Part Ⅰ: Theoretical Questions

1. Let expression is not special form because in L3 it is syntactic sugar. Any let expression is re-written as an app expression before it even gets to the interpreter. Therefore, there is no evaluation rule for let expression at all.  
   A syntax construct is a special form only if it needs special treatment from the interpreter (read: an evaluation rule made especially for it).  
   And as we said, in our case, it doesn’t since it has no evaluation rule.
3. Error 1: Referring to an undefined or undeclared variable.  
   For example, the following L3 program: (+ x 5)
4. Error 2: Applying a procedure with an operator that is not prim-op and not a closer  
   Example, the expression: (1 2)
5. Error 3: Applying a closure with different amount of arguments than it expects.  
   For example, the expression: (-)
6. Error 4: Applying a closure on arguments of types different than it expects.  
   For example: (+ #t 2)
   1. We’ll change the definition of CExp as follows:  
      CExp = AtomicExp | CompoundExp | Value  
      Where Value is the definition from L3-value.ts:  
      Value = number | boolean | string | PrimOp | Closure | SymbolSExp | EmptySExp | CompoundSExp

* 1. We would make a few changes:  
     We’ll first remove the only usages of valueToLitExp and instead just pass the values we calculated straight to substitute as is.  
     Now, we’ll also have to update the function substitute itself.  
     We’ll have to add a case in the sub-function sub for each type of value:  
     For number, string, boolean, EmptySExp, and SymbolSExp, we’ll just return them as is.  
     For CompoundSExp, we’ll just recurse down to both it’s values and then make a new CompoundSExp with both values when all of the free vars got substituted with the values.  
     For Closure, we’ll act very similarly to how we handle ProcExp.
  2. We think the better option is to use valueToLitExp.  
     That’s because we feel like changing the AST to allow values in it breaks separation between the layers of the AST and interpreter and increases the coupling between them, making it harder to make further changes in one without breaking and also changing the other (particularly the direction from the interpreter to the AST, as the changing the AST is very much likely to impact the interpreter anyway).  
     It also lowers the cohesion of the AST as now it does not always represent an L3 Program as it may contain values which live only at program runtime and die afterwards.

1. Because in normal evaluation, we substitute arguments in lambda bodies with CExps rather than values, in contrast to applicative eval where we first calculate the value of the CExp and then we have to convert the value back to an expression. The reason we have to use a CExp is because is because the function substitute expects CExps and not values.
2. A program where normal eval executes faster:  
   (define f (lambda (x) 5))

(f (+ 3 5))  
  
That’s because in normal evaluation, we have no need to evaluate the expression (+ 3 5), while in applicative eval we will do so anyway.

A program where applicative eval executes faster:

(define f (lambda (x) (+ x x)))

(f (+ 3 5))

That’s because in applicative eval we will evaluate the expression  
(+ 3 5) only once, while in normal evaluation we will pass the argument as an expression, and therefore evaluate it twice

Part III: Normal Environment Evaluator

3.1) the problem is that In your normal evaluation implementation variables are mapped to Values and not to CExp. Therefore, when creating bindings, you have to fully evaluate the expression instead of binding the variable to the expression itself.

3.2+3.3) We implemented the bonus. We did not change the type Closure at all.  
We mapped each variable to a new value type which we called ‘Promise’ which holds an Cexp and the enviroment it should be evaluated in.