МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 «Математика и компьютерные науки»

Отчет по курсовой работе
по дисциплине
«Функциональное программирование»
Вариант 5

Студент,
группы 5130201/20102 _____ Тищенко А. А.

Преподаватель,
к. т. н., доц. _____ Моторин Д. Е.

«_____»_____ 2024г.

Санкт-Петербург, 2024

Содержание

1 Постановка задачи

В рамках курсовой работы необходимо реализовать два синтаксических анализатора, решающих следующие задачи:

1. Разработка синтаксического анализатора для обработки строк из текстового файла. Требуется:
    - Читать строки, содержащие значения и бинарные операции, из текстового файла (.txt), название которого вводит пользователь.
    - Разбирать значения, представленные целыми числами в десятичной системе счисления.
    - Обрабатывать бинарные операции: сложение, вычитание, умножение, деление.
    - Вычислять результат выражения и выводить его на экран.
2. Разработка синтаксического анализатора текста и генератора продолжения текста. Задачи:
    (a) Считать текст из файла, название которого вводит пользователь, и выполнить его синтаксический анализ:
        - Разбить текст на предложения, используя следующие правила: слова состоят только из букв; предложения состоят из слов и разделяются символами: `. ! ? ; : ( )`.
        - Удалить из текста все символы пунктуации и цифры.

    (b) Построить модель N-грамм:
        - Использовать биграммы и триграммы.
        - Составить словарь, где ключами являются одно слово или пара слов, а значениями — списки всех уникальных возможных продолжений.
        - Сохранить словарь в текстовый файл (.txt).

    (c) Реализовать пользовательское взаимодействие:
        - При вводе одного или пары слов возвращать случайную строку длиной от 2 до 15 слов, основанную на созданном словаре.
        - Если введенное слово отсутствует в ключах словаря, выводить сообщение об этом.

    (d) Организовать диалог между двумя моделями N-грамм, созданными на основе двух различных текстов:
        - Пользователь задает начальное слово или пару слов и количество сообщений (глубину диалога).
        - Ответ каждой модели основывается на последнем слове (или предпоследнем, если последнее отсутствует в словаре) из предыдущего сообщения оппонента.

    В качестве текстов для построения моделей использовать произведения Антона Павловича Чехова.

2 Особенности реализации

Согласно заданию для каждой части работы был создан отдельный проект `stack`. Также все монадические вычисления были записаны без использования do-нотации, а лишь с помощью операторов `>>=` и `>>`. Все чистые функции были записаны в библиотеку `Lib.hs`, а доступ к вспомогательным функциям был ограничен.

2.1 Часть 1: Синтаксический анализ арифметических выражений

2.1.1 Тип Parser

Тип `Parser` обеспечивает разбор входной строки по заданным правилам. Он принимает на вход список токенов (например, символов) и пытается разобрать их в соответствии с описанными правилами, возвращая либо результат с оставшейся частью строки, либо `Nothing`, если разбор не удался.

Код типа `Parser` представлен в листинге 1.

- Вход: список токенов.
- Выход: результат разбора в виде `Maybe ([tok], a)`.

Для типа `Parser` определены представители классов типов для `Functor`, `Applicative` и `Alternative`. Представитель `Functor` позволяет применять функцию к результату разбора парсера. Представители `Applicative` и `Alternative` позволяют последовательно комбинировать разные парсеры и функции и составлять сложные парсеры из простых.

Листинг 1. Определение типа Parser и его представителей для классов типов Functor, Applicative и Alternative.

```haskell
newtype Parser tok a =
    Parser { runParser :: [tok] -> Maybe ([tok], a)}

instance Functor (Parser tok) where
    fmap g (Parser p) = Parser $ \xs ->
        case p xs of
            Nothing -> Nothing
            Just (cs, c) -> Just (cs, g c)

instance Applicative (Parser tok) where
    pure x = Parser $ \toks -> Just (toks, x)
    Parser u <*> Parser v = Parser $ \xs ->
        case u xs of
            Nothing -> Nothing
            Just (xs', g) ->
                case v xs' of
                    Nothing -> Nothing
                    Just (xs'', x) -> Just (xs'', g x)

instance Alternative (Parser tok) where
```

```
21      empty = Parser $ \_ -> Nothing
22      Parser u <|> Parser v = Parser $ \xs ->
23          case u xs of
24              Nothing -> v xs
25              z -> z
```

2.1.2 Работа с арифметическими операциями

В листинге 2 представлен код определения класса `Operation`, а также нескольких вспомогательных функций для работы с ним. Тип используется для хранения одной из четырёх арифметических операций: сложение, вычитание, умножение и деление. Функция `operationToString` принимает значение типа `Operation` и возвращает его строковое представление. Функция `operationToOperator` также принимает значение типа `Operation`, а возвращает функцию, соответствующую арифметической операции.

