МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

**"САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО"**

Институт компьютерных наук и технологий

Направление **02.03.01** : Математика и компьютерные науки

# ОТЧЕТ О ВЫПОЛНЕНИИ КУРСОВОЙ РАБОТЫ

Исполнитель: _____ Яшнова Дарья Михайловна
группа 5130201/20002

Руководитель: _____ Моторин Дмитрий Евгеньевич

«___»_____2024г

Санкт-Петербург, 2024

## Введение

Данная курсовая работа посвящена реализации парсера, который будет обрабатывать текстовые файлы, содержащие строки с бинарными операциями и значениями, а также реализации двух моделей на основе N-грамм для генерации текста.

Основные требования:

- Не использовать do-нотации для монадических вычислений.
- Доступ к вспомогательным функциям должен быть ограничен.

# Содержание

# 1 Постановка задачи

Задачами курсовой работы являются:

1. Разработка синтаксического анализатора (парсера), который будет обрабатывать текстовые файлы, содержащие строки с бинарными операциями и значениями. Парсер должен уметь вычислять выражения и выводить результат на экран.

2. Реализация синтаксического анализатора, который будет разбирать текст, прочитанный из файла, согласно заданным правилам. Текст должен быть разбит на предложения, из которых должны быть удалены все символы пунктуации и цифры.

3. Создание модели N-грамм (в данном случае триграмм) на основе разобранного текста. Модель должна быть сохранена в виде словаря, где ключами являются одно слово или пара слов, а значениями - список всех уникальных возможных продолжений триграммы.

4. Организация взаимодействия пользователя с двумя моделями N-грамм, созданными на основе разных текстов. Пользователь должен иметь возможность задавать начальное слово или пару слов для второй модели, а модели должны обмениваться сообщениями, основываясь на последнем слове из предложения оппонента.

Все чистые функции должны быть вынесены в Lib.hs.

# 2 Описание реализации

## 2.1 Часть 1

В первой части курсовой работы требуется разработать синтаксический анализатор, который будет обрабатывать текстовые файлы, содержащие строки с бинарными операциями и значениями. Парсер должен уметь вычислять выражения и выводить результат на экран.

Файл должен содержать:

- Значения: строки битов;
- Бинарные операции: сравнение $>$, $<$, $=$;

Пример текста, который должен обрабатываться парсером:

$101 = 101$ - Истина

$110 > 101$ - Истина

$011 < 100$ - Истина

$100 > 110$ - Ложь

*invalid line* - Неподходящая строка

Пользователь должен иметь возможность ввести имя файла самостоятельно.

### 2.1.1 Описание кода

**Функция checkStr** Код представляет собой функцию checkStr, которая проверяет корректность бинарного сравнения, представленную в виде строки.

```
1  checkStr :: String -> Maybe Bool
2  checkStr input = case runParser parseBinaryComparison input of
3    Just ("", result) -> Just result
4    _ -> Nothing
```

- input — это входная строка, которую мы хотим проверить.
- runParser parseBinaryComparison input — здесь вызывается функция runParser, которая принимает парсер parseBinaryComparison и входную строку input. Эта функция пытается разобрать строку с помощью указанного парсера.

- В данном случае используется конструкция case, чтобы обработать результат выполнения парсера.
- Если результат парсинга равен Just (, result), это означает, что парсер успешно разобрал всю строку (первый элемент — пустая строка, что указывает на то, что не осталось неразобранных символов) и возвращает значение result.
- В этом случае функция возвращает Just result, где result — это логическое значение, указывающее на успешность или неуспешность бинарного сравнения.

**Определение типа Parser** Обертка для парсера, который принимает список токенов (например, символов) и возвращает результат разбора или Nothing, если разбор не удался:

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

Для парсера реализованы следующие инстансы классов:

- Functor: позволяет применять функции к результатам парсера.
- Applicative: позволяет комбинировать парсеры.
- Monad: позволяет использовать последовательные операции разбора.
- Alternative: позволяет комбинировать парсеры с возможностью выбора между ними.

