```java
public class Puzzle10 {

    public static void main(String[] args) {
        ((Puzzle10) null).print();
    }

    private static void print() {
        System.out.println("Hello World!");
    }

}
```

Brainf*ck Lexical Analysis

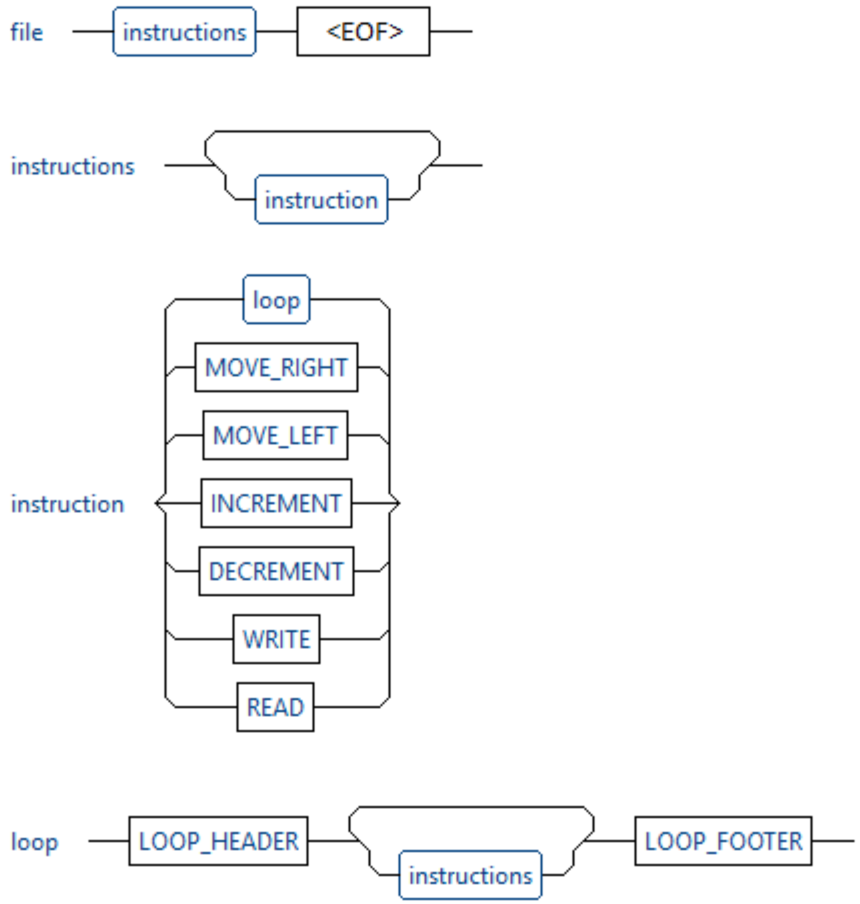
```
MOVE_RIGHT: '>';
MOVE_LEFT: '<';
INCREMENT: '+';
DECREMENT: '-';
WRITE: '.';
READ: ',';
LOOP_HEADER: '[';
LOOP_FOOTER: ']';
```

Program: ***++[>+[+]].***

Program Tokens: **INCREMENT INCREMENT LOOP_HEADER MOVE_RIGHT INCREMENT LOOP_HEADER INCREMENT LOOP_FOOTER LOOP_FOOTER WRITE <EOF>**

