Elixir

INTRODUCCIÓN PARA ALQUIMISTAS

MANUEL ÁNGEL RUBIO JIMÉNEZ

Elixir

Introducción para Alquimistas

Manuel Angel Rubio Jiménez

Elixir
Introducción para Alquimistas

Manuel Angel Rubio Jiménez

Resumen

Elixir nació como la solución a un gran problema que recaía sobre BEAM de ser una gran plataforma pero con un lenguaje ***raro***. Con influencia de otros lenguajes y una gran ejecución por parte de José Valim comenzaron a salir las primeras versiones del lenguaje en 2011 y muchos entusiastas comenzaron a dedicarse a mejorar y hacerlo crecer.

Este libro te ayuda a adentrarte en el mundo de Elixir. Cómo nació, cómo es su comunidad, su ecosistema y por supuesto el lenguaje. Las ventajas y lo que lo hacen único e incluso sus debilidades. Todo para ayudarte a dar tus primeros pasos en Elixir o aprender toda la base en caso de tener ya conocimientos previos.

Aprenderás cómo crear aplicaciones cliente-servidor y las mejores prácticas para desarrollar con Elixir, crear proyectos, integrar tu código con el de otros a través de la instalación de dependencias, publicar tus propias librerías y desplegar tus proyectos en producción.

Este libro cubre todos los aspectos principales de Elixir 1.7.

Depósito legal CO-2236-2018.

ISBN 978-84-945523-4-2

[1] http://altenwald.com/book/elixir
[2] https://creativecommons.org/licenses/by-nc-sa/3.0/deed.es

Capítulo 2. El lenguaje

Un mar tranquilo nunca hizo a un marinero hábil.
—Proverbio

La sintaxis de Elixir está basada en otros lenguajes como Ruby y Erlang y agrega el suficiente azúcar sintáctico para dar flexibilidad al programador para escribir código simple. Permite expresar fácilmente el código escrito. No obstante debemos recordar que Elixir es un lenguaje funcional. La especificación de la solución al problema a resolver mediante código difiere un poco al planteamiento en otros lenguajes imperativos como Java, C/C++, Perl o PHP.

En la programación funcional el código parece una función matemática:

```
def area(base, altura), do: base * altura
```

En este ejemplo definimos una función en Elixir. El área de un rectángulo. Los parámetros son ***base*** y ***altura***. El contenido de la función en una sola línea nos retorna la multiplicación de la base por la altura.

Para códigos imperativos como el siguiente no hay una correlación exacta:

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

Podemos obtener algo parecido a través de la construcción ***for*** de Elixir de la siguiente forma:

```
for i <- 1..10,
    clavar(i) == "si",
    do: martillea_clavo(i)
```

En estas construcciones simplificamos las expresiones definiendo qué queremos conseguir y no los pasos explícitos uno a uno. Otra construcción típica de Elixir para este problema podría ser:

```
1..10
|> Enum.filter(&(clavar(&1) == "si"))
|> Enum.each(&martillea_clavo/1)
```

Obteniendo los números del rango ***1..10*** filtramos llamando a la función `clavar/1` para cada uno de los elementos y para la lista restante llamamos a `martillea_clavo/1` para cada elemento.

También podemos expresar con evaluación perezosa el mismo código y obtenemos un sistema de ejecución en cadena:

```
1..10
|> Stream.filter(&(clavar(&1) == "si"))
|> Enum.each(&martillea_clavo/1)
```

Esto nos garantiza recorrer la lista de elementos una única vez aplicando todas las funciones por cada elemento en el momento que se va obteniendo del rango. Es muy recomendable para acciones como procesar las líneas de un fichero, los registros de una base de datos o el flujo de información proveniente desde una conexión de red.

No te preocupes si de momento no entiendes estos códigos a lo largo de los siguientes capítulos iremos abordando cada uno de estos conceptos.

1. El intérprete de Elixir (iex)

La mayoría del código de ejemplo lo vamos a ejecutar en la línea de comandos o el intérprete de Elixir. Antes de continuar te recomiendo que instales Elixir y pruebes a acceder al intérprete ejecutando **iex** en una consola de tu sistema operativo. Puedes ver cómo instalar Elixir en el ApéndiceA, ***Instalación de Elixir***.

Al acceder al intérprete deberías ver algo como lo siguiente:

```
$ iex
Erlang/OTP 21 [erts-10.1.3] [source] [64-bit] [smp:8:8]
 [ds:8:8:10] [async-threads:1] [hipe] [kernel-poll:false]
 [dtrace] ❶

Interactive Elixir (1.7.4) - press Ctrl+C to exit (type h()
 ENTER for help) ❷
iex(1)> ❸
```

❶ La primera línea es información sobre BEAM indicando la versión base de Erlang/OTP. En este caso ejecutamos una OTP 21 y según la versión de erts (10.1.3) es OTP 21.1. Nos indica la instalación desde código fuente (source) ejecutándose en un sistema de 64 bits, con uso de SMP[1], seguido por la configuración del programador de tareas (ds:8:8:10), los

[1]SMP es Symmetric Multi-Processing o Multiprocesamiento Simétrico. Indica que hay varias unidades de procesamiento (cores o núcleos) acceden a la misma memoria.

hilos asíncronos lanzados, el uso de HiPE[2], poll del kernel[3] y dtrace[4].

❷ Información del intérprete de Elixir. Nos proporciona la versión de Elixir que estamos usando (1.7.4), la forma de salir (**Ctrl**+**C**) y cómo obtener ayuda sobre los comandos especiales que podemos emplear dentro del intérprete. Puedes escribir **help** para revisar la ayuda.

❸ El símbolo del sistema nos da información de dónde estamos (**iex**) y el número del comando que vamos a escribir. Esto es útil para poder repetir comandos u obtener una salida anterior en un nuevo comando si olvidamos asignarlo a una variable:

```
iex(1)> 1 + 1
2
iex(2)> v(1) * 2
4
```

En lo sucesivo mostraremos los ejemplos pero recortando el número de comando de este símbolo del sistema para no distraer del código en sí.

Puedes revisar la sección anterior y probar a escribir los códigos de ejemplo proporcionados y jugar un poco antes de proseguir.

Para más información puedes ver el ApéndiceB, ***La línea de comandos***.

