CS 575
Software Testing and Analysis

Control Flow Data Analysis

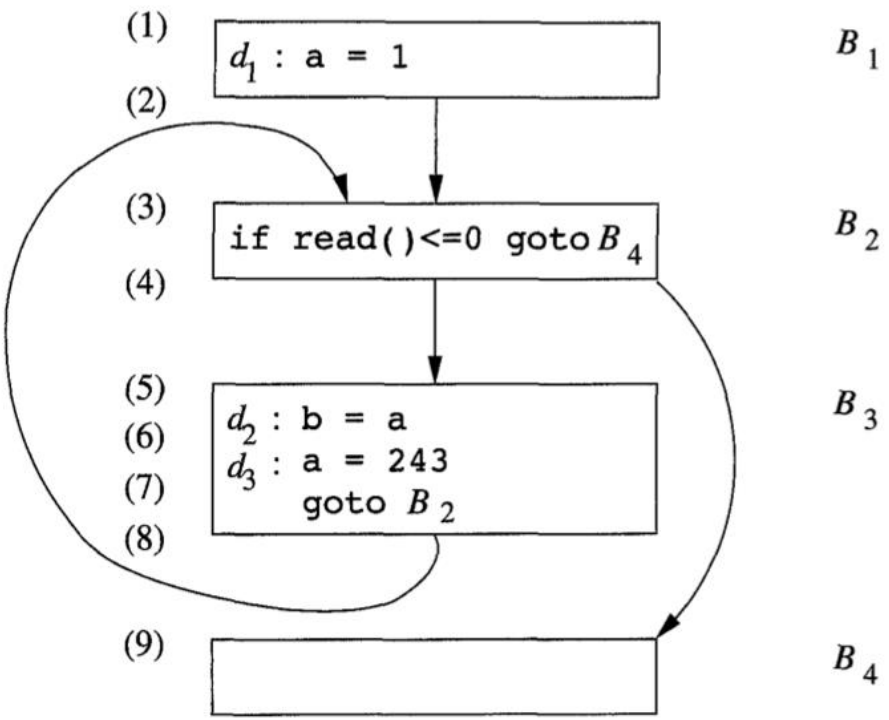
(c) Slides patially adopted from the book/slides of

- P. Amman & J. Offut and of M. Pezze and M. Young
- Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman
- K.D. Cooper and L. Torczon
- B. Aktemur

Software Testing and **Analysis**

- Revealing bugs
  - Null pointers, memory leaks, uninitialized variables, race conditions, buffer overflows..
- Code Optimization
- Defining Coverage Criteria
  - Based on data dependence:
  - Where does this value of x come from?
  - What would be affected by changing this?
  - ...

Data Flow Analysis: Example



- The first time program point (5) is executed, the value of *a* is 1 due to definition d1.
- In subsequent iterations, d3 reaches point (5) and the value of *a* is 243.
- At point (5), the value of *a* is one of {1,243}.
- It may be defined by one of {d1,d3}.

Def-Use Pairs

- A **def-use (du) pair** associates
  - a point in a program where a value is produced with
  - a point where it is used

- Extensively used for dataflow analysis in compilers
  - i.e., reaching definitions

Def (Definition)

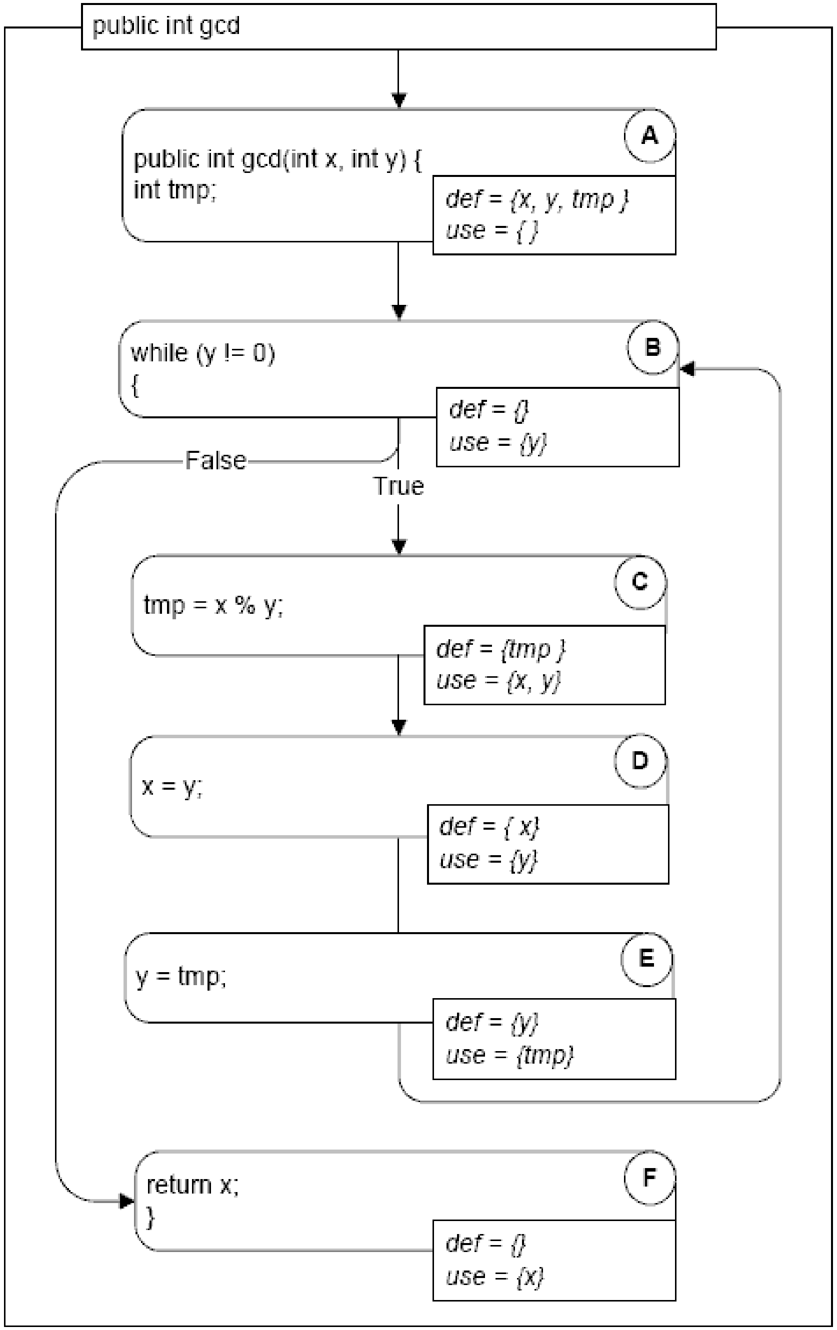
- Where a variable gets a value
  - Variable declaration (often the special value "uninitialized")
  - Variable initialization
  - Assignment
  - Values received by a parameter

Use

- Extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

Def-Use Sets: Example

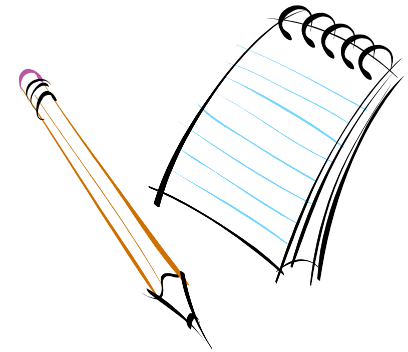
```
/**  Euclid's algorithm */
public class GCD
{
public int gcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {      // B: use y
        tmp = x % y;      // C: def tmp; use x, y
        x = y;            // D: def x; use y
        y = tmp;          // E: def y; use tmp
    }
    return x;             // F: use x
  }
```



