Question 1: Theoretical Questions

question 1.1

In pure functional programming, a function's body should consist only of a sole expression, since the only returned value is the last one, and there are no side-effects.

A function-body with multiple expressions is useful in the imperative paradigm, where a program is a list of consecutive instructions.

L3 functions are able to benefit from a function with multiple lines, since there are shortcuts with **let** command, but as we require to have defines only at the beginning of the program - they are not mandatory for a fully functional programming language.

question 1.2

a) special forms are required because they should be handled differently from a primitive operator.
   For example, the **define** command.
   If we'd handle it as a primitive command (like + for example),
   the command: **(define x 5)**
   will be handled as following:
   take x and 5, and perform a "define" operator on them, but since we are just initializing x's value - this will fail!

b) If **or** is to define natively (without shortcut semantics) it can be implemented as a primitive operator. since we'll simply take each operand and do a logical or with them (in Reduce for example).
   If **or** is to be defined with shortcut semantics, It'll be possible only in normal-order evaluation, since only in there it'll wait with the rest of the arguments - and if we're in a TRUE statement, it'd just stop there. In applicative order, we'll have to evaluate all of them - making it impossible.

question 1.3

a) the value of the program is 3, since the x inside the let's second expression refer to the x from the define with value 1.
   So it'll be evaluated to y = (* 1 3) which is 3.

b) let is scoping all its parameters in parallel, while let* scope all its parameters sequentially, making it a nested scope for each additional paremeter.
   so in the provided program, the x from the define will be override by the x from the first expression of the let* - making y = (* x 3) = (* 5 3) = 15.
   so the return value is 15.

c) As mentioned above, the equivalent let expression is a nested expression:

```
(define x 1)
(let ((x 5))
        (let ((y (* x 3)))
                y))
```

d) The code where each let call is replace by lambda:

```
((lambda (x)
        ((lambda (y) y)
                (* x 3)))
5)
```

question 1.4

a) In L3, the function **valueToLitExp** is needed due to a technical issue.
Since **applyClosure** is expecting literal expressions in its call of **evalSequence** function, we need to wrap the values into literal expressions to make the types fit.
b) In normal order, we don't need to call **valueToLitExp** to wrap the values, because they are not computed yet - so they are still literals.
c) In the environment model, we are not processing the arguments, so there's no need to wrap them with **valueToLitExp** like in the substitution model

question 1.5

a) A reason to switch from applicative order to normal order is when the program can be stuck in an infinite loop or crash in some condition.
So moving to normal order allows us to avoid these cases by lazy evaluation (we won't get to the bad cases).

b) A reason to switch from normal order to applicative order is when performance are becoming an issue.
Since in normal order, we can evaluate the same expression several times, while in applicative order we'll perform the evaluation only one time.

question 1.6

a) In the environment model, each proc knows the environment at the time of its creation, so the problem where a variable may point to the wrong varDec when called to inner function won't occur - making renaming redundant

b) No, because if the term being substituted is closed - the varDec will stay with the same scope as the variable itself, so it won't change its value.

Question 2D

1) Convert the ClassExps to ProcExps:

(define pi 3.14)
(define square (lambda (x) * x x))

(define circle (lambda (x y radius) (lambda (msg)
(if (eq? msg 'area) ((lambda () (* (square radius) pi))) (if (eq? msg 'perimeter) ((lambda () (* 2 pi radius))) #f)))))

(define c (circle 0 0 3))
(c 'area)

```
(define pi 3.14)
(define square (lambda (x) * x x))

(define circle (lambda (x y radius) (lambda (msg)
(if (eq? msg 'area) ((lambda () (* (square radius) pi)) )
 (if (eq? msg 'perimeter) ((lambda () (* 2 pi radius)) )
#f)))))

(define c (circle 0 0 3))
(c 'area)
```

2) List the expressions which are passed as operands in substitute model for the L3 program:

```
(L3
        (define pi 3.14)
        (define square (lambda (x) * x x))
        (define circle
                (class (x y radius)
                ((area (lambda () (* (square radius) pi))))
                (perimeter (lambda () (* 2 pi radius)))
        )
        )
        (define c (circle 0 0 3))
        (c 'area)
)
```

3.14
(* x x)
((* (square radius) pi) (* 2 pi radius))
(circle 0 0 3)
circle
0
0
3
c ‘area
c
‘area
(* (square 3) pi)
(* (square 3l pi)
*
(squre 3)
square
3
*
3
3
pi

3) Draw environment diagram for the computation of the program in environment model