Part 1: Theoretical Questions

1.What is the purpose of valueToLitExp and what problem does it solve?

The valueToLitExp function is used to convert a runtime value (represented as a Value type) into its corresponding literal expression (represented as a NumExp, BoolExp, StrExp, LitExp, PrimOp, or ProcExp).

The purpose of valueToLitExp is to enable the evaluation of procedure applications by substituting computed values into the body of the closure. Since closures have parameters that are represented as literal expressions, the computed values of the arguments need to be converted back into literal expressions to fit the types expected by the closure's body.

The function handles different cases depending on the type of the value:

- If the value is a number, it creates a NumExp using makeNumExp.
- If the value is a boolean, it creates a BoolExp using makeBoolExp.
- If the value is a string, it creates a StrExp using makeStrExp.
- If the value is a primitive operation, it returns the operation itself.
- If the value is a closure, it creates a ProcExp using makeProcExp with the closure's parameters and body.

By converting values to their corresponding literal expressions, the function ensures that the substituted expressions in the closure's body can be evaluated correctly.

2.valueToLitExp is not needed in the normal order evaluation strategy interpreter (L3-normal.ts). Why?

The normal order evaluation strategy directly substitutes variables with expressions, since the arguments aren’t evaluated before substitution, so we can directly substitute them without a type error.

3.What are the two strategies for evaluating a let expression?

A let expression is a construct in programming languages that allows the programmer to define a local variable within an expression. When evaluating a let expression, there are two common strategies that can be used:

1. Substitution strategy: In this strategy, the let expression is replaced by the expression it binds to the variable, and all occurrences of the variable within that expression are replaced by its value. For example, consider the following let expression:

let ((x 2))

(y 3))

(+ x y))

Using the substitution strategy, we replace x with 2 and y with 3 within the expression (+ x y), giving us (2 + 3), which evaluates to 5.

2. Environment strategy: In this strategy, a new environment is created to hold the bindings introduced by the let expression, and the expression is evaluated within this environment. The environment is then discarded after the expression has been evaluated. For example, continuing from the previous example, using the environment strategy, we first create a new environment with bindings for **x** and **y**, evaluate the expression **(+ x y)** within this environment, and then discard the environment. This gives us the same result of **5**.

Both strategies can be used to evaluate let expressions, and which one to use depends on the specific language and context. However, the environment strategy is more general and can handle cases where the substitution strategy may lead to errors, such as when the same variable is bound multiple times within a let expression.

4.List four types of semantic errors that can be raised when executing an L3 program- with an example for each type.

Semantic errors are errors that occur when the code is syntactically correct, but it violates the language semantics, resulting in unexpected or incorrect behavior. Here are four types of semantic errors that can be raised when executing an L3 program:

1. Type mismatch error: This error occurs when an operator or function is applied to operands of incompatible types. For example, trying to add a string and an integer:

```
x = "Hello"

y = 5

z = x + y  # Type mismatch error
```

2. Undeclared variable error: This error occurs when a variable is used before it is declared. For example:

```
x = y + 5  # Undeclared variable error

y = 10
```

3. Division by zero error: This error occurs when the code attempts to divide a number by zero. For example:

```
x = 5

y = 0

z = x / y  # Division by zero error
```

4. Out-of-bounds error: This error occurs when an attempt is made to access an array element or memory location that is outside the defined bounds of the array or memory. For example:

```
arr = [1, 2, 3]

x = arr[5]  # Out-of-bounds error
```

5.What is the difference between a special form and a primitive operator?

Most compound expressions are treated as procedure calls, which require first evaluating the operator and then, if the operator is a primitive operator or the evaluation is done in applicative order - evaluating all the operands before evaluating the whole expression, however - special forms have a different evaluation behavior which doesn't go by that rule and require special treatment, as described for example in the answer to question 1. Compound expressions which has a primitive operator.

6.What is the main reason for switching from the substitution model to the environment model? Give an example

When substituting variables for terms you need to traverse the whole AST each time you substitute, which may be very costly, especially when using recursion. For example consider the following program:

```
(define fact (lambda (n)
        (if (= n 0) 1 (* n (fact (- n 1))))
))
(fact 4)
```

Let's go over the evaluation of the program by substitution:

(fact 4)

=> ((lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) 4)

=> (if (= n 0) 1 (* n (fact (- n 1))))[n := 4]

=> (if (= 4 0) 1 (* 4 (fact (- 4 1))))
=> (if #f 1 (* 4 (fact (- 4 1))))

=> (* 4 (fact (- 4 1)))

=> (* 4 (fact 3))

=> (* 4 ((lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))) 3))

=> (* 4 (if (= n 0) 1 ((* n (fact (- n 1))))[n := 3]))

=> (* 4 (if (= 3 0) 1 (* 3 (fact (- 3 1)))))

=> ...

