Лекция 9. Ввод-вывод в Haskell. Чистый код и эффекты

Функциональное и логическое программирование (осень 2025)

Николай Кудасов

20 октября 2025

институт ispring

Содержание

Основные примитивы ввода-вывода

Простейшая программа

```haskell
module Main where

main :: IO ()
main = putStrLn "Привет, мир!"
```

1. `IO a` — тип программ с операциями ввода-вывода и возвращающих значения типа `a`
2. Функция `putStrLn` имеет тип `String -> IO ()`
3. Функция `putStrLn` — *чистая*!
4. Функция `putStrln` принимает строку типа `String` и возвращает *программу* типа `IO ()`, которая в свою очередь вычисляет значение типа `()`.

Простейшая программа

```haskell
module Main where

main :: IO ()
main = putStrLn "Привет, мир!"
```

1. `IO a` — тип программ с операциями ввода-вывода и возвращающих значения типа `a`
2. Функция `putStrLn` имеет тип `String -> IO ()`
3. Функция `putStrLn` — *чистая*!
4. Функция `putStrln` принимает строку типа `String` и возвращает *программу* типа `IO ()`, которая в свою очередь вычисляет значение типа `()`.

Простейшая программа

```haskell
module Main where

main :: IO ()
main = putStrLn "Привет, мир!"
```

1. `IO a` — тип программ с операциями ввода-вывода и возвращающих значения типа `a`
2. Функция `putStrLn` имеет тип `String -> IO ()`
3. Функция `putStrLn` — *чистая*!
4. Функция `putStrln` принимает строку типа `String` и возвращает *программу* типа `IO ()`, которая в свою очередь вычисляет значение типа `()`.

Простейшая программа

```haskell
module Main where

main :: IO ()
main = putStrLn "Привет, мир!"
```

1. `IO a` — тип программ с операциями ввода-вывода и возвращающих значения типа `a`
2. Функция `putStrLn` имеет тип `String -> IO ()`
3. Функция `putStrLn` — *чистая*!
4. Функция `putStrln` принимает строку типа `String` и возвращает *программу* типа `IO ()`, которая в свою очередь вычисляет значение типа `()`.

Простейшая программа

```haskell
module Main where

main :: IO ()
main = putStrLn "Привет, мир!"
```

1. `IO a` — тип программ с операциями ввода-вывода и возвращающих значения типа `a`
2. Функция `putStrLn` имеет тип `String -> IO ()`
3. Функция `putStrLn` — *чистая*!
4. Функция `putStrln` принимает строку типа `String` и возвращает *программу* типа `IO ()`, которая в свою очередь вычисляет значение типа `()`.

Простейшая программа

```
1 module Main where
2
3 main :: IO ()
4 main = putStrLn "Привет, мир!"
```

1. `IO a` — тип программ с операциями ввода-вывода и возвращающих значения типа `a`
2. Функция `putStrLn` имеет тип `String -> IO ()`
3. Функция `putStrLn` — *чистая*!
4. Функция `putStrln` принимает строку типа `String` и возвращает *программу* типа `IO ()`, которая в свою очередь вычисляет значение типа `()`.

do-нотация (последовательное выполнение)

```haskell
main :: IO ()
main = do
  putStrLn "Привет!"
  putStrLn "Пока!"
```

1. do-нотация в Haskell — синтаксический сахар для перегружаемого императивного программирования.
2. Мы будем использовать do-нотацию как данное.
3. Значение последней программы возвращается из всего блока.
4. Соответственно, тип последней программы совпадает с типом всего блока.

do-нотация (последовательное выполнение)

```haskell
main :: IO ()
main = do
  putStrLn "Привет!"
  putStrLn "Пока!"
```

1. do-нотация в Haskell — синтаксический сахар для перегружаемого императивного программирования.
2. Мы будем использовать do-нотацию как данное.
3. Значение последней программы возвращается из всего блока.
4. Соответственно, тип последней программы совпадает с типом всего блока.

do-нотация (последовательное выполнение)

```haskell
main :: IO ()
main = do
  putStrLn "Привет!"
  putStrLn "Пока!"
```

