There are 10 questions here, each worth 10 points.
In any question you may use atom? as a Scheme primitive. You need to write any other helper functions you use that aren’t standard parts of Scheme.
Please sign the Honor Pledge on the last page before handing this in.

1. **Write (flatten L)**, which takes an arbitrary list and turns it into a flat list with the same atoms in the same order. (flatten '((3 2) 1 (4 (5 6)))) returns (3 2 1 4 5 6). Note that in Lab 10 you wrote flatten-k. For this question you can write flatten in any style you like.
2. Remember the fold function:

\[
\text{(define fold} \\
\hspace{1cm} \text{(lambda (f base lat)}} \\
\hspace{2cm} \text{(letrec ([(h (lambda (s)}} \\
\hspace{3cm} \text{(cond} \\
\hspace{4cm} \text{[(null? s) base]}} \\
\hspace{5cm} \text{[else (f (car s) (h (cdr s)))]}]})} \\
\hspace{1cm} \text{)(h lat)))})
\]

**Use fold to write the procedure (BigTwo L), which returns a list with the largest two elements of flat list L. You can assume that all entries of L are non-negative. For example, (BigTwo `(4 2 1 8 6 3 9 1 2)) returns (9 8).**
3. Function (remove L a b c d) takes a flat list L and any number of atoms; the function removes every instance of each of those atoms from L and returns the result. So (remove '(1 2 3 2 1) 2) returns (3) while (remove '(1 2 3 2 1 2 2 4) 2 4) returns (1 3 1). Write function remove.
4. Suppose we define a tree data structure as
   (define new-tree tree (lambda (val list-of-children)
       (*tree val list-of-children))
   and we represent the empty tree with null.

   Here are some tree definitions using this datatype:
   (define A (new-tree 4 null))
   (define B (new-tree 9 (list A))
   (define C (new-tree 2))
   (define D (new-tree 7 null))
   (define E (new-tree 8 (list C D)))
   (define F (new-tree 1 null))
   (define G (new-tree 5 (list B F)))

   Write a procedure that will return a flat list of the leaves (only the leaves) of such a tree.
   For tree G in the example above this will return (4 2 7 1). You can assume there is a
   recognizer procedure (tree? x) and getter procedures (Value tree) and (Kids tree).
5. In my interpreter I parse a let expression into

(let-exp ids vals bod u)

**What is the corresponding case of** (eval-exp tree env) **where exp is a let-exp node?**

[Hint: you may want to make use of (extended-env ids values old-env)].
6. **Explain in one or two sentences** the difference between static or lexical binding and dynamic binding. **Give an example** of an expression that evaluates differently in the two binding schemes.
7. How could we use lambda expressions to represent environments? I would like env to be a lambda expression, so that (env ‘x) returns the value bound in the environment to symbol ‘x. If the symbol is not bound in the environment this should return ‘notFound.

   a) Write (extended-env syms vals old-env), which returns a new environment with the given bindings. Of course, this new environment should be a lambda expression.

   b) Write (EmptyEnv) a function with no arguments that returns a representation for an empty environment.

   c) Write (Lookup env x) a procedure that looks up the value bound to symbol x in env.
8. The Fibonacci numbers satisfy the equation $x_n = x_{n-1} + x_{n-2}$; $x_0 = 0$, $x_1 = 1$. We can define other sequences like this. **Create a stream** that satisfies $x_n = 2x_{n-1} + 5x_{n-2} + 1$; $x_0 = 1$, $x_1 = 3$. The first 5 elements of this sequence are 1, 3, 12, 40, and 141.

The stream primitives you can use for this are car$, cdr$, cons$ (of course), +$ and map$. Anything else you need to write yourself.
9. (sub a b L) is a function that takes atoms a and b and a general list L. This function replaces each instance of atom b in L with atom a (i.e., it substitutes a for b). So (sub 1 2 '(1 2 3 4)) is (1 1 3 4) and (sub 1 2 '((1 2) 3 (4 2 1))) is ((1 1) 3 (4 1 1)). Write function (sub-k a b L k) that implements sub in continuation-passing style.
10. Consider the following function:

\[
\text{(define prod (lambda (vec)
  (call/cc (lambda (k)
    (cond
      [(null? vec) 1]
      [(eq? 0 (car vec)) (k 99)]
      [else (* (car vec) (prod (cdr vec)))])]))})
\]

What does \( \text{prod } '(2 1 0 3 4) \) return?
You can use this page for extra space for any question; just indicate clearly which problem you are working on.

Please write and sign the Honor Pledge when you are finished with the exam.