Programmeertalen: Erlang

A distributed functional programming language

Ana Oprescu, Robin de Vries

Universiteit van Amsterdam

4 maart 2019

Programmeertalen: where are we now?

	Programming Language	**Lecture**			
Week 1	Bash	mo 4/2	11:00-13:00	SP C0.05	Bas
Week 2	Haskell	mo 11/2	11:00-13:00	SP C0.05	Ana
Week 3	Prolog	mo 18/2	11:00-13:00	SP C0.05	Koen
Week 4	Python	mo 25/2	11:00-13:00	SP C0.05	Bas
Week 5	**Erlang**	**mo 4/3**	**11:00-13:00**	**SP C0.05**	**Ana**
Week 6	Go	mo 11/3	11:00-13:00	SP C0.05	Ana
Week 7	C++	mo 18/3	11:00-13:00	SP C0.05	Bas
Week 8	Exam	thu 28/3	13:00-16:00	USC Sporthal 2 SP	

Today's menu

Introduction

Lists

Functions

Anonymous functions

Concurrency

Tips 'n tricks

Erlang

- Developed by Ericsson (see also Open Telecom Platform)
  - Er-lang, but also Agner Erlang
- First appeared in 1986, released as open source in 1998
- Joe Amstrong (co-creator) gets his PhD from KTH in 2003 `http://erlang.org/download/armstrong_thesis_2003.pdf`
- Distributed, functional
- Dynamic and strong typing
- Fault tolerant
  - Let it crash!
- Bytecode runs in a VM

Characteristics

- Functional programming language
- Prolog-like syntax
- Support for concurrency
- Joe Amstrong: ”Write once, run forever” :)

Learning Erlang – recommended reading[1]

- Learn You some Erlang for Great Good! `http://learnyousomeerlang.com/content`
- An Erlang Course `http://www.erlang.org/course/course.html`
- Erlang/OTP documentation `http://www.erlang.org/doc/`
- Erlang programming guidelines `http://www.erlang.se/doc/programming_rules.shtml`

[1] Some slides and examples in this presentation originate from this material.

Starting Erlang

Open a terminal and start the Erlang runtime system using **erl**

```
$ erl
Erlang/OTP 21 [erts-10.2.3] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:1] [hipe] ...

Eshell V10.2.3  (abort with ^G)
1>
```

Files, modules, comments and Hello World!

hello.erl

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

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

```
Eshell V6.4  (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:hello_world().
Hello World!
ok
```

Data Types

- Numbers: integers and floats
- Variables:
  - assignment only once "binding", begin with uppercase
  - only in the shell environment, variables may be reset: f(Variable)
- Atoms: begin with lowercase
  - enclosed in single quotes (') if it does not begin with a lower-case letter or if it contains other characters than alphanumeric characters, underscore (_), or @
  - reserved words: after and andalso band begin bnot bor bsl bsr bxor case catch cond div end fun if let not of or orelse query receive rem try when xor
  - atom table is not garbage collected!
- "Boolean"
- Tuples
- Lists

(In)variables assignment

- Assignment is actually a comparison with special treatment of unbound variables.

```
7> A=5.
5
8> A=5.0.
** exception error: no match of right hand side value 5.0
9> A=2+3.
5
10> f(A).
ok
11> A=5.0.
5.0
```

Arithmetic operations

```
1> 3 + 20.
23
2> 5 rem 4.
1
3> 19 div 3.
6
4> false and true.
false
5> true + false.
** exception error: an error occurred when evaluating an arithmetic expres
     in operator +/2
        called as true + false
6> not false.
true
7> 3 / 2.
1.5
```

And more, such as `or`, `xor`, `andalso` and `orelse`.

(In)equality

- Test (in)equality: =:= and =/=
- Test (in)equality with int/float conversion: == en /=

```
1> 1 >= 1.
true
2> 1 =< 1.0.
true
12> 4+1 == 2+3.
true
13> 4+1 =:= 2+3.
true
14> 4+1.0 =:= 2+3.
false
15> 4+1.0 == 2+3.
true
19> false < true.
true
6> wrong =:= 'wrong'.
true
```

Atoms and variables

```
3> 5+llama.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as 5 + llama
3> false + true.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as false + true
4> llama = 4.
** exception error: no match of right hand side value 4
5> Llama = 4.
4
6> 5 + Llama.
9
10> 4 == llama.
false
11> 4 == Llama.
true
12> 5 + true.
** exception error: an error occurred when evaluating an arithmetic expression
     in operator  +/2
        called as 5 + true
```

