CMPT 473
Software Quality Assurance

A Brief Intro to Automated Test Generation

Nick Sumner

Our Test Suites Are Still Limited

- There is only so much we can include
- Even covering interesting interactions is a challenge

Our Test Suites Are Still Limited

- There is only so much we can include
- Even covering interesting interactions is a challenge

Our first naive solution may not have been naive!

```
for test in allPossibleInputs:
    run_program(test)
```

Our Test Suites Are Still Limited

- There is only so much we can include
- Even covering interesting interactions is a challenge

Our first naive solution may not have been naive!

```
for test in allPossibleInputs:
    run_program(test)
```

How might this be pragmatically useful?

Automated Test Generation

- We can *continuously* run new tests

Automated Test Generation

- We can *continuously* run new tests
  - But manual testing this way won't work!

Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!
- Automated Test Generation
  - Use program analysis to derive new tests without the user

Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!
- Automated Test Generation
  - Use program analysis to derive new tests without the user
- 2 approaches are increasingly common

Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!
- Automated Test Generation
  - Use program analysis to derive new tests without the user
- 2 approaches are increasingly common
  - Fuzz Testing

Automated Test Generation

- We can continuously run new tests
  - But manual testing this way won't work!
- Automated Test Generation
  - Use program analysis to derive new tests without the user
- 2 approaches are increasingly common
  - Fuzz Testing
  - Symbolic Execution

Fuzz Testing

- An approach for generating test inputs

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

```
./grep “02d6…” RandomFile
```

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]

```
./grep "02d6…" RandomFile
```

It was distressingly effective at finding buffer overflows (25%-33% of programs).

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) Generational (model based)

  - Creates entirely new inputs

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) Generational (model based)
    - Creates entirely new inputs
    - Needs a *model* for the input

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) *Generational* (model based)

  - Creates entirely new inputs
  - Needs a *model* for the input

  a*bc(d|e)c*

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) Generational (model based)
  - Creates entirely new inputs
  - Needs a *model* for the input

a*bc(d|e)c*

$A \rightarrow aAb$
$A \rightarrow cA$
$A \rightarrow \varepsilon$

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types

  1) *Generational* (model based)

    - Creates entirely new inputs
    - Needs a *model* for the input

a*bc(d|e)c*

...

$A \to aAb$
$A \to cA$
$A \to \varepsilon$

Fuzz Testing

- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
  2) *Mutational* (heuristic change based)
     - Modify an existing test suite

Fuzz Testing

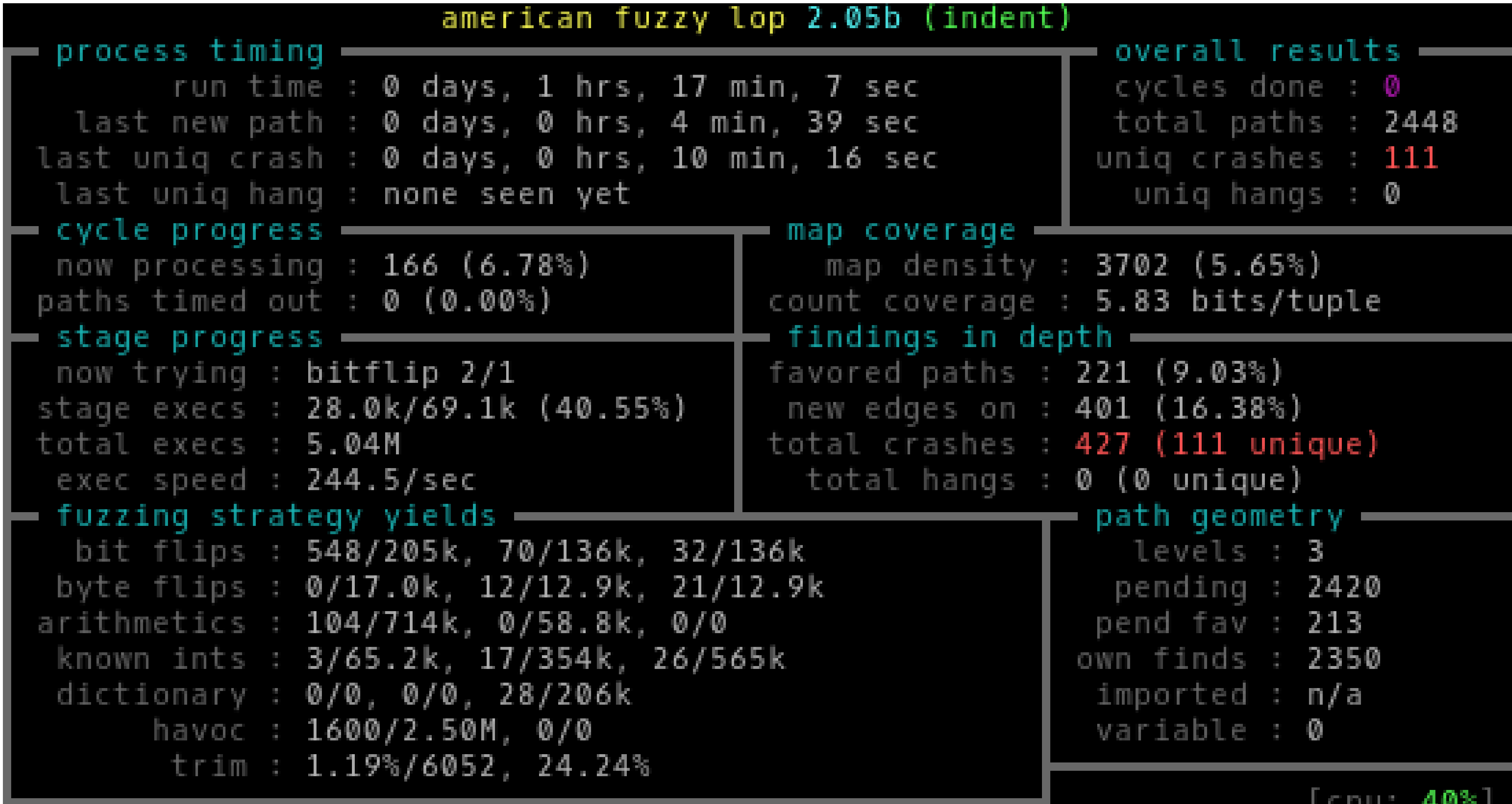
- An approach for generating test inputs
- Originally just feeding large random inputs to programs [Miller 1990]
- Now 2 main types
  1) *Generational* (model based)
  2) *Mutational* (heuristic change based)
     - Modify an existing test suite
     - Seeing a resurgance via *AFL* & *libFuzzer*

