```java
public class Puzzle14 {

    public static void main(String[] args) {
        char x = 'X';
        int i = 0;
        System.out.print(true ? x : 0);
        System.out.print(false ? i : x);
    }

}
```

Related Works

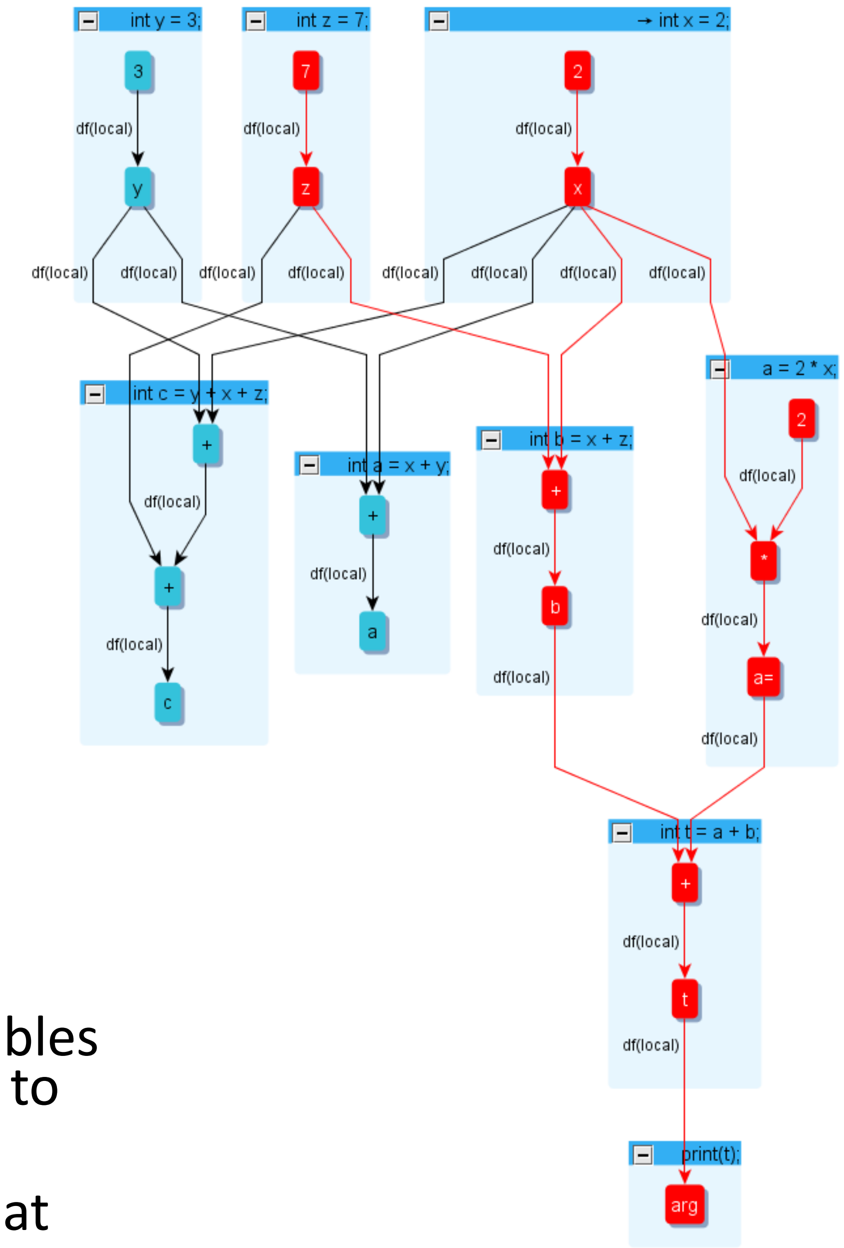
- Mark Weiser. 1981. Program slicing. In Proceedings of the 5th international conference on Software engineering (ICSE '81). IEEE Press, Piscataway, NJ, USA, 439-449.
- Ferrante, Jeanne, Karl J. Ottenstein, and Joe D. Warren. "The program dependence graph and its use in optimization." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9.3 (1987): 319-349.

Data Flow Graph (DFG)

