Q1

Q1.1

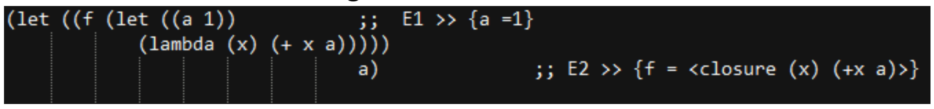
Let is indeed a special form in L3. To justify that, we will show that the evaluation of 'let' expression is 'special', since its different than the regular procedure call evaluation. For procedure call, L3 always evaluates the operator expressions and the operand expressions with the same evaluation rules (but no specific order).
Unlike the evaluation for 'let' expression, in which the vars arguments are not really expressions to be evaluated in the normal way (but the name of the storage place to put the value in)
Evaluation rule for Let :

- Evaluation of val-expr \ init-expr
- Evaluation of the location + assigning
- Evaluation of the body, hence the bindings between the valr names and vals

Q1.2

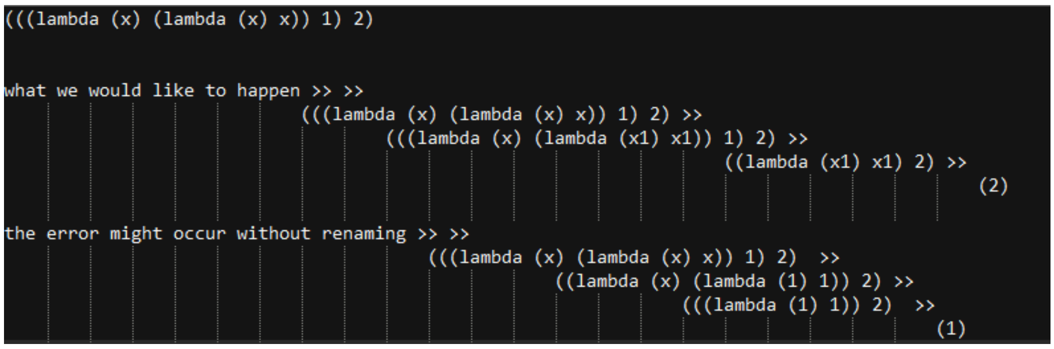
Semantics errors indicates an improper use of L3 statements and should be handled by the interpreter during run time.
We obtain run time error while having an incompatibility between the expected value and the actual value.

1. Example1 : **Undefined variable error** >>
In the following example we would get the above error, since the variable a that defined in E1 would no longer be accessible in E2 and therefor undefined in it.



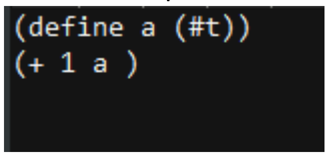
```
(let ((f (let ((a 1))          ;;  E1 >> {a =1}
           (lambda (x) (+ x a)))))
                 a)              ;; E2 >> {f = <closure (x) (+x a)>}
```

2. Example2 : **Multiple declarations for an identifier >>**
In case we have 2 occurrences of the same variable, one free and the other bound, wrong value might be computed :



