Assignment 2:

Question 1:

1.1 A function body with multiple expressions is not required in pure functional programming. In pure functional programming, functions are expected to be pure, meaning they do not have side effects and their output depends only on their input, not on previous expressions. According to our semantics, the value of a function is the last expression, unaffected by anything other than the input if there are no side effects.

In L3, we use define to define values of expressions. The main goal is to calculate the definitions and then go directly to the last expression that returns the value of the function.

1.2 a. Special forms are required in programming languages because they change the standard evaluation rules that apply to regular functions or operators. Meaning, we can evaluate them differently from the way we normally calculate operator activation.

The lambda special form evaluates to closures. This special evaluation is crucial for the correct functioning of these function objects, also calculating all the expressions immediately can lead to errors. For example: (lambda (x) (+ x 1)), if lambda were a primitive operator, the expressions (x) and (+ x 1) would be evaluated immediately. This would lead to errors because x is undefined at the time of evaluation. Additionally, immediate evaluation would make it impossible to create the function as a value to be passed around and invoked later. Instead, lambda creates a function that waits to be called with arguments, delaying the evaluation of its body ((+ x 1)) until the function is actually invoked. This mechanism is what makes lambda a special form, as it alters the usual evaluation rules to support the creation and use of closures.

b. The logical operation 'or' must be defined as a special form to implement shortcut semantics. This ensures that evaluation stops as soon as a '#t' value is appearing, preventing unnecessary computations and potential errors. This cannot be achieved if 'or' were defined as a primitive operator, as it would require the evaluation of all arguments upfront.

Example of expression with potential error: (or (< 3 4) (/ 6 0)), In this example the first condition evaluates to '#t' and If or were a primitive operator, it would then evaluate the second condition (/ 6 0), resulting in a division by zero and therefore an error. When 'or' is defined as a special form, it evaluates each expression in sequence and stops as soon as it finds a true value so in the example above, since < 3 4 evaluates to true, or stops and does not evaluate (/ 6 0), thus avoiding the error.

1.3 a. the value of the following program is 3.

Explanation:

First we define a global variable x with a value of 1.

Then we evaluate the let expression noticing that in a let expression, the initial values of all bindings are computed before any of the variables are bound, therefore:

- The initial value for x is 5.
- The initial value for y is (* x 3). At this point, the global x (which is 1) is still in effect, so (* x 3) evaluates to (* 1 3), which is 3.

Therefore, the value of y is 3. Thus, the value of this program is 3.

b. the value of the following program is 15.

Explanation:

First we define a global variable x with a value of 1.

Then we evaluate the let* expression noticing that in a let* expression, the bindings are performed sequentially from left to right and therefore the second binding is done in an environment in which the first binding is visible.

- x is first bound to 5.
- Then y is bound to the result of (* x 3). Since let* performs bindings sequentially, when evaluating y, x is already bound to 5, so (* x 3) evaluates to (* 5 3), which is 15. Therefore, the value of y is 15. Thus, the value of this program is 15.

c.

```
(let ((x 5))
  (let ((y (* x 3)))
   y))                =>  15
```

Explanation: The first let binds x to 5. side this let, we nest another let that binds y to (* x 3). Here, x is already bound to 5 in the outer let. Thus, y is correctly evaluated to 15 within the inner let.

d.

```
((lambda (x)
  ((lambda (y) y)
         (* x 3)))
    5)                =>  15
```

Explanation: The outer lambda (lambda (x) ...) creates a new scope where x is bound to 5. Inside this lambda, another lambda (lambda (y) ...) creates a new scope where y is bound to the result of (* x 3). The result of the inner lambda is y, which is evaluated to 15.

