U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO
DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Compilers
Design and Implementation

Data-Flow Analysis

Iterative Data-Flow Formulation

Outline

- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

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

Iterative Data-Flow Analysis Problem

- Problem Independent
  - We need to Build the Control-Flow Graph
    - Defines predecessors and successors
  - Calculate Gen and Kill Sets for each Basic Block
  - Iterative Propagation of Information until Convergence
  - Propagation of Information within the Basic Block

- Runs on a Round-Robin Work-list Algorithm
  - Initializes abstract valued for each node n
  - Iterates until it reaches a fixed point

Iterative Data-Flow Analysis Problem

- Solving other Data-Flow Analysis Problem?
  - Reuse Iterative Framework
  - Adapt Initialization and Equations

- **Key Issues:**
  - Will the iterative algorithm converge (terminate)?
  - How fast will it converge?
  - When is iterative solution correct?
  - How precise is the solution found by the iterative approach?

- **Observation:** Equations define "flow" of information
  - Forwards: Along Control-Flow
  - Backwards: Against control-Flow

Data-Flow Analysis : The Basics

- Semi-Lattice/Lattice
  - Abstract quantities V over which the analysis will operate
  - Two operations meet ($\bigwedge$) and join ($\bigvee$) over values of V
  - A top value ($\top$) and a bottom value ($\bot$)
  - Example: Sets of Available Expressions and Intersection
- Transfer Functions
  - How each instruction (statement) and control-flow construct affects the abstract quantities V
    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"...*

Meet 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 phs 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 $\top$
$\forall a \in L, \top \wedge a = a$

Meet Semilattices and Partial Order

- Partial Order for a Meet Semilattice (V, $\wedge$)
  - Defined by the meet operator ($\wedge$) with Properties:
    - $x \leq x$ $(reflexive)$
    - If $x \leq y$ and $y \leq x$ then $x = y$ $(antisymmetric)$
    - If $x \leq y$ and $y \leq z$ then $x \leq z$ $(transitive)$
  - The pair (V, $\leq$) is a poset.
  - The relation $<$ is defined as $x < y$ iff $(x \leq y)$ and $x \neq z$
- Partial Order for a Semilattice (V, $\wedge$)
  - Defines the operator $\leq$ such that $x \leq y$ iff $(x \wedge y) = x$
  - Because $\wedge$ is idempotent, commutative and associative, order $\leq$ is:
    - Reflexive
    - Antisymmetric
    - Transitive

Meet Semilattices and Partial Order

- Example:
  - Meet operators: Set Union (∪) and Set Intersection (∩)
  - Both idempotent, commutative and associative
  - Set Union (∪):
    - Top element $\top = \emptyset$
    - Bottom element $\bot = U$
  - Set Intersection (∩):
    - Top element $\top = U$
    - Bottom element $\bot = \emptyset$

Meet Semilattices and Lower Bounds

- For 2 elements x and y, the greatest lower bound ($g \in V$) is:
  - $g \leq x$
  - $g \leq y$ and
  - If z is any element such that $z \leq x$ and $z \leq y$, then $z \leq g$

- The meet $g = x \wedge y$ is the (single) greatest lower bound of x and y

Join Semilattices and Upper Bounds

- Partial Order for a Meet Semilattice (V, $\bigvee$)
  - Defined by the join operator ($\bigvee$) with Properties:
    - $x \leq x$ $(reflexive)$
    - If $x \leq y$ and $y \leq x$ then $x = y$ $(antisymmetric)$
    - If $x \leq y$ and $y \leq z$ then $x \leq z$ $(transitive)$
  - The pair (V, $\leq$) is a poset.
  - The relation < is defined as $x < y$ iff $(x \leq y)$and $x \neq z$
- For 2 elements x and y, the lowest upper bound ($b \in V$) is:
  - $x \leq b$,
  - $y \leq b$ and,
  - If z is any element such that $x \leq z$ and $y \leq z$, then $b \leq z$
- The join $b = x \bigvee y$ is the (single) lowest upper bound of x and y

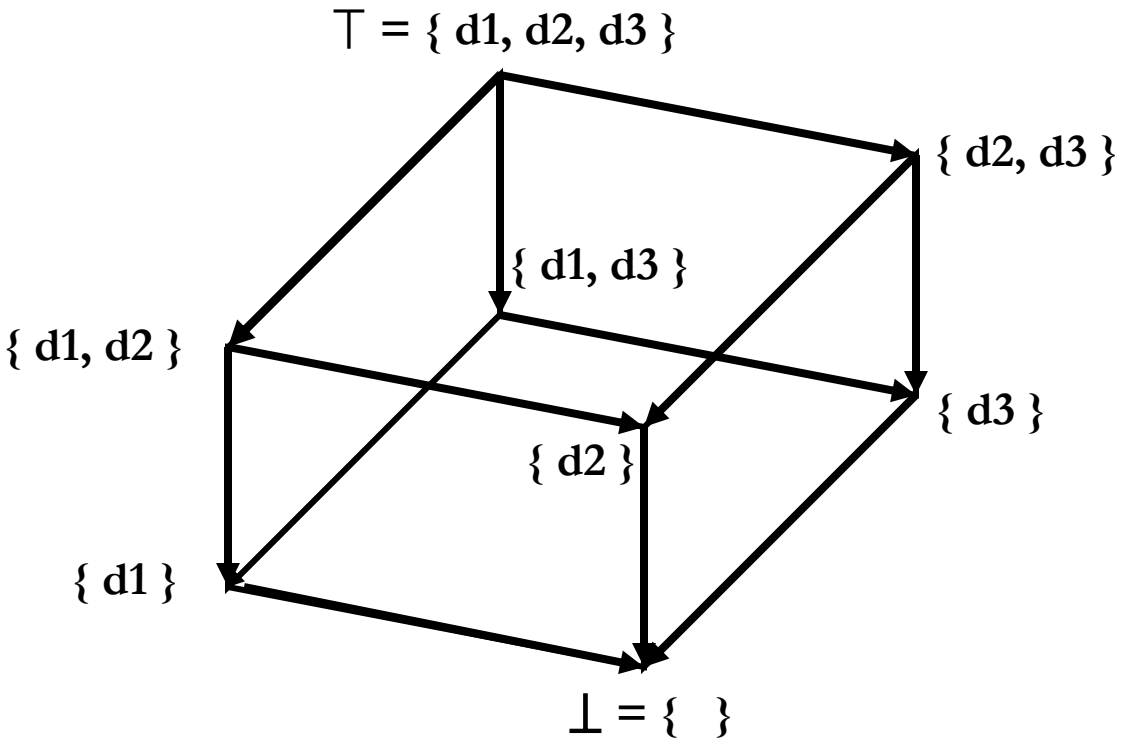
Join Semilattices and Upper Bounds

- For 2 elements x and y, the lowest upper bound ($b \in V$) is:
  - $x \leq b$,
  - $y \leq b$ and,
  - If z is any element such that $x \leq z$ and $y \leq z$, then $b \leq z$

- The join $b = x \bigvee y$ is the (single) lowest upper bound of x and y

Lattice and Lattice Diagrams

- A (True) Lattice L consists of
  - A Set of Values V
  - Two operations meet (⋀) and join (⋁)
  - A top value (⊤) and a bottom value (⊥)



Meet and Join Operators

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

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

- Meet and Join are Commutative:

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

- Meet and Join are Associative:

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

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

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

Meet and Join Operators

- Meet Operation: Greatest Lower Bound - GLB({x,y})
  - Example: Set Intersection
  - Follow the lines in L downwards from the two elements in the lattice until they meet at a single unique element of lattice.

- Join Operation: Lowest Upper Bound - LUB({x,y})
  - Example: Set Union
  - There is a unique element in the lattice from where there is a downwards path (with no shared segment) to both elements (same as a reverse path to a lowest common ancestor in the lattice).

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 !

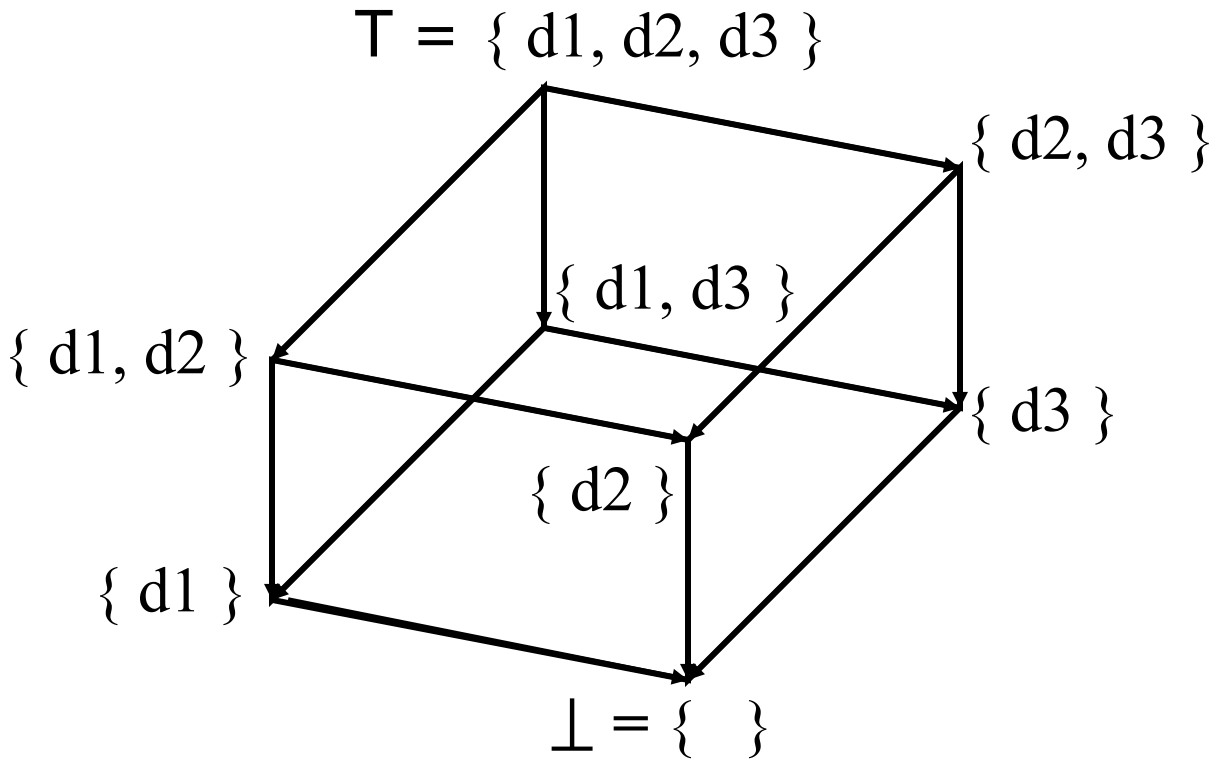
Reaching Definitions Problem

Def: *A definition* ***d*** *of variable* ***v*** *reaches instruction* ***i*** *iff instruction* ***i*** *reads* ***v*** *and there exists no path from* ***d*** *to* ***i*** *that (re)defines* ***v****.*

- Data-Flow (Forward) Problem:
  - Annotates program point in CFG with Reaches(n)
  - Initialization: Reaches(n) = ∅, ∀*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*

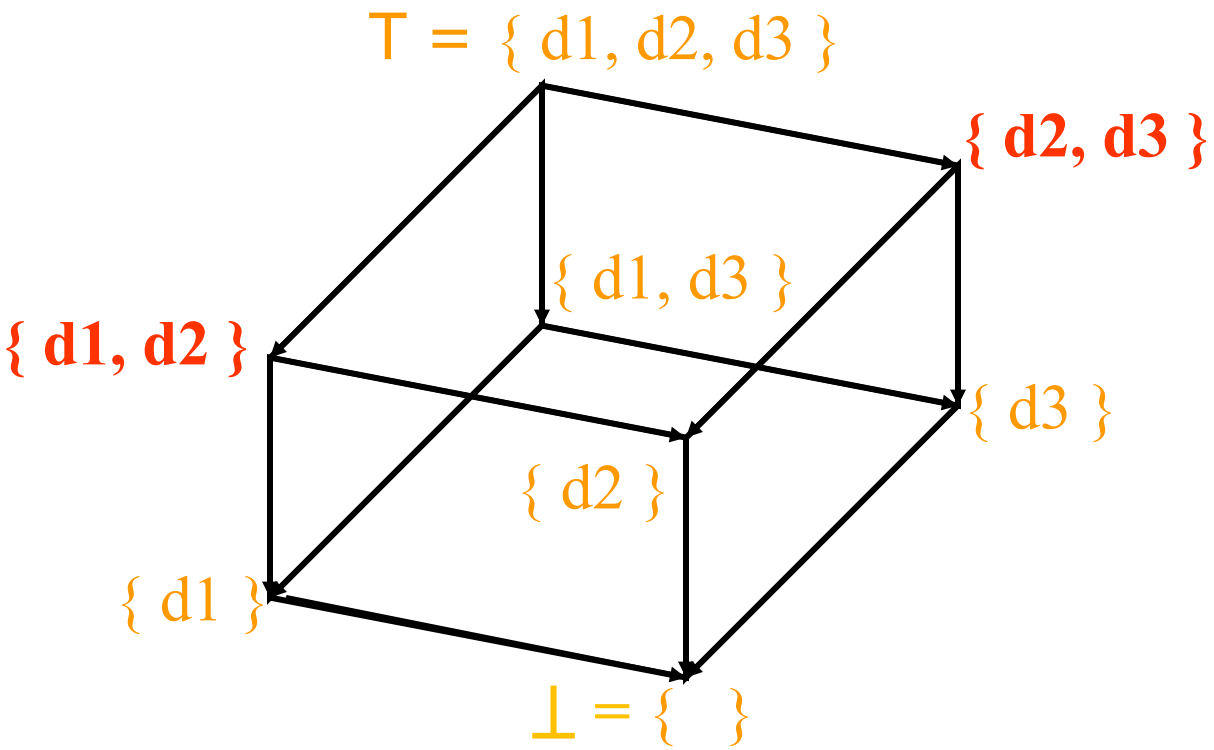
Reaching Definitions Lattice

- Lattice for the Reaching Definitions Problem
  - Where there are only 3 definitions: {d1, d2, d3}
- Need to Compute:
  - Which subset of these definitions reaches each program point



