Question 1 (Theoretical Questions)

1. Multiple expressions in a function body are mostly needed when there are side effects. This is because the program's value comes from the last expression. If there are no side effects, the previous expressions don't change the final result. In languages with side effects, you need to compute the earlier expressions first. But in pure functional programming, without side effects, you only need to look at the last expression to get the result. In the case of L3, In the case of L3, the language does support function bodies with multiple expressions. it is necessary to calculate only the defines, after which you can skip to the last expression.
2. A. Special forms are calculated differently from regular operators. With regular operators, the operands are first calculated and then the operator is applied to them. But with special forms like 'if', you don't need to calculate all the sub expressions the 'test', the 'then' part, and the 'else' part. You only need to calculate two of them.

   B. To support shortcut behavior, where some sub expressions don't need to be calculated, 'or' must be defined as a special form. Otherwise, all parts would be calculated first before applying the 'or' operator. This shortcut behavior is expected for the 'or' operator (even if it's a basic operator). If you don't need this shortcut behavior, you can use a regular operator instead, as some programming languages do.

3. A. The answer is 3. The explanation is: since y is not defined in the global environment, it evaluates to the primitive procedure y itself, which prints out some representation of the procedure when evaluated.

   B. The answer is 3. The explanation is: let* allows using the bindings defined earlier in the same let* expression. In this case, when y is bound to (* x 3) x refers to the newly bound value 5, not the outer x which is 1.

   C. (let ((x 5))

   (let ((y (* x 3)))

   y))

   D. ((lambda (x)

   ((lambda (y)

   y)

   (* x 3)))

   5)

4. A. In L3, the role of the valueToLitExp function is to convert a value which is a JavaScript object into a corresponding literal expression which is a nested array representation.

   B. The valueToLitExp function is not needed in the normal evaluation strategy interpreter because this interpreter does not construct new expressions from values.

Instead, it follows the normal order evaluation strategy, which means that it only evaluates function arguments when they are needed for the function application.

C. The valueToLitExp function is also not needed in the environment-model interpreter because this interpreter does not construct new expressions from values. Instead, it maintains an environment witch is a data structure that maps variable names to values and evaluates expressions by looking up variable values in the environment.

5. A. Efficiency: In cases where not all arguments are needed for a function application, normal order evaluation can be more efficient because it avoids unnecessary computations.

   Termination behavior: Normal order evaluation can terminate in cases where applicative order evaluation would not, because it only evaluates arguments when they are needed.

   An example:

```
(define loop (lambda () (loop)))
(define f (lambda (x) 5))
(f (loop))
```

   Applicative: the operand (loop) will be calculated in the activation of f, and

   will enter an infinite loop..

   Normal: the operand (loop) will be placed in f without calculating. Since there is no reference to it in f, it will not be calculated at all and the program will end as a series.

   B. Simplicity: Applicative order evaluation is generally simpler to understand and implement because it follows a more straightforward "evaluate all arguments first" approach.

   Strict evaluation: Applicative order evaluation is the basis for strict evaluation, which can be useful in certain contexts (e.g., when working with side effects or exceptions).

   An example : a function that computes the sum of a list of numbers
   (sum (list 1 2 3 (infinite-loop))).

   In normal order, the infinite-loop expression would be evaluated because it's needed for the application of sum. In applicative order, infinite-loop would never be evaluated because the evaluation would stop after encountering an error or non-terminating computation in one of the earlier arguments.

6. A. Renaming is not required in the environment model because the environment model maintains a separate environment for each level of nested function applications.

   B. In the substitution model, renaming is required even in cases where the term that is substituted does not contain free variables. This is because the substitution process can introduce name clashes between the variables in the term being substituted and the variables in the context where the substitution is occurring.

