1. Demostrar que los siguientes tipos son monadas:

```
newtype Id a = Id a
data Maybe a = Nothing | Just a
```

Es decir:

a) Dar la instancia de Monad para cada uno de ellos.

b) Demostrar que para cada instancia valen las leyes de las monadas.

Soluciones

a)

- `return a = Id a`
- `Id a >>= f = f a`

- `return a = Just a`
- `Nothing >>= f = Nothing`
  `Just a >>= f = f a`

---

b)

- `return a >>= f`
  $\equiv \langle def.return \rangle$
  `Id a >>= f`
  $\equiv \langle def. >>= \rangle$
  `f a`
- `Id a >>= return`
  $\equiv \langle def. >>= \rangle$
  `return a`
  $\equiv \langle def.return \rangle$
  `Id a`
- `(Id a >>= f) >>= g`
  $\equiv \langle def. >>= \rangle$
  `(f a) >>= g`
  $\equiv \langle def. >>= \rangle$
  `Id a >>= (\x -> f x >>= g)`

- `return a >>= f`

  $\equiv \langle def.return \rangle$

  `Just a >>= f`

  $\equiv \langle def. >>= \rangle$

  `f a`
- `Nothing >>= return`

  $\equiv \langle def. >>= \rangle$

  `Nothing`
- `Just a >>= return`

  $\equiv \langle def. >>= \rangle$

  `return a`

  $\equiv \langle def.return \rangle$

  `Just a`
- `(Nothing >>= f) >>= g`

  $\equiv \langle def. >>= \rangle$

  `Nothing >>= g`

  $\equiv \langle def. >>= \rangle$

  `Nothing`

  $\equiv \langle def. >>= \rangle$

  `Nothing >>= (\x -> f x >>= g)`
- `(Just a >>= f) >>= g`

  $\equiv \langle def. >>= \rangle$

  `f a >>= g`

  $\equiv \langle def. >>= \rangle$

  `Just a >>= (\x -> f x >>= g)`

2. Demostrar que el constructor de tipo [] (lista) es una monada.

Solucion

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat $ map f xs
```

- `return x >>= f`

  $\equiv \langle def.return \rangle$

  `[x] >>= f`

  $\equiv \langle def. >>= \rangle$

  `concat $ map f [x]`

  $\equiv \langle def.map \rangle$

  `concat [f x]`

  $\equiv \langle def.concat \rangle$

  `f x`

- `xs >>= return`

  $\equiv \langle def. >>= \rangle$

  `concat $ map return xs`

  $\equiv \langle def.return \rangle$

  `concat $ [xs]`

  $\equiv \langle def.concat \rangle$

  `xs`

- COMPLETAR.

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

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

   a) Probar que la instancia efectivamente define una monada.

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

Soluciones

a)

- `return x >>= f`

  $\equiv \langle def.return \rangle$

  ```
  St (\s -> (x, s)) >>= f
  ```

  $\equiv \langle def. >>= \rangle$

  ```
  St (\s -> let (x, s') = (\s -> (x, s)) s
            in runState (f x) s')
  ```

  $\equiv_\beta$

  ```
  St (\s -> let (x, s') = (x, s)
            in runState (f x) s')
  ```

  $\equiv \langle def.let \rangle$

  ```
  St (\s -> runState (f x) s)
  ```

  $\equiv_\eta$

  ```
  St (runState (f x))
  ```

  $\equiv \langle St \circ Runstate \equiv id \rangle$

  ```
  f x
  ```

- `(St h) >>= return`

  $\equiv \langle def. >>= \rangle$

  ```
  St (\s -> let (x, s') = h s
            in runState (return x) s')
  ```

  $\equiv \langle def.return \rangle$

  ```
  St (\s -> let (x, s') = h s
            in runState (St (\s -> (x, s))) s')
  ```

  $\equiv \langle def.runState \rangle$

  ```
  St (\s -> let (x, s') = h s
            in (\s -> (x, s)) s')
  ```

  $\equiv_\beta$

  ```
  St (\s -> let (x, s') = h s
            in (x, s'))
  ```

  $\equiv \langle def.let \rangle$

  ```
  St (\s -> h s)
  ```

  $\equiv_\eta$

  ```
  St h
  ```

- COMPLETAR

b)

- `set s = St (\s' -> ((), s))`
- `get = St (\s -> (s, s))`

4. Dado el tipo `data Tree a = Leaf a | Branch (Tree a) (Tree a)` y su correspondiente instancia de Functor:

```
instance Functor Tree where
  fmap f (Leaf a)     = Leaf (f a)
  fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
```

a) La funcion `numTree :: Tree a -> Tree Int`, permite numerar las hojas de un arbol de izquierda a derecha. Definir la funcion auxiliar:

```
mapTreeNro :: (a -> Int -> (b, Int)) -> Tree a -> Int -> (Tree b, Int)
```

de forma que la siguiente definicion de `numTree` sea correcta:

```
numTree :: Tree a -> Tree Int
numTree t = fst (mapTreeNro update t 0) where update a n = (n, n+1)
```

b) Para generalizar el caso del item anterior, se puede pensar que en lugar *Int* se lleva un estado de tipo *s*, quedando una funcion con la forma:

```
mapTreeSt :: (a -> s -> (b, s)) -> Tree a -> s -> (Tree b, s)
```

Esto conduce directamente a la utilizacion de la monada State s con la siguiente funcion:

```
mapTreeM :: (a -> State s b) -> Tree a -> State s (Tree b)
```

Definir con notacion do la funcion `mapTreeM`.

Soluciones

a)
```
mapTreeNro f (Leaf a) n = let (b, i) = f a n
                          in (Leaf b, i)
mapTreeNro f (Branch l r) n = let (l', i) = mapTreeNro f l n
                                  (r', j) = mapTreeNro f r i
                              in (Branch l' r', j)
```

b)
```
mapTreeM f (Leaf a) = do b <- f a
                         return $ Leaf b
mapTreeM f (Branch l r) = do l' <- mapTreeM f l
                             r' <- mapTreeM f r
                             return $ Branch l' r'
```

5. La clase Monoid clasifica los tipos que son monoides y esta definida de la siguiente manera:

```
class Monoid m where
  mempty  :: m
  mappend :: m -> m -> m
```

Se requiere que las instancias hagan cumplir que mappend sea asociativa, y que mempty sea un elemento neutro de mappend por izquierda y derecha.

a) Probar que *String* es un monoide.

b) Probar que el siguiente constructor de tipos es una monada, (asumiendo que el parametro w es un monoide):

```
newtype Output w a = Out (a, w)
```

c) Dar una instancia diferente de Monad para el mismo tipo. Esto prueba que un mismo tipo de datos puede tener diferentes instancias de monadas.

d) Definir una operacion `write :: Monoid w => w -> Output w ()`.

e) Usando Output String, modificar el evaluador monadico basico de la teoria para agregar una traza de cada operacion. Por ejemplo:

```
> eval (Div (Con 14) (Con 2))
El termino (Con 14) tiene valor 14.
El termino (Con 2) tiene valor 2.
El termino (Div (Con 14) (Con 2)) tiene valor 7.
7.
```

Soluciones

a)
```
instance Monoid String where
  mempty            = []
  mappend mempty ys = ys
  mappend (x:xs) ys = x:(xs `mappend` ys)
```

- `mempty `mappend` xs`

  $\equiv \langle def.mappend \rangle$

  `xs`

-
  - `mempty `mappend` mempty`
    $\equiv \langle def.mappend \rangle$
    `mempty`
  - `(x:xs) `mappend` mempty`
    $\equiv \langle def.mappend \rangle$
    `x:(xs `mappend` mempty)`
    $\equiv \langle H.I. \rangle$
    `x:xs`
-
  - `(mempty `mappend` ys) `mappend` zs`
    $\equiv \langle monoid.law.1 \rangle$
    `ys `mappend` zs`
    $\equiv \langle monoid.law.1 \rangle$
    `mempty `mappend` (ys `mappend` zs)`
  - `((x:xs) `mappend` ys) `mappend` zs`
    $\equiv \langle def.mappend \rangle$
    `(x: (xs `mappend` ys)) `mappend` zs`
    $\equiv \langle def.mappend \rangle$
    `x: ((xs `mappend` ys) `mappend` zs)`
    $\equiv \langle H.I. \rangle$
    `x: (xs `mappend` (ys `mappend` zs))`
    $\equiv \langle def.mappend \rangle$
    `(x:xs) `mappend` (ys `mappend` zs))`

b)
```
instance Monoid w => Monad (Output w) where
  return x = Out (x, mempty)
  Out (a, w) >>= f = let Out (a' ,w') = (f a)
                     in Out (a', w `mappend` w')
```

