1) Write procedure \((\text{insert } n \; a \; \text{lat})\) that inserts atom \(a\) at position \(n\) in \(\text{lat}\) and returns the resulting list. If \(n\) is greater than or equal to the length of \(\text{lat}\), insert atom \(a\) at the end of \(\text{lat}\).

So \((\text{insert } 0 \; 'x \; ('a \; b \; c \; d \; e))\) returns \((x \; a \; b \; c \; d \; e)\), \((\text{insert } 3 \; 'x \; ('a \; b \; c \; d \; e))\) returns \((a \; b \; c \; x \; d \; e)\), and \((\text{insert } 25 \; 'x \; ('a \; b \; c \; d \; e))\) returns \((a \; b \; c \; d \; e \; x)\)

\[
(\text{define insert} \; (\lambda (n \; a \; \text{lat}) \; \begin{cases} 
\text{null? lat) (list a)} \\
\text{(- 0 n) (cons a lat)} \\
\text{else} \; \text{cons} \; (\text{car lat}) \; \text{insert} \; (- \; n \; 1) \; \text{a} \; \text{(cdr lat))}}
\end{cases})
\]

2) Write procedure \((\text{replace } a \; b \; L)\) that replaces every instance of atom \(a\) with atom \(b\) in the general list \(L\). So \((\text{replace } 'a \; 'x \; ('a \; b \; (c \; (a \; d) \; a \; (b \; c)))\) returns \((x \; b \; (c \; (x \; d) \; x \; (b \; c)))\)

\[
(\text{define replace} \; (\lambda (a \; b \; L) \; \begin{cases} 
\text{null? L null} \\
\text{atom? (car L)} \\
\text{if (eq? (car L) a)} \\
\text{cons b (replace a b (cdr L))} \\
\text{cons (car L) (replace a b (cdr L))}} \\
\text{else} \; \text{cons} \; (\text{replace a b (car L)}) \; \text{(replace a b (cdr L))}\end{cases}))
\]
3) What will this expression evaluate to? If you are sure your answer is right that is all I need, but if your answer is wrong it would help to have an explanation.

(let ([a 3] [b 5])
    (let ([f (lambda (x) (+ x a))])
        (let ([a 10] [b 20])
            (f b))))

The environment in force when f is evaluated has a bound to 3; this is part of the closure that is the value of f. When (f b) is evaluated, b has value 20. (f b) is called by extending the closure environment of f (with a bound to 3) with a binding of x to 20, and evaluating the body of f (+ x a) in this environment. Since x is bound to 20 and a to 3, it evaluates to 23.

4) Explain step-by-step how the following expression will be evaluated in the top-level environment.

(let ([s 7] [f (lambda (x y) (+ x y))])
    (f 2 s))

1. The Let expression has two bindings: s to 7 and f to the value of the lambda expression. 7 and the lambda are both evaluated in the top-level environment. 7, of course, evaluates to 7 and the lambda expression evaluates to a closure \((x y) (+ x y)\)  top-level-env \}

The top-level environment is extended with bindings of s to 7 and f to this closure. Call this extended environment E.

2. We evaluate the body of the let, \((f 2 s)\) in E. In E 2 evaluates to 2 and s to 7, so we call f with these values.

3. To call f we extend the closure environment (the top-level env) with bindings of x and y to 2 and 7 (let’s call this new environment E’), then evaluate the closure body \((+ x y)\) in E’. Since x is bound to 2 and y to 7, the body evaluates to 9.
5) This problem concerns a data structure \textit{Grades} that represents the grades (numbers between 0 and 100) that students receive on an exam or assignment. The constructor for this structure is

\begin{verbatim}
(define MakeGrades (lambda (list-of-students list-of-grades)
                  (list 'grades list-of-students list-of-grades)))
\end{verbatim}

For example, \texttt{(MakeGrades '(harry ron hermione) '(90 65 100))} returns the triple

