Seven Languages in Seven Weeks

Correl Roush

July 29, 2015

Created	1986
Author	Joe Armstrong, Robert Virding, and Mike Williams

A functional language built for concurrency and reliability.

http://www.erlang.org/

- Lightweight processes
- No shared state
- Transparently distributed message passing
- Processes as actors

- "Let it crash" approach to error handling
- Process monitoring and restarting
- Hot code loading

The whole notion of "nondefensive" programming and "Let It Crash," which is the mantra of Erlang programming, is completely the opposite of conventional practice, but it leads to really short and beautiful programs. – Dr. Joe Armstrong

Erlang is the first of our functional languages:

- Your programs are going to be built entirely out of functions, with no objects anywhere.
- Those functions will usually return the same values, given the same inputs.
- Those functions will not usually have side effects, meaning they will not modify program state.
- You will only be able to assign any variable once.

```
% This is a comment

2 + 2.                                      % 4
2 + 2.0.                                    % 4
"string".                                   % "string"
[1, 2, 3].                                  % [1,2,3]
[72, 97, 32, 72, 97, 32, 72, 97].           % "Ha Ha Ha"
```

So, a `String` is really a `List`, and Agent Smith just laughed at your mamma.

```
variable = 4.
%% ** exception error: no match of right hand side value 4
```

This error message is really a reference to Erlang's pattern matching. It's breaking because `variable` is an atom. Variables must start with an uppercase letter.

```
Var = 1.
%% 1

Var = 2.
%% ** exception error: no match of right hand side value 2
```

As you can see, variables begin with a capital letter, and they are immutable. You can assign each value only once.

Example (Atoms)

```
red.
%% red
Pill = blue.
%% blue
Pill.
%% blue
```

Example (Lists)

```
[1, 2, 3].
%% [1,2,3]
[1, 2, "three"].
%% [1,2,"three"]
List = [1, 2, 3].
%% [1,2,3]
```

Example (Tuples)

```
{one, two, three}.
%% {one,two,three}
Origin = {0, 0}.
%% {0,0}
```

Example (Tuples as hashes)

```
{comic_strip,
 {name, "Calvin and Hobbes"},
 {character, "Spaceman Spiff"}}.
```

ADDENDUM

Data Structures

Records Provide structure and syntax around named tuples (tuples where the first element is an atom describing the contents of the tuple, e.g. `{alias, "Thomas A. Anderson", "Neo"}`).

Maps A new mapping type with its own syntax, added in Erlang/OTP 17.0. Allows keys of any type.

Example (Record)

```
-record(comic_strip,
        {name, character}).

Strip = #comic_strip{
           name = "Calvin and Hobbes",
           character = "Spaceman Spiff"}.
%% {comic_strip, "Calvin and Hobbes",
%%               "Spaceman Spiff"}

Strip#comic_strip.name.
%% "Calvin and Hobbes"
```

Example (Map)

```
Strip = #{name => "Calvin and Hobbes",
          character => "Spaceman Spiff"}.
%% #{name => "Calvin and Hobbes",
%%   character => "Spaceman Spiff"}.

maps:get(name, Strip).
%% "Calvin and Hobbes"
```

Property Lists Ordinary lists containing entries in the form of either tuples, whose first elements are keys used for lookup and insertion, or atoms, which work as shorthand for tuples `{Atom, true}`.
Property lists are useful for representing inherited properties, such as options passed to a function where a user may specify options overriding the default settings, object properties, annotations, etc.

Dictionaries Implements a `Key - Value` dictionary. The representation of a dictionary is not defined.

Example (Proplist)

```erlang
Strip = [{name, "Calvin and Hobbes"},
         {character, "Spaceman Spiff"}].
%% [{name, "Calvin and Hobbes"},
%%  {character, "Spaceman Spiff"}]

proplists:get_value(name, Strip).
%% "Calvin and Hobbes"
```

Example (Dict)

```erlang
Strip = dict:from_list(
          [{name, "Calvin and Hobbes"},
           {character, "Spaceman Spiff"}]).
%% ...

dict:fetch(name, Strip).
%% "Calvin and Hobbes"
```

```erlang
Person = {person,
          {name, "Agent Smith"},
          {profession, "Killing Programs"}}.

{person, {name, Name}, {profession, Profession}} = Person.

Name.
%% "Agent Smith"

Profession.
%% "Killing Programs"
```

Erlang will match up the data structures, assigning variables to the values in the tuples.

```
[Head | Tail] = [1, 2, 3].
%% Head = 1
%% Tail = [2,3]

[One, Two|Rest] = [1, 2, 3].
%% One = 1
%% Two = 2
%% Rest = [3]

[X|Rest] = [].
%% ** exception error: no match of right hand side value []
```

Example (Packing)

```
W = 1.
X = 2.
Y = 3.
Z = 4.
All = <<W:3, X: 3, Y:5, Z:5>>.
%% <<"(d">>
```

