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.
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.
3. Write (equals? L1 L2) which determines if the general lists \( L1 \) and \( L2 \) are structurally identical: they have the same elements in the same structures. For example
\[
\text{(equals? '(1 2 3 4)) '(1 (2 3) (4)))\]
returns \#t\ while
\[
\text{(equals? '(1 2 3 4)) '(1 2 3) 4)}\]
returns \#f\.
4. Remember the fold function:

```scheme
(define fold
  (lambda (recur base lat)
    (letrec ((h (lambda (s)
                   (cond
                     [(null? s) base]
                     [else (recur (car s) (h (cdr s)))])))
        (h lat))))
```

Use fold to write `(count a lat)`, which returns the number of instances of atom `a` in `lat`. Note for 2018: do this with foldl or foldr.
5. Here is a tree constructor:

\[
\text{(define new-tree (lambda (v list-of-children) (list 'tree v list-of-children)))}
\]

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

Define a Scheme procedure \textbf{(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.
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.
7. You remember \texttt{let*}, which evaluates its bindings sequentially:
   \[\texttt{(let* ([x 5] [y x]) y)\texttt{}}\] is a valid expression that evaluates to 5.
   What code would you use for (parse \texttt{exp}) and for (eval-exp \texttt{tree env}) to add \texttt{let*} to your miniScheme interpreter? You can use (without redefining) any data types you used in your interpreter but give some kind of explanation of what you are using so I can read your code.
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.
9. Write a Continuation Passing Style function called RemberBob 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 (RemberBob ‘(john (paul bob) ((george) ringo) bob) (lambda (y) y)) returns (john (paul) ((george) ringo)), while (RemberBob ‘(pete (roger bob PANIC) ((keith john))) (lambda (y) y)) returns ‘PANIC.
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.

```
(define f (lambda (lat)
  (call/cc (lambda (k)
    (let ([g (lambda (y)
        (call/cc (lambda (c)
          (if (= 1 y)
            (c 5)
            (+ y 2))))))]
      (cond
        [(null? lat) 0]
        [(= 0 (car lat)) (k 0)]
        [else (+ (g (car lat)) (f (cdr lat)))])))))
```
[This page is for scratch work.]
Please write and sign the Honor Pledge at the end of your solutions.