Erlang basic – exercises

Contents

[1. The shell 2](#_Toc489454495)

[1.1. Job Control Mode. 2](#_Toc489454496)

[2. Basic types 2](#_Toc489454497)

[2.1. Numbers, atoms and simple arithmetics. 2](#_Toc489454498)

[2.2. Term comparison. 3](#_Toc489454499)

[2.3. Tuples. 3](#_Toc489454500)

[2.4. Lists and strings. 3](#_Toc489454501)

[3. Sequential Erlang 4](#_Toc489454502)

[3.1. Modules and functions 4](#_Toc489454503)

[3.2. BIFs and guards 4](#_Toc489454504)

[2.3. Recursion 5](#_Toc489454505)

[3.4. Records and macros 5](#_Toc489454506)

[3.5. Library modules 6](#_Toc489454507)

[3.6. Funs and higher-order functions 6](#_Toc489454508)

[3.7. List comprehensions 7](#_Toc489454509)

[3.8. Bit syntax 8](#_Toc489454510)

[3.9. Error handling 9](#_Toc489454511)

[4. Concurrent Erlang 9](#_Toc489454512)

[4.1. Processes, message passing, links 9](#_Toc489454513)

[4.2. Process supervision 10](#_Toc489454514)

[5. Robust systems 11](#_Toc489454515)

[5.1. Software upgrade 11](#_Toc489454516)

# The shell

## Job Control Mode.

Go to Job Control Mode (C-g) and try it out.

$erl

1>

CTRL+G

User switch command

--> h

--> j

--> c 1

etc.

# Basic types

## Numbers, atoms and simple arithmetics.

Type the following terms in your shell and check the returned values:

1> 25.

2> 3.89.

3> 2.3e9 / 5.

4> 36#ZF7.

5> $\n.

6> 5 div 3.7.

7> 5 / 3.7.

8> 5 div 2.

9> 5 rem 2.

10> 2.8e-2.

11> 2#100101.

12> 16#BF10.

13> 37#4999.

14> $H.

15> hello@world.

16> 'Atoms are awesome'.

17> foo == 'foo'.

18> "hello world".

## Term comparison.

Type the following expressions and see the result.

Is the returned value always as you expect? Why?

19> 3 < 4.

20> 5 > 7.9.

21> foo > 3.

22> 2.0 == 2.

23> 2.0 =:= 2.

24> 0 =< 1.

25> true > false.

26> 5 =/= 5.0.

27> 5 /= 5.0.

## Tuples.

Create 10 {Key, Value} tuples (Key is atom, Value is any type) and bind them with variables. Using pattern matching, extract Value from the tuple using Key as a pattern.

1> Tuple1 = {size, 5}.

2> Tuple2 = {price, 200}.

3> Tuple3 = {color, blue}.

## Lists and strings.

a) build a list of integers from 0 to 6 using cons operator

b) take 4 first elements from the list created in a) and form a new list

c) reverse the list not using library functions

d) using ++ and -- operators, create a string from several words and then remove all "a" letters from it

# Sequential Erlang

## Modules and functions

Create a new module named *exercises.erl*. Export functions:

a) even/1

even(Arg) → boolean() when Arg :: integer().

If Arg is even, return ‘true’, otherwise - ‘false’.

1> c(exercises).

{ok,exercises}

2> exercises:even(0).

true

3> exercises:even(1).

false

4> exercises:even(2).

true

5> exercises:even(-2).

true

b) gnirts/1 (requires recursion!)

gnirts(Arg) → string() when Arg :: string().

The functions takes string as an argument and returns its reversed form. For example:

5> c(exercises).

{ok,exercises}

6> exercises:gnirts("hello world").

"dlrow olleh"

## BIFs and guards

Add convert/0 function to *exercises.erl*:

convert() -> tuple() | term().

First, prompt the user to enter some text in the shell, it can be any valid term (but you can simplify it to numbers and strings). Using BIFs, check what type is the entered value, convert it to atom (unless it is already an atom) and print this information to the screen (and return a 2-tuple {Type, Value}).

Hint: check the documentation for *io* module. Suggestion: use io:read/1 for reading from stdin and io:format/2 for writing to stdout (or skip this part as in below example if the tuple is returned as in function spec, then it is printed anyway).

