Práctica N° 1 - Programación Funcional

Para resolver esta práctica, recomendamos usar el intérprete "GHCI", de distribución gratuita, que puede bajarse de `https://www.haskell.org/ghc/`.

Para resolver los ejercicios **no** está permitido usar recursión explícita, a menos que se indique lo contrario.

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Currificación y tipos

Ejercicio 1 ★

Considerar las siguientes definiciones de funciones:

```
- max2 (x, y)  | x >= y = x
               | otherwise = y
- normaVectorial (x, y) = sqrt (x^2 + y^2)
- subtract = flip (-)
- predecesor = subtract 1
```

```
- evaluarEnCero = \f -> f 0
- dosVeces = \f -> f . f
- flipAll = map flip
- flipRaro = flip flip
```

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

II. Indicar cuáles de las funciones anteriores *no* están currificadas. Para cada una de ellas, definir la función currificada correspondiente. Recordar dar el tipo de la función.

Ejercicio 2 ★

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

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

III. ¿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.

Esquemas de recursión

Ejercicio 3 ★

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

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

III. Definir la función `sumasParciales :: Num a => [a] -> [a]`, que dada una lista de números devuelve 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]`.

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

V. Hacer lo mismo que en el punto anterior, pero en sentido inverso (el último elemento menos el anteúltimo, etc.). Pensar qué esquema de recursión conviene usar en este caso.

Ejercicio 4

I. Definir la función `permutaciones :: [a] -> [[a]]`, que dada una lista devuelve todas sus permutaciones. Se recomienda utilizar `concatMap :: (a -> [b]) -> [a] -> [b]`, y también `take` y `drop`.

II. Definir la función `partes`, que recibe una lista `L` y devuelve la lista de todas las listas formadas por los mismos elementos de `L`, en su mismo orden de aparición.

Ejemplo: `partes [5, 1, 2]` → `[[], [5], [1], [2], [5, 1], [5, 2], [1, 2], [5, 1, 2]]` (en algún orden).

III. Definir la función `prefijos`, que dada una lista, devuelve todos sus prefijos.

Ejemplo: `prefijos [5, 1, 2]` → `[[], [5], [5, 1], [5, 1, 2]]`

IV. Definir la función `sublistas` que, dada una lista, devuelve todas sus sublistas (listas de elementos que aparecen consecutivos en la lista original).

Ejemplo: `sublistas [5, 1, 2]` → `[[], [5], [1], [2], [5, 1], [1, 2], [5, 1, 2]]` (en algún orden).

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.

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)
```

a. 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).

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

c. 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.

Ejercicio 7

I. Definir la función `genLista :: a -> (a -> a) -> Integer -> [a]`, que genera una lista de una cantidad dada de elementos, a partir de un elemento inicial y de una función de incremento entre los elementos de la lista. Dicha función de incremento, dado un elemento de la lista, devuelve el elemento siguiente.

II. Usando `genLista`, definir la función `desdeHasta`, que dado un par de números (el primero menor que el segundo), devuelve una lista de números consecutivos desde el primero hasta el segundo.

Ejercicio 8 ★

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

I. `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`.

II. `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.

III. `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`.

Ejercicio 9

I. Escribir la función `sumaMat`, que representa la suma de matrices, usando `zipWith`. Representaremos una matriz como la lista de sus filas. Esto quiere decir que cada matriz será una lista finita de listas finitas, todas de la misma longitud, con elementos enteros. Recordamos que la suma de matrices se define como la suma celda a celda. Asumir que las dos matrices a sumar están bien formadas y tienen las mismas dimensiones.
`sumaMat :: [[Int]] -> [[Int]] -> [[Int]]`

II. Escribir la función `trasponer`, que, dada una matriz como las del ítem I, devuelva su traspuesta. Es decir, en la posición i, j del resultado está el contenido de la posición j, i de la matriz original. Notar que si la entrada es una lista de N listas, todas de longitud M, la salida debe tener M listas, todas de longitud N.
`trasponer :: [[Int]] -> [[Int]]`

Ejercicio 10 ★

Definimos la función `generate`, que genera listas en base a un predicado y una función, de la siguiente manera:

```
generate :: ([a] -> Bool) -> ([a] -> a) -> [a]
generate stop next = generateFrom stop next []

generateFrom:: ([a] -> Bool) -> ([a] -> a) -> [a] -> [a]
generateFrom stop next xs | stop xs   = init xs
                          | otherwise = generateFrom stop next (xs ++ [next xs])
```

I. Usando `generate`, definir `generateBase::([a] -> Bool) -> a -> (a -> a) -> [a]`, similar a `generate`, pero con un caso base para el elemento inicial, y una función que, en lugar de calcular el siguiente elemento en base a la lista completa, lo calcula a partir del último elemento. Por ejemplo: `generateBase (\l->not (null l) && (last l > 256)) 1 (*2)` es la lista las potencias de 2 menores o iguales que 256.

II. Usando `generate`, definir `factoriales::Int -> [Int]`, que dado un entero `n` genera la lista de los primeros `n` factoriales.

III. Usando `generateBase`, definir `iterateN :: Int -> (a -> a) -> a -> [a]` que, toma un entero `n`, una función `f` y un elemento inicial `x`, y devuelve la lista `[x, f x, f (f x), ..., f ( ...(f x) ...)]` de longitud `n`. **Nota:** `iterateN n f x = take n (iterate f x)`.

IV. Redefinir `generateFrom` usando `iterate` y `takeWhile`.

Otras estructuras de datos

En esta sección se permite (y se espera) el uso de recursión explícita *únicamente* para la definición de esquemas de recursión.

Ejercicio 11 ★

I. 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).

II. Utilizando `foldNat`, definir la función `potencia`.

Ejercicio 12

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)
```

Luego usar el esquema definido para escribir la función `evaluar :: Num a => a -> Polinomio a -> a` que, dado un número y un polinomio, devuelve el resultado de evaluar el polinomio dado en el número dado.

Ejercicio 13 ★

Considerar el siguiente tipo, que representa a los árboles binarios:
`data AB a = Nil | Bin (AB a) a (AB a)`

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

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

III. 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`.

IV. 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.

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

Ejercicio 14

Dado el tipo `AB a` del ejercicio 13:

I. Definir las funciones `ramas` (caminos desde la raíz hasta las hojas), `cantHojas` y `espejo`.

II. Definir la función `mismaEstructura :: AB a -> AB b -> Bool` que, dados dos árboles, indica si éstos tienen la misma forma, independientemente del contenido de sus nodos. **Pista:** usar evaluación parcial y recordar el ejercicio 8.

Ejercicio 15

Se desea modelar en Haskell los árboles con información en las hojas (y sólo en ellas). Para esto introduciremos el siguiente tipo:

```
data AIH a = Hoja a | Bin (AIH a) (AIH a)
```

a) Definir el esquema de recursión estructural `foldAIH` y dar su tipo. Por tratarse del primer esquema de recursión que tenemos para este tipo, se permite usar recursión explícita.

b) Escribir las funciones `altura :: AIH a -> Integer` y `tamaño :: AIH a -> Integer`.
Considerar que la altura de una hoja es 1 y el tamaño de un `AIH` es su cantidad de hojas.

Ejercicio 16 ★

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

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

III. Usando el esquema definido, escribir las siguientes funciones:

a) `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.

b) `distancias`, que dado un `RoseTree`, devuelva las distancias de su raíz a cada una de sus hojas.

c) `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.

Ejercicio 17 *(Opcional)*

Se desea representar conjuntos mediante Hashing abierto (*chain addressing*). El Hashing abierto consta de dos funciones: una *función de Hash*, que dado un elemento devuelve un valor entero (el cual se espera que no se repita con frecuencia), y una *tabla de Hash*, que dado un número entero devuelve los elementos del conjunto a los que la función de Hash asignó dicho número (es decir, la preimagen de la función de Hash para ese número).

