



# Functional Programming with Scheme

---

Keep reading: Scott, Chapter 11.5-  
11.6

# Lecture Outline

---

- Notes on writing “comments” for homework
- Scheme
  - Equality testing
  - Higher-order functions
  - `map`, `foldr`, `foldl`
  - Tail recursion

# Writing “Comments”

---

Each function should have the following sections:

;; **Contract:** `len : (list a) -> integer`

;; **Purpose:** to compute length of a list `lis`

;; **Example:** `(len '(1 2 3 4))` should return 4

;; **Definition:**

```
(define (len lis)
  (if (null? lis) 0 (+ 1 (len (cdr lis)))))
```

# Writing “Comments”

---

:: **Contract:**

:: **len** : (**list** **a**) -> number

- Has two parts. The first part, to the left of the colon, is the name of the function. The second part, to the right of the colon, states what type of data it consumes and what type of data it produces
- We shall use **a**, **b**, **c**, etc. to denote type parameters, and **list** to denote the **list** type
- Thus, **len** is a function, which consumes a list whose elements are of some type **a**, and produces a number

# Writing “Comments”

---

**:: Contract:**

**:: lis :** (**list integer**) -> (**list integer**)

- **lis :** is a function, which consumes a list of integers, and produces a list of integers (you may assume that the input is a list of integers)

**:: Purpose:** to compute ...

# Writing “Comments”

---

- Comments are extremely important in Scheme
- Why?
- Our “comments” amount to adding an unchecked type signature + an informal behavioral specification

# Lecture Outline

---

- Notes on writing “comments” for homework
- Scheme
  - Equality testing
  - Higher-order functions
  - `map`, `foldr`, `foldl`
  - Tail recursion

# Equality Testing

---

## eq?

- Built-in predicate that can check atoms for equal values
- Does not work on lists in the way you might expect!

## eql?

- Our predicate that works on lists

```
(define (eql? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x)) (not (atom? y))
            (eql? (car x) (car y))
            (eql? (cdr x) (cdr y)))))
```

## equal?

- Built-in predicate that works on lists



# Examples

---

(eq1? '(a) '(a)) **yields** what?

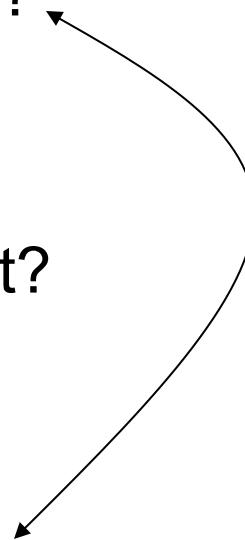
(eq1? 'a 'b) **yields** what?

(eq1? 'b 'b) **yields** what?

(eq1? '((a)) '((a))) **yields** what?

(eq? 'a 'a) **yields** what?

(eq? '(a) '(a)) **yields** what?



# Models for Variables

---

## ■ Value model for variables

- A variable is a **location** that holds a **value**
  - I.e., a named container for a value

■  $a := b$

↖  
l-value (the location)

↖  
r-value (the value held in that location)

## ■ Reference model for variables

- A variable is a **reference** to a **value**
- Every variable is an l-value
  - Requires dereference when r-value needed (usually, but not always implicit)

# Models for Variables: Example

`b := 2;`

`c := b;`

`a := b + c;`


## ■ Value model for variables

■ `b := 2`      `b:` 2

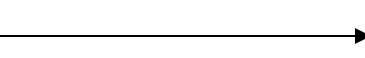
■ `c := b`      `c:` 2

■ `a := b+c`      `a:` 4

## ■ Reference model for variables

■ `b := 2`      `b`  2

■ `c := b`      `c`  2

■ `a := b+c`      `a`  4

# Equality Testing: How does eq? work?

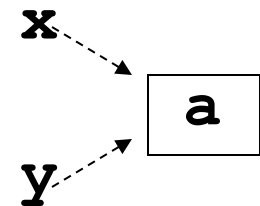
- Scheme uses the reference model for variables!

```
(define (f x y) (list x y))
```

