# Bucles a través de la recursividad

Debido a la inmutabilidad, los bucles en Elixir (como en cualquier lenguaje de programación funcional) se escriben de manera diferente a los lenguajes imperativos. Por ejemplo, en un lenguaje imperativo como C, uno escribiría:

```
for(i = 0; i < sizeof(array); i++) {
  array[i] = array[i] * 2;
}
```

En el ejemplo anterior, estamos mutando tanto la matriz como la variable i. La mutación no es posible en Elixir. En cambio, los lenguajes funcionales dependen de la recursividad: una función se llama de forma recursiva hasta que se alcanza una condición que detiene la acción recursiva. No hay datos mutados en este proceso. Considere el siguiente ejemplo que imprime una cadena un número arbitrario de veces:

```
defmodule Recursion do
  def print_multiple_times(msg, n) when n <= 1 do
    IO.puts msg
  end

  def print_multiple_times(msg, n) do
    IO.puts msg
    print_multiple_times(msg, n - 1)
  end
end

Recursion.print_multiple_times("Hello!", 3)
# Hello!
# Hello!
# Hello!
```

Similar al case, una función puede tener muchas cláusulas. Una cláusula particular se ejecuta cuando los argumentos pasados a la función coinciden con los patrones de argumento de la cláusula y su guard se evalúa como true.

Cuando print_multiple_times/2 se llama inicialmente en el ejemplo anterior, el argumento n es igual a 3.

La primera cláusula tiene un guard que dice «use esta definición si y solo si n es menor o igual que 1». Como este no es el caso, Elixir pasa a la siguiente definición de la cláusula.

La segunda definición coincide con el patrón y no tiene guard, por lo que se ejecutará. Primero imprime nuestro mensaje y luego se llama a sí mismo pasando n – 1 como segundo argumento.

Nuestro smg se imprime y print_multiple_times/2 se llama de nuevo, esta vez con el segundo argumento establecido en 1. Debido a que n ahora se establece en 1, el guard en nuestra primera definición de print_multiple_times/2 se evalúa como true, y ejecutamos esta definición particular. El mensaje se imprime y no queda nada para ejecutar.

Definimos print_multiple_times/2 para que, sin importar qué número se pase como segundo argumento, desencadene nuestra primera definición (conocida como caso base) o desencadene nuestra segunda definición, lo que garantizará que estemos exactamente un paso más cerca del caso base.

## Reducir y mapear algoritmos

Veamos ahora cómo podemos usar el poder de la recursividad para sumar una lista de números:

```
defmodule Math do
  def sum_list([head | tail], accumulator) do
    sum_list(tail, head + accumulator)
  end

  def sum_list([], accumulator) do
```

```
    accumulator
  end
end

IO.puts Math.sum_list([1, 2, 3], 0) #=> 6
```

Invocamos sum_list con la lista [1, 2, 3] y el valor inicial 0 como argumentos. Intentaremos cada cláusula hasta encontrar una que coincida de acuerdo con las reglas de coincidencia de patrones. En este caso, la lista [1, 2, 3] coincide con [head | tail] que une head a 1 y tail a [2, 3]. Y accumulator se establece en 0.

Luego, agregamos el encabezado de la lista al head + accumulator y llamamos a sum_list nuevamente, recursivamente, pasando el final de la lista como su primer argumento. La cola volverá a coincidir [head | tail] hasta que la lista esté vacía, como se ve a continuación:

```
sum_list [1, 2, 3], 0
sum_list [2, 3], 1
sum_list [3], 3
sum_list [], 6
```

Cuando la lista está vacía, coincidirá con la cláusula final que devuelve el resultado final de 6.

El proceso de tomar una lista y reducirla a un valor se conoce como algoritmo de reducción y es fundamental para la programación funcional.

¿Qué sucede si en cambio queremos duplicar todos los valores de nuestra lista?

```
defmodule Math do
  def double_each([head | tail]) do
    [head * 2 | double_each(tail)]
  end

  def double_each([]) do
    []
  end
end
```

```
$ iex math.exs
```

```
iex> Math.double_each([1, 2, 3]) #=> [2, 4, 6]
```

Aquí hemos utilizado la recursividad para recorrer una lista, duplicar cada elemento y devolver una nueva lista. El proceso de tomar una lista y mapearla se conoce como algoritmo de mapa.

La recursividad y la optimización de la cola son una parte importante de Elixir y se usan comúnmente para crear bucles. Sin embargo, cuando programe en Elixir, rara vez usará la recursividad como se describe arriba para manipular las listas.

El módulo Enum, que veremos en el próximo capítulo, ya ofrece muchas comodidades para trabajar con listas. Por ejemplo, los ejemplos anteriores podrían escribirse como:

```
iex> Enum.reduce([1, 2, 3], 0, fn(x, acc) -> x + acc end)
6
iex> Enum.map([1, 2, 3], fn(x) -> x * 2 end)
[2, 4, 6]
```

O usando la sintaxis de captura:

```
iex> Enum.reduce([1, 2, 3], 0, &+/2)
6
iex> Enum.map([1, 2, 3], &(&1 * 2))
[2, 4, 6]
```