МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление 02.03.01 Математика и компьютерные науки

Отчет по дисциплине «Математическая логика и теория автоматов»

Лабораторная работа №5

«Разработка LL(1) анализатора на основе заданной грамматики»

Вариант 19

Обучающийся: _____ Шклярова Ксения Алексеевна

Группа: 5130201/20102

Руководитель: _____ Востров Алексей Владимирович

«_____»_____20___г.

Санкт-Петербург, 2025

Содержание

Введение

Построить множества FIRST и FOLLOW для каждого нетерминала грамматики и таблицу выбора (lookup table). На их основе реализовать детерминированный левый анализатор (проверка принадлежности цепочки грамматике) и генератор цепочек. Назначить семантические действия части заданных продукций.

Грамматика для варианта 19:

S $\rightarrow$ SbBc|BdD

B $\rightarrow$ cD|ϵ

D $\rightarrow$ DBa|ϵ

1 Математическое описание

1.1 Классификация грамматик по Хомскому

Согласно Хомскому, формальные грамматики можно разделить на четыре типа. Для отнесения грамматики к тому или иному типу необходимо соответствие есех её правил (продукций) некоторым схемам.

Грамматика с фразовой структурой G - это алгебраическая структура, упорядоченная четвёрка ($\mathrm{V_T}, \mathrm{V_N}, \mathrm{P}, \mathrm{S}$), где:

- V_T - алфавит (множество) терминальных символов - терминалов,
- V_N - алфавит (множество) нетерминальных символов - нетерминалов,
- $V = V_T \cup V_N$ - словарь G, причём $V_T \cap V_N = \emptyset$
- P - конечное множество продукций (правил) грамматики, $P \subseteq V^+ \times V^*$
- S - начальный символ (источник).

Здесь V^* - множество всех строк над алфавитом V, а V^+- множество непустых строк над алфавитом V.

Тип 0 - неограниченные

К типу 0 по классификации Хомского относятся неограниченные грамматики — грамматики с фразовой структурой, то есть все без исключения формальные грамматики. Правила можно записать в виде:

$$\alpha \to \beta$$

где $\alpha \in V^+$- любая непустая цепочка, содержащая хотя бы один нетерминальный символ, а $\beta \in V^*$ - любая цепочка символов из алфавита.

Практического применения в силу своей сложности такие грамматики не имеют.

Тип 1 - контекстно-зависимые

К этому типу относятся контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики. Для грамматики $G\left(V_T, V_N, P, S\right), V = V_T \cup V_N$ все правила имеют вид:

- $\alpha A \beta \to \alpha \gamma \beta$, где $\alpha, \beta \in V^*, \gamma \in V^+, A \in V_N$. Такие грамматики относят к контекстно-зависимым.

- $\alpha \to \beta$, где $\alpha, \beta \in V^+, 1 \leq |\alpha| \leq |\beta|$. Такие грамматики относят к неукорачивающим. некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

Тип 2 - контекстно-свободные

К этому типу относятся контекстно-свободные (КС) грамматики. Для грамматики $G\left(V_T, V_N, P, S\right), V = V_T \cup V_N$ все правила имеют вид:

- $A \to \beta$, где $\beta \in V^+$(для неукорачивающих КС-грамматик) или $\beta \in V^*$ (для укорачивающих), $A \in V_N$. То есть грамматика допускает появление в левой части правила только нетерминального символа.

КС-грамматики широко применяются для описания синтаксиса компьютерных языков.

Тип 3 - регулярные

К третьему типу относятся регулярные грамматики (автоматные) - самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями.

Все регулярные грамматики могут быть разделены на два эквивалентных класса, которые для грамматики вида III будут иметь правила следующего вида:

- $A \to B\gamma$ или $A \to \gamma$, где $\gamma \in V_T^*, A, B \in V_N$ (для леволинейных грамматик).
- $A \to \gamma B$; или $A \to \gamma$, где $\gamma \in V_T^*, A, B \in V_N$ (для праволинейных грамматик).

