

Llenguatges de Programació

## Sessió 5: mònades



Jordi Petit

**UNIVERSITAT POLITÈCNICA DE CATALUNYA**  
**BARCELONATECH**

**Facultat d'Informàtica de Barcelona**



# Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis

# Functors

Ja sabem aplicar funcions:

```
λ> (+3) 2
```



5

Però...

```
λ> (+3) (Just 2)
```



En aquest cas, podem fer servir `fmap`!

```
λ> fmap (+3) (Just 2)
```



Just 5

```
λ> fmap (+3) Nothing
```



Nothing

I també funciona amb `Either`, llistes, tuples i funcions:

```
λ> fmap (+3) (Right 2)
```



Right 5

```
λ> fmap (+3) (Left "err")
```



Left "err"

```
λ> fmap (+3) [1, 2, 3]
```



[4, 5, 6]

-- igual que map

```
λ> fmap (+3) (1, 2)
```



(1, 5)

-- perquè (,) és un tipus

```
λ> (fmap (*2) (+1)) 3
```



8

-- igual que (.)

# Functors

`fmap` aplica una funció als elements d'un contenidor genèric `f a` retornant un contenidor del mateix tipus.

`fmap` és una funció de les instàncies de la classe `Functor`:

```
λ> :type fmap
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

On

```
λ> :info Functor
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

# Maybe és functor

El tipus `Maybe` és instància de `Functor`:

```
λ> :info Maybe
data Maybe a = Nothing | Just a
instance Ord a => Ord (Maybe a)
instance Eq a => Eq (Maybe a)
instance Applicative Maybe
*instance Functor Maybe
instance Monad Maybe
:
```

Concretament,

```
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

# Aplicació

Consulta a una BD:

- Llenguatge sense Maybe:

```
post = Posts.find(1234)
if post is None:
    return None
else:
    return post.title
```

- En Haskell:

```
fmap getPostTitle (findPost 1234)
```

o també:

```
getPostTitle `fmap` findPost 1234
```

o millor (<\$> és l'operador infix per a fmap): (es llegeix *fmap*)

```
getPostTitle <$> findPost 1234
```

# Either a és functor

El tipus `Either a` és instància de `Functor`:

```
instance Functor (Either a) where
  fmap f (Left x) = Left x
  fmap f (Right x) = Right (f x)
```

Fixeu-vos que `Either` té dos paràmetres:

- el tipus `Either a` és la instància de `Functor`,
- el tipus `Either` no.

# Les llistes són functors

El tipus `[]` (llista) és instància de `Functor`:

```
instance Functor [] where
    fmap = map                -- potser és al revés, poc importa
```



# Les funcions són functors

Les funcions també són instàncies de `Functor`:

```
instance Functor ((->) r) where  
    fmap = (.)
```

Exemple:

```
λ> (*3) <$> (+2) <$> Just 1      ➡ Just 9  
λ> (*3) <$> (+2) <$> Nothing    ➡ Nothing
```

# Lleis dels functors

Les instàncies de functors han de tenir aquestes propietats:

1. Identitat: `fmap id ≡ id`.
2. Composició: `fmap (g1 . g2) ≡ fmap g1 . fmap g2`.

**Nota:** Haskell no verifica aquestes propietats (però les pot utilitzar), és responsabilitat del programador fer-ho.

**Exercici:** Comproveu que `Maybe`, `Either a`, `[]`, `(,)` i `(->)` compleixen les lleis dels functors.

# Arbres binaris com a functors

Instanciació pròpia dels functors pels arbres binaris:

```
data Arbin a
  = Buit
  | Node a (Arbin a) (Arbin a)
  deriving (Show)
```

```
instance Functor (Arbin) where

  fmap f Buit = Buit
  fmap f (Node x fe fd) = Node (f x) (fmap f fe) (fmap f fd)
```

```
a = Node 3 Buit (Node 2 (Node 1 Buit Buit) (Node 1 Buit Buit))

