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»

Лабораторная работа №3
«Грамматика простого прошедшего времени немецкого языка»
по дисциплине
«Математическая логика и теория автоматов»
Вариант 15

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

Преподаватель _____ Востров А. В.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

Введение 4

1 Грамматика немецкого претеритума 5

1.1 Общая характеристика претеритума . . . 5
1.2 Структура предложений в претеритуме . . . 5
1.3 Образование форм претеритума . . . 5
1.3.1 Слабые (правильные) глаголы . . . 5
1.3.2 Сильные (неправильные) глаголы . . . 6
1.3.3 Смешанные глаголы . . . 6
1.4 Отрицание в претеритуме . . . 6
1.5 Особенности использования претеритума . . . 7

2 Математическое описание 8

2.1 Особенности грамматики . . . 8
2.2 Пораждающая грамматика Хомского . . . 8
2.3 Грамматика в БНФ . . . 10
2.4 Пример вывода предложения . . . 11
2.5 Алгоритм построения LL(1) таблицы синтаксического анализа . . . 13
2.6 Алгоритм разбора предложений . . . 14
2.7 Алгоритм генерации предложения . . . 15

3 Особенности реализации 16

3.1 Задание правил грамматики . . . 16
3.2 Класс `Grammar` . . . 16
3.2.1 Метод `_parse_productions` . . . 17
3.2.2 Метод `_calculate_first_sets` . . . 17
3.2.3 Метод `_calculate_follow_sets` . . . 18
3.2.4 Метод `_first_of_sequence` . . . 19
3.2.5 Метод `_fill_lookup_table` . . . 20
3.2.6 Метод `format_rules` . . . 21
3.2.7 Метод `format_lookup_table` . . . 22
3.2.8 Метод `format_first_sets` . . . 22
3.2.9 Метод `format_follow_sets` . . . 23
3.2.10 Метод `analyze` . . . 23
3.2.11 Метод `generate` . . . 24

3.2.12 Метод `generate_derivation_steps` 25
3.3 Функция `main` 26

4 Результаты работы программы **28**

Заключение **29**

Список литературы **30**

Введение

Лабораторная №3 заключается в построении контексно-свободной грамматики подмножества естественного языка и написании программы для генерации предложений языка и проверки предложений на принадлежность языку. В данном варианте рассматривается грамматика простого прошедшего времени немецкого языка.

Простое прошедшее время в немецком языке называется *Претерит* или *Претеритум* (нем. *Präteritum* или нем. *Imperfekt*). Это время не требует образования сложных конструкций, что позволяет относить его к простым формам.

1 Грамматика немецкого претеритума

1.1 Общая характеристика претеритума

Претеритум (Präteritum, Imperfekt) является одной из двух основных форм прошедшего времени в немецком языке, наряду с перфектом (Perfekt). В отличие от перфекта, претеритум представляет собой простую (синтетическую) форму, образуемую без вспомогательных глаголов, путём изменения основной формы глагола.

Претеритум используется преимущественно в письменной речи, особенно в художественной литературе, исторических текстах и формальных документах. В разговорной речи он встречается редко, за исключением глаголов *sein* (быть), *haben* (иметь), модальных глаголов, а также в некоторых диалектах северной Германии.

1.2 Структура предложений в претеритуме

Структура предложений в претеритуме соответствует общим правилам построения немецких предложений. Основные типы структур:

1. **Прямой порядок слов** (главное предложение):

   Подлежащее + глагол в претеритуме + дополнения и обстоятельства

   Например: *Der Mann las ein Buch.* (Мужчина читал книгу.)

2. **Обратный порядок слов** (инверсия):

   Обстоятельство + глагол в претеритуме + подлежащее + другие члены предложения

   Например: *Gestern las der Mann ein Buch.* (Вчера мужчина читал книгу.)

3. **Придаточное предложение**:

   ..., союз + подлежащее + второстепенные члены + глагол в претеритуме

   Например: *Ich wusste, dass der Mann ein Buch las.* (Я знал, что мужчина читал книгу.)

1.3 Образование форм претеритума

1.3.1 Слабые (правильные) глаголы

Образуются по формуле: основа глагола + суффикс *-te* + личное окончание:

Лицо	Окончание (machen - делать)
ich	mach**te**
du	mach**test**
er/sie/es	mach**te**
wir	mach**ten**
ihr	mach**tet**
sie/Sie	mach**ten**

1.3.2 Сильные (неправильные) глаголы

Образуются путём изменения корневой гласной без добавления суффикса *-te*:

Лицо	Окончание (lesen - читать)
ich	**las**
du	**las**st
er/sie/es	**las**
wir	**las**en
ihr	**las**t
sie/Sie	**las**en

1.3.3 Смешанные глаголы

Сочетают особенности сильных и слабых глаголов: меняют корневую гласную и получают суффикс *-te*. Например: *bringen* (приносить) – *brachte*, *kennen* (знать) – *kannte*.

1.4 Отрицание в претеритуме

Отрицание в предложениях с претеритом образуется с помощью отрицательной частицы *nicht*, которая обычно ставится после глагола и перед дополнениями:

Der Mann las nicht ein Buch. (Мужчина не читал книгу.)

При отрицании всего предложения *nicht* ставится в конце:

Der Mann las ein Buch nicht. (Мужчина не читал книгу [а что-то другое].)

Размещение *nicht* в конце предложения также часто используется в разговорной речи и может указывать на отрицание всего действия или ситуации в целом:

Er kam gestern nicht. (Он вчера не приходил.)

Das Kind spielte draußen nicht. (Ребенок не играл на улице.)

1.5 Особенности использования претеритума

В немецком языке выбор между претеритумом и перфектом часто определяется:

- **Стилем повествования:** претеритум более характерен для литературного стиля
- **Типом текста:** в биографиях, исторических текстах, сказках преимущественно используется претеритум
- **Региональными особенностями:** в Северной Германии претеритум используется чаще
- **Типом глагола:** для глаголов *sein*, *haben* и модальных глаголов претеритум используется даже в разговорной речи

В рамках данной работы мы рассматриваем формализованную грамматику, учитывающую основные структурные особенности предложений в претеритуме, без полного описания всех лексических и морфологических особенностей немецкого языка.

