Erlang

Ekaitz Zárraga

22 de marzo de 2015

Resumen

Índice

1. Introducción

En este apartado se introduce el funcionamiento general de Erlang, su tipado, sintaxis y filosofía general.

1.1. Funcionamiento

Erlang dispone de una máquina virtual donde se ejecutan los programas. Es parecido a lo que hace java con los bytecodes. Los programas hechos en Erlang tienen que ser compilados para esta máquina virtual.

Los archivos de código fuente tienen la extensión '.erl', de Erlang, y los compilados '.beam', que proviene de *Bogdan/Björn's Erlang Abstract Machine*, el nombre de la máquina virtual. Han existido otras máquinas virtuales pero están en desuso.

Más adelante se profundizará en la compilación.

1.2. La Shell

Erlang dispone de una shell. Sirve como emulador para poder ejecutar comandos pero también permite cosas como editar código en caliente etc. Para entrar: usar el comando *erl*.

```
ekaitz@DaComputa:~$ erl
Erlang R16B03 (erts-5.10.4) [source] [64-bit] [smp:8:8] [async-threads:10] [kernel
    -poll:false]

Eshell V5.10.4 (abort with ^G)
1>
```

La shell está basada en Emacs, usa comandos similares [Ctrl+A] para inicio de línea, [Ctrl+E] para final de línea, etc.

Como se aprecia arriba, [Ctrl+G] sirve para abortar. Una vez pulsado permite ejecutar unos comandos simples:

```
1>
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
 -->
```

Para terminar las líneas introducir '.', como cuando se pone ';' en MySQL. Las expresiones pueden separarse por comas pero sólo se mostrará el resultado de la última.

```
1> 2/3,3*4.
12
```

2. Conceptos básicos

En este apartado se explican los diferentes tipos de datos que tiene Erlang y la sintaxis básica del lenguaje.

2.1. Tipos y variables

2.1.1. Números

Erlang dispone de soporte para números de coma flotante (float) y enteros (integer) y los alternará dependiendo de sus necesidades. No le importará qué tipos estés introduciendo.

Para trabajar con diferentes bases (hexadecimal, octal, binario etc.): *BASE#NUMERO*.

```
1> 2#111.
7
```

2.1.2. Variables

La primera letra siempre tendrá que ser una mayúscula o *'_'*.

En Erlang se puede asignar valor a las variables una sola vez. Tiene un sentido más matemático el hecho de asignar un valor a una variable. Si en matemáticas *X=5*, *X=6* no tiene sentido porque no es una igualdad válida. En Erlang pasa algo parecido. La primera vez se definen y a partir de ese momento el nombre de esa variable tendrá siempre asignado el valor.

```
1> A=34.
34
2> A.
34
3> A=90.
** exception error: no match of right hand side value 90
4> 34=90.
** exception error: no match of right hand side value 90
```

Existe una variable especial llamada *'_'* que siempre se comporta como si no tuviese ningún valor asignado, siendo útil para descartar lo que se le asigne. Nunca se podrá recuperar un valor asignado a ella.

```
1> _=12.
12
2> _.
* 1: variable '_' is unbound
```

Para descartar el contenido de las variables, puede utilizarse la función *f(Variable)* que descartará la asignación de la *Variable* o simplemente *f()* que descartará todas las asignaciones.

```
1> A=3.
3
2> A.
3
3> f().
ok
4> A.
* 1: variable 'A' is unbound
```

2.1.3. Atoms

Los *atoms* son palabras literales. Siempre empiezan por minúsculas, por eso las variables siempre comenzarán con una letra en mayúsculas. Los *atoms* deben ser puestos entre comillas simples *(')* cuando empiezan por mayúsculas o contienen caracteres especiales.

Las palabras reservadas y las funciones son un caso concreto de *atom*.

