



... for OOP(rogrammers)

Una introducción amable al paradigma funcional

Paradigmas de Programación - UNLaM

¿Qué es Haskell?

1. Es un lenguaje de programación
2. Utiliza el paradigma funcional
 - a. Las funciones son ciudadanos de *primer nivel*.
 - b. Evaluación de expresiones en lugar de ejecución de instrucciones.
3. Es un lenguaje **funcional puro**
 - a. Es inmutable por diseño
 - b. El ser inmutable garantiza la ausencia de efectos secundarios
 - c. Tiene características de idempotencia
4. Es un lenguaje con evaluación tardía (lazy)

Primera prueba con Haskell

1. Abrir el navegador e ir a: <https://repl.it/languages/haskell>

Plan B: <https://tio.run/#haskell>

Plan C: <https://paiza.io/en/languages/haskell>

2. Tipeamos el siguiente código

```
square x = x * x
main = print (square 42)
```

3. Presionamos **Run**

4. ¿Qué obtenemos?

Evaluación temprana (call-by-value)

```
square (1 + 2)
=> square 3
=> 3 * 3
=> 9
```

Evaluación tardía (call-by-name)

```
square (1 + 2)      -- la expresión no se evalúa
=> (1 + 2) * (1 + 2) -- aquí ya debe comenzar a evaluarse
=> 3 * (1 + 2)
=> 3 * 3
=> 9
```

Evaluación tardía (call-by-need)

```
square (1 + 2)      -- la expresión no se evalúa
=> (1 + 2) * (1 + 2) -- aquí ya debe comenzar a evaluarse
=> 3 * 3
=> 9
```

Otro enfoque: ¿qué hace el siguiente código?

```
int[] lst = {2, 3, 5, 7, 11};

int total = 0;
for (int i = 0; i < lst.length; i++)
    total = total + 3 * lst[i]; // indexitis!

System.out.println(total);
```

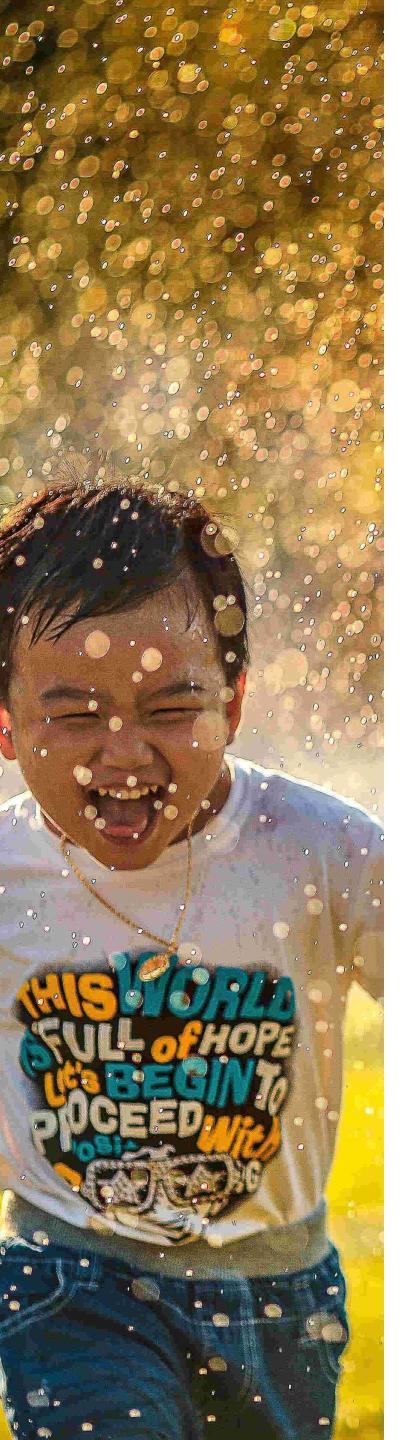
Mismo ejemplo, ahora con Haskell

```
int[] lst = {2, 3, 5, 7, 11};

int total = 0;
for (int i = 0; i < lst.length; i++)
    total = total + 3 * lst[i];

System.out.println(total);
```

```
lst = [2, 3, 5, 7, 11]
total = sum (map (3*) lst)
main = print total
```



Parte 1: La fácil

Funciones propias

```
sumar :: Int -> Int -> Int  
sumar x y = x + y  
  
main = print (sumar 10 15)
```

Entrando en calor

```
sumatoria :: Int -> Int
sumatoria 0 = 0
sumatoria n = n + sumatoria (n - 1)

main = print (sumatoria 10)
```

Condicionales

```
-- https://rosettacode.org/wiki/Hailstone_sequence#Haskell
hailstone :: Int -> Int
hailstone n = if even n
  then n `div` 2
  else 3 * n + 1

main = print (hailstone 3)
```

Ejercicio: Fibonacci

?

