Práctica de ejercicios #4 - Orden superior

Programación Funcional, Universidad Nacional de Hurlingham

22 de abril 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.*

Ejercicio 1) Reescribir las siguientes definiciones usando expresiones lambdas, y sin parámetros del lado izquierdo.

a. `succ x = x+1`
b. `doble x = x+x`
c. `cuadruple x = 4*x`
d. `sumaDos x = x+2`
e. `const x y = x`
f. `flip f x y = f y x`
g. `(f . g) x = f (g x)`
   OBSERVACIÓN: usar sección de operadores

Ejercicio 2) Definir las siguientes funciones:

a. `curry :: ((a,b) -> c) -> a -> b -> c`
b. `uncurry :: (a -> b -> c) -> (a,b) -> c`
   OBSERVACIÓN: la función argumento de **`curry`** es NO currificada y el resultado es una función currificada. Y al revés para **`uncurry`**.

Ejercicio 3) Comprobar que las siguientes igualdades son ciertas.

a. `pendiente (suma 2) = \n -> 1`

b. `pendiente (\x -> x^2) = \x -> 2*x+1`

Ejercicio 4) Definir las siguientes funciones usando la función **`consSiCumple`** dada en la clase teórica.

a. **`pares :: [Int] -> [Int]`**, que dada una lista de números, describe la lista que resulta de tomar solamente los números pares de la lista dada.

b. `novato :: [Entrenador] -> [Entrenador]`, que dada una lista de entrenadores Pokémon, describe la lista que contiene solamente a los entrenadores novatos.
NOTA: un entrenador es novato si no tiene Pokémon.
SUGERENCIA: usar funciones auxiliares…

Ejercicio 5) Dada la siguiente definición

```
opuesta :: (a->a->a) -> (a->a) -> a -> a -> a
opuesta f op x y = op (f (op x) (op y))
```

comprobar las siguientes equivalencias:

a. `opuesta max neg = min`

b. `opuesta min neg = max`

c. `opuesta (++) reverse = flip (++)`

Ejercicio 6) Definir las siguientes funciones:

a. `twice :: (a->a) -> (a->a)`, que dados una función y un valor, describe el resultado de aplicar la función al valor dos veces (o sea, al resultado de usar la función con el valor, volver a aplicarle la función).
b. `many :: Int -> (a->a) -> (a -> a)`, que dados un número, una función y un valor, describe el resultado de aplicar la función dada al valor dado tantas veces como indica el número dado.
AYUDA: usar recursión sobre el número.

Ejercicio 7) Usando reducción, comprobar el resultado de

a. `twice succ 10`
b. `twice twice succ 10`
c. `twice twice twice succ 10`
d. `many 2 succ 10`
e. `(many 3 twice) succ 10`

Ejercicio 8) ¿Por qué en el ejercicio anterior `twice` puede tener 3 o 4 argumentos, si en la definición solamente tiene 2 parámetros?

Ejercicio 9) Dadas las siguientes definiciones de tipos,

```
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
data ExpA = Valor Int
          | Sum ExpA ExpA
          | Prod ExpA ExpA
          | Neg ExpA
```

```
data Persona = P String String Int
             -- Nombre DNI    Edad
```

dar el tipo de las siguientes expresiones

a. `EmptyT`
b. `NodeT`
c. `Valor`
d. `Sum`
e. `Sum (Valor 2)`
f. `P`
g. `P "Fidel"`

Ejercicio 10) Definir las siguientes funciones sobre `Set a`

a. `list2set :: Eq a => [a] -> Set a`
b. `intersection :: Set a -> Set a -> Set a`
c. `complement :: Set a -> Set a`
   OBSERVACIÓN: el complemento tiene solamente aquellos elementos del tipo que NO están en el conjunto dado como argumento.
d. `difference :: Set a -> Set a -> Set a`
   OBSERVACIÓN: la diferencia simétrica solamente tiene los elementos que están solamente en uno de los 2 conjuntos, pero no en ambos.
e. `pares :: Set Int`, que describe el conjunto de todos los números pares.