2 Математическое описание

2.1 Особенности грамматики

Грамматика, рассматриваемая в рамках данной лабораторной работы, имеет следующие особенности:

- Рассматриваются только повествовательные и повествовательные отрицательные предложения.
- Учтены два возможных порядка членов предложения: прямой и обратный. В прямом порядке сначала идет подлежащее, потом сказуемое, а затем второстепенные члены предложения. В обратном порядке на первом месте стоит обстоятельство, на втором сказуемое, на третьем подлежащее, а затем второстепенные члены предложения. В *Претеритуме* сказуемое всегда стоит на втором месте.
- Придаточные предложения могут относится только к подлежащему и начинаться с союзов *welcher* (который), *welche* (которая), *welches* (которое). При этом вложенность придаточных предложений не ограничена.
- Порядок второстепеннных членов не фиксирован. Несмотря на то, что в немецком языке существует предпочтительный порядок слов, в данной грамматике он не учитывается, потому что порядок не является строгим и реальные предложения языка далеко не всегда ему соответствуют.
- Рассматриваются только простые сказуемые, не содержащие вспомогательных глаголов.
- Рассматриваются три вида обстоятельств: обстоятельства времени, места и образа действия. В реальных предложениях языка могут встречаться и другие виды, но перечисленные являются наиболее часто встречающимися.

2.2 Пораждающая грамматика Хомского

Формально пораждающую грамматику Хомского можно задать списком продукций:

```
Предложение -> Повествовательное "."
Повествовательное -> ПрямойПорядок | Инверсия
ПрямойПорядок -> Подлежащее ДополнениеКПодлежащему Глагол
                 ВторостепенныеЧлены Отрицание
Инверсия -> Обстоятельство Глагол Подлежащее
            ВторостепенныеЧлены Отрицание
Подлежащее -> ИменнаяГруппа ПридаточноеПредложение | Местоимение
ПридаточноеПредложение -> "," Союз Подлежащее Глагол Отрицание "," |
                          epsilon
ВторостепенныеЧлены -> ВторостепенныйЧлен ВторостепенныеЧлены |
```

```
                        epsilon
ВторостепенныйЧлен -> Обстоятельство | Дополнение
Обстоятельство -> ОбстоятельствоВремени | ОбстоятельствоМеста |
                  ОбстоятельствоОбразаДействия
ИменнаяГруппа -> АртикльЛибоМестоимение Прилагательные
                 Существительное
АртикльЛибоМестоимение -> Артикль | ПритяжательноеМестоимение |
                          epsilon
Прилагательные -> Прилагательное Прилагательные | epsilon
ДополнениеКПодлежащему -> Предлог Существительное | epsilon
Дополнение -> АртикльЛибоМестоимение Прилагательные Существительное
ОбстоятельствоМеста -> Предлог АртикльЛибоМестоимение Существительное
Союз -> "welcher" | "welche" | "welches"
Отрицание -> "nicht" | epsilon
Местоимение -> "ich" | "du" | "er" | "sie" | "es" | "wir" |
               "ihr" | "Sie"
ПритяжательноеМестоимение -> "mein" | "dein" | "sein" |
                             "unser" | "euer" | "ihre"
Артикль -> "der" | "die" | "das" |  "ein" |  "eine" | "einen" |
           "einem" | "einer"
Прилагательное -> "alt" | "jung" | "groß" | "klein" | "schön" |
                  "freundlich" | "süß" | "ruhig"
Существительное -> "Mann" | "Frau" | "Kind" | "Buch" | "Brief" |
                   "Freund" | "Abendessen" | "Suppe"
Предлог -> "in" | "auf" | "unter" | "aus" | "mit" | "für" |
           "zu" | "am"
ОбстоятельствоВремени -> "gestern" | "heute" | "morgen" | "damals" |
                         "jetzt" | "früh" | "spät" | "immer"
ОбстоятельствоОбразаДействия -> "schnell" | "langsam" | "gut" |
                                 "schlecht" | "laut" | "leise" |
                                 "gern" | "fleißig"
Глагол -> "las" | "schrieb" | "kochte" | "aß" | "ging" |
          "kam" | "sagte" | "machte"
```

Терминальными являются все символы в кавычках, все остальные символы являются нетерминальными.

Язык, порождаемый данной грамматикой, как и сама грамматика, является контекстно-свободным, так как:

- В левых частях всех продукций присутствует только один нетерминальный символ, а значит язык не является контекстно-зависимым.
- Язык не является автоматным из-за наличия придаточных предложений неограниченной вложенности. Придаточные предложения являются аналогом скобочной структуры, которая не может быть описана конечным автоматом.

Также эта грамматика является LL(1)-грамматикой. Множества FIRST и

FOLLOW для всех нетерминальных символов не пересекаются. В дальнейшем это особенность используется для распознавания предложений языка.

