<u>Question 1: Theoretical Questions</u>

1. Give an example of the following:
   a. <u>Primitive atomic expression:</u> 5
   b. <u>Non-Primitive atomic expression:</u> x
   c. <u>Non-primitive compound expression:</u> (+ 1 2)
   d. <u>Primitive atomic value:</u> value returned from 3
   e. <u>Non-Primitive atomic value:</u> the value of the empty list ‘()
   f. <u>Non-Primitive compound value:</u> list (0, 0, 0)
2. A special form is an expression which is not computed by the interpreter as a regular expression.
   For example: <define x 3>. The interpreter knows define is a special form which assigns 3 to x.
3. Free variable is a variable which wasn’t previously declared in a given expression.
   ((lambda (x) x) y)   => y is a free variable, while x is bounded because it was declared in the function argument.
4. S-Expressions is a tree of values which have a certain hierarchy.
   Example: ‘(+ 1 2)
5. Syntactic Abbreviation is a legal expression in the language which stands for another expression with the same meaning, while making it simpler for the programmer to understand and handle. examples:
   - ((lambda (x y) (+ x y)) 1 2)
     after Syntactic Abbreviation:
     (let ((x 1) (y 2)) (+ x y))
   - (if (< x y) (+ x y) (- x y))
     after Syntactic Abbreviation:
     (cond ((< x y) (+ x y)) (else (- x y)).

6. Any program in L3 can be transformed into L30.
   To represent list without list operation and the expression for list with items, we can use the keyword “const” for a pair constructor in a recursive manner.
   for example: the list ‘(1 2 3) in L3 can be presented as (cons 1 (cons 2 (cons 3 '()))) in L30.

7. PrimOp: PrimOp expression is the same as its value, so we don’t have to look for the PrimOp definition in the Global environment.
   Closure: Interpenter remains the same whenever we define new operations.

8. The different implementation of Map will yield equivalent results, because the Map function applies on each of its arguments separately - So the End values will be the same in both implementations.

- Reduce: implementation won’t be equivalent, because the order of the computation may have different effect, for example:

```
[7, 3].reduce((acc, cur) => acc - cur, cur) // 7 - 7 -3 = -3
[3, 7].reduce((acc, cur) => acc - cur, cur) // 3 - 3 -7 = -7
```

- Filter: similar to Map, filter operation effects on each element individually, therefore the implementation will be equivalent.
- Compose: implementation won’t always be equivalent:

```
compose(filter(isEven), map(plus3))([0,1,2,3]) // => [3, 5]
compose(map(plus3), filter(isEven))([0,1,2,3]) // => [4, 6]
```

Question 2 Contracts

; Signature: last-element(lst)
; Type: [list(T) -> T]
; Purpose: return the last element of the list.
; Pre-conditions: list is valid not null
; Post-conditions: return the last element of type T
; Tests: (list 1 3 4)) → 4

; Signature: power(n1 n2)
; Type: [Number*Number -> Number]
; Purpose: return n1 power n2
; Pre-conditions: (n1&n2) is valid n1&n2 >= 0
; Post-conditions: return n1 power n2 >= 0
; Tests: (power 5 3) → 125

; Signature: sum-lst-power(lst n)
; Type: [list(Number)*Number -> Number]
; Purpose: return return the sum of each element in the list power n
; Pre-conditions: (lst&n) is valid lst not null & n>=0
; Post-conditions:
; Tests: (list 1 4 2) 3) → 1^3+ 4^3 + 2^3 = 73

; Signature: num-from-digits(lst)
; Type: [list(Number) -> Number]
; Purpose: return the sum of each element in the list power n
; Pre-conditions: lst not null
; Post-conditions: return a valid number
; Tests: (num-from-digits '(3 1 4)) ==> 314

; Signature: is-narcissistic(lst)
; Type: [list -> Boolean]
; Purpose: checks if a list of numbers is a narcissistic list
; Pre-conditions: lst is valid not null
; Post-conditions: return boolean
; Tests: (is-narcissistic '(1 5 3)) ==> #t