CSSE 304 Assignment 9 Updated for Spring, 2016

Objectives To experiment with procedural abstraction.

Same rules as the previous assignments, including the prohibition of mutation.

No argument error-checking is required. You may assume that all arguments have the correct form.

#1 (60 points) `snlist-recur`. I have slightly adapted the book's definition of `s-lists` (EoPL, p 8) to allow numbers as well as symbols in the lists.

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

Define a procedure `snlist-recur` that is similar to `list-recur` (written in class), but it returns procedures that work on `sn-lists`. When we call `snlist-recur` with arguments of the correct types, it returns a procedure that

- takes an `sn-list` as its only argument, and
- traverses the entire `sn-list` and its sub-lists, and does the intended computation.

After you write `snlist-recur`, test it by using it to define the functions in (a)-(f). Each of them will have an `sn-list` as one of its arguments.

When you use `snlist-recur` to produce a procedure `f`**:** in some cases (namely parts b and e), you may need to first write a curried version of `f` (as we did in class when we wrote curried versions of member`?` and `map)` in order to get a procedure that recurs on only one argument (the `sn-list`). Procedures that you pass as arguments to `snlist-recur` **must not be explicitly recursive,** nor may they call `map,` which is a substitute for recursion. All of the recursion on the sn-lists in your solutions should be produced by `snlist-recur` itself. To reiterate, none of your functions in parts (a) - (f) should have any explicit recursive calls.

There are ways to write all of the required procedures without properly using `snlist-recur`. But that would miss the point of this problem. Thus, if you do not use `snlist-recur` as prescribed above, you will not earn any points, even if the grading program says that your code works for all of the test cases.

Note: When your `snlist-recur` procedure is applied to argument(s) of the proper type(s), it must return a procedure that expects exactly one argument (an `sn-list`).

Hint for this problem: `snlist-recur` will probably need to take three arguments (`snlist` refers to an argument passed to the procedure returned by a call to `snlist-recur`):

A base-value to be returned when `snlist` is the empty list.
A procedure to be applied when the `car` of `snlist` is a list (i.e., it is a pair or the empty list).
A procedure to be applied when the `car` of `snlist` is not a list (i.e., it is a symbol or number).

Note: However you design it, when your `snlist-recur` procedure is applied to arguments of the proper types, it must return a procedure that expects exactly one argument (an `sn-list`).

Notes:

1. Your code for these and for the other procedures you write may assume that all of their arguments are the correct type(s). You do not have to do any checking of arguments for validity.
2. Your code for each of the following parts can be fairly short. My longest one is 8 lines long.

(a) `(sn-list-sum snlst)` finds the sum of all of the numbers within `snlst` (which contains no symbols).

```
(sn-list-sum '((2 (3) 4) 5 ((1)) ())) ➔ 15
 sn-list-sum '()) ➔ 0
```

(b) `(sn-list-map proc snlst)` applies `proc` to each element of `snlst`.

```
(sn-list-map (lambda (x) (+ 1 x)) '((2 (3) 4) 5 ((1)) () 5))
  ➔    ((3 (4) 5) 6 ((2)) () 6)
```

(c) `(sn-list-paren-count snlst)` counts the number of parentheses required to produce the printed representation of `snlst`. (You can get this count by looking at `car`s and `cdr`s of `snlst`).

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

```
(sn-list-paren-count '()) ➔ 2
(sn-list-paren-count '(2 (3 4) 5)) ➔ 4
(sn-list-paren-count '(2 (3) (4 () ((5))))) ➔ 12
```

(d) `(sn-list-reverse snlst)` reverses `snlst` and all of its `sublists`.

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

(e) `(sn-list-occur s snlst)` counts how many times the symbol `s` occurs in the sn-list `snlst`)

```
(sn-list-occur 'a  '(() a ((a)) a (a a b a) (a a))) ➔ 8
```

(f) `(sn-list-depth snlst)` finds the maximum nesting-level of parentheses in the printed representation of `snlst`.

```
(sn-list-depth '()) ➔ 1
(sn-list-depth '(1 2 3)) ➔ 1
(sn-list-depth '(1 (2 3) 4)) ➔ 2
(sn-list-depth '(1 (2 (3)) (2 3))) ➔ 3
(sn-list-depth '(((3) (( )) 2) (2 3) 1)) ➔ 4
```

#2 (20 points) Recall the following syntax definition from page 9 of EOPL:

```
<bintree> ::= <number>  | (  <symbol> <bintree> <bintree>  )
```

Write a `bt-recur` procedure, similar to the `list-recur` and `snlist-recur` procedures from class and this homework. Calling `bt-recur` produces a procedure that recurs over all of the elements of a `bintree`.

Then use `bt-recur` to create the following two procedures:

- **`(bt-sum T)`** finds the sum of all of the numbers in the leaves of the `bintree T`.
- **`(bt-inorder T)`** creates a list of the symbols from the *interior* nodes of T, in the order that they would be visited in an inorder traversal of the binary tree.

