Taller de Álgebra I

Clase 6 - Listas

Segundo cuatrimestre 2022

Motivación

Algunas operaciones

- `maximo :: Int -> Int -> Int`

Motivación

Algunas operaciones

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

Motivación

Algunas operaciones

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

Motivación

Algunas operaciones

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

  ⋮
- `maximoN :: Int -> Int -> ··· -> Int`

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?

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?

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 son "listas" 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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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 son "listas" de elementos de un mismo tipo, cuyos elementos se pueden repetir.

Tipo Listas

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

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

Operaciones

Algunas operaciones

- `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?

Pensar las siguientes funciones

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

Recursión sobre listas

¿Se puede pensar recursivamente en listas? ¿Cómo?

Pensar las siguientes funciones

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

¿Cómo 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)

¿Cómo 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)

¿Cómo escribir la función `sumatoria :: [Int] -> Int` usando *pattern matching*?

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

Otro ejemplo:

```
longitud :: [a] -> Int
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)

¿Cómo escribir la función `sumatoria :: [Int] -> Int` usando *pattern matching*?

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

Otro ejemplo:

```
longitud :: [a] -> Int
longitud [] = 0
longitud (_:xs)  = 1 + longitud xs
```

Ejercicio: pertenece

Repensar la función `pertenece` utilizando pattern matching.

Ejercicios

Resolver primero sin y después con pattern matching sobre listas

- `productoria :: [Int] -> Int` que devuelve la productoria de los elementos de una lista.
- `sumarN :: Int -> [Int] -> [Int]` que dado un número *N* y una lista *xs*, suma N a cada elemento de *xs*.
- `sumarElPrimero :: [Int] -> [Int]` que dada una lista no vacía *xs*, suma el primer elemento a cada elemento de *xs*. Ejemplo: `sumarElPrimero [1,2,3]` $\rightsquigarrow$ `[2,3,4]`
- `sumarElUltimo :: [Int] -> [Int]` que dada una lista no vacía *xs*, suma el último elemento a cada elemento de *xs*. Ejemplo: `sumarElUltimo [1,2,3]` $\rightsquigarrow$ `[4,5,6]`
- `pares :: [Int] -> [Int]` que devuelve una lista con los elementos pares de una lista dada. Ejemplo: `pares [1,2,3,5,8]` $\rightsquigarrow$ `[2,8]`
- `multiplosDeN :: Int -> [Int] -> [Int]` que dado un número *N* y una lista *xs*, devuelve una lista con los elementos múltiplos N de *xs*.
- `reverso :: [Int] -> [Int]` que dada una lista invierte su orden.
- `maximo :: [Int] -> Int` que calcula el máximo elemento de una lista no vacía.
- `ordenar :: [Int] -> [Int]` que ordena los elementos de una lista de forma creciente.
- `quitar :: Int -> [Int] -> [Int]` que elimina la primera aparición del elemento en la lista (de haberla).
- `hayRepetidos :: [Int] -> Bool` que indica si una lista tiene elementos repetidos.
- `eliminarRepetidos :: [Int] -> [Int]` que deja en la lista una única aparición de cada elemento, eliminando las repeticiones adicionales.