MILCOM 2016

SECURE COMMUNICATIONS AT THE SPEED OF CYBER

Taint Analysis

Suresh C. Kothari
Richardson Professor
Department of Electrical and Computer Engineering

Ben Holland, Iowa State University

Acknowledgement: Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080

BALTIMORE, MD • NOVEMBER 1–3, 2016

Module Overview

- Data Dependence
- Control Dependence
- Taint Analysis

Motivating Example (1)

Example:

```
1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t);
```

 detected failure

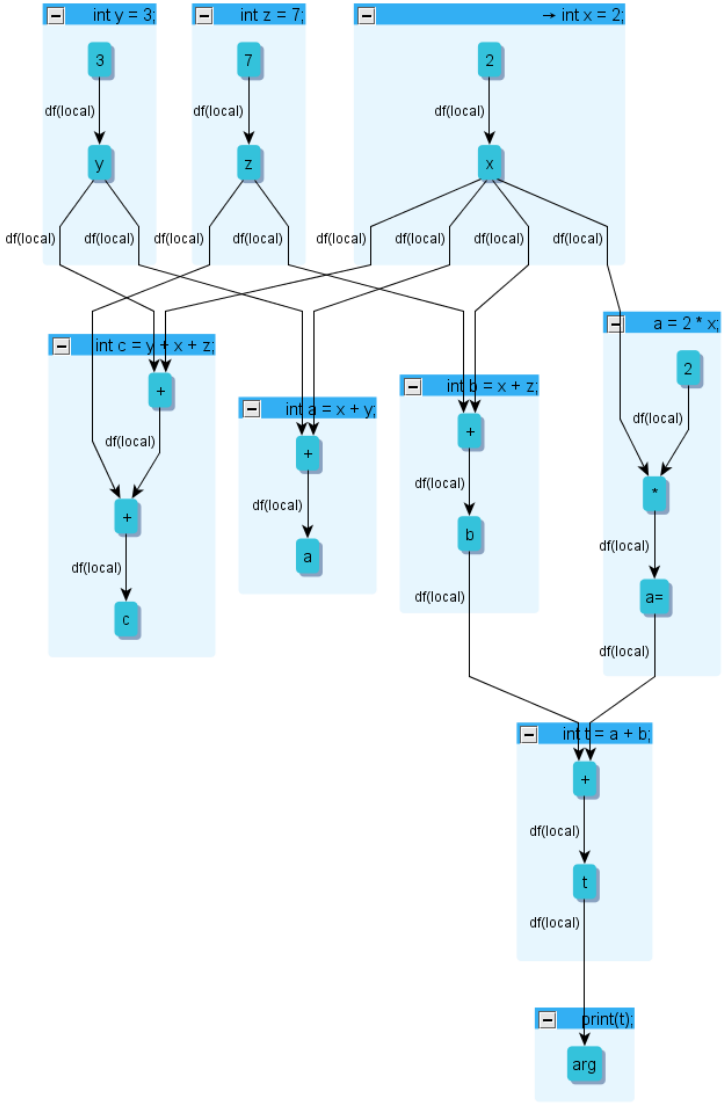
What lines must we consider if the value of *t* printed is incorrect?

Data Flow Graph

Example:

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t); ← detected failure

- Let each assignment represent an edge from the RHS to the LHS.
- Consider primitive operations as temporary intermediate assignments.
- Record the corresponding line number for each statement.



Data Flow Slice

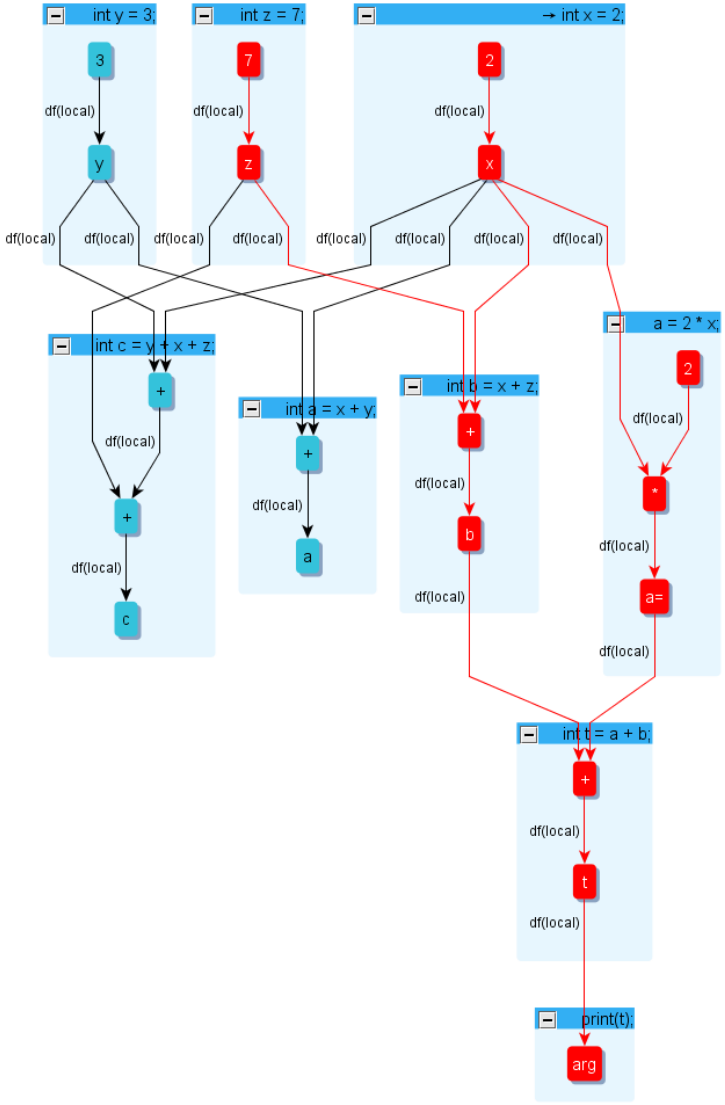
Example:

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t);

Relevant lines:
1,3,5,6,8

← detected failure

- The relevant statements are those that are reachable in a backwards traversal from the argument passed to *print.*



Def-Use (DU) Chain

The *backward dataflow slice* is constructed by applying DU chains.

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. Print t;

Statement 8 *defines* **t** and *uses* **a** and **b**

Equivalently, *write-set*(8) = {**t**} and *read-set*(8) = {**a**, **b**}

A *DU chain* consists of a *variable definition*, and *all the uses* of that variable reachable from the definition.

Statement 4 and 6 provide definitions of the variable **a**.

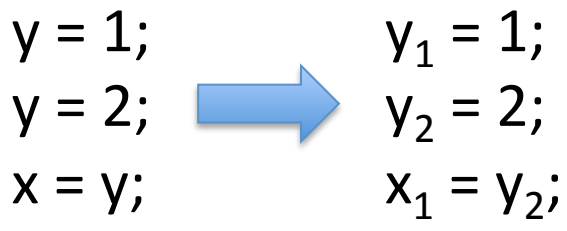
The definition 6 reaches the use of **a** at statement 8

The definition 4 is *killed* by the definition 6, thus it *cannot* reach the use at 8.

How can we have multiple definitions reaching the same use?

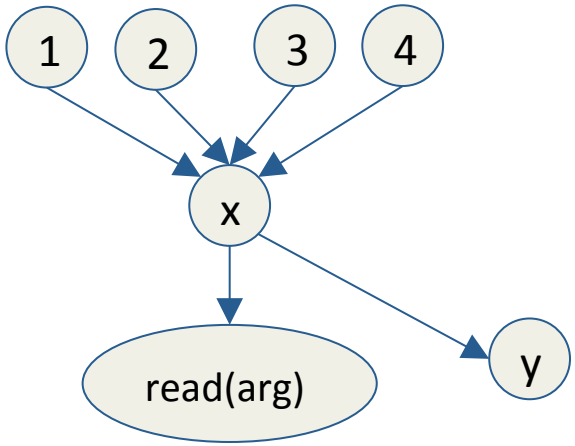
Static Single Assignment (SSA) Form

- Requirements:
  - Each variable may only be assigned exactly once.
  - Every variable is defined before its use.



Code Transformation (before): Static Single Assignment Form

1. x = 1;
2. x = 2;
3. if(condition)
4. x = 3;
5. read(x);
6. x = 4;
7. y = x;

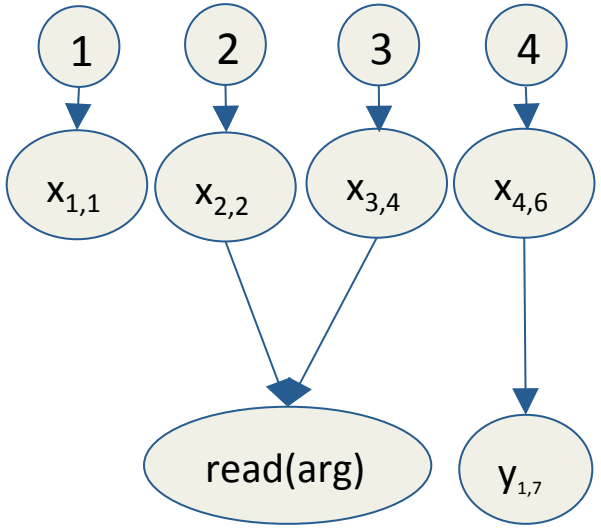


Resulting graph when statement ordering is not considered.

