编号：_____

实习	一	二	三	四	五	六	七	八	九	十	总评	教师签名
成绩												

武汉大学计算机学院

《编译原理》课程

语法分析

实习报告

专业（班）：编译原理

学生学号：2022302111297

学生姓名：饶浩杰

任课教师：杜 卓 敏

2025 年 6 月 3 日

第一部分 语言语法规则（自然语言描述）

我简单构造的Mini语言是一种类C语言的简化版本，基于上一次词法分析的任务，支持基本的语法结构。其主要语法规则如下：

1. **词法元素**：
    - 关键字：int、double、float、if、then、else、return、while
    - 运算符：+、-、*、/、=、==、<、<=、>、>=
    - 分隔符：(、)、[、]、{、}、,、;
    - 标识符：由字母开头，后跟字母、数字的序列
    - 常量：整型常量（如10、-20）和浮点型常量（如3.14、-2.5）
2. **注释**：支持单行注释，以'//'开头，到行尾结束
3. **表达式**：支持算术表达式、比较表达式和赋值表达式
4. **语句**：支持赋值语句、条件语句、循环语句、返回语句和变量声明语句
5. **程序结构**：由函数定义组成，每个函数包含一系列语句

第二部分 文法定义

其文法可以用 BNF 表示如下：

1. 程序结构层

定义整个程序的基本结构

```
<program> ::= <function-definition>+  // 程序由一个或多个函数定义组成
```

2. 函数定义层

定义函数的结构

```
<function-definition> ::= <type-specifier> <identifier> '(' <parameter-
list>? ')' <compound-statement> // 函数定义，eg. int func(int a, int b) {
... }
<type-specifier> ::= 'int' | 'double' | 'float'  // 函数返回类型
<parameter-list> ::= <parameter-declaration> | <parameter-list> ','
<parameter-declaration> // 参数列表，eg. int a, int b
<parameter-declaration> ::= <type-specifier> <identifier> // 参数声明，eg.
int a
```

3. 语句层

定义各种语句结构

```
<compound-statement> ::= '{' <statement-list>? '}'  // 复合语句（代码块）
<statement-list> ::= <statement> | <statement-list> <statement>
<statement> ::= <expression-statement>  // 表达式语句
              | <compound-statement>    // 复合语句
              | <selection-statement>   // 选择语句（if-else）
              | <iteration-statement>   // 循环语句（while）
              | <return-statement>      // 返回语句
              | <variable-declaration>  // 变量声明语句
```

4. 具体语句定义

定义各种具体语句的语法；

1. 句末必须以分号结束；
2. 循环/分支语句的条件放置于括号内；
3. 返回语句必须以return开始；
4. 支持带有then关键字的if语句形式；

```
<expression-statement> ::= <expression>? ';'
<selection-statement> ::= 'if' '(' <expression> ')' <statement> ('else'
<statement>)?
                        | 'if' '(' <expression> ')' 'then' <statement>
('else' <statement>)?
<iteration-statement> ::= 'while' '(' <expression> ')' <statement>
<return-statement> ::= 'return' <expression>? ';'
<variable-declaration> ::= <type-specifier> <identifier> ('='
<expression>)? ';'
```

5. 表达式层

定义表达式的层次结构

```
<expression> ::= <assignment-expression>
<assignment-expression> ::= <identifier> '=' <logical-or-expression> |
<logical-or-expression> // 赋值表达式，eg. a = b + c
```

6. 逻辑表达式层

定义逻辑运算的优先级

```
<logical-or-expression> ::= <logical-and-expression> | <logical-or-
expression> '||' <logical-and-expression> // 逻辑或表达式，eg. a || b
<logical-and-expression> ::= <equality-expression> | <logical-and-
```

```
expression> '&&' <equality-expression> // 逻辑与表达式，eg. a && b
<equality-expression> ::= <relational-expression> | <equality-expression>
'==' <relational-expression> // 相等表达式，eg. a == b
```

7. 关系表达式层

定义关系运算

```
<relational-expression> ::= <additive-expression> // 关系表达式，eg. a < b +
c
                          | <relational-expression> '<' <additive-
expression> // 小于表达式，eg. a < b
                          | <relational-expression> '>' <additive-
expression> // 大于表达式，eg. a > b
                          | <relational-expression> '<=' <additive-
expression> // 小于等于表达式，eg. a <= b
                          | <relational-expression> '>=' <additive-
expression> // 大于等于表达式，eg. a >= b
```

8. 算术表达式层

定义算术运算的优先级

```
<additive-expression> ::= <multiplicative-expression> // 加减表达式，eg. a +
b - c
                        | <additive-expression> '+' <multiplicative-
expression> // 加法表达式，eg. a + b
                        | <additive-expression> '-' <multiplicative-
expression> // 减法表达式，eg. a - b
                        | <additive-expression> '|' <multiplicative-
expression> // 位或表达式，eg. a | b
                        | <additive-expression> '&' <multiplicative-
expression> // 位与表达式，eg. a & b
<multiplicative-expression> ::= <primary-expression> // 乘除表达式，eg. a * b
/ c
                              | <multiplicative-expression> '*' <primary-
expression> // 乘法表达式，eg. a * b
                              | <multiplicative-expression> '/' <primary-
expression> // 除法表达式，eg. a / b
```

9. 基本表达式层

定义最基本的表达式元素

```
<primary-expression> ::= <identifier>  // 标识符，eg. a
                       | <constant>    // 常量，eg. 10
```

```
                        | '(' <expression> ')'  // 括号表达式，eg. (a + b)
                        | <identifier> '(' <argument-list>? ')' // 函数调用，
eg. factorial(5)
<argument-list> ::= <expression> | <argument-list> ',' <expression> // 参数
列表，eg. a, b+c
```

10. 词法元素层

定义基本的词法单元

```
<identifier> ::= <letter> (<letter> | <digit>)*  // 标识符：字母开头，后跟字母
或数字，eg. a, b, c
<constant> ::= <integer-constant> | <floating-constant>  // 常量：整数或浮点
数，eg. 10, 3.14
<integer-constant> ::= ['-']?<digit>+  // 整数常量：可选负号加一个或多个数字，eg.
10, -20
<floating-constant> ::= ['-']?<digit>+ '.' <digit>*  // 浮点常量：可选负号加数
字.数字，eg. 3.14, -2.5
<letter> ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'  // 字母，eg. a,
b, c
<digit> ::= '0' | '1' | ... | '9'  // 数字，eg. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

第三部分 语法分析算法

本项目采用递归下降分析方法进行语法分析。该方法是一种自顶向下的分析方法，通过一组互相递归的函数来实现对输入的分析。

递归下降分析的基本思想是：为文法中的每个非终结符设计一个函数，该函数负责分析该非终结符所代表的语法结构。当需要分析某个非终结符时，就调用相应的函数。

算法的主要步骤如下：

1. **初始化**：设置词法分析器，准备读取第一个Token
2. **开始分析**：从文法的开始符号（）对应的函数开始
3. **递归处理**：
    - 对于每个非终结符，调用相应的函数进行处理
    - 对于选择结构（|），通过前瞻一个或多个Token来决定选择哪个分支
    - 对于重复结构（+、*），使用循环处理
4. **匹配终结符**：当遇到终结符时，检查当前Token是否匹配，匹配则读取下一个Token，否则报错
5. **错误处理**：当发生语法错误时，尝试恢复并继续分析

在本项目的实现中，每个非终结符都对应着一个静态函数，这些函数相互调用形成递归下降分析器的骨架。例如：

