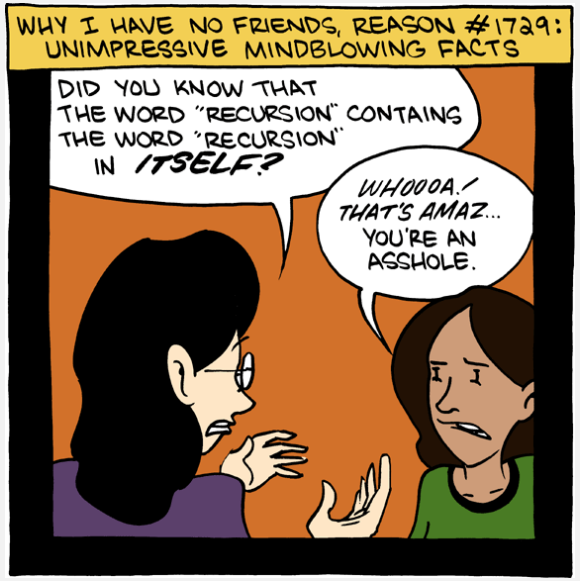
UMass Boston Computer Science

CS450 High Level Languages

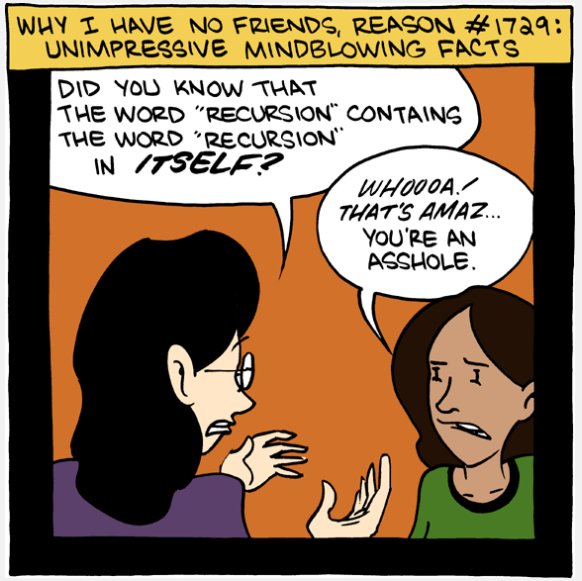
Recursion in the Lambda Calculus

Thursday, May 7, 2026



Logistics

- HW 12 out
  - ~~due: Thurs 5/7, 11am EST~~
- HW 13 out
  - due: Thurs 5/14, 11am EST



Previously

"high" level
(easier for humans
to understand,
"slower")

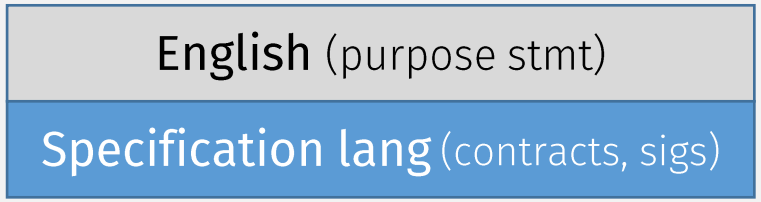
NOTE: This hierarchy is *approximate*

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

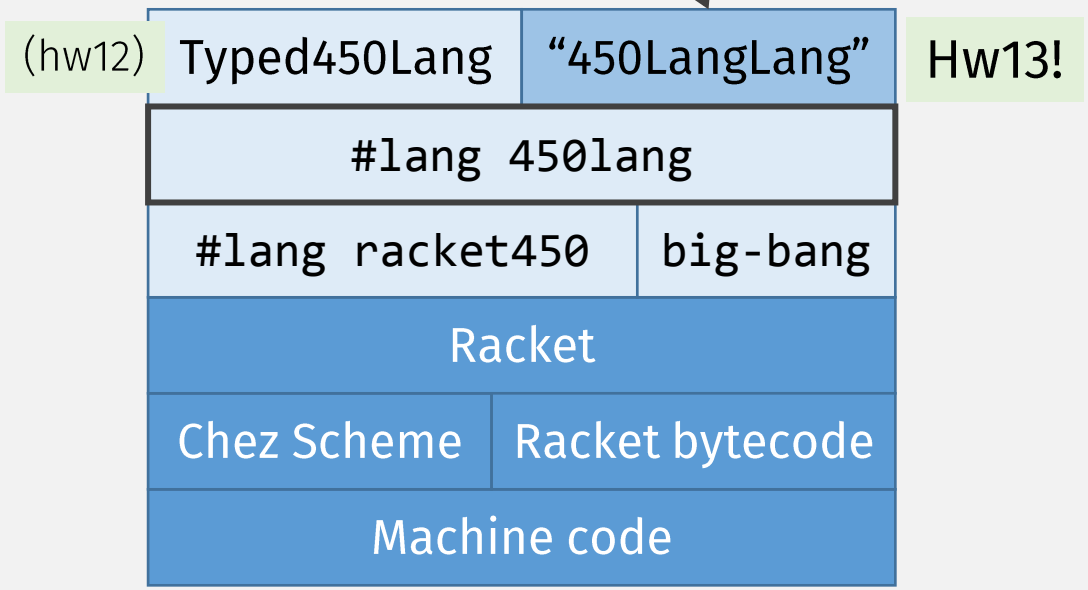
"low" level
(runs on cpu, hard to
read, more performant)

This Semester!