Example (Unpacking)

```
<<A:3, B:3, C:5, D:5>> = All.
%% A = 1
%% B = 2
%% C = 3
%% D = 4
```

```erlang
-module(basic).
-export([mirror/1]).

mirror(Anything) -> Anything.
```

Listing 1: basic.erl

```erlang
-module(matching_function).
-export([number/1]).

number(one)   -> 1;
number(two)   -> 2;
number(three) -> 3.
```

Listing 2: matching_function.erl

```erlang
-module(yet_again).
-export([another_factorial/1,
         another_fib/1]).

another_factorial(0) ->
    1;
another_factorial(N) ->
    N * another_factorial(N - 1).

another_fib(0) ->
    1;
another_fib(1) ->
    1;
another_fib(N) ->
    another_fib(N - 1) + another_fib(N - 2).
```

Listing 3: yet_again.erl

You're going to learn to apply functions to lists that can quickly shape the list into exactly what you need. Do you want to turn a shopping list into a list of prices? What about turning a list of URLs into tuples containing content and URLs? These are the problems that functional languages simply devour.

```erlang
Animal = "dog".
case Animal of
    "dog" -> underdog;
    "cat" -> thundercat
end.
%% underdog

case Animal of
    "elephant" -> dumbo;
    _ -> something_else
end.
%% something_else
```

```erlang
X = 0.

if
    X > 0 -> positive;
    X < 0 -> negative
end.
%% ** exception error: no true branch found when evaluating an if expression

if
    X > 0 -> positive;
    X < 0 -> negative;
    true  -> zero
end.
%% zero
```

```
Negate = fun(I) -> -I end.
%% #Fun<erl_eval.6.13229925>

Negate(1).
%% -1
Negate(-1).
%% 1
```

```erlang
Numbers = [1, 2, 3, 4].
Print = fun(X) -> io:format("~p~n", [X]).

lists:foreach(Print, Numbers).
%% 1
%% 2
%% 3
%% 4
%% ok

lists:map(fun(X) -> X + 1 end, Numbers).
%% [2,3,4,5]

Small = fun(X) -> X < 3 end.
lists:filter(Small, Numbers).
%% [1,2]
lists:all(Small, [0, 1, 2]).
%% true
lists:all(Small, [0, 1, 2, 3]).
%% false
```

```erlang
lists:any(Small, [0, 1, 2, 3]).
%% true
lists:any(Small, [3, 4, 5]).
%% false

lists:any(Small, []).
%% false
lists:all(Small, []).
%% true

lists:takewhile(Small, Numbers).
%% [1,2]
lists:dropwhile(Small, Numbers).
%% [3,4]
lists:takewhile(Small, [1, 2, 1, 4, 1]).
%% [1,2,1]
lists:dropwhile(Small, [1, 2, 1, 4, 1]).
%% [4,1]
```

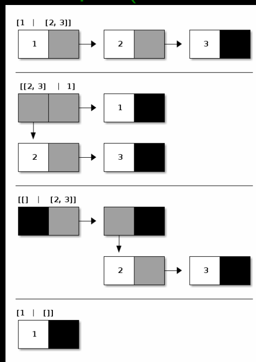
```erlang
Numbers.
%% [1,2,3,4]

Adder = fun(ListItem, SumSoFar) -> ListItem + SumSoFar end.
InitialSum = 0.

lists:foldl(Adder, InitialSum, Numbers).
%% 10
```

```
double_all([]) -> [];
double_all([First|Rest]) -> [First + First|double_all(Rest)].
```

Example (Erlang)

```
[1 | [2, 3]].
%% [1,2,3]

[[2, 3] | 1].
%% [[2,3]|1]

[[] | [2, 3]].
%% [[],2,3]

[1 | []].
%% [1]
```

Example (Box-and-pointer Diagrams)



List comprehensions provide a succinct syntax combining mapping, filtering, and pattern matching.

- Take the form `[Expression || Clause1, Clause2, ..., ClauseN]`.
- List comprehensions can have an arbitrary number of clauses.
- The clauses can be generators or filters.
  - A filter can be a boolean expression or a function returning a boolean.
  - A generator, of the form `Match <- List`, matches a pattern on the left to the elements on the right.

LIST COMPREHENSION EXAMPLES

```
Fibs = [1, 1, 2, 3, 5].
Double = fun(X) -> X * 2 end.
[Double(X) || X <- Fibs].
%% [2,2,4,6,10]

Cart = [{pencil, 4, 0.25}, {pen, 1, 1.20}, {paper, 2, 0.20}].
WithTax = [{Product, Quantity, Price, Price * Quantity * 0.08} ||
              {Product, Quantity, Price} <- Cart].
%% [{pencil,4,0.25,0.08},{pen,1,1.2,0.096},{paper,2,0.2,0.032}]
Cat = [{Product, Price} || {Product, _, Price} <- Cart].
%% [{pencil,0.25},{pen,1.2},{paper,0.2}]

[X || X <- [1, 2, 3, 4], X < 4, X > 1].
%% [2,3]

[{X, Y} || X <- [1, 2, 3, 4], X < 3, Y <- [5, 6]].
%% [{1,5},{1,6},{2,5},{2,6}]
```

I didn't say that it would be easy. I just said that it would be the truth. You have to let it all go. Fear, doubt, and disbelief. Free your mind.

