

# **Introducción a la programación funcional con Haskell**

The background of the slide is a repeating pattern of yellow bananas with black outlines, set against a light green background. The bananas are scattered across the entire area.

# **Introducción a la programación funcional con Haskell**

# ¡Hola!



**@nukyma**



**@ nukyma**



**1.**

**¿Por qué HASKELL?**



Universidad  
Rey Juan Carlos



# **PARADIGMAS DE LA PROGRAMACIÓN**



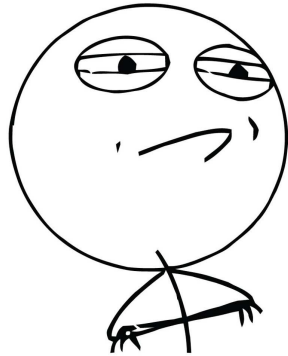
Banana's Edition



!!! Aprende HASKELL en 6 semanas !!!



Banana's Edition



**CHALLENGE ACCEPTED**



**2.**

**¿Qué es un Paradigma  
de Programación?**

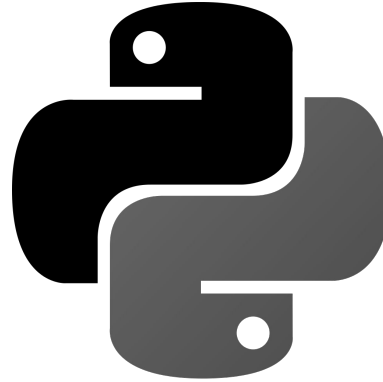
Un **paradigma de programación** es un **estilo** de desarrollo de programas.

Es decir, un **modelo** para resolver **problemas computacionales**.

- Paradigma Imperativo
- Paradigma Orientado a objetos
- Paradigma funcional
- Paradigma lógico



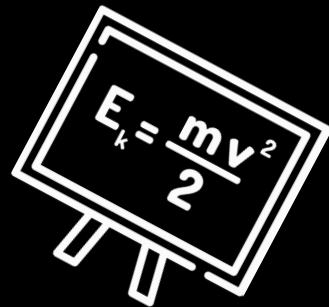
- **PARADIGMA FUNCIONAL**



- **ORIENTACIÓN A OBJETOS**
- **PARADIGMA IMPERATIVO**
- **PARADIGMA FUNCIONAL**

**3.**

**¿Qué es la  
programación funcional?**



# FUNCIONAL

$$E=Mc^2$$



La programación funcional define un programa como una **función matemática** que convierte unas **entradas** en unas **salidas**.



Los programas funcionales expresan mejor **cómo** hay que calcular y no “detallan” tanto **qué** cálculos hay que realizar.

Un programa funcional consiste en una serie de **definiciones por aplicación** y **composición** de otras más simples.

La ejecución de un programa es la **evaluación** de una expresión al aplicar funciones.



# **BASES**

## **DE LA PROGRAMACIÓN FUNCIONAL**

- **COMPOSICIÓN DE FUNCIONES**
- **CONSTRUCCIÓN CONDICIONAL**
- **RECURSIVIDAD**

Banana's Edition

**SERIOUSLY ?**



**4.**

**¿Qué es  
HASKELL?**



*Haskell is a computer programming language.  
In particular, it is*

- **Polymorphically**  
**statically typed**
- **lazy**
- **purely functional**

*language, quite different from most other  
programming languages.*



*The Haskell language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages.*



American mathematician and logician

1900 - 1982

*Haskell is based on the lambda calculus, hence the lambda we use as a logo.*



**5.**

**Haskell Básico**

Un **tipo** es una colección de valores relacionados

**Tipos de datos básicos en Haskell:**

- **Bool**
- **Char**
- **String** (cadena de caracteres)
- **Int** (enteros con precisión fija)
- **Integer** (enteros con precisión arbitraria)
- **Float**
- **Double**



- **Tupla**

- Se define una **n-tupla** como una **colección de n valores** que se tratan como algo unitario. No hay ninguna restricción para los tipos.

