Erlang/OTP

VOLUMEN I

Un Mundo Concurrente

3ª edición

MANUEL ÁNGEL RUBIO JIMÉNEZ

Erlang/OTP

Volumen I: Un Mundo Concurrente

Manuel Angel Rubio Jiménez

Erlang/OTP
Volumen I: Un Mundo Concurrente

Manuel Angel Rubio Jiménez

Resumen

El lenguaje de programación Erlang nació en 1986 en los laboratorios Ericsson de la mano de Joe Armstrong. Es un lenguaje funcional con base en Prolog, tolerante a fallos, y orientado al trabajo en tiempo real y a la concurrencia, lo que le proporciona ciertas ventajas en lo que a la declaración de algoritmos se refiere.

Como la mayoría de lenguajes funcionales Erlang requiere un análisis del problema y una forma de diseñar la solución diferente a como se haría en un lenguaje de programación imperativo. Sugiere una mejor y más eficiente forma de llevarlo a cabo. Se basa en una sintaxis más matemática que programática por lo que tiende más a la resolución de problemas que a la ordenación y ejecución de órdenes.

Todo ello hace que Erlang sea un lenguaje muy apropiado para la programación de elementos de misión crítica principalmente a nivel de servidor e incluso para el desarrollo de sistemas embebidos o incrustados por su eficiente gestión de CPU y memoria.

En este libro explicamos el lenguaje y su plataforma, cómo cubre las necesidades para las que fue creado, cómo sacarle el máximo provecho a su forma de realizar las tareas y a su orientación a la concurrencia. Es un repaso desde el principio sobre cómo programar de una forma funcional y concurrente en un entorno distribuido y tolerante a fallos.

Esta tercera revisión comprende hasta la **versión 24** exponiendo la nueva sintaxis para obtener el retorno de pila en una excepción, los alias de procesos, nuevas formas de trabajar con la memoria a través de los atómicos, contadores y términos persistentes, un nuevo capítulo dedicado a ***crypto*** otro capítulo dedicado a ***logger*** y el cambio a **rebar3** para la construcción de nuestros proyectos.

[1] http://erlang-otp.es/
[2] http://creativecommons.org/licenses/by-nc-sa/3.0/

Capítulo 2. El lenguaje

Solo hay dos tipos de lenguajes: aquellos de los que la gente se queja y aquellos que nadie usa.
— Bjarne Stroustrup

Erlang tiene una sintaxis muy particular. Hay gente a la que termina gustándole y otras personas que lo consideran incómodo. Hay que entender que es un lenguaje basado en Prolog y con tintes de Lisp por lo que se asemeja más a los lenguajes funcionales que a los imperativos.

En Erlang la sintaxis está diseñada como si se tratara de la definición de una función matemática o una proposición lógica. Cada elemento dentro de la función tiene un propósito: obtener un valor; el conjunto de todos esos valores, con o sin procesamiento, conforma el resultado. Un ejemplo básico:

```
area(Base, Altura) -> Base * Altura.
```

En este ejemplo puede verse la definición de la función ***area***. Los parámetros requeridos para obtener su resultado son ***Base*** y ***Altura***. A la declaración de parámetros le sigue el símbolo de consecución (→), como si se tratase de una proposición lógica. Por último está la operación interna que retorna el resultado que se quiere obtener.

Al tratarse de funciones matemáticas o proposiciones lógicas no existe una correlación directa entre imperativo y funcional. Para un código imperativo común como el que sigue:

```
para i <- 1 hasta 10 hacer
    si clavar(i) = 'si' entonces
        martillea_clavo(i)
    fsi
fpara
```

No existe en Erlang un equivalente que pueda transcribir una acción imperativa como tal. Para desarrollar en Erlang hay que pensar en el ***qué*** se quiere hacer más que en el ***cómo***. Si en un lenguaje funcional lo que se quiere es clavar los clavos que seleccione la función ***clavar*** martilleando, se podría hacer a través de una lista de comprensión:

```
[ martillea_clavo(X) || X <- Clavos, clavar(i) =:= 'si' ].
```

Hay que entender que para resolver problemas de forma funcional muchas veces la mentalidad imperativa es un obstáculo. Tenemos que pensar en los datos que tenemos y qué datos queremos obtener como resultado. Es lo que nos conducirá a la solución.

Erlang es un lenguaje de formato libre. Se pueden insertar tantos espacios y saltos de línea entre símbolos como se quiera. Esta función ***area*** es completamente equivalente a la anterior a nivel de ejecución:

```
area(
  Base,
  Altura
) ->
  Base * Altura
.
```

A lo largo de este capítulo revisaremos la base del lenguaje Erlang. Veremos lo necesario para poder escribir programas básicos de propósito general y entender esta breve introducción de una forma más detallada y clara.

1. Tipos de Datos

En Erlang se manejan varios tipos de datos. Por hacer una distinción rápida podemos decir que se distinguen entre: simples y complejos; otras organizaciones podrían conducirnos a pensar en los datos como: escalares y conjuntos o atómicos y compuestos. No obstante, la forma de organizarlos no es relevante con el fin de conocerlos, identificarlos y usarlos correctamente. Emplearemos la denominación simples y complejos (o compuestos), pudiendo referirnos a cualquiera de las otras formas de categorización si la explicación resulta más clara.

Como datos simples veremos en esta sección los **átomos** y los **números**. Como datos de tipo complejo veremos las **listas** y **tuplas**. También veremos las **listas binarias**, un tipo de dato bastante potente de Erlang y los **registros**, un tipo de dato derivado de las tuplas. Para terminar revisaremos un tipo de dato que se introdujo en la versión 17 de Erlang, los **mapas**.

