# ERLANG NOTES

Functional Programming // Nando Vieira

Notes taken from Programming Erlang, by Joe Armstrong
Version: 2009-01-05

# TABLE OF CONTENTS

## CHAPTER 1

# ABOUT ERLANG

Erlang is a functional language where the concurrency is tied to the language itself, and not to the operating system. The language uses message exchanging to interact with parallel processes without creating locks or synchronization methods. A simple Erlang program can create thousands to millions of lightweight processes that can be executed on single or multi-core processors, and even on a network of processors.

Erlang was designed for programming fault-tolerant systems. It was originally developed in the Computer Science Laboratory at the Swedish Telecom company Ericsson.

Erlang is about:

- Concurrency
- Distribution
- Fault tolerance
- Functional programming
- Speeding up applications on multi-core CPUs

CHAPTER 2

# GETTING STARTED

## Installing (Mac OS X)

Make sure you have XCode installed. Download the most recent Erlang source code from **http://www.erlang.org/download.html**.

```
wget -c http://www.erlang.org/download/otp_src_R12B-5.tar.gz
tar -xzf otp_src_R12B-5.tar.gz
cd opt_src_R12B-5
./configure --prefix=/usr/local
make
sudo make install
```

## Starting the shell

Codes can be typed right into the **shell**; to start it, just execute the command `erl`.

```
$ erl
Erlang (BEAM) emulator version 5.5.1 [source] [async-threads:0] [hipe]
Eshell V5.5.1 (abort with ^G)
```

Only expressions can be executed; you can't type things started by hyphens (such as `-module` or `-export`).

## About shell

The Erlang shell has a built-in line editor.

```
Command                 Description
-------------------------------------------------------
Ctrl+A                  Beginning of line
Ctrl+E                  End of line
Ctrl+F or right arrow   Forward character
Ctrl+B or left arrow    Backward character
Ctrl+P or up arrow      Previous line
Ctrl+N or down arrow    Next line
Ctrl+T                  Switch last two characters
Tab                     Expand module or function name
```

When isn't responding, type `Ctrl+G`; you can then type some commands. Type `h` to view the available options.

```
$ erl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] [async-threads:0] [kernel-poll:false]

Eshell V5.6.5  (abort with ^G)
1>
^G
User switch command
 --> h
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q                 - quit erlang
  ? | h             - this message
```

To clear all variable bindings, just execute the command `f()`.

```
1> X = 1.
1
2> X = 2.
**exception error: no match of right hand side value 2
3> f().
ok
4> X = 2.
2
```

You can clear a specific variable binding:

```
5> f(X).
6> X.
*1: variable 'X' is unbound
```

# Simple expressions

```
1> 1 + 1.
2
2> (3 + 2) * 4.
20
```

**Note:** The dialog starts with `>1`; this means that a new Erlang shell has been started. When an expression has been finished, the dialog will be increased (like `2>` and so on).

Calculation with large numbers:

```
3> 1234567890 * 987654321 * 9192939495969798.
1120919300402860579707820613605662 0
```

Integers using base 16 and base 32 notation:

```
4> 16#cafe * 32#sugar.
1577682511434
```

# Variables

Variables must start with an uppercase letter.

```
1> X = 1.
1
2> Name = "Nando Vieira".
"Nando Vieira"
```

Erlang has **single assignment variables**; this means that you can set the value once, changing from being **unbound** to **having a value**.

```
1> X = 1.
1
2> X = 2.
**exception error: no match of right hand side value 2
```

The `=` sign is a **pattern matching** operator, which behaves like assignment when you have an unbound variable.

The **scope** of a variable is the place where it was set. If `X` is used inside a function, then this function is its scope. If `X` occurs in different function, the all the values of `X` are different.

## Pattern Matching

In Erlang, `=` denotes a **pattern matching**; `Lhs = Rhs` means: evaluate the right side (`Rhs`) and then match the result against the pattern on the left side (`Lhs`).

