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. Способ решения задачи…………………………………….…........6

3. Блок-схема алгоритма……………………………………….….....9

4.Структуры, списки данных и иерархия классов…………………11

5. Примеры результатов работы программы…[.………………](#__RefHeading___Toc450234334)……...12

6. Заключение…………………………………………………………..13

[Список использованных источников…………………………...…](#__RefHeading___Toc450234338)..14

[Приложение (листинг программы)…………………………..…](#__RefHeading___Toc450234339)……15

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. Рассмотрим основные концепции и подходы в данной предметной области.

**Лексемы и токены**:

* **Лексема** - это последовательность символов в исходном тексте, которая распознается как единое целое и соответствует определенной категории. Лексемы являются основными строительными блоками для создания токенов. Примеры лексем включают ключевые слова (например, if, else), операторы (например, +, -), идентификаторы (например, variableName), числа (например, 123), и т.д.
* **Токен** - это пара, состоящая из типа токена и соответствующей лексемы. Тип токена указывает на категорию лексемы (например, ключевое слово, оператор, число), а сама лексема – это конкретная строка символов из исходного текста. Например, токен (KEYWORD, "if") означает, что лексема "if" принадлежит к категории ключевых слов.

**Цели лексического анализа**:

* Преобразование последовательности символов исходного кода в последовательность токенов.
* Удаление пробельных символов
* Обработка ошибок на уровне лексем, таких как недопустимые символы.

**Детерминированный конечный автомат (ДКА)**

Детерминированный конечный автомат (ДКА) - это математическая модель, которая описывает регулярные языки, где для каждого состояния и символа входного алфавита существует только один переход к следующему состоянию.

Вот переписанный список компонентов для ДКА:

* + Множество состояний.
  + Алфавит входных символов.
  + Множество переходов между состояниями.
  + Начальное состояние.
  + Множество конечных состояний.
  + В отличие от НКА, в ДКА для каждой пары состояние-символ существует только один следующий переход, что делает его более предсказуемым и легче понимаемым..

**Конечный автомат Мили и Мюллера**

Конечный автомат Мили (Mealy machine) – это конечный автомат, где выходные сигналы зависят от текущего состояния и входного символа.

Конечный автомат Мюллера (Moore machine) – это конечный автомат, где выходные сигналы зависят только от текущего состояния.

Примеры диаграммы переходов конечных автоматов Мура и Мили можно увидеть на рисунке 1.

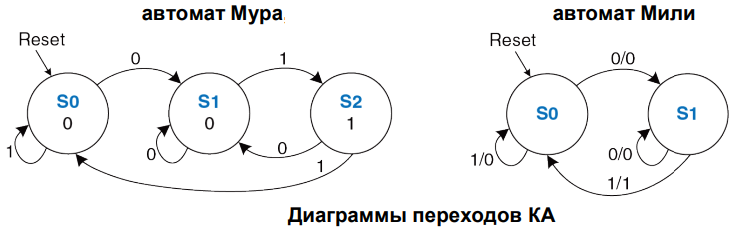


Рис. 1. Диаграммы переходов конечных автоматов Мура и Мили

2. Способ решения задачи

Для решения поставленной задачи был написан код на языке C++ с использованием детерминированного конченого автомата. Он имеет 7 состояний:

* S0 - начальное состояние.
* I - идентификаторы и ключевые слова.
* L - литералы.
* O - операторы.
* S - разделители.
* C – комментарии.
* F - конечное состояние.
* E - ошибочное состояние.

Таблица переходов показана в таблице 1.

Таблица 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Соcтояние | Буква или '\_' | Цифра | Комментарий | Оператор | Разделитель | Недопустимый символ |
| S0 | I | L | C | O | S | E |
| I | I | I | F | F | F | E |
| L | F | L | F | F | F | E |
| O | F | F | F | F | F | E |
| S | F | F | F | F | F | E |
| C | F | F | S0 | F | F | F |
| F | F | F | F | F | F | E |
| E | E | E | F | E | E | E |