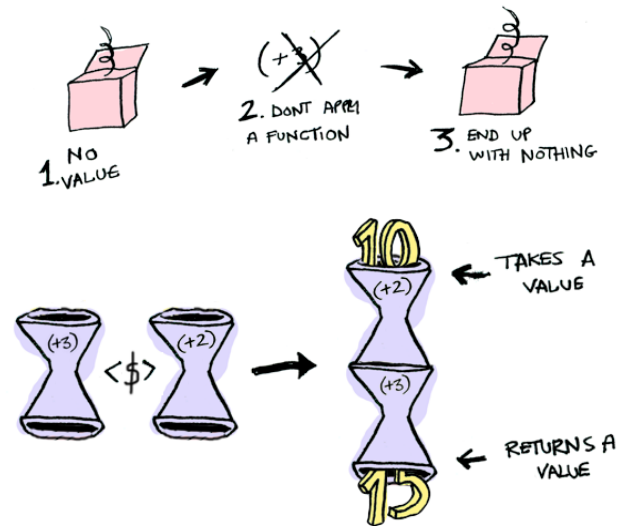
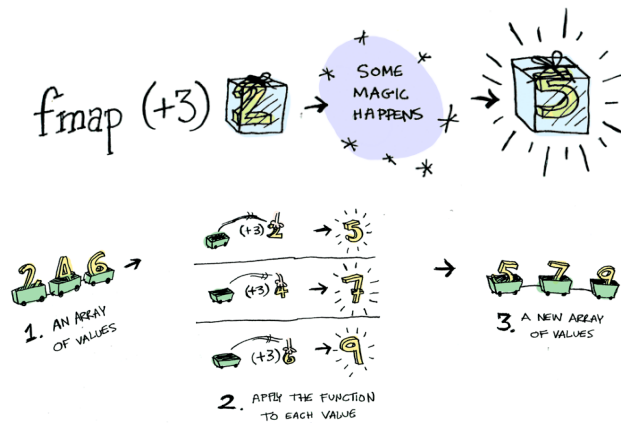
λ> fmap (*2) a
👉 Node 6 Buit (Node 4 (Node 2 Buit Buit) (Node 2 Buit Buit))

λ> fmap even a
👉 Node False Buit (Node True (Node False Buit Buit) (Node False Buit Buit))
```

**Exercici:** Comproveu que `Arbin` compleix les lleis dels functors.

# Sumari

La classe `Functor` captura la idea de tipus contenidor genèric al qual es pot aplicar una funció als seus elements per canviar el seu contingut (però no el contenidor).



Dibuixos: [adit.io](https://adit.io)

# Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis

# Aplicatius

Ja sabem aplicar funcions:

```
λ> (+3) 2
```

👉 5

I ho sabem fer sobre contenidors:

```
λ> fmap (+3) (Just 2)
```

👉 Just 5

Però què passa si la funció és en un contenidor?

```
λ> (Just (+3)) (Just 2)
```

✗

En aquest cas, podem fer servir `<*>`! (es llegeix *app*)

```
λ> Just (+3) <*> Just 2
```

👉 Just 5

```
λ> Just (+3) <*> Nothing
```

👉 Nothing

```
λ> Nothing <*> Just (+3)
```

👉 Nothing

```
λ> Nothing <*> Nothing
```

👉 Nothing

```
λ> Right (+3) <*> Right 2
```

👉 Right 5

```
λ> Right (+3) <*> Left "err"
```

👉 Left "err"

```
λ> Left "err" <*> Right 2
```

👉 Left "err"

```
λ> Left "err1" <*> Left "err2"
```

👉 Left "err1 "

```
λ> [(+2), (+2)] <*> [1, 2, 3]
```

👉 [2, 4, 6, 3, 4, 5]

# Aplicatius

L'operador `<*>` és una operació de la classe `Applicative` (que també ha de ser functor):