2.3 Грамматика в БНФ

Грамматика в БНФ имеет следующий вид:

```
<Предложение> ::= <Повествовательное> "."
<Повествовательное> ::= <ПрямойПорядок> | <Инверсия>
<ПрямойПорядок> ::= <Подлежащее> [ДополнениеКПодлежащему] <Глагол>
                    {ВторостепенныйЧлен} [Отрицание]
<Инверсия> ::= <Обстоятельство> <Глагол> <Подлежащее>
            {ВторостепенныйЧлен} [Отрицание]
<Подлежащее> ::= <ИменнаяГруппа> [ПридаточноеПредложение] | Местоимение
<ПридаточноеПредложение> ::= "," <Союз> <Подлежащее> <Глагол> [Отрицание] ","
<ВторостепенныйЧлен> ::= <Обстоятельство> | <Дополнение>
<Обстоятельство> ::= <ОбстоятельствоВремени> | <ОбстоятельствоМеста> |
                     <ОбстоятельствоОбразаДействия>
<ИменнаяГруппа> ::= [Артикль | ПритяжательноеМестоимение]
                   {Прилагательное} <Существительное>
<ДополнениеКПодлежащему> ::= <Предлог> <Существительное>
<Дополнение> ::= [Артикль | ПритяжательноеМестоимение]
                 {Прилагательное} <Существительное>
<ОбстоятельствоМеста> ::= <Предлог>
            [Артикль | ПритяжательноеМестоимение] <Существительное>
<Союз> ::= "welcher" | "welche" | "welches"
<Отрицание> ::= "nicht"
<Местоимение> ::= "ich" | "du" | "er" | "sie" | "es" | "wir" |
                "ihr" | "Sie"
<ПритяжательноеМестоимение> ::= "mein" | "dein" | "sein" |
                                "unser" | "euer" | "ihre"
<Артикль> ::= "der" | "die" | "das" |  "ein" |  "eine" | "einen" |
            "einem" | "einer"
<Прилагательное> ::= "alt" | "jung" | "groß" | "klein" | "schön" |
                     "freundlich" | "süß" | "ruhig"
<Существительное> ::= "Mann" | "Frau" | "Kind" | "Buch" | "Brief" |
                      "Freund" | "Abendessen" | "Suppe"
<Предлог> ::= "in" | "auf" | "unter" | "aus" | "mit" | "für" |
            "zu" | "am"
<ОбстоятельствоВремени> ::= "gestern" | "heute" | "morgen" |
                "damals" | "jetzt" | "früh" | "spät" | "immer"
<ОбстоятельствоОбразаДействия> ::= "schnell" | "langsam" | "gut" |
                                   "schlecht" | "laut" | "leise" |
                                   "gern" | "fleißig"
<Глагол> ::= "las" | "schrieb" | "kochte" | "aß" | "ging" |
            "kam" | "sagte" | "machte" | "liebte"
```

2.4 Пример вывода предложения

На Рис. 1 представлено дерево вывода предложения *«Mein alt Freund, welcher ich liebte, schrieb gestern einen schön Brief.»*

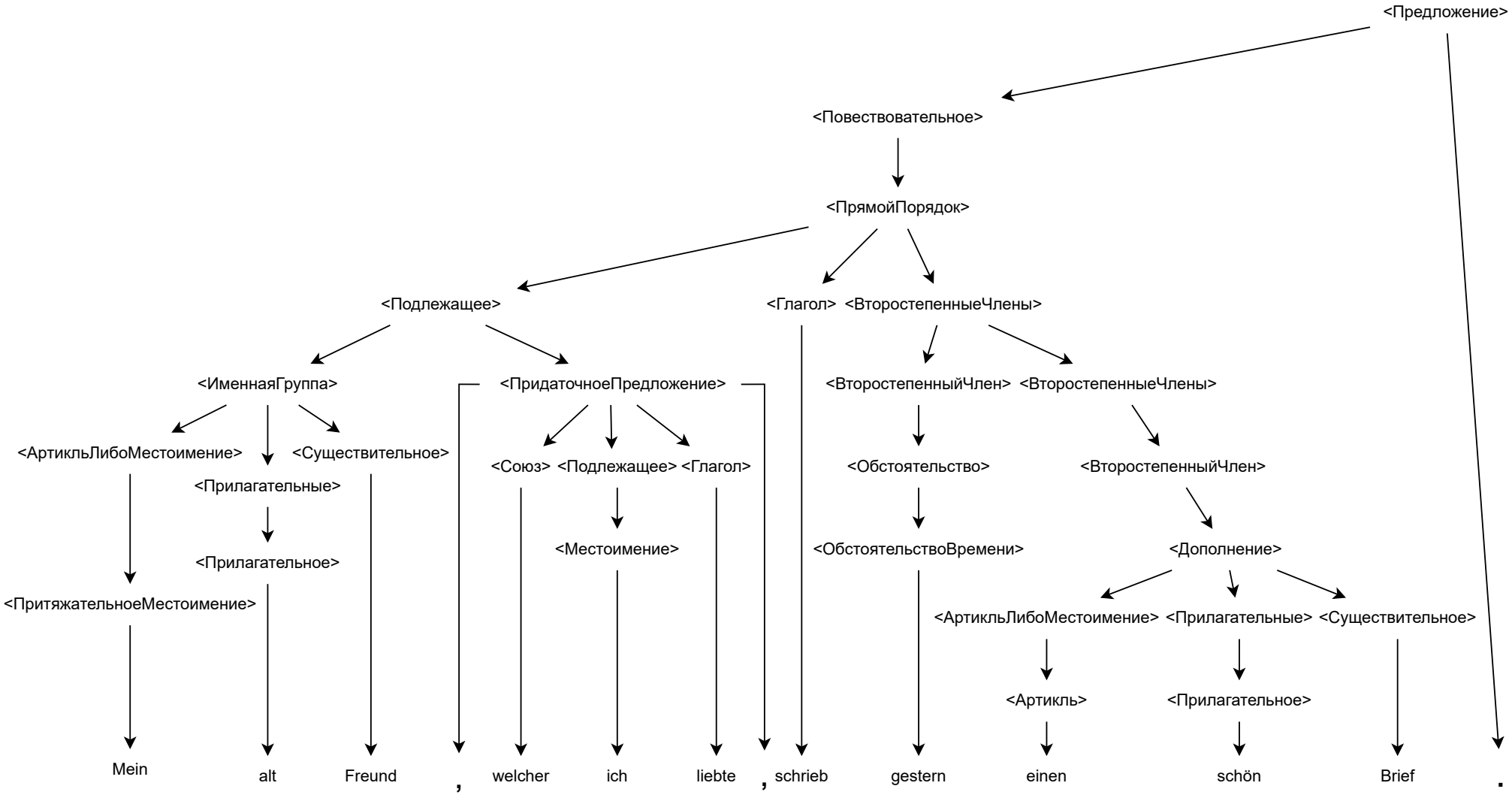

Рис 1. Пример дерева вывода предложения.

2.5 Алгоритм построения LL(1) таблицы синтаксического анализа

Чтобы заполнить таблицу синтаксического анализа, необходимо установить, какое правило грамматики синтаксический анализатор должен выбрать, если нетерминальное A находится на вершине его стека и символ a — во входном потоке. Легко заметить, что такое правило должно иметь форму $A \to w$ и что у языка, соответствующего w, должна быть по крайней мере одна строка, начинающаяся с a.

С этой целью мы определяем **множество FIRST() для** w, обозначенное как $\texttt{FIRST}(w)$, как множество терминалов, которые могут быть найдены в начале любой строки в w, и ε, если пустая строка также принадлежит w.

