编译原理实验报告

实验1	**词法分析器（Lexical Analyzer Programming）**
实验2	**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

实验一：词法分析器

a) 实验动机 / 目的

1. **理解词法分析原理：** 深入理解编译器前端的工作机制，特别是如何将源代码字符流转换为抽象的词法单元（Token）序列。
2. **掌握有限自动机（FA）应用：** 学习如何将正则表达式（REs）转化为确定有限自动机（DFA），并使用编程语言（C语言）实现基于 DFA 的状态跳转逻辑。
3. **提升编程实践能力：** 练习文件 I/O 操作、字符串处理以及缓冲区管理，为后续的语法分析实验打下基础。

b) 内容描述

本实验实现了一个简单的词法分析器（Scanner）。

- **输入：** 包含源代码的文本文件（字符流）。语言子集包含关键字（如 if, then）、标识符、数字、运算符（如 :=, <）和界符。
- **输出：** 识别出的 Token 序列，包含 Token 的类型（TAG）和属性值（Attribute）。输出结果同时显示在控制台并保存至 output.txt 文件中。
- **功能范围：**识别保留字：if, then, else, end, repeat, until, read, write。识别整数与简单标识符。识别双字符运算符（如 :=, <=, >=）及单字符运算符。过滤空白符（空格、换行、制表符）。处理并忽略注释（形如 { ... }）。基本的错误检测（非法字符处理）。

c) 设计思路 / 方法

本实验采用 **手工构造 DFA 并模拟运行** 的方法。

1. **正则定义：** 首先明确各类单词的正则表达式规则。
2. **状态机设计：** 设计一个确定有限自动机（DFA），定义初始状态、中间状态和接受状态。
3. **超前扫描（Lookahead）：** 针对 :=, <=, <> 等存在前缀重叠的运算符，采用“超前扫描”策略。即读入一个字符后，预读下一个字符以确定状态跳转；若不匹配，则利用 ungetc 回退字符。
4. **最大匹配原则：** 在识别标识符和数字时，尽可能多地读取符合规则的字符，直到遇到第一个不符合规则的字符为止。
5. **双输出流：** 利用缓冲区先格式化字符串，再分别写入 stdout 和文件流，实现同时输出。

d) 假设条件

1. **字符集：** 源代码仅包含 ASCII 字符。
2. **注释格式：** 假设注释以 { 开始，以 } 结束（根据代码实现），且不支持嵌套注释。
3. **标识符限制：** 标识符由字母开头，后跟字母或数字，长度不超过 99 个字符。
4. **大小写敏感：** 关键字（如 if）是大小写敏感的（即 IF 会被识别为标识符而非关键字）。
5. **数字格式：** 仅处理无符号整数（代码逻辑主要针对整数，遇到 . 会停止或报错，除非扩充浮点逻辑）。

e) 相关有限自动机 (FA) 描述

程序通过 switch-case 结构模拟了一个包含以下核心状态的 DFA：

- **START (初始态)：** 接收字符，根据字符类型跳转：Letter `→→` IN_IDDigit `→→` IN_NUM: `→→` IN_ASSIGN< `→→` IN_LESS> `→→` IN_GREATER{ `→→` IN_COMMENTSpace/Newline `→→` START (循环)
- **IN_ID (标识符态)：** 持续读取字母或数字，直到遇到非字母数字字符 `→→` DONE（查表区分 ID 或 Keyword）。
- **IN_NUM (数字态)：** 持续读取数字，直到遇到非数字 `→→` DONE。
- **IN_ASSIGN (赋值态)：** 已读入 :，检查下一字符：= `→→` DONE (识别 :=)其他 `→→` ERROR。
- **IN_LESS (小于态)：** 已读入 <，检查下一字符：= `→→` DONE (识别 <=)> `→→` DONE (识别 <>)其他 `→→` DONE (识别 <，回退指针)。
- **IN_COMMENT (注释态)：** 忽略所有字符，直到遇到 } `→→` START。
- **DONE (完成态)：** 标记一个 Token 识别完成，返回主循环。

f) 重要数据结构说明

1. **enum TokenType**用于枚举所有可能的 Token 类型，如 IF, ID, NUM, ASSIGN 等，便于代码中进行逻辑判断和分类。
2. **enum StateType**定义 DFA 的所有状态（START, IN_ID, IN_NUM 等），控制 switch-case 的流程流转。
3. **struct keywords**一个包含字符串和对应 Token 类型的结构体数组，作为**查找表。**用于在识别出字符串后，通过 lookup() 函数判断其是否为保留字。
4. **FILE *source, FILE *outputFile**文件指针，分别用于管理源代码的读取流和结果的写入流。

