**CSSE 304 Assignment 9**

**Objectives** To experiment with procedural abstraction, using it to produce all of the recursion for the example problems.

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

**#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 proceduref**,** 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 procedures in parts   
(a) - (f) should contain any explicit calls to recursive procedures that you write.

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.

**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. As usual, your code for these and for any 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 procedure definition is only 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 and returns the results in an sn-list that

has the same “shape” as snlist. .

(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 cars and cdrs 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. You may not use reverse in your arguments to snlist-recur.

(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.

Previous terms’ Piazza Q&A

Assignment 9: Efficiency for bt-inorder

Does bt-inorder needs to be in O(n) time? I'm currently using an append, which makes it O(n log n).

**Instructor answer:** You are allowed to do it the simple way, using append. The best case is Theta(n log n).  Worst case is Theta(n^2).