Учитывая грамматику с правилами $A_1 \to w_1, \ldots, A_n \to w_n$, можно вычислить $\texttt{FIRST}(w_i)$ и $\texttt{FIRST}(A_i)$ для каждого правила следующим образом:

1. Инициализировать каждое множество $\texttt{FIRST}(A_i)$ пустым множеством.
2. Добавить $\texttt{FIRST}(w_i)$ к $\texttt{FIRST}(A_i)$ для каждого правила $A_i \to w_i$, где $\texttt{FIRST}(w_i)$ определяется следующим образом:
   - $\texttt{FIRST}(aw') = \{a\}$ для каждого терминала a
   - $\texttt{FIRST}(Aw') = \texttt{FIRST}(A)$ для каждого нетерминального A с $\varepsilon \notin \texttt{FIRST}(A)$
   - $\texttt{FIRST}(Aw') = (\texttt{FIRST}(A) \setminus \{\varepsilon\}) \cup \texttt{FIRST}(w')$ для каждого нетерминального A с $\varepsilon \in \texttt{FIRST}(A)$, включая случай $A_i \to A$, то есть $w' = \varepsilon$, в этом случае $\texttt{FIRST}(Aw') = \texttt{FIRST}(A)$
   - $\texttt{FIRST}(\varepsilon) = \{\varepsilon\}$
3. Повторять шаг 2, пока в множествах FIRST происходят изменения.

Однако множеств FIRST() недостаточно, чтобы построить таблицу синтаксического анализа. Так происходит, потому что правая сторона w правила могла бы в конечном счёте быть приведена к пустой строке. Таким образом синтаксический анализатор должен также использовать правило $A \to w$, если $\varepsilon \in \texttt{FIRST}(w)$ и во входном потоке символ, который может следовать за A. Поэтому также необходимо построить **множество FOLLOW() для** A, которое определяется как множество терминалов a, таких что существует строка символов $\alpha A a \beta$, которая может быть получена из начального символа.

Вычисление множеств FOLLOW() для нетерминалов в грамматике выполняется следующим образом:

1. Инициализировать $\texttt{FOLLOW}(S) = \{\$\}$, а все остальные множества $\texttt{FOLLOW}(A_i)$ пустыми.
2. Если есть правило формы $A_j \to w A_i w'$, тогда:
   - Если терминал $a \in \texttt{FIRST}(w')$, то добавить a в $\texttt{FOLLOW}(A_i)$
   - Если $\varepsilon \in \texttt{FIRST}(w')$, то добавить $\texttt{FOLLOW}(A_j)$ в $\texttt{FOLLOW}(A_i)$

- Если w' имеет длину 0, то добавить `FOLLOW`(A_j) в `FOLLOW`(A_i)

3. Повторять шаг 2, пока в множествах `FOLLOW()` происходят изменения.

Теперь можно точно определить, какие правила будут содержаться в таблице синтаксического анализа. Если $T[A, a]$ обозначает ячейку в таблице для нетерминального A и терминала a, то:

- $T[A, a]$ содержит правило $A \to w$ тогда и только тогда, когда:
  - $a \in$ `FIRST`(w) при проходе правила $A \to w$, или
  - $\varepsilon \in$ `FIRST`(w) и $a \in$ `FOLLOW`(A)

Если таблица будет содержать не более одного правила в каждой ячейке, то синтаксический анализатор сможет однозначно определить, какое правило необходимо применить на каждом шаге разбора. В этом случае грамматику является **LL(1)** грамматикой.

2.6 Алгоритм разбора предложений

Перед запуском алгоритма разбора в стек помещаются два символа:

- Специальный символ \$ — признак конца входной цепочки.
- Начальный символ грамматики (на вершину стека).

Синтаксический анализатор выполняет три различных вида действий в зависимости от того, находится ли на вершине стека нетерминал, терминал или специальный символ \$.

1. **Если на вершине стека — нетерминал**, то в таблице синтаксического анализа ищется правило грамматики, находящееся на пересечении строки и столбца, соответствующих этому нетерминалу на вершине стека и текущему входному символу. Если же в указанной ячейке таблицы правило отсутствует — синтаксический анализатор останавливается и сообщает об ошибке.
2. **Если на вершине стека — терминал**, то синтаксический анализатор сравнивает его с текущим входным символом. Если они равны, то входной символ считывается, а соответствующий символ из вершины стека — удаляется.
3. **Если на вершине стека — специальный символ \$**, и текущий символ на ленте также равен \$, то синтаксический анализатор сообщает об успешном распознавании цепочки. В противном случае — сообщает об ошибке. В обоих случаях происходит останов анализатора.

Эти шаги повторяются до тех пор, пока не произойдёт останов. После останова мы получаем либо сообщение об ошибке, либо сообщение об успешном распознавании цепочки.

2.7 Алгоритм генерации предложения

Генерация предложения происходит следующим образом:

1. В список помещается начальный символ грамматики.
2. Осуществляется проход по списку. Первый встреченный нетерминал заменяется на его случайно выбранную продукцию, после чего проход начинается заново.
3. Алгоритм завершается, когда в списке не осталось нетерминалов.

Этот алгоритм не позволяет контролировать длину предложения. Также алгоритм может не сходиться в общем случае. Но вероятность того, что алгоритм попадет в петлю убывает геометрически с ростом длины цепочки, так что на практике этот метод работает.

3 Особенности реализации

3.1 Задание правил грамматики

Грамматика задаётся в виде текстового файла, в котором каждая строка соответствует одной продукции. Синтаксис задания продукций соответствует общепринятому, за исключением того, что вместо ε используется слово «`epsilon`». Терминалы записываются в кавычках, нетерминалы — без кавычек. Левая и правая части продукции разделяются двумя символами – «`->`». Таким образом содержимое файла, соответствует тексту описания грамматики в разделе 2.2.

3.2 Класс `Grammar`

Класс `Grammar` содержит в себе всю информацию о грамматике, а также методы для работы с ней. Код конструктора класса представлен в листинге 1. Конструктор класса принимает на вход строку, содержащую описание грамматики. Вызывает методы для парсинга продукций, нахождения терминалов, вычисления множеств FIRST и FOLLOW, заполнения таблицы синтаксического анализа.