2. Azúcar sintáctico

La definición de las funciones de ayuda es **h()** y **help**. Ambas son funciones y pueden incluso ejecutarse de forma diferente como **h** y **help()**. Esto forma parte del azúcar sintáctico. Es una construcción que no agrega ninguna nueva funcionalidad al lenguaje pero facilita la lectura y escritura. Es más fácil escribir **help** como comando en el intérprete que **help()**.

Donde más notaremos las mejoras que propone la sintaxis propia de Elixir es en el uso de funciones, en los bloques ***do..end*** y en la especificación de clausuras o funciones anónimas.

[2]High Performance Erlang (HiPE) o Erlang de Alto Rendimiento es un proyecto de BEAM para dotar a la máquina virtual de compilación nativa y por lo tanto mayor rendimiento.
[3]Poll es una técnica de E/S para delegar la espera por entrada de datos al núcleo del sistema operativo y liberar el programador de tareas.
[4]Herramienta originaria de Solaris pero disponible en BSD y Linux que permite obtener eventos del sistema operativo sobre el uso de recursos como la red, disco, memoria y mucho más.

Iremos viendo más adelante cada uno de estos edulcorantes del lenguaje.

3. Comentarios

Los comentarios son esa parte del código no ejecutable que sirve al programador para aclarar una parte del código. En elixir podemos agregar comentarios empleando el símbolo almohadilla (#). Todo lo que escribamos tras este símbolo será tomado como comentario e ignorado por el compilador.

Podemos ver un ejemplo de comentario en el siguiente código:

```
# Este código calcula el área del rectángulo:
area = base * altura  # multiplica área por base
```

Obviamente no necesitamos tantos comentarios en un código tan pequeño pero nos sirve para ilustrar cómo podemos agregar los comentarios.

4. Tipos de Datos

En Elixir disponemos de distintos tipos de datos desde simples átomos o números hasta estructuras compuestas por otros tipos de datos, mapas y listas. Cada tipo de dato tiene sus propias particularidades y su cometido.

En esta sección nos adentraremos en los tipos de datos para ver cada uno de ellos respondiendo tres preguntas básicas: ¿qué son?, ¿para qué se usan?, ¿cómo se usan?

Cada tipo de dato además tiene asociado un módulo que permite realizar acciones sobre el mismo o conversiones desde otros tipos de datos. No podemos ver en detalle cada función de cada módulo pero agregaremos un enlace a la documentación en línea de cada uno de ellos para que puedas revisar qué otras acciones puedes realizar con ese tipo de dato o qué opciones pueden ser usadas con las funciones mencionadas aquí.

4.1. Átomos

Los átomos son una unidad muy pequeña y auto-explicativa que podemos usar para agregar código más legible y errores más comprensibles en consola al emplear texto en lugar de números para indicar estados, códigos de error o indicadores para configuración entre otros.

Este tipo de dato nos ayuda a dar nombre al contenido de variables sin necesidad de ocupar mucho tamaño en memoria. Por ejemplo es

trivial crear valores numéricos en otros lenguajes para indicar estados, algo como asignar 0 a "nuevo", 1 a "en proceso", 2 a "cancelado" y 3 a "pagado". La lectura del código se puede dificultar e incluso si podemos crear constantes, tenemos el engorro de tener que crearlas y mantenerlas.

Otra solución sería emplear cadenas de texto para obtener mejor código para leer pero con mayor uso de memoria y más lento para procesar si un texto es igual a otro o no.

Para solventar este problema surgieron los átomos. Este tipo de dato se expresa con un texto comprensible para quien lee el código. En el ejemplo anterior, podemos usar los átomos: ***:nuevo***, ***:en_proceso***, ***:cancelado*** y ***:pagado***. El uso de memoria es incluso mejor en comparación con emplear números y el rendimiento mucho mayor que si empleásemos cadenas de texto.

Los átomos se pueden obtener de dos formas diferentes. La primera es escribiendo con la primera letra mayúscula el átomo y la segunda es anteponiendo al texto que emplearemos como átomo el símbolo de dos puntos (:).

```
iex> is_atom Atom
true
iex> is_atom :atom
true
iex> is_atom :my_atom_12
true
```

Hay varios textos que definen un átomo válido como la mezcla de letras, números, signo de subrayado (_) y arroba (@) como únicos valores válidos para formar el átomo. Pero de hecho hay bastantes excepciones. Cualquier operador puede ser un átomo también (***:===*** es un átomo válido). El signo de interrogación se permite pero solo como terminador: ***:is_binary?***.

Para el resto de casos podemos siempre emplear las dobles comillas (") y encerrar el código entre ellas para que nos permita crear átomos como ***:"12"***.

Importante

Cada átomo definido se inserta en una tabla. El número máximo de átomos por defecto es 1.048.576. Si se intentan definir más de este número se produce un error crítico y BEAM termina su ejecución al completo.

Podemos en este ejemplo lo que sucede:

```
iex> 0..2000000
  |> Enum.each(&(String.to_atom("a#{&1}")))
no more index entries in atom_tab (max=1048576)

Crash dump is being written to:
 erl_crash.dump...done
```

El código intenta generar 2 millones de átomos y finaliza la ejecución de BEAM al llegar al límite máximo de átomos.

Para evitar esto es importante no emplear átomos generados de forma dinámica y si en algún caso deben emplearse es preferible usar la función `String.to_existing_atom/1` siempre que sea posible. Esta función fallará si la cadena a transformar no encuentra un átomo ya existente en la tabla de átomos ya empleados.

El módulo ***Atom*** provee únicamente un par de funciones para convertir un átomo a una lista de caracteres con `Atom.to_charlist/1` o una cadena de texto `Atom.to_string/1`.

4.2. Booleanos o Lógicos

Este tipo de dato solo tiene dos representaciones posibles ***true*** o ***false***. Se obtienen principalmente a través del uso de los literales y los operadores de comparación además de los nexos lógicos.

Debido a ser un tipo de dato tan pequeño no hay disponible un módulo para él.

Un poco de azúcar...

En sí la representación interna de este dato no existe. Si pruebas a emplear los átomos ***:true*** y ***:false*** en su lugar verás que obtienes los mismos resultados. Incluso puedes hacer una comparación estricta para obtener la prueba de que en realidad Elixir emplea esos átomos internamente:

```
iex> true === :true
true
```

4.3. Valor nulo *nil*

El valor nulo o ***nil*** es otro átomo con significado especial que permite escribirse si el uso de los dos puntos (:). El significado de este átomo es la ausencia de tipo. Cuando asignamos este valor a una variable estamos indicando que no dispone de tipo.

