University of Toronto
CSC324 –Programming Languages, winter 2011

Assignment # 2

Details
- Topics: Scheme
- Weight: 15%
- Due Date: this assignment is due on Feb 28th, 2011 @ 11:59pm
- This assignment is to be done individually.

Marking
- Your code will be marked for correctness (80%) and code documentation (20%).
-Since we plan to test your code using an auto-marker, you must use the exact function names/arguments as specified and use the same file names and folder name as specified.
- You will lose 25% of the total mark if you use wrong file names or function names.
- You will lose 10% of the total mark if your submission fails to load.

Submission
Place all your files in a folder named after your UTORID and submit that folder zipped through the portal (if your UTORID is abcd, the file should be named abcd.zip). A link for submission is provided in the assignments folder in the portal (http://portal.utoronto.ca)

Resources

The Scheme Programming Language:	http://www.scheme.com/tspl2d/index.html
Google Scheme groups:	http://groups.google.com/group/comp.lang.scheme/
Scheme Interperter:	http://www.racket-lang.org/

Description
This assignment is to get you used to Scheme syntax, the programming environment and most importantly thinking as a functional language programmer. The assignment requires you to write a series of functions. Even the hardest ones are very short; the difficulty lies in turning your brain around to think about the solution in the right way!

- In order to help you with this new way of thinking, I am disallowing the use of any of Scheme's imperative features: do, begin and all imperative functions ending with ! (e.g. set!, set-car!, vector-set!…).
- You are also not allowed to use the high order functions: map and reduce
- You may use the following functions and special forms: and (and other logical predicate), append, atom, car, cdr(etc..), cond, cons, define, eq (and other equality predicates), length, list, listp, null, numberp, or, display, > (and other comparison predicates),+,-,*,/ This list is incomplete, however, you can probably solve the entire assignment using only the functions and special forms listed above. If there is something you'd like to use and that is not on the list, ask through discussion board.

(a) [5 marks] Write a function **apply-proc** which takes as arguments a unary function and a list, and returns a list of values obtained by applying the function to every element in the list, in order. You can assume that the input function is applicable to every element in the input list. *Sample invocations:*

```
]=> (apply-proc positive? '(1 2 5 -9))
(#t #t #t #f)
]=> (apply-proc length '( () (1) (1 2)  (1 (2 3)) ) )
(0 1 2 2)
```

Write your answer in a file called apply-proc.scm

Sample Solution:

```
;; (apply-func func lst) returns a list of values obtained by applying func to every
;; element in lst, in order.
;; Parameters: func  -- a unary function; lst      -- a list
;; Preconditions: func is applicable to every element of lst
;; Postconditions: none
;; Return values: a list of values obtained by applying func to every element of
;;                    lst, in order
(define apply-func
   (lambda (func  lst)
      (if (null? lst)
          ()
          (cons (func (car lst))
                (apply-func func (cdr lst))))))
```

(b) [5 marks] Use **apply-proc** to write a function **squareList** that takes as an argument a list of numbers and returns a list of squares of the numbers in the input list, in order.

Write your answer in a file called squareList.scm

Sample Solution:

```
;; (squareList lst) returns a list of squares of the numbers in lst, in order.
;; Parameters: lst  -- a list of numbers
;; Preconditions: apply-func is defined as above
;; Postconditions: none
;; Return values: a list of squares of the numbers in lst, in order
 (define squareList
   (lambda (lst)
     (apply-func (lambda (x) (* x x)) lst)))
```

(c) [10 marks] Write a function **min-max** that takes a non-empty list of numbers as an argument and returns a list of two elements: the largest number that appears in the input list and the smallest number that appears in the input list. *Sample invocation:*

```
]=> (min-max '(-10 0 3 -4))
(-10 3)
```

Write your answer in a file called min-max.scm

Sample Solution:

```
;; (min-max lst) returns a list of two elements: the largest number that appears in
;; lst and the smallest number that appears in lst
;; Parameters: lst  -- a list of numbers
;; Preconditions: lst is not empty
;; Postconditions: none
;; Return values: a list of two elements: the largest number that appears in lst
;;                  and the smallest number that appears in lst
(define min-max
   (lambda (lst)
      (if (null? lst)
          ()
          (list (listmax lst 0) (listmin lst 0)))))


;; (listmin lst n) returns min (the minimum element in lst, n)
;; Parameters: lst  -- a list of numbers
;;                n   -- a number
;; Preconditions: none
;; Postconditions: none
;; Return values: the minimum of n and the minimum element in lst
(define listmin
   (lambda (lst n)
      (if (null? lst)
          n
          (listmin (cdr lst) (min n (car lst))))))

;; (listmax lst n) returns max (maximum element in lst, n)
;; Parameters: lst  -- a list of numbers
;;                n   -- a number
;; Preconditions: none
;; Postconditions: none
;; Return values: the maximum of n and the maximum element in lst
(define listmax
   (lambda (lst n)
      (if (null? lst)
          n
```

