Programming Languages

CS 315

Fall 2018-2019

HOMEWORK 3

TAIL RECURSION IN SCHEME

Deniz Yüksel

21600880

SECTION 1

INSTRUCTOR: HALİL ALTAY GÜVENİR

27.12.2018

This is the report to the scheme language homework. The below section includes the answers to the questions below:

- the description of my function
- the code in Scheme
- explanation of how it works
- explanation of why it is tail recursive
- how it is invoked
- test results

My function foo works by calling another function, foo-aux. The function takes only a list. Then calls foo-aux function which takes the list and an integer to represent the running multiplication. The initial value passed to running multiplication is 1. To sum up, my two functions take a list and then multiplies all the elements in a tail recursive fashion, keeping a running cumulative multiplication.

My function:

```
(define (foo oneList)
  (foo-aux oneList 1))

(define (foo-aux oneList runningMult)
  (cond
      ((null? oneList) runningMult)
      ((list? (car oneList))
      (foo-aux(append (car oneList) (cdr oneList)) runningMult))

  (else (foo-aux (cdr oneList) (* (car oneList) runningMult)))))
```

First of all, the foo-aux function takes two parameters. The second parameter is given 1 in the foo method and it's stable. On the other hand, the list that is passed to the function always changes. If the list is null, then running multiplication is returned. If the first element taken from the list which is the car of the list, is a list, then append function is called with car of the list and cdr of the list. This value is the new version of the list, with the deep lists removed. Therefore, I call foo-aux with the new appended list, but without changing the value of running multiplication. So that when the newly called foo-aux function is evaluated, the car of the list is not a list but it is an atom, so evaluation can take place. The main process which is in the else statement is calling foo-aux with the cdr of the list, which is the list that contains all elements except the first one. The second parameter of foo-aux then will be the multiplication of running multiplication and the first element of the current list. This function is tail recursive because the function keeps the running multiplication which is not dependent on previous recursive calls. In other words, this tail recursive code is a greedy algorithm because it solves the problem in every step. The recursive calls that we were used to were waiting for the previous return values to reach a final conclusion.

This method can be invoked after typing scheme in the dijkstra server and pasting this code. Afterwards, if the user presses "Enter" then the user can type in for example:

(foo '(1 2 (3 2) ((5)) 1))

This call gives the list of (1 2 (3 2) ((5)) 1) to the function and will return the multiplication of these numbers which is 60.

In addition, my test results in dijkstra server are below:

Gambit v4.7.9

```
> (define (foo oneList)
  (foo-aux oneList 1))
>
(define (foo-aux oneList runningMult)
  (cond
        ((null? oneList) runningMult)
        ((list? (car oneList))
        (foo-aux(append (car oneList) (cdr oneList)) runningMult))

  (else (foo-aux (cdr oneList) (* (car oneList) runningMult)))))
>
(foo '(3 2 (4 5) (((2))) 1 2))
480

#<procedure #2 >>
> (define (foo-aux oneList runningMult)
  (cond
        ((null? oneList) runningMult)
        ((list? (car oneList))
        (foo-aux(append (car oneList) (cdr oneList)) runningMult))

  (else (foo-aux (cdr oneList) (* (car oneList) runningMult)))))
>
(foo '(1 1 1 (2 (22))2))
88
> (define myList '(1 2 3))
> (foo myList)
6
>
> (define myList '(1 2 a b))
> (foo myList)
*** ERROR IN foo-aux, (console)@10.32 -- (Argument 1) NUMBER expected
(* 'a 2)
```