

# Programming Distributed Systems

## Introduction to Erlang

Peter Zeller, Annette Bieniusa

AG Softech  
FB Informatik  
TU Kaiserslautern

Summer Term 2018

What is Erlang?



# Erlang

Functional programming language with built-in support for concurrency, distribution and fault tolerance

- Supervised processes as simple and powerful model for error containment and fault tolerance
- Concurrency and message passing as first class language features
- Transparent distribution mechanism
- Erlang runtime environment (BEAM)

What is it suitable for?

# Application areas

Distributed, reliable, soft real-time concurrent systems.

- Telecommunication systems, e.g. controlling a switch or converting protocols.
- Servers for Internet applications, e.g. a mail transfer agent, an IMAP-4 server, an HTTP server.
- Telecommunication applications, e.g. handling mobility in a mobile network or providing unified messaging.
- Database applications which require soft realtime behaviour.

## When to avoid Erlang

- *Short-running computations* because of the startup time of the Erlang VM
- *CPU-intensive work* because the Erlang VM is not optimized for this work
- *Share-memory parallel computation* because there is no shared memory
- *End-user desktop deployments* because it is difficult to have single-file binary executables

Is anybody using it in practise?

## Erlang in production systems

- **Facebook** uses Erlang to power the backend of its chat service, handling more than 100 million active users.
- **Amazon** uses Erlang to implement SimpleDB, providing database services as a part of the Amazon Elastic Compute Cloud (EC2).
- **WhatsApp** uses Erlang to run messaging servers, achieving up to 2 million connected users per server.
- **Yahoo!** uses it in its social bookmarking service, Delicious, which has more than 5 million users and 150 million bookmarked URLs.
- **T-Mobile** uses Erlang in its SMS and authentication systems.
- **Motorola** is using Erlang in call processing products in the public-safety industry.
- **Ericsson** uses Erlang in its support nodes, used in GPRS and 3G mobile networks worldwide.



## Learning resources (selection)

- Learn you some Erlang for Great Good  
<http://learnyousomeerlang.com/content>
- Erlang Website <https://www.erlang.org>
- Erlang Course <https://www.erlang.org/course>
- Erlang Master classes  
<https://www.cs.kent.ac.uk/ErlangMasterClasses/>

# Numbers

## Integers

```
10
-234
16#AB10F
2#110111010
```

## Floats

```
17.368
-56.654
12.34E-10.
```

- `B#Val` is used to store numbers in base `< B >`.
- `$Char` is used for ascii values (example `$A` instead of `65`).

# Atoms

```
abcef  
start_with_a_lower_case_letter  
'Blanks_can_be_quoted'  
'Anything_inside_quotes_\n\012'
```

- Starts with lower case letter (or in single quotes)
- Efficient memory representation

# Tuples

```
{123, bcd}  
{123, def, abc}  
{person, 'Joe', 'Armstrong'}  
{abc, {def, 123}, jkl}  
{}
```

- Used to store a fixed number of items.
- Tuples of any size are allowed.

## Lists

```
[123, xyz]
[123, def, abc]
[{person, 'Joe', 'Armstrong'},
 {person, 'Robert', 'Viriding'},
 {person, 'Mike', 'Williams'}
]
[head|tail]
[x1, x2, x3| tail]
"abcdefghi"
    % becomes - [97,98,99,100,101,102,103,104,105]
""
    % becomes - []
```

- Used to store a variable number of items.
- Lists are dynamically sized.
- “...” is short for the list of integers representing the ascii character codes of the enclosed within the quotes.

# Variables

Abc

A\_long\_variable\_name

AnObjectOrientatedVariableName

- Start with an Upper Case Letter.
- No "funny characters".
- Variables are used to store values of data structures.
- Variables can only be bound once! The value of a variable can never be changed once it has been set (bound).

