

Для того, чтобы наш код на Haskell был полноценным кодом, он должен быть организован надлежащим образом и мы должны уметь грамотно делать «ввод-вывод». И то, и другое подразумевает работу с монадами, конкретно, с монадой IO.

Однако, в качестве вводной нашего урока, попробуем достичь заявленной цели, избегая определения монады.

Структура «реальной программы»

Такая программа должна иметь точку входа, именованную «main», так же как и в других Си-подобных языках. Предполагаем, что ее тип должен быть **IO()**. Вот классический пример:

```
main = print "Hello, world!"
```

Для многострочных программ нам необходимо использовать ключевое слово **do**, полный смысл которого будет ясен также при изучении монад.

Вот как осуществляется получение аргументов командной строки в самом простом случае:

```
import System.Environment(getArgs)
main = do
    s <- getArgs
    print s
```

При запуске

```
runghc fisrt.hs hello you! best 123 4
```

мы получим следующий вывод:

```
["hello","you!","best","123","4"]
```

Для более серьезных случаев разбора опций и значений командной строки используются специальные библиотеки:

[GetOpt](#)

[wiki.haskell: GetOpt](#) и

[wiki.haskell: High-level option handling with GetOpt](#) (пример использования)

[ReadArgs](#) (несложная и понятная библиотека, которая позволяет передавать простые значения),

[Список всех парсеров командной строки](#)

Вот как мы получим аргументы с помощью библиотеки [ReadArgs](#):

```
{-# LANGUAGE ScopedTypeVariables #-}

import ReadArgs

main = do
    (x1 :: Integer, x2 :: Integer) <- readArgs
    putStrLn $ "Sum = " ++ show (x1 + x2)
```

Прагма

```
{-# LANGUAGE ScopedTypeVariables #-}
```

необходима для правильной работы программы. Запуск с передачей параметров можно сделать так:

```
runghc sample.hs 12 32  
Sum = 44
```

Если тип или число параметров неподходящее, то будет сообщение об ошибке вроде такого:

```
runghc sample.hs 12  
usage: sample.hs Integer Integer
```

Библиотеку надо устанавливать самостоятельно:

```
cabal install ReadArgs
```

Рассмотрим более сложный случай, когда нам необходимо получить данные из консоли:

```
main = do c <- getChar  
         print (c == 'y')
```

(здесь мы ожидаем, что пользователь введет `y` или что-либо другое — соответственно программа выведет `True` или `False`).

Во втором примере мы видим, что некоторым аналогом присваивания выступает конструкция `c <- getChar`.

В качестве функции для вывода в простейшем случае используем `print`, которая сама использует функцию `show` для преобразования разных величин в строку. Но при использовании не латинских букв, выведутся номера символов:

```
main = print "Привет мир!"
```

получим:

```
"\1055\1088\1080\1074\1077\1090 \1084\1080\1088!"
```

Поэтому рекомендуется использовать функции `putStr` или `putStrLn`, но с ними придется делать самим преобразование величин в символы.

```
x = 7::Int  
main = putStrLn $ "Привет мир:" ++ (show x)
```

UPD: 2019-11-14:

Или корректного вывода кириллицы в консоли, добавить в файл (сам файл исходника должен быть в utf8) такие волшебные строки:

```
{-# LANGUAGE FlexibleInstances #-}  
  
instance {-# OVERLAPPING #-} Show String where  
    show x = ["'"] ++ x ++ ["'"]
```

вот такой вот будет рабочий пример с «Hello,...»:

```
{-# LANGUAGE FlexibleInstances #-}

instance {-# OVERLAPPING #-} Show String where
    show x = ['"'] ++ x ++ ['"']

main = print "Привет мир!"

C:\code\hello>runghc hello.hs
"Привет мир!"
```

[Haskell IO with non English characters](#)

Взаимодействие с STDIN-STDOUT

Список базовых функций ввода-вывода может быть найден тут:

[Basic Input and output](#)

или на русском языке тут:

[Haskell-98. Основные операции ввода-вывода](#)

Нам важны такие функции для работы со стандартными устройствами ввода-вывода из **Prelude**: **putStr**, **print**, **getLine**, **readLn**.

