DIT027 – Assignment 2

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 problems to solve. Solving at least 18 of them is enough for passing the assignment.

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

Statement of contribution: Please include in the comments the information on which person from the pair had the leading role for different parts of the code.

The submission deadline is **8 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. Using the ++ and -- operators is is also not allowed.

Overcomplicated solutions will not be accepted.

Make sure that you follow ALL the instructions closely!

GOOD LUCK!

1. Download the module skeleton **assignment2.erl** from GUL (from Documents). You should be able to compile the module without modifications
2. **Basic recursion**
   These questions require writing simple recursive functions. Solutions for **A** and **D** are provided. You may need to define auxiliary functions.

   A. The function **sum_interval/2** when given two non-negative integers **N** and **M** computes the sum of all numbers between **N** and **M** (including **N** and **M**) if **N** =< **M**. When **N** > **M**, the function should return 0.

   Ex: sum_interval(3, 5) → 12

B. Write the function **mul_interval/2**, which given two non-negative integers **N** and **M** computes the product of all numbers between **N** and **M** (including **N** and **M**) if **N** =< **M**. When **N** > **M**, the function should return 1. Implement the solution using head recursion (not tail recursion), and without using compact forms.

Ex: mul_interval(3, 5) → 60

C. Implement the function **sum_sq_interval/2**, which given two non-negative integers **N** and **M** computes the sum of squares of all numbers between **N** and **M** (including **N** and **M**) if **N** =< **M**. When **N** > **M**, the function should return 0. Implement the solution using head recursion (not tail recursion), and without using compact forms.

Ex: sum_sq_interval(3, 5) → 50

D. The function **sum_interval_t/2**, behaves like **sum_interval/2**, but is implemented using tail recursion.

Ex: sum_interval_t(3, 5) → 12

E. Implement the function **mul_interval_t/2**, which behaves like **mul_interval/2**, but should be implemented using tail recursion.

Ex: mul_interval_t(3, 5) → 60

F. Implement the function **sum_sq_interval_t/2**, which behaves like **sum_sq_interval/2**, but should be implemented using tail recursion.

Ex: sum_sq_interval_t(3, 5) → 50

3. Recursion on lists

Solutions for **A** and **D** are provided.

A. The function **sum/1** when given a list of integers computes the sum of the list's elements.

Ex: sum([1, 3, 6]) → 10

B. Implement the function **mul/1**, which given a list of integers computes the product of the list's elements. Implement the solution using head recursion (not tail recursion).

Ex: mul([1, 3, 6]) → 18

C. Implement the function **sum_sq/1**, which given a list of integers computes the sum of squares of the list's elements. Implement the solution using head recursion (not tail recursion).

Ex: sum_sq([1, 3, 6]) → 46

D. The function **sum_t/1** has the same functionality as the function **sum/1**, bus is implemented using tail recursion.

Ex: sum_t([1, 3, 6]) → 10

E. Implement the function **mul_t/1**, which has the same behavior as **mul/1**, but is implemented using tail recursion.

Ex: mul_t([1, 3, 6]) → 18

F. Implement the function **sum_sq_t/1**, which has the same behavior as **sum_sq/1**, but is implemented using tail recursion.

Ex: sum_sq_t([1, 3, 6]) → 46

G. Implement the function **interval/2**, which given two non-negative integers **N** and **M** returns the list of consecutive numbers between **N** and **M** (including **N** and **M**) if **N** =< **M**. When **N** > **M**, the function should return [].

Ex: interval(3, 5) → [3, 4, 5]
interval(5, 3) → []

H. Implement the function **sum_interval_l/2**, which has the same behavior as **sum_interval/2**, but is making use of functions **sum/1** and **interval/2**.

Ex: sum_interval_l(3, 5) → 12

I. Implement the function **mul_interval_l/2**, which has the same behavior as **mul_interval/2**, but is making use of functions **mul/1** and **interval/2**.

Ex: mul_interval_l(3, 5) → 60

J. Implement the function **sum_sq_interval_l/2**, which has the same behavior as **sum_sq_interval/2**, but is making use of functions **sum_sq/1** and **interval/2**.

Ex: sum_sq_interval_l(3, 5) → 50

K. Implement the function **sum_sq_interval_l2/2**, which has the same behavior as **sum_sq_interval2/2**, but is making use of functions **sum/1** and **interval/2**, and of list comprehensions.

Ex: sum_sq_interval_l2(3, 5) → 50

L. Implement the function **concat_rev/2**, which takes two lists as its arguments and returns the reverse of the first list concatenated to the second list. The function should be tail-recursive.

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

M. Implement the function **reverse/1**, which reverses a list, making use of the **concat_rev/2** function.

Ex: reverse ([a, b, c]) → [c, b, a]

4. List comprehensions – In this problem you are supposed to implement the functions using list comprehensions.

A. The function **expand_circles/2** takes as arguments a positive integer **N**, and a list of pairs each consisting of the atom **circle** and a positive integer. The function returns a list of pairs of the same form as the list given as the second argument. The function's input represents circles of different radii. The function's output is a list of equal length as the input list, where each element is a circle from the input list whose radius has been enlarged **N** times.

