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»

Монады в Haskell

Курсовая работа

Студент,
группы 5130201/20101 _____ Астафьев И. Е.

Преподаватель _____ Моторин Д. Е.

«_____»_____ 2024г.

Санкт-Петербург, 2024

Содержание

Введение

В данном отчете описаны результаты выполнения курсовой работы: монады в Haskell.

Часть 1

Постановка задачи:

1. Написать синтаксический анализатор (парсер), разбирающий строки, прочитанного файла .txt. Файл должен содержать значения и бинарные операции.

    - Значения: строки битов
    - Бинарные операции: AND(&), OR(|), XOR($\oplus$)
    - Пример строки в файле 01010 & 0011. Вычислить проанализированное выражение результат вычисления на экран. Пользователь вводит название файла.

Часть 2

Постановка задачи:

1. Прочитать текст из файла, указываемого пользователем. Синтаксически проанализировать текст согласно правилам: слова состоят только из букв; предложения состоят только из слов и разделены символами: !?;:. Разбить текст на предложения. Удалить все символы пунктуации и цифры из слов и предложений.

2. Составить модель N-грамм. Использовать модель биграмм и триграмм. По списку предложений составить словарь. Ключами являются: одно слово, либо пара слов. Значениями в словаре является список всех уникальных возможных продолжений триграммы (т.е. список пар слов или одиночных слов). Словарь сохранить в файл .txt.

    Пример текста и словаря на его основе:

    Текст: [a b, c d e! b c d? e b c # a d. a f; f.]

    Словарь: ['a' : ['b', 'd', 'f', 'b c']; 'b' : ['c', 'c d', 'c a']; 'c' : ['d', 'a', 'd e', 'a d']; 'd' : ['e']; 'e' : ['b','b c']; f : []; 'a b' : ['c']; 'b c' : ['d', 'a']; 'c d' : ['e']; 'e b' : ['c']; 'c a' : ['d']; 'a d' : []; 'a f' : []; 'd e' : [].

3. Реализовать взаимодействие с пользователем. Пользователь вводит одно слово или пару слов. Программа возвращает стоку случайной длины в диапазоне от 2 до 15 слов, если задаваемого пользователем слова нет в ключах словаря, выдавать соответствующее сообщение. Фраза составляется путем добавления случайного слова (или пары) из списка значений текущего слова-ключа (или пары-ключа), до тех пор, пока либо не будет сформировано предложение нужной длины, либо не будет достигнут ключ, у которого нет значений.

4. Организовать диалог двух моделей N-грамм созданных на двух разных текстах. Тексты для второй модели выбрать самостоятельно. Пользователь задает начальное слово (или пару) и глубину M сообщений, которыми обмениваются модели. Ответ модели основывается на последнем слове из предложения оппонента (если последнее слово отсутствует в словаре, то предпоследнее и т.д. пока не будет найдено подходящее слово или не закончится предложение оппонента).

Автор текстов: Максим Горький.

1 Теоретические сведения

1.1 Синтаксический анализатор

Синтаксический анализатор, или парсер, является ключевым компонентом любой системы обработки текста, языков программирования или данных. Его задача заключается в разборе входной последовательности символов (или токенов, если предварительно был выполнен лексический анализ) и проверке её соответствия заданной грамматике, а также в построении структуры данных, описывающей синтаксис входной строки.

1.2 Основные этапы работы парсера

Работа синтаксического анализатора включает следующие этапы:

1. **Получение входных данных.** Парсер принимает последовательность символов или токенов на вход. Эти данные поступают либо напрямую из исходного текста, либо из лексического анализатора.
2. **Проверка соответствия грамматике.** Парсер проверяет, соответствует ли входная строка правилам грамматики, заданным в виде контекстно-свободной грамматики (КС-грамматики) или другого формализма.
3. **Построение выходной структуры.** Если строка соответствует грамматике, парсер формирует дерево разбора (дерево синтаксического анализа), абстрактное синтаксическое дерево (AST) или другую структуру данных, которая будет использоваться для последующей обработки.

Типы синтаксических анализаторов Существуют различные типы синтаксических анализаторов, которые различаются по методам работы и применяемым алгоритмам:

- **Восходящие парсеры (Bottom-up).** Эти парсеры строят дерево разбора, начиная с листьев (токенов) и продвигаясь вверх к корню. Примеры: алгоритмы LR, LALR и SLR.
- **Нисходящие парсеры (Top-down).** Такие парсеры начинают с корня дерева и рекурсивно обрабатывают правила грамматики, продвигаясь к листьям. Примером является рекурсивно-спускающийся парсер.
- **LL-парсеры.** Это подмножество нисходящих парсеров, которые обрабатывают грамматику слева направо (Left-to-right) и строят левую производную (Leftmost derivation).
- **Рекурсивно-спускающиеся парсеры.** Это парсеры, которые используют рекурсивные вызовы функций для обработки правил грамматики. Они просты в реализации и часто используются для разработки парсеров на языках программирования.

1.3 Функторы

Функторы — это абстракция, представляющая контейнеры, к элементам которых можно применять функции. Они принадлежат классу типов `Functor`, который определяет операцию `fmap`. Эта операция позволяет трансформировать содержимое контейнера, не изменяя его структуру. Например, с помощью `fmap` можно применить функцию ко всем элементам списка.

1.4 Аппликативные функторы

Аппликативные функторы расширяют возможности функторов, добавляя способ применения функций, заключённых в контексте, к значениям в том же контексте. Это реализуется через операции `<*>` и `pure` из класса типов `Applicative`. Операция `pure` помещает значение в контекст, а `<*>` применяет функцию из контекста к значениям, также находящимся в контексте. Аппликативные функторы полезны для работы с вычислениями, которые могут завершиться неудачей.

1.5 Класс типов `Alternative`

Класс типов `Alternative` используется для работы с контейнерами, которые могут содержать либо значения, либо быть пустыми. Основные операции:

- `empty` — представляет пустой контейнер.
- `<|>` — предоставляет способ комбинирования двух контейнеров. Например, выбирает первый непустой контейнер.

`Alternative` часто используется в парсерах, где можно выбирать между различными вариантами анализа текста.

1.6 Монады

Монады — это обобщение аппликативных функторов, позволяющее связывать последовательные вычисления. Они определяются через операции `>>=` (bind) и `return`. Операция `return` помещает значение в контекст, а `>>=` позволяет передавать результат одного вычисления следующему.

Монады обеспечивают удобство работы с вычислениями, которые могут быть последовательными, контекстуальными или содержать побочные эффекты. Они находят широкое применение в обработке ошибок, работе с вводом-выводом, парсерах и других задачах.

1.7 N-граммы

N-граммы — это последовательности из N элементов, получаемые из данных. В контексте обработки естественного языка (NLP) элементы обычно представляют собой слова или символы. Основной задачей N-грамм является моделирование и анализ последовательностей в тексте, что может быть полезно для различных задач, таких как предсказание следующего слова, анализ вероятности перехода от одного состояния к другому или классификация текста.