Таблица переходов

**Пояснение к таблице:**

* **S0**: начальное состояние. Здесь начинается анализ строки.
  + Если встречается буква или символ '\_', происходит переход в состояние **I**.
  + Если встречается цифра, происходит переход в состояние **L**.
  + Если встречается символ ‘/’, происходит переход в состояние **C.**
  + Если встречается оператор, происходит переход в состояние **O**.
  + Если встречается разделитель, происходит переход в состояние **S**.
  + Если встречается недопустимый символ или прочие символы, происходит переход в состояние **E**.
* **I**: состояние идентификаторов и ключевых слов.
  + Если встречается буква, цифра или '\_', состояние остается **I**.
  + В остальных случаях происходит переход в состояние **F** (конец идентификатора).
* **L**: состояние числовых литералов.
  + Если встречается цифра, состояние остается **L**.
  + В остальных случаях происходит переход в состояние **F** (конец литерала).
* **C**: состояние комментариев.
  + Если встречается символ или ‘/’, то происходит переход в **S0.**
* **O**: состояние операторов.
  + Если встречается символ или ‘\_’, то происходит переход в **E**, иначе в **F**.
* **S**: состояние разделителей.
  + Независимо от входного символа происходит переход в состояние **F** (конец разделителя).
* **F**: конечное состояние. Сюда попадают после завершения анализа лексемы.
  + Независимо от входного символа остается в состоянии **F**.
* **E**: ошибочное состояние. Сюда попадают при встрече недопустимых символов.
  + Независимо от входного символа остается в состоянии **E**.

Вид диаграммы переходов конечного автомата можно увидеть на рис. 2.

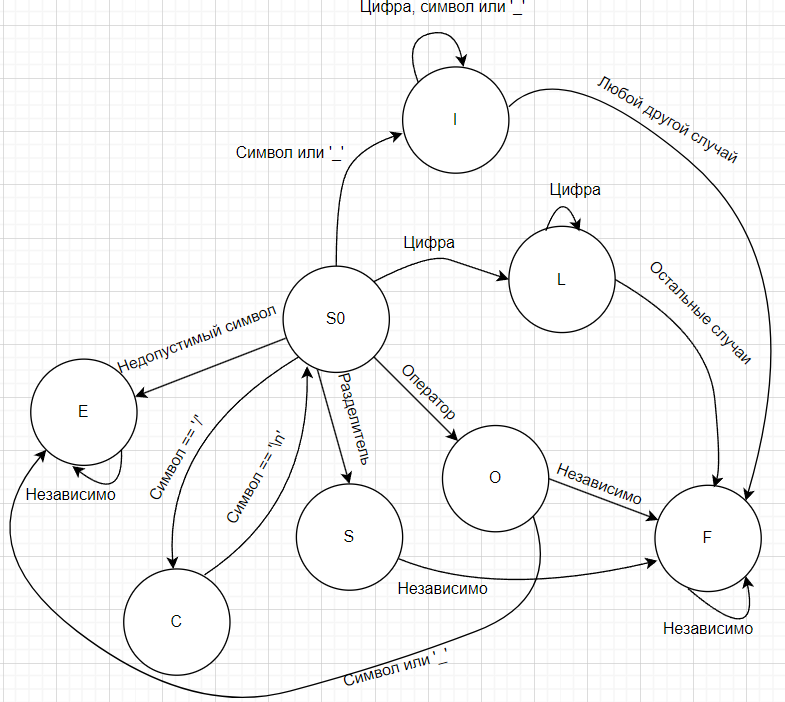


Рис. 2. Диаграмма переходов конечного автомата

3.Блок-схема алгоритма

На рисунке 3 и 4 приведена графическая блок-схема алгоритма анализатора лексем.

