

# Software Testing and Validation

A.A. 2025/2026

Corso di Laurea in Informatica

## Finite Models of Software

Igor Melatti

Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica



# Models in Testing

- Model checking is based on models of the artifact, testing addresses the artifact
- However, some modeling is often required also for testing
  - models for the environment (i.e., what is providing inputs)
  - models for plant, when the software is a controller
    - in some cases, testing on the final product in its “natural” environment only may be also dangerous
    - e.g., testing of the controller for a flying aircraft
  - models of the software itself
    - UML diagrams
    - control flow diagrams et al. (will be defined in the following)
    - help in devising better tests
- May be already available from specifications, or a modeling phase may be needed



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Models Must Be...

- Compact, i.e., understandable
  - often, they are for human inspection
  - if models are for some automatic procedure, then they must be manipulable in the given computational resources
    - this is exactly the case for model checking!
- Predictive, i.e., not too simple
  - at least be able to detect what is “bad” and what is “good”
  - different models may be used for the same artifact, when testing different aspects
  - e.g., model to predict airflow w.r.t. efficient passenger loading and safe emergency exit



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Models Must Be...

- Semantically meaningful
  - given something went bad, we need to understand why
  - identify the part with the failure
- Sufficiently general
  - not too specialized on some characteristics
  - otherwise, not useful
  - e.g., a C program analyzer which only works for programs without pointers



# Finite Abstraction of Behaviour

- Given a program, a state is an assignment for all variables in the program
  - state space*: set of all possible states
- A behaviour is a sequence of states, interleaved by program statements being executed
- The number of behaviours for non-trivial programs is extremely huge
  - infinite if we do not consider machine limitations
  - e.g., integers need not to be represented on maximum 64 bits
- An *abstraction* is a function from states to (reduced) states
  - some details are suppressed
  - e.g., some variables are not considered



DISIM  
Dipartimento di Ingegneria  
dell'Informazione e  
Matematica

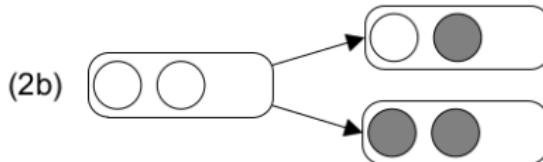
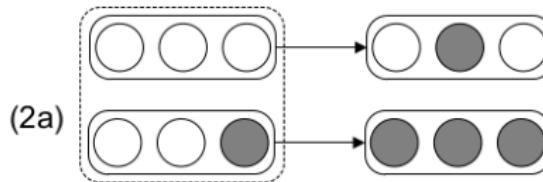
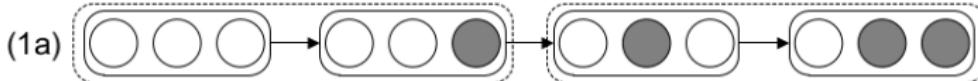
# Finite Abstraction of Behaviour

- Two different states may be considered the same by an abstraction
  - e.g., they differ by some variable, which is abstracted out
- States sequences may be squeezed
- Non-determinism may be introduced
  - e.g., when a choice was made by considering the value of some abstracted-out variable
- In model checking, this is done by hand for each system
  - here, instead, we will consider some standard models which are especially tailored for testing
  - in some cases, they may be automatically extracted from code



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Finite Abstraction of Behaviour



(Each circle is a binary variable...)

# (Intraprocedural) Control Flow Graphs

- Model close to the actual program source code
  - finite by construction
- Often compilers are also able to build the control flow graph
  - e.g., gcc -fdump-tree-cfg
- Directed graph:
  - nodes are program statements
    - may also be group of statements or fragments of statements
  - edges represent the possibility to go from a node to another
    - either by branch or by sequential execution
    - max outgoing degree is 2, excluding switches...
  - cycles in the code correspond to cycles in the CFG and viceversa
  - paths in the CFG correspond to executions of code and viceversa
  - connected, each path goes from start to finish



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



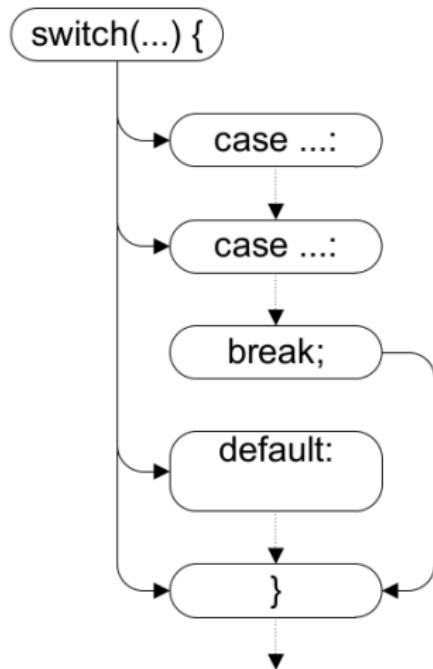
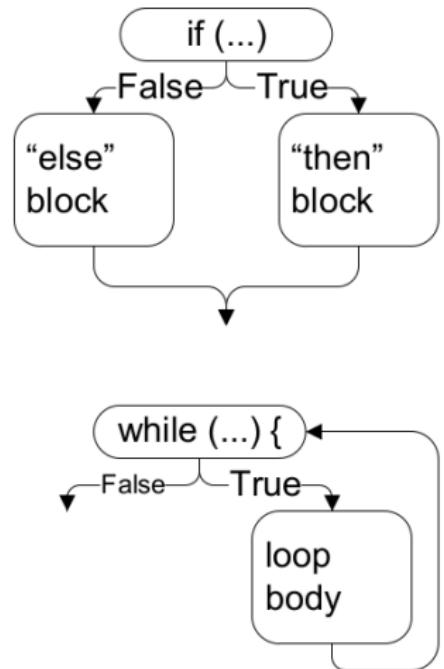
DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Control Flow Graphs (CFGs)

- Nodes usually are a maximal group of statements with a single entry and single exit
  - basic block
  - i.e., *always sequential* assignments are grouped together
  - in a maximal way
- On the contrary, it may happen that a single statement is broken down
  - because it is not always executed with a single entry and a single exit
  - e.g., the `for` statement
  - e.g., short-circuit evaluation
  - e.g., other strange cases: `a = (b++? c++ : ++d);`



# Control Flow Graphs



# Control Flow Graphs

```
1      /**
2      * Remove/collapse multiple newline characters.
3      *
4      * @param String string to collapse newlines in.
5      * @return String
6      */
7  public static String collapseNewlines(String argStr)
8  {
9      char last = argStr.charAt(0);
10     StringBuffer argBuf = new StringBuffer();
11
12     for (int cldx = 0 ; cldx < argStr.length(); cldx++)
13     {
14         char ch = argStr.charAt(cldx);
15         if (ch != '\n' || last != '\n')
16         {
17             argBuf.append(ch);
18             last = ch;
19         }
20     }
21
22     return argBuf.toString();
23 }
```

