Федеральное государственное бюджетное образовательное учреждение высшего

образования

«Чувашский государственный университет им. И.Н. Ульянова»

Кафедра вычислительной техники

Объектно-ориентированное программирование

Курсовая работа

на тему “Модель лексического анализатора языка C”

Выполнил: Иванов В.С.

студент группы ИВТ-41-22

Проверил: кандидат технических наук

Обломов Игорь Александрович

Чебоксары, 2024

Оглавление

Оглавление………………………………………………………………….……2

Введение…………………………………………………………………….……3

1. Обзор предметной области…………………………………………….…….4

2. Возможные варианты и способы решения задачи…………[..….………](#__RefHeading___Toc450234331)...6

3. Блок-схема алгоритма……………………………………………….……...10

4.Структуры, писки данных и иерархия классов…………………………....14

5. Примеры результатов работы программы……………[.…………………](#__RefHeading___Toc450234334).....16

[Список использованных источников……………………………………..…](#__RefHeading___Toc450234338).18

[Приложение (листинг программы)…………………………………..……](#__RefHeading___Toc450234339)…..19

Введение

В современном программировании языки компиляции, такие как C++, занимают важное место благодаря своей производительности и близости к аппаратному обеспечению. Разработка надежных и эффективных компиляторов для таких языков имеет решающее значение для создания высокопроизводительных и стабильных программ. Лексический анализатор является одним из первых и ключевых этапов в процессе компиляции, от которого зависит успешное выполнение последующих этапов, таких как синтаксический анализ и генерация кода.

Целью курсовой работы является создание программы на языке C++ с использованием объектно-ориентированного программирования, реализующую модель лексического анализатора. Программа классифицирует различные лексемы и выводит их тип.

Данная курсовая работа способствует развитию навыков объектно-ориентированного программирования.

1.Обзор предметной области

Лексический анализатор является одним из ключевых компонентов компилятора, играющим критическую роль в процессе преобразования исходного кода программы в машинный код. Чтобы глубже понять значимость и механизм работы лексического анализатора, рассмотрим основные концепции и подходы в данной предметной области.

Основные концепции лексического анализа

1. **Лексемы и токены**:

* **Лексема**: последовательность символов, объединенных в единицу, которая распознается лексическим анализатором как значимая. Примеры лексем включают ключевые слова, идентификаторы, операторы, литералы и разделители.
* **Токен**: абстрактное представление лексемы, которое включает тип и возможное значение. Токены передаются на следующую стадию компиляции — синтаксический анализ. В коде токены представлены объектами классов, такие как Identifier, Keyword, Operator, Literal, и Delimiter.

1. **Цели лексического анализа**:

* Преобразование последовательности символов исходного кода в последовательность токенов.
* Удаление пробельных символов и комментариев, которые не влияют на смысл программы.
* Обработка ошибок на уровне лексем, таких как недопустимые символы или слишком длинные идентификаторы.

Лексические анализаторы находят применение в различных областях:

* **Компиляторы**: основной компонент, обеспечивающий корректное разбиение исходного кода на токены.
* **Интерпретаторы**: используется для разбора и исполнения кода на лету.
* **Инструменты статического анализа кода**: проверяют исходный код на наличие ошибок, уязвимостей и несоответствий стилю кодирования.
* **Средства разработки**: обеспечивают функции подсветки синтаксиса, автодополнения и рефакторинга кода.

Лексический анализатор — это фундаментальный компонент компилятора, обеспечивающий преобразование исходного кода в токены.

2.Возможные варианты и способы решения задачи

Разработка лексического анализатора для языка программирования C включает несколько ключевых этапов, которые могут быть реализованы различными способами. Рассмотрим основные варианты и способы решения этой задачи:

1. Использование готовых инструментов для генерации лексических анализаторов

1. Flex (Fast Lexical Analyzer Generator):

* **Описание**: Flex — это инструмент для создания лексических анализаторов. Он принимает на вход спецификацию, написанную на языке регулярных выражений, и генерирует исходный код на C/C++.
* **Преимущества**:
  + Быстрая разработка благодаря использованию шаблонов.
  + Хорошо подходит для сложных проектов с многочисленными лексемами.
* **Недостатки**:
  + Требуется изучение синтаксиса Flex и его интеграции с остальным кодом.
* **Пример использования**: Написание спецификации для Flex, которая описывает правила распознавания идентификаторов, ключевых слов, операторов и других лексем, и генерация соответствующего кода.