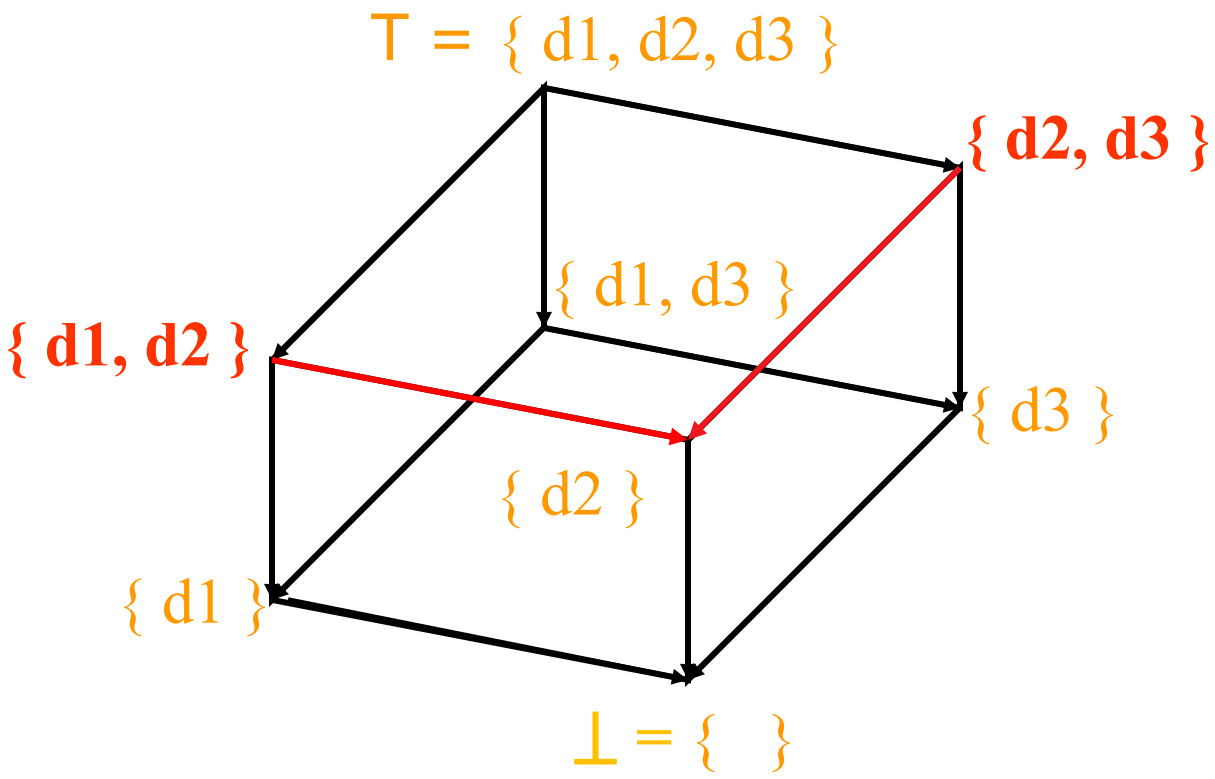
Meet and Join Operators

$\{ d1, d2 \} \wedge \{ d2, d3 \} = glb(\{ d1, d2 \}, \{ d2, d3 \}) = ???$



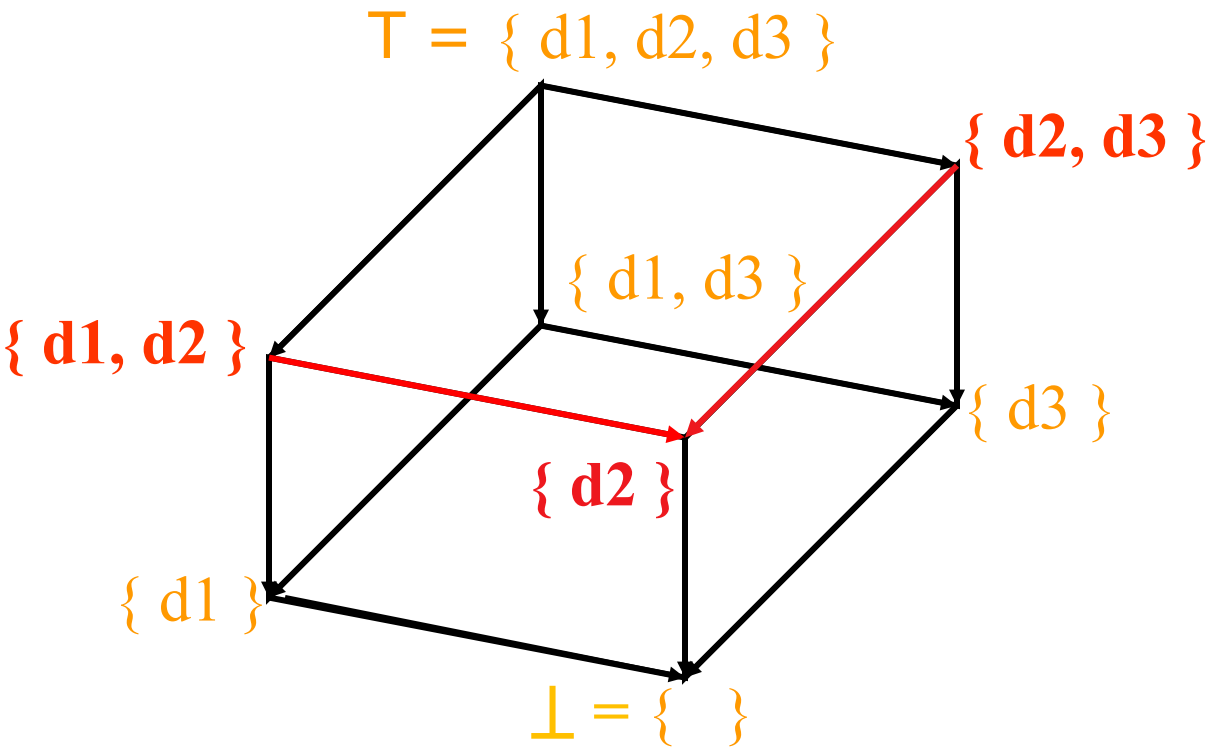
Meet and Join Operators

$\{ d1, d2 \} \wedge \{ d2, d3 \} = glb(\{ d1, d2 \}, \{ d2, d3 \}) = ???$



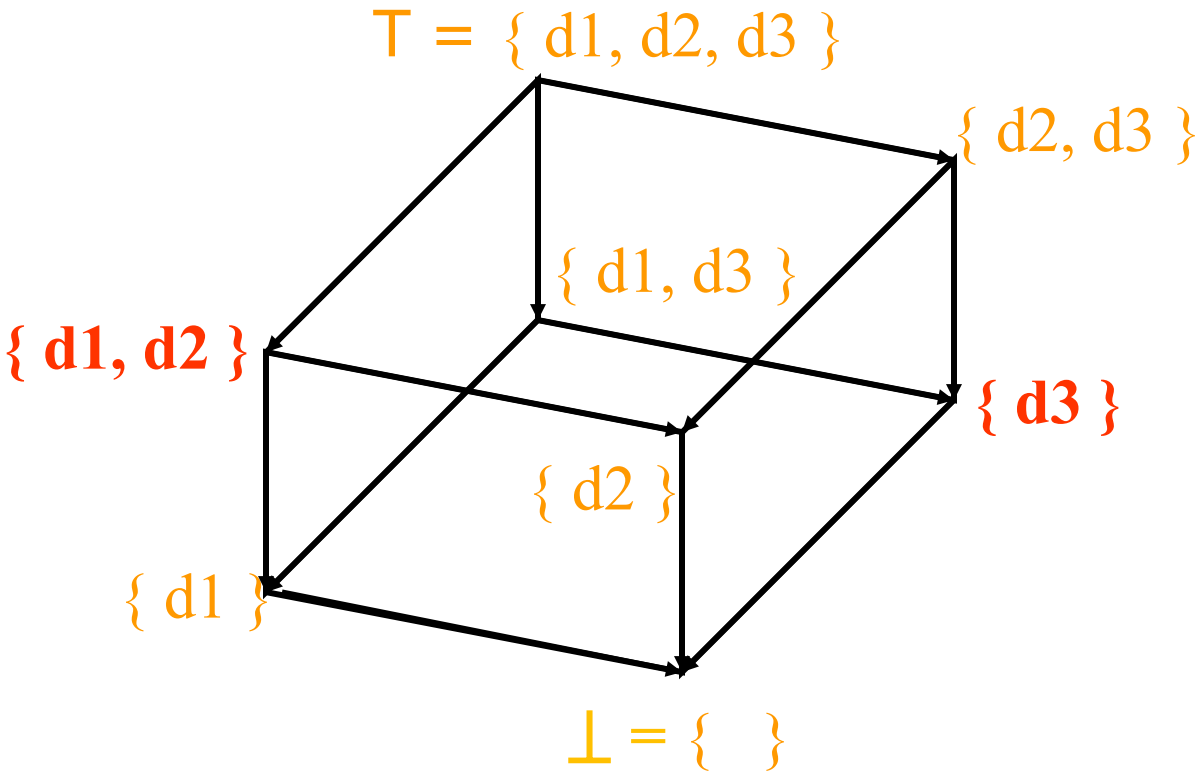
Meet and Join Operators

$$\{ d1, d2 \} \wedge \{ d2, d3 \} = glb(\{ d1, d2 \}, \{ d2, d3 \}) = \{d2\}$$



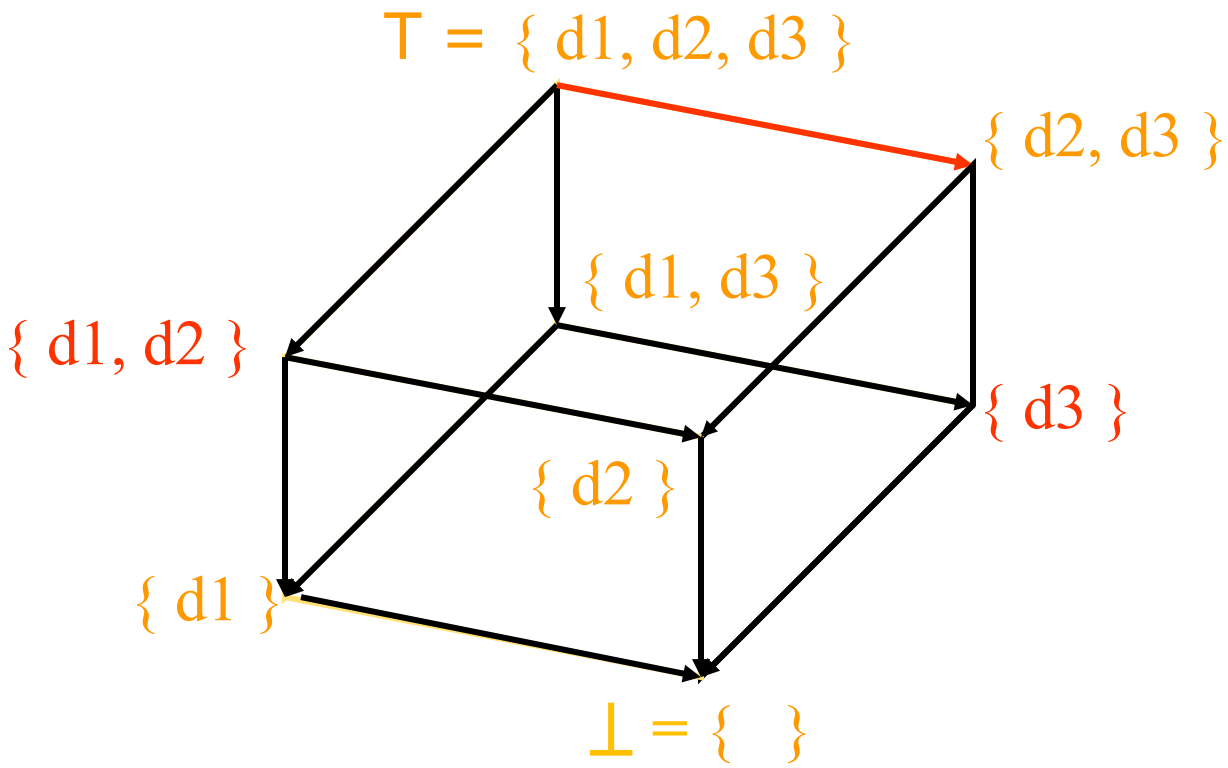
Meet and Join Operators

$\{d1, d2\} \vee \{d3\} = lub(\{d1, d2\},\{d3\}) = ???$



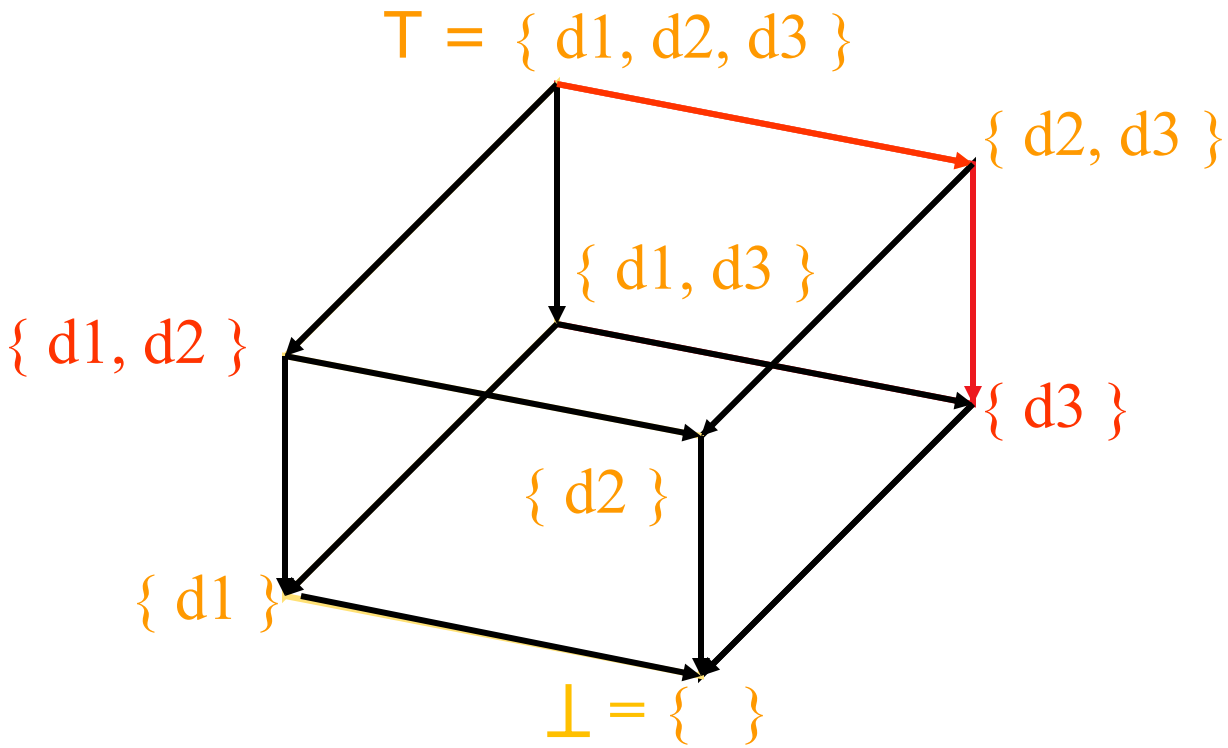
Meet and Join Operators

$\{d1, d2\} \vee \{d3\} = lub(\{d1, d2\},\{d3\}) = ???$



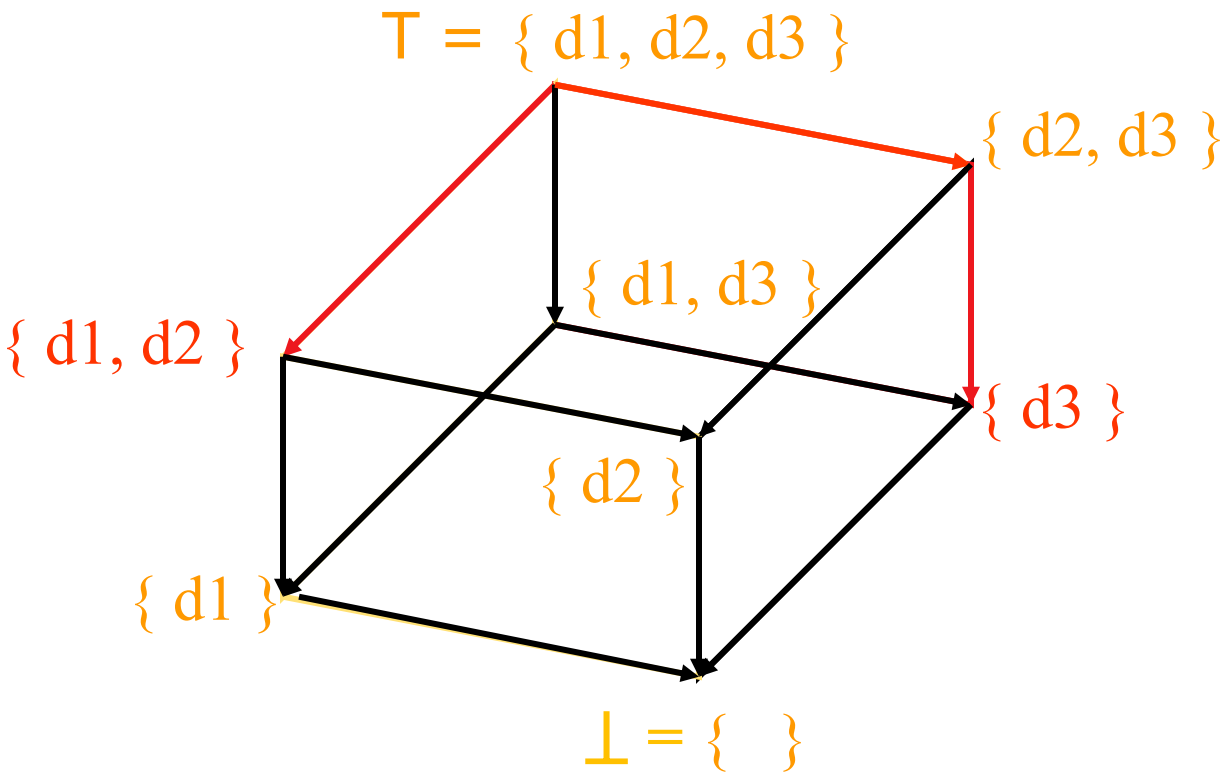
Meet and Join Operators

$\{d1, d2\} \vee \{d3\} = lub(\{d1, d2\},\{d3\}) = ???$



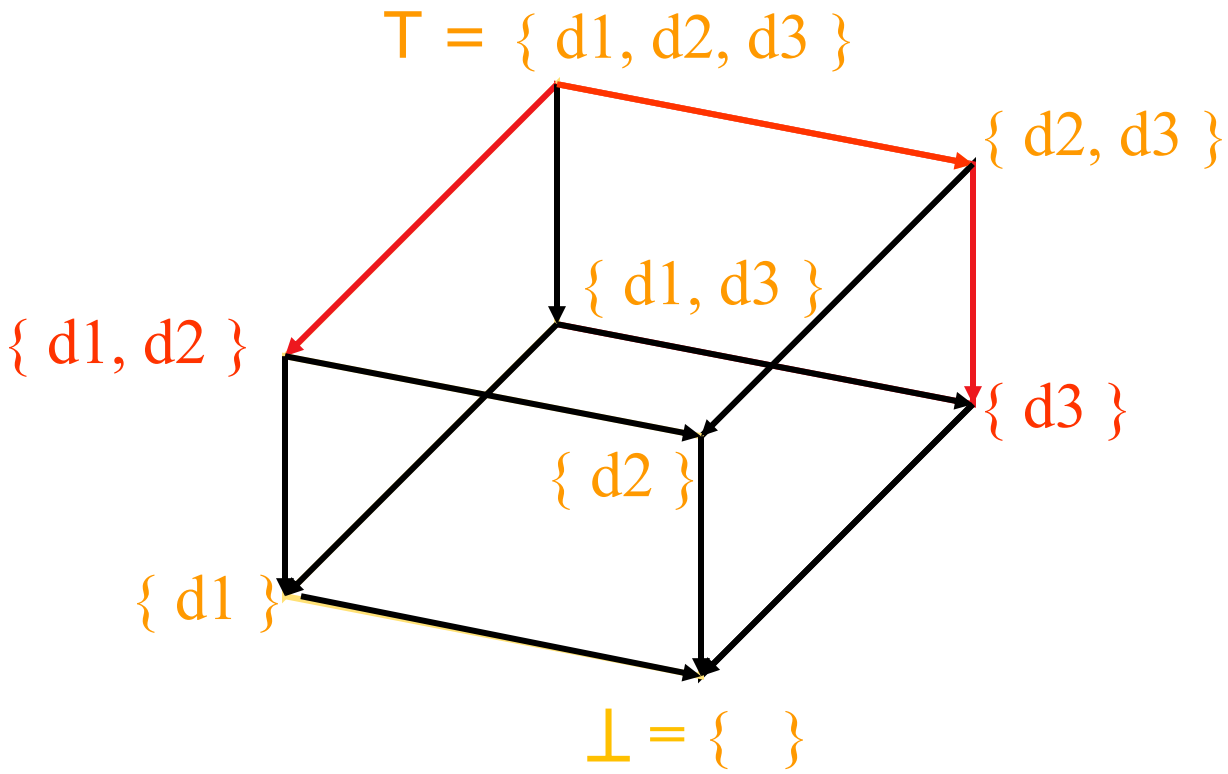
Meet and Join Operators

$\{d1, d2\} \vee \{d3\} = lub(\{d1, d2\},\{d3\}) = ???$



Meet and Join Operators

$$\{d1, d2\} \vee \{d3\} = lub(\{d1, d2\},\{d3\}) = \{d1,d2,d3\}$$



Transfer Functions

- Transfer functions: F: V→V properties:
  - F has an identity function I, such that I(x) for all x ∈ V
  - F is closed under composition, *i,e,* for any two functions f and g ∈ F, the function h(x) = g(f(x)) ∈ F

- Example: Reaching Definitions
  - Identify Function: gen = ∅ and kill = ∅
  - Closure under composition:
    - If $f_1(x) = Gen_1 \cup (x - Kill_1)$
    - And $f_2(x) = Gen_2 \cup (x - Kill_2)$
    - Then, $f_2(f_1(x)) = Gen_2 \cup ((Gen_1 \cup (x - Kill_1)) - Kill_2)$ which is equivalent to: $f_3(x) = Gen_3 \cup (x - Kill_3)$ with $Gen_3 = Gen_2 \cup (Gen_1 - Kill_2)$ and $Kill_3 = Kill_1 \cup Kill_2$

Monotonicity

- An Iterative Data-Flow framework is monotonic iff:
  - Given two members of x and y ∈ V, if we apply any transfer function f in F, and $x \leq y$, then $f(x) \leq f(y)$

    $\forall$ x and y ∈ V and f ∈ F, $x \leq y \Rightarrow f(x) \leq f(y)$
  - Equivalently:

    $\forall$ x and y ∈ V and f ∈ F, $f(x \wedge y) \leq f(x) \wedge f(y)$

- What are the implications for the Iterative Algorithm?

Distributivity

- Sometimes the iterative framework is even *distributive*

$$\forall\ x \text{ and } y \in V \text{ and } f \in F,\ f(x \wedge y) = f(x) \wedge f(y)$$

- Distributivity ⇒ Monotonicity (converse not true)
- What are the implications for the Iterative Algorithm?

Iterative Algorithm for Data-Flow

INPUT: A data-flow framework with the following components:

1. A control-flow graph (CFG), with specially labeled *Entry* and *Exit* nodes,
2. A direction of the data-flow D,
3. A set of values V,
4. A meet operator $\wedge$,
5. A set of functions F, where $f_B \in F$ is the transfer function for block B, and
6. A constant value v_{entry} or $v_{exit} \in V$, representing the boundary condition for forward and backward frameworks, respectively.

OUTPUT: Values in V for IN[B] and OUT[B] for each block B in the data-flow graph.

Algorithms for Iterative Data-Flow

```
1)  OUT[ENTRY] = v_ENTRY;
2)  for (each basic block B other than ENTRY) OUT[B] = ⊤;
3)  while (changes to any OUT occur)
4)          for (each basic block B other than ENTRY) {
5)                  IN[B] = ⋀_{P a predecessor of B} OUT[P];
6)                  OUT[B] = f_B(IN[B]);
            }
```

(a) Iterative algorithm for a forward data-flow problem.

```
1)  IN[EXIT] = v_EXIT;
2)  for (each basic block B other than EXIT) IN[B] = ⊤;
3)  while (changes to any IN occur)
4)          for (each basic block B other than EXIT) {
5)                  OUT[B] = ⋀_{S a successor of B} IN[S];
6)                  IN[B] = f_B(OUT[B]);
            }
```

(b) Iterative algorithm for a backward data-flow problem.

Critical Iterative Algorithm Properties

1. If the semilattice of the framework is monotone and of finite height, then the algorithm is guaranteed to converge (**termination**)
2. If Algorithm converges, the result is a solution to the data-flow equations (**correctness**)
3. If the framework is monotone, then the solution found is the maximum fixed-point (MFP) of the data-flow equations.
4. A MFP is a solution with the property that in any other solution, the values of IN[B] and OUT[B] are $\leq$ the corresponding values of the MFP (**bestness**)

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

Meaning of a Data-Flow Solution

- Iterative Algorithm computes Maximum Fixed-Point (MFP)
  - It is a "feasible" solution
  - What does it means from a program-semantics view point?
  - Can we do better?
- Ideal Solution
- Meet over all Paths (MOP) Solution

Meaning of a Data-Flow Solution

- **Ideal Solution:**
  - Solution for every block B is found by tracing and applying the transfer function of all possible execution paths leading from the program entry to the beginning of B.

$$P = \text{ENTRY} \rightarrow B_1 \rightarrow B_2 \rightarrow \ldots \rightarrow B_{k-1} \rightarrow B_k$$

$$\text{IDEAL}[B] = \bigwedge_{P, \text{ a possible path from ENTRY to } B} f_P(v_{\text{ENTRY}}).$$

  - A path is "possible" only if there is some computation of the program that follows exactly that path.
  - The ideal solution would then compute the data-flow value at the end of each possible path and apply the meet operator to these values to find their greatest lower bound.
  - Then no execution of the program can produce a smaller value for that program point.
  - The bound is tight; there is no greater data-flow value that is a glb for the value computed along every possible path to B in the flow graph.

Meaning of a Data-Flow Solution

- **Ideal Solution Claims:**
  - Any answer that is greater than IDEAL is incorrect.
  - Any value smaller than or equal to the ideal is conservative, i.e., ***safe***.

Meaning of a Data-Flow Solution

- **Meet Over All Paths Solution:**
  - We assume that every path in the flow graph can be taken.
  - We define the meet-over-paths solution for B to be

$$\text{MOP}[B] = \bigwedge_{P, \text{ a path from ENTRY to } B} f_P(v_{\text{ENTRY}})$$

  - The paths considered in the MOP solution are a superset of all the paths that are possibly executed.
  - Thus, the MOP solution meets together not only the data-flow values of all the executable paths, but also additional values associated with the paths that cannot possibly be executed.
  - Taking the meet of the ideal solution plus additional terms cannot create a solution larger than the ideal.
  - Thus, for all B we have MOP[B] ≤ IDEAL[B], and we will simply say that MOP ≤ IDEAL.

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

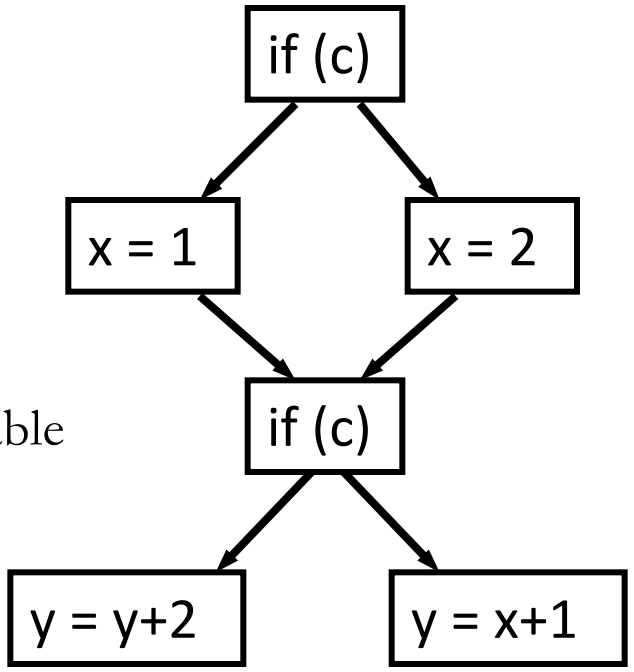
- **Critical Fact:** If transfer functions are distributive, then the solution to the dataflow equations is identical to the Meet-Over-All-Paths (MOP) solution:

$$MFP = MOP$$

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

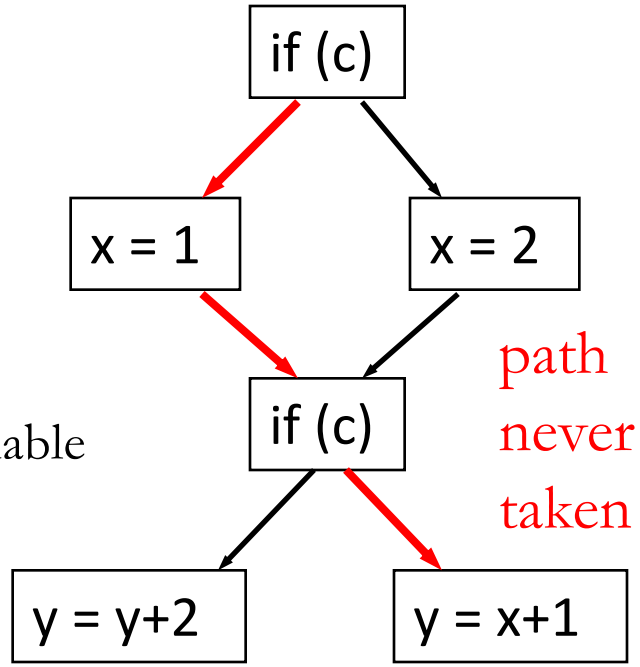
Can We Do Better Than MOP?

- 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



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



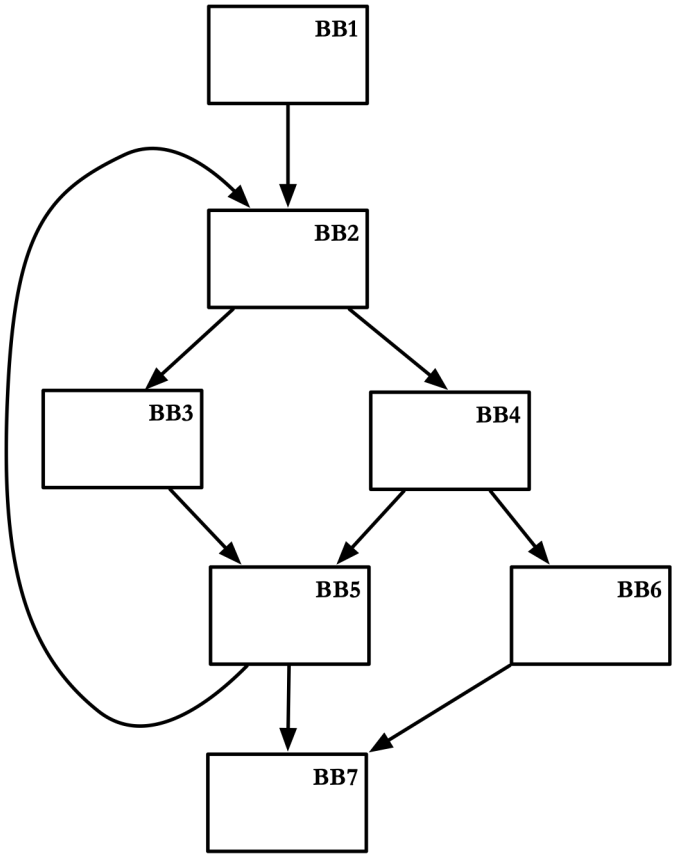
Limitations of Data-Flow Analysis

- 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
- Bottom-Line:
  - For scalar values, we can quickly solve simple problems

Speed

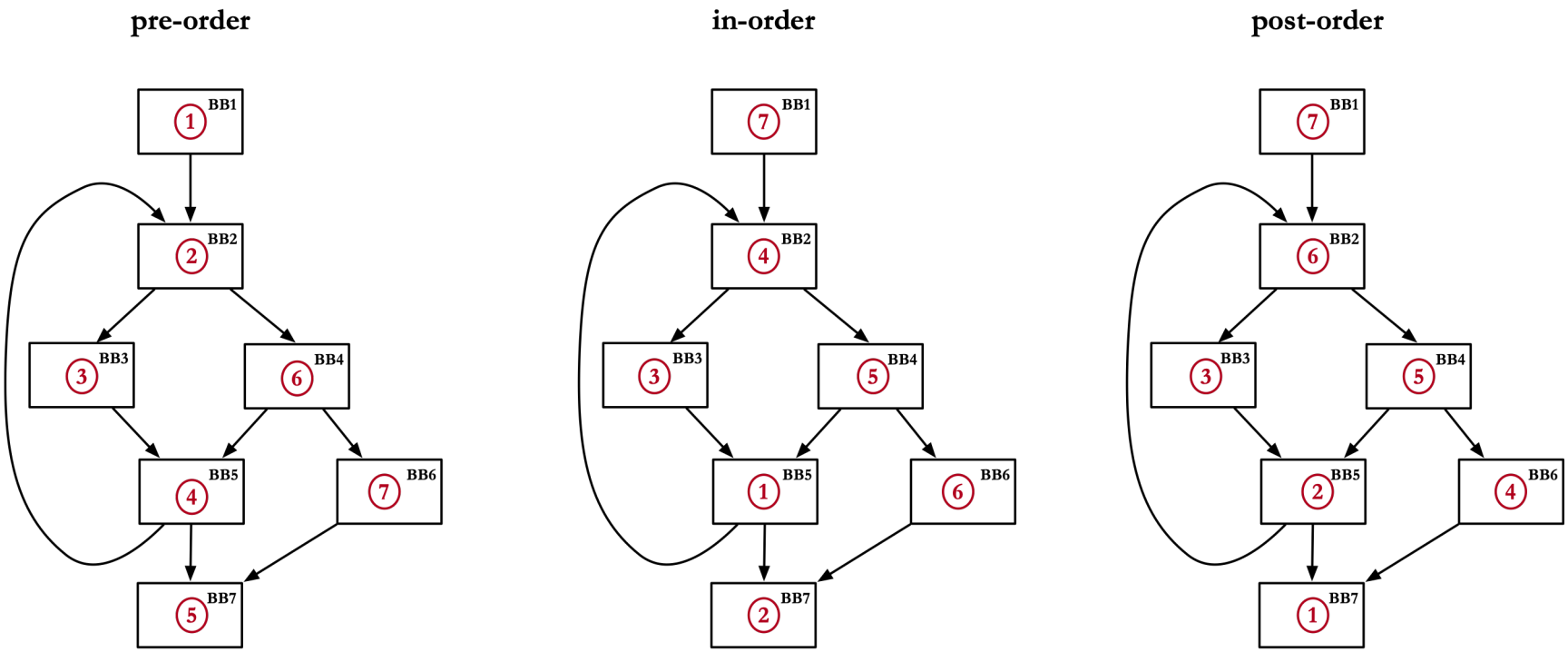
- If a Data-Flow Framework meets these admissibility conditions then is has a unique fixed-point solution
  - The iterative algorithm finds the (best) answer
  - The solution does not depend on the order of computation
  - Algorithm can choose an order that converges quickly
- Intuition:
  - Choose an 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
    - Easily Done with "visit" flag.

Order and Speed



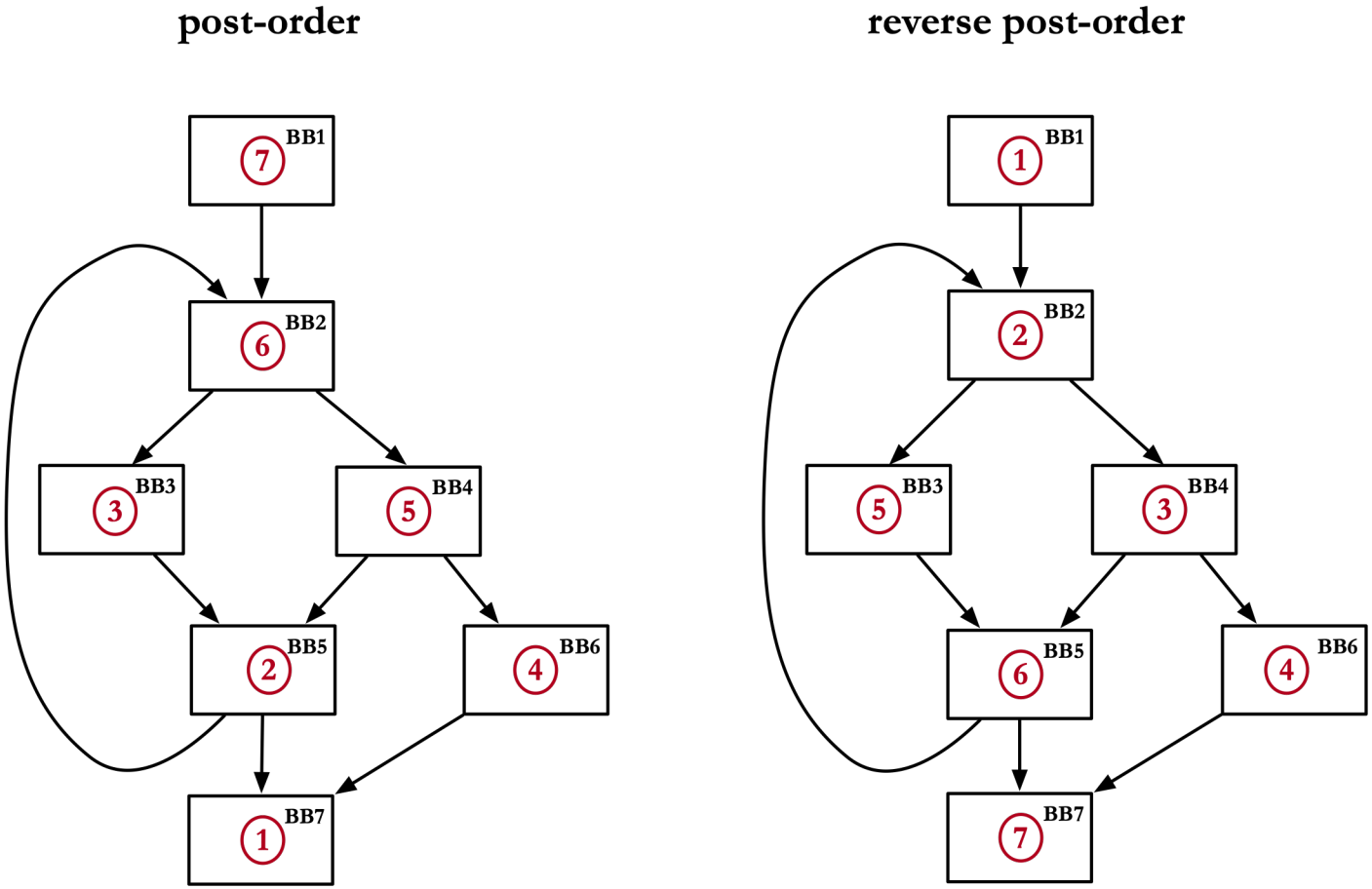
- Which Order for Node's Computation is Best?
  - Forward Problems vs Backwards Problems

Order and Speed



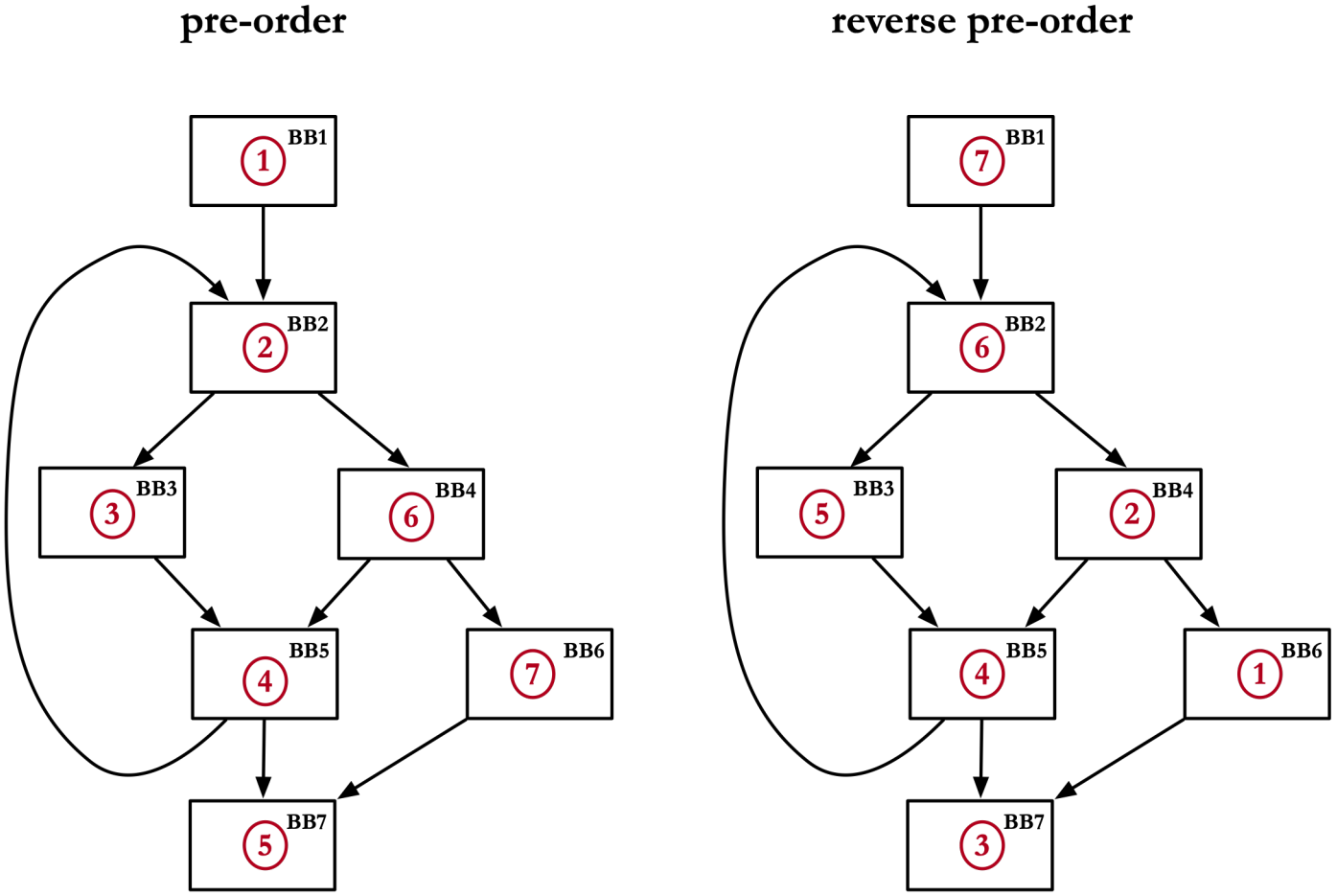
- Which Order for Node's Computation is Best?
  - Forward Problems vs Backwards Problems

Order and Speed for Forward Problems



- Reverse Post-Order (RPO) ensures:
  - For every node, all its predecessors are computed beforehand
  - Data Still Flow Forwards

Order and Speed for Backward Problems



- Reverse Pre-Order ensures:
  - For every node, all its successors are computed beforehand
  - Data Still Flow Backwards

Order and Speed

- Reverse Post-Order (Reverse Pre-Order):
  - Easily Computed order
  - Increases propagation per pass
- Round-Robin Iterative Algorithm
  - Visit All Nodes in a Consistent Order (Rev. Post or Pre)
  - Repeat Until Sets Stop Changing
- The Analysis Runs In (*Effectively*) Linear Time

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 (in general) Intractable
- Compilers use dataflow analysis and MFP

Outline

- 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.

