Práctico 4 de Programación Funcional.

UdelaR/FCien/CMat

xx de septiembre al yy de septiembre de 2016.

1. Se considera la función $s : \mathbb{N} \to \mathbb{N}$ tal que $s(n) = \sum_{i=0}^{n} i$.

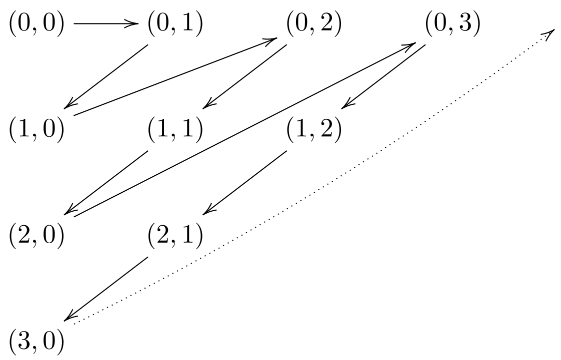
   (a) Definir recursivamente una función `sum1::Int -> Int` que calcule a s.

   (b) Definir `sum2::Int -> Int` que calcule a s, pero usando la fórmula $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

   (c) Definir `sum3::Int -> Int -> Int` que calcule s usando listas y la función `foldr`.

   Compilar con las opciones `-prof -fprof-auto -rtsopts` y correr el ejecutable con los parámetros `+RTS -p`. Comparar qué recursos utiliza cada versión. Redefinirlas en el tipo `Integer` y observar qué recursos usan las definiciones.

2. Se considera la enumeración de $\mathbb{N} \times \mathbb{N}$ descripta por el siguiente diagrama:



   Usaremos esta enumeración para codificar pares y, luego, listas finitas de enteros. Diremos que el *código* de un par `(x,y)` *es el entero positivo que le corresponde en la enumeración.*

   (a) Definir una función `encode :: (Integer, Integer) -> Integer` que calcule el código del par que se le pasa como argumento.

   (b) Definir una función `decode :: Integer -> (Integer, Integer)` que calcule, a partir de un entero, el par que lo tiene por código (**Sug.:**

Dado un entero n, calcular el mayor k tal que $\sum_{i=1}^{k} i \leq n$). Definir las funciones `first, second::Integer -> Integer` que calcula a partir de `x`, la primera y la segunda coordenada de `decode x`.

(c) Usar esta codificación para declarar el tipo `(Integer, Integer)` como instancia de la clase `Enum`. **Sug.:** Los datos de tipo `Integer` se pueden *forzar* dentro del tipo `Int` mediante una declaración explícita.

(d) Para codificar y decodificar listas de enteros procederemos de la siguiente manera:

- El código simple de una lista `[x]` (de largo 1) es `x`. Para una lista de la forma `x:xs`, el código simple es `encode (x, c)`, donde `c` es el código simple de `xs`.
- Esta codificación no es inyectiva, ya que para cada entero n y cada longitud de lista, existe una lista de la cual n es el código. Para resolver esto, definimos el código de una lista `xs` como `encode (l, c)` donde `l` es la longitud de `xs` y `c` es el código simple de `xs`.
- La decodificación de un entero procede ahora en dos etapas: primero se busca la primera coordenada codificada, para saber la longitud de la lista, y luego se decodifica la segunda coordenada, usando que sabemos la longitud de la lista que codifica.

Definir las funciones `encode::[Integer]-> Integer` y `decode::Integer->[Integer]` que codifican y decodifican listas de enteros.

(e) Usar la codificación de listas de enteros para declarar `[Integer]` como instancia de las clases `Eq, Ord, Enum`.

3. Se consideran los números combinatorios $\binom{m}{n}$ definidos como:

$$\binom{m}{n} := m!/n!(m-n)! \text{ donde } m \geq n \tag{1}$$

(a) Definir `choose1 :: Int -> Int -> Int` que calcule los números combinatorios $\binom{m}{n}$ usando la definición (1). Elegir la versión de factorial más eficiente entre las que se hallaron en el ejercicio 5 del práctico 1. **Sug.:** usar funciones que conviertan de enteros a flotantes y viceversa.

(b) Definir `choose2 :: Int -> Int -> Int` que calcule los números combinatorios $\binom{m}{n}$ usando la *fórmula de Stifel*:

$$\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$$

(c) Definir `nthPascalLine :: Int -> [Int]` que calcule la n-ésima fila del triángulo de Pascal. Usando esta función, definir `choose3 :: Int -> Int -> Int` que calcule los números combinatorios $\binom{m}{n}$.

Compilar con las opciones `-prof -fprof-auto -rtsopts` y correr el ejecutable con los parámetros `+RTS -p`, para observar los recursos que usa cada una de las definiciones.

4. Se considera la función $f : \mathbb{N} \to \mathbb{N}$ definida según:

$$f(x) = \begin{cases} 2 + 4 + \cdots + x & \text{si } x \text{ es par} \\ 1 + 3 + \cdots + x & \text{si } x \text{ es impar} \end{cases} \tag{2}$$

   (a) Definir `halfSum1 :: Int -> Int` calcule la función f, de modo recursivo y usando *"guardas"*.

   (b) Definir sin usar recursión `evenSum :: Int -> Int` que calcule f para argumentos *pares*. **Sug.:** inspirarse en la parte (b) del ejercicio 1.

   (c) Usar `evenSum` para definir `oddSum :: Int -> Int` que calcule f para argumentos impares.

   (d) Usar `evenSum` y `oddSum` para definir `halfSum2 :: Int -> Int` que calcule la función f sin usar recursión.

   (e) Definir `halfSum3 :: Int -> Int` usando las funciones `foldr` y `map` para listas.

   Compilar con las opciones `-prof -fprof-auto -rtsopts` y correr el ejecutable con los parámetros `+RTS -p`, para observar los recursos que usa cada una de las definiciones.

5. Se considera $f : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ definida según $(f(g))(n) := \sum_{i=0}^{n} g(i)$.

   (a) Definir `functSum :: (Int -> Int) -> Int -> Int` que calcule f.

   (b) Definir `sqrSum1 :: Int -> Int` que calcule la suma de los cuadrados de 0 a n usando `functSum`.

   (c) Definir `sqrSum2 :: Int -> Int` que calcule la suma de los cuadrados de 0 a n usando la fórmula $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

   (d) Definir `sqrSum3 :: Int -> Int` que calcule la suma de los cuadrados de 0 a n usando las funciones `foldr` y `map` de listas.

   Usar la opción $+$ `s` de Hughs para observar los recursos que usa cada una de las definiciones.

6. Definir `minDivisor :: Int -> Int` que calcule el menor divisor mayor que 1 de su argumento. **Sug.:** el menor divisor de n distinto de 1 es menor o igual que $\sqrt{n}$ o es igual a n.

   (a) Definir `isPrime :: Int -> Bool` que decida si su argumento es primo o no usando `minDivisor`.

   (b) Definir `gcd1 :: Int -> Int -> Int` que calcule el máximo común divisor de sus argumentos usando `minDivisor`.

7. Definir `reverse :: Int -> Int` que invierta los dígitos de un número. Por ejemplo, `reverse 367` debe reducir a `763`.

8. Definir `palindorme :: String -> Bool` que detecte si una palabra es un palíndrome o no.