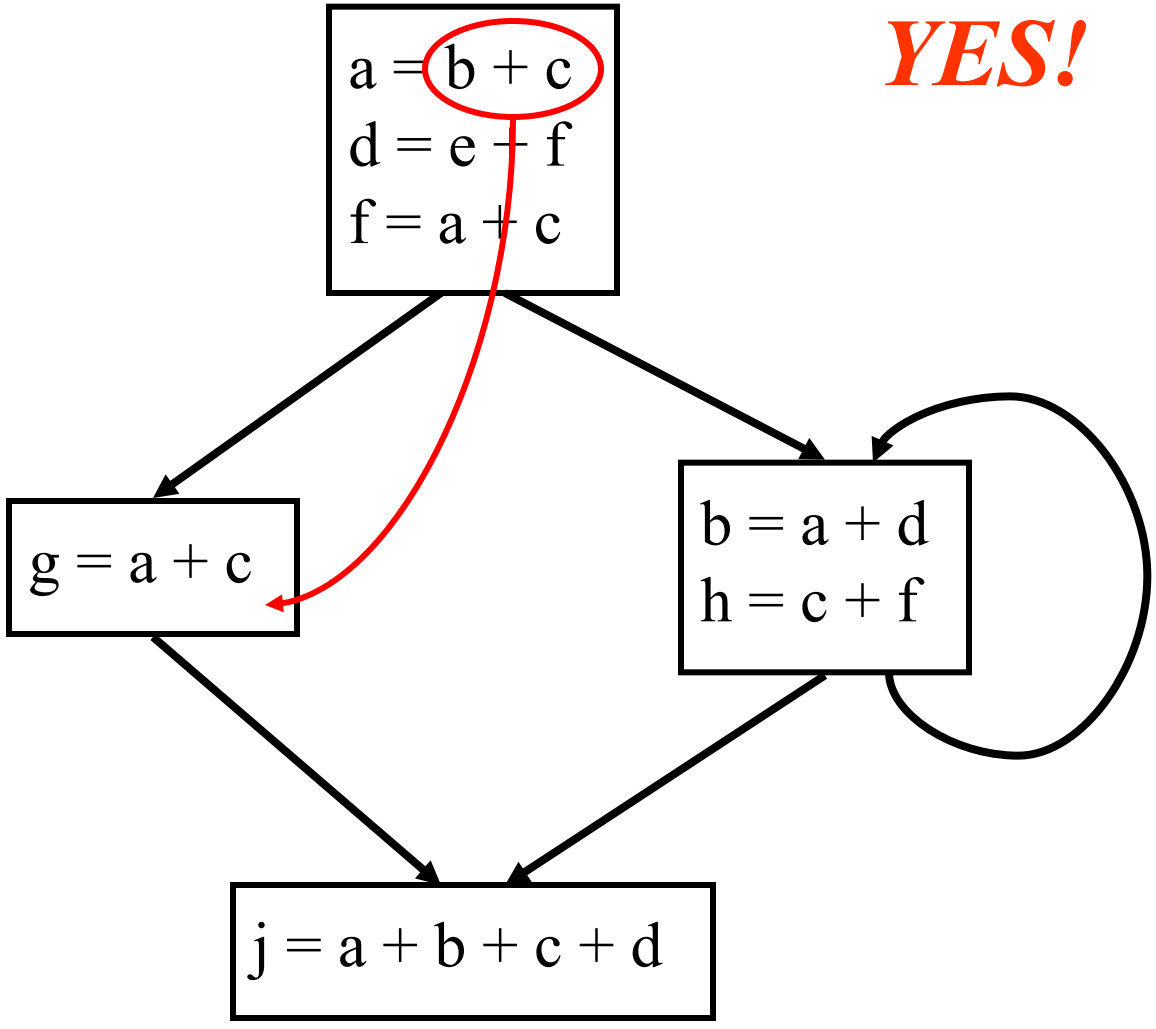


Is the Expression Available?



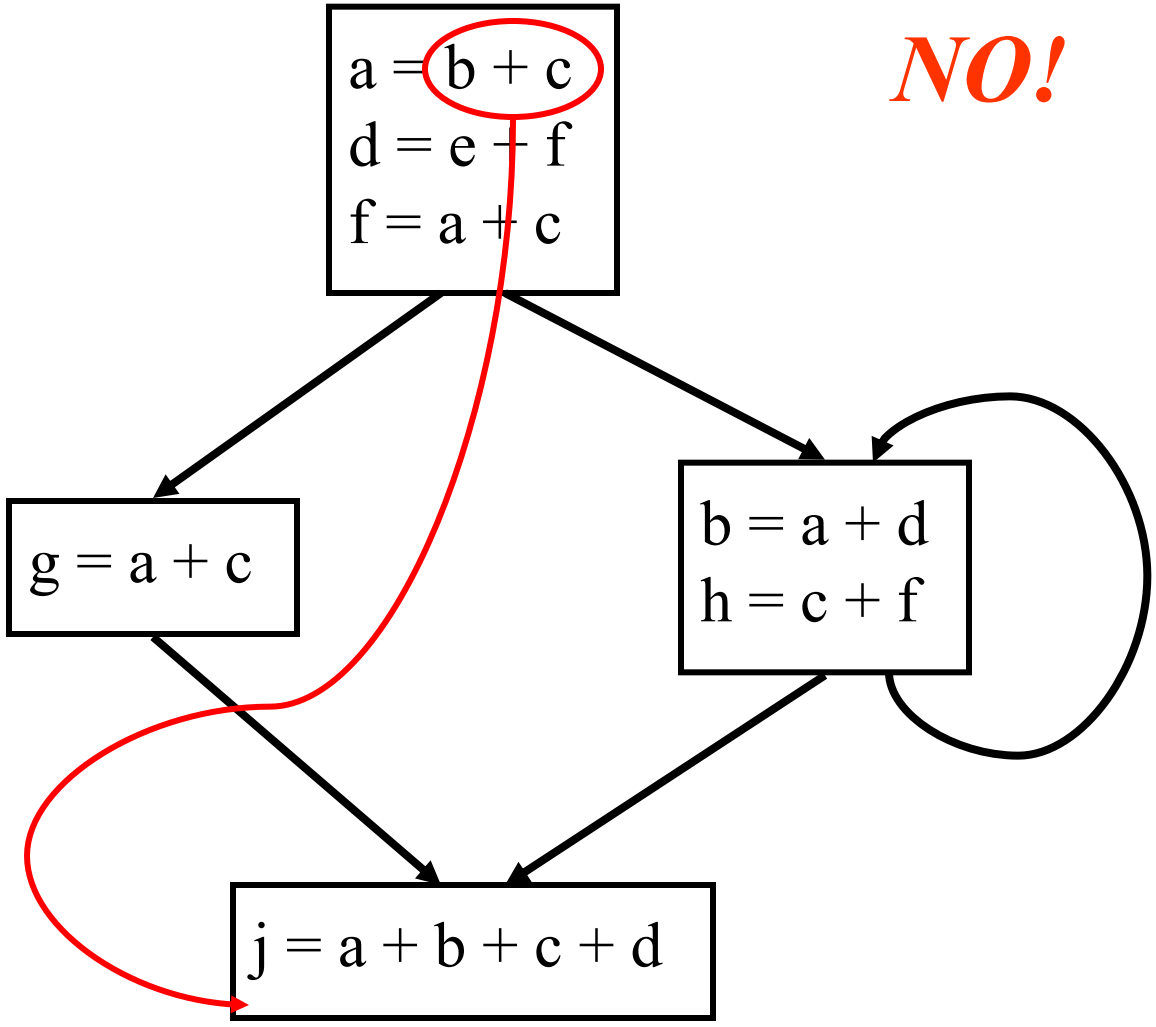
YES!

Is the Expression Available?



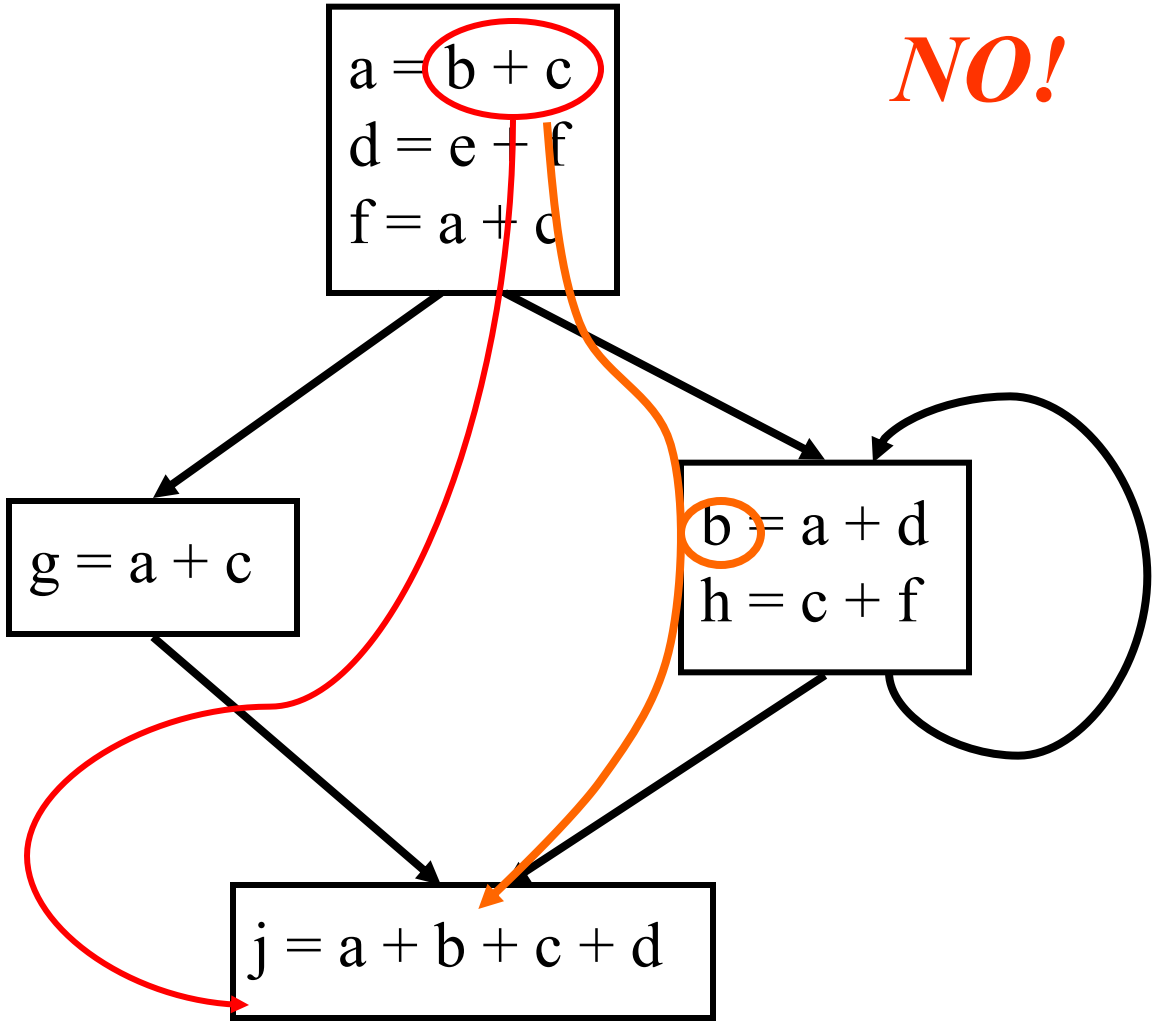
YES!

Is the Expression Available?



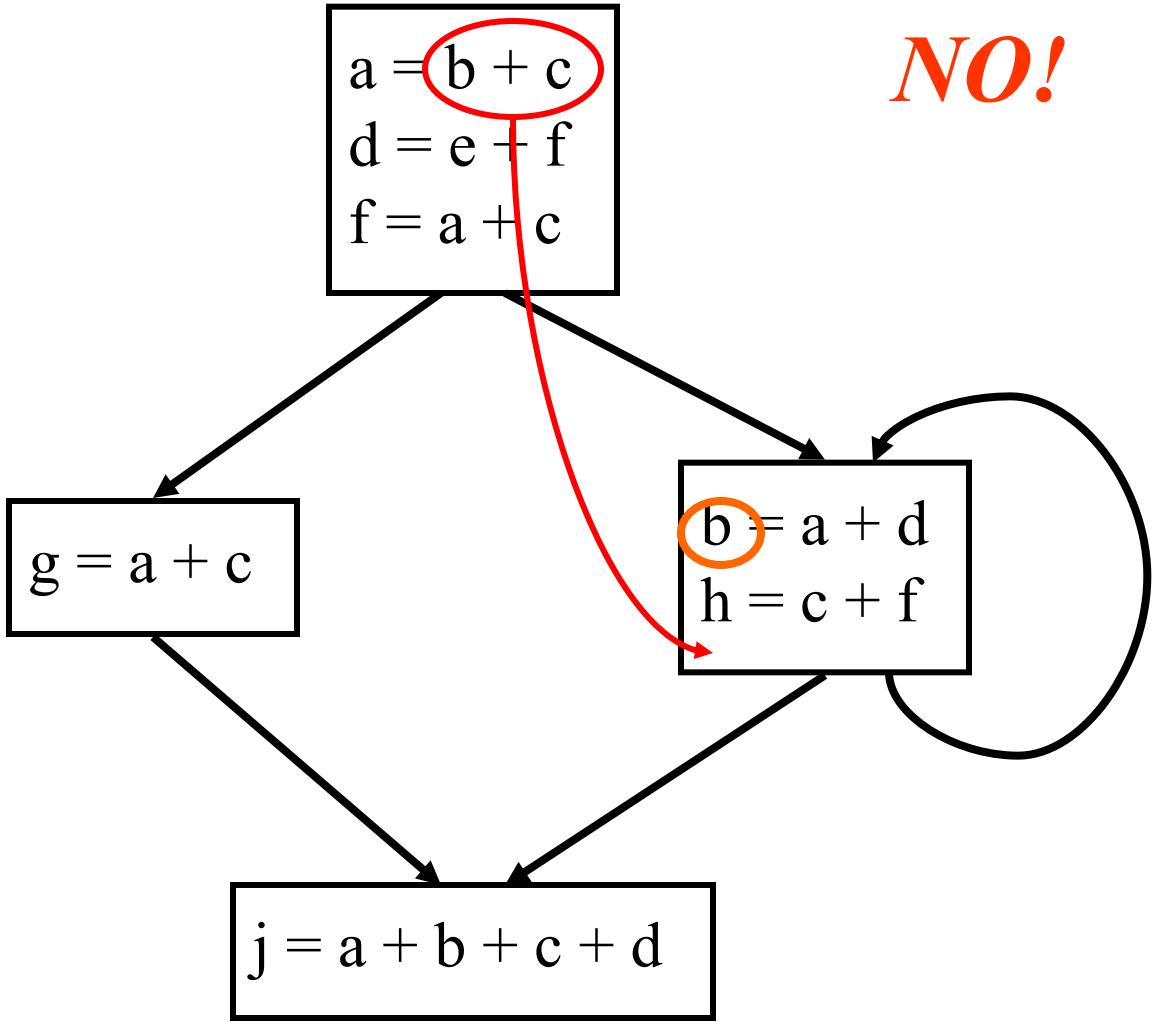
NO!

Is the Expression Available?



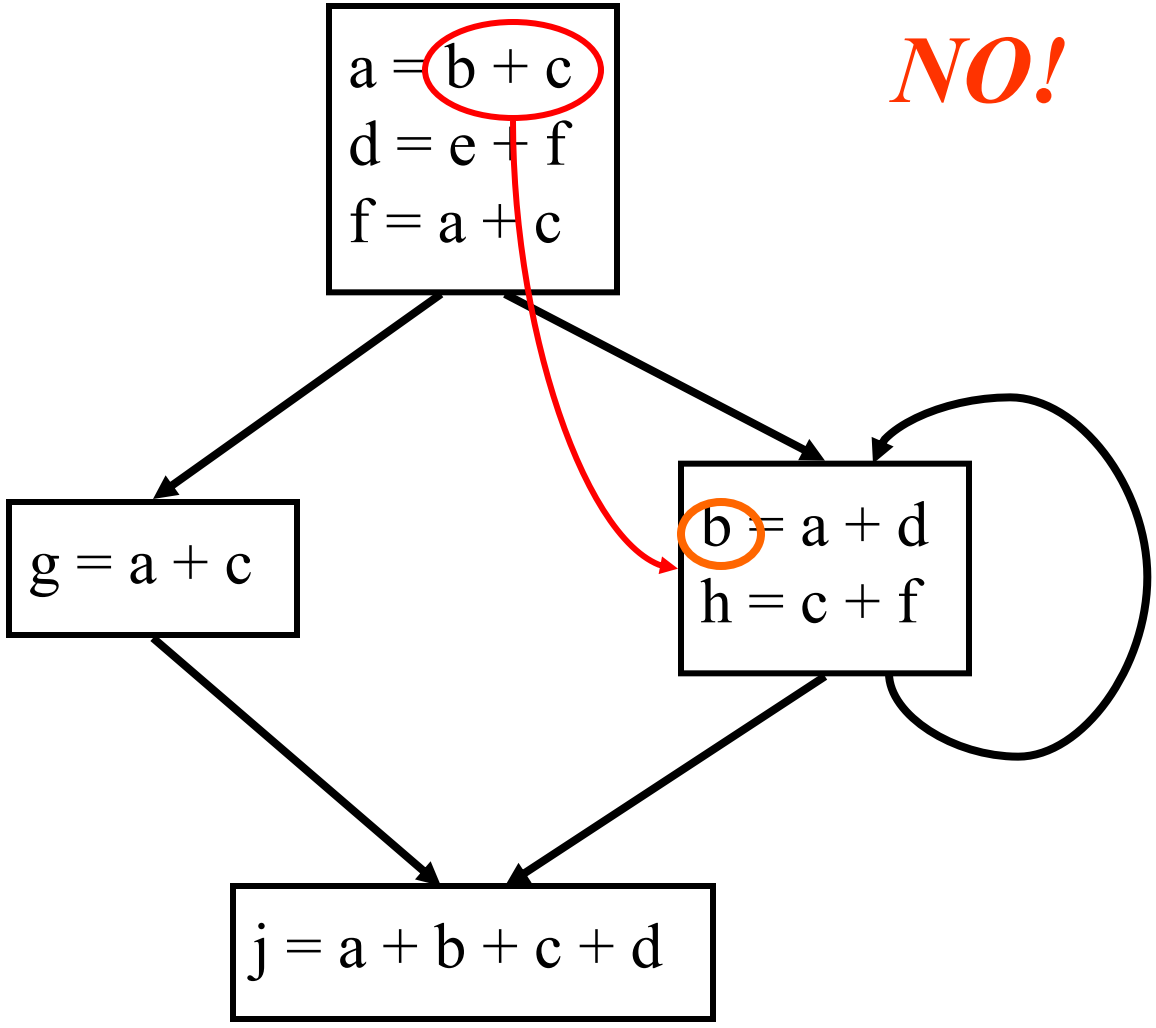
NO!

Is the Expression Available?



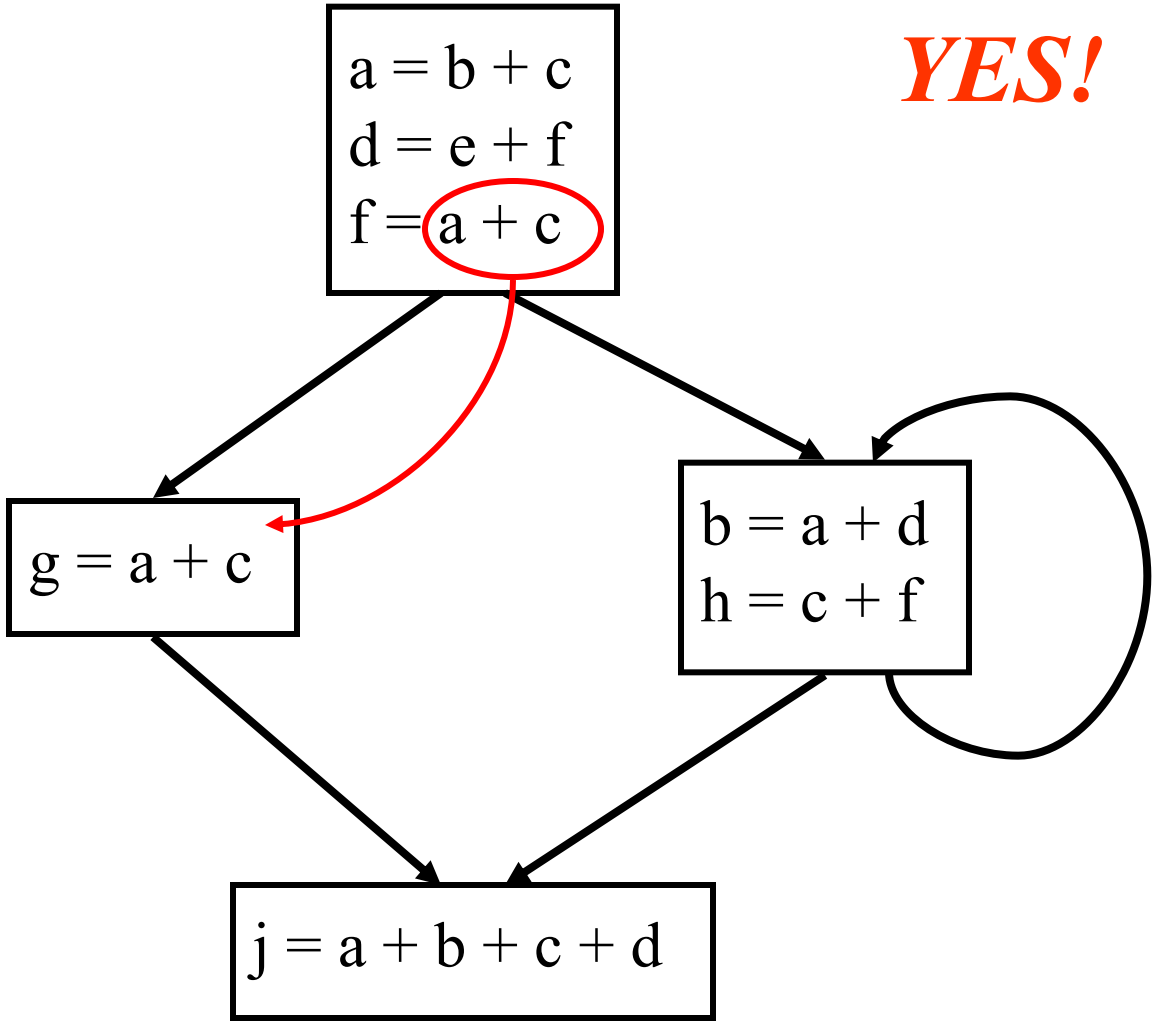
NO!

Is the Expression Available?



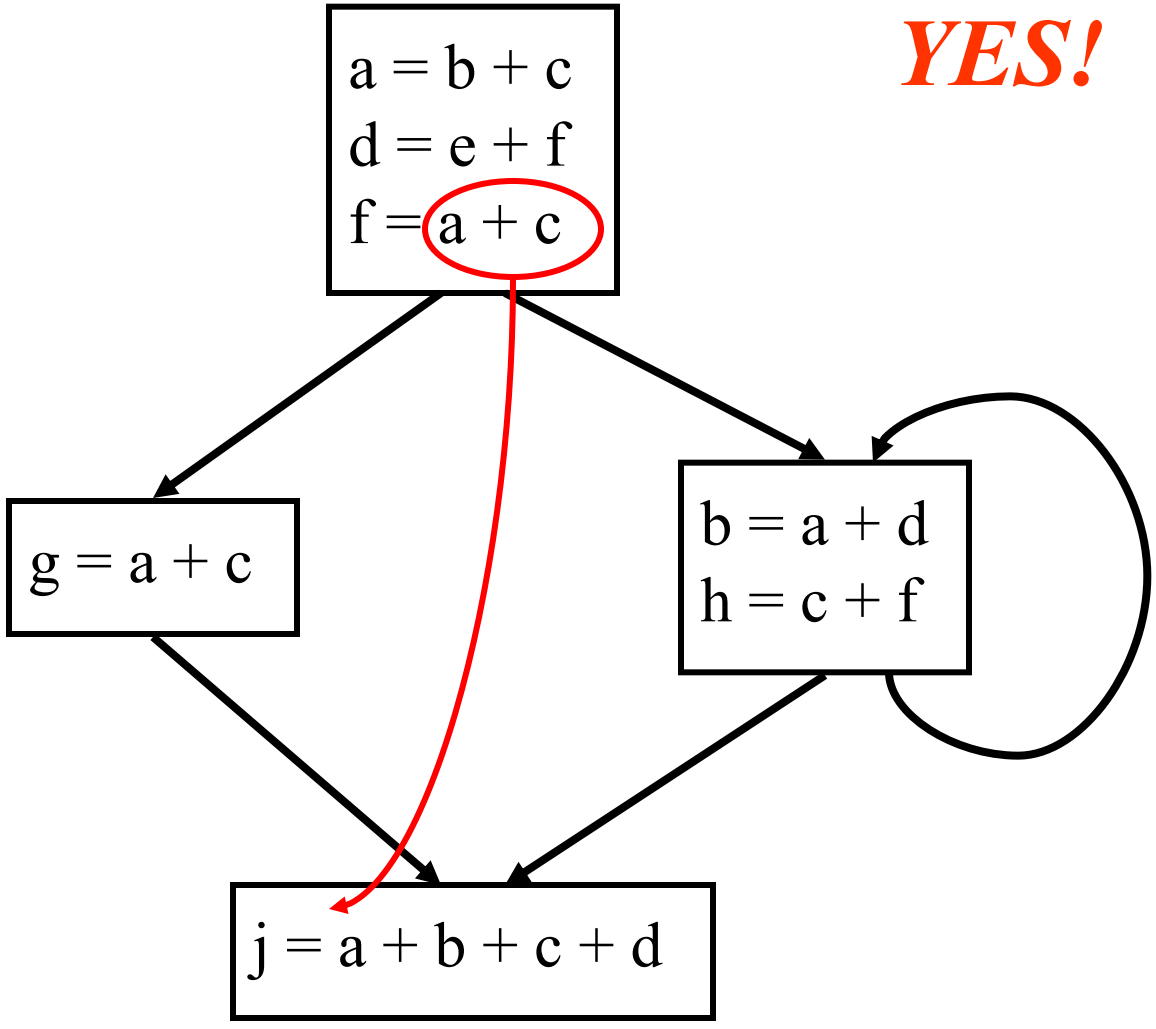
NO!

Is the Expression Available?



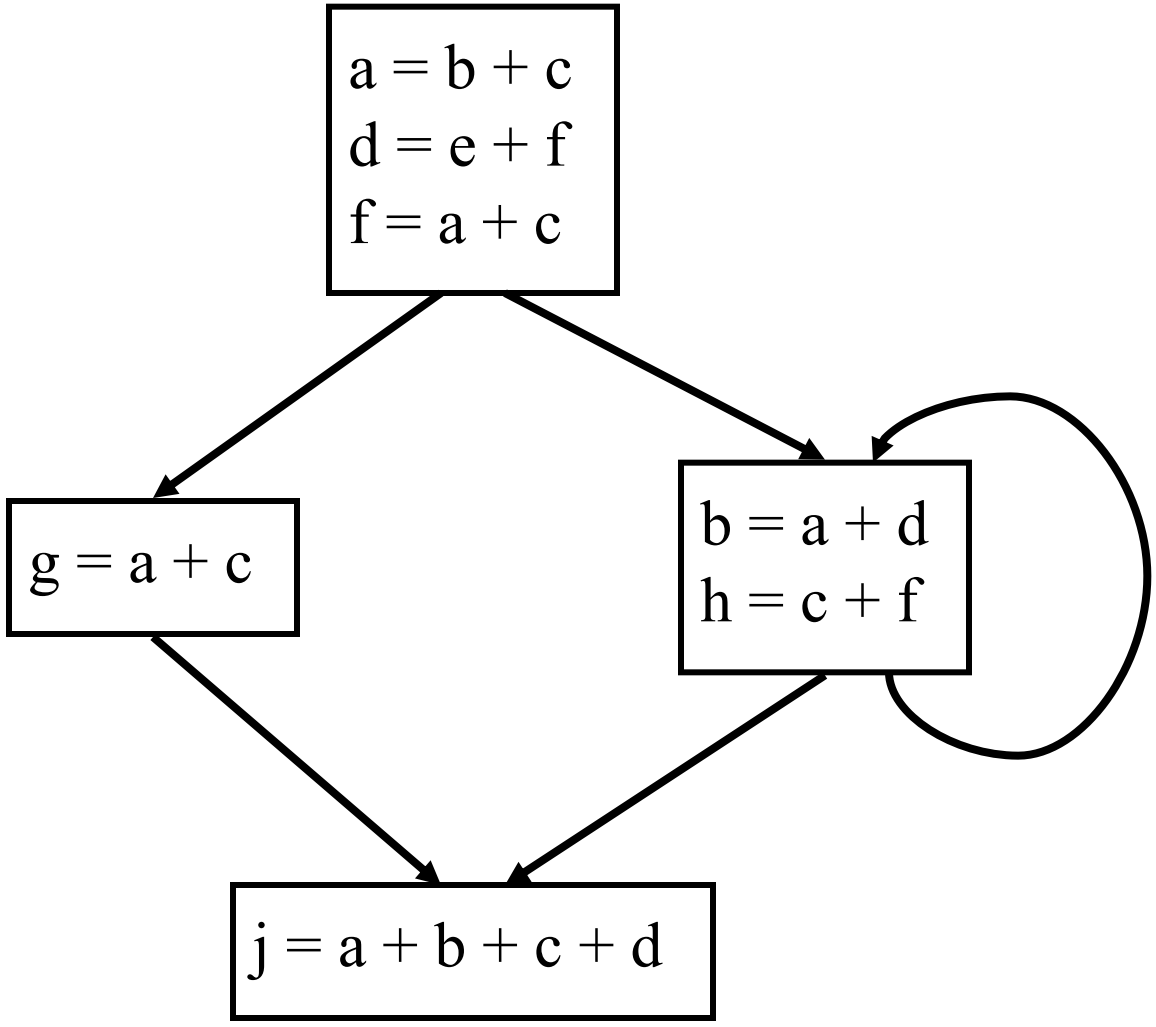
YES!

Is the Expression Available?

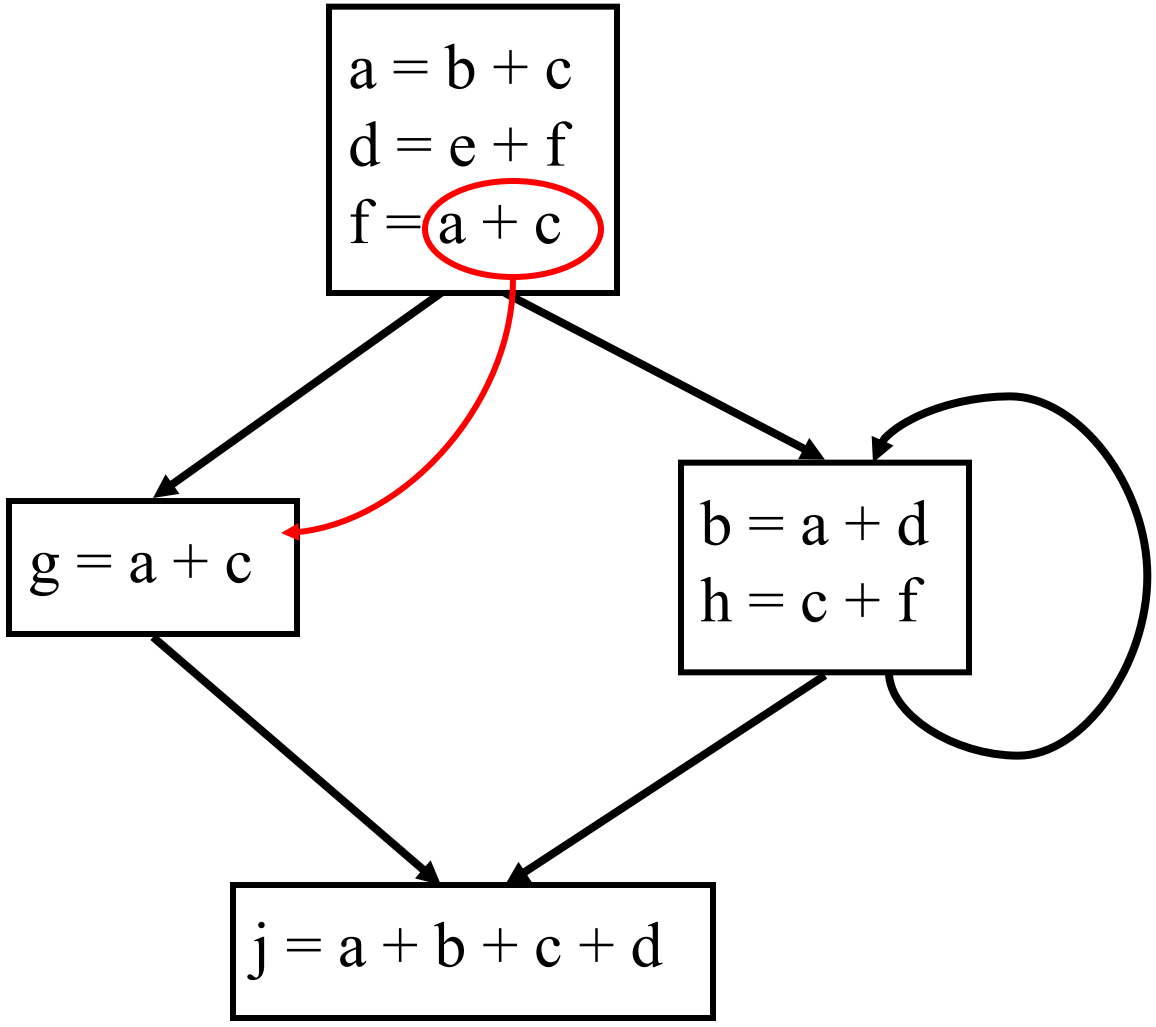


YES!