Рассмотрим простой пример считывания строки и числа из устройства стандартного ввода. Как результат, просто вернем полученные данные на стандартный вывод.

```
import System.IO (hFlush, stdout)
main = do
    putStr "Enter a string: "
    hFlush stdout
    str <- getLine
    putStr "Enter an integer: "
    hFlush stdout
    num <- readLn :: IO Int
    putStrLn $ str ++ " " ++ (show num)
```

(пример взят из [rosettacode: User input](#))

Здесь важно отметить, что во-первых, мы должны «помочь» команде **readLn** правильно считать строку и вернуть ее как число (по умолчанию, она останется строкой). Для этого мы указываем тип **:: IO Int** (просто **Int** — не можем). Во-вторых, мы должны самостоятельно делать сброс буфера — в документации указывается, что Haskell сам этого делать не будет.

Следующий, почти аналогичный пример взят из официального сборника [Haskell Report](#)

```
import System.IO
main = do
    hSetBuffering stdout NoBuffering
    putStr "Введите целое число: "
    x1 <- readNum
    putStr "Введите другое целое число: "
```

```

    x2 <- readNum
    putStr ("Их сумма равна " ++ show (x1+x2) ++ "\n")
  where readNum :: IO Integer
        readNum = readLn

```

Видим, что иначе решена проблема буферизации (отключена), и тоже решаем проблему ввода целых чисел (Указание сигнатуры типа позволяет избежать исправления типов `x1, x2` правилом по умолчанию).

Еще пример программы, которая меняет регистр вводимых букв:

```

import System.IO (hFlush, stdout)
import Char (toUpper)
main = do
  putStr "Enter a string: "
  hFlush stdout
  str <- getLine
  let str2 = map toUpper str
  putStrLn str2

```

Отметим, что если функция не из монады ввода-вывода (т.е. не возвращает подходящее значение), то мы используем конструкцию `let y = f x`, вместо конструкции `y <- f x`. Но в любом случае, переменные, однажды определенные, будут далее неизменны. Для работы с изменяемыми переменными нам необходимо монаду `IO` обернуть монадой `StateT`, что является нетривиальной задачей (будет в след. лекциях пример).

Первый проект

На лекции 3 мы разрабатывали функцию для решения квадратного уравнения. Теперь используем её в небольшой интерактивной программе, которая спросит у нас значения коэффициентов, а потом вычислит корни и выведет их.

Представим сначала в виде отдельной программы:

```

import System.IO

roots a b c =
  let d = b^2 - 4*a*c
      sd = sqrt d
      x1 = (-b - sd) / (2*a)
      x2 = (-b + sd) / (2*a)
  in (x1, x2)

main = do
  hSetBuffering stdout NoBuffering
  putStr "Enter an a: "
  a <- readLn :: IO Double
  putStr "Enter an b: "
  b <- readLn :: IO Double
  putStr "Enter an c: "
  c <- readLn :: IO Double
  putStrLn $ "Answer is: " ++ (show $ roots a b c)

```

```
let r = show $ roots a b c
print $ "Another method to do the same:" ++ r
```

Сохраняем ее как `first.hs` и либо запускаем на исполнение как скрипт:

```
runghc first.hs
```

Либо компилируем в исполняемый файл:

```
ghc first.hs
```

Теперь, оформим в виде самостоятельного модуля часть, решающую квадратное уравнение:

```
module Roots(roots) where
```

```
roots a b c =
  let d = b^2 - 4*a*c
      sd = sqrt d
      x1 = (-b - sd) / (2*a)
      x2 = (-b + sd) / (2*a)
  in (x1,x2)
```

сохраним в файле `Roots.hs`. Головная часть теперь будет импортировать наш модуль:

```
import Roots
import System.IO
```

```
main = do
  hSetBuffering stdout NoBuffering
  putStr "Enter an a: "
  a <- readLn :: IO Double
  putStr "Enter an b: "
  b <- readLn :: IO Double
  putStr "Enter an c: "
  c <- readLn :: IO Double
  putStrLn $ "Answer is: " ++ (show $ roots a b c)
  let r = show $ roots a b c
  print $ "Another method to do the same:" ++ r
```

Далее компилируем наши файлы:

```
ghc --make resolvroots.hs
```

указываем только головной файл, файл модуля `Roots.hs` будет скомпилирован и слинкован автоматически.