Листинг 1. Код конструктора класса `Grammar`.

```
1  class Grammar:
2      EPSILON: str = "epsilon"
3
4      def __init__(self, text: str):
5          self.productions: OrderedDict[str, list[list[str]]] = OrderedDict()
6          self.start_symbol: str = ""
7          self._parse_productions(text)
8
9          self.terminals: set[str] = set()
10         self._find_terminals()
11
12         self.first_sets: dict[str, set[str]] = {}
13         self._calculate_first_sets()
14
15         self.follow_sets: dict[str, set[str]] = {}
16         self._calculate_follow_sets()
17
18         self.lookup_table: dict[str, dict[str, list[str]]] = {}
19         self._fill_lookup_table()
20
21         # Сопостовляем уникальный номер с каждым правилом
22         self.rule_numbers = {}
23         rule_idx = 1
24         for nt, rules in self.productions.items():
25             for rule in rules:
26                 self.rule_numbers[(nt, tuple(rule))] = rule_idx
27                 rule_idx += 1
```

3.2.1 Метод `_parse_productions`

Метод `_parse_productions` выполняет разбор текстового описания грамматики и создаёт внутреннее представление продукций. Код метода представлен в листинге 2.

Метод принимает ссылку на объект класса Grammar и строку text типа str, содержащую текстовое представление грамматики.

Метод не возвращает значений явно, но заполняет словарь productions и устанавливает переменную start_symbol в объекте класса.

Листинг 2. Код метода `_parse_productions`.

```
def _parse_productions(self, text: str):
    for line in text.splitlines():
        line = line.strip()
        if not line:
            continue

        non_terminal, rule = line.split("->")
        if self.start_symbol == "":
            self.start_symbol = non_terminal.strip()

        non_terminal = non_terminal.strip()
        rules = [
            [
                symbol.strip('"')
                for symbol in re.findall(r"\".*?\"|\S+", rule.strip())
                if symbol.strip('"') != self.EPSILON
            ]
            for rule in rule.split("|")
        ]

        if non_terminal not in self.productions:
            self.productions[non_terminal] = []

        self.productions[non_terminal].extend(rules)
```

3.2.2 Метод `_calculate_first_sets`

Метод `_calculate_first_sets` вычисляет множества FIRST для всех символов грамматики. Код метода представлен в листинге 3.

Метод принимает ссылку на объект класса Grammar.

Метод не возвращает значений явно, но заполняет словарь first_sets в объекте класса.

Листинг 3. Код метода `_calculate_first_sets`.

```
def _calculate_first_sets(self):
    # Инициализация FIRST для всех символов
    for non_terminal in self.productions:
        self.first_sets[non_terminal] = set()

```

```
    # Для терминалов FIRST содержит только сам терминал
    for terminal in self.terminals:
        self.first_sets[terminal] = {terminal}

    # Вычисление FIRST для нетерминалов
    changed = True
    while changed:
        changed = False
        for non_terminal, rules in self.productions.items():
            for rule in rules:
                if not rule:  # Пустое правило эпсилон()
                    if "" not in self.first_sets[non_terminal]:
                        self.first_sets[non_terminal].add("")
                        changed = True
                else:
                    all_can_derive_epsilon = True
                    for i, symbol in enumerate(rule):
                        # Добавляем все символы из FIRST(symbol) кроме эпсилон
                        first_without_epsilon = self.first_sets[symbol] - {""}
                        old_size = len(self.first_sets[non_terminal])
                        self.first_sets[non_terminal].update(first_without_epsilon)
                        if len(self.first_sets[non_terminal]) > old_size:
                            changed = True

                        # Если symbol не может порождать эпсилон, прерываем
                        if "" not in self.first_sets[symbol]:
                            all_can_derive_epsilon = False
                            break

                    # Если все символы в правиле могут порождать эпсилон,
                    # то и нетерминал может порождать эпсилон
                    if (
                        all_can_derive_epsilon
                        and "" not in self.first_sets[non_terminal]
                    ):
                        self.first_sets[non_terminal].add("")
                        changed = True
```

3.2.3 Метод `_calculate_follow_sets`

Метод `_calculate_follow_sets` вычисляет множества FOLLOW для нетерминальных символов грамматики. Код метода представлен в листинге 4.

Метод принимает ссылку на объект класса Grammar.

Метод не возвращает значений явно, но заполняет словарь follow_sets в объекте класса.

Листинг 4. Код метода `_calculate_follow_sets`.

```
def _calculate_follow_sets(self):
    # инициализировать Fo(S) = { $ }, а все остальные Fo(Ai) пустыми множествами
    for non_terminal in self.productions:
        self.follow_sets[non_terminal] = set()

    # Добавляем символ конца строки $ для начального символа
```

```
7       self.follow_sets[self.start_symbol].add("$")
8
9       # Повторяем, пока в наборах Follow происходят изменения
10      changed = True
11      while changed:
12          changed = False
13
14          # Для каждого нетерминала Aj в грамматике
15          for non_terminal_j, rules in self.productions.items():
16
17              # Для каждого правила Aj → w
18              for rule in rules:
19
20                  # Для каждого символа в правиле
21                  for i, symbol in enumerate(rule):
22
23                      # Если символ – нетерминал Ai
24                      if symbol in self.productions:
25                          # w' – остаток правила после Ai
26                          remainder = rule[i + 1 :] if i + 1 < len(rule) else []
27
28                          # Если есть терминалы после Ai (w' не пусто)
29                          if remainder:
30                              # Вычисляем First(w')
31                              first_of_remainder = self._first_of_sequence(remainder)
32
33                              # Если терминал a находится в First(w'), то добавляем a к Follow(Ai)
34                              for terminal in first_of_remainder - {""}:
35                                  if terminal not in self.follow_sets[symbol]:
36                                      self.follow_sets[symbol].add(terminal)
37                                      changed = True
38
39                              # Если epsilon находится в First(w'), то добавляем Follow(Aj) к Follow(Ai)
40                              if "" in first_of_remainder:
41                                  old_size = len(self.follow_sets[symbol])
42                                  self.follow_sets[symbol].update(
43                                      self.follow_sets[non_terminal_j]
44                                  )
45                                  if len(self.follow_sets[symbol]) > old_size:
46                                      changed = True
47
48                          # Если w' пусто (Ai в конце правила), то добавляем Follow(Aj) к Follow(Ai)
49                          else:
50                              old_size = len(self.follow_sets[symbol])
51                              self.follow_sets[symbol].update(
52                                  self.follow_sets[non_terminal_j]
53                              )
54                              if len(self.follow_sets[symbol]) > old_size:
55                                  changed = True
```

3.2.4 Метод `_first_of_sequence`

Метод `_first_of_sequence` вычисляет множество FIRST для последовательности символов. Код метода представлен в листинге 5.