Podemos verlo de la siguiente forma:

```
iex> :nil
nil
iex> :nil == nil
true
```

Hay que tener presente que en funciones como `Kernel.is_atom/1` usando una variable o el valor directo obtenemos siempre ***true*** ya que se trata de un átomo.

4.4. Números Enteros y Reales

En Elixir puedes trabajar con dos tipos de números: enteros y reales. Los números enteros no tienen decimales y el tamaño representado puede ser tan grande como memoria tengas en el sistema.

Un poco de azúcar...

Para representar número grandes como 1000000 podemos emplear el símbolo de subrayado (_) como separador y mejorar así la legibilidad: 1_000_000. El compilador ignora estos símbolos y toma el número como si lo hubiésemos escrito sin ellos. En librerías como Money[5] si creamos un dato en euros el número entero a emplear lo escribimos en céntimos, por ejemplo así 10_50 para indicar 10 euros y 50 céntimos.

Podemos ver algunos ejemplos:

```
iex> 1 * 2
2
iex> 12_000_500
12000500
iex> 1_000 * 1_000 == 1_000_000
true
```

El módulo ***Integer*** nos provee de funciones útiles para convertir a cadena `Integer.to_string/1`, a lista de caracteres `Integer.to_charlist/1`, y otras más exóticas como

[5] https://github.com/liuggio/money

`Integer.digits/1` para separar los dígitos de un número o su contrapuesta `Integer.undigits/1` u obtener el máximo común divisor de dos números con `Integer.gcd/2`:

```
iex> Integer.to_string(1_000)
"1000"
iex> Integer.to_charlist(1_024)
'1024'
iex> Integer.digits 101
[1, 0, 1]
iex> Integer.undigits v(-1)
101
iex> Integer.gcd 10, 15
5
```

Podemos especificar la base en la que representamos los números enteros anteponiendo un prefijo específico a cada número escrito. Por ejemplo para los números hexadecimales anteponemos ***0x***, para los números binarios ***0b*** y para los octales el prefijo ***0o***. Podemos ver algunos ejemplos a continuación:

```
iex> 0o10
8
iex> 0x10
16
iex> 0b10
2
```

Los números reales se representan en su forma científica si exceden una cierta dimensión. A la hora de escribirlos podemos optar por escribir el número completo empleando como separador decimal el punto (.) o podemos escribirlos de forma científica especificando la base en formato decimal, la letra ***e*** (para indicar exponente en base a 10) y el número del exponente para la base 10. Escribir ***2.0e1*** (o 2 por 10 elevado a 1) nos da como resultado ***20.0***.

En el módulo ***Float*** podemos encontrar funciones para conversión a lista de caracteres `Float.to_charlist/1` o a cadena de texto `Float.to_string/1` funciones para redondear hacia abajo `Float.floor/1-2` y hacia arriba `Float.ceil/1-2`. También disponemos de un redondeo estándar `Float.round/1-2` y una función para obtener dos números enteros que en fracción generen el número dado: `Float.ratio/2`.

```
iex> Float.to_charlist 0.10
'0.1'
iex> Float.to_string 0.10
"0.1"
iex> Float.floor 0.10
0.0
iex> Float.ceil 0.10
```

```
1.0
iex> Float.round 0.10
0.0
iex> Float.round 0.51
1.0
iex> Float.round 0.52
1.0
iex> Float.ratio 0.5
{1, 2}
```

Por último, para convertir de cadena a número realizamos un análisis de la cadena. Debemos tener en cuenta que una cadena puede contener cualquier cosa y la conversión podría ser errónea. El retorno de las funciones `Float.parse/1` e `Integer.parse/1` retorna una tupla con dos elementos. El primer elemento será el número encontrado y segundo elemento el resto de texto que no corresponde al número:

```
iex> Float.parse "0.10"
{0.1, ""}
iex> Float.parse "a1"
:error
iex> Float.parse "1"
{1.0, ""}
```

Como podéis comprobar, si al principio de la cadena no se encuentra ningún número se retorna el átomo ***:error***.

Antes de terminar con los números me gustaría mostrar un ejemplo para ver la magnitud que un número entero puede tomar. Si ejecutamos el siguiente código veremos varias pantallas de números:

```
iex> List.duplicate(2048, 1000) |> Enum.reduce(&(&1 * &2))
213772730161759466399474188010951...
```

En resumen, el código se encarga de repetir mil veces 2048 dentro de una lista y la reducción se encarga de multiplicar cada elemento por el siguiente. Es lo mismo que elevar 2048 a 1000. Por pantalla obtenemos el resultado.

4.5. Variables e Inmutabilidad

Las variables son nombres a los que se asigna un valor. Podemos emplear un nombre que esté compuesto por letras (mayúsculas y/o minúsculas), números y guion bajo o subrayado (_) aunque con algunas salvedades. No podemos emplear una letra mayúscula al inicio del nombre ni podemos emplear ninguna de las palabras reservadas por Elixir.

En el ejemplo inicial definimos una función para calcular el área de un rectángulo. Si en el intérprete de Elixir definimos un par de variables

conteniendo un número entero cada una y realizamos una operación aritmética con ellas veremos el resultado:

```
iex> base = 10
10
iex> altura = 5
5
iex> base * altura
50
```

Las variables referencian valores. Estos valores pueden ser literales, el dato en sí como hemos visto en el ejemplo anterior, pueden ser otras variables o expresiones que contengan llamadas a funciones y cálculos aritméticos o lógicos.

La variable referenciará el valor asignado hasta que se asigne otro diferente.

A diferencia de otros lenguajes las variables en Elixir referencian espacios de memoria y cuando se asigna otro valor referencian a ese nuevo espacio de memoria. La variable en sí no apunta en ningún momento a un espacio mutable de memoria.

En Elixir existe la inmutabilidad para los datos. Es decir, un tipo de dato como una lista, un mapa, una cadena de texto o incluso un número no cambian. Su espacio de memoria no varía una vez se ha reservado para contenerlo. Si queremos modificar un dato se crea un espacio de memoria nuevo con la modificación. Esta forma de tratar la información interna da seguridad al proceso de control de concurrencia y tiende a ser más rápido que modificar partes de un dato existente.

```
iex> a = 10 ❶
10
iex> b = a ❷
10
iex> a = 5 ❸
5
iex> b
10 ❹
```

❶ Asignamos a la variable ***a*** el valor 10. Elixir lo que hace es reservar un espacio de memoria para un número entero y almacena el número 10.

