



CREER...CREAR...CRECER

Departamento de Computación
FCEFQN, Universidad Nacional de Río Cuarto

Tema: Introducción a Erlang

Miércoles 29 de octubre de 2014
Autores: Astorga Darío - Cucatti Gaston



Nociones generales

Que es Erlang?

Erlang es un lenguaje de programación funcional que soporta concurrencia y tiene un sistema de ejecución que incluye una máquina virtual y bibliotecas.

Fue diseñado en la compañía *Ericsson* con el fin de desarrollar software en el área de Telecomunicaciones realizando aplicaciones distribuidas, tolerantes a fallos, soft-real-time y de funcionamiento ininterrumpido.

Originalmente, el propietario de *Erlang* era la compañía *Ericsson*, pero fue cedido como open source en 1998.

Nociones generales

Características más destacadas de erlang

- Lenguajes funcional
- Soporta concurrencia.
- Tolerancia a fallas.
- Evaluación estricta.
- Asignación única.
- Tipado dinámico.
- Cambio de código en tiempo de ejecución (éste se puede cambiar sin parar el sistema).
- Brinda la posibilidad de conectar con código C, Java y otros lenguajes.

Nociones generales

Podemos identificar una función en un programa en *Erlang* mediante el **nombre** y la **aridad** de la función, dicha aridad es el número de datos de entrada que se permitirán en la función. Es decir, la función volumen/3, permite solo 3 datos de entrada mientras que la función area/2, permite solo 2 datos de entrada.

Algo importante que hay que mencionar es que pueden coexistir sin ningún problema dos funciones con el mismo nombre y diferente aridad, como por ejemplo: **multiplicación/2** y **multiplicación/3**, ya que las dos funciones son diferentes.

Nociones generales

Erlang es un poco diferente a otros lenguajes funcionales como Haskell.

En *Erlang* primero se tiene que **crear un módulo** con extensión `erl`. los módulos son las unidades mínimas de compilación.

Cada módulo tiene un nombre único en el programa y se almacena en un archivo y dicho archivo tiene el mismo nombre del módulo.

Cada instrucción termina con un punto, sino se hace esto el compilador no se da cuenta cuándo acaba la instrucción.

En el archivo creado hay que escribir: `-module(archivo.erl)`. Donde `archivo` es el nombre del documento que se creó.

A continuación un ejemplo donde en la primera línea del archivo contiene la definición del módulo y en la segunda línea las funciones que se exportan, las funciones que no sean exportadas, no podrán ser utilizadas fuera de este mismo módulo.

Nociones generales

-module(nombre_modulo)

-export([funcion1/2, funcion2/1, funcion3/4]).

Mediante la instrucción: `-export([función/1])` se exportan las funciones que se pueden usar por la consola. Recordar que el número que acompaña el nombre de la función dice cuántos parámetros acepta.

Programa Hola Mundo

```
+ x hola.erl
1 -module(hola).
2 -export([hello_world/0]).
3
4 hello_world() ->
5     "hello world".
```

```
x dario@N53SV: ~/Material Ayudantia Comparativo/Erlang
+ x ...tia Comparativo/Erlang
dario@N53SV:~/Material Ayudantia Comparativo/Erlang$ erl
Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.8.5 (abort with ^G)
1> c(hola.erl).
{ok,hola}
2> hola:hello_world().
"hello world"
3> 
```

Programa Factorial

```
-module(factorial).  
-export([factorial/1]).
```

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

Por consola:

```
1> c(factorial).  
    {ok,factorial}  
2> factorial:factorial(20).  
    2432902008176640000  
3>
```