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

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

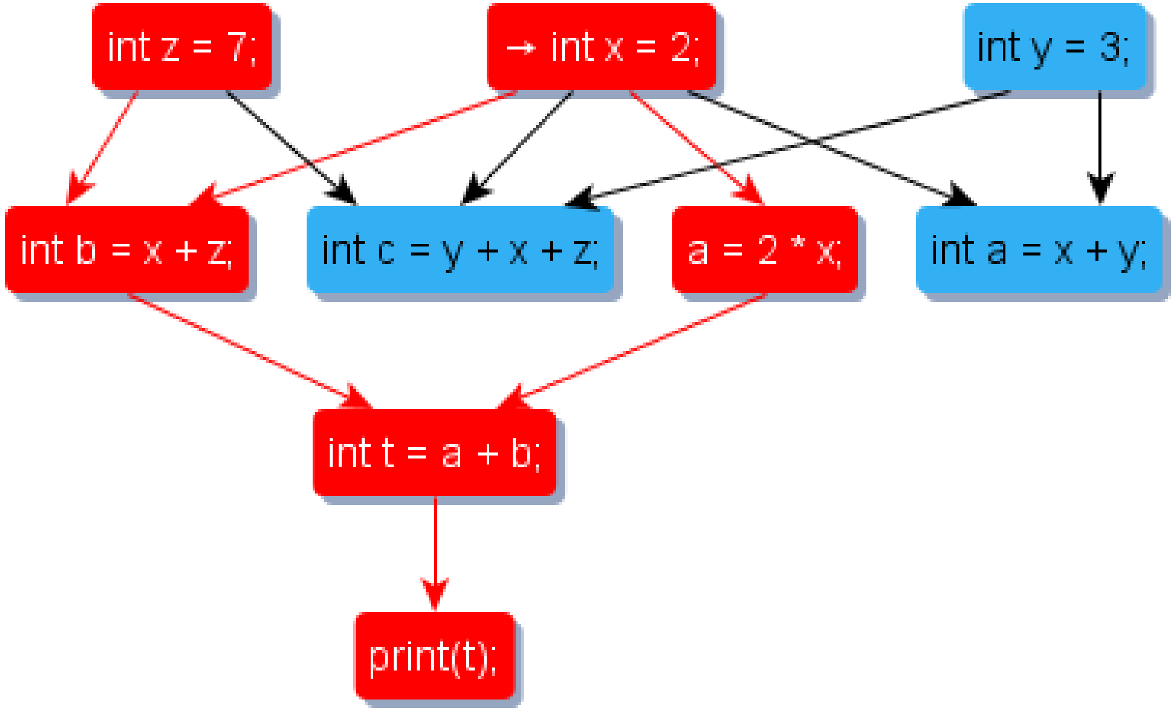


What lines must we consider if the value of *t* printed is incorrect?

- A *Data Flow Graph* creates a graph of primitives and variables where each assignment represents an edge from the RHS to the LHS of the assignment
- The *Data Flow Graph* represents global data dependence at the operator level (the atomic level) [FOW87]

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)
- DDG dependences represent only the *relevant* data flow relationships of a program [FOW87]

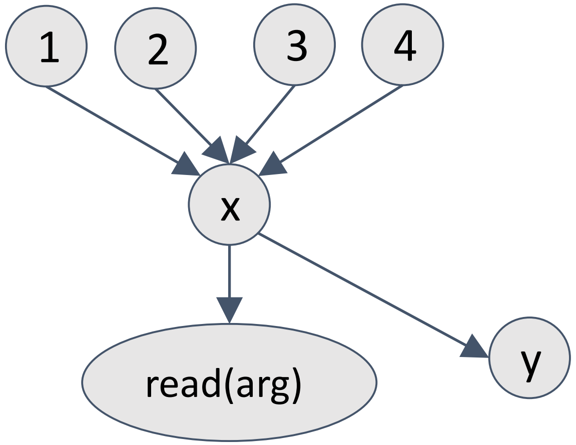


Data Dependence Slicing

- 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?

