

Въведение

ИТ Кариера



Учителски екип
Обучение за ИТ кариера

<https://it-kariera.mon.bg/e-learning>

Съдържание

- Парадигми за програмиране
- Функционални езици
- Входно/изходни операции
- Състояние на програма



Парадигми за програмиране

- Обектно-ориентирана парадигма
 - Следва императивен програмен модел
 - Променливи и обекти (изменяеми данни)
 - Функциите и стойностите са различни концепции
 - Странични ефекти при изпълнение (изпълнението води до промени в състоянието)

Парадигми за програмиране (...)

- Функционална парадигма
 - Следва декларативен програмен модел
 - Функциите са стойности
 - Стойностите не се променят по време на изпълнение на програмата (неизменяеми данни)
 - Липсва концепцията за състояние
 - Висока ефективност на изпълнение
 - Отложено изпълнение на код
 - По-малко възможности за грешки

Функционални езици

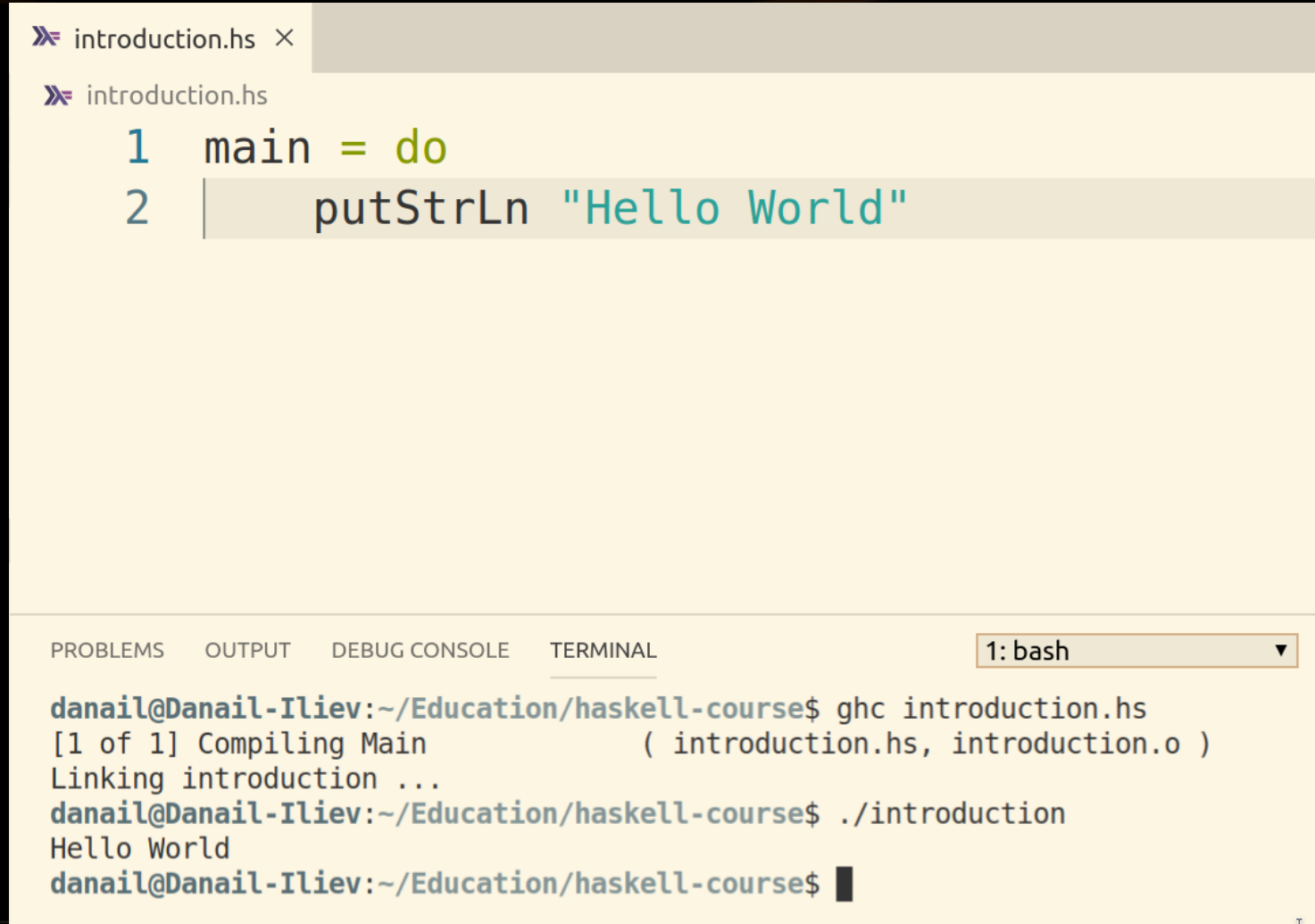
- Чисто функционални езици
 - Haskell
 - Mercury
 - Clean
- Нечисти функционални езици
 - Lisp
 - Scala
 - Clojure
 - F#