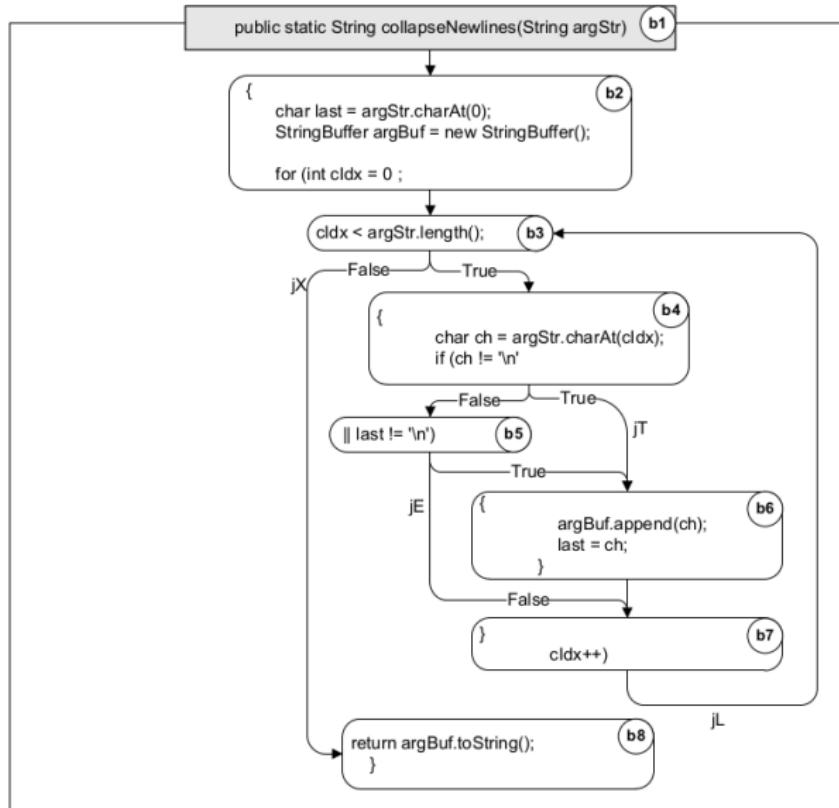


UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Control Flow Graphs



# Control Flow Graphs

- Let  $P$  be a (part of a) function or procedure for which testing must be performed
  - white-box testing: we know the code of  $P$  as a sequence  $\mathcal{C}(P) = \langle I_1, \dots, I_k \rangle$  of statements
  - we assume  $P$  is written in some imperative language
  - we assume that complex statements in  $\mathcal{C}(P)$  are already broken down in parts
    - short-circuited conditions, inline increments, function/procedure calls...
  - in the previous example (`collapseNewlines`),  $k = 11$



# Control Flow Graphs

- Let  $g = \langle i_1, \dots, i_m \rangle$  be a grouping for the statements of  $\mathcal{C}(P)$ 
  - $1 \leq i_j < i_{j+1} \leq k$  for all  $j = 1, \dots, m - 1$
  - e.g., for  $g = \langle 3, 5, 10 \rangle$  we will consider three blocks:
    - the first 3 statements, then other two statements, and finally the remaining 5 statements
  - we will call  $g$  *granularity* for a given  $\mathcal{C}(P)$
- Of course, granularities must comply with code
  - no flow branches (if, while, etc) inside a block  $I_{i_j+1} \dots I_{i_{j+1}}$
- Usually, maximal granularities are chosen
  - from a flow branch (or starting point) to another flow branch (or ending point)
  - in the previous example (`collapseNewlines`),  
 $g = \langle 3, 4, 6, 7, 9, 10, 11 \rangle$



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Control Flow Graphs

- A CFG for a program  $P$  with granularity  $g$  is a graph  $G = (V, E)$  s.t.
  - $V = \{\langle I_{g_{i-1}+1} \dots I_{g_i} \rangle \mid i = 1, \dots, |g|\}$ 
    - with  $g_0 = 0$
    - nodes are basic blocks and  $|V| = |g|$
  - $E = \{(u, v) \mid u, v \in V \wedge \text{control flow from last statement of } u \text{ and first of } v \text{ may take place}\}$
- Typically, nodes  $v_i \in V$  are labeled with the corresponding basic block  $\langle I_{g_{i-1}+1} \dots I_{g_i} \rangle$
- Typically, edges  $(u, v) \in E$  may be labeled by a boolean value if flow from  $u$  to  $v$  is conditioned
  - last statement of  $u$  is an if or a while
    - and similar, e.g., for, until etc
- In some cases, some alphanumeric label is added to ease references

# From CFG to LCSAJ

- Linear code sequences and jumps
  - maximal sequences of consecutive statements
  - may be directly derived from a CFG
- In a nutshell: all sequences of consecutive basic blocks
  - while a basic block cannot contain branches, LCSAJ can
  - while you can go back in a CFG, you cannot go back in a LCSAJ
    - see example: no  $b7 \rightarrow b3$
    - thus, conditional branches create overlapping LCSAJs
    - basic blocks cannot overlap
- Typically, there are 4x more LCSAJs than basic blocks



# From CFG to LCSAJ: Idea

- Look at the CFG (also the code, but it is easier in the CFG)
- You can go on till when you are forced to stop
  - you are forced to stop when, w.r.t. the code, you have to go more than a step further, or simply back
- You can stop also if there is the possibility to not going in the following step
- Let  $G = (V, E, L_1, L_2)$  be a labeled CFG
  - $L_1 : V \rightarrow \mathcal{L}_V, L_2 : E \rightarrow \mathcal{L}_E$  are two bijective labeling functions for nodes (basic blocks) and edges, respectively
  - no really need of having the labeling function: it simply makes the LCSAJ more readable



# Control Flow Graphs

```
1      /**
2      * Remove/collapse multiple newline characters.
3      *
4      * @param String string to collapse newlines in.
5      * @return String
6      */
7  public static String collapseNewlines(String argStr)
8  {
9      char last = argStr.charAt(0);
10     StringBuffer argBuf = new StringBuffer();
11
12     for (int cldx = 0 ; cldx < argStr.length(); cldx++)
13     {
14         char ch = argStr.charAt(cldx);
15         if (ch != '\n' || last != '\n')
16         {
17             argBuf.append(ch);
18             last = ch;
19         }
20     }
21
22     return argBuf.toString();
23 }
```

