# Developing the Clang Static Analyzer

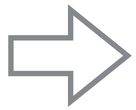
Artem Dergachev, Apple

# Clang Static Analyzer

- Finds bugs at compile time – by inspecting your source code
- Bugs it finds are more sophisticated than warnings or Clang-Tidy

# Clang Static Analyzer

```
1  int foo(int x) {
2    int y = x;
3    if (y == 0)
4      return 24 / x;
5    return 0;
6  }
```



```
1  int foo(int x) {
2    int y = x;
3    if (y == 0)
       1 Assuming 'y' is equal to 0 →
       2 ← Taking true branch →
4      return 24 / x;
                3 ← Division by zero
5    return 0;
6  }
```

# Clang Static Analyzer

- Natural!
  - Mimics normal program execution
  - Easy to understand why it “thinks” there is a bug
  - Takes all source code information into account
  - Explains bugs in terms of the source code

# Clang Static Analyzer

- Natural!
- Researchy!

- Deals with a lot of open problems
- People publish articles,
  defend BS/MS/Ph.D. theses on it
- Fully Open Source – lives in Clang repo

# Clang Static Analyzer

- Natural!
- Researchy!
- Practical!

- Used in industry
- Shipped with IDEs
- Finds bugs in your code before your users do!

# Clang Static Analyzer

- Natural!
- Researchy!
- Practical!
- Extensible!

- Finds over 50 kinds of bugs!
  - Memory leaks
  - Null dereferences
  - Use-after-free
  - Use-after-move
  - …
- “Building a Checker in 24 hours” – LLVM DevMtg 2012
  https://youtu.be/kdxlsP5QVPw