Листинг 2. Определение типа Operation и функций для работы с ним.

```
1  data Operation = Add | Sub | Mul | Div deriving Show
2
3  operationToString :: Operation -> String
4  operationToString op = case op of
5      Add -> "+"
6      Sub -> "-"
7      Mul -> "*"
8      Div -> "/"
9
10 operationToOperator :: Operation -> (Int -> Int -> Int)
11 operationToOperator op = case op of
12     Add -> (+)
13     Sub -> (-)
14     Mul -> (*)
15     Div -> div
```

2.1.3 Базовые парсеры

В этом разделе рассматриваются основные парсеры, используемые для разбора арифметических выражений. Эти парсеры являются строительными блоками для более сложных выражений. Их код представлен в листинге 3.

- `satisfy` — парсит символ, удовлетворяющий предикату, и возвращает его.
- `char` — парсит один заданный символ и возвращает его.
- `digit` — парсит одну цифру и возвращает в виде символа.
- `skipSpaces` — парсит все пробелы пока не встретить символ, который пробелом не является.
- `number` — парсит целое число (последовательность цифр) и возвращает в виде `Int`.

- `operation` — парсит арифметическую операцию (`+`, `-`, `*`, `/`) и возвращает как значение типа `Operation`.

Листинг 3. Базовые парсеры

```
satisfy :: (tok -> Bool) -> Parser tok tok
satisfy pr = Parser $ \case
    (c:cs) | pr c -> Just (cs, c)
    _             -> Nothing

char :: Char -> Parser Char Char
char c = satisfy (== c)

digit :: Parser Char Char
digit = satisfy isDigit

skipSpaces :: Parser Char String
skipSpaces = many (char ' ')

number :: Parser Char Int
number = skipSpaces *> (strToInt <$> some digit)
    where
    strToInt = foldl (\acc x -> acc * 10 + digitToInt x) 0

operation :: Parser Char Operation
operation = skipSpaces *> (
        char '+' *> pure Add <|>
        char '-' *> pure Sub <|>
        char '*' *> pure Mul <|>
        char '/' *> pure Div
    )
```

2.1.4 Парсер expression

Парсер `expression`, код которого представлен в листинге 4, парсит выражение вида `<число> <операция> <число>`. В случае успеха возвращает кортеж вида: `(Int - левый операнд, Operation - операция, Int - правый операнд)`. Является комбинацей парсеров `number` и `operation`. Не чувствителен к пробелам до выражения и внутри него, между операндами и оператором. Поглощает также пробелы после выражения с помощью парсера `skipSpaces`.

Листинг 4. Код функции expression

```
expression :: Parser Char (Int, Operation, Int)
expression = (,,) <$> number <*> operation <*> number <* skipSpaces
```

2.1.5 Функция processExpression

Код функции `processExpression` представлен в листинге 5. Функция принимает строку, парсит её как выражение, вычисляет результат и возвращает строку с ответом. При ошибке парсинга генерирует ошибку.

Вход: `String` — строка с выражением. Выход: `String` — результат вычисления в формате `a op b = result`.

Вспомогательная функция `calculateExpression` используется для вычисления результата. На вход она получает операнды и операцию, а возвращает вычисленное значение. Её код также представлен в листинге 5.

Листинг 5. Код функции processExpression

```
processExpression :: String -> String
processExpression s = case runParser expression s of
    Nothing -> error $ "Не удалось прочитать выражение: \"" ++ s ++ "\""
    Just (cs, (a, op, b)) -> case cs of
        [] -> show a ++ " " ++ operationToString op ++ " " ++
              show b ++ " = " ++ show (calculateExpression (a, op, b)) ++ "\n"
        _ -> error $ "Не удалось прочитать выражение: \"" ++ s ++ "\""


calculateExpression :: (Int, Operation, Int) -> Int
calculateExpression (a, op, b) = (operationToOperator op) a b
```

2.1.6 Функция main

Код функции `main` представлен в листинге 6. Функция `main` считывает имя файла у пользователя, читает файл, построчно обрабатывает каждое выражение с помощью `processExpression` и выводит результат.