Задание к уроку. Подготовить проект с вычислением собственной функции (суммы квадратов двух чисел, факториала числа и т.п.), при этом сделать два варианта ввода данных: интерактивно и с помощью списка аргумента командной строки.

Вот возможный вариант решения с инкрементом аргумента

```
import System.Environment(getArgs)
main = do
```

```
[s] <- getArgs
let n = (read s::Int) +1
print n
```

Запуск и компиляция

Запуск подготовленных указанным выше способом файлов возможен разными путями. Во-первых, мы можем по-прежнему из `ghci` загружать нужный файл и указывать на выполнение функцию `main`.

Во-вторых возможен запуск на прямое исполнение командой `runghc myfile.hs`. В Linux запуск скрипта можно сделать с указанием «шебанга» в первой строке:

```
#!/usr/bin/runghc
```

В-третьих, компиляция в исполняемый двоичный код:

```
ghc --make myfile.hs
```

Можно с указанием опций для оптимизации:

```
ghc --make -O2 myfile.hs
```

У нас получится исполняемые файл `myfile` или `myfile.exe`, в зависимости от используемой операционной системы.

Полученный довольно объемный файл можно порой существенно ужать утилитой (она есть в поставке Haskell Platform для Windows или установлена отдельно в Linux).

```
strip -s myfile
```

Флаг `--make` (компиляция с наличием зависимых модулей) для компиляции в нашей простой ситуации совершенно необязателен, по факту, обычно это происходит автоматически.

«Кабализация» проекта

В будущем, для удобства работы с большими проектами, удобно использовать систему сборки библиотек и программ [cabal](#) (в мире Haskell также существует конкурентная система [stack](#)).

Например, для нашего проекта с решением квадратного уравнения можно в директории с исходными файлами (пусть это `RootsProj`) создать файл `RootsProj.cabal` с таким минимальным содержанием:

```
cabal-version:      >=2.0
name:               RootsProj
version:            0.1.0.0
author:             Vladimir V.
build-type:         Simple

library
  exposed-modules:   Roots
  build-depends:     base ^>=4.12.0.0
```

```
default-language:      Haskell2010

executable resolvroots
  main-is:              resolvroots.hs
  build-depends:        base ^>=4.12.0.0
  default-language:     Haskell2010
```

Для сборки проекта будем использовать команду `cabal build`, а для запуска скомпилированного приложения `cabal run resolvroots`. Подготовка готового пакета с исходниками для распространения и дальнейшего помещения в публичный репозиторий делается командой `cabal sdist`.

Создание конфигурационного файла проекта можно провести в интерактивном режиме, запустив в директории проекта `cabal init` и ответив на ряд вопросов.

[Cabal User Guide](#)

[The Haskell Cabal | Overview](#)

[The Haskell Tool Stack](#)

[Sam Halliday. Why Not Both?](#)

[Д.Шевченко. Прощай, cabal. Здравствуй, stack!](#)

Императивное программирование на Haskell. Шаг-1

Итак, сделаем первый шаг по императивному программированию на Haskell. Он начинается тут:

```
main = do
```

Хорошим тоном будет писать с указанием сигнатуры:

```
main :: IO ()
main = do
```

То, что находится ниже этой строки, представляет собой особый императивный язык внутри монады **IO**, который эмулируется внутри *чистого* функционального языка Haskell. Нам проще пока считать, что это «язык в языке». То, что находится выше, как правило (но не обязательно всегда) — чистые функции Haskell.