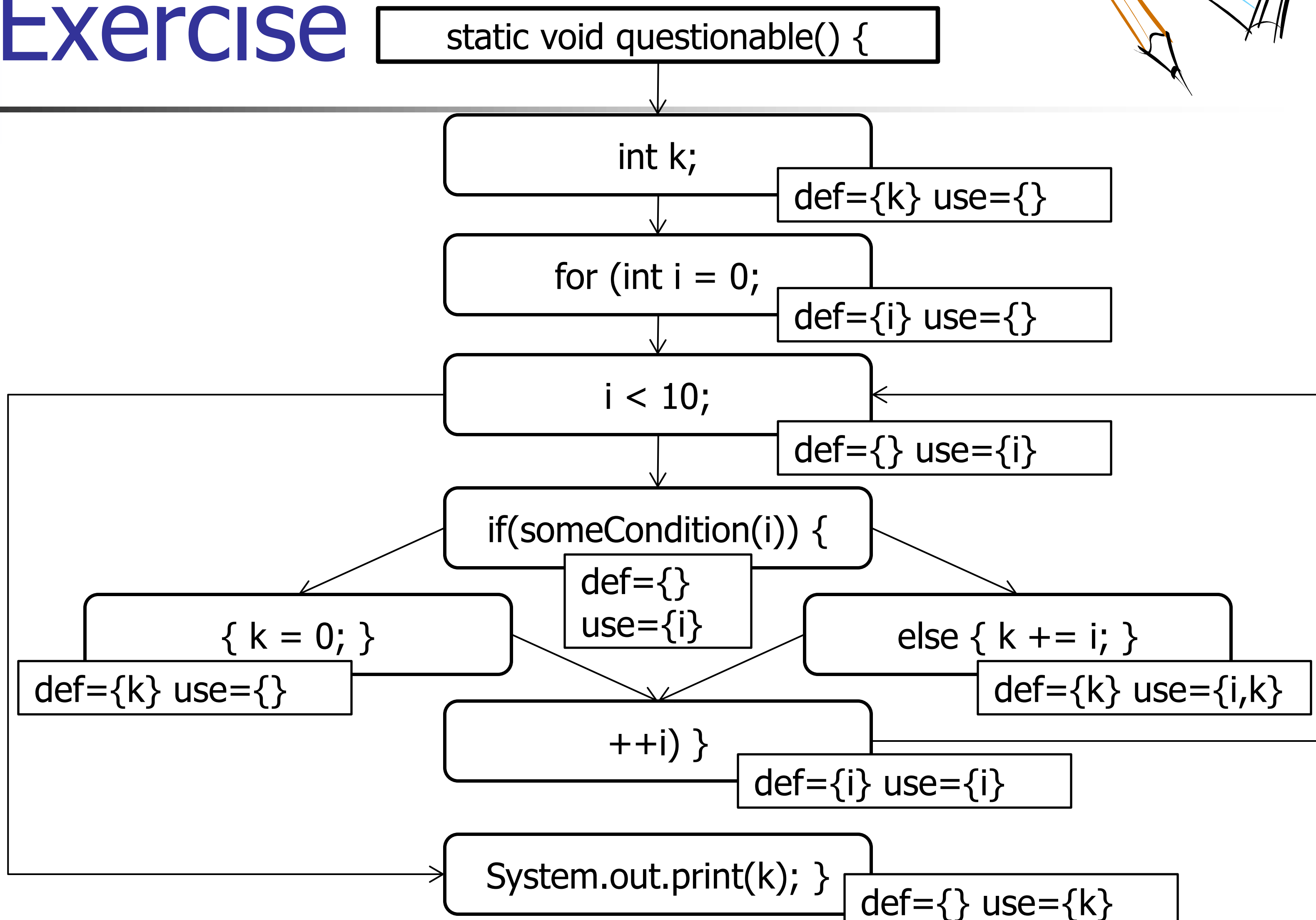
Exercise

- Draw the control flow graph for the method
- Annotate the nodes with *def* and *use* sets

```
static void questionable() {
  int k;
  for (int i = 0; i < 10; ++i) {
    if (someCondition(i)) {
      k = 0;
    } else {
      k += i;
    }
  }
  System.out.println(k);
}
```



