Tarea 4
Funtores, aplicativos y mónadas

Recuerde que para resolver la tarea, puede modularizar sus soluciones, a través de la definición y uso de funciones auxiliares. Todas las funciones auxiliares que defina deben ir acompañadas de su tip (más general) y un comentario describiendo su especificación.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a) source

14	16	15	13
6	8	7	5
2	4	3	1
10	12	11	9

(b) target

Figura 1: Matrices origen y destino de aplicar permutaciones de filas y columnas.

Ejercicio 1 (Mónada de listas) **14 Pt**

El objetivo de este ejercicio es determinar el número de permutaciones de filas o columnas que hay que aplicar a la matriz de la Figura 1a para obtener la matriz de la Figura 1b. Para ello contamos con las siguientes definiciones:

```
type Matrix = [[Int]]
type Size   = Int

source :: Matrix
source = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]

target :: Matrix
target = [[14,16,15,13],[6,8,7,5],[2,4,3,1],[10,12,11,9]]
```

(a) [2 Pt] Defina las funciones `swapRow, swapColumn :: Int -> Int -> Matrix -> Matrix` que permutan respectivamente dos filas y columnas de una matriz. Asuma que las filas y columnas empiezan a enumerarse desde 0. Para su definición, puede valerse de la función `swap :: Int -> Int -> [a] -> [a]` provista, que permuta dos elementos de una lista. No es necesario que incorpore ningún manejo particular de errores.

(b) [4 Pt] Dada una matriz `m` de tamaño `n`×`n`, defina la función `swapM :: Size -> Matrix -> [Matrix]` tal que `swapM n m` devuelva la lista de todas las matrices que se pueden obtener a partir de `m` permutando dos de sus filas o columnas. Su definición debe estar dada usando la `do`-*notation*. (Al permutar dos filas o columnas, asegúrese que tengan índice distinto, sino la permutación no tendrá ningún efecto.)

(c) [8 Pt] Defina la función `swapMUntil :: Size -> (Matrix -> Bool) -> (Int, [Matrix]) -> (Int, [Matrix])` que aplica repetidamente `swapM` a las matrices contenidas en una lista inicial, hasta que alguna de las matrices obtenidas satisface un predicado dado

(por el segundo argumento de `swapMUntil`). La función cuenta además el número de aplicaciones que son necesarias para ello. En particular

```
answer :: Int
answer = fst (swapMUntil 4 (== target) (0,[source]))
```

resuelve el problema originalmente planteado.
Al definir `swapMUntil`, explote la estructura monádica de las listas. Por ejemplo, si necesita aplicar `swapM` a una lista de matrices, hágalo a través del operador de *bind*.

Ejercicio 2 (Mónada de estado) **21 Pt**

En este ejercicio vamos a volver a resolver el problema de las Torres de Hanoi de la Tarea 2, pero usando la mónada de estado provista por Haskell (para tener acceso a la misma, asegúrese de tener instalada la librería mtl).

Dados los siguientes sinónimos de tipos,

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

el *estado* del juego está dado por el contenido (de tipo `Conf`) de las tres varillas (`L`, `C` y `R`, respectivamente).

(a) [5 Pt] Defina las funciones:
- `push :: Disk -> Peg -> State Conf Conf` que agrega un disco en (la parte superior de) una varilla y retorna la configuración resultante, y
- `pop :: Peg -> State Conf Disk` que retira el disco superior de una barilla, y retorna dicho disco.

(b) [3 Pt] Defina la función `step :: Move -> State Conf Conf` que mueve un disco a partir de una configuración dada, y devuelve la configuración resultante. La función debe ser definida en términos de `push` y `pop`, usando la `do`-*notation*. No es necesario que implemente ningún manejo particular de errores.

(c) [5 Pt] Defina la función `optStrategy :: Int -> Move -> State Conf [(Move,Conf)]` para que se comporte de la misma manera que en la Tarea 2. La definición va a ser recursiva, y va a utilizar `step`. Debe usar la `do`-*notation* para la misma.

(d) [3 Pt] ¿Qué diferencia encuentra entre la definición de `optStrategy` que dio en la Tarea 2 y la que dio aquí? ¿Si tuviera que elegir alguna de las dos, con cuál se quedaría? ¿Por qué?

