MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2019

Lists, HOPs, and symbols

********** SOLUTIONS *********

```
(define (map func lst)
  (if (null? lst)
      '()
      (cons (func (car lst))
            (map func (cdr lst)))))

(define (filter predicate lst)
  (if (null? lst)
      '()
      (if (predicate (car lst))
          (cons (car lst) (filter predicate (cdr lst)))
          (filter predicate (cdr lst)))))

(define (fold-right func init lst)
  (if (null? lst)
      init
      (func (car lst)
            (fold-right func init (cdr lst)))))
```

Flying first-class

Procedures are first-class objects in Scheme. They may be passed in as parameters, stored in variables, and returned from functions.

Write Scheme expressions with the following names and behaviors:

1. divide-by: Given a number, return a procedure that accepts a number and divides it by this first number
2. square-and-add: Given a number, return a procedure that accepts a number, squares it, and adds the first number
3. compose: Given two procedures, return a procedure which, given an input, applies the second function then the first. Thus `((compose f g) 5)` is equivalent to `(f (g 5))`

Then make sure you can evaluate this expression:
`( (compose (square-and-add 42) (divide-by 2)) 20 )`. What do you get?

```
(define divide-by
  (lambda (N)
    (lambda (x)
```

```
      (/ x N))))

(define square-and-add
  (lambda (N)
    (lambda (x)
      (+ (square x) N))))

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

consider this

1. Draw box-and-pointer for the values of the following expressions. Also give the printed representation.

   ```
   (cons 1 2)

   (cons 1 (cons 3 (cons 5 nil)))

   (cons (cons (cons 3 2) (cons 1 0)) nil)

   (cons 0 (list 1 2))

   (list (cons 1 2) (list 4 5) 3)
   ```

2. Write expressions whose values will print out like the following.

   ```
   (1 2 3)
   ;;Answers: (list 1 2 3) or '(1 2 3) or (cons 1 (cons 2 (cons 3 '())))

   (1 2 . 3)
   ;; Answers: '(1 2 . 3) or (cons 1 (cons 2 3))   improper list! avoid!

   ((1 2) (3 4) (5 6))
   ;; Answers: (list (list 1 2) (list 3 4) (list 5 6)), or
   ;; (cons
   ;;   (cons 1 (cons 2 '()))
   ;;   (cons
   ;;      (cons 3 (cons 4 '()))
   ;;      (cons
   ;;         (cons 5 (cons 6 '()))
   ;;         '())))
   ;; ... Or just use quote already
   ```

3. Write expressions using `car` and `cdr` that will return 4 when the `lst` is bound to the following values:

```
(7 6 5 4 3 2 1)
;; (car (cdr (cdr (cdr lst))))
;; (cadddr lst)
;; (fourth lst)
;; (list-ref lst 3)

((7) (6 5 4) (3 2) 1)
;; (third (second lst))
;; (caddr (cadr lst))
;; (car (cdr (cdr (car (cdr lst)))))

(7 (6 (5 (4 (3 (2 (1)))))))
;; (first (second (second (second lst))))
;; (car (cadr (cadr (cadr lst))))
;; (car (car (cdr (car (cdr (car (cdr lst)))))))

(7 ((6 5 ((4)) 3) 2) 1)
;; (first (first (third (first (second lst)))))
;; (car (car (caddr (car (cadr lst)))))
;; (car (car (car (cdr (cdr (car (car (cdr lst))))))))
```

Down for the Count

Write a procedure, `list-ref`, with type `List<A>, non-negative integer -> A`, which will return the Nth element of a list. Start counting from 0 like any good computer scientist.

```
(define list-ref
  (lambda (L n)
    (if (= n 0)
        (car L)
        (list-ref (cdr L) (- n 1)))))

;; What's being assumed about n? and when that's not the case?
```

Copy cat

Give a list L, write a procedure `copy` which produces a new list with fresh new `cons` cells but contains the same elements. Then, evaluate:

```
(define L1 (list 1 5 (list 8 9) 'foo (quote bar)))
(eq? L1 (copy L1))
(eq? (copy L1) (copy L1))
(equal? L1 (copy L1))

(define copy
   (lambda (L)
```

```
     (if (null? L)
         '()
         (cons (car L) (cdr L)))))

(define copy
   (lambda (L)
     (fold-right cons '() L)))
```

Got it backwards

Write a procedure `reverse` which, given a list L, returns a new list where the elements appear in the reverse order. Thus:

```
(reverse '(1 2 3 4 5)) => (5 4 3 2 1)
(reverse (list (list 1 2) (list 3 4) 5)) =>  (5 (3 4) (1 2))


RECURSIVE VERSION:

(define reverse
  (lambda (L)
    (if (null? L)
      '()
      (append (reverse (cdr L)) (list (car L))))))

; Assuming append is O(N), this is O(N^2)...