Далее, то что мы ранее называли выше в этой лекции *функциями*, следовало бы называть *действиями* или *акциями* (actions), чтобы их отличать от «чистых функций». И действительно, действия вроде `getLine`

```
getLine :: IO String
```

функциями в понимании Haskell (и тем более, в математическом понимании) не являются. Ведь они в разный момент в программе могут вернуть разное значение и иметь различные побочные эффекты! Кроме того, в отличие от чистых функций, действия не могут просто так появляться в других местах программного кода (только если он не «помечен», например, типом **IO**), и они, как операторы в императивном программировании, зависят от порядка применения!

Если действие возвращает какое-то значение, обернутое тэгом **IO**, например **IO String** как **getLine**, то мы прямо на месте можем извлечь это значение из-под тэга:

```
str <- getLine
```

и если посмотрим в `ghci`:

```
Prelude> :t str
str :: String
```

Другими словами, мы с большой натяжкой могли бы считать это как получение результата в переменную из действия. Только напомним себе, что `str` это как бы «одноразовая» переменная.

Если же мы хотим получить результат обычной чистой функции (говоря точнее, определить чистую функцию без аргументов или с аргументами) внутри этого мира, то используем усечённый вариант **let**:

```
main = do
  let x = 2 * 3
  print x
```

Если нужно указать тип такого результата (т.е., сигнатуру чистой функции)

```
main = do
  let
    x :: Int
    x = 2 * 3
  print x
```

или

```
main = do
  let x :: Int; x = 2 * 3
  print x
```

или даже

```
main = do
  let x = 2 * 3 :: Int
  print x
```

Кроме того, мы можем использовать ветвление **if-then-else** для оценки полученных чистых значений:

```
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  putStr "Type a number:"
  x <- readLn :: IO Int
  if x == 0
    then putStrLn "x is zero!"
    else putStrLn $ "x = " ++ show x
```

И можем использовать рекурсивные определения:


```

import System.IO

main = do
  hSetBuffering stdout NoBuffering
  putStr "Type a number:"
  k <- readLn :: IO Int

  if k < 0
  then putStrLn $ (show k) ++ " is negative!"
  else loop k
    where
      loop 0 = do putStrLn "That is all!"
      loop n = do
        putStrLn $ "n = " ++ show n
        loop (n-1)

```

Вот такой будет результат вывода:

```

runghc recursion.hs
Type a number:4
n = 4
n = 3
n = 2
n = 1
That is all!

```

И существуют даже аналоги циклов `for` (если нам не нужна эффективность, но нужна наглядность в императивном стиле):

```

import System.IO
import Control.Monad

main = do
  hSetBuffering stdout NoBuffering
  putStr "Type a number:"
  k <- readLn :: IO Int

  if k <= 0
  then putStrLn $ (show k) ++ " is not positive!"
  else
    forM_ [1..k] $ \i -> do
      putStrLn $ "i = " ++ show i

```

Здесь удачно организованная анонимная функция и `do`-блок отлично мимикрируют под настоящий оператор `for` в Си-стиле. Вот такой будет результат вывода:

```

runghc cycle.hs
Type a number:4
i = 1
i = 2
i = 3
i = 4

```

В следующих лекциях мы увеличим наше знание об императивных возможностях — прежде всего рассмотрим различные возможности изменяемых переменных.

Работа с текстовыми файлами. Шаг-1. Кодировки

Рассмотрим работу с текстовыми файлами, в самом простом варианте, когда мы сразу, хотя и «ленивым образом», считываем весь текстовый файл в одну строку (string), разбиваем (или не разбиваем) её в список строк (lines) по символу конца строки, затем обрабатываем.

Предположим, есть задача — все буквенные символы текстового файла перевести в верхний регистр. Вот всё решение:

```
import Data.Char
main = do
  file <- readFile "test.txt"
  writeFile "testUp.txt" (map toUpper file)
```

Здесь операция **readFile** считывает весь файл и передает его в «переменную» **file**. Ленивость позволяет на самом деле избежать загрузки всего файла сразу в оперативную память, и таким образом, в большинстве случаев, считывание файла, его обработка и запись идут по мере необходимости.