(e) [5 Pt] Defina la función `play :: Int -> Peg -> Peg -> IO()` para que se comporte de la misma manera que en la Tarea 2. Por ejemplo,

```
> 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])
```

Ejercicio 3 (Mónada de probabilidades) **25 Pt**

En este ejercicio vamos a utilizar la mónada de probabilidades provista por la libreria probability (para tener acceso a la misma debe instalarla, según las instrucciones de sistema Haskell; por ejemplo, para Cabal, debería hacer `cabal install probability`). Su *script* debe contener al comienzo las siguientes líneas:

```
import Numeric.Probability.Distribution

type Probability = Rational
type Dist a = T Probability a
```

Arriba, el primer sinónimo de tipos especifica el tipo de números que se va a usar para representar las probabilidades; si lo desea puede cambiarlo, por ejemplo, a `Float`. El uso de la librería es sencillo. Para definir las distribuciones de probabilidad que modelan, por ejemplo, el lanzamiento de un dado, de una moneda y un experimento de Bernoulli general hacemos:

```
data CoinSide = H | T  deriving (Eq, Ord, Show)

die :: Dist Int
die = uniform [1..6]

coin :: Dist CoinSide
coin = uniform [H,T]

bern :: Probability -> Dist Bool
bern p = choose p True False
```

Podemos inspeccionar una distribución de probabilidad haciendo:

```
> die
> fromFreqs [(1,1%6), (2,1%6), (3,1%6), (4,1%6), (5,1%6),
   (6,1%6)]
```

Vemos que `die` le asigna a cada valor en el intervalo $1 \ldots 6$ una probabilidad de $1/6$. Si queremos calcular la probabilidad de que al tirar el dado salga un número menor o igual que 2, hacemos:

```
> (<= 2) ?? die
> 1%3
```

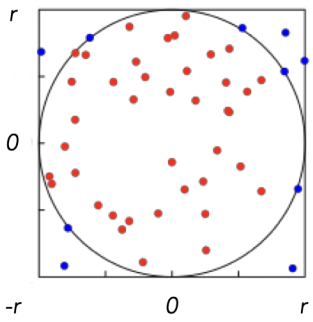
Podemos definir distribuciones de probabilidad más complejas utilizando todo el poder de Haskell, y la estructura monádica (por lo tanto también aplicativa y de functor) de las distribuciones de probabilidad. Por ejemplo, el siguiente programa recursivo genera una lista de un largo dado, donde cada elemento obedece la distribución `coin`:

```
coins :: Int -> Dist [CoinSide]
coins 0 = pure []
coins n = (:) <$> coin <*> coins (n-1)
```

Por último, el siguiente programa modela el experimento de tirar una monera; si sale cruz (`T`), devolver cruz, y si sale cara (`H`), devolver el resultado de un nuevo lanzamiento:

```
resampleIfHead :: Dist CoinSide
resampleIfHead = coin >>= f where
  f T = return T
  f H = coin
```

(a) Considere un círculo inscripto en un cuadrado como muestra la figura de la derecha. Si ubicamos un punto aleatoriamente en el cuadrado, es fácil ver que la probabilidad $Pr_{\odot}$ de que el punto caiga dentro del círculo satisface la ecuación



$$Pr_{\odot} = \frac{Area(\bigcirc)}{Area(\square)},$$

lo que inmediatamente da la identidad $\pi = 4Pr_{\odot}$.

Explotando esta observación, vamos a construir una aproximación de π.

I) [3 Pt] Como la mónada de probabilidades soporta sólo distribuciones discretas, va a necesitar discretizar el cuadrado de la figura. Comience definiendo la función `pointDist :: Int -> Dist (Int,Int)`: dado un entero `r`, `pointDist r` representa la ubicación de un punto que se ubica uniformemente distribuido en la grilla discreta $[-r, r] \times [-r, r]$. Utilice la `do`-*notation* para su definición.

II) [2 Pt] Ahora defina la función `resultE3a :: Int -> Probability` que devuelve un valor aproximado de π. El primer argumento representa a `r`.

(b) [20 Pt] 2 estudiantes de la UC y 8 de la UChile juegan la primera etapa de un torneo de tenis. Esta etapa consiste en 5 *singles* que se arman aleatoriamente, utilizando un bolillero: las dos primeras bolillas que salen forman el primer *single*, las siguientes dos bolillas forman el segundo *single*, y así sucesivamente.
Escriba un programa `resultE3b :: Probability` que devuelve la probabilidad que los 2 estudiantes de la UC no terminen en el mismo *single*.

Hint. Puede pensar que el "estado" del bolillero está dado por la cantidad de jugadores de cada universidad:

```
type Urn = (Int,Int)
```

y comenzar definiendo una función `pickPlayer :: Urn -> Dist (Uni, Urn)` que representa el proceso de sacar una bolilla, devolviendo la universidad del estudiante que la bolilla representa (junto al nuevo contenido del bolillero).

```
data Uni = Chile | Cato deriving Eq
```