Parcial 1 - Algoritmos I Taller: Tema F

Ejercicio 1

Se van a implementar algunos aspectos del juego (muy) simplificado de cartas *Magic* en Haskell. A continuación les dejamos una descripción del juego en el que está inspirado el ejercicio.

> **Magic:** The Gathering y frecuentemente abreviado como Magic, es un juego de cartas coleccionables diseñado en 1993 por Richard Garfield, profesor de matemáticas.
>
> Cada partida de Magic representa una batalla entre poderosos magos (en el juego conocidos como planeswalkers), en el que cada uno de estos es uno de los jugadores de la partida. Los jugadores pueden usar hechizos (conjuros, artefactos, tierras, criaturas fantásticas, etc.), representados individualmente en cada carta, para derrotar a sus oponentes. De este modo, el concepto original del juego se inspira de forma notable en los duelos de magos típicos de los juegos de rol tradicionales, como Dungeons & Dragons. La estructura del juego reemplaza los útiles usados en los juegos de aventura de papel y lápiz por una gran cantidad de cartas y unas reglas más complejas que la mayoría de otros juegos de cartas.

a) Definir el tipo `Color` que consta de los constructores `Rojo, Verde, Azul, Negro` y `Blanco`. Los constructores no toman parámetros. **El tipo `Color` no debe estar en la clase `Eq`**. Luego programa la función usando *pattern matching*:

```
mismoColor :: Color -> Color -> Bool
```

que dados dos valores `c1` y `c2` del tipo `Color` debe devolver `True` cuando `c1` y `c2` son el mismo color (se construyen con el mismo constructor) y `False` en caso contrario.

Si se usan más de seis casos, este apartado sumará menos puntaje.

b) Definir el tipo `CartaMagic` que representa una carta de *Magic*. Tiene constructores:

- Constructor `CartaDeTerreno`: Toma dos parámetros, el primero de tipo `String` (representando el `Nombre`) y el segundo de tipo `Color`
- Constructor `CartaDeCriatura`: Toma cuatro parámetros, el primero de tipo `String` (representando el `Nombre`), el segundo es un `Int` (representando el `Costo` de utilizar la carta), y los dos últimos parámetros son `Daño (Dano)` y `Resistencia`, también de tipo `Int`.

Es conveniente definir sinónimos para los tipos `Nombre`, `Costo`, `Dano` y `Resistencia`.

c) Programar la función Cuánto Daño:

```
cuantoDano :: CartaMagic -> Int
```

teniendo en cuenta que el `cuantoDano` de una carta de terreno sera 0.

d) Incluir el tipo `CartaMagic` en la clase `Ord` de manera tal que una carta se considere mayor que otra si el resultado de la función `cuantoDano` es más grande.

Ejercicio 2

a) Programar de manera recursiva la función

```
soloTerreno :: [CartaMagic] -> Color -> [Nombre]
```

que dada una lista de cartas `ns` y un color `c` devuelve una lista con los nombres de las cartas de terreno con color c.

b) Escribir una lista de cartas magic con al menos tres elementos, donde al menos uno de ellos debe ser una carta de terreno, y otro debe ser una carta de criatura.

c) Escribir el resultado de `soloTerreno` para la lista del punto b)

Ejercicio 3

Basados en el tipo `ListaAsoc` del *Proyecto 2*, programar la función:

```
la_menores :: ListaAsoc a b -> b -> ListaAsoc a b
```

que dada una lista de asociaciones `la` y un dato `x` devuelve una nueva lista de asociaciones con las asociaciones de `la` cuyos valores son menores que `x`. Completar el tipado de la función para incluir los *type clases* necesarios para programarla.

Ejercicio 4*

a) Programar la función

```
a_ord :: Ord a => a -> Arbol a -> Bool
```

Que se comporta de la siguiente manera: dado un árbol `en er = Rama` ar_i `n` ar_d , donde n es la raíz, y ar_i es el árbol izquierdo y ar_d es el árbol derecho de `er` . La función da True si y sólo si todos los elementos de ar_i son menores o iguales a `n` y además todos los elementos de ar_d son mayores o iguales a `n`.

b) Inventar un ejemplo de uso de la función creando un árbol con al menos 3 elementos

c) Escribir el resultado de la función aplicada al ejemplo del inciso b)