词法分析实验报告（C语言版）

一、实验要求

题目：C语言词法分析程序的设计与实现

实验内容及要求：

1. 可以识别出用C语言编写的源程序中的每个单词符号，并以记号的形式输出每个单词符号。
2. 可以识别并跳过源程序中的注释。
3. 可以统计源程序中的语句行数、各类单词的个数、以及字符总数，并输出统计结果。
4. 检查源程序中存在的词法错误，并报告错误所在的位置。
5. 对源程序中出现的错误进行适当的恢复，使词法分析可以继续进行，对源程序进行一次扫描，即可检查并报告源程序中存在的所有词法错误。

实现方法要求：分别用以下两种方法实现：

方法1：采用C/C++作为实现语言，手工编写词法分析程序。（必做）

方法2：编写LEX源程序，利用LEX编译程序自动生成词法分析程序。

二、实验环境

操作系统：Windos10

IDE：Visual Studio 2019（C++17）

GUI设计工具：Qt 5.14

三、实验原理

3.1 词法分析器的概念

词法分析（**lexical analysis**）是计算机科学中将字符序列转换为**标记**（token）序列的过程。进行词法分析的程序或者函数叫作**词法分析器**（lexical analyzer，简称lexer），也叫**扫描器**（scanner）。词法分析器一般以函数的形式存在，供语法分析器调用。

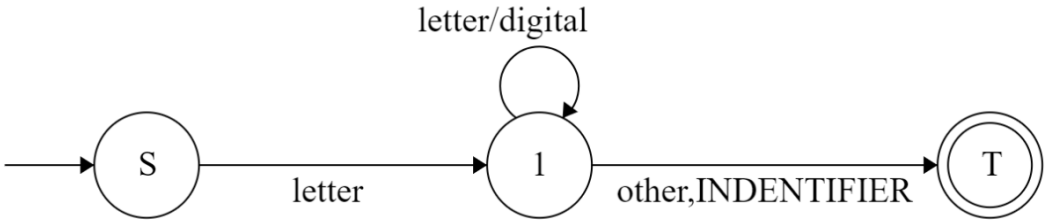
3.2 词法分析自动机

3.2.1自动机设计

词法分析自动机为一个具有初态、中间态、终态、错误态的Mealy自动机，在初态与中间态时不产生输出，在终态与错误态时产生相应的识别结果和错误信息输出，除此之外还输出词法分析器需要的回退字符数。于此同时，在Mealy自动机每次到达终态时，自动通过 ε 转移到初态，且不产生输出，准备进行下次的识别。

3.2.2 匹配标识符或关键字自动机

在自动机初始状态下，若当前匹配到的字符是**字母**，就继续匹配下一个字符，不断地接受**数字**或**字母**，直到下个字符不是**字母**、也不是**数字**为止，此时返回Mealy机到达终态，输出`IDENTIFIER`，且需要回退1位字符。而后取出识别出的字符串，判断这个字符串是不是**关键字**，如果是**关键字**，将其类型更改为对应的**关键字**，否则保留其**标识符**类型。最后，将其存入符号表中。



3.2.3 匹配数字自动机

在自动机初始状态下，若当前匹配到的字符是数字，就继续匹配下一个字符，不断地接收数字，直到下个字符不是数字为止。

此时分为三种情况：

1.当前匹配字符为`.`

进入实数识别状态。

- 如果在`.`之后输入的不是数字，则进入错误态，输出`INCOMPLETE_NUMBERIC_ERROR`，且需要回退1位字符，这是一个不完整的实数类型，有小数点却没有小数部分。
- 如果此时输入数字，则不断接收数字，直到
  - 匹配`E`或`e`：进入科学计数识别状态。
  - 匹配字符为非`E`或`e`的其他非数字字符：进入终态，输出`CONSTANT_REAL`，且需要回退1位字符，自动机识别结果为常实数。

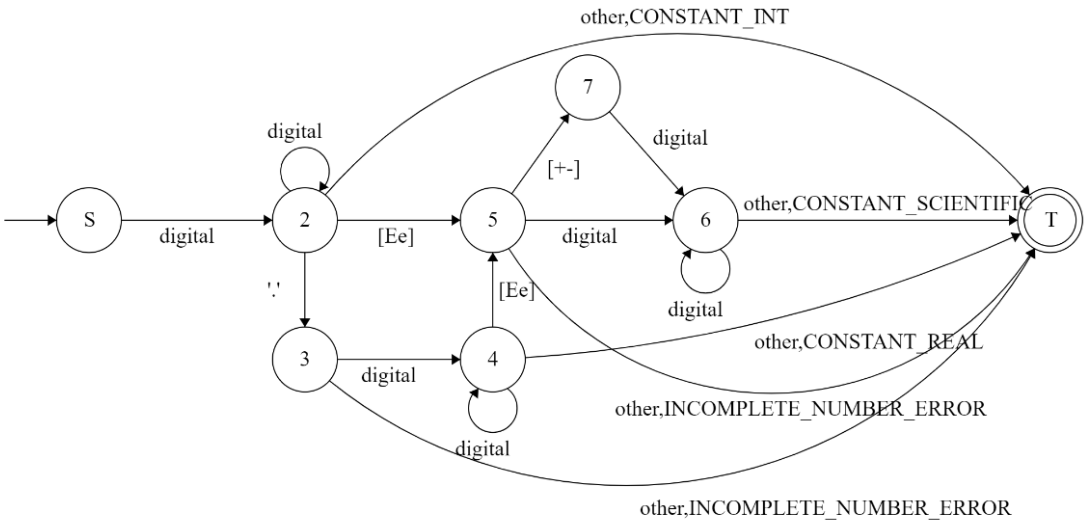
2.当前匹配字符为`E`或`e`

进入科学计数识别状态。

- 如果接收到的是数字，则不断接受数字，直到下个字符不是数字为止，此时进入终态，输出`CONSTANT_SCIENTIFIC`，且需要回退1位字符，自动机识别结果为常科学计数。
- 如果在`E`或`e`后第一个输入的是`+`或`-`，则接收该符号后转入当前分支下的第一种状态（见上一行）
- 如果在`E`或`e`后第一个输入的既不是`+`或`-`也不是数字，则进入错误态，输出`INCOMPLETE_NUMBERIC_ERROR`，且需要回退1位字符，这是一个不完整的科学计数类型，无指数部分的数字。

3.当前匹配字符为非`.`、`E`或`e`的其他字符

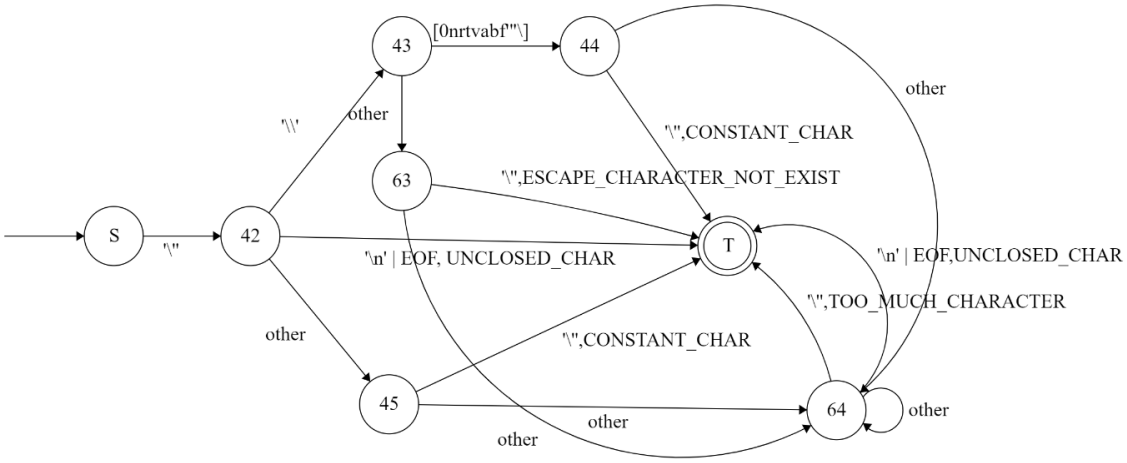
进入终态，输出`CONSTANT_INT`，且需要回退1位字符，自动机识别结果为常整数。



