Multicore Programming

Lab 2: Introduction to Erlang

Assistant: Janwillem Swalens
Mail: jswalens@vub.ac.be
Office: 10F719

19 February 2014

The assignment will introduce you to using the Erlang debugger, pattern matching, lists processing, control structures, records, and hot-swapping of code. To get started, all you need is a basic idea of how to read the syntax presented in the lecture.

By next Monday (at the latest at 23:59), send in your solutions to the exercises in section 2 by mailing them to `jswalens@vub.ac.be`.

1 Preparation

Before we start with the interesting part, we will get in touch with the infrastructure for the lab sessions and write a short "Hello World".

1.1 Infrastructure

1. Download the material for this lab from Pointcarré.
2. (Optional) Import the project into your Eclipse environment. The Eclipse plugin is supposed to provide debugging support and a REPL (read-eval-print loop), however, the integration is not perfect. TextMate, Vim, or Emacs are also good enough for our purposes.
3. Have a look at the included `Makefile` and `run.sh`. They will be used for convenience throughout the following lab sessions.

1.2 Execute Erlang Code in the Terminal

This assignment sheet will rely on the command line as found on the Mac computers in the computer room. To open a terminal/command line find the 'Applications' icon in the dock (typically at the bottom of the screen), navigate to the Utilities folder and execute `Terminal.app`.

Change the directory to the `src` folder, which is part of the archive downloaded from Pointcarré. In this folder, you can build the Erlang code by executing '`make`'. Similar to other languages such as C or Java, the Erlang code needs to be compiled after every change to get the desired result. Code can be executed either via the script `run.sh` (see the Hello World example), or via an interactive Erlang shell that can be started by executing `erl`.

1. Open the terminal, execute the command `erl` to start the Read-Eval-Print loop.
2. Play around, try the following commands (don't forget the dot at the end of each command):
   - `2 + 3.`
   - `[1, 2, 3].` to create a list.
   - `lists:sort([1, 4, 3, 2]).` to use the sort function in the lists module.
   - `lists:reverse([1, 4, 3, 2]).`
   - `lists:map(fun(X) -> 2*X end, [1, 2, 3]).`
   - Find more interesting functions at http://erldocs.com, particularly in the modules: lists, math, random, io, dict, sets.

1.3 Hello World

`hello_world.erl`

1. Create a file `hello_world.erl` and add the content of listing 1.
   Remark: Certain PDF readers do not handle copy/paste well, and insert additional whitespace, which can pose a problem.
2. Use '`make`' and '`./run.sh hello_world start`' to compile and execute it.

Listing 1: hello_world.erl

```
-module(hello_world).

%% Exported Functions
-export([start/0]).

%% External functions
start() ->
    io:fwrite("Hello World!~n").
```

1.4 Debugging

`debugging.erl`

Erlang includes a debugger, which we will explore in this task.

1. Start `erl` in the terminal.
2. Compile the debugging module: `c(debugging, debug_info).`
   This will make sure the module contains debug information. Don't forget the dot at the end of the command!
3. Start the debugger: `debugger:start().` (again, including the dot).
4. Load the module in the debugger Menu → Module → Interpret... → select the `debugging.erl` → Choose.
5. In the debugger window activate Auto Attach → On Break.
6. Double click on the debugging module.
7. Double click next to the line number 9 to create a break point.
8. In the REPL, execute `debugging:start().`
9. Explore the capabilities of the debugger a bit.

1.5 EUnit

EUnit is a unit testing framework for Erlang. It is similar to other common unit testing frameworks. In EUnit, tests are written as part of the modules themselves. The framework generates a `test/0` method for the module, which then can be called in the interactive Erlang shell or via `./run.sh`. The generated `test/0` will execute all methods ending with `_test` and `_test_` and report the test results.

EUnit features the typical `?assert(Expr)` to make sure that an expression is true. It also supports `?assertNot(...)`, `?assertMatch(...)`, `?assertEqual(...)` and similar assertions.

As simple unit tests looks like this:

```
simple_test() ->
    ?assert(1 + 1 =:= 2).
```

In contrast to other test frameworks, it also provides the notion of test generating functions, i.e. functions that generate unit tests. When a test function's name ends with `test_`, it is supposed to return an anonymous function, or a list of anonymous functions, which are then executed as unit tests. To facilitate this type of unit tests, EUnit provides macros that start with an underscore, for instance `?_assert(...)`.

For example, in exercise 2.1, the following test generating function is defined:

```
simple_match_test_() ->
    [?_assert({42, someAtom} == simple_match(1)),
     ?_assert({returnValue,simple_match(1)} == simple_match(2))].
```

The function `simple_match_test_()` returns a list of two tests. The EUnit framework will execute both tests and report the result.

