Práctica 0: Pre-práctica de programación funcional

Tomás Felipe Melli

March 20, 2025

Índice

1 Ejercicio 1

Nos piden dar los tipos y describir el comportamiento de las siguientes funciones del módulo *prelude* de Haskell.

1.1 null

Se utiliza para verificar si una lista está vacía. Se implementa como sigue :

```
null :: [a] -> Bool
null [] = True
null (_:xs) = False
```

Es decir, toma una lista $[a]$ y devuelve un *bool*.

1.2 head

Toma una lisa y devuelve el primer elemento.

```
head :: [a] -> a
head (x:_) = x
head []    = error "head: lista vac a"
```

1.3 tail

Devuelve todos los elementos de una lista, menos el primero.

```
tail :: [a] -> [a]
tail (_:xs) = xs
tail []     = error "tail: lista vac a"
```

1.4 init

Devuelve todos los elementos de una lista, menos el último.

```
init :: [a] -> [a]
init [x]    = []
init (x:xs) = x : init xs
init []     = error "init: lista vac a"
```

1.5 last

Devuelve el último elemento de una lista

```
last :: [a] -> a
last [x]    = x
last (_:xs) = last xs
last []     = error "last: lista vac a"
```

1.6 take

Toma los primeros n-elementos de una lista.

```
take :: Int -> [a] -> [a]
take 0 _      = []
take n (x:xs) = x : take (n-1) xs
take _ []     = []
```

1.7 drop

Elimina los primeros n-elementos de una lista

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop n (_:xs) = drop (n-1) xs
drop _ []     = []
```

1.8 (++)

Concatena dos listas.

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

1.9 concat

Concatena una lista de listas en una sola lista.

```
concat :: [[a]] -> [a]
concat []     = []
concat (x:xs) = x ++ concat xs
```

1.10 reverse

Invierte el orden de los elementos en una lista.

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

1.11 elem

Verifica si un elemento pertenece a una lista.

```
elem :: Eq a => a -> [a] -> Bool
elem _ []     = False
elem x (y:ys) = (x == y) || elem x ys
```

2 Ejercicio 2

2.1 valorAbsoluto

Nos piden, dado un número dar su valor absoluto.

```
valorAbsoluto :: Float -> Float
valorAbsoluto n
    | n < 0 = -n
    | otherwise = n
```

2.2 bisiesto

La función recibe un año y decide si es o no bisiesto. Para ello recordamos cuándo un año es bisiesto. Debe (divisible por 4) y (no divisible por 100 o divisible por 400)

```
bisiesto :: Int -> Bool
bisiesto n
    | mod n 4 == 0 && mod n 100 /= 0 = True
    | mod n 4 == 0 && mod n 400 == 0 = True
    | otherwise = False
```

2.3 factorial

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

2.4 cantDivisoresPrimos

Vamos a necesitar:

1. Una función que nos diga si un número es primo y su auxiliar
2. Una que nos devuelva los divisores
3. Una que cuente aquellos que son primos

```
esPrimo :: Int -> Bool
esPrimo 0 = False
esPrimo 1 = False
esPrimo n
    | n > 1 = not (divideAlguno n (n-1))

-- Divide alguno se fija si entre [1;n] con los extremos que no pertenecen al conj de divisores, divide
divideAlguno :: Int -> Int -> Bool
divideAlguno n 1 = False
divideAlguno n a
    | mod n a == 0 = True
    | otherwise = divideAlguno n (a-1)

-- sacamos divisores
divisores :: Int -> Int -> [Int]
divisores _ 1 = []
divisores a b
    | mod a b == 0 = b : divisores a (b-1)
    | otherwise = divisores a (b-1)

-- cu ntos primos ?
cantDivisoresPrimos :: Int -> Int
cantDivisoresPrimos 0 = 0
cantDivisoresPrimos 1 = 0
cantDivisoresPrimos n = contarPrimos (divisores n (n-1)) 0

-- cu ntos elementos de la lista son primos
contarPrimos :: [Int] -> Int -> Int
contarPrimos [] cuenta = cuenta
contarPrimos (x:xs) cuenta
    | esPrimo x = contarPrimos xs (cuenta + 1)
    | otherwise = contarPrimos xs cuenta
```

3 Ejercicio 3

En el *prelude* de Haskell tenemos definidos dos tipos **Maybe** y **Either**. Estos tipos están definidos como sigue :

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

Estos tipos están definidos para manejar errores o resultados opcionales. Un ejemplo podría ser una división donde, en caso de ser 0 el divisor, en caso de usar el tipo Maybe, retornaríamos *Nothing*. En caso de utilizar el tipo *Either*, podríamos definirle a *Left* retornar "Error : la división por 0 no está definida" y con *Rigth*, el resultado de hacer la correcta división de a por b.

```
DivMaybe :: Int -> Int -> Maybe Int
DivMaybe _ 0 = Nothing
DivMaybe x y = Just (div x y)
```

```
DivEither :: Int -> Int -> Either String Int
DivEither _ 0 = Left "Error: divisi n por cero"
DivEither x y = Right (div x y)
```

3.1 inverso

```
-- el inverso multiplicativo de un n mero a es el n mero b tal que 'a x b = 1'
inverso :: Float -> Maybe Float
inverso 0 = Nothing
inverso a = Just (1/a)
```

3.2 aEntero

```haskell
-- la idea es convertir una expresi n que puede ser booleana a su equivalente (0,1)
-- Se le pasa a la funci n el par metro como sigue : aEntero (Left 5) o aEntero (Right (True && False))
aEntero :: Either Int Bool -> Int
aEntero (Left n) = n
aEntero (Right False) = 0
aEntero (Right True) = 1
```

4 Ejercicio 4

4.1 limpiar

Es una función que elimina todas las apariciones de los caracteres del primer string en el segundo.