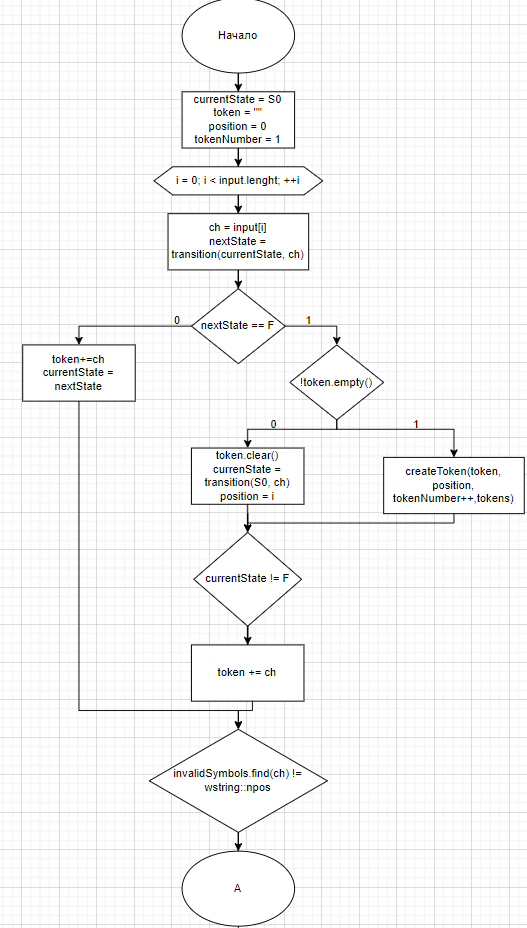


Рис. 3. Блок-схема алгоритма анализатора лексем

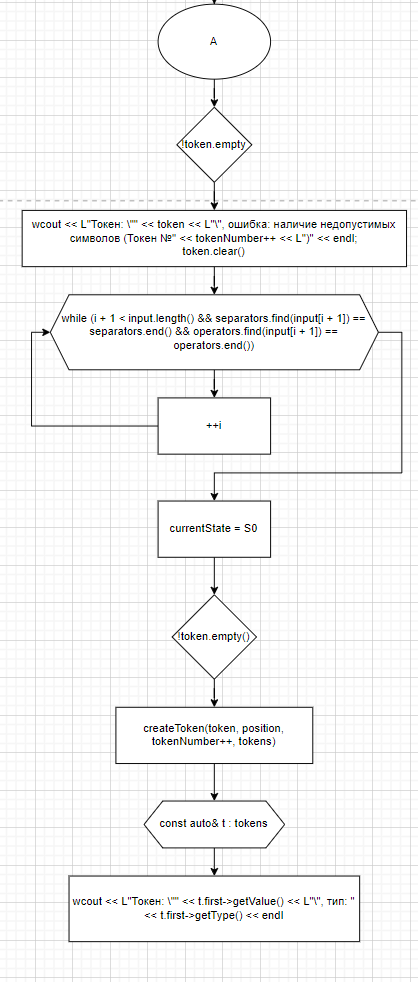


Рис. 4. Блок-схема алгоритма анализатора лексем

4.Структуры, списки данных и иерархия классов

В данной программе используется несколько классов для представления и анализа лексем. Основным классом является Token, который содержит абстрактные методы.

Производные классы включают IdentifierToken, OperatorToken, LiteralToken и SeparatorToken. Каждый из этих классов наследует от Lexeme и переопределяет методы getType() и getValue().

Также есть перечисление State, которое определяет состояния конечного автомата, наборы ключевых слов, операторов, разделителей и недопустимых символов.

Для работы программы используются следующие библиотеки: <iostream> для ввода и вывода, <string> для работы со строками, и <map> для ассоциативного массива, <cctype> для классификации символов, <vector> для динамических массивов и <fstream> для работы с файлами.

5.Примеры результатов работы программы

Для строки input = L"int main(){ int x = 42; if (x > 0) return x; }"

Программа выведет данный список лексем (рис.5):

