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 №5
«Реализация LL(1)-анализатора»
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

Лабораторная №5 заключается в следующем. Для заданной вариантом грамматики необходимо:

- Построить множества FIRST и FOLLOW для каждого нетерминала.
- Построить таблицу выбора (lookup table).
- Реализовать детерминированный левый анализатор (проверка принадлежности цепочки грамматике).
- Назначить семантические действия части заданных продукций (например, генерация машинноориентированного кода).

Исходная грамматика в варианте 15 состояла из следующих продукций:

```
S -> B A b
A -> a A B C | b B | a
B -> b
C -> c A
```

Однако, как будет показано в разделе математического описания, данная грамматика не является LL(1)-грамматикой из-за неоднозначности в выборе продукций для нетерминала A. Поэтому в работе используется модифицированная версия грамматики с удаленной продукцией $A \to a$.

1 Математическое описание

1.1 Анализ исходной грамматики и её модификация

Исходная грамматика варианта 15 содержала следующие продукции:

```
S -> B A b
A -> a A B C | b B | a
B -> b
C -> c A
```

При попытке построения LL(1)-анализатора для данной грамматики обнаруживается неоднозначность в таблице синтаксического анализа. Конкретно, при рассмотрении нетерминала A и входного терминала a возникает конфликт между двумя продукциями:

- $A \to a$
- $A \to aABC$

Обе продукции начинаются с терминала a, что означает, что $a \in \texttt{FIRST}(a)$ и $a \in \texttt{FIRST}(aABC)$. Следовательно, анализатор не может однозначно определить, какую продукцию применить при встрече символа a на входе и нетерминала A на вершине стека.

Для получения корректной LL(1)-грамматики была выполнена модификация: удалена продукция $A \to a$. Таким образом, итоговая грамматика, используемая в данной работе, имеет вид:

```
S -> B A b
A -> a A B C | b B
B -> b
C -> c A
```

Данная модификация устраняет неоднозначность и позволяет построить корректную таблицу синтаксического анализа для LL(1)-анализатора. Язык, порождаемый модифицированной грамматикой, остается достаточно выразительным для демонстрации принципов работы LL(1)-анализатора.

1.2 Иерархия грамматик Хомского

Иерархия Хомского — классификация формальных языков и формальных грамматик, согласно которой они делятся на 4 типа по их условной сложности. На Рис. 1 представлены все четыре типа грамматик (по типу продукций, из которых они состоят) и отношения между ними.

Любая автоматная грамматика является контекстно-свободной, любая контекстно-свободная грамматика является контекстно-зависимой, любая контекстно-зависимая грамматика является неограниченной.

Тип	Вид правил	Название
0	$\alpha \to \beta$	Неограниченные грамматики
1	$\alpha A \beta \to \alpha \gamma \beta$	Контекстно-зависимые грамматики (КЗ-грамматики)
2	$A \to \gamma$	Контекстно-свободные грамматики (КС-граммаатики)
3	$A \to aB$ $A \to a$ $A \to \varepsilon$	Автоматные грамматики (А- грамматики)

Рис. 1. Иерархия грамматик Хомского

1.3 Контекстно-свободные грамматики

Контекстно-свободные грамматики являются частным случаем формальных грамматик, в которых левые части всех продукций содержат только один нетерминальный символ. Это означает, что все продукции имеют вид $A \to \beta$, где A — нетерминальный символ, а β — произвольная цепочка терминалов и нетерминалов.

Модифицированная грамматика этого варианта лабораторной работы задаётся в виде следующего списка продукций:

```
S -> B A b
A -> a A B C | b B
B -> b
C -> c A
```

Она, очевидно, является контекстно-свободной, так как в левых частях всех продукций присутствует только один нетерминальный символ. Также можно сразу заметить, что грамматика не является автоматной, так как в списке есть продукции, в правых частях которых присутствуют несколько нетерминалов.

