《编译原理与设计》

词法分析程序的设计与实现

实验报告

班级 2014211304
学号 2014210336
姓名 杨炫越
日期 2016.10.21

1. 实验内容

设计并实现 C 语言的词法分析程序，要求如下。

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) 可以统计源程序汇总的语句行数、单词个数和字符个数，并输出统计结果。
4) 检查源程序中存在的错误，并可以报告错误所在的位置。
5) 发现源程序中存在的错误后，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，即可检查并报告出源程序中存在的词法错误。

2. 实验要求

采用 C/C++作为实现语言，手工编写词法分析程序。

3. 设计原理

本词法分析程序主体为一个表驱动的 DFA，通过其来识别 C 语言代码中的各类 token，并划分出来生成 token 流以便于后续语法分析，如下图所示：

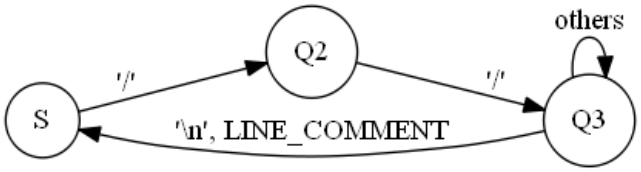
本程序中 token 的分类参考了 Clang 的词法分析结果，各 token 的识别方式如下：

3.1. 预编译命令 PREPROCESSOR

C 语言的预编译命令为以’#’开头的特定命令，因对此部分的处理是预处理器的职责，不需由词法分析承担，故该程序简单地将’#’后带任意单行字符串的符号视为预编译命令。其识别单行预编译命令的 DFA 如下：

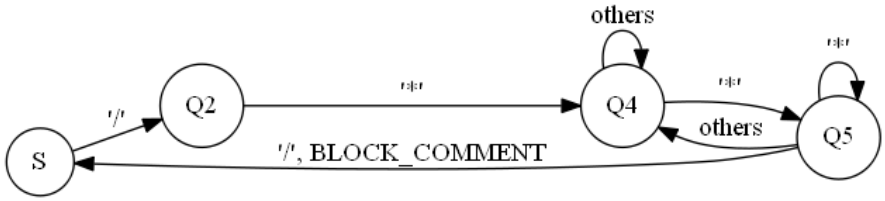
3.2. 行注释 LINE_COMMENT

C 语言的行注释为以”//”开头的任意单行字符串。识别行注释的 DFA 如下：



3.3. 块注释 BLOCK_COMMENT

C 语言的块注释为以”/*”与”*/”括起的任意多行字符串。识别块注释的 DFA 如下：



3.4. 运算符与标点

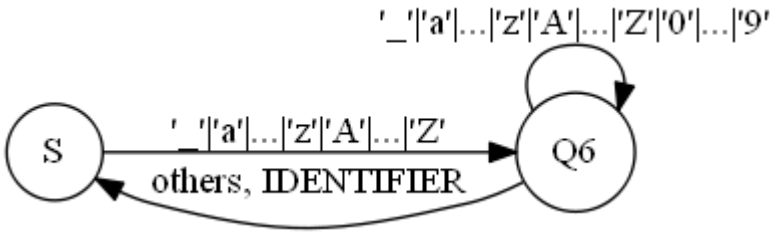
运算符与标点分类如下，因识别过程较为简单，各 DFA 不细述。其中代表确定运算的符号的以运算名为类型名（如’^’指按位异或、’%’指模），而有多义的运算符（如’*’可表示指针或乘法）或纯标点（如分号’;’）以符号名为类型名。

名称	类型名	符号
赋值	ASSIGN	=
加	ADD	+
自增	INC	++
加赋值	ADD_ASSIGN	+=
减	SUB	-
自减	DEC	--
乘赋值	MUL_ASSIGN	*=
除	DIV	/
模	MOD	%
模赋值	MOD_ASSIGN	%=
按位与赋值	BITWISE_AND_ASSIGN	&=

按位或	BITWISE_OR	|
按位或赋值	BITWISE_OR_ASSIGN	|=
按位异或	BITWISE_XOR	^
按位异或赋值	BITWISE_XOR_ASSIGN	^=
按位取反	BITWISE_NOT	~
与	AND	&&
与赋值	AND_ASSIGN	&&=
或	OR	||
或赋值	OR_ASSIGN	||=
非	NOT	!
左移	SHL	<<
左移赋值	SHL_ASSIGN	<<=
右移	SHR	>>
右移赋值	SHR_ASSIGN	>>=
小于	LESS	<
小于等于	LESS_EQUAL	<=
等于	EQUAL	==
不等	INEQUAL	!=
大于	GREATER	>
大于等于	GREATER_EQUAL	>=
拼接	CONCAT	##
星号	ASTERISK	*
&号	AMPERSAND	&
问号	QUESTION	?
逗号	COMMA	,
冒号	COLON	:
分号	SEMICOLON	;
点号	DOT	.
箭头	ARROW	->
左大括号	L_BRACE	{
右大括号	R_BRACE	}
左中括号	L_SQUARE	[
右中括号	R_SQUARE	]
左小括号	L_PAREN	(
右小括号	R_PAREN	)

3.5. 标识符 IDENTIFIER

C 语言标识符可由下划线'_'、字母与数字组成，开头不能为数字。识别标识符的 DFA 如下：



3.6. 保留字

C 语言保留字如下：

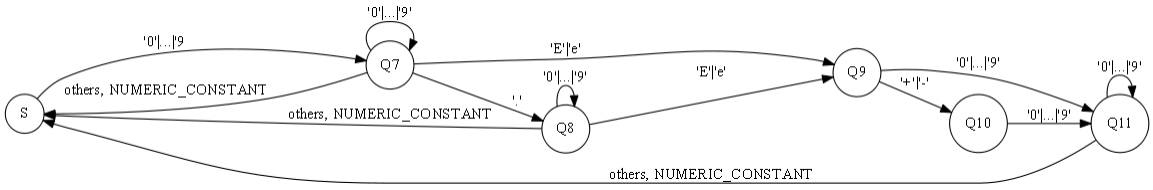
char	unsigned	union
int	signer	typedef
long	const	sizeof
float	static	if
double	extern	else
void	struck	

因保留字都是合法标识符，故本程序识别的保留字的方法是对每一个已识别的标识符，都查询其是否在保留字表中，若是则识别为一个保留字。为**加速查表过程**，本程序使用了 **Trie 树**这一数据结构来存取保留字表，其本质也为识别一系列给定字符串的 DFA，从而使得查表的时间复杂度从$\Theta(nl)$优化为$\Theta(l)$，其中 n 为保留字个数，l 为保留字平均长度。