```
// 程序入口
static bool program() {
```

```
    // 获取第一个token
    g_token = getNextToken();

    while (g_token.code != TK_EOF) {
        // 检查是否为函数定义的开始（类型说明符）
        if (g_token.code == KW_INT || g_token.code == KW_DOUBLE ||
g_token.code == KW_FLOAT) {
            if (!functionDefinition()) {
                // 处理错误...
            }
        } else {
            // 处理错误...
        }
    }

    return success;
}

// 表达式语句解析示例
static bool expressionStatement() {
    if (g_token.code != TK_SEMOCOLOM) {
        if (!expression()) {
            return false;
        }
    }

    if (!match(TK_SEMOCOLOM)) {
        addDetailedError("表达式语句缺少分号 ';'");
        return false;
    }

    return true;
}

// 匹配特定类型的Token
static bool match(TokenCode code) {
    if (g_token.code == code) {
        g_token = getNextToken();
        return true;
    }
    return false;
}
```

在这个实现中，每个语法分析函数通过调用其他函数和检查当前Token来确定是否符合文法规则。它们返回布尔值来表示分析是否成功。

第四部分 出错处理出口

为了提高编译器的健壮性和用户体验，本项目实现了以下错误处理机制：

1. **词法错误处理**：

- 非法标识符：出现非法字符时，捕获并报告错误
- 非法数字格式：如多个小数点，报告错误
- 未闭合的注释：在文件结束前注释未闭合，报告错误

2. **语法错误处理**：

- 详细的错误信息：包含当前处理的Token和具体的错误描述
- 同步集合（Synchronizing Sets）：为错误恢复定义同步集合，当发生错误时，跳过输入直到遇到同步集合中的Token
- 智能恢复：当检测到语法错误时，系统能够跳过出错的部分，继续分析后续的代码

3. **具体错误恢复策略**：

- **跳过直到同步点**：使用`skipUntil`函数跳过错误的Token，直到遇到可以重新开始分析的Token
- **增强的错误报告**：通过`addDetailedError`函数提供更详细的错误信息，包含当前Token的值
- **有限次数错误尝试**：避免无限循环解析错误的情况

实现的关键代码示例：

```
// 添加详细的错误信息
static void addDetailedError(const std::string& message) {
    std::string detailedMessage = message;
    if (g_token.code != TK_EOF) {
        detailedMessage += " (当前Token: '" + g_token.value + "')";
    }
    ParserError error = { g_token.line, detailedMessage };
    g_errors.push_back(error);  // 确保错误被添加到g_errors向量中
    g_hasError = true;  // 设置错误标志
    std::cerr << "Syntax Error at line " << g_token.line << ": " <<
detailedMessage << std::endl;
}

// 跳过错误的Token直到同步点
static void skipUntil(std::vector<TokenCode> syncSet) {
    // 记录跳过的token，用于错误报告
    std::string skippedTokens = "";
    int skipCount = 0;
    const int maxDisplayTokens = 3;  // 最多显示几个跳过的token

    while (g_token.code != TK_EOF) {
        for (TokenCode code : syncSet) {
            if (g_token.code == code) {
                if (skipCount > 0) {
                    std::string message = "已跳过 " +
std::to_string(skipCount) + " 个token";
                    if (!skippedTokens.empty()) {
                        message += " (包括: " + skippedTokens + ")";
                    }
                    std::cerr << "Info: " << message << std::endl;
                }
                return;
            }
```

```
            }

            // 记录跳过的token
            if (skipCount < maxDisplayTokens) {
                if (!skippedTokens.empty()) {
                    skippedTokens += ", ";
                }
                skippedTokens += "'" + g_token.value + "'";
            } else if (skipCount == maxDisplayTokens) {
                skippedTokens += "...";
            }
            skipCount++;

            g_token = getNextToken();
        }
    }
```

通过这样的错误处理机制，本编译器能够在遇到错误时提供有用的诊断信息，同时继续分析程序的其余部分，提高用户体验。

第五部分 测试计划与结果

为了验证Mini语言编译器的正确性和健壮性，我设计了以下测试方案：

测试环境

- 操作系统：macOS
- 编译器：Clang/GCC
- 测试工具：自定义测试脚本（run_tests.sh）

测试用例设计

我设计了三个主要测试用例，涵盖不同方面的语法特性和错误处理能力：

1. **test1.txt**：基本语法功能测试，包含：
    - 变量声明和初始化
    - 正负整数常量
    - if-else结构
    - if-then结构（扩展语法）
    - if-then-else结构
    - while循环结构
    - 返回语句
2. **test2.txt**：错误处理测试，包含各种语法错误：
    - 缺少分号
    - 条件语句缺少括号
    - 表达式缺少操作数

- 语句缺少分号
- 错位的大括号

3. **test3.txt**：复杂功能测试，包含：

- 函数定义和函数调用
- 递归函数（阶乘、斐波那契）
- 嵌套条件语句
- 未定义运算符的错误处理（模运算%）
- 复杂的控制流结构

测试结果

1. 基本语法功能测试（test1.txt）

成功解析了所有语法结构，包括：

- 正确识别变量声明和初始化
- 正确处理正负整数常量
- 支持多种if语句形式（if-else、if-then、if-then-else）
- 正确处理while循环和return语句

2. 错误处理测试（test2.txt）

系统成功检测并报告了所有语法错误：

- 检测到缺少分号的错误
- 检测到缺少括号的错误
- 检测到表达式缺少操作数的错误
- 检测到错位的大括号

同时，错误恢复机制使得解析器能够继续分析后续代码，而不是在第一个错误处终止。

3. 复杂功能测试（test3.txt）

成功解析了复杂的函数定义和调用，包括递归函数。同时，系统能够正确识别未定义的运算符（如模运算%）并报告错误。

测试总结

1. **功能完整性**：

- 成功实现了所有语法特性的解析
- 支持扩展的if-then语法
- 能够处理函数定义和调用，包括递归函数

2. **错误处理能力**：

- 能够准确定位并报告语法错误
- 错误信息详细，包含当前Token的值
- 错误恢复机制有效，能够跳过错误继续分析

3. **健壮性**：
    - 能够处理各种边界情况和不规范输入
    - 不会因为语法错误而崩溃

通过这些测试，验证了Mini语言编译器能够正确解析符合文法的程序，并能合理处理语法错误，提供有用的错误信息，帮助用户定位和修复问题。

第六部分 附录

完整代码

```
// lexer.h
#ifndef LEXER_H
#define LEXER_H

#include <string>
#include <vector>
#include <cstdio>

/* 单词编码 */
enum TokenCode
{
    /* 未定义 */
    TK_UNDEF = 0,
    /* 关键字 */
    KW_INT,      // int
    KW_DOUBLE,   // double
    KW_FLOAT,    // float
    KW_IF,       // if
    KW_THEN,     // then
    KW_ELSE,     // else
    KW_RETURN,   // return
    KW_WHILE,    // while
    /* 运算符 */
    TK_PLUS,     // +
    TK_MINUS,    // -
    TK_STAR,     // *
    TK_DIVIDE,   // /
    TK_ASSIGN,   // =
    TK_BITAND,   // &
    TK_AND,      // &&
    TK_EQ,       // ==
    TK_LT,       // <
    TK_LEQ,      // <=
    TK_GT,       // >
    TK_GEQ,      // >=
    TK_BITOR,    // |
    TK_OR,       // ||
    /* 分隔符 */
    TK_OPENPA,   // (
```

```
    TK_CLOSEPA,  // )
    TK_OPENBR,   // [
    TK_CLOSEBR,  // ]
    TK_BEGIN,    // {
    TK_END,      // }
    TK_COMMA,    // ,
    TK_SEMOCOLOM, // ;
    /* 常量 */
    TK_INT,      // 整型常量
    TK_DOUBLE,   // 浮点型常量
    /* 标识符 */
    TK_IDENT,
    /* 文件结束 */
    TK_EOF       // 文件结束标记
};

/* 符号表类型 */
enum TableTypeId {
    Table_TAG = 1,
    Table_CONSTANT,
};