Ejercicio: Fibonacci (Resolución)

```
fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)

main = print (fibonacci 10)
```

Funciones sobre listas

```
-- ¿Qué hace este código?  
misterio :: [Int] -> Int  
misterio [] = 0  
misterio (x:xs) = 1 + misterio xs  
  
main = print (misterio [1, 2, 3, 4, 5])
```

Funciones sobre listas

```
-- ¿Y este otro?  
misterioDos :: [Int] -> [Int]  
misterioDos []      = []  
misterioDos (x:xs) = (x * x) : (misterioDos xs)  
  
main = print (misterioDos [1, 2, 3, 4, 5])
```

Ejercicio: Contar los elementos pares de una lista

```
contarPares :: [Int] -> Int
```

Ejercicio: Contar los elementos pares de una lista (resolución)

```
contar :: Int -> Int
contar x = if x `mod` 2 == 0
  then 1
  else 0

contarPares :: [Int] -> Int
contarPares [] = 0
contarPares (x:xs) = (contar x) + contarPares xs

main = print (contarPares [1, 2, 3, 4, 5, 6])
```

Quiz: ¿Cómo hacemos para sumarlos?

Ejercicio: Sumar los elementos pares de una lista (resolución)

```
sumar :: Int -> Int
sumar x = if x `mod` 2 == 0
          then x
          else 0

sumarPares :: [Int] -> Int
sumarPares [] = 0
sumarPares (x:xs) = (sumar x) + sumarPares xs

main = print (sumarPares [1, 2, 3, 4, 5, 6])
```

Bonus: Contar "notables"

```
notable :: Int -> Int
notable x = if x `mod` 2 == 0
            then 1
            else 0

contarNotables :: (Int -> Int) -> [Int] -> Int
contarNotables f [] = 0
contarNotables f (x:xs) = (f x) + contarNotables f xs

main = print (contarNotables (notable) [1, 2, 3, 4, 5, 6])
```

Algunas funciones pre-cocidas sobre listas (1)

zip

Combina dos listas en una lista de tuplas

```
numbers1 = [1, 2, 3, 4]
numbers2 = [10, 20, 30, 40]
zipped = zip numbers1 numbers2
-- Resultado: [(1, 10), (2, 20), (3, 30), (4, 40)]
```

Algunas funciones pre-cocidas sobre listas (2)

fold (foldl / foldr)

Combina los elementos de una lista en un solo valor

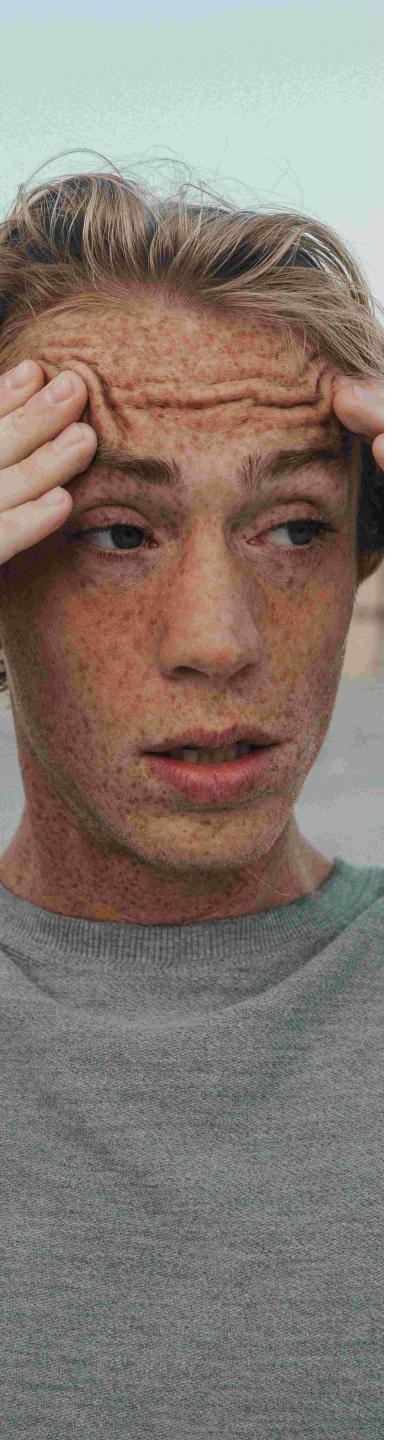
```
numbers = [1, 2, 3, 4, 5]
sumResult = foldl (+) 0 numbers
-- Resultado: 15
```