参考了 Clang 的做法，本程序对每个关键字都以其自身作为其类型名称，如"char"是一个 `CHAR` 类型的 token，"return"是一个 `RETURN` 类型的 token，等等。这样处理使得各保留字的词法意义更为具体，避免了保留字这一笼统的概念，使得不同功能的保留字得以分开，有利于后续的语法分析。

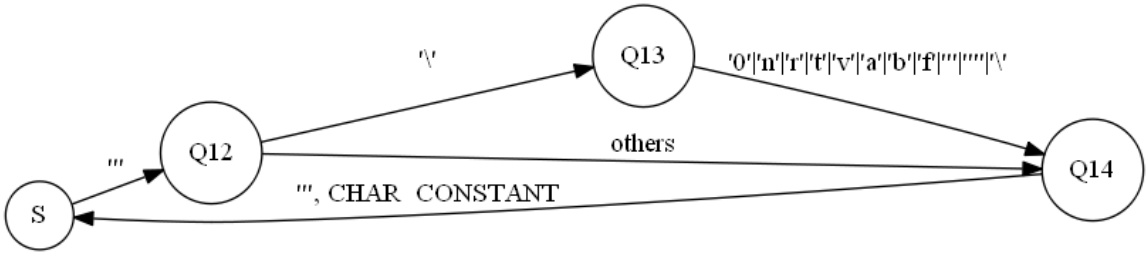
3.7. 数字常量 `NUMERIC_CONSTANT`

C 语言的数字常量可为整数或小数，后可接指数后缀 Ex 或 ex 表示$\times 10^x$。特殊的，小数点后可为空，代表".0"。识别数字常量的 DFA 如下：



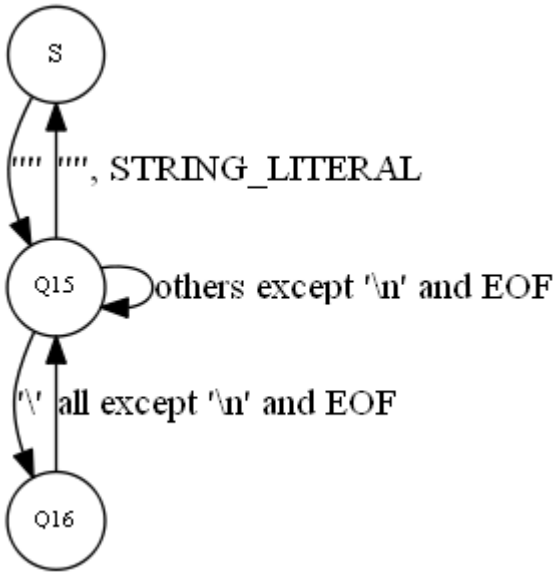
3.8. 字符常量 CHAR_CONSTANT

C 语言的字符常量除了以单括号括起的单个字符外，还有转义字符'\0'、'\n'、'\r'、'\t'、'\v'、'\a'、'\b'、'\f'、'\'、''\""、'\\'。识别字符常量的 DFA 如下：



3.9. 字符串字面量 STRING_LITERRAL

C 语言的字符串字面量为以双引号括起的任意字符串，但若字符串中含有转义字符'\"'，则其中的'""'不能作为字符串结尾标志。识别字符常量的 DFA 如下：

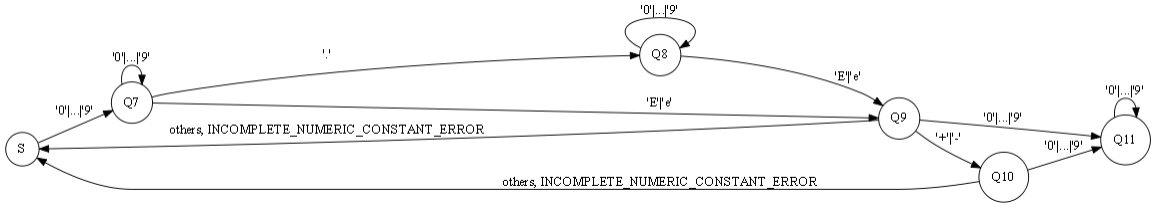


3.10. 错误类型

词法分析阶段能识别出的错误很有限。主流编译器如GCC、Clang的词法分析结果对于错误一般是以 unknown 标识，本程序尝试对错误类型进行适当的细化，保证能以纯词法分析的手段检测得到而不至于越界。归结出的几种错误类型如下：

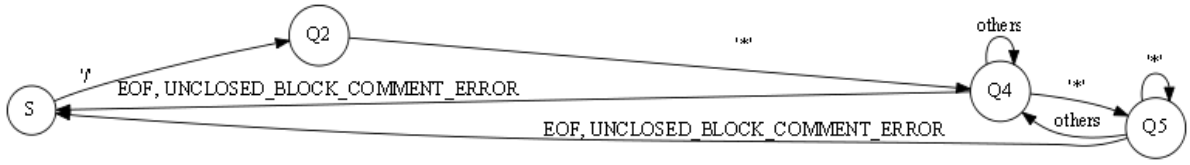
3.10.1. 不完整数字常量错误 INCOMPLETE_NUMERIC_CONMSTANT_ERROR

本程序将带指数后缀 E*x* 或 e*x* 但整数 *x* 的内容不完整，即为空或只有单个正负号的数字常量视为不完整数字常量。识别不完整数字常量错误的 DFA 如下：



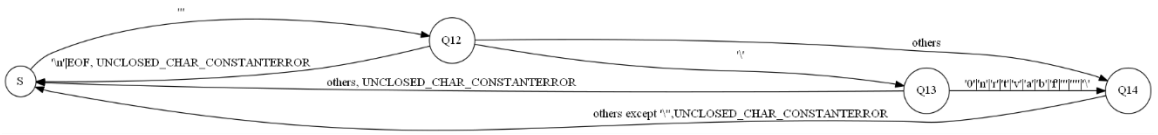
3.10.2. 未闭合块注释错误 UNCLOSED_BLOCK_COMMENT_ERROR