Question 2

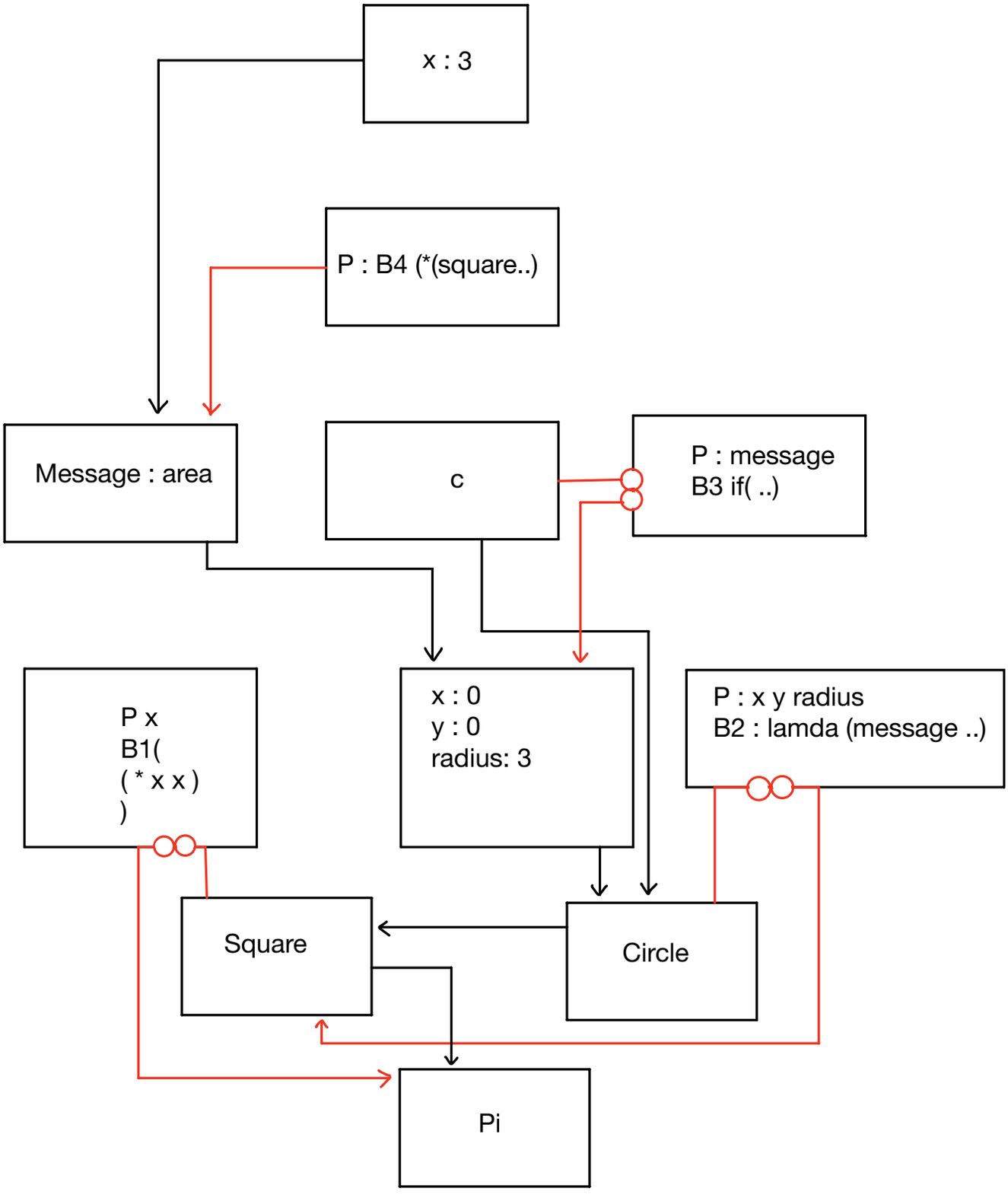
d)

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

the list of expressions:

1. 3.14
2. (lambda (x) * x x)
3. (lambda (x y radius) (lambda (msg) ...)) [The converted class expression]
4. (circle 0 0 3)
5. circle
6. 0
7. 0
8. 3
9. (lambda (msg__1) (if (eq? msg__1 'area) ...)) [Result of applying circle]
10. (c 'area)
11. c
12. 'area
13. (if (eq? 'area 'area) ((lambda () (* (square 3) pi))) ...)
14. (eq? 'area 'area)
15. eq?
16. 'area
17. 'area
18. ((lambda () (* (square 3) pi)))
19. (lambda () (* (square 3) pi))
20. (* (square 3) pi)
21. *
22. (square 3)
23. square
24. 3
25. *
26. 3
27. 3
28. Pi

The diagram:


x : 3
P : B4 (*(square..)
Message : area
c
P : message
B3 if(..)
P x
B1(
(* x x)
)
x : 0
y : 0
radius: 3
P : x y radius
B2 : lamda (message ..)
Square
Circle
Pi