



# Paradigma Funcional

**Módulo 4:  
Recursividad.  
Evaluación diferida.**

por **Fernando Dodino  
Carlos Lombardi  
Nicolás Passerini  
Daniel Solmirano**

**Versión 2.1  
Abril 2024**

# Contenido

[1 Introducción](#)

[2 Primer ejemplo: Factorial](#)

[2.1 Solución 1: Definición con guardas](#)

[2.2 Solución 2: Definición con pattern matching](#)

[2.3 Tipos](#)

[3 Repaso de inducción](#)

[3.1 Metáfora de las fichas de dominó](#)

[3.2 Metáfora de la fila del cine](#)

[3.3 Recursividad e inducción](#)

[4 Segundo ejemplo: Fibonacci](#)

[5 Ejercicio: determinar si un número es primo](#)

[6 Recursividad con listas](#)

[6.1 length](#)

[6.2 sum](#)

[6.3 last](#)

[6.4 take](#)

[6.5 drop](#)

[6.6 \(!! o devolver el elemento que está en la posición n](#)

[6.7 elem](#)

[6.8 \(++\) o concatenación](#)

[6.9 reverse](#)

[6.10 maximum](#)

[7 Evaluación diferida](#)

[7.1 Trabajo con listas infinitas](#)

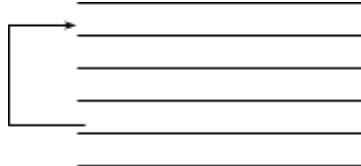
[7.2 Definición de evaluación diferida](#)

[7.3 Ventajas de la evaluación diferida](#)

[8 Resumen](#)

# 1 Introducción

Una función puede llamarse a sí misma, y ese mecanismo recursivo es el que implementa una estructura de repetición en el paradigma funcional.



## 2 Primer ejemplo: Factorial

Repasemos la definición matemática de factorial:

$$n! = 1 \times 2 \times 3 \times \dots \times (n - 1) \times n$$

Genéricamente:

$$n! = \begin{cases} 1 & n = 0 \\ n * (n - 1)! & n > 0 \end{cases}$$

Esta definición matemática la puedo pasar a Haskell, de varias maneras.

### 2.1 Solución 1: Definición con guardas

```
factorial n
| n == 0      = 1
| n > 0       = n * factorial (n - 1)
```

Aquí vemos que cualquier valor entero “matchea” con n (si quiero evaluar factorial 3, la variable n se unifica con el valor 3).

### 2.2 Solución 2: Definición con pattern matching

```
factorial 0  = 1
factorial n  = n * factorial (n - 1)
```

Aquí interviene el concepto de *pattern matching*: tratamos de hacer que encaje un determinado valor en una expresión:

- “factorial 3” no matchea con “factorial 0” (valor vs. valor),
- pero 3 sí es unificable a n (valor vs. variable en el sentido matemático).

Recordamos que como Haskell respeta la **unicidad de las funciones**, tanto las guardas como los patrones son excluyentes: si se cumple alguno, no puedo entrar en otro. Por eso el orden en el que escribimos las condiciones es importante.

## 2.3 Tipos

¿De qué tipo es factorial? Recibo un número, devuelvo otro número.

¿Qué tipo de número? Sólo enteros.

Entonces

```
λ :t factorial  
factorial :: Int -> Int
```

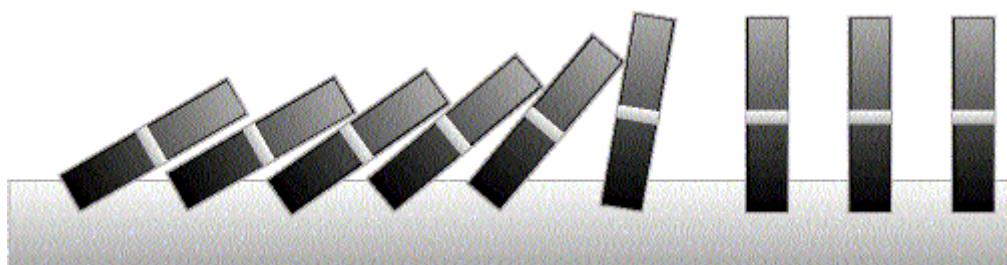
## 3 Repaso de inducción

### 3.1 Metáfora de las fichas de dominó

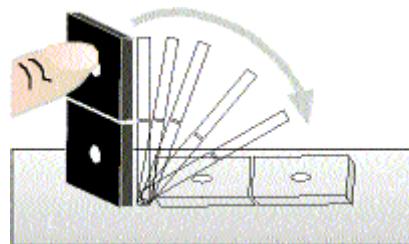
Para repasar inducción, utilizamos como metáfora las fichas de dominó:



Queremos determinar por inducción que si tiramos un dominó, van a caer todos los sucesivos...



