1. Static and dynamic binding
   a. Explain in English what static binding and dynamic binding mean. One sentence each should be sufficient.

   Static binding: Free variables in a function get their values from the environment where the function is created.

   Dynamic binding: Free variables in a function get their values from the environment where the function is called.

   b. Give an example that gives different values in the two binding systems.

   (let ([A 0])
     (let ([f (lambda (x) (+ x A))])
       (let ([A 20])
         (f 5))

   This evaluates to 5 with static binding, 25 with dynamic binding.

   c. Explain in English how you could change your implementation of MiniScheme to use dynamic binding

   We need two steps:

   i. Give apply-proc an environment argument and call it with the current environment.
   ii. Evaluate the function body in an extension of the current environment rather than an extension of the closure environment (the extension binds the function parameters to the argument values, as we have always done.)
2. Closures
   
a. What is a closure? One sentence is probably enough for this.

   A closure is the value of a lambda expression. It is a triple consisting of the parameters of the lambda expression, its body, and the environment at the time the lambda expression is evaluated.

b. What is the point of a closure? To call a function such as (lambda (x) (+ 3 x)) with an argument such as 5, we just need to know the parameter of the function: (x), the body of the function: (+ 3 x), and the value of the argument: 5. Why do we ever need anything else?

   Closures allow us to implement static binding.
3. We talked about 3 parameter-passing protocols: call-by-value, call-by-reference, and call-by-name. Give a brief sentence explaining what each of these means.

   Call-by-value: The values of the arguments are passed to the function and used for bindings of the parameters.

   Call-by-reference: The addresses (or boxes) of the arguments are passed to the function.

   Call-by-name: The text of the arguments are passed to the function.
4. State
   a. Explain in English (one or two sentences) what we mean by code having state.

      An expression has state if it can evaluate to different values over time, reflecting its or its system’s history.

   b. Give any example of code that doesn't have state, and any example of code that does have state.

      No state: (+ 2 3) This always evaluates to 5.

      State: (let ([val 0])
               (lambda (x)
               (begin
                  (set! val (+ val x))
                  val )))

      If we define foo to be the value of this expression, then (foo 5) is 5; if we call (foo 5) again it gives 10.
5. To implement let-expressions in Lab 6 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 exp) 'let) (new-let-exp ........)]
`

Give this line of code. If you want to call some helper functions give them as well.

```
[(eq? (car exp) 'let) (new-let-exp (map car (cadr exp))
  (map parse (map cadr (cadr exp)))
  (parse (caddr exp)) )]
```

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. In your MiniScheme implementation you implemented the conditional expression \((\text{if } a \ b \ c)\) as a type of expression. We could also have implemented it as a primitive procedure, by giving

\((\text{apply-primitive-op } \text{op } \text{arg-values})\) a line that says:

\[[\text{(eq? } \text{op } 'if) \text{ (if } \text{(eq? } 'False \text{ (car } \text{arg-values})) \text{ (caddr } \text{arg-values}) \text{ (cadr } \text{arg-values)))}]\]

In other words, if the first arg-value is ‘False return the third arg-value, otherwise return the second arg-value. (For simplicity I am ignoring the possibility that the condition evaluates to 0).

So we could implement if as a kind of expression or as a primitive procedure. Either explain why it doesn’t matter which implementation the MiniScheme interpreter uses, or give an example that works differently in the two implementations.

It matters. You can’t implement a recursive function if \(\text{if}\) is implemented as a procedure (and so always evaluates both of its args. But here is a simpler example:

\((\text{define forever } (\lambda (x) (\text{forever } x)))\)
\((\text{if } 'True 1 (\text{forever } 1))\) evaluates to 1 if \(\text{if}\) is implemented as a kind of expression, and recurses forever it \(\text{if}\) is implemented as a primitive procedure.
7. We parsed letrec expressions into equivalent code that doesn’t use letrec. Give the equivalent expression for

\[
\text{(letrec } ([\ f1 \ (\lambda n) (\text{if (equals? n 0) 2 (f2 (- n 1))})])
\text{ [ f2 (\lambda n) (\text{if (equals? n 0) 1 (f1 (- n 1))})]]) \text{ )}
\text{ (f1 45))}
\]

\[
\text{(let } ([f1 0] [f2 0])
\text{ (let } ([g1 (\lambda n) (\text{if (equals n 0) 2 (f2 (- n 1))})]
\text{ [g2 (\lambda n) (\text{if (equals n 0) 1 (f1 (- n 1))})]]) \text{ )}
\text{ (begin}
\text{ (set! f1 g1)
\text{ (set! f2 g2)
\text{ (f 45))))}
\]