3.2.4 匹配单个字符自动机

在自动机初始状态下，若当前匹配到的字符是`'`，进入字符匹配状态。考虑下一个匹配的字符：

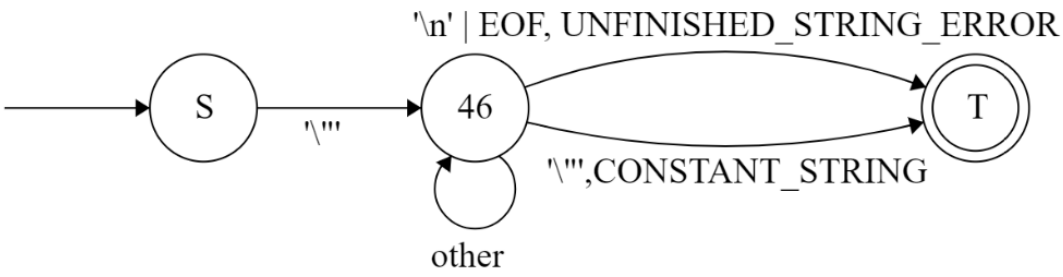
- 当前匹配字符为`'\'`。进入转义字符判断，若接下来输入的字符是`0nrtvabf'"\`的其中之一，则：
  - 下一个字符是`'`，完成字符识别，输出`CONSTANT_CHAR`，不需要回退。
  - 下一个字符不是`'`，则该单引号中包含字符不止一个，有可能是字符未闭合错误，也有可能是字符过多错误，需要进一步判断。
- 当前匹配字符为`'\'`。进入转义字符判断，若接下来输入的字符**不**是`0nrtvabf'"\`的其中之一，则：
  - 下一个字符是`'`，完成字符识别，此时的错误为该转义符不存在，输出`ESCAPE_CHARACTER_NOT_EXIST`，不需要回退。
  - 下一个字符不是`'`，则该单引号中包含字符不止一个，有可能是字符未闭合错误，也有可能是字符过多错误，需要进一步判断。
- 当前匹配字符不为`'\'`。进入普通字符判断，接下来输入的字符可以是任意除了`'\'`外的符号，考虑再下一个输入的字符：
  - 下一个字符是`'`，完成字符识别，输出`CONSTANT_CHAR`，不需要回退。
  - 下一个字符不是`'`，则该单引号中包含字符不止一个，有可能是字符未闭合错误，也有可能是字符过多错误，需要进一步判断。
- 若在第一个`'\'`后出现不止一个非`'`字符，则单引号中包含字符不止一个。继续执行自动机
  - 若读到`'`，则进入`TOO_MUCH_CHARACTER`错误态，两个单引号之间包含过多字符（形如`'abc'`这样的字符），不需要回退。
  - 若读到`\n`，则进入`UNCLOSED_CHAR_ERROR`错误态，只有一个单引号的字符（形如`'a`这样的字符），不需要回退。



3.2.5 匹配字符串自动机

在自动机初始状态下，若当前匹配到的字符是`"`，进入字符串匹配状态。不断读入字符，直到遇到`"`，`\n`，或EOF

- 若遇到`"`，则字符串正常识别完毕，进入终态，自动机输出`CONSTANT_STRING`，不需要回退
- 若遇到`\n`或EOF，则字符串不完整，进入错误态，自动机输出`UNFINISHED_STRING_ERROR`，不需要回退

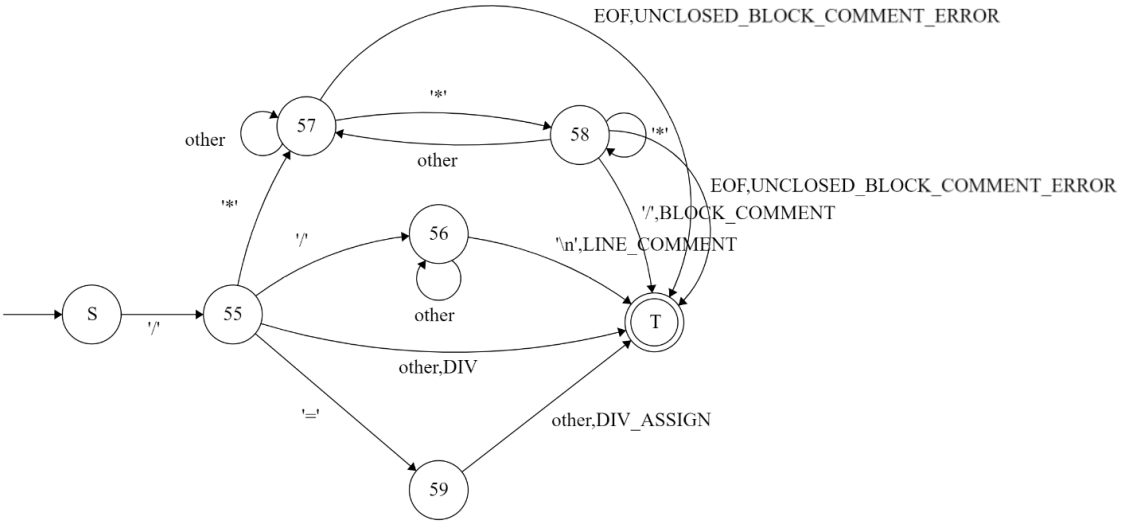


3.2.6 匹配行注释与注释块自动机

在自动机初始状态下，若当前匹配到的字符是`'/'`，则进入注释匹配状态。

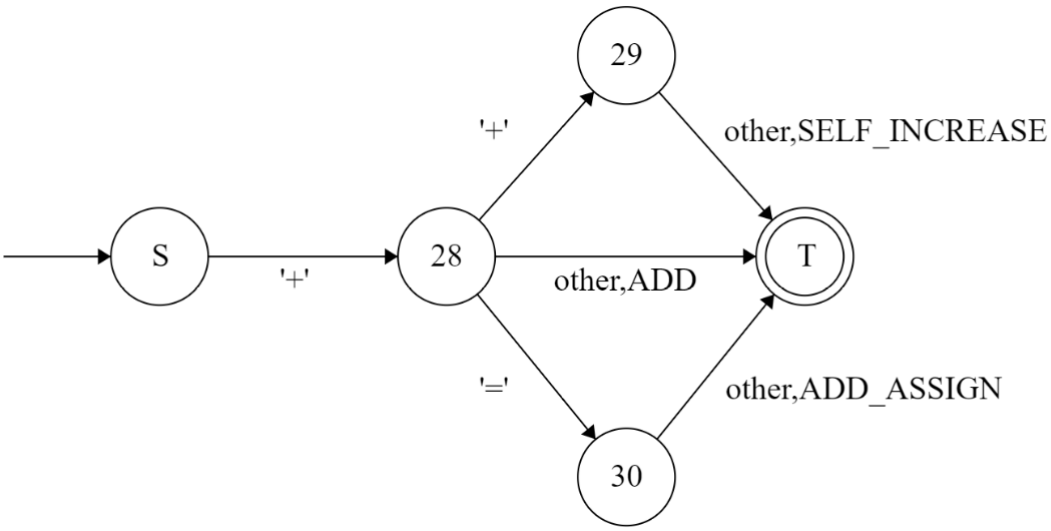
- 若下一个输入的字符是`'/'`，则进入行注释匹配状态，不断读入字符，直到`'\n'`时进入终态，完成行注释的识别，输出`LINE_BREAK`，不需要回退。
- 若下一个输入的字符是`'*'`，则进入块注释匹配状态，不断读入字符，直到：
  - 遇到`'*'`，继续读入字符，若为`*`保持当前状态；若此时读入EOF同下处理；若为`'/'`则进入终态，完成块注释的识别，输出`BLOCK_COMMENT`，不需要回退；
  - 遇到EOF，进入错误态，块注释未闭合，输出`UNCLOSED_BLOCK_COMMENT_ERROR`，不需要退回
  - 遇到除`'*'`与EOF以外的字符，退回上一状态，继续读入字符，直到遇到`'*'`进入此分支处理。
- 若遇到EOF进入错误态，块注释未闭合，输出`UNCLOSED_BLOCK_COMMENT_ERROR`，不需要退回

