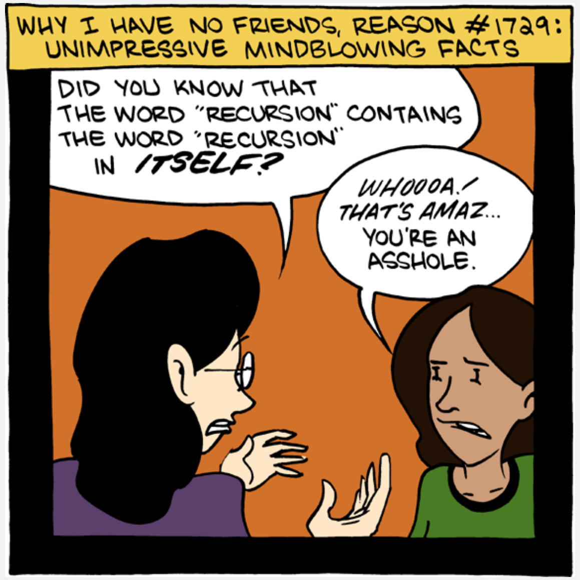
UMass Boston Computer Science

**CS450 High Level Languages** (section 2)

# Recursion in the Lambda Calculus

Wednesday, October 16, 2024



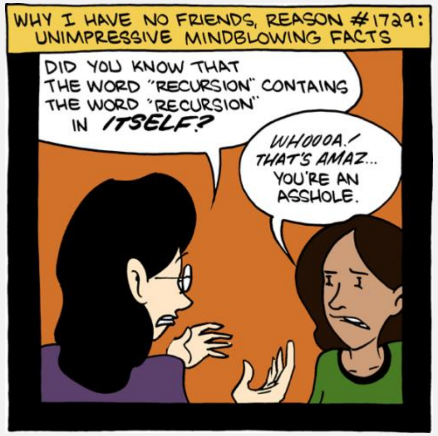
# Recursion in the Lambda Calculus

UMass Boston Computer Science

**CS450 High Level Languages** (section 2)

## Recursion in the Lambda Calculus

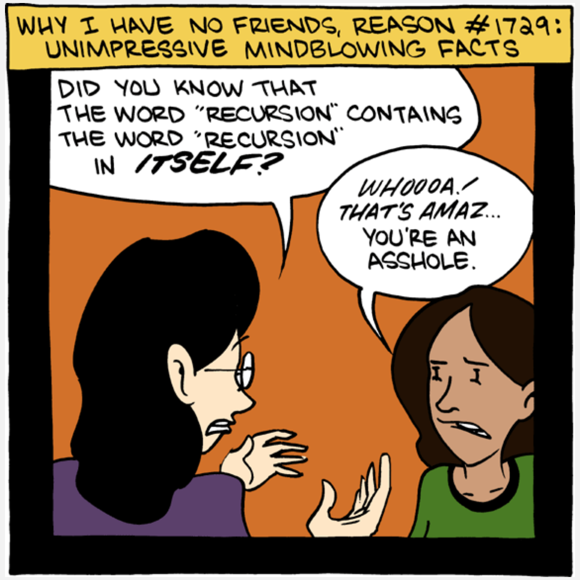
Wednesday, October 16, 2024



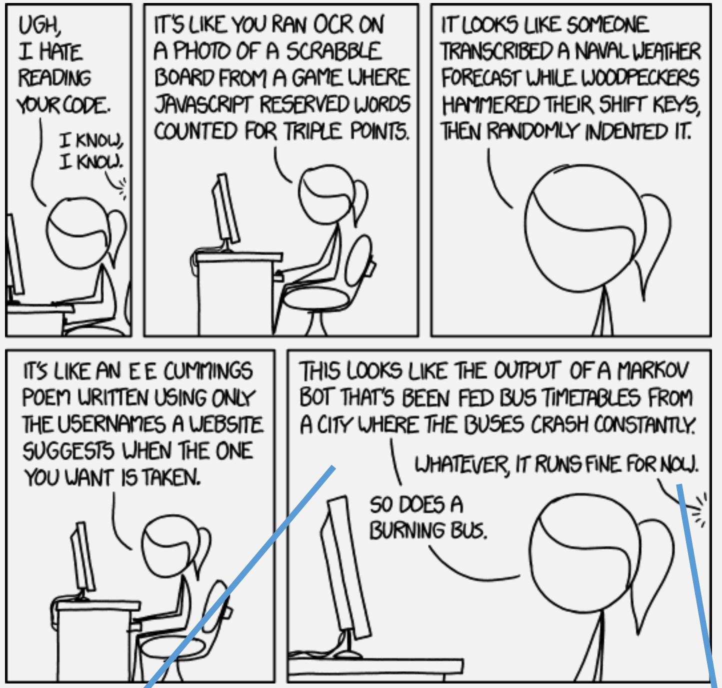
# *Logistics*

- HW 5 in
  - ~~due: Mon 10/14 12pm (noon) EST~~

- HW 6 out
  - due: Mon 10/21 12pm (noon) EST



# CS 450 so far …





<u>This class </u>teaches:

- a **high-level programming "process"**
- i.e., a **language-agnostic design recipe** for **creating clean, <u>readable programs</u>**

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

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

*From Lecture 1*

## NOTE: This hierarchy is *approximate*

"high" level (easier for humans to understand)

| 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 |

"low" level (runs on cpu)

**"declarative"** (Specification langs through Functional lang)

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

Core model: **Lambda Calculus**

**"imperative"** (JavaScript, Python through Machine code)

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

Core model: **Turing Machines**

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

"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 And($A$, $B$) = $B$

When $A$ = False,
then 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 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**):

**Function** (takes one argument)

```
((λ (x) (x x))
  (λ (x) (x x)))
```

**Function applies argument** (function) **to itself**

**Result is:** **The same program** (i.e., the **program "itself"**)

**Argument** (is also function)

- Can we do something else besides loop?

# Lambda Program that Prints "Itself"

- Program that prints "itself":

**Function** (takes one argument)

```
((λ (x) (print2x x))
 "(λ (x) (print2x x))")
```

**Apply function `print2x` to string argument**

**Argument** (string)

**Result is:** **The same program** (i.e., the **program "itself"**)

```
(define (print2x str)
  (printf "(~a\n ~v)\n" str str)))
```

Line break

(could have inlined this)

Function | Argument

# Lambda Program that Prints "Itself"

- Program that prints "itself":

Also "itself" (part of program)

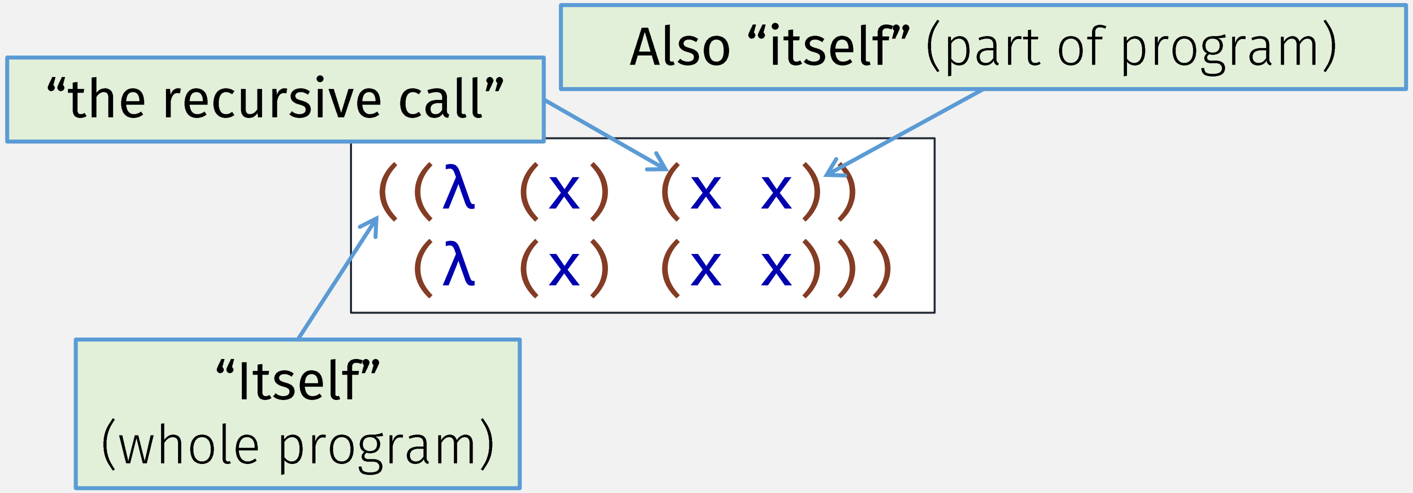
```
((λ (x) (print2x x))
 "(λ (x) (print2x x))")
```

"Itself"
(whole program)

- 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”

What do we do with this?

Delayed “recursive call”

“the recursive call”

“the recursive call”

```
((λ (x) (x x))
 (λ (x) (x x)))
```

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

Add a function parameter

Give “the recursive call” to another function that needs it

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

What function “needs” a recursive call?
*A Recursive function!*

# 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”

“the recursive call”

```
((λ (x) (x x))
 (λ (x) (x x)))
```

Delayed “recursive call”

“the recursive call”

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

Function that needs “recursive call”

f could be “fact-maker”

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

# Y Combinator

"the recursive call"

```
((λ (x) (x x))
 (λ (x) (x x)))
```

Delayed "recursive call"

"the recursive call"

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

**Y Combinator** "creates" recursive functions

f could be "fact-maker"

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

# Code Demo