Q1.1

In pure functional programming, a function-body with multiple expressions is not strictly required. Pure functional programming emphasizes immutability, stateless functions, and expressions that yield values without side effects. In such a paradigm, the focus is on single expressions that evaluate to a value, with the result of one expression often being passed as an argument to another.

However, function-bodies with multiple expressions can be useful in languages that support imperative features, where side effects are common and multiple statements are executed for their effects rather than their return values. These languages include most mainstream programming languages like Python, Java, and C++, where a function-body typically consists of a sequence of statements executed in order, with side effects such as variable assignments, I/O operations, and modifications of mutable data structures.

Q1.2

a.

Special forms are required in programming languages because they allow for control over the evaluation process of their arguments, which primitive operators cannot provide. Unlike primitive operators, special forms can manipulate the evaluation of their arguments to achieve desired control flow and computational behavior.

For example, in the case of the if special form:

(if condition

then-expression

else-expression)

The condition is evaluated first. Depending on its result, either the then-expression or the else-expression is evaluated, but not both. This selective evaluation is essential for efficient and correct conditional execution.

b.

The logical operation **or** must be defined as a special form to properly implement shortcut semantics. This ensures that the second operand is only evaluated if the first operand is false, which is crucial for both efficiency and correctness. Defining or as a primitive operator would force the evaluation of all operands, defeating the purpose of shortcut semantics and potentially leading to unnecessary computations and unintended side effects.

Q1.3

a.3

b.15

c.

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

d.

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

Q.4

a. The function valueToLitExp transforms various types of values into their corresponding literal expression representations within the AST. This is crucial for the interpreter to understand and process different types of values uniformly.

b.

The valueToLitExp function is not required in the normal evaluation strategy interpreter (L3-normal.ts) because normal evaluation delays the evaluation of expressions until they are explicitly needed. This lazy evaluation approach reduces the necessity for immediate conversion of values into specific AST node types, which is the primary function of valueToLitExp. As a result, the normal evaluation strategy can handle expressions more directly and efficiently without pre-converting values into their literal expression forms.

c.

The valueToLitExp function is not needed in the environment-model interpreter because the environment model directly manages variable bindings and values within the environment. The interpreter retrieves and uses these values as needed, without requiring their conversion into literal expression forms. This direct handling of values and immediate evaluation within the environment context makes valueToLitExp unnecessary.

Q.5

a.

Avoiding Unnecessary Computations

Handling Infinite Structures

Dealing with Conditional Expressions

Enabling Certain Program Transformations

Preventing Runtime Errors (for some cases)

Switching from applicative order to normal order evaluation is justified in scenarios where delayed evaluation can lead to performance improvements, enable the use of infinite data structures, simplify conditional computations, and support certain programming paradigms that rely on laziness. These benefits demonstrate the practical advantages of normal order evaluation in specific contexts.

Example: Consider a function that may not use all of its arguments:

```
(define (lazy-or a b)
  (if a
      #t
      b))
```

In applicative order, both a and b are evaluated before the function is called. If a is true, evaluating b is unnecessary.

In normal order, b is only evaluated if a is false, potentially saving computation time and resources.

b.

Justifications for Switching to Applicative Order Evaluation

Predictable Behaviour

Avoiding Resource Leaks

Improving Performance:

Example: Consider a function that performs expensive computations:

```
(define (compute-and-print x)
```

```
(begin
  (display "Computing...")
  (compute-something x)))
```

In normal order evaluation, if compute-and-print is called with an argument that is not used, the expensive computation is still performed.

In applicative order evaluation, all arguments are evaluated before the function is called, avoiding unnecessary computations.

Q6:

a.

the environment model of interpretation does not require renaming because it relies on unique variable bindings, lexical scoping, immutable bindings, symbolic representation of variables, and efficiency considerations to manage variable references and avoid naming conflicts. Renaming operations are unnecessary in this model due to its design and principles of variable management

b.

renaming is not required in the substitution model when substituting closed terms because there is no risk of variable capture or naming conflicts. Closed terms contain only bound variables, eliminating the need for renaming to preserve variable bindings in the surrounding context. This simplifies the substitution process and enhances efficiency without compromising correctness.

2.c1:

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

(if (eq? msg 'perimeter)

((lambda () (* 2 pi radius))) #f))))

2.c2

1. 3.14
2. (lambda (x) (* x x))
3. (lambda (x y radius)
   (lambda (msg)
     (if (eq? msg 'area)
       ((lambda () (* (square radius) pi)))
       (if (eq? msg 'perimeter)
         ((lambda () (* 2 pi radius)))
         #f))))
4. (circle 0 0 3)
5. circle
6. 0
7. 0
8. 3
9. (lambda (msg)
     (if (eq? msg 'area)
       ((lambda () (* (square radius) pi)))
       (if (eq? msg 'perimeter)
         ((lambda () (* 2 pi radius)))
         #f))))
10. (c 'area)
11. c
12. 'area
13. (if (eq? 'area 'area)
     ((lambda () (* (square 3) pi)))

```
    (if (eq? 'area 'perimeter)
        ((lambda () (* 2 pi 3)))
        #f)))
```

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. (* 3 3)
26. *
27. 3
28. 3
29. pi