Question 1:

Q1.1 Give an example for each of the following categories in L3:

- Primitive atomic expression : #f
- Non-primitive atomic expression : varible1, (as a variable name)
- Non-primitive compound expression : (+ 1 1)
- Primitive atomic value : the numerical value of 1
- Non-primitive atomic value : list - '()
- Non-primitive compound value : Closure – value of a function in L3.

Q1.2 What is a special form?

Special form is an expression that is evaluated in a special way – not like procedure application.

Example – (define gravity_constant 9.81)

Q1.3 What is a free variable?

Free variable is a term to describe the way a variable acquires in an expression,

A variable x occurs free in an expression E if and only if there is some use of x in E that is not bound by any declaration of x in E.

Example: (lambda (x) (+ x y)) , y is a free variable, but x is bounded so he is not free.

Q1.4 What is Symbolic-Expression (s-exp)?

Symbolic expression is a tree – hierarchical structure of values used in order to for the parser in a convenient way – a tree of program tokens is in input to the parser for example : ['+', ['+', '1','2'], ['-', '4', '3']]

Q1.5 5 What is 'syntactic abbreviation'?

Syntactic abbreviation means that when we define the operational semantic of the language, we do not need to define a new computation rule for this expression type, instead we indicate that this expression is equivalent to a combination of other syntactic constructs that mean the same thing.

Let is a syntactic abbreviation of lambda, for example:

(let ((x 8) (y 5)) (+ x y)) -is- ((lambda (x y) (+ x y)) 8 5)

Cond is a syntactic abbreviation of if, example:

(cond (((> x 1) 2) ((> y 3) 4) (else 5)) -is- (if (> x 1) 2 (if (< y 3) 4 5))

Q1.6 Let us define the L30 language as L3 excluding the list primitive operation and the literal expression for lists with items (there is still a literal expression for the empty list '()). Is there a program in L3 which cannot be transformed to an equivalent program in L30? Explain or give a contradictory example.

Every program in L3 can be transformed into an equivalent program in L30 we can use lists as syntactic abbreviation, change every list occurrence to cons

(a, b, c, d) => (a .(b .(c .(d . '())

Q1.7 In practical session 5, we dealt with two representations of primitive operations: PrimOp and Closure. List an advantage for each of the two methods.

- Closure advantage – easier to add new primitive operations, because then there is no need to change the interpreter.

- primOp advantage – there is no need for environment lookups making it faster.

Q1.8 In class, we implemented map in L3, where the given procedure is applied on the first item of the given list, then on the second item, and so on. Would another implementation which applies the procedure in the opposite order (from the last item to the first one), while keeping the original order of the items in the returned list, be equivalent? Would this be the case also for: reduce, filter, compos.

Map function doesn't change the array and only makes a new one, so changing the order of the applied function doesn't changes the result unless there are side effects...

In the reduce function the order matters and the result will not be equivalent for example: (reduce / 1 '(1 2 3))

Will return 1/6 with one order and return 3/2 for the opposite.

In case the function is comitative (order doesn't matter, and there are no side effects it will be equivalent.

filter function if it will keep the same order of the element the result will be the same, as we only take the desired elements and recollect them, there should be no side effects.

compose function this is same as reduce even clearer, the order matters and the functions are not equivalent,

for example (2 + 3) / 4 is different from 2 / (3 + 4).

In all the above cases if there are side effects the order can matter and the programs are not equivalent, a simple example will be printing to the console.

Question 2:

Q2.1

```
; Signature: (last-element list)
; Type: [T[] -> T]
; Purpose: return the last element of list
; Pre-conditions: list is not empty
; Tests: (last-element (list 1 3 4)) → 4
```

Q2.2

```
; Signature: (power n m)
; Type: [number * number -> number]
; Purpose: return n^m
; Pre-conditions: m positive number
; Tests: (power 2 4) → 16
```

Q2.3

```
; Signature: (sum-lst-power list n)
; Type: [ number[] * number -> number]
; Purpose: return the sum of all the elements of list in the power of n
; Pre-conditions: n positive
```

; Tests: (sum-lst-power (list 1 4 2) 3) → 1^3+ 4^3 + 2^3 = 73

Q2.4

; Signature: (num-from-digits list)

; Type: [number[] -> number]

; Purpose: returns the number consisted from digits in list

; Pre-conditions: list of positive integers

; Tests: (num-from-digits (list 2 4 6)) → 246

Q2.5

; Signature: (is-narcissistic list)

; Type: [number[] -> boolean]

; Purpose: return if the list of digits that represents a number is narcissistic

; Pre-conditions: list of positive integers

; Tests: (is-narcissistic (list 1 5 3)) → #t