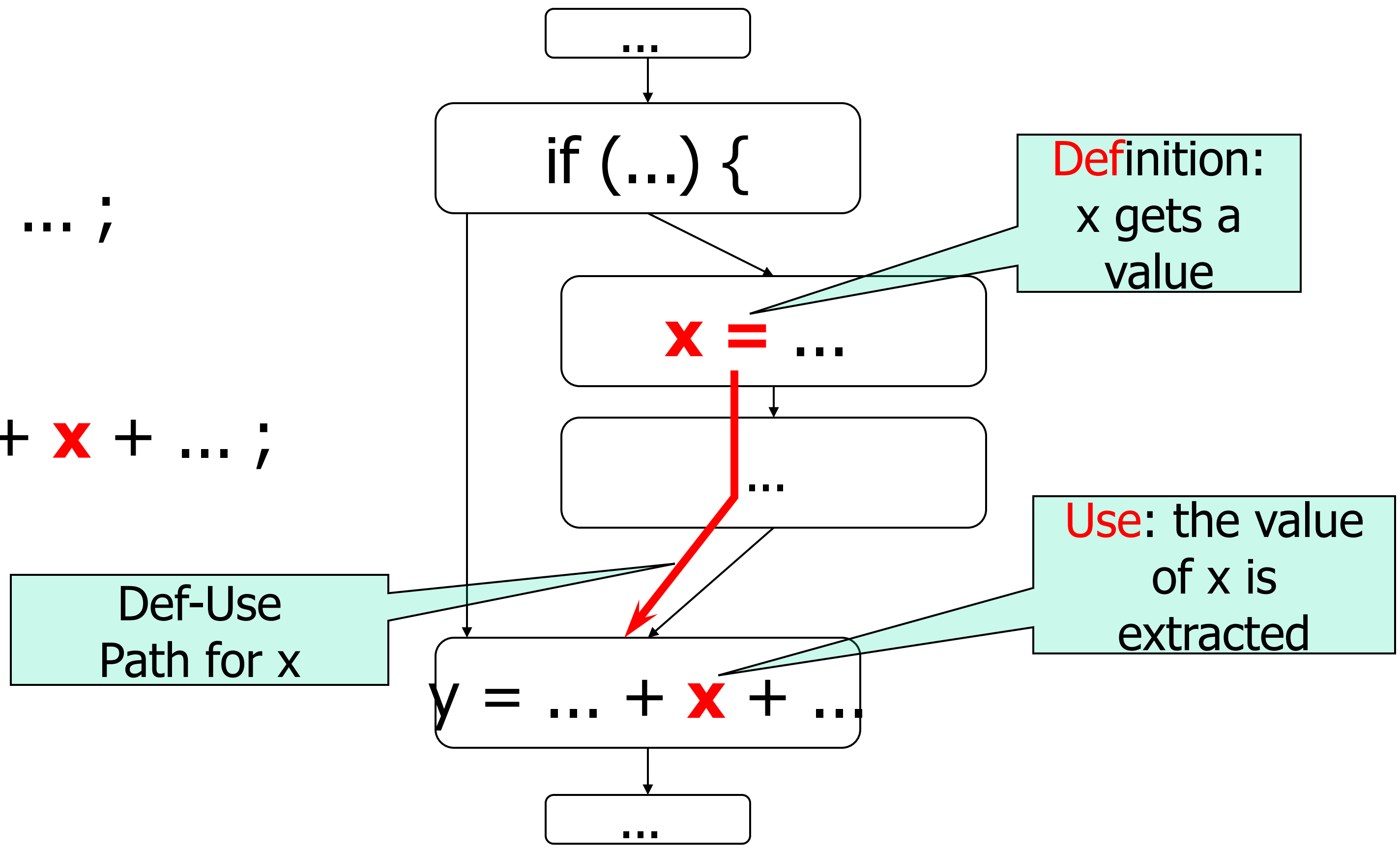
Exercise



static void questionable() {
int k;
def={k} use={}
for (int i = 0;
def={i} use={}
i < 10;
def={} use={i}
if(someCondition(i)) {
def={}
use={i}
{ k = 0; }
def={k} use={}
else { k += i; }
def={k} use={i,k}
++i) }
def={i} use={i}
System.out.print(k); }
def={} use={k}

Def-Use Pairs: Example

```
...
if (...) {
    x = ... ;
...
}
y = ... + x + ... ;
```



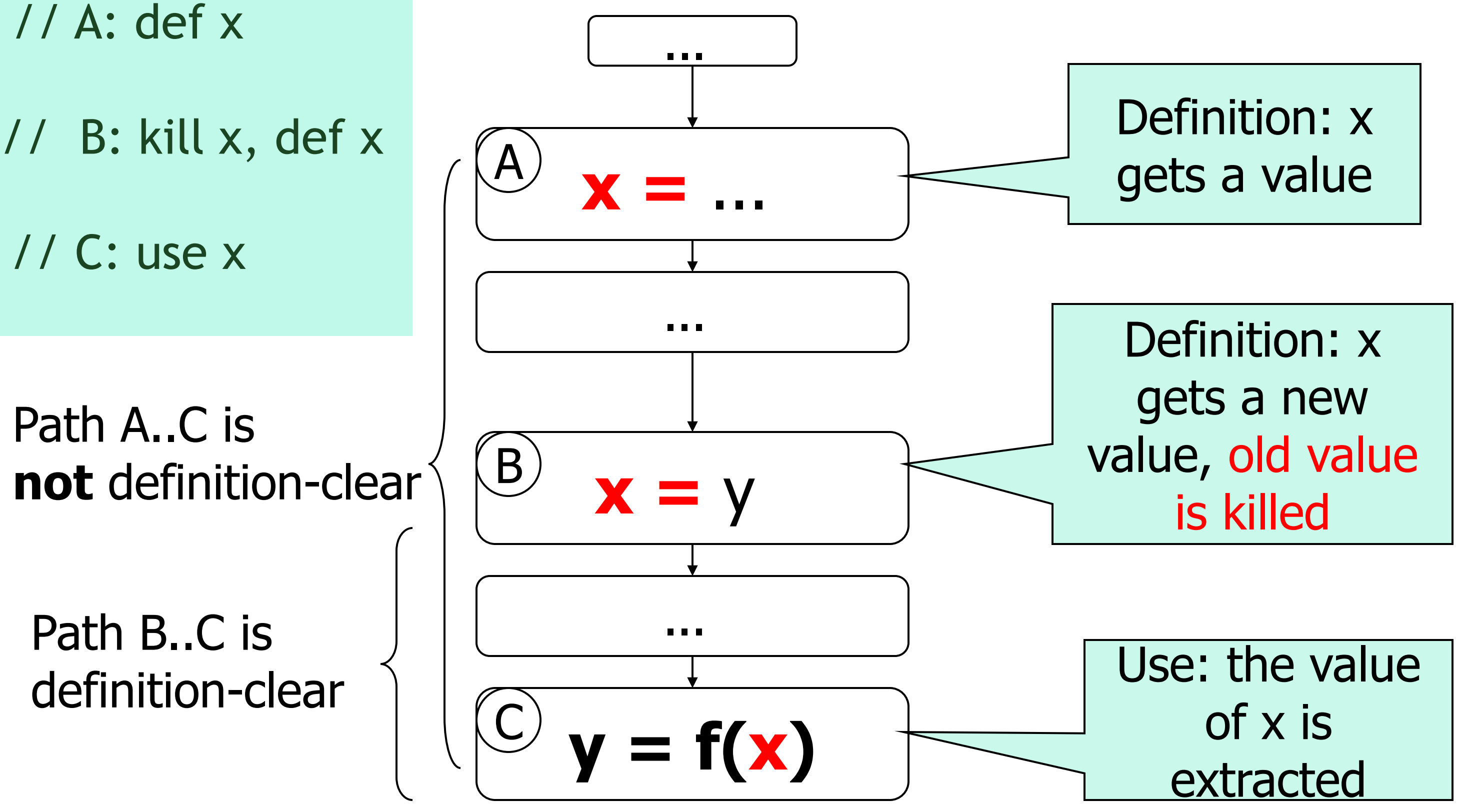
Killing (Overwriting) Definitions

- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without* another definition of the variable in-between
  - in case of any overwriting, the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

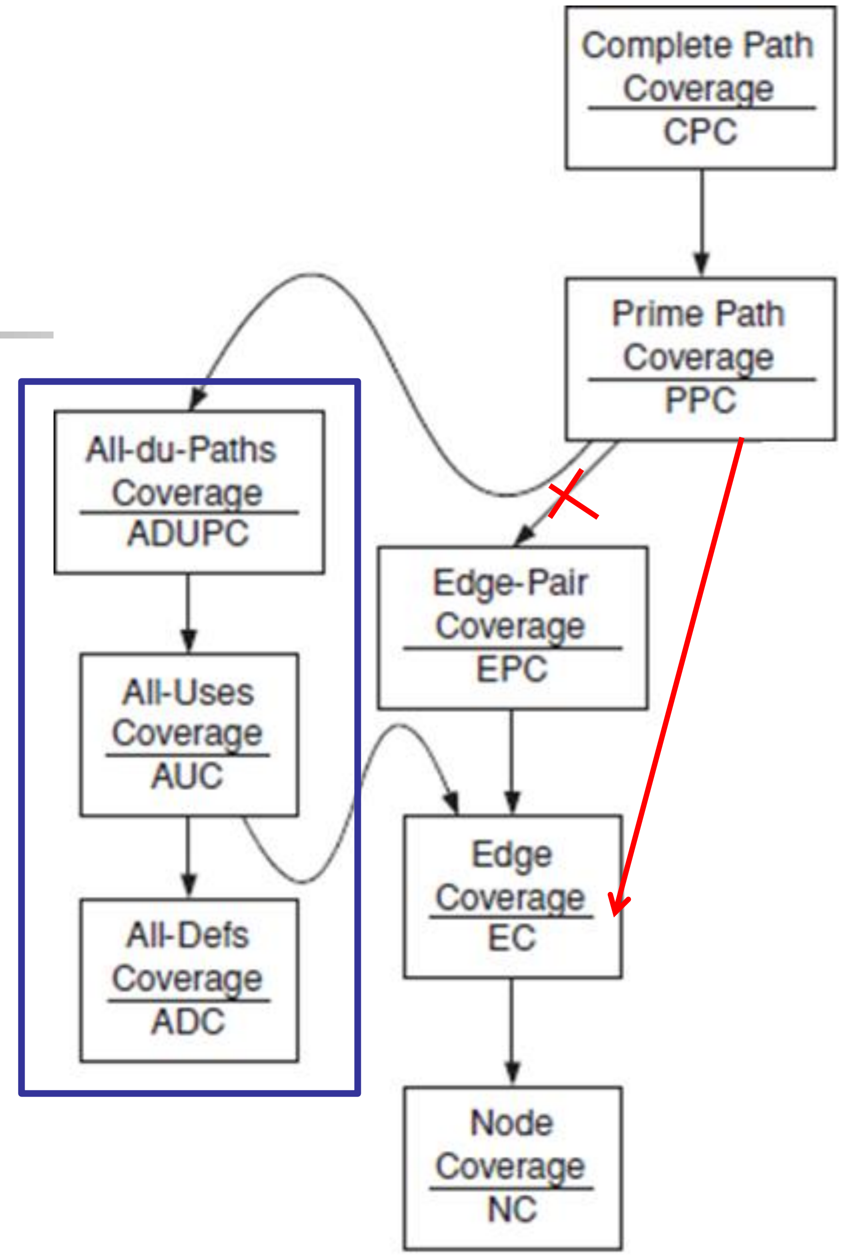
** In fact, sometimes it is impossible to know for sure whether two definitions affect the same variable or storage location.*