1> c(exercises).

{ok,exercises}

2> exercises:convert().

Enter a single term:aaa.

{atom,aaa}

**\*** **Advanced**: add a timestamp to the printed tuple (make it 3-tuple, try to format the timestamp in human-readable form, e.g. {"10:59.48", float, 8.45}).

Hint: check the documentation for *calendar* module.

## Recursion

1> c(exercises).

{ok,exercises}

2> exercises:convert().

Enter a single term:8.

{"14:16.09",integer,'8'}

a) add sum/1 function to *exercises.erl*:

sum(N) -> pos\_integer() when N :: pos\_integer().

The function takes a positive integer N and returns a sum of integers between 1 and N.

b) add sum/2 function to exercises.erl:

sum(N,M) -> pos\_integer() when M :: pos\_integer(), N :: pos\_integer().

The functions takes 2 positive integers N and M and, if N < M, it returns the sum of interval between N and M. If N >=M the program shall crash.

c) write functions that:

- reverses a list (reverse/1)

- **\*** **Advanced:** flattens a list (flatten/1)

d) write function create/1 which takes integer N and returns a list [1,2,3,.., N-1,N]. Write function rev\_create/1 which takes integer N and returns a list [N, N-1, .., 2,1].

e) write a function print/1 which takes an integer N and prints out the **even** integers between 1 and N.

## Records and macros

Create a new module db1.erl which will contain simple database of records:

-record(person, {

firstname,

lastname,

street}).

As a first approach, data will be stored in a list. Export functions add/2 and del/2 operating on the list. Usage example:

1> rr(db1).

[person]

2> P1 = #person{firstname = "John",lastname = "Smith",

street = "Sesame"}.

#person{firstname = "John",lastname = "Smith",

street = "Sesame"}

3> P2 = #person{firstname="Grazyna", lastname="Kowalska", street="Marszalkowska"}.

#person{firstname = "Grazyna",lastname = "Kowalska",

street = "Marszalkowska"}

4> P3 = #person{firstname="Janusz", lastname="Kowalski", street="Marszalkowska"}.

#person{firstname = "Janusz",lastname = "Kowalski",

street = "Marszalkowska"}

5> Data1 = db1:add(P1, []).

[#person{firstname = "John",lastname = "Smith",

street = "Sesame"}]

6> Data2 = db1:add(P2, Data1).

[#person{firstname = "Grazyna",lastname = "Kowalska",

street = "Marszalkowska"},

#person{firstname = "John",lastname = "Smith",

street = "Sesame"}]

7> Data3 = db1:add(P1#person{firstname="Anna"}, Data2).

[#person{firstname = "Anna",lastname = "Smith",

street = "Sesame"},

#person{firstname = "Grazyna",lastname = "Kowalska",

street = "Marszalkowska"},

#person{firstname = "John",lastname = "Smith",

street = "Sesame"}]

8> Data4 = db1:del(P1, Data3).

[#person{firstname = "Anna",lastname = "Smith",

street = "Sesame"},

#person{firstname = "Grazyna",lastname = "Kowalska",

street = "Marszalkowska"}]

## Library modules

Add function lookup/2 to *db1.erl*. The function shall search the database (list of records) based on #person.lastname value. Use appropriate functions from *lists* module.

## Funs and higher-order functions

Write functions (in *exercises.erl*) that:

a) applies a fun on every element on a list and returns a list of results (map/2)

b) filters a list based on user specified predicate, applies a fun on filtered elements and returns a list of results (mapfilter/3):

3> Pred = fun(X) -> X >= 25 end.

#Fun<erl\_eval.6.50752066>

4> Fun = fun(X) -> X div 3 end.

#Fun<erl\_eval.6.50752066>

5> Pred.

#Fun<erl\_eval.6.50752066>

6> Pred(3).

false

7> Pred(25).

true

8> Fun(49).

16

9> Fun(5).

1

10> exercises:mapfilter(Pred, Fun, [3, 45, 6, 76, 112, 0, 45]).

[15,25,37,15]

## List comprehensions

a) Add quicksort/1 function to exercises.erl:

quicksort(List) -> list() when List :: [integer()]

Quicksort algorithm takes the first element as Pivot and divides the rest of the list into 2 lists – one with all elements < Pivot and the other where all elements >= Pivot. Generating lists is done by list comprehension, further steps use recursion for processing every list separately, in the end all sorted lists are appended using ‘++’ operator.