Programa Operaciones

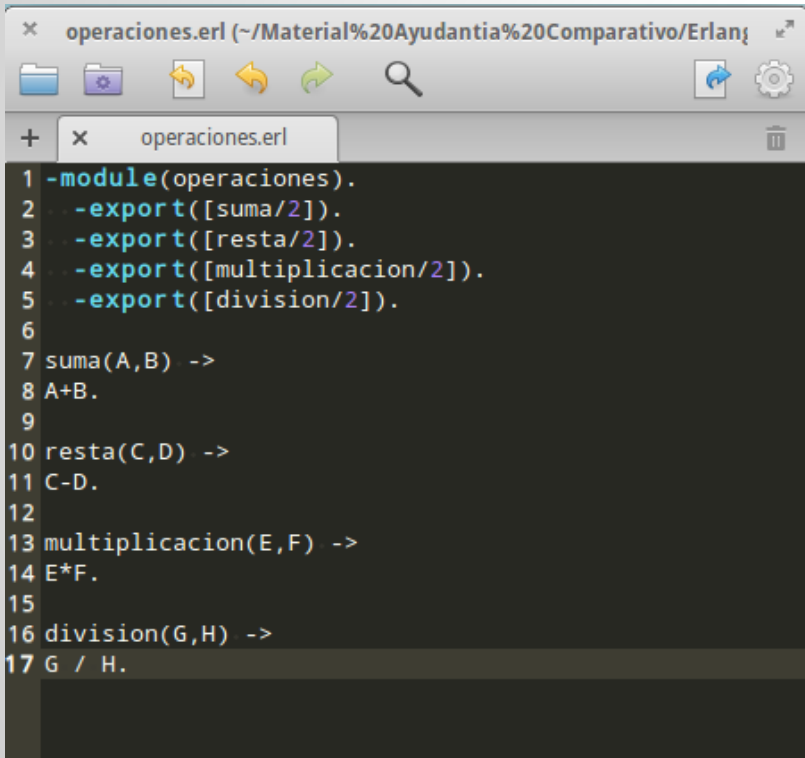
La extensión de los programas en Erlang es **.erl**

Además el nombre del módulo del programa deberá ser el mismo nombre del programa, para este caso el nombre del módulo es **“operaciones”**.

Al final de cada módulo y de cada función exportada deberá ir un punto “.”

Las funciones no pueden llevar acentos. Cada función tiene una aridad de 2, es decir cada función sólo puede recibir 2 datos de entrada.

Para salir de la ejecución del programa se escribe **halt()**.



```
1 -module(operaciones).
2 -export([suma/2]).
3 -export([resta/2]).
4 -export([multiplicacion/2]).
5 -export([division/2]).
6
7 suma(A,B) ->
8 A+B.
9
10 resta(C,D) ->
11 C-D.
12
13 multiplicacion(E,F) ->
14 E*F.
15
16 division(G,H) ->
17 G / H.
```

Programa Operaciones

Para compilar este programa, iremos a la terminal y escribiremos erl (si ya lo tenemos instalado), y después tecleamos lo siguiente:

c(operaciones).

Y si el programa esta totalmente bien, si ningún error, entonces nos aparecerá:

{ok,operaciones}

Y ahora sí, ya podemos introducir datos, para que sean realizadas las distintas operaciones.

Por ejemplo, para una suma, se deberá introducir, lo siguiente:

operaciones:suma(2,4).

Concurrencia Introducción

Una de las principales razones para el uso de *Erlang* en lugar de otros lenguajes funcionales es la capacidad de *Erlang* para manejar la concurrencia y programación distribuida.

El soporte de **concurrencia**, se dice que es la mayor fortaleza de *Erlang*, y como ya lo mencionamos anteriormente, la concurrencia es cuando se llevan a cabo o se ejecutan varios procesos al mismo tiempo.

Concurrencia Nociones Básicas

Creación de Procesos

No hay jerarquía inherente entre procesos.

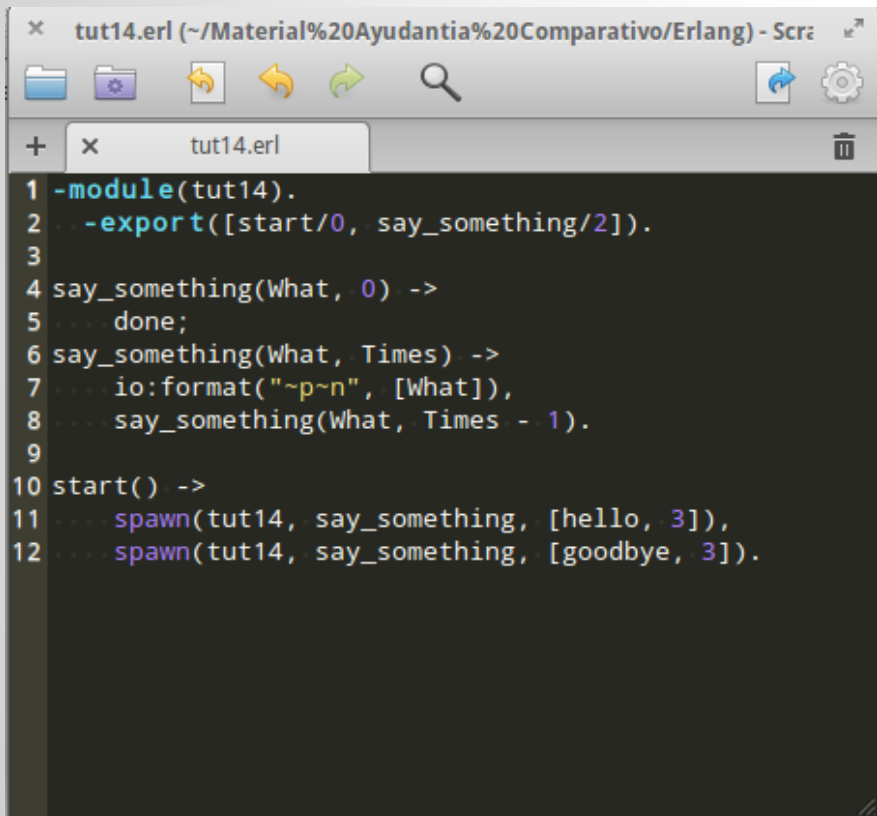
spawn/3 crea e inicia la ejecución de un nuevo proceso.

