Práctica Nº 1 - Programación Funcional

Para resolver esta práctica, recomendamos usar el "Hugs 98", de distribución gratuita, que puede bajarse de `http://www.haskell.org/hugs/`.

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 (-)
- restarUno = subtract 1
- evaluarEnCero = \f -> f 0
- dosVeces = \f -> f.f
- flipAll = map 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..2], y <- [x..3], (x + y) `mod` 2 == 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 `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`.

III. 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 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

I. 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). Sugerencia: usar `break` o `recr`.

II. Utilizar listas por comprensión para definir la función `permutaciones`, que dada una lista, devuelve todas sus permutaciones. Para esto se permite usar recursión explícita.

Ejercicio 12

I. Definir la función `genLista`, 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 13 ★

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.

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 a las dos listas. Esta función en Haskell se llama `zipWith`.

Ejercicio 14

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]]`

III. Escribir la función `zipWithList`, que, dada una lista de listas, un caso base y una función combinadora, hace un zipWith de los elementos correspondientes de todas las listas.
`zipWithList :: (a -> b -> b) -> b -> [[a]] -> [b]`

`zipWithList func base [[a1, ..., an], [b1, ..., bn], ..., [z1, ..., zn]]` reduce a
`[foldr func base [a1, ..., z1], foldr func base [a2, ..., z2], ..., foldr func base [an, ..., zn]]`

Se puede asumir que ninguna de las listas es infinita.

IV. ¿Cómo cambian las funciones de los ítems I y II si la matriz, en vez de representarse como una lista de listas, se representara como una función de `Int -> Int -> a` (más dos enteros para indicar la dimensión)? ¿Pueden definirse esquemas como `map` o `zipWith` para este tipo de estructura? ¿De qué manera, o por qué no?

Nota: se puede hacer el ítem II usando la función del ítem III o viceversa (si `trasponer` es lo suficientemente general como para aplicarse a listas de listas arbitrarias). Obviamente, no sería correcto hacer las dos cosas a la vez.

Ejercicio 15 ★

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 = 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 solo 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`, genere 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`.

Ejercicio 16

Este ejercicio está basado en un esquema particular de recursión que representa la técnica algorítmica *divide and conquer* aplicada a problemas de listas. El esquema `dac`, entonces, tendrá el siguiente tipo e implementación:

```
dac :: b -> (a -> b) -> ([a] -> ([a],[a])) -> ([a] -> b -> b -> b) -> [a] -> b
dac base base1 divide combine [] = base
dac base base1 divide combine [x] = base1 x
dac base base1 divide combine input = combine input (rec izquierda) (rec derecha)
        where rec = dac base base1 divide combine
              izquierda = fst (divide input)
              derecha = snd (divide input)
```

Aquí `base` representa el caso base del problema para listas vacías, `base1` el caso base cuando la lista tiene exactamente un elemento, `divide` la función que divide el problema en exactamente 2 partes más chicas. Pediremos que la unión (como conjunto) del par de salida al aplicar `divide` sea exactamente igual a su entrada y que ninguno de los componentes de su salida sea vacío (para que sea efectivamente una función de división y no genere una recursión infinita). Por último, `combine` combina los resultados recursivos (pudiendo utilizar información de la lista original).

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

II. Usar `dac` para implementar el algoritmo de ordenamiento *merge-sort*. Dicho algoritmo se comporta de la siguiente manera: se divide a la lista a ordernar a la mitad (si es impar, en 2 partes lo más iguales posibles), se ordenan esas partes y luego se fusionan ("merge") en orden lineal, tomando iterativamente el más chico de las dos cabezas hasta que una lista queda vacía, momento en el cual agrega al final todo lo que queda de la otra y devuelve eso como resultado.
`mergeSort :: Ord a => [a] -> [a]`

Nota: para hacer la función auxiliar `merge` se puede usar recursión explícita.

III. Usar `dac` para implementar el algoritmo de ordenamiento *quick-sort*. Dicho algoritmo se comporta de la siguiente manera: Se elige un elemento como pivote (en nuestro caso será siempre el primero de la lista) y se parte el resto eligiendo todos los menores o iguales para un lado y todos los estrictamente mayores para el otro. El pivote se incluye en la lista que haya quedado más corta (para evitar que alguna quede vacía). Ambas listas ya están ordenadas entre sí, por lo cual luego de ordenar cada una es simple obtener el resultado final.
`quickSort :: Ord a => [a] -> [a]`

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 17 ★

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 18

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Num a => 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 19 ★

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. Escribir la función `primeraAparición :: a -> [Conj a] -> Int` que, dados un elemento e y una lista de conjuntos (que puede ser finita o infinita), devuelva la primera posición en la cual el conjunto correspondiente contiene al elemento e. Se asume que e pertenece al menos a un conjunto de la lista.