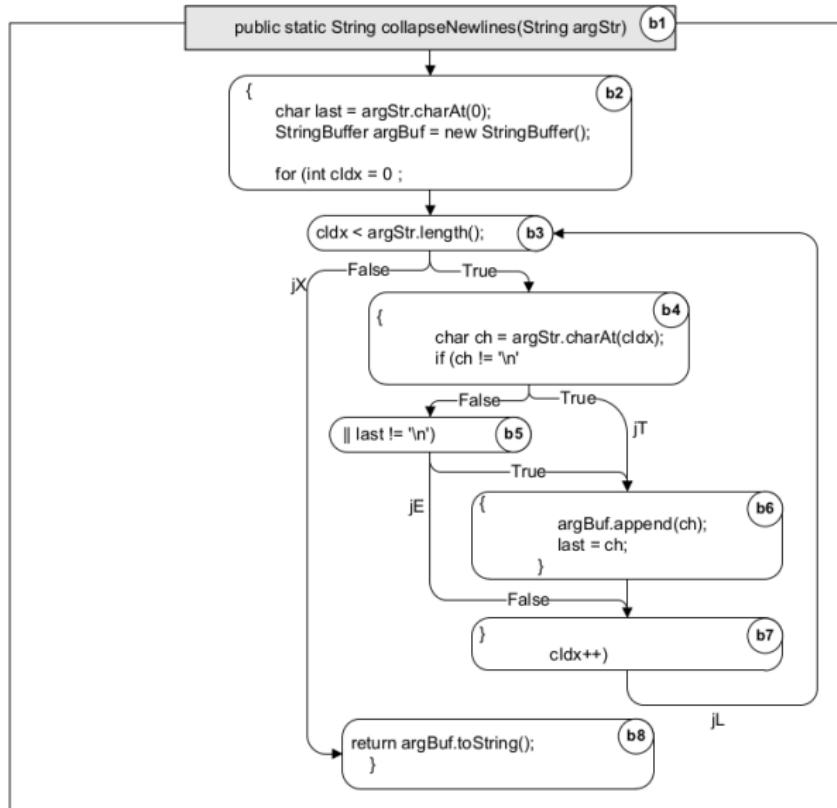


UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Control Flow Graphs



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# From CFG to LCSAJ

<i>From</i>	<i>Sequence of Basic Blocks</i>							<i>To</i>
entry	b1	b2	b3					jX
entry	b1	b2	b3	b4				jT
entry	b1	b2	b3	b4	b5			jE
entry	b1	b2	b3	b4	b5	b6	b7	jL
jX							b8	return
jL		b3	b4					jT
jL		b3	b4	b5				jE
jL		b3	b4	b5	b6	b7		jL



# From CFG to LCSAJ

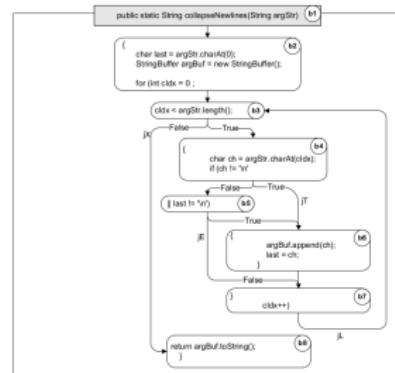
- Let  $G = (V, E, L_1, L_2)$  be a labeled CFG
- The LCSAJ associated to  $G$  is  
 $\mathcal{I}(G) = \{\langle l_1, \ell_2, l_3 \rangle \mid l_1, l_3 \in \mathcal{L}_E, \ell_2 \in \mathcal{L}_V^*\}$  s.t.:
  - $l_1$  arrives to the first statement of  $\ell_2$ 
    - that is: if  $L_2^{-1}(l_1) = (u, v)$ , then  $\ell_2$  begins with  $L_1(v)$
  - $l_3$  exits from the last statement of  $\ell_2$ 
    - that is: if  $L_2^{-1}(l_3) = (u, v)$ , then  $\ell_2$  ends with  $L_1(u)$
  - $\ell_2$  contains *consecutive* basic blocks of  $\mathcal{C}(P)$  connecting  $l_1$  to  $l_3$ 
    - that is,  $\ell_2 = v_1 \dots v_n$  implies that:
      - $v_i$  and  $v_{i+1}$  are consecutive basic blocks both in  $G$  and in the source code for all  $i = 1, \dots, n - 1$
      - $v_n$  is either followed by a control flow jump or it is the end of the unit
      - $v_1$  is either the beginning of the unit or the destination of a control flow jump



# From CFG to LCSAJ

From	Sequence of Basic Blocks	To
entry	b1 b2 b3	jX
entry	b1 b2 b3 b4	jT
entry	b1 b2 b3 b4 b5	jE
entry	b1 b2 b3 b4 b5 b6 b7	jL jX
jL	b3 b4	jT
jL	b3 b4 b5	jE
jL	b3 b4 b5 b6 b7	jL

b8 return



- b1 is the start; b3 and b8 are destinations of control flow jumps
  - thus, LCSAJs can start from one of them
- Starting from one of these, one different LCASJ each time you see a branch
- Many overlapping; they are combined in actual testing
  - so that ending and starting points coincide



# Algorithm for LCSAJs

```
LCSAJ( $C$ ) {
     $W \leftarrow \text{getStartingBlocks}(C)$ ;
     $L \leftarrow \emptyset$ ;
    for  $v \in W$  {
         $H \leftarrow \emptyset$ ;
        DFSLCSAJ( $C$ ,  $v$ );
    }
    return  $L$ ;
}
```



# Algorithm for LCSAJs

```
DFSLCSAJ( $C, v, S$ ) { //  $C = (V, E, s)$ 
     $H \leftarrow H \cup \{v\}$ ;  $S \leftarrow \text{push}(S, v)$ ;
     $A \leftarrow \{w \in V \mid (v, w) \in E\}$ ;
    if ( $v + 1 \notin A$ ) {
        if ( $S \notin L \wedge$ 
            ( $|S| > 1 \vee A = \emptyset$ )  $\wedge \forall i = 1, \dots, |S| - 1. S[i] = S[i] - 1$ )
             $L \leftarrow L \cup \{S\}$ ;
    }
    else {
        if ( $|A| > 1 \wedge S \notin L \wedge$ 
            ( $|S| > 1 \vee A = \emptyset$ )  $\wedge \forall i = 1, \dots, |S| - 1. S[i] = S[i] - 1$ )
             $L \leftarrow L \cup \{S\}$ ;
        for  $w \in A$  {
            if ( $w \notin H$ )
                DFSLCSAJ( $V, E, w$ );
        }
         $S \leftarrow \text{pop}(S)$ ;
    }
}
```

# Algorithm for LCSAJs

```
getStartingBlocks( $C$ ) {
    let  $C = (V, E, s)$ ;
     $H, V_1, V_2 \leftarrow \emptyset, \emptyset, \emptyset$ ;
    allStartingBlocks( $V, E, s$ );
     $H \leftarrow \emptyset$ ;
     $V_2 \leftarrow \text{correctStartingBlocks}(V, E, V_2)$ ;
    return  $\{s\} \cup V_1 \cup V_2$ ;
}

allStartingBlocks( $V, E, v$ ) {
     $H \leftarrow H \cup \{v\}$ ;
    for  $w \in V$  s.t.  $(v, w) \in E$  {
        if  $(w < v)$   $V_1 \leftarrow V_1 \cup \{w\}$ ;
        else if  $(w \neq v + 1)$   $V_2 \leftarrow V_2 \cup \{w\}$ ;
        if  $(w \notin H)$  allStartingBlocks( $V, E, w$ );
    }
}
```



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Algorithm for LCSAJs

```
correctStartingBlocks( $V, E, v$ ) {  
     $H \leftarrow H \cup \{v\}$ ;  
    for  $w \in V$  s.t.  $(v, w) \in E$  {  
        if  $(w = v + 1 \wedge w \in V_2)$   
             $V_2 \leftarrow V_2 \setminus \{w\}$ ;  
        if  $(w \notin H)$   
            correctStartingBlocks( $V, E, w$ );  
    }  
}
```