```
class Functor f => Applicative f where
    (<*>) :: f (a -> b) -> (f a -> f b)
    pure  :: a -> f a
```

- `<*>` aplica una funció dins d'un contenidor a uns valors dins d'un contenidor. Els contenidors són genèrics i del mateix tipus.
- `pure` construeix un contenidor amb un valor.

# Lleis dels aplicatius

Les instàncies d'aplicatius han de tenir aquestes propietats:

1. Identitat:

$$\text{pure id } <*> v \equiv v.$$

2. Homomorfisme:

$$\text{pure f } <*> \text{pure x} \equiv \text{pure (f x)}.$$

3. Intercanvi:

$$u <*> \text{pure y} \equiv \text{pure (\$ y)} <*> u.$$

4. Composició:

$$u <*> (v <*> w) \equiv \text{pure (.)} <*> u <*> v <*> w.$$

5. Relació amb el functor:

$$\text{fmap g x} \equiv \text{pure g } <*> x.$$



# Instanciacions d'aplicatius

Maybe és aplicatiu:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f <*> x = fmap f x
```

Either a és aplicatiu:

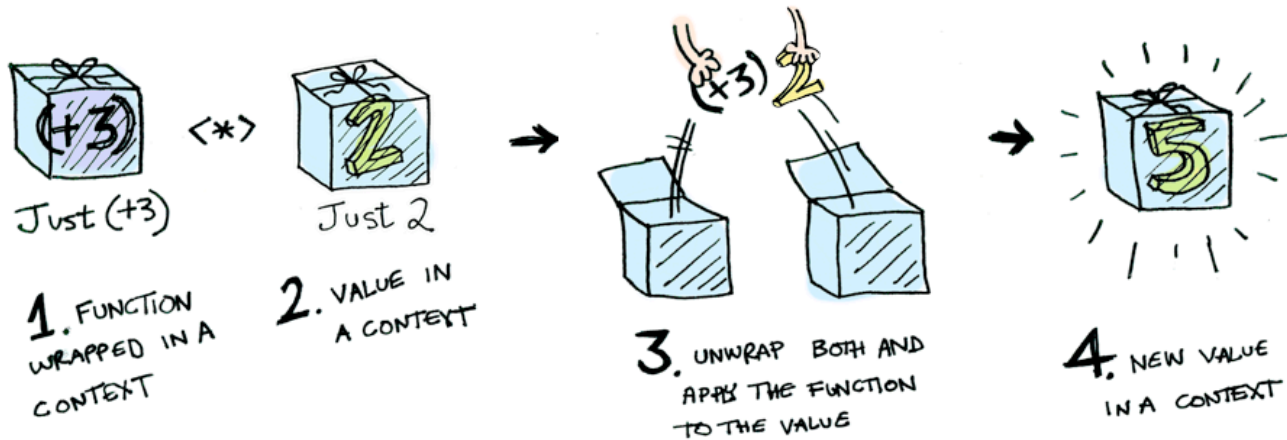
```
instance Applicative (Either a) where
  pure = Right
  Left x <*> _ = Left x
  Right f <*> x = fmap f x
```

**Exercici:** Instancieu les llistes com a aplicatius. Hi ha dues formes de fer-ho.

# Sumari

Els aplicatius permeten aplicar funcions dins d'un contenidor a objectes dins del mateix contenidor.

- `pure` construeix un contenidor amb un valor.
- `<*>` aplica una funció dins d'un contenidor a uns valors dins d'un contenidor:



Dibuixos: [adit.io](https://adit.io)

# Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis

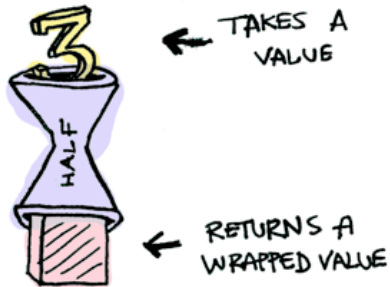
# Mònades

Considerem que `meitat` és una funció que només té sentit sobre parells:

```
meitat :: Int -> Maybe Int

meitat x
| even x    = Just (div x 2)
| otherwise = Nothing
```

Podem veure la funció així: Donat un valor, retorna un valor empaquetat.



Però llavors no li podem ficar valors empaquetats!



