МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №1

по дисциплине: Теория автоматов и формальных языков
тема: Формальные грамматики. Выводы

Выполнил: студент ПВ-233
Мороз Роман Алексеевич

Проверил:
Рязанов Юрий Дмитриевич

Белгород 2025 г.

Цель работы: изучить основные понятия теории формальных языков и грамматик.

Задания

1. Найти терминальную цепочку α, |α| > 10, для которой существует не менее двух левых выводов в заданной КС-грамматике (см. варианты заданий). Записать различные левые выводы этой цепочки. Построить деревья вывода. Определить последовательности правил, применяемые при этих выводах.

2. Написать программу, которая определяет, можно ли применить заданную последовательность правил при левом выводе терминальной цепочки в заданной КС-грамматике, формирует левый вывод и линейную скобочную форму дерева вывода. Обработать программой последовательности правил, полученные в п.1.

Примечание. Если к нетерминалу A в процессе вывода применяется правило с номером n, то в выводе и в линейной скобочной форме дерева вывода после нетерминала A должен быть символ с кодом n.

3. Найти последовательность правил p, |p| > 10, которую можно применить при произвольном выводе терминальной цепочки, но нельзя применить при левом или правом выводе в заданной КС-грамматике (см. варианты заданий).

Записать вывод v, в процессе которого применяется последовательность правил p. Построить дерево вывода. Записать левый и правый выводы, эквивалентные выводу v.

4. Написать программу, которая определяет, можно ли применить заданную последовательность правил p при выводе терминальной цепочки в заданной КС-грамматике и формирует линейную скобочную форму дерева вывода. Если последовательность правил p можно применить при выводе v терминальной цепочки, то программа должна вывести последовательность правил, применяемую при левом выводе, эквивалентном выводу v.

Вариант 9

1. S→SbSa
2. S→Sa
3. S→A
4. A→aS
5. A→aB
6. A→b
7. B→b
8. B→Aa

Задание 1

Левый вывод 1:

S ⇒ **S**bSa

⇒ **Sa**bSa (по правилу 2)

⇒ **Saa**bSa (по правилу 2)

⇒ **Saaa**bSa (по правилу 2)

⇒ **Saaaa**bSa (по правилу 2)

⇒ **Aaaaa**bSa (по правилу 3)

⇒ **aSaaaa**bSa (по правилу 4)

⇒ a**Aaaaa**bSa (по правилу 3)

⇒ a**b**aaaabSa (по правилу 6)

⇒ abaaaab**S**a

⇒ abaaaab**A**a (по правилу 3)

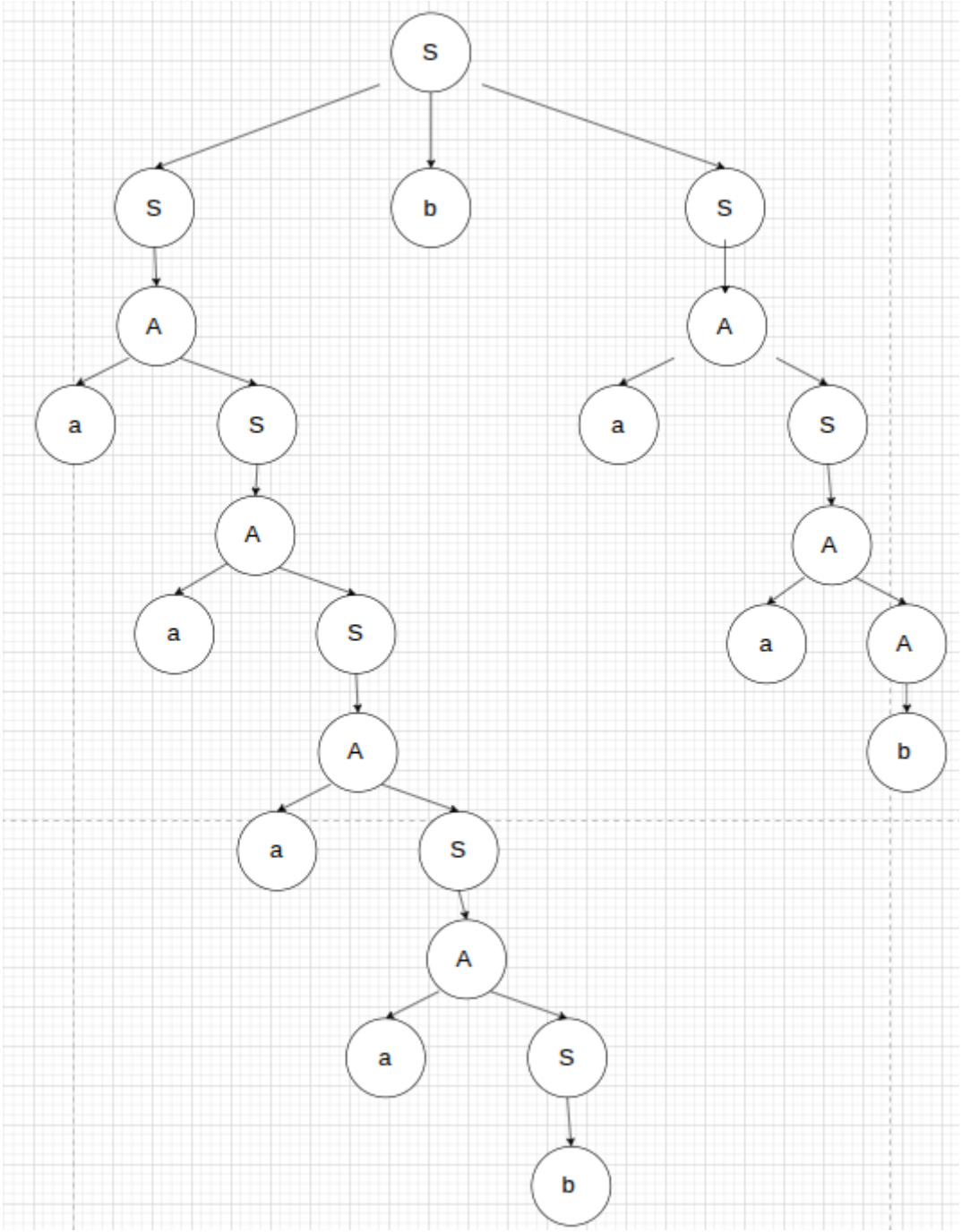
⇒ abaaaab**aS**a (по правилу 4)

