Структура программ. Работа с текстовыми файлами

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 должны иметь точку входа, именованную «main», так же как и в других Си-подобных языках. Предполагаем, что её тип должен быть `IO()`. Вот классический пример:

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

или, с указанием типа

```
main :: IO
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 этого скрипта с аргументами

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

мы получим следующий вывод:

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

Для более серьёзных 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 получем 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 неподходящие, то будет сообщение об ошибке вроде такого:

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

Библиотеку надо устанавливать самостоятельно:

```
cabal install ReadArgs --lib
```

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

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

(здесь мы ожидаем, что пользователь введёт `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 нелатинских букв, выведутся номера символов:

```
main = print "Приветмир !"
```

получим:

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

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

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

UPD: 2019-11-14:

Для корректного вывода кириллицы в консоли можно 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 "Приветмир !"
```

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

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 вернём 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 её 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 (включённый буфер для канала **stdout** может изменить порядок вывода или привести к другим неприятностям в интерактивной консольной программе).

Следующий, почти аналогичный пример взят из официального сборника Haskell Report

```
import System.IO
main = do
         hSetBuffering stdout NoBuffering
         putStr "Введитецелоечисло  : "
         x1 <- readNum
         putStr  "Введитедругоецелоечисло   : "
         x2 <- readNum
         putStr  ("Их суммаравна  " ++ 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).

Ещё 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 возращает подходящее значение), то мы используем конструкцию **`let`** `y = f x`, вместо конструкции `y <- f x`.

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 её 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. 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 объёмный 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, обёрнутое 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.

В Haskell переменные, однажды определённые, будут далее неизменны (точнее, обычно речь идёт о безаргументных функциях) . Для работы с изменяемыми переменными нам, например, необходимо монаду **IO** обернуть монадой `StateT`, или использоватьдругие трюки (будет в след. лекциях пример).

Но в данном случае, при связывании, «переменную» `str` можно считать как бы изменяемой, такой код демонстрирует эффект изменяемости:

```
...
putStr "Enter a string: "
str <- getLine
print str
putStr "Enter a string again: "
str <- getLine
print str
```

Но по факту, это разные «переменные», хотя и имеют одинаковые имена. Это хитрый эффект как работают связывания, сравните, например

```
g1 = \t -> (\x -> (x+t))
f1 = \x -> (\x -> (x+1))
```

или тут

```
g2 = let x=1 in let y=2 in (x+y)
f2 = let x=1 in let x=2 in (x+x)
```

где во внутреннем выражении мы как бы один `x` закрываем другим `x` (однако, выполнить код для `f1` нельзя, но для `f2` вполне можно).

Далее. 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
```

И тут так же возможно многократное «присваивание» (вводящее нас в заблуждение выше):

```
  let x = 4
  print x
  let x = 5
  print x
```

Если же нужно указать тип 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
```

Кстати, усечённый вариант **let** позволяет определять и «грязные функции», т.е. действия внутри do-блока:

```
main = do
  let pr = print
  pr "Hello"
```

Кроме того, внутри do-блока мы можем использовать ветвление **if-then-else**:

```
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!
```

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 текстовой файл в одну строку (string), разбиваем (или не разбиваем) её в список строк (lines) по символу конца строк, 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 передаёт его в «переменную» `file`. Ленивость позволят 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.

Кстати, были исследования, что ленивость тут не всегда ленивая на самом деле:

Understanding Iteratees

Oleg Kiselyov. Lazy vs correct IO

Ленивый hGetContents. Баг или фича?

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

Отметим, что по умолчанию в консоли Windows, операции чтения и записи из **Prelude** работают с кодировкой текстовых файлов и стандартного ввода-вывода в `CP866` (несмотря на то, что код программы в `UTF-8`). Ниже будем об этом говорить подробнее.

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.

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

Задание. Написать программку, которая считывая текстовой 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 к дескриптору, для которого мы можем настраивать параметры буфера или задавать кодировку.

В примере выше мы открыли файл text.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, 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 консоли 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 кодирвку `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. В Linux для аналогичного решения необходимо менять значение системной переменной