- 若遇到除`'/'`与`'*'`的以外字符（此时输入EOF是合法的），则识别该符号为除号，进入终态，输出`DIV`，需要退回一位。



3.2.7 匹配行运算符、分节符等其他符号的自动机

符号处理大体上相同，识别出具体符号后输出结果，并且需要回退一位，这里仅展示`'+'`开头的符号识别，详细符号处理见后文完成的自动机图示。

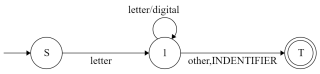


3.2.8 输入字符错误特殊处理

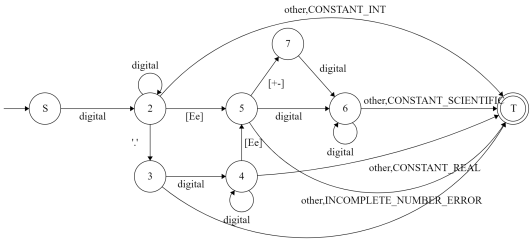
C语言中不存在以`$`与`@`开头的词法，且语言中不应出现非ASCII字符。

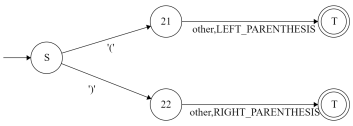
- 对于`$`与`@`，在初始状态下匹配到时进入错误态，输出`ILLEGAL_CHAR_ERROR`，且不退回。
- 对于非ASCII字符，在任何时候出现都立即跳过，不让其输入自动机，此时另外输出`ILLEGAL_CHAR_ERROR`，且不退回。

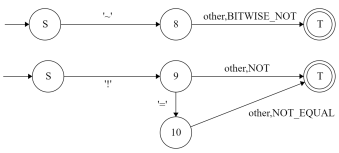
3.3 词法分析自动机总览

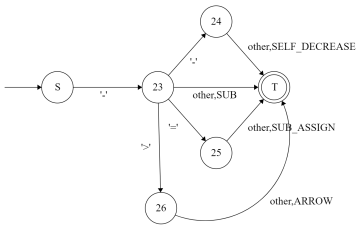

letter/digital
S
letter
1
other,INDENTIFIER
T

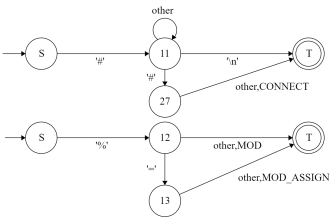

S
'*'
20
other,STAR
T

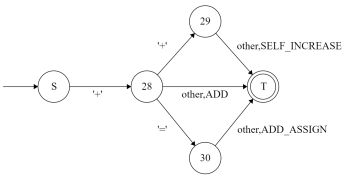

other,CONSTANT_INT
digital
digital
[+-]
digital
S
digital
2
[Ee]
5
digital
6
other,CONSTANT_SCIENTIFIC
T
'.'
[Ee]
digital
3
digital
4
other,CONSTANT_REAL
other,INCOMPLETE_NUMBER_ERROR
digital
other,INCOMPLETE_NUMBER_ERROR

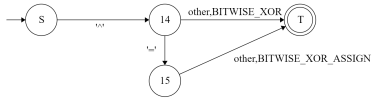

S
'('
21
other,LEFT_PARENTHESIS
T
')'
22
other,RIGHT_PARENTHESIS
T

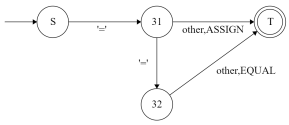

S
'~'
8
other,BITWISE_NOT
T
S
'!'
9
other,NOT
T
'='
other,NOT_EQUAL
10

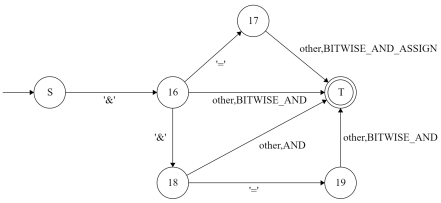

24
other,SELF_DECREASE
'-'
S
'-'
23
other,SUB
T
'='
other,SUB_ASSIGN
'>'
25
26
other,ARROW

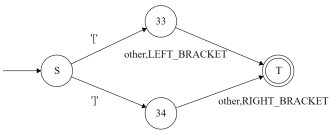

other
S
'#'
11
'\n'
T
'#'
27
other,CONNECT
S
'%'
12
other,MOD
T
'='
other,MOD_ASSIGN
13

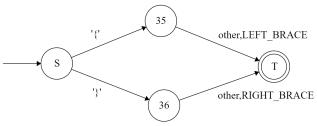

29
'+'
other,SELF_INCREASE
S
'+'
28
other,ADD
T
'='
other,ADD_ASSIGN
30

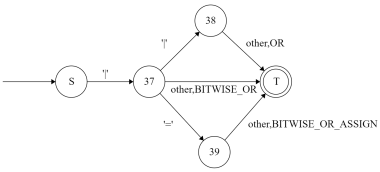

S
'^'
14
other,BITWISE_XOR
T
'='
other,BITWISE_XOR_ASSIGN
15

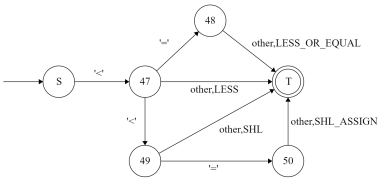

S
'='
31
other,ASSIGN
T
'='
other,EQUAL
32

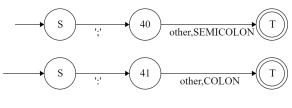

17
'='
other,BITWISE_AND_ASSIGN
S
'&'
16
other,BITWISE_AND
T
'&'
other,AND
other,BITWISE_AND
18
'='
19

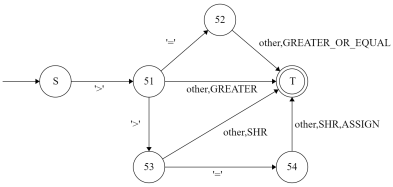

33
'['
other,LEFT_BRACKET
S
T
']'
34
other,RIGHT_BRACKET

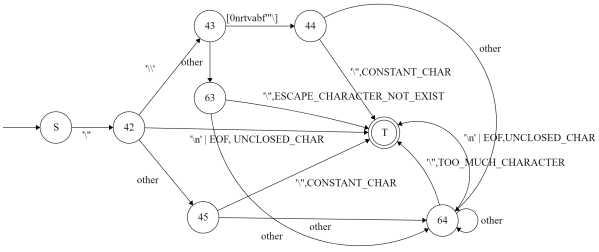

35
'{'
other,LEFT_BRACE
S
T
'}'
36
other,RIGHT_BRACE

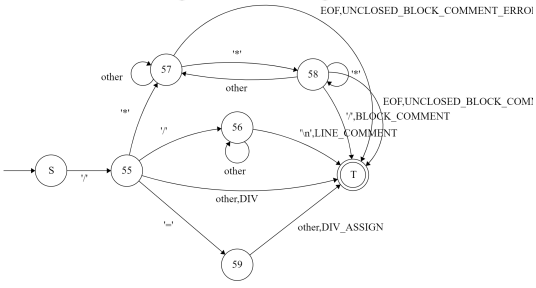

38
'|'
other,OR
S
'|'
37
other,BITWISE_OR
T
'='
other,BITWISE_OR_ASSIGN
39

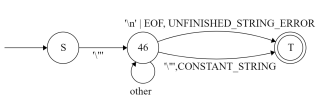

48
'='
other,LESS_OR_EQUAL
S
'<'
47
other,LESS
T
'<'
other,SHL
other,SHL_ASSIGN
49
'='
50