- **Lista**

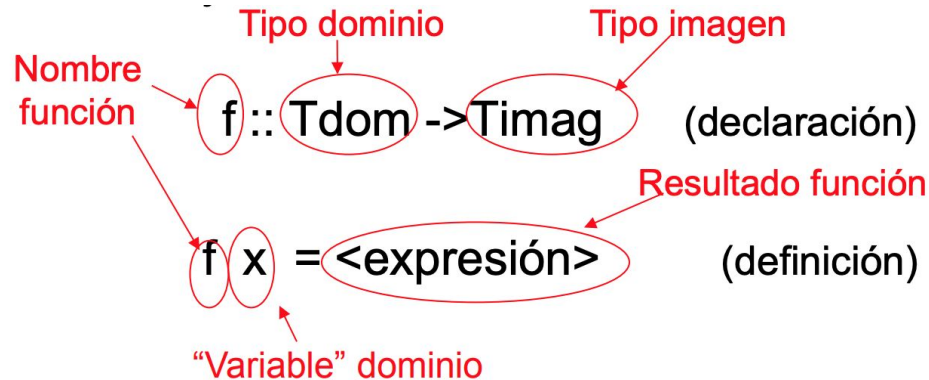
- Restricción de tipos.
- Definición:  $[1, 2, 3, 4, 5, 6] :: [\text{Int}]$   
 $[1, 3..12]$  **equivale a**  $[1, 3, 5, 7, 9, 11]$

### **Listas por comprensión**

$\{x \mid x \in \mathbb{N}, x \text{ es par}\}$ , que denota el conjunto de los  $x$  tales que  $x$  es un número natural par.

- Recorrer: **(x:xs)**  $[1, 2, 3, 4, 5, 6]$   $\mathbf{x=1}$  ,  $\mathbf{xs}=[2,3,4,5,6]$   
 $\mathbf{x=2}$  ,  $\mathbf{xs}=[3,4,5,6]$   
 $\mathbf{x=3}$  ,  $\mathbf{xs}=[4,5,6]$

# Funciones



Función que transforme una temperatura de grados Farenheit a Celsius

`celsius :: Int -> Int` (declaración)

`celsius f = (f - 32) * 5 `div` 9` (definición)

**6.**

**Código Haskell**

Implementar una función que dado un número entero devuelva en una lista todos sus factores.

```
factores :: Integer -> [Integer]
factores n = [ x | x <- [1..n], mod n x == 0 ]
```

Implementar una función que diga cuántos caracteres en mayúscula están contenidos en una frase dada.

```
cuantasMayusculas :: [Char] -> Int
cuantasMayusculas frase = length[ x | x <- frase, isUpper x == True]
```

The background of the slide is a repeating pattern of yellow bananas with black outlines, set against a light green background. The bananas are scattered across the entire area.

# **¿Y LA RECURSIVIDAD?**

**Implementa una función en Haskell que elimine de una lista de enteros aquellos números múltiplo de x.**

```
> cribar [0,5,8,9,-9,6,0,85,-12,15] 2
```

```
[5,9,-9,85,15]
```

**-- Con definición de listas por comprensión**

```
cribar :: [Integer] -> Integer -> [Integer]
```

```
cribar lista num = [ x | x <- lista, x /= num ]
```

Implementa una función en Haskell que elimine de una lista de enteros aquellos números múltiplo de x.

```
> cribar [0,5,8,9,-9,6,0,85,-12,15] 2
```

```
[5,9,-9,85,15]
```

-- Con recursividad no final (La última operación no es recursiva)

```
cribar' :: [Integer] -> Integer -> [Integer]
cribar' [] _ = []
cribar' (x:xs) num
    | x /= num = [x] ++ (cribar' xs num)
    | otherwise = cribar' xs num
```



Implementa una función en Haskell que elimine de una lista de enteros aquellos números múltiplo de x.

```
> cribar [0,5,8,9,-9,6,0,85,-12,15] 2  
[5,9,-9,85,15]
```

-- Con recursividad final o de cola (La última operación es recursiva)

```
cribar' :: [Integer] -> Integer -> [Integer]
```

```
cribar' [] _ = []
```

```
cribar' lista num = cribarAux lista num []
```

```
cribarAux :: [Integer] -> Integer -> [Integer] -> [Integer]
```

```
cribarAux (x:xs) num acum
```

```
    | x /= num = cribarAux xs num [x] ++ acum
```

```
    | otherwise = cribarAux xs num acum
```