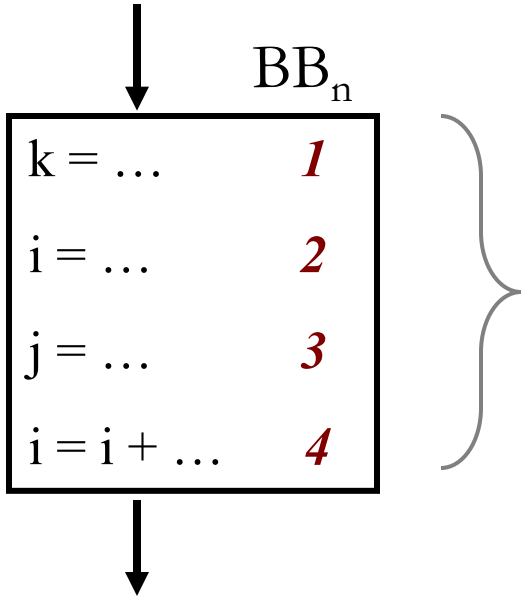
- Use-Def (UD) Chain
  - Connects a use of a variable v_k to all the possible definitions of that variable

DU-Chain Data-Flow 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*

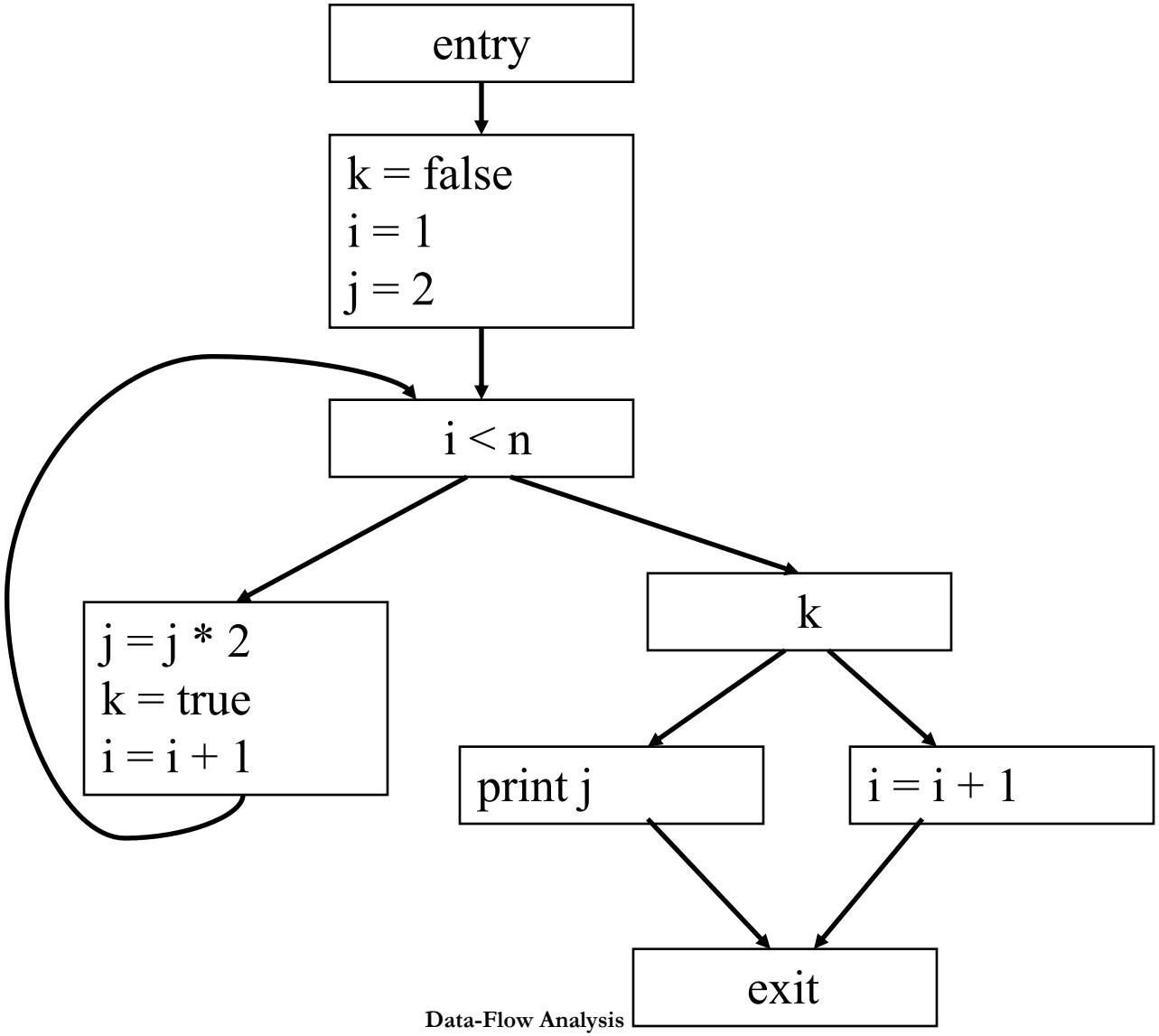
DU-Chain Gen and Kill Functions



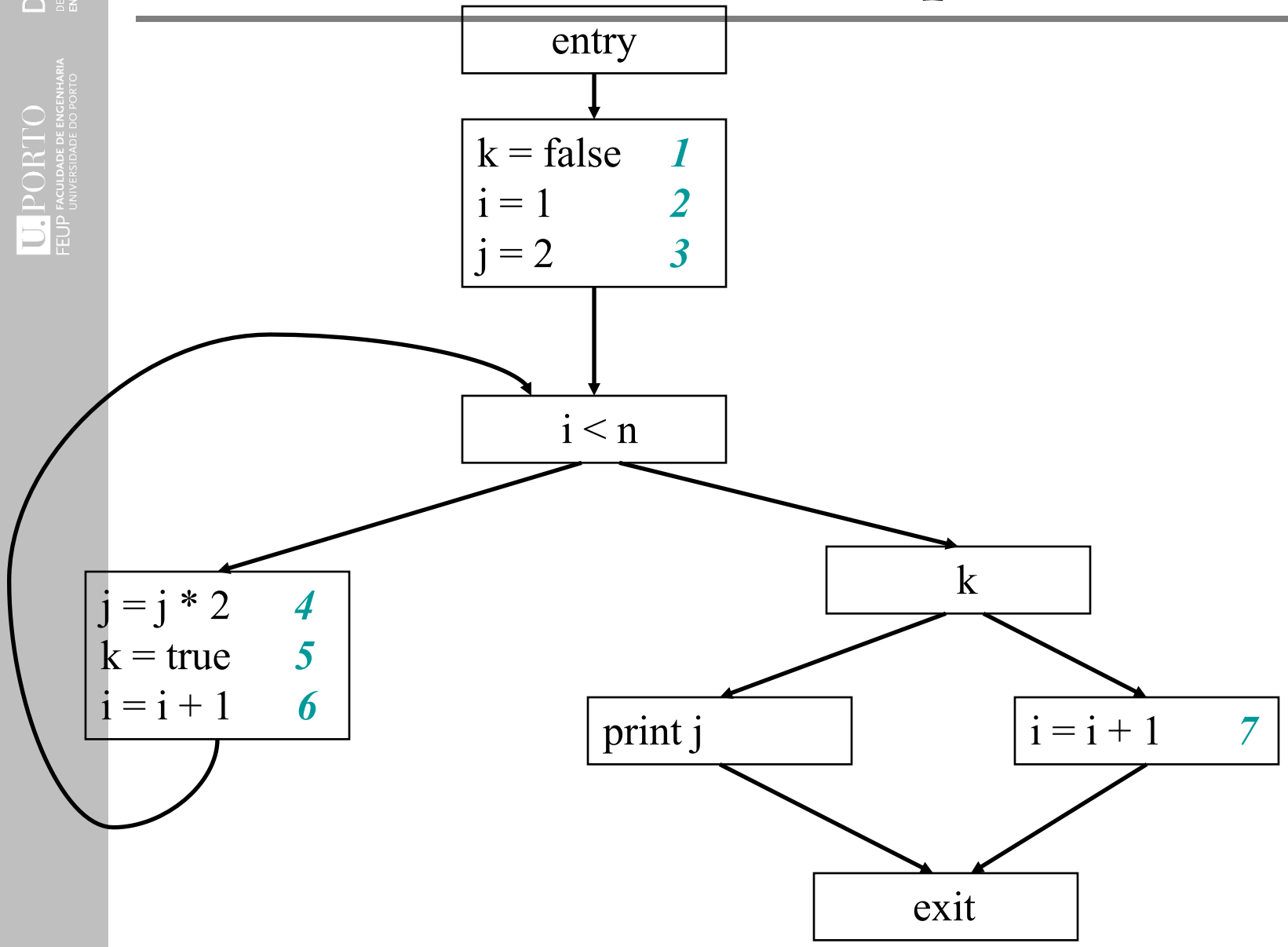
Gen(n) = { d1, d3 , d4 }
Kill(n) = { d1, d2, d3, d4, … }

- 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

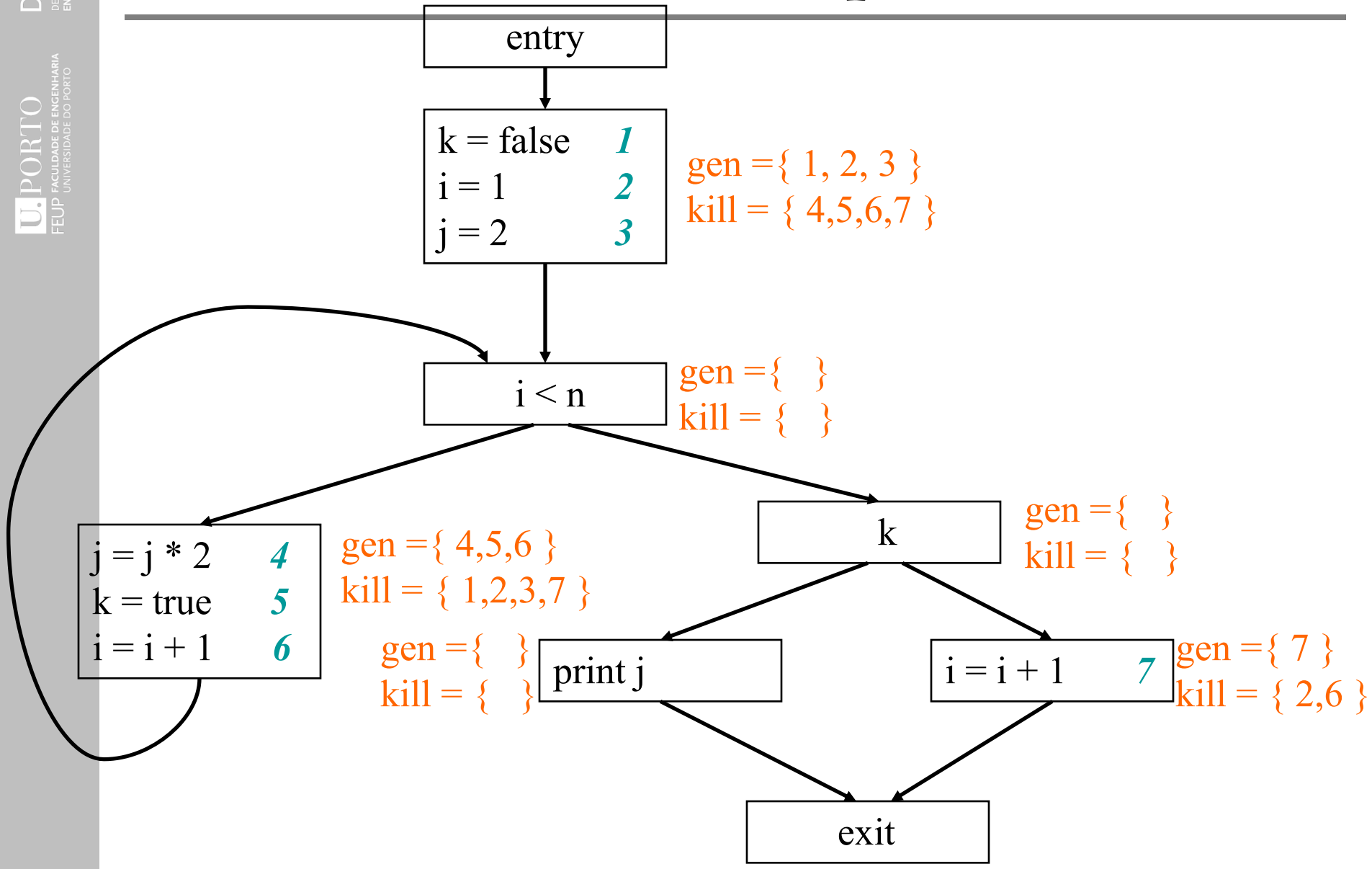
DU Example



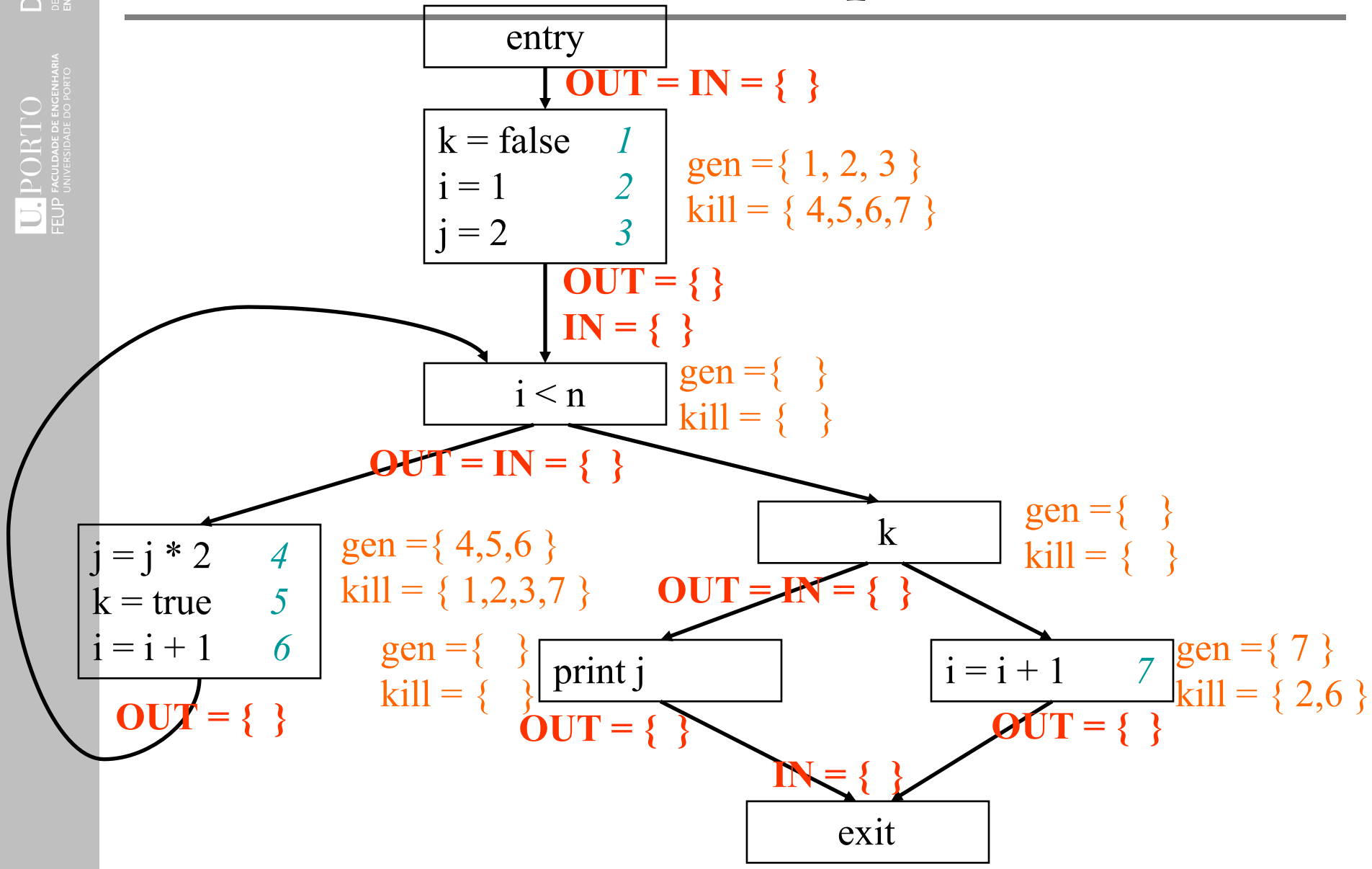
DU Example



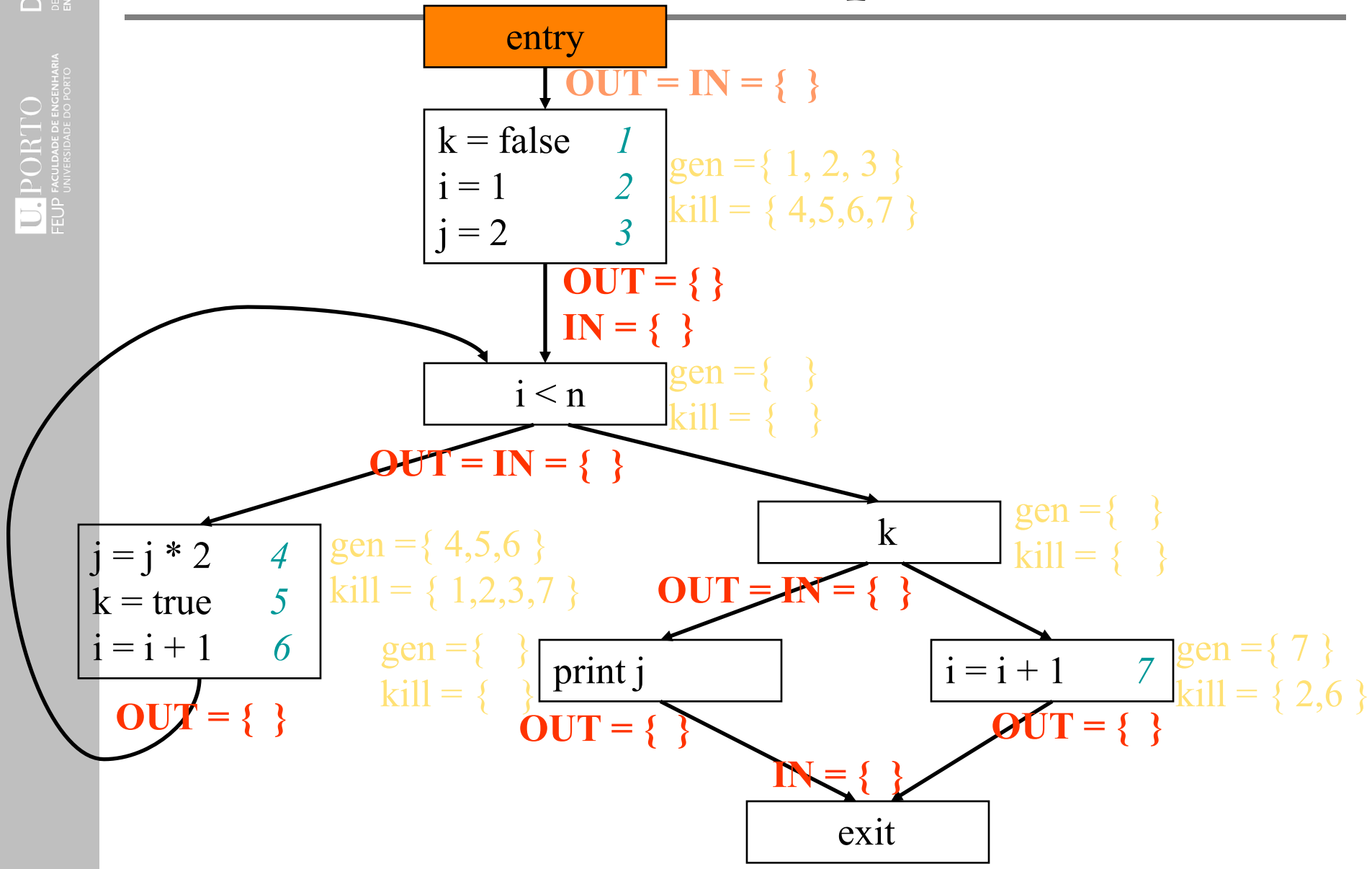
DU Example


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

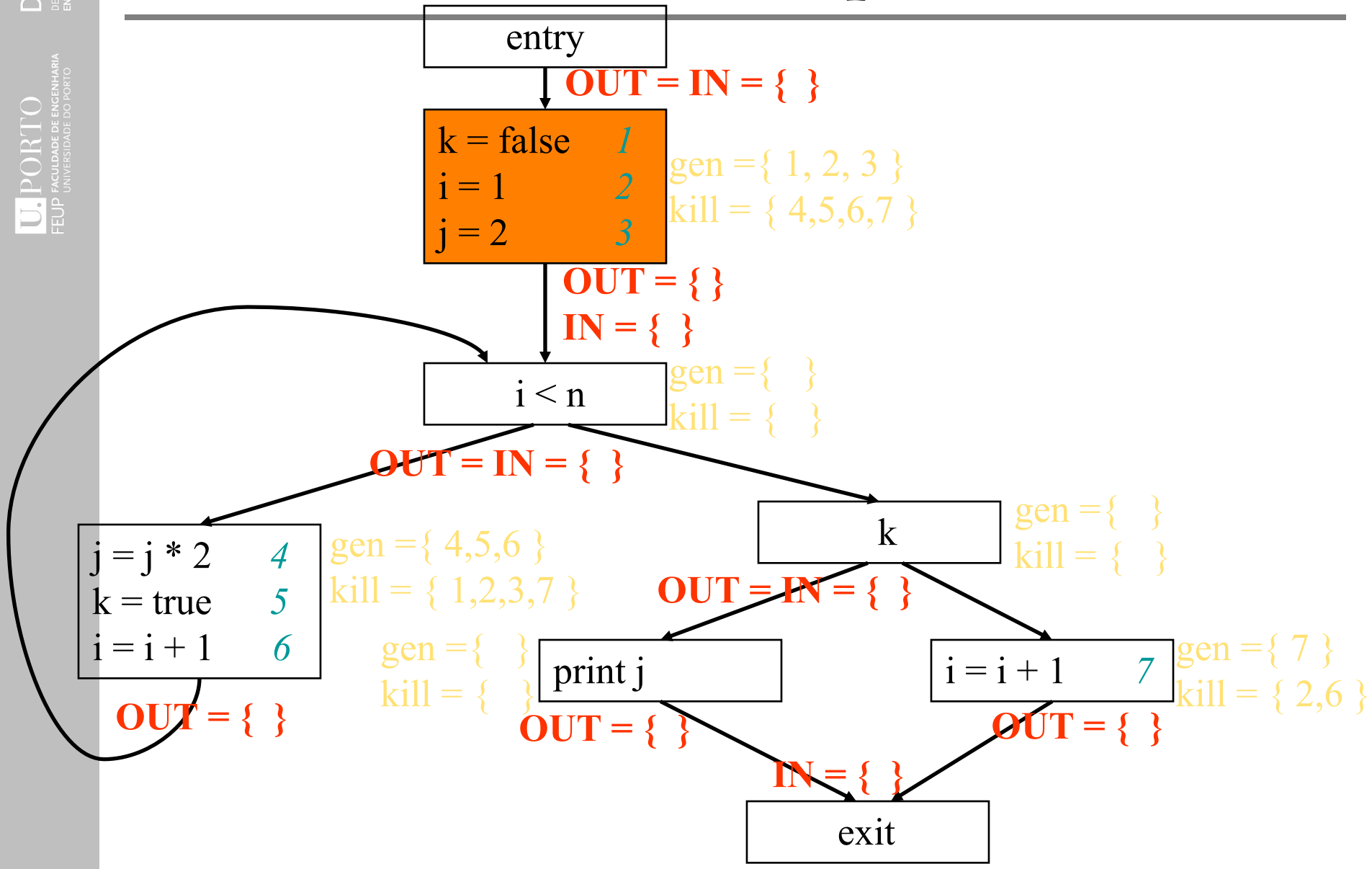
DU Example


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

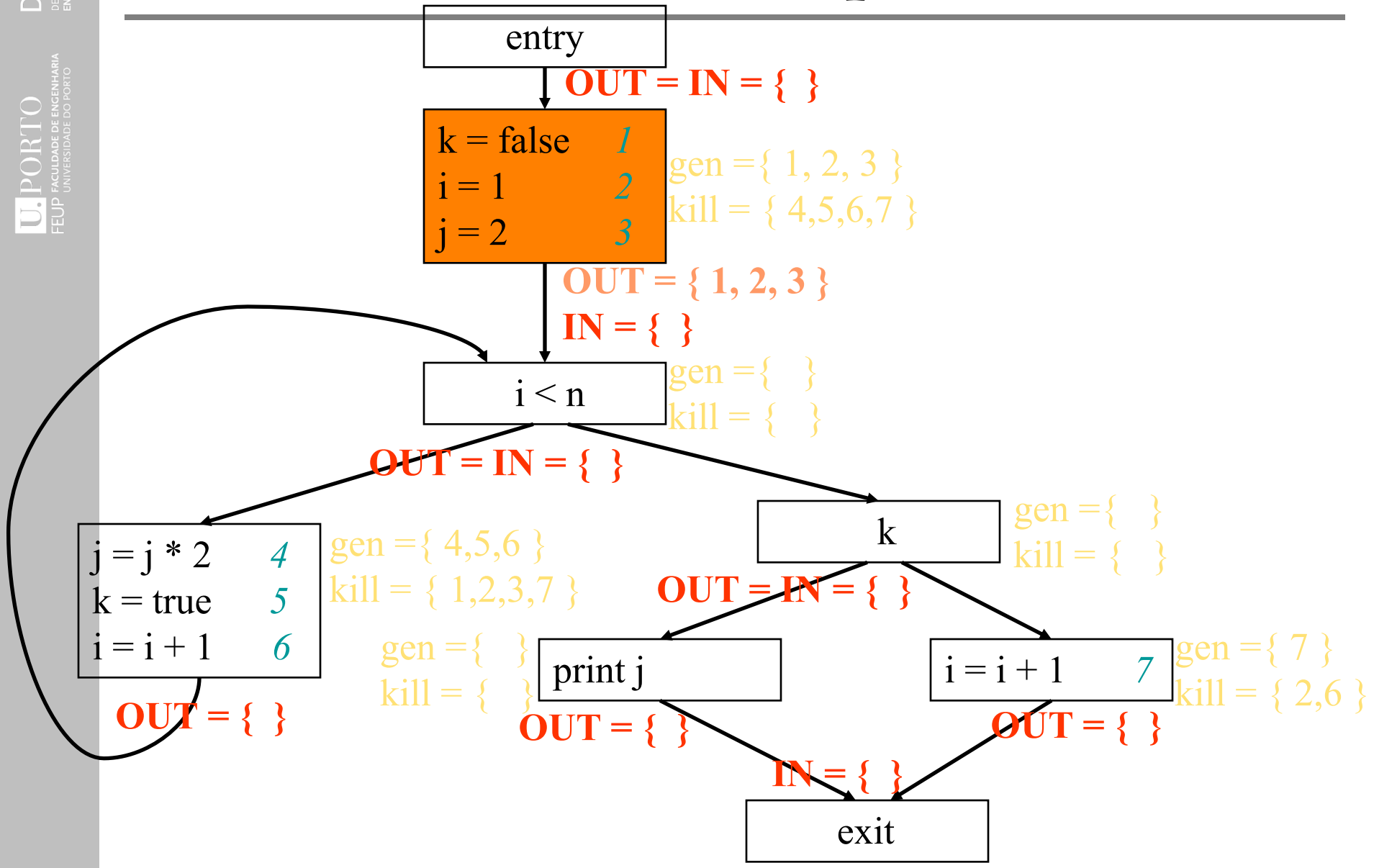
DU Example


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 = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

