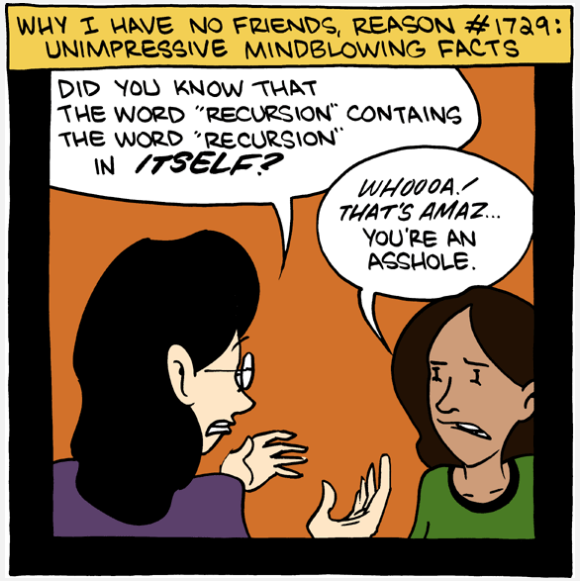
UMass Boston Computer Science

CS450 High Level Languages

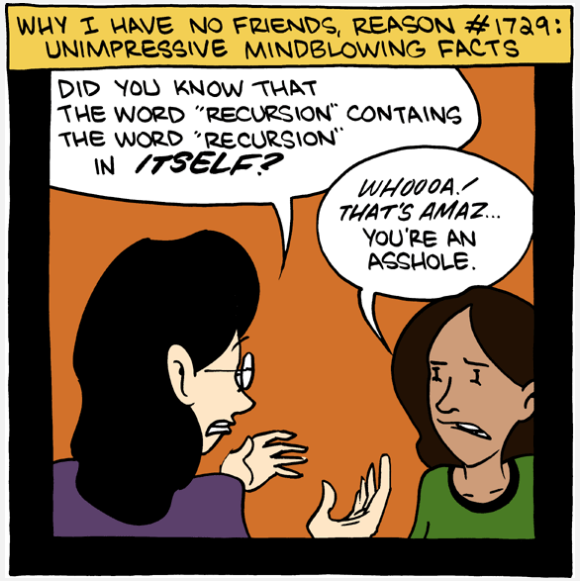
Recursion in the Lambda Calculus

Tuesday, March 25, 2025



Logistics

- HW 6 in
  - ~~due: Tue 3/25 11am EST~~
- HW 7 out
  - due: Tue 4/1 11am EST



Recursion vs Iteration: In Racket

Previously (Lecture 12)

Conclusion?
Recursion is <u>not</u> slower than iteration?

equivalent

Racket Recursion

```
;; iterative-sum-to : Nat -> Nat
;; Sums the numbers in the interval [0, x]
(define (iterative-sum-to x result)
  (if (zero? x)
      result
      (iterative-sum-to (sub1 x) (+ x result))))
```

accumulator

```
(time (iterative-sum-to BIG-NUMBER 0))
; cpu time: 15 real time: 13 gc time: 0
```

"for" in Racket is just
a macro (i.e., "syntactic sugar")
for a (tail) recursive function

Racket "Iteration"

```
(time (for/sum ([x (add1 BIG-NUMBER)]) x))
; cpu time: 15 real time: 6 gc time: 0
```

Racket **for** expressions

Generic "sequence"
(number, most data structures ...)

```
(for/list ([x lst]) (add1 x))
```

↓

```
(map add1 lst)
```

```
(for/list ([x n]) (add1 x))
```

↓

```
(build-list n add1)
```

```
(for/list ([x lst] #:when (odd? x)) (add1 x))
```

↓

```
(filter odd? (map add1 lst))
```

```
(for/sum ([x lst] #:when (odd? x)) (add1 x))
```

↓

```
(foldl + 0 (filter odd? (map add1 lst)))
```

Note:
These are still expressions!

Lots of variations!
(see docs)

Racket `for*` expressions

"nested" `for` loops

```
> (for* ([i '(1 2)]
         [j "ab"])
    (display (list i j)))
(1 a)(1 b)(2 a)(2 b)
```

```
> (for*/list ([i '(1 2)]
              [j "ab"])
    (list i j))
'((1 #\a) (1 #\b) (2 #\a) (2 #\b))
```

```
(for*/list (for
(for*/lists (id
  body-or-break
(for*/vector ma
(for*/hash (for
(for*/hasheq (f
(for*/hasheqv (
(for*/hashalw (
(for*/and (for-
(for*/or (for-c
(for*/sum (for-
(for*/product (
(for*/first (fo
(for*/last (for
(for*/fold ([ac
  body-or-break
(for*/foldr ([a
            (for
```

Useful in HW7?

Lots of variations! (see docs)

Previously (Lecture 6)

Kinds of Data Definitions

- Basic data
  - E.g., **numbers, strings,** etc
- Intervals
  - Data that is from a **range of values,** e.g., [0, 100)
- Enumerations
  - Data that is **one of a list of possible values**, e.g., `"green"`, `"red"`, `"yellow"`
- Itemizations
  - Data value that can be from a **list of possible other data definitions**
  - E.g., **either** a **string** or **number** (Generalizes enumerations)
- Compound Data
  - Data that is a **combination of values from other data definitions**

Combo of …

HW7!

Itemization of Compound Data - Example

```
;; A Shape is one of:
;; - (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; - (mk-Circ [r : Num] [c : Color])
;; interp: fields are radius and color
;; Represents a shape to be drawn on a canvas
```

Itemization of Compound Data - Template

```
;; A Shape is one of:
;; - (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; - (mk-Circ [r : Num] [c : Color])
;; interp: fields are radius and color
;; Represents a shape to be drawn on a canvas
```

```
;; shape-fn : Shape -> ???
(define (shape-fn sh)
 (cond
  [(Rect? sh) … (rect-h sh) … (rect-w sh) … (rect-c sh) … ]
  [(Circ? sh) … (circ-r sh) … (circ-c sh) … ]))
```

Itemization of Compound Data – 2nd way

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
```

```
;; A Rectangle is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; A Circle is a (mk-Circ [r : Num] [c : Color])
;; interp: fields are radius and colors
```

Itemization of Compound Data – template

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
```

```
;; A Rectangle is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; A Circle is a (mk-Circ [r : Num] [c : Color])
                                                  colors
```

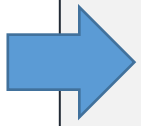
```
;; shape-fn : Shape -> ???
(define (shape-fn sh)
 (cond
  [(Rect? sh) … (rect-fn sh) … ]
  [(Circ? sh) … (circ-fn sh) … ]))
```

Itemization of Compound Data – function!

