CS 162 Programming languages

Lecture 15: Course Review & Software Robustness

Yu Feng

Winter 2021

A typical flow of a compiler

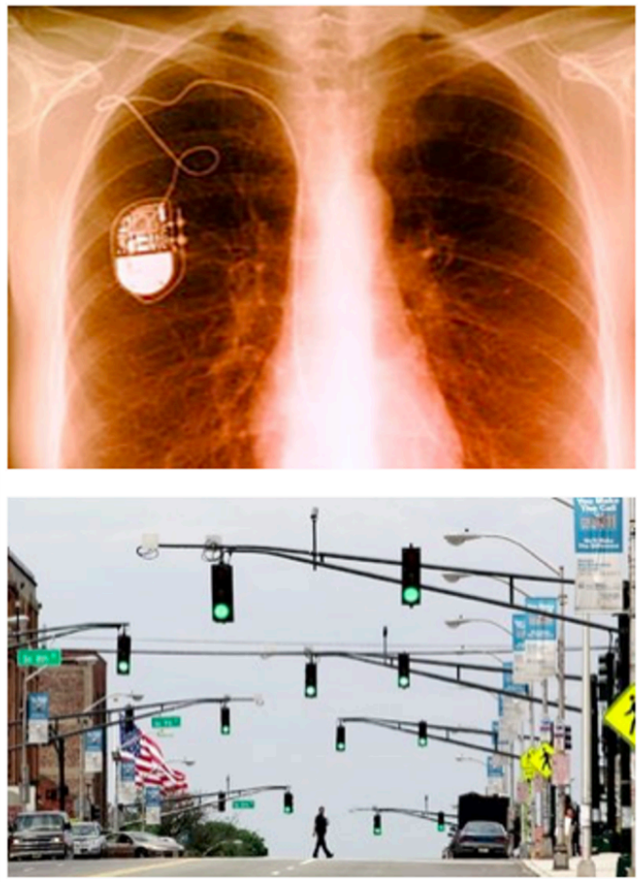
Want to learn compiler? Check out my CS160 in Fall

What we have learned

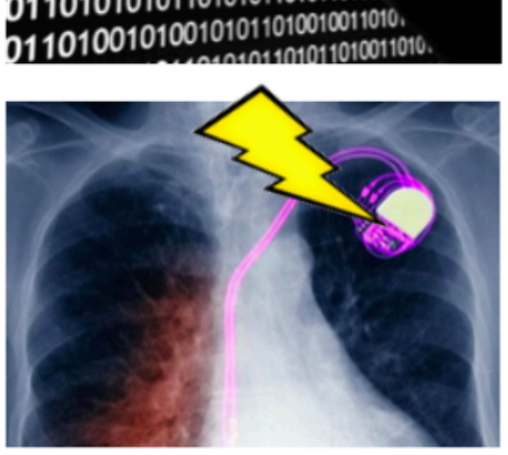
- Core PL
  - Lambda calculus (alpha-renaming, beta-reduction)
  - Operational semantics
  - Type checking
  - Type inference
- Functional programming
  - Higher-order functions, recursions
  - Data types


WHO
CARES?

Software is great



Software is NOT so great



How to make a robust bridge

Make it thicker!

How to make robust software

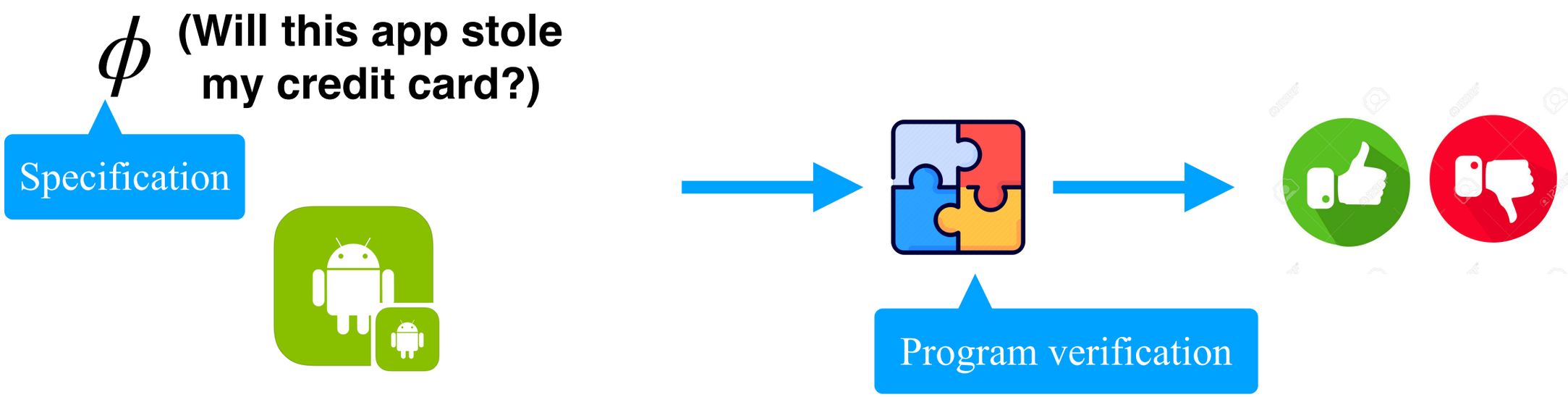
Testing

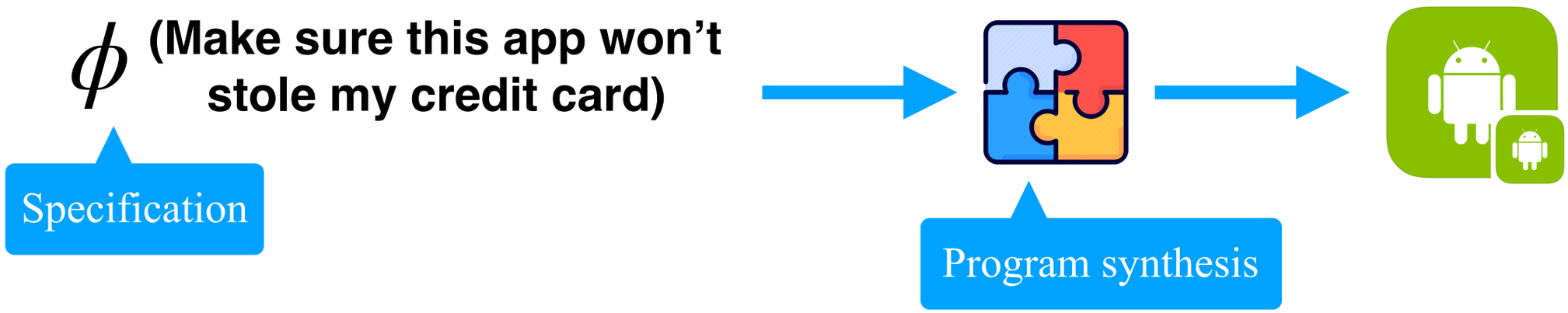
Screening

It is undecidable!

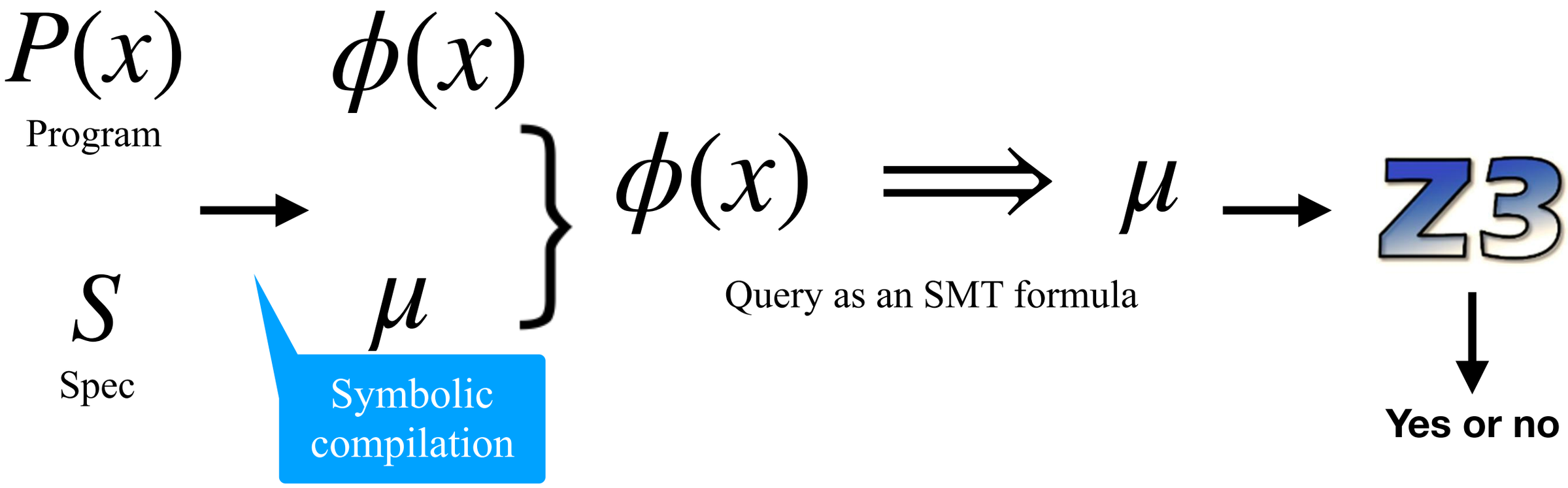
Let me show you a demo with a concrete example