g) 核心算法说明

核心算法是 getToken() 函数，其逻辑如下：

1. 初始化状态 state = START，清空 token 字符串缓冲区。
2. 进入 while (state != DONE) 循环：读取下一个字符 c = getNextChar()。根据当前 state 和输入 c 进行 switch 判断。
    - **状态转换：** 如果匹配规则，更新 state 到新状态。
    - **字符保存：** 如果该字符属于当前 Token 的一部分，将其存入 tokenString 数组。
    - **回退处理：** 如果读到了不属于当前 Token 的字符（例如在数字后读到了空格），调用 ungetNextChar(c) 将字符放回流中，并标记 state = DONE。
3. 循环结束后，如果 Token 类型是 ID，调用 lookup() 函数检查是否为关键字，修正 Token 类型。

4. 返回识别到的 TokenType。

h）实验代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// ============================================
// 1. 定义数据结构
// ============================================

// Token 类型定义
typedef enum {
    // 关键字
    IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE,
    // 类别
    ID, NUM,
    // 运算符
    PLUS, MINUS, TIMES, OVER,    // + - * /
    ASSIGN,                      // :=
    EQ, LT, GT, LTE, GTE, NEQ,   // = < > <= >= <>
    // 界符
    LPAREN, RPAREN, SEMI,        // ( ) ;
    // 结束与错误
    ENDFILE, ERROR
} TokenType;

// DFA 状态定义
typedef enum {
    START,
    IN_ID,
    IN_NUM,
    IN_ASSIGN,   // 读到了 :
    IN_LESS,     // 读到了 <
    IN_GREATER,  // 读到了 >
    IN_COMMENT,  // 读到了 {
    DONE         // 完成一个Token识别
} StateType;

// 全局变量
FILE *source;       // 输入源文件指针
FILE *outputFile;   // 输出结果文件指针
char tokenString[100];
int lineNo = 1;

// 关键字查找表
struct {
    char *str;
    TokenType tok;
} keywords[] = {
    {"if", IF}, {"then", THEN}, {"else", ELSE}, {"end", END},
```

```
    {"repeat", REPEAT}, {"until", UNTIL}, {"read", READ}, {"write", WRITE},
    {NULL, 0}
};

// ========================================
// 2. 辅助函数
// ========================================

// 查表判断是否为关键字
TokenType lookup(char *s) {
    for (int i = 0; keywords[i].str != NULL; i++) {
        if (strcmp(s, keywords[i].str) == 0)
            return keywords[i].tok;
    }
    return ID;
}

char getNextChar() {
    return fgetc(source);
}

void ungetNextChar(char c) {
    ungetc(c, source);
}


// 使用 sprintf 先格式化，然后同时写入屏幕和文件
void printToken(TokenType token, const char *tokenString) {
    char buffer[256]; // 临时缓冲区，用于存放一行输出信息

    switch (token) {
        case IF: case THEN: case ELSE: case END:
        case REPEAT: case UNTIL: case READ: case WRITE:
            sprintf(buffer, "KEYWORD: %s\n", tokenString); break;
        case ASSIGN: sprintf(buffer, "ASSIGN: :=\n"); break;
        case LT: sprintf(buffer, "RELOP: <\n"); break;
        case GT: sprintf(buffer, "RELOP: >\n"); break;
        case EQ: sprintf(buffer, "RELOP: =\n"); break;
        case NEQ: sprintf(buffer, "RELOP: <>\n"); break;
        case LTE: sprintf(buffer, "RELOP: <=\n"); break;
        case GTE: sprintf(buffer, "RELOP: >=\n"); break;
        case LPAREN: sprintf(buffer, "SEMI: (\n"); break;
        case RPAREN: sprintf(buffer, "SEMI: )\n"); break;
        case SEMI: sprintf(buffer, "SEMI: ;\n"); break;
        case PLUS: sprintf(buffer, "OP: +\n"); break;
        case MINUS: sprintf(buffer, "OP: -\n"); break;
        case TIMES: sprintf(buffer, "OP: *\n"); break;
        case OVER: sprintf(buffer, "OP: /\n"); break;
        case NUM: sprintf(buffer, "NUM: %s\n", tokenString); break;
        case ID: sprintf(buffer, "ID: %s\n", tokenString); break;
        case ENDFILE: sprintf(buffer, "EOF\n"); break;
        case ERROR: sprintf(buffer, "ERROR: Unexpected character '%s' at line %d\n",
tokenString, lineNo); break;
```