Killing (Overwriting) Definitions: Example

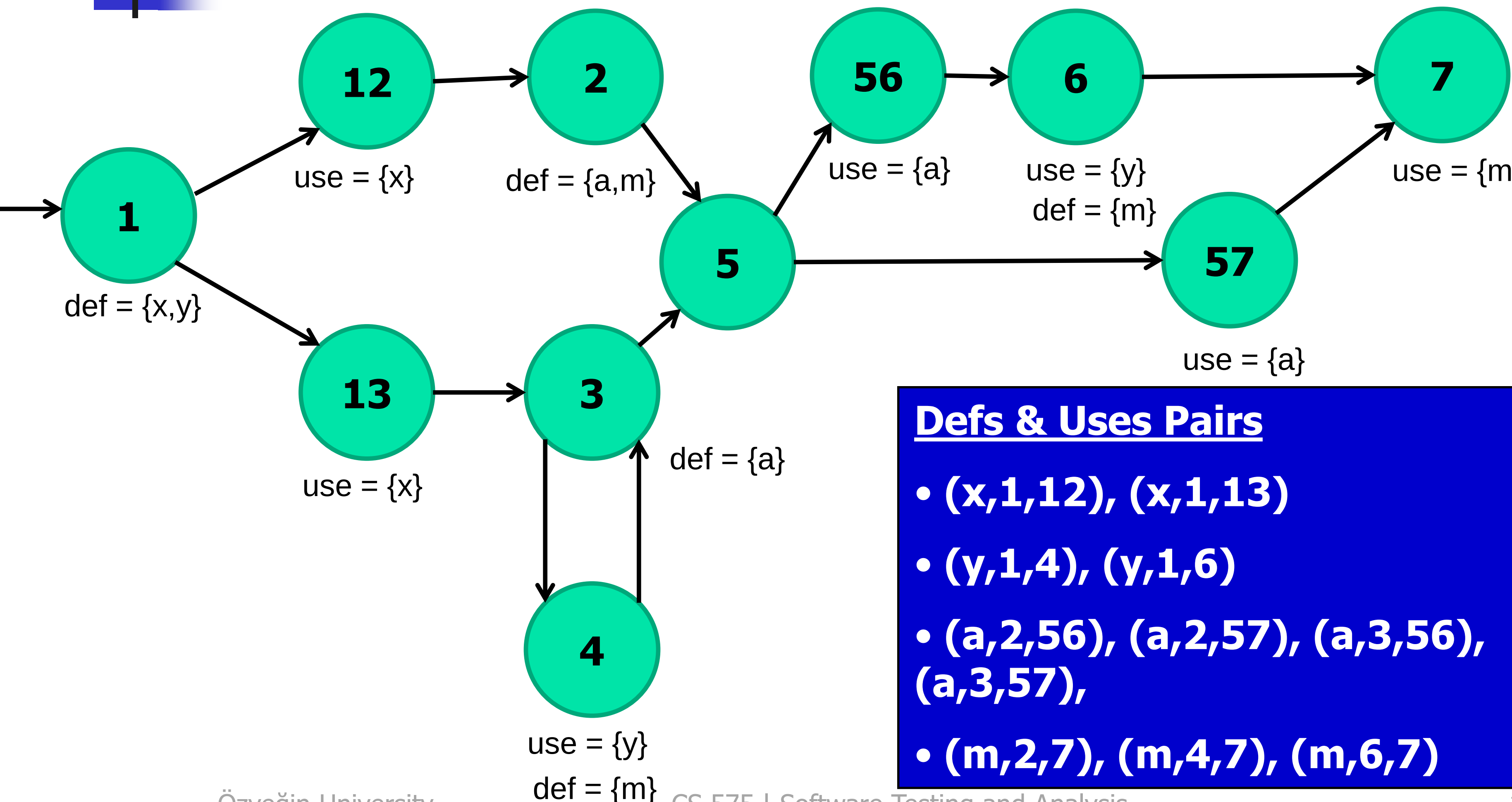
```
x = …       // A: def x
q = …
x = y;      //  B: kill x, def x
z = …
y = f(x);   // C: use x
```



Recall: Coverage Criteria Based on Data Flow



Data Flow Based Coverage Criteria: Example



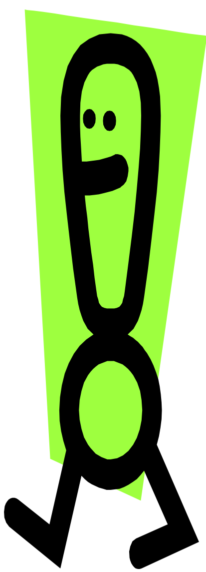
<u>Defs & Uses Pairs</u>

- **(x,1,12), (x,1,13)**
- **(y,1,4), (y,1,6)**
- **(a,2,56), (a,2,57), (a,3,56), (a,3,57),**
- **(m,2,7), (m,4,7), (m,6,7)**

Calculating def-use pairs

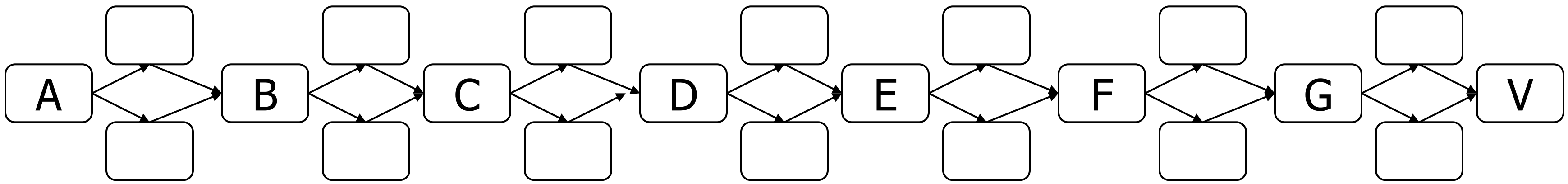
- Defined in terms of paths in the CFG:
  - There is an association *(d,u)* between a definition of variable *v* at *d* and a use of variable *v* at *u* if and only if
    - there is at least one control flow path from *d* to *u*
    - with no intervening definition of *v*
  - v_d **reaches** *u* (v_d is a **reaching definition** at *u*).
  - If a control flow path passes through another definition *e* of the same variable *v*, v_e **kills** v_d at that point.

