Theoretical Questions:

1.1) Examples of the following:

● Primitive atomic expression – Number:5

● Non-primitive atomic expression

Length – it is non primitive because we need to define this on our language.

● Non-primitive compound expression – define expression of square

(define square

(lambda (x)

(* x x)))

● Primitive atomic value

The numerical value 3

● Non-primitive atomic value

The enum 'RED'

● Non-primitive compound value

(define x length) – length becomes a non-primitive compound value of x, length is defined above.

1.2) Special form is a Parenthesis that on the left side there is a special operator like define, each one of them has his own way to evaluate the value of the expression.

For example: (define x (+ 1 2)), Eval (+ 1 2) and then add binding <x,3> to the environment.

1.3) Free variable sum in an expression Exp is to use sum in Exp and sum has no declaration in the bounds of exp.

Example: (lambda (x) (+ x y)) – y occurs free: he is in the expression, but he was never declared on the expression scope.

1.4) S-exp is a Hierarchical structure of a tokens array by the Syntactic rules.

S-exp help in the parsing process.

structure of Sexp is disjunction between Atomic tokens and Compound expressions.

Example: the Boolean true is an atomicSexp Boolean.

1.5) Syntactic abbreviation is the ability to define syntactic transformations from one expression to a second semantically equivalent expression.

That syntactic transformation leading to a simpler equivalent construct and is implemented with rewriting AST's.

Example 1: lambda can replace let expression (syntactic abbreviation).

(let ((x 2) (y 2)) (* x y)) is the same as: ((lambda (x y) (* x y)) 2 2).

Example 2: if can replace cond expression (syntactic abbreviation).

(cond ((< 4 5) “first") ((< 5 5) "second") ((< 6 5) “third") (else “otherwise”))

Is the same as:

(if (< 4 5) “first” (if (< 5 5) “second” (if (< 6 5) “third” “otherwise)))

1.6) No, all lists can also be transformed to constitutive pairs, where each pair composed from element and pair which can also be composed in the same way.

the last pair could be consisting of the last element and the empty list, which makes the whole expression as a list.

Therefore, all programs in L3 can be transformed to a program in L30.

1.7) PrimOp: implement primitive operators as syntactic expression help us to take some of the load off the parser.

Closure: define primitive operators as closures in the environment make it easier to add more primitive operators in the future, as well as removing some of the interpreter’s work.

1.8) If the given procedure has no side effects, the order doesn't matter and map and filter will behave the same because they act on each item of the collection separately.

On the case of reduce and compose it will behave the same only if the given procedure is associative.

Contracts for question 2

last-element:

/*

Purpose: return the last element in the list.

Signature: last-element (lst)

Type: [List(T1) -> T1]

Pre-conditions: the list is not empty.

Tests: (last-element (list 1 3 4)) → 4

*/

Power:

*/

Purpose: return n1 to the power of n2 (n1^n2).

Signature: power (n1, n2)

Type: [Number -> Number]

Pre-conditions: n1 and n2 are not negative.

Tests: (power 2 4) → 16, (power 5 3) → 125

/*

Sum-lst-power:

*/

Purpose: returns the sum of all the list elements in the power of N.

Signature: sum-lst-element(lst, n)

Type: [List (Number) -> Number]

Pre-conditions: the list elements are not negative numbers.

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

/*

Num-from-digits:

*/

Purpose: returns the number consisted from the list elements.

Signature: num-from-digits(lst)

Type: [List (Number) -> Number]

Pre-conditions: it is a list of not negative numbers.

Tests: (num-from-digits (list 2 4 6)) → 246, (num-from-digits (list 5 7)) → 57

/*

Is-narcissistic:

*/

Purpose: return if the number represents by the list elements equal to the sum of its own elements, each raised to the power of the number of the elements.

Signature: is-narcissistic(lst)

Type: [List (Number) -> Boolean]

Pre-conditions: it is a list of not negative numbers.

Tests: 153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153

/*