```
return x >>= f
```

$\equiv \langle def.return \rangle$

```
Out (x, mempty) >>= f
```

$\equiv \langle def. >>= \rangle$

```
let Out (a' ,w') = (f x)
in Out (a', mempty `mappend` w')
```

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

```
let Out (a' ,w') = (f x)
in Out (a', w')
```

$\equiv \langle def.let \rangle$

```
f x
```

```
Out (a, w) >>= return
```

$\equiv \langle def. >>= \rangle$

```
let Out (a' ,w') = (return a)
in Out (a', w `mappend` w')
```

$\equiv \langle def.return \rangle$

```
let Out (a' ,w') = Out (a, mempty)
in Out (a', w `mappend` w')
```

$\equiv \langle def.let \rangle$

```
Out (a, w `mappend` mempty)
```

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

```
Out (a, w)
```

```
(Out (a1, w1) >>= f) >>= g
```

$\equiv \langle def. >>= \rangle$

```
(let Out (a2, w2) = f a1
 in Out (a2, w1 `mappend` w2)) >>= g
```

$\equiv \langle def.let \rangle$

```
let Out (a3, w3) = let Out (a2, w2) = f a1
                   in Out (a2, w1 `mappend` w2)
in Out (a3, w3) >>= g
```

$\equiv \langle def. >>= \rangle$

```
let Out (a3, w3) = let Out (a2, w2) = f a1
                   in Out (a2, w1 `mappend` w2)
in let Out (a4, w4) = g a3
   in Out (a4, w3 `mappend` w4)
```

$\equiv \langle prop.let \rangle$

```
let Out (a2, w2) = f a1
    Out (a3, w3) = Out (a2, w1 `mappend` w2)
in let Out (a4, w4) = g a3
   in Out (a4, w3 `mappend` w4)
```

$\equiv \langle prop.let \rangle$

```
let Out (a2, w2) = f a1
    Out (a3, w3) = Out (a2, w1 `mappend` w2)
    Out (a4, w4) = g a3
in Out (a4, w3 `mappend` w4)
```

$\equiv \langle def.a3, w3 \rangle$

```
let Out (a2, w2) = f a1
    Out (a4, w4) = g a2
in Out (a4, (w1 `mappend` w2) `mappend` w4)
```

$\equiv \langle monoid.law.2 \rangle$

```
let Out (a2, w2) = f a1
    Out (a4, w4) = g a2
in Out (a4, w1 `mappend` (w2 `mappend` w4))
```

$\equiv \langle def.a3, w3 \rangle$

```
let Out (a2, w2) = f a1
    Out (a4, w4) = g a2
    Out (a3, w3) = Out (a4, w2 `mappend` w4)
in Out (a3, w1 `mappend` w3)
```

$\equiv \langle prop.let \rangle$

```
let Out (a3, w3) = let Out (a2, w2) = f a1
                       Out (a4, w4) = g a2
                   in Out (a4, w2 `mappend` w4)
in Out (a3, w1 `mappend` w3)
```

$\equiv \langle prop.let \rangle$

```
let Out (a3, w3) = let Out (a2, w2) = f a1
                   in let Out (a4, w4) = g a2
                      in Out (a4, w2 `mappend` w4)
in Out (a3, w1 `mappend` w3)
```

$\equiv_\beta \langle def. >>= \rangle$

