You may assume that atom? is a primitive procedure; you don’t need to define it. Other helper functions that aren’t a standard part of Scheme you need to write.

Each numbered question is worth 10 points. Please write and sign the Honor Pledge at the end of your solutions.

1. Write (index x lat) which returns the 0-based index of item x in the flat list lat. If x is not in lat this should return -1.

   (define inc (lambda (t) (if (= -1 t) t (+ 1 t))))
   (define index (lambda (x lat)
       (cond
         [(null? lat) -1]
         [(= x (car lat)) 0]
         [else (inc (index x (cdr lat)))])))
2. Write procedure (SameElts? lat1 lat2) that returns #t if lat1 and lat2 have the same elements (including multiplicities) though not necessarily in the same order. Naturally, SameElts? should return #f if it doesn’t return #t. For example, (SameElts? '(1 2 3) '(3 2 1)) returns #t while (SameElts? '(1 2 3) '(1 2 1 2 3)) returns #f.

(define SameElts (lambda (lat1 lat2)
  (cond
   [(null? lat1) (if (null? lat2) #t #f)]
   [(= (count (car lat1) lat1) (count (car lat1) lat2))
    (SameElts (rember-all (car lat1) lat1) (rember-all (car lat1) lat2))]
   [else #f]))

(define count (lambda (x lat)
  (cond
   [(null? lat) 0]
   [(eq? x (car lat)) (+ 1 (count x (cdr lat)))]
   [else (count x (cdr lat))]))

(define rember-all (lambda (x lat)
  (cond
   [(null? lat) null]
   [(eq? x (car lat)) (rember-all x (cdr lat))]
   [else (cons (car lat) (rember-all x (cdr lat)))])))
3. Write (equals? L1 L2) which determines if the general lists $L_1$ and $L_2$ are structurally identical: they have the same elements in the same structures. For example (equals? ‘(1 (2 3 (4))) ‘(1 (2 3) (4))) returns #t while (equals? ‘(1 (2 3 (4))) ‘(1 (2 3) 4)) returns #f.

(define Equals (lambda (L1 L2)
  (cond
   [(null? L1) (if (null? L2) #t #f)]
   [(and (pair? L1) (pair? L2)) (and (Equals (car L1) (car L2))
                                     (Equals (cdr L1) (cdr L2)))]
   [(or (pair? L1) (pair? L2)) #f]
   [else (eq? L1 L2)])
))
4. Use foldl or foldr to write \((\text{count} \ a \ \text{lat})\), which returns the number of instances of atom \(a\) in \(\text{lat}\).

\[
\text{(define Count (lambda (a lat)
    (foldr (lambda (x y) (if (eq? x a) (+ 1 y) y)) 0 lat)))}
\]
5. Here is a tree constructor:

    (define new-tree (lambda (v list-of-children) (list ‘tree v list-of-children)))

We’ll use the null list to represent an empty tree.

Define a Scheme procedure (height t) that produces the height of tree t which is built with this constructor. Remember that the height of a tree is the number of edges on the longest path from the root to a leaf.

    (define height (lambda (t)
        (cond
            [(leaf? t) 0]
            [else (+ 1 (apply max (map height (Kids t))))])))

Here are the helper functions:

    (define Kids (lambda (t) (caddr t)))
    (define leaf? (lambda (t) (null? (Kids t))))
6. We spent a lot of time talking about scoping rules. Scheme uses static (or lexical) scoping; early LISP used dynamic scoping. Explain in one or two sentences the difference between static and dynamic scoping, then give a Scheme expression that evaluates differently under the two scoping mechanisms. Say what your expression evaluates to under each mechanism.

Static scoping looks for free variables in the environment in which the function is created. Dynamic scoping looks for free variables in the environment in which the function is called.

\[
\text{(let ([A 1])}
\quad \text{(let ([f (lambda (x) (+ x A))])}
\quad \text{(let ([A 10])}
\quad \quad \text{(f 3))})
\]

With static scoping this evaluates to 4, with dynamic scoping to 13.
7. You remember \( \text{let\^*} \), which evaluates its bindings sequentially:

\[
(\text{let\^* } ([x \ 5] [y \ x]) \ y)
\]
is a valid expression that evaluates to 5.

