Μονάδες

Γιάννης Κασσιός

Δομή της Σημερινής Διάλεξης

- Εισαγωγή (μεγάλη!)
- Κατασκευαστές Τύπων και Υπερφόρτωση
- Κλάση `Monad`
- Σύνταξη `do`
- Είσοδος/έξοδος

Δομή της Σημερινής Διάλεξης

- Εισαγωγή (μεγάλη!)
- Κατασκευαστές Τύπων και Υπερφόρτωση
- Κλάση `Monad`
- Σύνταξη `do`
- Είσοδος/έξοδος

Δομή της Σημερινής Διάλεξης

- Εισαγωγή (μεγάλη!)
- Κατασκευαστές Τύπων και Υπερφόρτωση
- Κλάση `Monad`
- Σύνταξη `do`
- Είσοδος/έξοδος

Δομή της Σημερινής Διάλεξης

- Εισαγωγή (μεγάλη!)
- Κατασκευαστές Τύπων και Υπερφόρτωση
- Κλάση `Monad`
- Σύνταξη `do`
- Είσοδος/έξοδος

Δομή της Σημερινής Διάλεξης

- Εισαγωγή (μεγάλη!)
- Κατασκευαστές Τύπων και Υπερφόρτωση
- Κλάση `Monad`
- Σύνταξη `do`
- Είσοδος/έξοδος

- Σύνθεση συναρτήσεων $f_i :: T_i \texttt{->} T_{i+1}$ και $x :: T_0$:

$$f_{n-1}\,(\,f_{n-2}\,(\,\ldots f_0\;x\,)\,)\ldots :: T_n$$

- Ορίζουμε:

```
app x f = f x ; infixl 0 `app`
```

Τύπος: `a->(a->b)->b`

- Το παραπάνω γράφεται:

$$x \texttt{ `app` } f_0 \texttt{ `app` } \ldots \texttt{ `app` } f_{n-1} \texttt{ :: } T_n$$

- Σύνθεση συναρτήσεων $f_i :: T_i \texttt{->} T_{i+1}$ και $x :: T_0$:

$$f_{n-1}\,(\,f_{n-2}\,(\ldots f_0\ x\,)\,)\ldots :: T_n$$

- Ορίζουμε:
  `app x f = f x ; infixl 0 ‘app‘`
  Τύπος: `a->(a->b)->b`
- Το παραπάνω γράφεται:

$$x\ \texttt{‘app‘}\ f_0\ \texttt{‘app‘}\ \ldots\ \texttt{‘app‘}\ f_{n-1}\ \texttt{::}\ T_n$$

- Σύνθεση συναρτήσεων f_i `::` T_i `->` T_{i+1} και x `::` T_0:

$$f_{n-1}\,(\,f_{n-2}\,(\,\ldots f_0\;x\,)\,)\ldots \texttt{::}\; T_n$$

- Ορίζουμε:
  ```
  app x f = f x ; infixl 0 `app`
  ```
  Τύπος: `a->(a->b)->b`
- Το παραπάνω γράφεται:

$$x\ \texttt{`app`}\ f_0\ \texttt{`app`}\ \ldots\ \texttt{`app`}\ f_{n-1}\ \texttt{::}\ T_n$$

- Θέλουμε να ἐμπλουτίσουμε΄ τις συναρτήσεις με επιπλέον λειτουργικότητα: f'_i `::` T_i `->m` T_{i+1}
- Θέλουμε ο υπολογισμός μας να γίνεται από `m` T_0 σε `m` T_n:

$$x' \text{ `app` } f'_0 \text{ `app` } \ldots \text{ `app` } f'_{n-1} \texttt{ :: m } T_n$$

  όπου x' `::m` T_0
- Χρειαζόμαστε:
  - καινούρια συνάρτηση σύνθεσης `appm`
  - συνάρτηση `liftm` που να "ανεβάζει" μια τιμή `a` σε `m a`
- Τύποι:

```
appm :: m a->(a->m b)->m b
liftm :: a->m a
```

- Η σύνθεση γίνεται:

$$\texttt{liftm } x \text{ `appm` } f'_0 \text{ `appm` } \ldots \text{ `appm` } f'_{n-1} \texttt{ :: m } T_n$$

- Θέλουμε να ἐμπλουτίσουμε΄ τις συναρτήσεις με επιπλέον λειτουργικότητα: f'_i `::` T_i `->m` T_{i+1}
- Θέλουμε ο υπολογισμός μας να γίνεται από `m` T_0 σε `m` T_n:

$$x' \text{ `app` } f'_0 \text{ `app` } \ldots \text{ `app` } f'_{n-1} \texttt{ :: } \texttt{m } T_n$$

  όπου x' `::m` T_0
- Χρειαζόμαστε:
  - καινούρια συνάρτηση σύνθεσης `appm`
  - συνάρτηση `liftm` που να "ανεβάζει" μια τιμή `a` σε `m a`
- Τύποι:

```
appm ::  m a->(a->m b)->m b
liftm ::  a->m a
```

- Η σύνθεση γίνεται:

$$\texttt{liftm } x \text{ `appm` } f'_0 \text{ `appm` } \ldots \text{ `appm` } f'_{n-1} \texttt{ :: m } T_n$$

- Θέλουμε να ἐμπλουτίσουμε΄ τις συναρτήσεις με επιπλέον λειτουργικότητα: f'_i `::` T_i `->m` T_{i+1}
- Θέλουμε ο υπολογισμός μας να γίνεται από `m` T_0 σε `m` T_n:

$$x' \text{ `app` } f'_0 \text{ `app` } \ldots \text{ `app` } f'_{n-1} \texttt{ :: } \texttt{m } T_n$$

