# Tema 3: Paradigma Funcional (III)

## Lenguajes, Tecnologías y Paradigmas de Programación

Escuela Técnica Superior de Ingeniería **Informática**

etsinf

# Índice

# Índice

# Funciones anónimas

- Función anónima (o función sin nombre).
  - Haskell permite definir funciones anónimas de la forma
    `\ x -> e`

  Ejemplo: La función `cuadrado x = x*x` se puede definir en esta notación como `cuadrado = (\ x -> x*x)`

  - En general, $\backslash x_1\ x_2 \ldots x_n$ `->` `e` es equivalente a

$\backslash x_1$ `->` $(\backslash x_2$ `->` $(\cdots$ `->` $(\backslash x_n$ `->` `e`$) \cdots ))$

Ejemplo:

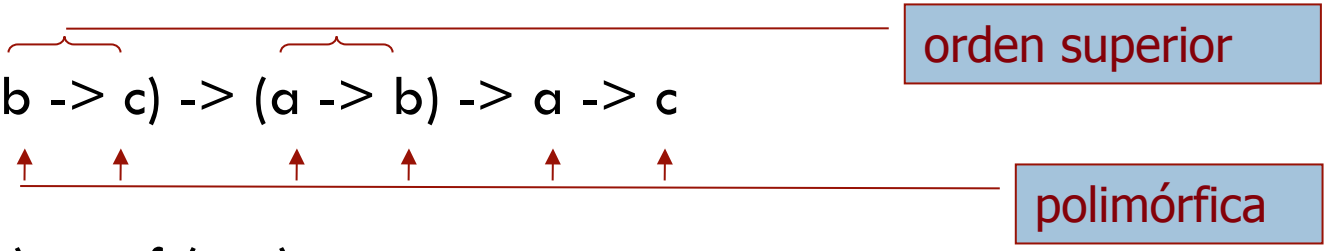
```
sumaCuadrados = \x y -> x*x + y*y
```

es equivalente a:

```
sumaCuadrados = \x -> (\y -> x*x + y*y)
```

# Composición de funciones

- Composición de funciones
  - (.) :: (b -> c) -> (a -> b) -> a -> c




(f . g) x = f (g x)

  - La composición está definida en el **prelude de Haskell** así:

(f . g) = \ x -> f (g x)

  - La composición de funciones es un patrón de cómputo muy habitual. Si la solución a un problema consta de varias etapas podemos abordar cada una como funciones independientes y componer todas para solucionar el problema.

# Composición de funciones

**Ejemplo:**

```
dosveces f x = (f . f ) x
```

point-wise

equivalentemente,

```
dosveces f = f . f
```

point-free

Como función anónima: `dosveces = \ f x -> f (f x)`

# Índice

# Iteradores y compresores

- Iteradores
  - Los iteradores permiten trabajar de forma eficiente en tiempo y memoria con *tipos de datos iterables* (por ejemplo, listas o secuencias).
  - iterate f x devuelve una lista infinita de aplicaciones repetidas de f a x: iterate f x es [x, f x, f (f x), ...]

```
iterate          :: (a -> a) -> a -> [a]
iterate f x      =  x : iterate f (f x)
```

Ejemplo: La función from del prelude de Haskell se define así:

```
from = iterate (1+)
```

# Iteradores y compresores

- Compresores

  Muchas funciones sobre listas se ajustan al esquema recursivo

  ```
  f   :: [a] -> b
  f [] = z
  f (x:xs) = x ⊗ f xs
  ```

  que transforma la lista $x_1:(x_2:(x_3:(x_4:[])))$ en $x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes z)))$

  Ejemplo:

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

  product   :: [Int] -> Int
  product [] = 1
  product (x:xs) = x * product xs
  ```

# Iteradores y compresores

- Podemos pensar en la definición de una función "foldr" que implementa de una vez y para siempre este tipo de transformación:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op z [] = z
foldr op z (x:xs) = x `op` (foldr `op` z xs)
```

- Así la función genérica f vista antes se definiría simplemente como f = foldr (⊗) z

  Y de la misma manera las funciones concretas:
  - sum = foldr (+) 0
  - product = foldr (*) 1

# Iteradores y compresores

- Ejemplo: la función sum del ejemplo de la *longitud de un camino* usando foldr o foldl

```
sum :: [Float] -> Float
sum  = foldr (+) 0.0
```

- Ejercicio Definid las funciones concat, and, or, y map usando foldr.