/* Token属性结构体 */
struct TokenAttr {
    TokenCode code;      // Token类型
    int line;            // 行号
    TableTypeId type;    // 符号表类型
    int table_row;       // 符号表行号
    std::string value;   // Token的值
};

/* 错误信息结构体 */
struct ErrorInfo {
    int line;               // 错误所在行
    std::string message;    // 错误信息
};

/* INFO 词法分析器接口 */

// 初始化词法分析器
void initLexer(FILE* fp);

// 获取下一个Token
TokenAttr getNextToken();

// 回退一个Token（用于预读）
void ungetToken();

// 获取当前行号
int getCurrentLine();

// 获取所有错误信息
const std::vector<ErrorInfo>& getErrors();
```

```
// 重置词法分析器
void resetLexer();

// 关闭词法分析器
void closeLexer();

std::string getTokenName(TokenCode code);

#endif /* LEXER_H */
```

```
// lexer.cpp
#include "lexer.h"
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>
#include <cstdlib>

/* 全局变量 */
static FILE* g_fp = nullptr;               // 文件指针
static int g_row = 1;                      // 当前行号
static TokenAttr g_lastToken;              // 上一个Token（用于回退）
static bool g_hasUnget = false;            // 是否有回退的Token
static std::vector<ErrorInfo> g_errors;    // 错误信息列表

/* 关键字表 */
static const int keyWordTokenNum = 8;
static const char keyWords[][10] = {
    "int", "double", "float", "if", "then", "else", "return", "while"
};

/* TokenCode到关键字的映射 */
static const TokenCode keyWordCodes[] = {
    KW_INT, KW_DOUBLE, KW_FLOAT, KW_IF, KW_THEN, KW_ELSE, KW_RETURN,
KW_WHILE
};

/**
 * 数字后可跟随的字符
 * 用于"吞噬"错误的数字词法，确保正确处理数字后接的合法字符。
 * 例如，在"10+"中，数字10后跟的+是合法的。
*/
static const int numberNextPassNum = 16;
static const char numberNextPass[100] = {
    '+', '-', '*', '/', '=', '>', '<', '&', '|',
    '^', '%', '?', ')', ',', ';', ']'
};

/* INFO 符号表 */
static std::map<TokenCode, int> tokenCodeMap;
```

```
static std::map<std::string, int> constantsMap;

/* 辅助函数 */
// 判断是否为字母
static bool isLetter(char ch) {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}

// 判断是否为非ASCII字符（多字节UTF-8字符）
// 支持处理中文等非ASCII字符，增强词法分析器的健壮性
static bool isNonAscii(unsigned char ch) {
    return (ch >= 0x80);  // UTF-8编码中，第一个字节>=0x80表示多字节字符
}

// 检查数字后是否为合法字符
// 如果返回true，说明ch是数字后面可以直接跟的合法字符
// 如果返回false，说明ch是非法字符，表明这是一个数字词法错误
static bool checkNumberNext(char ch) {
    for (int i = 0; i < numberNextPassNum; i++) {
        if (ch == numberNextPass[i])
            return true;
    }
    return false;
}

// 尝试获取关键字ID
// 如果token是关键字，返回其在keyWords数组中的索引
// 否则返回-1，表示这是一个普通标识符
static int tryGetKeyWordID(const std::string& token) {
    for (int i = 0; i < keyWordTokenNum; i++) {
        if (token == keyWords[i])
            return i;
    }
    return -1;
}

// 添加错误信息
// 将错误添加到错误列表中，并输出到标准错误流
static void addError(const std::string& message) {
    ErrorInfo error = { g_row, message };
    g_errors.push_back(error);
    std::cerr << "Error at line " << g_row << ": " << message <<
std::endl;
}

// 处理一个Token
// 这是词法分析器的核心函数，负责从输入流中读取字符并识别Token
static TokenAttr processToken() {
    TokenAttr result;
    result.line = g_row;
    result.type = Table_TAG;
    result.table_row = 0;

    char ch;
```

```
    std::string token = "";
    TokenCode code = TK_UNDEF;

    // 跳过空白字符（空格、制表符、换行符等）
    while (true) {
        ch = fgetc(g_fp);
        if (ch == EOF) {
            result.code = TK_EOF;
            result.value = "EOF";
            return result;
        }

        if (ch == '#')  { // 特殊终止符
            result.code = TK_EOF;
            result.value = "#";
            return result;
        }

        if (ch == '\n') {
            g_row++;  // 遇到换行符，行号加1
            continue;
        }

        if (ch == ' ' || ch == '\t' || ch == '\r')
            continue;

        break;  // 找到非空白字符，退出循环
    }

    // 处理各种Token类型
    if (isLetter(ch)) {  // 标识符或关键字
        token = ch;
        // 读取标识符或关键字的剩余部分
        while (true) {
            ch = fgetc(g_fp);
            if (!(isLetter(ch) || isdigit(ch))) {
                ungetc(ch, g_fp);  // 回退一个字符
                break;
            }
            token += ch;
        }

        // 检查是否为关键字
        int keywordIndex = tryGetKeyWordID(token);
        if (keywordIndex != -1) {
            code = keyWordCodes[keywordIndex];
        } else {
            code = TK_IDENT;  // 不是关键字，是标识符
        }
    }
    else if (isdigit(ch)) {  // 处理数字（整数或浮点数）
        token = ch;
        int dotCount = 0;
```

```
        while (true) {
            ch = fgetc(g_fp);
            if (!(isdigit(ch) || ch == '.')) {
                break;
            }

            if (ch == '.') {
                dotCount++;  // 记录小数点的数量
            }

            token += ch;
        }

        // 检查数字格式是否正确
        if (!checkNumberNext(ch)) {
            // 非法后缀，继续读取错误标记
            token += ch;
            while (true) {
                ch = fgetc(g_fp);
                if (!(isLetter(ch) || isdigit(ch))) {
                    ungetc(ch, g_fp);
                    break;
                }
                token += ch;
            }
            code = TK_UNDEF;
            addError("Invalid number format: " + token);
        }
        else if (dotCount > 1) {
            code = TK_UNDEF;
            addError("Invalid number format (multiple decimal points): " +
token);
        }
        else {
            ungetc(ch, g_fp);  // 回退用于检查的字符
            code = (dotCount == 1) ? TK_DOUBLE : TK_INT;  // 有小数点是浮点
数，否则是整数
        }
    }
    else {  // 处理运算符和分隔符
        token = ch;

        switch (ch) {
            case '+': code = TK_PLUS; break;
            case '-':
                // 检查是否为负数（如-10）或减号运算符
                ch = fgetc(g_fp);
                if (isdigit(ch)) {
                    // 处理负数
                    token += ch;  // 添加数字到token
                    int dotCount = 0;

                    while (true) {
                        ch = fgetc(g_fp);
```

```
                        if (!(isdigit(ch) || ch == '.')) {
                            break;
                        }

                        if (ch == '.') {
                            dotCount++;
                        }

                        token += ch;
                    }

                    // 检查数字格式是否正确
                    if (!checkNumberNext(ch)) {
                        // 非法后缀，继续读取错误标记
                        token += ch;
                        while (true) {
                            ch = fgetc(g_fp);
                            if (!(isLetter(ch) || isdigit(ch))) {
                                ungetc(ch, g_fp);
                                break;
                            }
                            token += ch;
                        }
                        code = TK_UNDEF;
                        addError("Invalid number format: " + token);
                    }
                    else if (dotCount > 1) {
                        code = TK_UNDEF;
                        addError("Invalid number format (multiple decimal
points): " + token);
                    }
                    else {
                        ungetc(ch, g_fp);  // 回退用于检查的字符
                        code = (dotCount == 1) ? TK_DOUBLE : TK_INT;  // 有
小数点是浮点数，否则是整数
                    }
                } else {
                    // 不是负数，是减号运算符
                    ungetc(ch, g_fp);
                    code = TK_MINUS;
                }
                break;
            case '*': code = TK_STAR; break;
            case '/': {
                // 检查是否为注释
                ch = fgetc(g_fp);
                if (ch == '/') {
                    // 处理单行注释，读取直到行尾
                    while ((ch = fgetc(g_fp)) != EOF && ch != '\n');
                    if (ch == '\n') {
                        g_row++; // 增加行号
                    }
                    // 递归调用以获取下一个真正的token
                    return processToken();
```

