PPL - Assignment 2

Part 1: Theoretical Questions

Q1.1

Primitive atomic expression: 2 (could be an expression as seen in ps3 when typed as a terminal's command)

Non-primitive atomic expression: height (non-primitive since we need to define it)

Non-primitive compound expression:

```
(define sum
  (lambda (x y)
    (+ x y)))
```

Primitive atomic value: (define height 2) – here the number 2 becomes primitive atomic value of height

Non-primitive atomic value: (define x height) – here height becomes a non-primitive atomic value of x, assuming height already defined

Non-primitive compound value: (define y sum) – here sum becomes a non-primitive compound value of y, assuming sum is defined (as above)

Q1.2

While in non-special form expressions evaluated by a general evaluation rule, where all the sub expressions are always evaluated, in a **Special form** expression a special evaluation rule exist. Therefore, not all the parts of the compound form are always evaluated and the order in which the parts is evaluated is determined by the computation rule of the compound form type.

Example: in 'if' special form, one sub-expression never evaluated

Q1.3

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) – here y occurs free since it meets both conditions: it appears in the expression, and it is not bound to any y declaration in that expression

Q1.4

Symbolic-Expression (s-exp) is a data notation, represented by significant language tokens wrapped with constituent brackets. An S-exp is defined inductively and its simplicity and uniformity help in the parsing process. The structure of the Sexp type is the usual disjunction between Atomic tokens and Compound expressions, when Compound expressions are encoded as arrays of embedded expressions.

Example: the number 2 is an atomicSexp number

Q1.5

Syntactic abbreviation is the ability to define syntactic transformations from one expression to a second semantically equivalent expression. This possible when an expression is equivalent to a combination of other syntactic constructs that mean the same thing. That syntactic transformation leading to a simpler equivalent construct and is implemented with rewriting ASTs.

Example 1: 'let' expression is a syntactic abbreviation that can be replaced with a lambda.

```
(let ((x 1) (y 2))
   (+ x y))
```

is equal to:

```
 ((lambda (x y) (+ x y))
   1 2)
```

Example 2: 'cond' expression is also a syntactic abbreviation. Which can be replaced with an 'if'.

```
(cond ((> 1 2) "first" ) ((> 2 2) "second") ((> 3 2) "third" ) (else "otherwise"))
```

can be replaced with:

```
(if (> 1 2) "first" (if (> 2 2) "second" (if (> 3 2) "third" "otherwise")))
```

Q1.6

No, all lists can also be represented by constitutive pairs, where each pair composed from one element and another pair which can also be composed in the same structure, and so on, until the last pair that could be consist 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 an equivalent program in L30.

Q1.7

PrimOp advantage:
By implementing primitive operators as syntactic expressions we take some of the load off the parser.

Closure advantage:
By defining primitive operators as closures in the global environment, we are making it easier to add more primitive operators in the future, as well as removing some of the interpreter's work.

Q1.8

Map and Filter be equivalent if there is no side-effect since they work separately and independently on each element. Reduce and Compose are equivalent only if the given procedure is commutative.

Q2

```
; Signature: empty? (lst)
; Type: [T1 -> Boolean]
; Purpose: To determine whether a given expression is the empty list
; Pre-conditions: none
; Tests: (empty? '()) ==> #t
```

; Signature: length (lst)
; Type: [List(T1) -> Number]
; Purpose: Return the length of a given list
; Pre-conditions: none
; Tests: (length (list 5 4 2)) ==> 3

Q2.1

; Signature: last-element (lst)
; Type: [List(T1) -> T1]
; Purpose: Return the last element of a given list
; Pre-conditions: List is not Empty
; Tests: (last-element (list 5 4 3 7 8)) ==> 8

Q2.2

; Signature: power (n1 n2)
; Type: [Number * Number -> Number]
; Purpose: Given two numbers n1 and n2, return n1 to the power of n2 (n1^n2)
; Pre-conditions: Given numbers are non-negative
; Tests: (power 2 4) ==> 16

Q2.3

; Signature: sum-lst-power (lst n)
; Type: [List(Number) * Number -> Number]
; Purpose: Given a list and a number N, returns the sum of all its elements in the power of N
; Pre-conditions: none
; Tests: (sum-lst-power (list 1 4 2) 3) ==> 73

Q2.4

; Signature: num-from-digits (lst)
; Type: [List(Number) ->Number]
; Purpose: Given a list of digits, returns the number consisted from these digits
; Pre-conditions: none
; Tests: (num-from-digits (list 2 4 6 5)) ==> 2465

Q2.5

; Signature: is-narcissistic (lst)
; Type: [List(Number) -> Boolean]
; Purpose: Given a list of digits, return if the number consisted from these digits is narcissistic or not
; Pre-conditions: none
; Tests: (is-narcissistic (list 1 5 3)) ==> #t