Министерство науки и высшего образования Российской Федерации
Калужский филиал
федерального государственного автономного
образовательного учреждения высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИУК «Информатика и управление»**

КАФЕДРА **ИУК5 «Системы обработки информации»**

ЛАБОРАТОРНАЯ РАБОТА №3

«Вычисления в контексте»

ДИСЦИПЛИНА: «Перспективные технологии разработки программных средств»

Выполнил: студент гр. ИУК4-31М _____ (Сафронов Н.С,)
(подпись) (Ф.И.О.)

Проверил: _____ (Кириллов В.Ю.)
(подпись) (Ф.И.О.)

Дата сдачи (защиты):

Результаты сдачи (защиты):

- Балльная оценка:
- Оценка:

Калуга, 2025

Цель:

Изучение контекстных вычислений.

Задачи:

- Контекст ввода-вывода и работа с файлами
- Применение контекстных классов
- Написание отчета о работе

Результаты выполнения работы

Задача 21.2

Напишите программу, запрашивающую у пользователя число n и возвращающую n-е число Фибоначчи (пример вычисления чисел Фибоначчи можно найти в уроке 8).

Листинг программы

```
fib :: Integer -> Integer
fib n
  | n == 0    = 0
  | n == 1    = 1
  | otherwise = fibIter 0 1 2
  where
    fibIter a b i
      | i > n     = b
      | otherwise = fibIter b (a + b) (i + 1)

main :: IO ()
main = do
  putStr "Введите номер числа Фибоначчи: "
  input <- getLine
  let n = read input :: Integer
  if n >= 0
    then putStrLn $ "F(" ++ show n ++ ") = " ++ show (fib n)
    else putStrLn "Пожалуйста, введите неотрицательное число."
```

Задача 22.2

Напишите программу, предлагающую пользователю выбрать число от 1 до 15 и затем выводящую известную цитату (выбор цитат предоставляется вам). После вывода цитаты программа должна спрашивать у пользователя о желании увидеть ещё одну. Если пользователь вводит "n", программа останавливается; иначе он получает ещё одну цитату.

Программа повторяет вышеописанные действия до тех пор, пока пользователь не введёт "n". Попробуйте использовать ленивые вычисления и работать с пользовательским вводом как со списком, вместо того чтобы рекурсивно вызывать main в конце.

Листинг программы

```
import Data.List (intercalate)

quotes :: [String]
quotes =
  [ "Запомни: всего одна ошибка - и ты ошибся."
  , "Делай, как надо. Как не надо, не делай."
  , "Работа - это не волк. Работа - ворк. А волк - это ходить."
  , "Не будьте эгоистами, в первую очередь думайте о себе!"
  , "Мы должны оставаться мыми, а они - ними."
  , "Марианскую впадину знаешь? Это я упал."
  , "Как говорил мой дед, \"Я твой дед\"."
  , "Жи-ши пиши от души."
  , "Без подошвы тапочки - это просто тряпочки."
  , "Слово - не воробей. Вообще ничто не воробей, кроме самого воробья."
  , "Если тебе где-то не рады в рваных носках, то и в целых туда идти не
стоит."
  , "Работа не волк. Никто не волк. Только волк волк."
  , "Если закрыть глаза, становится темно."
  , "Тут - это вам не там."
  , "Кто рано встаёт - тому весь день спать хочется."
  ]

-- Функция для получения цитаты по индексу (от 1 до 15)
getQuote :: Int -> String
getQuote n
  | n >= 1 && n <= length quotes = quotes !! (n - 1)
  | otherwise = "Цитата не найдена."

-- Обработка ввода: возвращает список строк вывода
processInputs :: [String] -> [String]
processInputs (input : rest) =
  case input of
    "x" -> ["\n"]
    _   -> case reads input of
      [(n, "")] | n >= 1 && n <= 15 ->
        let quote = getQuote n
        in (quote ++ "\n\nВыберите число от 1 до 15 или 'n' для выхода: ") :
processInputs rest
      _ -> ("Некорректный ввод. Введите число от 1 до 15 или 'n' для
выхода.\n\n" ++
            "Выберите число от 1 до 15 или 'n' для выхода: ") : processInputs
rest
processInputs [] = []

main :: IO ()
main = do
  putStr "Выберите число от 1 до 15 или 'n' для выхода: "
  inputLines <- fmap lines getContents
  let outputs = processInputs inputLines
  putStr $ intercalate "" outputs
```

Задача 24.1

Напишите свой аналог программы cp в Unix, копирующей файл и позволяющей его при этом переименовать (достаточно воспроизвести только базовый функционал; не волнуйтесь о специфических возможностях утилиты cp).

