

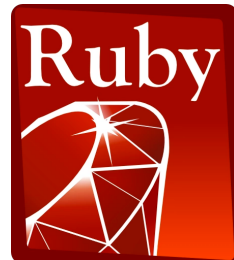
# CS 360

## Programming Languages

### Day 4



Swift



# *Today*

- Learn the common recursive paradigms that you will see in lots of Racket functions.
- Practice writing functions.

## *Example list functions*

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

```
(define (countdown num)
  (if (= num 0)
      '()
      (cons num (countdown (- num 1)))))
```

# *Recursion again*

Functions that process lists are usually recursive.

- Only way to “get to all the elements”
- What should the answer be for the empty list?
  - Usually, this is your base case.
- What should the answer be for a non-empty list?
  - Typically a combination of doing something with the **car** of the list and a recursive call on the **cdr** of the list.

Similarly, functions that produce lists of potentially any size will be recursive.

- You create a list out of smaller lists (with **cons**, **list**, or **append**).

# *The cond expression*

We have two "if-then-else" expressions in Racket:

- `(if test e1 e2)`
  - evaluates to `e1` if `test` is `#t`, otherwise evaluates to `e2`.
- `(cond (test1 e1)`  
    `(test2 e2)`  
    `...`  
    `(#t en))`
  - evaluates to `e1` if `test1` is `#t`
  - evaluates to `e2` if `test2` is `#t`
  - (etc)
  - evaluates to `en` if all prior tests are `#f`
  - The last `#t` clause is optional, but is useful as an "else".

# *Processing nested lists*

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

```
(define (length-nested lst)
  (cond ((null? lst) 0)
        ((list? (car lst))
         (+ (length-nested (car lst))
            (length-nested (cdr lst))))
        (#t (+ 1 (length-nested (cdr lst)))))
```

# *Other useful functions and reminders*

- `(and e1 e2...)`
- `(or e1 e2...)`
- `(not expr)`
  - e.g., `(not (= a b))`
- `(remainder x y)`
  - returns remainder of `x` divided by `y`
- Remember the differences between `cons`, `list`, and `append`:
- `(cons item lst)`
  - makes a new list with `item` as the first element, and the items in `lst` as the rest of the list.
- `(list a b c...)`
  - makes a new list of `(a b c...)`
- `(append lst1 lst2...)`
  - makes a new list of the items inside of `lst1`, then the items inside of `lst2...`