CPDS-Conc Lab 5
Basics on Erlang

Authors: Jorge Castro and Joaquim Gabarro

Goal: In the *training* part you should acquire some experience in Erlang. As a *homework* you have to provide an Erlang implementation of (1) the *mergesort* algorithm and (2) a toy example based on the token ring architecture.

Basic material:

- Course slides.
- Modules 2 & 3 in https://www.erlang.org/course
- *Programming Erlang: Software for a Concurrent World*, Joe Armstrong, 2007. Second edition 2013.
- Chapter 5 of the book *Erlang Programming*, Francesco Cesarini and Simon Thompson, O'Reilly, 2009.

3.1 Training in Erlang

1. *Pythagorean Triplets.* Pythagorean triplets are sets of integers `A,B,C` such that `A**2 + B**2 = C**2`. The function `pyth(N)` generates a list of all integers `A,B,C` such that `A**2 + B**2 = C**2` and where the sum of the sides is equal to or less than `N`.

   ```
   Pyth = fun(N) ->
              [{A,B,C} || A <- lists:seq(1,N), B <- ..., C<- ..., A+B+C =< ...,   ... == ... ]
          end.
   ```

   A possible behavior is:

   ```
   > Pyth(3).
   [].
   > Pyth(11).
   [].
   > Pyth(12).
   [{3,4,5},{4,3,5}]
   ```

2. *Erathostenes Sieve.* We develop a program that computes prime numbers known as the Primes Sieve of Eratosthenes, after the Greek mathematician who developed it. The algorithm to determine all the primes between 2 and n proceeds as follows . First, write down a list of all the numbers between 2 and n :

   `2 3 4 5 6 7 ... n`

   Then, starting with the first uncrossed-out number in the list, 2, cross out each number in the list which is a multiple of 2:

   `2 3 ~~4~~ 5 ~~6~~ 7... n`

   Now move to the next uncrossed-out number, 3, and repeat the above by crossing out multiples of 3. Repeat the procedure until the end of the list is reached. When finished, all the uncrossed-out numbers are primes.

   Design a `module(siv)` containing the functions `range` , `remove_multiples`, `sieve` and `primes`. Proceed step by step.

   - First design a function `range(Min, Max)` returning a list of the integers between `Min` and `Max`.

     ```
     -module(siv).
     -compile(export_all).

     range(N, N) -> [N];
     range(Min, Max) -> [Min | range(..., Max)].
     ```

     For instance:

     ```
     > siv:range(1,15).
     [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
     ```

   - Add a function `remove_multiples(N, L)` removing all multiples of `N` from the list `L`

     ```
     remove_multiples(N, [H|T]) when H rem N == 0 -> remove_multiples(... , ...);
     remove_multiples(N, [H|T]) -> [H | ... ];
     remove_multiples(_, []) -> [].
     ```

     For instance:

     ```
     > siv:remove_multiples(3,[1,2,3,4,5,6,7,8,9,10]).
     [1,2,4,5,7,8,10]
     ```

   - Finally the function `sieve(L)` retains the head of the list `L`, removes all multiples of the head of the list and recursively sieves this list.

     ```
     sieve([H|T]) -> [H | ...(...(H, T))];
     sieve([]) -> [].
     ```

     Using `sieve(L)`, the primes can be computed as:

     ```
     primes(Max) -> sieve(...(2, ...)).
     ```

     For instance:

     ```
     > siv:primes(25).
     [2,3,5,7,11,13,17,19,23]
     ```

3.2 Homework

3.2.1 Mergesort

We ask to develop two versions of the merge sort (one sequential and the other concurrent). We will sort floating-point numbers in order to avoid printing problems [1]. Below we use the list :

```
L = [27.0, 82.0, 43.0, 15.0, 10.0, 38.0, 9.0, 8.0].
```

Complete a `msort` module containing functions defined below:

1. Program a function `sep(L, N)` returning {`L1, L2`} so that `L1++L2 == L` and `length(L1) == N`. You can assume that nonnegative integer N is at most the lenght of the list L. For instance:

   ```
   32> msort:sep(L, 3).
   {[27.0,82.0,43.0],[15.0,10.0,38.0,9.0,3.0]}
   ```

   Hint:

   ```
   sep(L,0) -> {[], L};
   sep([H|T], N) -> {Lleft, Lright} = sep(.....),
                    ......
   ```

2. Program a function `merge(L1, L2)` returning the merge of two sorted lists, for instance:

   ```
   34> L1= [27.0, 43.0, 82.0].
   ...
   35> L2= [3.0, 9.0, 10.0, 15.0, 38.0].
   ...
   36> msort:merge(L1,L2).
   [3.0,9.0,10.0,15.0,27.0,38.0,43.0,82.0]
   ```

3. Complete the following sequential version of the merge sort function `ms(L)` returning `L` sorted.

   ```
   ms([]) -> ...;
   ms([A]) ->...];
   ms(L) ->
        {L1, L2} = sep(..., ....div 2),
        ....
        ....
   ```

[1] From Armstrong book (pag 29) we read: Remind: When the shell prints the value of a list it prints the list as a string, but only if all the integers in the list represent printable values. Given `L= [83, 117, 114, 112, 114, 105, 115, 101]`, if we ask execute `L` we get the string `"Surprise"`.

