CS3 – FALL 07: FINAL REVIEW SESSION

1. `count-sublists` (credit here to CS3 reader from Dan Garcia Fall 2003)
Write a procedure that takes a generalized list as an argument and returns how many sublists it contains. EX:

```
(count-sublists '(cs3 is cool)) → 0
(count-sublists '(((cs 3 is) really) ((cool)))) → 4
```

```
(define (count-sublists L)
        (cond
          ((null? L) 0)
          ((list? (car L))
                (+ 1 (count-sublists (car L))
                     (count-sublists (cdr L))))
         (else
           (count-sublists (cdr L)))))
```

2. `deep-filter`
Write `deep-filter` which takes in a list and a predicate, and returns the filtered list (much like `filter`). It must preserve the structure of the list. EX:

```
(deep-filter even? '(((3)) 2 1 ((2 3 4)))) → ((()) 2 ((2 4)))
```

```
(define (deep-filter pred? L)
  (cond ((null? L) '())
        ((list? (car L))
          (cons (deep-filter pred? (car L))
                (deep-filter pred? (cdr L))))
        ((pred? (car L))
          (cons (car L) (deep-filter pred? (cdr L))))
        (else (deep-filter pred? (cdr L)))))
```

3. `add-to-roster`
Write a procedure, called `add-to-roster`, that takes three lists — an association list called `current-roster`, a list of new player names, and a list of their numbers, correspondingly. This

procedure checks to see if the player is already in the roster, and if he is not, adds the new player, with its number, appropriately to the beginning of the roster. Should return the new roster at the end of the procedure. USE TAIL RECURSION and NO HELPER FUNCTIONS.
Example of current roster: `((Daei 10) (Mahdavikia 2) (Hashemian 9) (Karimi 8))`

```
(define (add-to-roster player-names current-roster)
        (cond ((null? player-names) current-roster)
              ((assoc (car player-names) current-roster)
                (add-to-roster (cdr player-names) (cdr player-numbers) current-roster))
              (else
                (add-to-roster (cdr player-names) (cdr player-numbers)
                                                    (cons  (list (car player-names)
                                                                 (car player-numbers))
                                                           current-roster)))))
```

4. `passed-tests?`
Complete the following without the use of explicit recursion or helper functions.

A. Write a procedure that takes in three lists as arguments. The first list is a list of procedures, each of which only takes a single argument. The second is a list of arguments for those procedures. The third is a list of expected results. Given these three lists, the procedure returns true if all of your test cases pass, false if any of them fail.

```
(define (passed-tests? proc-list arg-list result-list)
        (= (length (filter (lambda (p) p)  (map  (lambda (proc arg result)
                                                       (equal? (proc arg) result))
                                                 proc-list arg-list result-list)))
           (length proc-list)))
```

B. Provide a sample call to the procedure you just wrote, along with the correct return value.
(passed-tests? (list square car) ‘(2 (a b)) ‘(4 a)) => #t.
Be careful not to put ‘(square car) because quoting makes these words and no longer procedures.

5. `best-class` (Similar to the GPA lab exercise)
Write a procedure called `best-class` that takes a list as its argument. Each element in this list is also a list that contains two elements – the teacher's name and a list of his students' grades. `best-class` returns a list that contains the name of the teacher with the highest total of student grades. Do not write any additional helpers of your own. EX: `(best-class '((Johnson (A B A C NP A)) (Smith (C C D P F A)) (Doe (A A B A A NP)))` → `(Doe 24)`

A helper procedure `grade-conversion` is already written for you. It takes a letter grade as its argument and returns its corresponding numerical value. However, the only valid arguments are A, B, C, D and F. Here is the mapping for points: A=5, B=4, C=3, D=2, F=0.

```
(define (best-class class-list)
  (accumulate
   (lambda (l r)
     (if (> (cadr l)
            (cadr r))
         l
         r)
     )


  (map

    (lambda (c)
      (list (car c)

            (accumulate +

                        (map
                         (lambda (g)
                               (if (member g '(a b c d f))
                                   (grade-conversion g)
                                   0
                               ))
                         (cadr c)))))


   class-list)

   ))
```

6. Debug/Evaluate

```
(define (mystery1 lst)
   (map (lambda (elm)
           (cond ((not (list? elm)) elm)
                 (else (mystery1 elm))))
        lst))
```

A. Is there an argument that will make `mystery1` crash and generate an error? If so, what is it?

Non-list arguments

B. What does `(mystery1 '(a (b) ((c (d) e ((f)))) g))` return? (Assuming all possible bugs have been fixed.)

Same thing - Identity

```
(define (mystery2 lst)
   (cond ((null? lst) lst)
         ((list? (car lst))
          (append (map mystery2 (car lst))
                  (mystery2 (cdr lst))))
         (else (cons (car lst) (mystery2 (cdr lst))))))
```

C. Is there an argument that will make `mystery2` crash and generate an error? If so, what is the fix?

Yes – nested lists, or non list car.
(a (b c))

The bug is, we need a check to see if lst is a list so that we can take the car.
So add ((not (list? lst)) lst) as the line following the null? check.

PLEASE NOTE: During the review it was suggested that we simply change the (list? (car lst)) to (list? lst). This doesn't actually work. You'd have to change multiple calls to (car lst) as in the else case.

D. What does `(mystery2 '(a (b) ((c (d) e ((f)))) g))` return? (Assuming all possible bugs have been fixed.)

(a b (c d e (f)) g)

7. `black-out`

With Southwest, you can receive a free roundtrip after 8 roundtrips. However, these free tickets usually have black out dates. Write a procedure that returns the NON-blackout dates. The procedure `black-out` takes two lists as its argument. The first list indicates the blackout dates in this format: `((January 1 15) (July 4) (February 14) (November 27 28 29))`. The second list indicates the days you are willing to fly in each month, in this format: `((January (1 2 3 … 31)) (February (1 5 17)) (March (1 2 9 16 … 31)) (April ()) …)` The return value should be a subset of the days you are willing to fly, or in that same format but with blackout dates removed.

```
(black-out '((january (1 9 13 15 18 31)) (february 1 5 17) (march 1 8 17 28
31) (april ()) … (july (1 3 4 18)) (august (4 8 19)) …) '((january 1 15)
(july 4) (february 14))) → ((january (9 13 18 31)) (february 1 5 17) (march 1
8 17 28 31) (april ()) … (july (1 3 18) (august (4 8 19)) …)
```

```
(define (black-out black-list my-list)
  (cond ((null? my-list) '())
          ((null? black-list) my-list)
          ((member (caar my-list)
                   (map car black-list))
           (cons (list (caar my-list)
                       (remove-dates (car my-list)
                                     black-list))
                (black-out black-list (cdr my-list))))
          (else
           (cons (car my-list)
                (black-out black-list (cdr my-list))))))

(define (remove-dates month black-list)
  (if (equal? (car month)
              (caar black-list))
     (filter (lambda (d) (not (member d (cdar black-list)))) (cadr month))
     (remove-dates month (cdr black-list))))
```