Calculating def-use pairs



- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges.
- Practical algorithms therefore do not search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

Example: Exponential number of paths (even without loops)



2 paths from *A* to *B*

4 from *A* to *C*

8 from *A* to *D*

16 from *A* to *E*

...

128 paths from *A* to *V*

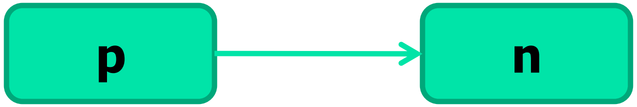
Tracing each path is not efficient, and we can do much better.

An Iterative Algorithm for Computing Reaching Definitions

- Based on the way reaching definitions at one node are related to the reaching definitions at an adjacent node

Propogation of Information Among Nodes of a CFG

- Suppose we are calculating the reaching definitions of node *n*, and there is an edge *(p,n)* from an immediate predecessor node *p*
  - If the predecessor node *p* can assign a value to variable *v*, then the definition v_p reaches *n*. We say the definition v_p is generated at *p*.
  - If a definition v_p of variable *v* reaches a predecessor node *p*, and if *v* is not redefined at that node, then the definition is propagated on from *p* to *n*.



Equations of node E (y = tmp)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;              // A: def x, y, tmp
    while (y != 0) {      // B: use y
        tmp = x % y;      // C: def tmp; use x, y
        x = y;            // D: def x; use y
        y = tmp;          // E: def y; use tmp
    }
    return x;             // F: use x
  }
```

- Reach(E) = ReachOut(D)
- ReachOut(E) = (Reach(E) \ $\{y_A\}$) $\cup$ $\{y_E\}$

Equations of node B (while (y != 0))

This line has two predecessors: Before the loop, end of the loop

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;               // A: def x, y, tmp
    while (y != 0) {       // B: use y
        tmp = x % y;       // C: def tmp; use x, y
        x = y;             // D: def x; use y
        y = tmp;           // E: def y; use tmp
    }
    return x;              // F: use x
  }
```

- $Reach(B) = ReachOut(A) \cup ReachOut(E)$
- $ReachOut(A) = gen(A) = \{x_A, y_A, tmp_A\}$
- $ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$

General equations for Reach analysis

$$\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$$

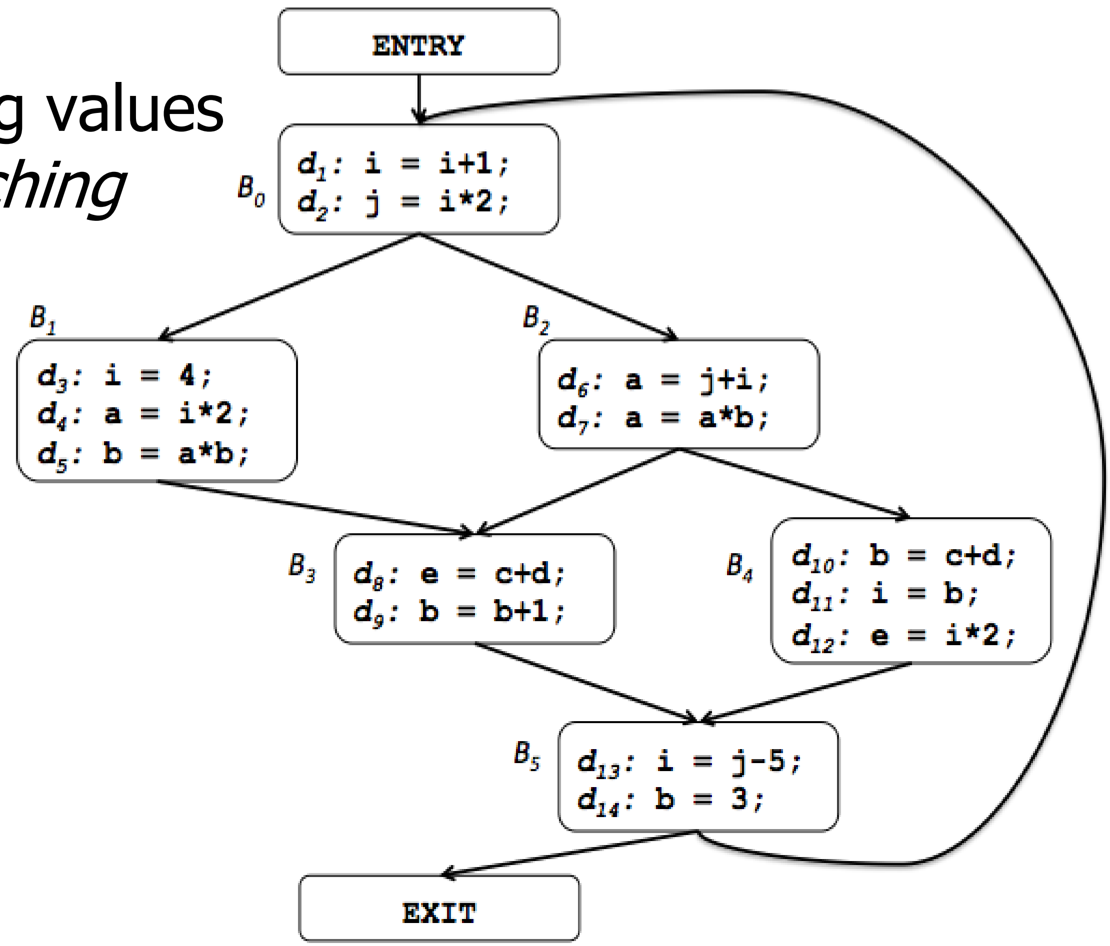
$$\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

$$\text{gen}(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$$

$$\text{kill}(n) = \{ v_x \mid v \text{ is (re)defined or modified at } x, x \neq n \}$$

