Sequential Erlang

Lectures F2 + F3 (Chapters 1..6)

Joe Armstrong

Ericsson AB

Concurrent Erlang

Really I'd like to start the course with concurrent and distributed programming. But we have to learn to write sequential programs before we can write concurrent programs.

Learning to write sequential Erlang programs takes a while. It's a functional programming language and it takes a while to get used to this.

Writing concurrent programs is *easy* - but debugging them can be hard.

Some distributed problems are impossible to solve *there are theorems about this* - we won't solve these :-)

 ©2014 – Joe Armstrong

What we'll do

- Learn Sequential Erlang.
- Learn Concurrent Erlang.
- Leann Distributed Erlang.
- Learn How to write concurrent and parallel programs.
- Make a bit-torrent like program "the Erlang way". 
- Start a company and earn a trillion dollars (not in this course).
- Change the world.
- Have fun.

 ©2014 – Joe Armstrong

and how we will do it...

- Each week will cover a continuous sequence of chapters from *Programming Erlang*.
- This week cover chapters 1..6.
- There is one set of slides and one set of problems per week.
- The lectures have a lot of live coding (not in the slides).
- The exercises must be done within a week.
- You get out of the course what you put into it.
- Warn me if I drift off topic :-) *we can always talk after the lectures*.

 ©2014 – Joe Armstrong

How to learn any programming language

- Get a decent book.
- Type in all the examples (do not cut and paste examples).
- There is a slow method to learn languages s Google for the answer to every problem...

 ©2014 – Joe Armstrong

The course website

- Assignments and other material are at `https://github.com/joearms/paradis/blob/master/week4_problems.erl`.

- It's a GIT repository. Please push changes to me. *help me improve the course better*.

 ©2014 – Joe Armstrong

Help me improve the course

- We start with a emacs org-mode, this file is `f2-f3.org`.
- An Erlang program `orgmode_parse.erl` transforms it into `f2-f3.org.tex`. This adds color coding of code and a few other nice things.
- Run `pdflatex` twice and get `f2-f3.org.pdf`.
- This system was written last week and is possibly buggy.
- Please help me improve the material. Push all changes to github.
- There are no course credits for helping.
- tell me if the examples are too easy or difficult. They should take N hours/week. What is N?

 ©2014 – Joe Armstrong

Let's get started

Agner Krarup Erlang (1878 - 1929)

 ©2014 – Joe Armstrong

Erlang

- Erlang was designed for building **fault-tolerant**, **concurrent**, **scaleable** applications.
- The world *is* concurrent.
- Erlang belongs to the **actor** family of languages. There are two types of concurrency:
- Shared-Memory Concurrency (Lectures F16..F18).
- Message Passing Concurrency - Sending and receiving messages is the only way for processes to exchange data.
- We never know if a message is received (it might get lost on the way) - If you want to know if a message has been received then send a reply.
- Message passing is “Location transparent” (like sending letter in the mail).

 ©2014 – Joe Armstrong

Erlang

- “Functional” core.
- “Math” variables.
- Beautiful syntax.
- No mutable data (not really true).
- Concurrency (and parallelism) is built-in.
- Inbuilt fault-tolerance.
- Modules.
- Not OO (Actors).
- Practical.
- Battle Tested (not a theory).

Not objekt orientated

Inte en teori

Klarna - konkurerade ut andra med hjälp av erlang

 ©2014 – Joe Armstrong

Starting the shell

- Mac OX-S, *nix:

```
> erl
```

- Windows:

```
Programs -> OTP ... -> Erlang
```

```
$ erl
Erlang (BEAM) emulator version 5.5.4 [source]
  [async-threads:0] [kernel-poll:false]
Eshell V5.5.4 (abort with ^G)
1> 1 + 2 * 3.
7
```

 ©2014 – Joe Armstrong

Stopping the shell

- `(ctrl)+\` – immediate exit
- `init:stop()` – controlled exit
- `erlang:halt()` – uncontrolled exit
- `(ctrl)+C`

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo
     (l)oaded (v)ersion (k)ill (D)b-tables
     (d)istribution
a
```

 ©2014 – Joe Armstrong

Shell commands

- Shell is `read-eval-print` loop.
- Commands end `.`WS.
- Repeated prompt means command is not yet finished:

```
1> 12234 * 12313.
150637242
2> math:sqrt(2).
1.41421
3> [123, abc,"hello"].
[123,abc,"hello"]
4> 1234 +
4> 34524249 *
4> 11112231.
383641429990753
```

 ©2014 – Joe Armstrong

Using the shell

- REPL (Read Eval Print Loop) is typical for this type of language.
- Same of all platforms (good for windows).
- A session: Show this: Variables in erlang don't vary