- 1) **Caso base:** Si presionamos un dominó en la parte de arriba, cae.



Como el dominó cae al suelo, verificamos que  $P(0)$  se cumple.

2) **Hipótesis:** Si el dominó es contiguo al que cae, caerá también.

### 3) Demostración

Si nos posicionamos en cualquier dominó  $P(N)$ ,  $P(N + 1)$  también caerá por estar contiguo a  $P(N)$ , entonces probamos por inducción que todos los dominó van a caer<sup>1</sup>.

## 3.2 Metáfora de la fila del cine

Llegamos a un cine y queremos sacar entradas para ver una película. Nosotros somos el caso base,  $P(0)$  se cumple porque vamos a sacar la entrada. Hay una fila enorme de gente, le pregunto a cualquiera: ¿es para sacar entrada para la película XXX? Como la respuesta es sí, asumo que el siguiente (y todos los demás) también están haciendo cola para sacar la entrada, entonces dado  $P(N)$  verifco  $P(N + 1)$ .

## 3.3 Recursividad e inducción

El concepto de recursividad viene atado al de **inducción**: verifco  $P(0)$ , y defino  $P(N + 1)$  en base a  $P(N)$ . Volviendo al ejemplo del factorial, primero definimos el caso base, que es el que corta la recursividad: para 0, el factorial es 1:

**factorial 0 = 1**

El factorial de un número positivo mayor a 0 se calcula como n multiplicado por... el factorial de ( $n - 1$ ), y aquí tenemos el caso recursivo:

**factorial n = n \* factorial (n - 1)**

Entonces recordemos: todo algoritmo recursivo debe tener

- un caso base para cortar la recursividad<sup>2</sup>
- un caso recursivo para que verdaderamente exista recursividad

## 4 Segundo ejemplo: Fibonacci

La [secuencia de Fibonacci](#) sigue este algoritmo:

$$f_0 = 1$$

$$f_1 = 1$$

$$f_n = f_{n-2} + f_{n-1}$$

---

<sup>1</sup> Metáfora extraída de [Math is Fun](#)

<sup>2</sup> Para evitar que nos pase como a los habitantes de [Santa Bernardina del Monte](#)

Como vemos, es una definición recursiva. Lo implementamos en Haskell mediante pattern matching:

```
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

Los dos primeros patrones representan los casos base, mientras que el patrón  $n$  encaja con la definición recursiva de la función fibonacci.

## 5 Ejercicio: determinar si un número es primo

Definir una función que me devuelva si un número es primo o no. Matemáticamente, un número es primo si es divisible sólo por sí mismo y por uno ([y son dos números distintos](#)). Esto lo podemos plantear recursivamente...

"Un número es primo si no es divisible por todos los números que lo preceden".

- ¿Cuál es el rango de números que lo preceden? Si tenemos un número  $n$ , los que lo preceden podrían ser  $2, 3, 4 \dots n - 2, n - 1$
- Entonces podríamos definir que
  - 1 no es primo
  - 2 es primo
  - y todo número  $n$  mayor a 2 es primo si no encuentro divisores de  $n$  desde 2 hasta  $(n - 1)$ .

```
primo 1 = False
primo 2 = True
primo n = noHayDivisores 2 (n - 1) n
```

Para saber si no hay divisores de un número en un rango

- si alguno de los números del rango es divisor, entonces pude probar que no se cumple (que no haya divisores)
- si llegué a cubrir todo el rango de números, pude probar que se cumple
- en caso contrario, tengo que continuar con el siguiente número en el rango y repetir la verificación.