  όπου x' `::m` T_0
- Χρειαζόμαστε:
  - καινούρια συνάρτηση σύνθεσης `appm`
  - συνάρτηση `liftm` που να "ανεβάζει" μια τιμή `a` σε `m a`
- Τύποι:

```
appm ::  m a->(a->m b)->m b
liftm ::  a->m a
```

- Η σύνθεση γίνεται:

$$\texttt{liftm } x \text{ `appm` } f'_0 \text{ `appm` } \ldots \text{ `appm` } f'_{n-1} \texttt{ :: m } T_n$$

- Θέλουμε να ἐμπλουτίσουμε΄ τις συναρτήσεις με επιπλέον λειτουργικότητα: f'_i `::` T_i `->m` T_{i+1}
- Θέλουμε ο υπολογισμός μας να γίνεται από `m` T_0 σε `m` T_n:

$$x' \text{ `app` } f'_0 \text{ `app` } \ldots \text{ `app` } f'_{n-1} \texttt{ :: m } T_n$$

  όπου x' `::m` T_0
- Χρειαζόμαστε:
  - καινούρια συνάρτηση σύνθεσης `appm`
  - συνάρτηση `liftm` που να "ανεβάζει" μια τιμή `a` σε `m a`
- Τύποι:

```
appm :: m a->(a->m b)->m b
liftm :: a->m a
```

- Η σύνθεση γίνεται:

```
liftm x `appm` f'0 `appm` ... `appm` f'n-1 :: m Tn
```

- Έλεγχος λαθών (m=Maybe)
- Συναρτήσεις Maybe a->Maybe b:
  - για κάθε οριζόμενη συνάρτηση f:
    ```
    f(Just x) = ...
    f Nothing = Nothing
    ```
  - για κάθε ανώνυμη συνάρτηση:
    ```
    \mbx->case mbx of Just x ->...
                      Nothing->Nothing
    ```
  - μέσα σε άλλους υπολογισμούς:
    ```
    case val of
     Nothing -> Nothing
     Just x -> case f x of
                 Nothing -> Nothing
                 Just y -> case g y of
                             Nothing -> Nothing
                             Just z -> Just z
    ```

- Έλεγχος λαθών (`m=Maybe`)
- Συναρτήσεις `Maybe a->Maybe b`:
  - για κάθε οριζόμενη συνάρτηση `f`:
    ```
    f(Just x) = ...
    f Nothing = Nothing
    ```
  - για κάθε ανώνυμη συνάρτηση:
    ```
    \mbx->case mbx of Just x ->...
                      Nothing->Nothing
    ```
  - μέσα σε άλλους υπολογισμούς:
    ```
    case val of
     Nothing -> Nothing
     Just x -> case f x of
                 Nothing -> Nothing
                 Just y -> case g y of
                             Nothing -> Nothing
                             Just z -> Just z
    ```

- Έλεγχος λαθών (m=Maybe)
- Συναρτήσεις Maybe a->Maybe b:
  - για κάθε οριζόμενη συνάρτηση f:
    ```
    f(Just x) = ...
    f Nothing = Nothing
    ```
  - για κάθε ανώνυμη συνάρτηση:
    ```
    \mbx->case mbx of Just x ->...
                      Nothing->Nothing
    ```
  - μέσα σε άλλους υπολογισμούς:
    ```
    case val of
     Nothing -> Nothing
     Just x -> case f x of
                 Nothing -> Nothing
                 Just y -> case g y of
                             Nothing -> Nothing
                             Just z -> Just z
    ```

- Έλεγχος λαθών (m=Maybe)
- Συναρτήσεις Maybe a->Maybe b:
  - για κάθε οριζόμενη συνάρτηση f:
    ```
    f(Just x) = ...
    f Nothing = Nothing
    ```
  - για κάθε ανώνυμη συνάρτηση:
    ```
    \mbx->case mbx of Just x ->...
                      Nothing->Nothing
    ```
  - μέσα σε άλλους υπολογισμούς:
    ```
    case val of
     Nothing -> Nothing
     Just x -> case f x of
                 Nothing -> Nothing
                 Just y -> case g y of
                             Nothing -> Nothing
                             Just z -> Just z
    ```

- Χρησιμοποιούμε `f :: a->Maybe b` και:
  - `liftMaybe = Just`
  - `appMaybe`
    ```
    appMaybe Nothing _ = Nothing
    appMaybe (Just x) f = f x
    ```
- Τώρα:
  - οριζόμενη συνάρτηση: `f x = ...`
  - ανώνυμη συνάρτηση: `\x->...`
  - μέσα σε άλλους υπολογισμούς:
    ```
    liftMaybe val `appMaybe` f `appMaybe` g
    ```

- Χρησιμοποιούμε `f :: a->Maybe b` και:
  - `liftMaybe = Just`
  - `appMaybe`
    ```
    appMaybe Nothing _ = Nothing
    appMaybe (Just x) f = f x
    ```
- Τώρα:
  - οριζόμενη συνάρτηση: `f x = ...`
  - ανώνυμη συνάρτηση: `\x->...`
  - μέσα σε άλλους υπολογισμούς:
    ```
    liftMaybe val ‘appMaybe‘ f ‘appMaybe‘ g
    ```