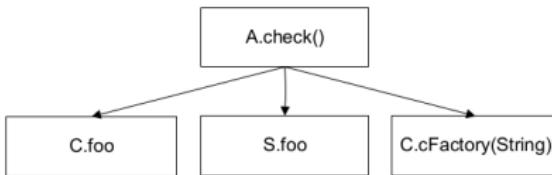
# Call Graphs

- CFG is typically intraprocedural; call graphs are interprocedural
  - simply a graph where nodes are defined functions
  - there is an edge from  $f$  to  $g$  iff  $f$  *may* call  $g$
  - order of calls is not important
  - thus, they may contain calls which are actually never made
  - sometimes arguments are made explicit
  - number of paths inside a call graph may be exponential, even without recursion



# Call Graphs

```
1  public class C {  
2  
3      public static C cFactory(String kind) {  
4          if (kind == "C") return new C();  
5          if (kind == "S") return new S();  
6          return null;  
7      }  
8  
9      void foo() {  
10         System.out.println("You called the parent's method");  
11     }  
12  
13     public static void main(String args[]) {  
14         (new A()).check();  
15     }  
16 }  
17  
18 class S extends C {  
19     void foo() {  
20         System.out.println("You called the child's method");  
21     }  
22 }  
23  
24 class A {  
25     void check() {  
26         C myC = C.cFactory("S");  
27         myC.foo();  
28     }  
29 }
```



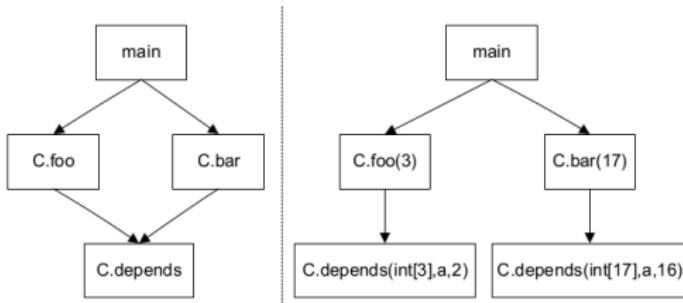
UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



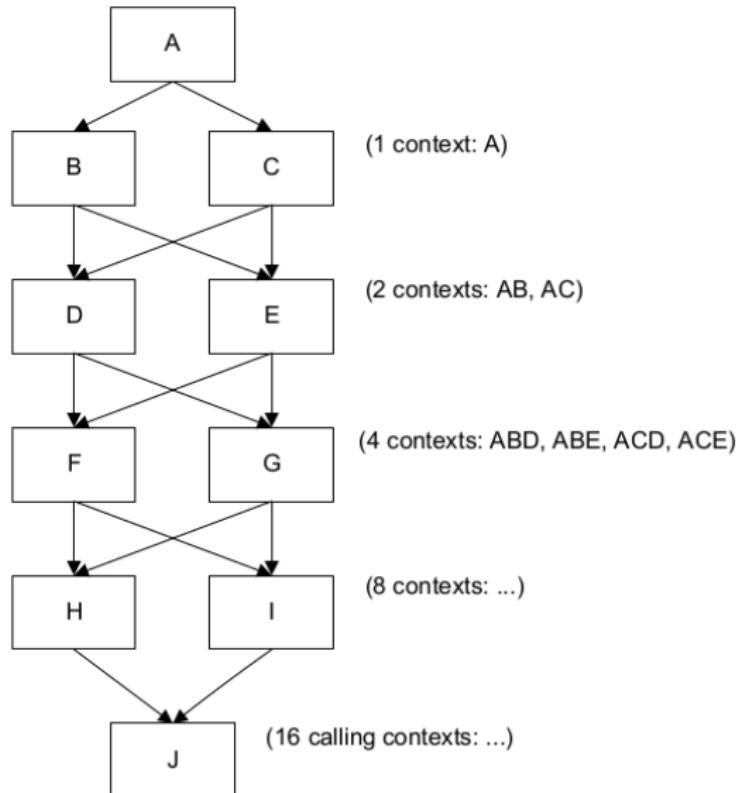
DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Call Graphs

```
1 public class Context {  
2     public static void main(String args[]) {  
3         Context c = new Context();  
4         c.foo(3);  
5         c.bar(17);  
6     }  
7  
8     void foo(int n) {  
9         int[] myArray = new int[ n ];  
10        depends( myArray, 2 );  
11    }  
12  
13    void bar(int n) {  
14        int[] myArray = new int[ n ];  
15        depends( myArray, 16 );  
16    }  
17  
18    void depends( int[] a, int n ) {  
19        a[n] = 42;  
20    }  
21 }
```

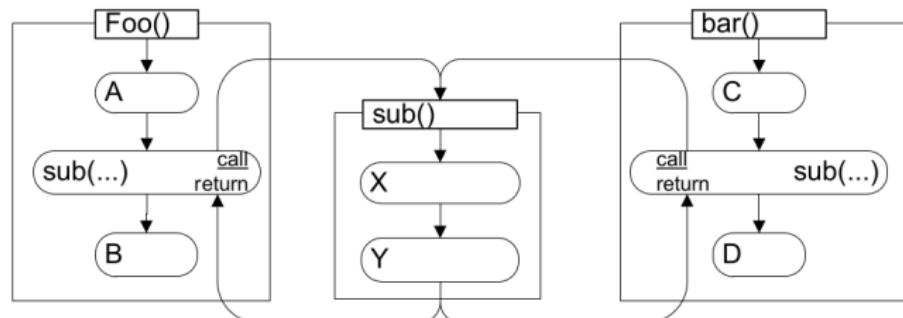


# Call Graphs



# Interprocedural Analysis

- Calls between different functions/methods, important, e.g., for the previous slide
- Simply following calls and returns in a CFG-like way is not practical: too many spurious paths
  - $(A, X, Y, B), (C, X, Y, D)$  are ok
  - $(A, X, Y, D), (C, X, Y, B)$  are impossible



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

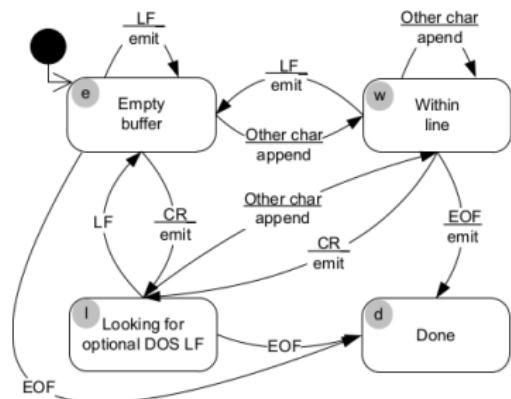
# Interprocedural Analysis

- To solve the problem, context is needed
  - if sub is called by A, it must return in B
- Number of contexts is exponential
  - may be ok for a small group of functions, e.g., a not-too-big single Java class
- Some special cases exist
  - the info needed to analyze the calling procedure must be small
  - e.g., proportional to the number of called procedures
  - the information about the called procedure must be context-independent
  - example: declaration of exception throwing in Java



# Finite State Machines

- Here we will focus on Mealy Machines
  - a graph where nodes are “modalities” of a given software
  - edges are labeled with input/output



	LF	CR	EOF	other
e	e / emit	l / emit	d / -	w / append
w	e / emit	l / emit	d / emit	w / append
l	e / -		d / -	w / append