Язык, порождаемый грамматикой, очевидно, не сложнее контекстно-свободного. Также, по продукции `A -> a A B C`, можно сделать вывод, что язык не является автоматным. Эта продукция является рекурсивной и порождает вложенную структуру, которая не может быть описана конечным автоматом.

Пример дерева вывода для цепочки языка, порождаемого грамматикой, представлен на Рис. 2.

1.4 LL(k)-анализаторы и LL(k)-грамматики

Синтаксический LL-анализатор – это нисходящий синтаксический анализатор для некоторого подмножества контекстно-свободных грамматик, известных как LL-грамматики. При этом не все контекстно-свободные грамматики являются LL-грамматиками.

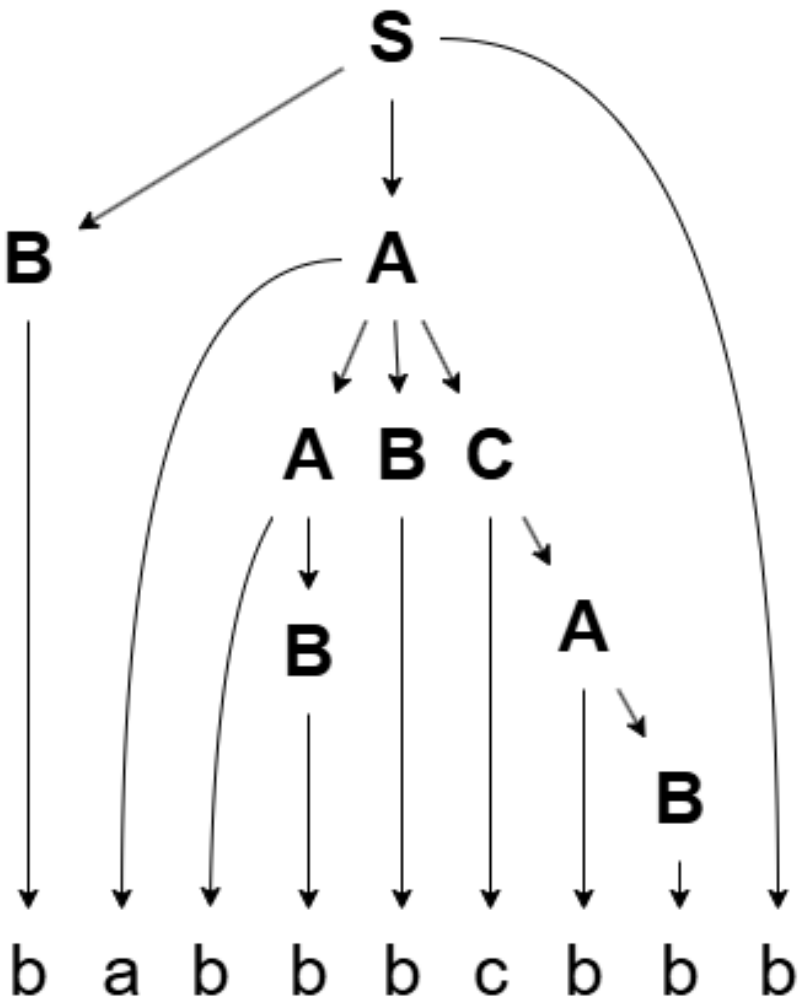

Рис. 2. Дерево вывода для цепочки *babbbcbbb*.

Буквы L в выражении «LL-анализатор» означают, что входная строка анализируется слева направо (left to right), и при этом строится её левосторонний вывод (leftmost derivation).

LL-анализатор называется LL(k)-анализатором, если данный анализатор использует предпросмотр на k токенов (лексем) при разборе входного потока. Грамматика, которая может быть распознана LL(k)-анализатором без возвратов к предыдущим символам, называется LL(k)-грамматикой. Язык, который может быть представлен в виде LL(k)-грамматики, называется LL(k)-языком.

