CS315 – HW3

Name: Elif Gülşah Kaşdoğan

ID: 21601183

Section: 3

Description of Function: My function counts the total number of atoms, it traverses through all depth of given list and returns the total count. Atoms of lists can be numbers or strings. *foo* takes only a list as parameter and calls an auxiliary function *foo_aux*. Elements of lists can be any combination of atoms and lists of any depth. Lists may have arbitrary length.

Code:

```
(define (foo lst)
  (foo_aux lst 0)
)
(define (foo_aux lst aux)
  (cond
    ((null? lst) aux)
    ((list? (car lst)) (foo_aux (cdr lst) (+ aux (foo (car lst)))))
    (else (foo_aux (cdr lst) (+ aux 1)));atom
   )
)
```

Explanation of how it works:

Auxiliary parameter aux is used for counting the number of elements as we go deeper in the given list. Conditions are arranged according to logical relations among each other as Scheme requires.

foo_aux function goes until it encounters a null element, it returns the count calculated so far when it reaches to a null element and it is our base case.

If the current element we are investigating is a list, foo function will be called with current element. For the rest of the list foo_aux will be called. Total number of elements of the new called auxiliary function will be the sum of current count and result of foo function called.

If the current element is not a list but an atom, number of elements will be incremented by one for the current element and foo_aux will be called for the rest of the list.

Explanation of why it is tail recursive:

A function is tail recursive if its recursive call is the last operation in function as defined in our course book. Meaning return value of the foo_aux is return value of the foo. Since we want our input to be lists of any depth, foo needed to be called for list elements which are lists. We will need new activation record for elements which are lists, however foo_aux activation records can still be overwritten. We will have one main foo activation record, internal calls to foo and foo_aux can be rewritten and this will not change the output rather will provide efficiency. foo_aux is tail recursive because of these reasons.

How it is invoked:

(function_name param1 param2 … paramN) is the form of function invocation in Scheme (foo '(1 2 3 4 5 6 7)) is the way I call my non-recursive function foo. Foo calls tail-recursive foo_aux function with 0 as second parameter which is the auxiliary parameter that holds the number of atoms counted so far. foo returns the total number of atoms throughout the list. Atoms are counted directly followed by a recursive call to rest of the list, for elements which are lists foo function is called again to follow a similar procedure.

Test results:

Test code:

```
(foo '(1 2 3 4 5 6 7 8))
(foo '(3 4 (1 2 5 6) 7 (8)))
(foo '((1 2) (3 4) 5 6 (7 8)))
(foo '((1) 2 (3) 4 (5) 6 (7) 8))
(foo '(1 (2) 3 (4) 5 (6) 7 (8)))
(foo '((1 2 3 4 5 6 7 8)))
(foo '(((1 (2) 3 (4)) 5 6) 7 (8)))
(foo '(a b c d))
(foo '(a (b c d)))
(foo '((a b c) d))
(foo '(a (b c) d))
```

Results:

```
> (foo '(1 2 3 4 5 6 7 8))
8
> (foo '(3 4 (1 2 5 6) 7 (8)))
8
> (foo '((1 2) (3 4) 5 6 (7 8)))
8
> (foo '((1) 2 (3) 4 (5) 6 (7) 8))
8
> (foo '(1 (2) 3 (4) 5 (6) 7 (8)))
8
> (foo '((1 2 3 4 5 6 7 8)))
8
> (foo '(((1 (2) 3 (4)) 5 6) 7 (8)))
8
> (foo '(a b c d))
4
> (foo '(a (b c d)))
4
> (foo '((a b c) d))
4
> (foo '(a (b c) d))
4
```