```java
public class Puzzle9 {

    public static void main(String[] args) {
        recurse();
    }

    private static void recurse() {
        try {
            recurse();
        } finally {
            recurse();
        }
    }

}
```

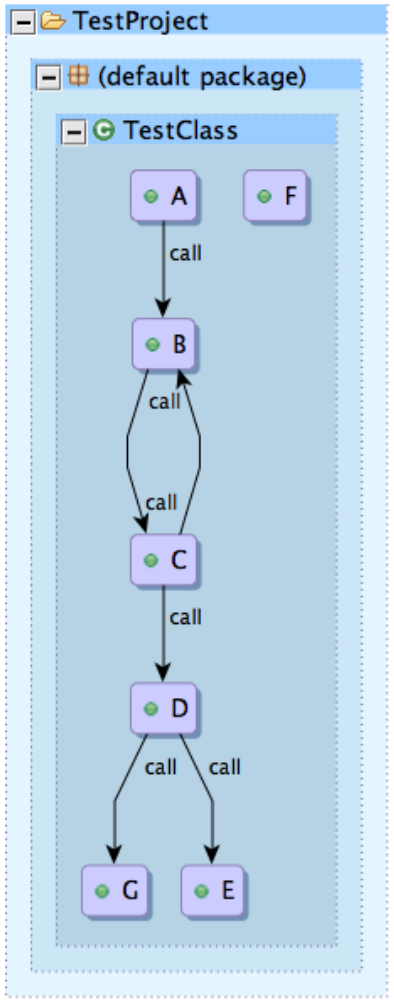
How do we build a static analysis tool?

```
1  public class TestClass {
2
3      public void A() {
4          B();
5      }
6
7      public void B() {
8          C();
9      }
10
11     public void C() {
12         B();
13         D();
14     }
15
16     public void D() {
17         G();
18         E();
19     }
20
21     public void E() {
22
23     }
24
25     public void F() {
26
27     }
28
29     public void G(){
30
31     }
32
33 }
```

Program Declarations, Control Flow, and Data Flow

Queryable Graph Database

2-way Source Correspondence



Lexical Analysis

- A program is data
- The first step is to recognize the key “tokens” and discard irrelevant information such as whitespace

```
if (ret) // probably true
  mat[x][y] = END_VAL;
```

This code produces the following sequence of tokens:

```
IF LPAREN ID(ret) RPAREN ID(mat) LBRACKET ID(x) RBRACKET LBRACKET
ID(y) RBRACKET EQUAL ID(END_VAL) SEMI
```

Lexical Analysis (Example Lexer Rules)

```
if                          { return IF; }
(                           { return LPAREN; }
)                           { return RPAREN; }
[                           { return LBRACKET; }
]                           { return LBRACKET; }
=                           { return EQUAL; }
;                           { return SEMI; }
/[ \t\n]+/                  { /* ignore whitespace */ }
/\/\/.*/                    { /* ignore comments */ }
/[a-zA-Z][a-zA-Z0-9]*"/     { return ID; }
```

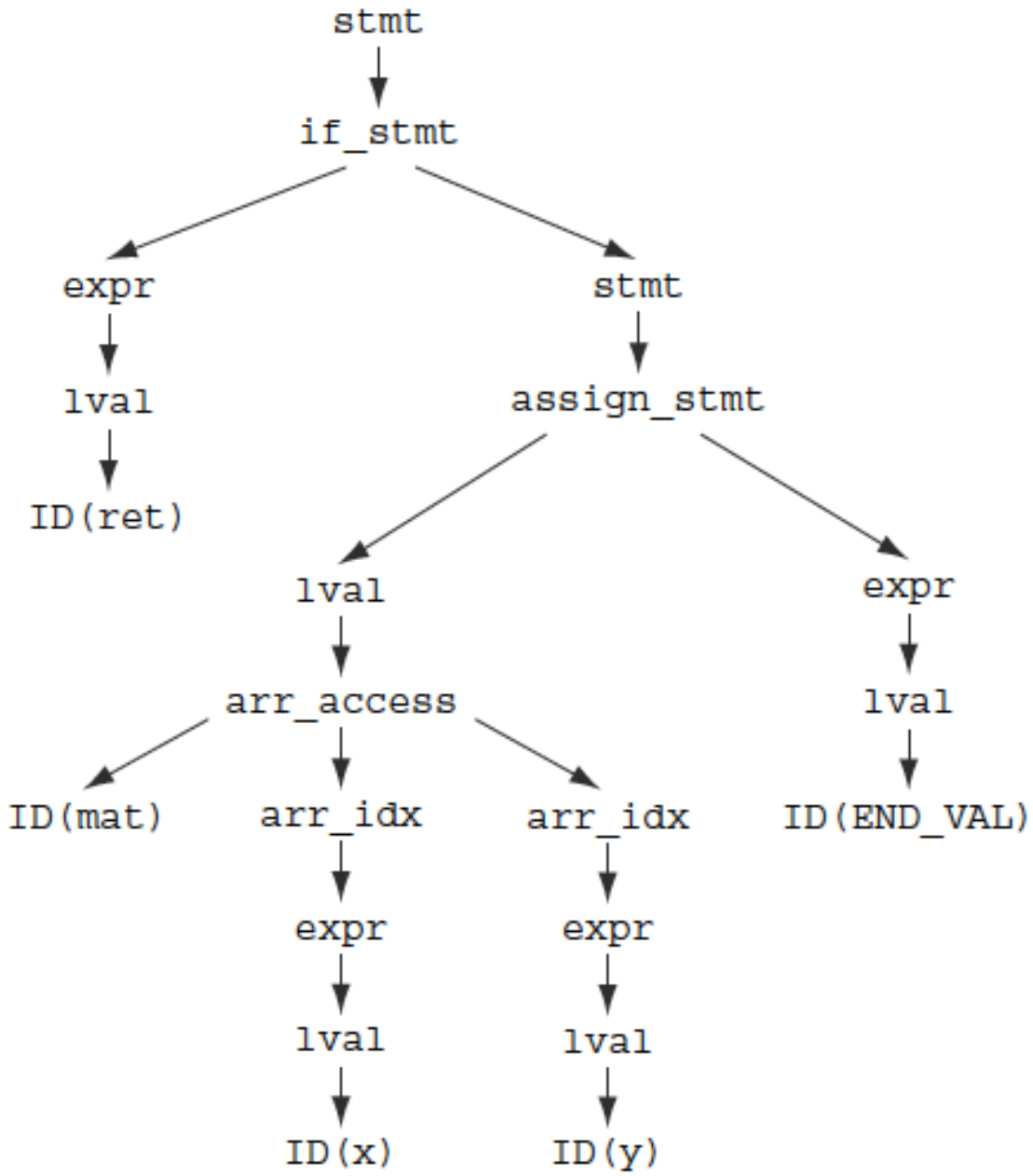
Parsing (Parser Grammar)

- A grammar consisting of a set of productions is used describe the symbols (elements) in the language

```
stmt := if_stmt | assign_stmt
if_stmt := IF LPAREN expr RPAREN stmt
expr := lval
assign_stmt := lval EQUAL expr SEMI
lval = ID | arr_access
arr_access := ID arr_index+
arr_idx := LBRACKET expr RBRACKET
```

Parsing (Parse Tree)

- The parser matches the token stream against the production rules.
- If each symbol is connected to the symbol from which it was derived, a parse tree is formed.

