Functores

1. Demostrar que los siguientes tipos de datos son functores. Es decir, dar su instancia de la clase Functor correspondiente y probar que se cumplen las leyes de los functores.

   a) `data Pair a = P (a,a)`

   b) `data Tree a = Empty | Branch a (Tree a) (Tree a)`

   c) `data GenTree a = Gen a [GenTree a]`

   d) `data Cont a = C ((a -> Int)  Int)`

Soluciones

a)

```
instance Functor Pair where
  fmap g (P (x,x)) = P (g x, g x)
```

- `fmap id (P (x,x))`
  $\equiv \langle def.fmap \rangle$
  `P (id x, id x)`
  $\equiv \langle def.id \rangle$
  `P(x,x)`
  $\equiv \langle def.id \rangle$
  `id (P (x,x))`

- `fmap (f.g) (P (x,x))`
  $\equiv \langle def.fmap \rangle$
  `P ((f.g) x, (f.g) x)`
  $\equiv \langle def.\circ \rangle$
  `P (f (g x), f (g x))`
  $\equiv \langle def.fmap \rangle$
  `fmap f (P (g x, g x))`
  $\equiv \langle def.fmap \rangle$
  `fmap f (fmap g (P (x,x)))`

---

b)
```
instance Functor Tree where
  fmap g Empty = Empty
  fmap g (Branch x l r) = Branch (g x) (fmap g l) (fmap g r)
```

- `fmap id Empty`
  $\equiv \langle def.fmap.1 \rangle$
  `Empty`
  $\equiv \langle de.id \rangle$
  `id Empty`
- `fmap id (Branch x l r)`
  $\equiv \langle def.fmap.2 \rangle$
  `Branch (id x) (fmap id l) (fmap id r)`
  $\equiv \langle H.I \rangle$
  `Branch (id x) (id l) (id r)`
  $\equiv \langle def.id \rangle$
  `Branch x l r`
  $\equiv \langle def.id \rangle$
  `id (Branch x l r)`

- `fmap (f.g) Empty`
  $\equiv \langle def.fmap.1 \rangle$
  `Empty`
  $\equiv \langle def.fmap.1 \rangle$
  `fmap g Empty`
  $\equiv \langle fmap\ g\ Empty \equiv Empty + def.fmap.1 \rangle$
  `fmap f (fmap g Empty)`
- `fmap (f.g) (Branch x l r)`
  $\equiv \langle def.fmap.2 \rangle$
  `Branch ((f.g) x) (fmap (f.g) l) (fmap (f.g) r)`
  $\equiv \langle def.\circ \rangle$
  `Branch (f (g x)) (fmap (f.g) l) (fmap (f.g) r)`
  $\equiv \langle H.I. \rangle$
  `Branch (f (g x)) (fmap f (fmap g l)) (fmap f (fmap g r))`
  $\equiv \langle def.fmap.2 \rangle$
  `fmap f (Branch (g x) (fmap g l) (fmap g r))`
  $\equiv \langle def.fmap.2 \rangle$
  `fmap f (fmap g (Branch x l r))`

c) COMPLETAR.

d) COMPLETAR.

2. Probar que las siguientes instancias no son correctas (no se cumplen las leyes de los functores).

a)
```
data Func a = Func (a -> a)
  instance Functor Func where
    fmap g (Func h) = Func id
```

b)
```
data Br b a = B b (a,a)
  instance Functor (Br b) where
    fmap f (B x (y,z)) = B x (f z, f y)
```

Soluciones

a) `fmap id (Func f)`

$\equiv \langle def.fmap \rangle$

`Func id`

$\not\equiv$

`Func f`

$\equiv \langle def.id \rangle$

`id (Func f)`

b) `fmap id (B x (y,z))`

$\equiv \langle def.fmap \rangle$

`B x (id z, id y)`

$\equiv \langle def.id \rangle$

`B x (z,y)`

$\not\equiv$

`B x (y,z)`

$\equiv \langle def.id \rangle$

`id (B x (y,z))`

Monadas como substitucion

3. Consideremos el siguiente (AST de un) lenguaje:

```
data A a = Num Int | Sum (A a) (A a) | Mul (A a) (A a)
         | Res (A a) (A a) | Var a
```

para el cual implementamos sustitucion simultanea:

```
(>>=) :: A a -> (a -> A b) -> A b
Num n >>= v = Num n
Sum t u >>= v = Sum (t >>= v) (u >>= v)
Mul t u >>= v = Mul (t >>= v) (u >>= v)
Res t u >>= v = Res (t >>= v) (u >>= v)
Var a >>= v = v a
```

Dados los terminos `x = Var "x"`, `y = Var "y"` y `z = Var "z"`, dar el tipo y definicion de g y h tal que:

- `Sum x (Mul y z) >>= g`

  $\equiv \langle def.g \rangle$

  `Sum (Var 1)(Mul (Res (Var 3) (Var 1)) (Mul (Var 2) (Var 2)))`

  y

- `Sum x (Mul y z) >>= h`

  $\equiv \langle def.h \rangle$

  `Sum (Var 1) (Res (Var 2) (Var 3))`

¿Cuantas soluciones existen en cada caso?