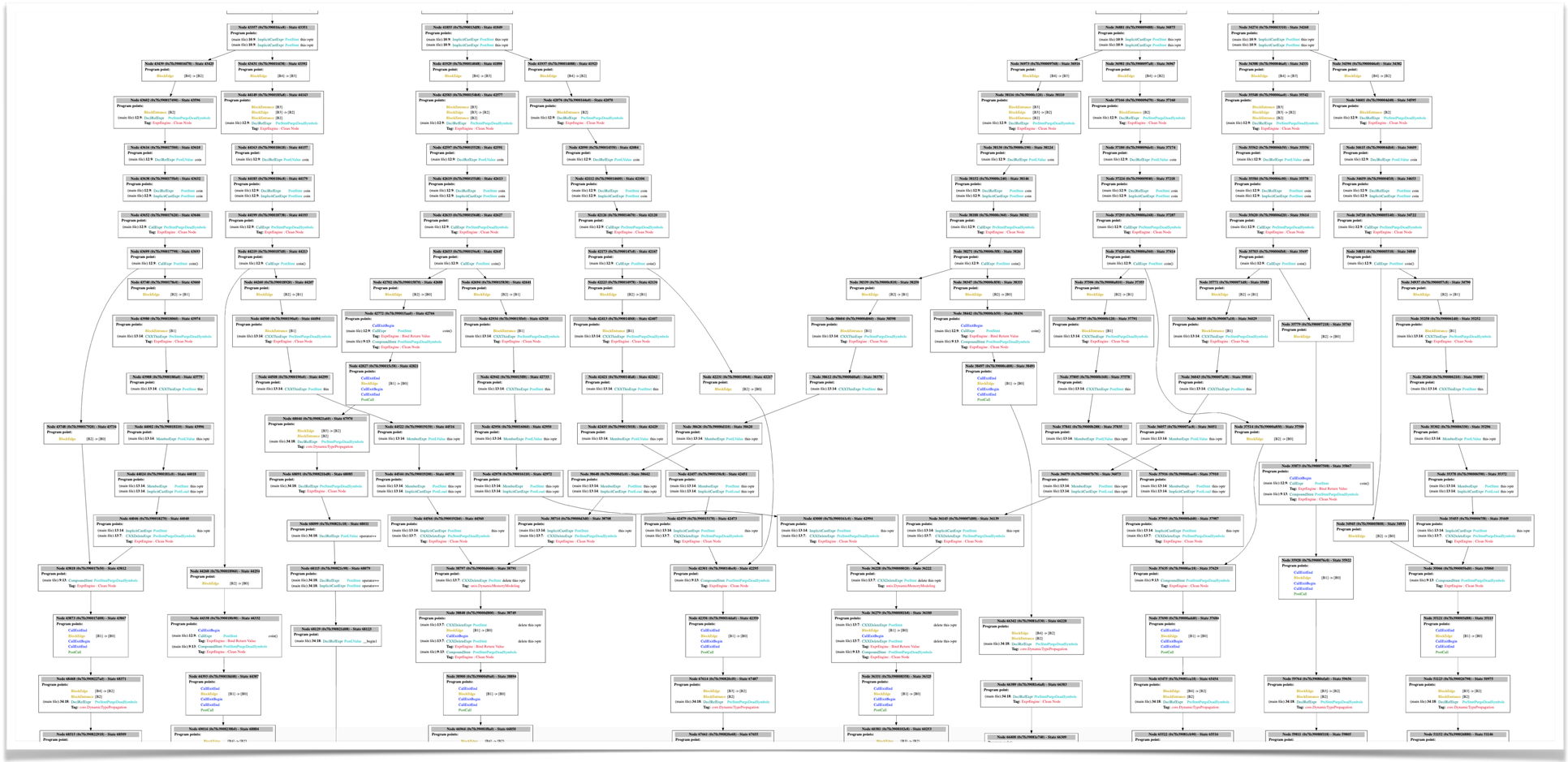
# Clang Static Analyzer

- Natural!
- Researchy!
- Practical!
- Extensible!
- 🎃 Spooky!

- 🎃 Symbolic Execution
- 🎃 Dead, Undead, Zombie and Schrödinger Symbols, The Reaper
- 🎃 Body Farms

# Clang Static Analyzer

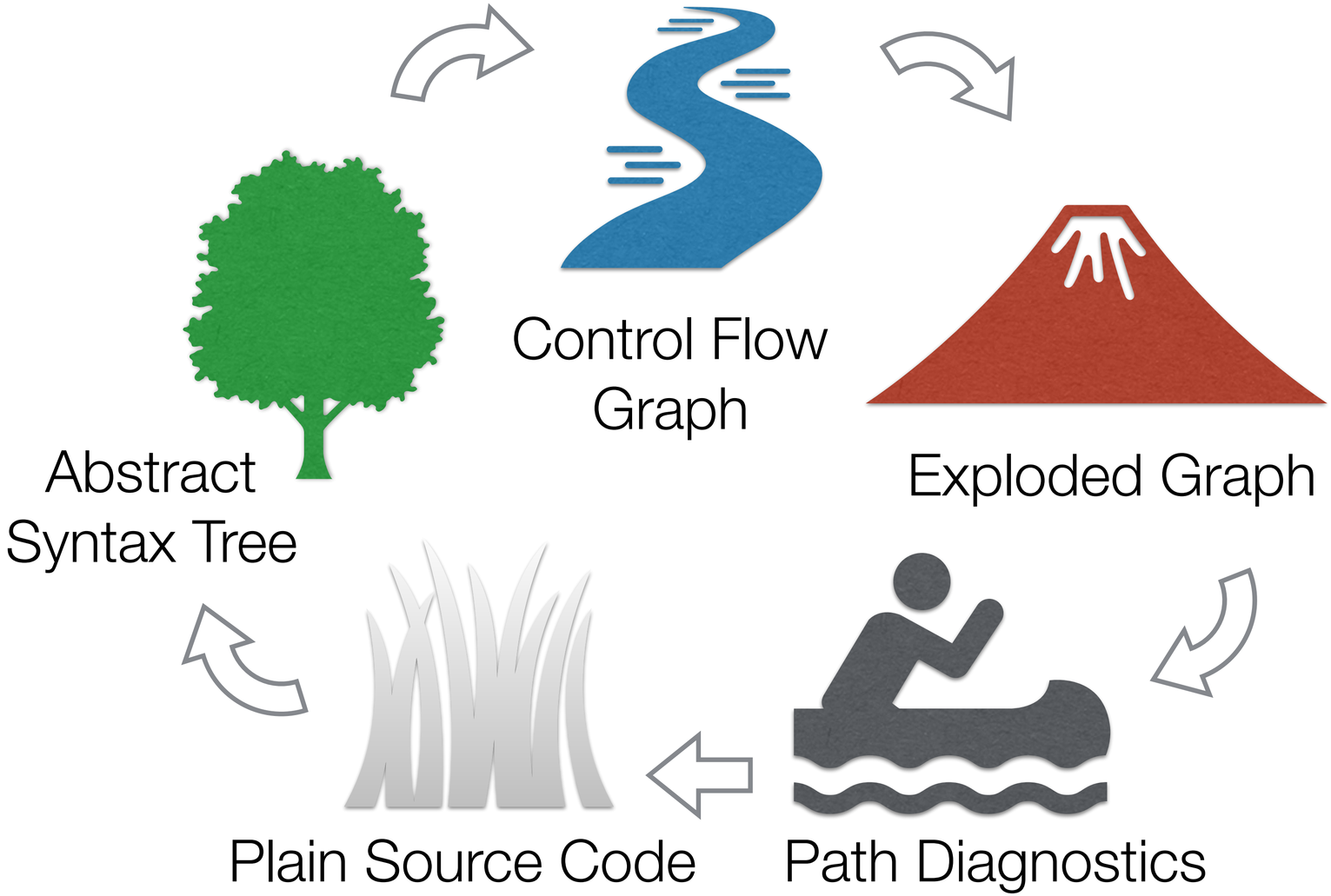
- Natural!
- Researchy!
- Practical!
- Extensible!
- 🎃 Spooky!
- Exciting!