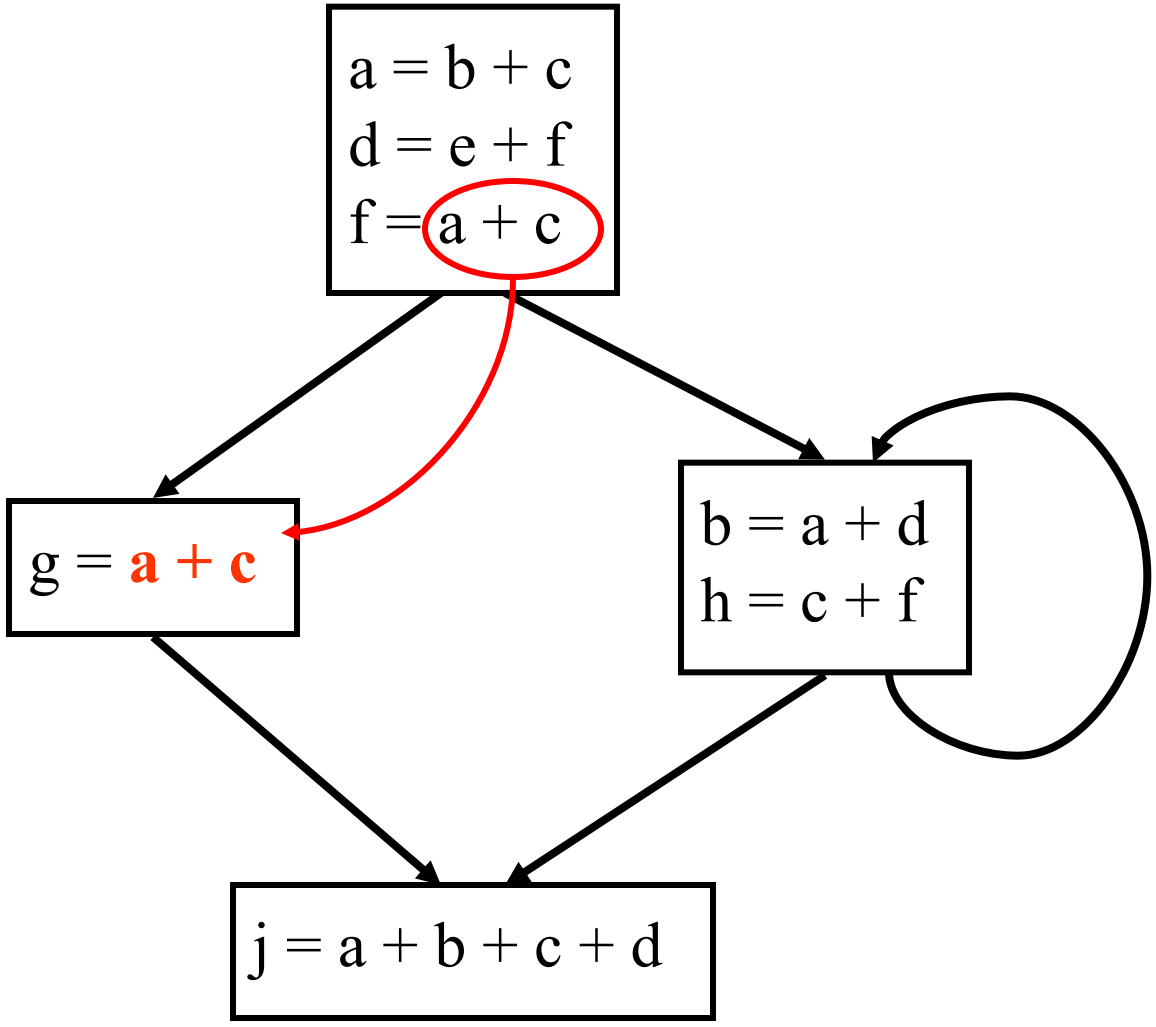
Use of Available Expressions



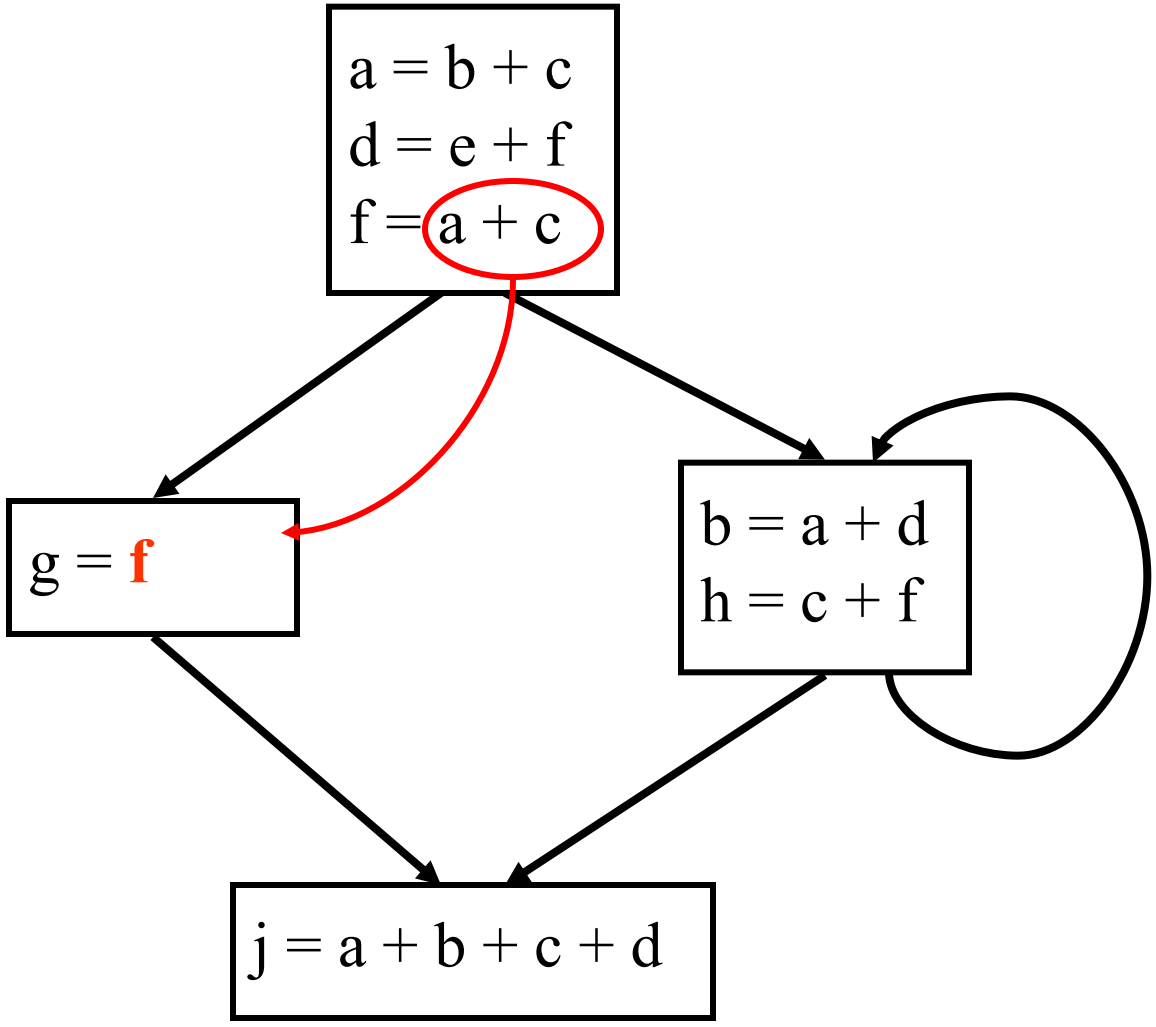
Use of Available Expressions



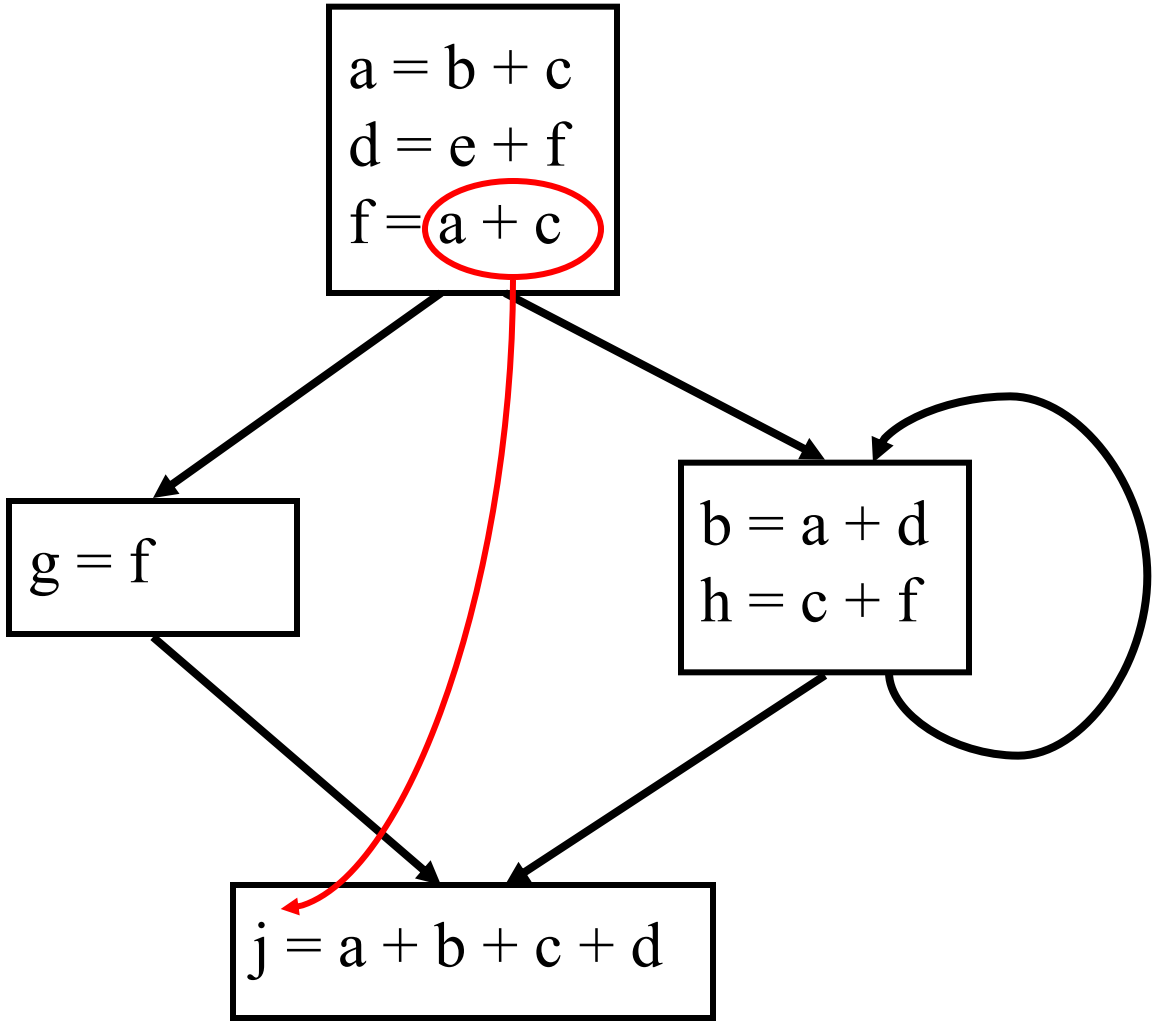
Use of Available Expressions



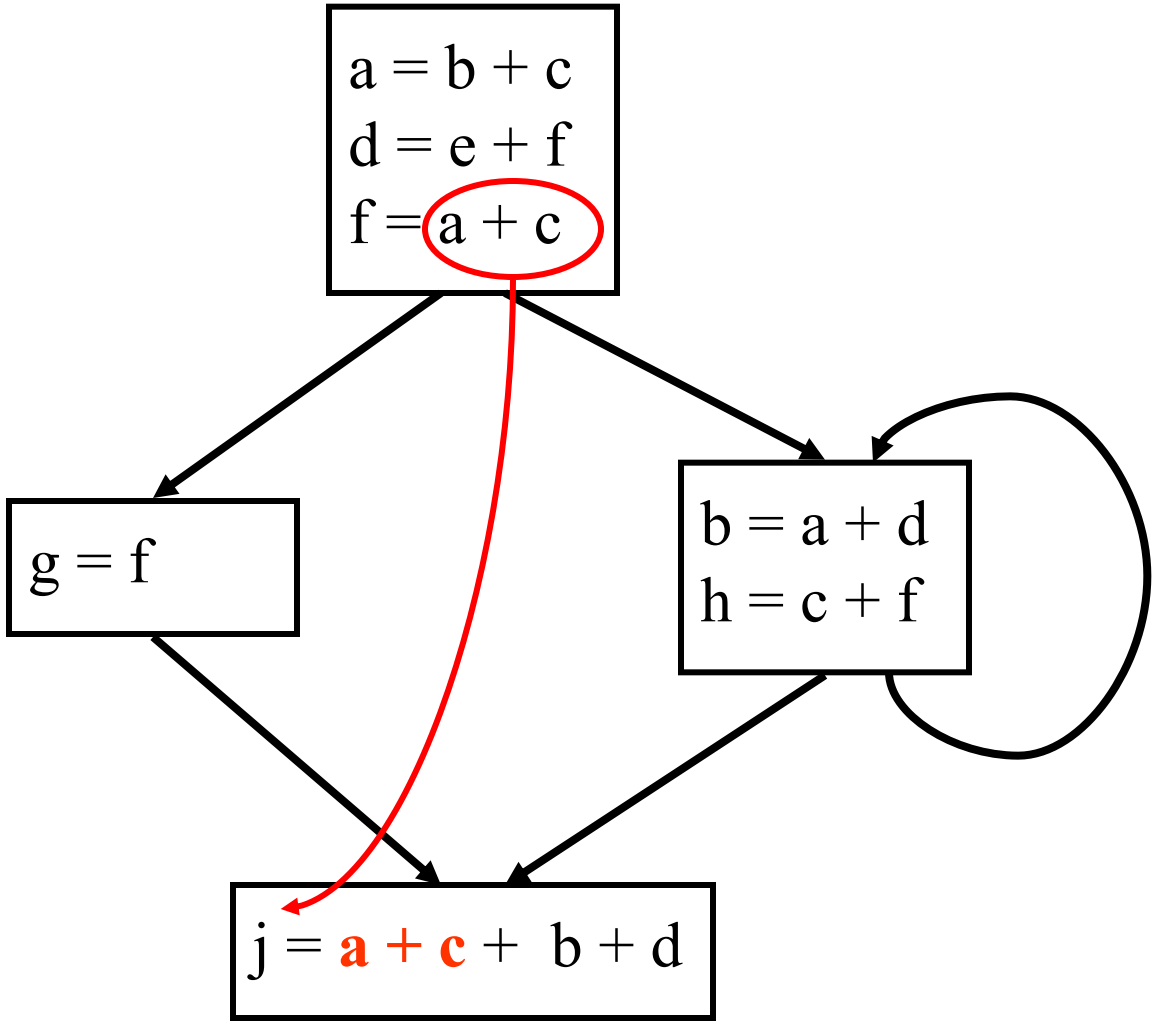
Use of Available Expressions



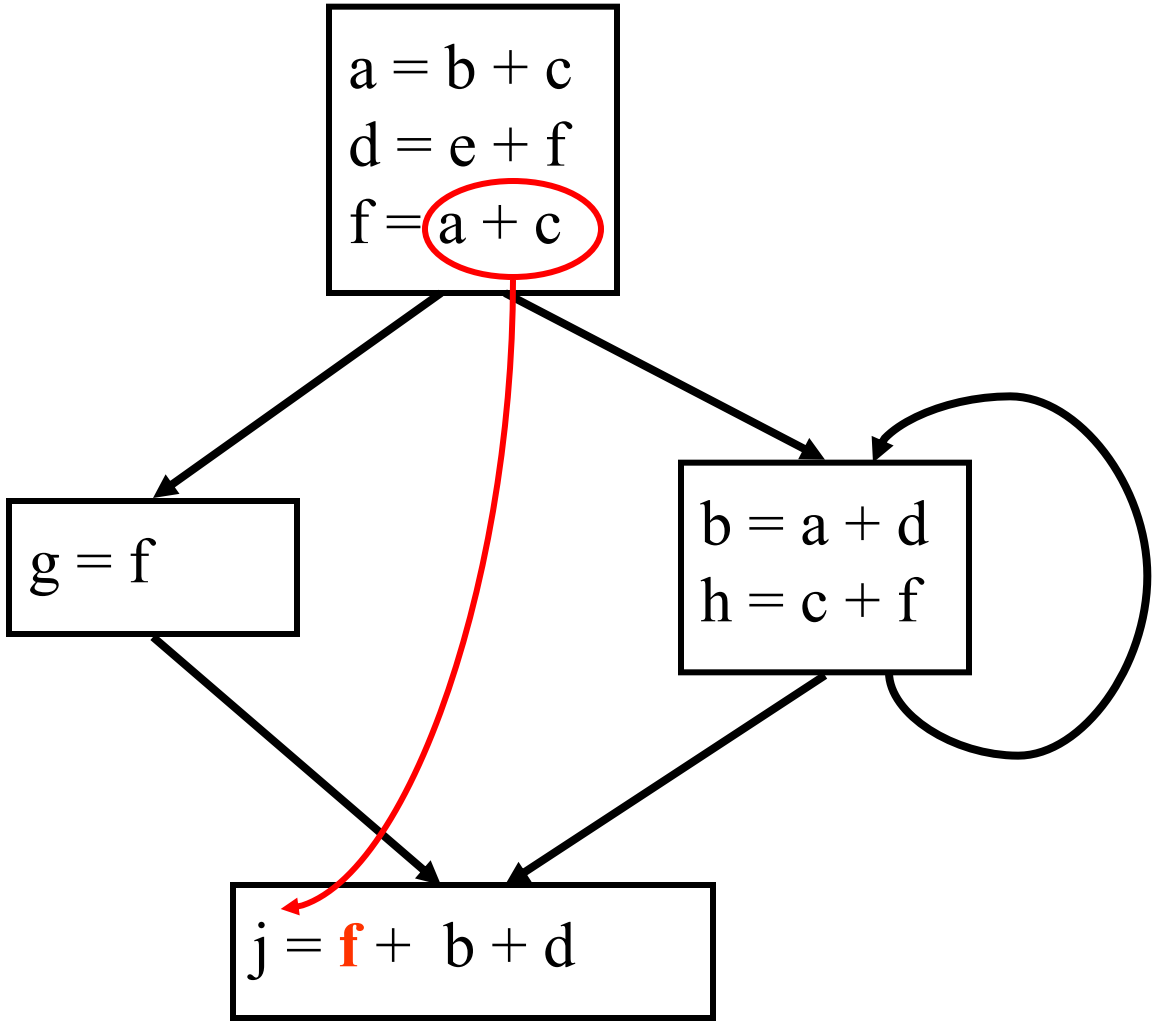
Use of Available Expressions



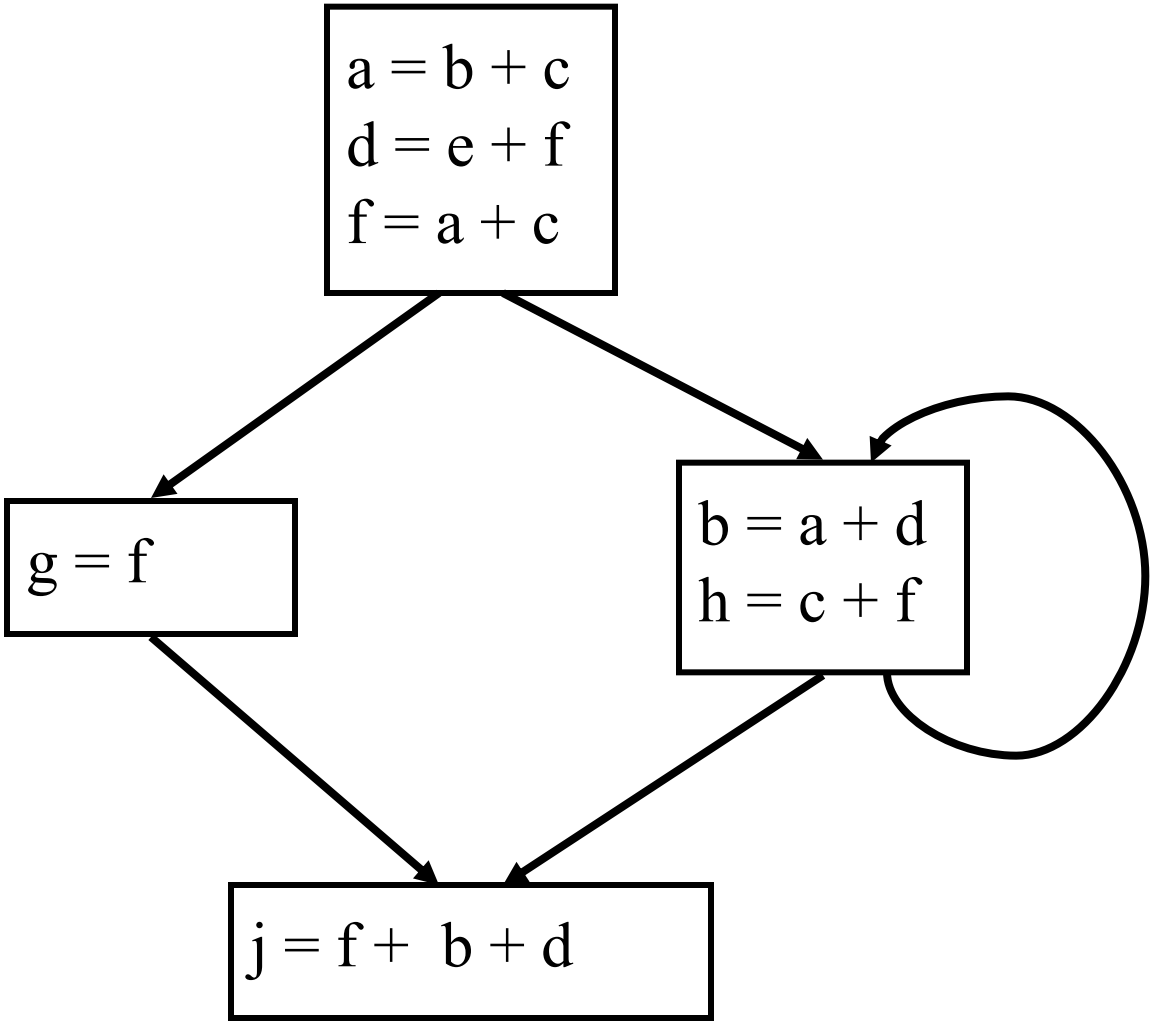
Use of Available Expressions



Use of Available Expressions



Use of Available Expressions



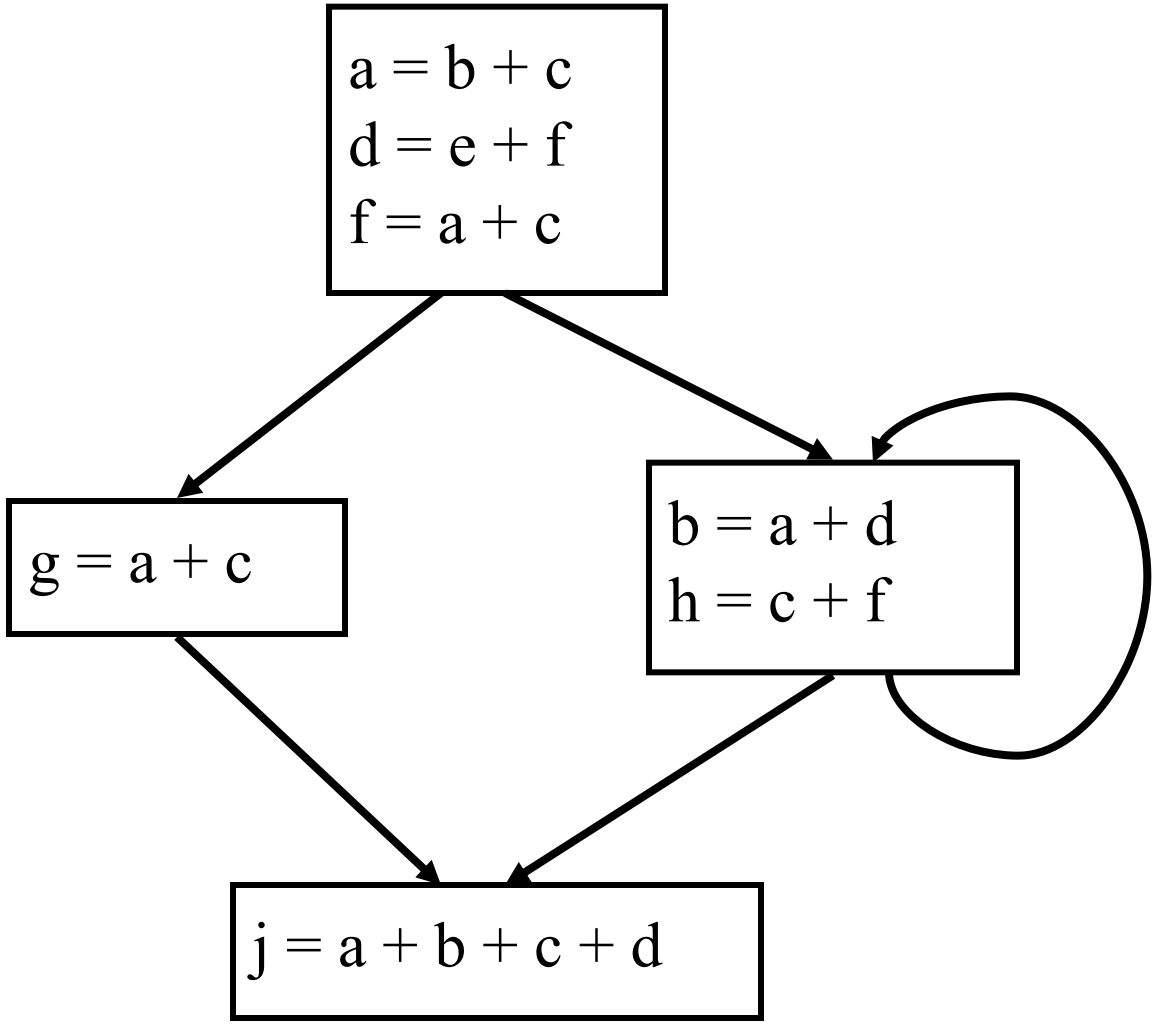
Outline

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

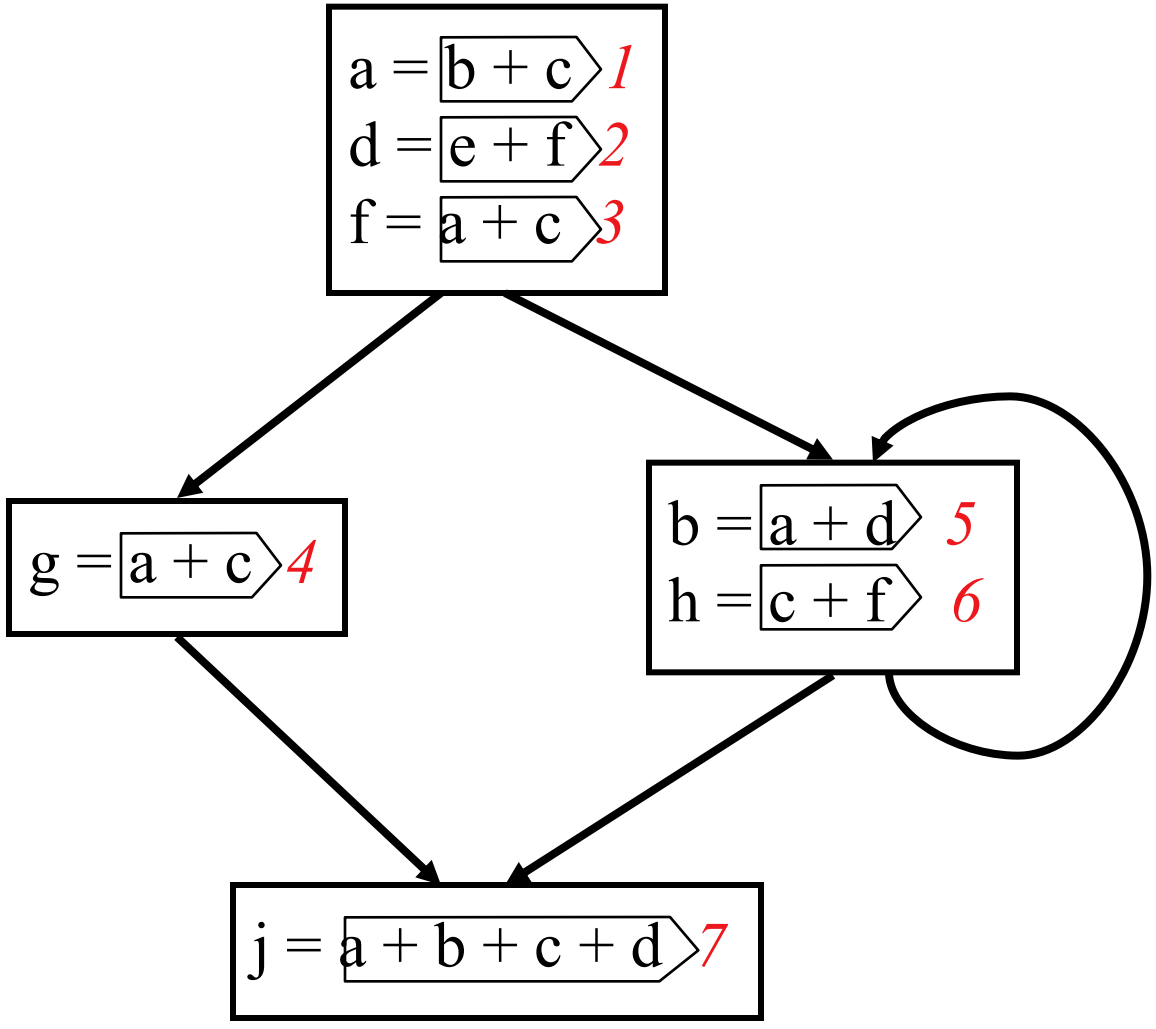
Algorithm for Available Expression

- Assign a Number to each Expression in the Program

Example: Available Expression



Example: Available Expression



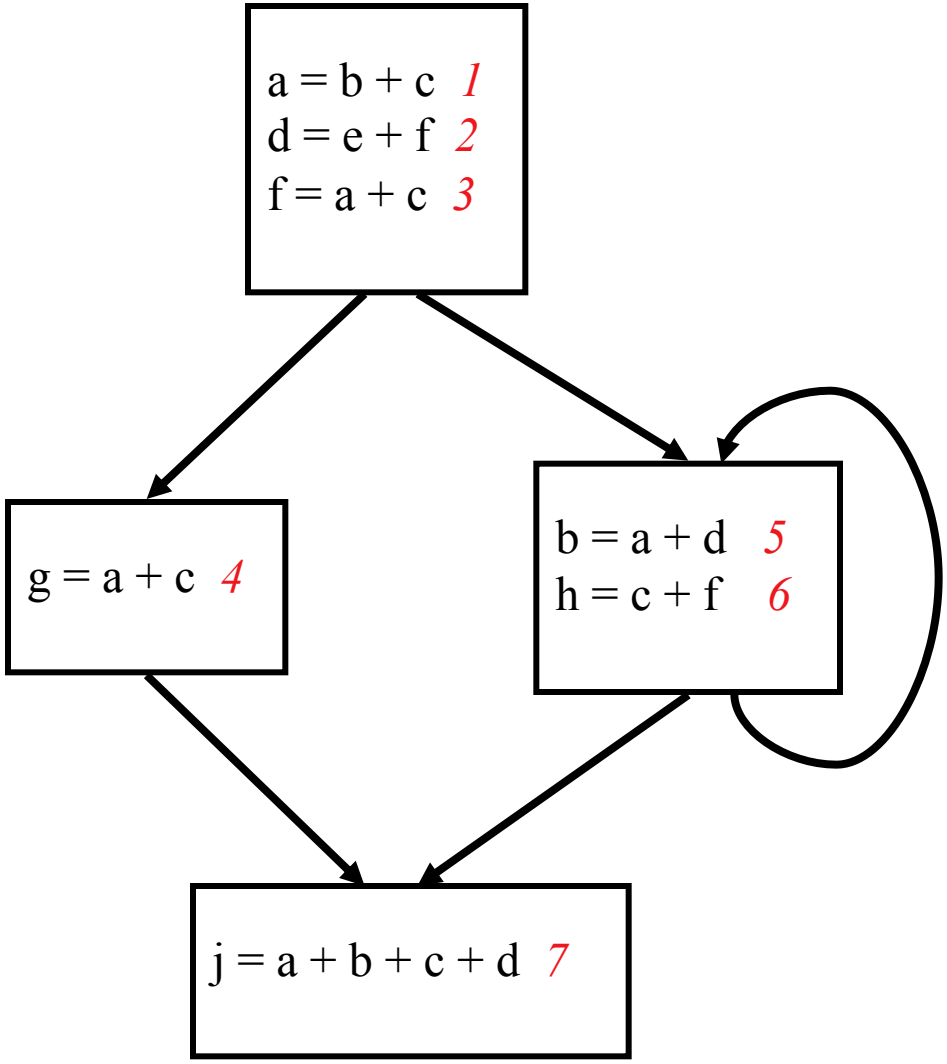
Gen and Kill Sets

- Gen Set
  - If a Basic Block (or instruction) defines the expression then the expression number is in the Gen Set for that Basic Block (or instruction)
- Kill Set
  - If a Basic Block (or instruction) (re)defines a variable in the expression then that expression number is in the Kill Set for that Basic Block (or instruction)
  - Expression is thus not valid after that Basic Block (or instruction)

Algorithm for Available Expression

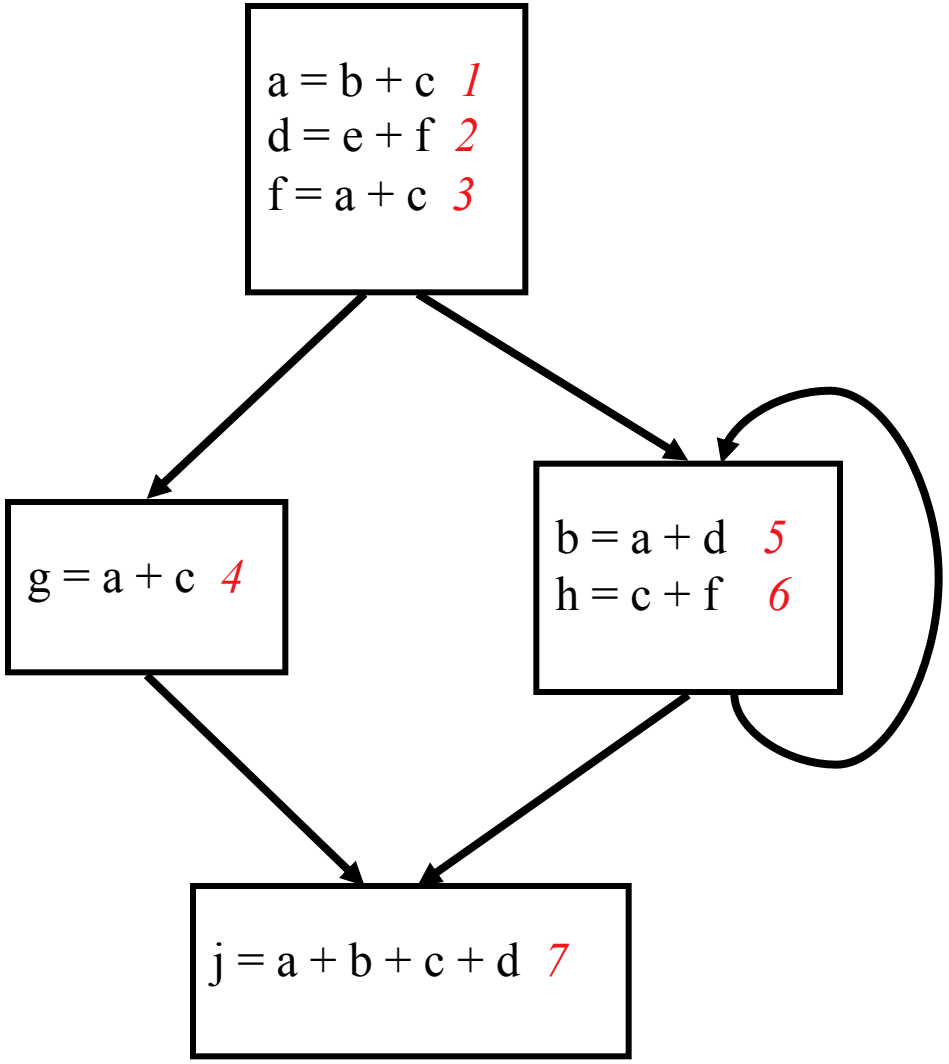
- Assign a Number to each Expression in the Program
- Compute Gen Set and Kill Set for each Basic Block (or instruction)
  - Compute Gen Set and Kill Set for each Instruction in Basic Block

Gen and Kill Sets



Gen and Kill Sets

a = b + c	*1*
d = e + f	*2*
f = a + c	*3*



Gen and Kill Sets

a = b + c *1*

gen = { b + c }

kill = { any expr with a }

d = e + f *2*

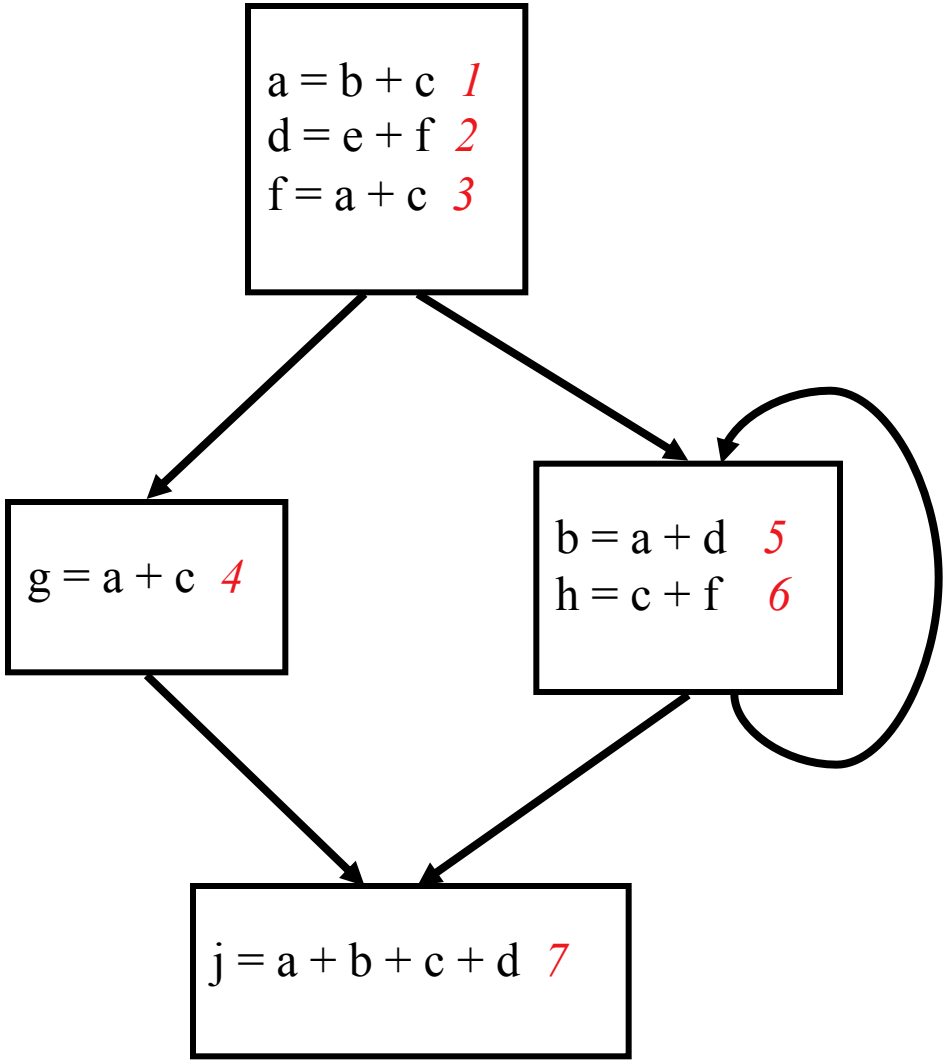
gen = { e + f }

kill = { any expr with d }

f = a + c *3*

gen = { a + c }

kill = { any expr with f }



Gen and Kill Sets

a = b + c *1*

gen = { 1 }

kill = { 3, 4, 5, 7 }

d = e + f *2*

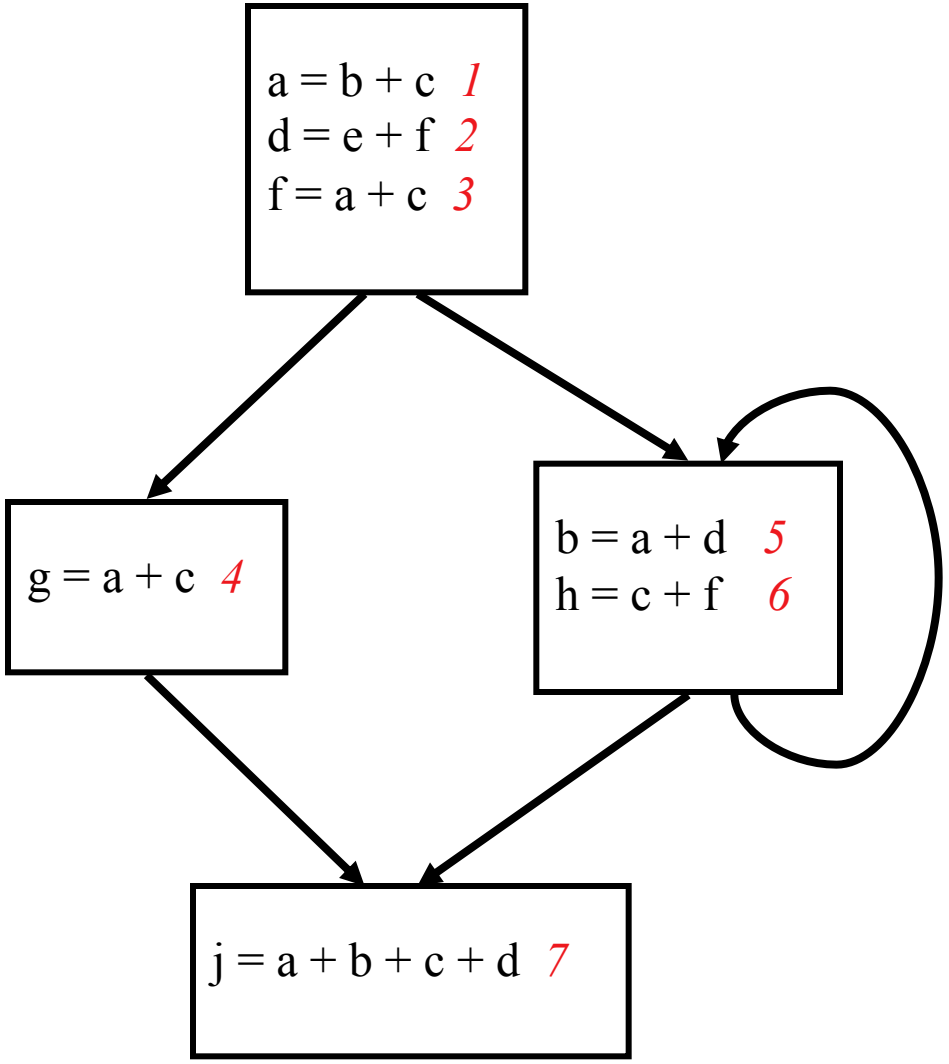
gen = { 2 }

kill = { 5, 7 }

f = a + c *3*

gen = { 3 }

kill = { 2, 6 }



Algorithm for Available Expression

- Assign a Number to each Expression in the Program
- Compute Gen Set and Kill Set for each Basic Block (or instruction)
  - Compute Gen Set and Kill Set for each Instruction in Basic Block
  - Compose them to create Basic Block Gen and Kill Sets

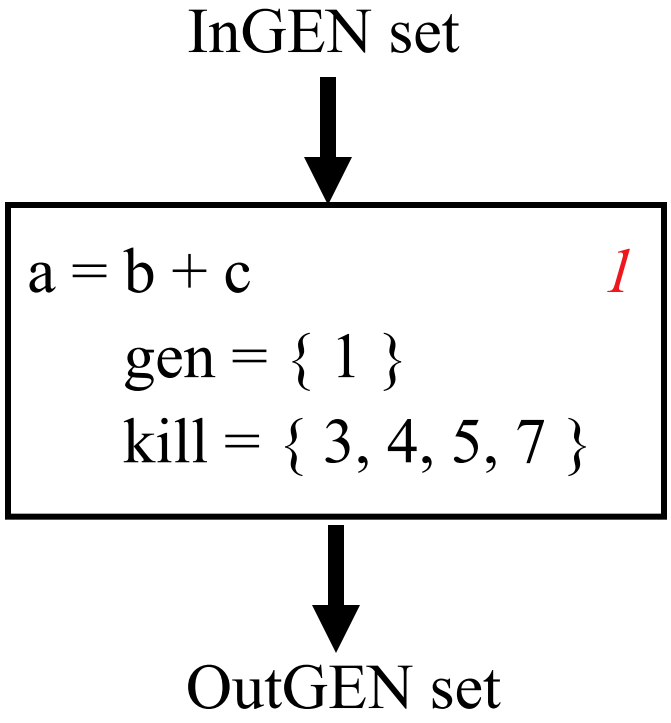
Aggregate Gen and Kill Sets

```
a = b + c                    1
    gen = { 1 }
    kill = { 3, 4, 5, 7 }

d = e + f                    2
    gen = { 2 }
    kill = { 5, 7 }

f = a + c                    3
    gen = { 3 }
    kill = { 2, 6 }
```

- Propagate all the Gen Sets and Kill Sets from top of the basic block to the bottom of the basic block

- How?

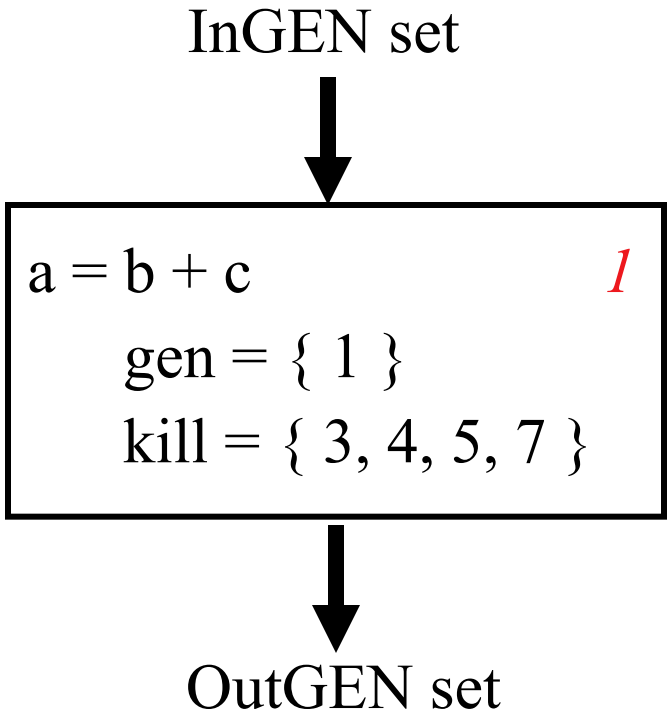
Aggregate Gen Set



OutGEN =

Aggregate Gen Set

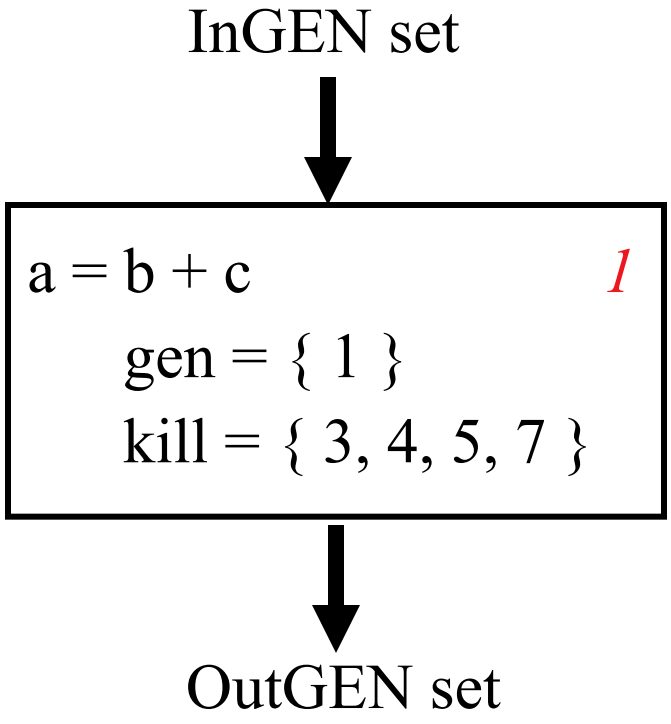
- An expression in the Gen Set in the current instruction should be in the OutGEN Set



OutGEN = gen

Aggregate Gen Set

- An expression in the Gen Set in the current instruction should be in the OutGEN Set
- Any expression in the InGEN Set that is not killed should be in the OutGEN Set



$$OutGEN = gen \cup (InGEN - kill)$$

Aggregate Gen Set

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = genU(InGEN - kill)

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 } ∪ ({ } - { 3,4,5,7})

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = genU(InGEN - kill)

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = { 2 } ∪ ({ 1 } - { 5,7 })

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = { 1, 2 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = { 1, 2 }

InGEN = { 1, 2 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutGEN = genU(InGEN - kill)

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = { 1, 2 }

InGEN = { 1, 2 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutGEN = { 3 } ∪ ({1, 2} - {2, 6})

Aggregate Gen Set

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = { 1, 2 }

InGEN = { 1, 2 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutGEN = { 1, 3 }

Aggregate Gen Set

GEN = { 1, 3 }

InGEN = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutGEN = { 1 }

InGEN = { 1 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutGEN = { 1, 2 }

InGEN = { 1, 2 }

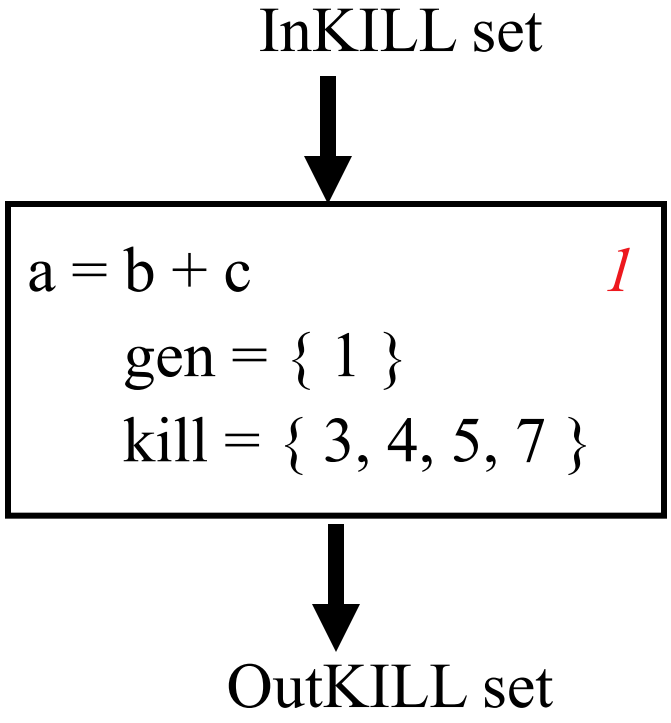
f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutGEN = { 1, 3 }

Aggregate Kill Set

- An expression in the Kill Set in the current instruction should be in the OutKILL set
- Note that different instruction define differently numbered expressions

⇒ expressions are unique



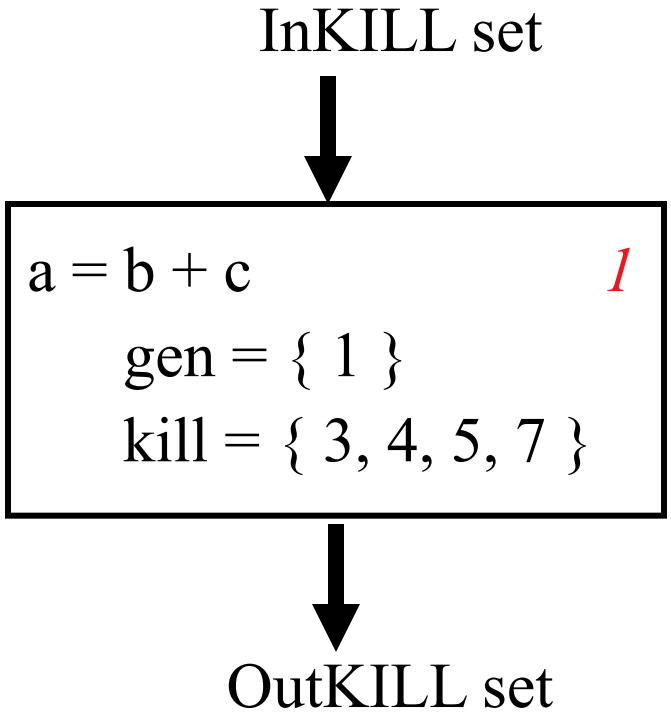
OutKILL =

Aggregate Kill Set

- An expression in the Kill Set in the current instruction should be in the OutKILL set
- Note that different instruction define differently numbered expressions

⇒ expressions are unique

- Any expression in the InKILL set should be in OutKILL



OutKILL = kill

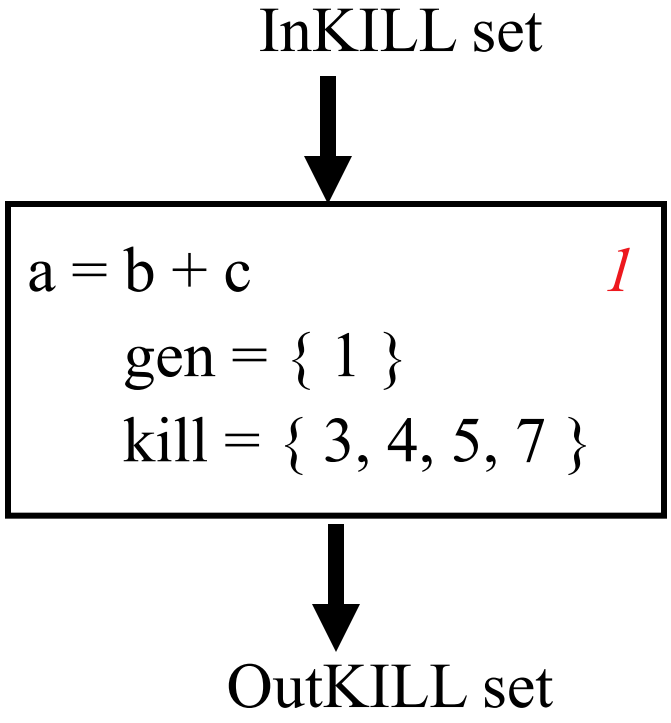
Aggregate Kill Set

- An expression in the Kill Set in the current instruction should be in the OutKILL set
- Note that different instruction define differently numbered expressions

⇒ expressions are unique

- Any expression in the InKILL set should be in OutKILL

$$OutKILL = kill \cup InKILL$$



Aggregate Kill Set

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = kill ∪ InKILL

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 } ∪ { }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutKILL = kill ∪ InKILL

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutKILL = { 5, 7 } ∪ { 3, 4, 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutKILL = { 3, 4, 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutKILL = kill ∪ InKILL

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutKILL = { 2, 6 } ∪ { 3, 4, 5, 7 }

Aggregate Kill Set

InKILL = { }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

OutKILL = { 3, 4, 5, 7 }

InKILL = { 3, 4, 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

OutKILL = { 2, 3, 4, 5, 6, 7 }

Aggregate Kill Set

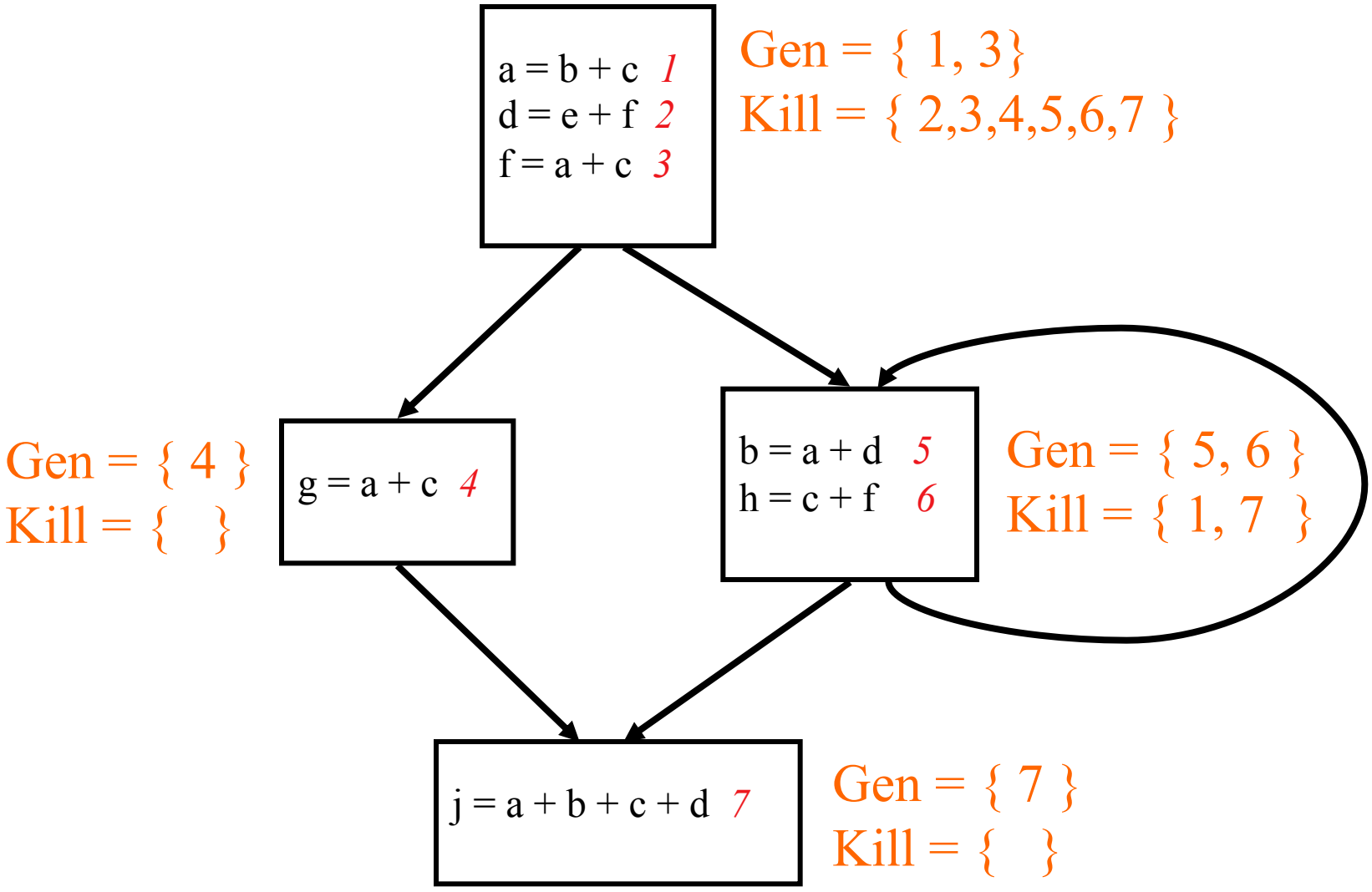
KILL = { 2, 3, 4, 5, 6, 7 }

a = b + c *1*
gen = { 1 }
kill = { 3, 4, 5, 7 }

d = e + f *2*
gen = { 2 }
kill = { 5, 7 }

f = a + c *3*
gen = { 3 }
kill = { 2, 6 }

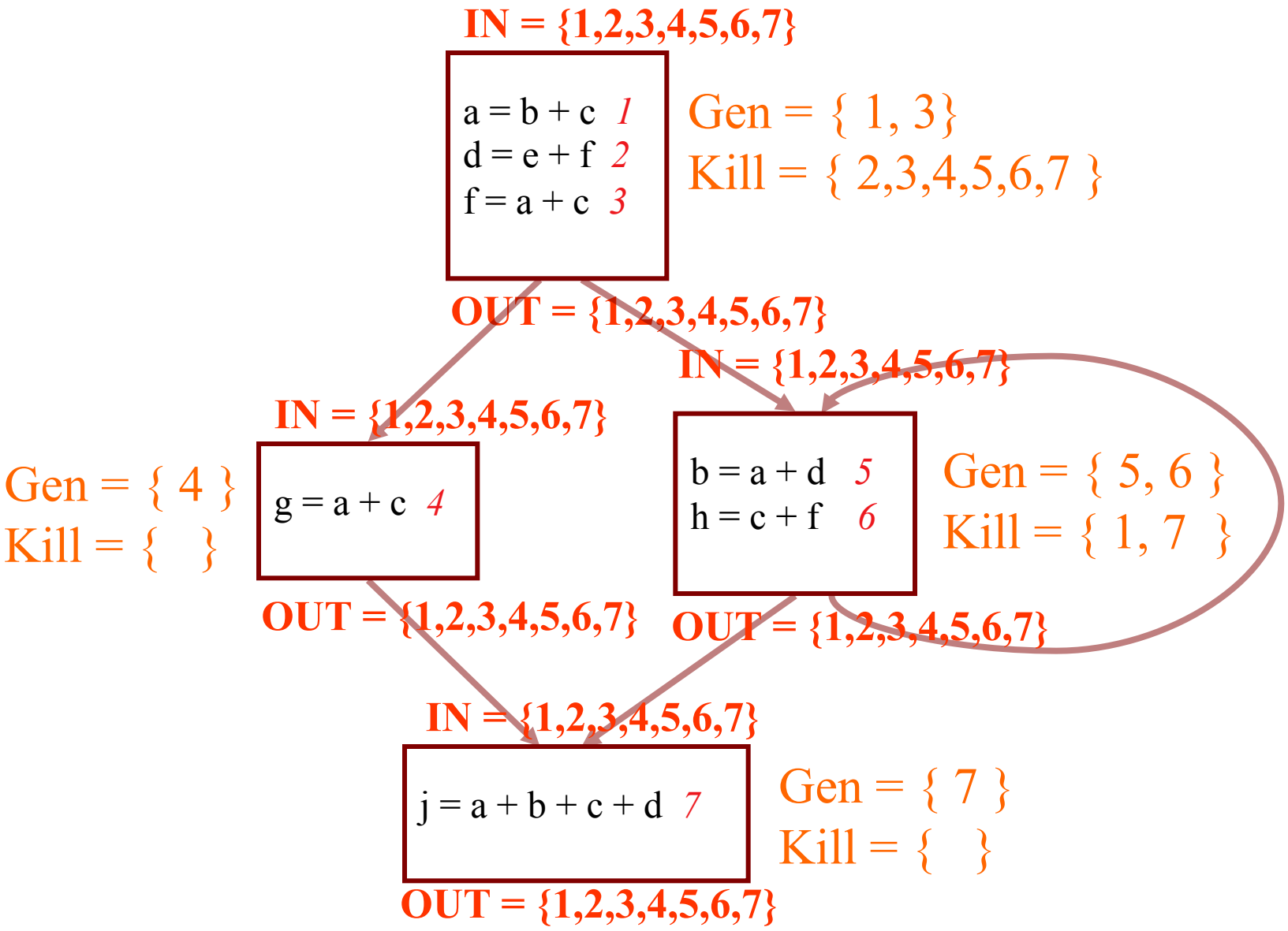
Aggregate Gen and Kill Sets



Algorithm for Available Expression

- Assign a Number to each Expression in the Program
- Compute Gen and Kill Sets for each Instruction
- Computer **Aggregate** Gen and Kill Sets for each Basic Block
- Initialize Available Set at each Basic Block as follows:
  - IN and OUT as the Entire Set (Universe of the set of expressions)
  - Exception: IN = ∅ for first Basic Block

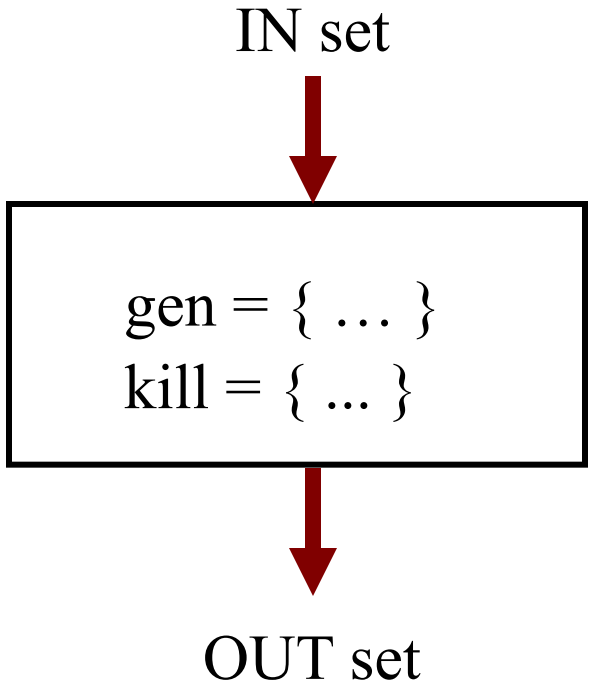
Aggregate Gen and Kill Sets



Algorithm for Available Expression

- Assign a Number to each Expression in the Program
- Compute Gen and Kill sets for each Instruction
- Compute Aggregate Gen and Kill sets for each Basic Block
- Initialize Available Set at each Basic Block as follows:
  - IN and OUT as the Entire Set (Universe of the set of expressions)
  - Exception: IN = ∅ for first Basic Block
- Iteratively propagate available expression set over the CFG

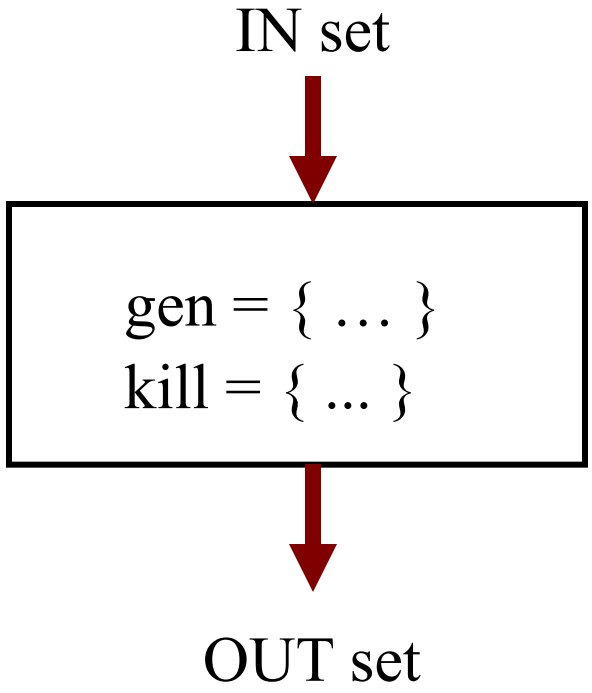
Propagate Available Expression Set



OUT =

Propagate Available Expression Set

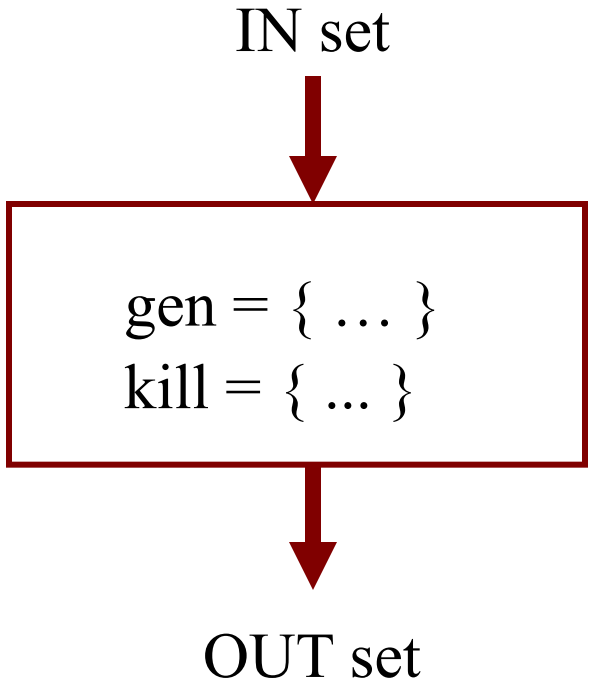
- If the expression is generated (in the Gen set) then it is available at the end
  - should in the OUT set



OUT = gen

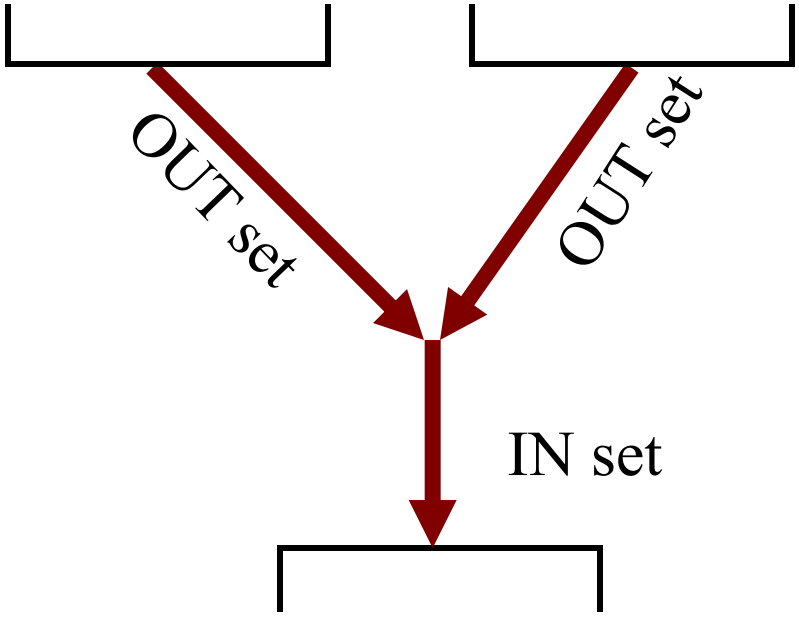
Propagate Available Expression Set

- If the expression is generated (in the Gen set) then it is available at the end
  - should in the OUT set
- Any expression available at the input (in the IN set) and not killed should be available at the end



$$OUT = gen \cup (IN - kill)$$

Propagate Available Expression Set

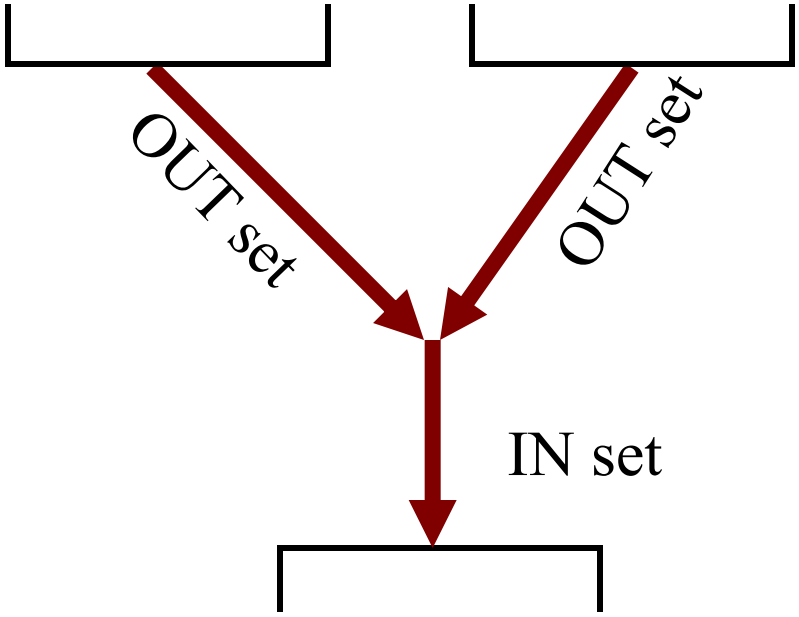


IN =

OUT = gen ∪ (IN - kill)

