

**信息科学技术学院**

**《词法分析器实验报告》**

**课程名称** 编译原理

**学生姓名** 魏鹏超

**学生学号** 2208010423

计算机科学与技术 **专业** 224 **班**

**目 录**

[第1章 实验背景 1](#_Toc197601171)

[1.1 实验目的 1](#_Toc197601172)

[1.2 实验要求 1](#_Toc197601173)

[第2章 总体设计思路 1](#_Toc197601174)

[第3章 详细设计思路 3](#_Toc197601175)

[3.1 单词分类 3](#_Toc197601176)

[3.2 单词的形式化描述 4](#_Toc197601177)

[3.3 识别工具设计 5](#_Toc197601178)

[3.4 识别过程分析 13](#_Toc197601179)

[第4章 结果测试 14](#_Toc197601180)

[第5章 实验总结 15](#_Toc197601181)

[附录 15](#_Toc197601182)

# 实验背景

## 实验目的

1. 通过实验能够把握编译程序设计的方法和技术，模拟实现编译过程中重要的分析过程词法分析、语法分析、中间代码生成等，这是编译程序设计的核心。
2. 通过实验能够掌握、理解所学的计算机软件的经典基础理论，使学生具备解决难度较大的问题，处理复杂系统的设计与实现的能力以及选择适当的工具和开发环境，按照规范开发系统的能力，为专业能力的不断推升提供支持。

## 实验要求

设计**词法分析器**，完成以下目标 :