```
            } else {
                // 不是注释，回退字符
                ungetc(ch, g_fp);
                code = TK_DIVIDE;
            }
            break;
        }
        // 处理各种分隔符
        case '(': code = TK_OPENPA; break;
        case ')': code = TK_CLOSEPA; break;
        case '[': code = TK_OPENBR; break;
        case ']': code = TK_CLOSEBR; break;
        case '{': code = TK_BEGIN; break;
        case '}': code = TK_END; break;
        case ',': code = TK_COMMA; break;
        case ';': code = TK_SEMOCOLOM; break;

        case '=': {  // 处理 = 或 ==
            ch = fgetc(g_fp);
            if (ch == '=') {
                token += ch;
                code = TK_EQ;  // 等于运算符 ==
            } else {
                ungetc(ch, g_fp);
                code = TK_ASSIGN;  // 赋值运算符 =
            }
            break;
        }

        case '<': {  // 处理 < 或 <=
            ch = fgetc(g_fp);
            if (ch == '=') {
                token += ch;
                code = TK_LEQ;  // 小于等于运算符 <=
            } else {
                ungetc(ch, g_fp);
                code = TK_LT;  // 小于运算符 <
            }
            break;
        }

        case '>': {  // 处理 > 或 >=
            ch = fgetc(g_fp);
            if (ch == '=') {
                token += ch;
                code = TK_GEQ;  // 大于等于运算符 >=
            } else {
                ungetc(ch, g_fp);
                code = TK_GT;  // 大于运算符 >
            }
            break;
        }

        case '&': {  // 处理 &&
```

```
                ch = fgetc(g_fp);
                if (ch == '&') {
                    token += ch;
                    code = TK_AND;  // 逻辑与运算符 &&
                } else {
                    ungetc(ch, g_fp);
                    code = TK_BITAND;
                }
                break;
            }

            case '|': {  // 处理 ||
                ch = fgetc(g_fp);
                if (ch == '|') {
                    token += ch;
                    code = TK_OR;  // 逻辑或运算符 ||
                } else {
                    ungetc(ch, g_fp);
                    code = TK_BITOR;
                }
                break;
            }

            default:
                if (isNonAscii((unsigned char)ch)) {
                    // 跳过非ASCII字符（如中文），不报错
                    // 读取此UTF-8字符的剩余字节
                    int byteCount = 0;
                    if ((ch & 0xE0) == 0xC0) byteCount = 1;      // 2字节字
符
                    else if ((ch & 0xF0) == 0xE0) byteCount = 2; // 3字节字
符
                    else if ((ch & 0xF8) == 0xF0) byteCount = 3; // 4字节字
符

                    // 跳过剩余字节
                    for (int i = 0; i < byteCount; i++) {
                        fgetc(g_fp);
                    }

                    // 递归调用以获取下一个真正的token
                    return processToken();
                } else {
                    code = TK_UNDEF;
                    addError("Unknown symbol: " + token);
                }
                break;
        }
    }

    // 填充结果
    result.code = code;
    result.value = token;
```

```
    // 处理符号表
    if (code == TK_IDENT) {
        if (tokenCodeMap.find(code) == tokenCodeMap.end()) {
            tokenCodeMap[code] = tokenCodeMap.size() + 1;
        }
        result.table_row = tokenCodeMap[code];
    }
    else if (code == TK_INT || code == TK_DOUBLE) {
        result.type = Table_CONSTANT;
        if (constantsMap.find(token) == constantsMap.end()) {
            constantsMap[token] = constantsMap.size() + 1;
        }
        result.table_row = constantsMap[token];
    }

    return result;
}

/* 接口实现 */
// 初始化词法分析器
void initLexer(FILE* fp) {
    g_fp = fp;
    g_row = 1;
    g_hasUnget = false;
    g_errors.clear();
    tokenCodeMap.clear();
    constantsMap.clear();
}

// 获取下一个Token
// 如果有回退的Token，则直接返回；否则处理并返回新的Token
TokenAttr getNextToken() {
    if (g_hasUnget) {  // 回退 token
        g_hasUnget = false;
        return g_lastToken;
    }

    g_lastToken = processToken();
    return g_lastToken;
}

// 回退一个Token
// 标记有回退的Token，下次调用getNextToken时将返回此Token
void ungetToken() {
    g_hasUnget = true;
}

// 获取当前行号
int getCurrentLine() {
    return g_row;
}

// 获取所有词法错误信息
const std::vector<ErrorInfo>& getErrors() {
```

```
    return g_errors;
}

// 重置词法分析器
// 将文件指针重置到文件开头，重新开始词法分析
void resetLexer() {
    if (g_fp) {
        rewind(g_fp);
        g_row = 1;
        g_hasUnget = false;
    }
}

// 关闭词法分析器
void closeLexer() {
    g_fp = nullptr;
    g_hasUnget = false;
}
```

```
// parser.h
#ifndef PARSER_H
#define PARSER_H

#include "lexer.h"
#include <vector>
#include <string>

// 语法分析结果状态
enum ParserResult {
    RESULT_SUCCESS,      // 语法分析成功
    RESULT_ERROR         // 语法分析失败
};

// 语法错误信息结构体
struct ParserError {
    int line;                // 错误所在行
    std::string message;     // 错误信息
};

/* INFO 语法分析器接口 */

// 初始化语法分析器
void initParser(FILE* fp);

// 执行语法分析
ParserResult parse();

// 获取所有语法错误信息
const std::vector<ParserError>& getParserErrors();

// 重置语法分析器
```

```
void resetParser();

// 关闭语法分析器
void closeParser();

#endif /* PARSER_H */
```

```
// parser.cpp
#include "parser.h"
#include <iostream>
#include <map>
#include <string>
#include <vector>

/* 全局变量 */
static TokenAttr g_token;               // 当前分析的Token
static std::vector<ParserError> g_errors; // 语法错误列表
static bool g_hasError = false;         // 是否有语法错误

/*
 * Mini语言BNF文法定义 - 分层结构
 * ============================
 *
 * 第1层：程序结构层
 * <program> ::= <function-definition>+  // 程序由一个或多个函数定义组成
 *
 * 第2层：函数定义层
 * <function-definition> ::= <type-specifier> <identifier> '(' <parameter-
list>? ')' <compound-statement>
 * <type-specifier> ::= 'int' | 'double' | 'float'  // 函数返回类型
 * <parameter-list> ::= <parameter-declaration> | <parameter-list> ','
<parameter-declaration>
 * <parameter-declaration> ::= <type-specifier> <identifier>
 *
 * 第3层：语句层
 * <compound-statement> ::= '{' <statement-list>? '}'  // 复合语句（代码块）
 * <statement-list> ::= <statement> | <statement-list> <statement>
 * <statement> ::= <expression-statement>  // 表达式语句
 *               | <compound-statement>    // 复合语句
 *               | <selection-statement>   // 选择语句（if-else）
 *               | <iteration-statement>   // 循环语句（while）
 *               | <return-statement>      // 返回语句
 *               | <variable-declaration>  // 变量声明语句
 *
 * 第4层：具体语句定义
 * <expression-statement> ::= <expression>? ';'
 * <selection-statement> ::= 'if' '(' <expression> ')' <statement> ('else'
<statement>)?
 *                         | 'if' '(' <expression> ')' 'then' <statement>
('else' <statement>)?
 * <iteration-statement> ::= 'while' '(' <expression> ')' <statement>
```