**Основные парсеры**

- satisfy: разбирает токен, если он соответствует предикату.
- char: проверяет, совпадает ли токен с заданным символом.
- spaces: разбирает пробелы.
- lexeme: обрабатывает токены и игнорирует пробелы после них.

**Парсеры для двоичных чисел и операций**

- binaryDigit: разбирает двоичные цифры (0 или 1).
- binaryNumber: разбирает последовательность двоичных цифр и преобразует их в целое число.
- comparisonOp: разбирает операторы сравнения (==, >, <).
- parseBinaryComparison: разбирает выражение вида "число оператор число"с игнорированием пробелов.

**Взаимодействие с пользователем** Функция libTaskMode в представленном коде выполняет несколько ключевых задач:

- Запрос имени файла:

  Функция начинает с запроса у пользователя имени файла, из которого будут считываться данные. Это позволяет динамически определять источник данных, что делает программу более гибкой.

- Чтение файла:

  После получения имени файла функция использует readFile для чтения содержимого указанного файла. Это позволяет загружать данные для последующей обработки.

- Обработка строк:

  Содержимое файла разбивается на строки с помощью функции lines. Каждая строка затем обрабатывается с помощью функции processLine, что позволяет выполнить проверку каждой строки на валидность и истинность.

- Вывод информации:

  Функция отвечает за вывод информации о процессе обработки. Она информирует пользователя о том, что происходит.

Основная задача функции processLine — принимать строку и выполнять над ней определенные операции. Это может включает в себя разбор данных, преобразование форматов.

## 2.2 Часть 2

### 2.2.1 N-граммы

N-граммы — это последовательности из N элементов (обычно слов или символов), которые используются в различных областях, таких как обработка естественного языка, статистика и машинное обучение.

N-грамма — это последовательность из N элементов, взятых из некоторого текста или последовательности. Например, в контексте обработки текста, если N=2, то это будут биграммы (пары слов), если N=3 — триграммы (тройки слов) и так далее.

Например, для фразы «I love programming»:

Биграммы: «i -> love», «love -> programming»

Триграммы: «i love -> programming»

В данной реализации требуется составить словарь биграмм и триграмм, сохранить этот словарь в файл. Далее требуется реализовать диалог двух моделей по двум разным текстам. В качестве текстов былы взяты отрывки из мемуаров Александра Ивановича Герцена. В text1.txt записана глава 2, а в text2.txt записана глава 3 из мемуаров Александра Ивановича Герцена [2]. Предложения разделяются знаками «.;:)(!?», Nграммы генерируются для каждого предложения отдельно.

### 2.2.2 Описание кода

**NGram.hs** В файле NGram.hs реализуется модуль для работы с n-граммами, который включает функции для генерации n-грамм из текста, генерации предложений на основе n-грамм, нормализации текста и сохранения n-грамм в файл.

1. Тип данных NGram:

   ```
   type NGram = Map.Map [String] [String]
   ```

   NGram определен как ассоциативный массив (словарь), где ключами являются списки строк (n-граммы), а значениями — списки строк, представляющие следующие слова, которые могут следовать за этими n-граммами.

2. Удаление знаков препинания:

```
removePunctuation :: String -> String
removePunctuation = filter (c -> isAlpha c || c == ' ')
```

Эта функция удаляет все символы, которые не являются буквами или пробелами.

3. Замена знаков препинания в конце предложений:

```
replaceEndPunctuations :: String -> String
```

Функция заменяет знаки конца предложения на точку.

4. Разбиение текста на предложения:

```
splitIntoSentences :: String -> [String]
```

Эта функция разбивает текст на предложения, используя точку как разделитель. Она также очищает каждое предложение от лишних пробелов и знаков препинания.

5. Нормализация текста:

```
normalizeText :: String -> [String]
```

Функция преобразует текст в нижний регистр и разбивает его на слова, заменяя все неалфавитные символы на пробелы.

6. Генерация n-грамм для одного предложения:

```
generateNGramsForSentence :: Int -> [String] -> NGram
```

Эта функция создает n-граммы для одного предложения, используя заданный порядок n. Она использует функцию tails, чтобы получить все возможные подсписки слов.

