Recursión sobre listas

Taller de Álgebra I

Segundo cuatrimestre de 2016

Recordando recursión

La clase pasada vimos:

- **Reducción**. El modo en que Haskell evalúa las expresiones.
- **Recursión**. El modo de pensar funciones recursivas.

Para pensar funciones recursivas:

1. Casos bases: identificar el o los casos bases.
2. Casos recursivos: **suponiendo que la llamada recursiva es correcta**, ¿qué tengo que hacer para completar la solución?

```
sumaLosPrimerosNImpares :: Integer -> Integer
sumaLosPrimerosNImpares n
      | n == 1 = 1
      | n > 0 = (?) sumaLosPrimerosNImpares (n-1)
```

- Fíjense que el último llamado recursivo (n=2) es efectivamente correcto, es el caso base.
- Si podemos dar una solución correcta en base a una llamada recursiva correcta entonces, por inducción, ¡todos van a ser correctos! En particular el segundo. Y por lo tanto el tercero. Y por...

Con el paso anterior resuelto: ¿Qué falta para que el nuevo paso esté resuelto?

```
      | n > 0 = (n_esimoImpar) + sumaLosPrimerosNImpares (n-1)
```

Cambiamos el problema: ahora solo falta resolver `n_esimoImpar`.

Resolución de ejercicio: Funciones auxiliares

Ejercicio de la clase pasada

- `sumaImparesCuyoCuadSeaMenorQue :: Integer -> Integer`
  Suma los números impares positivos cuyo cuadrado sea menor que n.
  `sumaImparesCuyoCuadSeaMenorQue 30` $\rightsquigarrow$ `1 + 3 + 5` $\rightsquigarrow$ `9`.

Piensen una función auxiliar que

- les permita hacer la recursión sobre un parámetro.
- les permita recordar cuál es el umbral en otro parámetro.

```
sumaImparesCuyoCuadSeaMenorQue :: Integer -> Integer
sumaImparesCuyoCuadSeaMenorQue umbral = sumaAuxiliar ... ...
```

```
-- Construimos una funcion que nos facilita el trabajo
sumaAuxiliar :: Integer -> Integer -> Integer
sumaAuxiliar umbral num
    | num ^ 2 >= umbral = 0
    | otherwise = num + sumaAuxiliar umbral (num + 2)
```

¿Qué hace la función `sumaAuxiliar`?

Operaciones del tipo Lista

Algunas operaciones

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

Ejemplos

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

Recursión sobre listas

Para calentar motores

```
sumatoria :: [Integer] -> Integer
```

```
sumatoria lista
    | length lista == 0 = 0
    | otherwise = head lista + sumatoria (tail lista)
```

Implementemos las siguientes funciones

- Implementar `productoria :: [Integer] -> Integer`
- Implementar `reverso :: [a] -> [a]`

Recursión sobre listas

Las funciones a resolver en esta diapositiva las vamos a usar la clase que viene. ¡Es importante haberlas resuelto todas antes de irse!

Implementar las siguientes funciones

- `pertenece :: Integer -> [Integer] -> Bool`
  que indica si un elemento aparece en la lista.
- `hayRepetidos :: [Integer] -> Bool`
  que indica si una lista tiene elementos repetidos.
- `quitar :: Integer -> [Integer] -> [Integer]`
  que elimina la primera aparición del elemento en la lista (de haberla).

Últimas

- `eliminarRepetidos :: [Integer] -> [Integer]`
  que deja en la lista una única aparición de cada elemento, eliminando las repeticiones adicionales.
- `maximo :: [Integer] -> Integer`
  que dada una lista no vacía calcula el mayor elemento de la misma.
- `ordenar :: [Integer] -> [Integer]`
  que dada una lista ordena sus elementos de forma creciente.

Ejercicios extra

1. Definir `suma :: [Integer] -> [Integer] -> [Integer]` que dadas dos listas del mismo tamaño, encuentra la suma elemento a elemento. Por ejemplo, `suma [1, 4, 6] [2, -1, 0]` $\rightsquigarrow$ `[3, 3, 6]`.
2. Definir `prodInterno :: [Float] -> [Float] -> Float` que calcula el producto interno entre dos listas del mismo tamaño, pensadas como vectores en $\mathbb{R}^n$. Por ejemplo: `prodInterno [1, -2, 3, 4] [1, 0, 3, 2]` $\rightsquigarrow$ `1*1 + (-2)*0 + 3*3 + 4*2` $\rightsquigarrow$ `18`.
3. Implementar `noTieneDivisoresHasta :: Integer -> Integer -> Bool`

   `noTieneDivisoresHasta m n` da `True` sii ningún número entre 2 y `m` divide a `n`.
4. Utilizando `noTieneDivisoresHasta`, programar `esPrimo :: Integer -> Bool`.