⇒ abaaaaba**A**a (по правилу 3)

⇒ abaaaaba**b**a (по правилу 6)

Последовательность правил для вывода 1:

1, 2, 2, 2, 2, 3, 4, 3, 6, 3, 4, 3, 6



Левый вывод 2:

S ⇒ **S**bSa

⇒ **Sa**bSa (по правилу 2)

⇒ **Saa**bSa (по правилу 2)

⇒ **Saaa**bSa (по правилу 2)

⇒ **Saaaa**bSa (по правилу 2)

⇒ **Aaaaa**bSa (по правилу 3)

⇒ **aSaaaa**bSa (по правилу 4)

⇒ a**Aaaaa**bSa (по правилу 3)

⇒ a**b**aaaabSa (по правилу 6)

⇒ abaaaab**S**a

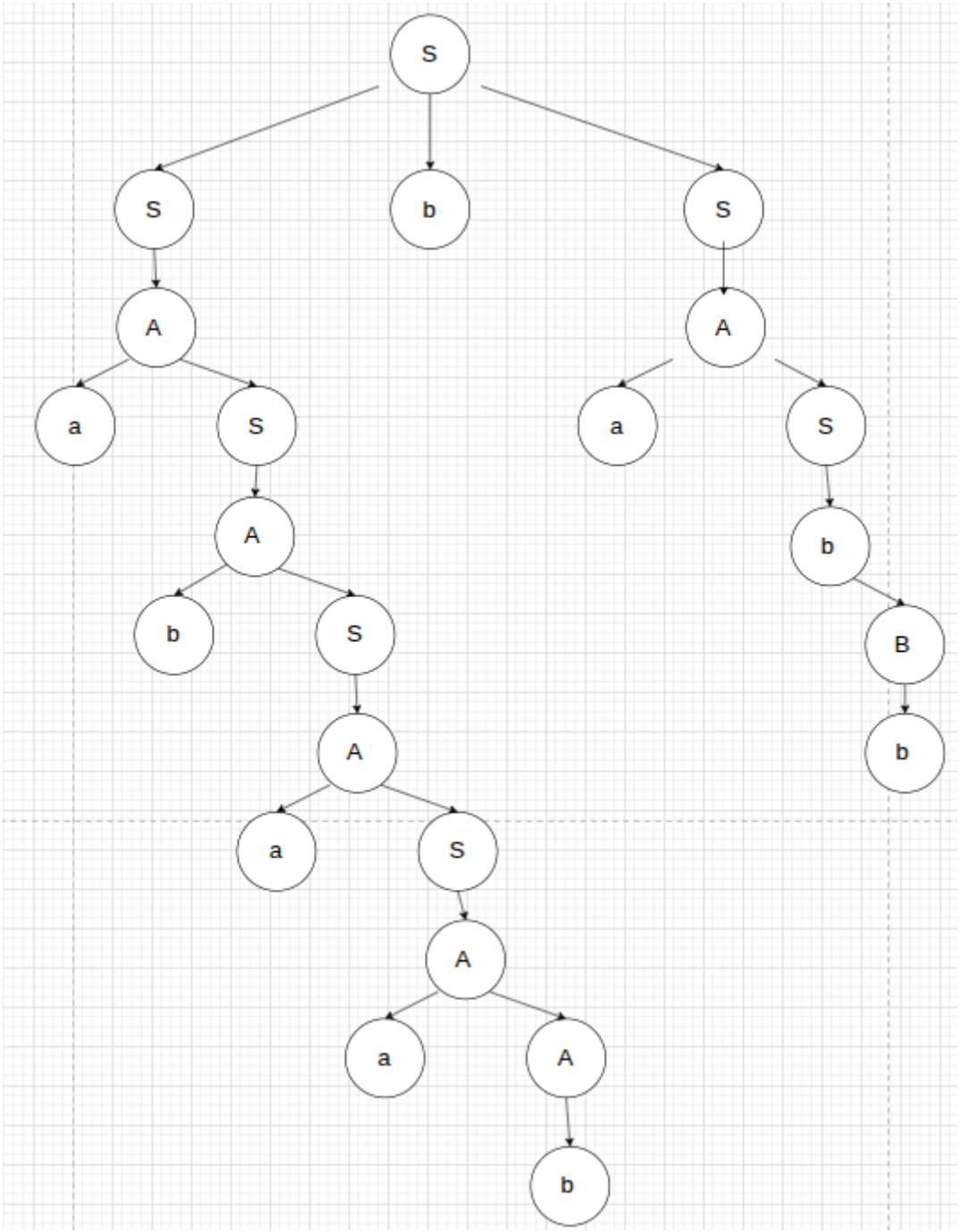
⇒ abaaaab**A**a (по правилу 3)

⇒ abaaaab**aB**a (по правилу 5)

⇒ abaaaaba**b**a (по правилу 7)

Последовательность правил для вывода 2:

1, 2, 2, 2, 2, 3, 4, 3, 6, 3, 5, 7



Задание 2

Код программы:

```
/**
* Программа выполняет следующие задачи:
* 1. Загружает предопределенную КС-грамматику (Вариант 9).
* 2. В интерактивном режиме запрашивает у пользователя последовательность номеров правил.
* 3. Выполняет симуляцию левого вывода, сохраняя состояние цепочки на каждом шаге.
* 4. Выводит пошаговый процесс левого вывода.
```

```
* 5. Формирует и выводит линейную скобочную форму дерева.
* 6. Строит и визуализирует псевдографическое дерево вывода. Для каждого узла,
*    где применялось правило, отображается полное состояние выводимой цепочки
*    сразу после применения этого правила.
*/

#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <stdexcept>
#include <cctype>

// --- Определение структур ---

/**
* @struct Rule
* @brief Хранит одно правило КС-грамматики.
*/
struct Rule {
   char lhs;           ///< Левая часть правила (нетерминал).
   std::string rhs;    ///< Правая часть правила (цепочка символов).
};

/**
* @struct TreeNode
* @brief Узел в дереве вывода.
*/
struct TreeNode {
   std::string value;                  ///< Значение узла (например, "S<1>" или терминал "a").
   std::string derivation_state_string;///< Полная рабочая цепочка после применения правила в этом узле.
   std::vector<TreeNode*> children;    ///< Дочерние узлы.

   /**
    * @brief Деструктор для рекурсивного освобождения памяти, выделенной под дерево.
    */
   ~TreeNode() {
       for (auto child : children) {
```

```
            delete child;
        }
    }
};


// --- Вспомогательные функции ---


/**
* @brief Проверяет, является ли символ нетерминалом (прописная буква).
* @param c Символ для проверки.
* @return true, если символ - нетерминал.
*/
bool isNonTerminal(char c) {
    return isupper(c);
}


/**
* @brief Находит индекс самого левого нетерминала в строке.
* @param s Строка для поиска.
* @return Индекс первого найденного нетерминала или -1, если не найден.
*/
int findLeftmostNonTerminalIndex(const std::string& s) {
    for (size_t i = 0; i < s.length(); ++i) {
        if (isNonTerminal(s[i])) {
            return i;
        }
    }

    return -1;
}


/**
* @brief Разбирает строку с правилами, разделенными запятыми, в вектор чисел.
* @param input Входная строка (например, "1, 2, 3").
* @param output Выходной вектор для сохранения чисел.
* @return true, если разбор успешен, иначе false.
*/
```

```cpp
bool parseRuleSequence(const std::string& input, std::vector<int>& output) {
    output.clear();
    std::stringstream ss(input);
    std::string segment;
    while (std::getline(ss, segment, ',')) {
        try {
            output.push_back(std::stoi(segment));
        } catch (const std::invalid_argument&) {
            std::cerr << "\n[ОШИБКА ВВОДА] Фрагмент '" << segment << "' не является корректным числом.\n";
            return false;
        } catch (const std::out_of_range&) {
            std::cerr << "\n[ОШИБКА ВВОДА] Число '" << segment << "' слишком большое.\n";
            return false;
        }
    }

    return true;
}



// --- Функции построения и отрисовки дерева ---

/**
* @brief Рекурсивно строит дерево вывода, используя предварительно вычисленные состояния вывода.
* @param nonTerminal Текущий нетерминал, для которого строится поддерево.
* @param grammar Ссылка на вектор правил грамматики.
* @param ruleSequence Последовательность применяемых правил.
* @param currentRuleIndex Ссылка на индекс текущего правила в последовательности.
* @param derivationStates Вектор, содержащий состояния всей цепочки на каждом шаге вывода.
* @return Указатель на корень построенного поддерева.
*/
TreeNode* buildTree(char nonTerminal, const std::vector<Rule>& grammar, const std::vector<int>& ruleSequence, size_t& currentRuleIndex, const std::vector<std::string>& derivationStates) {
    auto* node = new TreeNode();
    if (currentRuleIndex >= ruleSequence.size()) {
        node->value = "[ОШИБКА: Последовательность правил слишком коротка]";
        return node;
    }
```

```
    int ruleNum = ruleSequence[currentRuleIndex];


    // Проверяем корректность правила
    if (ruleNum <= 0 || (size_t)ruleNum >= grammar.size() || grammar[ruleNum].lhs != nonTerminal)
{
        node->value = "[ОШИБКА: Некорректное применение правила #" + std::to_string(ruleNum) +
"]";
        return node;
    }


    // Присваиваем значения узлу
    node->value = std::string(1, nonTerminal) + "<" + std::to_string(ruleNum) + ">";
    // Состояние цепочки ПОСЛЕ применения правила с индексом currentRuleIndex
    node->derivation_state_string = derivationStates[currentRuleIndex + 1];


    currentRuleIndex++;
    const Rule& rule = grammar[ruleNum];


    for (char symbol : rule.rhs) {
        if (isNonTerminal(symbol)) {
            node->children.push_back(buildTree(symbol, grammar, ruleSequence, currentRuleIndex,
derivationStates));
        } else {
            auto* leaf = new TreeNode();
            leaf->value = std::string(1, symbol);
            node->children.push_back(leaf);
        }
    }
    return node;
}


/**
* @brief Рекурсивно обходит дерево и генерирует его линейную скобочную форму.
* @param node Текущий узел дерева.
* @param ss Поток для записи результата.
*/
void generateBracketForm(const TreeNode* node, std::stringstream& ss) {
    if (!node) return;


    if (node->children.empty()) {
```

```
        ss << node->value;
    } else {
        ss << "[" << node->value;
        for (const auto* child : node->children) {
            ss << " ";
            generateBracketForm(child, ss);
        }
        ss << "]";
    }
}

/**
* @brief Вспомогательная рекурсивная функция для отрисовки псевдографического дерева.
* @param node Узел дерева для отрисовки.
* @param prefix Строка-префикс для форматирования.
* @param isLast Является ли этот узел последним в списке дочерних узлов.
*/
void printAsciiTreeRecursive(const TreeNode* node, const std::string& prefix, bool isLast) {
    if (!node) return;
    std::cout << prefix << (isLast ? "└─ " : "├─ ") << node->value;
    if (!node->derivation_state_string.empty()) {
        std::cout << " -> \"" << node->derivation_state_string << "\"";
    }
    std::cout << std::endl;

    std::string childPrefix = prefix + (isLast ? "    " : "│   ");
    for (size_t i = 0; i < node->children.size(); ++i) {
        printAsciiTreeRecursive(node->children[i], childPrefix, i == node->children.size() - 1);
    }
}

/**
* @brief Запускает отрисовку дерева вывода в псевдографическом виде.
* @param root Указатель на корневой узел дерева.
*/
void printAsciiTree(const TreeNode* root) {
    if (!root) return;
    std::cout << root->value;
```

```
    if (!root->derivation_state_string.empty()) {
        std::cout << " -> \"" << root->derivation_state_string << "\"";
    }
    std::cout << std::endl;
    for (size_t i = 0; i < root->children.size(); ++i) {
        printAsciiTreeRecursive(root->children[i], "", i == root->children.size() - 1);
    }
}



// --- Основная логика ---


/**
* @brief Выводит на экран правила заданной грамматики.
* @param grammar Вектор правил.
*/
void printGrammar(const std::vector<Rule>& grammar) {
    std::cout << "--- Грамматика (Вариант 9) ---\n";
    for (size_t i = 1; i < grammar.size(); ++i) {
        std::cout << i << ". " << grammar[i].lhs << " -> " << grammar[i].rhs << "\n";
    }
    std::cout << "--------------------------------\n\n";
}


/**
* @brief Обрабатывает одну последовательность правил: строит вывод и оба представления дерева.
* @param title Заголовок для блока вывода.
* @param grammar Ссылка на грамматику.
* @param ruleSequence Последовательность правил для обработки.
*/
void processDerivation(const std::string& title, const std::vector<Rule>& grammar, const
std::vector<int>& ruleSequence) {
    std::cout << "\n============================================================\n";
    std::cout << title << "\n";
    std::cout << "------------------------------------------------------------\n";


    // --- Этап 1: Симуляция левого вывода и сбор состояний ---
    std::vector<std::string> derivationStates;
    std::string currentString = "S";
```

