# Announcements

- HW4, Scheme, is out on Schedule page:
  - https://www.cs.rpi.edu/~milanova/csci4430/schedule.html
  - Submitty autograder coming soon
- If you haven't started with DrRacket, please do so soon!
  - http://racket-lang.org/
  - Remember to revert language to R5RS





# Functional Programming with Scheme

Keep reading: Scott, Chapter 11.5-11.6

## Lecture Outline

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



## Writing Contracts

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 Contracts

;; 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
- Use a, b, c, etc. to denote type parameters, and use list to denote the list type
- Thus, **len** is a function consuming a list with elements of some type a, and produces a number



## Writing Contracts

;; Contract:

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

- **lis** : is a function that consumes a list of integers and produces a list of integers

;; Purpose: to compute …

## Writing Contracts

- Comments are extremely important in Scheme
- Why?
- Our "comments" amount to adding an unchecked type signature, plus an informal behavioral specification





## Recursive Functions

```
(define (app  x  y)
  (cond  ((null?  x)  y)
         ((null?  y)  x)
         (else
            (cons  (car x)
                   (app (cdr x)  y)))))
```

app is a shallow recursive function

- What does app do?

(app '() '()) yields ?
(app '() '(1 4 5)) yields ?
(app '(5 9) '(a (4) 6)) yields ?

# Exercise

```
(define (len x)
  (cond ((null? x) 0) (else (+ 1 (len (cdr x))))))
```

Write a version of len that uses if instead of cond

Write a function countlists that counts the number of list elements in a list. E.g.,

`(countlists '(a))` yields `0`

`(countlists '(a (b c (d)) (e)))` yields `2`

Use type predicate list?.





# Recursive Functions

```
(define  (fun  x)
  (cond ((null? x)  0)
        ((atom? x)  1)
        (else  (+ (fun  (car x))
                  (fun  (cdr x)))) ))
```

fun is a deep recursive function

What does fun do?

```
(define (atom? obj)
  (not (pair? obj)))

(atom? 'a) → #t
(atom? '(a)) → #f
(atom? '()) → #t
```

# fun counts atoms in a list

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

atomcount is a deep recursive function

`(atomcount '(a))` **yields 1**
`(atomcount '(1 (2 (3)) (5)))` **yields 4**

Counts Name and Number atoms nested (arbitrarily deep) in x.

Trace: (atomcount '(1 (2 (3)) )
1> (+ (atomcount 1) (atomcount '( (2 (3)) ) ))
2> (+ (atomcount '(2 (3)) ) (atomcount '( ) ) )
3> (+ (atomcount 2) (atomcount '((3)) )
4> (+ (atomcount '(3)) (atomcount '( )) )
5> (+ (atomcount 3) (atomcount '( )))

**0**

**1**

11

11

# Exercise

- Write a function flatten that flattens a list

`(flatten '(1 (2 (3))))` **yields** `(1 2 3)`

```
(define (flatten x)
   (cond ((null? x) '())
         ((atom? x) (cons x '()))   ;; constructs list (x)
         (else (append (flatten (car x))
                       (flatten (cdr x))))
   )
)
```

## Lecture Outline

- *Notes on writing "comments" for homework*
- *Scheme*
  - *Recursive functions*
  - *Equality testing*
  - *Higher-order functions*
  - ***map, foldr, foldl***
  - *Tail recursion*

Programming Languages CSCI 4430, A. Milanova/B. G. Ryder 13

13

## 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

14

14

# Examples

(eql? ‘(a) ‘(a)) **yields** what? #t

(eql? ‘a ‘b) **yields** what? #f

(eql? ‘b ‘b) **yields** what? #t

(eql? ‘((a)) ‘(a)) **yields** what? #f

(eq? ‘a ‘a) **yields** what? #t

(eq? ‘(a) ‘(a)) **yields** what? #f





# 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)

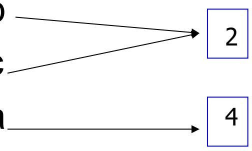
# Models for Variables: Example

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
  - c := b c
  - a := b+c a








# Equality Testing: How does eq? work?

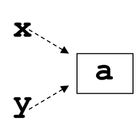
- Scheme uses the reference model for variables

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

Call **(f 'a 'a)** yields **(a a)**

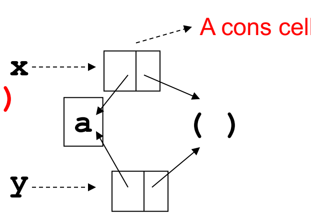
**x** refers to atom **a** and **y** refers to atom **a**.