```
Out (a1, w1) >>= (\x -> let Out (a2, w2) = f x
                        in let Out (a4, w4) = g a2
                           in Out (a4, w2 `mappend` w4)
```

$\equiv \langle def. >>= \rangle$

```
Out (a1, w1) >>= (\x -> let Out (a2, w2) = f x
                        in Out (a2, w2) >>= g)
```

$\equiv \langle def.let \rangle$

```
Out (a1, w1) >>= (\x -> f x >>= g)
```

c)
```
instance Monoid w => Monad (Output w) where
  return x = Out (x, mempty)
  Out (a, w) >>= f = let Out (a', w') = (f a)
                     in Out (a', w' `mappend` w)
```

d)
```
write w = Out ((), w)
```

e)
```
trace :: (Show t, Show m) => t -> m -> Output String ()
trace t m = write $ "El termino "    ++ (show t) ++
                    " tiene valor " ++ (show m) ++ "\n"

eval :: Exp -> Output String Int
eval (Const n)   = return n
eval (Plus t u)  = do m <- eval t
                      trace t m
                      n <- eval u
                      return (m+n)
eval (Div t u)   = do m <- eval t
                      trace t m
                      n <- eval u
                      return (m+n)

doEval :: Exp -> IO ()
doEval e = let Out (a, w) = eval e
           in putStr $ w ++ (show a) ++ "\n"
```

6. Sea M una monada. Dados los siguientes operadores:

```
(>>)  :: M a -> M b -> M b
(>>=) :: M a -> (a -> M b) -> M b
```

   a) De ser posible, escribir >> en funcion de >>=.

   b) De ser posible, escribir >>= en funcion de >>.

   Soluciones

   a) `x >> y = x >>= (\_ -> y)`

   b) No es posible.

7. Las siguientes funciones se definen sobre un constructor de tipos m arbitrario:

   a) Definir la funcion `sequence :: Monad m => [m a] -> m [a]`, de una manera tal que `sequence xs` evalue todos los valores monadicos en la lista xs, de izquierda a derecha, y devuelva un valor en la misma monada que calcule la lista de los «resultados» de dichas evaluaciones. Por ejemplo si $xs = [m1, m2]$ entonces:

```
sequence xs = m1 >>= \x1 ->
              m2 >>= \x2 ->
              return [x1, x2]
```

   b) Definir la funcion `liftM :: Monad m => (a -> b) -> m a -> m b`, tal que `liftM f x` aplique f al contenido de x dentro de la monada m. Por ejemplo `liftM sum (Just [1..10]) = Just 55`. Notar que $liftM$ coincide con $fmap$ (ejercicio de la practica anterior).

   c) Definir la funcion `liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c`, tal que `liftM2 f m1 m2` aplique la funcion binaria f a los contenidos de $m1$ y $m2$ dentro de la monada m. Por ejemplo:

```
liftM2 (&&) [True, False] [True, True] = [True, True, False, False]
```

   d) Expresar *sequence* como un $foldr$. (Sugerencia: usar $liftM2$).

   Soluciones

   a) `sequence = mapM id`

   b) `liftM f x = x >>= (return . f)`

c)
```
liftM2 f ma mb = do a <- ma
                    b <- mb
                    return $ f a b
```

d) `sequence = foldr (liftM2 (:)) (return [])`

8. Dado el siguiente tipo de datos: `data Error e a = Raise e | Return a`:

   a) Demostrar que es una monada.

   b) Dar una definicion total de las siguientes funciones, utilizando la monada *Error String*:

      1) *shead* y *stail*, correspondientes a las operaciones sobre listas.

      2) *spush* y *spop*, correspondientes a las operaciones sobre pilas.

Soluciones

a)
```
instance Monad (Error e) where
  return         = Return
  Raise  x >>= _ = Raise x
  Return x >>= f = f x
```

- `return x >>= f`

  $\equiv \langle def.return \rangle$

  `Return x >>= f`

  $\equiv \langle def. >>= \rangle$

  `f x`

- `Raise x >>= return`

  $\equiv \langle def. >>= \rangle$

  `Raise x`

- `Return x >>= return`

  $\equiv \langle def. >>= \rangle$

  `return x`

  $\equiv \langle def.return \rangle$

  `Return x`

- COMPLETAR.

b)

1)

