1. Let’s start with implementing a datatype. I want to add a do-loop to miniScheme; this has the form (do n exp); it will execute expression exp n times. For example we might say (do 10 (set! x (+ x 2))). My parser will need a do-exp datatype. This consists of 4 functions: constructor (new-do-exp n exp), recognizer (do-exp? x) that tells me whether or not any object x is a do-exp, and two getters (do-count tree) and (do-body tree) that respectively return n and exp if tree was constructed as (new-do-exp n exp). The constructor and getters don’t need to do any error checking; you can assume the getters are only called on a tree that is a do-exp. **Give Scheme code for** (new-do-exp n exp), (do-exp? x) (do-count tree) and (do-body tree).

```scheme
(define new-do-exp (lambda (n exp) (list 'do-exp n exp)))
(define do-exp? (lambda (x)
    (cond
        [(not (pair? x)) #f]
        [else (eq? (car x) 'do-exp)])))
(define do-count (lambda (tree) (cadr tree)))
(define do-body (lambda (tree) (caddr tree)))
```
2. We have said in class that let-expressions and lambda-expressions are interchangeable. Give an expression involving a lambda-expression that is equivalent to 
(\( (\text{let}} \ [[x \ (+2 \ 3)] [y \ 4]] \ (* \ y \ (+ \ x \ 1))] \))

\(( (\lambda x \ y \ (* \ y \ (+ \ x \ 1))) \ (+ \ 2 \ 3) \ 4)\)
3. Closures
   a. What is a closure? What is it used for? What data does it contain? One or two sentences should be enough for this.

   A closure is the value of a lambda-expression. It contains the parameter list and body of the lambda, and also the environment in which the lambda was evaluated.

   b. What is the point of a closure? To call a function such as (lambda (x) (+ x 3)) with an argument such as (+ 4 1), we just need to know the argument value, the parameter (x) and the body (+ x 3). Why do we ever need anything else? Again, one or two sentences should be sufficient.

   Closures are what we need to implement lexical scoping. The closure environment is used to look up free variables in the lambda-expression’s body.
4. Parameter Passing

a. Explain in one sentence, or two if you must, the difference between call-by-value and call-by-reference. Which does Scheme use?

Call-by-value passes only the value of the argument, while call-by-reference passes the address (or box) of the argument, so assignments to a parameter of the function actually change the corresponding argument. Scheme uses call-by-value.

b. Give an example that evaluates differently in call-by-value than in call-by-reference, and say what it evaluates to under each parameter-passing mechanism.

(let ([A 10])
  (let ( [f (lambda (x) (set! x 5))])
    (begin (f A) A))

With call-by-value this returns 10 because f doesn’t change variable A. With call-by-reference it returns 5 because the parameter x becomes another name for the argument A; assignments to x modify A.
5. To implement let-expressions in Lab 7 we parsed them into a let-exp datatype. Let’s assume the constructor for the let-exp is new-let-exp and it takes 3 arguments: a list of the binding symbols, a list of parse-trees for the binding values, and the parsed body.

   a. there is a line in the parser for parsing expression exp that starts
      
      
      ```
      [(eq? (car input) 'let) (new-let-exp .......)]
      
      Give this line of code. If you want to call some helper functions give them as well.
      ```
      
      ```
      [(eq? (car input) 'let) (new-let-exp (map car (cadr input)))
      (map parse (map cadr (cadr input)))
      (parse (caddr input))]
      ```

   b. Assume that our let-exp data type has getters (let-syms tree) (let-vals tree) (let-body tree). You don’t need to write those. The (eval-exp tree env) procedure in the interpreter has a line for evaluating let-expressions. It starts

      ```
      [(let-exp? tree) ...........]
      
      Give this line of code. If you want to call some helper functions give them as well.
      ```

      ```
      [(let-exp? tree) (eval-exp (let-body tree)
      (extended-env
      (let-syms tree)
      (map (lambda (t) (eval-exp t env)) (let-vals tree))
      env))]
      ```
6. We implemented letrec in lab 8 by translating letrec expressions into equivalent expressions that don’t use letrec. **Give the equivalent expression for**

\[(\text{letrec } ([a \ 3] [b \ 4] [c \ 5]]) (* \ a \ (+ \ b \ c)))\]

Note: I realize this doesn’t need a letrec but I’m trying to make this easy to write out. Give the nested let-expression that your parser converts this letrec into.

\[
\begin{align*}
& (\text{let } ([a \ 0] [b \ 0] [c \ 0]) \\
& \quad (\text{let } ([g_1 \ 3] [g_2 \ 4] [g_3 \ 5] )) \\
& \quad \text{(begin} \\
& \quad \quad \text{(set! } a \ g_1) \\
& \quad \quad \text{(set! } b \ g_2) \\
& \quad \quad \text{(set! } c \ g_3) \\
& \quad \quad (* \ a \ (+ \ b \ c))))
\end{align*}
\]
7. I want to add a new kind of expression to miniScheme. If x is any variable in the environment, (inc x) should add 1 to the value of x and return the new value. So the code (let ([x 5]) (inc x)) returns 6 and (let ([bob 4] (* (inc bob) (inc bob)))) returns 30 (since the first expression (inc bob) returns 5 and the second (inc bob) returns 6.

a. Suppose we have a datatype inc-exp that holds one symbol (the variable being incremented); for this datatype we have constructor (new-inc-exp sym), recognizer (inc-exp? x) and getter (inc-sym tree) that returns the symbol stored int the inc-exp. **Give one line of code that extends the function (parse input) to handle the case where input is an inc-expression.** It starts

```
[(eq? (car input) 'inc) (new-inc-exp .......)]
```

```
[(eq? (car input) 'inc) (new-inc-exp (cadr input))]
```

b. Now give a line that extends (eval-exp tree env) in the case where tree is an inc-exp. This begins

```
[(inc-exp? tree) .......]
```

```
[(inc-exp? tree) (let ([var (lookup env (inc-sym tree))])
  (begin
    (set-box! var (+ 1 (unbox var)))
    (unbox var)))]
```