PPL - Assignment 3

Part 1: Theoretical Questions

1.

Let is a special form in L3. While in non-special form expressions evaluated by a general evaluation rule, where all the sub expressions are always evaluated, in a Special form expression a special evaluation rule exist. In our case, the binding variables are not evaluated, the let expression only define local variables with the corresponding binding values and replaces every occurrence of the parameter in the body of the function with the corresponding value assigned to it. Furthermore, every let expression can be written as a lambda expression and since a lambda expression is a special form, let expression is also a special form.

2.

Four types of **semantics errors** are:

- Trying to activate an essentially wrong action as a primitive, such as dividing by zero => (/ 5 0)
- Trying to activate a primitive on a non-suitable type (type error) => (+ x y) when x,y are non-numbers
- Trying to activate a closure with Incorrect number of parameters => ((lambda (x y) (+ x y)) 1 2 3)
- Trying to activate non-primitive or non-closure expression as an operator => (3 2 1)

3.1

Updates in syntax:

Changes in L3-ast.ts file:

- import { …, Value } from './L3-value'
- Add the Value expression to the CExp type: export type CExp = AtomicExp | CompoundExp | Value;

3.2

Updates in the interpreter:

Changes in L3-eval.ts file:

- import { …, isSExp } from "./L3-value";
- Changes in L3applicativeEval function:

  Add the following line to the function: isSExp(exp) ? makeOk(exp) :

  (if the exp is a value return it with no change wrapped in a Result)

- Changes in applyClosure function:

  Delete the following line inside the function: const litArgs = map(valueToLitExp, args)

  (we don't need to make the transformation anymore, since Value is a legal expression in the AST tree)

  Change the third input when calling the substitute function from litArgs to args as follows:

  substitute(body, vars, args)

(since as mentioned above, we delete the liArgs variable)

- Changes in valueToLitExp function:

  Delete the whole function

  (since it is no longer in use)

3.3

Using the valueToLitExp function is preferable since it keeps logic and order. In our opinion, the addition of Value to CExp is misleading and illogical, since Value is not an expression. Moreover, its confusing to have those conceptual double meanings, such as: NumExp and a number value, a BoolExp and a boolean value etc. We think it is better to stick with the expressions only.

4.

The **valueToLitExp** function is not needed **in the normal evaluation** strategy interpreter because when we want to substitute var-ref with value in this model we get the values as c-exp which his legal expression in the AST tree so we don't need to convert the value when doing substitute.

5.

Here **applicative order evaluation will execute faster than normal order**:

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

(define f (lambda (a) (square (+ a 1))))

(f 5)
```

In normal order we will evaluate (+ 5 1) twice inside 'square' body, where in applicative order we will evaluate it once, therefore applicative order evaluation is faster.

Here **normal order will execute faster than applicative order evaluation**:

```
(define if-square (lambda (x)

        (if(> 2 1) 1 (* x x))))

(define f (lambda (a) (if-square (+ a 1))))

(f 5)
```

In normal order we will not evaluate (+ 5 1) at all, where in applicative order we will evaluate it once, therefore normal order is faster

Part 3: Theoretical Questions

3.1

#lang lazy
(define x (-))
x

The following program return a #<promise:x> (a promise to supply the value at some point in the future).The program avoids evaluating the val argument in the DefineExp. Afterwards, calling to x, substitute x with (-), but the program still avoids evaluating that argument, since it is not necessary yet (it will be necessary, for instance, when we need to apply a primitive procedure).

#lang lazy
(define x (-))
1

The following program return the value 1, since the program avoids evaluating the val argument in the defineExp, from the same reason we mentioned above. In the applicative evaluation, the val argument (-) will evaluate immediately and return an error since it expected at least one argument and given 0.

The code found in file **test-define-normal.ts does not behave as the #lazy language** because in the current implementation, define evaluate the (-) exp as part of the binding process between x and (-). Specifically, in evalDefineExps implementation if isDefineExp(def) is true, we first evaluate def.val, where in that point at the #lazy language we avoid evaluating def.val .

The solution would make the Env without evaluating def.val yet, by changing the evalDefineExps function, possibly by wrapping every appExp with a promise as implement in the #lazy language.