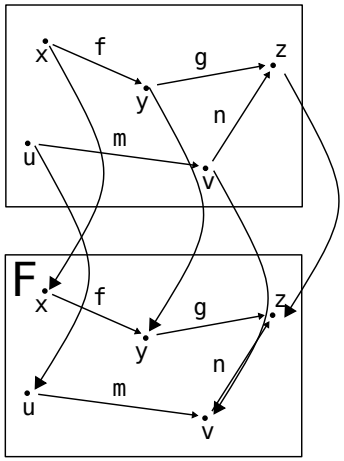
Funktorok (folytatás)

Funktorok: Ismétlés

$$
\begin{array}{l}
\textbf{class}\ \mathit{Functor}\ (\varphi :: \star \to \star)\ \textbf{where} \\
\quad \mathit{fmap} :: (\alpha \to \beta) \to (\varphi\,\alpha \to \varphi\,\beta) \\
(<\$>) = \mathit{fmap}\ \text{-- } \mathit{Control.Applicative}
\end{array}
$$

$$
\begin{array}{l}
\text{-- } \mathit{Control.Applicative} \\
\textbf{class}\ \mathit{Functor}\ \varphi \Rightarrow \mathit{Applicative}\ (\varphi :: \star \to \star)\ \textbf{where} \\
\quad \mathit{pure} \quad :: \alpha \to \varphi\,\alpha \\
\quad (<*>) :: \varphi\,(\alpha \to \beta) \to \varphi\,\alpha \to \varphi\,\beta \\
\textbf{infixl}\ 4\ <\$>,\ <*>
\end{array}
$$



Funktortörvények:

$\mathit{fmap}\ \mathit{id}$	$\equiv \mathit{id}$	(identitás)
$\mathit{fmap}\ (f \circ g)$	$\equiv \mathit{fmap}\ f \circ \mathit{fmap}\ g$	(kompozíció)
$\mathit{pure}\ \mathit{id}\ <*>\ v$	$\equiv v$	(identitás)
$\mathit{pure}\ (\circ)\ <*>\ u\ <*>\ v\ <*>\ w$	$\equiv u\ <*>\ (v\ <*>\ w)$	(kompozíció)
$\mathit{pure}\ f\ <*>\ \mathit{pure}\ x$	$\equiv \mathit{pure}\ (f\ x)$	(homomorfizmus)
$u\ <*>\ \mathit{pure}\ y$	$\equiv \mathit{pure}\ (\$\ y)\ <*>\ u$	(felcserélhetőség)

Egyszerű példák funktorokra

```
instance Functor [] where
  fmap = map

instance Applicative [] where
  pure x      = [x]
  fs <*> xs   = [f x | f <- fs, x <- xs]

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

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

Számítási környezet társítása típusokkal

```
type Name        = String
type Env         = [(Name, Integer)]
newtype Expr α   = E (Env → α)

var :: Name → Expr Integer
var n = E (λ e . case (lookup n e) of
  Just v → v
  _      → error ("Variable is not defined : " ++ show n))

cond :: Expr Bool → Expr α → Expr α → Expr α
cond b x y = E (λ e . if (eval b e) then (eval x e) else (eval y e))

bind :: Name → Expr Integer → Expr α → Expr α
bind n x body = E (λ e . eval body ((n, eval x e) : e))

instance Functor Expr where
  fmap f x = pure f <*> x

instance Applicative Expr where
  pure x     = E (λ _ . x)
  f <*> x    = E (λ e . (eval f e) (eval x e))

