1. What is the role of the function `value->lit-exp` in the substitution-evaluator? Why is it needed?

   Answer:

   the Purpose of the value->lit-exp function is to transform a value into a literal expression denoting this value. The value is of one of the types :
   Num-Exp ,Bool-exp ,Prim-op ,Proc-exp.
   It is needed because there is a typing problem with previous operations defined in L1 – in app-exp case, they transform var-ref expressions into actual values, and then the app-exp is reduced. So the app-exp after the substitution may not be a valid AST (may contain values, not expressions).
   for example -
   (define func (lambda (x) (* x x))
   (func 5)

   Func Will be saved as (closure [(var-decl x)] [(app-exp (prim-op *) ((var-ref x) (var-ref x))

   and without the role the body would be substituted into (app-exp (prim-op *) (5 5)), and the resulting body is not a valid AST – 5 is a value, not an expression, and we can't apply the app expression.

2. The normal evaluator doesn't need value->lit-exp. Why?

   Answer:

   we do not need to turn the values back into expression before we apply the substitution in the body, since the rands are passed as non-evaluated expressions. So the rands are reduced only when the app-exp is reduced , and the problem we mentioned in the previous question is not relevant here (we don't evaluate var-ref-exps before reducing app-exp).

3. The environment-evaluator doesn't need value->lit-exp. Why?

   Answer:

   The environment-based model is a **lazy operation**: instead of replacing var-refs before reduction, it replaces it in the reduction process. So the problem we mentioned in the first question is not relevant here (we don't evaluate var-ref-exps before reducing app-exp).

4. In the substitution model procedure application involves renaming. Find an example that requires repeated renaming.

   Answer:

   (define z (lambda (x y) (* x y)))

   (((lambda (x) (lambda (z) (x z))) ; 1
   (lambda (y) (z y 3))) ; 2
   2)

5. List the advantages and disadvantages of keeping a small language core and a large library of derived expressions.

Answer:

Advantages - It has a very simple syntax, with few details, can be taught quickly. The derived expressions allow a more understandable language, but we can write code without them.

Disadvantages –after we've learned the syntax it's easier to perform complicated actions – there are more options in the language.

6. What are the reasons for switching from the Substitution Model to the Environment Model?

   Answer:

   The main problem of substitution model 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. The environment model, : instead of substitute before reduction, is substituting in the reduction process, only if we reached the var-ref in the body. So we don't substitute all the vars in every application, only when it's needed (although we do compute all the vars ahead). That prevents unused memory allocation / garbage collection, and the process is faster.

7. Can the apply-procedure procedure have as a parameter a Racket closure (i.e. not a primitive)? explain and address interpreters.

   Answer:

   No, because the function need to verify two conditions if the first parameter is a prim-op or closure, a Racket closure is neither of them, and therefor will not return those conditions.

8. Explain why we can typecheck letrec expressions without specific problems related to recursion and without the need for recursive environment like we had in the interpreter.

   Answer:

   Because the procedure of type-check is doing in static way before the evaluations, and therefore when we procedures all the type checks we can access all the bonds in letrec.

9. In the type-inference.rkt implementation - we represent Type Variables (TVar) with a content field (which is a box which contains a Type Expression value or #f when empty). In this representation, we can have a TVar refer in its content to another TVar - repeatedly, leading to a chain of TVars. Design a program which, when we pass it to the type inference algorithm, creates a chain of length 4 of Tvar1->Tvar2->Tvar3->Tvar4. Write a test to demonstrate this configuration.

   Answer:

   Program: (lambda(a) (lambda(b) (lambda(c) (lambda(d) (#f)))))

   Test:

   (typeof-exp (parseL5 (lambda(a) (lambda(b) (lambda(c) (lambda(d) (#f))))) (make-empty-tenv)) => #f.