Tarea 3

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.

Ejercicio 1 (Mónada de listas) — 10 Pt

Una hormiga parte del $(0, 0)$ de una cuadrícula, y en cada paso se mueve 1 unidad hacia arriba o hacia la derecha. Sin embargo, tiene una restricción adicional: en todo momento, la diferencia (en valor absoluto) entre el número de pasos totales hacia arriba y hacia la derecha debe ser menor o igual a 3.

(a) [5 Pt] Considerando el siguiente sinónimo de tipo para designar la posición de la hormiga:

```
-- First component: # moves to the right (x-coordinate)
-- Second component: # moves up (y-coordinate)
type Pos = (Int,Int)
```

implemente la función `possibleMoves :: Pos -> [Pos]` que dada la posición actual de la hormiga, devuelve la lista de posiciones que puede alcanzar haciendo un paso (válido). Para ello explote la estructura monádica de las listas.

(b) [5 Pt] Defina ahora la función `nSteps :: Int -> Pos -> [Pos]` que generaliza la función de arriba, al considerar n pasos en vez de 1 (lo lista devuelta no debe contener elementos repetidos). Defina la función recursivamente sobre n, y para ello utilice la función `possibleMoves`. Además, explote la estructura monádica de las listas para su definición.

Ejercicio 2 (Mónada de parcialidad recargada) — 15 Pt

De manera similar a `Maybe`, el constructor de tipo `Result` permite modelar funciones parciales, siendo la única diferencia que permite reportar el error de manera más informativa:

```
data Result e g = Fail e | Good g
```

Para cualquier tipo concreto `e`, `Result e` es un funtor, como se desprende de la siguiente declaración:

```
instance Functor (Result e) where
  -- fmap :: (a->b) -> Result e a -> Result e b
  fmap f (Fail err) = Fail err
  fmap f (Good res) = Good (f res)
```

(a) [6 Pt] Declare a `Result e` como instancias de las clases `Applicative` y `Monad`.

(b) [9 Pt] Alice está organizando un carrete para este fin de semana en su casa, y para ello va a ir en bicicleta a varias tiendas a comprar todo lo que necesita para que el carrete sea inolvidable. Desafortunadamente algo puede salir mal cuando viaja de una tienda a otra (e.g. se le pincha un neumático) o cuando hace la compra en cada tienda (e.g. no tienen el producto que buscaba). Esto lo modelamos con el siguiente par de funciones:

```
cycle :: Shop -> Shop -> Result String ()
shop  :: Shop -> Result String ()
```

Usando este par de funciones podemos definir una función que dada la lista de tiendas que tiene que recorrer (en su respectivo orden), lleva a cabo todas las compras:

```
doGroceries :: [Shop] -> Result String ()
doGroceries []         = Fail "No Shops to visit!"
doGroceries [x]        = shop x
doGroceries (x1:x2:xs) = case shop x1 of
    Fail e1 -> Fail e1
    Good _  -> case cycle x1 x2 of
        Fail e2 -> Fail e2
        Good _  -> doGroceries (x2:xs)
```

Reimplemente la función `doGroceries` usando la `do`-*notation* y explotando la estructura monádica de `Result`.

Ejercicio 3 (Mónada de estado) **12 Pt**

Recuerde que para manipular la mónada de estado, ademas de los operadores tradicionales de toda mónada (`return` y `(>>=)`), tambien puede utilizar las siguientes funciones (provistas por la librería `mtl`, que debe instalar siguiendo las instrucciones de su sistema de Haskell):

```
-- Retrieve the state (leaving it as is)
get :: State s s
get = State (\s -> (s, s))

-- Set the state (returning unit)
put :: s -> State s ()
put s = State (\_ -> ((), s))

-- Modify the state according to an update function (returning
   unit)
modify :: (s -> s) -> State s ()
modify f = State (\s -> ((), f s))
```

Alice y Bob juegan al siguiente juego. Bob tiene un número anotado en su mano y otro en una hoja de papel. Alice le puede dar las siguientes instrucciones a Bob:

- `Inc`: Bob debe sumarle 1 al número que tiene en su mano
- `Dec`: Bob debe restarle 1 al número que tiene en su mano
- `Mult`: Bob debe multiplicar el número que tiene en su mano por el que está escrito en la hoja de papel.

- `Store`: Bob debe sobrescribir el número que está en la hoja de papel, reemplazándolo por el que está escrito en su mano.

Dada la siguiente declaración de tipo

```
data Command = Inc | Dec | Mult | Store
```

y considerando que estado del juego está dado por los números en la hoja de papel y en la mano de Bob

```
-- Primer componente: numero en la hoja de papel
-- Segundo componente: numero en la mano de Bob
type GameState = (Int,Int)
```

defina la función `playGame :: [Command] -> State GameState Int` que a partir de un estado inicial (números en la hoja y mano de Bob iniciales) y una secuencia de instrucciones entregada por Alice, retorna el número final resultante en la mano de Bob. La definición debe ser dada usando la `do`-*notation*.

```
> evalState (playGame [Inc,Inc]) (0,0)
2
> evalState (playGame [Inc,Inc,Store,Mult]) (0,0)
4
```

Ejercicio 4 (Mónada de probabilidades) **23 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). 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
```

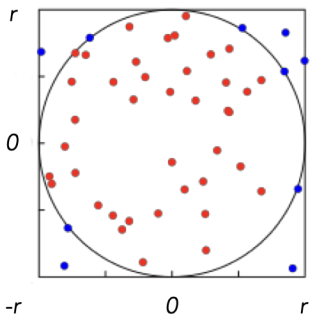
Podemos definir distribuciones de probabilidad más complejas utilizando todo el poder de Haskell, y la estructura monádica (por lo tanto también aplicativa y de functor) de las distribuciones de probabilidad. Por ejemplo, el siguiente programa recursivo genera una lista de un largo dado, donde cada elemento 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) [18 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
```