Question 1

Yes, the interpreter of L3 language have a special rule to evaluate let expressions where it rewrites all let expressions onto application expression before evaluating them.

Question 2

The valueToLitExp function converts a value to a literal expression. We have to do so in the substitution model (In particular, in the applicative order) since we replace all occurrences of the varRefs with their appropriate values (after computing them). Since we are dealing with ASTs, we can not replace varRef with value, so we need to convert them onto literal expressions first so they can fit the AST.

Question 3

In the normal evaluation we do not compute the arguments before passing them to the closures, instead we replace all varRefs with the appropriate CExps, and since CExp can fit the ASTs, we do not need to convert it.

Question 4

In the environment-model interpreter we do not modify the AST created by the parser, so we do not need to convert values to literal expressions.

Question 5

- Avoid infinite recursions.
  Example:
  ```
  (define loop (lambda (x) (loop x)))
  (define f (lambda (x) 5))
  (f (loop 0))
  ;;(loop 0) is assigned to x in f without computing it
  ```
- Avoid runtime errors (such that arithmetic errors).
  Example:
  ```
  ((lambda (x y z) (if x y z))
  #t 5 (\ 5 0))
  ;;we do not reach the computation of (\ 5 0)
  ```
- We do not use the valueToLitExp function.
- Faster evaluation.
  Example:
  ```
  ((lambda (x y z) (if x y z))
  #t 3 (\ 30 3 2))
  ;;not all operands are computed
  ```

Question 6

- Faster evaluation (when there is repeated occurrences of the same variable).
  Example:

```
((lambda (x) (+ (* x x) (* x x))) (+ 3 4))
;; in the applicative evaluation we only compute (+ 3 4) once and replace all
;; occurrences of x with 7, while in the normal evaluation we replace all x's with
;; (+ 3 4) and thus we compute (+ 3 4) 4 times instead.
```

Question 7

If the term we are dealing with is a closed term, then we do not have free variables and all our variables are bound variables.

Previously we renamed the variables which appeared in the term so that we do not substitute a free variable with an operand/argument which is associated to a parameter with the same name of that variable.

Since we do not have free variables, we can change all appearing variables in the term according to the operands given and we will not do any mistake since all variables which appear in the term are bound variables (all the variables appear in the term are actually parameters of the procedure we are substituting its body!).

Evaluation rules

When we come appExp (we consider let as syntactic sugar) we do the following:

Applicative order:

1- Get a list of all parameters of the operator.

2- Compute all the operands and get list of values.

3- Convert all the values we computed to LitExp using the valueToLitExp function and get a list of LitExp.

4- Use the substitute function to replace all the app parameters to the Lit Expressions computed in step 2 (according to where they appear, for example if the operator's parameters are {x,y} and the operands are {NumExp(1), NumExp(2)), then x= NumExp(1) and y= NumExp(2)).

Normal Order

1- Get a list of all parameters of the operator.

2- Save all the operands given in a list.

3- Use the substitute function to replace all the app parameters to the operands from step 1 According to their order (according to where they appear, for example if the procedure's parameters are (x,y) and the operands are (1,2), then x=1 and y=2).

Question 8