Определение: N-грамма — это последовательность из N элементов, где каждый элемент может быть как словом, так и символом, в зависимости от задачи. Если N = 1, то это называется *унитарной граммой* или *униграммой*, если N = 2 — *биграммой*, N = 3 — *триграммой* и так далее. Например:

- Униграмма для текста "Я люблю читать": ["Я "люблю "читать"].
- Биграмма для того же текста: ["Я люблю "люблю читать"].
- Триграмма: ["Я люблю читать"].

Применение N-грамм в обработке естественного языка:

- *Языковое моделирование:* N-граммные модели широко используются для оценки вероятности появления слова в определенном контексте. Например, биграммная модель будет оценивать вероятность появления слова на основе предыдущего слова.

- *Предсказание текста:* Используя вероятности N-грамм, можно предсказать следующее слово в предложении. Если модель обучена на больших корпусах текста, она может точно угадывать следующее слово, основываясь на предыдущих N-1 словах.

- *Классификация текста:* N-граммные признаки могут использоваться для представления текста в задачах классификации, таких как определение тональности текста или его тематической принадлежности.

В некоторых задачах моделирования языка или предсказания текста, для представления N-грамм удобно использовать словарь, где ключами являются N-граммы (представленные в виде строк), а значениями — последовательности элементов, которые могут следовать за данным контекстом.

Пример структуры такого словаря для биграмм может выглядеть следующим образом:

```
"a": ["b "d "b c"], "b c": ["f"], "f": []
```

Здесь:

- Ключ "a"ассоциируется с массивом значений `["b "d "b c"]`, что означает, что после слова "a"могут следовать слова "b "d"или "b c".

- Ключ "b c"ассоциируется с массивом `["f"]`, что означает, что после биграммы "b c"может следовать слово "f".

- Ключ "f"имеет пустой массив `[]`, что означает, что после слова "f"не ожидаются другие слова или элементы.

Этот словарь может использоваться в различных задачах обработки текста, таких как генерация текста или анализ вероятности появления слов. Основные преимущества этого подхода заключаются в том, что такой словарь позволяет:

- **Хранить контексты** для каждой N-граммы, то есть, для каждого слова или последовательности слов хранить информацию о том, какие элементы могут следовать за ними.

- **Быстро искать возможные продолжения** для каждой N-граммы, что полезно при решении задач предсказания следующего слова или анализа текста.

- **Использовать для предсказания вероятности**, вычисляя вероятность появления следующего слова как частоту появления той или иной N-граммы в обучающем корпусе текста.

2 Часть 1. Парсер строк бинарных чисел и операций

2.1 Реализация программы

Полный исходный код представлен в Приложение А. Парсер бинарных операций.

Описание Parser

В данном разделе представлен синтаксический анализатор (парсер), реализованный на языке Haskell. Его цель — поэтапный разбор текста с возможностью обработки ошибок и комбинирования простых парсеров в более сложные конструкции.

Парсер реализован как абстракция, представляющая собой функцию, которая принимает текст на вход и возвращает результат анализа в виде оставшегося текста и результата парсинга. В случае неудачного анализа результатом работы является специальное значение `Nothing`.

Основные возможности парсера обеспечиваются реализацией нескольких стандартных классов Haskell:

- Класс `Functor` позволяет применять функции к результату парсинга, не изменяя процесс самого анализа. Это достигается за счёт обёртки результата существующего парсера в новый парсер, который трансформирует результат с помощью переданной функции.
- Класс `Applicative` позволяет комбинировать несколько парсеров, применяя функции с несколькими аргументами к их результатам. Этот механизм используется для построения последовательных цепочек парсеров. Например, можно объединить парсеры для разбора чисел и операторов в одном выражении.
- Класс `Alternative` предоставляет средства для обработки альтернативных вариантов и восстановления после ошибок. С его помощью можно определять парсеры, которые пробуют несколько вариантов анализа и выбирают первый успешный результат.

Основной особенностью реализации является использование концепций функционального программирования, таких как иммутабельность данных и композиция функций, что делает код простым для комбинирования и тестирования. Такой подход позволяет строить мощные анализаторы для обработки текста с минимальными усилиями.

Разработанный парсер служит основой для построения высокоуровневых разборщиков, таких как парсер бинарных выражений, представленный в рамках данной работы.

Описание функций проверки и парсинга символов

Для реализации базовых операций парсинга в коде используются следующие функции:

- `isBinaryChar`

  Данная функция проверяет, является ли символ допустимым двоичным символом (`'0'` или `'1'`). Она используется в парсерах, которые разбирают двоичные числа.

- `isOperation`

  Эта функция проверяет, относится ли символ к допустимым логическим операциям: `'&'` (AND), `'|'` (OR) или `'⊕'` (XOR). Она помогает распознать оператор в выражении.

- `isSpace`

  Проверяет, является ли символ пробелом (`' '`). Функция используется для обработки пробелов между элементами выражения.

- `satisfy`

  Универсальная функция-парсер, которая принимает предикат (функцию `Char -> Bool`) и создаёт парсер, возвращающий первый символ текста, удовлетворяющий предикату. Если текст пустой или первый символ не подходит под предикат, парсер возвращает `Nothing`.

  Алгоритм работы:

  1. Сначала извлекается первый символ строки с помощью функции `T.uncons`.
  2. Если строка пуста (`Nothing`), парсер возвращает `Nothing`.
  3. Если символ присутствует, проверяется выполнение предиката.
     - При успешной проверке возвращается пара: оставшийся текст и символ.
     - При неудаче возвращается `Nothing`

Простые парсеры

В данном разделе описаны базовые парсеры, которые используются для распознавания и обработки текстовых данных, таких как двоичные числа, логические операции и пробелы. Эти парсеры являются ключевыми элементами для построения более сложных синтаксических анализаторов.

- **Парсер двоичных чисел.**

  Этот парсер считывает последовательность двоичных символов (`0` или `1`) из входного текста. Если первый символ строки является двоичным, он добавляется к результату. Парсер рекурсивно продолжает обработку оставшейся строки.

  На выходе:

  - Если парсинг завершился успешно, возвращается `Just (remainingText, parsedText)`, где `remainingText` — остаток строки после обработки, а `parsedText` — строка, состоящая из всех двоичных символов, считанных подряд.
  - Если первый символ не является двоичным или строка пустая, возвращается `Nothing`.

- **Парсер операций.**

  Этот парсер использует функцию `satisfy`, чтобы найти первый символ, который соответствует одному из предопределённых операторов (`&`, `|` или `?`). Функция `isOperation` проверяет, является ли символ одним из допустимых операторов. Если символ подходит, он возвращается как результат парсера.

  На выходе:

- Если первый символ является одним из допустимых операторов, парсер возвращает `Just (remainingText, op)`, где `remainingText` — остаток строки после обработки, а `op` — сам символ оператора.
- Если первый символ не является допустимым оператором, парсер возвращает `Nothing`.

- **Парсер одиночного пробела.**

  Этот парсер использует функцию `satisfy`, чтобы найти первый символ, который является пробелом. Функция `isSpace` проверяет, является ли символ пробелом. Если символ подходит, он возвращается как результат парсера.

  На выходе:

  - Если первый символ — пробел, парсер возвращает `Just (remainingText, ' ')`, где `remainingText` — остаток строки после обработки, а `' '` — символ пробела.
  - Если первый символ не является пробелом, парсер возвращает `Nothing`.

- **Парсер последовательности пробелов.**

  Этот парсер использует комбинацию парсеров для обработки пробелов. Он рекурсивно использует `oneSpace` для поиска первого пробела, затем применяет себя (парсер `spaces`) для обработки оставшихся пробелов. В случае, если пробелы больше не найдены, используется `pure T.empty`, чтобы вернуть пустую строку.

  На выходе:

  - Если строка начинается с пробела, парсер возвращает строку из пробелов. Например, `Just (remainingText, )` в случае одного пробела или `Just (remainingText, )` в случае двух пробелов.
  - Если строка не содержит пробелов или они закончились, парсер возвращает `Just (remainingText, )`, то есть пустую строку.
  - Если первый символ не является пробелом, парсер возвращает `Nothing`.

Обработка логических операций и форматирование выражений

Данный раздел описывает функции, которые выполняют вычисление логических операций над двоичными числами и форматируют выражения для удобного представления результата.

- **Функция для применения операций.** Функция `applyOperation` принимает символ, обозначающий логическую операцию (`&`, `|`, `^`), и возвращает соответствующую функцию, которая применяется к двум целым числам.

  - Оператор `&` интерпретируется как побитовое *AND*.
  - Оператор `|` интерпретируется как побитовое *OR*.
  - Оператор `^` интерпретируется как побитовое *XOR*.

  Этот подход обеспечивает компактный и удобный способ выбора операции в зависимости от символа.

- **Парсер для выражений с двоичными числами.**

  Этот парсер представляет собой комбинацию нескольких простых парсеров, который парсит строку, содержащую два бинарных числа и операцию между ними, игнорируя пробелы. Он использует парсер `spaces` для обработки возможных пробелов в начале, середине и конце выражения.

  Сначала парсится первое бинарное число с помощью парсера `binary`, который распознаёт последовательности символов `'0'` и `'1'`. Затем, между ними парсится операция (например, `&`, `|` или `^`) с помощью парсера `operation`. После этого снова пропускаются пробелы с помощью `spaces`. В завершение, парсится второе бинарное число с использованием того же парсера `binary`.

  Вся эта структура композиционно собрана с помощью оператора `<*>` из типа `Applicative`, что позволяет комбинировать несколько парсеров, обеспечивая удобную работу с каждым компонентом выражения. Полученные результаты передаются в функцию `formatExpression`, которая форматирует их в строку, включая результат вычисления операции.

- **Форматирование и вычисление выражения.** Функция `formatExpression` преобразует разобранные элементы выражения (два числа и операцию) в итоговый текст, включающий результат вычисления. Алгоритм:

  – Преобразует оба двоичных числа в целые числа.

  – Применяет указанную операцию к этим числам с помощью функции `applyOperation`.

  – Преобразует результат обратно в двоичный формат.

  – Формирует строку вида `<число1> <операция> <число2> = <результат>`.

  Пример: строка `"101 & 010"` будет преобразована в `"101 & 010 = 0"`.

Эти функции обеспечивают корректную обработку двоичных выражений, их вычисление и представление результата в удобочитаемой форме, что делает парсер удобным и эффективным инструментом для анализа логических выражений.

3 Часть 2. Генератор предложений с помощью N-грамм

3.1 Реализация программы

Полный исходный код представлен в Приложение Б. Генератор предложений.

Описание Parser

Описание Parser идентично описанному в разделе Описание Parser в Части 1.

Описание функций проверки и парсинга символов

Для реализации базовых операций обработки текста в коде используются следующие функции:

- `isPunctuation`

  Эта функция проверяет, является ли символ знаком препинания. Она возвращает `True`, если символ входит в список `['.', '!', '?', ';', ':', '(', ')']`, и `False` в противном случае. Эта функция используется для определения наличия знаков препинания в строках, что может быть полезно при анализе текста или парсинге.

- `joinWords`

  Функция принимает список строк `[Text]` и объединяет их в одну строку, вставляя между словами пробелы. Для этого используется функция `T.intercalate`, которая вставляет заданный разделитель (в данном случае пробел) между элементами списка и возвращает единую строку. Эта функция полезна для объединения слов или фраз в одно предложение или текст.

- `isWordChar`

  Эта функция проверяет, является ли символ частью слова. Она возвращает `True`, если символ является буквой (проверяется с помощью функции `isLetter`) или если символ — это апостроф (`'`). Таким образом, эта функция может быть полезна для распознавания частей слов, например, для обработки таких случаев, как апострофы в английских словах (например, `"don't"` или `"it's"`).

- `satisfy`

  Эта функция принимает предикат `pr` (функцию, проверяющую условие для символа) и возвращает парсер, который пытается применить этот предикат к первому символу строки. Если первый символ удовлетворяет предикату, парсер возвращает оставшуюся строку и сам символ, иначе возвращает `Nothing`. Функция полезна для парсинга отдельных символов, удовлетворяющих заданному условию, и может быть использована в различных парсерах для обработки символов, таких как цифры, пробелы или буквы.

Простые парсеры

В данном разделе описаны парсеры, которые используются для распознавания слов, пробелов, пунктуации и игнорирования ненужных символов. Эти парсеры являются важными для обработки текста и составляют основу для более сложных анализаторов.

- **Парсер слов.**

  Этот парсер использует `some` и `satisfy`, чтобы считывать последовательность символов, удовлетворяющих предикату `isWordChar`. Он обрабатывает символы, которые могут быть буквами или апострофами. После того как слово собрано, оно очищается от апострофов с обеих сторон с помощью `T.dropAround`. Если слово состоит только из апострофов, парсер возвращает `Nothing`.

  На выходе:

  - Если слово успешно разобрано, парсер возвращает `Just (remainingText, word)`, где `remainingText` — остаток строки после обработки, а `word` — разобранное слово.
  - Если слово состоит только из апострофов, парсер возвращает `Nothing`.

- **Парсер одиночного пробела.**

  Этот парсер использует функцию `satisfy`, чтобы найти первый символ, который является пробелом. Он использует предикат `isSpace` для проверки, является ли символ пробелом. Если символ является пробелом, он возвращается как результат парсера.

  На выходе:

  - Если первый символ — пробел, парсер возвращает `Just (remainingText, ' ')`, где `remainingText` — остаток строки после обработки, а `' '` — символ пробела.
  - Если первый символ не является пробелом, парсер возвращает `Nothing`.

- **Парсер последовательности пробелов.**

  Этот парсер использует комбинацию парсеров для обработки одного или нескольких пробелов. Он рекурсивно вызывает `oneSpace`, чтобы найти первый пробел, а затем повторно использует себя (`spaces`), чтобы обработать оставшиеся пробелы. Если пробелы закончились, используется `pure T.empty`, чтобы вернуть пустую строку.

  На выходе:

  - Если строка содержит пробелы, парсер возвращает строку из пробелов, например, `Just (remainingText, )` или `Just (remainingText, )` в случае нескольких пробелов.
  - Если строка не содержит пробелов, парсер возвращает `Just (remainingText, )`, то есть пустую строку.
  - Если первый символ не является пробелом, парсер возвращает `Nothing`.

