Part 1

**q.1**

In L3, the let expression is replaced by lambda, a special form which is syntactically equivalent to let expression. Lambda () is a special form with its own evaluation rule. We rewrite let to lambda, therefore let uses lambda's evaluation rule and is not a primitive operator, but a special form in L3.

**q.2**

In L3 converts values (of type such as , , etc.) to literal expressions such as , , etc. It is used in the substitution which takes place in (when evaluating ). It is needed because when substituting variables, we need to replace them with expressions rather than values, which were evaluated beforehand because of the applicative order evaluation.

**q.3**

In normal evaluation strategy we evaluate expressions only when we need them. When evaluating we pass arguments as expressions (rather than values in applicative order) to , then substituting can be applied without the need to convert to back to expressions. So, in the case of normal evaluation strategy is irrelevant.

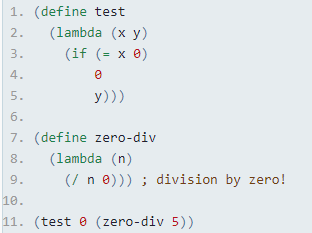
**q.4**

is a supporting function used to implement the substitution model. In environmental model we do not use substitution. (we compute according to relevant environment). So, in the case of environment-model interpreter is irrelevant.

**q.5**

In normal evaluation order unnecessary procedures arguments will not be evaluated.

We also saw on practical session 5 that some code can crash while using applicative order evaluation while normal evaluation order will not:

****

In the example above applicative evaluation order will return an error (division by zero), while normal evaluation order will return 0 and terminate.

In cases such as this it will be preferred to use normal order evaluation over applicative order evaluation.

**q.6**

**Consider the next code:**

**(define a (+ 1 2 3 4 5))**

**(define b (lambda (c) (+ c c c c c)))**

**(b a)**

The main difference between the normal order evaluation and that of the applicative approach is that in the applicative we evaluate the procedure arguments first, before applying the procedure. In the example above If we are calling the same variable many times, we can save computation time by evaluating it once over evaluating same variable multiple times, which what occurs in normal order. In cases such as this it will be preferred to use applicative order evaluation over normal.

**q.7a**

Consider the following program: (from class material)

(**define** z (**lambda** (**x**) (\* x x)))

(((**lambda** (**x**) (**lambda** (**z**) (**x** z))) *; 1*

(**lambda** (**w**) (**z** w))) *; 2*

2)

Let us assume that we will not use the renaming of bound variables.

So, in the 2nd expression x will be replaced (captured) by (lambda (w) (z w)) context in which we operated the substitution. As the result we get the next body:

(**lambda** (**z**) ((**lambda** (**w**) (**z** w)) z))

The inner var-ref z refers to the function (lambda (z)…) var-decl instead of referring to the global (define z …) var-decl.

Renaming the bound variables in the body of the procedure can fix that problem.

For example:

**(lambda (x) x) ==> (lambda (x1) x1)**

In other words, free variables without renaming can cause loss of the program integrity and make it work incorrectly. The problem is known as 'capturing free variables'. In closed term (w/o free variables) we do not have to deal with that problem, therefore we can skip the step of renaming without impairing the correctness of the program. In conclusion with closed term, we can perform the naïve substitution without renaming.

**q.7.b**

Generally, when referring to the solution using substitution, using the applicative evaluation order, we evaluated the , (which is used when evaluating ), according to this rules:

~~- Rename all the bound variables that occur in the body with "fresh" names.~~

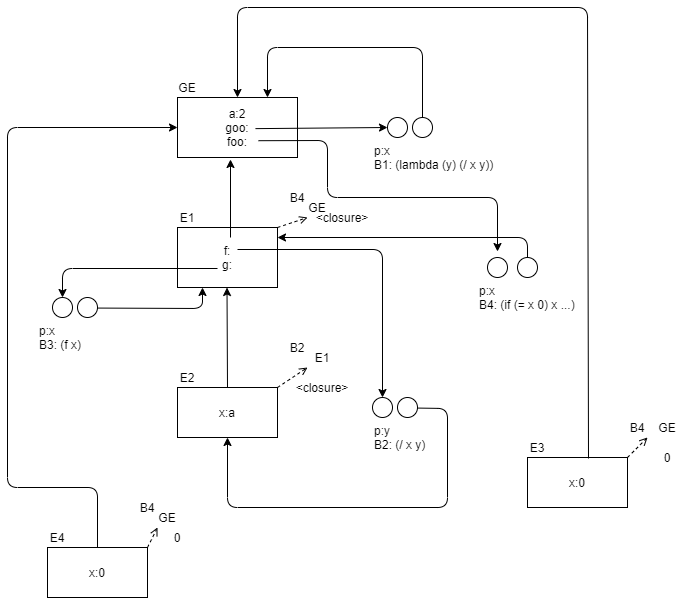
- Map arguments (Values) to lit-exps

- Perform the substitution

- Evaluate the new body according to the current environment

As we explained in part 7a, we can remove the first evaluation step**.**

**q.8**

****