Call `(f 'a 'a)` yields `(a a)`

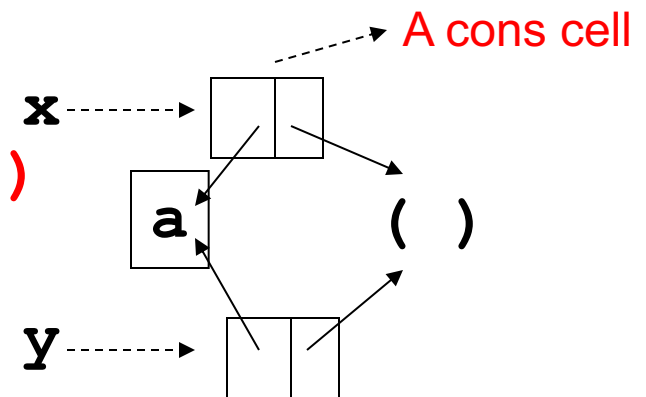
`x` refers to atom `a` and `y` refers to atom `a`.

`eq?` checks that `x` and `y` both point to the same place.



Call `(f '(a) '(a))` yields `((a) (a))`

`x` and `y` do not refer to the same list.



# Models for Variables

---

- C/C++, Pascal, Fortran
  - Value model
- Java
  - Mixed model: value model for simple types, reference model for class types
- JS, Python, R, etc.
  - Reference model
- Scheme
  - Reference model! **eq?** is “reference equality” (akin of Java’s ==), **equal?** is value equality

# Equality Testing

---

- In languages with reference model for variables we have two tests for equality
  - One tests reference equality, whether two references refer to the same object
    - `eq?` in Scheme `(eq? ` (a) ` (a) )` yields `#f`
    - `==` in Java
  - Other tests value equality. Even if the two references do not refer to the same object, they may refer to objects that have the same value
    - `equal?` in Scheme `(equal? ` (a) ` (a) )` yields `#t`
    - `.equals()` method in Java

# Lecture Outline

---

- Notes on writing “comments” for homework
- Scheme
  - Equality testing
  - Higher-order functions
  - `map`, `foldr`, `foldl`
  - Tail recursion

# Higher-order Functions

---

- Functions are first-class values
- A function is said to be a **higher-order function** if it takes a function as an argument or returns a function as a result
- Functions as arguments

```
(define (f g x) (g x))  
(f number? 0) yields #t  
(f len '(1 (2 3))) yields what?  
(f (lambda (x) (* 2 x)) 3) yields what?
```



# Higher-order Functions

---

- Functions as return values

```
(define (fun)  
  (lambda (a) (+ 1 a)))
```

**(fun 4) yields what?**

**((fun) 4) yields what?**

# Higher-order Functions: `map`

---

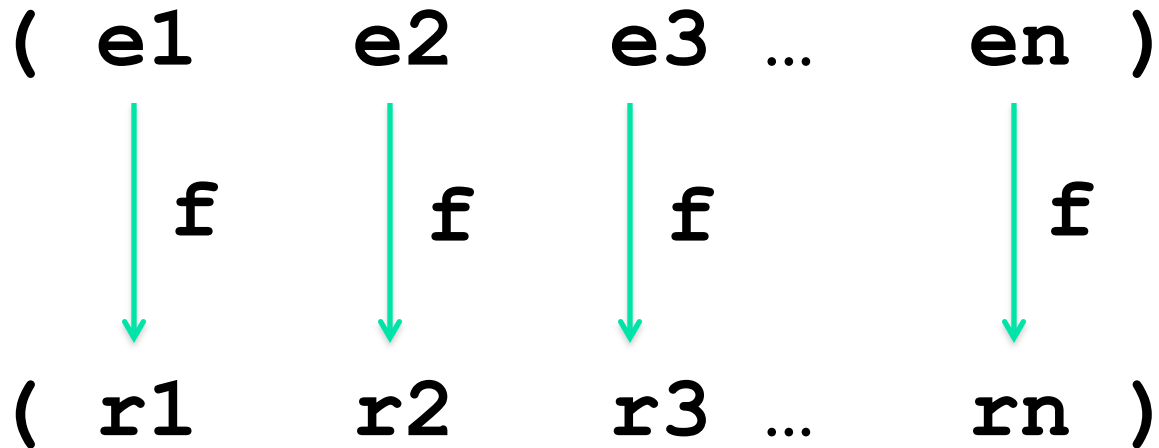
- Higher-order function used to apply another function to every element of a list
- Takes 2 arguments: a function `f` and a list `lis` and builds a new list by applying the `f` to each element of `lis`

```
(define (mymap f lis)
  (if (null? lis) '()
      (cons (f (car lis)) (mymap f (cdr lis)))))
```