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
```

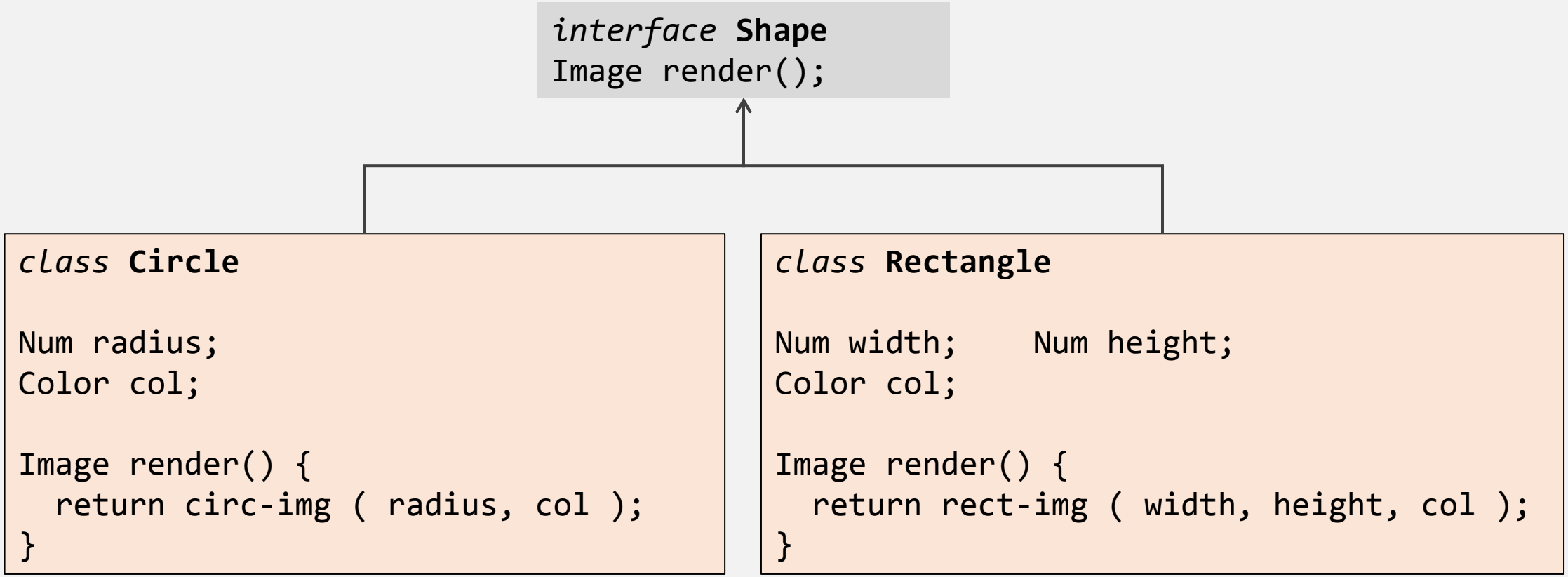
```
;; A Rectangle is a (mk-Rect [h : Num] [w : Num] [c : Color])
;; interp: fields are width, height, color
;; A Circle is a (mk-Circ [r : Num] [c : Color])
```

```
;; shape-fn : Shape -> ???
(define (shape-fn sh)
 (cond
  [(Rect? sh) … (rect-fn sh) … ]
  [(Circ? sh) … (circ-fn sh) … ]))
```

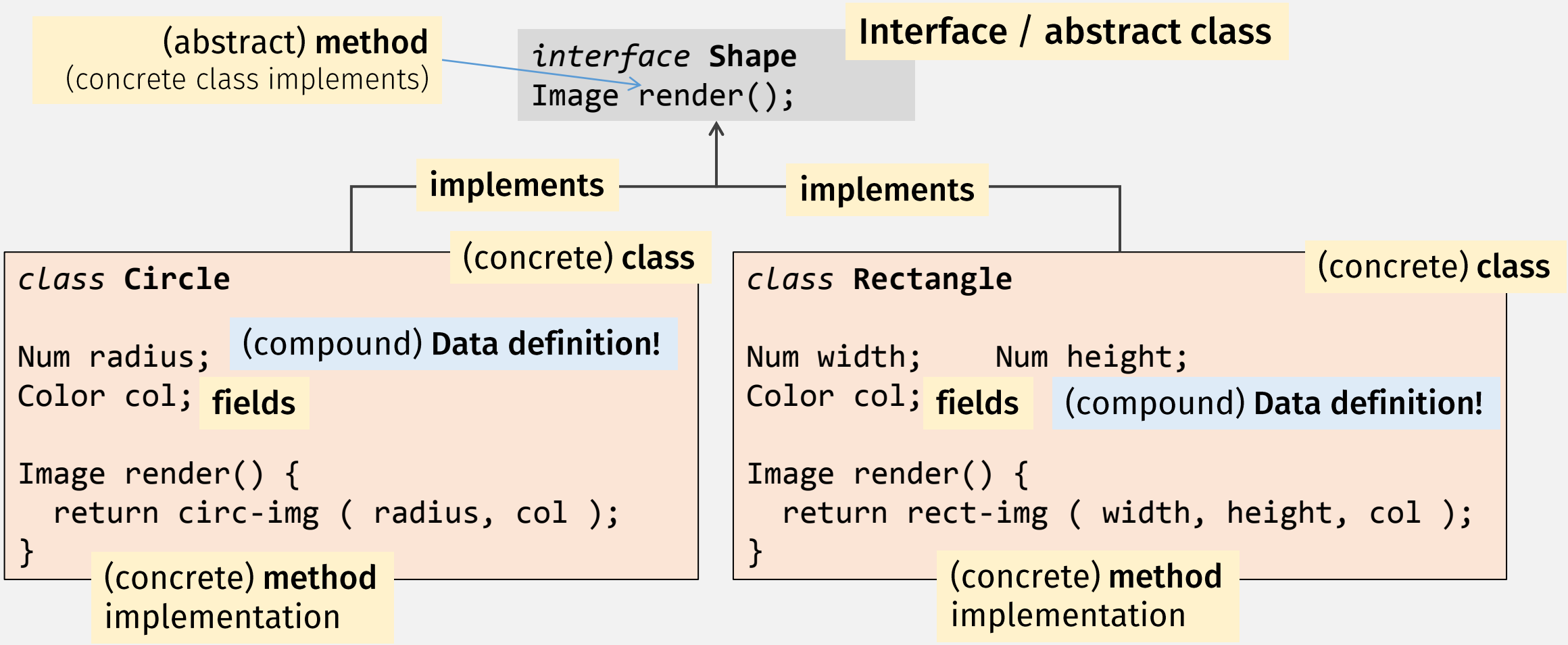


```
;; render : Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (rect-img sh)]
  [(Rirc? sh) (circ-img sh)]))
```

A Simple OO Example: Shapes



A Simple OO Example: Terminology



CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields, separate fns process data

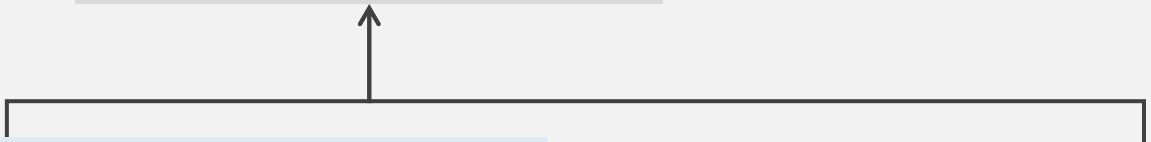
OO Programming

- **Compound data** (`class`) group fields and methods together!

A Simple OO Example: Compare to CS450

(itemization) **Data definition**