"high" level
(easier for humans to understand, "slower")

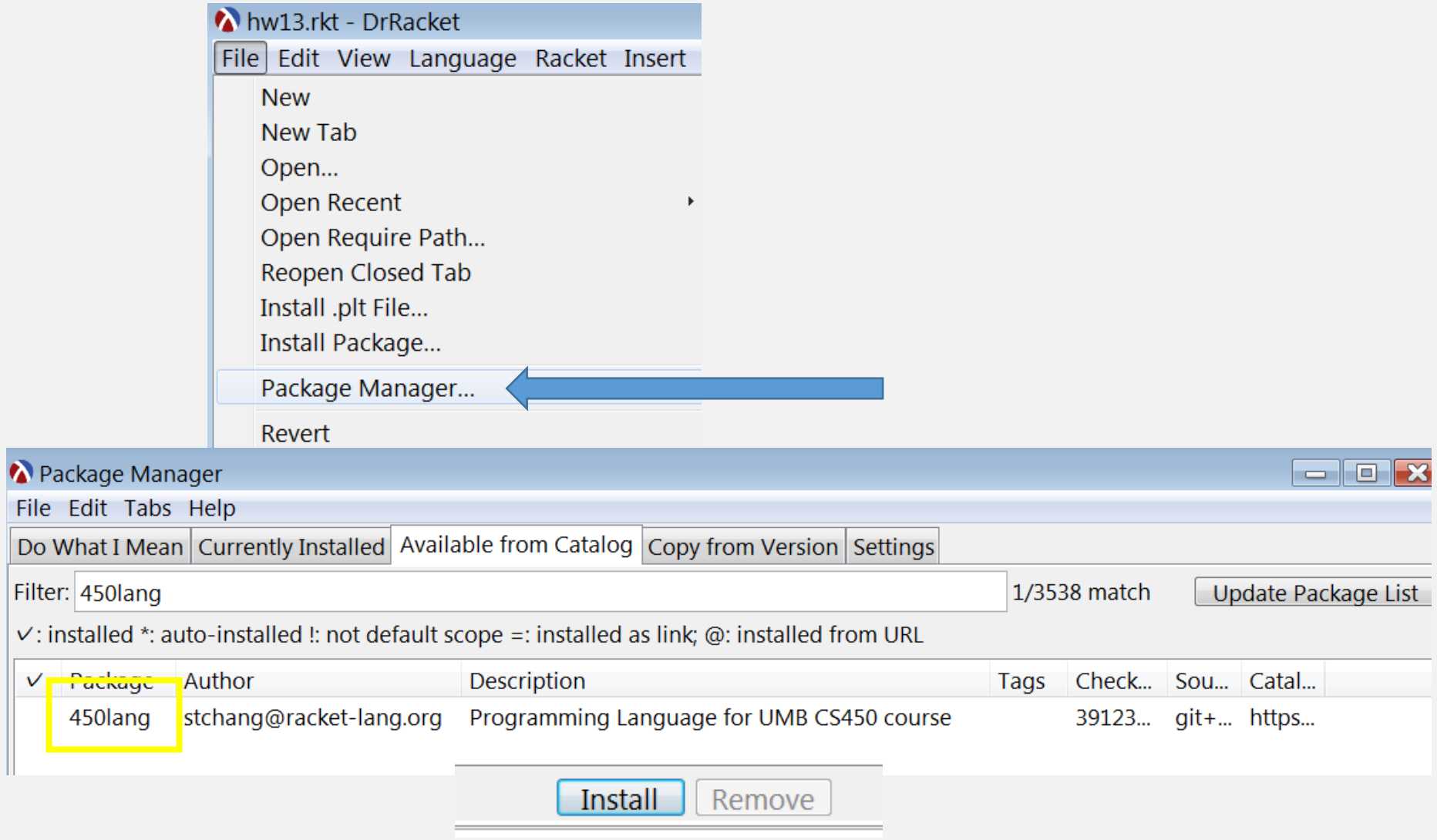


A self-hosted language is a programming language whose compiler or interpreter is written in the same language it compiles or interprets. For instance, a C compiler written in C, or a Rust compiler written in Rust, is self-hosted. This technique allows developers to use their own language to improve its compiler. Wikipedia +3