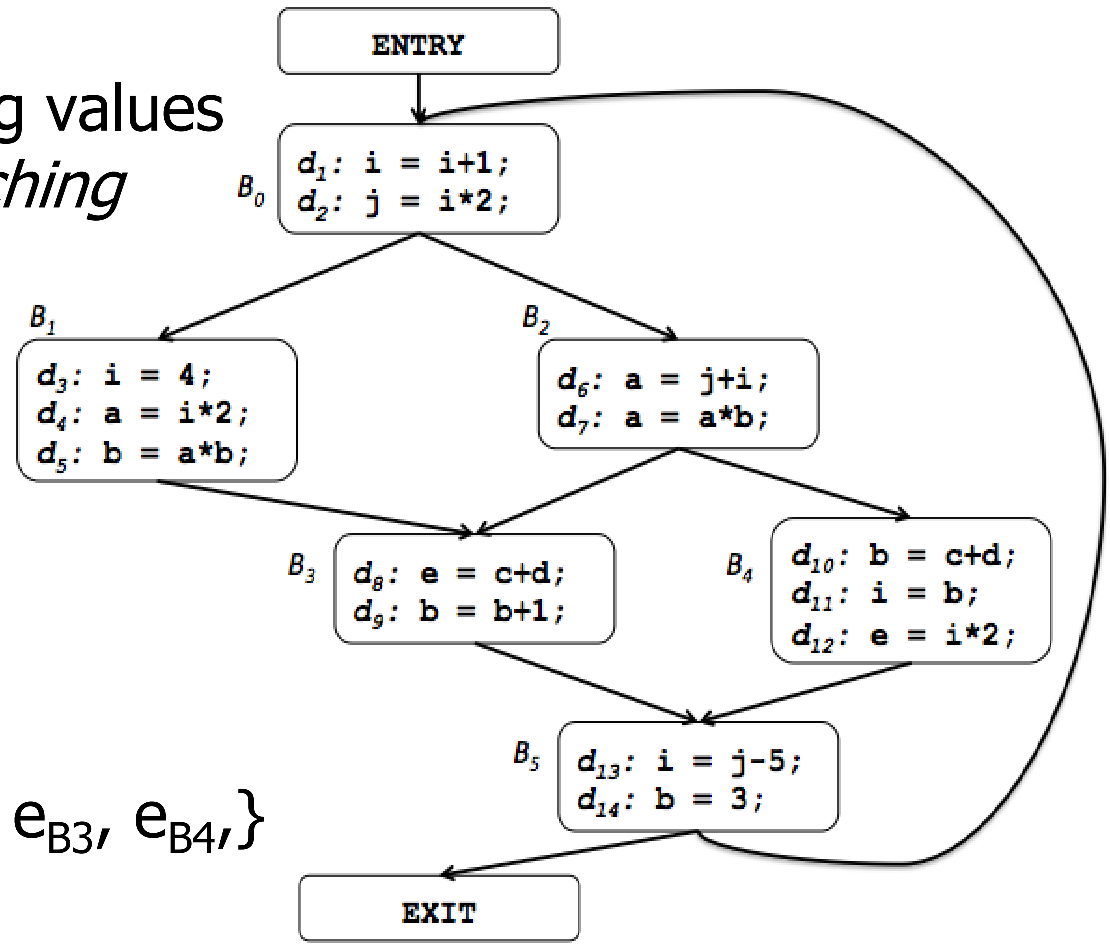
Exercise

- What are the following values according to the *Reaching Definitions* analysis?
  - gen(B_0)
  - kill(B_0)
  - ReachOut(B_0)



Exercise

- What are the following values according to the *Reaching Definitions* analysis?
  - $gen(B_0)=$ $\{i_{B0}, j_{B0}\}$
  - $kill(B_0)=$ $\{i_{B1}, i_{B4}, i_{B5}\}$
  - $ReachOut(B_0)=$ $\{i_{B0}, j_{B0}, a_{B1}, a_{B2}, b_{B5}, e_{B3}, e_{B4},\}$



Avail equations (available expressions)

$$\text{Avail}(n) = \bigcap_{m \in \text{pred}(n)} \text{AvailOut}(m)$$

"all paths" rather than "any path"

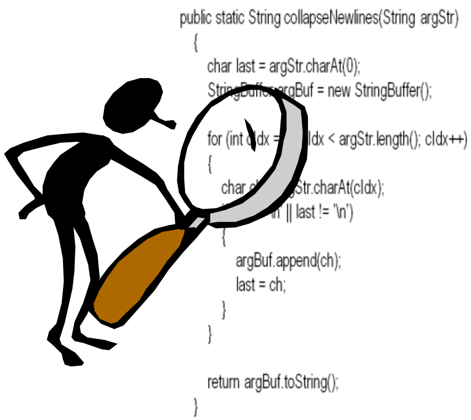
$$\text{AvailOut}(n) = (\text{Avail}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$$

gen(n) = { exp | exp is computed at n }

kill(n) = { exp | exp has variables assigned at n }

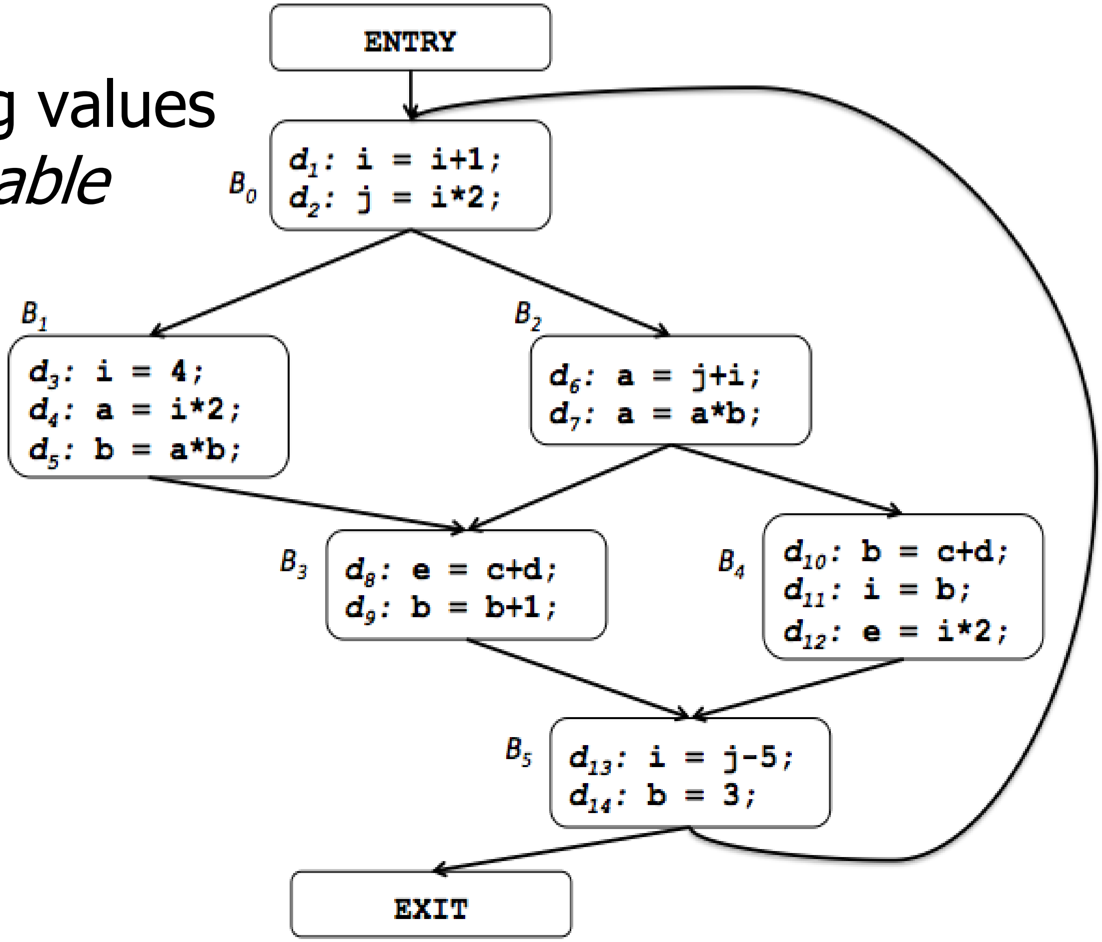
Possible Application of Avail analysis

- Compiler Optimization
  - If an expression is available, do not recompute it
- Enforcing Variable Initialization
  - Java requires a variable to be initialized before use on all execution paths
    - **kill** sets are empty since there is no way to “unitialize” a variable in Java



