1.1:

No, it's not necessary. Pure functional programming typically relies on single expressions, though multiple expressions can be used when needed. It's useful in Functional languages like Scheme, Multi-paradigm languages such as Python and JavaScript. In L3, multiple expressions can be utilized within a function. Constructs like “let” can be used.

1.2:

A:

Special forms enable control flow operations such as define. These require special handling because they either do not evaluate all their arguments or evaluate them in a specific manner.

1. **Short-Circuiting**: Logical operators like and and or conditionally evaluate their arguments. For instance, in (and a b), if a is false, b is not evaluated.
2. **Variable Binding**: Forms like define create new variable bindings and scopes, necessitating special evaluation rules to manage them correctly.

B:

Yes, but it must be a special form to apply shortcut semantics.

Due to these semantics, if the first operand is true, the result is true without evaluating the second operand. If the first operand is false, the result depends on the second operand. The `or` operator needs to conditionally evaluate its arguments, only evaluating the second operand if the first operand is false, unlike other primitive operators.

1.3:

A:

The value of the program is 3. Within the `let` expression, a local variable `x` is set to 5. Then, `y` is calculated as 3 times the global variable `x`, which is 1. Therefore, `y` becomes 3, which is returned as the result of the `let` expression. The value of `x` in the expression `(* x 3)` is 1 (the value of the global `x` variable) because you cannot use one variable defined in the `let` to define another variable within the same `let` expression.

B:

Let* expression is making bindings sequentially, which means that the local variable "x" would set to 5 before the program will calculate "y". This resulting y*(local x) = y*3 = 5*3 = 15.
Therefore, the value of the program is 15.

C:

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

D:

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

1.4:

A:

In L3, the `valueToLitExp` function converts a Scheme value into a literal expression, representing the Scheme code required to recreate that value. This function is primarily used in the interpreter to convert values like numbers, strings, booleans, and symbols into their corresponding literal expressions in L3 syntax. It is crucial for representing the interpreter's output and for debugging purposes.

B:

The `valueToLitExp` function is unnecessary in the normal evaluation strategy interpreter (L3-normal.ts). This is because normal order evaluation, or lazy evaluation, postpones the evaluation of expressions until their values are actually needed. Therefore, the interpreter manages unevaluated expressions directly, eliminating the requirement to convert already evaluated values back into their literal forms. By retaining and operating on the original expressions throughout the evaluation process, normal order evaluation simplifies expression management, making the `valueToLitExp` function redundant.

C:

In the environment-model interpreter, the function `valueToLitExp` is not needed because the interpreter manages variable bindings and their values directly within environments. There is no necessity to convert values back into literal expressions, as the interpreter efficiently handles expressions and their evaluations within the context of these environments.

5:

A:
Transitioning from applicative order to normal order evaluation can be justified based on efficiency and flexibility considerations. Normal order evaluation postpones the evaluation of arguments until they are required, thus preventing unnecessary computations and facilitating the handling of infinite data structures.
Example:
((lambda (x y) x) 1 (/ 1 0))
While is applicative order we will get an error due to trying to calculate
1/0, in normal order the result will be 1.

B:
Switching from normal order to applicative order evaluation can be justified to enforce immediate evaluation of all arguments, ensuring consistent behavior and improving error handling by detecting errors early in the evaluation process. Furthermore, applicative order evaluation typically offers faster performance compared to normal order evaluation in situations where programs do not encounter infinite loops or errors.

6:

A: In the environment model, renaming is unnecessary because each variable is uniquely bound to a binding within its lexical scope. Variables retain their identities throughout the program's execution, and new bindings within nested environments do not impact outer scopes. This lexical scoping effectively manages variable bindings without the need for renaming.

B: No, renaming is unnecessary in the substitution model for closed terms, which contain no free variables. Substituting such terms directly into other expressions avoids issues like variable capture or shadowing. This simplifies the process without conflicts or ambiguities in variable names between the substituted term and its application context.

Qustion 2d:

-Convert the ClassExp to ProcExp(as defined in 2c):

as we see in the circle body we have a ”class” so we need to convert the circle class into a

procedure expression.

so we convert this Original Class Definition:

```
(class (x y radius)
((area (lambda () (* (square radius) pi)))
(perimeter (lambda () (* 2 pi radius))))
)
```

to Procedure Expression like this :

```
(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)))))
```

after the transformation we get this Transformed program:

```
(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))
```

-List the expressions which are passed as operands to the L3applicativeEval function

during the computation of the program, (after the conversion), for the case of

substitution model.:

3.14
(lambda (x) * x x)
(lambda (x y radius) (lambda (msg) (if (eq? msg 'area) ((lambda () (* (square radius) pi))) (if (eq? msg 'perimeter) ((lambda () (* 2 pi radius))) #f))))
(circle 0 0 3)
circle
0
0
3
(lambda (msg_1) (if (eq? msg1 'area) ((lambda () (* (square 3) pi))) (if (eq? msg_1 'perimeter) ((lambda () (* 2 pi 3))) #f)))
(c 'area)
c
'area
(if (eq? 'area 'area) ((lambda () (* (square 3) pi))) (if (eq? 'area 'perimeter) ((lambda () (* 2 pi 3))) #f))
(eq? 'area 'area)
eq?
'area
'area
((lambda () (* (square 3) pi)))
(lambda () (* (square 3) pi))
(* (square 3) pi)
*
(square 3)
square
3
*
3
3
pi

-Draw the environment diagram for the computation of the program (after the conversion), for the case of the environment model interpreter:

2d:

```
(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))
```