Haskell

- Чисто функционален език
- Статично типизиран
- Бързодействие
- Отложено изпълнение
- Инструменти
 - Платформата Haskell
 - GHCi (Read, Evaluate, Print Loop)
 - VSCode with Haskell Syntax Highlighting plugin

Hello World

- Работа с конзолата



The screenshot shows an IDE window with a tab for 'introduction.hs'. The code in the editor is:

```
1 main = do
2   putStrLn "Hello World"
```

Below the editor is a terminal window with tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active, showing the following commands and output:

```
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main                ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
Hello World
danail@Danail-Iliev:~/Education/haskell-course$
```

Входно/изходни операции

- Всяка функция в Haskell си има тип - от какъв тип са функциите, които взаимодействат с външния свят?
- Входно/изходните операции в Haskell стават посредством IO действия
- Най-просто обяснено IO действията са парчета код, които взаимодействат с външния свят
- ``main`` сам по себе си е IO действие - това означава, че реално се изпълнява и всички странични ефекти реално се случват
- `do`-блоковете се използват, за да се опишат няколко последователни действия, които IO действието да изпълни

Входно/изходни операции



The image shows a Haskell IDE window with a file named `introduction.hs`. The code in the editor is as follows:

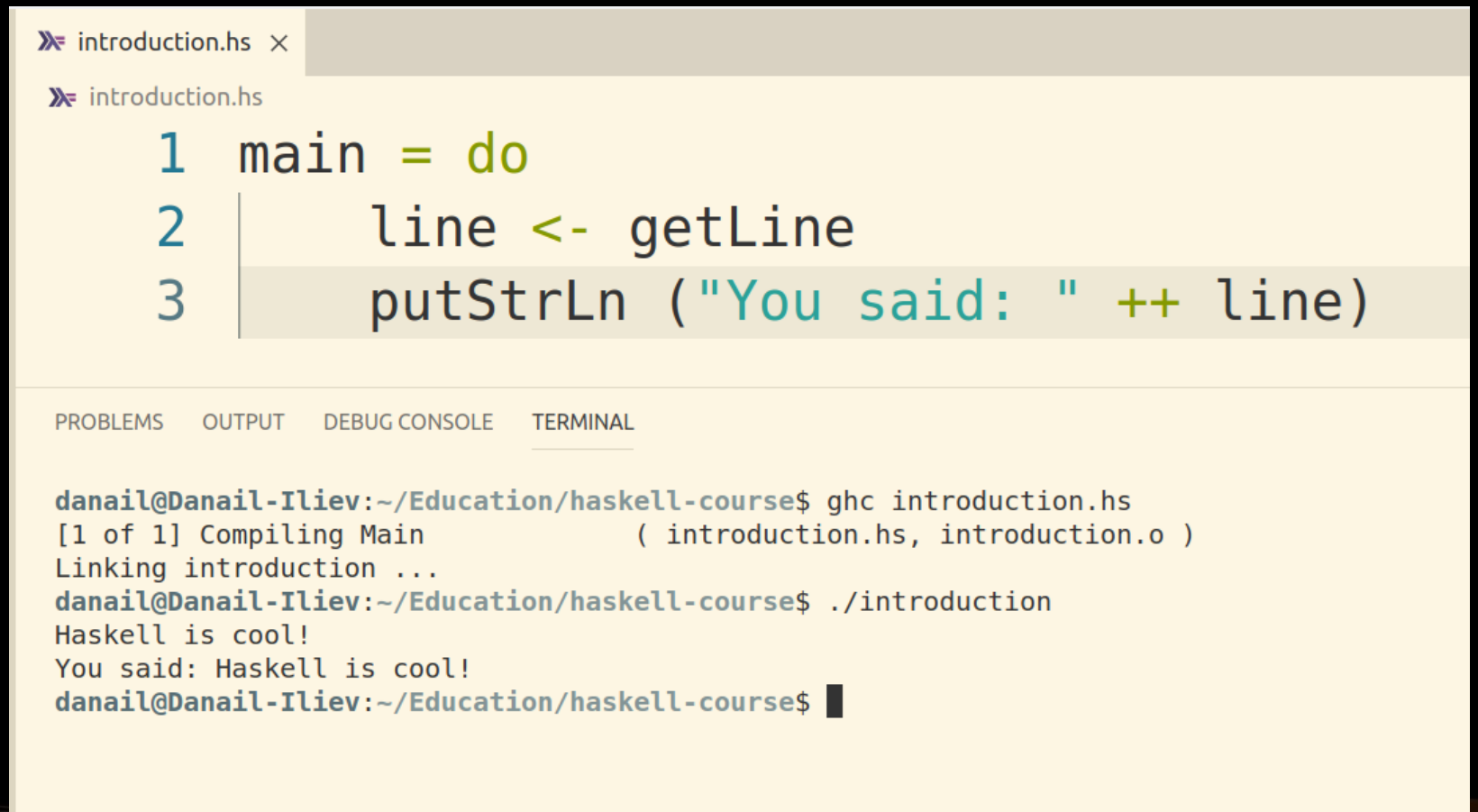
```
1 main = do
2     putStrLn "Hello,"
3     putStrLn "World"
4     putStrLn "!"
```

