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

Лабораторная работа №5
«Реализация детерминированного левого анализатора для грамматики LL(1)»
по дисциплине «Теория алгоритмов»
Вариант 21

Обучающийся: _____ Яшнова Дарья Михайловна
группа 5130201/20002

Руководитель: _____ Востров Алексей Владимирович

«___»_____ 2025г

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). Назначить семантические действия части заданных продукций (например, генерация машинноориентированного кода).

Вариант 21.

$S \to BdA|cA$

$A \to Ac|a$

$B \to \varepsilon|bS$

1 LL(k)-грамматики

1.1 Основные определения

Грамматика $G = (V_N, V_T, P, S)$ называется **LL(k)-грамматикой**, если для любых двух левых выводов:

$$S \Rightarrow^* wA\alpha \Rightarrow w\beta\alpha \Rightarrow^* wx$$
$$S \Rightarrow^* wA\alpha \Rightarrow w\gamma\alpha \Rightarrow^* wy$$

где $x, y \in V_T^*$, выполняется условие:

$$\text{FIRST}_k(x) = \text{FIRST}_k(y) \implies \beta = \gamma$$

- Первая 'L' - чтение входной цепочки **слева направо** (Left-to-right)
- Вторая 'L' - построение **левого вывода** (Leftmost derivation)
- 'k' - количество символов **предпросмотра** (lookahead)

Расшифровка определения LL(k) грамматик представлена на рис.1.

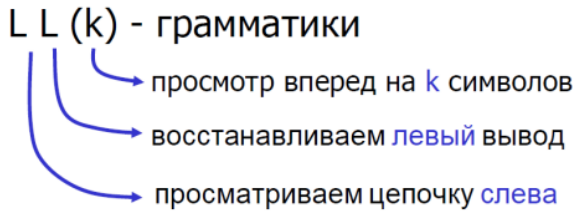

Рис. 1: Расшифровка терминологии LL(k)

1.2 Свойства LL(k)-грамматик

Грамматика является LL(k) тогда и только тогда, когда для каждого нетерминала A и любых двух различных продукций $A \to \alpha$ и $A \to \beta$ выполняется:

$$\text{FIRST}_k(\alpha\text{FOLLOW}_k(A)) \cap \text{FIRST}_k(\beta\text{FOLLOW}_k(A)) = \varnothing$$

2 LL(1)-грамматики

2.1 Критерий LL(1)

Грамматика является LL(1), если выполняются следующие условия для каждого нетерминала A с продукциями $A \to \alpha_1|\alpha_2|...|\alpha_n$:

1. $\text{FIRST}_1(\alpha_i) \cap \text{FIRST}_1(\alpha_j) = \varnothing$ для всех $i \neq j$
2. Если $\alpha_i \Rightarrow^* \varepsilon$, то $\text{FIRST}_1(\alpha_j) \cap \text{FOLLOW}_1(A) = \varnothing$ для всех $j \neq i$

2.2 Функции FIRST и FOLLOW для LL(1)

- $\text{FIRST}_1(\alpha) = \{a \in V_T \mid \alpha \Rightarrow^* a\beta\} \cup \{\varepsilon \mid \alpha \Rightarrow^* \varepsilon\}$
- $\text{FOLLOW}_1(A) = \{a \in V_T \mid S \Rightarrow^* \beta Aa\gamma\} \cup \{\$ \mid S \Rightarrow^* \beta A\}$

2.3 Алгоритм построения таблицы разбора

Для LL(1)-грамматики строится таблица разбора $M[A, a]$, где:

$$M[A, a] = \begin{cases} A \to \alpha, & \text{если } a \in \text{FIRST}_1(\alpha) \\ A \to \alpha, & \text{если } \varepsilon \in \text{FIRST}_1(\alpha) \text{ и } a \in \text{FOLLOW}_1(A) \\ \text{ошибка}, & \text{иначе} \end{cases}$$

2.4 Преимущества и недостатки LL(1)

+ Простота реализации нисходящих анализаторов

+ Высокая скорость разбора

- Ограниченный класс грамматик

- Необходимость устранения левой рекурсии и левой факторизации

3 Приведение грамматики

Исходная грамматика содержала левую рекурсию:
$S \to BdA|cA$
$A \to Ac|a$
$B \to \varepsilon|bS$

3.0.1 Для нетерминала A

Преобразуем правила для A:
$A \to aA_1$
$A_1 \to cA_1|\varepsilon$

3.1 Итоговая грамматика

После всех преобразований получаем:
$S \to BdA|cA$
$A \to aA_1$
$A_1 \to cA_1|\varepsilon$
$B \to \varepsilon|bS$
Множества First и Follow для грамматики представлены в таблице 1.