Регулярные грамматики применяются для описания простейших конструкций: идентификаторов, строк, констант, а также языков ассемблера, командных процессоров и др.

В данной работе используются регулярные и контекстно-свободные грамматики.

1.2 LL(k) грамматики

LL(k) грамматики — это класс грамматик, для которых возможен нисходящий синтаксический анализ, при котором входная цепочка просматривается слева направо, и на каждом шаге анализатор может заглядывать вперед не более чем на k символов для восстановления левого канонического вывода данной терминальной цепочки.

Формально, грамматика является LL(k), если выполняются следующие условия:

- $S \to^* pA\beta \to p\alpha\beta \to^* py\eta$
- $S \to^* pA\beta \to p\alpha'\beta \to^* py\xi$

где p и y — это терминальные цепочки, представляющие уже разобранную часть слова w, A — нетерминал грамматики, для которого существуют правила $A \to \alpha$ и $A \to \alpha'$. При этом $\alpha, \alpha', \beta, \eta, \xi$ — последовательности из терминалов и нетерминалов. Если при условиях $|y| = k$ или $|y| < k$ и $\eta = \xi = \varepsilon$ следует равенство $\alpha = \alpha'$, то такая грамматика называется LL(k)-грамматикой.

Для построения анализатора LL(k) грамматик необходимо определить следующие множества:

- FIRST(α) — множество терминальных символов, с которых могут начинаться цепочки, выводимые из α. Если FIRST(α) $\cap$ FIRST(γ) $= \emptyset$, можно сделать однозначный выбор между правилами $A \to \alpha$ и $A \to \gamma$;

- FOLLOW(A) — множество терминальных цепочек, длиной до k символов, которые могут следовать за A в выводах. Если FIRST(α) $\cap$ FOLLOW(A) = $\emptyset$, можно сделать однозначный выбор между правилами $A \to \alpha$ и $A \to \varepsilon$.

LL(k) грамматики являются подмножеством контекстно-свободных грамматик и часто используются в построении синтаксических анализаторов, поскольку они позволяют эффективно и однозначно разбирать входные цепочки при помощи предсказательных парсеров.

1.2.1 LL(1) грамматики

LL(1) грамматики - подкласс LL(k) грамматик, где k = 1, а также:

- $A \to \alpha, A \to \beta, A \in N \to FIRST(\alpha) \cap FIRST(\beta) = \oslash$
- $A \to \alpha, A\ß \to \beta, A \in N, \varepsilon \in FIRST(\alpha) \to FOLLOW(A) \cap FIRST(\beta) = \oslash$

1.3 Заданная грамматика

1.3.1 Приведение заданной грамматики к форме LL(1)

Для приведения исходной грамматики к новой форме были выполнены следующие шаги:

1. Устранение левой рекурсии в правиле $S \to SbBc \mid BdD$

Исходное правило содержит левую рекурсию ($S \to SbBc$), поэтому применяем преобразование для её устранения:

- Вводим новый нетерминал S' и переписываем правила:

$$S \to BdDS'$$
$$S' \to bBcS' \mid \epsilon$$

2. Устранение неоднозначности в правиле $S' \to bBcS'$

B может раскрываться в cD или ϵ, что может вызывать неоднозначность. Вводим новый нетерминал B', чтобы явно разделить варианты:

- $S' \to bB'S'$
- $B' \to cS''$ (где S'' обрабатывает возможное продолжение)
- $S'' \to Dc \mid \epsilon$

3. Устранение левой рекурсии в правиле $D \to DBa \mid \epsilon$

Исходное правило содержит непосредственную левую рекурсию ($D \to DBa$), что требует преобразования. Процесс устранения выполняется в два этапа:

1. Стандартное преобразование левой рекурсии:

   Применяя классический метод устранения непосредственной левой рекурсии, вводим новый нетерминал D' и переписываем правила:

$$D \to D'$$
$$D' \to BaD' \mid \epsilon$$

   Однако после подстановки $B \to cD \mid \epsilon$ возникает проблема косвенной рекурсии через нетерминал B.