- Χρησιμοποιούμε `f :: a->Maybe b` και:
  - `liftMaybe = Just`
  - `appMaybe`
    ```
    appMaybe Nothing _ = Nothing
    appMaybe (Just x) f = f x
    ```
- Τώρα:
  - οριζόμενη συνάρτηση: `f x = ...`
  - ανώνυμη συνάρτηση: `\x->...`
  - μέσα σε άλλους υπολογισμούς:
    ```
    liftMaybe val ‘appMaybe‘ f ‘appMaybe‘ g
    ```

- Χρησιμοποιούμε `f :: a->Maybe b` και:
  - `liftMaybe = Just`
  - `appMaybe`
    ```
    appMaybe Nothing _ = Nothing
    appMaybe (Just x) f = f x
    ```
- Τώρα:
  - οριζόμενη συνάρτηση: `f x = ...`
  - ανώνυμη συνάρτηση: `\x->...`
  - μέσα σε άλλους υπολογισμούς:
    ```
    liftMaybe val `appMaybe` f `appMaybe` g
    ```

- Χρησιμοποιούμε `f :: a->Maybe b` και:
  - `liftMaybe = Just`
  - `appMaybe`
    ```
    appMaybe Nothing _ = Nothing
    appMaybe (Just x) f = f x
    ```
- Τώρα:
  - οριζόμενη συνάρτηση: `f x = ...`
  - ανώνυμη συνάρτηση: `\x->...`
  - μέσα σε άλλους υπολογισμούς:
    ```
    liftMaybe val `appMaybe` f `appMaybe` g
    ```

- Χρησιμοποιούμε `f :: a->Maybe b` και:
  - `liftMaybe = Just`
  - `appMaybe`
    ```
    appMaybe Nothing _ = Nothing
    appMaybe (Just x) f = f x
    ```
- Τώρα:
  - οριζόμενη συνάρτηση: `f x = ...`
  - ανώνυμη συνάρτηση: `\x->...`
  - μέσα σε άλλους υπολογισμούς:
    ```
    liftMaybe val `appMaybe` f `appMaybe` g
    ```

- Έστω

```
saferev x = if x==0 then Nothing else Just(1/x)
```

- Ασφαλής υπολογισμός του 1/(x*z)

```
liftMaybe x `appMaybe` saferev
          `appMaybe` \rx->liftMaybe z
                     `appMaybe` saferev
                     `appMaybe`
                      \rz->liftMaybe(rx*rz)
```

- Σαφέστερος κώδικας:

```
liftMaybe x `appMaybe` \x->
saferev x  `appMaybe` \rx->
liftMaybe z `appMaybe` \z->
saferev z  `appMaybe` \rz->
liftMaybe (rx*rz)
```

- Έστω

```
saferev x = if x==0 then Nothing else Just(1/x)
```

- Ασφαλής υπολογισμός του 1/(x*z)

```
liftMaybe x ‘appMaybe‘ saferev
            ‘appMaybe‘ \rx->liftMaybe z
                         ‘appMaybe‘ saferev
                         ‘appMaybe‘
                          \rz->liftMaybe(rx*rz)
```

- Σαφέστερος κώδικας:

```
liftMaybe x ‘appMaybe‘ \x->
saferev x  ‘appMaybe‘ \rx->
liftMaybe z ‘appMaybe‘ \z->
saferev z  ‘appMaybe‘ \rz->
liftMaybe (rx*rz)
```

- Έστω

```
saferev x = if x==0 then Nothing else Just(1/x)
```

- Ασφαλής υπολογισμός του 1/(x*z)

```
liftMaybe x `appMaybe` saferev
            `appMaybe` \rx->liftMaybe z
                         `appMaybe` saferev
                         `appMaybe`
                          \rz->liftMaybe(rx*rz)
```

- Σαφέστερος κώδικας:

```
liftMaybe x `appMaybe` \x->
saferev x  `appMaybe` \rx->
liftMaybe z `appMaybe` \z->
saferev z  `appMaybe` \rz->
liftMaybe (rx*rz)
```

- Θέλουμε να προσθέσουμε δυνατότητες logging στις συναρτήσεις μας
  - παράδειγμα:
    ```
    f x = (...some_result...  , "f was called")
    ```
- `m=Log` όπου
  ```
  type Log a = (a,String)
  ```
- Συναρτήσεις `liftLog` και `appLog`:
  ```
  liftLog r = (r,"")
  appLog (r,s) f =
     (r',s++s') where (r',s')=f r
  ```

- Θέλουμε να προσθέσουμε δυνατότητες logging στις συναρτήσεις μας
  - παράδειγμα:
    ```
    f x = (...some_result...  , "f was called")
    ```
- `m=Log` όπου
  ```
  type Log a = (a,String)
  ```
- Συναρτήσεις `liftLog` και `appLog`:
  ```
  liftLog r = (r,"")
  appLog (r,s) f =
     (r',s++s') where (r',s')=f r
  ```

- Θέλουμε να προσθέσουμε δυνατότητες logging στις συναρτήσεις μας
  - παράδειγμα:
    ```
    f x = (...some_result...  , "f was called")
    ```
- `m=Log` όπου
  ```
  type Log a = (a,String)
  ```
- Συναρτήσεις `liftLog` και `appLog`:
  ```
  liftLog r = (r,"")
  appLog (r,s) f =
     (r',s++s') where (r',s')=f r
  ```

- Θέλουμε να προσθέσουμε δυνατότητες logging στις συναρτήσεις μας
  - παράδειγμα:
    ```
    f x = (...some_result...  , "f was called")
    ```
- `m=Log` όπου
  ```
  type Log a = (a,String)
  ```
- Συναρτήσεις `liftLog` και `appLog`:
  ```
  liftLog r = (r,"")
  appLog (r,s) f =
     (r',s++s') where (r',s')=f r
  ```