American Fuzzy Lop

- Increasingly used mutational fuzzer
  - Effective at finding buffer overflows

American Fuzzy Lop

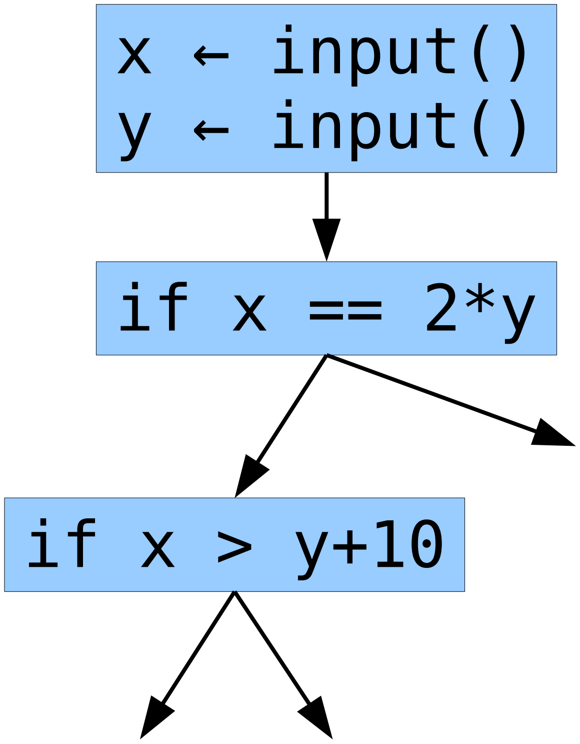
- Increasingly used mutational fuzzer



Symbolic Execution

Cadar & Sen, 2013

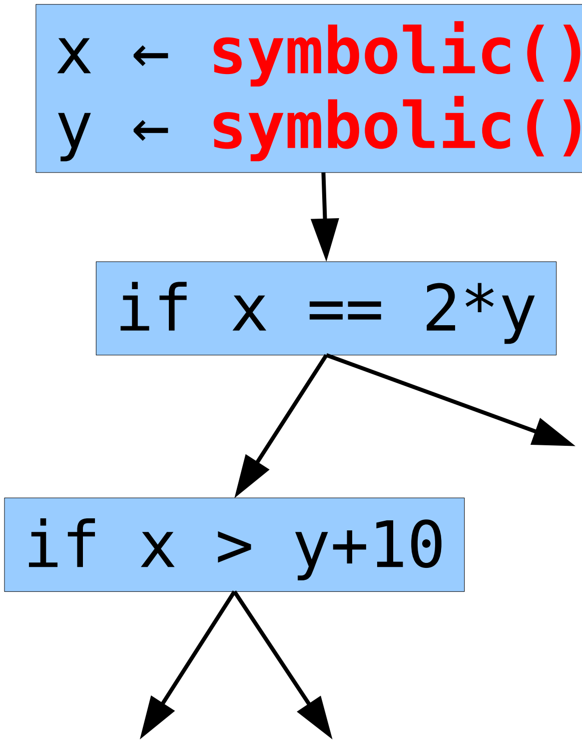
- An approach for generating test inputs.



Symbolic Execution

Cadar & Sen, 2013

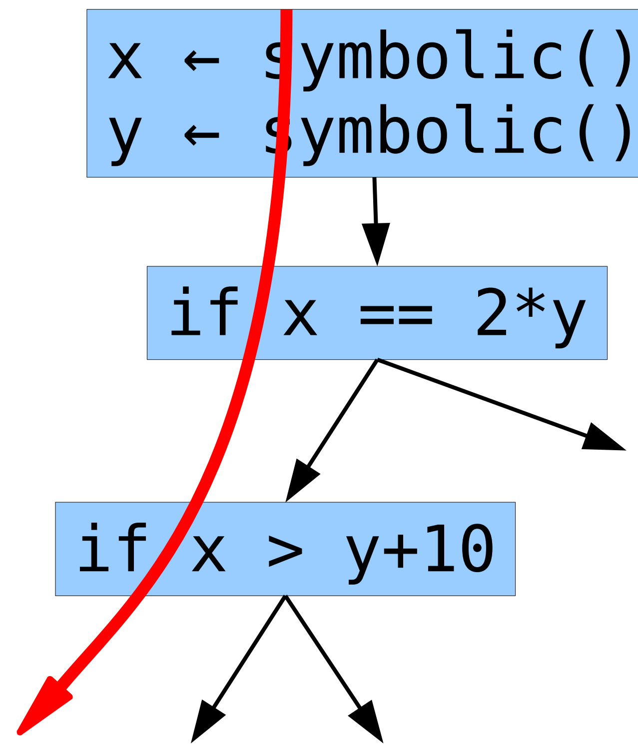
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values



Symbolic Execution

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute the program along a path using the symbolic values to build a formula over the input symbols.