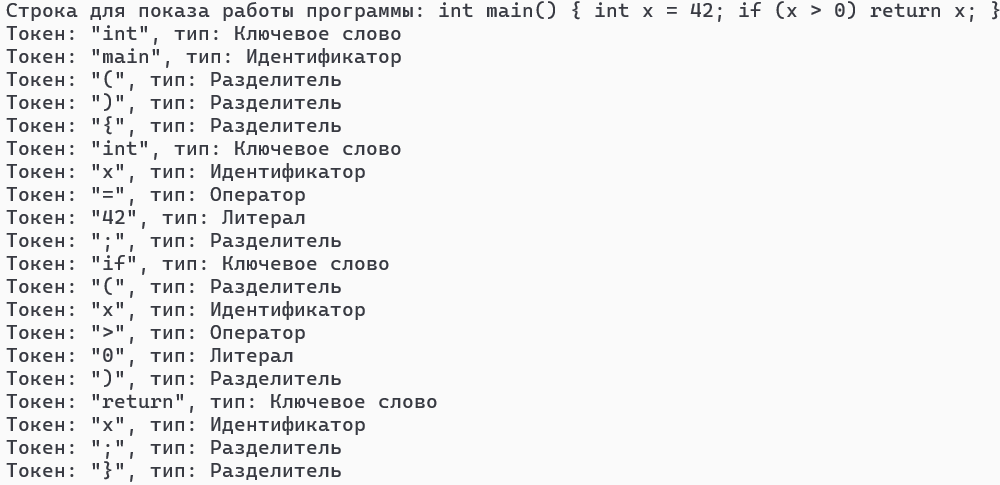


Рис. 5. Пример обработки строки без ошибок

Однако, если мы умышленно введем ошибки в некоторые лексемы, и получим строку input = L"int maiававn() Pnt\_22 22pytr { int x = 42; if (x > 0) return x; } // Это комментарий

if (x > 0) return x; // Еще один комментарий

}", то программа выведет сперва ошибочные токены, а после нормальные. Вывод данного случая показан на рисунке 6.

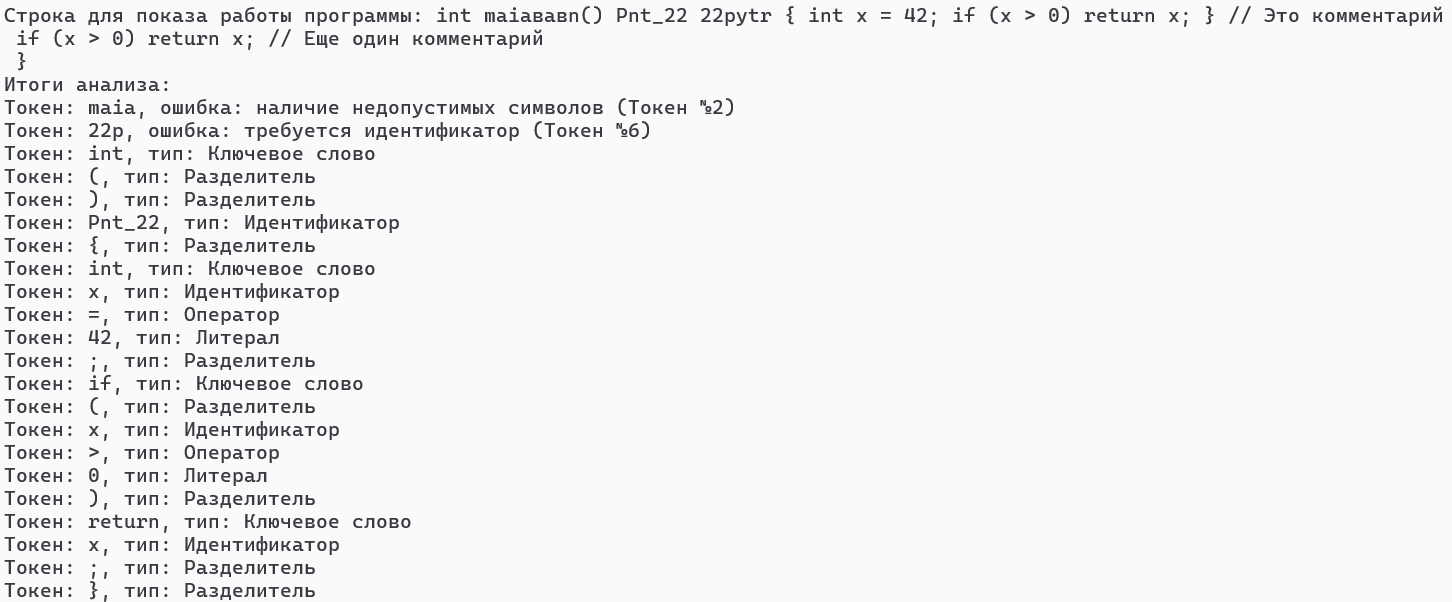


Рис. 6. Пример обработки строки с ошибками

Заключение

В ходе данной курсовой работы была реализована модель лексического анализатора на основе конечного автомата.