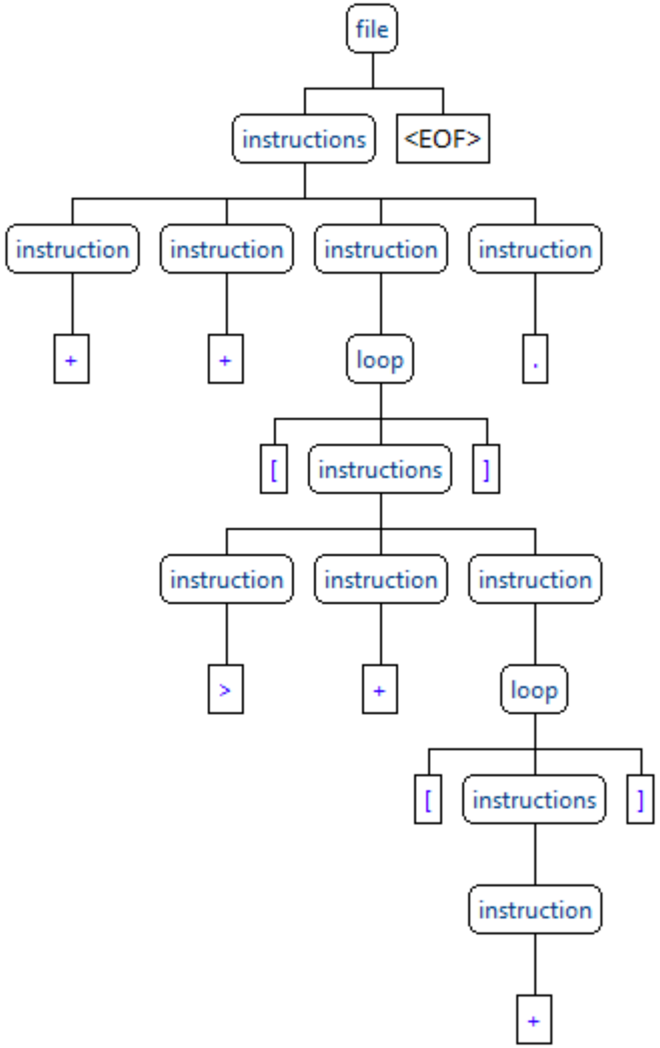
Brainf*ck Parsing Rules

```
file: instructions EOF;

instructions: instruction+;

instruction: loop
           | MOVE_RIGHT
           | MOVE_LEFT
           | INCREMENT
           | DECREMENT
           | WRITE
           | READ
           ;

loop: LOOP_HEADER instructions+ LOOP_FOOTER;
```