```
export LANG=ru_RU.UTF-8
```

Корректная работа с кодировками на уровне потоков ввода-вывода

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 «грязной функции» (монадического действия из модуля `GHC.IO.Encoding`) из известных имён кодировок

```
mkTextEncoding :: String -> IO TextEncoding

cp <- mkTextEncoding "cp1251"
```

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

Перекодирование отдельных строк

Возможности перекодировки отдельных строк и фрагментов текстовых файлов (а также целиком файлов) предоставляют две библиотеки iconv и encoding. Первая — это обёртка над системной библиотекой `libiconv`, вторая — чистый код на Haskell. Познакомимся с их возможностями.

Библиотека iconv

Если же библиотека установлена, то работа с перекодировкой целого файла (или отдельных строк) 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 readystr = TE.decodeUtf8 convcont

    TLIO.hPutStr writehandle (T.toUpper readystr)
```

```
    S.hClose openhandle
    S.hClose writehandle
```

С помощью функции **openFile** системного модуля **System.IO** мы назначаем дескриптор файла в openhandle `test.txt` на чтение и `writehandle` — на запись, при этом стандартными средствами Haskell записывать мы будем в `utf8`.

Здесь мы начали использовать «байтовые строки» для работы с бинарными данными: «грязная» функция **hGetContents** из модуля Data.ByteString.Lazy «лениво» читает сразу всё содержимое файла в виде списка байтов.

Собственно, сама перекодировка делает функция `convert` из библиотеки iconv. Она преобразует последовательность байт из одной кодировки в другую, которые ей передаются в качестве аргументов.

После этого функцией `decodeUtf8` модуля работы с текстом Data.Text.Lazy.Encoding `decodeUtf8` преобразуем байтовые строки в текстовое представление. В общем-то, можно было бы и традиционные строки использовать, но это преобразование всё равно будет идти (?? надо проверить ??) через тип `Text`. Поэтому, выбираем работу с ним напрямую в этом месте. Тем более, что там есть необходимые функции, напр., позволяющие изменить регистр для текстовой строки сразу **toUpper**, без представление в виде символов.

Запись полученного текстового представления мы делаем с помощью действия **hPutStr** из модуля `Data.Text.Lazy.IO` из этого же пакета работы с текстом.

Data.Text и строковые литералы

В следующей лекции мы будем говорить подробно о типе данных Data.Text и его возможностях.

На прошлой лекции мы изучали поиск и замену средствами модуля `Data.Text`. Рассмотрим ниже «строгий вариант» этого модуля. Вввод-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 #-}
```

для избавления от конвертаций из текста в троки и обратно. Эта прагма позволяет использовать строковый литерал `"..."` представляя его в типе **String** или `Text` в зависимости от места использования:

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 выраженией Text.Regex.PCRE.Heavy также заявлено умение работать с текстовым представлением.

Рассмотрим пример работы с текстовым представлением и заменой.

```
{-# 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
```

Библиотека encoding

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 = "Русскийязык "

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 по конвертации всего файла, которую было легче и удобнее сделать с `GHC.IO.Encoding`, что было выше в этой 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 = "Русскийязык \n"

s2 = encodeLazyByteString CP1251 s

main =  do
          L.putStr s2
```

(этот пример позволяет вывести даже «кракозяблы» без изменения локальной кодировки консоли, но правильнее сменить кодировку консоли на `CP1251`)

В примере выше можно было бы использовать перекодирование из строки в строку:

```
s2 = encodeString CP1251 s
```

Но вывести результат в консоль мы не сможем.

И последний пример показывает работу с текстовым представлением и использованием явных преобразований между типами (`ByteString` и `Text`) для смены кодировки и работы

с текстовым файлом в текстовом представлении `Text` в качестве основного. И тоже, не слишком удобно.

```
import qualified System.IO as S
import qualified Data.ByteString.Lazy as L
import qualified Data.Encoding as E
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 = E.encodeLazyByteString UTF8 $
      E.decodeLazyByteString CP1251 contents

    let readystr = TE.decodeUtf8 convcont

    TLIO.hPutStr writehandle (T.toUpper readystr)

    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 преобразованиям 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.

Приложение

Установка соответствующих пакетов и необходимых системных библиотек рассмотрена в разделе «Техническое» в ClassRoom.