Code Transformation (after): Static Single Assignment Form

1. x = 1;
2. x = 2;
3. if(condition)
4. x = 3;
5. read(x);
6. x = 4;
7. y = x;

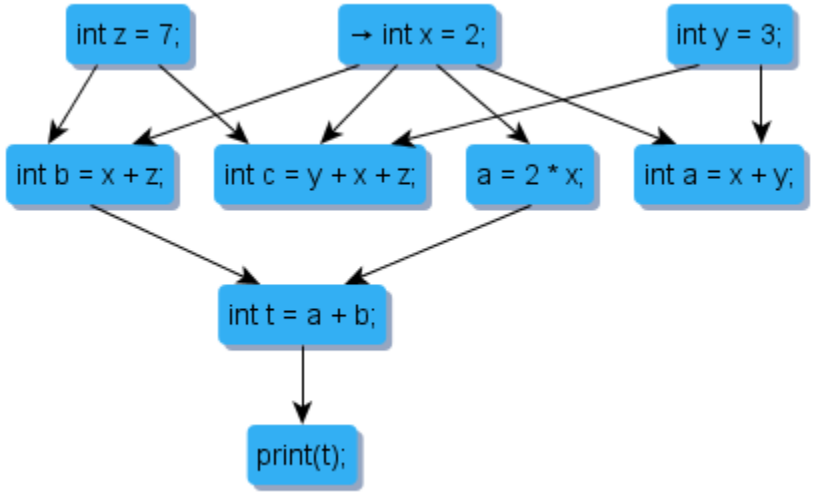
1. $x_{1,1}$ = 1;
2. $x_{2,2}$ = 2;
3. if(condition)
4. $x_{3,4}$ = 3;
5. read($x_{2,2}$,$_{3,4}$);
6. $x_{4,6}$ = 4;
7. $y_{1,7}$ = $x_{4,6}$;



Note: <Def#,Line#>

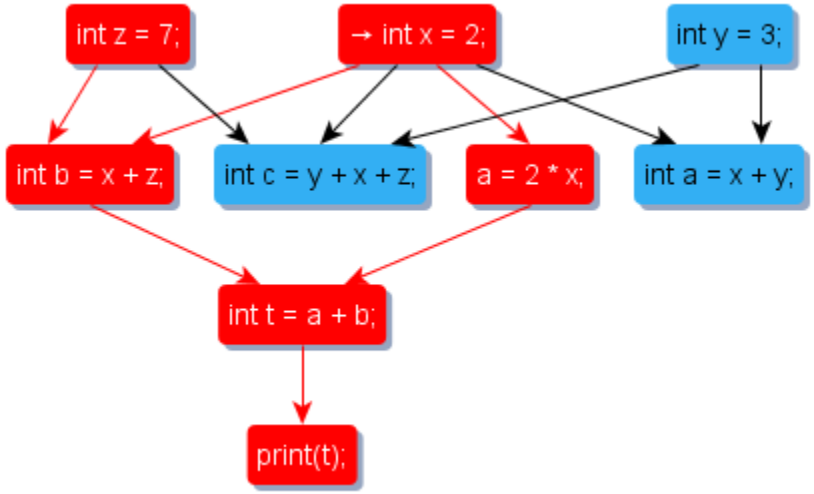
Data Dependence Graph (DDG)

- Note that we could summarize data flow on a per statement level.
- This graph is called a Data Dependence Graph (DDG)



Data Dependence Graph (DDG)

- Note that we could summarize data flow on a per statement level.
- This graph is called a Data Dependence Graph (DDG)



Data Dependence Slicing

- Reverse Data Dependence Slice
  - What statements influence the assigned value in this statement?
- Forward Data Dependence Slice
  - What statements could the assigned value in this statement influence?

Motivating Example (2)

Example:

```
1. i = readInput();
2. if(i == 1)
3.    print("test");
   else
4.    i = 1;
5. print(i);
6. return; // terminate program
```


detected failure

What lines must we consider if the program always prints "1"?

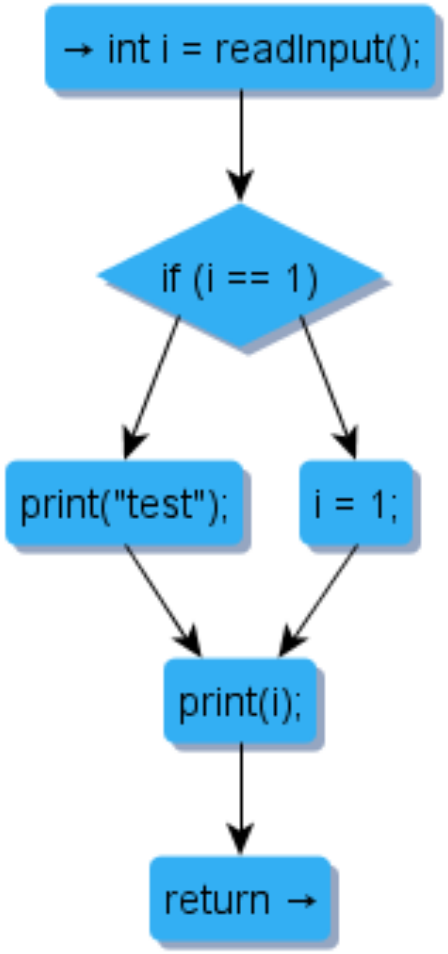
Control Flow Graph

Example:

```
1. i = readInput();
2. if(i == 1)
3.    print("test");
   else
4.    i = 1;
5. print(i);
6. return; // terminate program
```