S
';'
40
other,SEMICOLON
T
S
':'
41
other,COLON
T

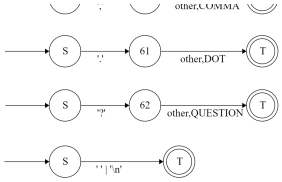

52
'='
other,GREATER_OR_EQUAL
S
'>'
51
other,GREATER
T
'>'
other,SHR
other,SHR_ASSIGN
53
'='
54


43
[0nrtvabf'"\]
44
other
'\\'
other
"\",CONSTANT_CHAR
63
"\",ESCAPE_CHARACTER_NOT_EXIST
S
"\"
42
T
"\n' | EOF, UNCLOSED_CHAR
"\n' | EOF,UNCLOSED_CHAR
"\",TOO_MUCH_CHARACTER
other
"\",CONSTANT_CHAR
45
64
other
other
other


EOF,UNCLOSED_BLOCK_COMMENT_ERRO
'*'
other
57
58
'*'
other
EOF,UNCLOSED_BLOCK_COM
'*'
'/'
56
"\n',LINE_COMMENT
'/',BLOCK_COMMENT
S
'/'
55
other
T
other,DIV
'='
other,DIV_ASSIGN
59


"\n' | EOF, UNFINISHED_STRING_ERROR
S
"\""
46
T
"\"",CONSTANT_STRING
other

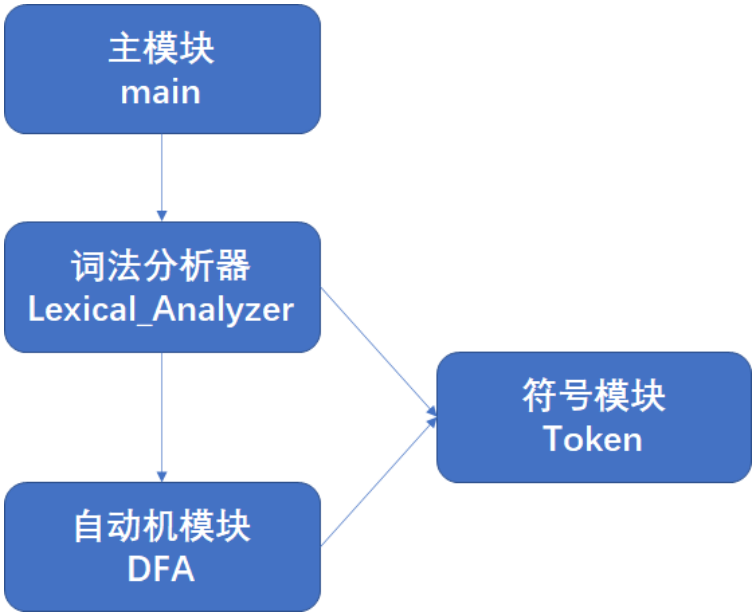

S
60
T



四、软件设计

4.1 总体框架

本词法分析软件分为四个模块，分别为：主模块、词法分析模块、符号模块、自动机模块，相互调用关系如下：



主模块负责代码文件的读入、调用词法分析器

词法分析器负责运行词法分析程序，将代码字符输入自动机，对自动机返回值进行分析，并将结果存入符号表中

自动机模块接收词法分析器输入的字符，进行状态转移，并输出识别结果

符号模块存储所有可能出现的符号类型与错误类型

4.2 数据结构

4.2.1 符号枚举结构

用一个emun枚举结构存储所有的符号类型

```
enum TokenType
{
    /*初始类型*/
    UNKNOWN = 0,

    /*跳过类型*/
    IGNORE,

    /*关键字*/
    CHAR,
    INT,
    LONG,
    FLOAT,
```

```
    DOUBLE,
    VOID,
    UNSIGNED,
    SIGNED,
    CONST,
    STATIC,
    EXTERN,
    STRUCT,
    UNION,
    TYPEDEF,
    SIZEOF,
    IF,
    ELSE,
    DO,
    WHILE,
    FOR,
    SWITCH,
    CASE,
    DEFAULT,
    CONTINUE,
    BREAK,
    GOTO,
    RETURN,

    /*运算符号*/
    ASSIGN,
    ADD,
    SELF_INCREASE,
    ADD_ASSIGN,
    SUB,
    SELF_DECREASE,
    DEC,
    SUB_ASSIGN,
    MUL_ASSIGN,
    DIV,
    DIV_ASSIGN,
    MOD,
    MOD_ASSIGN,
    BITWISE_AND,
    BITWISE_AND_ASSIGN,
    BITWISE_OR,
    BITWISE_OR_ASSIGN,
    BITWISE_XOR,
    BITWISE_XOR_ASSIGN,
    BITWISE_NOT,
    SHL,
    SHL_ASSIGN,
    SHR,
    SHR_ASSIGN,
    AND,
    AND_ASSIGN,
    OR,
    OR_ASSIGN,
    NOT,
    LESS,
    LESS_OR_EQUAL,
    EQUAL,
    NOT_EQUAL,
```

```
    INEUQUAL,
    GREATER,
    GREATER_OR_EQUAL,
    CONNECT,
    ASTERISK,
    AMPERSAND,
    QUESTION,
    COMMA,
    COLON,
    SEMICOLON,
    DOT,
    STAR,
    ARROW,
    LEFT_PARENTHESIS,
    RIGHT_PARENTHESIS,
    LEFT_BRACKET,
    RIGHT_BRACKET,
    LEFT_BRACE,
    RIGHT_BRACE,

    /*行分隔符*/
    LINE_BREAK,

    /*注释、标识符与常量类型*/
    LINE_COMMENT,
    BLOCK_COMMENT,
    IDENTIFIER,
    CONSTANT_INT,
    CONSTANT_REAL,
    CONSTANT_SCIENTIFIC,
    CONSTANT_CHAR,
    CONSTANT_STRING,

    /*错误类型*/
    INCOMPLETE_NUMBERIC_ERROR,
    UNCLOSED_BLOCK_COMMENT_ERROR,
    UNCLOSED_CHAR_ERROR,
    ESCAPE_CHARACTER_NOT_EXIST,
    TOO_MUCH_CHARACTER,
    UNFINISHED_STRING_ERROR,
    ILLEGAL_CHAR_ERROR,
};
```

4.2.2 关键词识别Map

在自动机输出识别结果为标识符后，还需要进一步识别是否是关键词。在这里我为每一种C关键词生成了一个同名的`TokenType`类型，并使用一个Map结构存储所有的`<string, TokenType>`键值对，通过判断自动机返回的当前标识符是否在其中存在，可以判断它是否为一个关键词，并得到具体的符号类型。

```
inline map<string, TokenType> ReservedWord = {
    {"char", CHAR},
    {"int", INT},
    {"long", LONG},
    {"float", FLOAT},
    {"double", DOUBLE},
    {"void", VOID},
```

```
    {"unsigned", UNSIGNED},
    {"signed", SIGNED},
    {"const", CONST},
    {"static", STATIC},
    {"extern", EXTERN},
    {"struct", STRUCT},
    {"union", UNION},
    {"typedef", TYPEDEF},
    {"sizeof", SIZEOF},
    {"if", IF},
    {"else", ELSE},
    {"do", DO},
    {"while", WHILE},
    {"for", FOR},
    {"switch", SWITCH},
    {"case", CASE},
    {"default", DEFAULT},
    {"continue", CONTINUE},
    {"break", BREAK},
    {"goto", GOTO},
    {"return", RETURN}
};
```