Algunas funciones pre-cocidas sobre listas (3)

filter

Filtrá una lista por una condición dada

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evenNumbers = filter even numbers
-- Resultado: [2, 4, 6, 8, 10]
```



Parte 2: La moderada

Pattern Matching

```
quitaTres :: [a] -> [a]
quitaTres (_:_:_:xs) = xs
quitaTres _           = []
```

Dos operadores notables

- El operador `++` sirve para concatenar dos listas
- El operador `:`, en cambio, sirve para agregar elementos antes de las listas

```
x:xs    -- válido  
x++xs   -- no válido
```

```
xs:ys   -- no válido  
xs++ys  -- válido
```

Ejercicios:

1. Hacer funciones para manejo de Cola (enqueue / dequeue)
2. Hacer funciones para manejo de Pila (push / pop)

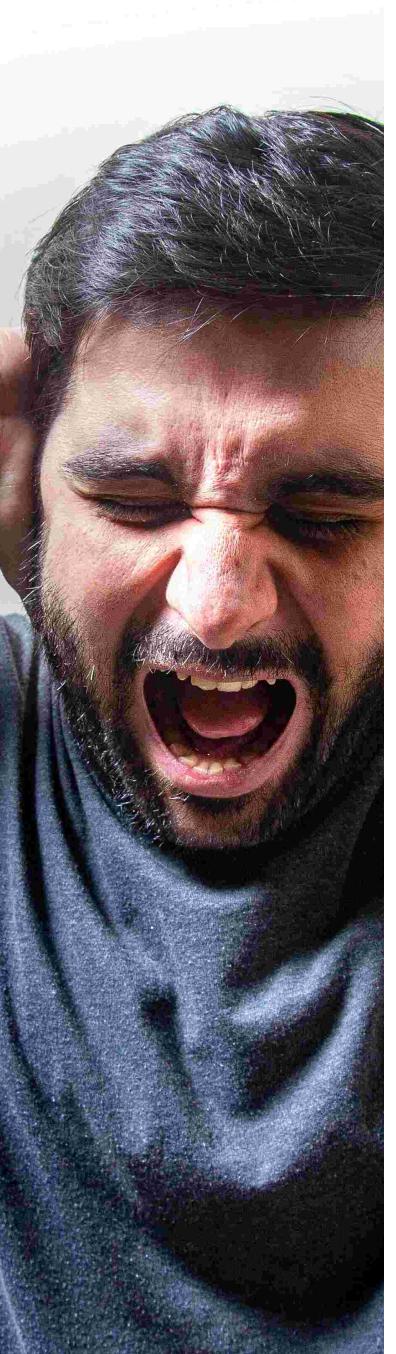
Resoluciones

```
-- cola  
enqueue :: a -> [a] -> [a]  
enqueue x xs = xs++[x]
```

```
dequeue :: [a] -> (a, [a])  
dequeue (x:xs) = (x, xs)
```

```
-- pila  
push :: a -> [a] -> [a]  
push x xs = x:xs
```

```
pop :: [a] -> a  
pop (x:xs) = (x, xs)
```



Parte 3: La difícil

Currying

Es el proceso de transformar una función de múltiples argumentos en una secuencia de funciones que toman un solo argumento. Esto permite la aplicación parcial de argumentos y la creación de funciones más genéricas y flexibles.

```
-- function :: Int -> Int -> Int
function x y = x + y

-- fun :: Int -> Int
fun y = function 3 y

main = print (fun 2)
```

Currying: Un ejemplo práctico

```
sumar x y = x + y
incrementar = sumar 1

main = print (incrementar 10)
```

En el intérprete vemos...

```
> 11
> :type sumar
sumar :: Num a => a -> a -> a
> :type incrementar
incrementar :: Integer -> Integer
```

Otro ejemplo

```
incrementar x = x + 1

twice :: (a -> a) -> a -> a
twice f x = f (f x)

main = print (twice incrementar 0)
-- ¿Cuál es el resultado?
```

Referencias

1. Lipovača, M. (2011). Learn You a Haskell for Great Good! Recuperado de <http://www.learnyouahaskell.com>. Accedido por última vez el 2023-06-05.