Below the editor is a terminal window. The terminal shows the compilation and execution of the program:

```
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main          ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
Hello,
World
!
danail@Danail-Iliev:~/Education/haskell-course$
```

Входно/изходни операции

- Четене на стойност от конзолата



```
introduction.hs x
introduction.hs
1 main = do
2     line <- getLine
3     putStrLn ("You said: " ++ line)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main                ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
Haskell is cool!
You said: Haskell is cool!
danail@Danail-Iliev:~/Education/haskell-course$
```

Входно/изходни операции

- Четене на стойност от конзолата

Операторът за присвояване `<-` може да се използва само в рамките на `do`-блок, както и присвоената променлива може да се използва само в `do`-блока

```
introduction.hs x
introduction.hs

1 main = do
2     line <- getLine
3     putStrLn ("You said: " ++ line)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main                ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
Haskell is cool!
You said: Haskell is cool!
danail@Danail-Iliev:~/Education/haskell-course$
```

Входно/изходни операции

- Функцията `return` е функция, която приема стойност и създава IO действие, който при извикване не прави нищо, а веднага връща тази стойност

```
dummyGetLine :: IO String  
dummyGetLine =  
    return "I'm not really doing anything"
```

Входно/исходни операции

introduction.hs x

introduction.hs

```
1 dummyGetLine :: IO String
2 dummyGetLine =
3     return "I'm not really doing anything"
4
5 main :: IO ()
6 main = do
7     line <- dummyGetLine
8     putStrLn line
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main             ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
I'm not really doing anything
danail@Danail-Iliev:~/Education/haskell-course$
```


Входно/изходни операции

- Някои полезни IO действия:

```
putStrLn :: String -> IO ()
```

- Принтира символен низ на конзолата, след което добавя нов ред

```
getLine :: IO String
```

- Чете ред от конзолата

Входно/изходни операции

- Принтира стойност, представена като символен низ, на конзолата

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

```
readFile :: FilePath -> IO String
```

- Чете цял файл като “мързелив” символен низ

Входно/изходни операции

- Пише символен низ във файл

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

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

- Добавя символен низ на края на файл

Входно/изходни операции

- Примери:

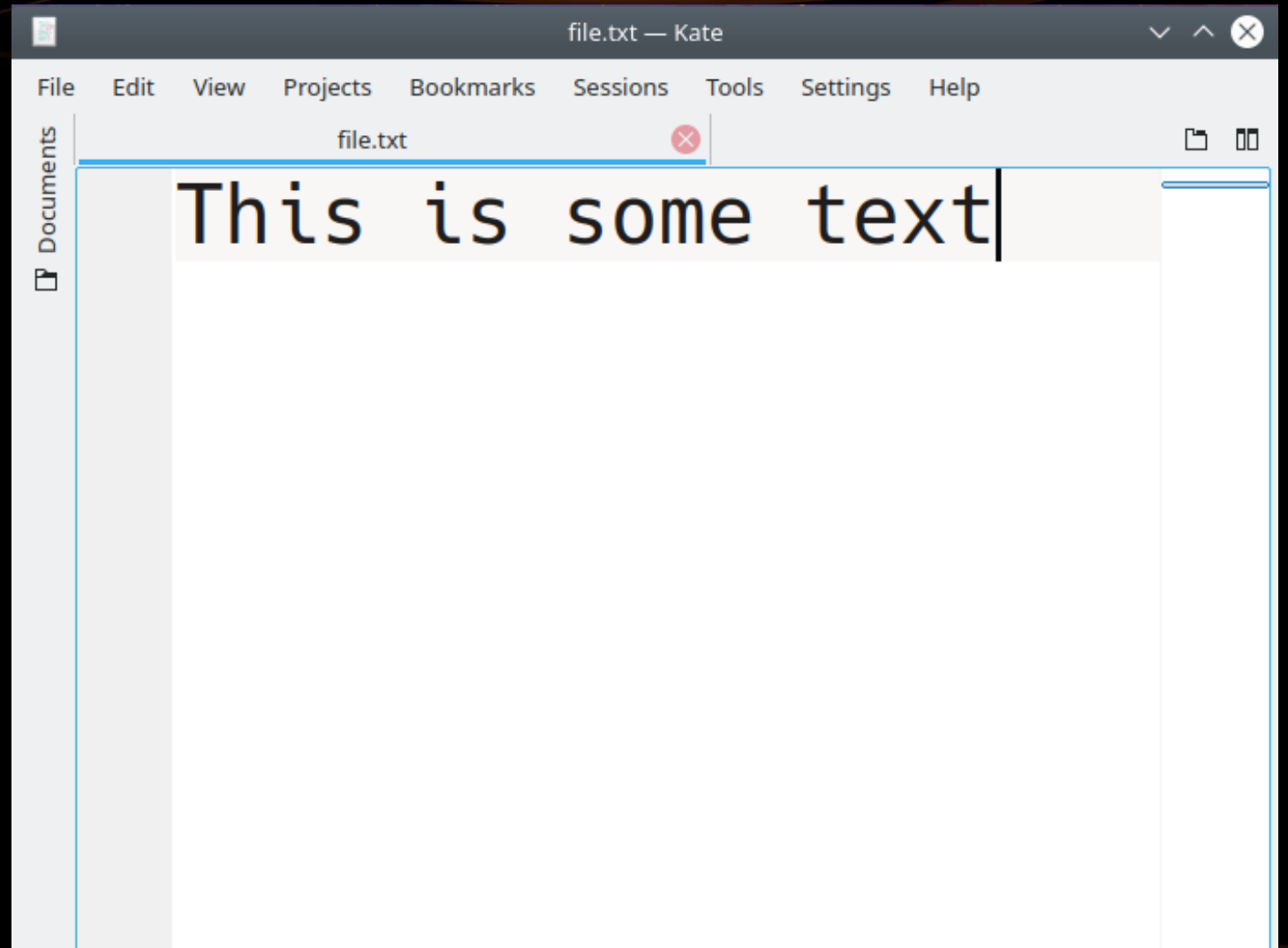
```
introduction.hs ×
introduction.hs
1 desktopFilePath = "/home/danail/Desktop/file.txt"
2
3 main :: IO ()
4 main = do
5     writeFile desktopFilePath "This is some text"

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main             ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
danail@Danail-Iliev:~/Education/haskell-course$
```

Входно/изходни операции

- Създава нов файл или пренаписва съдържанието на вече съществуващ такъв



Входно/изходни операции

- Четене на файл като “мързелив” символен низ

```
introduction.hs X
introduction.hs
1 desktopFilePath = "/home/danail/Desktop/file.txt"
2
3 main :: IO ()
4 main = do
5     file <- readFile desktopFilePath
6     putStrLn file
7

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main          ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
This is some text
danail@Danail-Iliev:~/Education/haskell-course$
```