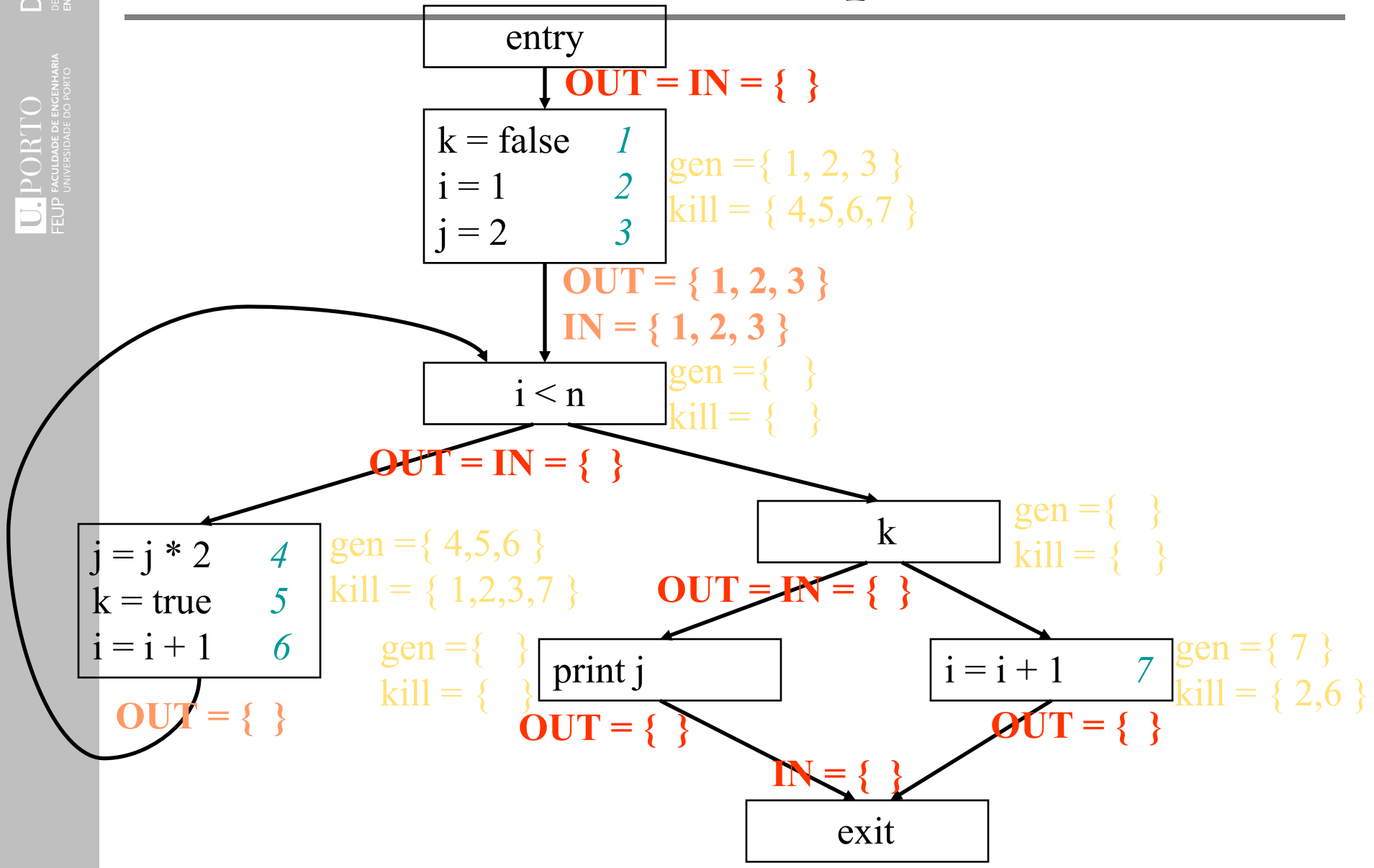
DU Example


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 = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

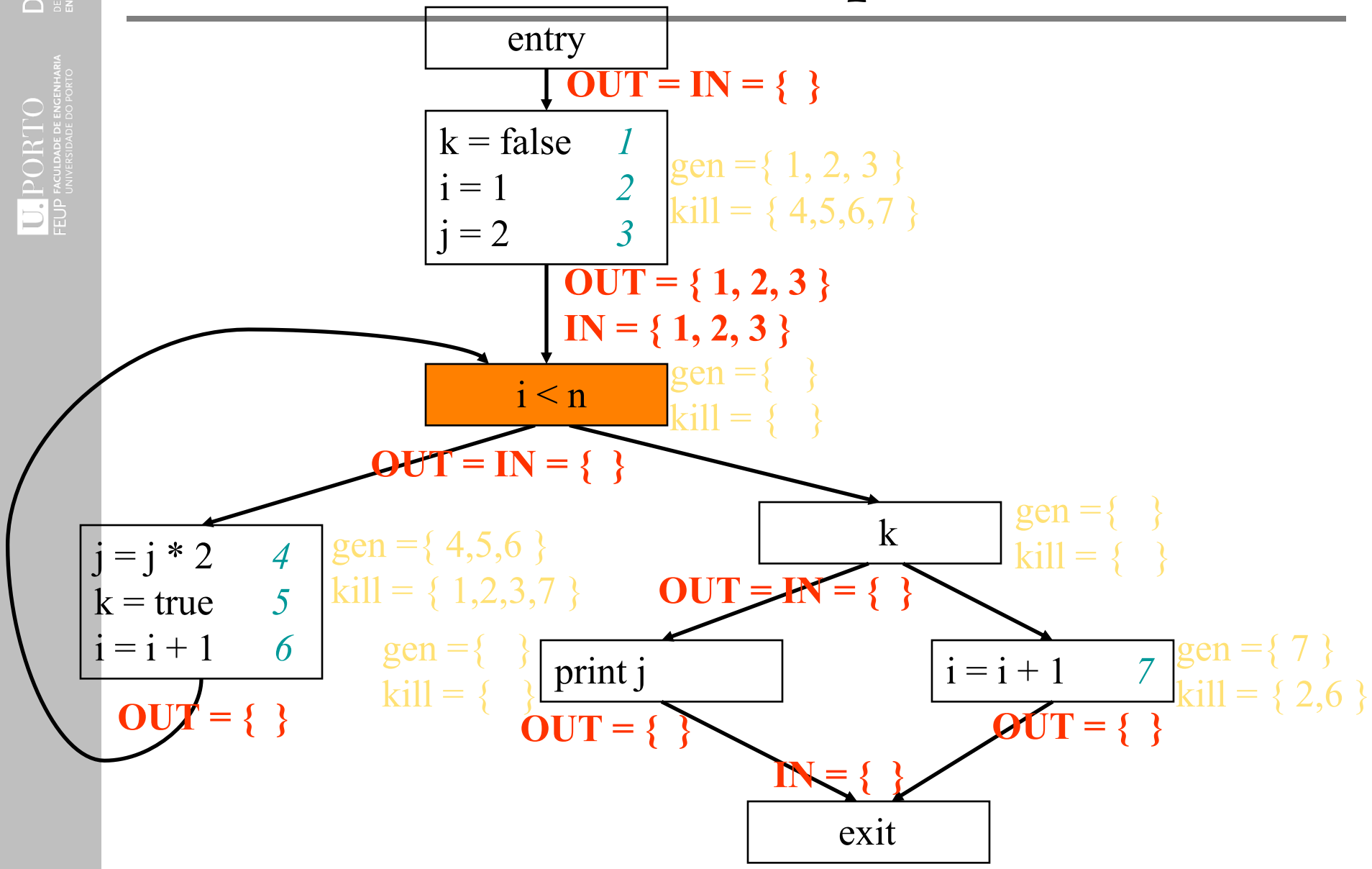
DU Example


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 = { }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

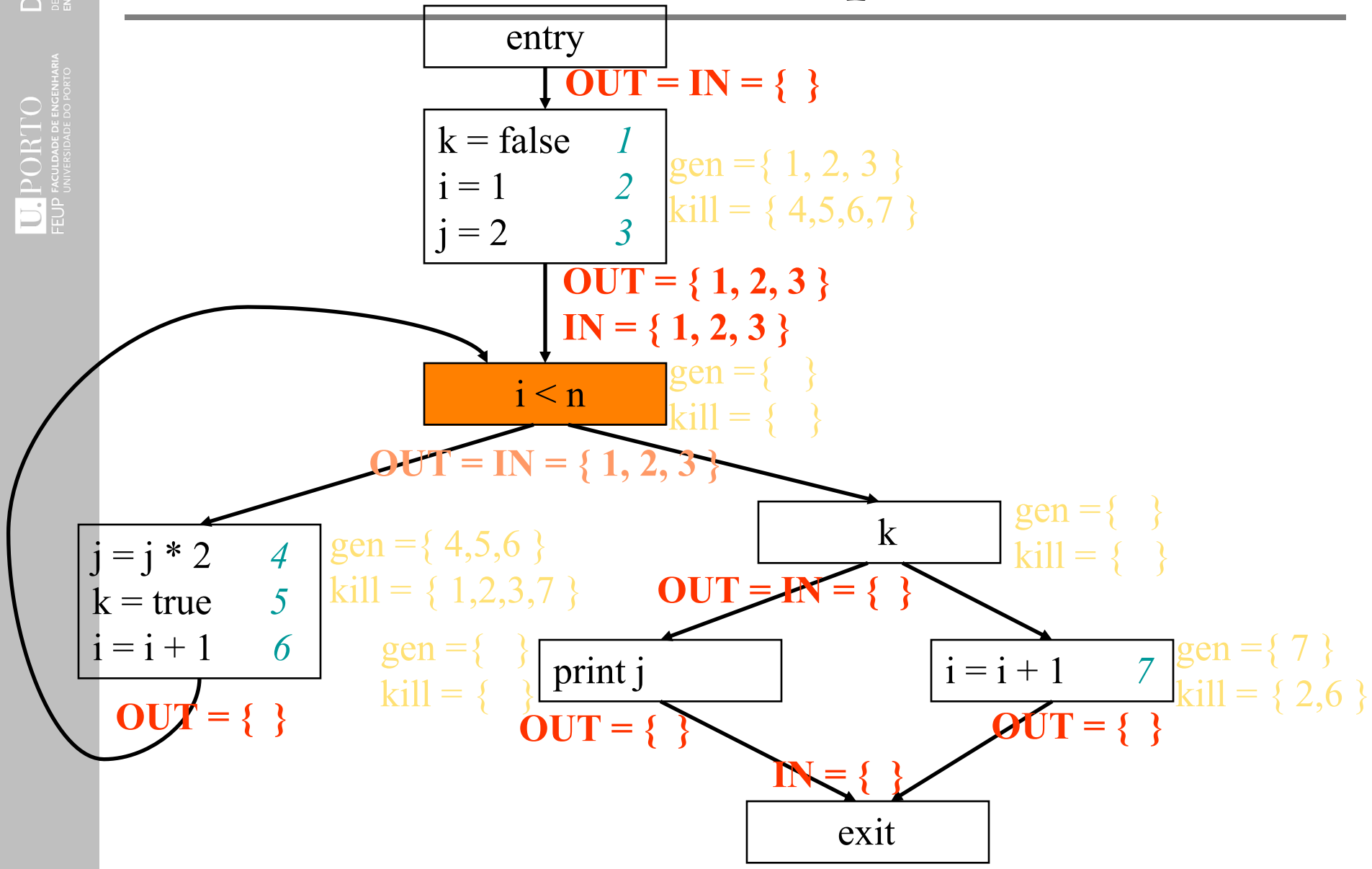
DU Example


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

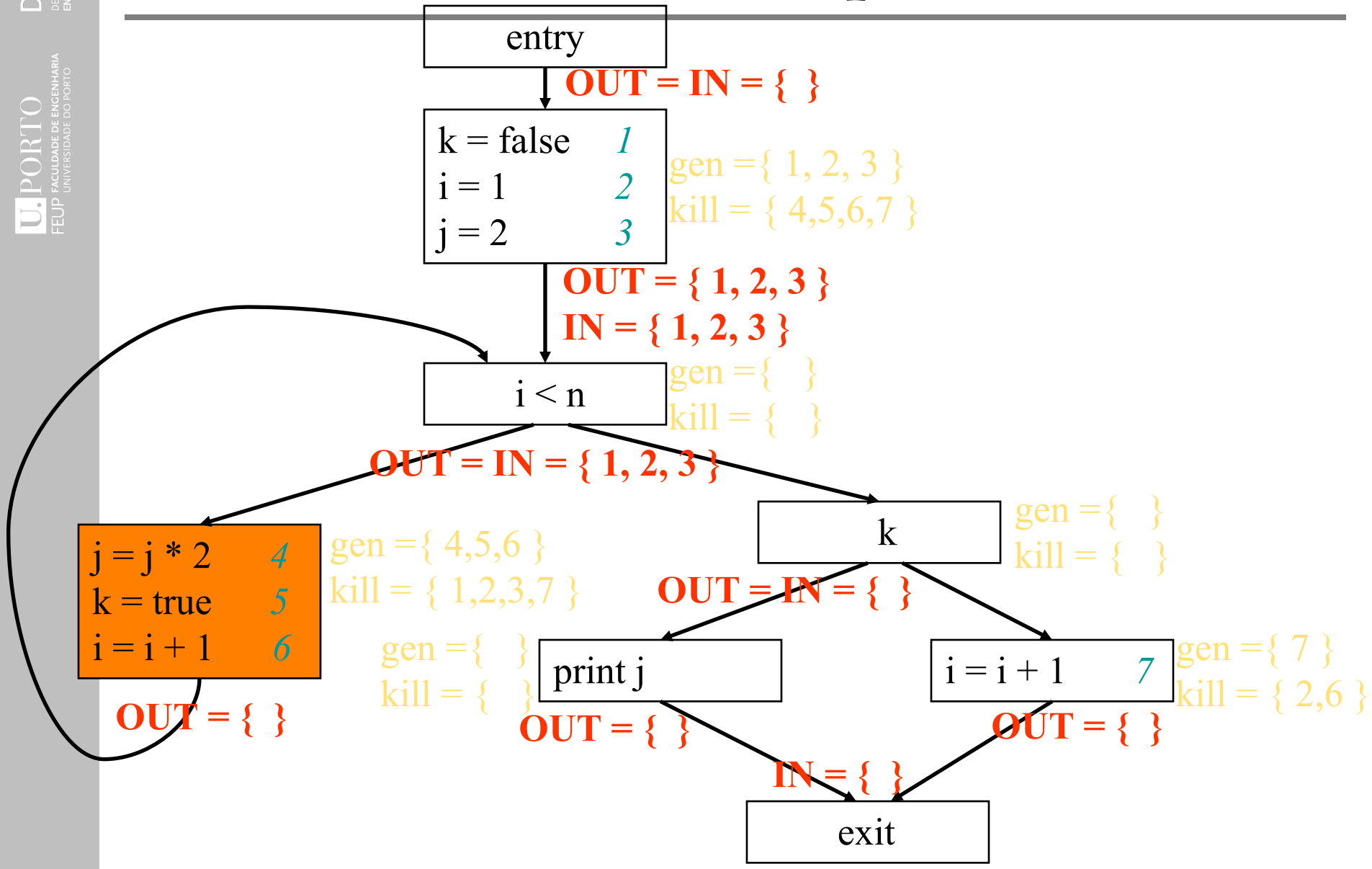
DU Example


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 = { }
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

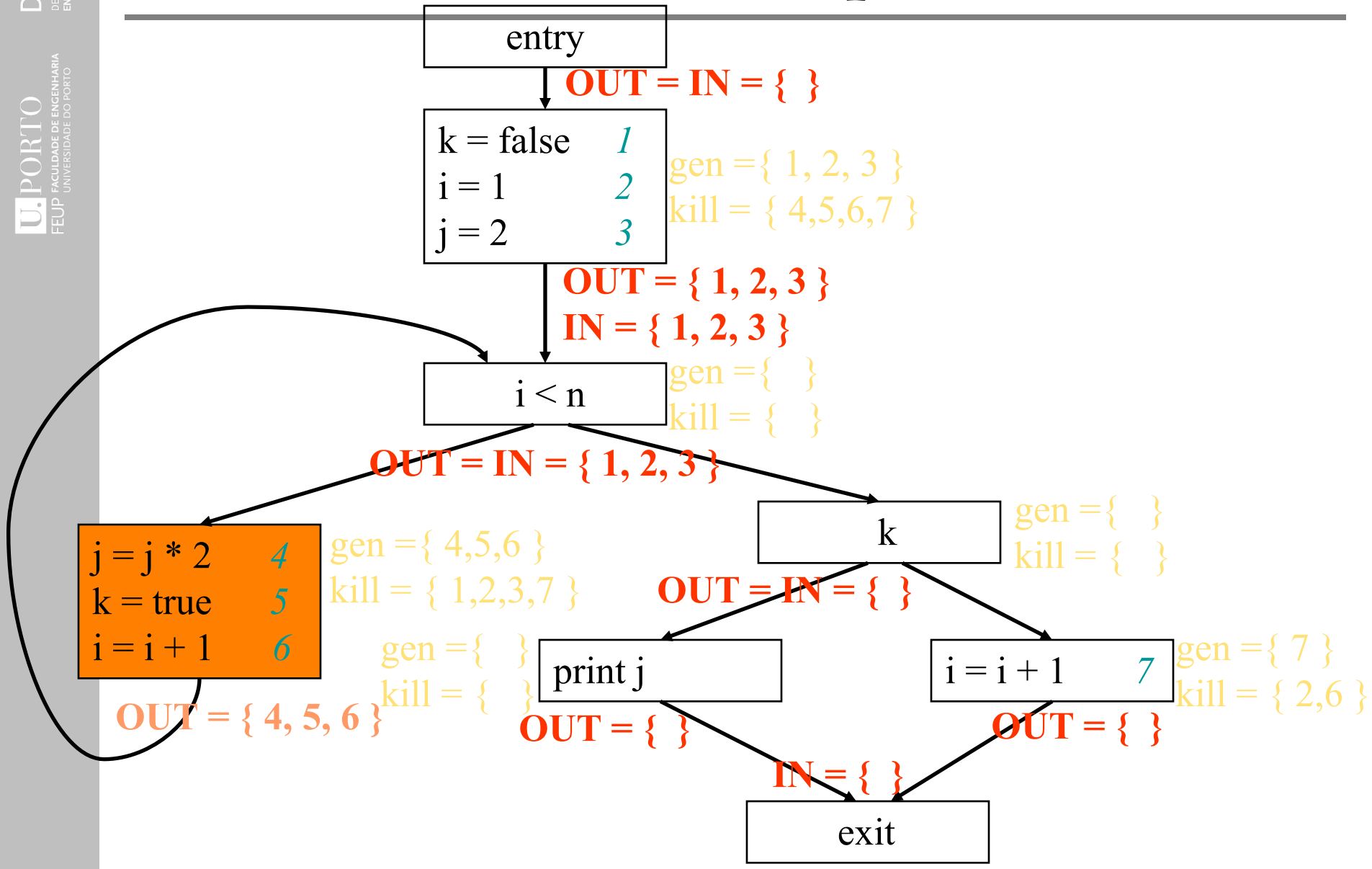
DU Example


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

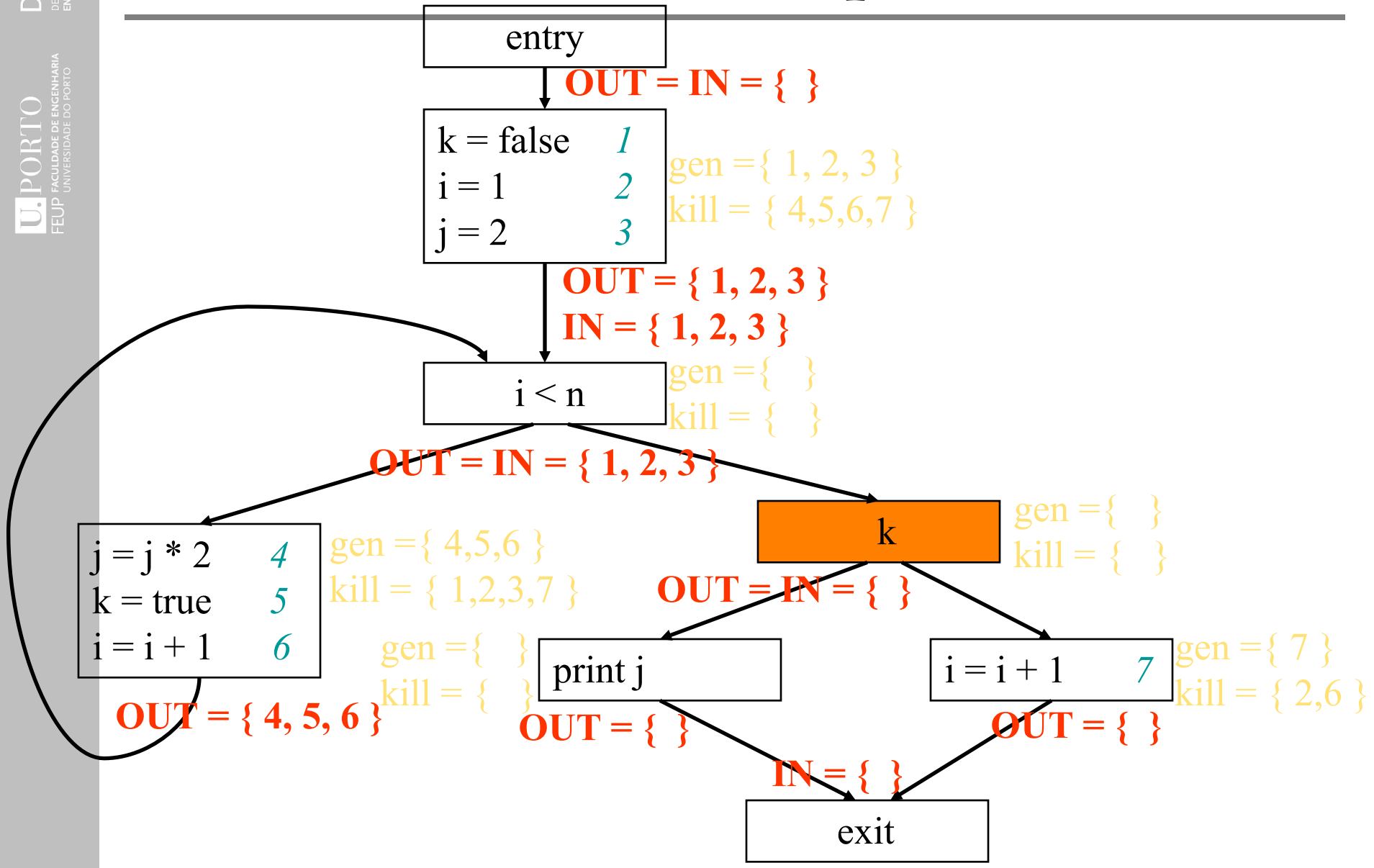
DU Example


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 = { }
k
gen ={ }
kill = { }
OUT = IN = { }
print j
gen ={ }
kill = { }
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

