TCSS 380, Winter 2020, Lab 5

The objective of this lab assignment is to give you more practice with Erlang tuples and list syntax, as well as list logic in functional programming

You must work with a partner during the lab. Find a partner of your own choosing – it could be a person you worked with before but not during the last lab. If you cannot find a partner, ask the lab instructor for help. Before the end of the lab submit all the parts you complete during the lab session – one submission per partnership. Remember to list both names at the top of each file on which you work together. At the very least half of the lab must be completed during the lab session to receive full credit. If you do not complete some part within the two-hour session, finish at home on your own or with your lab partner by the designated due date. If you complete this part of the lab on your own, list your name only. If you complete this part with your lab partner, list both names and submit only once per partnership. All together, you need to upload:

- lab05.erl
- lab05_tests.erl (could be completed after the lab)
- screenshot of running lab05_tests.erl in the shell (png, jpg, or gif) (could be completed after the lab)

In the text editor of your choosing create a file **lab05.erl** and write function definitions described below – make sure you use the exact same names as indicated in function descriptions. Instead of exporting all functions one by one, use command *-compile(export_all)* at the top of your file. Download the file **lab05_tests.erl**, add Erlang unit test library to it, and as you work on your solutions, write unit tests for your functions (unit tests for the first three functions are provided).

1. function *fractions* that that takes two tuples representing fractions (each tuple is a fraction consisting of a numerator and a denominator) and calculates and returns the result of multiplying these fractions by one another; the resulting fraction should also be a tuple and you do NOT need to simplify the result, e.g. if {1, 2} and {2, 5} are sent to this function, then {2, 10} is returned (hint: to pattern match a function on a tuple consisting of 2 elements, you can use *functionName ({X, Y}) ->* … type of a clause)
2. function *isOlder* that takes two dates (each date is a 3-int tuple *mm, dd, yyyy*) and evaluates to true or false. It evaluates to true if the first argument is a date that comes before the second argument (e.g. 3/4/2012 comes before 3/30/2012, hence a person born on 3/4/2012 is older). (If two dates are the same, the result is false.). Think about how you can use relational operators on entire tuples to simplify processing instead of writing nested ifs
3. function *flipAll* that takes a list of numbers and returns the original list with signs of all elements flipped, e.g. if the original list contains [1, -2, 3], then [-1, 2, -3] is returned; <u>you are NOT allowed to use ++ or built-in functions or list comprehension</u> (you should use the cons operator |)
4. function *myMin* that takes a list and returns its minimum value – use tail recursion to find list minimum and return an atom *empty_list* if a list is empty; <u>you are NOT allowed to use built-in functions;</u> add appropriate test cases to the test file that use *assertEqual* macro
5. function *removeNonPerfects* that takes a list of integers and returns the copy of this list with only perfect squares remaining; a perfect square is an integer that can be expressed as the product of two equal integers, e.g 9 can be expressed as the product of 3 x 3; so if this function is called with [1,2,3,5,6,7,8,9,25,36], the resulting list should contain [1,4,9,25,36]
   To determine whether a number is a perfect square use the following helper function:

```
isPerfectSquare(X) ->
    Sqrt = math:sqrt(X),
    (Sqrt - math:floor(Sqrt)) == 0.
```

   When using this functions in *removeNonPerfects*, use the case expression – it is not mandatory to do so but it will be easier to write the matching statement using the case expression.

you are NOT allowed to use built-in functions or list comprehension; add appropriate test cases to the test file that use *true* matching pattern (i.e. the other method that does NOT use *assertEqual*)

6. function *calculateBill* that takes a list of tuples, where a tuple is of the form *name, price, taxRate*, and returns a list of tuples, where each tuple is of the form *name, total*
   where *total = price + price * taxRate*
   add appropriate test cases to the test file
7. function *generate* – takes three ints as arguments and generates a list of integers from arg1 to arg2 (inclusive) in increments indicated by arg3, e .g. if arg1 = 3, arg2 = 8, and arg 3 = 2, then the function returns [3, 5, 7]. If arg1 > arg2, returns an empty list; assume arg3 will be a valid number; you are NOT allowed to use built-in functions or list comprehension; add appropriate test cases to the test file
8. function *getnth* that takes a list and an int n and returns the nth element of the list, where the head of the list is the 1st element. You are only allowed to use list functions hd and tl in your solution – no other built-in functions are allowed. If the list is empty or n is invalid, return a tuple:
   {error, no_such_element}
   add the following test cases to the test file:
   getnth_1_test() -> true = {error, no_such_element} == lab05:getnth([ ], 2).
   getnth_2_test() -> true = {error, no_such_element} == lab05:getnth(["hello", "there"], 3).
   getnth_3_test() -> true = {error, no_such_element} == lab05:getnth(["hello", "there"], 0).
   getnth_4_test() -> true = "there" == lab05:getnth(["hello", "there"], 2).
   getnth_5_test() -> true = "there" == lab05:getnth(["hello", "there", "where"], 2).
   getnth_6_test() -> true = "where" == lab05:getnth(["hello", "there", "where"], 3).
   getnth_7_test() -> true = "where" == lab05:getnth(["hello", "there", "where", "here"], 3).
9. function *repeat* that takes a list of integers and a list of nonnegative integers and returns a list that repeats the integers in the first list according to the numbers indicated by the second list; add the following test cases to the test file – these test files also illustrate the logic of repetition to be followed
   repeat_1_test() -> [2, 2, 2, 2, 3] = lab05:repeat([1, 2, 3], [0, 4, 1]).
   repeat_2_test() -> [ ] = lab05:repeat([ ], [0, 4, 1]).
   repeat_3_test() -> [ ] = lab05:repeat([1, 2, 3], [ ]).
   repeat_4_test() -> [4,4] = lab05:repeat([4,5,6], [2]).

--- CONGRATS – YOU ARE DONE – UNTIL NEXT LAB THAT IS ☺ ---