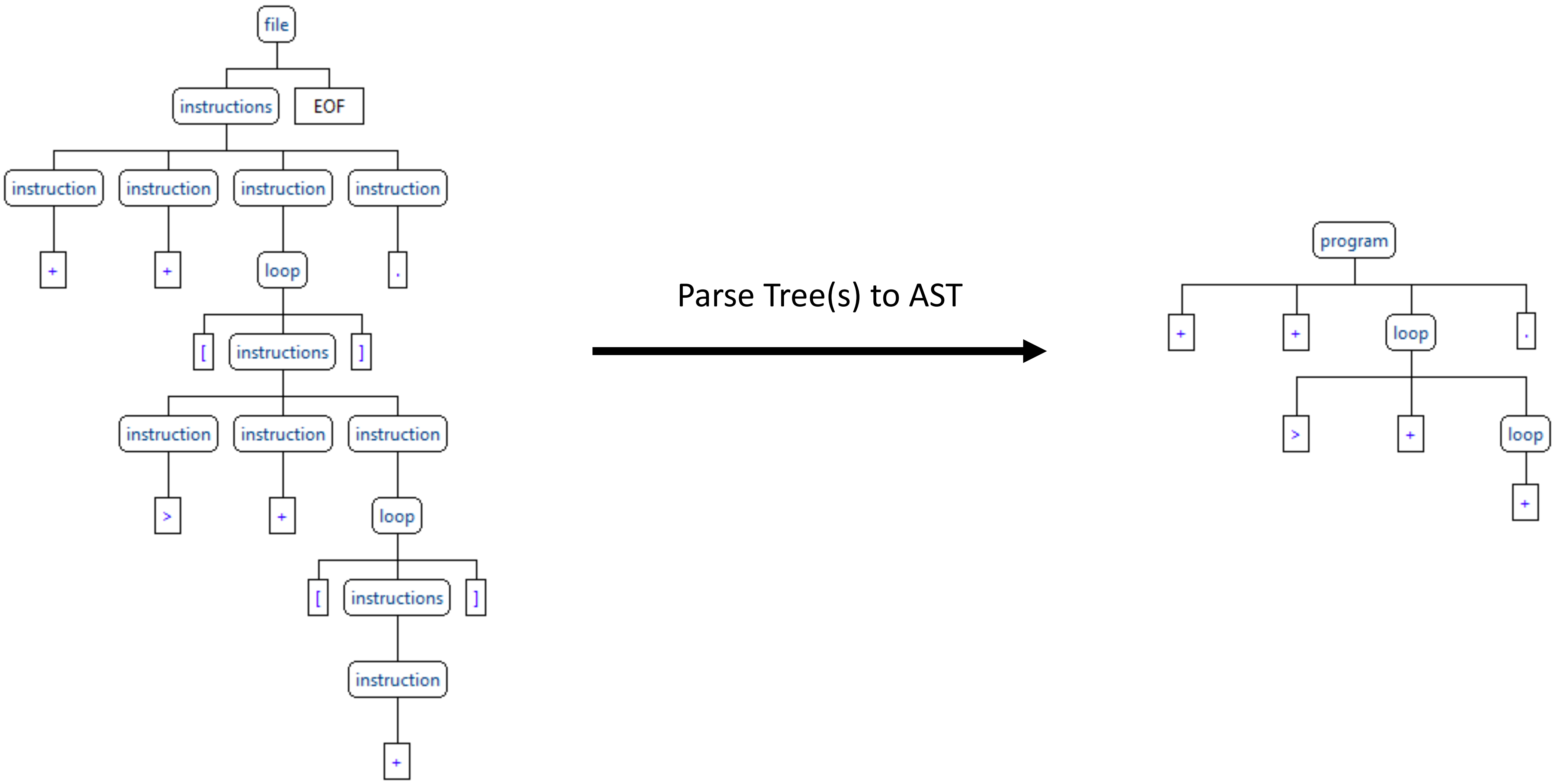


Brainf*ck Parse Tree

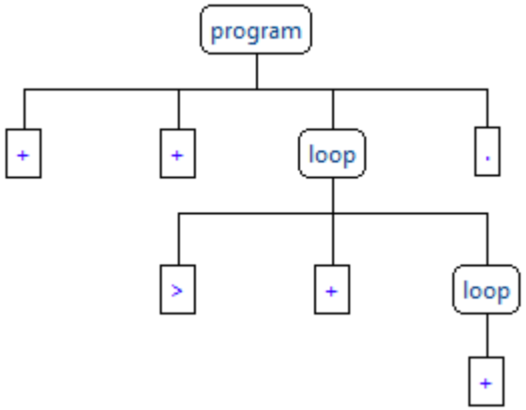
Program: ***++[>+[+]].***



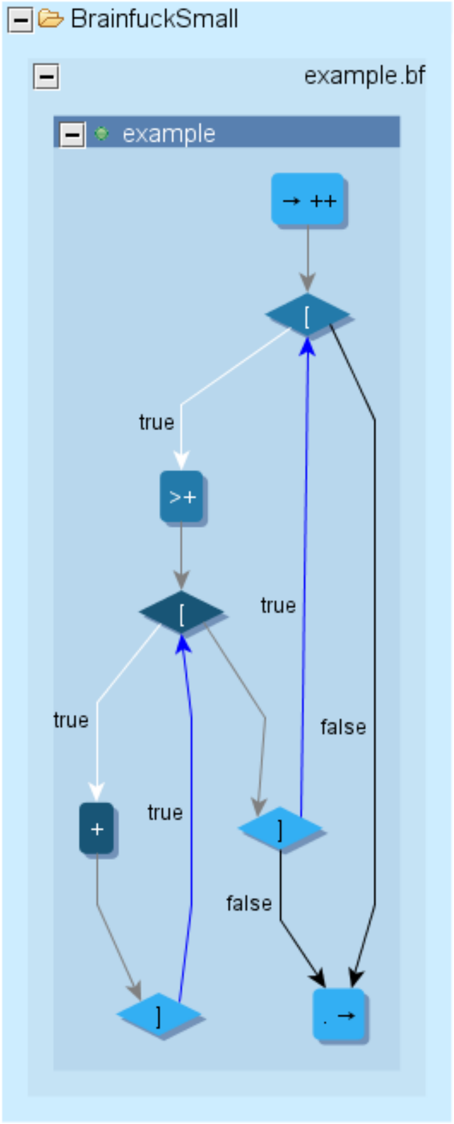
Brainf*ck Abstract Syntax Tree (AST)



