CI4251 - Programación Funcional Avanzada
Tarea 2

Ernesto Hernández-Novich
86-17791
<emhn@usb.ve>

Mayo 12, 2016

Yo dawg! I heard you liked functors...

```
import Control.Applicative
```

Considere el siguiente tipo de datos, en el cual `f` puede ser *cualquier* `Functor`.

```
data Bonus f a = Malus a
               | Bonus (f (Bonus f a))
```

Escriba las instancias `Functor`, `Applicative` y `Monad` para el tipo `Functor f`.

```
instance Functor f => Functor (Bonus f) where

instance Functor f => Applicative (Bonus f) where

instance Functor f => Monad (Bonus f) where
```

Práctica obligada

Reescriba la librería `MiniLogo` construyendo un monad combinado que aproveche `RWS`, `Exception` (estilo nuevo – **no** use `Error`) e `IO`. Al reescribirla, incorpore las siguientes restricciones:

- La tabla de colores debe estar en el ambiente «sólo lectura».
- El diagrama en curso debe estar en el ambiente «acumulador».
- Es un error fatal usar el color rojo después del verde (o viceversa).
- Es un error fatal ir hacia la izquierda dibujando con rojo. Note que «izquierda» es cualquier dirección con tendencia a la izquierda (más de 90 y menos de 270 grados), siendo necesario saber cuán «izquierdosa» fue la tortuga. No se complique: limpie la ventana y redibuje.
- Una vez presentado el dibujo, el programa espera por una tecla. Si se oprime la 'a', debe «borrarse» el último paso del dibujo; si se oprime la 's' debe «repetirse» el siguiente paso del dibujo; y si se oprime cualquier otra tecla debe terminar el programa.
- Si ocurrió un error, no interesa el estado de la simulación.

Can I has pizza?

Tengo más hambre que el país. Tengo tanta hambre que lloro. Haría lo que fuera por pizza. Mentira: nunca programaría en Python, Java, ni (¡asco!) PHP. Tampoco en JavaScript. Pero nos desviamos del tema y es que **necesito** pizza...

```
data Want a = Want ((a -> Pizza) -> Pizza)
```

Una cosa es comer la pizza, tanto mejor si está en un combo 2x1, pero de todas todas, al final uno queda feliz.

```
data Pizza = Eat (IO Pizza)
           | Combo Pizza Pizza
           | Happy
```

Y la pizza es un espectáculo digno de ver, y aunque el proceso de comerla (`IO`) oculte lo que tiene, es posible hacerse una imagen aproximada – sobre todo para hacer «pizza debugging».

```
instance Show Pizza where
   show (Eat x)     = " :-P "
   show (Combo x y) = " combo(" ++ show x
                                ++ ","
                                ++ show y ++ ") "
   show Happy       = " :-D "
```

Como dije, quiero pizza, y haría cualquier cosa por obtenerla hasta ser feliz.

```
want :: Want a -> Pizza
want = undefined
```

Eso no quiere decir que sin pizza no se pueda ser feliz – aunque suene difícil o inconcebible, a veces uno simplemente es feliz.

```
happy :: Want a
happy = undefined
```

Imagínate la pizza. Es tuya. La devoras. Y con cada bocado es inevitable querer más...

```
nomnom :: IO a -> Want a
nomnom = undefined
```

Y mientras masticas la pizza piensas que hoy es jueves. ¡Jueves de 2x1! Hay que aprovechar el combo, hasta que no quede nada.

```
combo :: Want a -> Want ()
combo = undefined
```

«Es demasiada pizza» – obviamente una falacia: nunca es demasiada pizza, porque *siempre* hay un pana con el cual compartir las ganas de pizza (sobre todo si pagan por partes iguales).

```
pana :: Want a -> Want a -> Want a
pana = undefined
```

Es momento de ir a por ella. Toda la que haya. Toda la que se pueda. Suficiente para todos. Allí se pide el combo. Todos comen. Y cada uno es feliz.

```
pizzeria :: [Pizza] -> IO ()
pizzeria = undefined
```

Porque las ganas son sólo el contexto. Porque cada topping que se agrega no hace sino anticipar la secuencia de bocados, que uno, sólo o acompañado, quiere satisfacer. La pizza no será un `Monad`, pero las ganas de comerla si.

```
instance Monad Want where
  return x       = undefined
  (Want f) >>= g = undefined
```

Nota: el propósito de este ejercicio es que noten que son los *tipos* los que deben describir el comportamiento. Hay sólo una manera correcta de escribir todas las funciones aquí solicitades, para lo cual lo único necesario es considerar el tipo a producir y los tipos de los argumentos. Así mismo, para escribir la instancia `Monad`, sólo es necesario respetar las firmas.
Para saber si su código funciona, primero debe compilar todo sin errores. Luego, ejecute

```
ghci> tengo hambre
```

y el contenido le hará entender si lo logró.