PPL Assignment 2

Part 1: Theoretical Questions

1) Special forms required in programming languages because it allows us to use our own evaluation rules and therefore they can't be defined as primitive operators. That is, Scheme (or any other programming language) doesn't evaluate all the subexpressions, instead, each special form has its own evaluation rules. For example: 'if' special form, (if #t 1 0).
2) **First** method include more than 1 expression, that can be done in parallel:
   (define x(+ 3 (* 2 2)))
   (define y(+ 2 (- 5 2)))

   _____

   Second method include more than 1 expression, that can't be done in parallel :
   (define x (+ (* 2 3) (* 4 5)))
   (define y (+ 2(+ x 3)))

3) There Is **not** a program in L1 which cannot be transformed to an equivalent program in L0. The reason is that the word 'define' helps us to express compound expressions by giving names, but any such name can be "replaced" by the compound expression itself.
4) (define factorial
   (lambda(n)
   (if (= n 0)
   1
   (* n (factorial (- n 1))))))
   Explanation: In L2 there isn't a way to define recursive call and use it without the special form "define" , So in L20, there isn't a way to transform this L2 statement to a L20 statement.
5) **- map**: can be applied in **parallel**. Because there is no dependence between the values in the returned list, we can calculate for each value itself the change from the given function.
   - reduce: should be **sequential**. Because each 'calculation' in the reduce procedure depends on the result that came out before it (according to the order of the list).
   - filter: can be applied in **parallel**. Because there is no dependence between the values in the list, we can calculate for each value itself whether it 'passes' the given filter.
   - all: can be applied in **parallel**. Because there is no dependence between the values in the list, we can calculate for each value itself whether it 'passes' the given boolean function.
   - compose: should be **sequential**. Because each 'calculation' in the compose procedure depends on the result that came out before it. For example, if we compose on a multiplication and addition operation, the order of the calculation is important (if multiplication first then addition or the opposite).

6) Let us explain the evaluation of this program :
   The first lambda that getting in-progress is :
   lambda c (p34 ‘f)
   5
   that lambda , have a C-exp , which is class & method of the class .
   Then, the C-exp ‘p34’ getting in-progress and with first of 3 and second of 4.
   The function is called , and letting (a= 3, b=4 , c=2) taking place,
   As the c of the first lambda isn’t a local variable - so the general value that gets defined at the second sentence “defince c 2 “ taking place for this c .
   In conclusion , the program will lead to print : **‘9’** which is 2+3+4 .

Part 2: Design by Contract

1) ; Signature: **append(lst1 lst2)**
   ; Type: **[List(T) * List(T) => List(T)]**
   ; Purpose: **returns the concatenation of two lists**
   ; Pre-conditions: **lst1 and lst2 are lists**
   ; Tests: **(append '(1 2) '(3 4)) => '(1 2 3 4)**
2) ; Signature: **reverse(lst)**
   ; Type: **[List(T) => List(T)]**
   ; Purpose: **reverses a list**
   ; Pre-conditions: **lst is a list**
   ; Tests: **(reverse '(1 2 3)) => '(3 2 1)**
3) ; Signature: **duplicate-items(lst dup-count)**
   ; Type: **[List(T) * List(number) => List(T)]**
   ; Purpose: **duplicates each item of lst according to the number defined in the same**
   ; **position in dup-count. In case dup-count length is smaller than lst, dup-count should**
   ; **be treated as a cyclic list**
   ; Pre-conditions: **lst is a list and dup-count is a list of natural numbers**
   ; Tests: **(duplicate-items '(1 2 3) '(1 0)) => '(1 3), (duplicate-items '(1 2 3) '(2 1 0 10 2)) => '(1 1 2)**
4) ; Signature: **payment(n coins-lst)**
   ; Type: **[number * List(number) => number]**
   ; Purpose: **returns the number of possible ways to pay the money with these coins**
   ; Pre-conditions: **coins-lst is a list**
   ; Tests: **(payment 10 '(5 5 10)) => 2, (payment 5 '(1 1 1 2 2 5 10)) => 3**
5) ; Signature: **compose-n(f n)**
   ; Type: **[(T=>T) * number => (T=>T)]**
   ; Purpose: **gets an unary function f and a number n and returns the closure of the n-th self-composition of f**
   ; Pre-conditions: **f is unary function and n > 0**
   ; Tests: **(define mul8 (compose-n (lambda (x) (* 2 x)) 3)); (mul8 3) => 24**