- Ορισμοί με logging:

```
f n =
  (n*2, "f(" ++ show n ++ ") was called.  ")
g n =
  (n+1, "g(" ++ show n ++ ") was called.  ")
```

- Χρήση:

```
liftLog 10 ‘appLog‘ \x->
f x ‘appLog‘ \y->
g y ‘appLog‘ \z-> liftLog(x+y+z)
```

επιστρέφει

```
(51,"f(10) was called.  g(20) was called.  ")
```

- Ορισμοί με logging:

```
f n =
  (n*2, "f(" ++ show n ++ ") was called.  ")
g n =
  (n+1, "g(" ++ show n ++ ") was called.  ")
```

- Χρήση:

```
liftLog 10 ‘appLog‘ \x->
f x ‘appLog‘ \y->
g y ‘appLog‘ \z-> liftLog(x+y+z)
```

επιστρέφει

```
(51,"f(10) was called.  g(20) was called.  ")
```

Ορισμοί με αναδρομή:

```
logg str = (undef,str)
factorial 0 =
  logg ("factorial(0) was called.  ") `appLog` \_->
  liftLog 1
factorial n =
  logg("factorial(" ++ show n ++ ") was called.  ")
      `appLog` \_->
  factorial(n-1) `appLog` \x->
  liftLog (x*n)
```

- Για την υποστήριξη της νέας λειτουργικότητας `m` χρειαζόμαστε
  - `liftm :: a->m a`
  - `appm :: m a->(a->m b)->m b`
- Οι συναρτήσεις πρέπει να ικανοποιούν:

```
liftm x `appm` f = f x
mx `appm` liftm = mx
mx `appm` (\x -> f x `appm` g)
   = (mx `appm` f) `appm` g
```

- Η τριάδα (`m, liftm, appm`) ονομάζεται μονάδα (monad)

- Για την υποστήριξη της νέας λειτουργικότητας `m` χρειαζόμαστε
  - `liftm :: a->m a`
  - `appm :: m a->(a->m b)->m b`
- Οι συναρτήσεις πρέπει να ικανοποιούν:

```
liftm x `appm` f = f x
mx `appm` liftm = mx
mx `appm` (\x -> f x `appm` g)
   = (mx `appm` f) `appm` g
```

- Η τριάδα (`m, liftm, appm`) ονομάζεται μονάδα (monad)

- Για την υποστήριξη της νέας λειτουργικότητας `m` χρειαζόμαστε
  - `liftm :: a->m a`
  - `appm :: m a->(a->m b)->m b`
- Οι συναρτήσεις πρέπει να ικανοποιούν:

```
liftm x `appm` f = f x
mx `appm` liftm = mx
mx `appm` (\x -> f x `appm` g)
   = (mx `appm` f) `appm` g
```

- Η τριάδα (`m, liftm, appm`) ονομάζεται μονάδα (monad)

- Για την υποστήριξη της νέας λειτουργικότητας `m` χρειαζόμαστε
  - `liftm :: a->m a`
  - `appm :: m a->(a->m b)->m b`
- Οι συναρτήσεις πρέπει να ικανοποιούν:

```
liftm x `appm` f = f x
mx `appm` liftm = mx
mx `appm` (\x -> f x `appm` g)
   = (mx `appm` f) `appm` g
```

- Η τριάδα (`m, liftm, appm`) ονομάζεται μονάδα (monad)

- Για την υποστήριξη της νέας λειτουργικότητας `m` χρειαζόμαστε
  - `liftm :: a->m a`
  - `appm :: m a->(a->m b)->m b`
- Οι συναρτήσεις πρέπει να ικανοποιούν:

```
liftm x `appm` f = f x
mx `appm` liftm = mx
mx `appm` (\x -> f x `appm` g)
   = (mx `appm` f) `appm` g
```

- Η τριάδα (`m, liftm, appm`) ονομάζεται μονάδα (monad)

- Για την υποστήριξη της νέας λειτουργικότητας `m` χρειαζόμαστε
  - `liftm :: a->m a`
  - `appm :: m a->(a->m b)->m b`
- Οι συναρτήσεις πρέπει να ικανοποιούν:

```
liftm x `appm` f = f x
mx `appm` liftm = mx
mx `appm` (\x -> f x `appm` g)
   = (mx `appm` f) `appm` g
```

- Η τριάδα (`m, liftm, appm`) ονομάζεται μονάδα (monad)

Κατασκευαστές Τύπων και Υπερφόρτωση

- Ένας κατασκευαστής τύπου (type constructor) είναι μία συνάρτηση από τύπους σε τύπο
- Παραδείγματα:

```
[]
data Maybe a = Just a | Nothing
```

- Υπερφόρτωση και κλάσεις υποστηρίζονται και σε κατασκευαστές τύπων:

```
class SomeClass cons where
   somefunction ::  cons a -> String
instance SomeClass Maybe where
   somefunction (Just _) = "Correct"
   somefunction Nothing = "Error"
instance SomeClass [] where
   somefunction [] = "Empty"
   somefunction _ = "Non-empty"
```

Κατασκευαστές Τύπων και Υπερφόρτωση

- Ένας κατασκευαστής τύπου (type constructor) είναι μία συνάρτηση από τύπους σε τύπο
- Παραδείγματα:

```
[]
data Maybe a = Just a | Nothing
```