1. ANTLR (Another Tool for Language Recognition):

* **Описание**: ANTLR — это инструмент для создания лексических и синтаксических анализаторов. Он использует язык описания грамматик, из которых генерирует код на различных языках программирования, включая C++.
* **Преимущества**:
  + Возможности для обработки сложных грамматик.
  + Поддержка различных целевых языков программирования.
* **Недостатки**:
  + Более сложный синтаксис по сравнению с Flex.
  + Может требовать дополнительной настройки для интеграции
* **Пример использования**: Определение грамматики для языка C++, включающей правила для лексем и синтаксиса, и генерация кода лексического анализатора с помощью ANTLR.

2. Ручная реализация лексического анализатора

1. Использование регулярных выражений:

* **Описание**: Регулярные выражения могут быть использованы для определения шаблонов, по которым распознаются лексемы.
* **Преимущества**:
  + Гибкость и контроль над процессом анализа.
  + Легкость в использовании для простых грамматик.
* **Недостатки**:
* Может стать сложным и трудоемким для больших и сложных языков.
* Возможны проблемы с производительностью при обработке больших объемов кода.
* **Пример использования**: Написание кода на C++ с использованием стандартной библиотеки регулярных выражений для поиска и классификации лексем.

1. Использование конечных автоматов:

* **Описание**: Конечные автоматы (детерминированные или недетерминированные) могут быть использованы для построения лексического анализатора.
* **Преимущества**:
  + Высокая производительность.
  + Хорошая основа для оптимизации и расширения.
* **Недостатки**:
  + Требует знаний теории автоматов и ручного создания автоматов для каждой лексемы.
* **Пример использования**: Реализация конечного автомата на C++ для распознавания лексем, где каждое состояние автомата представляет различную часть процесса распознавания.

3. Объектно-ориентированное программирование

1. Создание иерархии классов для лексем:

* **Описание**: Определение базового класса Lexeme и производных классов для различных типов лексем, таких как Identifier, Keyword, Operator, Literal и Delimiter.
* **Преимущества**:
  + Легкость в расширении и модификации.
  + Ясная структура и поддерживаемость кода.
* **Недостатки**:
  + Может потребовать больше времени на начальную настройку и проектирование.
* **Пример использования**: Создание классов лексем и их атрибутов, реализация метода для анализа и классификации строк входного кода.

Выбор способа реализации лексического анализатора зависит от требований проекта, доступных ресурсов и знаний разработчика. Готовые инструменты, такие как Flex и ANTLR, могут значительно ускорить процесс разработки, особенно для сложных языков. Ручная реализация на основе регулярных выражений или конечных автоматов предоставляет больше контроля и гибкости, но требует большего объема работы и знаний. Объектно-ориентированный подход помогает создать легко расширяемую и поддерживаемую систему.

В данной курсовой работе выбран объектно-ориентированный подход, который демонстрирует основные принципы лексического анализа и позволяет легко расширять функциональность анализатора в будущем.

3.Блок-схема алгоритма

На рисунке 1 приведена графическая блок-схема алгоритма анализатора лексем.