1> c(exercises).

{ok,exercises}

2> exercises:quicksort([2,9,33,0,8,4]).

[0,2,4,8,9,33]

b) pretty print\* ASCII codes of all letters from some string (use list comprehensions to extract the letters, then io:format for pretty printing, what side effects do you see?)

\*by pretty printing I mean that the shell should print the ASCII codes separated by spaces, not the string:

„ala ma kota!”

will be printed similar to:

97 108 97 32 109 97 32 107 111 116 97 33

c) rewrite lookup/2 from *db1.erl* using either list comprehensions or recursive list handling.

Advanced: create a module *db2.erl* and implement the database as a list of records, but handled internally - the new API:

start() -> []

start/0 shall initiate the database and enter a loop where the list will be handed over as a loop data recursively, at every iteration waiting for user input:

A - shall prompt for data and add new element of type #person{} to the database

D - shall prompt for data and delete the element which matches lastname value in #person{} content

L - shall print the whole database content

Q - shall return the current database content and quit the program.

71> rr(db2).

[person]

72> db2:start().

Enter command:

A - add new user

D - delete user by last name

L - list all users

F - find user

Q - quit program

>A

Enter first name

>Jadzia

Enter last name

>Cholewa

Enter street

>Bajkowa

User {person,"Jadzia","Cholewa","Bajkowa"} added

Enter command:

A - add new user

D - delete user by last name

L - list all users

F - find user

Q - quit program

>Q

[#person{firstname = "Jadzia",lastname = "Cholewa",

street = "Bajkowa"}]

## Bit syntax

Using either a bit comprehensions or recursion, write a function coding an integer to binary BCD:

Hint: use integer\_to\_list BIF first to isolate the digits.

1> codecs:int\_2\_bcd(99568).

<<153,86,8:4>>

2> codecs:int\_2\_bcd(13).

<<19>>

Let's take an integer: 95740. integer\_to\_list will create a list of ASCII values: [$9, $5, $7, $4, $0],

or [57, 53, 55, 52, 48]. Next step is to take each element from the list and write it on 4 bits in some new bitstring. First, 48 must be substracted from every element (equal to $0) so we get [9, 5, 7, 4, 0] as input. This is not really necessary, as e.g. 57 = 0011 1001, so if the 4 most significant bits will be cut off due to bitstring construction (only 4 bits are allowed), it is the same as to substract 48 from the number  
((0011) 1001 -> 1001)

( 57(-48) -> 9)

## Error handling

See example *exceptions.erl*. Compile the module and run:

1> self().

2> exceptions:generate\_exception(1).

3> self().

4> exceptions:generate\_exception(2).

5> self().

6> exceptions:generate\_exception(3).

7> self().

8> exceptions:generate\_exception(4).

9> self().

10> exceptions:generate\_exception(5).

Then, instead of generate\_exception/1 function, run exceptions:catchme/1 with the same arguments (check own Pid as above between runs).

Add and export new catchme2/1 function which uses catch instead of try...catch. What is the difference now?

Add and export new catchme3/1 function and add stacktrace to the branch where errors are generated. Recompile, run and compare.

Advanced: Rewrite *exceptions.erl* so it produces 2 error messages: a polite message for the user (returned to the shell) and detailed message (with stacktrace) to the developer (saved to logfile).

# Concurrent Erlang

## Processes, message passing, links

a) rewrite *db.erl* to make the program concurrent - server part shall work as a separate process accepting requests from client processes by message passing. Use *echo.erl* as an inspiration.

*db3.erl* API:

start/0 - spawns the server process and registers it under the module’s name

stop/0 - stops the server process

add/1 - adds data to the database (#person{})

del/1 - removes the record #person if #person.lastname given as del(LastName) matches with database content (neglige duplicates for now); hint: use lists:keydelete/3

list/0 - returns the whole database content

lookup/1 - returns #person{} matching #person.lastname of false if not found (hint: use lists:keyfind/3)

b) create *calc.erl* module - a simple calculation server accepting requests  
{'+', ListOfNumbers}  
{'\*', ListOfNumbers}  
and returning calculated results to requestors. The server shall work in a separate process.