Relevant lines:
1,2,4

detected failure



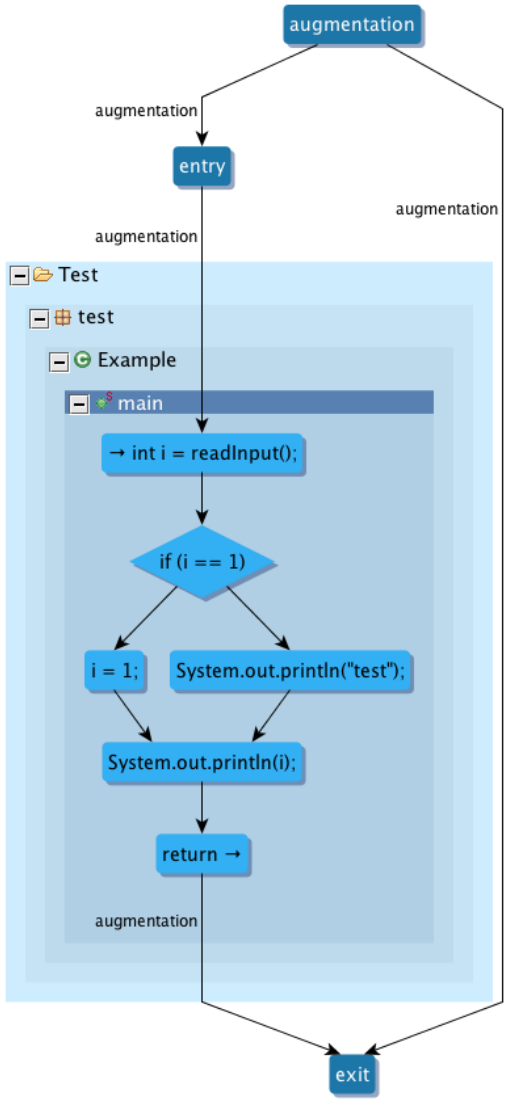
- Strategy:
  - Start at line 5. Line 4 may not execute depending on condition in line 2. The "test" string is not printed so line 3 was not executed, meaning line 4 must have been executed. The condition on line 2 depends on input from line 1.
- A *Control Flow Graph* (CFG) represents the possible sequential execution orderings of each statement in a program.

Control Dependence Graph (CDG)

- If a statement X determines whether a statement Y can be executed then statement Y is control dependent on X.
  - Classic algorithm: Ferrante, Ottenstein, and Warren (FOW) algorithm

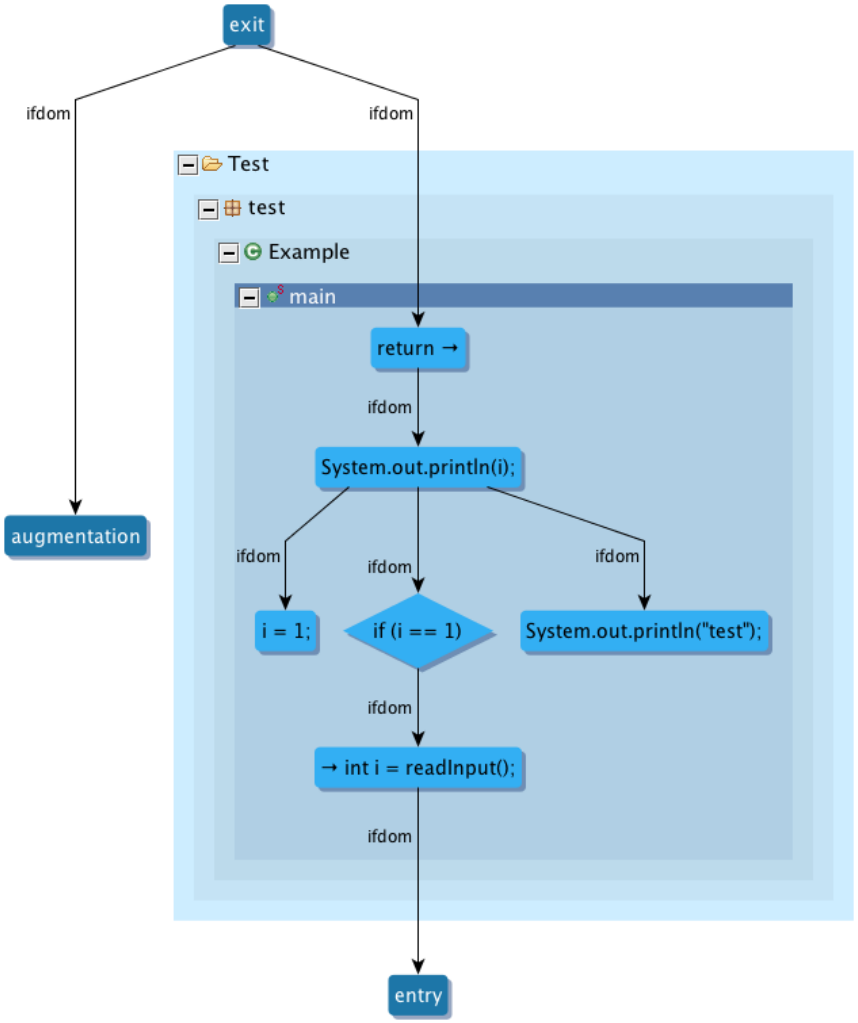
Building a CDG (1)

- First augment the CFG with a single “entry” node and single “exit” node.
- Create an “augmentation” node which has the “entry” and “exit” nodes as children.



