DIT027 – Exercises on Sequential Erlang

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 5 problems to solve. Some sub-problems are marked as optional, you should try to solve them, but don't spend too much time if you get stuck.

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

The submission deadline is **19/9 at 8:00!**

To simplify your work with testing your solutions I will provide some test cases at a later date. I will send a message when the test cases are ready, together with instructions on how to use them!

Make sure that you follow ALL the instructions closely!

GOOD LUCK!

0. (Not really a problem) Download the module skeleton **seq_erlang.erl** from GUL (under Documents/Assignment 1). You should write the solutions to all problems in this single file/module. In this way it will be easier to (semi-automatically) correct your submissions. There is a skeleton for each function you should implement. Feel free to write helper-functions where you like, but do not export these functions in your final submission (it could be good to export them during development and testing though...)

Note: You are of course not allowed to use lists-library functions and BIFs unless something else is stated in the assignment.

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

   A. Write a function **sum/1**, which given a positive integer N returns the sum of all integers between 1 and N.

   Ex: sum(5) → 15

   B. Write a function **mul/1**, which given a positive integer N returns the product of all integers between 1 and N.

   Ex: mul(4) → 24

   C. Write the function **take_snd/1,** which given a list of two-tuples, returns a list containing the second element of each tuple.

   Ex: take_snd([{1,2},{a,b},{foo,bar},{17,21}]) → [2,b,bar,21]

2. **Fibonacci numbers -** The *Fibonacci numbers* are defined by:

   Fib(0) = 1

   Fib(1) = 1

   Fib(n+2) = Fib(n+1) + Fib(n)

   So Fib(2) = 2, Fib(3) = 3, Fib(4) = 5, …

   Write a recursive function **fib/1** which calculates the n:th Fibonacci number.

   (Optional) The recursive implementation of fib is very inefficient, try it for n > 35 and you'll see the bad performance. There are more efficient (but less elegant) ways of implementing the fib-function. Try to implement a more efficient version in **eff_fib/1**.

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]`
   Ex: `digitize(8) → [8]`
   Ex: `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).

   Examples:
   - `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))"

   C. Write a parser (the function **expr_parse/**1) for expressions, turning *strings* into an Erlang representation, such as

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

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

   Note: In this exercise you are allowed to use functions from the standard `lists` library. The BIFs integer_to_list/1 and list_to_integer/1 could also be useful.

   (Optional) Make sure that you handle operator precedence correctly. This means that 3*4+5 = 17 and not 27. This also includes not printing unnecessary parentheses in expr_print and correctly implementing the parser expr_parse. Name the optional functions expr_eval_2, expr_print_2 and expr_parse_2. This is a hard problem; I do not expect you to solve it!