Departamento de Computación – FCEyN – UBA

Taller de Álgebra I - Recuperatorio

PRIMER CUATRIMESTRE 2018 – *18 de julio de 2018*

Aclaraciones

- El parcial se aprueba con tres ejercicios bien resueltos.
- Programe todas las funciones en lenguaje Haskell. El código debe ser autocontenido. Si utiliza funciones que no existen en Haskell, debe programarlas. Incluya la signatura de todas las funciones que escriba.
- No está permitido alterar los tipos de datos presentados en el enunciado, ni utilizar técnicas no vistas en clase para resolver los ejercicios.

Ejercicio 1

Un año es bisiesto si es múltiplo de 4, salvo que sea múltiplo de 100, caso en que no es bisiesto, salvo si es múltiplo de 400, caso en que sí es bisiesto. Por ejemplo, 1900 no fue bisiesto, 1968 y 2000 sí lo fueron.

Programar una función `bisiesto :: Integer -> Bool` que dada un año devuelva si es bisiesto.

Por ejemplo:
`bisiesto 1901` $\rightsquigarrow$ `False`, `bisiesto 1904` $\rightsquigarrow$ `True`,
`bisiesto 1900` $\rightsquigarrow$ `False`, `bisiesto 2000` $\rightsquigarrow$ `True`.

Ejercicio 2

Dados naturales n y k con $k \leq n$, el *número de Stirling de segunda especie* $\left\{ {n \atop k} \right\}$ es la cantidad de formas de separar un conjunto de n elementos en k subconjuntos no vacíos. Se puede probar que $\left\{ {n \atop 1} \right\} = \left\{ {n \atop n} \right\} = 1$ y que $\left\{ {n+1 \atop k} \right\} = k\left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\}$.

Programar una función `stirling :: Integer -> Integer -> Integer` que dados n y k devuelva $\left\{ {n \atop k} \right\}$.

Por ejemplo: `stirling 3 2` $\rightsquigarrow$ `3`, `stirling 4 2` $\rightsquigarrow$ `7`, `stirling 4 3` $\rightsquigarrow$ `6`.

Ejercicio 3

Dada la función

$$a : \mathbb{N} \to \mathbb{N}, \quad a(n) = \begin{cases} n+1 & \text{si } n \text{ es impar} \\ n/2 & \text{si } n \text{ es par,} \end{cases}$$

programar `composiciones :: Integer -> Integer` que dado un natural `n` cuenta cuántas veces hay que aplicar a para llegar a `1`.

Por ejemplo:
`composiciones 1` $\rightsquigarrow$ `0`,
`composiciones 2` $\rightsquigarrow$ `1` (porque $a(2) = 1$)
`composiciones 5` $\rightsquigarrow$ `5` (porque $a(5) = 6$, $a(6) = 3$, $a(3) = 4$, $a(4) = 2$, $a(2) = 1$)

Ejercicio 4

Programar una función `lugarCuadrado :: [Integer] -> Integer` que dada una lista cuente cuántos elementos coinciden con el cuadrado de su posición.

Por ejemplo:
`lugarCuadrado [1, 7]` $\rightsquigarrow$ 1 (porque $1 = 1^2$ pero $7 \neq 2^2$),
`lugarCuadrado [1, 2, 9, 5]` $\rightsquigarrow$ 2,
`lugarCuadrado [0, 4, 2, 16]` $\rightsquigarrow$ 2.

Ejercicio 5

Se cuenta con el tipo `type Set a = [a]` visto en clase para representar conjuntos mediante listas sin repetidos, en las cuales el orden de los elementos es irrelevante. Programar una función `difsim :: [Set Integer] -> Set Integer` que, dada una lista de conjuntos de enteros, devuelve el conjunto de los enteros que están en una cantidad impar de dichos conjuntos.

Por ejemplo: `difsim [[5,7,8],[],[5,8,4],[5]]` $\rightsquigarrow$ `[5,7,4]`.

Nota de color: esta función calcula la diferencia simétrica de todos los conjuntos.

18/07

Recuperatorio Taller Oficial

1) bisiesto :: Entero → Bool

bisiesto x | mod x 400 == 0 = TRUE
| mod x 100 == 0 = FALSE
| mod x 4 == 0 = TRUE
| otherwise = FALSE

-- asumo que el usuario no ingresara valores x > n o x < 0 y n < 0

2) stirling :: Entero → Entero → Entero

stirling n k | n == k = 1
| k == 1 = 1
| n > k = k * (stirling (n-1) k) + stirling (n-1) (k-1)

3) esPar :: Entero → Bool

esPar x | mod x 2 == 0 = True
| otherwise = False

se puede escribir esPar x = mod x 2 == 0

f :: Entero → Entero

f x | esPar x == True = div x 2 (no hace falta)
| esPar x == False = n+1

B

```
composicion :: Integer -> Integer -> Integer
composicion cont x | f(x) == 1 = cont + 1
                   | otherwise = composicion (cont+1) (f(x))

composicions :: Integer -> Integer
composicions x = composicions 0 x
```

No funciona para $x == 1$ (devuelve 2)

M

```
4) esCuadradoAux :: Integer -> Integer -> Integer
   esCuadradoAux _ 0 = 1
   esCuadradoAux 1 0 = 0        1 ≠ 0²
   esCuadradoAux x y | y² == x = 0
                     | otherwise = esCuadradoAux (y-1)

   esCuadrado :: Integer -> Bool
   esCuadrado x | esCuadradoAux x (x-1) == 0 = TRUE
                | otherwise                   = False
```

falta un parametro

```
sumaCuadrado :: [Integer] -> Integer
sumaCuadrado [] = 0
sumaCuadrado (x:xs) | esCuadrado x == TRUE = 1 + sumaCuadrado xs
                    | otherwise            = sumaCuadrado xs
```

Esto contaría la cantidad de cuadrados
No la cantidad de números que son el cuadrado de su posición.