本程序将已读取到文件末尾但仍未以完整”*/”终结的块注释视为未闭合块注释错误。识别未闭合块注释的 DFA 如下：



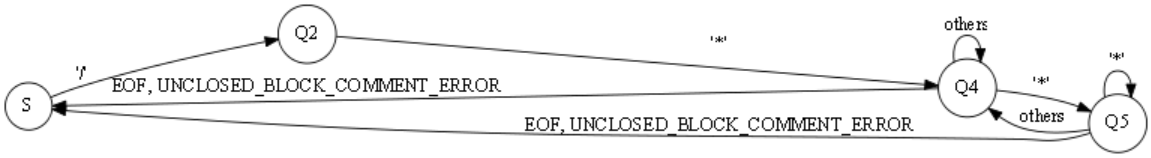
3.10.3. 未闭合字符常量错误 UNCLOSED_CHAR_CONSTANT_ERROR

本程序中将出现一个单引号后未正常终结的字符或者非法转义字符视为未闭合字符常量错误。识别未闭合字符常量错误的 DFA 如下所示：



3.10.4. 未闭合字符串字面量错误 UNCLOSED_STRING_LITERAL_ERROR

程序将中途有换行或已读取到文件结尾但仍未以”“”终结的字符串字面量视为未闭合字符串常量错误。识别未闭合字符串常量错误的 DFA 如下所示：



3.10.5. 非法字符错误 ILLEGAL_CHAR_ERROR

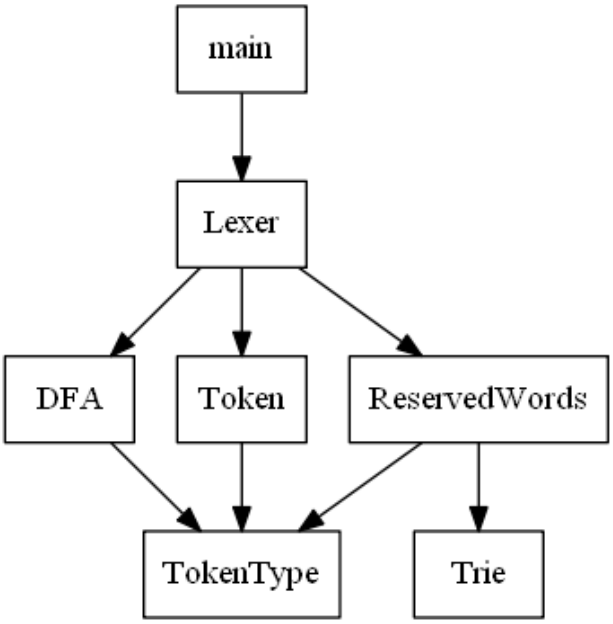
只考虑 ASCII 码，C 语言中出现的非法字符只可能为’$’或’@’，故当读取到该两个字符时，将识别为非法字符错误，并从下一个字符重新开始识别。

4. 程序实现

本词法分析程序采用了 C++来编写，程序中实现了一个 Lexer 类作为词法分析器的对外接口，并为其重载了输入输出流操作符以模仿词法分析的输入输出过程，外部调用方式如以下代码所示：

```
Lexer lexer; //词法分析器
ifstream in(IN_FILE); //输入文件为C语言代码
ofstream out(OUT_FILE); //输出文件为经分析得到的token流
in >> lexer;
out << lexer;
```

本程序的结构示意图如下所示：



各模块实现简述如下：

4.1. TokenType

该模块以枚举常量 TokenType 定义了各种 token 的类型，大致如下：

```
enum TokenType {
    UNKNOWN,

    CHAR,
    //...
    IF,
    //...
    ADD,
    //...
    PREPROCESSOR,
    IDENTIFIER,
    //...
};
```

并定义了各种 TokenType 对应的字符串与符号数组用于输出：

```
const array<string, TOKEN_TYPE_NUM> tokenTypeStrs = {
    "UNKNOWN",
    "CHAR",
    //...
};

const array<string, TOKEN_TYPE_NUM> tokenVals = {
    "unknown",
    "char",
    //...
};
```

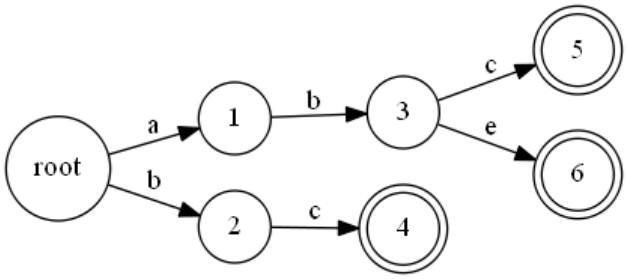
4.2. Token

定义了 Token 结构体，大致如下：

```
struct Token {
    TokenType type; //类型
    int symbolPos; //表示的符号在符号表中位置
    int row, col; //在代码中的行列位置
};
```

4.3. Trie

Trie 树是一种用于快速在一组字符串中检索某个字符串是否出现的数据结构，实现思路是各个字符串按其公共前缀存储在一棵树上，每个字符串对应一条从根结点到叶节点的路径。例如三个字符串”abc”、”abe”、”bc”被存储为：



其本质就是一个能识别一组给定字符串的最简 DFA，同时其还可以在各结点上存储信息，构成一个 Moore 状态机，从而**实现字符串到某种信息的映射**。该模块定义了 Trie 树结构体大致如下：

```
template <typename T, T DefaultVal>
class Trie {
    private:
    enum {
        SIGMA_SIZE = 128 //字母表大小
    };
    struct Node { //结点
        int next[SIGMA_SIZE]; //读取一个字符后转移到的下一个状态
        T val; //该节点上存储的信息
    };
    Node* nodes; //结点表
    int size; //结点数目
    int root; //根节点标号

    public:
    //将一个带val信息的字符串插入Trie树中
    void insert(const string& word, const T& val);
    //从Trie树中查询某一个字符串所带信息
    T search(const string& word);
};
```

在本程序中 Trie 树有以下两个应用：

1) 从**保留字表**中快速查询一个标识符是否为保留字，并获取其类型；
2) 从**符号表**中快速查询一个字符串符号是否已在其中，并获取其位置。

4.4. ReservedWords

本模块以 Trie 树为基础构造了保留字表。

4.5. DFA

本模块构造以表驱动的方式构造了识别各种 token 的 DFA。利用数组 `states[cur].next[c]`存下当前状态 `cur` 在读取一个字符 `c` 后转移到的下一个状态，则在转移的时候查表即可，而避免了 `switch` 分支的耗时。定义的 DFA 结构体大致如下：

```
class DFA {
    enum {
        STATE_NUM = 50, //状态数
        SIGMA_SIZE = 128, //字母表大小
    };
    struct State {
        int next[SIGMA_SIZE]; //读取一个字母后转移到的下一个状态
        TokenType output[SIGMA_SIZE]; //读取一个字母后输出已识别的token类型
        bitset<SIGMA_SIZE> retractFlg; //标记读取一个字母后是否需回退
    } states[STATE_NUM];
    int cur; //当前状态标号
    TokenType output; //当前输出
    bool retractFlg; //当前回退标记

    public:
    enum {
        START = 0 //起始状态标号
    };
    //初始化
    void init();
    //读取字母c进行状态转移
    void trans(char c);
    //获取当前状态标号
    int getState();
    //获取当前输出
    TokenType getOutput();
    //判断当前是否需回退
    bool needsRetract();
};
```

