编译原理-实验报告二

实验名称	Lab2 语法分析器 (Syntax Parser)
姓名	熊宇祺
学号	09023112
报告日期	2025年12月24日

---

a) 实验动机 / 目的

1. **理解语法分析原理**：深入理解编译器前端的核心组件——语法分析器的工作机制，掌握如何将线性的 Token 流转换为具有层级结构的语法树（Syntax Tree）。
2. **掌握 LL(1) 方法**：学习并实践自顶向下（Top-Down）的分析方法，具体通过**递归下降分析法（Recursive Descent Parsing）**来实现。
3. **文法转换技巧**：掌握上下文无关文法（CFG）的预处理技术，包括消除左递归（Left Recursion Elimination）和提取左因子（Left Factoring），以满足 LL(1) 文法的要求。
4. **程序实现能力**：编写 C 语言程序，能够对包含赋值、循环、条件判断及算术表达式的源代码进行正确的推导。

b) 内容描述

本实验实现了一个针对类 C 语言子集的语法分析器。

- **输入**：一串字符流（源代码），例如x = 0.5 * (y + 10 + z) * 3; while(x) { x = x + 1; }。
- **处理**：**词法分析**：识别输入中的标识符（ID）、数字（Num）、关键字（if/while/else）及运算符。**语法分析**：根据预定义的 LL(1) 文法，分析 Token 序列的结构。
- **输出**：语法推导序列（Sequence of Derivations）。该序列展示了如何从文法的开始符号（Start Symbol）一步步推导出输入的 Token 串，隐式地构建了语法树。

c) 设计思路 / 方法

1. 核心方法

采用 **递归下降分析法 (Recursive Descent Parsing)**，**LL（1）**。

- 为文法中的每一个**非终结符**（Non-terminal）编写一个递归函数。
- 函数的逻辑基于 LL(1) 预测分析表：根据当前的输入符号（Lookahead Token）决定调用哪个产生式。

2. 文法设计与预处理

原始的数学表达式文法通常包含左递归

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id | num
```

，这会导致递归下降陷入死循环。同时 if-else 结构存在回溯问题。设计思路如下：

- **消除左递归**：原式：`E→E+T∣TE→E+T∣T` 变换后：`E→TE′E→TE′`，`E′→+TE′∣ϵE′→+TE′∣ϵ`
- **提取左因子**（处理 if 和 if-else）：原式：`S→if (E)S∣if (E)S else SS→if (E)S∣if (E)S else S` 变换后：引入新非终结符 ElsePart，即 `S→if (E)S ElsePartS→if (E)S ElsePart`，其中 `ElsePart→else S∣ϵElsePart→else S∣ϵ`。

d) 假设条件

1. **输入格式**：假设输入是 ASCII 编码的字符流。
2. **词法限制**：标识符由字母或下划线开头，数字支持整数和小数，不支持科学计数法。
3. **语法范围**：仅支持基本的整型/浮点型算术运算，不涉及复杂的类型检查或作用域管理。
4. **错误处理**：采用“恐慌模式”的简化版，遇到第一个语法错误即报错并终止程序，不进行复杂的错误恢复。

e) 相关有限自动机 (FA) 与文法描述

虽然语法分析核心是下推自动机（PDA），但词法部分依赖有限自动机（FA），语法部分依赖 CFG。

1. 最终采用的 LL(1) 文法 (CFG)