# map

---

`(map f lis)`



There is a build-in function `map`

# map

---

```
(define (mymap f l)
  (if (null? l) '()
      (cons (f (car l)) (mymap f (cdr l)))))
```

`(mymap abs '(-1 2 -3 -4))` **yields** `(1 2 3 4)`

`(mymap (lambda (x) (+ 1 x)) '(1 2 3))` **yields**  
what?

`(mymap (lambda (x) (abs x)) '(-1 2 -3))` **yields**  
what?

# map

---

Remember **atomcount**, calculates the number of atoms in a list

```
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (+ (atomcount (car s))
                  (atomcount (cdr s))))))
```

We can write **atomcount2**, using map:

```
(define (atomcount2 s)
  (cond ((atom? s) 1)
        (else (apply + (map atomcount2 s)))))
```

# map

---

```
(define (atomcount2 s)
  (cond ((atom? s) 1)
        (else (apply + (map atomcount2 s)))))
```

(atomcount2 '(1 2 3)) yields 3

(atomcount2 '((a b) d)) yields 3

(atomcount2 '(1 ((2) 3) (((3) (2) 1)))) ?

# Question

---

My **atomcount2** defined below

```
(define (atomcount2 s)
  (cond ((atom? s) 1)
        (else (apply + (map atomcount2 s))))))
```

has a bug :). Can you find it?

Answer: It counts the null list ``()` as an atom.

E.g., `(atomcount2 `())` will return 1.

# Exercise

---

```
(define (atomcount2 s)
  (cond ((atom? s) 1)
        (else (apply + (map atomcount2 s))))))
```

Now, let's write **flatten2** using **map**

```
(flatten2 '(1 ((2) 3) (((3) (2) 1)))) yields
(1 2 3 3 2 1)
```

```
(define (flatten2 s)
```

...

Hint: you can use **(apply append (...**



# foldr

---

- Higher-order function that “folds” (“reduces”) the elements of a list into one, from right-to-left
- Takes 3 arguments: a binary operation **op**, a list **lis**, and initial value **id**. **forldr** “folds” **lis**

(define (foldr op lis id)

(if (null? lis) id

(op (**car lis**) (foldr op (**cdr lis**) id)) ))

(foldr + '(10 20 30) 0) **yields** 60

it is 10 + (20 + (30 + 0))

(foldr - '(10 20 30) 0) **yields** ?

# foldr

---

(foldr op lis id)

( e<sub>1</sub> ... e<sub>n-1</sub> e<sub>n</sub> ) id

( e<sub>1</sub> ... e<sub>n-1</sub> ) res<sub>1</sub>

...