Листинг 6. Код функции main

```
main :: IO ()
main =
    putStrLn "Введите имя файла:" >>
    getLine >>= \fileName ->
    readFile fileName >>= \content ->
    let expressions = lines content in
    putStrLn $ concatMap processExpression expressions

```

2.2 Часть 2: Синтаксический анализ текста и генерация фраз

2.2.1 Функция splitText

На первом этапе необходимо разделить текст на предложения и слова. Предложения определяются с помощью разделителей `.!?;:()`. В словах удаляются небуквенные символы и цифры. Код функции `splitText`, ответственной за разбиение текста и очистку слов, представлен в листинге 7. Функция принимает на вход строку – исходный текст, а возвращает список предложений, где каждое предложение представлено в виде списка слов.

Листинг 7. Функция splitText для разбора текста на предложения и слова

```haskell
splitText :: String -> [[String]]
splitText text = filter (not . null) $ map (processSentence . words) (splitSentences text)
  where
    splitSentences :: String -> [String]
    splitSentences [] = []
    splitSentences s =
        let (sentence, rest) = break isSeparator s
            rest' = dropWhile isSeparator rest
        in if null sentence
           then splitSentences rest'
           else sentence : splitSentences rest'

    isSeparator :: Char -> Bool
    isSeparator c = c `elem` ".!?;:()"

    processSentence :: [String] -> [String]
    processSentence = filter (not . null) . map cleanWord

    cleanWord :: String -> String
    cleanWord = map toLower . filter isLetter
```

2.2.2 Функция buildDictionary

На основе полученных предложений строится словарь, где ключами являются либо отдельные слова, либо пары слов, а значениями — списки возможных продолжений (следующее слово или пара слов для триграмм). Для этого используются биграммы и триграммы. Код функции `buildDictionary`, формирующей словарь представлен в листинге 8.

Листинг 8. Функция buildDictionary для формирования словаря N-грамм

```haskell
buildDictionary :: [[String]] -> Map String [String]
buildDictionary sentences =
    let bigrams   = [ (w1, w2) | s <- sentences, (w1:w2:_) <- tails s ]
        trigrams  = [ (w1, w2, w3) | s <- sentences, (w1:w2:w3:_) <- tails s ]
        singleKeys = foldr (\(w1, w2) acc -> Map.insertWith (++) w1 [w2] acc) Map.empty bigrams
        singleKeys' = foldr (\(w1, w2, w3) acc -> Map.insertWith (++) w1 [w2 ++ "␣" ++ w3] acc)
      singleKeys trigrams
        doubleKeys = foldr (\(w1, w2, w3) acc -> Map.insertWith (++) (w1 ++ "␣" ++ w2) [w3] acc)
       Map.empty trigrams
        combined = Map.unionWith (++) singleKeys' doubleKeys
    in Map.map nub combined
```

2.2.3 Функция saveDictionary

Функция `saveDictionary`, код которой представлен в листинге 9, сохраняет словарь с N-граммами в текстовый файл. Она принимает на вход путь до файла и сам словарь, явно ничего не возвращает, но перезаписывает содержимое файла. Для получения текстового представления списков вместо стандартной функции `show`, используется `ushow` из библиотеки `unescaping-print` [1]. `ushow` отображает кириллицу напрямую, без экранирования, в отличии от стандартной функции `show`.

Листинг 9. Функция saveDictionary для сохранения словаря N-грамм в файл.

```
saveDictionary :: FilePath -> Map String [String] -> IO ()
saveDictionary filePath dict = withFile filePath WriteMode $ \h ->
    mapM_ (\(k,v) -> hPutStrLn h $ ushow k ++ ":␣" ++ ushow v) (Map.toList dict)
```

2.2.4 Функция generatePhrase

Программа случайным образом формирует фразу длиной от 2 до 15 слов, используя словарь. На каждом шаге выбирается случайное продолжение, пока не будут исчерпаны возможные варианты или не достигнута заданная длина. Код функции для генерации фразы приведён в листинге 10.