```
1.  Program     -> Stmts
2.  Stmts       -> Stmt Stmts | epsilon
3.  Stmt        -> Block | AssignStmt | IfStmt | WhileStmt
4.  Block       -> { Stmts }
5.  AssignStmt  -> id = E ;
6.  IfStmt      -> if ( E ) Stmt ElsePart
7.  ElsePart    -> else Stmt | epsilon
8.  WhileStmt   -> while ( E ) Stmt
9.  E           -> T E'
10. E'          -> + T E' | epsilon
11. T           -> F T'
12. T'          -> * F T' | epsilon
13. F           -> ( E ) | id | num
```

2. LL(1) 分析表构造依据 (First/Follow 集)

这是编写 switch-case 逻辑的基础：

- **First(Stmt)** = { {, id, if, while }
- **First(F)** = { (, id, num }
- **Follow(E')** = {), ; } (决定何时选用 `E′→ϵE′→ϵ`)
- **Follow(ElsePart)** = Follow(IfStmt) (决定何时结束 if 语句)

f) 重要数据结构说明

1. **Token 枚举类型 (TokenType)**:
   用于区分不同的终结符，如 TOK_ID, TOK_NUM, TOK_IF, TOK_PLUS 等。codeC `typedef enum { TOK_ID, TOK_NUM, TOK_ASSIGN, ... } TokenType;`
2. **Token 结构体 (struct Token)**:
   存储词法分析的结果，包含类型和实际字符串值（用于打印）。codeC `typedef struct {    TokenType type;    char str[100]; } Token;`
3. **输入缓冲区 (inputBuffer)**:
   字符数组，用于存储从文件或硬编码读取的源代码字符串。
4. **当前 Token (currentToken)**:
   全局变量，充当 Lookahead 符号，所有递归函数通过检查它来决定下一步操作。

g) 核心算法说明

1. 词法获取 (getNextToken)

一个简单的状态机实现。

- 跳过空白符。
- 如果字符是字母 `→→` 读取直到非字母数字 `→→` 查表判断是 ID 还是 Keyword。
- 如果字符是数字 `→→` 读取直到非数字（处理小数点）。
- 如果是符号 `→→` 返回对应 Token 类型。

2. 递归下降分析

每个非终结符对应一个函数。以 parseStmts 为例：

- **算法逻辑**：检查 currentToken 是否属于 First(Stmt) (即 {, id, if, while)。如果是，打印推导规则 Stmts -> Stmt Stmts，并依次调用 parseStmt() 和 parseStmts()。如果否，检查是否属于 Follow(Stmts) (即 } 或 EOF)。如果是，则打印 Stmts -> epsilon 并返回。否则，报错。

3. 匹配终结符 (match)

- 验证当前 Token 是否等于预期的终结符。
- **成功**：调用 getNextToken() 前进到下一个 Token。
- **失败**：调用 error() 打印错误信息并退出。

h）实验代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdarg.h> // 用于可变参数的日志输出

/* ========================================
   1. 定义与全局变量
```

```
   ========================================= */

// 定义所有可能的 Token 类型
typedef enum {
    TOK_ID,          // 标识符 (x, y, count)
    TOK_NUM,         // 数字 (10, 0.5)
    TOK_ASSIGN,      // =
    TOK_SEMI,        // ;
    TOK_LPAREN,      // (
    TOK_RPAREN,      // )
    TOK_LBRACE,      // {
    TOK_RBRACE,      // }
    TOK_PLUS,        // +
    TOK_MULT,        // *
    TOK_IF,          // if 关键字
    TOK_ELSE,        // else 关键字
    TOK_WHILE,       // while 关键字
    TOK_EOF,         // 输入结束
    TOK_ERROR        // 词法错误
} TokenType;

// Token 结构体
typedef struct {
    TokenType type;
    char str[100]; // 存储 Token 的原始字符串值
} Token;

// 全局变量
char inputBuffer[2048]; // 输入字符流缓冲区
int pos = 0;            // 当前字符流读取位置
Token currentToken;     // 当前正在分析的 Token
FILE *fileOut = NULL;   // 输出文件指针

/* =========================================
   2. 函数声明
   ========================================= */

// 工具函数
void logOutput(const char *format, ...); // 同时输出到屏幕和文件
void getNextToken();                     // 词法分析器 (Scanner)
void match(TokenType expected);          // 匹配终结符
void error(const char *msg);             // 报错处理

// 语法分析函数 (对应每个非终结符)
void parseProgram();    // 起始符号
void parseStmt();       // 语句 (Statement)
void parseBlock();      // 代码块 (Block)
void parseStmts();      // 语句列表
void parseAssignStmt(); // 赋值语句
void parseIfStmt();     // 条件语句
void parseElsePart();   // else 部分 (提取左因子后)
void parseWhileStmt();  // 循环语句
```

```
// 表达式分析 (消除左递归后的文法)
void parseE();  // E -> T E'
void parseE_(); // E' -> + T E' | epsilon
void parseT();  // T -> F T'
void parseT_(); // T' -> * F T' | epsilon
void parseF();  // F -> ( E ) | id | num

/* ========================================
   3. 主函数 (入口)
   ======================================== */
int main() {
    // 1. 打开输出文件
    fileOut = fopen("output.txt", "w");
    if (fileOut == NULL) {
        printf("Error: Could not create output.txt\n");
        return 1;
    }

    // 2. 准备输入数据 (模拟字符流)
    // 你可以在这里修改测试用例
    // 测试用例涵盖: 赋值, 算术运算, 括号, 循环结构