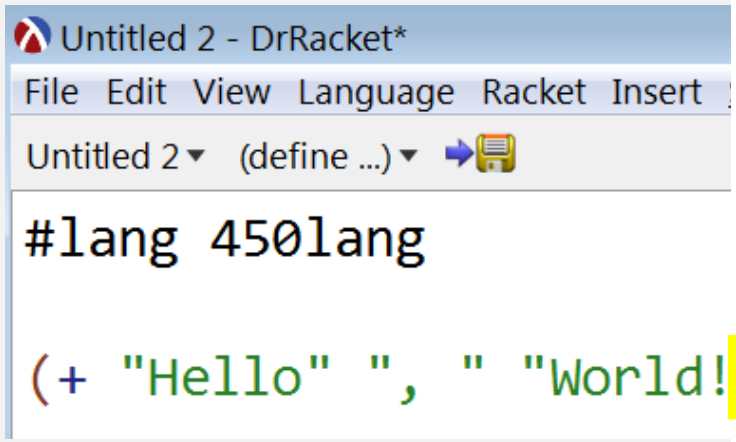
"low" level
(runs on cpu, hard to read, more performant)

Installing “450 Lang”



Using “450 Lang”

Read the **Programming Language Specification** linked from HW description!



“quotes” are **implicitly inserted b**

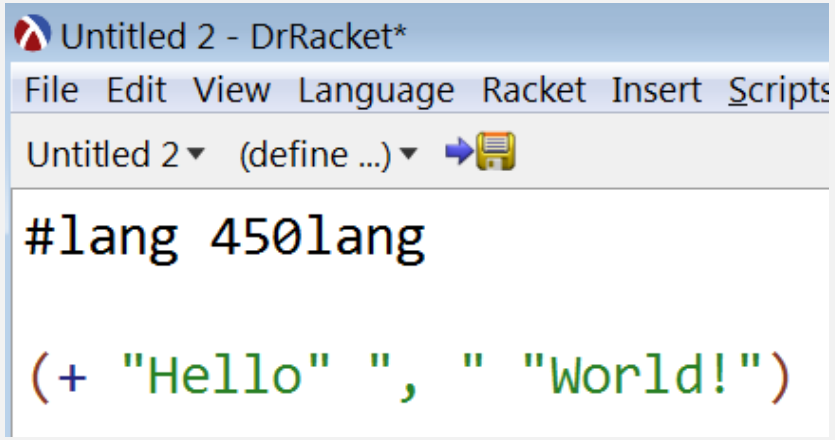
Programming language specification

From Wikipedia, the free encyclopedia

In computer programming, a **programming language specification** (or **standard** or **definition**) is a documentation artifact that defines a programming language so that users and implementors can agree on what programs in that language mean. Specifications are typically detailed and formal, and primarily used by implementors, with users referring to them in case of ambiguity; the C++ specification is frequently cited by users, for instance, due to the complexity. Related documentation includes a programming language reference, which is intended expressly for users, and a programming language rationale, which explains why the specification is written as it is; these are typically more informal than a specification.

A specification is more formal than user reference documentation!

Using “450 Lang”



“quotes” are implicitly inserted by the language

Taking requests …

Ask for additional primitives in `INIT-ENV`

Read the **Programming Language Specification** linked from HW description!

Added features:

- Lists
- More arith fns: `-`, `abs`
- Logical operations: ¬, ∧, ∨
- “top-level” `bind/rec` Like `define`
- `rackunit` equivalents

Not as “good” as `racket450/testing`

Design Recipe even more important now

Debugging will be **“impossible” – your only hope** is to **follow the Design Recipe**

DO NOT “save” writing tests until the end!!

(you’ve been warned)

Previously

The Function is Universal i.e., Lambda (λ) Calculus

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

No numbers???

- **Equivalent** in **“computational power”** to
  - Turing Machines
  - And … **your favorite programming language!**

How???

Previously

Church Numerals

```
;; A ChurchNum is a function with two arguments:
;; "fn" : a function to apply
;; "base" : a base ("zero") value to apply to
;;
;; Represents: a number where the given function is
;; applied that number of times to the given base
```

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

Function f applied zero times

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

Function f applied one times

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

Function f applied two times

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

Function f applied three times

Possible "instantiations":
- **base** = symbol "0"
- **f** = "add 1" operation

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 n**

Returns: **a ChurchNum …**

(we know “**n**” will apply **f** **n** times)

Total: n + 1

… that **adds an extra application** of **f** to ”**n**”

```
;; A ChurchNum is a function with two arguments:
;; “fn” : a function to apply
;; “base” : a base ("zero") value to apply to
```

```
(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 n m**

Returns: **a ChurchNum …**

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

Total: **n + m**

… that **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”

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$(A, B) = A$ ☑

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