Листинг программы

```
import System.Environment (getArgs)
import qualified Data.ByteString.Lazy as B

main :: IO ()
main = do
  args <- getArgs
  case args of
    [src, dest] -> copyFile' src dest
    _ -> putStrLn "Использование: mycp <исходный_файл> <новый_файл>"

copyFile' :: FilePath -> FilePath -> IO ()
copyFile' src dest = do
  content <- B.readFile src
  B.writeFile dest content
```

Задача 24.2

Напишите программу под названием capitalize.hs, принимающую имя файла в качестве аргумента, считывающую этот файл и перезаписывающую его теми же символами, но прописными буквами

Листинг программы

```
import System.Environment (getArgs)
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import Data.Char (toUpper)

main :: IO ()
main = do
  args <- getArgs
  case args of
    [fileName] -> capitalizeFile fileName
    _ -> putStrLn "Использование: capitalize <имя_файла>"

capitalizeFile :: FilePath -> IO ()
capitalizeFile fileName = do
  content <- TIO.readFile fileName
  let upperContent = T.map toUpper content
  TIO.writeFile fileName upperContent
```

Задача 25.1

Напишите программу, которая сравнивает количество символов в текстовом файле с его размером в байтах.

Листинг программы

```
import System.Environment (getArgs)
import System.IO (withBinaryFile, IOMode(ReadMode), hFileSize)
```

```
import qualified Data.ByteString as BS

main :: IO ()
main = do
  args <- getArgs
  case args of
    [fileName] -> compareFileSize fileName
    _ -> putStrLn "Использование: compare_size <имя_файла>"

compareFileSize :: FilePath -> IO ()
compareFileSize fileName = do
  content <- readFile fileName
  let charCount = length content

  fileSize <- withBinaryFile fileName ReadMode hFileSize

  putStrLn $ "Количество символов: " ++ show charCount
  putStrLn $ "Размер файла в байтах: " ++ show fileSize
  putStrLn $ if fromIntegral charCount == fileSize
             then "Количество символов совпадает с размером в байтах."
             else "Количество символов не совпадает с размером в байтах."
```

Задача 25.2

Реализуйте ещё один способ добавления помех под названием randomReverseBytes, который переворачивает случайную подпоследовательность байтов в файле.

Листинг программы

```
import System.Environment (getArgs)
import qualified Data.ByteString as BS
import System.Random (randomRIO)
import Data.List (sortBy)
import Data.Ord (comparing)

main :: IO ()
main = do
  args <- getArgs
  case args of
    [fileName] -> randomReverseBytes fileName
    _ -> putStrLn "Использование: random_reverse_bytes <имя_файла>"

randomReverseBytes :: FilePath -> IO ()
randomReverseBytes fileName = do
  content <- BS.readFile fileName
  let len = BS.length content
  if len < 2
    then putStrLn "Файл слишком мал для реверса подпоследовательности."
    else do
      start <- randomRIO (0, len - 2)
      end   <- randomRIO (start + 1, len - 1)
      let reversedContent = reverseSubBytes content start end
      BS.writeFile fileName reversedContent
      putStrLn $ "Реверс байтов с " ++ show start ++ " по " ++ show end ++ "
выполнен."
```

```
reverseSubBytes :: BS.ByteString -> Int -> Int -> BS.ByteString
reverseSubBytes bs start end =
  let (before, rest) = BS.splitAt start bs
      (sub, after) = BS.splitAt (end - start + 1) rest
  in BS.concat [before, BS.reverse sub, after]
```