Code Transformation (before – flow insensitive): 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;
```

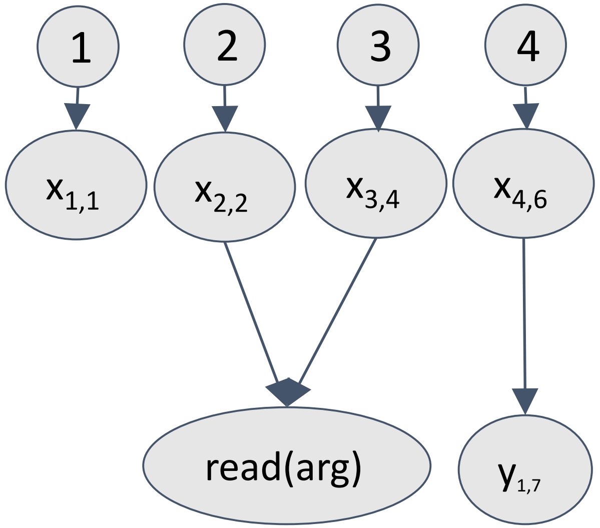


Resulting graph when statement ordering is not considered.

Code Transformation (after – flow sensitive): 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#>

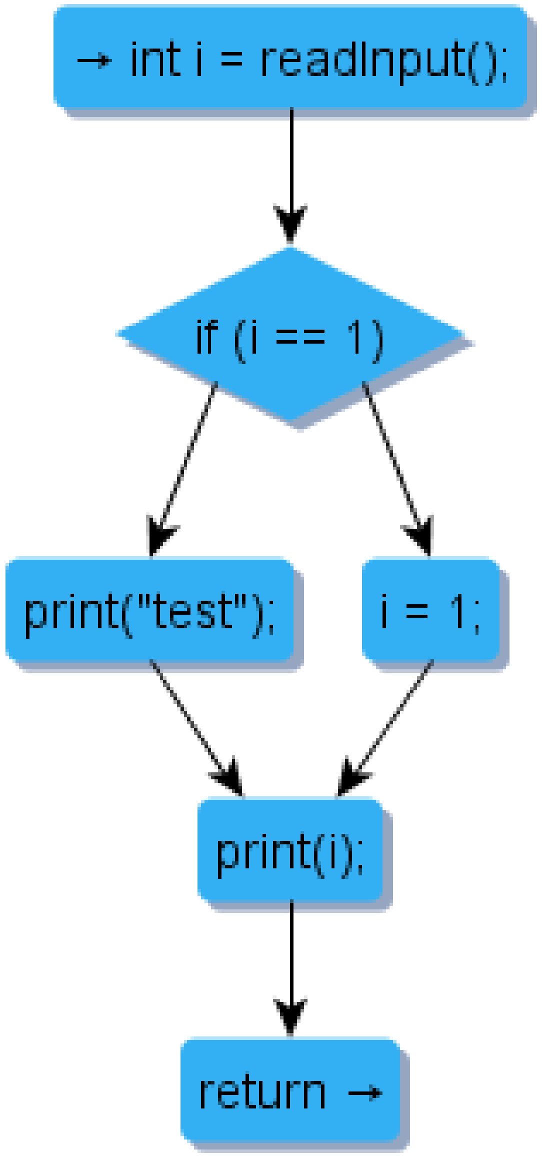
Control Flow Graph (CFG)

Example:

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

Relevant lines:
1,2,4

detected failure

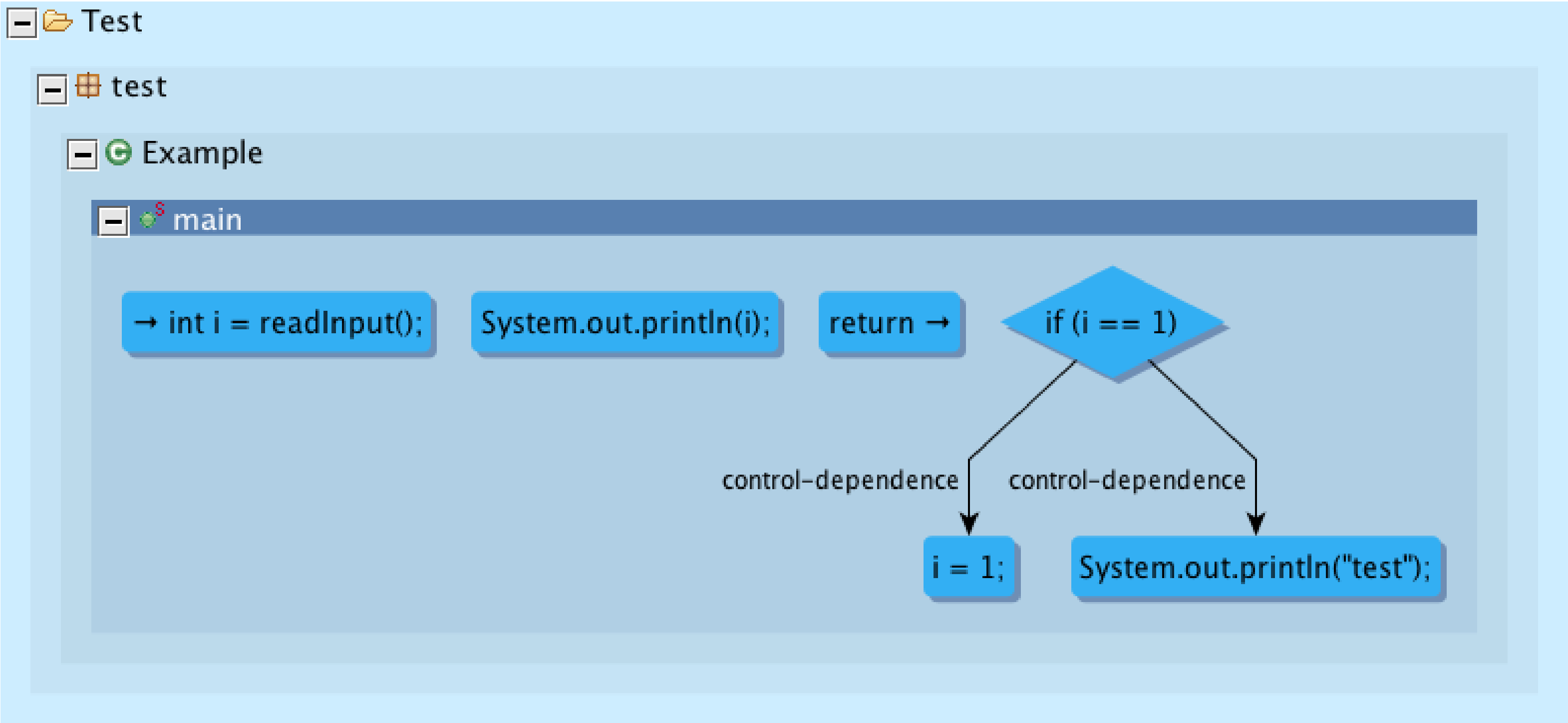


What lines must we consider if the value of *i* printed is incorrect?