# Operators

Binary operators:

---

orelse

andalso

==      /=    <    >=    >      ::=    =/=

++

-

+

-

bor

bxor

bsl

or

xor

/

\*

div

rem

band

and

---

Unary operators: + - bnot not

# Complex Data Structures

```
[{ {person, 'Joe', 'Armstrong'},  
  {telephoneNumber, [3,5,9,7]},  
  {shoeSize, 42},  
  {pets, [{cat, tubby},{cat, tiger}]},  
  {children, [{thomas, 5},{claire,1}]}},  
{ {person, 'Mike', 'Williams'},  
  {shoeSize, 41},  
  {likes, [boats, beer]}},  
  ...
```

- Arbitrary complex structures can be created.
- Data structures are created by writing them down (no explicit memory allocation or deallocation is needed etc.).
- Data structures may contain bound variables.



# Pattern Matching

```
A = 10
```

```
  % Succeeds - binds A to 10
```

```
{B, C, D} = {10, foo, bar}
```

```
  % Succeeds - binds B to 10, C to foo and D to bar
```

```
{A, A, B} = {abc, abc, foo}
```

```
  % Succeeds - binds A to abc, B to foo
```

```
{A, A, B} = {abc, def, 123}
```

```
  % Fails
```

```
[A,B,C] = [1,2,3]
```

```
  % Succeeds - binds A to 1, B to 2, C to 3
```

```
[A,B,C,D] = [1,2,3]
```

```
  % Fails
```

## Pattern Matching (Cont)

```
[A,B|C] = [1,2,3,4,5,6,7]
  % Succeeds - binds A = 1, B = 2,
  % C = [3,4,5,6,7]
```

```
[H|T] = [1,2,3,4]
  % Succeeds - binds H = 1, T = [2,3,4]
```

```
[H|T] = [abc]
  % Succeeds - binds H = abc, T = []
```

```
[H|T] = []
  % Fails
```

```
{A,_, [B|_], {B}} = {abc,23,[22,x],[22]}
  % Succeeds - binds A = abc, B = 22
```

- Note the use of "\_", the anonymous (don't care) variable.

## Pattern Matching: Question

Given the following definition:

```
Person = {person,  
          {name,  
            {first, joe},  
            {last, armstrong}},  
          {footsize, 42}}.
```

Write a pattern that extracts the first name from `Person`.

\_\_\_\_\_ = `Person`.

# Case Expression

```
case Expr of  
  Pattern1 -> Expr1;  
  Pattern2 -> Expr2;  
  ...  
  PatternN -> ExprN  
end
```

```
% Example:  
case X > Y of  
  true -> X;  
  false -> Y  
end.
```

# If Expression

```
if  
  Cond1 -> Expr1;  
  Cond2 -> Expr2;  
  ...  
  CondN -> ExprN  
end
```

```
% Example:
```

```
if  
  X > Y -> X;  
  X =< Y -> Y;  
end
```

# Function Calls

```
% different module:  
module:func(Arg1, Arg2, ... Argn)  
  
% same module:  
func(Arg1, Arg2, .. Argn)
```

- Arg1 .. Argn are any Erlang data structures. atoms.
- A function can have zero arguments. (e.g. `date()` - returns the current date).
- Functions are defined within Modules.
- Functions must be exported before they can be called from outside the module where they are defined.

# Module System

```
-module (demo) .  
-export ([double/1]) .
```

```
double(X) ->  
    times(X, 2) .
```

```
times(X, N) ->  
    X * N.
```

- double can be called from outside the module, times is local to the module.
- double/1 means the function double with one argument (Note that double/1 and double/2 are two different functions).

# Function Syntax

Is defined as a collection of clauses.

```
func (Pattern1_1, Pattern1_2, ...) -> Expr1 ;  
func (Pattern2_1, Pattern2_2, ...) -> Expr2 ;  
...  
func (PatternN_1, PatternN_2, ...) -> Expr3 .
```

## Evaluation Rules

- Clauses are scanned sequentially until a match is found.
- When a match is found all variables occurring in the head become bound.
- Variables are local to each clause, and are allocated and deallocated automatically.
- The body is evaluated sequentially.



