CPDS-Conc Lab 4
Peterson exclusion algorithm
& Basics on Erlang

Authors: Jorge Castro and Joaquim Gabarro
Comments and suggestions to `gabarro@lsi.upc.edu` with header `CPDS-ConcLab` .

Goal: In the *training* part you should acquire some experience in Erlang. As a *homework* you have to model, check and implement in Java the Peterson algorithm as an example of a basic mutual exclusion algorithm. Moreover, an Erlang implementation of the *mergesort* algorithm has to be developed.

Basic material:

- Course slides.
- Basic reading: Chapters 7 of the book: *Concurrency, State Models & Java Programs*, Jeff Magee and Jeff Kramer, Wiley, Second Edition, 2006 (M&K for short).
- *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(N) ->
    [ {A,B,C} || A <- lists:seq(1,N), B <- ..., C<- ..., A+B+C =< ...,   ... == ... ].
```

   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` and recursively removes all multiples of the head of the list from the sieved tail of the 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 Peterson's exclusion algorithm

1. *M&K 7.7, Peterson's Algorithm for two processes (Peterson, G.L., 1981).* Fortunately for the warring neighbors, Gary Peterson visits one day and explains his algorithm to them. He explains that, in addition to the flags, the neighbors must share a turn indicator which can take the values 1 or 2. This is used to avoid potential deadlock.

   When one neighbor wants to enter the field, he raises his flag and sets the turn indicator to indicate his neighbor turn. If he sees his neighbor's flag and it is his neighbor's turn, he can not enter but must try again later. Otherwise, he can enter the field and pick berries and must lower his flag after leaving the field.

   For instance, neighbor `n1` behaves as shown below, and neighbor `n2` behaves symmetrically.

   ```
   while (true) {
      flag1 = true; turn = 2;
      while (flag2 and turn==2) {};
      enterField; pickBerries;
      flag1 = false;
   }
   ```

   Solve the following questions:

   (a) Model Peterson's Algorithm for the two neighbors and provide its FSP description. Check that it does indeed avoid deadlock and satisfy the mutual exclusion (safety) and berry-picking (progress) properties, even in adverse conditions as when both neighbors are greedy.

   Hint : The following FSP can be used to model the turn indicator.

   ```
   set CardActions = {set1,set2,[1],[2]}

   CARDVAR = VAL[1],
   VAL[i:Card] = ( set1 -> VAL[1]  | set2 -> VAL[2] | [i] -> VAL[i] ).
   ```

   (b) Implement in Java the warring neighbors system using the Peterson's protocol. Check that this program version avoid the progress problems present in the previous week code version when neighbors are greedy (code corresponding to exercise 7.6 of M&K).

3.2.2 Basic Erlang

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:

(a) 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(.....),
                 ......
```

(b) 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]
```

(c) 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),
     ....
     ....
```

(d) Design a parallel version `pms` of the merge sort along the lines suggested going from `qs` to `pqs`.

[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"`.

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

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

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