eval :: Expr α → (Env → α)
eval (E expr) = expr
```

Számítási környezet társítása típusokkal

$$
\begin{array}{l}
gcd \;::\; Expr\ Integer \\
gcd \;=\; \\
\quad cond\ ((\equiv)\ <\$>\ var\ a\ <*>\ var\ b) \\
\quad\quad (var\ a) \\
\quad\quad (cond\ ((>)\ <\$>\ var\ a\ <*>\ var\ b) \\
\quad\quad\quad (bind\ a\ ((-)\ <\$>\ var\ a\ <*>\ var\ b)\ gcd) \\
\quad\quad\quad (bind\ b\ ((-)\ <\$>\ var\ b\ <*>\ var\ a)\ gcd)) \\
\quad \mathbf{where}\ [a, b]\ =\ ["a", "b"]
\end{array}
$$

Így például:

$$
\begin{array}{lcl}
eval\ gcd\ [("a", 113), ("b", 56)] & \rightarrow_\beta^* & 1 \\
eval\ gcd\ [("a", 56), ("b", 98)] & \rightarrow_\beta^* & 14 \\
eval\ gcd\ [("a", 42), ("b", 42)] & \rightarrow_\beta^* & 42
\end{array}
$$

Szintaktikai elemzés

Valósítsunk meg szintaktikai elemzőket Haskellben, függvényként:

$$\textbf{newtype}\ \mathit{Parser}\ \alpha\ =\ P\ (\mathit{String}\ \rightarrow\ [(\alpha,\ \mathit{String})])$$

```
-- char :: Char → String → [(Char, String)]
char :: Char → Parser Char
char c = P (λ s . case s of
  (x : xs) | (x ≡ c) → [(x, xs)]
  _                  → [])
```

$$\mathit{runParser}\ ::\ \mathit{Parser}\ \alpha\ \rightarrow\ \mathit{String}\ \rightarrow\ [(\alpha,\ \mathit{String})]$$
$$\mathit{runParser}\ (P\ p)\ s\ =\ p\ s$$

$$\mathit{parseAs}\ ::\ \mathit{Parser}\ \alpha\ \rightarrow\ \mathit{String}\ \rightarrow\ \mathit{Maybe}\ \alpha$$
$$\mathit{parseAs}\ p\ s\ =\ \mathit{fst}\ <\$>\ \mathit{find}\ (\lambda\ (x, s)\ .\ \mathit{null}\ s)\ (\mathit{runParser}\ p\ s)$$

Kompozicionális szintaktikai elemzés

```
instance Functor Parser where
  fmap f x = pure f <*> x

instance Applicative Parser where
  (P pf) <*> (P q) =
    P (λ s . [ (f x, s2) | (f, s1) ← pf s, (x, s2) ← q s1 ])
  pure x = P (λ s . [ (x, s) ])

token :: String → Parser String
token = foldr (λ x xs . (:) <$> char x <*> xs) (pure "")
```

Alternatív funktorok

Megadható még egy, a szintaktikai elemzéshez még jobban illeszkedő funktorfajta:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Applicative}\ \varphi \Rightarrow \mathit{Alternative}\ (\varphi :: \star \to \star)\ \textbf{where} \\
\quad \mathit{empty} \quad :: \varphi\ \alpha \\
\quad (<|>) \quad :: \varphi\ \alpha \to \varphi\ \alpha \to \varphi\ \alpha \\
\\
\quad \mathit{some} \quad :: \varphi\ \alpha \to \varphi\ [\alpha] \\
\quad \mathit{some}\ p = (:)\ <\$>\ p\ <*>\ \mathit{many}\ p \\
\\
\quad \mathit{many} \quad :: \varphi\ \alpha \to \varphi\ [\alpha] \\
\quad \mathit{many}\ p = \mathit{some}\ p\ <|>\ \mathit{pure}\ [] \\
\textbf{infixl}\ 3\ <|>
\end{array}
$$

$$
\begin{array}{l}
\mathit{optional} \quad :: \mathit{Alternative}\ \varphi \Rightarrow \varphi\ \alpha \to \varphi\ (\mathit{Maybe}\ \alpha) \\
\mathit{optional}\ v = \mathit{Just}\ <\$>\ v\ <|>\ \mathit{pure}\ \mathit{Nothing}
\end{array}
$$

Az elemző mint alternatív funktor

```
instance Functor Parser where
  fmap f x = pure f <*> x

instance Applicative Parser where
  (P pf) <*> (P q) =
    P (λ s . [ (f x, s2) | (f, s1) ← pf s, (x, s2) ← q s1 ])
  pure x = P (λ s . [ (x, s) ])

instance Alternative Parser where
  (P p) <|> (P q) = P (λ s . (p s) ++ (q s))
  empty = P (λ s . [])
```

Szintaktikai elemzés funktorokkal: Összefoglaló példa