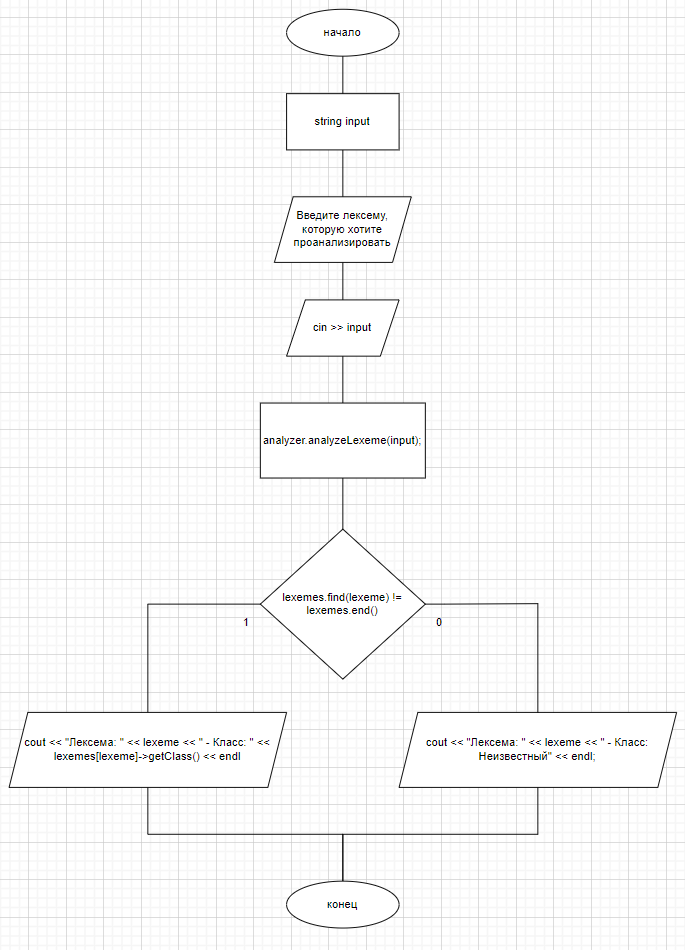


Рис. 1. Блок-схема алгоритма анализатора лексем

Блок-схема класса LexicalAnalyzer изображена на рисунах 2, 3 и 4.