- **Парсер пунктуации.**

  Этот парсер использует `satisfy`, чтобы найти первый символ, который является одним из символов пунктуации. Для этого используется предикат `isPunctuation`, который проверяет, является ли символ одним из символов пунктуации, таких как `'.'`, `'!'`, `'?'`, `';'`, `':'`, `'('`, `')'`.

  На выходе:

  - Если первый символ является символом пунктуации, парсер возвращает `Just (remainingText, punctuation)`, где `remainingText` — остаток строки после обработки, а `punctuation` — сам символ пунктуации.

- Если первый символ не является символом пунктуации, парсер возвращает `Nothing`.

- **Парсер для пропуска ненужных символов.**

  Этот парсер использует `T.dropWhile` для пропуска всех символов, которые не являются буквами или символами пунктуации. Он продолжает обработку строки до тех пор, пока не встретит допустимый символ. После этого возвращает остаток строки. Этот парсер полезен для игнорирования символов, которые не имеют значения для анализа (например, пробелы или другие разделители).

  На выходе:

  - Если строка содержит ненужные символы, которые можно пропустить, парсер возвращает остаток строки, начиная с первого допустимого символа. Результат парсинга — `()`.
  - Если строка не содержит допустимых символов, парсер возвращает `Nothing`.

Разбиение текста на предложения

Данный раздел описывает парсеры, которые анализируют текст, извлекая из него предложения, а затем преобразуют их в различные форматы для дальнейшей обработки.

- **Парсер для предложения.**

  Этот парсер разбивает текст на слова и пунктуацию, создавая список слов, которые составляют предложение. Он использует комбинацию парсеров `word` для получения слов и `punctuation` для обработки знаков препинания. Для обработки ненужных символов применяется парсер `skipJunk`.

  Парсер работает следующим образом:

  - Сначала он парсит одно или несколько слов, используя парсер `word`, каждый из которых может быть отделён ненужными символами, которые пропускаются с помощью `skipJunk`.
  - Затем он ожидает один или несколько символов пунктуации (например, точку, восклицательный знак, вопросительный знак).
  - После этого снова пропускаются лишние символы с помощью `skipJunk`.
  - Если после этих шагов не осталось текста, возвращается пустой результат (используется `empty`).

  На выходе:

  - Если предложение успешно разобрано, парсер возвращает `Just (remainingText, words)`, где `remainingText` — остаток строки, а `words` — список слов.
  - Если не удалось распарсить предложение, возвращается `Nothing`.

- **Парсер для всех предложений.**

  Этот парсер использует `some`, чтобы рекурсивно разобрать несколько предложений с помощью парсера `sentence`. Он находит все предложения в тексте, разделённые символами, и возвращает их как список предложений.

  На выходе:

- Если предложения успешно разобраны, возвращается список всех предложений: `Just (remainingText, sentences)`.
- Если не удалось распарсить предложения, возвращается `Nothing`.

- **Парсер для предложения, представленного как текст.**

  Этот парсер схож с парсером для предложения, но вместо списка слов он возвращает строку, состоящую из слов, соединённых пробелами. Он использует парсер `joinWords` для объединения списка слов в одну строку. Этот парсер может быть полезен, если нужно собрать предложение в текстовом формате, а не в виде списка отдельных слов.

  На выходе:

  - Если предложение успешно разобрано, парсер возвращает строку, представляющую предложение в текстовом виде.
  - Если не удалось распарсить предложение, возвращается `Nothing`.

- **Парсер для всех предложений в текстовом виде.**

  Этот парсер использует `some`, чтобы разобрать все предложения с помощью парсера `sentenceAsText`. Он извлекает все предложения, преобразуя каждое в строку, и возвращает их в виде списка текстовых строк.

  На выходе:

  - Если предложения успешно разобраны, возвращается список строк, представляющих предложения.
  - Если не удалось распарсить предложения, возвращается `Nothing`.

Создание N-грамм

В данном разделе представлен набор функций для генерации N-грамм из списка слов.

- **Тип `NGramMap`.** Определён тип `NGramMap` как список пар (`(Text, [Text])`), где каждый элемент представляет собой пару: слово (или несколько слов) и список слов, которые следуют за ним в тексте. Это основная структура данных для представления N-грамм.

- **Функция `toBiGrams`.** Эта функция генерирует биграммы (двуграммы) из списка слов. Биграммы — это пары, состоящие из текущего слова и следующего за ним. Функция использует стандартную функцию `zip` для формирования пар слов: первое слово из списка и следующее за ним.

- **Функция `toBiGramsJoined`.** В этой функции генерируются биграммы, где второе слово в паре состоит из двух слов, соединённых пробелом. Для этого используется вспомогательная функция `triple`, которая разбивает список на тройки подряд идущих слов. После чего слова во второй части тройки объединяются в одну строку с пробелом между ними.

- **Функция `triple`.** Эта функция используется для разбиения списка на тройки подряд идущих элементов.

- **Функция** `toTriGrams`. Генерирует триграммы из списка слов. В отличие от биграмм, триграммы состоят из двух первых слов, соединённых пробелом, и третьего слова, которое следует за ними. Триграммы представляют собой пары: первые два слова объединяются в строку, а третье слово идёт отдельно.

- **Функция** `groupPairs`. Эта функция принимает список пар и группирует их по первому элементу. Например, если пары содержат одинаковое первое слово, они будут собраны в одну группу, а вторые элементы (слова) будут собраны в список. Это позволяет собрать N-граммы для каждого слова, указывающего на следующие слова или фразы в тексте.

- **Функция** `makeNGrams`. Функция `makeNGrams` комбинирует все три предыдущие функции, генерируя полный набор N-грамм из списка слов. Она создаёт:

  – Биграммы, где каждое слово сопоставляется с последующим.

  – Биграммы, где каждое слово сопоставляется с двумя следующими словами, соединёнными пробелом.

  – Триграммы, где комбинация из двух слов сопоставляется с третьим словом.

  Все эти данные собираются в общий список N-грамм.

Генерация предложений

Этот код реализует функционал для обработки текста, создания N-грамм и генерации предложений на основе этих N-грамм.

- **Функция** `processText`. Эта функция принимает текст и генерирует набор N-грамм для этого текста. В процессе работы:

  – Текст парсится с использованием парсера `allSentences`, который извлекает все предложения.

  – Все слова из предложений извлекаются и дублируются (с помощью `nub`, чтобы исключить повторения).

  – N-граммы генерируются с помощью функции `makeNGrams`, а дубликаты удаляются.

  – Добавляются слова, которые не имеют следующих слов в N-граммах. Эти слова добавляются в виде пар с пустыми списками.

  – Результат сортируется по алфавиту по ключам слов.

  Выходом функции является отсортированный список N-грамм, который затем используется для генерации предложений.

- **Функция** `generatePhrase`. Эта функция генерирует фразу на основе заданного начального слова и N-грамм, используя генератор случайных чисел. Процесс генерации фразы включает следующие этапы:

  – Если первое слово не найдено в словаре N-грамм, возвращается ошибка с сообщением, что слово не найдено.