4. Design a parallel version `pms` of the merge sort filling in the following lines:

```
call(Pid) -> receive
                {Pid, L} -> L
             end.

pms(L) -> Pid = spawn(mergesort, p_ms, [self(), L]),
          call(Pid).

p_ms(Pid, L) when length(L) < 100 -> Pid ! {self(), ms(L)};
p_ms(Pid, L) -> {Lleft, Lright} = sep(.....),
                Pid1 = .............,
                Pid2 = .............,
                L1 = call(Pid1),
                L2 = call(Pid2),
                .... ! .........
```

3.2.2 Token Ring Architecture

A "control" process has to offer a user function `go(N,M)` that generates a lists `L` of `M` random integer numbers in the set {`1,2,...,M`}, sets up ring of `N` processes (so-called "workers") and sends a token to the first worker. Token possesion grants the worker possesor permission to eat. When worker `k` receives a token, it sends a message {`self(), eat`} to control and sends the token to next worker.

When control receives a message {`Pid, eat`}, it withdraws the head `H` of the list `L` and appends to a result list the tuple {`Pid, H`}. When list `L` is empty, control sends a stop message to the ring that terminates the workers and prints the result list.

There are two ways to solve this problem:

1. control spawns all the workers in the ring.
2. control spawns only the first worker in the ring. Every new worker in the ring, but the last one, spawns the next worker.

Following the first solution idea, we describe below control and worker modules. The worker module is complete and cannot be changed. The control module is only sketched. You have to complete this module.

Avoid unnecesary synchronization between processes (it is desired a *concurrent* solution). Pay attention to border cases, for instance `N = 1`, etc..., and be sure that all the processes end in a convenient way. Run several consecutive times your code in order to check everything is ok.

Execution examples:

```
11> control:go(3,7).
[5,4,1,2,2,1,1]
<0.68.0> eats
<0.69.0> eats
<0.70.0> eats
```

```
<0.68.0> eats
<0.69.0> eats
<0.70.0> eats
<0.68.0> eats
[{<0.68.0>,1},{<0.70.0>,1},{<0.69.0>,2},{<0.68.0>,2},{<0.70.0>,1},{<0.69.0>,4},{<0.68.0>,5}
stop
12> control:go(1,4).
[4,2,1,1]
<0.72.0> eats
<0.72.0> eats
<0.72.0> eats
<0.72.0> eats
[{<0.72.0>,1},{<0.72.0>,1},{<0.72.0>,2},{<0.72.0>,4}]
stop
13> control:go(4,6).
[1,4,4,1,2,1]
<0.74.0> eats
<0.75.0> eats
<0.76.0> eats
<0.77.0> eats
<0.74.0> eats
<0.75.0> eats
[{<0.75.0>,1},{<0.74.0>,2},{<0.77.0>,1},{<0.76.0>,4},{<0.75.0>,4},{<0.74.0>,1}]
stop
```

Worker module:

```
-module(worker).
-export([worker_node/1]).


worker_node(PidControl) ->
    receive
        PidNext -> dowork(PidControl, PidNext)
    end.


dowork(PidControl, PidNext) ->
    receive
        stop -> PidNext ! stop;
        token ->
                 PidControl ! {self(), eat},
                 PidNext ! token,
                 dowork(PidControl, PidNext)
    end.
```

Suggested control module (just sketched!):

```
-module(control).
-export([go/2]).

% N is the number of ring processes, N >= 1
% M is the range of targets
% flush the mailbox to erase obsolet info,
% creates the  worker ring and starts the game
%
go(N, M) -> flush_mailbox(),
            TargetList = generate(M),
            io:format("~p~n", [TargetList]),
            [FirstWorker|_]  = worker_ring(N, self()),
            FirstWorker ! token,
            ResultList = controlgame(TargetList, []),
            io:format("~w~n", [ResultList]),
            FirstWorker ! stop.


%generates a list of M random numbers in rtrange 1..M
generate(0) -> [];
generate(M) -> [rand:uniform(M)|generate(M-1)].




controlgame([], ResultList) -> ResultList;
controlgame(TargetList, ResultList) ->  .......
                                        .......

flush_mailbox() -> receive
                       _Any -> flush_mailbox()
                   after 0 -> ok
                   end.


%
%
% Some auxiliar functions may be necessary
%
%

..........
..........


% sets up a ring of N workers, each one running  worke_node function from  worker module
```

```
worker_ring(N, PidControl) -> .....
                              .....
```

Have fun!