4.6. Lexer

该模块定义了程序对外的接口类 Lexer，用于完成与外部的输入输出，根据内置 DFA 的转移结果进行 token 的存储及符号表的构造，并完成一些统计工作。Lexer 的类定义大致如下：

```
class Lexer {
    friend istream& operator >> (istream& in, Lexer& rhs); //输入流操作符
    friend ostream& operator <<(ostream& out, const Lexer& rhs); //输出流操作符

    private:
    static DFA* dfa; //DFA
    static ReservedWords* reservedWords; //保留字表
    vector<Token> tokens; //识别出的token流
    vector<string> symbols; //符号表
    Trie<int, -1>* posInSymbols; //查询当前符号在符号表中位置的Trie
    int rowCnt; //行计数器
    int charCnt; //字符计数器
    array<int, TOKEN_TYPE_NUM> tokenTypeCnts; //各类token的计数器
    vector<int> errorTokenIds; //错误token标号
};
```

其实现细节如下：

```
istream& operator >>(istream& in, Lexer& rhs) {
    //初始化
    rhs.tokens.clear();
    rhs.symbols.clear();
    rhs.posInSymbols = new Trie<int, -1>(Lexer::MAX_TRIE_SIZE);
    rhs.dfa->init();
    rhs.charCnt = 0;
    fill(rhs.tokenTypeCnts.begin(), rhs.tokenTypeCnts.end(), 0);
    rhs.errorTokenIds.clear();

    Token curToken; //当前Token
    string curSymbol; //当前符号
    int curRow(1), curCol(1); //当前行列
    bool reachesEOF = false; //标记是否已读至文件末尾
    bool expectsNewToken = true; //标记是否即将读入一个新Token
    for (char c; !reachesEOF; ) {
        if ((c = in.get()) == EOF) {
            //for recognition of token just before EOF
            //since '\0' does not exist in the C-code text,
            //use it to substitute EOF = -1
            c = '\0';
            reachesEOF = true;
        }
        if (expectsNewToken && !isBlankChar(c)) {
            curToken.row = curRow; //此时记录下新Token的起始行列位置
            curToken.col = curCol;
            expectsNewToken = false; //将标记复位
        }
        Lexer::dfa->trans(c); //自动机读入c进行状态转移
        if (Lexer::dfa->needsRetract()) {
            in.putback(c); //若需回退则将c放回输入流
        } else {
            if (Lexer::dfa->getState() != DFA::START) {
                curSymbol += c; //非起始状态，空白符不可丢弃
            } else if (!isBlankChar(c)) {
                curSymbol += c; //起始状态，空白符需丢弃
            }
            ++rhs.charCnt; //字符计数器递增
            ++curCol;
            if (c == '\n') {
                ++curRow; //换行
                curCol = 1;
            }
        }
        if (Lexer::dfa->getOutput() != UNKNOWN) { //识别出一个非未知(UNKNOWN)Token
            curToken.type = Lexer::dfa->getOutput();
            if (curToken.type == IDENTIFIER) { //若是标识符，需查询是否为保留字
                curToken.type = Lexer::reservedWords->search(curSymbol);
            }
            switch (curToken.type) {
                case INCOMPLETE_NUMERIC_CONSTANT_ERROR:
                case UNCLOSED_BLOCK_COMMENT_ERROR:
                case UNCLOSED_CHAR_CONSTANT_ERROR:
                case UNCLOSED_STRING_LITERAL_ERROR:
                case ILLEGAL_CHAR_ERROR:
                    rhs.errorTokenIds.push_back(rhs.tokens.size()); //将错误归档
                case PREPROCESSOR:
                case LINE_COMMENT:
                case BLOCK_COMMENT:
                case IDENTIFIER:
                case NUMERIC_CONSTANT:
                case CHAR_CONSTANT:
                case STRING_LITERAL: {
```

```
                    int pos(rhs.posInSymbols->search(curSymbol));
                    if (!~pos) { //若识别出的符号不在符号表中，则新建符号表项存入该符号
                        pos = rhs.symbols.size();
                        rhs.posInSymbols->insert(curSymbol, pos);
                        rhs.symbols.push_back(curSymbol);
                    }
                    curToken.symbolPos = pos;
                    break;
                }
                default:
                    curToken.symbolPos = curToken.type;
            }
            rhs.tokens.push_back(curToken); //将该Token放入Token流中
            ++rhs.tokenTypeCnts[curToken.type]; //递增该Token类型的计数器
            curSymbol.clear(); //清空当前符号
            expectsNewToken = true; //将标记置位
        }
        rhs.rowCnt = curRow; //行计数器等于最后一行行号
    }
    return in;
}
```

5. 程序测试

测试代码如下（有意使格式与结构混乱以增加测试压力）：

```
#include <stdio.h>

//this is a line comment
/*
 * this is a block comment
 * "in block comment"
 */

typedef struct Struct {
	int m1, m2;
	union {
		char m3;
		float m4;
	}
} Struct;

int arr[10];

int doSth() {
static int i = 0;
	doSth(); //u r kiddin'
	i += i, i /= i, i |= i;

	return i <<= (1 << (3 >> 1));
}

int main(    int   argc,  char* argv[  ]) {

	long long LL = 5;
	int _i_1 = 123abc;
	float f = 1.25;
```

```
double db1 = 1.3e6, db2 = 2.E+7, db3 = 3.5e;

Struct s1, *ps2 = &s1;
s1.m1 = 5;
ps2->m2 = 6;

char* str = "abcdef\"/*this is not a comment*/";


char *empty = "";

char* nstr = "asdfa   sdfasdfas

char c1 = 'f', c2 = '\t', c3 = '\m', c4 = 'sdaf', c5 = 'd

int a = /*
block comment
block comment

block comment
block comment
block comment
block comment


block comment
block comment block***block*/3;
LL++;
--LL;
LL += _i_1;

123abc = 6 ^ 7 | 8 & ~9 / 10+9/7%8;

unsigned long $a, @f;

label:

#if 1
#endif

doSth();

while (     1    && doSth()) {
      int i = sizeof(Struct);
      for (; i < 10; ++i) {
            if (i == 0) {
                        continue;
            } else if (3 != 7 && a >=5 ||6<9)
            int umm;


            switch (i & 1) {
                  case 0:


                        if (i > 5)
```

```
                {
                                        printf("nonsense %d", i);
                                } else if (i < 2) {
                                        return 0;
                                } else {
                                        umm = i | 10? 3+4 :  4 * 6;
                                }
                                break;
                        default:
                                goto label;
                                break;
                    }
                }
        }

        return 0;
}
 /*fasdfasdfasd /*fasdfasdfasd
```