```
(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 “If”

```
;; cif: ChurchBool Any Any -> Any
;; test p is either Church "true" or "false":
;; if p = true, result is first branch
;; if p = false, result is second branch
```

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

Returns: first arg

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

Returns: second arg

```
(define cif
  (lambda (test then else)
    (test then else)))
```

Church Pairs (Lists), i.e., data structures

```
;; 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))))
```

Input: ChurchPair

Output: “gets” the first item

```
(define csecond ; i.e., "rest"
  (lambda (cc)
    (cc (lambda (x y) y))))
```

Output: “gets” the second item

The Lambda (λ) Calculus

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

- “Language” can express:
  - Numbers
  - Booleans and conditionals
  - Lists
  - …
  - Recursion!?

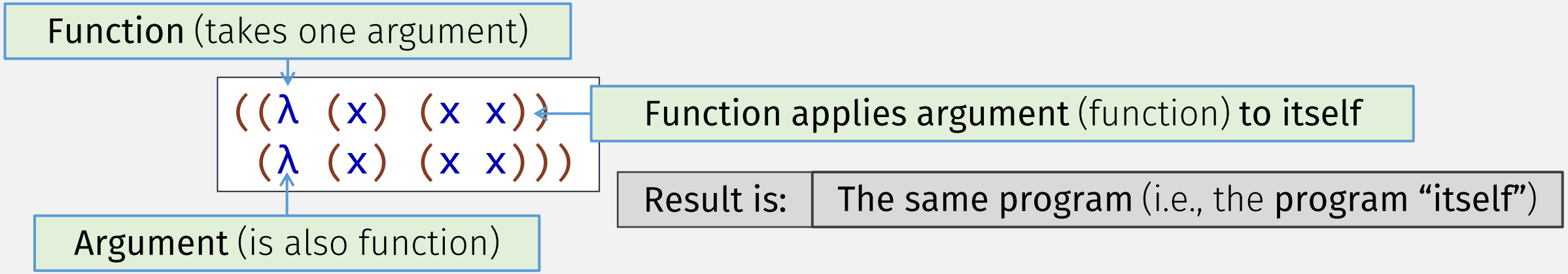
Recursion in the Lambda Calculus

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

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

Lambda Program that Knows "Itself"

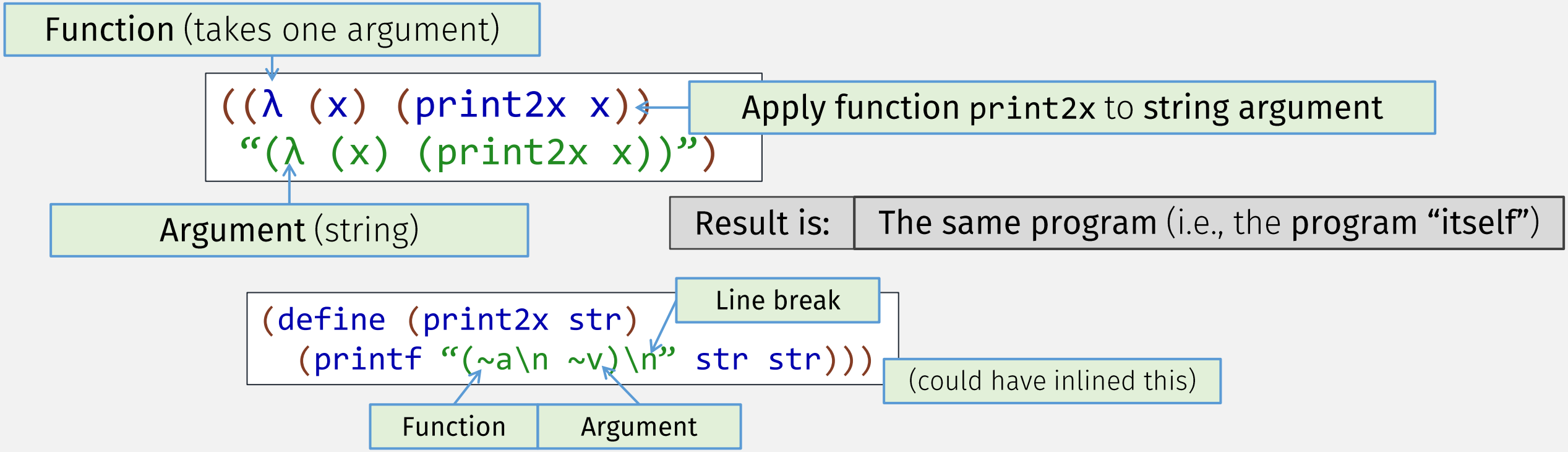
- **Program** that **runs "itself" repeatedly** (i.e., **it infinite loops**):