(?? были исследования, что ленивость тут не всегда ленивая на самом деле... необх. уточнить!!)

Затем, с помощью «чистой функции» **toUpper** мы преобразуем каждый Char-символ в строке **file**, используя функцию **map** для обработки списков — в нашем случае, напомним, строки — это списки символов. И, наконец, результат записывает в другой файл **testUp.txt**.

Отметим, что по умолчанию, операции чтения и записи из **Prelude** работают с кодировкой текстовых файлов и стандартного ввода-вывода в CP866 (несмотря на то, что код в UTF-8).

Рассмотрим это же решение, но с разбивкой на строки — возможно, когда-нибудь нам понадобится обработка отдельных строк.

```
import Data.Char

transform :: String -> String
transform str = map toUpper str

main = do
  file <- readFile "test.txt"
  let str_lst = lines file
  let up_lst = map transform str_lst
  writeFile "testUp.txt" (unlines up_lst)
```

Здесь «чистая функция» **transform** преобразует в строках символы к верхнему регистру. «Чистая функция» **lines** разбивает входящую строку на список строк по признаку конца строки, а функция **unlines**, наоборот, получив список строк, склеивает их в одну.

Работа с «чистыми функциями» в монадическом коде (для нас пока это означает «внутри блока **do**») должна осуществляться с помощью конструкции **let**.

Задание к уроку. Написать программку, которая считывая текстовый файл, оставляет только буквы, а их в свою очередь приводит в нижний регистр. (*Указание:* использовать **filter**, **isAlpha**, **isLetter** или **isLower**, **toLower**, см. документацию по модулю [Data.Char](#))

Более тонкое управление и работа с файловыми дескрипторами осуществляется с помощью библиотеки [System.IO](#).

Давайте посмотрим её возможности. Повторим решение задачи выше.

```
import Data.Char
import System.IO

main :: IO ()
main = do

    openhandle <- openFile "test.txt" ReadMode
    writehandle <- openFile "testUp.txt" WriteMode
    hSetEncoding writehandle utf8

    contents <- hGetContents openhandle
    hPutStr writehandle (map toUpper contents)

    hClose openhandle
    hClose writehandle
```

Здесь **openFile** открывает файл в одной из следующих мод, задаваемых типом **IOMode**:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Этот тип содержит перечисление режимов открытия файла. Мы можем привязать это действие к дескриптору, для которого мы можем настраивать параметры буфера или задавать кодировку.

В примере выше мы открыли файл `test.txt` и прочитали его в кодировке по умолчанию (для Windows это CP866, для Linux — utf8), а записали явным образом в кодировке utf8.

К сожалению, «из коробки» под Windows доступны только две кодировки: CP866, которая работает без указаний по умолчанию в стандартной консоли, и utf8, которую мы должны явно указывать. Открытые текстовые файлы в CP866 неявно транслируются в utf8 при обработке, а затем, если не указано utf8, а затем опять неявно транслируются при записи в файл или на стандартный вывод в CP866.

Поэтому, пока, если под Windows (да и под Linux) необходимо работать со строками в разных кодировках или с файлами с разными кодировками, то можно в `far` или командной строке сменить текущую кодировку на необходимую:

```
chcp 1251
1251
```

и после этого запускать необходимую программу. Например, теперь код выше будет по умолчанию работать с кодировкой CP1251 для стандартного ввода/вывода и для записи

в файл. После работы не забыть установить стандартную кодировку CP866! Кодировка utf-8 для консоли Windows тоже может быть установлена явно:

```
chcp 65001
65001
```

Но работает она «глючно» что в стандартной консоли Windows, что в консоли PowerShell.

Как это же самое делать для кодировок вроде CP1251 средствами Haskell долгое время было непонятно, но для других кодировок есть например, пакет [code-page](#) который позволяет устанавливать некоторые кодовые страницы из haskell-программ. Но...

upd: 2020 Удалось решить!

