Assigments week 4

Joe Armstrong

January 21, 2014

Contents

1 Introduction

These problems are for lectures F2 and F3 held in week 4 of 2014. The completed assignments should be submitting to our ilearn2-server. *before* lecture F5.

Additional material can be found in Chapters 1..6 of the course book *Programming Erlang, 2'nd edition.*

2 How to test your code

We will use TDD[1] to develop the code and to decide when to stop working on the code. Here's what you have to do:

[1] Test Driven Development.

1. Make sure Erlang runs on your machine.
2. Download the file `week4_problems.erl` from my github repository `http://joearms.github.com/paradis/src/week4_problems.erl` or from the `ilearn2` site.
3. Implement the module `week4_solutions.erl` (and any modules that `week4_solutions` calls).
4. Compile `week4_problems.erl`, `week4_solutions.erl` and any other code you need.
5. Evaluate the function `week4_problems:test_easy()`. If this evaluates to `horray` your program has passed its unit test and will be graded as *Godkänd*.
6. To get *Väl Godkänd* you should also pass the tests in the function `week4_problems:test_hard()`.
7. Submit you solution on the `ilearn2` site.

3 Problems

All your code should be in the module `week4_solutions.erl` it should export the functions `factorial/1`, `rotate/2` etc. (The notation `F/N` means the function `F` with `N` arguments). The functions that you have to implement are those which are called from `week4_problems.erl`. Each function is described in a separate subsection below:

3.1 factorial(N)

Compute `factorial(N)` where `N` is an integer.

Factorial `N` is defined as `N * N-1 * N-2 * ... * 1`

For example `factorial(5) = 5 * 4 * 3 * 2 * 1` which is `120`

Note: As you can see, the test function in `week4_solutions.erl`[2] starts:

```
test() ->
    120 = week4_solutions:factorial(5),
    ...
```

This is a unit test that tests that you can correctly compute `factorial(5)`.

[2]The listing is at the end of this document.

3.2 rotate(N, L)

`rotate(1, L)` rotates the list `N` by one place. For example a one place rotation of the list `[a,b,c,d,e,f]` is the list `[b,c,d,e,f,a]`. The element at the head of the list `a` is moved to the end of the list.

`rotate(N, L)` for positive `N` performs a one place rotation `N` times.

`rotate(-1,L)` rotates the list in the opposite direction and moves the last element of the list to the beginning so `rotate(-1, [a,b,c,d,e,f])` is the list `[f,a,b,c,d,e]`.

`rotate(N, L)` for negative `N` performs `rotate(-1, L)` `N` times.

3.3 count_atoms(A, X)

Returns the number of Erlang atoms `X` in the data structure `X`.

In the test case we parse the program `week4_problems.erl`. The module `epp` exports `parse_file` which returns the parse tree of the program. The parse tree is dumped into the file `debug` so you can see what it looks like. The first time you run the program you should take a look in the file and see what the parse tree of the program looks like.

Having dumped the parse tree, take a look at the parse tree and see if you can figure out what the internal representation of an atom is. Then write the function `count(A, X)` to count the number of occurrences of the atom `A` in the data structure `X`.

Hint: Use the BIF[3] `tuple_to_list`

3.4 expand_markup(Str)

`expand_markup(Str)` expands a simple markup language into HTML. "`**`" is a bold toggle. The first occurrence of `**` in the string is replaced by `<b>`, the second by `<\b>` and so on.

Thus `expand_markup("**BB**")` expands to `<b>BB</b>`.

"`__`" toggles an italic flag. So `expand_markup("__II__")` evaluates to `<i>II</i>`.

HTML fragments should be "well nested" so `<b>..<i>..</i>..</b>` is well nested, but `<b>..<i>..</b>..</i>` is badly nested.

`expand_markup` should always return well nested HTML. So, for example, `expand_markup("**B__I**C__")` should expand to
`<b>B<i>I</i></b><i>C</i>`.

Note: Designing markup languages might look simple, but they have subtleties that are often missed. The last example is an example of this.

Getting `expand_markup` right for all inputs is difficult. Don't worry if this exercise takes you a few hours to solve.

[3] Built in function

If you fail to solve the tricky case comment out the test in the unit test program and do the other exercises.

4 A successful testrun

Here's what it looks like if everything goes right:

```
$ erl
Erlang R16B (erts-5.10.1) ...
Eshell V5.10.1  (abort with ^G) ...
1> c(week4_problems).
{ok,week4_problems}
2> c(week4_solutions).
{ok,week4_solutions}
3> week4_problems:test_easy().
horray
4> week4_problems:test_hard().
written:"debug"
whoopy
```

5 Hints

- Print small data structures in the shell with `io:format("~p", [X])`.
- Use `week4_problems:dump/2` to pretty print large data structures.
- Make sure all your Erlang programs are in the same directory. If you code is in different directories you might run into problems with code paths.
- To compile all program in a directory give the shell command: `erlc *.erl` (OS-X and Linux only, not windows).

6 The test program

```
-module(week4_problems).
-export([test_easy/0, test_hard/0, dump/2]).

test_easy() ->
    M = week4_solutions,
    120 = M:factorial(5),
    L = [a,b,c,d,e,f],
    [b,c,d,e,f,a] = M:rotate(1,L),
    [f,a,b,c,d,e] = M:rotate(-1,L),
    [d,e,f,a,b,c] = M:rotate(3, L),
    [e,f,a,b,c,d] = M:rotate(-2, L),
    [b,c,d,e,f,a] = M:rotate(7,L),
    [b,c,d,e,f,a] = M:rotate(10000000000000000*6+1,L),
    "<b>BB</b>"   = M:expand_markup("**BB**"),
    "<b>BB</b><i>II</i>"   = M:expand_markup("**BB**__II__"),
    "<b>BB<i>II</i>CC</b>" = M:expand_markup("**BB__II__CC**"),
    horray.

test_hard() ->
    "<b>BB<i>II</i></b><i>CC</i>" =
        week4_solutions:expand_markup("**BB__II**CC__"),
    test_count_atoms(),
    whoopy.


test_count_atoms() ->
    X = epp:parse_file("week4_problems.erl",[],[]),
    dump("debug", X),
    8 = week4_solutions:count_atoms(a, X).

dump(File, X) ->
    {ok, S} = file:open(File, [write]),
    io:format(S, "~p~n", [X]),
    file:close(S),
    io:format("written:~p~n",[File]).
```