2. Упрощение грамматики:

   Поскольку B может порождать пустую цепочку (ϵ), преобразуем правило к итеративной форме, исключив промежуточный нетерминал:

$$D \to aD \mid \epsilon$$

Преобразованная грамматика

S -> BdDS'

S' -> bB'S' | ϵ

S" -> Dc | ϵ

B -> cD | ϵ

B' -> cS"

D -> aD | ϵ

1.3.2 Вывод таблицы выбора

Построим множество FIRST

FIRST(S) = FIRST(BdDS') = {d, c}

FIRST(S') = FIRST(bB'S') $\cup$ FIRST(ϵ) = {b, ϵ}

FIRST(S") = FIRST(Dc) $\cup$ FIRST(ϵ) = {a, c, ϵ}

FIRST(B) = FIRST(cD) $\cup$ FIRST(ϵ) = {c, ϵ}

FIRST(B') = FIRST(cS") = {c}

FIRST(D) = FIRST(aD) $\cup$ FIRST(ϵ) = {a, ϵ}

Построим множество FOLLOW

FOLLOW(S) = {$}

FOLLOW(S') = {$}

FOLLOW(S") = {b, $}

FOLLOW(B) = {d}

FOLLOW(B') = {b, $}

FOLLOW(D) = {b, c, d$}

Построим таблицу выбора

Таблица 1. Таблица выбора

	a	**b**	**c**	**d**	**$**
S	ошибка	ошибка	S -> BdDS'	S -> BdDS'	S -> BdDS'
S'	ошибка	S' -> bB'S'	ошибка	ошибка	S' -> ϵ
S''	S'' -> Dc	S'' -> ϵ	S'' -> Dc	ошибка	S'' -> ϵ
B	ошибка	ошибка	B -> cD	B -> ϵ	ошибка
B'	ошибка	B' -> cS''	B' -> cS''	ошибка	B' -> cS''
D	D -> aD	D -> ϵ	D -> ϵ	D -> ϵ	D -> ϵ

1.4 Пример вывода дерева цепочки

На рис. 1 представлен пример дерева вывода цепочки "cdaaa"в заданной грамматики.

