Homework 1: Functions
CPSC3223: Programming Languages and Translation

Instructions:

- Complete all of the following problems.
- Use the supplied unit tests to test your solutions.
- Do not include error checking (assuming argument types are correct, for example).
- When you are asked to write a function, you may write additional functions that help you simplify the problem.
- Do not use **def** or **defn** inside of functions. These define global variables and using them inside a function usually indicates a misunderstanding of proper functional programming.
- Do not use any of the built-in Clojure functions/macros except: **def, defn, fn, if, cond, let, arithmetic operations, comparison operations, and, or, not, str, conj, first, rest, list, concat, zero?, list?, empty?, nil?, symbol?, odd?, even?, assoc**
- Submit your solution on Canvas as a single `.tar.gz` or `.zip` file containing your entire project folder.

Defining simple functions

1. Define a function `times3` that takes a single argument and returns the value of this argument multiplied by three. Do not include any error checking (assume that the argument is a number).
2. Define a function `times-3-or-4` that takes a single argument and if the argument is odd, returns 3 times the argument otherwise 4 times the argument.
3. Define a function `xcubed-plus-one` of a single argument x that returns $x^3 + 1$.
4. Write a recursive procedure `(sum-squares n)` which computes:

$$\sum_{i=1}^{n} i^2, \quad n > 0$$

Your solutions must use recursion (do not use any of the looping constructs supplied by Clojure).

HOMEWORK CONTINUED ON NEXT PAGE!

Maps

Create the following functions. You will need some understanding of data types that were part of the reading.

5. `(deposit account amount)` accepts a map, `account`, and a number, `amount`, as arguments. The map `account` will map the keyword `:balance` to a number. Your function `deposit` must return a new map with the `:balance` value increased by `amount`. (Note that the REPL may not print the elements in the same order as my demo below but order doesn't matter for maps.)

```
> (deposit {:owner "Fred", :id 12983, :balance 1000} 150)
{:owner "Fred", :id 12983, :balance 1150}
> (deposit {:owner "Barney", :id 7187, :balance 3000} 89)
{:owner "Barney", :id 7187, :balance 3089}
```

6. `(account-str account)` accepts a map with the keys `:owner, :id` and `:balance` and returns a string formatted as shown in the examples below.

```
> (account-str {:owner "Fred", :id 12983, :balance 1000})
Account 12983 owned by Fred with balance $1000
> (account-str {:owner "Barney", :id 7187, :balance 3000})
Account 7181 owned by Barney with balance $3000.
```

Listmania!

Create the following functions (many of these problems are modified with permission from "Principles of Programming Languages" by Friedman and Wand). Remember the restrictions on the list of functions. No credit will be given to solutions that use functions that are not on that list. You will use recursion on every one of these problems. If your brain doesn't hurt by the end of this, you aren't doing it right.

7. `(duple n x)` returns a list containing `n` copies of `x`.

```
> (duple 2 3)
(3 3)
> (duple 4 '(ha ha))
((ha ha) (ha ha) (ha ha) (ha ha))
> (duple 0 'blah)
()
```

8. `(invert lst)`, where `lst` is a list of 2-lists (lists of length two), returns a list with each 2-list reversed.

```
> (invert '((a 1) (a 2) (1 b) (2 b)))
((1 a) (2 a) (b 1) (b 2))
```

(**Hint: Create one or more "helper" functions to decompose the problem into simpler functions.**)

9. `(down lst)` wraps each top-level element of `lst` in a list.

```
> (down '(1 2 3))
((1) (2) (3))
> (down '((a) (fine) (idea)))
(((a)) ((fine)) ((idea)))
> (down '(a (more (complicated)) object))
((a) ((more (complicated))) (object))
```

10. `(swapper s1 s2 slist)` returns a list the same as `slist`, but with all occurrences of `s1` replaced by `s2` and all occurrences of `s2` replaced by `s1`.

```
> (swapper 'a 'd '(a b c d))
(d b c a)
> (swapper 1 2 '(1 2 3 4 3 2 1))
(2 1 3 4 3 1 2)
> (swapper 'a 'd '(a d () c d))
(d a () c a)
> (swapper 'x 'y '((x) y (z (x))))
((y) x (z (y)))
```

(**Hint: Create one or more "helper" functions to decompose the problem into simpler functions.**)

11. `(list-set lst n x)` returns a list like `lst`, except that the `n`-th element, using zero-based indexing, is `x`. `n` will always satisfy `(and (>= n 0)(< (count lst)))` (that is, it will be a valid **existing** index).

```
> (list-set '(a b c d) 1 'monkey)
(a monkey c d)
> (list-set '(a b c d) 2 '(1 2))
(a b (1 2) d)
```

12. `(count-occurrences s lst)` returns the number of occurrences of `s` in `lst`.

```
> (count-occurrences 'x '((f x) y (((x z) x))))
3
> (count-occurrences 'x '((f x) (y (((x z) () x)))))
3
> (count-occurrences 'w '((f x) y (((x z) x))))
0
```

13. `(product lst1 lst2)`, where `lst1` and `lst2` are lists without repetitions, returns a list of 2-lists that represents the Cartesian product of `lst1` and `lst2`. The 2-lists may appear in any order.

```
> (product '(a b c) '(x y))
((a x) (a y) (b x) (b y) (c x) (c y))
> (product '(1 2) '(q s t))
((1 q) (1 s) (1 t) (2 q) (2 s) (2 t))
```

(**Hint: Create one or more "helper" functions to decompose the problem into simpler functions.**)