```
noHayDivisores minimo maximo n
| mod n minimo == 0 = False
| minimo == maximo = True
| otherwise          = noHayDivisores (minimo + 1) maximo n
```

Podemos mejorar la expresividad de la función noHayDivisores, si consideramos que

`mod n minimo == 0`  
quiere decir que `minimo` es divisor de `n`.

Y la definición nos queda:

```
noHayDivisores minimo maximo n
| minimo `esDivisorDe` n = False
| minimo == maximo      = True
| otherwise              = noHayDivisores (minimo + 1) maximo n

esDivisorDe unNumero otroNumero = mod otroNumero unNumero == 0
```

Para transportar la definición matemática de número primo a Haskell, hemos escrito una función recursiva que utiliza los parámetros para almacenar el contexto de ejecución o estado. Efectivamente `noHayDivisores` necesita al menos dos valores:

- **n**: es el número que queremos verificar si tiene números divisores
- **minimo**: es el que toma valores diferentes ( $2, 3, 4 \dots n - 2, n - 1$ )
- **maximo**: es  $n - 1$  (podríamos no necesitarlo, dado que se puede obtener en función de  $n$ )

Anotamos entonces: una forma de almacenar estado es pasando valores como argumentos, así Haskell va reduciendo los valores hasta determinar si un número es primo:

```
primo 4 = noHayDivisores 2 (4 - 1) 4
          = noHayDivisores 2 3 4
          = False // porque esDivisor 2 4 devuelve True

primo 5 = noHayDivisores 2 (5 - 1) 5
          = noHayDivisores 2 4 5
          = noHayDivisores 3 4 5
          = noHayDivisores 4 4 5
          = True // porque minimo == maximo, 4 == 4 es True
```

Recordemos que una vez que encajé un patrón, no puedo entrar en otro.

**Observación:** Esta es la única vez que les contamos cómo reduce el motor de Haskell las expresiones... el chiste de trabajar en forma declarativa es que no me importe cómo se termina resolviendo y concentrarme en qué es lo que hay que

hacer. Igual si quieren pueden probar [Haskelite](#) para ver cómo se reducen las expresiones.

Busquemos dominio e imagen de la función:

```
primo :: Int -> Bool  
noHayDivisores :: Int -> Int -> Int -> Bool
```

## 6 Recursividad con listas

Repasemos la definición de una lista, que tiene una estructura recursiva:

- el caso base es la lista vacía
- el caso recursivo es una lista de 1 ó más elementos, cuya cabeza es el primer elemento y cuya cola es una lista con los restantes

El segundo patrón es utilizado en una gran variedad de funciones que aplican sobre listas y que veremos a continuación.

### 6.1 length

La longitud de una lista se calcula como

- caso base: 0 si la lista es vacía
- caso recursivo: la longitud de la cola + 1.

```
length [] = 0  
length (x:xs) = 1 + length xs
```

Esta función es uno de los pocos ejemplos en los que es aceptable definir el pattern matching como (x:xs), ya que la longitud aplica a listas de cualquier tipo (no tiene un dominio específico). Fuera de estos casos, deberíamos siempre considerar que x es un nombre poco representativo para una variable.

Entonces, ¿de qué tipo es length?

Si la lista es de cualquier tipo, hablamos en general que es una lista de *a*. Y dada una lista de elementos cualquiera, la longitud es un número entero.

```
length :: [a] -> Int
```

### 6.2 sum

La lógica es similar a la longitud. Si no hay elementos, la suma es 0. Si hay, es el valor de la cabeza + la sumatoria de la cola.

```
sum [] = 0  
sum (x:xs) = x + sum xs
```

El tipo de sum necesita que la lista sea de números que sepan sumarse, por eso

```
sum :: Num a => [a] -> a
```

En el capítulo de inferencia de tipos hablaremos más en detalle de las restricciones que aplican sobre el dominio de las funciones.

## 6.3 last

Last devuelve el último elemento. El algoritmo se puede pensar como:

- el último elemento de una lista de un solo elemento es su cabeza
- o el último elemento de una lista de más de un elemento es el último elemento de la cola

```
last [x] = x
last (x:xs) = last xs
```

Aquí estamos usando como pattern matching del caso base una lista de un elemento, en lugar de una lista vacía. Recordemos que

- [x] implica que la lista tiene un solo elemento (no matchea con dos, tres elementos ni con una lista vacía)
- (x:xs) permite matchear con una lista que tiene al menos un elemento. No obstante, el primer patrón tiene precedencia por lo que el efecto que tiene en la definición de la función last es que el segundo patrón solo encaja para listas de dos o más elementos: no es posible encajar en dos definiciones diferentes por el concepto de unicidad de función.
- ¿Qué pasa si pedimos last de lista vacía? ¿Tiene sentido?

El tipo de last es