1.1. Átomos

Los átomos son identificadores de tipo carácter que se emplean como palabras clave y ayudan a dar semántica al código.

Un átomo es una palabra que comienza por una letra en minúscula y va seguido de letras en mayúscula o minúscula, números y/o subrayados. También se pueden emplear letras en mayúscula al inicio, espacios y lo que queramos, siempre y cuando encerremos la expresión entre comillas simples. Algunos ejemplos:

```
> is_atom(cuadrado).
true
```

```
> is_atom(a4).
true
> is_atom(alta_cliente).
true
> is_atom(bajaCliente).
true
> is_atom(alerta_112).
true
> is_atom(false).
true
> is_atom('HOLA').
true
> is_atom('     eh??? ').
true
```

Los átomos tienen como única finalidad ayudar al programador a identificar estructuras, algoritmos y código específico empleando muy poca memoria para ello.

Hay átomos que se emplean con mucha frecuencia como son: true, false y undefined.

Importante

Cada vez que se emplea un átomo, se agrega la representación a una tabla interna. Esta tabla tiene un tamaño finito pero configurable. Por defecto el valor máximo de átomos diferentes que pueden emplearse en una máquina virtual de Erlang es 1048576. Este valor se puede ampliar con el parámetro ***+t***.

Una de las formas para asegurar no sobrepasar el límite de átomos disponibles es emplearlos de forma explícita en el código. De esta forma se tiene un control de cuántos átomos se están usando y al momento de cargar el código nos puede avisar de que el número ha sido sobrepasado.

Si necesitamos convertir de otros tipos de datos a átomos y asegurarnos de no sobrepasar el límite podemos emplear el conjunto de funciones `*_to_existing_atom/1,2`.

Los átomos junto con los números enteros y reales y las cadenas de texto componen los ***literales***. Los literales son datos con un significado de por sí, y se pueden asignar a una variable directamente.

Nota

Como literales se pueden especificar números, pero también valores de representaciones de la tabla de caracteres. Erlang permite dar el valor de un carácter específico a través el uso de la sintaxis: $A, $1, $!. Esto retornará el valor numérico para el símbolo indicado tras el símbolo del dólar en la tabla de caracteres.

1.2. Números Enteros y Reales

En Erlang, los números pueden ser de dos tipos, tal y como se ve en este ejemplo de código en la consola:

```
> is_float(5).
false
> is_float(5.0).
true
> is_integer(5.0).
false
> is_integer(5).
true
> is_integer(1_000_000).
true
```

Curiosidad

Para facilitar la lectura, a partir de la versión 23 podemos emplear los símbolos de subrayado para separar números y hacerlos más legibles. Por ejemplo, si queremos escribir una cifra grande: 1_234_567. Estos separadores podemos ponerlos de forma arbitraria para separar el número sin importar si lo hacemos cada 2, 3, 4 o más números.

Otra de las cosas que sorprende de Erlang es su precisión numérica. Si multiplicamos números muy altos veremos como el resultado sigue mostrándose en notación real, sin usar la notación científica que muestran otros lenguajes cuando una operación supera el límite de cálculo de los números enteros (o valores erróneos por ***overflow***):

```
> 102410241024 * 102410241024 * 1234567890.
1294797206315341928712675262464O
```

Esta característica hace de Erlang una plataforma muy precisa y adecuada para cálculos de intereses bancarios, tarificación telefónica, índices bursátiles, valores estadísticos, posición de puntos tridimensionales, etc.

Nota

Los números se pueden indicar también anteponiendo la base en la que queremos expresarlos y usando como separador la almohadilla (#). Por ejemplo, si queremos expresar los números en base octal, lo haremos anteponiendo la base al número que queremos representar 8#124. Análogamente 2#1011 representa un número binario y 16#f42a un número hexadecimal.

1.3. Variables

Las variables, como en matemáticas, son símbolos a los que se enlaza un valor y solo uno a lo largo de toda la ejecución del algoritmo específico. Esto quiere decir que cada variable durante su tiempo de vida solo puede contener un valor.

El formato de las variables se inicia con una letra mayúscula, seguida de tantas letras, números y subrayados como se necesiten o deseen. Una variable puede tener esta forma:

```
> Pi = 3.1415.
3.1415
> Telefono = "666555444".
"666555444"
> Depuracion = true.
true
```

Sobre las variables se pueden efectuar expresiones aritméticas, en caso de que contenga números, operaciones de listas o emplearse como parámetro en llamadas a funciones. Un ejemplo de variables conteniendo números:

```
> Base = 2.
2
> Altura = 5.2.
5.2
> Base * Altura.
10.4
```

Si en un momento dado, queremos que ***Base*** tenga el valor 3 en lugar del valor 2 inicialmente asignado veríamos lo siguiente:

```
> Base = 2.
2
> Base = 3.
** exception error: no match of right hand side value 3
```

Lo que está ocurriendo es que ***Base*** ya está enlazado al valor 2 y que la concordancia (o match) con el valor 2 es correcto, mientras que si lo intentamos encajar con el valor 3 resulta en una excepción.

Nota

Para nuestras pruebas, a nivel de consola y para no tener que salir y entrar cada vez que queramos que Erlang ***olvide*** el valor con el que se enlazó una variable, podemos emplear:

```
> f(Base).
ok
> Base = 3.
3
```

Para eliminar todas las variables que tenga memorizadas la consola se puede emplear: `f().`

La ventaja de la asignación única es la facilidad de analizar código aunque muchas veces no se considere así. Si una variable durante toda la ejecución de una función solo puede contener un determinado valor el comportamiento de dicha función es muy fácil de verificar[1].