输出分为 Token Stream、Error、Analysis 三部分，分别为经词法分析得到的 Token 流、错误信息及各统计分析信息（总行数、字符总数、各类 Token 计数）。如下所示：

```
Errors:

<INCOMPLETE_NUMERIC_CONSTANT_ERROR, "3.5e"> in (32 : 41)
<UNCLOSED_STRING_LITERAL_ERROR, ""asdfa   sdfasdfas"> in (43 : 15)
<UNCLOSED_CHAR_CONSTANT_ERROR, "'\"> in (45 : 33)
<UNCLOSED_CHAR_CONSTANT_ERROR, "',"> in (45 : 36)
<UNCLOSED_CHAR_CONSTANT_ERROR, "'s"> in (45 : 44)
<UNCLOSED_CHAR_CONSTANT_ERROR, "',"> in (45 : 49)
<UNCLOSED_CHAR_CONSTANT_ERROR, "'d"> in (45 : 57)
<ILLEGAL_CHAR_ERROR, "$"> in (65 : 16)
<ILLEGAL_CHAR_ERROR, "@"> in (65 : 20)
<UNCLOSED_BLOCK_COMMENT_ERROR, "/*fasdfasdfasd"> in (106 : 2)
```

```
Analysis:

row count = 106
char count = 1421
count of each type =
        CHAR : 6
        INT : 10
        LONG : 3
        FLOAT : 2
        DOUBLE : 1
        UNSIGNED : 1
```

```
STATIC : 1
STRUCT : 1
UNION : 1
TYPEDEF : 1
SIZEOF : 1
IF : 4
ELSE : 3
WHILE : 1
FOR : 1
SWITCH : 1
CASE : 1
DEFAULT : 1
CONTINUE : 1
BREAK : 2
GOTO : 1
RETURN : 3
ASSIGN : 22
ADD : 2
INC : 2
ADD_ASSIGN : 2
DEC : 1
DIV : 2
DIV_ASSIGN : 1
MOD : 1
BITWISE_OR : 2
BITWISE_OR_ASSIGN : 1
BITWISE_XOR : 1
BITWISE_NOT : 1
SHL : 1
SHL_ASSIGN : 1
SHR : 1
AND : 2
OR : 1
LESS : 3
EQUAL : 1
INEQUAL : 1
GREATER : 1
GREATER_EQUAL : 1
ASTERISK : 6
AMPERSAND : 3
```

```
QUESTION : 1
COMMA : 11
COLON : 4
SEMICOLON : 37
DOT : 1
ARROW : 1
L_BRACE : 11
R_BRACE : 11
L_SQUARE : 2
R_SQUARE : 2
L_PAREN : 16
R_PAREN : 16
PREPROCESSOR : 3
LINE_COMMENT : 2
BLOCK_COMMENT : 2
IDENTIFIER : 71
NUMERIC_CONSTANT : 41
CHAR_LITERAL : 2
STRING_LITERAL : 3
INCOMPLETE_NUMERIC_CONSTANT_ERROR : 1
UNCLOSED_BLOCK_COMMENT_ERROR : 1
UNCLOSED_CHAR_CONSTANT_ERROR : 5
UNCLOSED_STRING_LITERAL_ERROR : 1
ILLEGAL_CHAR_ERROR : 2
```

```
<PREPROCESSOR, "#include <stdio.h>"> in (1 : 1)
<LINE_COMMENT, "//this is a line comment"> in (3 : 1)
<BLOCK_COMMENT, "/*
 * this is a block comment
 * "in block comment"
 */"> in (4 : 1)
<TYPEDEF, "typedef"> in (9 : 1)
<STRUCT, "struct"> in (9 : 9)
<IDENTIFIER, "Struct"> in (9 : 16)
<L_BRACE, "{"> in (9 : 23)
<INT, "int"> in (10 : 2)
<IDENTIFIER, "m1"> in (10 : 6)
<COMMA, ","> in (10 : 8)
<IDENTIFIER, "m2"> in (10 : 10)
<SEMICOLON, ";"> in (10 : 12)
<UNION, "union"> in (11 : 2)
<L_BRACE, "{"> in (11 : 8)
<CHAR, "char"> in (12 : 3)
<IDENTIFIER, "m3"> in (12 : 8)
<SEMICOLON, ";"> in (12 : 10)
<FLOAT, "float"> in (13 : 3)
<IDENTIFIER, "m4"> in (13 : 9)
<SEMICOLON, ";"> in (13 : 11)
<R_BRACE, "}"> in (14 : 2)
<R_BRACE, "}"> in (15 : 1)
<IDENTIFIER, "Struct"> in (15 : 3)
<SEMICOLON, ";"> in (15 : 9)
```

```
<INT, "int"> in (17 : 1)
<IDENTIFIER, "arr"> in (17 : 5)
<L_SQUARE, "["> in (17 : 8)
<NUMERIC_CONSTANT, "10"> in (17 : 9)
<R_SQUARE, "]"> in (17 : 11)
<SEMICOLON, ";"> in (17 : 12)
<INT, "int"> in (19 : 1)
<IDENTIFIER, "doSth"> in (19 : 5)
<L_PAREN, "("> in (19 : 10)
<R_PAREN, ")"> in (19 : 11)
<L_BRACE, "{"> in (19 : 13)
<STATIC, "static"> in (20 : 1)
<INT, "int"> in (20 : 8)
<IDENTIFIER, "i"> in (20 : 12)
<ASSIGN, "="> in (20 : 14)
<NUMERIC_CONSTANT, "0"> in (20 : 16)
<SEMICOLON, ";"> in (20 : 17)
<IDENTIFIER, "doSth"> in (21 : 2)
<L_PAREN, "("> in (21 : 7)
<R_PAREN, ")"> in (21 : 8)
<SEMICOLON, ";"> in (21 : 9)
<LINE_COMMENT, "//u r kiddin'"> in (21 : 11)
<IDENTIFIER, "i"> in (22 : 2)
<ADD_ASSIGN, "+="> in (22 : 4)
<IDENTIFIER, "i"> in (22 : 7)
<COMMA, ","> in (22 : 8)
<IDENTIFIER, "i"> in (22 : 10)
<DIV_ASSIGN, "/="> in (22 : 12)
<IDENTIFIER, "i"> in (22 : 15)
<COMMA, ","> in (22 : 16)
<IDENTIFIER, "i"> in (22 : 18)
<BITWISE_OR_ASSIGN, "|="> in (22 : 20)
<IDENTIFIER, "i"> in (22 : 23)
<SEMICOLON, ";"> in (22 : 24)
<RETURN, "return"> in (24 : 2)
<IDENTIFIER, "i"> in (24 : 9)
<SHL_ASSIGN, "<<="> in (24 : 11)
<L_PAREN, "("> in (24 : 15)
<NUMERIC_CONSTANT, "1"> in (24 : 16)
<SHL, "<<"> in (24 : 18)
<L_PAREN, "("> in (24 : 21)
<NUMERIC_CONSTANT, "3"> in (24 : 22)
<SHR, ">>"> in (24 : 24)
<NUMERIC_CONSTANT, "1"> in (24 : 27)
<R_PAREN, ")"> in (24 : 28)
<R_PAREN, ")"> in (24 : 29)
<SEMICOLON, ";"> in (24 : 30)
<R_BRACE, "}"> in (25 : 1)
<INT, "int"> in (27 : 1)
<IDENTIFIER, "main"> in (27 : 5)
<L_PAREN, "("> in (27 : 9)
<INT, "int"> in (27 : 14)
<IDENTIFIER, "argc"> in (27 : 20)
<COMMA, ","> in (27 : 24)
<CHAR, "char"> in (27 : 27)
<ASTERISK, "*"> in (27 : 31)
<IDENTIFIER, "argv"> in (27 : 33)
```

```
<L_SQUARE, "["> in (27 : 37)
<R_SQUARE, "]"> in (27 : 40)
