CMSC 10500-1: Homework 8

(due on Wednesday July 21st)

This will be the last homework, so there are two sets of problems and they are due on Wednesday (not Monday). The final exam will be held on Friday, from 1.30 to 4.30 pm (in the Mac Lab). You can either come directly there, or come to the class by 1.15 and we can go from there.

Generating all permutations

In class we saw how to write a program which given a list generates all permutations of the same. In this exercise we do the same problem using higher order functions and a different approach.

In class the approach was as follows. For each element x in the list, remove x from the list and generate all permutations of the remaining list. Then add x in front of each generated permutation. Accumulate all these permutation for different x's, and output this as the answer.

The approach we use here is as follows: Remove the first element of the list. Generate all permutations of remaining list. Now insert the first element in each position of each permutation so far computed. This generates all the permutations.

For example consider the list `(list 1 2 3)`. Remove the first element `1`, and generate all permutations of `(list 2 3)`, namely `(list (list 2 3) (list 3 2))`. Now insert `1` into each position of each permutation we have so far. This gives us

```
(list (list 1 2 3) (list 2 1 3) (list 2 3 1)
      (list 1 3 2) (list 3 1 2) (list 3 2 1))
```

In order to do that, you will need the `interval` function from a previous homework. Here is another implementation of the `interval` function using higher order functions.

```
;; interval: number number -> list-of-numbers
;; list of all numbers from low to high (both inclusive)
(define (interval low high)
    (build-list (add1 (- high low)) (lambda (x) (+ low x)))
)
```

`build-list` takes two arguments `n` and `fn` and returns the list `(list (f 0) (f 1) ... (f n))`. In our case $n = high - low + 1$ and $f(x) = low + x$.

Now proceed using the following outline:

1. **(3 pts)** Write a scheme function `insert-at` which consumes three arguments `k s lst`. $1 \leq k$ is a number, `s` is of arbitrary type and `lst` is a list (with arbitrary contents). Your function should produce a list consisting of `lst` with `s` inserted at position `k`. For eg, if `k=1`, then it should be inserted at the beginning, and if `k > (length lst)` then it should be inserted at the end.

2. **(5 pts)** Write a scheme function `ins-every` which consumes two arguments `s` and `lst`. `s` is of any type (its type should not affect your program, but for concreteness you may assume it is a symbol), and `lst` is a list (with arbitrary contents). This function should produce a list of lists obtained by inserting `s` at every position in `lst` (including the first and the last). Do not use recursion.

3. **(5 pts)** Write a scheme function `ins-all-every` which consumes two arguments `s` and `lol`. `s` is as before, and `lol` is a list of lists. This function should produce a list of lists obtained by inserting `s` at every position in all sublists of `lol`. Do not use recursion.

4. **(2 pts)** Finally write a scheme function `permute` which consumes a list and produces a list of all permutations of the given list.

```
> (permute (list 'a 1 2 3))
(list (list 'a 1 2 3) (list 1 'a 2 3) (list 1 2 'a 3) (list 1 2 3 'a)
      (list 'a 2 1 3) (list 2 'a 1 3) (list 2 1 'a 3) (list 2 1 3 'a)
      (list 'a 2 3 1) (list 2 'a 3 1) (list 2 3 'a 1) (list 2 3 1 'a)
      (list 'a 1 3 2) (list 1 'a 3 2) (list 1 3 'a 2) (list 1 3 2 'a)
      (list 'a 3 1 2) (list 3 'a 1 2) (list 3 1 'a 2) (list 3 1 2 'a)
      (list 'a 3 2 1) (list 3 'a 2 1) (list 3 2 'a 1) (list 3 2 1 'a)
)
```

Traversals of a binary tree

5. **(10 pts)** Write a scheme function which consumes two lists `in` and `pre` containing the same symbols/number (in a different order), and produces a binary tree whose in-order traversal is `in` and pre-order traversal is `pre`. You may assume that all the elements in the list are distinct. You can use `(equal?  x y)` which returns `true` if `x` and `y` are the same (works for numbers, boolean, symbols, structs, ...).

   Hint: By looking at the in-order and the pre-order traversal of a binary tree, identify the root, elements in the left and right subtrees and proceed recursively.

```
;; inpre-bt: list-of-X list-of-X -> binary tree
;;   return a binary tree whose inorder traversal and
;;   preorder traversal are as given.
(define (inpre-bt in pre)
    ...)
```

```
> (inpre-bt (list 1 2 3 4 5 6 7) (list 4 2 1 3 6 5 7))
(make-bt
 (make-bt (make-bt empty 1 empty) 2 (make-bt empty 3 empty))
 4
 (make-bt (make-bt empty 5 empty) 6 (make-bt empty 7 empty)))
```

6. **(5 pts)** Repeat the previous problem for in-order and post-order (instead of pre-order). Again assume that all the elements in the list are distinct.

```
;; inpost-bt: list-of-X list-of-X -> binary tree
;;   return a binary tree whose inorder traversal and
;;   postorder traversal are as given.
(define (inpost-bt in pre)
    ...)

> (inpost-bt (list 1 2 3 4 5 6 7) (list 1 3 2 5 7 6 4)
(make-bt
 (make-bt (make-bt empty 1 empty) 2 (make-bt empty 3 empty))
 4
 (make-bt (make-bt empty 5 empty) 6 (make-bt empty 7 empty)))
```