Principles of Programming Languages 202
Assignment 3

Ben Gindi - 205874142
Shira Segev - 208825349

Part Ⅰ: Theoretical Questions

Q1

let in L3 is a special form, since it has an evaluation rule of its own (according to special form definition).
Another explanation is that **let** "behaves" the same as **lambda**, and lambda is a special form.

Q2

Executing an L3 program can raise some semantic errors.

We define it as the errors that occurs during evaluation.
The Idea that stand behind the following examples is: When executing operations in the normal-order form, since computation of the value is left to the end, the program will run while skipping the invalid expressions.
Also, we can see in evalPrimitive file, what makes the methods return a Failure.

Here are four types (& examples) of semantic errors:

1. Invalid numerical operations.
   Example: Divide by 0
   ```
   (define try
      (lambda (a b)
        (if (= a 0)
            1
            b)))
    (try 0 (/ 1 0))
   ```
2. Evaluating something different than a procedure.
   Example: `(0 1 2)`
3. Mismatching types of operands when applying a primitive procedure.
   Example:
   ```
   (define exception
         (lambda (a b) a))
   (exception 0 (+ "string" 5))
   ```
4. Illegal number of arguments
   Example: `(- 0 1 2)`

Q3

Instead of using valueToLitExp in the substitution model (L3-eval.ts), it is possible to extend the type definitions of the AST to accept values.

3.1. We wish to update the syntax specification of L3 accordingly. We will do so as follows: define/ transfer/ consider `number | bool | string | ect.` (<sexp>) to/ as <cexp> next to `NumExp, BoolExp, StrExp, ect.` This means that we want to identify the actual value of those expressions.

3.2. The updates we suggested before follows with some other updates in the interpreter:

- The method substitute is still needed (even when we don't use valueToLitExp), since substitute are used in order to exchange parameters appearance with their given values.
- The method applyClosure won't call the method valueToLitExp anymore of course, but instead, since we support values now at cexp as described at 3.1, the value is reachable, and we can have it by calling makeCExp instead.

3.3. Both strategies (valueToLit vs. change types) have their pros and cons.

- On the one hand, the option of using valueToLitExp doesn't require any changes (types and the interpreter remain the same) (☺).
  But on the other hand, this option is inefficient (☹).
- Regarding the option suggested in this question, we will say it is prefered since now the AST is smaller, it is easier to understand the program flow (and distinguish between an expression and it's value), and converting (from expression to value and back) is not needed now.

Q4

The valueToLitExp function is not needed in the normal evaluation strategy interpreter (L3-normal.ts). That is because in the Normal-Eval it does not evaluate expressions' value unless it returns the value, so when passing arguments to a closure it delivers the expressions and not the values (unlike the applicative evaluation algorithm).
Also, in the substitution procedure, valueToLitExp procedure is used in order to map the evaluated arguments to expressions to ensure that the result of the substitution is a well-typed AST which can be evaluated. And this of course, is not an issue in the normal evaluation strategy interpreter.

Q5

Normal order evaluation is prefered and will execute faster than applicative order evaluation, when the operands' values are not necessarily needed for evaluation.
In the following example, the computation of `(/ 30 32)` won't be evaluated (since `a` is `#t` and `f` will return `3`) while in applicative-eval, it wold anyway.
For example:

```
(define f
    (lambda (a b c)
        (if a
        b
        c)))
(f (#t 3 (/ 30 32)))
```

In contrast, the following program demonstrates where applicative is faster than normal. In general, applicative eval is prefered when the **value** of an operand is used many times. That way, we avoid reevaluating the same expression.
In the following example, first the expression `(* 2 3)` will be computed, and then x will be substituted by the value 6. In contrast, in normal order, `(* 2 3)` will be computed twice.
For example:

```
(define f
    (lambda (x) (+ x x)))
(f (* 2 3))
```

Part III: Normal Environment Evaluator

3.1. Handling Define in Normal Evaluation

Program	`#lang lazy`
	`(define x (-))`
	`x`
		`#lang lazy`
		`(define x (-))`
		`1`

Output	#<promise:x>	1
Behavior	lazy, normal-order behavior. The computation (\ evaluation) is left to the end, so x is “saved” in the environment as a CExp.	applicative-eval behavior. x is not even being evaluated when the program prints the value `1`.

Now, considering the code found in file test-define-normal.ts, we observe that the implementation of normal evaluation strategy is incorrect when handling the `define` expression (it does not behave as the #lazy language).

- The problem is: The program tries to evaluate the expression we defined as “define”.
  The received error is: `` `Type error: - expects number` ``.

  Lengthy, the value of DefineExp (received as AppExp) is evaluated before the assignment.
  Then, ‘+’ is computed and the value ‘0’ is accepted (even though ‘0’ is not the value of #lazy evaluation to the given expression).
  Since evaluating ‘+’ is is done by reduce, which adds operands’ values to ‘0’, and since ‘-’ evaluation fails, the type error will occur since no operands were received for evaluation.
  In contrast, in lazy, the evaluation of the value is not even taken in to consideration (DefineExp’s value won’t be evaluated, until it has a use).
  That is why, the error received then, won’t be accepted now (throughout defining the expression), not even when it is returned (of course, when It has a use, a computation will be made, and the exception will be received).

- We think that the solution should be: evaluating the primOp only when the defined parameter is being read (delay its’ calculation).

3.3. Normal Order Evaluation - Bonus

We Decided to go for the bonus. As we implemented it, Closure type remained the same, but now the environment structure (ExtEnv) holds two environments- current and next.