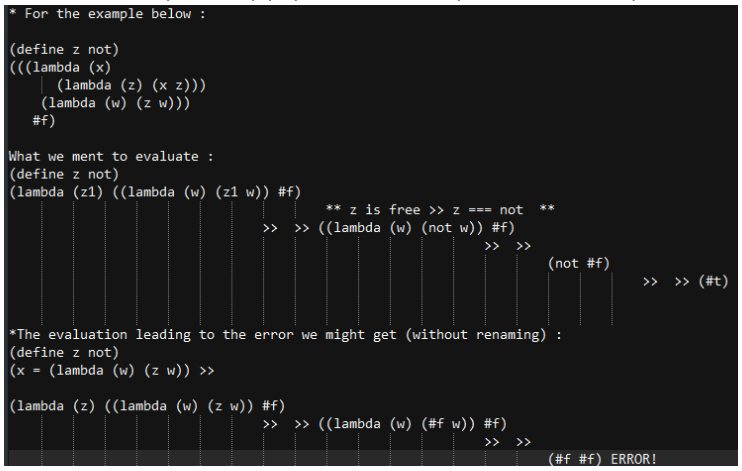
```
(((lambda (x) (lambda (x) x)) 1) 2)


what we would like to happen >> >>
                                       (((lambda (x) (lambda (x) x)) 1) 2) >>
                                              (((lambda (x) (lambda (x1) x1)) 1) 2) >>
                                                                        ((lambda (x1) x1) 2) >>
                                                                                          (2)

the error might occur without renaming >> >>
                                                (((lambda (x) (lambda (x) x)) 1) 2)  >>
                                                         ((lambda (x) (lambda (1) 1)) 2) >>
                                                                  (((lambda (1) 1)) 2)  >>
                                                                                  (1)
```

3. Example3 : **Incompatible types of operands >>** In the example below, we will get error since the primOp '+' is not defined for boolean operands.



```
(define a (#t))
(+ 1 a )
```

4. Example4 : **Incompatible usage of expressions >>** In the example below we get an attempt to apply boolean expression as a procedure



```
* For the example below :

(define z not)
(((lambda (x)
     (lambda (z) (x z)))
    (lambda (w) (z w)))
   #f)

What we ment to evaluate :
(define z not)
(lambda (z1) ((lambda (w) (z1 w)) #f)
                                        ** z is free >> z === not  **
                                 >>  >> ((lambda (w) (not w)) #f)
                                                            >>  >>
                                                                    (not #f)
                                                                                >>  >> (#t)

*The evaluation leading to the error we might get (without renaming) :
(define z not)
(x = (lambda (w) (z w)) >>

(lambda (z) ((lambda (w) (z w)) #f)
                                 >>  >> ((lambda (w) (#f w)) #f)
                                                            >>  >>
                                                                    (#f #f) ERROR!
```

Q1.3

Q1.3.1

<cexp>::= …| <value>
<value>::= <number> | <boolean> | <string> | <PrimOp> | <Closure> | <SymbolSExp> | <EmptySExp> | <CompoundSExp>
<number> ::= a number token.
<boolean> ::= #t | #f
<string> ::= an identifier token
<Closure>::= (<VarDecl>* <CExp>*) /Closure(params:VarDecl[],body:CExp[])
<SymbolSExp>::=(<string>)   /SymbolSExp(val:string)
<EmptySExp> ::= empty
<CompoundSExp>::= (<SymbolSExp> <SymbolSExp>) / CompoundSExp(val1: SymbolSExp, val2: SymbolSExp)

Q1.3.2

In ‘applyClosure’ we’ll send to substitute args instead of litArgs.
In ‘L3applicativeEval’ we’ll need to add condition – if value – return the value.

Q1.3.3

On one hand, e increase efficiency by not evaluating expressions more than once, but on the other hand out safety and correctness harmed, since we get an AST with values as expressions.

Q1.4

The procedure valueToLitExp is needed while substituting variables with values, which doesn’t come along with the expected type in the AST (expression instead of values). In normal order evaluation, we make the substitution before the evaluation of the arguments, therefore the function valueToLitExp is not needed.

Q1.5

Normal order better than Applicative order -

For the following program, the applicative evaluation eval both of test operands (0,(alt 1)) when normal order will eval only 0 :

Given the following program:

```
(define test
        (lambda (x y)
                (if ( = x 0)
                        0
                        y)))
(define alt
        (lambda (z)
                (+ z 0 ))) ;
```

and the following apply :

(test 0 (alt 1))

