

# Práctica 0

Haskell: lo elemental

## Paradigmas de Programación

alias "Paradigmas de Lenguajes de Programación"  
alias "PLP"

Departamento de Ciencias de la Computación  
Universidad de Buenos Aires

16 de agosto de 2024

✨ Podemos definir funciones

```
f x = x + 1
```

✨ Podemos aplicar esas funciones

```
f 5  
(>) 6 1
```

✨ ¿Cuál es la diferencia entre una variable de Haskell y las variables de lenguajes imperativos?

Las diferencias son:

- **Inmutabilidad**: en Haskell las variables son inmutables; una vez asignado un valor, no puede cambiar. En cambio en los lenguajes imperativos las variables son mutables.
- **Declaración y Asignación**: en Haskell la declaración y asignación son simultáneas, mientras que en los lenguajes imperativos se puede declarar y asignar por separado.
- **Efectos Secundarios**: en Haskell la inmutabilidad evita efectos secundarios, mientras que no ocurre lo mismo en los lenguajes imperativos.
- **Referencias y Estados**: en Haskell el estado mutable se maneja a través de monadas como IO o ST, que permiten controlar el estado de manera segura y predecible. En cambio, en lenguajes imperativos las variables actúan como referencias en memoria y pueden modificar el estado del programa.
- **Evaluación**: en Haskell la evaluación es perezosa; los valores se calculan solo cuando se necesitan. En cambio en los lenguajes imperativos la evaluación es estricta; las variables se calculan en el momento de su asignación o uso.

## Ejercicio

```
promedio :: Fractional a => a -> a -> a  
promedio x y = (x + y) / 2
```

Definir en Haskell las siguientes funciones:

- promedio, que toma dos números y devuelve su promedio.
- máximo, que toma dos números y devuelve el mayor.
- factorial, que toma un número entero y devuelve su factorial (el producto de ese número y todos sus anteriores hasta el 1).

```
maximo :: Ord a => a -> a -> a  
maximo x y | x > y = x  
           | otherwise = y
```

```
factorial :: Int -> Int  
factorial 0 = 1  
factorial n = n * factorial (n - 1)
```

## Recursión

✨ Una generalización ingeniosa de una función partida:

```
factorial 0 = 1  
factorial 1 = 1  
factorial 2 = 2  
factorial 3 = 6  
factorial 4 = 24  
factorial 5 = 120  
:  
:
```

✨ ¿Cuánto más tengo que seguir? [Infinitamente](#)

✨ En lugar de definir cada caso de forma aislada, defino un caso en función del otro:

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

✨ ¿Eso lo soluciona para cualquier caso? [Falta el caso base](#)

# Recursión

✧ No se olviden del caso base

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

✧ ¿Sólo se puede hacer recursión sobre números naturales? ¿Sobre qué otras cosas se les ocurre que se puede hacer recursión?

Listas, árboles, cadenas, etc

# Listas

✧ Descripción de una lista por extensión:

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

✧ Descripción de una lista de forma **recursiva**:

```
1 : (2 : (3 : (4 : (5 : []))))
```

✧ ¿Por qué escribiría una lista de esa forma?

# Recursión sobre listas

```
f [] = ...
f (x:xs) = ... x ... (f xs) ...
```

# Recursión sobre listas

```
f [] = c
f (x:xs) = g x (f xs)
```

## Ejemplo

```
incN n [] = []
incN n (x:xs) = (n + x) : incN n xs
```

- ¿Qué hace incN? incN suma n a todos los elementos de una lista
- ¿Qué devuelve incN 2 [3, 2, 3]? [5, 4, 5]
- ¿Qué devuelve incN [2, 3, 2] []?

devuelve un error ya que [3, 2, 3] no es un número que se pueda sumar a cada elemento de una lista

## Tipos

¿De qué tipo son las siguientes expresiones?

```
3 :: Int
True :: Bool
even :: Int -> Bool
[1, 2, 3] :: [Int]
[1, True] :: error
[[1]] :: [[Int]]
[] :: [a]
```

## Variables de tipo

```
[] :: [a]
id :: a -> a
head :: [a] -> a
tail :: [a] -> [a]
const :: a -> b -> a
length :: [a] -> Int
```

## Ejemplo

¿Qué funciones son?

Esta función suma x e y. Solo funciona cuando y es mayor o igual a 0

```
a1 x 0 = x
a1 x y = a1 x (y - 1) + 1
```

Esta función multiplica a x por y. Solo funciona bien para x e y mayores o iguales a 0

```
a2 x 0 = 0
a2 x y = a2 x (y - 1) + x
```

Esta función eleva a x por y. Solo funciona cuando y es mayor o igual a 0

```
a3 x 0 = 1
a3 x y = a3 x (y - 1) * x
```

# Tipo de funciones

```
f1 :: Int -> (Int -> Int)
f2 :: (Int -> Int) -> Int
f3 :: Int -> Int -> Int
```

¿De qué tipo son las siguientes expresiones?

```
f1 5 :: Int -> Int
f1 5 8 :: Int
f3 5 8 :: Int
f3 5 :: Int -> Int
f2 5 :: error
f2 (+1) :: Int
```

# Convenciones de precedencia y asociatividad

✧ Los tipos tienen asociatividad a derecha

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c) \neq (a \rightarrow b) \rightarrow c$$

✧ La aplicación tiene asociatividad a izquierda

$$f \ x \ y = (f \ x) \ y \neq f \ (x \ y)$$

✧ La aplicación tiene mayor precedencia que los operadores binarios

$$f \ x \ + \ y = (f \ x) \ + \ y \neq f \ (x \ + \ y)$$

✧ Los operadores binarios se pueden usar como funciones

$$x \ + \ y = (+) \ x \ y$$

✧ Las funciones se pueden usar como operadores binarios

$$f \ x \ y = x \ 'f' \ y$$

# Tipos de datos algebraicos

```
data Bool = True | False
True :: Bool
False :: Bool
```

```
data Maybe a = Nothing | Just a
Nothing :: Maybe a
Just :: a -> Maybe a

data Either a b = Left a | Right b
Left :: a -> Either a b
Right :: b -> Either a b
```

- ✧ Definir la función `inverso :: Float -> Maybe Float` que dado un número devuelve su inverso multiplicativo si está definido, o `Nothing` en caso contrario.
- ✧ Definir la función `aEntero :: Either Int Bool -> Int` que convierte a entero una expresión que puede ser booleana o entera. En el caso de los booleanos, el entero que corresponde es 0 para `False` y 1 para `True`.

```
inverso :: Float -> Maybe Float
inverso 0 = Nothing
inverso n = Just (1 / n)

aEntero :: Either Int Bool -> Int
aEntero x = case x of
  (Left n) -> n
  (Right b) -> if b then 1 else 0
```