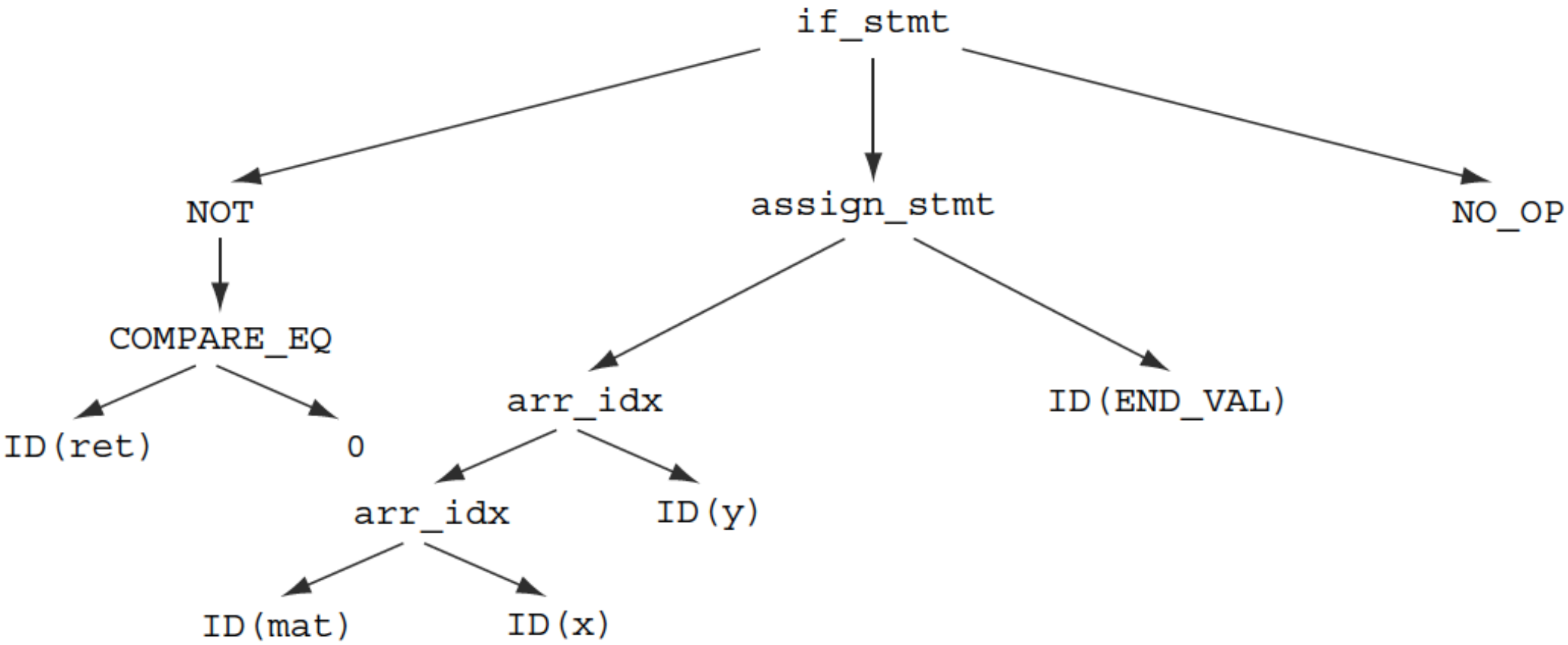


Abstract Syntax Tree (AST)

It is feasible to do significant analysis on a parse tree, and certain types of stylistic checks are best performed on a parse tree because it contains the most direct representation of the code just as the programmer wrote it. However, performing complex analysis on a parse tree can be inconvenient for a number of reasons. The nodes in the tree are derived directly from the grammar's production rules, and those rules can introduce nonterminal symbols that exist purely for the purpose of making parsing easy and non-ambiguous, rather than for the purpose of producing an easily understood tree; it is generally better to abstract away both the details of the grammar and the syntactic sugar present in the program text. A data structure that does these things is called an abstract syntax tree (AST).

- Secure Programming with Static Analysis By Brian Chess, Jacob West

Abstract Syntax Tree (AST)



ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

http://www.antlr.org

Calculator Expression Example:

- https://stackoverflow.com/a/1932664/475329

Brainf*ck Language

- Designed by Urban Müller in 1992 with the goal of implementing the smallest possible compiler.
- Compiler can be implemented in less than 100 bytes
- Implements a Turing machine
- https://en.wikipedia.org/wiki/Brainfuck

Character	Meaning
>	increment the data pointer (to point to the next cell to the right).
<	decrement the data pointer (to point to the next cell to the left).
+	increment (increase by one) the byte at the data pointer.
-	decrement (decrease by one) the byte at the data pointer.
.	output the byte at the data pointer.
,	accept one byte of input, storing its value in the byte at the data pointer.
[	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it *forward* to the command after the *matching*] command.
]	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it *back* to the command after the *matching* [command.