1. do-нотация в Haskell — синтаксический сахар для перегружаемого императивного программирования.
2. Мы будем использовать do-нотацию как данное.
3. Значение последней программы возвращается из всего блока.
4. Соответственно, тип последней программы совпадает с типом всего блока.

do-нотация (последовательное выполнение)

```haskell
main :: IO ()
main = do
  putStrLn "Привет!"
  putStrLn "Пока!"
```

1. do-нотация в Haskell — синтаксический сахар для перегружаемого императивного программирования.
2. Мы будем использовать do-нотацию как данное.
3. Значение последней программы возвращается из всего блока.
4. Соответственно, тип последней программы совпадает с типом всего блока.

do-нотация (последовательное выполнение)

```haskell
main :: IO ()
main = do
  putStrLn "Привет!"
  putStrLn "Пока!"
```

1. do-нотация в Haskell — синтаксический сахар для перегружаемого императивного программирования.
2. Мы будем использовать do-нотацию как данное.
3. Значение последней программы возвращается из всего блока.
4. Соответственно, тип последней программы совпадает с типом всего блока.

do-нотация (связывание)

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

1. Стрелка (`<-`) *связывает* результат программы (`getLine`) с переменной (`name`).
2. Переменная `name` может быть использована в последующих строках в блоке.
3. `getLine :: IO String` — программа, возвращающая строку
4. `name :: String` — результат программы
5. `name` — **неизменяемая** переменная (как и обычно)

do-нотация (связывание)

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

1. Стрелка (`<-`) *связывает* результат программы (`getLine`) с переменной (`name`).
2. Переменная `name` может быть использована в последующих строках в блоке.
3. `getLine :: IO String` — программа, возвращающая строку
4. `name :: String` — результат программы
5. `name` — **неизменяемая** переменная (как и обычно)

do-нотация (связывание)

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

1. Стрелка (`<-`) *связывает* результат программы (`getLine`) с переменной (`name`).
2. Переменная `name` может быть использована в последующих строках в блоке.
3. `getLine :: IO String` — программа, возвращающая строку
4. `name :: String` — результат программы
5. `name` — **неизменяемая** переменная (как и обычно)

do-нотация (связывание)

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

1. Стрелка (`<-`) *связывает* результат программы (`getLine`) с переменной (`name`).
2. Переменная `name` может быть использована в последующих строках в блоке.
3. `getLine :: IO String` — программа, возвращающая строку
4. `name :: String` — результат программы
5. `name` — **неизменяемая** переменная (как и обычно)

do-нотация (связывание)

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

1. Стрелка (`<-`) *связывает* результат программы (`getLine`) с переменной (`name`).
2. Переменная `name` может быть использована в последующих строках в блоке.
3. `getLine :: IO String` — программа, возвращающая строку
4. `name :: String` — результат программы
5. `name` — **неизменяемая** переменная (как и обычно)

do-нотация (связывание)

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

1. Стрелка (`<-`) *связывает* результат программы (`getLine`) с переменной (`name`).
2. Переменная `name` может быть использована в последующих строках в блоке.
3. `getLine :: IO String` — программа, возвращающая строку
4. `name :: String` — результат программы
5. `name` — **неизменяемая** переменная (как и обычно)

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Полезные стандартные операции ввода-вывода

- `putStr :: String -> IO ()`
- `putStrLn :: String -> IO ()`
- `print :: Show a => a -> IO ()`
- `getLine :: IO String`
- `getContents :: IO String`
- `readFile :: FilePath -> IO String`
- `writeFile :: FilePath -> String -> IO ()`

См. больше операций в модуле `System.IO`.

Интактивные программы на Haskell

План

1. Реализовать простое интерактивное приложение.
2. Добавить работу с состоянием (память программы).
3. Добавить разбор и обработку команд пользователя.
4. Разделить ввод-вывод и чистую логику приложения.
5. По возможности, обобщить полученный результат.

Вечный цикл

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
```

Как превратить эту программу в вечный цикл (аналогично `while (True)`)?

Вечная рекурсия

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
  main
```

Используем рекурсию!

Как можно выйти из цикла?

Вечная рекурсия

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  name <- getLine
  putStrLn ("Привет, " ++ name ++ "!")
  main
```

Используем рекурсию!

Как можно выйти из цикла?

Прерывание рекурсии

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  input <- getLine
  handle input
  where
    handle "пока" = putStrLn "Пока!"
    handle name = do
      putStrLn ("Привет, " ++ name ++ "!")
      main
```

Выход может быть осуществлён любым условным оператором, например:

- Функция с образцами и/или охранными выражениями.
- Выражение `if ... then ... else ...`.
- Выражение `case ... of ...`.

Прерываение рекурсии с case-выражением

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  input <- getLine
  case input of
    "пока" -> putStrLn "Пока!"
    name -> do
      putStrLn ("Привет, " ++ name ++ "!")
      main
```

Сейчас каждая интерация независима. Как добавить память (состояние) в рекурсивную программу?

Прерываение рекурсии с case-выражением

```haskell
main :: IO ()
main = do
  putStrLn "Как тебя зовут?"
  input <- getLine
  case input of
    "пока" -> putStrLn "Пока!"
    name -> do
      putStrLn ("Привет, " ++ name ++ "!")
      main
```

Сейчас каждая интерация независима. Как добавить память (состояние) в рекурсивную программу?

Трекер задач

```haskell
newtype Item = Item { getItem :: String }
newtype TodoState = TodoState { getTodoState :: [Item] }

todoApp :: TodoState -> IO ()
todoApp (TodoState items) = do
  putStrLn "Введите запрос:"
  input <- getLine
  case input of
    "выход" -> putStrLn "Пока!"
    itemTitle -> do
      putStrLn ("Добавлена задача: " ++ itemTitle ++ "!")
      todoApp (TodoState (Item itemTitle : items))

main = todoApp (TodoState [])
```

Список дел

```haskell
newtype Item = Item { getItem :: String }
newtype TodoState = TodoState { getTodoState :: [Item] }
```

Реализуйте рекурсивную функцию, печатающую список дел:

```haskell
printItems :: [Item] -> IO ()
```

Список дел (рекурсивная функция)

```haskell
printItems :: [Item] -> IO ()
printItems [] = putStrLn "Нет дел (всё сделано)."
printItems [item] = putStrLn (getItem item)
printItems (item : items) = do
  putStrLn (getItem item)
  printItems items
```

Это допустимая реализация, но почему бы использовать `map`?

Список дел (рекурсивная функция)

```haskell
printItems :: [Item] -> IO ()
printItems [] = putStrLn "Нет дел (всё сделано)."
printItems [item] = putStrLn (getItem item)
printItems (item : items) = do
  putStrLn (getItem item)
  printItems items
```

Это допустимая реализация, но почему бы использовать `map`?

Список дел (через `map`)

```haskell
printItems :: [Item] -> IO ()
printItems [] = putStrLn "Нет дел (всё сделано)."
printItems items = map (putStrLn . getItem) items
```

```
Expected type: IO ()
 Actual type: [IO ()]
```

Список дел (через `map`)

```haskell
printItems :: [Item] -> IO ()
printItems [] = putStrLn "Нет дел (всё сделано)."
printItems items = map (putStrLn . getItem) items
```

```
Expected type: IO ()
 Actual type: [IO ()]
```

Список дел (через `map` и `sequence_`)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs

return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Список дел (через `map` и `sequence_`)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs

return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Список дел (через `map` и `sequence_`)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs

return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Список дел (через `map` и `sequence_`)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs
```

```haskell
return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Список дел (через `map` и `sequence_`)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs
```

```haskell
return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Список дел (через map и sequence_)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs

return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Список дел (через map и sequence_)

```haskell
printTasks :: [Task] -> IO ()
printTasks [] = putStrLn "Нет дел (всё сделано)."
printTasks tasks = sequence_ (map (putStrLn . getItem) tasks)
```

```haskell
sequence_ :: [IO ()] -> IO ()
sequence_ [] = return ()
sequence_ (program : programs) = do
  program
  sequence_ programs

return :: a -> IO a -- просто вернуть значение, без ввода-вывода
```

Функция `return`

```haskell
return :: a -> IO a
```

Функция `return` **не влияет** на поток управления!

```haskell
example :: IO ()
example = do
  x <- getLine
  return ()               -- ничего не происходит
  n <- return (length x) -- связывает (length x) с n
  let m = length x        -- эквивалентно строчке выше
  putStrLn ("n = " ++ show n)
  putStrLn ("m = " ++ show m)
```

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM  :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_  :: (a -> IO b) -> [a] -> IO ()`
- `sequence  :: [IO a] -> IO [a]`
- `sequence_  :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM  :: [a] -> (a -> IO b) -> IO [b]`
- `forM_  :: [a] -> (a -> IO b) -> IO ()`
- `void  :: IO a -> IO ()`
- `when  :: Bool -> IO () -> IO ()`
- `liftM  :: (a -> b) -> IO a -> IO b`
- `ap  :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Полезные комбинаторы для ввода-вывода (и не только)

Вариации `map` и `sequence` (специализированные для `IO` и списков):

- `mapM :: (a -> IO b) -> [a] -> IO [b]`
- `mapM_ :: (a -> IO b) -> [a] -> IO ()`
- `sequence :: [IO a] -> IO [a]`
- `sequence_ :: [IO a] -> IO ()`

Дополнительные комбинаторы из `Control.Monad` (для `IO` и списков):

- `forM :: [a] -> (a -> IO b) -> IO [b]`
- `forM_ :: [a] -> (a -> IO b) -> IO ()`
- `void :: IO a -> IO ()`
- `when :: Bool -> IO () -> IO ()`
- `liftM :: (a -> b) -> IO a -> IO b`
- `ap :: IO (a -> b) -> IO a -> IO b`

Some useful IO combinators (generalized)

Вариации `map` и `sequence` (в общем виде):

- `mapM  :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)`
- `mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()`
- `sequence  :: (Traversable t, Monad m) => t (m a) -> m (t a)`
- `sequence_ :: (Foldable t, Monad m) => t (m a) -> m ()`

Дополнительные комбинаторы из `Control.Monad` (в общем виде):

- `forM  :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)`
- `forM_ :: (Foldable t, Monad m) => t a -> (a -> m b) -> m ()`
- `void   :: Functor f => f a -> f ()`
- `when   :: Applicative f => Bool -> f () -> f ()`
- `liftM  :: Applicative f => (a -> b) -> f a -> f b`
- `ap     :: Monad m => m (a -> b) -> m a -> m b`

Разбор и обработка запросов

Тип запросов

Начнём с алгебраического типа для возможных запросов:

```haskell
data Request
  = Exit              -- Выход.
  | ListItems         -- Перечислить дела.
  | AddItem Item      -- Добавить дело.
  | RemoveItem Int    -- Выполнить дело.
```

По сути, этот тип представляет собой интерфейс для приложения:

1. Каждый конструктор значений (запрос) аналогичен *методу* приложения.
2. Заметьте, что запросы могут иметь параметры.

Разбор запросов

```haskell
import Text.Read (readMaybe)

parseRequest :: String -> Maybe Request
parseRequest input = case words input of
  ["выход"] -> Just Exit
  ["дела"] -> Just ListItems
  ["сделано", strIndex] -> case readMaybe strIndex of
    Nothing -> Nothing
    Just n -> Just (RemoveItem n)
  _ -> Just (AddItem (Item input))
```

1. Разбор может завершиться неудачей (почему?), поэтому мы используем `Maybe`.
2. Функция `words` разбивает строку по пробельным символам.
3. Функция `readMaybe` пытается распознать данные в строке (в данном случае, данные типа `Int`).

Разбор с классом типов Read

```haskell
import Text.Read (readMaybe)

read      :: Read a => String -> a
readMaybe :: Read a => String -> Maybe a
```

1. `Read` — это класс типов, значения которых можно распознать в строке.
2. `Read a =>` — это ограничение на тип `a` в сигнатуре функции.
3. `read` принимает один аргумент (строку типа `String`) и возвращает значение типа `a`.
4. `read` **небезопасная функция** (почему?) и не должна быть использована в 99% случаев!
5. `readMaybe` — безопасная версия `read`, возвращающая `Nothing`, если разбор не удался.

Разбор с классом типов Read