Brainf*ck AST to Program Graph



Parse Tree(s) to AST

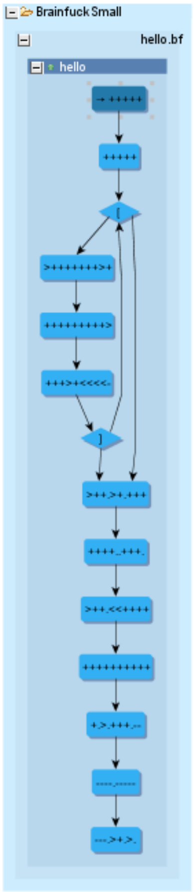


hello.bf

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

- Brainf*ck Hello World Program
- Graph contains information necessary to execute program
- This language should be simple to analyze right???
    - No variables, just tape cells
    - How many behaviors could there be?

Graph 2



Error Log | Search | Problems | Atlas Shell | Console | Progress | Analysis Keys | Git Staging

```
Operation completed successfully...

show(cfg(universe.nodes(XCSG.Function)))

com.benjholla.atlas.brainfuck.interpreter.BrainfuckGraphInterpreter.execute(selected)
res1: String =
"Hello World!
"
```

Evaluate:

Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental
  - Goal is to be basic, not to be tiny
  - Separates looping and branching
  - New features to explore impacts of modern language features

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
,	(Store) Read byte value from input into current tape cell
.	(Recall) Write byte value to output from current tape cell
(	(Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[	(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{*[0-9]+*}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"*[0-9]+*"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
'*[0-9]+*'	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental
  - Goal is to be basic, not to be tiny
  - Separates looping and branching
  - New features to explore impacts of modern language features

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
,	(Store) Read byte value from input into current tape cell
.	(Recall) Write byte value to output from current tape cell
(	(Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[	(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{*[0-9]+*}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"*[0-9]+*"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
'*[0-9]+*'	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Elemental: A Brainf*ck Derivative

- github.com/benjholla/Elemental
  - Goal is to be basic, not to be tiny
  - Separates looping and branching
  - New features to explore impacts of modern language features

- '?' could pass control to any function!
- '&' could jump to any line!
- Goto labels with '?' or '&' could be simulated with branching or loops
- These blur control flow with data

Instruction	Description
+	Increment the byte at the current tape cell by 1
-	Decrement the byte at the current tape cell by 1
<	Move the tape one cell to the left
>	Move the tape one cell to the right
,	(Store) Read byte value from input into current tape cell
.	(Recall) Write byte value to output from current tape cell
(	(Branch) If the byte value at the current cell is 0 then jump to the instruction following the matching), else execute the next instruction
[	(While Loop) If the byte value at the current cell is 0 then jump to the instruction following the matching], else execute instructions until the matching] and then unconditionally return to the [
[0-9]+:	(Function) Declares a uniquely named function (named [0-9]+ within range 0-255)
{*[0-9]+*}	(Static Dispatch) Jump to a named function
?	(Dynamic Dispatch/Function Pointer) Jumps to a named function with the value of the current cell
"*[0-9]+*"	(Label) Sets a unique label (named [0-9]+ within range 0-255) within a function
'*[0-9]+*'	(GOTO) Jumps to a named label within the current function
&	(Computed GOTO) Jumps to the named label within the current function with the value of the current cell
#	A one line comment

Positive Trend – Addressing the Languages

- Data drives execution
  - Data is half of the program!
  - “The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program.”
- Crema: A LangSec-Inspired Programming Language
  - Giving a developer a Turning complete language for every task is like giving a 16 year old a formula one car (something bad is bound to happen soon)