Normal eval :

```
   * normal_eval[(test 0 (alt 1))]
   * normal_eval[test] ==> <closure (lambda (x y) (if (= x 0) 0 y))>

         * (if (= x 0) 0 y))(x = 0) ==> (if (= 0 0) 0 y))
         * (if (= 0 0) 0 y))(y = (alt 1)) => (if (= 0 0) 0 (alt 1)))

                 *reduce:
                         * normal_eval[(if (= 0 0) 0 (alt 1)))]
                         * normal_eval[(= 0 0)]
                         * normal_eval[ = ] ==>#<primOp =>
                         * normal_eval[ 0 ] ==>0
                         * normal_eval[ 0 ] ==>0
                                                                   ==> #t
         *normal_eval[0] ==> 0
                                                        ==>0
```

Applicative eval :

```
         * app_eval[(test 0 (alt 1))]
   *app_eval[test] ==> <closure (lambda (x y) (if (= x 0) 0 y))>
   * app_eval[0] ==> 0
   * app_eval[(alt 1)]
      * app_eval[alt]==><closure (lambda (z) (+ z 0))>
      * app_eval[1]==> 1
                              ==> 1
```

>> applicative evaluation eval (alt 1) while normal doesn't.

Applicative order evaluation faster than normal order :
Applicative order evaluation gives us the opportunity to avoid re-evaluations of the same expression, therefor, the program code below applicative evaluation will be faster than the normal order one.

```
( define opExp (* 10 10) )
( define exp_8 (lambda (x) (* x x x x x x x x )) )
(exp_8 opExp)
```

For the Applicable evaluation of the expression (exp_8 opExp) we will compute the opExp expression one time and then the evaluation will proceed this way :

```
        * app_eval [(lambda (x) (* x x x x x x x x))] => <closure (x) (* x x x x x x x x )>
        *app_eval[opExp] => app_eval [(* 10 10 )] =>
                                * app_eval [*] => <primOp : * >
                                * app_eval [10] => 10
                                * app_eval [10] => 10
                                                                    => => 100
                * substitue *
                (* x x x x x x x x ) ( x = 100 ) => ( * 100 100 100 100 100 100 100 100)
                *   reduce   *
                        app_eval [(* 100 100 100 100 100 100 100 100)] =>
                                * app_eval [*] => <primOp : * >
                                * app_eval [100] => 100
                                        reduce ( ((acc,curr) => acc = curr *acc), 1,
[100,100,100,100,100,100,100,100] ) => acc = 100^8
```

While for the Normal eval order the opExp will be computed 8 times during the evaluation :

```
* normal_eval [(lambda (x) (* x x x x x x x x))  opExp ]=>
        normal_eval [(lambda (x) (* x x x x x x x x)) (* 10 10) ]

* normal_eval [(lambda (x) (* x x x x x x x x))] => <closure (x) (* x x x x x x x x )>
        *  substitue  *
                        (* x x x x x x x x ) ( x = (* 10 10)) => ( * (* 10 10) (* 10 10) (* 10
10) (* 10 10) (* 10 10) (* 10 10) (* 10 10) (* 10 10) )
                *   reduce   *
                        normal_eval [( * (* 10 10) (* 10 10) (* 10 10) (* 10 10) (* 10
10) (* 10 10) (* 10 10) (* 10 10) )] =>
                                * normal_eval [*] => <primOp : * >
                                * normal_eval [10] => 10
                                * normal_eval [10] => 10
                                                                        =>  => 100
```

reduce (((acc,curr) => acc = curr*acc), 1, [(* 10 10) ,(* 10 10) ,(* 10 10), (* 10 10), (* 10 10), (* 10 10), (* 10 10), (* 10 10))])

=> => 100^8

opExp evaluation computed once in the applicative compare to 8 times in the normal order, and that is why applicative eval faster than normal eval for this program.

Q3

Q3.1

- In both cases, a promise called x and holding AppExp (-) is created. In the "lazy" language the promise is evaluated only when applied, so in both cases the interpreter don't notice the error of creating AppExp with prim op '-' without arguments. The first run value is the promise x, the second one's value is 1.
- The problem is in our 'evalDefineExps', which evaluate DefineExp val before adding it to the environment, instead of adding it as a CExp.