Los *atoms* no se liberan durante la recolección de basura. Se guardan en una tabla, por eso no deben cargarse de forma dinámica.

```
1> atom.
atom
```

2.1.4. Tuplas

Las tuplas (*tuple* en inglés) son la estructura básica para agrupar variables. Se definen entre llaves y se separa cada elemento con comas. Aquí empieza tomar importancia la variable anónima '_', puesto que sirve para descartar partes de la tupla que no necesitemos:

```
11> {A,B}={14,17}.
{14,17}
12> A.
14
13> {A,B,_}={14,17,19}.
{14,17,19}
15> {A,B}={14,17,19}.
** exception error: no match of right hand side value {14,17,19}
16> {celsius, X}={kelvin, 60}.
** exception error: no match of right hand side value {kelvin,60}
```

Las tuplas pueden agrupar cualquier tipo de elemento y no todos los elementos tienen que ser del mismo tipo.

2.1.5. Listas

Las listas (*list* en inglés) se definen entre corchetes.

Las listas se forman por una cabeza (*head*) y una cola (*tail*), siendo la cabeza el primer elemento comenzando por la izquierda y la cola la lista formada por el resto. Las funciones *hd()* y *tl()* sirven para obtener la cabeza y la cola de una lista, respectivamente.

Las listas pueden contener elementos de cualquier tipo.

```
20> [1,2,3].
[1,2,3]
21> hd([1,2,3]).
1
22> tl([1,2,3]).
[2,3]
23> tl([1,2,{hola, ekaitz}]).
[2,{hola,ekaitz}]
```

Por defecto, Erlang considera las listas como cadenas de caracteres siempre que todos los elementos de las mismas puedan decodificarse como caracteres. Esto suele ser un engorro.

```
42> [45,46,47,89,90].
"-./YZ"
43> [45,46,47,89,90,1].
[45,46,47,89,90,1]
```

2.2. Operadores

2.2.1. Comparaciones

Las comparaciones son las habituales de otros lenguajes de programación, pero la sintaxis no es la habitual.

Igualdad	=:=
Desigualdad	=/=
Igualdad numérica	==
Desigualdad numérica	/=
Mayor o igual que	>=
Menor que igual que	=<
Mayor que	>
Menor que	<

La comparación numérica (== y /=) difiere de la comparación normal (=:= y =/=) en que, en el primer caso la comparación se hace a nivel de significado numérico, es decir: 5 y 5.0 son iguales, mientras que en el segundo caso se considerarán diferentes por tener un tipado distinto (integer vs float).

Por otro lado, se pueden comparar objetos de diferentes tipos, pero el resultado será el siguiente:

```
number < atom < reference < fun < port < pid < tuple < list < bit string
```

Nota: *true* y *false* son *atoms* y no concuerdan con ningún valor numérico como en otros lenguajes. Como se puede apreciar en la lista superior, siempre serán mayores que cualquier número.

2.2.2. Operadores booleanos

Los operadores booleanos son los habituales: *and*, *or*, *xor*, *not* (como son *atoms*, se expresan en minúsculas). Para ejecutar primero el de la izquierda y luego el de la derecha en función del primer resultado (*short-circuit evaluation*[1]) utilizar *andalso* y *orelse*.

2.2.3. Operadores matemáticos

Los operadores matemáticos más comunes son: *+*, -, *, */*, *div* (división entera) y *rem* (*remainder*, el resto de la división). Pueden utilizarse paréntesis para agrupar operaciones y cambiar su prioridad como en otros lenguajes de programación o en matemáticas.

2.2.4. Operadores para listas

Añadir y quitar elementos: Para añadir elementos a las listas se utiliza *++* y para retirarlos *– –*. Cuidado: Son right-associative:

```
37> [1,2,3]--[1,2]++[3].
[]
```

Operador cons: El operador *cons* (de constructor) se define con la pleca (|). Sirve tanto para hacer coincidir patrones o para construir listas. Su objetivo es separar la cabeza de la cola en las listas de la siguiente manera (*[HEAD|TAIL]*):

```
26> [Nombre | Apellidos] = ['Ekaitz', 'Zarraga', 'Rio'].
['Ekaitz','Zarraga','Rio']
27> Nombre.
'Ekaitz'
28> Apellidos.
['Zarraga','Rio']
29> [1|[2,3,4]].
[1,2,3,4]
```

Tal y como se ha dicho antes, la cola es una lista así que, a la hora de construir listas con este operador, lo que se añada como cola siempre deberá ser una lista. [1|2] no es correcto, lo correcto sería: [1|[2,[]]] o [1,2|[]].