# El esquema Mapreduce

- El uso combinado de funciones tan optimizadas como map y fold ha inspirado un estilo muy eficiente de procesamiento de secuencias, conocido como *MapReduce*, que tiene gran impacto en el procesamiento de datos a gran escala (> 1Tb), con 1000's procesadores y 100.000's discos.
- El modelo funcional MapReduce fue popularizado por Google y cuenta con importantes aplicaciones:
  - acceso y recuperación de información
  - cloud computing (servicios de computación prestados a clientes externos por empresas como google, yahoo, …)
  - y muchas otras…

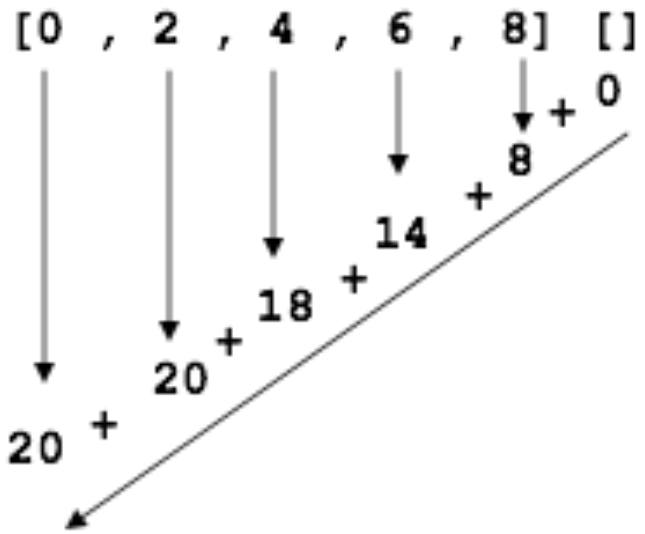
# Uso de Mapreduce

- Su uso en **Google:**
  - Construcción de índices para Google Search
  - Clasificación de artículos para Google News
  - Traducción automática
- Su uso en **Yahoo!:**
  - Yahoo! Search
  - Detección de spam en Yahoo! Mail
- Su uso en **Facebook:**
  - Minería de datos
  - Optimización de anuncios publicitarios
  - Detección de spam

# Uso de Mapreduce

La idea inspiradora:

sumaLista = foldr (+) 0



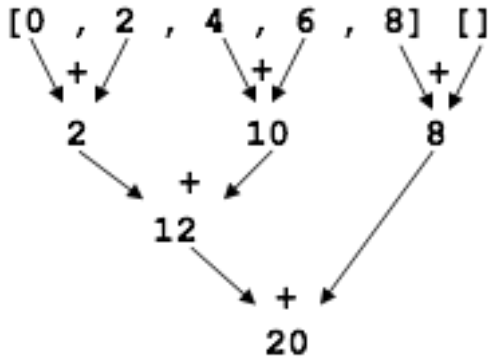

- Esta computación se realiza de izquierda a derecha y *requiere tantos pasos como elementos tiene la lista*
- *Pero la (+) es asociativa y conmutativa !*

  -> ***podemos paralelizar (automáticamente) el proceso***

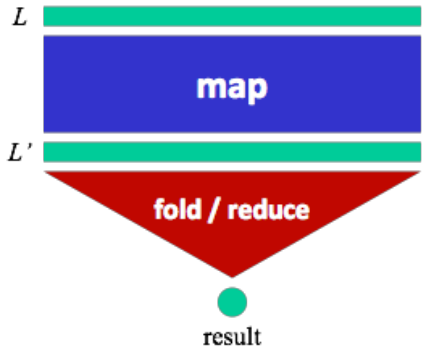
  ***¡y distribuir la carga sobre cientos/miles de procesadores!***