Building a CDG (2)

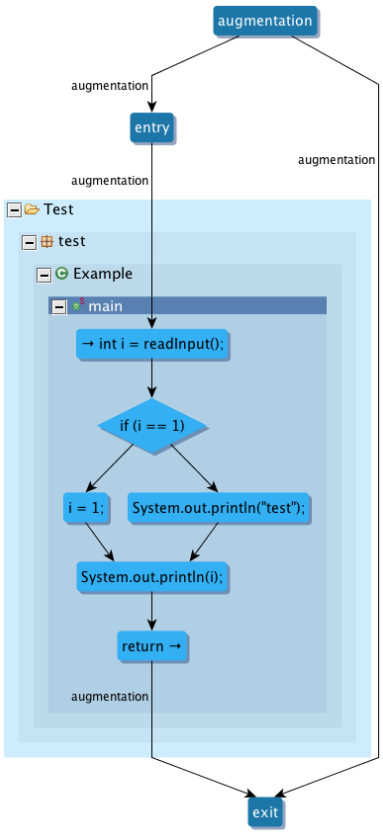
- X *dominates* Y if every path from the entry node to Y must go through X
- A *dominator tree* is a tree where each node's children are those nodes it immediately dominates
- Compute a forward dominance tree (i.e. post-dominance analysis) of the augmented CFG



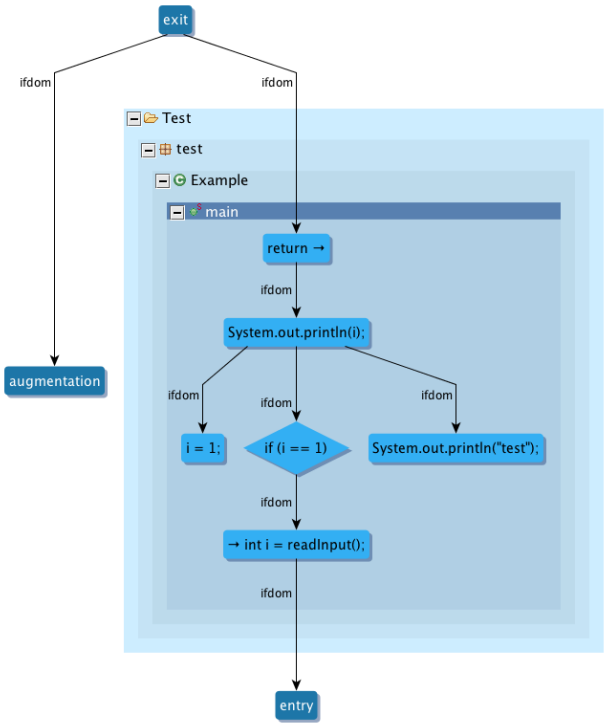
Building a CDG (3)

- The *least common ancestor* (LCA) of two nodes X and Y is the deepest tree node that has both X and Y as descendants
- For each edge (X → Y) in CFG, find nodes in FDT from LCA(X,Y) to Y, which are *control dependent* on X.
  - Exclude LCA(X,Y) if LCA(X,Y) is not X

Augmented CFG (ACFG)



Forward Dominance Tree (FDT)



Building a CDG (4)

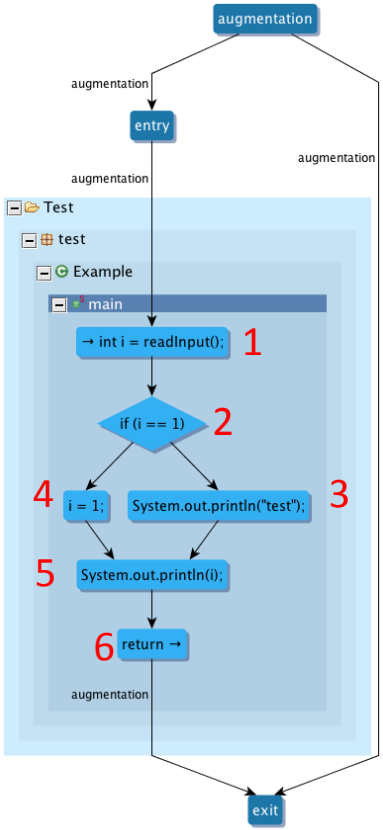
Edge X→Y in ACFG	LCA(X,Y) in FDT	FDT Nodes Between(LCA, Y)
1 → 2	2	~~2~~
2 → 3	5	~~5~~, 3
2 → 4	5	~~5~~, 4
4 → 5	5	~~5~~
3 → 5	5	~~5~~
5 → 6	6	~~6~~