```haskell
import Text.Read (readMaybe)

read      :: Read a => String -> a
readMaybe :: Read a => String -> Maybe a
```

1. `Read` — это класс типов, значения которых можно распознать в строке.
2. `Read a =>` — это ограничение на тип `a` в сигнатуре функции.
3. `read` принимает один аргумент (строку типа `String`) и возвращает значение типа `a`.
4. `read` **небезопасная функция** (почему?) и не должна быть использована в 99% случаев!
5. `readMaybe` — безопасная версия `read`, возвращающая `Nothing`, если разбор не удался.

Разбор с классом типов Read

```haskell
import Text.Read (readMaybe)

read      :: Read a => String -> a
readMaybe :: Read a => String -> Maybe a
```

1. `Read` — это класс типов, значения которых можно распознать в строке.
2. `Read a =>` — это ограничение на тип `a` в сигнатуре функции.
3. `read` принимает один аргумент (строку типа `String`) и возвращает значение типа `a`.
4. `read` **небезопасная функция** (почему?) и не должна быть использована в 99% случаев!
5. `readMaybe` — безопасная версия `read`, возвращающая `Nothing`, если разбор не удался.

Разбор с классом типов Read

```haskell
import Text.Read (readMaybe)

read      :: Read a => String -> a
readMaybe :: Read a => String -> Maybe a
```

1. `Read` — это класс типов, значения которых можно распознать в строке.
2. `Read a =>` — это ограничение на тип `a` в сигнатуре функции.
3. `read` принимает один аргумент (строку типа `String`) и возвращает значение типа `a`.
4. `read` **небезопасная функция** (почему?) и не должна быть использована в 99% случаев!
5. `readMaybe` — безопасная версия `read`, возвращающая `Nothing`, если разбор не удался.

Разбор с классом типов Read

```haskell
import Text.Read (readMaybe)

read      :: Read a => String -> a
readMaybe :: Read a => String -> Maybe a
```

1. `Read` — это класс типов, значения которых можно распознать в строке.
2. `Read a =>` — это ограничение на тип `a` в сигнатуре функции.
3. `read` принимает один аргумент (строку типа `String`) и возвращает значение типа `a`.
4. `read` **небезопасная функция** (почему?) и не должна быть использована в 99% случаев!
5. `readMaybe` — безопасная версия `read`, возвращающая `Nothing`, если разбор не удался.

Разбор с классом типов Read

```haskell
import Text.Read (readMaybe)

read      :: Read a => String -> a
readMaybe :: Read a => String -> Maybe a
```

1. `Read` — это класс типов, значения которых можно распознать в строке.
2. `Read a =>` — это ограничение на тип `a` в сигнатуре функции.
3. `read` принимает один аргумент (строку типа `String`) и возвращает значение типа `a`.
4. `read` **небезопасная функция** (почему?) и не должна быть использована в 99% случаев!
5. `readMaybe` — безопасная версия `read`, возвращающая `Nothing`, если разбор не удался.

Использование разбора в приложении

```haskell
todoApp state@(TodoState items) = do
  putStrLn "Введите запрос:"
  input <- getLine
  case parseRequest input of
    Nothing -> do
      putStrLn "Не получилось понять ваш запрос :("
      todoApp state                          -- рекурсия!
    Just Exit -> putStrLn "Пока!"
    Just ListItems -> do
      printTasks state
      todoApp state                          -- рекурсия!
    Just (AddItem item) -> do
      putStrLn ("Добавлено дело: " ++ getItem item)
      todoApp (TodoState (item : items))  -- рекурсия!
    Just (RemoveItem i) -> do
      case deleteAt i items of
        Just (Item name, items') -> do
          putStrLn ("Дело выполнено: " ++ name)
          todoApp (TodoState items')      -- рекурсия!
        Nothing -> do
          putStrLn ("Дела номер " ++ show i ++ " не существует")
          todoApp state                      -- рекурсия!
```

Обработка запросов

Теперь мы выделим *чистые* обработчики запросов, чтобы избежать “загрязнения” `IO` повсюду.