```
matches :: (Char → Bool) → Parser Char
matches p = P (λ s . case s of
  (x : xs) | p x  → [(x, xs)]
  _                → [])
byRadix :: [Char] → Parser Integer
byRadix symbols = foldl' (λ n d . n * radix + d) 0 <$> some digit
  where
    radix = genericLength symbols
    digit = (toInteger ∘ digitToInt ∘ toUpper) <$> matches (`elem` symbols)
decimal, octal, hexadecimal :: Parser Integer
decimal     = byRadix ['0'..'9']
octal       = byRadix ['0'..'7']
hexadecimal = byRadix (['0'..'9'] ++ ['A'..'F'])
integer :: Parser Integer
integer =
  decimal <|>
  ((token "0o" <|> token "0O") *> octal) <|>
  ((token "0x" <|> token "0X") *> hexadecimal)
```

Bővebben:

http://hackage.haskell.org/package/parsec

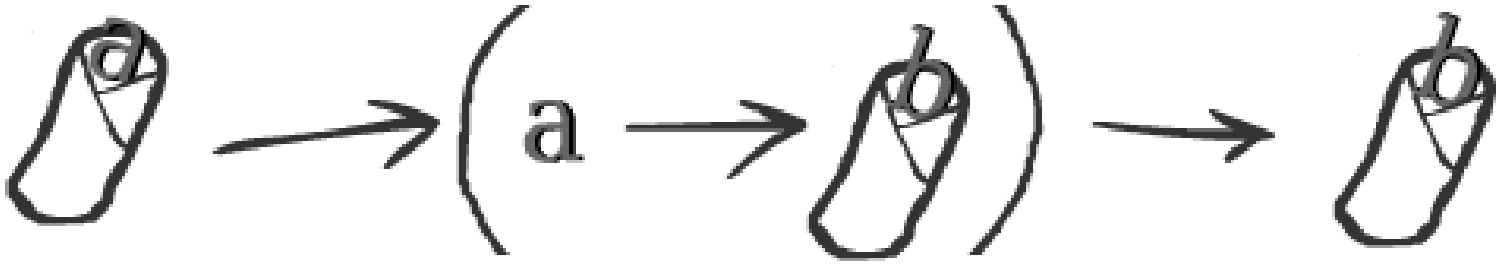
Monádok

Programozás monádokkal: Bevezetés

https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/

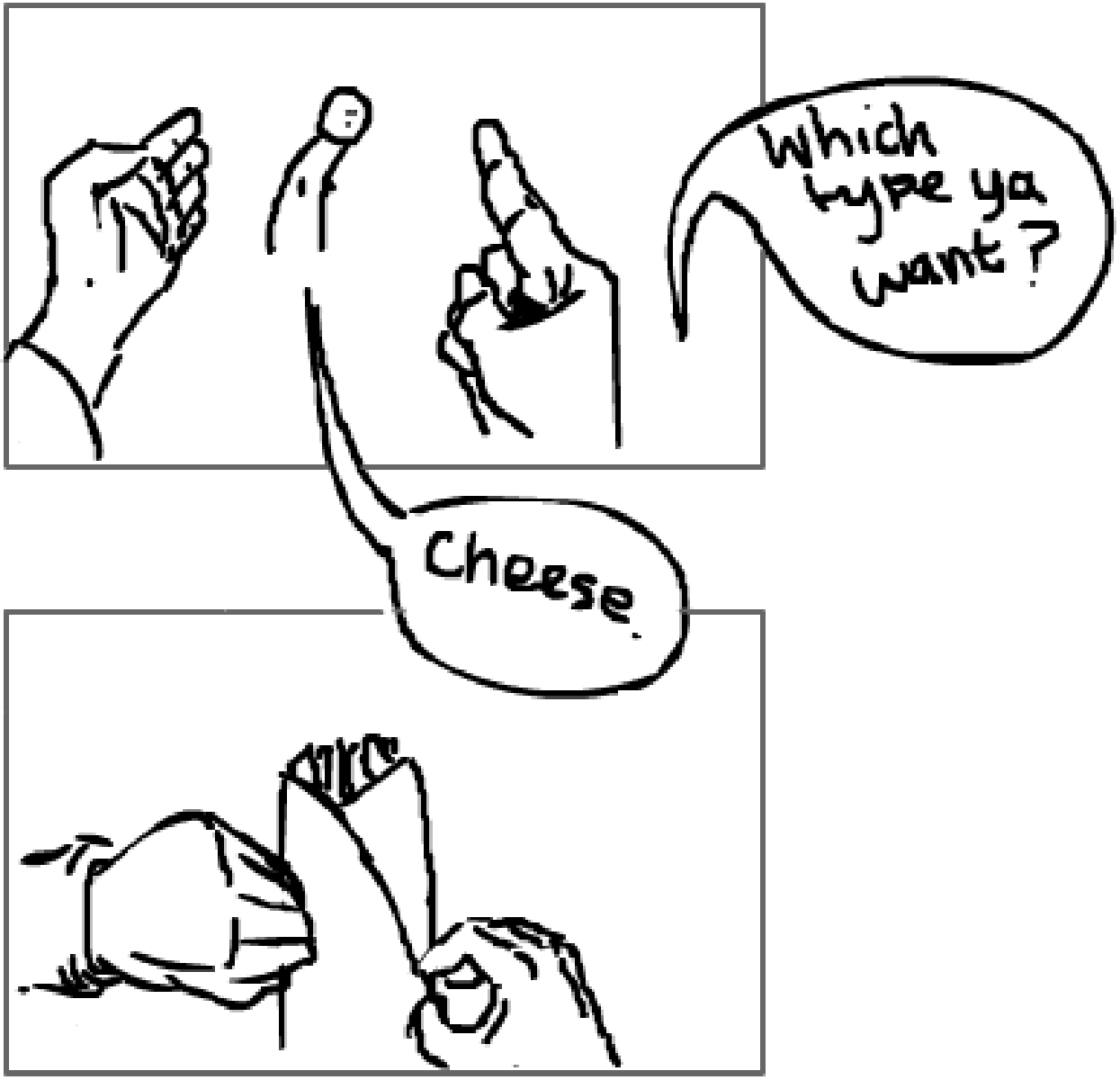
Programozás burritokkal

monads are burritos?



Programozás burritokkal

Programozás burritokkal

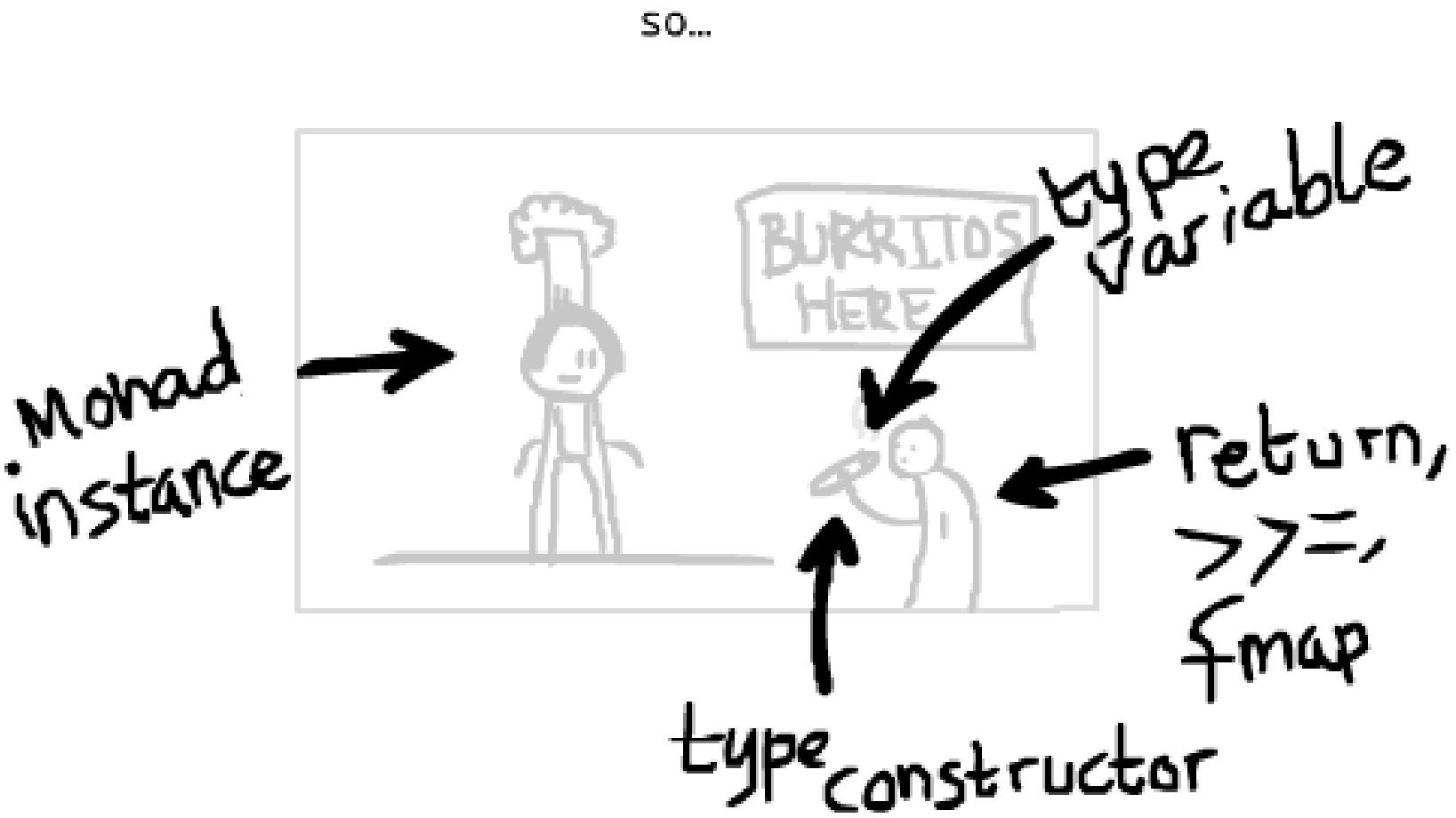


Programozás burritokkal

Programozás burritokkal

Programozás burritokkal

Programozás burritokkal



Mi nem a monád?

A következő állítások mindegyike *hamis*:

- A monádok nem tisztán funkcionálisak.
- A monádok programbeli hatásokat modelleznek.
- A monádok az állapotról szólnak.
- A monádok utasítások sorbarendezését teszik lehetővé.
