DIT027 – Assignment 3

The exercises are supposed to be done in pairs. However, it is allowed to discuss the problems with other groups and share ideas (but not code). There are 22 tasks to solve. Solving at least 14 of them is enough for passing the assignment.

Don't forget to write comments in your code to make your solutions clearer!

The submission deadline is extended to **15 September 2017** at 23:55.

A test suite will be provided to test the module that you create. It is required to test your solution with the test suite before submitting, and to include the output from running the test suite on your code.

The test suite will also be used to evaluate your code. Passing the test suite is required, but not sufficient to pass the assignment.

You should also not use library functions, but define your own. Unless there are specific requirements on the implementation, you can use functions that you have already implemented for implementing other functions.

Overcomplicated solutions will not be accepted.

Make sure that you follow ALL the instructions closely!

GOOD LUCK!

1. Download the module skeleton **assignment3.erl** from GUL (from Documents). You should be able to compile the module without modifications

2. **Recursive functions on lists –** In this problem you are supposed to write a few basic Erlang functions handling lists. Please name them exactly as in the examples. You can assume that the input is correct, i.e. that only non-negative integers, and well-formed lists are given to the functions.

   A. Implement function **adj_duplicates/1**, which takes a list and returns a list containing only the elements that have an adjacent duplicate (i.e., all elements E such that the element following E is identical to E).

   Ex: adj_duplicates([1,1,2,2,3,3]) → [1,2,3]

adj_duplicates([1,2,2,2,3]) → [2,2]
adj_duplicates([a,a,b,c,d]) → [a]
adj_duplicates([7,3,4,3,1]) → []

B. Write the function **split/2** that given a non-negative integer **N** and a list splits the list into a pair of lists. If the length of the argument list is at least **N** then the length of the first result list should be **N**, while the other result list should contain the remaining elements of the argument list. Otherwise if the length of the argument list is less than N, the second result list should be empty.

Ex: split(2, [a, b, c, d, e]) → {[a, b], [c, d, e]}
split(3, [1, 2, 3]) → {[1, 2, 3], []}
split(2, [a]) → {[a], []}

C. Write the function **normalize/1**, which given a list of numbers, out of which at least one is positive, normalizes that list, that is divides each number by the largest one in the original list, such that the largest number in the new list is equal 1. Use list comprehensions to define the function. You may need to define an auxiliary function to solve this problem.

Ex: normalize([0,1,2,3,4]) → [0.0,0.25,0.5,0.75,1.0]
normalize([-5,3,-1,1]) →
[-1.6666666666666667,1.0,-0.3333333333333333,
0.3333333333333333]

D. Reimplement function **normalize/1** as **normalize2/1** using **lists:map/2**.

E. The function **last/1** takes a list as argument and returns its last element. When called with the empty list, it crashes. The function's implementation is unnecessarily complex, and very inefficient for long lists due to the calls to **length/1**. Modify the function keeping the same functionality, but improving the efficiency and implementing it as a function with multiple clauses and pattern matching. The function should not use case- or if-expressions.

Ex: last([a, b, c, d]) → d
last([]) → (crash)
last([x]) → x

F. Write the function **find/2**, which given a predicate and a list of values attempts to find a value in the list that satisfies the predicate. If such value is found, the pair **{found, Val}** should be returned, where **Val** is the first value that satisfies the predicate. If no such value is found, the atom **not_found** should be returned.

Ex: find(fun erlang:is_list/1, [a, [], 1]) → {found, []}
find(fun (X) -> X > 5 end, [1, 2, 3]) → not_found

G. Write the function **sort/1**, which given a list of values returns a sorted list containing the same elements. The implementation does not have to be efficient. Hint: You may use (reverse) insertion sort: To sort a list of N + 1 elements, first sort the tail of the list, and then insert the first element into the right place in the sorted list.

Ex: sort([3, 1, 2]) → [1, 2, 3]
sort([c, a, b, c]) → [a, b, c, c]

H. Write the function **map/2**, which takes a unary function and a list as arguments, applies the function to every element of the list, and returns a list of results the function. The function should behave in the same way as **lists:map/2**, and be implemented using recursion.

Ex: map(fun (X) -> X + 2 end, [3, 1, 2]) → [5, 3, 4]
map(fun (X) -> hd(X) end, [[a, b], [c]]) → [a, c]

I. Write the function **map2/2**, which takes a unary function and a list as arguments, applies the function to every element of the list, and returns a list of results the function. The function should behave in the same way as **lists:map/2**, and be implemented using a list comprehension.

Ex: map(fun (X) -> X + 2 end, [3, 1, 2]) → [5, 3, 4]
map(fun (X) -> hd(X) end

J. Write the function **filter/2**, which takes a unary predicate (a function that returns a Boolean value) and a list as arguments, applies the function to every element of the list, and returns a list of values for which the result was **true**. The function should behave in the same way as **lists:filter/2**, and be implemented using recursion.