Dibuixos: [adit.io](https://adit.io)

# Mònades

Cal una funció que desempaqueti, apliqui `meitat` i deixi empaquetat.

Aquesta funció es diu `>>=` (es llegeix *bind*)

```
λ> Just 40 >>= meitat    👉 Just 20
λ> Just 31 >>= meitat    👉 Nothing
λ> Nothing >>= meitat    👉 Nothing

λ> Just 20 >>= meitat >>= meitat    👉 Just 5
λ> Just 20 >>= meitat >>= meitat >>= meitat    👉 Nothing
```

L'operador `>>=` és una operació de la classe `Monad`:

```
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  -- i més coses
```

El tipus `Maybe` és instància de `Monad`:

```
instance Monad Maybe where
  Nothing >>= f    = Nothing
  Just x  >>= f    = f x
```

# Mònades

De fet, les mònades tenen tres operacions:

```
class Monad m where
  return :: a -> m a
  (>=)    :: m a -> (a -> m b) -> m b
  (>>)    :: m a -> m b -> m b

  r >> k    =    r >= (\_ -> k)
```

- `return` empaqueta.
- `>=` desempaqueta, aplica i empaqueta.
- `>>` és purament estètica.

# Instanciacions

Els tipus `Maybe`, `Either a` i `[]` són instàncies de `Monad`:

```
instance Monad Maybe where
    return      = Just
    Nothing >=> f = Nothing
    Just x  >=> f = f x

instance Monad (Either a) where
    return      = Right
    Left x  >=> f = Left x
    Right x >=> f = f x

instance Monad [] where
    return x      = [x]
    xs >=> f      = concatMap f xs
```

# Lleis de les mònades

Les instàncies de mònades han de tenir aquestes propietats:

1. Identitat per l'esquerra:

```
return x >=> f ≡ f x.
```

2. Identitat per la dreta:

```
m >=> return ≡ m.
```

3. Associativitat:

```
(m >=> f) >=> g ≡ m >=> (\x -> f x >=> g).
```

**Nota:** Haskell no verifica aquestes propietats (però les pot utilitzar), és responsabilitat del programador fer-ho.

**Exercici:** Comproveu que `Maybe`, `Either a i []` compleixen les lleis de les mònades.



# Notació **do**

La **notació do** és sucre sintàctic per facilitar l'ús de les mònades.

⇒ Amb **do**, codi funcional *sembla* codi imperatiu amb assignacions.

Els còmputs es poden **seqüenciar**:

```
do { e1 ; e2 }
```

≡

```
do  
  e1  
  e2
```

≡

```
e1 >> e2
```

≡

```
e1 >>= \_ -> e2
```

I amb **<-** **extreure** el seus resultats:

```
do { x <- e1 ; e2 }
```

≡

```
do  
  x <- e1  
  e2
```

≡

```
e1 >>= \x -> e2
```

# Notació do: Exemple

Tenim llistes associatives amb informació sobre propietaris de cotxes, les seves matrícules, els seus models i les seves etiquetes d'emissions:

```
data Model = Seat127 | TeslaS3 | NissanLeaf | ToyotaHybrid deriving (Eq, Show)
data Etiqueta = Eco | B | C | Cap deriving (Eq, Show)

matricules = [("Joan", 6524), ("Pere", 6332), ("Anna", 5313), ("Laia", 9999)]
models = [(6524, NissanLeaf), (6332, Seat127), (5313, TeslaS3), (7572, ToyotaHybrid)]
etiquetes = [(Seat127, Cap), (TeslaS3, Eco), (NissanLeaf, Eco), (ToyotaHybrid, B)]
```

Donat un nom de propietari, volem saber quina és la seva etiqueta d'emissions:

```
etiqueta :: String -> Maybe Etiqueta
```

És `Maybe` perquè, potser el propietari no existeix, o no tenim la seva matrícula, o no tenim el seu model, o no tenim la seva etiqueta...