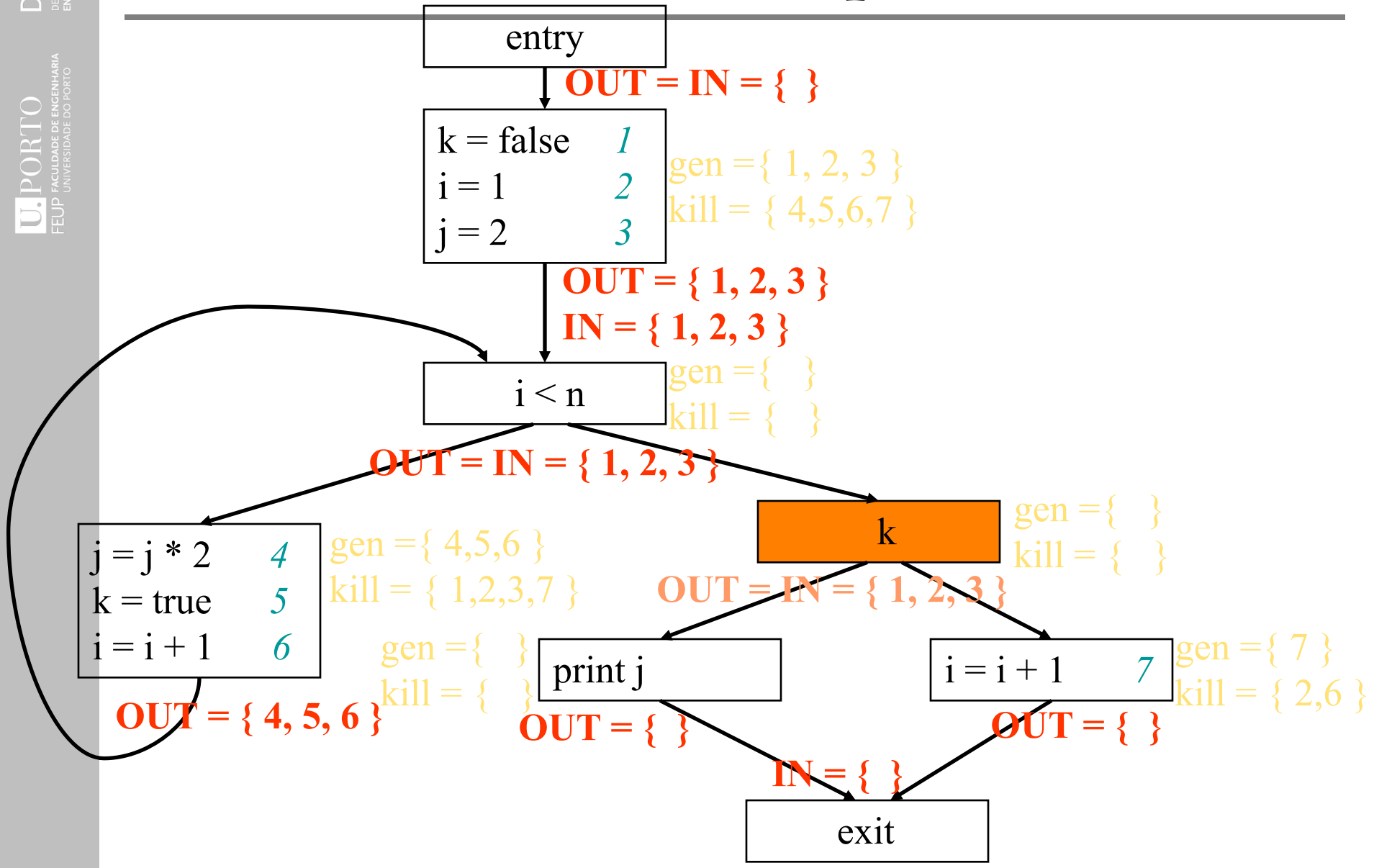
DU Example


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

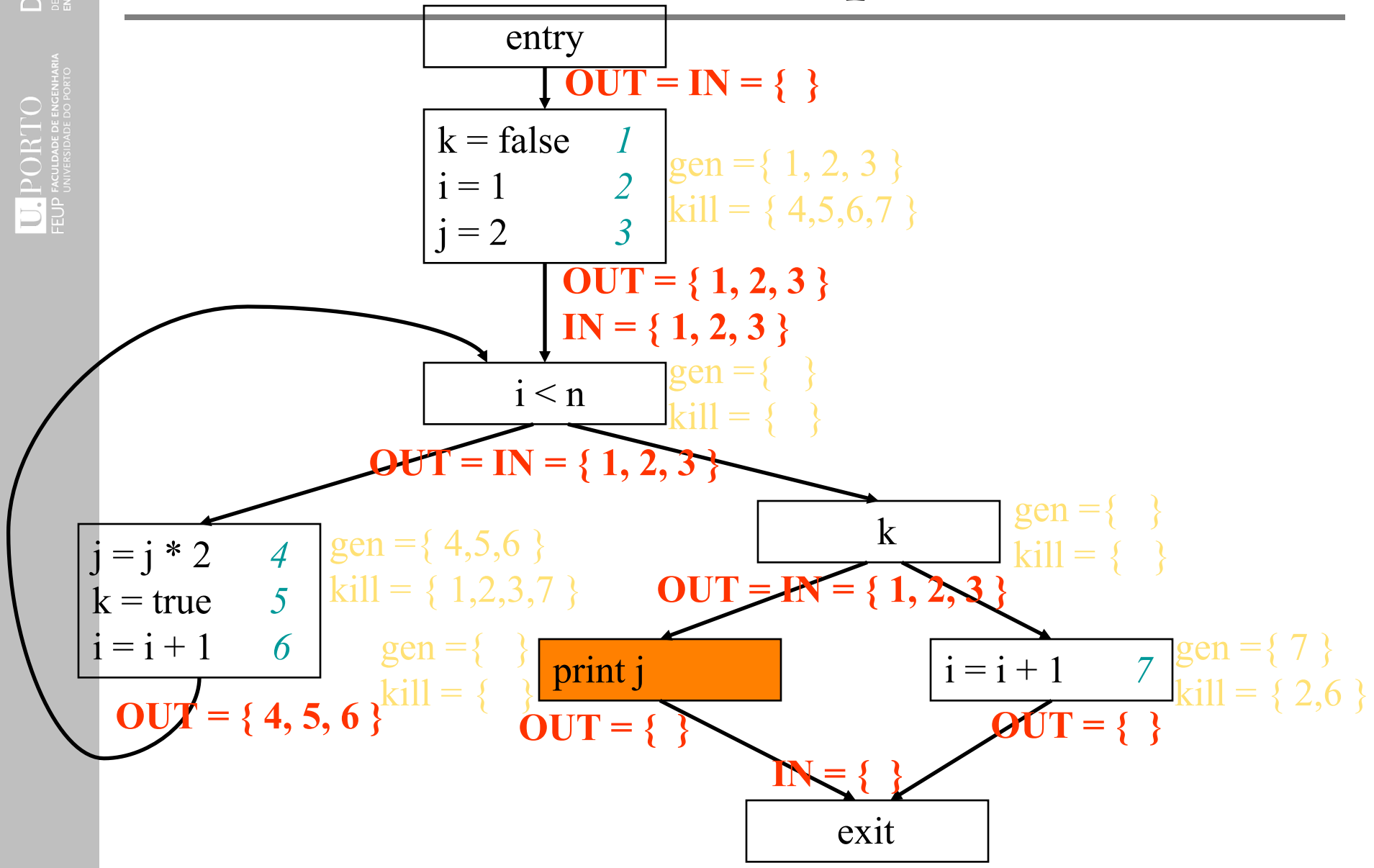
DU Example


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

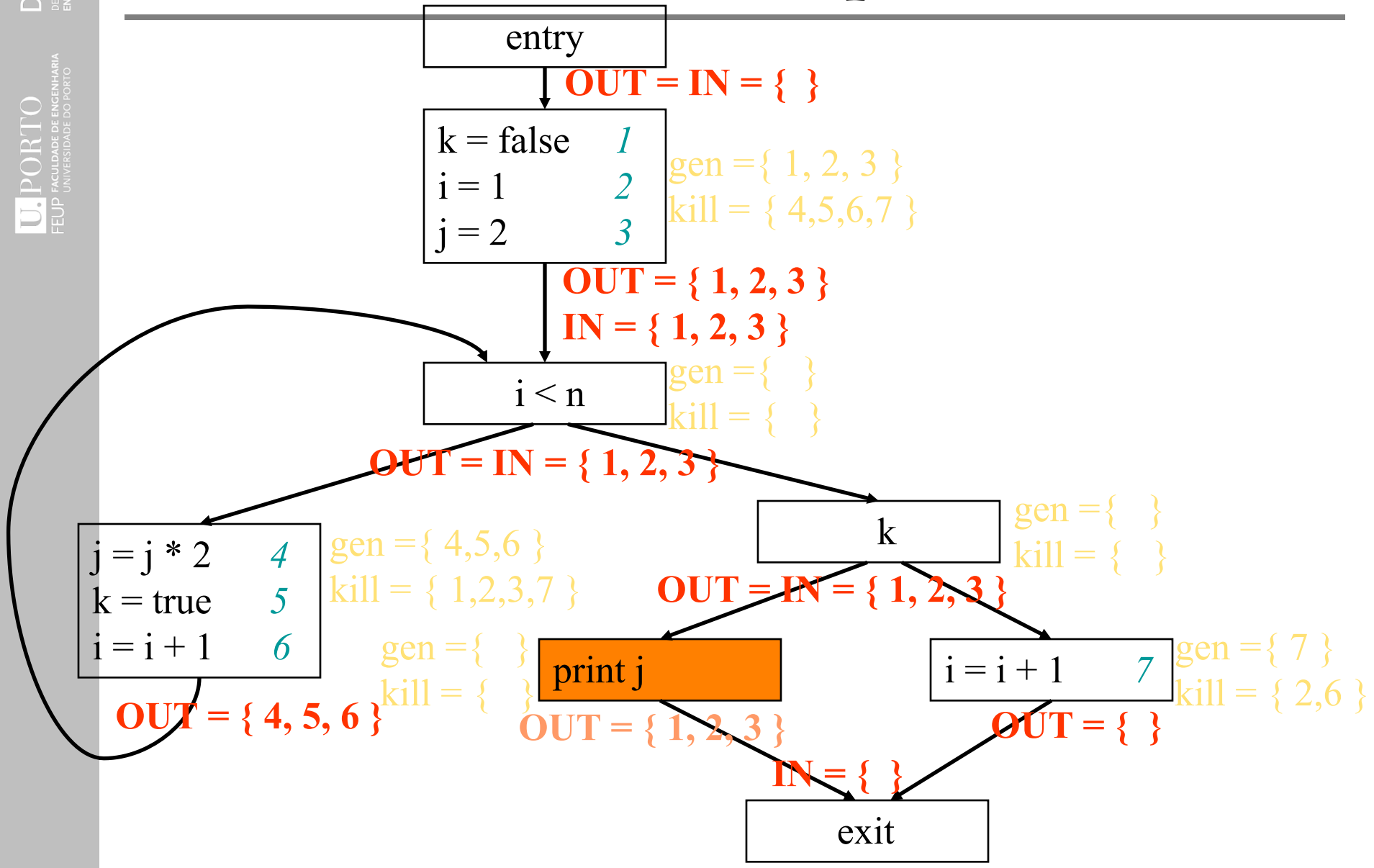
DU Example


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

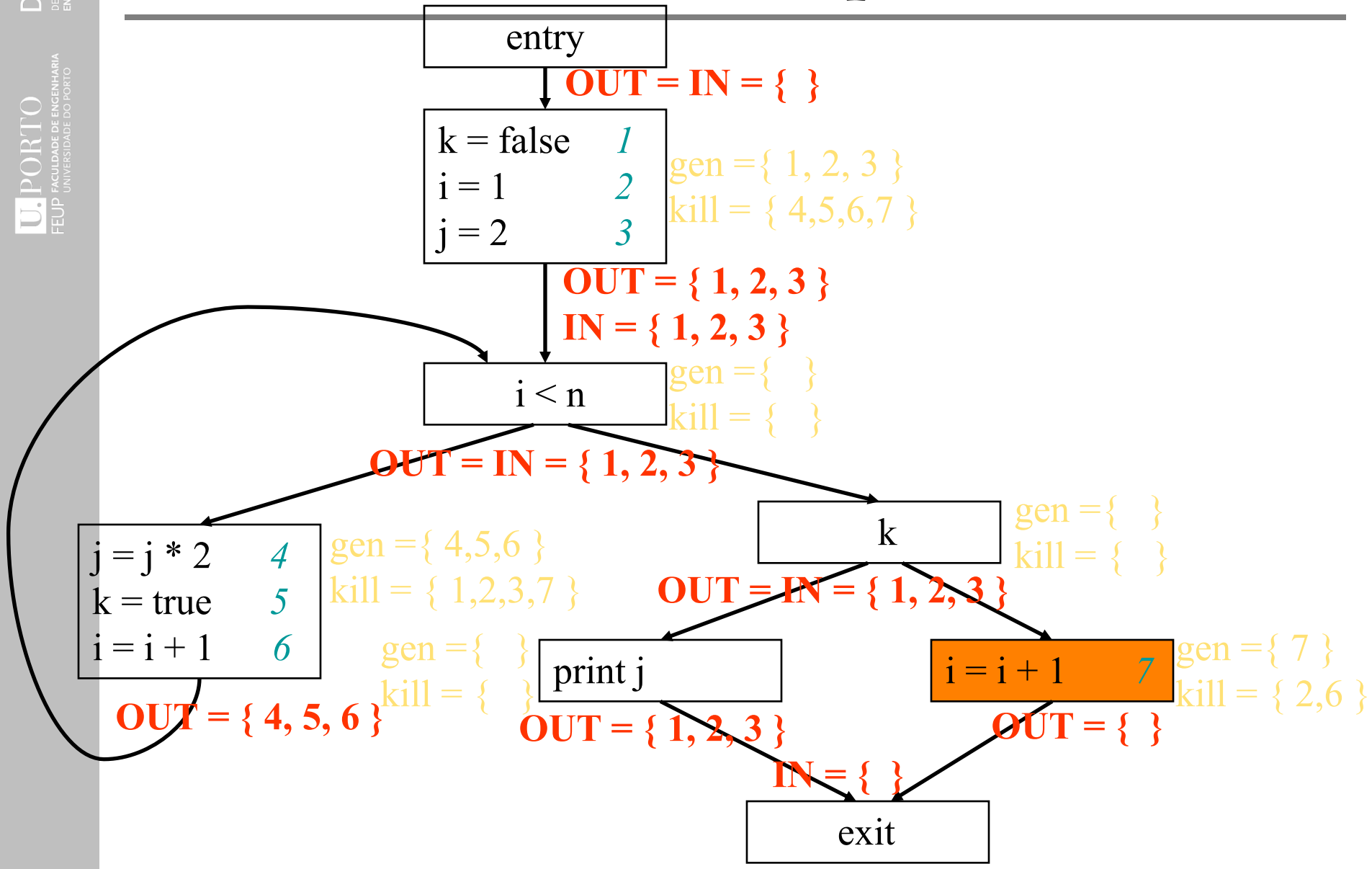
DU Example


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 }
gen ={ }
kill = { }
print j
OUT = { }
i = i + 1 7
gen ={ 7 }
kill = { 2,6 }
OUT = { }
IN = { }
exit

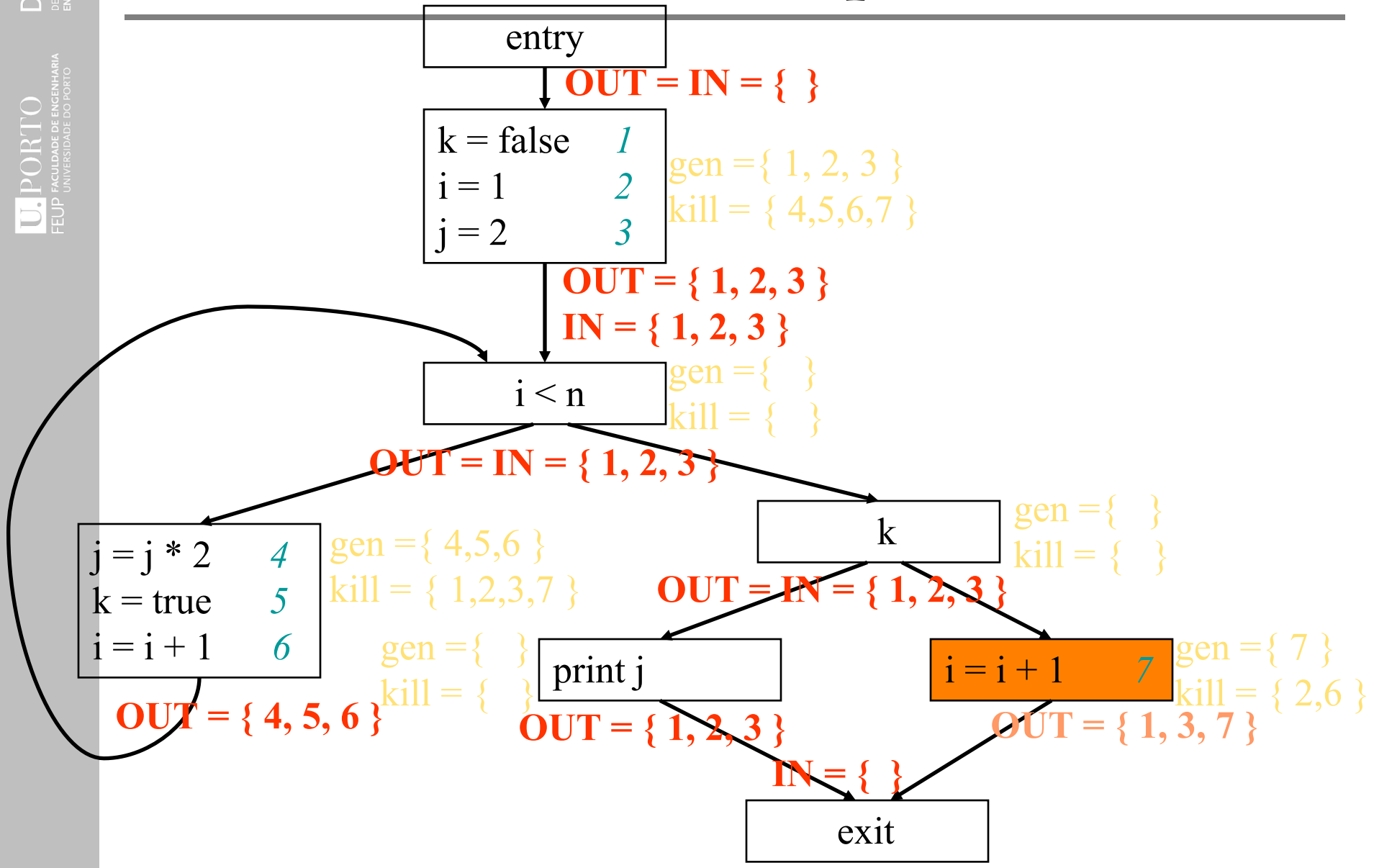
DU Example


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 = { 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