- ```
  shead [] = Raise "Head de lista vacia."
  shead (x:xs) = Return x
  ```
- ```
  stail [] = Raise "Tail de lista vacia."
  stail (x:xs) = Return xs
  ```

2)

- ```
  spush x p = do p' <- p
                 return $ x:p'
  ```
- ```
  spop p = do p' <- p
              h <- shead p'
              t <- stail p'
              return $ (h, t)
  ```

9. Se desea implementar un evaluador para un lenguaje sencillo, cuyos terminos seran 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 division por cero. Se plantea el siguiente tipo de datos para representar una monada de evaluacion:

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

y entonces el evaluador puede escribirte de esta manera:

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

y el computo resultante se podria ejecutar mediante una funcion auxiliar:

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

a) Dar la instancia de la monada $M\ s\ e$.

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

c) Reescribir *eval*, sin usar notacion do y luego expandir las definiciones de >>=, *return*, *raise* y *modify*, para obtener un evaluador no monadico.

Soluciones

a)
```
instance Monad (M s e) where
  return x = M (\s -> Return (x, s))
  M f >>= g = M (\s-> let m = f s
                      in m >>= \(a, s') -> runM (g a) s')
```

b)

- ```
  raise :: a -> M b a b
  raise a = M (_ -> Raise a)
  ```
- ```
  modify :: (s -> s) -> M s e ()
  modify f = M (\s -> Return ((), f s))
  ```

c)

- ```
  eval (Con n)     = return n
  eval (Div t1 t2) = eval t1 >>= \v1 ->
                     eval t2 >>= \v2 ->
                     if v2 == 0
                     then raise "Error: Division por cero."
                     else (modify (+1)) >> (return $ div v1 v2)
  ```
- COMPLETAR.

10. El tipo de datos `Cont r a` representa *continuaciones* en las que dado el resultado de una funcion (de tipo a) y la continuacion de la computacion ($a \to r$), devuelve un valor en r. Probar que `Cont r` es una monada. Ayuda: Guiarse con los tipos.

```
data Cont r a = C ((a -> r) -> r)
```

Solucion

```
instance Monad (Cont r) where
  return x  = C (\f -> f x)
  C f >>= g = C (\br -> f (\a -> let C h = g a
                                 in h br))
```
```

- COMPLETAR.
- COMPLETAR.
- COMPLETAR.

11. Dado el siguiente tipo de datos: `data M m a = Mk (m (Maybe a))`

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

    b) Definir una operacion auxiliar `throw :: Monad m => M m a` que lanza una excepcion.

    c) Dada la monada de estado $StInt$ y el siguiente tipo N:

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

    Definir opreaciones `get :: N Int` y `put :: Int -> N ()`, que lean y actualizen (respectivamente) el estado de la monada.

    d) Usando N, definir un interprete monadico para un lenguaje de expresiones aritmeticas y una sola variable dado por el siguiente AST:

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

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

Soluciones

a)

- ```haskell
  instance Monad m => Monad (M m) where
    return x = Mk (return (Just x))
    Mk x >>= f = Mk $ do x' <- x
                         case x' of
                           Nothing -> return Nothing
                           Just a  -> let Mk y = f a in y
  ```
- COMPLETAR.
- COMPLETAR.
- COMPLETAR.

b) `throw = Mk (return Nothing)`

c)

- `get = Mk (St (\s -> (Just s, s)))`
- `put s = Mk (St (\_ -> (Just (), s)))`

d)
```haskell
eval :: Expr -> N Int
eval Var        = get
eval (Cont x)   = return x
eval (Let x y)  = do s <- eval x
                     put s
                     eval y
eval (Add x y)  = do x' <- eval x
                     y' <- eval y
                     return $ x' + y'
eval (Div x y)  = do x' <- eval x
                     y' <- eval y
                     if y' == 0 then throw
                                else return $ div x' y'

doEval :: Expr -> (Maybe Int, Int)
doEval e = let Mk (St runSt) = eval e
           in runSt 0
```
```