По материалу из статьи 2020-го года

[Haskell with UTF-8](#)

удалось узнать о «внутренностях» GHC по работе с кодировками (локальной и произвольной). Эта информация «упрятана» во внутреннем пакете:

[GHC.IO.Encoding](#)

И, таким образом, если нам необходима работа с ним, мы должны импортировать его:

```
import GHC.IO.Encoding
```

И далее, можно, например, посмотреть локальную кодировку:

```
lc <- getLocaleEncoding
print lc
```

или установить какую-либо (из стандартных):

```
import Data.Char
import System.IO
import GHC.IO.Encoding
```

```
main :: IO ()
main = do
```

```
    openhandle <- openFile "test.txt" ReadMode
    -- hSetEncoding openhandle
```

```
    lc <- getLocaleEncoding
    print lc
```

```
    cp <- mkTextEncoding "cp1251"
    print cp
```

```
    writehandle <- openFile "testUp.txt" WriteMode
    -- hSetEncoding writehandle utf8
    hSetEncoding writehandle cp
```

```
    contents <- hGetContents openhandle
```

```
hPutStr writehandle (map toUpper contents)
```

```
hClose openhandle
```

```
hClose writehandle
```

Здесь мы вновь открыли файл `text.txt` и прочитали его в кодировке по умолчанию (для Windows это CP866, для Linux — utf8), а записали явным образом в кодировке cp1251 в файл `testUp.txt`.

Необходимая кодировка создаётся с помощью монадического действия из пакета `GHC.IO.Encoding`

```
mkTextEncoding :: String -> IO TextEncoding
```

и затем передаётся в переменную `cp` (уже в чистом типе `TextEncoding`).

Ранее был хороший и доступный пакет [encoding](#), который позволял открывать и записывать файлы в разных кодировках, транслировать отдельные строки. Его достоинство: независимость от бинарных «сишных» библиотек и возможность работы с большим числом актуальных кодировок.

К сожалению, он теперь нормально компилируется и работает только под Linux. Мне удалось собрать его и под Windows, но заставить его нормально работать пока не удалось.

Вот, пока памятка, как я собирал недостающие части для него и проводил компиляцию. Если мне получится его собрать и заставить работать, то подробный отчет я выложу.

UPD 2021: Пакет удалось скомпилировать и он работает. Детали будут добавлены уже после лекции в текст! Инструкция по сборке будет в техническом разделе.

Приложение

```
pkgfile -s langinfo.h  
msys/msys2-runtime-devel
```

```
pacman -S msys2-runtime-devel
```

Потом

```
cabal install encoding -v3  
--extra-include-dirs=C:\\msys64\\usr\\include  
--extra-lib-dirs=C:\\msys64\\usr\\lib
```

и ждать полчаса...

Либо, (и это более правильное решение) открывать и закрывать файлы в «бинарном режиме», а затем транслировать в различные кодировки с помощью `haskell`-биндинга к известной библиотеке [iconv](#). Эта библиотека хорошо устанавливается под Linux:

```
cabal install iconv
```

а вот под Windows тоже ряд мучений при установке, но есть хорошая инструкция, по которой вполне успешно можно собрать эту библиотеку и заставить ее нормально работать, см. тут в ответе на вопросы:

[Linking c++ dll with Haskell-Platform on Windows](#)

И в нашем Classroom'е:

[Установка пакетов с “внешними либами” в новой haskell-dev](#)

[Установка пакетов с “внешними либами” в старой haskell-platform](#)

Если же библиотека установлена, то работа с ней выглядит примерно так:

```
import qualified Data.ByteString.Lazy as L
import qualified System.IO as S
import qualified Codec.Text.IConv as IConv
import qualified Data.Text.Lazy as T
import qualified Data.Text.Lazy.Encoding as TE
import qualified Data.Text.Lazy.IO as TLIO

main :: IO ()
main = do

    openhandle <- S.openFile "test.txt" S.ReadMode

    writehandle <- S.openFile "testUp.txt" S.WriteMode
    S.hSetEncoding writehandle S.utf8

    contents <- L.hGetContents openhandle
    let convcont = IConv.convert "CP1251" "UTF-8" contents
    let readysr = TE.decodeUtf8 convcont

    TLIO.hPutStr writehandle (T.toUpper readysr)

    S.hClose openhandle
    S.hClose writehandle
```