```
        default: sprintf(buffer, "UNKNOWN TOKEN\n"); break;
    }

    // 1. 输出到屏幕
    printf("%s", buffer);

    // 2. 输出到文件 (如果文件打开成功)
    if (outputFile != NULL) {
        fprintf(outputFile, "%s", buffer);
    }
}

// ========================================
// 3. 核心算法：DFA 驱动
// ========================================
TokenType getToken() {
    int tokenStringIndex = 0;
    TokenType currentToken;
    StateType state = START;
    int save;

    while (state != DONE) {
        char c = getNextChar();
        save = 1;

        switch (state) {
            case START:
                if (isdigit(c)) state = IN_NUM;
                else if (isalpha(c)) state = IN_ID;
                else if (c == ':') state = IN_ASSIGN;
                else if (c == '<') state = IN_LESS;
                else if (c == '>') state = IN_GREATER;
                else if (c == ' ' || c == '\t' || c == '\r') save = 0;
                else if (c == '\n') { save = 0; lineNo++; }
                else if (c == '{') { save = 0; state = IN_COMMENT; }
                else {
                    state = DONE;
                    switch (c) {
                        case EOF: save = 0; currentToken = ENDFILE; break;
                        case '=': currentToken = EQ; break;
                        case '+': currentToken = PLUS; break;
                        case '-': currentToken = MINUS; break;
                        case '*': currentToken = TIMES; break;
                        case '/': currentToken = OVER; break;
                        case '(': currentToken = LPAREN; break;
                        case ')': currentToken = RPAREN; break;
                        case ';': currentToken = SEMI; break;
                        default: currentToken = ERROR; break;
                    }
                }
                break;

            case IN_COMMENT:
```

```
                save = 0;
                if (c == '}') state = START;
                else if (c == '\n') lineNo++;
                else if (c == EOF) { state = DONE; currentToken = ENDFILE; }
                break;

            case IN_ASSIGN:
                state = DONE;
                if (c == '=') currentToken = ASSIGN;
                else { ungetNextChar(c); save = 0; currentToken = ERROR; }
                break;

            case IN_LESS:
                state = DONE;
                if (c == '=') currentToken = LTE;
                else if (c == '>') currentToken = NEQ;
                else { ungetNextChar(c); save = 0; currentToken = LT; }
                break;

            case IN_GREATER:
                state = DONE;
                if (c == '=') currentToken = GTE;
                else { ungetNextChar(c); save = 0; currentToken = GT; }
                break;

            case IN_NUM:
                if (!isdigit(c) && c != '.') {
                    ungetNextChar(c); save = 0; state = DONE; currentToken = NUM;
                }
                break;

            case IN_ID:
                if (!isalnum(c)) {
                    ungetNextChar(c); save = 0; state = DONE; currentToken = ID;
                }
                break;

            case DONE: default:
                state = DONE; currentToken = ERROR; break;
        }

        if ((save) && (tokenStringIndex < 99)) tokenString[tokenStringIndex++] = c;
        if (state == DONE) {
            tokenString[tokenStringIndex] = '\0';
            if (currentToken == ID) currentToken = lookup(tokenString);
        }
    }
    return currentToken;
}

// ============================================
// 4. 主程序
// ============================================
```

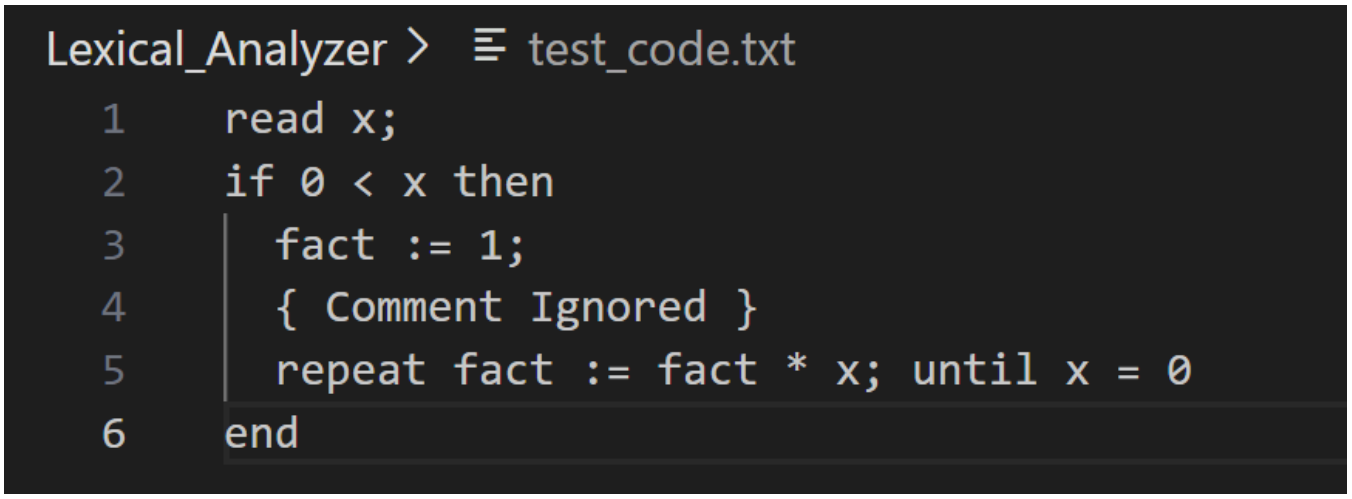
```c
int main() {
    // 1. 生成测试用的输入文件
    FILE *fp = fopen("test_code.txt", "w");
    if (fp) {
        fprintf(fp, "read x;\n");
        fprintf(fp, "if 0 < x then\n");
        fprintf(fp, "  fact := 1;\n");
        fprintf(fp, "  { Comment Ignored }\n");
        fprintf(fp, "  repeat fact := fact * x; until x = 0\n");
        fprintf(fp, "end");
        fclose(fp);
    }

    // 2. 打开输入文件
    source = fopen("test_code.txt", "r");
    if (source == NULL) {
        printf("Error: Could not open source file.\n");
        return 1;
    }

    // 3. 【修改】打开输出文件
    outputFile = fopen("output.txt", "w");