```
$erl
...
1> X = 23+10.
33
2> X + X.
66
...
```

Man kan tilldela variabel en gång

- All expressions end in ".".
- Quit with `q()` or `^C`.
- Emacs conventions apply in the shell.

 ©2014 – Joe Armstrong

Variables don’t vary

- Variable start with an uppercase letter and are bound with =:

```
$erl
...
1> X = 10.
10
2> X = 20.
  exception error:
   no match of right hand side value 20
...
3> X1 = 20.
20
```

- Use a new variable each time (more later)

Single static variables - tilldelas en gång och försvinner sedan när deras scope tar slut

 ©2014 – Joe Armstrong

Data Types

två datatyper.
1. Primitiv: int, atoms, floats
2. Compound data types: består av inopsatta primitiva.

Erlang has two types of data. There are *Primitive data types* (atoms, integers, floats) etc. and *Compound data types*. Compound data types glue together data. The two most common forms of glue are *Lists* and *Tuples*. Records provide syntactic sugar for accessing tuples. *maps* are associative Key-Value stores.

```
> X = abc.
abc
> Y = 123.
123
> L = [abc,123].
[abc,123]
> M = [xyz,L,1234].
[xyz,[abc,123],1234]
```

Strings

- There are no strings in Erlang.

- Strings are lists of integers:

```
1> "abc".
"abc".
2> [abc|"abc"].
[abc,97,98,99]
```

monday.

monday <-- ex på atom. Går bara att jämföra

 ©2014 – Joe Armstrong

Primitive Data Types

- Atoms `monday tuesday` - Remember atoms start with a lower case letters.

- Booleans `true false`.

- Integers 123, 21309103801830183081038103801 8, 16#f234, 2#23, $a:

```
> x = abc.
  exception error:
  no match of right hand side value abc
> x = x.
x
```

 ©2014 – Joe Armstrong

Numbers

- Integers:
  - 1234
  - 27391836713581739719319837917391739173218361836
  - 16#ca23ad12 – hex
  - 2#1010101 – base 2
  - N#DDDDD – base N
  - $a (asci code for a = 97)
- Floats.
- Atoms:
  - Booleans `true`, `false`
  - Constants `monday`, `tuesday`

 ©2014 – Joe Armstrong

Compound Data (Lists)

- Lists are containers for a variable number of items.
- `X = [1,2,3,abc,true]` - Even complex items can be in the list.
- `[Head|Tail]` is used to construct or deconstruct a list:

```
> L = [1,2,3,4].
[1,2,3,4]
> [H|T] = [1,2,3,4].
[1,2,3,4]
> H.
1
> T.
[2,3,4]
```

 ©2014 – Joe Armstrong

Constructing a list

```
> T = [1,2,3].
[1,2,3]
> H = a.
a
> [H|T].
[a,1,2,3]
```

 ©2014 – Joe Armstrong

What is a list (really)

- It's a cons cell in LISP.
- It's "linked list" in C.

 ©2014 – Joe Armstrong

Tuples

- Tuples are containers for a fixed number of items:

```
> X = {1,2,3}.
{1,2,3}.
> {_,Y,_} = X.
{1,2,3}.
> Y.
2
```

 ©2014 – Joe Armstrong

Patterns

- A “Term” is an atomic or compound data value.
- A “Pattern” is a data value or a variable.
- Variables are bound in pattern matching operations:

```
> X = {1,2,1}.
{1,2,1}
> {Z,A,Z} = X. ???
> {P,Q,R} = X. ???
...
```

- If variables are repeated in a pattern then they must bind to the same value.
- _ is a wildcard (matches anything).

 ©2014 – Joe Armstrong

Unpacking a list

```
> L = [1,2,3,4].
[1,2,3,4]
> [H|T] = L.
[1,2,3,4]
> H.
1
> T.
[2,3,4]
```

- Show lot's of examples.

Functions

- Multiple Entry Points

```
area({square,X}) -> X*X;
...
area({rectangle,X,Y}) -> X*X.
```

- C/Javascript/... have single entry points so don't write:

```
function area(X){
   if(X.type == 'square'){
      ...
   } elseif{X.type='rectangle'){
      ...
}
```

 ©2014 – Joe Armstrong

Modules

- All code is defined in modules.
- Modules are the unit of compilation.
- Modules can live-upgraded.
- Modules limit the visibility of internal functions.

 ©2014 – Joe Armstrong

Structure of a Module

- Modules look like this:

```
-module(math1).
-export([area/1]).