1.4 a. In L3, the valueToLitExp function is used to convert evaluated values back into expressions. The purpose of this function is to handle compound expressions that are made of subexpressions - which in them the variables being replaced with expressions during the substitution process. Applicative order evaluation requires that arguments are evaluated into values before substitution, which changes their type. To implement this, we first evaluate arguments into values, then use the valueToLitExp function to convert values back into expressions. This conversion ensures that the values can be correctly substituted as subexpressions within the larger expression.

b. Normal order evaluation strategy avoids the conversion step because arguments are not evaluated before substitution. Instead, expressions are substituted directly, and evaluation is delayed until the expression is actually needed. Since argument values aren't evaluated before substitution, expressions are used directly without being converted into values first. Therefore, there is no need for a function like 'valueToLitExp' to convert values back into expressions, and allowing substitution without type errors.

c. In the environment-model we do not need to place the arguments inside the body of the procedures and therefore no substitution is performed (we manage argument bindings through the environment rather than substitution). This avoids the problem for which we use the 'valueToLitExp' function (since valueToLitExp function is used to handle the process of substituting evaluated values into the procedure body), so there will be no type error and therefore there is no need for this function in the environment-model interpreter.

1.5 a. Avoiding Unnecessary Computation**:** Certain parts of an expression sometimes do not need to be evaluated at all. By using normal order, we avoid expensive computations that are never used. An example of this use -

```
((lambda (x y z)
  (if x y z))
  #t 3 (* 82 81))
```

In the applicative order, all three operands are calculated in advance, including the expression (* 82 81). In normal order, only x is evaluated initially, avoiding the unnecessary computation of (* 82 81).

Avoiding Errors in Conditional Expressions: Normal order evaluation can prevent runtime errors and Infinite loops in conditional expressions by not evaluating arguments that are not needed, sometimes it can be useful, Example:

```
(define (divide x y)
 (if (= y 0)
    'undefined
     (/ x y)))

(divide 10 0)
```

b. Switching from normal order to applicative order evaluation can be justified for several reasons:

Efficiency: Applicative order evaluation evaluates each argument only once, whereas normal order might evaluate the same argument multiple times. This can lead to performance improvements. An example of this use –

```
((lambda (x)(+ x x)) (* 65 65))
```

In applicative order, we first calculate (* 65 65) to get 4225, and then substitute it, and get (+ 4225 4225). In the normal order we substitute (* 65 65) without calculating it at this stage, this expression will be calculating later when it is required for the + primitive operator to be invoked. we will receive: (+ (* 65 65) (* 65 65)), which evaluates (* 65 65) twice and therefore applicative order is more efficient here.

Compatibility with Side Effects: If there are side effects calculating an expression multiple times can cause unnecessary side effects (applicative order ensures side effects occur in a predictable order because it evaluates arguments before applying the function). Example:

`((lambda (n) (* n n)) ((lambda () (display 3) 4)))`, Normal order would display "3" twice due to repeated evaluation of the inner lambda, and if we want to avoid that we can use applicative order.

1.6 a. Renaming is not required in the environment model because instead of substituting variables with their values when applying a closure, the body of the closure is left untouched. In the environment model, we maintain a current environment data structure. When we evaluate a procedure expression, we construct a closure value that saves the environment at the time of closure creation. When evaluating an application expression, a new environment is constructed by extending the closure's environment with mappings of parameters to arguments. This approach avoids the need for renaming because the environment keeps track of variable bindings directly, ensuring correct variable resolution without direct substitution.

b. Renaming is not required in the substitution model when substituting a "closed" term, meaning it has no free variables. This ensures there is no risk of introducing a binding that conflicts with a free variable in the original expression. Since all variables in the closed term are bound, the substitution process is straightforward and safe without needing renaming.

Question 2 – part d:

Converting 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)
```

For the substitution model, we list the expressions evaluated by the interpreter:

- `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? msg__1 'area) ((lambda () (* (square 3) pi))) (if (eq? msg__1 'perimeter) ((lambda () (* 2 pi 3))) #f)))`
- `(c 'area)`
- `c`
- `'area`
- `(if (eq? msg 'area) ((lambda () (* (square radius) pi)))(if (eq? msg 'perimeter)((lambda () (* 2 pi radius))) #f))`
- `(eq? 'area 'area)`
- `eq?`
- `'area`
- `'area`
- `((lambda () (* (square 3) pi)))`
- `(lambda () (* (square 3) pi))`
- `(* (square 3) pi)`
- `*`
- `(square 3)`
- `Square`
- `3`

- `(* 3 3)`
- `*`
- `3`
- `3`
- `pi`

Environment diagram for the computation of the program: