Part 1:

1. No, because L3's interpreter doesn't directly support the evaluation of let expressions, but rewrite let expressions in applications, so let expressions actually evaluate like application expressions.

2. No, closures are not created during the evaluation of a let expression, because it would result in false variable capture effects if we tried to do so.
   This is based on the strategy of evaluating let as a normal function.

3. The interpreter must be able to detect errors at runtime and report them correctly.
   1. dividing by 0.
   Example: (/ 5 0)

   2. The expected number of arguments did not match the specified number expected
   Example: expected 2 got 3.
   Closure: ((lambda (x y) (+ x y)) 4 5 6)

   3. Apply a primitive operator on incompatible types:
      Example: (+ “n” 7)

   4. Trying to apply something that isn’t a primitive operator or a closure in an application:
      Example: here we tried to apply 5 on 6 and 7, which is obviously bad.
      (5 6 7)

4. Purpose:
   Maps evaluated arguments to expressions to ensure that the result of the substitution is a well-typed AST that can be evaluated.

   the problem that it solves is:
   Regarding the system architecture, the system architecture consists of different types, so adding VALUE mixes types, which can confuse users of the interpreter, especially when working with code. So when we have a problem, we can't know if it's a value problem or a syntactic or semantic problem, so valueToLitExp makes debugging easier.

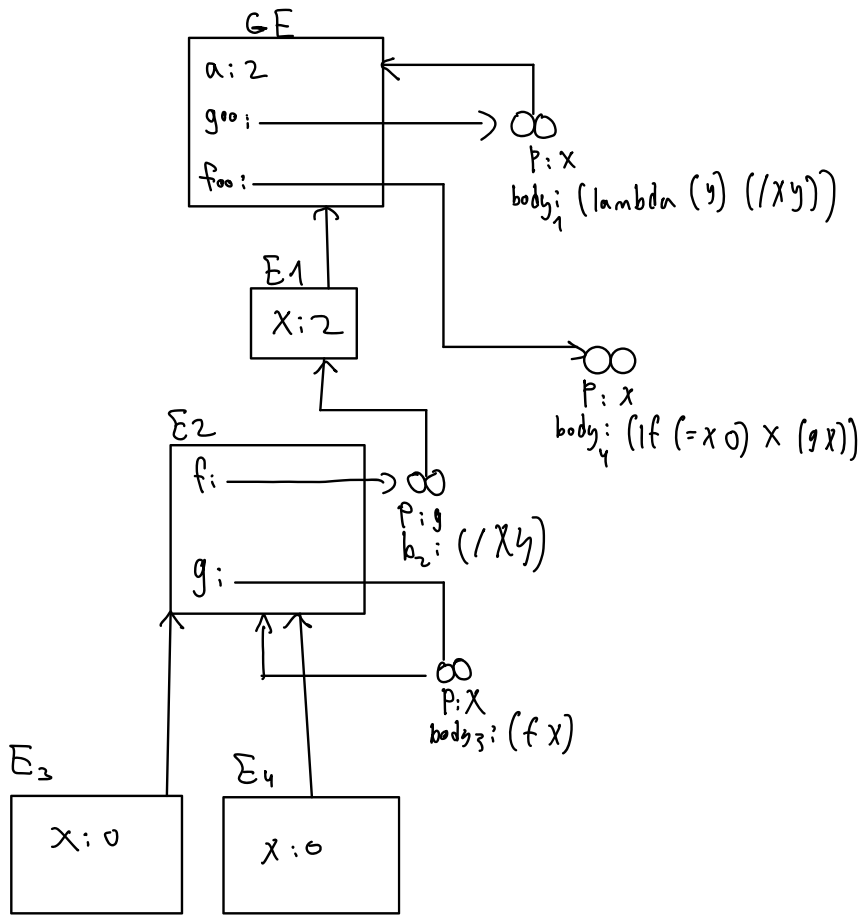
5. The normal evaluation strategy interpreter doesn't need the valueToLitExp function because the arguments are not evaluated until passed to the closure. So applyClosure takes CExps as a parameter and replaces varRefs in the body with CExps - which is correct according to the type definition of CExp .

6. Special forms require special evaluation rules according to special operators. On the other hand, for primitive operators, the evaluation is as follows:
   (a) Evaluate all subexpressions.
   (b) apply the procedure that is the value of the first subexpression to
   other subexpressions. For custom programs, the app includes
   Replace procedure parameters in procedure body expressions with parameter values. The value returned is the value of the last expression
   in the body.

7. The main reason for the change is that substitution requires an iterative analysis of the body of the program. Operations on the AST duplicate the structure of the entire AST, resulting in large memory allocations.

   Example: ((* x ((lambda (x) (* x 7) 13)) ((lambda (x) (+ x x)) 5)

8. Diagram:


GE
a: 2
goo:
foo:
p: x
body1: (lambda (y) (/ x y))
E1
x: 2
p: x
body4: (if (= x 0) x (g x))
E2
f:
g:
p: g
b2: (/ x y)
p: x
body3: (f x)
E3
x: 0
E4
x: 0