A variable, such as `X`, is a simple form of pattern. The first time a variable is assigned, as in `X = SomeExpression`, Erlang binds `X` to the value of `SomeExpression`, so the statement becomes valid. The expression `X = AnotherExpression` will succeed only if `SomeExpression` and `AnotherExpression` are identical.

```
1> X = (2+4).
6
2> Y = 10.
10
3> X = 6.
6
4> X = Y.
**exception error: no match of right hand side value 10
5> Y = 10.
10
6> Y = 4.
**exception error: no match of right hand side value 4
7> Y = X.
**exception error: no match of right hand side value 6
```

The expression `2+4` was assigned to `X`, and the answer was `6`. Then `10` was assigned to `Y`. When we try to evaluate `X = Y`, the match fails and then error message is printed.

# Floating-Point Numbers

```
1> 5/3.
1.6666666666666667
2> 5 div 2.
2
3> 5 rem 3.
2
4> 4/2.
2.0
5> Pi = 3.14159.
3.14159
6> R = 5.
```

```
5
7> Pi * R * R.
78.53975
```

On Erlang, the operator `/` always returns a float. Floating-point numbers must have a decimal point followed by at least one digit. The expressions `N div M` and `N rem M` are used for integer division and remainder.

# Atoms

**Atoms** are constants that represents non-numerical values like `true`, `false`, `ok` and `error`; they're similar to Ruby's symbols. They're started by lowercase letters, followed by alphanumeric characters, underscore or at sign, like `december`, `test123`, `some@host`, `a_long_name`. Atoms can also be quoted with a single quotation mark; then you can create atoms like `'Name'`, `'+'`, `'*'`, `'an atom with spaces'`, `'name'`.

The value of an atom is the atom itself, which will be printed on the shell.

```
1> name.
name
2> name = 'name'.
name
```

# Tuples

**Tuples** are immutable lists. They're used to group a fixed number of items into a single entity. Tuples doesn't have keys, so is recommended to use an atom as the first element, describing the tuple.

```
P = {person,
        {name, "Nando Vieira"}
        {blog, "http://simplesideias.com.br"}}.
```

## Extracting values from tuples

To extract values from tuples you need to do pattern matching. Let's extract the name and blog values.

```
{_, {_, Name}, {_, Blog}} = P.
```

The _ symbol is called **anonymous variable**; you can have several occurrences of it in the same pattern.

# Lists

**Lists** can store variable number of items. The first element of a list is called **head**. If you remove the **head** from the list, you have the **tail** of the list; that is, the remaining items. Here's what a list looks like:

```
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}].
```

If T (tail) is a list, then H|T (head and tail) is also a list. The vertical bar | separates the head from its tail. When creating lists like [...|T], make sure you create **properly formed** lists; that is, T needs to be a list. Otherwise you'll have a **improper list** that won't work for most library functions.

You can add more elements to the list with the syntax [E1, E2, ..., En|T].

```
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}].
ThingsToBuy1 = [{oranges, 12}, {newspaper, 1}|ThingsToBuy]
```

## Extracting elements from a list

As always, you need to do some pattern matching operation.

```
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}].
[Apples|RemainingThingsToBuy] = ThingsToBuy.
```

The matching above will bind `Apples => {apples, 10}` and `RemainingThingsToBuy => [{milk, 3}, {pears, 6}]`.

You can extract several values at once and set a new variable with the remaining items.

```
ThingsToBuy = [{apples, 10}, {pears, 6}, {milk, 3}, {newspaper, 1}].
[Buy1, Buy2|RemainingThingsToBuy] = ThingsToBuy.
```

Now, `Buy1 => {apples, 10}`, `Buy2 => {pears, 6}` and `RemainingThingsToBuy => [{milk, 3}, {newspaper, 1}]`.

