Parcial 1 - Algoritmos I Taller: Tema D

Ejercicio 1

En las siguientes preguntas marque la respuesta correcta.

a) Si tengo una función con la siguiente declaración de tipos de la función `f`
`f :: Ord a => a -> [a] -> a` puedo decir que:
   1) Es una función polimórfica Paramétrica
   2) Es un función polimórfica Ad hoc
   3) Es una función recursiva
   4) es un constructor
   5) ninguna de las anteriores.

b) Si tengo una función con la siguiente declaración

```
f :: a -> [a] -> [a]
f y [] = []
f y (x:xs) | y == x = x:(f y xs)
           | otherwise = f y xs
```

puedo decir que:
   1) La definición de la función es incorrecta, debería utilizar pattern matching.
   2) Es un función polimórfica Ad hoc
   3) La declaración de tipos está mal, porque debería incluir una clase de tipos.
   4) es un constructor
   5) ninguna de las anteriores.

c) Dada la siguiente declaración de función en Haskell:

```
pegar :: Maybe Char -> Maybe Char -> Maybe String
pegar Nothing Nothing = Nothing
pegar Nothing _ = Just ""
pegar _ Nothing = Just ""
pegar (Just a) (Just b) = Just (a:[b])
```

¿Cuál es el propósito de la función `pegar` y cómo maneja el valor `Nothing`?
   1) La función `pegar` lanza un error cuando el segundo elemento es `Nothing`
   2) La función `pegar` lanza un error cuando se le pasa `Nothing` en algún parámetro.
   3) La función `pegar 'a' 'b'` devuelve `Just "ab"`,y `pegar Nothing _` devuelve `Nothing`.
   4) La función `pegar` solo funciona con valores Nothing y no con Just.
   5) ninguna de las anteriores.

d) Dada la siguiente declaración de tipo en Haskell:

```
data Pila a = Descargada | Contiene a (Pila a)
```

Puedo afirmar que :

1) El tipo `Pila` tiene dos constructores, uno sin parámetros y el otro constructor con un parámetros.
2) No se puede definir un tipo de esa manera.
3) El tipo está mal definido porque ambos constructores no toman parámetros.
4) Ese tipo está mal definido, debería haber utilizado el comando `type`.
5) ninguna de las anteriores.

Ejercicio 2

Se van a representar los elementos de la heladera usando tipos en Haskell. Los productos que tendremos en cuenta son: `Leche, Carne, Queso`. La idea es poder detallar para tipo de producto, las características más importantes. En tal sentido identificamos las siguientes características de cada uno de estos productos a tener en cuenta:

`Leche`

- `TipoDeLeche`, que es un tipo enumerado con las siguientes opciones: `Descremada, Entera, Condenzada, Polvo`
- `UsoDeLeche`, que es un tipo enumerado con las siguientes opciones: `Bebida, Preparaciones`
- `Precio`, que es un sinónimo de `Int` indicando el precio

`Carne`

- `Corte,` que es un tipo enumerado con las siguientes opciones: `Bife, Molida, Pulpa`
- `Peso`, que es un sinónimo de `Float` indicando el largo de la vigueta
- `Precio`, que es un sinónimo de `Int` indicando el precio

`Queso`

- `TipoDeQueso`, que es un tipo enumerado con las siguientes opciones: `Barra, Cremoso, Duro .`
- `Peso`, que es un sinónimo de `Float` indicando el largo de la vigueta
- `Precio`, que es un sinónimo de `Int` indicando el precio

Para ello:

a) Definir el tipo `Perecedero` que consta de los constructores `Leche, Carne` y `Queso`, constructores con parámetros descritos arriba (Se deben definir también los tipos enumerados `TipoDeLeche, UsoDeLeche, Corte, TipoDeQueso`, y los sinonimos de `Precio` y de `Peso`). **Los tipos** `Perecedero` **y** `TipoDeQueso` **no deben estar en la clase** `Eq` , ni en la clase `Ord`.

b) Definir la función `cuantosQuesos` de la siguiente manera:

```
cuantosQuesos :: [Perecedero] -> TipoDeQueso -> Int
```

que dada una lista de `Perecedero lp` y un valor `q` de `TipoDeQueso`, me devuelve un entero indicando la cantidad de quesos que hay en `lp` con el de tipo `q`.
NOTA: Dejar como comentario un ejemplo donde hayas probado la función `cuantosQuesos` con una lista con al menos 3 `Perecedero`.

c) Definir igualdad para el tipo de `Perecedero`: de tal manera que, dos elementos de tipo `Leche` son iguales sólo si tienen el mismo **tipo de leche** y el mismo **uso de leche**, dos `Carne` son iguales solo si tienen el mismo **corte**, mientras que dos **quesos** son iguales si

tiene el mismo tipo de queso. Como es de suponer las `Leches`, `Carnes` y `Quesos` son distintos entre sí.
NOTA: Dejar como comentario en el código dos ejemplos en los que probaste la igualdad.

Ejercicio 3

Queremos hacer un programa, para que los de una escuela de fútbol puedan saber si sus alumnos de un categoría pueden pasar al siguiente nivel o no.

a) Definir un tipo recursivo `NotasDelClub`, que permite guardar las notas que tuvo cada alumno de una categoría en el año. El tipo `NotasDelClub`, tendrá dos constructores:

1) `EvolucionDelJugador`, que tiene 6 parámetros:

- `String`, para el nombre y apellido del alumno
- `Division`(tipo enumerado con la `categoría` actual del jugador, `Septima`, `Sexta`, `Quinta y Cuarta`)
- `Nota` (Evalúa la capacidad defensiva del jugador, Int entre 1 y 10)
- `Nota` (Evalúa capacidad de ataque del jugador, Int entre 1 y 10)
- `Nota` (Evalúa la calidad de pases, Int entre 1 y 10)
- `NotasDelClub`, recursión con el resto de las notas.

2) `NoHayMasJugadores`, que es un constructor sin parámetros, similar al de la lista vacía, para indicar que se terminaron las notas.

b) La condición para poder promover a la siguiente división se describen a continuación, basándose en las notas obtenidas:

- Si el jugador está en `Séptima` o `Sexta` división, debe evaluar por sobre 7 en alguna de las capacidades (`Ataque` y/o `Defensa`), y haber realizado Pases con `calidad` de al menos 6.
- Si el jugador está en `Quinta`, debe tener al menos un 7 en cada capacidad, y al menos un 8 calidad de `pase`.

Programar la función `pasaDeDivision`, que toma como primer parámetro `notas` del tipo `NotasDelClub`, y como segundo parámetro `nombre` de tipo `String` y retorna un valor de tipo `Bool`, indicando si el alumno llamado `nombre` **pasa de división o no**.

```
pasaDeDivision :: NotasDelClub -> String -> Bool
```

NOTA: Dejar como comentario un ejemplo donde hayas probado `PasaDeDivision` con un parámetro de tipo `NotasDelClub` que tenga al menos 3 alumnos.

c) Programar la función `devolverDivision` con la siguiente declaración:

```
devolverDivision :: NotasDelClub -> String -> Maybe Division
```

que toma una variable `notas` de tipo `NotasDelClub`, y como segundo argumento un `nombre`, que identifica el alumno, y en caso que el alumno esté en `notas` (con una división `d)`, retorna `Just d` y `Nothing` en caso contrario.

NOTA: Dejar como comentario un ejemplo donde hayas probado la función.