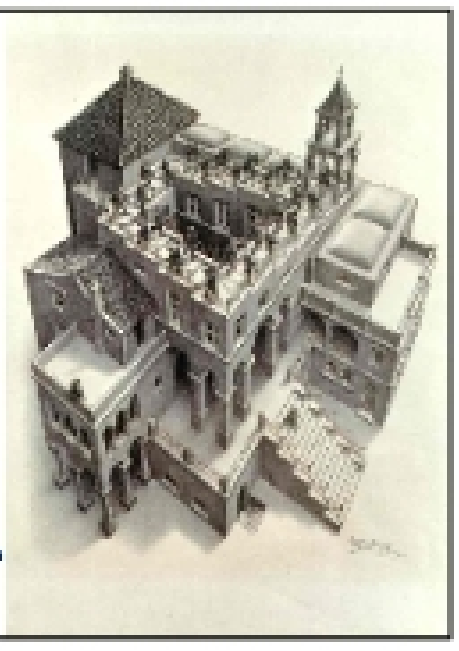
Lecture 5: Recursing on Lists



CS150: Computer Science
University of Virginia
Computer Science

Menu

- Implementing cons, car, cdr
- PS1
- List Recap
- List Recursion

Everyone who submitted a registration survey should have received an email yesterday with your PS2 partner. If you didn't come talk to me after class.

Implementing cons, car and cdr

```
(define (cons a b)
  (lambda (w) (if (w) a b)))

(define (car pair) (pair #t)
(define (cdr pair) (pair #f)
```

Scheme provides primitive implementations for cons, car, and cdr. But, we could define them ourselves.

CS150 PS Grading Scale

★Gold Star – Excellent Work. You got everything I wanted on this PS. (No Gold Stars on PS1)

★Green Star – Better than good work

★Blue Star – Good Work. You got most things on this PS, but some answers could be better.

★Silver Star – Some problems. Make sure you understand the solutions on today's slides.

PS1 Average: ★

No upper limit

★★ - Double Gold Star: exceptional work! Better than I expected anyone would do.

★★★- Triple Gold Star: Better than I thought possible (moviemosaic for PS1)

★★★★- Quadruple Gold Star: You have broken important new ground in CS which should be published in a major journal!

★★★★★- Quintuple Gold Star: You deserve to win a Turing Award! (a fast, general way to make the best non-repeating photomosaic on PS1, or a proof that it is impossible)

Question 2

- Without Evaluation Rules, Question 2 was "guesswork"
- Now you know the Evaluation Rules, you can answer Question 2 without any guessing!

2d

(100 + 100)

Evaluation Rule 3. Application.

a. Evaluate all the subexpressions

100 <primitive:+> 100

b. Apply the value of the first subexpression to the values of all the other subexpressions

Error: 100 is not a procedure, we only have apply rules for procedures!

2h

(if (not "cookies") "eat" "starve")

Evaluation Rule 4-if. Evaluate $Expression_0$. If it evaluates to **#f**, the value of the if expression is the value of $Expression_2$. Otherwise, the value of the if expression is the value of $Expression_1$.

Evaluate (not "cookies")

Evaluate (not "cookies")

Evaluation Rule 3. Application.

a. Evaluate all the subexpressions

<primitive:not> "cookies"

The quotes really matter here!

Without them what would cookies evaluate to?

b. Apply the value of the first subexpression to the values of all the other subexpressions

(not *v*) evaluates to #t if *v* is #f, otherwise it evaluates to #f. (SICP, p. 19)

So, (not "cookies") evaluates to #f

Defining not

(not *v*) evaluates to **#t** if *v* is **#f**, otherwise it evaluates to **#f**.

(SICP, p. 19)

(define (not v) (if v #f #t)

2h

(if (not "cookies") "eat" "starve")

Evaluation Rule 4-if. Evaluate $Expression_0$. If it evaluates to **#f**, the value of the if expression is the value of $Expression_1$. Otherwise, the value of the if expression is the value of $Expression_2$.

Evaluate (not "cookies") => **#f**

So, value of if is value of $Expression_2$

=> **"starve"**

DrScheme Languages

- If you didn't set the language correctly in DrScheme, you got different answers!
- The "Beginning Student" has different evaluation rules
  - The rules are more complex
  - But, they gave more people what they expected

Comparing Languages

```
Welcome to DrScheme, version 205.
Language: Pretty Big (includes MrEd and Advanced).
> +
#<primitive:+>
```

Welcome to DrScheme, version 205.
Language: Beginning Student.

```
> +
+: this primitive operator must be applied to arguments;
expected an open parenthesis before the primitive
operator name
> ((lambda (x) x) 200)
function call: expected a defined name or a primitive
operation name after an open parenthesis, but found
something else
```


closer-color? (Green Star version)

```
(define (closer-color? sample color1 color2)
  (<
    (+ (abs (- (get-red color1) (get-red sample)))
       (abs (- (get-blue color1) (get-blue sample)))
       (abs (- (get-green color1) (get-green sample))))
    (+ (abs (- (get-red color2) (get-red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
  ))
```



```
(+ (abs (- (get-red color1) (get-red sample)))
   (abs (- (get-blue color1) (get-blue sample)))
   (abs (- (get-green color1) (get-green sample))))
(define (closer-color? sample color1 color2)
  (<
    (+ (abs (- (get-red color2) (get-red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
  ))
```



```
(lambda (                              )
  (+ (abs (- (get-red color1 ) (get-red sample )))
     (abs (- (get-blue color1) (get-blue sample )))
     (abs (- (get-green color1) (get-green sample))))
(define (closer-color? sample color1 color2)
  (<
    (+ (abs (- (get-red color2) (get-red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
  ))
```


(define color-difference (lambda (colora colorb))

```
(+ (abs (- (get-red colora ) (get-red colorb  )))
   (abs (- (get-blue colora) (get-blue colorb  )))
   (abs (- (get-green colora) (get-green colorb )))) ))
(define (closer-color? sample color1 color2)
  (<
    (- (color-difference color2 sample) -red sample)))
       (abs (- (get-blue color2) (get-blue sample)))
       (abs (- (get-green color2) (get-green sample))))
  ))
```



```
(define color-difference
  (lambda (colora colorb)
    (+ (abs (- (get-red colora) (get-red colorb)))
       (abs (- (get-green colora) (get-green colorb)))
       (abs (- (get-blue colora) (get-blue colorb))))))

(define (closer-color? sample color1 color2)
  (<  (color-difference color1 sample)
      (color-difference color2 sample)))
```

What if you want to use square instead of abs?