UT4 Repartido de ejercicio

El siguiente texto incluye ejercicios de programación en Haskell. Corresponde a la unidad temática UT4 *Tipos algebraicos* del curso de *"Programación Funcional"* dictado por Ignacio Pacheco y Leonardo Val.

La realización de estos ejercicios es opcional, y no forma parte de ninguna forma de la calificación final del curso. Los docentes se reservan el derecho de corregir, modificar y ampliar el siguiente documento según crean conveniente.

— Tipos `data` —

Tipos enumerados

`nextMonth` & `previousMonth`

El siguiente tipo enumerado `Month` representa los días de la semana.

```
data Month = January | February | March
  | April | May | June | July | August
  | September | October | November
  | December deriving (Eq, Show)
```

Definir las función `nextMonth` y `previousMonth` que tome un `Month`, y retorne el mes siguiente y anterior respectivamente.

Variantes:

1. Agregar un argumento que indique cuantos meses hacia adelante o atrás (respectivamente) se quiere ir. Las funciones como están pedidas serían equivalente a tener este argumento extra en 1.
2. Permitir que el argumento anterior sea negativo.

Tipos monomórficos

`Colour`

Definir un tipo data `Colour` para representar colores. Un caso debe tener las componentes RGB. Deben haber otros casos para: rojo, azul, verde, blanco y negro. Definir la función `toRGB` que convierte cualquier `Colour` al constructor RGB. Por ejemplo:

- `toRGB Green == (RGB 0 255 0)`
- `toRGB White == (RGB 255 255 255)`
- `toRGB (RGB 1 2 3) == (RGB 1 2 3)`

Variantes:

1. Definir las funciones `red`, `green`, `blue` que toman un `Colour` y retornan el valor entero de la componente correspondiente.

`AstroDistance`

Se define un tipo para representar distancias en unidades usadas en astronomía.

```
data AstroDistance
  = AstroUnits Double | LightYears Double
  | Parsecs Double deriving (Eq, Show)
```

Definir las funciones de conversión entre unidades. Por ejemplo:

- `toAstroUnits (LightYears 1.0) = (AstroUnits 63241.0)`
- `toLightYears (Parsecs 1.0) = (LightYears 3.26156)`
- `toParsecs (AstroUnits 1.0) = (Parsecs 4.84815e-6)`
- `toAstroUnits (AstroUnits 2.34) = (AstroUnits 2.34)`
- `toLightYears (LightYears (-1.0)) = (LightYears (-1.0))`
- `toParsecs (Parsecs 1.0) = (Parsecs 1.0)`

Variantes:

1. Definir las funciones `asAstroUnits`, `asLightYears`, `asParsecs`, que convierten un valor `AstroDistance` a la unidad correspondiente, pero retornan el valor `Double`.

Juegos de cartas

`Naipe`

El siguiente tipo data `Naipe` representa las cartas de la baraja española.

```
data Naipe = Oros Int | Copas Int
  | Espadas Int | Bastos Int
  deriving (Show, Eq)
```

Escribir las siguientes definiciones:

- Función `numNaipe` para obtener el número de un naipe.
- Función `suitNaipe` para obtener el palo de un naipe: 1 para oros, 2 para copas, 3 para espadas y 4 para bastos.
- La lista `baraja` con todos los naipes de la baraja.

Tipos recursivos o polimórficos

Árboles binarios

El siguiente tipo `BinTree` representa árboles binarios.

```
data BinTree a = Empty
  | Node a (BinTree a) (BinTree a)
  deriving (Show, Eq)
```

Definir las funciones para los recorridos `preorder`, `inorder`, `postorder`.

Árboles binarios de búsqueda

Dado el tipo `BinTree` definido en un ejercicio anterior, un árbol binario de búsqueda tendría los nodos ordenados.

Definir la función `isSortedBinTree`, que toma un `(BinTree a)` y determina si sus elementos están ordenados. El tipo `a` debe implementar la clase `Ord`.

Definir la función `searchInBinTree`, que toma un `(BinTree a)` y un valor `a` y retorna si el valor está en el árbol o no. El árbol debe suponerse ordenado, y debe tener un orden logarítmico.

Definir la función `insertOrdBinTree`, que toma un `(BinTree a)` y un valor `a`, y retorna un nuevo árbol igual al anterior con el valor insertado de manera ordenada. El tipo `a` debe implementar la clase `Ord`.

— Clases de tipos —

Definición de instancias

Clases para `Colour`

Definir una instancia de la clase `Eq` para el tipo `Colour` definido en un ejercicio anterior. Los casos para colores deben considerarse iguales a los casos definidos con las componentes RGB.

Clases para `AstroDistance`

Definir instancias para la clases `Eq`, `Ord` y `Num` para el tipo `AstroDistance` definido en un ejercicio anterior. Las consideraciones son las siguientes:

- Dos valores con distancias equivalente expresadas en unidades distintas deben considerarse iguales.
- Las comparaciones por `<` y `>` (así como `<=` y `>=`) también deben de considerar las conversiones entre unidades.
- Las operaciones de `+`, `-` y `*` deben devolver el resultado en la unidad del operando izquierdo.
- Las operaciones `abs` y `signum` deben devolver el resultado en la misma unidad del argumento.
- La función `fromInteger` debe devolver el resultado expresado en años luz.

Variantes:

1. Definir una instancia de la clase `Fractional` para el tipo `AstroDistance`.

Definición de clases

Clase `Booly`

Definir la clase `Booly` que incluye dos funciones: `truthy` y `falsy`. Ambas funciones toman un valor del tipo que implementa la clase y devuelven un valor booleano. Por defecto una se debe definir como la negación de la otra, de manera que la definición mínima para una instancia cualquiera incluye la implementación de una de las dos funciones.

La instancia para el tipo `Bool` sería así:

```
instance Booly Bool where
  truthy b = b
```

Definir luego instancias para tipos estándar de Haskell `Int`, `Double`, `Char`, `[a]`. Un valor se considera *falsy* si se trata del valor numérico `0` (o `0.0` o `'\0'`), una lista vacía; sino debe ser considerado *truthy*.

Variantes:

1. Definir una instancia de `Booly` para el tipo `(Maybe a)`, que no necesite que el tipo `a` instancie `Booly`. El único valor *falsy* debería ser `Nothing`.
2. Definir una instancia de `Booly` para el tipo `(Either a b)`, que necesite que los tipos `a` y `b` instancien `Booly`. Los valores *falsy* deberían corresponder con los valores *falsy* para los tipos `a` y `b`.