Таблица 1: Множества First и Follow для грамматики

Нетерминал	**First**	**Follow**
S	{d, c, b}	{$, d}
A	{a}	{$, d}
A_1	{c}	{$, d}
B	{b}	{d}

3.2 Таблица выбора (LL(1) Parsing Table)

Таблица выбора правил для данной грамматики представлена в таблице 2. На рис.2 пред-

Таблица 2: Таблица разбора

	S	**b**	**c**	**a**	**d**
S		$S \to B\,d\,A$	$S \to c\,A$		$S \to B\,d\,A$
A				$A \to a\,A_1$	
$\mathbf{A_1}$	$A_1 \to \epsilon$		$A_1 \to c\,A_1$		$A_1 \to \epsilon$
B		$B \to b\,S$			$B \to \epsilon$

ставлено дерево разбора для строки bcacbdac.

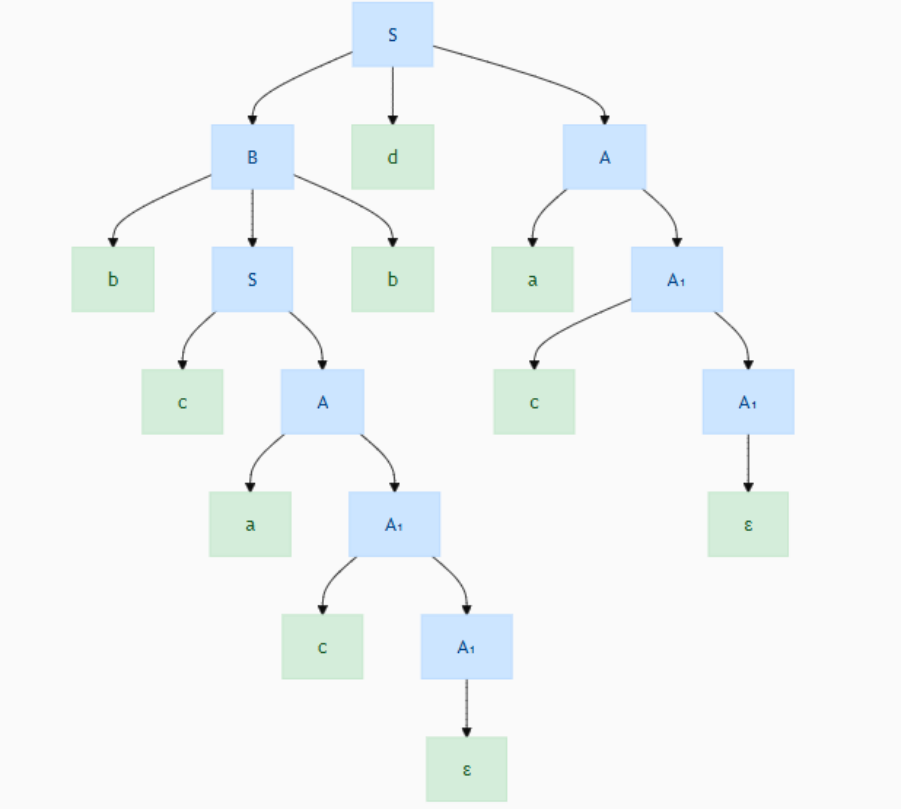

Рис. 2: Дерево разбора строки b c a c b d a c

3.3 Семантические действия

Для всех продукций были назначены следующие семантические действия:

```
S->c A - Магазин открыт
S->B d A - Новая поставка фруктов
A->a A - Продано 1 яблоко
A'->c A' - Продан 1 апельсин
A'->? - Продан 1 ананас
B->b S - Поставка принята
B->? - Новые фрукты выложены на витрину
```

4 Особеннности реализации

4.1 Структуры данных

- **Правила грамматики** (`rules`) - список строк в форме Бэкуса-Наура:

```
rules = ["S -> B d A | c A",
         "A -> a A'",
         "A' -> c A' | ?",
         "B -> b S  | ?"]
```