Atoms and variables

- **Total ordering** is important! which order, not.
- number < atom < reference < fun < port < pid < tuple < list < bit string

```
13> true == llama.
false
12> f(Llama).
ok
16> 5 < true.
true
17> 5 < llama.
true
18> true < llama.
false
20> Llama < true.
* 1: variable 'Llama' is unbound
21> Llama = 4.
4
22> Llama < true.
true
23> Llama < 5.
true
26> ((false < true) and (true < Llama)) and (Llama < 5).
false
```

Atoms and variables

- **Total ordering** is important!

```
7> ((false < true) and (true < Llama)) and (Zebra = 6).
** exception error: bad argument
     in operator  and/2
        called as false and 6
28> ((false < true) and (true < Llama)) and (Zebra == 6).
* 1: variable 'Zebra' is unbound
29> ((false < true) and (true < Llama)) and ((Zebra = 6) == 6).
false
36> ((false < true) and (true < Llama)) andalso ((Lion = 9) == 6).
false
37> Lion.
* 1: variable 'Lion' is unbound
38> ((false < true) and (true < Llama)) and ((Lion = 9) == 6).
false
39> Lion.
9
```

Today's menu

Introduction

Lists

Functions

Anonymous functions

Concurrency

Tips 'n tricks

Lists

Collection of homogeneous elements. Homogeneity is not enforced in Erlang.

Lists

Collection of homogeneous elements. Homogeneity is not enforced in Erlang.

```
1> X = [1,2,3,4,5,6].
[1,2,3,4,5,6]
2> [H|T] = X.
[1,2,3,4,5,6]
3> H.
1
4> T.
[2,3,4,5,6]
5> hd(X).
1
6> tl(X).
[2,3,4,5,6]
7> length(X).
6
```

Strings

```
1> [67,98,99].
"Cbc"
2>length(tl("wrong")).
4
```

Strings

Strings are a list of integers.

The interpreter chooses automatically the representation.

```
2> [1|"hoooi"].
[1,104,111,111,111,105]
3> [233].
"   "

4> [1,233].
[1,233]
```

Note: Erlang prints lists of numbers as numbers only when at least one of them could not also represent a letter.

Lists

Two lists can be concatenated using the ++ operator. The -- operator performs the opposite. Both operators are *right-associative*.

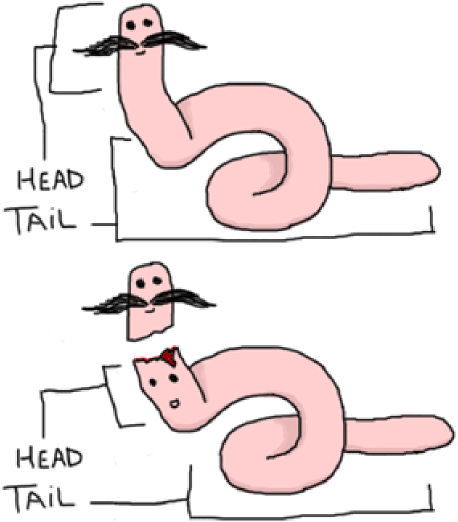
```
1> [1,2,3] ++ "abc" ++ "def".
[1,2,3,97,98,99,100,101,102]

2> [1,2,3] -- [1,2] -- [3].
[3]

3> [1,2,3] -- [1,2] -- [2].
[2,3]

4> ([1,2,3] -- [1,2]) -- [3].
[]
```

Lists

Similar to Prolog:

```
1> [1,2,3,4,5,6,7].
[1,2,3,4,5,6,7]
2> [1,2 | [3,4,5,6,7]].
[1,2,3,4,5,6,7]
3> [1,2,3 | [4,5,6,7]].
[1,2,3,4,5,6,7]
4> [1,2 | [[3,4,5,6,7]]].
[1,2,[3,4,5,6,7]]
5> [[1,2] | [3,4,5,6,7]].
[[1,2],3,4,5,6,7]
```

- Consult the documentation for more built-in functions (BIFs) `http://www.erlang.org/doc/man/lists.html`
- Watch out for improper lists texttt[1—2].



