Práctica de ejercicios #5 - Esquemas de recursión sobre listas

Programación Funcional, Universidad Nacional de Hurlingham

10 de mayo de 2025

Aclaraciones:

- ***Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados.*** *No se saltee ejercicios sin consultar antes a un docente.*
- ***Recuerde que puede aprovechar en todo momento las funciones que ha definido****, tanto las de esta misma práctica como las de prácticas anteriores.*
- ***Pruebe todas sus implementaciones****, al menos en una consola interactiva.*
- *Es sumamente aconsejable* ***resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en videos publicados o clases presenciales****, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

ACLARACIÓN GENERAL: Cuando se pide *"definir sin utilizar recursión en forma explícita"*, quiere decir que solamente se puede utilizar **`foldr`** o **`recr`**, en conjunto con algunos combinadores (como por ejemplo **`flip`**). En particular, las definiciones NO deben tomar la lista de forma explícita (o sea, la función que transforma la lista en el resultado DEBE ser el resultado de un **`foldr`** con solamente 2 argumentos, quizás combinada con otras funciones).

Ejercicio 1) Definir las siguientes funciones SIN utilizar recursión en forma explícita.

a. **`sum :: [Int] -> Int`**, que dada una lista de números, describe la suma de todos ellos.
b. **`length :: [a] -> Int`**, que dada una lista, describe la longitud de la misma (la cantidad de elementos que tiene).
c. **`reverse :: [a] -> [a]`**, que dada una lista, describe la lista que tiene los mismos elementos, pero en el orden inverso (el último, primero, etc.).

Ejercicio 2) Definir las siguientes funciones SIN utilizar recursión en forma explícita.

a. **`map :: (a -> b) -> [a] -> [b]`**, que dada una función de transformación de elementos y una lista, describe la lista que resulta de aplicar la función a cada uno de los elementos de la lista dada.
b. **`filter :: (a -> Bool) -> [a] -> [a]`**, que dado un predicado sobre elementos y una lista, describe la lista que contiene solamente aquellos elementos de la lista dada que cumplen el predicado.
c. **`find :: (a -> Bool) -> [a] -> Maybe a`**, que dado un predicado sobre elementos y una lista, describe el primer elemento de la lista dada que cumple el predicado, o indica que no lo encontró si ninguno lo cumple.
d. **`any :: (a -> Bool) -> [a] -> Bool`**, que dado un predicado sobre elementos y una lista, indica si alguno de los elementos cumple el predicado.

e. **`all :: (a -> Bool) -> [a] -> Bool`**, que dado un predicado sobre elementos y una lista, indica si todos los elementos de la lista cumplen el predicado.
f. **`countBy :: (a -> Bool) -> [a] -> Int`**, que dado un predicado sobre elementos y una lista, describe la cantidad de elementos de la lista que cumplen con ese predicado.
g. **`takeWhile :: (a -> Bool) -> [a] -> [a]`**, que dado un predicado sobre elementos y una lista, describe la lista que contiene solamente los elementos de la lista dada que cumplen el predicado y aparecen antes que el primer elemento que no lo cumple.
h. **`partition :: (a -> Bool) -> [a] -> ([a], [a])`**, que dado un predicado sobre elementos y una lista, describe un par de listas tal que la primera tiene todos los elementos de la lista dada que cumplen el predicado y la segunda tiene todos los elementos que no lo cumplen.
i. **`scanr :: (a -> b -> b) -> b -> [a] -> [b]`**, que dada una función de acumulación de elementos en un resultado, un resultado inicial, y una lista de elementos, describe la lista de resultados obtenida al acumular los elementos de la lista, desde la derecha, utilizando la función. Por ejemplo:

```
scanR (+) 0 [1,2,3,4,5] = [15, 14, 12, 9, 5, 0]
```

j. **`groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`**, que dada una propiedad que habla de la relación entre dos elementos y lista de esos elementos, describe la lista que resulta de agrupar todos los elementos contiguos que cumplen la propiedad. Por ejemplo:

```
groupBy :: (\n m -> n == m) [1,1,2,2,2,1,1,3,3,2]
  = [[1,1],[2,2,2],[1,1],[3,3],[2]]
```

Ejercicio 3) Definir las siguientes funciones como resultado de utilizar otras funciones (NO debe tomarse la lista como argumento, sino que deben usarse funciones aplicadas parcialmente para devolver el resultado).

a. **`elem :: Eq a => a -> [a] -> Bool`**, que dado un elemento y una lista, indica si el elemento está en la lista dada.
b. **`number :: [a] -> [(Int, a)]`**, que dada una lista, describe el resultado de combinar cada elemento con el número que corresponde a su posición. Por ejemplo:

```
number ['a','b','c'] = [(0,'a'),(1,'b'),(2,'c')]
```