spawn/3 crea un nuevo proceso concurrente para evaluar la función y devuelve el PID (identificador de proceso) del proceso recién creado.

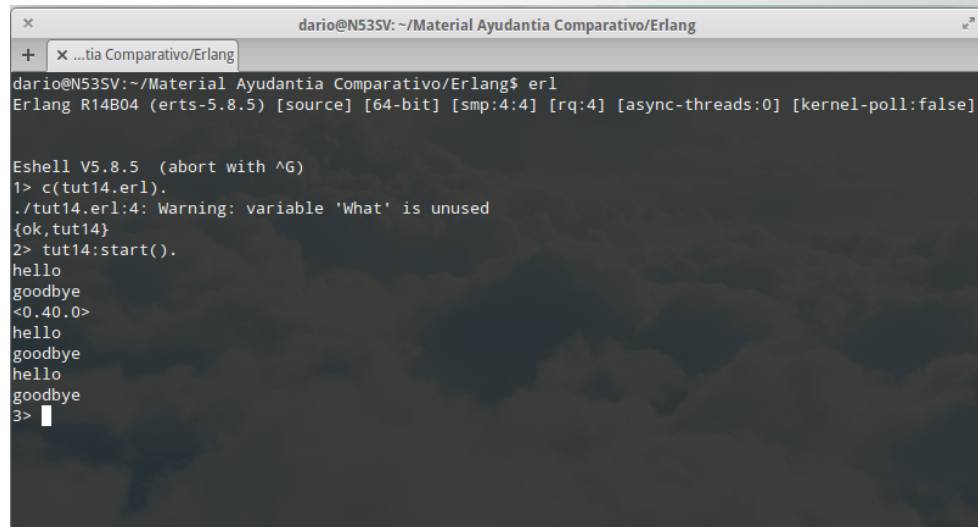
Los **Pids** son usados para todas las formas de comunicación con los procesos.

Pid = **spawn**(Module, FunctionName, ArgumentList)

Concurrencia Ejemplo



```
tut14.erl (~/Material%20Ayudantia%20Comparativo/Erlang) - Scr
+ x tut14.erl
1 -module(tut14).
2 -export([start/0, say_something/2]).
3
4 say_something(What, 0) ->
5   done;
6 say_something(What, Times) ->
7   io:format("~p~n", [What]),
8   say_something(What, Times - 1).
9
10 start() ->
11   spawn(tut14, say_something, [hello, 3]),
12   spawn(tut14, say_something, [goodbye, 3]).
```



```
dario@N535V: ~/Material Ayudantia Comparativo/Erlang
+ x ...tia Comparativo/Erlang
dario@N535V:~/Material Ayudantia Comparativo/Erlang$ erl
Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.8.5 (abort with ^G)
1> c(tut14.erl).
./tut14.erl:4: Warning: variable 'What' is unused
{ok,tut14}
2> tut14:start().
hello
goodbye
<0.40.0>
hello
goodbye
hello
goodbye
3> 
```

Concurrencia Ejemplo (continuo).

Se inician dos procesos uno que escribe "hello" en tres ocasiones y uno que escribe "goodbye" tres veces. Ambos de estos procesos utilizan la función **say_something**.

Tener en cuenta que una función que se utiliza de esta manera por **spawn** para iniciar un proceso debe ser exportada desde el módulo (es decir, en la - export al inicio del módulo).

Concurrencia Ejemplo (continuo).

Por consola:

```
1>tut14:start().
```

```
hello
```

```
goodbye
```

```
<0.63.0>
```

```
hello
```

```
goodbye
```

```
hello
```

```
goodbye
```

Notar que no escribió "hello" tres veces y luego "goodbye" tres veces, lo que sucedió fue que el primer proceso, escribió un "hello", el segundo un "goodbye", el primer otro "hello" y así sucesivamente.

