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

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

   c. Explain in English how you could change your implementation of MiniScheme to use dynamic binding.
2. Closures
   a. What is a closure? One sentence is probably enough for this.

   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?
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.
4. State
   a. Explain in English (one or two sentences) what we mean by code having state.

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

   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.
6. In your MiniScheme implementation you implemented the conditional expression \( (\textbf{if} \ a \ b \ c) \) 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:

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

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.
7. We parsed letrec expressions into equivalent code that doesn’t use letrec. Give the equivalent expression for

\[
\text{letrec } ([ f1 (lambda(n) (if (equals? n 0) 2 (f2 (- n 1))))]
\text{  [ f2 (lambda(n) (if (equals? n 0) 1 (f1 (- n 1))))] )
\text{  ( f1 45))}
\]
You can use this page as extra space for any question on the exam.

Please write and sign the Honor Pledge when you have finished the exam.