<R_PAREN, ")"> in (27 : 41)
<L_BRACE, "{"> in (27 : 43)
<LONG, "long"> in (29 : 2)
<LONG, "long"> in (29 : 7)
<IDENTIFIER, "LL"> in (29 : 12)
<ASSIGN, "="> in (29 : 15)
<NUMERIC_CONSTANT, "5"> in (29 : 17)
<SEMICOLON, ";"> in (29 : 18)
<INT, "int"> in (30 : 2)
<IDENTIFIER, "_i_1"> in (30 : 6)
<ASSIGN, "="> in (30 : 11)
<NUMERIC_CONSTANT, "123"> in (30 : 13)
<IDENTIFIER, "abc"> in (30 : 16)
<SEMICOLON, ";"> in (30 : 19)
<FLOAT, "float"> in (31 : 2)
<IDENTIFIER, "f"> in (31 : 8)
<ASSIGN, "="> in (31 : 10)
<NUMERIC_CONSTANT, "1.25"> in (31 : 12)
<SEMICOLON, ";"> in (31 : 16)
<DOUBLE, "double"> in (32 : 2)
<IDENTIFIER, "db1"> in (32 : 9)
<ASSIGN, "="> in (32 : 13)
<NUMERIC_CONSTANT, "1.3e6"> in (32 : 15)
<COMMA, ","> in (32 : 20)
<IDENTIFIER, "db2"> in (32 : 22)
<ASSIGN, "="> in (32 : 26)
<NUMERIC_CONSTANT, "2.E+7"> in (32 : 28)
<COMMA, ","> in (32 : 33)
<IDENTIFIER, "db3"> in (32 : 35)
<ASSIGN, "="> in (32 : 39)
<INCOMPLETE_NUMERIC_CONSTANT_ERROR, "3.5e"> in (32 : 41)
<SEMICOLON, ";"> in (32 : 45)
<IDENTIFIER, "Struct"> in (34 : 2)
<IDENTIFIER, "s1"> in (34 : 9)
<COMMA, ","> in (34 : 11)
<ASTERISK, "*"> in (34 : 13)
<IDENTIFIER, "ps2"> in (34 : 14)
<ASSIGN, "="> in (34 : 18)
<AMPERSAND, "&"> in (34 : 20)
<IDENTIFIER, "s1"> in (34 : 21)
<SEMICOLON, ";"> in (34 : 23)
<IDENTIFIER, "s1"> in (35 : 2)
<DOT, "."> in (35 : 4)
<IDENTIFIER, "m1"> in (35 : 5)
<ASSIGN, "="> in (35 : 8)
<NUMERIC_CONSTANT, "5"> in (35 : 10)
<SEMICOLON, ";"> in (35 : 11)
<IDENTIFIER, "ps2"> in (36 : 2)
<ARROW, "->"> in (36 : 5)
<IDENTIFIER, "m2"> in (36 : 7)
<ASSIGN, "="> in (36 : 10)
<NUMERIC_CONSTANT, "6"> in (36 : 12)
<SEMICOLON, ";"> in (36 : 13)
<CHAR, "char"> in (38 : 2)
<ASTERISK, "*"> in (38 : 6)
```

```
<IDENTIFIER, "str"> in (38 : 8)
<ASSIGN, "="> in (38 : 12)
<STRING_LITERAL, ""abcdef\"/*this is not a comment*/""> in (38 :
14)
<SEMICOLON, ";"> in (38 : 49)
<CHAR, "char"> in (41 : 2)
<ASTERISK, "*"> in (41 : 7)
<IDENTIFIER, "empty"> in (41 : 8)
<ASSIGN, "="> in (41 : 14)
<STRING_LITERAL, """"> in (41 : 16)
<SEMICOLON, ";"> in (41 : 18)
<CHAR, "char"> in (43 : 2)
<ASTERISK, "*"> in (43 : 6)
<IDENTIFIER, "nstr"> in (43 : 8)
<ASSIGN, "="> in (43 : 13)
<UNCLOSED_STRING_LITERAL_ERROR, ""asdfa   sdfasdfas"> in (43 : 15)
<CHAR, "char"> in (45 : 2)
<IDENTIFIER, "c1"> in (45 : 7)
<ASSIGN, "="> in (45 : 10)
<CHAR_LITERAL, "'f'"> in (45 : 12)
<COMMA, ","> in (45 : 15)
<IDENTIFIER, "c2"> in (45 : 17)
<ASSIGN, "="> in (45 : 20)
<CHAR_LITERAL, "'\t'"> in (45 : 22)
<COMMA, ","> in (45 : 26)
<IDENTIFIER, "c3"> in (45 : 28)
<ASSIGN, "="> in (45 : 31)
<UNCLOSED_CHAR_CONSTANT_ERROR, "'\"> in (45 : 33)
<IDENTIFIER, "m"> in (45 : 35)
<UNCLOSED_CHAR_CONSTANT_ERROR, "',"> in (45 : 36)
<IDENTIFIER, "c4"> in (45 : 39)
<ASSIGN, "="> in (45 : 42)
<UNCLOSED_CHAR_CONSTANT_ERROR, "'s"> in (45 : 44)
<IDENTIFIER, "daf"> in (45 : 46)
<UNCLOSED_CHAR_CONSTANT_ERROR, "',"> in (45 : 49)
<IDENTIFIER, "c5"> in (45 : 52)
<ASSIGN, "="> in (45 : 55)
<UNCLOSED_CHAR_CONSTANT_ERROR, "'d"> in (45 : 57)
<INT, "int"> in (47 : 2)
<IDENTIFIER, "a"> in (47 : 6)
<ASSIGN, "="> in (47 : 8)
<BLOCK_COMMENT, "/*
      block comment
      block comment

      block comment
      block comment
      block comment
      block comment


      block comment
      block comment block***block*/"> in (47 : 10)
<NUMERIC_CONSTANT, "3"> in (58 : 31)
<SEMICOLON, ";"> in (58 : 32)
<IDENTIFIER, "LL"> in (59 : 2)
<INC, "++"> in (59 : 4)
```

```
<SEMICOLON, ";"> in (59 : 6)
<DEC, "--"> in (60 : 2)
<IDENTIFIER, "LL"> in (60 : 4)
<SEMICOLON, ";"> in (60 : 6)
<IDENTIFIER, "LL"> in (61 : 2)
<ADD_ASSIGN, "+="> in (61 : 5)
<IDENTIFIER, "_i_1"> in (61 : 8)
<SEMICOLON, ";"> in (61 : 12)
<NUMERIC_CONSTANT, "123"> in (63 : 2)
<IDENTIFIER, "abc"> in (63 : 5)
<ASSIGN, "="> in (63 : 9)
<NUMERIC_CONSTANT, "6"> in (63 : 11)
<BITWISE_XOR, "^"> in (63 : 13)
<NUMERIC_CONSTANT, "7"> in (63 : 15)
<BITWISE_OR, "|"> in (63 : 17)
<NUMERIC_CONSTANT, "8"> in (63 : 19)
<AMPERSAND, "&"> in (63 : 21)
<BITWISE_NOT, "~"> in (63 : 23)
<NUMERIC_CONSTANT, "9"> in (63 : 24)
<DIV, "/"> in (63 : 26)
<NUMERIC_CONSTANT, "10"> in (63 : 28)
<ADD, "+"> in (63 : 30)
<NUMERIC_CONSTANT, "9"> in (63 : 31)
<DIV, "/"> in (63 : 32)
<NUMERIC_CONSTANT, "7"> in (63 : 33)
<MOD, "%"> in (63 : 34)
<NUMERIC_CONSTANT, "8"> in (63 : 35)
<SEMICOLON, ";"> in (63 : 36)
<UNSIGNED, "unsigned"> in (65 : 2)