В программе использовались принципы ООП, а также контейнеры из библиотеки STL, а именно последовательный контейнер vector и упорядоченный ассоциативный массив пар ключ-значение map.

Итогом курсовой работы можно назвать закрепление знаний в области навыков программирования на С++.

Список использованной литературы

1. Ф. Льюис, Д. Розенкранц, Р. Стирнз Теоретические основы проектирования компиляторов / Москва: издательство «МИР», 1979. – 645 с.
2. С чем едят конечный автомат // Хабр URL: https://habr.com/ru/companies/timeweb/articles/717628/ (дата обращения: 20.05.2024).
3. Конечный автомат: теория и реализация // Tproger URL: https://tproger.ru/translations/finite-state-machines-theory-and-implementation (дата обращения: 21.05.2024).

ПРИЛОЖЕНИЕ

#include <iostream>

#include <string>

#include <map>

#include <cctype>

#include <vector>

using namespace std;

enum State {

S0, // Начальное состояние

I, // Идентификаторы и ключевые слова

L, // Литералы

O, // Операторы

S, // Разделители

C, // Комментарий

F, // Конечное состояние

E // Ошибочное состояние

};

const map<wstring, wstring> keywords = {

{L"int", L"KEYWORD"}, {L"float", L"KEYWORD"}, {L"if", L"KEYWORD"},

{L"else", L"KEYWORD"}, {L"while", L"KEYWORD"}, {L"return", L"KEYWORD"},

{L"char", L"KEYWORD"}, {L"double", L"KEYWORD"}, {L"for", L"KEYWORD"}, {L"void", L"KEYWORD"}

};

const map<wchar\_t, wstring> operators = {

{L'+', L"OPERATOR"}, {L'-', L"OPERATOR"}, {L'\*', L"OPERATOR"},

{L'/', L"OPERATOR"}, {L'=', L"OPERATOR"}, {L'<', L"OPERATOR"},

{L'>', L"OPERATOR"}, {L'!', L"OPERATOR"}

};

const map<wchar\_t, wstring> separators = {

{L'\n', L"SEPARATOR"}, {L'\t', L"SEPARATOR"},

{L'(', L"SEPARATOR"}, {L')', L"SEPARATOR"}, {L'{', L"SEPARATOR"},

{L'}', L"SEPARATOR"}, {L';', L"SEPARATOR"}, {L',', L"SEPARATOR"}

};

const wstring invalidSymbols = L"@#$абвгдеёжзийклмнопрстуфхцчшщъыьэюяАБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";

class Token {

public:

virtual wstring getType() const = 0;

virtual wstring getValue() const = 0;

virtual ~Token() = default;

};

class IdentifierToken : public Token {

private:

wstring value;

public:

IdentifierToken(const wstring& val) : value(val) {}

wstring getType() const override {

if (keywords.count(value)) {

return L"Ключевое слово";

}

else {

return L"Идентификатор";

}

}

wstring getValue() const override {

return value;

}

};

class LiteralToken : public Token {

private:

wstring value;

public:

LiteralToken(const wstring& val) : value(val) {}

wstring getType() const override {

return L"Литерал";

}

wstring getValue() const override {

return value;

}

};

class OperatorToken : public Token {

private:

wchar\_t value;

public:

OperatorToken(wchar\_t val) : value(val) {}

wstring getType() const override {

return L"Оператор";

}

wstring getValue() const override {

return wstring(1, value);

}

};

class SeparatorToken : public Token {

private:

wchar\_t value;

public:

SeparatorToken(wchar\_t val) : value(val) {}

wstring getType() const override {

return L"Разделитель";

}

wstring getValue() const override {

return wstring(1, value);

}

};

// Функция переходов конечного автомата

State transition(State current, wchar\_t input) {

switch (current) {

case S0:

if (iswalpha(input) || input == L'\_') return I;

if (iswdigit(input)) return L;

if (input == L'/') return C;

if (operators.count(input)) return O;

if (separators.count(input)) return S;

return F;

case I:

if (iswalnum(input) || input == L'\_') return I;

return F;

case L:

if (iswdigit(input)) return L;

if (iswalpha(input) || input == L'\_') return E; // Если после цифр идут буквы или '\_', переходим в ошибочное состояние

return F;

case O:

if (input == L'/') return C; // Переход в состояние комментария

return F;

case C:

if (input == L'\n') return S0; // Комментарий завершается новой строкой

return C; // Оставаться в состоянии комментария

case S:

return F;

default:

return F;

