

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.S184—Zombies Drink Caffeinated 6.001
 IAP 2012

Lecture 2

Lists, HOPs, and symbols

***** SOLUTIONS *****

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))))
;; (caddr 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 *N*th element of a list. Start counting from 0 like any good computer scientist.

```
(define list-ref
  (lambda (L n)
    (if (= n 0)
        (first L)
        (list-ref (rest 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)
        '()
        (adjoin (first L) (rest 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 (rest L)) (list (first L))))))

(oof, O(N^2)...)

```

ITERATIVE VERSION:

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

```

You can also use `fold-left` to accomplish this¹

```
(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))

(define (reverse lst)
  (fold-left (lambda (x y) (cons y x)) '() lst))

```

While DrScheme's `foldr` matches the `fold-right` presented in lecture (also known as `accumulate`), DrScheme's built-in `foldl`² is not the same as the canonical fold-left algorithm.³

¹http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-15.html#%_idx_1814

²http://docs.plt-scheme.org/reference/pairs.html#%28def._%28%28lib._scheme/base.ss%29.foldl%29%29

³[http://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function))

Now write a variant, `deep-reverse` which also reverses any sub-lists. For example:

```
(deep-reverse '(1 (2 3) (4 (5 6 (7 8) 9) 10) 11)) => (11 (10 (9 (8 7) 6 5) 4) (3 2) 1)
```

The strategy here is to reverse the list itself, and then reverse each element in the list. For elements of the original list that aren't lists, leave them as-is.

```
(define (deep-reverse L)
  (map (lambda (x)
        (if (not (list? x))
            x
            (deep-reverse x)))
       (reverse L)))
```

Or more concisely, but probably less clearly:

```
(define (deep-reverse L)
  (if (not (list? L))
      L
      (map deep-reverse (reverse L))))
```

Boy, those nested parentheses are irritating, right? OK, write a procedure `flatten` which, given a list, removes all of the inner parentheses, producing a list with all elements of sublists as a single list, such as:

```
(flatten '(1 (2 3) (4 (5 6 (7 8) 9) 10) 11)) => (1 2 3 4 5 6 7 8 9 10 11)
```

```
(define flatten
  (lambda (L)
    (if (null? L)
        '()
        (if (not (list? L))
            (list L)
            (append (flatten (first L)) (flatten (rest L)))))))
```

We just flatten the car and the cdr, and stick 'em back together again. Sometimes the car will be a list, sometimes not; for the cases it isn't, we teach `flatten` to turn a single element into a list of that element, so the `append` works. `flatten` must always return a list.

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)
        '()
        (adjoin (first lst)
                 (unique (remove (first lst) (rest 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 (first lst)) 1 0)
         (count elem (rest 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))
     (first lst)
     (rest lst))))
```

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

Taking classes, abstractly

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 registration. Ben started working on this, but then got a call from Alyssa P. Hacker who really needs help with some sort of adventure game she's working on. Ben's only got an abstraction for a class so far:

```
(define (make-units C L H)    ;; Class, Lecture, Homework
  (list C L H))
(define get-units-C car)
(define get-units-L cadr)
(define get-units-H caddr)

(define (make-class number units)
  (list number units))
(define get-class-number car)
(define get-class-units cadr)
(define (get-class-total-units class)
  (let ((units (get-class-units class)))
    (+ (get-units-C units)
       (get-units-L units)
       (get-units-H units))))
(define (same-class? c1 c2)
  (= (get-class-number c1) (get-class-number c2)))
```

A schedule abstraction

The next thing that's needed is a way of representing a schedule, a set of classes.

1. Write a constructor that returns an empty schedule.

```
(define (empty-schedule)
  '())
```

2. Write a procedure that when given a class and a schedule, returns a new schedule including the new class:

```
(define (add-class class schedule)
  (cons class schedule))
```

3. Write a procedure that computes the total number of units in a schedule.

```
(define (total-scheduled-units sched)
  (if (null? sched)
      0
      (+ (get-class-total-units (first sched))
         (total-scheduled-units (rest sched)))))
```

4. Write a procedure that drops a particular class from a schedule and returns the new schedule

```
(define (drop-class sched classnum)
  (cond ((null? sched) nil)
        ((= (get-class-number (car sched)) classnum) (cdr sched))
        (else
         (cons (car sched) (drop-class sched classnum)))))
```

5. Implement the freshman credit limit by taking in a schedule, and removing classes until the total number of units is less than max-credits.

```
(define (credit-limit sched max-credits)
  (if (> (total-scheduled-units sched) max-credits)
      (credit-limit (cdr sched) max-credits)
      sched))
```

A student abstraction

Obviously you'll also need some students. From w20, Ben zephyrs you the following code:

```
(define (make-student number sched-checker)
  (list number (list) sched-checker))
(define get-student-number car)
(define get-student-schedule cadr)
(define get-student-checker caddr)

(define (update-student-schedule student schedule)
  (if ((get-student-checker student) schedule)
      (list (get-student-number student)
            schedule
            (get-student-checker student))
      "invalid schedule"))
```

6. Finish the call to `make-student` to limit the student to taking at least 1 class.

```
(make-student 575904467 (lambda (sched) (not (null? sched))))
```

7. Finish the call to `make-student` to create a first-term freshman (limited to 54 units).

```
(make-student 575904467 (lambda (sched) (< (total-scheduled-units sched) 54)))
```

8. Write a procedure that takes a schedule and returns a list of the names of the classes in the schedule. Use `map`.

```
(define (class-names schedule)
  (map get-class-number sched))
```

A Higher power

1. Rewrite `drop-class` to use `filter`.
2. Rewrite `total-scheduled-units` to use `map` and `fold-right`.
3. Rewrite `credit-limit` to use `fold-right`.

```
(define (drop-class sched classnum)
  (filter (lambda (class) (not (= (get-class-number class) classnum)))
    sched))
```

```
(define (total-scheduled-units sched)
  (fold-right 0 + (map get-class-total-units sched)))
```

```
(define (credit-limit sched limit)
  (fold-right
    (lambda (class sched)
      (if (< (total-scheduled-units (add-class class sched)) limit)
          (add-class class sched)
          sched))
    (empty-schedule)
    sched))
```

;; slick solution: ($O(n)$, not $O(n^2)$)

```
(define (credit-limit sched limit)
  (first
    (fold-right
      (lambda (class sched)
        (if (< (+ (get-class-total-units class) (second sched)) limit)
            (list (add-class class (car sched))
                  (+ (get-class-total-units class) (second sched)))
            sched))
      (list (empty-schedule) 0)
      sched)))
```

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!