- Can we **do something else besides loop?**

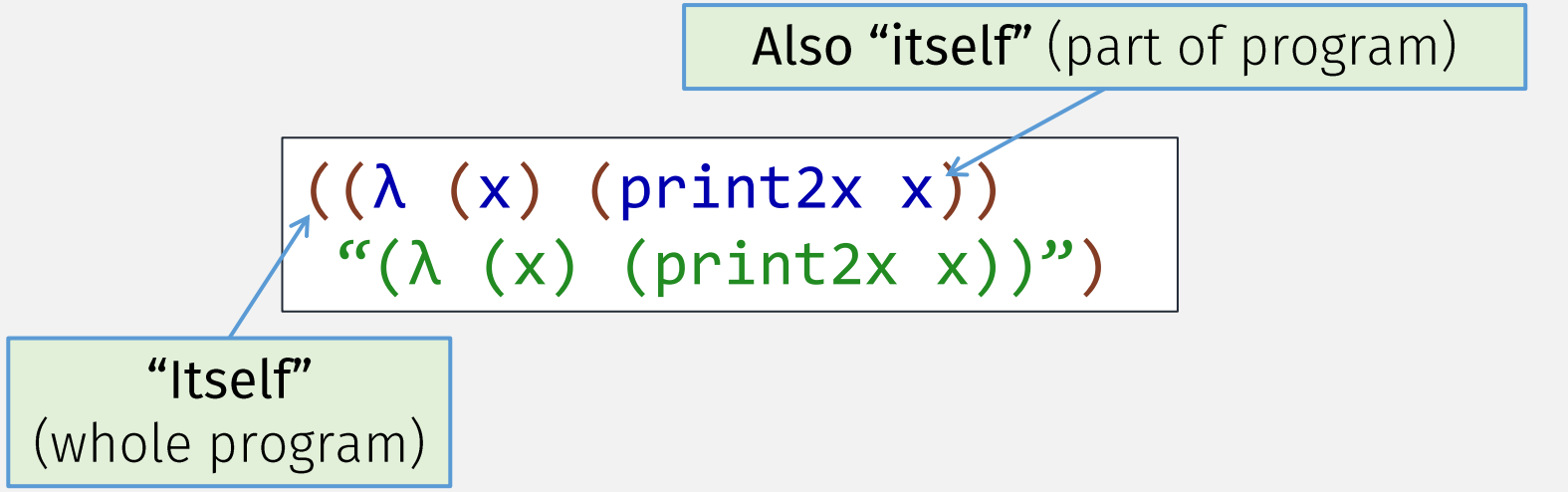
Lambda Program that Prints “Itself”

- **Program** that **prints “itself”:**



Lambda Program that Prints “Itself”

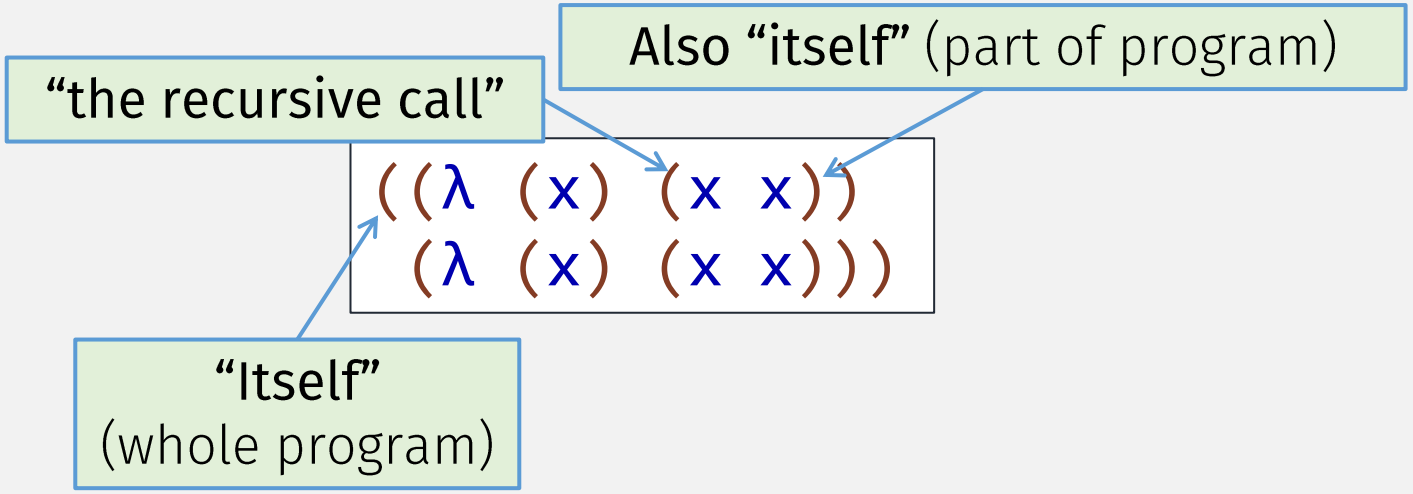
- **Program** that **prints “itself”:**



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

Lambda Program that Knows “Itself”