Propagate Available Expression Set

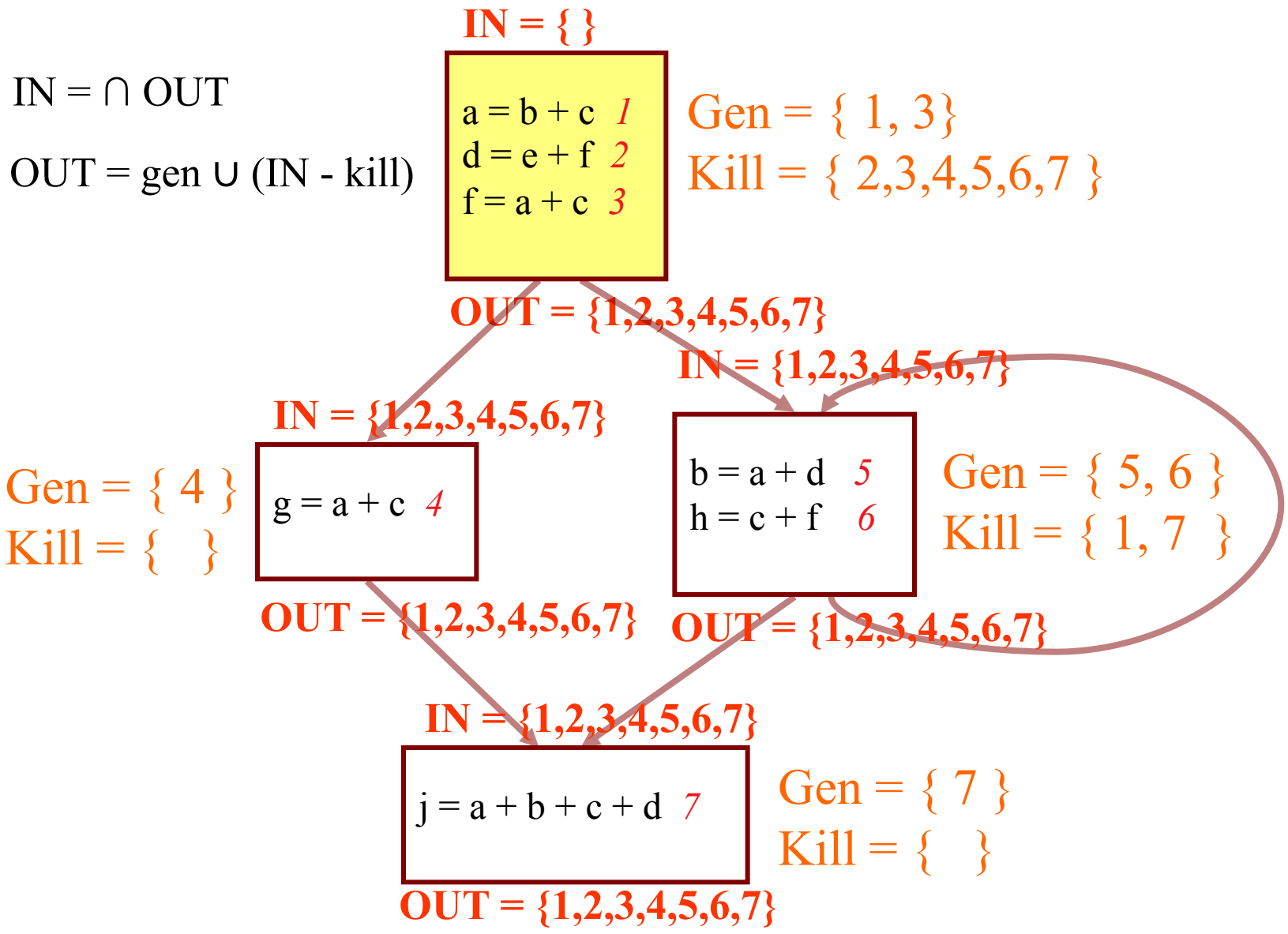
- Expression is available only if it is available in *All Input Paths*



$$IN = \bigcap OUT$$

$$OUT = gen \cup (IN - kill)$$

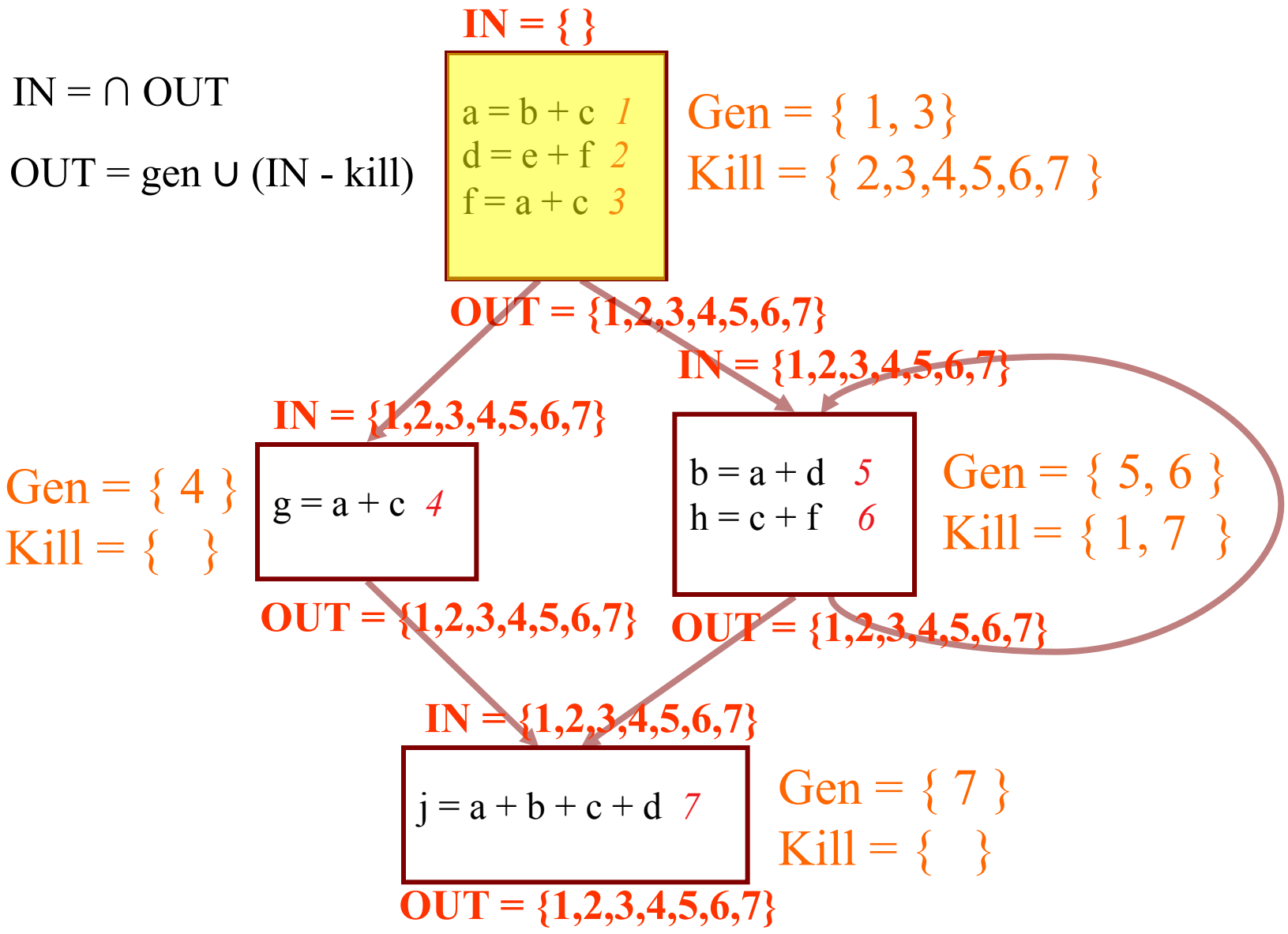
Aggregate Gen and Kill Sets



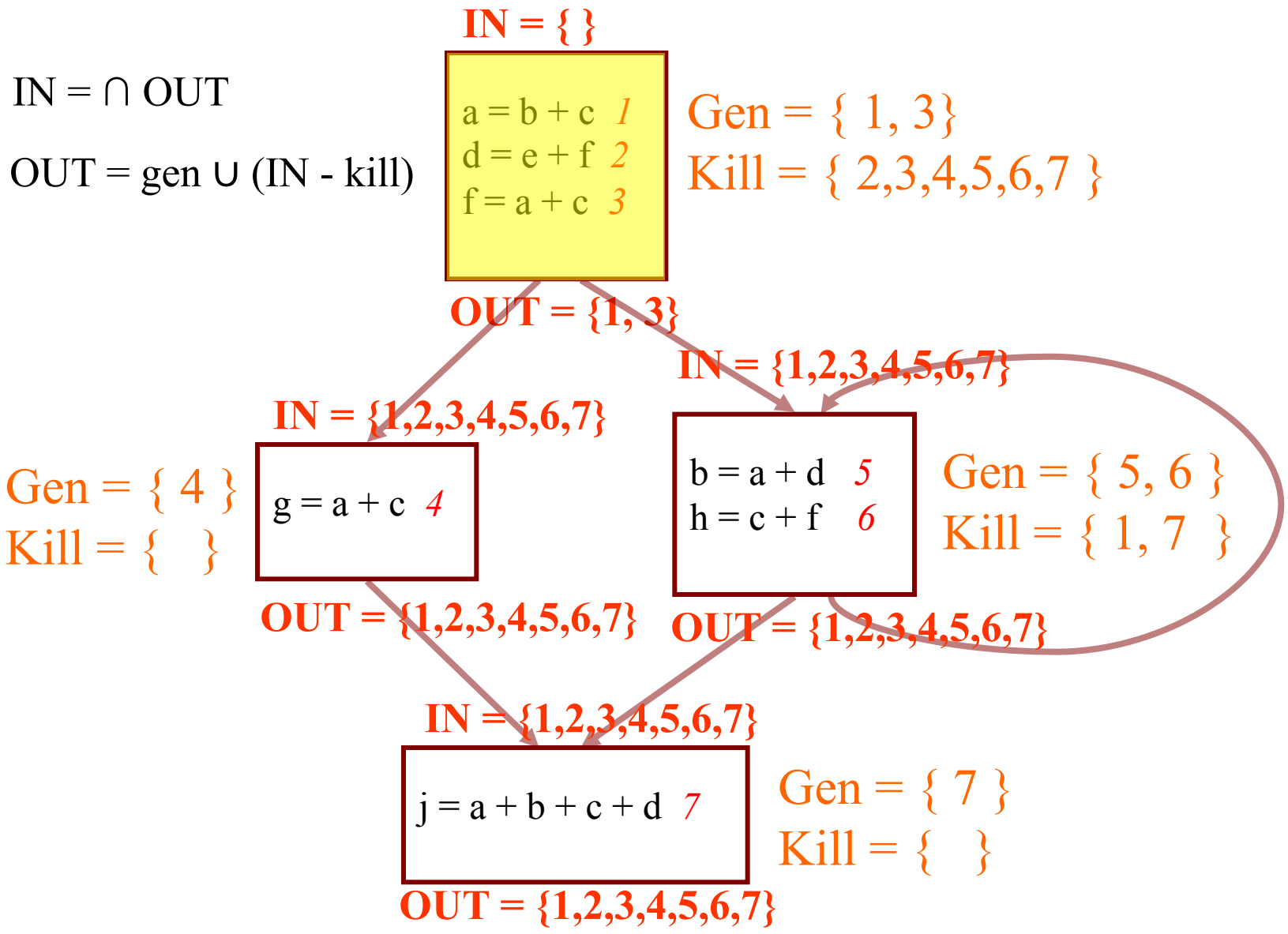
Aggregate Gen and Kill Sets

IN = ∩ OUT

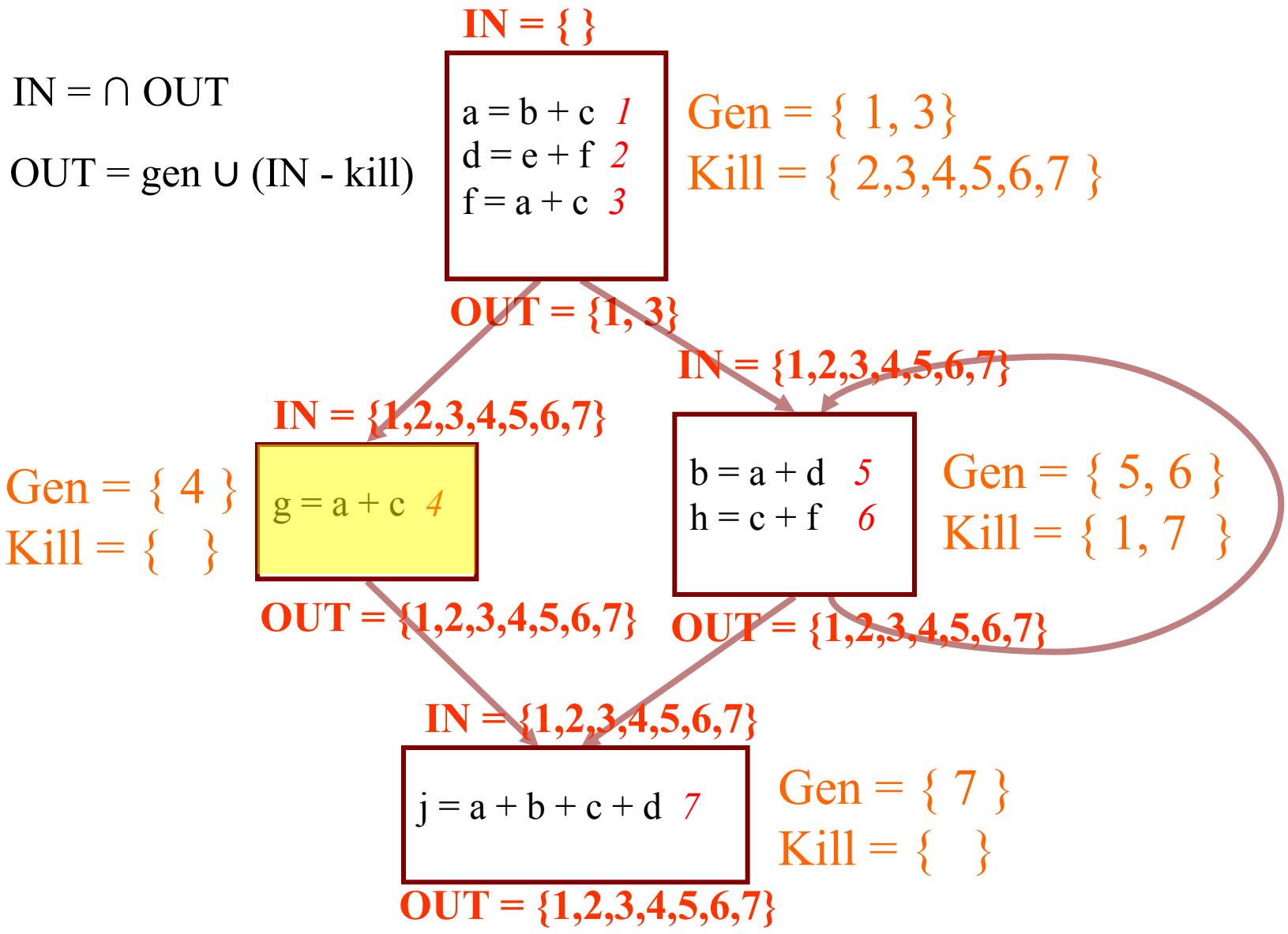
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets



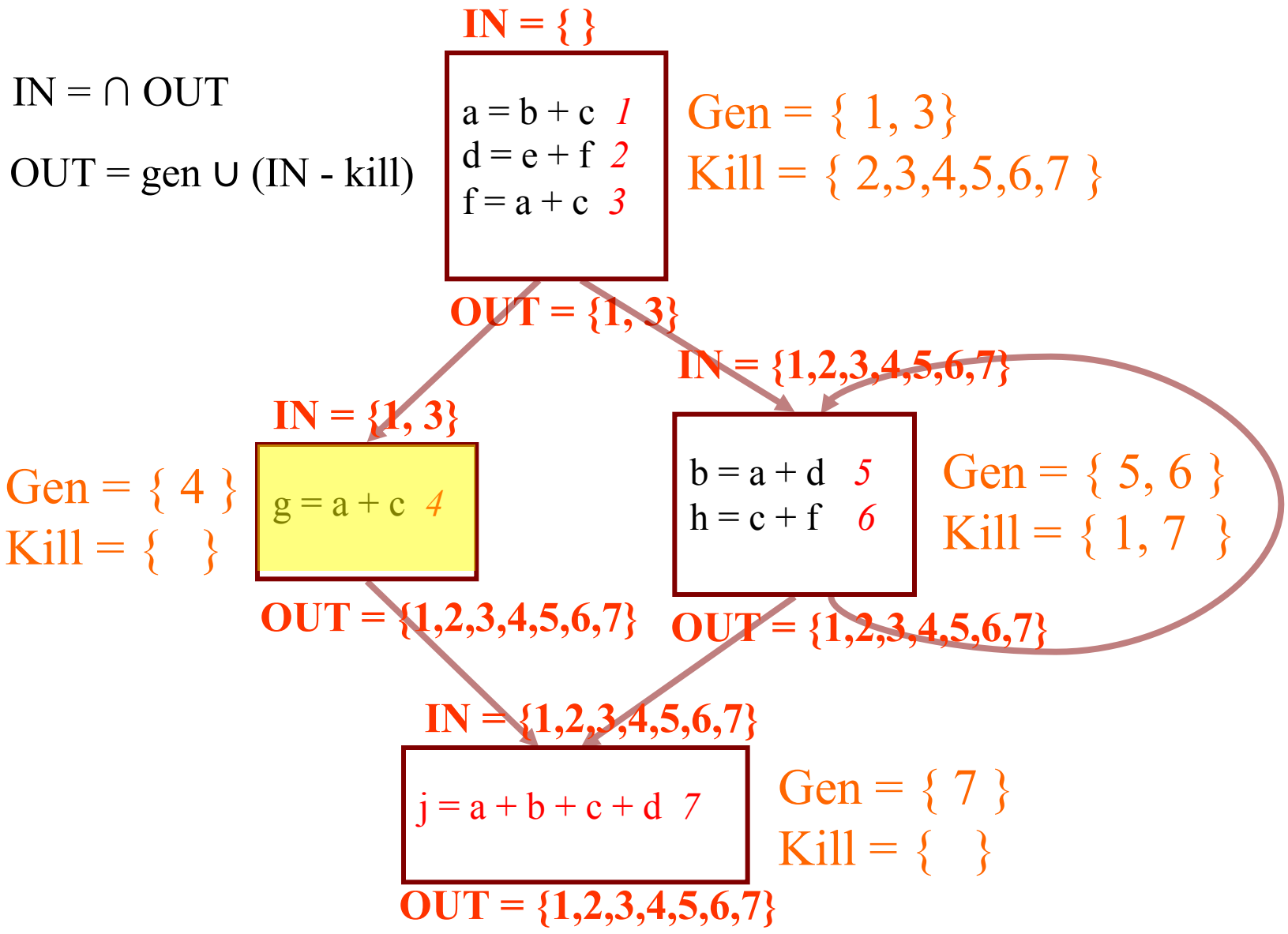
Aggregate Gen and Kill Sets



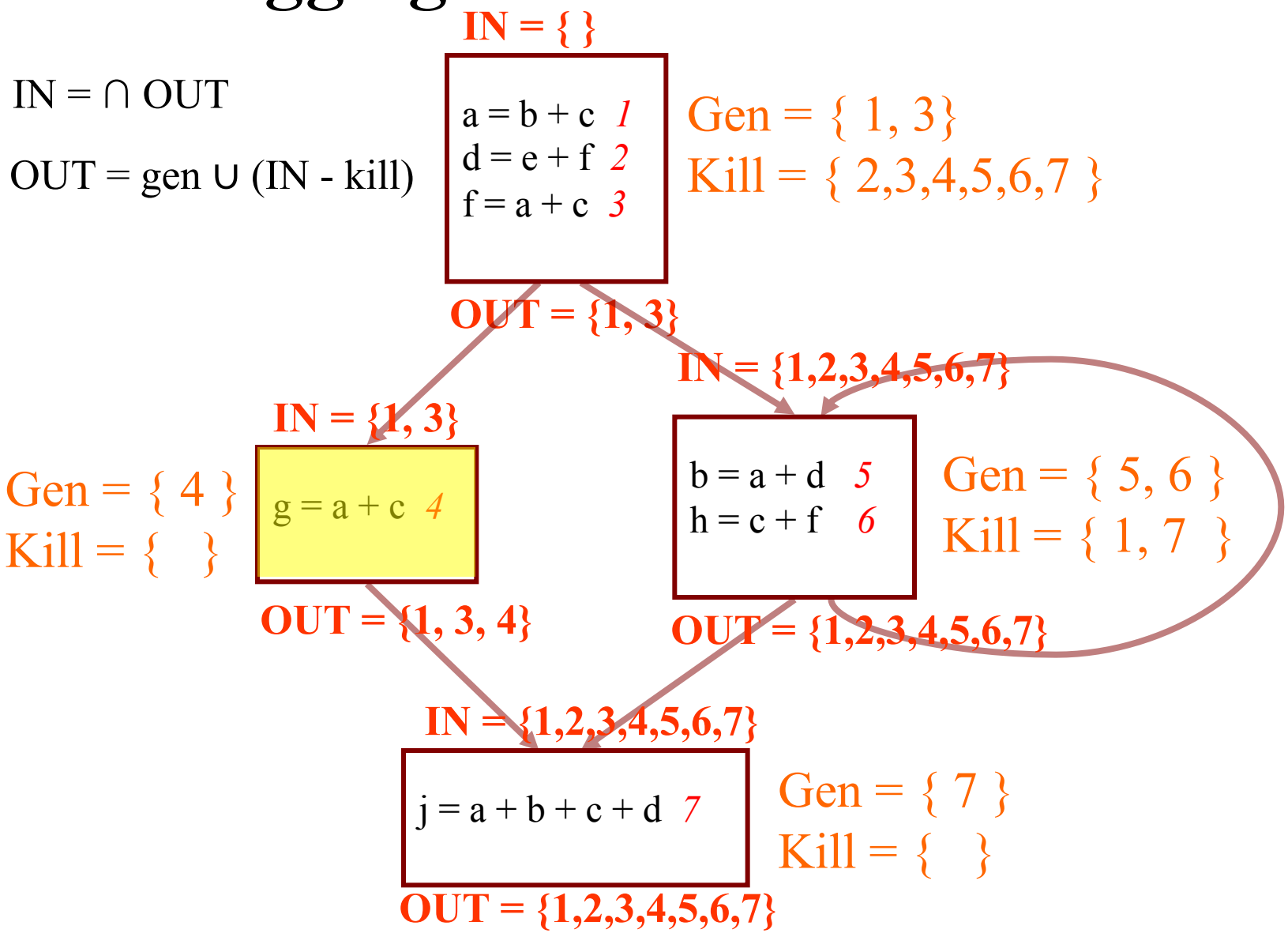
Aggregate Gen and Kill Sets

IN = ∩ OUT

OUT = gen ∪ (IN - kill)



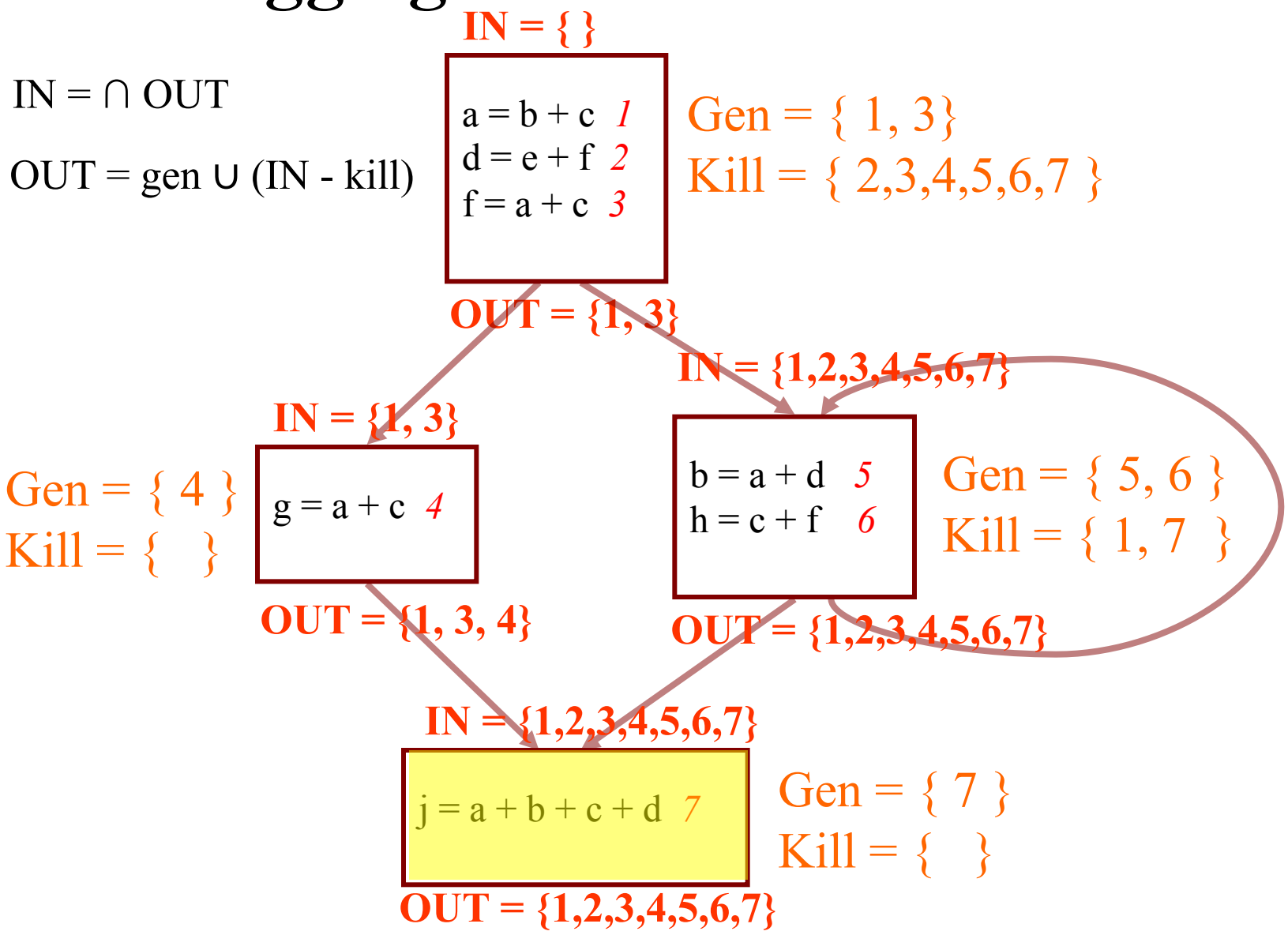
Aggregate Gen and Kill Sets



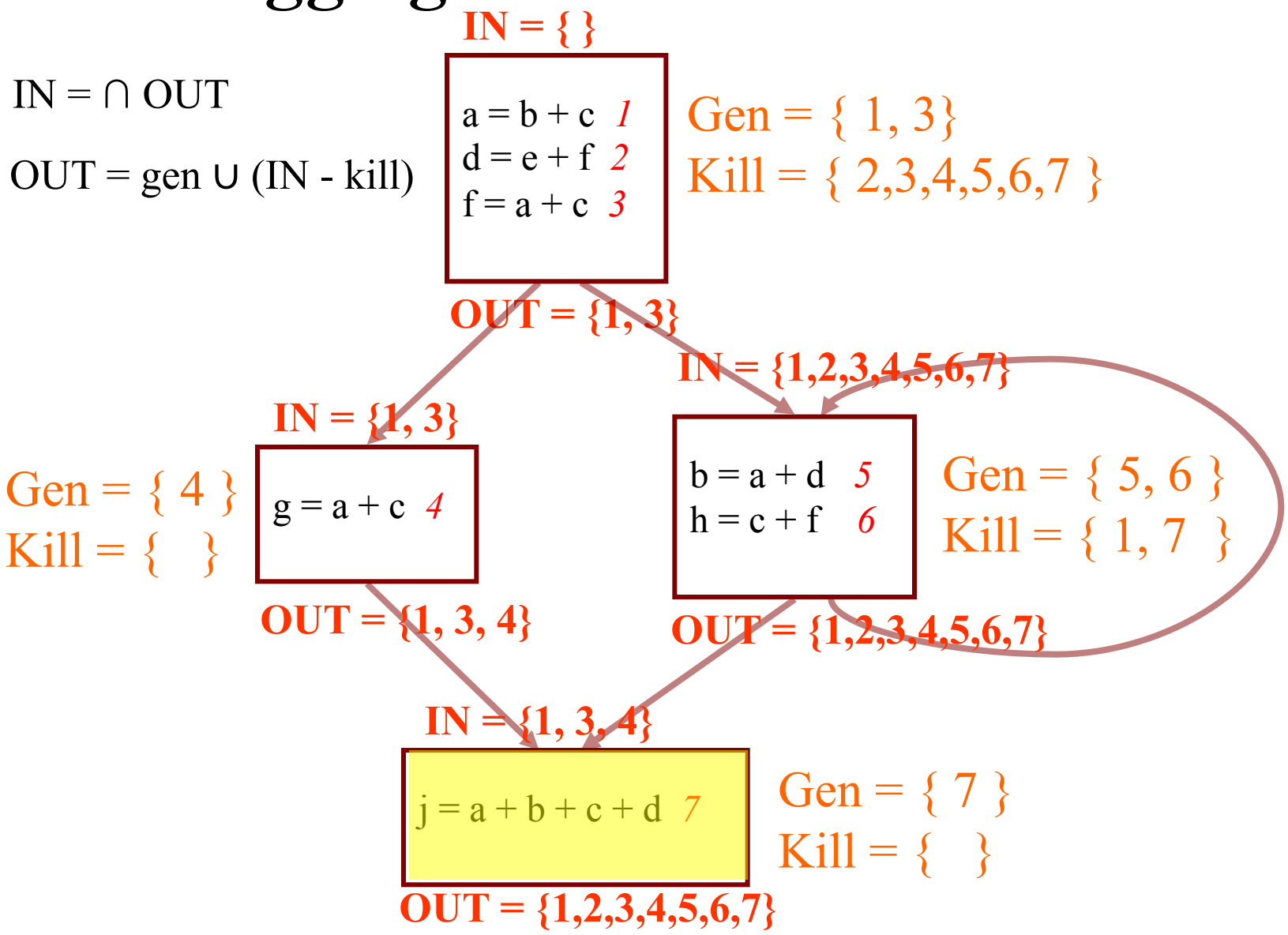
Aggregate Gen and Kill Sets

IN = ∩ OUT

OUT = gen ∪ (IN - kill)



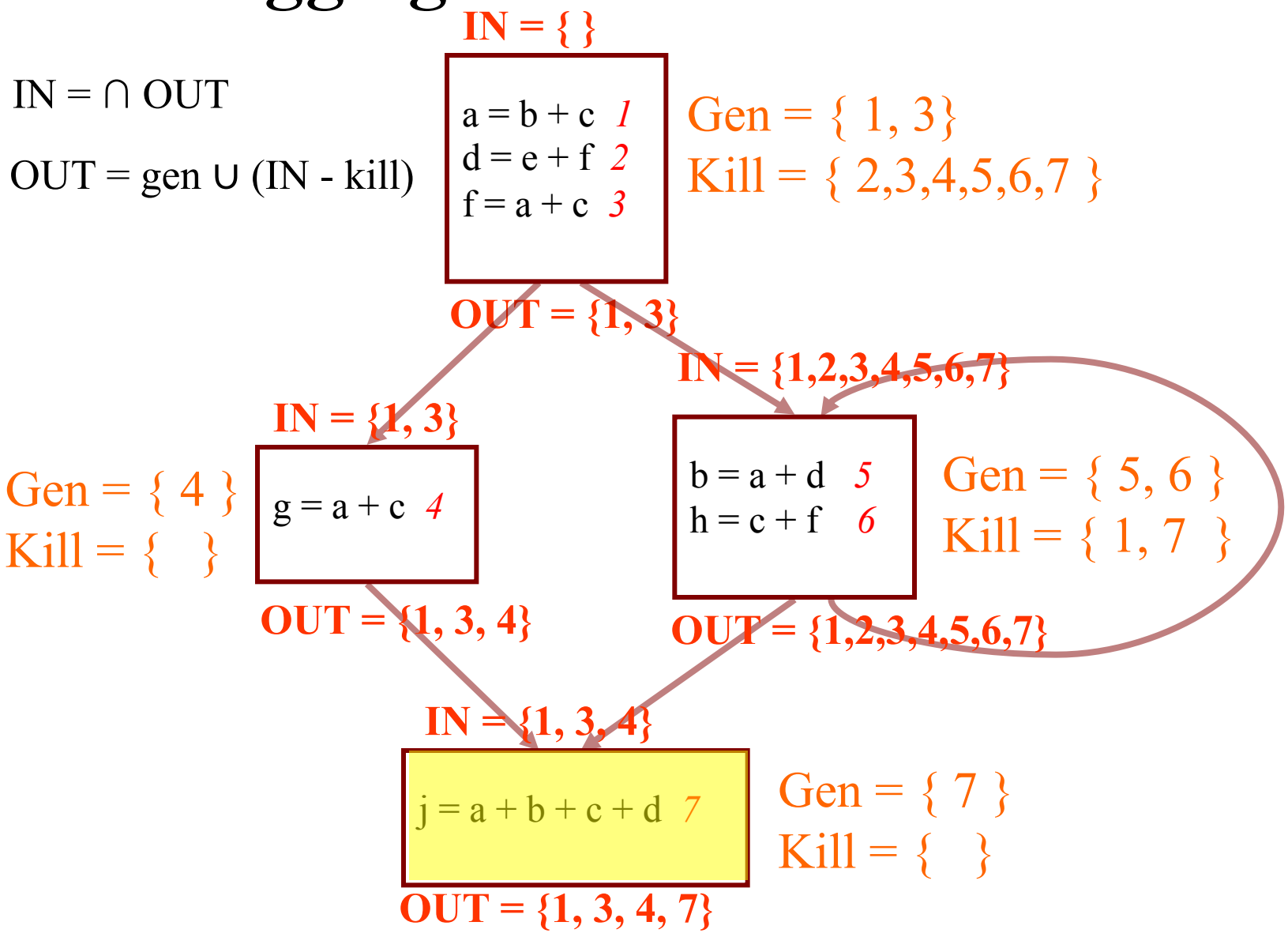
Aggregate Gen and Kill Sets



Aggregate Gen and Kill Sets

IN = ∩ OUT

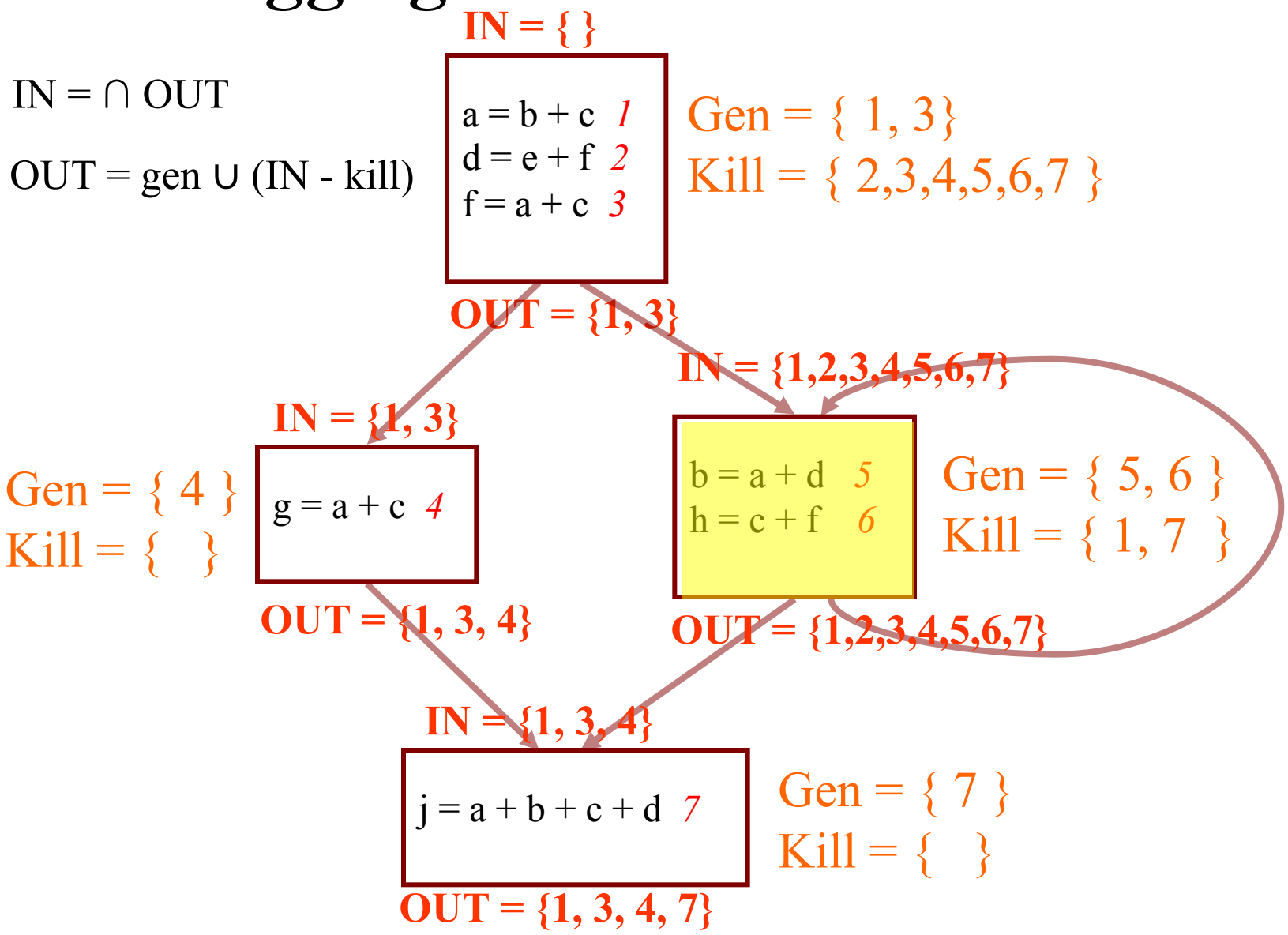
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

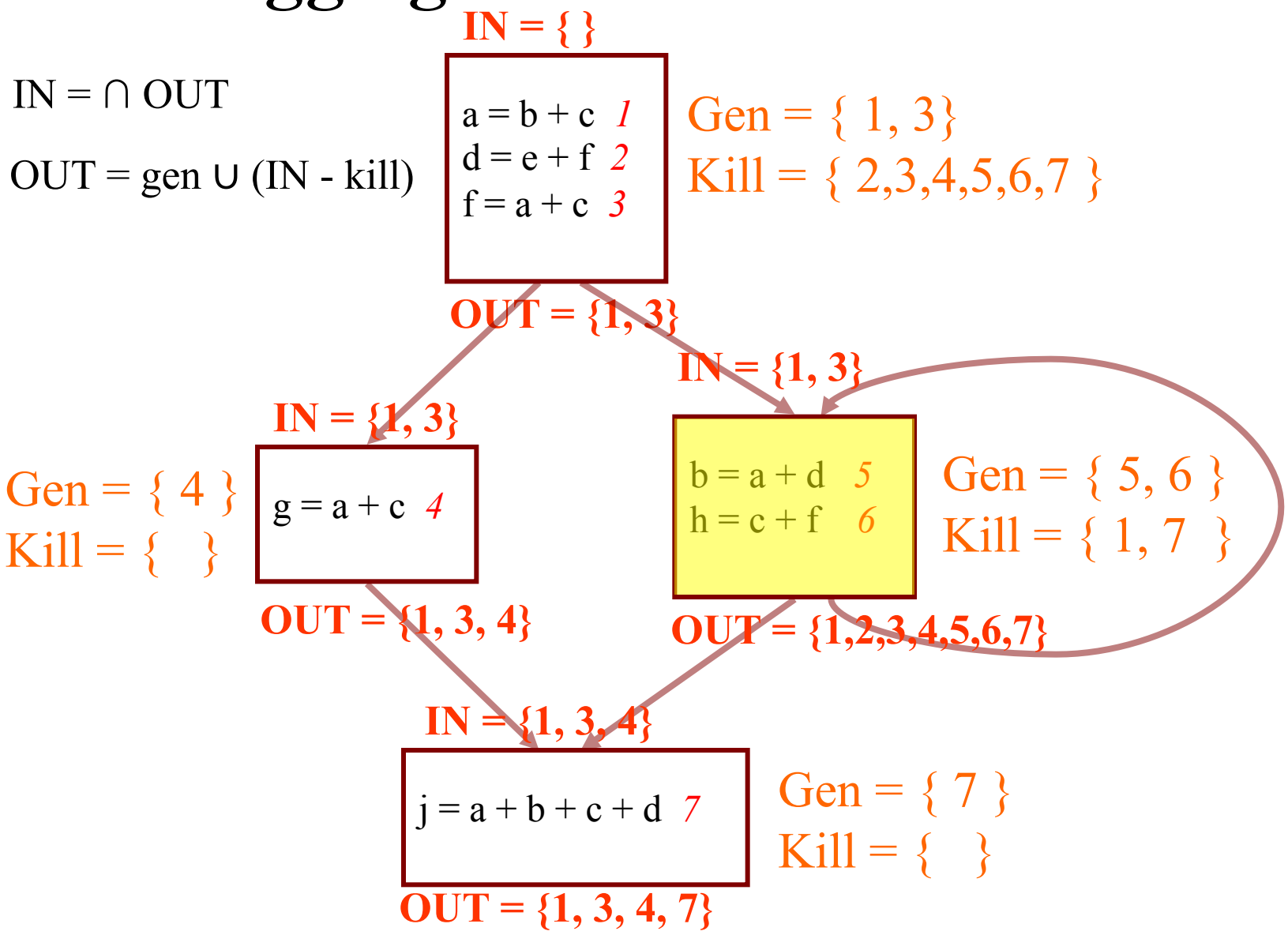
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

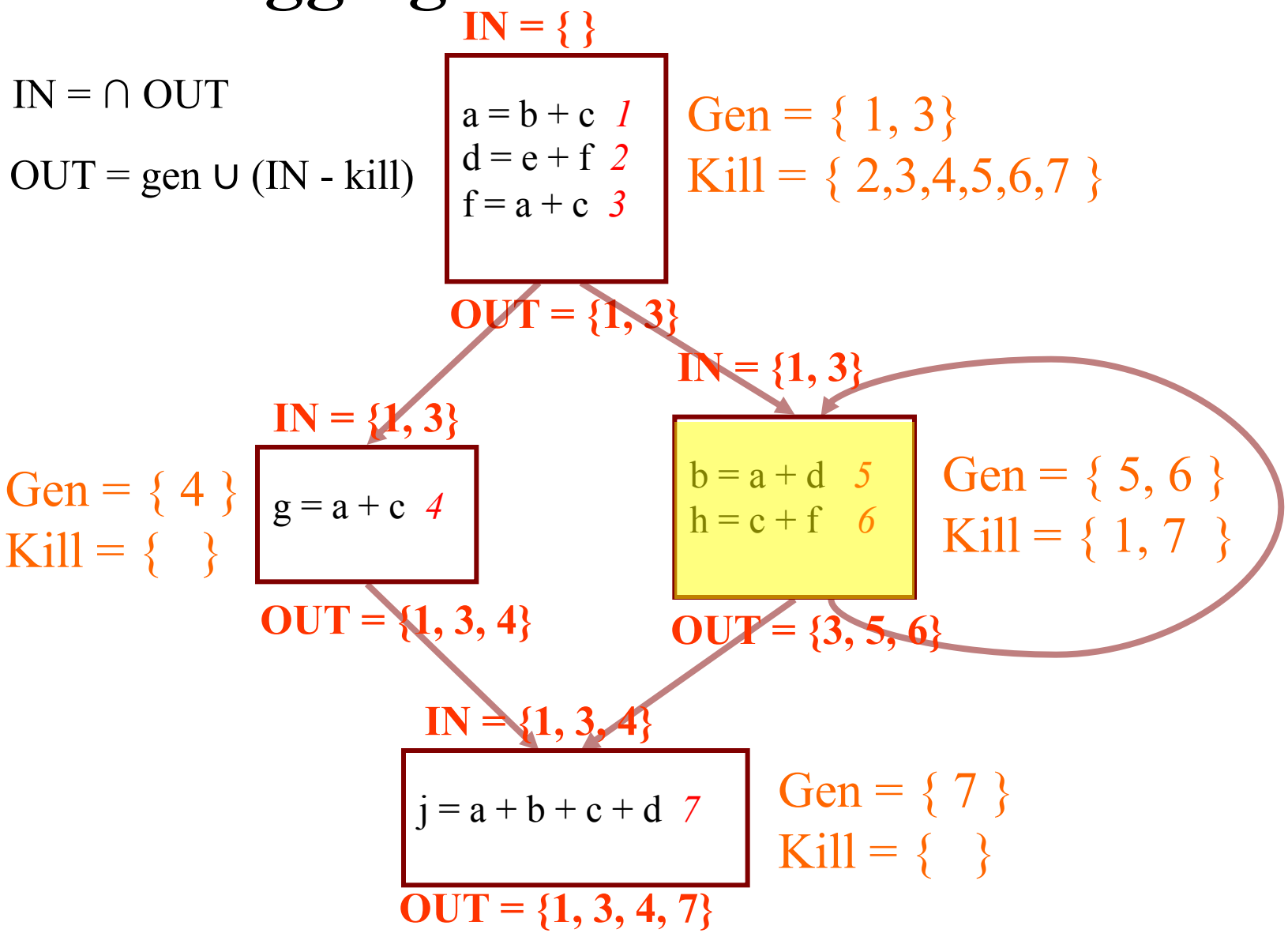
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

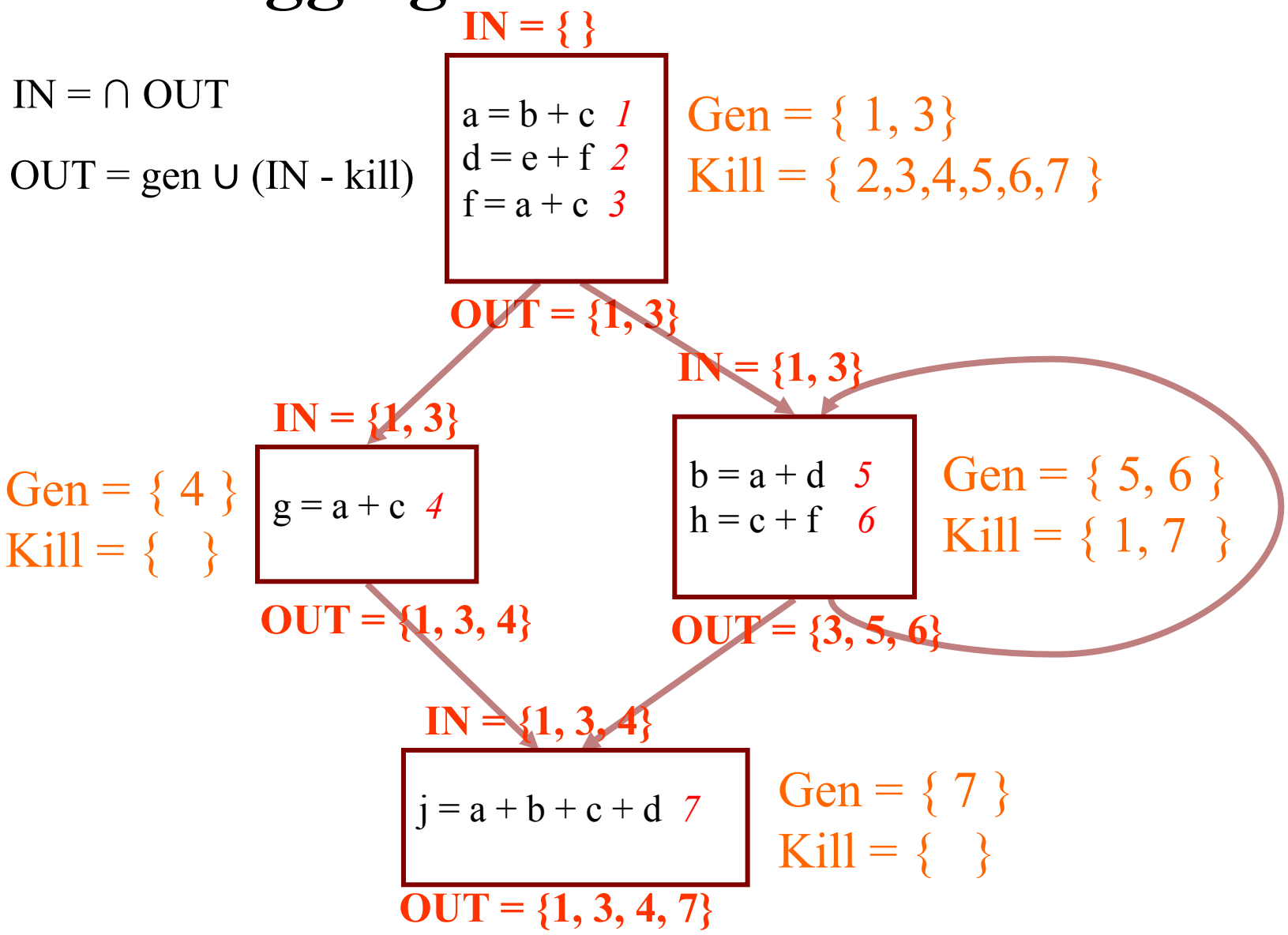
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

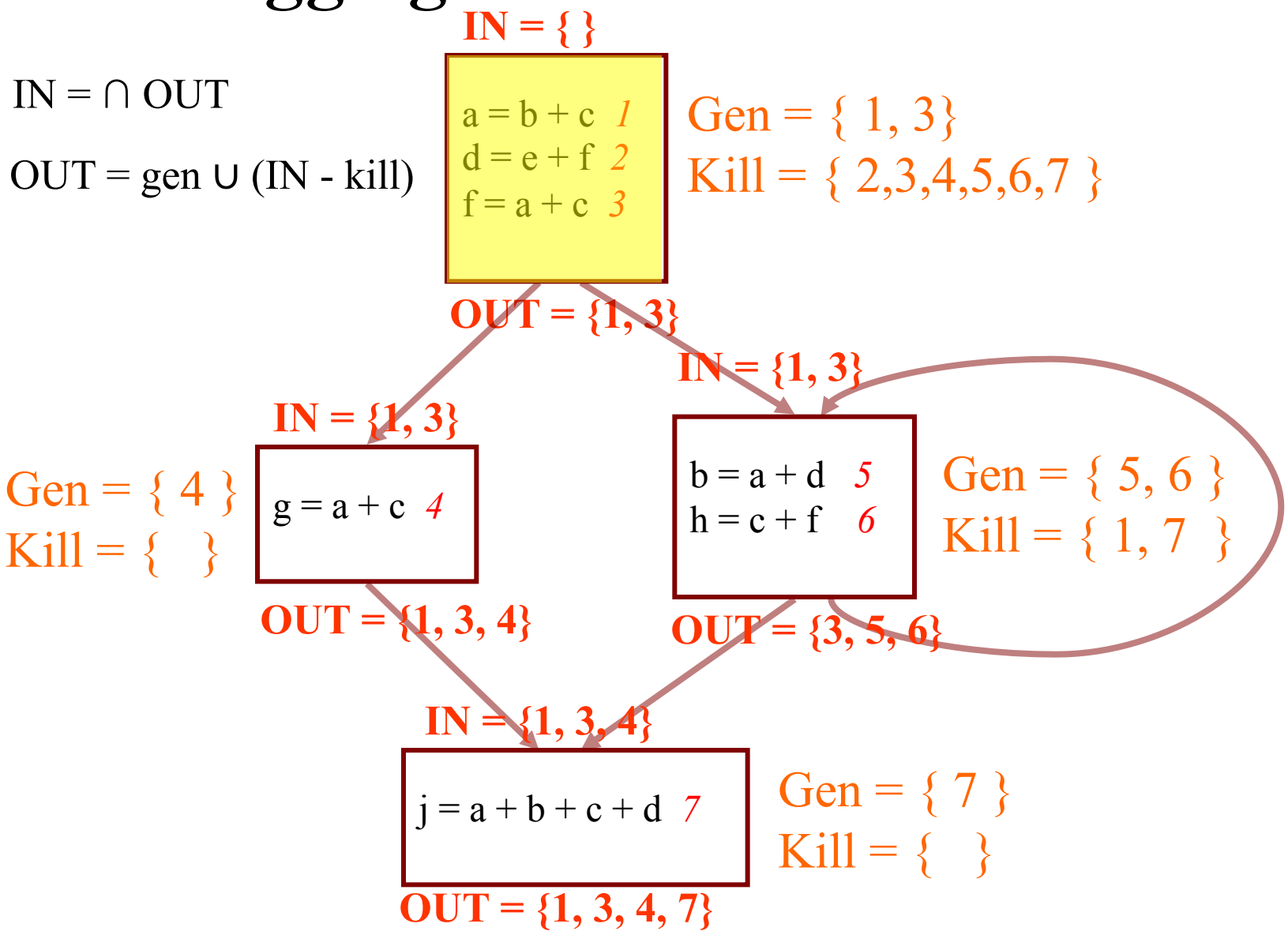
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

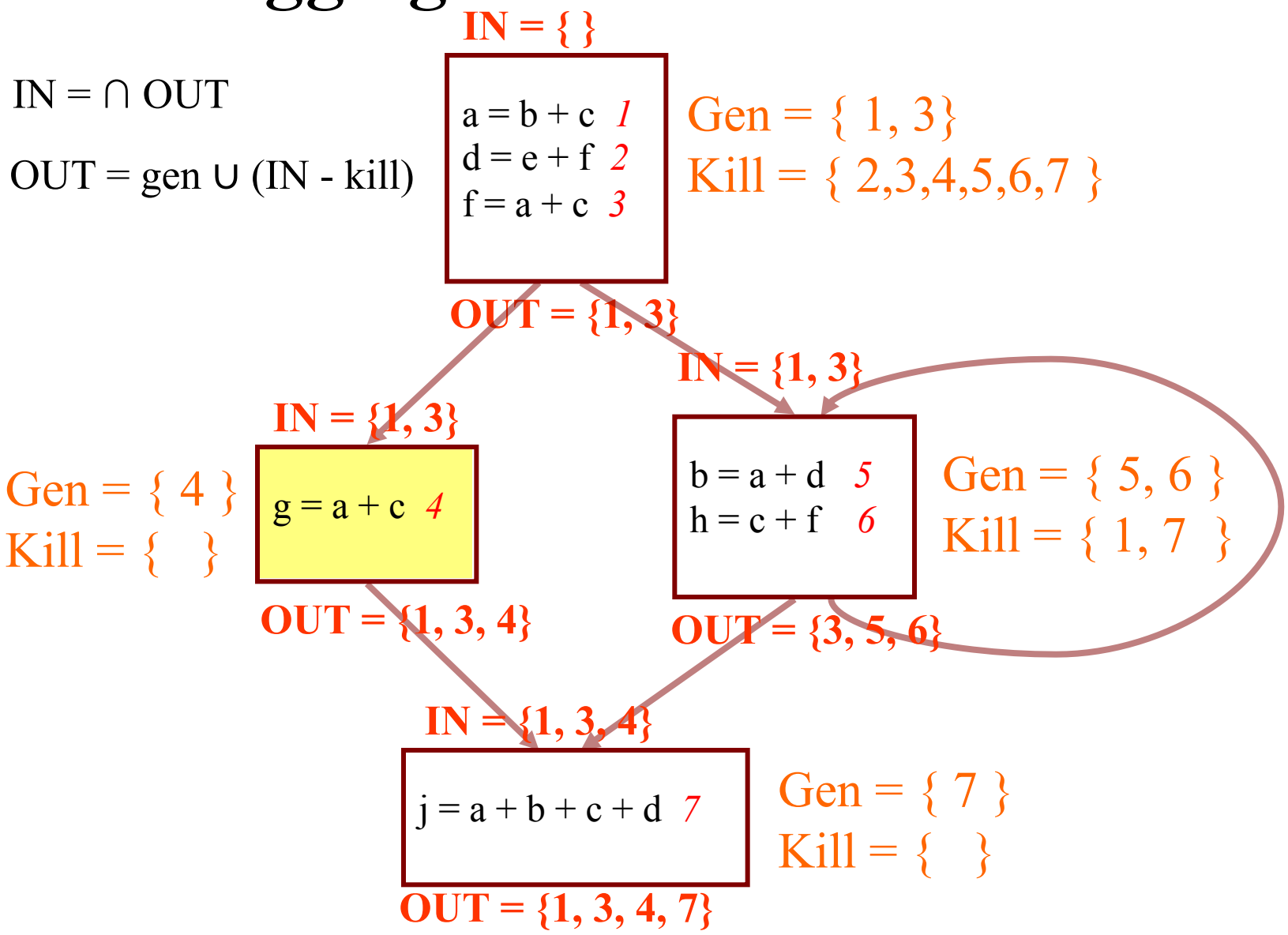
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

