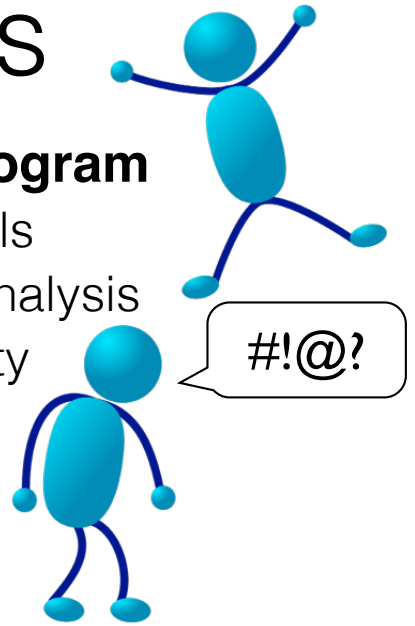
Introducing
Symbolic Execution

Software has **bugs**

- To **find them**, we use **testing** and **code reviews**

- But some **bugs are** still **missed**
  - *Rare features*
  - *Rare circumstances*
  - *Nondeterminism*

Static analysis



- Can **analyze all possible runs of a program**
  - An explosion of interesting ideas and tools
  - Commercial companies sell, use static analysis
  - Great potential to improve software quality

- But: **Can it find deep, difficult bugs?**
  - Our experience: yes, but **not often**
  - Commercial viability implies you must deal with developer confusion, false positives, error management,..
  - This means that companies specifically aim to keep the **false positive rate down**
    - They often do this **by purposely missing bugs**, to keep the analysis simpler

One issue: Abstraction

- Abstraction lets us model all possible runs
  - But **abstraction introduces conservatism**

- ***-sensitivities add precision**, to deal with this
  - * = **flow**-, **context**-, **path**-, *etc.*
  - But **more precise abstractions are more expensive**
    - Challenges scalability
    - Still have false alarms or missed bugs

- **Static analysis abstraction ≠ developer abstraction**
  - Because the developer didn't have them in mind

Symbolic execution

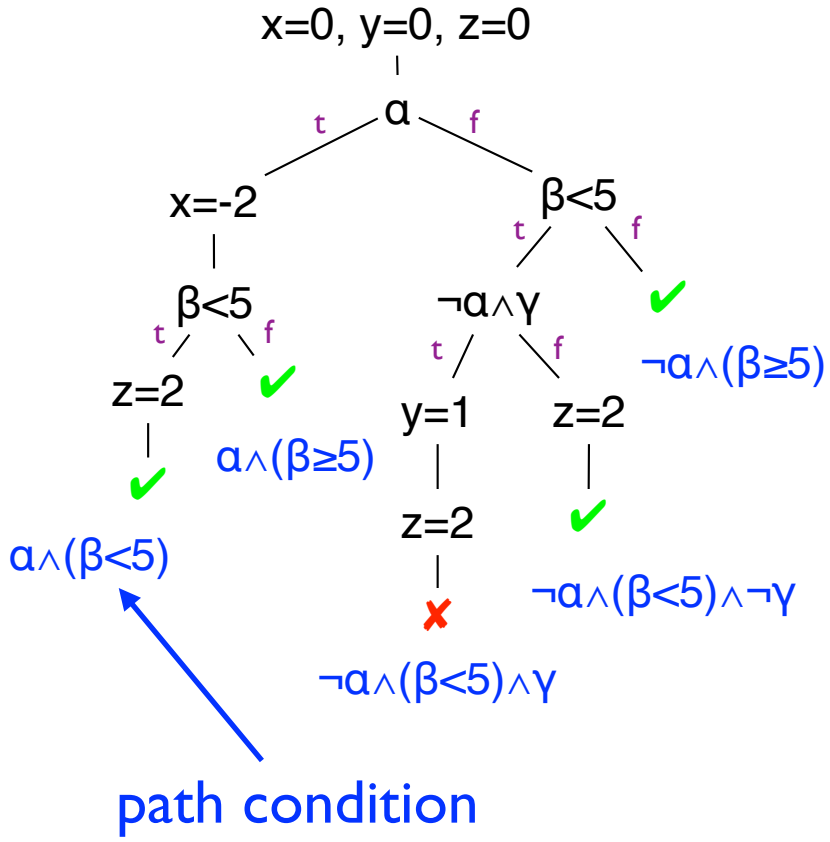
A middle ground

- Testing works: reported bugs are real bugs
  - But, **each test** only explores **one possible execution**
    - **assert(f(3) == 5)**
    - In short, **complete**, but **not sound**
  - We *hope* test cases generalize, but no guarantees

- **Symbolic execution generalizes testing**
  - "More sound" than testing
  - Allows unknown symbolic variables α in evaluation
    - **y = α; assert(f(y) == 2*y-1);**
  - If execution path depends on unknown, conceptually fork symbolic executor
    - **int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }**

Symbolic execution example

```
1. int a = α, b = β, c = γ;
2.                    // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z != 3)
```



Insight

- Each **symbolic execution path** stands for ***many*** actual **program runs**
  - In fact, exactly the set of runs whose concrete values satisfy the path condition
- Thus, we can **cover a lot more of the program's execution space than testing**
- Viewed **as a static analysis, symbolic execution** is
  - **Complete**, but not sound (usually doesn't terminate)
  - **Path, flow, and context sensitive**