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

- The filename must be `math1.erl`:

```
$ erl
1> c(math1).
{ok,math1}
2> math1:area({square,12}).
144
```

 ©2014 – Joe Armstrong

Punctuation

- DOT whitespace ends a function.
- Semicolon ";" separates clauses.
- Comma "," separates arguments.
- Getting the punctuation wrong is the single biggest mistake beginners make.
- Use a text editor that matches parentheses.

 ©2014 – Joe Armstrong

Add Unit tests

- Add unit test like this:

```
-module(math2).
-export([test/0, area/1]).

test() ->
   144 = area({square,12}),
   200 = area({rectangle,10,20}),
   hooray.

area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```

 ©2014 – Joe Armstrong

Exports

```
-module(mod1).
-export([func1/2, func3/2]). %% public stuff

func1(X, Y) -> %% A public function
  boo(X, Y, 12).

boo(X, Y, X) -> %% A private function
  ...
```

- Cheat `-compile(export_all).`

 ©2014 – Joe Armstrong

Imports

- Imports permit a short form of the calling sequence.
- Late Binding - always calls the latest version of the code:

```
-module(mod1).
-import(lists, [reverse/1]).

func1(L) ->
  L1 = reverse(L). %% it's really lists:reverse
```

- Cheat `-compile(export_all).`

 ©2014 – Joe Armstrong

Assignments

Fetch https://github.com/joearms/paradis/blob/master/week4_problems.erl

It's *approximately*:

```
-module(week4_problems).
-compile(export_all).
-export([test/0]).

test() ->
   120 = ex1:factorial(5),
   L = [a,b,c,d,e,f],
   [b,c,d,e,f,a] = ex1:rotate(1,L),
   [f,a,b,c,d,e] = ex1:rotate(-1,L),
   ...
   horray.
```

 ©2014 – Joe Armstrong

Did I get the assignments right?

- Check in the shell

```
$ erl
1> c(week4_problems).
{ok,week4_problems}
2> c(ex1).
{ok,ex1}
3> week4_problems:test().
horray
```

 ©2014 – Joe Armstrong

case expression

```
area(A) ->
   case A of
      {square, X} ->
         X*X;
      {rectangle, X, Y} ->
         X*Y
      ...
   end
```

Or:

```
area({square,X}) -> X*X;
area({rectangle,X,Y}) -> X*Y.
```


©2014 – Joe Armstrong

if expression

```
weekend(Day) ->
   if
      Day == saturday ->
         true;
      Day == sunday ->
         true;
      true ->
         false
   end.
```

- if is an expression and not an statement.

 ©2014 – Joe Armstrong

Funs

- define own control abstractions:

```
for(Max,Max,F) -> [F(Max)];
for(I, Max, F) -> [F(I) | for(I+1,Max,F)].
```

```
1> D = fun(X) -> 2*X end.
#Fun<erl_eval.6.17052888>
2> hofs:for(1,10,D).
[2,4,6,8,10,12,14,16,18,20]
3>
```


©2014 – Joe Armstrong

Simple list recursion

```
double([]) -> [];
double([H|T]) -> [2*H|double(T)].
```

```
3> c(math1).
{ok,math1}
4> math1:double([1,2,3,4,5]).
[2,4,6,8,10]
```

 ©2014 – Joe Armstrong

Accumulators

```
sum(L) -> sum_helper(L, 0).

sum_helper([], N) -> N;
sum_helper([H|T], N) ->
   N1 = N + H,
   sum_helper(T, N1).
```

```
1> math1:sum([1,2,3,4,5]).
15
```

 ©2014 – Joe Armstrong

sum_and_double

- travserses the list twice

```
sum_and_double(L) ->
    Sum = sum(L),
    Double = double(L),
    {Sum, Double}.

double([]) -> [];
double([H|T]) -> [2*H|double(T)].

sum([]) -> 0;
sum([H|T]) -> H + sum(T).
```

```
8> lists1:sum_and_double([1,2,3,4]).
{10,[2,4,6,8]}
```

 ©2014 – Joe Armstrong

sum_and_double1

- traverses the list once – gets the answer wrong

```
sum_and_double1(L) ->
    sum_and_double_helper1(L, [], 0).

sum_and_double_helper1([H|T], L, N) ->
    sum_and_double_helper1(T, [2*H|L], N + H);
sum_and_double_helper1([], L, N) ->
    {N, L}.
```

```
8> lists1:sum_and_double1([1,2,3,4]).
{10,[8,6,4,2]}
```

 ©2014 – Joe Armstrong

sum_and_double2

- traverses the list once – gets the answer right

```
sum_and_double2(L) ->
    sum_and_double_helper2(L, [], 0).

sum_and_double_helper2([H|T], L, N) ->
    sum_and_double_helper2(T, [2*H|L], N + H);
sum_and_double_helper2([], L, N) ->
    {N, lists:reverse(L)}.
```

```
8> lists1:sum_and_double2([1,2,3,4]).
{10,[2,4,6,8]}
```

 ©2014 – Joe Armstrong

sum_and_double3

- Renamed the helper function.

