Programación Funcional

Ejercicios de Práctica Nro. 13

Mónadas

Aclaraciones:

- Los ejercicios siguen un orden de complejidad creciente, y cada uno puede servir a los siguientes. No se recomienda saltear ejercicios sin consultar antes a un docente.
- Recordar que se pueden aprovechar en todo momento las funciones ya definidas, tanto las de esta misma práctica como las de prácticas anteriores.
- Probar todas las implementaciones, al menos en una consola interactiva.
- Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evalúan principalmente este aspecto. Para utilizando formas alternativas al resolver los ejercicios consultar a los docentes.

Ejercicio 1) Revisar el código de los archivos indicados (que se encuentra en el **Apéndice A** y también pueden encontrarse en un recurso en el aula) y entenderlo. Se sugiere empezar por **`Main`** y desandar el camino de imports.

a. **Archivo** **`Main.hs`**
b. **Archivo** **`Monadas.hs`**
c. **Archivo** **`MaybeMonad.hs`**
d. **Archivo** **`ErrorMonad.hs`**
e. **Archivo** **`OutputMonad.hs`**
f. **Archivo** **`RevOutputMonad.hs`**
g. **Archivo** **`PowerOutputMonad.hs`**
h. **Archivo** **`OutputErrorMonad.hs`**
i. **Archivo** **`Eval.hs`**

Ejercicio 2) Completar el módulo **`OutputErrorMonad.hs`**, ítem **`h`** del ejercicio anterior, para que implemente un tipo que pueda ser instancia de **`ErrorMonad`** y **`PrintMonad`**, y probarlo en el **`Main`** (descomentando la última línea de la función **`probar`** y haciendo lo necesario para que funcione).

EJERCICIOS DE BONUS (un poco más complejos)

Ejercicio 3) Revisar el código de los archivos indicados (que se encuentra en el **Apéndice B** y también pueden encontrarse en un recurso en el aula) y entenderlo (se trata de un proyecto diferente del anterior). Se sugiere empezar por **`Main`** y desandar el camino de imports.

a. **Archivo** **`Main.hs`**
b. **Archivo** **`Monadas.hs`**
c. **Archivo** **`RNExp.hs`**
d. **Archivo** **`ReaderMonad.hs`**
e. **Archivo** **`Mem.hs`**
f. **Archivo** **`ListMem.hs`**

Ejercicio 4) Completar la implementación de la función **`evalNE`** en el módulo **`RNExp.hs`** del ejercicio anterior, de forma de lograr que **`main`** imprima el siguiente resultado:

```
Main> main
El resultado de (x+1) cuando x es 16 es: 17
El resultado de (x+1) cuando x es 41 es: 42
```

Nota: Observar que en este caso, la implementación de una mónada de prueba ya está hecha, y lo que debe hacerse es definir una función genérica que la pruebe.

Ejercicio 5) Intentar realizar otra implementación de la mónada del ejercicio 3.

Apéndice A: Módulos para el ejercicio 1

Archivo `Main.hs`

```
import Monadas           -- Las definiciones de las clases necesarias
import MaybeMonad        -- La definición de la mónada Maybe
import ErrorMonad        -- La definición de la mónada Error
import OutputMonad       -- La definición de la mónada Output
import RevOutputMonad    -- La definición de la mónada RevOutput
import PowerOutputMonad -- La definición de la mónada PowerOutput
import OutputErrorMonad -- La definición de la mónada OutputError
import Eval              -- La aplicación (diferentes evals para E)

main = probar 2

probar :: Int -> IO ()
probar i = do print (prueba i)
              print (pruebaM1 i)
              print (pruebaM2 i)
              print (pruebaME1 i)
              print (pruebaME2 i)
              print (pruebaMP1 i)
              print (pruebaMP2 i)
              print (pruebaMP3 i)
              -- print (pruebaMPE i)

prueba :: Int -> Float
prueba 0 = eval ej0
prueba 1 = eval ej1
prueba 2 = eval ej2

pruebaM1 :: Int -> Maybe Float
pruebaM1 0 = evalM ej0
pruebaM1 1 = evalM ej1
pruebaM1 2 = evalM ej2

pruebaM2 :: Int -> Error Float
pruebaM2 0 = evalM ej0
pruebaM2 1 = evalM ej1
pruebaM2 2 = evalM ej2

pruebaME1 :: Int -> Maybe Float
pruebaME1 0 = evalME ej0
pruebaME1 1 = evalME ej1
pruebaME1 2 = evalME ej2

pruebaME2 :: Int -> Error Float
pruebaME2 0 = evalME ej0
pruebaME2 1 = evalME ej1
pruebaME2 2 = evalME ej2
```

```
pruebaMP1 :: Int -> Output Float
pruebaMP1 0 = evalMP ej0
pruebaMP1 1 = evalMP ej1
pruebaMP1 2 = evalMP ej2

pruebaMP2 :: Int -> RevOutput Float
pruebaMP2 0 = evalMP ej0
pruebaMP2 1 = evalMP ej1
pruebaMP2 2 = evalMP ej2

pruebaMP3 :: Int -> PowerOutput Float
pruebaMP3 0 = evalMP ej0
pruebaMP3 1 = evalMP ej1
pruebaMP3 2 = evalMP ej2

-- pruebaMPE :: Int -> <<poner acá el nombre de tu monada>> Float
pruebaMPE 0 = eval PE ej0
pruebaMPE 1 = evalMPE ej1
pruebaMPE 2 = evalMPE ej2
```