# Plan For Today!

- Algorithms and Data Structures of the Static Analyzer
- How to Fix a Static Analyzer Bug in 24 minutes

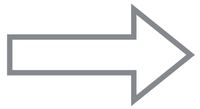
# Algorithms and Data Structures of the Static Analyzer

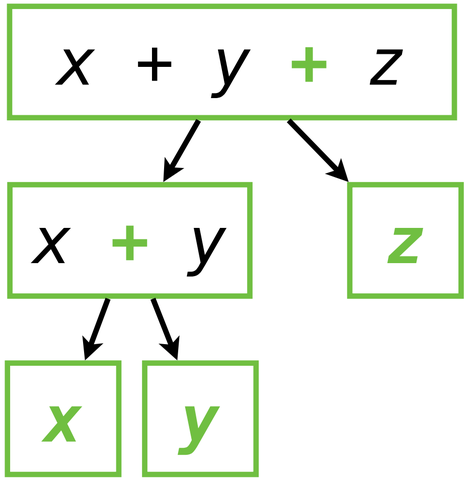


# AST: How Compiler Sees Your Code

- *Nodes*: statements, declarations, types – annotated and cross-referenced
- *Edges*: “is-part-of” relation

$x + y + z$

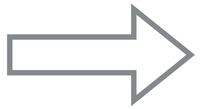


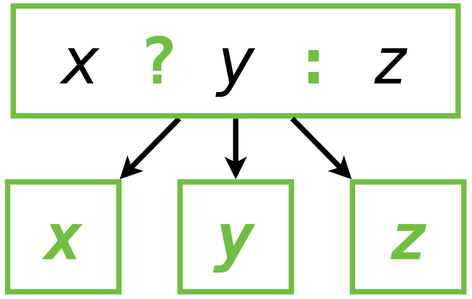



# AST: How Compiler Sees Your Code

- *Nodes*: statements, declarations, types – annotated and cross-referenced
- *Edges*: “is-part-of” relation

