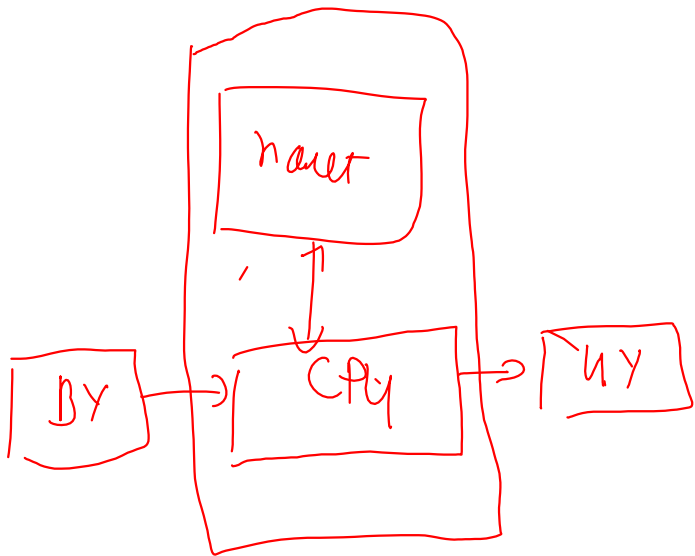


# Входно-изходни операции в Haskell

Трифон Трифонов

Функционално програмиране, спец. Информатика, 2016/17 г.

19 януари 2017 г.



# Странични ефекти в Haskell

- Функциите в Haskell нямат странични ефекти

# Странични ефекти в Haskell

- Функциите в Haskell нямат странични ефекти
- Но входно-изходните операции по природа са странични ефекти!



# Странични ефекти в Haskell

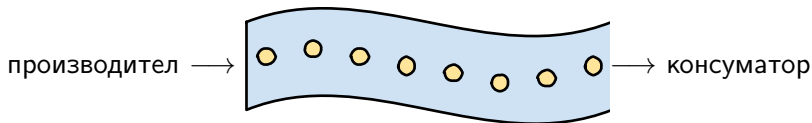
- Функциите в Haskell нямат странични ефекти
- Но входно-изходните операции по природа са странични ефекти!
- Как можем да се справим с този парадокс?

# Странични ефекти в Haskell

- Функциите в Haskell нямат странични ефекти
- Но входно-изходните операции по природа са странични ефекти!
- Как можем да се справим с този парадокс?
- **Идея:** Можем да си мислим за входно-изходните операции като поточна обработка на данни

# Странични ефекти в Haskell

- Функциите в Haskell нямат странични ефекти
- Но входно-изходните операции по природа са странични ефекти!
- Как можем да се справим с този парадокс?
- **Идея:** Можем да си мислим за входно-изходните операции като поточна обработка на данни



## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

**Решение:** Дефинираме трансформация над стандартните вход и изход, която:

- приема  $n$  като параметър

## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

**Решение:** Дефинираме трансформация над стандартните вход и изход, която:

- приема  $n$  като параметър
- трансформира входния поток, като **консумира** от него  $n$  числа, записвайки ги в списък

## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

**Решение:** Дефинираме трансформация над стандартните вход и изход, която:

- приема  $n$  като параметър
- трансформира входния поток, като **консумира** от него  $n$  числа, записвайки ги в списък
- пресмята средното аритметично `avg` на числата в списъка

## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

**Решение:** Дефинираме трансформация над стандартните вход и изход, която:

- приема  $n$  като параметър
- трансформира входния поток, като **консумира** от него  $n$  числа, записвайки ги в списък
- пресмята средното аритметично `avg` на числата в списъка
- трансформира изходния поток, като **произвежда** върху него низовото представяне на `avg`



## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

**Решение:** Дефинираме трансформация над стандартните вход и изход, която:

- приема  $n$  като параметър
- трансформира входния поток, като **консумира** от него  $n$  числа, записвайки ги в списък
- пресмята средното аритметично `avg` на числата в списъка
- трансформира изходния поток, като **произвежда** върху него низовото представяне на `avg`

**Трансформирането** на входно-изходните потоци несъмнено е страничен ефект, но **конструирането на трансформацията** няма нужда от странични ефекти!