7. Генерация n-грамм из списка предложений:

```
generateNGrams :: Int -> [String] -> NGram
```

Эта функция объединяет n-граммы, сгенерированные для каждого предложения, в одну структуру данных NGram.

8. Генерация предложения на основе n-грамм:

```
generateSentence :: NGram -> [String] -> Int -> IO [String]
```

Эта функция генерирует предложение, начиная с заданного списка слов (seed). Она использует случайный выбор следующего слова на основе текущего состояния предложения и n-грамм.

9. Сохранение n-грамм в файл:

```
saveNGramsToFile :: FilePath -> NGram -> IO ()
```

Эта функция сохраняет n-граммы в текстовый файл. Каждая запись формируется в виде строки, где ключи и значения форматируются для удобства чтения.

**Dialog.hs** Модуль Dialog создает интерактивную систему диалога, в которой две модели могут обмениваться сообщениями на основе заданных текстов и n-грамм. Пользователь может ввести начальные слова, а система будет генерировать ответы, чередуя их между двумя моделями, пока не будет достигнуто заданное количество обменов или не произойдет зацикливание.

1. Функция startDialog:

```
startDialog :: Int -> Int -> [String] -> [String] -> IO ()
```

Параметры:

n: порядок n-грамм.

m: максимальное количество обменов сообщениями в диалоге.

text1 и text2: тексты, на основе которых будут созданы n-граммы для двух моделей.

Функция генерирует n-граммы для обоих текстов и запрашивает у пользователя начальные слова. Если пользователь не вводит слова, выбирается случайное начальное слово из первой модели. Затем начинается цикл диалога с помощью функции dialogLoop.

2. Функция dialogLoop:

```
dialogLoop :: Int -> Int -> NGram ->
NGram -> [String] -> [[String]] -> IO ()
```

Параметры:

m: оставшееся количество обменов сообщениями.

ngram1 и ngram2: n-граммы для двух моделей.

currentSeed: текущее семя для генерации следующего ответа.

history: история предыдущих ответов.

Функция генерирует ответ от первой модели и проверяет, не был ли он уже использован (чтобы избежать зацикливания). Если ответ уникален, генерируется ответ от второй модели, после чего цикл продолжается до тех пор, пока не будет достигнуто максимальное количество обменов.

3. Функция findNextSeed

```
findNextSeed :: NGram -> [String] -> Bool
```

Эта функция проверяет, существует ли следующее слово в n-граммах. Она принимает два параметра:

ngram: n-грамма.

nextSeed: слово, который нужно проверить.

4. Функция prevWord

```
prevWord :: [String] -> Int -> NGram -> [String]
```

Эта функция ищет предыдущее слово в ответе, которое может быть использовано как новое семя для следующей модели.

Она принимает три параметра:

response1: список слов, из которого нужно извлечь предыдущее слово.

i: индекс, с которого начинается поиск.

ngram: n-грамма для проверки наличия слова.

5. Функция generateResponse:

```
generateResponse :: NGram -> [String] -> [[String]] -> IO [String]
```

Эта функция отвечает за генерацию предложения на основе заданного семени и n-грамм. Она использует функцию generateSentence, которая, вероятно, генерирует предложение длиной до 10 слов.

**Main.hs и взаимодействие пользователей** Код функции main и связанные с ней функции, реализуют различные режимы работы программы. Этот код предоставляет пользователю возможность взаимодействовать с программой через консоль.

- Функция main:

```
main :: IO ()
```

Эта функция является точкой входа в программу. Она предлагает пользователю выбрать один из четырех режимов работы.

В зависимости от выбранного режима вызывается соответствующая функция:

«1»: Генерация предложения.

«2»: Сохранение n-грамм в файл.

«3»: Запуск диалога между двумя моделями.

«4»: Выполнение задачи из модуля Lib.hs.

Если пользователь вводит некорректный вариант, программа выводит сообщение об ошибке и завершает работу.

