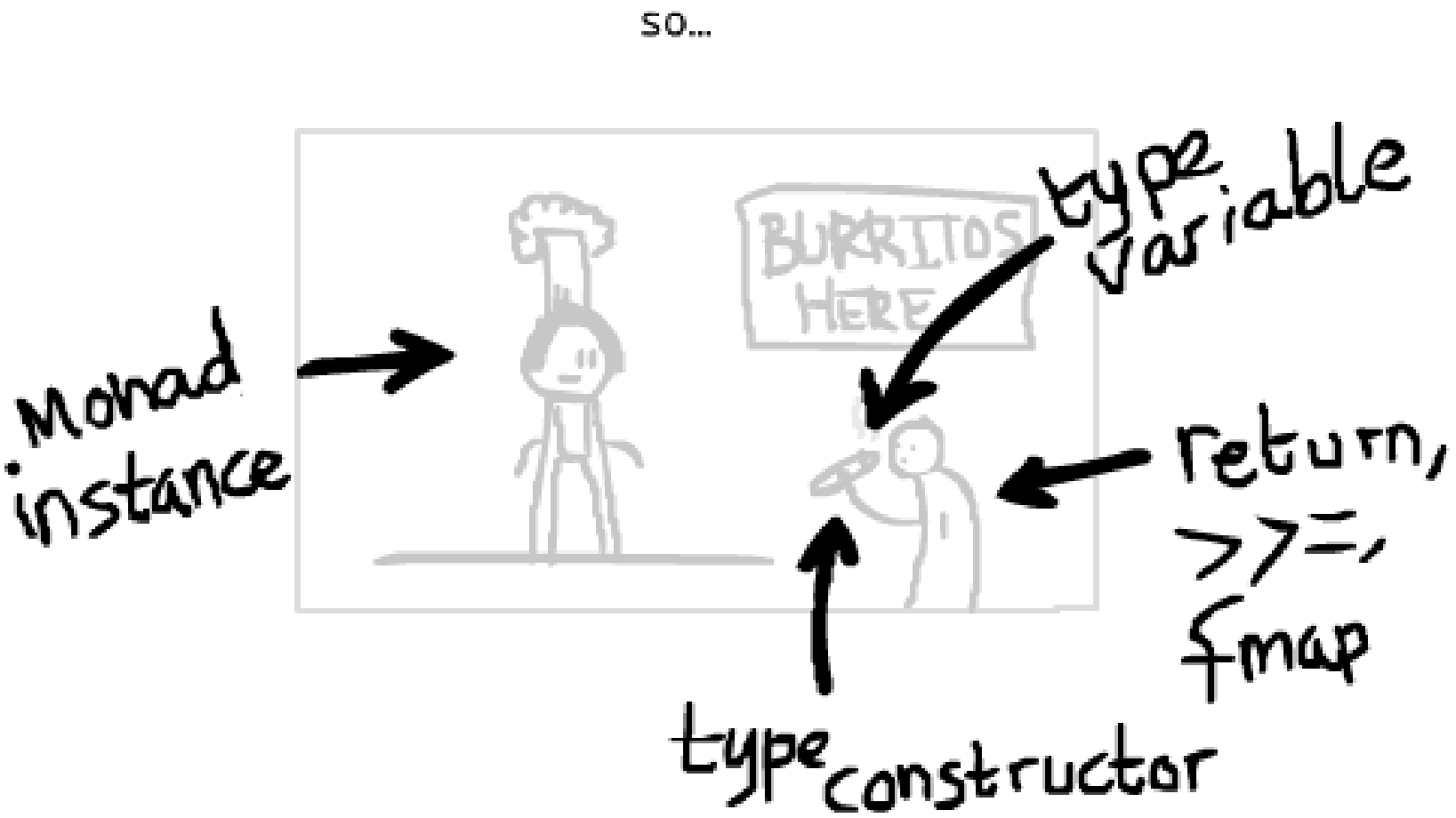
Monádok (folytatás)

Programozás burritokkal



Programozás monádokkal: Programstrukturálás

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

-- putStr :: String → IO ()

