Announcements

- HW5 due March 28
  - Homework Server link is up
  - I will have office hours, Fri or Mon, check Announcements page

- Functional correctness: 75%, comments: 25%
- Writing "comments". We require that you use the style from "How to Design Programs" (some modifications by me)

Writing "Comments"

Each function should have the following sections:

;; Contract: len : (list a) -> number
;; Purpose: to compute length of a list lis
;; Example: (len '(1 (2 3) 4)) should return 3
;; 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"

Comments are extremely important in Scheme. Perhaps more so than in Java, C++ and C

Why?

Our "comments" amount to adding an unchecked type signature + an informal behavioral spec

Writing "Comments"

;; Contract:
;; lcs : (list symbol) * (list symbol) -> (list symbol)
- lcs : is a function, which consumes a list of symbols, another list of symbols, and returns a list of symbols (you may assume that the inputs are lists of symbolic constants such as a, b, sam)
- Purpose: to compute the longest common sublist of two lists

Last Class

- Functional Programming Languages
- Scheme
  - S-expressions, lists
  - cons, car, cdr
- Defining Functions
- Recursive Functions
  - Shallow recursion, Deep 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.
  ```scheme
  (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
  - 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

```scheme
b := 2;
c := b;
a := b + c;
```

- `b := 2` becomes `b:`
- `c := b` becomes `c:`
- `a := b + c` becomes `a:`
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 \texttt{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 \(a\).
  \texttt{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
    - \texttt{eq?} in Scheme \texttt{(eq? '(a) '(a))} yields \texttt{#f}
    - == 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
    - \texttt{equal?} in Scheme \texttt{(equal? '(a) '(a))} yields \texttt{#t}
    - \texttt{.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
    (define (f g x) (g x))
    (f number? 0) \texttt{yields} \texttt{#t}
    (f len '(1 2 3)) \texttt{yields} what?
    (f (lambda (x) (* 2 x)) 3) \texttt{yields} what?
  - Functions as return value

Higher-order Functions: \texttt{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)

\(\begin{array}{cccccccc}
\text{e1} & \text{e2} & \text{e3} & \ldots & \text{en} \\
\text{f} & \text{f} & \text{f} & \text{f} \\
\text{r1} & \text{r2} & \text{r3} & \ldots & \text{rn}
\end{array}\)

There is a build-in function \texttt{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))))))

(atomcount2 '(1 2 3)) yields 3
(atomcount2 '((a b) d)) yields 3
(atomcount2 '(1 ((2) 3)) (((3) 2) 1))) yields what?

---

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 us 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. 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 ?

foldr

(foldr op lis id)

( e1 ... en-1 en  ) id
( e1 ... en-1 ) res1
...
( e1 ) resn-1

resn
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

- 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) ( …

\[ (define \text{foldl} \ (\text{foldl} \ \lambda (x\ y) (\text{cons} \ y \ x)) \ \text{lis} \ '()) \]

Next element
Partial result

\[ (\) \ (a\ b\ c) \]
\[ (\ a\ ) \ (b\ c) \]
\[ (\ b\ a\ ) \ (c) \]
\[ (\ c\ b\ a\ ) \ () \]

Exercise

Can you write the contract for \text{foldl}?
\[ (define \text{foldl} \ \text{op} \ \text{lis} \ \text{id}) \]
\[ (if \ (null? \ \text{lis}) \ \text{id} \]
\[ \ (\text{foldl} \ \text{op} \ (\text{cdr} \ \text{lis}) \ (\text{op} \ \text{id} \ (\text{car} \ \text{lis})))) \]

;; Contract:
;; \text{foldl} \ : \ ('b \ * \ 'a \ -> \ 'b) \ * \ (\text{list} \ 'a) \ * \ 'b \ -> \ 'b

Exercise

How about the contract for \text{foldr}?
\[ (define \text{foldr} \ \text{op} \ \text{lis} \ \text{id}) \]
\[ (if \ (null? \ \text{lis}) \ \text{id} \]
\[ \ (\text{op} \ (\text{car} \ \text{lis}) \ (\text{foldr} \ \text{op} \ (\text{cdr} \ \text{lis}) \ \text{id}))) \]

;; Contract:
;; \text{foldr} \ : \ ('a \ * \ 'b \ -> \ 'b) \ * \ (\text{list} \ 'a) \ * \ 'b \ -> \ 'b