- Функция generateSentenceMode:

```
generateSentenceMode :: IO ()
```

Пользователь вводит имя файла, содержащего текст для генерации n-грамм.

Текст считывается и разбивается на предложения с помощью функции splitIntoSentences.

Затем создаются n-граммы с помощью функции generateNGrams.

Пользователь вводит начальные слова, которые будут использованы для генерации предложения.

С помощью функции generateSentence генерируется предложение длиной до 15 слов, и результат выводится на экран.

- Функция saveNGramsMode:

```
saveNGramsMode :: IO ()
```

Эта функция позволяет пользователю сохранить n-граммы в файл. Пользователь вводит имя входного файла для чтения текста и имя выходного файла для сохранения n-грамм. Текст считывается, разбивается на предложения, и создаются n-граммы с помощью generateNGrams. Затем n-граммы сохраняются в указанный файл с помощью функции saveNGramsToFile, и программа уведомляет пользователя о завершении операции.

- Функция startDialogMode:

```
startDialogMode :: IO ()
```

Эта функция запускает диалог между двумя моделями, используя два текста.

# 3 Результаты

На рис.1 представлены результаты разбора файла с булевыми высказываниями.

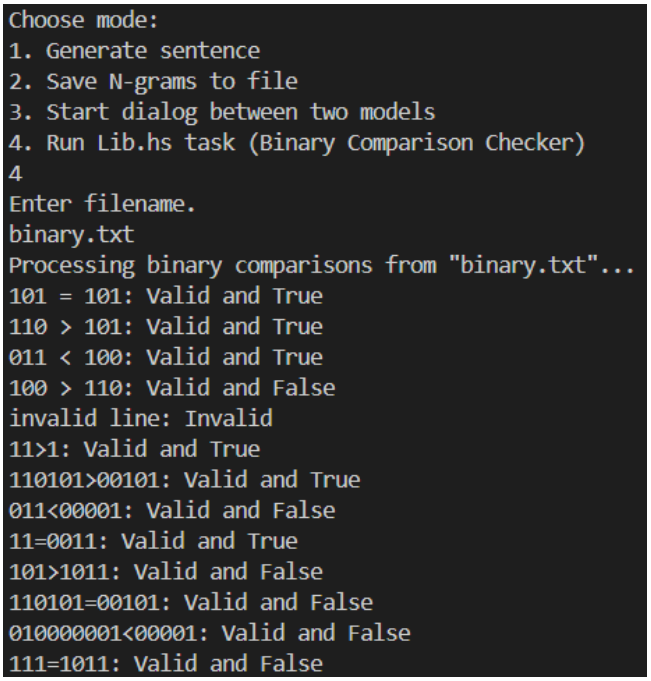


Рис. 1: Результат разбора файла с бинарными высказываниями

При разборе определяется истинно высказывание или ложно и подходит оно для разбора или нет.

На рис.2 представлен вывод для режима сохранения словаря в файл.

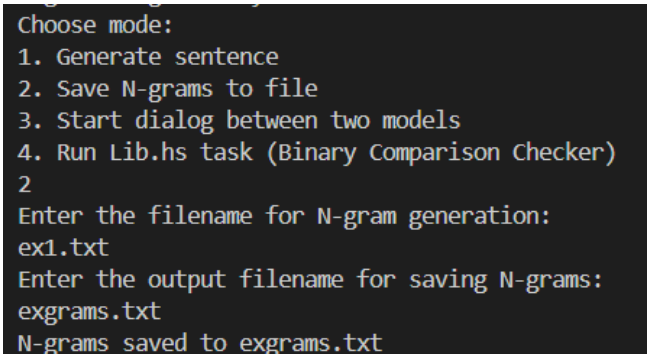


Рис. 2: Вывод для режима сохранения словаря в файл

Далее представлен текст из файлов text.txt и exgrams.txt.