List comprehensions: Es una manera de construir listas de forma matemática del estilo de la siguiente, que construye la lista [3,4,5,6]:

$$\{x \epsilon \mathbb{N}, x \leq 4 : y = x + 2\} \tag{1}$$

[1]Consultar: http://en.wikipedia.org/wiki/Short-circuit_evaluation

En Erlang funciona igual, sólo cambia la sintaxis:

```
38> [2+X || X<-[1,2,3,4]].
[3,4,5,6]
```

Pueden añadirse más condiciones:

```
39> [2+X || X<-[1,2,3,4], X rem 2 =:= 0].
[4,6]
```

2.2.5. Operadores binarios

Erlang es un lenguaje con un soporte fuerte para valores en binario puesto que está diseñado para las telecomunicaciones.

Separación de binarios: Los datos binarios pueden colocarse entre << y >> para dividirlos en secciones legibles.

3. Módulos

Un módulo (*module* en inglés) es un conjunto de funciones agrupadas en un único archivo, definido con un nombre con la extensión *.erl*.

Para ejecutar funciones de un módulo concreto la sintaxis es la siguiente:

```
Modulo:Función(Argumentos).
```

En el siguiente ejemplo *io* sería el módulo, la función *format* y los argumentos el string: *"Hola Mundo!"*:

```
77> io:format("Hola Mundo!~n").
Hola Mundo!
ok
```

En los módulos hay dos tipos de cosas: Funciones y Atributos.

Los atributos son el conjunto de metadatos que describen el módulo y se definen así:

```
-Nombre(Valor).
```

- El atributo mínimo para un módulo es el nombre:

  ```
  -module(NombreDelModulo).
  ```

- Para definir qué funciones serán mostradas al exterior del módulo se utiliza el atributo *-export()* [2]:

  ```
  -export([Función1/Aridad, Función2/Aridad...]).
  ```

- Para importar funciones de otros módulos:

  ```
  -import(Modulo,[Función1/Aridad, Función2/Aridad...]).
  ```

- Para definir macros: *-define(Nombre,Valor).* que se utilizan *?Nombre*. Por ejemplo, la macro *-define(sub(X,Y), X-Y).* utilizada como *?sub(13,5)* tomará el valor *13-5*.

Las funciones se definen de la siguiente manera:

```
Nombre(Argumentos)-> Cuerpo.
```

- Nombre: Es un *atom*, así que debe comenzar en minúsculas.
- Cuerpo: Expresiones separadas por comas, la última expresión ejecutada será el valor devuelto.
- Argumentos: Separados por comas.

[2] Aridad: En inglés *arity*. En el análisis matemático, la aridad de un operador matemático o de una función es el número de argumentos necesarios para que dicho operador o función se pueda calcular.

Todos los módulos tienen (automáticamente) una función *module_info/0* que muestra los metadatos del módulo. Para llamarla:

```
modulo:module_info().
```

Ejemplo de módulo:

```
-module(functions).
-export([head/1,second/1]).

head([H|_]) -> H.
second([_,X|_]) -> X.
```

3.1. Compilación

Tal y como se ha dicho antes el código fuente de Erlang debe ser compilado para la máquina virtual. Existen muchas formas de hacerlo:

- Desde la shell de Erlang:

  ```
  1> c(modulo).
  {ok,functions}
  ```

  Si el módulo no se encuentra en la carpeta actual navegar usando *cd("path").* y *ls().* o introducir el path hasta el archivo.

- Desde la shell de unix:

  ```
  ekaitz@DaComputa:~$ erlc flags modulo.erl
  ```

- Desde la shell de Erlang o desde un módulo:

  ```
  compile:file(modulo).
  ```

4. Escribir modulos