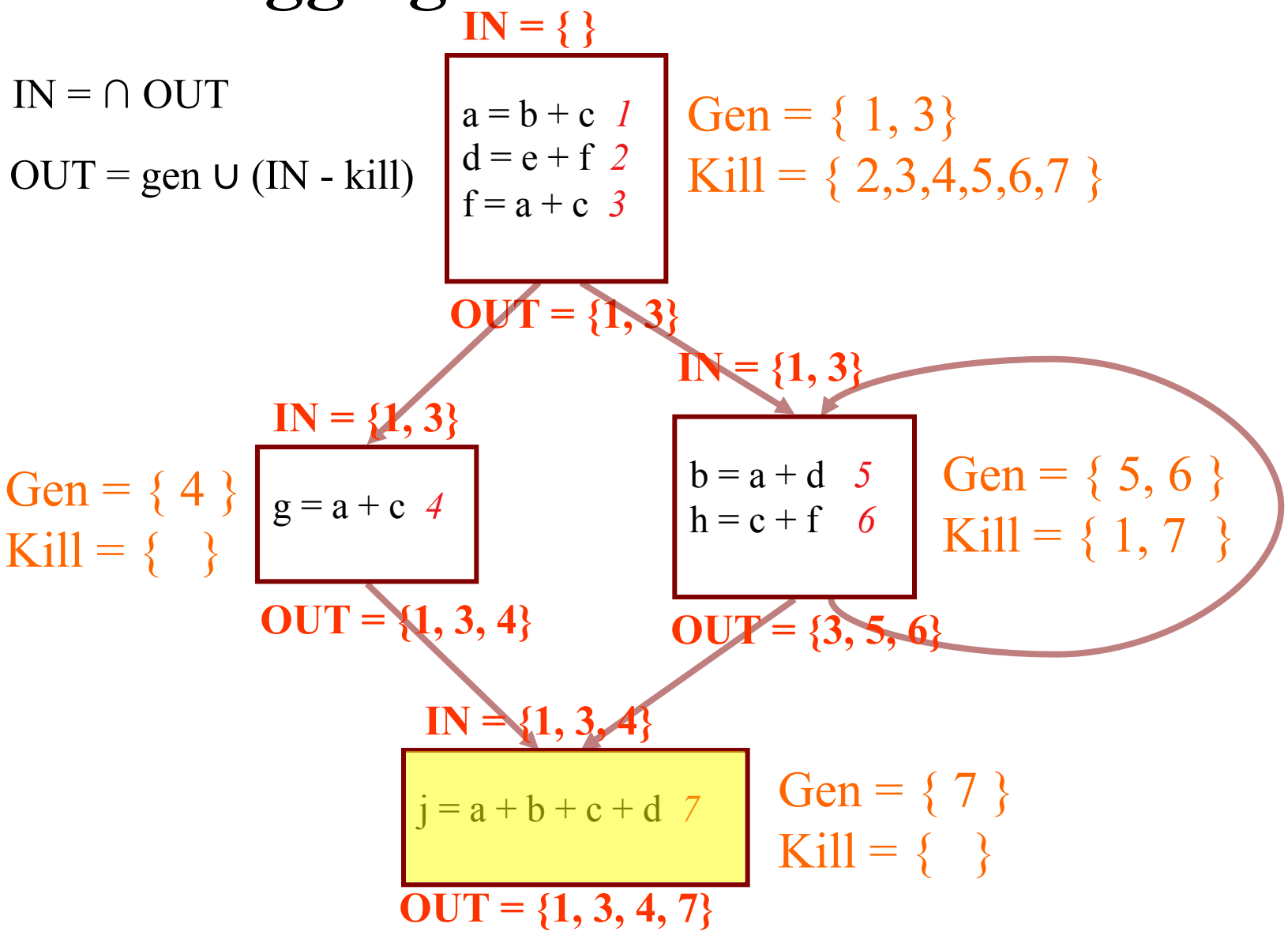
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

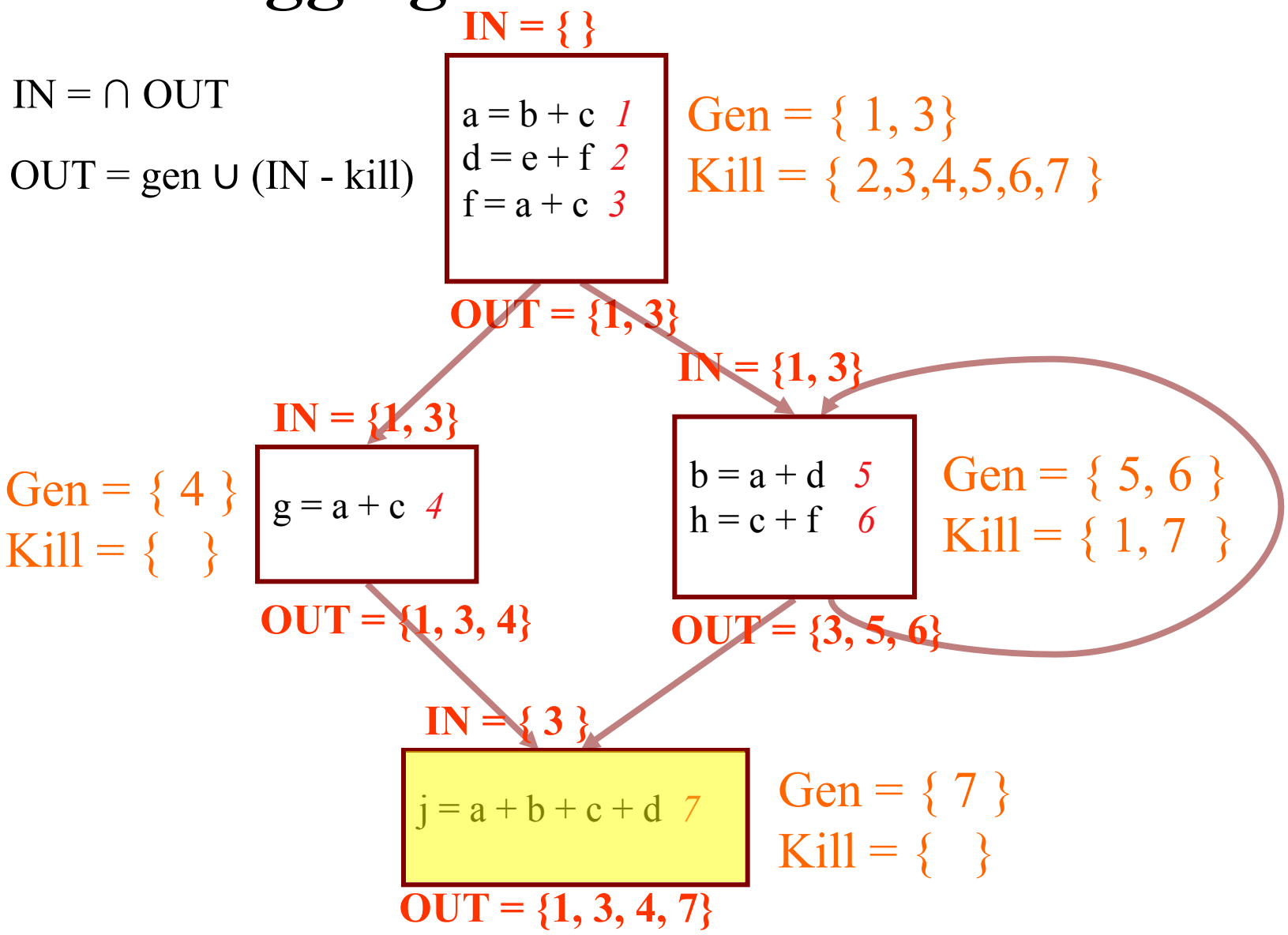
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

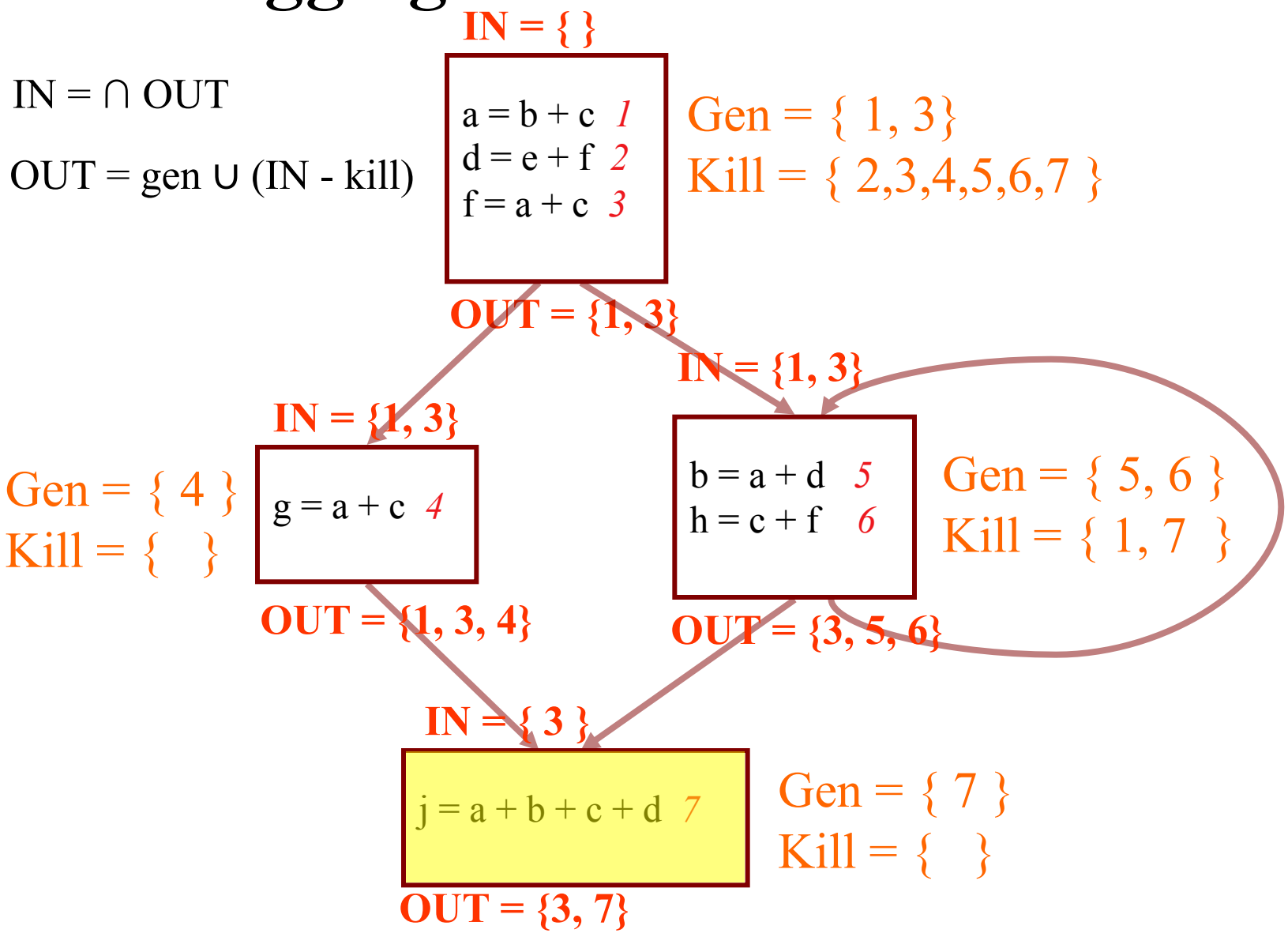
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

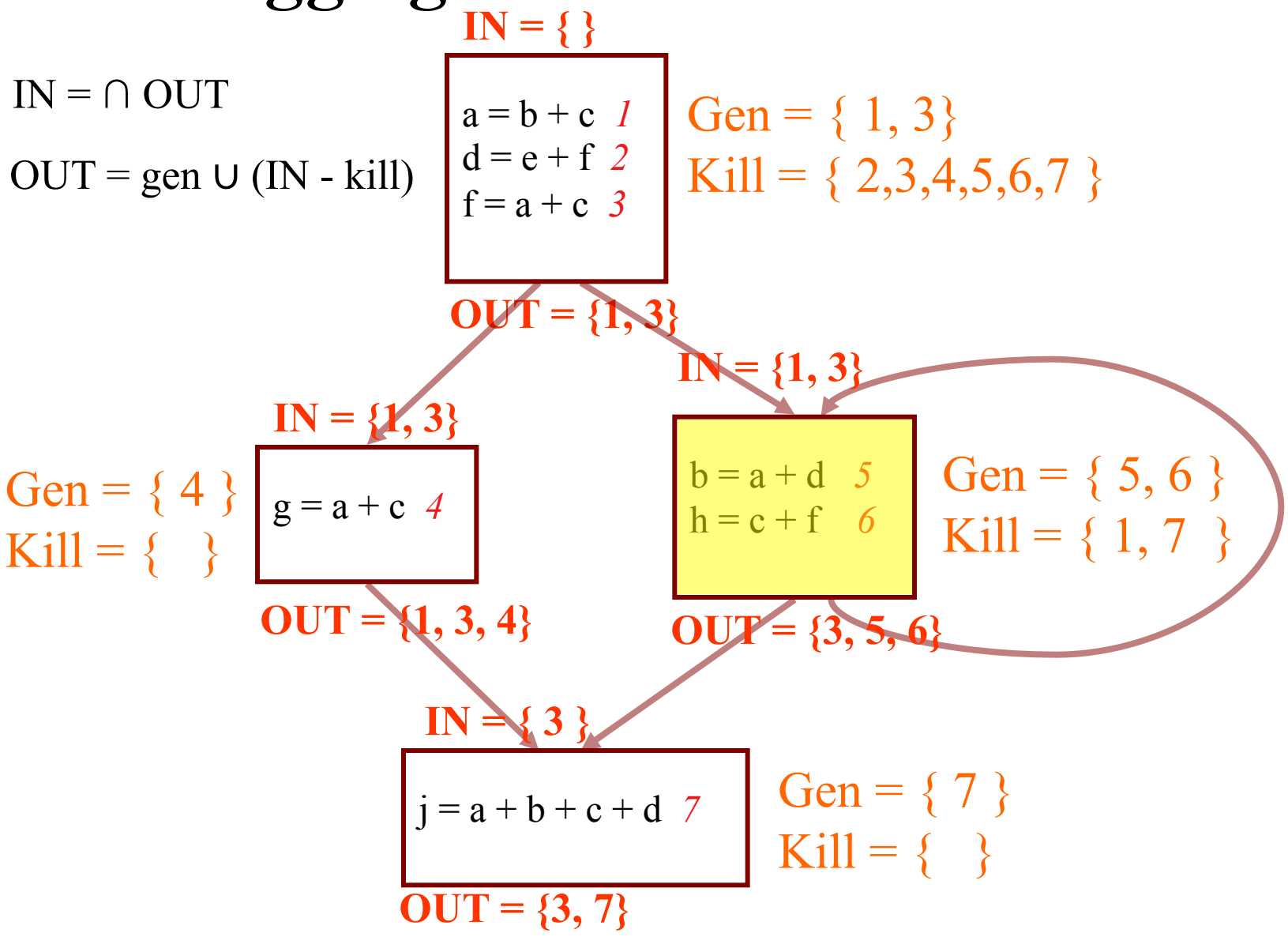
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

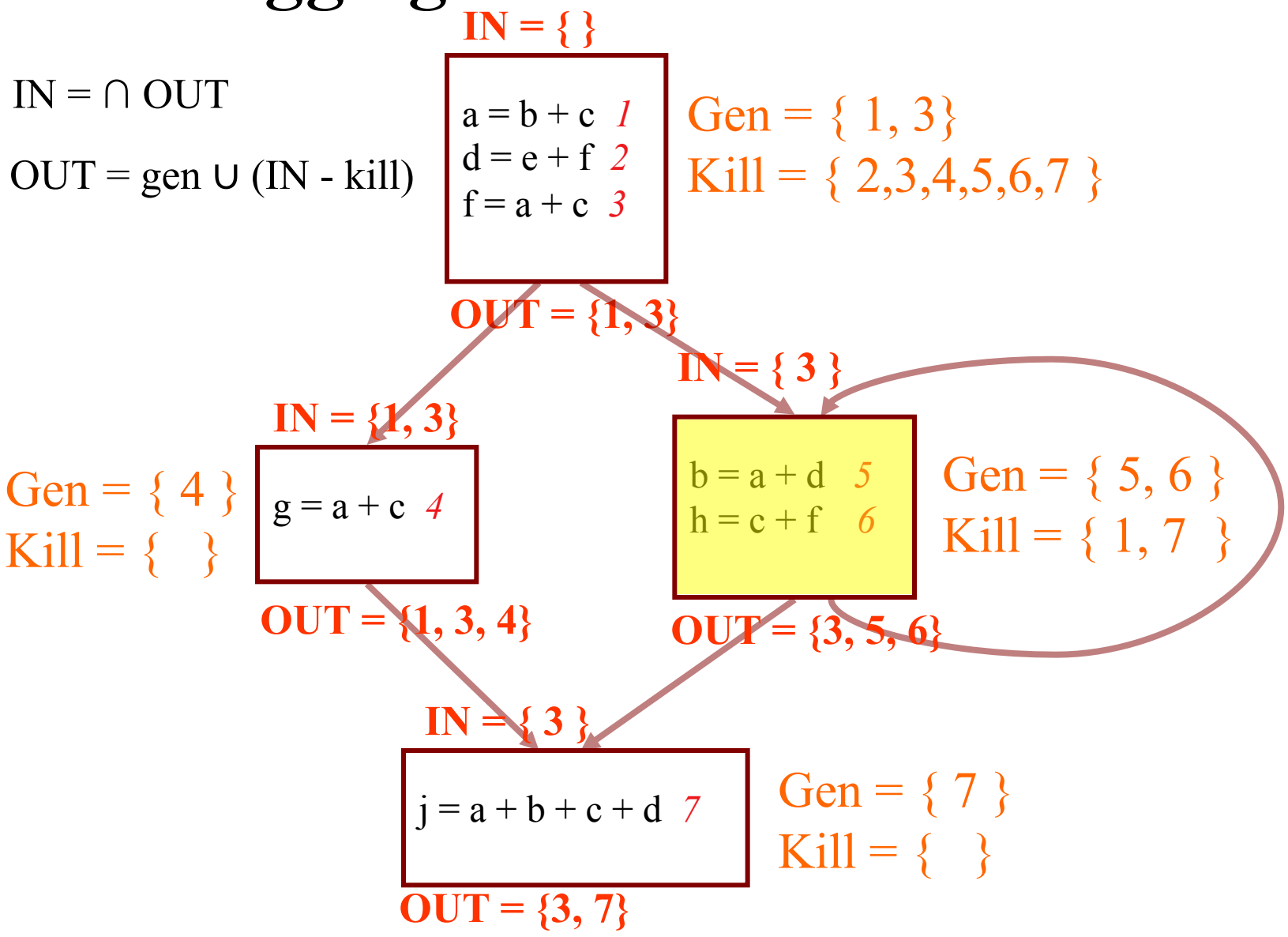
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

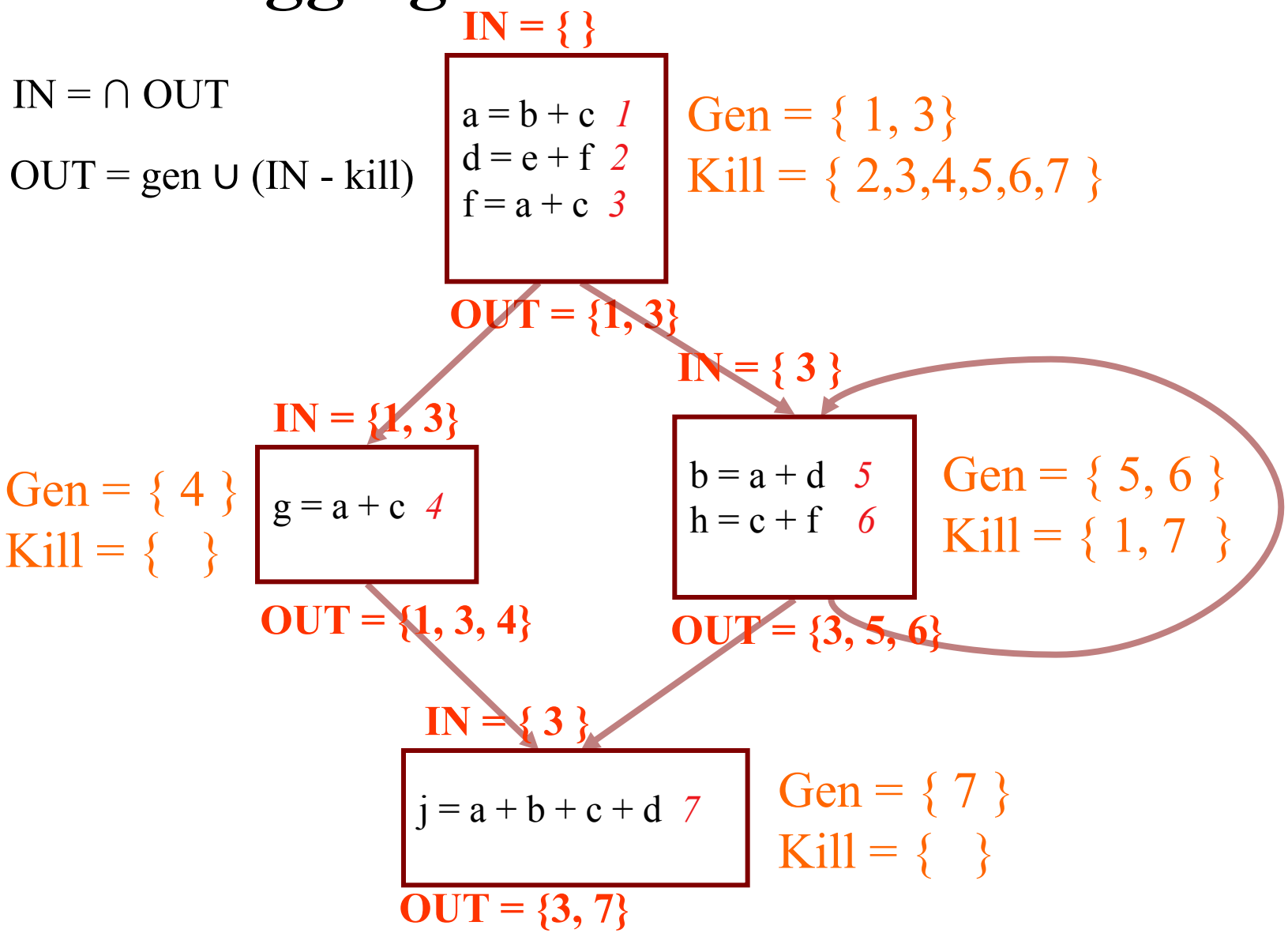
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

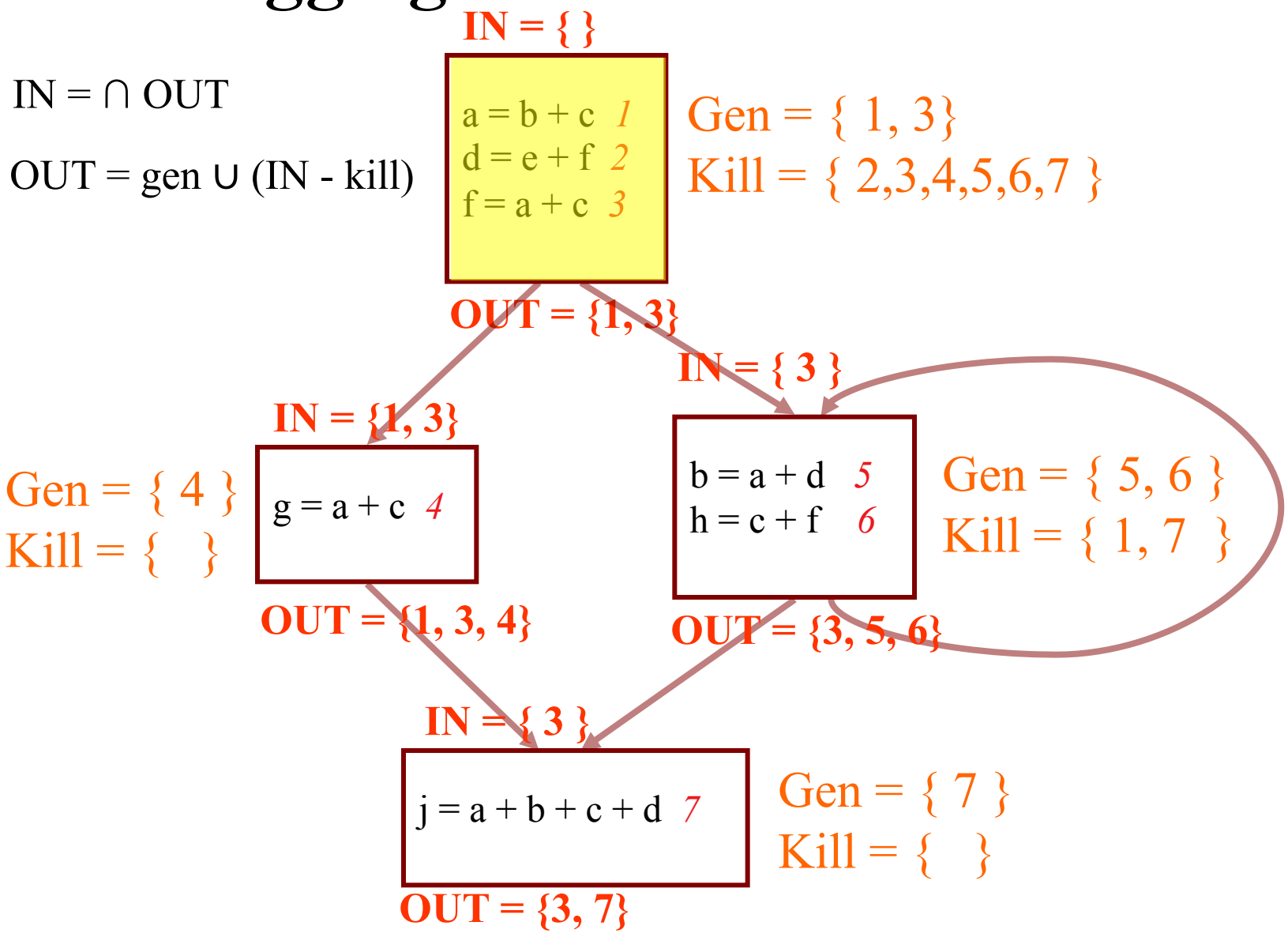
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

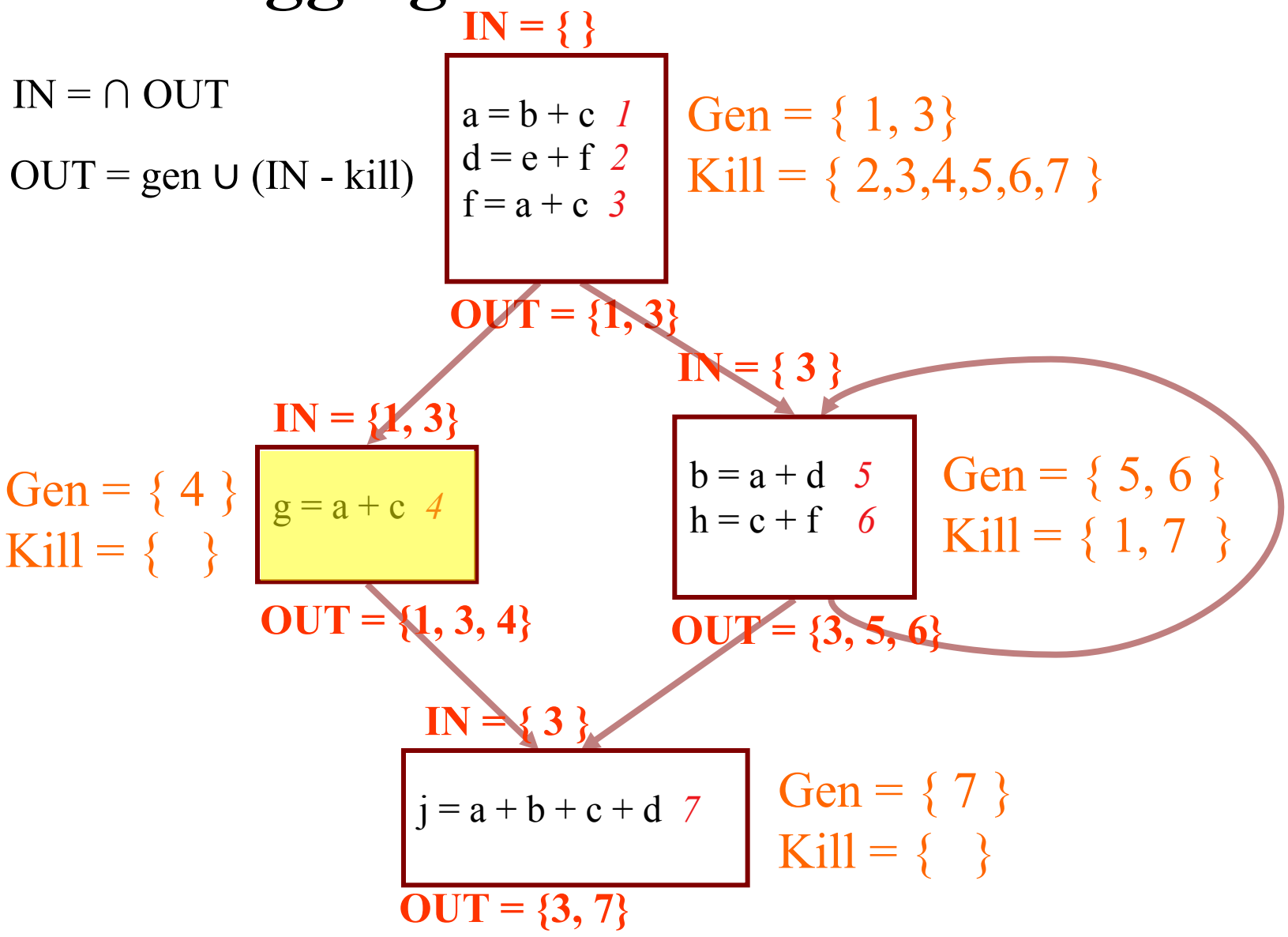
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

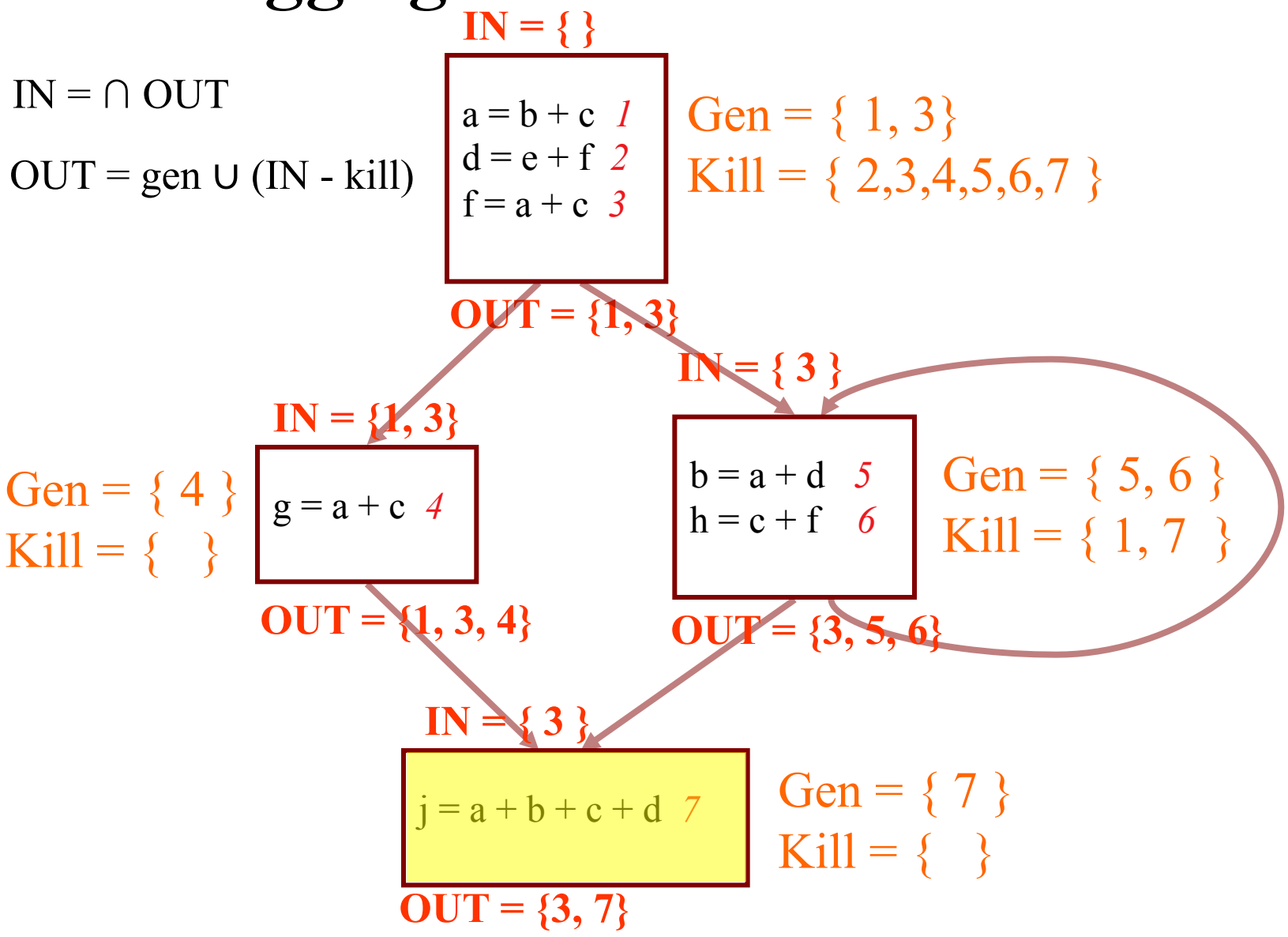
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

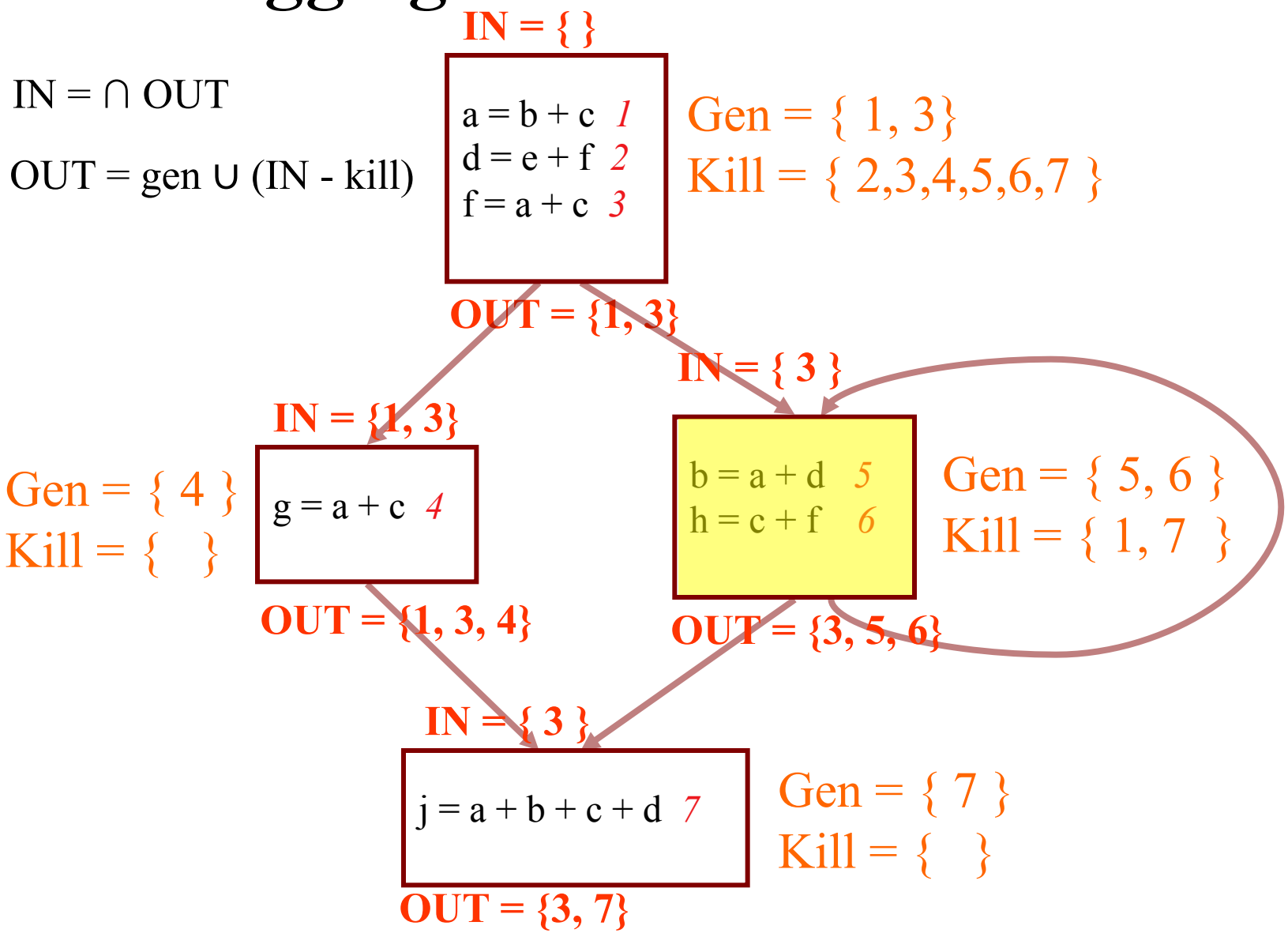
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

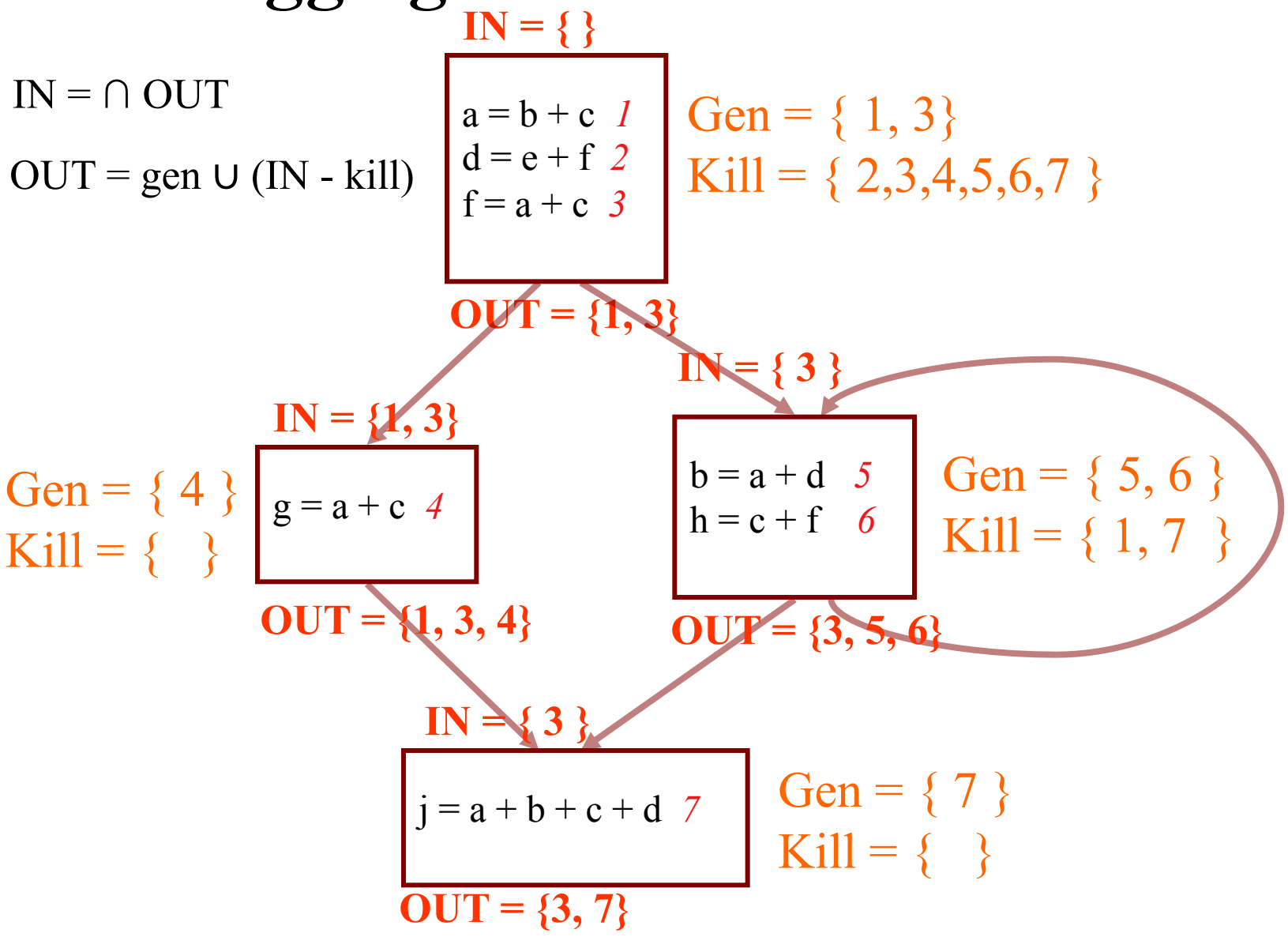
OUT = gen ∪ (IN - kill)



Aggregate Gen and Kill Sets

IN = ∩ OUT

OUT = gen ∪ (IN - kill)

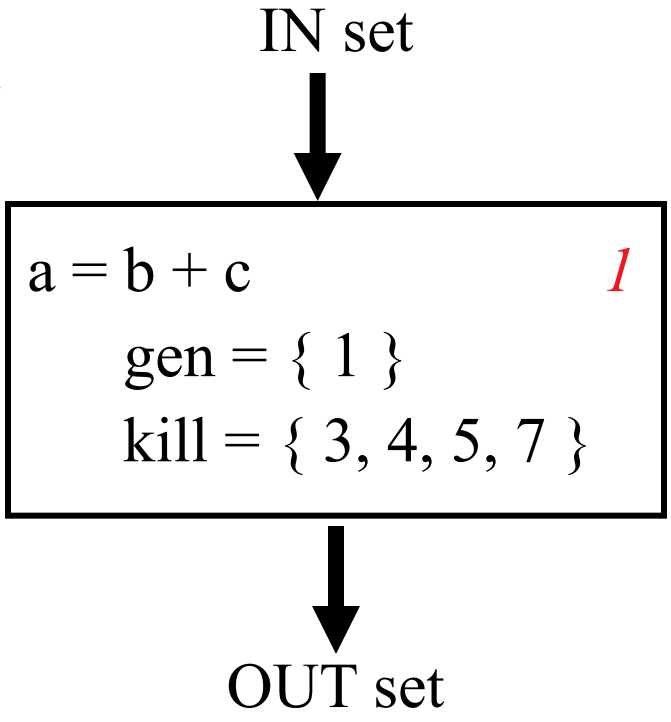


Algorithm for Available Expression

- Assign a Number to each Expression in the Program
- Calculate Gen and Kill Sets for each Instruction
- Calculate **aggregate** Gen and Kill Sets for each Basic Block
- Initialize Available Set at each Basic Block to be the entire set
- Iteratively propagate Available Expression set over the CFG
- Propagate within the Basic Block

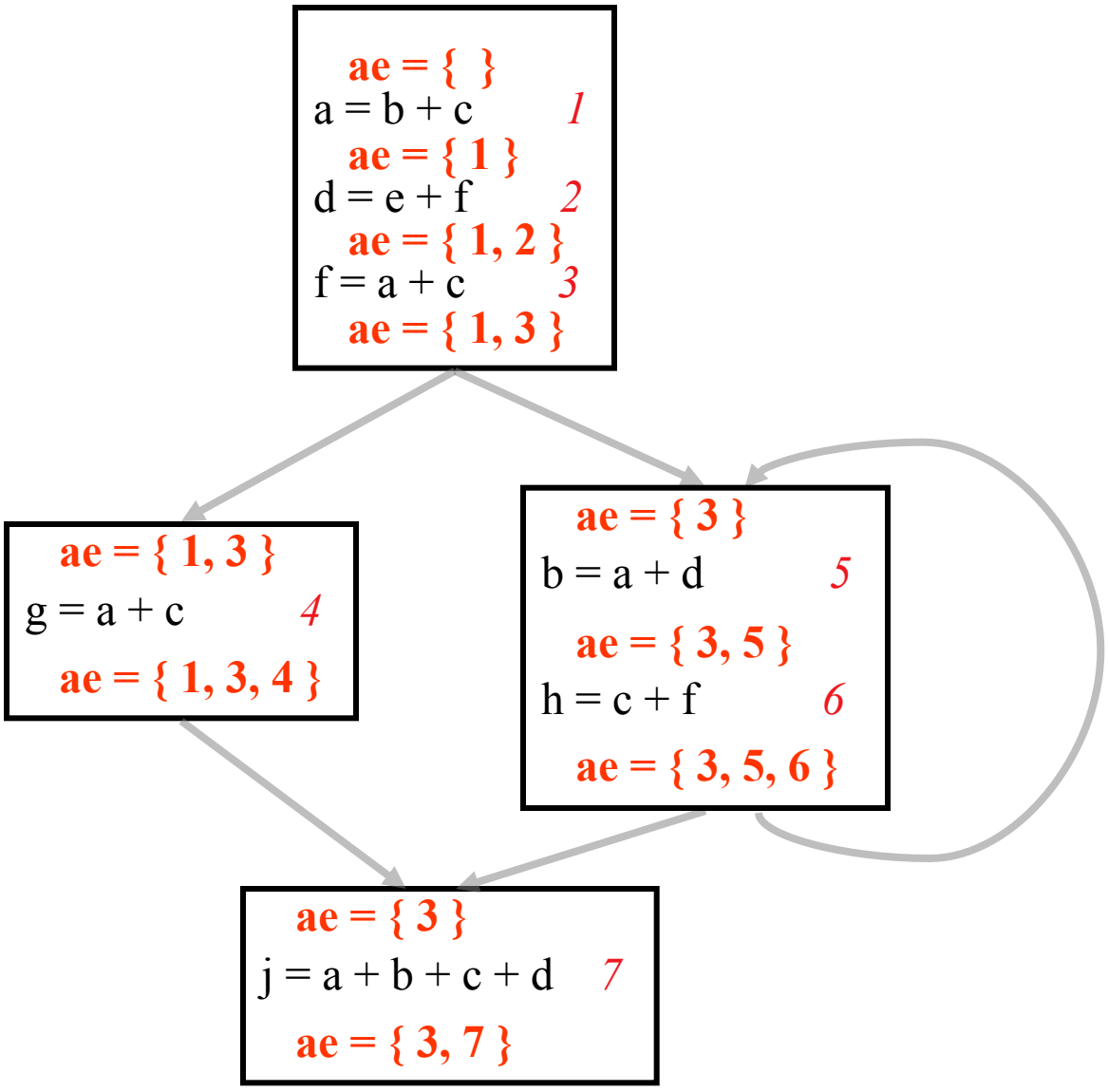
Propagate within the Basic Block

- Start with the IN set of available expressions
- Linearly propagate down the basic block
  - same as data-flow step
  - single pass since no back edges



$$OUT = gen \cup (IN - kill)$$

Available Expressions



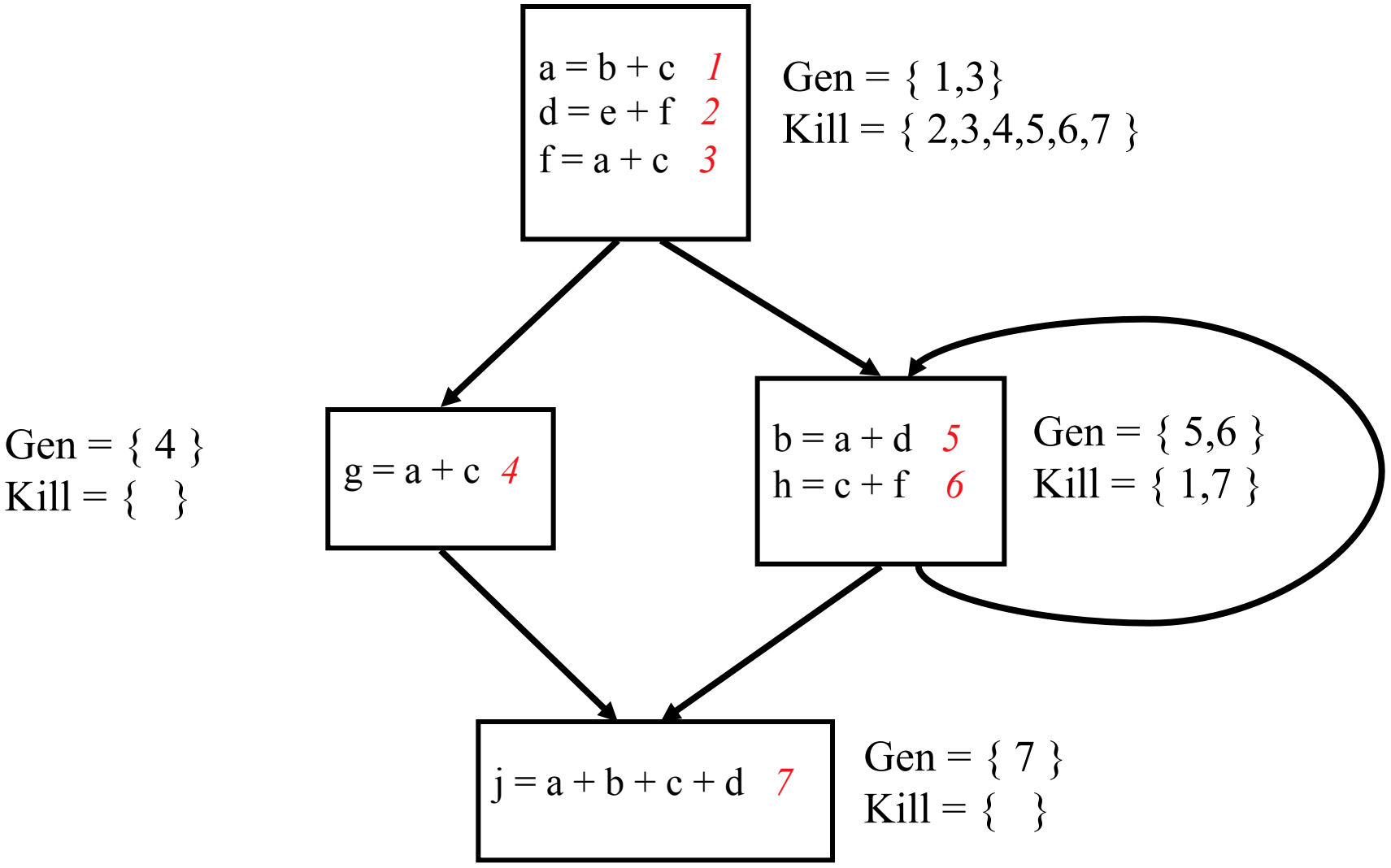
Outline

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

Practical Issues: Bit Sets

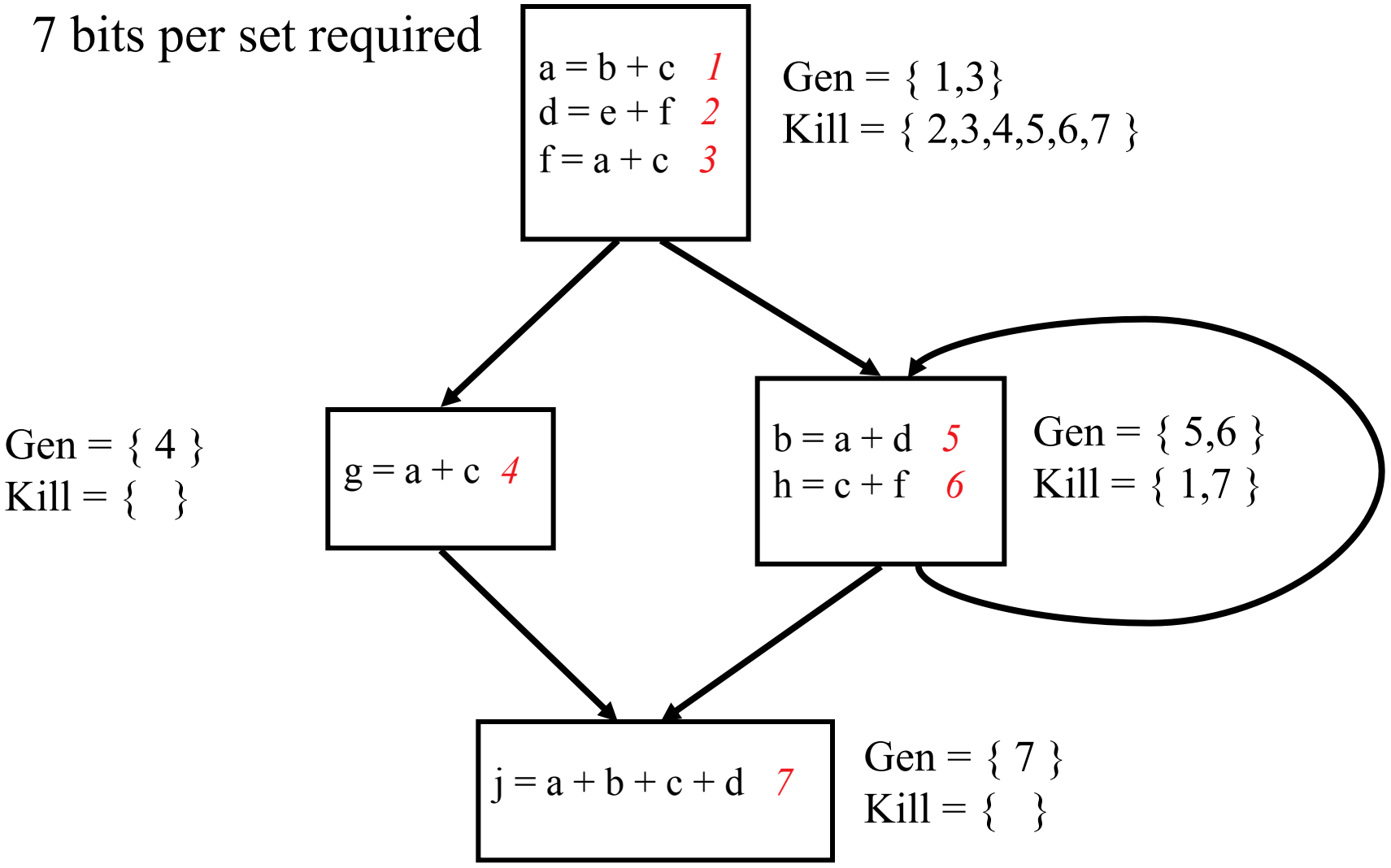
- Assign a bit to each element of the set
  - Union ⇒ bit OR
  - Intersection ⇒ bit AND
  - Subtraction ⇒ bit NEGATE and AND
- Fast implementation
  - 32 elements packed to each word
  - AND and OR are single instructions

Aggregate Gen and Kill Sets



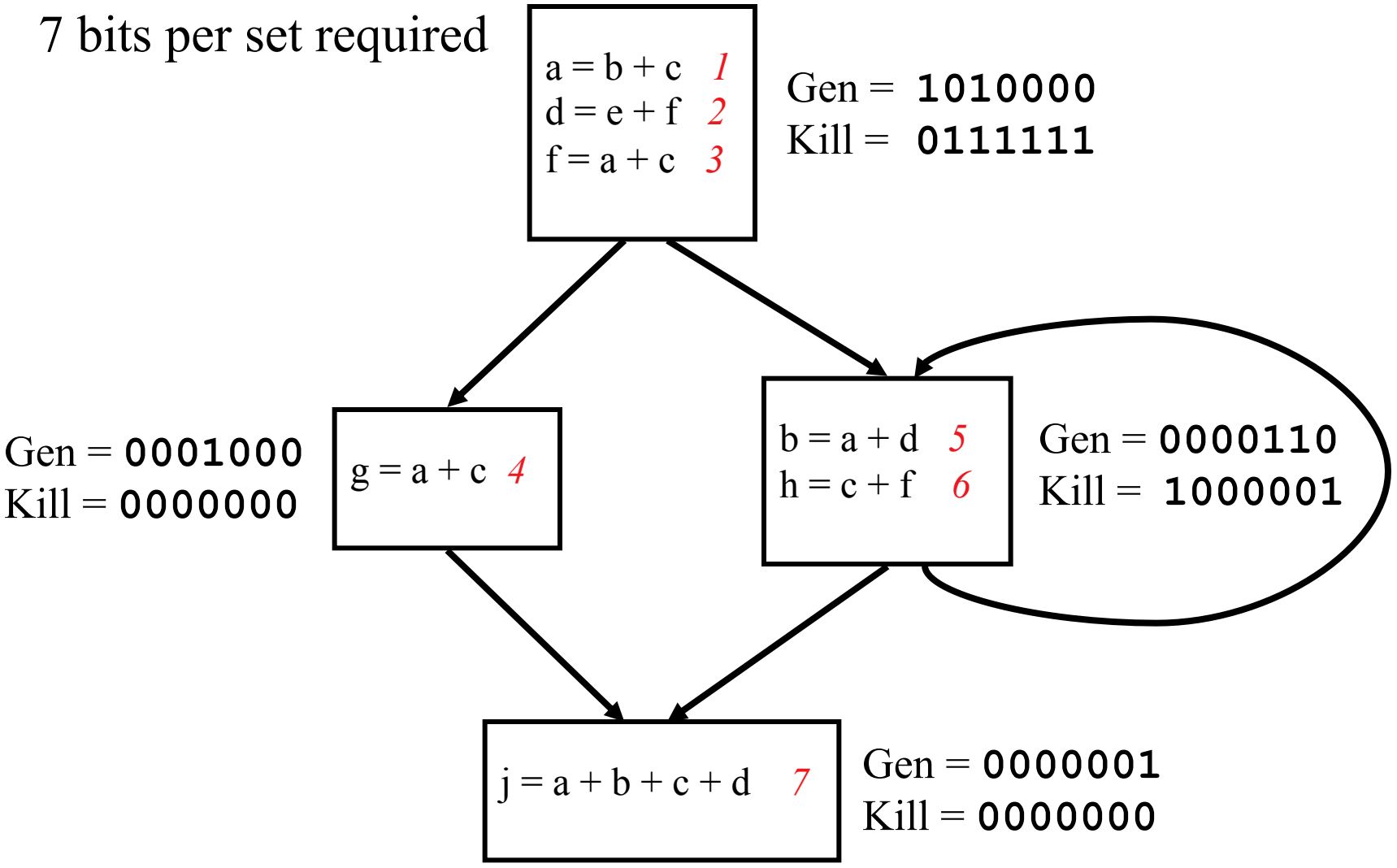
Aggregate Gen and Kill Sets

7 bits per set required



Aggregate Gen and Kill Sets

7 bits per set required



Outline

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

Formulating an Iterative Data-Flow Analysis Problem

- Problem Independent
  - Calculate Gen and Kill Sets for each Basic Block
  - Iterative Propagation of Information until Convergence
  - Propagation of Information within the Basic Block

Formulating an Iterative Data-Flow Analysis Problem

- We need to Build the Control-Flow Graph
  - Defines predecessors and successors
- Run the Round-Robin Work-list Algorithm
  - Initializes abstract valued for each node n
  - Iterates until it reaches a fixed point
- To Solve another Data-Flow Problem
  - Replace the initialization step and the fixed-point equations
  - Fixed-point equations includes direction of propagation
  - Predecessors or successors, as needed

Formulating an Iterative Data-Flow Analysis Problem

$DOM(n_0) \leftarrow \emptyset$

for i ← 1 to |N|
    $DOM(n_i) \leftarrow \{ N \}$

change ← true

while (change)
    change ← false
    for i ← 0 to |N|
        $TEMP \leftarrow \{ n_i \} \cup (\cap_{p \in pred(ni)} DOM(p))$
if $DOM(n_i) \neq TEMP$ then
        change ← true
        $DOM(n_i) \leftarrow TEMP$

- Questions:
  - Termination: Does it Halt?
  - Correctness: What answer does it Produce?
  - Speed: How quickly does it find that Answer?

