Assignment 2 part 1:

1) When we define an evaluation function for a language, there are some expressions that do not fit in the general definition of how we evaluate atomic or compound expression. For that reason, we define special forms, which are evaluated in non-standard way (not a procedure application). An example for why we cannot define them as primitive operator, let's look at the special form we saw in L1: $(define < var > < exp >)$. When evaluating a $define$ expression the sub-expression is not evaluated. In contrast to primitive operators which we evaluate all its sub-$< var >$ expressions.

2) L1 program that can be done in parallel:

```
(+ 1 1)
(* 2 1)
```

L1 program that cannot be done in parallel:

```
(define x 2)
(define y 1)
(+ y x)
```

Here the interpreter cannot run a thread that evaluates only the last expression.

3) We can transform every program in L1 to an equivalent L0 program.
For each $(define < var > < exp >)$ we can replace all the occurrences of $< var >$ that comes after the define, with$< exp >$ and removing the $define$ expression. When the interpreter evaluates $< var >$ after it was defined, it translates it to its bounded value which has the same result as replacing $< var >$ with $< exp >$. Since every program in L1 eventually stops, we have finite only finite replacements to make. Also, since L1 is functional, we know that $< var >$ will not be mutated so we will not lose information when replacing its occurrence.

4) Here we can use the fact that there are programs in L2 that never stop, to write a program that needs to have infinite replacements of $< var >$ in the 'define' expression. An example for such a program:

```
(define id
   (lambda (x)
      (if (> x 0)
         (id x)
         x
      )
   )
)
(id 1)
```

Here 'id' is a closure of a recursive function which is infinite when calling (id 1). We cannot exclude the 'define' expression since it requires infinite replacements of the closure in the recursive call.

5) –map: **Can** be done in parallel. The procedure 'f' map gets as input, acts independently on the elements of the list, since L3 is functional with no side effects. For each element 'x' in 'lst' we simply calculate (f x) which is independent from calculations made on other elements of the list.

-reduce: **Cannot** be done in parallel. Since reduce is in essential a function that accumulate the result of the 'reducer' on the list, each application of 'reducer' depends on the previous results. This property makes it impossible to apply the 'reducer' on each element of the list in parallel.

-filter: **Can** be done in parallel. For each element 'x' in 'lst' we return a boolean expression (pred x), where 'pred' is the function filter gets as input. This calculation on 'x' is independent from other calculations made on elements of the list, since L3 is functional with no side effects, which means that we can evaluate (pred x) for each 'x' in 'lst' in parallel.

-all: **Can** be done in parallel. Can be explained exactly the same way we explained 'filter'.
In addition, we can simply transform any 'all' function to a **one** 'filter' function by:

```
(define all
   (lambda (pred lst)
      (eq? (filter (not pred) lst) '())
   )
)
```

Using the fact this fact, together with the fact that filter can be done in parallel, we conclude that also 'all' can be done in parallel.

-compose: **Cannot** be done in parallel. Since function composition does not commute, we cannot grantee that different order of composition will result the same function. Better explained with example:

```
(define f (lambda (x) (+ 1 x)))
(define g (lambda (x) (*x 2)))
```

composition of (list (f g)) results a different function since (f (g 2)) => 5 and (g (f 2)) => 6, so the composed functions are not equivalent. For that reason 'compose' cannot be done in parallel.

6) The value of the program is **9**.

When evaluating the expression ((lambda (c) (p34 'f)) 5) we basically return the value of the application of (p34 'f). When evaluating (define p34 (pair 3 4)) above, we create biding between p34 to the evaluation of (pair 3 4), which is a class expression that eventually creates a biding between 'f' to (lambda () (+ a b c)). Now, since a,b,c are variable reference, we need to check where is their closest declaration when climbing up the AST. For a and b, the declaration is when calling (pair 3 4) which binds {a=3, b=4}, notice that (define b 1) is higher in the AST. For 'c' there is only one option for evaluation since traveling up in the AST at this point gives us the declaration of (define c 2).

So we can conclude that the evaluation of (lambda () (+ a b c)) at this point gives us (lambda () (9)). Now, since it is independent from the substitution of 'c' in (lambda (c) (p34 'f)), can evaluate the whole expression:

((lambda (c) (p34 'f)) 5) => ((lambda (5) (lambda () (9))) => 9.

Part 2 – contracts:

```
;Q2.1
;Signature: append(lst1, lst2)
;Type: [List(T1) * List(T2) => List((T1|T2))]
;Purpose: return the concatenation of two lists
```

```
;Pre-Conditions: True
;Tests: (append '(1 2) '(3 4) -> '(1 2 3 4))


;Q2.2
;Signature: reverse(lst)
;Type: [List(T) => List(T)]
;Purpose: return the reverse of a list
;Pre-Conditions: True
;Tests: (reverse '(1 2 3 4 5) -> '(5 4 3 2 1))


;Q2.3
;Signature: duplicate-items(lst, dup-count)
;Type: [List(T) * List(Number) => List(T)]
;Purpose: Duplicate each item in lst number of times defined in the same position at dup-count,
          where dup-count is treated as cyclic list.
;Pre-Conditions: dup-count is a non-empty list of non-negative Numbers.
;Tests: (duplicate-items '(1 2 3) '(1 2) -> '(1 2 2 3))
        (duplicate-items '(1 2 3) '(1 2 3 4 5) -> '(1 2 2 3 3 3))


;Q2.4
;Signature: payment(n, coins-lst)
;Type: [Number*List<Number> => Number]
;Purpose: returns in how many different ways we can choose numbers from coins-list such that the
sum is n.
Pre-Conditions: n in a non-negative Number, coins-lst is non-empty list of positive Numbers
;Tests: (payments 2 '(1 1 1 2 2) -> 2)
        (payments 5 '(5 2 1 2) -> 2)
        (payments 0 '(1 23 12) -> 1)


;Q2.5
;Signature: compose-n(f, n)
;Type: [(T => T) * Number => (T => T)]
;Purpose: return a closure if the n-th self-composition of f.
;Pre-Conditions: n is a positive Number. f is an unitary function that returns as output the same type
of parameter that it gets to the input. In other words, the image of f contains  the domain of f.
;Tests: ((compose-n (lambda (x) (+ 1 x)) 3) 5) -> 8
        ((compose-n (lambda (x) (* 2 x)) 4) 1) -> 16
```