```
 * <return-statement> ::= 'return' <expression>? ';'
 * <variable-declaration> ::= <type-specifier> <identifier> ('='
<expression>)? ';'
 *
 * 第5层：表达式层
 * <expression> ::= <assignment-expression>
 * <assignment-expression> ::= <identifier> '=' <logical-or-expression> |
<logical-or-expression>
 *
 * 第6层：逻辑表达式层
 * <logical-or-expression> ::= <logical-and-expression> | <logical-or-
expression> '||' <logical-and-expression>
 * <logical-and-expression> ::= <equality-expression> | <logical-and-
expression> '&&' <equality-expression>
 * <equality-expression> ::= <relational-expression> | <equality-
expression> '==' <relational-expression>
 *
 * 第7层：关系表达式层
 * <relational-expression> ::= <additive-expression>
 *                           | <relational-expression> '<' <additive-
expression>
 *                           | <relational-expression> '>' <additive-
expression>
 *                           | <relational-expression> '<=' <additive-
expression>
 *                           | <relational-expression> '>=' <additive-
expression>
 *
 * 第8层：算术表达式层
 * <additive-expression> ::= <multiplicative-expression>
 *                         | <additive-expression> '+' <multiplicative-
expression>
 *                         | <additive-expression> '-' <multiplicative-
expression>
 * <multiplicative-expression> ::= <primary-expression>
 *                               | <multiplicative-expression> '*'
<primary-expression>
 *                               | <multiplicative-expression> '/'
<primary-expression>
 *
 * 第9层：基本表达式层
 * <primary-expression> ::= <identifier>  // 标识符
 *                        | <constant>    // 常量
 *                        | '(' <expression> ')'  // 括号表达式
 *                        | <identifier> '(' <argument-list>? ')' // 函数调
用
 * <argument-list> ::= <expression> | <argument-list> ',' <expression>
 */

/* 前向声明所有解析函数 */
static bool program();
static bool functionDefinition();
static bool typeSpecifier();
static bool parameterList();
```

```
    static bool parameterDeclaration();
    static bool compoundStatement();
    static bool statementList();
    static bool statement();
    static bool expressionStatement();
    static bool selectionStatement();
    static bool iterationStatement();
    static bool returnStatement();
    static bool variableDeclaration();
    static bool expression();
    static bool assignmentExpression();
    static bool logicalOrExpression();
    static bool logicalAndExpression();
    static bool equalityExpression();
    static bool relationalExpression();
    static bool additiveExpression();
    static bool multiplicativeExpression();
    static bool primaryExpression();
    static bool functionCall();
    static bool argumentList();

    /* INFO 辅助函数 */

    // 添加语法错误
    static void addError(const std::string& message) {
        ParserError error = { g_token.line, message };
        g_errors.push_back(error);
        g_hasError = true;  // 设置错误标志
        std::cerr << "Syntax Error at line " << g_token.line << ": " <<
    message << std::endl;
    }

    // 增强的错误报告函数，包含当前Token的详细信息
    static void addDetailedError(const std::string& message) {
        std::string detailedMessage = message;
        if (g_token.code != TK_EOF) {
            detailedMessage += " (当前Token: '" + g_token.value + "')";
        }
        ParserError error = { g_token.line, detailedMessage };
        g_errors.push_back(error);  // 确保错误被添加到g_errors向量中
        g_hasError = true;  // 设置错误标志
        std::cerr << "Syntax Error at line " << g_token.line << ": " <<
    detailedMessage << std::endl;
    }

    // 匹配特定类型的Token
    static bool match(TokenCode code) {
        if (g_token.code == code) {
            g_token = getNextToken();  // INFO get next token
            return true;
        }
        return false;
    }
```

```
// 判断当前Token是否为目标类型
static bool isToken(TokenCode code) {
    return g_token.code == code;
}

// 记录并跳过语法错误
static void skipUntil(std::vector<TokenCode> syncSet) {
    // 记录跳过的token，用于错误报告
    std::string skippedTokens = "";
    int skipCount = 0;
    const int maxDisplayTokens = 3;  // DEBUG 最多显示几个跳过的token

    while (g_token.code != TK_EOF) {
        for (TokenCode code : syncSet) {
            if (g_token.code == code) {
                if (skipCount > 0) {
                    std::string message = "已跳过 " +
std::to_string(skipCount) + " 个token";
                    if (!skippedTokens.empty()) {
                        message += " (包括: " + skippedTokens + ")";
                    }
                    std::cerr << "Info: " << message << std::endl;
                }
                return;
            }
        }

        // 记录跳过的token
        if (skipCount < maxDisplayTokens) {
            if (!skippedTokens.empty()) {
                skippedTokens += ", ";
            }
            skippedTokens += "'" + g_token.value + "'";
        } else if (skipCount == maxDisplayTokens) {
            skippedTokens += "...";
        }
        skipCount++;

        g_token = getNextToken();
    }
}

/* INFO 递归下降分析函数实现 */

// <program> ::= <function-definition>+
static bool program() {
    bool success = true;

    // 获取第一个token
    g_token = getNextToken();

    while (g_token.code != TK_EOF) {
        // 检查是否为函数定义的开始（类型说明符）
        // std::cout << "program: " << getTokenName(g_token.code) <<
```

```
    std::endl;  // TEST
            if (g_token.code == KW_INT || g_token.code == KW_DOUBLE ||
    g_token.code == KW_FLOAT) {
                if (!functionDefinition()) {
                    success = false;
                    // 提供更详细的错误信息
                    addDetailedError("函数定义语法错误");
                    // 尝试同步到下一个函数定义
                    std::vector<TokenCode> syncSet = {KW_INT, KW_DOUBLE,
    KW_FLOAT, TK_EOF};
                    skipUntil(syncSet);
                }
            } else {
                // 遇到了非函数定义的「开始」，报错并跳过
                addDetailedError("程序中只能包含函数定义，遇到意外的标记");
                std::vector<TokenCode> syncSet = {KW_INT, KW_DOUBLE, KW_FLOAT,
    TK_EOF};
                skipUntil(syncSet);
                success = false;
            }
        }

        return success;
    }

    // 第2层：函数定义层
    // <function-definition> ::= <type-specifier> <identifier> '(' <parameter-
    list>? ')' <compound-statement>
    static bool functionDefinition() {
        if (!typeSpecifier()) {
            addDetailedError("函数定义缺少类型说明符");
            return false;
        }

        if (!isToken(TK_IDENT)) {
            addDetailedError("函数定义缺少函数名");
            return false;
        }
        match(TK_IDENT);

        if (!match(TK_OPENPA)) {
            addDetailedError("函数名后缺少左括号 '('");
            return false;
        }

        // 可选的参数列表
        if (!isToken(TK_CLOSEPA)) {
            parameterList();
        }

        if (!match(TK_CLOSEPA)) {
            addDetailedError("参数列表后缺少右括号 ')'");
            return false;
        }
```

```
    if (!compoundStatement()) {
        addDetailedError("函数定义缺少函数体");
        return false;
    }

    return true;
}

// 第2层：函数定义层
// <type-specifier> ::= 'int' | 'double' | 'float'
static bool typeSpecifier() {
    if (match(KW_INT) || match(KW_DOUBLE) || match(KW_FLOAT)) {
        return true;
    }
    return false;
}