1.4. Listas

Las listas en Erlang son vectores de información heterogénea, es decir, pueden contener información de distintos tipos, ya sean números, átomos, tuplas u otras listas.

Las listas son una de las potencias de Erlang. Erlang maneja las listas como lenguaje de alto nivel, en modo declarativo, permitiendo construcciones como las listas de comprensión o la agregación y eliminación de elementos específicos como si de conjuntos se tratase.

1.4.1. ¿Qué podemos hacer con una lista?

Una lista de elementos se puede definir de forma directa tal y como se presenta a continuación:

```
> [ 1, 2, 3, 4, 5 ].
[1,2,3,4,5]
> [ 1, "Hola", 5.0, hola ].
[1,"Hola",5.0,hola]
```

A estas listas se les pueden agregar o sustraer elementos con los operadores especiales ***++*** y ***--***. Tal y como se presenta en los siguientes ejemplos:

```
> [1,2,3] ++ [4].
[1,2,3,4].
> [1,2,3] -- [2].
[1,3]
```

[1] Muestra de ello es ***dialyzer***, una buena herramienta para comprobar el código escrito en Erlang.

Otro de los usos comunes de las listas es la forma en la que se puede ir tomando elementos de la cabecera de la lista dejando el resto en otra sublista. Esto se realiza con esta sencilla sintaxis:

```
> [H|T] = [1,2,3,4].
[1,2,3,4]
> H.
1
> T.
[2,3,4]
> [H1,H2|T2] = [1,2,3,4].
[1,2,3,4]
> H1.
1
> H2.
2
> T2.
[3,4]
```

De esta forma tan sencilla la implementación de los conocidos algoritmos de ***push*** y ***pop*** de inserción y extracción en pilas resultan tan triviales como:

```
> Lista = [].
[]
> Lista2 = [1|Lista].
[1]
> Lista3 = [2|Lista2].
[2,1]
> [Extrae|Lista2] = Lista3.
[2,1]
> Extrae.
2
> Lista2.
[1]
```

No obstante, el no poder mantener una única variable para la pila dificulta su uso en consola. Este asunto lo analizaremos más adelante con el tratamiento de los procesos y las funciones.

1.4.2. Cadenas de Texto

Las cadenas de texto son un tipo específico de lista. Se trata de una lista homogénea de elementos representables como caracteres. Erlang detecta que si una lista en su totalidad cumple con esta premisa, es una cadena de caracteres.

Por tanto, la representación de la palabra ***Hola*** en forma de lista, se puede hacer como lista de enteros que representan a cada una de las letras o como el texto encerrado entre comillas dobles ("). Una demostración:

```
> "Hola" = [72, 111, 108, 97].
"Hola"
```

Como puede apreciarse, la asignación no da ningún error ya que ambos valores, a izquierda y derecha, son iguales para Erlang.

Al igual que con el resto de listas, las cadenas de caracteres soportan también la agregación de elementos, de modo que la concatenación se podría realizar de la siguiente forma:

```
> "Hola, " ++ "mundo!".
"Hola, mundo!"
```

Una de las ventajas de la asignación única de Erlang es que si encuentra una variable no enlazada a ningún valor, automáticamente se enlaza al valor necesario para hacer cierta la ***ecuación***. Erlang intenta hacer siempre iguales los elementos a ambos lados del signo de asignación. Un ejemplo:

```
> "Hola, " ++ A = "Hola, mundo!".
"Hola, mundo!"
> A.
"mundo!"
```

Esta notación tiene sus limitaciones, en concreto la variable no asignada debe estar al final de la expresión, ya que de otra forma el código para realizar el ***encaje*** sería mucho más complejo.

1.4.3. Listas binarias

Las cadenas de caracteres se forman por conjuntos de enteros, es decir, se consume el doble de memoria para una cadena de caracteres almacenada en una lista en Erlang que si empleásemos una cadena de datos binarios. Tiene sus ventajas cuando empleamos Unicode, pero la mayoría de sistemas siempre ha trabajado con representaciones basadas en bytes. Para este caso, tenemos las listas binarias. Las listas binarias nos permiten almacenar cadenas de caracteres con tamaño de byte y nos permite realizar trabajos específicos con secuencias de bytes o incluso a nivel de bit.

La sintaxis de este tipo de listas es como sigue:

```
> <<"Hola">>.
<<"Hola">>
> <<72, 111, $l, $a>>.
<<"Hola">>
```

La lista binaria no tiene las mismas funcionalidades que las listas vistas anteriormente. No se pueden agregar elementos ni emplear el formato de anexión y supresión de elementos tal y como se había visto antes. Pero se puede hacer de otra forma mucho más versátil.

Por ejemplo, la forma en la que tomábamos la cabeza de la lista en una variable y el resto lo dejábamos en otra variable, se puede simular de la siguiente forma:

```
> <<H:1/binary, T/binary>> = <<"Hola">>.
<<"Hola">>
> H.
<<"H">>
> T.
<<"ola">>
```

La concatenación en el caso de las listas binarias no se realiza como con las listas normales empleando el operador ++. En este caso debe realizarse de la siguiente forma:

```
> A = <<"Hola ">>.
<<"Hola ">>
> B = <<"mundo!">>.
<<"mundo!">>
> C = <<A/binary, B/binary>>.
<<"Hola mundo!">>
```

Para obtener el tamaño de la lista binaria empleamos la función `byte_size/1`. En el caso anterior para cada una de las variables empleadas:

```
> byte_size(A).
5
> byte_size(B).
6
> byte_size(C).
11
```

Curiosidad

Cuando un binario ocupa más de 64 bytes se desplaza a una zona de memoria especial y cualquier concordancia que provoque una nueva variable esta será una referencia al binario original con un valor de desplazamiento y tamaño. Por ejemplo, si en la variable A tenemos una cadena de 100 bytes y hacemos una concordancia para obtener los primeros 80 bytes:

```
> <<B:80/binary, _/binary>> = A.
<<"12345678901234567890123456789012345678...">>
> byte_size(B).
80
> binary:referenced_byte_size(B).
100
```

Vemos cómo la tamaño referenciado sigue siendo de 100 bytes en lugar de los 80 bytes que ocupa la cadena. Para obtener una cadena completamente nueva y la anterior pueda ser recolectada por el recolector de basura, deberíamos emplear la función `binary:copy/1`. Comentaremos esta y otras optimizaciones en el capítulo de funciones y módulos.