## Functions (cont)

```
-module (mathStuff).  
-export ([factorial/1, area/1]).  
  
factorial(0) -> 1;  
factorial(N) -> N * factorial(N-1).  
  
area({square, Side}) ->  
    Side * Side;  
area({circle, Radius}) ->  
    % almost :-)  
    3 * Radius * Radius;  
area({triangle, A, B, C}) ->  
    S = (A + B + C)/2,  
    math:sqrt(S*(S-A)*(S-B)*(S-C));  
area(Other) ->  
    {invalid_object, Other}.
```

## Evaluation example

```
factorial(0) -> 1;
```

```
factorial(N) -> N * factorial(N-1)
```

```
> factorial(3)
```

```
  matches N = 3 in clause 2
```

```
  == 3 * factorial(3 - 1)
```

```
  == 3 * factorial(2)
```

```
  matches N =2 in clause 2
```

```
  == 3 * 2 * factorial(2 - 1)
```

```
  == 3 * 2 * factorial(1)
```

```
  matches N = 1 in clause 2
```

```
  == 3 * 2 * 1 * factorial(1 - 1)
```

```
  == 3 * 2 * 1 * factorial(0)
```

```
  == 3 * 2 * 1 * 1 (clause 1)
```

```
  == 6
```

- Variables are local to each clause.
- Variables are allocated and deallocated automatically.

## Traversing Lists

```
average(X) -> sum(X) / len(X).
```

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

```
sum([]) -> 0.
```

```
len([_|T]) -> 1 + len(T);
```

```
len([]) -> 0.
```

- Note the pattern of recursion is the same in both cases.

### Two other common patterns:

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

```
double([]) -> [].
```

```
member(H, [H|_]) -> true;
```

```
member(H, [_|T]) -> member(H, T);
```

```
member(_, []) -> false.
```

## Lists and Accumulators

```
average(X) -> average(X, 0, 0).
```

```
average([H|T], Length, Sum) ->  
    average(T, Length + 1, Sum + H);  
average([], Length, Sum) ->  
    Sum / Length.
```

- Only traverses the list ONCE
- Executes in constant space (tail recursive)
- The variables Length and Sum play the role of accumulators
- N.B. `average([])` is not defined - (you cannot have the average of zero elements) - evaluating `average([])` would cause a run-time error.

## Task: Functions

1. Write functions `f2c(F)` and `c2f(C)` which convert between centigrade and Fahrenheit scales. (hint:  $5(F-32) = 9C$ )
2. Write a function `convert(Temperature)` which combines the functionality of `f2c` and `c2f`. Example:

```
> temp:convert({c,100}).
{f,212}
> temp:convert({f,32}).
{c,0}
```

3. Write a function `mathStuff:perimeter(Form)` which computes the perimeter of different forms. Form can be one of:
  - `{rect, Center, Width, Height}`
  - `{circle, Center, Radius}`
  - `{polynom, Points}`, where `Points` is a List of `{X,Y}` coordinates

Hint: use the `math:pi/0` and `math:sqrt/1` functions

## Guarded Function Clauses

```
factorial(0) -> 1;  
factorial(N) when N > 0 ->  
    N * factorial(N - 1).
```

- The reserved word **when** introduces a guard.
- Fully guarded clauses can be re-ordered.

```
factorial(N) when N > 0 ->  
    N * factorial(N - 1);  
factorial(0) -> 1.
```

- This is NOT the same as:

```
factorial(N) ->  
    N * factorial(N - 1);  
factorial(0) -> 1.
```

- (incorrect!!)

## Examples of Guards

```

number(X)           % X is a number
integer(X)         % X is an integer
float(X)           % X is a float
atom(X)            % X is an atom
tuple(X)           % X is a tuple
list(X)            % X is a list

length(X) == 3     % X is a list of length 3
size(X) == 2       % X is a tuple of size 2.

X > Y + Z           % X is > Y + Z
X == Y              % X is equal to Y
X ::= Y             % X is exactly equal to Y
                    %           (i.e. 1 == 1.0 succeeds but
                    %           1 ::= 1.0 fails)
  
```

- All variables in a guard must be bound.
- See the User Guide for a full list of allowed guards

## Functions as values

```
F = fun (X) -> 2 * X end.
```

```
F(3).
```

```
% patterns:
```

```
G = fun ({a,X}) -> X; ({b, X}) -> 2*X end.
```

```
% function references:
```

```
H = fun math:sqrt/1.
```



# Higher order functions

Functions that take functions as argument.

