Homework 1: Functions
CPSC3220: 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 (assume 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.
- 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, cons, first, rest, nth, zero?, empty?, nil?, odd?, even?, assoc**
- Submit your solution on Canvas as a single `.tar.gz` or `.zip` file containing your entire project folder.

Simple functions

1. Define a function `times3` that takes a single numeric argument and returns the value of this argument multiplied by three.
2. Define a function `times-3-or-4` that takes a single integer argument and if the argument is odd, returns 3 times the argument otherwise 4 times the argument.
3. Write a Clojure function called `poly` that takes a single numeric argument `x` and returns $-x^3 + 3x^2 - 4x - 17$.
4. Define a function `perform` that takes three arguments: `operation`, `value1` and `value2`. The argument `operation` will be one of the following symbols: `add, subtract, multiply divide`. The arguments `value1` and `value2` will be numbers. Return the result of the specified operation on the two numeric arguments. Examples:

```
> (perform 'add 1 2)
3
> (perform 'subtract 1 2)
-1
> (perform 'multiply 5 2)
10
> (perform 'divide 1 3)
1/3
```

Lists and vectors

5. Define a function `third` that takes a single list argument (containing at least 3 elements) and returns the third element of that list.
6. Define a function `tenth` that takes a single vector argument (containing at least ten elements) and returns the tenth element of that vector. Yes, this is very similar to the previous question :)
7. Define a function `sum3` that takes a single list argument (containing at least 3 numeric elements) and returns the sum of the first three elements of that list.
8. Define a function `firstodd?` that takes a single vector argument (containing at least 1 numeric element) and returns true if the first element of the vector is odd or false otherwise.
9. Define a function `list-info-str` that takes a single argument that will either be a list or `nil` and returns a string as follows:
   - If the argument is nil, return the string `"nil"`;
   - if the argument is a empty list, return the string `"empty"`;
   - if the argument has exactly one element, return the string `"1␣element"`;
   - otherwise return the string `"<n>␣elements"` where `<n>` is replaced by the number of elements in the list.

   Important: in addition to the functions listed in the instructions, you may use the `count` function for this problem. A couple examples:

```
> (list-info-str nil)
"nil"
> (list-info-str '(1 2 3))
"3␣elements"
> (list-info-str '(a))
"1␣element"
```

Maps

Note: in the descriptions below you may notice that when I describe a function I show how it would be called. This is typical notation in Clojure. So, for example, the first question is asking you to define a function called `deposit` that takes two arguments called `account` and `amount`. I will use this notation throughout the course.

10. Define a function `(make-account id owner)` that takes integer (id) and string (owner) as arguments. The function must return a map with key `:owner` set to the owner argument, key `:id` set to the id argument and key `:balance` set to 0 (zero). Examples:

```
> (make-account 10 "Fred")
{:owner "Fred", :id 10, :balance 0}
> (make-account 99 "Barney")
{:owner "Barney", :id 99, :balance 0}
```

11. Define a function `(deposit account amount)` that 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`, if `amount` is positive or unchanged (if amount is not positive).

```
> (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}
> (deposit {:owner "Wilma", :id 111, :balance 2000} -3)
{:owner "Wilma", :id 111, :balance 2000}
> (deposit {:balance 33} 5)
{:balance 38}
```

12. `(account-str account)` accepts a map with (at least) 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.
> (account-str {:id 5, :balance 10, :owner "Dino", :who-cares 55})
Account 5 owned by Dino with balance $10.
```