<LONG, "long"> in (65 : 11)
<ILLEGAL_CHAR_ERROR, "$"> in (65 : 16)
<IDENTIFIER, "a"> in (65 : 17)
<COMMA, ","> in (65 : 18)
<ILLEGAL_CHAR_ERROR, "@"> in (65 : 20)
<IDENTIFIER, "f"> in (65 : 21)
<SEMICOLON, ";"> in (65 : 22)
<IDENTIFIER, "label"> in (67 : 2)
<COLON, ":"> in (67 : 7)
<PREPROCESSOR, "#if 1"> in (69 : 2)
<PREPROCESSOR, "#endif"> in (70 : 2)
<IDENTIFIER, "doSth"> in (72 : 2)
<L_PAREN, "("> in (72 : 7)
<R_PAREN, ")"> in (72 : 8)
<SEMICOLON, ";"> in (72 : 9)
<WHILE, "while"> in (74 : 2)
<L_PAREN, "("> in (74 : 8)
<NUMERIC_CONSTANT, "1"> in (74 : 14)
<AND, "&&"> in (74 : 19)
<IDENTIFIER, "doSth"> in (74 : 22)
<L_PAREN, "("> in (74 : 27)
<R_PAREN, ")"> in (74 : 28)
<R_PAREN, ")"> in (74 : 29)
<L_BRACE, "{"> in (74 : 31)
<INT, "int"> in (75 : 3)
<IDENTIFIER, "i"> in (75 : 7)
<ASSIGN, "="> in (75 : 9)
<SIZEOF, "sizeof"> in (75 : 11)
```

```
<L_PAREN, "("> in (75 : 17)
<IDENTIFIER, "Struct"> in (75 : 18)
<R_PAREN, ")"> in (75 : 24)
<SEMICOLON, ";"> in (75 : 25)
<FOR, "for"> in (76 : 3)
<L_PAREN, "("> in (76 : 7)
<SEMICOLON, ";"> in (76 : 8)
<IDENTIFIER, "i"> in (76 : 10)
<LESS, "<"> in (76 : 12)
<NUMERIC_CONSTANT, "10"> in (76 : 14)
<SEMICOLON, ";"> in (76 : 16)
<INC, "++"> in (76 : 18)
<IDENTIFIER, "i"> in (76 : 20)
<R_PAREN, ")"> in (76 : 21)
<L_BRACE, "{"> in (76 : 23)
<IF, "if"> in (77 : 4)
<L_PAREN, "("> in (77 : 7)
<IDENTIFIER, "i"> in (77 : 8)
<EQUAL, "=="> in (77 : 10)
<NUMERIC_CONSTANT, "0"> in (77 : 13)
<R_PAREN, ")"> in (77 : 14)
<L_BRACE, "{"> in (77 : 16)
<CONTINUE, "continue"> in (78 : 10)
<SEMICOLON, ";"> in (78 : 18)
<R_BRACE, "}"> in (79 : 4)
<ELSE, "else"> in (79 : 6)
<IF, "if"> in (79 : 11)
<L_PAREN, "("> in (79 : 14)
<NUMERIC_CONSTANT, "3"> in (79 : 15)
<INEQUAL, "!="> in (79 : 17)
<NUMERIC_CONSTANT, "7"> in (79 : 20)
<AND, "&&"> in (79 : 22)
<IDENTIFIER, "a"> in (79 : 25)
<GREATER_EQUAL, ">="> in (79 : 27)
<NUMERIC_CONSTANT, "5"> in (79 : 29)
<OR, "||"> in (79 : 31)
<NUMERIC_CONSTANT, "6"> in (79 : 33)
<LESS, "<"> in (79 : 34)
<NUMERIC_CONSTANT, "9"> in (79 : 35)
<R_PAREN, ")"> in (79 : 36)
<INT, "int"> in (80 : 4)
<IDENTIFIER, "umm"> in (80 : 8)
<SEMICOLON, ";"> in (80 : 11)
<SWITCH, "switch"> in (83 : 4)
<L_PAREN, "("> in (83 : 11)
<IDENTIFIER, "i"> in (83 : 12)
<AMPERSAND, "&"> in (83 : 14)
<NUMERIC_CONSTANT, "1"> in (83 : 16)
<R_PAREN, ")"> in (83 : 17)
<L_BRACE, "{"> in (83 : 19)
<CASE, "case"> in (84 : 5)
<NUMERIC_CONSTANT, "0"> in (84 : 10)
<COLON, ":"> in (84 : 11)
<IF, "if"> in (87 : 6)
<L_PAREN, "("> in (87 : 9)
<IDENTIFIER, "i"> in (87 : 10)
<GREATER, ">"> in (87 : 12)
```

```
<NUMERIC_CONSTANT, "5"> in (87 : 14)
<R_PAREN, ")"> in (87 : 15)
<L_BRACE, "{"> in (89 : 4)
<IDENTIFIER, "printf"> in (90 : 7)
<L_PAREN, "("> in (90 : 13)
<STRING_LITERAL, ""nonsense %d""> in (90 : 14)
<COMMA, ","> in (90 : 27)
<IDENTIFIER, "i"> in (90 : 29)
<R_PAREN, ")"> in (90 : 30)
<SEMICOLON, ";"> in (90 : 31)
<R_BRACE, "}"> in (91 : 6)
<ELSE, "else"> in (91 : 8)
<IF, "if"> in (91 : 13)
<L_PAREN, "("> in (91 : 16)
<IDENTIFIER, "i"> in (91 : 17)
<LESS, "<"> in (91 : 19)
<NUMERIC_CONSTANT, "2"> in (91 : 21)
<R_PAREN, ")"> in (91 : 22)
<L_BRACE, "{"> in (91 : 24)
<RETURN, "return"> in (92 : 7)
<NUMERIC_CONSTANT, "0"> in (92 : 14)
<SEMICOLON, ";"> in (92 : 15)
<R_BRACE, "}"> in (93 : 6)
<ELSE, "else"> in (93 : 8)
<L_BRACE, "{"> in (93 : 13)
<IDENTIFIER, "umm"> in (94 : 7)
<ASSIGN, "="> in (94 : 11)
<IDENTIFIER, "i"> in (94 : 13)
<BITWISE_OR, "|"> in (94 : 15)
<NUMERIC_CONSTANT, "10"> in (94 : 17)
<QUESTION, "?"> in (94 : 19)
<NUMERIC_CONSTANT, "3"> in (94 : 21)
<ADD, "+"> in (94 : 22)
<NUMERIC_CONSTANT, "4"> in (94 : 23)
<COLON, ":"> in (94 : 25)
<NUMERIC_CONSTANT, "4"> in (94 : 28)
<ASTERISK, "*"> in (94 : 30)
<NUMERIC_CONSTANT, "6"> in (94 : 32)
<SEMICOLON, ";"> in (94 : 33)
<R_BRACE, "}"> in (95 : 6)
<BREAK, "break"> in (96 : 6)
<SEMICOLON, ";"> in (96 : 11)
<DEFAULT, "default"> in (97 : 5)
<COLON, ":"> in (97 : 12)
<GOTO, "goto"> in (98 : 6)
<IDENTIFIER, "label"> in (98 : 11)
<SEMICOLON, ";"> in (98 : 16)
<BREAK, "break"> in (99 : 6)
<SEMICOLON, ";"> in (99 : 11)
<R_BRACE, "}"> in (100 : 4)
<R_BRACE, "}"> in (101 : 3)
<R_BRACE, "}"> in (102 : 2)
<RETURN, "return"> in (104 : 2)
<NUMERIC_CONSTANT, "0"> in (104 : 9)
<SEMICOLON, ";"> in (104 : 10)
<R_BRACE, "}"> in (105 : 1)
<UNCLOSED_BLOCK_COMMENT_ERROR, "/*fasdfasdfasd"> in (106 : 2)
```