- Υπερφόρτωση και κλάσεις υποστηρίζονται και σε κατασκευαστές τύπων:

```
class SomeClass cons where
   somefunction ::  cons a -> String
instance SomeClass Maybe where
   somefunction (Just _) = "Correct"
   somefunction Nothing = "Error"
instance SomeClass [] where
   somefunction [] = "Empty"
   somefunction _ = "Non-empty"
```

Κατασκευαστές Τύπων και Υπερφόρτωση

- Ένας κατασκευαστής τύπου (type constructor) είναι μία συνάρτηση από τύπους σε τύπο
- Παραδείγματα:

```
[]
data Maybe a = Just a | Nothing
```

- Υπερφόρτωση και κλάσεις υποστηρίζονται και σε κατασκευαστές τύπων:

```
class SomeClass cons where
   somefunction ::  cons a -> String
instance SomeClass Maybe where
   somefunction (Just _) = "Correct"
   somefunction Nothing = "Error"
instance SomeClass [] where
   somefunction [] = "Empty"
   somefunction _ = "Non-empty"
```

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

Η κλάση Monad

- Η κλάση κατασκευαστών τύπων `Monad` έχει στιγμιότυπα μονάδες
  - η συνάρτηση `liftm` ονομάζεται `return`
  - η συνάρτηση `appm` γίνεται τελεστής `>>=`
  - υποστηρίζεται τελεστής `>>` που πετάει την τιμή που επιστρέφει το αριστερό του όρισμα
  - οι ιδιότητες μονάδας δε μπορούν να επαληθευτούν από τη Haskell

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

- Ο κατασκευαστής `Maybe` είναι ήδη μονάδα:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= k = Nothing
  return = Just
```

- Παράδειγμα υπολογισμού `1/(x*z)`:

```
return x >>= \x->
saferev x  >>= \rx->
return z >>= \z->
saferev z  >>= \rz->
return (rx*rz)
```

- Ο κατασκευαστής `Maybe` είναι ήδη μονάδα:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= k = Nothing
  return = Just
```

- Παράδειγμα υπολογισμού `1/(x*z)`:

```
return x >>= \x->
saferev x  >>= \rx->
return z >>= \z->
saferev z  >>= \rz->
return (rx*rz)
```

```
data Log a = Log (a, String)
instance Monad Log where
  return r = Log (r,"")
  (Log (r,s)) >>= k =
     Log (r',s++s') where Log (r',s')=k r
logg str = Log (undef,str)

f n =
  logg("f(" ++ show n ++ ") was called.  ") >>
  return (2*n)
g n =
  logg("g(" ++ show n ++ ") was called.  ") >>
  return (n+1)
```

```
data Log a = Log (a, String)
instance Monad Log where
  return r = Log (r,"")
  (Log (r,s)) >>= k =
     Log (r',s++s') where Log (r',s')=k r
logg str = Log (undef,str)

f n =
  logg("f(" ++ show n ++ ") was called.  ") >>
  return (2*n)
g n =
  logg("g(" ++ show n ++ ") was called.  ") >>
  return (n+1)
```

```
 exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)

factorial 0 =
  logg"factorial(0) was called.  " >>
  return 1
factorial n =
  logg
    ("factorial(" ++ show n ++ ") was called.  ")
    >>
  factorial(n-1) >>= \x->
  return(x*n)
```

Σύνταξη do

Βασική Σύνταξη

- Η σύνταξη `do` για τις μονάδες καθαρίζει τον κώδικα από τη συνεχή χρήση των >>= και >>
  - ξεκινάμε με τη λέξη `do`
  - ακολουθούν ευθυγραμμισμένες σε γραμμές οι εντολές, χωρίς >> και >>= (που υπονοούνται)
  - η σύνταξη

    ```
    command >>= \variable->
    ```

    αντικαθίσταται από

    ```
    variable <- command
    ```

Σύνταξη do

Βασική Σύνταξη

- Η σύνταξη `do` για τις μονάδες καθαρίζει τον κώδικα από τη συνεχή χρήση των >>= και >>
  - ξεκινάμε με τη λέξη `do`
  - ακολουθούν ευθυγραμμισμένες σε γραμμές οι εντολές, χωρίς >> και >>= (που υπονοούνται)
  - η σύνταξη

    ```
    command >>= \variable->
    ```

    αντικαθίσταται από

    ```
    variable <- command
    ```

- Η σύνταξη `do` για τις μονάδες καθαρίζει τον κώδικα από τη συνεχή χρήση των >>= και >>
  - ξεκινάμε με τη λέξη `do`
  - ακολουθούν ευθυγραμμισμένες σε γραμμές οι εντολές, χωρίς >> και >>= (που υπονοούνται)
  - η σύνταξη
    ```
    command >>= \variable->
    ```
    αντικαθίσταται από
    ```
    variable <- command
    ```

- Η σύνταξη `do` για τις μονάδες καθαρίζει τον κώδικα από τη συνεχή χρήση των >>= και >>
  - ξεκινάμε με τη λέξη `do`
  - ακολουθούν ευθυγραμμισμένες σε γραμμές οι εντολές, χωρίς >> και >>= (που υπονοούνται)
  - η σύνταξη
    ```
    command >>= \variable->
    ```
    αντικαθίσταται από
    ```
    variable <- command
    ```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - I

```
f n =
  logg("f(" ++ show n ++ ") was called.  ") >>
  return (2*n)
g n =
  logg("g(" ++ show n ++ ") was called.  ") >>
  return (n+1)
```

