

## **Tema 5. Polimorfismo**

### 5.1 Funciones Polimórficas

La función identidad

Polimorfismo en Tuplas

Polimorfismo en Listas

Composición de funciones

El operador (\$)

## 5.1 Funciones Polimórficas

---

- ✓ Tienen sentido independientemente del tipo
- ✓ Ventaja: código más reutilizable y fácil de mantener

### La función identidad

---

Ejemplo simple: La función predefinida *identidad*

$$\begin{aligned} id &:: a \rightarrow a \\ id\ x &= x \end{aligned}$$

Uso:

```
? id 'd'  
'd' :: Char
```

```
? id 120  
120 :: Integer
```

```
? id [1, 2, 0]  
[1, 2, 0] :: [Integer]
```

- ✓ La  $a$  en el tipo es una *variable de tipo* (en minúscula): denota un tipo arbitrario
- ✓ El tipo del argumento,  $a$ , indica que  $id$  puede tomar argumentos de cualquier tipo
- ✓ El tipo del resultado,  $a$ , indica que  $id$  devuelve un valor cuyo tipo coincide con el del argumento

## Polimorfismo en Tuplas

---

Los selectores predefinidos *fst* y *snd* permiten extraer componentes:

$$\begin{aligned}fst & \quad :: \quad (a, b) \rightarrow a \\fst (x, \_ ) & = x \\snd & \quad :: \quad (a, b) \rightarrow b \\snd (\_ , y) & = y\end{aligned}$$

Uso:

? *fst* (1 , 'd')

1 :: *Integer*

? *snd* (1 , 'd')

'd' :: *Char*

? *snd* (1 , 2)

2 :: *Integer*

- ✓ Se usan dos variables de tipo distintas: *a* y *b*
- ✓ Esto indica que los tipos de ambas componentes pueden ser distintos
- ✓ El resultado de *fst* tiene siempre el mismo tipo que la primera componente del argumento
- ✓ Dos variables de tipo distintas pueden corresponder a dos tipos distintos, aunque no es obligatorio (p. ej. *snd*(1, 2))

## Polimorfismo en Listas

---

La función predefinida *length* calcula la longitud de listas de cualquier tipo:

```
length           :: [a] → Int  
length []       = 0  
length ( _ : xs) = 1 + length xs
```

Uso:

```
? length [10, 11, 12]  
3 :: Int
```

```
? length [True, False]  
2 :: Int
```

```
? length [ [10, 11, 12], [13, 14, 15, 16] ]  
2 :: Int
```

Los selectores predefinidos pueden ser utilizados con listas de cualquier tipo:

```
head           :: [a] → a    -- Cabeza de la lista  
head (x : _ ) = x  
tail          :: [a] → [a]   -- Cola de la lista  
tail ( _ : xs) = xs  
last          :: [a] → a    -- Último elemento de la lista  
last [x]      = x  
last ( _ : xs) = last xs  
init         :: [a] → [a]   -- Toda menos el último elemento  
init [x]      = []  
init (x : xs) = x : init xs
```

## Polimorfismo en Listas (2)

---

Concatenación de listas:

```
infixr 5 ++
(++)      :: [a] -> [a] -> [a]
[]        ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

La función *map*:

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

La función *filter*:

```
filter  :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x : xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs
```

Usos:

```
? [1, 3, 5] ++ [2, 4]
[1, 3, 5, 2, 4] :: [Integer]
```

```
? map (+1) [1, 2, 3]
[2, 3, 4] :: [Integer]
```

```
? map even [1, 2, 3, 4]
[False, True, False, True] :: [Bool]
```

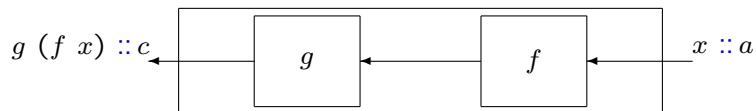
```
? filter even [1, 2, 3, 4]
[2, 4] :: [Integer]
```

## Composición de funciones

Si  $f :: a \rightarrow b$  y  $g :: b \rightarrow c$



se define  $g . f$ :



- ✓ El resultado de la composición es otra función con tipo  $g.f :: a \rightarrow c$
- ✓ Si el tipo del resultado de  $f$  no coincide con el del argumento de  $g$  las funciones no se pueden componer.
- ✓ En Haskell,  $(.)$  está predefinido como

```
infixr 9 .  
(.) :: (b -> c) -> (a -> b) -> (a -> c)  
g . f = \x -> g (f x)
```

Ejemplos:

```
esPar :: Integer -> Bool  
esPar x = (x `mod` 2 == 0)
```

```
esImpar :: Integer -> Bool -- Recordemos que not :: Bool -> Bool  
esImpar = not . esPar
```

```
fun :: Integer -> Integer  
fun = (+1) . (*2) . (+2)
```

```
? esImpar 5  
True :: Bool
```

```
? fun 10  
25 :: Integer
```

## El operador (\$)

---

Operador polimórfico predefinido, que permite aplicar una función a su argumento:

```
infixr 0 $
($) :: (a -> b) -> a -> b
f $ x = f x
```

Uso:

```
? f 5 where f x = 2 * x
10 :: Integer
```

```
? f $ 5 where f x = 2 * x
10 :: Integer
```

Su baja prioridad (mínima) lo hace útil para evitar paréntesis:

```
? f 5 + 3 where f x = 2 * x
13 :: Integer
```

```
? f (5 + 3) where f x = 2 * x
16 :: Integer
```

```
? f $ 5 + 3 where f x = 2 * x
16 :: Integer
```

```
? (+1) . (*2) . (+2) $ 10
25 :: Integer
```

# Objetivos del tema

---

El alumno debe:

- ✓ Comprender las definiciones de funciones y tipos polimórficos
- ✓ Saber definir y utilizar funciones polimórficas
- ✓ Conocer algunos de los operadores y funciones polimórficas predefinidas
- ✓ Saber utilizar el operador de composición de funciones para definir nuevas funciones a partir de otras.
- ✓ Saber el tipo de las funciones que se obtienen por composición de otras.