Esta sintaxis es un poco más elaborada que la de las listas, pero se debe a que nos adentramos en la verdadera potencia que tienen las listas binarias: el manejo de bits.

1.4.4. Trabajando con Bits

En la sección anterior vimos la sintaxis básica para simular el comportamiento de la cadena al tomar la cabeza de una pila. Esta sintaxis se basa en el siguiente formato: ***Var:Tamaño/Tipo***; siendo opcionales Tamaño y Tipo.

El tamaño está ligado al tipo, ya que una unidad de medida no es nada sin su cuantificador. En este caso, el cuantificador (o tipo) que hemos elegido es ***binary***. Este tipo indica que la variable será de tipo lista binaria, con lo que el tamaño será referente a cuántos elementos de la lista contendrá la variable.

Si no indicamos el tamaño, se asume tanto como el tipo soporte y/o hasta ***encajar*** el valor al que debe de igualarse (si es posible), por ello en el ejemplo anterior la variable T se queda con el resto de la lista binaria.

Los tipos también tienen modificadores, podemos indicar varios elementos para completar la definición de los mismos. Estos elementos son, en orden de especificación: ***Endian-Signo-Tipo-Unidad***; vamos a ver los posibles valores para cada uno de ellos:

Endian

Es la forma en la que los bits son leídos en la máquina, si es en formato Intel o Motorola, es decir, ***little*** o ***big*** respectivamente. Además de estos dos, es posible elegir ***native***, que empleará el formato nativo de la máquina en la que se esté ejecutando el código. El valor por defecto es ***big***.

```
> <<1215261793:32/big>>.
<<"Hola">>
> <<1215261793:32/little>>.
<<"aloH">>
> <<1215261793:32/native>>.
<<"aloH">>
```

En este ejemplo se ve que la máquina de la prueba es de tipo ***little*** u ordenación Intel.

Signo

Se indica si el número indicado se almacenará en formato con signo o sin él, es decir, ***signed*** o ***unsigned***, respectivamente.

Tipo

Es el tipo con el que se almacena el dato en memoria. Según el tipo el tamaño es relevante para indicar precisión o número de bits, por ejemplo. Los tipos disponibles son: ***integer***, ***float*** y ***binary***.

Unidad

Este es el valor de la unidad, por el que multiplicará el tamaño. En caso de enteros y coma flotante el valor por defecto es 1, y en caso de binario es 8. Por lo tanto: ***Tamaño x Unidad = Número de bits***; por ejemplo, si la unidad es 8 y el tamaño es 2, los bits que ocupa el elemento son 16 bits.

Si quisiéramos almacenar tres datos de color rojo, verde y azul en 16 bits, tomando para cada uno de ellos 5, 5 y 6 bits respectivamente, tendríamos que la partición de los bits se podría hacer de forma algo dificultosa. Con este manejo de bits, componer la cadena de 16 bits (2 bytes) correspondiente, por ejemplo, a los valores 20, 0 y 6, sería así:

```
> <<20:5, 0:5, 60:6>>.
<<" <">>
```

Nota

Para obtener el tamaño de la lista binaria en bits podemos emplear la función `bit_size/1` de la siguiente manera:

```
> bit_size(<<"Hola mundo!">>).
88
```

1.4.5. Listas de E/S

Existe en Erlang una construcción específica para tratar datos que serán empleados para una salida o recibidos en una entrada de datos. Este dato se conoce como lista de entrada/salida o ***iolist***.

Estas listas son heterogéneas pudiendo contener caracteres, binarios u otras listas. La ventaja de estas construcciones radica en poder crear listas con toda la información de salida de forma rápida y mediante la función ***iolist_to_binary/1*** convertir esa lista en un dato binario (o lista binaria).

Por ejemplo, si encontramos la siguiente lista:

```
> Hola = ["Hola", $\s ["Manuel"], $\s, <<"a Erlang!">>].
```

Vemos una combinación extraña entre listas de caracteres, caracteres y binarios para conformar un texto, posiblemente para ser enviado a través de una conexión de red, imprimirlo a consola o guardarlo en un fichero. Cualquier opción es válida y este formato será aceptado por la mayoría de las funciones.

Si probamos a realizar una conversión con la función mencionada obtendremos:

```
> iolist_to_binary(Hola).
<<"Hola Manuel a Erlang!">>
```

Este tipo de dato es muy útil para tratar listas de caracteres, modificar elementos agregando subelementos con formato de otras listas, caracteres o incluso binarios, sin preocupación y convertirlos posteriormente a listas binarias.

1.5. Tuplas

Las tuplas son tipos de datos organizativos en Erlang. Se pueden crear listas de tuplas para conformar conjuntos de datos homogéneos de elementos individuales heterogéneos.