Ensure robustness with PL





Checking correctness



Symbolic compilation can take years of effort!

https://github.com/Z3Prover/z3

Checking correctness

```
foo () {
  x = 10;
  y = 5;
}
```

$$x = 10 \wedge y = 5$$

```
foo (int a) {
  if (a > 0)
    x = 10;
  else
    y = 5;
}
```

$$a > 0 \implies x = 10 \wedge a <= 0 \implies y = 5$$

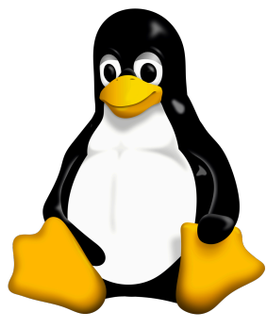
```
foo (int a) {
  if (a > 0)
    x = 10;
  else
    y = 5;
  assert y > 4
}
```

$$(a > 0 \implies x = 10 \wedge a <= 0 \implies y = 5)$$

$$\implies y > 4$$

Checking correctness

??



??

How to deal with complex systems?

It is undecidable!

A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and debugging.

Solver-aided programming

```
p(x) {
  v = 12
}
assert safe(x, p(x))
```

```
p(x) {
  v = ??
  …
}
assert safe(x, p(x))
```

Find an input on which the program fails.

Localize bad parts of the program.

Find values that repair the failing run.

Find code that repairs the program.

Racket

- Like ML, functional focus with imperative features: Anonymous functions, closures, pattern-matching, etc

- No static type system: accepts more programs, but most errors do not occur until run-time

- Advanced features like macros, modules, quoting/eval, continuations, contracts, ... (Will do only macros)

File structure

- Start every file with a line containing only
  `#lang racket`

- A comment starts with semicolon “;”

- A file is a module containing a *collection of definitions* (bindings)

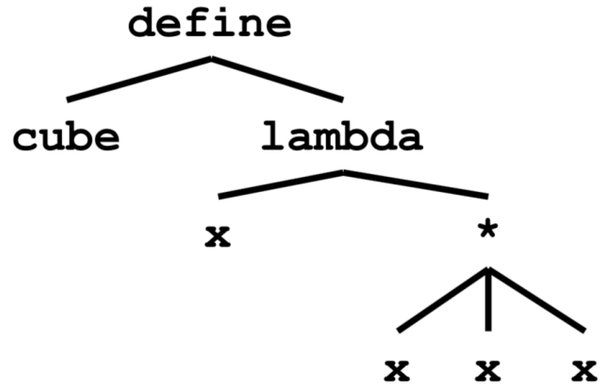
Example

```
#lang racket

(define x 3)
(define y (+ x 2))

(define cube ; function
  (lambda (x)
    (* x (* x x))))

(define pow ; recursive function
  (lambda (x y)
    (if (= y 0)
      1
      (* x (pow x (- y 1))))))
```

Why is this good?

- By parenthesizing everything, converting the program text into a tree representing the program (*parsing*) is trivial and unambiguous
  - Atoms are leaves
  - Sequences are nodes with elements as children
  - (No other rules)

```
(define cube
  (lambda (x)
    (* x x x)))
```



- Also makes indentation easy
- No need to discuss “operator precedence” (e.g., **x + y * z**)

Our old friend: currying/β-reduction

```
#lang racket

(define pow
  (lambda (x)
    (lambda (y)
      (if (= y 0)
          1
          (* x ((pow x) (- y 1)))))))

(define three-to-the (pow 3))
(define eightyone (three-to-the 4))
(define sixteen ((pow 2) 4))
```

Another old friend: list

- Empty list: **null**
- Cons constructor: **cons**
- Access head of list: **car**
- Access tail of list: **cdr**
- Append two lists: **append**
- Check for empty: **null?**
- Notes
  - Empty list is represented as **'() or (list) or null**
  - **(list e1 ... en)** for building lists

Another old friend: list

```
#lang racket

(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(define (my-append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (my-append (cdr xs) ys))))

(define (my-map f xs)
  (if (null? xs)
      null
      (cons (f (car xs)) (my-map f (cdr xs)))))
```