Programación Funcional Avanzada

Aplicando Monads – Máquina Virtual Logo
Monad Reader

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Logo

- Lenguaje de alto nivel
  - Imperativo procedural.
  - Iteración determinada e indeterminada.
  - Procedimientos.
  - Recursión.
  - I/O simple.
- Orientado a la enseñanza de niños y jóvenes
  - Los programas mueven una "tortuga".
  - La tortuga tiene lápices de colores.
  - El "rastro" permite construir dibujos.

UNIVERSIDAD SIMÓN BOLÍVAR

`MiniLogo` – Nuestro proyecto de hoy

- Implantar un subconjunto de Logo en forma de "Máquina Virtual"
  - Expresión del programa separado de su interpretación.
  - Interpretación del programa separado de los efectos gráficos.
  - Extensible con facilidad.

UNIVERSIDAD SIMÓN BOLÍVAR

`MiniLogo` – Nuestro proyecto de hoy

- Implantar un subconjunto de Logo en forma de "Máquina Virtual"
  - Expresión del programa separado de su interpretación.
  - Interpretación del programa separado de los efectos gráficos.
  - Extensible con facilidad.
- Aplicando los principios de progamación funcional
  - Programas Logo – tipo de datos abstracto Haskell.
    - Explotar `Data.Map` y `Data.Seq`
    - Explotar genericidad de `Functor` y `Foldable`.
  - Interpretación – transformación sobre un Monad `State`.
  - Efectos gráficos – librería `HGL`.

UNIVERSIDAD SIMÓN BOLÍVAR

Resultado final

El programa ...

```
star = Seq $ DS.fromList [
  Pd, Pc "Yellow",
  Rep 36 $ DS.fromList [
    Rep 4 $ DS.fromList [
      Fd 200, Rt 90
    ], Rt 10
  ]
]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Resultado final

El programa ...

```
star = Seq $ DS.fromList [
  Pd, Pc "Yellow",
  Rep 36 $ DS.fromList [
    Rep 4 $ DS.fromList [
      Fd 200, Rt 90
    ], Rt 10
  ]
]
```

...se interpreta ...

```
> runLogoProgram
    600 600
    "MiniLogo"
    star
```

UNIVERSIDAD SIMÓN BOLÍVAR

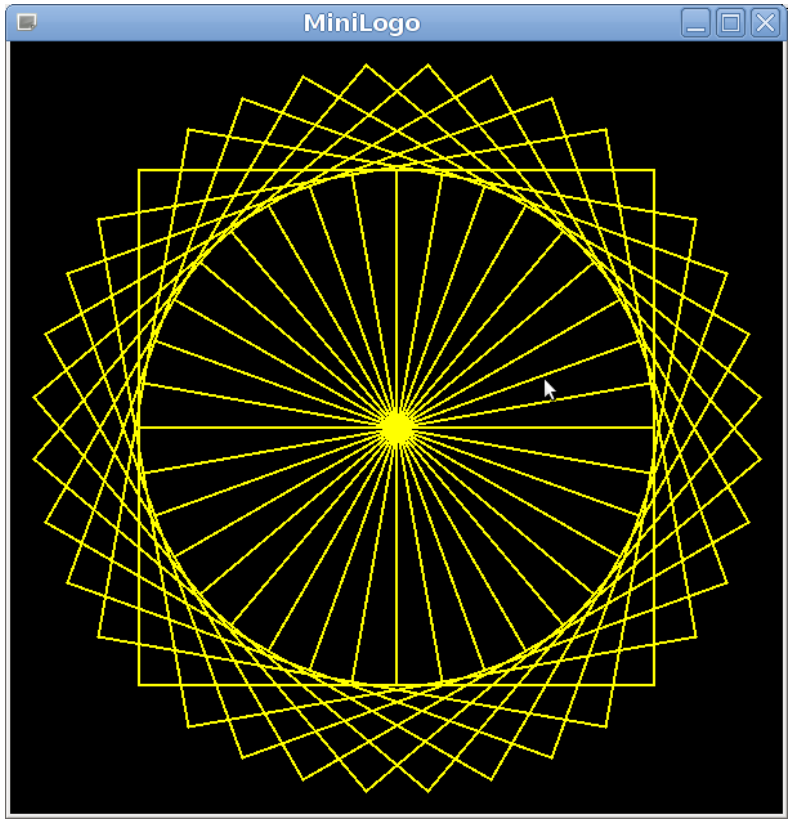
Resultado final

El programa ...

