Theoretical Questions

Part 1:

1) valueToLitExp purpose is to transform values into expressions to replace variables as SExpressions (compound expression consists only of SExpressions, SExpressions are expressions). The applicative order evaluation approach require evaluating the arguments prior to substitution, so it converts them from expressions into values and evaluates them. So in order to turn the values back into their original type after evaluation we use valueToLitExp.

2) In Normal Order Evaluation the values of expressions don't need to evaluate before substitution. It directly substitutes them with expressions without type errors.

3) 1. Applicative Order Evaluation.

2. Normal Order Evaluation.

4)

- Arithmetic error: $(/\ 5\ 0)$
- Incorrect arguments' number: $(lambda\ (x\ k\ u)\ (+\ x\ x)\ 5)$
- Type error: $(lambda\ (x)\ (+x\ x)\ \#t)$
- Undefined variable error: $(+5\ y)$

5) Special form: not all the parts of the compound form are always evaluated. The order in which the parts is evaluated is determined by the computation rule of the compound form type and which is different from normal compound expression.
Primitive operator: The value of a primitive operator expression is itself when we evaluate the expression.

6) It may be highly expensive, especially when using recursion, to scan the entire AST each time you substitute variables for terms. For example:

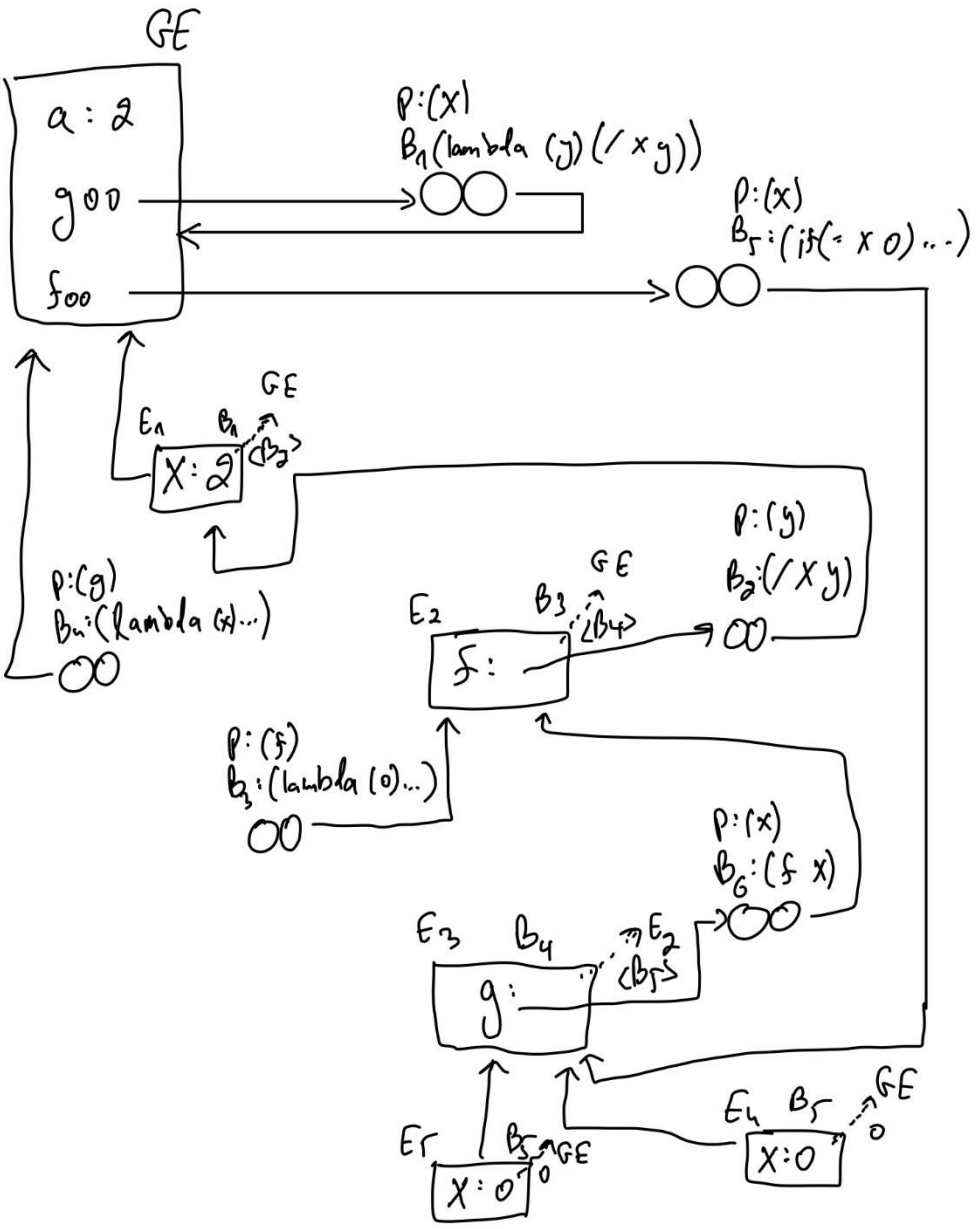
```
(define func (lambda (x)
    (if (= 1 x) 1 (+ (* x x)  (func (- x 1))))))

(func 3)
=> ((lambda (x) (if (= 1 x) 1 (+ (* x x)  (func
(- x 1))))) 3)
=> (if (= 1 x) 1 (+ (* x x) (func (- x 1))))
[x := 3]
=> (if (= 1 3) 1 (+ (* 3 3)  (func (- 3 1))))
=> (if #f  1 (+ (* 3 3)  (func (- 3 1))))
=> (+ (* 3 3)  (add (- 3 1))))
=> (+ 9 (func (2))))
=> ((lambda (x) (if (= 1 x) 1 (+ (* x x)
(func (- x 1))))) 2)
=> …
```

Observe how each recursive call requires the traverse of the whole AST of the function by substituting n for 4 in three separate places, shown in here:

```
(if (= 1 x) 1 (+ (* x x)  (func (- x 1))))
```

7) we use boxing to enable mutation and to properly model recursion and the global environment and before implementing environment using box all variables are immutable.

8)


GE
a : 2
goo
foo
p:(x)
B1(lambda (y) (/ x y))
p:(x)
B5:(if(= x 0)...)
E1
B1
<B2>
X:2
GE
p:(g)
Bn:(lambda (x)...)
p:(y)
B2:(/ x y)
E2
B3
<B4>
f:
GE
p:(f)
B3:(lambda (0)...)
p:(x)
B6:(f x)
E3
B4
E2
<B5>
g:
E5
B5
GE
X:0
0
E4
B5
GE
X:0
0

Part 2:

1.3) bound? Should be a special form rather than user function or primitive. As explained in question 5 in part 1 above, bound? Has a special and specific way of evaluation and executing. It can not be a user function or a primitive because it need an access to the current environment, while primitive don't have access and user function can only get the environment through its given arguments which would not have the required information to the current environment.

2.2) similar to the previews to question 1.3 in part 2, (time <cexp>) has to modify the evaluation process to return the desired output in L4. So user function and primitive can not achieve such thing, which means the only way is through using special form.