// 第2层：函数定义层
// <parameter-list> ::= <parameter-declaration> | <parameter-list> ','
<parameter-declaration>
static bool parameterList() {
    if (!parameterDeclaration()) {
        return false;
    }

    while (match(TK_COMMA)) {
        if (!parameterDeclaration()) {
            addDetailedError("逗号后缺少有效的参数声明");
            return false;
        }
    }

    return true;
}

// 第2层：函数定义层
// <parameter-declaration> ::= <type-specifier> <identifier>
static bool parameterDeclaration() {
    if (!typeSpecifier()) {
        addDetailedError("参数声明缺少类型说明符");
        return false;
    }

    if (!isToken(TK_IDENT)) {
        addDetailedError("参数声明缺少参数名");
        return false;
    }
    match(TK_IDENT);

    return true;
}

// 函数体的大括号结构
```

```
// <compound-statement> ::= '{' <statement-list>? '}'
static bool compoundStatement() {
    if (!match(TK_BEGIN)) {
        addDetailedError("复合语句缺少左大括号 '{'");
        return false;
    }

    // 可选的语句列表
    if (!isToken(TK_END)) {
        statementList();
    }

    if (!match(TK_END)) {
        addDetailedError("复合语句缺少右大括号 '}'");
        return false;
    }

    return true;
}

// 函数体的内部语句列表
// <statement-list> ::= <statement> | <statement-list> <statement>
static bool statementList() {
    bool success = true;

    while (g_token.code != TK_END && g_token.code != TK_EOF) {
        if (!statement()) {
            success = false;

            // 提供更详细的错误信息
            std::string tokenName = "'" + g_token.value + "'";
            addDetailedError("无法解析语句，遇到意外的标记: " + tokenName);

            // 尝试同步到下一个语句
            std::vector<TokenCode> syncSet = {TK_SEMOCOLOM, TK_BEGIN,
TK_END, KW_IF, KW_WHILE, KW_RETURN, TK_EOF};
            skipUntil(syncSet);

            // 跳过分号以尝试继续解析
            if (g_token.code == TK_SEMOCOLOM) {
                match(TK_SEMOCOLOM); // 跳过分号
            }
        }
    }
    return success;
}

// INFO 不同的语句入口
// <statement> ::= <expression-statement> | <compound-statement> |
<selection-statement> | <iteration-statement> | <return-statement> |
<variable-declaration>
static bool statement() {
    switch (g_token.code) {
        case TK_BEGIN:
```

```
            return compoundStatement();
        case KW_IF:
            return selectionStatement();
        case KW_WHILE:
            return iterationStatement();
        case KW_RETURN:
            return returnStatement();
        case KW_INT:
        case KW_DOUBLE:
        case KW_FLOAT:
            return variableDeclaration();
        default:
            return expressionStatement();
    }
}

// 表达式语句
// <expression-statement> ::= <expression>? ';'
static bool expressionStatement() {
    if (g_token.code != TK_SEMOCOLOM) {
        if (!expression()) {
            return false;
        }
    }

    if (!match(TK_SEMOCOLOM)) {
        addDetailedError("表达式语句缺少分号 ';'");
        return false;
    }

    return true;
}

// 选择语句
// <selection-statement> ::= 'if' '(' <expression> ')' <statement> ('else'
<statement>)?
//                         | 'if' '(' <expression> ')' 'then' <statement>
('else' <statement>)?
static bool selectionStatement() {
    if (!match(KW_IF)) {
        addDetailedError("预期关键字 'if'");
        return false;
    }

    if (!match(TK_OPENPA)) {
        addDetailedError("if语句缺少左括号 '('");
        return false;
    }

    if (!expression()) {
        addDetailedError("if条件表达式无效或缺失");
        return false;
    }
```

```
    if (!match(TK_CLOSEPA)) {
        addDetailedError("if条件缺少右括号 ')'");
        return false;
    }

    // 检查是否有 then 关键字
    bool hasThen = match(KW_THEN);

    if (!statement()) {
        if (hasThen) {
            addDetailedError("'then'后应有语句块");
        } else {
            addDetailedError("if语句体缺失或无效");
        }
        return false;
    }

    // 可选的else部分
    if (match(KW_ELSE)) {
        if (!statement()) {
            addDetailedError("else语句体缺失或无效");
            return false;
        }
    }

    return true;
}

// 循环语句
// <iteration-statement> ::= 'while' '(' <expression> ')' <statement>
static bool iterationStatement() {
    if (!match(KW_WHILE)) {
        addDetailedError("预期关键字 'while'");
        return false;
    }

    if (!match(TK_OPENPA)) {
        addDetailedError("while语句缺少左括号 '('");
        return false;
    }

    if (!expression()) {
        addDetailedError("while条件表达式无效或缺失");
        return false;
    }

    if (!match(TK_CLOSEPA)) {
        addDetailedError("while条件缺少右括号 ')'");
        return false;
    }

    if (!statement()) {
        addDetailedError("while循环体缺失或无效");
        return false;
```

```
    }

    return true;
}

// 返回语句
// <return-statement> ::= 'return' <expression>? ';'
static bool returnStatement() {
    if (!match(KW_RETURN)) {
        addDetailedError("预期关键字 'return'");
        return false;
    }

    // 可选的表达式
    if (g_token.code != TK_SEMOCOLOM) {
        if (!expression()) {
            return false;
        }
    }

    if (!match(TK_SEMOCOLOM)) {
        addDetailedError("return语句缺少分号 ';'");
        return false;
    }

    return true;
}

// 表达式
// <expression> ::= <assignment-expression>
static bool expression() {
    return assignmentExpression();
}

// 赋值表达式
// <assignment-expression> ::= <identifier> '=' <logical-or-expression> |
<logical-or-expression>
static bool assignmentExpression() {
    if (g_token.code == TK_IDENT) {
        TokenAttr savedToken = g_token;
        match(TK_IDENT);

        if (match(TK_ASSIGN)) {
            if (!logicalOrExpression()) {
                addDetailedError("赋值运算符 '=' 后缺少有效的表达式");
                return false;
            }
            return true;
        } else {
            // 不是赋值表达式，回退Token
            ungetToken();  // INFO unget token
            g_token = savedToken;
        }
    }
```

```
    return logicalOrExpression();
}

// 逻辑或表达式
// <logical-or-expression> ::= <logical-and-expression> | <logical-or-
expression> '||' <logical-and-expression>
static bool logicalOrExpression() {
    if (!logicalAndExpression()) {
        return false;
    }

    while (g_token.code == TK_OR) {
        match(TK_OR);
        if (!logicalAndExpression()) {
            addDetailedError("'||'运算符后缺少有效的表达式");
            return false;
        }
    }

    return true;
}

// 逻辑与表达式
// <logical-and-expression> ::= <equality-expression> | <logical-and-
expression> '&&' <equality-expression>
static bool logicalAndExpression() {
    if (!equalityExpression()) {
        return false;
    }

    while (g_token.code == TK_AND) {
        match(TK_AND);
        if (!equalityExpression()) {
            addDetailedError("'&&'运算符后缺少有效的表达式");
            return false;
        }
    }

    return true;
}

// 相等表达式
// <equality-expression> ::= <relational-expression> | <equality-
expression> '==' <relational-expression>
static bool equalityExpression() {
    if (!relationalExpression()) {
        return false;
    }

    while (g_token.code == TK_EQ) {
        match(TK_EQ);
        if (!relationalExpression()) {
            addDetailedError("'=='运算符后缺少有效的表达式");
```