# Strings

**Strings** are just a list of integers quoted with `"`. Here's a sample:

```
1> Name = "Nando Vieira".
"Nando Vieira"
2> NickName = [102, 110, 97, 110, 100, 111].
"fnando"
3> [1, 2, 3].
[1,2,3]
```

The third expression was printed as it is because `1`, `2` and `3` are not printable characters. To know which integer represents a character, just use the `$` symbol like in `$a`.

```
1> $a.
97
2> $A.
65
```

If you mix printable with non-printable characters, Erlang won't convert the list.

```
1> [$a, 1].
[97,1]
```

**Note:** In Erlang, remember that single quoted strings are just **atoms**; so `'a'` is not `"a"`.

You can even use a different character set in strings. The Swedish name `Håkan` will be encoded as `[72,229,107,97,110]`.

# Comments

Comments in Erlang start with `%` and extend to the end of the line. There are no block comments.

```
% this is just a comment
X = "Some string".
```

Sometimes `%%` is used to comment out the code; this is recognized by Emacs and enable the automatic indentation of commented lines.

CHAPTER 3

# SEQUENTIAL PROGRAMMING

## Modules

**Modules** store all the functions you write and need to be saved in `.erl` files. All modules need to be compiled to '.beam'[1] files before you can use it.

Create a new file `geometry.erl`.

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

area({rectangle, Width, Height})  -> Width * Height;
area({circle, Radius})            -> 3.14159 * Radius * Radius.
```

The keyword `module` defines a new module named `geometry`. The `export` clause says what functions are being exported; the `[area/1]` notation defines the function name and how many arguments are expected (*arity*). To export several functions, you can do this:

```
-module (some_module).
-export([func1/1]).
-export([func2/1]).
```

Or this:

1. Beam is short for Bogdan's Erlang Abstract Machine; Bogumil (Bogdan) Hausman wrote an Erlang compiler in 1993 and designed a new instruction set for Erlang.

```
-module (some_module).
-export([func1/1, func2/1]).
```

If you want to export all module functions, you can use `-compile (export_all)`.

```
-module (some_module).
-compile (export_all).
```

The `area` keyword defines the function name. In the example above we have two `area` *clauses*; think of it as **method overriding** based on pattern matching. Clauses must be separated by semicolons (`;`), and the final clause must be terminated by a dot-whitespace.

Let's try it out. From the shell, execute the command `c(geometry)`. You'll probably receive this error:

```
1> c(geometry).
./geometry.erl:none: no such file or directory
error
```

This error will occur if you're trying to compile a module but the file is not present; you must be in the wrong directory or the file hasn't been created yet. Check the current directory with `pwd()`.

```
1> pwd().
/Users/fnando
ok
```

You can quit the shell and go to the source directory, or you can execute `cd(<path>)`.

```
2> cd("/Users/fnando/Sites/github/notes/erlang/code").
/Users/fnando/Sites/github/notes/erlang/code
ok
```

Try to compile the module once again.

```
3> c(geometry).
{ok,geometry}
```

The compiler has returned `{ok, geometry}`, which means that the compiler has succeed and the module `geometry` has been compiled and loaded. You can now execute the functions you created:

```
4> geometry:area({rectangle, 10, 5}).
50
5> geometry:area({circle, 6}).
113.09723999999999
```

Let's extend our module by adding a square to our geometry module.

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

area({rectangle, Width, Height})  -> Width * Height;
area({circle, Radius})            -> 3.14159 * Radius * Radius;
area({square, X})                 -> X * X.
```

The order of the clauses doesn't matter; this is possible only because the patterns are mutually exclusive. Remember to set the correct order you expect to be executed when you cannot create exclusive patterns.