Soluciones

- ```
  g :: String -> A Int
  g "x" = Var 1
  g "y" = Res (Var 3) (Var 1)
  g "z" = Mul (Var 2) (Var 2)
  ```
- No existen soluciones.
  ```

4. Dado el tipo de datos

```
data BT a = IfBoton (Bool -> BT a)
          | Beep (BT a)
          | Var a
```

donde IfBoton detecta la pulsacion de un boton y Beep produce un beep, escribir un programa que haga un beep cada dos pulsaciones de boton.

Soluciones

```
beep2 = beep2' 0 where
  beep2' 0 = IfBoton (\b -> if b then beep2' 1 else beep2)
  beep2' 1 = IfBoton (\b -> if b
                            then (Beep $ Var ()) >> beep2
                            else beep2' 1)
```

```
beep2 = Var 0 >>= g where
  g 0 = IfBoton (\b -> if b then Var 1 >>= g else Var 0 >>= g)
  g 1 = IfBoton (\b -> if b then Beep (Var 2) >>= g else Var 1 >>= g)
  g 2 = Var 0 >>= g
```

---

5. Definimos el AST de un lenguaje entrada/salida de la siguiente manera:

```
data ES = Read (Char -> ES) | Write Char (ES)
```

Si el lenguaje lee un caracter de entrada y en base a eso decide como seguir o escribe un caracter y continua la ejecucion ¿Que hacen los siguientes programas?

a) `t1 = Read (\c -> Write c (Write c t1))`

b) `t2 = Read (\c -> Write "(" (Write c (Write ")" t2)))`

c) `t3 = Write "(" (Read (\c -> Write c (Write ")" t3)))`

d) `t4 = Read (\_ -> t4)`

Soluciones

a) Cada caracter leido, es escrito dos veces en pantalla.

b) Cada caracter leido, es escrito entre parentesis.

c) Escribe en pantalla «(«, lee un caracter y lo imprime, y antes de volver a empezar escribe «)» en pantalla.

d) No imprime los caracteres leidos.

6. Teniendo en cuenta los siguientes tipos:

```
data ES a = Read (Char -> ES a) | Write Char (ES a) | Var a
```

Escribir un programa:

a) `writeChar :: Char -> ES ()` que escriba su argumento y finalice con una variable `()`.

b) `readChar :: ES Char` que lea un caracter y finalice con la variable cuyo nombre es el caracter leido.

c) Usando `writeChar`, escribir un programa `writeStr :: String -> ES ()` que imprima la cadena que se pasa como argumento.

Soluciones

a)
```
writeChar c = Write c (Var ()).
```

b)
```
readChar = Read (\c -> Var c).
```

c)
```
writeStr [] = Var ()
writeStr (x:xs) = writeChar x >> writeStr xs
```

7. Dada la siguiente implementacion de la sustitucion para terminos:

```
(>>=) :: ES a -> (a -> ES b) -> ES b
Read k >>= v = Read (\c -> k c >>= v)
Write c t >>= v = Write c (t >>= v)
Var a >>= v = v a
```

¿Que hace el siguiente programa?

```
f :: ES String
f = readChar >>= g where g '\n' = Var []
                         g c = f >>= \xs -> Var (c:xs)
```

Solucion Lee caracteres (sin imprimirlos) hasta presionar enter, y el estado terminal es la cadena leida.

Uso de Monadas y notacion do

8. Probar que toda monada es un functor, es decir, proveer una instacia

```
instance Monad m => Functor m where
  fmap...
```

y probar que las leyes de los functores se cumplen para su definicion de fmap.

Solucion

```
instance Monad m => Functor m where
  fmap f m = m >>= (return.f)
```

- `fmap id m`

  $\equiv \langle def.fmap \rangle$

  `m >>= (return.id)`

  $\equiv \langle f \circ id \equiv f \rangle$

  `m >>= return`

  $\langle monad.law.2 \rangle$

  `m`

  $\equiv \langle def.id \rangle$

  `id m`

- `fmap (f.g) m`

  $\equiv \langle def.fmap \rangle$

  `m >>= (return.(f.g))`

  $\equiv \langle \eta - redex + def. \circ + def. \circ \rangle$

  `m >>= (\x -> return (f (g x)))`

  $\equiv \langle def. \circ \rangle$

  `m >>= (\x -> (return.f) (g x))`

  $\equiv \langle monad.law.1 \rangle$

  `m >>= (\x -> return (g x) >>= (return.f))`

  $\equiv \langle def. \circ \rangle$

  `m >>= (\x -> (return.g) x >>= (return.f))`

  $\equiv \langle monad.law.3 \rangle$

  `m >>= (return.g) >>= (return.f)`

  $\equiv \langle def.fmap \rangle$

  `fmap f (m >>= (return.g))`

  $\equiv \langle def.fmap \rangle$

  `fmap f (fmap g m)`

---

9. Definir las siguientes funciones:

   a) `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`, tal que `mapM f xs` aplique la funcion monadica f a cada elemento de la lista xs, retornando la lista de resultados encapsulada en la monada.

   b) `foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a`, analogamente a $foldl$ para listas, pero con su resultado encapsulado en la monada. Ejemplo:

```
foldM f e1 [x1, x2, x3] = do e2 <- f e1 x1
                             e3 <- f e2 x2
                             f e3 x3
```

Soluciones

a)
```
mapM _ [] = return []
mapM f (x:xs) = do x' <- f x
                   xs' <- mapM f xs
                   return $ x':xs'
```

b)
```
foldM _ e1 [] = return e1
foldM f e1 (x:xs) = do e2 <- f e1 x
                       foldM f e2 xs
```

10. Escribir el siguiente fragmento de programa monadico usando notacion do:

```
(m >>= \x -> h x) >>= \y -> f y >>= \z -> return (g z)
```

Solucion

```
do x <- m
   y <- h x
   z <- f y
   return (g z)
```

11. Escribir el siguiente fragmento de programa en terminos de `>>=` y `return`:

```
do x <- (do z <- y
            w <- f z
            return (g w z))
   y <- h x 3
   if y then return 7
        else do z <- h x 2
                return (k z)
```

Solucion

```
(y >>= \z -> (f z >>= \w -> return $ g w z)) >>=
\x -> (h x 3 >>= \y -> if y
                       then return 7
                       else h x 2 >>= \z -> return $ k z)
```

12. Escribir las leyes de las monadas usando la notacion do.

    Solucion

    - `do { x <- return a; f x }` $\equiv$ `f a`.
    - `do { x <- m; return x }` $\equiv$ `m`.
    - ```
      do x <- m
         y <- f x
         g y
      ```
      $\equiv$
      ```
      do x <- m
         do y <- f x
            g y
      ```

I/O Monadico

13. Escribir y *compilar* un programa (usando ghc en lugar de ghci) que imprima en pantalla la cadena «Hola mundo!».

    Solucion

    ```
    main = putStrLn "Hola mundo!"
    ```

14. Escribir un programa interactivo que implemente un juego en el que hay que adivinar un numero secreto predefinido. El jugador ingresa por teclado un numero y la computadora le dice si el numero ingresado es menor o mayor que el numero secreto o si el jugador adivino, en cuyo caso el juego termina. Ayuda: para convertir una String en Int pude sar la funcion `read :: String -> Int`.
    ```

Solucion

```
import System.IO

getPass = do hSetEcho stdin False
             pass <- getPass'
             hSetEcho stdin True
             return pass

getPass' = do x <- getChar
              if x == '\n' then do putChar x
                                   return []
                           else do putChar '*'
                                   xs <- getPass'
                                   return (x:xs)

game :: Int -> Int -> IO ()
game n 0 = do putStrLn "El juego ha finalizado."
game n t = do putStrLn $ "Intentos restantes: " ++ show t
              putStr "Ingrese un numero: "
              x <- getLine
              guess n t (read x)

guess n t x | x < n = do putStrLn "Incorrecto. El numero es mas grande."
                         game n (t - 1)
            | x > n = do putStrLn "Incorrecto. El numero es mas chico."
                         game n (t - 1)
            | otherwise = do putStrLn "Correcto!"

main = do putStr "Ingrese el numero secreto: "
          n <- getPass
          putStr "Ingrese la cantidad de intentos: "
          t <- getLine
          game (read n) (read t)
```

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

    a) *****

    b) ****

    c) ***

    d) **

    e) *

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

Solucion

```
printB []     = do putStrLn ""
printB (x:xs) = do putStrLn x
                   printB xs
valid 1 (x:xs) = (not.null) x
valid n b@(x:xs)
    | n <= 0 || n > length b = False
    | otherwise = valid (n-1) xs
play 1 (xs:xss) = (tail xs):xss
play n (xs:xss) = xs : (play (n-1) xss)
winner [] = True
winner (xs:xss) = (null xs) && (winner xss)
ask player board = do printB board
                      putStr $ "Jugador " ++
                               (show $ player+1) ++
                               ", ingrese su jugada: "
                      c <- getLine
                      if valid (read c) board then
                         let newboard = play (read c) board
                         in  if winner newboard
                             then putStrLn "Usted ha ganado."
                             else ask (mod (player+1) 2) newboard
                      else do putStrLn "JUGADA INVALIDA!"
                              ask player board
main = ask 0 start ["*****", "****", "***", "**", "*"]
```

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

    Solucion COMPLETAR.