Ex: filter(fun (X) -> X rem 2 == 0 end, [1, 2, 3, 4]) → [2, 4]
filter(fun (X) -> length(X) > 1 end, [[a, b], [c]]) → [[a, b]]

K. Write the function **filter2/2**, which takes a unary predicate (a function that returns a Boolean value) and a list as arguments, applies the function to every element of the list, and returns a list of values for which the result was **true**. The function should behave in the same way as **lists:filter/2**, and be implemented using list a comprehension.

Ex: filter(fun (X) -> X rem 2 == 0 end, [1, 2, 3, 4]) → [2, 4]
filter(fun (X) -> length(X) > 1 end, [[a, b], [c]]) → [[a, b]]

3. **Digitify a number –** write a function **digitize/1**, which given a positive number N returns a list of the digits in that number.

   Ex: digitize(473) → [4, 7, 3]
   digitize(8) → [8]
   digitize(-123) → should crash/generate an exception error

4. **Happy numbers –** A positive number is happy if by repeated application of the procedure below the number 1 is reached.

   1. Square each of the digits of the number
   2. Compute the sum of all the squares

   For example, if you start with 19:

   - 1 * 1 + 9 * 9 = 1 + 81 = 82
   - 8 * 8 + 2 * 2 = 64 + 4 = 68
   - 6 * 6 + 8 * 8 = 36 + 64 = 100
   - 1 * 1 + 0 * 0 + 0 * 0 = 1 + 0 + 0 = 1 (i.e. 19 is a happy number)

   How do you know when a number is not happy? In fact, every unhappy number will eventually reach the cycle 4, 16, 37, 58, 89, 145, 42, 20, 4, … thus it is sufficient to look for any number in that cycle (say 4), and conclude that the original number is unhappy.

   Write the functions **is_happy/1**, and **all_happy/2**, which returns whether a number is happy or not (true or false) and all happy numbers between N and M respectively. (**Hint:** use the functions digitize and sum).

   Ex: is_happy(28) → true
   is_happy(15) → false
   all_happy(5, 25) → [7, 10, 13, 19, 23]

5. **Expressions –** We define a small expression language, and write a *parser, pretty printer,* and *evaluator.* Expressions are for example:

((5+8)*3) 7 ((2*3)+(7-2))

It is enough if you handle the (binary) operators (+,-,*). To simplify the problem, all parentheses are mandatory and expressions contain no spaces or extra parentheses. Let us represent the binary operators by atoms, **`plus`**, **`minus`**, and **`mul`**. The numbers are tagged with **`num`.**

A. Write an evaluation function (**expr_eval/1**). It should evaluate an expression (assume only well-structured expressions!) and return a number.

{mul,{plus,{num,5},{num,8}},{num,3}} → 39

{plus,{num, 17},{num,10}} → 27

B. Write a pretty printer (the function **expr_print/1**) which returns a string containing the expression in a nicely formatted way. Don't forget to add all parentheses!

{mul,{plus,{num,5},{num,8}},{num,3}} → "((5+8)*3)"

{plus,{mul,{num,2},{num,3}},{minus,{num,7},{num,2}}} →
"((2*3)+(7-2))"

Hints:

To create a string containing a single digit, you may use the expression [$0 + N], where N is equal to the digit.

You may use the library functions **lists:append/1** and **lists:append/2** to solve this problem.

6. **Dictionary –** In this task, you are supposed to implement a dictionary data structure, which is a mapping of a finite number of keys (any Erlang value can be a key) into values (again any Erlang value). A dictionary provides the following operations implemented as functions: **dict_new()**, **dict_get(Dict, Key)** and **dict_put(Dict, Key, Val)**. The function **dict_new()** returns an empty dictionary. The function **dict_get(Dict, Key)** checks if there is a value corresponding to the specified key in the dictionary, returns **{found, Val}** if a value is found, and **non_found** otherwise. The function **dict_put(Dict, Key, Val)** returns a pair **{NewDict, Result}**, where **NewDict** is a new dictionary with added **Key** mapped to **Value** if the key was not mapped previously. If the key was already mapped, its value is updated. The second part of the function's result (**Result**) should be **{previous, OldVal}** if the key was previously mapped to **OldVal**, and **fresh** otherwise.

A dictionary should be represented by a pair **{dict, List}**, where **List** is a list of pairs of the form **{Key, Val}**. The list should contain the pair **{Key1, Val1}** once if and only if the **Key1** is mapped to **Val1**.

