



Introducción a la Programación Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2024

Listas. Recursión sobre listas

Polimorfismo

Repasando...

- ▶ Se llama polimorfismo a una función que puede aplicarse a distintos tipos de datos (sin redefinirla).
- ▶ se usa cuando el comportamiento de la función no depende paramétricamente del tipo de sus argumentos
- ▶ lo vimos en el lenguaje de especificación con las funciones genéricas.
- ▶ En Haskell los polimorfismos se escriben usando **variables de tipo** y conviven con el tipado fuerte.
- ▶ Ejemplo de una función polimórfica: la función identidad.

Variables de tipos

¿Qué tipo tienen las siguientes funciones?

```
identidad x = x
```

```
primero x y = x
```

```
segundo x y = y
```

```
constante5 x y z = 5
```

Variables de tipo

- ▶ Son parámetros que se escriben en la signatura usando variables minúsculas
- ▶ En lugar de valores, denotan tipos
- ▶ Cuando se invoca la función se usa como argumento el tipo del valor

Variables de tipo (cont.)

Funciones con variables de tipo

```
identidad :: t -> t
identidad x = x

primero :: tx -> ty -> tx
primero x y = x

segundo :: tx -> ty -> ty
segundo x y = y

constante5 :: tx -> ty -> tz -> Int
constante5 x y z = 5

mismoTipo :: t -> t -> Bool
mismoTipo x y = True
```

Si dos argumentos deben tener el mismo tipo, se debe usar la misma variable de tipo

- Luego, primero `True 5 :: Bool`, pero `mismoTipo 1 True 0` no tipa

Especificación de un problema: Extensión

Variables de tipo

- ▶ Vamos a querer describir funciones polimórficas con nuestro lenguaje de especificación
- ▶ Veamos cómo podemos hacerlo...

Especificación de un problema: Extensión

Variables de tipo

```
problema nombre(parámetros) : tipo de dato del resultado {  
  requiere etiqueta: { condiciones sobre los parámetros de entrada }  
  asegura etiqueta: { condiciones sobre los parámetros de salida }  
}
```

- ▶ *nombre*: nombre que le damos al problema
 - ▶ será resuelto por una función con ese mismo nombre
- ▶ *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
 - ▶ Nombre del parámetro
 - ▶ Tipo de datos del parámetro o una **variable de tipo**
- ▶ *tipo de dato del resultado*: tipo de dato del resultado del problema (inicialmente especificaremos funciones) o una **variable de tipo**
 - ▶ En los asegura, podremos referenciar el valor devuelto con el nombre de res
- ▶ *etiquetas*: son nombres **opcionales** que nos servirán para nombrar declarativamente a las condiciones de los requiere o asegura.

Especificación de un problema: Extensión

Variables de tipo

- ▶ El símbolo o nombre (letra) de la variable de tipo no se corresponde con ninguno de los tipos de datos conocidos. Es una representación genérica.
- ▶ Cada ocurrencia de una variable de tipo, **siempre** representa al mismo tipo de datos.

```
problema segundo( $x : U, y : T$ ) :  $T$  {  
  asegura devuelveElSegundo:  $\{res = y\}$   
}
```

```
problema cantidadDeApariciones( $s : seq\langle T \rangle, e : T$ ) :  $\mathbb{Z}$  {  
  asegura:  $\{res = \sum_{i=0}^{|s|-1} (if\ s[i] = e\ then\ 1\ else\ 0\ fi)\}$   
}
```

Especificación de un problema: Extensión

Variables de tipo con restricciones

- Se puede restringir los posibles tipos de una variable de tipo mediante un *requiere*

```
problema suma( $x : T, y : T$ ) :  $T$ {  
  requiere:  $\{T \in [\mathbb{N}, \mathbb{Z}, \mathbb{R}]\}$   
  asegura:  $\{res = x + y\}$   
}
```


Pensemos en listas: Motivación

Algunas operaciones

► `maximo :: Int -> Int -> Int`

Pensemos en listas: Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`

Pensemos en listas: Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`
- ▶ `maximo4 :: Int -> Int -> Int -> Int -> Int`