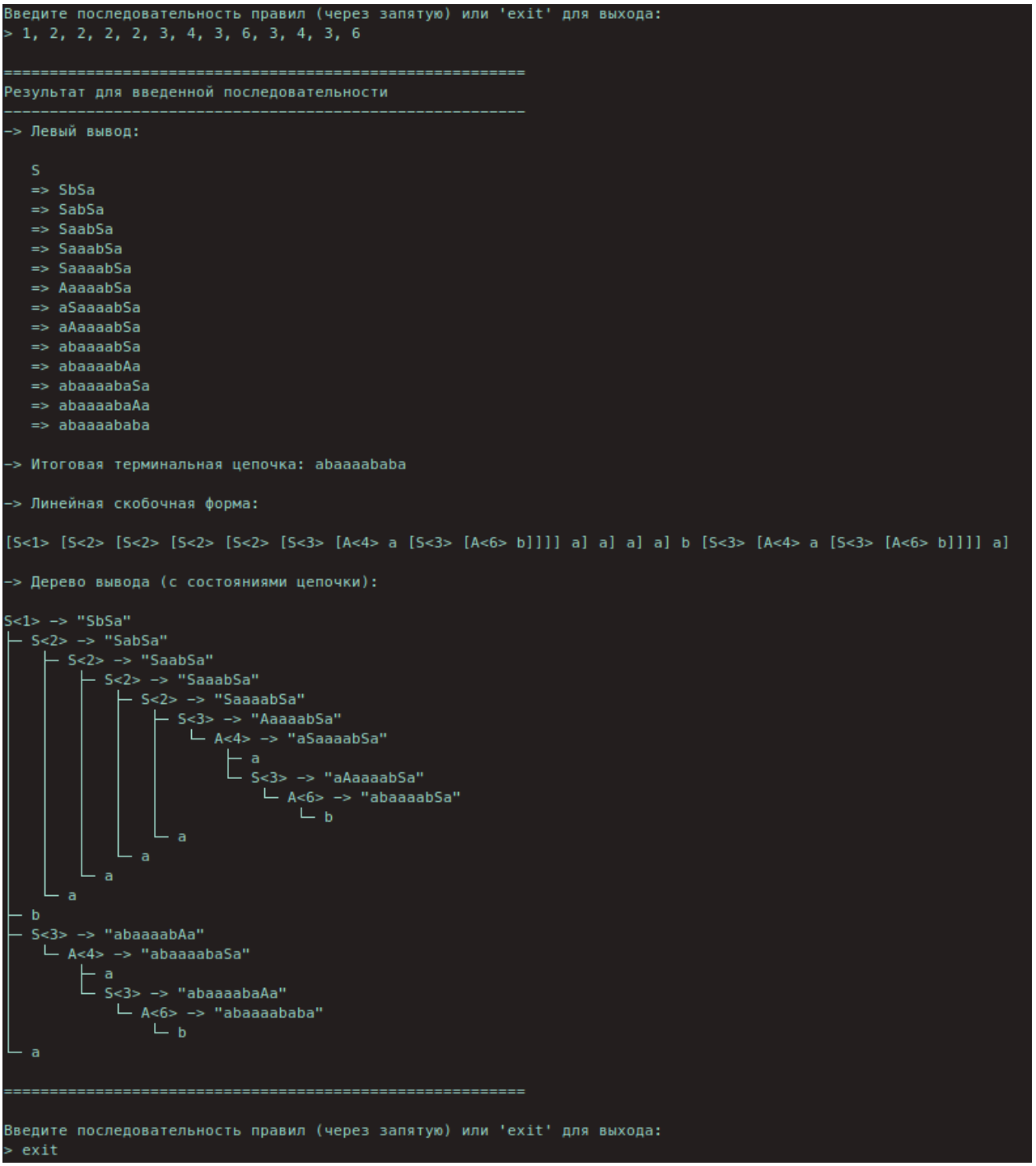
```
    derivationStates.push_back(currentString);


    std::cout << "-> Левый вывод:\n\n   " << currentString << "\n";

    for (int ruleNum : ruleSequence) {

        int index = findLeftmostNonTerminalIndex(currentString);

        if (index == -1) {

            std::cout << "\n[ОШИБКА ВЫВОДА] В цепочке нет нетерминалов, но правила еще есть.\n";

            return;

        }

        char leftmostNonTerminal = currentString[index];

        if (ruleNum <= 0 || (size_t)ruleNum >= grammar.size() || grammar[ruleNum].lhs !=
leftmostNonTerminal) {

             std::cout << "\n[ОШИБКА ВЫВОДА] Правило #" << ruleNum << " неприменимо к '" <<
leftmostNonTerminal << "'.\n";

             return;

        }


        const Rule& ruleToApply = grammar[ruleNum];

        currentString.replace(index, 1, ruleToApply.rhs);

        derivationStates.push_back(currentString); // Сохраняем состояние ПОСЛЕ применения

        std::cout << "   => " << currentString << "\n";

    }

    std::cout << "\n-> Итоговая терминальная цепочка: " << currentString << "\n";


    // --- Этап 2: Построение дерева и генерация представлений ---

    size_t ruleIndex = 0;

    TreeNode* root = buildTree('S', grammar, ruleSequence, ruleIndex, derivationStates);


    std::stringstream bracket_ss;

    generateBracketForm(root, bracket_ss);

    std::cout << "\n-> Линейная скобочная форма:\n\n" << bracket_ss.str() << "\n";


    std::cout << "\n-> Дерево вывода (с состояниями цепочки):\n\n";

    printAsciiTree(root);


    delete root; // Освобождаем память

    std::cout << "\n============================================================\n\n";

}
```

```
/**
* @brief Главная функция программы.
* @return 0 в случае успешного завершения.
*/
int main() {
   const std::vector<Rule> grammar = {
       {},
       {'S', "SbSa"}, {'S', "Sa"},   {'S', "A"},
       {'A', "aS"},   {'A', "aB"},   {'A', "b"},
       {'B', "b"},    {'B', "Aa"}
   };

   printGrammar(grammar);
   std::string userInput;
   while (true) {
       std::cout << "Введите последовательность правил (через запятую) или 'exit' для выхода:\n>
";
       std::getline(std::cin, userInput);

       if (userInput == "exit" || userInput == "quit") break;
       if (userInput.empty()) continue;

       std::vector<int> ruleSequence;
       if (parseRuleSequence(userInput, ruleSequence)) {
           if (ruleSequence.empty()) {
                std::cout << "[ПРЕДУПРЕЖДЕНИЕ] Вы ввели пустую последовательность.\n\n";
                continue;
           }
           processDerivation("Результат для введенной последовательности", grammar,
ruleSequence);
       } else {
           std::cout << "Пожалуйста, попробуйте еще раз.\n\n";
       }
   }

   std::cout << "\nПрограмма завершена.\n";

   return 0;
```

```
}
```

