Listas

Taller de Álgebra I

Segundo cuatrimestre 2018

Un nuevo tipo: Listas

Tipo Lista

Las listas son "listas" de elementos de un mismo tipo. Los elementos se pueden repetir. Por ejemplo:

- `[1, 2, 1]`
- `[True, False, False, True]`
- `[]` (lista vacía)

Un nuevo tipo: Listas

Tipo Lista

Las listas son "listas" de elementos de un mismo tipo. Los elementos se pueden repetir. Por ejemplo:

- `[1, 2, 1]`
- `[True, False, False, True]`
- `[]` (lista vacía)

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

- `[True, False, False] :: [Bool]`
- `[1, 2, 3, 4] :: [Integer]`
- `[div 1 1, div 2 1] :: [Integer]`
- `[1.0, 2] :: [Float]`
- `[[1], [2,3], [], [1,1000,2,0]] :: [[Integer]]`

Un nuevo tipo: Listas

Tipo Lista

Las listas son "listas" de elementos de un mismo tipo. Los elementos se pueden repetir. Por ejemplo:

- `[1, 2, 1]`
- `[True, False, False, True]`
- `[]` (lista vacía)

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

- `[True, False, False] :: [Bool]`
- `[1, 2, 3, 4] :: [Integer]`
- `[div 1 1, div 2 1] :: [Integer]`
- `[1.0, 2] :: [Float]`
- `[[1], [2,3], [], [1,1000,2,0]] :: [[Integer]]`
- `[1, True]` NO ES UNA LISTA VÁLIDA, ¿por qué?
- `[1.0, div 1 1]` NO ES UNA LISTA VÁLIDA, ¿por qué?

Un nuevo tipo: Listas

Tipo Lista

Las listas son "listas" de elementos de un mismo tipo. Los elementos se pueden repetir. Por ejemplo:

- `[1, 2, 1]`
- `[True, False, False, True]`
- `[]` (lista vacía)

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

- `[True, False, False] :: [Bool]`
- `[1, 2, 3, 4] :: [Integer]`
- `[div 1 1, div 2 1] :: [Integer]`
- `[1.0, 2] :: [Float]`
- `[[1], [2,3], [], [1,1000,2,0]] :: [[Integer]]`
- `[1, True]` NO ES UNA LISTA VÁLIDA, ¿por qué?
- `[1.0, div 1 1]` NO ES UNA LISTA VÁLIDA, ¿por qué?
- `[(1,2), (3,4), (5,2)]` ¿Cuál es el tipo de esta lista?
- `[]` ¿Cuál es el tipo de esta lista?

Operaciones

Algunas operaciones

- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `(:) :: a -> [a] -> [a]`
- `(++) :: [a] -> [a] -> [a]`
- `length :: [a] -> Integer`

Operaciones

Algunas operaciones

- `head :: [a] -> a`
- `tail :: [a] -> [a]`
- `(:) :: a -> [a] -> [a]`
- `(++) :: [a] -> [a] -> [a]`
- `length :: [a] -> Integer`

Tipar y evaluar las siguientes expresiones

- `head [(1,2), (3,4), (5,2)]`
- `tail [1,2,3,4,4,3,2,1]`
- `head []`
- `head [1,2,3] : [2,3]`
- `[True, True] ++ [False, False]`
- `[1,2] : []`

Formas rápidas para crear listas

Prueben las siguientes expresiones en GHCI

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

Listas

Formas rápidas para crear listas

Prueben las siguientes expresiones en GHCI

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

Ejercicios

- Definir la función `listar :: a -> a -> a -> [a]` que toma 3 elementos y los convierte en una lista.
- Escribir una expresión que denote la lista estrictamente decreciente de enteros que comienza con el número 1 y termina con el número -100.

Recursión sobre listas

¿Se puede pensar recursivamente en listas? ¿Cómo?

Pensar las siguientes funciones

- `sumatoria :: [Integer] -> Integer`
  que indica la suma de los elementos de una lista.
- `pertenece :: Integer -> [Integer] -> 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

- `sumatoria :: [Integer] -> Integer`
  que indica la suma de los elementos de una lista.
- `pertenece :: Integer -> [Integer] -> 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`

¿Me sirve de algo para definir la función, el resultado sobre la cola de la lista?

- ¿Me sirve para sumar la lista `[a, b, c]` saber cuánto es la suma de la lista `[b, c]` (la cola de la lista) y saber que el primer elemento es `a`?
- ¿Me sirve para saber si `pertenece x [a, b, c, d]` saber si `pertenece x [b, c, d]` y saber que el primer elemento es `a`?

Recursión sobre listas

¿Se puede pensar recursivamente en listas? ¿Cómo?

Pensar las siguientes funciones

- `sumatoria :: [Integer] -> Integer`
  que indica la suma de los elementos de una lista.
- `pertenece :: Integer -> [Integer] -> 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`

¿Me sirve de algo para definir la función, el resultado sobre la cola de la lista?

- ¿Me sirve para sumar la lista `[a, b, c]` saber cuánto es la suma de la lista `[b, c]` (la cola de la lista) y saber que el primer elemento es `a`?
- ¿Me sirve para saber si `pertenece x [a, b, c, d]` saber si `pertenece x [b, c, d]` y saber que el primer elemento es `a`?

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`, `Integer`, tuplas).
¿Se puede hacer *pattern matching* en listas?

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Integer`, 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 :: [Integer] -> Integer` usando *pattern matching*?

Pattern matching en listas

Ya vimos cómo hacer *pattern matching* sobre distintos tipos (`Bool`, `Integer`, 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 :: [Integer] -> Integer` 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`, `Integer`, 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 :: [Integer] -> Integer` usando *pattern matching*?

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

Las listas también admiten el patrón _, que se corresponde con cualquier valor, pero no liga ninguna variable. Por ejemplo:

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