Por ejemplo, sean los siguientes conjuntos:

```
conjunto1 = agregar 'a' (agregar 'b' (agregar 'c' (agregar 'a' vacío)))
conjunto2 = agregar 'a' (agregar 'd' (agregar 'a' vacío))
conjunto3 = agregar 'b' (agregar 'd' (agregar 'e' vacío))
```

y la lista: `listaInfinita = conjunto1:conjunto2:conjunto3:listaInfinita`

`primeraAparición 'a' listaInfinita` devuelve `0`.
`primeraAparición 'd' listaInfinita` devuelve `1`.
`primeraAparición 'e' listaInfinita` devuelve `2`.
`primeraAparición 'c' (tail listaInfinita)` devuelve `2`.

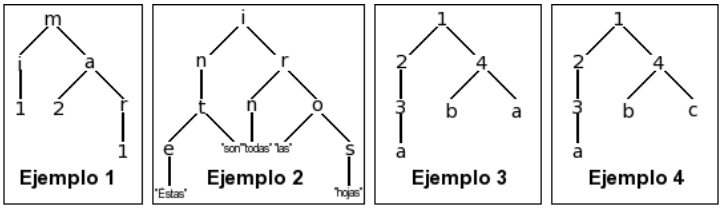
Ejercicio 20 ★

Consideremos el siguiente tipo de datos:

```
data AHD tInterno tHoja = Hoja tHoja
                        | Rama tInterno (AHD tInterno tHoja)
                        | Bin (AHD tInterno tHoja) tInterno (AHD tInterno tHoja)
```

que representa un árbol binario no vacío cuyos nodos internos pueden tener datos de un tipo diferente al de sus hojas. (AHD = árbol con hojas distinguidas).

Por ejemplo:
`Bin (Hoja ''hola'') 'b' (Rama 'c' (Hoja ''chau'''))` tiene tipo `AHD Char String`
`Rama 1 (Bin(Hoja True)(-2)(Hoja False))` tiene tipo `AHD Int Bool`
A continuación mostramos algunos ejemplos de forma más gráfica:



I. Escribir el esquema de recursión estructural (`fold`) para este tipo de datos.

```
foldAHD :: (tHoja -> b) -> (tInterno -> b -> b) -> (b -> tInterno -> b -> b)
           -> AHD tInterno tHoja -> b
```

Para este inciso está permitido utilizar recursión explícita.

II. Escribir, usando `foldAHD`, la función `mapAHD :: (a -> b) -> (c -> d) -> AHD a c -> AHD b d`, que actúa de manera análoga al `map` de listas, aplicando la primera función a los nodos internos y la segunda a las hojas. Por ejemplo:

`mapAHD (+1) not (Bin(Rama 1 (Hoja False)) 2 (Bin(Hoja False) 3 (Rama 5 (Hoja True))))`
devuelve `Bin (Rama 2 (Hoja True)) 3 (Bin (Hoja True) 4 (Rama 6 (Hoja False)))`.

III. Escribir una función que analice un `AHD` de la siguiente manera: si el árbol posee al menos una hoja repetida, se debe devolver una función que, dado un elemento del tipo de las hojas, indique su cantidad de apariciones como hoja del árbol. En caso contrario (si no hay hojas repetidas), se debe devolver el recorrido DFS de los nodos internos del árbol. (Recordar que el recorrido DFS comienza por la raíz y explora cada rama en profundidad, pasando una sola vez por cada nodo).

Para esto utilizaremos el tipo `Either` del Prelude. De esta manera, el tipo de la función pedida es:

`analizar :: Eq tHoja => AHD tInterno tHoja -> Either (tHoja -> Int) [tInterno]`

Ejemplos (para los árboles dibujados arriba):

`analizar ejemplo2` devuelve `Right` "`internos`".
`analizar ejemplo3` devulelve `Left repeticionesEj3`, donde `repeticionesEj3` es una función con el siguiente comportamiento:

repeticionesEj3 'a' = 2
repeticionesEj3 'b' = 1
repeticionesEj3 x = 0 *para todo* carácter x *distinto de* 'a' *y* 'b'.

`analizar ejemplo4` devuelve `Right [1,2,3,4]`

Ejercicio 21

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

I. Definir el esquema de recursión estructural (*fold*) para estos árboles, y dar su tipo.

II. Definir las funciones `esNil`, `altura`, `ramas` (caminos desde la raíz hasta las hojas), `#nodos`, `#hojas` y `espejo` (para `esNil` puede utilizarse `case` en lugar de `fold`).

III. 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 14.

Ejercicio 22 ★

I. Definir el tipo de datos `RoseTree` de árboles no vacíos, donde cada nodo tiene una cantidad indeterminada de hijos.

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.