Announcements

- Quiz 5

- HW6 due October 23
  - Functional correctness: 75%, comments: 25%
  - Writing “comments”: We require that you use the style from “How to Design Programs” (with some modifications by me)

Notes on HW6

- Do not define functions myand, myor, etc., then try to force evaluation of input list!

- Parse and interpret the expression
  - A recursive evaluate (recursive descent)
    - If expression is a myand => do this ...
    - If expression is a myor => do something else ...
    - etc.
  - Note from assignment text: “In order to maintain the set of bindings, consider using a list where each element of the list is one specific binding. A binding is really just a pair of an identifier and a boolean value.”

Notes on HW6

- 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:
;; lis : (list number) -> (list number)
;; lis : is a function, which consumes a list of numbers, and returns a list of numbers (you may assume that the input is a list of integer)
;; Purpose: to compute …
```
Last Class

- Functional programming languages
- Scheme
  - S-expressions, lists
  - cons, car, cdr
  - Defining Functions
  - Recursive Functions
    - Shallow recursion, Deep recursion
- Equality testing

Functional Programming with Scheme

Keep reading: Scott, Chapter 11

Lecture Outline

- Scheme
  - Equality testing
  - Higher-order functions
    - map, foldr, foldl
  - Tail recursion

Equality Testing

`eq?`
- Built-in predicate that can check atoms for equal values
- Doesn’t work on lists as you would expect

`equal?`
- Built-in predicate that works on lists

`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))))

Equality testing: Examples

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

Built-in equal? works like our eql?

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

Models for Variables (Scott, p. 225)

- Value model for variables
  - A variable is a location that holds a value
    - A named container for a value
    - e.g., a := b
      l-value (the location)  v-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 v-value needed (usually implicit)
Models for Variables: Example

- Value model for variables
  \[ b := 2; \]
  \[ c := b; \]
  \[ a := b + c; \]

- Reference model for variables
  \[ b := 2 \]
  \[ c := b \]
  \[ a := b + c; \]

Questions

- What is the model for variables in C/C++?
  - Value model

- Python?
  - Reference model

- Java?
  - Mixed. Value model for variables of simple type, reference model for variables of reference type

- Scheme?

Equality Testing: How does eq? work?

- Scheme uses the reference model for variables

  \[
  \text{(define (f x y) (list x y))}
  \]

  Call \( f \ '(a) \) yields \( (a a) \)
  \( x \) refers to atom \( a \) and \( y \) refers to atom \( a \).
  \( \text{eq?} \) 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.

Equality Testing

- In languages with reference model for variables we need two tests for equality
  - One tests reference equality, whether two references refer to the same object
    - \( \text{eq?} \) in Scheme \( (\text{eq?} \ 'a\ '(a)) \) yields \#t
    - \( == \) in Java
  - Other tests value equality. Even if the two references do not refer to the same object, the objects can have the same value
    - \( \text{equal?} \) in Scheme \( (\text{equal?} \ 'a\ '(a)) \) yields \#t
    - \( .equals() \) in Java

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

  \[
  \text{(define (f g x) (g x))}
  \]

  \( f \) number? 0 \ yields \#t
  \( f \) len \( (\{ 1 \ 2 \ 3 \}) \) \ yields \ what?
  \( f \) (lambda (x) \( (*\ 2\ x) \)) \ yields \ what?

Higher-order Functions

- Functions as return values

  \[
  \text{(define (fun)}
  \]

  \( \text{(lambda (a) (+ 1 a))} )\)

  \( (\text{fun 4}) \ yields \ what? \)

  \( (\text{(fun 4)}) \ yields \ what? \)
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 $lis$

$\text{(define (mymap } f \ lis) \text{)}$

$(\text{if (null? } lis) \text{) (})$

$(\text{cons (f (car lis)) (mymap } f \ \text{(cdr lisis)))))$
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. It “folds” ("reduces") 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 ?

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

- 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))))
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!

Tail Recursion: Two Definitions of Length

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

(len '(3 4 5))

(define (len h lis total)
  (if (null? lis) total
   (len h (cdr lis) (+ 1 total)))
)

(define (len lis) (len 0 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