LINFO1131 Practical Exercises
Lab 8: Learn some Erlang

This week we are going to refresh your Erlang skills, and dig a bit into it's special message passing semantics.

Installation

To install Erlang, use your preferred package manager or go to http://www.erlang.org/downloads.

There are several ways to run a Erlang program. Ensure that you have a working installation by compiling the following code and executing the function `hello:hello_world()` in the Erlang shell.

```
-module(hello).
-export([hello_world/0]).

hello_world() ->
    io:fwrite("Hello, World!\n").
```

You may find more information about running Erlang functions and programs in the first part of last year's introduction.

Getting your feet wet

1. Improve the hello world example by making a server that prints `"Hello World!"` when it receives the message `print`. You will need to use `spawn(mod_name, fun_name, [])` to start the server. This returns the `PID` of the process.
2. Test that it works by sending `print` messages to the server with `PID ! print`.
3. If it is not the case yet, write a `serve_hello/0` function that starts the server for you.
4. Send some random messages to the server, and then another `print` message. What happens?
5. Inspect the server mailbox with `erlang:process_info(PID, messages)`. What do you see?
6. Fix your implementation to silently discard useless messages from the mailbox. Is this a good idea in general?
7. Accept a `stop` message that gracefully stops the server. Does it work as intended? What happens if you send more `print` messages after the shutdown?

Rock, paper, scissors!

In this game, we will implement the classical *rocks, paper, scissors* game in Erlang. We provide some boilerplate functions to get you started.

```
-module(ppc).
-export().
-define(MOVES, [pierre, papier, ciseaux]).

random_move() ->
    lists:nth(rand:uniform(length(?MOVES)), ?MOVES).

play(pierre, pierre) -> tie;
play(pierre, papier) -> lost;
play(pierre, ciseaux) -> won;
play(papier, pierre) -> won;
play(papier, papier) -> tie;
play(papier, ciseaux) -> lost;
play(ciseaux, pierre) -> lost;
play(ciseaux, papier) -> won;
play(ciseaux, ciseaux) -> tie.
```

1. Write a basic server that receives `{play, Pid, Move}` commands and replies with a `{you, [win,lose,tie]}` according to the game results with a random move computed internally. Test that it works by sending several moves and observing the results. You can send messages with `Server ! {play, self(), pierre}`.

2. In the previous setup, there is no real opponent. Change your server to wait for two moves before computing the result. How do you reply to the two players ? Did you use some sort of state to remember them?

3. It is a bit annoying tho read the messages in your mailbox while sending other ones. You can use `erlang:process_info(self(), messages)` but that does not actually consume the message. Write a small process that prints every message it receives to the standard output. How can you redirect the game server reply to your printer process? Check that message are no more coming to `self()` with `process_info`.

4. With you implementation, a player can play against himself. Change the server so that it waits for moves from two different processes before computing the winner.

5. What happens if you send multiple messages at once from the same process? Remember to use `process_info` to inspect the message queues if needed.

6. Implement a random player that continuously attempts to play a random move on the server. Ensure that it does not spam the server.

7. Test your code by sending several moves to the server. Do you observe random results? Can you guess what move was played by the random player by looking at the result?