# Notació `do`: Exemple

Ens anirà bé usar aquesta funció predefinida de cerca:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Solució amb `case`: 🤖

```
etiqueta nom =  
  case lookup nom matricules of  
    Nothing -> Nothing  
    Just mat -> case lookup mat models of  
      Nothing -> Nothing  
      Just mod -> lookup mod etiquetes
```

Solució amb notació `do`: 💜

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

# Notació **do**: Exemple

Amb notació **do**:

```
etiqueta nom = do  
  mat <- lookup nom matricules  
  mod <- lookup mat models  
  lookup mod etiquetes
```

Transformació de notació **do** a funcional:

```
etiqueta nom =  
  lookup nom matricules >>= \mat -> lookup mat models >>= \mod -> lookup mod eti
```

Amb un format diferent queda clara l'equivalència: 🤪

```
etiqueta nom =  
  lookup nom matricules >>= \mat ->  
  lookup mat models >>= \mod ->  
  lookup mod etiquetes
```

# Funcions predefinides per a mònades

Moltes funcions predefinides tenen una extensió per la classe `Monad`:

- `mapM`, `filterM`, `foldM`, `zipWithM`, ...

També disposem d'operacions per estendre (*lift*) operacions per treballar amb elements de la classe `Monad`. S'han d'importar:

```
import Control.Monad
liftM  :: Monad m => (a -> b) -> m a -> m b
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

Per exemple, podem crear una funció per suma `Maybe`:

```
sumaMaybes :: Num a => Maybe a -> Maybe a -> Maybe a
sumaMaybes = liftM2 (+)
```

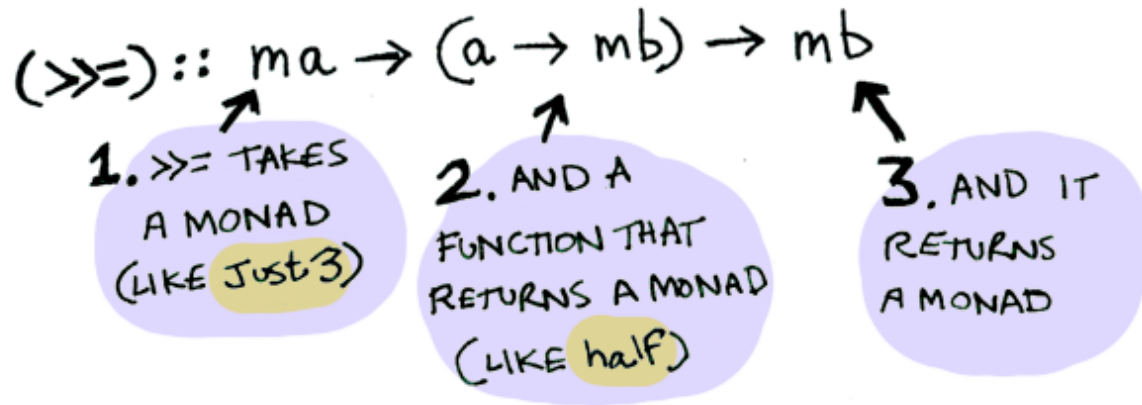
```
λ> sumaMaybes (Just 3) (Just 4) 🙋 Just 7
```

O fer-ho directament:

```
λ> liftM2 (+) (Just 3) (Just 4) 🙋 Just 7
```

# Sumari (1)

- Les mònades permeten aplicar una funció que retorna un valor en un contenidor a un valor en un contenidor.



Dibuixos: [adit.io](https://adit.io)

- Molts tipus predefinitos són instàncies de mònades.
- La notació `do` simplifica l'ús de les mònades.

# Sumari (2)

- Aplicacions:
  - IO
  - Parsers
  - Logging
  - Estat mutable
  - No determinisme
  - Paral·lelisme
- Lectura recomanada: [Monads for functional programming](#) de P. Wadler.

# Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis



# Entrada/Sortida

L'entrada/sortida en Haskell es basa en una mònada:

- El programa principal és `main :: IO ()`
- S'usa el constructor de tipus `IO` per gestionar l'entrada/sortida.
- `IO` és instància de `Monad`.
- Es sol usar amb notació `do`.

Algunes operacions bàsiques:

```
getChar    :: IO Char      -- obté següent caràcter
getLine    :: IO String    -- obté següent línia
getContents :: IO String    -- obté tota l'entrada

putChar     :: Char -> IO () -- escriu un caràcter
putStr      :: String -> IO () -- escriu un text
putStrLn    :: String -> IO () -- escriu un text i un salt de línia
print       :: Show a => a -> IO () -- escriu qualsevol showable

readFile    :: FilePath -> IO String    -- getContents d'un arxiu
writeFile   :: FilePath -> String -> IO () -- operació inversa
```

`()` és una tupla de zero camps i `()` és l'únic valor de tipus `()`.  
( $\Leftrightarrow$  `void` de C).

# Hello world!

```
main = do
  putStrLn "Com et dius?"
  nom <- getLine
  putStrLn $ "Hola " ++ nom ++ "!"
```

Compilació i execució:

```
*> ghc programa.hs
[1 of 1] Compiling Main                ( programa.hs, programa.o )
Linking programa ...

*> ./programa
Com et dius?
*Jordi
Hola Jordi!
```

# Del revés

```
main = do
  x <- getLine
  let y = reverse x
  putStrLn x
  putStrLn y
```

## Compilació i execució:

```
*> ghc programa.hs
[1 of 1] Compiling Main
Linking programa ...

( programa.hs, programa.o )

*> ./programa
*GAT
GAT
TAG
```

# Example

Llegir seqüència de línies acabades en \* i escriure cadascuna del revés:

```
main = do
  line <- getLine
  if line /= "*" then do
    putStrLn $ reverse line
    main
  else
    return ()
```

# Example

Llegir seqüència de línies i escriure cadascuna del revés:

```
main = do
  contents <- getContents
  mapM_ (putStrLn . reverse) (lines contents)
```

L'E/S també és *lazy*, no cal preocupar-se perquè l'entrada sigui massa llarga.

# where en notació do

Degut a la definició del `>>=`, el `where` pot donar problemes:

```
main = do
  x <- getLine
  print f
    where f = factorial (read x)
```

✗ error: **Variable** not **in** scope: `x :: String`

Si ho escrivim amb `>>=`, tenim

```
main = getLine >>= \x -> print f
    where f = factorial (read x)
```

que no pot ser, ja que a les definicions del `where` no podem usar la variable abstracta `x`.

# let en notació do

Amb el `do` cal usar el `let` (sense `in`):


```
main = do
  x <- getLine
  let f = factorial (read x)
  print f
```

Alternativament (més lleig):

```
main = do
  x <- getLine
  f <- return $ factorial (read x)
  print f
```

# Intuïció sobre la mónada IO

Entrada/sortida com funcions que modifiquen el món: món1  $\rightsquigarrow$  món2.

Cadascuna s'encadena amb l'anterior, com un relleu. 

**Exemple:** Llegir i escriure dos caràcters.

```
data World = ... -- descripció del món

myGetChar :: World -> (World, Char)

myPutChar :: Char -> World -> (World, ())

myMain :: World -> (World, ())


myMain w0 = let (w1, c1) = myGetChar w0
                (w2, c2) = myGetChar w1
                (w3, ()) = myPutChar c1 w2
                (w4, ()) = myPutChar c2 w3
            in (w4, ())
```

(1) Passant el relleu.



# Intuïció sobre la mónada IO

Entrada/sortida com funcions que modifiquen el món: món1  $\rightsquigarrow$  món2.

Cadascuna s'encadena amb l'anterior, com un relleu. 

**Exemple:** Llegir i escriure dos caràcters.

```
data World = ... -- descripció del món

myGetChar :: World -> (World, Char)

myPutChar :: Char -> World -> (World, ())

myMain :: World -> (World, ())

myMain w0 = let (w1, c1) = myGetChar w0
                (w2, c2) = myGetChar w1
                (w3, ()) = myPutChar c1 w2
                (w4, ()) = myPutChar c2 w3
            in (w4, ())
```

(1) Passant el relleu.