```
star = Seq $ DS.fromList [
  Pd, Pc "Yellow",
  Rep 36 $ DS.fromList [
    Rep 4 $ DS.fromList [
      Fd 200, Rt 90
    ], Rt 10
  ]
]
```

...se interpreta ...

```
> runLogoProgram
    600 600
    "MiniLogo"
    star
```



UNIVERSIDAD SIMÓN BOLÍVAR

Expresando los programas MiniLogo

```
import Data.Sequence as DS

data LogoProgram = Fd Int              -- En pixels
                 | Bk Int              -- En pixels
                 | Rt Int              -- En grados
                 | Lt Int              -- En grados
                 | Pu
                 | Pd
                 | Pc String
                 | Say String
                 | Home
                 | Seq (DS.Seq LogoProgram)
                 | Rep Int (DS.Seq LogoProgram)
                deriving (Show, Eq)
```

- Más instrucciones – más constructores.
- Estructura de bloques anidados.

UNIVERSIDAD SIMÓN BOLÍVAR

Primer desvío – manejo de colores

- Tabla de "Colores Válidos" – ¡mutable!
- `MiniLogo` usa `String` – internamente cambia a `Color`.

```
import qualified Data.Map      as DM
import qualified Graphics.HGL as G

validColors :: DM.Map String G.Color
validColors = DM.fromList [ ("red", G.Red), ... ]

toColor :: String -> G.Color
toColor s = case f of
  Just c  -> c
  Nothing -> error $
      "'" ++ s ++ "' es un color invalido"
  where f = DM.lookup (map toLower s) validColors
```

UNIVERSIDAD SIMÓN BOLÍVAR

El interpretador

- Estado inicial.
  - Centro de la ventana.
  - Lápiz blanco levantado.
  - Orientado hacia “arriba” (90 grados).

El interpretador

- Estado inicial.
  - Centro de la ventana.
  - Lápiz blanco levantado.
  - Orientado hacia "arriba" (90 grados).
- Cada instrucción transforma el estado.
- Algunas instrucciones producen efectos
  - Si nos "metemos" en el Monad `IO` no podemos escapar ...
  - Construiremos el dibujo y lo presentaremos al final

UNIVERSIDAD SIMÓN BOLÍVAR

El interpretador

- Estado inicial.
  - Centro de la ventana.
  - Lápiz blanco levantado.
  - Orientado hacia "arriba" (90 grados).
- Cada instrucción transforma el estado.
- Algunas instrucciones producen efectos
  - Si nos "metemos" en el Monad `IO` no podemos escapar ...
  - Construiremos el dibujo y lo presentaremos al final
- Usaremos un Monad `State`
  - El estado combina la tortuga y el dibujo.
  - Transformadores para cada instrucción.
  - Ya sabemos como opera – descansaremos sobre la infraestructura de tipos y funciones provistas en `Control.Monad.State`

UNIVERSIDAD SIMÓN BOLÍVAR

Segundo desvío – los dibujos

- Como toda buena librería funcional, `HGL` separa su preocupaciones
  - Geometría – ¿**qué** dibujar?
  - Visualización – ¿**cómo** dibujarlo?
- `MiniLogo` solamente dibuja líneas y texto
  - `Point` (una tupla) – extremos de las líneas.
  - `HGL` provee polígonos y texto – usados directamente.
  - Nuestro interpretador construye el dibujo usando un tipo algebráico que pueda convertirse a esas primitivas.

UNIVERSIDAD SIMÓN BOLÍVAR

Modelando el estado de la máquina

```
type Direction    = Int
data PenStatus    = Up | Down
                  deriving (Show,Eq)

data Figure = Poly G.Color [G.Point]        -- poligono
            | Text G.Color G.Point String -- texto
            | Empty                       -- centinela
            deriving (Show,Eq)

data LogoState = LogoState {
   pos :: G.Point,              -- (x,y)
   dir :: Direction,            -- En grados
   pns :: PenStatus,
   pnc :: G.Color,
   drw :: DS.Seq Figure
} deriving (Show)
```

UNIVERSIDAD SIMÓN BOLÍVAR

La magia de `Control.Monad.State`

- Modelo para el estado – ¡envoltorio automático!

