DIT027 – Assignment 1

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 7 problems to solve. Part 2.C is optional, and is not needed to pass the assignment.

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

The submission deadline is **5 September 2016** at 12:00.

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 are expected to use only features of Erlang that were covered so far in the lectures, so no case-expressions, if-expressions or lists.

You should also not use library functions, but define your own.

Overcomplicated solutions will not be accepted.

Make sure that you follow ALL the instructions closely!

GOOD LUCK!

1. Download the module skeleton **assignment1.erl** from GUL (from Documents). You should be able to compile the module without modifications

2. **Basic functions**

   A. Write a function **even_odd/1**, which given a non-negative integer returns the atom **even** if the argument is even and the atom **odd** when it is odd.

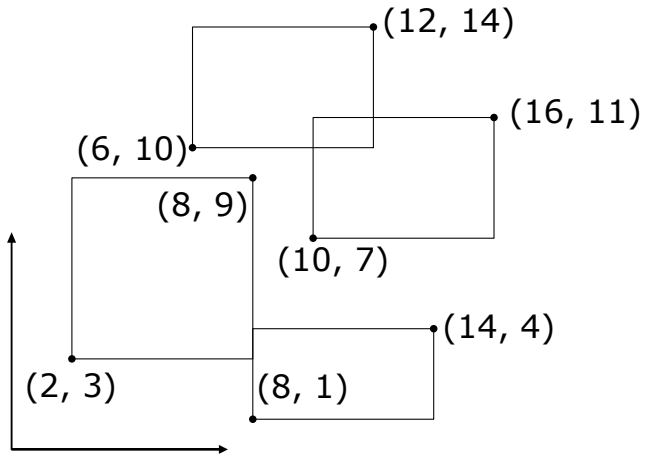
   Ex: even_odd(3) → odd
   even_odd(6) → even

   B. Write a function **range_overlap/2**, which given two ranges, represented as pairs of integers, computes their overlap. When the ranges overlap, the function should return a pair of the atom **overlap**

and the range, which is the common part of the argument ranges. When the ranges don't overlap, the atom **no_overlap** should be returned. If the ranges only share one point (one starts where the other ends), then the function should return **no_overlap**. The representation of ranges ensures that the second component of the pair is always larger than the first component. Builtin functions **min/2** and **max/2** might be useful for solving this question.

Ex: range_overlap({1, 4}, {3, 5}) → {overlap, {3, 4}}
range_overlap({4, 6}, {1, 2}) → no_overlap
range_overlap({2, 3}, {3, 6}) → no_overlap

C. (optional) Write the function **rect_overlap/2,** which given two rectangles computes their overlap. Rectangles are represented as triples **{rectangle, {X1, Y1}, {X2, Y2}}**, where **{X1, Y1}** and **{X2, Y2}** are the coordinates of the lower left and upper right corners of the rectangle. The origin of the coordinate system is in the lower left corner, and the **X** and **Y** values are non-negative integers. When the rectangles given to the function **rect_overlap/2** overlap, the function should return a pair of the atom **overlap** and the rectangle, which is the common part of the argument rectangles. When the rectangles don't overlap, the atom **no_overlap** should be returned. If the rectangles only 'touch' eachother (they share a single point or a line), then the function should return **no_overlap**. Builtin functions **min/2** and **max/2** might be useful for solving this question. The picture below shows rectangles used in the following examples.



Ex: rect_overlap({rect, {6, 10}, {12, 14}},
{rect, {10, 7}, {16,11}})
→ {overlap, {rect,{10, 10},{12, 11}}}
rect_overlap({rect, {10, 7}, {16, 11}},

{rect, {2, 3}, {8, 9}})
→ no_overlap
rect_overlap({rect, {2, 3}, {8, 9}},
{rect, {8, 1}, {14, 4}})
→ no_overlap

D. Write the function **get_amp/2**, that takes a description of an amplification circuit and returns its amplification factor. The circuit is represented as a tuple **{amplifier, PreAmp, Factor, Noise}** where **PreAmp** is a description of the pre-amplifier, **Factor** is a non-negative integer describing the amplification factor (in dB), and **Noise** is a non-negative integer describing the noise introduced by the amplifier. The pre-amplifier is represented either by the atom **no_preamp** indicating no pre-amplifier, or by the tuple **{preamp, Factor}**, where **Factor** describes its amplification factor as previously. The function should return the sum of the amplification factors of the amplifier and of the pre-amplifier, or just the amplification factor of the amplifier if there is no pre-amplifier. Hint: you can define auxiliary functions to solve this (and other) questions.

Ex: get_amp({amplifier, no_preamp, 12, 2}) → 12
get_amp({amplifier, {preamp, 4}, 14, 3}) → 18
get_amp({amplifier, {preamp, 5}, 12, 5}) → 17

E. Write functions **f2c(Fahrenheit)**and **c2f(Celsius)**, which convert between Fahrenheit and Celsius temperature

Hint: 5 * (F - 32) = 9 * C

Ex: f2c(4) → -15.5555555
f2c(45) → 7.22222222
c2f(-4) → 24.8

F. Write the function **convert(Temperature)**, which combines the functionality of f2c and c2f. A usage example follows:

Ex: convert({f, 4}) → {c, -15.5555555}
convert({c, -4}) → {f, 24.8}

3. Non-linear patterns

A. The module contains **measure/3**, which computes a measurement from three integers based on some rule. The function uses guards (the

when X construction) to check if the values of two arguments are the same. Modify the function so that its functionality stays the same, but the modified function uses non-linear patterns (patterns where the same variable occurs twice) instead of guards.

Ex: measure(2, 5, 5) → 3

B. Implement the function **any_2_equal/3**, which takes 3 arguments and returns **true** if any two of them are equal. Otherwise it returns **false**. You should use non-linear patterns instead of guards to implement this function.

Ex: any_2_equal(a, a, 3) → true
any_2_equal(2, e, 2) → true
any_2_equal(b, 3, 1) → false
any_2_equal(7, 7, 7) → true