    strcpy(inputBuffer, "x = 0.5 * (y + 10 + z) * 3; while(x) { x = x + 1; }");

    logOutput("Lab2 LL(1) Syntax Parser\n");
    logOutput("----------------------------------------\n");
    logOutput("Input Stream: %s\n", inputBuffer);
    logOutput("----------------------------------------\n");
    logOutput("Sequence of Derivations (Syntax Tree Construction):\n\n");

    // 3. 初始化并启动分析
    pos = 0;
    getNextToken(); // 预读第一个 Token
    parseProgram(); // 进入递归下降分析

    // 4. 结束检查
    if (currentToken.type == TOK_EOF) {
        logOutput("\n----------------------------------------\n");
        logOutput("Result: SUCCESS. Syntax tree built successfully.\n");
        logOutput("Output saved to 'output.txt'.\n");
    } else {
        error("Unexpected characters at the end of input.");
    }

    fclose(fileOut);
    return 0;
}

/* ========================================
   4. 工具函数实现
   ======================================== */

// 双路输出日志函数 (屏幕 + 文件)
void logOutput(const char *format, ...) {
```

```
    va_list args;

    // 输出到控制台
    va_start(args, format);
    vprintf(format, args);
    va_end(args);

    // 输出到文件
    if (fileOut != NULL) {
        va_start(args, format);
        vfprintf(fileOut, format, args);
        va_end(args);
    }
}

// 报错函数
void error(const char *msg) {
    logOutput("\n[Syntax Error] %s (Current Token: '%s')\n", msg, currentToken.str);
    if (fileOut) fclose(fileOut);
    exit(1);
}

// 终结符匹配函数
void match(TokenType expected) {
    if (currentToken.type == expected) {
        getNextToken(); // 匹配成功，读取下一个 Token
    } else {
        char errBuf[100];
        sprintf(errBuf, "Expected token type %d but found '%s'", expected,
currentToken.str);
        error(errBuf);
    }
}

/* ========================================
   5. 词法分析器 (Lexer)
   ======================================== */
void getNextToken() {
    // 1. 跳过空白字符 (空格, Tab, 换行)
    while (inputBuffer[pos] == ' ' || inputBuffer[pos] == '\t' ||
           inputBuffer[pos] == '\n' || inputBuffer[pos] == '\r') {
        pos++;
    }

    // 2. 判断字符串结束
    if (inputBuffer[pos] == '\0') {
        currentToken.type = TOK_EOF;
        strcpy(currentToken.str, "EOF");
        return;
    }

    char c = inputBuffer[pos];
```

```
    // 3. 识别标识符 (Identifier) 或 关键字 (Keywords)
    if (isalpha(c) || c == '_') {
        int i = 0;
        while (isalnum(inputBuffer[pos]) || inputBuffer[pos] == '_') {
            currentToken.str[i++] = inputBuffer[pos++];
        }
        currentToken.str[i] = '\0';

        // 检查是否是保留字
        if (strcmp(currentToken.str, "if") == 0) currentToken.type = TOK_IF;
        else if (strcmp(currentToken.str, "else") == 0) currentToken.type = TOK_ELSE;
        else if (strcmp(currentToken.str, "while") == 0) currentToken.type = TOK_WHILE;
        else currentToken.type = TOK_ID;
        return;
    }

    // 4. 识别数字 (Numbers, 支持小数)
    if (isdigit(c)) {
        int i = 0;
        while (isdigit(inputBuffer[pos]) || inputBuffer[pos] == '.') {
            currentToken.str[i++] = inputBuffer[pos++];
        }
        currentToken.str[i] = '\0';
        currentToken.type = TOK_NUM;
        return;
    }

    // 5. 识别运算符和界符
    // 每次处理完后 pos++ 移动指针
    currentToken.str[0] = c;
    currentToken.str[1] = '\0';
    pos++;

    switch(c) {
        case '=': currentToken.type = TOK_ASSIGN; break;
        case ';': currentToken.type = TOK_SEMI; break;
        case '+': currentToken.type = TOK_PLUS; break;
        case '*': currentToken.type = TOK_MULT; break;
        case '(': currentToken.type = TOK_LPAREN; break;
        case ')': currentToken.type = TOK_RPAREN; break;
        case '{': currentToken.type = TOK_LBRACE; break;
        case '}': currentToken.type = TOK_RBRACE; break;
        default:  currentToken.type = TOK_ERROR; break;
    }
}

/* ========================================
   6. 语法分析器 (LL(1) Logic)