```
last :: [a] -> a
```

## 6.4 take

```
λ take 1 ["hola" , "mundo", "loco"]
["hola"]
```

```
λ take 3 [1, 4, 2, 6, 7, 10]
[1, 4, 2]
```

Para tomar los primeros n elementos tenemos dos casos base

- si ya saqué n elementos, no hay más elementos para sacar
- si la lista se vació, no hay más elementos para sacar: ese “no hay más elementos” es la lista vacía

En el caso recursivo, la lista resultante se forma con el elemento que está en la cabeza y los ( $n - 1$ ) elementos que le saque a la cola

```
take n _ | n <= 0 = []
take _ []           = []
take n (x:xs)      = x : take (n - 1) xs
```

Aquí vemos que como no nos interesa utilizar la lista en la primera definición, utilizamos la variable anónima, que se denota con guión bajo o *underscore* (`_`). Lo mismo pasa cuando la lista es vacía, es indistinto el valor que tenga  $n$ , así que usamos nuevamente la variable anónima.

Analicemos el tipo de `take`:

- el primer parámetro es un número entero
- el segundo es una lista de cualquier tipo

```
take :: Int -> [a] -> [a]
```

## 6.5 drop

```
λ drop 3 [1,2,3,4,5]
[4,5]
```

```
λ drop 3 [1,2]
[]
```

Para descartar los primeros  $n$  elementos, tomamos la definición de `take` pero debemos invertir lo que hace cada caso

- el caso base es que ya llegué a la enésima posición, entonces devuelvo la lista que quedó
- el caso recursivo es que estoy sacando los elementos que están antes de la posición  $n$ , entonces simplemente los descarto y sigo llamando a `drop` con la cola y un índice menos

```
drop 0 xs      = xs
drop n (_:xs) = drop (n - 1) xs
```

El tipo de `drop` es

```
drop :: Int -> [a] -> [a]
```

Les dejamos la versión que contempla números negativos o hacer un drop de más posiciones que elementos en la lista, soportada por el Prelude:

```
drop :: Int -> [a] -> [a]
drop _ [] = []
drop n xs | n <= 0 = xs
drop n (_:xs) = drop (n - 1) xs
```

## 6.6 (!! o devolver el elemento que está en la posición n

```
λ "Buenos días" !! 3
'n'
```

```
λ [1..10] !! 5
6
```

Para encontrar el elemento que está en la posición n,

- el caso base es que busco el elemento en la posición 0, es el que está en la cabeza de esa lista
- el caso recursivo es que para encontrar el enésimo elemento de una lista, es equivalente a encontrar el enésimo elemento (- 1) de la cola

```
(!!) (x:_)
 (!!)(_:xs) n = xs !! (n - 1)
```

El tipo del operador (!! ) es

```
(!!) :: [a] -> Int -> a
```

## 6.7 elem

```
λ elem 1 [1,2,3,4,5]
True
```

```
λ elem 3 [1,2,3,4,5]
True
```

```
λ elem 3 [1,2,4,5]
False
```

La función elem determina si un elemento está en una lista.

Los casos base son

- si llegué a la lista vacía, el elemento no está
- si el elemento a buscar está en la cabeza lo encontré

El caso recursivo es verificar si el elemento a buscar está en la cola.

Otra forma de verlo es “un elemento está en una lista si está en la cabeza o está en la cola”, lo que se traduce como:

```
elem _ []      = False
elem e (x:xs) = e == x || elem e xs
```

No podemos repetir la misma variable del lado izquierdo de la igualdad, debemos usar dos variables diferentes y verificar si son iguales.

La función elem necesita que los elementos se puedan comparar por igual:

```
elem :: Eq a => a -> [a] -> Bool
```

## 6.8 (++) o concatenación

```
λ "guarda" ++ "meta"
```

```
"guardameta"
```

```
λ [1..4] ++ [5..7]
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Para concatenar dos listas,

- si la primera lista es vacía, la lista resultante es la segunda lista
- en cambio, si hay elementos en la primera lista, estos elementos formarán parte de la lista resultante (respetando ese orden)

```
concat []      x2 = x2
```

```
concat (x:xs) x2 = x:concat xs x2
```

El tipo del operador (++) o la función concat es

```
concat :: [a] -> [a] -> [a]
```

(las listas deben ser del mismo tipo para poderlas concatenar en una sola)