Data-Flow Analysis : The Basics

- Data-Flow Sets drawn from a Semi-Lattice, L, of facts
- Sets are modified by Transfer Functions, f_i, that model effect of code on contents of the Sets
- Function Space of all possible Transfer Functions is F
  - Properties of Semi-Lattice L and Transfer Functions F govern termination, correctness, & speed

To reason about the properties of a data-flow problem, we cast it into a lattice-theory framework and prove some simple theorems about the problem

Data-Flow Analysis : The Basics

- Lattice
  - Abstract quantities over which the analysis will operate
  - Example: Sets of Available Expressions
- Transfer Functions
  - How each control-flow and computational construct affects the abstract quantities
    Example: the OUT equation for each statement
- Merging of Control-Flow Paths
  - Combining Operator of Data-Flow "meet" Operation
  - Typically Union or Intersection
  - *not the same as the lattice "meet" or "join"...*

Data-Flow Analysis: Semilattice

A semilattice is a set L and a meet operation $\wedge$ such that,

$\forall\ a, b, \& c \in L$:

1. $a \wedge a = a$
2. $a \wedge b = b \wedge a$
3. $a \wedge (b \wedge c) = (a \wedge b) \wedge c$

The meet operator combines the sets when two paths converge, or meet

$\wedge$ imposes an order on L, $\forall\ a, b, \& c \in L$:

1. $a \geq b \Leftrightarrow a \wedge b = b$
2. $a > b \Leftrightarrow a \geq b$ and $a \neq b$

A semilattice has a bottom element, denoted $\bot$

1. $\forall\ a \in L, \bot \wedge a = \bot$
2. $\forall\ a \in L, a \geq \bot$

Sometimes we work with a lattice, which has a top element, denoted $\top$
$\forall a \in L, \top \wedge a = a$

Reaching Definitions Problem

It may be important to know which are the set of possible variable definitions that reach each specific variable use.

Why?

- If the set is a singleton and a constant then we can propagate the value...
- Basis for the Webs in Register Allocation !
- Remember SSA...

Reaching Definitions Problem

Def: *A definition **d** of a variable **v** reaches instruction **i** iff instruction **i** reads the value of **v** and there exists a path from **d** to **i** that does not (re)define **v**.*

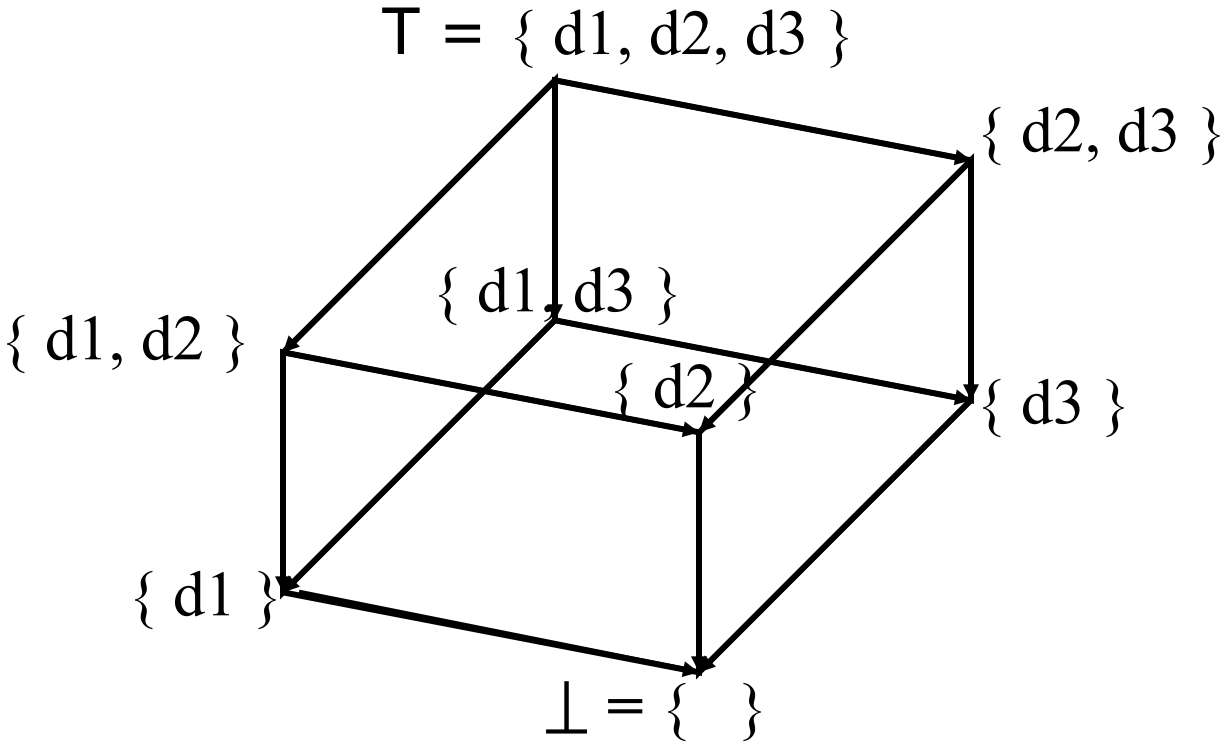
- Data-Flow (Forward) Problem:
  - Annotates program point in CFG with Reaches(n)
  - Initialization: Reaches(n) = $\emptyset$, $\forall n$
- Equation: $\text{Reaches(n)} = \bigcup_{m \in \text{preds(n)}} (\text{DEDef(m)} \cup (\text{Reaches(m)} \cap \overline{\text{DefKill(m)}}))$
  - where DEDef(*m*) is the set of downward-exposed definition in *m*: those definitions in *m* that are not subsequently redefined;
  - DefKill(*m*) contains **all** the definition points that are obscured by a definition of the same variable ***v*** in *m*

Lattice

- A Lattice L consists of
  - A Set of Values
  - Two operations meet ($\bigwedge$) and join ($\bigvee$)
  - A top value ($\top$) and a bottom value ($\bot$)

Lattice

- Example: the Lattice for the Reaching Definitions
  Problem where there are only 3 definitions: {d1, d2, d3}



Meet and Join Operators

- Meet and Join forms a Closure
  - For all a, b ∈ L there exist a unique c and d ∈ L such that:

    $a \wedge b = c$ $a \vee b = d$

- Meet and Join are Commutative:

  $a \wedge b = b \wedge a$ $a \vee b = b \vee a$

- Meet and Join are Associative:

  $(a \wedge b) \wedge c = b \wedge (a \wedge c)$ $(a \vee b) \vee c = b \vee (a \vee c)$

- There exist a unique Top element ($\top$) and Bottom element ($\bot$) in L such that:

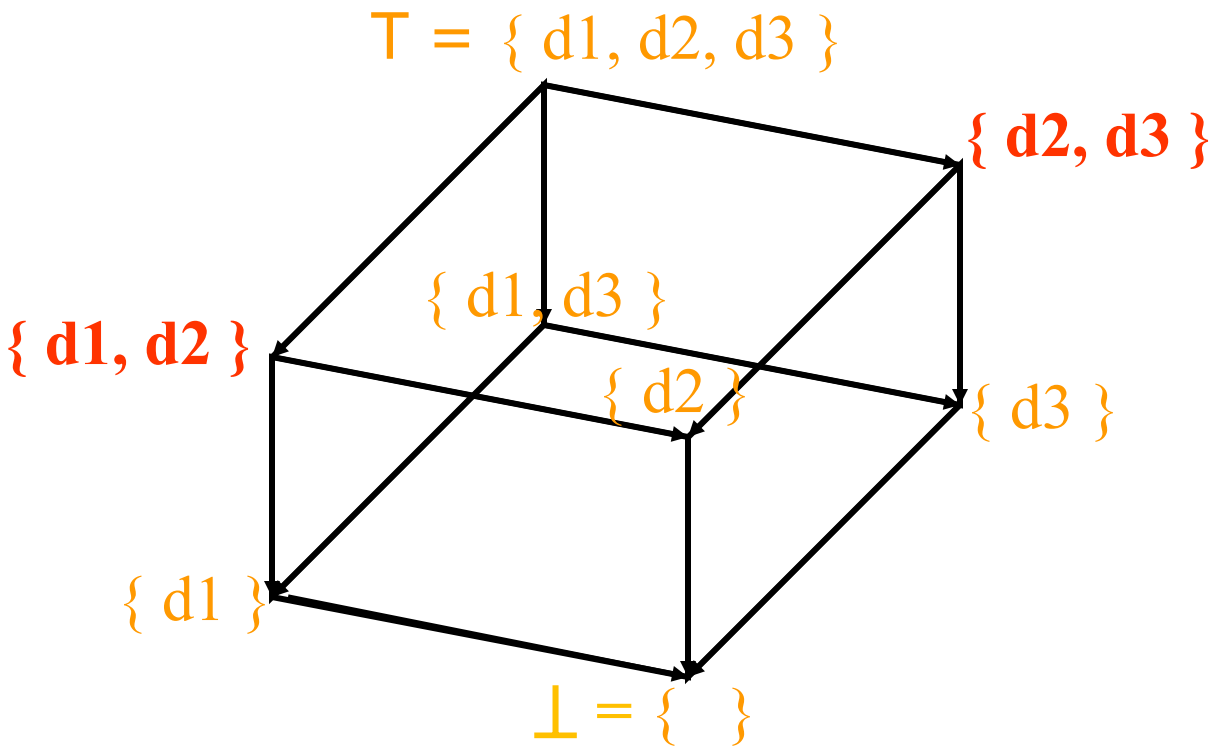
  $a \wedge \bot = \bot$ $a \vee \top = \top$

Meet and Join Operators

- Meet Operation: Greatest Lower Bound - GLB({x,y})
  - Typically Set Intersection
  - Follow the lines downwards from the two elements in the lattice until they meet at a single unique element
- Join Operation: Lowest Upper Bound - LUB({x,y})
  - Typically Set Union
  - There is a unique element in the lattice from where there is a downwards path (with no shared segment) to both elements

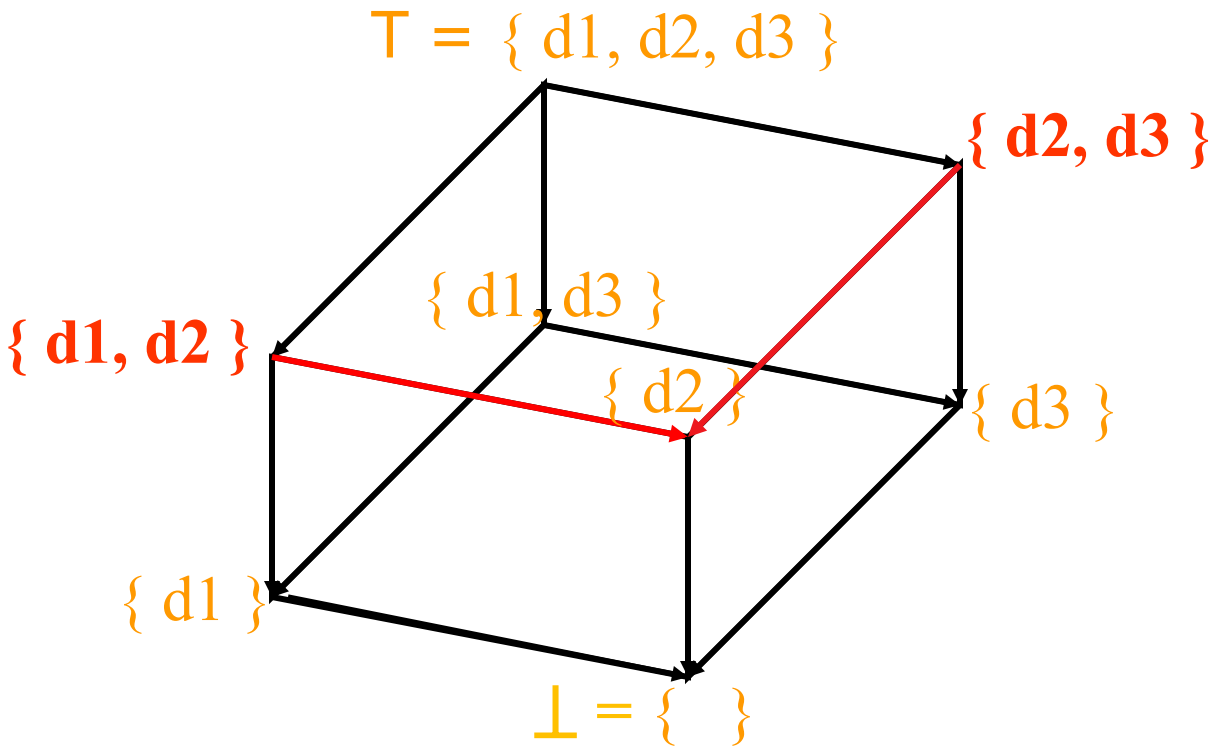
Meet and Join Operators

$\{ d1, d2 \} \wedge \{ d2, d3 \} = ???$



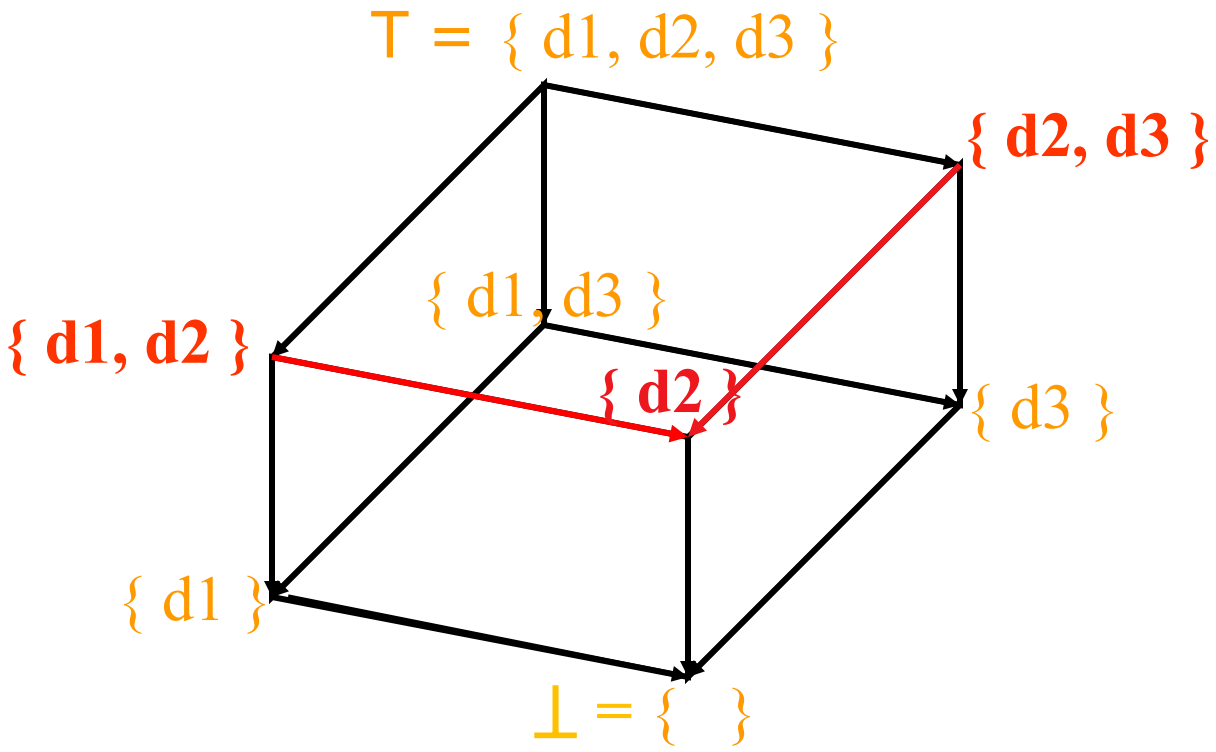
Meet and Join Operators

$\{ d1, d2 \} \wedge \{ d2, d3 \} = ???$



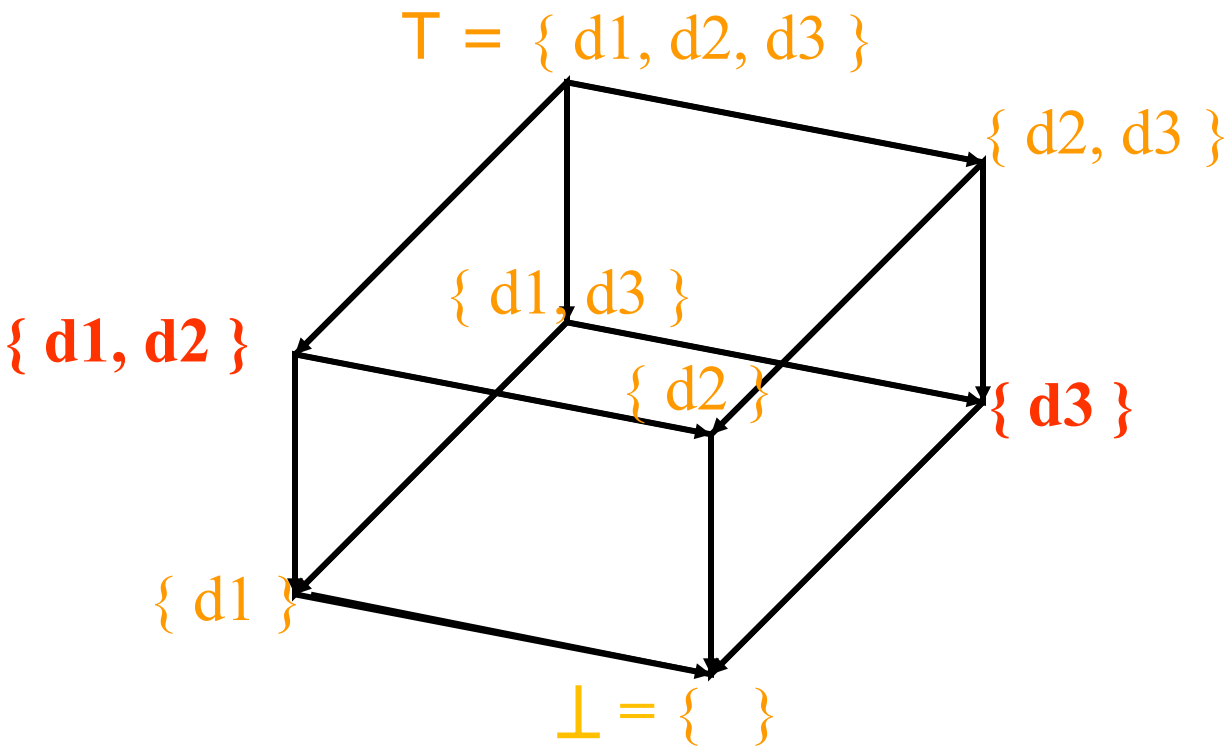
Meet and Join Operators

$$\{ d1, d2 \} \wedge \{ d2, d3 \} = \{ d2 \}$$



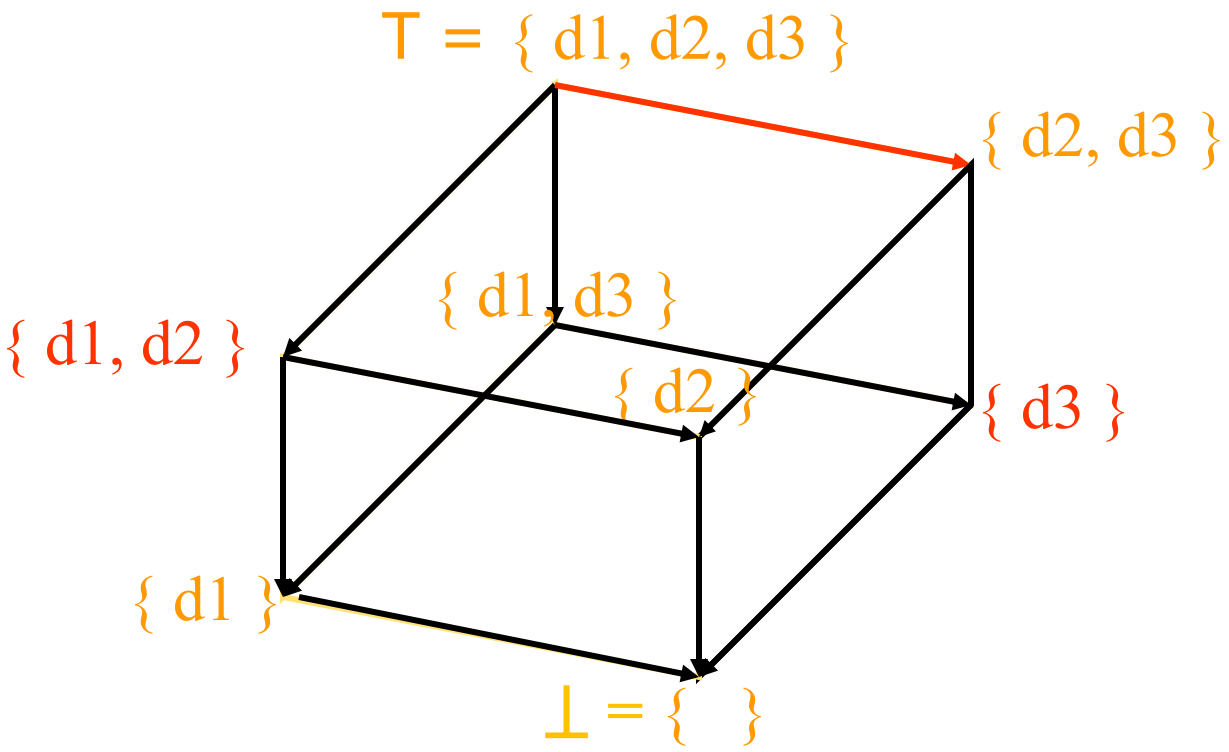
Meet and Join Operators

$\{ d1, d2 \} \vee \{ d3 \} = ???$



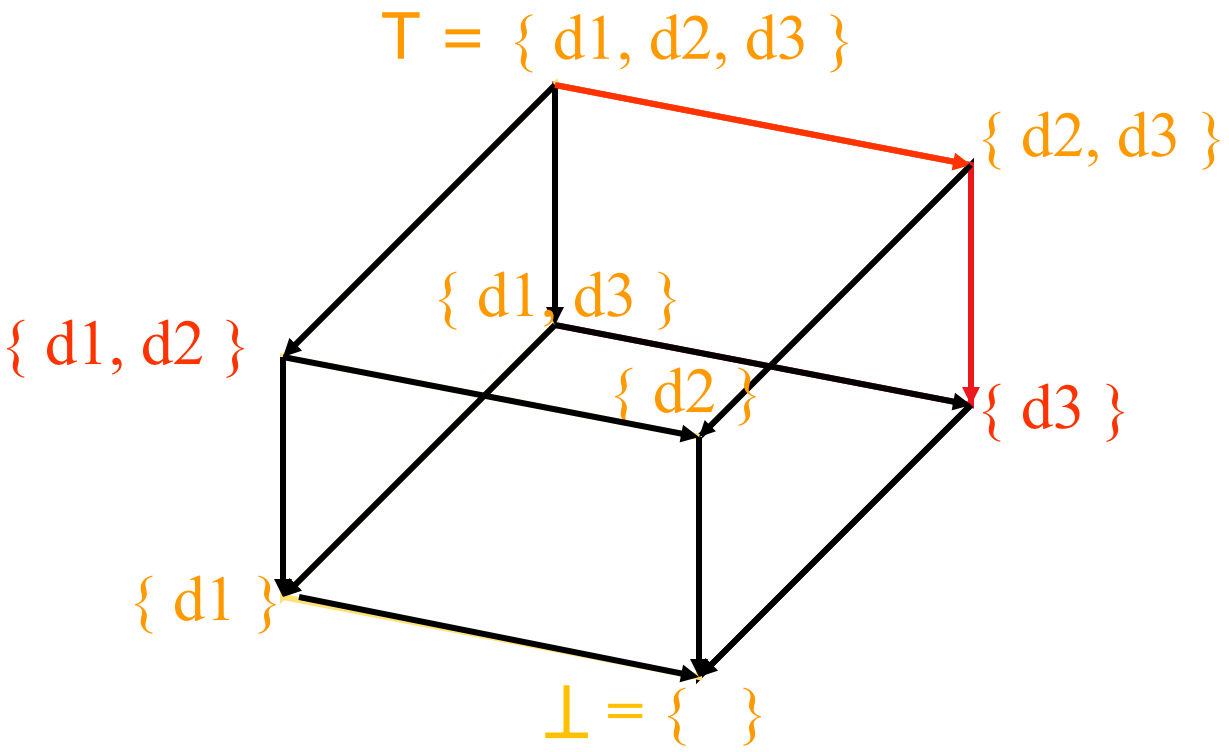
Meet and Join Operators

{ d1, d2 } ∨ { d3 } = ???



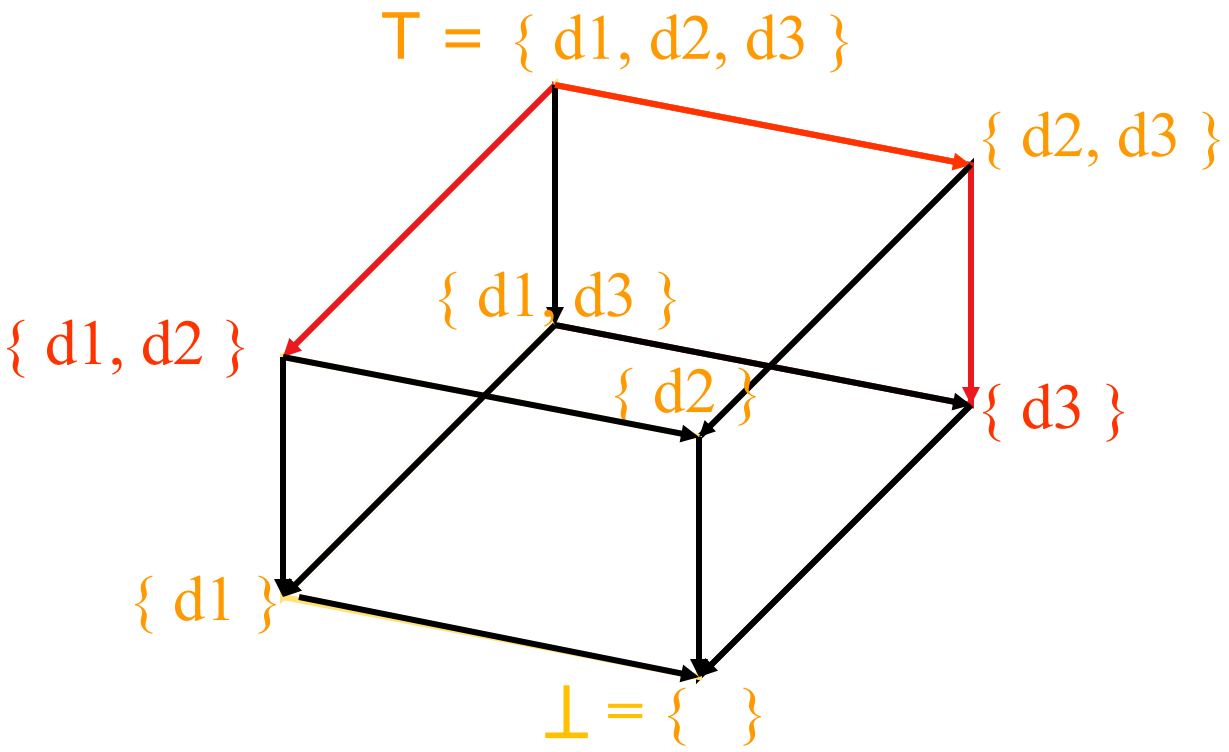
Meet and Join Operators

$\{ d1, d2 \} \vee \{ d3 \} = ???$



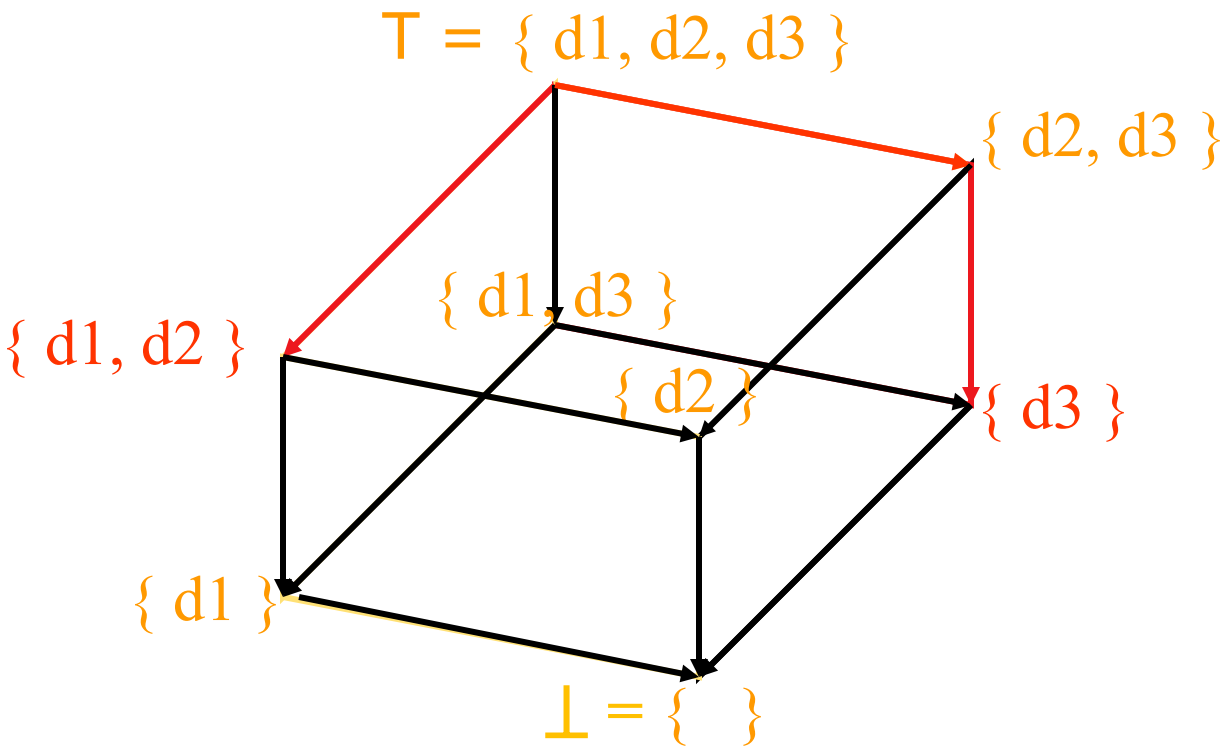
Meet and Join Operators

{ d1, d2 } ∨ { d3 } = ???



Meet and Join Operators

$$\{ d1, d2 \} \vee \{ d3 \} = \{ d1, d2, d3 \}$$



Intuition about Termination

- Data-Flow Analysis starts assuming most optimistic (or conservative) values (⊤)
- Each Stage applies a Flow Function
  - Vnew ⊆ Vprev
  - Moves Downwards/Upwards in the Lattice
- Until stable (values don't change)
  - A fixed point is reached at every basic block

- Lattice has a finite height ⇒ should terminate

Termination

If every $f_n \in F$ is monotone, i.e., $x \sqsubseteq y \Rightarrow F(x) \sqsubseteq F(y)$ and

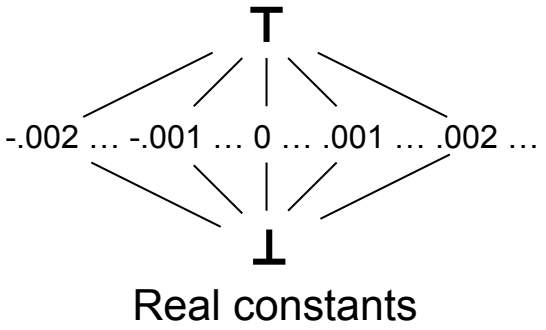
If the lattice is bounded, i.e, every descending chain is finite:

- Chain is a sequence $x_1, x_2, ..., x_n$ where $x_i \in L$
- $x_i > x_{i+1}$, $1 \leq i \leq n \Rightarrow$ chain is descending

Then

- The set of values can only change finite number of times
- The iterative algorithm must halt on an instance of the problem

- Observations:
  - Any finite semilattice is bounded
  - Some infinite semilatices are bounded (see righ



Correctness

- If every $f_n \in F$ is monotone, *i.e.*, $x \leq y \Rightarrow f(x) \leq f(y)$, and
- If the semilattice is bounded, *i.e.*, every descending chain is finite
  - Chain is sequence $x_1, x_2, \ldots, x_n$ where $x_i \in L$, $1 \leq i \leq n$
  - $x_i > x_{i+1}$, $1 \leq i < n \Rightarrow$ chain is descending

$f^k(x)$ is the application of f to x *k* times

Given a bounded semilattice *S* and a monotone function space *F*

- $\exists\ k$ such that $f^k(\bot) = f^j(\bot)\ \forall\ j > k$
- $f^k(\bot)$ is called the least fixed-point of *f* over *S*
- If *L* has a $\top$, then $\exists\ k$ such that $f^k(\top) = f^j(\top)\ \forall\ j > k$ and $f^k(\top)$ is called the maximal fixed-point of *f* over *S*

optimism

Correctness

- If every $f_n \in F$ is monotone, *i.e.*, $f(x \wedge y) \leq f(x) \wedge f(y)$, and
- If the lattice is bounded, *i.e.*, every descending chain is finite
  - Chain is sequence $x_1, x_2, \ldots, x_n$ where $x_i \in L$, $1 \leq i \leq n$
  - $x_i > x_{i+1}$, $1 \leq i < n \Rightarrow$ chain is descending

Then

- The round-robin algorithm computes a least fixed-point (LFP)
- The uniqueness of the solution depends on other properties of *F*

- Unique solution ⇒ it finds the one we want
- Multiple solutions ⇒ we need to know which one it finds

Correctness

- Does the iterative algorithm compute the desired answer?

Admissible Function Spaces

1. $\forall f \in F, \forall x,y \in L, f(x \wedge y) = f(x) \wedge f(y)$
2. $\exists f_i \in F$ such that $\forall x \in L, f_i(x) = x$
3. $f,g \in F \exists h \in F$ such that $h(x) = f(g(x))$
4. $\forall x \in L, \exists$ a finite subset $H \subseteq F$ such that $x = \wedge_{f \in H} f(\perp)$

If meet does not distribute over function application, then the fixed point solution may not be unique. The iterative algorithm will find a LFP.

If F meets these four conditions, then an instance of the problem will have a unique fixed point solution *(instance ⇒ graph + initial values)*

⇒ LFP = MFP = MOP

⇒ order of evaluation does not matter

Limitations of Data-Flow Analysis

- Precision – "up to symbolic execution"
  - Assume all paths are taken
- Solution – cannot afford to compute MOP solution
  - Large class of problems where MOP = MFP = LFP
  - Not all problems of interest are in this class
- Arrays – treated naively in classical analysis
  - Represent whole array with a single fact
- Pointers – difficult (and expensive) to analyze
  - Imprecision rapidly adds up
  - Need to ask the right questions
- Summary
  - For scalar values, we can quickly solve simple problems

Maximal Fixed Point Solution (MFP)

- Claim:

  Among all the solutions to the system of dataflow equations, the iterative solution is the most precise

- Intuition:
  - We start with the top element at each program point (i.e. most imprecise or *conservative* information)
  - Then refine the information at each iteration to satisfy the dataflow equations
  - Final result will be the closest to the top

- Iterative solution for dataflow equations is called Maximal Fixed Point solution (MFP)
- For any Fixed-Point (FP) solution of the equations: FP $\sqsubseteq$ MFP

Meet-Over-All-Paths Solution (MOP)

- Is MFP the best solution to the Analysis Problem?

- Another approach: consider a lattice framework, but use a different way to compute the solution
  - Let G be the control flow graph with start node n_0
  - For each path $p_k=[n_0, n_1, \ldots, n_k]$ from entry to node n_k:

    $in[p_k] = F_{nk-1} (\ldots (F_{n1}(F_{n0}(d_0))))$
  - Compute solution as

    $in[n] = \bigvee \{ in[p_k] \mid$ all paths p_k from n_0 to $n_k \}$

- This solution is the Meet Over All Paths solution (MOP)

MFP versus MOP

- It can be proven that Meet Over All Paths (MOP) solution is always more precise than Maximal Fixed Point (MFP):

$$MFP \sqsubseteq MOP$$

- Why not use MOP?
- MOP is intractable in practice…

  1. Exponential number of paths: for a program consisting of a sequence of N if statements, there will 2^N paths in the CFG

  2. Infinite number of paths: for loops in the CFG

Importance of Distributivity

- Critical Fact: If transfer functions are distributive, then the solution to the dataflow equations is identical to the Meet-Over-All-Paths (MOP) solution:

$$MFP = MOP$$

- For distributive transfer functions, can compute the intractable MOP solution using the iterative fixed-point algorithm

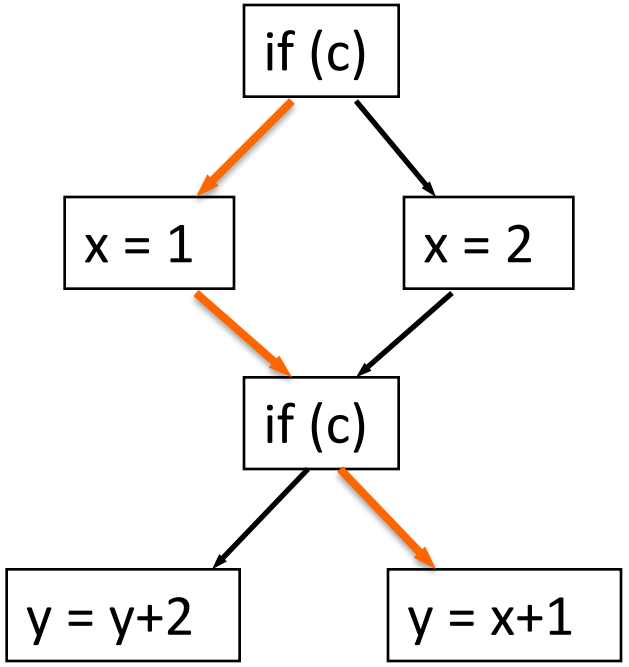
Can We Do Better Than MOP?

- Is MOP the best solution to the analysis problem?

- MOP computes solution for all paths in the CFG
- There may be paths which will never occur in any execution
- So MOP is (still) conservative

- IDEAL = solution which takes into account only paths which occur in some execution

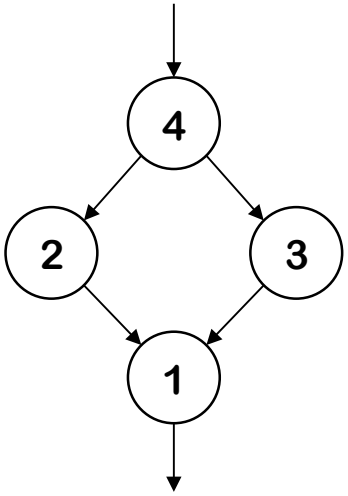
- This is the best solution
  - but it is undecidable



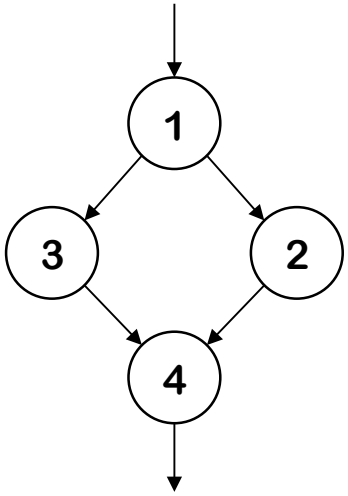
Speed

- If a Data-Flow Framework meets those admissibility conditions then is has a unique fixed-point solution
  - The iterative algorithm finds the (best) answer
  - The solution does not depend on the order of computation
  - Algorithm can choose an order that converges quickly
- Intuition:
  - Choose an order so that changes propagate as far as possible on each "sweep" or "pass" over the CFG
    - Process a node's predecessors before the node
  - Cycles pose problems, naturally
    - Ignore back edges when computing evaluation order

Speed



Postorder



Reverse Postorder

- Reverse postorder visits predecessors before visiting a node
- Use reverse preorder for backward problems
  - Reverse postorder on reverse CFG is reverse preorder

Speed

What does this mean?

- Reverse postorder
  - Easily computed order that increases propagation per pass
- Round-robin iterative algorithm
  - Visit all the nodes in a consistent order (RPO)
  - Do it again until the sets stop changing
- Rapid condition
  - Most classic global data-flow problems meet this condition

These conditions are easily met

- Admissible framework, rapid function space
- Round-robin, reverse-postorder, iterative algorithm

⇒ The analysis runs in (*effectively*) linear time

Data-Flow Analysis Framework

- A Data-Flow Analysis Framework consists of:
  - A lattice $(L, \subseteq, \cap, \bot)$ where:
    - L is the dataflow information
    - $\subseteq$ is the ordering relation
    - $\cap$ is the meet operation (GLB)
    - $\bot$ is the bottom element
  - Transfer Functions $F_n : L \rightarrow L$ for each CFG node n
  - Boundary Data-Flow information d_0
    - Before CFG entry node for a Forward Analysis
    - After CFG exit node for a Backward Analysis

Data-Flow Equations

- Forward Data-Flow Analysis:

  $in[n_0] = d_0$, where n_0 = CFG entry node
  $out[n] = F_n(in[n])$, for all n
  $in[n] = \cap \{out[n'] \mid n' \in pred(n)\}$, for all n

- Backwards Data-Flow Analysis:

  $out[n_0] = d_0$, where n_0 = CFG exit node
  $in[n] = F_n(out[n])$, for all n
  $out[n] = \cap \{in[n'] \mid n' \in succ(n)\}$, for all n

Solving the Data-Flow Equations

- The Constraints (Forward Analysis):

  $in[n_0] = d_0$, where n_0 = CFG entry node

  $out[n] = F_n(in[n])$, for all n

  $in[n] = \cap \{out[n'] \mid n' \in pred(n)\}$, for all n

- Solution = the set of all in[n], out[n] for all n's, such that all Constraints are satisfied

- Fixed-Point Algorithm to Solve Constraints:
  - Initialize $in[n_0]=d_0$
  - Initialize everything else to $\perp$
  - Repeatedly enforce Constraints
  - Stop when Data-Flow Solution do not Change

Worklist Algorithm (Forward)

$in[n_0] = d_0$

$in[n] = \bot$, for all $n \neq n_0$

$out[n] = \bot$, for all n

worklist = $\{n_0\}$
while (worklist $\neq \emptyset$)
    Remove a node n from the worklist
    out[n] = Fn(in[n])

    for each successor n' :
        in[n'] = in[n'] $\cap$ out[n] - *distributivity at work*
        if (in[n'] has changed)
            add n' to the worklist

An Implementation

```
void analyzeForward(Method m, DataflowInfo d0) {
  result.put(m.getCFG().getEntryNode(), d0);

  Stack<CFGNode> worklist = new Stack<CFGNode>();
  while (!worklist.isEmpty()) {
    CFGNode n = worklist.pop();
    DataflowInfo in = result.get(n);
    DataflowInfo out = transferFunction(n,in);
    for (CFGNode succ : n.getSuccessors())
      if (merge(succ, out))
        worklist.add(succ);
    }
}
```

```
boolean merge(CFGNode n, DataflowInfo d) {
  DataflowInfo info = result.get(n);
  if (info == null) {
     result.put(n, d.clone());
     return true;
  }
  return info.meet(d);
}
```

Summary

- Data-Flow Analysis
  - Sets up System of Equations
  - Iteratively computes MFP
  - Terminates because Transfer Functions are monotonic and Lattice has finite height
- Other possible solutions: FP, MOP, IDEAL
- All are safe solutions, but some are more precise:

$$FP \sqsubseteq MFP \sqsubseteq MOP \sqsubseteq IDEAL$$

- MFP = MOP if distributive transfer functions
- MOP and IDEAL are intractable
- Compilers use dataflow analysis and MFP

Outline

- Overview of Control-Flow Analysis
- Available Expressions
- Algorithm for Calculating Available Expressions
- Practical Issues: Bit sets
- Formulating a Data-Flow Analysis Problem
- DU chains
- SSA form

Def-Use and Use-Def Chains

- Def-Use (DU) Chain
  - Connects a definition of each variable v_k to all the possible uses of that variable
  - A definition of v_k at point p reaches point q if there is a path from p to q where v_k is not redefined.