- **Словарь грамматики** (`diction`) - словарь, где ключи это нетерминалы, а значения - списки продукций:

```
diction = {
    'S': [['B', 'd', 'A'], ['c', 'A']],
    'A': [['a', "A'"]],
    "A'": [['c', "A'"], ['?']],
    'B': [['b', 'S'], ['?']]
}
```

- **Множества FIRST и FOLLOW**:

```
firsts = {
    'S': {'b', 'c', 'd'},
    'A': {'a'},
    "A'": {'?', 'c'},
    'B': {'?', 'b'}
}

follows = {
    'S': {'$', 'd'},
    'A': {'$', 'd'},
    "A'": {'$', 'd'},
    'B': {'d'}
}
```

4.1.1 Таблицы анализа

- **Таблица синтаксического анализа** (`parsing_table`) - матрица строк:

```
parsing_table = [
    ['', 'S->c A', 'S->B d A', 'S->B d A', ''],
    ["A->a A'", '', '', '', ''],
    ['', "A'->c A'", '', "A'->?", "A'->?"],
    ['', '', 'B->b S ', 'B->?', '']
]
```

- **Таблица семантического анализа** (`parsing_table_semantics`) - аналогичная структура с аннотациями:

```
parsing_table_semantics = [
    ['', 'S->c A - Магазин открыт', ...],
    ...
]
```

- **Список терминалов** (`tabTerm`) - список терминальных символов с $ в конце:

```
tabTerm = ['a', 'c', 'b', 'd', '$']
```

4.1.2 Рабочие структуры

- **Стек** (`stack`) - список символов, начинается со стартового символа и $:

```
stack = [start_symbol, '$']
```

- **Буфер** (`buffer`) - список символов входной строки (в обратном порядке) с $:

```
buffer = ['$'] + input_string.split()[::-1]
```

4.2 Функция `validateStringUsingStackBuffer`

Входные параметры:

- `parsing_table`: таблица разбора LL(1)-грамматики.
- `grammarll1`: флаг, указывающий, является ли грамматика LL(1).
- `table_term_list`: список терминалов, соответствующих столбцам таблицы.
- `input_string`: строка для проверки.
- `term_userdef`: пользовательские терминалы.
- `start_symbol`: стартовый нетерминал грамматики.

Выходные данные:

- Сообщение о результате разбора: `"Valid String!"` или `"Invalid String!"`.

Алгоритм:

1. Инициализация стека и буфера.
2. Цикл разбора:
    - Если стек и буфер пусты (кроме `$`), строка valid.
    - Если верх стека — нетерминал, ищем правило в таблице.
    - Если верх стека — терминал, сравниваем с буфером.

3. Вывод состояния на каждом шаге.

