CSSE 304 Assignment 8 Updated for Spring, 2016

Problem 4 is a challenging problem. It is also very important, because later assignments will ask you to write variations of it.

Objectives To practice **Following the Grammar** when writing recursive code.

Same rules as the previous assignments. In particular, mutation is not allowed.
No argument error-checking is required. You may assume that all arguments have the correct form.

You may find *Chez* Scheme's `trace-let` to be helpful for debugging your code; if you use it (or any of the other `trace` syntactic forms), be sure to remove it before submitting to the grading server.

#1 (50 points) These s-list procedures have a lot in common with the s-list procedures that we wrote during our Session 8 class. Recall the extended BNF grammar for s-lists:

```
<s-list>       ::= ( {<s-expression>}* )
<s-expression> ::=  <symbol> | <s-list>
```

(a) `(slist-map proc slist)` applies `proc` to each element of `slist`.

```
(slist-map
   (lambda (x)
     (let ([s (symbol→string x)])
       (string→symbol (string-append s s))))
   '((b (c) d) e ((a)) () e)) ➔  ((bb (cc) dd) ee ((aa)) () ee)
```

(b) `(slist-reverse slist)` reverses `slist` and all of its sublists.

```
(slist-reverse '(a (b c) ( ) (d (e f)))) ➔ (((f e) d) ( ) (c b) a)
```

(c) `(slist-paren-count slist)` counts the number of parentheses required to produce the printed representation of `slist`. You must do this by traversing the structure, not by having Scheme give you a string representation of the list and counting parenthesis characters. You can get this count by looking at `car`s and `cdr`s of `slist`).

```
(slist-paren-count '()) ➔ 2
(slist-paren-count '(a (b c) d)) ➔ 4
(slist-paren-count '(a (b) (c () ((d))))) ➔ 12
```

Note: s-lists are always *proper* lists.

(d) `(slist-depth slist)` finds the maximum nesting-level of parentheses in the printed representation of `slist`. You must do this by traversing the structure, and ***not*** by having Scheme give you a string representation of the list and counting the maximum nesting of parenthesis characters.

```
(slist-depth '()) ➔ 1
(slist-depth '(a b c)) ➔ 1
(slist-depth '(a (b c) d))) ➔ 2
(slist-depth '(a (b (c)) (a b))) ➔ 3
(slist-depth '(((a) (( )) b) (c d) e)) ➔ 4
```

(e) `(slist-symbols-at-depth slist d)` returns a list of the symbols from whose depth is the positive integer `d`. They should appear in the same order in the return value as in the original s-list. This one has the basic pattern of the other s-list procedures, but when writing the solution, I found it easier to use a slight variation on that pattern.

```
(slist-symbols-at-depth '(a (b c) d) 2) ➔ (b c)
(slist-symbols-at-depth '(a (b c) d) 1) ➔ (a d)
(slist-symbols-at-depth '(a (b c) d) 3) ➔ ()
```

#2 (15 points) `(group-by-two ls)` takes a list `ls`. It returns a list of lists: the elements of `ls` in groups of two. If `ls` has an odd number of elements, the last sublist of the return value will have one element.

```
> (group-by-two '())
()
> (group-by-two '(a))
((a))
> (group-by-two '(a b))
((a b))
> (group-by-two '(a b c))
((a b) (c))
> (group-by-two '(a b c d e f g))
((a b) (c d) (e f) (g))
> (group-by-two '(a b c d e f g h))
((a b) (c d) (e f) (g h))
```

#3 (20 points) `(group-by-n ls n)` takes a list `ls` and an integer `n` (you may assume that $n \geq 2$). Returns a list of lists: the elements of `ls` in groups of `n`. If `ls` has a number of elements that is not a multiple of `n`, the length of the last sublist of the return value will be less than n.

```
> (group-by-n '() 3)
()
> (group-by-n '(a b c d e f g) 3)
((a b c) (d e f) (g))
> (group-by-n '(a b c d e f g) 4)
((a b c d) (e f g))
> (group-by-n '(a b c d e f g h) 4)
((a b c d) (e f g h))
> (group-by-n '(a b c d e f g h i j k l m n o) 7)
((a b c d e f g) (h i j k l m n) (o))
> (group-by-n '(a b c d e f g h) 17)
((a b c d e f g h))
> (group-by-n '(a b c d e f g h i j k l m n o p q r s t) 17)
((a b c d e f g h i j k l m n o p q) (r s t))
```

#4 (40 points) On pp 20-22 of EoPL, you should have read about `(subst new old slist)`, which substitutes *new* for each occurrence of symbol *old* in the s-list *slist*. *We also wrote this procedure during Session 9 (in Fall, 2015 term; may happen in a different session in a later term).*

Now write `subst-leftmost`, which takes the same arguments (plus a comparison predicate, described below), but only substitutes *new* for the **leftmost** occurrence of *old*. By "leftmost", I mean the occurrence that would show up first if Scheme printed the list. Another way of saying it is "the one that is encountered first in a preorder traversal". Your procedure must "short-circuit", i.e. avoid traversing the `cdr` of any sublist if the substitution has already been done anywhere in the `car` of that sublist. You should only traverse the parts of the tree that are necessary to find the leftmost occurrence and do the substitution, then copy the references to all of the remaining `cdr`s without traversing the sublists of those `cdr`s. Also, ***you must not traverse the same sublist twice*.**
Hint: if your code calls `equal?` or `contains?` or any other procedure that traverses an entire s-list, you are probably violating the "don't-traverse-twice" rule.

In order to make the procedure slightly more general (and easier for me to test the above constraint), *subst-leftmost* will have an additional argument that *subst* does not have. It is an equality procedure, used to determine whether an individual symbol or number in the list matches *old*.

```
(subst-leftmost 'k 'b '((c d a (e () f b (c b)) (a b)) (b)) eq?) ➔
                      ((c d a (e () f k (c b)) (a b)) (b))
(subst-leftmost 'b 'a '(c (A e) a d)
     (lambda (x y) (string-ci<=? (symbol->string x) (symbol->string y)))) ➔ (c (b e) a d)
```

```
(define subst-leftmost   ; substitute new for leftmost occurrence of old in slist (match defined by equality-pred?)
  (Lambda (new old slist equality-pred?)    ; you fill in the rest.
        ... ))
```

Note: Mutation could possibly be used to do in this problem, but I want you to get a bit more practice on purely functional programming, and this problem will certainly give you that practice! It has a lot in common with `bt-max-interior` from a previous assignment.