CECS 342 - Lab Assignment 5 - Functional Programming

Due Date: Monday, April 22

Team Members: Bryan Tineo & Maxwell Guillermo

Completion of Lab Assignment:

Both team members contributed equally and collaborated throughout the completion of the lab assignment.

Code:

1. [5 points] Write tail-recursive versions of the following:

```
;; compute integer log, base 2
;; (number of bits in binary representation)
;; works only for positive integers
(define log2
(lambda (n)
(if (= n 1) 0 (+ 1 (log2 (quotient (+ n 1) 2))))))
```

```
#lang scheme

;; Define a function named log2 that takes one argument, n
(define (log2 n)
  ;; Define an inner helper function, log2-iter, which is tail-recursive.
  ;; It takes two arguments: n (the current value being processed)
  ;; and depth (the depth of recursion, i.e., the log base 2 result)
  (define (log2-iter n depth)
    ;; If n is less than or equal to 1, the base case is reached.
    ;; Return the depth, which now contains the log base 2 of the original n.
    (if (<= n 1)
        depth
        ;; Else, recursively call log2-iter with n divided by 2 and increment the depth by 1.
        ;; This is the tail call. Since it's the last operation performed, the function is
tail-recursive.
        (log2-iter (/ n 2) (+ 1 depth))))
```

;; Initiate the recursive process by calling log2-iter with the initial value of n and 0 as the initial depth value.
  (let ((result (log2-iter n 0)))
    (display "The log base 2 of the number is: ")
    (display result) ;; Display the result to the user
    (newline)   ;; Print a newline for clean output
    result))   ;; Return the result as well

;; Call the function with an example number to see the output.
(log2 64) ;; For example, calling log2 with the argument 64

2. [5 points] Write purely functional Scheme functions to return a list containing all elements of a given list that satisfy a given predicate.

   For example, (filter (lambda (x) (< x 5)) ‘(3 9 5 8 2 4 7)) should return (3 2 4).

#lang scheme
;; Define a function named 'filter' that takes a predicate function and a list
(define (filter pred lst)
  ;; Define a helper function that takes two arguments: the list and an accumulator for the result
  (define (filter-acc pred lst depth)
    (cond ((null? lst)   ;; If the list is empty, return the accumulated list
           (reverse depth))
          ((pred (car lst))   ;; If the predicate is true for the first element,
           (filter-acc pred   ;; Recursively call with the rest of the list

                    ;; If the predicate is true for the first element, recursively call filter-acc
                    ;; with the rest of the list (cdr lst) and add the first element to the accumulator (cons (car lst) acc).

                    (cdr lst) ;; Retrieves the "rest" of the list lst, that is, all the elements except for the first one.
                    (cons (car lst) depth)))
          (else   ;; If the predicate is false,
           (filter-acc pred   ;; Recursively call with the rest of the list

```
            (cdr lst) ;; without adding the element to the accumulator
            depth))))
  ;; Call the helper function with the initial list and an empty list as the accumulator
  (filter-acc pred lst '()))

;; Example usage of the filter function
(filter (lambda (x) (< x 5)) '(3 9 5 8 2 4 7))
```

3. [5 points] Write purely functional Scheme functions to return all rotations of a given list. For example, (rotate ‘(a b c d e)) should return ((a b c d e) (b c d e a) (c d e a b) (d e a b c) (e a b c d)) (in some order).

```
#lang scheme
;; Define the main rotation function that takes a list as an argument.
(define (rotate lst)
  ;; Define a helper function that will carry out the rotations.
  ;; It takes the current list (lst) to rotate and a counter (count) to track the number of
rotations done.
  (define (rotate-helper lst count)
    ;; Check if the count equals the length of the list. This is our base case,
    ;; signifying that all possible rotations have been done since a list of n elements has n
rotations.
    (if (= count (length lst))
      '()  ; Return an empty list, signifying no more rotations need to be added.
      ;; Otherwise, construct the list of rotations by:
      ;; 1. cons-ing the current list (lst) to the front of the list of rotations we are
accumulating.
      ;; 2. Calling rotate-helper recursively to compute the next rotation.
      (cons lst  ; Add the current list (lst) as the latest rotation to the accumulator.
        (rotate-helper (append (cdr lst)  ; Create a new list that is the original list without
the first element...
                         (list (car lst)))  ; ...plus the first element of the original list
appended to the end.
                       (+ count 1)))))  ; Increment the count because we have completed
another rotation.
```

```scheme
  ;; Begin the rotation process by calling rotate-helper with the original list (lst) and an initial count of 0.
  (rotate-helper lst 0))

;; Example usage of the rotate function
;; This call should produce a list containing all rotations of the input list: '(a b c d e)
(rotate '(a b c d e))  ;
```

4. [5 points] Write a Scheme function called reverse to reverse a list passing to the function as an argument. In this problem, you are allowed to use only cond, append, cdr, car, and cons.

