Question 1

Section 1

Multiple expressions in a function's body is not required in pure functional programming languages because they don't allow side effects (we can send functions as arguments instead). In impure programming languages though, a function body with multiple expressions is useful – it allows data manipulation and better code organization. L3 is supposed to be a pure functional programming language, thus a single expression is sufficient in a function body.

Section 2

Part a

Special forms are required in programming languages because they allow specific/unique evaluation rules that don't necessarily follow the language/interpreter "regular" evaluation rules. An example to that could be the "if" statement – we'd like to (have the option to) evaluate the "then" part if-and-only-if the test is true, and the "else" part otherwise. The normal evaluation rules may be to evaluate everything and then decide what to do – making us evaluate both the "then" and the "else".

Part b

The "or" operator can be defined as a primitive operator that evaluates both inputs and the perform logical OR on them, like any other function. However, there is the option of "short circuit or" which evaluates the first input and only if it is false – the second. In that case, the evaluation is "unique" and thus "or" may be defined as a special form.

Section 3

Part a

The value of this program is 3. At the time "Let" evaluates "y" – which is the value of the "let" expression, the only "x" it knows is the x that was defined before the let expression and its value is 1. "y" equals 3 * 1 = 3, then.

Part b

The value of this program is 15. At the time "Let*" evaluates "y" – which is the value of the "let" expression, it knows the "x" that appeared earlier in that let* expression and its value is 5. "y" equals 3 * 5 = 15, then.

Part c

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

Part d

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

Section 4

Part a

The role of this function is to convert a value back to its appropriate expression type, this is necessary because the evaluation process delas with expressions, and not directly with values (in applyClass and in applyClosure).

Part b

The valueToLitExp function is not needed in the normal evaluation strategy interpreter because in that strategy expression are evaluated only when they are needed – thus no need to convert them back (we use the value as it is needed).

Part c

The valueToLitExp function is not needed in the normal evaluation strategy interpreter because in that strategy expression are evaluated only when they are needed – thus no need to convert them back (we use the value as it is needed).

Part d

The valueToLitExp function is not needed in the environment model interpreter because in that model we do not perform substitutions – instead, values are stored in environments and are used directly from them – thus no need for conversion at all.

Section 5

Part a

We'll want to switch from applicative order to normal order, for example, when we pass functions that may be "risky" – for example when they contain a loop that may never end – we may want to pass it as an argument, without necessarily running it. In that scenario, we may get stuck in an infinite loop without needing to.

Part b

We'll want to switch from normal order to applicative order, for example, when we are required to perform many recurring computations – using the normal order we'll evaluate them again and again and using the applicative order – we can simply evaluate each once and then use the computed value.

Section 6

Part a

Renaming is not required in the environment model because in that model recurring names are overshadowed by more recent/outer frames – thus no complication occurs.

Part b

No, renaming is not required in the substitution model when all variables are bounded – that is because each variable is used as in its last definition (for example in a lambda expression).

Question 2

Section d

Program after conversion:

```
(L3
      (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 () (* (pi ) radius)) )
                                    #f)))))
(define c (circle 0 0 3))
(c 'area))
```

Operands passed to L3ApplicativeEval:

```
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 () (* (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 () (* (pi ) 3)) ) #f))
(c 'area)
c
```

area'

if (eq? 'area 'area) ((lambda () (* (square 3) pi))) (if (eq? 'area) ('perimeter) ((lambda () (* (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