Práctica 1 - Programación Funcional

Paradigmas de programación (PLP)

beekkles

Ejercicio 1

¿Cuál es el tipo de cada función? (Suponer que todos los números son de tipo Float).

Indicar cuáles de las funciones anteriores no están currificadas. Para cada una de ellas, definir la función currificada correspondiente.

```haskell
max2 :: (Float, Float) -> Float
max2(x, y) | x >= y = x
           | otherwise = y

--Y su versión currificada sería

max2 :: Float -> Float -> Float
max2 x y | x >= y = x
         | otherwise = y
```

```haskell
normaVectorial :: (Float, Float) -> Float
normaVectorial (x, y) = sqrt(x^2 + y^2)

--Y su versión currificada sería

normaVectorial :: Float -> Float -> Float
normaVectorial x y = sqrt(x^2 + y^2)
```

```haskell
subtract :: Float -> Float -> Float
subtract = flip (-)
```

```haskell
predecesor :: Float -> Float
predecesor = subtract 1
```

```haskell
evaluarEnCero(Float -> a) -> a
evaluarEnCero = \f -> f 0
```

```haskell
dosVeces :: (a -> a) -> (a -> a)
dosVeces = \f -> f . f
```

```haskell
flipAll :: [a -> b -> c] -> [b -> a -> c]
flipAll = map flip
```

```haskell
flipRaro :: --Preguntar...
flipRaro = flip flip
```

Ejercicio 2

Definir la función curry, que dada una función de dos argumentos, devuelve su equivalente currificada.

Definir la función uncurry, que dada una función currificada de dos argumentos, devuelve su versión no currificada equivalente. Es la inversa de la anterior.

Se podría definir una función curryN, que tome una función de un número arbitrario de argumentos y devuelva su versión currificada? Sugerencia: pensar cuál sería el tipo de la función.

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y

--Preguntar curryN
```

Ejercicio 3

Redefinir usando foldr las funciones sum, elem, (++), filter y map.

Definir la función mejorSegun :: (a -> Bool) -> [a] -> a que debe devolver el máximo elemento de la lista según una función de comparación, utilizando foldr1. Por ejemplo, maximum = mejorSegun (>).

Definir la función sumasParciales :: Num a -> [a] -> [a], que dada una lista de números debe devolver otra de la misma longitud, que tiene en cada posición la suma parcial de los elementos de la lista original desde la cabeza hasta la posición actual. Por ejemplo, sumasParciales [1,4,-1,0,5] → [1,5,4,4,9].

Definir la función sumaAlt, que realiza la suma alternada de los elementos de una lista. Es decir, de como resultado el primer elemento, menos el segundo, más el tercero, menos el cuarto, etc. Usar foldr.

Hacer lo mismo que en el punto anterior, pero con un orden inverso (el último elemento menos el antecesor, ...)

I.

```
sum :: Num a => [a] -> a
sum xs = foldr1 (\x acc -> x + acc) xs

elem :: [a] -> Bool
elem = foldr (\x acc -> x || acc) true

-- Preguntar igual :D
(++) :: [a] -> [a] -> [a]
```

```haskell
(++) xs ys = foldr (\e acc -> e:acc) ys xs

filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\y acc -> if p y then y:acc else acc) []

map :: (a -> b) -> [a] -> [b]
map f = foldr (\x acc -> (f x):acc) []
```

II.

```haskell
mejorSegún :: (a -> a -> Bool) -> [a] -> a
mejorSegún = foldr1 (\x acc -> if x > acc then x else acc)
```

III.

```haskell
sumasParciales :: Num a => [a] -> [a]
sumasParciales [] = []
sumasParciales [x] = [x]
sumasParciales (x:y:xs) = x : sumasParciales ((x+y):xs)

-- Falta migrarla a estructural
```

IV.

```haskell
sumaAlt :: Num a => [a] -> a
sumaAlt = foldr (\x acc -> x+(-1)*acc) 0
```

IV.

```haskell
sumaAlt :: Num a => [a] -> a
sumaAlt = foldl (\x acc -> (-1)*x+acc) 0
```

Ejercicio 4

Ejercicio 5

Considerar las siguientes funciones:

```
elementosEnPosicionesPares :: [a] -> [a]
elementosEnPosicionesPares [] = []
elementosEnPosicionesPares (x:xs) =
                                  if null xs
                                  then [x]
                                  else x : elementosEnPosicionesPares (tail xs)
--De elementosEnPosicionesPares preguntar una forma no tan rebuscada!

entrelazar :: [a] -> [a] -> [a]
entrelazar [] = id
entrelazar (x:xs) = \ys -> if null ys
                           then x : entrelazar xs []
                           else x : head ys : entrelazar xs (tail ys)
```

Indicar si la recursión utilizada en cada una de ellas es o no estructural. Si lo es, reescribirla utilizando foldr. En caso contrario, explicar el motivo.

No son estructurales polque son explícitos

Forma estructural:

```
elementosEnPosicionesPares :: Num a => [a] -> [a]
elementosEnPosicionesPares xs =
              foldr (\(i, x) acc -> if even i then x:acc else acc) [] (zip [1..] xs)

entrelazar :: [a] -> [a] -> [a]
entrelazar
```

Ejercicio 6

Ejercicio 7

Ejercicio 8

Ejercicio 9

Ejercicio 10

Ejercicio 11

Ejercicio 12

Ejercicio 13

Ejercicio 14

Ejercicio 15