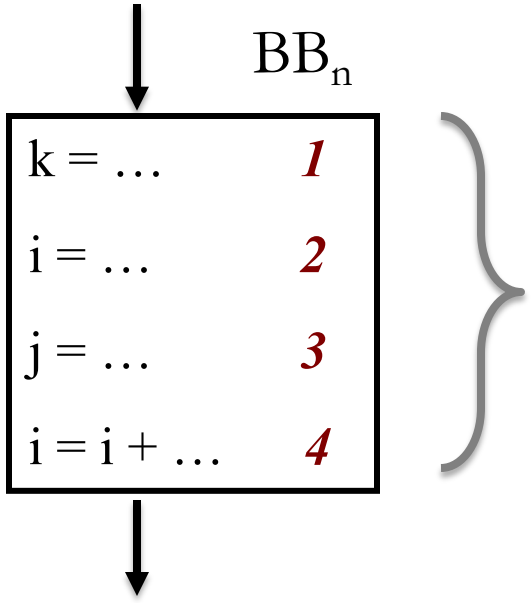
- Use-Def (UD) Chain
  - Connects a use of a variable v_k to all the possible definitions of that variable

DU-Chain Data-Flow Problem Formulation

- Lattice: The Set of Definitions
  - Bit-vector format: a bit for each variable definition in the procedure
  - Label the definitions in the input program as $d_{1,vk}$, $d_{2,vk}$, …
- Direction: Forward Flow
- Flow Functions:
  - Gen(n) = { d_i,…, $d_{i,n}$ | where d_i,v_k = 1 for definition d_i,v_k in n which is not killed inside the basic block by a subsequent definition*}
  - Kill(n) = { d_i,…, $d_{i,n}$ | where d_i,v_k = 1 iff variable v_k is defined in n}
  - OUT(n) = Gen(n) ∪ (IN(n) – Kill(n))
  - IN(n) = ⋃ OUT(p) for all predecessors nodes p of node n

** This is the notion of downwards exposed definition*

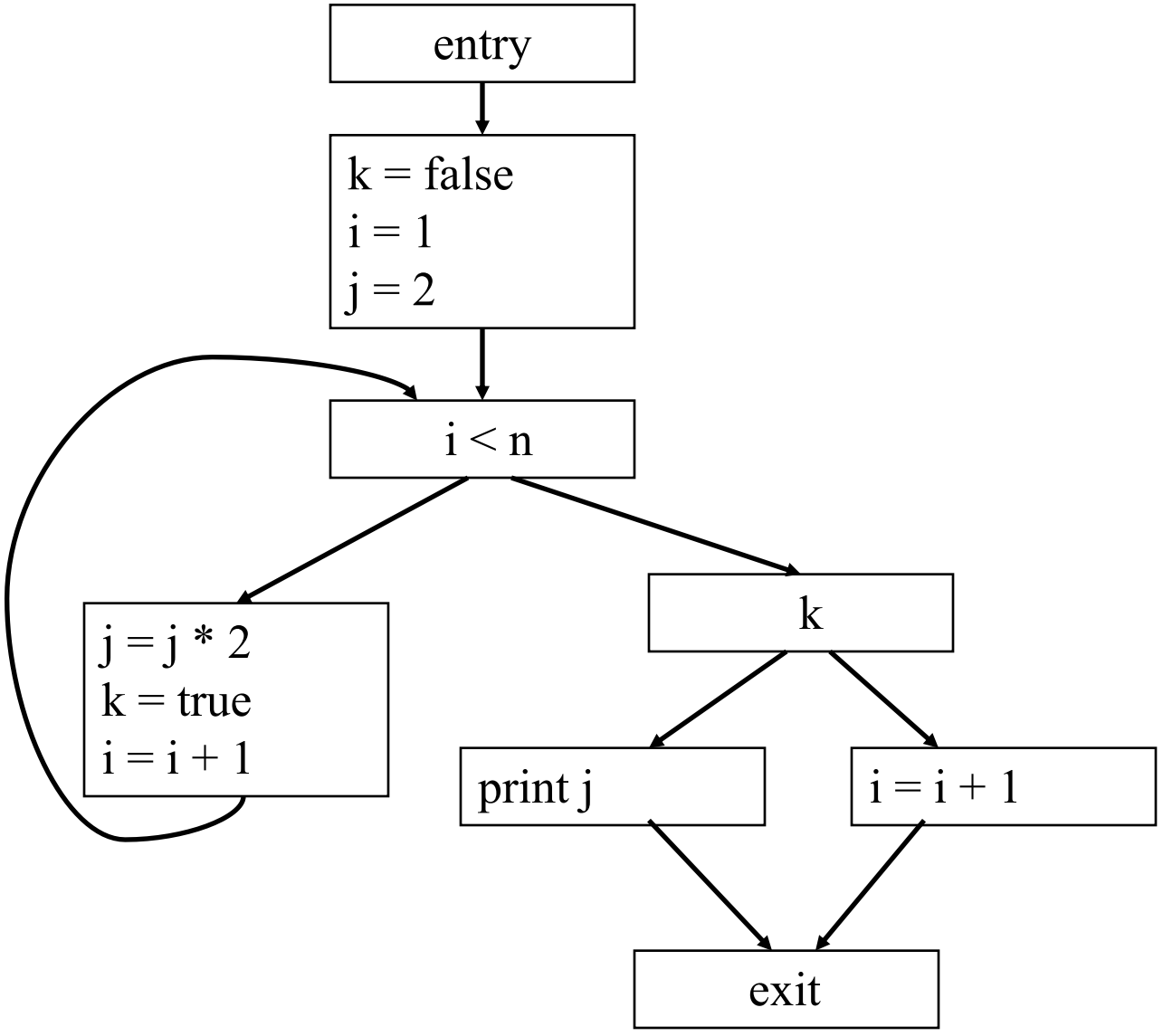
DU-Chain Gen and Kill Functions



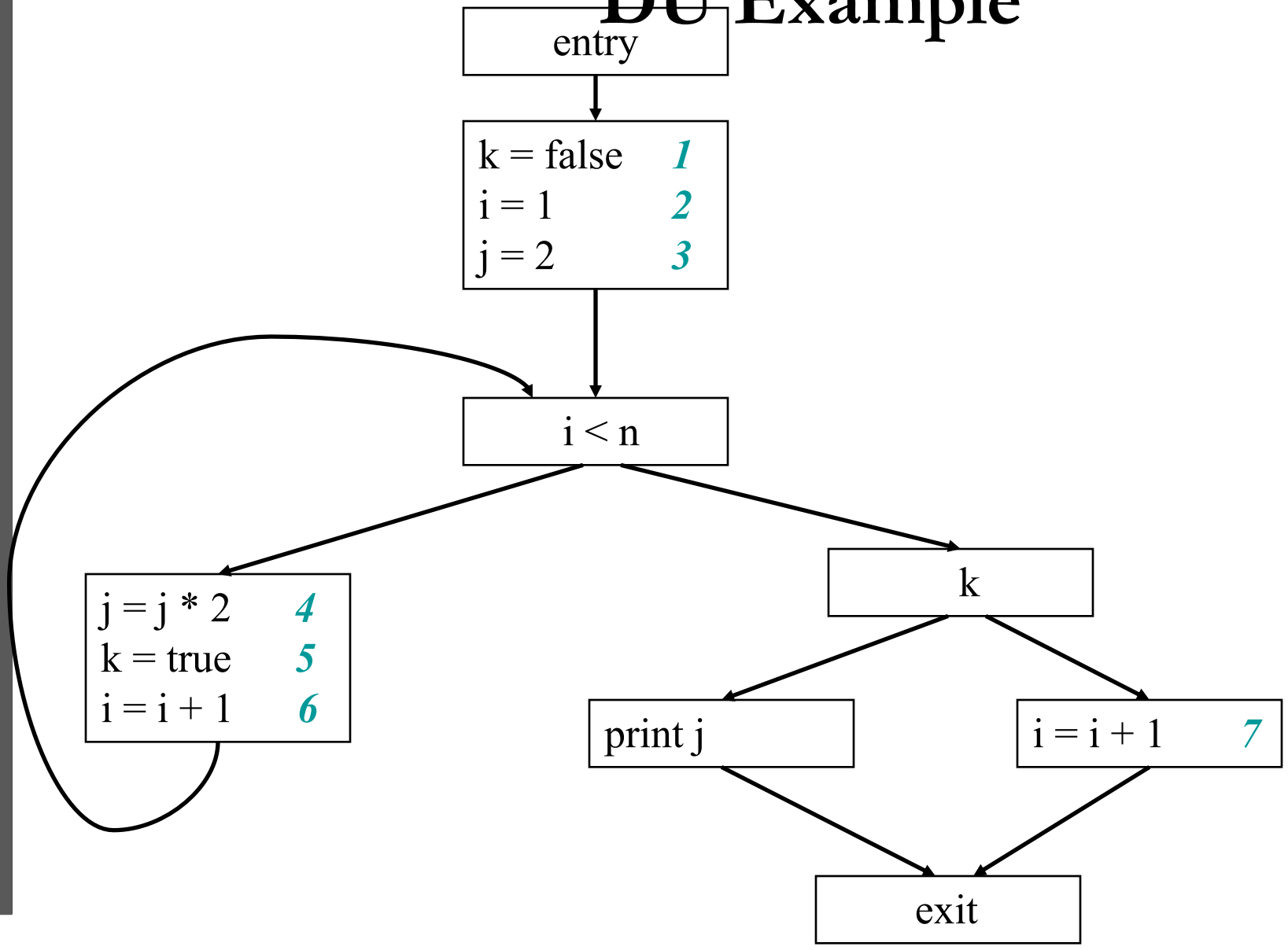
Gen(n) = { d1, d3 , d4 }
Kill(n) = { d1, d2, d3, d4, … }

- Gen = Downwards Exposed Definitions
  - Dual to Upwards Exposed Reads (see Live Variables Analysis later)
  - Can be Computed on a Forward pass of the Basic Block
    - Keep a list of variables defined in the block
    - Remove and keep only the last definition as you go along

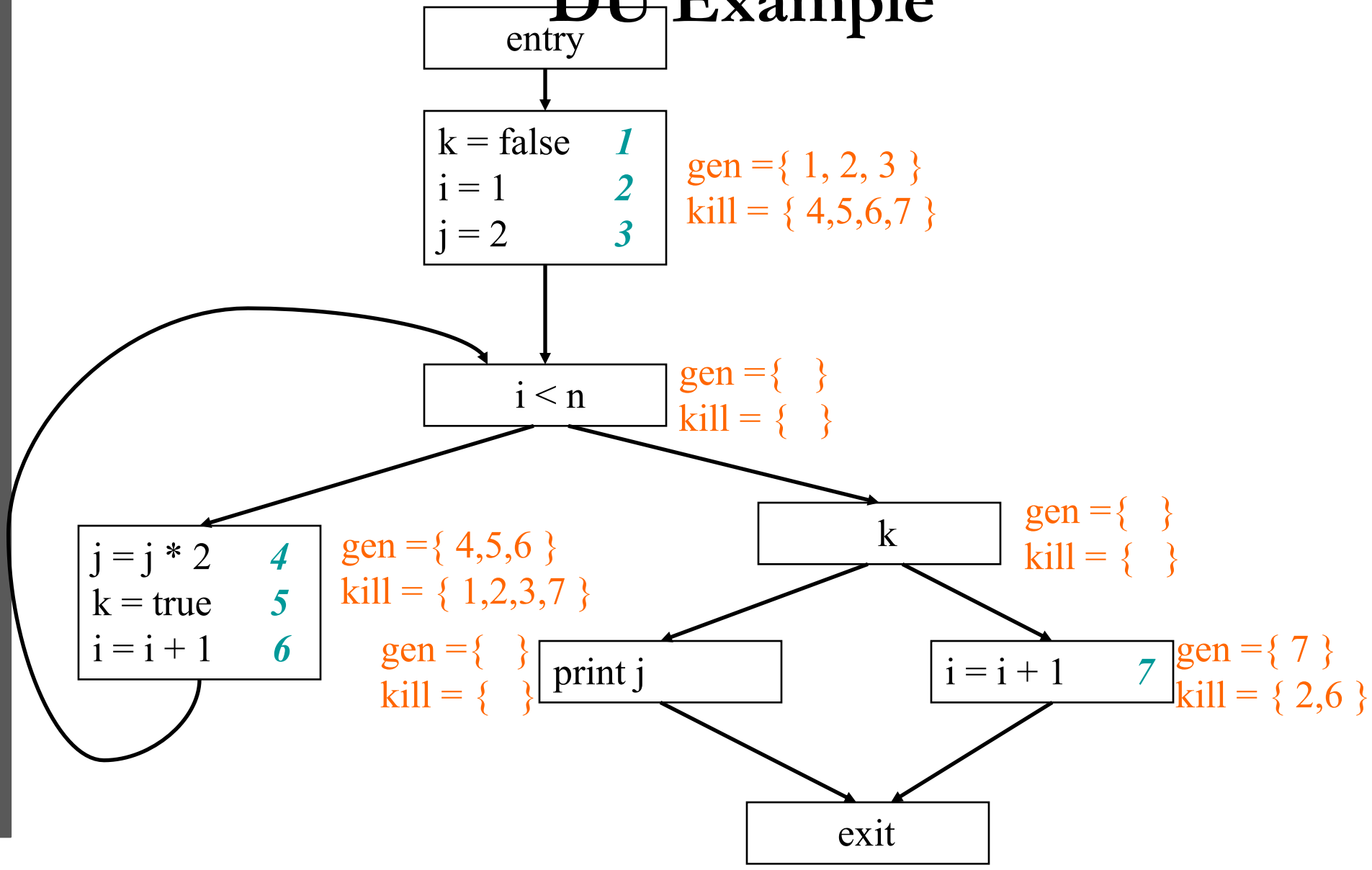
DU Example



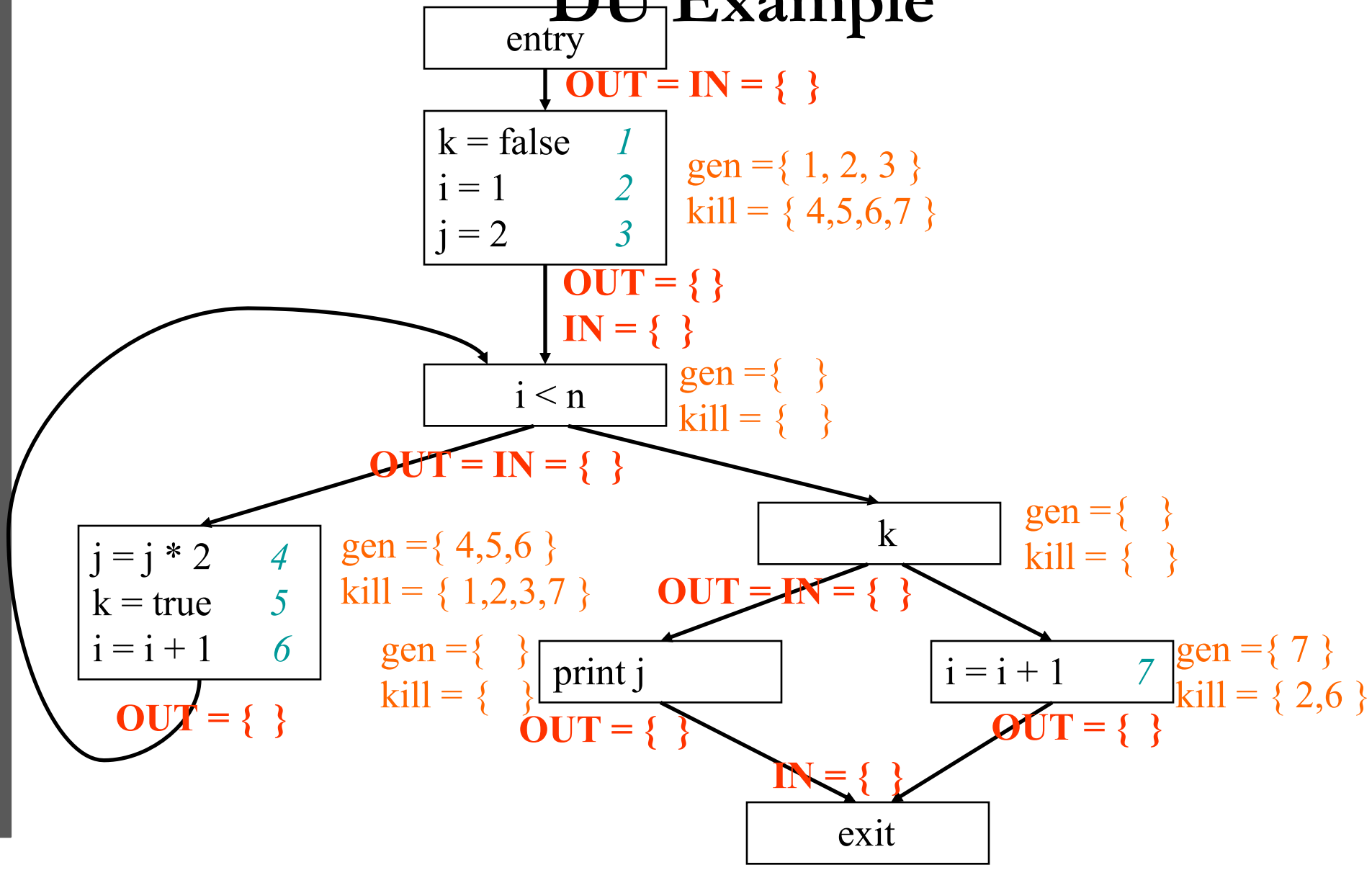
DU Example



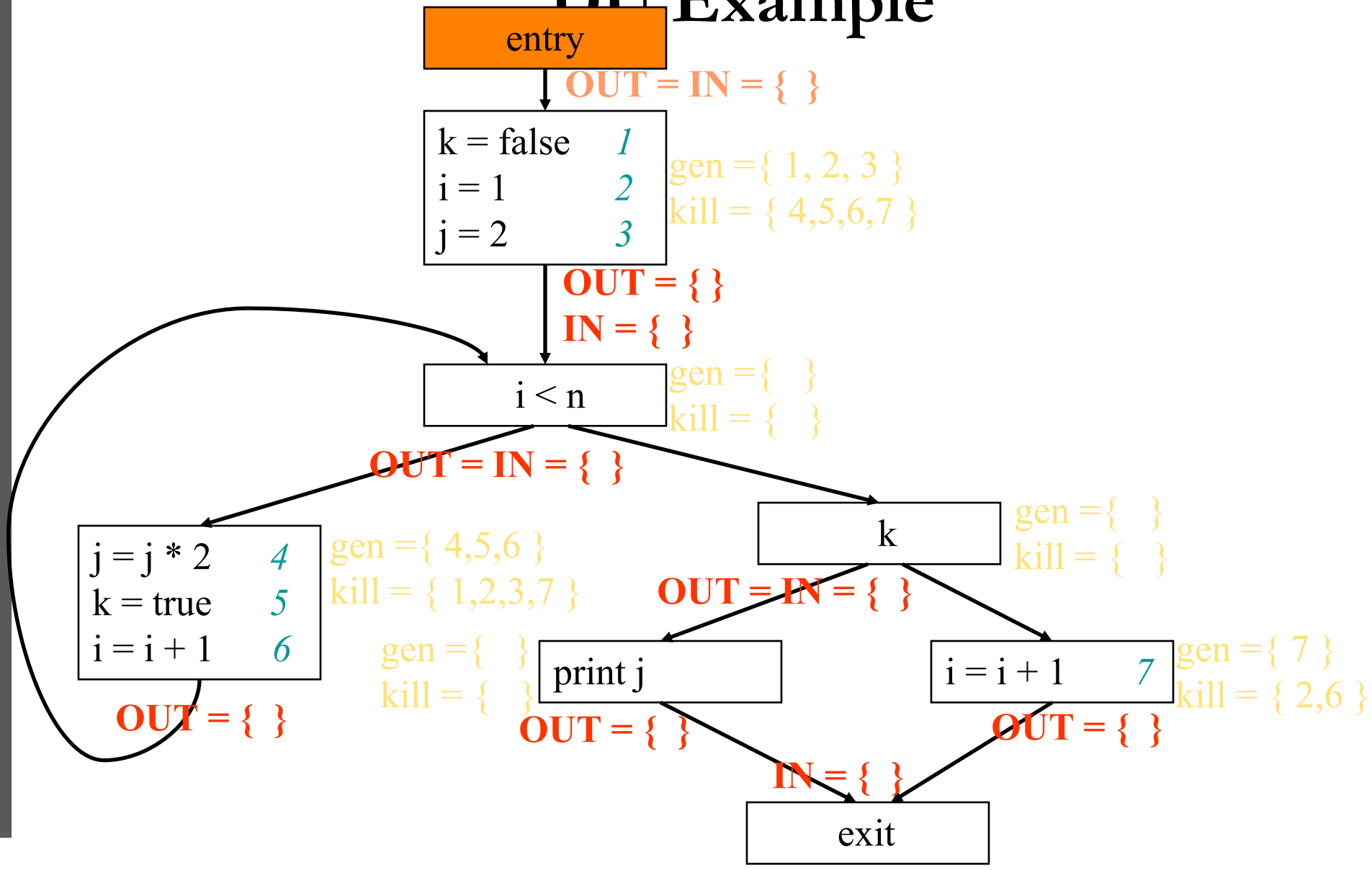
DU Example


entry
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
i < n
gen ={ }
kill = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
k
gen ={ }
kill = { }
gen ={ }
kill = { }
print j
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
exit

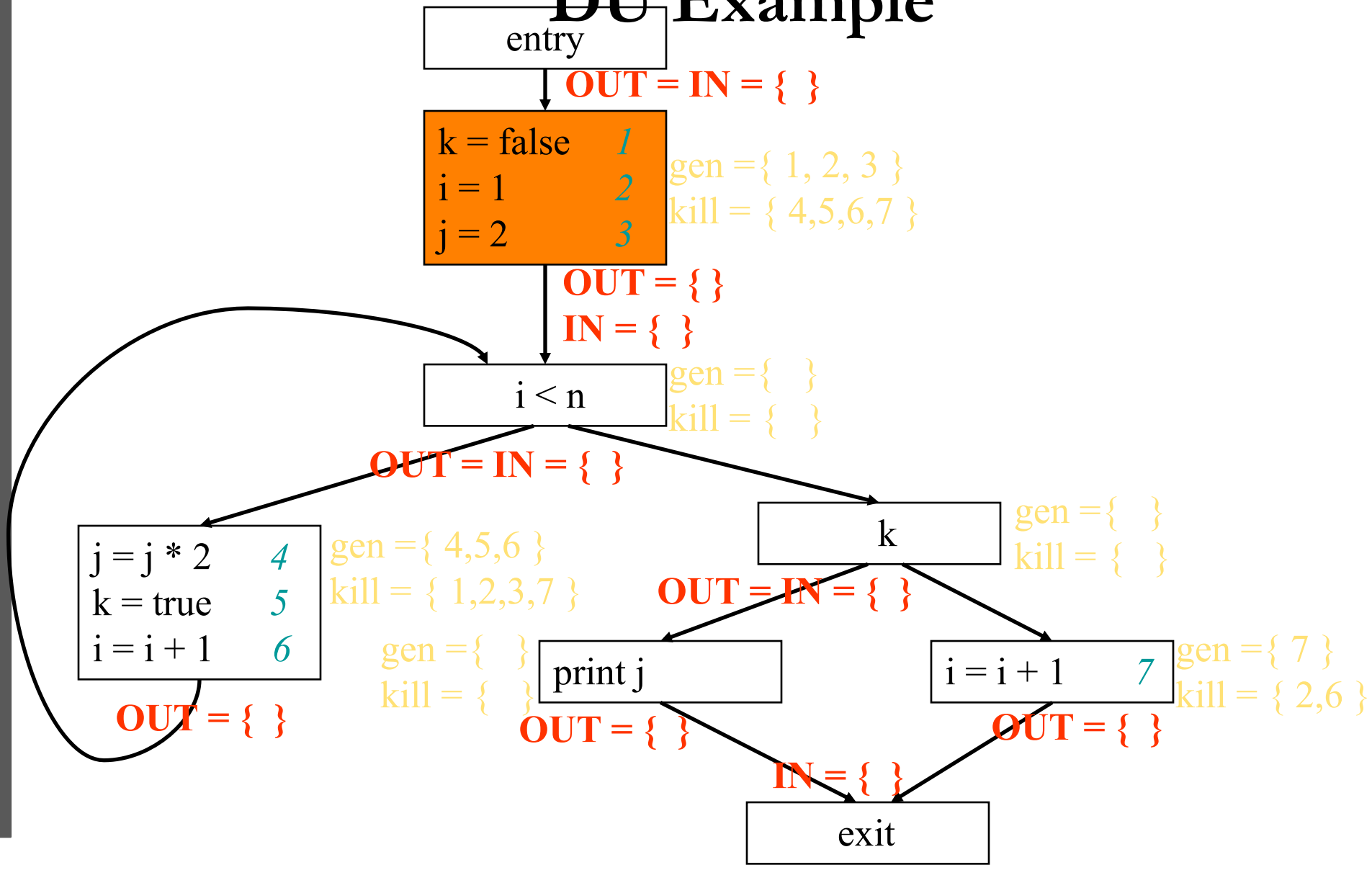
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { }
IN = { }
i < n
gen ={ }
kill = { }
OUT = IN = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

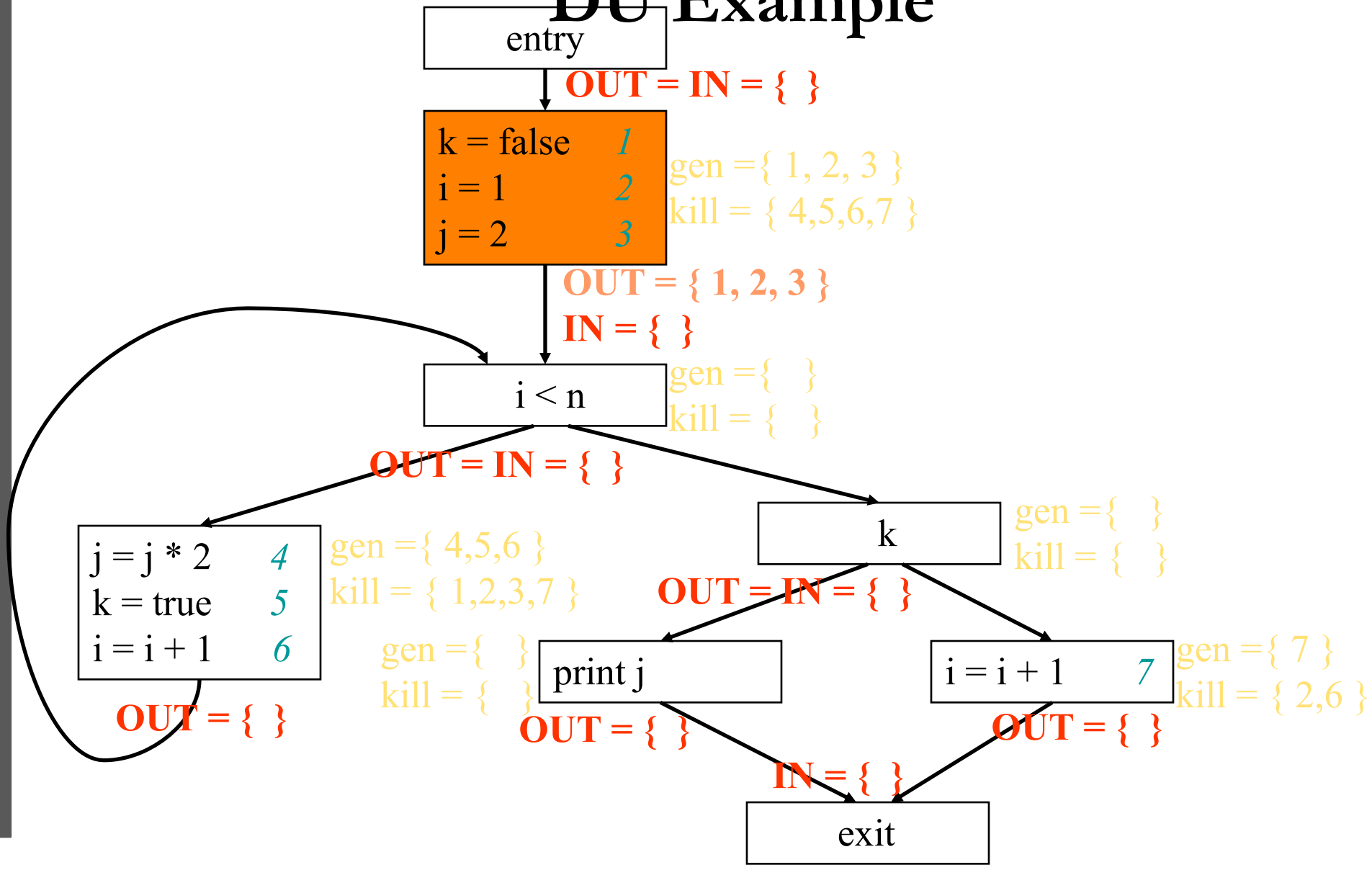
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { }
IN = { }
i < n
gen ={ }
kill = { }
OUT = IN = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

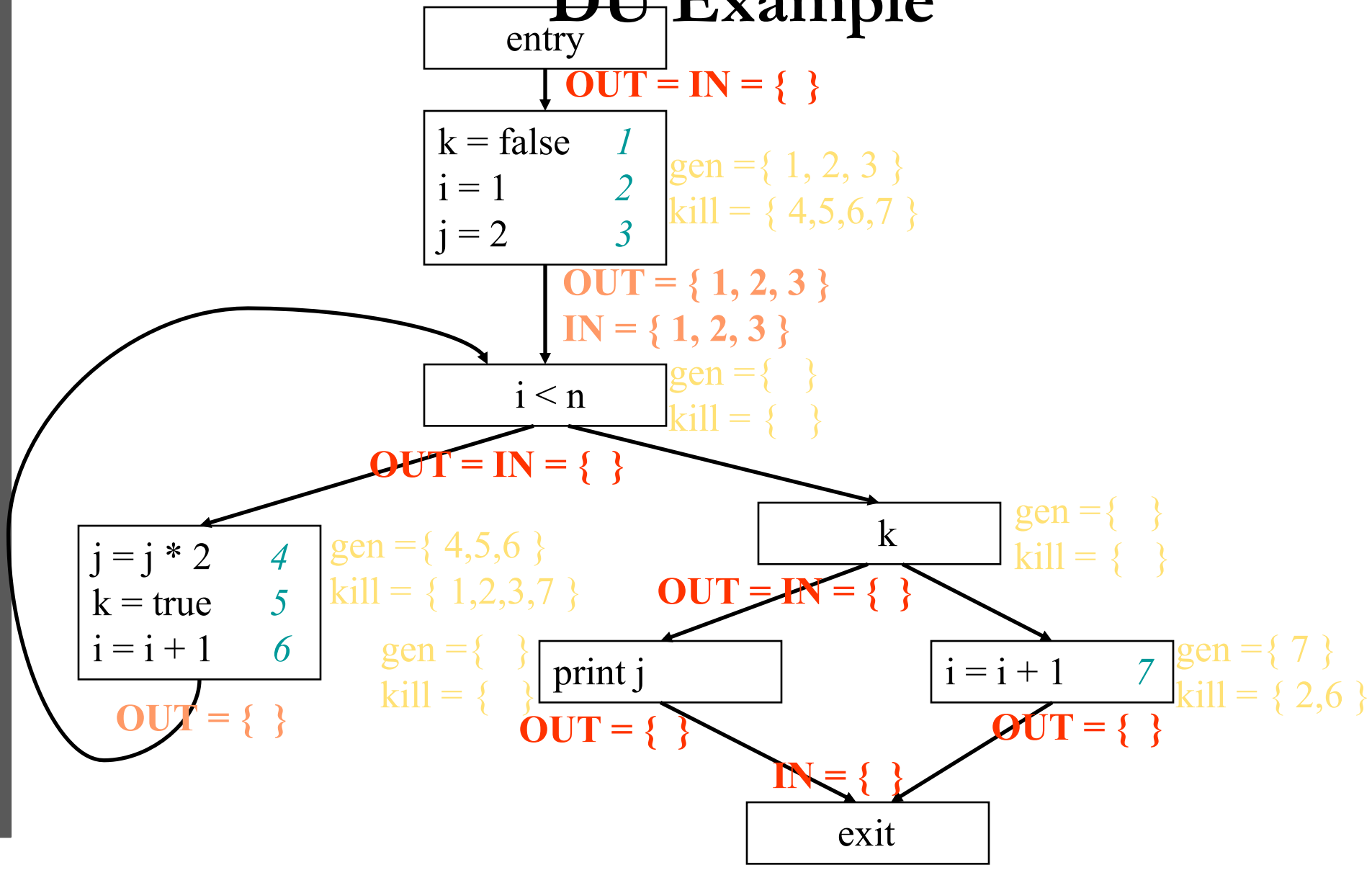
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { }
IN = { }
i < n
gen ={ }
kill = { }
OUT = IN = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

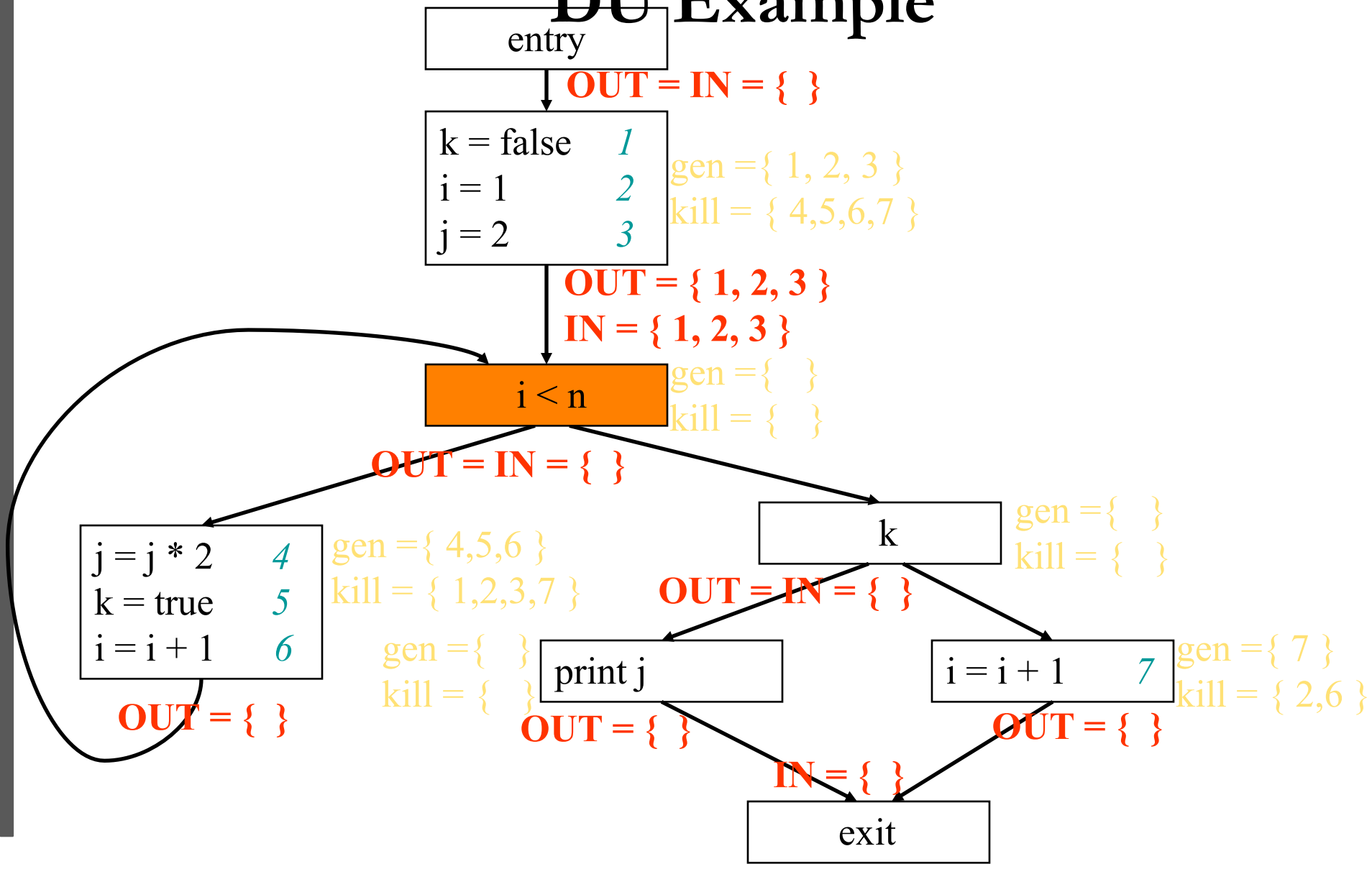
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { }
i < n
gen ={ }
kill = { }
OUT = IN = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

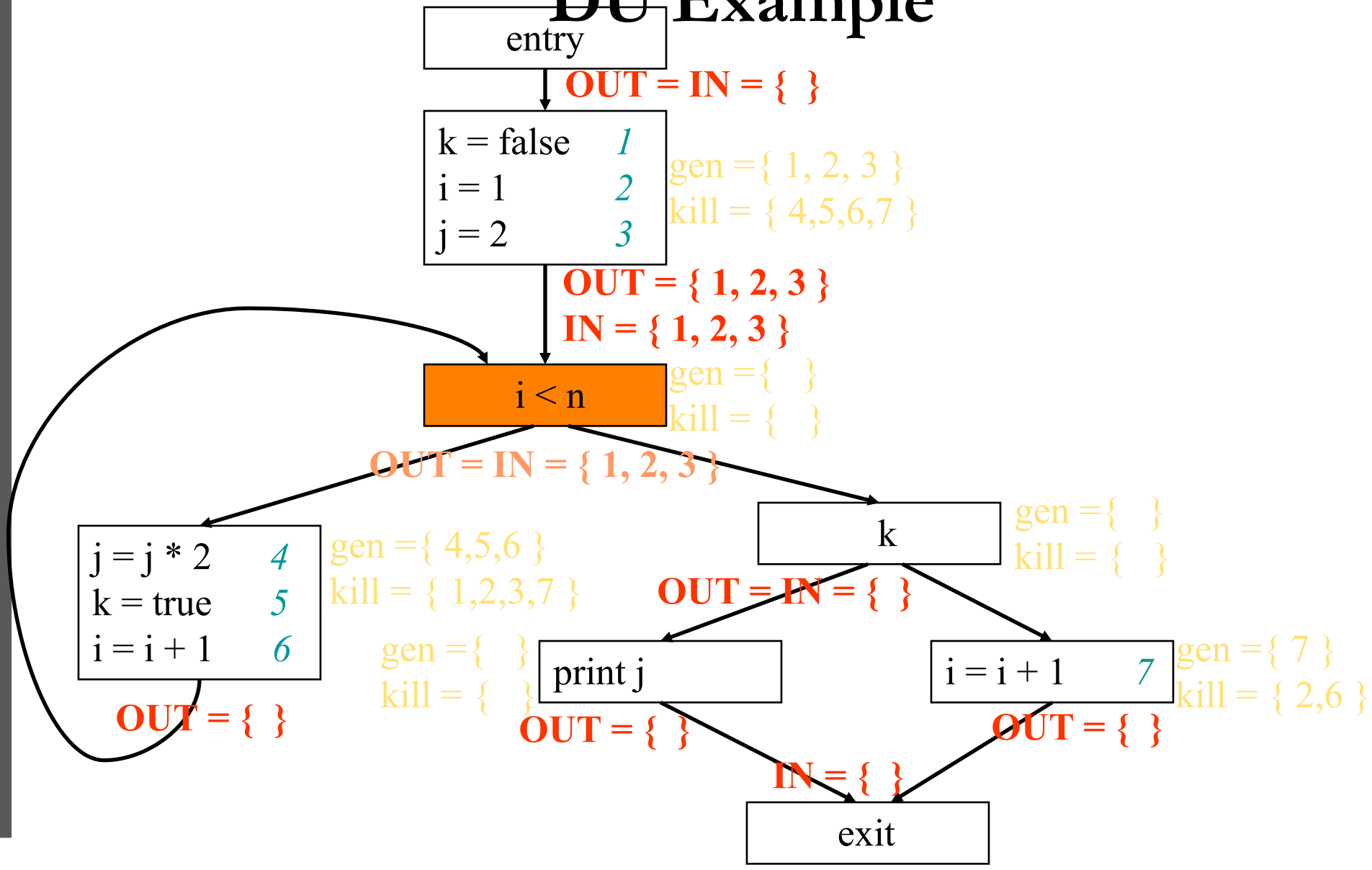
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

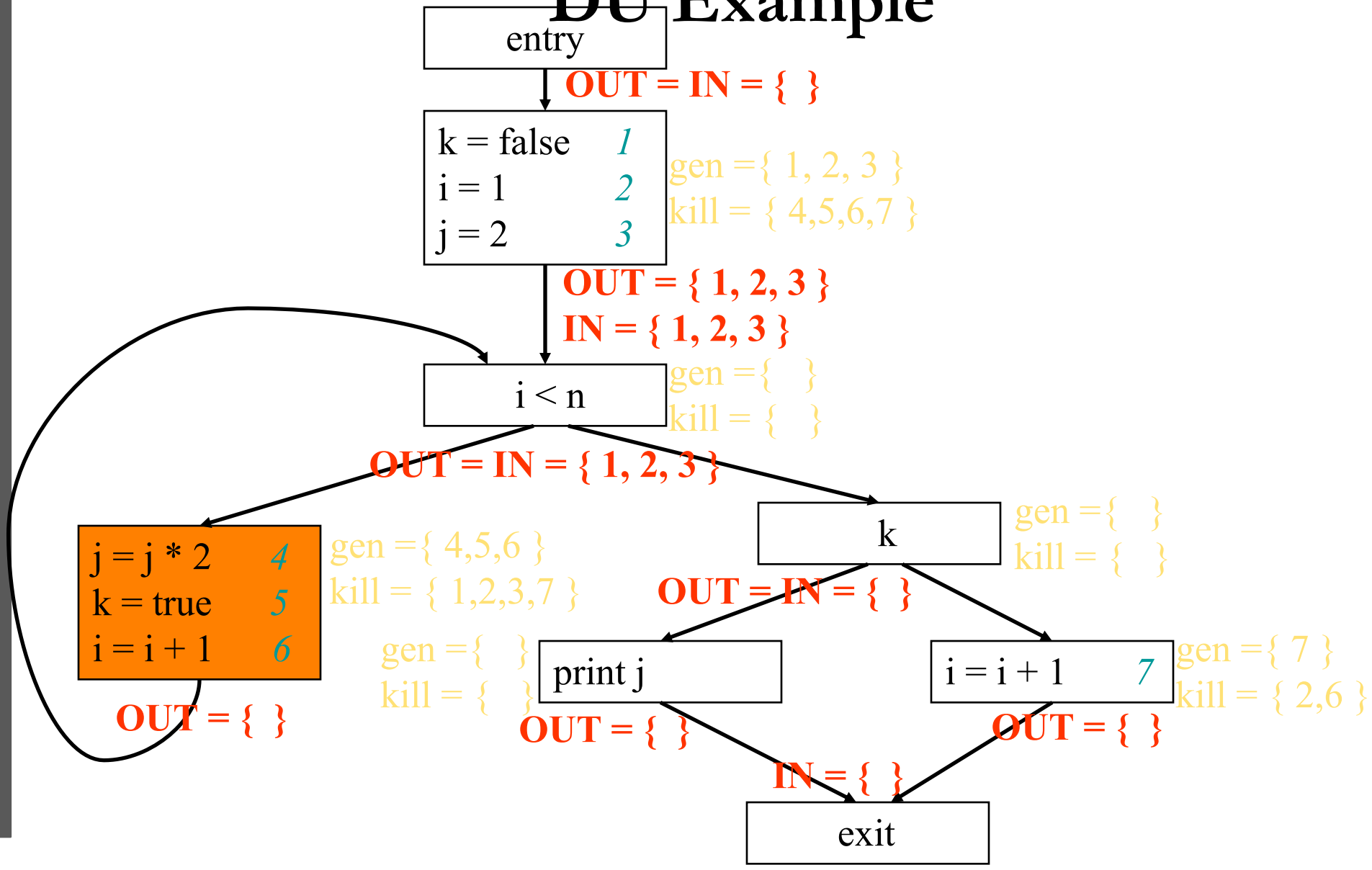
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

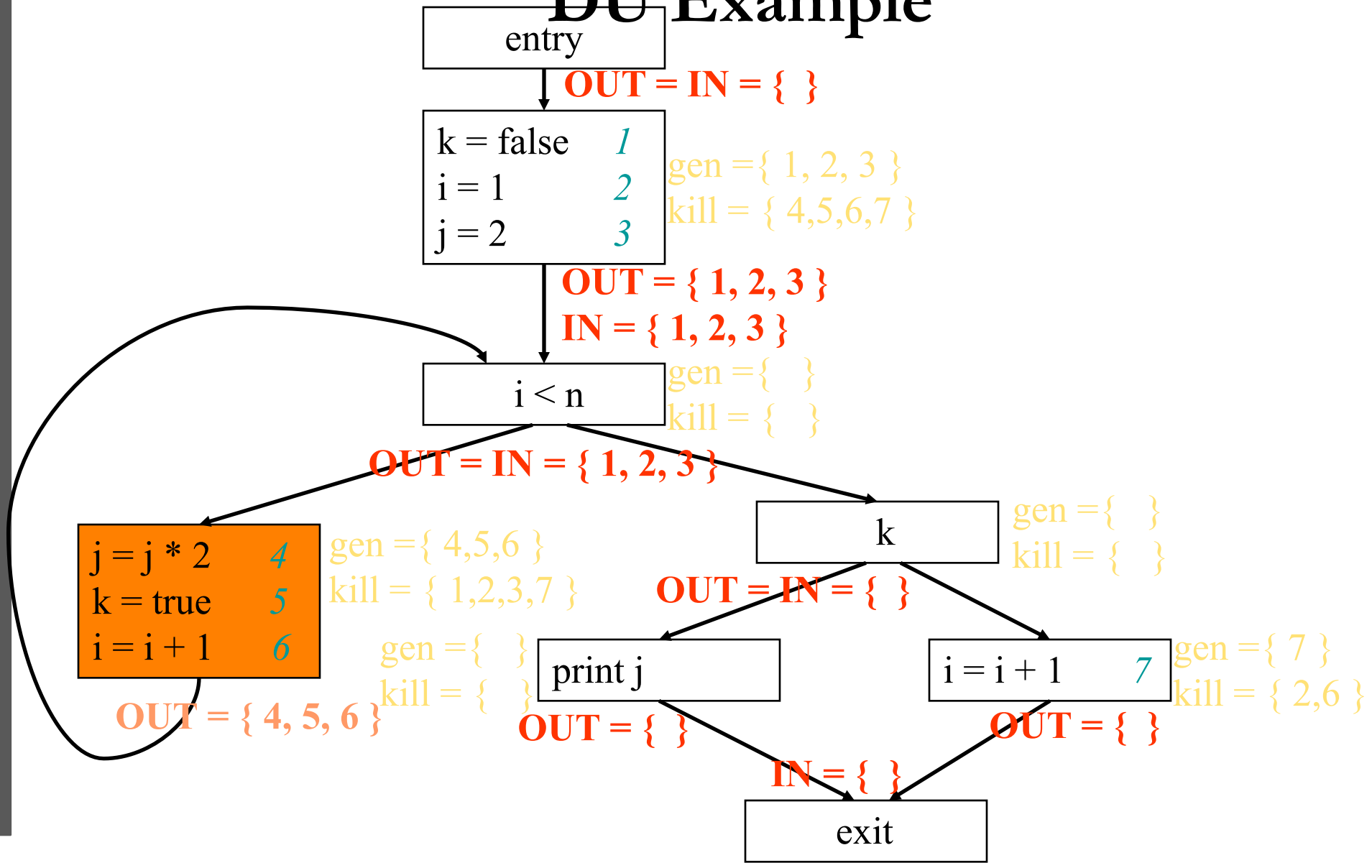
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

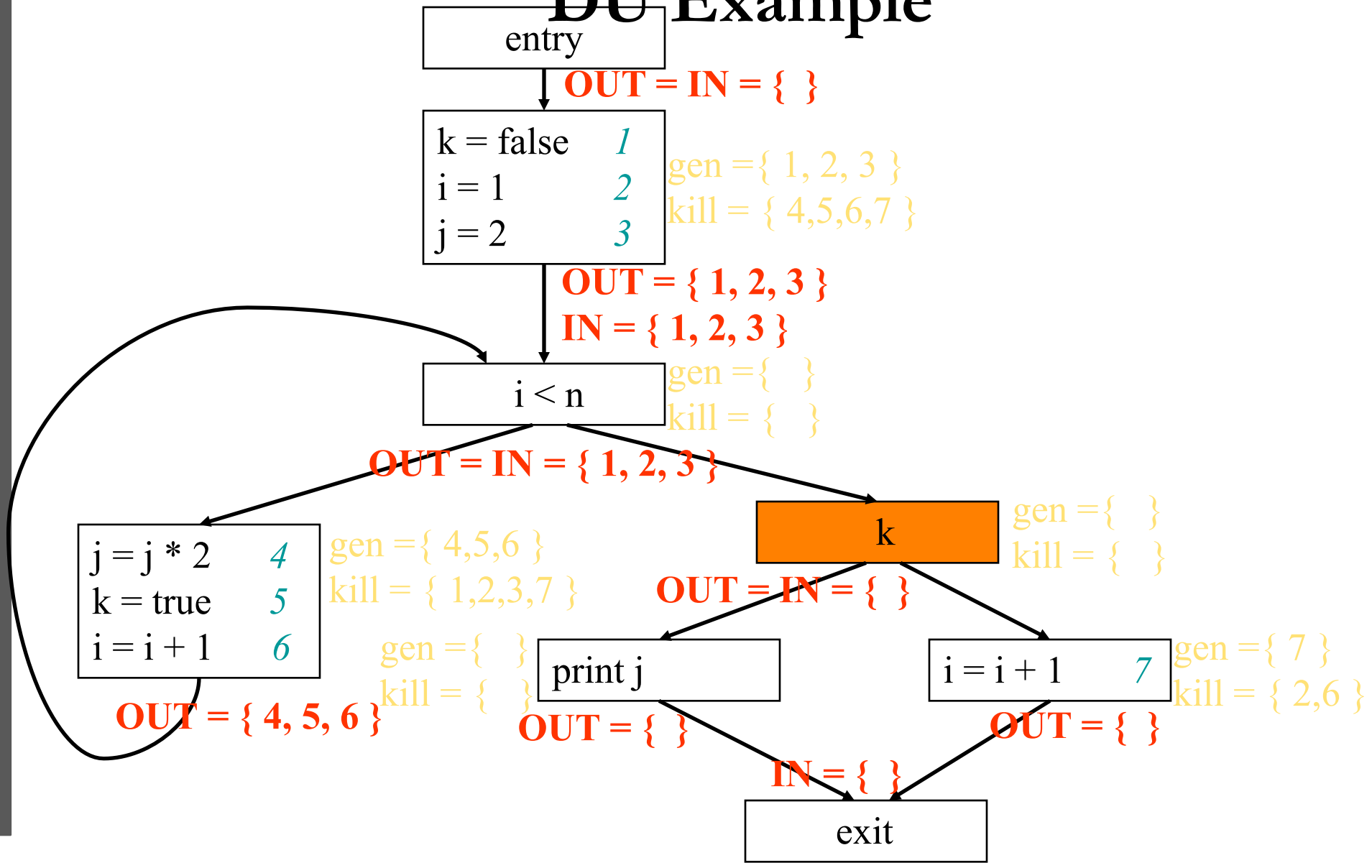
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

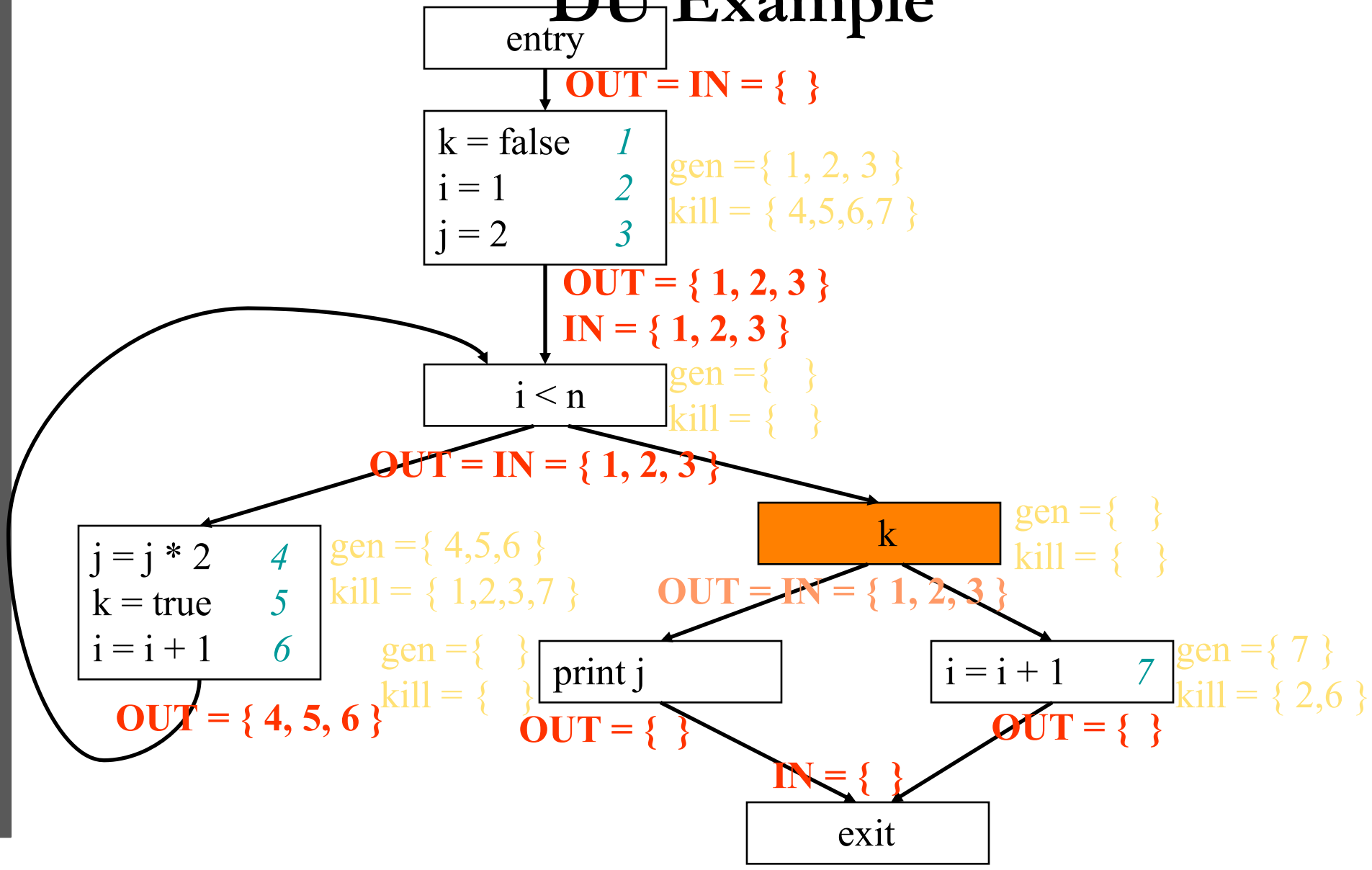
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

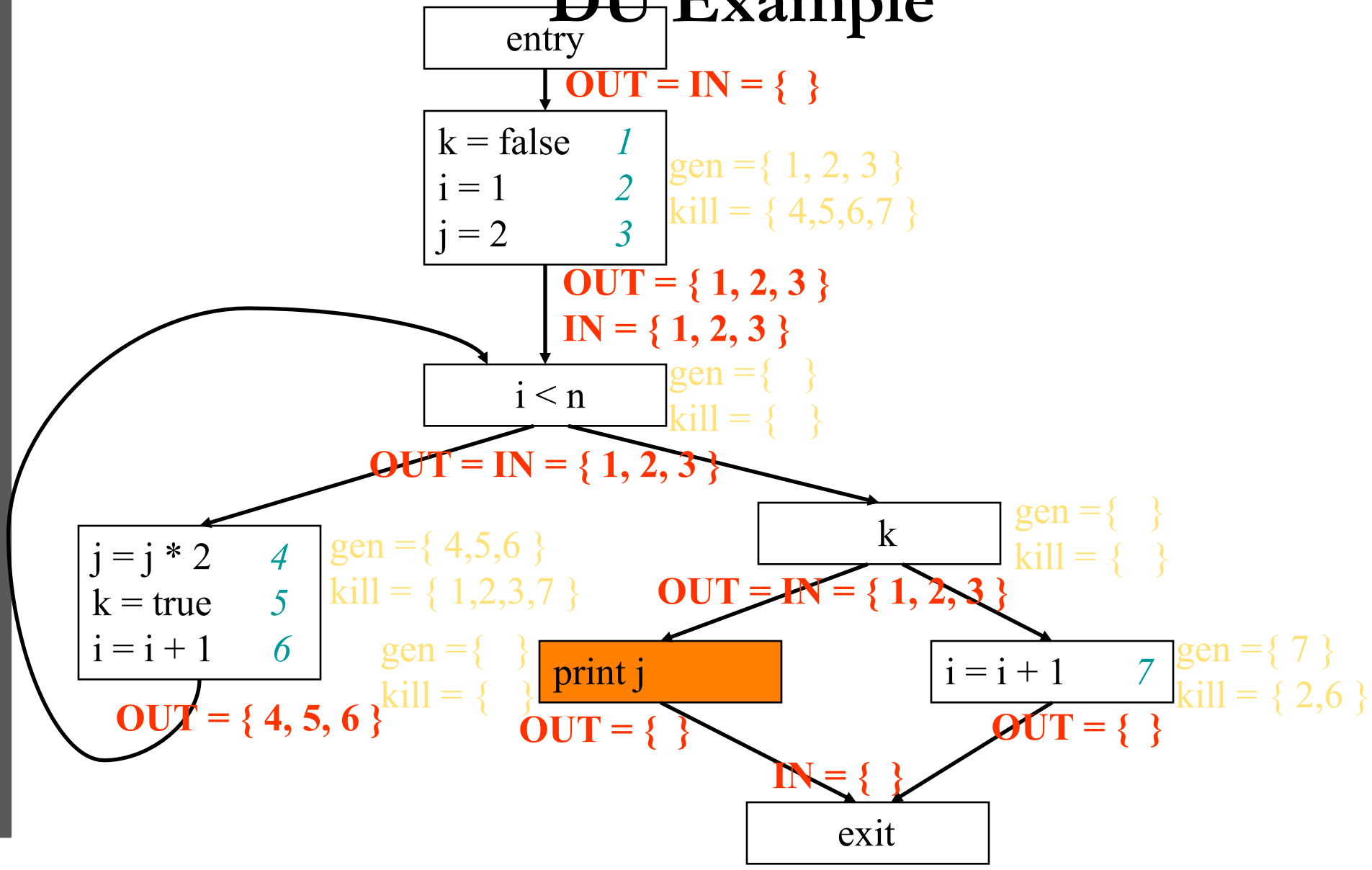
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

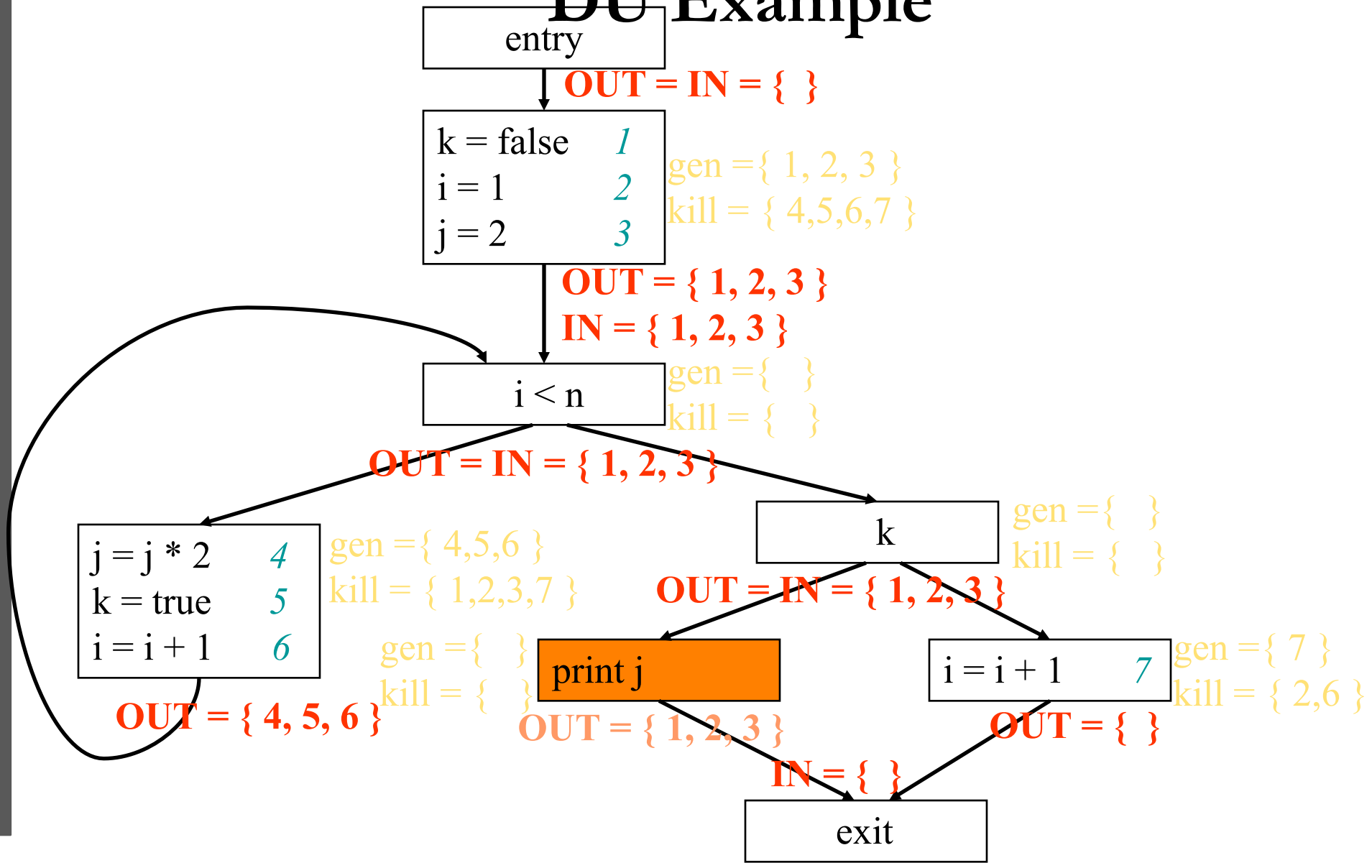
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