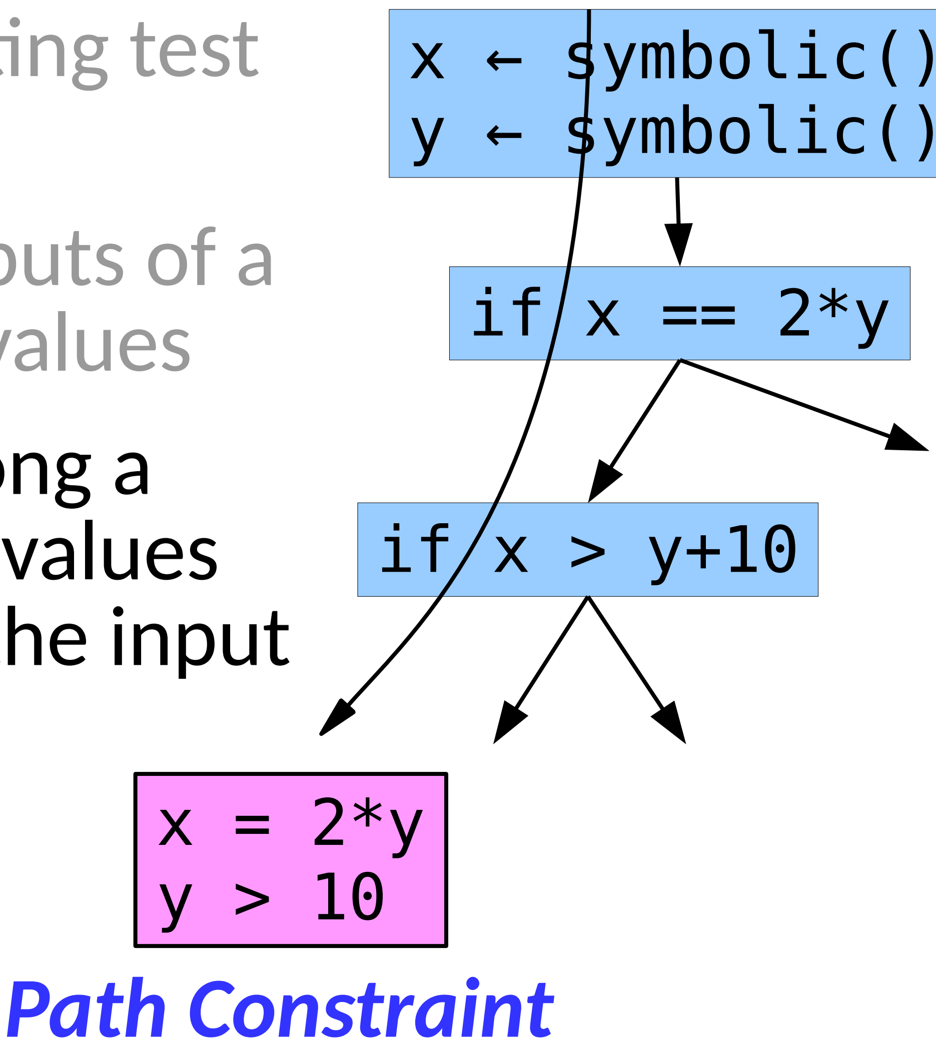
Cadar & Sen, 2013



Symbolic Execution

Cadar & Sen, 2013

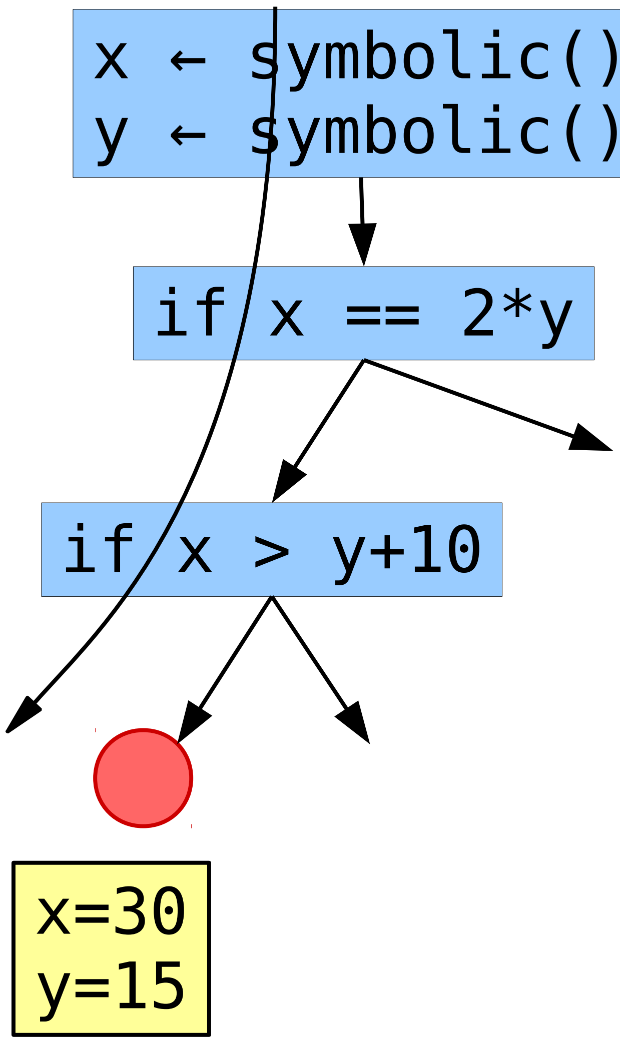
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute the program along a path using the symbolic values to build a formula over the input symbols.



