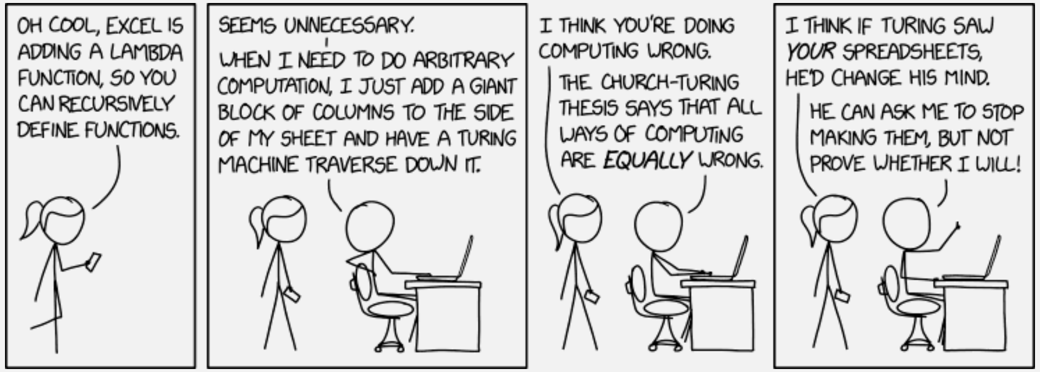
UMass Boston Computer Science

**CS450** **High Level Languages**

# Function "Arithmetic" and the Lambda Calculus

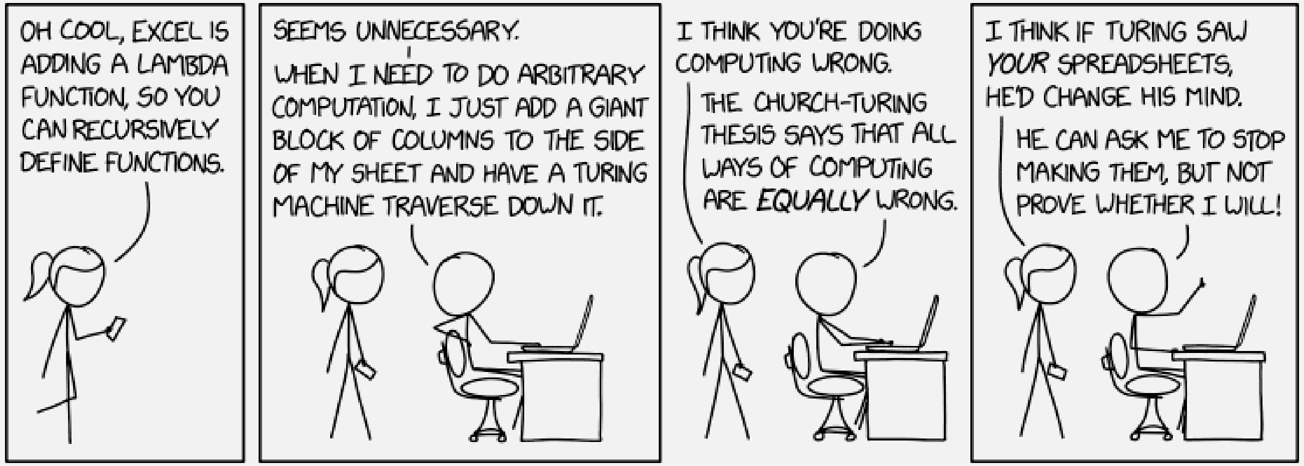
Thursday, March 13, 2025



# Logistics

- HW 6 out
  - Due: **Tues 3/25 11am EST** (2 weeks)

- Reminder: **Spring Break next week!**
  - **No lecture**



*Previously*

# Common list function #3: `filter`

```
;; filter: (X -> Boolean) Listof<X> -> Listof<X>
;; Returns a list containing elements of given list
;; for which the given predicate returns true
```

```
(define (filter pred? lst)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
             (cons (first lst) (filter (rest lst)))
             (filter (rest lst)))]))
```

# Common list function #3: `filter`

```
;; filter: (X -> Boolean) Listof<X> -> Listof<X>
;; Returns a list containing elements of g
;; for which the given predicate returns t
```

```
(define (filter pred? lst)
  (cond
   [(empty? lst) empty]
   [else (if (pred? (first lst))
             (cons (first lst) (filter (re
             (filter (rest lst)))]))
```

