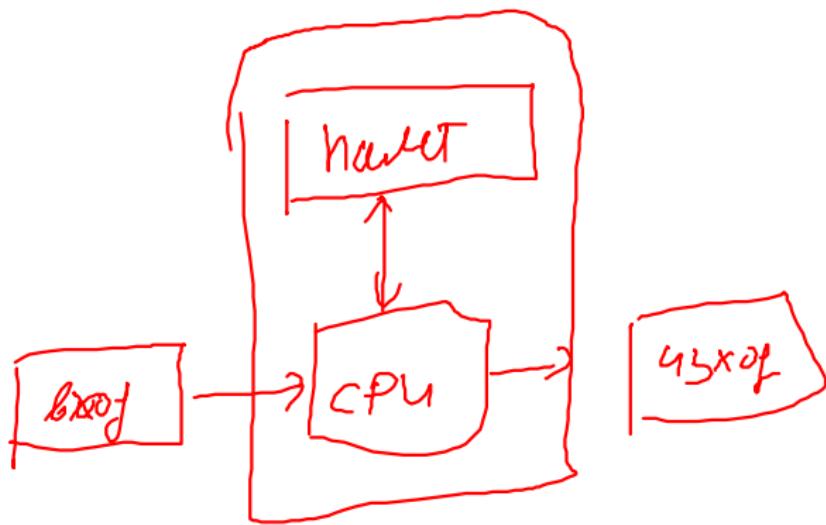


Входно-изходни операции в Haskell

Трифон Трифонов

Функционално програмиране, 2017/18 г.

18 януари 2018 г.



Страницни ефекти в Haskell

- Функциите в Haskell нямат страницни ефекти

Страницни ефекти в Haskell

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

Страницни ефекти в Haskell

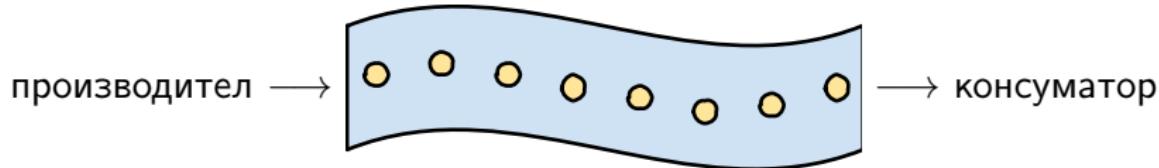
- Функциите в Haskell нямат страницни ефекти
- Но входно-изходните операции по природа са страницни ефекти!
- Как можем да се справим с този парадокс?

Страницни ефекти в Haskell

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

Страницни ефекти в Haskell

- Функциите в Haskell нямат страницни ефекти
- Но входно-изходните операции по природа са страницни 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` а задава тип на входно/изходна трансформация, резултатът от която е от тип а.

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

Типът IO a

Стандартният генеричен тип `IO` а 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 а

Стандартният генеричен тип `IO` а 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?`
- `Hello :: 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?`
- `Hello :: 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?`
- `Hello :: 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?`
- `Hello :: 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** а

Конструкцията do

В Haskell има специален синтаксис за композиране на трансформации:

do { <трансформация> }

<трансформация> може да бъде:

- произволен израз от тип **IO** 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** а
- <име> **<->** <трансформация>
 - <трансформация> е от тип **IO** а

Конструкцията do

В Haskell има специален синтаксис за композиране на трансформации:

do { <трансформация> }

<трансформация> може да бъде:

- произволен израз от тип **IO** а
- <име> **<->** <трансформация>
 - <трансформация> е от тип **IO** 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** а
- <име> **<->** <трансформация>
 - <трансформация> е от тип **IO** а
 - 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** а
- <име> **<-** <трансформация>
 - <трансформация> е от тип **IO** а
 - 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** а
- <име> **<-** <трансформация>
 - <трансформация> е от тип **IO** а
 - 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** а
- <име> **<-** <трансформация>
 - <трансформация> е от тип **IO** а
 - 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** а
- <име> **<-** <трансформация>
 - <трансформация> е от тип **IO** а
 - 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 а от трансформация от тип `IO` а
- `return` фиктивно “замърсява” резултат от тип а за да стане от тип `IO` а
- 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 а от трансформация от тип `IO` а
- `return` фиктивно “замърсява” резултат от тип а за да стане от тип `IO` а
- 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 а от трансформация от тип `IO` а
- `return` фиктивно “замърсява” резултат от тип а за да стане от тип `IO` а
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?
- Създава се локалната дефиниция `<име> = <израз>`!
- Понеже 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 а от трансформация от тип `IO` а
- `return` фиктивно "замърсява" резултат от тип а за да стане от тип `IO` а
- Какъв е ефектът от `<име> <- return <израз>` в `do` конструкция?
- Създава се локалната дефиниция `<име> = <израз>`!
- Понеже 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` фактивно “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>`!
- Понеже 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 = putStrLn . 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 = putStrLn . 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 = putStrLn . 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 = putStrLn . 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 = 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`
- `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 putStrLn "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 putStrLn "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 putStrLn "Моля, въведете брой: "
                 n <- getInt
                 s <- readAndSum n
                 return $ (fromIntegral s) / (fromIntegral n)