- A monádok az I/O-t modellezik.
- A monádok működéséhez szükséges a lusta kiértékelés.
- A monádok segítségével lehet trükkösen mellékhatásokat használni.
- A monádok egy beágyazott nyelv a Haskellen belül.
- A monádokat csak matematikusok érthetik meg.

A monád fogalmának megértése 8 egyszerű lépésben

1. Ne olvassuk tutorialokat!
2. Ne olvassuk tutorialokat!
3. Tanuljunk a típusokról (→ „Nyelvek típusrendszere” tárgy)!
4. Tanuljuk meg a típusosztályokat (→ „Funkcionális nyelvek” tárgy)!
5. Tanulmányozzuk a Typeclassopediát*!
6. Tanulmányozzuk a monádok definícióját!
7. Programozzunk monádokkal (→ beadandók)!
8. Ne írjunk tutorialokat!

* `http://www.cs.tufts.edu/comp/150FP/archive/brent-yorgey/tc.pdf`

Programozás monádokkal: Programstrukturálás

Tegyük fel, hogy meg szeretnénk írni egy tisztán funkcionális nyelven az alábbi algoritmussal rendelkező programot:

- Írjuk ki a képernyőre, hogy "*Provide me a word* $>$ ".
- Olvassuk be a felhasználó által begépelt szót.
- Az adott szótól függően tegyük a következőt:
  - Ha palindróma, akkor írjuk ki a képernyőre, hogy "*Palindrome*."
  - Ha nem palindróma, akkor írjuk ki a képernyőre, hogy "*Not a palindrome*."

Adottak:

$$\begin{aligned} &putStr &&:: String \rightarrow World \rightarrow World \\ &getLine &&:: World \rightarrow (String,\ World) \end{aligned}$$

Programozás monádokkal: Programstrukturálás