Ex: dict_new() -> {dict, []}

dict_get({dict, [{b, 2}, {a, 5}, {c, 3}]}, c) → {found, 3}

dict_get({dict, [{b, 2}, {a, 5}, {c, 3}]}, d) → not_found

dict_put({dict, [{b, 2}, {a, 5}, {c, 3}]}, d, 1) →
{{dict, [{b, 2}, {a, 5}, {c, 3}, {d, 1}]}, fresh}

dict_put({dict, [{b, 2}, {a, 5}, {c, 3}]}, a, 4) →
{{dict, [{b, 2}, {a, 4}, {c, 3}]}, {previous, 5}}

A. Implement the functions **dict_new()**, **dict_get(Dict, Key)** and **dict_put(Dict, Key, Val)**. You may use the functions that you already defined for other problems.

B. Implement the function **dict_wellformed(Dict)**, which checks whether its argument is a valid representation of a dictionary, and returns **true** if it is or **false** if it is not.

Ex: dict_wellformed({dict, [{b, 2}, {a, 5}, {c, 3}]}) → true
dict_wellformed({dict, [{b, 2}, {a, 5}, {b, 3}]}) → false

C. Implement the function **dict_map_values(F, Dict)**, which takes a function and a dictionary, and returns a dictionary with the same keys, whose values are the corresponding values from the source dictionary transformed by the function.

Ex: dict_map_values(fun (X) -> X + 2 end,
{dict, [{b, 2}, {a, 5}, {c, 3}]})
→ {dict, [{b, 4}, {a, 7}, {c, 5}]}

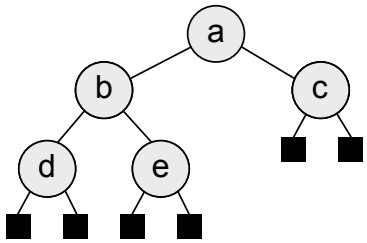
7. **Trees –** In this task, you are supposed to implement a data type of binary trees with labels in their internal nodes. Tasks **B**, **C** and **D** are quite advanced. A tree is either leaf, which contains no data, or a branch, which consists of two trees and a label. The following representation should be used for trees:
   - A leaf should be represented with the atom **leaf**.
   - A branch should be represented with a tuple **{branch, T1, T2, L}**, where **T1** and **T2** are subtrees, each of which is a leaf or another branch, and **L** is a label, which can be any Erlang value.

A. Implement the function **tree_wellformed(Tree)**, which given an argument returns true if the argument is a well-formed tree, and false otherwise

Ex: tree_wellformed(leaf) → true
tree_wellformed({branch, {branch, leaf, leaf, 1}, leaf, 2}) → true
tree_wellformed({branch, leaf, {branch, aaa, leaf, b}, a}) → false

B. Implement the function **tree_make_bfs(List)**, which given a list creates a complete tree (its lowest level does not have to be full, but the nodes have to be as far left as possible). The resulting tree should have as many internal nodes as the length of the argument list. The labels of the internal nodes should be drawn from the list in the BFS order.

Ex. tree_make_bfs([a, b, c, d, e]) results in the following tree being created (the squares denote leaves).



C. Implement the following functions:

- **tree_bind(Tree1, Tree2)**, which takes two trees as arguments, and creates a new tree, which is the same as **Tree1** with every leaf replaced with **Tree2**.
- **tree_flatten(Tree)**, which takes a tree as argument and returns a list of its labels in BFS order.

  Ex. tree_flatten(tree_make_bfs([a, b, c, d, e])) → [a, b, c, d, e]
- **tree_map(F, Tree1)**, which takes a function and a tree as arguments, and creates a new tree, which is the same as **Tree1,** but with every leaf transformed using function **F**.

D. Implement the following functions:

- **tree_dfs(Tree)**, which takes a tree as argument and returns a list of its labels in DFS order.

  Ex. tree_dfs(tree_make_bfs([a, b, c, d, e])) → [a, b, d, e, c]

- **tree_sorted(Tree)**, which takes a tree as argument and returns the boolean **true** if the tree is sorted or **false** if it is not. A tree is sorted if for each internal node, the nodes in its left subtree have lower or equal labels to the node, while nodes in its right subtree have greater or equal labels.

  Ex. tree_sorted(tree_make_bfs([a, b, c, d, e])) → false
  tree_sorted(tree_make_bfs([b, a, c])) → true

- **tree_find(Tree1, Tree2)**, which takes two trees as arguments and attempts to find a subtree, which is equal to **Tree1** within **Tree2**. If such subtree is found, the atom **true** should be returned. Otherwise, **false** should be returned.

  Ex. tree_find(tree_make_bfs([b, d, e]),
  tree_make_bfs([a, b, c, d, e])) → true

  tree_find(tree_make_bfs([1, 2, 3]),
  tree_make_bfs([a, b, c, d, e])) → false