# Finite State Machines

```
1  /** Convert each line from standard input */
2  void transduce() {
3
4      #define BUflen 1000
5      char buf[BUflen]; /* Accumulate line into this buffer */
6      int pos = 0; /* Index for next character in buffer */
7
8      char inChar; /* Next character from input */
9
10     int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12     while ((inChar = getchar()) != EOF) {
13         switch (inChar) {
14             case LF:
15                 if (atCR) { /* Optional DOS LF */
16                     atCR = 0;
17                 } else { /* Encountered CR within line */
18                     emit(buf, pos);
19                     pos = 0;
20                 }
21             break;
22             case CR:
23                 emit(buf, pos);
24                 pos = 0;
25                 atCR = 1;
26             break;
27             default:
28                 if (pos >= BUflen-2) fail("Buffer overflow");
29                 buf[pos++] = inChar;
30             } /* switch */
31         }
32         if (pos > 0) {
33             emit(buf, pos);
34         }
35     }
```



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Mealy Machine Formal Definition

- A Mealy machine is a 6-tuple  $\mathcal{M} = (S, S_0, \Sigma, \Lambda, T, G)$  consisting of the following:
  - a finite set of states  $S$
  - a start state (also called initial state)  $S_0 \in S$
  - a finite set called the input alphabet  $\Sigma$
  - a finite set called the output alphabet  $\Lambda$
  - a (deterministic!) transition function  $T : S \times \Sigma \rightarrow S$  mapping pairs of a state and an input symbol to the corresponding next state
  - an output function  $G : S \times \Sigma \rightarrow \Lambda$  mapping pairs of a state and an input symbol to the corresponding output symbol.
- Given an input  $w \in \Sigma^*$ ,  $\mathcal{M}$  outputs  $o \in \Lambda^*$ ,  $|o| = |w|$  s.t.
  - $\forall i = 1, \dots, |w|. s_i = T(s_{i-1}, w_i) \wedge o_i = G(s_{i-1}, w_i)$
  - $s_0 = S_0$



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Data Flow Models

- CFGs, FSMs etc are a good way to represent *control flow*
- What about *data flow*?
- Again, ideas are borrowed from compilers theory
  - data flow is used to detect errors for type checking, or also opportunities for optimization
  - also used in software engineering tout court, for refactoring or reverse engineering
- As for testing, useful for:
  - select test cases based on dependence information
  - detect anomalous patterns that indicate probable programming errors, e.g. usage of uninitialized values



# Definition-Use Pairs

- Definition of a variable: either its declaration or a write access
  - for languages like Python, mostly write access...
  - write access may be:
    - left part of an assignment
    - parameter initialization in function calls
    - other special cases such as ++ construct in C-like languages
- Use of a variable: a read access
  - right part of an assignment
  - variable passed in function calls
  - variable used without being modified
- The same line of code may be both definition and use
  - typically, nearly all lines either define and/or use at least one variable
  - ++ construct is both definition and use on the same variable



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Definition-Use Pairs

```
1  public int gcd(int x, int y) {           /* A: def x,y */
2      int tmp;                          /*      def tmp */
3      while (y != 0) {                  /* B: use y */
4          tmp = x % y;                /* C: use x,y, def tmp */
5          x = y;                      /* D: use y, def x */
6          y = tmp;                    /* E: use tmp, def y */
7      }                                /* F: use x */
8      return x;
9  }
```



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione e Matematica

# Definition-Use Pairs

- A variable has only definitions? it is useless
- A variable has only uses? there is some error
- For a given definition, there may be many uses, and viceversa
  - of course, for a fixed variable
  - see  $y$  in the previous slide: 2 definitions, 3 uses...
- A definition-use pair combines a given use with the *closest* definition
  - w.r.t. some possible execution (*path*) of the code
- Other definitions behind the closest one are *killed*



# Definition-Use Pairs

- Consider an execution path  $\pi = s_1, \dots, s_m$ :
  - $s_i$  are *statements* and  $s_i, s_{i+1}$  may be contiguous in  $\pi$  iff the control flow may go from  $s_i$  to  $s_{i+1}$
  - e.g., from the previous code: 1,2,3,8,9 and 1,2,3,4,5,6,7,3,4,5,6,7,8,9 and 1,2,3,4,5,6,7,3,4
  - if we consider the corresponding CFG  $G$ , then  $\pi$  is a path of  $G$
- Consider an execution path  $\pi = s_1, \dots, s_m$  and a variable  $v$ :
  - if  $\exists k. \text{use}(v) \in s_k$ , let  $L = \{\ell < k \mid \text{def}(v) \in s_\ell\}$
  - $(d, u) = (\max L, k)$  is a definition-use pair
  - $v_d$  reaches  $u$  or  $v_d$  is a *reaching definition* of  $u$
  - $s_\ell$  is a *killed definition* if  $\ell \in L \wedge \ell \neq \max L$
  - the sub-path  $s_d \dots s_k$  is *definition-clear*