    if (outputFile == NULL) {
        printf("Error: Could not create output file.\n");
        // 即使输出文件创建失败，我们仍然继续运行，只是只输出到屏幕
    }

    printf("Analysis started. Results will be saved to 'output.txt'.\n");
    printf("============================================================\n");

    TokenType token;
    while ((token = getToken()) != ENDFILE) {
        printToken(token, tokenString);
    }

    // 4. 清理资源
    fclose(source);
    if (outputFile != NULL) {
        fclose(outputFile);
        printf("============================================================\n");
        printf("Analysis finished. Check 'output.txt' for results.\n");
    }

    return 0;
}
```

i) 运行测试用例

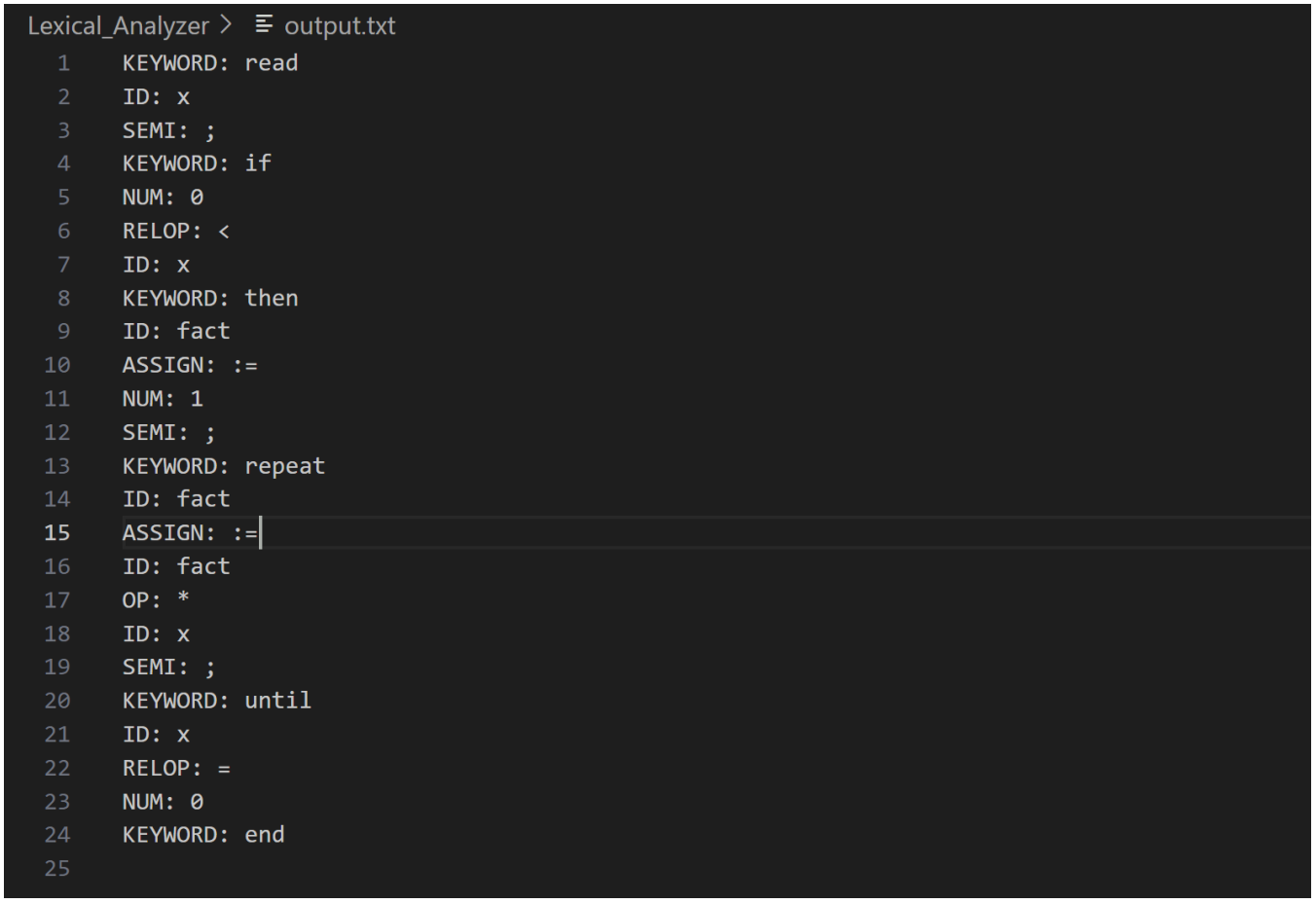
输入文件 (test_code.txt):

