Principles of Programming, Spring 2006
Practice 5

Daejun Park, Heejong Lee
Programming Research Lab.@SNU

April 6, 2006

1. 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? (car seqs))
      nil
      (cons (accumulate op init <??>)
            (accumulate-n op init <??>))))
```

2. 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 5.1.)

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

3. The procedure `prime-sum-pairs` takes a positive integer n, and find all ordered pairs of distinct positive integers i and j, where $1 \leqq j < i \leqq n$, such that $i + j$ is prime. For example, if n is 6, then the pairs are the following:

   ```
   ((2 1 3) (3 2 5) (4 1 5) (4 3 7) (5 2 7) (6 1 7) (6 5 11))
   ```

   Fill in the missing expressions in the following definition:

   ```
   (define (prime-sum-pairs n)
     (map make-pair-sum
          (filter prime-sum?
                  (accumulate
                    append
                    null
                    (map <??>
                         (enumerate-interval 1 (+ n 1)))))))
   ```

4. The “eight-queens puzzle” asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in figure 1. One way to solve the puzzle is to work across the board, placing a queen in each column. Once we have placed $k - 1$ queens, we must place the kth queen in a position where it does not check any of the queens already on the board. We can formulate this approach recursively: Assume that we have already generated the sequence of all possible ways to place $k - 1$ queens in the first $k - 1$ columns of the board. For each of these ways, generate an extended set of positions by placing a queen in each row of

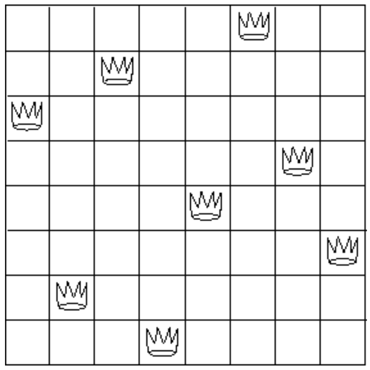

Figure 1: a solution to the eight-queens puzzle

the kth column. Now filter these, keeping only the positions for which the queen in the kth column is safe with respect to the other queens. This produces the sequence of all ways to place k queens in the first k columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle.

We implement this solution as a procedure `queens`, which returns a sequence of all solutions to the problem of placing n queens on an $n \times n$ chessboard. `queens` has an internal procedure `queen-cols` that returns the sequence of all ways to place queens in the first k columns of the board.

```
(define (queens bs)
  (define (queen-cols k)
    (if (< k 0)
        (list empty-b)
        (filter
         (lambda (p) (safe? p))
         (accumulate append
                     null
                     (map <??>
                          (queen-cols (- k 1)))))))
  (queen-cols (- bs 1)))
```

In this procedure `rest-of-queens` is a way to place $k-1$ queens in the first $k-1$ columns, `new-row` is a proposed row in which to place the queen for the kth column, the procedure `adjoin-position` adjoins a new row-column position to a set of positions, and `empty-board` represents an empty set of positions. The procedure `safe?` determines for a set of positions, whether the queen in the kth column is safe with respect to the others. (Note that we need only check whether the new queen is safe – the other queens are already guaranteed safe with respect to each other.) Fill in the missing expressions in the above definition.