```
            return false;
        }
    }

    return true;
}

// 关系表达式
// <relational-expression> ::= <additive-expression>
//                          | <relational-expression> '<' <additive-
expression>
//                          | <relational-expression> '>' <additive-
expression>
//                          | <relational-expression> '<=' <additive-
expression>
//                          | <relational-expression> '>=' <additive-
expression>
static bool relationalExpression() {
    if (!additiveExpression()) {
        return false;
    }

    while (g_token.code == TK_LT || g_token.code == TK_GT ||
           g_token.code == TK_LEQ || g_token.code == TK_GEQ) {
        match(g_token.code);
        if (!additiveExpression()) {
            addDetailedError("关系运算符后缺少有效的表达式");
            return false;
        }
    }

    return true;
}

// 加减表达式
// <additive-expression> ::= <multiplicative-expression>
//                         | <additive-expression> '+' <multiplicative-
expression>
//                         | <additive-expression> '-' <multiplicative-
expression>
//                         | <additive-expression> '|' <multiplicative-
expression>
//                         | <additive-expression> '&' <multiplicative-
expression>
static bool additiveExpression() {
    if (!multiplicativeExpression()) {
        return false;
    }

    while (g_token.code == TK_PLUS || g_token.code == TK_MINUS ||
g_token.code == TK_BITOR || g_token.code == TK_BITAND) {
        match(g_token.code);
        if (!multiplicativeExpression()) {
            addDetailedError("'+' 或 '-' 运算符后缺少有效的表达式");
```

```
            return false;
        }
    }

    return true;
}

// 乘除表达式
// <multiplicative-expression> ::= <primary-expression>
//                              | <multiplicative-expression> '*'
<primary-expression>
//                              | <multiplicative-expression> '/'
<primary-expression>
static bool multiplicativeExpression() {
    if (!primaryExpression()) {
        return false;
    }

    while (g_token.code == TK_STAR || g_token.code == TK_DIVIDE) {
        match(g_token.code);
        if (!primaryExpression()) {
            addDetailedError("'*' 或 '/' 运算符后缺少有效的表达式");
            return false;
        }
    }

    return true;
}

// 基本表达式
// <primary-expression> ::= <identifier>
//                       | <constant>
//                       | '(' <expression> ')'
//                       | <identifier> '(' <argument-list>? ')' // 函数调用
static bool primaryExpression() {
    if (g_token.code == TK_IDENT) {
        TokenAttr savedToken = g_token;
        match(TK_IDENT);

        // 检查是否为函数调用
        if (g_token.code == TK_OPENPA) {
            // 回退Token，以便在functionCall中正确识别函数名
            ungetToken();  // INFO unget token
            g_token = savedToken;
            return functionCall();
        }

        return true;
    } else if (g_token.code == TK_INT || g_token.code == TK_DOUBLE) {
        match(g_token.code);
        return true;
    } else if (match(TK_OPENPA)) {
        if (!expression()) {
            addDetailedError("括号内缺少有效的表达式");
```

```
            return false;
        }

        if (!match(TK_CLOSEPA)) {
            addDetailedError("表达式后缺少右括号 ')'");
            return false;
        }

        return true;
    } else {
        addDetailedError("预期标识符、常量或左括号 '('");
        return false;
    }
}

// 变量声明
// <variable-declaration> ::= <type-specifier> <identifier> ('='
<expression>)? ';'
static bool variableDeclaration() {
    if (!typeSpecifier()) {
        addDetailedError("变量声明缺少类型说明符");
        return false;
    }

    if (!isToken(TK_IDENT)) {
        addDetailedError("变量声明缺少变量名");
        return false;
    }
    match(TK_IDENT);

    // 可选的赋值表达式
    if (match(TK_ASSIGN)) {
        if (!logicalOrExpression()) {  // 使用logicalOrExpression而不是
expression避免递归问题
            addDetailedError("赋值运算符 '=' 后缺少有效的表达式");
            return false;
        }
    }

    if (!match(TK_SEMOCOLOM)) {
        addDetailedError("变量声明缺少分号 ';'");
        return false;
    }

    return true;
}

// 函数调用
// <function-call> ::= <identifier> '(' <argument-list>? ')'
static bool functionCall() {
    if (!isToken(TK_IDENT)) {
        addDetailedError("函数调用缺少函数名");
        return false;
    }
```

```
    match(TK_IDENT);

    if (!match(TK_OPENPA)) {
        addDetailedError("函数名后缺少左括号 '('");
        return false;
    }

    // 可选的参数列表
    if (!isToken(TK_CLOSEPA)) {
        if (!argumentList()) {
            return false;
        }
    }

    if (!match(TK_CLOSEPA)) {
        addDetailedError("函数调用缺少右括号 ')'");
        return false;
    }

    return true;
}

// 参数列表
// <argument-list> ::= <expression> | <argument-list> ',' <expression>
static bool argumentList() {
    if (!expression()) {
        addDetailedError("函数调用参数无效");
        return false;
    }

    while (match(TK_COMMA)) {
        if (!expression()) {
            addDetailedError("逗号后缺少有效的参数表达式");
            return false;
        }
    }

    return true;
}

/* INFO 接口实现 */

void initParser(FILE* fp) {
    initLexer(fp);
    g_errors.clear();
    g_hasError = false;  // 初始化错误标志
    // 不要在这里预先获取第一个token
}

ParserResult parse() {
    bool success = program();
    // 如果有语法错误，返回错误结果
    return (success && !g_hasError) ? RESULT_SUCCESS : RESULT_ERROR;
}
```

```
const std::vector<ParserError>& getParserErrors() {
    return g_errors;
}

void resetParser() {
    resetLexer();
    g_errors.clear();
    g_hasError = false;  // 重置错误标志
    g_token = getNextToken();
}

void closeParser() {
    closeLexer();
}
```

```
// main.cpp
#include "lexer.h"
#include "parser.h"
#include <iostream>
#include <string>
#include <fstream>
#include <sys/stat.h>
#include <sys/types.h>
#include <vector>
#include <map>
#include <algorithm>

// 符号表和常量表
std::map<TokenCode, int> tokenCodeMap;
std::map<std::string, int> constantsMap;
std::vector<TokenAttr> tokenList;

// 函数声明
void showUsage(const char* programName);

// 输出TokenCode对应的字符串描述
std::string getTokenName(TokenCode code) {
    switch (code) {
        case TK_UNDEF: return "UNDEFINED";
        case KW_INT: return "KEYWORD_INT";
        case KW_DOUBLE: return "KEYWORD_DOUBLE";
        case KW_FLOAT: return "KEYWORD_FLOAT";
        case KW_IF: return "KEYWORD_IF";
        case KW_THEN: return "KEYWORD_THEN";
        case KW_ELSE: return "KEYWORD_ELSE";
        case KW_RETURN: return "KEYWORD_RETURN";
        case KW_WHILE: return "KEYWORD_WHILE";
        case TK_PLUS: return "OPERATOR_PLUS";
        case TK_MINUS: return "OPERATOR_MINUS";
        case TK_STAR: return "OPERATOR_MULTIPLY";
```

```
        case TK_DIVIDE: return "OPERATOR_DIVIDE";
        case TK_ASSIGN: return "OPERATOR_ASSIGN";
        case TK_EQ: return "OPERATOR_EQUAL";
        case TK_BITOR: return "OPERATOR_BITOR";
        case TK_BITAND: return "OPERATOR_BITAND";
        case TK_AND: return "OPERATOR_AND";
        case TK_OR: return "OPERATOR_OR";
        case TK_LT: return "OPERATOR_LESS_THAN";
        case TK_LEQ: return "OPERATOR_LESS_EQUAL";
        case TK_GT: return "OPERATOR_GREATER_THAN";
        case TK_GEQ: return "OPERATOR_GREATER_EQUAL";
        case TK_OPENPA: return "DELIMITER_OPEN_PARENTHESIS";
        case TK_CLOSEPA: return "DELIMITER_CLOSE_PARENTHESIS";
        case TK_OPENBR: return "DELIMITER_OPEN_BRACKET";
        case TK_CLOSEBR: return "DELIMITER_CLOSE_BRACKET";
        case TK_BEGIN: return "DELIMITER_BEGIN_BRACE";
        case TK_END: return "DELIMITER_END_BRACE";
        case TK_COMMA: return "DELIMITER_COMMA";
        case TK_SEMOCOLOM: return "DELIMITER_SEMICOLON";
        case TK_INT: return "CONSTANT_INTEGER";
        case TK_DOUBLE: return "CONSTANT_DOUBLE";
        case TK_IDENT: return "IDENTIFIER";
        case TK_EOF: return "END_OF_FILE";
        default: return "UNKNOWN";
    }
}