```
interface Shape
Image render();
```



(itemization) **Data definition** item

```
class Circle

Num radius;
Color col;

Image render() {
  return circ-img ( radius, col );
}
```

(compound) **Data definition**

function implementation (one **cond** clause) for **Shape** data (split up)

(itemization) **Data definition** item

```
class Rectangle

Num width;    Num height;
Color col;

Image render() {
  return rect-img ( width, height, col );
}
```

(compound) **Data definition**

function implementation (one **cond** clause) for **Shape** data (split up)

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined

OO Programming

- **Compound data** (`class`) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions

CS450 vs OO Comparison

CS 450 Design Recipe

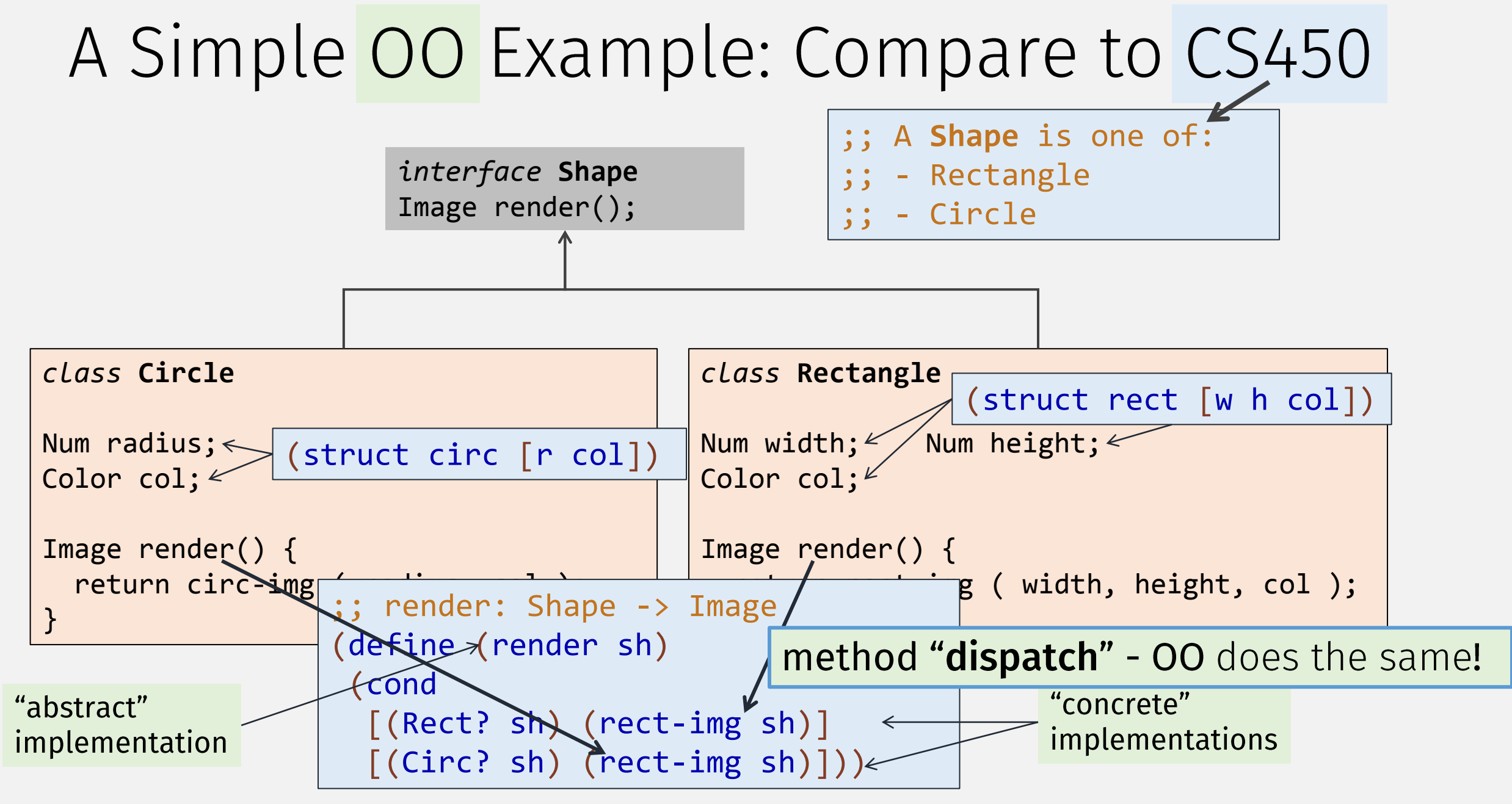
- **Compound data** (`struct`) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!

OO Programming

- **Compound data** (`class`) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!

1 function,
1 task, … processes
1 data definition!

A Simple OO Example: Compare to CS450



CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)

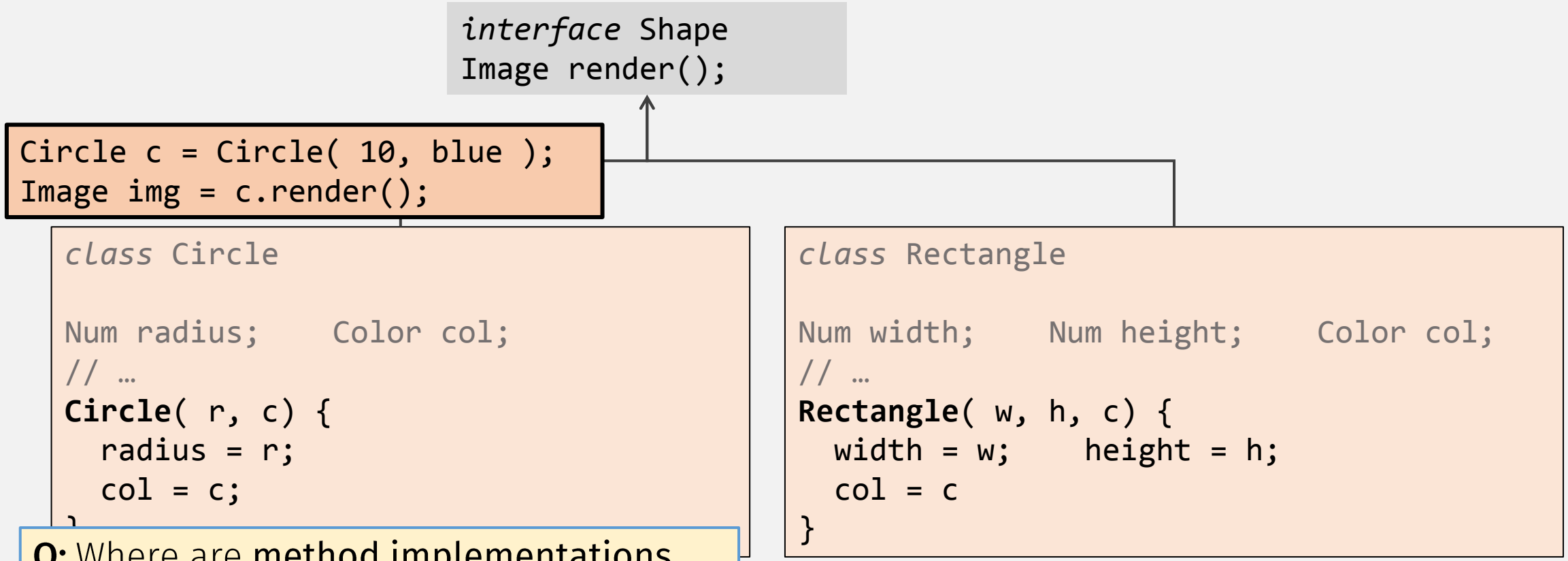
```
;; (explicit) render: Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (rect-img sh)]
  [(Circ? sh) (circ-img sh)]))
```

OO Programming

- **Compound data** (`class`) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**