Как будет показано далее, грамматика, рассматриваемая в этом варианте лабораторной работы является LL(1)-грамматикой, так как для неё можно однозначно заполнить таблицу LL(1) синтаксического анализа. Это означает, что для распознавания любой цепочки грамматики, анализатору достаточно просматривать входной поток только на один символ вперёд для принятия решения о том, какое правило грамматики необходимо применить.

1.5 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, где \$ — специальный терминал, служащий для указания конца входной цепочки.
2. Если есть правило формы $A_j \to wA_iw'$, 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 $\texttt{FOLLOW}(A_j)$ в $\texttt{FOLLOW}(A_i)$

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:
  - $a \in \texttt{FIRST}(w)$ при проходе правила $A \to w$, или
  - $\varepsilon \in \texttt{FIRST}(w)$ и $a \in \texttt{FOLLOW}(A)$

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.

1.6 Множества FIRST и FOLLOW для заданной грамматики

Приведём множества FIRST и FOLLOW для всех нетерминалов заданной грамматики.

Множества FIRST:

```
FIRST(S) = {b}
FIRST(A) = {a, b}
FIRST(B) = {b}
FIRST(C) = {c}
```

Множества FOLLOW:

```
FOLLOW(S) = {$}
FOLLOW(A) = {b}
FOLLOW(B) = {a, b, c}
FOLLOW(C) = {b}
```

1.7 Таблица синтаксического анализа для заданной грамматики

Таблица синтаксического анализа для заданной грамматики представлена в таблице 1.

Таблица 1. Таблица синтаксического анализа для заданной грамматики.

	a	**b**	**c**	**$**
S	-	B A b	-	-
A	a A B C	b B	-	-
B	-	b	-	-
C	-	-	c A	-

Если при распознавании цепочки синтаксический анализатор встречает ячейку с прочерком, то это означает, что входная цепочка не принадлежит языку, порождаемому грамматикой. В таком случае синтаксический анализатор завершает работу и возвращает сообщение об ошибке.

1.8 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.

1.9 Алгоритм генерации цепочек

Генерация цепочек 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.

1.10 Семантические действия

Каждой продукции можно сопоставить некоторое семантическое действие, которое будет выполняться при применении этой продукции, во время распознавания цепочки. Таким образом, можно, например, генерировать машинный код.

В данной лабораторной работе в качестве семантических действий решено было использовать генерацию инструкций для Java Virtual Machine. JVM это стековая виртуальная машина, её инструкции изменяют содержимое стека. Примеры инструкций:

- `iconst_1` — положить целочисленное значение 1 на вершину стека.
- `isub` — вычесть верхнее целочисленное значение стека из предыдущего и положить результат на вершину стека.
- `iadd` — сложить два верхних целочисленных значения стека и положить результат на вершину стека.

При распознавании цепочки можно, например, генерировать инструкции, в результате выполнения которых на вершине стека окажется следующее целочисленное значение:

$$\#b - 2 \times \#a + 3 \times \#c$$

где $\#a$, $\#b$, $\#c$ — это количества вхождений символов a, b, c в цепочку соответственно.

Для этого продукции были сопоставлены со следующими инструкциями:

- $S \to BAb \rightsquigarrow$ `iconst_1`
- $A \to aABC \rightsquigarrow$ `iconst_2 isub`
- $A \to bB \rightsquigarrow$ `iconst_1 iadd`
- $B \to b \rightsquigarrow$ `iconst_3 iadd`
- $C \to cA \rightsquigarrow$ `iconst_3 iadd`

Тогда при левом выводе цепочки `babbbcbbb`, будет сгенерирована следующая последовательность инструкций:

```
iconst_1      # [1] - значение на вершине стека
iconst_1 iadd # [2]
iconst_2 isub # [0]
iconst_1 iadd # [1]
iconst_1 iadd # [2]
iconst_1 iadd # [3]
iconst_3 iadd # [6]
iconst_1 iadd # [7]
iconst_1 iadd # [8]
```

Значение на вершине стека в конце выполнения всех инструкций равно 8. Поскольку в цепочке `babbbcbbb` 7 символов b, 1 символ a и 1 символ c, то и значение выражения $\#b - 2 \times \#a + 3 \times \#c$ для этой цепочки равно 8.