1. 程序设计语言中单词的分类
2. 单词的形式化描述
3. 单词识别工具的设计
4. 单词的识别过程分析

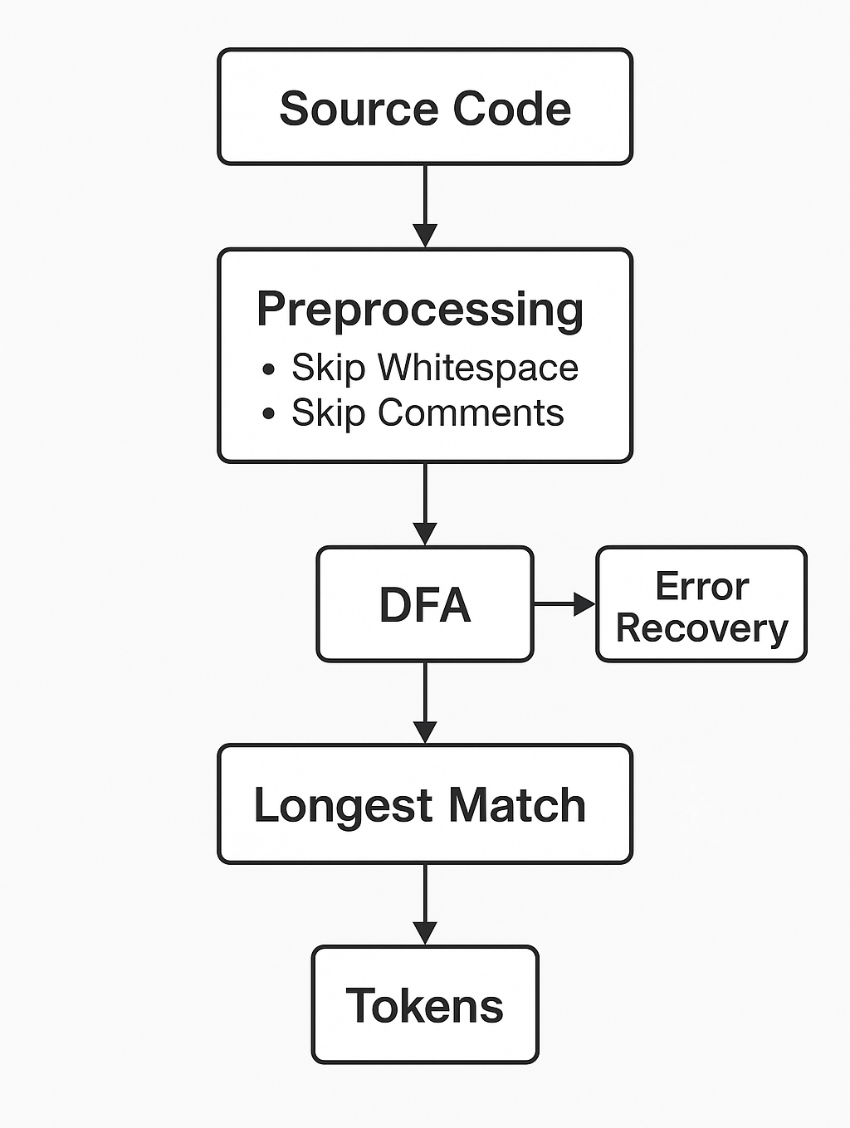
# 总体设计思路

本实验旨在设计并实现一个面向 C++ 语言的词法分析器，能够将源代码文本解析为规范的单词符号（Token）流，为后续语法分析和语义分析阶段提供基础支撑。总体设计思路采用**基于有限自动机（DFA）的逐字符扫描策略**，结合最长匹配优先原则，实现对标识符、关键字、数值常量、字符串、字符常量、运算符、分隔符等多种单词类别的有效识别。

在输入源代码后，分析器首先进行**预处理操作**，包括跳过空白符、换行符及注释，为正式分析做好准备。随后，分析器从当前位置驱动一个**有限状态自动机**（DFA），不断读取字符并进行状态转移，同时记录最近进入的接受状态和位置，实现**回溯到最长合法词素**的策略。每识别出一个词素，系统将其映射为对应的 Token 对象（包含类型、文本值、行列位置），并根据结果进行关键字判别或错误处理。

DFA 的构建采用**手动构造状态表的方式**完成，设计合理的状态编号及转移规则来处理不同类型的单词结构。例如，识别浮点数时需要识别整数部分、小数点、小数部分及科学计数法的 e/E 表达式；识别字符串与字符常量则需额外处理转义字符。整个自动机以表驱动方式实现，具有良好的可扩展性和调试性。

此外，为提高语法兼容性与用户体验，系统还支持**错误恢复机制**：若遇到非法词素，不直接终止分析过程，而是报告错误并跳过非法字符，继续分析后续内容。最终，分析器按 token 流的形式输出所有识别结果，为后续语法分析器提供稳定可靠的输入。



1. 设计流程图

# 详细设计思路

## 单词分类

在词法分析阶段，需要将源代码划分成若干“单词”（token），每个单词属于一个类别。

1. 单词分类表

|  |  |  |
| --- | --- | --- |
| **类别** | **说明** | **示例** |
| 关键字 (KEYWORD) | 语言保留标识符，具有特殊含义 | if, while |
| 标识符 (IDENT) | 用户自定义名字，用于变量、函数、类型等 | myVar, \_temp |
| 整数字面量 | 不含小数点和指数部分的数字 | 0, 123 |
| 浮点数字面量 | 含小数点或指数部分的数字 | 3.14, 1e-5 |
| 字符串字面量 | 用双引号括起来，可含转义字符 | "hello\n" |
| 字符字面量 | 用单引号括起来的单个字符或转义序列 | 'a', '\t' |
| 运算符 (OPERATOR) | 算术、逻辑、关系等操作符 | |  | | --- | |  |  |  | | --- | | +, ==, && | |
| 分隔符/界符 | 句末分号、逗号、括号、花括号等 | ;, ,, (, } |
| 注释 | // 或 /*...*/ 形式，通常被跳过 | // comment |
| 文件结束 (EOF) | 输入结束标志 | — |
| 错误 (UNKNOWN) | 不能识别或非法的字符序列 | @, 不完整的串字面量 |

## 单词的形式化描述

#### 标识符（IDENT）

IDENT → [A-Za-z\_] [A-Za-z0-9\_]\*

#### 整数与浮点字面量

INT\_LITERAL → [0-9]+

FLOAT\_LITERAL → ([0-9]\*\.[0-9]+ | [0-9]+\.[0-9]\*)([eE][+-]?[0-9]+)?

| [0-9]+[eE][+-]?[0-9]+

#### 字符串字面量 (STR\_LITERAL)

STR\_LITERAL → " ( [^"\\] | \\ . )\* "

#### 字符字面量 (CHAR\_LITERAL)

CHAR\_LITERAL → ' ( [^'\\] | \\ . ) '

#### 运算符与界符 (OPERATOR / PUNCTUATOR)

1）单字符：+ - \* / % = < > ! & | ^ ~ ; , ( ) { } [ ] : ?