```
;; (implicit) render: Shape -> Image
Image render (Shape sh)
  if (sh instanceof Rectangle){ rect-img(sh); }
  else if (sh instanceof Circle){ circ-img(sh); }
```

A Simple OO Example: Constructors



Q: Where are **method implementations** for an object instance **"stored"?**

A: It's **another** (hidden) **field** (see "**method table**")**!**

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) have fields, separate fns process data
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)
- Struct **Constructor** explicitly includes method defs ???

OO Programming

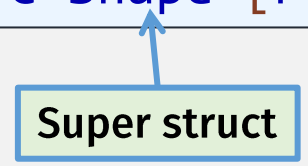
- **Compound data** (`class`) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**
- Object **Constructor** implicitly includes method defs

OO-style Constructors … with structs!

Shape “interface” definition

```
(struct Shape [render-method])
```

```
(struct circ Shape [r col])
```



Interlude: Inheritance and "Super" Structs

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
(struct rect [w h c])
(struct circ [r c])
```

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
(struct Shape [])
(struct rect Shape [w h c])
(struct circ Shape [r c])
```

"abstract" struct (implicitly defines **Shape?** predicate)

Alternatively …

"super" struct declaration

```
(define (Shape? s)
  (or (rect? s) (circ? s)))
```

e.g., if **r** = `(rect 1 2 'red)`
then both `(rect? r)` = true
and `(Shape? r)` = true

Useful in HW7?

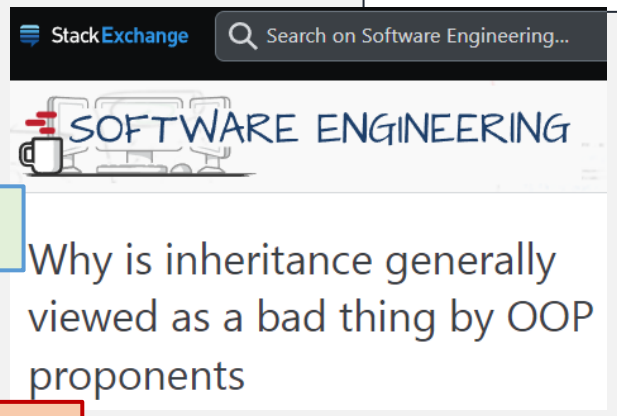
Interlude: Inheritance and "Super" Structs

This kind of **"polymorphic" "abstract" data definition** is what we've been creating all semester!

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
(struct Shape [])
(struct rect Shape [w h c])
(struct circ Shape [r c])
```

"abstract" struct (implicitly defines **Shape?** predicate)

"super" structs are just a **convenience for the same thing** (when all itemizations are structs)



WAIT, I heard "Inheritance is bad"???

NO, accepted OO principles says:

Inheritance of **implementations** is **bad** ☒ (violates "1 task, 1 function")

Interfaces and **abstract classes** are **ok** ☑ (i.e., "**itemizations**")

OO-style Constructors … with structs!

Shape “dispatch” function

```
;; render : Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (rect-img sh)]
  [(Circ? sh) (circ-img sh)]))
```

Shape “interface” definition

Method implementation (as a field)

```
(struct Shape [render-method])
(struct circ Shape [r col])
```

Super struct

circ constructor must be given 3 args

Shape constructors

```
(define (mk-Circ r col
                 [circ-render-fn circ-img])
  (circ circ-render-fn r col))
```

default

(make method an optional argument, with default)

Q: Where are **method implementations** for an object **instance “stored”?**

A: It’s **another** (hidden) **field!**

Then create same definitions for **rect** …

CS450 vs OO Comparison

CS 450 Design Recipe

- **Compound data** (`struct`) has (possibly function) fields!
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)
- Struct **Constructor** explicitly includes method defs

OO Programming

- **Compound data** (`class`) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**
- Object **Constructor** implicitly includes method defs

CS450 vs OO Comparison

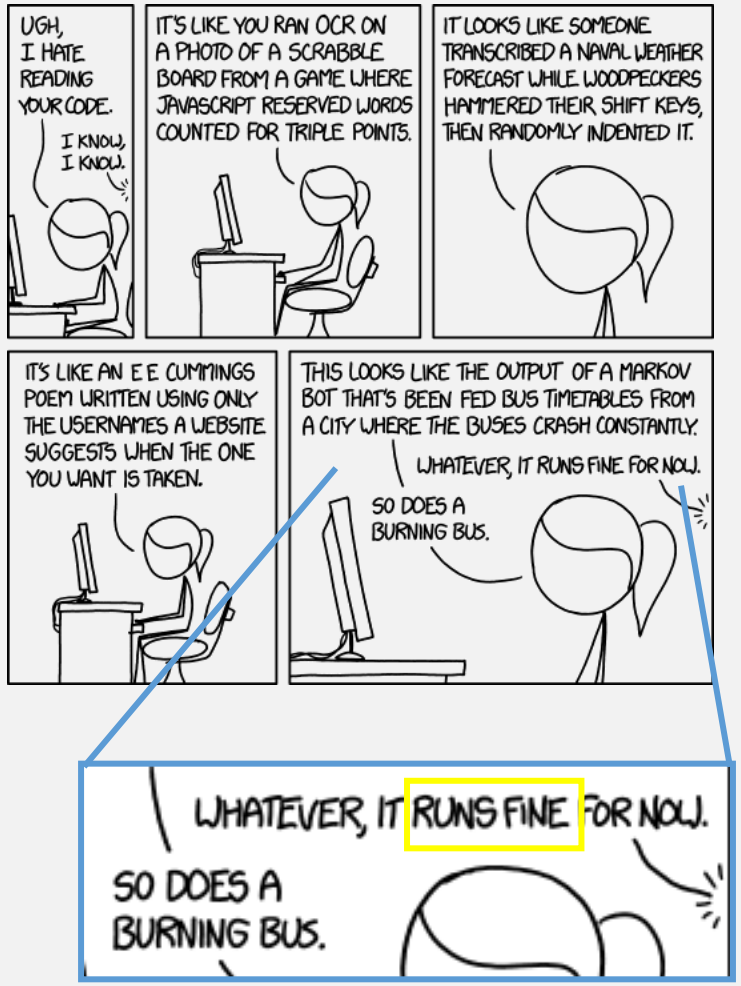
CS 450 Design Recipe

- **Compound data** (`struct`) has (possibly function) fields!
- **Itemization** Data Defs explicitly defined
- **Functions** organized by the kind of data they process!
- Explicit itemization **dispatch** (cond)
- **Constructor** explicitly includes method defs
- Data to process is explicit arg