$$
\begin{aligned}
&(\$) :: (\alpha \to \beta) \to \alpha \to \beta \\
&f \,\$\, x = f\, x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{program} :: \mathit{World} \to \mathit{World} \\
&\mathit{program}\, w = \\
&\quad (\lambda\, (s,\ w)\, .\, \textbf{if}\, (s \equiv \mathit{reverse}\, s) \\
&\qquad \textbf{then}\ \mathit{putStr}\ \text{"}\mathit{A\ palindrome.}\text{"}\ w \\
&\qquad \textbf{else}\ \mathit{putStr}\ \text{"}\mathit{Not\ a\ palindrome.}\text{"}\ w)\ \$ \\
&\quad \mathit{getLine}\ \$ \\
&\quad \mathit{putStr}\ \text{"}\mathit{Provide\ me\ a\ word} > \text{"}\ \$\ w
\end{aligned}
$$

Programozás monádokkal: Programstrukturálás

$$(\triangleright) :: \alpha \to (\alpha \to \beta) \to \beta$$
$$x \triangleright f = f\ x$$

$$\begin{array}{l}
\mathit{program}' :: \mathit{World} \to \mathit{World} \\
\mathit{program}'\ w = w \triangleright \\
\quad \mathit{putStr}\ \text{"}\mathit{Provide\ me\ a\ word} > \text{"} \triangleright \\
\quad \mathit{getLine} \triangleright \lambda\,(s,\ w)\,. \\
\quad \textbf{if}\ (s \equiv \mathit{reverse}\ s) \\
\qquad \textbf{then}\ \mathit{putStr}\ \text{"}\mathit{A\ palindrome.}\text{"}\ w \\
\qquad \textbf{else}\ \mathit{putStr}\ \text{"}\mathit{Not\ a\ palindrome.}\text{"}\ w
\end{array}$$

Programozás monádokkal: Programstrukturálás

$$\textbf{type}\ IO\,\alpha\ =\ World\ \rightarrow\ (\alpha,\ World)$$