Листинг 10. Функция generatePhrase для генерации фразы

```
generatePhrase :: Map String [String] -> String -> StdGen -> [String]
generatePhrase dict start initGenState =
    let (len, initGenState') = randomR (2,15 :: Int) initGenState
    in reverse $ gp start [] len initGenState'
    where
        gp :: String -> [String] -> Int -> StdGen -> [String]
        gp key acc n genState
            | n <= 0 = acc
            | otherwise =
                case Map.lookup key dict of
                    Nothing -> acc
                    Just [] -> acc
                    Just vals ->
                        let (i, newGenState) = randomR (0, length vals - 1) genState
                            next = vals !! i
                        in gp next (next:acc) (n - length (words next)) newGenState
```

2.2.5 Функция processInput

Функция `processInput`, код которой представлен в листинге 11, проверяет, существует ли введённое пользователем слово (или пара слов) в словаре, и генерирует фразу, используя функцию `generatePhrase`. Функция принимает на вход словарь и строку с пользовательским вводом. Явно ничего не возвращает, но выводит результаты в консоль.

Листинг 11. Функция processInput для обработки пользовательского ввода

```
processInput :: Map String [String] -> String -> IO ()
processInput dict input =
    if Map.member input dict then
        newStdGen >>= \gen ->
        putStrLn $ unwords $ generatePhrase dict input gen
    else
        putStrLn "Нет␣в␣словаре"
```

2.2.6 Функция twoModelsDialog

Функция `twoModelsDialog`, код которой представлен в листинге 12, симулирует диалог между двумя моделями N-грамм. Начальное слово или пару слов задаёт пользователь, затем модели по очереди генерируют ответы, основываясь на словах, из которых состоит последнее сообщение их собеседника. Функция `twoModelsDialog` принимает на вход два словаря N-грамм, строку с пользовательским вводом, в котором содержится стартовая N-грамма, и число – наибольшее количество сообщений от каждой модели.

Внутри `twoModelsDialog` используются две вспомогательные функции:

- `findKeyForResponse` – ищет в ответе собеседника слово, на которое модель способна дать ответ. Принимает на вход словарь N-грамм и список строк – предложение с ответом собеседника. Возвращает строку с подходящим словом, если его удалось найти. Поиск слов идёт с конца предложения собеседника.
- `dialogStep` – генерирует ответ с помощью функции `generatePhrase` на основе слова из предложения собеседника, найденного с помощью `findKeyForResponse`. Принимает на вход словарь N-грамм и список N-грамм – предложение собеседника. Если удалось сгенерировать ответ, то возвращает его в виде списка N-грамм.

Листинг 12. Функция twoModelsDialog для организации диалога между двумя моделями

```
1  twoModelsDialog :: Map String [String] -> Map String [String] -> String -> Int -> IO ()
2  twoModelsDialog dict1 dict2 start m =
3      newStdGen >>= \gen ->
4      let first = generatePhrase dict1 start gen
5      in putStrLn ("Модель 1: (" ++ start ++ ") " ++ unwords first) >>
6          loop dict1 dict2 first m
7      where
8          loop :: Map String [String] -> Map String [String] -> [String] -> Int -> IO ()
9          loop _ _ _ 0 = return ()
10         loop d1 d2 prev i =
11             putStr "Модель 2: " >>
12             dialogStep d2 prev >>= \resp ->
13             if null resp then return () else
14             putStr "Модель 1: " >>
15             dialogStep d1 resp >>= \resp2 ->
16             if null resp2 then return () else
17             loop d1 d2 resp2 (i-1)
18
19 findKeyForResponse :: Map String [String] -> [String] -> Maybe String
20 findKeyForResponse dict ws =
21     case dropWhile (\w -> Map.notMember w dict) (reverse ws) of
22         []    -> Nothing
23         (x:_) -> Just x
24
25 dialogStep :: Map String [String] -> [String] -> IO [String]
26 dialogStep dict prevPhrase =
27     case findKeyForResponse dict (words $ unwords prevPhrase) of
28         Nothing -> putStrLn "Нет в словаре" >> return []
```

```
29          Just key ->
30              newStdGen >>= \gen ->
31              let p = generatePhrase dict key gen
32              in putStrLn ("(" ++ key ++ ")␣" ++ unwords p) >> return p
```

2.2.7 Функция Main

Код функции `main` представлен в листинге 13. Функция `main` обрабатывает пользовательский ввод и с помощью функций `splitText` и `buildDictionary` строит две модели N-грамм на файлах указанных пользователем. Предлагает пользователю ввести слово или пару слов, на которые потом генерируется ответ с помощью функции `processInput`. Также запускает диалог между созданными моделями N-грамм с помощью функции `twoModelsDialog`. Словари с N-граммами моделей сохраняются в файлы `dict.txt` и `dict2.txt` с помощью функции `saveDictionary`.