Positive Trend – Addressing the Languages

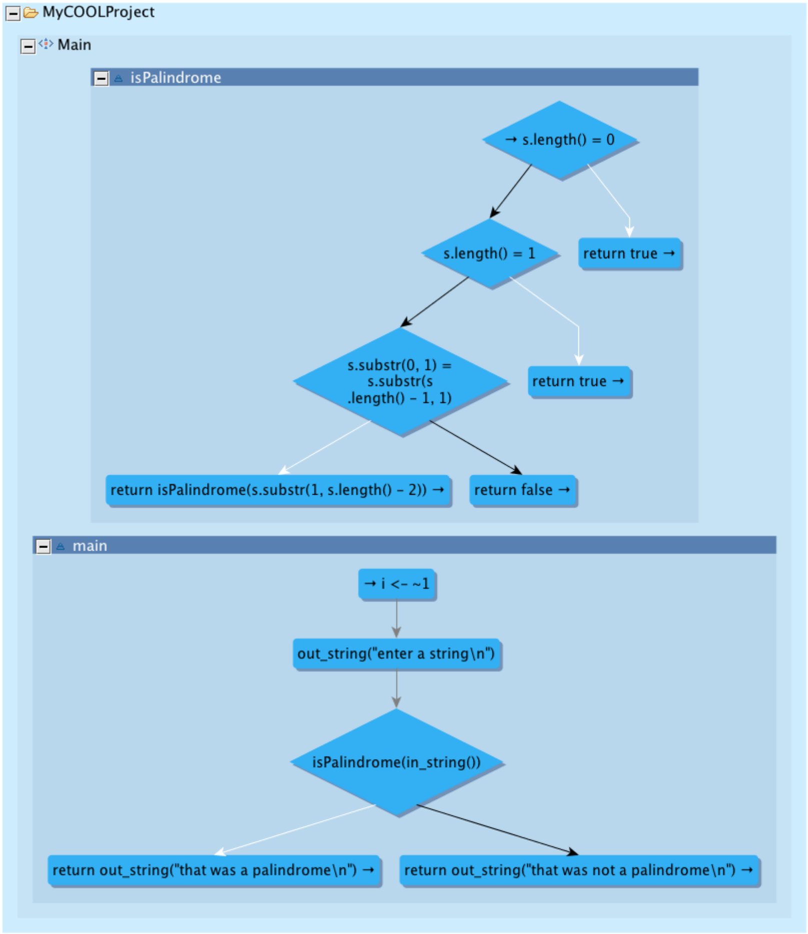
- Data drives execution
  - Data is half of the program!
  - "The illusion that your program is manipulating its data is powerful. But it is an illusion: The data is controlling your program."

- Crema: A LangSec-Inspired Programming Language (DARPA Pilot Study)
  - Giving a developer a Turing complete language for every task is like giving a 16 year old a formula one car (something bad is bound to happen soon)
  - Apply principle of least privilege to computation (least computation principle)
    - Computational power exposed to attacker ***is*** privilege. Minimize it.
    - Try copy-pasting the XML billion-laughs attack from Notepad into MS Word if you want to see why…

Scaling Up: Program Analysis for COOL

- Classroom Object Oriented Language (COOL)
  - https://en.wikipedia.org/wiki/Cool_(programming_language)
  - http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=Compilers

- COOL Program Graph Indexer
  - Type hierarchy
  - Containment relationships
  - Function / Global variable signatures
  - Function Control Flow Graph
  - Data Flow Graph (in progress)
  - Inter-procedural relationships:
    - Call Graph (implemented via compliance to XCSG!)
  - https://github.com/benjholla/AtlasCOOL (currently private)



