Assignment 3

Submittors: Moran Neptune 208474544 and Bar Perlman 305026882

Part 1: Theoretical Questions:

1.
The difference is in the evaluation way of the expression.
Evaluating a primitive operator is part of the regular way to evaluate an expression in the applicative / Normal order way, while an expression which is characterized as special form has different way to be calculated.
Lets observe in L3-eval.ts which has been taught in class:
We can see in the function evalExps that there is a different path for calculating the defineExp (special form) in opposed to other expressions such as PrimOp (primitive procedure).
Farther more, we can see that the actual calculating expression to value define in L3 happens at the function evalDefineExp while the one of the PrimOp for the primitive procedure starts at "L3ApplicativeEval" as the other non special form expressions.

2.
In PrimOp, the value of the operator is a structure which indicates that this is primitive operator with its string. The type of this expression is PrimOp.
The interpreter defines the value of the operator it is activated. There is a specific code to deal with each operator and the activation of the operator means calling to this code. L1 languagh using this approach.
In varRef approach the primitive operator value is like procedures defines by the user (closure for example). It means that the type is varRef and in the interpreter the primitive functions are defined as part of of the global environment.
Activation of primitive procedure means get the function by the name of the operator from the environment and activate this function. Scheme languagh uses the varRef approach.

3.
Equivalent:

In both, the following program returns the value 3.

```
((lambda (x y)
        (+ x y)
        )
1 2)
```

Non-Equivalent:

The following code from practicak session #5 represents the non equivalence.
In Normal order, the program stop and return 7 while in applicative order the program entered into infinith loop.

```
(define loop
        (lambda (x)
                (loop x)))
```

```
(define g
        (lambda (x y)
                y))

(g (loop 0) 7)
```

4.
In the function “applyClosure”, after renaming the body of the closure to avoid capturing free variables, we map the evaluated arguments to expressions by calling valueToLitExp with those arguments in purpose to ensure that the result of the substitution is a well typed AST which can be evaluated.

5.
This function is not needed in the environment interpreter because we don’t need to substitute variables with values. We are just using varRef and new environments in the computations.

6.
In case that applicative order cause the program to stuck in infinith loop as in the example that was given in section 3 for the non equivalent executions.
In addition, there are two more reasins discussed in class:
applying a function with arguments where some of them causing to an error (1) or side effects (2) when they are evaluated to a value. For example:

(1)
```
(define test
        (lambda (x y)
                if(= x 0)
                        0
                        y)))

(define zero-div
        (lambda (n)
                (\ n 0)))

(test 0 (zero-div 5))
```

(2)
```
(define applic
        (lambda ()
                (display ‘applic)
                0))

(define test
        (lambda (x) 1))
(test (applic))
```

7.
When a function is using the same received argument many times and its evaluation takes a lot of cpu runtime then executiong the program in applicative eval is will decrease cpu runtime significantly.
We can see it in the following example:

```
(lambda (x)
        (+ x x x)
        (* 2 3 4)))
```

In the example above the evaluation of the expression (* 2 3 4) is performed only once in the applicative order while in Normal order it is performed3 times so its better to run it in applicative order.

8.
- In class we saw that we can write let as lambda expression, so its evaluation process will include the following:
(a) write it as a lambda expression.
(b) evaluate it as lambda expression, which does creation of a closure.
The code that responsible for the above is located in rewrite.ts:

```
const rewriteLet = (e: LetExp): AppExp => {
   const vars = map((b) => b.var, e.bindings);
   const vals = map((b) => b.val, e.bindings);
   return makeAppExp(
         makeProcExp(vars, e.body),
         vals);
}
```

- Another strategy we saw is to evaluate the let expression directly as part of the environment model. This means computing values, extend the environment and evaluating body without creating a closure.
The suit code in L4-eval.ts:

```
 const evalLet = (exp: LetExp, env: Env): Value | Error => {
   const vals = map((v: CExp) => applicativeEval(v, env), map((b: Binding) => b.val,
exp.bindings));
   const vars = map((b: Binding) => b.var.var, exp.bindings);
   if (hasNoError(vals)) {
      return evalExps(exp.body, makeExtEnv(vars, vals, env));
   } else {
      return Error(getErrorMessages(vals));
   }
}
```