CSE1729 - Introduction to Programming	September 23, 2018
Laboratory Assignment 4	

Objectives

- Work with "tail recursion"
- Work with higher order functions

Activities

1. Recall from Problem Set 3: The Lucas numbers are a sequence of integers, named after Édouard Lucas, which are closely related to the Fibonacci sequence. In fact, they are defined in very much the same way:

$$L_n = \begin{cases} 2 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ L_{n-1} + L_{n-2} & \text{if } n > 1. \end{cases}$$

   (a) Ask your SCHEME interpreter to compute L_{30}, then L_{35}, then L_{40}. What would you suspect to happen if you asked it to compute L_{50}?

   (b) *Computing with a promise.* Consider the following SCHEME code for a function of four parameters called `fast-Lucas-help`. The function call

   `(fast-Lucas-help n k lucas-a lucas-b)`

   is supposed to return the nth Lucas number *under the promise that it is provided with any pair of previous Lucas numbers.* Specifically, if it is given a number $k \leq n$ and the two Lucas number L_k and L_{k-1} (in the parameters `lucas-a` and `lucas-b`), it will compute L_n. The idea is this: If it was given L_n and L_{n-1} (so that $k = n$), then it simply returns L_n, which is what is was supposed to compute. Otherwise assume $k < n$, in which case it knows L_k and L_{k-1} and wishes to make some "progress" towards the previous case; to do that, it calls `fast-Lucas-help`, but provides L_{k+1} and L_k (which it can compute easily from L_k and L_{k-1}). The code itself:

```
(define (fast-Lucas-help n k lucas-a lucas-b)
  (if (= n k)
      lucas-a
      (fast-Lucas-help n (+ k 1) (+ lucas-a lucas-b) lucas-a)))
```

   With this, you can define the function `fast-Lucas` as follows:

```
(define (fast-Lucas n) (fast-Lucas-help n 1 1 2))
```

   (After all, $L_0 = 2$ and $L_1 = 1$.)

   Enter this code into your SCHEME interpreter. First check that `fast-Lucas` agrees with your previous recursive implementation (Lucas) of the Lucas numbers (on, say, $n = 3, 4, 5, 6$). Now evaluate `(fast-Lucas 50)` or `(fast-Lucas 50000)`.

   There seems to be something qualitatively different between these two implementations. To explain it, consider a call to `(Lucas k)`; how many total recursive calls does this generate to the function `Lucas` for $k = 3, 4, 5, 6$? Now consider the call to `(fast-Lucas-help k 1 1 2)`; how many recursive calls does this generate to `fast-Lucas-help` for $k = 3, 4, 5, 6$? Specifically, populate the following table (values for $k = 1, 2$ have been filled-in):

	Recursive calls made by (Lucas k)	Recursive calls made by (fast-Lucas-help k 1 1 2)
$k=1$	0	0
$k=2$	2	1
$k=3$		
$k=4$		
$k=5$		
$k=6$		

2. Abstracting the summation of a series

   (a) Consider the harmonic numbers $H_n = \frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$. Last week you wrote a recursive SCHEMEfunction (named `harmonic`) which, given a number n, computes H_n. Revise your `harmonic` function, keeping the name `(harmonic n)`, to take advantage of the `sum` function seen in the textbook (Section 1.3.1) and shown below:

   ```
   (define (sum term a next b)
       (if (> a b)
           0
           (+ (term a)
              (sum term (next a) next b))))
   ```

   Of course, your new and improved definition of harmonic should not be recursive itself and should rely on `sum` to do the hard work.

   (b) The above definition of sum is a recursive process. Write an iterative version `sum-i` that solves the same problems as `sum` in an iterative fashion. Demonstrate that it works by using it to define `harmonic-i`.

   (c) Show that your harmonic functions work for 1, 50, and 100.

3. SICP Exercise 1.42 - Let f and g be two one-argument functions. The composition f after g is defined to be the function $x \mapsto f(g(x))$. Define a procedure, named `(compose f g)`, that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

   ```
   ((compose square inc) 6)
   49
   ```

4. SICP Expercise 1.43 - If f is a numerical function and n is a positive integer, then we can form the n^{th} repeated application of f, which is defined to be the function whose value at x is $f(f(...(f(x))...))$. For example, if f is the function $x \mapsto x+1$, then the n^{th} repeated application of f is the function $x \mapsto x+n$. If f is the operation of squaring a number, then the n^{th} repeated application of f is the function that raises its argument to the 2^n th power. Write a procedure named `(repeated f n)`, that takes as inputs a procedure that computes f and a positive integer n and returns the procedure that computes the n^{th} repeated application of f. Your procedure should be able to be used as follows:

   ```
   ((repeated square 2) 5)
   625
   ```

   You may want to take advantage of the `compose` function you wrote.

5. The 91 function was introduced in papers published by Zohar Manna, Amir Pnueli and John McCarthy in 1970. These papers represented early developments towards the application of formal methods to program verification. Consider the following form for the 91 function:

   $$f(x) = \begin{cases} x-10, & \text{if } x > 100 \\ f^{91}(x+901), & \text{if } x \le 100 \end{cases}$$

   where $f^{91}(y)$ stands for $f(f(\cdots f(y)\cdots))$, the 91-times-repeated application of f. That is, f composed with itself 90 times. Write a SCHEME function, named `(m91 x)` which computes the 91 function as defined above. You should notice it has some interesting behavior for $x \le 100$.