List comprehensions

Similar to Haskell:

NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN, Condition1, Condition2, ... ConditionM]

```
1> L = lists:seq(1,20).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

3> [ 2 * X || X <- L, X rem 2 =:= 1].
[2,6,10,14,18,22,26,30,34,38]

4> Weather = [{toronto, rain}, {montreal, storms}, {london, fog}, {paris,
...
5> RainyPlaces = [X || {X, rain} <- Weather].
[toronto,amsterdam]
```

Pythagoras triplets?

List comprehensions

Similar to Haskell:
NewList = [Expression || GeneratorExp1, GeneratorExp2, ..., GeneratorExpN, Condition1, Condition2, ... ConditionM]

```
1> L = lists:seq(1,20).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

3> [ 2 * X || X <- L, X rem 2 =:= 1].
[2,6,10,14,18,22,26,30,34,38]

4> Weather = [{toronto, rain}, {montreal, storms}, {london, fog}, {paris,
...
5> RainyPlaces = [X || {X, rain} <- Weather].
[toronto,amsterdam]
```

Pythagoras triplets?

```
6> [ {X,Y,Z} || X <- L, Y <- L, Z <- L, X*X+Y*Y =:= Z*Z, X < Y].
[{3,4,5},{5,12,13},{6,8,10},{8,15,17},{9,12,15},{12,16,20}]
```

Chocolate time!

Define your own version of `lists:reverse()/1`.

```
1> lists:reverse(”Hoooi”).
”ioooH”
```

Chocolate time!

Define your own version of `lists:reverse()/1`.

```
1> lists:reverse("Hoooi").
"ioooH"
```

toy.erl

```
-module(toy).
-export([tail_reverse/1,myreverse/1]).

tail_reverse(L) ->
             tail_reverse(L,[]).

tail_reverse([],Acc) -> Acc;
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).

myreverse([]) -> [];
myreverse([H|T]) -> myreverse(T)++[H].
```

```
2> lists:reverse("Hoooi").
```

One more chance at chocolate!

Define your own version of `lists:member/2`.

```
1> lists:member(1,[1,2,3]).
true
2> lists:member(4,[1,2,3]).
false
```

One more chance at chocolate!

Define your own version of `lists:member/2`.

```
1> lists:member(1,[1,2,3]).
true
2> lists:member(4,[1,2,3]).
false
```

lecture.erl

```
member(_, [])     -> false;
member(E, [E|_]) -> true;
member(E, [_|T]) -> member(E, T).
```

```
3> lecture:member(1,[1,2,3]).
true
4> lecture:member(4,[1,2,3]).
false
```

Today's menu

Introduction

Lists

Functions

Anonymous functions

Concurrency

Tips 'n tricks

Pattern-matching and unbound-variables

lecture.erl

```erlang
lucky(4) -> lucky;
lucky(6) -> doomed;
lucky(7) -> very_lucky;
lucky(_) -> not_so_lucky.
```

```
1> lecture:lucky(6).
doomed
2> lecture:lucky(7).
very_lucky
3> lecture:lucky(10).
not_so_lucky
```

Guards, Guards!

Older than 18?

```
old_enough(0)  -> false;
old_enough(1)  -> false;
old_enough(2)  -> false;
...
old_enough(16) -> false;
old_enough(17) -> false;
old_enough(_)  -> true.
```

Guards, Guards!

Older than 18?

```
old_enough(0)  -> false;
old_enough(1)  -> false;
old_enough(2)  -> false;
...
old_enough(16) -> false;
old_enough(17) -> false;
old_enough(_)  -> true.
```

Using guards:

```
old_enough(X) when X >= 18 -> true;
old_enough(_)              -> false.
```

Guards, Guards! (2)

Older than 18, younger than 104

```
right_age(X) when X >= 18, X =< 104 ->  true;
right_age(_)                        ->  false.
```

Watch out for , versus ;! similar to `andalso` versus `orelse`. However, only `andalso` and `orelse` can be nested in guard statements.

```
wrong_age(X) when X < 18; X > 104 -> true;
wrong_age(_)                      -> false.
```

If-statements Erlang-style

```erlang
answer_to_life(X) ->
    if
        X =:= 42  -> galaxy;
        X =:= 666 -> lucifer;
        true      -> false  % else-statement Erlang-style
    end.
```

Case

```
lucky_case(X) ->
    case X of
        4 -> lucky;
        6 -> doomed;
        7 -> very_lucky;
        _ -> not_so_lucky
    end.
```

Case with when

- Case with Pattern Matching

```erlang
beach(Temperature) ->
  case Temperature of
    {celsius, N} when N >= 20, N =< 45       -> 'favorable';
    {kelvin, N} when N >= 293, N =< 318      -> 'scientifically favorable';
    {fahrenheit, N} when N >= 68, N =< 113 -> 'favorable in the US';
    _                                         -> 'avoid beach'
  end.
```

Input and output

lecture.erl

```
{erlang}
length_input() ->
   X = io:get_line("Input sentence: "),
   L = length(X),
   io:format("Sentence: ~s, length: ~w~n",[X,L]).
```

Consult `http://erlang.org/doc/man/io.html` for information regarding functions in the `IO-module`.