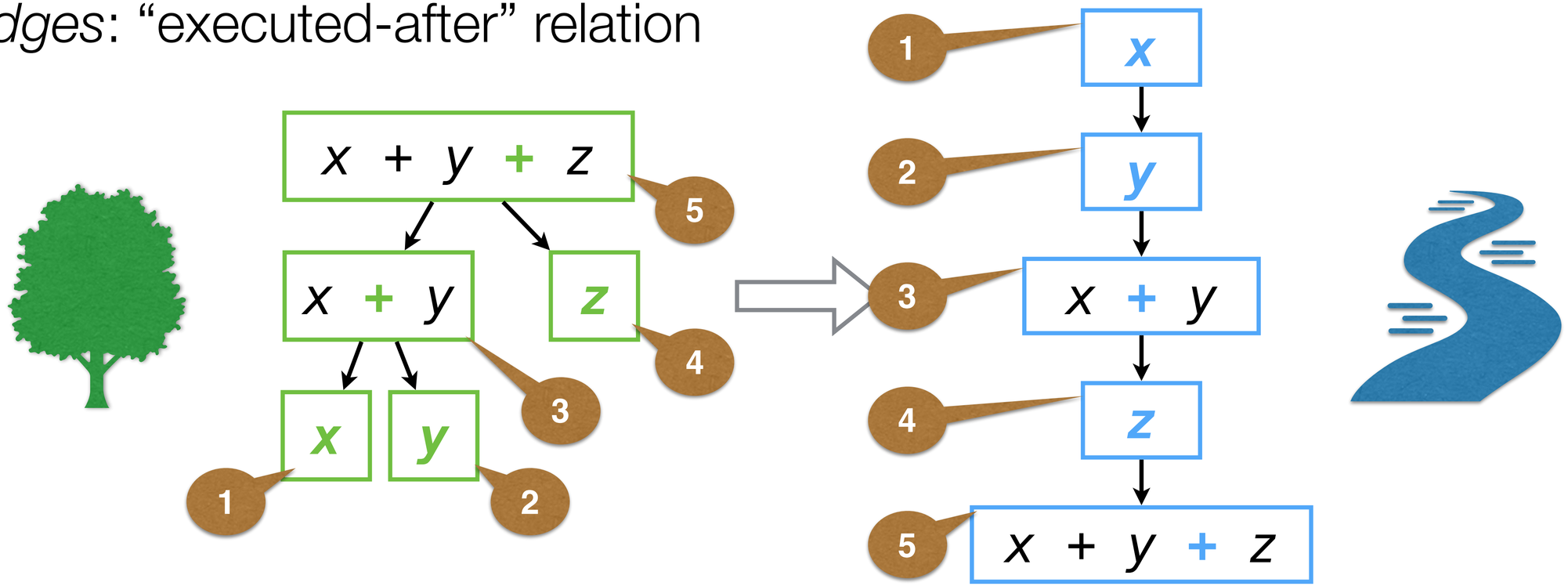
`x ? y : z`





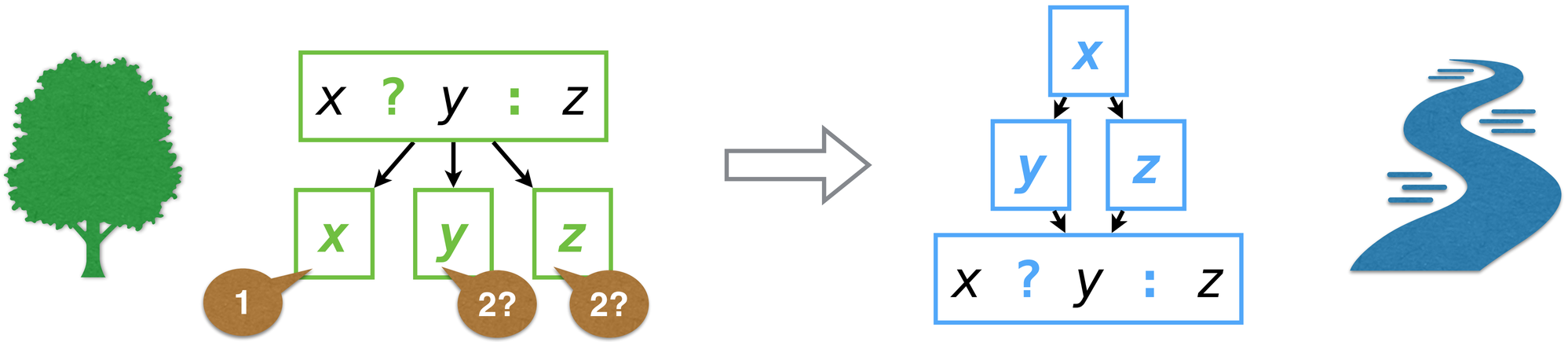

# CFG: Order in which Statements are Executed

- *Nodes*: usually AST statements
- *Edges*: “executed-after” relation

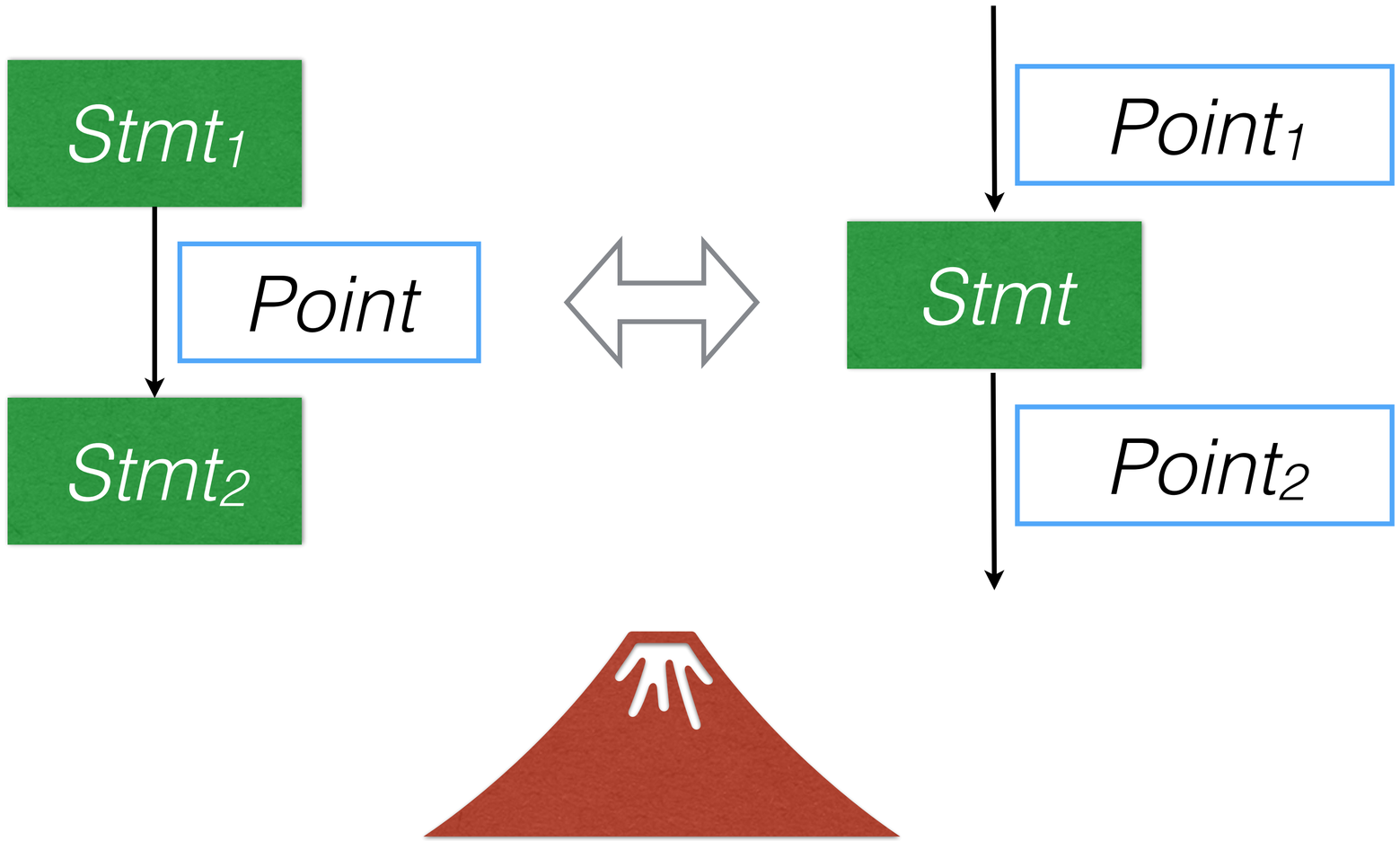