   ======================================== */

// Grammar: Program -> Stmts
void parseProgram() {
    logOutput("Program -> Stmts\n");
```

```
    parseStmts(); // 解析语句列表
}

// Grammar: Stmts -> Stmt Stmts | epsilon
// 解释: 如果当前 token 属于 First(Stmt)，则解析 Stmt；否则如果是 '}' 或 EOF，则推导为空。
void parseStmts() {
    // First(Stmt) = { id, if, while, { }
    if (currentToken.type == TOK_ID || currentToken.type == TOK_IF ||
        currentToken.type == TOK_WHILE || currentToken.type == TOK_LBRACE) {
        logOutput("Stmts -> Stmt Stmts\n");
        parseStmt();
        parseStmts();
    } else {
        // Follow(Stmts) = { '}', EOF }
        logOutput("Stmts -> epsilon\n");
    }
}

// Grammar: Stmt -> Block | AssignStmt | IfStmt | WhileStmt
void parseStmt() {
    if (currentToken.type == TOK_LBRACE) {
        logOutput("Stmt -> Block\n");
        parseBlock();
    }
    else if (currentToken.type == TOK_ID) {
        logOutput("Stmt -> AssignStmt\n");
        parseAssignStmt();
    }
    else if (currentToken.type == TOK_IF) {
        logOutput("Stmt -> IfStmt\n");
        parseIfStmt();
    }
    else if (currentToken.type == TOK_WHILE) {
        logOutput("Stmt -> WhileStmt\n");
        parseWhileStmt();
    }
    else {
        error("Expected start of a statement (id, if, while, {)");
    }
}

// Grammar: Block -> { Stmts }
void parseBlock() {
    logOutput("Block -> { Stmts }\n");
    match(TOK_LBRACE);
    parseStmts();
    match(TOK_RBRACE);
}

// Grammar: AssignStmt -> id = E ;
void parseAssignStmt() {
    logOutput("AssignStmt -> id = E ;\n");
    match(TOK_ID);      // 匹配 id
```

```
    match(TOK_ASSIGN); // 匹配 =
    parseE();          // 解析表达式
    match(TOK_SEMI);   // 匹配 ;
}

// Grammar: WhileStmt -> while ( E ) Stmt
void parseWhileStmt() {
    logOutput("WhileStmt -> while ( E ) Stmt\n");
    match(TOK_WHILE);
    match(TOK_LPAREN);
    parseE();
    match(TOK_RPAREN);
    parseStmt(); // 循环体
}

// Grammar: IfStmt -> if ( E ) Stmt ElsePart
// 注意: 提取左因子，处理 else
void parseIfStmt() {
    logOutput("IfStmt -> if ( E ) Stmt ElsePart\n");
    match(TOK_IF);
    match(TOK_LPAREN);
    parseE();
    match(TOK_RPAREN);
    parseStmt();     // if 的主体
    parseElsePart(); // 处理可能的 else
}

// Grammar: ElsePart -> else Stmt | epsilon
void parseElsePart() {
    if (currentToken.type == TOK_ELSE) {
        logOutput("ElsePart -> else Stmt\n");
        match(TOK_ELSE);
        parseStmt();
    } else {
        // 如果不是 else，推导为空 (epsilon)
        logOutput("ElsePart -> epsilon\n");
    }
}

/* --- 算术表达式处理 (消除左递归) --- */

// Grammar: E -> T E'
void parseE() {
    logOutput("E -> T E'\n");
    parseT();
    parseE_();
}

// Grammar: E' -> + T E' | epsilon
void parseE_() {
    if (currentToken.type == TOK_PLUS) {
        logOutput("E' -> + T E'\n");
        match(TOK_PLUS);
```

```
        parseT();
        parseE_();
    } else {
        // Follow(E') 包括 ), ;
        logOutput("E' -> epsilon\n");
    }
}

// Grammar: T -> F T'
void parseT() {
    logOutput("T -> F T'\n");
    parseF();
    parseT_();
}

// Grammar: T' -> * F T' | epsilon
void parseT_() {
    if (currentToken.type == TOK_MULT) {
        logOutput("T' -> * F T'\n");
        match(TOK_MULT);
        parseF();
        parseT_();
    } else {
        // Follow(T') 包括 +, ), ;
        logOutput("T' -> epsilon\n");
    }
}

// Grammar: F -> ( E ) | id | num
void parseF() {
    if (currentToken.type == TOK_LPAREN) {
        logOutput("F -> ( E )\n");
        match(TOK_LPAREN);
        parseE();
        match(TOK_RPAREN);
    } else if (currentToken.type == TOK_ID) {
        logOutput("F -> id (%s)\n", currentToken.str);
        match(TOK_ID);
    } else if (currentToken.type == TOK_NUM) {
        logOutput("F -> num (%s)\n", currentToken.str);
        match(TOK_NUM);
    } else {
        error("Invalid Factor. Expected '(', identifier, or number.");
    }
}
```