# Back to Shopping

Let's go back to the shopping list below:

```
[{oranges,4}, {newspaper,1}, {apples,10}, {pears,6} ,{milk,3}]
```

Suppose we'd like to calculate total price. Create the module `shop` for this.

```
-module (shop).
-export ([cost/1]).

cost(oranges)     -> 2;
cost(newspaper)   -> 3;
cost(apples)      -> 4;
cost(pears)       -> 5;
cost(milk)        -> 6.
```

The function `cost` is composed by 5 clauses. Let's test it:

```
1> c(shop).
{ok,shop}
2> shop:cost(oranges).
5
3> shop:cost(apples).
2
4> shop:cost(xbox360).
**exception error: no function clause matching shop:cost(xbox360)
```

Now, we need to calculate the total price from a given list. Here's how you can create the `total` function. Add the code below to the `shop.erl` file.

```
-module (shop).
-export ([cost/1]).
-export ([total/1]).

cost(oranges)     -> 2;
cost(newspaper)   -> 3;
cost(apples)      -> 4;
cost(pears)       -> 5;
cost(milk)        -> 6.

total([])         -> 0;
total([{Product, Quantity}|T]) ->
  cost(Product) * Quantity + total(T).
```

The `total` function expects a list as argument; if an empty list is provided, then we return `0`; otherwise we retrieve the first product tuple, do some basic calculation (`cost(Product) * Quantity`) and sum the total from the remaining items (tail).

Recompile the `shop` module and calculate a shopping list:

```
1> c(shop).
{ok,shop}
2> Buy = [{apples, 2}, {newspaper, 1}, {milk, 3}].
[{apples,3},{newspaper,4},{milk,5}]
3> shop:total(Buy).
29
```

# Functions with the same name and different arity

The **arity** of a function is the number of arguments that the function expects. In Erlang, functions with the same name but different arity are **entirely** different. This is what Erlang programmers do to create **auxiliary functions**. Create a new file named `misc.erl`.

```
-module (misc).
-export ([sum/1]).

% main function
sum(L)  -> sum(L, 0).

% auxiliary functions
sum([], N)      -> N;
sum([H|T], N)   -> sum(T, H+N).
```

Note that we are exporting only the function `sum(L)`.

# Funs

**Funs** are just anonymous functions.

```
1> Double = fun(X) -> 2*X end.
#Fun<erl_eval.6.13229925>
2> Double(3).
6
3> Double().
**exception error: interpreted function with arity 1 called with no arguments
```

They can have any number of arguments.

```
4> UserInfo = fun(Name, Age) -> io:format("~s has ~p years-old~n", [Name, Age]) end.
#Fun<erl_eval.12.113037538>
5> UserInfo("fnando", 29).
fnando has 29 years-old
ok
```

They can have multiple clauses.

```
6> Temp =     fun({c, C}) -> {f, 32 + C*9/5};
6>               ({f, F}) -> {c, (F-32)*5/9}
6>            end.
7> Temp({c, 33}).  % from C to F
8> Temp({f, 115}). % from F to C
```

Functions can accept funs as arguments or return funs as result. They're called **higher-order functions**.

## Functions that accept funs as arguments

The standard library `lists` have many functions whose arguments are funs. Here's a example using the function `lists:map`:

```
1> Double = fun(N) -> N * 2 end.
#Fun<erl_eval.6.13229925>
2> lists:map(Double, [1,2,3,4,5]).
[2,4,6,8,10]
```

Here's another example using the function `lists:filter`:

```
1> Even = fun(N) -> (N rem 2) =:= 0 end.
#Fun<erl_eval.6.13229925>
2> Even(1).
false
3> Even(2).
true
4> lists:map(Even, [1,2,3,4,5]).
[false,true,false,true,false]
5> lists:filter(Even, [1,2,3,4,5,6]).
[2,4,6]
```

## Functions that return funs

Functions can also return *funs*.

```
Fruits = [apple, pear, orange].
MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.
IsFruit = MakeTest(Fruits).     % #Fun<erl_eval.6.13229925>
IsFruit(lemon).                 % false
IsFruit(orange).                % true
```