Los representaremos en Haskell de la siguiente manera:

```
data HashSet a = Hash (a -> Integer) (Integer -> [a])
```

Por contexto de uso, vamos a suponer que la tabla de Hash es una función total, que devuelve listas vacías para los números que no corresponden a elementos del conjunto. Este es un **invariante** que deberá preservarse en todas las funciones que devuelvan conjuntos.

Definir las siguientes funciones:

I. `vacío :: (a -> Integer) -> HashSet a`, que devuelve un conjunto vacío con la función de Hash indicada.

II. `pertenece :: Eq a => a -> HashSet a -> Bool`, que indica si un elemento pertenece a un conjunto. Es decir, si se encuentra en la lista obtenida en la tabla de Hash para el número correspondiente a la función de Hash del elemento.

Por ejemplo:
`pertenece 5 $ agregar 1 $ agregar 2 $ agregar 1 $ vacío (flip mod 5)` devuelve `False`.
`pertenece 2 $ agregar 1 $ agregar 2 $ agregar 1 $ vacío (flip mod 5)` devuelve `True`.

III. `agregar :: Eq a => a -> HashSet a -> HashSet a`, que agrega un elemento a un conjunto. Si el elemento ya estaba en el conjunto, se debe devolver el conjunto sin modificaciones.

IV. `intersección :: Eq a => HashSet a -> HashSet a -> HashSet a` que, dados dos conjuntos, devuelve un conjunto con la misma función de Hash del primero y con los elementos que pertenecen a ambos conjuntos a la vez.

V. `foldr1`(no relacionada con los conjuntos). Dar el tipo y definir la función `foldr1` para listas **sin usar recursión explícita**, recurriendo a alguno de los esquemas de recursión conocidos.

Se recomienda usar la función `error :: String -> a` para el caso de la lista vacía.

Generación infinita (opcional)

Para resolver los ejercicios de esta parte se recomienda leer el apunte "Las tres leyes de la generación infinita", que se encuentra en la sección Útil del campus.

Ejercicio 18

¿Cuál es el valor de esta expresión?

```
[ x | x <- [1..3], y <- [x..3], (x + y) `mod` 3 == 0 ]
```

Ejercicio 19

Definir la lista infinita `paresDeNat::[(Int,Int)]`, que contenga todos los pares de números naturales: (0,0), (0,1), (1,0), etc.

Ejercicio 20

Una tripla pitagórica es una tripla (a, b, c) de enteros positivos tal que $a^2 + b^2 = c^2$.
La siguiente expresión intenta ser una definición de una lista (infinita) de triplas pitagóricas:

```
pitagóricas :: [(Integer, Integer, Integer)]
pitagóricas = [(a, b, c) | a <- [1..], b <-[1..], c <- [1..], a^2 + b^2 == c^2]
```

Explicar por qué esta definición no es útil. Dar una definición mejor.

Ejercicio 21

Escribir la función `listasQueSuman :: Int -> [[Int]]` que, dado un número natural n, devuelve todas las listas de enteros positivos (es decir, mayores o iguales que 1) cuya suma sea n. Para este ejercicio **se permite usar recursión explícita**. Pensar por qué la recursón utilizada no es estructural. (Este ejercicio no es de generación infinita, pero puede ser útil para otras funciones que generen listas infinitas de listas).

Ejercicio 22

Definir en Haskell una lista que contenga todas las listas finitas de enteros positivos (esto es, con elementos mayores o iguales que 1).

Ejercicio 23

Dado el tipo de datos `AIH a` definido en el ejercicio 15:

a) Definir la lista (infinita) de todos los `AIH` cuyas hojas tienen tipo `()`[1]. Se recomienda definir una función auxiliar. Para este ejercicio **se permite utilizar recursión explícita**.

b) Explicar por qué la recursión utilizada en el punto a) no es estructural.

[1] El tipo (), usualmente conocido como *unit*, tiene un único valor, denotado como ().