Метод принимает ссылку на объект класса Grammar и список символов sequence.

Метод возвращает множество (set) строк, представляющих FIRST для заданной последовательности.

Листинг 5. Код метода `_first_of_sequence`.

```
def _first_of_sequence(self, sequence):
    """Вычисляет множество FIRST для последовательности символов"""
    if not sequence:
        return {""}

    result = set()
    all_can_derive_epsilon = True

    for symbol in sequence:
        # Добавляем First(symbol) без эпсилон к результату
        result.update(self.first_sets[symbol] - {""})

        # Если symbol не может порождать эпсилон, останавливаемся
        if "" not in self.first_sets[symbol]:
            all_can_derive_epsilon = False
            break

    # Если все символы могут порождать эпсилон, добавляем эпсилон к результату
    if all_can_derive_epsilon:
        result.add("")

    return result
```

3.2.5 Метод `_fill_lookup_table`

Метод `_fill_lookup_table` заполняет таблицу синтаксического анализа для LL(1)-грамматики. Код метода представлен в листинге 6.

Метод принимает ссылку на объект класса Grammar.

Метод не возвращает значений явно, но заполняет словарь lookup_table в объекте класса или вызывает исключение, если грамматика не является LL(1).

Листинг 6. Код метода `_fill_lookup_table`.

```
def _fill_lookup_table(self):
    # Для каждого нетерминала A
    for non_terminal, rules in self.productions.items():
        # Формируем таблицу синтаксического анализа
        self.lookup_table[non_terminal] = {}

        # Для каждого правила A → w
        for rule in rules:
            # Вычисляем First(w)
            first_of_rule = self._first_of_sequence(rule)

            # Для каждого терминала a в First(w)
            for terminal in first_of_rule - {""}:
                # Если T[A,a] уже содержит правило, грамматика не LL(1)
```

```
15                 if terminal in self.lookup_table[non_terminal]:
16                     raise ValueError(
17                         "Грамматика не является LL(1)грамматикой.\n"
18                         f"Конфликт в ячейке [{non_terminal}, {terminal}]\n"
19                         f"Новое правило: {non_terminal} -> {' '.join(rule)};\n"
20                         f"Существующее правило: {non_terminal} -> {self.lookup_table[non_terminal][
    terminal]}"
21                     )
22
23                 # Добавляем правило в таблицу
24                 self.lookup_table[non_terminal][terminal] = rule
25
26             # Если эпсилон в First(w), то для каждого b в Follow(A)
27             if "" in first_of_rule:
28                 for terminal in self.follow_sets[non_terminal]:
29                     # Если T[A,b] уже содержит правило, грамматика не LL(1)
30                     if terminal in self.lookup_table[non_terminal]:
31                         raise ValueError(
32                             "Грамматика не является LL(1)грамматикой.\n"
33                             f"Конфликт в ячейке [{non_terminal}, {terminal}]\n"
34                             f"Новое правило: {non_terminal} -> {' '.join(rule)};\n"
35                             f"Существующее правило: {non_terminal} -> {self.lookup_table[
    non_terminal][terminal]}"
36                         )
37
38                     # Добавляем правило в таблицу
39                     self.lookup_table[non_terminal][terminal] = rule
```

3.2.6 Метод `format_rules`

Метод `format_rules` форматирует все правила грамматики для удобного представления. Код метода представлен в листинге 7.

Метод принимает ссылку на объект класса Grammar.

Метод возвращает строку (str), содержащую все правила грамматики с их номерами.

Листинг 7. Код метода `format_rules`.

```
1  def format_rules(self) -> str:
2      result = []
3      sorted_rules = sorted(self.rule_numbers.items(), key=lambda x: x[1])
4
5      for rule, number in sorted_rules:
6          non_terminal, symbols = rule
7          rule_text = f"{number}: {non_terminal} -> {' '.join(symbols)}"
8          result.append(rule_text)
9
10     return "\n".join(result)
```

3.2.7 Метод `format_lookup_table`

Метод `format_lookup_table` форматирует таблицу синтаксического анализа для удобного представления. Код метода представлен в листинге 8.

Метод принимает ссылку на объект класса Grammar.

Метод возвращает строку (str), содержащую таблицу синтаксического анализа в виде таблицы с использованием библиотеки PrettyTable.

Листинг 8. Код метода `format_lookup_table`.

```
def format_lookup_table(self) -> str:
    table = PrettyTable()
    terminals = list(self.terminals)
    table.field_names = [""] + terminals + ["$"]

    for non_terminal in self.productions:
        row = [non_terminal]
        for terminal in terminals + ["$"]:
            if terminal in self.lookup_table[non_terminal]:
                rule = self.lookup_table[non_terminal][terminal]
                rule_num = self.rule_numbers.get((non_terminal, tuple(rule)), "")
                row.append(f"{rule_num}: {' '.join(rule)}")
            else:
                row.append(" - ")
        table.add_row(row)
    return str(table)
```

3.2.8 Метод `format_first_sets`

Метод `format_first_sets` форматирует множества FIRST для удобного представления. Код метода представлен в листинге 9.

Метод принимает ссылку на объект класса Grammar.

Метод возвращает строку (str), содержащую множества FIRST для всех символов грамматики.

Листинг 9. Код метода `format_first_sets`.

```
def format_first_sets(self) -> str:
    """Форматирует множества FIRST в читаемый вид."""
    result = []
    result.append("Множества FIRST:")
    result.append("=" * 40)

    # Сортируем для гарантии порядка вывода
    for symbol in sorted(self.first_sets.keys()):
        # Заменяем пустую строку на эпсилон для лучшей читаемости
        first_set = {
            self.EPSILON if item == "" else item for item in self.first_sets[symbol]
        }
        result.append(f"FIRST({symbol}) = {{{', '.join(sorted(first_set))}}}")

    return "\n".join(result)
```

3.2.9 Метод `format_follow_sets`

Метод `format_follow_sets` форматирует множества FOLLOW для удобного представления. Код метода представлен в листинге 10.

