Part Ⅰ: Theoretical Questions

1. A special form is a compound expression that his sub-expressions are evaluated each one by his specific meaning and not all of them evaluated before the value of the exp is computed.
   let expression is a special form because we first test whether the evaluation of the bindings or body sub-expression returns an error.

   If it is the case, the whole expression is reduced to this error - without evaluating any of the sub-expressions bindings or body
   Only if the evaluation of the bindings and the body sub-expression is OK ,we will evaluate the whole expression.

2. Semantic errors happen in running time when we make an error in the expression semantics.
Those kinds are error will not be revealed in compilation time.

Types of semantic errors when executing an L3 program:
1.semantic errors can happen in expressions for example:
(+ 5 true)
; //in this case according to L3 rules the app exp will get rands of cexp kind so there will not be compilation error, but in running time we will get an error.
2.
(/ 10 0)
// in this case according to L3 rules the app exp will get rands of cexp kind so there will not be compilation error, but in running time we will get an error because we will try to divide by 0.
3.
(if 5 true 3)
//in this case according to L3 rules the if exp will get test of cexp kind so there will not be compilation error, but in running time we will get an error.
4.
(L3 (define z (* 6 14)) (* c x)) - when x is not defined
//in this case according to L3 the semantic rules are true, but when x is not defined we will get running time error.

3.

3.1- We will add **<value>::= <str-exp | <num-exp>| <bool-exp> | <prim-op> | <lit-exp> | <proc-exp>**

3.2- We will need to change the substitute to accept values in the arguments.

His signature will be

export const substitute = (body: CExp[], vars: string[] , exps: Values[])

Also we will need to add the interface Value and the functions makeValue and isValue

3.3-

We think that “valueToLit” is preferable because in normal-eval we use expressions and not values, the function “valueToLit” purpose is to make value to expression. Because we don’t use values is normal- eval there is no need in this function.
Also in term of efficiency this way is more efficient because we don’t need to change the expression to value and back.

4. When we need to substitute variables with value we use “valueToLitExp”, this is contradiction to the type we expect AST nodes to be (expression instead of values), while in normal evaluation we substitute the expressions before the evaluation of then, so “valueToLitExp” in not needed.

5. <u>Example where normal order will execute faster then applicative:</u>

```
(L3

   (define loop (lambda (x) (loop x)))

   (define looper (lambda (x) 5))

        (looper (loop 0)))
```

In normal order, the application (loop 0) is not evaluated while In applicative order: the call (looper (loop 0)) enters into an infinite loop. So the value in normal- eval will execute faster then applicative.

Example where applicative order will execute faster the normal:

(define s1 (lambda (x) (* x x)))

(define sum-of-squares (lambda (x y) (+ (s1 x) (s1 y))))

(define f (lambda (a) (sum-of-squares (+ a 1) (* a 2)))

(f 5) ;; 136

In this example the same computations are repeated in the normal evaluation algorithm, while they were processed only once in applicative order, for example: (* 5 2) when it is passed to the function square is not computed before the substitution into the body of square - which leads to the computation of (* (* 5 2) (* 5 2)) in normal order instead of (* 10 10) in applicative order.

So the value in applicative - eval will execute faster than normal.

Part III

3.1

When we write “define” with “-“, the value in the define expression is evaluated as app expression, therefore it enters to the function “applyPrimitve” and then to the function “minusPrime” which check if there is any operands to “-“, if there isn’t it throws error.

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

In the lazy language the function won’t compute the define and automatically returns 1 (the same as normal evaluation which will evaluate only at the end when we want to use it.)

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

In the lazy language the function will throw an error and wont print the value of x because there are no operands to evaluate the value of x.

In p2 and p3 we will get an error because we use the evalNormalProgram that compute all the variables in the program even if there is no need of them. So in both cases we will get an error because the App exp didn't get operands.

To fix it we will change the evaluate of define exp such that we will evaluate the exp only when we will have to use the variable.