Pensemos en listas: Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`
- ▶ `maximo4 :: Int -> Int -> Int -> Int -> Int`
- ▶ `⋮`
- ▶ `maximoN :: Int -> Int -> ... -> Int`

Pensemos en listas: Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`
- ▶ `maximo4 :: Int -> Int -> Int -> Int -> Int`
- ▶ `⋮`
- ▶ `maximoN :: Int -> Int -> ... -> Int`

Pregunta

¿Hay alguna manera de definir funciones que nos permitan trabajar con cantidades arbitrarias de elementos?

Pensemos en listas: Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`
- ▶ `maximo4 :: Int -> Int -> Int -> Int -> Int`
- ▶ `⋮`
- ▶ `maximoN :: Int -> Int -> ... -> Int`

Pregunta

¿Hay alguna manera de definir funciones que nos permitan trabajar con cantidades arbitrarias de elementos?

Más concretamente, ¿podemos definir una función máximo que funcione por igual para 2, 10 o una cantidad N de elementos?

Pensemos en listas: Motivación

Algunas operaciones

- ▶ `maximo :: Int -> Int -> Int`
- ▶ `maximo3 :: Int -> Int -> Int -> Int`
- ▶ `maximo4 :: Int -> Int -> Int -> Int -> Int`
- ▶ `⋮`
- ▶ `maximoN :: Int -> Int -> ... -> Int`

Pregunta

¿Hay alguna manera de definir funciones que nos permitan trabajar con cantidades arbitrarias de elementos?

Más concretamente, ¿podemos definir una función máximo que funcione por igual para 2, 10 o una cantidad N de elementos?

Respuesta: ¡Sí!, usando **listas**.

Un nuevo tipo: Listas

Expresiones

► [1, 2, 1]

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: []`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: [[Int]]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: [[Int]]`
- ▶ `[1, True]`

Un nuevo tipo: Listas

Expresiones

- ▶ `[1, 2, 1]`
- ▶ `[True, False, False, True]`
- ▶ `[]` (símbolo distinguido para denotar una lista vacía, es decir, una lista sin elementos)

Las listas en Haskell son listas o secuencias de elementos de un mismo tipo, cuyos elementos se pueden repetir.

El tipo de una lista se escribe como: `[tipo]`

- ▶ `[True, False, False] :: [Bool]`
- ▶ `[1, 2, 3, 4] :: [Int]`
- ▶ `[div 10 5, div 2 2] :: [Int]`
- ▶ `[[1], [2,3], [], [1,1000,2,0]] :: [[Int]]`
- ▶ `[1, True]`
- ▶ `[(1,2), (3,4), (5,2)]` ¿Cuál es el tipo de esta lista?

Operaciones

Algunas operaciones que nos brinda el Preludio de Haskell

- ▶ `head :: [a] -> a`
- ▶ `tail :: [a] -> [a]`
- ▶ `(:) :: a -> [a] -> [a]`

Operaciones

Algunas operaciones que nos brinda el Prelude de Haskell

- ▶ `head :: [a] -> a`
- ▶ `tail :: [a] -> [a]`
- ▶ `(:) :: a -> [a] -> [a]`

Tipar y evaluar las siguientes expresiones

- ▶ `head [(1,2), (3,4), (5,2)]`
- ▶ `tail [1,2,3,4,4,3,2,1]`
- ▶ `[1,2] : []`
- ▶ `head []`
- ▶ `head [1,2,3] : [4,5]`
- ▶ `head ([1,2,3] : [4,5])`
- ▶ `head ([1,2,3] : [4,5] : [])`

Creando listas

Formas rápidas para crear listas

Prueben las siguientes expresiones en GHCi

- ▶ `[1..100]`
- ▶ `[1,3..100]`
- ▶ `[100..1]`
- ▶ `[1..]`

Ejercicio

- ▶ Escribir una expresión que denote la lista estrictamente decreciente de enteros que comienza con el número 1 y termina con el número -100.
- ▶ Escribir una expresión que denote la lista estrictamente creciente de enteros entre -20 y 20 que son congruentes a 1 módulo 4.