( e<sub>1</sub> ) res<sub>n-1</sub>

res<sub>n</sub>

# Exercise

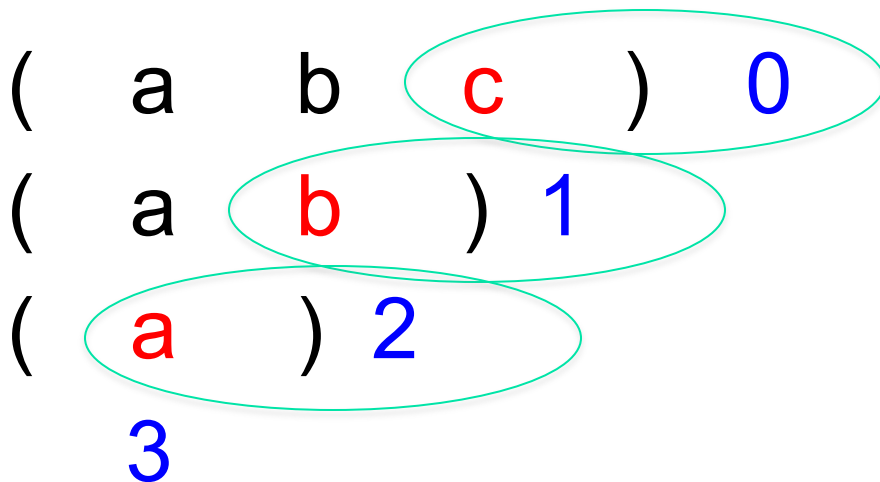
---

- What does  
`(foldr append '((1 2) (3 4)) '())` yield?  
Recall that `append` appends two lists:  
`(append '(1 2) '((3) (4 5)))` yields `(1 2 (3) (4 5))`
- Now, define a function `len2` that computes the length of a list using `foldr`  
`(define (len2 lis)`  
    `(foldr ...`

# foldr

`(define (len2 lis) (foldr (lambda (x y) (+ 1 y)) lis 0))`

List element Partial result

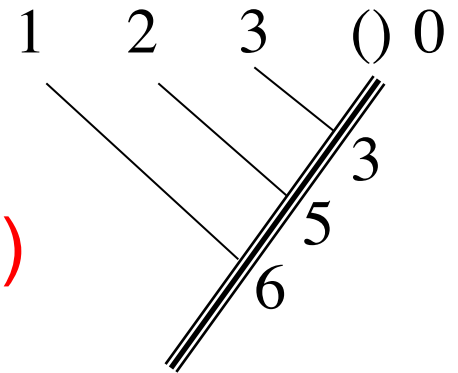


# foldr

- **foldr** is right-associative

- E.g., **(foldr + '(1 2 3) 0)** is **1 + (2 + (3 + 0))**
- Partial results are calculated in order down the else-branch

```
(define (foldr op lis id)
  (if (null? lis) id
      (op (car lis) (foldr op (cdr lis) id))))
```



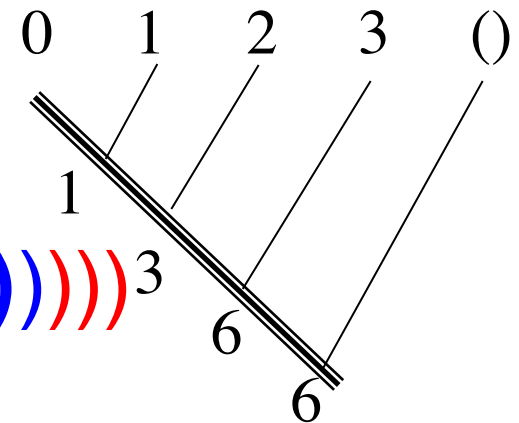
# foldl

- **foldl** is left-associative and (as we shall see) more efficient than **foldr**
  - E.g., **(foldl + '(1 2 3) 0)** is **((0 + 1) + 2) + 3**
  - Partial results are accumulated in **id**