-- getLine :: IO String

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

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

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

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

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

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

Programozás monádokkal: A *do* szintaktika

Átírási szabályok:

$$\begin{array}{ll}
\textbf{do}\ \{\, e_1;\ e_2 \,\} & \longrightarrow e_1 \gg \textbf{do}\ \{\, e_2 \,\} \\
\textbf{do}\ \{\, p \leftarrow e_1;\ e_2 \,\} & \longrightarrow e_1 \gg\!= \lambda\, x\,.\ \textbf{case}\ x\ \textbf{of}\ p \rightarrow \textbf{do}\ \{\, e_2 \,\} \\
 & \qquad\qquad\qquad\qquad\qquad\quad \_ \rightarrow \mathit{fail}\ ''\mathit{error}'' \\
\textbf{do}\ \{\, \textbf{let}\ p = e1;\ e2 \,\} & \longrightarrow \textbf{let}\ p = e_1\ \textbf{in}\ \textbf{do}\ \{\, e_2 \,\} \\
\textbf{do}\ \{\, e \,\} & \longrightarrow e
\end{array}$$

Például:

$$\begin{array}{lclcl}
\textbf{do}\ \{ & & \textbf{do} & & \\
\quad x \leftarrow f; & & \quad x \leftarrow f & & f \gg\!= \lambda\, x\,. \\
\quad y \leftarrow g; & & \quad y \leftarrow g & & g \gg\!= \lambda\, y\,. \\
\quad z \leftarrow h; & \longrightarrow & \quad z \leftarrow h & \longrightarrow & h \gg\!= \lambda\, z\,. \\
\quad \textbf{let}\ t = (x, y, z); & & \quad \textbf{let}\ t = (x, y, z) & & \textbf{let}\ t = (x, y, z)\ \textbf{in} \\
\quad \mathit{return}\ t & & \quad \mathit{return}\ t & & \mathit{return}\ t \\
\} & & & &
\end{array}$$

Monádtörvények a gyakorlatban

$$\begin{array}{l} \mathit{return} \gg\!= f \equiv f : \\ \mathbf{do}\ x \leftarrow m \equiv \mathbf{do}\ m \\ \quad \mathit{return}\ x \end{array}$$

$$\begin{array}{l} f \gg\!= \mathit{return} \equiv f : \\ \mathbf{do}\ y \leftarrow \mathit{return}\ x \equiv \mathbf{do}\ f\ x \\ \quad f\ y \end{array}$$

$$\begin{array}{l} (f \gg\!= g) \gg\!= h \equiv f \gg\!= (g \gg\!= h) : \\ \mathbf{do}\ y \leftarrow \mathbf{do}\ x \leftarrow m \equiv \mathbf{do}\ x \leftarrow m \equiv \mathbf{do}\ x \leftarrow m \\ \qquad\qquad\quad f\ x \qquad\qquad\quad y \leftarrow f\ x \qquad \mathbf{do}\ y \leftarrow f\ x \\ \quad g\ y \qquad\qquad\qquad\quad g\ y \qquad\qquad\quad g\ y \end{array}$$

Milyen monádok léteznek?

$$\textbf{data}\ \mathit{IO}\ \alpha\ = ?$$

Az *IO* monáddal tetszőleges mellékhatást tudunk modellezni, ez a „jolly joker”.

Például:

- *putStrLn*: írás a szabványos kimenetre
- *getLine*: olvasás a szabványos bemenetről
- *IORef*: módosítható hivatkozások, „mutatók”
- *IOError*: kivétel(kezelés)
- *IOArray*: hatékony, felülírható elemeket tartalmazó tömb
- *forkIO*: (konkurens) szálak létrehozása

„run” függvény: a futtató rendszer, *unsafePerformIO*

Ilyen monádok léteznek?

$$\textbf{data}\ [\alpha]\ =\ []\ |\ \alpha\ :\ [\alpha]$$

```
instance Monad [] where
  return x    = [x] -- α → [α]
  xs >>= f    = concat [ f x | x ← xs ] -- [α] → (α → [β]) → [β]
  fail _      = [] -- String → [α]
```

```
multiplyToNM :: (Num α, Eq α, Enum α) ⇒ α → [(α, α)]
multiplyToNM n = do
  x ← [1..n]
  y ← [x..n]
  if (x * y ≡ n)
    then return (x, y)
    else fail "Not applicable."
```

$$multiplyToN\ n\ =\ [\ (x, y)\ |\ x\ \leftarrow\ [1..n],\ y\ \leftarrow\ [x..n],\ x\ *\ y\ \equiv\ n\ ]$$

