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.

```
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
```

```
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)
```

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

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

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

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

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

```
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.

```haskell
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.

```haskell
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)

-- ver un esquema más sencillo

sumasParciales :: Num a => [a] -> [a]
sumasParciales xs = map (\i -> sum (take i xs)) [1..length xs]
```

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)

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 xs ys =
    foldr (\(x, y) acc -> x:y:acc) [] (zip xs ys) ++ (drop (length ys) xs ++ drop
(length xs) ys)
```

Ejercicio 6

El siguiente esquema captura la recursión primitiva sobre listas.

```
recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
recr _ z [] = z
recr f z (x : xs) = f x xs (recr f z xs)
```

Definir la función sacarUna :: Eq a => a -> [a] -> [a], que dados un elemento y una lista devuelve el resultado de eliminar de la lista la primera aparición del elemento (si está presente).

Explicar por qué el esquema de recursión estructural (foldr) no es adecuado para implementar la función sacarUna del punto anterior.

Definr la función insertarOrdenado :: Ord a => a -> [a] -> [a] que inserta un elemento en una lista ordenada (de manera creciente), de manera que se preserva el ordenamiento.

```haskell
sacarUna :: Eq a => a -> [a] -> [a]
sacarUna e = recr (\x xs acc -> if e == x then xs else x:acc) []
```

Es inadecuado el uso de foldr ya que necesitamos poder "ver" el resto de la lista de forma directa, se puede pero es más engorroso.

```haskell
enMedio :: Ord a => a -> a -> a -> Bool
enMedio x e x' = x <= e && e <= x'

insertarOrdenado :: Ord a => a -> [a] -> [a]
insertarOrdenado e = recr (\x xs acc -> if enMedio x e (head xs) then x:e:xs else
x:acc) []
```

Ejercicio 7

Definir las siguientes funciones para trabajar sobre listas, y dar su tipo. Todas ellas deben poder aplicarse a listas **finitas** e **infitas**.

mapPares, una versión de map que toma una función currificada de dos argumentos y una lista de pares de valores, y devuelve la lista de aplicaciones de la función a cada par. **Pista:** recordar curry y uncurry.

armarPares, que dadas dos listas arma una lista de pares que contiene, en cada posición, el elemento correspondiente a esa posición en cada una de las listas. Si una de las listas es más larga que la otra, ignorar los elementos que sobran (el resultado tendrá la longitud de la lista más corta). Esta función en Haskell se llama zip. Pista: aprovechar la currificación y utilizar evaluación parcial.

mapDoble, una variante de mapPares, que toma una función currificada de dos argumentos y dos listas (de igual longitud), y devuelve una lista de aplicaciones de la función a cada elemento correspondiente de las dos listas. Esta función en Haskell se llama zipWith.

```haskell
mapPares :: (a -> b -> c) -> [(a, b)] -> [c]
mapPares f = foldr (\(x,y) acc -> f x y:acc) []

armarPares :: [a] -> [b] -> [(a,b)]
armarPares [] _ = []
armarPares _ [] = []
armarPares (x:xs) (y:ys) = (x,y):armarPares xs ys

mapDoble :: (a -> b -> c) -> [a] -> [b] -> [c]
mapDoble f xs ys = foldr (\(x,y) acc -> f x y : acc) [] (armarPares xs ys)
```

Ejercicio 8

Ejercicio 9

Definir y dar el tipo del esquema de recursión foldNat sobre los naturales. Utilizar el tipo Integer de Haskell (la función va a estar definida sólo para los enteros mayores o iguales que 0).

Utilizando foldNat, definir la función potencia.

Ejercicio 10

Ejercicio 11

Ejercicio 12

Considerar el siguiente tipo, que representa a los árboles binarios:

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Usando recursión explícita, definir los esquemas de recursión estructural (foldAB) y primitiva (recAB), y dar sus tipos.

Definir las funciones esNil, altura y cantNodos (para esNil puede utilizarse case en lugar de foldAB o recAB).

Definir la función mejorSegún :: (a -> a -> Bool) -> AB a -> a, análoga a la del ejercicio 3, para árboles. Se recomienda definir una función auxiliar para comparar la raíz con un posible resultado de la recursión para un árbol que puede o no ser Nil.

Definir la función esABB :: Ord a => AB a -> Bool que chequea si un árbol es un árbol binario de búsqueda. Recordar que, en un árbol binario de búsqueda, el valor de un nodo es mayor o igual que los valores que aparecen en el subárbol izquierdo y es estrictamente menor que los valores que aparecen en el subárbol derecho.

Justificar la elección de los esquemas de recursión utilizados para los tres puntos anteriores.

Ejercicio 13

Ejercicio 14

Ejercicio 15

Definir el tipo RoseTree de árboles no vacíos, con una cantidad indeterminada de hijos para cada nodo.

Escribir el esquema de recursión estructural para RoseTree. Importante escribir primero su tipo.

Usando el esquema definido, escribir las siguientes funciones:

- hojas, que dado un RoseTree, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el RoseTree.
- distancias, que dado un RoseTree, devuelva las distancias de su raíz a cada una de sus hojas.
- altura, que devuelve la altura de un RoseTree (la cantidad de nodos de la rama más larga). Si el RoseTree es una hoja, se considera que su altura es 1.