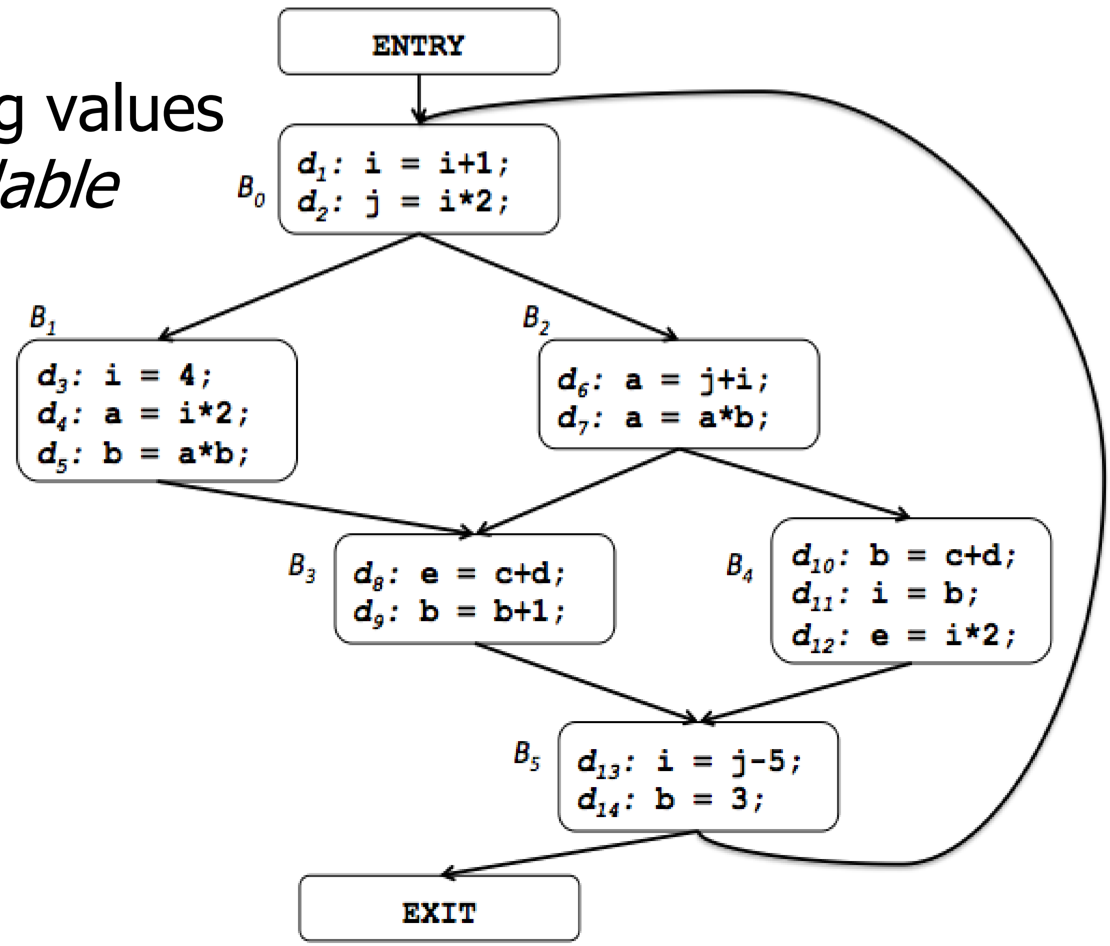
Exercise

- What are the following values according to the *Available Expressions* analysis?
  - gen(B_0)
  - kill(B_0)
  - Avail(B_5)



Exercise

- What are the following values according to the *Available Expressions* analysis?
  - gen(B_0)= {i*2}
  - kill(B_0)= {j+i, j-5, i+1}
  - Avail(B_5)= {i*2, c+d}



Live variable equations

$$Live(n) = \bigcup_{m \in succ(n)} LiveOut(m)$$

if the variable might be used in "any following path"

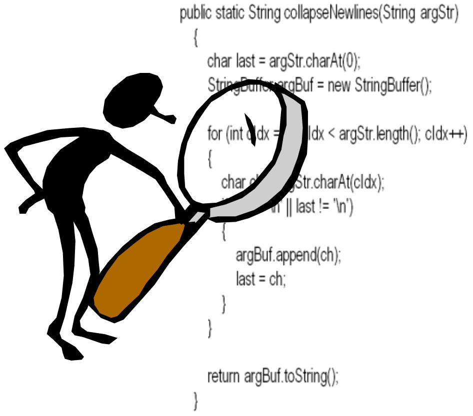
$$LiveOut(n) = (Live(n) \setminus kill(n)) \cup gen(n)$$

$$gen(n) = \{ v \mid v \text{ is used at } n \}$$

$$kill(n) = \{ v \mid v \text{ is modified at } n \}$$

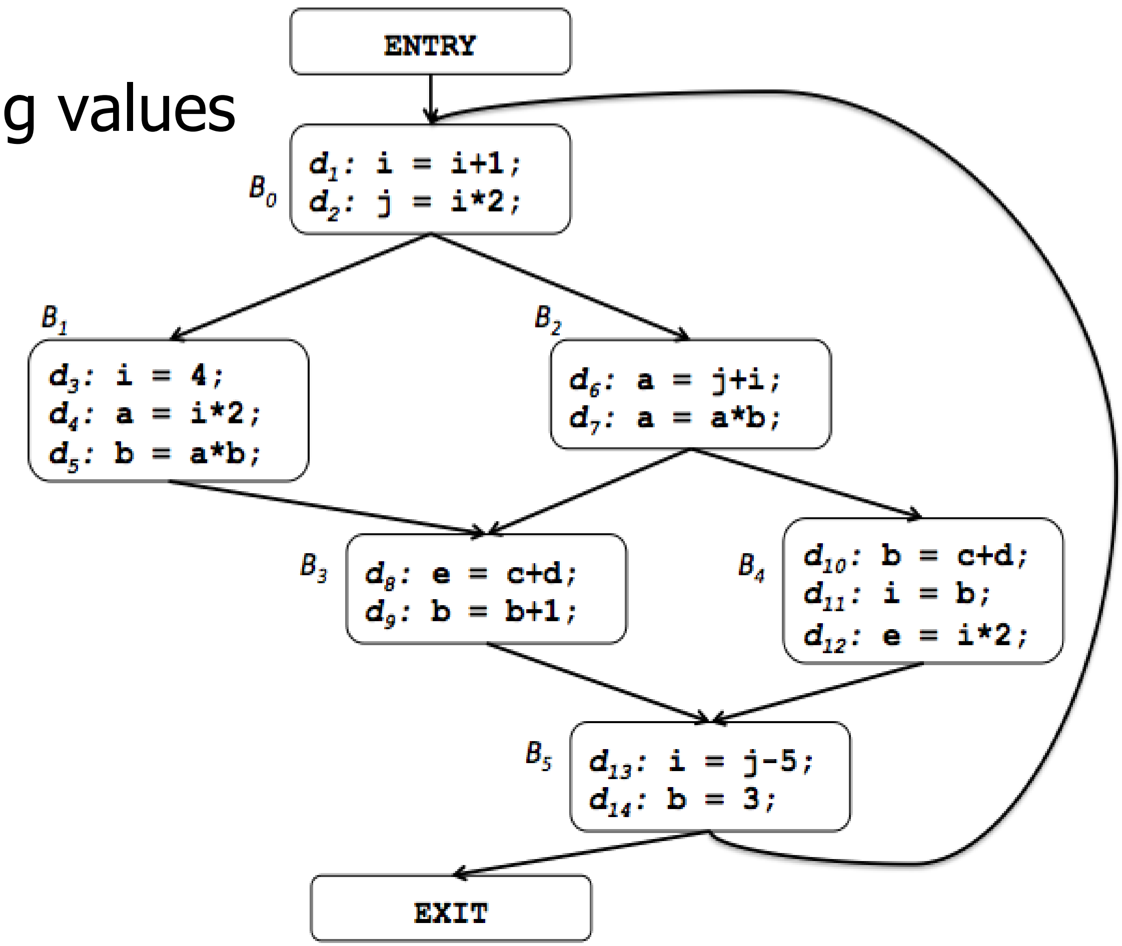
Possible Application of Live Variable Analysis