2）双字符：== != <= >= ++ -- += -= \*= /= %= && || |= &= ^= ->

#### 注释

1）行注释：// .\* (\n | EOF)

2）块注释：/\* ( [^\*] | \\*+[^\*/] )\* \\*+/

## 识别工具设计

#### TokenType 枚举与 Token 结构

*enum class TokenType {*

*IDENT, KEYWORD,*

*INT\_LITERAL, FLOAT\_LITERAL,*

*STR\_LITERAL, CHAR\_LITERAL,*

*OPERATOR, PUNCTUATOR,*

*END\_OF\_FILE, UNKNOWN*

*};*

*struct Token {*

*TokenType type;*

*std::string lexeme;*

*int line, col;*

*};*

TokenType：列举了所有需要识别的 token 类型，包括标识符、关键字、各种常量、运算符、分隔符，以及文件结束和异常类型。

Token：保存了一个 token 的三个关键信息：

* type：token 的种类（如 INT\_LITERAL）。
* lexeme：对应的字符串，例如 "3.14e-2"。
* line, col：该 token 在源代码中的起始行列位置，用于错误定位或后续语义分析。

#### 关键字查找表

*static const std::unordered\_map<std::string, TokenType> keywords = {*

*{"if", TokenType::KEYWORD}, {"else", TokenType::KEYWORD},*

*{"while", TokenType::KEYWORD}, {"for", TokenType::KEYWORD},*

*{"return", TokenType::KEYWORD}, {"int", TokenType::KEYWORD},*

*{"float", TokenType::KEYWORD}, {"double", TokenType::KEYWORD},*

*{"char", TokenType::KEYWORD}, {"void", TokenType::KEYWORD},*

*};*

代码首先用一个哈希表列出了所有保留字。在识别到一个 IDENT 后，会查询这个表；如果匹配，就把它从 IDENT 转为 KEYWORD。

#### Lexer 类总体架构

**成员变量**

*class Lexer {*

*public:*

*Lexer(const std::string& src);*

*Token nextToken();*

*private:*

*void initTransitionTable();*

*void skipWhitespaceAndComments();*

*char peek() const;*

*char getChar();*

*[[noreturn]] void error(const std::string& msg);*

*std::string source;*

*size\_t curPos;*

*int line, col;*

*std::vector<std::vector<int>> transition;*

*std::unordered\_map<int, TokenType> acceptState;*

*};*

* source：保存整个待分析的源代码文本。
* curPos：当前读取位置（字符索引）。
* line, col：当前光标行列，用于记录 token 位置。
* transition：DFA 转移表；transition[state][c] 给出下一个状态编号，-1 表示无效转移。
* acceptState：将 DFA 中的“接受状态”映射为对应 TokenType。

**核心方法**

* initTransitionTable()：在构造函数中被调用，用于手动填充 transition 和 acceptState。
* skipWhitespaceAndComments()：在每次识别前，跳过空白字符和单/多行注释。
* nextToken()：驱动 DFA，执行最长匹配策略，产生下一个 token。
* peek() 与 getChar()：分别用于查看当前字符而不消费，或消费并更新行列。
* error()：遇到无法匹配的字符序列时，输出错误并终止程序。

#### 构造 DFA：initTransitionTable()

首先进行状态划分，将状态划分为**基础状态**和**操作符/分隔符**。

其中基础状态如*表-2*所示，操作符/分隔符从 20 开始，每遇到一种单字符或双字符符号，就分配新的状态编号。

1. 基础状态表

|  |  |
| --- | --- |
| **状态编号** | **对应状态** |
| 0 | 初始态 |
| 1 | 标识符 |
| 2 | 整数部分 |
| 3，4 | 浮点数小数点与小数部分 |
| 5-7 | 浮点数指数部分 |
| 8-10 | 字符串常量 |
| 11-13 | 字符常量 |