- A *Control Flow Graph* (CFG) represents the possible sequential execution orderings of each statement in a program
- Data flow influences control flow, so this graph is not enough

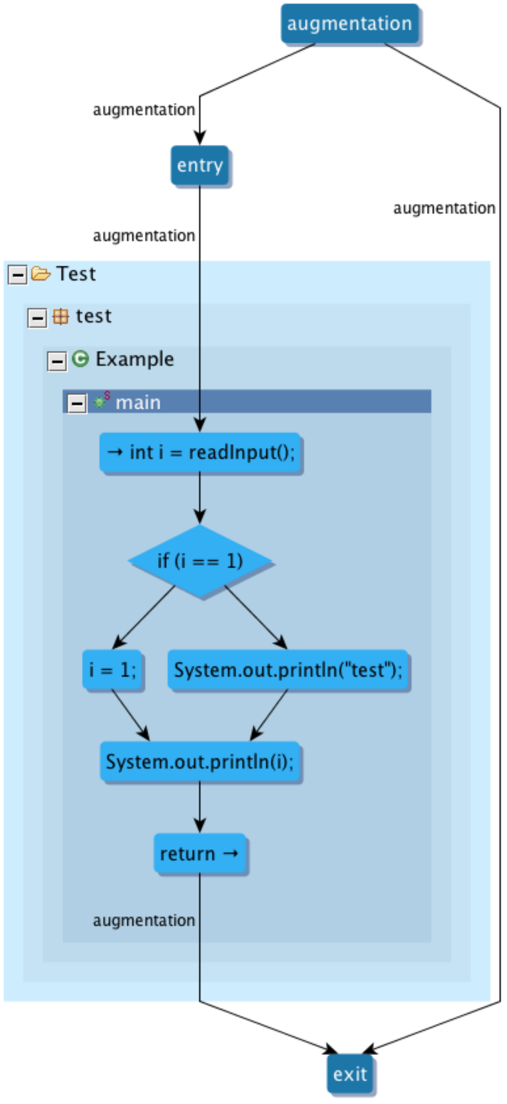
Control Dependence Graph (CDG)

- If a statement X determines whether a statement Y can be executed then statement Y is *control dependent* on X
- Control dependence exists between two statements, if a statement directly controls the execution of the other statement [FOW87]



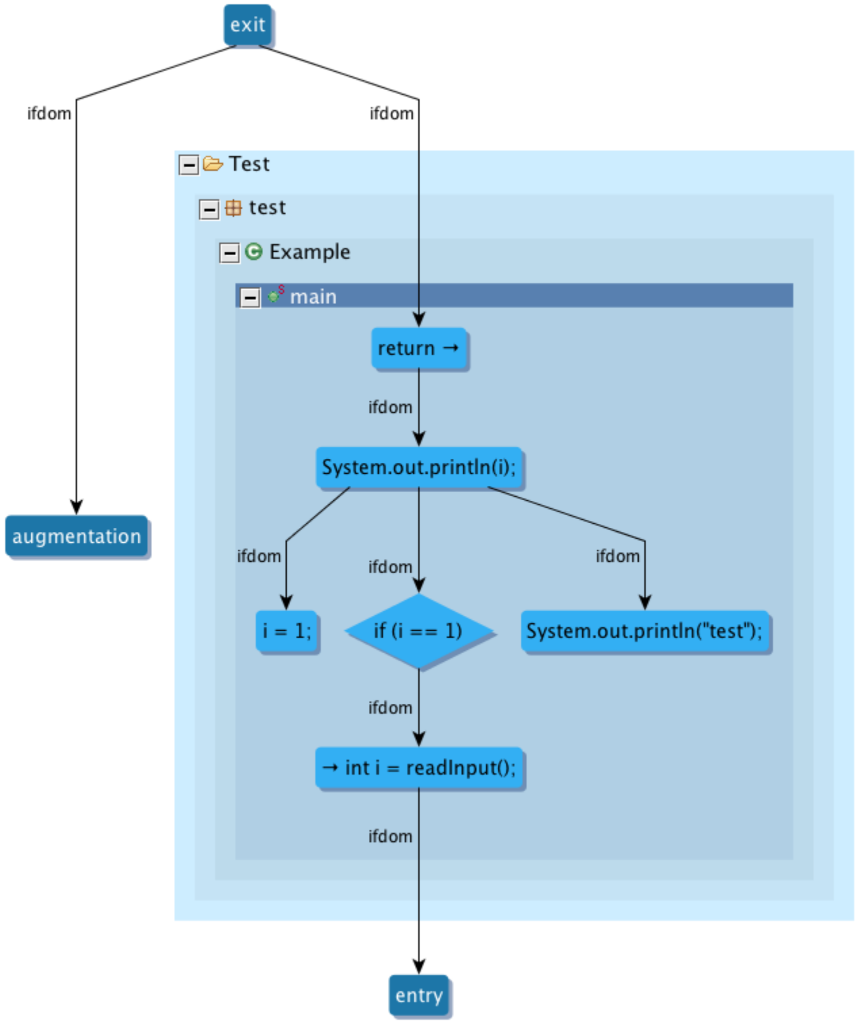
Building a CDG (1)

- 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)

