Тема 15

Програмиране на входно-изходни операции в Haskell

Основни принципи

Наличието на входно-изходни операции (на входни операции) в една програма може да доведе до нарушаване на строго функционалния стил.

Пример. Оценката на обръщението към функцията

```
funny :: Int -> Int
funny n = inputInt + n
```

зависи от резултата от изпълнението от входната операция, осъществена от inputInt.

Четене на стойности

Операцията, при която се прочита един ред с текст от стандартното входно устройство (стандартния входен поток) и се връща резултат от тип String, съвпадащ с прочетения ред, се осъществява с помощта на вградената функция

```
getLine :: IO String
```

Аналогично функцията

```
getchar :: IO Char
```

предизвиква прочитане на отделен (следващия) знак от входния поток.

Едноелементният тип ()

В Haskell е дефиниран типът (), който съдържа само един елемент. Този елемент се записва като ().

Стойността от тип () не носи информация и затова този тип се използва рядко. Той обаче е полезен при програмирането на входно-изходните операции, от които не се очаква върната стойност. Съответните програми имат тип

```
IO ()
```

и връщат стойност ().

Извеждане на символни низове

Основната операция по извеждане на символен низ се реализира от функцията

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

С нейна помощ може например да бъде дефинирана функция, която извежда даден низ без заграждащите го кавички:

```
helloworld :: IO ()
helloworld = putStr "Hello, World!"
```

С използване на putStr може да бъде дефинирана функция, която предизвиква извеждане на даден низ на отделен ред (т.е. извеждане на цял ред в изходния поток):

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

Извеждане на стойности в общия случай

В стандартната прелюдия на Haskell е дефиниран класът Show с функция от сигнатурата си

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

Тази функция може да бъде използвана за извеждане на стойности от много типове. Например с нейна помощ може да бъде дефинирана функцията за отпечатване на стойност

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

Връщане на стойност: return

Ако е необходимо наред с изпълнението на дадена входно-изходна операция да бъде върната някаква стойност, това може да се осъществи с помощта на вградената функция

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

Ефектът от **`return x`** е връщане на резултат **`x`**.

Конструкцията do

Конструкцията do предоставя гъвкав механизъм за формиране на последователности от входно-изходни операции и свързване на променливи със стойности – резултати от извършване на входно-изходни операции с цел предаване на тези стойности за по-нататъшна обработка.

Пример 1. Функция, която предизвиква последователно извеждане на даден низ и newline.

```
putStrLn :: String -> IO ()
putStrLn str = do putStr str
                  putStr "\n"
```

Пример 2. Функция, която предизвиква 4-кратно извеждане на даден низ.

```
put4times :: String -> IO ()
put4times str
   = do putStrLn str
        putStrLn str
        putStrLn str
        putStrLn str
```

Пример 3. Функция, която предизвиква n-кратно извеждане на даден низ.

```
putNtimes :: Int -> String -> IO ()
putNtimes n str
   = if n <= 1
     then putStrLn str
     else do putStrLn str
             putNtimes (n-1) str

put4times = putNtimes 4
```

Пример 4. Функция, която предизвиква прочитане на два последователни реда от входния поток.

```
read2lines :: IO ()
read2lines
   = do getLine
        getLine
        putStrLn "Two lines read."
```

Свързване на променливи (имена) с прочетени стойности

Пример 1. Именуване на резултата от входна операция.

```
getNput :: IO ()
getNput = do line <- getLine
             putStrLn line
```

Пример 2. Прочитане на два реда и извеждане на съдържанието им в обратен и обърнат ред.

```
reverse2lines :: IO ()
reverse2lines
   = do line1 <- getLine
        line2 <- getLine
        putStrLn (reverse line2)
        putStrLn (reverse line1)
```

Локални дефиниции в израз do

Конструкцията **`var <- getLine`** именува резултата от изпълнението на getLine и по такъв начин действа като дефиниция. Възможно е също да бъдат включени локални дефиниции в израз do.

Пример: нова дефиниция на функцията reverse2lines.

```
reverse2lines :: IO ()
reverse2lines
   = do linel <- getLine
        line2 <- getLine
        let revl = reverse linel
        let rev2 = reverse line2
        putStrLn rev2
        putStrLn revl
```

Четене на стойности в общия случай

В стандартната прелюдия на Haskell е дефиниран класът Read с функция от сигнатурата си

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

Тази функция извършва синтактичен анализ на символен низ, представящ стойност от определен тип, и връща „прочетената“ от низа стойност.

Пример 1. Прочитане на цяло число от един ред от стандартния входен поток.

```
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)
```

Пример 2. Намиране на сумата на редица от цели числа.

```
sumInts :: IO Int
sumInts
   = do n <- getInt
        if n==0
           then return 0
           else (do m <- sumInts
                    return (n+m))
```

Пример 3. Интерактивна програма за намиране на сумата на редица от цели числа.

```
sumInteract :: IO ()
sumInteract
   = do putStrLn "Enter integers one per line"
        putStrLn "These will be summed until zero
                  is entered"
        sum <- sumInts
        putStr "The sum was "
        print sum
```