2 Особенности реализации

2.1 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`» и любые символы грамматики должны быть разделены пробелами.

2.2 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 и объект типа `Callable[[int, tuple[str, list[str]]], None]` – функцию-колбэк, которая будет вызываться при применении продукции. Эта функция должна принимать два аргумента: номер продукции и кортеж, содержащий левую и правую части продукции. Далее конструктор 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__(
5       self,
6       text: str,
7       semantic_action: Callable[[int, tuple[str, list[str]]], None] | None = None,
8   ):
9       self.productions: OrderedDict[str, list[list[str]]] = OrderedDict()
10      self.start_symbol: str = ""
11      self._parse_productions(text)
12
13      self.terminals: set[str] = set()
14      self._find_terminals()
15
16      self.first_sets: dict[str, set[str]] = {}
17      self._calculate_first_sets()
18
19      self.follow_sets: dict[str, set[str]] = {}
20      self._calculate_follow_sets()
21
22      self.lookup_table: dict[str, dict[str, list[str]]] = {}
23      self._fill_lookup_table()
24
25      # Сопостовляем уникальный номер с каждым правилом
26      self.rule_numbers = {}
27      rule_idx = 1
28      for nt, rules in self.productions.items():
29          for rule in rules:
```

```
30                 self.rule_numbers[(nt, tuple(rule))] = rule_idx
31                 rule_idx += 1
32
33         # Semantic action callback
34         self.semantic_action = semantic_action
```

2.2.1 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`.

```
1  def _parse_productions(self, text: str):
2      for line in text.splitlines():
3          line = line.strip()
4          if not line:
5              continue
6
7          non_terminal, rule = line.split("->")
8          if self.start_symbol == "":
9              self.start_symbol = non_terminal.strip()
10
11         non_terminal = non_terminal.strip()
12         rules = [
13             [
14                 symbol.strip('"')
15                 for symbol in re.findall(r"\".*?\"|\S+", rule.strip())
16                 if symbol.strip('"') != self.EPSILON
17             ]
18             for rule in rule.split("|")
19         ]
20
21         if non_terminal not in self.productions:
22             self.productions[non_terminal] = []
23
24         self.productions[non_terminal].extend(rules)
```

2.2.2 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
```

2.2.3 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
    self.follow_sets[self.start_symbol].add("$")

    # Повторяем, пока в наборах Follow происходят изменения
    changed = True
    while changed:
        changed = False

        # Для каждого нетерминала Aj в грамматике
        for non_terminal_j, rules in self.productions.items():

            # Для каждого правила Aj → w
            for rule in rules:

                # Для каждого символа в правиле
                for i, symbol in enumerate(rule):

                    # Если символ – нетерминал Ai
                    if symbol in self.productions:
                        # w' – остаток правила после Ai
                        remainder = rule[i + 1 :] if i + 1 < len(rule) else []

                        # Если есть терминалы после Ai (w' не пусто)
                        if remainder:
                            # Вычисляем First(w')
                            first_of_remainder = self._first_of_sequence(remainder)

                            # Если терминал a находится в First(w'), то добавляем a к Follow(Ai)
                            for terminal in first_of_remainder - {""}:
                                if terminal not in self.follow_sets[symbol]:
                                    self.follow_sets[symbol].add(terminal)
                                    changed = True

                            # Если epsilon находится в First(w'), то добавляем Follow(Aj) к Follow(Ai)
                            if "" in first_of_remainder:
                                old_size = len(self.follow_sets[symbol])
                                self.follow_sets[symbol].update(
                                    self.follow_sets[non_terminal_j]
                                )
                                if len(self.follow_sets[symbol]) > old_size:
                                    changed = True