Symbolic Execution

Cadar & Sen, 2013

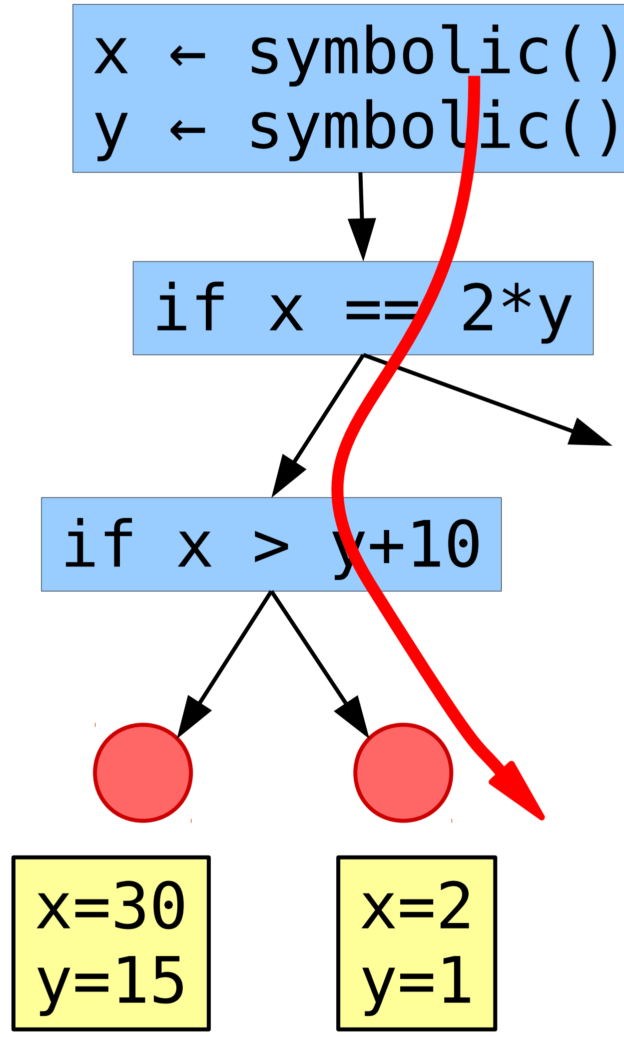
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute the program along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



Symbolic Execution

Cadar & Sen, 2013

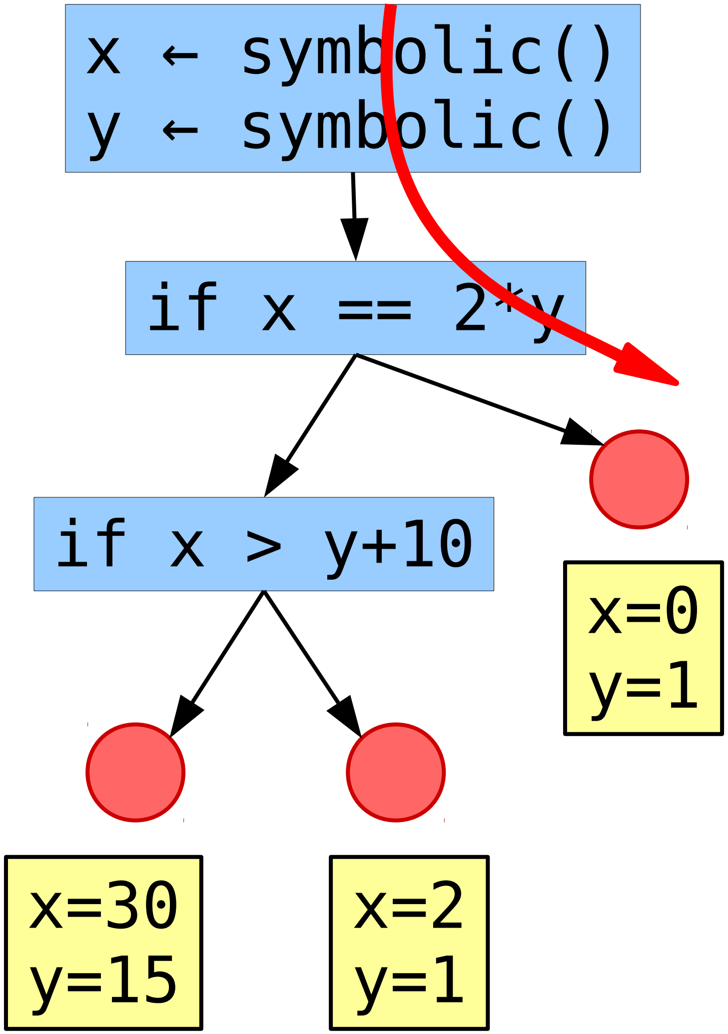
- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute the program along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



Symbolic Execution

Cadar & Sen, 2013

- An approach for generating test inputs.
- Replace the concrete inputs of a program with symbolic values
- Execute the program along a path using the symbolic values to build a formula over the input symbols.
- Solve for the symbolic symbols to find inputs that yield the path.



How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

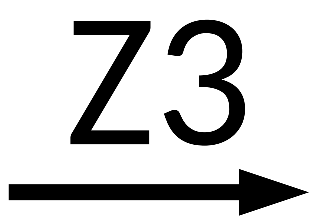
```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)



How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

```
x=22
y=11
```

```
sat
(model
    (define-fun y () Int 11)
    (define-fun x () Int 22)
)
```

How Can We Solve Constraints?

- SMT Solvers
  - Satisfiability Modulo Theories
  - SAT with extra logic
  - Standard interfaces through SMTLIB2

