CS3 – SP06: FINAL REVIEW SOLUTION

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. USE TAIL RECURSION and NO HELPER FUNCTIONS.

Example of current roster: `((Daei 10) (Mahdavikia 2) (Hashemian 9) (Karimi 8))`

```
(define (add-roster player-names player-numbers current-roster)
   (cond ((null? player-names) current-roster)
         ((assoc (car player-names) current-roster)
          (add-roster (cdr player-names) (cdr player-numbers) current-roster))
         (else (add-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)
   (equal? result-list
           (map (lambda (proc arg)
                  (proc arg))
                proc-list
                arg-list)))
```

Here is another solution:

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

B. Provide a sample call to the procedure you just wrote, and also include the correct return value. The procedure list should contain `car`, `even?`, and `square`.

```
(passed-tests? (list car vowel? square)
               (list '(how are you) 'k 5)
               (list 'how #f 25) )
```

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 and 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-lst)
   (reduce (lambda (new so-far)
              (if (> (cadr so-far) (cadr new))
                  so-far
                  new))
           (map (lambda (class)
                  (list (car class)
                        (reduce + (map grade-conversion
                                       (filter (lambda (grade)
                                                  (not (member? grade '(P
NP))))
                                               (cadr class))))))
                class-lst)))
```

6. `chunking`

Write a procedure that groups together similar adjacent elements. Its argument is a list that contains a mix of numbers, letters, list of numbers, and list of letters, not necessarily in this order. Examples of similar adjacent elements are a number and a list of numbers like `(1 (2 3))`, and a number and a number like `(1 2)`. The return value would be a list of sublist, and each sublist contains elements of the same type. Use HOF to solve this problem, and use NO helper procedures. EX:

`(chunking '(1 a b (c) (2))` → `((1) (a b c) (2))`
`(chunking '((1 2) a b (d e f) 3 4 5 g 6))` → `((1 2) (a b d e f) (3 4 5) (g) (6))`

```
(define (chunking lst)
   (reduce (lambda (new so-far)
              (let ((fixed-new (if (word? new) (list new) new))
                    (fixed-sf (cond ((word? so-far) (list (list so-far)))
                                    ((not (list? (car so-far))) (list so-far))
                                    (else so-far))))
                   (cond ((equal? (number? (caar fixed-sf))
                                  (number? (car fixed-new)))
                          (cons (append fixed-new (car fixed-sf))
                                (cdr fixed-sf)))
                         (else (cons fixed-new fixed-sf)))))
           lst))
```

7. Debug/Evaluate

Part 1.

```
(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?
No.

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

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

C. What does `mystery1` do?
`mystery1` returns the argument as is.

Part 2. For each sublist in the argument, `mystery2` removes the outer list. Look at the sample call to a correct `mystery2`. Note the difference in parentheses between the argument and the return value.

```
(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))))))

(mystery2 '(a ((b) (c d)) f ((g ((h)) i))))
      → (a (b) (c d) f (g (h) i) )
```

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

```
(mystery2 '((a)))

(define (mystery2 lst)
   (cond ((null? lst) '())
         ((word? Lst) lst)  ;;ADD THIS LINE
         ((list? (car lst)) (append …))
         (else (cons (car lst) (mystery2 (cdr lst))))))
```

E. 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)
```