                        # Если w' пусто (Ai в конце правила), то добавляем Follow(Aj) к Follow(Ai)
                        else:
                            old_size = len(self.follow_sets[symbol])
                            self.follow_sets[symbol].update(
                                self.follow_sets[non_terminal_j]
                            )
                            if len(self.follow_sets[symbol]) > old_size:
                                changed = True
```

2.2.4 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
```

2.2.5 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)
```

```
10            first_of_rule = self._first_of_sequence(rule)
11
12            # Для каждого терминала a в First(w)
13            for terminal in first_of_rule - {""}:
14                # 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
```

2.2.6 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)
```

2.2.7 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)
```

2.2.8 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)
```

2.2.9 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)
```

2.2.10 Метод `analyze`

Метод `analyze` выполняет синтаксический анализ входной последовательности токенов с использованием LL(1)-алгоритма. После применения продукции метод вызывает функцию-колбэк, представляющую семантические действия, если она была задана при создании объекта класса Grammar. 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]

```

```
14          current_symbol = (
15              input_tokens[input_pos] if input_pos < len(input_tokens) else "$"
16          )
17
18          # Случай 1: Верхний символ стека — нетерминал
19          if top in self.productions:
20              # Ищем правило в таблице синтаксического анализа
21              if current_symbol in self.lookup_table[top]:
22                  # Получаем правило
23                  production = self.lookup_table[top][current_symbol]
24
25                  # Удаляем нетерминал из стека
26                  stack.pop()
27
28                  # Добавляем правило в rules_applied
29                  rule_number = self.rule_numbers[(top, tuple(production))]
30                  rules_applied.append(rule_number)
31
32                  # Execute semantic action if provided
33                  if self.semantic_action:
34                      self.semantic_action(rule_number, (top, production))
35
36                  # Добавляем правило в стек в обратном порядке
37                  for symbol in reversed(production):
38                      stack.append(symbol)
39              else:
40                  expected_symbols = list(self.lookup_table[top].keys())
41                  raise ValueError(
42                      f"Syntax error: expected one of {expected_symbols}, got '{current_symbol}'"
43                  )
44
45          # Случай 2: Верхний символ стека — терминал
46          elif top != "$":
47              if top == current_symbol:
48                  # Удаляем терминал из стека
49                  stack.pop()
50                  # Переходим к следующему символу ввода
51                  input_pos += 1
52              else:
53                  raise ValueError(
54                      f"Syntax error: expected '{top}', got '{current_symbol}'"
55                  )
56
57          # Случай 3: Верхний символ стека — $
58          else:  # top == "$"
59              if current_symbol == "$":
60                  # Успешный синтаксический анализ
61                  stack.pop()  # Удаляем $ из стека
62              else:
63                  raise ValueError(
64                      f"Syntax error: unexpected symbols at end of input: '{current_symbol}'"
65                  )
66
67      return rules_applied
```

2.2.11 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
```

2.2.12 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
```

2.3 Функция `load_grammar`

Функция `load_grammar` загружает грамматику из файла и возвращает объект класса `Grammar`. Функция принимает необязательный параметр filename типа str, указывающий имя файла с грамматикой. Функция возвращает None в случае ошибки и объект класса Grammar в случае успешной загрузки грамматики. Код функции представлен в листинге 14. Также в ней задаются семантические действия, которые будут выполняться при применении продукций, и сохраняются в файлы множества FIRST и FOLLOW и таблица синтаксического анализа.

