Homework 2: More Clojure
CPSC3220: Programming Languages and Translation

Instructions:

- Complete all of the following problems.
- **Include a complete set of unit tests for your solutions.** You must write unit tests for each function that you are asked to write.
- 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 indicates a misunderstanding of proper functional programming.
- Except in the "Listmania" section, you may use any of the built-in Clojure functions that you like.
- Submit your solution on Canvas as a single `.tar.gz` or `.zip` file containing your entire project folder.

Begin by using `lein` to create a project called `hw2`. **You must use precisely this project name and place your code and tests in the correct files for a lein-based project.**

Listmania!

For this section 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, seq, cons, first, rest, nth, zero?, seq?, empty?, nil?, odd?, even?, assoc**. No credit will be given to solutions that use functions that are not on this list.

Create the following functions (many of these problems are modified with permission from "Principles of Programming Languages" by Friedman and Wand). 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.

1. `(prepend n x coll)` returns a sequence (a **seq**!) containing `n` copies of `x` and the remainder of the elements of the collection `coll`.

```
> (prepend 2 'monkey [1 2 3])
(monkey monkey 1 2 3)
> (prepend 3 [1 2] '(:x :y :z))
([1 2] [1 2] [1 2] :x :y :z)
> (prepend 0 :a [:x :y :z])
(:x :y :z)
```

2. `(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.**)

3. `(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))
```

4. `(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.**)

5. `(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)
```

6. `(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
```

7. `(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.**)

Other stuff

You can use any built-in Clojure functions to complete the remainder of these exercises.

8. (Useful: `map`, `nth`) Write a function, (`column` table n), where `n < (count (first table))` is a number and `table` is a non-empty vector of vectors. All nested vectors in `table` will have the same non-zero size. That is, `table` is a "matrix." Return a **seq** containing the elements of the `n`-th column (zero-based indexing) of `table`. Sample runs:

```
> (def table [[1 2 'a 'b]
              [3 4 'apple 'banana]
              ['x 'y 'p 'z]])
> (column table 0)
(1 3 x)
> (column table 1)
(2 4 y)
> (column table 3)
(b banana z)
```

9. Create a function of 3 arguments, `(merge-pred pred lst1 lst2)`, where `pred` is a predicate function of two arguments, and `lst1` and `lst2` are lists. `pred` will indicate an ordering relationship. `(pred a b)` returns true if and only if $a \leq_p b$ and returns false otherwise. The arguments `lst1` and `lst2` must already be sorted in increasing order according to `pred`.

   In the event of a tie (`pred` considers two elements to be "equal"), the elements from `lst1` should appear before the tied elements of `lst2`. Note: "equal" does not mean the same. For example, if `pred` compares elements based on their string length, "hello" and "olleh" would be considered equal.

   The return value of `merge-pred` should be the results of merging the elements of the lists in sorted order (according to `pred`). Sample runs:

```
> (merge-pred <= '(1 4) '(1 2 8))
(1 1 2 4 8)
> (merge-pred #(<= (count %1) (count %2)) '("a" "ab" "abcdef") '("c"
    "abc"))
("a" "c" "ab" "abc" "abcdef")
```

10. (Useful: `take`, `drop`) Create a function of 2 arguments, `(sort-pred pred lst)` where `pred` is an ordering predicate as described in the previous problem and `lst` is a list. The return result of `sort-pred` should be the elements of `lst` in increasing order according to `pred`. In the event of a tie (`pred` considers two elements to be equal), the resulting order of the elements may be ambiguous.

    You may not use any of Clojure's built-in sorting functions or consult their source code in your solution.

    Sample runs:

    ```
    > (sort-pred <= '(8 9 5 -1 0 2 3 -15))
    (-15 -1 0 2 3 5 8 9)
    > (sort-pred >= '(8 9 5 -1 0 2 3 -15))
    (9 8 5 3 2 0 -1 -15)
    ```

    Hint: implement the merge sort algorithm using your `merge-pred` function from the previous problem.