```haskell
limpiar :: String -> String -> String
limpiar [] str2 = str2
limpiar (x:xs) str2
    | elem x str2 = limpiar xs (eliminarChar x str2)
    | otherwise = limpiar xs str2

eliminarChar :: Char -> String -> String
eliminarChar _ [] = []
eliminarChar x (y:ys)
    | x == y = eliminarChar x ys
    | otherwise = y : eliminarChar x ys
```

4.2 difPromedio

Es una función que recibe una lista de números de tipo float, calcula el promedio de ella y luego devuelve una lista con la diferencia entre cada número y el promedio.

```haskell
sumatoria :: [Float] -> Float
sumatoria [] = 0
sumatoria (x:xs) = x + sumatoria xs

elementos :: [Float] -> Float
elementos [] = 0
elementos (x:xs) = 1 + elementos xs

promedio :: [Float] -> Float
promedio [] = 0
promedio x = (sumatoria x) / (elementos x)

difPromedio :: [Float] -> [Float]
difPromedio [] = []
difPromedio lista =
    let prom = promedio lista
    in [x - prom | x <- lista]
```

La notación **let...in** nos permite hacer un cálculo para usarlo luego en... cierta expresión. En este caso, con la *compresión de lista*, nos permite decir que cada elemento de la lista *x ¡- lista* será ahora el resultado de *restarle el promedio*. También se puede definir con **where** como sigue :

```haskell
difPromedio :: [Float] -> [Float]
difPromedio [] = []
difPromedio lista =
        let prom = promedio lista
        in [x - prom | x <- lista]
    where
        promedio :: [Float] -> Float
        promedio [] = 0
        promedio x = (sumatoria x) / (elementos x)
```

Existe la función **map** que lo que hace es aplicar la misma operación a cada elemento de la lista.

```haskell
map :: (a -> b) -> [a] -> [b]
```

Es decir, toma una función que convierte un elemento de tipo 'a' en un elemento de tipo 'b' (es decir, una función de tipo (a -¿ b)), y una lista de tipo [a]. Luego devuelve una lista de tipo [b], que es la lista resultante después de aplicar la función a cada elemento de la lista original. Cómo sería con lo anterior ?

```haskell
difPromedioMap :: [Float] -> [Float]
difPromedioMap [] = []
difPromedioMap lista = map(\x -> x - promedio(lista)) lista
```


```haskell
difPromedioMap :: [Float] -> [Float]
difPromedioMap [] = []
difPromedioMap lista = map(\x -> x - promedio(lista)) lista
```


4.3 todosIguales

Tenemos que chequear si todos los elementos (enteros) de un array son iguales.

```haskell
todosIguales :: [Int] -> Bool
todosIguales [] = True
todosIguales (x:xs)
    | xs /= [] && x == head xs = todosIguales xs
    | xs == [] = True
    | otherwise = False
```

5 Ejercicio 5

Vamos a trabajar con árboles binarios. Aclaraciones : Se definen como sigue :

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

- **data** se utiliza para declarar un nuevo tipo de datos
- **AB** es el nombre que le damos al tipo de datos
- **a** es el parámetro tipo
- **Nil** es el constructor que representa el árbol vacío
- **Bin (AB a) a (AB a)** es el constructor que representa un nodo en el árbol que define el subárbol izquierdo - valor del nodo - subárbol derecho

Luego vamos a crearnos dos árboles, uno vacío y otro con un elemento :

```haskell
arbolVacio :: AB Int
arbolVacio = Nil

arbolNoVacio :: AB Int
arbolNoVacio = Bin Nil 1 Nil
```

5.1 vacioAB

Esta función nos dice si el árbol es vacío o no.

```haskell
vacioAB :: AB a -> Bool
vacioAB Nil = True
vacioAB _ = False
```

5.2 negacionAB

Nos pide trabajar sobre un árbol donde los nodos son booleanos y tenemos que retornar un árbol negado. El primer paso es declarar el árbol de bool :

```haskell
arbolBool :: AB Bool
arbolBool = Bin (Bin Nil True Nil) False (Bin Nil True Nil)
```

Luego tenemos que ver cómo se recorre un árbol. La idea es algo así :

```haskell
recorrerAB :: AB a -> [a]
recorrerAB Nil = []
recorrerAB (Bin izq x der) = x : recorrerAB izq ++ recorrerAB der
```

Con esto en mente, queremos negar cada nodo y construir uno nuevo.

```haskell
negacionAB :: AB Bool -> AB Bool
negacionAB Nil = Nil
negacionAB (Bin izq x der) = Bin (negacionAB izq) (not x) (negacionAB der)
```

5.3 productoAB

Nos piden hacer el producto total entre los elementos de un árbol de enteros. Definimos el árbol de enteros y uno vacío para probar que no se rompa en ese caso.

```
arbolEnteros :: AB Int
arbolEnteros = Bin (Bin Nil 2 Nil) 3 (Bin Nil 4 Nil)
```

Ya vimos cómo recorrer un árbol binario y devolver una lista, con esto, vamos a construirnos la lista para hacer las multiplicaciones. Además reutilizamos la función 'elementos' pero ahora con Int.

```
elementos2 :: [Int] -> Int
elementos2 [] = 0
elementos2 (x:xs) = 1 + elementos2 xs

productoAB :: AB Int -> Int
productoAB arbol
    | total == 0 = 0
    | otherwise = producto lista
    where
        lista = recorrerAB arbol
        total = elementos2 lista
        producto :: [Int] -> Int
        producto list
            | list == [] = 1
            | otherwise = head list * producto (tail list)
```