Luego tenemos que **spawn** devuelve un identificador de proceso, o PID, que identifica de forma única el proceso. Por lo tanto **<0.63.0>** es el pid de la última función.

Por último tenemos que “~ p” escribe los datos con la sintaxis estándar.

Concurrencia Ejemplo (Mensajes).

La comunicación entre procesos.

Como lo mencionamos anteriormente la interacción o **comunicación** que hay entre los threads (**procesos**), se realiza mediante **mensajes**.

Erlang emplea el paso de mensajes. Existe un buzón en cada proceso al que se le puede enviar información (cualquier dato) y el código del proceso puede trabajar con ese dato de cualquier forma que necesite.

Se envía un mensaje a otro proceso utilizando la primitiva (enviar) “!”:

Pid ! Message

Pid es el identificador del proceso al que se envía el mensaje.

En el siguiente ejemplo vamos a crear dos procesos que envían mensajes entre sí varias veces.

Concurrencia Mensajes (Primitivas)

Con el fin de controlar un conjunto de actividades paralelas *Erlang* tiene primitivas para múltiples procesamiento.

spawn comienza un cómputo paralelo (llamado proceso).

send envía un mensaje a un proceso.

receive recibe un mensaje de un proceso.

La sintaxis **Pid ! Msj** se utiliza para enviar un mensaje.

Msj es el mensaje que se va a enviar a Pid. Por ejemplo: Pid ! {a, 12}

significa enviar el mensaje **{a, 12}** para el proceso con el identificador PID

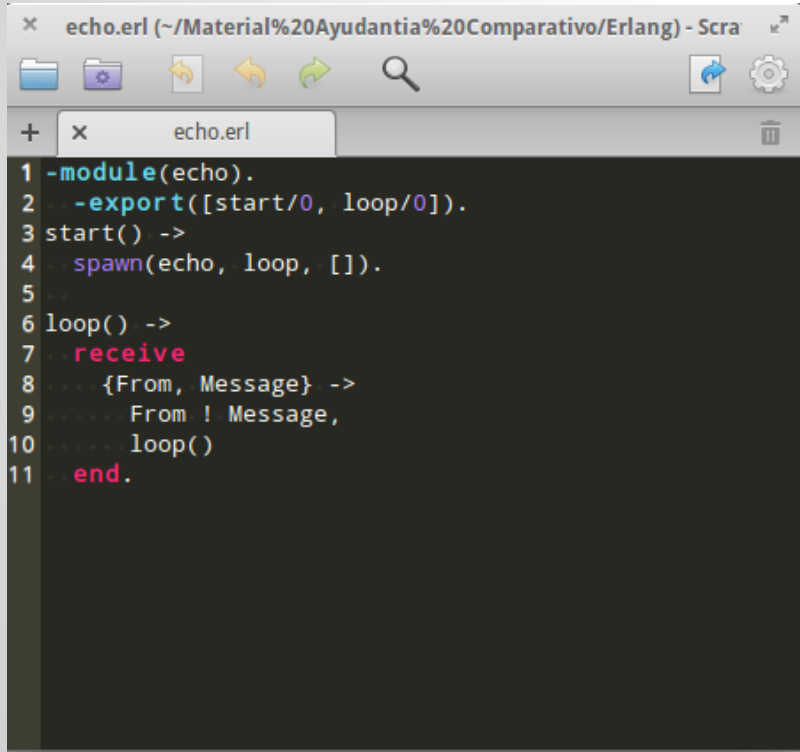
Concurrencia Ejemplo (Primitivas).

Todos los argumentos se evalúan antes de enviar el mensaje, por lo que:

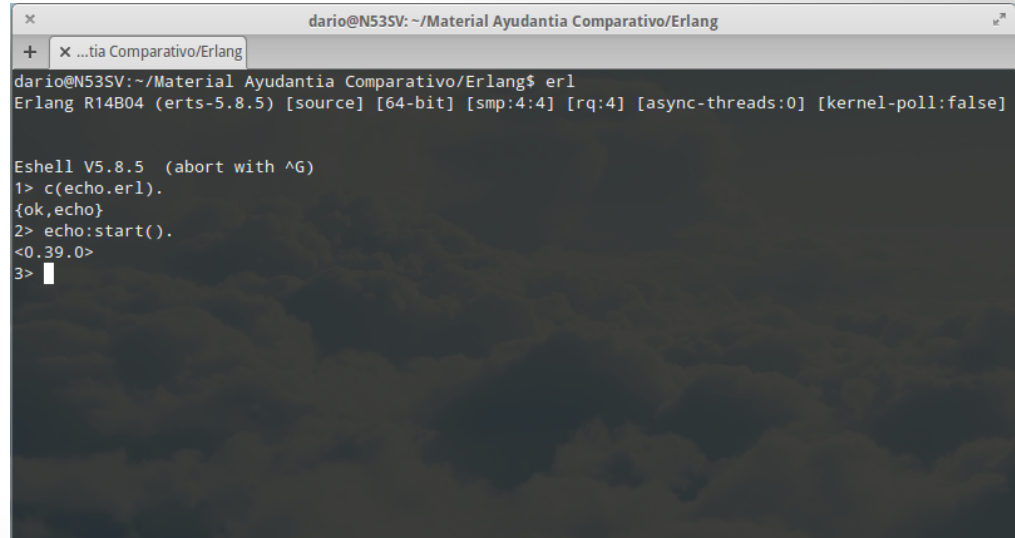
foo(12) ! math3:area({square, 5})

significaría evaluar la función **foo(12)** (la cual debe dar un identificador de proceso válido) y evaluar **math3: área ({cuadrado, 5})** a continuación, el cual envía el resultado (es decir, 25) como un mensaje al proceso.

Concurrencia Ejemplo 1 (Mensajes)



```
1 -module(echo).
2 -export([start/0, loop/0]).
3 start() ->
4   spawn(echo, loop, []).
5
6 loop() ->
7   receive
8     {From, Message} ->
9       From ! Message,
10      loop()
11  end.
```



```
dario@N53SV: ~/Material Ayudantia Comparativo/Erlang
dario@N53SV:~/Material Ayudantia Comparativo/Erlang$ erl
Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.8.5 (abort with ^G)
1> c(echo.erl).
{ok,echo}
2> echo:start().
<0.39.0>
3> 
```

Concurrencia Ejemplo 2 (Mensajes)

```
+ x tut15.erl
1 -module(tut15).
2
3 -export([start/0, ping/2, pong/0]).
4
5 ping(0, Pong_PID) ->
6   Pong_PID ! finished,
7   io:format("ping finished~n", []);
8
9 ping(N, Pong_PID) ->
10  Pong_PID ! {ping, self()},
11  receive
12    pong ->
13      io:format("Ping received pong~n", [])
14  end,
15  ping(N - 1, Pong_PID).
16
17 pong() ->
18  receive
19    finished ->
20      io:format("Pong finished~n", []);
21    {ping, Ping_PID} ->
22      io:format("Pong received ping~n", []),
23      Ping_PID ! pong,
24      pong()
25  end.
26
27 start() ->
28  Pong_PID = spawn(tut15, pong, []),
29  spawn(tut15, ping, [3, Pong_PID]).
```

```
x dario@N53SV: ~/Erlang
+ x dario@N53SV: ~/Erlang
dario@N53SV:~/Erlang$ erl
Erlang R14B04 (erts-5.8.5) [source] [64-bit] [smp:4:4] [rq:4] [async-threads:0] [kernel-poll:false]

Eshell V5.8.5 (abort with ^G)
1> c(tut15.erl).
{ok,tut15}
2> tut15:start().
Pong received ping
<0.40.0>
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
ping finished
Pong finished
3>
```

Concurrencia Ejemplo 2 (Mensajes)

Primero la función start crea un proceso llamado "pong" y luego crea otro proceso "ping".

```
Pong_PID = spawn(tut15, pong, [])  
spawn(tut15, ping, [3, Pong_PID]),
```

En caso de que quisiéramos ejecutarlo hacemos por terminal:

```
tut15:ping(3, Pong_PID)
```

Concurrencia Ejemplo Deadlock

```
Foo = spawn(fun() -> receive
```

```
  {From, foo} ->
```

```
    From ! {self(), bar},
```

```
    io:format("foo done~n")
```

```
  end
```

```
end).
```

```
Bar = spawn(fun() -> receive
```

```
  {From, bar} ->
```

```
    From ! {self(), foo},
```

```
    io:format("bar done~n")
```

```
  end
```

```
end).
```

Bibliografía

http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Erlang#.E2.80.9CHola_Mundo.E2.80.9D

Armstrong, J., Virding, R., Wikström, C., & Williams, M. (1996). Concurrent Programming in ERLANG. Älvsjö: Prentice Hall.

<http://www.erlang.org/doc.html>

Erlang/OTP Volumen I: Un Mundo Concurrente - Manuel Angel Rubio Jiménez

http://www.erlang.org/doc/getting_started/conc_prog.html