Az *Identity* monád (intuíció)

$$\textbf{type}\ \mathit{Identity}\ \alpha\ =\ \alpha$$

$$\begin{aligned}
&\text{-- } \alpha \to (\alpha \to \beta) \to \beta \\
&(\gg\!=)\ ::\ \mathit{Identity}\ \alpha\ \to\ (\alpha\ \to\ \mathit{Identity}\ \beta)\ \to\ \mathit{Identity}\ \beta \\
&x\ \gg\!=\ f\ =\ f\ x
\end{aligned}$$

$$\begin{aligned}
&\text{-- } \alpha \to \alpha \\
&\mathit{return}\ ::\ \alpha\ \to\ \mathit{Identity}\ \alpha \\
&\mathit{return}\ x\ =\ x
\end{aligned}$$

$$\begin{aligned}
&\text{-- } \alpha \to \alpha \\
&\mathit{runIdentity}\ ::\ \mathit{Identity}\ \alpha\ \to\ \alpha \\
&\mathit{runIdentity}\ x\ =\ x
\end{aligned}$$

Az *Identity* monád (definíció)

```
-- Control.Monad.Identity
data Identity α = I α

runIdentity :: Identity α → α
runIdentity (I x) = x

instance Monad Identity where
  return x   = I x
  x >>= f    = f (runIdentity x)
```

Az *Identity* monád (példa)

$$
\begin{aligned}
&m :: Int \rightarrow Identity\ Int \\
&m\ x = return\ (x * x)
\end{aligned}
$$

$$
\begin{aligned}
&m_1 :: Int \rightarrow Identity\ Int \\
&m_1\ x = \mathbf{do} \\
&\quad return\ (x * x) \\
&\quad return\ (x * 2) \\
&\quad return\ (x * 3)
\end{aligned}
$$

$$
\begin{aligned}
&m_2 :: Int \rightarrow Identity\ Int \\
&m_2\ n = \mathbf{do} \\
&\quad x \leftarrow m_1\ n \\
&\quad y \leftarrow return\ (x * n) \\
&\quad m\ y
\end{aligned}
$$

$$y = runIdentity\ (m_2\ 4)$$

A *Maybe* monád

```
data Maybe α = Just α | Nothing

instance Monad Maybe where
  -- (>>=) :: Maybe α → (α → Maybe β) → Maybe β
  (Just x) >>= f = f x
  Nothing >>= f = Nothing

  -- return :: α → Maybe α
  return = Just

testMaybe :: Maybe Int
testMaybe =
  Just 3 >>= λ x .
  Just 4 >>= λ y .
  return (x + y)
```

A *Reader* monád (intuíció)

$$\textbf{type}\ \mathit{Reader}\ \varepsilon\ \alpha\ =\ \varepsilon\ \to\ \alpha$$

-- $(\varepsilon \to \alpha) \to (\alpha \to (\varepsilon \to \beta)) \to (\varepsilon \to \beta)$
$(\gg\!=)\ ::\ \mathit{Reader}\ \varepsilon\ \alpha\ \to\ (\alpha\ \to\ \mathit{Reader}\ \varepsilon\ \beta)\ \to\ \mathit{Reader}\ \varepsilon\ \beta$
$(x\ \gg\!=\ f)\ e\ =\ f\ (x\ e)\ e$

-- $\alpha \to (\varepsilon \to \alpha)$
$\mathit{return}\ ::\ \alpha\ \to\ \mathit{Reader}\ \varepsilon\ \alpha$
$(\mathit{return}\ x)\ e\ =\ x$

-- $\varepsilon \to \varepsilon$
$\mathit{ask}\ ::\ \mathit{Reader}\ \varepsilon\ \varepsilon$
$(\mathit{ask})\ e\ =\ e$

-- $(\varepsilon \to \alpha) \to \varepsilon \to \alpha$
$\mathit{runReader}\ ::\ \mathit{Reader}\ \varepsilon\ \alpha\ \to\ \varepsilon\ \to\ \alpha$
$\mathit{runReader}\ f\ =\ f$

A *Reader* monád (definíció)

```
-- Control.Monad.Reader
newtype Reader ε α = R (ε → α)

runReader :: Reader ε α → ε → α
runReader (R f) = f

instance Monad (Reader ε) where
  return x   = R $ λ e . x
  x >>= f = R $ λ e.
    let x1 = runReader x e in
    runReader (f x1) e

ask :: Reader ε ε
ask = R (λ e . e)
```

A *Reader* monád (példa)

```
type Identifier    = String
type Bindings α    = Data.Map.Map Identifier α

valueOf :: Identifier → Bindings α → α
valueOf name binds =
  maybe (error "Not found") id (Data.Map.lookup name binds)

bindings :: [(Identifier, α)] → Bindings α
bindings = Data.Map.fromList

-- getValuesM :: [Identifier] → Bindings α → [α]
getValuesM :: [Identifier] → Reader (Bindings α) [α]
getValuesM ids = do
  binds ← ask
  return [ valueOf id binds | id ← ids ]

getValues :: [Identifier] → Bindings α → [α]
getValues ids names = runReader (getValuesM ids) names
```