6. 分析总结

经逐一对照，上述输出结果对合法的 C 语言代码分析完全正确，而对于错误代码的分析结果值得斟酌。对比 Clang 的词法分析结果，有如下几例差异：

1) 对于不合法标识符或数字常量表达式”123abc”，本例将其识别为一个数字常量”123”与一个标识符”abc”，由于两者之间没有其他 Token 隔开，故将在语法分析阶段发现错误；而 Clang 直接将其识别为一个数字常量，从而再语法分析阶段解析该字符串的过程也会发现错误。
2) 对于未闭合字符常量及字符串字面量，Clang 直接识别为 unknown 类型，没有具体化。此外，形如’xxx’，即单引号括起超过一个非转义字符的形式也会被 Clang 识别为字符常量。
3) 对于未闭合注释，Clang 会将其全部跳过。其实在平常的 IDE 或文本编辑器中，未闭合注释也被作为到文件尾全为注释处理。

在这次实验中，本人前前后后为各种涉及 Token 表示方式设计、数据结构设计、类层次结构及接口设计、性能优化及健壮性的细节都花了一定心思，经历了一次结果较为满意的工程实践。通过亲自实现各种细节，也对词法分析的思路有了更深入的理解。然词法分析得到的成果局限性较大，期望在接下来的语法分析实验得到更大的锻炼。