CSE 335
winter 2017 Homework 2

IMPORTANT:
Always check the tests to see the full usage of the functions. Not all use cases are listed in this file.
--
This homework introduces recursion over list and higher order procedures.

Given your background in Java you are probably extremely familiar and comfortable with using loops and an iterative approach to process arrays and collections. In functional programming, loops are not very common, programmers rely on the recursive nature of the data structures they use, so they write recursive algorithms to do the computation.

As you might have noticed lists can be define recursively as follows:

```
<list> ::=  null
          | data <list>

;this is BNF(Backus-Naur form), we will learn more about it
;in later homework
```

In plain english, the above description says that a list is either null (empty) or it contains some data and another list. This is similar to how you, probably, implemented linked lists in your algorithms classes; a node contained some data and a reference to the next node, thus defining a node in terms of itself.

To reflect on this a bit more, in Racket, say we have the list l:

```
>(define l '(1 2 3))
>(car l); this would correspond to the "data"
        ; element described in
        ; the grammar from now on referred to
        ; as the "head" of the list


>(cdr l); this value corresponds to the remainder
        ;of the list, from now on
        ; referred to as the "tail" of the list.
```

To better grasp the concept, evaluate the following expression and relate the return value of the function to the recursive definition of a list:

```
>(cdr '(one-element))
```

The implication of the above on how we process lists is that we will be using recursive algorithms. For example, consider the very simple

function that takes a list of numbers as arguments and adds them together:

```
(define (add-all lst)
   ;first our base case, or stopping condition.
   ;If the list is empty then we return 0
   (if (null? lst)
        0
        ;otherwise we add the head of the list
        ;to whatever value is computed
        ;for the rest of the list
        (+ (car lst) (add-all (cdr lst))))
)
```

Suggested reading:
STRUCTURE AND INTERPRETATION OF COMPUTER PROGRAMS:
   SECTIONS 1.1.6, 1.1.7, 1.3

==========

1. [10p] list-of-even-numbers

Implement a function list-of-even-numbers? that takes a list as argument and tells whether or not the list contains only even numbers.

```
>(list-of-even-numbers? '(4 20 16 2))
#t
>(list-of-even-numbers? '(1))
#f
>(list-of-even-numbers? (list "a-string" 'a-symbol 42))
#f
```

As you can see, you may not assume that the list will contain ONLY integers.

Note:
Generally, in Scheme/Racket, when you encounter a function that ends in a question mark, "?", it is an indication that this function never terminates in an error, but only returns #t or #f. Such functions are called predicates.

Hint:
You may want to use the library functions "number?" and "even?". These ones also follow the rule of thumb described above.

==========

2. [10p] Series

In mathematics, a series, is a sum of the terms of a progression. A sequence is a string of numbers that progress based on a certain rule. For instance, the progression described by the rule An = 2∗n results in: 0 2 4 6 8 10 …

Then the series constructed from this progression would be:
  Sn = 0 + 2 + 4 + 6 + 8 + 10 + …

Write a function, for each of the series bellow, that computes the sum up to the nth term (inclusively), where n is the only parameter of the function.
Look at the tests to see what the computed values are.
--
2.a [5p]

```
        1
An = -------
      (n^2)
```

```
;for n > 0
Sn = 1/1 + 1/4 + 1/9 + 1/16 + …
```

--
2.b [5p]

```
       (-1)^n
An = -----------
       (n + 1)!
```

```
;for n >= 0
Sn = 1 - 1/2 + 1/6 - 1/24 + …
```

--
Note:
You will notice that the test values are written as fractions. By default, Racket, does not use floating point notation to do arithmetic because it is imprecise. To force it to use floating point you have to write a #i in front of at least one number that is involved in the computation. For instance:

```
> (/ 1 2)
1/2
> (/ #i1 2)
0.5
```

You can find more information on how numbers are represented in Racket here:
           http://docs.racket-lang.org/guide/numbers.html

==========
For problems 3, 4 consider using list manipulation functions introduced in class (See Jan18 Lecture notes, in particular slide 7)
==========
3. [15p] Carpet

Write a procedure carpet that takes one non-negative natural number as

argument and produces a list as shown in the examples.

You may assume:
 -that the argument will always be a non-negative natural number.

Output properties:
 - output does not have to be pretty-printed and it must contain NO spaces at the head or tail of the list
 - '+ and '% have to be symbols, not strings.

```
> (carpet 0)
‘((%))

> (carpet 1)
‘(( + + + )
 ( + % + )
 ( + + + ))

> (carpet 2)
‘( ( % % % % % )
  ( % + + + % )
  ( % + % + % )
  ( % + + + % )
  ( % % % % % ))

>(carpet 3)
( ( + + + + + + + )
  ( + % % % % % + )
  ( + % + + + % + )
  ( + % + % + % + )
  ( + % + + + % + )
  ( + % % % % % + )
  ( + + + + + + + ))
```

==========

4. [15p] Pascal's Triangle

The pattern you see below called Pascal's Triangle:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.