- Spawning a process with `spawn`
- Sending a message with `!`
- Receiving a message with `receive`

```
-module(translate).
-export([loop/0]).

loop() ->
    receive
        "casa" ->
            io:format("house~n"),
            loop();
        "blanca" ->
            io:format("white~n"),
            loop();
        _ ->
            io:format("I don't understand.~n"),
            loop()
    end.
```

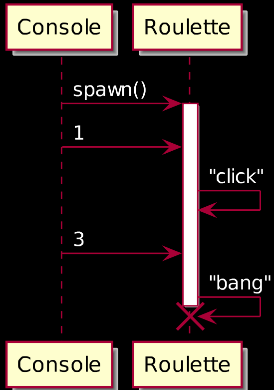
Example (Usage)

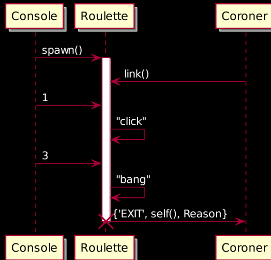
```
Pid = spawn(fun translate:loop/0).
Pid ! "casa".
%% "house"
%% "casa"
```

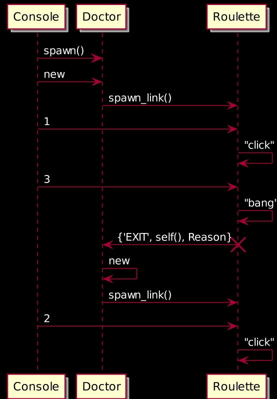
```erlang
-module(translate_service).
-export([loop/0, translate/2]).

loop() ->
    receive
        {From, "casa"} ->
            From ! "house",
            loop();
        {From, "blanca"} ->
            From ! "white",
            loop();
        {From, _} ->
            From ! "I don't understand.",
            loop()
end.

translate(To, Word) ->
    To ! {self(), Word},
    receive
        Translation -> Translation
    end.
```

Example (Usage)

```erlang
Translator = spawn(fun translate_service:loop/0
%% <0.38.0>>
translate_service:translate(Translator, "blanca
%% "white"
translate_service:translate(Translator, "casa")
%% "house"
```

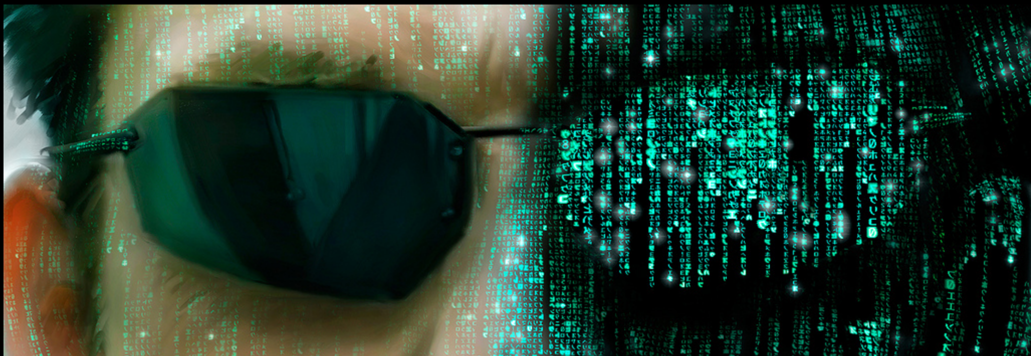

Example (Linking)
Console
Roulette
spawn()
1
"click"
3
"bang"
Console
Roulette


Example (Coroner)
Console
Roulette
Coroner
spawn()
link()
1
"click"
3
"bang"
{'EXIT', self(), Reason}
Console
Roulette
Coroner


Example (Doctor)
Console
Doctor
Roulette
spawn()
new
spawn_link()
1
"click"
3
"bang"
{'EXIT', self(), Reason}
new
spawn_link()
2
"click"
Console
Doctor
Roulette

OTP

The Open Telecom Platform



There's way too much information to decode in the Matrix. You get used to it, though. OTP does the translating. I don't even see the code. All I see is supervisor, gen_server, release...

WRAPPING UP ERLANG/OTP: STRENGTHS

- Reliable
- Lightweight, share-nothing processes
- OTP, the enterprise libraries
- Let It Crash

- Niche
- Syntax
- Integration

Erlang does seem to be gathering momentum because it solves the right problems in the right way at the right time.