❷ Copiamos la referencia de ***a*** a ***b***. En esta ocasión como el dato no varía de una variable a otra se copia la referencia y ambas apuntan al mismo espacio de memoria.

❸ Asignamos a la variable ***a*** un nuevo espacio de memoria que contiene el valor 5. El espacio de memoria anterior queda inmutable. No se modifica. Pero la variable ***a*** comienza a apuntar al nuevo espacio de memoria que contiene 5.

❹ La variable ***b*** sigue apuntando al espacio de memoria anterior y por lo tanto sigue conteniendo 10.

El uso de memoria no suele ser un problema porque cada proceso tiene su espacio de memoria y cuando una función termina la reserva de memoria se desecha completamente quedando únicamente los datos del retorno de la función. Si el lenguaje trabajase con referencias a otros datos o incluso retornase referencias, no sería seguro realizar esta limpieza o sería mucho más complicado de realizar.

Por este motivo Elixir puede realizar tareas de tiempo real blando. Al finalizar la ejecución de una función todas las variables internas se desechan y son recogidas por el recolector de basura.

4.6. Listas

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

Las listas son una de las potencias de Elixir y otros lenguajes funcionales. Al igual que en Lisp, Elixir maneja las listas como lenguaje de alto nivel, en modo declarativo, permitiendo el uso de herramientas como las listas de comprensión o la agregación y eliminación de elementos específicos como si de conjuntos se tratase.

4.6.1. ¿Qué podemos hacer con una lista?

Podemos definir una lista de elementos de forma directa tal y como la presentamos a continuación:

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

Podemos realizar modificaciones de estas listas agregando o sustrayendo elementos con los operadores especiales **++** y **--** tal y como podemos ver en los siguientes ejemplos:

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

```
[1,3]
```

Otra ventaja de las listas es la forma en la que podemos ir tomando elementos de la cabeza de la lista dejando el resto en otra sublista. Podemos realizar esta acción con esta sencilla sintaxis:

```
iex> [cabeza|cola] = [1,2,3,4]
[1,2,3,4]
iex> cabeza
1
iex> cola
[2,3,4]
iex> [cabeza1,cabeza2|cola] = [1,2,3,4]
[1,2,3,4]
iex> cabeza1
1
iex> cabeza2
2
iex> cola
[3,4]
```

De esta forma tan sencilla podemos implementar los algoritmos de manejo de pilas como son ***push*** (empujar a la pila) y ***pop*** (extraer de la pila) de la siguiente forma:

```
iex> lista = []
[]
iex> lista = [1|lista]
[1]
iex> lista = [2|lista]
[2,1]
iex> [extrae|lista] = lista
[2,1]
iex> extrae
2
iex> lista
[1]
```

De esta forma hemos modificado nuestra lista agregando elementos primero y extrayendo un elemento después en nuestra implementación básica de una cola LIFO[6].

4.6.2. Listas de Caracteres

Las listas de caracteres son un tipo específico de lista. Se trata de una lista homogénea de elementos representables como caracteres. Elixir detecta si una lista en su totalidad cumple con esta premisa para tratarla como lista 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 simples ('). Una demostración:

[6] ***Last In First Out***, último en entrar primero en salir.

```
iex> 'Hola' = [72,111,108,97]
'Hola'
```

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

Importante

Esta forma de tratar las cadenas es muy similar a la que se emplea en lenguaje C, donde el tipo de dato ***char*** es un dato de 8 bits en el que podemos almacenar un valor de 0 a 255 y que las funciones de impresión tomarán como representaciones de la tabla de caracteres en uso por el sistema. En Elixir la única diferencia es que cada dato no es de 8 bits sino que es un entero lo que conlleva un mayor consumo de memoria pero mejor soporte de nuevas tablas como la de UTF-16 o las extensiones de UTF-8 y similares.

Si empleamos librerías nativas de BEAM y relativas a Erlang es muy posible que empleemos este tipo de dato en lugar de las cadenas de caracteres.

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

```
iex> 'Hola, ' ++ 'mundo!'
'Hola, mundo!'
```

Una de las ventajas de la asignación es la capacidad de realizar concordancias. Si en una asignación se encuentra una variable que no ha sido enlazada a ningún valor, automáticamente cobra el valor necesario para que la ***ecuación*** sea cierta. Elixir intenta hacer siempre que los elementos a ambos lados del signo de asignación sean iguales. Un ejemplo:

```
iex> 'Hola, ' ++ quien = 'Hola, mundo!'
'Hola, mundo!'
iex> quien
'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 la concordancia sería mucho más complejo.

Las listas de caracteres también nos permiten realizar interpolación. La interpolación es la capacidad de agregar la salida de un código dentro de una parte de la lista de caracteres. Por ejemplo:

```
iex> IO.puts 'Una hora son #{60 * 60} segundos'
Una hora son 3600 segundos
:ok
```

El resultado de la operación es intercambiado por el código dentro de la cadena. Todo lo que esté dentro de las llaves y con el símbolo de almohadilla delante (#{}) es intercambiado por el resultado de evaluar ese código.

4.7. Colecciones

En general las listas son consideradas colecciones de datos en Elixir. Pero no solo las listas. Podemos encontrarnos otros tipos de datos que se consideran colecciones y sobre ellos podemos aplicar las funciones del módulo ***Enum***.

Las funciones contenidas en ***Enum*** nos permiten realizar acciones sobre las colecciones como iterar cada elemento retornando un resultado por elemento (`Enum.map/2`) o sin retorno (`Enum.each/2`). Además de ordenar (`Enum.sort/1`), obtener solo los valores únicos (`Enum.uniq/1`), buscar un elemento (`Enum.find/2`) y otras funciones muy potentes y configurables.

El requisito para que un dato pueda emplear las funciones de ***Enum*** es que implemente el protocolo ***Enumerable***. Los módulos que implementan ***Enumerable*** son: Date.Range, File.Stream, Function, GenEvent.Stream, HashDict, HashSet, IO.Stream, List, Map, MapSet, Range y Stream.

Por ejemplo si estamos realizando una salida por pantalla de una lista de elementos y queremos numerarlos:

```
iex> ["pan", "leche", "huevos"] |>
...> Enum.with_index() |>
...> Enum.each(fn({item, i}) -> IO.puts("#{i}.- #{item}") end)
0.- pan
1.- leche
2.- huevos
:ok
```

O si estamos desarrollando una página HTML y queremos agregar tres capas (***div***) dentro de otra que las contenga a modo de simular filas y columnas:

```
iex> ["erlang", "elixir", "golang", "rust"] |>
...> Enum.map(&("    <div>#{&1}</div>\n")) |>
...> Enum.chunk_every(2) |>
...> Enum.map(&("<div>\n#{Enum.join(&1)}</div>\n")) |>
...> Enum.join() |>
...> IO.puts()
```

```
<div>
    <div>erlang</div>
    <div>elixir</div>
</div>
<div>
    <div>golang</div>
    <div>rust</div>
</div>

:ok
```

Más adelante veremos más usos de este módulo.

4.8. Cadenas de texto

Las cadenas de texto o cadenas de caracteres son similares a las listas de caracteres. Permiten almacenar cadenas de caracteres con tamaño de byte y permite realizar trabajos específicos con secuencias de bytes o incluso a nivel de bit.

Los literales los escribimos encerrados en comillas dobles ("). La sintaxis es como sigue:

```
iex> "Hola"
"Hola"
iex> <<72,111,?l,?a>>
"Hola"
```

La cadena de texto puede ser representada como lista binaria tal y como hemos visto en la segunda sintaxis. Estas listas no tiene las mismas funcionalidades que las listas vistas anteriormente. Podemos concatenar cadenas unas a otras con el operador **<>** de la siguiente forma:

```
iex> "Hola " <> "mundo!"
"Hola mundo!"
```

En las cadenas de texto también podemos realizar la interpolación. Este método es más sencillo para concatenar elementos dentro de una cadena. Por ejemplo en el caso anterior si uno de los trozos de la cadena está en una variable y queremos construir la cadena completa podemos hacer:

```
iex> quien = "mundo"
"mundo"
iex> "Hola #{quien}!"
"Hola mundo!"
```

La sintaxis de la almohadilla seguida por las llaves (#{}) dentro de una cadena de texto nos permite intercambiar todo lo encerrado entre esas llaves por la evaluación del código encerrado.

Por último, para trabajar con cadenas de texto disponemos del módulo ***String***. Este módulo nos proporciona funciones para poder cambiar el texto a mayúscula (`String.upcase/1`) o a minúscula (`String.downcase/1`) e incluso capitalizar[7] el texto (`String.capitalize/1`). Podemos darle la vuelta al texto (`String.reverse/1`), obtener su tamaño (`String.length/1`), tomar el primer carácter (`String.first/1`), el último (`String.last/1`) u otro de la cadena especificando posición (`String.at/2`). Veamos algunos ejemplos:

```
iex> titulo = "don Quijote de la Mancha"
"don Quijote de la Mancha"
iex> String.upcase titulo
"DON QUIJOTE DE LA MANCHA"
iex> String.downcase titulo
"don quijote de la mancha"
iex> String.capitalize titulo
"Don quijote de la mancha"
iex> String.reverse titulo
"ahcnaM al ed etojiuQ nod"
iex> String.length titulo
24
iex> String.first titulo
"d"
iex> String.last titulo
"a"
iex> String.at titulo, 4
"Q"
```

Además disponemos en el módulo de funciones que nos permiten comprobar si el texto termina con un sufijo dado (`String.ends_with?/2`), o si comienza con un prefijo dado (`String.starts_with?/2`) o incluso si contiene un trozo de texto dado (`String.contains?/2`). Podemos ver cómo funcionan con estos ejemplos:

```
iex> String.ends_with? titulo, "Mancha"
true
iex> String.ends_with? titulo, "Celestina"
false
iex> String.starts_with? titulo, "Don"
false
iex> String.starts_with? titulo, "don"
true
iex> String.contains? titulo, "Quijote"
true
iex> String.contains? titulo, "De"
false
```

Puedes ver estas y más funciones en la documentación del módulo ***String***[8].

[7] Un texto capitalizado es en el que la primera letra está en mayúscula y el resto en minúsculas.
[8] https://hexdocs.pm/elixir/String.html

4.9. Trabajando con Binarios

Las cadenas de caracteres nos permiten trabajar con cadenas de texto como listas binarias. Esta funcionalidad nos permite empaquetar en forma binaria un conjunto de información teniendo control incluso a nivel de bit cada uno de sus componentes.

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 así:

```
iex> <<cabeza::binary-size(1), cola::binary>> = "Hola"
"Hola"
iex> cabeza
"H"
iex> cola
"ola"
```

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:

```
iex> byte_size(cabeza)
1
iex> byte_size(cola)
3
```

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.

4.10. 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::tipo-tamaño(n)***.

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

Como tipo podemos indicar varios elementos. Los tipos tienen una forma compleja pero siguen una sintaxis: ***endian-signo-tipo-unidad***; vamos a ver los posibles valores para cada uno de ellos:

endian

La forma en 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.

```
iex> <<1215261793::big-size(32)>>
"Hola"
iex> <<1215261793::little-size(32)>>
"aloH"
iex> <<1215261793::native-size(32)>>
"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 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. Los tipos disponibles son: ***integer***, ***float***, ***binary***, ***bits*** (alias para bitstring), ***bitstring***, ***bytes*** (alias para binary), ***utf8***, ***utf16*** y ***utf32***. El tipo por defecto es ***integer***.

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 queremos almacenar tres datos de color rojo, verde y azul en 16 bits, tomando para cada uno de ellos 5, 5 y 6 bits respectivamente, tenemos la partición de los bits algo dificultosa. Con este manejo de bits, componer la cadena de 16 bits (2 bytes) correspondiente a los valores 20, 0 y 6, sería así:

```
iex> <<20::size(5), 0::size(5), 60::size(6)>>
<<160, 60>>
```

Nota

Para obtener el tamaño de la lista binaria en bits podemos emplear la función `Kernel.bit_size/1` que nos retornará el tamaño de la lista binaria:

```
iex> bit_size("Hola mundo!")
88
```

En el módulo ***Bitwise*** podemos encontrar no solo funciones sino también operadores para trabajar con bits. Al agregar el módulo para su uso automáticamente se definen en el ámbito actual todos los operadores y funciones. Podemos ver algunos ejemplos de Y (`Bitwise.band/2` o el operador ***&&&***), O inclusivo (`Bitwise.bor/2` o el operador ***|||***), O exclusivo (`Bitwise.bxor/2` o el operador **^^^**) y NO (`Bitwise.bnot/1` o el operador **~~~**):

```
iex> use Bitwise
Bitwise
iex> band 0b101, 0b110
4
iex> 0b101 &&& 0b110
4
iex> bor 0b101, 0b010
7
iex> 0b101 ||| 0b010
7
iex> bnot 0b111
-8
iex> ~~~0b111
-8
iex> bxor 0b101, 0b111
2
iex> 0b101 ^^^ 0b111
2
```