1. **示例: 标识符的 DFA 转移**

*// 初始态见到字母或下划线，进入 S\_IDENT (状态 1)*

*for(char c='A';c<='Z';++c) transition[0][c]=1;*

*for(char c='a';c<='z';++c) transition[0][c]=1;*

*transition[0]['\_']=1;*

*// 在状态 1，自环接受字母、数字和下划线*

*for(char c='A';c<='Z';++c) transition[1][c]=1;*

*// ... 同理填充其他字符 ...*

*acceptState[1]=TokenType::IDENT;*

1. **示例: 双字符运算符 ==**

*// addTwo 会生成两个新状态：s1 和 s2*

*auto addTwo = [&](char c, char c2, TokenType tp){*

*int s1=newState(), s2=newState();*

*transition[0][c]=s1;*

*transition[s1][c2]=s2;*

*acceptState[s1]=tp; // 单字符也接受，如单独的 '='*

*acceptState[s2]=tp; // 两字符接受 '=='*

*};*

*addTwo('=','=',TokenType::OPERATOR);*

1. **示例: 浮点数与指数**

*// 从整数状态 2 遇到 '.' 转到小数点后状态 3*

*transition[2]['.']=3;*

*// 在 3 上见数字，进入浮点主态 4，自环数字*

*for(char c='0';c<='9';++c) transition[3][c]=4;*

*for(char c='0';c<='9';++c) transition[4][c]=4;*

*acceptState[4]=TokenType::FLOAT\_LITERAL;*

*// 支持 e/E 指数*

*transition[4]['e']=transition[4]['E']=5;*

#### 主要逻辑：nextToken()

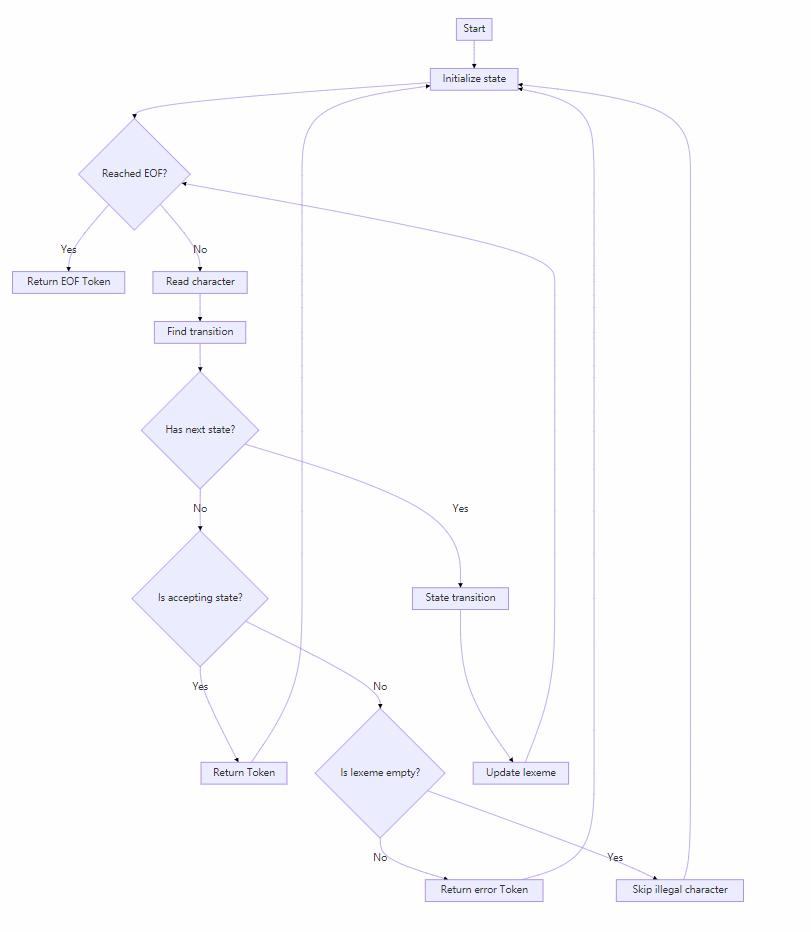
在词法分析的过程中，nextToken() 是核心驱动函数，它的作用是从源程序中逐字符读取内容，并按照事先构造好的有限状态自动机（DFA）进行状态转移，从而识别出一个合法的单词（Token）。每次调用该函数，词法分析器都会从当前位置开始尝试匹配一个完整的词素，直到成功识别出一个 Token，或发现非法输入，或到达文件末尾。

首先，nextToken() 会初始化状态变量，将状态设为起始状态 0，并记录当前字符所在的行号和列号，以便生成的 Token 能准确对应源代码中的位置。随后，进入主循环，每次从源字符串中读取下一个字符，并结合当前状态，通过查找状态转移表 transition[state][input] 来决定下一个状态。如果读取的字符是非法的（如不可打印字符或超出 ASCII 范围），会被统一映射为一个特殊输入。

若成功找到下一个状态，则将读取的字符追加到词素（lexeme）中，并更新状态，继续进行下一个字符的分析。这一过程持续进行，直到状态转移失败（即没有定义的下一个状态）为止。此时，程序会判断当前状态是否是一个接受状态。如果是接受状态，说明此时已经成功识别了一个合法的 Token，可以将其类型和内容打包返回；如果当前状态不是接受状态，但已经累积了一部分字符，则将这些字符作为非法 Token 报错返回；否则，说明当前字符本身就是非法的，词法器会跳过此字符，并尝试继续分析下一个词素。