Advanced: try to spawn N processes connected to main server which will do the actual calculations - the server shall redirect the requests to one of the „slave” processes.

c) create N processes working as process ring - every spawn process shall send a number of messages to its neighbour, and shall also accept messages; if stop is received, the process shall exit. Think of the way how to make the neighbour’s Pid known to the process.

Example API:

*ring.erl*:

start(M, N, Message) - M= number of messages to send, N= number of processes, Message = message content.

## Process supervision

Take *simple\_supervisor.erl* example and modify it so it will be possible to add the servers from 4.1 as worker processes under supervisor control. Run the servers via the supervisor’s API and try to terminate them abnormally. Observe if the supervisor recreates the servers.

# Robust systems

## Software upgrade

See the example modules:  
*a.erl*

-module(a).

-compile(export\_all).

-vsn(1.0).

start(Tag) ->

spawn(fun() -> loop(Tag) end).

loop(Tag) ->

sleep(),

Val = b:x(),

io:format("Vsn1 (~p) b:x() = ~p~n", [Tag, Val]),

loop(Tag).

sleep() ->

receive

after 3000 -> true

end.

*b.erl*

Compile both modules and run:

-module(b).

-export([x/0]).

-vsn(1.0).

x() -> 1.

Now modify and recompile *b.erl*:

5> c(a).

{ok,a}

6> c(b).

{ok,b}

7> a:start(one).

<0.56.0>

Vsn1 (one) b:x() = 1

Vsn1 (one) b:x() = 1

Vsn1 (one) b:x() = 1

8> a:start(two).

<0.58.0>

Vsn1 (one) b:x() = 1

Vsn1 (two) b:x() = 1

Vsn1 (one) b:x() = 1

Vsn1 (two) b:x() = 1

-module(b).

-export([x/0]).

-vsn(2.0).

x() -> 2.

Now let’s see what happens if *a.erl* is changed:  
  
  
  
Let’s start another process (no recompile):

Vsn1 (one) b:x() = 1

Vsn1 (two) b:x() = 1

9> c(b).

{ok,b}

Vsn1 (one) b:x() = 2

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

-module(a).

-compile(export\_all).

-vsn(2.0).

start(Tag) ->

spawn(fun() -> loop(Tag) end).

loop(Tag) ->

sleep(),

Val = b:x(),

io:format("Vsn2 (~p) b:x() = ~p~n", [Tag, Val]),

loop(Tag).

sleep() ->

receive

after 3000 -> true

end.

10> c(a).

{ok,a}

Vsn1 (one) b:x() = 2

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

Now it’s time to change *b.erl* again:  
  
Finally, *a.erl* will get a new (3rd) version:

Vsn1 (one) b:x() = 2

11> a:start(three).

<0.70.0>

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

Vsn2 (three) b:x() = 2

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

Vsn2 (three) b:x() = 2

-module(b).

-export([x/0]).

-vsn(3.0).

x() -> 3.

Vsn1 (two) b:x() = 2

Vsn1 (one) b:x() = 2

12> c(b).

{ok,b}

Vsn2 (three) b:x() = 3

Vsn1 (two) b:x() = 3

Vsn1 (one) b:x() = 3

Vsn2 (three) b:x() = 3

Vsn1 (two) b:x() = 3

Recompile and observe:

-module(a).

-compile(export\_all).

-vsn(3.0).

start(Tag) ->

spawn(fun() -> loop(Tag) end).

loop(Tag) ->

sleep(),

Val = b:x(),

io:format("Vsn3 (~p) b:x() = ~p~n", [Tag, Val]),

loop(Tag).

sleep() ->

receive

after 3000 -> true

end.

Vsn2 (three) b:x() = 3

Vsn1 (two) b:x() = 3

Vsn1 (one) b:x() = 3

13> c(a).

{ok,a}

Vsn2 (three) b:x() = 3

Vsn2 (three) b:x() = 3

Vsn2 (three) b:x() = 3

14> a:start(four).

<0.82.0>

Vsn2 (three) b:x() = 3

Vsn3 (four) b:x() = 3

Vsn2 (three) b:x() = 3

Vsn3 (four) b:x() = 3