Recursión sobre listas

¿Se puede pensar recursivamente en listas? ¿Cómo?

Implementar las siguientes funciones (en el pizarrón)

1. `longitud :: [Int] -> Int`
que indica cuántos elementos tiene una lista.
2. `sumatoria :: [Int] -> Int`
que indica la suma de los elementos de una lista.
3. `pertenece :: Int -> [Int] -> Bool`
que indica si un elemento aparece en la lista. Por ejemplo:
`pertenece 9 [] ~> False`
`pertenece 9 [1,2,3] ~> False`
`pertenece 9 [1,2,9,9,-1,0] ~> True`

Idea: Pensar cómo combinar el resultado de la función sobre la cola de la lista con el primer elemento. Recordar:

- ▶ `head [1, 2, 3] ~> 1`
- ▶ `tail [1, 2, 3] ~> [2, 3]`

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (Bool, Int, tuplas). ¿Se puede hacer *pattern matching* en listas?

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (Bool, Int, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ [] (lista vacía)
- ▶ algo : lista (lista no vacía)

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Int`, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] -> Int` usando *pattern matching*

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Int`, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] -> Int` usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Int`, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] -> Int` usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

Escribir la función `sumatoria :: [Int] -> Int` usando *pattern matching*

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Int`, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] -> Int` usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

Escribir la función `sumatoria :: [Int] -> Int` usando *pattern matching*

```
sumatoria [] = 0
sumatoria (x:xs) = sumatoria xs + x
```

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Int`, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] -> Int` usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

Escribir la función `sumatoria :: [Int] -> Int` usando *pattern matching*

```
sumatoria [] = 0
sumatoria (x:xs) = sumatoria xs + x
```

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Int`, tuplas). ¿Se puede hacer *pattern matching* en listas?

¿Cuál es la verdadera forma de las listas?

Las listas tienen dos “pintas”:

- ▶ `[]` (lista vacía)
- ▶ `algo : lista` (lista no vacía)

Escribir la función `longitud :: [Int] -> Int` usando *pattern matching*

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs
```

Escribir la función `sumatoria :: [Int] -> Int` usando *pattern matching*

```
sumatoria [] = 0
sumatoria (x:xs) = sumatoria xs + x
```

Ejercicio: volver a implementar la función `pertenece` utilizando *pattern matching*.

Guía 5: Ejercicio 4

- ▶ `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.

Guía 5: Ejercicio 4

- ▶ `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.
- ▶ `contarPalabras :: [Char] -> Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.

Guía 5: Ejercicio 4

- ▶ `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.
- ▶ `contarPalabras :: [Char] -> Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.
- ▶ `palabras :: [Char] -> [[Char]]`, que dada una lista arma una nueva lista con las palabras de la lista original.

Guía 5: Ejercicio 4

- ▶ `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.
- ▶ `contarPalabras :: [Char] -> Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.
- ▶ `palabras :: [Char] -> [[Char]]`, que dada una lista arma una nueva lista con las palabras de la lista original.
- ▶ `palabraMasLarga :: [Char] -> [Char]`, que dada una lista de caracteres devuelve su palabra más larga.

Guía 5: Ejercicio 4

- ▶ `sacarBlancosRepetidos :: [Char] -> [Char]`, que reemplaza cada subsecuencia de blancos contiguos de la primera lista por un solo blanco en la segunda lista.
- ▶ `contarPalabras :: [Char] -> Integer`, que dada una lista de caracteres devuelve la cantidad de palabras que tiene.
- ▶ `palabras :: [Char] -> [[Char]]`, que dada una lista arma una nueva lista con las palabras de la lista original.
- ▶ `palabraMasLarga :: [Char] -> [Char]`, que dada una lista de caracteres devuelve su palabra más larga.
- ▶ `aplanar :: [[Char]] -> [Char]`, que a partir de una lista de palabras arma una lista de caracteres concatenándolas.