**Question 1: Theoretical Questions**

Q 1.1:

Why are special forms required in programming languages?

Special form has it’s own evaluation rule, so we need it in part of expressions used (if for example).

Why can’t we simply define them as primitive operators?

Primitive operators evaluate expressions in from leftmost to right, but it cannot fit to every special form expression, need to be explained by own rule.

Example for special form:

**Define-**(define size 6) **If-**(if (> x 2) x (\* x 2))

Q 1.2:

**Can be done in parallel:**

(define x 1)

(define y 1)

**Cannot be done in parallel:**

(define x 1)

(define y x)

Q 1.3:

Without ‘define ‘ we cannot bind values with variables in global environment, so every program in L1 that uses ‘define’ cannot be transformed to L0 language. Example from a lecture.

(**define** size 6)

(\* 2 size) *;--> 12 ;;*

Q 1.4:

The example for a program which cannot be transformed to L20 can be a factorial from the practical session, without bind the recursion isn’t possible because function called by name but cannot be binded:

1. (define fact
2. (lambda (n)
3. (if    (= n 0)
4. 1
5. (\* n (fact (- n 1))))))

Q 1.5:

**Map:** every item on the list is independent on map so the application can be done in parallel.

**Reduce:** application must be sequential , for example we have a list of 1,2,15 and want to calculate the quotient (divide) of everything on accumulator (1). Sequential application will return us 1/30, but any other evaluation (e.g the reverse order 15->2->1=7.5) cannot promise us desired result. Must be sequential.

**Filter:** like a map, filter is applying the predicate on each member of the list, independent from other list members so can be done in **parallel**.

**All:** Final result depends on all the members of the list so can be **parallel**.

**Compose:** Like in mathematics, we know that not every f ,g functions f(g(x)))=g(f(x)). For example, f(x)=x+2, g(x)=x\*5 , so for x= 5 f(g(x))=35, g(f(x))=27, so application can be only **sequential**.

Q 1.6:

Although we bind b and c with values at the start of the code, that’s not the same b we give to pair p34. So, the evaluation of **(lambda (c) (p34 ‘f)),** we get the next expression: 3+4+2 and final value of the code is **9.**

**Question 2: Pprogramming in L3 (Design by contract part)**

**Q 2.1:**

**; Signature: append (l1,l2)**

**;Type: [List(T1)\*List(T2)-> List (T3)]**

**;Purpose: To concatenate l1 and l2 lists**

**;Pre-conditions: true**

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

**Q 2.2:**

**; Signature: reverse (l1)**

**;Type: [List(T1)->List(T1)]**

**;Purpose: Gets a list and reverses it.**

**;Pre-conditions: true**

**;Tests: (reverse '(1 2 3)) → '(3 2 1)**

**Q 2.3:**

**; Signature: duplicate-items(lst , dup-count)**

**;Type: [List<any>\*List<number> -> List <any>]**

**;Purpose: 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 should be treated as a cyclic list.**

**;Pre-conditions: dup-count contains numbers and non-empty**

**;Tests: (duplicate-items '(1 2 3) '(1 0))→ '(1 3)**

**(duplicate-items '(1 2 3) '(2 1 0 10 2))→ '(1 1 2)**

**Q 2.4:**

**; Signature: payment ( sum, l1)**

**;Type: [number\*List<number>->number]**

**;Purpose: count and return the number of possible ways to pay the money with these coins.**

**;Pre-conditions: sum is not negative, list is not empty**

**;Tests: (payment 10 ‘(5 5 10)) → 2**

**(payment 5 ‘(1 1 1 2 2 5 10) → 3**

**Q 2.5:**

**; Signature: compose-n ( f, n)**

**;Type: [ [T1->T2] \* number -> [T1->T2] ]**

**;Purpose: returns the closure of the n-th self-composition of f**

**;Pre-conditions: f is unary function, n is positive**

**;Tests: (define mul8 (compose-n (lambda (x) (\* 2 x)) 3))**

**(mul8 3) → 24**