Práctica 6 - Mónadas

I/O Monádico

1. Escribir y **compilar** un programa (usando `ghc` en lugar de `ghci`) que imprima en pantalla la cadena `"Hola mundo!"`.

2. Dar una definición de la función getChars :: Int → IO String, que dado n lea n caracteres del teclado, usando las funciones sequenceA y replicate.

3. Escribir un programa interactivo que implemente un juego en el que hay que adivinar un número secreto predefinido. El jugador ingresa por teclado un número y la computadora le dice si el número ingresado es menor o mayor que el número secreto o si el jugador adivinó, en cuyo caso el juego termina. Ayuda: para convertir una String en Int puede usar la función read :: String → Int.

4. El juego nim consiste en un tablero de 5 filas numeradas de asteriscos. El tablero inicial es el siguiente:

1 : * * * * *
2 : * * * *
3 : * * *
4 : * *
5 : *

Dos jugadores se turnan para sacar una o mas estrellas de alguna fila. El ganador es el jugador que saca la última estrella. Implementar el juego en Haskell. Ayuda: para convertir una String en Int puede usar la función read :: String → Int.

5. Un programa pasa todos los caracteres de un archivo de entrada a mayúsculas y los guarda en un archivo de salida. Hacer un programa compilado que lo implemente tomando dos argumentos en la línea de comandos, el nombre de un archivo de entrada y el nombre de un archivo de salida.

Mónadas para modelar estados y continuaciones

6. Se desea modelar computaciones con un estado global s. Para esto se define el siguiente tipo de datos e instancia de mónada:

```
newtype State s a = St {runState :: s → (a, s)}
instance Monad (State s) where
  return x       = St (λs → (x, s))
  (St h) >>= f = St (λs → let (x, s') = h s
                          in runState (f x) s')
```

a) Probar que la instancia efectivamente define una mónada.

b) Definir operaciones set :: s → State s () y get :: State s s que permiten actualizar el estado y leerlo, respectivamente.

7. El tipo de datos Cont r a representa *continuaciones* en las que dado el resultado de una función (de tipo a) y la continuación de la computación $(a \to r)$, devuelve un valor en r. Probar que Cont r es una mónada. Ayuda: Guiarse por los tipos.

```
data Cont r a = Cont ((a → r) → r)
```

Combinando efectos

8. Se desea implementar un evaluador para un lenguaje sencillo, cuyos términos serán representados por el tipo de datos:

```
data T = Con Int | Div T T
```

Se busca que el evaluador cuente la cantidad de divisiones, y reporte los errores de división por cero. Se plantea el siguiente tipo de datos para representar una mónada de evaluación:

```
newtype M s e a = M {runM :: s → Error e (a, s)}
```

y entonces el evaluador puede escribirse de esta manera:

```
eval              :: T → M Int String Int
eval (Con n)      = return n
eval (Div t1 t2) = do v1 ← eval t1
                      v2 ← eval t2
                      if v2 ≡ 0 then raise "Error: Division por cero."
                                else do modify (+1)
                                        return (v1`div`v2)
```

y el cómputo resultante se podría ejecutar mediante una función auxiliar:

```
doEval  :: T → Error String (Int, Int)
doEval t = runM (eval t) 0
```

a) Dar la instancia de la mónada M s e.

b) Determinar el tipo de las funciones raise y modify, y dar su definición.

c) Reescribir eval sin usar notación **do** y luego expandir las definiciones de $\gg=$, return, raise y modify, para obtener un evaluador no monádico.

9. Dado el siguiente tipo de datos:

```
data M m a = Mk (m (Maybe a))
```

a) Probar que para toda mónada m, M m es una mónada.

b) Definir una operación auxiliar throw :: Monad $m \Rightarrow$ M m a que lanza una excepción.

c) Dada la mónada de estado StInt y el siguiente tipo N.

```
data StInt a = St (Int → (a, Int))
type N a = M StInt a
```

Definir operaciones get :: N Int y put :: Int → N (), que lean y actualizen (respectivamente) el estado de la mónada.

d) Usando N, definir un intérprete mónadico para un lenguaje de expresiones aritméticas y una sola variable dado por el siguiente AST.

```
data Expr = Var
          | Con Int
          | Let  Expr Expr
          | Add Expr Expr
          | Div  Expr Expr
```

El constructor Var corresponde a dereferenciar la única variable, Con corresponde a una constante entera, Let t, a asignar a la única variable el valor de la expresión t, y Add y Div corresponden a la suma y la división respectivamente. La variable tiene un valor inicial 0. El intérprete debe ser una función total que devuelva el valor de la expresión y el valor de la (única) variable. Por ejemplo, si llamamos a la única variable $\square$, la expresión: **let** $\square = (2 + 3)$ **in** $\square / 7$, queda representada en el AST por la expresión:

Let (Add (Con 2) (Con 3)) (Div Var (Con 7)