Assignment 3

1. Primitive procedure – procedures which are pre-evaluated and are fixed. In contrast, special procedure requires special evaluation rule is not fixed in its value.

Code:

Example of a primitive procedure – well defined.

```
proc.op === "not" ? ! args[0] :
```

Example of a special procedure - if expression that has special evaluation rules:

```
const evalIf4 = (exp: IfExp4, env: Env): Value4 | Error => {
    const test = L4applicativeEval(exp.test, env, true);
    return isError(test) ? test :
        isTrueValue(test) ? L4applicativeEval(exp.then, env, true) :
        L4applicativeEval(exp.alt, env , true);
};
```

2.

```
With shortcut semantics
eval(<OR-exp exp>,env) =>
    if exp.params is empty
            return false
        let test: Value: eval(first(exp.params),env)
        if test is considered a true value
            return true
        else
            return eval(make-OR-exp (rest(exp.params),env))

according to non-shortcut semantics
eval(<OR-exp exp>,env) =>
    if exp.params is empty
        return false
    else
        let newOrExp = make-OR-exp(rest(exp.params),env))
        return  eval(first(exp.params)) OR eval(make-OR-exp
(rest(exp.params),env))
```

3. We'd rather use the PrimOp. If we want, for instance, to add another primitive procedure to the language using varRef, we should implement the procedure, make the binding of the var to the implementation and then add the binding to the global environment. In primOp, all we need is to create the implementation.

4. If we want to get an answer from a procedure regardless of the fact that some of its content may lead to infinite loop or error, we might want to use normal-eval rather than application-eval. The point is - normal eval only evaluates expressions which are needed for the calculation of the program. Thus, if some expression holds side effects like massages, and it is not needed for the calculation of the program, the side-effects won't be executed. If that's our wish, we should choose that.

5. In contrast, if we want to pay attention for any possible error/infinite loops in our code, we should use application-eval rather than normal-eval. In addition, the behavior on side-effects is different. If we want to print all the massages expressions written in our code(maybe for debugging purposes) we should use application-eval because normal-eval only evaluates expressions which are necessary for the computation of the program. Thus, the massages within these not needed expressions won't be printed. We might also use it to avoid the repetitive calculations, as might occur with normal-eval.

6. The substitution model works great when speaking of enabling recursions and avoiding free variable capture. Its main disadvantage is its run-time and its memory-usage . Substitution is consisted of two parts:
   a. consistent renaming of the expression E and the expression in S. Where E = the body of the procedure and S = substitution.
   b. Simultaneous replacement of all free occurrences of the variables of S in the renamed E by the corresponding renamed expressions of S.

We can clearly see the operation of renaming enforces the creation of new body for the procedure. Why? See for instance the procedure:

```
(define z (lambda (x) (* x x)))

(((lambda (x) (lambda (z) (x z))) ; 1
  (lambda (w) (z w)))             ; 2
 2)
```

In order to evaluate this expression, we must rename the variable z in lambda(z) so to avoid free variable capture.

The main problem of this approach is that substitution requires repeated analysis of procedure bodies. In every application, the entire procedure body is repeatedly renamed, substituted and reduced. These operations on ASTs actually **copy** the structure of the whole AST - leading to extensive memory allocation / garbage collection when dealing with large programs.

The environment-model supplies neat and effective solution to this problem. It replaces substitution and renaming by a data structure – the environment – which is associated with every procedure application. It is created when a closure is created and accessed when a closure is applied. Thus we avoid copying the structure of the whole AST.

In simple words, when dealing with large programs we should definitely switch to the environment-model.

7.

```
(L3
    (define try
        (lambda (a b)
            (if (= a 0)
                1
                b)))
    (try 0 (/ 1 0)))
```

In normal order, this program returns 1. In applicative order, it throws a divide by 0 exception.

```
(L3
    (define loop (lambda (x) (loop x)))
    (define g (lambda (x) 5))
    (g (loop 0)))
```

In normal order, the application (loop 0) is not evaluated and this program returns 5. In applicative order: the call (g (loop 0)) enters into an infinite loop.

```
(L3
    (define f (lambda (x) (display x) (newline) (+ x 1)))
    (define g (lambda (x) 5))
    (g (f 0)))
```

With applicative-eval, this program prints `0` then returns 5. In contrast, in normal-eval, this program returns 5 **without printing** anything.

8. The valueToLitExp procedure is not needed in normal order interpreter because the step of argument evaluation occurs just before a primitive procedure is applied:

   isAppExp(exp) ? L3applyProcedure(L3applicativeEval(exp.rator, env), exp.rands, env).
   Thus, the value of the arguments needs to be evaluated only once. As a consequence of that, we don't need to recondition the AST tree for further evaluations of a given argument. It was needed only if the evaluation of an argument is used itself as an argument for another procedure.

9. The valueToLitExp procedure is not needed in the environment interpreter because whenever we encounter procedure, we create closure that binds its variables to the given arguments via environment. In other words, we don't substitute variables with given arguments each time we face procedure. Instead, we save the binding in the environment. So, if a value that was evaluated is used as an argument for another procedure, we just set up the binding: var->value, and when evaluating the body of the procedure, we replace var with value using applyEnv.

10.
   Substitution model:
   Using short semantics, in L3 let expressions are translated into an expressions of the form: AppExp(ProcExp, vals). Because ProcExp creates a closure, let also creates a closure.

```
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);
}
```

In applicative eval:

```
const evalProc = (exp: ProcExp, env: Env): Value =>
  makeClosure(exp.args, exp.body);
```

In normal eval:

```
isProcExp(exp) ? makeClosure(exp.args, exp.body) :
```

Although it is implemented this way, it is not necessary. Closure is created in order to maintain the structure of the procedure until arguments are applied. In let expression arguments are applied at the moment. Thus, it is not necessary.

Env model:

In this model a closure isn’t created when evaluating an eval expression using applicative – eval and normal-eval. The reason is that there is no need to bind between the procedure and the environment in which it is evaluated, because we apply the arguments to the procedure at the moment of evaluation of let and they are evaluated the the moment.

```
const evalLet4 = (exp: LetExp4, env: Env): Value4 | Error => {
  const vals = map((v) => L4applicativeEval(v, env), map((b) => b.val,
exp.bindings));
  const vars = map((b) => b.var.var, exp.bindings);
  if (hasNoError(vals)) {
    return evalExps(exp.body, makeExtEnv(vars, vals, env));
  } else {
    return Error(getErrorMessages(vals));
  }
}
```