PPL Assignment 3

Part 1: Theoretical Questions

1) The 'Let' expression isn't a special form in L3 as the difference of L2 and L3 are the ability to take care of "Local variables" and "Compound values" , so ,
   we can make every let as "define" and "lambda" special forms .
   That means, Let isn't a special form but syntactic abbreviation .
2) The role of the function ***valueToLitExp*** is to solve the typing problem that occurs in the substitution model when executing a function. Before executing the function the arguments are evaluated, i.e. arguments are now represented as a value. But when substituting the arguments with their value within the body of the function, a problem occurs: the body of the function is of the expression type, while the values are of the value type. To solve the problem, we convert the values to their literal expression, meaning that the arguments will again be of expression type and there is no typing problem :)
3) The valueToLitExp function is needed when we are about to substitute variables with values, which contradict the expected type for AST nodes - expression rather than values. In normal evaluation, in contrast, we make the substitution before the evaluation of the arguments. Therefore this function is not needed.
4) Same as is in normal evaluation, we are not substituting values with variables. There is no substitution at all and the typing problem doesn't occur. Hence this function is not needed.
5) The reasons that would justify switching from applicative order to the normal order evaluation is describe as follow :
   This risk of non termination, even when the guilty argument will never be used in the call, is one of the reasons to use normal order evaluation instead .
   Another option for switching the forms is i.e a divide by 0 .
   While in 'normal order' it passes subexpressions as they are, without evaluation, and proceeds with the evaluation only when the corresponding formal parameter is actually to be itself evaluated , while in applicative order the opposite is happen , as every subexpression getting evaluation right after it ready to get evaluated , and that means there is chance to get an error .
6) The reasons that would justify switching from normal order to the applicative order evaluation is describe as follow :
   While normal-order evaluation may result in doing extra work by requiring function arguments to be evaluated more than once, applicative order may help us to reduce the number of evaluations , i.e. lambda (x)-> x+x , given 5+6 .
   In that case, applicative will make (5+6) , and then (11+11) , while normal order will do (5+6)+(5+6) , which is more "expensive" .

7) **a.** As seen in class , renaming in substitution is required .
But , if the term is “closed” , we don't need to do so .
The reason we don't have to rename those variables , is the fact that when the interpreter is looking for this variable he will find it in the procedure he is at the moment , and no “mistake” such finding free-variable can exist in that case .

b. We will explain the naive substitution rules :
1) if primitive -> make Evaluation .
2) if Compound -> make Evaluation for each variable .
3) search for the bound-variables to make VarRef if needed.
4) complete the Evaluation .
The problem in this naive substitution is that we might capture the free-variable , which we wanna avoid of course .
For example ,
we learned in lectures that in this program :

```
(define z (lambda (x) (* x x)))
(((lambda (x) (lambda (z) (x z)))  //Line 1
 (lambda (w) (z w)))  // Line 2
```

x from Line 1, will get (lambda (w) (z w)) from Line 2.
and then z will capture by the “lambda (z)” in Line 1
and not by the global “define z” that we tried to make .

8)