Funcional 4: Restricciones de tipos, Polimorfismo

Nota Previa: Restricciones de tipos

¿Qué es una restricción de tipos? Se puede pensar como una condición que tiene que cumplir un tipo para que las expresiones de ese tipo sean válidas en un contexto.
Eq **no** es un tipo, es una condición. Algunos tipos (p.ej. todos los tipos numéricos, las listas de cualquier tipo numérico y los booleanos) sí cumplen la condición, otros (p.ej. las funciones) no. Entonces si una función tiene el tipo

Eq a => a -> a

puedo pasarle como parámetro un número, una lista de números o un booleano como parámetros (porque su tipo cumple con la condición Eq) pero no una función (porque su tipo no cumple con la condición).

Las restricciones de tipos que usamos son:

Eq	los tipos que puedo comparar por ==
Ord	los tipos que tienen semántica de mayor y menor, o sea puedo aplicarles <, >, <=, >=
Num	los tipos cuyas expresiones representan números, o sea puedo sumarlos, restarlos, etc.

Ejercicio extra

Para esta práctica también obtengan el tipo de cada función que definan, para esto utilicen :t.

Ej: Si quiero saber el tipo de la función **elem/**2 puedo hacer lo siguiente en Haskell.

```
Hugs> :t elem
elem :: Eq a => a -> [a] -> Bool
```

Ahora sí los ejercicios

1.1. Calcular el tipo de las siguientes expresiones

```
f1 a1 a2 a3    = a1 a3      == a2 a3
f2 a1 a2 a3    = a1 (a3 + 1)  == a2 (a3 + 1)
f3 a1 a2 a3    = (a1 a3) + 1  == (a2 a3) + 1
f4 a1        = length a1 + 4
f5 a1        = head a1 + 4
f6 a1 a2 a3    = (a1 a2) + (a1 a3)
```

1.2. Calcular el tipo de las siguientes funciones

```
fCondicional f1 f2 f3 x
            | f1 x          = f2 x
            | otherwise     = f3 x

transformarCond f1 f2 f3 l = map (fCondicional f1 f2 f3) l
```

Pongan especial interés en las variables de tipo (p.ej. la a en "a -> Bool"), dense cuenta cuáles tienen que ser la misma letra y en qué casos van letras distintas (la diferencia entre "a -> a" y "a -> b").

1.2.1. Implementar la función **llevarAPares/**1, que recibe una lista de números y devuelve otra que lleva los impares al número siguiente (para hacerlo par) y deja los pares intactos. Usar la función del ejercicio anterior, **transformarCond**. P.ej.

```
Main> llevarAPares [3,6,9,7,2]
   [4,6,10,8,2]
```

Ayuda: investigar la función id.

1.2.2. Retomando el ejercicio 3.4 de la guía 2 funcional (Parte Funciones de Orden Superior con Manejo de Listas) implementar la función **separarNotas** , que dada la información de un curso devuelve una lista en la cúal de los que aprobaron todos los parciales deja la nota más alta y de los que no aprobaron alguno la más baja. P.ej.

```
Main> separarNotas [[8,6,2,4],[7,9,4,5],[6,2,4,2],[9,6,7,10]]
[2,9,2,10]
```

Nota: Usar **transformarCond** y las funciones definidas ejercicio 3.4 de la guía 2 que les sirvan.

1.2.3. Siguiendo con el mismo dominio, implementar la función textosFinales, que dada la información de un curso devuelva una lista con el string "aprobo" para los que aprobaron, y "no aprobo" para los que no. P.ej.

```
Main>  textosFinales [[8,6,2,4],[7,9,4,5],[6,2,4,2],[9,6,7,10]]
["no aprobo","aprobo","no aprobo","aprobo"]
```

Nota: Usar transformarCond y las funciones definidas ejercicio 3.4 de la guía 2 que les sirvan.

1.3.1. Definir el tipo de la siguiente función **fPrincipal** y **fAuxiliar**, justificando la decisión.

```
fAuxiliar a [] = []
fAuxiliar a (x:xs) | a x = fAuxiliar a xs
                   | otherwise = x : fAuxiliar a xs

fPrincipal a b xs = fAuxiliar a ( fAuxiliar b xs)
```

1.3.2. Mostrar un ejemplo de invocación y respuesta de la función fPrincipal.

1.4.1. Definir la función **compararListas**, que dada una función y 2 listas de valores devuelva la lista "mayor", en base a la comparación de distintos resultados de la función aplicada a cada elemento. En caso de empate devolver cualquiera de las dos. Si la longitud de ambas listas difieren comparar los elementos en base a la lista más corta.

```
Main> compararListas (^2) [1,3,1] [-2,0,2]
[-2,0,2]
```

Ya que gana en mas oportunidades que la primera.

```
Main> compararListas length ["abrac", "adabra"] ["pata", "de", "cabra"]
["abrac", "adabra"]
```

Se toman las primeras 2 comparaciones.