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

Пример: средно аритметично на редица от числа

```

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

readAndSum :: Int -> IO Int
readAndSum 0 = return 0
readAndSum n = do putStrLn "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`
 - `getInts = sequence . ('replicate' 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`
 - `getInts = sequence . (‘replicate’ 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`
 - `getInts = sequence . (‘replicate’ 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`
 - `getInts = sequence . ('replicate' 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`
 - `getInts = sequence . (‘replicate‘ getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn $ 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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`
 - `getInts = sequence . ('replicate' getInt)`
- `mapM :: (a -> IO b) -> [a] -> IO [b]`
 - композира списък от трансформации по списък от стойности
 - `mapM = sequence . map`
 - `printRead s = do putStrLn 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 putStrLn $ "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!"

```

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци
- `getContents :: IO String` — връща списък от **всички** символи на стандартния вход

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци
- `getContents :: IO String` — връща списък от **всички** символи на стандартния вход
- списъкът се оценява лениво, т.е. прочита се при нужда

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци
- `getContents :: IO String` — връща списък от **всички** символи на стандартния вход
- списъкът се оценява лениво, т.е. прочита се при нужда
- Пример:

```
noSpaces = do text <- getContents  
              putStrLn $ filter (/=' ') text
```

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци
- `getContents :: IO String` — връща списък от **всички** символи на стандартния вход
- списъкът се оценява лениво, т.е. прочита се при нужда
- Пример:

```
noSpaces = do text <- getContents  
              putStrLn $ filter (/=' ') text
```

- `interact :: (String -> String) -> IO ()` — лениво прилага функция над низове над стандартния вход и извежда резултата на стандартния изход

Ленив вход и изход

- Ленивото оценяване в Haskell ни позволява да работим с входно/изходни потоци
- `getContents :: IO String` — връща списък от **всички** символи на стандартния вход
- списъкът се оценява лениво, т.е. прочита се при нужда
- **Пример:**

```
noSpaces = do text <- getContents  
              putStrLn $ filter (/=' ') text
```

- `interact :: (String -> String) -> IO ()` — лениво прилага функция над низове над стандартния вход и извежда резултата на стандартния изход
- **Пример:**

```
noSpaces = interact $ filter (/=' ')
```

Работа с файлове

- **IO** позволява работа с произволни файлове, не само със стандартните вход и изход

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим
 - `type FilePath = String`

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим
 - `type FilePath = String`
 - `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим
 - `type FilePath = String`
 - `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
- Има варианти на функциите за вход/изход, които работят с `Handle`

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим
 - `type FilePath = String`
 - `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
- Има варианти на функциите за вход/изход, които работят с `Handle`
- `hGetLine, hGetChar, hPutStr, hPutStrLn, hGetContents...`

Работа с файлове

- `IO` позволява работа с произволни файлове, не само със стандартните вход и изход
- `import System.IO`
- `openFile :: FilePath -> IOMode -> IO Handle` — отваря файл със зададено име в зададен режим
 - `type FilePath = String`
 - `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
- Има варианти на функциите за вход/изход, които работят с `Handle`
- `hGetLine, hGetChar, hPutStr, hPutStrLn, hGetContents...`
- **Пример:**

```
encrypt cypher inFile outFile =
    do h1 <- openFile inFile ReadMode
       text <- hGetContents h1
       h2 <- openFile outFile WriteMode
       hPutStr h2 $ map cypher text
```

Монади

- `IO` е пример за монада

Монади

- `IO` е пример за монада
- Монадите са конструкции, които “опаковат” обекти от даден тип

Монади

- `IO` е пример за монада
- Монадите са конструкции, които “опаковат” обекти от даден тип
- Примери:

Монади

- `IO` е пример за монада
- Монадите са конструкции, които “опаковат” обекти от даден тип
- Примери:
 - `IO` опакова стойност във входно/изходна трансформация

Монади

- `IO` е пример за монада
- Монадите са конструкции, които “опаковат” обекти от даден тип
- Примери:
 - `IO` опакова стойност във входно/изходна трансформация
 - `Maybe` опакова стойност с “флаг” дали стойността съществува

