Part 1

1. Let is a special form - because he is not evaluated like regular compound expression.
   Compound expression is evaluated by the rules:
   a) evaluate the sub-expressions recursively
   b) apply the value of the operator on the values of the operands.

2.
   a) Applying wrong parameter types to expression: (* #f #t)
   b) Evaluating free variable: (+ x 5) – when this is our whole program.
   c) Errors while computing expression: (/ 5 0)
   d) Evaluation of compound expression when his operator is not of compatible type: (1 2)

3. 1. We can change the rule from:

```
type CExp =  AtomicExp | CompoundExp;
to:
type CExp =  AtomicExp | CompoundExp | SExp;
```

2.The change in part 1 of this question will be enough – we will simply remove the function valueToLit in the makeClosure function (turning the value to CExpiration) since now they are recognized as valid CExps and the substitute function will work correctly.

3. First option is to add option adds the SExp to the interpreter as a CExp so we don't have to check special cases. This makes the AST a more complicated but simple for understanding.

Other way will be not changing the AST instead we will make a spatial case in makeClosure, this was learned in class and it is a little more complicated.

In my opinion I think it does't really matter with way we will use as both lead to the same result, I personally like the easy implementation so I would peek the first way.

4. Because we don't place values but exp and we compute only when we need it.
5.

Normal faster → (L3 (define loop (lambda (x) (loop x)))

(define g (lambda (x) 5))

(g (loop 0)))

Because applicative will enter infinite loop and never ends.

Applicative faster →

(define square (lambda (x) (* x x)))

(define sum-of-squares (lambda (x y) (+ (square x) (square y))))

(define f (lambda (a) (sum-of-squares (+ a 1) (* a 2))) (f 5)

Normal eval computes several times the same expression, like (5 * 2)

While applicative computes them only once.

Part 3)

We did the bonus, and were not sure if we need to mention it here.

In

#lang lazy

(define x (-)) x

We do not evaluate expirations unless we have to in lazy or normal eval so running the given program with the #lang lazy flag we do not eval x, the result we get is #<promise:x> - a promise to give a value in the future.

#lang lazy

(define x (-)) 1

Again with the lazy flag the evaluation is delayed till its application is needed. But this time we do not request the value of x only the value 1 with is simply 1 – x is never called.

In our program the problem is that when handling a define expression we evaluate the value of the define expression in order to add it to the environment when evaluating the program.

Instead we should link the var of the define with its value without the computation in other words add it to environment as CExp.