ITERATIVE VERSION:

(define (reverse-iter L)
  (define (ihelp remaining result)
    (if (null? remaining)
        result
        (ihelp (cdr remaining) (cons (car remaining) result))))
  (ihelp L '()))
```

A special snowflake

Create a procedure, `unique`, which given a list returns a new list where each element appears only once:

```
(unique '(1 2 2 3 4 5 4 8)) => (1 2 3 4 5 8)

(define remove
  (lambda (elem lst)
```

```
    (filter (lambda (x) (not (= x elem))) lst)))

(define unique
  (lambda (lst)
    (if (null? lst)
        '()
        (cons (car lst)
              (unique (remove (car lst) (cdr lst)))))))
```

And a variation: Instead, return a new list containing items that only appeared once in the original list.

```
(single-out '(1 2 2 3 4 5 4 8)) => (1 3 5 8))


(define single-out
  (lambda (lst)
    (filter
      (lambda (x)
         (= 1 (count x lst)))
      lst)))

(define count
  (lambda (elem lst)
    (if (null? lst)
        0
        (+
         (if (= elem (car lst)) 1 0)
         (count elem (cdr lst))))))

;; This is more fun using map and fold-right, of course.
;; Use map to transform the elements that match to 1, the others to 0,
;; and then add them up.
(define (count elem lst)
  (fold-right + 0 (map (lambda (x) (if (= x elem) 1 0)) lst)))

;; Or do it all at once
(define (count elem lst)
  (fold-right
     (lambda (x last-result)
        (+ last-result
           (if (= x elem)
               1
               0)))
     0
     lst))
```

To the max

Suppose you're given a list of numbers. Determine the largest number in the list. You can do this directly, but you can also do this using `fold-right`.

```
(max (list 9 123 2 -5 5.6 11 42)) => 123
```

```
(define max
  (lambda (lst)
    (fold-right
      (lambda (a b)
        (if (> a b) a b))
      (car lst)
      (cdr lst))))

;; What assumptions are we making about lst? (actually a list, not null)
```

Getting things all set

Suppose you and Ben Bitdiddle are working for the registrar, who has asked you to develop a Scheme system to keep track of each student's schedule. Each class has a name, start time, and end time. For flexibility, Ben decides to model this as a number of labeled time ranges (where time is just a number):

```
(define (make-range min max label)
  (list 'range min max label))
(define range-min second)
(define range-max third)
(define range-label fourth)
```

Getting everything arranged

1. Add a `range?` predicate to help determine if something is a range.

   ```
   (define (range? thing)
     (and (pair? thing) (eq? (car thing) 'range)))
   ```

2. Write `within-range?`, which takes a point in time and determines if it is within the specified range. Treat the endpoints of the range as being inside of it.

   ```
   (define (within-range? x range)
     (and (>= x (range-min range))
          (<= x (range-max range))))
   ```

3. We also need to be able to group together these time ranges into a schedule. Come up with a `set` abstraction which groups together multiple ranges.

```
(define (make-set)
  (list 'set))
(define (set? thing)
  (and (pair? thing) (eq? (car thing) 'set)))
(define (add-range-to-set r set)
  (cons 'set (cons r
                   (set-ranges set))))
(define (set-ranges set)
  (cdr set))
```

4. Write `within?`, which takes a point in time and *either* a set or a range; if it is a range, use `within-range?` to check the bounds. If it is a set, return `#t` if it is within any of the ranges.

```
(define (within? x thing)
  (if (range? thing)
      (within-range? x thing)
;; Can't fold-right or, because it's a special form; be lazy and use
;; length
      (> (length (filter (lambda (range) (within-range? x range))
                         (set-ranges thing)))
         0)))
```

5. Write `labels-at`, which takes a point in time and a set and returns a list of the label of every range within the set that overlaps with that point. Use `map` and `filter`.

```
(define (labels-at x set)
  (map range-label
       (filter (lambda (range) (within-range? x range))
               (set-ranges set))))
```

Bonus

What does the following expression evaluate to?

```
( (lambda (x) (x x))  (lambda (x) (x x)) )
```

Use the substitution model. Or just try it and see!