  – Если первое слово найдено в словаре, генерируется случайная длина фразы (между 2 и 15 словами).

- С помощью рекурсивной функции `generatePhraseHelper` фраза строится поэтапно, начиная с первого слова и добавляя к фразе подходящие следующие слова.

На каждом шаге выбирается следующее слово случайным образом из списка возможных слов (на основе текущего слова). Функция рекурсивно добавляет слова в итоговую фразу, пока не будет достигнута заданная длина.

- **Функция** `generatePhraseHelper`**.** Эта вспомогательная функция реализует рекурсивную генерацию фразы. Она продолжает добавлять слова в фразу, пока не будет достигнута максимальная длина. Если для текущего слова нет подходящих вариантов продолжения, рекурсия прекращается. Для каждого шага выбирается случайное слово из возможных, добавляется к фразе, и процесс повторяется для нового слова.

- **Функция** `findLastValidWord`**.** Эта функция ищет первое подходящее слово в фразе, которое встречается в словаре N-грамм. Процесс начинается с последнего слова в фразе и движется к первому. Если слово найдено в словаре, оно возвращается, в противном случае поиск продолжается для предыдущего слова. Если подходящее слово не найдено, возвращается `Nothing`.

- **Типы** `DialogueResponse` **и** `DialogueTurn`**.**

  - `DialogueResponse` — это тип, представляющий возможный ответ в диалоге: либо ошибку (`Left`), либо сгенерированную фразу (`Right`).
  - `DialogueTurn` — это кортеж, состоящий из номера модели и её ответа в виде `DialogueResponse`.

Генерация диалога моделей

Этот код реализует алгоритм для генерации диалога между двумя моделями, которые чередуют свои ответы, используя два набора N-грамм. Алгоритм генерирует последовательность фраз, опираясь на случайные выборы из N-грамм, при этом ограничивает глубину диалога и переключает модели после каждого ответа.

- **Тип** `DialogueResponse`**.** Этот тип представляет возможный ответ в диалоге. Он может быть:

  - `Left` — ошибка, если не удалось сгенерировать фразу.
  - `Right` — успешный ответ в виде списка слов (фразы).

- **Тип** `DialogueTurn`**.** Этот тип представляет один оборот в диалоге, состоящий из:

  - Номера модели (1 или 2).
  - Ответа модели, представленного типом `DialogueResponse`.

- **Функция** `generateDialogue`**.** Основная функция для генерации диалога. Она принимает случайный генератор (`gen`), начальное слово (`firstWord`), два набора N-грамм (`dict1` и `dict2`), и максимальную глубину диалога (`depth`).

  - Сначала генерируется первый ответ модели 1 с использованием функции `generatePhrase`.
  - Затем начинается рекурсивная генерация диалога с помощью вспомогательной функции `generateDialogueHelper`.

В результате функция возвращает список диалогов, каждый из которых состоит из номера модели и её ответа.

- **Функция** `generateDialogueHelper`**.** Эта функция рекурсивно генерирует последующие фразы диалога.
  - Каждый оборот в диалоге начинается с выбора следующей модели (переключение между моделью 1 и моделью 2).
  - Для каждой модели выбирается соответствующий набор N-грамм.
  - Для каждого ответа извлекается последнее слово, которое подходит для продолжения диалога (с помощью функции `findLastValidWord`).
  - Если подходящее слово найдено, генерируется новый ответ с использованием `generatePhrase`.
  - Если слово не найдено, возвращается ошибка с сообщением о невозможности продолжения диалога.
  - После каждого ответа переключается модель, и процесс повторяется до достижения максимальной глубины диалога (`depth`).

  Рекурсия завершается, когда глубина диалога достигает нуля.

- **Функция** `findLastValidWord`**.** Эта функция ищет последнее подходящее слово из предыдущего ответа, которое существует в словаре N-грамм для следующей модели. Она перебирает слова с конца списка ответа и проверяет их наличие в словаре. Если подходящее слово найдено, оно возвращается, в противном случае возвращается `Nothing`.

4 Результаты

4.1 Часть 1. Парсер строк бинарных чисел и операций

Для проверки работы парсера используется следующий текстовый файл:

```
    101010   &   00010
   1101  |  1011
  1010 ^ 1001
1111 & 1100
  10101   |  11011
   110   ^  100
   10102 & 00010
  1101 $  1011
abcdeftrhg
asd 10101 || 11011
  10101 || 11011
 1111 & 1100 extra
   10101 |
 |  11011
 110 ^  ^ 100
     10101 | 11011
    101010 & 00010
  101010 &  00010
  1101    | 1011
 101010&00010
```

Результаты прохождения тестов, описанных выше, представлены на Рис. 1.

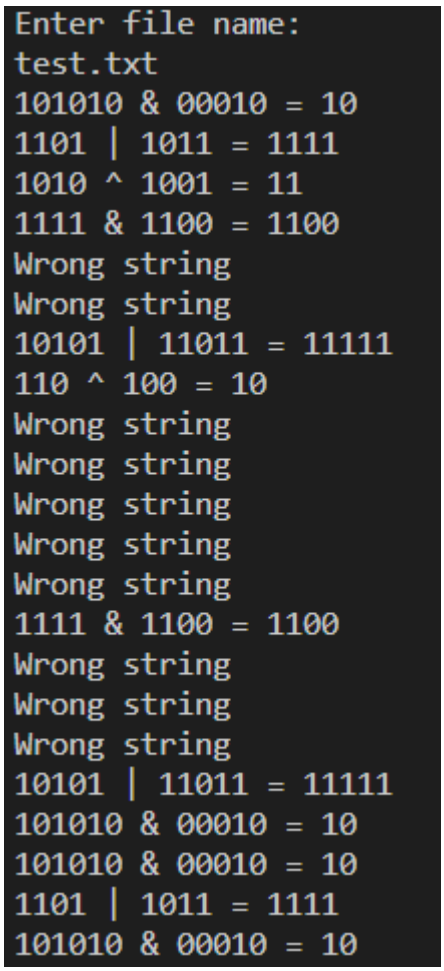

Рис. 1. Результаты прохождения тестов

При запуске программы, пользователь вводит пусть до файла, после чего для каждой строки из файла выводится результат вычисления выражения или сообщение о неверном формате строки.

4.2 Часть 2. Генератор предложений с помощью N-грамм

Для проверки работы парсера используются текстовые 2 файла содержащих: главы 1 и 2 произведения «Старуха Изергиль» на английском языке Максима Горького и стенограмму эпизода 304 «Твик против Крейга» мультсериала «Южный парк».

Сначала пользователь вводит начальное слово, с которого будет начинаться предложение первой модели (Рис. 2).

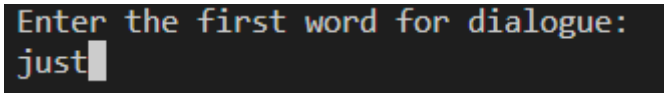

Рис. 2. Ввод начального слова

Затем пользователь вводит глубину сообщений в диалоге (Рис. **??**).

