Assignment 2 - Theoretical Questions

1. There is no need for functions with multiple bodies in pure functional programming because in pure functional programming there are no side effects (it's needed only when there're side effects). The value of a program is the value of the last expression. If there are no side effects the value of the last expression is not affected by the expressions preceding it. In L3 it's necessary to calculate only the 'defines', after which you can skip to the last expression.
2. The answers:
   a. What defines the language and distinguishes it from other languages are the various syntactic structures, and in particular the collection of special forms. In other words, adding a new special form to a language creates a new language. The way special forms are calculated is different from the primitive operator activation where the operator and operands are calculated first and then the operator is applied to the placement. For example, for the special form if there is no need to calculate the three sub-expressions (test, then, else) but only two of them. Also, we will treat the addition of primitives as updating the interpreter and not as creating a new language.
   b. In order to support shortcut semantics, 'or' must be defined as a special form, otherwise all its parameters will be calculated first, the same as when using an operator, even if it is primitive. If this semantics is not required, we can treat 'or' as a primitive operator as is done in L1-L3 languages.
3. The answers:
   a. The value of the program is 3. The bindings that 'let' creates are all performed separately. Thus, when there is a reference to a variable within the binding part of the 'let', the performance will be made based on definitions that were made before the 'let'. Therefore, at the line (y (* x 3)) the value of x is 1.
   b. The value of the program is 15. The bindings in 'let*' are performed sequentially. Thus the value of the line (y (* x 3)) is in relation to the value of x binded in the binding part of 'let*' is 5.
   c.
```
(define x 1)
(let ((x 5))
  (let ((y (* x 3)))
     y)
)
```
   d.
```
(define x 1)
(
(lambda (x) (
(lambda (y) (
  (lambda () y)
```

```
    )
  ) (* x 3)
  )
 )
5)
```

4. The answers:
   - **a.** The role of the function valueToLitExp in L3 is converting the list of values to be placed into a corresponding list of expressions in applyClosure function in L3-eval.ts. Required for type compatibility considerations: the body of the placement procedure is defined in the CExp concepts of the parser, while the values to be placed are already in the Value concepts of the interpreter.
   - **b.** The valueToLitExp function is not needed in the normal evaluation strategy interpreter because there is no need to convert the operands back from values to expressions in the case of isClosure in the L3normalApplyProc (value2LitExp) procedure because they are not calculated (i.e. their type is CExp and not Value yet). The normal order evaluation strategy directly substitutes variables with expressions, since the arguments aren't evaluated before substitution, so we can directly substitute them without a type error.
   - **c.** The valueToLitExp function is designed for the substitution model to convert computed values back into literal expressions suitable for substitution. In the environment model, this conversion is unnecessary because values are directly associated with variable names in the environment and are looked up as needed. Thus, the environment model inherently handles values more directly and efficiently, without requiring an intermediate conversion back to expressions. Application of a function involves evaluating its body in a new environment frame that includes the argument bindings. This is different from the substitution model, where you replace variables with values in the function body textually.
5. The answers:
   - **a.** Switching from applicative to normal is more efficient in cases where there are programs where their calculations are not needed. Moreover, there are cases where there's division by zero in alt (in if statement) and we'll never encounter this error in this case: ((lambda (x y z) (if x y z) # 3 (/ 30 0)) but the error will be executed in applicative form. In addition, there're cases where we'll be encountered with an infinity loop in applicative form but not in normal form, let's have a look to this case:

```
(define loop (lambda () (loop)))
(define f (lambda (x) 5))
(f (loop))
```

Another example:

```
(
  (lambda (x y z) (if x y z))
   #t 3 (sqrt 2)
  )
```

In this example the value of (sqrt 2) is never needed. Applicative will calculate (sqrt 2) even though it is not needed. Normal form will not calculate (sqrt 2).

b. Switching from normal to applicative is more efficient where the same program is referenced multiple times. The program will be calculated one time instead of calculating the program in every time it is referenced.

Example:

```
(
 (lambda (x) (+ x x))
 (sqrt 2)
)
```

In normal form (sqrt 2) will be calculated twice. In applicative only once.

6. The answers:

a. The environment model naturally handles variable scopes and bindings through its use of environment frames. Each frame corresponds to a different scope, ensuring that variable names are correctly resolved without needing to rename variables. This separation of environments ensures that variables with the same name but in different scopes do not interfere with each other, making renaming unnecessary. Since each function call creates a new environment frame, there is no risk of variable name clashes. Variables in different scopes are kept in separate environment frames, and the interpreter can resolve each variable name to the correct value based on the current environment.

b. Renaming in the substitution model is used to avoid conflicts and ensure that variable bindings remain clear and unambiguous. However, when the term being substituted is closed, renaming is unnecessary because the closed term is self-contained and does not interact with any external variable bindings. This makes the substitution straightforward and safe without any risk of name clashes. When substituting a closed term, there is no risk of variable name clashes because all variables in the term are already bound within the term itself. This ensures that no external bindings (from the surrounding context where the substitution is taking place) can interfere with the variables within the term.

Question 2.d

1. The converted program from ClassExps to ProcExps as defined in 2.c is:

```
(define pi 3.14)
```

```
(define square (lambda (x) (* x x)))
(define circle
 (lambda (x y radius)
  (lambda (msg)
   (if (eq? msg 'area)
     ((lambda () (* (square radius) pi)))
     (if (eq? msg 'perimeter)
       ((lambda () (* 2 pi radius)))
       #f)))))
(define c (circle 0 0 3))
(c 'area)
```

2. The expressions which are passed as operands to the L3applicativeEval function during the computation of the program, (after the conversion), for the case of substitution model are:

- 3.14
- (lambda (x) (* x x))
- (lambda (x y radius) (lambda (msg) ...))
- (circle 0 0 3)
- circle
- 0
- 0
- 3
- (lambda (msg) (if (eq? msg 'area)…))
- (c 'area)
- c
- 'area
- (if (eq? 'area 'area)...)
- (eq? 'area 'area)
- eq?
- 'area
- 'area
- ((lambda () (* (square 3) pi)))
- (lambda () (* (square 3) pi))
- (* (square 3) pi)
- *

- (square 3)
- square
- 3
- (* 3 3)
- *
- 3
- 3
- pi
- 28.26

3. The environment diagram for the computation of the program (after the conversion), for the case of the environment model interpreter: