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)
)
```

Solution

```
;; insert-at: number X list-of-X -> list-of-X
;;    insert s at position k>= 1 in given list.
;;    if k > length of list, add at last position
(define (insert-at k s los)
  (cond
    [(empty? los) (cons s empty)]
    [(= k 1) (cons s los)]
    [else (cons (first los) (insert-at (- k 1) s (rest los)))]
  )
)

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

;; ins-every: X list-of-X -> list-of-list-of-X
;;   list of lists obtained by inserting s at every position in los
(define (ins-every1 s los)
  (map (lambda (k) (insert-at k s los)) (interval 1 (add1 (length los)))))

;; another implementation of ins-every
;; using build-list directly
(define (ins-every2 s los)
  (build-list (add1 (length los)) (lambda (k) (insert-at (add1 k) s los))))

(define ins-every ins-every1) ; Choose first implementation

;; ins-all-every: X list-of-list-of-X
;;    insert s at every position in every list in lol
;;    and generate the list of lists
(define (ins-all-every s lol)
  (foldr append empty (map (lambda (lst) (ins-every s lst)) lol)))

;; permute: list -> list-of-list
;;   generate all permutations of the given list
(define (permute los)
  (cond
    [(empty? los) (list empty)]
    [else (ins-all-every (first los) (permute (rest los)))]
  )
)
```

There are two ways to implement `ins-every`. One uses `map` and `interval` and the other uses `build-list` directly.

Note that in `ins-all-every` we are essentially using the `flatmap` of previous exercise.

Traversals of a binary tree

5. **(6 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`. Assume all elements of the tree are distinct.

   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.

6. **(4 pts)** Repeat the previous problem for in-order and post-order (instead of pre-order). Assume all elements of the tree are distinct.

Solutions

Before we get to the scheme code, we need to reason about this problem a little. That will tell us what kind of helper functions, we need to solve the problem.

We start by identifying the root. In a pre-order traversal, the root of the tree is the first element. Having identified the element stored at the root of the tree, we then observe that in the in-order traversal, the root lies in between the left subtree and the right subtree. Thus given the root we can use the in-order traversal to find the elements of the left subtree (those elements of the list which occur before the root in the in-order traversal) and the right subtree (those elements occurring after the root). Not only does this give us the elements of the left and right subtrees, it also gives us the in-order traversals of the left and right subtrees. Having identified the elements which make up the left subtree, we observe that the pre-order traversal, consists of the root, the pre-order traversal of the left subtree followed by the pre-order traversal of right subtree. We also know that all the elements of the tree are distinct. Thus the pre-order traversal of the left subtree will be those elements in pre-order traversal of the main tree, which also occur in the left subtree (which we have already identified). The order in which these elements occur in the pre-order traversal of the main tree, gives us the pre-order traversal of the left subtree. Similarly we get the pre-order traversal of the right subtree.

After having done all this, we have reduced the original problem of finding the main tree (given the in-order and pre-order traversals) to the same problem for its left and right subtrees. Hence by recursion we are done.

It only remains to identify the base cases. In case of the empty list, the tree has to be empty. If the in-order traversal has only one element, then that must be the root (and the only node) of the tree. This completes the first problem.

If we are given the in-order and the post-order traversal, the same analysis above holds. Only difference is that the root is now the last element of the post-order traversal.

From the above analysis, we see that we need functions to do the following:

- Given a list `lst` and an element `elt`, generate a list of elements of `lst` which occur before `elt`.
- Similarly for list of elements which occur after `elt`.
- Finally, given a small list and a large list, we need a function which returns all the elements of the small list in the order in which they are found in the large list.

The complete code is given below:

```
(define-struct bt (left data right))
```

```
;;before: list-of-X X -> list-of-X
;;  all elements of input list before given item
(define (before lst x)
  (cond
    [(empty? lst) empty]
    [(equal? x (first lst)) empty]
    [else (cons (first lst) (before (rest lst) x))]
  )
)

;; after: list-of-X X -> list-of-X
;;  all elements of input list after given item
(define (after lst x)
  (cond
    [(empty? lst) empty]
    [(equal? x (first lst)) (rest lst)]
    [else (after (rest lst) x)]
  )
)

;; sublist: list-of-X list-of-X -> list-of-X
;;    Return all the elements of big found in small
;;    in the order in which they are found in big
(define (sublist small big)
  (local (
      (define (isin elt) ; is elt in the list lst
          (not (empty? (filter (lambda (x) (equal? x elt)) small))))
      )
  (filter isin big)
  )
)

(define (inpre-bt-helper in pre)
  (local (
      (define root (first pre))
      (define inleft (before in root))
      (define inright (after in root))
      (define preleft (sublist inleft pre))
      (define preright (sublist inright pre))
      )
   (make-bt (inpre-bt inleft preleft) root (inpre-bt inright preright))
  )
)

;; Construct the binary tree given
;;   inorder and pre order traversals
(define (inpre-bt in pre)
  (cond
    [(empty? in) empty]
    [(empty? (rest in)) (make-bt empty (first in) empty)]
    [else (inpre-bt-helper in pre)]
  )
)

(define (inpost-bt-helper in post)
  (local (
      (define root (first (reverse post)))
      (define inleft (before in root))
      (define inright (after in root))
      (define postleft (sublist inleft post))
      (define postright (sublist inright post))
    )
    (make-bt (inpost-bt inleft postleft) root (inpost-bt inright postright))
  )
)

;; Construct the binary tree given inorder and pre order traversal
(define (inpost-bt in post)
  (cond
    [(empty? in) empty]
    [(empty? (rest in)) (make-bt empty (first in) empty)]
    [else (inpost-bt-helper in post)]
  )
)
```