Здесь мы начали использовать «байтовые строки» для работы с бинарными данными и библиотекой `iconv`. А с помощью функции `decodeUtf8` преобразуем байтовые строки в текстовой вид. В общем-то, можно было бы и традиционные строки использовать, но это преобразование все равно будет идти (?? надо проверить ??) через тип `Text`. Поэтому, выбираем работу с ним напрямую в этом месте. Тем более, что есть необходимые функции, напр., позволяющие изменить регистр для текстовой строки сразу `toUpper`, без представление в виде символов.

На прошлой лекции мы изучали поиск и замену средствами модуля `Data.Text`. Рассмотрим ниже «строгий вариант» этого модуля. Ввод-вывод тоже организуем средствами `Data.Text.IO` этой библиотеки. Кроме того, чтобы не проводить упаковку строк в текст

```
pack :: String -> Text
```

мы используем прагму

```
{-# LANGUAGE OverloadedStrings #-}
```

И вот рабочий код:

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T
import qualified Data.Text.IO as TIO

main :: IO ()
main = do
```

```
    text <- TIO.readFile "test.txt"
    TIO.writeFile "repltest.txt" (T.replace "ты" "я" text)
```

Для рассмотренной на прошлой лекции библиотеки обработки регулярных выражений [Text.Regex.PCRE.Heavy](#) также заявлено умение работать с текстовым представлением.

Однако, в этом случае, работать пример отказался... :((2018)

```
{-# LANGUAGE OverloadedStrings, QuasiQuotes, FlexibleContexts #-}

import Data.Text as T
import qualified Data.Text.IO as TIO
import Text.Regex.PCRE.Heavy

main = do
    text <- TIO.readFile "mailtxt.txt"
    let
        text2 :: Text
        text2 = gsub ([re|@[A-Za-z]+\.[A-Za-z]+|]) ("@mail.ru" :: T.Text) text
    TIO.writeFile "mailtxt.ready.txt" text2
```

upd. 2019: Всё сработало!

Такое вот решение вполне себе работает в любом случае:

```
{-# LANGUAGE OverloadedStrings, QuasiQuotes, FlexibleContexts #-}

import Data.Text
import qualified Data.Text.IO as TIO
import Text.Regex.PCRE.Heavy

main :: IO ()
main = do
    text <- TIO.readFile "mailtxt.txt"
    let mailstr = pack "@mail.ru"
    let
        text2 :: Text
        text2 = gsub ([re|@[A-Za-z]+\.[A-Za-z]+|]) mailstr text
    TIO.writeFile "mailtxt.ready.txt" text2
```

upd. 2021: Установка пакета `encoding`

После долгого перерыва возобновлена поддержка пакета [encoding](#). Его теперь можно собрать в среде Windows (проверено для Windows 7 64bit, ghc 8.6.5)

Перед установкой желательно сделать настройки, указанные в тех. статьях в Classroom, посвящённых установке дополнительных библиотек. Также, в оболочке `msys2` желательно выполнить команды по установке `msys2-runtime-devel`:

```
>pkgfile -s langinfo.h
msys/msys2-runtime-devel

>pacman -S msys2-runtime-devel
```

Хотя, судя по всему, современная версия установки под Windows от этого уже не зависит.

Установка пакета [encoding](#) под Windows должна происходить с флагом `-f-systemencoding`

```
cabal install encoding -f-systemencoding
```

как это указано тут:

[In Haskell, how do I install the encoding package on Windows?](#)

Компиляция и сборка идёт довольно долго, не менее 10 мин...

После установки, работа с пакетом напоминает таковую с `iconv`. Рассмотрим примеры.

Наиболее удобно (и необычно) работать в этой библиотеке с так называемым неявным параметром

6.11.5. Implicit parameters

В данном примере мы меняем текущую кодировку для консоли на CP1251, автоматически меняется и кодировка вывода