**lambda rules:**

- May skip **design recipe** steps, BUT
- **name**, **description**, and **signature** must be "obvious"
- **code** must be arithmetic only
- otherwise, **create standalone function** with **`define`**

```
;; smaller-than: Listof<Int> Int -> Listof<Int>
;; Returns a list containing elements of given list less than the given int
```

```
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst)
```

lambda creates an anonymous "inline" function (expression)

# Functions as Values

- In **high-level languages, functions** are **just another kind of data!**
  - **no different from other data** (e.g., numbers)
- They can **be passed around**, or **be the result of a function**

```
;; make< : Int -> (Int -> Bool)
;; makes a function that returns true
;; for values less than the given thresh value
```

```
(define (make< thresh)
  (lambda (x) (< x thresh)))
```

```
(define (smaller-than lst thresh)
  (filter (make< thresh)) lst)
```

- **`lambda`** is **just one possible way** to **"make" functions**
- We can also **do "arithmetic" with functions** to **compute new fns**

# Currying


- A **curried** function is **partially applied to some** (not all) **args**
- **Result is another function**

NOTE: First argument is <u>first arg</u> to fn

```
(curry < 4)
;; = a function that returns true when given a number greater than 4
```

```
(lambda (x) (< 4 x))
```

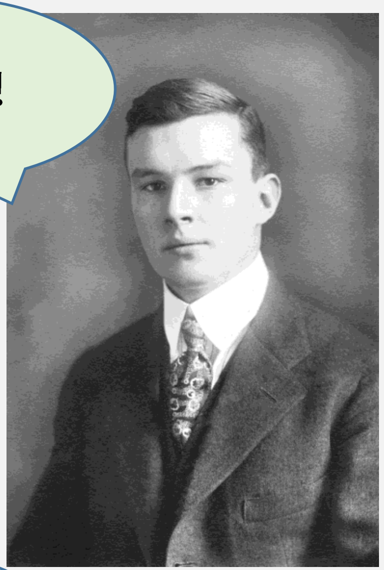
```
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst)
```

```
(define (smaller-than lst thresh)
  (filter (curry > thresh)) lst)
```

```
(define (smaller-than lst thresh)
  (filter (curryr < thresh)) lst)
```

NOTE: First argument is <u>last arg</u> to fn

# *History Lesson:* Haskell B. Curry



Go Cs!

Sorry Steph!

- Mathematician / Logician
- Born in Millis, MA, in year 1900

- "currying" functions is named after him
- and also, the "Haskell" (functional) programming language

- Invented "combinatory logic",
  i.e., a system of function "arithmetic"

# More Function Arithmetic

- **compose** combines multiple functions into one function
  - last one is applied first

```
(compose sqrt add1)
;; = a function that first applies add1 to its argument, then sqrt
```

```
(lambda (x) (sqrt (add1 x)))
```

```
((compose sqrt add1) 8)    ; = 3
```

# Composing Many Functions

- **compose** combines multiple functions into one function
  - last one is applied first

Step 1

```
(apply
 above
 (build-list
  5
  (compose (curryr square "solid" "blue")
           (curry * 20)
           add1)))
```

*Previously*

# Fold "dual": build-list

```
(build-list n proc) → list?                                procedure
  n : exact-nonnegative-integer?
  proc : (exact-nonnegative-integer? . -> . any)
```

Creates a list of *n* elements by applying *proc* to the integers from `0` to `(sub1 n)` in order. If *lst* is the resulting list, then `(list-ref lst i)` is the value produced by `(proc i)`.

