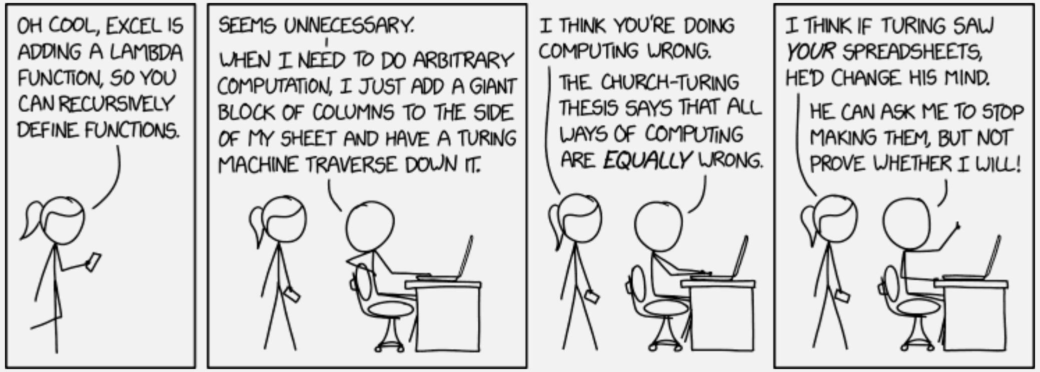
UMass Boston Computer Science

CS450 High Level Languages (section 2)

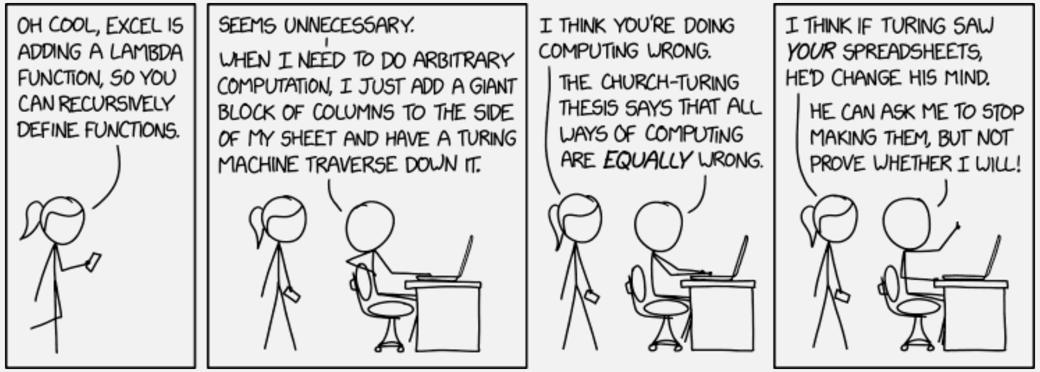
Function "Arithmetic" and the Lambda Calculus

Wednesday, October 18, 2023



Logistics

- HW 4 out
  - due: Sun 10/22 11:59 pm EST



Last Time

Common List Function #1: **map**

```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Produces a list resulting from applying
;; a given fn to each element of a given lst
```

```
(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map (rest lst)))]))
```

function "**application**" (in high-level languges) = function "call" (in imperative languages)

```
(map proc lst ...+) → list?        procedure
  proc : procedure?
  lst : list?
```

Applies *proc* to the elements of the *lsts* from the first elements to the last. The *proc* argument must accept the same number of arguments as the number of supplied *lsts*, and all *lsts* must have the same number of elements. The result is a list containing each result of *proc* in order.

Examples:

```
> (map (lambda (number1 number2)
         (+ number1 number2))
       '(1 2 3 4)
       '(10 100 1000 10000))
'(11 102 1003 10004)
```

RACKET's map takes multiple lists

`map` in other high-level languages

Array.prototype.map()

The `map()` method of `Array` instances creates a new array populated with the results of calling a provided function on every element in the calling array.

