1. Suppose we have defined a let-exp datatype with constructor
(new-let-exp binding-symbols binding-vals body), recognizer (let-exp? tree) and getters
(Binding-Symbols tree), (Binding_Vals tree) and (Let-body tree). Give code to parse a let-
expression into such a let-exp tree.

(define parse ) (lambda (exp)
   (cond
      ....
      [(eq? (car exp) 'lambda) (new-let-exp (map car (cadr exp))
             (map parse (map cadr (cadr exp)))
             (parse (caddr exp))))])
2. I have parsed a lambda expression into a tree datatype (new-lambda-exp syms body); the lambda-exp type has getters (Lambda-Syms tree) and (Lambda-Body tree). Give code for the case of (eval-exp tree env) that evaluates a lambda-exp tree. If you need additional datatypes just describe them; you don’t need to give code defining the datatypes.

```scheme
(define eval-exp (lambda (tree env)
    (cond
        ....
    ([lambda-exp? tree] (new-closure (Lambda-Syms tree)
                                        (Lambda-Body tree)
                                        env))
```
3. a) Explain in one sentence, or two if you must, the difference between call-by-value and call-by-reference.
   b) Give an example that evaluates differently in call-by-reference than in call-by-value.
   c) Explain in English how you would modify your miniScheme interpreter to implement call-by-reference.

   a) In call-by-value the values of the arguments are passed to the procedure while in call-by-reference the addresses of the arguments are passed.

   b) The big distinction between the two comes when there is an assignment to a parameter:
      (let ([x 2])
       (let ([f (lambda (y) (set! y 23))])
        (begin
         (f x)
         x)))
      With call-by-value this returns 2 while with call-by-reference it returns 23

   c) To implement call-by-reference you need to do 2 things. Remember that the arguments are variables, so they are represented in the environment by boxes. Don’t unbox them. Second, when extending the environment in apply-proc, don’t rebox the parameters (which are already boxes).
4. We talked in class about the difference between static scoping (or static binding) and dynamic scoping. Your miniScheme interpreter implements static scoping. Give code for the non-primproc case of apply-proc that will change the interpreter to dynamic scoping. You can make any assumptions you want about datatypes.

    (define apply-proc (lambda (p arg-values env)
        (cond
            [(prim-proc? p) .....]
            [(closure? p) (eval-exp (Body p) (extended-env (Params p) arg-values env))]
        )
    )

Note that, unlike the version in your miniScheme interpreter, this takes the current environment as an argument when it is called by eval-exp.
5. We added letrec expressions to miniScheme by translating them into equivalent let-expressions.

Give a let-expression that is equivalent to the following letrec:

(letrec ([f (lambda (n) (if (zero? n) 1 (g (- n 1)))]
  [g (lambda (n) (if (zero? n) 2 (f (- n 1))))])
  (f 23))

(let )(f 0) (g 0))

(let ([f1 (lambda (n) (if (zero? n) 1 (g (- n 1)))]
  [g1 (lambda (n) (if (zero? n) 2 (f (- n 1))))])
  (begin
    (set! f f1)
    (set! g g1)
    (f 23))))
6. I want to add streams to miniScheme. If you recall, streams are based on the \textit{delay} and \textit{force} operations. In real Scheme once an expression is forced its value is remembered so it never needs to be evaluated again. For this question don’t worry about this remembering. I just want to implement \texttt{(delay exp)} so it delays the evaluation of \texttt{exp}, and to implement \texttt{(force exp)} so if \texttt{exp} is a delayed expression \texttt{(force exp)} evaluates it.

\textbf{a)} What code would you use to parse \texttt{(delay exp)} and \texttt{(force exp)}?
\textbf{b)} What code would you use in \texttt{(eval-exp tree env)} for the trees that come from parsing \texttt{(delay exp)} and \texttt{(force exp)}?

\begin{verbatim}
a) (define parse (lambda (exp)
    (cond
        
        ;; rest of the cond body

    b) (define eval-exp (lambda (tree env)
    (cond
        
        ;; rest of the cond body

\end{verbatim}
[This is extra space. If you want me to read what is here, please indicate that clearly.]
Please write and sign the Honor Pledge when you are finished with the exam.