Programación Funcional

Ejercicios de Práctica Nro. 11

Esquemas de funciones I

Aclaraciones:

- Los ejercicios siguen un orden de complejidad creciente, y cada uno puede servir a los siguientes. No se recomienda saltear ejercicios sin consultar antes a un docente.
- Recordar que se pueden aprovechar en todo momento las funciones ya definidas, tanto las de esta misma práctica como las de prácticas anteriores.
- Probar todas las implementaciones, al menos en una consola interactiva.
- Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evalúan principalmente este aspecto. Para utilizando formas alternativas al resolver los ejercicios consultar a los docentes.

Ejercicio 1) Definir las siguientes funciones utilizando recursión estructural explícita sobre **`Pizza`**:

a. **`cantidadCapasQueCumplen`**
   `:: (Ingrediente -> Bool) -> Pizza -> Int`
b. **`conCapasTransformadas`**
   `:: (Ingrediente -> Ingrediente) -> Pizza -> Pizza`
c. **`soloLasCapasQue`**
   `:: (Ingrediente -> Bool) -> Pizza -> Pizza`

Ejercicio 2) Definir las siguientes funciones utilizando alguna de las definiciones anteriores:

a. **`sinLactosa :: Pizza -> Pizza`**
b. **`aptaIntolerantesLactosa :: Pizza -> Bool`**
c. **`cantidadDeQueso :: Pizza -> Int`**
d. **`conElDobleDeAceitunas :: Pizza -> Pizza`**

Ejercicio 3) Definir,
`pizzaProcesada :: (Ingrediente -> b -> b) -> b -> Pizza -> b`, que expresa la definición de fold para la estructura de **`Pizza`**.

Ejercicio 4) Resolver todas las funciones de los puntos 1) y 2) utilizando la función **`pizzaProcesada`**.

Ejercicio 5) Resolver las siguientes funciones utilizando **`pizzaProcesada`** (si resulta demasiado complejo resolverlas, dar primero una definición por recursión estructural explícita, y usar la técnica de los “recuadros”):

a. **`cantidadAceitunas :: Pizza -> Int`**
b. **`capasQueCumplen`**
   `:: (Ingrediente -> Bool) -> Pizza -> [Ingrediente]`

c. `conDescripcionMejorada :: Pizza -> Pizza`
d. `conCapasDe :: Pizza -> Pizza -> Pizza`, que agrega las capas de la primera pizza sobre la segunda
e. `primerasNCapas :: Int -> Pizza -> Pizza`

Ejercicio 6) Demostrar las siguientes propiedades sobre el tipo `Pizza`, utilizando las definiciones por recursión estructural explícita para cada función:

a. para todo **f**. `length . capasQueCumplen f = cantidadDe f`

b. para todo **f**. para todo p_1. para todo p_2.
```
cantidadCapasQueCumplen f (conCapasDe p1 p2)
  = cantidadCapasQueCumplen f p1 +
    cantidadCapasQueCumplen f p2
```

c. para todo **f**. para todo p_1. para todo p_2.
```
conCapasTransformadas f (conCapasDe p1 p2)
  = conCapasDe (conCapasTransformadas f p1)
               (conCapasTransformadas f p2)
```

d. para todo **f**. `cantidadCapasQueCumplen f . soloLasCapasQue f`
`= cantidadCapasQueCumplen f`

Ejercicio 7) Definir las siguientes funciones de esquemas sobre listas, utilizando recursión estructural de forma explícita:

a. `map :: (a -> b) -> [a] -> [b]`
b. `filter :: (a -> Bool) -> [a] -> [a]`
c. `foldr :: (a -> b -> b) -> b -> [a] -> b`
d. `recr :: b -> (a -> [a] -> b -> b) -> [a] -> b`
e. `foldr1 :: (a -> a -> a) -> [a] -> a`
f. `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
g. **(Desafío)** `scanr :: (a -> b -> b) -> b -> [a] -> [b]`

Ejercicio 8) Demostrar las siguientes propiedades utilizando las definiciones anteriores (y algunas otras de prácticas anteriores):

a. para todo f. para todo g. `map f . map g = map (f . g)`
b. para todo f. para todo xs. para todo ys.
   `map f (xs ++ ys) = map f xs ++ map f ys`
c. para todo f. `concat . map (map f) = map f . concat`
d. `foldr ((+) . suma') 0 = sum . map suma'`
e. para todo f. para todo z. `foldr f z . foldr (:) [] = foldr f z`
f. para todo f. para todo z. para todo xs. para todo ys.
   `foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs`
g. `(+1) . foldr (+) 0 = foldr (+) 1`
h. para todo n. para todo f. `many n f = foldr (.) id (replicate n f)`
   siendo `many 0 f = id`
   `many n f = f . many (n - 1)`
i. para todo f. para todo xs. para todo ys.
   `zipWith (f . swap) xs ys`
   `= map (uncurry f) (flip zip xs ys)`
j. **(Desafío)**
   para todo f. para todo g. para todo h. para todo z.
   si para todo x. para todo y. `h (f x y) = g x (h y)`
   entonces `h . foldr f z = foldr g (h z)`
   Dar un ejemplo de uso específico de esta propiedad.

Ejercicio 9) Definir las siguientes funciones utilizando solamente `foldr`:

a. `sum :: [Int] -> Int`
b. `length :: a -> Int`
c. `map :: (a -> b) -> [a] -> [b]`
d. `filter :: (a -> Bool) -> [a] -> [a]`
e. `find :: (a -> Bool) -> [a] -> Maybe a`
f. `any :: (a -> Bool) -> [a] -> Bool`
g. `all :: (a -> Bool) -> [a] -> Bool`
h. `countBy :: (a -> Bool) -> [a] -> Int`
i. `partition :: (a -> Bool) -> [a] -> ([a], [a])`
j. `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
k. `scanr :: (a -> b -> b) -> b -> [a] -> [b]`
l. `takeWhile :: (a -> Bool) -> [a] -> [a]`
m. `take :: Int -> [a] -> [a]`
n. `drop :: Int -> [a] -> [a]`
o. `(!!) :: Int -> [a] -> a`

Ejercicio 10) Indicar cuáles de las siguientes expresiones tienen tipo, y para aquellas que lo tengan, decir cuál es ese tipo:

a. `filter id`
b. `map (\x y z -> (x, y, z))`
c. `map (+)`
d. `filter fst`
e. `filter (flip const (+))`
f. `map const`

g. `map twice`
h. `foldr twice`
i. `zipWith fst`
j. `foldr (\x r z -> (x, z) : r z) (const [])`