**(define (foldl op lis id)**

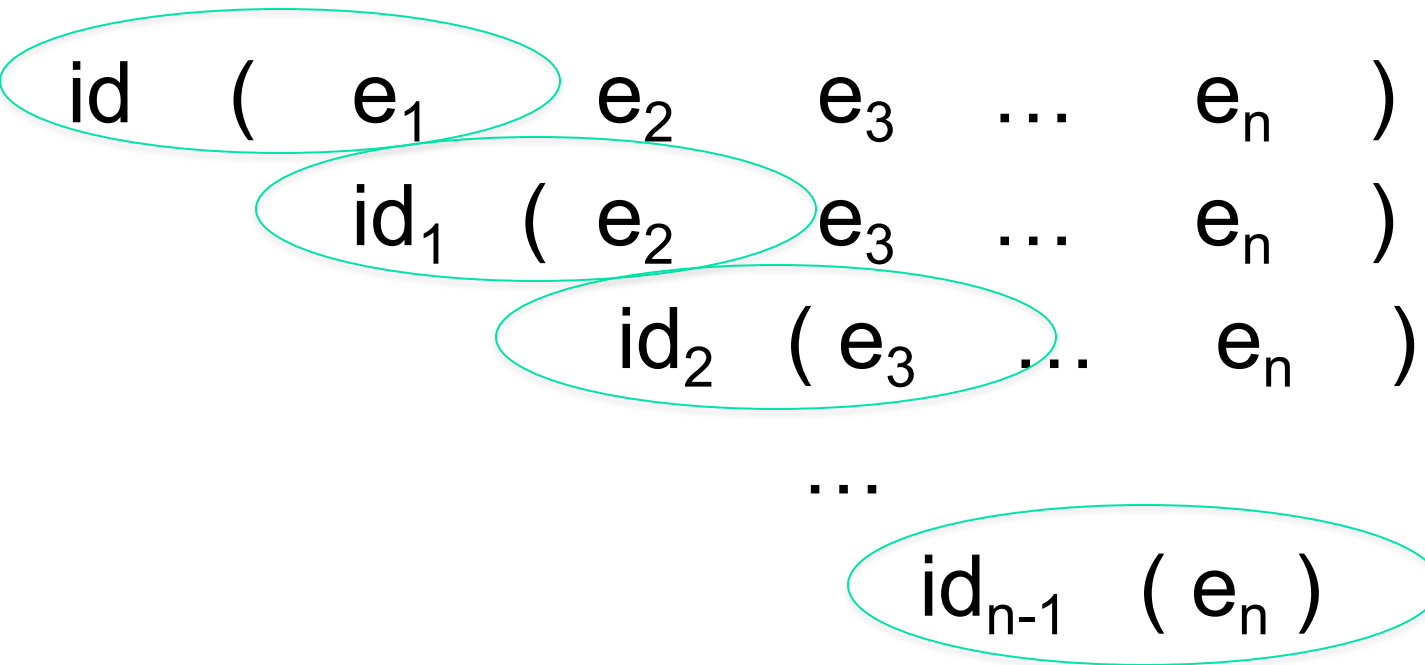
**(if (null? lis) id**

**(foldl op (cdr lis) (op id (car lis))))**



# foldl

(foldl op lis id)



id<sub>n</sub>

# Exercise

---

```
(define (foldl op lis id)
  (if (null? lis) id
      (foldl op (cdr lis) (op id (car lis))))))
```

- Define a function **rev** computing the reverse of a list using **foldl**  
E.g., (rev '(a b c)) yields (c b a)

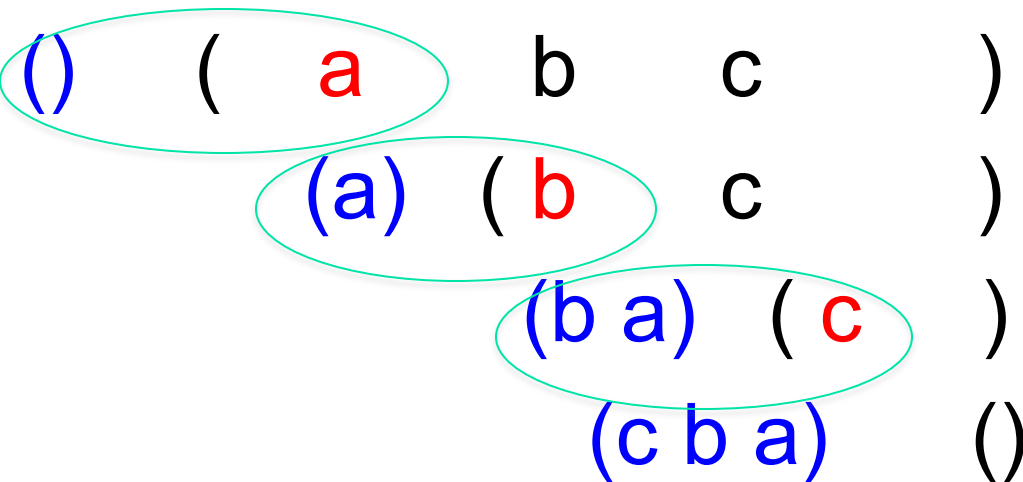
```
(define (rev lis)
  (foldl (lambda (x y) (...
```



