Tutorial 5

February 16, 2007

1. Write a procedure `norm` that takes a list, which represents a vector, and computes its Eucledean norm. Whenever you have a choice between using HOPs and using recursion, use recursion. You may use built-in `sqrt`, but not built-in `square`. *Example*:

```
 1 ]=> (norm ())
;Value: 0
1 ]=> (norm '(1))
;Value: 1
1 ]=> (norm '(3 4))
;Value: 5
1 ]=> (norm '(1 2 3 -4 -5 -6))
;Value: 9.539392014169456
```

The solution:

```
;; (square x) returns the square of x
;; Pre: x is a number
;; Return: the square of x
(define (square x)
   (* x x))

;; (sum-of-squares lst) returns the sum of the squares
;;  of the numbers in the list lst
;; Pre: lst is a list (flat) of numbers
;; Return: the sum of the squares of the numbers in lst
(define (sum-of-squares lst)
  (if (null? lst)
      0
      (+ (square (car lst)) (sum-of-squares (cdr lst)))))

;; (norm lst) returns a Euclidean norm of a vector,
;;  represented by a list lst
;; Pre: lst is a flat list of numbers
;; Return: a Euclidean norm of a vector, represented by lst
(define (norm lst)
  (sqrt (sum-of-squares lst)))
```

2. Redo the question, only this time you may not use recursion.

```
;; (square x) returns the square of x
;; Args: x - a number, the square if which is returned
;; Pre: x is a number
;; Post: none
;; Return: the square of x
(define (square x)
   (* x x))

;; (norm lst) returns a Euclidean norm of a vector,
;; represented by a list lst
;; Args: lst - a list representation of a vector
;; Pre: lst is a flat list of numbers
;; Post: none
;; Return: a Euclidean norm of a vector, represented by lst
(define (norm lst)
  (sqrt (apply + (map square lst))))
```

3. Redo the question, only this time you may not use recursion and you may not use any helper procedures.

```
;; (norm lst) returns a Euclidean norm of a vector,
;;  represented by a list lst
;; Args: lst - a list representation of a vector
;; Pre: lst is a flat list of numbers
;; Post: none
;; Return: a Euclidean norm of a vector, represented by lst
(define (norm lst)
  (let ((square (lambda (x)
                  (* x x))))
    (sqrt (apply + (map square lst)))))
```

```
MUCH BETTER:

(define (norm lst)
  (sqrt (apply + (map (lambda (x) (* x x)) lst))))
```

We represent a matrix as a list of lists. For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 0 & 1 & 2 \end{bmatrix} \quad \text{is represented by} \quad (\ (1\ 2\ 3\ 4)\ (5\ 6\ 7\ 8)\ (9\ 0\ 1\ 2)\ )$$

1. Write a procedure `add` to perform matrix addition for matrices represented as above.

   ```
   ;; (add matrixA matrixB) returns the sum of matrixA and matrixB
   ;; Pre: matrixA, matrixB - repesented as described above,
   ;;           and have the same number of rows and columns
   ;; Return: the sum of matrixA and matrixB
   (define (add matrixA matrixB)
        (map (lambda (rowA rowB)
               (map + rowA rowB))
             matrixA matrixB))
   ```

2. Write a function `column1` to extract the first column of a matrix.

   ```
   ;; (column1 matrix) returns the first column of matrix
   ;; Pre: matrix - repesented as described above and is non-empty
   ;; Return: the first column of matrix represented as a list
   (define (column1 matrix)
        (map car matrix))
   ```

3. Write a function `columnN` to extract the Nth column of a matrix. (Start counting from 1)

   ```
   ;; (columnN matrix N) returns the Nth column of matrix
   ;; Pre: matrix - repesented as described above and has
   ;;               at least N columns
   ;; Return: the Nth column of matrix represented as a list
   (define (columnN matrix N)
        ( if (= N 1)
             (map car matrix)
             (columnN (map cdr matrix) (- N 1))))
   ```

4. Write a function `sum-Nth-col` to sum the Nth column of a matrix. (Start counting from 1)

```
;; (sum-Nth-col matrix N) return the sum of the nth column of matrix
;; Pre: matrix has an nth column
;; Return: the sum of the numbers in the nth column of matrix
(define (sum-Nth-col matrix N)
  (if (= N 1)
    (apply + (map car matrix))
    (sum-Nth-col (map cdr matrix) (- N 1))))
```

5. Write a procedure `mult` to perform multiplication of a matrix by a scalar.

```
;; (mult c matrix) returns the multiplication of matrix by c
;; Pre: matrix - repesented as described above
;;      c - scalar
;; Return: the multiplication of matrix by c
(define (mult c matrix)
  (map (lambda (row)
         (map (lambda (x) (* c x))
               row))
        matrix))
```

6. Write a procedire `matrix_mult` to perform matrix multiplication.

```
(define (matrix_mult matrixA matrixB)
  (letrec ((num_cols (length (car matrixB)))
           (onerow (lambda (row col)
                     (if (> col num_cols) ()
                         (cons (apply + (map *
                                             row
                                             (columnN matrixB col)))
                               (onerow row (+ col 1))))))
           (mult_row (lambda (row) (onerow row 1))))
    (map mult_row matrixA)))
```