Задача 26

Листинг программы

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as E
import Data.Text.Encoding.Error (OnDecodeError, strictDecode,
UnicodeException)
import Data.Maybe
import System.Environment (getArgs)
import Data.Char (chr, ord)

main :: IO ()
main = do
  args <- getArgs
  case args of
    [inputFile, outputFile] -> do
      marcData <- B.readFile inputFile
      let processed = processRecords 500 marcData
      -- Пришлось фильтровать, потому что где-то попадается не-UTF символ в
имени книги
      let safeProcessed = T.filter (\c -> ord c /= 65533) processed
      TIO.writeFile outputFile safeProcessed
      putStrLn $ "Файл " ++ outputFile ++ " успешно создан."
    _ -> putStrLn "Использование: 26.exe <входной.mrc> <выходной.html>"

type Author = T.Text
type Title = T.Text

data Book = Book {
  author :: Author
 ,title :: Title } deriving Show

type Html = T.Text

bookToHtml :: Book -> Html
bookToHtml book = mconcat ["<p>\n"
                     ,titleInTags
                     ,authorInTags
                     ,"</p>\n"]
  where titleInTags = mconcat ["<strong>", title book, "</strong>\n"]
        authorInTags = mconcat ["<em>", author book, "</em>\n"]

booksToHtml :: [Book] -> Html
booksToHtml books = mconcat ["<html>\n"
                            ,"<head><title>books</title>"
```

```
                              ,"<meta charset='utf-8'/>"
                              ,"</head>\n"
                              ,"<body>\n"
                              ,booksHtml
                              ,"\n</body>\n"
                              ,"</html>"]
  where booksHtml = (mconcat . map bookToHtml) books
type MarcRecordRaw = B.ByteString
type MarcLeaderRaw = B.ByteString

leaderLength :: Int
leaderLength = 24

getLeader :: MarcRecordRaw -> MarcLeaderRaw
getLeader = B.take leaderLength

rawToInt :: B.ByteString -> Int
rawToInt = read . T.unpack . E.decodeUtf8

getRecordLength :: MarcLeaderRaw -> Int
getRecordLength leader = rawToInt (B.take 5 leader)

nextAndRest :: B.ByteString -> (MarcRecordRaw, B.ByteString)
nextAndRest marcStream = B.splitAt recordLength marcStream
  where recordLength = getRecordLength marcStream

allRecords :: B.ByteString -> [MarcRecordRaw]
allRecords marcStream = if marcStream == B.empty
                        then []
                        else next : allRecords rest
  where (next, rest) = nextAndRest marcStream

type MarcDirectoryRaw = B.ByteString

getBaseAddress :: MarcLeaderRaw -> Int
getBaseAddress leader = rawToInt (B.take 5 remainder)
  where remainder = B.drop 12 leader

getDirectoryLength :: MarcLeaderRaw -> Int
getDirectoryLength leader = getBaseAddress leader - (leaderLength + 1)

getDirectory :: MarcRecordRaw -> MarcDirectoryRaw
getDirectory record = B.take directoryLength afterLeader
  where directoryLength = getDirectoryLength record
        afterLeader = B.drop leaderLength record

type MarcDirectoryEntryRaw = B.ByteString

dirEntryLength :: Int
dirEntryLength = 12

splitDirectory :: MarcDirectoryRaw -> [MarcDirectoryEntryRaw]
splitDirectory directory = if directory == B.empty
                           then []
                           else nextEntry : splitDirectory restEntries
```

```
  where (nextEntry, restEntries) = B.splitAt dirEntryLength directory

data FieldMetadata = FieldMetadata {tag :: T.Text
                                   ,fieldLength :: Int
                                   ,fieldStart :: Int } deriving Show

makeFieldMetadata :: MarcDirectoryEntryRaw -> FieldMetadata
makeFieldMetadata entry = FieldMetadata textTag theLength theStart
  where (theTag, rest) = B.splitAt 3 entry
        textTag = E.decodeUtf8 theTag
        (rawLength, rawStart) = B.splitAt 4 rest
        theLength = rawToInt rawLength
        theStart = rawToInt rawStart