Листинг 14. Код функции `load_grammar`.

```
def load_grammar(filename: str = "grammar.txt") -> Grammar | None:
    try:
        # #b - 2 * #a + 3 * #c
        actions = [
            lambda rule_number, applied_count, _: print(
                f"Rule #{rule_number} (applied x{applied_count} times): iconst_1"
            ),
            lambda rule_number, applied_count, _: print(
                f"Rule #{rule_number} (applied x{applied_count} times): iconst_2 isub"
            ),
            lambda rule_number, applied_count, _: print(
                f"Rule #{rule_number} (applied x{applied_count} times): iconst_1 iadd"
            ),
            lambda rule_number, applied_count, _: print(
```

```
15                  f"Rule␣#{rule_number}␣(applied␣x{applied_count}␣times):␣iconst_1␣iadd"
16              ),
17              lambda rule_number, applied_count, _: print(
18                  f"Rule␣#{rule_number}␣(applied␣x{applied_count}␣times):␣iconst_3␣iadd"
19              ),
20          ]
21
22          with open(filename, "r", encoding="utf-8") as file:
23              text = file.read()
24          grammar = Grammar(text, ActionsListWithAppliedCount(actions))
25
26          # Сохраняем информацию о грамматике в файлы
27          with open("grammar_rules.txt", "w", encoding="utf-8") as output_file:
28              output_file.write(grammar.format_rules())
29              print("Правила␣грамматики␣с␣номерами␣сохранены␣в␣grammar_rules.txt")
30
31          with open("grammar_lookup_table.txt", "w", encoding="utf-8") as output_file:
32              output_file.write(grammar.format_lookup_table())
33              print(
34                  "Таблица␣синтаксического␣анализа␣сохранена␣в␣grammar_lookup_table.txt"
35              )
36
37          with open("grammar_first.txt", "w", encoding="utf-8") as output_file:
38              output_file.write(grammar.format_first_sets())
39              print("Множества␣FIRST␣сохранены␣в␣grammar_first.txt")
40
41          with open("grammar_follow.txt", "w", encoding="utf-8") as output_file:
42              output_file.write(grammar.format_follow_sets())
43              print("Множества␣FOLLOW␣сохранены␣в␣grammar_follow.txt")
44
45          print(f"Грамматика␣успешно␣загружена␣из␣файла␣{filename}")
46          return grammar
47      except FileNotFoundError:
48          print(f"Ошибка:␣Файл␣{filename}␣не␣найден")
49          return None
50      except ValueError as e:
51          print(f"Ошибка␣при␣загрузке␣грамматики:␣{e}")
52          return None
53      except Exception as e:
54          print(f"Неизвестная␣ошибка:␣{e}")
55          return None
```

2.4 Функция `main`

Функция `main` реализует интерактивную консольную оболочку для взаимодействия с пользователем. Код функции представлен в листинге 15.

Листинг 15. Код функции `main`.

```
1  def main():
2      print("Программа␣для␣работы␣с␣LL(1)грамматиками−")
3      print("=" * 60)
4      print("Варианты␣команд:")
5      print("␣−␣load␣файл<>␣−␣загрузить␣грамматику␣из␣файла␣по(␣умолчанию␣grammar.txt)")
```

```
6     print("  check строка<>  проверить, соответствует ли строка грамматике")
7     print("  generate  сгенерировать случайную строку по грамматике")
8     print("  exit  выход из программы")
9     print("=" * 60)
10
11    # Загружаем грамматику по умолчанию при старте
12    grammar = load_grammar()
13
14    while True:
15        command = input("\Введитен команду: ").strip()
16
17        if not command:
18            continue
19
20        parts = command.split(maxsplit=1)
21        cmd = parts[0].lower()
22
23        if cmd == "exit":
24            print("Выход из программы.")
25            break
26
27        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")
```

3 Результаты работы программы

Результаты работы программы представлены на Рис. 3.

```
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: b a b b b c b b b
Применённые правила: [1, 4, 2, 3, 4, 4, 5, 3, 4]
Подробный результат генерации сохранен в generation_result.txt

Введите команду: check b b b b
Проверка строки: 'b b b b'
Rule #1 (applied x1 times): iconst_1
Rule #4 (applied x1 times): iconst_1 iadd
Rule #3 (applied x1 times): iconst_1 iadd
Rule #4 (applied x2 times): iconst_1 iadd
Результат: Строка соответствует грамматике
Применённые правила: [1, 4, 3, 4]
Подробный результат анализа сохранен в analysis_result.txt

Введите команду: check
Проверка строки: ''
Результат: Строка не соответствует грамматике
Ошибка: Syntax error: expected one of ['b'], got '$'

Введите команду: exit
Выход из программы.
```

Рис. 3. Результаты работы программы.