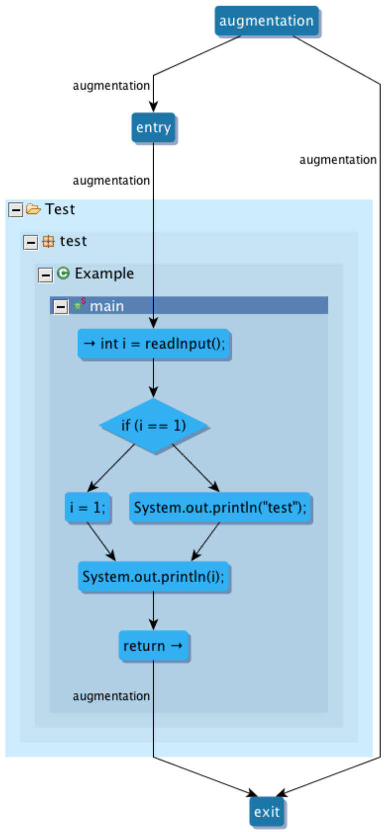
- X *dominates* Y if every path from the entry node to Y must go through X
- A *dominator tree* is a tree where each node's children are those nodes it immediately dominates
- Compute a forward dominance tree (i.e. post-dominance analysis) of the augmented CFG



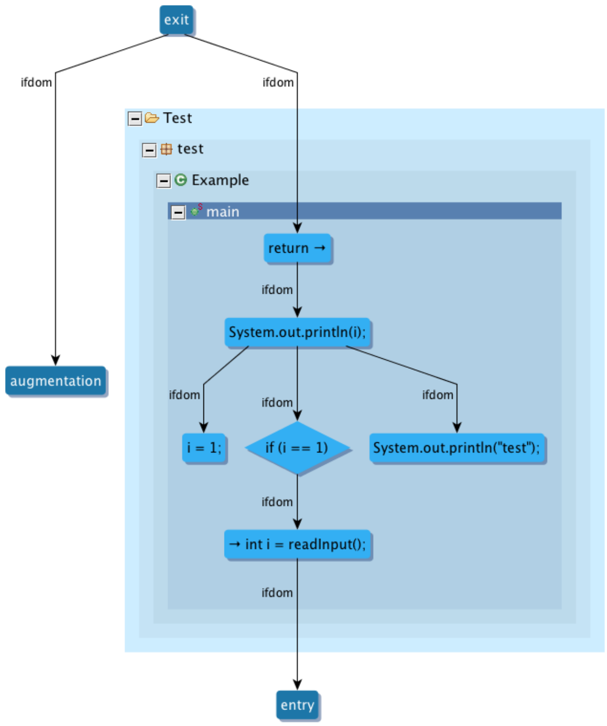
Building a CDG (3)

- 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)

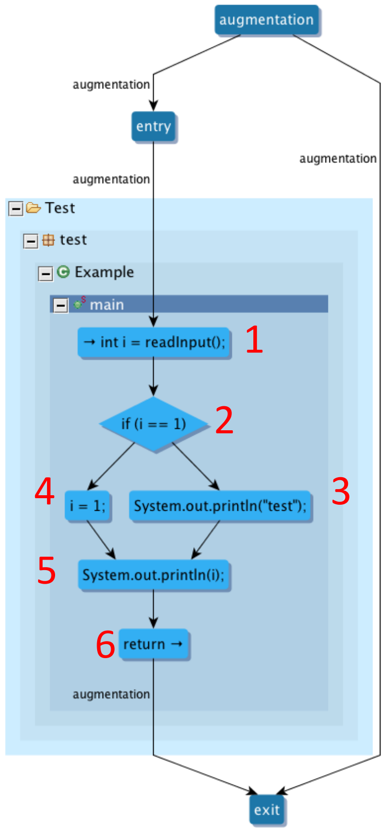
Edge X→Y in ACFG	LCA(X,Y) in FDT	FDT Nodes Between(LCA, Y)
1 → 2	2	~~2~~
2 → 3	5	~~5~~, 3
2 → 4	5	~~5~~, 4
4 → 5	5	~~5~~
3 → 5	5	~~5~~