```
newtype State s a = State {runState :: s -> (a,s)}
instance Monad (State s) where
  return a = ...
  (>>=)    = ...
```

UNIVERSIDAD SIMÓN BOLÍVAR

La magia de `Control.Monad.State`

- Modelo para el estado – ¡envoltorio automático!

```
newtype State s a = State {runState :: s -> (a,s)}
instance Monad (State s) where
  return a = ...
  (>>=)    = ...
```

- Dado el envoltorio, acceso automático al estado

```
class (Monad m) => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
```

UNIVERSIDAD SIMÓN BOLÍVAR

La magia de `Control.Monad.State`

- Modelo para el estado – ¡envoltorio automático!

```
newtype State s a = State {runState :: s -> (a,s)}
instance Monad (State s) where
  return a = ...
  (>>=)    = ...
```

- Dado el envoltorio, acceso automático al estado

```
class (Monad m) => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
```

- Así, si se escribe código monádico con la *firma* adecuada, la librería instancia el Monad `State` **automágicamente**

```
meh :: Int -> State Foo Bar
meh n = do ...
```

  - `Foo` y `Bar` – tipos para el estado y resultados.
  - En el código mónadico podemos usar `get` y `put`.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `State`

A escribir transformadores ...

- Modelaremos el estado (`s`) usando `LogoState`.
- La máquina no produce resultados (`a`) – usaremos `()`.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `State`

A escribir transformadores ...

- Modelaremos el estado (`s`) usando `LogoState`.
- La máquina no produce resultados (`a`) – usaremos `()`.
- Combinador trivial que no hace nada

```
noop :: State LogoState ()
noop = return ()
```

...ya verán su utilidad más adelante.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `State`

A escribir transformadores ...

- Modelaremos el estado (`s`) usando `LogoState`.
- La máquina no produce resultados (`a`) – usaremos `()`.
- Combinador trivial que no hace nada

```
noop :: State LogoState ()
noop = return ()
```

...ya verán su utilidad más adelante.

- Combinador para cambiar el color del lápiz

```
pc :: String -> State LogoState ()
pc c = do
  s <- get
  put $ s { pnc = toColor c }
```

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Subir el lápiz

```
pu :: State LogoState ()
pu = do
  s <- get
  case pns s of
    Down -> put $ s { pns = Up, drw = drw' }
            where drw' = case d of
                           (ds :> Empty) -> ds
                           _             -> drw s
                         where d = DS.viewr $ drw s

    Up   -> put $ s
```

- `Empty` es un centinela – "terminó el dibujo en curso".
- Aprovecho `ViewR` para examinar y descomponer el extremo derecho del `Data.Sequence`.

Usando Monad State

Bajar el lápiz

```
pd :: State LogoState ()
pd = do
  s <- get
  case pns s of
    Down -> put $ s
    Up   -> put $ s { pns = Down,
                      drw = (drw s) |> Empty }
```

- `Empty` es un centinela – "terminó el dibujo en curso".
- Agrego al extremo derecho del `Data.Sequence`.
- El `drw` izquierdo construye el nuevo valor –
  el `drw` derecho obtiene el valor actual.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Escribiendo texto