Notice how with each recursive call you need to substitute n for 3 in three different places, requiring traversal of the whole AST of:

(if (= n 0) 1 (* n (fact (- n 1))))

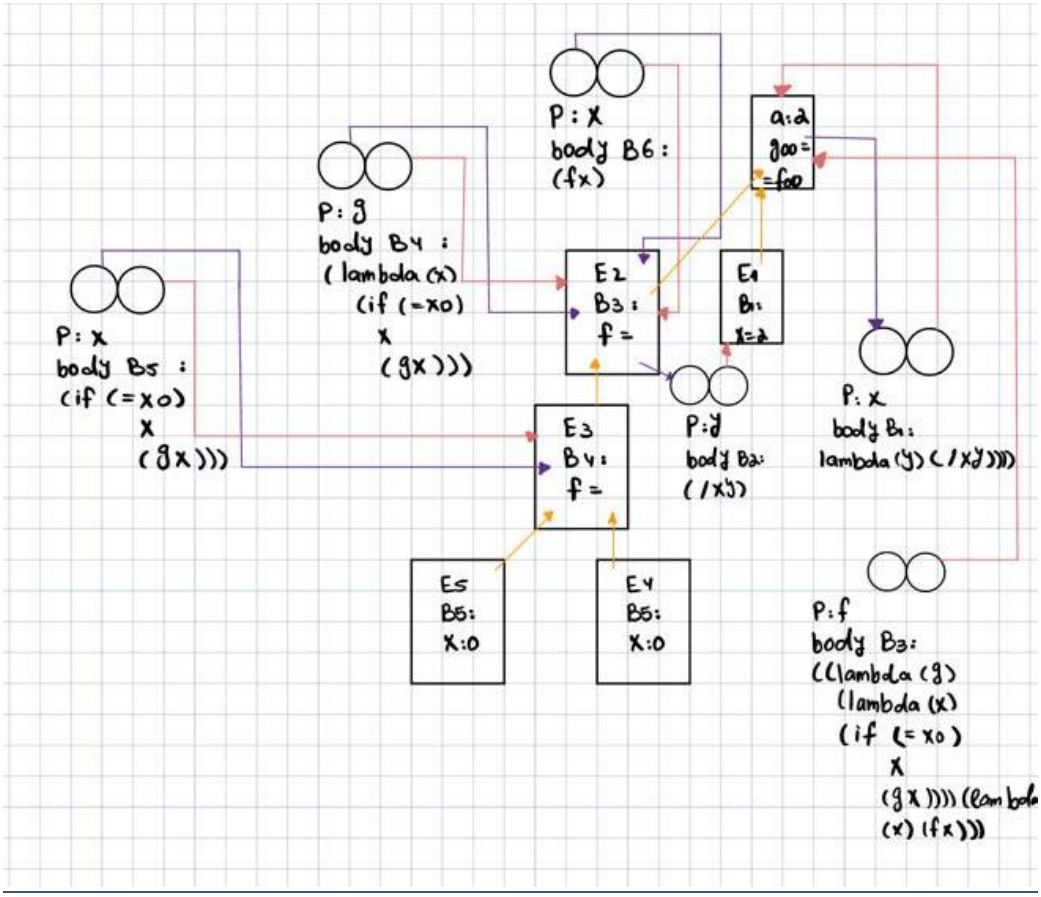
with each recursive call.

7. What is the main reason for implementing an environment using box?

The main reason for implementing an environment using a box is to allow for mutable variables in a programming language. In a functional programming language, variables are typically immutable, which means that their values cannot be changed once they have been assigned. However, in languages that support mutation, variables can have their values updated or changed during program execution.

Using a box to implement an environment allows us to support mutable variables while maintaining the benefits of functional programming. A box is a data structure that contains a value and allows for the value to be updated. By using a box to hold the value of a variable in an environment, we can change the value of the variable during program execution without violating the immutability of the environment.

8. Draw an environment diagram for the following computation. Make sure to include the lexical block markers, the control links and the returned values.



2.1.3 Theoretical

Why is bound? expression has to be a special form, and cannot be a primitive or a user function?

The "bound?" expression needs to be a special form rather than a primitive or a user function due to several reasons:

Evaluation Time: The "bound?" expression requires accessing the current environment at evaluation time to determine if a variable is bound. Special forms have the ability to directly interact with the evaluation environment during the evaluation process. Primitives and user functions, on the other hand, operate on the evaluated arguments and do not have direct access to the environment.

Evaluation Order: Special forms allow for control over the evaluation order of their arguments. In the case of "bound?", it needs to evaluate its argument (the variable reference) before checking if it is bound in the environment. Special forms provide the flexibility to define custom evaluation rules for their arguments, which is not possible with primitives or user functions.

Syntax and Parsing: Special forms often involve specific syntax and parsing rules that are different from regular function calls. The "bound?" expression has a unique syntax that distinguishes it from regular function calls. Special forms can be designed to handle this distinct syntax and perform the necessary checks against the environment's bindings.

Language Semantics: Special forms introduce language constructs with special evaluation semantics. They allow the language designer to define unique behavior that cannot be achieved with primitives or user functions. The "bound?" expression requires a specific evaluation rule to determine if a variable is bound, which is best implemented as a special form.

By making "bound?" a special form, the language ensures that the expression has the necessary capabilities, evaluation order, and syntax handling to perform the desired functionality of checking if a variable is bound in the environment.

2.2.2 Theoretical Question

Can it be implemented as a user function, primitive or special form?

The timing tool described in 2.2.1 cannot be implemented as a user function or primitive in L4. This is because it requires the ability to intercept and measure the execution time of an expression, which goes beyond the capabilities of ordinary user functions or primitives. Instead, it needs to be implemented as a special form in the L4 language.