Листинг 13. Код функции main

```
1  main :: IO ()
2  main =
3      putStrLn "Введите␣имя␣файла:" >>
4      getLine >>= \fileName ->
5      readFile fileName >>= \content ->
6      let sentences = splitText content in
7      let dict = buildDictionary sentences in
8      saveDictionary "dict.txt" dict >>
9      putStrLn "Введите␣слово␣или␣пару␣слов␣для␣генерации␣фразы:" >>
10     getLine >>= \input ->
11     processInput dict input >>
12
13     putStrLn "Введите␣имя␣второго␣файла:" >>
14     getLine >>= \fileName2 ->
15     readFile fileName2 >>= \content2 ->
16     let dict2 = buildDictionary (splitText content2) in
17     saveDictionary "dict2.txt" dict2 >>
18     putStrLn "Введите␣начальное␣слово␣или␣пару␣слов␣для␣старта␣диалога:" >>
19     getLine >>= \input2 ->
20     putStrLn "Введите␣количество␣сообщений␣от␣каждой␣модели:" >>
21     getLine >>= \ms ->
22     let m = read ms :: Int in
23     twoModelsDialog dict dict2 input2 m
```

3 Результаты работы программы

3.1 Часть 1: Синтаксический анализ арифметических выражений

Результаты работы программы представлены на Рис. 1. Программа предлагает пользователю ввести название файла, а затем выводит в консоль результаты разбора.

Если какую-то строку разобрать невозможно, то программа выведет ошибку, последующие строки анализироваться не будут. Пример такого сценария показан на Рис. 2. Программа также выводит в консоль строку, которую не удалось разобрать.

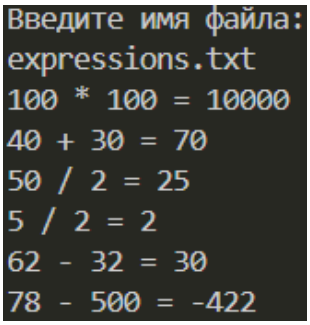

Рис. 1. Результат успешного разбора арифметических выражений.

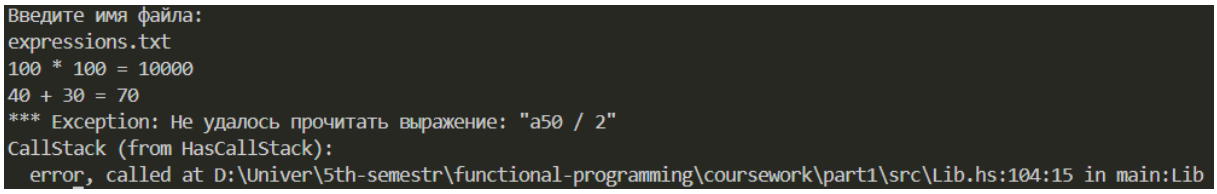

Рис. 2. Результат неудачного разбора арифметических выражений.

Пример содержимого файла `expressions.txt` представлен ниже:

```
100 * 100
40 + 30
50 / 2
5 / 2
62 - 32
78 - 500
```

3.2 Часть 2: Синтаксический анализ текста и генерация фраз

Результаты работы программы представлены на Рис. 3. Программа предлагает пользователю ввести имя файла с текстом, а потом слово или пару слов, на основе которой генерируется и выводится ответ. Затем пользователю предлагается ввести

имя ещё одного файла с текстом, задать начальное слово и размер диалога. После чего выводится диалог между полученными моделями, в скобках указано слово, с которого модель начала генерацию предложения.

```
Введите имя файла:
text1.txt
Введите слово или пару слов для генерации фразы:
ты
на стороне честных свободных людей и самому лгать улыбаться и
Введите имя второго файла:
text2.txt
Введите начальное слово или пару слов для старта диалога:
он
Введите количество сообщений от каждой модели:
3
Модель 1: (он) поднялся варенька узнала его
Модель 2: (его) следов но раньше ни звука а чемодан сел и стал тереться около его
Модель 1: (его) не было видно в потемках
Модель 2: (потемках) и в дрожании хвостика чувствовалось
Модель 1: (в) своей берлоге а может быть может для того что вотде в городе в
Модель 2: (в) страхе попятилась назад еще раз перебежала дорогу к тому же вся квартира битком
Модель 1: (же) этот человечек ходивший всегда в калошах и нам теперь казалось странным
```

Рис. 3. Результаты работы программы для синтаксического анализа текста и генерации фраз.