```
say :: String -> State LogoState ()
say m = do
  s <- get
  case pns s of
    Down -> case d of
      (ds :> Empty) -> put $ s { drw = ds       |> t }
      _             -> put $ s { drw = (drw s) |> t }
      where d = DS.viewr $ drw s
            t = Text (pnc s) (pos s) m
    Up   -> put $ s
```

- HGL se encargará de dibujar el texto.
- Empty es un centinela – "terminó el dibujo en curso".
- Aprovechamos todo lo que nos ofrece Data.Sequence – examinar y descomponer por derecha, agregar a la derecha.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Girar a izquierda y derecha en grados

```
lt :: Int -> State LogoState ()
lt n = get >>= put . turnLeft n

rt :: Int -> State LogoState ()
rt n = get >>= put . turnLeft (negate n)

turnLeft :: Int -> LogoState -> LogoState
turnLeft n s = s { dir = (dir s + n) `mod` 360 }
```

- Típico caso idiomático para usar »= en lugar de do monádico – el estado pasa directo hacia la función auxiliar.
- Aritmética modular para ignorar múltiples “vueltas”.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

Avanzar y retroceder

```
fd :: Int -> State LogoState ()
fd n = get >>= put . moveForward n

bk :: Int -> State LogoState ()
bk n = get >>= put . moveForward (negate n)
```

- Típico caso idiomático para usar »= en lugar de do monádico – el estado pasa directo hacia la función auxiliar.
- Moverse n pasos depende del *ángulo* hacia el cual apunta la tortuga, cosa que calcularemos en moveForward ...

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

La función auxiliar – la más complicada del programa

```
moveForward :: Int -> LogoState -> LogoState
moveForward n s | pns s == Up =
  s { pos = move (pos s) n (dir s) }
moveForward n s = case d of
  (ds :> Empty)      -> s { pos = np, drw = ds |> t}
  (ds :> Poly pc l) -> if (pc == cc)
    then s { pos = np, drw = ds |> Poly cc (np:l)  }
    else s { pos = np, drw = drw s |> t  }
  _                  -> s { pos = np, drw = drw s |> t
}
  where cc = pnc s
        cp = pos s
        d  = DS.viewr $ drw s
        np = move cp n (dir s)
        t  = Poly cc [ np, cp ]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

...el resto es trignonometría

```
move :: (Int,Int) -> Int -> Int -> (Int,Int)
move (x,y) n d =
  let direc  = (pi * (fromIntegral d)) / 180
      nn     = fromIntegral n
      nx     = x + (round (nn * (cos direc)))
      ny     = y + (round (nn * (sin direc)))
  in (nx,ny)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad State

El estado inicial

```
initial :: LogoState
initial = LogoState { pos = (0,0),
                      dir = 90,
                      pns = Up,
                      pnc = White,
                      drw = DS.empty }
```

Es un valor simple – el Monad lo transformará

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `state`

Combinadores sobre el estado inicial

```
home :: State LogoState ()
home = put $ initial

goHome :: State LogoState ()
goHome = do
  s <- get
  put $ s { pos = (0,0), dir = 90 }
```

- `home` – establece el estado inicial.
- `goHome` – restablece la *actitud* inicial

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Monad `State`

El combinador para repetición

```
repN :: Int -> State LogoState () -> State LogoState ()
repN 0 p = noop
repN n p = p >> (repN (pred n) p)
```

- Debe repetir una transformación de estado n veces.
- Repetir 0 veces es "no hacer nada" – `noop`
- Repetir n veces es "hacer una" y repetir $n - 1$

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo interpretar el programa?

- El tipo `LogoProgram` representa nuestro programa `MiniLogo`.
- Combinadores monádicos correspondientes – transforman el estado cuando tiene sentido.
- ¿Como transformamos un `LogoProgram` arbitrario en la secuencia monádica correspondiente?

UNIVERSIDAD SIMÓN BOLÍVAR

¿Cómo interpretar el programa?

- El tipo `LogoProgram` representa nuestro programa `MiniLogo`.
- Combinadores monádicos correspondientes – transforman el estado cuando tiene sentido.
- ¿Como transformamos un `LogoProgram` arbitrario en la secuencia monádica correspondiente?

¡Con un catamorfismo sobre `LogoProgram`!

UNIVERSIDAD SIMÓN BOLÍVAR

Interpretar el programa

Catamorfismo sobre `LogoProgram` – `foldLP` escrito a mano obligatoriamente