Входно/изходни операции

- Добавяне на текст в края на файл

```
introduction.hs x
introduction.hs
1 desktopFilePath = "/home/danail/Desktop/file.txt"
2
3 main :: IO ()
4 main = do
5     appendFile desktopFilePath "This is appended text"
6     file <- readFile desktopFilePath
7     putStrLn file
8

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
danail@Danail-Iliev:~/Education/haskell-course$ ghc introduction.hs
[1 of 1] Compiling Main          ( introduction.hs, introduction.o )
Linking introduction ...
danail@Danail-Iliev:~/Education/haskell-course$ ./introduction
This is some textThis is appended text
danail@Danail-Iliev:~/Education/haskell-course$
```

Състояние на програма

- Казваме, че една система/програма има състояние, когато е създадена да помни потребителски интеракции или предхождащи евенти
- Състоянието обикновено се пази в променливи, които представляват заделена компютърна памет
- Глобално състояние на програма
 - Достъпно от всеки контекст на програмата
- Локално състояние на програма
 - Достъпно само в рамките на определена функция/package

Състояние на програма

- Глобално състояние в Haskell
- Не се препоръчва използването му

```
import Data.IORef
import System.IO.Unsafe

globalVariable :: IORef Int
-- {-# NOINLINE myGlobalVar #-}
globalVariable = unsafePerformIO (newIORef 17)
```

Състояние на програма

- Monad
 - Стратегия за комбиниране на изчисления/операции в по-сложни такива
 - Всеки Monad представя
 - return функция
 - Комбинаторна функция bind
 - Типизиран конструктор
- State Monad
 - Може да бъде използван за да се симулира състояние на програма в Haskell

Състояние на програма

```
import Control.Monad.State
```

```
myState :: State (Double, Double) Double
```

```
get :: State s s
```

```
put :: s -> State s ()
```

```
evalState :: State s a -> s -> a
```

Състояние на програма

Добавяне на
нужната библиотека

```
import Control.Monad.State
```

```
myState :: State (Double, Double) Double
```

```
get :: State s s
```

```
put :: s -> State s ()
```

```
evalState :: State s a -> s -> a
```

Състояние на програма

```
import Control.Monad.State
```

```
myState :: State (Double, Double) Double
```

```
get :: State s s
```

Деклариране на променлива,
която да пази състоянието

```
put :: s -> State s ()
```

```
evalState :: State s a -> s -> a
```

Състояние на програма

```
import Control.Monad.State
```

```
myState :: State (Double, Double) Double
```

```
get :: State s s
```

С функцията get се
извлича състоянието

```
put :: s -> State s ()
```

```
evalState :: State s a -> s -> a
```

Състояние на програма

```
import Control.Monad.State
```

```
myState :: State (Double, Double) Double
```

```
get :: State s s
```

Функцията put поставя
стойност в променливата,
която пази състоянието

```
put :: s -> State s ()
```

```
evalState :: State s a -> s -> a
```


Състояние на програма

```
import Control.Monad.State
```

```
myState :: State (Double, Double) Double
```

```
get :: State s s
```

```
put :: s -> State s ()
```

Функцията evalState връща
крайния резултат от
състоянието на програмата

```
evalState :: State s a -> s -> a
```

Синтаксис

- Дефиниране на променлива

```
getFive = 5
```

- Дефиниране на променлива (в GHCi)

```
let getFive = 5
```

- Бележка: let се използва и при дефиниране на променлива в тялото на функция

Синтаксис

- Дефиниране на променлива (функция)

```
getFive = 5
```

- Дефиниране на променлива (в GHCi)

```
let getFive = 5
```

- Бележка: let се използва и при дефиниране на променлива в тялото на функция

Синтаксис

- Дефиниране на променлива

Оператор за
присвояване

```
getFive = 5
```

- Дефиниране на променлива (в GHCi)

```
let getFive = 5
```

- Бележка: let се използва и при дефиниране на променлива в тялото на функция

Синтаксис

- Дефиниране на променлива

литерал

```
getFive = 5
```

- Дефиниране на променлива (в GHCi)

```
let getFive = 5
```

- Бележка: let се използва и при дефиниране на променлива в тялото на функция

Синтаксис

- Дефиниране на променлива

```
getFive = 5
```

Не е нужно поставянето
на ; на края на реда

- Дефиниране на променлива (в GHCi)

```
let getFive = 5
```

- Бележка: let се използва и при дефиниране на променлива в тялото на функция

Синтаксис

- Дефиниране на променлива

```
getFive = 5
```

- Дефиниране на променлива (в GHCi)

```
let getFive = 5
```

- Бележка: `let` се използва и при дефиниране на променлива в тялото на функция

В контекста на GHCi променливите се декларират с ключовата дума `let`

Синтаксис

- Оператори за сравнение:
 - $<$ - по-малко
 - $>$ - по-голямо
 - $<=$ - по-малко или равно
 - $>=$ - по-голямо или равно
 - $==$ - равно на
 - \neq - различно от
- Всеки от тези оператори връща булева стойност

