Submission

Course:	Design of Distributed Systems
Teacher:	Toth Melinda
Assignment:	Test1
Deadline:	2018-10-17, 20:05:00
Time left:	-day(s) --:--:--
Reamining:	1

Deadline is reached.

2018-10-17, 19:55:0250%

Description

Please take into account the followings:

- **do not use list comprehensions**
- **do not use if expressions**
- **you may use library functions, but minimum one solution has to be implemented as a recursive function**
- **using class materials, additional Erlang files, any source from the internet are not allowed!**

Task no. 0:

Copy the following declaration form to your Erlang file and fill in the required data:

```
%% <Name>
%% <Neptun ID>
%% <DDS, TEST1>
%% <17.10.2018.>
%% This solution was submitted and prepared by <Name, Neptun ID> for the DDS Retake Test Sequential.
%% I declare that this solution is my own work.
%% I have not copied or used third-party solutions.
%% I have not passed my solution to my classmates, neither made it public.
%% Students' regulation of Eötvös Loránd University (ELTE Regulations Vol. II. 74/C. § ) states that as long as a student presents another student's work - or at least the significant
```

The link of the documentation.

Task no. 1: Counting

Define a function `count/1` that takes a list `L` as an argument. The function returns a list of pairs (two-element tuples: {Elem, {Smaller, Greater}}). The first element of each tuple is the element of the list (Elem), the second element of the pair is a pair containing the number of elements of the list that are smaller than Elem and elements of the list that are greater then Elem.

To calculate the number of smaller and greater elements you have to use one list traversing!

```
count(L::list(any())) -> Result::list(tuple(any(), tuple(integer(), integer())))
```

Test cases:

Do not forget to change the name of the module!

```
test:count([2,1,3,0]) == [{2,{2,1}},{1,{1,2}},{3,{3,0}},{0,{0,3}}]
test:count([]) == []
test:count("apple") == [{97,{0,4}},{112,{3,0}},{112,{3,0}},{108,{2,2}},{101,{1,3}}]
test:count([3,apple, pear,2, 3, apple, "apple"]) == [{3,{1,4}}, {apple,{3,2}}, {pear,{5,1}}, {2,{0,6}}, {3,{1,4}}, {apple,{3,2}}, {"apple",{6,0}}]
test:count([3,4,5,1,2,3,1,0]) == [{3,{4,2}}, {4,{6,1}}, {5,{7,0}}, {1,{1,5}}, {2,{3,4}}, {3,{4,2}}, {1,{1,5}}, {0,{0,7}}]
```

Task no. 2: Raising

Define a function `change/3` that takes three arguments. The first is an atom, `EmpId`, that represents the identifier of an employee. The second argument is a number, `Rise`, representing the percentage of the expected rise. The third one is a map, `Map`, representing the set of employee data. The keys in the map are atoms, the values in the map are represented as tuples: containing the name of the employee as a string and the salary as a number. For example `#{melinda=>{"Melinda Tóth", 100}}`.

Your task is to update the salary of the employee whose id is `EmpId` with the percentage `Rise`. The return value contains the name of the employee, the new salary, and the updated map container.

Conversions between maps and lists are not allowed!

```
change(EmpId::atom(), Rise::integer(), Map::map()) -> Result::tuple(tuple(string(), number()), map()))
```

Test cases:

Do not forget to change the name of the module!

```
test:change(melinda, 20, #{melinda=>{"Melinda Tóth", 100}}) == {{"Melinda Tóth",120.0}, #{melinda => {"Melinda Tóth",120.0}}}
test:change(melinda, 20, #{melinda=>{"Melinda Tóth", 100}, daniel =>{"Dániel Lukács", 100} }) == {{"Melinda Tóth",120.0}, #{daniel => {"Dániel Lukács",100}, melinda => {"Melinda Tóth"
test:change(daniel, 100, #{melinda=>{"Melinda Tóth", 100}, daniel =>{"Dániel Lukács", 100} }) == {{"Dániel Lukács",200.0}, #{daniel => {"Dániel Lukács",200.0}, melinda => {"Melinda Tó
test:change(daniel, 0, #{melinda=>{"Melinda Tóth", 100}, daniel =>{"Dániel Lukács", 100} }) == {{"Dániel Lukács",100.0}, #{daniel => {"Dániel Lukács",100.0}, melinda => {"Melinda Tóth
test:change(nonexisting, 0, #{melinda=>{"Melinda Tóth", 100}, daniel =>{"Dániel Lukács", 100} }) == {not_found,#{daniel => {"Dániel Lukács",100}, melinda => {"Melinda Tóth",100}}}
test:change(nonexisting, 0, #{})== {not_found,#{}}
```

Task no. 3: Special zipping

Define a function `c_zip/2` that takes two lists as arguments: `List1` and `List2`. The function traverses the two lists simultaneously and merges/zips the two lists into a single list of tuples, where the first element of the tuple is the element of `List1` and second is the element of `List2`. The specialty of this function is in the lists which lengths are different. Once you run out elements from the shorter list, you have to restart the packing from the first element again. For example, when the lists are `[1,2]` and `[a,b,c]`, you can create the tuples `{1,a}` and `{2,b}`, but `c` has no pair in the first list, so you have to restart and create the third tuple as `{1, c}`.

```
c_zip(List1::list(), List2::list()) -> Result::list(tuple())
```

Test cases:

Do not forget to change the name of the module!

```
test:c_zip([1,2], [a, b]) == [{1,a},{2,b}]
test:c_zip([1,2], [a, b, c, d]) == [{1,a},{2,b},{1,c},{2,d}]
test:c_zip([1,2], [a, b, c, d, e]) == [{1,a},{2,b},{1,c},{2,d},{1,e}]
test:c_zip([1,2], [a]) == [{1,a},{2,a}]
test:c_zip([1,2], []) == []
```

Task no. 4: Selecting elements based on occurence

Define a function `select/1` that takes a list as an argument and returns the elements that occur at least 3-times in the list. The maximum point can be gained only if you do not do unnecessary list traversing!

```
select(List::list()) -> Result::list()
```

Test cases:

Do not forget to change the name of the module!

```
test:select([2,3,4,3,1]) == []
test:select([2,3,2,3,1, 3, 2, 3,3,3,3,3]) == [3,2]
test:select([2,3,2,3,1, 3, 2, 3]) == [3,2]
test:select([2,3,2,3,1, 3,3]) == [3]
test:select([]) == []
test:select([2,3,2,3,1, 3, 2]) == [3,2]
```