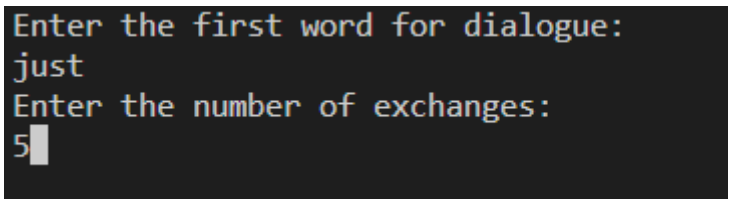

Рис. 3. Ввод глубины сообщений диалога

После этого программа выводит сгенерированный диалог состоящий из реплик двух моделей (Рис. ??).

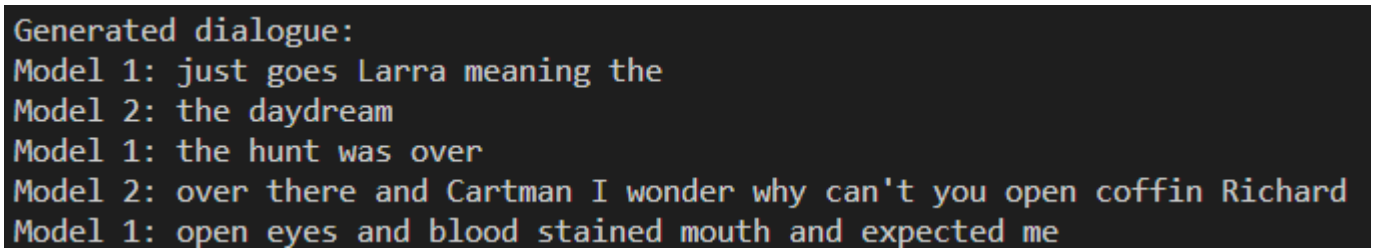

Рис. 4. Результат работы программы

По выводу программы видно, что каждая реплика начинается с последнего слова предыдущей реплики или, при отсутствии такого слова в словаре модели, используется предпоследнее слово и так далее.

Заключение

Часть 1

В данной работе был разработан синтаксический анализатор для разбора строк, содержащих битовые выражения с операциями AND, OR и XOR. Анализатор принимает на вход файл формата .txt, содержащий такие выражения, и выводит результаты их вычислений на экран. Реализованный парсер позволяет корректно обрабатывать строки с указанными операциями, обеспечивая удобство использования для конечного пользователя.

Часть 2

В ходе выполнения данного задания была разработана программа, которая решает задачу синтаксического анализа текста, построения модели N-грамм и генерации фраз на основе этих моделей. В рамках первой части задачи был реализован алгоритм чтения текста из файла, разбивки его на предложения и удаления символов пунктуации и цифр. Во второй части было создано два типа моделей N-грамм: биграммы и триграммы, а также сформирован словарь, который сохранен в файл формата .txt.

Третья часть задачи включала реализацию взаимодействия с пользователем, где пользователь мог вводить одно слово или пару слов, а программа генерировала фразу случайной длины от 2 до 15 слов. Если заданного пользователем слова не находилось среди ключей словаря, выводилось соответствующее сообщение.

Четвертая часть заключалась в организации диалога между двумя моделями N-грамм, созданными на основе различных текстов Максима Горького. Пользователю предоставлялась возможность задать начальное слово или пару слов и указать количество сообщений, которыми будут обмениваться модели. Модели строят свои ответы на основе последнего слова предыдущего сообщения оппонента.

Результаты работы программы демонстрируют корректность алгоритмов обработки текста и построения моделей N-грамм. Полученная система может быть использована для создания простых чат-ботов или других приложений, основанных на анализе естественного языка.

Приложение А. Парсер бинарных операций

Lib.hs

```haskell
module Lib
    ( parseAndPrint
    , readFileLines
    ) where

import Data.Text (Text)
import qualified Data.Text as T
import qualified Data.ByteString as B
import qualified Data.Text.Encoding as TE
import qualified Data.Text.IO as TIO
import Control.Applicative
import Data.Char (digitToInt, intToDigit)
import Data.Bits ((.&.), (.|.), xor)
import Numeric (showIntAtBase)

newtype Parser a = Parser { runParser :: Text -> Maybe (Text, a) }

instance Functor Parser where
    -- хотим применить func над результатом парсера p
    fmap func (Parser p) = Parser f where
        -- парсер f возвращает:
        f origText = case p origText of
            Nothing -> Nothing -- Nothing, если парсер p возвращает Nothing
            Just (remainingP, resP) -> Just (remainingP, func resP) -- (остаток, resP
            ↪  обработанный функцией func), если p возвращает (остаток, resP)

instance Applicative Parser where
    -- возвращаем всегда (изначальная строка, передаваемое значение)
    pure text = Parser (\orig -> Just(orig, text))

    -- хотим чтобы был парсер, который применяет к остатку 1 парсера 2 парсер,
    -- а затем применяет 1 парсер ко 2
    (Parser u) <*> (Parser v) = Parser f where
        f origText = case u origText of
            Nothing -> Nothing
            -- remainingU - остаток 1 парсера
            Just (remainingU, resU) -> case v remainingU of
                Nothing -> Nothing
                -- remainingV - итоговый остаток, resU применяем над resV
                Just (remainingV, resV) -> Just (remainingV, resU resV)

instance Alternative Parser where
    -- парсер всегда возвращающий Nothing
    empty = Parser $ \_ -> Nothing

    Parser u <|> Parser v = Parser f where
```

```haskell
        -- пытаемся применить парсер u
        f origText = case u origText of
            -- если он вернул Nothing, то применям парсер v
            Nothing -> v origText
            -- если вернул какой то результат, то оставляем результат
            res -> res

isBinaryChar :: Char -> Bool
isBinaryChar c =
    c == '0' || c == '1'

isOperation :: Char -> Bool
isOperation c =
    c == '&' || c == '|' || c == '?'

isSpace :: Char -> Bool
isSpace c = c == ' '

satisfy :: (Char -> Bool) -> Parser Char
satisfy pr = Parser f where
    -- берем первый символ
    f cs = case T.uncons cs of
        Nothing -> Nothing
        -- если первый элемент соответствует предикату pr
        Just (fstChar, remainingText)
            | pr fstChar -> Just (remainingText, fstChar) -- то возвращаем (остаток,
            ↪  подоходящий первый элемент)
            | otherwise -> Nothing
    f _ = Nothing

binary :: Parser Text
binary = Parser $ \text ->
    -- если первый элемент
    case runParser (satisfy isBinaryChar) text of
        Nothing -> Nothing -- не подошел, то строка очевидно сразу не подходит
        -- (рекурсивно) если парсер на остатке
        Just (remaining, c) -> case runParser binary remaining of
            Nothing -> Just (remaining, T.singleton c) -- первый элемент не подошел, то
            ↪  берем (старый остаток, единственный подошедший элемент)
            Just (remaining', rest) -> Just (remaining', T.cons c rest) -- сработал
            ↪  штатно, то (новый остаток, добавляем подошедший элемент к остальным)

binToInt :: Text -> Int
binToInt text = T.foldl' (\acc c -> acc * 2 + digitToInt c) 0 text

intToBin :: Int -> Text
intToBin n = T.pack (showIntAtBase 2 intToDigit n "")

binaryInt :: Parser Int
binaryInt = binToInt <$> binary
```

```haskell

operation :: Parser Char
operation = satisfy isOperation

oneSpace :: Parser Char
oneSpace = satisfy isSpace

spaces :: Parser Text
spaces = (T.cons) <$> oneSpace <*> spaces <|> pure T.empty

applyOperation :: Char -> Int -> Int -> Int
applyOperation op
    | op == '&' = (.&.)
    | op == '|' = (.|.)
    | op == '?' = xor

binaryExpression :: Parser Int
binaryExpression =
    (\_ b1 _ op _ b2 -> applyOperation op b1 b2)
    <$> spaces
    <*> binaryInt
    <*> spaces
    <*> operation
    <*> spaces
    <*> binaryInt

binaryExpressionFormatted :: Parser Text
binaryExpressionFormatted =
    (\_ b1 _ op _ b2 -> formatExpression b1 op b2)
    <$> spaces
    <*> binary
    <*> spaces
    <*> operation
    <*> spaces
    <*> binary

formatExpression :: Text -> Char -> Text -> Text
formatExpression b1 op b2 =
    let int1 = binToInt b1
        int2 = binToInt b2
        result = applyOperation op int1 int2
    in b1 <> (T.pack " ") <> T.singleton op <> (T.pack " ") <> b2 <> (T.pack " = ") <>
    ↪  intToBin result

readFileLines :: FilePath -> IO [T.Text]
readFileLines filePath = fmap (T.lines . TE.decodeUtf8) (B.readFile filePath)

parseAndPrint :: T.Text -> IO ()
parseAndPrint input =
    case runParser binaryExpressionFormatted input of
```

```
142         Just (_, result) -> TIO.putStrLn result
143         Nothing -> putStrLn "Wrong string"
```

