DIT027 – Assignment 3

The exercises are supposed to be done individually. However, it is allowed to discuss the problems with other students and share ideas with other students. There are 6 problems containing 25 tasks to solve (some problems contain only 1 task). Solving at least 14 of the tasks (this includes 1.A) 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 **21 September 2016** 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. **(solved)** Write a function **sum/1**, which given a list of integers returns the sum of all integers in that list. The function should not use case- or if-expressions.

   Ex: sum([3, 4, 1, 6]) → 14
   sum([]) → 0

B. Write a function **sum_interval/2**, which given integers N and M returns the sum of all integers between N and M inclusive. When N > M, the result of **sum_interval** is 0. The function should be implemented as a recursive function.

Ex: sum_interval(2,4) → 9
sum_interval(4,2) → 0

C. Write the function **interval/2,** which given integers N and M, returns the list of all integers between N and M inclusive. When N > M, the resulting list should be empty.

Ex: interval(2,4) → [2,3,4]
interval(4,2) → []

D. Reimplement function **sum_interval/2** as **sum_interval2/2** using **sum/1** and **interval/2** functions.

E. 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]) → []

F. Write a function **even_print/1**, which given a list of integers prints all the even numbers in that list in the order they appear. Each number should be printed in a separate line. The function returns the atom ok. The function should be implemented as a recursive function.

Ex: even_print([3,0,5,11,8]) → ok (printing ”0~n8~n”, where ~n is new line)

G. Write a function **even_odd/1**, which given an integer N decides if it’s even or odd. When N is even, the function should return the atom even, and the atom odd otherwise.

Ex: even_odd(4) → even
even_odd(7) → odd
even_odd(-2) → even

H. Reimplement function **even_print/1** as **even_print2/1** using list comprehensions and **even_odd/1**.

I. Write a 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]

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

K. Write a function **sum2/1**, which has exactly the same behaviour as **sum/1**. The function should only have one clause and use a case-expression.

L. 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

M. Write a function **mul/1**, which given a list of integers returns the product of the numbers.

Ex: mul([1, 2, 3]) → 6
mul([]) → 1

N. Write a 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

O. Write a 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.

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

3. **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}]}
```

4. **Trees –** In this task, you are supposed to design a data type of binary trees with labels in their internal nodes. A tree is either leaf, which contains no data, or a branch, which consists of two trees and a label.

   Implement the following functions:

   A. Implement the following functions:

   - **tree_leaf()**, which returns a leaf.
   - **tree_branch(Tree, Tree, Label)**, which given two trees and a label creates a branch having the two argument trees as children and is labeled with the argument label.
   - **tree_deconstruct(Tree)**, which given a tree returns **dec_leaf** if the tree is a leaf, and **{dec_branch, Tree1, Tree2, Label}** if it is a branch, with **Tree1**, **Tree2** and **Label** being the children trees and the label of the node.

Ex:
```
tree_deconstruct(tree_leaf()) → dec_leaf

case tree_deconstruct(tree_branch(tree_leaf(),
                                   tree_leaf(), a)) of
    {dec_branch, T1, T2, L} -> {dec_branch,
                                   tree_deconstruct(T1),
                                   tree_deconstruct(T2), L}
end → {dec_branch, dec_leaf, dec_leaf, a}
```

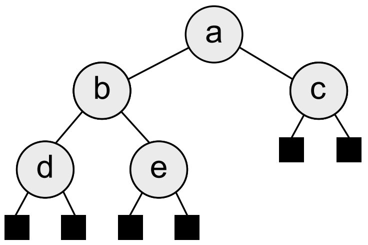
```
case tree_deconstruct(tree_branch(tree_leaf(),
                                  tree_branch(tree_leaf(),
                                              tree_leaf(),
                                              b),
                                  a)) of
    {dec_branch, T1, T2, L} ->
       {dec_branch, tree_deconstruct (T1),
                    case tree_deconstruct(T2) of
                       {dec_branch, T21, T22, L2} ->
                          {dec_branch,
                              tree_deconstruct(T21),
                              tree_deconstruct(T22),
                              L2}
                    end, L}
end → {dec_branch, dec_leaf, {dec_branch,
                              dec_leaf,
                              dec_leaf, b}, a}
```

The tree data type should not be opaque, i.e. its structure should be exposed to the user and well-defined. It is a common practice in Erlang to expose the structure of data types to the user to make programming with pattern matching possible. Implement the following additional function:

- **tree_wellformed(Tree)**, which given an argument returns true if the argument is a well-formed tree (according to your definition), and false otherwise.

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_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

5. 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

6. **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]

7. **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.