Today's menu

Introduction

Lists

Functions

Anonymous functions

Concurrency

Tips 'n tricks

Defined functions as arguments

lecture.erl

```
one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

How can we call `add/2` such that the outcome is 3?

Defined functions as arguments

lecture.erl

```
one() -> 1.
two() -> 2.

add(X,Y) -> X() + Y().
```

How can we call add/2 such that the outcome is 3?

```
1> lecture:add(fun lecture:one/0,fun lecture:two/0).
3
```

Anonymous functions

Syntax:

```
fun(Args1) ->
Expression1, Exp2, ..., ExpN;
(Args2) ->
Expression1, Exp2, ..., ExpN;
(Args3) ->
Expression1, Exp2, ..., ExpN
end
```

Example:

```
1> lists:map(fun(A) -> A + 1 end, [1,2,3]).
[2,3,4]

2> lists:filter(fun(A) -> A > 10 end, lists:seq(1,20)).
[11,12,13,14,15,16,17,18,19,20]
```

Anonymous functions (2)

Erlang has `foldl` and `foldr`, but with a different syntax than Haskell.

```
1> lists:foldl(fun(X,Y) -> X+2*Y end, 4, [1,2,3]).
43

2> lists:foldr(fun(X,Y) -> X+2*Y end, 4, [1,2,3]).
49
```

Today's menu

Introduction

Lists

Functions

Anonymous functions

Concurrency

Tips 'n tricks

Spawning processes

Processes can be started using `spawn`. `spawn` returns the Process Identifier (PID) of the new process.

```
1> F = fun() -> io:format("Hello process!~n") end.
#Fun<erl_eval.20.90072148>
2> F().
Hello process!
ok
3> spawn(F).
Hello process!
<0.43.0>
```

Spawning processes (2)

lecture.erl

```
hello_process(X) ->
    io:format("Hello ~w~n",[X]).
```

```
1> spawn(lecture, hello_process,[world]).
Hello world
2> spawn(hofuns, hello_process,[", Ana!"]).
Hello [44,32,65,110,97,33]
<0.108.0>
```

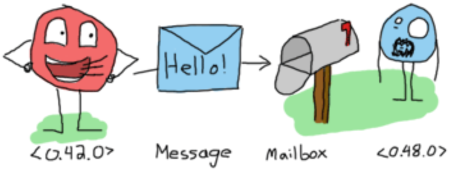
lecture.erl

```
hello_process(X) ->
    io:format("Hello ~s~n",[X]).
```

```
2> spawn(hofuns, hello_process,[", Ana!"]).
Hello , Ana!
<0.99.0>
```

Sending messages

Using the !-operator (bang symbol) you can send messages to processes.



```
3> self() ! hello.
hello
4> self() ! world.
world
5> flush().
Shell got hello
Shell got world
ok
```

Receiving messages

lecture.erl

```
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

Receiving messages

lecture.erl

```
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

```
1> P = spawn(lecture,hi_process,[]).
<0.111.0>
2> P ! {print, "Hello!"}.
Message: Hello!
{print,"Hello!"}
6>  P ! {_, "Hello!"}.
* 2: variable '_' is unbound
7>  P ! {write, "Hello!"}.
What?
{write,"Hello!"}
```

Receiving messages

lecture.erl

```erlang
hi_process() ->
    receive
        {print, Message} -> io:format("Message: ~s~n",[Message]),
                            hi_process();
        stop             -> io:format("Goodbye!~n");
        _                -> io:format("What?~n"),
                            hi_process()
    end.
```

```
8>  P ! {stop}.
What?
{stop}
9>  P ! stop.
Goodbye!
stop
10>  P ! {stop}.
{stop}
```

A functional mindset of concurrency

How can we create ten processes at once?

A functional mindset of concurrency

How can we create ten processes at once?

lecture.erl