AYUDA: es necesario suponer la existencia de la lista **`nats`**, que describe la lista de número desde `0` en adelante, hasta el número que sea necesario.
c. **`disyuncion :: [Bool] -> Bool`**, que dada una lista de booleanos describe **`True`** si alguno de sus elementos es **`True`**.
d. **`conjuncion :: [Bool] -> Bool`**, que dada una lista de booleanos describe **`True`** si todos sus elementos son **`True`**.
e. **`facturar :: [Helado] -> Int`**, que dada una lista de helados, describe el precio que se debe pagar por ellos.
AYUDA: suponer que existe una función **`precioHelado`**, que dado un helado describe su precio.

f. **`poderDelEntrenador :: [Entrenador] -> Int`**, que dado un entrenador Pokémon (definido en la práctica 2) , describe el poder del mismo. El poder de un entrenador es la suma de los poderes de todos sus Pokémon.

Ejercicio 4) Indicar cuáles de las siguientes expresiones tienen tipo, y para aquellas que lo tengan, decir cuál es ese tipo. DEBE razonarse cada caso, y NO utilizar el intérprete de Haskell para determinarlo (o sea, el objetivo es ayudar a comprender cómo funcionan los esquemas, y usar el intérprete solamente da resultados, no comprensión…).

a. **`filter id`**
b. **`map (\x y z -> (x, y, z))`**
c. **`map (\n m -> n + m)`**
d. **`filter fst`**
e. **`filter (flip const (\n m -> n + m))`**
f. **`map const`**
g. **`map twice`**
h. **`foldr twice`**
i. **`zipWith fst`**
j. **`foldr (\x r z -> (x, z) : r z) (const [])`**

Ejercicio 5) Definir las siguientes funciones SIN utilizar recursión en forma explícita.
AYUDA: estas funciones requieren recursión primitiva.

a. **`last :: [a] -> a`**, que dada una lista no vacía, describe el elemento final de la misma. Falla si la lista está vacía.
b. **`maximum :: Ord a => [a] -> a`**, que dada una lista no vacía de elementos ordenables, describe el más grande de ellos. Falla si la lista está vacía.
c. **`minimum :: Ord a => [a] -> a`**, que dada una lista no vacía de elementos ordenables, describe el más chico de ellos. Falla si la lista está vacía.
d. **`dropWhile :: (a -> Bool) -> [a] -> [a]`**, que dado un predicado sobre elementos y una lista, describe la lista que contiene solamente los elementos de la lista dada que quedan después de eliminar a todos los que cumplen el predicado y aparecen antes que el primer elemento que no lo cumple.
e. **`agregar :: Eq a => a -> [a] -> [a]`**, que dado un elemento y una lista, lo agrega a la misma siempre que aún no esté en ella. Por ejemplo:

```
agregar 2 [3,4] = [3,4,2]
agregar 4 [3,4,2] = [3,4,2]
```

f. **`insert :: Ord a => a -> [a] -> [a]`**, que dado un elemento y una lista, y suponiendo que la lista dada está ordenada de menor a mayor, describe la lista que resulta de insertar el elemento dado en el lugar que le corresponde según el orden (o sea, la lista resultante también estaría ordenada).
g. **`intersperse :: a -> [a] -> [a]`**, que dado un elemento y una lista, describe el resultado de agregar el elemento dado entre cada par de

elementos de la lista dada. Por ejemplo:

```
intersperse 0 [1,2,3,4] = [1,0,2,0,3,0,4]
```

Ejercicio 6) Definir las siguientes funciones SIN utilizar recursión en forma explícita.

a. `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`, que dada una función que combina elementos de dos tipos, y dos listas, cada una con elementos de los tipos correspondientes, describe la lista que resulta de usar la función con cada par de elementos de las listas dadas, en orden. La lista resultante debe ser de la longitud de la más corta de las listas de entrada.

b. `unir :: Eq a => [a] -> [a] -> [a]`, que dadas dos listas de elementos, describe el resultado de agregar a la 2da lista todos los elementos de la primera lista que aún no están en ella. Por ejemplo:

```
unir [4,2,7] [6,7,8] = [6,7,8,2,4]
```

SUGERENCIA: usar recursión estructural en la primera de las listas y utilizar alguna función anteriormente definida en cada paso.

c. `elemAt :: Int -> [a] -> a`, que dado un número `i` y una lista, describe el elemento que se encuentra en la posición `i` de la lista (el primer elemento es el de índice 0). Falla si el número no indica una posición dentro de la lista.

d. `take :: Int -> [a] -> [a]`, que dado un número `n` y una lista, describe la lista de los primeros `n` elementos de la lista dada.

e. `drop :: Int -> [a] -> [a]`, que dado un número `n` y una lista, describe la lista de los elementos que quedan después de descartar los primeros `n` elementos de la lista dada.

