Tutorial 4

Week of October 10, 2005

1 More practice with recursive procedures

1. Develop a procedure that takes a list of numbers and computes the sum of all even numbers in the list.

```
1 ]=> (sum-list-even '(1 2 3 4 5 6))
;Value: 12

1 ]=> (sum-list-even '())
;Value: 0

1 ]=> (sum-list-even '(5 5 5))
;Value: 0


A solution

(define (even? n)
   (= 0 (modulo n 2))
)

;; Note: there is a built-in procedure even?
;; We can use cond or if (cond recommended).

  (define (sum-list-even l)
     (cond ((null? l) 0)
           ((even? (car l)) (+ (car l) (sum-list-even (cdr l))))
           (else (sum-list-even (cdr l)))
    )
  )

  (define (sum-list-even1 l)
    (if (null? l)
            0
           (if (even? (car l)) (+ (car l) (sum-list-even1 (cdr l)))
                               (sum-list-even1 (cdr l)))
    )
  )
```

2. Define a procedure which adds all the numbers in a nested list, on all levels. Assume there are only numbers in the list.

   Distinguish between 2 classes of problems i) where you have a list that you are treating as a flat list and processing as such. This often uses cdr-recursion. ii) the input list is a nested list and you actually have to apply the procedure recursively to each nested, nested, nested... list within the input list. For this you use car-cdr recursion. This problem is an example of car-cdr recursion.

```
1 ]=> (sum-all-levels '(1 2 3))
;Value: 6

1 ]=> (sum-all-levels '(1 (2 3)))
;Value: 6

1 ]=> (sum-all-levels '(() (1) (2 (3))))
;The object (), passed as the first argument to integer-add, is not the corre
type.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify an argument to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
2 error> (restart 1)
;Abort!

1 ]=> (sum-all-levels '((1) (2 (3))))
;Value: 6

A solution:
;;
;; This is a helper procedure discussed in lecture
;;

(define (atom? x)
    (not (pair? x))
)

;;
```

```
;;

(define (sum-all-levels l)
        (cond ((null? l) 0)
              ((atom? (car  l)) (+ (car l) (sum-all-levels (cdr l))))
              (else (+ (sum-all-levels (car l))
                       (sum-all-levels (cdr l))))
        )
)
```

2 Higher Order Procedures

1. MAP

```
    (map <procN> <list1> ... <listN>),

where <procN> is an N-ary procedure
                    (a procedure that takes N arguments)
      <list1> is (e11 e12 ... e1M), a list of M elements
      <list2> is (e21 e22 ... e2M), a list of M elements
       .....
      <listN> is (eN1 eN2 ... eNM), a list of M elements

return the following list:

       (r1 r2 ... rM),

where  r1 = (<procN> e11 e21 ... eN1)
       r2 = (<procN> e12 e22 ... eN2)
         .....
       rM = (<procN> e1M e2M ... eNM)
```

For example,

```
1 ]=>  (map null? '( 1 () 2 (3 4) () 5))
;Value 1: (() #t () () #t ())

1 ]=> (map - '(1 2 3 4) '(1 1 1 1))
;Value 2: (0 1 2 3)

1 ]=> (map cons '(1 2 3 4) '((2 3) (2 3) (2 3) (2 3)))
;Value 4: ((1 2 3) (2 2 3) (3 2 3) (4 2 3))
```

2. APPLY

```
    (apply <procN> <list>),

where <procN> is an N-ary procedure
                    (a procedure that takes N arguments)
  and <list> is (arg1 arg2 ... argN), a list of N elements

return the result of evaluating:

      (<procN> arg1 arg2 ... argN)
```

For example,

```
1 ]=> (apply null? '( () ))
;Value: #t

1 ]=> (apply - '(10 5))
;Value: 5

1 ]=> (apply cons '( a (b c d)))
;Value 5: (a b c d)

1 ]=> (apply + '(1 1 1 1 1 1 1))
;Value: 7

1 ]=> (apply append '( (1) (2) (3 4) (5) () ))
;Value 6: (1 2 3 4 5)
```

3 Using HOPs

1. Write a procedure `norm` that takes a list, which represents a vector, and computes its Eucledean norm. You must use recursion in your solution. You can 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
;; 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))

;; (sum-of-squares lst) returns the sum of the squares
;;  of the numbers in the list lst
;; Args: lst - a list of numbers
;; Pre: lst is a list (flat) of numbers
;; Post: none
;; 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
;; 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 (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)
  (sqrt (apply + (map (lambda (x) (* x x)) lst))))
```

4 More HOPs

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

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

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

```
;; (add matrixA matrixB) returns the sum of matrixA and matrixB
;; Args: matrixA, matrixB - matrices to be added
;; Pre: matrixA, matrixB - repesented as described above,
;;           and have the same number of rows and columns
;; Post: none
;; Return: the sum of matrixA and matrixB
(define (add matrixA matrixB)
     (map (lambda (rowA rowB) (map + rowA rowB)) matrixA matrixB))
```

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

```
;; (column1 matrix) returns the first column of matrix
;; Args: matrix - a matrix
;; Pre: matrix - repesented as described above and is non-empty
;; Post: none
;; Return: the first column of matrix represented as a list
(define (column1 matrix)
     (map car matrix))
```

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

```
;; (columnN matrix N) returns the Nth column of matrix
;; Args: matrix - a matrix, N - a positive number
;; Pre: matrix - repesented as described above and has
;;               at least N columns
;; Post: none
;; 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 procedure `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
(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 c) returns the multiplication
;;  of matrix by c
;; Args: matrix - matrix to be multiplied
;;       c - scalar
;; Pre: matrix - repesented as described above
;;      c - scalar
;; Post: none
;; Return: the multiplication of matrix by c
(define (mult c matrix)
  (map (lambda (row)
         (map (lambda (x) (* c x))
              row))
       matrix))
```