\texttt{('grades 'harry ron hermione) '(90 65 100))}

a) Give a function \texttt{(LookupGrade(person g))} which returns the grade saved for the given person in a grades-structure formed by the constructor above. If the person is not found in this structure you should return 0. For example, if we say

\begin{verbatim}
(define G1 (MakeGrades '(harry ron hermione) '(90 65 100)))
\end{verbatim}

\texttt{(LookupGrade 'ron G1)} returns 65 and \texttt{(LookupGrade 'neville G1)} returns 0.

\begin{verbatim}
(define LookupGrade (lambda (person g)
                       (letrec ([h (lambda (person-grade)
                                (cond
                                 [(null? person-grade) 0]
                                 [(eq? (car person-grade) person) (car grade-list)]
                                 [else (h (cdr person-grade) (cdr grade-list))]))]
                        (h (cadr g) (caddr g)))))
\end{verbatim}

b) Give a function \texttt{(AverageGrade person g1 g2 g3 \ldots gn)} that takes a name and any number of grades-structures and returns the person’s average grade. Note that there is a standard function \texttt{(length L)} that gives the number of entries of list \texttt{L}; you can use this to find the number of grades for the average. If we say

\begin{verbatim}
(define G1 (MakeGrades '(harry ron hermione) '(90 65 100)))
(define G2 (MakeGrades '(harry ron hermione) '(95 61 100)))
\end{verbatim}

then \texttt{(AverageGrade 'ron G1 G2)} returns 63.

\begin{verbatim}
(define AverageGrade (lambda args
                        (/ (apply + (map (lambda (g) (LookupGrade (car args) g)) (cdr args)))
                           (length (cdr args)))))
\end{verbatim}
6) Write a procedure (Sum lat). As you might expect, this returns the sum of the numbers in lat. However, if lat contains the atom ‘squares’ (Sum lat) returns the sum of the squares of the numbers in lat. For example (Sum '(1 2 3 4)) returns 10, and (Sum '(1 2 3 squares 4)) returns 30. You can assume that any entry in lat other than the atom ‘squares’ will be a number.

An easy, and a bit cheesy, way to do this is to ask if ‘squares’ is a member of lat; if it is sum the squares of the numerical members of lat; if it isn’t sum the members (which must all be numbers, of lat).

I was hoping some of you would realize that tail recursions give a way to modify what a function is going to return. Here is an accumulator-passing version:

(define sq (lambda (x) (* x x)))
(define Sum1 (lambda (lat)
    (letrec ([Sum-a (lambda (L a1 a2)
        (cond
            [(null? L) a1]
            [(eq? (car L) 'squares)  (+ a2 (apply + (map sq (cdr L))))]
            [else (Sum-a (cdr L) (+ a1 (car L)) (+ a2 (sq (car L))))])])
        (Sum-a lat 0 0)))))

Here is a similar continuation-passing version:

(define Sum (lambda (lat)
    (letrec ([Sum-k (lambda (L k1 k2)
        (cond
            [(null? L) (k1 0)]
            [(eq? (car L) 'squares)  (k2 (apply + (map sq (cdr L))))]
            [else (Sum-k (cdr L)
                (lambda (x) (k1 (+ x (car L))))
                (lambda (y) (k2 (+ y (sq (car L)))))))])]
        (Sum-k lat (lambda (x) x) (lambda (x) x)))))
7) Use foldl or foldr to write a function \textbf{(minMax lat)} that takes a flat list of numbers and returns a pair containing the smallest and largest values in that list. For example \textbf{(minMax '(5 2 8 6 3 6 1))} returns \textbf{(1 8)}. You can assume all of the numbers in \textit{lat} are between 0 and 100.

\begin{verbatim}
(define minMax (lambda (lat)
  (foldr (lambda (x y)
    (cond
      [(null? (car y)) (list x x)]
      [(< x (car y)) (list x (cadr y))]
      [(> x (cadr y)) (list (car y) x)]
      [else y))]
  (list null null)
  lat)))
\end{verbatim}