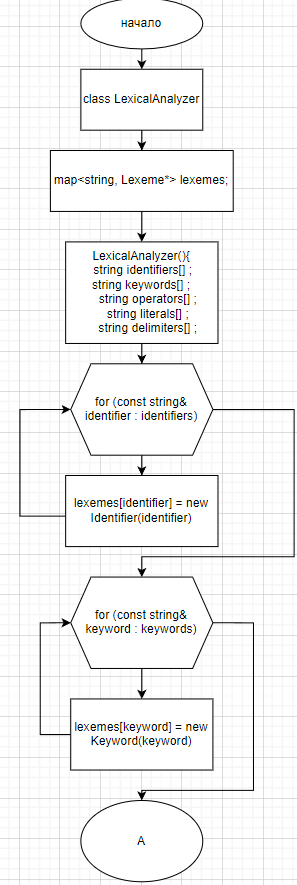


Рис. 2. БСА класса LexicalAnalyzer

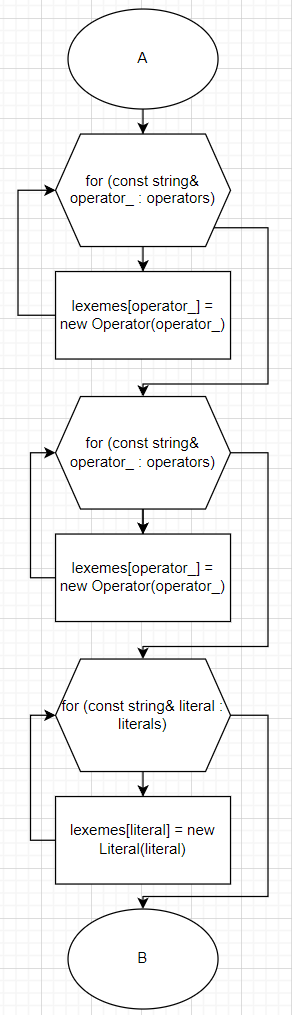


Рис. 3. БСА класса LexicalAnalyzer

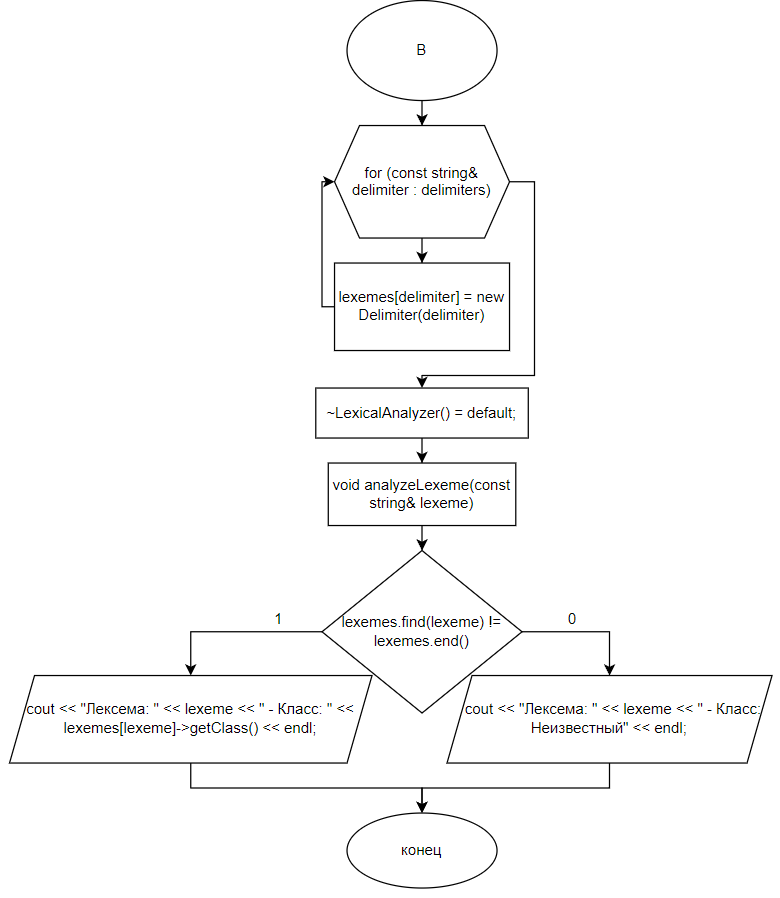


Рис. 4. БСА класса LexicalAnalyzer

4.Структуры, списки данных и иерархия классов

В данной программе используется несколько классов для представления и анализа лексем. Основным классом является LexicalAnalyzer, который использует map для хранения лексем разных типов. Программа разделяет лексемы на идентификаторы, ключевые слова, операторы, литералы и разделители. Конструктор класса LexicalAnalyzer инициализирует этот словарь.

Базовым классом является Lexeme, который содержит поле string name для хранения имени лексемы. Конструктор класса принимает это имя, а метод virtual string getClass() const = 0 является чисто виртуальным и должен быть переопределен в производных классах для возвращения класса лексемы.

Производные классы включают Identifier, Keyword, Operator, Literal и Delimiter. Каждый из этих классов наследует от Lexeme и переопределяет метод getClass(), чтобы возвращать соответствующий класс лексемы. Конструкторы этих классов принимают имя лексемы и передают его базовому классу Lexeme.

Класс LexicalAnalyzer содержит поле map<string, Lexeme\*> lexemes, которое представляет собой ассоциативный массив для хранения лексем. Конструктор класса заполняет этот массив, создавая объекты соответствующих классов для каждой строки из предварительно определенных массивов строк, таких как identifiers, keywords, operators, literals и delimiters.

Конструктор класса LexicalAnalyzer инициализирует массивы строк для различных категорий лексем и заполняет контейнер lexemes, перебирая каждый массив строк и создавая объекты соответствующих классов. Эти объекты затем добавляются в map<string, Lexeme\*> lexemes с использованием строки как ключа.

Метод void analyzeLexeme(const string& lexeme) класса LexicalAnalyzer принимает лексему и анализирует ее, выводя на экран класс лексемы, если она найдена в lexemes, или сообщение о неизвестной лексеме, если не найдена.

В функции main программа запрашивает у пользователя лексему для анализа, создает объект класса LexicalAnalyzer, и использует его метод analyzeLexeme для определения класса лексемы. Также анализируются несколько заранее заданных лексем, таких как "int", "if", "/", "123", "(" и "xyz", чтобы продемонстрировать работу программы.

Для работы программы используются следующие библиотеки: <iostream> для ввода и вывода, <string> для работы со строками, и <map> для ассоциативного массива. Программа предоставляет гибкий и расширяемый способ для анализа лексем путем добавления новых категорий лексем и методов анализа.

5.Примеры результатов работы программы

Каждый массив типов лексем содержат по 10-15 ключевых слов:  
 string identifiers[] = { "int", "float", "char", "string", "bool", "double",

"long", "short", "unsigned", "signed", "void", "struct", "class", "enum",

"typedef", "namespace", "const", "static", "extern", "auto" };

string keywords[] = { "if", "else", "while", "for", "return",

"switch", "case", "break", "continue", "do", "class", "public",

"private", "protected", "virtual" };

string operators[] = { "+", "-", "\*", "/", "=", "==", "!=", "<", ">",

"<=", ">=", "&&", "||", "++", "--", "+=", "-=", "\*=", "/="

string literals[] = { "123", "3.14", "'a'", "\"Hello\"", "true", "false",

"0xFF", "0b1010", "1.23e4", "0" };

string delimiters[] = { ";", ",", "(", ")", "{", "}", "[", "]", ":",

".", "->", "::", "#", "//", "/\*", "\*/" };

Примеры работы программы для лексемы типа идентификатор показан на рисунке 5:

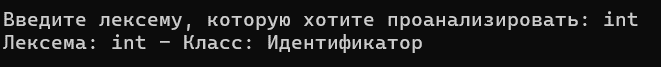


Рис. 5. Пример для лексемы типа идентификатор

Примеры работы программы для лексемы типа ключевое слово показан на рисунке 6:



Рис. 6. Пример для лексемы типа ключевое слово

Примеры работы программы для лексемы типа оператор показан на рисунке 7:



Рис. 7. Пример для лексемы типа оператор

Подобным образом программа работает и для остальных случаев.

