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 en Haskell

Ejercicio 1 ★

Sean 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? (Asumir que todos los números son de tipo Float).

II. ¿Alguna de las funciones anteriores no está currificada? De ser así, escribir la versión currificada junto con su tipo para cada una de ellas.

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?

Listas por comprensión

Ejercicio 3

¿Cuál es el valor de esta expresión?

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

Ejercicio 4 ★

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 5 ★

Generar la lista de los primeros mil números primos. Observar cómo la evaluación *lazy* facilita la implementación de esta lista.

Ejercicio 6

Usando listas por comprensión, escribir la función `partir :: [a] -> [([a], [a])]` que, dada una lista xs, devuelve todas las maneras posibles de partirla en dos sublistas xs_1 y xs_2 tales que $xs_1 ++ xs_2 == xs$.

Ejemplo: `partir [1, 2, 3]` → `[([], [1, 2, 3]),([1], [2, 3]), ([1, 2], [3]), ([1, 2, 3], [])]`

Ejercicio 7 ★

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.

Ejercicio 8 ★

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

Esquemas de recursión

Ejercicio 9 ★

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]` $\rightsquigarrow$ `[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.

VI. 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`.

Ejercicio 10

I. 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).

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

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

III. 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 11 ★

Sean 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 12 ★

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 `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.

d. La función `listasQueSuman` del ejercicio 7, ¿se ajusta al esquema de recursión `recr`? ¿Por qué o por qué no?

Ejercicio 13

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b = (a -> Bool) -- determina si es o no el caso trivial
                       -> (a -> b)      -- resuelve el caso trivial
                       -> (a -> [a])    -- parte el problema en sub-problemas
                       -> ([b] -> b)    -- combina resultados
                       -> a             -- estructura de entrada
                       -> b             -- resultado
```

Definir las siguientes funciones:

I. `dc :: DivideConquer a b` que implementa la técnica. Es decir, completar la siguiente definición:
`dc trivial solve split combine x = ...`
La forma en que funciona es, dado un dato x, verifica si es o no un caso base utilizando la función *trivial*. En caso de serlo, utilizaremos *solve* para dar el resultado final. En caso de no ser un caso base, partimos el problema utilizando la función *split* y luego combinamos los resultados recursivos utilizando *combine*. Por ser este un esquema de recursión, puede utilizarse recursión explícita para definirlo.

II. Implementar la función `mergeSort :: Ord a => [a] -> [a]` en términos de `dc`.
`mergeSort = dc ...` (se recomienda utilizar `break` y aplicación parcial para definir la función de *combine*).

III. Utilizar el esquema `dc` para reimplementar `map` y `filter`.
`map :: (a -> b) -> [a] -> [b]`
`filter :: (a -> Bool) -> [a] -> [a]`

Ejercicio 14

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 15 ★

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 16

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, entonces el resultado debe tener M listas, todas de longitud N.
`trasponer :: [[Int]] -> [[Int]]`

Ejercicio 17 ★

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 18 ★

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 19

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`

Ejercicio 20 ★

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si c es un conjunto y e un elemento, la expresión $c\ e$ devuelve `True` sii e pertenece a c.

I. Definir la constante `vacío :: Conj a`, y la función `agregar :: Eq a => a -> Conj a -> Conj a`.

II. Escribir las funciones `intersección` y `unión` (ambas de tipo `Conj a -> Conj a-> Conj a`).

III. Definir un conjunto de funciones que contenga infinitos elementos, y dar su tipo.

IV. Definir la función `singleton :: Eq a => a -> Conj a`, que dado un valor genere un conjunto con ese valor como único elemento.

V. ¿Puede definirse un `map` para esta estructura? ¿De qué manera, o por qué no?

Ejercicio 21

En este ejercicio trabajaremos con matrices infinitas representadas como funciones:
`type MatrizInfinita a = Int->Int->a`
donde el primer argumento corresponde a la fila, el segundo a la columna y el resultado al valor contenido en la celda correspondiente.

Por ejemplo, las siguientes definiciones:

```
identidad = \i j->if i==j then 1 else 0
cantor = \x y->(x+y)*(x+y+1)`div`2+y
pares = \x y->(x,y)
```

corresponden a las matrices:

1	0	0	⋯
0	1	0	⋯
0	0	1	⋯
⋮	⋮	⋮	⋱

identidad

0	2	5	⋯
1	4	8	⋯
3	7	12	⋯
⋮	⋮	⋮	⋱

cantor

(0,0)	(0,1)	(0,2)	⋯
(1,0)	(1,1)	(1,2)	⋯
(2,0)	(2,1)	(2,2)	⋯
⋮	⋮	⋮	⋱

pares

Definir las siguientes funciones:

I. `fila::Int->MatrizInfinita a->[a]` y `columna::Int->MatrizInfinita a->[a]` que, dado un índice, devuelven respectivamente la fila o la columna correspondiente en la matriz (en forma de lista infinita). Por ejemplo, `fila 0 identidad` devuelve la lista con un 1 seguido de infinitos 0s.

II. `trasponer::MatrizInfinita a->MatrizInfinita a`, que dada una matriz devuelve su transpuesta.

III. `mapMatriz::(a->b)->MatrizInfinita a->MatrizInfinita b`,
`filterMatriz::(a->Bool)->MatrizInfinita a->[a]` y
`zipWithMatriz::(a->b->c)->MatrizInfinita a->MatrizInfinita b->MatrizInfinita c`, que se comportan como `map`, `filter` y `zipWith` respectivamente, pero aplicadas a matrices infinitas. En el caso de `filterMatriz` no importa el orden en el que se devuelvan los elementos, pero se debe pasar una y sólo una vez por cada posición de la matriz.

IV. `suma::Num a=>MatrizInfinita a->MatrizInfinita a->MatrizInfinita a`, y
`zipMatriz::MatrizInfinita a->MatrizInfinita b->MatrizInfinita (a,b)`. Definir ambas utilizando `zipWithMatriz`.

Ejercicio 22 ★

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

I. Definir los esquemas de recursión estructural (`foldAB`) y primitiva (`recAB`), y dar su tipo.

II. Definir las funciones `esNil` 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 9, 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.

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

Ejercicio 23

Dado el tipo `AB a` del ejercicio 22:

I. Definir las `altura`, `ramas`, `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 16.

Ejercicio 24

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.

c) 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.

d) Explicar por qué la recursión utilizada en el punto c) no es estructural.

Ejercicio 25 ★

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.

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

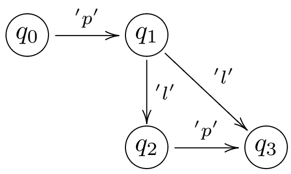
Ejercicio 26 ★

Las máquinas de estados no determinísticas (MEN) se pueden ver como una descripción de un sistema que, al recibir como entrada una constante de un alfabeto (que en general llamamos Σ), y encontrándose en un estado q, altera su estado según lo indique una función de transición (que en general llamamos δ). Observemos que al ser una máquina no determinística, el resultado de esta función de transición no es un único estado sino un conjunto de ellos. Modelaremos estos autómatas mediante el tipo `MEN`[2].

```
data MEN a b = AM {sigma :: [a], delta :: (a -> b -> [b])}
```

Luego, el sistema representado por `m :: MEN a b` que se encuentra en un estado `q :: b` , después de recibir una entrada `s :: a` tal que `s` $\in$ `sigma m`, se encontrará en alguno de los estados `delta m s q` (si esta lista es vacía significa que `s` es una transición inválida, mientras que si contiene muchos estados, significa que puede alcanzar cuaquiera de ellos, sin que podamos suponer nada sobre cuál será).

M_{PLP} =




```
M_PLP :: MEN Char Char
M_PLP = AM ['l','p'] tran
   where tran s e | e == q0 && s == 'p' = [q1]
                  | e == q1 && s == 'l' = [q2,q3]
                  | e == q2 && s == 'p' = [q3]
                  | otherwise = []
```

Se pide definir las siguientes funciones, **sin utilizar recursión explícita**: [3].

I. *a*) `agregarTransicion :: a -> b -> b -> MEN a b -> MEN a b` que, dada una constante s y dos estados q_0 y q_f, agrega al autómata la transición por s desde q_0 a q_f. Si lo necesita, puede suponer que la transición no está previamente definida en el autómata, que s ya pertenece al alfabeto y que está definida la igualdad para los tipos `a` y `b`, indicando qué suposiciones realiza y por qué.

b) `interseccion :: Eq a => MEN a b -> MEN a c -> MEN a (b,c)` que dados dos autómatas m y n, devuelve el autómata intersección, cuyo alfabeto es la intersección de los dos alfabetos, cuyos estados son el producto cartesiano del conjunto de estados de cada uno y que puede moverse de (q_m, q_n) por el símbolo s al estado (q'_m, q'_n) si y solo si m puede moverse de q_m a q'_m por s y n puede moverse de q_n a q'_n por el mismo s.

II. `consumir :: MEN a b -> b -> [a] -> [b]` que, dados un autómata m, un estado q y una cadena de símbolos ss, devuelve todos los estados en los que se puede encontrar m después de haber leido los símbolos de ss (en ese orden), habiendo partido del estado q. Si lo necesita, puede suponer definida la igualdad para los tipos `a` y `b`.

En el autómata de ejemplo, `consumir` M_{PLP} q_0 "*pl*" $\rightsquigarrow$ `[`q_2`,`q_3`]`

III. `trazas :: MEN a b -> b -> [[a]]` que, dado un autómata m y un estado q, devuelve la lista con todas las trazas posibles en m a partir de q, es decir, todas las cadenas de símbolos que pueden llevar a m desde q a algún estado mediante transiciones válidas.
Asumir que existe al menos un ciclo en el autómata, por lo que la lista resultante es infinita. Si lo necesita, puede suponer que tanto el alfabeto como el resultado de las transiciones son finitos, y que está definida la igualdad para los tipos `a` y `b`. Deberá indicar qué suposiciones realiza y por qué.

En el autómata de ejemplo, `trazas` M_{PLP} q_0 $\rightsquigarrow$ `[['p'], ['p', 'l'], ['p', 'l', 'p']]`

[2] Observar que con esta definición, automáticamente se definen las funciones `sigma :: MEN a b -> [a]` que devuelve el alfabeto de la máquina y `delta :: MEN a b -> (a -> b -> [b])` que devuelve su función de transición.

[3] Se recomienda utilizar la función `union :: Eq a => [a] -> [a] -> [a]`.