## Поточна обработка

**Задача.** Да се въведат  $n$  числа и да се изведе тяхното средно аритметично.

**Решение:** Дефинираме трансформация над стандартните вход и изход, която:

- приема  $n$  като параметър
- трансформира входния поток, като **консумира** от него  $n$  числа, записвайки ги в списък
- пресмята средното аритметично `avg` на числата в списъка
- трансформира изходния поток, като **произвежда** върху него низовото представяне на `avg`

**Трансформирането** на входно-изходните потоци несъмнено е страничен ефект, но **конструирането** на трансформацията няма нужда от странични ефекти!

**Функциите, които работят с вход и изход, по същество дефинират композиция на входно-изходни трансформации.**

## Типът IO a

Стандартният генеричен тип `IO` а задава тип на входно/изходна трансформация, резултатът от която е от тип `a`.

## Типът IO a

Стандартният генеричен тип `IO a` задава тип на входно/изходна трансформация, резултатът от която е от тип `a`.

**Частен случай:** `IO ()` задава трансформация, която връща празен резултат.

## Типът IO a

Стандартният генеричен тип `IO a` задава тип на входно/изходна трансформация, резултатът от която е от тип `a`.

**Частен случай:** `IO ()` задава трансформация, която връща празен резултат.

### Входни трансформации:

- `getChar :: IO Char` — връща символ, прочетен от входа
- `getLine :: IO String` — връща ред, прочетен от входа

## Типът IO a

Стандартният генеричен тип `IO a` задава тип на входно/изходна трансформация, резултатът от която е от тип `a`.

**Частен случай:** `IO ()` задава трансформация, която връща празен резултат.

### Входни трансформации:

- `getChar :: IO Char` — връща символ, прочетен от входа
- `getLine :: IO String` — връща ред, прочетен от входа

### Изходни трансформации:

- `putChar :: Char -> IO ()` — извежда символ на изхода
- `putStr :: String -> IO ()` — извежда низ на изхода
- `putStrLn :: String -> IO ()` — извежда ред на изхода

## Главна функция main

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.

## Главна функция `main`

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.



## Главна функция `main`

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`

## Главна функция main

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`
- Можем ли да дефинираме  
`main = putStrLn ("Въведохте: " ++ getLine)?`

## Главна функция main

- Функцията `main :: IO ()` от модула `Main` в Haskell е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`
- Можем ли да дефинираме `main = putStrLn ("Въведохте: " ++ getLine)?`
- **Не!** `getLine :: IO String`

## Главна функция main

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`
- Можем ли да дефинираме  
`main = putStrLn ("Въведохте: " ++ getLine)?`
- **Не!** `getLine :: IO String`
- Композицията на входно-изходни трансформации не работи като композицията на функции

## Главна функция main

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`
- Можем ли да дефинираме  
`main = putStrLn ("Въведохте: " ++ getLine)?`
- **He!** `getLine :: IO String`
- Композицията на входно-изходни трансформации не работи като композицията на функции
- Низът, който връща `getLine` е “замърсен” от входно-изходна операция

## Главна функция main

- Функцията `main :: IO ()` от модула `Main` в `Haskell` е специална: тя е входната точка на компилираната програма.
- По същество тя дефинира входно-изходна трансформация, която се прилага към стандартния вход и изход при изпълнение на програмата.
- **Пример:** `main = putStrLn "Hello, world!"`
- Можем ли да дефинираме  
`main = putStrLn ("Въведохте: " ++ getLine)?`
- **He!** `getLine :: IO String`
- Композицията на входно-изходни трансформации не работи като композицията на функции
- Низът, който връща `getLine` е “замърсен” от входно-изходна операция
- Как да композираме трансформации?