Однако, если введенная лексема не найдена в map<string, Lexeme\*> lexemes, то в консоль выводится, что она принадлежит неизвестному классу. Пример ввода неизвестной лексемы можно увидеть на рисунке 8.



Рис. 8. Пример ввода лексемы, отсутствующей в массиве лексем

Список использованной литературы

1. Ф. Льюис, Д. Розенкранц, Р. Стирнз Теоретические основы проектирования компиляторов / Москва: издательство «МИР», 1979. – 645 с.

ПРИЛОЖЕНИЕ

#include <iostream>

#include <string>

#include <map>

using namespace std;

// Базовый класс для всех лексем

class Lexeme {

string name;

public:

Lexeme(const string& name) : name(name) {}

virtual string getClass() const = 0;

};

class Identifier : public Lexeme {

public:

Identifier(const string& name) : Lexeme(name) {}

string getClass() const override {

return "Идентификатор";

}

};

class Keyword : public Lexeme {

public:

Keyword(const string& name) : Lexeme(name) {}

string getClass() const {

return "Ключевое слово";

}

};

class Operator : public Lexeme {

public:

Operator(const string& name) : Lexeme(name) {}

string getClass() const {

return "Знак операции";

}

};

class Literal : public Lexeme {

public:

Literal(const string& name) : Lexeme(name) {}

string getClass() const {

return "Литерал";

}

};

class Delimiter : public Lexeme {

public:

Delimiter(const string& name) : Lexeme(name) {}

string getClass() const {

return "Разделитель";

}

};

class LexicalAnalyzer {

private:

map<string, Lexeme\*> lexemes;

public:

LexicalAnalyzer() {

string identifiers[] = { "int", "float", "char", "string", "bool", "double",

"long", "short", "unsigned", "signed", "void", "struct", "class", "enum",

"typedef", "namespace", "const", "static", "extern", "auto" };//Идентификаторы

string keywords[] = { "if", "else", "while", "for", "return",

"switch", "case", "break", "continue", "do", "class", "public",

"private", "protected", "virtual" };//Ключевые слова

string operators[] = { "+", "-", "\*", "/", "=", "==", "!=", "<", ">",

"<=", ">=", "&&", "||", "++", "--", "+=", "-=", "\*=", "/=" };//Операторы

string literals[] = { "123", "3.14", "'a'", "\"Hello\"", "true", "false",

"0xFF", "0b1010", "1.23e4", "0" };//Литералы(константные значения)

string delimiters[] = { ";", ",", "(", ")", "{", "}", "[", "]", ":",

".", "->", "::", "#", "//", "/\*", "\*/" };//Разделители

for (const string& identifier : identifiers) {

lexemes[identifier] = new Identifier(identifier);

}

for (const string& keyword : keywords) {

lexemes[keyword] = new Keyword(keyword);

}

for (const string& operator\_ : operators) {

lexemes[operator\_] = new Operator(operator\_);

}

for (const string& literal : literals) {

lexemes[literal] = new Literal(literal);

}

for (const string& delimiter : delimiters) {

lexemes[delimiter] = new Delimiter(delimiter);

}

}

~LexicalAnalyzer() = default;

void analyzeLexeme(const string& lexeme) {

if (lexemes.find(lexeme) != lexemes.end()) {

cout << "Лексема: " << lexeme << " - Класс: " << lexemes[lexeme]->getClass() << endl;

}

else {

cout << "Лексема: " << lexeme << " - Класс: Неизвестный" << endl;

}

}

/\*void Out() {

for (const auto& lexeme : lexemes) {

cout << "Лексема: " << lexeme.first << " - Класс: " << lexeme.second->getClass() << endl;

}

}\*/

};

int main() {

setlocale(LC\_ALL, "Russian");

string input;

LexicalAnalyzer analyzer;

//analyzer.Out();

cout << "Введите лексему, которую хотите проанализировать: ";

cin >> input;

analyzer.analyzeLexeme(input);

/\*analyzer.analyzeLexeme("int");

analyzer.analyzeLexeme("if");

analyzer.analyzeLexeme("/");

analyzer.analyzeLexeme("123");

analyzer.analyzeLexeme("(");

analyzer.analyzeLexeme("xyz"); // Неизвестная лексема\*/

return 0;

}