4.2.3 符号输出映射Map

通过`<TokenType, string>`键值对将符号表中存储的符号类型TokenType转为同名字符串，用于符号表输出。

```
inline map<TokenType, string> TypeName = {
    {UNKNOWN, "UNKNOWN"},
    {IGNORE, "IGNORE"},
    {CHAR, "CHAR"},
    {INT, "INT"},
    ……
}
```

4.2.4 符号表类

符号表中每个元素Token包含：**符号类型TokenType**， **行号linePos**， **列号colPos**，及**属性val**

```
class Token
{
public:
    Token();
    Token(const int, const int);
    Token(const int, const int, const TokenType);
    ~Token();

    TokenType type;
    int linePos, colPos;
    string val;
};
```

4.2.5 Mealy自动机类

```
struct Mealy
{
    int next;
    TokenType output;
    int trace_back;
    void operator=(const int _next) { next = _next; output = UNKNOWN; trace_back
= 0; };
    Mealy() : next(0), output(UNKNOWN) {};
    Mealy(const int _next, const TokenType _output = UNKNOWN, const int
_trace_back = 0) :
        next(_next), output(_output), trace_back(_trace_back) {};
};
```

用`Mealy mealy[MAXMINE_STATE][MAXMINE_CHAR]`，`MAXMINE_STATE`包含自动机的所有状态，`MAXMINE_CHAR`包含所有字符，存储自动机所有的转移路径与输出，存储在每个状态下输入不同的字符对应的输出。

`next`代表自动机在当前状态、当前输入下的下一状态

`output`代表自动机当前的输出，在未识别完毕时输出`UNKNOWN`中间态，在识别完毕时。输出对应的识别结果，在识别到错误时输出对应的错误信息。

`trace_back`代表当前识别指针需要回退的字符数。

在错误信息的处理上，输出错误信息时，自动将自动机转移到终态，保证自动机能在出现错误时自动恢复，继续运行。

同时，为了在识别完一个符号后可以进入下一个符号的识别，在终态与初态间添加一个ε转移，保证自动机在进入终态后自动回到起点。

自动机有两种赋值方式：

1. 对于中间态使用重载`=`，如`Mealy[STATE_S]['[']] = 23`，此时将23赋给`next`，`trace_back`与`output`赋0，因为中间态不产生输出与回退
2. 对于终态使用完整结构体赋值，如`mealy[58]['/'] = Mealy(STATE_T, BLOCK_COMMENT, 0);`

4.2.6 符号与错误记录表

```
Token symbol_table[MAX_TOKEN];  //记录符号表
Token error_symbol[MAX_TOKEN];  //记录错误信息
TokenCount token_number[200];   //记录各类符号数量
```

4.3 词法分析器模块

```
/*初始化词法分析器*/
Lexical_Analyzer::Lexical_Analyzer(const string _code)
{
    code = _code;
    code += char(1);        //在读入代码尾部加入不可能在代码出出现的char(1)字符，用来表示
EOF
    pointer = 0;
    tot_len = code.length();
    line = 1;
    col = 1;
    tot_error = 0;
```

```
    tot_token = 1;
    symbol_table[tot_token] = Token(1, 1);
}

/*执行词法分析器*/
void Lexical_Analyzer::Run()
{
    while (pointer < tot_len)       //若当前指针没有指向末尾，则可以进入分析
    {
        int token_fin = 0;
        Mealy out = dfa.Input(code[pointer]);       //向Mealy自动机输入字符，得到输出
        TokenType tokentype = out.output;

        if (tokentype != UNKNOWN)       //如果当前自动机输出不是UNKNOWN，说明达到终态或
错误
        {
            if (tokentype == IGNORE) {  //如果当前自动机输出是IGNORE，不记录并跳转到下
一个词
                symbol_table[tot_token] = Token(line, col + 1);
            }
            else {
                switch (tokentype)
                {
                case LINE_BREAK:        //如果当前自动机输出是LINE_BREAK，将行号加1，
列号清零
                    line++;
                    col = 0;
                    symbol_table[tot_token] = Token(line, col + 1);
                    break;
                case IDENTIFIER:        //如果当前自动机输出是IDENTIFIER，则判断其是否
是关键字
                    if (ReservedWord.find(symbol_table[tot_token].val) ==
ReservedWord.end())
                    {
                        symbol_table[tot_token].type = tokentype;
                    }
                    else
                        symbol_table[tot_token].type =
ReservedWord[symbol_table[tot_token].val];
                    token_fin = 1;      //打上标记、在末尾处理
                    break;
                case INCOMPLETE_NUMBERIC_ERROR:
                case UNCLOSED_BLOCK_COMMENT_ERROR:
                case UNCLOSED_CHAR_ERROR:
                case ESCAPE_CHARACTER_NOT_EXIST:
                case TOO_MUCH_CHARACTER:
                case UNFINISHED_STRING_ERROR:   //如果当前是错误
                    error_symbol[++tot_error] =
Token(symbol_table[tot_token].linePos,

 symbol_table[tot_token].colPos,
                                                        tokentype);   //存入错误表
                    if (code[pointer] == '\n')  //如果错误结尾是换行符，也要将行号加1，
列号清零
                    {
                        line++;
                        col = 0;
                    }
```

```
                    symbol_table[tot_token] = Token(line, col + 1);
                    break;
                default:                        //非错误、非标识符的其他终态输出，存入符号表
即可
                    symbol_table[tot_token].type = tokentype;
                    token_fin = 1;
                    break;
                }

            }
        }
        else
        {
            symbol_table[tot_token].val += code[pointer];
        }
        pointer++;
        col++;
        pointer -= out.trace_back;       //指针回溯
        col -= out.trace_back;           //列数回溯
        if (token_fin == 1)          //如果当前刚存下一个符号
        {
            tot_token++;             //将当前行、列保存，这将是下一个符号的起使位置
            symbol_table[tot_token] = Token(line, col);
        }
    }
    tot_token--;
    tot_len--;
    line--;
}

void Lexical_Analyzer::Print_Table()
{
    cout << "统计结果: "
        << "共 " << line << " 行 , "
        << "共 " << tot_error << " 个错误 , "
        << "共 " << tot_token << " 个单词" << endl
        << "统计结果如下:" << endl ;

    if (tot_error > 0)
    {
        cout << endl << "错误:" << endl;
        for (int i = 1; i <= tot_error; i++)
        {
            cout << TypeName[error_symbol[i].type] << " in "
                << "( " << error_symbol[i].linePos << " , "
                << error_symbol[i].colPos << " )"
                << endl;
        }
    }



    cout << endl << "符号表:" << endl;
    for (TokenType i = UNKNOWN; i <= ILLEGAL_CHAR_ERROR; i = TokenType(i + 1))
    {
        token_number[i].type = i;
        token_number[i].cnt = 0;
    }
```

```
    for (int i = 1; i <= tot_token; i++)
        token_number[symbol_table[i].type].cnt++;
    sort(token_number, token_number + 199);
    for (int i = 0; i < 200 && token_number[i].cnt > 0; i++)
        cout << TypeName[token_number[i].type] << " : " << token_number[i].cnt
<< endl;
    cout << "符号表为:" << endl;
    for (int i = 1; i <= tot_token; i++)
    {
        cout    << "<"
            << TypeName[symbol_table[i].type] << " , "
            << '"' << symbol_table[i].val << "\" > in "
            << "( " << symbol_table[i].linePos << " , "
            << symbol_table[i].colPos << " )"
            << endl;
    }
}
```