Las tuplas, a diferencia de las listas, no pueden incrementar ni decrementar su tamaño salvo por la redefinición completa de su estructura o el uso de funciones específicas. Se emplean para agrupar datos con un propósito específico. Por ejemplo, imagina que tenemos un directorio con unos cuantos ficheros. Queremos almacenar esta información para poder tratarla y sabemos que los datos son: ruta, nombre, tamaño y fecha de creación.

Podemos almacenar esta información en forma de tupla de la siguiente forma:

```
{ "/home/yo", "texto.txt", 120, {{2011, 11, 20}, {0, 0, 0}} }.
```

Las llaves indican el inicio y fin de la definición de la tupla, y los elementos separados por comas conforman su contenido.

Nota

En el ejemplo se puede ver que la fecha y hora se ha introducido de una forma un tanto peculiar. En Erlang, las funciones de los módulos de su librería estándar, trabajan con este formato, y si se emplea, es más fácil tratar y trabajar con fechas. Por ejemplo, si ejecutásemos:

```
> {date(), time()}.
{{2011,12,6},{22,5,17}}
```

1.5.1. Modificación dinámica de tuplas

Hay momentos en los que necesitamos agregar un valor más a una tupla, eliminar un valor o tomar el valor que está en una posición sin necesidad de realizar una concordancia. Para ello podemos optar por emplear alguna de las siguientes funciones:

`erlang:setelement/3`

Cambia el elemento de una tupla sin modificar el resto de los elementos:

```
> setelement(2, {a, b, c}, 'B').
{a,'B',c}
```

`erlang:append_element/2`

Agrega un elemento al final de la tupla:

```
> erlang:append_element({a, b, c}, d).
{a,b,c,d}
```

`erlang:element/2`

Obtiene un elemento de la tupla dado su índice:

```
> element(1, {a, b, c}).
a
```

`erlang:delete_element/2`

Elimina un elemento de una tupla:

```
> erlang:delete_element(2,{a,b,c}).
{a,c}
```

1.5.2. Listas de Propiedades

Una lista de propiedades es una lista de tuplas de dos elementos: clave y valor. O tan solo clave para indicar la existencia de ese valor como booleano. Se gestiona mediante la librería ***proplists***. Las listas de propiedades son muy usadas para almacenar configuraciones o en general cualquier información variable que se requiera almacenar.

Nota

Antes de que llegasen los mapas en la versión 17, las listas de propiedades eran un recurso muy empleado para almacenar información usando una clave para obtener un valor. A la llegada de los mapas, las listas de propiedades se han visto relegadas principalmente a configuración.

Supongamos que tenemos la siguiente muestra de datos:

```
> A = [{path, "/"}, debug, {days, 7}].
```

Ahora supongamos que de esta lista, que se ha cargado desde algún fichero o mediante cualquier otro método, queremos consultar si debemos realizar o no la depuración del sistema, es decir, mostrar mensajes de log si la propiedad ***debug*** es igual a ***true***:

```
> proplists:get_value(debug, A).
true
```

El valor ***debug*** es un caso especial, cualquier átomo presente en la lista es equivalente a la construcción ***{debug, true}***. Además, podemos emplear otra función específica para asegurar el valor booleano de la clave incluso aunque su contenido sea diferente:

```
> proplists:get_bool(debug, A).
```

```
true
> proplists:get_bool(days, A).
false
```

Como el valor de ***days*** no es booleano se asume ***false***.

Como es muy posible que no conozcamos las claves que existen en un determinado momento dentro de la lista existen las funciones `proplists:is_defined/2`, o `proplists:get_keys/1` para poder obtener una lista de claves de la lista.

Otro ejemplo para ayudarnos a obtener los días del mes dado el nombre del mes:

```
> Meses = [
    {enero, 31}, {febrero, 28}, {marzo, 31},
    {abril, 30}, {mayo, 31}, {junio, 30},
    {julio, 31}, {agosto, 31}, {septiembre, 30},
    {octubre, 31}, {noviembre, 30}, {diciembre, 31}
].
> proplists:get_value(enero, Meses).
31
> proplists:get_value(junio, Meses).
30
```

El empleo de las listas de propiedades de esta forma nos facilita el acceso a los datos que sabemos que existen dentro de una colección (o lista) y extraer únicamente los datos requeridos.

Nota

El módulo ***proplists*** contiene muchas más funciones útiles para tratar este tipo de colección de datos de forma fácil. No es mala idea dar un repaso al mismo para ver el partido que podemos sacarle en nuestros programas.

1.6. Registros

Los registros son un tipo específico de tupla que facilita el acceso a los datos individuales dentro de la misma mediante un nombre y una sintaxis de acceso mucho más cómoda para el programador. Por ejemplo, el preprocesador puede proporcionarnos información en caso de escribir el nombre de un campo no existente para el registro a través de un fallo de compilación.

Curiosidad

Internamente para Erlang, los registros realmente no existen. A nivel de preprocesador son intercambiados por tuplas.

Como los registros se emplean a nivel de preprocesador, en la consola solo podemos definir registros empleando un comando específico de consola. Además, podemos cargar los registros existentes en un fichero y emplearlos desde la propia consola para definir datos o para emplear los comandos propios de manejo de datos con registros.