Если заданного слова нет в словаре, то программа выводит сообщение об этом. По этой же причине диалог между моделями может прерваться преждевременно, о чём также будет сообщено пользователю. Пример такого сценария показан на Рис. 4.

```
Введите имя файла:
text1.txt
Введите слово или пару слов для генерации фразы:
абракадабра
Нет в словаре
Введите имя второго файла:
text2.txt
Введите начальное слово или пару слов для старта диалога:
футляр
Введите количество сообщений от каждой модели:
5
Модель 1: (футляр) который уединил бы его варенька была недурна собой интересна она была пасмурная дождливая
Модель 2: (была) довольна своими длинными речами она пошла за ним со всех сторон послышались рев и
Модель 1: (и) за печью и всё
Модель 2: (всё) время знакомства
Модель 1: (знакомства) и все
Модель 2: (все) на перекладине этого деревянного грубо сколоченного п висел колокол то собаки
Модель 1: (собаки) заворчали
Модель 2: Нет в словаре
```

Рис. 4. Пример ситуации, в которой модель не может сгенерировать продолжение текста.

Файлы `text1.txt` и `text2.txt` содержат произведения Антона Павловича Чехова – «Человек в футляре» и «Каштанка». В первом тексте содержится 4146 слов, а во втором – 5963. Первый абзац текста из файла `text1.txt` представлен ниже:

```
На самом краю села Мироносицкого, в сарае старосты Прокофия расположились
на ночлег запоздавшие охотники. Их было только двое: ветеринарный врач
Иван Иваныч и учитель гимназии Буркин. У Ивана Иваныча была довольно
странная, двойная фамилия - Чимша-Гималайский, которая совсем не шла ему,
и его во всей губернии звали просто по имени и отчеству; он жил около
города на конском заводе и приехал теперь на охоту, чтобы подышать чистым
воздухом. Учитель же гимназии Буркин каждое лето гостил у графов П. и в
этой местности давно уже был своим человеком.
```

Программа сохраняет словари N-грамм в файлы `dict1.txt` и `dict2.txt`. В результате запуска программы на текстах, описанных выше, файл `dict1.txt` содержит 4511 строк, а файл `dict2.txt` – 6250. Каждая строка файла представляет собой ключ и значение из словаря. Ключ это одна из N-грамм, а значение это список N-грамм, которые являются возможными продолжениями текста. Первые десять строк содержимого файла `dict1.txt` представлены ниже:

```
"а в": ["последние"]
"а варенька": ["поет"]
"а возле": ["бродит"]
"а вот": ["подчинились"]
"а вы": ["оставайтесь"]
"а главное": ["это"]
"а держал": ["повара"]
"а дома": ["как"]
"а на": ["педагогических","хуторе"]
"а он": ["зеленый","только"]
```

Заключение

В рамках курсовой работы были разработаны и реализованы два `stack` проекта на языке Haskell, соответствующих поставленным задачам.

Первый проект представляет собой синтаксический анализатор для обработки строк, содержащих целые числа и бинарные операции. Реализация включала чтение данных из файла, синтаксический разбор выражений и вычисление их результатов. Код проекта состоит из 115 строк.

Второй проект — синтаксический анализатор текста и генератор продолжения текста, основанный на модели N-грамм. Проект, содержащий 139 строк кода, включает функционал по синтаксическому разбору текста, удалению пунктуации и цифр, построению словаря биграмм и триграмм, генерации текста, а также ведению диалога между моделями, созданными на основе двух разных текстов. Для демонстрации работы программы были взяты два произведения А. П. Чехова: «Человек в футляре» (4146 слов) и «Каштанка» (5963 слова).

В ходе работы были выполнены все поставленные задачи, а полученные знания могут быть использованы в других проектах на языке Haskell.

Список литературы

[1] Hackage – unescaping-print: Tiny package providing unescaping versions of show and print, URL: https://hackage.haskell.org/package/unescaping-print, Дата обращения: 09.12.2024.