# Uso de Mapreduce

- El coste se reduciría a **O(log n)** si se hiciera de esta forma:

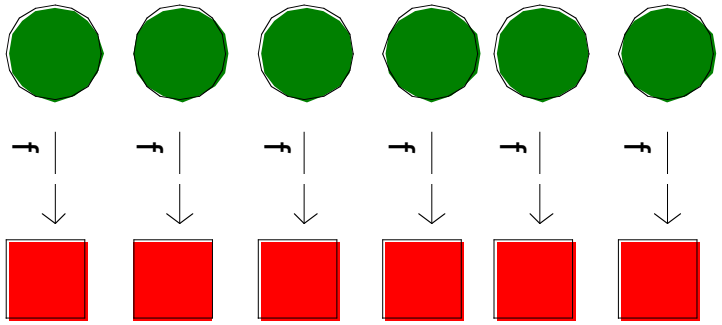



- Podemos lograrlo combinando map y fold adecuadamente.

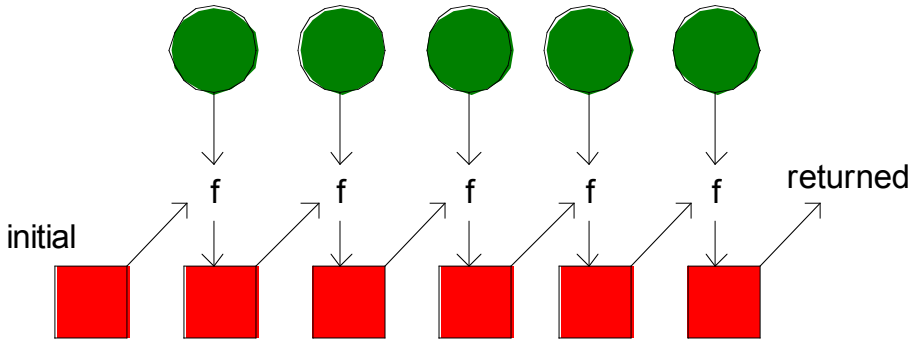



- Y podemos generalizar la idea:
  ***éste es el secreto que explota el esquema MapReduce***

- map f: crea una nueva lista aplicando f a la lista de entrada




- fold f z xs: recorre una lista, aplicando f a cada elemento y acarrendo un *acumulador*.
  La función f *devuelve el nuevo valor del acumulador, que se combina con el siguiente elemento de la lista*




# Uso de Mapreduce

- El esquema MapReduce resulta ser una abstracción muy útil que simplifica y optimiza la computación a gran escala.
- MapReduce ha inspirado el diseño de librerías para otros lenguajes:
  - C++ dispone ahora de una biblioteca MapReduce donde las tareas map() se dividen en bloques de 64 MB
    (la misma talla de los trozos –chunks- del Google File System).
  - Java dispone de una biblioteca similar.
  - Ventajas: permite focalizar en el problema, delegando en la biblioteca los detalles "sucios" (división y acceso por claves, etc).

# Bibliografía


- BÁSICA
  - Bird, R. Introducción a la programación funcional con Haskell, Prentice-Hall, 2000. Traducción de Ricardo Peña.
  - Ruiz, B.C.; Gutiérrez, F.; Guerrero, P.; Gallardo, J.E. Razonando con Haskell, Thomson Editores, 2004.
- HASKELL
  - Lipovaca, M. Learn You a Haskell for Great Good!: A Beginner's Guide. http://learnyouahaskell.com/
  - O'Sullivan B., Goerzen, J, and Stewart D. Real world Haskell, O'Reilly, 2008.