Question 1: Theoretical Questions [30 points]

1.1 Is a function-body with multiple expressions required in a pure functional programming? In which type of languages is it useful? What about L3? [2 points]

In pure functional programming, the primary focus is on functions and expressions that avoid side effects. Therefore, a function-body with multiple expressions is not required. The reason for that is that according to semantics, the value of a program is the value of the last expression- If there is no effects-side, the value of the last expression is not affected by the previous expressions.
A function-body with multiple expression is useful in languages that support both pure and impure features, or in managing side effects using specific constructs like monads.
In L3, it is essential to evaluate the defined expressions first, and subsequently, the evaluation proceeds directly to the last expression in the function body.

Q1.2

a. **Why are special forms required in programming languages? Why can't we simply define them as primitive operators? Give an example [3 points]**

   Special forms are essential in programming languages because they enable specific evaluation rules that differ from the default procedure application semantics. Typically, in procedure application, the operator and all operands are evaluated before applying the function. However, certain constructs require customized evaluation behavior to function correctly.
   For example, consider the *"define"* special form used to create variable bindings. Unlike standard operators, *"define"* does not evaluate its first operand (the variable name) before the binding occurs. If *"define"* were treated as a primitive operator, the evaluation of its operands prior to execution would lead to incorrect behavior or errors.

b. **Can the logical operation 'or' be defined as a primitive operator, or must it be defined as a special form? Refer in your answer to the option of shortcut semantics. [3 points]**

The logical operation ‘or’ can be defined as a primitive operator. In that case, both operands are always evaluated, regardless of the value of the first operand. This can lead to unnecessary computation or potential errors.
If or is defined as a special form, it can implement shortcut semantics, meaning it evaluates operands conditionally (in the same way if is defined). The second operand is only evaluated if the first operand is false, which contributes to efficient computation and prevents evaluation that may lead to errors.

Q1.3

a. **What is the value of the following L3 program? Explain. [2 points]**

   (define x 1)

   (let ((x 5)

   (y (* x 3)))

   y)

The value of the program is 3. In a “let expression”, the initial values are computed before any of the variables become bound. Therefore, At the time y is evaluated, the x inside (* x 3) refers to the outer x (defined as 1) because the bindings are done simultaneously, so y becomes (* 1 3) which evaluates to 3.

b. **Read about let* here.**

   What is the value of the following program? Explain. [2 points]

   (define x 1)

   (let* ((x 5)

   (y (* x 3)))

   y)

The value of the program is 15. In a “let * expression”, the bindings and evaluations are performed sequentially. Therefore, At the time y is evaluated,

the x inside (* x 3) refers to the inner x (defined as 5), since x is already bound to 5 in this local environment. So y becomes (* 5 3) which evaluates to 15.

c. Define the let* expression above as an equivalent let expression [2 points]

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

d. Define the let* expression above as an equivalent application expression (with no let) [2 points]

```
(define x 1)

((lambda (x)

   ((lambda (y)

      y)

   (* x 3)))

 5)
```

Q.4 [6 points]

a. In L3, what is the role of the function valueToLitExp?

The L3applyProcedure procedure receives arguments which are all of type Value (proc is a Value which can be either a PrimOp or a Closure value).

In the case in which proc is a Closure value, the function applyClosure is called. In the substitution model, our goal is to replace all VarRef occurrences in the body with the corresponding values of the arguments.

The body of the closure is a list of CExp expressions, while the received arguments is a list of values.

If we replace the arguments with the variable names, the resulting body will not be a valid AST.

The role of the function **valueToLitExp** is to address this discrepancy- the function maps the values of the arguments to corresponding expressions.

b. The valueToLitExp function is not needed in the normal evaluation strategy interpreter (L3-normal.ts). Why?

The only difference between normal-eval and applicative-eval is the handling of parameters in apply-procedure: In applicative-eval, we first evaluate the parameters, then substitute, then reduce. In normal-eval, we substitute the parameters non-evaluated, then reduce.