Пример работы программы:

```
Введите последовательность правил (через запятую) или 'exit' для выхода:
> 1, 2, 2, 2, 2, 3, 4, 3, 6, 3, 5, 7

============================================================
Результат для введенной последовательности
------------------------------------------------------------
-> Левый вывод:

   S
   => SbSa
   => SabSa
   => SaabSa
   => SaaabSa
   => SaaaabSa
   => AaaaabSa
   => aSaaaabSa
   => aAaaaabSa
   => abaaaabSa
   => abaaaabAa
   => abaaaabaBa
   => abaaaababa

-> Итоговая терминальная цепочка: abaaaababa

-> Линейная скобочная форма:

[S<1> [S<2> [S<2> [S<2> [S<2> [S<3> [A<4> a [S<3> [A<6> b]]]] a] a] a] a] b [S<3> [A<5> a [B<7> b]]] a]

-> Дерево вывода (с состояниями цепочки):

S<1> -> "SbSa"
├─ S<2> -> "SabSa"
│  ├─ S<2> -> "SaabSa"
│  │  ├─ S<2> -> "SaaabSa"
│  │  │  ├─ S<2> -> "SaaaabSa"
│  │  │  │  ├─ S<3> -> "AaaaabSa"
│  │  │  │  │  └─ A<4> -> "aSaaaabSa"
│  │  │  │  │     ├─ a
│  │  │  │  │     └─ S<3> -> "aAaaaabSa"
│  │  │  │  │        └─ A<6> -> "abaaaabSa"
│  │  │  │  │           └─ b
│  │  │  │  └─ a
│  │  │  └─ a
│  │  └─ a
│  └─ a
├─ b
├─ S<3> -> "abaaaabAa"
│  └─ A<5> -> "abaaaabaBa"
│     ├─ a
│     └─ B<7> -> "abaaaababa"
│        └─ b
│
```



