Sequential Erlang Tutorial

A first contact with the functional programming paradigm

Dr. Eng. Sebastià Vila-Marta

Enginyeria de Sistemes TIC
Universitat Politècnica de Catalunya
http://itic.cat

October 29, 2015

Lecture 1. First steps

1 Introducction

2 First concepts

3 Erlang data types

4 Variables

5 Functions

6 Modules

7 Examples

Why this tutorial? I

FACTS

- Traditional approaches introduce programming through imperative languages (Python, C, C++, Java,...)
- The imperative languages are by far the most usual in industry.

HOWEVER

- This is practical but reductive: there are other interesting paradigms.
- We require the engineers to have a broader knowledge of programming languages. The experience in non-imperative paradigms is fundamental.

Why this tutorial? II

THEN

- It is important to learn some non-imperative language.

WHY THIS TUTORIAL?

- To learn `Erlang`, a functional language.
- To prepare you for future courses where `Erlang` plays an important role.
- To reinforce your programming skills.
- To increase your abstraction capabilities.
- To have to rack your brains.

Erlang: the history

- Erlang is a (almost) pure functional language.
- Designed at Ericsson circa 1986.
- It's name follows the best UNIX naming tradition:
  - Agner Krarup Erlang, the Danish mathematician.
  - Ericsson language.
- The language is designed to cope with distributed control systems requirements.
- It has an industrial strength implementation (efficient tools, good library, etc.)
- Erlang programs are interpreted by a virtual machine.
- Used in some starred projects: ejabberd, whatsapp, RabbitMQ...

A very first program

Example

Design a program that, given $n > 0$, computes $S(n) = \sum_{i=1}^{n} i$.

Here it is!

```
S(1) -> 1;
S(N) -> N + S(N-1).
```

Some technical characteristics

- Functional language:
  - A program is an expression.
  - Functions are the fundamental flow control structure.
  - No iterations.
  - Recursive functions.
- Functions are pure: no side effects.
- Functions can be defined as a sequence of function clauses. Every clause applies to a function subdomain.
- Single assignment: variables can be only assigned once.

Simple programs

Example (Simplest program)

```
1.
```

Example (Slightly more complex program)

```
1 + 1.
```

Example (A far more complex program)

```
A = 1 + 1, A + 2.
```

Some practical issues

- Install `Erlang` into your computer:

  ```
  $ apt-get install erlang
  ```

- In Ubuntu distributions `Erlang` lacks `wx` modules. If you need them install `Erlang` from `http://www.erlang.org`. Follow the instructions that you will find there.
- `Erlang` shell is named `erl`.
  It uses `emacs` bindings ☺.

An `erl` session

```
1> 2 * 4.
8
2> (2 * 4) rem 3.
2
3> A = 2 + 2.
4
4> A.
4
5> A = 6.
** exception error: no match of right hand side value 6
6> A = 4.
4
7> q().
$
```

Erlang simple data types

- Integers

  `-2, 345, 3 rem 2, 4 div 5, $a, 2#101`

- Floating point reals

  `3.56, -4.002, 4.6e-10,`

- Atoms: named constant value. Identifiers begin with lowercase chars or they are enclosed in single quotes.

  `un_atom, red, blue, euro, 'Sebas'`

- Booleans: not a real data type

  `true, false`

Tuples I

- Tuples are fixed size collections of values (like Python tuples).
- Syntax:

```
1> A = {2, 3, new_orleans}.
{2, 3, new_orleans}
```

- There are some predefined functions (BIF's) for tuples:

```
2> tuple_size(A).
3
3> B = setelement(2, A, 4000).
{2, 4000, new_orleans}
4> element(3, B).
new_orleans
```

Tuples II

- It is a current practice to «tag» the tuples using the first element:

```
{jazz_player, 'John Coltrane', saxo}
{jazz_player, 'Ed Thigpen', drums}
```

Lists

- The classical lineal data structures also found in Python.
- They are immutable.
- Syntax:

  []
  [trumpet, drums, piano**]**
  [{0.0,0.0**}**, **[**a, b**]**, figure2**]**

- Strings are lists of integers (char codes):

```
1> A="jazz".
"jazz"
```

Note that the interpreter introduces some sugar!

```
1> A=[65,65].
"AA"
2> B=[$b,$a,$s,$s].
"bass"
```

- Atoms are not strings!

List operations

- There are some predefined operations:

  1> [a,b] ++ [c,d].
  [a,b,c,d]
  2> [a,b,c,d,c] -- [a,c].
  [b,d,c]

- The most interesting operation allows to split a list in head and tail. Let A=[1,2,3], then
  - The head of A is 1.
  - The tail of A is [2,3].
- Some equivalent expresions:

  [1,2,3] = [1|[2,3]]
  [1,2,3,4] = [1,2|[3,4]]
  [[1,2],3,4,5] = [[1,2]|[3,4,5]]

Lists library module

- Erlang library contains a specific module with list operations.
- Examples:

```
1> lists:max([1,2,3]).
3
2> lists:sum([1,2,3]).
6
3> lists:zip([a,b,c],[1,2,3]).
[{a,1},{b,2},{c,3}]
4> lists:reverse([1,2,3,4]).
[4,3,2,1]
5> lists:member(3, [4,3,5]).
true
```

- Look at the docs in `http://www.erlang.org/doc/man/lists.html`.