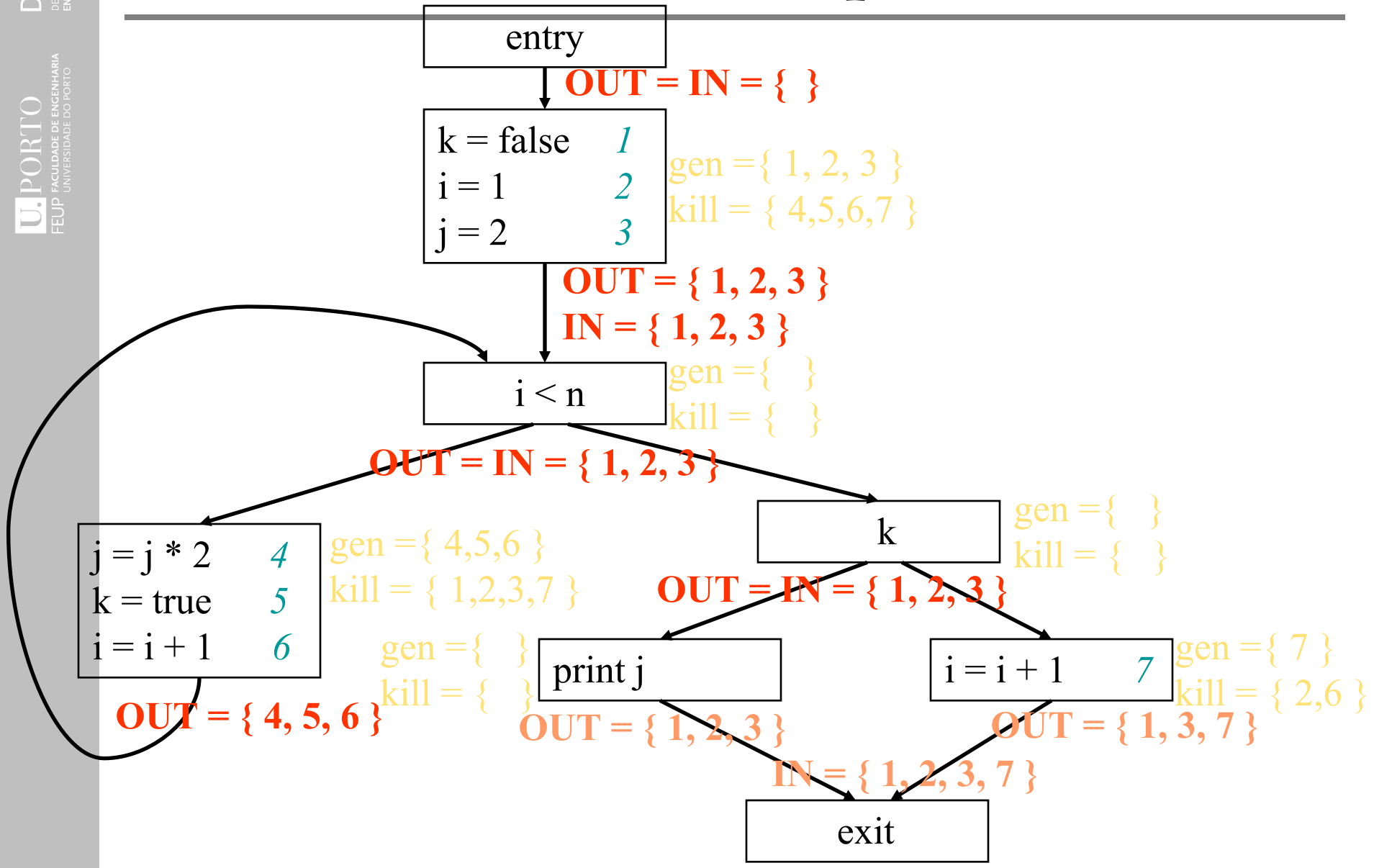
DU Example


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 = { }
IN = { }
exit

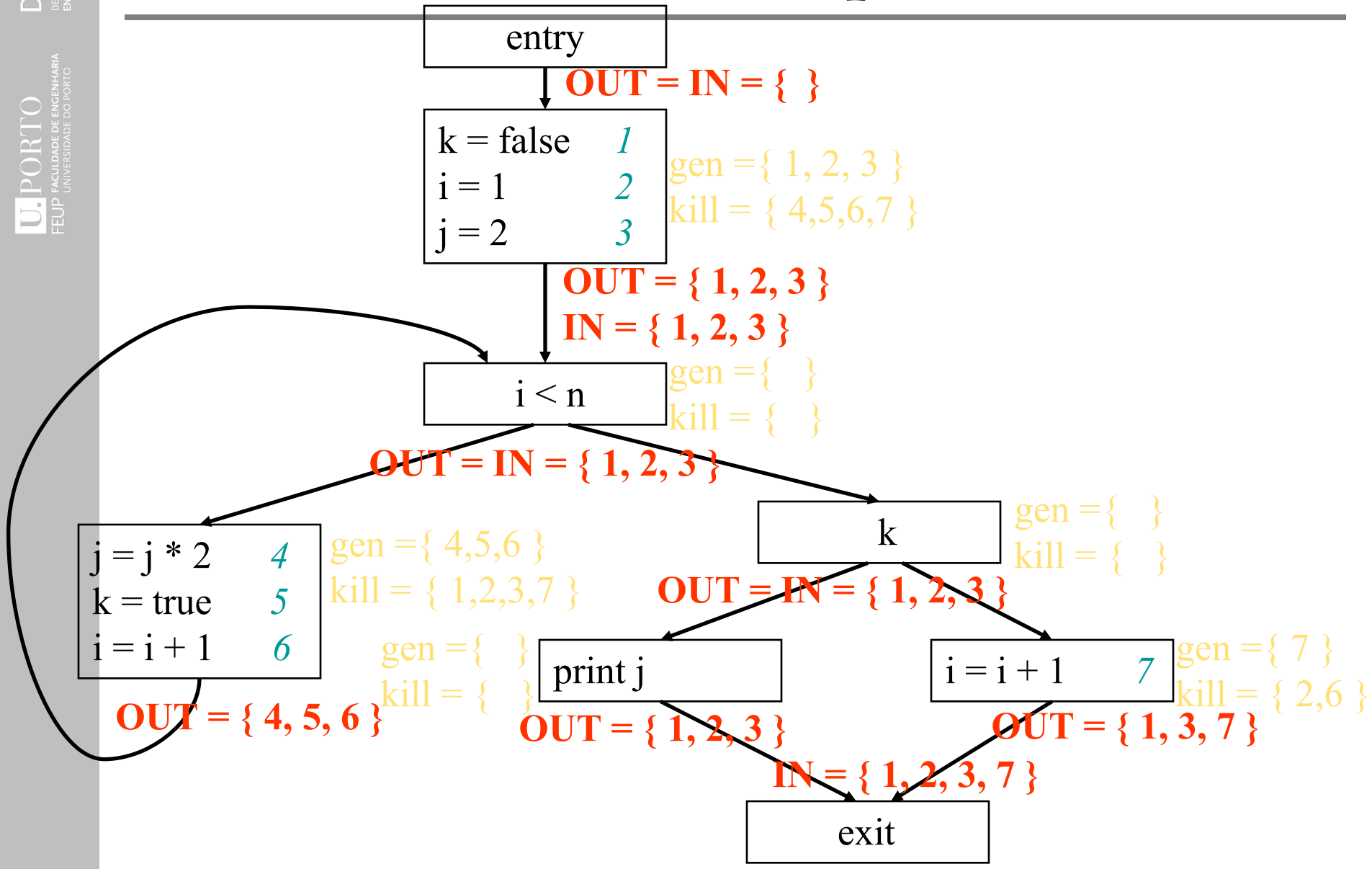
DU Example


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

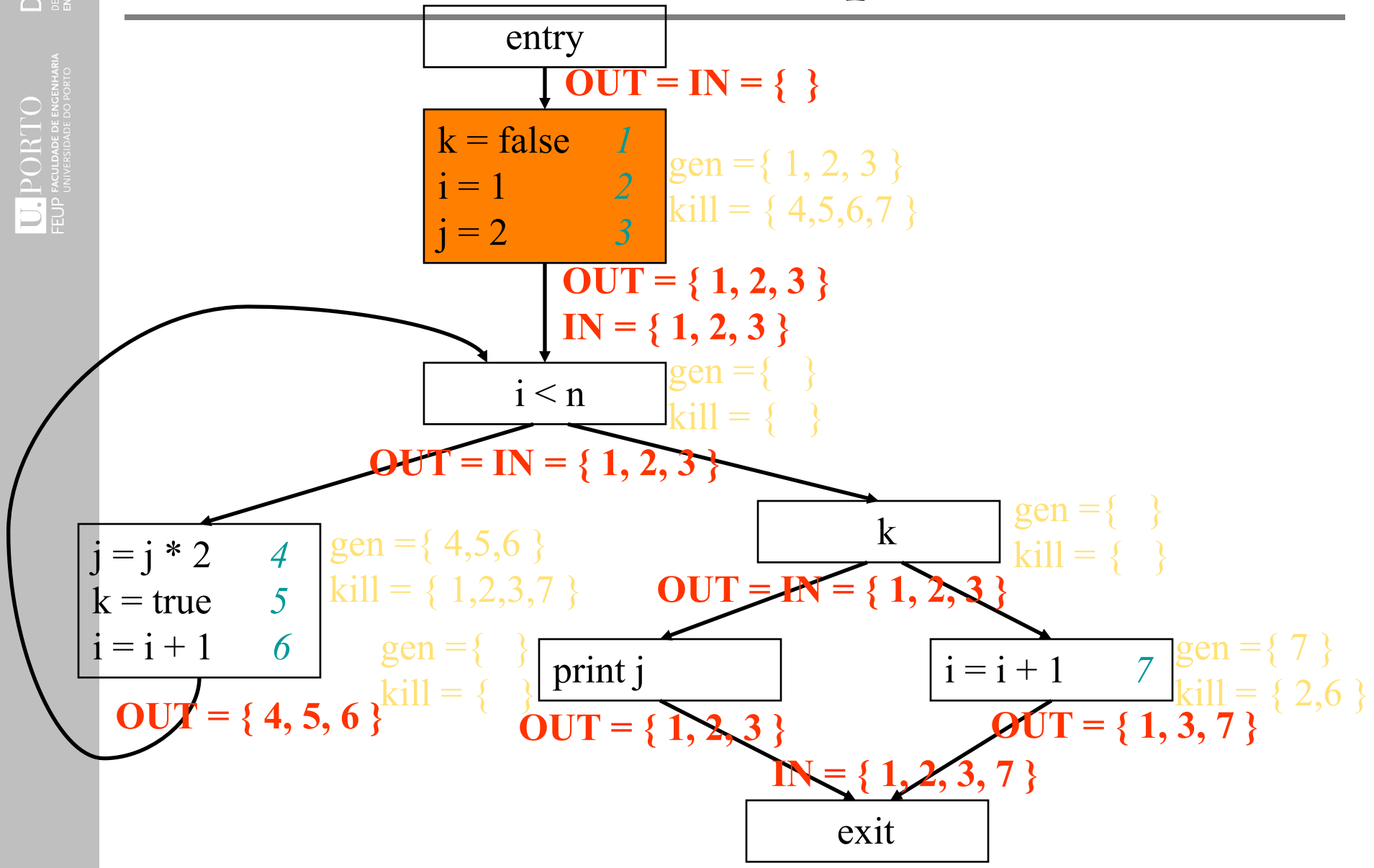
DU Example


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

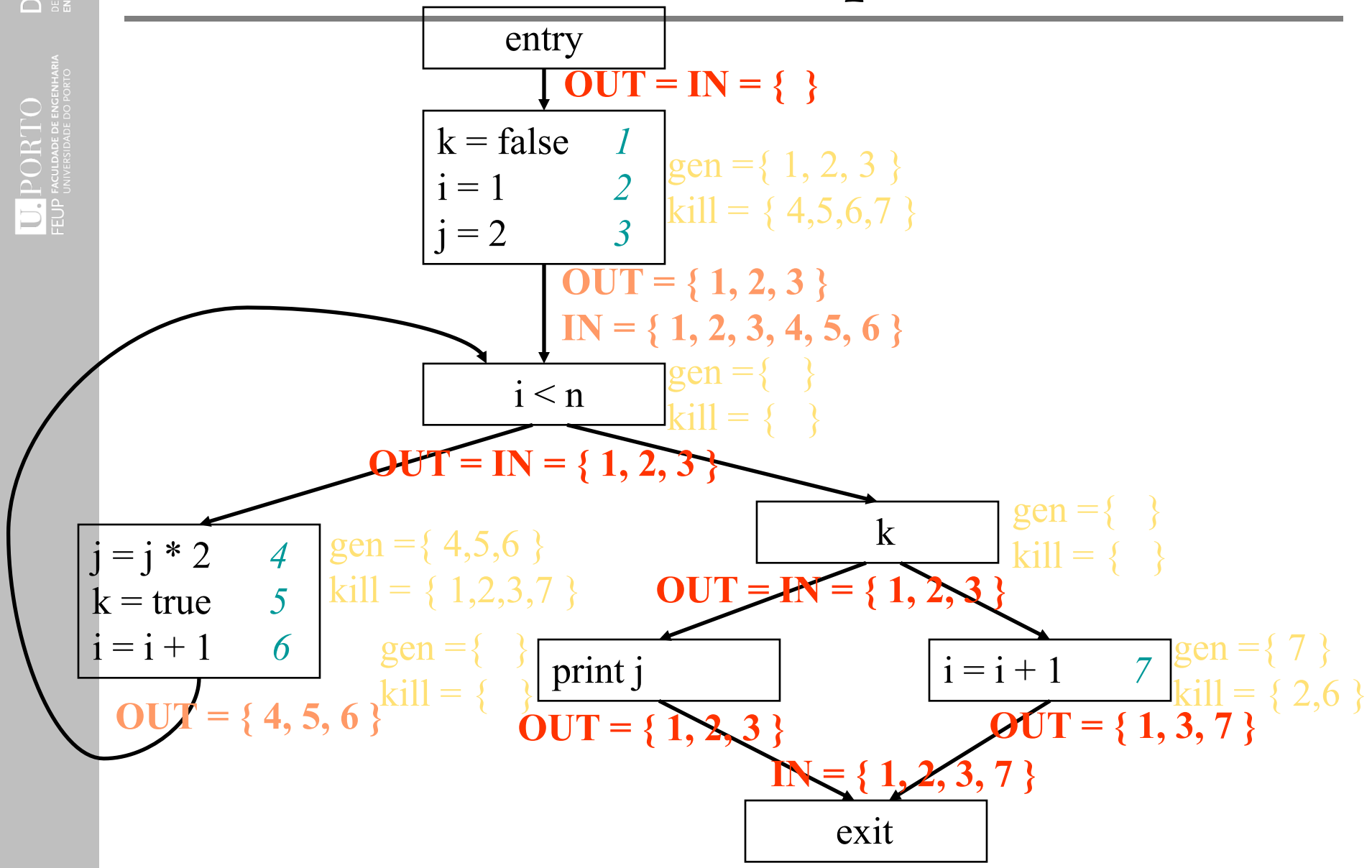
DU Example


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

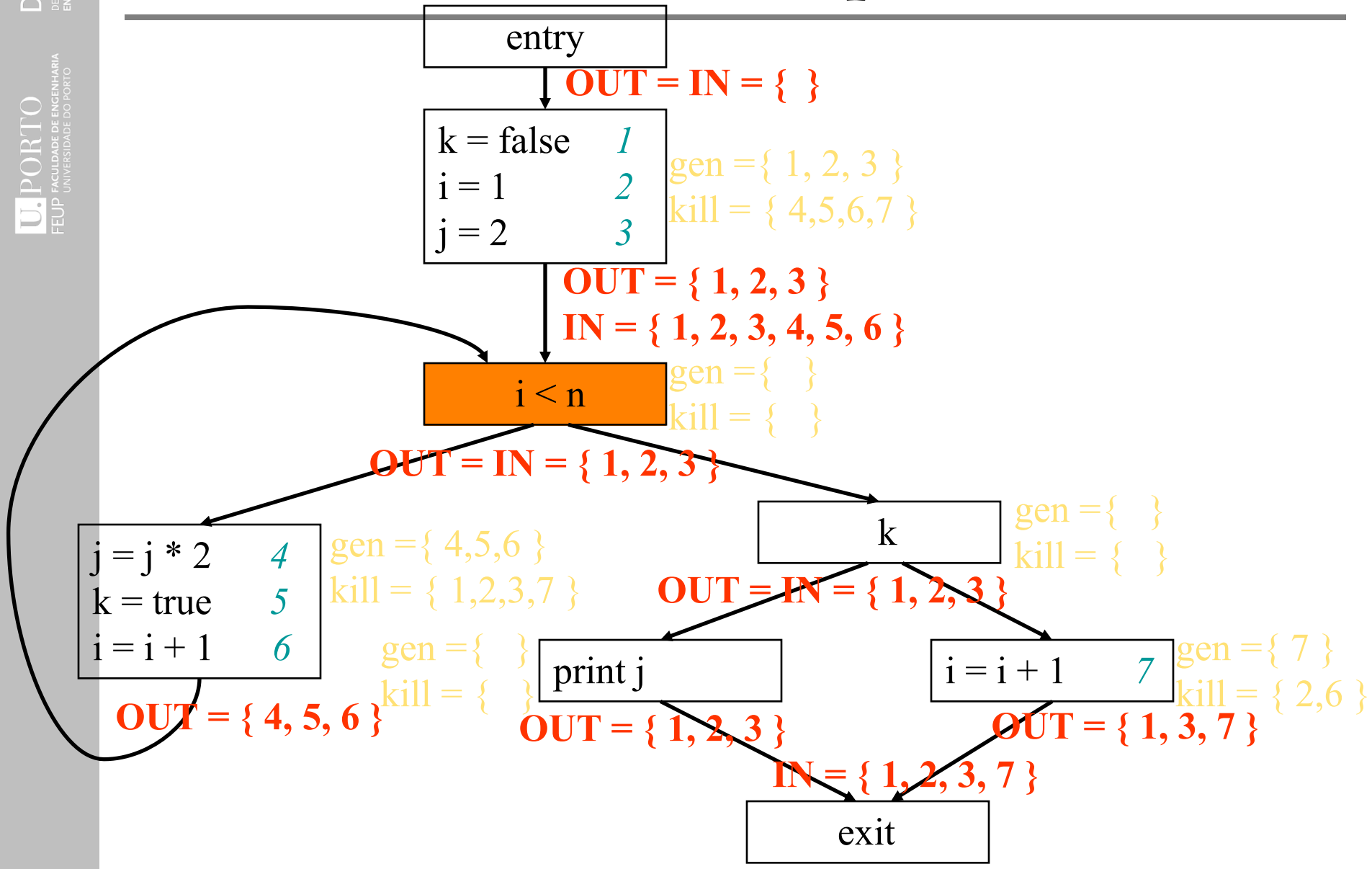
DU Example


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

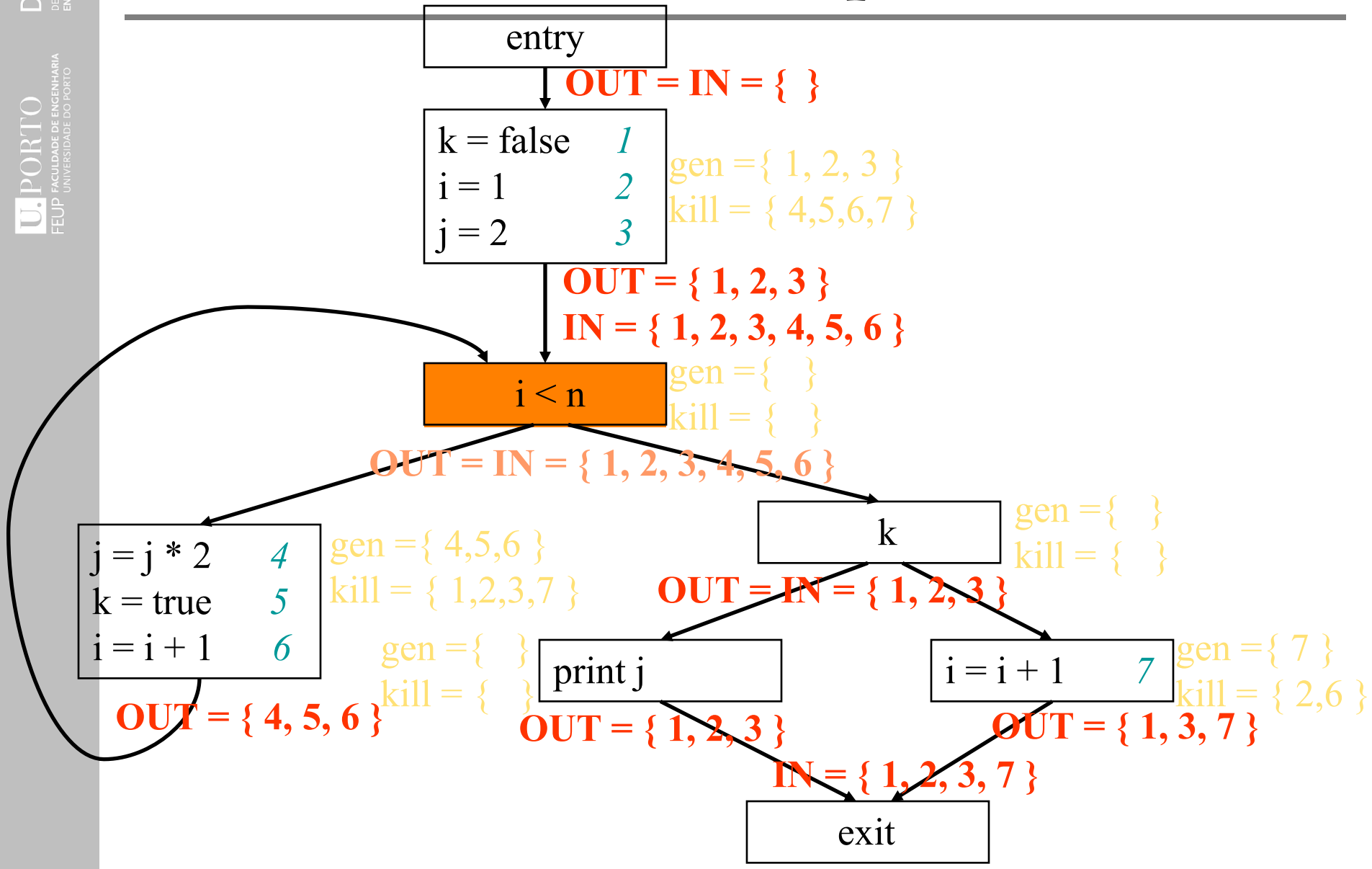
DU Example


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

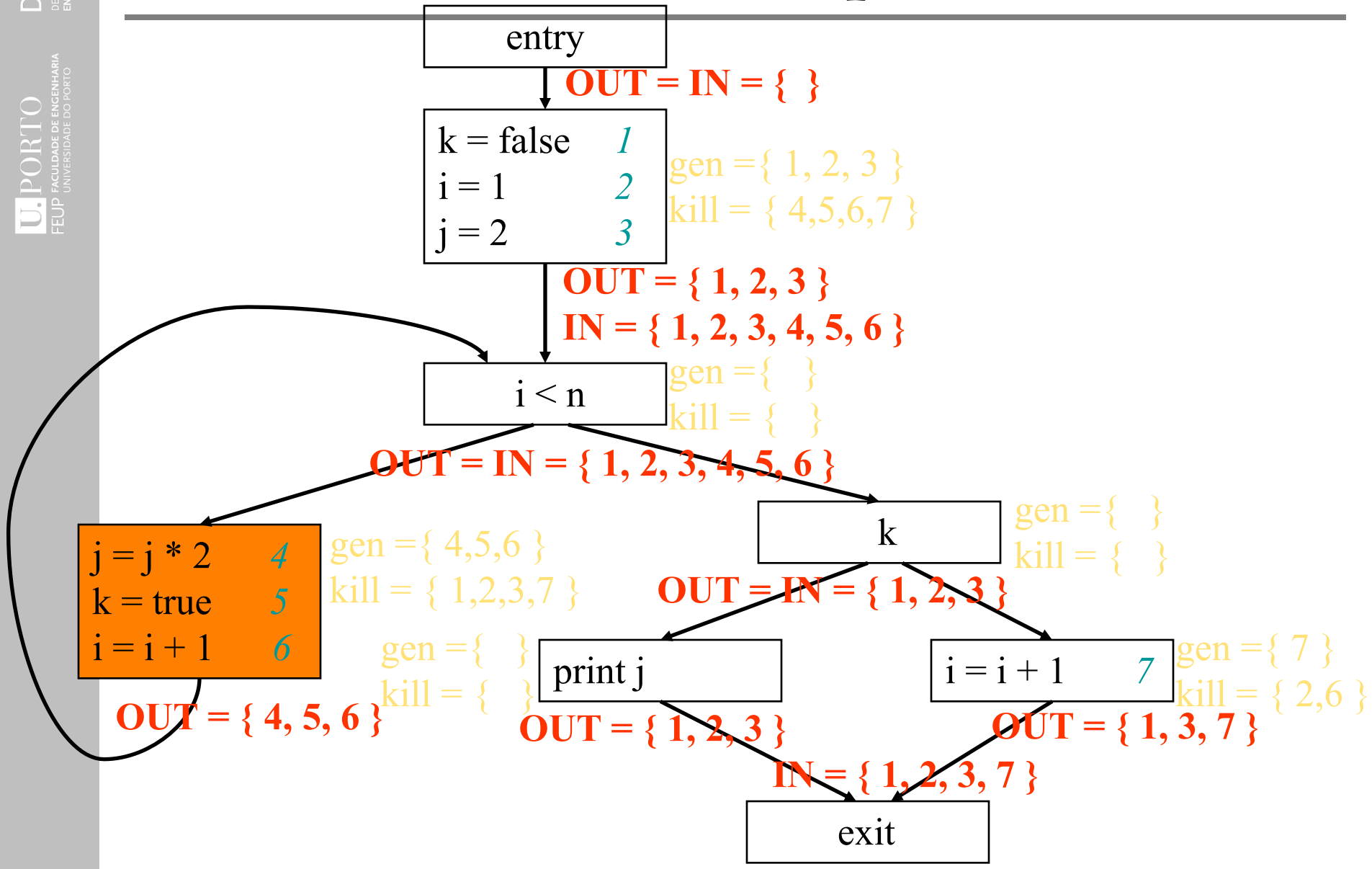
DU Example


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

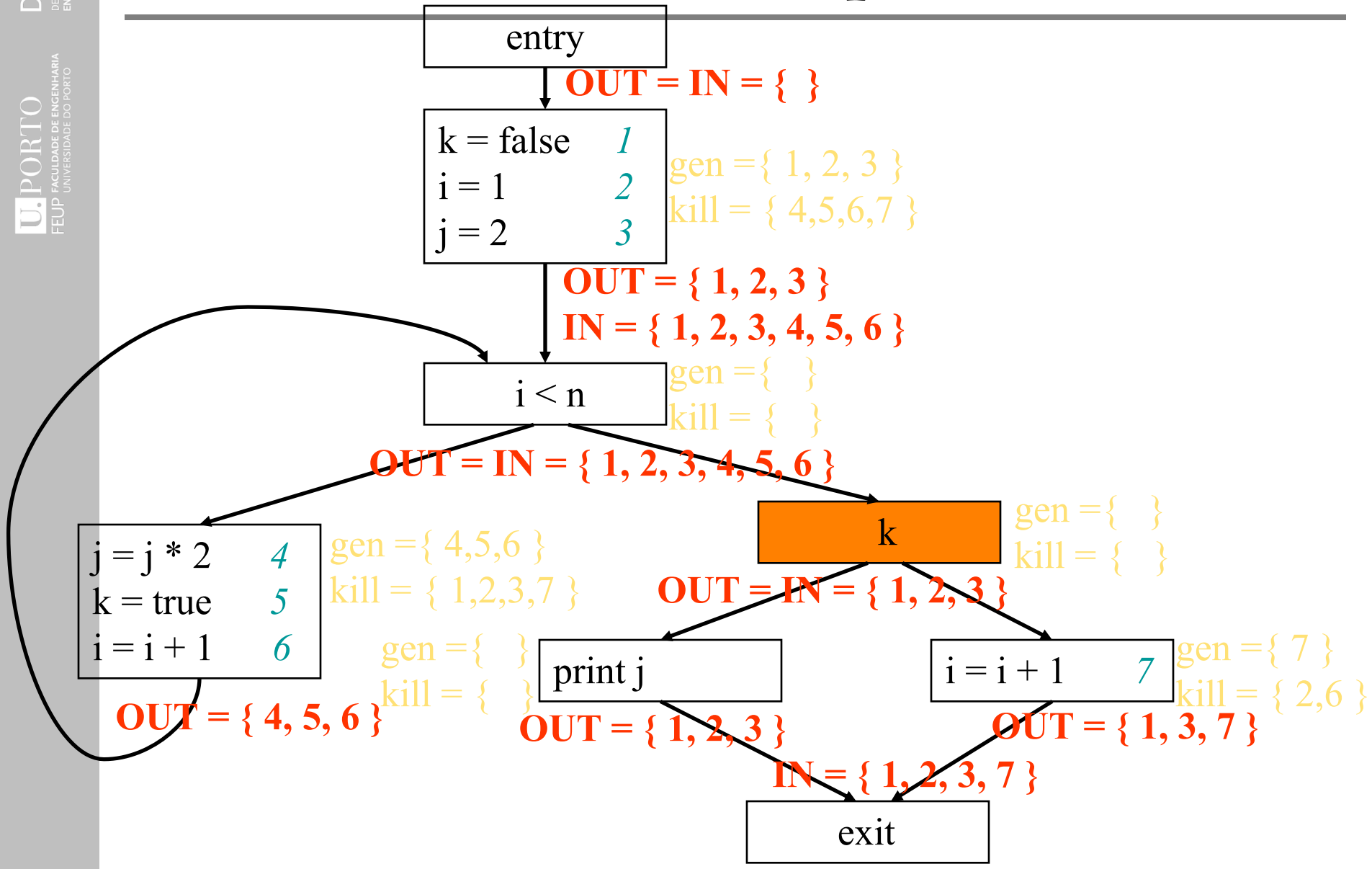
DU Example


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

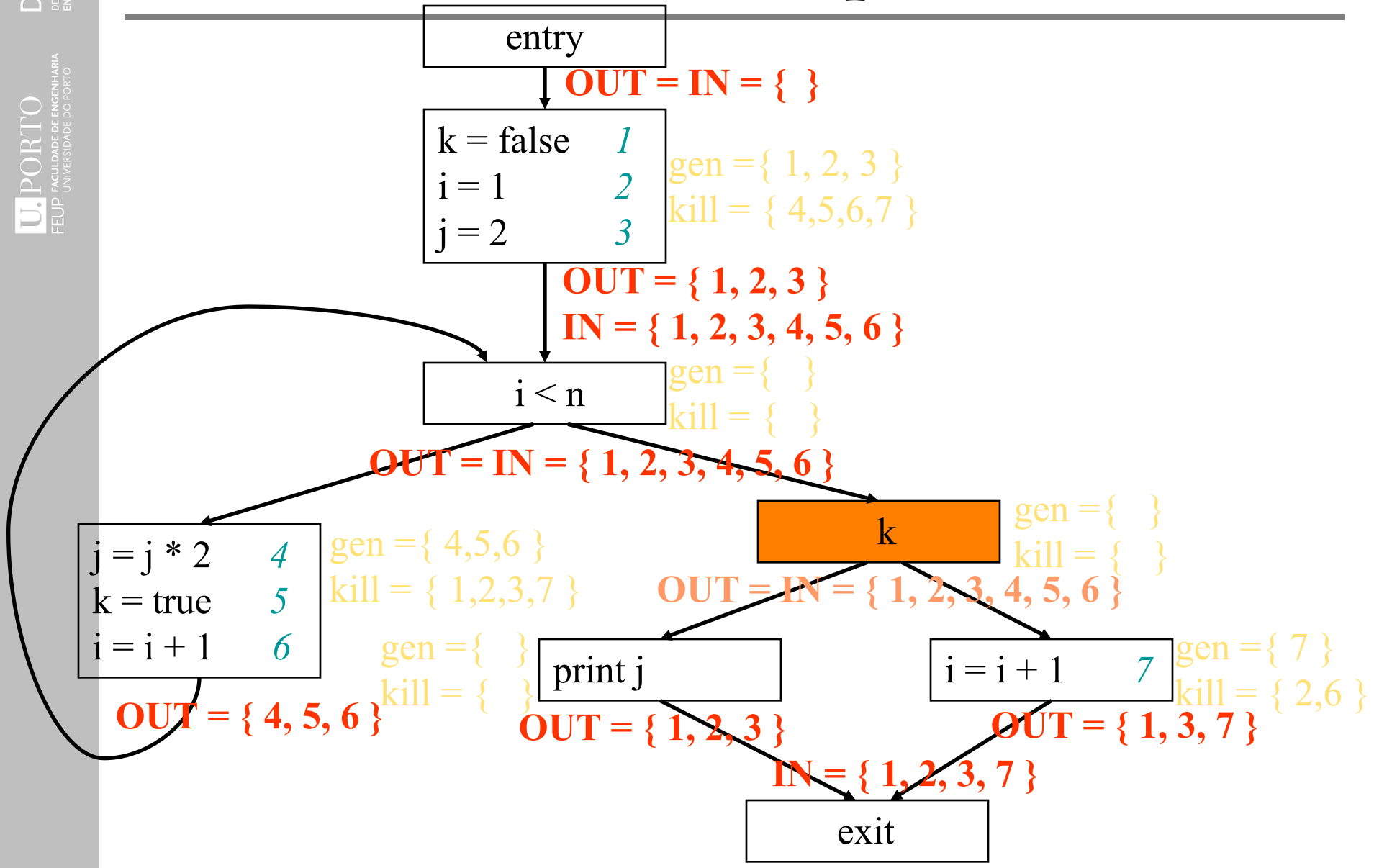
DU Example


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

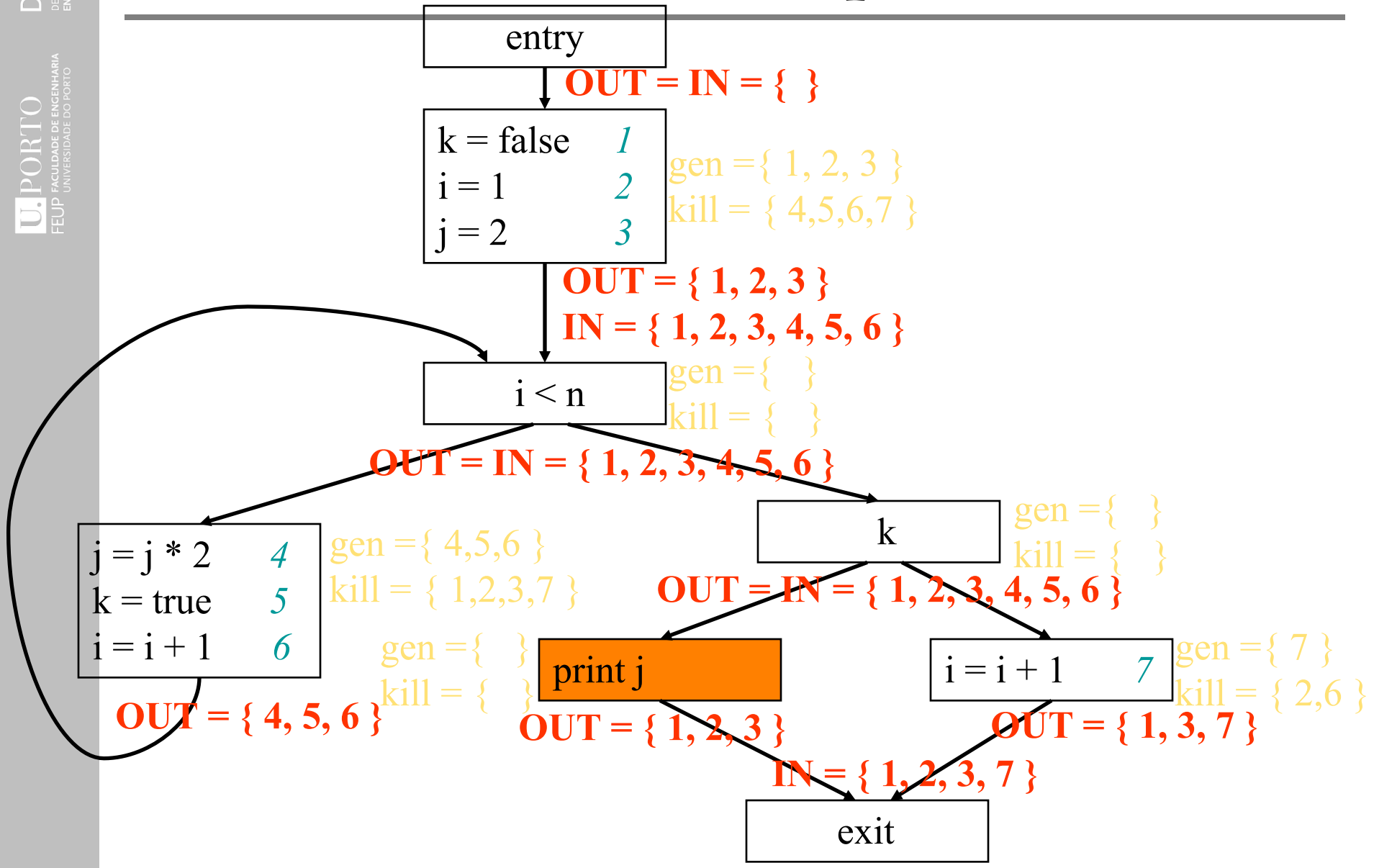
DU Example


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

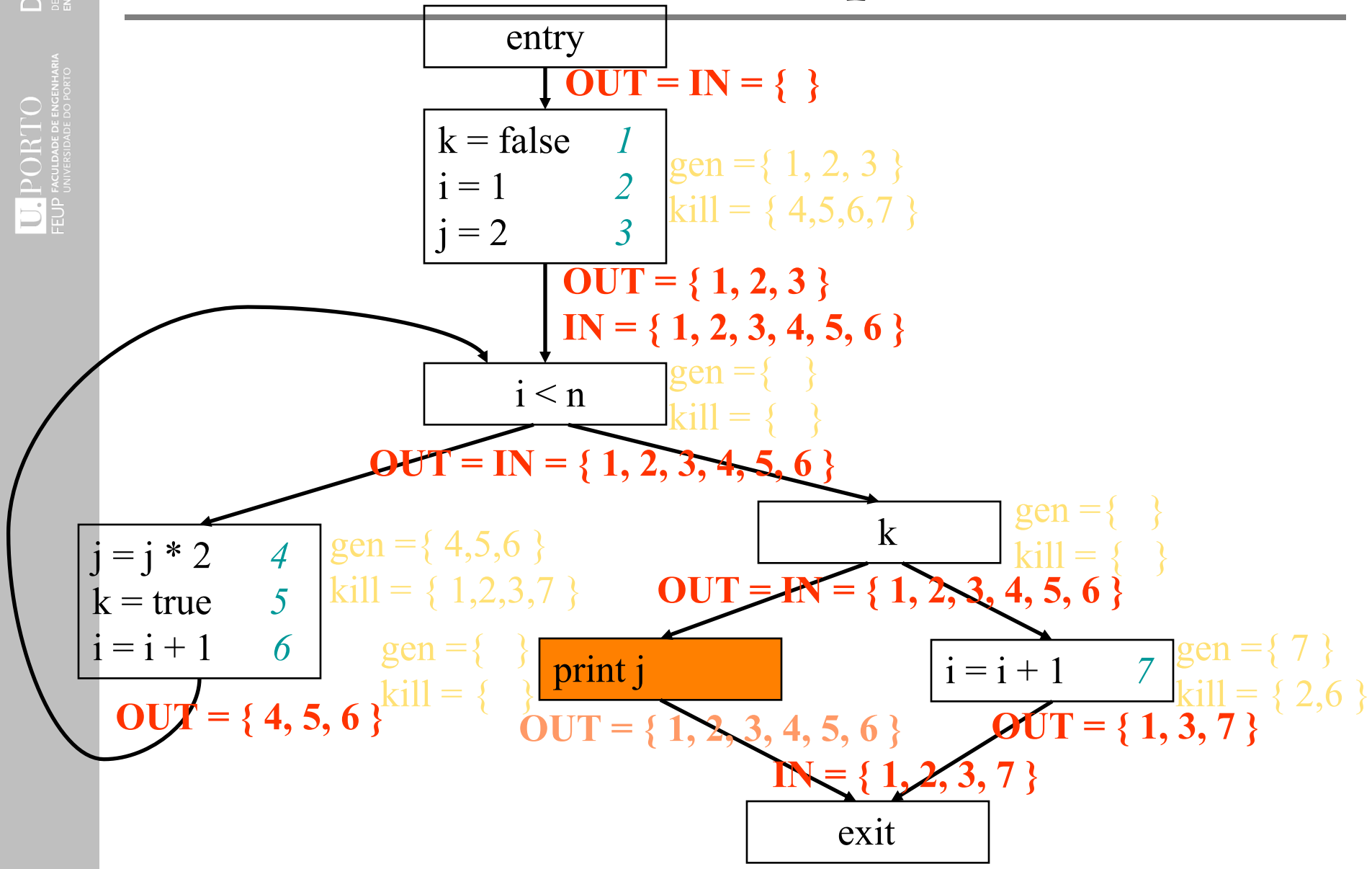
DU Example


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

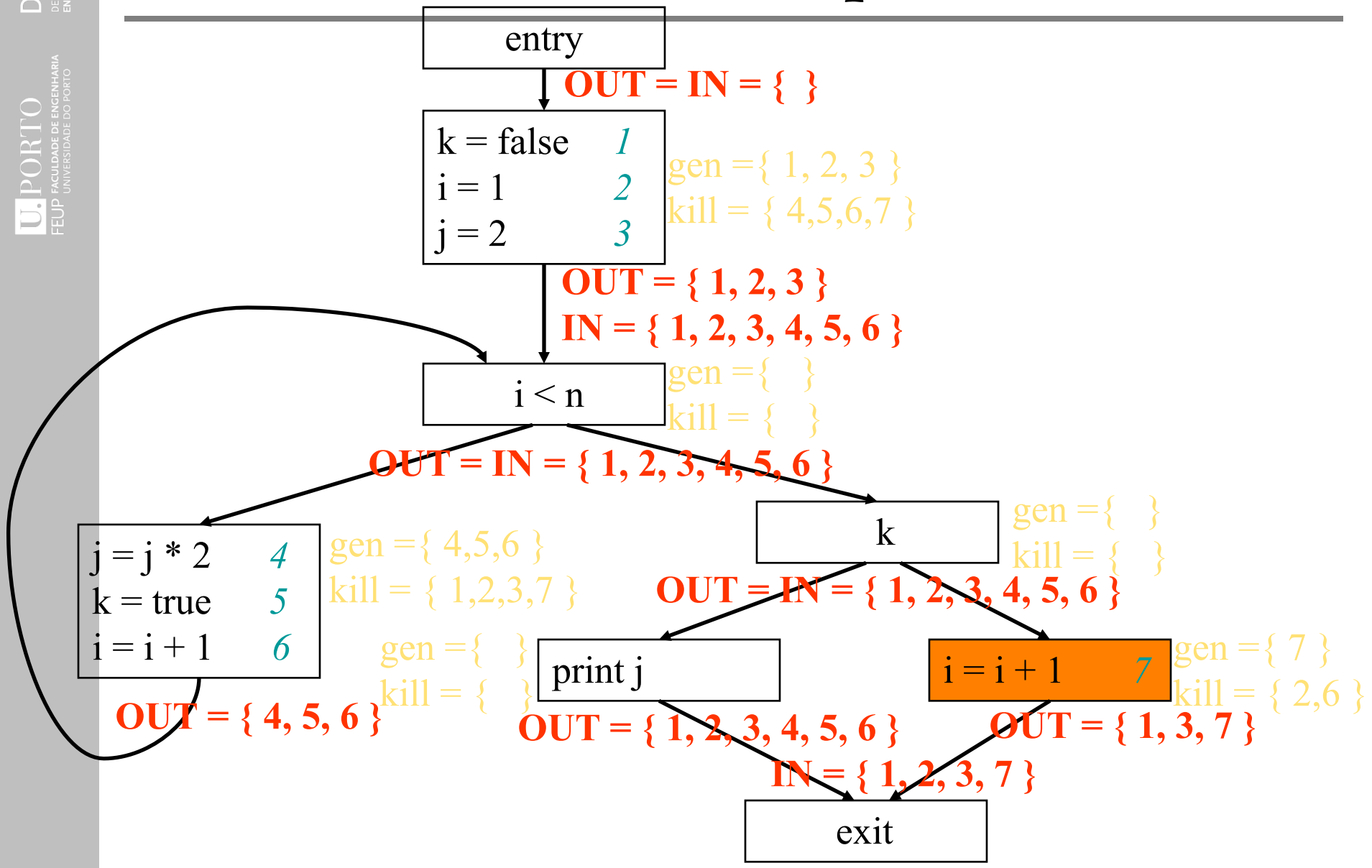
DU Example


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

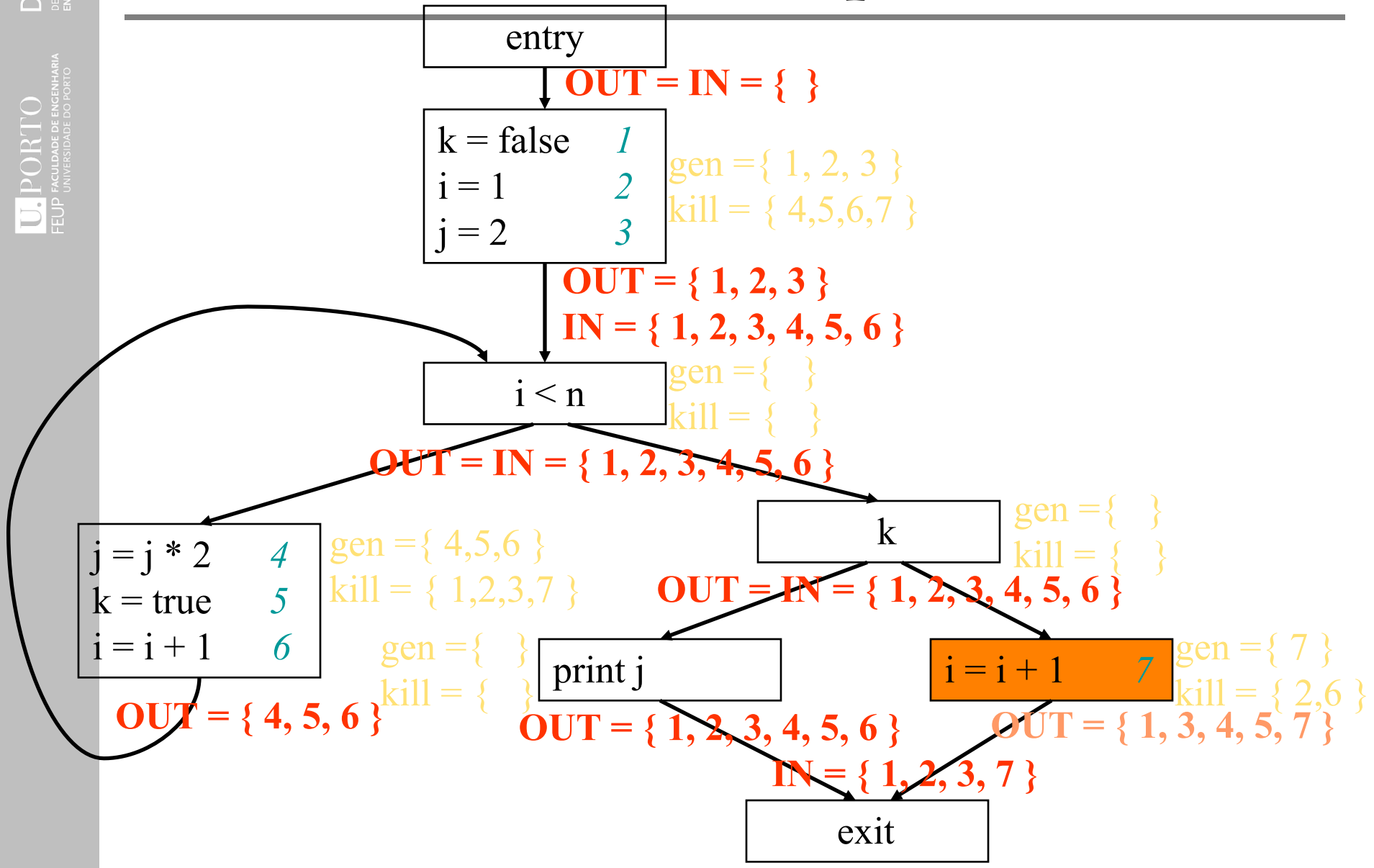
DU Example


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 }
gen ={ }
kill = { }
print j
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