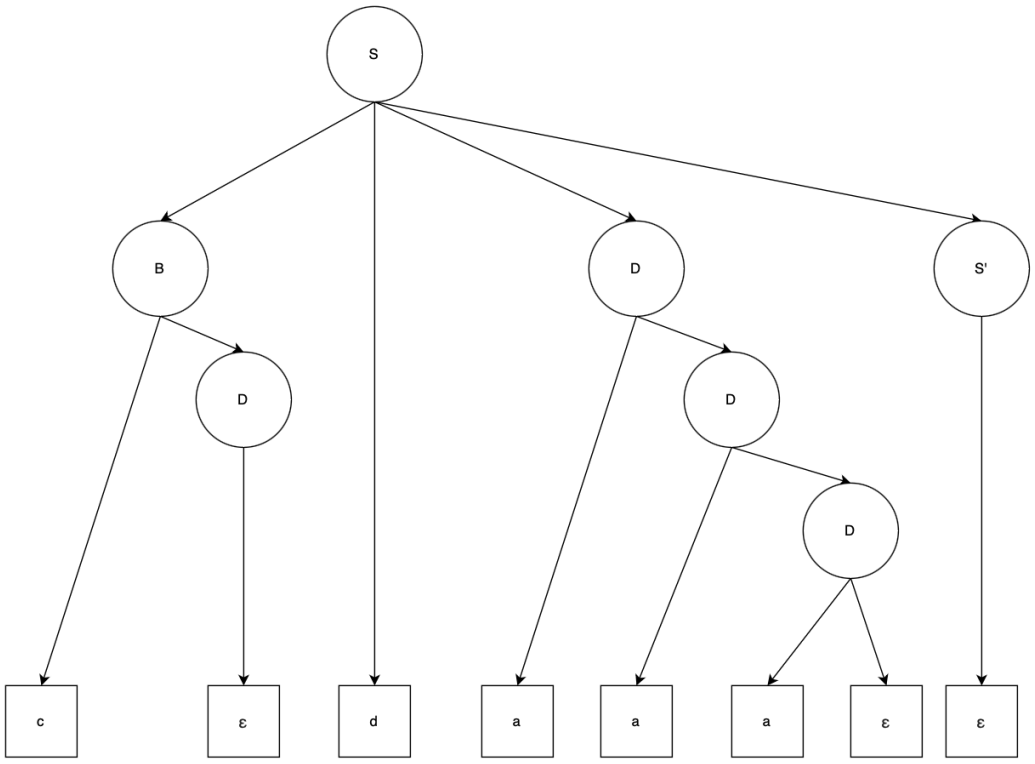

Рис. 1. Пример вывода цепочки

1.5 Заданные семантические действия

y1 = S → BdDS': "начать оформление посылки"

y2 = S' → bB'S': "проверить документы"

y3 = S' → : "завершить оформление"

y4 = S" → Dc: "проверить содержимое"

y5 = S" → : "подтвердить корректность"

y6 = B' → cS": "заполнить заявление"

y7 = B → cD: "внести платеж"

y8 = B → : "пропустить платеж"

y9 = D → aD: "добавить предмет"

y10 = D → : "завершить комплектацию"

2 Особенности реализации

2.1 Класс PackageProcessor

Этот класс представляет собой парсер для формальной грамматики, реализованной на основе LL(1)-таблицы. Он также моделирует бизнес-процесс оформления посылки: проверка документов, заполнение заявления, оплата, добавление предметов и т.д.

2.1.1 Инициализация внутренних данных и таблицы разбора

Метод __init__(self) создаёт LL(1)-таблицу разбора (parse_table) для заданной грамматики. Определяет множество нетерминалов (non_terminals) и начальный символ (start_symbol).

Таблица содержит правила вывода, текстовые описания правил и семантические действия. Каждый ключ — это нетерминал. Внутри каждого нетерминала — словарь, где ключ — входной символ, а значение — кортеж из правил вывода, текст правила, семантическая функция действия.

Реализацию инициализации см. в листинге 1.

Листинг 1. Реализация инициализации внутренних данных

```
    def __init__(self):
        self.parse_table = {
            'S': {
                'c': (['B', 'd', 'D', "S'"], "S     B d D S'", self.start_packing),
                'd': (['B', 'd', 'D', "S'"], "S     B d D S'", self.start_packing)
            },
            "S'": {
                'b': (['b', "B'", "S'"], "S'     b B' S'", self.check_documents),
                '$': ([''  '], "S'       ", self.finish_packing),
                'd': ([''  '], "S'       ", self.finish_packing),
                'c': ([''  '], "S'       ", self.finish_packing)
            },
            "S''": {
                'a': (['D', 'c'], "S''     D c", self.check_contents),
                'c': (['D', 'c'], "S''     D c", self.check_contents),
                'b': ([''  '], "S''       ", self.confirm_correctness),
                '$': ([''  '], "S''       ", self.confirm_correctness),
                'd': ([''  '], "S''       ", self.confirm_correctness)
            },
            "B'": {
                'c': (['c', "S''"], "B'     c S''", self.fill_application)
            },
            'B': {
                'c': (['c', 'D'], "B     c D", self.make_payment),
                'd': ([''  '], "B       ", self.skip_payment),
```

```
26                'a': (['  '], "B       ", self.skip_payment),
27                '$': (['  '], "B       ", self.skip_payment)
28            },
29            'D': {
30                'a': (['a', 'D'], "D     a D", self.add_item),
31                'b': (['  '], "D       ", self.finish_packing_list),
32                'c': (['  '], "D       ", self.finish_packing_list),
33                'd': (['  '], "D       ", self.finish_packing_list),
34                '$': (['  '], "D       ", self.finish_packing_list)
35            }
36        }
37        self.non_terminals = {'S', "S'", "S''", "B'", 'B', 'D'}
38        self.start_symbol = 'S'
```

