Programación Funcional

Curso 2019-20

FUNCIONES DE LISTAS EN HASKELL

Listas: Funciones de `Prelude`

- Cabeza y resto de una lista

  `head::[a] -> a` , `tail::[a] -> [a]`

  No muy usadas, la verdad...
- Último elemento de una lista no vacía

  `last::[a] -> a`

```
> last [1,3,5,7]
7
```

```
> last []
Exception
```

- Todos menos el último elemento de una lista no vacía

  `init::[a] -> [a]`

```
> init [1,3,5,7]
[1,3,5]
```

```
> init []
Exception
```

- Test de vacuidad de una lista

  `null::[a] -> Bool`

```
> null [1,3]
False
```

```
> null []
True
```

Listas: más funciones de `Prelude`

- Longitud de una lista

```
length::[a] -> Int
```

```
> length [1,3,5,7]
4
```

```
> length []
0
```

- Concatenación de dos listas

```
(++)::[a] -> [a] -> [a]
```

```
> [1,3] ++ [2,4,6]
[1,3,2,4,6]
```

```
> [1] ++ [3] ++ [] ++[1]
[1,3,1]
```

- Inversión de una lista

```
reverse::[a] -> [a]
```

```
> reverse [1,3,5,7]
[7,5,3,1]
```

```
> reverse []
[]
```

Listas: más funciones de `Prelude`

- Concatenación de los elementos de una lista de listas

```
concat::[[a]] -> [a]
```

```
> concat [[1,3,5],[2,4],[],[6]]
[1,3,5,2,4,6]
```

```
> concat [[]]
[]
```

```
> concat []
[]
```

- Test de pertenencia a una lista

```
elem::Eq a => a -> [a] -> Bool
```

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

```
> elem 3 [1,2]
False
```

```
> elem 3 []
False
```

```
> elem 3 [1 `div` 0]
Exception
```

Listas: más funciones de `Prelude`

- Selección del n-simo elemento (contando desde 0)

```
(!!)::[a] -> Int -> a
```

```
> [1,3,5,7] !!  2
5
```

```
> [1,3] !!  2
Exception
```

- Selección de los primeros n elementos

```
take::Int -> [a] -> [a]
```

```
> take 2 [1,3,5,7]
[1,3]
```

```
> take 0 [1,3,5,7]
[]
```

```
> take 8 [1,3,5,7]
[1,3,5,7]
```

```
> take 2 []
[]
```

Listas: más funciones de `Prelude`

- Eliminación de los primeros *n* elementos

```
drop::Int -> [a] -> [a]
```

```
> drop 2 [1,3,5,7]
[5,7]
```

```
> drop 0 [1,3,5,7]
[1,3,5,7]
```

```
> drop 4 [1,3,5,7]
[]
```

```
> drop 2 []
[]
```

- Separar los primeros *n* elementos de los demás

```
splitAt::Int -> [a] -> ([a],[a])
```

```
> splitAt 2 [1,3,5,7]
([1,3],[5,7])
```

```
> splitAt 0 [1,3,5,7]
([],[1,3,5,7])
```

Listas: más funciones de `Prelude`

- Sumar (multiplicar) los elementos de una lista de números

```
sum , product::Num a => [a] -> a
```

```
> sum [1,3,5]
9
```

```
> product [1,3,5]
15
```

```
> sum []
0
```

```
> product []
1
```

- Hacer conjunción (disyunción) de los elementos de una lista de booleanos

```
and , or::[Bool] -> Bool
```

```
> and [True,False]
False
```

```
> or [True,False]
True
```

```
> and []
True
```

```
> or []
False
```

Listas: más funciones de `Prelude`

- Emparejar dos listas elemento a elemento

```
zip::[a] -> [b] -> [(a,b)]
```

```
> zip [1,2] ['a','b','c']
[(1,'a'),(2,'b')]
```

```
> zip [1,2] []
[]
```

- Desemparejar una lista de parejas

```
unzip::[(a,b)] -> ([a],[b])
```

```
> unzip [(1,'a'),(2,'b')]
([1,2],['a','b'])
```

```
> unzip []
([],[])
```

Notación `[1,2,3]` es azúcar sintáctico

```
> 1:2:3:[]
[1,2,3]
> 1:2:3:[] == [1,2,3]
True
```

Notación alternativa para listas

`:` es una constructora de datos

Comprobamos que son lo mismo!

`:` es una constructora de aridad 2

`[]` es una constructora de aridad 0

Toda lista es o bien `[]`

o bien de la forma `x:xs`

`x` es la *cabeza* y `xs` el *resto*

`xs` ha de ser otra lista

```
> 1:2:3:[] == 1:(2:(3:[]))
True
> 1:2:3:[] == ((1:2):3):[]
Error de tipo
> 1:2:3:4:[] == 1:2:[3,4]
True
```

Podemos prescindir de los paréntesis porque `:` asocia por la derecha

Se pueden mezclar notaciones

Listas: más funciones de `Prelude`

- Secuencias finitas

```
> [1..4]
[1,2,3,4]
```

```
> [1,3..7]
[1,3,5,7]
```

```
> [1,3..8]
[1,3,5,7]
```

```
> [4..0]
[]
```

```
> [4,3..0]
[4,3,2,1,0]
```

- Secuencias infinitas

```
> [1..]
[1,2,3,......]
```

```
> [1,3..
[1,3,5,......
```

- La evaluación perezosa permite procesar partes finitas

```
> take 2 [1..]
[1,2]
```

```
> [1..]==[0..]
False
```

```
> ([2,7,5]++[0..])!!5
2
```

```
> ([0..]++[2,7,5])!!5
5
```

- Pero no siempre, claro

```
> length [1..]
No termina
```

```
> [1..]==[1..]
No termina
```

Notación [1..4] es azúcar sintáctico

- `[1..4]` $\equiv$ `enumFromTo 1 4`
- `[1,3..7]` $\equiv$ `enumFromThenTo 1 3 7`
- `[1..]` $\equiv$ `enumFrom 1`
- `[1,3..]` $\equiv$ `enumFromThen 1 3`