# foldl

(define (rev lis) (foldl (lambda (x y) (cons y x)) lis '()))

Next element  
Partial result



# Exercise

---

```
(define (foldr op lis id)
  (if (null? lis) id
      (op (car lis) (foldr op (cdr lis) id)) ))
```

```
(define (foldl op lis id)
  (if (null? lis) id
      (foldl op (cdr lis) (op id (car lis))))) )
```

- Write **len**, which computes the length of the list, using **foldl**
- Write **rev**, which reverses the list, using **foldr**

# Exercise

---

```
(define (foldl op lis id)
  (if (null? lis) id (foldl op (cdr lis) (op id (car lis)))) )
```

- Write **len**, which computes the length of the list, using **foldl**

```
(define (len lis) ...
```

# Exercise

---

Can you write the contract for **foldl** ?

```
(define (foldl op lis id)
```

```
  (if (null? lis) id
```

```
      (foldl op (cdr lis) (op id (car lis))))))
```

:: **Contract:**

:: foldl : (**b** \* **a** -> **b** ) \* (list **a**) \* **b** -> **b**

# Exercise

---

How about the contract for **foldr** ?

```
(define (foldr op lis id)
```

```
  (if (null? lis) id
```

```
      (op (car lis) (foldr op (cdr lis) id))))
```

:: **Contract:**

:: foldr : (a \* b -> b ) \* (list a) \* b -> b

# foldr vs. foldl

---

```
(define (foldr op lis id)
```

```
  (if (null? lis) id
```

```
      (op (car lis) (foldr op (cdr lis) id)) ))
```

```
(define (foldl op lis id)
```

```
  (if (null? lis) id
```

```
      (foldl op (cdr lis) (op id (car lis))) ))
```

- Compare underlined portions of these 2 functions
  - **foldr** contains a recursive call, but it is not the entire return value of the function
  - **foldl** returns the value obtained from the recursive call to itself!

# Tail Recursion

---

- If the result of a function is computed without a recursive call OR it is the result of an immediate recursive call, then the function is said to be **tail recursive**
  - E.g., **foldl**
- Tail recursion can be implemented efficiently
  - Result is accumulated in one of the arguments, and stack frame creation can be avoided!
  - Scheme implementations are required to be “properly tail-recursive”

# Tail Recursion: Two Definitions of Length

---

```
(define (len lis)
  (if (null? lis)
      0
      (+ 1 (len (cdr lis)))))
```

(len '(3 4 5))

```
(define (lenh lis total)
  (if (null? lis)
      total
      (lenh (cdr lis) (+ 1 total))))
```

```
(define (len lis) (lenh lis 0))
```

(len '(3 4 5))

Lenh is tail recursive. total accumulates the length



# Tail Recursion: Two Definitions of Factorial

---

```
(define (factorial n)
  (cond ((zero? n) 1)
        ((eq? n 1) 1)
        (else (* n (factorial (- n 1))))))
```

```
(define (fact2 n acc)
  (cond ((zero? n) 1)
        ((eq? n 1) acc)
        (else (fact2 (- n 1)
                      (* n acc)))))
```

```
(define (factorial n)
  (fact2 n 1))
```

**fact2** is tail recursive

# The End

---