Realizamos la definición de registros en la consola de la siguiente forma:

```
> rd(agenda, {nombre, apellidos, telefono}).
```

Para declarar un registro en un archivo debemos emplear este formato:

```
-record(agenda, {nombre, apellidos, telefono}).
```

La declaración puede complicarse un poco más si agregamos valores por defecto en la definición anterior:

```
-record(agenda, {
    nombre,
    apellidos = "",
    telefono
}).
```

O desde la consola de esta forma:

```
> rd(agenda, {nombre, apellidos="", telefono}).
```

Los valores por defecto de cada uno de los elementos del registro siempre es ***undefined***. En el caso anterior, podemos comprobarlo de la siguiente forma:

```
> #agenda{}.
#agenda{nombre=undefined, appellidos=[], telefono=undefined}
```

En el momento de declarar o emplear un registro podemos querer cambiar la mayoría de campos por un valor específico y diferente de ***undefined***:

```
> #agenda{_=nodefinido}.
#agenda{nombre=nodefinido, apellidos=nodefinido, telefono=nodefinido}
> #agenda{nombre="Manuel",_=no}
#agenda{nombre = "Manuel",apellidos = no,telefono = no}
```

Nota

Los ficheros de código de Erlang normalmente tiene la extensión ***erl***, sin embargo, cuando se trata de códigos de tipo ***cabecera***, estos ficheros cambian la primera letra por una hache (h) de cabecera en inglés. Su extensión es: ***hrl***. En estos ficheros se introducirán normalmente definiciones y registros.

Veamos con una pequeña prueba que si creamos una tupla A, Erlang la reconoce como tupla de cuatro elementos. Si cargamos después el archivo `registros.hrl` cuyo contenido es la definición del registro agenda el tratamiento de la tupla se modifica automáticamente y ya podemos emplear la notación para registros de los ejemplos subsiguientes:

```
> A = {agenda, "Manuel", "Rubio", 666666666}.
{agenda,"Manuel","Rubio",666666666}
> rr("registros.hrl").
[agenda]
> A.
#agenda{nombre = "Manuel",apellidos = "Rubio",
        telefono = 666666666}
```

Erlang reconoce como primer dato de la tupla el nombre del registro y como cuenta con el mismo número de elementos la considera automáticamente como un registro. También se pueden seguir empleando las funciones y elementos típicos de la tupla ya que a todos los efectos sigue siéndolo.

Nota

Para obtener la posición dentro de la tupla de un campo, basta con escribirlo de la siguiente forma:

```
#agenda.nombre
```

Esto nos retornará la posición relativa definida como nombre con respecto a la tupla que contiene el registro de tipo ***agenda***.

Para tratar los datos de un registro, podemos realizar cualquiera de las siguientes acciones:

```
> A#agenda.nombre.
"Manuel"
> A#agenda.telefono.
666666666
> A#agenda{telefono = 911232323}.
#agenda{nombre = "Manuel",apellidos = "Rubio",
        telefono = 911232323}
> #agenda{nombre = "Juan Antonio", apellidos = "Rubio"}.
```

```
#agenda{nombre = "Juan Antonio",apellidos = "Rubio",
        telefono = undefined}
```

Recordemos siempre que la asignación sigue siendo única.

Para acceder al contenido de un dato de un campo del registro, accederemos indicando que es un registro (dato#registro, A#agenda en el ejemplo) y después agregaremos un punto y el nombre del campo al que queremos acceder.

Para modificar los datos de un registro existente en lugar del punto emplearemos las llaves. Dentro de las llaves estableceremos tantas igualdades clave-valor como necesitemos (separadas por comas), tal y como se ve en el ejemplo anterior.

Para obtener en un momento dado información sobre los registros, podemos emplear la función `record_info/2`. Esta función tiene dos parámetros, el primero es un átomo que puede contener **fields** si queremos que retorne una lista de átomos con el nombre de cada campo; o **size**, para retornar el número de elementos que tiene la tupla donde se almacena el registro.

Importante

Como se ha dicho anteriormente, los registros son entidades que trabajan a nivel de lenguaje pero Erlang no los contempla en tiempo de ejecución. Esto quiere decir que el preprocesador trabaja para convertir cada instrucción concerniente a registros para que sean relativas a tuplas y por tanto la función `record_info/2` no podemos emplearla con variables:

```
> A = agenda, record_info(fields, A).
```

Nos retornará ***illegal record info***.

Como los registros son internamente tuplas cada campo puede contener a su vez cualquier otro tipo de dato, no solo átomos, cadenas de texto o números, sino también otros registros, tuplas o listas. Con ello, esta estructura nos propone un sistema organizativo interesante para poder acceder directamente al dato que necesitemos en un momento dado facilitando la labor del programador enormemente.

1.7. Mapas

Los mapas son estructuras de datos. Cada elemento se almacena bajo un índice o clave y contiene un valor. El índice puede ser de cualquier tipo al igual que su contenido. Podemos definir un mapa de la siguiente forma:

```
> M = #{ nombre => "Manuel" }.
#{nombre => "Manuel"}
```

Podemos agregar y cambiar datos del mapa tal y como haríamos con un registro, de la siguiente forma:

```
> M2 = M#{ apellido => "Rubio" }.
#{apellido => "Rubio",nombre => "Manuel"}
```

También podemos indicar un cambio sobre un índice existente. El cambio de indicador y su función son útiles para modificar datos definidos inicialmente en el mapa. Si el dato no existe, el sistema fallará:

```
> M3 = M2#{ nombre := "Juan" }.
#{apellido => "Rubio",nombre => "Juan"}
> M4 = M3#{ telefono := 666555444 }.
** exception error: bad key: telefono
     in function  maps:update/3
        called as maps:update(telefono,666555444,#{apellido
=> "Rubio",nombre => "Juan"})
        *** argument 3: not a map
```

Para extraer un valor debemos realizar concordancia de valores. En el ejemplo anterior, para extraer el nombre:

```
> #{ nombre := Nombre } = M3.
#{apellido => "Rubio",nombre => "Juan"}
> Nombre.
"Juan"
```

En este caso se emplea el símbolo := para indicar la existencia requerida de la clave para obtener su valor.