Program Analysis for Contemporary Languages

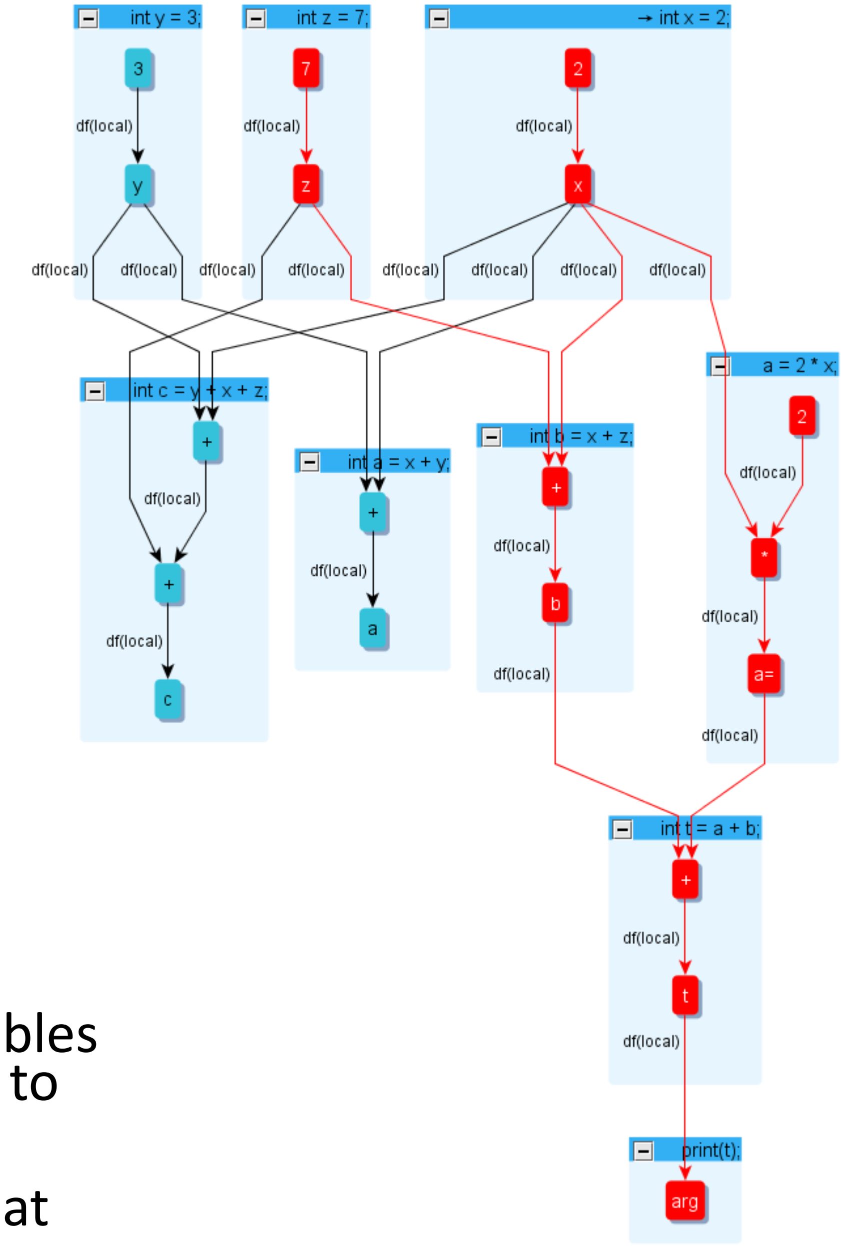
- http://www.ensoftcorp.com/atlas (Atlas)
  - C, C++, Java Source, Java Bytecode, *and now Brainfuck/COOL!*
- https://scitools.com (Understand)
  - C, C++ Source
- http://mlsec.org/joern (Joern)
  - C, C++, PHP Source
- https://www.hex-rays.com/products/ida (IDA)
- https://binary.ninja (Binary Ninja)
- https://www.radare.org (Radare)

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

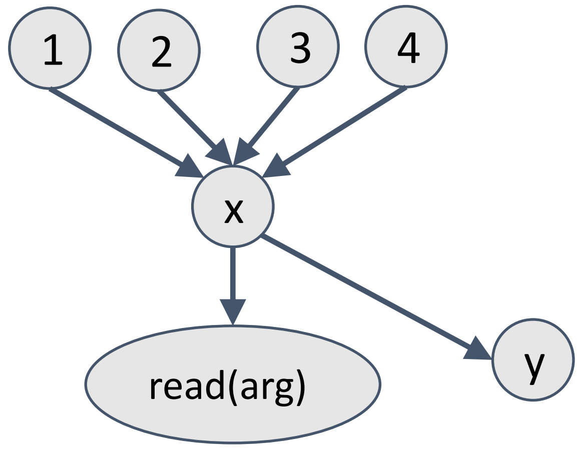


What lines must we consider if the value of *t* printed is incorrect?

