Question 1: Theoretical Questions

Q1.1

A special form is a form whose first expression is a special operator. Special forms are evaluated by the special evaluation rules of their special operators. Primitive operators on the other hand are operators with fixed and defined evaluation functions. Ultimately defining special forms as primitive operators will make it impossible to deviate from the set collection of operators recognized by the parser.

Q1.2

A program where the evaluation can be done in parallel:

```
(define a (+ 1 2))
(define b (+ 2 3))
(+ a b)
```

A program where the evaluation cannot be done in parallel:

```
(define a 1)
(define b (+ a 2))
(+ b 3)
```

Q1.3

Any program in L1 can be transformed into L0, every program is a computational result of many small app-expressions so we can just insert a defined variable as a computational result in the fitting app-expression.

Q1.4

Let's look at the program:

```
(define rec
     (lambda (a b)
          (if (= a 0)
               b
               rec ((- a 1) (+ b 1)))))
```

Without the special form "define" we couldn't have called the function "rec" and preformed the recursion.

Q1.5

a. <u>Map</u> **-** the procedure application on the list items can be applied in parallel, because the function activated on each argument of the list independently of others arguments.
b. <u>Reduce</u> - the procedure application on the list items must be sequential because there are operators (division for example) that are not commutative so there is a importance to the order of the arguments.
   for example $(reduce\ /\ 1\ '(2\ 4\ 8))\ --\!>\ (+\ 2\ (+\ 4\ (+\ 8\ 0))) = 1$ not equal to $(reduce\ /\ 1\ '(2\ 8\ 4))\ --\!>\ (+\ 2\ (+\ 8\ (+\ 4\ 0))) = \frac{1}{4}$
c. <u>Filter</u> - the procedure application on the list items can be applied in parallel, because the predicate is activating on each argument of the list independently of others arguments.
d. <u>All</u> - the procedure application on the list items can be applied in parallel, because the predicate is activating on each argument of the list independently of others arguments, so all will return #t just if all the items will return #t from the predicate. Not matter the order of the check.
e. <u>Compose</u> - the procedure application on the list items must be sequential because the sequence can affect the result.
   For example:
   const add4 = (a: number) => a + 4
   const mult2 = (a: number) => a * 2
   const $comp_1$ = compose (mult2, add4)
   const $comp_2$ = compose (add4, mult2)

$$comp_1(2) = (2+4)*2 = 12 \neq 8 = (2*2)+4 = comp_2(2)$$

<u>Q1.6</u>

The value of the program is: 9.

We first define p34 which is a pair whose values are $a = 3, b = 4$. When defining the pair p34 its 'c' variable is replaced by the primitive number 2, as defined in that scope. Computing the value of a function that receives an argument 'c' and calculates the result of computing 'f on p34 does not change the 'c' value of p34. Ultimately the calculation is as follows: $3 + 4 + 2 = 9$.

Question 2: Contracts

Q2.1

Signature: append (lst1 lst2)

Type: $[list * list -> list]$

Purpose: concatenating two lists into one list when the first part is of lst1 and the rest is of lst2.

Pre-conditions: None

Tests:

(append ‘(1 2) (#t 5))-> ‘(1 2 #t 5)

Q2.2

Signature: reverse (lst)

Type: $[list -> list]$

Purpose: reverse the order of lst arguments

Pre-conditions: None

Tests:

(reverse (‘(1 3 #f 9))- > ‘(9 #f 3 1))

Q2.3

Signature:

duplicate-items (lst dup-count)

Type: $[list * list -> list]$

Purpose: duplicates each item of lst according to the number defined in the same position in dup-count.
Pre-conditions: dup-count contains numbers and is not empty.

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: $[number * list -> number]$

Purpose: calculates the number of possible ways to pay a certain sum of money (n). with a collection of coins (coins-lst).
Pre-conditions: n is non-negative.

Tests:

(payment 10 '(5 5 10) → 2

(payment 5 '(1 1 1 2 2 5 10) → 3

(payment 0 '(1 2 5) → 1

(payment 5 '() → 0

Q2.5

Signature:

Compose-n (f n)

Type: $[function * number -> function]$

Purpose: return the function that is resulted by self-composition of f n times.
Pre-conditions: f is unary function. N is positive.

Tests:

(compose-n (lambda(x) (+ 2 x)) 2 → (lambda(x) (+ 4 x))

(compose-n (lambda(x) (* 2 x)) 2 → (lambda(x) (* 4 x))