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`



## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a` а
- `<име> <- <трансформация>`

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- `<име> <- <трансформация>`
  - `<трансформация>` е от тип `IO a`

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a` а
- `<име> <- <трансформация>`
  - <трансформация> е от тип `IO a` а
  - резултатът от <трансформация> се свързва с <име>

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- `<име> <- <трансформация>`
  - <трансформация> е от тип `IO a`
  - резултатът от <трансформация> се свързва с <име>
- `return <израз>`

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- `<име> <- <трансформация>`
  - <трансформация> е от тип `IO a`
  - резултатът от <трансформация> се свързва с <име>
- `return <израз>`
  - празна трансформация, която връща <израз> като резултат

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- `<име> <- <трансформация>`
  - <трансформация> е от тип `IO a`
  - резултатът от <трансформация> се свързва с <име>
- `return <израз>`
  - празна трансформация, която връща <израз> като резултат
  - `return :: a -> IO a`

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- `<име> <- <трансформация>`
  - <трансформация> е от тип `IO a`
  - резултатът от <трансформация> се свързва с <име>
- `return <израз>`
  - празна трансформация, която връща <израз> като резултат
  - `return :: a -> IO a`
- резултатът от цялата конструкция `do` е резултатът от последната трансформация в композицията

## Конструкцията `do`

В Haskell има специален синтаксис за композиране на трансформации:

```
do { <трансформация> }
```

<трансформация> може да бъде:

- произволен израз от тип `IO a`
- `<име> <- <трансформация>`
  - <трансформация> е от тип `IO a`
  - резултатът от <трансформация> се свързва с <име>
- `return <израз>`
  - празна трансформация, която връща <израз> като резултат
  - `return :: a -> IO a`
- резултатът от цялата конструкция `do` е резултатът от последната трансформация в композицията

```
main = do line <- getLine
        putStrLn ("Въведохте: " ++ line)
```



## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a`

## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a`
- `return` фиктивно “замърсява” резултат от тип `a` за да стане от тип `IO a`

## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a`
- `return` фиктивно “замърсява” резултат от тип `a` за да стане от тип `IO a`
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?

## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a`
- `return` фиктивно “замърсява” резултат от тип `a` за да стане от тип `IO a`
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?
- Създава се локалната дефиниция `<име> = <израз>!`

## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a`
- `return` фиктивно “замърсява” резултат от тип `a` за да стане от тип `IO a`
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?
- Създава се локалната дефиниция `<име> = <израз>!`
- Понже се налага често, има специален синтаксис за това:  
`let <име> = <израз>`

## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a` а
- `return` фиктивно “замърсява” резултат от тип `a` за да стане от тип `IO a` а
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?
- Създава се локалната дефиниция `<име> = <израз>!`
- Понже се налага често, има специален синтаксис за това:  
`let <име> = <израз>`
- Да не се бърка с `let <име> = <израз> in <израз>!`

## Локални дефиниции в `do`

В някакъв смисъл `<-` и `return` са обратни една на друга операции:

- `<-` извлича “чист” резултат от тип `a` от трансформация от тип `IO a`
- `return` фиктивно “замърсява” резултат от тип `a` за да стане от тип `IO a`
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?
- Създава се локалната дефиниция `<име> = <израз>`!
- Понже се налага често, има специален синтаксис за това:  
`let <име> = <израз>`
- Да не се бърка с `let <име> = <израз> in <израз>`!

### Пример:

```
main = do putStrLn "Моля, въведете палиндром: "
         line <- getLine
         let revLine = reverse line
         if revLine == line then putStrLn "Благодаря!"
         else do putStrLn (line ++ " не е палиндром!")
         main
```

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?



## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStrLn . show`

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStr . show`
- `read :: (Read a) => String -> a`

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStr . show`
- `read :: (Read a) => String -> a`
- `read "1.23" → ?`

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStr . show`
- `read :: (Read a) => String -> a`
- `read "1.23" → Грешка!`

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStr . show`
- `read :: (Read a) => String -> a`
- `read "1.23" → Грешка!`
- Haskell не може да познае типа на резултата, понеже е генеричен!

## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStrLn . show`
- `read :: (Read a) => String -> a`
- `read "1.23" → Грешка!`
- Haskell не може да познае типа на резултата, понеже е генеричен!
- `getInt :: IO Int`



## Вход и изход на данни

Как можем да извеждаме и въвеждаме данни от типове различни от `Char` и `String`?

