Tarea 2

Tipos algebraicos y razonamiento ecuacional

Ejercicio 1 (Either) **5 Pt**

En clase vimos que podemos usar el constructor de tipos `Maybe` para definir funciones parciales. La limitación que tiene este constructor de tipos es que en caso de que haya múltiples causas de error, no permite distinguirlas (todas son mapeadas a `Nothing`). Para estos casos podemos usar en vez el constructor de tipos

```
data Either a b = Left a | Right b
```

provisto por el preludio estándar: usamos el constructor de valores `Left` para denotar un error y el constructor de valores `Right` para denotar un resultado exitoso.

(a) [4 Pt] Dados los siguientes sinónimos de tipos para representar errores y tablas asociativas

```
type Assoc k v = [(k,v)]
type Error = String
```

se pide definir la función de look-up

```
find :: (Eq k,Show k,Eq v) => k -> Assoc k v -> Either Error
   v
```

que permite distinguir el origen de los errores: clave no encontrada o clave asociada a múltiples valores:

```
> find 2 [(2,'a'),(1,'b')]
> Right 'a'

> find 2 [(2,'a'),(1,'b'), (2,'a')]
> Right 'a'

> find 3 [(2,'a'),(1,'b')]
> Left "Key 3 not found"

> find 2 [(2,'a'),(1,'b'),(2,'c')]
> Left "Multiple values for key 2"
```

Para la definición le puede resultar de utilidad la keyword `case` ... `of` ... que permite hacer pattern matching en lado derecho de una definición (i.e. en el cuerpo de la función que se está definiendo).

(b) [1 Pt] Explicar por qué se requiere cada uno de los constraints de tipos `Eq k`, `Show k`, `Eq v` en la definición de `find`.

Ejercicio 2 (Chequeador de tautologías) **15 Pt**

En este ejercicio adaptaremos el chequeador de tautologías visto en la clase 12.

(a) [5 Pt] Dar el tipo y definir la función `foldF` que captura el esquema de recursión primitiva asociado al tipo recursivo `Formula`. Con respecto al orden de los argumentos de `foldF`, seguir el mismo orden en el que aparecen los constructores de `Formula` en su declaración.

(b) [5 Pt] Redefinir las funciones `eval`, `fvar` utilizando `foldF`. Cuando defina `eval` puede asumir que siempre que evaluemos una fórmula lo vamos a hacer sobre una valuación que le asigna un único valor a cada variable (i.e. no es necesario implementar ningún tratamiento especial de errores). Para la definición de `eval` (en el caso de variables) debe usar la función `find` del ejercicio anterior.

(c) [5 Pt] Definir una función `isTaut :: Formula -> Maybe Valuation` que devuelva `Nothing` si la fórmula es una tautología, y `Just v` si la formula no es una tautología, donde `v` es una valuación (cualquiera si hay más de una) que haga la fórmula falsa.

```
> isTaut (Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (
    Var 'B'))
> Nothing

> isTaut (Imply (Var 'A') (And (Var 'A') (Var 'B')))
> Just [('A',True),('B',False)]
```

Ejercicio 3 (Torres de Hanoi) **17 Pt**

En este ejercicio modelaremos el juego de las *Torres de Hanoi*. Comenzar leyendo el archivo `Hanoi.pdf` donde se detallan las reglas y el objetivo del juego.

Se proveen las siguientes declaraciones para modelar las barras (palillos), discos, configuraciones y movimientos.

```
data Peg  = L | C | R deriving (Eq, Show)
type Disk = Int
type Conf = Peg -> [Disk]
type Move = (Peg,Peg)
```

Por ejemplo, la configuración que se muestra en la p. 10 de `Hanoi.pdf` corresponde a

```
c :: Conf
c L = [1..5]
c C = []
c R = []
```

Observar que la cabeza de la lista representa el disco superior de la barra. Los movimientos están representados por su barra origen (primer componente del par) y su barra destino (segunda componente del par). `L`, `C` y `R` son abreviaciones de *Left* (barra de la izquierda), *Center* (barra del medio) y *Right* (barra de la derecha).

(a) [5 Pt] Definir la función `step :: Move -> Conf -> Conf` que mueve un disco a partir de una configuración dada, y devuelve la configuración resultante. En caso de que se

intente realizar un movimiento ilegal (por ejemplo, desde una barra que esté vacía, o hacia una barra cuyo disco superior sea más pequeño que el disco que se intenta mover), se debe lanzar un error con un mensaje significativo.
Por ejemplo:

```
> step (L,R) c
> ([2,3,4,5],[],[1])

> step (R,C) c
> *** Exception: Trying to move from empty peg
```

Hint. Para la definición de `step` le puede resultar conveniente definir primero las funciones auxiliares `push :: Disk -> Peg -> Conf -> Conf` y `pop :: Peg -> Conf -> Conf` que ponen/sacan un disco en/desde una barra dada.