```
foldLP a b c d e f g h i j k inst =
  case inst of
    (Fd n)     -> a n
    (Bk n)     -> b n
    (Rt n)     -> c n
    (Lt n)     -> d n
    Pu         -> e
    Pd         -> f
    (Pc s)     -> g s
    (Say s)    -> h s
    Home       -> i
    (Seq l)    ->
      j (fmap (foldLP a b c d e f g h i j k) l)
    (Rep n l) ->
      k n (fmap (foldLP a b c d e f g h i j k) l)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Interpretar el programa

. . . y armados con ese catamorfismo

```
monadicPlot :: LogoProgram -> State LogoState ()
monadicPlot =
  foldLP fd bk rt lt pu pd pc say home seq rep
  where seq s   = if DS.null s then noop
                                else DF.sequence_ s
        rep n s = repN n (seq s)

repN :: Int -> State LogoState () -> State LogoState ()
repN 0 p = noop
repN n p = p >> (repN (pred n) p)
```

- La conversión es directa – secuenciar operaciones monádicas.
- Usamos `noop` para el caso borde en que no haya operaciones.
- Aplicamos `sequence_` de `Data.Foldable` sobre `Data.Sequence`.

UNIVERSIDAD SIMÓN BOLÍVAR

Tercer desvío – ventanas y gráficos

Hundirnos en `IO` usando `HGL`

- `openWindow` abre una ventana con título y dimensiones.
  - `getKey` espera por alguna tecla sobre la ventana – hay más eventos, pero con este nos basta.
  - `closeWindow` cierra la ventana
- `drawInWindow` aplica geometría sobre una ventana.
  - `withColor` cambia el color de una geometría.
  - `polyline` genera un polígono.
  - `text` genera texto.
  - `overGraphics` combina varias geometrías en una sola.
- `runGraphics` efectúa el I/O de la geometría en la ventana.

UNIVERSIDAD SIMÓN BOLÍVAR

Algoritmo para la visualización

- Construir la ventana con las dimensiones y título solicitadas.
- Interpretar el programa suministrado
  - Establecer el estado inicial.
  - Aplicar la transformación del estado hasta el estado final

```
execState :: State s a -> s -> s
```

  - Obtener el dibujo generado.
- Convertir el dibujo a geometría de `HGL` – sólo hay polígonos y texto.
  - `MiniLogo` tiene origen en el centro –
    `HGL` tiene origen en el vértice superior izquierdo.
  - Coordenada x crece hacia la derecha en ambos.
  - Coordenada y crece hacia arriba en `MiniLogo` y hacia abajo en `HGL`.
- Aplicar la geometría sobre la ventana y dibujarla.
- Esperar una tecla para terminar.

UNIVERSIDAD SIMÓN BOLÍVAR

Sólo nos queda dibujar ...

```
runLogoProgram w h t p =
  G.runGraphics $ do
    window <- G.openWindow t (w,h)
    G.drawInWindow window $ G.overGraphics (
      let f (Poly c p)    = G.withColor c $
                              G.polyline (map fix p)
          f (Text c p s) = G.withColor c $
                              G.text (fix p) s
          (x0,y0)          = origin w h
          fix (x,y)        = (x0 + x, y0 - y)
      in
      DF.toList $
      fmap f (drw (execState (monadicPlot p) initial))
      )
    G.getKey window
    G.closeWindow window
```

UNIVERSIDAD SIMÓN BOLÍVAR

Aspectos claves de la solución

Aspectos claves de la solución

- Generalización – instrucciones como un tipo de datos.
- Separación de preocupaciones – transformación vs. acción.

UNIVERSIDAD SIMÓN BOLÍVAR

Aspectos claves de la solución

- Generalización – instrucciones como un tipo de datos.
- Separación de preocupaciones – transformación vs. acción.
- Abstracción
  - La "plomería" en el Monad `State`.
  - La interpretación como un catamorfismo.
  - Explotar librerías concretas vía comportamientos genéricos.
- Encapsulamiento – el módulo sólo exporta el tipo de datos y la función de interpretación.