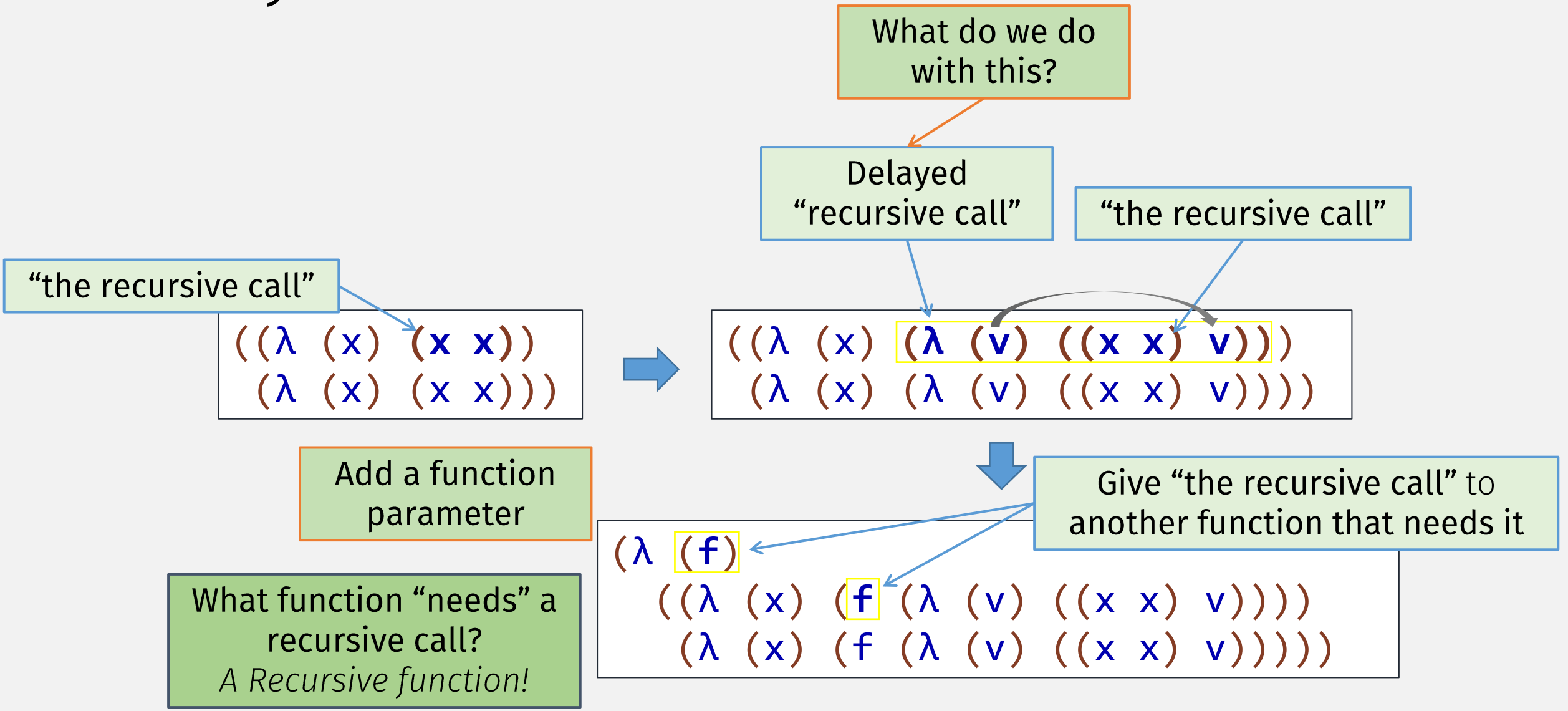
- **Program** that **runs “itself” repeatedly** (i.e., **it infinite loops**):