For example `map`: Applies function `F` on all elements in a list.

```
map(F, [H|T]) -> [F(H) | map(F, T)];  
map(F, []) -> [].
```

Usage:

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

## Standard Library Functions

```
map(Fun, List1) -> List2
```

```
% select elements that match Pred  
filter(Pred, List1) -> List2
```

```
% traverse list from left to right using accumulator  
foldl(Fun, Acc0, List) -> Acc1
```

```
% traverse list from right to left using accumulator  
foldr(Fun, Acc0, List) -> Acc1
```

```
% check if all elements match Pred  
all(Pred, List) -> boolean()
```

```
% check if any element matches Pred  
any(Pred, List) -> boolean()
```

```
% like map, but Fun can return a list of elements  
flatmap(Fun, List1) -> List2
```

## Examples: Using Higher Order Functions

```
% Square all numbers in the list:  
> lists:map(fun(X) -> X*X end, [1,2,3,4,5]).  
[1,4,9,16,25]
```

```
% select even numbers from list  
> lists:filter(fun(X) -> X rem 2 == 0 end, [1,2,3,4,5]).  
[2,4]
```

```
% Sum all elements in list  
% Starts with accumulator 0, and  
% adds each number to the accumulator  
>lists:foldl(fun(X,Acc) -> X+Acc end, 0, [1,2,3,4,5]).  
15
```

## Quiz: Higher order functions

■ `lists.map(fun cook/1, [ 🐮 , 🥔 , 🐔 , 🌽 ])`.

## Quiz: Higher order functions

■ `lists:map(fun cook/1, [ 🐮 , 🥔 , 🐔 , 🌽 ])`  
`[ 🍔 , 🍟 , 🍗 , 🍿 ]`

## Quiz: Higher order functions

- `lists:map(fun cook/1, [ 🐮 , 🥔 , 🐔 , 🌽 ])`.  
`[ 🍔 , 🍟 , 🍗 , 🍿 ]`
- `lists:filter(fun isVeg/1, [ 🍔 , 🍟 , 🍗 , 🍿 ])`.

## Quiz: Higher order functions

- `lists:map(fun cook/1, [ 🐮 , 🥔 , 🐔 , 🌽 ])`.  
`[ 🍔 , 🍟 , 🍗 , 🍿 ]`
- `lists:filter(fun isVeg/1, [ 🍔 , 🍟 , 🍗 , 🍿 ])`.  
`[ 🍟 , 🍿 ]`

## Quiz: Higher order functions

- `lists:map(fun cook/1, [ 🐮 , 🥔 , 🐔 , 🌽 ])`.  
`[ 🍔 , 🍟 , 🍗 , 🍿 ]`
- `lists:filter(fun isVeg/1, [ 🍔 , 🍟 , 🍗 , 🍿 ])`.  
`[ 🍟 , 🍿 ]`
- `lists:foldl(fun feed/2, 😊 , [ 🍔 , 🍟 , 🍗 , 🍿 ])`.



## Quiz: Higher order functions

- `lists:map(fun cook/1, [ 🐮 , 🥔 , 🐔 , 🌽 ])`.  
`[ 🍔 , 🍟 , 🍗 , 🍿 ]`
- `lists:filter(fun isVeg/1, [ 🍔 , 🍟 , 🍗 , 🍿 ])`.  
`[ 🍟 , 🍿 ]`
- `lists:foldl(fun feed/2, 😊 , [ 🍔 , 🍟 , 🍗 , 🍿 ])`.  
`💩`

## Quiz: Higher order functions

Can you implement functions `cook`, `isVeg`, and `feed`, such that the examples work?

```
> C = lists:map(fun cook/1, [cow, potato, chicken, corn]).  
[burger, fries, chicken_drum, popcorn]  
> lists:filter(fun isVeg/1, C).  
[fries, popcorn].  
> lists:foldl(fun feed/2, hungry, C).  
digestion_complete
```

Bonus question: Digestion is complete, only when all 4 different items have been consumed (any order, each at least once).