Principles of Programming, Fall 2009
Practice 4
Managing Data Structure Using List

Woosuk Lee, Suwon Jang, Sungkeun Cho
Programming Research Lab.@SNU

September 28, 2009

1. In Practice 3-Exercise 1, we defined function `subsets`. For example, if the argument of `subsets` is *(1 2 3)*, then the obtained result will be *(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))*. Now, we want to define function `ordered-subsets` which is similar to `subsets` except that it gives us the sorted subsets of a set. For instance,

```
(ordered-subsets '(1 2 3))
;(() (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3))
```

   Complete the following definition of a procedure `sort` that sorts the set of subsets of a set in ascending order.

```
(define (ordered-subsets s)
    (if (null? s)
        (list null)
        (let ((rest (ordered-subsets (cdr s))))
            (sort (append rest (map (lambda (x) (cons (car s) x))
rest))))))

(define (sort lst) ...)
```

2. (SICP 2.3.3) Consider the following set operators:

`member?`	:	$element \times set \rightarrow bool$
`adjoin`	:	$element \times set \rightarrow set$
`union`	:	$set \times set \rightarrow set$
`intersection`	:	$set \times set \rightarrow set$

`member?` is a predicate that determines whether a given element is a member of a set. `adjoin` takes an object and a set as arguments and returns a set that contains the elements of the original set and also the adjoined element. `union` computes the union of two sets and `intersection` computes the intersection of two sets.

One way to represent a set is as a list of its elements in which no element appears more than once. Another way to speed up our set operations is to change the representation so that the set elements are listed in increasing order.

Define two kind of set operators, using unordered list and ordered list.

3. (SICP ex. 2.36) The procedure `accumulate-n` is similar to `accumulate` except that it takes as its third argument a sequence of sequences, which are all assumed to have the same number of elements. It applies the designated accumulation procedure to combine all the first elements of the sequences, all the second elements of the sequences, and so on, and returns a sequence of the results. For instance, if `s` is a sequence containing four sequences, `((1 2 3) (4 5 6) (7 8 9) (10 11 12))`, then the value of `(accumulate-n + 0 s)` should be the sequence `(22 26 30)`. Fill in the missing expressions in the following definition of `accumulate-n`:

```
(define (accumulate-n op init seqs)
  (if (null? <??>)
      nil
      (cons (accumulate op init <??>)
            (accumulate-n op init <??>))))
```

4. (SICP ex. 2.37) Suppose we represent vectors $v = (v_i)$ as sequences of numbers, and matrices $m = (m_{ij})$ as sequences of vectors (the rows of the matrix). For example, the matrix

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{pmatrix}$$

is represented as the sequence `((1 2 3 4) (4 5 6 6) (6 7 8 9))`. With this representation, we can use sequence operations to concisely express the basic matrix and vector operations. These operations (which are described in any book on matrix algebra) are the following:

`(dot-product` v w`)`	*returns the sum* $\sum_i v_i w_i$
`(matrix-vector` m v`)`	*returns the vector* t*, where* $t_i = \sum_j m_{ij} v_j$
`(matrix-matrix` m n`)`	*returns the matrix* p*, where* $p_{ij} = \sum_k m_{ik} n_{kj}$
`(transpose` m`)`	*returns the matrix* n*, where* $n_{ij} = m_{ji}$

We can define the dot product as

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)))
```

Fill in the missing expressions in the following procedures for computing the other matrix operations. (The procedure `accumulate-n` is defined in practice 5.2.)

```
(define (matrix-vector m v)
  (map <??> m))
(define (transpose m)
  (accumulate-n <??> <??> m))
(define (matrix-matrix m n)
  (let ((cols (transpose n)))
    (map <??> m)))
```