```
x = 2*y
y > 10
```

```
x=22
y=11
```

```
(declare-const x Int)
(declare-const y Int)
(assert (= x (* 2 y)))
(assert (> y 10))
(check-sat)
(get-model)
```

Z3 →

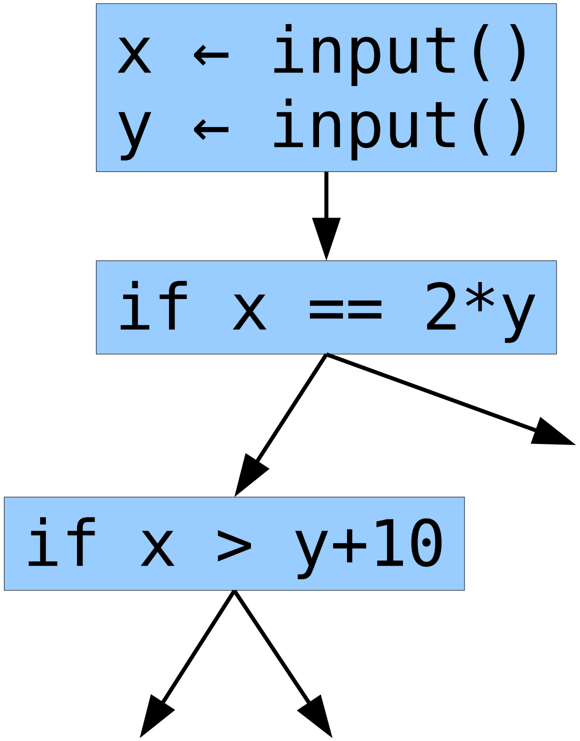
```
sat
(model
   (define-fun y () Int 11)
   (define-fun x () Int 22)
)
```

Try it online:
http://www.rise4fun.com/Z3/tutorial/

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.

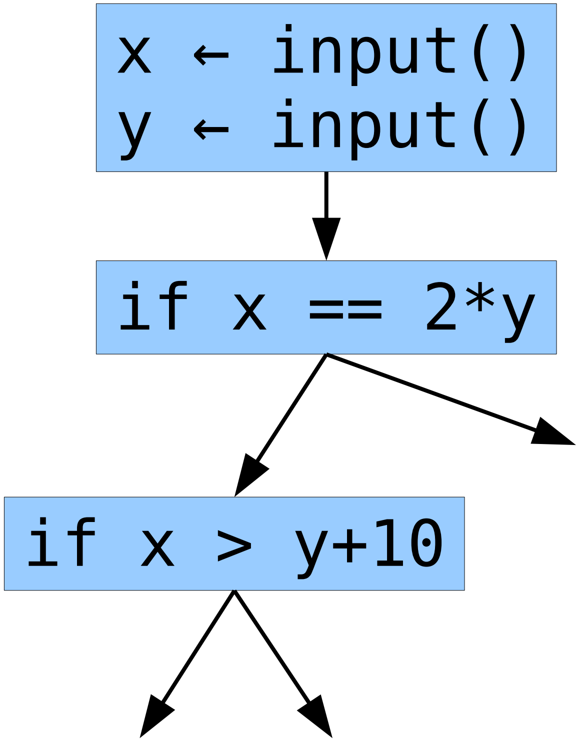
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.

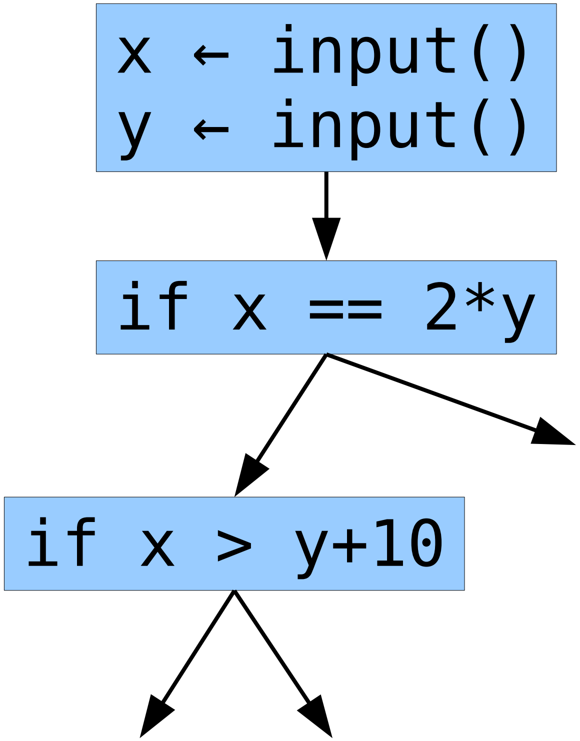
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches

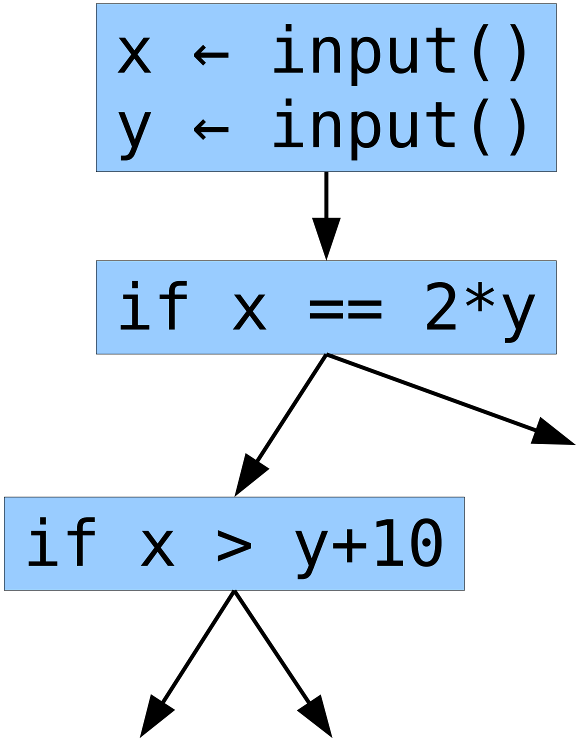
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  – Concolic (dynamic symbolic)