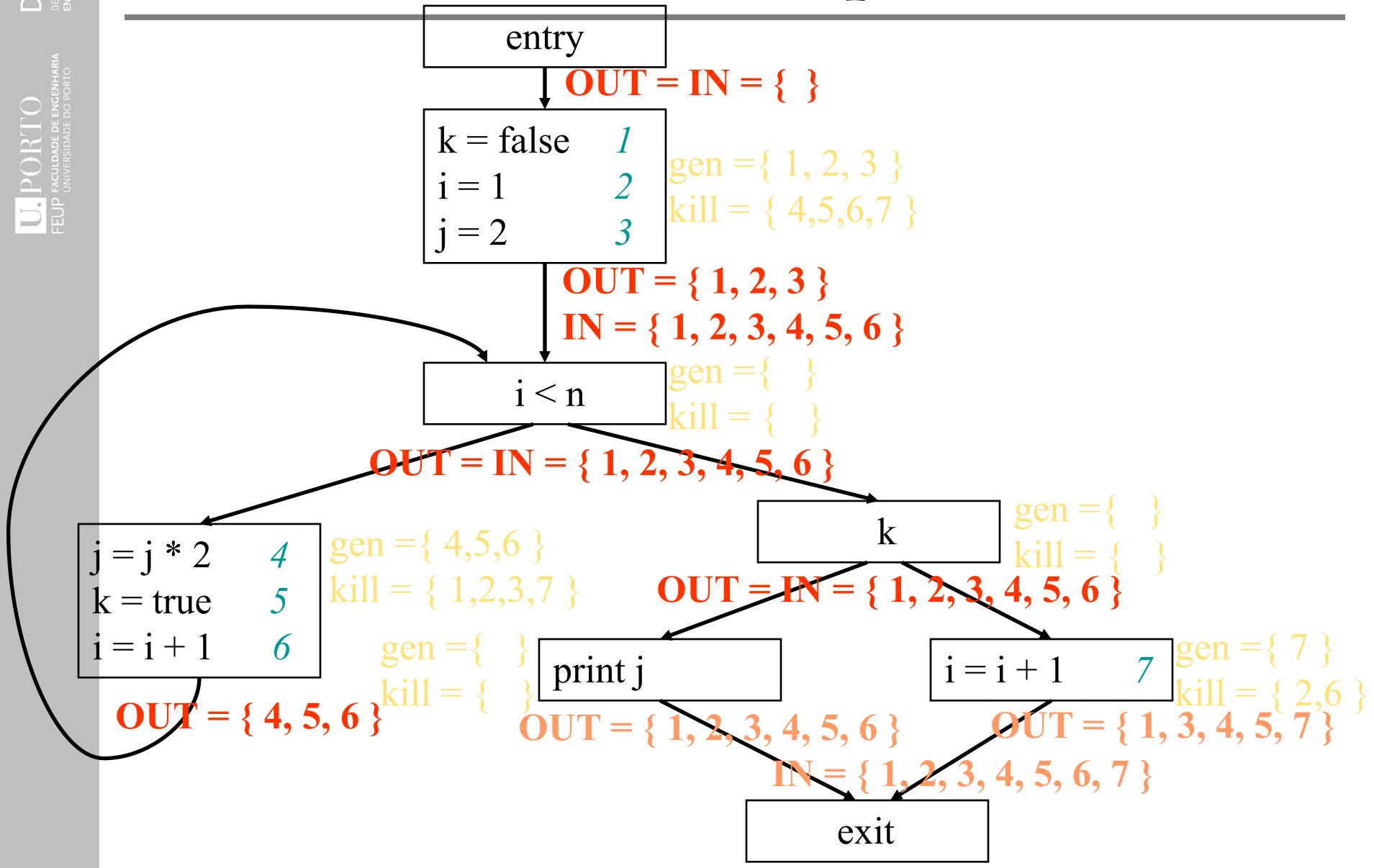
DU Example


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

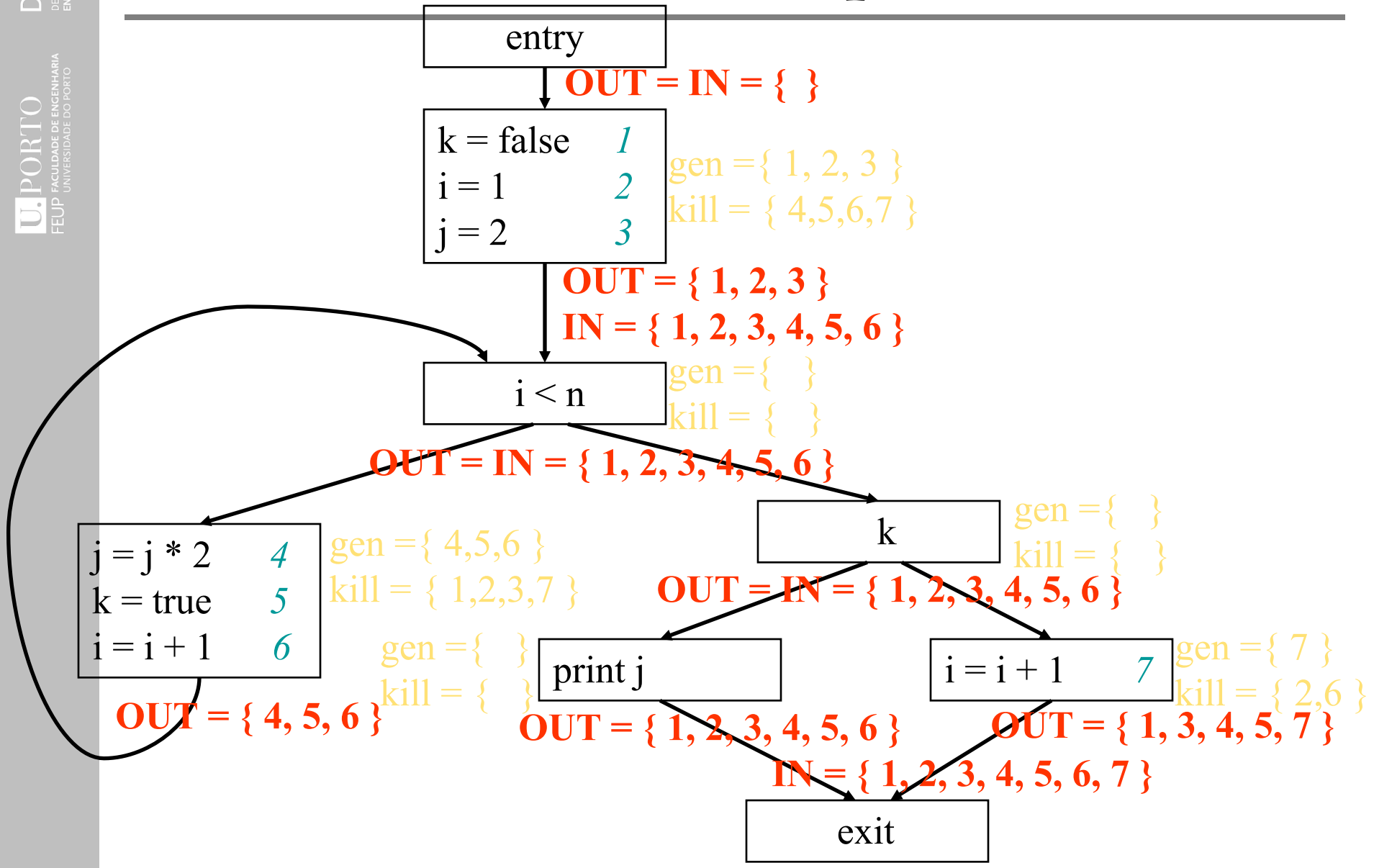
DU Example


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

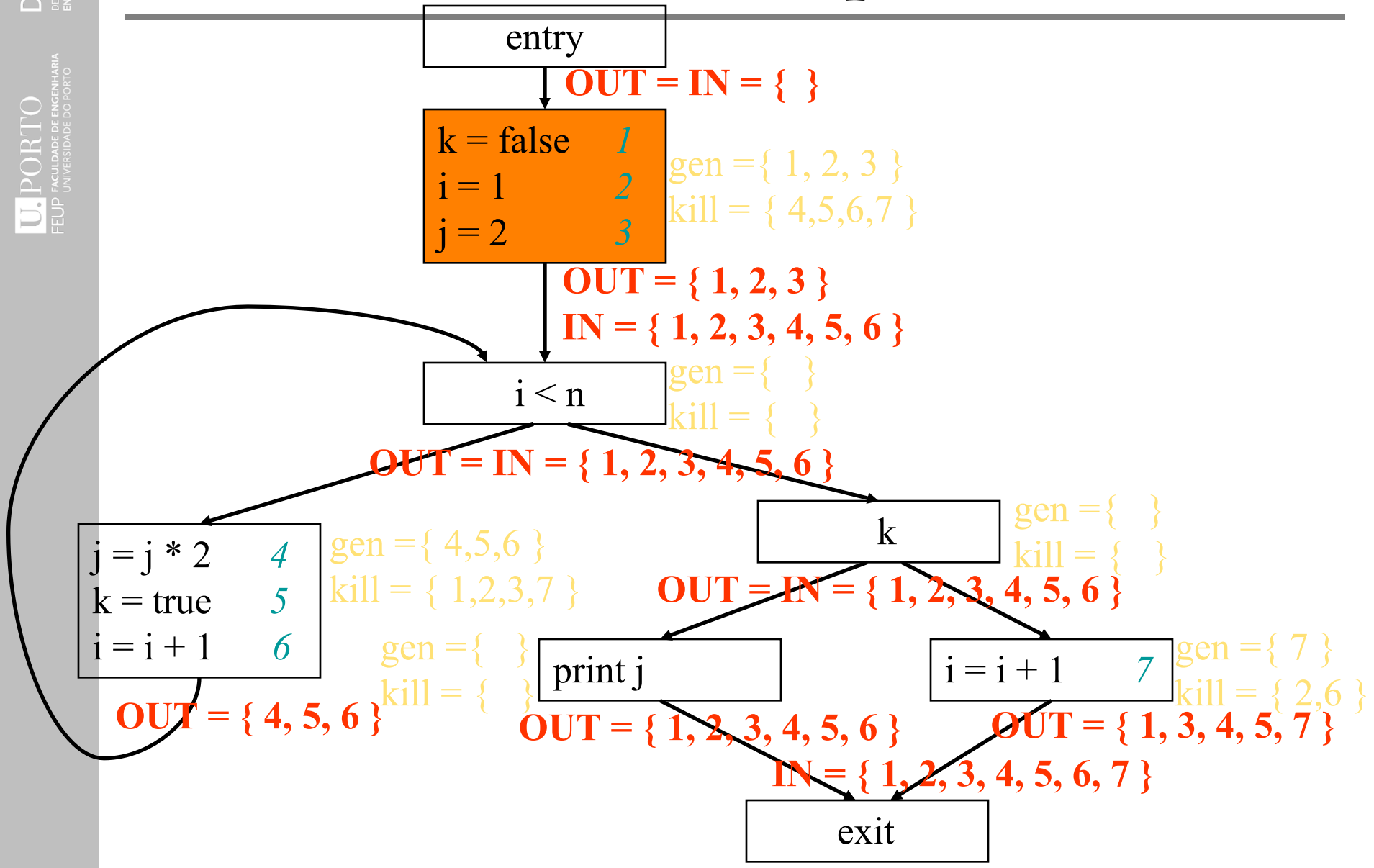
DU Example


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

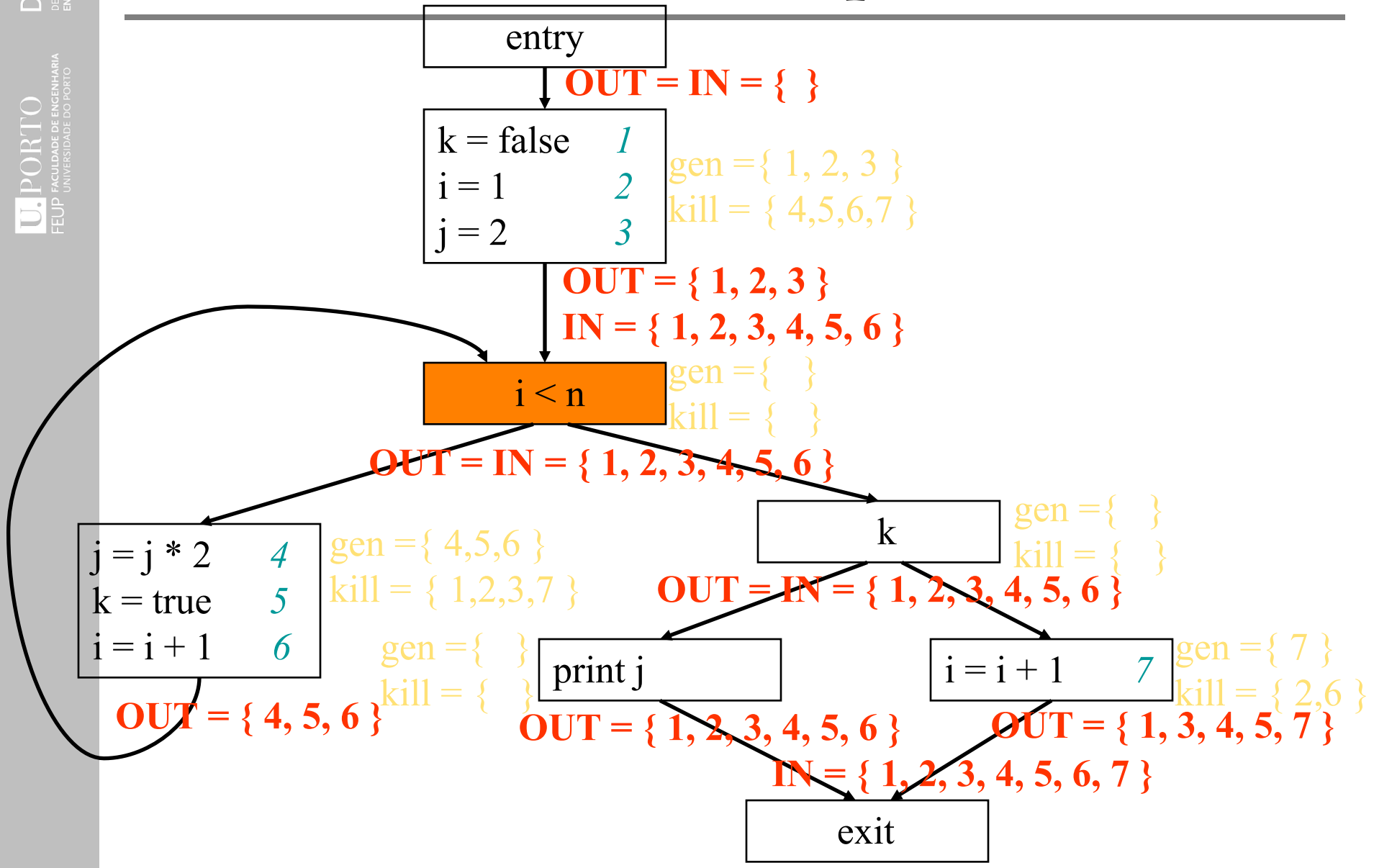
DU Example


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

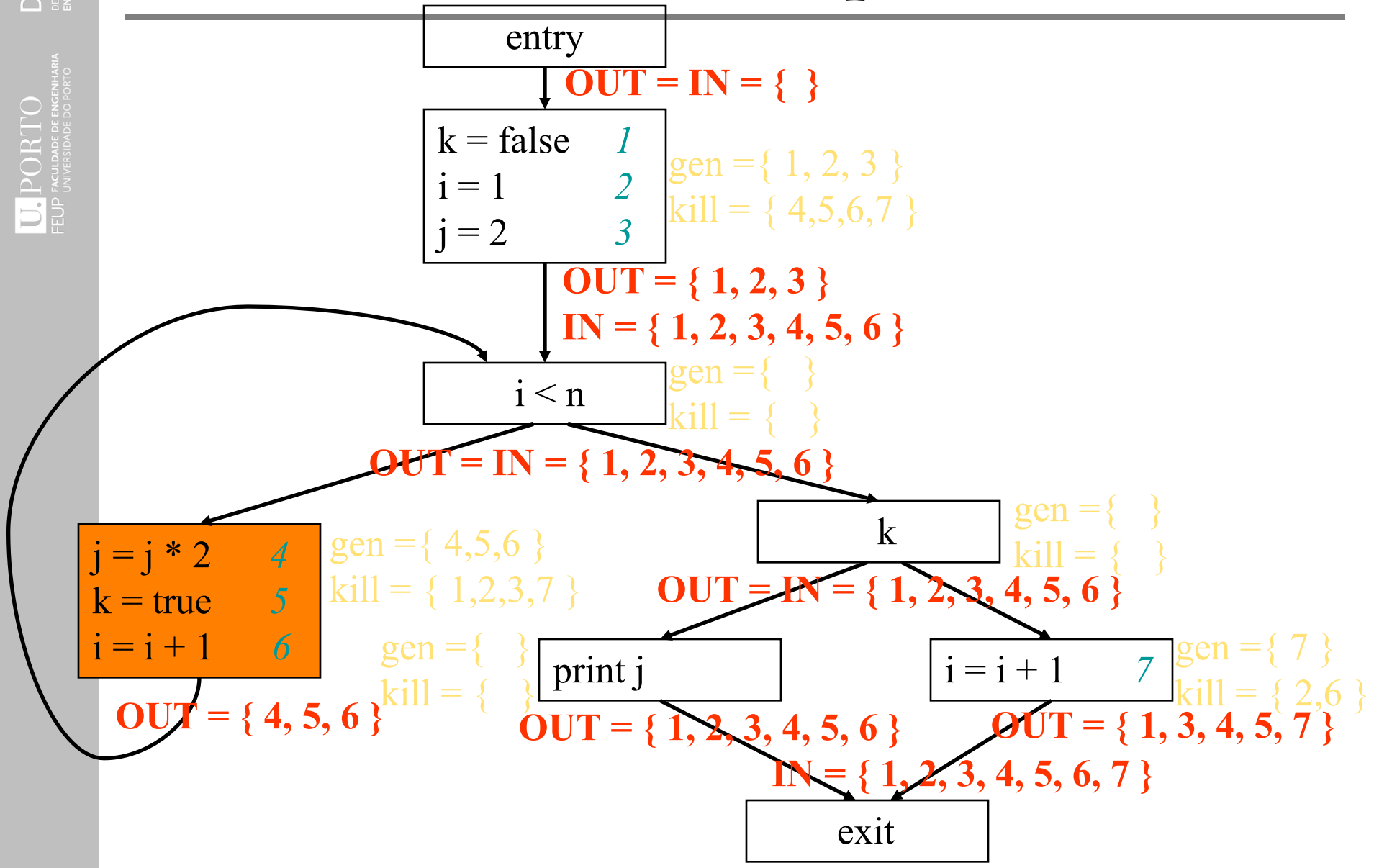
DU Example


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

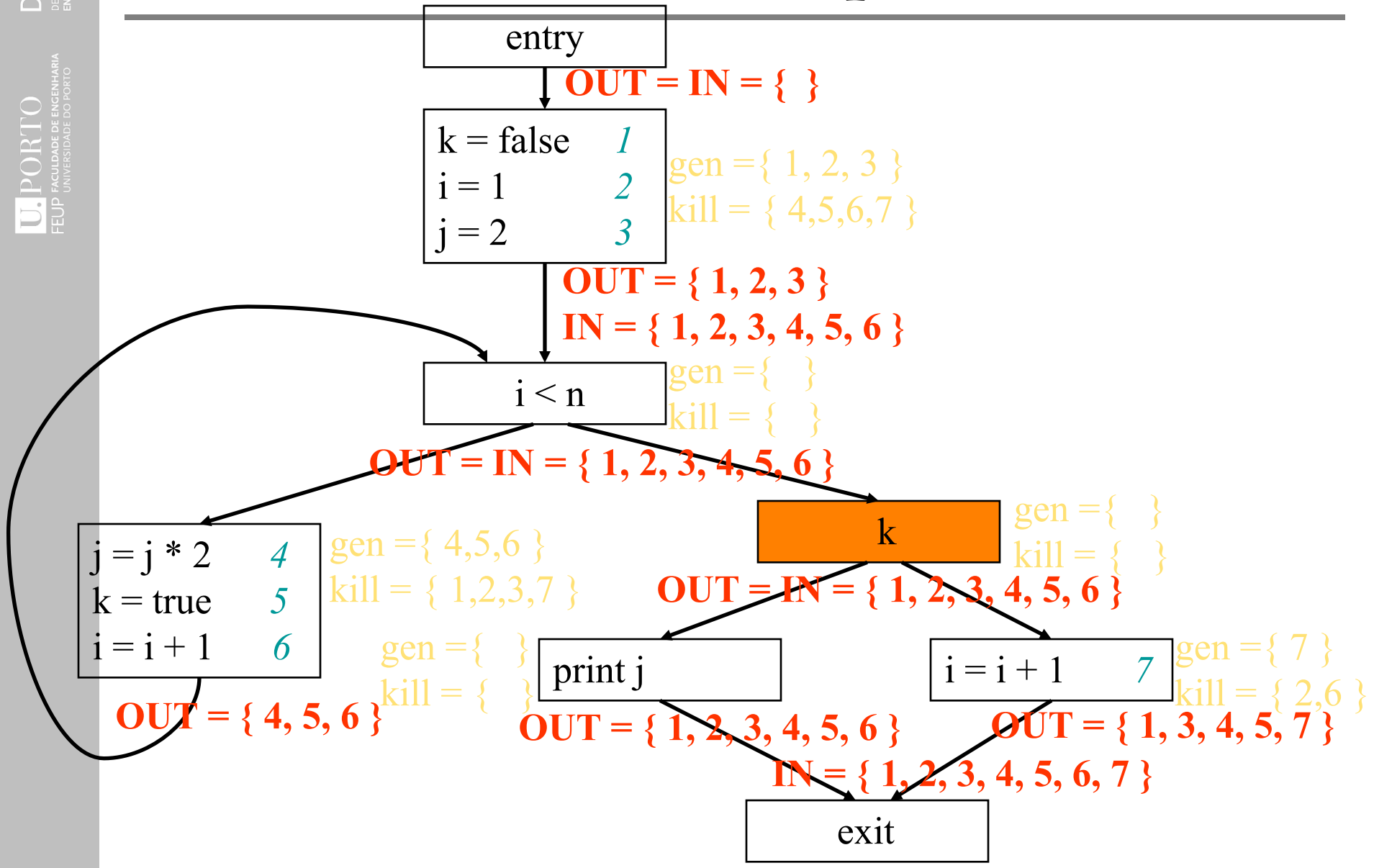
DU Example


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

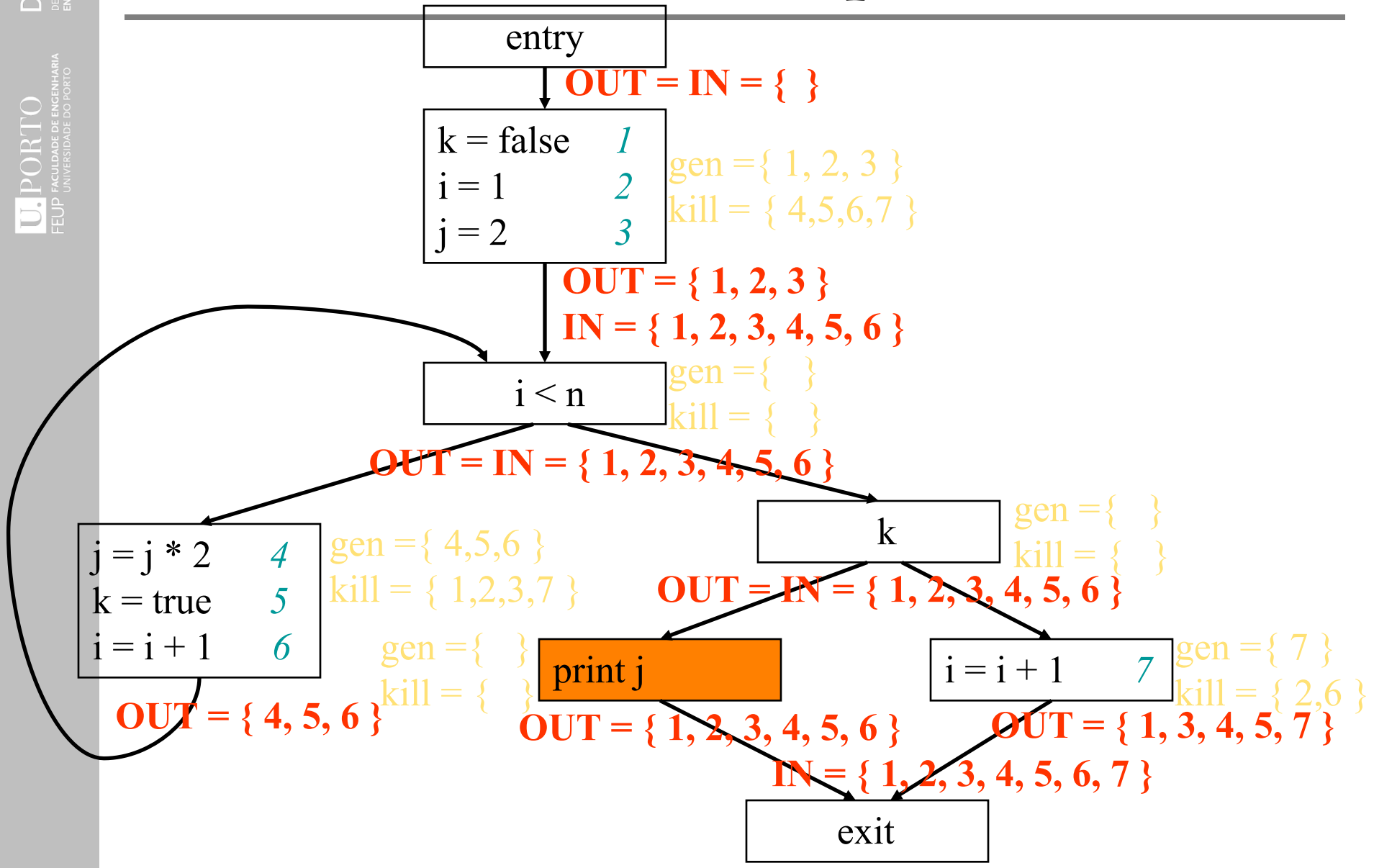
DU Example


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

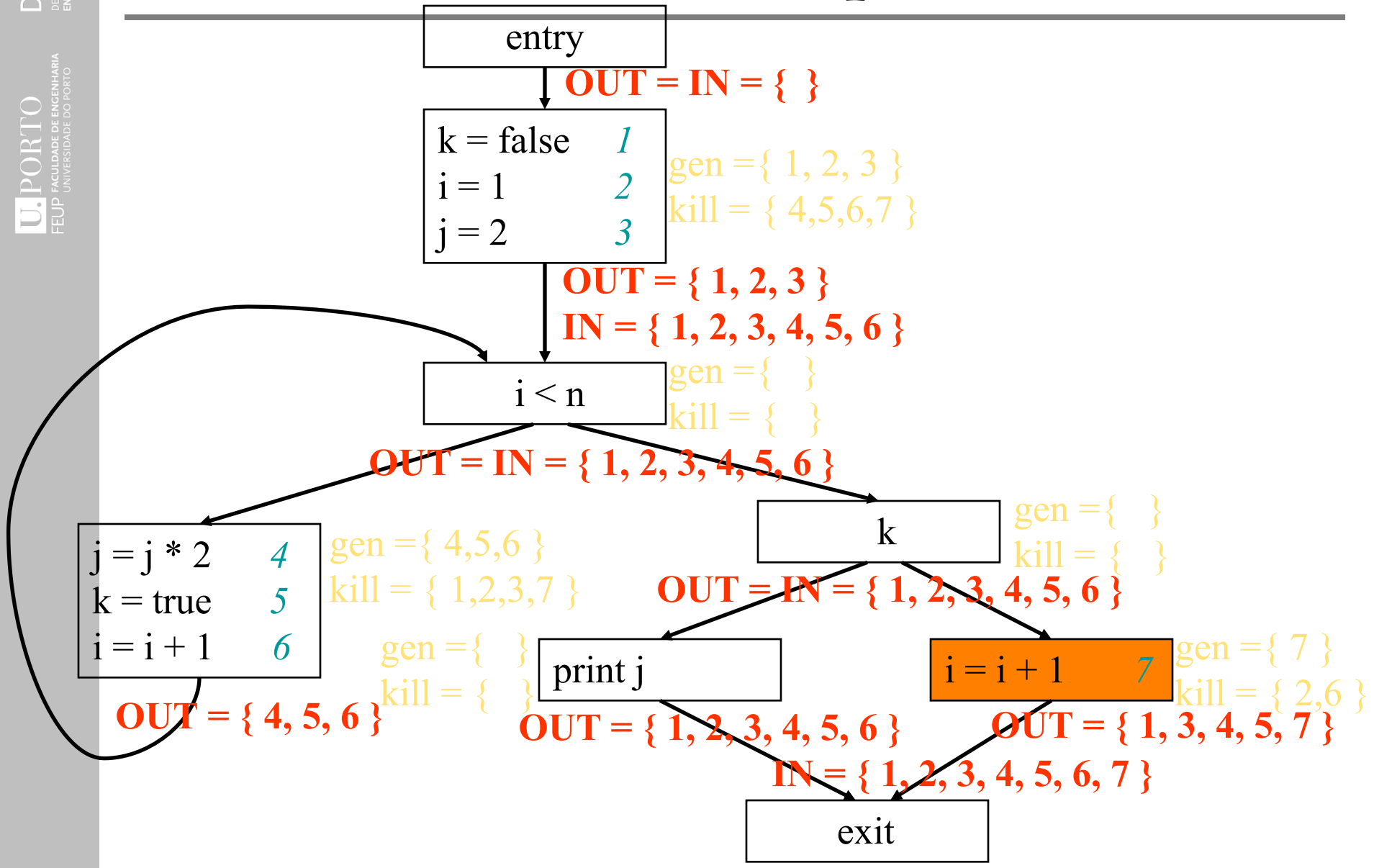
DU Example


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

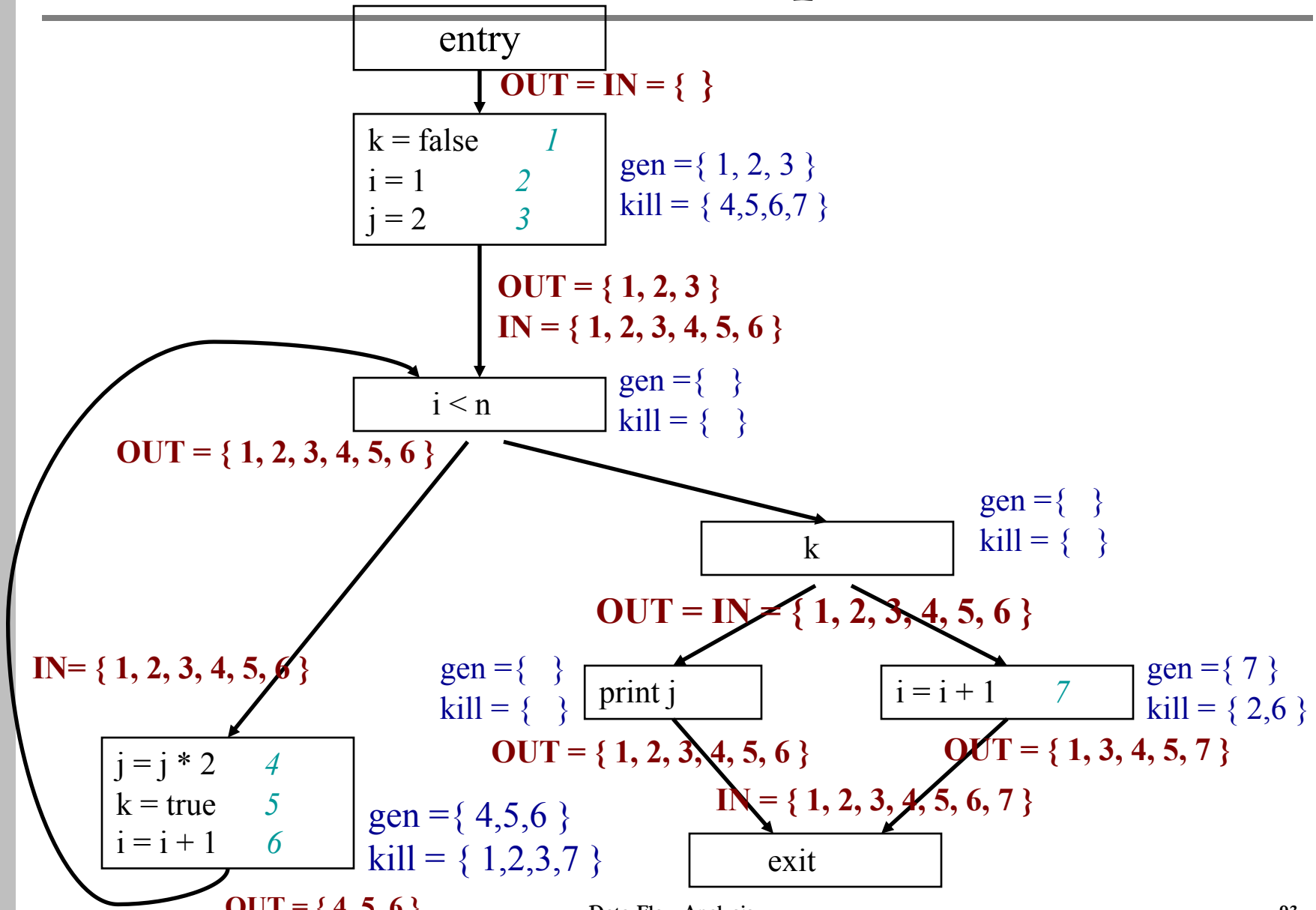
DU Example


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

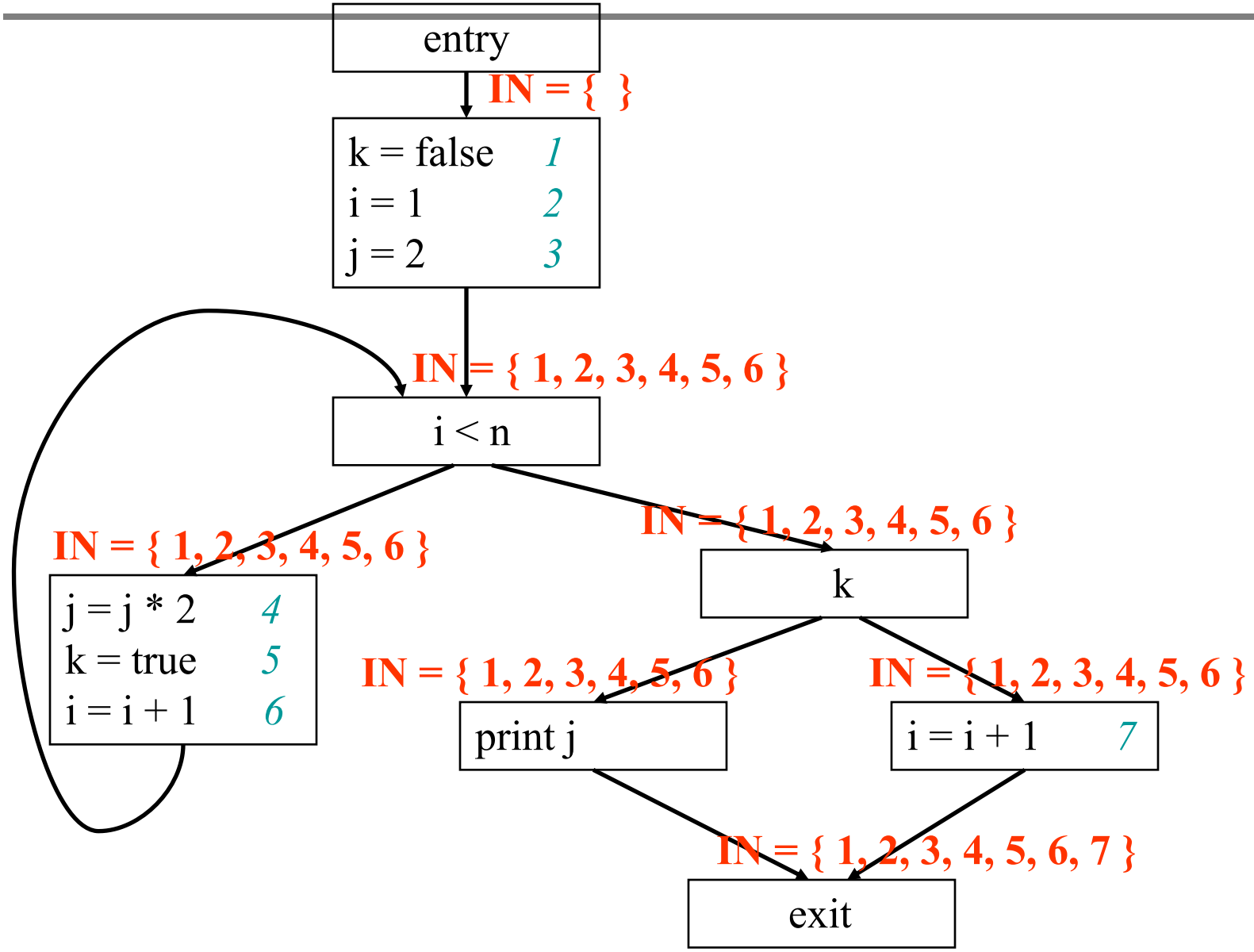
DU Example


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

- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form

Static Single Assignment (SSA) Form

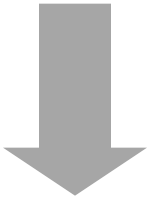