Curiosidad

El EEP-0043[2] indica la posibilidad de extraer un valor simple empleando otra sintaxis más fácil y sin requerir el uso de una variable para la concordancia. En la versión OTP 24 aún no ha sido implementado aunque aparece en el estándar. Dado que los mapas fueron implementados en la versión 17 no está muy claro que esta característica llegue a estar finalmente implementada:

```
> M3#{ nombre }.
"Juan"
```

El módulo ***maps*** contiene las funciones que permiten eliminar una clave, buscar usando una función en lugar de la concordancia y algunas otras

[2] http://www.erlang.org/eeps/eep-0043.html

opciones. También disponemos de funciones en el módulo de `erlang` (BIF) para poder emplearlas como guardas y detectar si es un mapa o no, obtener el número de claves o si disponemos de una clave con un nombre concreto:

```
> maps:remove(nombre, M3).
#{apellido => "Rubio"}
> maps:get(nombre, M3).
"Juan"
> is_map(M3).
true
> map_size(M3).
2
> maps:keys(M3)
[apellido,nombre]
> is_map_key(nombre, M3).
true
```

Recomiendo revisar la documentación del módulo para ver las posibilidades con estas y otras funciones.

1.8. Conversión de datos

Es importante saber convertir tipos de datos. Principalmente para poder cambiar un dato de tipo cadena de caracteres a cadena binaria, o hacia un átomo o como número, o convertir una tupla o un mapa a una lista o viceversa.

Muchas veces es necesario convertir a otro formato una entrada de datos recogida desde un fichero, una conexión desde/hacia otra máquina o desde una entrada estándar. Estas comunicaciones nos proporcionan los datos siempre como listas binarias y poder darles otro formato depende de nosotros.

Las conversiones más útiles son las que podemos realizar entre los formatos de cadena de caracteres y de tipo numérico. Por ejemplo, si tenemos una cadena de caracteres conteniendo un número decimal y queremos convertirlo a número, ejecutamos la siguiente función:

```
> list_to_integer(101).
101
```

También podemos indicar a la función otra base diferente de la decimal para convertir el texto como 2 (binario), 8 (octal), 16 (hexadecimal) o 36 (alfanumérica):

```
> list_to_integer("101", 2).
5
> list_to_integer("101", 8).
65
> list_to_integer("101", 16).
```

```
257
> list_to_integer("101", 36).
1297
> list_to_integer("101", 64).
** exception error: bad argument
   in function  list_to_integer/2
    called as list_to_integer("101",64)
    *** argument 2: not an integer in the range 2 through 36
```

Como nos dice el error, el número de base máximo para la representación de un número es 36 y el inferior es 2. Si intentamos indicar un número superior o inferior al rango dado, veremos el mismo error. Esto es debido al rango de símbolos elegido al especificar la base: 10 dígitos numéricos y 26 letras.

Curiosidad

A partir de la versión R16 de Erlang se agregaron las funciones para convertir desde binario a entero y viceversa. Anteriormente había que convertir primero a lista para poder convertir después o a entero o a binario.

Las funciones de conversión son las siguientes:

atom_to_list/1, atom_to_binary/2

Estas funciones se encargan de convertir un dato de átomo a lista de caracteres o binario. El segundo parámetro en la conversión a binario indica el conjunto de caracteres a emplear. Los más usados son ***utf8*** y ***latin1***, pero existen muchos más.

binary_to_atom/2, binary_to_float/1, binary_to_integer/1-2, binary_to_list/1-3

Desde binario a átomo, número en coma flotante, entero y lista de caracteres. Para el átomo hay que indicar el conjunto de caracteres. Para entero opcionalmente podemos indicar la base y para lista de caracteres podemos especificar el inicio y fin (comenzando a contar desde 1) para tomar únicamente los elementos del binario a incluir en la lista de caracteres.

```
> binary_to_atom(<<"1234">>, utf8).
'1234'
> binary_to_float(<<"1234">>).
** exception error: bad argument
   in function  binary_to_float/1
    called as binary_to_float(<<"1234">>)
    *** argument 1: not a textual representation of a float
> binary_to_float(<<"1234.0">>).
1234.0
> binary_to_integer(<<"1234">>).
1234
```

```
> binary_to_integer(<<"1234">>, 16).
4660
> binary_to_integer(<<"ff">>, 16).
255
```

list_to_atom/1, list_to_binary/1, list_to_float/1, list_to_integer/1-2, list_to_pid/1

Convierte el contenido de la lista al tipo de dato especificado en la función.

integer_to_binary/1-2, integer_to_list/1-2, float_to_binary/1-2, float_to_list/1-2

Vimos ejemplos de estas funciones anteriormente. El número almacenado en la variable es representado y almacenado en una variable de tipo binario o lista de caracteres. El segundo parámetro permite especificar la base en la que será convertido el número.

list_to_tuple/1, tuple_to_list/1

Convierte una lista en una tupla y una tupla en una lista, respectivamente:

```
> list_to_tuple([1,2,3,4]).
{1,2,3,4}
> tuple_to_list({1,2,3,4}).
[1,2,3,4]
```

maps:to_list/1, maps:from_list/1

Convierte un mapa en una lista y una lista en un mapa. Convertir un mapa a una lista no suele ser problema, pero una lista no puede siempre ser convertido a un mapa. La conversión solo puede llevarse a cabo si la lista está formada estrictamente por tuplas de dos elementos.

2. Imprimiendo por pantalla

Muchas veces se nos presentará la necesidad de mostrar datos por pantalla. De momento, toda la información que vemos es porque la consola nos la muestra, como resultado de salida del código que vamos escribiendo. No obstante, hay momentos, en los que será necesario realizar una salida concreta de un dato con información más completa.