- Recognizing useless definitions
  - Often symptomatic for a fault, e.g., misplelling a variable name



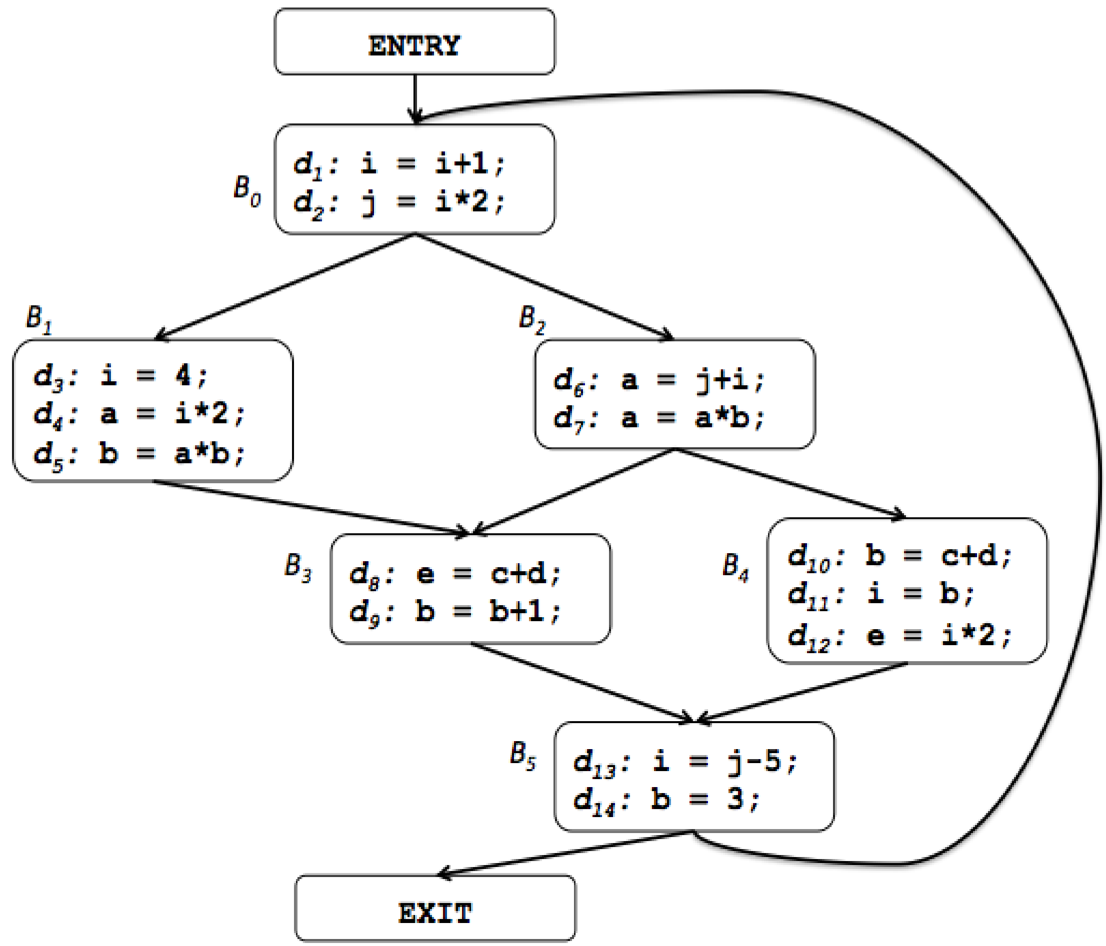
Exercise

- What are the following values according to the *Live Variables* analysis?
  - gen(B_0)
  - kill(B_0)
  - Live(B_0)



Exercise

- What are the followir according to the *Live Variables* analysis?
  - gen(B_0)= {i}
  - kill(B_0)= {i, j}
  - Live(B_0)= {i, j, b, c, d}



Classification of analyses

- Forward/backward: a node’s set depends on that of its predecessors/successors
- Any-path/all-path: a node’s set contains a value iff it is coming from any/all of its inputs

	Any-path ($\cup$)	**All-paths ($\cap$)**
Forward (pred)	Reach	Avail
Backward (succ)	Live	“inevitable”

Inevitability Definition

$$Inev(n) = \bigcap_{m \in succ(n)} InevOut(m)$$

all paths

backward analysis

$$InevOut(n) = (Inev(n) \setminus kill(n)) \cup gen(n)$$

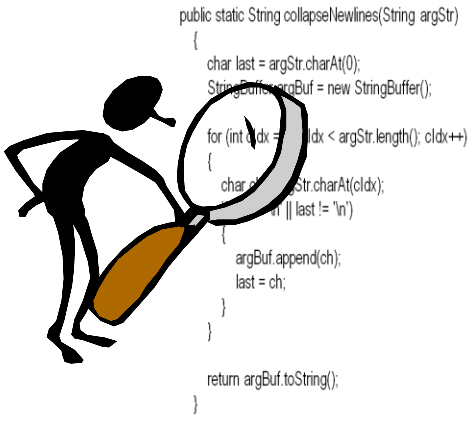
$$gen(n) = \{ v \mid v = q \}$$

$$kill(n) = \{ \}$$

Here, we are interested in the accessibility of a node, not (re)definition or modification of variables

“Inevitability” Analysis

- Example usage scenarios:
  - Ensuring that interrupts are reenabled after executing an interrupt-handling routine
  - Ensuring that files are closed after opening them
  - …



Iterative Solution of Dataflow Equations

- Initialize values (first estimate of answer)
  - For “any path” problems, first guess is “nothing” (empty set) at each node
  - For “all paths” problems, first guess is “everything” (set of all possible values = union of all “gen” sets)
- Repeat until nothing changes
  - Pick some node and recalculate (new estimate)

This will converge on a “fixed point” solution where every new calculation produces the same value as the previous guess.

Data flow analysis with arrays and pointers

- Arrays and pointers introduce uncertainty:
  Do different expressions access the same storage?
  - a[i] same as a[k] when i = k
  - a[i] same as b[i] when a = b (**aliasing**)
- The uncertainty is accomodated depending to the kind of analysis
  - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
  - All-path: vice versa