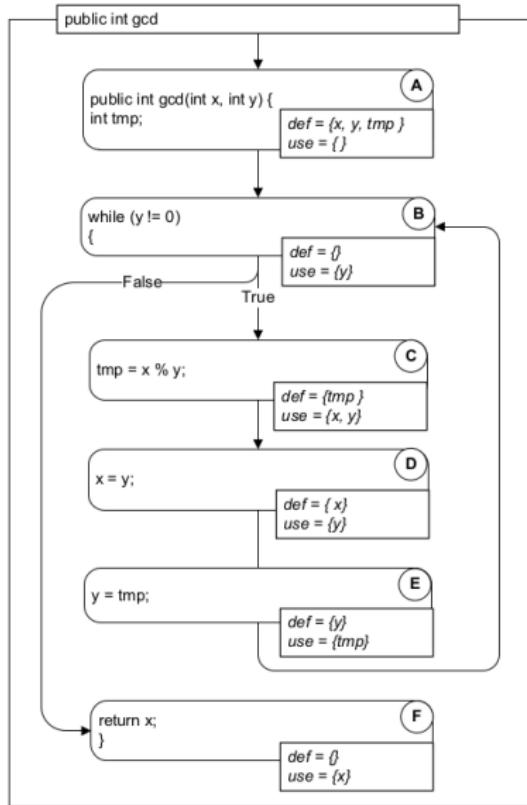


UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Definition-Use Pairs

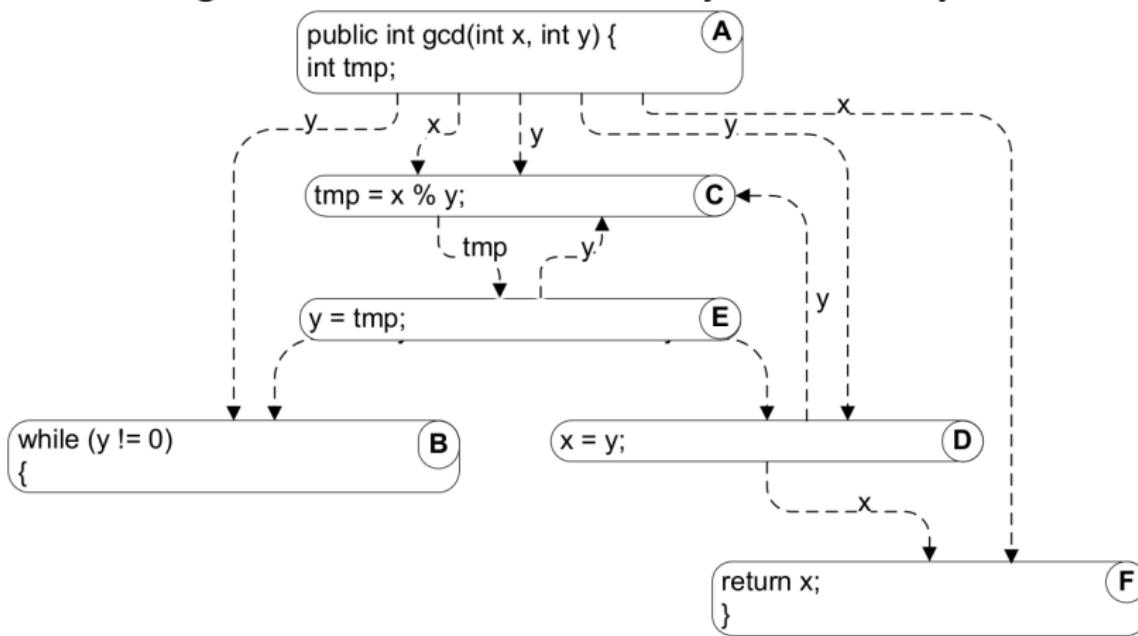


# Definition-Use Pairs

- Use-definition pairs defines a *direct data dependence*, can be used to build the *data dependence graph*
  - there is an edge  $(s, t)$  with label  $v$  iff  $(s, t)$  is a definition-use pair for variable  $v$  (for some path)
- *Granularity* on nodes may be tuned according to needs:
  - single expressions (especially for compilers)
  - statements (figure below)
  - basic blocks
  - etc

# Definition-Use Pairs

Errata corrigé: D→C with x, E→D with y, E→B with y



# Algorithm to Generate All Reaching Definitions

## Algorithm *Reaching definitions*

Input: A control flow graph  $G = (\text{nodes}, \text{edges})$   
 $\text{pred}(n) = \{m \in \text{nodes} \mid (m, n) \in \text{edges}\}$   
 $\text{succ}(m) = \{n \in \text{nodes} \mid (m, n) \in \text{edges}\}$   
 $\text{gen}(n) = \{v_n\}$  if variable  $v$  is defined at  $n$ , otherwise  $\{\}$   
 $\text{kill}(n) = \text{all other definitions of } v \text{ if } v \text{ is defined at } n, \text{ otherwise } \{\}$

Output:  $\text{Reach}(n) = \text{the reaching definitions at node } n$

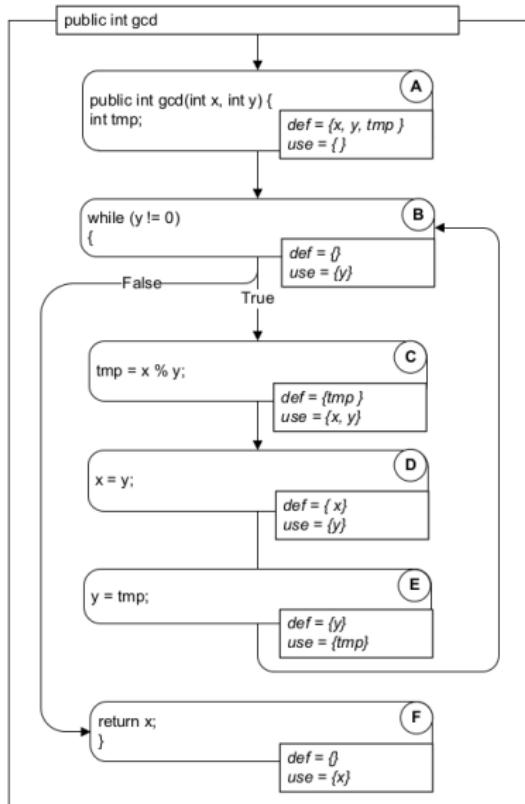
```
for  $n \in \text{nodes}$  loop
     $\text{ReachOut}(n) = \{\}$  ;
end loop;
 $\text{workList} = \text{nodes}$  ;
while ( $\text{workList} \neq \{\}$ ) loop
    // Take a node from worklist (e.g., pop from stack or queue)
     $n = \text{any node in workList}$  ;
     $\text{workList} = \text{workList} \setminus \{n\}$  ;

     $\text{oldVal} = \text{ReachOut}(n)$  ;

    // Apply flow equations, propagating values from predecessors
     $\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$  ;
     $\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$  ;
    if (  $\text{ReachOut}(n) \neq \text{oldVal}$  ) then
        // Propagate changed value to successor nodes
         $\text{workList} = \text{workList} \cup \text{succ}(n)$ 
    end if;
end loop;
```



# All Reaching Definitions



A →  $\emptyset$

B →  $\{x_A, x_D, t_A, t_C, y_E, y_A\}$

C →  $\{x_A, x_D, t_A, t_C, y_E, y_A\}$

D →  $\{x_A, x_D, t_C, y_E, y_A\}$

E →  $\{x_D, y_A, y_E, t_C\}$

F →  $\{x_A, x_D, t_A, t_C, y_A, y_E\}$

$x_A$  is not in E...

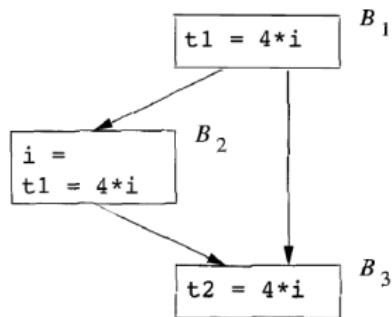
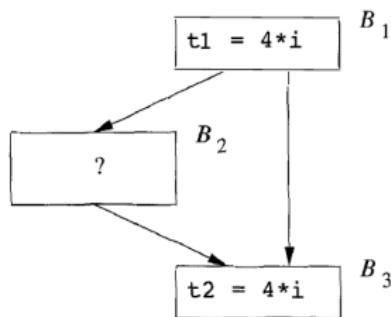


# Available Expressions

- Other uses of the control flow graph: available expressions
  - again, mutuated from compilers: when a given expression can be evaluated just once and stored for later use
  - testing: available expressions should be always tested
- An expression  $E$  is:
  - *generated* when its value is computed
  - *killed* when at least one of the variables involved changes its value
    - not necessarily by assignments, could be a side effect of a function call...
  - *available* at some point  $p$  iff, for all paths  $\pi$  from start to  $p$ ,  $E$  is generated but not subsequently killed in  $\pi$
- Algorithm is very similar to the reaching definitions one:
  - for available expressions, is a forward all-paths analysis
  - for reaching definitions, is a forward any-path analysis



# Available Expressions



# Available Expressions

Statement	Available Expressions
	$\emptyset$
$a = b + c$	$\{b + c\}$
$b = a - d$	$\{a - d\}$
$c = b + c$	$\{a - d\}$
$d = a - d$	$\emptyset$

# Algorithm to Generate All Available Expressions

## Algorithm Available expressions

Input: A control flow graph  $G = (\text{nodes}, \text{edges})$ , with a distinguished root node  $\text{start}$ .

