Directions: There are 7 numbered problems worth 14 points each; you get 2 points for free. You don’t need to write helper functions via letrecs; you can define them at the top level (and that might make your code easier to read). You can assume there is a predicate (atom? x; you don’t need to write that. Any other helper functions you should write yourself. In the following questions argument lat is always a flat list, such as ‘(a b c) and argument L is a general list, such as ‘(a (b c (d e)) f). There is a place on the last page for you to sign the Honor Pledge.

1. Write procedure (get n lat) which returns the element of lat at index n. If n is greater than or equal to the length of lat, get should return the last element of lat. So (get 0 ‘(a b c d e)) returns a, (get 3 ‘(a b c d e)) returns d, and (get 100 ‘(a b c d e)) returns e.

(define get (lambda (n lat)
  (cond
   [(null? (cdr lat)) (car lat)]
   [ (= n 0) (car lat)]
   [else (get (- n 1) (cdr lat))])))

2. Use foldr or foldl to write (replace old new lat) which replaces each instance of atom old with atom new in lat, a flat list of atoms.

(define replace (lambda (old new lat)
  (foldr
   (lambda (x y)
     (if (eq? x old) (cons new y) (cons x y)))
   null
   lat)))

3. Write any way you wish procedure (replace* old new L) which replaces each instance of old with new in the general list L.

(define replace* (lambda (old new L)
  (cond
   [(atom? L) (if (eq? L old) new L)]
   [else (map (lambda (t) (replace* old new t)) L))])))
4. Write *(Alternates lat)* which returns a pair of lists, the first with the even-indexed elements of *lat* and the second with the odd-indexed elements. The two returned lists should have elements in the same order as the original *lat*. For example, *(Alternates ‘(a b c d e f g))* returns ‘((a c e g) (b d f)).

\[
\text{(define EveryOther (lambda (lat))}
\begin{align*}
\text{  (cond} & \text{[(null? lat) null]} \\
\text{              } & \text{[(null? (cdr lat)) lat]} \\
\text{              } & \text{[else (cons (car lat) (EveryOther (cddr lat))))]} \\
\text{(define Alternates (lambda (lat))} & \text{(list (EveryOther lat) (EveryOther (cdr lat))))}
\end{align*}
\]

5. Use map and apply to write *(Count a L)* which returns the number of times atom *a* occurs in general list *L*.

\[
\text{(define Count (lambda (a L))}
\begin{align*}
\text{  (cond} & \text{[(atom? L) (if (eq? L a) 1 0)]} \\
\text{              } & \text{[else (apply + (map (lambda (t) (Count a t))) ])}}
\end{align*}
\]

6. Let’s say that a *count list* for a list *L* is a list of pairs, where the first element of each pair is one of the atoms of *L* and the second element of the pair is how often that atom occurs in *L*. So a count list for ‘(a b a c d d a) is ( (a 3) (b 1) (c 1) (d 2) ). Write function *(CountList lat)* that returns a count list for flat list *lat*. Your solution can list the elements of *lat* in any order you wish.

\[
\text{(define CountList (lambda (lat))}
\begin{align*}
\text{  (cond} & \text{[(null? lat) null]} \\
\text{              } & \text{[else (cons (list (car lat) (count (car lat) lat))} \\
\text{                     } & \text{(CountList (rember-all (car lat) lat))))} \\
\end{align*}
\]

\[
\text{(define rember-all (lambda (a lat))}
\begin{align*}
\text{  (cond} & \text{[(null? lat) null]} \\
\text{              } & \text{[(eq? (car lat) a) (rember-all a (cdr lat))]} \\
\text{              } & \text{[else (cons (car lat) (rember-all a (cdr lat))))]} \\
\end{align*}
\]
7. Explain step-by-step how the following expression will be evaluated in the top-level environment.

\[
\left( \text{let } \{(a \leftarrow 5\}) \ \text{lambda } (x) \ (+ x \ a) \right) \ 6
\]

Note that it consists of a let expression:

\[
\text{let } \{(a \leftarrow 5\}) \\
\text{lambda } (x) \ (+ x \ a)
\]

and a call to value of this let expression with argument 6. In particular you should say what the let expression evaluates to, and how this value is called with argument 6.

First the let expression is evaluated:

The top-level environment is extended with a binding of a to 5; call this extended environment E.

The lambda expression is evaluated in environment E, producing a closure with parameter list \(x\), body \(+ x \ a\), closure environment E.

This closure is the value of the let expression.

The argument 6 is evaluated, producing value 6.

Finally, the closure is applied to the argument: The closure environment E is extended with a binding of the closure parameter \(x\) to the argument 6; call this environment E’.

The closure body \(+ x \ a\) is evaluated in E’. E’ has \(x\) bound to 6 and \(a\) to 5, so \(+ x \ a\) in this environment evaluates to 11.