2.1.2 Семантические методы

Эти методы связаны с правилами грамматики и моделируют действия в процессе оформления посылки. Все методы принимают на вход ссылку на текущий экземпляр класса, а результатом выполнения является вывод сообщения о действиях.

start_packing(self) выводит сообщение о начале оформления посылки.

check_documents(self) выводит сообщение о проверке документов.

finish_packing(self) выводит сообщение о завершении оформления посылки.

check_contents(self) выводит сообщение о проверке содержимого посылки.

confirm_correctness(self) выводит сообщение о подтверждении корректности данных.

fill_application(self) выводит сообщение о заполнении заявления.

make_payment(self) выводит сообщение о внесении платежа.

skip_payment(self) выводит сообщение о пропуске платёжа.

add_item(self) выводит сообщение о добавлении предмета в посылку.

finish_packing_list(self) выводит сообщение о завершении комплектации.

Реализацию данных методов см. в листинге 2.

Листинг 2. Реализация семантических методов

```
1     def start_packing(self):
2         print("начать оформление посылки")
3
4     def check_documents(self):
5         print("проверить документы")
6
7     def finish_packing(self):
8         print("завершить оформление")
9
```

```
10      def check_contents(self):
11          print("проверить содержимое")
12
13      def confirm_correctness(self):
14          print("подтвердить корректность")
15
16      def fill_application(self):
17          print("заполнить заявление")
18
19      def make_payment(self):
20          print("внести платеж")
21
22      def skip_payment(self):
23          print("пропустить платеж")
24
25      def add_item(self):
26          print("добавить предмет")
27
28      def finish_packing_list(self):
29          print(f"завершить комплектацию")
```

2.1.3 Анализатор цепочки

Метод parse реализует LL(1)-разбор входной строки согласно заданной грамматике.

Принимает на вход строку input_string (цепочка терминалов) и ссылку на текущий экземпляр класса. Возвращает True, если цепочка принадлежит языку грамматики, иначе False. Логика работы : сначала добавляется маркер конца строки $. Используется стек, начиная с начального символа грамматики. Пока стек не пуст: если вершина стека совпадает с текущим символом — удаляет из стека и переходит к следующему символу. Если вершина — нетерминал, ищет правило в parse_table по текущему символу и заменяет вершину стека на правую часть правила, применяя семантическое действие. При ошибке (нет правила или неожиданный символ) возвращается False.

Реализацию данного метода см. в листинге 3.

Листинг 3. Реализация анализатора цепочек

```
1       def parse(self, input_string):
2           input_string = input_string + '$'
3           stack = [self.start_symbol]
4           ptr = 0
5
6           while stack:
7               top = stack[-1]
8               current_char = input_string[ptr]
```

```
9
10              if top == current_char:
11                  stack.pop()
12                  ptr += 1
13              elif top in self.non_terminals:
14                  if current_char in self.parse_table[top]:
15                      production, rule, action = self.parse_table[top][current_char]
16                      print(f"{rule.ljust(15)} ", end="")
17                      stack.pop()
18                      if production != ['  ']:
19                          stack.extend(reversed(production))
20                      action()
21                  else:
22                      print(f"\nОшибка: нет правила для {top} с входным символом {
                            current_char}")
23                      return False
24              else:
25                  print(f"\nОшибка: неожиданный символ {current_char}")
26                  return False
27
28          return ptr == len(input_string) - 1
```

2.1.4 Генератор цепочек

Метод generate генерирует случайную цепочку, которая может быть распознана парсером. Принимает минимальную и максимальную длину цепочки, ссылку на текущий экземпляр класса. Возвращает сгенерированную строку, если она прошла парсинг; иначе None. Логика работы : начинает с начального символа. Случайным образом выбирает подходящие правила из parse_table. Раскрывает нетерминалы, пока стек не станет пустым. Проверяет, чтобы длина цепочки была в пределах [min_length, max_length]. Вызывает parse() для проверки корректности. Если всё верно — возвращает строку.