UNIVERSIDAD SIMÓN BOLÍVAR

Aspectos claves de la solución

- Generalización – instrucciones como un tipo de datos.
- Separación de preocupaciones – transformación vs. acción.
- Abstracción
  - La "plomería" en el Monad `State`.
  - La interpretación como un catamorfismo.
  - Explotar librerías concretas vía comportamientos genéricos.
- Encapsulamiento – el módulo sólo exporta el tipo de datos y la función de interpretación.
- Aislamiento – hacer I/O en el último momento posible.
  - `HGL` lo hace así.
  - ¡Nosotros también!

UNIVERSIDAD SIMÓN BOLÍVAR

Aspectos claves de la solución

- Generalización – instrucciones como un tipo de datos.
- Separación de preocupaciones – transformación vs. acción.
- Abstracción
  - La "plomería" en el Monad `State`.
  - La interpretación como un catamorfismo.
  - Explotar librerías concretas vía comportamientos genéricos.
- Encapsulamiento – el módulo sólo exporta el tipo de datos y la función de interpretación.
- Aislamiento – hacer I/O en el último momento posible.
  - `HGL` lo hace así.
  - ¡Nosotros también!

DRY – Don't Repeat Yourself

UNIVERSIDAD SIMÓN BOLÍVAR

Monad `Reader`

Ambiente constante "sólo lectura"

- ¿Y si el estado debe ser **inmutable** parcial o totalmente?
  - La configuración de una aplicación.
  - Monad `State` sirve, pero no impide cambios "accidentales".

Monad `Reader`

Ambiente constante "sólo lectura"

- ¿Y si el estado debe ser **inmutable** parcial o totalmente?
  - La configuración de una aplicación.
  - Monad `State` sirve, pero no impide cambios "accidentales".
- Si se tratara de una función sería `e -> a` ...
  - ... recibiría un "ambiente" – `e` por *environment*.
  - ... para calcular algún resultado – `a` arbitrario.
- El cómputo debe aprovechar el ambiente
  - Para obtener valores del ambiente – `ask`
  - Debe dejarlo disponible para el *próximo* cómputo en la secuencia.

Se repite la receta de `State`.

UNIVERSIDAD SIMÓN BOLÍVAR

Monad Reader

Cómputos sobre un ambiente constante

```
newtype Reader e a = Reader { runReader :: (e -> a) }

instance Monad (Reader e) where
    return a          = Reader $ \e -> a
    (Reader r) >>= f = Reader $ \e -> f (r e) e

class MonadReader e m | m -> e where
    ask   :: m e
    local :: (e -> e) -> m a -> m a

instance MonadReader (Reader e) where
    ask       = Reader id
    local f c = Reader $ \e -> runReader c (f e)

asks :: (MonadReader e m) => (e -> a) -> m a
asks sel = ask >>= return . sel
```

¡No entiendo »=!

Lo noté . . . A todos nos pasa

El inyector es bastante obvio, pero *bind* luce "raro" –
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

¡No entiendo »=!

Lo noté . . . A todos nos pasa

El inyector es bastante obvio, pero *bind* luce "raro" – veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- `r` está dentro del Reader – es una función `(e -> a)`

¡No entiendo »=!

Lo noté . . . A todos nos pasa

El inyector es bastante obvio, pero *bind* luce "raro" –
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- `r` está dentro del Reader – es una función `(e -> a)`
- Por tanto `(r e)` no es más que aplicarla – produce algo del tipo `a`.

UNIVERSIDAD SIMÓN BOLÍVAR

¡No entiendo »=!

Lo noté . . . A todos nos pasa

El inyector es bastante obvio, pero *bind* luce "raro" –
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- `r` está dentro del Reader – es una función `(e -> a)`
- Por tanto `(r e)` no es más que aplicarla – produce algo del tipo `a`.
- `f` es una función `(a -> Reader e a)` – se aplica sobre `(r e)`.