Задание 3

Произвольный вывод v

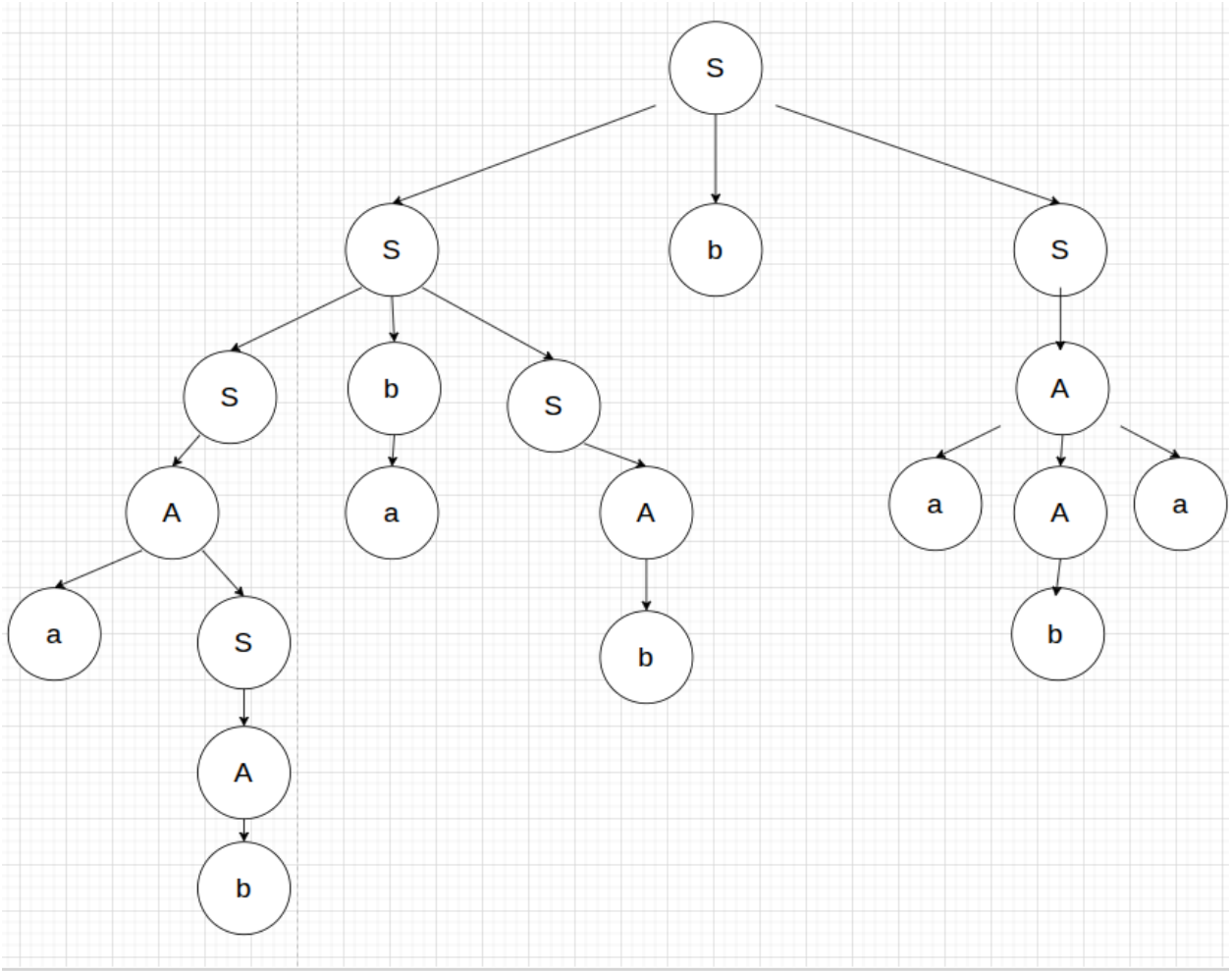
На шаге 3 мы заменяем средний нетерминал, нарушая правила левого и правого вывода

1. **S** ⇒ **SbSa** (по правилу **1**)
2. **S**bSa ⇒ **SbSa**bSa (по правилу **1**)

3. Sb**S**abSa ⇒ Sb**A**abSa (по правилу **3**)
4. Sb**A**abSa ⇒ Sb**b**abSa (по правилу **6**)
5. **S**bbabSa ⇒ **A**bbabSa (по правилу **3**)
6. **A**bbabSa ⇒ **aS**bbabSa (по правилу **4**)
7. a**S**bbabSa ⇒ a**A**bbabSa (по правилу **3**)
8. a**A**bbabSa ⇒ a**b**bbabSa (по правилу **6**)
9. abbab**S**a ⇒ abbab**Sa**a (по правилу **2**)
10. abbab**S**aa ⇒ abbab**Saa**a (по правилу **2**)
11. abbab**S**aaa ⇒ abbab**A**aaa (по правилу **3**)
12. abbab**A**aaa ⇒ abbab**b**aaa (по правилу **6**)

Итоговая терминальная цепочка abbababbaaa

Последовательность правил при правом выводе: (1, 1, 3, 6, 3, 4, 3, 6, 2, 2, 3, 6)



Эквивалентный левый вывод

1. **S** ⇒ **SbSa** (по правилу 1)
2. **S**bSa ⇒ **SbSa**bSa (по правилу 1)
3. **S**bSabSa ⇒ **A**bSabSa (по правилу 3)
4. **A**bSabSa ⇒ **aS**bSabSa (по правилу 4)
5. a**S**bSabSa ⇒ a**A**bSabSa (по правилу 3)
6. a**A**bSabSa ⇒ a**b**bSabSa (по правилу 6)
7. ab**S**abSa ⇒ ab**A**abSa (по правилу 3)
8. ab**A**abSa ⇒ ab**b**abSa (по правилу 6)
9. abbab**S**a ⇒ abbab**Sa**a (по правилу 2)
10. abbab**S**aa ⇒ abbab**Saa**a (по правилу 2)
11. abbab**S**aaa ⇒ abbab**A**aaa (по правилу 3)
12. abbab**A**aaa ⇒ abbab**b**aaa (по правилу 6)

Итоговая терминальная цепочка abbababbaaa

Последовательность правил при левом выводе: (1, 1, 3, 4, 3, 6, 3, 6, 2, 2, 3, 6).