Since the substitution in the applicative-eval occurs after the values of the parameters are evaluated, we need to use the valueToLitExp function. On the other hand, in the normal-eval method, there is no need for this function because we substitute the parameters in their non-evaluated form (there is no need to convert values back to expressions).

d. The valueToLitExp function is not needed in the environment-model interpreter. Why?

In the environment model, the process of applying a procedure does not involve substituting argument values directly into the body of the procedure. Instead, it involves creating a new environment where the formal parameters of the procedure are bound to the argument values. Therefore, there is no need for this function because we don't perform any substitution.

Q.5 [4 points]

a. What are the reasons that would justify switching from applicative order to normal order evaluation? Give an example.

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

- Avoiding Unnecessary Computations**:** Normal order evaluation delays the evaluation of an expression until its value is actually needed. This can save computation time and resources if certain expressions are not required for the final result.
- Avoiding Errors and Infinte Loops**:** If an expression could cause an error or exception if evaluated too early, normal order evaluation can prevent these

errors by delaying the evaluation until it's clear whether the expression's value is needed.

Example:

((lambda (a b c) (if a b c)) #t 3 (/ 30 0))

In applicative order, all arguments are evaluated before the function is applied, therefore the division by zero error occurs before the lambda function is even executed, leading to an immediate runtime error.

In normal order (lazy evaluation), arguments are not evaluated until their values are needed in the function body. In this case, the value of (/ 30 0) is never needed because the condition a (which is #t) is true, and thus the expression evaluates to b (which is 3). The potential division by zero error is avoided entirely because the computation of (/ 30 0) is deferred and ultimately never performed.

b. What are the reasons that would justify switching from normal order to applicative order evaluation? Give an example.

One reason is memory usage - Applicative order evaluation can be more efficient because it evaluates each argument exactly once. In contrast, normal order evaluation might repeatedly evaluate the same expression, especially if it is used multiple times within the function body.
Another reason is Predictability and simplicity. applicative evaluation can make the flow of the program more predictable and easier to understand, as all expressions are evaluated as soon as they are bound to a variable.
Example:
((lambda (a) (+ a a a)) (* 5 4))
In applicative order the argument (* 5 4) is evaluated only once, but in normal order, it is evaluated three times.

Q.6 [4 points]

a. Why is renaming not required in the environment model?

Renaming is not required in the environment model because variable references are resolved dynamically using the current environment, which keeps track of all variable bindings. This dynamic resolution ensures that

each variable reference is correctly matched to its corresponding value without the need for upfront substitution and renaming.

e. **Is renaming required in the substitution model for a case.where the term that is substituted is "closed", i.e., does not contain free variables? explain.**
   Renaming is not required when substituting a “closed” term because such terms have no free variables that could be unintentionally captured by the surrounding context. The absence of free variables in closed terms eliminates the risk of variable conflicts, making renaming unnecessary.

Question 2: Adding Class to L3 [70 points]:

d. Given the following program:

(define pi 3.14)

(define square (lambda (x) * x x))

(define circle

(class (x y radius)

((area (lambda () (* (square radius) pi)))

(perimeter (lambda () (* 2 pi radius))

)

)

(define c (circle 0 0 3))

(c 'area)

- Convert the ClassExps to ProcExps (as defined in 2.c)
(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)