UNIVERSIDAD SIMÓN BOLÍVAR

¡No entiendo »=!

Lo noté . . . A todos nos pasa

El inyector es bastante obvio, pero *bind* luce "raro" –
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- `r` está dentro del Reader – es una función `(e -> a)`
- Por tanto `(r e)` no es más que aplicarla – produce algo del tipo `a`.
- `f` es una función `(a -> Reader e a)` – se aplica sobre `(r e)`.
- El resultado de `f (r e)` es un `Reader e a` –
  ¡eso es una función `(e -> a)` otra vez!

UNIVERSIDAD SIMÓN BOLÍVAR

¡No entiendo »=!

Lo noté . . . A todos nos pasa

El inyector es bastante obvio, pero *bind* luce "raro" –
veamos que no es para nada raro, paso a paso

```
(Reader r) >>= f = Reader $ \e -> f (r e) e
```

- `r` está dentro del Reader – es una función `(e -> a)`
- Por tanto `(r e)` no es más que aplicarla – produce algo del tipo `a`.
- `f` es una función `(a -> Reader e a)` – se aplica sobre `(r e)`.
- El resultado de `f (r e)` es un `Reader e a` –
  ¡eso es una función `(e -> a)` otra vez!
- Entonces `(f (r e))` se puede aplicar al `e`

Keep calm and Curry on. . .

UNIVERSIDAD SIMÓN BOLÍVAR

Monad Reader

Funciones auxiliares

- ask ofrece acceso al ambiente – en este caso un Int

```
> runReader (ask >>= \x -> return (x*3))
            14
42
```

UNIVERSIDAD SIMÓN BOLÍVAR

Monad Reader

Funciones auxiliares

- `ask` ofrece acceso al ambiente – en este caso un `Int`

```
> runReader (ask >>= \x -> return (x*3))
            14
42
```

- `local` permite ofrecer definiciones locales sobre el ambiente – piensen "alcance dinámico"

```
> runReader (ask >>=
             local (+9) \x -> return (x*3))
            14
69
```

  - Noten que el primer argumento de `local` es una *función*.
  - Esa función podría modificar el ambiente "temporalmente".

Monad Reader

```
import Data.Maybe
import qualified Data.Map as M
import Control.Monad.Reader

data VarType = TInt | TFloat deriving (Eq)
type Symbols = M.Map String VarType

isInteger :: String -> Reader Symbols Bool
isInteger sym = do
  t <- asks (lookupVar sym)
  return (t == TInt)

lookupVar :: String -> Symbols -> VarType
lookupVar n b = fromJust (M.lookup n b)

table = M.fromList [("foo",TInt), ("bar",TFloat)]
```

UNIVERSIDAD SIMÓN BOLÍVAR

Generalización de Monads

- `Prelude` ofrece lo mínimo para operar con Monads.
- Aprovechar los módulos específicos.
  - `Data.Maybe` – Monad `Maybe`
  - `Control.Monad` – Clases de tipos y combinadores generales
  - `Control.Monad.State` – Monad `State`
  - `Control.Monad.Reader` – Monad `Reader`
  - `Control.Monad.Writer` – Monad `Writer`
  - `Control.Monad.RWS` – Monad que combina `Reader`, `Writer` y `State` (esperen una semana antes de verla).
  - `System.IO` – Monad `IO` para I/O general
  - `System.Random` – Monad para RNG y PRNG.
- ...y muchos más.

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Documentación de `Graphics.HGL`
  Librería gráfica simple y agnóstica (X11 y Win32)
- Documentación de `Control.Monad.Reader`
- La solución "correcta" debería combinar Monad `State` y `Writer` – la próxima semana aprenderemos a construir Monad combinados.
- ¿Qué debe hacerse para extender el lenguaje y permitir una iteración controlada por un *predicado* arbitrario?
  - El predicado sería una función en Haskell.
  - Ahora puede mezclar Haskell y LogoMachine para escribir programas.
  - ¿Qué pasa si escribe un programa LogoMachine que nunca termina?

UNIVERSIDAD SIMÓN BOLÍVAR