**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

19

19

## 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

20

20

## Lecture Outline

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



## Higher-order Functions

- In Scheme, functions are first-class values
- A function is 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)))
```

→ The plus-1 function.

**(fun 4) yields** what?

ERROR!

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

5



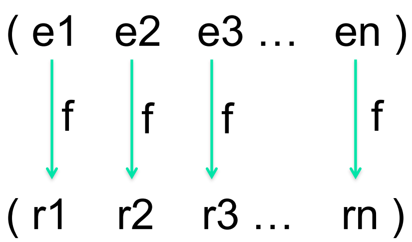
## Higher-order Functions: map

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

```
(define (my-map f lis)
  (if (null? lis) '()
    (cons (f (car lis)) (my-map f (cdr lis)))))
```

## map

(my-map f lis)




There is a build-in function **map**



## map

```
(define (my-map f l)
    (if (null? l) '( )
      (cons (f (car l)) (my-map f (cdr l)) )))
```

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

`(my-map (lambda (x) (+ 1 x)) '(1 2 3))` **yields** what? (2 3 4)

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

## map

(apply + (1 2 3)) → 6

Remember atomcount, counts 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

(apply + (1 1 1)) → 3

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

(apply + (2 1)) → 3

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

(apply + (1 2 3)) → 6

## Question

My atomcount2 defined below

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

Add ((null? s) 0) clause!

has a subtle bug :). Can you find it?

Answer: 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)
  (cond ((null? s) '())
        ((atom? s) (cons s '()))
        (else (apply append (map flatten2 s)))
  )
)
```

Hint: you can use `(apply append (…`

# Exercise





# foldr

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

as 10 + (20 + (30 + 0))

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

# foldr

(foldr op lis id)

( $e_1$ … $e_{n-1}$ $e_n$ ) id  (op)

( $e_1$ … $e_{n-1}$ ) $res_1$  (op)

…

( $e_1$ ) $res_{n-1}$  (op)

$res_n$





# Exercise

- What does
  `(foldr append '((1 2) (3 4)) '())` yield?
  Recall that append appends two lists: (1 2 3 4)
  `(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 (lambda (x y) (+ 1 y)) lis 0))
```

# foldr

List element

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

Partial result

op: (lambda (x y) (+ 1 y))

( a b c ) 0

( a b ) 1

( a ) 2

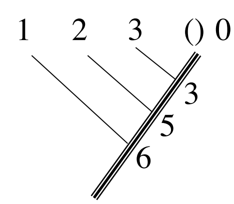
3





# 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

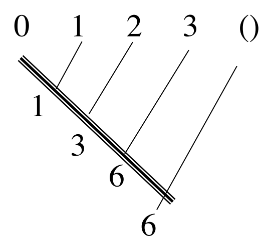



```
(define (foldr op lis id)
   (if (null? lis) id
      (op (car lis)
          (foldr op (cdr lis) id))))
```

# 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)

op

id ( $e_1$ $e_2$ $e_3$ … $e_n$ )

op

$id_1$ ( $e_2$ $e_3$ … $e_n$ )

$id_2$ ( $e_3$ … $e_n$ )

…

op

$id_{n-1}$ ( $e_n$ )

$id_n$

# 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) (cons y x)) lis '())
 )
```





# foldl

Next element

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

Partial result

op: (lambda (x y) (cons y x))

() ( a b c )

(a) ( b c )

(b a) ( c )

(c b a) ()

## 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, computing the length of the list, using foldl
- Write rev, reversing the list, using foldr





## Exercise

- Write len, computing the length of the list, using foldl

```
(define (len lis) …
```

## Exercise

b*a->b (list a) b

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

Connection between the arguments and return of "op" and list elements and "id" is important.



## 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 two functions
  - foldr contains a recursive call, but it is **not** the entire return value of the function
  - foldl's recursive call is the entire return value!



## 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 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))
```



## 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))
```

# The End