Part Ⅰ: Theoretical Questions

1. Let is not special form in L3 because any let expression is re-written as an app expression.
3. Error 1: trying to use an un-define variable. (+ x 5)
4. Error 2: trying to apply procedure with a rator that is not prim-op and not a closer
5. Error 3: try to apply closer with an more or less args then defined by the procs exp.
6. Error 4: try to apply an arithmetic prim-op with opparnd that are no a number
   1. <program> ::= (L3 <exp>+) // Program(exps:List(Exp))

<exp> ::= <define> | <cexp> / DefExp | CExp

<define> ::= ( define <var> <cexp> ) / DefExp(var:VarDecl, val:CExp)

<var> ::= <identifier> / VarRef(var:string)

<cexp> ::= <num-exp> / NumExp(val:number)

| <bool-exp> / BoolExp(val:boolean)

| <str-exp> / StrExp(val:string)

| ( lambda ( <var>\* ) <cexp>+ ) / ProcExp(args:VarDecl[], body:CExp[]))

| ( if <cexp> <cexp> <cexp> ) / IfExp(test: CExp, then: CExp, alt: CExp)

| ( let ( binding\* ) <cexp>+ ) / LetExp(bindings:Binding[], body:CExp[]))

| ( quote <sexp> ) / LitExp(val:SExp)

| ( <cexp> <cexp>\* ) / AppExp(operator:CExp, operands:CExp[]))

<binding> ::= ( <var> <cexp> ) / Binding(var:VarDecl, val:Cexp)

<prim-op> ::= + | - | \* | / | < | > | = | not | and | or | eq? | string=? | cons | car | cdr | pair? | number? | list  
| boolean? | symbol? | string?  
  
<num-exp> ::= a number token

<bool-exp> ::= #t | #f

<str-exp> ::= a sequence of characters between double quotes

<var-ref> ::= an identifier token

<var-decl> ::= an identifier token

<sexp> ::= symbol | number | bool | string |

(<sexp>+ . <sexp>) | ( <sexp>\* ) ##### L3

1. Because in normal evaluation, we substitute arguments in lambda bodies with CExps rather than values and we have no need to convert an expression to itself, in contrast to applicative eval where we first calculate the value of the CExp and then we have to convert the value back to an expression
2. A program where normal eval executes faster:  
   (define f (lambda (x) 5))

(f (+ 3 5))  
  
That’s because in normal evaluation, we have no need to evaluate the expression (+ 3 5), while in applicative eval we will do so anyway.

A program where applicative eval executes faster:

(define f (lambda (x) (+ x x)))

(f (+ 3 5))

That’s because in applicative eval we will evaluate the expression  
(+ 3 5) only once, while in normal evaluation we will pass the argument as an expression, and therefore evaluate it twice