También podemos realizar el desplazamiento a la derecha o izquierda de los bits de un número con las funciones `Bitwise.bsl/2` (mueve el número de bits indicado como segundo parámetro a la izquierda) y `Bitwise.bsr/2` (mueve el número de bits indicado como segundo parámetro a la derecha). Igualmente en lugar de las funciones podemos emplear los operadores **<<<** y **>>>** respectivamente. Podemos ver un ejemplo:

```
iex> bsl 0b011, 1
6
iex> 0b011 <<< 1
6
iex> bsr 0b110, 1
3
iex> 0b110 >>> 1
3
```

4.11. Tuplas

Las tuplas son tipos de datos organizativos en Elixir. 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. 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 va a ser: ruta, nombre, tamaño y fecha de creación.

Esta información se podría almacenar en forma de tupla de la siguiente forma:

```
{ "/home/user", "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. También hemos escrito la fecha y hora a través de tuplas.

4.11.1. 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. Los módulos ***Kernel*** y ***Tuple*** nos ayudan a realizar estas operaciones. Veamos cómo:

Kernel.put_elem/3

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

```
iex> put_elem {:a, :b, :c}, 1, B
{:a, B, :c}
```

Tuple.append/2

Agrega un elemento al final de la tupla:

```
iex> Tuple.append {:a, :b, :c}, :d
{:a, :b, :c, :d}
```

Kernel.elem/2

Obtiene un elemento de la tupla dado su índice:

```
iex> elem {:a, :b, :c}, 1
:b
```

Tuple.delete_at/2

Elimina un elemento de una tupla:

```
iex> Tuple.delete_at {:a, :b, :c}, 1
{:a, :c}
```

Importante

No obstante la modificación de tuplas no es rápida en comparación con las listas. Podemos emplear estos métodos de modificación de tuplas en actualizaciones en caliente de datos para convertir de una versión a otra los estados internos pero no es recomendable realizar este tipo de acciones en el funcionamiento normal de la aplicación.

4.11.2. Listas de Propiedades

Aunque las listas de propiedades puedan verse en muchas ocasiones como un tipo de dato completamente diferente internamente se componen de una lista de tuplas de dos elementos. Es muy útil para la especificación de opciones y configuración.

Podemos emplear esta sintaxis para especificar una lista de propiedades:

```
iex> opts = [enabled: true, updated_at: {2018, 5, 20}]
[enabled: true, updated_at: {2018, 5, 20}]
```

La única restricción es que las claves sean átomos. Los valores pueden contener cualquier valor.

Al tratarse de una lista podemos emplear las funciones de lista para tratar la lista de propiedades. No obstante la lista de propiedades tiene agregado el acceso a sus elementos de forma simplificada a través de *Access*. Veremos en detalle este módulo más adelante en el Capítulo 5, *Meta-Programación*.

Un poco de azúcar...

La sintaxis de la lista de propiedades parte de la forma original de lista conteniendo tuplas de dos elementos donde el primer elemento es siempre un átomo. Lo podemos ver como:

```
iex> [{:enabled, true}]
[enabled: true]
```

El cambio para esta sintaxis se componen del movimiento de los dos puntos (:) al final del átomo y la eliminación de la coma (,) y las llaves ({}). Facilita y simplifica la escritura.

Por ejemplo podemos acceder a los elementos de la lista anterior de la siguiente forma:

```
iex> opts[:enabled]
true
iex> opts[:updated_at]
{2018, 5, 20}
iex> opts[:created_at]
nil
```

Al acceder a cada elemento obtenemos su valor y cuando el elemento no existe obtenemos en respuesta el valor nulo (***nil***).

4.12. Mapas

Los mapas son una estructura de datos donde cada elemento se almacena bajo una clave. La clave puede ser de cualquier tipo igual que su contenido. Podemos definir un mapa de la siguiente forma:

```
iex> m = %{ :nombre => "Manuel" }
%{nombre: "Manuel"}
```

Podemos además cambiar la información contenida en el mapa así:

```
iex> m = %{m | :nombre => "Miguel" }
#{nombre: "Miguel"}
```

Esta sintaxis solo nos servirá para modificar el contenido de una clave existente dentro del mapa. Si necesitamos agregar un nuevo elemento al mapa tenemos dos funciones bajo el módulo ***Map***. La función `Map.put_new/3` agrega el elemento solo si no existe previamente en la lista. La función `Map.put/3` agrega o actualiza una clave para darle un nuevo valor.

Veamos unos ejemplos de uso de estas funciones:

```
iex> m = %{ :nombre => "Manuel" }
%{nombre: "Manuel"}
iex> m = Map.put(m, :apellido, "Rubio")
%{apellido: "Rubio", nombre: "Manuel"}
iex> m = Map.put_new(m, :nombre, "Miguel")
%{apellido: "Rubio", nombre: "Manuel"}
iex> m = Map.put_new(m, :edad, 38)
%{apellido: "Rubio", edad: 38, nombre: "Manuel"}
```

Para extraer un valor podemos proceder de varias formas. Podemos usar concordancia como veremos más adelante en la Sección 2.1, "Concordancia" del Capítulo 3, ***Expresiones, Estructuras y Errores***, también podemos usar los corchetes ([]) para acceder al dato o usando un punto (.) a través del nombre (siempre que sea un átomo).

La sintaxis usando ***Access*** (o los corchetes) nos permite acceder a una clave incluso cuando su clave no es un átomo:

```
iex> m[12]
nil
iex> m[:nombre]
"Manuel"
```

En caso de no existir la clave (como el primer caso) el retorno es ***nil***. Probamos ahora la otra sintaxis:

```
iex> m.nombre
"Manuel"
iex> m.direccion
** (KeyError) key :direccion not found in: %{apellido:
 "Rubio", edad: 38, nombre: "Manuel"}
```

Podemos acceder sin problemas a las claves existentes pero cuando indicamos una clave no existente obtenemos una excepción. Si en el código que estás desarrollando necesitas obtener un valor y este valor debe existir este es un buen método para fallar rápido y poder hacer una corrección temprana.

El módulo ***Map*** contiene las funciones que permiten eliminar una clave, buscar usando una función en lugar de la concordancia, obtener el número de claves y algunas otras opciones más. No obstante otras funciones como `Kernel.is_map/1` o `Kernel.map_size/1` pertenecen al módulo ***Kernel***.

Algunos ejemplos:

```
iex> Map.delete m, :nombre
#{apellido: "Rubio", edad: 38}
iex> Map.get m, :nombre
"Manuel"
iex> is_map m
true
```

```
iex> map_size m
3
iex> Maps.keys m
[:apellido, :edad, :nombre]
```

4.13. Cambios en Profundidad

Uno de los problemas de la inmutabilidad y la anidación de la información es cómo modificar un dato dentro de una lista, a su vez dentro de otra lista, y otra lista y así sucesivamente. No hay forma simple. Debemos extraer el dato a modificar y después modificar todos sus antecesores para introducir las modificaciones.

Los creadores de Elixir han tenido este problema en cuenta y han desarrollado un par de funciones para ayudar a modificar información anidada. Se trata de las funciones `Kernel.get_in/2,3` y `Kernel.put_in/3,4`.

Estas funciones nos permiten especificar una ruta como parámetro y el valor final a modificar. La ruta puede especificarse de dos formas:

```
iex> data = %{
...>   "alpha" => [
...>     primero: %{"one" => 1, "two" => 3},
...>     segundo: %{"yksi" => 1, "kaksi" => 2}
...>   ],
...>   "beta" => 2
...> }
%{
  "alpha" => [
    primero: %{"one" => 1, "two" => 3},
    segundo: %{"kaksi" => 2, "yksi" => 1}
  ],
  "beta" => 2
}
iex> put_in(data["alpha"][:primero]["two"], 2)
%{
  "alpha" => [
    primero: %{"one" => 1, "two" => 2},
    segundo: %{"kaksi" => 2, "yksi" => 1}
  ],
  "beta" => 2
}
iex> put_in(data, ["alpha", :primero, "two"], 2)
%{
  "alpha" => [
    primero: %{"one" => 1, "two" => 2},
    segundo: %{"kaksi" => 2, "yksi" => 1}
  ],
  "beta" => 2
}
```

En la primera especificamos la forma de acceder al elemento a modificar y en la segunda empleamos una lista con cada una de las claves. En el primer caso optamos por la forma provista por ***Access*** empleando

los corchetes ([]) y en el segundo caso el sistema emplea de forma automática también ***Access*** con las claves dadas.

4.14. Conjuntos

Otro tipo de dato que podemos emplear en Elixir son los conjuntos. Estos datos no tienen una sintaxis específica para poder crearse. Debemos emplear siempre el módulo ***MapSet*** para manejarlos.

Veamos un ejemplo con código de cómo trabajar con conjuntos:

```
iex> set = MapSet.new ❶
#MapSet<[]>
iex> set = MapSet.put set, "rojo" ❷
#MapSet<["rojo"]>
iex> set = MapSet.put set, "azul"
#MapSet<["azul", "rojo"]>
iex> set = MapSet.put set, "rojo" ❸
#MapSet<["azul", "rojo"]>
iex> MapSet.member? set, "verde" ❹
false
iex> set = MapSet.delete set, "rojo" ❺
#MapSet<["azul"]>
```

❶ Creamos un mapa vacío.

❷ Agregamos un elemento dentro del conjunto.

❸ Al intentar agregar un elemento duplicado se retorna el conjunto sin modificaciones.

❹ Podemos comprobar si un elemento está dentro del conjunto a través la función `MapSet.member?/2`.

❺ Eliminamos un elemento del conjunto.

Podemos realizar operaciones de más alto nivel para conjuntos como la unión a través de `MapSet.union/2`, la intersección `MapSet.intersection/2`, la resta de conjuntos `MapSet.difference/2` y otras funciones para determinar si dos conjuntos son iguales (`MapSet.equal?/2`), si uno es subconjunto de otro (`MapSet.subset?/2`) e incluso si son disjuntos (que no comparten ningún elemento `MapSet.disjoint?/2`).

4.15. Rangos

Una de las ventajas de Elixir es la evaluación perezosa. Los rangos emplean la evaluación perezosa para definir un conjunto de números

enteros y permitir a las funciones su iteración generando cada número cuando es necesario. La creación del rango lo podemos realizar a través de la función `Range.new/2` especificando el número inicial y final para el rango o a través del operador **..** tal y como podemos ver en estos ejemplos:

```
iex> 1..10
1..10
iex> Range.new 20, 100
20..100
```

Los rangos pueden emplearse como origen de datos en la mayoría de funciones del módulo ***Enum***.

4.16. Sigilos

Los sigilos[9] son representaciones especiales para dar una mayor semántica a un dato específico. Los sigilos pueden ser definidos por el programador y existen algunos predefinidos en Elixir.

La sintaxis de los sigilos comienza con la virgulilla (~)[10] seguida de una letra y uno de los 8 delimitadores válidos para contener la información. Los delimitadores válidos son: /, |, ", ', (), [], {}, <>. Los 4 últimos tienen un símbolo diferente para apertura y cierre mientras que los 4 primeros usan el mismo símbolo tanto para abrir como para cerrar.

Para muchos de los sigilos la ventaja es la posibilidad de escribir un texto dentro sin necesidad de tener que realizar el escapado de cada uno de los caracteres que actúan normalmente delimitadores:

```
iex> ~c[<input type='text' name='nombre'/>]
'<input type=\'text\' name=\'nombre\'/>'
```

Los sigilos definidos en Elixir son:

~c, ~C

Lista de caracteres. Es equivalente a encerrar un texto entre comillas simples ('):

```
iex> ~c(Esto es un texto)
'Esto es un texto'
```

[9] Del latín ***sello***, un símbolo usado en magia: https://es.wikipedia.org/wiki/Sigilo_(magia).
[10] En teclados españoles este símbolo no suele aparecer. Según el sistema operativo puede obtenerse presionando **AltGr** + **4** o la combinación de teclas **Option** + **Ñ** (en Mac además habrá que presionar espacio).

~D

Fecha. Gestionado desde el módulo ***Date***. Veremos más adelante este dato.

~N

Fecha "ingenua"[11]. Gestionado desde el módulo ***NaiveDateTime***. Veremos más adelante este dato.

~r, ~R

Expresiones regulares. Este sigilo nos permite construir una expresión regular y emplearla en comparaciones de forma sencilla. Veremos más adelante las expresiones regulares.

~s, ~S

Cadena de texto. Es equivalente a encerrar un texto entre comillas dobles ("):