Реализацию данного метода см. в листинге 4.

Листинг 4. Реализация генератора цепочек

```
1  def generate(self, min_length=5, max_length=30):
2          for _ in range(1000):
3              stack = [self.start_symbol]
4              output = []
5              steps = 0
6
7              while stack and steps < max_length:
8                  steps += 1
```

```
            top = stack[-1]

            if top in self.non_terminals:
                possible_expansions = self.parse_table[top]
                possible_chars = list(possible_expansions.keys())

                if len(output) < min_length:
                    possible_chars = [c for c in possible_chars if
                        possible_expansions[c][0] != ['  ']]
                selected_char = random.choice(possible_chars)
                production, rule, action = possible_expansions[selected_char]
                stack.pop()
                if production != ['  ']:
                    stack.extend(reversed(production))
            else:
                terminal = stack.pop()
                if terminal != '  ':
                    output.append(terminal)

        generated_string = ''.join(output)

        if min_length <= len(generated_string) <= max_length and self.parse(
            generated_string):
            print(f"\nСгенерированная цепочка: {generated_string}")
            return generated_string

    print("Не удалось сгенерировать допустимую цепочку")
    return None
```

2.2 Пользовательское меню

Функция get_user_input() -> str запрашивает у пользователя ввод цепочки вручную, возвращает строку или None (если пользователь решил отменить ввод). На вход принимает введенную пользователем строку. Возвращаемое значение: str — введённая пользователем строка с цепочкой, None — если пользователь ввёл q для отмены.

Функция show_menu() отображает консольное меню и вызывает соответствующую функцию. На вход принимает введенное пользователем значение, результатом выполнения является вызов соответствующей функции.

Реализацию данных функций см. в листинге 5.

Листинг 5. Реализация пользовательского меню

```
def get_user_input() -> str:
```

```
    while True:
        user_input = input("\nВведите строку для проверки или 'q' для отмены: ").strip
            ()

        return user_input


def show_menu():
    while True:
        print("\nМеню:")
        print("1. Сгенерировать цепочку случайно")
        print("2. Ввести цепочку вручную")
        print("3. Выход")

        choice = input("Выберите действие (1-3): ")

        if choice == "1":
            min1 = random.randint(0, 9)
            processor.generate(min1, 30)
        elif choice == "2":
            date = get_user_input()
            if processor.parse(date):
                print("\nГотово!")
            else:
                print("Цепочка не принадлежит грамматике")
        elif choice == "3":
            print("Выход из программы.")
            break
        else:
            print("Некорректный ввод.")
```

3 Результаты работы программы

На рис. 2-6 показаны результаты работы программы.

```
Меню:
1. Сгенерировать цепочку случайно
2. Ввести цепочку вручную
3. Выход
Выберите действие (1-3): ц
Некорректный ввод.
```

Рис. 2. Неверный ввод при выборе пункта в меню

```
Меню:
1. Сгенерировать цепочку случайно
2. Ввести цепочку вручную
3. Выход
Выберите действие (1-3): 1
S → B d D S'    начать оформление посылки
B → c D         внести платеж
D → a D         добавить предмет
D → a D         добавить предмет
D → a D         добавить предмет
D → a D         добавить предмет
D → a D         добавить предмет
D → ε           завершить комплектацию
D → ε           завершить комплектацию
S' → b B' S'    проверить документы
B' → c S''      заполнить заявление
S'' → ε         подтвердить корректность
S' → b B' S'    проверить документы
B' → c S''      заполнить заявление
S'' → D c       проверить содержимое
D → ε           завершить комплектацию
S' → ε          завершить оформление

Сгенерированная цепочка: caaaaadbcbcc
```

