Expressions in *SCHEME*

Write a SCHEME expression to evaluate:

1. the sum of 4, 8, 15, 16, 23 and 42

   Solution:

   ```
   (+ 4 8 15 16 23 42)
   ```

2. the product of $653,854,321$ and $241,304,201$

   Solution:

   ```
   (* 633854321 241304201)
   ```

3. $\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}$

   Solution:

   ```
   (/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))
   ```

Functions

1. Simple Functions

   (a) Define a SCHEME function which takes a number as a parameter and returns the absolute value of that number.

   Solution:

   ```
   (define (absolute x) (sqrt (* x x)))
   ```

   (b) Define a function to convert a temperature from Fahrenheit to Celsius, and another to convert in the other direction. The two formulas are $F = \frac{9}{5}C + 32$ and $C = \frac{5}{9}(F - 32)$.

   Solution:

   ```
   (define (fahr-to-cels f) (* (- f 32) (/ 5 9)))
   (define (cels-to-fahr c) (+ (* (/ 9 5) c) 32))
   ```

(c) Define a SCHEME function named `discount` which takes a product's price and discount (as a percent) and returns the discounted price of the product.

Solution:

```
(define (discount price percent)
  (* price (- 1 percent)))
```

(d) Write a procedure to compute the tip you should leave at a restaurant. It should take the total bill as its argument and return the amount of the tip. It should tip by 15%, but it should know to round up so that the total amount of money you leave (tip plus original bill) is a whole number of dollars. (Use the `ceiling` procedure to round up.)

Solution:

```
(define (tip bill) (- (ceiling (* 1.15 bill)) bill))
```

(e) Most latex paints cover approx. 400 sq. ft. of area. Stains and varnishes cover 500 sq.ft. of area. Write a SCHEME function that takes the length and width (or height) of an area to be covered as well as if the area is to be painted or stained (a boolean parameter would be useful here) and returns the number of gallons required to finish one coat.

Solution:

```
(define (num-gallons l w p?)
  (let ((area (* l w)))
    (if p?
        (ceiling (/ area 400))
        (ceiling (/ area 500)))))
```

2. Recursion

(a) (Towers of Hanoi) The Towers of Hanoi is a famous puzzle with a recursive solution. There are many versions of the legend. One version of the description tells of a temple where the monks spend their days transferring large disks of different sizes between three pegs (each forming a tower). Disks can only be moved according to the following rules:

1. Can only move one disk at a time
2. Disks must be properly stacked, disks may only be placed on larger disks
3. All disks but one must be on one of the three pegs

The recursive solution to this problem labels the three pegs as the source, destination and temporary pegs. The recursive decomposition follows the following form; a stack of n disks on the source peg can be moved to the destination peg by moving the top $n-1$ disks to the temporary peg, moving the largest disk to the destination peg and then moving the $n-1$ smallest disks to the destination peg. This gives us the following algorithm for a function that takes four parameters. The pegs labeled *source*, *temp* and *dest* as well as the number of disks to be moved.

```
(define (towers-of-hanoi n source temp dest)
```

1. `(towers-of-hanoi n-1 source dest temp)`
2. `move disk n from source to dest`
3. `(towers-of-hanoi n-1 temp source dest))`

Define a SCHEME function that lists steps one can follow to solve the Towers of Hanoi puzzle. You can use the SCHEME function `display` to print a message indicating which disk to move and which pegs to move it from and to (step (b) in the algorithm above). Parameters representing pegs can accept integers identifying each of the pegs (e.g. 1 2 and 3).

Solution:

```
(define (towers-of-hanoi n origin dest temp)
  (cond ((not (= n 0))
         (begin (towers-of-hanoi (- n 1) origin temp dest)
                (display "Move␣disk␣")
                (display n)
                (display "␣to␣peg␣")
                (display dest)
                (display #\newline)
                (towers-of-hanoi (- n 1) temp dest origin)))))
(towers-of-hanoi 4 'A 'B 'C)
```

3. Tail Recursion

   (a) Write a SCHEME function that takes two integers as parameters and returns their greatest common divisor (GCD) using Euclid's Algorithm. From Wikipedia: "The GCD of two numbers is the largest number that divides both of them without leaving a remainder. The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the smaller number is subtracted from the larger number. For example, 21 is the GCD of 252 and 105 ($252 = 21 \times 12; 105 = 21 \times 5$); since 252 - 105 = 147, the GCD of 147 and 105 is also 21. Since the larger of the two numbers is reduced, repeating this process gives successively smaller numbers until one of them is zero. When that occurs, the GCD is the remaining nonzero number."

   Solution:

```
(define (GCD a b)
  (cond ((= a 0) b)
        ((= b 0) a)
        ((< a b) (GCD a (- b a)))
        (else (GCD (- a b) b))))
```

4. Higher-Order Functions

   (a) (SICP Exercise 1.43) If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(...(f(x))...))$. For example, if f is the function $x \mapsto x+1$, then the n^{th} repeated application of f is the function $x \mapsto x+n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the $2^{n^{th}}$ power. Write a procedure that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

   Hint: You may find it convenient to use compose from exercise 1.42.