Boolean operators I

They have some inusual syntax:

==	Equal to
/=	Not equal to
=:=	Exactly equal to
=/=	Exactly not equal to
=<	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than

- Can be applied to any data type.
- Tuples and lists are ordered according to lexicographic order.
- 2==2.0 is **true** but 2=:=2.0 is **false**.

Boolean operators II

and	And
or	Or
not	Not
xor	Exclusive or
andalso	Short-circuit and
orelse	Short-circuit or

Variables

- The variable identifiers begin with capital letters.

  A, A_variable,

- Variables are single assignment: once a variable is bounded to a value it cannot be modified.
- Variables are always local to some function: there are no global variables.

Pattern matching I

- Pattern matching is used to:
  - Assign values to variables.
  - Control the execution flow of programs.
  - Extract values from compound data types.
- Pattern matching syntax:

  `Pattern=Expression.`
- A pattern is a data structure that contains bound and unbound variables. After a succesful matching the variables become bound.

```
1> {A, [b, B]} = {3.1415, [b, {0,0}]}.
{3.1415,[b,{0,0}]}
2> B.
{0,0}
```

- When a matching fails an exception is thrown.

Pattern matching II

- It’s easy to combine with the list split operator:

```
1> [H|T]=[a,b,c,d,e].
[a,b,c,d,e]
2> H.
a
3> T.
[b,c,d,e]
4> [H1,H2|R]=[f|T].
[f,b,c,d,e]
5> H2.
b
6> [A|B]=[w].
[w]
7> B.
[]
```

Pattern matching III

- Patterns can contain a «dont't care» variable _. It means «any value».

```
1> [H|_]=[1,2,3].
[1,2,3]
2> H.
1
3>[_,_|R]=[a,b,c,d,e].
[a,b,c,d,e]
```

Functions

- An Erlang program is a set of functions calling each other.
- Functions are defined as a sequence of clauses. A clause is choosed based on pattern matching:

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

- The order of the clauses is important.

```
a({square, Side}) -> Side * Side;
a({circle, Radius}) -> math:pi() * Radius * Radius;
a(_) -> error.
```

Guards

- Function clauses can be protected with guards.

```
absol(X) when X >= 0 -> X;
absol(X) when X < 0 -> -X.
```

- A clause applies only if the guard holds.
- Guard expressions are constrained:
  - Boolean operators.
  - Arithmetic expressions.
  - Type testing BIF's (is_atom, is_float, etc.)
- Guards can be combined as a disjuntion of conjuntions by using operators "," and ";":

```
f(X) when X<0 -> 1;
f(X) when X>0, X<10 -> 2;
f(X) when X==0; X>10, X<20 -> 3;
```

Modules I

- Erlang allows to organize code into modules.
- A module is a container for functions.
- A module introduces a local scope for functions: public and private functions.
- A module is stored in a file named `<modname>.erl`. File basename and module name must be the same.

Modules II

- Module syntax:

```
-module(example).
-export([absol/1]).

%% public function
absol(X) -> absolute(X).

%% private function
absolute(X) when X<0 -> -X;
absolute(X) -> X.
```

- Public functions of a module are called by prefixing the identifier with module name.

```
A = example:absol(6).
```

Examples I

Problem

Function to add the elements of a list.

```
addlist([]) -> 0;
addlist([E|R]) -> E + addlist(R).
```

Problem

Function to compute the maximum of an integers list.

```
max([E|R]) -> max(R,E).

max([],M) -> M;
max([E|R],M) when E>M -> max(R,E);
max([E|R],M) -> max(R,M).
```

Examples II

Problem

Function to compute an scalar product of two vectors. Vectors are represented as a list of floats.

```
sprod([],[]) -> 0;
sprod([A|R1],[B|R2]) ->
    A*B + sprod(R1,R2).
```

Problem

Design a function to make the following computation: given a list $l_0, l_1, l_2, \ldots$ obtain a list of pairs $(l_0, l_1), (l_1, l_2), (l_2.l_3), \ldots .$

```
mpairs([A,B]) -> [{A,B}];
mpairs([A,B|R]) -> [{A,B}|mpairs([B|R])];
mpairs(_) -> out_of_domain.
```

Examples III

Problem

Design a predicate to test if a string contains the substring "aa".

```
containaa([]) -> false;
containaa([_]) -> false;
containaa([$a, $a |_]) -> true;
containaa([_|R]) -> containaa(R).
```

Examples IV

Problem

Design a function to reverse the elements of a list.

```
rev([]) -> [].
rev([A|R]) -> rev(R) ++ [A].
```

or else:

```
rev(L) -> rev(L,[]).

rev([E|R], A) -> rev(T, [E|A]);
rev([], A) -> A.
```

Examples V

Problem

Design a stack module.

```
-module(stack).
-export([stack/0, push/2, top/1, pop/1, empty/1]).

stack() -> [].

push(S,E) -> [E|S].

top[E|_] -> E.

pop([_|S]) -> S.

empty([]) -> true;
empty(_) -> false.
```