4.4 Mealy自动机模块

```
/*进行状态转移路径的初始化*/
DFA::DFA()
{
    for (int i = 0; i < MAXMINE_STATE; i++)
        FOR_OTHER(j) mealy[i][j] = Mealy(STATE_T, UNKNOWN, 0);

    /*Begin with letter*/
    FOR_LETTER(j)
        mealy[STATE_S][j] = 1;
    TO_END(1, IDENTIFIER);
    FOR_LETTER(j)
        mealy[1][j] = 1;
    FOR_DIGITAL(j)
        mealy[1][j] = 1;

    /*begin with digital*/
    FOR_DIGITAL(j)
        mealy[STATE_S][j] = 2;

    TO_END(2, CONSTANT_INT);
    FOR_DIGITAL(j)
        mealy[2][j] = 2;
    mealy[2]['.'] = 3;
    mealy[2]['e'] = 5;
    mealy[2]['E'] = 5;

    TO_END(3, INCOMPLETE_NUMBERIC_ERROR);
    FOR_DIGITAL(j)
        mealy[3][j] = 4;

    TO_END(4, CONSTANT_REAL)
    FOR_DIGITAL(j)
        mealy[4][j] = 4;
    mealy[4]['e'] = 5;
    mealy[4]['E'] = 5;
```

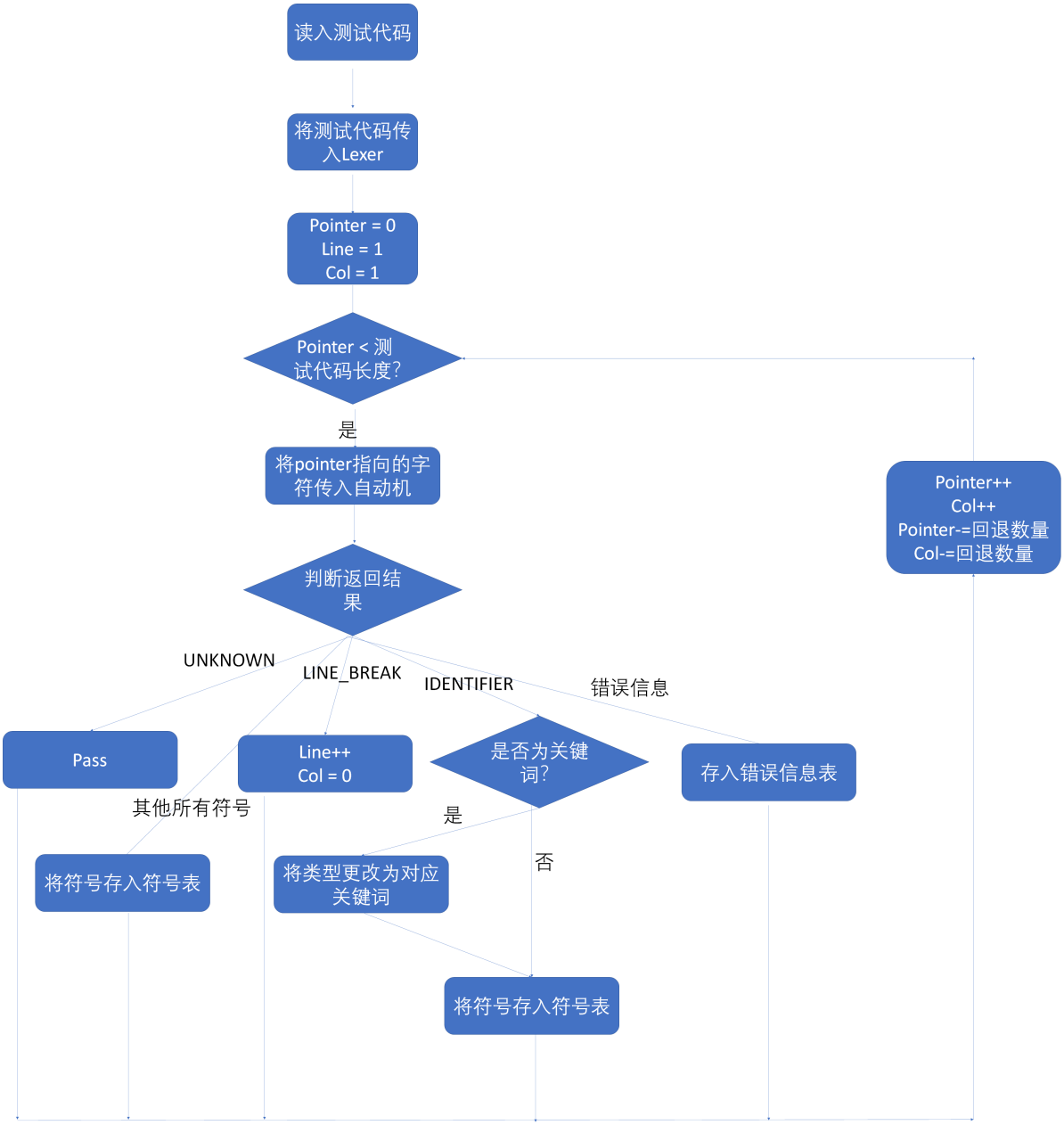
```
    TO_END(5, INCOMPLETE_NUMBERIC_ERROR);
    FOR_DIGITAL(j)
        mealy[5][j] = 6;
    mealy[5]['+'] = 7;
    mealy[5]['-'] = 7;

    TO_END(6, CONSTANT_SCIENTIFIC);
    FOR_DIGITAL(j)
        mealy[6][j] = 6;

    FOR_DIGITAL(j)
        mealy[7][j] = 6;

    /*begin with signal*/
    mealy[STATE_S]['~'] = 8;
    TO_END(8, BITWISE_NOT);

    mealy[STATE_S]['!'] = 9;
    TO_END(9, NOT);
    mealy[9]['='] = 10;
    TO_END(10, NOT_EQUAL);

    mealy[STATE_S]['#'] = 11;
    FOR_OTHER(j)
        mealy[11]['#'] = 11;
    mealy[11]['\n'] = Mealy(STATE_T, IGNORE, 1);
    mealy[11]['#'] = 27;
    TO_END(27, CONNECT);

    mealy[STATE_S]['%'] = 12;
    TO_END(12, MOD);
    mealy[12]['='] = 13;
    TO_END(13, MOD_ASSIGN);

    mealy[STATE_S]['&'] = 16;
    TO_END(16, BITWISE_AND);
    mealy[16]['='] = 17;
    mealy[16]['&'] = 18;
    TO_END(17, BITWISE_AND_ASSIGN);
    TO_END(18, AND);
    mealy[18]['='] = 19;
    TO_END(19, BITWISE_AND)

    mealy[STATE_S]['*'] = 20;
    TO_END(20, STAR);

    mealy[STATE_S]['('] = 21;
    TO_END(21, LEFT_PARENTHESIS);

    mealy[STATE_S][')'] = 22;
    TO_END(22, RIGHT_PARENTHESIS);

    mealy[STATE_S]['-'] = 23;
    TO_END(23, SUB);
    mealy[23]['-'] = 24;
    mealy[23]['='] = 25;
    mealy[23]['>'] = 26;
    TO_END(24, SELF_DECREASE);
```

```
    TO_END(25, SUB_ASSIGN);
    TO_END(26, ARROW);

    mealy[STATE_S]['+'] = 28;
    TO_END(28, ADD);
    mealy[28]['+'] = 29;
    mealy[28]['='] = 30;
    TO_END(29, SELF_INCREASE);
    TO_END(30, ADD_ASSIGN);

    mealy[STATE_S]['='] = 31;
    TO_END(31, ASSIGN);
    mealy[31]['='] = 32;
    TO_END(32, EQUAL);

    mealy[STATE_S]['['] = 33;
    TO_END(33, LEFT_BRACKET);
    mealy[STATE_S][']'] = 34;
    TO_END(34, RIGHT_BRACKET);

    mealy[STATE_S]['{'] = 35;
    TO_END(35, LEFT_BRACE);
    mealy[STATE_S]['}'] = 36;
    TO_END(36, RIGHT_BRACE);

    mealy[STATE_S]['|'] = 37;
    TO_END(37, BITWISE_OR);
    mealy[37]['|'] = 38;
    mealy[37]['='] = 39;
    TO_END(38, OR);
    TO_END(39, BITWISE_OR_ASSIGN);

    mealy[STATE_S][';'] = 40;
    TO_END(40, SEMICOLON);

    mealy[STATE_S][':'] = 41;
    TO_END(41, COLON);

    mealy[STATE_S]['\''] = 42;
    FOR_OTHER(j)
        mealy[42][j] = 45;
    mealy[42]['\\'] = 43;
    mealy[42]['\n'] = Mealy(STATE_T, UNCLOSED_CHAR_ERROR, 0);