Листинг 1: Код функции validateStringUsingStackBuffer

```
1
2  def validateStringUsingStackBuffer(parsing_table, grammarll1,
3                                     table_term_list, input_string,
4                                     term_userdef,start_symbol):
5
6      print(f"\nValidate String => {input_string}\n")
7
8
9      if grammarll1 == False:
10         return f"\nInput String = " \
11                f"\"{input_string}\"\n" \
12                f"Grammar is not LL(1)"
13
14
15     stack = [start_symbol, '$']
16     buffer = []
17
18     input_string = input_string.split()
19     input_string.reverse()
20     buffer = ['$'] + input_string
21
22     print("{:>30} {:>30} {:>30}".
23           format("Buffer", "Stack","Action"))
24
25     while True:
26         if stack == ['$'] and buffer == ['$']:
27             print("{:>30} {:>30} {:>30}"
28                   .format(' '.join(buffer),
29                           ' '.join(stack),
30                           "Valid"))
31             return "\nValid String!"
32         elif stack[0] not in term_userdef:
33             x = list(diction.keys()).index(stack[0])
34             y = table_term_list.index(buffer[-1])
35             if parsing_table[x][y] != '':
36                 entry = parsing_table[x][y]
37                 entry2 = parsing_table_semantics[x][y]
38                 print("{:>30} {:>30} {:>30}".
39                       format(' '.join(buffer),
40                              ' '.join(stack),
41                              f"T[{stack[0]}][{buffer[-1]}] = {entry2}"))
42                 lhs_rhs = entry.split("->")
43                 lhs_rhs[1] = lhs_rhs[1].replace('?', '').strip()
44                 entryrhs = lhs_rhs[1].split()
45                 stack = entryrhs + stack[1:]
46             else:
47                 return f"\nInvalid String! No rule at " \
48                        f"Table[{stack[0]}][{buffer[-1]}]."
49         else:
50             if stack[0] == buffer[-1]:
51                 print("{:>30} {:>30} {:>30}"
52                       .format(' '.join(buffer),
53                               ' '.join(stack),
54                               f"Matched:{stack[0]}"))
55                 buffer = buffer[:-1]
56                 stack = stack[1:]
57             else:
58                 return "\nInvalid String! " \
```

```
                        "Unmatched terminal symbols"


sample_input_string = None


rules = ["S -> B d A | c A",
         "A -> a A'",
         "A' -> c A' | ?",
         "B -> b S  | ?"]

nonterm_userdef = ['A','S','B',"A'"]

term_userdef = ['a','c','b','d']


diction = {'S': [['B', 'd', 'A'], ['c', 'A']],
           'A': [['a', "A'"]],
           "A'": [['c', "A'"], ['?']],
           'B': [['b', 'S'], ['?']]}
firsts = {'S': {'b', 'c', 'd'},
          'A': {'a'},
          "A'": {'?', 'c'},
           'B': {'?', 'b'}}
follows = {'S': {'$', 'd'},
           'A': {'$', 'd'},
             "A'": {'$', 'd'},
             'B': {'d'}}
allowed_chars = {'a', 'b', 'c', 'd'}

start_symbol = list(diction.keys())[0]
parsing_table =  [['', 'S->c A', 'S->B d A', 'S->B d A', ''],
                ["A->a A'", '', '', '', ''],
                ['', "A'->c A'", '', "A'->?", "A'->?"],
                ['', '', 'B->b S', 'B->?', '']]

parsing_table_semantics =  [['', 'S->c A - Магазин открыт', 'S->B d A - Новая поставка фруктов', 'S->B d A - Новая поставка фруктов', ''],
                            ["A->a A - Продано 1 яблоко'", '', '', '', ''],
                            ['', "A'->c A' - Продан 1 апельсин", '', "A'->? - Продан 1 ананас", "A'->? - Продан 1 ананас"],
                            ['', '', 'B->b S - Поставка принята', 'B->? - Новые фрукты выложены на витрину', '']]


tabTerm = ['a', 'c', 'b', 'd', '$']


print("\nВведите строки для проверки. Для выхода введите 'q'.")
while True:
    user_input = input("\nВведите строку: ").strip()

    if user_input.lower() == 'q':
        print("\nЗавершение программы...")
        break

    if not set(user_input) <= allowed_chars:
        print("Invalid string, chars : a,b,c,d allowed only")
        continue
```

```
115
116     if not user_input:
117         print("empty string is invalid for this grammar")
118         continue
119
120     user_input = ' '.join(user_input)
121
122     validity = validateStringUsingStackBuffer(parsing_table, True,
            tabTerm, user_input, term_userdef, start_symbol)
123     print(validity)
```

4.3 Функции генератора

- `generate()`:
  - **Вход:** Максимальная глубина рекурсии, генератор случайных чисел.
  - **Выход:** Строка, соответствующая грамматике.
  - **Действие:** Запускает генерацию с нетерминала S.
- `S(depth)`:
  - **Вход:** Текущая глубина рекурсии.
  - **Выход:** Подстрока для $S \to B\ d\ A \mid c\ A$.
  - **Действие:** Рекурсивно генерирует B и A или c A.
- `B(depth)`:
  - **Вход:** Текущая глубина рекурсии.
  - **Выход:** Подстрока для $B \to b\ S \mid \varepsilon$.
  - **Действие:** Возвращает b S или пустую строку.
- `A(depth)`:
  - **Вход:** Текущая глубина рекурсии.
  - **Выход:** Подстрока для $A \to a\ A'$.
  - **Действие:** Всегда возвращает a A'.
- `A'(depth)`:
  - **Вход:** Текущая глубина рекурсии.
  - **Выход:** Подстрока для $A' \to c\ A' \mid \varepsilon$.
  - **Действие:** Возвращает c A' или пустую строку.