Ejercicio 7) Definir las siguientes funciones SIN utilizar recursión en forma explícita.

a. `gustos :: [Helado] -> [Gusto]`, que dada una lista de helados, describe la lista de todos los gustos dados, sin repetir ninguno. Se puede suponer que los gustos se pueden comparar por igualdad (o sea, `Eq Gusto` es válido).

SUGERENCIA: definir una función que dado un helado devuelva la lista de sus gustos.

b. `llamar :: Nombre -> [Persona] -> Maybe Persona`, que dado un nombre y una lista de personas, describe a la persona con ese nombre, si está en la lista, o nada si no está.

c. `elMasViejo :: [Persona] -> Persona`, que dada una lista no vacía de personas describe a la persona más vieja de la lista. Falla si la lista está vacía.

Ejercicio 1)

```
sum = foldr (+) 0

length = foldr (\x n -> n+1) 0

reverse = foldr (\x rs -> rs ++ [x]) []
```

Ejercicio 2)

```
map f = foldr (\x xs -> f x : xs) []

filter p = foldr (\x xs -> if p x then x : xs else xs)

find p = foldr (\x mr -> if p x then Just x else mr) Nothing

any p = foldr (\x b -> p x || b) False

all p = foldr (\x b -> p x && b) True

countBy p = foldr (\x n -> if p x then 1 + n else n) 0

takeWhile p = foldr (\x xs -> if p x then x : xs else []) []

partition p = foldr (\x (bs, ms) -> if p x
                                    then (x:bs, ms)
                                    else (bs, x:ms) ([], [])

scanr f z = foldr (\x (r:rs) -> f x r : r : rs) [z]

groupBy p = foldr (\x gs ->
                    case gs of
                      []          -> [[x]]
                      (y:ys):gs' -> if p x y
                                     then (x:y:ys) : gs'
                                     else [x] : (y:ys) : gs')
                  []
```

Ejercicio 3)

```
elem e = any (\x -> x==e)

number = zipWith (\i x -> (i,x)) nats

disyuncion = any id

conjuncion = all id

facturar = sum . map precioHelado
```

```
poderDelEntrenador =  sum . map poder . pokemonDe
```

Ejercicio 4)

```
filter id :: [Bool] -> [Bool]

map (\x y z -> (x, y, z))

map (\n m -> n + m) :: [Int] -> [Int->Int]

filter fst :: [(Bool, a)] -> [(Bool, a)]

filter (flip const (+)) :: [Bool] -> [Bool]

map const :: [a] -> [b->a]

map twice :: [a->a] -> [a->a]

foldr twice :: a -> [a->a] -> a

zipWith fst :: [(a -> b,c)] -> [a] -> [b]

foldr (\x r z -> (x, z) : r z) (const []) :: [a] -> b -> [(a,b)]
```

Ejercicio 5)

```
last = recr (error "No está") (\x xs y -> if null xs then y else x)

maximum = recr (error "No hay elementos")
               (\x xs m -> if null xs then x else max x m)

minimum = recr (error "No hay elementos")
               (\x xs m -> if null xs then x else min x m)

dropWhile p = recr [] (\x xs rs -> if p x then rs else x:xs)

agregar e = recr [e] (\x xs xes -> if e==x then x:xs else x:xes)

insert e = recr [e] (\x xs xes -> if e<x then e:x:xs else x:xes)

intersperse e = recr [] (\x xs xes -> if null xs
                                      then [x]
                                      else x:e:xes)
```

Ejercicio 6)

```
zipWith f = foldr (\x h ys -> case ys of
                                [] -> []
                                (y:ys') -> f x y : h ys'
                  ) []
```

```
unir = foldr (\x h ys -> agregar x (h ys)) (\ys -> ys)

elemAt = flip (foldr (\x h n -> if n==0 then x else h (n-1))
                     (error "El índice cae fuera de la lista."))

take = flip (foldr (\x h n -> if n == 0 then [] else x : h (n-1))
                   (\_ -> []))

drop = flip (recr (\_ -> [])
                  (\x xs h n -> if n == 0 then xs else h (n-1)))
```

Ejercicio 7)

```
gustos = foldr (\h gs -> let newgs = gustos h
                         in unir newgs gs) []

gustos (Vasito g) = [g]
gustos (Cucurucho g1 g2) = [g1, g2]
gustos (Pote g1 g2 g3) = [g1, g2, g3]

llamar n = foldr (\p mp -> if nombre p == n then Just p else mp)
                 Nothing

elMasViejo = recr (error "No hay personas")
                  (\p ps v -> if null ps then p
                                         else if edad p > edad v
                                               then p else v)
```