    mealy[42][1] = Mealy(STATE_T, UNCLOSED_CHAR_ERROR, 0);
    FOR_OTHER(j)
        mealy[43][j] = 63;
    mealy[43]['0'] = 44;
    mealy[43]['n'] = 44;
    mealy[43]['r'] = 44;
    mealy[43]['t'] = 44;
    mealy[43]['v'] = 44;
    mealy[43]['a'] = 44;
    mealy[43]['b'] = 44;
    mealy[43]['f'] = 44;
    mealy[43]['\''] = 44;
    mealy[43]['\"'] = 44;
    mealy[43]['\\'] = 44;
    FOR_OTHER(j)
```

```
        mealy[44][j] = 64;
    mealy[44]['\''] = Mealy(STATE_T, CONSTANT_CHAR, 0);
    FOR_OTHER(j)
        mealy[45][j] = 64;
    mealy[45]['\''] = Mealy(STATE_T, CONSTANT_CHAR, 0);
    FOR_OTHER(j)
        mealy[63][j] = 64;
    mealy[63]['\''] = Mealy(STATE_T, ESCAPE_CHARACTER_NOT_EXIST, 0);
    FOR_OTHER(j)
        mealy[64][j] = 64;
    mealy[64]['\''] = Mealy(STATE_T, TOO_MUCH_CHARACTER, 0);
    mealy[64]['\n'] = Mealy(STATE_T, UNCLOSED_CHAR_ERROR, 0);
    mealy[64][1] = Mealy(STATE_T, UNCLOSED_CHAR_ERROR, 0);

    mealy[STATE_S]['\"'] = 46;
    FOR_OTHER(j)
        mealy[46][j] = 46;
    mealy[46]['\n'] = Mealy(STATE_T, UNFINISHED_STRING_ERROR, 0);
    mealy[46][1] = Mealy(STATE_T, UNFINISHED_STRING_ERROR, 0);
    mealy[46]['\"'] = Mealy(STATE_T, CONSTANT_STRING, 0);

    mealy[STATE_S]['<'] = 47;
    TO_END(47, LESS);
    mealy[47]['='] = 48;
    mealy[47]['<'] = 49;
    TO_END(48, LESS_OR_EQUAL);
    TO_END(49, SHL);
    mealy[49]['='] = 50;
    TO_END(50, SHL_ASSIGN);

    mealy[STATE_S]['>'] = 51;
    TO_END(51, GREATER);
    mealy[51]['='] = 52;
    mealy[51]['>'] = 53;
    TO_END(52, GREATER_OR_EQUAL);
    TO_END(53, SHR);
    mealy[53]['='] = 54;
    TO_END(54, SHR_ASSIGN);

    mealy[STATE_S]['/'] = 55;
    TO_END(55, DIV);
    mealy[55]['*'] = 57;
    mealy[55]['/'] = 56;
    mealy[55]['='] = 59;
    FOR_OTHER(j)
        mealy[56][j] = 56;
    mealy[56]['\n'] = Mealy(STATE_T, LINE_COMMENT, 0);
    FOR_OTHER(j)
        mealy[57][j] = 57;
    mealy[57]['*'] = 58;
    mealy[57][1] = Mealy(STATE_T, UNCLOSED_BLOCK_COMMENT_ERROR, 0);
    FOR_OTHER(j)
        mealy[58][j] = 57;
    mealy[58]['*'] = 58;
    mealy[58]['/'] = Mealy(STATE_T, BLOCK_COMMENT, 0);
    mealy[58][1] = Mealy(STATE_T, UNCLOSED_BLOCK_COMMENT_ERROR, 0);
    TO_END(59, DIV_ASSIGN);
```

```
    mealy[STATE_S][','] = 60;
    TO_END(60, COMMA);

    mealy[STATE_S]['.'] = 61;
    TO_END(61, DOT);

    mealy[STATE_S]['?'] = 62;
    TO_END(62, QUESTION);

    mealy[STATE_S]['\n'] = Mealy(STATE_T, LINE_BREAK);
    mealy[STATE_S][' '] = Mealy(STATE_T, IGNORE);
    mealy[STATE_S][1] = Mealy(STATE_T, IGNORE);
    mealy[STATE_S]['$'] = Mealy(STATE_T, ILLEGAL_CHAR_ERROR);
    mealy[STATE_S]['@'] = Mealy(STATE_T, ILLEGAL_CHAR_ERROR);
    state = STATE_S;
}

/*自动机的运行：输入为当前匹配字符，输出为自动机输出*/
Mealy DFA::Input(const char c)
{
    if (!(c >= 0 && c < 128))       //如果当前输入符号不是一个ASCII字符，则直接返回错
误，避免将其放入数组后越界
    {
        return Mealy(0, ILLEGAL_CHAR_ERROR, 0);
    }
    Mealy out = mealy[state][c];    //如果当前输入符号是一个ASCII字符，直接将其放入数
组，得到新的state和当前输出
    state = mealy[state][c].next;
    if (state == STATE_T)           //如果当前到达终态，自动转移到起始位置，便于下一个单
词的识别
        state = STATE_S;
    return out;
}
```