5 → 6	6	~~6~~

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

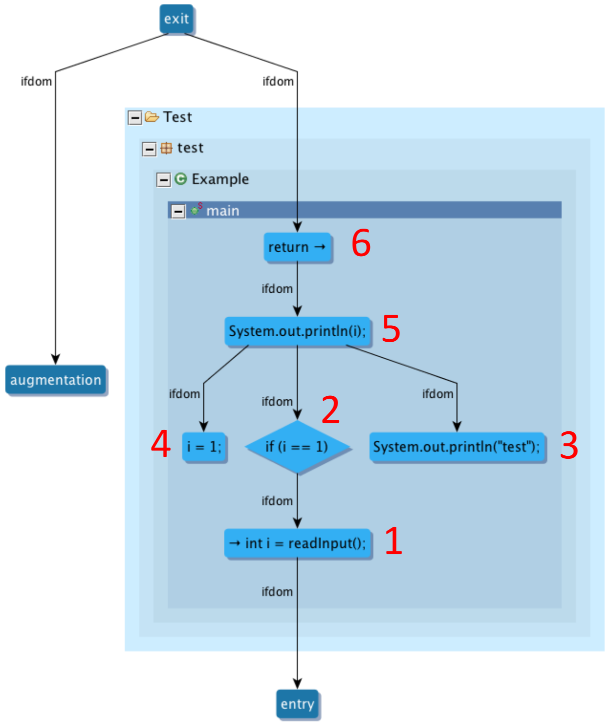
Example:

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

Augmented CFG (ACFG)



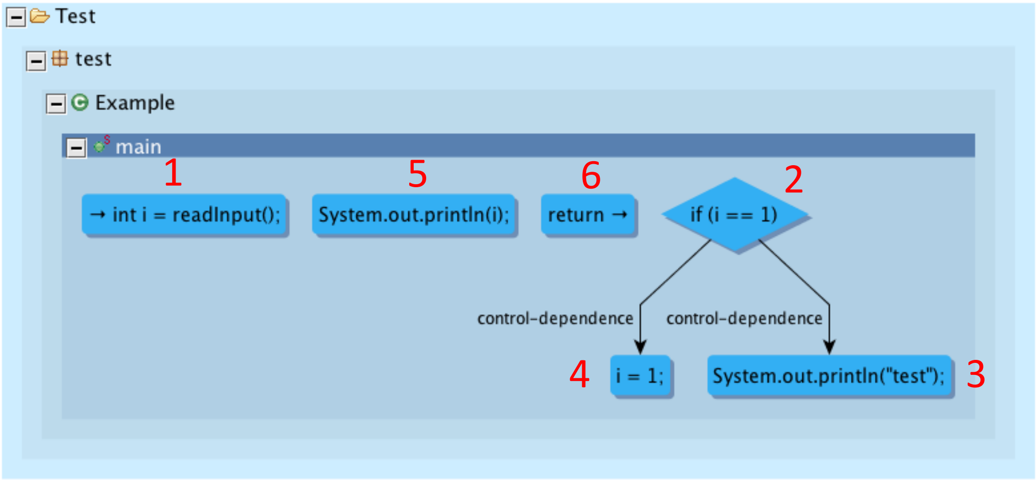
Forward Dominance Tree (FDT)



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.



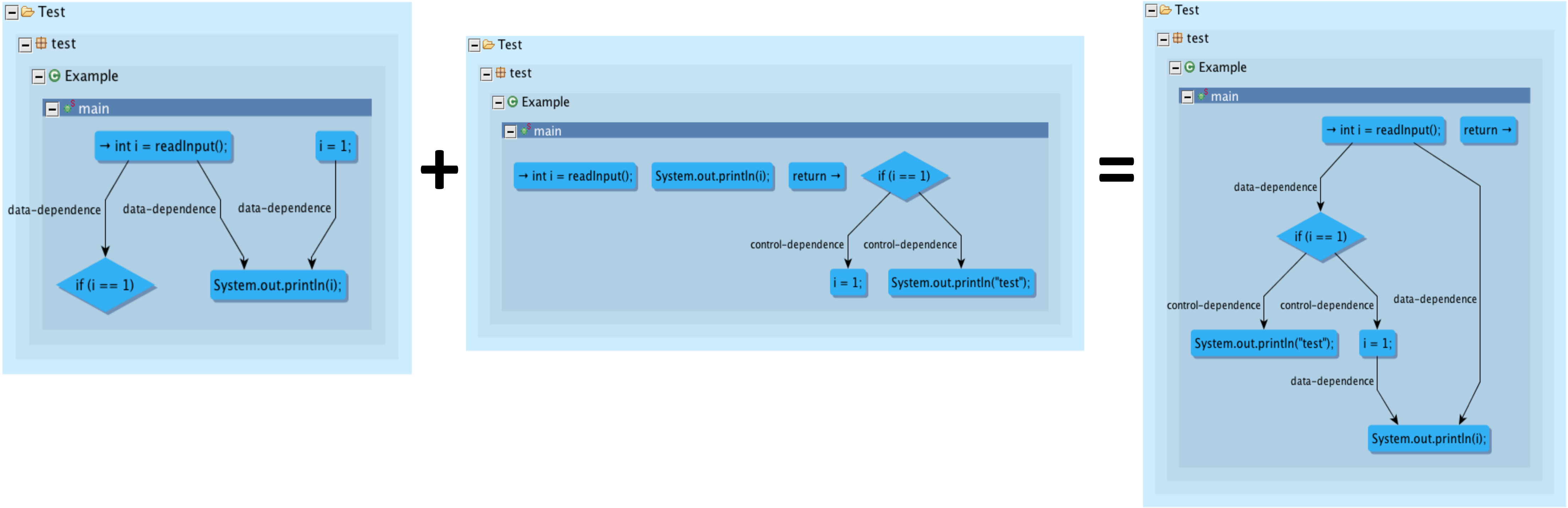
Control Dependence Slicing

- 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?

Program Dependence Graph (PDG)

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

DDG CDG PDG



Program Slicing (Impact Analysis)

- Reverse Program Slice