Archivo `Monadas.hs`

```
module Monadas where
class Monad m => ErrorMonad m where
  throw :: String -> m a
class Monad m => PrintMonad m where
  printf :: String -> m ()
```

Archivo `MaybeMonad.hs`:

```
module MaybeMonad where
import Monadas
instance ErrorMonad Maybe where
  throw msg = Nothing
```

Archivo `ErrorMonad.hs`:

```
module ErrorMonad where
import Monadas
data Error a = Throw String | Ok a   deriving Show

instance Monad Error where
  return x = Ok x
  m >>= k  = case m of
              Throw s -> Throw s
              Ok v    -> k v
  fail msg = Throw msg

instance ErrorMonad Error where
  throw msg = Throw msg
```

Archivo `OutputMonad.hs`:

```
module OutputMonad where
import Monadas

type Screen = String
data Output a = O (a, Screen)   deriving Show

instance Monad Output where
  return x = O (x, "")
  m >>= k  = let O (v, scr1) = m
             in let O (res, scr2) = k v
                 in O (res, scr1 ++ scr2)
  fail msg = error msg

instance PrintMonad Output where
  printf msg = O ((), msg ++ "\n")
```

Archivo `RevOutputMonad.hs`:

```
module RevOutputMonad where
import Monadas
type Screen = String
data RevOutput a = O2 (Screen, a)   deriving Show

instance Monad RevOutput where
  return x = O2 ("", x)
  m >>= k  = let O2 (scr1, v) = m
             in let O2 (scr2, res) = k v
                 in O2 (scr2 ++ scr1, res)
  fail msg = error msg

instance PrintMonad RevOutput where
  printf msg = O2 (msg ++ "\n", ())
```

Archivo `PowerOutputMonad.hs`:

```
module PowerOutputMonad where
import Monadas
type PowerScreen = [ String ]
data PowerOutput a = O3 (a, PowerScreen)   deriving Show

instance Monad PowerOutput where
  return x = O3 (x, [])
  m >>= k  = let O3 (v, scr1) = m
             in let O3 (res, scr2) = k v
                 in O3 (res, scr1 ++ scr2)
```

```
  fail msg = error msg

instance PrintMonad PowerOutput where
  printf msg = O3 ((), [ msg ])
```

Archivo `OutputErrorMonad.hs`:

```
module OutputErrorMonad where
-- COMPLETAR
```

Archivo `Eval.hs`:

```
module Eval where
import Monadas
-- Un tipo de expresiones para probar mónadas
data E = Cte Float | Div E E

-- Una operación con bottom
--  (porque el (/) ahora usa la porquería de IEEE 1.#INF...)
_ ./. 0 = error "Undefined"
n ./. m = n/m

ej0 = Div (Cte 1) (Cte 0)
ej1 = Div (Cte 18) (Cte 3)
ej2 = Div (Div (Cte 80) (Cte 10))
          (Div (Cte 20) (Cte 5))

armarDiv v1 v2 = show v1 ++ " / "
              ++ show v2 ++ " = "
              ++ show (v1./.v2)

-- El evaluador básico (este se puede ejecutar sin más trámite)
eval :: E -> Float
eval (Cte n)     = n
eval (Div e1 e2) = eval e1 ./. eval e2

-- El evaluador monádico (para poder ejecutar este hay que indicar
--   de alguna forma cuál es la mónada. Una forma típica de hacerlo
--   es dar el tipo del resultado:
--         evalME (Cte 2) :: Maybe Float )
evalM :: Monad m => E -> m Float
evalM (Cte n)     = return n
evalM (Div e1 e2) = evalM e1 <./.> evalM e2

(<./.>) :: Monad m => m Float -> m Float -> m Float
m1 <./.> m2 = liftM2 (./.) m1 m2
```

```
liftM2 f m1 m2 = m1 >>= \v1 ->
                 m2 >>= \v2 ->
                 return (f v1 v2)

-- El evaluador, con control de error (mismo tema que con todos los
--    que tienen clases en las restricciones de contexto)
evalME :: ErrorMonad m => E -> m Float
evalME (Cte n)     = return n
evalME (Div e1 e2) = evalME e1 </> evalME e2

(</>) :: ErrorMonad m => m Float -> m Float -> m Float
m1 </> m2 = m1 >>= \v1 ->
            m2 >>= \v2 ->
            if v2==0
              then throw "No puedo dividir por cero"
              else return (v1./.v2)

-- El evaluador, con impresión de traza (mismo tema que con todos
--   los que tienen clases en las restricciones de contexto)
evalMP :: PrintMonad m => E -> m Float
evalMP (Cte n)     = return n
evalMP (Div e1 e2) = evalMP e1 <<./.>> evalMP e2

(<<./.>>) :: PrintMonad m => m Float -> m Float -> m Float
m1 <<./.>> m2 = m1 >>= \v1 ->
                m2 >>= \v2 ->
                printf (armarDiv v1 v2) >>= \_ ->
                return (v1./.v2)

-- El evaluador, combinando traza y control de error (mismo tema
--   que con todos los que tienen clases en las restricciones
--   de contexto)
evalMPE :: (ErrorMonad m, PrintMonad m) => E -> m Float
evalMPE (Cte n)     = return n
evalMPE (Div e1 e2) = evalMPE e1 <</>> evalMPE e2

(<</>>) :: (ErrorMonad m, PrintMonad m)
             => m Float -> m Float -> m Float
m1 <</>> m2 = m1 >>= \v1 ->
              m2 >>= \v2 ->
              if v2==0
               then throw "No puedo dividir por cero"
               else printf (armarDiv v1 v2) >>= \_ ->
                    return (v1./.v2)
```