4.5 运行流程



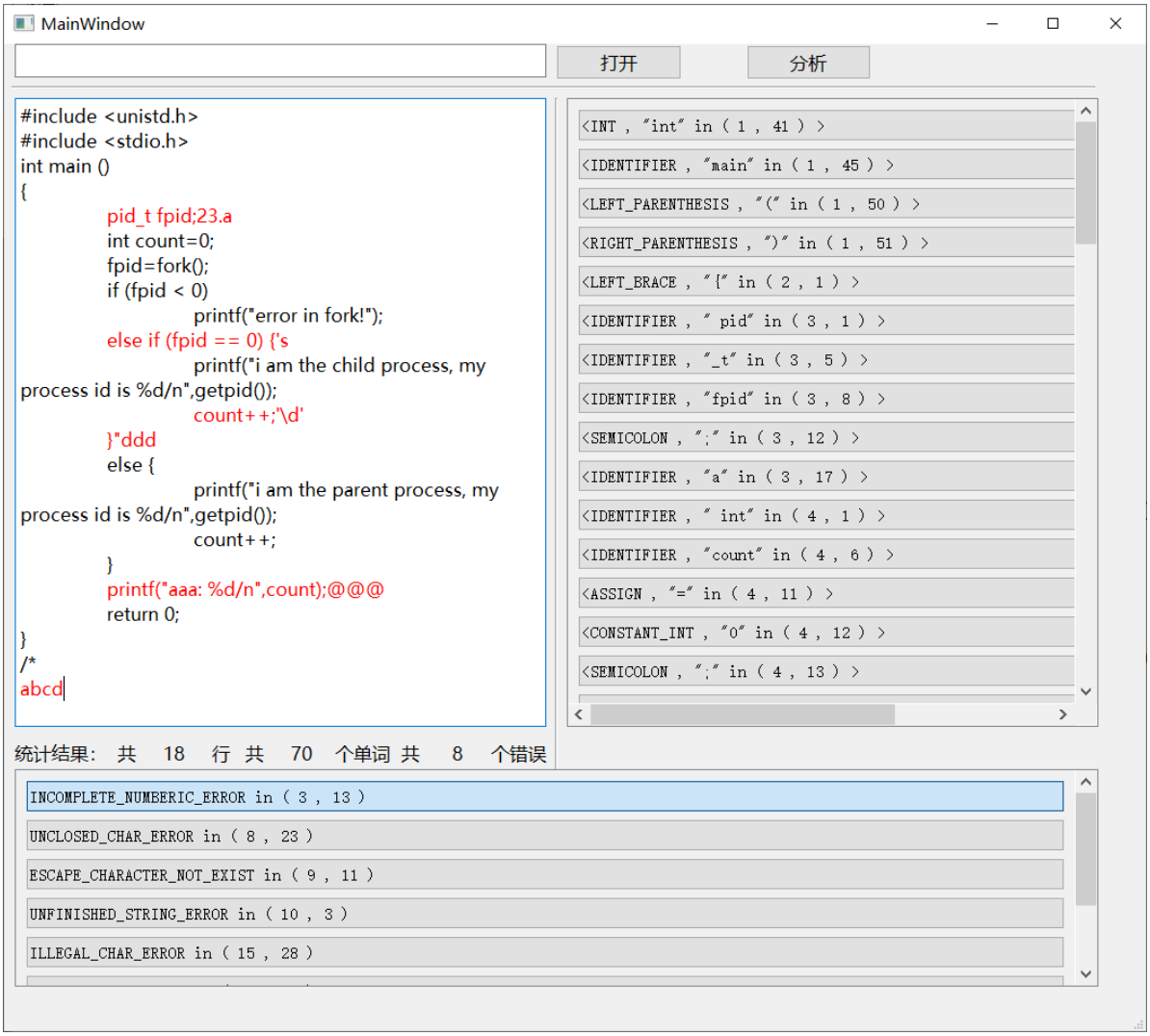
五、软件测试

测试代码

```
#include <unistd.h>
#include <stdio.h>
int main ()
{
    pid_t fpid;23.a
    int count=0;
    fpid=fork();
    if (fpid < 0)
        printf("error in fork!");
    else if (fpid == 0) {'s
        printf("i am the child process, my process id is %d/n",getpid());
        count++;'\d'
    }"ddd
    else {
        printf("i am the parent process, my process id is %d/n",getpid());
        count++;
    }
```

```
    printf("aaa: %d/n",count);@@@
    return 0;
}
/*
abcd
```

测试结果



共有18行代码，70个单词，8个错误。（其中错误所在行被标红）

其中错误分别为：

位于（3，13）的不完整数字错误

位于（8，23）的未闭合字符错误

位于（9，11）的非法转义符错误

位于（10，3）的未闭合字符串错误

位于（15，28），（15，29），（15，30）的非法字符串错误

位于（18，1）的未闭合块注释错误

不同类别的符号统计结果：

```
"IDENTIFIER"  :  22
"SEMICOLON"  :  9
"RIGHT_PARENTHESIS"  :  8
"LEFT_PARENTHESIS"  :  8
```

```
"CONSTANT_INT"  :  4
"CONSTANT_STRING"  :  3
"LEFT_BRACE"  :  3
"RIGHT_BRACE"  :  3
"SELF_INCREASE"  :  2
"ASSIGN"  :  2
"COMMA"  :  2
"INT"  :  1
"IF"  :  1
"LESS"  :  1
"EQUAL"  :  1
```

符号表:

```
<INT , "int" > in ( 1 , 41 )
<IDENTIFIER , "main" > in ( 1 , 45 )
<LEFT_PARENTHESIS , "(" > in ( 1 , 50 )
<RIGHT_PARENTHESIS , ")" > in ( 1 , 51 )
<LEFT_BRACE , "{" > in ( 2 , 1 )
<IDENTIFIER , " pid" > in ( 3 , 1 )
<IDENTIFIER , "_t" > in ( 3 , 5 )
<IDENTIFIER , "fpid" > in ( 3 , 8 )
<SEMICOLON , ";" > in ( 3 , 12 )
<IDENTIFIER , "a" > in ( 3 , 17 )
<IDENTIFIER , " int" > in ( 4 , 1 )
<IDENTIFIER , "count" > in ( 4 , 6 )
<ASSIGN , "=" > in ( 4 , 11 )
<CONSTANT_INT , "0" > in ( 4 , 12 )
<SEMICOLON , ";" > in ( 4 , 13 )
<IDENTIFIER , " fpid" > in ( 5 , 1 )
<ASSIGN , "=" > in ( 5 , 6 )
<IDENTIFIER , "fork" > in ( 5 , 7 )
<LEFT_PARENTHESIS , "(" > in ( 5 , 11 )
<RIGHT_PARENTHESIS , ")" > in ( 5 , 12 )
<SEMICOLON , ";" > in ( 5 , 13 )
<IDENTIFIER , " if" > in ( 6 , 1 )
<LEFT_PARENTHESIS , "(" > in ( 6 , 5 )
<IDENTIFIER , "fpid" > in ( 6 , 6 )
<LESS , "<" > in ( 6 , 11 )
<CONSTANT_INT , "0" > in ( 6 , 13 )
<RIGHT_PARENTHESIS , ")" > in ( 6 , 14 )
<IDENTIFIER , "     printf" > in ( 7 , 1 )
<LEFT_PARENTHESIS , "(" > in ( 7 , 9 )
<CONSTANT_STRING , ""error in fork!" > in ( 7 , 10 )
<RIGHT_PARENTHESIS , ")" > in ( 7 , 26 )
<SEMICOLON , ";" > in ( 7 , 27 )
<IDENTIFIER , " else" > in ( 8 , 1 )
<IF , "if" > in ( 8 , 7 )
<LEFT_PARENTHESIS , "(" > in ( 8 , 10 )
<IDENTIFIER , "fpid" > in ( 8 , 11 )
<EQUAL , "==" > in ( 8 , 16 )
<CONSTANT_INT , "0" > in ( 8 , 19 )
<RIGHT_PARENTHESIS , ")" > in ( 8 , 20 )
<LEFT_BRACE , "{" > in ( 8 , 22 )
<IDENTIFIER , "     count" > in ( 9 , 1 )
<SELF_INCREASE , "++" > in ( 9 , 8 )
<SEMICOLON , ";" > in ( 9 , 10 )
```

```
<RIGHT_BRACE , "    }" > in ( 10 , 1 )
<IDENTIFIER , " else" > in ( 11 , 1 )
<LEFT_BRACE , "{" > in ( 11 , 7 )
<IDENTIFIER , "     printf" > in ( 12 , 1 )
<LEFT_PARENTHESIS , "(" > in ( 12 , 9 )
<CONSTANT_STRING , ""i am the parent process, my process id is %d/n" > in ( 12 ,
10 )
<COMMA , "," > in ( 12 , 58 )
<IDENTIFIER , "getpid" > in ( 12 , 59 )
<LEFT_PARENTHESIS , "(" > in ( 12 , 65 )
<RIGHT_PARENTHESIS , ")" > in ( 12 , 66 )
<RIGHT_PARENTHESIS , ")" > in ( 12 , 67 )
<SEMICOLON , ";" > in ( 12 , 68 )
<IDENTIFIER , "     count" > in ( 13 , 1 )
<SELF_INCREASE , "++" > in ( 13 , 8 )
<SEMICOLON , ";" > in ( 13 , 10 )
<RIGHT_BRACE , "    }" > in ( 14 , 1 )
<IDENTIFIER , " printf" > in ( 15 , 1 )
<LEFT_PARENTHESIS , "(" > in ( 15 , 8 )
<CONSTANT_STRING , ""aaa: %d/n" > in ( 15 , 9 )
<COMMA , "," > in ( 15 , 20 )
<IDENTIFIER , "count" > in ( 15 , 21 )
<RIGHT_PARENTHESIS , ")" > in ( 15 , 26 )
<SEMICOLON , ";" > in ( 15 , 27 )
<IDENTIFIER , " return" > in ( 16 , 1 )
<CONSTANT_INT , "0" > in ( 16 , 9 )
<SEMICOLON , ";" > in ( 16 , 10 )
<RIGHT_BRACE , "}" > in ( 17 , 1 )
```