1. yes let is a special form, because it is defined as (let ((<var> <exp>)*) <body>) and its value is computed in a special way. L3 interpreter has to rewrite the Let expression into an App expression, i.e there is a special rule specifically for let expressions so obviously it is not an ordinary form expression.
2. The role of the function valueToLitExp is to take a v of type value and convert it to a Cexp according to its type. This is needed since there is a typing problem in the substitution process: the body of the closure is a list of CExp expressions.The purpose of the substitution is to replace a varRef with its value, but if we replace the varRef with its value, the resulting body of the closure isn't a valid AST, so we use the valueToLitExp function to resolve this problem by mapping the values of the arguments to the corresponding expressions.
3. The value2LitExp function isn't needed in the normal order strategy interpreter because the literal expression isn't evaluated right away, as the normal order just substitutes each varRef in the closure with the body of this var, without evaluating right away the value.
4. The value2LitExp isn't needed in the environmental method because when the interpreter evaluates a closure in this method, it searches for each varRef value by searching in the frames in the environment for the first value of that varRef, so Substituting isn't relevant.
5. A reason to switch from applicative order to normal order would be to avoid crashes or infinite loops, as for some code the applicative order may get stuck in an infinite loop while the normal order wouldn't. this can't happen the other way around.
   example: (the normal order won't reach (f 0) because there's no need to evaluate it)
   (define f (lambda (x) (f x)))
   (define g(lambda (x) 5))
   (g (f 0))
6. A reason to switch from normal order evaluation to applicative order would be efficiency, because the normal order doesn't compute the value of an expression until it is needed.For example:
   (lambda (x) (* x x)) (+ 1 2)
   the normal order will replace x with (+ 1 2), and then compute it's value twice, while the applicative order would compute it only once.
7. a) In general substitution requires renaming. However, when the term that is substituted is "closed" then no renaming is required and naive substitution is correct. The reason why renaming is required is because when we want to evaluate an exp we need to substitute the varDecl with the appropriate Exp, and naive substitution may cause for a non equivalent program if there is a free variable with the same name as another variable somewhere in the code, for example:
   (define x 5)
   ((+ x ((lambda (x) (+ x 3)) 4)))
   in the naive substitution method this may be renamed to the following:
   (define x 5)
   ((+ x1 ((lambda (x1) (+ x1 3)) 4)))
   but this is obviously not equivalent to the original program because the first x in the second line is bounded to the {x=5} exp, and the other x's are bounded in the lambda expression.

Therefore, in a term with only bounded variables, no such problems may occur because each exp is independent from the others, so a naive substitution will do just fine.

b) The following algorithm is a general algorithm for the naive substitution method:

for each exp:

evaluate exp

for each varDecl x in exp: substitute x with its binding.

8.