CS 360
Programming Languages
Day 8

- Recap everything we missed
    - First-class function
    - Lambdas
    - Higher-order functions
- Practice:
    - Map
    - Filter
    - Foldr

Recap

- First-class Functions

- Lambda expressions
  - **Warm-up:** write a lambda expression which takes in two points, a and b, and returns the distance between them

- Higher-order Functions
  - **Warm-up:** write a function **pairs-to-nums** which takes in a function **f** and a list of pairs **pairs** and applies **f** to each pair to construct a list of numbers.

Map

- **pairs-to-nums** is an example of a higher-order function which *maps* pairs to numbers according to some rule (or function) **f**

- This behavior is a common pattern:
  - Map numbers to their squares
  - Map usernames to email addresses
  - Map strings to their lengths

```
(define (map f lst)
  (if (null? lst) `()
    (cons (f (car a)) (map f (cdr lst)))))
```

Filter

- Similarly, it’s common to **filter** out certain elements

- Examples:
  - Keep all positive numbers
  - Keep all “valid” strings
  - Keep all courses that will make

```
(define (filter f a)
  (cond ((null? a) '())
        ((f (car a)) (cons (car a)
                           (filter f (cdr a))))
        (else (filter f (cdr a)))))
```

What else can we do?

```
(define (length lst)
  (if (null? lst) 0
    (+ 1 (length (cdr lst)))))
```

```
(define (sum-list lst)
  (if (null? lst) 0
    (+ (car lst) (sum-list (cdr lst)))))
```

```
(define (map func lst)
  (if (null? lst) '()
    (cons (func (car lst)) (map func (cdr lst)))))
```

All of these have:

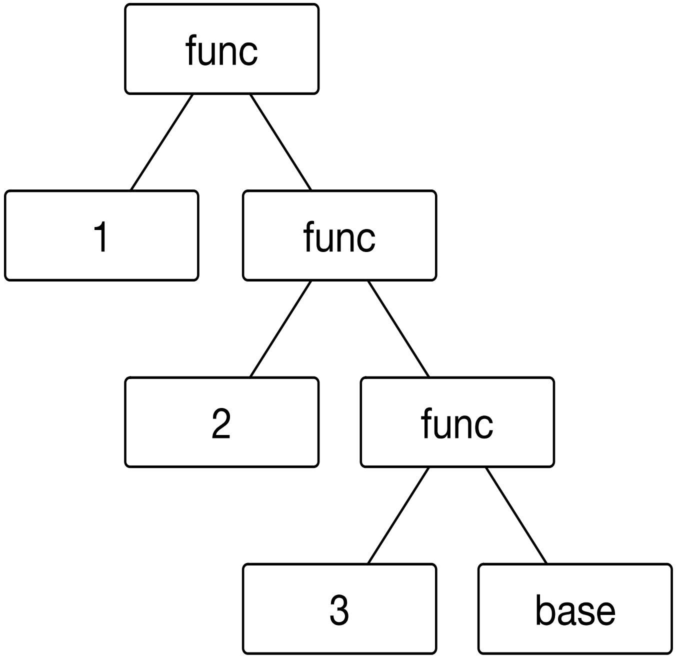
- A base case when the list is null (orange)
- A return value for the base case (green)
- A recursive case where we combine (red) something with the car of the list (purple) with a recursive call on the cdr (blue)

One function to rule them all

```
(define (foldr func base lst)
  (if (null? lst) base
    (func (car lst)
              (foldr func base (cdr lst)))))
```

`(foldr func base lst)`

Say `lst = '(1 2 3)`



- Foldr applies **`func`** repeatedly to pairs of items, starting from the right end of the list.
- The first two items are the last item in the list and the base element.
- The function must be a function of two items.

  `(f 1 (f 2 (f 3 base)))`
- In general, for `lst = (x1 x2 … xn)`
- **`(f x1 (f x2 (f x3 (f … (f xn base)))…)`**

Examples

- Identify the **func** and the **base** and try to describe the behavior:

- `(foldr + 0 lst)`

- `(foldr (lambda (item acc) (+ 1 acc)) 0 lst)`

Examples with foldr

These are useful and do not use “private data”

```
(define (f1 lst) (foldr + 0 lst))
(define (f2 lst)
  (foldr (lambda (x acc) (and (>= x 0) acc)) #t lst))
```

These are useful and do use “private data”

```
(define (f3 lo hi lst)
  (foldr
    (lambda (x acc)
      (+ (if (and (>= x lo) (<= x hi)) 1 0) acc)) 0 lst))

(define (f4 g lst)
  (foldr (lambda (x acc) (and (g x) acc)) #t lst))
```

You try:

- Write reverse using foldr.

- Write max using foldr.
    - Try to make it so the "base" argument to foldr is not a huge negative number. (write it this way first if it's easier, then change it)

- Write map using foldr.

- Write filter using foldr.