Examples:

```
> (build-list 10 values)
'(0 1 2 3 4 5 6 7 8 9)
> (build-list 5 (lambda (x) (* x x)))
'(0 1 4 9 16)
```

```
(build-list 4 add1)
;; = (map add1 (list 0 1 2 3))
            ;; = (list 1 2 3 4)
```

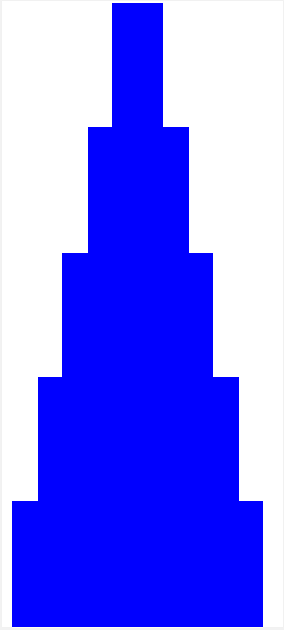
# Composing Many Functions

- **compose** combines multiple functions into one function
  - last one is applied first

```
6(apply
5 above
1 (build-list              ; = (list 0 1 2 3 4)
   5          map
   (compose 4(curryr square "solid" "blue")
            3(curry * 20)  ; = (list 20 40 60 80 100)
            2 add1)))      ; = (list 1 2 3 4 5)
```

```
; = (above (square 20 “solid” “blue”)
           (square 40 “solid” “blue”)
           (square 60 “solid” “blue”)
           (square 80 “solid” “blue”)
           (square 100 “solid” “blue”))
```

```
; = (list (square 20 “solid” “blue”)
          (square 40 “solid” “blue”)
          (square 60 “solid” “blue”)
          (square 80 “solid” “blue”)
          (square 100 “solid” “blue”))
```



# The Lambda (λ) Calculus

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

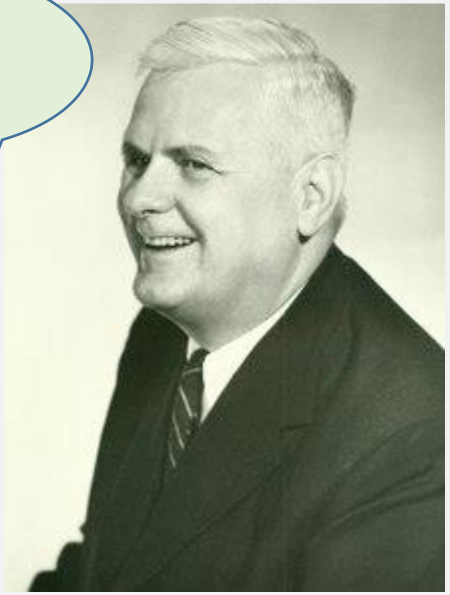
No numbers???

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

How???

# *History Lesson:* Alonzo Church

Go CS!



- **Mathematician, logician, computer scientist**
- **Invented the Lambda Calculus** ← No numbers??? How to do add??
- And (half of) **Church-Turing Thesis**
  - <u>**Any "computable" function**</u> has:
    - an **equivalent Turing Machine**, and
    - an **equivalent Lambda Calculus program**
  - so, **a Turing Machine = a lambda**

How???

# 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**”

```
(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)))))
```

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

# 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)))))
```

# Code Demo 1 - 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
```

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

```
(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

# 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"

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

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

The truth table of $A \land B$:

| $A$ | $B$ | $A \land B$ |
|---|---|---|
| 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 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 "Or"

```
;; cor: ChurchBool ChurchBool-> ChurchBool
;; "or" the given ChurchBools together
```

```
(define cor
  (lambda (A B)
    (A A B)))
```

| $A$ | $B$ | $A \vee B$ |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

When $A$ = True,
**want:** $\mathrm{Or}(A, B) = A$ ☑

When $A$ = False,
**want:** $\mathrm{Or}(A, B) = B$ ☑

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

(Returns first arg)

```
;; if A = ctrue
;; then (A A B) = A
;; want (cor A B) = A
```

☑

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

(Returns second arg)

```
;; if A = cfalse
;; then (A A B) = B
;; want (cor A B) = B
```

☑

# Church "If"

```
;; cif: ChurchBool Any Any -> Any
;; Church "if" same as 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)))
```

# Code Demo 2 – Church Booleans

# 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))))
```

Input ChuchPair

“Gets” the first item

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

“Gets” the second item

# Code Demo 3 – Church Pairs

# The Lambda Calculus

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

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

# In-class exercise: Self-printing Program

Write a program that prints “itself”:

```
(define (print-myself ???)
  (printf “print-myself ??? ~a ???\n” ???)))
```

“Format” string

argument