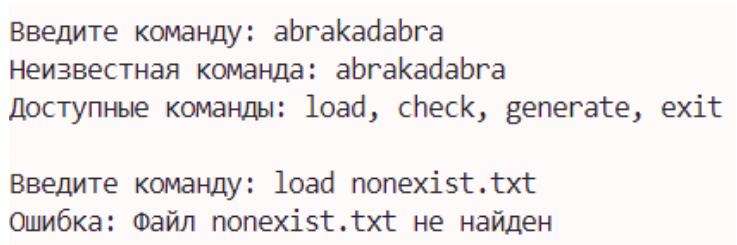

Рис. 4. Реакция программы на некорректный пользовательский ввод.

На Рис. 4 представлена реакция программы на некорректный пользовательский ввод.

Листинг 16. Содержимое файла `grammar_first.txt`.

```
Множества
 FIRST:
========================================
FIRST(A) = {a, b}
FIRST(B) = {b}
FIRST(C) = {c}
FIRST(S) = {b}
FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
```

Листинг 17. Содержимое файла `grammar_follow.txt`.

```
Множества
 FOLLOW:
========================================
FOLLOW(A) = {b}
FOLLOW(B) = {a, b, c}
FOLLOW(C) = {b}
FOLLOW(S) = {$}
```

Листинг 18. Содержимое файла `grammar_lookup_table.txt`.

```
+---+------------+----------+--------+-----+
|   |     a      |    b     |   c    |  $  |
+---+------------+----------+--------+-----+
| S |     -      | 1: B A b |   -    |  -  |
| A | 2: a A B C |  3: b B  |   -    |  -  |
| B |     -      |   4: b   |   -    |  -  |
| C |     -      |    -     | 5: c A |  -  |
+---+------------+----------+--------+-----+
```

В листингах 16, 17 и 18 представлено содержимое файлов `grammar_first.txt`, `grammar_follow.txt` и `grammar_lookup_table.txt` соответственно.

Заключение

В ходе выполнения лабораторной работы была реализована программа – синтаксический анализатор для LL(1)-грамматик. С его помощью для заданной грамматики были построены множества FIRST и FOLLOW для всех нетерминалов и таблица синтаксического анализа. Программа-анализатор также включает в себя функции для проверки принадлежности цепочки языку, заданному грамматикой, и генерации случайных цепочек. Также программа позволяет задать семантические действия, которые будут выполняться при применении продукций. Для демонстрации возможностей программы в качестве семантических действий решено было использовать генерацию инструкций для Java Virtual Machine.

Из достоинств выполнения лабораторной работы можно выделить, во-первых, возможность задания грамматики в отдельном текстовом файле, что позволяет легко изменять и расширять её без модификации программного кода. Во-вторых, программа автоматически проверяет, что введенная грамматика является LL(1)-грамматикой, и вычисляет множества FIRST и FOLLOW для всех нетерминалов, а также таблицу синтаксического анализа. В противном случае, программа выводит сообщение об ошибке, в котором указывается на конкретные правила грамматики, между выбором которых возникает неоднозначность. В третьих, программа позволяет легко задавать произвольные семантические действия в виде функций или объектов колбэков, которые будут вызываться при применении продукций.

К недостаткам текущей реализации можно отнести то, что алгоритм генерации не контролирует длину предложений, что может приводить к избыточно длинным или коротким цепочкам.

Функционал программы несложно масштабировать. Например, несложно добавить функцию для визуализации дерева вывода распознаваемой цепочки, пусть даже в текстовом виде, в классе Grammar уже есть все необходимые для этого данные. Кроме того, класс Grammar может служить хорошей основой для создания полноценного LL(k) анализатора.

На выполнение лабораторной работы ушло около 4 часов, основная часть кода была взята из лабораторной работы №3. 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.