(b) [7 Pt] Definir la función `optStrategy :: Int -> Move -> Conf -> [(Move,Conf)]` que modela la estrategia óptima de resolución del juego descrita en la p. 13 del archivo `Hanoi.pdf`. Concretamente, `optStrategy n (s,t) c` mueve los `n` discos superiores de la barra `s` hacia la barra `t` partiendo de la configuración `c`, y devuelve la secuencia (en forma de lista) de movimientos que se requiere para ello, junto con las configuraciones intermedias generadas. La definición de `optStrategy` va a ser recursiva, y va a utilizar `step`.
Para testear la función `optStrategy` mientras la implementa, puede utilizar la función provista `play` (que invoca a `optStrategy`). Por ejemplo, `play 3 L R` describe la estrategia óptima para mover 3 discos desde la barra izquierda hacia la barra derecha, ilustrada en la p. 12 del archivo `Hanoi.pdf`.

```
> play 3 L R
> ([1,2,3],[],[])
   -> (L,R) -> ([2,3],[],[1])
   -> (L,C) -> ([3],[2],[1])
   -> (R,C) -> ([3],[1,2],[])
   -> (L,R) -> ([],[1,2],[3])
   -> (C,L) -> ([1],[2],[3])
   -> (C,R) -> ([1],[],[2,3])
   -> (L,R) -> ([],[],[1,2,3])
```

(c) [5 Pt] Testee la función `optStrategy` usando `QuickCheck`. Concretamente, verifique que:

1. Cuando se ejecuta `optStrategy` desde una configuración inicial (como la exhibida en la la p. 11 del archivo `Hanoi.pdf`), todas las configuraciones intermedias generadas son válidas (incluyendo, la configuración final). (Una configuración es *válida* si los discos de cada una de las 3 barras están en orden.) Observe que cuenta con la función `makeInit` para generar configuraciones iniciales.
2. Cuando se ejecuta `optStrategy` desde una configuración inicial con n discos, se requieren $2^n - 1$ movimientos para desplazar los discos a la barra destino.

Ejercicio 4 (Recursión primitiva sobre los naturales) **8 Pt**

Usando `foldNat` para capturar la recursión, dar una definición de la función `sumsqr :: Nat -> Nat` tal que $sumsqr(n) = \sum_{i=0}^{n} i^2$.

Ejercicio 5 (Inducción estructural) $\boxed{\textbf{15 Pt}}$

Considere la siguiente declaración de árboles binarios y del *fold* asociado.

```
data BinTree a = Leaf a | InNode (BinTree a) a (BinTree a)

foldBT :: (b -> a -> b -> b) -> (a -> b) -> (BinTree a -> b)
foldBT f g (Leaf v)           = g v
foldBT f g (InNode t1 v t2) = f (foldBT f g t1) v (foldBT f g t2)
```

(a) [3 Pt] Sean `f :: b -> a -> b -> b`, `g :: a -> b`, `f' :: b' -> a -> b' -> b'`, `g :: a -> b'` y `h :: b -> b'` tales que

1. `h (g v) = g' v`
2. `h (f x1 v x2) = f' (h x1) v (h x2)`

Usando inducción estructural, probar que `h . foldBT f g = foldBT f' g'`.

(b) [4 Pt] Usar el resultado del item (a) para probar que `length . faltenBT = sizeBT`, donde `flattenBT` y `sizeBT` devuelven la versión aplanada (en forma de lista) y el número de nodos de un árbol:

```
mirrorBT :: BinTree a -> BinTree a
mirrorBT = foldBT (\r1 v r2 -> InNode r2 v r1) Leaf

sizeBT :: BinTree a -> Int
sizeBT = foldBT (\r1 v r2 -> r1 + 1 + r2) (const 1)
```

Hint. Para su prueba puede asumir que `length (xs ++ ys) = length xs + length ys`.

(c) [4 Pt] Usar el resultado del item (a) para probar que `mirrorBT . mirrorBT = idBT` , donde `mirrorBT` devuelve el "espejo" de un árbol e `idBT` representa la función identidad sobre árboles:

```
mirrorBT :: BinTree a -> BinTree a
mirrorBT = foldBT (\r1 v r2 -> InNode r2 v r1) Leaf

idBT :: BinTree a -> BinTree a
idBT = foldBT InNode Leaf
```

(d) [4 Pt] Haciendo inducción sobre la estructura de `t`, probar que `map f (flattenBT t) = flattenBT (mapBT f t)`, donde:

```
mapBT :: (a -> b) -> BinTree a -> BinTree b
mapBT f (Leaf v)           = Leaf (f v)
mapBT f (InNode t1 v t2) = InNode (mapBT f t1) (f v) (mapBT f
    t2)
```

Para la prueba, puede asumir que `map f (xs ++ ys) = map f xs ++ map f ys`.