QUESTION 1:

Q1.1:

Primitive atomic expression: + , 1 , #f
Non-primitive atomic expression: y
Non-primitive compound expression: (if #f 3 4), (* 6 7)
Primitive atomic value: numerical value of 5
Non-primitive atomic value: symbol 'green
Non-primitive compound expression: '(1 3 4)

Q1.2:
An expression that the calculation of its value is not doing by the regular way, like: (if (> 3 4) 8 9)

Q1.3:
A variable x occurs free in an expression E ⟺ there is a VarRef to x at E and there isn't a VaerDecl of x at E. For example:
(lambda (x) (* y x)) , y occurs free

Q1.4:
SExp is a concept (structure) that allows us to represent value hierarchy: a tree of values. It's the literal expression of our languages, that with it we can define every combination of values. SExp can be a tree of program tokens for example: ['if', ['>', '3', '4'], '8', '9'].

Q1.5:
Syntatic abbreviation is an expression that could be defined with the existing primitives of the language, but was created for a syntactic purposes:
let is a syntactic abbreviation of lambda:

```
(let
   ((x 6) (y 5))
   (+ x y))
⟺
((lambda (x y)
   (+ x y))
      (6 5))
```

cond is a syntactic abbreviation of if:

```
(cond ((#f 7)
       ((> 7 8) 5)
       (else 9))
⟺
 (if #f 7
   (if (> 7 8) 5
      9))
```

Q1.6:
No, every program in L3 can be transform to an equivalent program in L30 by define a list with items inductively as pair of item and list, when the deepest pair is a pair of the last item and the empty list.
For example: '(1 2 3) will be transformed to (cons 1 (cons 2 (cons 3 '()))). Of course the other literal expressions will stay the same.

Q1.7:
PrimOp: We don't need to access the env every time we "see" a PrimOp
Closure: We don't need to update the interpreter every time we add a new primitive operation.

Q1.8:
map: In functional programing there are no side effects and the application of the procedure on each item of the list is independent, so applying the procedure in the opposite order will be equivalent.
reduce: Returns the accumulator value that depends on the previous items, so applying the procedure in the opposite order will not be equivalent necessarily:
(reduce \ 2 '(4 8)) will return 1 from one way and 4 from the other way.
filter: Like in the "map" case, we are in functional programing and the satisfaction of the predicate by one element does not depend on the other elements, so applying the procedure in the opposite order will be equivalent.
compose: composition of a list of functions from a different will not necessarily return the same output:
(compose (lambda(x)(* x x) lambda(x)(+ x x))) with argument 3 will return 18 from one way and 36 from the other way.

QUESTION 2 CONTRACTS:

```
;Q2.1

;Signature: last-element(lst)

;Type: [List<any>] -> [any]

;Purpose: Return the last element of lst

;Pre-conditions: lst is a non-empty List

;Tests: (last-element(list 1 2 3)) -> 3
```

;Q2.2

;Signature: power(n1, n2)

;Type: [number, number] -> [number]

;Purpose: Return n1 to the power of n2 (n1^n2)

;Pre-conditions: n1 is a non-negative number and n2 is a natural number (include 0)

;Tests: (power(5 3)) -> 125

;Q2.3

;Signature: sum-lst-power(lst, n)

;Type: [List<number>, number] -> [number]

;Purpose: Return the sum of all the element of lst in the power of n

;Pre-conditions: lst is a non-negative numbers List and n is a natural number (include 0)

;Tests: (sum-lst-power((list 1 4 2) 3) -> 73

;Q2.4

;Signature: num-from-digits(lst)

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

;Purpose: Return the number consisted from the digits of lst

;Pre-conditions: the numbers of lst are in the range of 0-9

;Tests: (num-from-digits(list 1 4 2) -> 142

;Q2.5

;Signature: is-narcissistic(lst)

;Type: [List<number>] -> [boolean]

;Purpose: Return if the number consisted from the digits of lst is narcissistic (or not), a
;number that is the sum of its own digits each raised to the power of the number of the
;digits

;Pre-conditions: the numbers of lst are in the range of 0-9

;Tests: (is-narcissistic (list 1 5 3) -> #t; (is-narcissistic (list 1 2 3) -> #f;