A *State* monád (intuíció)

```
type State σ α = σ → (α, σ)

-- (σ → (α, σ)) → (α → (σ → (β, σ))) → (σ → (β, σ))
(>>=) :: State σ α → (α → State σ β) → State σ β
(x >>= f) s = (f v) s_mod where (v, s_mod) = x s

-- α → (σ → (α, σ))
return :: α → State σ α
(return x) s = (x, s)

-- σ → (σ, σ)
get :: State σ σ
(get) s = (s, s)

-- σ → ((), σ)
put :: σ → State σ ()
(put x) s = ((), x)

-- (σ → (α, σ)) → σ → (α, σ))
runState :: State σ α → σ → (α, σ))
runState f = f
```

A *State* monád (definíció)

```
-- Control.Monad.State
newtype State σ α = S (σ → (α, σ))

runState :: State σ α → σ → (α, σ)
runState (S f) = f

instance Monad (State σ) where
  return x = S $ λ s . (x, s))
  x >>= f = S $ λ s .
    let (y, s_mod) = runState x s in
    runState (f y) s_mod

get :: State σ σ
get = S $ λ s . (s, s)

put :: σ → State σ ()
put x = S $ λ _ . ((), x)
```

A *State* monád (példa)

```
data Tree α = Node a (Tree a) (Tree a) | Leaf
  deriving Show

numberTree :: (Eq α) ⇒ Tree α → Tree Int
numberTree t = fst (runState (numberTreeM t) [])

numberTreeM :: (Eq α) ⇒ Tree α → State [α] (Tree Int)
numberTreeM Leaf            = return Leaf
numberTreeM (Node x lt rt)  = do
  table ← get
  let (table_mod, pos) = case (Data.List.findIndex (≡ x) table) of
                           Just i  → (table, i)
                           _       → (table ++ [x], length table)
  put table_mod
  lt_mod ← numberTreeM lt
  rt_mod ← numberTreeM rt
  return (Node pos lt_mod rt_mod)
```

A *Writer* monád (egyszerűsített)

```
-- Control.Monad.Writer
newtype Writer ω α = W (α, ω)

runWriter :: (Monoid ω) ⇒ Writer ω α → (α, ω)
runWriter (W (x, w)) = (x, w)

instance (Monoid ω) ⇒ Monad (Writer ω) where
  return x = W (x, mempty)
  x >>= f = W $
    let (x1, w1) = runWriter x in
    let (x2, w2) = runWriter (f x1) in
    (x2, w1 <> w2)

tell :: (Monoid ω) ⇒ ω → Writer ω ()
tell x = W ((), x)

censor :: (Monoid ω) ⇒ (ω → ω) → Writer ω α → Writer ω α
censor f (W (x, w)) = W (x, f w)
```

A *Writer* monád (*Monoid*)

```
-- Data.Monoid
class Monoid α where
  mempty :: α
  mappend :: α → α → α
(<>) :: (Monoid α) ⇒ α → α → α
(<>) = mappend
```

Monoidtörvények:

$$
\begin{aligned}
&\mathit{mempty} \ <> \ x && \equiv x \\
&x \ <> \ \mathit{mempty} && \equiv x \\
&x \ <> \ (y \ <> \ z) && \equiv (x \ <> \ y) \ <> \ z \\
&\mathit{mconcat} && \equiv \mathit{foldr}\ (<>)\ \mathit{mempty}
\end{aligned}
$$

Például:

```
instance Monoid [α] where
  mempty  = []
  mappend = (++)
instance (Monoid α, Monoid β) ⇒ Monoid (α, β) where
  mempty                  = (mempty, mempty)
  mappend (x1, y1) (x2, y2) = (x1 <> x2, y1 <> y2)
```

A *Writer* monád (példa)

```
gcdWithLog :: Int → Int → (Int, [String])
gcdWithLog x y = runWriter (censor reverse (gcdWithLogM x y))

gcdWithLogM :: Int → Int → Writer [String] Int
gcdWithLogM x y | y ≡ 0 = do
  tell ["Finished with" ++ show x]
  return x
gcdWithLogM x y = do
  result ← gcdWithLogM y (x `mod` y)
  let msg = show x ++ " mod " ++ show y ++ " = " ++ show (x `mod` y)
  tell [msg]
  return result
```