   Solution:

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (repeated f n)
  (if (= n 1)
      f
      (compose f (repeated f (- n 1)))))
```

   (b) (SICP Exercise 1.44) The idea of smoothing a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of $f(x-dx)$, $f(x)$, and $f(x+dx)$. Write a procedure smooth that takes as input a procedure that computes f and returns a procedure that computes the smoothed f.

   Solution:

```
(define (smooth f dx)
  (lambda (x) (/ (+ (f (- x dx))
                    (f x)
                    (f (+ x dx)))
                 3)))
```

LISt Processing

5. Pairs

(a) Define a function that takes two pairs, representing two points, as parameters and returns a pair consisting of the slope and y-intercept of the line intersecting those two points.

Solution:

```
(define (line-equation a b)
  (let ((m (/ (- (cdr a) (cdr b)) (- (car a) (car b)))))
    (cons m
          (- (cdr a) (* m (car a))))))
```

(b) (Unusual Canceling) The fraction $\frac{64}{16}$ exhibits the unusual property that its reduced value of 4 may be obtained by "canceling" the 6 in the numerator with the 6 in the denominator. Define a SCHEME function which takes a fraction represented as a pair and returns a boolean value indicating whether the value of the fraction remains unchanged after this unusual canceling.

If xy is the two digit number (e.g. if $n = 64$ then $x = 6$ and $y = 4$) then

1. $n = 10x + y$
2. $x =$ `(floor (/ n 10))`
3. $b =$ `(modulus n 10)`

Solution:

```
(define (unusual-canceling num denom)
  (let ((x-num (floor (/ num 10)))
        (y-num (modulo num 10))
        (x-denom (floor (/ denom 10)))
        (y-denom (modulo denom 10)))

    (if (= x-num y-denom)
        (= (/ num denom) (/ y-num x-denom))
        #f)))
```

6. Lists

(a) Write a function that computes the largest distance between any two adjacent elements of a list.

Solution:

```
(define (max-pair-dist lst)
  (define (max-pair-dist-aux lst max)
    (if (null? (cdr lst))
        max
        (if (> (absolute (- (car lst) (cadr lst))) max)
              (max-pair-dist-aux (cdr lst)
                                 (absolute (- (car lst)
                                              (cadr lst))))
              (max-pair-dist-aux (cdr lst) max))))
  (max-pair-dist-aux lst 0))
```

(b) Write a function that, given a list $(a_1 \ldots a_k)$, returns the list $(b_1 \ldots b_{k-1})$ so that $b_i = a_{i+1} - a_i$.

Solution:

```
(define (difference-list lst)
  (if (null? (cdr lst))
      (list)
      (cons (- (cadr lst) (car lst)) (difference-list (cdr lst)))))
```

(c) Write a function which takes a list of numbers as parameters and returns a pair representing the range of the numbers in the list. That is, the first in the pair returned should be the minimum element of the list and the second should be the maximum.

Solution:

```
(define (list-range lst)
  (define (list-range-aux lst min max)
    (if (null? lst)
        (cons min max)
        (let ((new-min (if (< min (car lst)) min (car lst)))
              (new-max (if (> max (car lst)) max (car lst))))
          (list-range-aux (cdr lst) new-min new-max))))
  (list-range-aux (cdr lst) (car lst) (car lst)))
```

(d) Define a SCHEME function which uses the solution to the Pairs question b to produce a list of pairs representing all of the fractions which exhibit the unusual canceling property. Note: you may limit yourself to fractions which have only two digits in their numerators and denominators.

Solution:

```
(define (list-unusual-canceling)
  (define (list-unusual-canceling-aux num denom)
    (cond ((> num 99) (list))
          ((> denom 99)
           (list-unusual-canceling-aux (+ num 1) 10))
          ((unusual-canceling num denom)
           (cons (cons num denom)
                 (list-unusual-canceling-aux num (+ denom 1))))
          (else (list-unusual-canceling-aux num (+ denom 1)))))
  (list-unusual-canceling-aux 10 10))
```

7. Trees

```
(define (maketree v left-tree right-tree)
   (list v left-tree right-tree))

(define (value T) (car T))
(define (left T)  (cadr T))
(define (right T) (caddr T))

(define (insert x T)
  (cond ((null? T) (make-tree x '() '()))
        ((eq? x (value T)) T)
        ((< x (value T)) (make-tree (value T)
                                    (insert x (left T))
                                    (right T)))
        ((> x (value T)) (make-tree (value T)
                                    (left T)
                                    (insert x (right T))))))
```

(a) Define a SCHEME function named `count-one-child` which returns the number of internal nodes of a binary search tree which have exactly one child.

Solution:

```
(define (count-one-child tree)
  (let ((left-child (left tree))
        (right-child (right tree)))
    (cond ((and (null? left-child)
                (null? right-child)) 0)
          ((and (not (null? left-child))
                (not (null? right-child)))
           (+ (count-one-child left-child)
              (count-one-child right-child)))
          ((null? left-child)
           (+ 1 (count-one-child right-child)))
          (else (+ 1 (count-one-child left-child))))))
```

(b) Define a SCHEME function named count which given a binary search tree and a value, returns the number of occurrences of the value in the tree.

Solution:

```
(define (occurances-in-tree T val)
   (if (null? T) 0
         (if (= (value T) val)
             (+ 1
                (occurances-in-tree (left T) val)
                (occurances-in-tree (right T) val))

             (+ (occurances-in-tree (left T) val)
                (occurances-in-tree (right T) val)))))
```

(c) Building a binary search tree with inputs in random order.
Define a SCHEME function that, given a vector of distinct numbers in sorted order, builds a binary search tree by inserting the elements of the vector in random order. Do not insert an element more than once.

Solution:

```
#lang racket
;random insert into tree from sorted vector
(define (random-order-insert v t)
  (define (ro-aux v nused)
    (if (= nused (vector-length v)) "done"
        (begin (let* ((raddress (random (vector-length v)))
                      (trial (vector-ref v raddress)))
                 (cond ((not(number? trial))  (ro-aux v nused))
                       (#t (begin
                              ((t 'insert) trial)
                             (vector-set! v raddress "Nan")
                             (ro-aux v (+ 1 nused)))))))))
  (ro-aux v 0))


(define ordered (make-vector 12))
(vector-set! ordered 0 1)
(vector-set! ordered 1 2)
(vector-set! ordered 2 3)
(vector-set! ordered 3 4)
(vector-set! ordered 4 5)
(vector-set! ordered 5 6)
(vector-set! ordered 6 7)
(vector-set! ordered 7 8)
(vector-set! ordered 8 9)
```

```
(vector-set! ordered 9 10)
(vector-set! ordered 10 11)
(vector-set! ordered 11 12)
ordered
(define (make-heap sz smaller?)
  (let ((heap (make-vector sz '()))
        (capacity sz)
        (occupied-size 0)
        (ofs 1))
    (define (resize-vector v)
      (let* ((old-length  (vector-length v))
             (new-length (* 2 old-length))
             (result (make-vector new-length)))
        (begin
          (do ((i 0 (+ i 1)))
            ((>= i old-length) (set! heap result))
            (vector-set! result i (vector-ref v i))))))
    (define (heap-insert h k v)
      (define (heap-restore h k)
        (begin
          (cond ((< k 2) "done")
                ((smaller? (vector-ref h (floor (/ k 2)))
                           (vector-ref h k)) "done")
                (#t (let ((temp (vector-ref h k))
                          (parent (floor (/ k 2))))
                      (begin
                        (vector-set! h k (vector-ref h parent))
                        (vector-set! h parent temp)
                        (heap-restore h parent)))))))
      (let ((return (+ 1 k)))
        (begin
          (vector-set! h k v)
          (if (>= return capacity)
              (begin (resize-vector heap)
                     (set! capacity (* 2 capacity)))
              "not␣yet")
          (heap-restore h k)
          ;(display return)
          return)))
    (define (repair-heap h k)
      (define (rh-aux h k p); p is where the sinking value is now
        (let* ((lc_addr (* 2 p))
               (rc_addr (+ lc_addr 1))
               (temp (vector-ref h p))
               )
          (cond ((< k lc_addr) (display "nothing␣left␣to␣do"))
                ((= k lc_addr) (display "nothing␣left␣to␣do"))
                ((and (= k (+ 1 lc_addr))
                      (smaller? (vector-ref h lc_addr) temp))
                 (begin
                   (vector-set! h p (vector-ref h lc_addr))
                   (vector-set! h lc_addr temp)))
                ((and (smaller? temp (vector-ref h lc_addr) )
                      (smaller? temp (vector-ref h rc_addr) ))
                 (display "nothing␣left␣to␣do"))
                (#t (let ((sub-heap
                           (if (smaller? (vector-ref h lc_addr)
                                          (vector-ref h rc_addr))
                                lc_addr rc_addr)))
                      (begin
                        (display "sinking")
                        (vector-set! h p (vector-ref h sub-heap))
                        (vector-set! h sub-heap temp)
                        (rh-aux h k sub-heap)))))))
      (begin (rh-aux h k 1) (display h)))
    (define (show) heap)
    (define (size) occupied-size)
    (define (empty?) (= occupied-size 0))
    (define (insert v) (begin
                         (set! occupied-size (+ occupied-size 1))
                         (heap-insert heap occupied-size v)))
    (define (extract) (begin
                        (let ((temp (vector-ref heap 1)))
                          (vector-set! heap 1
                                        (vector-ref heap
                                                    occupied-size))
                          (vector-set! heap occupied-size "NaN")
                          (set! occupied-size (- occupied-size 1))
                          (repair-heap heap occupied-size))))
    (define (dispatch method)
      (cond ((eq? method 'show) show)
            ((eq? method 'size) size)
            ((eq? method 'empty?) empty?)
            ((eq? method 'insert) insert)
```