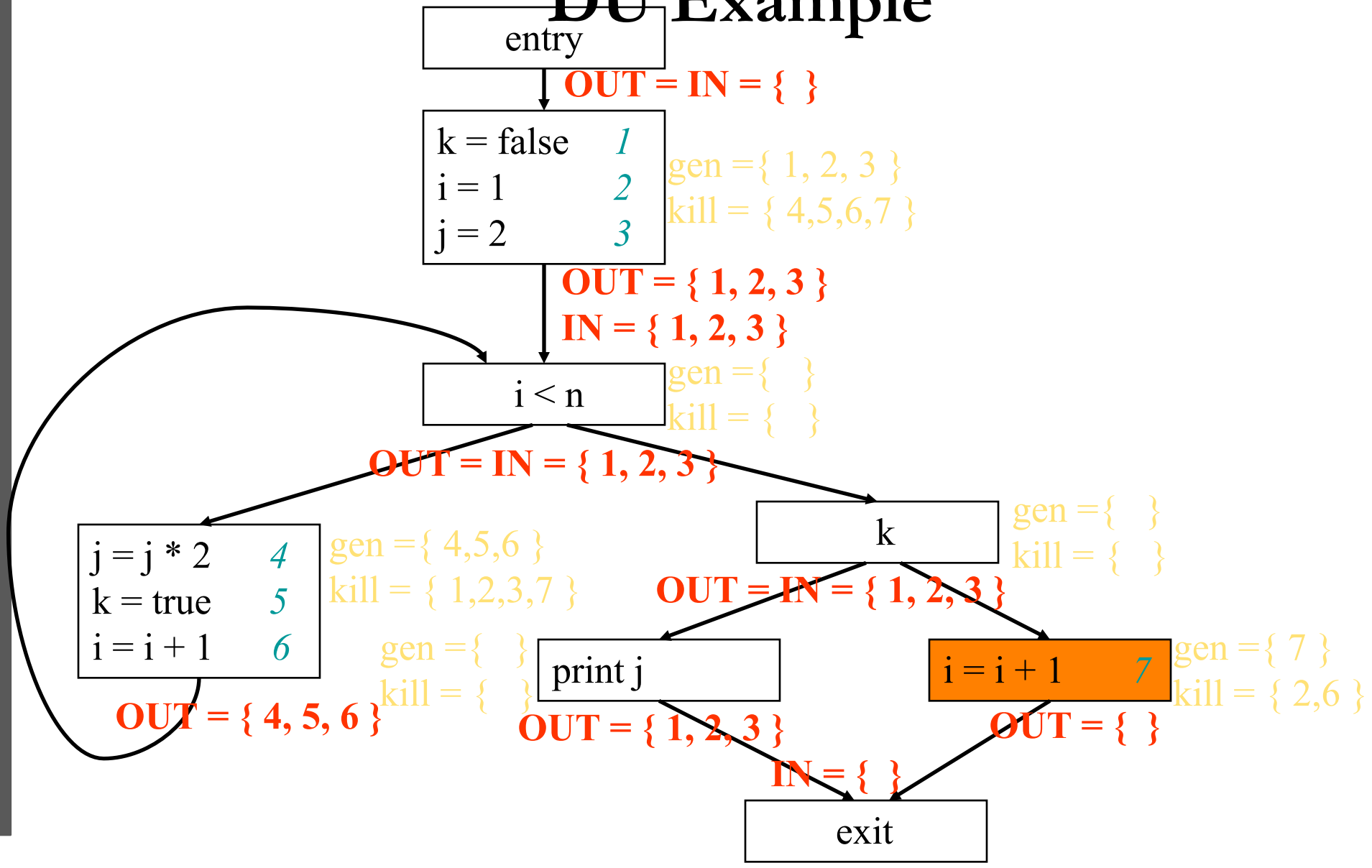
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

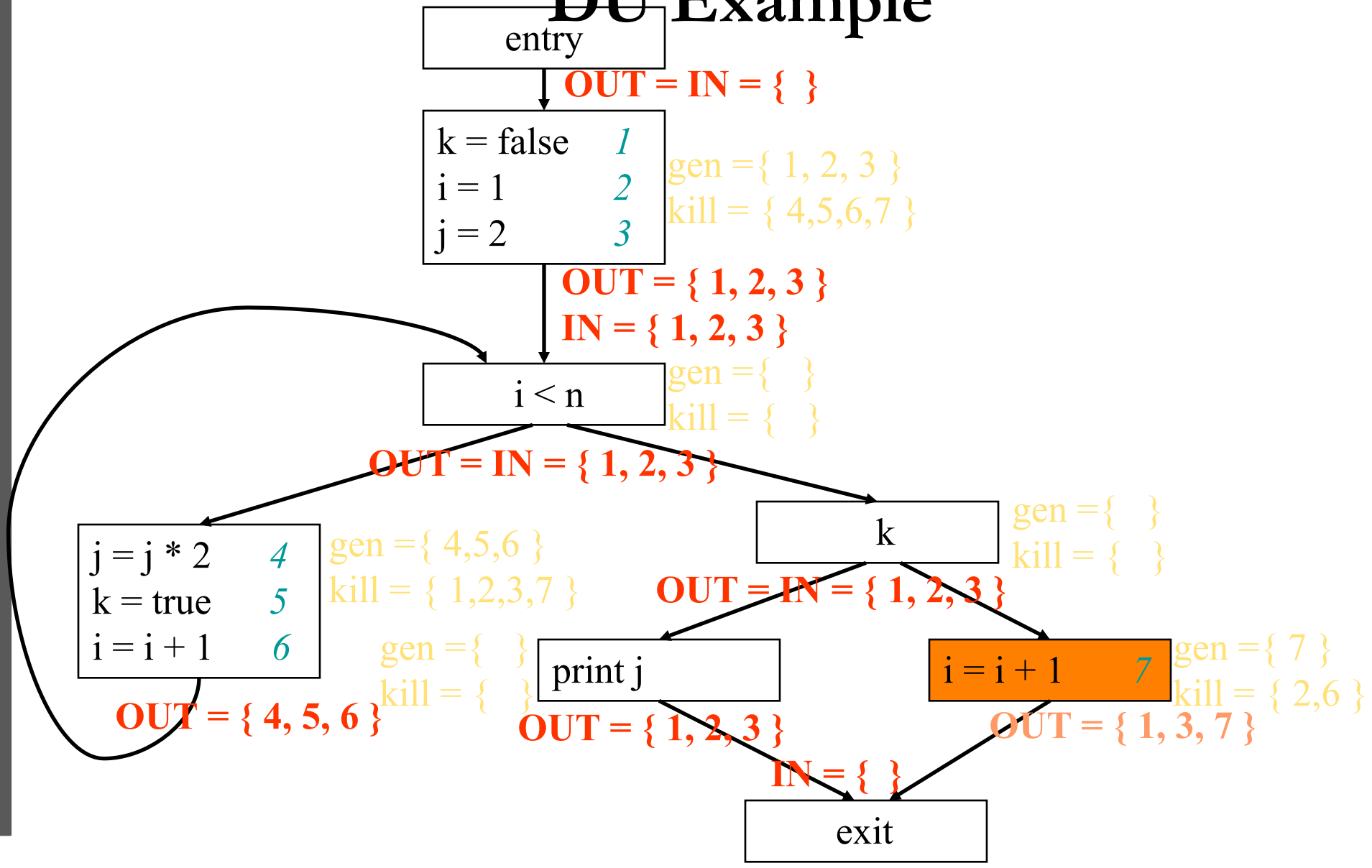
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
gen ={ }
kill = { }
print j
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

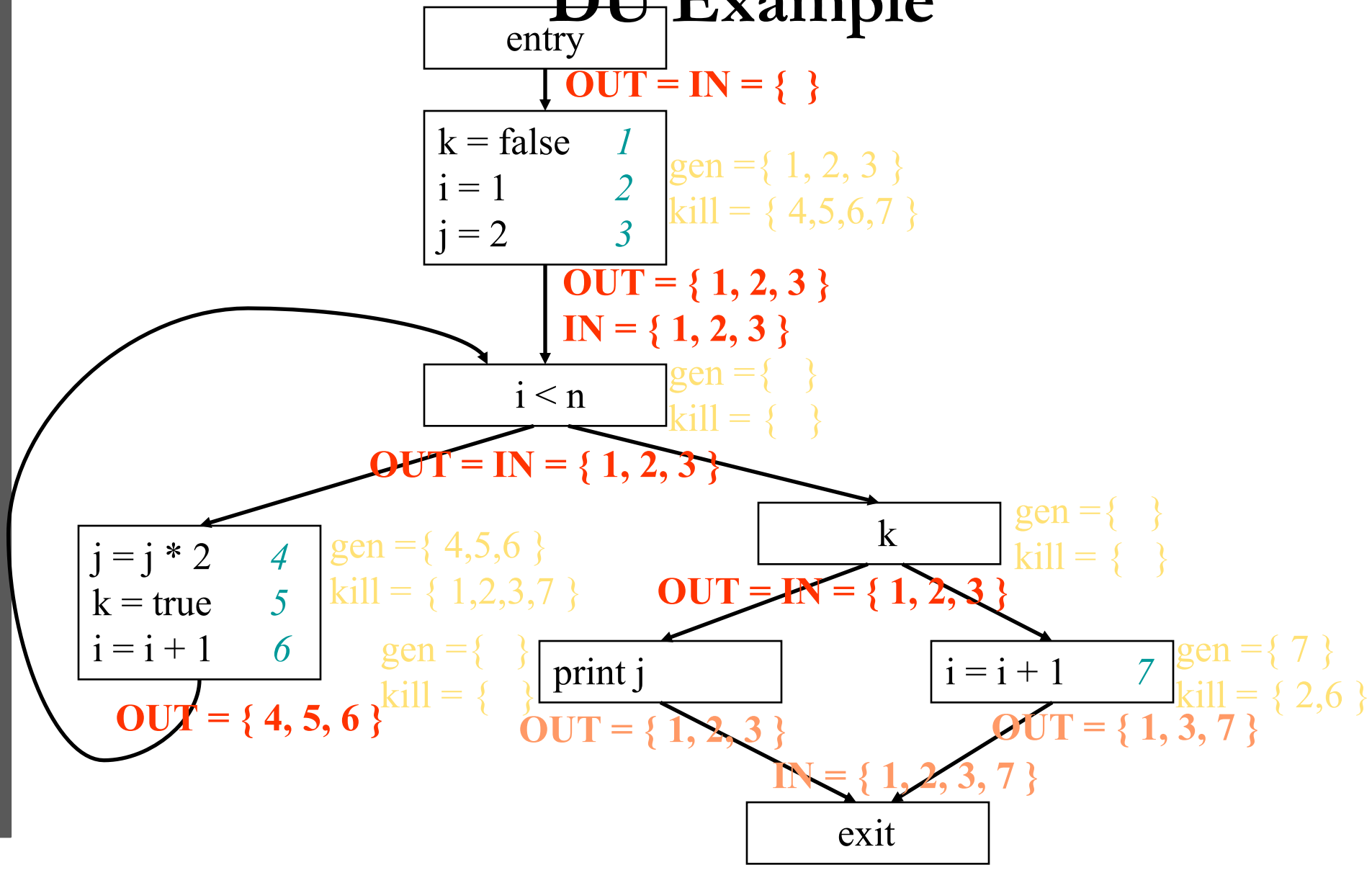
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

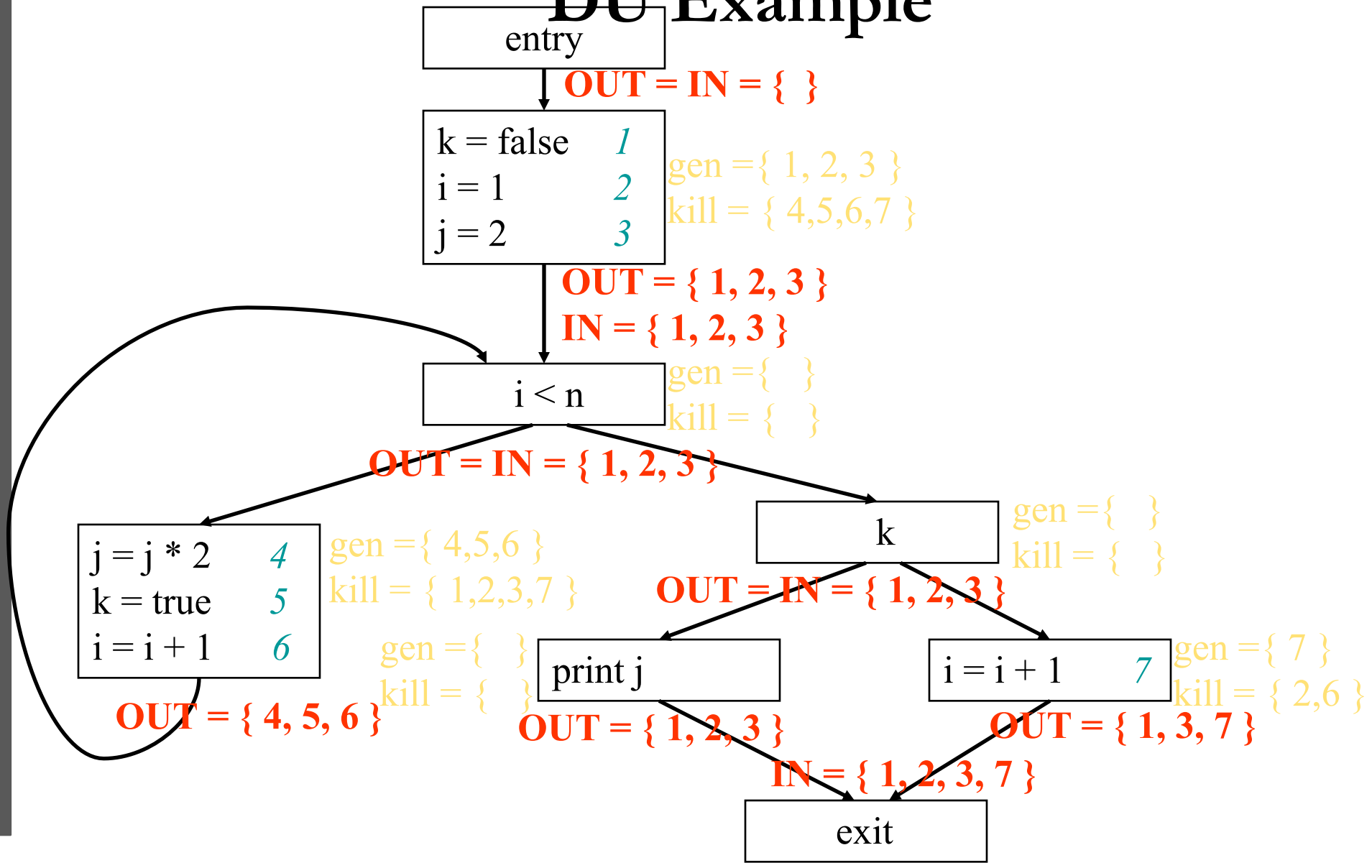
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { }
exit

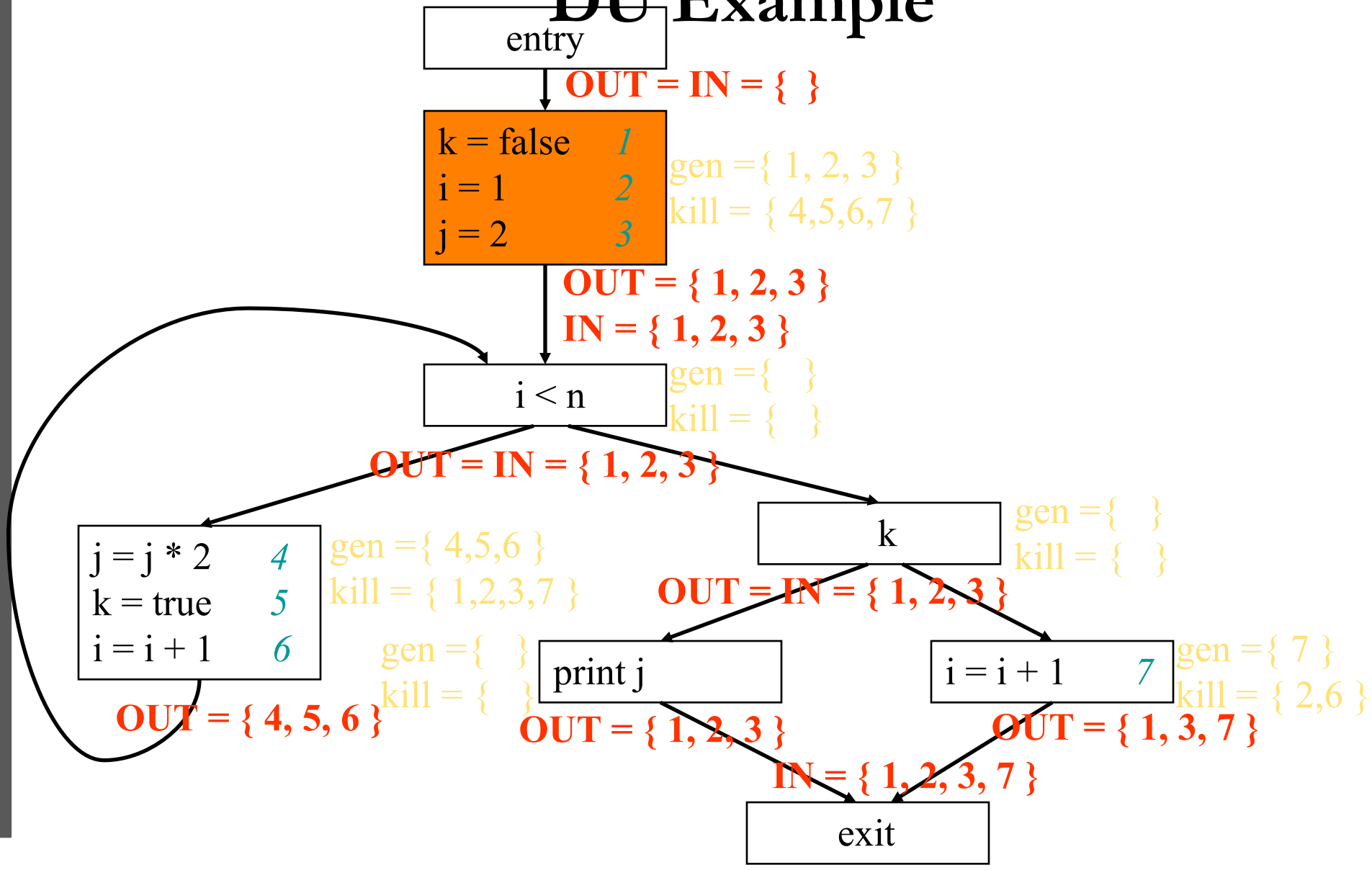
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

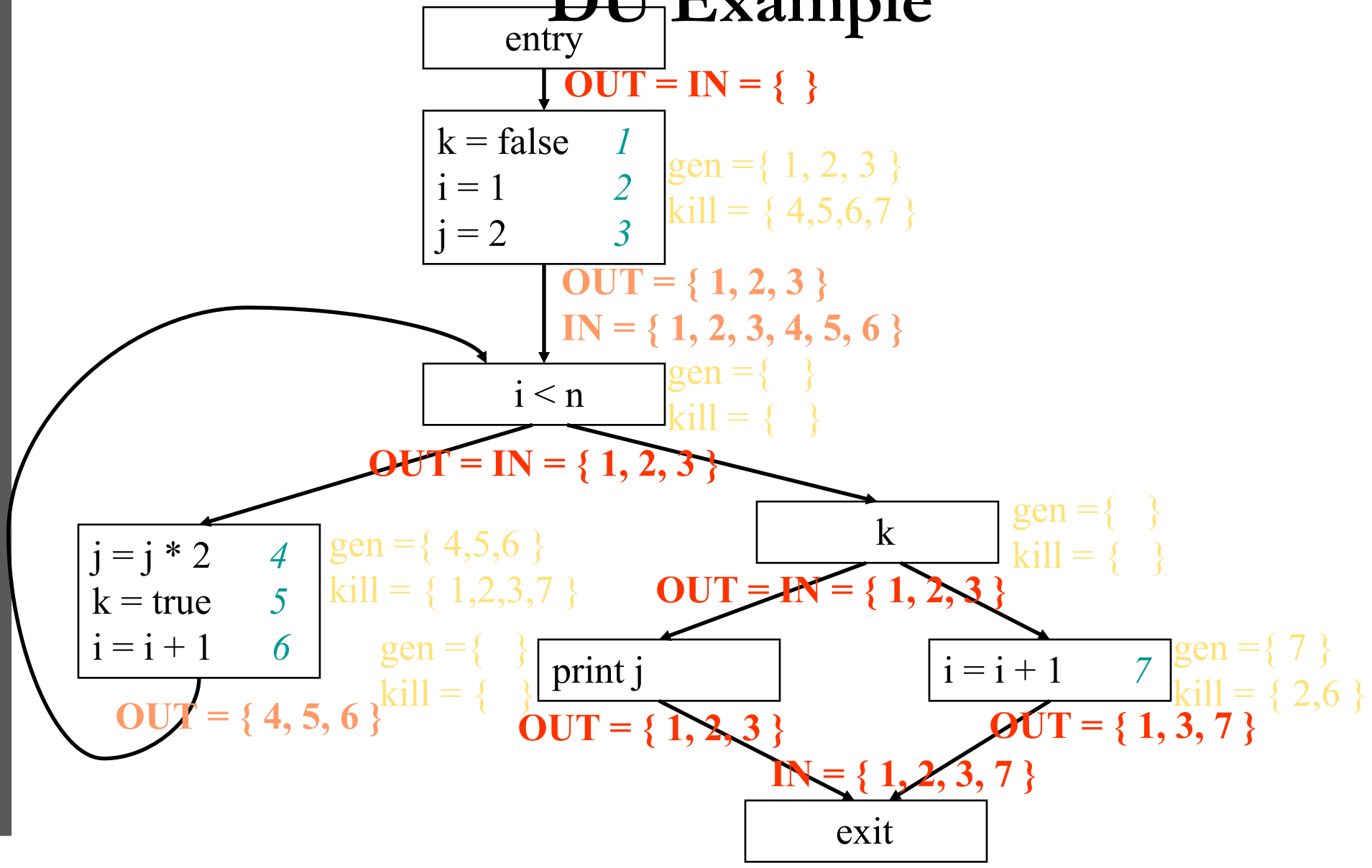
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

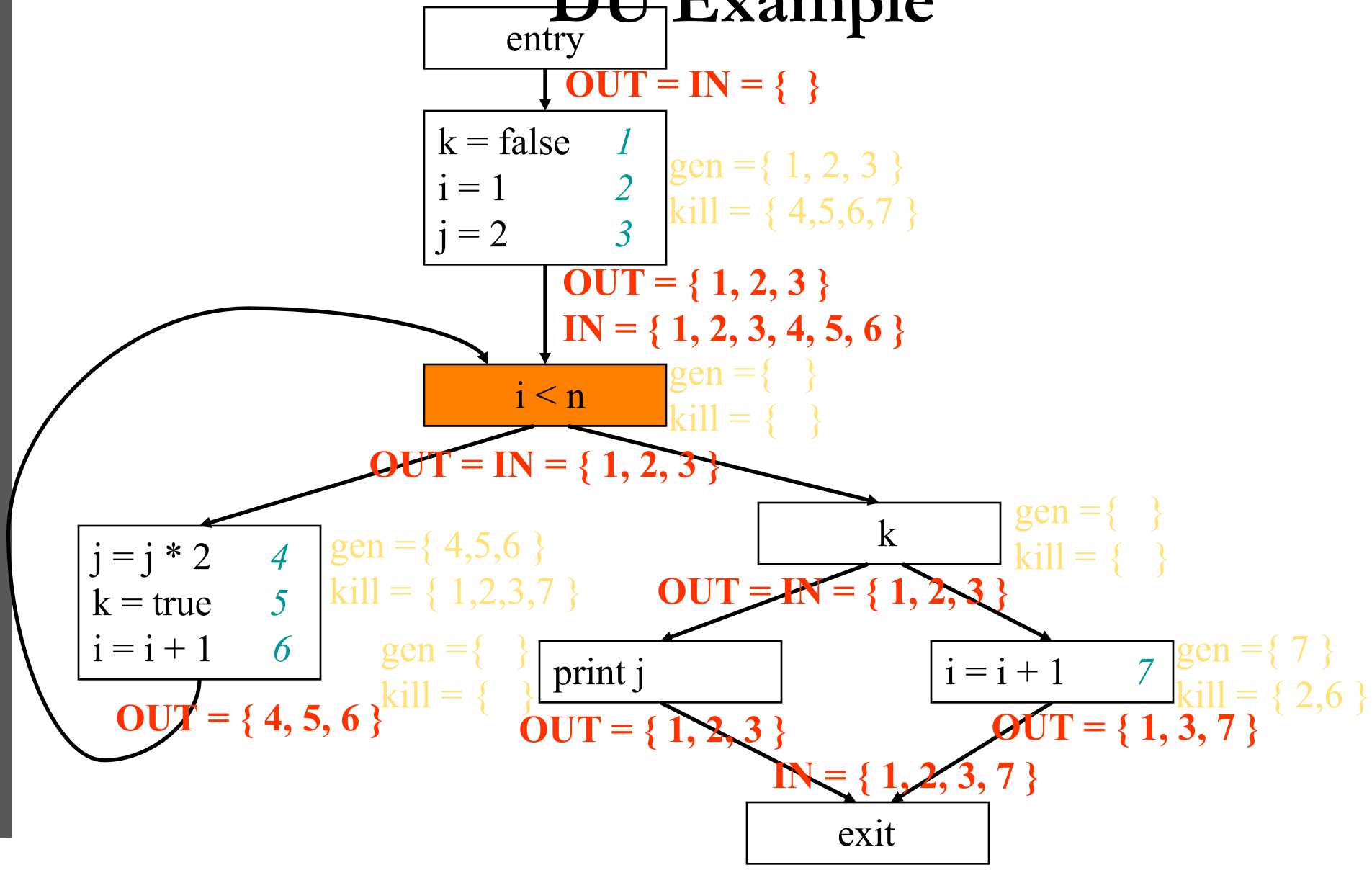
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

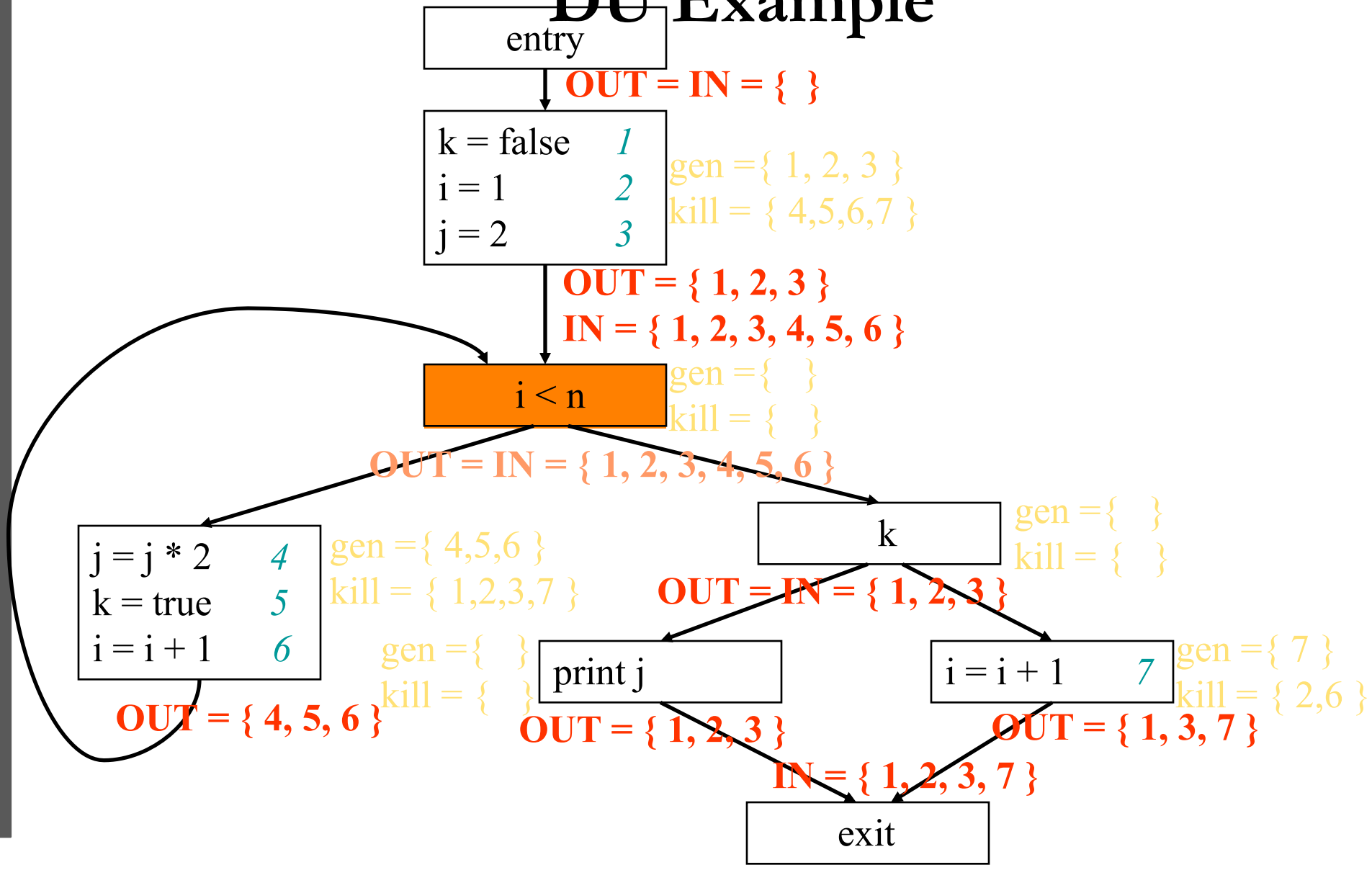
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

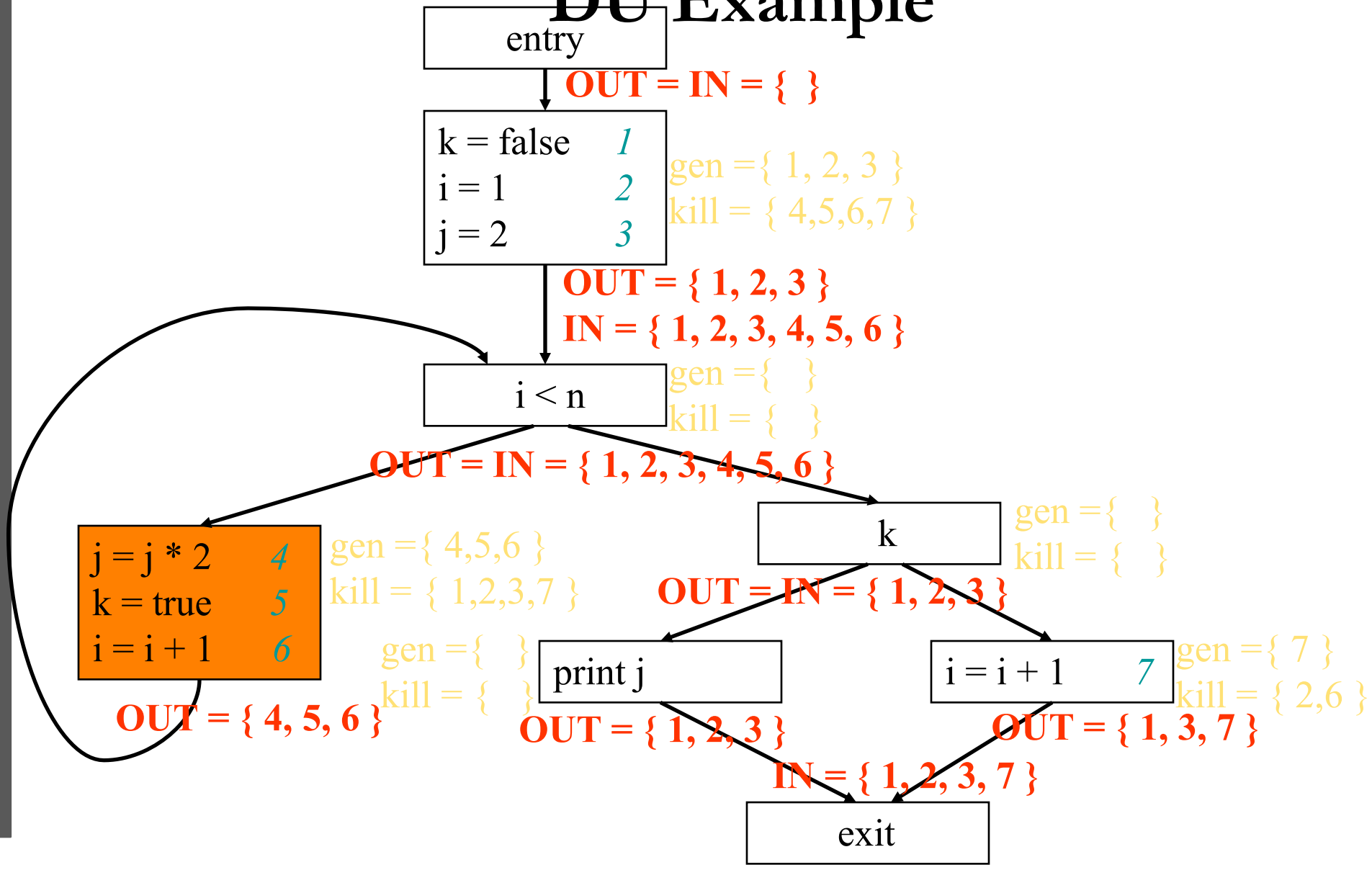
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

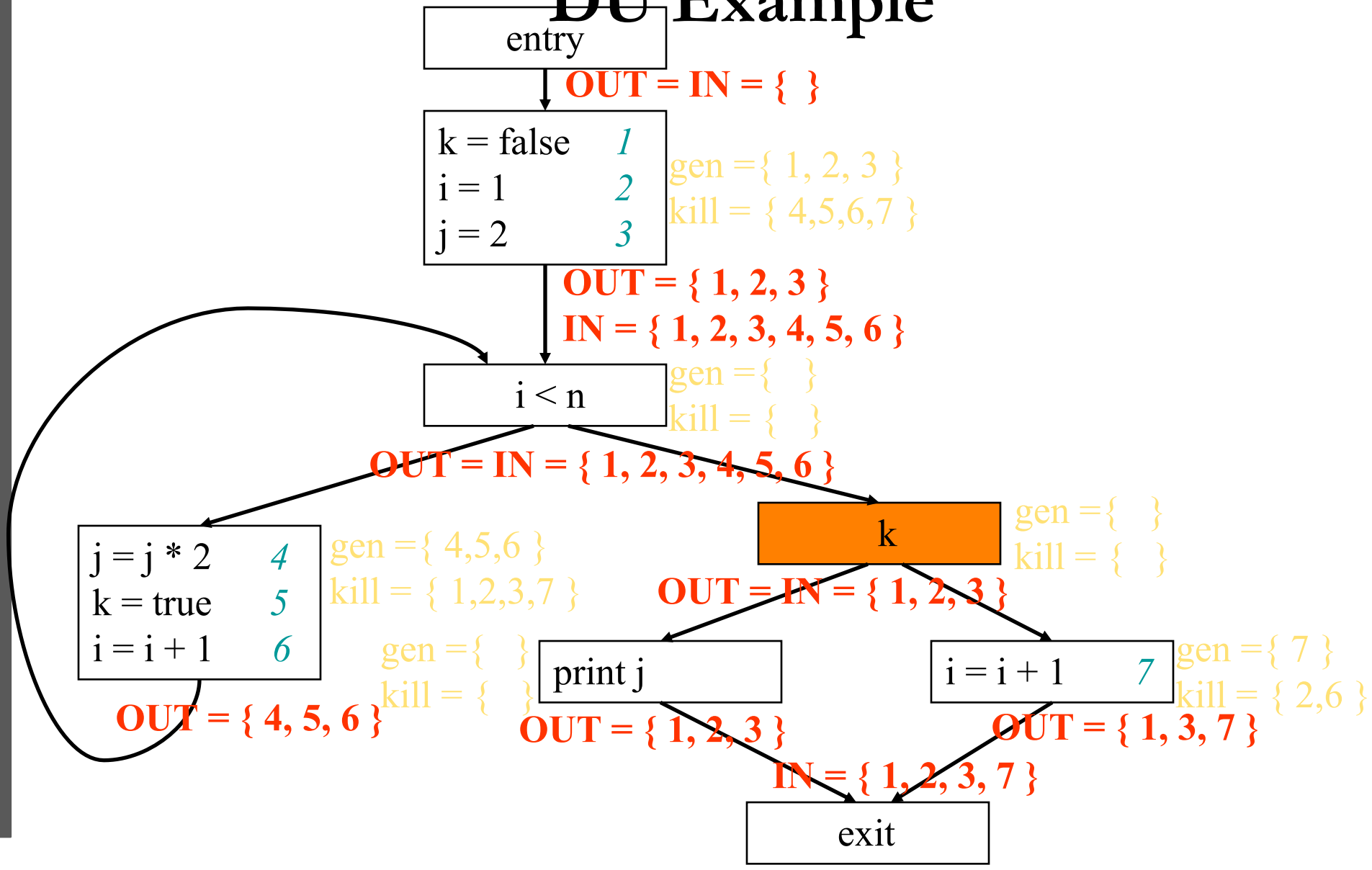
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

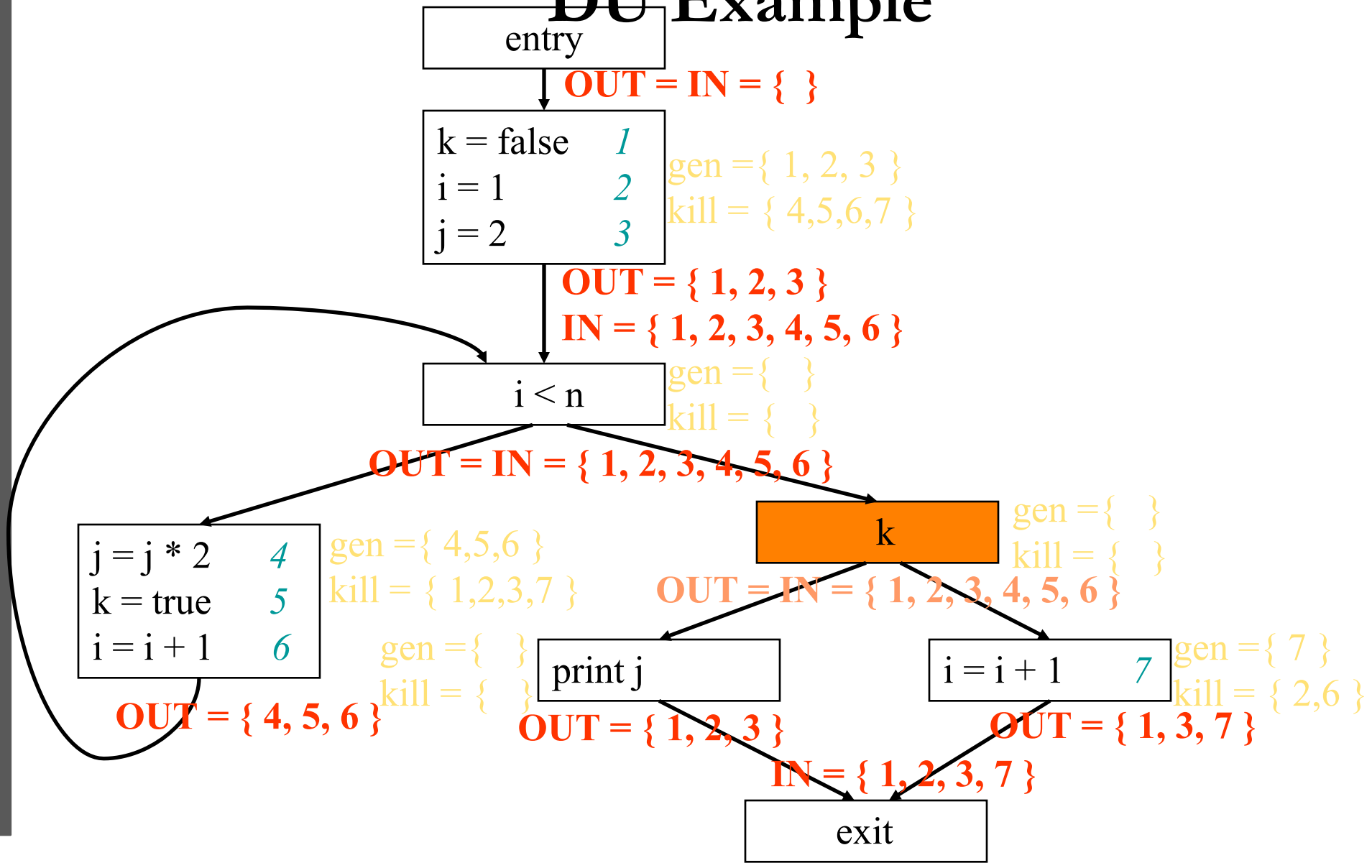
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

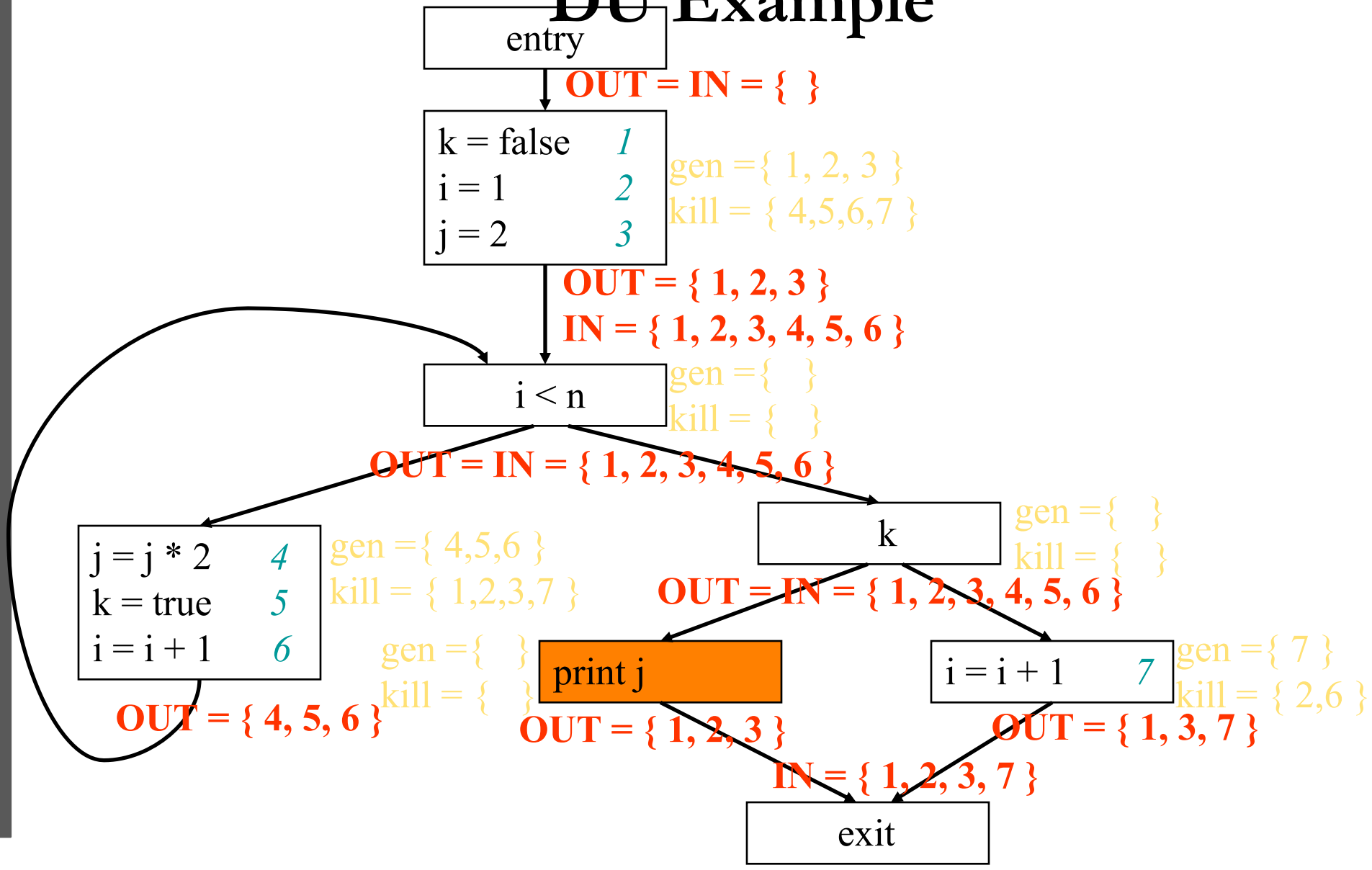
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

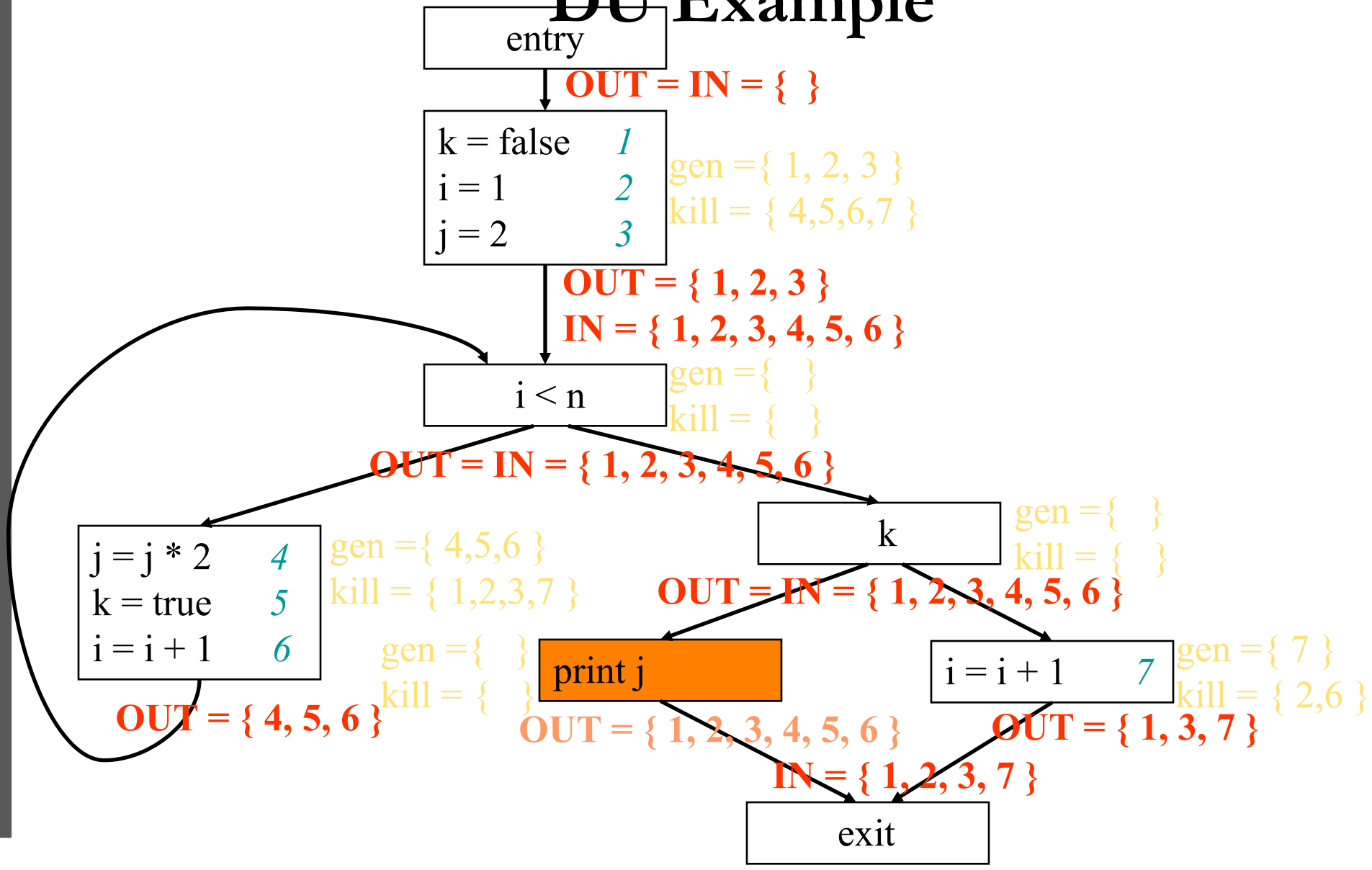
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

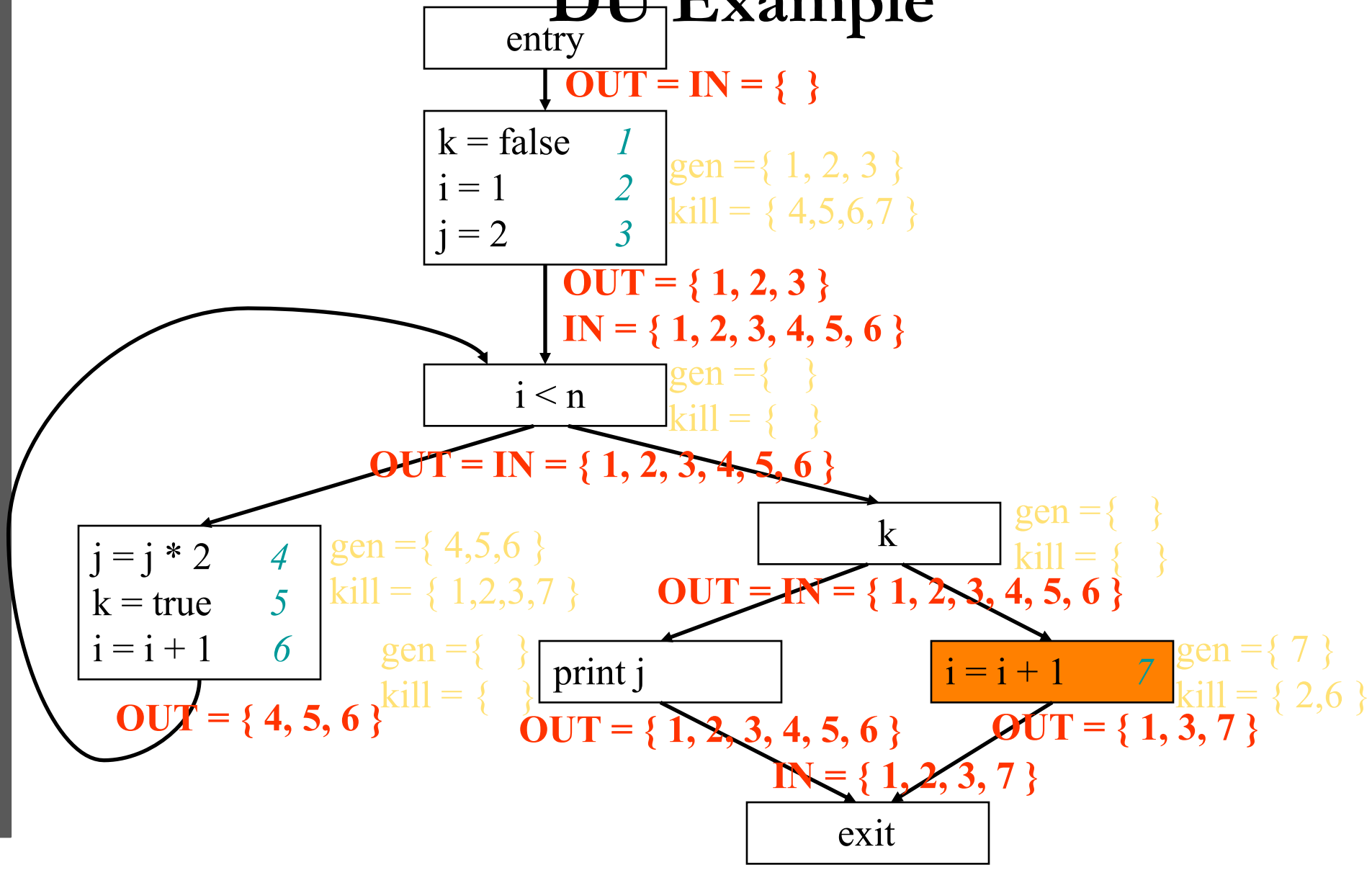
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