  Answers: What statements does this statement's execution depend on?

- Forward Program Slice

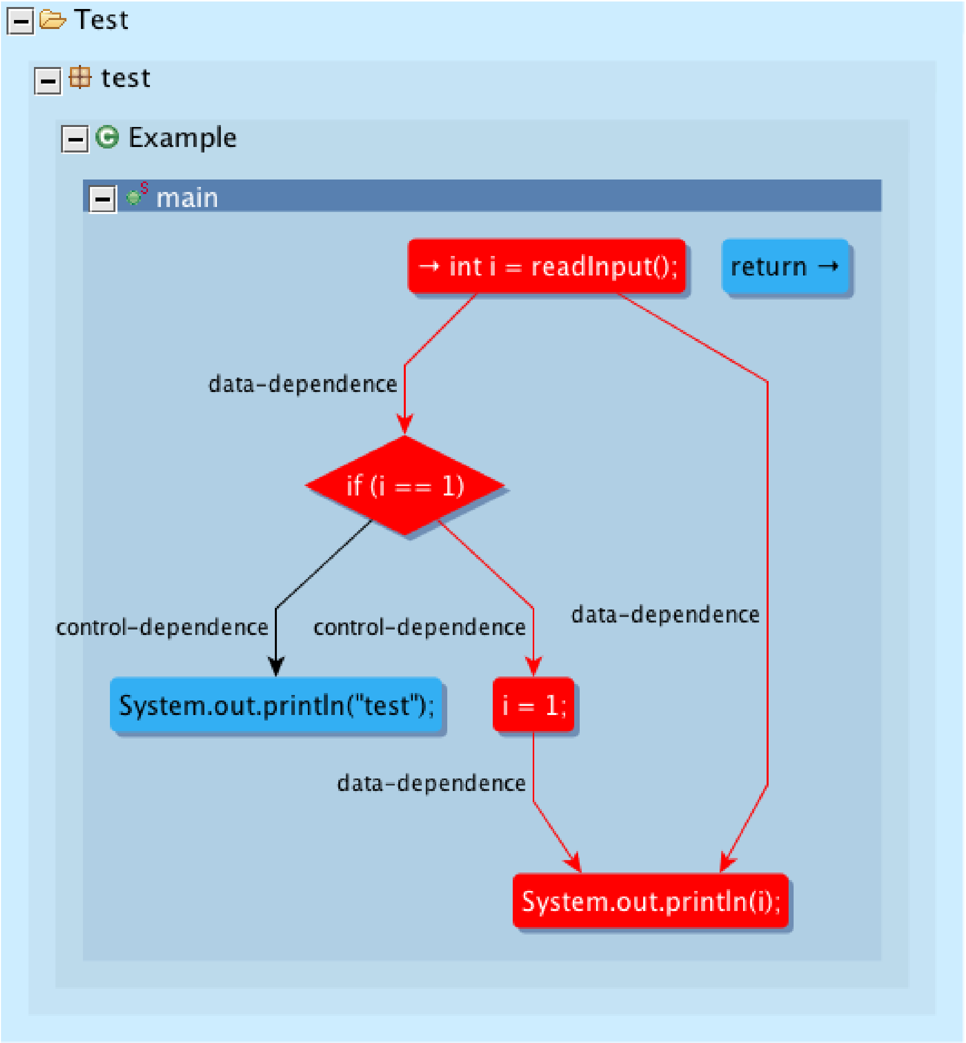
  Answers: What statements could execute as a result of this statement?

Example:

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

Relevant lines: 1,2,4

5. print(i); ← detected failure



Taint Analysis

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

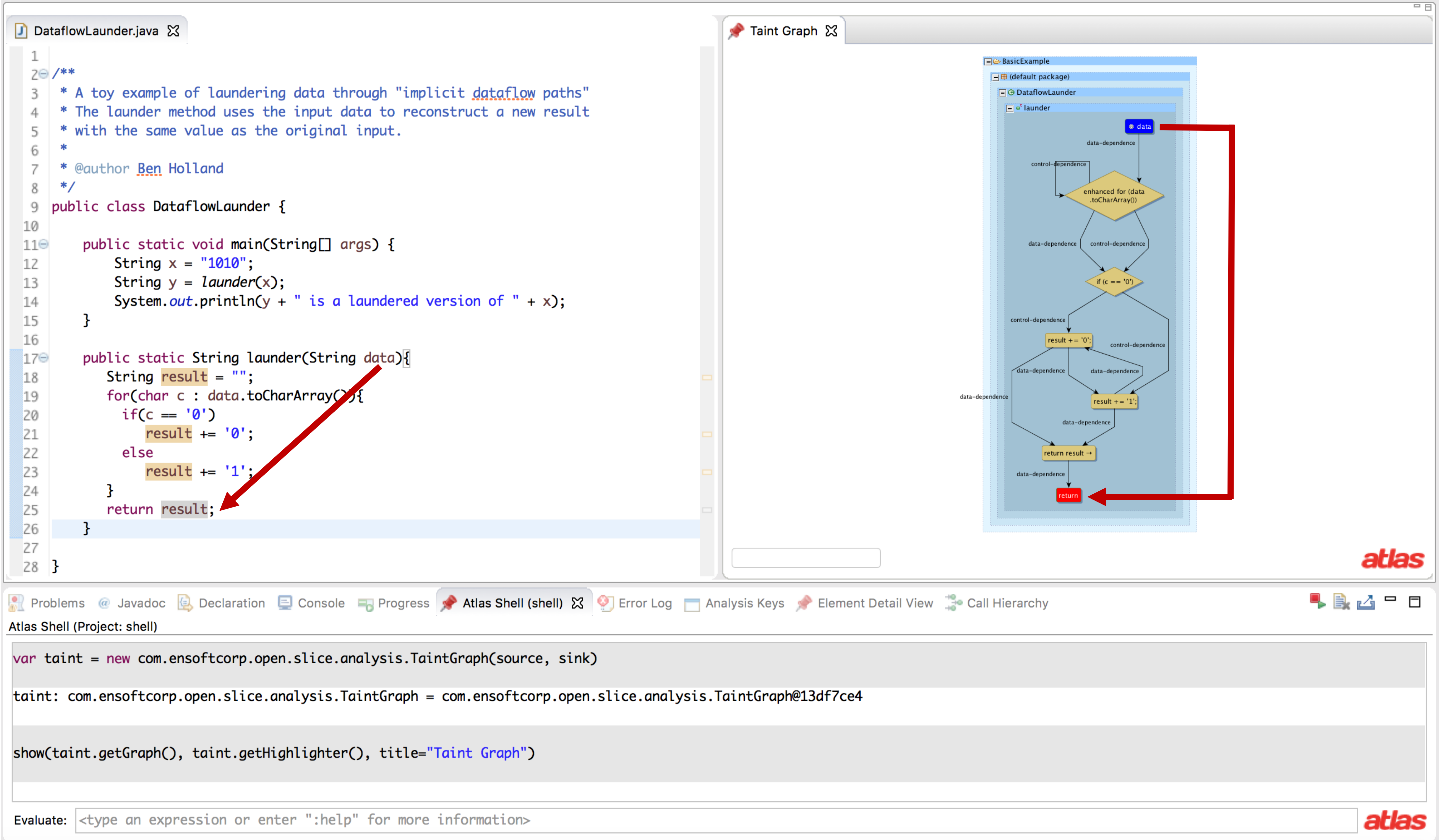
- Neither DFG/DDG nor CFG/CDG alone are enough to answer whether *x* flows to *y*
- Taint = (forward slice of *source)* intersection (reverse slice of *sink)*

