PPL Assignment 2

Part 1:

1.1) A special form is an expression that is evaluated in a non-standard way. Special forms are required in programming languages because sometimes we want to change the regular evaluation rules – instead of evaluating each subexpression first, and then apply the result of the computation to the procedure (the first argument), we would like to create a new set of rules. For example – the special form (if (= x 0) (+ x 1) (/ 1 x)).
If we define this form as primitive operator, the subexpression (/ 1 x) would be evaluated before applying "if", thus causing division by zero if the condition is "true".

1.2) Example of program in L1 which can be done in parallel:

```
(+ 5 2)
(+ 10 3)
```

Example of program in L1 which cannot be done in parallel:

```
(define radius 1)
(define pi 3.14)
(define perimeter (* (* 2 radius) pi))
(+ perimeter (* 2 3))
```

*Obviously, the last expression cannot be evaluated before the previous lines because perimeter is defined in those expressions.

1.3) Every program in L1 can be transformed to a program in L0. If a program in L1 does not have a "define" special form, then the program belongs also to L0. Otherwise, we can replace every usage of the "variable" from the program with the expression that the var is bind to (that was evaluated in "define").

1.4) Example of program in L2 which cannot be transformed to a program in L20:

```
(define fib (lambda (n)
        (if (< n 2) n
                (+ (fib (- n 1)) (fib (- n 2))))))
```

This program is recursive and returns the Nth number in the Fibonacci sequence.
In L20, we cannot implement recursive programs which depends on unknown parameter, because we do not know how many lines of code we should write beforehand.

1.5)
map: The procedure application on the list items can be applied in parallel because the map procedure applies the given function on each of the list items independently, meaning evaluating the function on one item does not interfere or related to other evaluations.

reduce: In reduce, the procedure application on the list items cannot be applied in parallel because the function is maintaining an accumulator that iterates on the list items. For example – if the accumulator subtracts each item from its current value, parallel execution can affect the correctness of the program ("2-1" is not equal to "1-2").

filter: In filter, the procedure application on the list items can be applied in parallel because the function returns a list of the items that applying the predicate on them returned "true". Evaluating the predicate on each item is done independently from the other items and does not interfere with other evaluations.

all: In all, the procedure application on the list items can be applied in parallel because the function returns true only if the application of the function on every list item returns true, and like in filter, evaluating each function application on list item can be done in parallel because they are independent.

compose: In compose, the order of execution of the given functions is important, therefore it must be sequential. Compose applies every function and passes the returned value to the next function executed. As a result, the execution of the functions cannot be done in parallel as the correctness of the program will be affected. For example, if F1 is filter that returns the even items in the list, and F2 multiplies every item on the list by 2, executing F1 before F2 may return a different result than the other way around.

1.6) The value of the above program is **9**.
A closure is a procedure that records what environment it was created in. When one calls it, that environment is restored before the actual code is executed. The same applies in our code. 'c' is defined in the closure (c = 2) because this is the value of ‘c’ in the environment where the closure is created. Therefore, when function 'f' is called, p34 is defined as the pair (3 4) and c is equal to 2 – because this is the value of ‘c’ in the closure’s environment, and not the value 5. Finally, f returns the value 9, which is the sum of 3,4,2 - (3 + 4 + 2 = 9).

Part 2:

2.1:

- Signature: append (lst1, lst2)
- Type: [List(any) * List(any) → List(any)]
- Purpose: To append/ thread two lists into one list.
- Pre-conditions: true
- Tests: (append '(1 2) '(3 4)) → '(1 2 3 4)

2.2:

- Signature: reverse (lst)
- Type: [List(any) → List(any)]

- Purpose: To reverse the appearance order of the elements in the list.
- Pre-conditions: true
- Tests: (reverse '(1 2 3)) → '(3 2 1)

2.3:

- Signature: duplicate-items (lst, dup-count)
- Type: [List(any) * List(Numbers) → List(any)]
- Purpose: To duplicates each item of lst according to the number defined in the same position in dup-count. In case dups-count length is smaller than lst, dup-count is treated as a cyclic list.
- 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)

2.4:

- Signature: payment (n, coins-lst)
- Type: [Number * List(Numbers) → Number]
- Purpose: To calculate the number of possible ways to pay 'n' money with the coins available in 'coins-lst'.
- Pre-conditions: true
- Tests: (payment 10 ‘(5 5 10)) → 2
  (payment 5 ‘(1 1 1 2 2 5 10) → 3

2.5:

- Signature: compose-n (f, n)
- Type: [(T)→T * Number → Number]
- Purpose: To compute 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