Recursión 2 + Recursión sobre listas

Taller de Álgebra I

Primer cuatrimestre de 2016

Recordando Recursión

Ejercicios

- `potencia :: Float -> Integer -> Float`
  Que calcule la potencia: a^n con $(n \geq 0)$.
  Debe utilizarse el producto y no $**$ ni similares.

Recordando Recursión

Ejercicios

- `potencia :: Float -> Integer -> Float`
  Que calcule la potencia: a^n con $(n \geq 0)$.
  Debe utilizarse el producto y no $**$ ni similares.

- (Ejercicio 4 clase pasada) `sumaImparesCuyoCuadSeaMenorQue :: Integer -> Integer`
  Suma los números impares positivos cuyo cuadrado sea menor que n.
  `sumaImparesCuyoCuadSeaMenorQue 30` $\rightsquigarrow$ `1 + 3 + 5` $\rightsquigarrow$ `9`.

Recordando Recursión

Ejercicios

- `potencia :: Float -> Integer -> Float`
  Que calcule la potencia: a^n con $(n \geq 0)$.
  Debe utilizarse el producto y no $**$ ni similares.

- (Ejercicio 4 clase pasada) `sumaImparesCuyoCuadSeaMenorQue :: Integer -> Integer`
  Suma los números impares positivos cuyo cuadrado sea menor que n.
  `sumaImparesCuyoCuadSeaMenorQue 30` $\rightsquigarrow$ `1 + 3 + 5` $\rightsquigarrow$ `9`.

```
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)
```

Algoritmo de división

Teorema: Dados a, $d \in \mathbb{Z}$, $d \neq 0$, existen únicos $q, r \in \mathbb{Z}$ tales que

- $a = qd + r$,
- $0 \leq r < |d|$.

Algoritmo de división

Teorema: Dados $a, d \in \mathbb{Z}$, $d \neq 0$, existen únicos $q, r \in \mathbb{Z}$ tales que

- $a = qd + r$,
- $0 \leq r < |d|$.

Idea de la demostración: (caso $a \geq 0$, $d > 0$). Por inducción en a.

- Si $0 \leq a < d$, tomamos $q = 0$, $r = a$.

Algoritmo de división

Teorema: Dados $a, d \in \mathbb{Z}$, $d \neq 0$, existen únicos $q, r \in \mathbb{Z}$ tales que

- $a = qd + r$,
- $0 \leq r < |d|$.

Idea de la demostración: (caso $a \geq 0$, $d > 0$). Por inducción en a.

- Si $0 \leq a < d$, tomamos $q = 0$, $r = a$.
- Si no, dividimos $a - d$ por d. Eso da un cociente q' y un resto r'. Tomamos $r = r'$, $q = q' + 1$.

Algoritmo de división

Teorema: Dados $a, d \in \mathbb{Z}$, $d \neq 0$, existen únicos $q, r \in \mathbb{Z}$ tales que

- $a = qd + r$,
- $0 \leq r < |d|$.

Idea de la demostración: (caso $a \geq 0$, $d > 0$). Por inducción en a.

- Si $0 \leq a < d$, tomamos $q = 0$, $r = a$.
- Si no, dividimos $a - d$ por d. Eso da un cociente q' y un resto r'. Tomamos $r = r'$, $q = q' + 1$.

Implementar la siguiente función

```
division :: Integer -> Integer -> (Integer, Integer)
```

Debe funcionar para $a \geq 0$, $d > 0$ y no se pueden usar `div`, `mod` ni `/`.

Algoritmo de división

```
division :: Integer -> Integer -> (Integer, Integer
)
division a d | a < d = (0, a)
division a d | otherwise
                = (fst (division (a-d) d) + 1,
                   snd (division (a-d) d))
```

Algoritmo de división

```
                division :: Integer -> Integer -> (Integer, Integer
    )
                division a d | a < d = (0, a)
                division a d | otherwise
                                = (fst (division (a-d) d) + 1,
                                   snd (division (a-d) d))
```

¿Se puede no poner dos veces `division (a-d) d`? Sí:

Algoritmo de división

```
division :: Integer -> Integer -> (Integer, Integer
)
division a d | a < d = (0, a)
division a d | otherwise
                = (fst (division (a-d) d) + 1,
                   snd (division (a-d) d))
```

¿Se puede no poner dos veces `division (a-d) d`? Sí:

```
division :: Integer -> Integer -> (Integer, Integer
)
division a d | a < d = (0, a)
division a d | otherwise = (fst qr' + 1, snd qr')
             where qr' = division (a-d) d
```

Primos

Un entero $p > 1$ es primo si ningún natural k tal que $1 < k < p$ divide a p.