```
type IO a = World -> (World, a)

getChar :: IO Char

putChar :: Char -> IO ()


main :: IO ()

main =
    getChar >>= \c1 ->
    getChar >>= \c2 ->
    putChar c1 >>
    putChar c2
```

(2) Fent que IO sigui instància de Monad.

# Intuïció sobre la mónada IO

Entrada/sortida com funcions que modifiquen el món: món1  $\rightsquigarrow$  món2.

Cadascuna s'encadena amb l'anterior, com un relleu. 

**Exemple:** Llegir i escriure dos caràcters.

```
type IO a = World -> (World -> a)

getChar :: IO Char

putChar :: Char -> IO ()

main :: IO ()

main =
  getChar >>= \c1 ->
  getChar >>= \c2 ->
  putChar c1 >>
  putChar c2
```

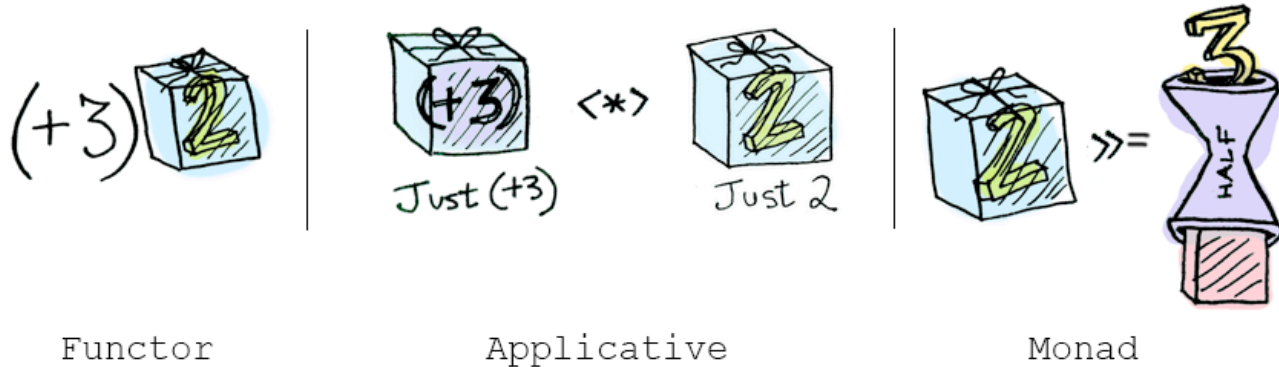
(2) Fent que `IO` sigui instància de `Monad`.

```
main = do
  c1 <- getChar
  c2 <- getChar
  putChar c1
  putChar c2
```

(3) Usant notació `do`.

# Sumari

- Hem vist tres classes predefinides molt importants en Haskell: Functors, Aplicatius, Mònades.



- Molts tipus predefinitos són instàncies d'aquestes classes: `Maybe`, `Either`, llistes, tuples, funcions, `IO`, ...
- La notació `do` simplifica l'ús de les mònades.
- La classe `IO` permet disposar d'entrada/sortida en un llenguatge funcional pur.

# Final

L'**estat d'un programa** descriu tota la informació que no és local a una funció en particular. Això inclou:

- variables globals
- entrada
- sortida

Pensar sobre un programa amb estat és difícil perquè:

- L'estat perviu d'una crida d'una funció a una altra.
- L'estat és a l'abast de totes les funcions.
- L'estat és mutable.
- L'estat canvia en el temps.
- Cap funció és responsable de l'estat.

Estat: 🤖

Sense estat: 💜

Les mònades no eliminen la noció d'estat en un programa, però eliminen la necessitat de mencionar-lo.

# Contingut

- Functors
- Aplicatius
- Mònades
- Entrada/sortida
- Exercicis

# Exercicis

Feu aquests problemes de Jutge.org:

- Functors, aplicatius i mònades:
  - [P70540](#) Expressions
  - [P50086](#) Cua 2
  - [P58738](#) Arbres amb talla
- Entrada/sortida:
  - [P87974](#) Hola / Adéu
  - [P87082](#) Índex massa corporal