1.6 Links and Literature

- Erlang documentation: http://www.erlang.org/doc/
- Erlang course: http://www.erlang.org/course/course.html
- Alternative Erlang documentation (easier to search): http://erldocs.com
- Google usually helps best with a query like: `erlang man $searchTerm`
- Concurrent Programming in Erlang by J. Armstrong, R. Virding, C. Wikström, M. Williams: http://www.erlang.org/download/erlang-book-part1.pdf
- EUnit manual: http://www.erlang.org/doc/apps/eunit/chapter.html

2 Exercises

For all exercises, you can use `make` to compile the Erlang source files (`*.erl`) with `erlc`. The `./run.sh` script provides you with a simple way to test your Erlang code on the command line. As mentioned before, a method `test/0` is generated. It can be called with:

```
./run.sh $module_name test
```

Most of the exercises involve reading some unit tests and answering questions or fixing the test cases.

2.1 Pattern Matching

`pattern_matching.erl`

Read the code and find out how it uses pattern matching. There are three small mistakes which have to be corrected. One of the mistakes is actually in the test code.

Might be helpful: `erlang:display/1`

Tasks:

- Take some time to understand what is going.
- Fix the errors reported by EUnit.
- Which of the lines of Erlang code are utilizing pattern matching?

2.2 Lists and Pattern Matching

`pm_lists.erl`

Tasks:

- Have a look at part 1 of the file and implement your own `nth/2` function, which returns the n^{th} element. (See http://erlang.org/doc/man/lists.html)

- Test your function with unit tests for the corner cases.
- Fix part 2 of this file. The unit tests are correct for this task.
- Give an explanation for the reason of the observed behavior.

2.3 Control Structures

`control.erl`

This example should make you familiar with Erlang's basic control structures.
Tasks:

- Fix `case_statement(Value)` to satisfy the test case.
- Answer the questions for the `if` construct, and fix the problem.
  Tip 1: take a look at http://erlang.org/doc/reference_manual/expressions.html#id77482 for the definition of >=.
  Tip 2: `is_number` might be helpful.
- The exception example is only informative.

2.4 Records

`records.erl`

Erlang also supports a syntax for records. It is explained in more detail at http://www.erlang.org/doc/reference_manual/records.html.
Tasks:

- The address is missing a field for a city. That's a major bug. Fix it and also make sure that the test cases check for a city. (Uncomment the corresponding line in the test.) Insert the city field right after the field for the street.
- Discuss the usefulness of this syntax feature. Why do you/do you not want to use it?

Note: the next version of Erlang, R17, introduces maps as a replacement for records[1].

2.5 Hot Code Swapping

`swap.erl, swap.v5.erl`

Erlang was designed to run on telecommunication systems which need an extraordinary reliability. Thus, the runtime supports patching a running system.
Tasks:

[1] http://joearms.github.io/2014/02/01/big-changes-to-erlang.html

- `swap.erl` contains a small interactive shell application that reads input from standard input. It should display the input and an identifier for the current code version. `swap.v5.erl` contains a new version of this program.
- Use Erlang's mechanisms for code loading (see module `code`) after processing the input to replace the currently executed code by a new version. Test it as follows:
  1. Compile and run version 3. Keep it running.
  2. Rename `swap.erl` to `swap.v3.erl`, and `swap.v5.erl` to `swap.erl`.
  3. Compile the new `swap.erl` (version 5).
  4. Write `reload` in the running program (version 3) to reload the `swap` module, you should now be running version 5.

3 Snakes, Actors, and Artificial Intelligences

In case you are already familiar with Erlang, please try to solve the following assignment. Furthermore, it would be beneficial to approach the task using pair programming.

In the directory `snake` is a simple implementation of the classic snake game. A snake is moving on a 10×10 field and grows when it eats apples. The game ends when the snake bites itself. It already comes with a simple AI, but all snakes are on their own field. New features you could add:

- Extend the game so multiple players/AIs can play on the same field. Currently, each player is on a separate field.
- Each player should be represented by an actor.
- The game is turn-based (with each turn lasting 500 ms). During each turn the players can submit the new direction in which they want to move (perhaps multiple times, only the last one is taken into account).
- There should be a timer/heartbeat actor. After the game master gets a time signal, the result of the next move is computed and the new board configuration is made available. The game master checks for collisions and eaten apples.
- The players should always have access to the latest board, i.e. the status after the last completed turn.

Start by opening all files and determining their function. To understand what each file does, read the comments at the top of the file. You might want to draw a diagram to understand the relations between the different actors.

We will continue this exercise next week.