Apéndice B: Módulos para el ejercicio 3

Archivo Main.hs:

```
import Monadas -- Las definiciones de clases necesarias
               --   (mayor que la parte1)
import RNExp   -- El modelo de ejemplo (con la función a COMPLETAR)

main = do putStr "El resultado de (x+1) cuando x es 16 es: "
          print ej1
          putStr "El resultado de (x+1) cuando x es 41 es: "
          print ej2
```

Archivo Monadas.hs:

```
module Monadas where

class Monad m => ErrorMonad m where
  throw :: String -> m a

class Monad m => PrintMonad m where
  printf :: String -> m ()

class Monad m => ReaderMonad r m | m -> r where
  ask   :: m r
  runRM :: m a -> r -> a
```

Archivo RNExp.hs:

```
module RNExp where

import Monadas
import ReaderMonad  -- La mónada Reader
import Mem          -- La clase de las memorias (¿es un TAD?)
import ListMem      -- La memoria hecha con listas

data NExp = Var Variable
          | NCte Int
          | UOp UnOp NExp
          | BOp BinOp NExp NExp

data UnOp = Neg | Sqr
data BinOp = Add | Mul | Div | Mod

ejNE = BOp Add (Var "x") (NCte 1)

ejM1, ejM2 :: ListMem
```

```
ejM1 = recordar "x" 16 enBlanco
ejM2 = recordar "x" 41 enBlanco

-- Esta es otra forma de especializar una función sobrecargada
--  para usar un tipo específico:
--  (la declaración del tipo ES FUNDAMENTAL para que esto sirva)
miEvalNE :: NExp -> Reader ListMem Int
miEvalNE e = evalNE e

-- Funciona porque miEvalNE NO es genérica (con evalNE da error,
--   porque no puede saber qué mónada queremos para reader)
ej1 = runRM (miEvalNE ejNE) ejM1
ej2 = runRM (miEvalNE ejNE) ejM2

-- COMPLETAR para que evalNE sea una función total que implemente
--  el significado de su argumento como una mónada reader
evalNE :: (Mem mem, ReaderMonad mem m) => NExp -> m Int
evalNE _ = error "Impleméntenme"
-- OBSERVAR que la mónada m DEBE tener como contexto a una memoria
-- OBSERVAR también que ni la mónada ni la memoria pueden
--          estar fijos

evalUO Neg x = return (-x)
evalUO Sqr x = return (x^2)

evalBO Add x y = return (x+y)
evalBO Mul x y = return (x*y)
evalBO Div x y = return (div x y)
evalBO Mod x y = return (mod x y)
```

Archivo ReaderMonad.hs:

```
module ReaderMonad where
import Monadas

data Reader r a = R (r -> a)

instance Monad (Reader r) where
  return x = R (\r -> x)
  m >>= k  = R (\r -> let R fm = m
                      in let R fk = k (fm r)
                          in fk r)

instance ReaderMonad r (Reader r) where
  ask = R (\r -> r)
  runRM m r = let R fm = m
              in fm r
```

Archivo Mem.hs:

```
module Mem where
-- La definición del TAD Mem
type Variable = String

class Mem mem where
  enBlanco   :: mem
  cuantoVale :: Variable -> mem -> Maybe Int
  recordar   :: Variable -> Int -> mem -> mem
```

Archivo ListMem.hs:

```
module ListMem where
-- Una posible implementación del TAD Mem
import Mem
data ListMem = LM [(Variable, Int)]

instance Mem ListMem where
  enBlanco              = LM []
  cuantoVale v (LM vns) = lookup v vns
  recordar v n (LM vns) = LM (recordarRep v n vns)

  recordarRep v n []            = [(v,n)]
  recordarRep v n ((v',n'):vns) =
     if v==v'
      then (v',n)  : vns
      else (v',n') : recordarRep v n vns
```