Primos

Un entero $p > 1$ es primo si ningún natural k tal que $1 < k < p$ divide a p. ¿Se puede hacer una función `esPrimo :: Integer -> Bool` que diga si un número entero mayor a cero es primo o no? Una posibilidad sería buscar todos los divisores. Algo como

$$divisores(n) = \{k \in \mathbb{Z} \mid 1 \leq k \leq n \text{ y } k|n\}$$

Primos

Un entero $p > 1$ es primo si ningún natural k tal que $1 < k < p$ divide a p. ¿Se puede hacer una función `esPrimo :: Integer -> Bool` que diga si un número entero mayor a cero es primo o no? Una posibilidad sería buscar todos los divisores. Algo como

$$divisores(n) = \{k \in \mathbb{Z} \mid 1 \leq k \leq n \text{ y } k|n\}$$

Esto se puede hacer en Haskell de forma directa, aunque todavía no vimos cómo. Podemos sin embargo definir una lista "parcial" de divisores:

$$divParcial(n, m) = \{k \in \mathbb{Z} \mid 1 \leq k \leq m \text{ y } k|n\}$$

Primos

Un entero $p > 1$ es primo si ningún natural k tal que $1 < k < p$ divide a p. ¿Se puede hacer una función `esPrimo :: Integer -> Bool` que diga si un número entero mayor a cero es primo o no? Una posibilidad sería buscar todos los divisores. Algo como

$$divisores(n) = \{k \in \mathbb{Z} \mid 1 \leq k \leq n \text{ y } k|n\}$$

Esto se puede hacer en Haskell de forma directa, aunque todavía no vimos cómo. Podemos sin embargo definir una lista "parcial" de divisores:

$$divParcial(n, m) = \{k \in \mathbb{Z} \mid 1 \leq k \leq m \text{ y } k|n\}$$

Implementar las siguientes funciones

- `divParcial :: Integer -> Integer -> [Integer]`
  Tiene que funcionar bien `divParcial n m` cuando $\texttt{m} \leq \texttt{n}$.
- Utilizando divParcial, programar
  `divisores :: Integer -> [Integer]`
- Utilizando divisores, programar
  `esPrimo :: Integer -> Bool`

Primos

```
divParcial :: Integer -> Integer -> [Integer]
divParcial n 1 = [1]
divParcial n m | mod n m == 0 = m : divParcial n (m
-1)
               | otherwise    = divParcial n (m-1)
```

Primos

```
divParcial :: Integer -> Integer -> [Integer]
divParcial n 1 = [1]
divParcial n m | mod n m == 0 = m : divParcial n (m
-1)
               | otherwise    = divParcial n (m-1)
```

```
divisores :: Integer -> [Integer]
divisores n = divParcial n n
```

Primos

```
divParcial :: Integer -> Integer -> [Integer]
divParcial n 1 = [1]
divParcial n m | mod n m == 0 = m : divParcial n (m
-1)
               | otherwise    = divParcial n (m-1)
```

```
divisores :: Integer -> [Integer]
divisores n = divParcial n n
```

```
esPrimo :: Integer -> Bool
esPrimo n = length (divisores n) == 2
```

No todo son números

```
suma :: [Integer] -> Integer
suma lista | length lista == 0 = 0
suma lista | otherwise = head lista + suma (tail lista)
```

Implementemos las siguientes funciones

- Implementar y dar el tipo de la productoria de una lista.
- Implementar la función `reverso :: [a] -> [a]`
- Mostrar los pasos de reducción para la evaluación de `reverso ['a', 'b', 'c']`
- Implementar la función `capicua :: [Integer] -> Bool` que devuelve verdadero si el reverso de una lista de números es la misma lista.

Recursión sobre Listas

La clase pasada vimos ejemplos de definiciones recursivas. Hoy vamos a continuar con el tema. Recordemos un ejemplo:

```
reverso :: [Integer] -> [Integer]
reverso [] = []
reverso xs = (reverso (tail xs)) ++ [head xs]
```

Recursión sobre Listas

La clase pasada vimos ejemplos de definiciones recursivas. Hoy vamos a continuar con el tema. Recordemos un ejemplo:

```
reverso :: [Integer] -> [Integer]
reverso [] = []
reverso xs = (reverso (tail xs)) ++ [head xs]
```

Truco, cómo pensar recursivamente con listas

- Dada una lista cualquiera: `[1,2,3,4,5]`
- Si ya tengo el resultado recursivo sobre la cola de la lista, es decir el reverso de `[2,3,4,5]`, ¡¿cómo puedo combinarlo con la cabeza de la lista (1) para obtener el resultado que quiero?

Ejercicios

1. Definir `suma :: [Integer] -> [Integer] -> [Integer]` que dadas dos listas del mismo tamaño encuentra la suma. Por ejemplo,
   `suma [1, 4, 6] [2, -1, 0]` $\rightsquigarrow$ `[3, 3, 6]`.
2. Definir `prodInterno :: [Float] -> [Float] -> Float` que calcule 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. Adaptar `division` para $a < 0$ y/o $d < 0$. Observación: las funciones `div` y `mod` de Haskell no coinciden con el algoritmo de división cuando $d < 0$. Ver también `quot` y `rem`.
4. Implementar `noTieneDivisoresHasta :: Integer -> Integer -> Bool`
   `noTieneDivisoresHasta m n` da `True` sii ningún número entre 2 y `m` divide a `n`.
5. Utilizando `noTieneDivisoresHasta`, programar `esPrimo :: Integer -> Bool`