На помощ идват класовете `Show` и `Read`:

- `show :: (Show a) => a -> String`
- `print :: (Show a) => a -> IO ()`
- `print = putStr . show`
- `read :: (Read a) => String -> a`
- `read "1.23" → Грешка!`
- Haskell не може да познае типа на резултата, понеже е генеричен!
- `getInt :: IO Int`
- `getInt = do line <- getLine  
          return $ read line`

## Пример: средно аритметично на редица от числа

```
findAverage :: IO Double
```

## Пример: средно аритметично на редица от числа

```
findAverage :: IO Double
findAverage = do putStr "Моля, въведете брой: "
                 n <- getInt
                 s <- readAndSum n
                 return $ (fromIntegral s) / (fromIntegral n)
```

## Пример: средно аритметично на редица от числа

```
findAverage :: IO Double
findAverage = do putStr "Моля, въведете брой: "
                n <- getInt
                s <- readAndSum n
                return $ (fromIntegral s) / (fromIntegral n)

readAndSum :: Int -> IO Int
```

## Пример: средно аритметично на редица от числа

```

findAverage :: IO Double
findAverage = do putStr "Моля, въведете брой: "
                n <- getInt
                s <- readAndSum n
                return $ (fromIntegral s) / (fromIntegral n)

readAndSum :: Int -> IO Int
readAndSum 0 = return 0
readAndSum n = do putStr "Моля, въведете число: "
                  x <- getInt
                  s <- readAndSum (n - 1)
                  return $ x + s

```

## Пример: средно аритметично на редица от числа

```

findAverage :: IO Double
findAverage = do putStr "Моля, въведете брой: "
                n <- getInt
                s <- readAndSum n
                return $ (fromIntegral s) / (fromIntegral n)

readAndSum :: Int -> IO Int
readAndSum 0 = return 0
readAndSum n = do putStr "Моля, въведете число: "
                  x <- getInt
                  s <- readAndSum (n - 1)
                  return $ x + s

main = do avg <- findAverage
         putStrLn $ "Средното аритметично е: " ++ show avg

```

## Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`

## Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`



## Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък

## Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`

## Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`

## Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`



# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  - Също като `mapM`, но изхвърля резултата

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  - Също като `mapM`, но изхвърля резултата
  - `printList = mapM_ print`

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  - Също като `mapM`, но изхвърля резултата
  - `printList = mapM_ print`
- `forever :: IO a -> IO b`

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  - Също като `mapM`, но изхвърля резултата
  - `printList = mapM_ print`
- `forever :: IO a -> IO b`
  - безкрайна композиция на една и съща трансформация

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  - Също като `mapM`, но изхвърля резултата
  - `printList = mapM_ print`
- `forever :: IO a -> IO b`
  - безкрайна композиция на една и съща трансформация
  - както `repeat` за списъци

# Управляващи функции

Можем да работим с трансформации с функции от по-висок ред:

- `import Control.Monad`
- `sequence :: [IO a] -> IO [a]`
  - композира трансформации и събира резултатите им в списък
  - `getInts n = sequence (replicate n getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
  - композира списък от трансформации по списък от стойности
  - `mapM = sequence . map`
  - `printRead s = do putStr (s ++ " = "); getInt`
  - `readCoordinates = mapM printRead ["x", "y", "z"]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
  - Също като `mapM`, но изхвърля резултата
  - `printList = mapM_ print`
- `forever :: IO a -> IO b`
  - безкрайна композиция на една и съща трансформация
  - както `repeat` за списъци
  - `forever $ do line <- getLine; putStrLn line`

## Средно аритметично на числа v2.0

```

readInt :: String -> IO Int
readInt s = do putStr $ "Моля, въведете " ++ s ++ ": "
              getInt

findAverage :: IO Double
findAverage = do n <- readInt "брой"
                 l <- mapM (readInt . ("число #"++) . show) [1..n]
                 let s = sum l
                 return ((fromIntegral s) / (fromIntegral n))

main = forever $
      do avg <- findAverage
         putStrLn $ "Средното аритметично е: " ++ show avg
         putStrLn "Хайде отново!"

```