in1)What is the difference between 'special form' and 'primitive operator'? Demonstrate your answer by pointing to code fragments in the interpreter of L3.

The difference between a special form and a primitive operator is in their evaluation.
In special form we evaluate expressions according to a special evaluation rule and while in primitive operator the evaluation is done in a linear way (we can assume that), one expression after the other.

```
const L3applicativeEval = (exp: CExp | Error, env: Env): Value | Error =>
    isAppExp(exp) ? L3applyProcedure(L3applicativeEval(exp.rator, env),
                                     map((rand) => L3applicativeEval(rand, env),
                                         exp.rands),
                                     env) :
```

```
//only one expression (then or alt) is evaluated according to the result of test.
const evalIf = (exp: IfExp, env: Env): Value | Error => {
    const test = L3applicativeEval(exp.test, env);
    return isError(test) ? test :
        isTrueValue(test) ? L3applicativeEval(exp.then, env) :
        L3applicativeEval(exp.alt, env);
}
```

2) Define an evaluation rule for a new operator 'OR': (a) according to shortcut semantics; (b) according to non-shortcut semantics.

shortcut semantics:

```
 eval (<or-exp exp>,env) =>
   if exp.rators is empty return false
   let head: Value = eval(first(exp.rators), env)
   if head is considered a true value
        return true
   else
      let or-exp: Value =make or-expression from rest(exp.rators)
      return eval (or-exp, env)
```

non-shortcut semantics:

```
eval (<or-exp exp>,env) =>
  if exp.rators is empty return false
  let or-exp: Value =make or-expression from rest(exp.rators)
  let tail : Value = eval (or-exp, env)
  let head: Value = eval(first(exp.rators), env)
  if head is considered a true value
      return true
  else if tail considered a true value
      return true
  else return false
```

3) Two ways of representation for primitive operators were presented in class and in the practical session: PrimOp vs. VarRef. Which one would you prefer, in terms of 'language maintenance'

We would prefer using varRef in terms of 'language maintenance'.
with this implementation, it is easy to add a primitive to the interpreter:
we only need to add the proper binding in the initialization of the global environment.

4) What are reasons that would justify switching from applicative order to normal order evaluation? Give an example.

When there are multiple expressions and we only need the values of some of these expressions (according to some strategy) we will switch from applicative order to normal order evaluation;
this is especially useful when some of the expressions might not terminate – infinite loop, or when one of the expressions returns an error after evaluation.

For example:

```
(define (p) (p))
(define (test x y) (if (= x 0)
                       0
                       y))

(test 0 (p))
```

In Applicative evaluation we will enter an infinite loop while in normal evaluation we will evaluate first the procedure, then substitution is performed on the non-evaluated parameters and reduction. The result of that is 0.

5) What are reasons that would justify switching from normal order to applicative order evaluation? Give an example.

The main reason for switching from normal eval to applicative eval is if we have a repeated expression – in applicative eval it will be evaluated only once and then will be substituted while in normal it will be evaluated again and again. Therefore, it is more efficient. In addition, if side effects are needed we would choose applicative.

<u>For Example:</u>
(define foo (lambda (x) (display x)(newline) x))
(define bar (lambda (x) (+ x x x x x)))
(foobar (foo 5)) ; // (foo 5) will only be evaluated once in applicative eval.

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

"The substitution operation applies the pairing of procedure parameters with the corresponding arguments. Renaming is an annoying by-product of substitution - and it cannot be "compiled away" easily in this model - we need to rename the body of the closure each time it is applied (convince yourself of this by finding an example that requires repeated renaming).

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. In fact, the substitution interpreter we reviewed is so slow that it is barely usable."

(from class material)

<u>In general</u> – substitution is inefficient as we change the AST (possibly multiple times).

<u>Example:</u>

(+ (x z) ((lambda (x) x) y)) o {x = (lambda (x) x) ,y = 7,z =8}

<u>substitution model:</u>
The AST will change so that the bound variables x and y will be renamed. Then, the AST will change again to replace those renamed variables with the new values (x will have the result of the evaluated procedure and z will have 8). Note that this replacement requires both evaluating the parameter expressions and then converting the to literals (only to be evaluated again) Only then the expression can be evaluated.

<u>environment model</u>

A new environment will be created, where x and y will have their values. When the expression will be evaluated, the bound variables will look their values up in it. No change to the AST is required

9)The valueToLitExp procedure is not needed in the environment interpreter. Why?

As explained previously, the procedure is called only when a substitution within the AST is required. In the environment interpreter, the value is stored within the environment and there is no need to substitute it within the AST, therefore the function is not used.

10)Does the evaluation of 'let' expression involve a creation of a closure? Refer to various strategies of evaluation of let in different interpreters discussed in class, and provide justification by showing code samples from the interpreters code.

Substitution: We create a closure.

We replace let expressions with ProcExp

```
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);
}
```

We then create a closure for these ProcExps. From L3 eval:
isProcExp(exp) ? makeClosure(exp.args, exp.body) :

Environment: We evaluate directly without creating a closure, and extend the environment.

```
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));
```

```
    }
}
```