Question 1: Theoretical Questions

Q1.1 Special forms are required in programming languages because they are enrich our programming language in a way that primitive variables cannot, for an example the special form “if”. We cannot simply define them as primitive operators due to that each special expression has a different way of calculating.

Q1.2

- (+ 4 5)
  (- 4 5)
- (define x 10)
  (define y (+ x 10)

Q1.3 There is no such program, the sole purpose of “define” is to give a name for an expression. Therefor we could omit it and only use the actual expression instead.

Q1.4 There is a program in L2 which cannot be transformed to an equivalent program in L20, for example:

```
(define func
  (lambda (x)
    (if (= x 81)
        x
      (func (* x x)))))
```

Q1.5 **map, filter, all** - the order doesn't matter because each procedure happens on a single expression on the list separately and does not affect the others.
reduce - the order does matter because for example of a reducer function which is to concat the given list . if that would happen in parallel the order of the list may change as the order of the thread's work.
compose - the order matters as well because in order to preserve the output of a compound function the program should run in a sequential way.

Q1.6

```
(define p34 (pair 3 4)); constructs a pair:(a=3, b=4) *the global var b was overwritten
((lambda (c) (p34 'f)) 5); c=5 -> 3+4+5=12 *the global var c was overwritten
```

The value of the following program is 12.

Question 2

Q2.1

```
; Signature: append (lst1 lst2)
; Type: [List(T)*List(T)->List(T)]
; Purpose: gets two lists and returns their concatenation.
; Pre-conditions: none
```

```
; Tests: (append '(1 2) '(3 4)) → '(1 2 3 4)
```

Q2.2

```
; Signature: reverse (lst)
; Type: [List(T)->List(T)]
; Purpose: gets a list and reverses it.
; Pre-conditions:none
; Tests: (reverse '(1 2 3)) → '(3 2 1)
```

Q2.3

```
; Signature: duplicate-items (lst dup-count)
; Type: [List(T)*List(T)->List(T)]
; Purpose: gets two lists - lst, dup-count - and duplicates
each item of lst according to the number defined in the same position in
dup-count.
; Pre-conditions: (empty? dup-count)->#f
; Tests: (duplicate-items '(1 2 3) '(1 0)) → '(1 3); (duplicate-items '(1 2 3) '(2 1 0
10 2)) → '(1 1 2)
```

Q2.4

```
; Signature: payment (n coins-lst)
; Type: [number*List(number)->number]
; Purpose: gets a sum of money and list of available coins, and
returns the number of possible ways to pay the money with these coins.
; Pre-conditions: none
; Tests: (payment 10 '(5 5 10)) → 2; (payment 5 '(1 1 1 2 2 5 10) → 3
```

Q2.5

```
; Signature: compose-n (f n)
; Type: [(T->T)*number->(T->T)]
; Purpose: gets a unary function f and a number n (>0) and
returns the closure of the n-th self-composition of f.
; Pre-conditions: n>0
; Tests: (define mul8 (compose-n (lambda (x) (* 2 x)) 3)); (mul8 3) → 24
```