OO Programming

- **Compound data** (`class`) group fields and methods together!
- **Itemization** Data Defs implied by interface / class definitions
- **Methods** organized by the kind of data they process!
- Implicit itemization **dispatch**
- **Constructor** implicitly includes method defs
- Data to process (“`this`”) is implicit arg

CS 450 so far …



So far, this class teaches:

- How to **use high-level languages**
- i.e., a **high-level programming "process"**
- i.e., a **language-agnostic design recipe** for **creating clean, readable programs**

- How to do well: **learn** and **follow the "process" (design recipe)**

- How to not do well: **just focus on "getting the code working"**
  - (code does not "run fine")

From Lecture 1

"high" level
(easier for humans to understand)

NOTE: This hierarchy is *approximate*

English	
Specification langs	Types? pre/post cond?
Markup (`html, markdown`)	tags
Database (`SQL`)	queries
Logic Program (`Prolog`)	relations
Lazy lang (`Haskell, R`)	Delayed computation
Functional lang (`Racket`)	Expressions (no stmts)
JavaScript, Python	"eval"
C# / Java	GC (no alloc, ptrs)
C++	Classes, objects
C	Scoped vars, fns
Assembly Language	Named instructions
Machine code	0s and 1s

This class: **how to program in a high-level more "human friendly" way**

"**Computation**" = "arithmetic" of expressions

"declarative"

Core model: **Lambda Calculus**

"**Computation**" = sequence of instructions / statements

"imperative"

Core model: **Turing Machines**

"low" level
(runs on cpu)

"Nicer" for humans to use

Last Time

The Lambda (λ) Calculus

- A “**programming language**” consisting of only:
  - Lambda functions
  - Function application

- **Equivalent** in “**computational power**” to
  - Turing Machines
  - Your favorite programming language!

Last Time

Church Numerals

```
;; A ChurchNum is a function with two arguments:
;; “f” : a function to apply
;; “base” : a base ("zero") value to apply to
;;
;; For a specific number, its "Church" representation
;; applies the given function that number of times
```

```
(define czero
  (lambda (f base) base))
```

f applied zero times

```
(define cone
  (lambda (f base) (f base)))
```

f applied one time

```
(define ctwo
  (lambda (f base) (f (f base))))
```

f applied two times

```
(define cthree
  (lambda (f base) (f (f (f base)))))
```

f applied three times

Church “Add1”

```
;; cplus1 : ChurchNum -> ChurchNum
;; “Adds” 1 to the given Church num
```

```
(define cplus1
  (lambda (n)
    (lambda (f base)
      (f (n f base)))))
```

Input ChurchNum

Returns ChurchNum that …

(we know “n” will apply f n times)

… adds an extra application of f

```
(define czero
  (lambda (f base) base))
```

```
(define cone
  (lambda (f base) (f base)))
```

```
(define ctwo
  (lambda (f base) (f (f base))))
```

```
(define cthree
  (lambda (f base) (f (f (f base)))))
```

Church Addition

```
;; cplus : ChurchNum ChurchNum -> ChurchNum
;; "Adds" the given ChurchNums together
```

```
(define cplus
  (lambda (m n)
    (lambda (f base)
      (m f (n f base)))))
```

Input ChurchNums

Returns a ChurchNum that …

(we know "n" will apply f n times)

… adds "m" extra applications of f

```
(define czero
  (lambda (f base) base))
```

```
(define cone
  (lambda (f base) (f base)))
```

```
(define ctwo
  (lambda (f base) (f (f base))))
```

```
(define cthree
  (lambda (f base) (f (f (f base)))))
```

Church Booleans

```
;; A ChurchBool is a function with two arguments,
;; where the representation of:
;; “true” returns the first arg, and
;; “false” returns the second arg
```

```
(define ctrue
  (lambda (a b) a))
```

Returns first arg

```
(define cfalse
  (lambda (a b) b))
```

Returns second arg

Review: "And"

The truth table of $A \wedge B$:

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False

When A = True, then $\mathrm{And}(A, B) = B$

When A = False, then $\mathrm{And}(A, B) = A$

Church “And”

```
;; cand: ChurchBool ChurchBool-> ChurchBool
;; “ands” the given ChurchBools together
```

The truth table of $A \wedge B$:

A	B	$A \wedge B$
True	True	True
True	False	False
False	True	False
False	False	False

When A = True,
want: And(A, B) = B ☑

When A = False,
want: And(A, B) = A ☑

```
(define cand
  (lambda (A B)
    (A B A)))
```

```
(define ctrue
  (lambda (a b) a))
```

(Returns first arg)

```
;; if A = ctrue
;; then (A B A) = B ☑
;; want (cand A B) = B
```

```
(define cfalse
  (lambda (a b) b))
```

(Returns second arg)

```
;; if A = cfalse
;; then (A B A) = A ☑
;; want (cand A B) = A
```

Church Pairs (Lists)

```
;; A ChurchPair<X,Y> 1-arg function, where
;; the arg fn is applied to (i.e., "selects") the X and Y data values
```

```
;; ccons: X Y -> ChurchPair<X,Y>
(define ccons
  (lambda (x y)
    (lambda (get)
      (get x y))))
```

```
(define cfirst
  (lambda (cc)
    (cc (lambda (x y) x))))
```

"Gets" the first item

```
(define csecond
  (lambda (cc)
    (cc (lambda (x y) y))))
```

"Gets" the second item

Last Time

The Lambda (λ) Calculus

- A **“programming language”** consisting of only:
  - Lambda functions
  - Function application

- “Language” has:
  - Numbers
  - Booleans and conditionals
  - Lists
  - …
  - Recursion?

Recursion in the Lambda Calculus

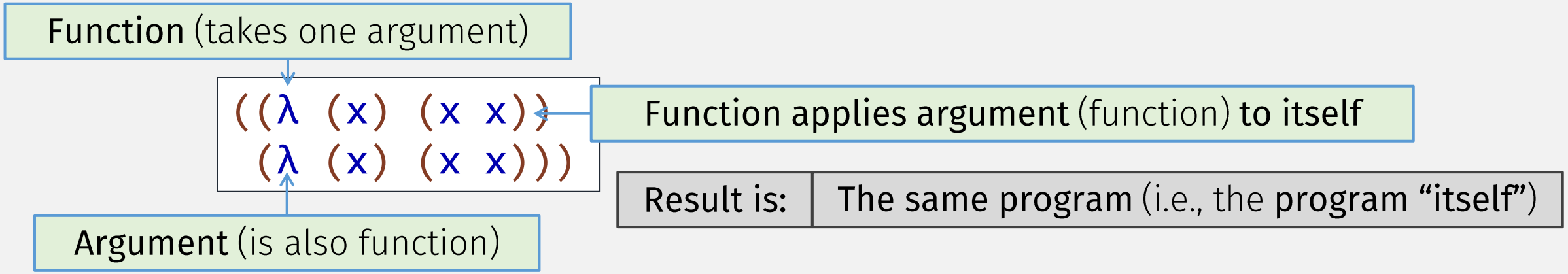
Q: How can we write recursive programs with no-name lambdas?