Метод принимает ссылку на объект класса Grammar.

Метод возвращает строку (str), содержащую множества FOLLOW для всех нетерминалов грамматики.

Листинг 10. Код метода `format_follow_sets`.

```
def format_follow_sets(self) -> str:
    """Форматирует множества FOLLOW в читаемый вид."""
    result = []
    result.append("Множества FOLLOW:")
    result.append("=" * 40)

    # Обрабатываем только нетерминалы
    for non_terminal in sorted(self.productions.keys()):
        follow_set = self.follow_sets.get(non_terminal, set())
        result.append(
            f"FOLLOW({non_terminal}) = {{{', '.join(sorted(follow_set))}}}"
        )

    return "\n".join(result)
```

3.2.10 Метод `analyze`

Метод `analyze` выполняет синтаксический анализ входной последовательности токенов с использованием LL(1)-алгоритма. Код метода представлен в листинге 11.

Метод принимает ссылку на объект класса Grammar и список input_tokens типа list[str], представляющий последовательность входных токенов.

Метод возвращает список (list) целых чисел, содержащий номера применённых правил в процессе анализа.

Листинг 11. Код метода `analyze`.

```
def analyze(self, input_tokens: list[str]) -> list[int]:
    input_tokens = input_tokens.copy()
    input_tokens += ["$"]
    input_pos = 0

    # Инициализируем стек с терминальным символом и начальным символом
    stack = ["$", self.start_symbol]

    rules_applied = []

    while stack:
        top = stack[-1]

        current_symbol = (
            input_tokens[input_pos] if input_pos < len(input_tokens) else "$"
        )
```

```
17
18          # Случай 1: Верхний символ стека — нетерминал
19          if top in self.productions:
20              # Ищем правило в таблице синтаксического анализа
21              if current_symbol in self.lookup_table[top]:
22                  # Получаем правило
23                  production = self.lookup_table[top][current_symbol]
24
25                  # Удаляем нетерминал из стека
26                  stack.pop()
27
28                  # Добавляем правило в rules_applied
29                  rule_number = self.rule_numbers[(top, tuple(production))]
30                  rules_applied.append(rule_number)
31
32                  # Добавляем правило в стек в обратном порядке
33                  for symbol in reversed(production):
34                      stack.append(symbol)
35              else:
36                  expected_symbols = list(self.lookup_table[top].keys())
37                  raise ValueError(
38                      f"Syntax error: expected one of {expected_symbols}, got '{current_symbol}'"
39                  )
40
41          # Случай 2: Верхний символ стека — терминал
42          elif top != "$":
43              if top == current_symbol:
44                  # Удаляем терминал из стека
45                  stack.pop()
46                  # Переходим к следующему символу ввода
47                  input_pos += 1
48              else:
49                  raise ValueError(
50                      f"Syntax error: expected '{top}', got '{current_symbol}'"
51                  )
52
53          # Случай 3: Верхний символ стека — $
54          else:  # top == "$"
55              if current_symbol == "$":
56                  # Успешный синтаксический анализ
57                  stack.pop()  # Удаляем $ из стека
58              else:
59                  raise ValueError(
60                      f"Syntax error: unexpected symbols at end of input: '{current_symbol}'"
61                  )
62
63      return rules_applied
```

3.2.11 Метод `generate`

Метод `generate` генерирует случайное предложение по заданной грамматике. Код метода представлен в листинге 12.

Метод принимает ссылку на объект класса Grammar и необязательный параметр symbol типа str или None, указывающий начальный символ для генерации.

Метод возвращает кортеж (tuple), содержащий список терминалов и список номеров применённых правил.

Листинг 12. Код метода `generate`.

```
def generate(self, symbol: str | None = None) -> tuple[list[str], list[int]]:
    """Генерирует предложение по заданной грамматике. Возвращает список терминалов
    и список номеров применённых правил."""

    if symbol is None:
        return self.generate(self.start_symbol)

    # Если символ — терминал, возвращаем его
    if symbol not in self.productions:
        return [symbol], []

    # Выбираем случайное правило для нетерминала
    rules = self.productions[symbol]
    chosen_rule = random.choice(rules)

    # Получаем номер выбранного правила
    rule_number = self.rule_numbers[(symbol, tuple(chosen_rule))]

    # Инициализируем результаты
    terminals = []
    rule_numbers = [rule_number]

    # Разворачиваем каждый символ в правой части правила
    for s in chosen_rule:
        sub_terminals, sub_rules = self.generate(s)
        terminals.extend(sub_terminals)
        rule_numbers.extend(sub_rules)

    return terminals, rule_numbers
```

3.2.12 Метод `generate_derivation_steps`

Метод `generate_derivation_steps` преобразует список номеров правил в последовательность шагов вывода. Код метода представлен в листинге 13.

Метод принимает ссылку на объект класса Grammar и список rule_numbers типа list[int], содержащий номера правил для применения.

Метод возвращает список (list) строк, представляющий каждый шаг вывода предложения.

Листинг 13. Код метода `generate_derivation_steps`.

```
def generate_derivation_steps(self, rule_numbers: list[int]) -> list[str]:
    """Преобразует список номеров правил в последовательность шагов вывода.
    Возвращает список строк, представляющих каждый шаг вывода."""

    # Получаем соответствие между номерами правил и самими правилами
    rule_details = {num: rule for rule, num in self.rule_numbers.items()}

    # Начинаем с начального символа
```

```
    current = self.start_symbol
    steps = [current]

    # Применяем каждое правило по порядку
    for rule_num in rule_numbers:
        if rule_num in rule_details:
            non_terminal, replacement = rule_details[rule_num]

            # Находим первое вхождение нетерминала и заменяем его
            words = current.split()
            for i, word in enumerate(words):
                if word == non_terminal:
                    words[i : i + 1] = replacement
                    break

            current = " ".join(words)
            steps.append(current)

    return steps
```

3.3 Функция `main`

Функция `main` реализует интерактивную консольную оболочку для взаимодействия с пользователем. Код функции представлен в листинге 14.

