# 编译原理 实验报告一

| 实验名称 | Lab 1: Lexical Analyzer Programming |
| --- | --- |
| 姓名 | 熊宇祺 |
| 学号 | 09023112 |
| 报告日期 | 2025年11月25日 |

---

## 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
// ============================================
int main() {
```

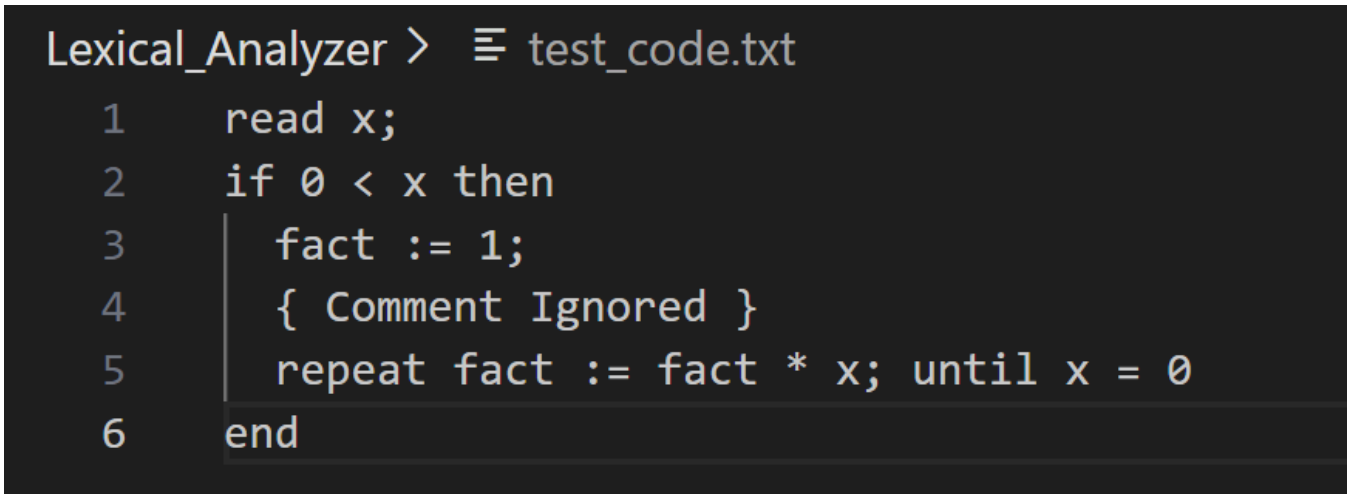
```
    // 1. 生成测试用的输入文件
    FILE *fp = fopen("test_code.txt", "w");
    if (fp) {
        fprintf(fp, "read x;\n");
        fprintf(fp, "if 0 < x then\n");
        fprintf(fp, "  fact := 1;\n");
        fprintf(fp, "  { Comment Ignored }\n");
        fprintf(fp, "  repeat fact := fact * x; until x =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

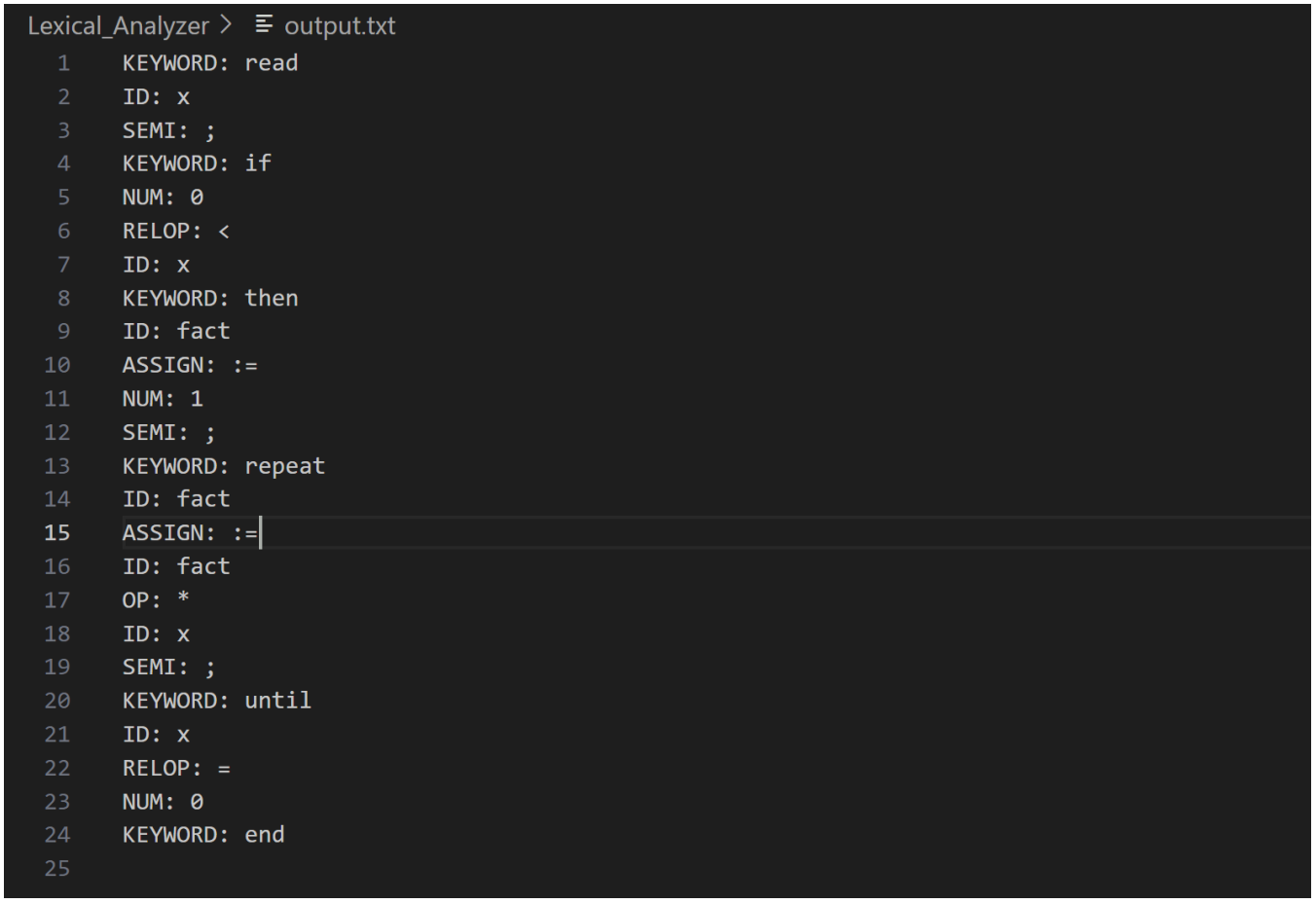
**输入文件 (test_code.txt ):**

```
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。