   Use a display function to display the reverse list.
   (display(reverse '(A B B C D D E F G G))) should output the following:
   G G F E D D C B B A
   You can use your input.

```scheme
#lang scheme

(define (reverse lst)
  ;; Check if the list is empty. If it is, return the empty list.
  ;; This serves as the base case for our recursion.
  (if (null? lst)
      '()
      ;; If the list is not empty, recursively call `reverse` on the cdr of the list (the sublist excluding the first element).
      ;; Then append the result of that recursive call to a list containing just the first element (car lst).
      ;; This step effectively puts the first element at the end of the reversed sublist.
      (append (reverse (cdr lst)) (list (car lst)))))

;; The `display` function is used to show the reversed list in the console.
;; The `reverse` function is called with the list `(A B C D E F G)` and the result is displayed.
(display (reverse '(A B C D E F G)))
```

Output:

Question 1 Output:

```scheme
#lang scheme

;; Define a function named log2 that takes one argument, n
(define (log2 n)
  ;; Define an inner helper function, log2-iter, which is tail-recursive.
  ;; It takes two arguments: n (the current value being processed)
  ;; and depth (the depth of recursion, i.e., the log base 2 result)
  (define (log2-iter n depth)
    ;; If n is less than or equal to 1, the base case is reached.
    ;; Return the depth, which now contains the log base 2 of the original n.
    (if (<= n 1)
        depth
        ;; Else, recursively call log2-iter with n divided by 2 and increment the depth by 1.
        ;; This is the tail call. Since it's the last operation performed, the function is tail-recursive.
        (log2-iter (/ n 2) (+ 1 depth))))
  ;; Initiate the recursive process by calling log2-iter with the initial value of n and 0 as the initial depth
  (let ((result (log2-iter n 0)))
    (display "The log base 2 of the number is: ")
    (display result)  ;; Display the result to the user
    (newline)         ;; Print a newline for clean output
    result))          ;; Return the result as well

;; Call the function with an example number to see the output.
(log2 64)  ;; For example, calling log2 with the argument 64
```

```
Welcome to DrRacket, version 8.12 [cs].
Language: scheme, with debugging; memory limit: 128 MB.
The log base 2 of the number is: 6
6
>
```

Question 2 Output:

```scheme
#lang scheme
;; Define a function named 'filter' that takes a predicate function and a list
(define (filter pred lst)
  ;; Define a helper function that takes two arguments: the list and an accumulator for the result
  (define (filter-acc pred lst depth)
    (cond ((null? lst)            ;; If the list is empty, return the accumulated list
           (reverse depth))
          ((pred (car lst))       ;; If the predicate is true for the first element,
           (filter-acc pred       ;; Recursively call with the rest of the list

                      ;; If the predicate is true for the first element, recursively call filter-acc
                      ;; with the rest of the list (cdr lst) and add the first element to the accumulator (cc

                      (cdr lst) ;; Retrieves the "rest" of the list lst, that is, all the elements except for
                      (cons (car lst) depth)))
          (else                   ;; If the predicate is false,
           (filter-acc pred       ;; Recursively call with the rest of the list
                      (cdr lst) ;; without adding the element to the accumulator
                      depth))))
  ;; Call the helper function with the initial list and an empty list as the accumulator
  (filter-acc pred lst '()))

;; Example usage of the filter function
(filter (lambda (x) (< x 5)) '(3 9 5 8 2 4 7))
```

```
Welcome to DrRacket, version 8.12 [cs].
Language: scheme, with debugging; memory limit: 128 MB.
(3 2 4)
>
```

Question 3 Output:

```scheme
#lang scheme
;; Define the main rotation function that takes a list as an argument.
(define (rotate lst)
  ;; Define a helper function that will carry out the rotations.
  ;; It takes the current list (lst) to rotate and a counter (count) to track the number of rotations done.
  (define (rotate-helper lst count)
    ;; Check if the count equals the length of the list. This is our base case,
    ;; signifying that all possible rotations have been done since a list of n elements has n rotations.
    (if (= count (length lst))
        '()  ; Return an empty list, signifying no more rotations need to be added.
        ;; Otherwise, construct the list of rotations by:
        ;; 1. cons-ing the current list (lst) to the front of the list of rotations we are accumulating.
        ;; 2. Calling rotate-helper recursively to compute the next rotation.
        (cons lst  ; Add the current list (lst) as the latest rotation to the accumulator.
              (rotate-helper (append (cdr lst)  ; Create a new list that is the original list without the first
                                     (list (car lst)))  ; ...plus the first element of the original list appenc
                             (+ count 1)))))  ; Increment the count because we have completed another rotatior
  ;; Begin the rotation process by calling rotate-helper with the original list (lst) and an initial count of 0
  (rotate-helper lst 0))

;; Example usage of the rotate function
;; This call should produce a list containing all rotations of the input list: '(a b c d e)
(rotate '(a b c d e))  ;
```

```
Welcome to DrRacket, version 8.12 [cs].
Language: scheme, with debugging; memory limit: 128 MB.
((a b c d e) (b c d e a) (c d e a b) (d e a b c) (e a b c d))
> 
```

Question 4 Output:

```scheme
#lang scheme

(define (reverse lst)
  ;; Check if the list is empty. If it is, return the empty list.
  ;; This serves as the base case for our recursion.
  (if (null? lst)
      '()
      ;; If the list is not empty, recursively call `reverse` on the cdr of the list (the sublist excluding the
      ;; Then append the result of that recursive call to a list containing just the first element (car lst).
      ;; This step effectively puts the first element at the end of the reversed sublist.
      (append (reverse (cdr lst)) (list (car lst)))))

;; The `display` function is used to show the reversed list in the console.
;; The `reverse` function is called with the list `(A B C D E F G)` and the result is displayed.
(display (reverse '(A B C D E F G)))
```

```
Welcome to DrRacket, version 8.12 [cs].
Language: scheme, with debugging; memory limit: 128 MB.
(G F E D C B A)
> 
```