**דור כרמי: 205789662**

**עודד בסרבה: 315564641**

# <u>עבודה 2 ב-PPL:</u>

**<u>Q1.1:</u>**

| | |
|---|---|
| ● Primitive atomic expression | 3 |
| ● Non-primitive atomic expression | x |
| ● Non-primitive compound expression | (- 7 2) |
| ● Primitive atomic value | The numeric value of 10 |
| ● Non-primitive atomic value | The definition of the symbol Sunday |
| ● Non-primitive compound value | (pair 1 2) |

**<u>Q1.2:</u>** What is a special form?

A special form is an expression that is evaluated in a non-standard way, unlike the primitive forms. For example:

((Lambda (x) (* x x)) 5)

**<u>Q1.3:</u>** what is a free variable?

x is a free variable (occurs free) in the expression E if and only if:

- There is a reference (VarRef) to x in E
- x is not declared (VarDecl) in E

For example:

```
((lambda (x) x) y)
```

- `x` occurs bound - since the 2nd occurrence of x in the body of the lambda is bound by the first occurrence in the formals of the lambda.
- `y` occurs free.

**<u>Q1.4:</u>** What is Symbolic-Expression (s-exp)?

A literal expression of our language allows to define any combination of the values.

Datum (also called s-exp for Symbol-Expression) corresponds to the external notation for values that can be read by the "read" primitive procedure and manipulated as a value.

It can be modeled as a tree structure of values.

An example:

```
(if (= x 3)
        4
        5)
```

**Q1.5:** what is a '`syntactic abbreviation`'?

It is a shorthand for expression to make the writing more comfortable, fluently and easier.

Examples:

- The let expressing is a `syntactic abbreviation` and can be defined like this:

```
(let
  ( (a 3) (b 4) )
    (+ a b)
)
→
(
 (lambda (a b)
   (+ a b)
 )
 3 4
)
```

- Another example: the cond expression can be defined as if expression:

```
(cond (((> x 3) 4)
((> y 8) 5)
(else 6))
⇒
(if (> x 3)
4
(if (< y 8)
5
6))
```

**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?

- ⇨ There is no program in L3 that cannot be transformed to an equivalent program in L30. A list in L3 can structured by pairs of elements. Each element pair in the list will contain a value from the list as the first element in the pair, and a nested pair as the second element, which will contain the rest of the list.

- ⇨ The last element in the list (in L30) will be represented as a pair of the last element value and an empty list (which as defined is still in L30)
- ⇨ Each go threw on the elements of the list which check on any step if it reached the end of the list will be equivalent to taking the car of the pair and recursively running the function or the operation we want on the cdr of the pair.
- ⇨ For example, the list: [1 2 3] will turn into: (cons 1 (cons 2 (cons 3 ‘())))

**Q1.7:** PrimOp vs Closure:

PrimOp: no need to check for the variables and the condition of the environment to use the applicant.

Closure: no need for interpreter modification for new defined primitive operations- it can just be declared and the use will be simple- more compatible for changes and adaptation.

**Q1.8:**
On map procedure of the regular version **will** be equivalent to the version of running on an opposite order:

- The operation on each element is independent and not effected by the operation on the rest of the elements
- The map procedure is functional with no side effects
- ⇨ That is why procedure of *map* in regular order will be equivalent to the opposite order: the outcome and the run stages will be the same, and:
  - If one will fail the other will fail too
  - If one will run forever the other will too

About the other procedures:

- Filter: same as map, the outcome of checking the predicate on each element is independent, and there is no side effect- so as the map procedure, it will be equivalent
- Reduce: it won't be the same because the ordered matters and effects the outcome differently. An example: (reduce \ ( 1 '(1 2 3)) would return 1/6 for one order, and 3/2 for the opposite order.

**Question 2:**

```
; Signature: empty? (lst)
; Type: [Any -> Boolean]
; Purpose: checks if a list is empty
; Pre-conditions: true
; Tests: (empty? '()) => true, (empty? '(1)) => false

(define empty?
  (lambda (lst)
    (eq? lst '())))
```

```
;Q2.1:
; Signature: last-element(lst)
; Type: [list of Any -> Any]
; Purpose: returns the last element of a list
; Pre-conditions: lst is list
; Tests: (last-element? '(1 2 3)) => 3, (last-element? '(a b c)) => c


(define last-element
  (lambda (lst)
    (if (empty?(cdr lst))
      (car lst)
      (last-element(cdr lst)))
  )
)

;Q2.2:
; Signature: power(n1 n2)
; Type: [Number X Number -> Number]
; Purpose: returns the result of n1 ^ n2
; Pre-conditions: n1 and n2 are numbers
; Tests: (power '(1 3)) => 1, (last-element? '(2 4)) => 16
(define power
   (lambda (n1 n2)
     (if (= n2 0)
      1
      (* n1 (power n1 (- n2 1))))
   )
)
;Q2.3:
; Signature:  sum-lst-power(lst n1)
; Type: [list X Number -> Number]
; Purpose: returns the sum of the power of every element in the list with n1
; Pre-conditions:lst is a list of numbers and n1 is a number
; Tests: (sum-lst-power '(1 3) 2) => 10, (last-element? '(2 4) 1) => 6
(define sum-lst-power
   (lambda (lst n)
     (if (empty? lst)
          0
          (+ (power (car lst) n) (sum-lst-power (cdr lst) n)))
   )
)

 ;Q2.4.1:
; Signature: num-from-digits(lst)
; Type: [list -> Number]
; Purpose: "builds" a number from seperated digits, in decimal base
; Pre-conditions:lst is a list of numbers
; Tests: (num-from-digits '(1 3)) => 13, (last-element? '(2 4) ) => 24
(define num-from-digits
  (lambda (lst)
     (num-from-digits2 lst 0)
  )
)
```

**;Q2.5.1:**

```
; Signature:  is-narcissistic(lst)
; Type: [list -> Boolean]
; Purpose:check if the number built from the list is a nurcissistic number
; Pre-conditions:lst is a list of numbers
; Tests: ( is-narcissistic '(1 5 3)) => #t,(  is-narcissistic '(1 2 3)) =>#f
(define is-narcissistic
  (lambda (lst)
      (if (= (sum-lst-power lst (findLength lst)) (num-from-digits lst))
       #t
       #f)
   )
)
```

**;Q2.4.2:**

```
; Signature:  num-from-digits2(lst ,n1)
; Type: [listXnumber-> Number]
; Purpose: "builds" a number from seperated digits of the list, in decimal base and
store them in acc
; Pre-conditions:lst is a list of numbers and acc is 0
; Tests:( num-from-digits2( '(1 5 3) 0)) => 153,( num-from-digits2( '(1 2 3) 0))  =>123
(define num-from-digits2
(lambda (lst acc)
  (if (empty? lst)
        acc
        (num-from-digits2 (cdr lst) (+ (* 10 acc) (car lst))))))
```

**;Q2.5.2:**

```
; Signature:   findLength(lst)
; Type: [list-> Number]
; Purpose: count the length of the list
; Pre-conditions:true
; Tests:( findLength '(1 5 3) ) =>3,( findLength '(1 2 ))  =>2
(define findLength
  (lambda (lst)
   (if (empty? lst)
      0
      (+ 1 (findLength (cdr lst))))))
```