getFieldMetadata :: [MarcDirectoryEntryRaw] -> [FieldMetadata]
getFieldMetadata = map makeFieldMetadata

type FieldText = T.Text

getCharacterCodingScheme :: MarcLeaderRaw -> Char
getCharacterCodingScheme leader = chr $ fromEnum $ B.index leader 9

getTextField :: MarcRecordRaw -> FieldMetadata -> FieldText
getTextField record fieldMetadata = decodeBasedOnCoding record
byteStringValue
  where recordLength = getRecordLength record
        baseAddress = getBaseAddress record
        baseRecord = B.drop baseAddress record
        baseAtEntry = B.drop (fieldStart fieldMetadata) baseRecord
        byteStringValue = B.take (fieldLength fieldMetadata) baseAtEntry

skipDecodeError :: OnDecodeError
skipDecodeError x y = Nothing

decodeBasedOnCoding :: MarcRecordRaw -> B.ByteString -> FieldText
decodeBasedOnCoding record raw =
  if codingScheme == 'a' || codingScheme == ' '
    then E.decodeUtf8 raw
    else E.decodeUtf8With skipDecodeError raw
  where codingScheme = getCharacterCodingScheme record

fieldDelimiter :: Char
fieldDelimiter = toEnum 31

titleTag :: T.Text
titleTag = "245"

titleSubfieldA :: Char
titleSubfieldA = 'a'

titleSubfieldB :: Char
titleSubfieldB = 'b'

authorTag :: T.Text
authorTag = "100"
```

```
authorSubfield :: Char
authorSubfield = 'a'

lookupFieldMetadata :: T.Text -> MarcRecordRaw -> Maybe FieldMetadata
lookupFieldMetadata aTag record = if null results
                                  then Nothing
                                  else Just (head results)
  where metadata = (getFieldMetadata . splitDirectory . getDirectory) record
        results = filter ((== aTag) . tag) metadata

lookupAllSubfields :: Maybe FieldMetadata -> Char -> MarcRecordRaw ->
[T.Text]
lookupAllSubfields Nothing subfield record = []
lookupAllSubfields (Just fieldMetadata) subfield record =
  let rawField = getTextField record fieldMetadata
      subfields = T.split (== fieldDelimiter) rawField
  in map (T.drop 1) $ filter ((== subfield) . T.head) subfields

lookupTitle :: MarcRecordRaw -> Maybe Title
lookupTitle record = do
  fieldMetadata <- lookupFieldMetadata titleTag record
  let aSubfields = lookupAllSubfields (Just fieldMetadata) titleSubfieldA
record
      bSubfields = lookupAllSubfields (Just fieldMetadata) titleSubfieldB
record
  if null aSubfields
    then Nothing
    else Just $ T.intercalate " " (aSubfields ++ bSubfields)

lookupAuthor :: MarcRecordRaw -> Maybe Author
lookupAuthor record = do
  fieldMetadata <- lookupFieldMetadata authorTag record
  let subfields = lookupAllSubfields (Just fieldMetadata) authorSubfield
record
  if null subfields
    then Nothing
    else Just $ head subfields

marcToPairs :: B.ByteString -> [(Maybe Title, Maybe Author)]
marcToPairs marcStream = zip titles authors
  where records = allRecords marcStream
        titles = map lookupTitle records
        authors = map lookupAuthor records

pairsToBooks :: [(Maybe Title, Maybe Author)] -> [Book]
pairsToBooks pairs = map (\(title, author) -> Book {
                            title = fromJust title
                           ,author = fromJust author }) justPairs
  where justPairs = filter (\(title, author) -> isJust title
                                             && isJust author) pairs