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)))))))
```

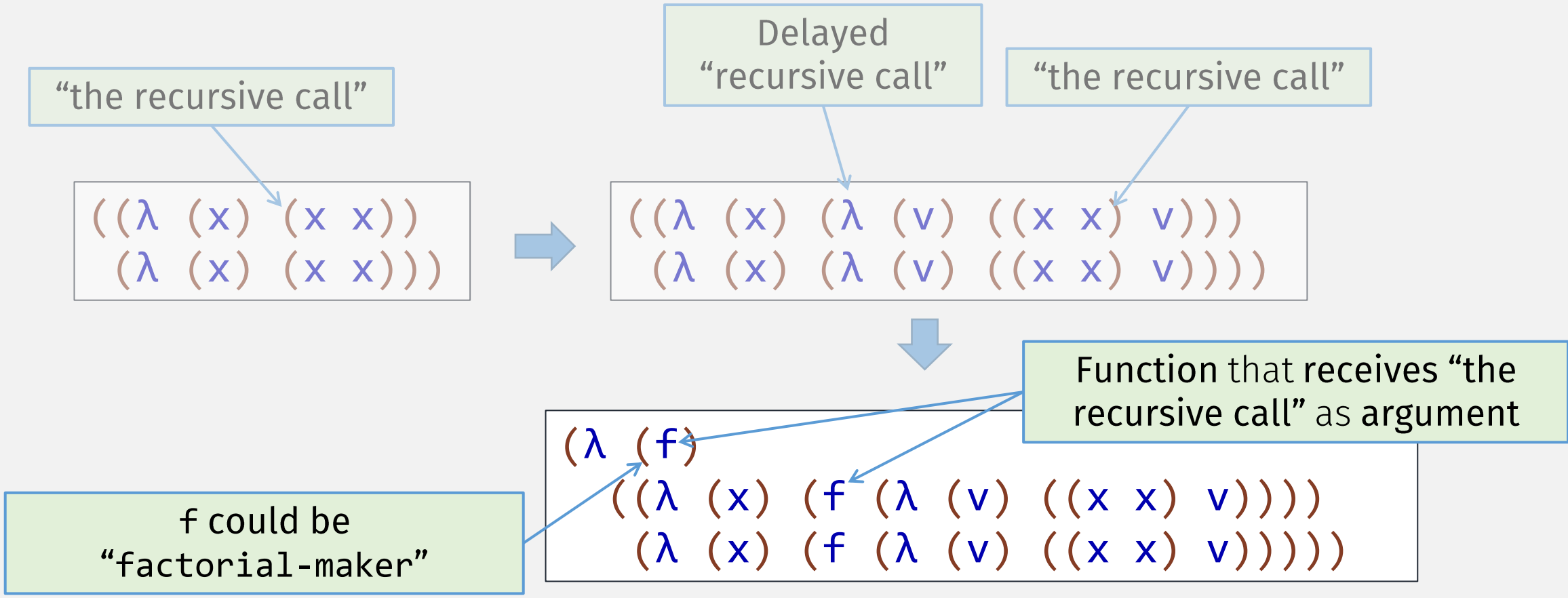
Make “the recursive call” a parameter

A Recursive Function without recursion

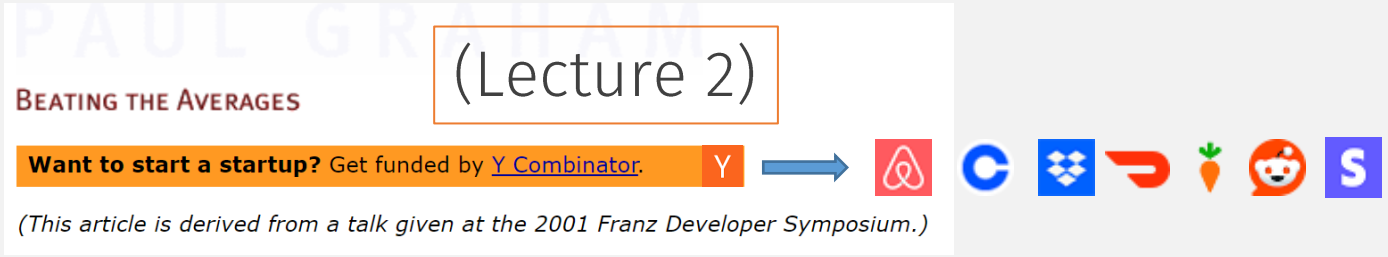
```
(define ~~factorial~~ factorial-maker
  (λ (THE-RECURSIVE-CALL)
    (λ (n)
      (if (zero? n)
        1
        (* n (THE-RECURSIVE-CALL (sub1 n)))))))
```

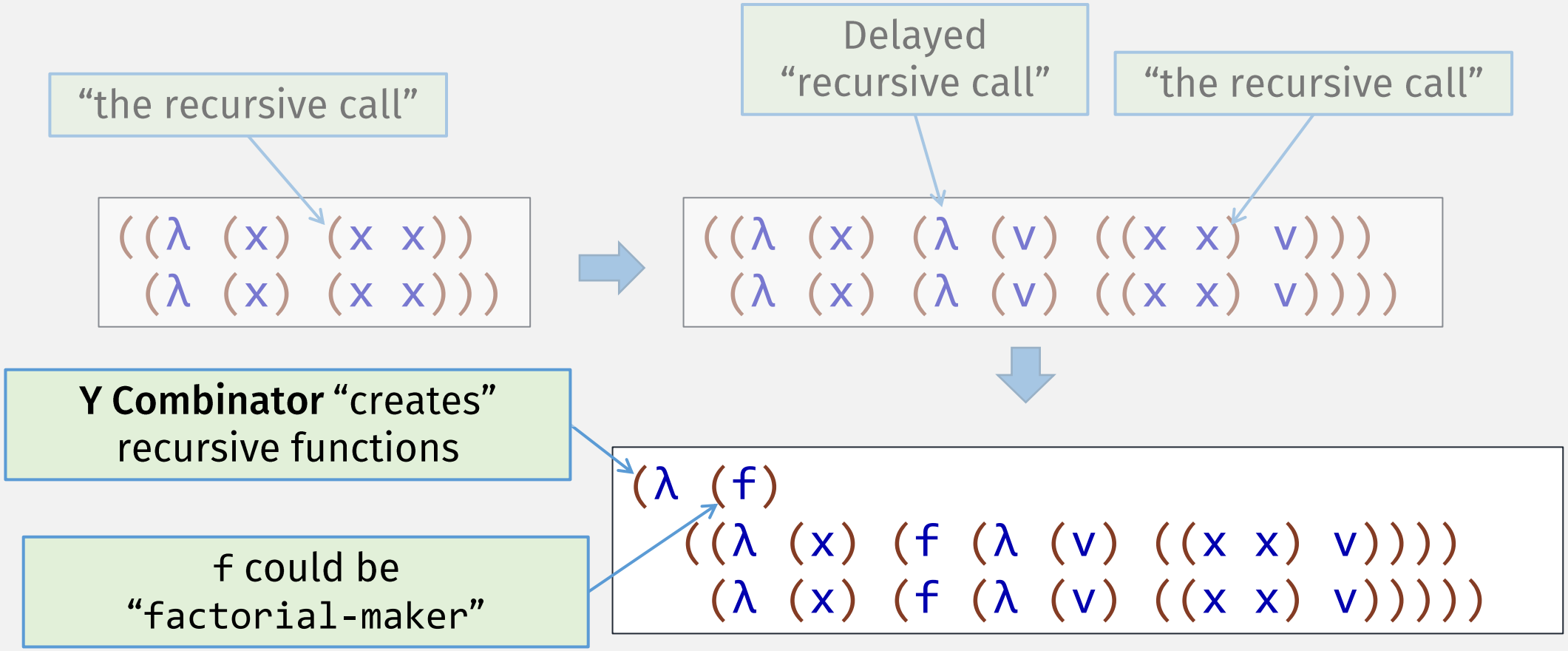
Need to **pass in "the recursive call"** to complete **`factorial`** function definition

Delay “the recursive call”



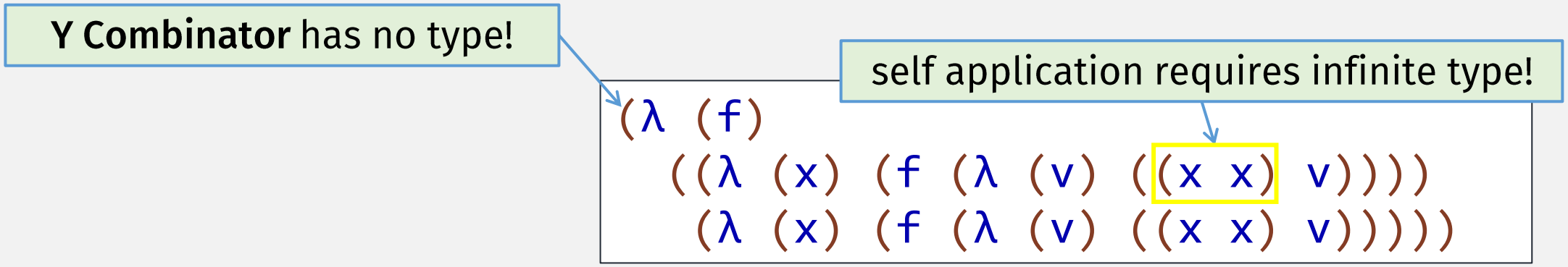
Y Combinator





Code Demo

Typed Y Combinator?



Typed Y Combinator?

Assume: fn **x** has **type A → B**

Then: arg **x** must have **type A** (to match fn arg type)

So **x** has type **A = A → B = A → B → B = …**

Y Combinator has no type!

Solution? need a **new (recursive) type!**

```
(λ (f)
  ((λ (x) (f (λ (v) ((x x) v))))
   (λ (x) (f (λ (v) ((x x) v))))))
```

Previously

The Function is Universal i.e., Lambda (λ) Calculus

untyped

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

- “Language” can express:
  - **Numbers**
  - **Booleans and conditionals**
  - **Lists**
  - **…**
  - **Recursion**

The Function is Universal i.e., Lambda (λ) Calculus

Typed

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

- “Language” can express:
  - Numbers
  - Booleans and conditionals
  - Lists
  - …
  - ~~Recursion~~