1. Use fold 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)
       (fold (lambda (x y) (if (eq? x old)
           (cons new y)
           (cons x y)))
       null lat)))

2. Write (replace* old new L) which replaces each instance of old with new in the general list L

   (define replace* (lambda (old new L)
       (cond
           [(null? L) null
            [(not (pair? L)) (if (eq? L old) new L)]
            [else (cons (replace* old new (car L))
                         (replace* old new (cdr L)))]))))
3. Write (Alternate s lat) which returns a pair of lists, the first with the even-indexed elements of lat and the second with the odd-indexed elements. For example, (Alternates ‘(a b c d e f g)) returns ‘( (a c e g) (b d f) ).

Here’s one solution, using fold:

(define Alternates (lambda (lat0
          (fold (lambda (x y) (list (cons x (cadr y)) (car y))) (list null null) lat)))

Here’s another solution, using accumulator-style. This builds the lists backwards; if that bothers you use append rather than cons with the recursive call:

(define Alternates (lambda (lat a b)
      (cond
        [(null? lat) (list a b)]
        [(null? (cdr lat)) (list (cons (car lat) a) b)]
        [else (Alternates (cddr lat) (cons (car lat) a) (cons (cadr lat) b))]]))

The most popular solution was this. Function H finds the even-numbered elements:

(define H (lambda (lat)
      (cond
        [(null? lat) null]
        [(null? (cadr lat)) (list (car lat))]
        [else (cons (car lat) (H (cddr lat)))])))

(define Alternates (lambda (lat)
      (list (H lat) (H (cdr lat)))))

4. Write (MyOr v1 v2 v3 v4 …) which takes any number of arguments and returns #t if any of the arguments are #t and returns #f otherwise.

(define MyOr (lambda args
      (cond
        [(null? args) #f]
        [(eq? #t (car args)) #t]
        [else (apply MyOr (cdr args))])))
5. Write \((\text{Count a L})\) which returns the number of times atom \(a\) occurs in general list \(L\).

\[
(\text{define Count} \ (\lambda \ (a \ L) \\
\qquad (\text{cond} \\
\qquad \quad [(\text{null?} \ L) \ 0] \\
\qquad \quad [(\text{not} \ (\text{pair?} \ L)) \ (\text{if} \ (\text{eq?} \ L \ a) \ 1 \ 0) \\
\qquad \quad [\text{else} \ ((\text{apply} \ + \ ((\lambda \ (x) \ (\text{Count} \ x) \ L))))]))]
\]

6. Let’s say that a \textit{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 \((\text{CountList lat})\) that returns a count list for flat list \(lat\).

\[
(\text{define CountList} \ (\lambda \ (lat) \\
\qquad (\text{cond} \\
\qquad \quad [(\text{null?} \ lat) \ \text{null}] \\
\qquad \quad [(\text{null?} \ (\text{cdr} \ lat)) \ (\text{list} \ (\text{list} \ (\text{car} \ lat) \ 1))] \\
\qquad \quad [\text{else} \ ((\text{AddTo} \ (\text{car} \ lat) \ (\text{CountList} \ (\text{cdr} \ lat))))])))
\]

\[
(\text{define AddTo} \ (\lambda \ (x \ lop) \\
\qquad (\text{cond} \\
\qquad \quad [(\text{null?} \ lop) \ (\text{list} \ (\text{list} \ x \ 1))] \\
\qquad \quad [(\text{eq?} \ x \ (\text{caar} \ lop)) \ (\text{cons} \ (\text{list} \ x \ (+ \ 1 \ (\text{cadar} \ lop))) \ (\text{cdr} \ lop))] \\
\qquad \quad [\text{else} \ ((\text{cons} \ (\text{car} \ lop) \ (\text{AddTo} \ x \ (\text{cdr} \ lop))))])]
\]

Another popular way to do this is

\[
(\text{define CountList} \ (\lambda \ (lat) \\
\qquad (\text{cond} \\
\qquad \quad [(\text{null?} \ lat) \ \text{null}] \\
\qquad \quad [\text{else} \ ((\text{cons} \ (\text{list} \ (\text{car} \ lat) \ (\text{Count} \ (\text{car} \ lat) \ lat))) \\
\qquad \qquad (\text{CountList} \ (\text{rember-all} \ (\text{car} \ lat) \ lat))))])
\]
7.

a) Consider the expression

\[
\text{(let ([a 5])} \\
\text{  (lambda (x) (+ a 9))})
\]

What is the value of this expression when it is evaluated in some environment E1?

b) If symbol f is bound to the value of the expression in (7a), explain how Scheme evaluates the expression

\[
\text{(let ([a 1])} \\
\text{  (f a))}
\]

in some environment E2. Note that I am looking for an algorithm – what steps Scheme takes to evaluate this, not just the result.

The expression in (a) evaluates to a closure, which consist of

- Environment E1+[a bound to 5]
- Parameters (x)
- Body (+ a 9)

When expression (b) is evaluated, first environment E2 is expanded with [a bound to 1]. Then f is evaluated, giving the closure described in (a), a is evaluated in E2+[a 1], giving 1 and the closure is applied to argument 1. This takes the closure’s environment: E1+[a 5], extends it with [x 1], and evaluates the body: (+ a 9) in this extended environment. Since a is bound to 5 in this environment, it evaluates to 14.