JavaScript Demo: Array.map()

```javascript
const array1 = [1, 4, 9, 16];

// Pass a function to map
const map1 = array1.map((x) => x * 2);

console.log(map1);
// Expected output: Array [2, 8, 18, 32]
```

Lambda
(“arrow function expression”)

Python3

```python
# Add two lists using map and lambda

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

lambda

Last Time

Common List Function #2: **foldl** / **foldr**

```
;; foldr: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, last-element-first
```

```
(1 + (2 + (3 + 0)))
(1 - (2 - (3 - 0)))
```

```
(define (foldr fn initial lst)
  (cond
   [(empty? lst) initial]
   [else (fn (first lst) (foldr fn initial (rest lst)))]))
```

```
;; foldl: (X Y -> Y) Y Listof<X> -> Y
;; Computes a single value from given list, determined by given fn and initial val.
;; fn is applied to each list element, first-element-first
```

```
(((1 + 0) + 2) + 3)
(((1 - 0) - 2) - 3)
```

```
(define (foldl fn result-so-far lst)
  (cond
   [(empty? lst) result-so-far]
   [else (foldl fn (fn (first lst) result-so-far) (rest lst))]))
```

Last Time

`fold` (`reduce`) in other high-level languages

JavaScript Demo: Array.reduce()

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce((resultSoFar, x) => resultSoFar + x, initial);

console.log(sumWithInitial);
// Expected output: 10

```

“list”

lambda

“initial”

JavaScript Demo: Array.reduceRight()

```
const array1 = [
  [0, 1],
  [2, 3],
  [4, 5],
];

const result = array1.reduceRight((resultSoFar, x) => resultSoFar.concat(x));

console.log(result);
// Expected output: Array [4, 5, 2, 3, 0, 1]

```

“initial” optional?

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)
```

Fold “alternative”: **apply** (and variable-arity fns)

```
(foldl + 0 (list 1 2 3 4)) ; = (+ (+ (+ (+ 1 0) 2) 3) 4)) = 10
```

```
(apply + (list 1 2 3 4)) ; = (+ 1 2 3 4) = 10
```

```
(apply string-append (list “a” “b” “cd”)) ; = “abcd”
```

- apply applies its function arg to the contents of its `list` arg
- function arg to apply must accept:
  # of arguments = <u>length</u> of `list` arg

Common list function #3: `filter`

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

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

`filter` in other high-level languages

JavaScript Demo: Array.filter()

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter((word) => word.length > 6);

console.log(result);
// Expected output: Array ["exuberant", "destruction", "present"]
```

Common list function #3: `filter`

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

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

lambda rules:

- Can skip the **design recipe** steps, BUT
- **name**, **description**, and **signature** must be "obvious"
- **code** is arithmetic only
- otherwise, **create standalone function** `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 no different from other values** (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 way to "make" functions
- We can also do "arithmetic" with functions

Currying

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

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

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

⇓

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

History Lesson: Haskell B. Curry

- 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"

Currying

```
(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 gets curried first

Composing Functions

- 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
```

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

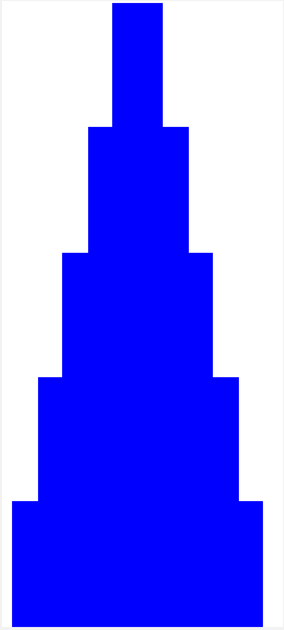
Composing Functions

- 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
    (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

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

History Lesson: Alonzo Church

- Mathematician, logician, computer scientist
- Invented the lambda calculus
- And (half of) Church-Turing Thesis
  - Any function that can be "computed" has an equivalent Turing Machine
  - And an equivalent program in the lambda calculus
  - so, a Turing Machine = a lambda

Church Numerals

```
;; A ChurchNum is a function with two arguments:
;; “fn” : 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))
```

Function applied <u>zero</u> times

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

Function applied <u>one</u> times

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

Function applied <u>two</u> times

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

Function applied <u>three</u> 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 a ChurchNum …

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

… that 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 ...

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

... that adds "m" 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)))))
```

Code Demo 1

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	$A \lor B$
True	True	True
True	False	True
False	True	True
False	False	False

When A = True, want Or(A, B) = A ☑

When A = False, want 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
```

☑

Code Demo 2

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

Code Demo 3

The Lambda Calculus

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

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

Check-In Quiz 10/18

on gradescope

(due 1 minute before midnight)