Para ello tenemos el módulo ***io***, del que emplearemos de momento solo la función ***format***. Esta función nos permite imprimir por pantalla la información que queramos mostrar basado en un formato específico que se pasa como primer parámetro.

Por ejemplo, si quieres mostrar una cadena de texto por pantalla, podemos escribir lo siguiente:

```
> io:format("Hola mundo!").
Hola mundo!ok
```

Esto sale así porque el retorno de la función es ***ok***, por lo que se imprime la cadena de texto y seguidamente el retorno de la función (el retorno de función se imprime siempre en consola). Para hacer un retorno de carro, debemos de insertar un carácter especial. A diferencia de otros lenguajes donde se usan los caracteres especiales, Erlang no usa la barra invertida, sino que emplea la virgulilla (~), y tras este símbolo, los caracteres se interpretan de forma especial. Tenemos:

~

Imprime el símbolo de la virgulilla.

c

Representa un carácter que será reemplazado por el valor correspondiente pasado en la lista como segundo parámetro. Antes de la letra ***c*** se pueden agregar un par de números separados por un punto. El primer número indica el tamaño del campo y la justificación a izquierda o derecha según el signo positivo o negativo del número. El segundo número indica las veces que se repetirá el carácter. Por ejemplo:

```
> io:format("[~c,~5c,~5.3c,~-5.3c]~n", [$a,$b,$c,$d]).
[a,bbbbb,  ccc,ddd  ]
ok
```

e / f / g

Se encargan de presentar números en coma flotante. El formato de ***e*** es científico (X.Ye+Z) mientras que ***f*** lo presenta en formato con coma fija. El formato ***g*** es una mezcla ya que presenta el formato científico (***e***) si el número se sale del rango [0.1,10000.0], y en caso contrario presenta el formato de coma fija (***f***). Los números que se pueden anteponer a cada letra indican, el tamaño que se quiere representar y justificación (como se vió antes). Tras el punto la precisión. Unos ejemplos:

```
> io:format("[~7.2e,~7.2f,~7.4g]", [10.1,10.1,10.1]).
[ 1.0e+1,  10.10,  10.10]ok
> Args = [10000.67, 10123.23, 1220.32],
> io:format("~11.7e | ~11.3f | ~11.7g ", Args).
1.000067e+4 |   10123.230 |    1220.320 ok
```

s

Imprime una cadena de caracteres. Similar a ***c***, pero el significado del segundo número en este caso es la cantidad de caracteres de la lista que se mostrará. Veamos algunos ejemplos:

```
> Hola = "Hola mundo!",
> io:format("[~s,~-7s,~-7.5s]", [Hola, Hola, Hola]).
[Hola mundo!,Hola mu,Hola   ]ok
```

w / W

Imprime cualquier dato con su sintaxis estándar. Se usa sobretodo para poder imprimir tuplas, pero imprime igualmente listas, números, átomos, etc. La única salvedad, es que una cadena de caracteres será considerada como una lista. Los números antepuestos se emplean de la misma forma que en ***s***. Un ejemplo:

```
> Data = [{hola,mundo},10,"hola",mundo],
> io:format("[~w,~w,~w,~w]~n", Data).
[{hola,mundo},10,[104,111,108,97],mundo]
ok
```

La versión de ***W*** es similar a la anterior aunque toma dos parámetros de la lista de parámetros. El primero es el dato que se va a imprimir, el segundo es la profundidad. Si imprimimos una lista con muchos elementos, podemos mostrar únicamente un número determinado de ellos. A partir de ese número agrega puntos suspensivos. Un ejemplo:

```
> io:format("[~W]", [[1,2,3,4,5],3]).
[[1,2|...]]ok
```

p / P

Es igual que ***w***, pero intenta detectar si una lista es una cadena de caracteres para imprimirla como tal. Si la impresión es demasiado grande, la parte en varias líneas. La versión en mayúscula, también es igual a su homónimo ***W***, aceptando un parámetro extra para profundidad.

b / B / x / X / + / #

Imprimen números según la base indicada. Los números anteriores a cada letra (o símbolo) indican, el primero la magnitud y justificación de la representación y el segundo la base en la que se expresará el número. Todos ellos imprimen números, pero existen diferencias entre ellos.

B imprime el parámetro numérico. Si la base es mayor a 10 las letras serán impresas en mayúsculas. Con ***b*** las letras son impresas en minúsculas.

Con ***X*** y ***x*** son iguales a las anteriores agregando la posibilidad de emplear un prefijo tomado del siguiente parámetro que haya en la lista de parámetros, consecutivo al valor a representar.

El símbolo de almohadilla (#) siempre antepone la base en formato Erlang: 10#20 (decimal), 8#65 (octal), 16#1A (hexadecimal). El símbolo de suma (+) es igual pero escribiendo las letras en minúscula.

Un ejemplo:

```
> io:format("[~.2b,~.16x,~.16#]", [21,21,"0x",21]).
[10101,0x15,16#15]ok
```

i

Ignora el parámetro que toque emplear. Es útil si el formato de los parámetros que se pasa es siempre el mismo y en un formato específico se desea ignorar uno concreto.

n

Retorno de carro, hace un salto de línea, de modo que se pueda separar por líneas diferentes lo que se desee imprimir por pantalla.

Nota

Existe también el módulo ***io_lib*** que dispone también de la función `format`. Esta función a diferencia de la vista durante esta sección en lugar de presentar por pantalla la cadena resultante la retorna como una lista de entrada/salida.

Erlang/OTP

Volumen I: Un Mundo Concurrente

Manuel Angel Rubio Jiménez