μετατρέπεται σε:

```
f n =
  do logg("f(" ++ show n ++ ") was called.  ")
     return (2*n)
g n =
  do logg("g(" ++ show n ++ ") was called.  ")
     return (n+1)
```

```
f n =
  logg("f(" ++ show n ++ ") was called.  ") >>
  return (2*n)
g n =
  logg("g(" ++ show n ++ ") was called.  ") >>
  return (n+1)
```

μετατρέπεται σε:

```
f n =
  do logg("f(" ++ show n ++ ") was called.  ")
     return (2*n)
g n =
  do logg("g(" ++ show n ++ ") was called.  ")
     return (n+1)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do

Βασική Σύνταξη - Παραδείγματα - II

```
exLog =
  return 10 >>= \x->
  f x >>= \y->
  g y >>= \z->
  return(x+y+z)
```

```
exLog =
  do x<-return 10
     y<-f x
     z<-g y
     return(x+y+z)
```

```
factorial 0 =
  logg"..." >>
  return 1
factorial n =
  logg("...") >>
  factorial(n-1) >>= \x->
  return(x*n)
```

```
factorial 0 =
  do logg"..."
     return 1
factorial n =
  do logg("...")
     x<-factorial(n-1)
     return(x*n)
```

Σύνταξη do
let

`do let` *v* `=` *E*

κώδικας

ισοδυναμεί με

`(\`*v*`->do` *κώδικας*`)`*E*

Παράδειγμα:

```
exLog =
  do let x=10
     y<-f x
     z<-g y
     return(x+y+z)
```

Σύνταξη do
let

`do let` *v* `=` *E*

κώδικας

ισοδυναμεί με

`(\`*v*`->do` *κώδικας*`)`*E*

Παράδειγμα:

```
exLog =
  do let x=10
     y<-f x
     z<-g y
     return(x+y+z)
```

Σύνταξη do

Single Assignment - I

- Η σύνταξη do μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Οι μεταβλητές δεν αντιστοιχούν σε θέσεις μνήμης!
- Αυτοσχέδιο while:

```
while test action =
  do c<-test
     if c then do action
                  while test action
          else return 0
```

Σύνταξη do

Single Assignment - I

- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Οι μεταβλητές δεν αντιστοιχούν σε θέσεις μνήμης!
- Αυτοσχέδιο `while`:

```
while test action =
  do c<-test
     if c then do action
                  while test action
          else return 0
```

Σύνταξη do

Single Assignment - II

- Λάθος χρήση:

```
badfact n =
  do let s = 1
     while (return (n>0))
           (do let s = s*n
               let n = n-1
               logg "X"
           )
     return s
```

- `badfact 0` επιστρέφει 1, αλλά `badfact 2` κολλάει!
- Single Assignment: κάθε νέα εμφάνιση του n είναι νέα μεταβλητή
- Μόνο αναδρομικές λύσεις!

Σύνταξη do

Single Assignment - II

- Λάθος χρήση:

```
badfact n =
  do let s = 1
     while (return (n>0))
           (do let s = s*n
               let n = n-1
               logg "X"
           )
     return s
```

- `badfact 0` επιστρέφει 1, αλλά `badfact 2` κολλάει!
- Single Assignment: κάθε νέα εμφάνιση του n είναι νέα μεταβλητή
- Μόνο αναδρομικές λύσεις!

Σύνταξη do

Single Assignment - II

- Λάθος χρήση:

```
badfact n =
  do let s = 1
     while (return (n>0))
           (do let s = s*n
               let n = n-1
               logg "X"
           )
     return s
```

- `badfact 0` επιστρέφει 1, αλλά `badfact 2` κολλάει!
- Single Assignment: κάθε νέα εμφάνιση του n είναι νέα μεταβλητή
- Μόνο αναδρομικές λύσεις!

Σύνταξη do

Single Assignment - II

- Λάθος χρήση:

```
badfact n =
  do let s = 1
     while (return (n>0))
           (do let s = s*n
               let n = n-1
               logg "X"
           )
     return s
```

- `badfact 0` επιστρέφει `1`, αλλά `badfact 2` κολλάει!
- Single Assignment: κάθε νέα εμφάνιση του `n` είναι νέα μεταβλητή
- Μόνο αναδρομικές λύσεις!

Σύνταξη do

Single Assignment - II

- Λάθος χρήση:

```
badfact n =
  do let s = 1
     while (return (n>0))
           (do let s = s*n
               let n = n-1
               logg "X"
           )
     return s
```

- `badfact 0` επιστρέφει `1`, αλλά `badfact 2` κολλάει!
- Single Assignment: κάθε νέα εμφάνιση του `n` είναι νέα μεταβλητή
- Μόνο αναδρομικές λύσεις!

Άλλες Μονάδες

- Ο κατασκευαστής `[]` είναι μονάδα:

```
instance Monad [] where
  return x = [x]
  m >>= k = concat (map k m)
```

- Χρησιμοποίηση για μη ντετερμινιστικούς υπολογισμούς:

```
f x = [x,x+1]
g x = [x,2*x]
s = do x<-return 5
       y<-f x
       z<-g y
       return z
```

αποτέλεσμα `s=[5,6,10,12]`

- Τετριμμένη μονάδα `Id`:

```
data Id a = Id a
instance Monad Id where
  return = Id
  (Id x) >>= k = k x
```

Άλλες Μονάδες

- Ο κατασκευαστής `[]` είναι μονάδα:

```
instance Monad [] where
  return x = [x]
  m >>= k = concat (map k m)
```

- Χρησιμοποίηση για μη ντετερμινιστικούς υπολογισμούς:

```
f x = [x,x+1]
g x = [x,2*x]
s = do x<-return 5
       y<-f x
       z<-g y
       return z
```

αποτέλεσμα `s=[5,6,10,12]`

- Τετριμμένη μονάδα `Id`:

```
data Id a = Id a
instance Monad Id where
  return = Id
  (Id x) >>= k = k x
```

Άλλες Μονάδες

- Ο κατασκευαστής `[]` είναι μονάδα:

```
instance Monad [] where
  return x = [x]
  m >>= k = concat (map k m)
```

- Χρησιμοποίηση για μη ντετερμινιστικούς υπολογισμούς:

```
f x = [x,x+1]
g x = [x,2*x]
s = do x<-return 5
       y<-f x
       z<-g y
       return z
```

αποτέλεσμα `s=[5,6,10,12]`

- Τετριμμένη μονάδα `Id`:

```
data Id a = Id a
instance Monad Id where
  return = Id
  (Id x) >>= k = k x
```

- Μέχρι τώρα δεν έχουμε επίδραση με το χρήστη κατά τη διάρκεια εκτέλεσης ενός προγράμματος
  - μη επαρκές για μεγάλα προγράμματα
- Η αλληλεπίδραση με το περιβάλλον δεν ταιριάζει στο συναρτησιακό στυλ:
  `read - read`
  - η σειρά αποτίμησης επηρρεάζει το αποτέλεσμα υπολογισμών
  - παρενέργειες: το παραπάνω δεν ισούται με `0`
  - ακόμα χειρότερες παρενέργειες: `f = read + 42`

- Μέχρι τώρα δεν έχουμε επίδραση με το χρήστη κατά τη διάρκεια εκτέλεσης ενός προγράμματος
  - μη επαρκές για μεγάλα προγράμματα
- Η αλληλεπίδραση με το περιβάλλον δεν ταιριάζει στο συναρτησιακό στυλ:
  `read - read`
  - η σειρά αποτίμησης επηρρεάζει το αποτέλεσμα υπολογισμών
  - παρενέργειες: το παραπάνω δεν ισούται με `0`
  - ακόμα χειρότερες παρενέργειες: `f = read + 42`

- Προσέγγιση (α) (ML, Lisp): αποδεχόμαστε τις παρενέργειες
- Προσέγγιση (β) (παλιά Haskell): μοντελοποιούμε την είσοδο και την έξοδο σαν οκνηρούς άπειρους πίνακες
  - πρόβλημα με το χρονισμό των πράξεων: είσοδος μπορεί να είναι διαθέσιμη πριν αυτό να είναι επιθυμητό
- Μαθηματική προσέγγιση: οι συναρτήσεις `read` διαβάζουν τον "κόσμο" και οι συναρτήσεις `write` αλλάζουν τον "κόσμο":

```
read :: World->(InputType,World)
write :: (OutputType,World)->World
```

  - ο χρήστης πρέπει να εμποδιστεί από το να χρησιμοποιήσει τον ίδιο κόσμο δύο φορές: εφικτό μαθηματικά, αλλά ανέφικτο στην πραγματικότητα
- Λύση: χρήση ΑΤΔ `IO a`

- Προσέγγιση (α) (ML, Lisp): αποδεχόμαστε τις παρενέργειες
- Προσέγγιση (β) (παλιά Haskell): μοντελοποιούμε την είσοδο και την έξοδο σαν οκνηρούς άπειρους πίνακες
  - πρόβλημα με το χρονισμό των πράξεων: είσοδος μπορεί να είναι διαθέσιμη πριν αυτό να είναι επιθυμητό
- Μαθηματική προσέγγιση: οι συναρτήσεις `read` διαβάζουν τον "κόσμο" και οι συναρτήσεις `write` αλλάζουν τον "κόσμο":

```
read :: World->(InputType,World)
write :: (OutputType,World)->World
```

  - ο χρήστης πρέπει να εμποδιστεί από το να χρησιμοποιήσει τον ίδιο κόσμο δύο φορές: εφικτό μαθηματικά, αλλά ανέφικτο στην πραγματικότητα
- Λύση: χρήση ΑΤΔ `IO a`

- Προσέγγιση (α) (ML, Lisp): αποδεχόμαστε τις παρενέργειες
- Προσέγγιση (β) (παλιά Haskell): μοντελοποιούμε την είσοδο και την έξοδο σαν οκνηρούς άπειρους πίνακες
  - πρόβλημα με το χρονισμό των πράξεων: είσοδος μπορεί να είναι διαθέσιμη πριν αυτό να είναι επιθυμητό
- Μαθηματική προσέγγιση: οι συναρτήσεις `read` διαβάζουν τον "κόσμο" και οι συναρτήσεις `write` αλλάζουν τον "κόσμο":

```
read :: World->(InputType,World)
write :: (OutputType,World)->World
```

  - ο χρήστης πρέπει να εμποδιστεί από το να χρησιμοποιήσει τον ίδιο κόσμο δύο φορές: εφικτό μαθηματικά, αλλά ανέφικτο στην πραγματικότητα
- Λύση: χρήση ΑΤΔ `IO a`

- Προσέγγιση (α) (ML, Lisp): αποδεχόμαστε τις παρενέργειες
- Προσέγγιση (β) (παλιά Haskell): μοντελοποιούμε την είσοδο και την έξοδο σαν οκνηρούς άπειρους πίνακες
  - πρόβλημα με το χρονισμό των πράξεων: είσοδος μπορεί να είναι διαθέσιμη πριν αυτό να είναι επιθυμητό
- Μαθηματική προσέγγιση: οι συναρτήσεις `read` διαβάζουν τον "κόσμο" και οι συναρτήσεις `write` αλλάζουν τον "κόσμο":