```
iex> ~s[Esto es un texto]
"Esto es un texto"
```

~T

Hora. Gesionado desde el módulo ***Time***. Veremos más adelante este dato.

~w, ~W

Lista de palabras. Permite generar una lista de palabras. Cada palabra será una cadena de caracteres:

```
iex> ~w/rojo azul verde/
["rojo", "azul", "verde"]
```

Hemos visto que hay posibilidad en algunos sigilos de emplear letras mayúsculas o minúsculas. Para estos sigilos (***s***, ***c***, ***r*** y ***w***) es una forma de diferenciar la posibilidad de agregar caracteres de escape e interpolación (en caso de las minúsculas) o no permitir ninguno de estos (las letras en mayúscula).

Podemos ver un ejemplo de esto mismo a continuación:

```
iex> ~w{rojo\nverde azul\namarillo gris\nnegro}
["rojo", "verde", "azul", "amarillo", "gris", "negro"]
iex> ~W{rojo\nverde azul\namarillo gris\nnegro}
["rojo\\nverde", "azul\\namarillo", "gris\\nnegro"]
```

[11] Es llamada "ingenua" por no contener información de la zona horaria.

Los caracteres de escape más habituales son los siguientes:

Caracter	**Descripción**
\\	Barra invertida
\a	Sonido de alerta (beep).
\b	Retroceso. Retrocede un caracter.
\d	Suprimir. Simula la pulsación de la tecla suprimir.
\e	Escape. Simula la pulsación de la tecla escape.
\f	Salto de página. No suele tener efecto.
\n	Salto de línea. Mueve el cursor a la siguiente línea.
\r	Retorno de carro. Mueve el cursor al inicio de la línea actual.
\s	Espacio. Escribe un espacio.
\t	Tabulador. Escribe un conjunto de espacios (normalmente 8).
\v	Tabulador vertical. No suele tener efecto.
\0	Caracter nulo.
\xDD	Imprime un caracter dado su índice en hexadecimal.
\uDDDD o \u{DD...}	Imprime un caracter dado su índice en la tabla Unicode.

Veremos en la Sección 12, "Sigilos" del Capítulo 4, ***Las funciones y los módulos*** más sobre cómo definir nuestros propios sigilos.

5. Conversión de datos

A lo largo del capítulo hemos visto los tipos de datos de que disponemos. Hemos visto también algunas conversiones entre cadenas de texto y números y estos dos en átomos y viceversa.

No obstante no todos los datos pueden convertirse en otros tipos. Una cadena de texto cualquiera no puede convertirse en una lista de propiedades o un mapa ni viceversa.

Para convertir cadenas conteniendo números a enteros y reales hemos visto las funciones `Integer.parse/1` y `Float.parse/1`. De la misma forma hemos visto funciones como `Integer.to_string/1` o `Float.to_string/1` para la conversión de números a cadena.

Ahora veremos cómo podemos convertir un mapa en una lista de propiedades y una lista de propiedades en un mapa.

Para convertir a lista de propiedades un mapa la forma más fácil es emplear la función `Map.to_list/1`. Esta función exporta el contenido del mapa en formato a una lista. Si todas las claves del mapa son átomos entonces veremos el formato saliente como una lista de propiedades. Si el tipo de una o más claves fuese diferente entonces veríamos únicamente una lista con tuplas de dos elementos:

```
iex> l = Map.to_list(m)
[apellido: "Rubio", edad: 38, nombre: "Manuel"]
iex> Map.to_list(%{ "inicia" => 10 })
[{"inicia", 10}]
```

En el primer caso todas las claves son átomos y obtenemos una lista de propiedades. En el segundo caso sin embargo la única clave existente es una cadena de caracteres y obtenemos por tanto una lista simple.

Para convertir de nuevo las listas en mapas debemos usar una nueva función del módulo ***Enum***. Esta función es `Enum.into/2`. Como primer parámetro pasamos el dato a convertir y como segundo dato el tipo de dato al que queremos convertirlo.

Los tipos de datos aceptados y vistos de momento son listas y mapas. Veamos cómo funciona para convertir la lista a mapa:

```
iex> Enum.into l, %{}
%{apellido: "Rubio", edad: 38, nombre: "Manuel"}
```

En caso de querer convertir un mapa en lista a través de esta función podemos pasar como segundo parámetro unos corchetes ([]).

6. Imprimiendo por pantalla

La impresión por pantalla nos permite mostrar texto y mensajes en la salida estándar (también conocida en sistemas tipo Unix como ***stdio***[12]). Podemos hacer una impresión por pantalla simple usando:

```
iex> IO.puts "Hola mundo!"
"Hola mundo!"
```

[12] Acortación de ***Standard Input/Output*** o Entrada/Salida Estándar.

```
:ok
```

La función debe retornar siempre ***:ok***. Podemos hacer salida por pantalla de textos, números enteros y reales y átomos. Otras estructuras de datos más complejas deben ser convertidas para poder ser impresas. No obstante disponemos de la función `Kernel.inspect/1`. Esta función presenta como cadena de texto cualquier estructura de datos compleja. Su formato es el mismo que podemos ver por consola:

```
iex> IO.puts "map => #{inspect %{}}"
map => %{}
:ok
```

Como puedes ver podemos emplear la función `Kernel.inspect/1` dentro de la interpolación. De esta forma la salida queda más compacta. También podemos emplear la función `IO.inspect/1` para realizar la salida directa de un dato por pantalla:

```
iex> IO.inspect %{}
%{}
%{}
```

El retorno es el valor pasado como parámetro y por ello podemos ver en la salida primero el mapa impreso y después el valor retornado.

Dentro del módulo ***IO.ANSI*** podemos encontrar muchos comandos para facilitar la visualización de la salida. La función `IO.puts/1` nos permite enviar una lista con cadenas de texto en su interior. Si queremos escribir un mensaje de error resaltando parte del texto podemos hacer:

```
iex> IO.puts [IO.ANSI.red(),
                                      IO.ANSI.blink_slow(),
                                      "¡ERROR!",
                                      IO.ANSI.reset(),
                                      " no pudimos leer el
 fichero correctamente"]
```

La función ***IO.ANSI.reset/0*** nos permite volver al modo de texto inicial después de haber impreso en rojo y con parpadeos el texto ***¡ERROR!***.

Puedes ver más opciones de este módulo en la documentación oficial[13].

[13] https://hexdocs.pm/elixir/IO.ANSI.html