Note: Remove LCA(X,Y) if LCA(X,Y) != X

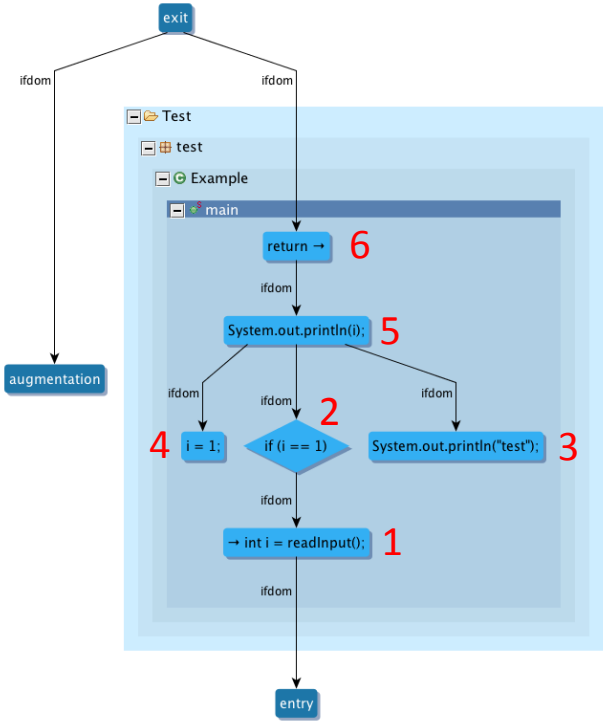
Example:

1. i = readInput();
2. if(i == 1)
3. print("test");
   else
4. i = 1;
5. print(i);
6. return; // terminate program

Augmented CFG (ACFG)



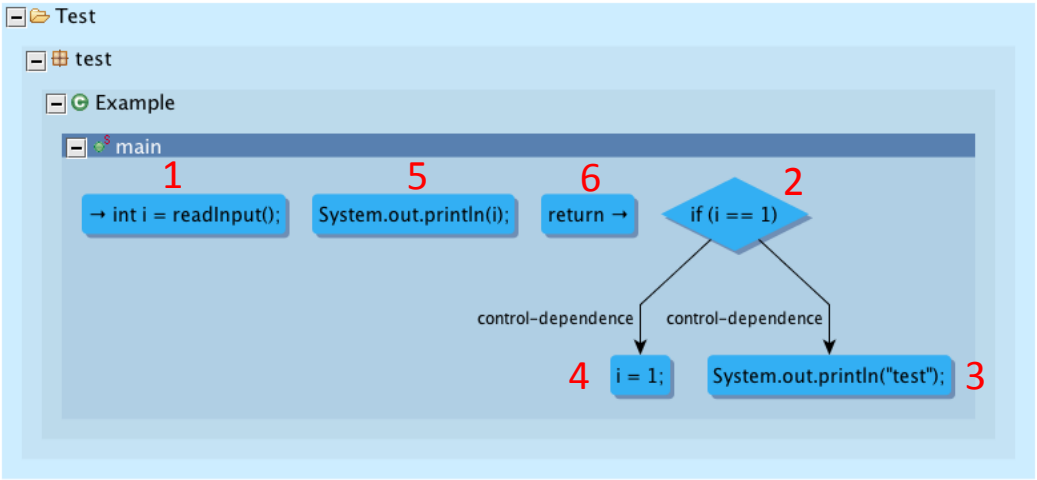
Forward Dominance Tree (FDT)



Control Dependence Graph

Edge X→Y in ACFG	LCA(X,Y) in FDT	FDT Nodes Between(LCA, Y)
1 → 2	2	~~2~~
2 → 3	5	~~5~~, 3
2 → 4	5	~~5~~, 4
4 → 5	5	~~5~~
3 → 5	5	~~5~~
5 → 6	6	~~6~~

FDT Nodes Between(LCA, Y) are *Control Dependent* on X.



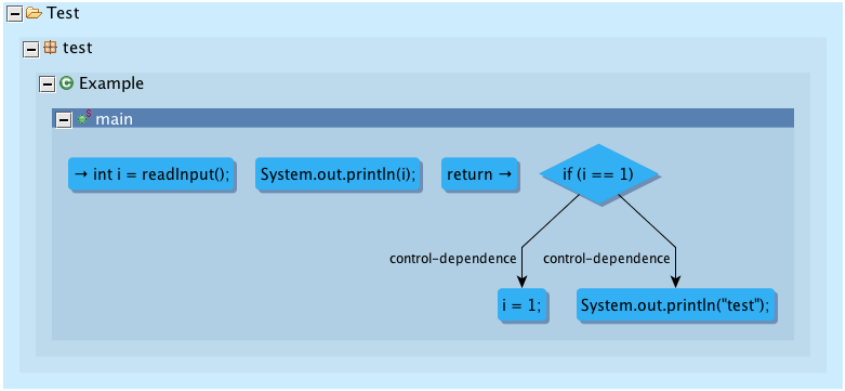
Control Dependence Slicing

- Reverse Control Dependence Slice
  - What statements does this statement's execution depend on?
- Forward Control Dependence Slice
  - What statements could execute as a result of this statement?