processRecords :: Int -> B.ByteString -> Html
processRecords n = booksToHtml . pairsToBooks . take n . marcToPairs
```

Результаты выполнения программы

Topper (The Jovial Ghosts: The Misadventures of Topper *Smith, Thorne.*

The Stray Lamb *Smith, Thorne.*

The Bishop's Jaegers *Smith, Thorne.*

The Night Life of the Gods *Smith, Thorne.*

Topper Takes a Trip *Smith, Thorne.*

Turnabout *Smith, Thorne.*

The Glorious Pool *Smith, Thorne.*

Skin and Bones *Smith, Thorne.*

Rain in the Doorway *Smith, Thorne.*

John Cornelius *Walpole, Hugh.*

The Joyful Delaneys *Walpole, Hugh.*

On th Spot *Wallace, Edgar.*

So Well Remembered *Hilton, James.*

The Private Life of Helen of Troy *Erskine, John.*

Round the Fire Stories *Doyle, Arthur Conan.*

The Dutchess of Wiltshire's Diamonds *Boothby, Guy.*

Seven, Seven, Seven--City: A Tale of the Telephone *Chambers, Julius.*

Рисунок 1 – Пример выполнения программы на MARC 21 файле из Проекта Гуттенберга

Задача 27.3

Напишите интерфейс командной строки для базы данных компонентов роботов partsDB, который позволит пользователю искать стоимость предмета по его идентификатору. Используйте тип Maybe для обработки случая, когда пользователь запрашивает отсутствующий предмет.

Листинг программы

```
import Data.Map (Map)
import qualified Data.Map as Map

type Cost = Double

type PartsDB = Map String Cost

partsDB :: PartsDB
```

```
partsDB = Map.fromList
  [ ("R001", 25.99)
  , ("R002", 15.50)
  , ("R003", 42.75)
  , ("R004", 10.20)
  ]

lookupCost :: String -> PartsDB -> Maybe Cost
lookupCost = Map.lookup

main :: IO ()
main = do
  putStrLn "Введите идентификатор компонента (или ':q' для выхода):"
  input <- getLine
  if input == ":q"
    then putStrLn "\n"
    else do
      let result = lookupCost input partsDB
      case result of
        Nothing -> putStrLn $ "Компонент с идентификатором '" ++ input ++ "'
не найден."
        Just cost -> putStrLn $ "Стоимость компонента: " ++ show cost ++ "
у.е."
      main
```

Задача 28.3

Вспомните тип RobotPart из предыдущего урока:

```
data RobotPart = RobotPart
{ name :: String
, description :: String
, cost :: Double
, count :: Int
} deriving Show
```

Реализуйте консольное приложение, которое хранит базу из нескольких значений типа RobotPart (минимум 5), принимает от пользователя ID двух частей, а возвращает ту деталь, цена которой ниже. Обработайте случай, когда пользователь вводит ID, отсутствующий в базе.

Листинг программы

```
import Data.Map (Map)
import qualified Data.Map as Map
import Data.Maybe (fromMaybe)

data RobotPart = RobotPart
  { name :: String
  , description :: String
  , cost :: Double
  , count :: Int
  } deriving Show

partsDB :: Map String RobotPart
partsDB = Map.fromList
  [ ("R001", RobotPart "Motor" "High-torque motor" 45.99 10)
```

```
  , ("R002", RobotPart "Wheel" "Rubber wheel" 12.50 50)
  , ("R003", RobotPart "Sensor" "Proximity sensor" 28.75 20)
  , ("R004", RobotPart "Battery" "Rechargeable battery" 35.00 15)
  , ("R005", RobotPart "Controller" "Main controller" 120.00 5)
  ]

getPart :: String -> Maybe RobotPart
getPart partId = Map.lookup partId partsDB

cheaperPart :: RobotPart -> RobotPart -> RobotPart
cheaperPart p1 p2 = if cost p1 <= cost p2 then p1 else p2

getPartOrFail :: String -> IO (Maybe RobotPart)
getPartOrFail partId = return $ getPart partId

main :: IO ()
main = do
  putStrLn "Введите ID первой части:"
  id1 <- getLine
  putStrLn "Введите ID второй части:"
  id2 <- getLine