```
            ((eq? method 'extract) extract)))
    dispatch))
(define myh (make-heap 10 <))
(random-order-insert ordered myh)
((myh 'show))
```

(d) Minimum spanning tree[1] Suppose you have a bunch of processors all connected onto a network, where a processor can have multiple paths between it and the network. You wish to send a message to everyone on this network, and you do not wish the path traversed by the message to be any more costly than necessary. Given a matrix A, with individual locations A_{ij}. Each location holds a cost for getting between processor i and j. It is possible to find the least costly path that touches every processor, using the pseudocode below. Define a SCHEME function `kruskal`
Given as input a matrix (vector of vectors all of size n) A_{ij},
return a list containing s path (a path element is a pair (i,j), a path is a list of path elements, such that the cdr of pair k is the car of pair $k+1$), that includes all processes and is no more costly than any other path connecting all of the processes.

CycleCheck:
Given a path, discover whether it contains a cycle:
A path in which a node appears twice contains a cycle.

Consider this algorithm:

```
f = '()
sort the edges in order minimum first
do e= edge in edge list
    if adding $e$ to $F$ would not create a cycle, add e to F
return F.
```

Solution:

```
;min spanning tree
;kruskal
;sort edge set from lightest to heaviest
;then, with sorted edge set
;really looking at the vertices, but we'll use the letter e
(define (cycle-check e)
  (define (cdr-down-edge-set1 e)
    (define (cdr-down-edge-set2 one e)
      (cond ((null? e) #f)
            ((eq? one (car e)) #t)
            (#t (cdr-down-edge-set2 one (cdr e)))))
    (cond ((null? e) #f)
```

[1]Moore and Mertens, p. 65

```
        ((= 1 (length e)) #f)
        ((cdr-down-edge-set2 (car e) (cdr e)) #t)
        (#t   (cdr-down-edge-set1 (cdr e)))))
  (cond ((null? e) #f)
        ((= 1 (length e)) #f)
        (#t  (cdr-down-edge-set1 e))))

(define vertices (list 1 2 3 1))
(cycle-check vertices)
"now␣not"
(define vertices2 (list 1 2 3 4))
(cycle-check vertices2)
```

---

8. Linked Lists

Singly-Linked Lists

(a) Define a SCHEME function `listInsert`, taking three arguments: The linked list, the index into the list (`0` based) where the new item is to go, and the new item.

Solution:

```
(define (listInsertAfter ll index item); inserts after indexed item
  ;(list payload (prev next))
  (define (next node) (mcdr (cdr node)))
  (define (prev node) (mcar (cdr node)))
  (define (set-next! node value)
    (let ((pointers (cdr node)))
      (set-mcdr! pointers value)))
  (define (set-prev! node value)
    (begin
      (let ((pointers (cdr node)))
        (set-mcar! pointers value))))
  (define (find-index node index k);returns node
    (begin
      (if (= index k) (begin (display node)(newline) node)
          (find-index (next node) index (+ k 1)))))
  (define (linkIn ll nodeUp nodeNew)
    (let* ((downStream (next nodeUp)))
      (begin

        (set-prev! nodeNew nodeUp)
        (set-next! nodeNew downStream)
        (set-next! nodeUp nodeNew)
        (if (null? downStream) "nothing␣to␣do"
            (set-prev! downStream nodeNew) )))
    ll)
  (let ((upstream (find-index ll index 1)))
    (linkIn ll upstream item)))
```

(b) (The Josephus Problem) The Josephus problem resembles the game of musical chairs, a version of the problem is as follows:

A travel agent selects n customers to compete in the finals of a contest for a two week vacation in Maui. The agent places the customers in a circle and then draws a number m out of a hat. the game is played by having the agent walk clockwise around a circle and stopping at every m^{th} customer and asking that customer to leave the circle, until only one remains. The one remaining customer is the winner.

Define a SCHEME function that takes a circular linked list as a parameter, generates a pseudo-random number and follows the Josephus algorithm to eliminate all but one node in the list and returns the value in the last remaining node in the list.

Solution:

```
(require racket/mpair)
(define (make-circular-list lst)
  (define (make-circular-list-aux lst head)
    (if (null? (mcdr lst))
        (set-mcdr! lst head)
        (make-circular-list-aux (mcdr lst) head)))
  (make-circular-list-aux lst lst))

(define mylist (mlist 1 2 3 4 5 6 7 8 9 10 11 12))
(make-circular-list mylist)

(define (josephus lst n)
  (define (josephus-aux lst x prev)
    (cond ((= (mcar lst) (mcar (mcdr lst))) (mcar lst))
          ((= x n) (begin (set-mcdr! prev (mcdr lst))
                          ;;You can view the list after a removal
                          ;;(display (mcdr lst))
                          ;;(display #\newline)
                          (josephus-aux prev 0 lst)))
          (else (josephus-aux (mcdr lst) (+ x 1) lst))))
  (josephus-aux lst 0 lst))
```

Doubly-Linked Lists

(a) Define a SCHEME function that solves the Josephus Problem for doubly-linked lists.

Solution:

```
(define (make-dll lst)
  (define (set-prev node prev) (set-mcar! (mcdr node) prev))
  (define (set-next node next) (set-mcdr! (mcdr node) next))
  (define (new-dll-node val) (mlist val (mcons 0 0)))
  (let* ((head (new-dll-node (mcar lst)))
         (tail head)
         (new-node (list)))
    (define (make-dll-aux lst)
      (if (null? lst)
          (begin (set-next tail head)
                 (set-prev head tail)
                 head)
          (begin (set! new-node (new-dll-node (mcar lst)))
                 (set-prev new-node tail)
                 (set-next tail new-node)
                 (set! tail new-node)
                 (make-dll-aux (mcdr lst)))))
      (make-dll-aux (mcdr lst))))
(define mydll (make-dll (mlist 1 2 3 4 5 6 7 8 9 10 11 12)))