# CFG: Order in which Statements are Executed

- *Nodes*: usually AST statements
- *Edges*: “executed-after” relation



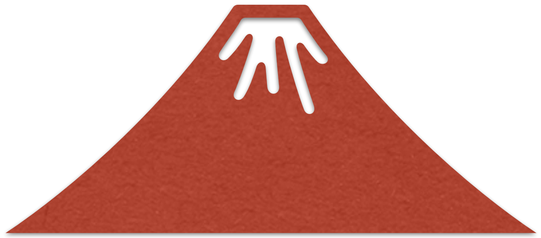
# Program Points



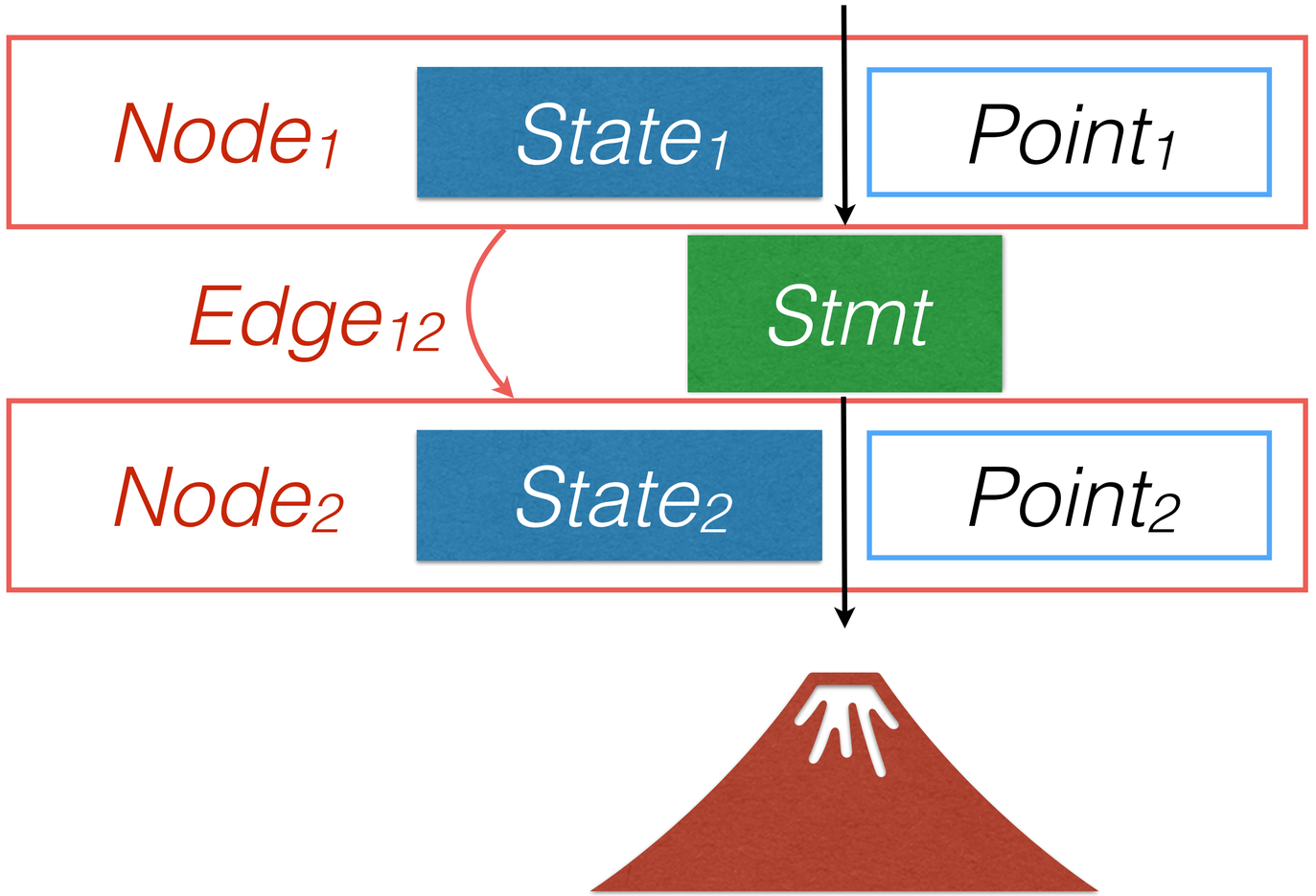
# Exploded Graph: Paths Through CFG

- *Nodes*: (*Point*, *State*) pairs
  - **Program Point:** A point between statements (usually)
  - **Program State:** A record of effects of statements evaluated so far

- *Edges*: An edge from (*Point*$_1$, *State*$_1$) to (*Point*$_2$, *State*$_2$) means that the statement between *Point*$_1$ and *Point*$_2$ updates *State*$_1$ to *State*$_2$



