Erlang assignment 1

Parallel and distributed programming

Isak Samsten

Spring, 2025

Introduction

Introductory assignments for sequential Erlang. This week has three assignments to test your knowledge of the Erlang basics, i.e., its syntax, semantics and functional programming. The tasks are awarded a maximum of 20 points. The grades are distributed according to Table 1.

Table 1: Point to grade conversion table

Point	**Grade**
0 – 8	F
9 – 10	E
11 – 12	D
13 – 15	C
16 – 17	B
18 – 20	A

Submission

Please submit your solutions as `task1.erl`. Make sure you submit a single file and that your module-exports are exactly as stated in the task description.

Oral presentation

You are required to deliver an oral presentation of your solutions at Presentation 1.

Problems

Problem 1 (4 points)

Write a function `eval/1` which takes as input a tuple and evaluates the mathematical expression it denotes. For instance, the call `eval({add, 1, 1})` would return `{ok, 2}`, and the call `eval({mul, {add, 2, 2}, 4})` would return `{ok, 16}`[1]. More generally, the function accepts as input an expression tuple `E` of three (3) elements `{Op, E1, E2}`, where `Op` is `add`, `mul`, `'div'`[2] or `sub` and `E1` and `E2` are either numbers or expression tuples (see example), and return the answer as the tuple `{ok, Value}`, or the atom `error` if the evaluation fails for any reason.

Implement the function `eval/1` in the module `task1` and export it.

Example 1

```
1> eval({add, 1, 2}).
{ok, 3}
2> eval({add, 1, x}).
error
3> eval({mul, 2, {mul, 1, 2}})
{ok, 4}
```

[1] Note the evaluation order is from left to right, i.e., $(2 + 2) * 4$

[2] Note that `div` is a reserved keyword so to make it an atom you have to surround it with `'div'`

Problem 2 (5 points)

Write a function `eval/2` which is functionally equivalent to `eval/1`, but accepts as its second argument a `map` which maps `atoms` to numbers. For instance, the call `eval({add, a, b}, #{a => 1, b => 2})` return 3 and the call `eval({mul, {add, a, 3}, b}, #{a => 1, b => 2})` return `{ ok, 8}`[3]. More generally, `eval(E, L)` accepts as input an expression tuple `E` of three elements `{Op, E1, E2}` where `Op` is defined in Task 1 and `E1` and `E2` is either a number, **atom** or an expression tuple, and an Erlang map `L` that acts as lookup table for atoms. The function returns either `{ok, Value }` or `{error, Reason}`, where `Reason` is either `variable_not_found` if an atom does not exist in the lookup table or `unknown_error`.

Implement the function `eval/2` in the module `task1` and export it.

Example 2

```
1> eval({add, 1, 2}, #{}).
{ok, 3}
2> eval({add, a, b}, #{a=1}).
{error, variable_not_found}
3> eval({add, {add, a, b}, {add, 1, 2}}, #{a=>2, b=>3}).
{ok, 8}
```

[3] $(1 + 3) * 2$

Problem 3 (11 points)

Implement the higher-order functions in Table **using tail recursion** but **without using list-comperhensions or the `lists`-module**[4]. Ensure that the functions preserve the order of elements.

Function	Definition
`map(F, L)`	Return a new list which is the result of applying the function `F` to every element in `L`. Awarded a maximum of **2 points**.
`filter(P, L)`	Return a new list which is the result of filtering out the elements in `L` for which the function `P` returns true. Awarded a maximum of **2 points**.
`split(P, L)`	Return a tuple with two lists, `{True, False}` where `True` is a list containing the elements of `L` for which `P` returns true and `False` is a list containing the elements of `L` for which `P` returns `false`. Awarded a maximum of **3 points**.
`groupby(F, L)`	Return a map with `#{K1 => I1, ..., Kp => Ip}` where `Ii` is a lists of indices to the values in `L` where `F` returns `Ki` (see example). Note that the function should be able to accept any function returning any atom. Awarded a maximum of **4 points**.

Example 3

```
1> map(fun (X) -> X*2 end, [1,2,3])
[2,4,6]

2> filter(fun (X) -> X > 0 end, [-1, 2, 0]).
[2]

3> split(fun (X) -> X > 0 end, [-1, 2, 0]).
{[2], [-1, 0]}

3> groupby(fun (X) -> if X < 0 -> negative;
                         X > 0 -> positive;
                         true ->  zero
                      end,
           end, [-1, 11, 10, -4, 0]).
#{negative => [1, 4], positive => [2, 3], zero => [5]}
```

[4] You are, however, allowed to use `lists:reverse/1`