1. What does \((\lambda (x \ y) (+ x y))\) parse to? What does it evaluate to in environment \(env\)? In other words, if this expression parses to \(tree\) then what is \(tree\) and what is \((eval-exp tree env)\)? Your answers to these need some datatypes. You can either give code for the datatypes or explain in English what fields they contain.

\[
\begin{align*}
tree &= (\text{'parse-exp} (x \ y) (\text{application-exp} (\text{varref} +) ((\text{varref} x) (\text{varref} y)))) \\
(eval-exp tree env) &= (\text{'closure} (x \ y) (\text{application-exp} (\text{varref} +) ((\text{varref} x) (\text{varref} y))) \text{ env})
\end{align*}
\]
2. How could we extend MiniScheme to allow for unrestricted lambdas? For example, we might want to write

\[
\text{(let ((sum (lambda nums (apply + nums)))))}
\]

\[
\text{(sum 1 2 3)}
\]

This expression should evaluate to 6. **How would you parse an unrestricted lambda?**

**How would you evaluate it in environment env?** Your answer can be in English or in code, but you should give enough detail that I could implement it.

There are many ways to answer this. It is important to make a distinction between evaluating a lambda expression and calling the expression.

I would parse `(lambda nums (apply + nums))` as

\[
\text{(lambda-exp nums (application-exp (varref apply) ((varref +) (varref nums)))})
\]

and evaluate it as

\[
\text{(closure nums (application-exp (varref apply) ((varref +) (varref nums))) env)}
\]

Those are both the same as you would get from parsing and evaluating a normal lambda. You can tell these come from an unrestricted lambda because the parameter is an atom instead of a list. The difference in calling an unrestricted lambda comes in apply-exp, and the question doesn’t ask about that.
3. We implemented `if` as a kind of MiniScheme expression:
   
   (if <condition> <true-branch> <false-branch>).
   
   We could also have implemented `if` as a primitive procedure with 3 arguments. **How would primitive procedure `if` act differently than an if-expression?**
   
   Primitive procedures (like all procedures) evaluate all of their arguments when they are called. Our if-expression evaluates the condition and only one of the two branches.
4. a) Give an example of an expression that has a different value under dynamic binding than under static binding. Say what the expression evaluates to under each binding scheme.

The binding schemes determine where the unbound variables in a function get their values, so any example needs to define a function with an unbound variable. For example:

```
(let ([A 0])
  (let ((f (lambda (x) (+ x A))))
    (let ([A 23])
      (f 10))))
```

With static binding the A in function f has value 0, and (f 10) evaluates to 10.
With dynamic binding the A has value 23, and (f 10) evaluates to 33.

b) Give an example of an expression that has a different value under call-by-reference than under call-by-value. Say what the expression evaluates to under each scheme.

With call-by-reference the parameter becomes another name for the argument; the example needs to be something where this (rather than the parameter getting the value of the argument) makes a difference.

```
(let ([A 0])
  (let ((f (lambda (x) (set! x 10) )))
    (begin
      (f A)
      A)))
```

With call-by-value f has no way to change A so the expression evaluates to 0.
With call-by-reference x becomes another name for variable A and the call to f changes the value of A to 10, so the expression evaluates to 10.
5. We went to a lot of trouble to implement letrec. Why did we need to do that? Explain why the following expression gives an error in standard Scheme:

\[
\text{(let ([f (lambda (x) (if (= x 0) 1 (* x (f (- x 1))))])
  (f 5))}
\]

When the let-expression is evaluated the lambda expression is evaluated in the top-level environment. The lambda expression evaluates to a closure with the top-level environment as the closure environment. The let expression extends the top-level environment with a binding of \( f \) to this closure. Call this extended environment \( E \). We can call \( (f 5) \) in \( E \) because \( f \) is bound to a closure in \( E \). In executing that call we extend the closure environment (which we said is the top-level environment) with a binding of \( x \) to the argument 5. Call this environment \( E_1 \); we evaluate the body of \( f \) in \( E_1 \). However, this fails when we get to the recursive call because \( f \) is not bound in \( E_1 \).
6.  
   a) **What do people mean when they say an expression has “state”?**  
   Your answer to this can be as short as one sentence.

   An expression has state if its value can change over time as a result of the history of the system it is part of.

   
   b) **What would be different in the interpreter project if MiniScheme was stateless?**

   If MiniScheme was stateless it wouldn’t have set!, whose primary role is to create state. Without set! we would have no need for boxes. Without state there would be no reason to use a begin-expression, since begins would always return the value of their last expression.
7. I would like to add cond expressions to MiniScheme. Here is a typical cond expression:

```
(cond
  [(< a b) (- b a)]
  [(< b a) (- a b)]
  [else 0])
```

So the word ‘cond is followed by a sequence of cases. Each case is a pair. If the first element of the pair is the word ‘else we return the value of the second element of the pair. If the first element of the pair evaluates to ‘True we return the value of the second element of the pair. Otherwise we go on to the next case. If there are no more cases just return null. It is easy to parse such an expression into (‘cond-exp cases) where cases is a list of pairs of parse trees. You can assume we have a datatype cond-exp with recognizer cond-exp? and getter cond-cases, and that the parsing is already completed.

**What code would you add to eval-exp to evaluate a cond-exp in environment env?**

Add a line to (eval-exp tree env):

```
[(cond-exp? tree) (cond-helper (cond-cases tree) env)]
```

where we define cond-helper as

```
(define cond-helper (lambda (cases env) _
  (cond
    [(null? cases) null]
    [(eq? ‘else (car cases)) (eval-exp (cadar cases) env)]
    [(eq? ‘True (eval-exp (car cases) env)) eval-exp (cadar cases) env]]
    [else (cond-helper (cdr cases)))]))
```