}

}

void createToken(const wstring& token, size\_t position, size\_t tokenNumber, vector<pair<Token\*, wstring>>& tokens) {

bool hasInvalid = false;

for (wchar\_t ch : token) {

if (invalidSymbols.find(ch) != wstring::npos) {

hasInvalid = true;

break;

}

}

if (hasInvalid) {

wcout << L"Токен: " << token << L", ошибка: наличие недопустимых символов (Токен №" << tokenNumber << L")" << endl;

return; // Пропускаем токен с недопустимыми символами

}

wstring tokenType;

auto it\_kw = keywords.find(token);

if (it\_kw != keywords.end()) {

tokenType = it\_kw->second;

tokens.push\_back(make\_pair(new IdentifierToken(token), tokenType));

return;

}

auto it\_op = operators.find(token[0]);

if (it\_op != operators.end()) {

tokenType = it\_op->second;

tokens.push\_back(make\_pair(new OperatorToken(token[0]), tokenType));

return;

}

auto it\_sep = separators.find(token[0]);

if (it\_sep != separators.end()) {

tokenType = it\_sep->second;

tokens.push\_back(make\_pair(new SeparatorToken(token[0]), tokenType));

return;

}

if (token.find\_first\_not\_of(L"0123456789") == wstring::npos) {

tokenType = L"Литерал";

tokens.push\_back(make\_pair(new LiteralToken(token), tokenType));

return;

}

tokenType = L"Идентификатор"; // Если не обнаружено недопустимых символов, считаем токен идентификатором

tokens.push\_back(make\_pair(new IdentifierToken(token), tokenType));

}

void lexicalAnalysis(const wstring& input) {

State currentState = S0;

wstring token;

vector<pair<Token\*, wstring>> tokens;

size\_t position = 0;

size\_t tokenNumber = 1;

for (size\_t i = 0; i < input.length(); ++i) {

wchar\_t ch = input[i];

State nextState = transition(currentState, ch);

if (nextState == F || nextState == E) {

if (!token.empty()) {

if (nextState == E) {

wcout << L"Токен: " << token + ch << L", ошибка: требуется идентификатор (Токен №" << tokenNumber++ << L")" << endl;

token.clear();

while (i < input.length() && !iswspace(input[i]) && separators.find(input[i]) == separators.end() && operators.find(input[i]) == operators.end()) {

++i;

//token += input[i];

}

currentState = S0;

continue;

}

else {

createToken(token, position, tokenNumber++, tokens);

}

}

token.clear();

currentState = transition(S0, ch);

position = i;

if (currentState != F && currentState != C && !iswspace(ch)) {

token += ch;

}

}

else if (nextState == C) {

currentState = C;

while (i < input.length() && input[i] != L'\n') {

++i;

}

}

else {

token += ch;

currentState = nextState;

}

if (invalidSymbols.find(ch) != wstring::npos) {

if (!token.empty()) {

wcout << L"Токен: " << token << L", ошибка: наличие недопустимых символов (Токен №" << tokenNumber++ << L")" << endl;

token.clear();

}

while (i + 1 < input.length() && separators.find(input[i + 1]) == separators.end() && operators.find(input[i + 1]) == operators.end()) {

++i;

}

currentState = S0;

}

}

if (!token.empty()) {

createToken(token, position, tokenNumber++, tokens);

}

for (const auto& t : tokens) {

wcout << L"Токен: " << t.first->getValue() << L", тип: " << t.first->getType() << endl;

delete t.first;

}

}

int main() {

setlocale(LC\_ALL, "RUS");

wstring input = L"int maiававn() Pnt\_22 22pytr { int x = 42; if (x > 0) returШn x; } // Это комментарий\n if (x > 0) return x; // Еще один комментарий\n }";

wcout << L"Строка для показа работы программы: " << input << endl;

cout << "Итоги анализа: " << endl;

lexicalAnalysis(input);

return 0;

}