```java
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
/**
* 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;
}
}
Taint Graph
BasicExample
(default package)
DataflowLaunder
launder
data
data-dependence
control-dependence
enhanced for (data .toCharArray())
data-dependence
control-dependence
if (c == '0')
control-dependence
result += '0';
control-dependence
data-dependence
data-dependence
data-dependence
result += '1';
data-dependence
return result
data-dependence
return
atlas
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>
atlas

Atlas Extensible Common Software Graph (XCSG) Schema

- XCSG's use of XCSG.DataFlow_Edge and XCSG.ControlFlow_Edge are compatible with the definitions of control and data flow as put forward by FOW87 paper
- https://ensoftatlas.com/wiki/Extensible_Common_Software_Graph

Continuations

- System Dependence Graphs
  - Horwitz, Susan, Thomas Reps, and David Binkley. "Interprocedural slicing using dependence graphs." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990): 26-60.
  - Reps, Thomas, Susan Horwitz, and Mooly Sagiv. "Precise interprocedural dataflow analysis via graph reachability." *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995.
- Survey of Slicing Techniques
  - Tip, Frank. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994.

References

- Parts of this slide deck were influenced by examples in
  - https://www.cs.colorado.edu/~kena/classes/5828/s00/lectures/lecture15.pdf
  - https://www.cc.gatech.edu/~harrold/6340/cs6340_fall2009/Slides/BasicAnalysis4.pdf