在识别标识符（如 x, abc）之后，程序还会检查该词素是否属于关键字集合（如 int, if, while 等），如果是，则将其 Token 类型标记为关键字类型，而不是普通标识符。整个过程通过高效的状态跳转和逐字符分析实现，保证了词法分析器在处理不同类型的语言符号（如关键字、标识符、常量、分隔符、运算符等）时的准确性与可扩展性。

简而言之，nextToken() 实现了从源代码中提取语法单元的自动化机制，它结合了 DFA 状态转移、词素累积、接受状态判断、关键字匹配与错误处理等核心操作，是整个词法分析阶段的中枢模块。



1. nextToken()算法流程图

*Token Lexer::nextToken() {*

*if (pos >= (int)source.length()) return Token(-1, "", line, col); // EOF*

*int startLine = line;*

*int startCol = col;*

*int state = 0;*

*std::string lexeme;*

*while (true) {*

*char c = peekChar();*

*int inputChar = c; // For transition table*

*if (c == EOF || !isprint(c)) inputChar = 0;*

*int nextState = (inputChar >= 0 && inputChar < 128) ? transition[state][inputChar] : -1;*

*if (nextState == -1) {*

*// 处理接受状态*

*if (accepting[state] != -1) {*

*int tokenType = accepting[state];*

*if (tokenType == 0 && keywords.count(lexeme)) // 检查是否为关键字*

*tokenType = 0;*

*return Token(tokenType, lexeme, startLine, startCol);*

*} else {*

*if (!lexeme.empty())*

*return Token(-2, lexeme, startLine, startCol); // 错误 token*

*getChar(); // 跳过错误字符*

*return nextToken(); // 递归继续处理*

*}*

*} else {*

*state = nextState;*

*lexeme += getChar(); // 移动指针并记录词素*

*}*

*}*

*}*

#### 驱动程序：main()

main 函数主要用于从标准输入逐行读取，拼接成一个完整的字符串。然后不断调用 nextToken()，直到返回 END\_OF\_FILE。最后打印 token 类型（用整数表示）、lexeme 及其位置信息。

*int main() {*

*std::string code, line;*

*while (std::getline(std::cin, line)) {*

*code += line + "\n";*

*}*

*Lexer lexer(code);*

*Token tok;*

*while ((tok = lexer.nextToken()).type != TokenType::END\_OF\_FILE) {*

*std::cout << "Token(" << int(tok.type)*

*<< ", \"" << tok.lexeme << "\") at ["*

*<< tok.line << "," << tok.col << "]\n";*

*}*

*return 0;*

*}*

## 识别过程分析

**1. 输入扫描**

词法分析器从源代码中逐个字符地读取输入，直到遇到空白字符或注释时跳过。输入字符可能是标识符的起始字符（字母或下划线）、数字字符、运算符、分隔符等。

**2. DFA 状态转换**

依据当前字符，分析器根据 DFA 转移表决定当前状态的转移方向。

对于每个状态转移，检查是否达到了一个接受状态，接受状态表示一个有效的词法单元。

**3. 输出 Token**

达到接受状态后，生成一个 token。

返回 token 类型、文本内容以及该 token 在源代码中的位置信息（如行号、列号），并将光标移动到下一个字符位置，继续识别下一个词法单元。

**4. 错误检测**

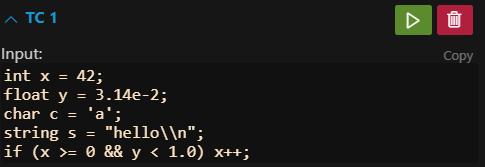
若当前字符无法转移到任何有效状态，则判断为非法字符或词法错误，报告错误并跳过该字符，继续分析后续部分。

**5. 结束标识符**

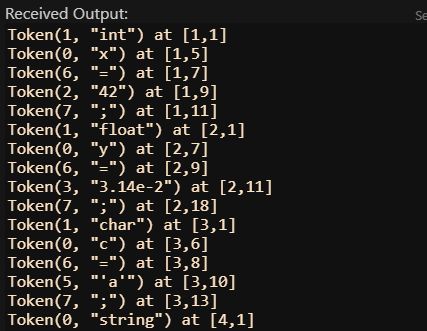
识别到文件末尾时，返回 END\_OF\_FILE token，表示词法分析结束。