```erlang
hi_process(N) ->
    receive
        {print, Message} -> io:format("~w: Message: ~s~n",[N,Message]),
                            hi_process(N);
        stop             -> io:format("Goodbye from ~w!~n",[N]);
        _                -> io:format("~w What?~n",[N]),
                            hi_process(N)
    end.
```

```
1> L = [ spawn(lecture, hi_process, [X]) || X <- lists:seq(1,10) ].
[<0.46.0>,<0.47.0>,<0.48.0>,<0.49.0>,<0.50.0>,<0.51.0>,
 <0.52.0>,<0.53.0>,<0.54.0>,<0.55.0>]
```

A functional mindset of concurrency (2)

How do we send a message to all ten processes?

A functional mindset of concurrency (2)

How do we send a message to all ten processes?

```
2> lists:foreach( fun(P) -> P ! {print, ”Hello”} end, L).
1: Message: Hello
2: Message: Hello
3: Message: Hello
4: Message: Hello
5: Message: Hello
6: Message: Hello
8: Message: Hello
9: Message: Hello
7: Message: Hello
10: Message: Hello
ok
```

There is no guarantee that the messages would arrive in this order!

A functional mindset of concurrency (3)

```
3> lists:foreach( fun(P) -> P ! {write, "Hello"} end, L).
2 What?
3 What?
5 What?
7 What?
8 What?
1 What?
10 What?
4 What?
6 What?
9 What?
ok
```

There is no guarantee that the messages would arrive in this order!

A functional mindset of concurrency (4)

```
3> lists:map( fun(P) -> P ! stop end, L).
Goodbye from 8!
Goodbye from 9!
Goodbye from 10!
Goodbye from 1!
Goodbye from 2!
Goodbye from 3!
Goodbye from 4!
Goodbye from 5!
Goodbye from 6!
Goodbye from 7!
[stop,stop,stop,stop,stop,stop,stop,stop,stop,stop]
```

There is no guarantee that the messages would arrive in this order!

Round Robin: the rotate game!

lecture.erl

```
rotate([H|T]) -> T ++ [H].

rotate_game(X) ->
    receive
        stop     -> io:format("~w game is over~n",[X]);
        {25, Ps} -> io:format("~w ends game~n",[X]),
                    lists:foreach( fun(P) -> P ! stop end, Ps);
        {N, Ps } -> io:format("~w increases ~w~n",[X, N]),
                    hd(Ps) ! {N + 1, rotate(Ps)},
                    rotate_game(X)
    end.
```

Round Robin: the rotate game!

lecture.erl

```
rotate([H|T]) -> T ++ [H].

rotate_game(X) ->
    receive
        stop      -> io:format("~w game is over~n",[X]);
        {25, Ps} -> io:format("~w ends game~n",[X]),
                    lists:foreach( fun(P) -> P ! stop end, Ps);
        {N, Ps } -> io:format("~w increases ~w~n",[X, N]),
                    hd(Ps) ! {N + 1, rotate(Ps)},
                    rotate_game(X)
    end.
```

```
1> Ps = [spawn(lecture, rotate_game, [X]) || X <- lists:seq(1,10) ].
2> hd(Ps) ! {0, Ps}.
```

Behaviours

Erlang conceptually distinguishes between work and supervision

Hierarchical organization of code to deliver fault-tolerance

Behaviours are formalizations of patterns, i.e., servers, finite-state machines, event handlers

- generic part: behaviour module
- specific part: callback module, supplied by the developer

asynchronous calls (cast) are expected to return a tuple {noreply, State}

synchronous calls (call) are expected to return a tuple {reply, Reply, State}

Distributed Erlang

Erlang conceptually maps a VM to a node

Nodes may be named `name@host`

- long names: `host` is the full host name
- short names: `host` is the first part of the host name

Processes may be spawned on another node

Today's menu

Introduction

Lists

Functions

Anonymous functions

Concurrency

Tips 'n tricks

Tips 'n tricks

- Use `halt().` or Ctrl+\to terminate the Erlang interpreter.
- Reset a variable: `f/1`
- Reset all variables: `f/0`
- Server Behaviour: `http://erlang.org/doc/design_principles/gen_server_concepts.html`
- Supervisor Behaviour: `http://erlang.org/doc/design_principles/sup_princ.html`
- make sure the epmd is running before naming a node

Next week

Go!