```
read x;
if 0 < x then
  fact := 1;
  { Comment Ignored }
  repeat fact := fact * x; until x = 0
end
```

截图如下：



运行结果 (截取自 output.txt):



(注：程序成功跳过了 { Comment Ignored } 这一行注释)

行号	Token 类型 (Tag)	属性值 (Attribute/String)	说明 (Description)
1	**KEYWORD**	read	关键字 read
2	**ID**	x	标识符 x
3	**SEMI**	;	分号
4	**KEYWORD**	if	关键字 if
5	**NUM**	0	数字常量
6	**RELOP**	<	小于号
7	**ID**	x	标识符 x
8	**KEYWORD**	then	关键字 then
9	**ID**	fact	标识符 fact
10	**ASSIGN**	:=	赋值符号
11	**NUM**	1	数字常量
12	**SEMI**	;	分号
13	**KEYWORD**	repeat	关键字 repeat
14	**ID**	fact	标识符 fact
15	**ASSIGN**	:=	赋值符号
16	**ID**	fact	标识符 fact
17	**OP**	*	乘法运算符
18	**ID**	x	标识符 x
19	**SEMI**	;	分号
20	**KEYWORD**	until	关键字 until
21	**ID**	x	标识符 x
22	**RELOP**	=	等于号
23	**NUM**	0	数字常量
24	**KEYWORD**	end	关键字 end
25	**EOF**	-	文件结束符

j) 遇到的问题及解决方案

1. **问题：注释处理逻辑混淆**
    - 描述： 最初将注释视为一种特殊的 Token，导致输出中包含注释内容。

    - *解决：* 修改 DFA，在 IN_COMMENT 状态下不保存字符（save = 0），并且在遇到结束符 } 后直接跳转回 START 状态，而不是 DONE 状态。
2. **问题：区分关键字与标识符**
    - 描述： 所有的关键字（如 if）本质上符合标识符的正则规则，很容易被误判为 ID
    - *解决：* 采用"先整体后局部"的策略。DFA 将所有字母开头的串都先识别为 ID，识别结束后，统一查表（Lookup Table）。如果在表中找到，则修正为对应的关键字类型，否则保留为 ID。
3. **问题：多读字符的处理**
    - 描述：识别 NUM 或 ID 时，必须读到非数字/非字母才能停止，这导致文件指针多前进了一位。
    - *解决：* 使用标准库函数 ungetc() 实现回退机制，将多读的那个字符放回输入流，供下一次 getToken 使用。

k) 心得体会

通过本次实验，我深刻体会到了有限自动机在计算机科学中的基础作用。虽然手动编写大量的 switch-case 看起来有些繁琐，但它清晰地展示了状态流转的过程，让我明白了编译器是如何"理解"源代码的第一步的。

这种基于 DFA 的设计具有良好的扩展性。如果将来需要增加新的运算符或语法规则（如支持浮点数或字符串字面量），只需在 switch 结构中增加相应的状态分支即可。此外，同时输出到屏幕和文件的功能也让我复习了 C 语言的文件操作，是一次非常有价值的实践。

实验二：语法分析器

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∣T` `E→E+T∣T` 变换后：`E→TE′` `E→TE′`，`E′→+TE′∣ϵ` `E′→+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):

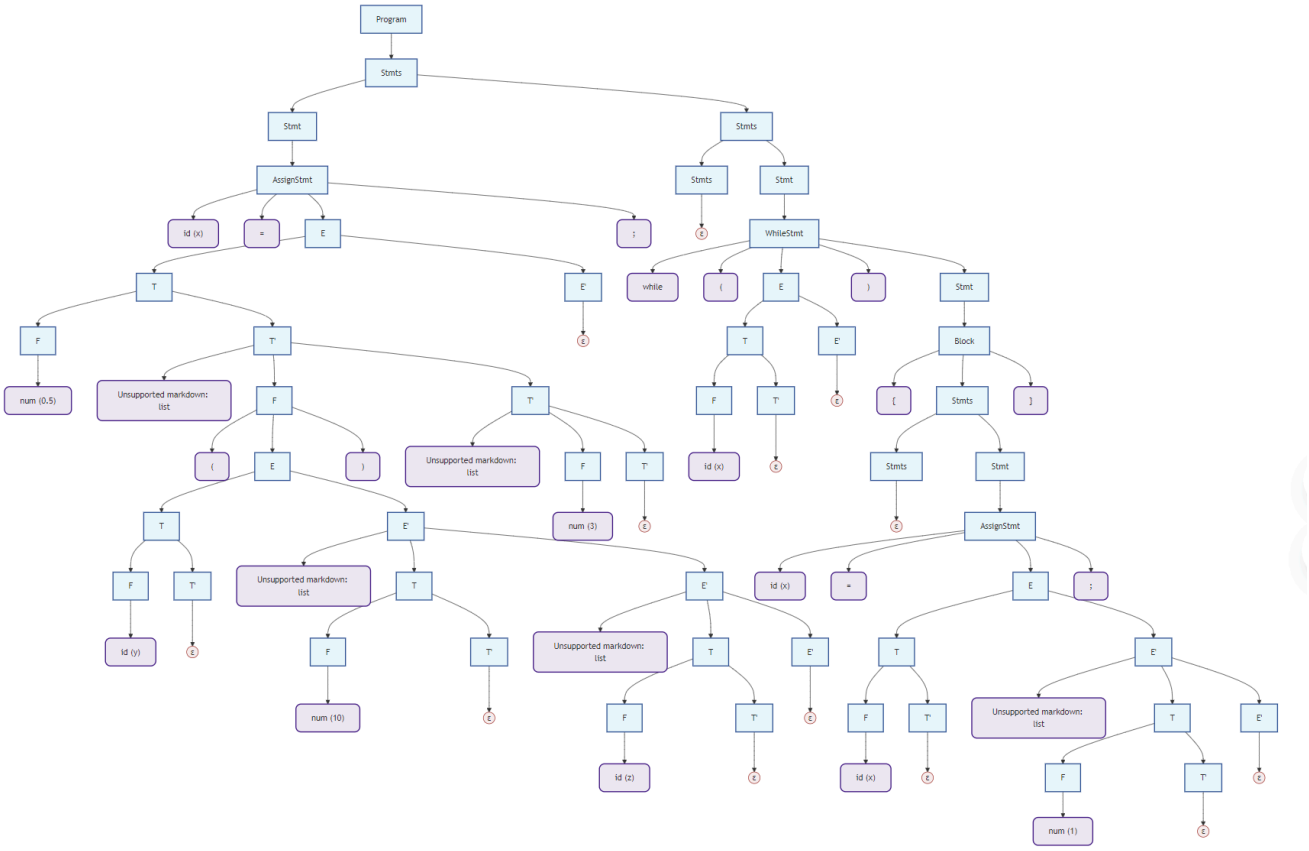
Syntax_Parser > output.txt

```
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
```

语法分析树（parsing tree）：



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 `ϵϵ`。
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。