Theoretical questions: Assignment 3

1. Of course, Let is a special form in L3.
Non-special form expression evaluated by a general evaluation rule, and all the sub expressions are always evaluated. In Special form expression a special evaluation rule exists.
In Let, the binding variables are not evaluated.
The let expression only define local variables with the corresponding binding values and replaces every occurrence of the parameter in the body with the corresponding value assigned to it.

2. Four types of semantics errors are:
activate a closure with Incorrect number of parameters: ((lambda (x) (* x x)) 2 3)
activate an essentially wrong action as a primitive, like divide by zero: (/ 2 0)
activate an operator from non-primitive or non-closure expression: (10 5 7)
activate a primitive on a non-suitable type (type error): (* x x) when x is non-number

3.1: Updates in syntax: In L3-ast.ts file:

1) Import { Value } from './L3-value'
2) Add the Value expression to the CExp type: export type CExp = AtomicExp | CompoundExp | Value;

3.2: Updates in the interpreter: In L3-eval.ts file:

1)Import { isSExp } from "./L3-value";
2) In L3applicativeEval function:
   We want to check if the exp is a value, if yes return it with no change into result.
   Add the following line: isSExp(exp) ? makeOk(exp)
3) In applyClosure function:
   A. Transformation id not needed anymore, since Value is a legal expression in the AST tree.
   Delete the following line inside: const litArgs = map(valueToLitExp, args)
   B. Change the third input when calling the substitute function from litArgs to args as follows: substitute(body, vars, args).
4) Delete the valueToLitExp function.

3.3: Using the valueToLitExp function is preferable.
Since it keeps logic and order. In our opinion, the addition of Value to CExp is misleading and illogical, since Value is not an expression.
Moreover, its confusing to have those conceptual double meanings, like: NumExp and a number value, a BoolExp and a boolean value etc. We think it is better to stick with the expressions only.

4. The valueToLitExp function is not needed in the normal evaluation strategy interpreter because when we want to substitute var-ref with value in this model we get the values as c-exp which his legal expression in the AST tree so we don't need to convert the value when doing substitute.

5.

A. normal order will execute faster than applicative order evaluation:

```
(define if-square (lambda (x) (if(> 2 1) 1 (* x x))))
(define f (lambda (y) (if-square (* 2 y))))
(f 10)
```

In normal order we will not evaluate (* 2 10) at all, where in applicative order we will evaluate it once, therefore normal order is faster

B. applicative order evaluation will execute faster than normal order:

```
(define square (lambda (x) (* x x)))
(define f (lambda (y) (square (* 2 y))))
(f 5)
```

In normal order we will evaluate (* 2 5) twice inside 'square' body, where in applicative order we will evaluate it once, therefore applicative order evaluation is faster.

Part 3: Theoretical Questions

3.1

#lang lazy - (define x (-)) x

The following program return a #<promise:x> (a promise to supply the value at some point in the future).The program avoids evaluating the val argument in the DefineExp. Afterwards, calling to x, substitute x with (-), but the program still avoids evaluating that argument, since it is not necessary yet (it will be necessary, for instance, when we need to apply a primitive procedure).

#lang lazy - (define x (-)) 1

The following program return the value 1, since the program avoids evaluating the val argument in the defineExp, from the same reason we mentioned above. In the applicative evaluation, the val argument (-) will evaluate immediately and return an error since it expected at least one argument and given 0.

The code found in file test-define-normal.ts does not behave as the #lazy language because in the current implementation, define evaluate the (-) exp as part of the binding process between x and (-). Specifically, in evalDefineExps implementation if isDefineExp(def) is true, we first evaluate def.val, where in that point at the #lazy language we avoid evaluating def.val .

The solution would make the Env without evaluating def.val yet, by changing the evalDefineExps function, possibly by wrapping every appExp with a promise as implement in the #lazy language.