(listmax (cdr lst) (max n (car lst)))))

(d) **[10 marks]** Write a function **intersect** that computes the intersection of two lists. In other words, given two lists as arguments, it returns a list of elements contained in both lists. Note that you cannot use the built-in function member. *Sample invocations:*

```
]=> (intersect '(1 2 3 4) '(10 2 4 100) )
    (2 4)
]=> (intersect '(john david) '(david 2 sky 4) )
    (david)
```

Write your answer in a file called intersect.scm

Sample Solution:

```
;; (intersect lst1 lst2) returns a list of elements contained in both lst1 and lst2
;; Parameters: lst1 and lst2 are lists
;; Preconditions: none
;; Postconditions: none
;; Return values: a list of elements contained both in lst1 and lst2
(define intersect
   (lambda (lst1 lst2)
      (cond ((null? lst1) ())
            ((contains? lst2 (car lst1)) (cons (car lst1)
                                               (intersect (cdr lst1) lst2))
            (else ((intersect (cdr lst1) lst2))))))


;; (contains? lst element) tests for membership of element in lst
;; Parameters: element – and arbitrary element to be tested for membership
in lst
;;             lst – a list
;; Preconditions: none
;; Postconditions: none
;; Return values: #t , if element is contained in lst, and (), otherwise
(define contains?
   (lambda (lst element)
      (cond ((null? lst) ())
            ((equal? element (car lst))  #t)
            (else (contains? element (cdr lst))))))
```

(e) **[10 marks]** Define a function **select-even** that takes a list of numbers as input and returns a list that contains all even numbers from the input list, in the same order as they appear in the input list. You can use built-in predicate **even?** *Sample invocation:*

```
]=> (select-even '(1 2 3 4))
(2 4)
```

Write your answer in a file called select-even.scm

Sample Solution:

```
;; (select-even lst) returns a list of all even numbers from lst, in the same order as
;;                   they appear in lst
;; Parameters: lst  - a list of numbers
;; Preconditions: none
;; Postconditions: none
;; Return values: a list of all even numbers from lst, in the same order as they
;;                appear in lst
(define select-even
   (lambda (lst)
      (cond ((null? lst) ())
            ((even? (car last)) (cons (car lst) (select-even (cdr lst))))
            (else (select-even (cdr lst))))))
```

(f) **[20 marks]** Define a function **mult-num** that takes a list as input and returns the product of all numbers in the list. List elements that are not numbers should be ignored. If the input contains no numbers, 1 should be returned. *Sample invocations:*

```
]=> (mult-num '(1 2 3 f 4))
24
]=> (mult-num '(14 () (100) 2.5))
35
```

Write your answer in a file called mult-num.scm

Sample Solution:

```
;; (mult-num lst) returns the product of all numbers in lst
;; Parameters: lst  - a list (which may contain elements other than numbers)
;; Preconditions: none
;; Postconditions: none
;; Return values: if lst contains numbers, then their product
;;                otherwise, 1
(define mult-num
   (lambda (lst)
      (cond  ((null? lst) 1)
             ((number? (car lst)) (* (car lst) (mult-num (cdr lst))))
             (else (mult-num (cdr lst))))))
```

(g) **[20 marks]** Define a function **select** that takes a unary predicate and a list as input and returns a list that contains all elements from the input list, of which the predicate is true, in the same order as they appear in the input list. You can assume that the input predicate is applicable to every element of the input list. *Sample invocation;*

```
]=> (select null? '( () (1 2 3) ))
( () )
```

Write your answer in a file called select.scm

Sample solution:

```
;; (select pred lst) returns a list of elements from lst, of which pred holds
;; Parameters: lst – a list; pred – a unary predicate
;; Preconditions: pred is applicable to every element of lst
;; Postconditions: none
;; Return values: a list of elements from lst, of which pred holds, in the same
;;                order, in which they appear in lst
(define select
   (lambda (pred lst)
      ( cond ((null? lst) lst)
             ((pred (car lst)) (cons (car lst) (select pred (cdr lst))))
             (else (select pred (cdr lst))))))
```