## Típica estructura recursiva...

```
f [] = valor
```

```
f (x:xs) = x 'operación' (f xs)
```

# Funciones de Orden Superior

- Función de plegado de listas: **foldr** y **foldl**

**foldr (+) e [w,x,y,z]** es igual al valor de la expresión **(w + (x + (y + (z + e))))**

```
productoLista :: [Int] -> Int
```

```
productoLista = foldr (*) 1
```

```
sumaLista :: [Int] -> Int
```

```
sumaLista = foldr (+) 0
```

```
sumaLista :: [Int] -> Int
```

```
sumaLista = foldr (+) 0
```

sumaLista [2,7,5]

> 5 +  $\emptyset$  = 5

> 7 + 5 = 12

> 2 + 12 = 14

# Funciones de Orden Superior


- Función de plegado de listas + Expresiones LAMBDA

```
f lista = foldl (\lambda) caso_base lista
```

Dada una lista de números enteros obtenga un número entero con el resultado de calcular el doble de cada uno de los elementos de la lista original y sumarlos todos

-- Función de plegado de listas + Expresiones LAMBDA

```
sumaDobles :: [Integer] -> Integer
```




```
sumaDobles lista = foldl (\ acumulador x -> (x*2)+acumulador) 0 lista
```

sumadobles [3,7,2]

→

$$> \underline{X=3} \Rightarrow \underline{(3*2)} + \underline{\emptyset} = \underline{6}$$

$$> \underline{X=7} \Rightarrow \underline{(7*2)} + \underline{6} = \underline{20}$$

$$> \underline{X=2} \Rightarrow \underline{(2*2)} + \underline{20} = \underline{22}$$


**Dada una lista de números enteros obtenga un número entero con el resultado de calcular el doble de cada uno de los elementos de la lista original y sumarlos todos**

**-- Recursividad no final**

```
sumaDobles :: [Integer] -> Integer
```

```
sumaDobles [] = 0
```

```
sumaDobles (x:xs) = sum([x*2] ++ [sumaDobles xs])
```



**Dada una lista de números enteros obtenga un número entero con el resultado de calcular el doble de cada uno de los elementos de la lista original y sumarlos todos**

### **-- Recursividad Final**

```
sumaDobles :: [Integer] -> Integer
```

```
sumaDobles [] = 0
```

```
sumaDobles lista = sumaDoblesAux lista 0
```

```
sumaDoblesAux :: [Integer] -> Integer -> Integer
```

```
sumaDoblesAux [] acumulador = acumulador
```

```
sumaDoblesAux (l:ls) acum = sumaDoblesAux ls (l*2)+acum
```

**Dada una lista de números enteros obtenga un número entero con el resultado de calcular el doble de cada uno de los elementos de la lista original y sumarlos todos**

-- Utilizando expresiones orden superior (MAP)

```
sumaDobles :: [Integer] -> Integer
```

```
sumaDobles lista = sum (map (*2) lista)
```

**7.**

**Otros lenguajes  
funcionales...**



Clojure



Scala

**8.**

**Programación Funcional**  
**Ventajas y Desventajas**

# VENTAJAS

---

- **Evitan estados intermedios**
- **No manejan datos mutables**
- **No tiene ningún efecto lateral (*Transparencia referencial*)**
- **Fácil testear**
- **Validación formal**
- **Alta abstracción**

# INCONVENIENTES

---

- No existe el concepto de posición de memoria.  
NO HAY VARIABLES → No hay necesidad de una instrucción de asignación
- NO HAY BUCLES → Recursividad
- Alta abstracción

# LINKS

- <https://www.haskell.org/>
- <https://wiki.haskell.org/Introduction>
- <http://aprendehaskell.es/main.html>
- <https://relopezbriega.github.io/blog/2015/02/01/programacion-funcional-con-python/>
- [https://github.com/nukyma/haskell\\_class/](https://github.com/nukyma/haskell_class/)



Banana's Edition


**¡Gracias!**



Banana's Edition

**¿Preguntas?**





**¡Hasta el  
próximo  
bananas!**



**@nukyma**

# TRANSPARENCIA REFERENCIAL

El resultado de evaluar una **expresión compuesta** depende únicamente del resultado de evaluar las **subexpresiones que la componen** y de nada más.

El resultado no depende de:

- La historia del programa en ejecución
- El orden de evaluación de las subexpresiones que la componen.

**No tenemos que preocuparnos de que las modificaciones que hagamos en una parte del código afecten los cálculos que se hacen en otra parte.**

Banana's Edition