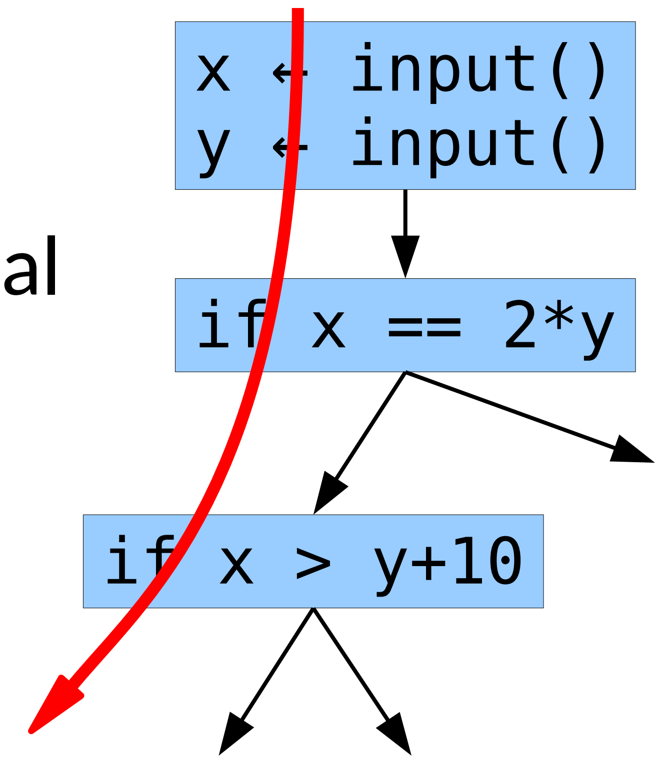
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

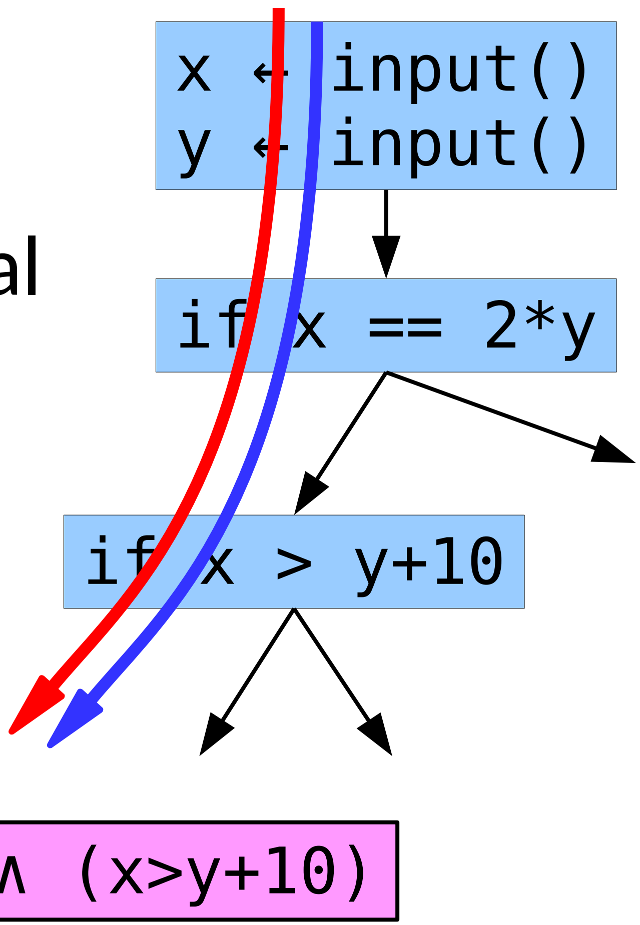
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

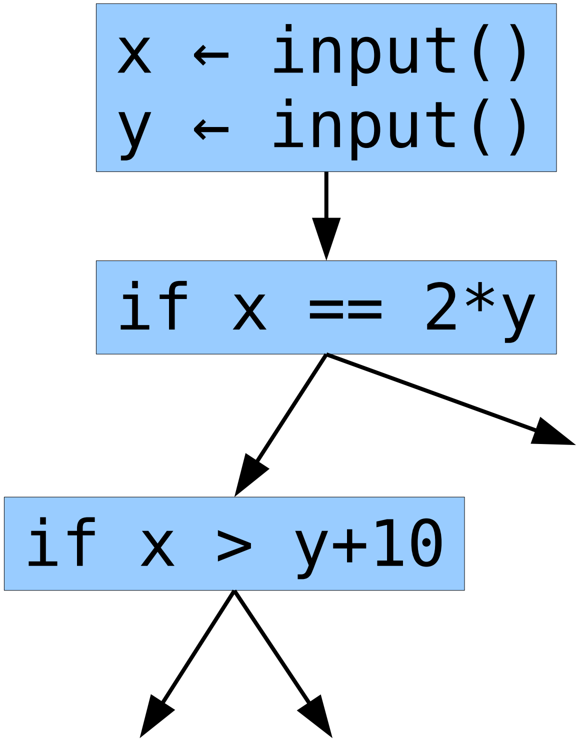
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

Cadar & Sen, 2013

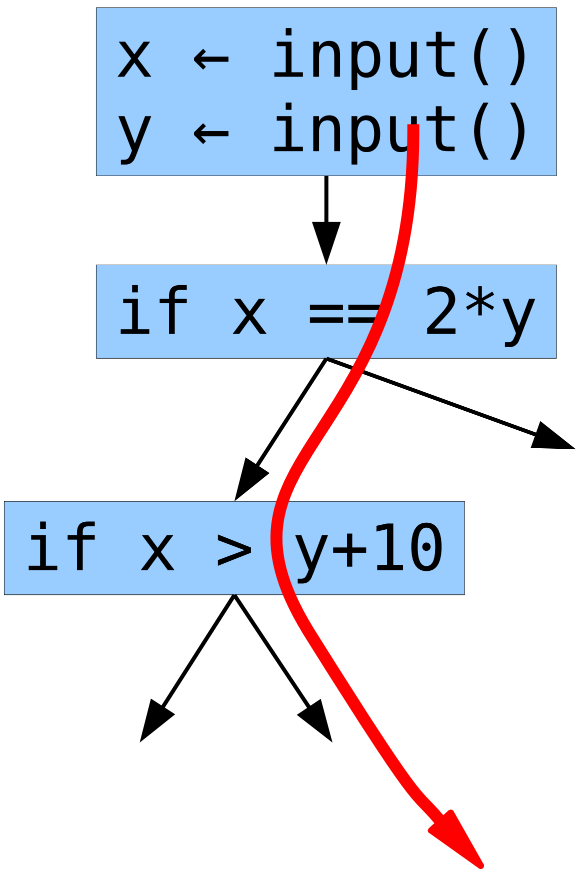


$(x=2*y) \land \neg(x>y+10)$

Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)

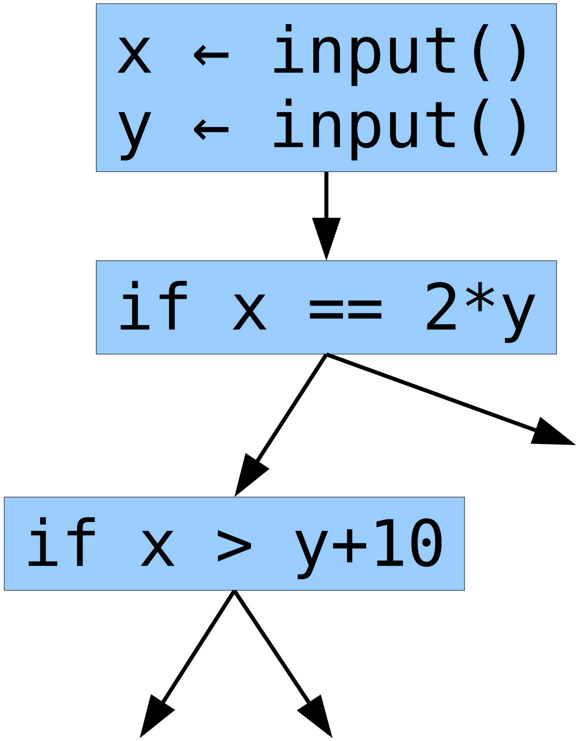
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

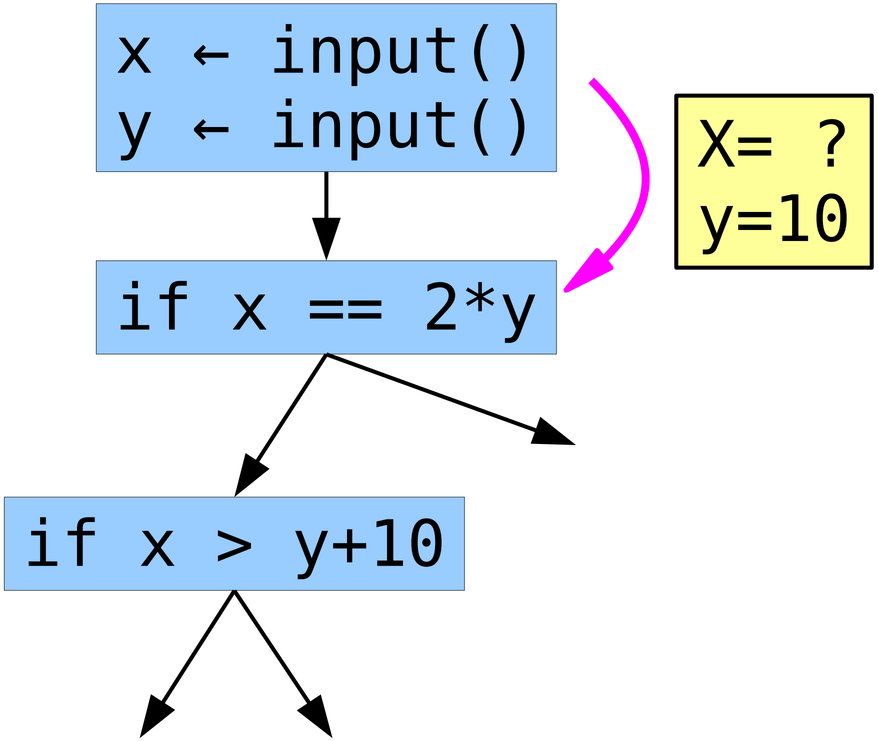
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic symbolic)