Ejercicio 11) Definir los siguientes esquemas de funciones sobre listas, en forma similar a como se definió la función `map` en la teoría:

a. `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
b. `zipApply :: [a->b] -> [a] -> [b]`
c. `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.
d. `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.
e. `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.
f. `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.

g. `any :: (a -> Bool) -> [a] -> Bool`, que dado un predicado sobre elementos y una lista, indica si alguno de los elementos cumple el predicado.
h. `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.
i. `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.
j. `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.
k. `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]
```

Ejercicio 12) Definir las siguientes funciones utilizando solamente esquemas sobre listas y aplicación parcial

a. `sueldos :: [Empleado] -> [Int]`, que describe la lista de los sueldos de una lista de empleados. Se supone definido un tipo `Empleado`, y una función

```
sueldo :: Empleado -> Int
```

b. `gestionarSueldos :: [[Empleado]] -> [[Int]]`, que describe las listas de sueldos de una lista de empresas, donde se representa a una empresa como `[Persona]`. Intentar hacerlo de dos formas: sin utilizar la función `sueldos` y utilizándola.
c. `sinFaltas :: [Empleado] -> [Empleado]`, que describe la lista que resulta de filtrar la lista dada para quedarse solamente con las personas que no faltaron nunca. Se supone definida una función

```
nroFaltas :: Empleado -> Int
```

d. `sinVacias :: [[Caramelo]] -> [[Caramelo]]`, que dada una lista de bolsas de caramelos, describe la lista que resulta de remover todas las bolsas vacías de la lista original. Se supone que una bolsa de caramelos se representa con el tipo `[Caramelo]` y que hay definido un tipo `Caramelo` para representar caramelos.
e. `sumatoriaDe :: (Int -> Int) -> [Int] -> Int`, que describe el resultado de realizar la sumatoria de todos los resultados de usar la función dada en cada elemento de la lista dada, o sea,

```
sumatoriaDe f [x1,…,xn] = f x1 + … + f xn
```

Para realizarla, utilizar la función `repetir` de la práctica 2, y algún esquema de los vistos en esta práctica.

Ejercicio 13) Definir las siguientes funciones utilizando solamente esquemas sobre listas y aplicación parcial

a. **`armarVasitos :: [Gusto] -> [Helado]`**, que dada una lista de gustos de helado, describe la lista de vasitos de helado con los gustos correspondientes.
b. **`usarTodas :: [Int->Int] -> Int -> [Int]`**, que dada una lista de funciones de números en números, y un número base, describe la lista de resultados de usar cada función con el número dado. Por ejemplo,

```
usarTodas [ succ, doble, sqr, cuadruple ] 3
     = [ 4, 6, 9, 12 ]
```

c. **`aplicarACada :: [Int->Int] -> [Int] -> [Int]`**, que dada una lista de funciones y una lista de números, describe la lista de resultados correspondientes a aplicar la primera función al primer número, la segunda función al segundo número, etc. Ignora las funciones o números que no tengan un elemento correspondiente en la otra lista.
d. **`danCero :: [Int->Int] -> [Int->Int]`**, que dada una lista de funciones, describe la lista de aquellas funciones de las dadas que dan cero cuando se las aplica al número cero. Por ejemplo,

```
danCero [ succ, doble, sqr, \x->x+2 ]
     = [ doble, sqr ]
```

Ejercicio 14) Un multiconjunto es similar a un conjunto, pero los elementos en lugar de simplemente pertenecer o no al conjunto, pueden aparecer una o más veces, o no aparecer (en multiconjuntos se habla de “apariciones” u “ocurrencias”, en lugar de “pertenencia” como en los conjuntos). Así, un cierto elemento puede aparecer 3 veces en el multiconjunto, mientras que otro puede aparecer solamente 1, y los demás ninguna. Dada la siguiente representación para multiconjuntos:

```
data MultiSet a = MS (a -> Int)
```

definir las siguientes funciones sobre multiconjuntos:

a. **`emptyMS :: MultiSet a`**, el multiconjunto sin elementos.
b. **`singleton :: Eq a => a -> MultiSet a`**, el multiconjunto que tiene exactamente una ocurrencia del elemento dado y ningún elemento más (o sea, el elemento dado aparece exactamente 1 vez).
c. **`agregarN :: Eq a => a -> Int -> MultiSet a -> MultiSet a`**, que agregar la cantidad dada apariciones del elemento dado al multiconjunto dado.

d. **`apariciones :: a -> MultiSet a -> Int`**, que indica la cantidad de veces que el elemento aparece en el multiconjunto dado.
e. **`union :: MultiSet a -> MultiSet a -> MultiSet a`**, que dados dos multiconjuntos, calcula la unión de ellos. En la unión de dos multiconjuntos, un elemento aparece tantas veces como en ambos multiconjuntos (por ejemplo, si en el primero aparece 3 veces y en el segundo 2 veces, en el resultado aparece 5 veces).
f. **`interseccion :: MultiSet a -> MultiSet a -> MultiSet a`**, que dados dos multiconjuntos, calcula la intersección de ellos. En la intersección de dos multiconjuntos, un elemento aparece tantas veces como en el multiconjunto que aparece menos veces (por ejemplo, si en el primero aparece 3 veces y en el segundo 2, en el resultado aparece solo 2 veces).