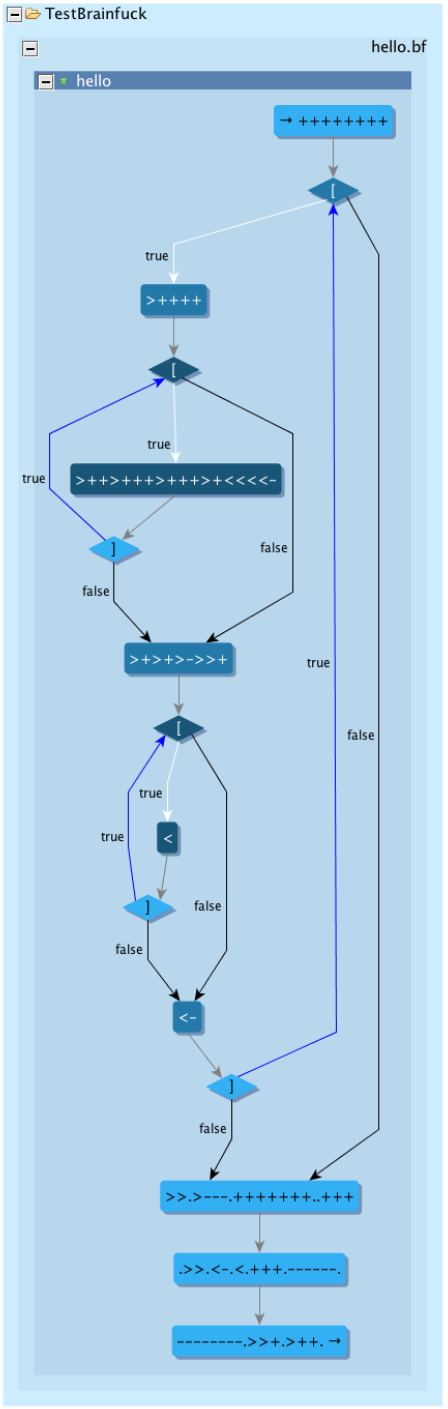
Brainf*ck (Hello World)

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---
.+++++++..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

Extra Credit Assignment

- Create a Brainf*ck static analysis tool
  1. Brainfuck Lexer
  2. Brainfuck Parser
  3. Brainfuck AST
  4. Brainfuck Control Flow Graph

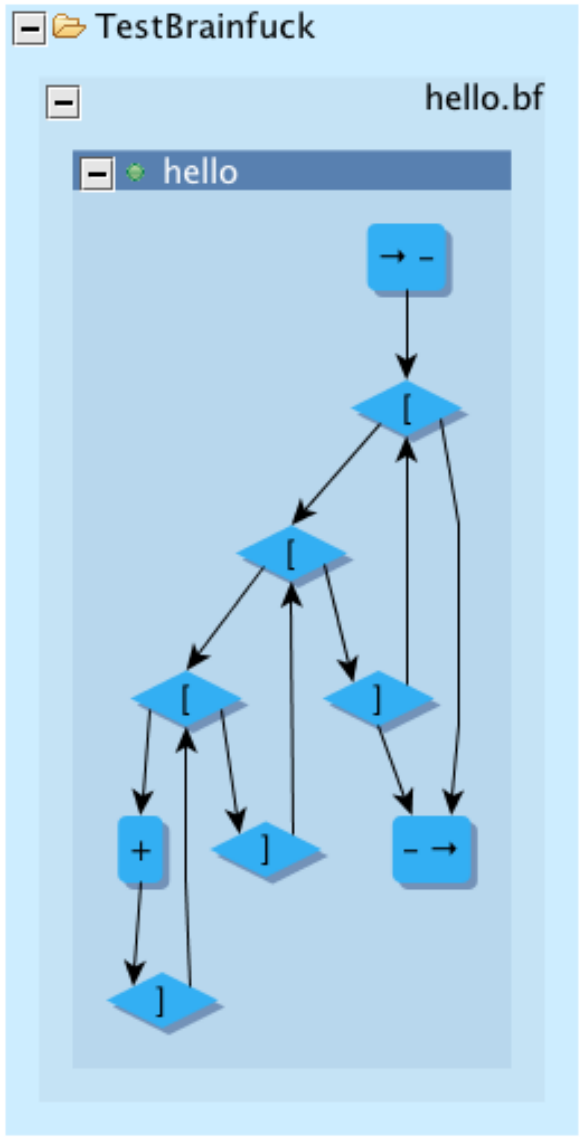
- Testing
  1. Interpret Brainf*ck program from AST
  2. Interpret Brainf*ck program from CFG