# Exploded Graph Edges

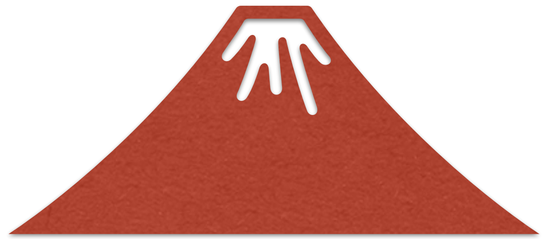


# Effects of Assignments: Store

**Statement:**
$x = 7$

**Program State:**
Nothing Yet!

**Program State:**
**Store:**
$x \rightarrow 7$

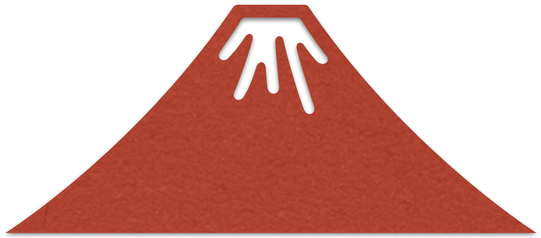


# Values of Expressions: Environment

**Statement:**
`x + 5;`

**Program State:**
**Store:**
`x -> 7`

**Program State:**
**Store:**
`x -> 7`
**Exprs:**
`x + 5 -> 12`

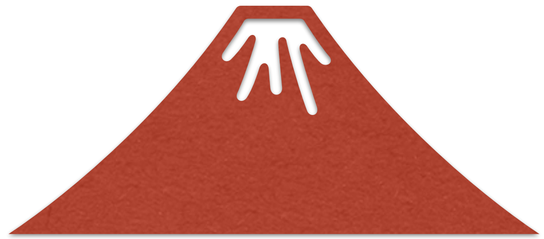


# Focus on One Operation at a Time

**Statement:**
`(x + 5) / 2;`

**Program State:**
**Exprs:**
`x + 5 -> 12`

**Program State:**
**Exprs:**
`(x + 5) / 2 -> 6`



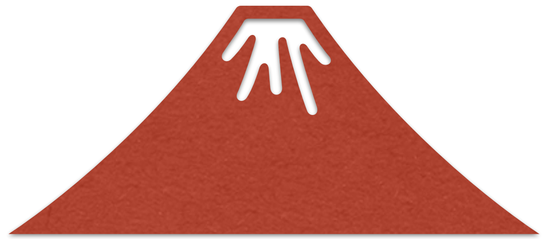
# What If It's Not In The Store?

**Statement:**
`x;`

**Program State:**
Nothing Yet!

**Program State:**
**Exprs:**
`x -> reg_$0<int x>`

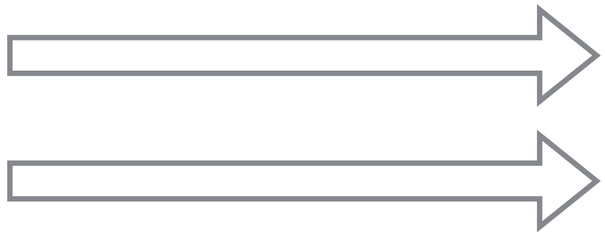
```
// example:
int foo(int x) {
  return x;
}
```



# Effects of Branches: Constraints

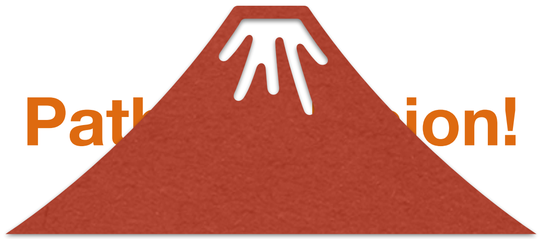
**Program State:**
**Exprs:**
`x -> reg_$0<int x>`

**Statement:**
`if (x > 5) …`



**Program State:**
**Ranges:**
`reg_$0<int x> > 5`

**Program State:**
**Ranges:**
`reg_$0<int x> <= 5`



# Symbolic Execution Recipe

- Just execute the program as you normally would
- Don’t know the value? – Denote it with a symbol
- Branch depends on a symbol? – Split up, record constraints
- Don’t explore paths on which constraints contradict each other

# Demo:
# How to Fix a Static Analyzer Bug in 24 minutes!

# Summary!

- Static Analyzer finds bugs by exploring sequences of events that may occur during the execution of the program.

- You can understand and study the internal logic of the static analyzer by looking at exploded graph dumps and setting conditional breaks on individual nodes.

- Sometimes these graphs are huge, so you should use `utils/analyzer/exploded-graph-rewriter.py` with various flags to extract useful information from the dump.

- See clang-analyzer.llvm.org for more information!