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