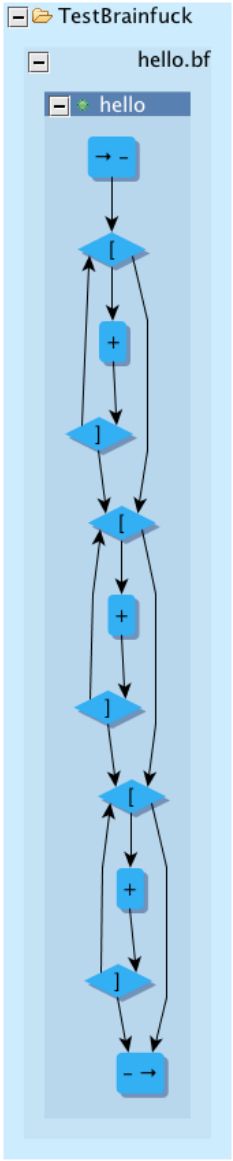


Brainf*ck Control Flow Paths

- Nested
  - Example: -[[[+]]]-
- Non-nested
  - Example: -[+][+][+]-

- Observation: Even though we may not know if a path is feasible, we always know what the next instruction *could be* for this language (at most 2 CFG successors)





A New Brainf*ck Language Feature

&	(Computed GOTO) Jumps to the n^{th} instruction where n is defined by the current cell value

- With this language addition how would you draw the CFG?
  - Conservatively from the & to all nodes?
  - This mixing of control and data makes program analysis hard

Liveness Analysis

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

Use-Definition Chains (UD Chain) is a data structure that consists of a use, U, of a variable, and all the definitions, D, of that variable that can reach that use without any other intervening definitions.

A definition is alive if it has a use at a later statement in the sequential execution of all statements.

A definition kills all previous definitions for the same variables.

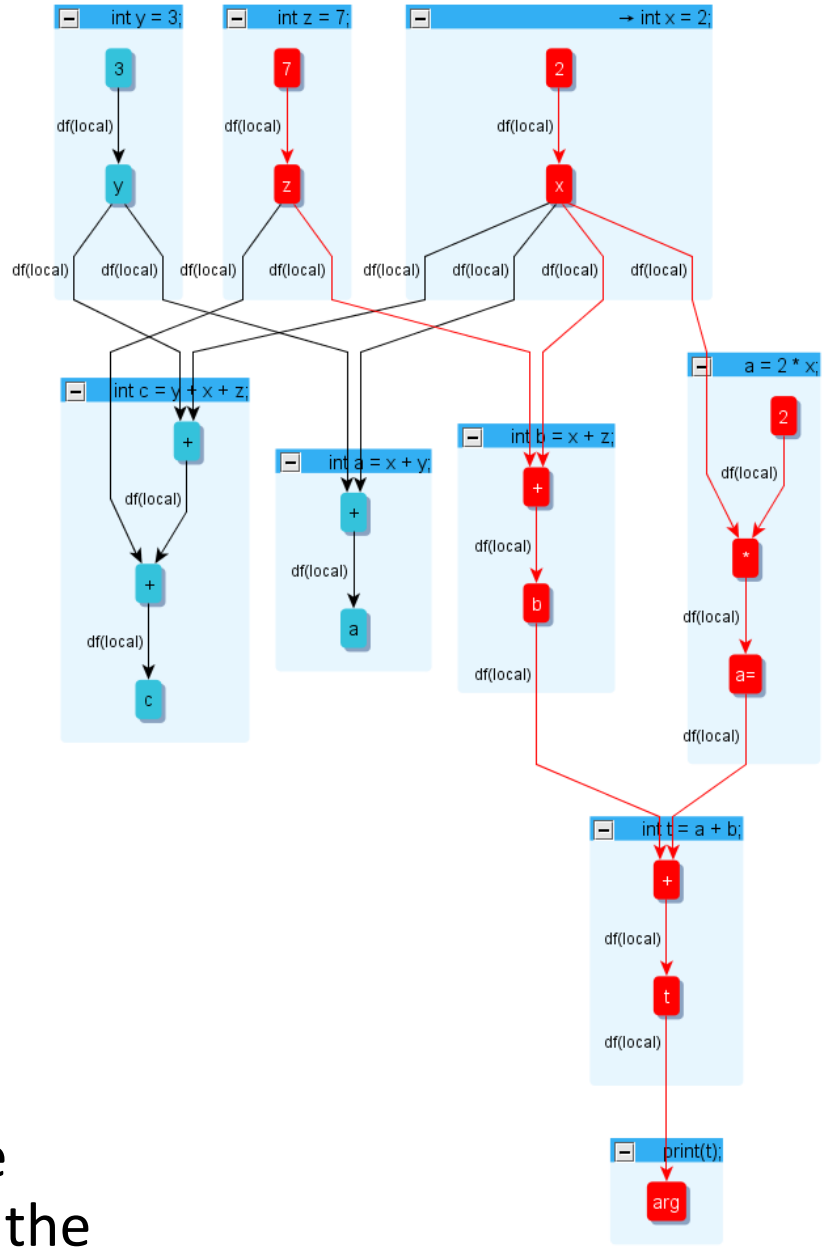
What lines must we consider if the value of *t* printed is incorrect?