Control Dependence Slicing

Example:

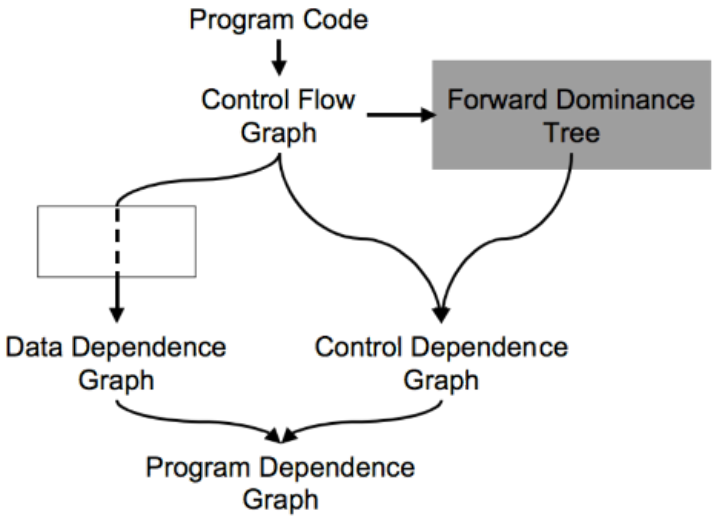
1. i = readInput();
2. if(i == 1)
3. print("test");
   else
4. i = 1;
5. print(i); ← detected failure
6. return; // terminate program



What lines must we consider if the value of *t* printed is incorrect?

Is the CDG sufficient to answer the question?

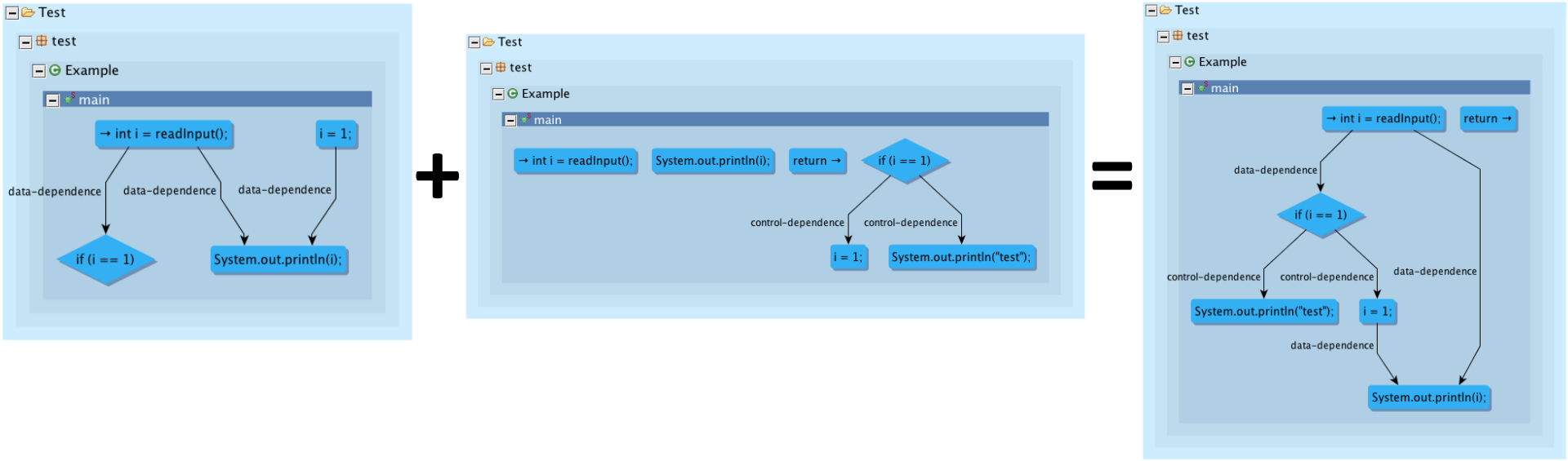
Program Dependence Graph (PDG)



- Both DDG and CDG nodes are statements
- The union of a DDG and the CDG is a PDG

Program Dependence Graph (PDG)