Монади

- `IO` е пример за **монада**
- Монадите са конструкции, които “опаковат” обекти от даден тип
- **Примери:**
 - `IO` опакова стойност във входно/изходна трансформация
 - `Maybe` опакова стойност с “флаг” дали стойността съществува
 - `[a]` опакова няколко “алтернативни” стойности в едно

Монади

- `IO` е пример за монада
- Монадите са конструкции, които “опаковат” обекти от даден тип
- Примери:
 - `IO` опакова стойност във входно/изходна трансформация
 - `Maybe` опакова стойност с “флаг” дали стойността съществува
 - `[a]` опакова няколко “алтернативни” стойности в едно
 - `r -> a` опакова стойност от тип `a` в “машинка”, която я пресмята при подаден параметър от тип `x`

Монади

- `IO` е пример за монада
- Монадите са конструкции, които “опаковат” обекти от даден тип
- Примери:
 - `IO` опакова стойност във входно/изходна трансформация
 - `Maybe` опакова стойност с “флаг” дали стойността съществува
 - `[a]` опакова няколко “алтернативни” стойности в едно
 - `r -> a` опакова стойност от тип `a` в “машинка”, която я пресмята при подаден параметър от тип `r`
 - `s -> (a,s)` опакова стойност от тип `a` в “действие”, което променя дадено състояние от тип `s`

Монадни операции

- Monad е клас от **типови конструктори**, които са монади

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!
- ($>>=$) :: Monad m => m a -> (a -> m b) -> m b

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!
- $(>>=) :: \text{Monad } m \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b$
- оператор за “свързване” на опаковани стойности

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!
- $(>>=) :: \text{Monad } m \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b$
- оператор за “свързване” на опаковани стойности
- $b = a >>= f$:

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!
- $(>>=) :: \text{Monad } m \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b$
- оператор за “свързване” на опаковани стойности
- $b = a >>= f$:
 - поглеждаме стойността x в опаковката a

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!
- $(>>=) :: \text{Monad } m \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b$
- оператор за “свързване” на опаковани стойности
- $b = a >>= f$:
 - поглеждаме стойността x в опаковката a
 - прилагаме функцията f над x

Монадни операции

- Monad е клас от **типови конструктори**, които са монади
- “Опаковката” често е **еднопосочна**: един път опакована, не можем да извадим стойността извън опаковката...
- ... но можем да я преопаковаме!
- $(>>=) :: \text{Monad } m \Rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b$
- оператор за “свързване” на опаковани стойности
- $b = a >>= f$:
 - поглеждаме стойността x в опаковката a
 - прилагаме функцията f над x
 - и получаваме нова опакована стойност b

Императивен стил чрез монади

- `do` всъщност е синтактична захар за поредица от “свързвания”

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

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

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

```
main = getLine >>= (\line -> putStrLn $ "Въведохте: " ++ line)
```

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

```
main = getLine >>= putStrLn . ("Въведохте: " ++)
```

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от "свързвания"
- **Примери:**

```
main = getLine >>= putStrLn . ("Въведохте: " ++)
```

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

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

```
main = getLine >>= putStrLn . ("Въведохте: " ++)
```

```
findAverage = putStrLn "Моля, въведете брой: " >>=
    (\_ -> getInt >>=
     (\n -> readAndSum n >>=
      (\s -> return $ (fromIntegral s) /
                     (fromIntegral n))))
```

Императивен стил чрез монади

- `do` всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

```
main = getLine >>= putStrLn . ("Въведохте: " ++)
```

```
findAverage = putStrLn "Моля, въведете брой: " >>=
    (\_ -> getInt >>=
     (\n -> readAndSum n >>=
      (\s -> return $ (fromIntegral s) /
                     (fromIntegral n))))
```

- работи за произволни монади, не само за `IO`!

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

```
main = getLine >>= putStrLn . ("Въведохте: " ++)
```

```
findAverage = putStrLn "Моля, въведете брой: " >>=
    (\_ -> getInt >>=
     (\n -> readAndSum n >>=
      (\s -> return $ (fromIntegral s) /
          (fromIntegral n))))
```

- работи за произволни монади, не само за **IO!**
- позволява абстракхиране от страничните ефекти и моделиране на поредица от инструкции

Императивен стил чрез монади

- **do** всъщност е синтактична захар за поредица от “свързвания”
- **Примери:**

```
main = getLine >>= putStrLn . ("Въведохте: " ++)
```

```
findAverage = putStrLn "Моля, въведете брой: " >>=
    (\_ -> getInt >>=
     (\n -> readAndSum n >>=
      (\s -> return $ (fromIntegral s) /
          (fromIntegral n))))
```

- работи за произволни монади, не само за **IO!**
- позволява абстракционе от страничните ефекти и моделиране на поредица от инструкции
- императивен стил във функционалното програмиране