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

Haskell: lo elemental

- ✨ Podemos definir funciones

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

Haskell: lo elemental

✨ Podemos definir funciones

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

✨ Podemos aplicar esas funciones

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

Haskell: lo elemental

✨ 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?

Ejercicio

Definir en Haskell las siguientes funciones:

a) `promedio`, que toma dos números y devuelve su promedio.

Ejercicio

Definir en Haskell las siguientes funciones:

a) `promedio`, que toma dos números y devuelve su promedio.

b) `máximo`, que toma dos números y devuelve el mayor.

Ejercicio

Definir en Haskell las siguientes funciones:

a) `promedio`, que toma dos números y devuelve su promedio.

b) `máximo`, que toma dos números y devuelve el mayor.

c) `factorial`, que toma un número entero y devuelve su factorial (el producto de ese número y todos sus anteriores hasta el 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
```

⋮

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?

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?

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

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

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?

✨ 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?

Recursión

✨ No se olviden del caso base

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

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

✨ Descripción de una lista por extensión:

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

Listas

✨ Descripción de una lista por extensión:

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

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

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

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
```

Ejemplo

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

✨ ¿Qué hace `incN`?

Ejemplo

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

✨ ¿Qué hace `incN`?

✨ ¿Qué devuelve `incN 2 [3, 2, 3]`?

Ejemplo

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

- ✨ ¿Qué hace `incN`?
- ✨ ¿Qué devuelve `incN 2 [3, 2, 3]`?
- ✨ ¿Qué devuelve `incN [2, 3, 2] []`?

Tipos

¿De qué tipo son las siguientes expresiones?

```
3
True
even
[1, 2, 3]
[1, True]
[[1]]
[]
```

Tipos

¿De qué tipo son las siguientes expresiones?

```
   3 :: Int
True
even
[1, 2, 3]
[1, True]
[[1]]
[]
```

Tipos

¿De qué tipo son las siguientes expresiones?

```
   3 :: Int
True :: Bool
even
[1, 2, 3]
[1, True]
[[1]]
[]
```

Tipos

¿De qué tipo son las siguientes expresiones?

```
   3 :: Int
True :: Bool
even :: Int -> Bool
```

```
[1, 2, 3]
[1, True]
  [[1]]
    []
```

Tipos

¿De qué tipo son las siguientes expresiones?

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

Tipos

¿De qué tipo son las siguientes expresiones?

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

Tipos

¿De qué tipo son las siguientes expresiones?

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

Tipos

¿De qué tipo son las siguientes expresiones?

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

Variables de tipo

```
[]
id
head
tail
const
length
```

Variables de tipo

```
   [] ::  [a]
   id
 head
 tail
const
length
```

Variables de tipo

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

Variables de tipo

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

Variables de tipo

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

Variables de tipo

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

Variables de tipo

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

Ejemplo

¿Qué funciones son?

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

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

```
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
```

Tipo de funciones

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

¿De qué tipo son las siguientes expresiones?

Tipo de funciones

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

¿De qué tipo son las siguientes expresiones?

```
  f1 5
f1 5 8
f3 5 8
  f3 5
  f2 5
f2 (+1)
```

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
f3 5 8
  f3 5
  f2 5
f2 (+1)
```

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
   f3 5
   f2 5
f2 (+1)
```

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
   f2 5
f2 (+1)
```

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
f2 (+1)
```