Main.hs

```
1  module Main (main) where
2
3  import Lib
4  import System.IO
5  import qualified Data.Text.Encoding as TE
6  import qualified Data.ByteString as B
7
8  main :: IO ()
9  main = do
10     hSetEncoding stdout utf8
11     putStrLn "Enter file name:"
12     fileName <- getLine
13     lines <- readFileLines fileName
14     -- mapM_ (TE.decodeUtf8 . B.putStrLn) lines
15     mapM_ parseAndPrint lines
```

Приложение Б. Генератор предложений

Lib.hs

```haskell
module Lib
    ( allSentences
    , runParser
    , generatePhrase
    , processText
    , NGramMap
    , generateDialogue
    ) where

import Control.Applicative
import Data.Text (Text)
import qualified Data.Text as T
import Data.Char (isLetter, isSpace)
import Data.List (sortBy, groupBy, nub)
import System.Random (RandomGen, randomR, next, split)

-- Parser
-- instances

newtype Parser a = Parser { runParser :: Text -> Maybe (Text, a) }

instance Functor Parser where
    -- хотим применить func над результатом парсера p
    fmap func (Parser p) = Parser f where
        -- парсер f возвращает:
        f origText = case p origText of
            Nothing -> Nothing -- Nothing, если парсер p возвращает Nothing
            Just (remainingP, resP) -> Just (remainingP, func resP) -- (остаток, resP
            ↪  обработанный функцией func), если p возвращает (остаток, resP)

instance Applicative Parser where
    -- возвращаем всегда (изначальная строка, передаваемое значение)
    pure text = Parser (\orig -> Just(orig, text))

    -- хотим чтобы был парсер, который применяет к остатку 1 парсера 2 парсер,
    -- а затем применяет 1 парсер ко 2
    (Parser u) <*> (Parser v) = Parser f where
        f origText = case u origText of
            Nothing -> Nothing
            -- remainingU - остаток 1 парсера
            Just (remainingU, resU) -> case v remainingU of
                Nothing -> Nothing
                -- remainingV - итоговый остаток, resU применяем над resV
                Just (remainingV, resV) -> Just (remainingV, resU resV)

instance Alternative Parser where
```

```haskell
    -- парсер всегда возвращающий Nothing
    empty = Parser $ \_ -> Nothing

    Parser u <|> Parser v = Parser f where
        -- пытаемся применить парсер u
        f origText = case u origText of
            -- если он вернул Nothing, то применям парсер v
            Nothing -> v origText
            -- если вернул какой то результат, то оставляем результат
            res -> res

-- functions

isPunctuation :: Char -> Bool
isPunctuation c = c `elem` ['.', '!', '?', ';', ':', '(', ')']

joinWords :: [Text] -> Text
joinWords = T.intercalate (T.pack " ")

-- parsers

satisfy :: (Char -> Bool) -> Parser Char
satisfy pr = Parser f where
    -- берем первый символ
    f cs = case T.uncons cs of
        Nothing -> Nothing
        -- если первый элемент соответствует предикату pr
        Just (fstChar, remainingText)
            | pr fstChar -> Just (remainingText, fstChar) -- то возвращаем (остаток,
            ↪ подоходящий первый элемент)
            | otherwise -> Nothing
    f _ = Nothing

isWordChar :: Char -> Bool
isWordChar c = isLetter c || c == '\''

word :: Parser Text
word = Parser f where
    f input = case runParser (some (satisfy isWordChar)) input of
        Nothing -> Nothing
        Just (remaining, chars) ->
            -- проверяем что слово начинается или заканчивается апострофом
            let word = T.pack chars
                cleanWord = T.dropAround (== '\'') word
            in if T.null cleanWord
                then Nothing  -- если слово состоит только из апострофов, то возвращаем
                ↪ Nothing
                else Just (remaining, word)

oneSpace :: Parser Char
```

```haskell
oneSpace = satisfy isSpace

spaces :: Parser Text
spaces = (T.cons) <$> oneSpace <*> spaces <|> pure T.empty

punctuation :: Parser Char
punctuation = satisfy isPunctuation

skipJunk :: Parser ()
skipJunk = Parser f where
    f input = Just (T.dropWhile (\c -> not (isLetter c || isPunctuation c)) input, ())

sentence :: Parser [Text]
sentence = (\words _ -> words)
    <$> some (word <* skipJunk)
    <*> some punctuation
    <* skipJunk
    <|> empty

allSentences :: Parser [[Text]]
allSentences = some sentence

sentenceAsText :: Parser Text
sentenceAsText = (\words _ -> joinWords words)
    <$> some (word <* skipJunk)
    <*> some punctuation
    <* skipJunk
    <|> empty

allSentencesAsText :: Parser [Text]
allSentencesAsText = some sentenceAsText

-- N-gram

type NGramMap = [(Text, [Text])]

toBiGrams :: [Text] -> [(Text, Text)]
toBiGrams words = zip words (tail words)

toBiGramsJoined :: [Text] -> [(Text, Text)]
toBiGramsJoined ws =
    [(w1, T.concat [w2, T.pack " ", w3]) | (w1,w2,w3) <- triple ws]

triple :: [a] -> [(a, a, a)]
triple (x:y:z:rest) = (x,y,z) : triple (y:z:rest)
triple _ = []

toTriGrams :: [Text] -> [(Text, Text)]
toTriGrams ws =
    [(T.concat [w1, T.pack " ", w2], w3) | (w1,w2,w3) <- triple ws]
```

```haskell


groupPairs :: [(Text, Text)] -> NGramMap
groupPairs pairs = map (\group -> (fst $ head group, map snd group))
                 $ groupBy (\x y -> fst x == fst y)
                 $ sortBy (\x y -> compare (fst x) (fst y)) pairs

makeNGrams :: [Text] -> [(Text, Text)]
makeNGrams words =
    -- создаем биграммы вида (word -> next word)
    toBiGrams words ++
    -- создаем биграммы вида (word -> two next words joined)
    toBiGramsJoined words ++
    -- создаем триграммы вида (two words -> next word)
    toTriGrams words

processText :: Text -> NGramMap
processText text = case runParser allSentences text of
    Nothing -> []
    Just (_, sentences) -> let
        -- берем все слова из всех предложений
        allWords = nub $ concat sentences
        -- создаем n-граммы из всех предложений и удаляем дубликаты
        allNGrams = groupPairs
                  $ nub -- удаляем дубликаты
