Part 1: Concept Questions

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

   answer:
   'special form' it is a compound expression which is not evaluated like regular compound expressions. 'primitive operators' are expressions whose evaluation process and hence their values are built into the language tools. 'Primitive operator' is evaluated in a sequential order from the first argument according to the operator.
   We will demonstrate our answer using define as 'special form' and +procedure as 'primitive operator':

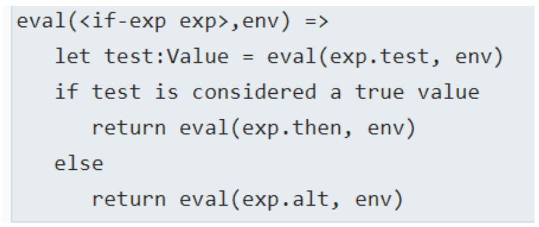
```
const evalDefineExps = (exps: Exp [], env): Value | Error => {
let def = first(exps);
let rhs = L3applicativeEval (def.val, env);
if (isError(rhs))
    return rhs;
else {
    let newEnv = makeEnv (def.var.var, rhs, env);
    return evalExps(rest(exps), newEnv);
 }}
```

   we can see that evaluation of define expression is the evaluation of the value and then binding the variable and the evaluation result to the Global Environment.

```
export const applyPrimitive = (proc: PrimOp, args: Value []): Value | Error =>
proc.op === "+" ? (allT (isNumber, args) ? reduce((x, y) => x + y, 0, args) : Error ("+ expects
numbers only")) :
```

   The evaluation of +procedure expression is calculating the sum of all the arguments in a sequent order.

2. **Define an evaluation rule for a new operator 'OR':**
   (a) according to shortcut semantics;
   (b) according to non-shortcut semantics.
   Use the formal form presented in class, *e.g.*:



```
eval(<if-exp exp>,env) =>
   let test:Value = eval(exp.test, env)
   if test is considered a true value
      return eval(exp.then, env)
   else
      return eval(exp.alt, env)
```

answer:

(a)

```
eval(<or-exp exp>,env) =>
let first:Value = eval(exp.first, env)
let second:Value = eval(exp.second, env)
return (first | second)
```

(b)

```
eval(<OR-exp exp>,env) =>
    {R.reduce((acc,curr)=>(acc | curr), false, exp.args)}
```

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' (that is, how easy to make changes to the interpreter when the language definition is changed).**

answer:

Because one can add a primitive operator, change it by edit the initialized Global Environment (the function makeGlobalEnv) or remove it, we prefer the VarRef version in terms of 'language maintenance'. Furthermore this option makes the interpreter more general than doing it in the interpreter itself.

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

answer:

The reason that would justify switching from applicative order to normal order evaluation is risk of non-termination, even when the guilty argument will be used in the call, is one of the reasons to use instead normal order evaluation, which may have better termination properties.
For example:

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

We will never ever get to finish the call to the function test, since evaluation of its arguments oes not terminate.
Another example is:

```
(define loop  (lambda (x)
      (loop x)))
(define g (lambda (x) 7))
(g (loop 0))
```

On normal we return 7, and in applicative we will get infinite loop.

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

   answer:
   The reason that would justify switching from normal order to applicative order evaluation is that the arguments are evaluated immediately in applicative order, before the procedure is reduced. As a result, there will not be a case of repeating same large computation.

```
( (lambda (y)
      (* y y y) )
      (+ 2 ( + ( /  8 2)  4 ))
```

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

   answer:
   The main reason for switching from the substitution model to the environment model is that the environment model it is a optimization of substitution applicative model of the operational semantic.

   As we learn in substitution applicative model every application of a procedure the entire procedure body repeatedly renamed, substituted and reduced.

   As result it reduces the runtime and memory efficiency.

   As we learn that is in contrast to the environment model which replace the substitution and renaming by the environment data structure which is associated with every procedure application.

   Is it created when a closure is created accessed when the closure is applied.

7. **Give code examples for equivalent and non-equivalent executions of applicative and normal order.**

   answer:
   Examples for equivalent executions of applicative and normal order (the code is in the next page):
   The first example will do the same thing in normal, applicative.
   The second example will return 'neq in normal, applicative.
   Examples for non-equivalent executions of applicative and normal order(the code is on the next page):
   The first example will get into an infinite loop in applicative order but will return 7 in normal order.
   The second example will get into an infinite loop in applicative order but will return 2 in normal order.

```
;first example for equivalent is:
( (lambda (x)
   x)
 ( + 1 2 ) )

;second example for equivalent from midexam 2017:
(define y 7)

(define double (lambda (x)
     (* x 2)))

(define g (lambda (x)
     (if (= y (double y)) 'eq 'neq)))


;first example for non-equivalent is:
 (define loop (lambda (x)
         (loop x)))

 (define g (lambda (x) 7))

 (g (loop 0))

;second example for non-equivalent from midexam 2008 is:
 (define f (lambda (x y)
         (if (> x 0)
             x
             (f (+ x y) y))))
;when we try to run it with:
 (f 2 (f -1 0))
```

8. The valueToLitExp procedure is not needed in the normal order interpreter. Why?

<u>answer:</u>

Normal-evaluation does not evaluate expressions value unless it returns the value.

When applying a procedure, arguments are evaluated only when needed to use them in the body.

That is in contrast to applicative-evaluation which arguments to closure are evaluated immediately before using them in the closure calculation and the evaluation results are instead to the body but if we will insert value to the body, we will get an invalid AST, so we use valueToLitExp to translate the values to the corresponding AST so the body will be valid.

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

<u>answer:</u>

The environment-evaluator uses a data structure that keeps for every variable its value, and it exchange the variable with its value only when the program really need to use this variable. Expressions are evaluated with respect to an environment (replaces the former substitution). Therefore, the environment-evaluator doesn't need valueToLitExp.

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.**

the way we change 'let' to closure depends on the model that we use.
In the old models that we used we were changing a let expression to a closure like in the function rewriteLet:

```
/* Purpose: rewrite a single LetExp as a lambda-application form Signature: rewriteLet(cexp)
Type: [LetExp => AppExp] */

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

however, in the L4 model, we used the evalLetL4 which didn’t create a closure and saves the relevant data in the environment and delayed the building of the closure.