$$\text{pred}(n) = \{m \in \text{nodes} \mid (m, n) \in \text{edges}\}$$

$$\text{succ}(m) = \{n \in \text{nodes} \mid (m, n) \in \text{edges}\}$$

$$\text{gen}(n) = \text{all expressions } e \text{ computed at node } n$$

$$\text{kill}(n) = \text{expressions } e \text{ computed anywhere, whose value is changed at } n;$$

$$\text{kill}(\text{start}) \text{ is the set of all } e.$$

Output:  $\text{Avail}(n)$  = the available expressions at node  $n$

```
for  $n \in \text{nodes}$  loop
    AvailOut( $n$ ) = set of all  $e$  defined anywhere ;
end loop;
workList = nodes ;
while (workList  $\neq \{\}$ ) loop
    // Take a node from worklist (e.g., pop from stack or queue)
     $n$  = any node in workList ;
    workList = workList \ { $n$ } ;
    oldVal = AvailOut( $n$ ) ;
    // Apply flow equations, propagating values from predecessors
    Avail( $n$ ) =  $\bigcap_{m \in \text{pred}(n)}$  AvailOut( $m$ );
    AvailOut( $n$ ) = (Avail( $n$ ) \ kill( $n$ ))  $\cup$  gen( $n$ ) ;
    if ( AvailOut( $n$ )  $\neq$  oldVal ) then
        // Propagate changes to successors
        workList = workList  $\cup$  succ( $n$ )
    end if;
end loop;
```



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



# Algorithm to Generate All Reaching Definitions

## Algorithm *Reaching definitions*

**Input:** A control flow graph  $G = (\text{nodes}, \text{edges})$   
 $\text{pred}(n) = \{m \in \text{nodes} \mid (m, n) \in \text{edges}\}$   
 $\text{succ}(m) = \{n \in \text{nodes} \mid (m, n) \in \text{edges}\}$   
 $\text{gen}(n) = \{v_n\}$  if variable  $v$  is defined at  $n$ , otherwise  $\{\}$   
 $\text{kill}(n) = \text{all other definitions of } v \text{ if } v \text{ is defined at } n, \text{ otherwise } \{\}$

**Output:**  $\text{Reach}(n) = \text{the reaching definitions at node } n$

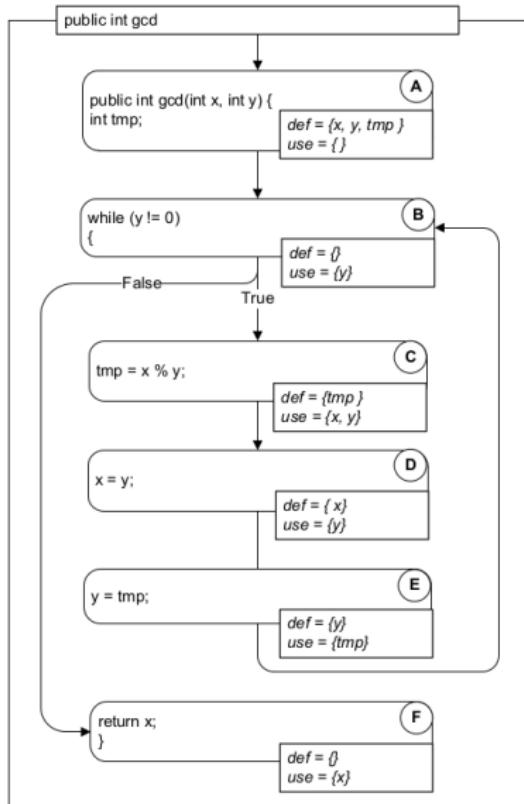
```
for  $n \in \text{nodes}$  loop
     $\text{ReachOut}(n) = \{\}$  ;
end loop;
 $\text{workList} = \text{nodes}$  ;
while ( $\text{workList} \neq \{\}$ ) loop
    // Take a node from worklist (e.g., pop from stack or queue)
     $n = \text{any node in workList}$  ;
     $\text{workList} = \text{workList} \setminus \{n\}$  ;

     $\text{oldVal} = \text{ReachOut}(n)$  ;

    // Apply flow equations, propagating values from predecessors
     $\text{Reach}(n) = \bigcup_{m \in \text{pred}(n)} \text{ReachOut}(m)$  ;
     $\text{ReachOut}(n) = (\text{Reach}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$  ;
    if (  $\text{ReachOut}(n) \neq \text{oldVal}$  ) then
        // Propagate changed value to successor nodes
         $\text{workList} = \text{workList} \cup \text{succ}(n)$ 
    end if;
end loop;
```



# All Available Expressions



A → ∅

B → ∅

C → ∅

D → ∅

E → ∅

F → ∅



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione e Matematica

# Control Dependence

- *Control dependence tree*
  - models the effects of conditional branches
  - nodes are statements, but again granularity may change
  - to define edges, the notion of *dominators* is needed
  - a node  $n$  is dominated by node  $m$  iff, for all paths  $\pi$  from the root to  $n$ ,  $m$  is also in  $\pi$
  - the (unique) *immediate dominator* of  $n$  is the closest dominator of  $n$ 
    - i.e., with the minimum path to reach  $n$
    - also stated as: the dominator of  $n$  which does not dominate any other dominator of  $n$
  - *dominator tree*: there is an edge  $(s, t)$  iff  $s$  is the immediate dominator of  $t$ 
    - for all reachable nodes there is exactly one immediate dominator
  - *post-dominators* (also *forward-dominators*): same definition, but in the reverse graph
    - an exit node must be present: all paths from there to the exit...

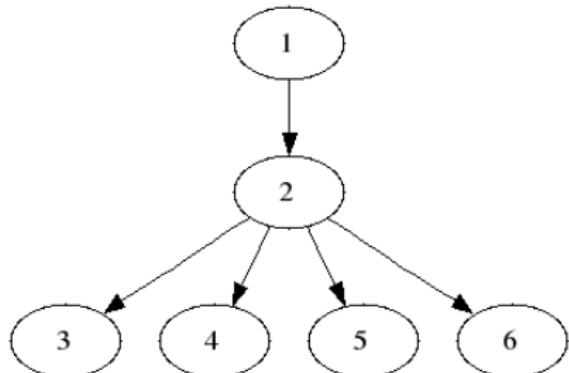
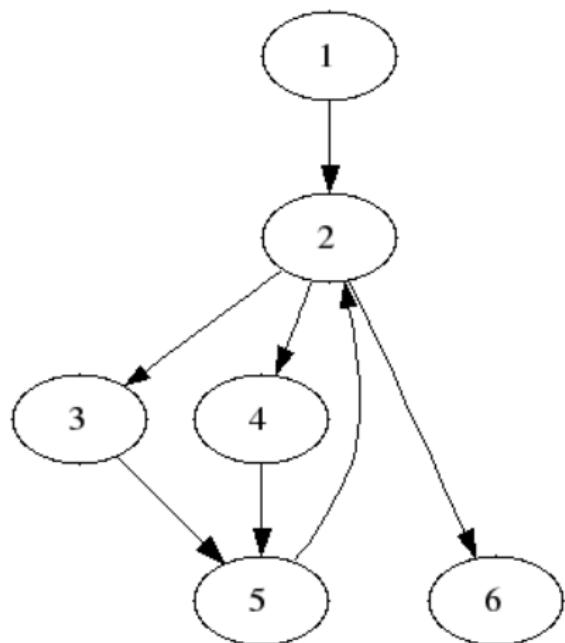