                  $ concatMap makeNGrams sentences
        -- добавляем слова, которые не имеют следующих слов
        singleWords = map (\w -> (w, []))
                    $ filter (\w -> not $ any (\(prefix, _) -> prefix == w) allNGrams)
                    $ allWords
        in sortBy (\x y -> compare (fst x) (fst y))
           $ allNGrams ++ singleWords

generatePhrase :: RandomGen g => g -> Text -> NGramMap -> Either Text [Text]
generatePhrase gen firstWord nGrams =
    -- если первого слова нет в словаре, то возвращаем ошибку (Left ошибка, Right [слова
    ↪  для фразы])
    case lookup firstWord nGrams of
        Nothing -> Left $ T.concat [T.pack "Word '", firstWord, T.pack "' not found in
        ↪  the dictionary"]
        -- если первое слово есть в словаре, то начинаем генерировать фразу
        Just nextWords ->
            let (targetLength, newGen) = randomR (2, 15) gen
            in Right $ generatePhraseHelper newGen [firstWord] firstWord nGrams
            ↪  targetLength

generatePhraseHelper :: RandomGen g => g -> [Text] -> Text -> NGramMap -> Int -> [Text]
generatePhraseHelper gen acc lastKey nGrams targetLength
    | length acc >= targetLength = take targetLength acc  -- достигли максимальной
    ↪  длины
```

```haskell
    | null possibleNextWords = acc  -- нет больше слов
    | otherwise =
        let (idx, newGen) = randomR (0, length possibleNextWords - 1) gen
            nextWord = possibleNextWords !! idx
            -- если два слова, то разбиваем на два
            nextWords = T.words nextWord
            -- и добавляем в список слов фразы
            newAcc = acc ++ nextWords
            -- используем следующее слово (или два слова) как ключ для следующего шага
            newKey = nextWord
        -- рекурсивно вызываем функцию для следующего шага
        in generatePhraseHelper newGen newAcc newKey nGrams targetLength
  where
    -- находим список слов по данному ключу (Just [Text]), если такого ключа в словаре
    ↪  нет, то Nothing
    possibleNextWords = maybe [] id $ lookup lastKey nGrams

-- Найти первое подходящее слово от конца фразы, которое есть в словаре
findLastValidWord :: [Text] -> NGramMap -> Maybe Text
findLastValidWord [] _ = Nothing
findLastValidWord (word:rest) nGrams =
    case lookup word nGrams of
        Just _ -> Just word
        Nothing -> findLastValidWord rest nGrams

type DialogueResponse = Either Text [Text]  -- Left - ошибка, Right - фраза
type DialogueTurn = (Int, DialogueResponse)  -- (номер модели, ответ)

generateDialogue :: RandomGen g =>
                    g ->
                    Text ->
                    NGramMap ->
                    NGramMap ->
                    Int ->
                    [DialogueTurn]  -- возвращает список (номер модели, ответ)
generateDialogue gen firstWord dict1 dict2 depth =
    let firstResponse = generatePhrase gen firstWord dict1
        (_, newGen) = next gen
    in (1, firstResponse) : generateDialogueHelper newGen firstResponse 1 dict1 dict2
    ↪  depth []

generateDialogueHelper :: RandomGen g =>
                          g ->
                          DialogueResponse -> -- последний ответ
                          Int ->              -- номер модели
                          NGramMap ->         -- первый словарь
                          NGramMap ->         -- второй словарь
                          Int ->              -- остаток глубины
                          [DialogueTurn] ->   -- накопленный диалог
                          [DialogueTurn]      -- финальный диалог
```

```haskell
generateDialogueHelper _ _ _ _ _ 0 acc = reverse acc
generateDialogueHelper gen lastResponse speaker dict1 dict2 depth acc =
    let
        -- следующая модель
        nextSpeaker = if speaker == 1 then 2 else 1

        -- словарь для следующей модели
        currentDict = if nextSpeaker == 1 then dict1 else dict2

        -- последнее подходящее слово из предыдущего ответа
        lastWords = case lastResponse of
            Right phrase -> reverse phrase  -- если слово есть
            Left _ -> []                    -- если нет, то пустой список

        -- последнее подходящее слово из предыдущего ответа
        nextWord = findLastValidWord lastWords currentDict

        -- следующий ответ (даже если не нашли подходящего слова)
        nextResponse = case nextWord of
            Nothing -> Left (T.pack "No valid word found to continue dialogue")
            Just word ->
                let (newGen1, _) = split gen
                in generatePhrase newGen1 word currentDict

        (_, newGen2) = split gen
    in generateDialogueHelper
        newGen2
        nextResponse
        nextSpeaker
        dict1
        dict2
        (depth - 1)
        ((nextSpeaker, nextResponse) : acc)
```

Main.hs

```haskell
module Main (main) where

import Lib
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import System.Random (newStdGen)

main :: IO ()
main = do

    input1 <- TIO.readFile "input.txt"
    input2 <- TIO.readFile "input2.txt"

```

```
    let nGrams1 = processText input1
    let nGrams2 = processText input2

    -- print nGrams1
    -- print "\n"
    -- print nGrams2
    -- print "\n"

    TIO.putStrLn (T.pack "Enter the first word for dialogue:")
    firstWord <- T.strip <$> TIO.getLine

    TIO.putStrLn (T.pack "Enter the number of exchanges:")
    depthStr <- getLine
    let depth = (read depthStr :: Int) - 1

    gen <- newStdGen
    let dialogue = generateDialogue gen firstWord nGrams1 nGrams2 depth
    TIO.putStrLn (T.pack "\nGenerated dialogue:")
    mapM_ (printDialogueTurn . formatDialogueTurn) dialogue
  where
    formatDialogueTurn (speaker, response) =
        (T.pack $ "Model " ++ show speaker ++ ": ",
         case response of
            Left err -> err
            Right phrase -> T.unwords phrase)

    printDialogueTurn (prefix, message) =
        TIO.putStr prefix >> TIO.putStrLn message
```