Эквивалентный правый вывод

1. **S** ⇒ **SbSa** (по правилу 1)
2. Sb**S**a ⇒ Sb**Sa**a (по правилу 2)
3. Sb**S**aa ⇒ Sb**Saa**a (по правилу 2)
4. Sb**S**aaa ⇒ Sb**A**aaa (по правилу 3)
5. Sb**A**aaa ⇒ Sb**b**aaa (по правилу 6)
6. **S**bbaaa ⇒ **SbSa**bbaaa (по правилу 1)
7. SbS**a**bbaaa ⇒ Sb**A**ab_baaa (по правилу 3, примененное к S)
8. Sb**A**ab_baaa ⇒ Sb**b**ab_baaa (по правилу 6)
9. **S**bbab_baaa ⇒ **A**bbab_baaa (по правилу 3)
10. **A**bbab_baaa ⇒ **aS**bbab_baaa (по правилу 4)
11. a**S**bbab_baaa ⇒ a**A**bbab_baaa (по правилу 3)
12. a**A**bbab_baaa ⇒ a**b**bbab_baaa (по правилу 6)

Итоговая терминальная цепочка abbababbaaa

Последовательность правил при правом выводе: (1, 2, 2, 3, 6, 1, 3, 6, 3, 4, 3, 6)

Задание 4

Код программы:

```
/**
* Программа позволяет пользователю точно верифицировать конкретный произвольный вывод.
* 1. В моменты неоднозначности (когда правило можно применить к нескольким нетерминалам),
*    программа запрашивает выбор у пользователя, подсвечивая кандидатов.
* 2. Если вывод успешен, программа выводит:
*    a) Начальную и итоговую терминальную цепочки.
*    b) Линейную скобочную форму построенного дерева.
*    c) Последовательность правил для эквивалентного левого вывода.
*/


#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <stdexcept>
#include <cctype>
#include <algorithm>
#include <utility>


// --- Определение структур ---
struct Rule {
   char lhs;
   std::string rhs;
   Rule(char l, std::string r) : lhs(l), rhs(std::move(r)) {}
};


struct TreeNode {
   std::string value;
```

```
    std::vector<TreeNode*> children;
    bool is_terminal;
    TreeNode(std::string val, bool term) : value(std::move(val)), is_terminal(term) {}
    ~TreeNode() { for (auto child : children) { delete child; } }
};

// --- Вспомогательные функции и анализаторы ---
bool parseRuleSequence(const std::string& input, std::vector<int>& output);
int extractRuleNumber(const std::string& value);
void generateBracketForm(const TreeNode* node, std::stringstream& ss);
void findLeftmostSequence(const TreeNode* node, std::vector<int>& sequence);
void getTerminalString(const TreeNode* node, std::stringstream& ss);

// --- Основная логика ---

/** @brief Выводит на экран правила заданной грамматики. */
void printGrammar(const std::vector<Rule>& grammar) {
    std::cout << "--- Грамматика (Вариант 9) ---\n";

    for (size_t i = 1; i < grammar.size(); ++i) {
        std::cout << i << ". " << grammar[i].lhs << " -> " << grammar[i].rhs << "\n";
    }

    std::cout << "------------------------------\n\n";
}

/**
* @brief Главная функция, обрабатывающая произвольный вывод в интерактивном режиме.
*/
void processArbitraryDerivation(const std::vector<Rule>& grammar, const std::vector<int>&
ruleSequence) {
    std::cout << "\n============================================================\n";
    std::cout << "Анализ произвольного вывода\n";
    std::cout << "------------------------------------------------------------\n";

    auto* root = new TreeNode("S", false);
    std::vector<TreeNode*> worklist = {root}; // Отслеживает "живые" нетерминалы

    for (size_t i = 0; i < ruleSequence.size(); ++i) {
```

```
        int ruleNum = ruleSequence[i];


        if (ruleNum <= 0 || (size_t)ruleNum >= grammar.size()) {
            std::cerr << "[ОШИБКА] Неверный номер правила: " << ruleNum << std::endl;
            delete root; return;
        }


        const Rule& rule = grammar[ruleNum];


        std::vector<TreeNode*> candidate_nodes;
        for (TreeNode* node : worklist) {
            if (node->value[0] == rule.lhs) {
                candidate_nodes.push_back(node);
            }
        }


        if (candidate_nodes.empty()) {
            std::stringstream currentString;
            for(auto n : worklist) currentString << n->value;


            std::cerr << "[ОШИБКА] Правило #" << ruleNum << " (" << rule.lhs << " -> ...)
неприменимо. Доступные нетерминалы: \"" << currentString.str() << "\"\n";


            delete root;
            return;
        }


        TreeNode* node_to_expand = nullptr;
        if (candidate_nodes.size() == 1) {
            node_to_expand = candidate_nodes[0];
        } else {
            std::cout << "\nШаг " << i + 1 << ": Применяем правило #" << ruleNum << " (" <<
rule.lhs << " -> " << rule.rhs << ")\n";
            std::cout << "Текущая цепочка нетерминалов: ";
            std::stringstream highlighted_ss;
            int candidate_counter = 1;
            for (TreeNode* node : worklist) {
                bool is_candidate = false;
                for(auto c : candidate_nodes) { if(c == node) { is_candidate = true; break; } }
```

```
                if (is_candidate) {

                    highlighted_ss << "\033[1;31m" << node->value[0] << "_" << candidate_counter++
<< "\033[0m ";

                } else {

                    highlighted_ss << node->value << " ";

                }

            }

            std::cout << highlighted_ss.str() << "\n";

            std::cout << "Найдено несколько нетерминалов '" << rule.lhs << "'. Выберите, какой
заменить (укажите номер):\n";

            int choice = 0;

            while (true) {

                std::cout << "> ";

                std::cin >> choice;



                if (std::cin.good() && choice > 0 && (size_t)choice <= candidate_nodes.size()) {

                    std::cin.ignore(10000, '\n'); break;

                }



                std::cin.clear(); std::cin.ignore(10000, '\n');

                std::cerr << "Неверный ввод. Пожалуйста, введите число от 1 до " <<
candidate_nodes.size() << ".\n";

            }



            node_to_expand = candidate_nodes[choice - 1];

        }



        node_to_expand->value = std::string(1, rule.lhs) + "<" + std::to_string(ruleNum) + ">";



        for(char symbol : rule.rhs) {

            node_to_expand->children.push_back(new TreeNode(std::string(1, symbol),
!isupper(symbol)));

        }



        std::vector<TreeNode*> next_worklist;

        for (TreeNode* node : worklist) {

            if (node == node_to_expand) {

                for (TreeNode* child : node_to_expand->children) {

                    if (!child->is_terminal) { next_worklist.push_back(child); }
```

```
                }
            } else {
                next_worklist.push_back(node);
            }
        }
        worklist = next_worklist;
    }

    std::cout << "\n-----------------------------------------------------------\n";
    std::cout << "Произвольный вывод успешно завершен!\n";

    std::stringstream terminal_ss;
    getTerminalString(root, terminal_ss);

    std::cout << "\n-> Начальная цепочка (стартовый символ):\nS\n";
    std::cout << "\n-> Итоговая терминальная цепочка:\n" << terminal_ss.str() << "\n";

    std::stringstream bracket_ss;
    generateBracketForm(root, bracket_ss);

    std::cout << "\n-> Линейная скобочная форма дерева:\n" << bracket_ss.str() << "\n";

    std::vector<int> leftmost_sequence;
    findLeftmostSequence(root, leftmost_sequence);

    std::cout << "\n-> Эквивалентная последовательность правил для ЛЕВОГО вывода:\n";

    for(size_t i = 0; i < leftmost_sequence.size(); ++i) {
        std::cout << leftmost_sequence[i] << (i == leftmost_sequence.size() - 1 ? "" : ", ");
    }

    std::cout << "\n===========================================================\n\n";

    delete root;
}

int main() {
    const std::vector<Rule> grammar = {
```

```
        Rule{' ', ""}, Rule{'S', "SbSa"}, Rule{'S', "Sa"},   Rule{'S', "A"},

        Rule{'A', "aS"},   Rule{'A', "aB"},   Rule{'A', "b"}, Rule{'B', "b"},

        Rule{'B', "Aa"}

    };



    printGrammar(grammar);

    std::string userInput;



    while (true) {

        std::cout << "Введите последовательность правил `p` (через запятую) или 'exit' для
выхода:\n> ";

        std::getline(std::cin, userInput);



        if (userInput == "exit" || userInput == "quit") break;



        if (userInput.empty()) continue;



        std::vector<int> ruleSequence;



        if (parseRuleSequence(userInput, ruleSequence)) {

            processArbitraryDerivation(grammar, ruleSequence);

        }

    }



    std::cout << "\nПрограмма завершена.\n";



    return 0;

}



bool parseRuleSequence(const std::string& input, std::vector<int>& output) {

    output.clear();

    std::stringstream ss(input);

    std::string segment;



    while (std::getline(ss, segment, ',')) {

        try {

            output.push_back(std::stoi(segment));

        } catch (const std::exception&) {
```

```
            std::cerr << "\n[ОШИБКА ВВОДА] Некорректный фрагмент: '" << segment << "'.\n"; return
false;

        }

    }


    return true;

}


int extractRuleNumber(const std::string& value) {

    size_t start = value.find('<');

    size_t end = value.find('>');


    if (start == std::string::npos || end == std::string::npos)

        return -1;


    return std::stoi(value.substr(start + 1, end - start - 1));

}


void generateBracketForm(const TreeNode* node, std::stringstream& ss) {

    if (!node) return;

    if (node->is_terminal) { ss << node->value; }

    else {

        ss << "[" << node->value;

        for (const auto* child : node->children) { ss << " "; generateBracketForm(child, ss); }

        ss << "]";

    }

}


void findLeftmostSequence(const TreeNode* node, std::vector<int>& sequence) {

    if (!node || node->is_terminal) return;

    int ruleNum = extractRuleNumber(node->value);

    if (ruleNum != -1) { sequence.push_back(ruleNum); }

    for (const auto* child : node->children) { findLeftmostSequence(child, sequence); }

}


void getTerminalString(const TreeNode* node, std::stringstream& ss) {

    if (!node) return;

    if (node->is_terminal) { ss << node->value; return; }

    for (const auto* child : node->children) { getTerminalString(child, ss); }
```

```
}
```