UNIVERSITÀ  
DEGLI STUDI  
DELLA SVIZZERA  
ITALIANA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Dominators



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione e  
Matematica

# Control Dependence

- Back to the control dependence tree: given nodes  $s, t$ , we have that  $(s, t)$  is an edge iff  $t$  is *control dependent* on  $s$
- To define when  $t$  is control dependent on  $s$ , the following holds:
  - $t$  is reached on all (finite!) execution paths
    - then,  $t$  is control dependent on the root only
    - it may actually be the root itself
  - $t$  is reached on some but not all execution paths; then for  $s$  the following must hold:
    - the outgoing degree of  $s$  in the CFG is at least 2
    - one of the successors of  $s$  in the CFG is post-dominated by  $t$
    - ( $t$  may also be a direct successor of  $s$ )
    - $s$  is not post-dominated by  $t$
- Complexity is  $V^3$



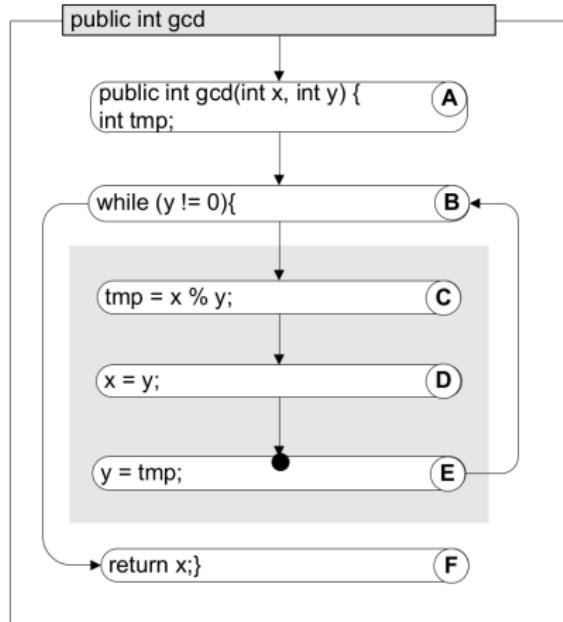
UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
dell'Informazione  
e Matematica

# Control Dependence

Proof that  $B$  is control dependent on  $E$

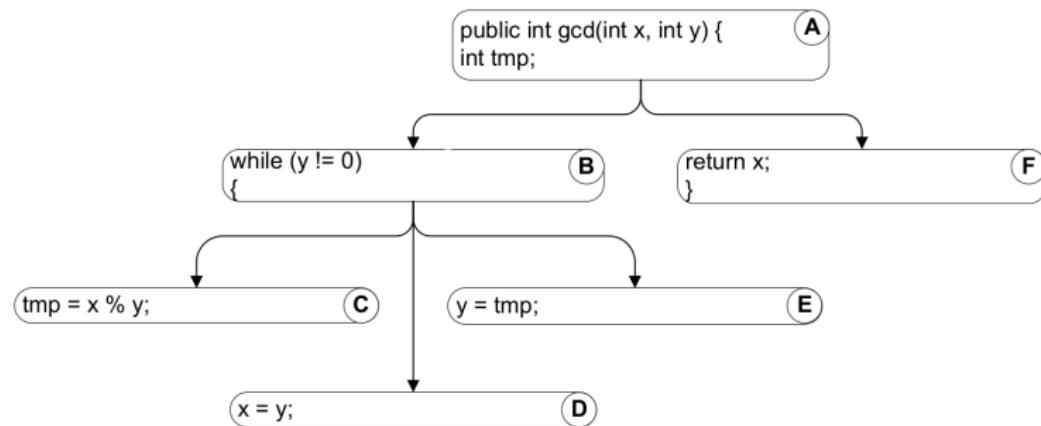


Gray region: nodes post-dominated by  $E$   
Node  $B$  has successors both within and outside the gray region  
 $\rightarrow E$  is control-dependent on  $B$



# Control Dependence

Full control dependence tree



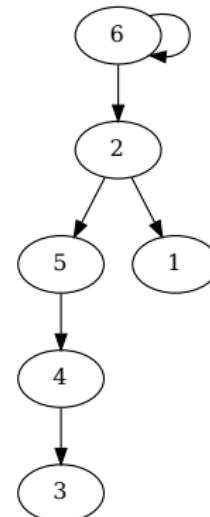
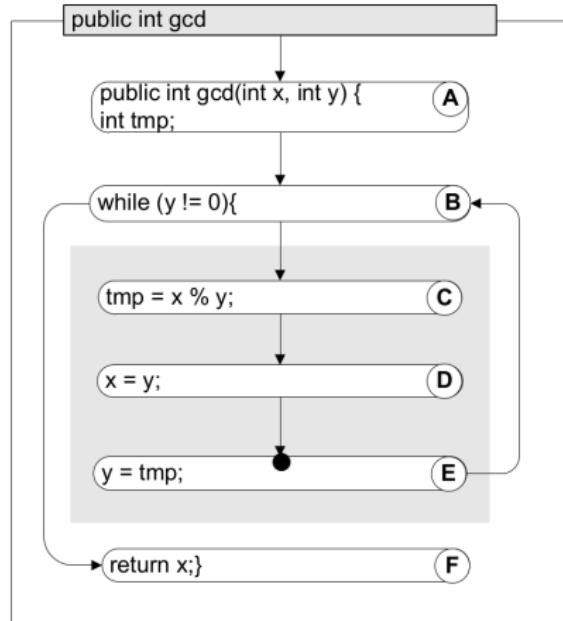
UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Control Dependence: Post Dominators

Full immediate post dominators tree



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica

# Control Dependence Tree, Characteristics

- There may be edges going out by two types of nodes only:
  - the root (A in the example)
  - nodes in which a choice is done (B in the example)
    - the outgoing degree in the original CFG is at least 2
- If a node (like F) is always reached, then the only dependence is in executing the unit, i.e. on the root
- If a node (like B) makes a choice, then all its “forced” content (without further branches) is control dependent on B



DISIM  
Dipartimento di Ingegneria  
dell'Informazione e Matematica

# Data Flow Analysis with Arrays and Pointers

- Easy to perform data flow analysis on single variables
- When considering pointers and/or arrays, many difficulties arise
- Difficulty 1: definition-use on an array referenced by variables
  - e.g.:  $a[i] = 1; k = a[j];$  is a definition-use pair iff  $i == j$
  - too difficult to determine if such a condition is always true, always false, or sometimes true and sometimes false
- Difficulty 2: aliases obtained by full array assignment
  - e.g.,  $b = a; a[2] = 42; i = b[2];$  is a definition-use pair (or triple?) in Java



## Difficulty 3: Arguments Passing

```
fromCust == toCust? fromHome == fromWork? toHome ==  
toWork?
```

```
public void transfer (CustInfo fromCust, CustInfo toCust) {
```

```
    PhoneNum fromHome = fromCust.gethomePhone();  
    PhoneNum fromWork = fromCust.getworkPhone();
```

```
    PhoneNum toHome = toCust.gethomePhone();  
    PhoneNum toWork = toCust.getworkPhone();
```



UNIVERSITÀ  
DEGLI STUDI  
DELL'AQUILA



DISIM  
Dipartimento di Ingegneria  
e Scienze dell'Informazione  
e Matematica