

## Funtores

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

- 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))`  
 $\equiv \langle \text{def.fmap} \rangle$   
`P (id x, id x)`  
 $\equiv \langle \text{def.id} \rangle$   
`P(x,x)`  
 $\equiv \langle \text{def.id} \rangle$   
`id (P (x,x))`
- 
- `fmap (f.g) (P (x,x))`  
 $\equiv \langle \text{def.fmap} \rangle$   
`P ((f.g) x, (f.g) x)`  
 $\equiv \langle \text{def.o} \rangle$   
`P (f (g x), f (g x))`  
 $\equiv \langle \text{def.fmap} \rangle$   
`fmap f (P (g x, g x))`  
 $\equiv \langle \text{def.fmap} \rangle$   
`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)`

$\equiv \langle def.o \rangle$

`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 funtores).

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`

$\neq$

`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)`

$\neq$

`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 substitucion 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  $x = \text{Var } "x"$ ,  $y = \text{Var } "y"$  y  $z = \text{Var } "z"$ , dar el tipo y definicion de  $g$  y  $h$  tal que:

- $\text{Sum } x (\text{Mul } y z) \gg= g$   
 $\equiv \langle \text{def}.g \rangle$   
 $\text{Sum } (\text{Var } 1) (\text{Mul } (\text{Res } (\text{Var } 3) (\text{Var } 1)) (\text{Mul } (\text{Var } 2) (\text{Var } 2)))$   
 $y$
- $\text{Sum } x (\text{Mul } y z) \gg= h$   
 $\equiv \langle \text{def}.h \rangle$   
 $\text{Sum } (\text{Var } 1) (\text{Res } (\text{Var } 2) (\text{Var } 3))$

¿Cuántas soluciones existen en cada caso?

## Soluciones

- $g :: \text{String} \rightarrow A \text{ Int}$   
 $g "x" = \text{Var } 1$   
 $g "y" = \text{Res } (\text{Var } 3) (\text{Var } 1)$   
 $g "z" = \text{Mul } (\text{Var } 2) (\text{Var } 2)$
- 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 leído, es escrito dos veces en pantalla.
- b) Cada caracter leído, 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 leídos.

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 leído.
- 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 leída.

## Uso de Monadas y notacion do

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

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

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

## Solucion

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

---

```

■ fmap id m
  ≡ ⟨def.fmap⟩
  m >>= (return.id)
  ≡ ⟨f ∘ id ≡ f⟩
  m >>= return
  ⟨monad.law.2⟩
  m
  ≡ ⟨def.id⟩
  id m
```

---

```

■ fmap (f.g) m
  ≡ ⟨def.fmap⟩
m >>= (return.(f.g))
  ≡ ⟨η - redex + def.∘ + def.∘⟩
m >>= (\x -> return (f (g x)))
  ≡ ⟨def.∘⟩
m >>= (\x -> (return.f) (g x))
  ≡ ⟨monad.law.1⟩
m >>= (\x -> return (g x) >>= (return.f))
  ≡ ⟨def.∘⟩
m >>= (\x -> (return.g) x >>= (return.f))
  ≡ ⟨monad.law.3⟩
m >>= (return.g) >>= (return.f)
  ≡ ⟨def.fmap⟩
fmap f (m >>= (return.g))
  ≡ ⟨def.fmap⟩
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 función monádica  $f$  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`, análogamente 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 } ≡ f a.
▪ do { x <- m; return x } ≡ m.
▪ do x <- m
    y <- f x
    g y
  ≡
  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` puede ser la funcion `read :: String -> Int`.

### Solucion

```

import System.Random

getNumber s = do { putStrLn s; readLn }

game r 0 = do putStrLn ("Fin del juego. El numero era: " ++ show r)
game r i = do n <- getNumber "Ingrese un numero: "
              putStrLn (guess r n)
              if n /= r then game r (i - 1) else return ()

```

```

guess r n | n < r      = "Incorrecto. El numero es mas grande."
          | n > r      = "Incorrecto. El numero es mas chico."
          | otherwise = "Correcto!"

main = do m <- getNumber "Ingrese numero maximo: "
         r <- randomRIO (0, m) :: IO Int
         game r (floor (logBase 2 $ fromIntegral m))

```

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

```

valid r b q | r <= 0 || r > length b      = False
            | q <= 0 || q > length (b !! (r - 1)) = False
            | otherwise = not $ null $ b !! (r - 1)

play r b q = [if i == r then drop q xs else xs | (xs, i) <- zip b [1..]]

getNumber s = do { putStr s; readLn }

ask p b | and (map null b) = putStrLn "Usted gano!"
        | otherwise = do mapM print (zip [1..] b)
                        putStrLn $ "\nJugador " ++ (show $ p + 1)
                        r <- getNumber "Fila: "
                        q <- getNumber "Cantidad: "
                        if valid r b q then ask (mod (p + 1) 2) (play r b q)
                        else ask p b

main = ask 0 ["*****", "****", "***", "**", "*"]

```

16. 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.

**Solucion** COMPLETAR.