# 结果测试

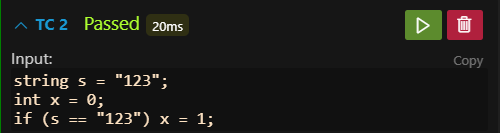
1. **测试用例一**



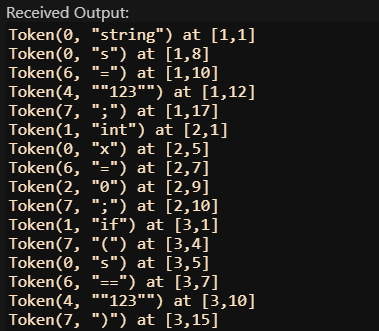
1. 测试用例1



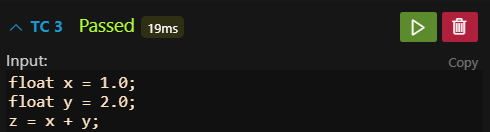
1. 输出结果
2. **测试用例二**



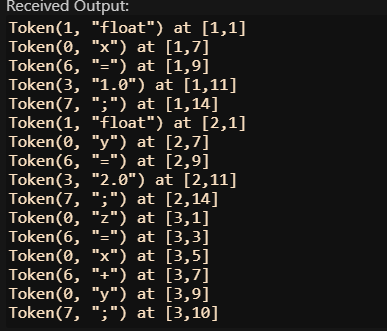
1. 测试用例2



1. 输出结果
2. **测试用例三**



1. 测试用例2



1. 输出结果

# 实验总结

在本次实验中，我实现了一个简单的词法分析器（Lexer），用于分析和识别程序设计语言中的基本单词符号（Token）。整个实验围绕词法分析的核心任务展开：读取源代码字符串，跳过空白符和注释，通过有限状态自动机（DFA）识别并输出各类合法Token及其位置信息。为了支持语言的基本语法结构，首先明确了常见Token的分类，包括关键字（如if、while等）、标识符、整数和浮点字面量、字符串和字符字面量、运算符与分隔符等。接着通过构建状态转移表，形式化地描述了各种Token的识别规则，并将这些规则内嵌于自动机结构中，用程序逻辑实现状态迁移和接受状态的判断。

在具体实现过程中，我设计了一个Lexer类，并利用二维数组构建了DFA的转移表，通过读取输入字符并逐步状态转移实现Token的识别。当识别成功时，将相应的词法单元返回；当遇到非法字符时，则报错提示位置，并继续分析。这一过程中还处理了浮点数的科学计数法、字符串和字符中的转义字符、以及单行和多行注释的跳过等情况，使词法分析器更为健壮。

通过本次实验，我对词法分析的整体流程有了清晰的认识，特别是自动机在词法规则中的应用机制。动手实现DFA转移表和接受状态，加深了我对抽象理论与实际编程之间关系的理解。通过对各种Token边界情况的处理，也增强了我对代码健壮性与容错设计的关注。此外，封装Lexer类的过程中，提高了我的程序结构设计能力和调试能力。

本次实验不仅提升了我对编译原理中“词法分析”阶段的理解，也让我在实践中体会到了编译器前端设计的复杂性与工程性。未来如果进一步开发语法分析器，当前的Lexer模块也可作为前端的基础模块重复利用，为完整的编译器实现打下良好基础。

附录

代码 :

#include <iostream>

#include <string>

#include <vector>

#include <unordered\_map>

#include <cctype>

// Token 类型枚举

enum class TokenType {

    IDENT, KEYWORD,

    INT\_LITERAL, FLOAT\_LITERAL,

    STR\_LITERAL, CHAR\_LITERAL,

    OPERATOR, PUNCTUATOR,

    END\_OF\_FILE, UNKNOWN

};

struct Token {

    TokenType type;

    std::string lexeme;

    int line, col;

};

static const std::unordered\_map<std::string, TokenType> keywords = {

    {"if", TokenType::KEYWORD}, {"else", TokenType::KEYWORD},

    {"while", TokenType::KEYWORD}, {"for", TokenType::KEYWORD},

    {"return", TokenType::KEYWORD}, {"int", TokenType::KEYWORD},

    {"float", TokenType::KEYWORD}, {"double", TokenType::KEYWORD},

    {"char", TokenType::KEYWORD}, {"void", TokenType::KEYWORD},

};

class Lexer {

public:

    Lexer(const std::string& src)

      : source(src), curPos(0), line(1), col(1) { initTransitionTable(); }

    Token nextToken() {

        skipWhitespaceAndComments();

        if (curPos >= source.size())

            return {TokenType::END\_OF\_FILE, "", line, col};

        int state = 0, lastAccept = -1;

        size\_t startPos = curPos, lastPos = curPos;

        int startLine = line, startCol = col;

        std::string lexeme;

        while (true) {

            // peek : [CS] Read Data

            char c = peek();

            int nxt = (c >= 0 && c < 128) ? transition[state][(unsigned char)c] : -1;

            if (nxt < 0) break;

            state = nxt;

            lexeme.push\_back(getChar());

            if (acceptState.count(state)) {

                lastAccept = state;

                lastPos = curPos;

            }

        }

        if (lastAccept >= 0) {

            curPos = lastPos;

            std::string tk = source.substr(startPos, lastPos - startPos);

            TokenType tp = acceptState[lastAccept];

            // KEYWORD is the subset of IDENT

            if (tp == TokenType::IDENT && keywords.count(tk)) tp = TokenType::KEYWORD;

            return {tp, tk, startLine, startCol};

        }

        std::string bad = lexeme.empty() ? std::string(1, peek()) : lexeme;

        error("Invalid token: " + bad);

        getChar();

        return {TokenType::UNKNOWN, bad, line, col};

    }

private:

    std::string source;

    size\_t curPos;

    int line, col;

    std::vector<std::vector<int>> transition;

    std::unordered\_map<int, TokenType> acceptState;

    void initTransitionTable() {

        int N = 100;

        transition.assign(N, std::vector<int>(128, -1));

        acceptState.clear();

        const int S\_IDENT=1, S\_INT=2, S\_DOT=3, S\_FLOAT=4, S\_EXP=5, S\_EXP\_SIGN=6, S\_EXP\_NUM=7;

        const int S\_STR=8, S\_STR\_CONT=9, S\_STR\_ESC=10;

        const int S\_CHAR=11, S\_CHAR\_CONT=12, S\_CHAR\_ESC=13;

        int st=20;

        auto newState=[&](){ return st++; };

        // identifier

        for(char c='A';c<='Z';++c) transition[0][c]=S\_IDENT;

        for(char c='a';c<='z';++c) transition[0][c]=S\_IDENT;

        transition[0]['\_']=S\_IDENT;

        for(char c='A';c<='Z';++c) transition[S\_IDENT][c]=S\_IDENT;

        for(char c='a';c<='z';++c) transition[S\_IDENT][c]=S\_IDENT;

        for(char c='0';c<='9';++c) transition[S\_IDENT][c]=S\_IDENT;

        transition[S\_IDENT]['\_']=S\_IDENT;

        acceptState[S\_IDENT]=TokenType::IDENT;

        // integer & float

        for(char c='0';c<='9';++c) transition[0][c]=S\_INT;

        for(char c='0';c<='9';++c) transition[S\_INT][c]=S\_INT;

        transition[S\_INT]['.']=S\_DOT;

        acceptState[S\_INT]=TokenType::INT\_LITERAL;

        transition[0]['.']=S\_DOT;

        for(char c='0';c<='9';++c) transition[S\_DOT][c]=S\_FLOAT;

        for(char c='0';c<='9';++c) transition[S\_FLOAT][c]=S\_FLOAT;

        acceptState[S\_FLOAT]=TokenType::FLOAT\_LITERAL;

        transition[S\_FLOAT]['e']=transition[S\_FLOAT]['E']=S\_EXP;

        for(char c='0';c<='9';++c) transition[S\_EXP][c]=S\_EXP\_NUM;

        transition[S\_EXP]['+']=transition[S\_EXP]['-']=S\_EXP\_SIGN;

        for(char c='0';c<='9';++c) transition[S\_EXP\_SIGN][c]=S\_EXP\_NUM;

        for(char c='0';c<='9';++c) transition[S\_EXP\_NUM][c]=S\_EXP\_NUM;

        acceptState[S\_EXP\_NUM]=TokenType::FLOAT\_LITERAL;