Write a function with the behavior:

```
>(pascal 1)
‘((1))
```

```
>(pascal 2)
‘(  ( 1 )
  ( 1 1 )
)

>(pascal 3)
‘(   ( 1 )
   ( 1 1 )
  ( 1 2 1 )
)

>(pascal 4)
‘(   ( 1 )
   ( 1 1 )
  ( 1 2 1 )
 ( 1 3 3 1 )
)
```

You may assume:
 -that the argument will always be a natural number, which is greater or equal to one.

Output properties:
 - output does not have to be pretty-printed. That is,

```
‘(   ( 1 )
   ( 1 1 )
  ( 1 2 1 )
 ( 1 3 3 1 )
)
```

is exactly the same as: ‘((1) (1 1) (1 2 1) (1 3 3 1)) .

==========

5. [20p] Parenthesis balancing

Write a function that takes a string as parameter and tells us whether or not it contains balanced parentheses.

```
>(balanced? "(balanced? ())")
#t

>(balanced? ":'(")
#f
```

Prior to doing any computation it is *highly* recommended that you transform the string to a list of characters using the (string->list "some-string") function.

Working recursively with the head and tail of a list will greatly reduce the complexity of the problem.

You may assume:
  - that the input is always a string
  - an empty string or one with no parentheses is considered balanced

==========
The reason why Scheme/Racket is called a functional language is because functions are first class citizens, meaning: they can be treated as any other value in the program. It was mentioned in the description of homework 1, and repeated in class that whenever the interpreter encounters an unquoted open parenthesis it expects the first value to be a function.

The consequences of these two statements can be summarized by the following example:

```
> (define add-alias +)
> (add-alias 1 2 3 4)
10
```

The first expressions binds the standard library function "+" to the variable "add-alias". The second expression uses this variable as if it were the add function itself, no additional scheme (pun intended) is required.

This language feature is very useful for creating good abstractions, thus reducing code cloning.

By now you might have noticed that function declarations can be written in two different ways:

```
>(define (foo arg1 arg2)
  (+ arg1 arg2))


>(define foo
  (lambda(arg1 arg2)
    (+ arg1 arg2))
)
```

What the lambda expression (sometimes referred to as lambda abstraction) does is that it creates an anonymous function with the specified number of arguments.

The second version is akin to simple variable binding:

```
>(define forty-two 42)
```

Here you bind the value 42 to the variable forty-two, similarly you bind the function *as a value* created by the lambda to the variable foo.

==========
6. [15p]

In light of this new information let us reconsider problem 1. If you want to write a function that tests whether a list contains only strings, odd numbers, or other lists that contain only even numbers you will notice that the code that iterates through the list stays the same, while only the code that does the concrete verification has to be changed. If we were to write a function for each of the data types listed above, it would be a sign of bad abstraction.

By passing in a parameter, which is a function, that can handle the specific data in the list, and by letting the "list-of-all?" function deal with the list processing logic - which is another way to look at responsibility assignment as you learned in O-O design - we can write a more generic version that is used as follows:

```
>(list-of-all? string? '("a" "b" "42"))
#t

>(list-of-all? number? '(1 2 3 "not-a-number"))
#f

>(list-of-all?

       ;this lambda abstraction tells us if the argument is an odd
number
       (lambda (n) (and (integer? n) (odd? n)))
       '(1 3 5)
)
#t
```

Implement the function list-of-all? to work in the manner presented above and in the test cases.

Note:
Unfortunately, a common pitfall is to write code like:

```
(define (list-of-all? fun lst)
  (if (equal? fun number?) (number? (car lst)))
   ...
)
```

In this example the function checks to see if the parameter function is equal to a specific function and then using the later function

explicitly. Although this approach can work, it defeats the purpose of the whole abstraction, you are still tied to the very few cases you can cover AND it doesn't work when you pass in lambda abstractions. If you do this, or something similar, you will not receive any points for this problem.

There is a similar standard library function called "listof". You are NOT allowed to use that one in this particular implementation. You are, however, encouraged to use it in any other problem.

==========
7. [15p] create-mapping

A map data structure is described as either a set of key-value pairs or a set of keys and a set of values whose relation is described by a function with a one-to-one mapping from keys to values.

Write a function, create-mapping, which takes a list of symbols (keys) and a list of any type of Scheme values (vals) and returns a function that takes one symbol argument (the key) and returns the corresponding value.

The mapping is determined by the position of the elements in the two lists. For
example:

```
;keys
> (define roman-numerals-keys '(I II III IV V))

;values
> (define arabic-numerals-vals '(1 2 3 4 5))

;The pairs mapped will be:
;I   -> 1
;II  -> 2
;III -> 3
;and so forth

;the (create-mapping) function returns a function that we bind for
later use
> (define roman-to-arabic (create-mapping roman-numerals-keys arabic-
numerals-vals))
> (roman-to-arabic 'I)
1
> (roman-to-arabic 'V)
5
> (roman-to-arabic 'some-symbol)
uncaught exception: "Could not find mapping for symbol 'some-symbol"
;hint: create the error message in a similar way you did for problem
14 of hw01.
```