## 6.9 reverse

```
λ reverse [1,2,3,4,5]
```

```
[5, 4, 3, 2, 1]
```

```
λ reverse "Neuquen"
```

```
"neuqueN"
```

Para dar vuelta el orden de los elementos de una lista

- el caso base es que la reversa de una lista vacía es la lista vacía

- el caso recursivo implica concatenar la inversa de la cola con el elemento que está en la cabeza, solo que el operador (:) separa cabeza y cola que no son del mismo tipo. Entonces utilizamos el pattern matching de una lista con un solo elemento: así tenemos dos listas y se pueden concatenar, de la siguiente manera

```
reverse []      = []
reverse (x:xs) = (reverse xs) ++ [x]
```

El tipo de la función reverse es

```
reverse :: [a] -> [a]
```

## 6.10 maximum

```
λ maximum [1, 12, 3, 24, 5]
24
```

Para obtener el mayor de una lista, comparamos los dos primeros elementos y nos quedamos con el mayor para seguir comparando con el resto de la lista (caso recursivo). El caso base es cuando la lista tiene un solo elemento, y el mayor es ése. No puede haber mayor de una lista vacía.

```
maximum [x] = x
maximum (x:y:ys)
| x > y    = maximum (x:ys)
| otherwise = maximum (y:ys)
```

El lector recordará del módulo anterior el pattern matching (x:y:ys) que implica que la lista tiene al menos dos elementos para poder compararlos.

La función maximum necesita comparar los elementos para poder obtener el mayor, entonces aplicamos una restricción adicional (que veremos en el capítulo de inferencia de tipos):

```
maximum :: Ord a => [a] -> a
```

## 7 Evaluación diferida

### 7.1 Trabajo con listas infinitas

Analicemos la siguiente función:

```
muchosDe n = n:(muchosDe n)
```

Si consultamos en el intérprete

```
λ muchosDe 5
```

Esta función no podría terminar nunca, ya que no hay ningún punto en el que se corte la recursividad. Sin embargo, lo que sí podríamos hacer con esta función es usarla en un contexto que acote la ejecución:

```
λ (head . muchosDe) 5  
5
```

```
λ (sum . take 10 . muchosDe) 5  
50
```

Si el algoritmo diverge, Haskell no puede hacer nada al respecto:

```
λ (sum . muchosDe) 5
```

## 7.2 Definición de evaluación diferida

Como estamos acostumbrados a trabajar en lenguajes con evaluación ansiosa, nos resulta extraño pensar que esta expresión

```
λ (sum . take 10 . muchosDe) 5
```

pueda converger a un valor sin entrar en un loop infinito. Sin embargo, Haskell y muchos otros lenguajes del paradigma funcional trabajan con el concepto de **evaluación diferida o perezosa** (*lazy evaluation*), de manera de evaluar los argumentos a medida que los va necesitando.



Esto es equivalente a pedirle a un ferretero 10 cm. de cable, y que el vendedor desenrolle todo el cable de 100 metros para luego cortar los 10 cms que pedimos. El ferretero del ejemplo trabaja con evaluación ansiosa, nosotros preferimos hacerlo con evaluación diferida.

## 7.3 Ventajas de la evaluación diferida

En los lenguajes imperativos estamos acostumbrados a que las cosas funcionen en forma ansiosa, pero en funcional se nos abre una nueva puerta.

- Con la evaluación diferida sólo se evalúa aquello que realmente se necesita.
  - Como corolario, puedo trabajar con estructuras potencialmente infinitas (como las listas), mientras asegure que el algoritmo converge
- Si una expresión puede andar, con evaluación diferida seguro que anda. Si se rompe, es porque necesité algo que se rompió.
- ¿Por qué no tengo esto en C? Por el efecto colateral. Yo solo puedo cambiar el orden esperado en la evaluación de mis expresiones si sé que este cambio no va a afectar al resto del mundo. Y funcional es un paradigma atemporal: se relaja la idea de secuencia en el paradigma ya que no hay un antes y un después en una función matemática.

## 8 Resumen

A lo largo de este capítulo hemos visto que la recursividad en Haskell se relaciona con la estructura de iteración de los paradigmas imperativos pero también es cercana a la definición por inducción que conocemos de la matemática. Todo algoritmo recursivo necesita un caso base para cortar la recursividad y un caso recursivo.

Otro concepto que aprendimos es la evaluación perezosa, que permite diferir la resolución de expresiones hasta el momento en que sea necesario utilizarlas. Esto permite utilizar listas potencialmente infinitas dentro de algoritmos convergentes y evitar la evaluación de expresiones que luego no se utilicen.