        // string literal

        transition[0]['"']=S\_STR;

        for(int i=0;i<128;++i) transition[S\_STR][i]=S\_STR\_CONT;

        for(int i=0;i<128;++i) transition[S\_STR\_CONT][i]=S\_STR\_CONT;

        transition[S\_STR\_CONT]['\\']=S\_STR\_ESC;

        for(int i=0;i<128;++i) transition[S\_STR\_ESC][i]=S\_STR\_CONT;

        int S\_STR\_END=newState();

        transition[S\_STR\_CONT]['"']=S\_STR\_END;

        acceptState[S\_STR\_END]=TokenType::STR\_LITERAL;

        // char literal

        transition[0]['\'']=S\_CHAR;

        for(int i=0;i<128;++i) transition[S\_CHAR][i]=S\_CHAR\_CONT;

        for(int i=0;i<128;++i) transition[S\_CHAR\_CONT][i]=S\_CHAR\_CONT;

        transition[S\_CHAR\_CONT]['\\']=S\_CHAR\_ESC;

        for(int i=0;i<128;++i) transition[S\_CHAR\_ESC][i]=S\_CHAR\_CONT;

        int S\_CHAR\_END=newState();

        transition[S\_CHAR\_CONT]['\'']=S\_CHAR\_END;

        acceptState[S\_CHAR\_END]=TokenType::CHAR\_LITERAL;

        // operators & punctuators

        auto addOp=[&](char c, TokenType tp){ int s=newState(); transition[0][c]=s; acceptState[s]=tp; };

        auto addTwo=[&](char c, char c2, TokenType tp){ int s1=newState(), s2=newState(); transition[0][c]=s1; transition[s1][c2]=s2; acceptState[s1]=tp; acceptState[s2]=tp; };

        addTwo('=','=',TokenType::OPERATOR);

        addTwo('!','=',TokenType::OPERATOR);

        addTwo('<','=',TokenType::OPERATOR);

        addTwo('>','=',TokenType::OPERATOR);

        addTwo('+','+',TokenType::OPERATOR); addTwo('+','=',TokenType::OPERATOR);

        addTwo('-','-',TokenType::OPERATOR); addTwo('-','=',TokenType::OPERATOR); addTwo('-','>',TokenType::OPERATOR);

        addTwo('\*','=',TokenType::OPERATOR); addTwo('/','=',TokenType::OPERATOR); addTwo('%','=',TokenType::OPERATOR);

        addTwo('&','&',TokenType::OPERATOR); addTwo('&','=',TokenType::OPERATOR);

        addTwo('|','|',TokenType::OPERATOR); addTwo('|','=',TokenType::OPERATOR);

        addTwo('^','=',TokenType::OPERATOR); addOp('~',TokenType::OPERATOR);

        addOp(';',TokenType::PUNCTUATOR); addOp(',',TokenType::PUNCTUATOR);

        addOp('(',TokenType::PUNCTUATOR); addOp(')',TokenType::PUNCTUATOR);

        addOp('{',TokenType::PUNCTUATOR); addOp('}',TokenType::PUNCTUATOR);

        addOp('[',TokenType::PUNCTUATOR); addOp(']',TokenType::PUNCTUATOR);

        addOp(':',TokenType::PUNCTUATOR); addOp('?',TokenType::PUNCTUATOR);

    }

    char getChar() { char c=source[curPos++]; if(c=='\n'){line++;col=1;}else col++; return c; }

    char peek() const { return curPos<source.size()?source[curPos]:'\0'; }

    void skipWhitespaceAndComments() {

        while(true) {

            char c=peek();

            if(isspace(c)){getChar();continue;}

            if(c=='/'&&curPos+1<source.size()){

                if(source[curPos+1]=='/') { getChar();getChar(); while(peek()!='\n'&&peek()!='\0') getChar(); continue; }

                if(source[curPos+1]=='\*') { getChar();getChar(); while(!(peek()=='\*'&&curPos+1<source.size()&&source[curPos+1]=='/')){ if(peek()=='\0') return; getChar(); } getChar();getChar(); continue; }

            }

            break;

        }

    }

    [[noreturn]] void error(const std::string& msg){ std::cerr<<"[Lexer Error] Line "<<line<<", Col "<<col<<": "<<msg<<std::endl; exit(1); }

};

int main() {

    std::string code, line;

    while (std::getline(std::cin, line)) { code += line + "\n"; }

    Lexer lexer(code);

    Token tok;

    while ((tok = lexer.nextToken()).type != TokenType::END\_OF\_FILE) {

        std::cout << "Token(" << int(tok.type) << ", \"" << tok.lexeme << "\") at ["

                  << tok.line << "," << tok.col << "]\n";

    }

    return 0;

}