(define (josephus-dll lst n)
  (define (get-prev node) (mcar (mcdr node)))
  (define (get-next node) (mcdr (mcdr node)))
  (define (set-prev node prev) (set-mcar! (mcdr node) prev))
  (define (set-next node next) (set-mcdr! (mcdr node) next))
  (define (josephus-aux lst x prev)
    (cond ((eq? lst (get-next lst)) (mcar lst))
          ((= x n) (begin (set-next prev (get-next lst))
                          (set-prev (get-next lst) prev)
                          (josephus-aux prev 0 lst)))
          (else (josephus-aux (get-next lst) (+ x 1) lst))))
  (josephus-aux lst 0 lst))
(josephus-dll mydll 2)
```

9. Vectors

(a) Write a function named `vector-to-list`, that takes a vector as a parameter and returns a list with the same values, in the same order, as the vector.

Solution:

```
(define (vector-to-list vec)
  (let ((vlist (list)))
    (do ((i (- (vector-length vec) 1) (- i 1)))
      ((< i 0) vlist)
      (set! vlist (cons (vector-ref vec i) vlist)))))
(define myvector (vector 1 2 3 4 5 6))
(vector-to-list myvector)
```

(b) Write a function that, given a vector $(a_1 \ldots a_k)$, returns the vector $(b_1 \ldots b_{k-1})$ so that $b_i = a_{i+1} - a_i$.

Solution:

```
(define (vector-of-diff vec)
  (let ((diff-vec (make-vector (- (vector-length vec) 1))))
    (do ((i 0 (+ i 1)))
      ((>= i (vector-length diff-vec)) diff-vec)
      (vector-set! diff-vec i (- (vector-ref vec (+ i 1))
                                 (vector-ref vec i))))))
```

(c) (Windchill Temperature) When the weather becomes cold, the windchill temperature is used to quantify the effect of wind on the perceived temperature. If W is the wind speed measured in miles per hour, F is the ambient temperature in degrees Fahrenheit, the windchill temperature is given by the formula:

$$T = 1.05 + 0.93T - 3.65W + 3.62\sqrt{W} + 0.1.03T\sqrt{W} + 0.0439W^2$$

Define a SCHEME function which computes the windchill factor for temperatures ranging from 10° to —30° and wind speeds of 0 to 40 mph. Store these values in a table represented by a vector of vectors.

Solution:

```
(define (windchill)
  (define vlength 41)
  (define rows (make-vector vlength (list)))
  (define (make-table)
    (do ((i 0 (+ i 1)))
      ((>= i vlength) rows)
      (vector-set! rows i (make-vector vlength 0))))
  (begin (make-table)
         (do ((r 0 (+ r 1)))
           ((>= r vlength) (vector-ref rows 0))
           (do ((c 0 (+ c 1)))
             ((>= c vlength) (vector-ref rows r))
             (vector-set! (vector-ref rows r)
                          c
                          (+ 1.05
                             (* 0.93 (- 10 r))
                             (- 0 (* 3.65 c))
                             (* 3.62 (sqrt c))
                             (* 0.103 (- 10 r) (sqrt c))
                             (* 0.0439 c c)))))
         rows))
(define table (windchill))
(define (display-table tbl)
  (let ((vlength (vector-length tbl)))
    (do ((r 0 (+ r 1)))
      ((>= r vlength) (display #\newline))
      (do ((c 0 (+ c 1)))
        ((>= c vlength) (display "␣|"))
        (begin (display (+ 10 r))
               (display "␣|␣")
               (display (vector-ref (vector-ref tbl r) c)))))))

(display-table table)
```

(d) (Sieve of Eratosthenes) In ancient Greece, Eratosthenes gave the following algorithm for finding all prime numbers up to a specified number M:

1. Write down the numbers $2, 3, \ldots M$.
2. Cross out the numbers as follows:
   (a) Keep 2 but cross out all multiples of 2 (i.e. cross out $4, 6, 8, \ldots$).
   (b) Keep 3 but cross out all multiples of 3 (i.e. cross out $6, 9, 12, \ldots$).
   (c) Since 4 is already crossed out, go on to the next number that is not crossed out.
   (d) Keep 5 but cross out all multiples of 5.

General Step In general, suppose you have just processed the number P. Go on to the next number that is not crossed out - Q. Keep Q, but cross out all multiples of Q.
Once you reach M, all of the numbers not crossed out are prime numbers.
Define a SCHEME function that takes M as a parameter and uses the Sieve of Eratosthenes algorithm to compute all the prime numbers less than M. You can use a vector of boolean values to store whether a number has been "crossed out" or not.

Solution:

```
(define (sieve-eratosthenes m)
  (let ((is-prime (make-vector (+ 1 m) #t)))
    (begin (vector-set! is-prime 0 #f)
           (do ((i 2 (+ i 1)))
             ((> i m) is-prime)
             (cond ((vector-ref is-prime i)
                    (do ((j (+ i i) (+ j i)))
                      ((> j m) is-prime)
                      (vector-set! is-prime j #f))))))))
```

(e) (Binary Search) Define a SCHEME function that takes a **sorted** vector v and three integers *first* and *last* and *value* representing the position of the first and last locations in the vector to search for *value* and returns a boolean value representing if *value* is present in the vector or not. Your function should use a binary search algorithm to search the vector for *value*.

Binary Search Algorithm Given a target value, the algorithm begins the search by selecting the midpoint in the vector. If the value at the midpoint matches the target, the search is done. Otherwise, because the vector is sorted, the search should continue in either the first half or the second half of the vector. If the target is less than the value at the midpoint, repeat the search by recursively calling the function with the current midpoint as the last. Alternately, if the target is greater than the value at the midpoint, recursively calling the function using the midpoint as the first position in the search. The process is repeated until the target is found or the first and last are the same entry in the vector.

Solution:

```
(define (binary-search vect val)
  (define (binary-search-aux begin end)
    (let ((mid (floor (+ (/ (- end begin) 2) begin))))
      (cond ((= (vector-ref vect mid) val) #t)
            ((= (vector-ref vect end) val) #t)
            ((= begin mid) #f)
            ((< val (vector-ref vect mid))
             (binary-search-aux begin mid))
            (else (binary-search-aux mid end)))))
  (binary-search-aux 0 (- (vector-length vect) 1)))
```

(f) Define a SCHEME function `mergesort`, using the algorithm below:

Using vectors as the data structure, given a vector of numbers, return a vector of numbers in sorted order.

Define a SCHEME function `split` which takes a vector as an argument and "splits it" into two vectors v1and v2 of roughly the same size. (In particular, the sizes of v1and v2 should differ by no more than one.) Since the function must return two vectors, they can be returned as a pair.

Define a SCHEME function `merge` which takes two sorted vectors as arguments and merges them. Remember that the algorithm only need to consult the first element of the two sorted vectors; these two elements could be obtained, for example, by popping a stack.

Combine the `split` and `merge` functions to produce a sorted vector from an unsorted vector.

Solution:

```
#lang racket
(define ( mergesort v )
   (define (subset-vector v a b)
     (let* ((sub-length (+ (- b a) 1))
            ( output (make-vector sub-length)))
       (do ((i 0 (+ i 1)))
         ((>= i sub-length) output)
         (vector-set! output i (vector-ref v (+ i a))))))
   (define ( split v )
      (let ((vec-length  (vector-length v)))
        (if (= vec-length 2) (let* ((a (make-vector 1))
                                    (b (make-vector 1)))
                               (begin
                                 (vector-set! a 0
                                 (vector-ref v 0))
                                 (vector-set! b 0
                                  (vector-ref v 1))
                                 (cons a b)))
             (begin
               (cons (subset-vector v 0 (- (/ vec-length 2) 1))
                     (subset-vector v  (/ vec-length 2)
                                    (- vec-length 1)))))))

   (define (vector-postpend v1 v2 outp v2p)
     (let* ( (length-v2 (vector-length v2)))
       (do ((i 0 (+ i 1)))
         ((>= i (- length-v2 v2p)) v1)
         (vector-set! v1 (+ i outp) (vector-ref v2 (+ i v2p))))))

   ( define ( merge v1 v2 p1 p2 output outp)
      ( cond (( = p1 (vector-length v1 ))
      (vector-postpend output v2 outp p2))
             (( = p2 (vector-length v2 ))
              (vector-postpend output v1 outp p1) )
```

```
          ( else ( if ( < ( vector-ref v1 p1 )
                          ( vector-ref v2 p2 ))
                      (begin
                        (vector-set! output outp
                                     (vector-ref v1 p1))
                        (merge v1 v2 (+ p1 1)
                               p2 output (+ outp 1)))
                      (begin
                        (vector-set! output outp
                                     (vector-ref v2 p2))
                        (display output)
                        (merge v1 v2 p1 (+ p2 1) output
                               (+ outp 1)))))))

( cond ((>= 1 (vector-length v))  v )
       ( else ( let* (( parts ( split v ))
                      ( sorted1 ( mergesort ( car parts )))
                      ( sorted2 ( mergesort ( cdr parts ))))
                 ( merge sorted1 sorted2 0 0
                         (make-vector (+ (vector-length sorted1)
                                         (vector-length sorted2)))
                                         0)))))
```

(g) (Vector of lists) Define a SCHEME function hash that maintains a 26 place vector, and in each place n maintains a list of words beginning with the letter in the English alphabet corresponding to n, e.g. "a" is 1, "x" is 24. The lists do not have to be sorted. Provide one method to insert a string, and another method to read out the strings beginning with a given letter.

Solution:

```
(define (create-hash)
  (let ((hash (make-vector 26)))
    (vector-set! hash 0 '())
    (vector-set! hash 1 '())
    (vector-set! hash 2 '())
    (vector-set! hash 3 '())
    (vector-set! hash 4 '())
    (vector-set! hash 5 '())
    (vector-set! hash 6 '())
    (vector-set! hash 7 '())
    (vector-set! hash 8 '())
    (vector-set! hash 9 '())
    (vector-set! hash 10 '())
    (vector-set! hash 11 '())
    (vector-set! hash 12 '())
    (vector-set! hash 13 '())
    (vector-set! hash 14 '())
    (vector-set! hash 15 '())
    (vector-set! hash 16 '())
    (vector-set! hash 17 '())
    (vector-set! hash 18 '())
    (vector-set! hash 19 '())
    (vector-set! hash 20 '())
    (vector-set! hash 21 '())
    (vector-set! hash 22 '())
    (vector-set! hash 23 '())
    (vector-set! hash 24 '())
    (vector-set! hash 25 '())


    (define (insert theString)
      (let* ((listOfString (string->list theString))
             (firstLetter (car listOfString)))
        (vector-set! hash (- (char->integer firstLetter)
        (char->integer #\a))
                     (cons theString (vector-ref hash
                      (- (char->integer firstLetter)
                       (char->integer #\a)))))))

    (define (show firstLetter)
      (vector-ref hash (- (char->integer firstLetter)
       (char->integer #\a))))

    (define (dispatcher method)
      (cond ((eq? method 'insert) insert)
            ((eq? method 'show) show)))

    dispatcher))
```

10. Objects

    (a) Modify the following code from Lab to require a password (by adding another parameter) for balance inquiries, deposits and withdrawals on the (new-account) object. The password should be set as an additional parameter to the new-account function.

```
(define (new-account initial-balance)
  (let ((balance initial-balance)
        (interestrate 0.01))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
      (begin
        (set! balance
              (- balance f))
        balance))
    (define (bal-inq) balance)
    (define (accrue) (begin (set! balance
                                  (+ balance
                                     (* balance
                                        1
                                        interestrate)))
                            balance))
    (define (setrate r) (set! interestrate r))
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method ' balance-inquire) bal-inq)
            ((eq? method 'accrue) accrue)
            ((eq? method 'setrate) setrate)))))
```

Solution:

```
(define (new-account initial-balance password)
  (let ((balance initial-balance)
        (interestrate 0.01)
        (pw password))
    (define (deposit f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (+ balance f))
        balance))))
    (define (withdraw f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (- balance f))
        balance))))
    (define (bal-inq)
      (cond ((equal? pw password) balance)))
    (define (accrue) (begin (set! balance
                                  (+ balance
                                     (* balance
                                        1
                                        interestrate)))
                            balance))
    (define (setrate r) (set! interestrate r))
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method ' balance-inquire) bal-inq)
            ((eq? method 'accrue) accrue)
            ((eq? method 'setrate) setrate)))))
```

(b) Add a method to the new-account object to change the password. It should take the old password and new password as parameters.

Solution:

```
(define (new-account initial-balance password)
  (let ((balance initial-balance)
        (interestrate 0.01)
        (pw password))
    (define (deposit f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (+ balance f))
        balance))))
    (define (withdraw f password)
      (cond ((equal? pw password)
      (begin
        (set! balance
              (- balance f))
        balance))))
    (define (bal-inq)
      (cond ((equal? pw password) balance)))
    (define (accrue) (begin (set! balance
                                        (+ balance
                                           (* balance
                                              1
                                              interestrate)))
                                  balance))
    (define (setrate r) (set! interestrate r))
    (define (change-password old new)
      (cond ((equal? old pw) (set! pw new))))
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method ' balance-inquire) bal-inq)
            ((eq? method 'accrue) accrue)
            ((eq? method 'setrate) setrate)
            ((eq? method 'change-pw) change-password)))))
```

(c) Write a SCHEME function that implements an object representing a rational number. It should use the GCD function (number a in the Simple Functions section) to reduce the fraction represented by the object.

Solution:

```
(define (make-fraction numerator denominator)
  (define (GCD a b)
    (cond ((= a 0) b)
          ((= b 0) a)
          ((< a b) (GCD a (- b a)))
          (else (GCD (- a b) b))))
  (let ((numer (/ numerator  (GCD numerator denominator)))
        (denom (/ denominator (GCD numerator denominator))))
    (define (value) (/ numer denom))
    (define (show) (begin (display numer)
                          (display "␣/␣")
                          (display denom)
                          (display #\newline)))
    (lambda (method)
      (cond ((eq? method 'value) value)
            ((eq? method 'show) show)))))
```

(d) Define a SCHEME object to model a traffic-light that is always in one of four states, represented by the symbols 'red, 'yellow, 'green, and 'flashing-red. A traffic light can respond to four messages:

show: which returns the current state.

emergency! , which sets the state to 'flashing-red, regardless of what it was before.

set-light , which takes one argument – one of the symbols 'red, 'yellow, or 'green – and sets the state to that symbol.

cycle , which has no effect if the current state is 'flashing-red, but changes a 'green state to 'yellow, a 'yellow state to 'red, and a 'red state to 'green.

Solution:

```
(define (make-traffic-light)
  (let ((state 'red ))
    (define (emergency)
      (set! state 'flashing-red))
    (define (cycle)
      (cond ((eq? state 'green) (set! state 'yellow))
            ((eq? state 'yellow) (set! state 'red))
            ((eq? state 'red) (set! state 'green))))
    (define (set-light symbol) (set! state symbol))
    (define (show) state)
    (lambda (method)
      (cond ((eq? method 'cycle) cycle)
            ((eq? method 'set-light) set-light)
            ((eq? method 'emergency) emergency)
            ((eq? method 'show) show)))))
```

11. Streams Recall that a SCHEME stream is a pair of the form

    `(a . (lambda () ...)).`

    The car of the pair is the first element of the stream. The job of the cdr of the pair (the lambda expression) is to return the rest of the stream (which is another stream, and must have exactly the same structure) when it is called with no arguments.

    Using this convention, one can define

```
(define (first str) (car str))
(define (rest str) ((cdr str)))
```

As an example, the function `int-stream` returns the stream of positive integers $a, a+1, a+2, \ldots$:

```
(define (int-stream a)
  (cons a (lambda () (int-stream (+ 1 a)))))
```

Then, to continue our example

```
> (define a (int-stream 5))
> a
'(5 . #<procedure>)
> (rest a)
'(6 . #<procedure>)
> (rest (rest a))
'(7 . #<procedure>)
> (first (rest (rest a)))
7
```

There are natural ways to *operate* on streams. For example, the following function takes two streams and returns the stream containing the "elementwise product."

```
(define (str-product s t)
  (cons (* (first s) (first t))
        (lambda () (str-product (rest s) (rest t)))))
```

Then, for example

```
> (define a (str-product (int-stream 5) (int-stream 2)))
> a
'(10 . #<procedure>)
> (rest a)
'(18 . #<procedure>)
> (rest (rest a))
'(28 . #<procedure>)
```

You will notice that in the example above, we created a function that returned the stream; when such a stream is created, it will have embedded in the `cdr` of the stream a call to this function. We mention that there are circumstances where it may be helpful for the definition of the stream to carry additional information (beyond simply the first element of the stream). As an example, consider the stream of Fibonacci numbers.

```
(define (fib-stream current next)
  (cons current (lambda ()
                  (fib-stream next (+ current next)))))
```

To generate the stream of regular Fibonacci numbers, you would use the call `(fib-stream 0 1)`.

A final comment: In class, we defined the special functions

```
  (define (delay x) (lambda () x))
  (define (force x) (x))
```

and then used these in our definitions of streams. This is a well-established convention and it would be great to use it in your code. Note, however, that SCHEME has built-in functions (called `delay` and `force`) that behave almost exactly the as the functions above do; one quirk, however, is that if an expression has been delayed (via `(delay E)`), the only way you can get SCHEME to evaluate it is via (force ...). (You cannot simply call it on no arguments.) There is actually a good reason for this.

---

12. Write a function `merge` that *merges* two streams. Specifically, if s and t are two streams of positive numbers in increasing order, `(merge s t)` should be the stream containing all elements appearing in either s or t, also in increasing (sorted) order. For example, if s is the stream of all perfect squares

$$1, 4, 9, 16, 25, 36, \ldots$$

and t is the stream of all perfect cubes

$$1, 8, 27, 64, 125, 216, \ldots$$

then `(merge s t)` would be the stream

$$1, 1, 4, 8, 9, 16, 25, 27, 36, \ldots.$$

Note that there are numbers, such as 1 and $(2^3)^2 = (2^2)^3 = 2^6 = 64$ that are *both* perfect squares and perfect cubes. In this case, your merge function should output the numbers twice.

Solution:

```
(define (merge-streams s t)
  (cond ((null? s) t)
        ((null? t) s)
        ((< (first s) (first t))
         (cons (first s)
               (delay (merge-streams (rest s) t))))
        (else
         (cons (first t)
               (delay (merge-streams s (rest t)))))))
```

13. Picking up from the previous problem, define a merge function that avoids repeated elements (which is to say that if an element appears in both streams, it should appear in the output stream only once).

    Use this to build a stream that generates the ordered list of all positive numbers that are divisible by 2, 3, or 5. To do this, merge together individual streams for the multiples of 2, 3, and 5. Your stream should output these numbers in sorted order, and should output any given integer only once.

Solution:

```
(define (merge-clean-s s t)
  (cond ((null? s) t)
        ((null? t) s)
        ((< (first s) (first t))
         (cons (first s)
               (delay (merge-clean-s (rest s) t))))
        ((< (first t) (first s))
         (cons (first t)
               (delay (merge-clean-s s (rest t)))))
        (else (merge-clean-s (rest t) s))))
```

14. **(Stream zip.)** Define a `stream-zip` function that takes two streams

$$s = s_1, \quad s_2, \quad \ldots \qquad \text{and} \qquad t = t_1, \quad t_2, \quad \ldots$$

as input and returns the stream of pairs

$$(s_1 \,.\, t_1), \quad (s_2 \,.\, t_2), \quad \ldots$$

Solution:

```
(define (zip-s s t)
  (cond ((null? s) '())
        ((null? t) '())
        (else (cons (cons (first s) (first t))
                    (delay (zip-s (rest s) (rest t)))))))
```