  - Execution Generated Testing

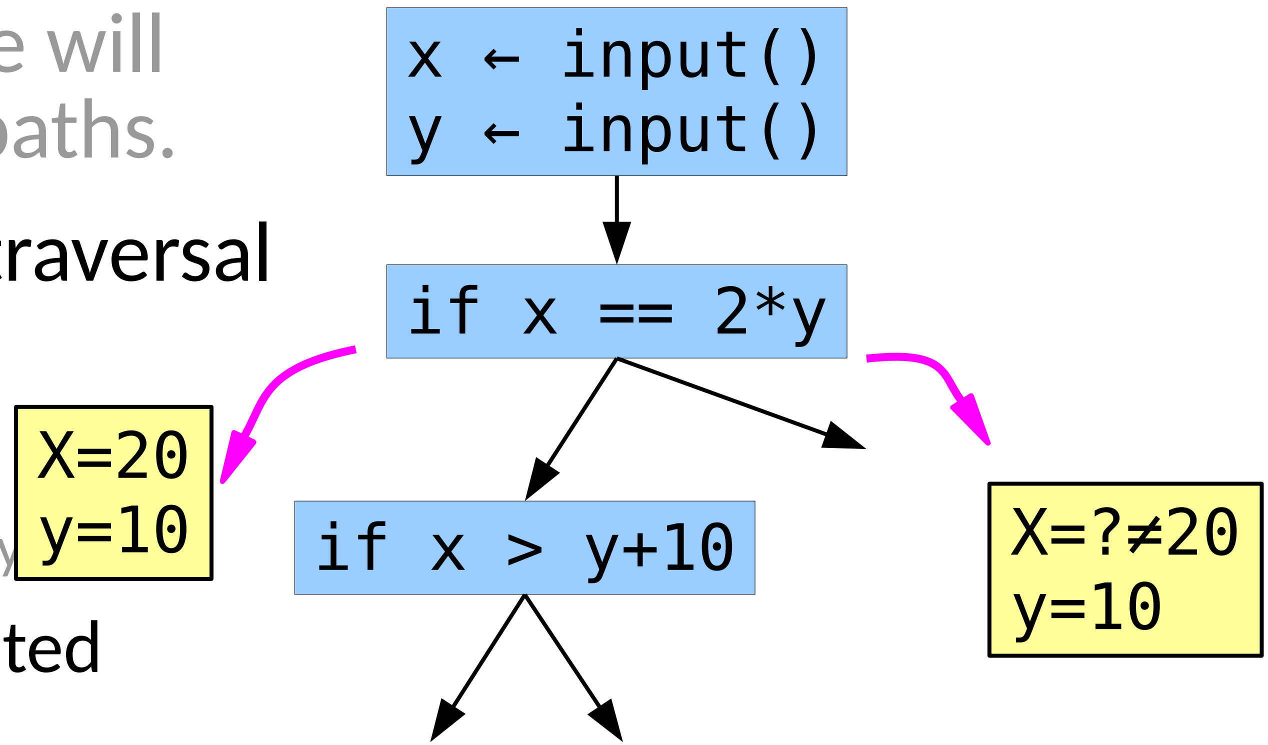
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic sy
  - Execution Generated Testing

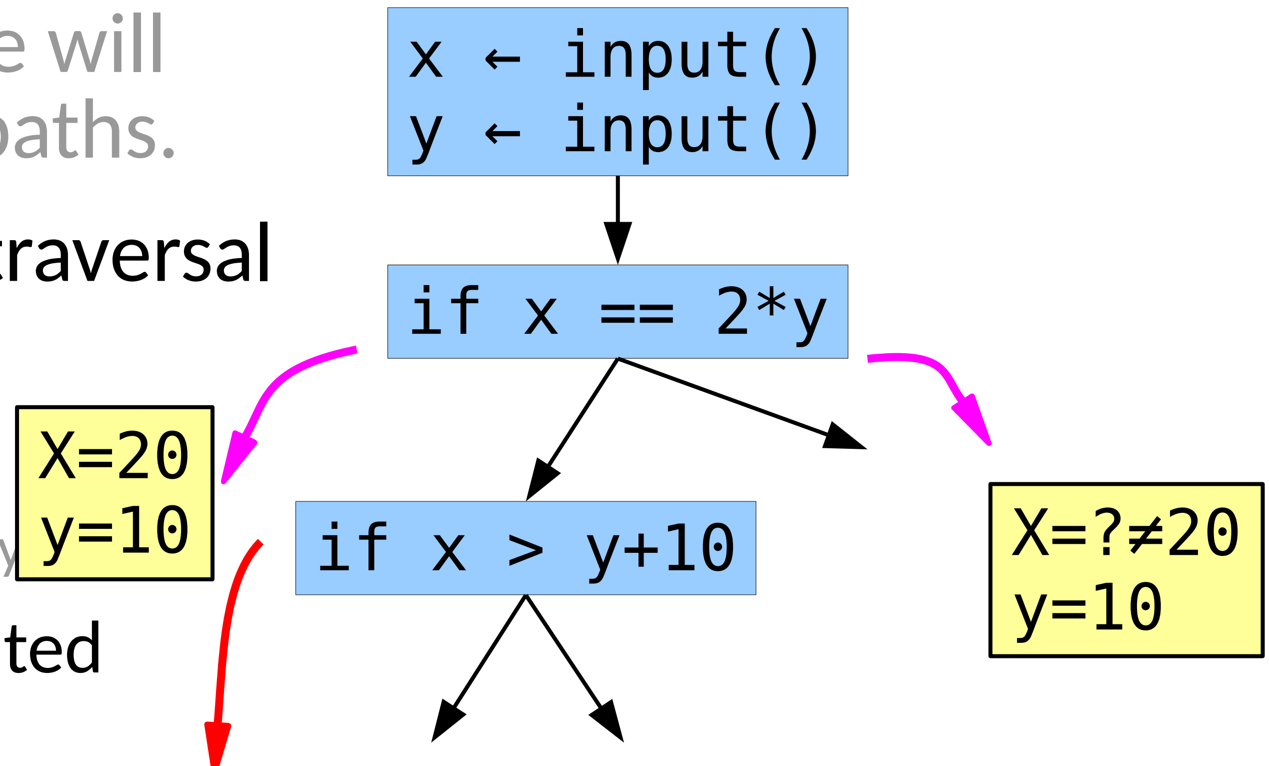
Cadar & Sen, 2013



Exploring the Execution Tree

- The possible paths of a program form an ***execution tree***.
- Traversing the tree will yield tests for all paths.
- Mechanizing the traversal yields two main approaches
  - Concolic (dynamic sy
  - Execution Generated Testing

Cadar & Sen, 2013



Execution on this side is concrete from this point on.

Symbolic Execution

- Increasingly scalable every year

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many used beyond just testing.

Symbolic Execution

- Increasingly scalable every year
- Can automatically generate test inputs from constraints
- The resulting symbolic formulae have many used beyond just testing.

Try it out:
https://github.com/klee/klee

Where They Fit in the Process

- Automated test generation is a continual process.

Where They Fit in the Process

- Automated test generation is a continual process.
- Just as much a part of modern QA as continuous integration

Where They Fit in the Process

- Automated test generation is a continual process.
- Just as much a part of modern QA as continuous integration
- Especially crucial as part of maintaining security (more on this later!)