BBM 301 - Programming Languages - Fall 2020
Assignment 2

Due Date: 10 January 2021, 23:59

Tail Recursion in Scheme

Objective: In this homework, you will learn how to convert the recursive Scheme functions into tail recursive ones, and writing iterative functions.

PART A: Converting recursive functions to tail recursive ones
Recall that a function is *tail recursive* if its recursive call is the last operation in the function. A tail recursive function can be automatically converted by a compiler to use iteration, making it faster.

We have seen that a function can be made tail recursive by using a helper function, which we will call as accumulator, as in the following example.

Original:
```
(define (factorial n)
(if (= n 0)
      1
      (* n (factorial (- n 1)))
))
```

Tail recursive:
```
(define (fact-accumulator n factpartial)
(if (= n 0)
      factpartial
      (fact-accumulator (- n 1) (* n factpartial))
))
(define (factorial n)
        (fact-accumulator n 1))
```

We can write the tail recursive function also as follows.

```
(define (factorial n)
(letrec
((fact-accumulator (lambda (n factpartial)
                   (if (= n 0)
                   factpartial
                   (fact-accumulator (- n 1) (* n factpartial))
                   ))

))
(fact-accumulator n 1)
))
```

Here we make fact_accumulator a local procedure using letrec and lambda. (for the use of letrec see https://edoras.sdsu.edu/doc/mit-scheme-9.2/mit-scheme-ref/Lexical-Binding.html and https://www.cs.cmu.edu/Groups/AI/html/r4rs/r4rs_6.html)

Task1 (10 pts):
Now, consider the following example for finding the length of a list.

```
(define (length lst)
   (if (null? lst)
       0
       (+ 1 (length (cdr lst)))))
```

If we step through the evaluation of this functions on `(list 1 2 3 4 5)` it will be like:

```
(+ 1 (+ 1 (+ 1 (+ 1 (+ 1 0))))) => 5
```

We can write the tail recursive function as follows.

```
(define (length lst)
(letrec (
        (length_helper (lambda (lst current_length)
                             (if (null? lst)
                              current_length
                             (length_helper (cdr lst)(+ 1 current_length))
                        ))
        ))
(length_helper lst 0)
))
```

Here, we ask you to step through the evaluation of the tail recursive function. Write briefly about your observations.

Task2 (20 points):
Consider the following recursive function that computes the sum of squares of the first n numbers.

```
(define sum-of-squares
       (lambda (n)
       (if (= n 0)
       0
       (+ (sum-of-squares (- n 1)) (* n n)))) )
```

a) Write a tail recursive version for the same function using `letrec`.
b) Then, step through the evaluation of the original and tail recursive functions for `(sum-of-squares 5)`. Write briefly about your observations.

Task3 (30 points)

a) Write a recursive function `sum-of-factorials-of-elements` that takes a list, and returns the sum of factorials of the elements of the list. Call the factorial function defined above.

   For example `(sum-of-factorials-of-elements '(1 3))` should return `(+ (factorial 1) (factorial 3)) => 7`

b) Turn the above function into a tail recursive function.

c) Step through the evaluation of the original and tail recursive functions for the following call. Use the result of `factorial` directly.

```
(sum-of-factorials-of-elements '(3 2 5 1 4))
```

PART B: Writing iterative functions

The syntax for a do loop in Scheme is as follows:

```
(do ((<variable> <initial-value> <update>) ...)
(<termination-test> <expression> ...)
<statement> ...)
```

For example, the following code determines the length of a list iteratively:

```
(define (length lst)
(do ((len 0 (+ len 1)))
((null? lst) len)
(set! lst (cdr lst))))
```

Here is another example that sums the numbers between *1* and *n* iteratively:

```
(define (sum n)
(do ((i 1 (+ i 1))
(sum 0))
((> i n) sum)
(set! sum (+ sum i)))
)
```

Task4 (10 points):
Write an iterative function for `sum-of-squares`.

Task5 (20 points):
Write an iterative function for `sum-of-factorials-of-elements`.

Task6 (10 points)
Compare the recursive, tail recursive and iterative versions of the `sum-of-squares` function based on the requirement for intermediate storage and time for very large numbers (e.g. 30000000)

Write a report including your answers to the above tasks. Submit your report **in pdf format**, to gradescope (Course Entry Code: **JBB784**).

Important: *Your report **will be checked against plagiarism**, so make sure that you have correctly and fully cited all the references that you have used, and make sure that you **answer***

***using your own sentences**. Significant overlap with existing resources will result in significant loss of points.*