Ex: expand_circles(2, [{circle, 4}, {circle, 2}, {circle, 5}])
→ [{circle, 8}, {circle, 4}, {circle, 10}]

B. Implement the function **print_circles/1**, which takes as argument a list of pairs each consisting of the atom **circle** and a positive integer. The function should print a line "Circle N" for each circle in the list, where N is the radius of the circle. Lines should be printed in the same order as the corresponding elements are in the list, and a trailing newline should also be printed. The function should return the atom **ok** after printing all lines.

Ex: print_circles([{circle, 3}, {circle, 2}, {circle, 4}])
should return **ok** and print the following lines:

Circle 3
Circle 2
Circle 4

C. Implement the function **even_fruit/1**, which takes as argument a list of pairs each consisting of an atom and a positive integer, and returns a list of atoms. The function's input represents different kinds of fruit and their quantities (numbers). The function's output should be a

list of atoms representing fruit, with an atom for each fruit that has an even quantity in the input list. Fruit appearing many times in the input list should be treated as separate. You should use your **even_odd/1** function from assignment 1.

Ex: even_fruit([{orange, 3}, {apple, 2}, {banana, 4},
{orange, 2}, {leechee, 5}, {apple, 6}])
→ [apple, banana, orange, apple]

D. You are operating a small ferry that can carry two vehicles. There is a number of vehicles to be transported. However, due to weight restrictions not all pairs may be transported together. Write a function **ferry_vehicles/2**, which takes as arguments an integer **N** and a list of pairs of the form **{Vehicle, Weight}**, where **Vehicle** describes the kind of the vehicle and **Weight** is a positive integer describing its weight. The vehicle descriptions in the list are all unique. The function's output should be a list of triples **{Vehicle1, Vehicle2, M}**, where **Vehicle1** and **Vehicle2** are vehicle descriptions from the input for two (different) vehicles, and **M** is their combined weight, which must be most **N**. All valid pairs of vehicles should be present in the output twice (also as **{Vehicle2, Vehicle1, M}**).

Ex: ferry_vehicles(3200, [{compact, 1500}, {crossover, 1900},
{mini, 1200}, {keicar, 800}, {minibus, 2200}])
→ [{compact,mini,2700}, {compact,keicar,2300},
{crossover,mini,3100}, {crossover,keicar,2700},
{mini,compact,2700}, {mini,crossover,3100},
{mini,keicar,2000}, {keicar,compact,2300},
{keicar,crossover,2700}, {keicar,mini,2000},
{keicar,minibus,3000}, {minibus,keicar,3000}]

(This is only one of possible results)

E. Write a function **ferry_vehicles2/2**, which takes the same arguments as **ferry_vehicles/2**, but returns a list of valid vehicle pairs (together with combined weights) without duplicate pairs. Hint: the Erlang's < operator can compare any two Erlang values, and is **asymmetric**, that is if V1 < V2 then not (V1 > V2).

Ex: ferry_vehicles(3200, [{compact, 1500}, {crossover, 1900},
{mini, 1200}, {keicar, 800}, {minibus, 2200}])
→ [{mini,compact,2700}, {mini,crossover,3100},
{mini,keicar,2000}, {keicar,compact,2300},
{keicar,crossover,2700}, {minibus,keicar,3000}]

(This is only one of possible results)

5. Recursion and side-effects

These exercises concern recursive functions that perform side-effects.

A. Function **print_0_n/1** when given a non-negative integer **N** prints numbers from **0** to **N** separated by newlines. The function returns the atom **ok**.

Ex: print_0_n(3) should return **ok** and print:

0
1
2
3

B. Implement the function **print_n_0/1**, which given a non-negative integer **N** prints numbers from **N** to **0** separated by newlines. The function should return the atom **ok**. Implement the solution using an auxiliary recursive function that starts the recursion from **0** and goes up.

Ex: print_n_0(3) should return **ok** and print:

3
2
1
0

C. Implement the function **print_0_n_0/1**, which given a non-negative integer **N** prints numbers from **0** to **N** followed by numbers from **N** to **0** separated by newlines. The function should return the atom **ok**. Implement the solution using a single auxiliary recursive function that starts the recursion from **0** and goes up.

Ex: print_0_n_0(2) should return **ok** and print:

0
1
2
2
1
0

D. Write a function **print_sum_0_n/1**, which given a non-negative integer **N** prints numbers from 0 to **N** separated by newlines, and also computes the sum of numbers from 0 to **N**. The function should return the sum of the numbers. The function could be implemented easily by running two separate recursive functions, but this task requires that all work is implemented as one recursive function (and possibly more non-recursive functions).

Ex: print_sum_0_n(3) should return **6** and print:

0
1
2
3