- Each definition has a unique variable name
  - Original name + a version number
- Each variable use refers to a unique name definition
- What about multiple possible definitions?
  - Add special merge nodes so that there can be only a single definition (ϕ functions)

Static Single Assignment (SSA) Form

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

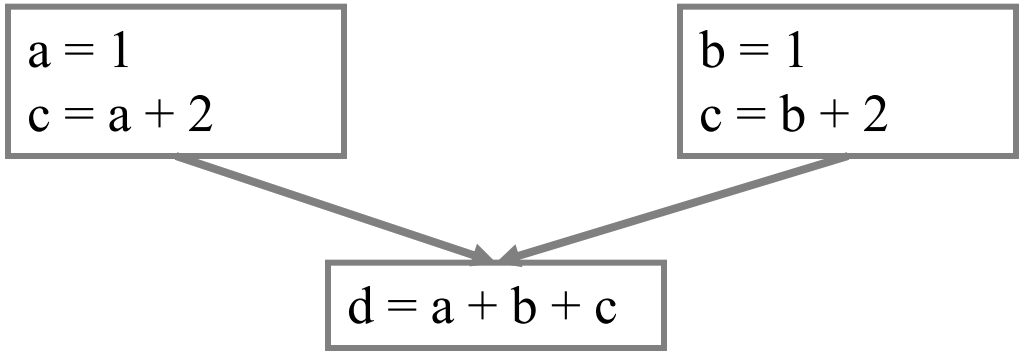
Static Single Assignment (SSA) Form

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

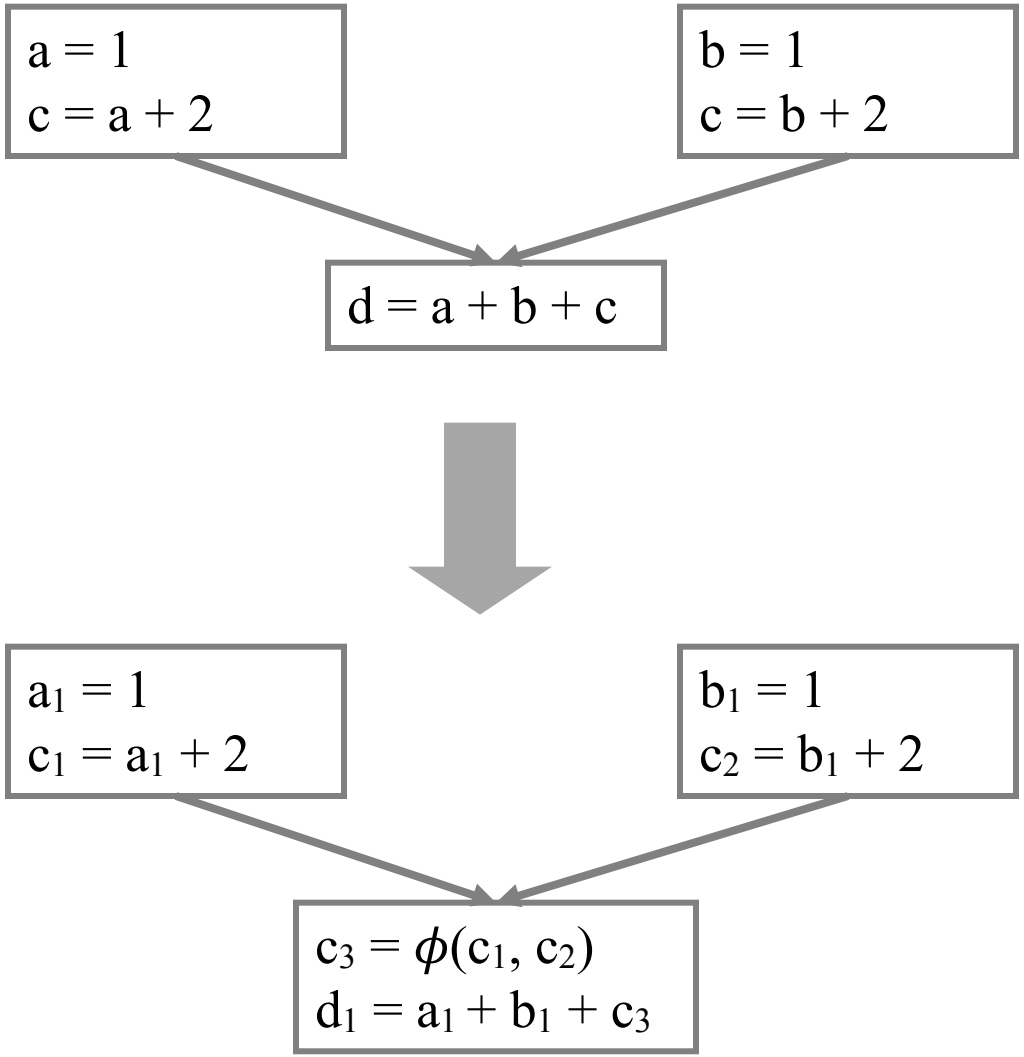


```
a1 = 1
b1 = a1 + 2
c1 = a1 + b1
a2 = a1 + 1
d1 = a2 + b1
```

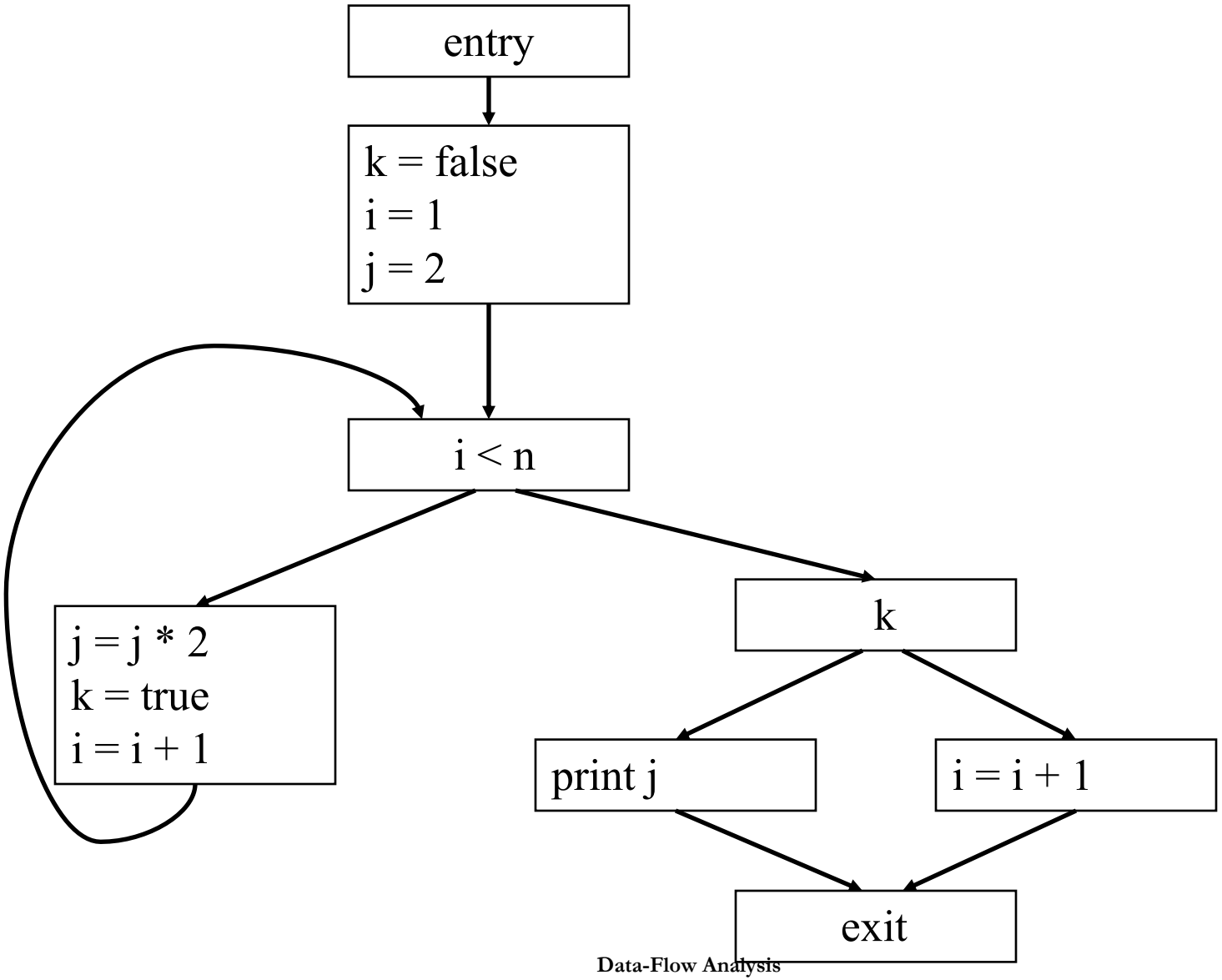
Static Single Assignment (SSA) Form



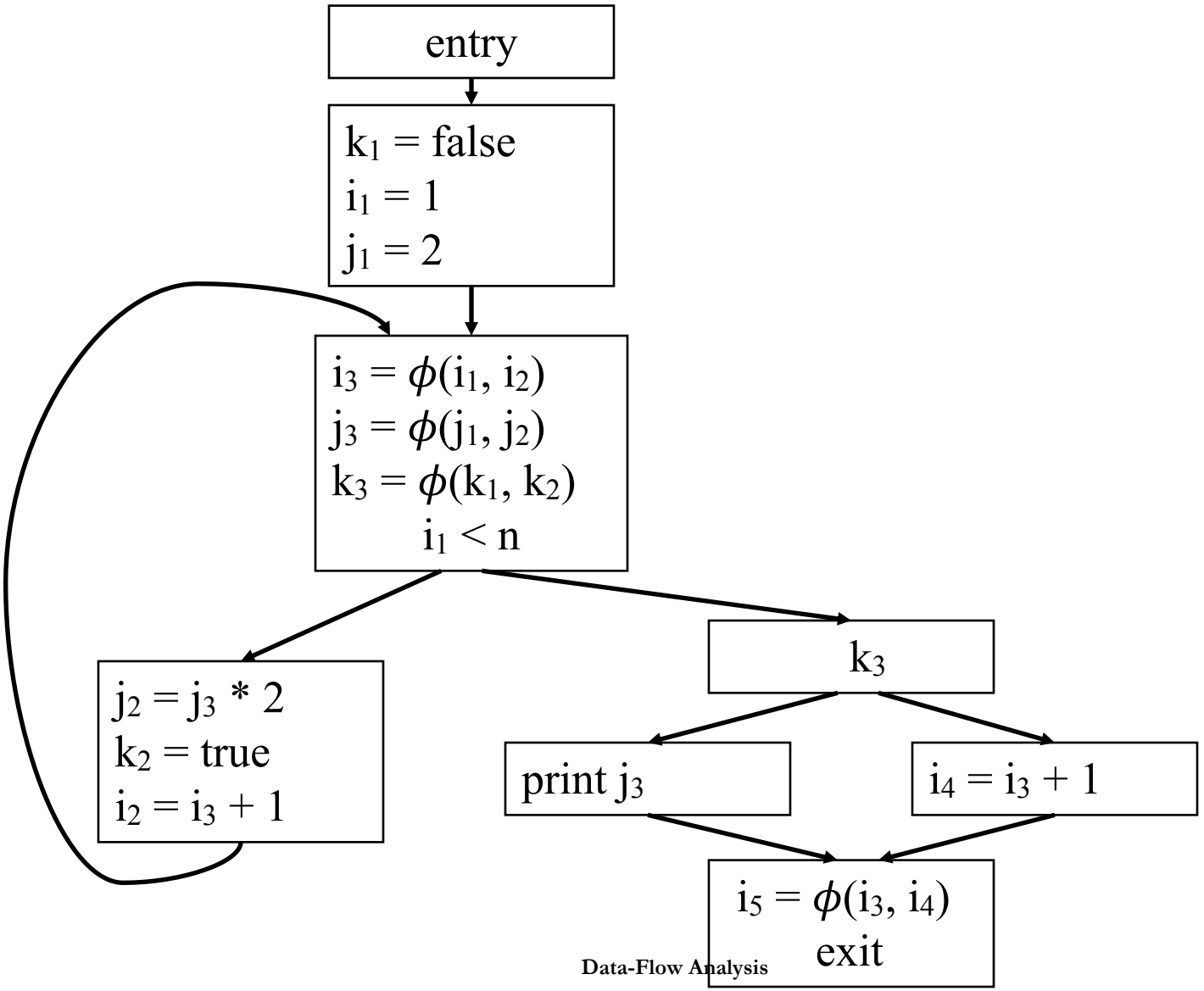
Static Single Assignment (SSA) Form



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

Static Single Assignment (SSA) Form

- What's the Point?
  - Simplifies many analyses and "optimizations"
  - Used in virtually any compiler, e.g., gcc, LLVM, Jikes, Mozilla's IonMonkey, Ocelot, etc

- Algorithm for Placement of ϕ nodes?
  - DU Chains or Reaching Definition
  - Dominators (Dominance Frontiers)

Summary

- Formulating a Data-Flow Analysis Problem
- DU Chains
- SSA Form