```
read :: World->(InputType,World)
write :: (OutputType,World)->World
```

  - ο χρήστης πρέπει να εμποδιστεί από το να χρησιμοποιήσει τον ίδιο κόσμο δύο φορές: εφικτό μαθηματικά, αλλά ανέφικτο στην πραγματικότητα
- Λύση: χρήση ΑΤΔ `IO a`

- `IO a`: κώδικας που αλληλεπιδρά με το χρήστη και επιστρέφει τιμή τύπου `a`
- Μη επιστροφή τιμής → χρήση τύπου "μηδενικής πλειάδας" `()`. Όπως `void` στις C/C++/Java.
- `IO` μονάδα → σειρά εντολών είναι απόλυτα σαφής
- Εντολές εισόδου/εξόδου:

```
getChar ::  IO Char
getLine ::  IO String
putChar ::  Char->IO()
putStr ::  String->IO()
isEOF ::  IO Bool
```

- Εισαγωγή τμήματος:

```
import IO
```

- `IO a`: κώδικας που αλληλεπιδρά με το χρήστη και επιστρέφει τιμή τύπου `a`
- Μη επιστροφή τιμής → χρήση τύπου "μηδενικής πλειάδας" `()`. Όπως `void` στις C/C++/Java.
- `IO` μονάδα → σειρά εντολών είναι απόλυτα σαφής
- Εντολές εισόδου/εξόδου:

```
getChar :: IO Char
getLine :: IO String
putChar :: Char->IO()
putStr :: String->IO()
isEOF :: IO Bool
```

- Εισαγωγή τμήματος:

```
import IO
```

- `IO a`: κώδικας που αλληλεπιδρά με το χρήστη και επιστρέφει τιμή τύπου `a`
- Μη επιστροφή τιμής → χρήση τύπου "μηδενικής πλειάδας" `()`. Όπως `void` στις C/C++/Java.
- `IO` μονάδα → σειρά εντολών είναι απόλυτα σαφής
- Εντολές εισόδου/εξόδου:

```
getChar :: IO Char
getLine :: IO String
putChar :: Char->IO()
putStr :: String->IO()
isEOF :: IO Bool
```

- Εισαγωγή τμήματος:

```
import IO
```

- `IO a`: κώδικας που αλληλεπιδρά με το χρήστη και επιστρέφει τιμή τύπου `a`
- Μη επιστροφή τιμής → χρήση τύπου "μηδενικής πλειάδας" `()`. Όπως `void` στις C/C++/Java.
- `IO` μονάδα → σειρά εντολών είναι απόλυτα σαφής
- Εντολές εισόδου/εξόδου:

```
getChar ::  IO Char
getLine ::  IO String
putChar ::  Char->IO()
putStr ::  String->IO()
isEOF ::  IO Bool
```

- Εισαγωγή τμήματος:

```
import IO
```

- `IO a`: κώδικας που αλληλεπιδρά με το χρήστη και επιστρέφει τιμή τύπου `a`
- Μη επιστροφή τιμής → χρήση τύπου "μηδενικής πλειάδας" `()`. Όπως `void` στις C/C++/Java.
- `IO` μονάδα → σειρά εντολών είναι απόλυτα σαφής
- Εντολές εισόδου/εξόδου:

```
getChar ::  IO Char
getLine ::  IO String
putChar ::  Char->IO()
putStr ::  String->IO()
isEOF ::  IO Bool
```

- Εισαγωγή τμήματος:

```
import IO
```

```
import IO
copy =
  do e<-isEOF
     if e then do return ()
          else do c<-getChar
                  putChar c
                  copy
```

- Είσοδος και έξοδος χωρίς file handles:

```
readFile ::  FilePath -> IO String
writeFile ::  FilePath -> String -> IO ()
appendFile ::  FilePath -> String -> IO ()
```

- Υπάρχει και interface για άνοιγμα αρχείων με handles όμοιο με αυτό της C

- Είσοδος και έξοδος χωρίς file handles:

```
readFile ::  FilePath -> IO String
writeFile ::  FilePath -> String -> IO ()
appendFile ::  FilePath -> String -> IO ()
```

- Υπάρχει και interface για άνοιγμα αρχείων με handles όμοιο με αυτό της C

Επισκόπηση

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`

Επισκόπηση

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`

Επισκόπηση

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`

Επισκόπηση

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`

Επισκόπηση

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`

Επισκόπηση

- Χρησιμοποιούμε μονάδες για να εμπλουτίσουμε τις συναρτήσεις μας με καινούργια λειτουργικότητα διαχωρισμένη από τη βασική τους λειτουργία
- Μία μονάδα είναι ένας μοναδιαίος κατασκευαστής τύπου `m` συνοδευόμενος από συναρτήσεις `return` και `>>=` που ικανοποιούν συγκεκριμένες ιδιότητες
- Η κλάση μοναδιαίων κατασκευαστών `Monad` της Haskell υπερφορτώνει αυτές τις συναρτήσεις
- Η σύνταξη `do` για τις μονάδες απλοποιεί σημαντικά τον κώδικα
- Η σύνταξη `do` μοιάζει αλλά δεν είναι προστακτικός προγραμματισμός. Δεν υπάρχει κατάσταση και μία μεταβλητή παίρνει τιμή μόνο μία φορά
- Η βασική χρήση των μονάδων στη Haskell είναι η είσοδος/έξοδος που γίνεται με τη βοήθεια του ΑΤΔ/μονάδας `IO`