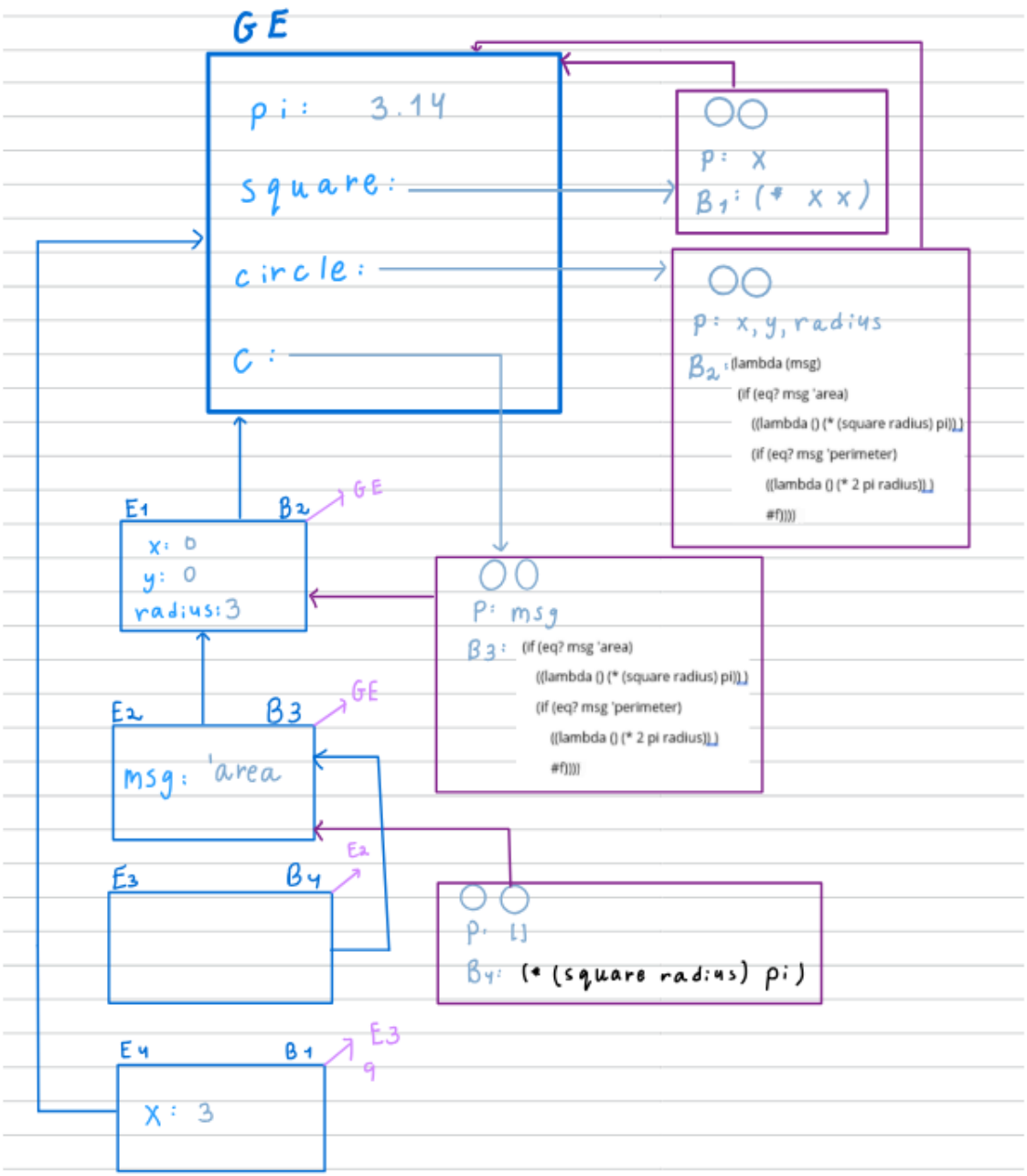
```
    (lambda (msg)
     (if (eq? msg 'area)
        ((lambda () (* (square radius) pi)) )
        (if (eq? msg perimeter)
           ((lambda () (* 2 pi radius)) )
              #f))))
```

- (circle 0 0 3)
- circle
- 0
- 0
- 3
- (lambda (msg__1)

```
(if (eq? msg__1 '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))
- (* (square 3) pi)
- *
- (square 3)
- Square
- 3
- (* 3 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


GE
pi: 3.14
square:
circle:
C:
p: x
B1: (* x x)
p: x, y, radius
B2: (lambda (msg)
(if (eq? msg 'area)
((lambda () (* (square radius) pi)))
(if (eq? msg 'perimeter)
((lambda () (* 2 pi radius)))
#f))))
E1
B2
GE
x: 0
y: 0
radius: 3
p: msg
B3: (if (eq? msg 'area)
((lambda () (* (square radius) pi)))
(if (eq? msg 'perimeter)
((lambda () (* 2 pi radius)))
#f))))
E2
B3
GE
msg: 'area
E3
B4
E2
p: []
B4: (* (square radius) pi)
E4
B1
E3
q
X: 3