// 输出结果到文件
void outputResults(const std::string& filename, bool lexOnly, bool
parseSuccess = true, const std::vector<ParserError>& savedParseErrors =
std::vector<ParserError>()) {
    // 创建输出目录
    std::string dirName = filename + "-output";
    struct stat info;

    if (stat(dirName.c_str(), &info) != 0) { // 检查目录是否存在
        if (mkdir(dirName.c_str(), 0777) == -1) {
            std::cerr << "错误: 无法创建目录 " << dirName << std::endl;
            return;
        }
    } else if (!(info.st_mode & S_IFDIR)) { // 如果存在但不是目录
        std::cerr << "错误: " << dirName << " 已存在但不是目录" << std::endl;
        return;
    }

    // 输出Token列表
    std::ofstream tokenFile(dirName + "/tokens.txt");
    if (tokenFile.is_open()) {
        tokenFile << "行号\t类型\t\t值\n";
        tokenFile << "-----------------------------------\n";

        for (const auto& token : tokenList) {
            tokenFile << token.line << "\t"
                      << getTokenName(token.code) << "\t"
```

```
                        << token.value << "\n";
        }
        tokenFile.close();
    }

    // 输出词法错误信息
    const std::vector<ErrorInfo>& lexErrors = getErrors();
    if (!lexErrors.empty()) {
        std::ofstream errorFile(dirName + "/lex_errors.txt");
        if (errorFile.is_open()) {
            errorFile << "行号\t错误信息\n";
            errorFile << "------------------------------------\n";

            for (const auto& error : lexErrors) {
                errorFile << error.line << "\t" << error.message << "\n";
            }
            errorFile.close();
        }
    }

    // 输出语法错误信息
    if (!lexOnly) {
        const std::vector<ParserError>& parseErrors =
savedParseErrors.empty() ? getParserErrors() : savedParseErrors;
        // 始终创建语法错误文件，即使没有错误
        std::ofstream parseErrorFile(dirName + "/parse_errors.txt");
        if (parseErrorFile.is_open()) {
            parseErrorFile << "行号\t错误信息\n";
            parseErrorFile << "------------------------------------\n";

            for (const auto& error : parseErrors) {
                parseErrorFile << error.line << "\t" << error.message <<
"\n";
            }

            if (parseErrors.empty()) {
                parseErrorFile << "无语法错误\n";
            }

            parseErrorFile.close();
        }
    }

    // 简洁的摘要输出
    std::cout << "\n=== 分析完成 ===\n";
    std::cout << "文件: " << filename << "\n";

    if (lexOnly) {
        std::cout << "词法分析结果: " << (lexErrors.empty() ? "成功" : "有错
误") << "\n";
        std::cout << "Token总数: " << tokenList.size() << "\n";
        std::cout << "词法错误总数: " << lexErrors.size() << "\n";
    } else {
        const std::vector<ParserError>& parseErrors =
```

```
    savedParseErrors.empty() ? getParserErrors() : savedParseErrors;
            std::cout << "词法分析结果: " << (lexErrors.empty() ? "成功" : "有错
误") << "\n";
            std::cout << "语法分析结果: " << (parseSuccess ? "成功" : "有错误") <<
"\n";
            std::cout << "Token总数: " << tokenList.size() << "\n";
            std::cout << "词法错误总数: " << lexErrors.size() << "\n";
            std::cout << "语法错误总数: " << parseErrors.size() << "\n";
        }

        std::cout << "结果已输出到: " << dirName << "\n";
    }

    int main(int argc, char* argv[]) {
        // 输入文件路径
        std::string filename;
        FILE* fp;
        bool showProcess = true;   // 是否显示分析过程
        bool lexOnly = false;      // 是否仅进行词法分析
        bool parseSuccess = true;  // 语法分析是否成功
        std::vector<ParserError> parseErrors; // 保存语法错误

        // 检查命令行参数
        if (argc < 2) {
            showUsage(argv[0]);
            return 1;
        }

        // 解析命令行选项
        for (int i = 1; i < argc; i++) {
            std::string arg = argv[i];

            if (arg == "-h" || arg == "--help") {
                showUsage(argv[0]);
                return 0;
            } else if (arg == "-v" || arg == "--version") {
                std::cout << "Mini语言编译器 v1.0\n";
                return 0;
            } else if (arg == "-q" || arg == "--quiet") {
                showProcess = false;
            } else if (arg == "-l" || arg == "--lex-only") {
                lexOnly = true;
            } else if (arg[0] == '-') {
                std::cerr << "错误: 未知选项 " << arg << "\n";
                showUsage(argv[0]);
                return 1;
            } else {
                filename = arg;
            }
        }

        if (filename.empty()) {
            std::cerr << "错误: 未指定输入文件\n";
            showUsage(argv[0]);
```

```
        return 1;
    }

    // 尝试打开文件
    fp = fopen(filename.c_str(), "r");
    if (fp == nullptr) {
        std::cerr << "错误: 无法打开文件 " << filename << std::endl;
        return 1;
    }

    // INFO 仅进行词法分析
    if (lexOnly) {
        // 初始化词法分析器
        initLexer(fp);

        // 进行词法分析
        TokenAttr token;
        if (showProcess) {
            std::cout << "开始词法分析...\n";
        }

        do {
            token = getNextToken();
            tokenList.push_back(token);

            // 显示分析过程（可选）
            if (showProcess && token.code != TK_EOF) {
                std::cout << "行 " << token.line << ": ["
                          << getTokenName(token.code) << "] "
                          << token.value << std::endl;
            }
        } while (token.code != TK_EOF);

        // 关闭文件
        fclose(fp);

        // 输出分析结果
        outputResults(filename, true);
    } else { // 进行词法和语法分析
        // 初始化解析器
        initParser(fp);

        if (showProcess) {
            std::cout << "开始分析...\n";
        }

        // 首先收集所有token用于输出
        {
            // 保存当前文件位置
            long filePos = ftell(fp);
            // 重置文件指针到开头
            rewind(fp);
            // 重新初始化词法分析器
            initLexer(fp);
```

```
            TokenAttr token;
            do {
                token = getNextToken();
                tokenList.push_back(token);
            } while (token.code != TK_EOF);

            // 恢复文件位置
            fseek(fp, filePos, SEEK_SET);
            // 重新初始化解析器
            initParser(fp);
        }

        // 执行语法分析
        ParserResult result = parse();
        parseSuccess = (result == RESULT_SUCCESS);

        // 保存语法错误信息
        parseErrors = getParserErrors();

        // 关闭文件
        fclose(fp);

        // 输出分析结果
        if (showProcess) {
            if (parseSuccess) {
                std::cout << "语法分析成功! \n";
            } else {
                std::cout << "语法分析失败。\n";
            }
        }

        outputResults(filename, false, parseSuccess, parseErrors);
    }

    return 0;
}

// 显示使用说明
void showUsage(const char* programName) {
    std::cout << "用法: " << programName << " [选项] <文件路径>\n\n";
    std::cout << "选项:\n";
    std::cout << "  -h, --help      显示此帮助信息\n";
    std::cout << "  -v, --version   显示版本信息\n";
    std::cout << "  -q, --quiet     安静模式，不显示分析过程\n";
    std::cout << "  -l, --lex-only  仅进行词法分析，不进行语法分析\n\n";
    std::cout << "示例: " << programName << " ./example.txt\n";
    std::cout << "      " << programName << " -q ./example.txt\n";
    std::cout << "      " << programName << " -l ./example.txt\n";
}
```