**text.txt**

```
a b, c d e! b c d & e b c # a ^ d. a f; f.
```

**exgrams.txt**

```
a -> 'b' 'b c' 'd' 'f'
a b -> 'c'
b -> 'c' 'c a' 'c d'
b c -> 'a' 'd'
```

```
c -> 'a' 'a d' 'd' 'd e'
c a -> 'd'
c d -> 'e'
d -> 'e' 'e b'
d e -> 'b'
e -> 'b' 'b c'
e b -> 'c'
```

На рис.3 представлен результат генерации предложения по заданному слову.

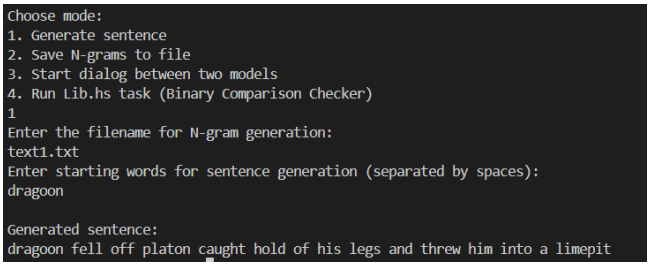


Рис. 3: Результат генерации предложения по заданному слову

На рис.4 представлен результат работы режима генерации разговора между двумя моделями.

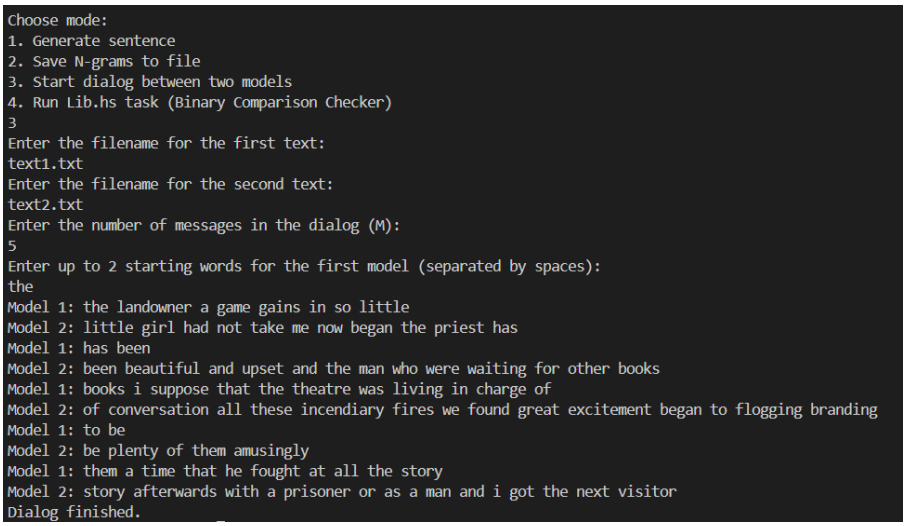


Рис. 4: Результат работы режима генерации разговора между двумя моделями

Можно видеть, что если модель не находит слово в своем словаре, она производит поиск по предыдущему слову в предложении.

# Выводы

В ходе выполнения курсовой работы была разработана система, состоящая из нескольких взаимосвязанных компонентов, каждый из которых выполняет свою уникальную функцию.

Успешно реализован парсер, который обрабатывает текстовые файлы с бинарными операциями и значениями. Парсер способен вычислять выражения, поддерживая бинарные операции, и выводить результаты на экран.

Реализован анализатор, который разбивает текст на предложения, удаляя все символы пунктуации и цифры. Разработана модель n-грамм, основанная на разобранном тексте. Модель сохраняется в виде словаря, где ключами выступают одно слово или пара слов, а значениями — списки всех уникальных возможных продолжений триграммы.

Организовано взаимодействие между двумя моделями N-грамм, созданными на основе разных текстов. Пользователь может задавать начальное слово или пару слов для модели. Модели обмениваются сообщениями, основываясь на последнем слове из предложения оппонента.

# Список литературы

1. W. Kurt. Get Programming with Haskell. Москва: ДМК, 2019.
2. A. Herzen. Memoirs of Alexander Herzen. 1923, YALE UNIVERSITY PRESS. URL: https://www.gutenberg.org/files/67882/67882-h/67882-h.htm