```haskell
data Response a = Response String (Maybe a)
type Handler = TodoState -> Response TodoState

handleRequest :: Request -> Handler
```

1. `Handler` — это обработчик запроса, который
   - может обновить состояние приложения
   - может завершить приложение (для этого используем `Maybe`)
   - **должен** предоставить какой-то вывод для пользователя
2. Каждому запросу соответствует обработчик.
3. Диспетчеризация запросов происходит в функциии `handleRequest`.

Обработчик для `ListItems`

Обработчик для печати списка дел не требует изменения состояния.

Комбинатор `readonly` позволяет (удобнее) определять такие обработчики.

```
readonly :: (TodoState -> String) -> Handler
readonly f state = Response (f state) (Just state)

handleListItems :: Handler
handleListItems = readonly (prettyItems . getTodoState)
  where
    prettyItems [] = "Нет дел (всё сделано)."
    prettyItems tasks = unlines (map getItem tasks)
```

Обработчик для **AddItem**

```
1  handleAddItem :: Item -> Handler
2  handleAddItem item state
3    = Response response (Just newState)
4    where
5      response = "Добавлено дело: " ++ getItem item
6      TodoState items = state
7      newState = TodoState (item : items)
```

Обработчик для `RemoveItem`

```haskell
handleRemoveItem :: Int -> Handler
handleRemoveItem i state =
  case deleteAt i items of
    Just (removedItem, items') ->
      Response (responseSuccess removedItem) (Just (TodoState items'))
    Nothing -> Response responseFailure (Just state)
  where
    TodoState items = state
    responseSuccess (Item name) = "Дело выполнено: " ++ name
    responseFailure = "Дела номер " ++ show i ++ " не существует"
```

Обработчик для `Exit`

```haskell
handleExit :: Handler
handleExit _state = Response "Пока!" Nothing
```

Диспетчер запросов

Осталось сопоставить каждый запрос своему обработчику:

```
handleRequest :: Request -> Handler
handleRequest request =
  case request of
    Exit          -> handleExit
    ListItems     -> handleListItems
    AddItem item  -> handleAddItem item
    RemoveItem i  -> handleRemoveItem i
```

Общая картина

```haskell
todoApp :: TodoState -> IO ()
todoApp state = do
  putStrLn "Введите запрос:"
  input <- getLine
  case parseRequest input of
    Nothing -> do
      putStrLn "Не получилось понять ваш запрос :("
      todoApp state
    Just request ->
      case handleRequest request state of
        Response response nextState -> do
          putStrLn response
          case nextState of
            Nothing -> return ()
            Just newState -> todoApp newState
```

Обобщение

Параметризация по разбору и обработке запросов

```
1  todoApp
2    :: (String -> Maybe Request)
3    -> (Request -> Handler)
4    -> TodoState
5    -> IO ()
6  todoApp parseRequest handleRequest state = do
7    ...
```

Мы можем вынести функции разбора и обработки запросов в параметры, не меняя тела функции `todoApp`.

Это, в частности, позволяет использовать разные функции при тестировании.

Параметризация по типу запросов

```
1  todoApp
2    :: (String -> Maybe request)
3    -> (request -> Handler)
4    -> TodoState
5    -> IO ()
6  todoApp parseRequest handleRequest state = do
7    ...
```

Мы можем также параметризоваться по **типу** запросов.

Действительно, тело `todoApp` больше никак не зависит от конкретного типа запросов!

Параметризация по типу состояния

```
type Handler state = state -> Response state

simpleCommandLineApp
  :: (String -> Maybe request)
  -> (request -> Handler state)
  -> state
  -> IO ()
simpleCommandLineApp parseRequest handleRequest state = do
  ...
```

Аналогично мы можем параметризоваться по **типу** состояния приложения.

Теперь мы уже определяем не трекер задач, а произвольное (простое) консольное приложение!

Спасибо за внимание!