```
{-# LANGUAGE ImplicitParams #-}

import System.IO (stdout)
import System.IO.Encoding (hPutStrLn)
import Data.Encoding.CP1251

s = "Русский язык"

main = do
    let ?enc = CP1251
    hPutStrLn (stdout) s
```

(работает только для `import System.IO.Encoding (hPutStrLn)`, для `import System.IO (hPutStrLn)` работать не будет, и не будет работать для стандартного `putStrLn`)

Следующий пример показывает работу, которую было легче и удобнее сделать с `GHC.IO.Encoding`, что было выше в лекции.

```
{-# LANGUAGE ImplicitParams #-}

import Data.Char
```



```

import System.IO hiding (hGetContents, hPutStr)
import System.IO.Encoding
import Data.Encoding.CP1251

main :: IO ()
main = do
    let ?enc = CP1251

    openhandle <- openFile "test1251.txt" ReadMode
    -- hSetEncoding openhandle CP1251

    writehandle <- openFile "testUp.txt" WriteMode
    hSetEncoding writehandle utf8

    contents <- hGetContents openhandle
    hPutStr writehandle (map toUpper contents)

    hClose openhandle
    hClose writehandle

```

Ну и, наконец, перекодировка отдельных строк, аналогичная таковой в `iconv` и без использования `?enc`. Библиотека менее удобна чем `iconv`, так как большинство преобразований трансформируют либо **String** в `ByteString`, либо наоборот (нет преобразований одного типа, нет преобразований с `Text`).

Следующий пример показывает явное преобразование строки **String** в байтовую цепочку типа `ByteString` в кодировке CP1251

```

import Data.Encoding
import Data.Encoding.CP1251
import qualified Data.ByteString.Lazy as L

s = "Русский язык\n"

s2 = encodeLazyByteString CP1251 s

main = do
    L.putStr s2

```

И последний пример показывает работу с текстовым представлением и использованием явных преобразований между всеми тремя типами (`Text`, **String**, `ByteString`) для смены кодировки и работы с текстовым файлом в текстовом представлении `Text` в качестве основного.

```

import qualified Data.ByteString.Lazy as L
import qualified System.IO as S
import Data.Encoding
import Data.Encoding.CP1251
import Data.Encoding.UTF8
import qualified Data.Text.Lazy as T
import qualified Data.Text.Lazy.Encoding as TE

```

```

import qualified Data.Text.Lazy.IO as TLIO

main :: IO ()
main = do

    openhandle <- S.openFile "test.txt" S.ReadMode

    writehandle <- S.openFile "testUp.txt" S.WriteMode
    S.hSetEncoding writehandle S.utf8

    contents <- L.hGetContents openhandle
    let convcont =
        encodeLazyByteString UTF8 $ decodeLazyByteString CP1251 contents

    let readysttr = TE.decodeUtf8 convcont

    TLIO.hPutStr writehandle (T.toUpper readysttr)

    S.hClose openhandle
    S.hClose writehandle

```

Вывод по выбору инструмента работы с кодировками

Рассмотренные три модуля (и пакета) обладают своими достоинствами и недостатками:

- системный модуль `GHC.IO.Encoding` является универсальным системным решением, не требующим дополнительных установок, однако он ограничен работой с дескрипторами и с файлами *целиком*, не поддерживая перекодировку отдельных фрагментов текстов и строк;
- пакет `encoding` является портабельным (кросс-платформенным) решением, позволяя работать как на уровне файлов, так и на уровне отдельных фрагментов строк, но ограничиваясь преобразованиями между `String` и `ByteString` (отсутствует прямая работа с `Text`), кроме того, интерфейс менее развит, чем в `iconv`;
- пакет `iconv` — быстрое и проверенное решение из мира Си и платформы Linux, позволяющее удобнее (понятнее интерфейс, умеет работать с типом `Text`) делать то же самое, что и пакет `encoding`, однако он линкуется на бинарном уровне и будет испытывать проблемы с портабельностью.