i) 运行测试用例

输入:

```
x = 0.5 * (y + 10 + z) * 3; while(x) { x = x + 1; }
```

程序运行输出 (output.txt):

Syntax_Parser > output.txt

```
Lab2 LL(1) Syntax Parser
----------------------------------------
Input Stream: x = 0.5 * (y + 10 + z) * 3; while(x) { x = x + 1; }
----------------------------------------
Sequence of Derivations (Syntax Tree Construction):

Program -> Stmts
Stmts -> Stmt Stmts
Stmt -> AssignStmt
AssignStmt -> id = E ;
E -> T E'
T -> F T'
F -> num (0.5)
T' -> * F T'
F -> ( E )
E -> T E'
T -> F T'
F -> id (y)
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num (10)
T' -> epsilon
E' -> + T E'
T -> F T'
F -> id (z)
T' -> epsilon
E' -> epsilon
```

Syntax_Parser > output.txt

```
T' -> * F T'
F -> num (3)
T' -> epsilon
E' -> epsilon
Stmts -> Stmt Stmts
Stmt -> WhileStmt
WhileStmt -> while ( E ) Stmt
E -> T E'
T -> F T'
F -> id (x)
T' -> epsilon
E' -> epsilon
Stmt -> Block
Block -> { Stmts }
Stmts -> Stmt Stmts
Stmt -> AssignStmt
AssignStmt -> id = E ;
E -> T E'
T -> F T'
F -> id (x)
T' -> epsilon
E' -> + T E'
T -> F T'
F -> num (1)
T' -> epsilon
E' -> epsilon
Stmts -> epsilon
Stmts -> epsilon
```

j) 遇到的问题及解决方案

1. **问题：左递归导致的无限循环**
   - 描述：*最初按照数学习惯定义文法 E -> E + T，导致 parseE() 刚开始就无限调用 parseE()，造成栈溢出。*
   - 解决：将文法改写为右递归形式 E -> T E'，并引入空产生式 `єє`。
2. **问题：悬空 Else (Dangling Else)**
   - 描述：*解析 if (E) if (E) S else S 时，无法确定 else 属于哪个 if。*
   - 解决：通过提取左因子，将 if 语句定义为必须包含 ElsePart。在代码中，parseElsePart 优先匹配 else 关键字，这隐式地实现了“else 与最近的未匹配 if 匹配”的规则（贪心匹配）。
3. **问题：输出格式混乱**
   - 描述：*需要同时在控制台查看进度，并保存结果到文件，单纯用 printf 不够。*
   - 解决：封装了 logOutput 函数，使用 stdarg.h 库处理可变参数，实现了一次调用双重输出。

k) 心得体会

通过本次实验，我从理论到实践完整地走通了语法分析的流程。

1. **文法即代码**：我深刻体会到了递归下降分析法的优美之处——文法的结构直接映射为代码的函数调用结构。只要文法设计得当（满足 LL(1)），代码编写几乎是机械式的翻译过程。
2. **预处理的重要性**：实验中最困难的部分不是写代码，而是设计文法。消除左递归和二义性是保证分析器正常工作的前提。

3. **局限性**：LL(1) 文法对写法的限制较大（不能左递归），这使得表达式的文法变得不如原始文法直观（如 E' 的引入）。相比之下，LR 分析法虽然手工构造复杂，但能处理更广泛的文法，这是后续值得深入学习的方向。