DDG + CDG = PDG



Program Slicing

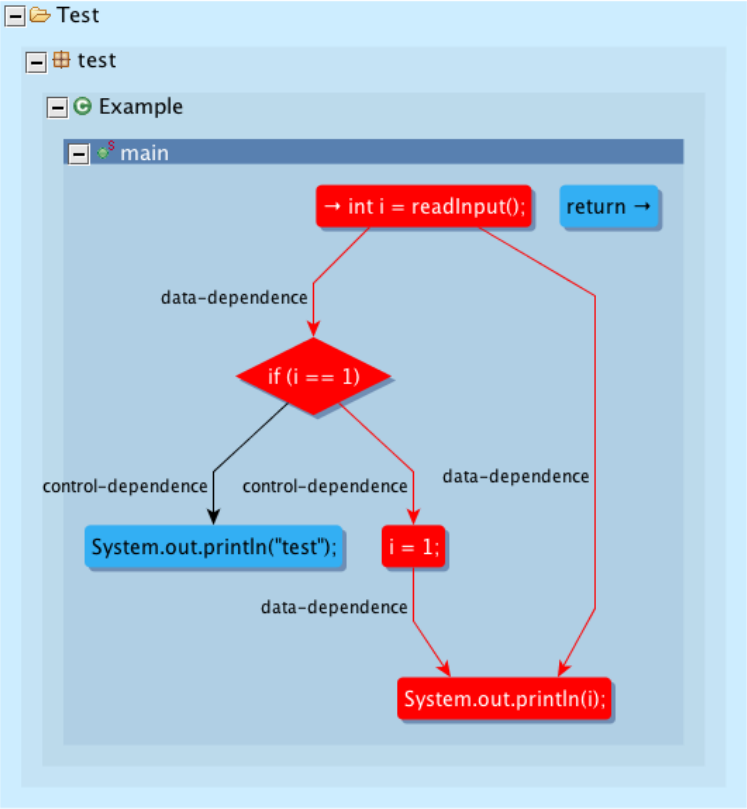
- Reverse Program Slice
  - What statements does this statement's execution depend on?
- Forward Program Slice
  - What statements could execute as a result of this statement?
  - This is also known as "impact analysis"

Program Slicing

Example:

1. i = readInput();
2. if(i == 1)
3. print("test");
   else
4. i = 1;
5. print(i); ← detected failure
6. return; // terminate

program



What lines must we consider if the value of *t* printed is incorrect?

Implicit Data Flow

```java
/**
 * A toy example of laundering data through "implicit dataflow paths"
 * The launder method uses the input data to reconstruct a new result
 * with the same value as the original input.
 *
 * @author Ben Holland
 */
public class DataflowLaunder {

        public static void main(String[] args) {
                String x = "1010";
                String y = launder(x);
                System.out.println(y + " is a laundered version of " + x);
        }

        public static String launder(String data){
           String result = "";
           for(char c : data.toCharArray()){
             if(c == '0')
                result += '0';
             else
                result += '1';
           }
           return result;
        }

}
```

DataFlowLaunder.java hosted with ♥ by GitHub view raw

Motivating Example (3)

How can we track the flow of data from the source (*x*) to the sink (*y*)?

Taint Analysis

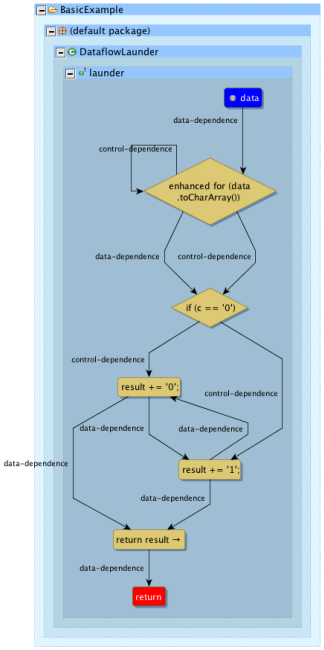
- Taint can be characterized as a forward program slice intersected with a reverse program slice (between traversal)
  - The forward traversal starts from a *source*
  - The reverse traversal starts from a *sink*
- If a path exists from *source* to *sink* then the source *taints* the sink.

Taint Analysis

DataflowLaunder.java

```
1
2  /**
3   * A toy example of laundering data through "implicit dataflow paths"
4   * The launder method uses the input data to reconstruct a new result
5   * with the same value as the original input.
6   *
7   * @author Ben Holland
8   */
9  public class DataflowLaunder {
10
11     public static void main(String[] args) {
12         String x = "1010";
13         String y = launder(x);
14         System.out.println(y + " is a laundered version of " + x);
15     }
16
17     public static String launder(String data){
18        String result = "";
19        for(char c : data.toCharArray()){
20          if(c == '0')
21             result += '0';
22          else
23             result += '1';
24        }
25        return result;
26     }
27
28 }
```

Taint Graph



Problems | Javadoc | Declaration | Console | Progress | Atlas Shell (shell) | Error Log | Analysis Keys | Element Detail View | Call Hierarchy

Atlas Shell (Project: shell)

```
var taint = new com.ensoftcorp.open.slice.analysis.TaintGraph(source, sink)

taint: com.ensoftcorp.open.slice.analysis.TaintGraph = com.ensoftcorp.open.slice.analysis.TaintGraph@13df7ce4

show(taint.getGraph(), taint.getHighlighter(), title="Taint Graph")
```

Evaluate: <type an expression or enter ":help" for more information>