-- putStr :: String → IO ()

-- getLine :: IO String

$$(\gg\!=)\ ::\ IO\,\alpha\ \rightarrow\ (\alpha\ \rightarrow\ IO\,\beta)\ \rightarrow\ IO\,\beta$$

$$(x\ \gg\!=\ f)\ w\ =\ (f\ x')\ w_{mod}\ \textbf{where}\ (x',\ w_{mod})\ =\ x\ w$$

$$(\gg)\ ::\ IO\,\alpha\ \rightarrow\ IO\,\beta\ \rightarrow\ IO\,\beta$$

$$x\ \gg\ f\ =\ x\ \gg\!=\ \lambda\,\_\,.\,f$$

$$program''\ ::\ IO\,()$$

$$program''\ =$$

$$\quad putStr\ ''Provide\ me\ a\ word>\ ''\ \gg$$

$$\quad getLine\ \gg\!=\ \lambda\,s\,.$$

$$\quad \textbf{if}\ (s\ \equiv\ reverse\ s)$$

$$\qquad \textbf{then}\ putStr\ ''A\ palindrome.''$$

$$\qquad \textbf{else}\ putStr\ ''Not\ a\ palindrome.''$$

Programozás monádokkal: Programstrukturálás

```
main :: IO ()
main = do
  putStr "Provide me a word > "
  s ← getLine
  if (s ≡ reverse s)
    then putStr "A palindrome."
    else putStr "Not a palindrome."
```

Programozás monádokkal: A *Monad* típusosztály

```
-- Control.Monad
class Applicative μ ⇒ Monad (μ :: ⋆ → ⋆) where
  return :: α → μ α
  return = pure

  (>>=) :: μ α → (α → μ β) → μ β

  (>>) :: μ α → μ β → μ β
  x >> f = x >>= λ_. f

  fail :: String → μ α
  fail = error

(>=>) :: Monad μ ⇒ (α → μ β) → (β → μ γ) → (α → μ γ)
f >=> g = λ x . f x >>= g
```

Monádtörvények:

```
return >=> f        ≡ f
f >=> return        ≡ f
(f >=> g) >=> h ≡ f >=> (g >=> h)
```

+ „run” függvény