  let part1 = getPart id1
  let part2 = getPart id2

  case (part1, part2) of
    (Nothing, Nothing) -> putStrLn $ "Обе части с ID '" ++ id1 ++ "' и '" ++
id2 ++ "' не найдены."
    (Nothing, _) -> putStrLn $ "Часть с ID '" ++ id1 ++ "' не найдена."
    (_, Nothing) -> putStrLn $ "Часть с ID '" ++ id2 ++ "' не найдена."
    (Just p1, Just p2) -> do
      let cheaper = cheaperPart p1 p2
      putStrLn $ "Часть с меньшей ценой: " ++ show cheaper
```

Задача 29.3

Пользуясь приведёнными ниже данными и выполняя недетерминированные вычисления со списками, определите, сколько пива вам нужно купить, чтобы его наверняка хватило:

- вчера вы купили пиво, но не помните, была ли это упаковка из 6 или 12 бутылок;
- прошлой ночью вы с соседом выпили по две бутылки пива;
- сегодня к вам могут прийти двое или трое друзей, смотря как у них будет получаться;
- ожидается, что за ночь один человек выпьет три или четыре бутылки.

Листинг программы

```
-- Сколько бутылок было изначально
initialBeers :: [Int]
initialBeers = [6, 12]
```

```
-- Сколько уже выпито
alreadyDrunk :: Int
alreadyDrunk = 4  -- вы и сосед по 2 бутылки

-- Сколько друзей может прийти
numFriends :: [Int]
numFriends = [2, 3]

-- Сколько выпьет один человек за ночь
beersPerPerson :: [Int]
beersPerPerson = [3, 4]

-- Сколько всего выпьют за ночь

-- Общее потребление сегодня:
totalNeededTonight :: [Int]
totalNeededTonight = (*) <$> numFriends <*> beersPerPerson

-- Остаток от вчерашней покупки:
remainingBeers :: [Int]
remainingBeers = (\initial -> initial - alreadyDrunk) <$> initialBeers

-- Для каждой комбинации: сколько не хватает?
-- Если остаток >= потребности → 0
-- Иначе → потребность - остаток
deficit :: [Int]
deficit =
  [ max 0 (needed - remaining)
  | remaining <- remainingBeers
  , needed <- totalNeededTonight
  ]

-- Минимальное количество пива, которое нужно докупить, чтобы хватило всегда:
-- Это максимальный дефицит среди всех сценариев.
beersToBuy :: Int
beersToBuy = maximum deficit
```

Задача 30.2

Чтобы доказать, что Monad строго мощнее Applicative, напишите универсальную версию <*> под названием allApp, которая определит <*> для всех членов класса типов Monad. Так как она должна работать для всех экземпляров Monad, вы можете использовать только методы из определения класса Monad (и лямбда-функции). Вот тип этой функции:

```
allApp :: Monad m => m (a -> b) -> m a -> m b
```

Это задание гораздо хитрее предыдущего. Дадим две подсказки:

- попробуйте рассуждать исключительно в терминах типов;
- если хочется, примените <$>, заменив её своим ответом к задаче 29.1.

Листинг программы

```
allApp :: Monad m => m (a -> b) -> m a -> m b
```

```
allApp monadicFunc monadicValue = do
  func <- monadicFunc
  val <- monadicValue
  return (func val)
```

Задача 31.3

Перепишите код функции maybeMain из предыдущего упражнения так, чтобы он работал с любой монадой. Вам понадобится изменить типовую аннотацию и убрать из тела функции специфические для конкретного типа фрагменты кода.

Листинг программы

```
type Pizza = (Double, Double)

areaGivenDiameter :: Double -> Double
areaGivenDiameter size = pi * (size / 2)^2

costPerInch :: Pizza -> Double
costPerInch (size, cost) = cost / areaGivenDiameter size

comparePizzas :: Pizza -> Pizza -> Pizza
comparePizzas p1 p2 = if costP1 < costP2
                      then p1
                      else p2
  where costP1 = costPerInch p1
        costP2 = costPerInch p2

describePizza :: Pizza -> String
describePizza (size, cost) = "The " ++ show size ++ " pizza " ++
                             "is cheaper at " ++ show costSqInch ++
                             " per square inch"
  where costSqInch = costPerInch (size, cost)