Q: Is there a way for a lambda program to reference itself?

From Lecture 2

Lambda Program that Knows "Itself"

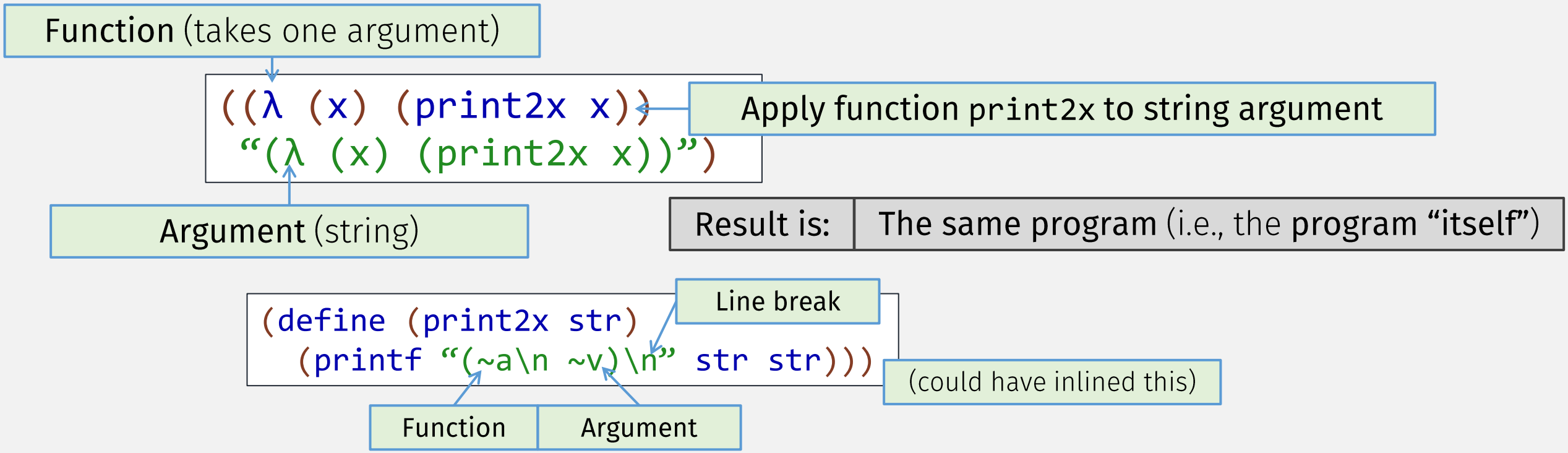
- Program that runs "itself" repeatedly (i.e., it infinite loops):



- Can we do something else besides loop?

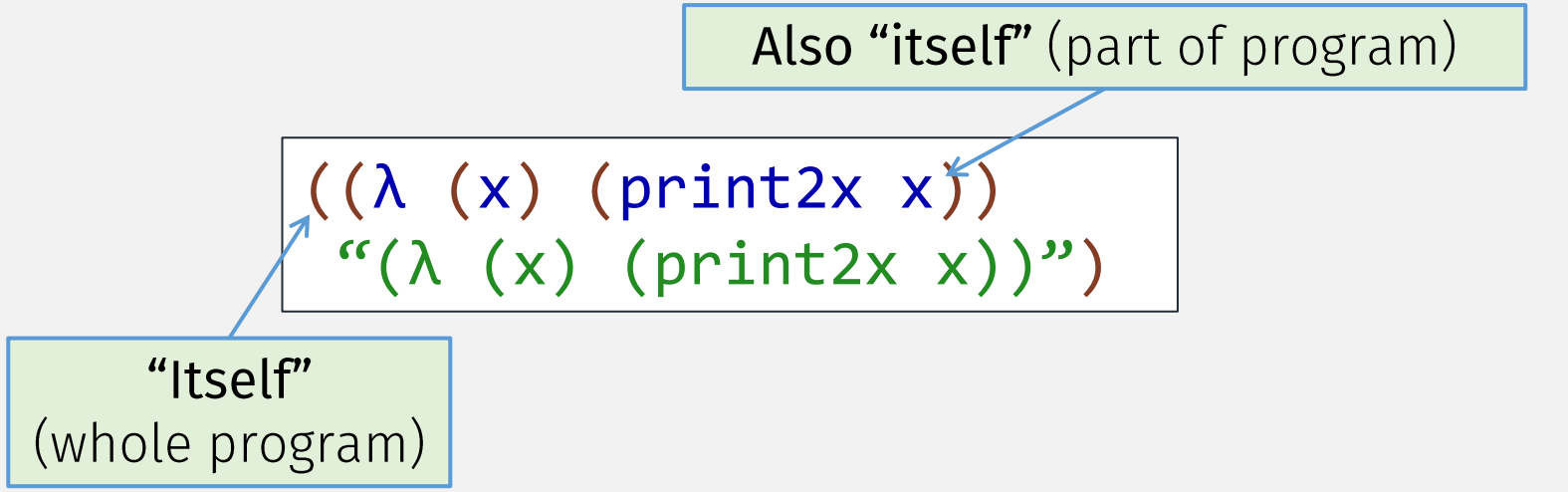
Lambda Program that Prints “Itself”

- Program that prints “itself”:



Lambda Program that Prints “Itself”

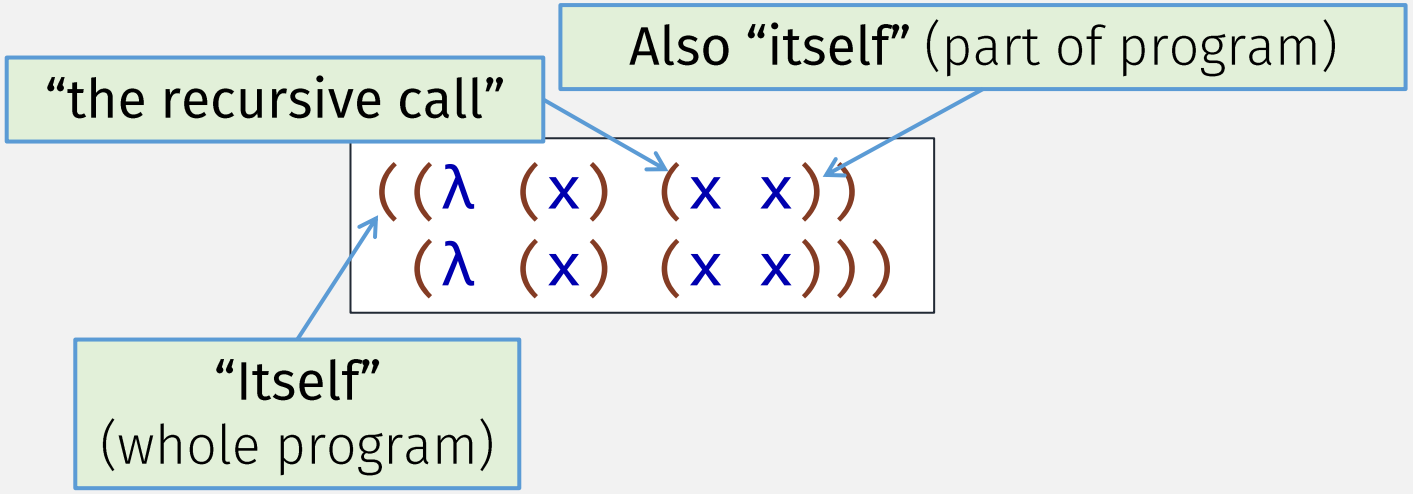
- Program that prints “itself”:



- Q: Which part of the program is “itself”?

Lambda Program that Knows “Itself”

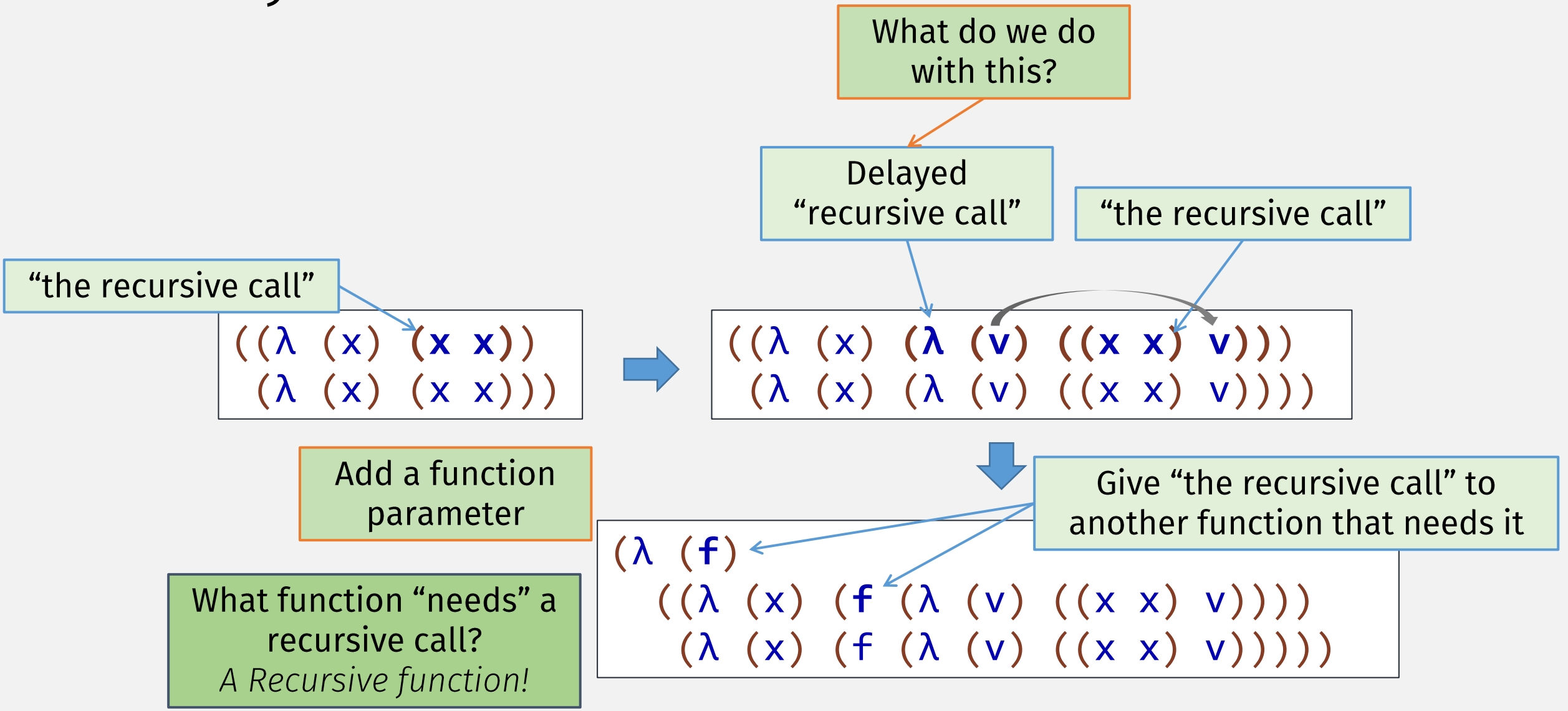
- Program that runs “itself” repeatedly (i.e., it infinite loops):



Insight:
“itself” = “the recursive call”

- Q: Which part of the program is “itself”?
- Can we do something more useful with “the recursive call”?

Delay “the recursive call”



A Recursive Function

```
(define (factorial n)
  (if (zero? n)
   1
   (* n (factorial (sub1 n)))))
```

A Recursive Function, as lambda

```
(define factorial
  (λ (n)
    (if (zero? n)
     1
     (* n (factorial (sub1 n))))))
```

A Recursive Function without recursion

```
(define factorial
  (λ (n)
    (if (zero? n)
     1
     (* n (THE-RECURSIVE-CALL (sub1 n))))))
```

Where does this come from?

Make it a parameter!

A Recursive Function without recursion

```
(define factorial
  (λ (THE-RECURSIVE-CALL)
    (λ (n)
      (if (zero? n)
       1
       (* n (THE-RECURSIVE-CALL (sub1 n)))))))
```

