Assignment 2

Submittors: Moran Neptune 208474544 and Bar Perlman 305026882

Question 1: Theoretical Questions:

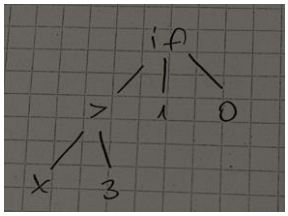
1.1.
* Primitive atomic expression: =
* Non-primitive atomic expression: x
* Non primitive compound expression: (+ x y)
* Primitive atomic value: 3
* Non primitive atomic value: <closure(x) (x*x)>
* Non primitive compound value: p3

clarification about the definition of p3:
define p1 ‘(1 . 5)
define p2 (cons 2 4)
define p3 (cons p1 p2)

1.2.
A special form in some languagh is an expression which has its own calculation rule which is differ from the calculation rule that fitts the general calculation rule of operators in the languagh.
For example:
The special form **define** in L1 and **lambda** in L2.

1.3.
Free variable is a variable which gets its associated value from the global environment and not from the context of the scope.
Example:
```
(let
    (b (* 3 x))
)
```
here **x** is a free variable.

1.4.
s-exp is a tree structure.
In this tree, each node constitutes an expression which is not characterized yet.
Example:
for the expression:
```
(if (> x 3)
    1
    0)
```
The suit s-exp is:



In addition, it can be represented as the list:
['if',['>','x','3'],'1','0']

1.5
The syntactic abbreviation makes our code more readable without extending the languagh abilities.
Example1:
let structure has the same internal form as of lambda:

```
(let
    (
    (b (* 3 x))
    (c (+ 4 x))
    )
    (
      (- b c)
    )
)
```

Example2:
list has the same internal form as of cons but more readable:
we can write '(1 2 3 4) instead of (cons 1 (cons 2 (cons 3 (cons 4 '()))))

1.6.
No.
By definition,define is an operator that gives a name to some expression or value.
(the described operation above is called binding)
so any program written in L1 can be transformed to an equivalent program in L0, although it might be less readable and/longer, because each defuined variable in L0 will be replaced by entire expression or a value with no significant name.
Note: The above based on the fact there is no expression in L0 which is built on the define expression of L1.

1.7.
Yes.
We can't implement any program written in L2 which is implemented in a manner of recursive calls or use the names of functions defined in L2 by define operator in L20 without using the define operator.

1.8.

Advantage of the closure method:

By adding the proper binding in the initialization of the global environment, its easy to add primitives to the interpreter when interpreter starts up so the programmer don't need to define them.

Advantage of the primOp as a symbol representation:

The interpreter recognizes it as the value of itself so we don't get compilation error when the primitive is typed alone.

1.9.

In case of implementation of a **map**, map applies a unary function to each element in the sequence without any dependency of the other elements in the sequence so implementation of a map in which the order applying the procedure is from last element to the first won't make any difference and the mentioned implementation **are equivalent.**

In case of **reduce** , the influence of the order in which the elements are accumulated depends on the selected operator.

For associative operators like add and max it makes no difference, but other operators like division it can, so here the implementation are **not equivalent.**

2.

```
/*
Signature: empty?(lst)
Purpose: To check if the given expression is the empty list
Type: [symbol  -> boolean]
Example: (empty? '()) should produce #t
Pre-conditions: none
Post-condition: result = #t if empty list, else #f
Tests: (empty? '()) ⇒ #t
       (empty? '(1 2 3)) ⇒ #f
*/
(define empty?
  (lambda (lst)
    (if (or (eq? '() lst)
            (eq? (list) lst))
      #t
      #f
    )
  )
)

/*
Signature: list?(lst)
Purpose: To check if the given expression is a list
Type: [symbol  -> boolean]
Example: (list? '()) should produce #t
Pre-conditions: none
Post-condition: result = #t if the expression is a list, else #f
```

```
Tests: (list? '()) ⇒ #t
       (list? (cons 1 2) ⇒ #f
*/
(define list?
  (lambda (lst)
    (if (pair? lst)
      (list? (cdr lst))
      (empty? lst)
    )
  )
)

/*
Signature: equal-list?(lst1,lst2)
Purpose: To check if the given two expressions are equal lists
Type: [list(symbol)*list(symbol) -> boolean]
Example: (equal-list? '(1 2) '(1 2)) should produce #t
Pre-conditions: list? ls1 & list? ls2
Post-condition: result = #t if the given two expressions are equal lists, else #f
Tests: (equal-list? '(1 2) '(1 2)) ⇒ #t
       (equal-list? '(1 2) '(1 3)) ⇒ #f
*/
(define equal-list?
  (lambda (lst1 lst2)
    (if (not (and (list? lst1)
                  (list? lst2)))
      #f
      (if (not (or (empty? lst1) (empty? lst2)))
        (if (or (eq? (car lst1) (car lst2))
                (equal-list? (car lst1) (car lst2)))
          (equal-list? (cdr lst1) (cdr lst2))
          #f
        )
        (if (or (and (empty? lst1)
                     (not (empty? lst2)))
                (and (empty? lst2)
                     (not (empty? lst1))))
          #f
          #t
        )
      )
    )
  )
)

/*
Signature: append(lst1,lst2)
Purpose: To take two lists and to return the first list appended with
the second
```

```
Type: [list(symbol)*list(symbol) -> list(symbol)]
Example: (append '(1 2 3) '(4 5 6)) should produce '(1 2 3 4 5 6)
Pre-conditions: list? ls1 & list? ls2
Post-condition: result = the appended list of the two lists
Tests: (append '(1 2 3) '(4 5 6)) ⇒ '(1 2 3 4 5 6)
*/
(define append
  (lambda (lst1 lst2)
     (if (empty? lst1)
        lst2
        (cons (car lst1) (append (cdr lst1) lst2))
     )
   )
)

/*
Signature: append3(lst1,lst2,num)
Purpose: To take two lists and one item, and to return the first list
appended with the second and the item
Type: [list(symbol)*list(symbol)* (number|boolean|string) -> list(symbol)]
Example: (append3 '(1 2 3) '(4 5 6) 7) should produce '(1 2 3 4 5 6 7)
Pre-conditions: list? ls1 & list? ls2 and num is an number or Boolean or string
Post-condition: result = the appended list of the two lists with the number
Tests: (append3 '(1 2 3) '(4 5 6) 7) ⇒ '(1 2 3 4 5 6 7)
*/
(define append3
  (lambda (lst1 lst2 num)
    (append lst1 (append lst2 (cons num '()))))
   )
)

/*
Signature: pascal(n)
Purpose: To take an integer n, and to compute the nth row of Pascal's triangle
Type: [number -> list(number)]
Example:  (pascal 5) should produce '(1 4 6 4 1)
Pre-conditions: n must be natural number (positive and an integer)
Post-condition: result = the nth row of Pascal's triangle
Tests:  (pascal 5) => '(1 4 6 4 1)
*/
(define pascal
   (lambda (n)
      (tempRowPascal (- n 1) 0)
   )
)

(define tempRowPascal
   (lambda (n k)
      (if (< k n)
```

```
        (append (list (nCk n k)) (tempRowPascal n (+ k 1)))
        '(1)
      )
    )
)

(define nCk
  (lambda (n k)
    (if (= 0 k)
      1
      (/ (* n (nCk (- n 1) (- k 1))) k)
    )
  )
)
```