- 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]

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;



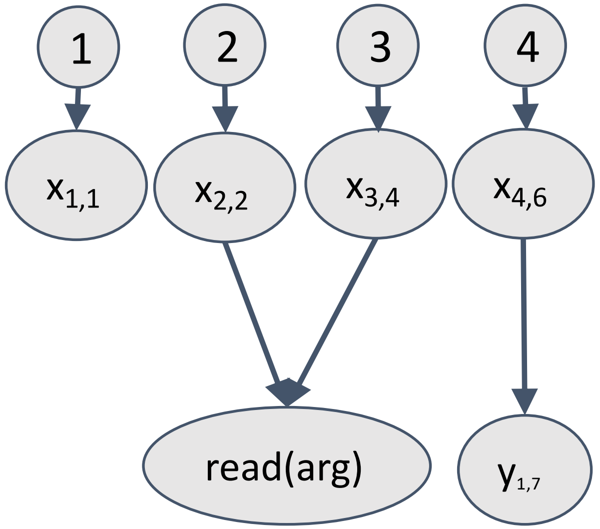
Resulting graph when statement ordering is not considered.

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#>



Points-to (Pointer) Analysis

- Could we answer whether or not two variables point-to the same value in memory?
- Why do we even care?
  - “Virtually all interesting questions one may want to ask of a program will eventually need to query the possible values of a pointer expression, or its relationship to other pointer expressions.”
  - Constant propagation
  - Precise call graph construction
  - Dead code elimination
  - Immutability analysis
  - Etc.

Points-to Analysis

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

Points-to Analysis

- Easy (useless) Solution:
  - A variable *must* at least point-to nothing (null)
  - Every variable *may* at most point-to anything

- Perfect (impossible) Solution:
  - A perfect Points-to is undecidable [Landi1992] [Ramalingan1994]

Andersen-style Points-to Analysis

- Flow-insensitive
  - The order of statements is not considered (does not leverage control flow graph)
- Analysis
  1. Identify each memory value to track
  2. Consider pointer assignments as subset constraints

Constraint type	Assignment	Constraint	Meaning
Base	a = &b	$a \supseteq \{b\}$	$loc(b) \in pts(a)$
Simple	a = b	$a \supseteq b$	$pts(a) \supseteq pts(b)$
Complex	a = *b	$a \supseteq *b$	$\forall v \in pts(b).\ pts(a) \supseteq pts(v)$
Complex	*a = b	$*a \supseteq b$	$\forall v \in pts(a).\ pts(v) \supseteq pts(b)$

Andersen-style Points-to Analysis

- Fixed-point Algorithm Sketch (for Java)
  1. Identify each value to track (i.e. “new” → XCSG.Instantiation) and assign it a unique “address”
  2. Create a worklist of nodes with addresses to propagate and initialize with each addressed node
  3. If the worklist is not empty, remove a node from the worklist
     - Propagate the addresses of the node to each data flow successor node
     - If the data flow successor node received new addresses then add the successor node to the worklist
     - Repeat step 3
  4. When the algorithm reaches a fixed-point (no addresses left to propagate) then the points-to sets have been computed

Andersen-style Points-to Analysis

- Worst Case Performance?
- Worst Case: Every variable is assigned to every other variable.
  - This is the handshake problem → n* (n-1) → $O(n^2)$ for each iteration
  - Statements are being processed out of order, so processing a new statement could cause you to redo all previous work → $n*(n^2)$ → $O(n^3)$

Problem 1

(10 points)

A. (*2 points*) Given a program with n branch conditions and no loops, how many live definitions can there be for a given use? Explain.

B. (*2 points*) Given a program with n branch conditions and no loops, how many uses can there be for a given definition? Explain.

C. (*4 points*) List the different programming language features that allow a function `foo` to pass data to or from another function `bar`. For each language feature, provide an example and indicate the underlying memory architecture (stack or heap) that enables the transfer of information.

D. (*2 points*) Can data flow be obscured through control flow? Explain. Likewise can control flow be obscured through data flow? Explain.