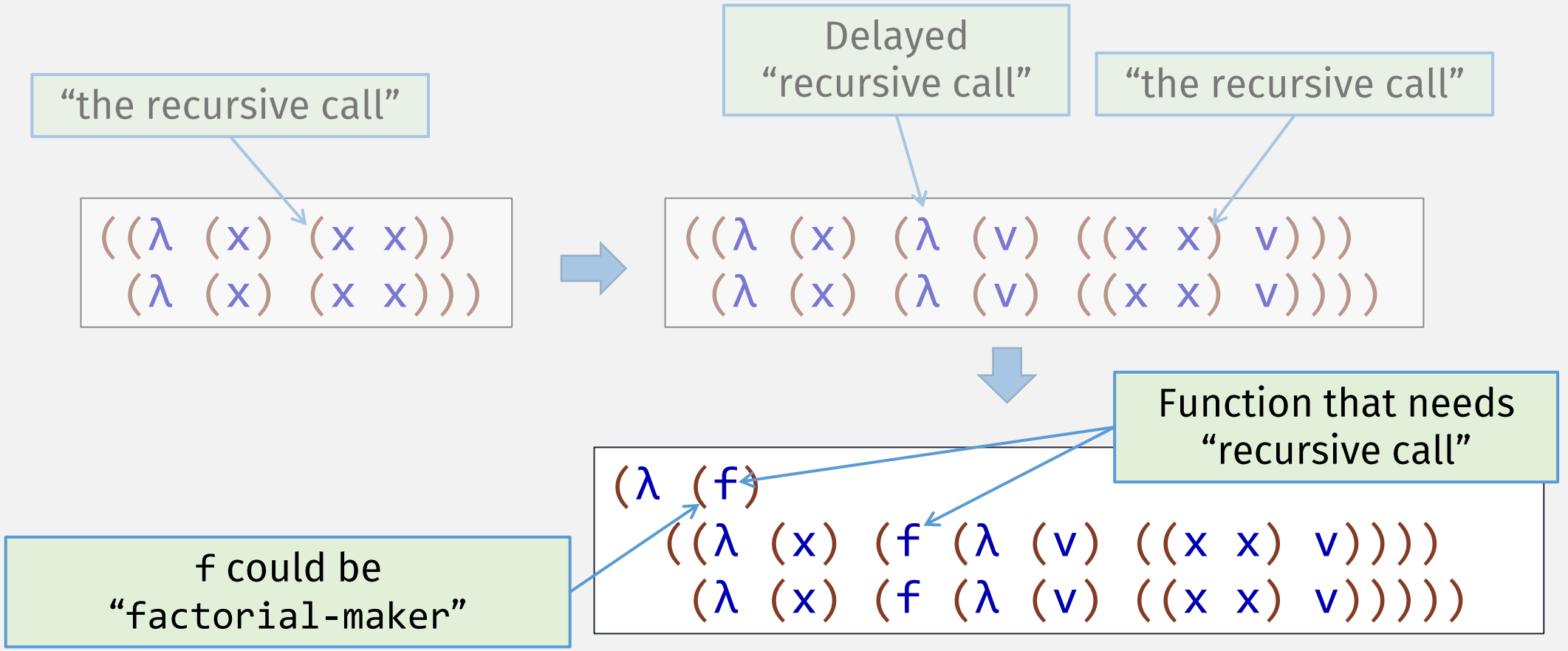
Make “the recursive call” a parameter

A Recursive Function without recursion

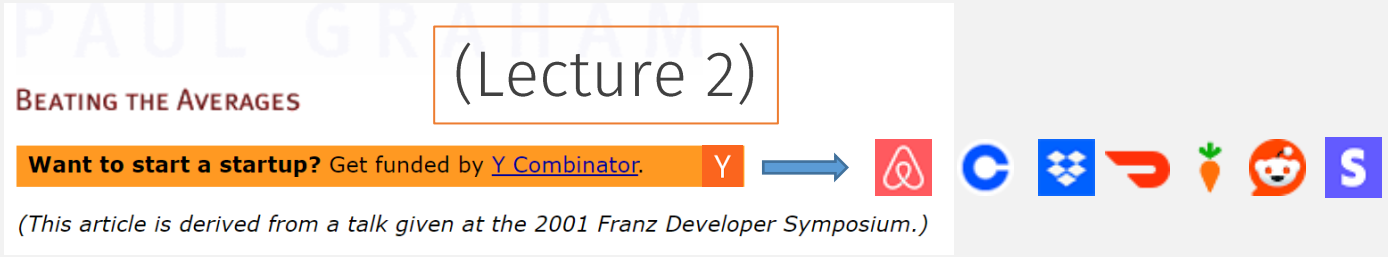
```
(define ~~factorial~~ factorial-maker
  (λ (THE-RECURSIVE-CALL)
    (λ (n)
      (if (zero? n)
       1
       (* n (THE-RECURSIVE-CALL (sub1 n)))))))
```

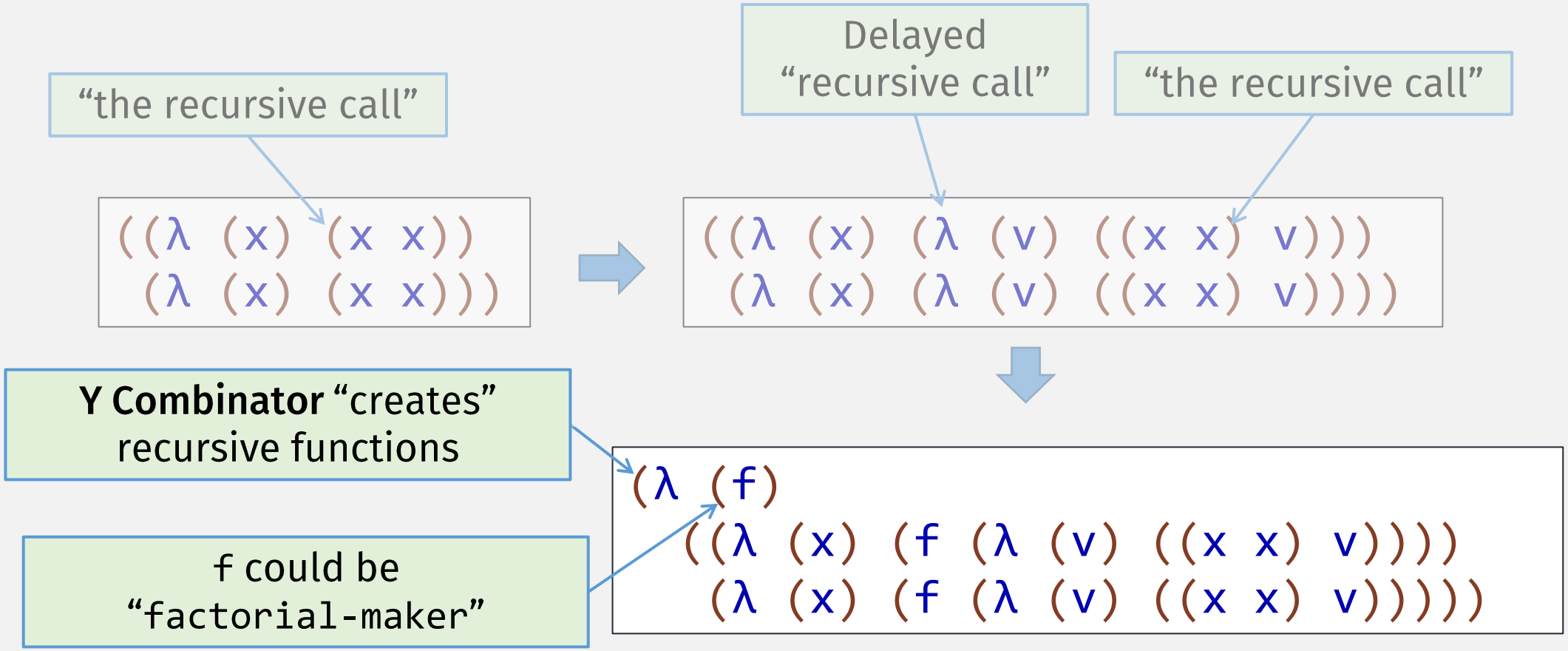
Make “the recursive call” a parameter

Delay “the recursive call”



Y Combinator





Code Demo