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

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

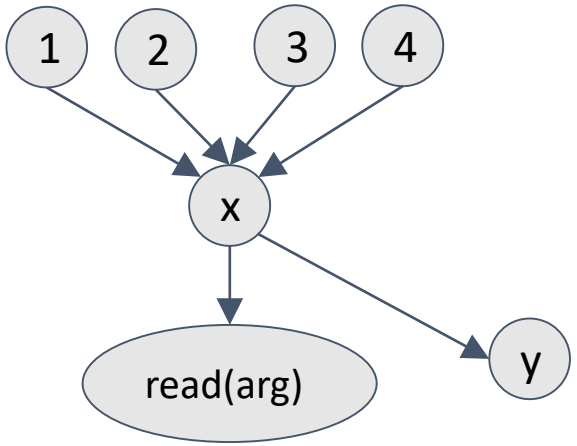


What lines must we consider if the value of *t* printed is incorrect?

- A *Data Flow Graph* (DFG) creates a graph of atomic primitive, variables, and operator relationships where each assignment represents an edge from the RHS to the LHS of the assignment.

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;



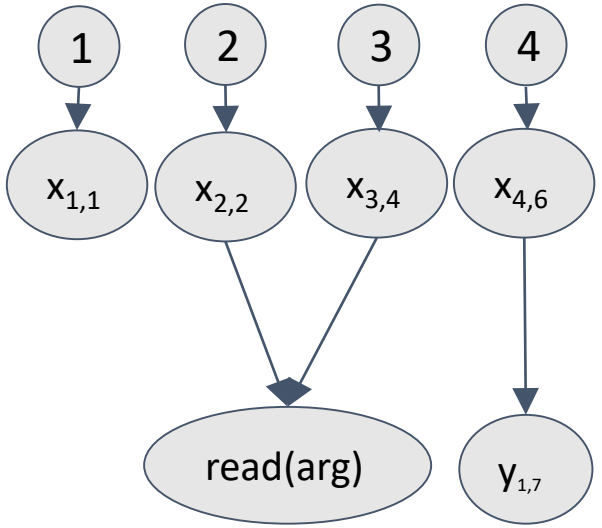
Resulting graph when statement ordering is not considered.

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#>



Implicit Data Flow

- Control impacts data
- Data impacts control

```
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;
        }

}
```

Points-to Analysis

- Could we answer whether or not two variables could point-to the same value in memory?