What code would you use for (parse exp) and for (eval-exp tree env) to add \( \text{let\^*} \) to your miniScheme interpreter? You can use (without redefining) any datatypes you used in your interpreter but give some kind of explanation of what you are using so I can read your code.

My approach to this is to parse a \( \text{let\^*} \) into a nested sequence of lets, which means we have to modify the parser but not the interpreter.

The parser should use the condition:

\[
[(\text{eq?} \ (\text{car} \ \text{exp}) \ \text{‘let\^*}) \ (\text{makeLet\^*} \ (\text{map} \ \text{car} \ (\text{cadr} \ \text{exp})) \ (\text{map} \ \text{parse} \ (\text{map} \ \text{cadr} \ (\text{cadadr} \ \text{exp})))
\]

Procedures make\( \text{let\^*} \) converts the \( \text{let\^*} \) expression into a sequence of nested lets:

\[
(\text{define} \ \text{makeLet\^*} \ (\lambda (\text{syms} \ \text{vals} \ \text{body})
\]  
(\text{cond}
  
    [(\text{null?} \ \text{syms}) \ \text{body}]
  
    [\text{else} \ (\text{newLetExp} \ (\text{list} \ (\text{car} \ \text{syms})) \ (\text{list} \ (\text{car} \ \text{vals}))
       \ (\text{makeLet\^*} \ (\text{cdr} \ \text{syms}) \ (\text{cdr} \ \text{vals}) \ \text{body})))]
\]

Here new\text{LetExp} is the constructor for the tree data type that represents a standard let expression. It takes 3 arguments: a list of binding symbols, a list of parsed binding values, and a parsed body.
8. Here is a stream: (0 1 1 2 2 4 3 8 4 16 5 32 6 64 ..... ) As you can see, its values alternate between n and $2^n$. Give a Scheme expression that produces this stream.

(define S (lambda (n p) (cons n (cons p (S (+ n 1) (* 2 p))))))
9. Write a Continuation Passing Style function called RemerBob that takes a general list argument and removes all of the instance of the atom 'bob. If the list contains the atom 'PANIC the function returns only 'PANIC, regardless of how deeply it is buried in the recursion when it finds this atom. Remember that CPS procedures need to be tail-recursive.

For example (RemerBob '(john (paul bob) ((george) ringo) bob) (lambda (y) y)) returns (john (paul) ((george) ringo)), while (RemerBob '(pete (roger bob PANIC) ((keith john))) (lambda (y) y)) returns 'PANIC.

(define RemerBob (lambda (L k)
  (cond
      [(null? L) (k null)]
      [(pair? (car L))
        (RemerBob (car L)
          (lambda (x) (RemerBob (cdr L)
            (lambda (y) (k (cons x y))))))]]
    [(eq? (car L) 'bob)  (RemerBob (cdr L) (lambda (x) (k x)))]
    [(eq? (car L) 'PANIC) 'PANIC]
    [else (RemerBob (cdr L) (lambda (x) (k (cons (car L) x))))])))
10. What will the following procedure return if I call it with \((f \ (3 \ 2 \ 1 \ 0 \ 1 \ 2 \ 3))\)? If your answer is correct it will get full credit, but in case your answer isn’t correct it would help if you gave some explanation.

\[
\begin{align*}
(\text{define } f &\ (\lambda (\text{lat})
\quad (\text{call/cc} \ (\lambda (k))
\quad (\text{let} \ ((g \ (\lambda (y)
\quad (\text{call/cc} \ (\lambda (c)
\quad (\text{if} \ (= \ 1 \ y)
\quad \quad (c \ 5)
\quad \quad (+ \ y \ 2)))))))
\quad (\text{cond}
\quad \quad [(\text{null? lat}) \ 0]
\quad \quad [ (= \ 0 \ (\text{car lat})) \ (k \ 0)]
\quad \quad [\text{else} \ (+ \ (g \ (\text{car lat})) \ (f \ (\text{cdr lat})))]))))))
\end{align*}
\]

This evaluates to 14. \((g \ 3)\) is 5, \((g \ 2)\) is 4, \((g \ 1)\) is 5. Procedure \(f\) sums the value of \(g\) applied to each of the elements of the \(\text{lat}\) prior to the first 0. 5+4+5 is 14