1.4.2. Definir el tipo de la función **compararListas**.

1.5.1. Definir la función **fCompone** que recibe una función f, una lista de funciones gs y 2 valores n m y devuelve la lista con las funciones de gs cada una de las cuales, f compuesta con ella aplicada en n da por resultado m.

Ej:

fCompone (+1) [(*1),(+1),((-)1)] 1 2 = [(*1)]
fCompone (* 1) [(*1),(+1),((-)1)] 1 2 = [(+1)]

1.5.2. Definir el tipo de la función **fCompone**.

1.6. Dada la siguiente función:

```
funcion x y z = (map y . filter x) z
```

1.6.1. Definir el tipo de la misma.

1.6.2. Dada una lista de productos:

Cada tupla representa un producto (nombreProducto, marca, precio, peso).

productos = [("yogur", "sancor", 3, 200), ("queso", "pirulo", 15, 200), ("yerba", "elNorte", 5, 500), ("aceite", "natura", 8, 500), ("café", "laEstancia", 7, 250), ("leche", "laSerenisima", 3, 1000)]

marcasConocidas = ["laSerenisima", "sancor", "natura"]

Utilizando la función dada, definir las siguiente funciones y todas sus auxiliares:

1.6.2.1. productosAccesibles. Devuelve una lista de tuplas (nombreProducto , marca), de todos aquellos productos que tengan un precio menor a 6.

1.6.2.2. productosConocidos. Devuelve una lista de tuplas (nombreProducto, marca), de todos aquellos productos que sean de una marca conocida.

1.6.2.3. productosMenorPeso. Devuelve una lista de tuplas (nombreProducto, marca), de todos aquellos productos que pesen menos de 300 gramos.

1.7. Dada una lista de empleados, con el siguiente formato.

empleados = [("martin", 23, 2005, ["creatividad", "rapidezParaAprender"]), ("sofia", 25, 2004, ["prolijidad", "innovacion", "solidaridad"]), ("emilio", 24, 2003, ["prolijidad", "trabajoEnEquipo","creatividad"]), ("marcelo", 21, 2004, ["sociabilidad"]), ("ana", 25, 2007, ["facilidadDeExpresion", "liderazgo"])].

Cada tupla tiene este formato (nombre, edad, año de Ingreso, puntosFuertes).

1.7.1. Definir una única función **ordenar** que permita ordenar los empleados de acuerdo a los siguiente criterios:
Ordenar los empleados por año de ingreso a la compañía.
Ordenar los empleados por edad.
Ordenar los empleados por la cantidad de puntos fuertes, de mayor a menor.

1.7.2. Definir el tipo de la función
Nota: La función ordenar debe ser una función genérica y única que me permita ordenar a la lista de empleados por cualquiera de los criterios.

Funcional: Expresiones Lambda

Nota Previa:

Permite definir y utilizar funciones sin darles un nombre explícitamente mediante una expresión lambda.

Por Ej: La función cuadrado en Expresión Lambda es la siguiente.

λ x : x * x

Pasado a Haskell, tengo:

(\x -> x * x)

Y lo ejecuto así:

```
Hugs> (\x -> x * x) 2
4
```

Si quiero calcular el producto entre 2 números utilizando expresiones lambda, puedo hacer.

```
Hugs> (\x y -> x * y) 2 3
6
```

Ahora sí los ejercicios

2.1. Utilizando expresiones lambda, definir la función **intersect**, que dada 2 listas de elemento devuelve los elementos comunes a las 2 listas.

Ayuda: Completar la siguiente definición de **intersect**.

```
intersect l1 l2 = filter (\x -> ...) l1
```

2.2. Definir la función **diferencia/**2 recibe dos listas y devuelve los elementos de la primera que no están en la segunda.

Completar la siguiente definición de **diferencia**

```
diferencia l1 l2 = filter (\x -> ...) l1
```

2.3. Definir la función es **hayAlgunMultiploDe**/2, recibe un n y una lista, devuelve verdadero si es el n es múltiplo de alguno de los elementos de la lista.

Completar la siguiente definición de **hayAlgunMultiploDe**

```
hayAlgunMultiplo n l1 = any (\x -> ...) l1
```

2.4. En una local de frutas y verduras se conoce el total en kilos de cada uno de los productos que llegan para vender, se tienen la siguiente información.

productos = [("manzana", 50), ("banana", 30), ("naranja", 40), ("papa", 40), ("tomate", 25)]

Nota: productos es una función constante.

2.4.1. Definir la función **totalKilosProductos**, me devuelve la cantidad total de kilos de productos.

```
Main> totalKilosProductos
185
```

Nota: a) Resolverlo utilizando foldl + funciones lambda.
b) Resolverlo utilizando map + funciones lambda.