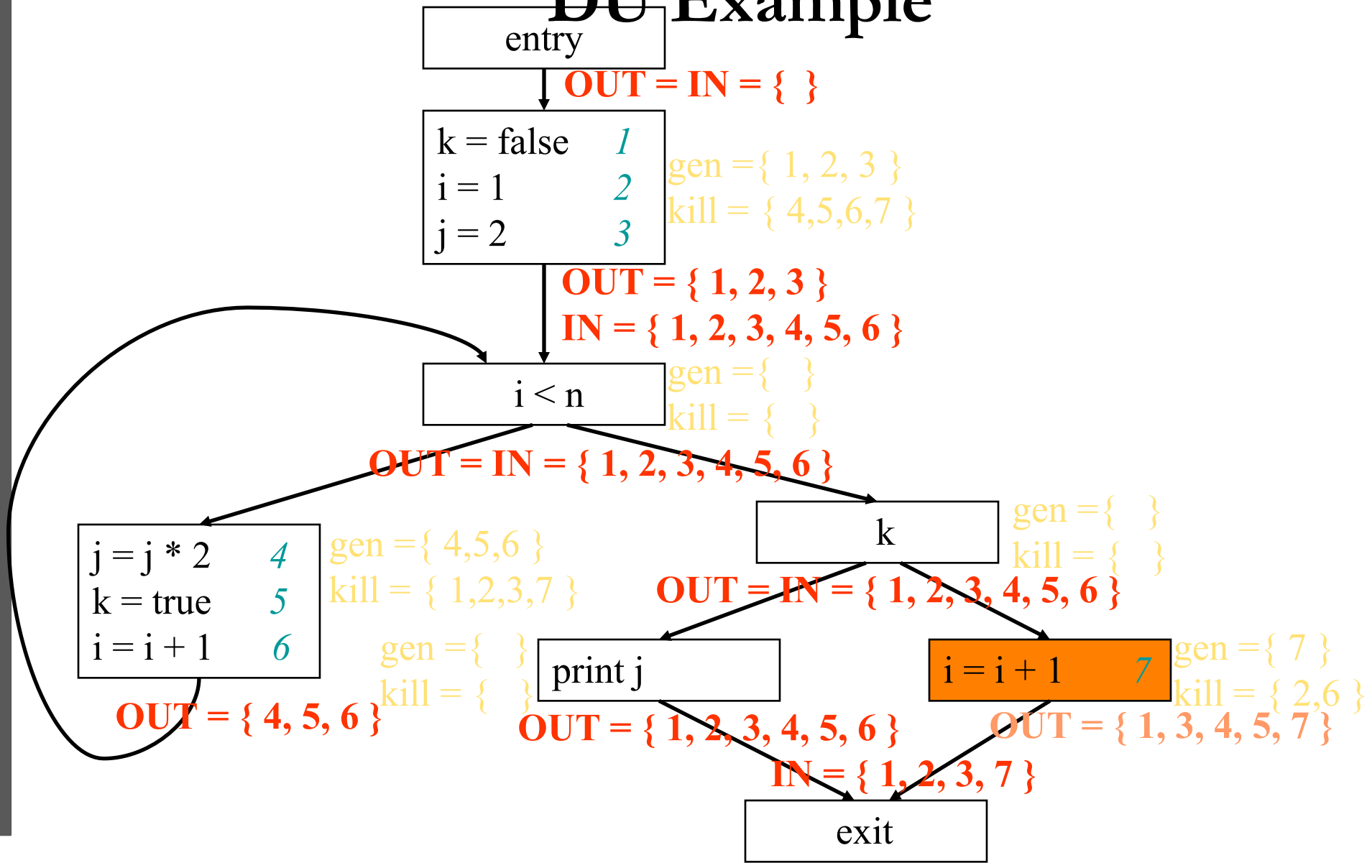
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

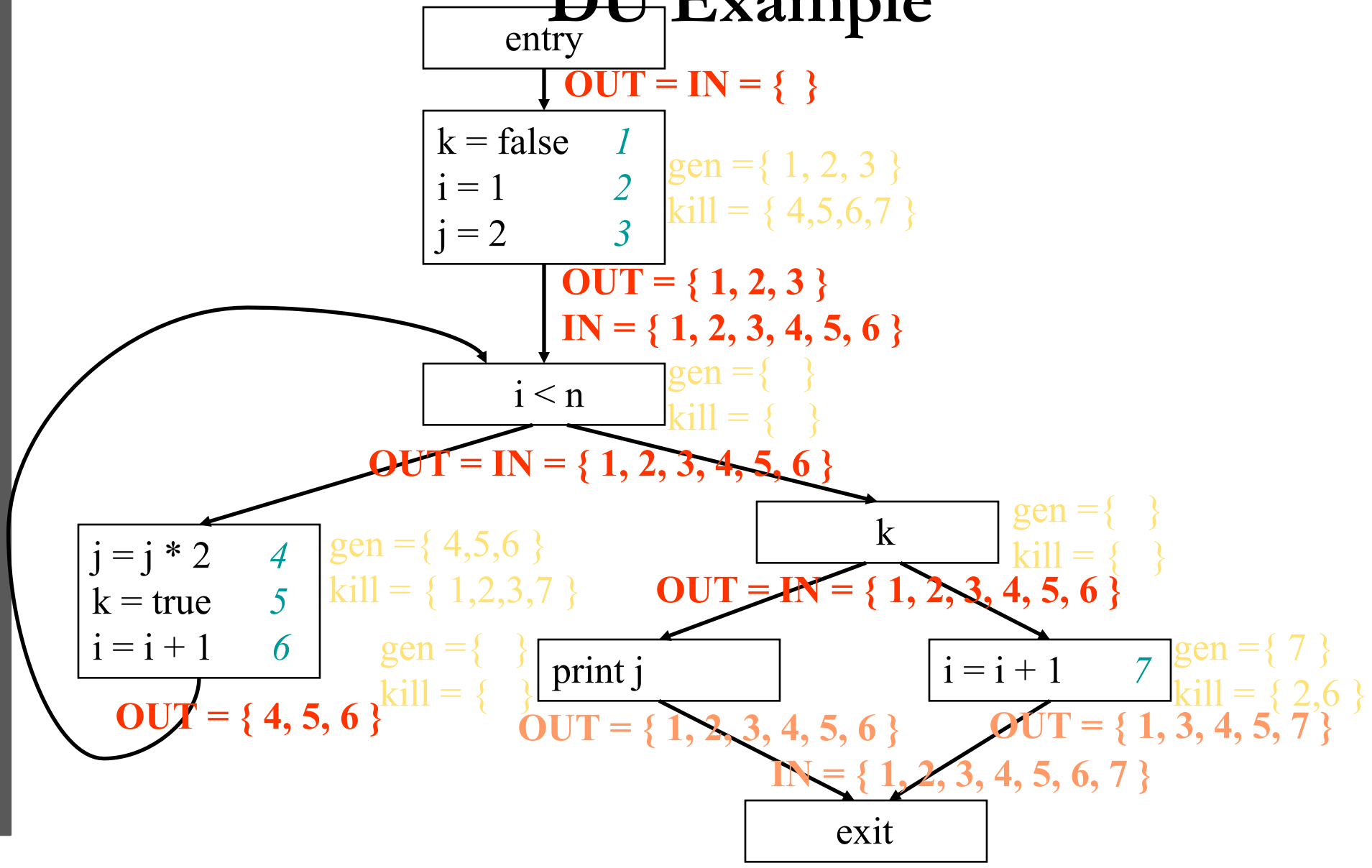
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 7 }
IN = { 1, 2, 3, 7 }
exit

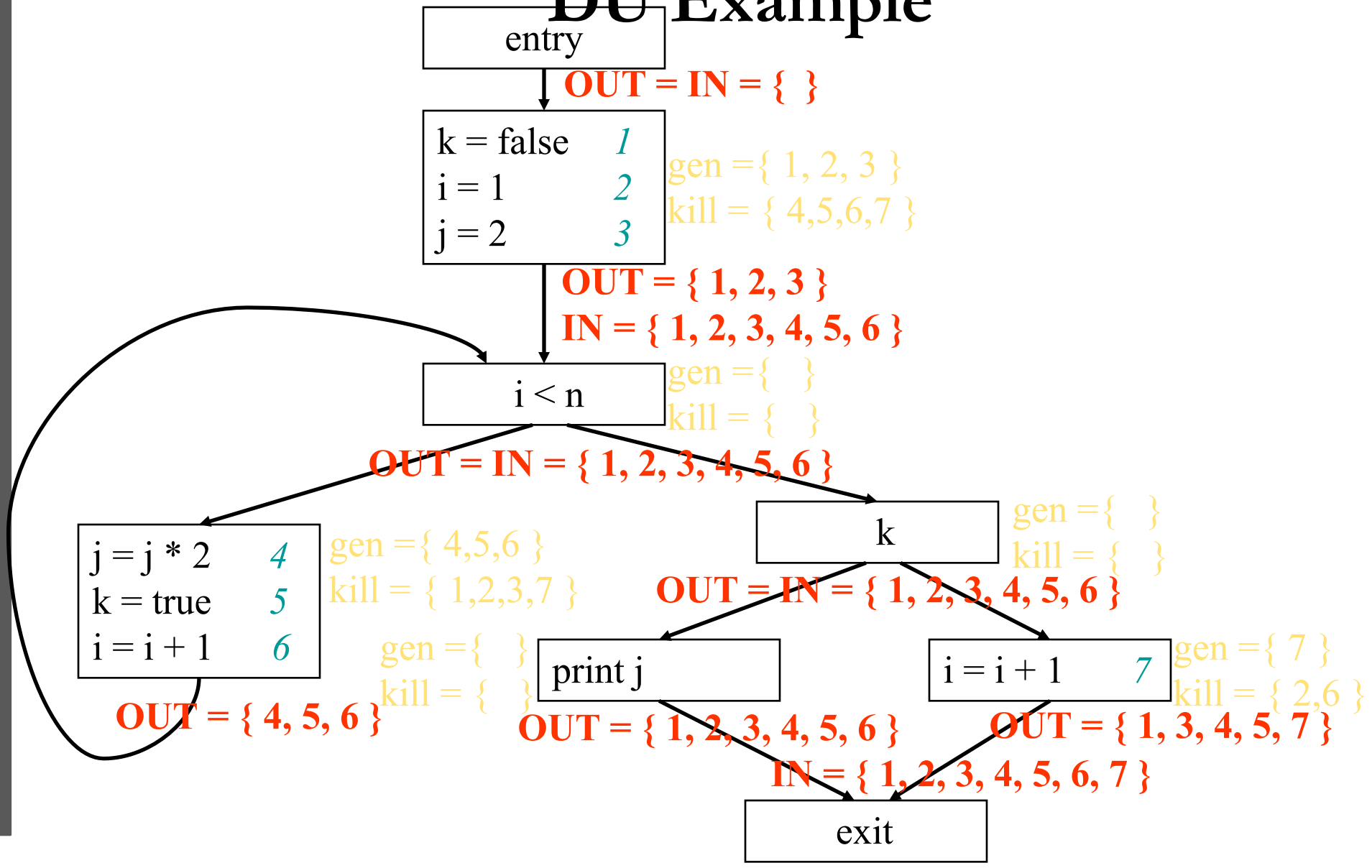
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 7 }
exit

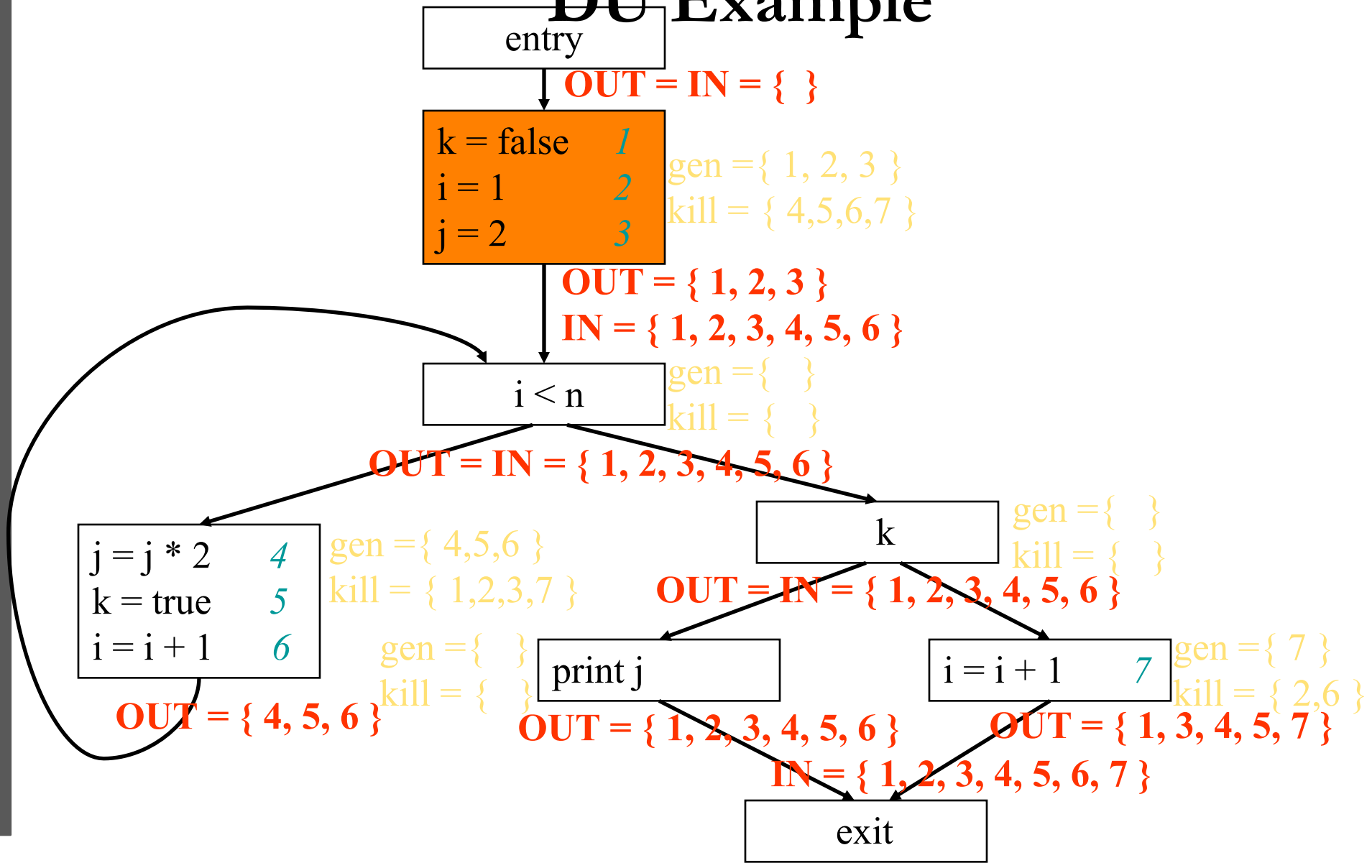
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

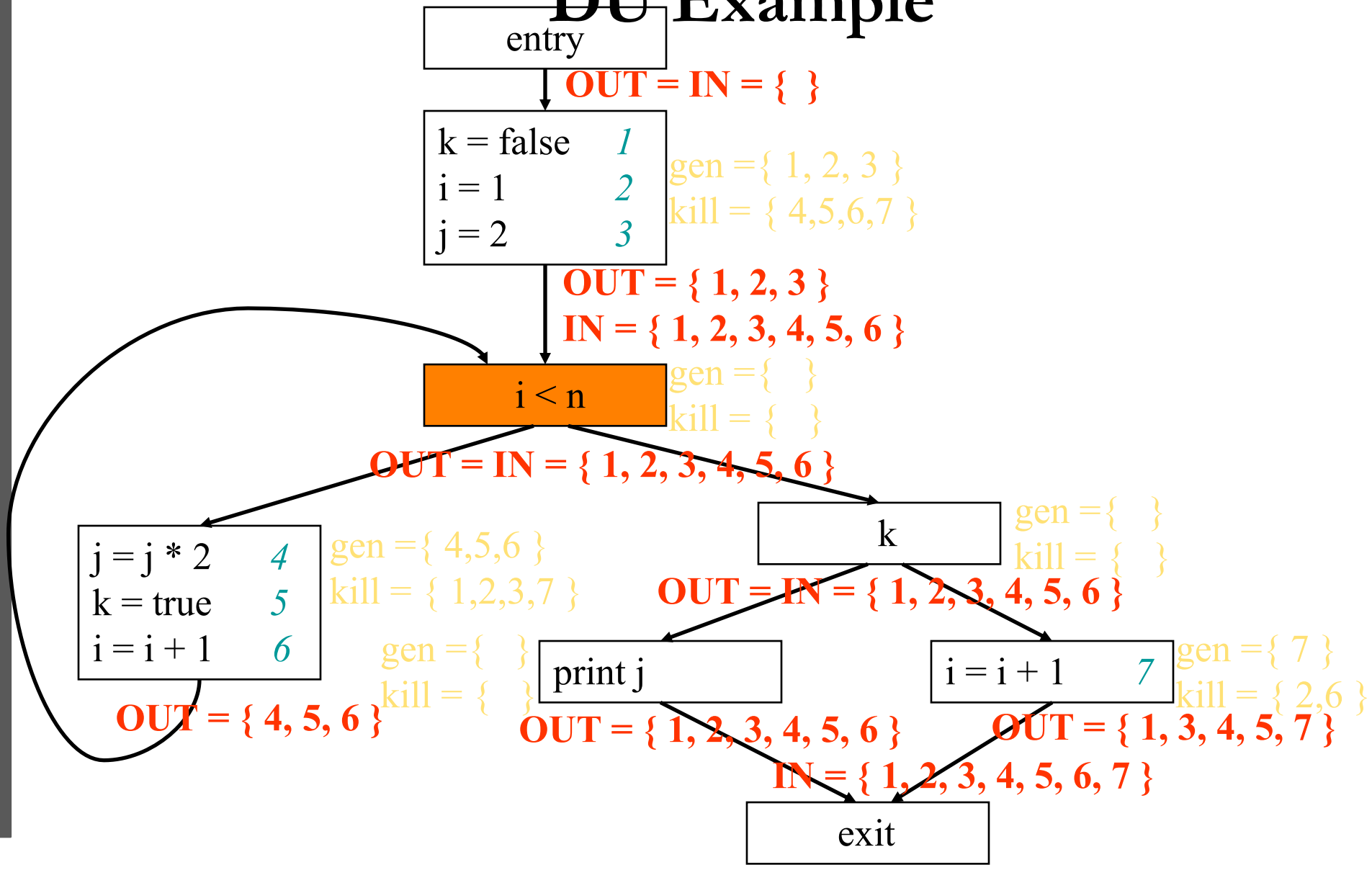
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

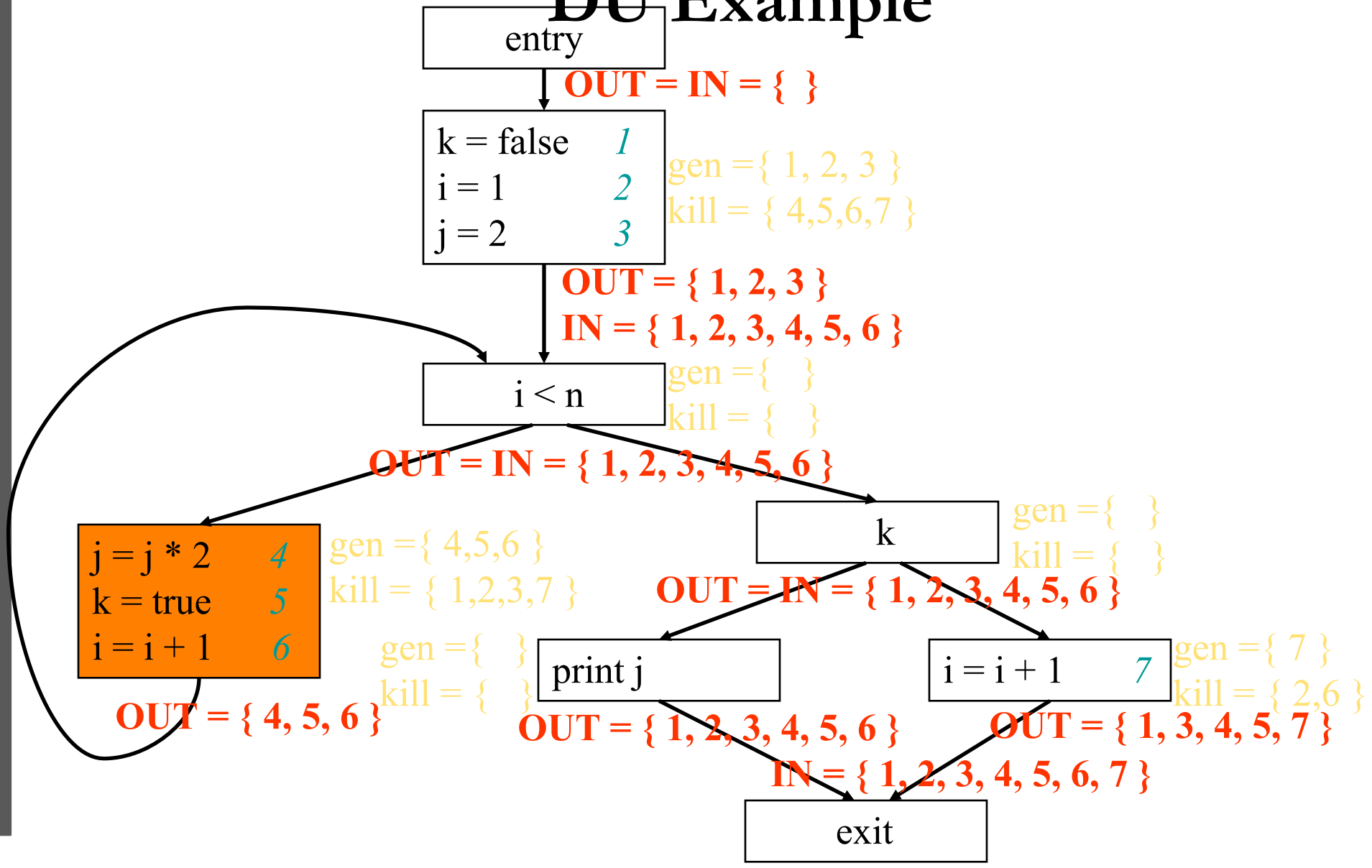
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

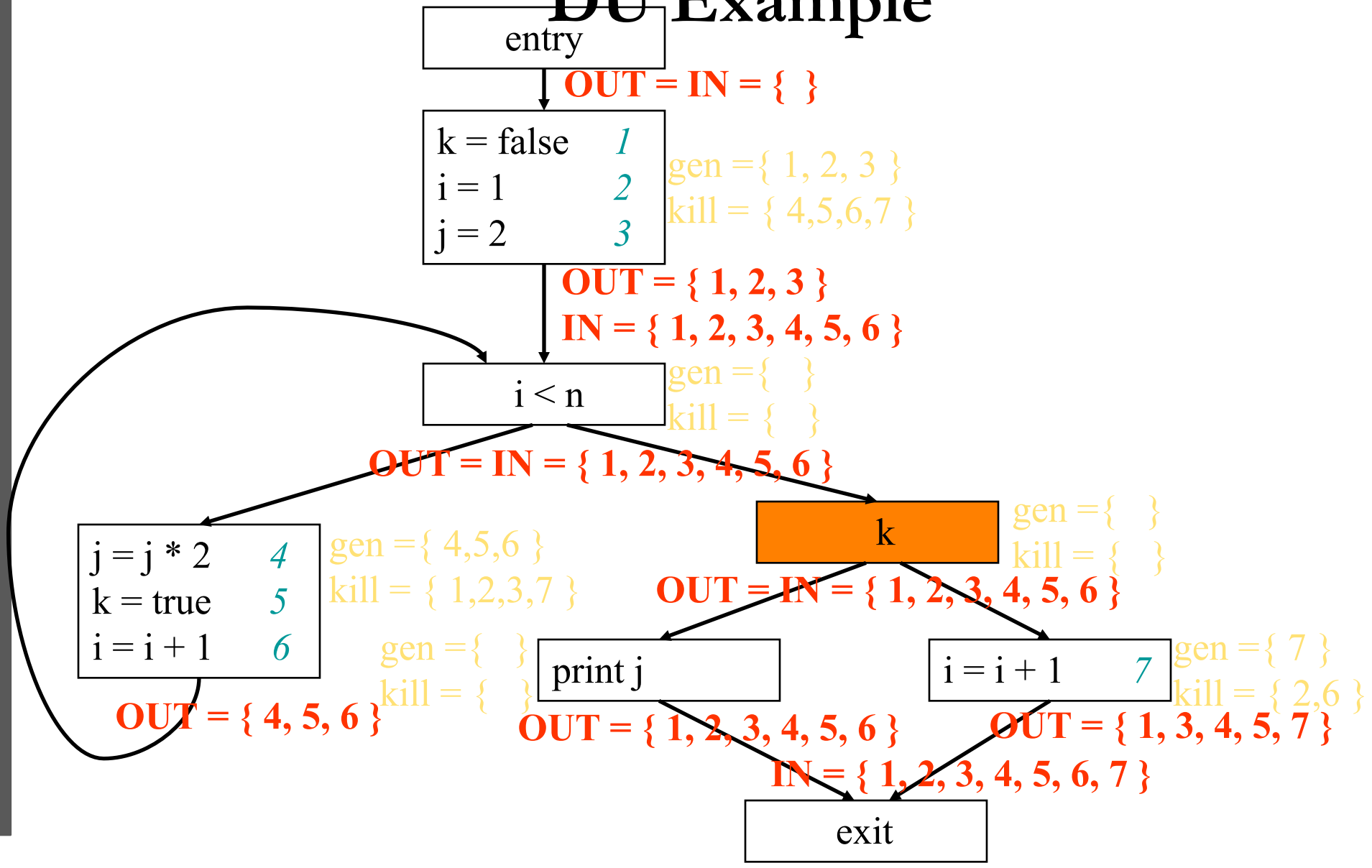
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

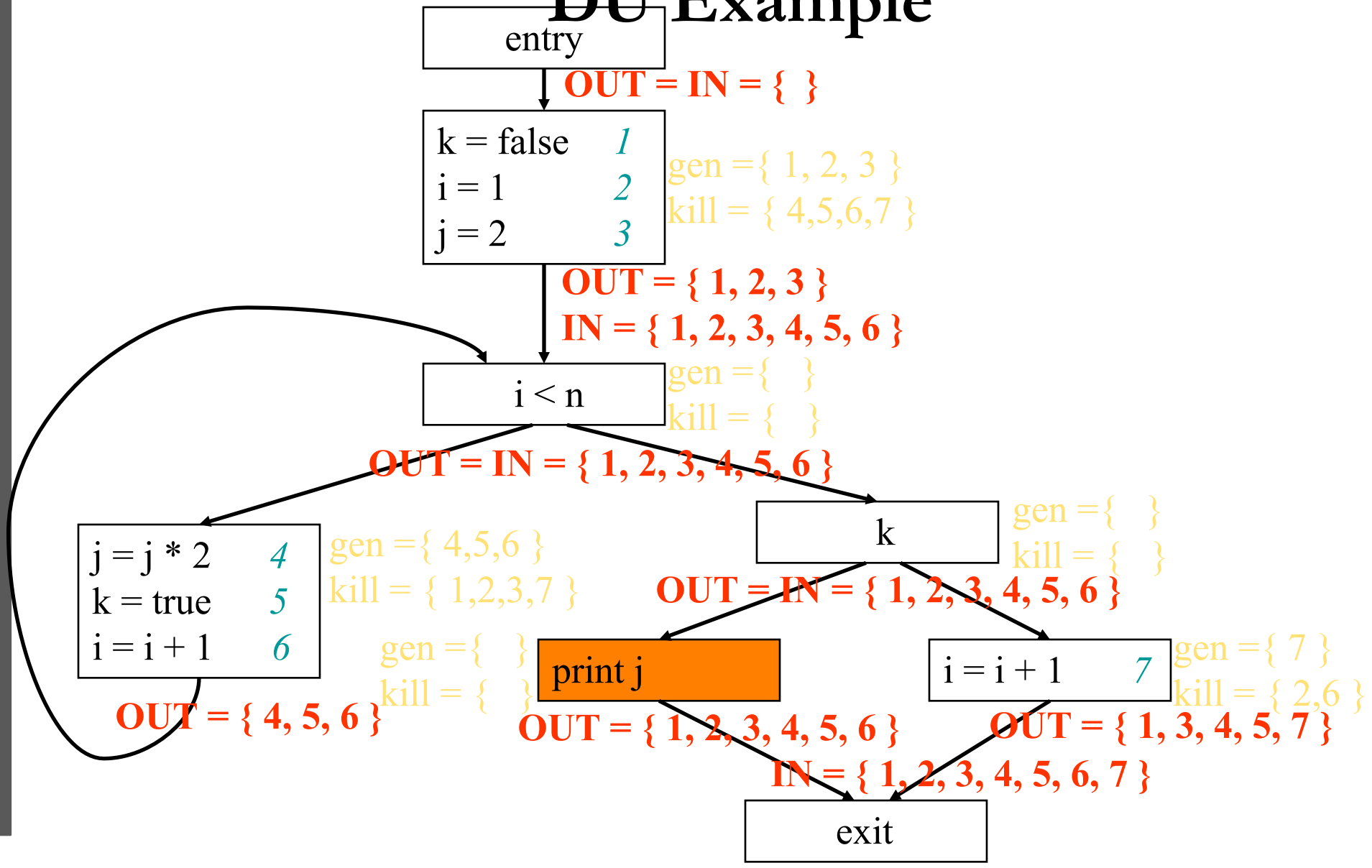
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

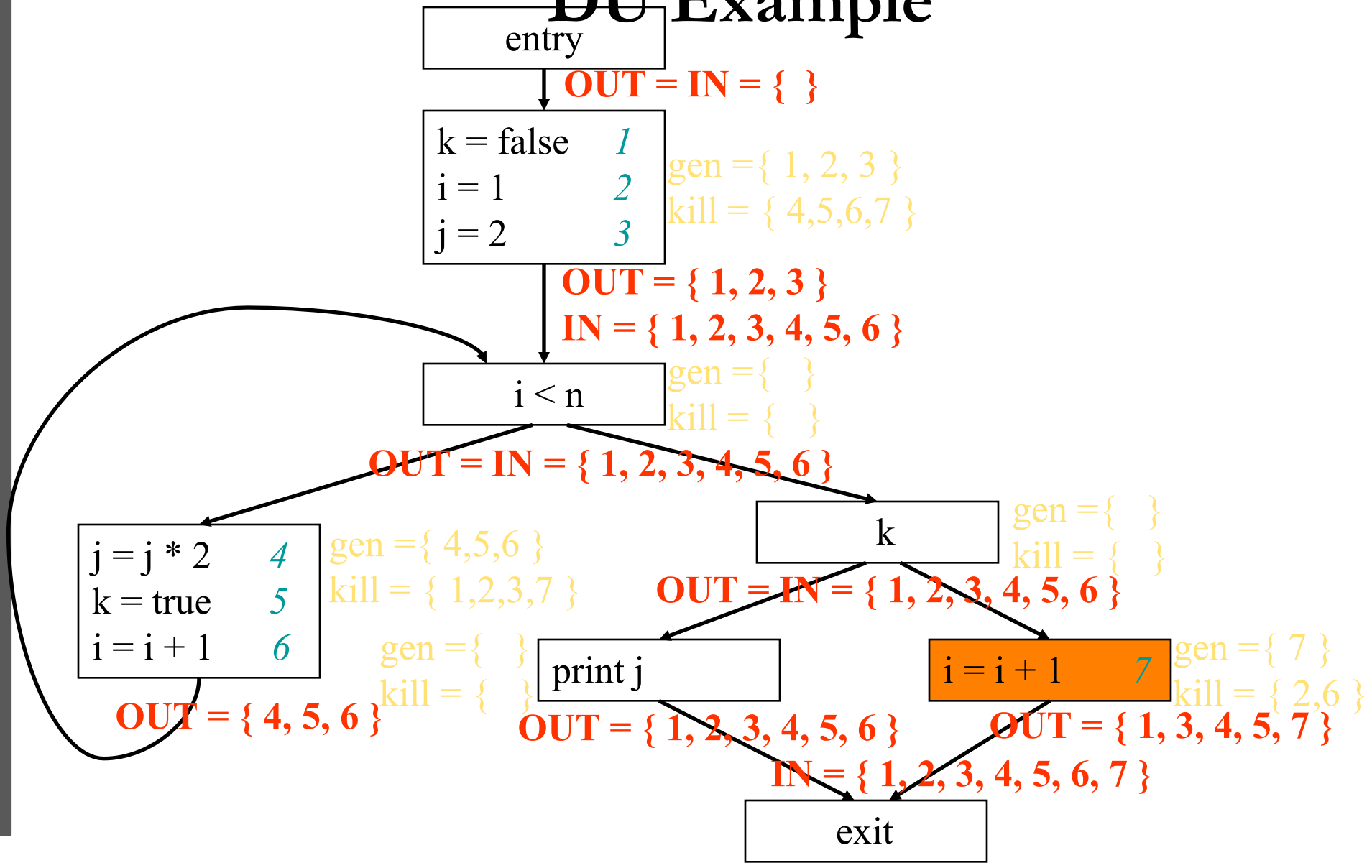
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

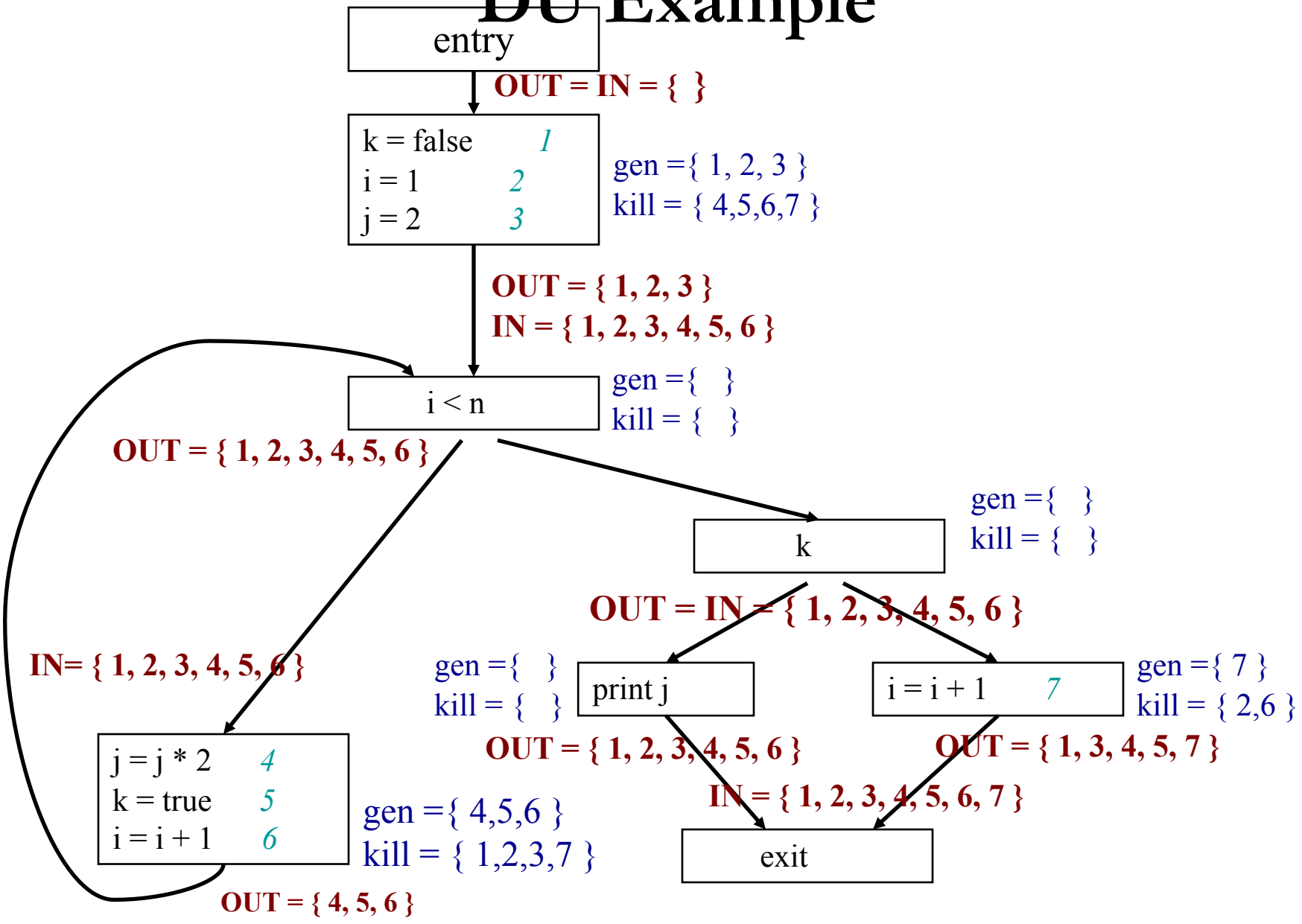
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

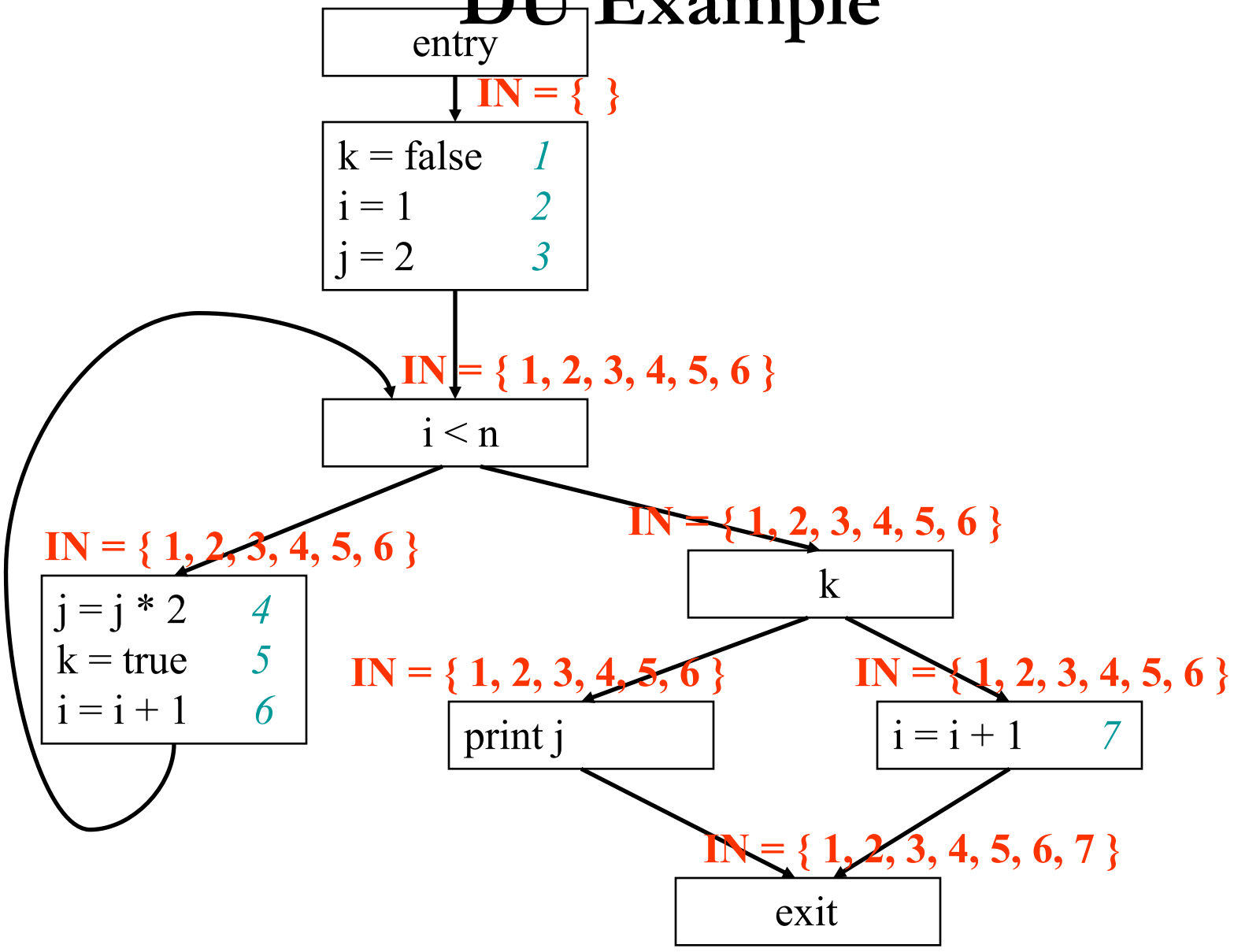
DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
OUT = { 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
print j
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

DU Example


entry
OUT = IN = { }
k = false 1
i = 1 2
j = 2 3
gen ={ 1, 2, 3 }
kill = { 4,5,6,7 }
OUT = { 1, 2, 3 }
IN = { 1, 2, 3, 4, 5, 6 }
i < n
gen ={ }
kill = { }
OUT = { 1, 2, 3, 4, 5, 6 }
k
gen ={ }
kill = { }
OUT = IN = { 1, 2, 3, 4, 5, 6 }
IN= { 1, 2, 3, 4, 5, 6 }
gen ={ }
kill = { }
print j
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { 1, 2, 3, 4, 5, 6 }
OUT = { 1, 3, 4, 5, 7 }
IN = { 1, 2, 3, 4, 5, 6, 7 }
j = j * 2 4
k = true 5
i = i + 1 6
gen ={ 4,5,6 }
kill = { 1,2,3,7 }
exit
OUT = { 4, 5, 6 }

DU Example


entry
IN = { }
k = false 1
i = 1 2
j = 2 3
IN = { 1, 2, 3, 4, 5, 6 }
i < n
IN = { 1, 2, 3, 4, 5, 6 }
j = j * 2 4
k = true 5
i = i + 1 6
IN = { 1, 2, 3, 4, 5, 6 }
k
IN = { 1, 2, 3, 4, 5, 6 }
print j
IN = { 1, 2, 3, 4, 5, 6 }
i = i + 1 7
IN = { 1, 2, 3, 4, 5, 6, 7 }
exit

DU Chains

- At each use of a variable, indicates all its possible definitions (and thus its points)
  - Very Useful Information
  - Used in Many Optimizations

- Information can be Incorporate in the Intermediate Representation
  - SSA From (next)

Outline

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

Static Single Assignment (SSA) Form

- Each definition has a unique variable name
  - Original name + a version number
- Each use refers to a unique name definition
- What about multiple possible definitions?
  - Add special merge nodes so that there can be only a single definition (ϕ functions)

Static Single Assignment (SSA) Form

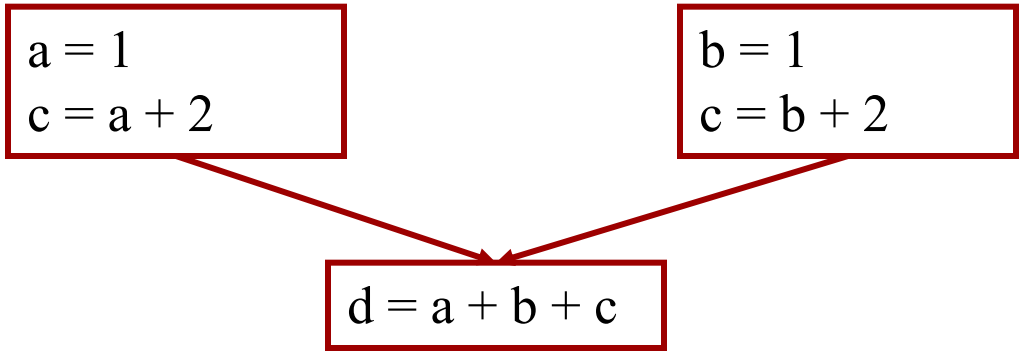
```
a = 1
b = a + 2
c = a + b
a = a + 1
d = a + b
```

Static Single Assignment (SSA) Form

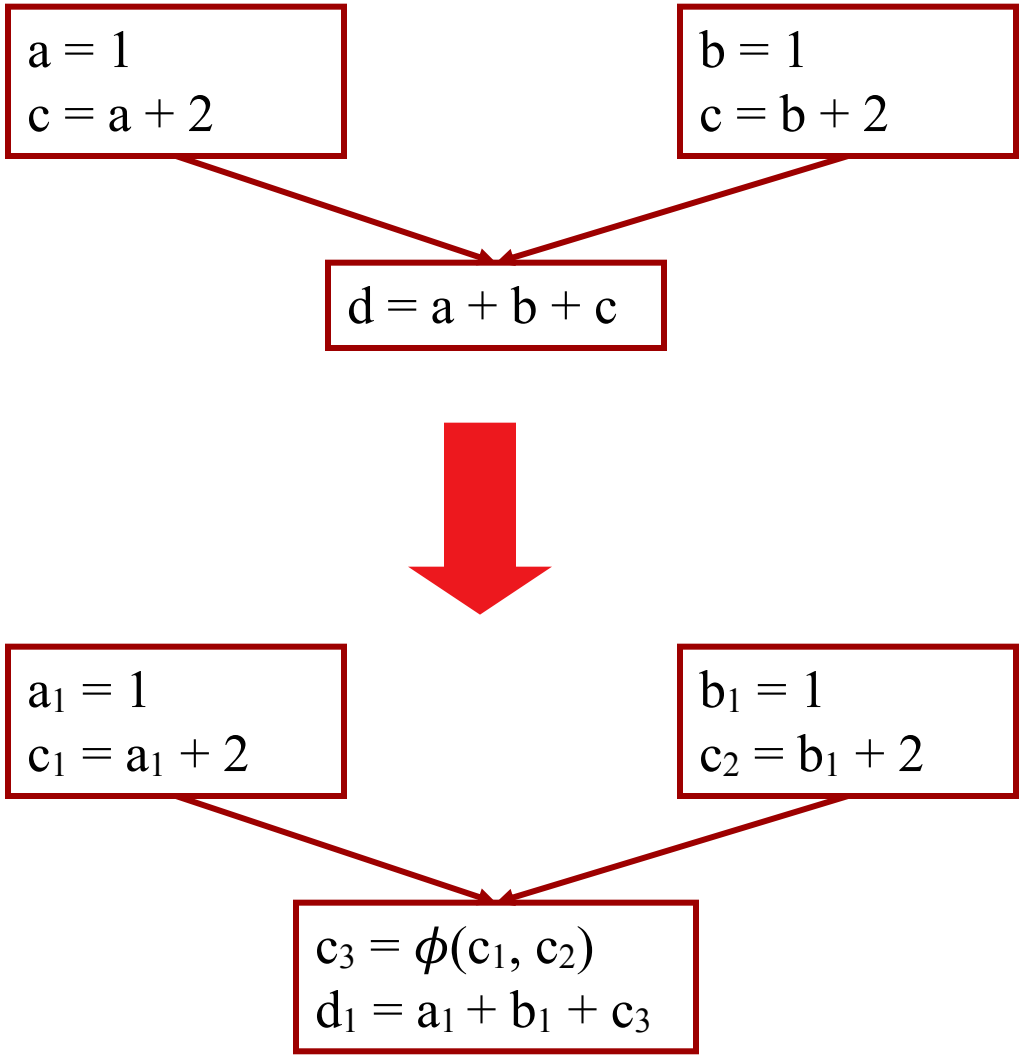
$$
\begin{aligned}
a &= 1 \\
b &= a + 2 \\
c &= a + b \\
a &= a + 1 \\
d &= a + b
\end{aligned}
$$

$$
\begin{aligned}
a_1 &= 1 \\
b_1 &= a_1 + 2 \\
c_1 &= a_1 + b_1 \\
a_2 &= a_1 + 1 \\
d_1 &= a_2 + b_1
\end{aligned}
$$

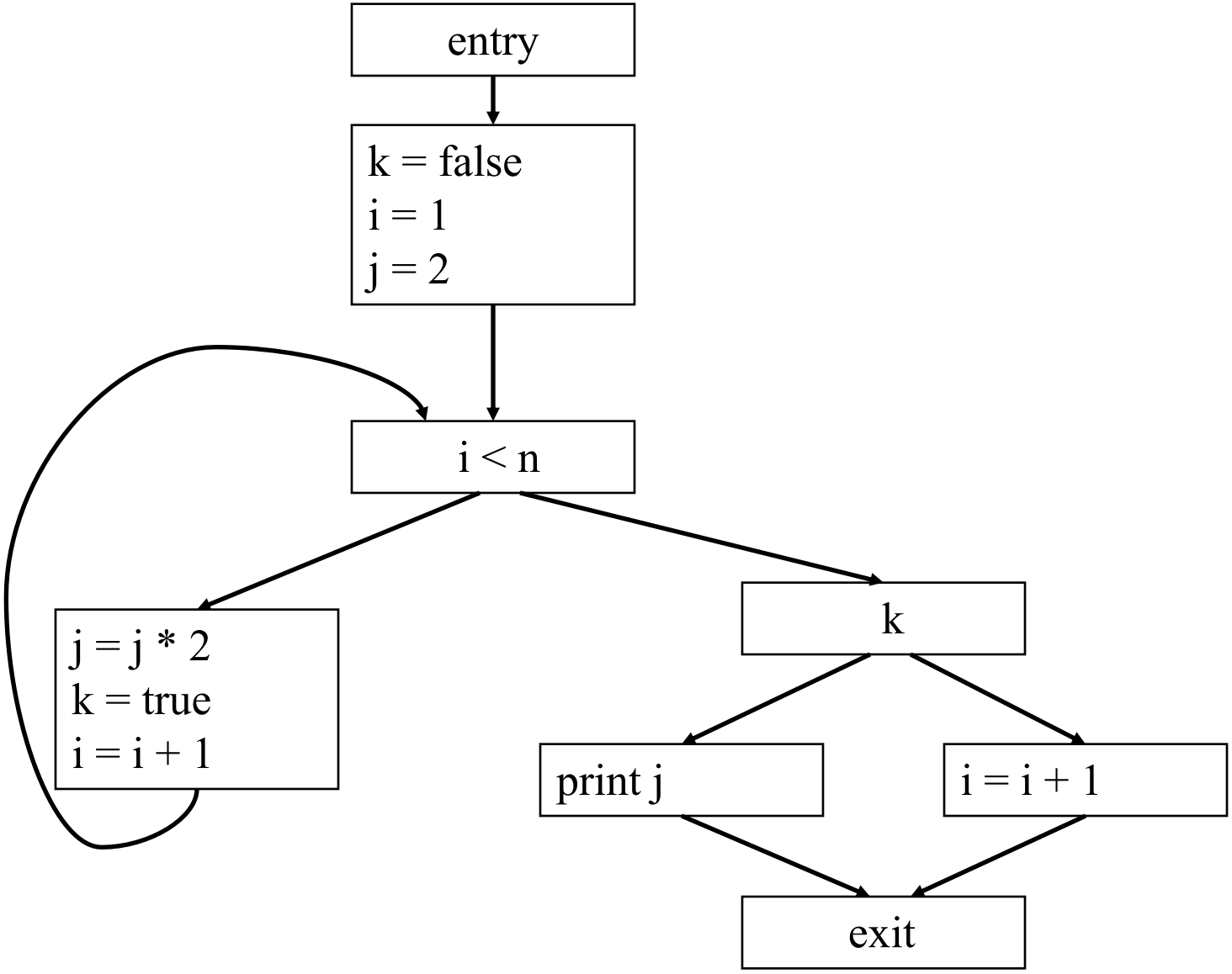
Static Single Assignment (SSA) Form



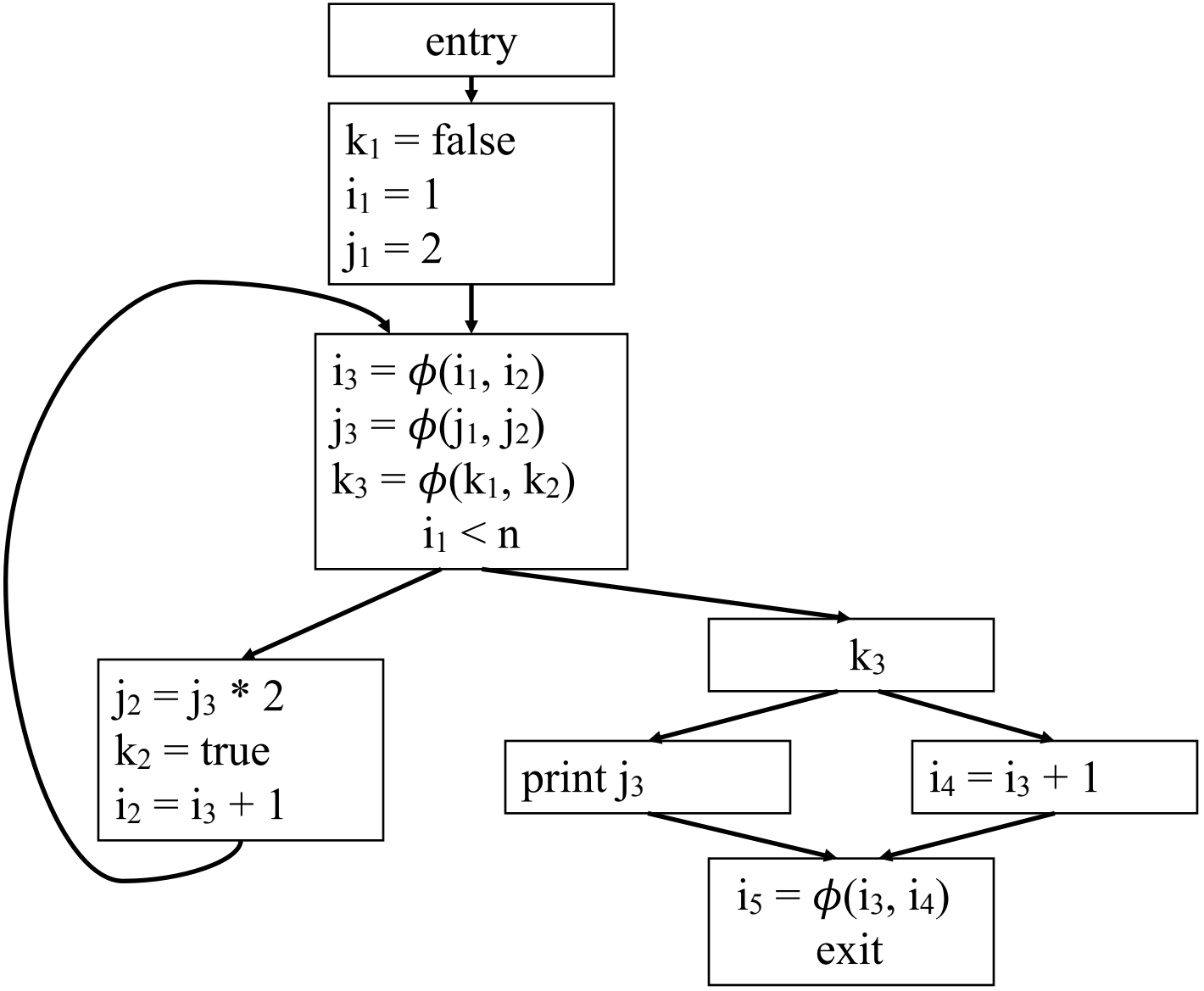
Static Single Assignment (SSA) Form



DU Example



DU Example


entry
k_1 = false
i_1 = 1
j_1 = 2
$i_3 = \phi(i_1, i_2)$
$j_3 = \phi(j_1, j_2)$
$k_3 = \phi(k_1, k_2)$
$i_1 < n$
$j_2 = j_3 * 2$
k_2 = true
$i_2 = i_3 + 1$
k_3
print j_3
$i_4 = i_3 + 1$
$i_5 = \phi(i_3, i_4)$
exit

Summary

- Overview of Control-Flow Analysis
- Available Expressions Data-Flow Analysis Problem
- Algorithm for Computing Available Expressions
- Practical Issues: Bit Sets
- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form