Листинг 14. Код функции `main`.

```
def main():
    print("Программа для работы с LL(1)грамматиками—")
    print("=" * 60)
    print("Варианты команд:")
    print(" — load файл<> — загрузить грамматику из файла по( умолчанию grammar.txt)")
    print(" — check строка<> — проверить, соответствует ли строка грамматике")
    print(" — generate — сгенерировать случайную строку по грамматике")
    print(" — exit — выход из программы")
    print("=" * 60)

    # Загружаем грамматику по умолчанию при старте
    grammar = load_grammar()

    while True:
        command = input("\Введите команду: ").strip()

        if not command:
            continue

        parts = command.split(maxsplit=1)
        cmd = parts[0].lower()

        if cmd == "exit":
            print("Выход из программы.")
            break

        elif cmd == "load":
```

```
28              filename = "grammar.txt"
29              if len(parts) > 1:
30                  filename = parts[1].strip()
31              grammar = load_grammar(filename)
32
33          elif cmd == "check":
34              input_string = ""
35              if len(parts) > 1:
36                  input_string = parts[1].strip()
37              check_string(grammar, input_string)
38
39          elif cmd == "generate":
40              generate_string(grammar)
41
42          else:
43              print(f"Неизвестная команда: {cmd}")
44              print("Доступные команды: load, check, generate, exit")
```

4 Результаты работы программы

Результаты работы программы представлены на Рис. 2.

```
Программа для работы с LL(1)-грамматиками
============================================================
Варианты команд:
 - load <файл> - загрузить грамматику из файла (по умолчанию grammar.txt)
 - check <строка> - проверить, соответствует ли строка грамматике
 - generate - сгенерировать случайную строку по грамматике
 - exit - выход из программы
============================================================
Правила грамматики с номерами сохранены в grammar_rules.txt
Таблица синтаксического анализа сохранена в grammar_lookup_table.txt
Множества FIRST сохранены в grammar_first.txt
Множества FOLLOW сохранены в grammar_follow.txt
Грамматика успешно загружена из файла grammar.txt

Введите команду: generate
Сгенерированная строка: Abendessen, welches die Abendessen aß, unter Suppe kam nicht.
Применённые правила: [1, 2, 4, 6, 17, 20, 22, 68, 8, 29, 6, 17, 18, 47, 22, 68, 9, 97, 31, 23, 72, 69, 99, 11, 30]
Подробный результат генерации сохранен в generation_result.txt

Введите команду: check Mein alt Freund, welcher ich liebte, schrieb gestern einen schön Brief.
Проверка строки: 'Mein alt Freund, welcher ich liebte, schrieb gestern einen schön Brief.'
Результат: Строка соответствует грамматике
Подробный результат анализа сохранен в analysis_result.txt

Введите команду: check
Проверка строки: ''
Результат: Строка не соответствует грамматике
Ошибка: Syntax error: expected one of ['Abendessen', 'einem', 'morgen', 'das', 'auf', 'ihre', 'leise', 'einer', 'am',
'freundlich', 'er', 'schlecht', 'euer', 'wir', 'alt', 'die', 'sie', 'Brief', 'fleißig', 'einen', 'dein', 'Frau', 'schn
ell', 'früh', 'Freund', 'unter', 'mein', 'süß', 'Sie', 'ihr', 'ich', 'laut', 'sein', 'der', 'gestern', 'eine', 'ruhig'
, 'Suppe', 'mit', 'immer', 'langsam', 'damals', 'spät', 'zu', 'jetzt', 'es', 'in', 'ein', 'jung', 'Mann', 'groß', 'heu
te', 'gern', 'Buch', 'gut', 'aus', 'schön', 'klein', 'unser', 'du', 'Kind', 'für'], got '$'

Введите команду: exit
Выход из программы.
```

Рис. 2. Результаты работы программы.

```
Введите команду: abrakadabra
Неизвестная команда: abrakadabra
Доступные команды: load, check, generate, exit

Введите команду: load nonexist.txt
Ошибка: Файл nonexist.txt не найден
```

Рис. 3. Реакция программы на некорректный пользовательский ввод.

На Рис. 3 представлена реакция программы на некорректный пользовательский ввод.

Заключение

В ходе выполнения лабораторной работы была построена контекстно-свободная грамматика для подмножества немецкого языка, описывающая простое прошедшее время Претерит. На основе разработанной грамматики была реализована программа, которая проверяет принадлежность входной строки заданному языку и генерирует случайные корректные предложения. Для анализа предложений использовался алгоритм LL(1)-разбора, основанный на построении множеств FIRST и FOLLOW для всех нетерминалов грамматики и создании таблицы синтаксического анализа.

Из достоинств выполнения лабораторной работы можно выделить возможность задания грамматики в отдельном текстовом файле, что позволяет легко изменять и расширять её без модификации программного кода. Также программа автоматически проверяет, что введенная грамматика является LL(1)-грамматикой. В противном случае, программа выводит сообщение об ошибке, в указывается на конкретные правила грамматики, между выбором которых возникает неоднозначность.

К недостаткам текущей реализации можно отнести ограниченность словарного запаса, что сужает разнообразие генерируемых предложений. Также алгоритм генерации не контролирует длину предложений, что может приводить к избыточно длинным или коротким конструкциям. В текущей версии система не учитывает некоторые грамматические особенности немецкого языка, например, склонение прилагательных и согласование артиклей с родом существительных.

Функционал программы несложно масштабировать. Грамматику легко расширять, добавляя новые слова и правила в текстовый файл без необходимости изменения программного кода. Класс Grammar может служить хорошей основой для создания полноценного LL(k) анализатора.

На выполнение лабораторной работы ушло около 12 часов. Работа была выполнена в среде разработки Visual Studio Code. Программа написана на Python версии 3.13.

Список литературы

[1] Востров, А.В. Курс лекций по дисциплине «Математическая логика». URL https://tema.spbstu.ru/compiler/ (дата обращения 01.04.2025 г.)

[2] Лутц, М. Изучаем Python. 5-е изд. / М. Лутц. — СПб.: Питер, 2019. — 1216 с.