monadMain :: Monad m => m Double -> m Double -> m Double -> m Double -> m
String
monadMain m1 m2 m3 m4 = do
  size1 <- m1
  cost1 <- m2
  size2 <- m3
  cost2 <- m4
  let pizza1 = (size1, cost1)
  let pizza2 = (size2, cost2)
  let betterPizza = comparePizzas pizza1 pizza2
  return (describePizza betterPizza)

sizeList1 :: [Double]
sizeList1 = [6]

costList1 :: [Double]
costList1 = [34.2]

sizeList2 :: [Double]
sizeList2 = [6, 12]
```

```
costList2 :: [Double]
costList2 = [32.3, 64]

main :: IO ()
main = do
  mapM_ putStrLn (monadMain sizeList1 costList1 sizeList2 costList2)
```

Задача 32.2

Перепишите решение предыдущего упражнения с использованием do-нотации, а затем выполните рассахаривание в методы класса типов Monad и лямбды.

Листинг программы

```
daysInMonth :: [Int]
daysInMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

calendarDates :: [(Int, Int)]
calendarDates = do
  month <- [1..12]
  day <- [1..daysInMonth !! (month - 1)]
  return (month, day)
```

Задача 33

Листинг программы

```
import Control.Monad
import Control.Applicative

main :: IO ()
main = do
    print (_where (startsWith 'J' . firstName) (_select studentName
students))
    print (_join teachers courses teacherId teacher)
    print (runHINQ query1)
    print (runHINQ query2)
    print (runHINQ maybeQuery1)
    print (runHINQ maybeQuery2)
    print studentEnrollments
    print (getEnrollments "English")
    print (getEnrollments "French")

data Name = Name
          { firstName :: String
          , lastName :: String }

instance Show Name where
    show (Name first last) = mconcat [first, " ", last]

data GradeLevel = Freshman
                | Sophmore
                | Junior
                | Senior deriving (Eq, Ord, Enum, Show)

data Student = Student
```

```
              { studentId :: Int
              , gradeLevel :: GradeLevel
              , studentName :: Name } deriving Show

students :: [Student]
students = [Student 1 Senior (Name "Audre" "Lorde")
           ,Student 2 Junior (Name "Leslie" "Silko")
           ,Student 3 Freshman (Name "Judith" "Butler")
           ,Student 4 Senior (Name "Guy" "Debord")
           ,Student 5 Sophmore (Name "Jean" "Baudrillard")
           ,Student 6 Junior (Name "Julia" "Kristeva")]


_select :: Monad m => (a -> b) -> m a -> m b
_select prop vals = do
    prop <$> vals

_where :: (Monad m, Alternative m) => (a -> Bool) -> m a -> m a
_where test vals = do
    val <- vals
    guard (test val)
    return val

startsWith :: Char -> String -> Bool
startsWith char string = char == head string

data Teacher = Teacher
             { teacherId :: Int
             , teacherName :: Name } deriving Show

teachers :: [Teacher]
teachers = [Teacher 100 (Name "Simone" "De Beauvior")
           ,Teacher 200 (Name "Susan" "Sontag")]

data Course = Course
            { courseId :: Int
            , courseTitle :: String
            , teacher :: Int} deriving Show

courses :: [Course]
courses = [Course 101 "French" 100
          ,Course 201 "English" 200]

_join :: (Monad m, Alternative m, Eq c) => m a -> m b -> (a -> c) -> (b -> c)
-> m (a, b)
_join data1 data2 prop1 prop2 = do
    d1 <- data1
    d2 <- data2
    let joinedPairs = (d1, d2)
    guard (prop1 (fst joinedPairs) == prop2 (snd joinedPairs))
    return joinedPairs