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)
```

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

$$\texttt{a -> b -> c} = \texttt{a -> (b -> c)} \neq \texttt{(a -> b) -> c}$$

Convenciones de precedencia y asociatividad

- ✨ Los tipos tienen asociatividad a derecha

$$\texttt{a -> b -> c} = \texttt{a -> (b -> c)} \neq \texttt{(a -> b) -> c}$$

- ✨ La aplicación tiene asociatividad a izquierda

$$\texttt{f x y} = \texttt{(f x) y} \neq \texttt{f (x y)}$$

Convenciones de precedencia y asociatividad

- ✨ Los tipos tienen asociatividad a derecha

$$\texttt{a -> b -> c} = \texttt{a -> (b -> c)} \neq \texttt{(a -> b) -> c}$$

- ✨ La aplicación tiene asociatividad a izquierda

$$\texttt{f x y} = \texttt{(f x) y} \neq \texttt{f (x y)}$$

- ✨ La aplicación tiene mayor precedencia que los operadores binarios

$$\texttt{f x + y} = \texttt{(f x) + y} \neq \texttt{f (x + y)}$$

Convenciones de precedencia y asociatividad

- ✨ Los tipos tienen asociatividad a derecha

$$\texttt{a -> b -> c} = \texttt{a -> (b -> c)} \neq \texttt{(a -> b) -> c}$$

- ✨ La aplicación tiene asociatividad a izquierda

$$\texttt{f x y} = \texttt{(f x) y} \neq \texttt{f (x y)}$$

- ✨ La aplicación tiene mayor precedencia que los operadores binarios

$$\texttt{f x + y} = \texttt{(f x) + y} \neq \texttt{f (x + y)}$$

- ✨ Los operadores binarios se pueden usar como funciones

$$\texttt{x + y} = \texttt{(+) x y}$$

Convenciones de precedencia y asociatividad

- Los tipos tienen asociatividad a derecha

$$\texttt{a -> b -> c} = \texttt{a -> (b -> c)} \neq \texttt{(a -> b) -> c}$$

- La aplicación tiene asociatividad a izquierda

$$\texttt{f x y} = \texttt{(f x) y} \neq \texttt{f (x y)}$$

- La aplicación tiene mayor precedencia que los operadores binarios

$$\texttt{f x + y} = \texttt{(f x) + y} \neq \texttt{f (x + y)}$$

- 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
```

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.

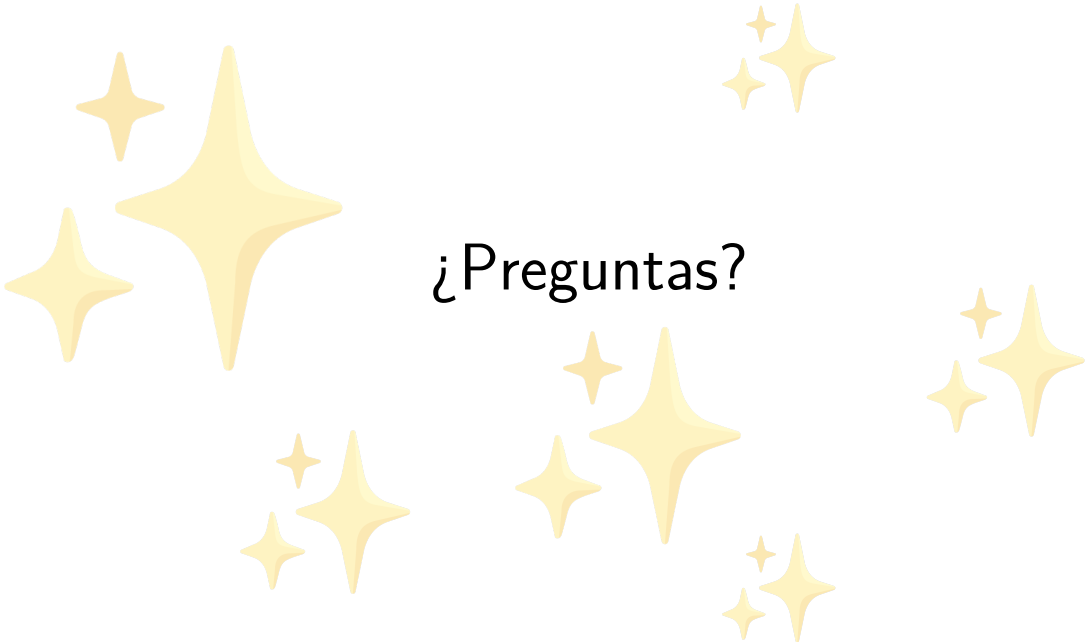
Tipos de datos algebraicos

```
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`.


¿Preguntas?