Листинг 2: Код генератора

```
1  import random
2
3  MAX_DEPTH = 20
4
5  def generate():
6      return S(0)
7
8  def S(depth):
9      if depth > MAX_DEPTH:
```

```
        return 'c' + A(depth + 1)
    else:
        if random.choice([0, 1]) == 0:
            return B(depth + 1) + 'd' + A(depth + 1)
        else:
            return 'c' + A(depth + 1)

def B(depth):
    if depth > MAX_DEPTH:
        return ''
    else:
        if random.choice([0, 1]) == 0:
            return 'b' + S(depth + 1)
        else:
            return ''

def A(depth):
    return 'a' + A_prime(depth + 1)

def A_prime(depth):
    if depth > MAX_DEPTH:
        return ''
    else:
        if random.choice([0, 1]) == 0:
            return 'c' + A_prime(depth + 1)
        else:
            return ''

if __name__ == "__main__":
    for _ in range(10):
        print(generate())
```

5 Результаты работы программы

Далее представлены результаты работы программы для строк, соответствующих грамматике (рис.3-6).

```
Validate String => b b c a d a c d a c c c c

                     Buffer                          Stack                          Action
   $ c c c c a d c a d a c b b                          S $ T[S][b] = S->B d A - Новая поставка фруктов
   $ c c c c a d c a d a c b b                      B d A $ T[B][b] = B->b S - Поставка принята
   $ c c c c a d c a d a c b b                    b S d A $                          Matched:b
     $ c c c c a d c a d a c b                      S d A $ T[S][b] = S->B d A - Новая поставка фруктов
     $ c c c c a d c a d a c b                  B d A d A $ T[B][b] = B->b S - Поставка принята
     $ c c c c a d c a d a c b                b S d A d A $                          Matched:b
       $ c c c c a d c a d a c                  S d A d A $ T[S][c] = S->c A - Магазин открыт
       $ c c c c a d c a d a c                c A d A d A $                          Matched:c
         $ c c c c a d c a d a                 A d A d A $ T[A][a] = A->a A - Продано 1 яблоко'
         $ c c c c a d c a d a               a A' d A d A $                          Matched:a
           $ c c c c a d c a d                A' d A d A $ T[A'][d] = A'->? - Продан 1 ананас
           $ c c c c a d c a d                   d A d A $                          Matched:d
             $ c c c c a d c a                    A d A $ T[A][a] = A->a A - Продано 1 яблоко'
             $ c c c c a d c a                  a A' d A $                          Matched:a
               $ c c c c a d c                    A' d A $ T[A'][c] = A'->c A' - Продан 1 апельсин
               $ c c c c a d c                  c A' d A $                          Matched:c
                 $ c c c c a d                    A' d A $ T[A'][d] = A'->? - Продан 1 ананас
                 $ c c c c a d                       d A $                          Matched:d
                   $ c c c c a                       A $ T[A][a] = A->a A - Продано 1 яблоко'
                   $ c c c c a                     a A' $                          Matched:a
                     $ c c c c                       A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                     $ c c c c                     c A' $                          Matched:c
                       $ c c c                       A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                       $ c c c                     c A' $                          Matched:c
                         $ c c                       A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                         $ c c                     c A' $                          Matched:c
                           $ c                       A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                           $ c                     c A' $                          Matched:c
                             $                       A' $ T[A'][$] = A'->? - Продан 1 ананас
                             $                          $                             Valid

Valid String!
```

Рис. 3: Вывод для строк, соответствующих грамматике

```
Validate String => b c a c c d a c c

                    Buffer                      Stack                          Action
          $ c c a d c c a c b                     S $ T[S][b] = S->B d A - Новая поставка фруктов
          $ c c a d c c a c b                   B d A $ T[B][b] = B->b S - Поставка принята
          $ c c a d c c a c b                 b S d A $                        Matched:b
            $ c c a d c c a c                   S d A $ T[S][c] = S->c A - Магазин открыт
            $ c c a d c c a c                 c A d A $                        Matched:c
              $ c c a d c c a                   A d A $ T[A][a] = A->a A - Продано 1 яблоко'
              $ c c a d c c a                 a A' d A $                        Matched:a
                $ c c a d c c                   A' d A $ T[A'][c] = A'->c A' - Продан 1 апельсин
                $ c c a d c c                 c A' d A $                        Matched:c
                  $ c c a d c                   A' d A $ T[A'][c] = A'->c A' - Продан 1 апельсин
                  $ c c a d c                 c A' d A $                        Matched:c
                    $ c c a d                   A' d A $ T[A'][d] = A'->? - Продан 1 ананас
                    $ c c a d                     d A $                        Matched:d
                      $ c c a                       A $ T[A][a] = A->a A - Продано 1 яблоко'
                      $ c c a                     a A' $                        Matched:a
                        $ c c                       A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                        $ c c                     c A' $                        Matched:c
                          $ c                       A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                          $ c                     c A' $                        Matched:c
                            $                       A' $ T[A'][$] = A'->? - Продан 1 ананас
                            $                         $                          Valid

Valid String!
```