```
sum_and_double3(L) ->
    sum_and_double3(L, [], 0).

sum_and_double3([H|T], L, N) ->
    sum_and_double3(T, [2*H|L], N + H);
sum_and_double3([], L, N) ->
    {N, lists:reverse(L)}.
```

```
1> lists1:sum_and_double2([1,2,3,4]).
{10,[2,4,6,8]}
```

 ©2014 – Joe Armstrong

Natural order in lists

- Write the code using accumulators.
- Don't bother if the lists come out in the wrong order.
- Reverse the order at the end.
- `lists:reverse` is a BIF not a function call but it looks like a function call.

 ©2014 – Joe Armstrong

Tail Recursion

- The last thing you do is call another routine
- really “last call optimization”

```
function a(){
   call x
   call y
}

call x is compiled as:
  push address of y
  call x

the call to y (a lastcall) is compiled as
  jmp y
```

 ©2014 – Joe Armstrong

BIFS

- Do things that are impossible in erlang (`atom_to_list(abc)`).
- Do things that are slow in erlang (`lists:reverse/1`).
- Look like erlang function calls

 ©2014 – Joe Armstrong

Apply

- `apply(Mod, Func, [Arg1, Arg2, ..., ArgN])` – same as `Mod:Func(Arg1, Arg2, ..., ArgN)`

```
1> apply(lists, reverse, [[1,2,3,4]]).
[4,3,2,1]
```

 ©2014 – Joe Armstrong

Guards

```
upcase(X) when $a =< X andalso X =< $z ->
   X -$a + $A;
upcase(X) ->
   X.
```

```
1> c(lists1).
{ok,lists1}
2> lists1:upcase($a).
65
3> $a.
97
```

 ©2014 – Joe Armstrong

List Comprehensions

- `[ Expression || Pattern <- List]`

```
1> [lists1:upcase(I) || I <- "hello"].
"HELLO"
2> [{X,Y} || X <- [1,2,3], Y <- [a,b,c]].
[{1,a},{1,b},{1,c},
 {2,a},{2,b},{2,c},
 {3,a},{3,b},{3,c}]
```

 ©2014 – Joe Armstrong

Tuple programs

- Store tree like data structures in tuple

- XML

  The XML `<tag1 a1="abc" a2="def"> ... </tag1>` can be represented as

  `{tag1, [{a1,"abc"},{a2,"def"}], [ ...]}`

```
path_search([Tag|T],[{Tag,_,Children}|_]) ->
   path_search(T, Children);
...
```

 ©2014 – Joe Armstrong

Records

- What do the elements in a tuple mean?

```
X = {person,"zabdog","polgelzipper",42,22,....}
```

```
-record(person,
  {firstname, lastname, footsize, age,
   ....}).

X = #person{age=20, footsize=10}

birthday(X) ->
    Age = X#person.age,
    X#person{age=Age+1}.
```

 ©2014 – Joe Armstrong

Typed Records 1

- We can add types to the fields.
- Can be checked by the dialyzer.
- Informative only:

```
-record(person,
  {firstname :: string(),
   lastname :: string(),
   footsize :: integer(),
   age :: integer(),
   ....}).
```

- What's wrong with this?

 ©2014 – Joe Armstrong

Typed Records 2

```
-type months() :: integer().

-record(person,
  {firstname :: string(),
   lastname :: string(),
   footsize :: integer(),
   age :: months(),
   ....}).
```

 ©2014 – Joe Armstrong

Catch

```
1> X = atom_to_list(1).
** exception error: bad argument
     in function atom_to_list/1
        called as atom_to_list(1)
2> X.
 1: variable 'X' is unbound
3> X = (catch atom_to_list(1)).
{'EXIT',{badarg,
   [{erlang,atom_to_list,[1],[]},
    {erl_eval,do_apply,6,
      [{file,"erl_eval.erl"},{line,573}]},
        ...
```

 ©2014 – Joe Armstrong

Try ... catch, catch .. throw

```
try F()
catch
   exit:... ->
   throw:... ->
   error:... ->
after
   ...
end
```

 ©2014 – Joe Armstrong

++ and --

- X `++` Y is an infix notation for `lists:append(X,Y)`

- Takes time `O(length(X))`.

```
> "abc" ++ "123".
"abc123"
> "abc123" -- "abc".
"123"
> "abc123" -- "123".
"abc"
> "abc123xyz" -- "123".
"abcxyz"
> "abc12xyz" -- "123".
"abcxyz"
```

 ©2014 – Joe Armstrong