Рис. 3. Генерация цепочки №1

```
Меню:
1. Сгенерировать цепочку случайно
2. Ввести цепочку вручную
3. Выход
Выберите действие (1-3): 1
S → B d D S'    начать оформление посылки
B → c D         внести платеж
D → a D         добавить предмет
D → a D         добавить предмет
D → a D         добавить предмет
D → a D         добавить предмет
D → ε           завершить комплектацию
D → ε           завершить комплектацию
S' → b B' S'    проверить документы
B' → c S''      заполнить заявление
S'' → ε         подтвердить корректность
S' → ε          завершить оформление

Сгенерированная цепочка: caaaadbc
```

Рис. 4. Генерация цепочки №2

```
Меню:
1. Сгенерировать цепочку случайно
2. Ввести цепочку вручную
3. Выход
Выберите действие (1-3): 2

Введите строку для проверки или 'q' для отмены:

Ошибка: нет правила для S с входным символом $
Цепочка не принадлежит грамматике
```

Рис. 5. Ввод и провекра цепочки

```
Меню:
1. Сгенерировать цепочку случайно
2. Ввести цепочку вручную
3. Выход
Выберите действие (1-3): 2

Введите строку для проверки или 'q' для отмены: d
S → B d D S'    начать оформление посылки
B → ε           пропустить платеж
D → ε           завершить комплектацию
S' → ε          завершить оформление

Готово!

Меню:
1. Сгенерировать цепочку случайно
2. Ввести цепочку вручную
3. Выход
Выберите действие (1-3): 2

Введите строку для проверки или 'q' для отмены: cadaa
S → B d D S'    начать оформление посылки
B → c D         внести платеж
D → a D         добавить предмет
D → ε           завершить комплектацию
D → a D         добавить предмет
D → a D         добавить предмет
D → ε           завершить комплектацию
S' → ε          завершить оформление

Готово!
```

Рис. 6. Ввод и проверка цепочки

Заключение

В рамках данной работы были выполнены преобразования исходной контекстно-свободной грамматики с целью приведения её к LL(1)-форме. Для всех нетерминальных символов грамматики были построены множества FIRST и FOLLOW, на основе которых была сформирована таблица выбора. На её основе реализован детерминированный левый анализатор типа LL(1), предназначенный для проверки принадлежности входных цепочек контекстно-свободному языку, порождаемому заданной грамматикой. В состав анализатора включены семантические действия для каждой продукции.

На основе таблицы выбора была создана программа, обеспечивающая:

• Генерацию строк, соответствующих заданной грамматике. Генератор использует случайный выбор правил грамматики, начиная с начального символа, для формирования цепочки терминальных символов. Для каждой сгенерированной строки проверяется её длина и соответствие синтаксическим правилам с использованием реализованного анализатора. В случае успешной проверки строка возвращается как результат. Генератор позволяет автоматизировать создание тестовых данных, соответствующих спецификации грамматики.

• Проверку принадлежности произвольной строки к языку, порождаемому грамматикой. Анализатор, построенный на основе LL(1)-грамматики, осуществляет детерминированный левый разбор входной строки. Для принятия решения о выборе правила вывода используются заранее вычисленные множества FIRST и FOLLOW, а также таблица синтаксического анализа, которая для каждой пары (нетерминальный символ, входной символ) указывает соответствующее правило вывода.

Достоинства: Грамматика представлена в виде таблицы разбора, что делает её наглядной и легко модифицируемой. Все правила вывода сопровождаются текстовым описанием (например, "S → B d D S'"), что упрощает отладку.

Недостатки: Грамматика зафиксирована в коде, и её изменение требует ручного редактирования parse_table. Нет поддержки более сложных конструкций (например, рекурсивных правил с условиями).

Масштабирование: можно добавить новые правила и нетерминалы для поддержки более сложных структур. Так же можно разрешить пользователю задавать грамматику в виде текстового файла.

Работа выполнена на языке программирования Python в среде разработки PyCharm версии 2023.3.4.

Список литературы

[1] Востров, А. В. Математическая логика

URL:https://tema.spbstu.ru/compiler (Дата обращения: 20.05.2025).

[2] Сети, Р.; Ахо, А. Компиляторы: принципы, технологии и инструменты / Р. Сети, А. Ахо. - М.: Издательство «Наука», 2006. - С. 104.