Синтаксис

- Логически оператори:
 - `||` - логическо или
 - `&&` - логическо и
 - `not(BooleanExpression)` - обръща стойността на булевия израз
 - `.|.` - логическо или (битова операция)
 - `.&.` - логическо и (битова операция)
- Всеки от тези оператори връща булева стойност

Синтаксис

- Оператори за математически операции:
 - + - събиране
 - — - изваждане
 - * - умножение
 - / - деление
 - sqrt - корен квадратен
 - abs - абсолютна стойност

Синтаксис

- Условни оператори:
 - if-else
 - guards
 - case

Синтаксис

- Условни оператори (if-else)

```
simpleFunction a =  
  if a == 5  
  then "It's five :)"  
  else if a == 6  
        then "It's six :)"  
        else "It's neither 5 nor 6 :("
```

Синтаксис

- Условни оператори (if-else)

```
simpleFunction a =  
  if a == 5  
  then "It's five :)"  
  else if a == 6  
        then "It's six :)"  
        else "It's neither 5 nor 6 :("
```

Булева променлива или
израз, връщащ булев
резултат

Синтаксис

- Условни оператори (if-else)

```
simpleFunction a =  
  if a == 5  
  then "It's five :)"  
  else if a == 6  
        then "It's six :)"  
        else "It's neither 5 nor 6 :("
```

Изпълнява се само в
случай, че булевия
израз връща True

Синтаксис

- Условни оператори (if-else)

```
simpleFunction a =  
  if a == 5  
  then "It's five :)"  
  else if a == 6  
        then "It's six :)"  
        else "It's neither 5 nor 6 :("
```

Вложено условие

Синтаксис

- Условни оператори (if-else)

```
simpleFunction a =  
  if a == 5  
  then "It's five :)"  
  else if a == 6  
        then "It's six :)"  
        else "It's neither 5 nor 6 :("
```

Изпълнява се, ако нито
едно от условията не е
удовлетворено

Синтаксис

- Условни оператори (guards)

```
simpleFunction' a
  | a == 5 = "It's five :)"
  | a == 6 = "It's six :)"
  | otherwise = "It's neither 5 nor 6 :("
```

- Оператор подобен на switch-case използван в други езици

Синтаксис

- Условни оператори

Условията се дефинират с оператора `|` - връща се резултат отговарящ на условието

```
simpleFunction a
| a == 5 = "It's five :)"
| a == 6 = "It's six :)"
| otherwise = "It's neither 5 nor 6 :("
```

- Оператор подобен на switch-case използван в други езици

Синтаксис

- Условни оператори (guards)

```
simpleFunction' a  
  | a == 5 = "It's five :)"  
  | a == 6 = "It's six :)"  
  | otherwise = "It's neither 5 nor 6 :("
```

Ако нито едно условие не е
удовлетворено се връща резултата
след ключовата дума otherwise

- Оператор подобен на switch-case използван в други
езици

Синтаксис

- Условни оператори (case)

```
simpleFunction' a = case a of  
    5 -> "It's five :)"  
    6 -> "It's six :)"  
    _  -> "It's neither 5 nor 6 :("
```

- Условният оператор започва с `case <име на параметъра> of`
- От лявата страна на оператора `->` е условието, което трябва да е удовлетворено, а от дясната резултата, който се връща, ако това се случи
- `_` - хваща всички други случаи, които не са описани

Обобщение

- Парадигми за програмиране
- Входно/изходни операции
- Състояние на програма



Министерство на образованието и науката (МОН)

- Настоящият курс (презентации, примери, задачи, упражнения и др.) е разработен за нуждите на Национална програма "**Обучение за ИТ кариера**" на МОН за подготовка по професия "Приложен програмист"



Министерство
на образованието
и науката



Национална
програма
„Обучение за
ИТ кариера“

- Курсът се разпространява под свободен лиценз **CC-BY-NC-SA**