The following transcript should help your understand what `bt-sum` and `bt-inorder` do. I do not show the code that was used to construct `t1`.

```
> t1
(a (b 1 4) (c (d 2 5) 3))
> (bt-sum t1)
15
> (bt-inorder t1)
(b a d c)
> (define t2 (list 'e 6 t1))
> t2
(e 6 (a (b 1 4) (c (d 2 5) 3)))
> (bt-sum t2)
21
> (bt-inorder t2)
(e b a d c)
```

Note: As in the `snlist-recur` problems from the previous problem, the definitions of `bt-sum` and `bt-inorder` should not contain any explicit recursive calls. All recursion must be produced by `bt-recur`.

(continued on next page)

3. (25 points) Predefined Scheme procedures like `cadr` and `cdadr` are compositions of up to four `car`s and `cdr`s. You are to write a generalization called `make-c...r`, which does the composition of any number of `car`s and `cdr`s. It takes one argument, a string of `a`'s and `d`'s, which are used like the `a`'s and `d`'s in the names of the pre-defined functions. For example, `(make-c..r "addddr")` is equivalent to `(compose car cdr cdr cdr cdr)`.

```
> (define caddddr (makec...r "adddd"))
> (caddddr '(a (b) (c) (d) (e) (f)))
(e)
> ((make-c...r "") '(a b c))
(a b c)
> ((make-c...r "a") '(a b c))
a
> ((make-c...r "ddaddd") '(a b c ((d e f g) h i j)))
(i j)
> ((make-c...r "adddddddddd") '(a b c d e f g h i j k l m))
l
```

I have provided the code for `compose`. For full credit, you should write `make-c...r` in a functional style that only uses built-in procedures, anonymous procedures, and `compose` in the definition of `make-c...r`.

> **Hint:** My solution uses the built-in procedures `map`, `apply`, `string->list`, `list->string`, `string->symbol`, and `eval`; also the character constants `#\c` and `#\r`. You are not required to use these, but you may find them helpful. I do not assume that you are already familiar with all of them; *The Scheme Programming Language* contains info on them; my intention is that you demonstrate an ability to look up and use new procedures. My solution calls `map` multiple times.

```
(define compose
  (case-lambda
   [() (lambda (x) x)]
   [(first . rest)
    (let ([composed-rest (apply compose rest)])
      (lambda (x) (first (composed-rest x))))]))
```

#4 (30 points). S-lists are defined on page 8 of EoPL. You are to write a procedure called `make-slist-leaf-iterator`. This procedure takes an `s-list` as its argument, and returns an iterator procedure that takes no arguments (a.k.a. a *thunk*). Each time the iterator procedure is called, it returns the next symbol from the `s-list`. If the iterator is called again after all of the symbols from the `s-list` have been returned, it returns `#f`. An example should help you to understand what an s-list leaf iterator is supposed to do (things in **bold** are the things that I typed, the others are Scheme's responses):

```
> (define iter (make-slist-leaf-iterator '((a (b c) () d) () e)))
> (iter)
a
> (iter)
b
> (iter)
c
> (iter)
d
> (iter)
e
> (iter)
```

```
#f
> (iter)
#f
```

The iterator procedure must maintain a mutable state if it is to exhibit this behavior. Mutation is allowed for this problem.

One simple approach to creating the iterator would be to simply call `flatten` on the s-list, and then use `cdr` to traverse the resulting flat list. However, this approach has a property that no iterator should have! It requires visiting every symbol in the s-list before the iterator is ever asked to return a symbol. If we create an iterator procedure for an s-list that contains thousands of symbols, but then we call that iterator only a few times, the iterator should not have to deal with all of the symbols in the s-list.
Thus, due to efficiency, you are not allowed to use any such approach that "preprocesses" the entire s-list in order to make the iteration simpler.

The standard way to do tree iterators is to use a stack to keep track of subtrees whose left side we have already visited, but not the right side. In this case, the stack will keep track of `cdr`s of the pairs whose `car`s we have already visited. The idea is similar to the tree iterators presented in chapter 18 of the CSSE 230 book: Mark Allen Weiss, *Data Structures and Problem Solving using Java*. You may want to write mutually-recursive helper procedures, similar in function to Weiss's `first` and `advance` methods.

Should an s-list leaf iterator do a preorder or postorder traversal? It doesn't matter, since we are only iterating the leaves; preorder and postorder visit the leaves in the same order.

Be careful about **empty sublists**. Notice in the example above that the iterator skips them.

In my code, an s-list leaf iterator procedure has only one persistent local variable, whose value is a stack object.

Here is my code for constructing a stack (It's in http://www.rose-hulman.edu/class/csse/csse304/201630/Homework/Assignment_09/stack.ss). Actually, the code in that file has an error which we corrected in the Session 11 class to get the code below.

```
(define make-stack
 (lambda ()
  (let ([stk '()])
   (lambda (msg  . args )
    (case msg    ; Scheme's case is a similar to switch in some other languages.
      [(empty?) (null? stk)]
      [(push)   (set! stk (cons (car args) stk))]
      [(pop)    (let ([top (car stk)])
                  (set! stk (cdr stk))
                  top)]
      [else (errorf 'stack "illegal message to stack object: ~a" msg)])))))
```