_hinq selectQuery joinQuery whereQuery = (selectQuery . whereQuery) joinQuery

finalResult :: [Name]
```

```
finalResult = _hinq (_select (teacherName . fst))
                    (_join teachers courses teacherId teacher)
                    (_where ((== "English") . courseTitle . snd))

teacherFirstName :: [String]
teacherFirstName = _hinq (_select firstName)
                         finalResult
                         (_where (const True))

data HINQ m a b =
     HINQ (m a -> m b) (m a) (m a -> m a)
   | HINQ_ (m a -> m b) (m a)

runHINQ :: (Monad m, Alternative m) => HINQ m a b -> m b
runHINQ (HINQ sClause jClause wClause) = _hinq sClause jClause wClause
runHINQ (HINQ_ sClause jClause) = _hinq sClause jClause (_where (const True))

query1 :: HINQ [] (Teacher, Course) Name
query1 = HINQ (_select (teacherName . fst))
              (_join teachers courses teacherId teacher)
              (_where ((== "English") . courseTitle . snd))

query2 :: HINQ [] Teacher Name
query2 = HINQ_ (_select teacherName) teachers

possibleTeacher :: Maybe Teacher
possibleTeacher = Just (head teachers)

possibleCourse :: Maybe Course
possibleCourse = Just (head courses)

maybeQuery1 :: HINQ Maybe (Teacher, Course) Name
maybeQuery1 = HINQ (_select (teacherName . fst))
                   (_join possibleTeacher possibleCourse teacherId teacher)
                   (_where ((== "French") . courseTitle . snd))

missingCourse :: Maybe Course
missingCourse = Nothing

maybeQuery2 :: HINQ Maybe (Teacher, Course) Name
maybeQuery2 = HINQ (_select (teacherName . fst))
                   (_join possibleTeacher missingCourse teacherId teacher)
                   (_where ((== "French") . courseTitle . snd))

data Enrollment = Enrollment
                { student :: Int
                , course :: Int } deriving Show

enrollments :: [Enrollment]
enrollments = [Enrollment 1 101
              ,Enrollment 2 101
              ,Enrollment 2 201
              ,Enrollment 3 101
              ,Enrollment 4 201
              ,Enrollment 4 101
```

```
              ,Enrollment 5 101
              ,Enrollment 6 201]

studentEnrollmentsQ = HINQ_ (_select (\(st, en) -> (studentName st, course
en)))
                            (_join students enrollments studentId student)

studentEnrollments :: [(Name, Int)]
studentEnrollments = runHINQ studentEnrollmentsQ

englishStudentsQ = HINQ (_select (fst . fst))
                        (_join studentEnrollments courses snd courseId)
                        (_where ((== "English") . courseTitle . snd))

englishStudents :: [Name]
englishStudents = runHINQ englishStudentsQ

getEnrollments :: String -> [Name]
getEnrollments courseName = runHINQ courseQuery
  where courseQuery = HINQ (_select (fst . fst))
                           (_join studentEnrollments courses snd courseId)
                           (_where ((== courseName) . courseTitle . snd))
```

Оценка времени, затраченного на выполнение

На выполнение заданий ушло около 10 часов, из которых примерно 3 часа было потрачено на реализацию задач по do-нотации и вводу-выводу, 1 час на работу с MARC-файлами, 2 часа на реализацию Functor, Applicative, Monad, 1 час на задачу с HINQ.

Оценка времени, затраченного на выполнение

Материал оказался понятным, основная сложность в итоговых проектах (особенно в задаче с работой с форматом MARC).

Выводы о наиболее сложных на текущей стадии аспектах языка

- Различие между Functor, Applicative и Monad остаётся теоретически тонким. Хотя все три класса типов позволяют применять функции к значениям в контексте, граница между ними - особенно между Applicative и Monad - не всегда очевидна на практике. Понимание того, что Applicative подходит для независимых вычислений, а Monad - для зависимых, приходит не сразу.

- Работа с эффектами ввода-вывода и обработка ошибок через типы вроде Maybe или Either - мощный инструмент, но он требует отказа от привычной модели исключений. Вместо того чтобы «поймать» ошибку, программа явно моделирует её как возможный результат, что делает код более предсказуемым, но увеличивает сложность адаптации к языку.

Аналогии с другими знакомыми средствами разработки программ

- fmap - то же, что .map() в JS или map() в Python, но применим к любому контексту: Maybe, спискам, IO.
- <*> - как Promise.all(): применяет несколько значений в контексте одновременно, но для любого типа, а не только промисов.
- >>= - как .then() в JS: цепочка зависимых операций, где результат одного влияет на следующий.

Вывод: в процессе выполнения лабораторной работы изучены контекстных вычислений.