Principles of Programming languages, Assignment 1.2

Due date: ***Tuesday 14.4.2015***

Iterative & recursive computational process and high order procedures.

Submission instructions:

- You are provided with an archive file named ***id1_id2.zip*** (see the assignments page). After extracting the contents of this archive file, you will find two other files: a word document, ***ex1-2.doc*** and a racket file ***ex1-2.rkt****.*
- In ***ex1-2.doc****,* write your answers to theoretical questions (1-2). You may include your answers in a PDF file ***ex1-1.pdf****.*
- In ***ex1-2.rkt***, write your solutions to the programming question (3). Within the file, you will find partial contracts for the procedures you should write. Implement each procedure right below its corresponding contract and complete the contract. **A contract must be included for EVERY procedure you implement**.
- You should submit the archive file ***id1_id2.zip*** (like this: 123456789_987654321.zip) containing <u>only</u> the files mentioned above. The zip file should not create any additional directories when inflated.

Important Note: Your implementations are tested automatically. Please make sure to name all procedures and file names by the **exact same names** as appears in the assignment description. The archive file is provided this for this purpose.

---

1. **(20 Points)** Recall that the let construct in Scheme is implemented as a syntactic sugar, such that its evaluation is performed by translation into an application of a lambda expression.
   a) What would be the result of such translation on this expression?

```
(let ((x 2))
    (let ((x 3)
          (y x))
         ((lambda(x y +) (+ x y)) (- x y) x *)))
```

   b) And what would be the result of such translation on that expression?

```
(let ((x 4)
      (y (+ x y)))
     (+ x y (let ((x y)
                  (y x))
                 (+ x y))))
```

2. **(40 Points)** Consider the following procedure for counting the number of steps in the challenge known as the *Towers of Hanoi*:

```
; Signature: count-1(n)
; Type: [Number -> Number]
; Purpose: Count the number of steps in moving n disks from one tower to
; another, via a third tower, according to the challenge known as the Towers of
; Hanoi: this is a recursive implementation that generates a tree-recursive
; process; the two recursive calls directly follow the intention of first
; moving n-1 disks to the third tower, then moving the remaining disk to the
; target tower, and finally moving the n-1 disks from the third to the target
; tower.
; Example: (count-1 1) = 1, (count-1 2) = 3, (count-1 3) = 7
; Pre-conditions: n is an integer, such that 1 <= n
; Post-conditions: n = 2^(n-1)

(define count-1 (lambda (n)
  (if (= n 1)
      1
      (+ (count-1 (- n 1)) 1 (count-1 (- n 1))))))
```

   a) Consider the three alternative implementations below. Complete the contract of each of the procedures (including helper procedures) in the *ex1-2.doc* file. As in the example above, in the "Purpose" part of each implementation explain the kind of process it generates at runtime. Is it a tree-recursive process again? Or is it a linear-recursive, or even an iterative process?

```
  ; Signature:
  ; Type:
  ; Purpose:
  ; Pre-conditions:
  ; Tests:

(define count-2 (lambda (n)
  (if (= n 1)
      1
      (+ 1 (* 2 (count-2 (- n 1)))))))



(define count-3 (lambda (n) (count-3-helper n 0 1)))

(define count-3-helper (lambda (n a m)
  (if (= n 0)
      a
      (count-3-helper (- n 1) (+ a m) (* m 2)))))



(define count-4 (lambda (n) (count-4-helper n (lambda (x) x)))

(define count-4-helper (lambda (n cont)
  (if (= n 1)
      (cont 1)
      (count-4-helper (- n 1) (lambda(res) (cont (+ 1 (* 2 res))))))))
```

b) Consider one more alternative implementation (below). Comparing it to the implementation of `count-4` above: Which of the corresponding helper functions is tail-recursive? Are the applications of `cont` in both cases tail recursive? Briefly explain the implications of your answers on the memory consumption at runtime.

```
(define count-5 (lambda (n) (count-5-helper n (lambda () 1)))

(define count-5-helper (lambda (n cont)
  (if (= n 1)
      (cont)
       (count-5-helper (- n 1) (lambda() (+ 1 (* 2 (cont)))))))))
```

3. **(40 Points)** In this question you are requested to implement some arithmetic calculators, using simple, and then high-order procedures.
   a. Your first implementation should be of the procedure `simple-calc(op n1 n2)`, where `op` is a number that denotes an elementary arithmetic operation (0 for addition, 1 for subtraction, 2 for multiplication, and 3 for division) and where `n1`,`n2` are two numbers. The expected result is that of applying the corresponding operation to the two numbers, `n1` and `n2`. For example:

```
> (simple-calc 0 1 2)
3
```

   b. In a second step, you are expected to implement the procedure `advanced-calc(calc op n1 n2)`, where `calc` is a procedure with the same type and pre-condition as that of `simple-calc` above, and where `op`, `n1` and `n2` are again numbers, `op` being an integer between 0 and 5 this time. When the integer number `op` is between 0 and 3, the advanced calculator returns the result of applying `calc` on `op`, `n1` and `n2`; otherwise, when `op` is 4 the operation to perform is of raising `n1` to the power of `n2`, and when `op` is 5 we return the average of `n1` and `n2`.
   For example:

```
> (advanced-calc simple-calc 0 1 2)
3
> (advanced-calc simple-calc 5 10 20)
15
```

   c. Finally, implement the procedures `fun-add` and `fun-sub` according to the following contracts (and the example interaction below it):

```
; Signature: fun-add(f g)
; Type: [[Number -> Number]*[Number -> Number] -> [Number -> Number]]
; Purpose: given two functions f and g, construct a function that given a
; number x returns the sum of applying f to x and g to x
; Pre-condition: true
; Post-conditions: result = closure r, such that r(x) = f(x)+g(x)
; Tests: ((fun-add (lambda (x) x) (lambda (x) 1)) 0) => 1

; Signature: fun-sub(f g)
; Type: [[Number -> Number]*[Number -> Number] -> [Number -> Number]]
; Purpose: given two functions f and g, construct a function that given a
; number x returns the difference of f applied to x and g applied to x
; Pre-condition: true
; Post-conditions: result = closure r, such that r(x) = f(x)-g(x)
; Tests: ((fun-add (lambda (x) x) (lambda (x) 1)) 0) => -1
```

For example:

```
> (define 1+ (lambda(x) (+ x 1)))
> (define 2+ (lambda(x) (+ x 2)))
> ((fun-add 1+ 2+) 0)
3
> ((fun-sub 1+ 2+) 0)
-1
```

GOOD LUCK!