Exercise 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

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

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 $n_{ij} = m_{ij}$.

We can define the dot product as[1]

```
(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 exercise 2.36.)

```
(define (matrix-*-vector m v)
  (map <??> m))

(define (transpose mat)
  (accumulate-n <??> <??> mat))

(define (matrix-*-mantrix m n)
  (let ((cols (transpose n)))
    (map <??> m)))
```

Answer.

We learn in linear algebra that multiplying a matrix by a vector yeilds another vector, whose elements are the dot-product of the corresponding row of the matrix and the vector. This idea can be approached by mapping a procedure which computes the dot-product of the corresponding row of the matrix and the specified vector to the matrix

```
(define (matrix-*-vector m v)
  (map (lambda (i)
         (dot-product v i))
       m))
```

Now, consider the problem of transposing a matrix. A usual practice would be by turing the first column of the original matrix into the first row of the transposed matrix, and the second column, and so on. This is in fact equivalent to combining all the first elements of the sequences into a sequence, all the second elements of the sequences, and so on, and returns a sequence of sequences. Notice that we can use `accumulate-n` to depict this strategy by specifying a procedure which compose a sequence and setting its initial value

```
(define (transpose mat)
  (accumulate-n (lambda (x y) (cons x y))
```


1. This definition uses the extended version of `map` described in footnote 12.

```
                nil
                mat))
```

Finally, let's come to the operation of multiplying two matrices. We know mathematically that the product of two matrices M and N is still a matrix, whose entries are given by *dot-product* of the corresponding row of M and the corresponding column of N. This idea can be expressed in a different way in which we first obtian N^T, the transposition of matrix N, then multiply N^T by each row of M to obtain a sequence of sequences, and finally combine these sequences into the resulting matrix. And we do this by mapping a procedure which computes thr product of N^T by a row of M to matrix M

```
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map (lambda (v)
           (matrix-*-vector cols v))
         m)))
```