Рис. 4: Вывод для строк, соответствующих грамматике

```
Validate String => c a c c

                      Buffer                    Stack                        Action
                    $ c c a c                     S $ T[S][c] = S->c A - Магазин открыт
                    $ c c a c                   c A $                        Matched:c
                      $ c c a                     A $ T[A][a] = A->a A - Продано 1 яблоко'
                      $ c c a                   a A' $                        Matched:a
                        $ c c                     A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                        $ c c                   c A' $                        Matched:c
                          $ c                     A' $ T[A'][c] = A'->c A' - Продан 1 апельсин
                          $ c                   c A' $                        Matched:c
                            $                     A' $ T[A'][$] = A'->? - Продан 1 ананас
                            $                       $                          Valid

Valid String!
```

Рис. 5: Вывод для строк, соответствующих грамматике

```
Validate String => c a

                     Buffer                     Stack                          Action
                     $ a c                        S $ T[S][c] = S->c A - Магазин открыт
                     $ a c                      c A $                        Matched:c
                       $ a                        A $ T[A][a] = A->a A - Продано 1 яблоко'
                       $ a                      a A' $                        Matched:a
                         $                        A' $ T[A'][$] = A'->? - Продан 1 ананас
                         $                          $                          Valid

Valid String!
```

Рис. 6: Вывод для строк, соответствующих грамматике

На рис.7 представлены результаты обработки строк, несоответствующих грамматике.

```
Validate String => a b a b a b c d

                        Buffer                          Stack                          Action

Invalid String! No rule at Table[S][a].

Validate String => b a b c

                        Buffer                          Stack                          Action
                     $ c b a b                            S $ T[S][b] = S->B d A - Новая поставка фруктов
                     $ c b a b                        B d A $ T[B][b] = B->b S - Поставка принята
                     $ c b a b                    b S d A $                      Matched:b

Invalid String! No rule at Table[S][a].


Validate String =>
empty string is invalid for this grammar


Validate String => aquqyh
Invalid string, chars : a,b,c,d allowed only
```

Рис. 7: Строки, несоответствующие грамматике

На рис.8 представлены результаты генерации строк.

```
cacc
bcacda
ca
bbcadacdacccc
bdadac
dacc
cac
```

Рис. 8: Генерация строк

Заключение

В ходе выполнения лабораторной работы была реализована программа для синтаксического анализа для заданной грамматики с использованием LL(1)-подхода. Данная грамматика принадлежит к контекстно-свободному типу

Успешно устранена левая рекурсия в исходной грамматике. Введен новый нетерминал A_1, что позволило преобразовать правила к форме, пригодной для LL(1)-анализа. Реализован левый детерминированный анализатор и некоторым продукциям присвоены семантические действия. Всем родукциям была присвоена семантика.

Недостатки реализации:

- Использование большого количества заранее посчитанных структур. Некоторые из структур дублируются (parsing_table), чего тоже можно было избежать.

Достоинства реализации: - Эффективность – алгоритм работает за линейное время (O(n)) относительно длины входной строки. В данном случае каждый символ входной строки обрабатывается ровно один раз, что даёт линейную зависимость от длины строки.

- Вывод ошибок достаточно информативен.

Масштабируемость:

Можно расширить анализатор для получения более естественных семантических действий (например, проведение транзакций или команды машинного кода), добавив некоторые продукции и семантику для них.

Список литературы

[1] Электронный ресурс ВШТИИ
URL:https://tema.spbstu.ru/compiler/
(Дата обращения: 13.04.2025)

[2] Карпов, Ю. Г. Теория автоматов, Санкт-Петербург : Питер, 2003.
URL:https://djvu.online/file/eeLVKnyRZPXfl
(Дата обращения: 17.04.2025)