Пример работы программы:

```
❯ g++ task4.cpp
❯ ./a.out
--- Грамматика (Вариант 9) ---
1. S -> SbSa
2. S -> Sa
3. S -> A
4. A -> aS
5. A -> aB
6. A -> b
7. B -> b
8. B -> Aa
------------------------------

Введите последовательность правил `p` (через запятую) или 'exit' для выхода:
> 1, 1, 3, 6, 3, 4, 3, 6, 2, 2, 3, 6

============================================================
Анализ произвольного вывода
------------------------------------------------------------

Шаг 2: Применяем правило #1 (S -> SbSa)
Текущая цепочка нетерминалов: S_1 S_2
Найдено несколько нетерминалов 'S'. Выберите, какой заменить (укажите номер):
> 1

Шаг 3: Применяем правило #3 (S -> A)
Текущая цепочка нетерминалов: S_1 S_2 S_3
Найдено несколько нетерминалов 'S'. Выберите, какой заменить (укажите номер):
> 2

Шаг 5: Применяем правило #3 (S -> A)
Текущая цепочка нетерминалов: S_1 S_2
Найдено несколько нетерминалов 'S'. Выберите, какой заменить (укажите номер):
> 1

Шаг 7: Применяем правило #3 (S -> A)
Текущая цепочка нетерминалов: S_1 S_2
Найдено несколько нетерминалов 'S'. Выберите, какой заменить (укажите номер):
> 1

------------------------------------------------------------
Произвольный вывод успешно завершен!

-> Начальная цепочка (стартовый символ):
S

-> Итоговая терминальная цепочка:
abbbabbaaa

-> Линейная скобочная форма дерева:
[S<1> [S<1> [S<3> [A<4> a [S<3> [A<6> b]]]] b [S<3> [A<6> b]] a] b [S<2> [S<2> [S<3> [A<6> b]] a] a] a]

-> Эквивалентная последовательность правил для ЛЕВОГО вывода:
1, 1, 3, 4, 3, 6, 3, 6, 2, 2, 3, 6
============================================================
```

Вывод: изучили основные понятия теории формальных языков и грамматик.