北京邮电大学《编译原理与技术》课程实验报告

<table>
<tr><td>实验名称</td><td colspan="2">C 语言词法分析程序的设计与实现</td><td>学院</td><td>计算机</td><td>指导教师</td><td>王雅文</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">成绩</td></tr>
<tr><td>2020211323</td><td>11</td><td colspan="2">2020211429</td><td>陈宇</td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td></td><td colspan="2"></td><td></td><td colspan="2"></td></tr>
<tr><td>实验内容</td><td colspan="6">编写 C 语言程序实现以下功能：
可以识别出用 C 语言编写的原程序中的每个单词符号，并以记号的形式出每个单词符号。
可以识别并跳过源程序中的注释。
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。
检查源程序中存在的词法错误，并报告错误所在的位置。
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</td></tr>
<tr><td>学生实验报告</td><td colspan="6">(详见“实验报告和源程序”册)</td></tr>
<tr><td>课程设计成绩评定</td><td colspan="6">评语:

成绩:

指导教师签名:

年 月 日</td></tr>
</table>

注：评语要体现每个学生的工作情况，可以加页。

C 语言词法分析程序的设计与实现

一、实验内容以及要求

1.1 实验内容

编写 C 语言程序实现以下功能：

可以识别出用 C 语言编写的原程序中的每个单词符号，并以记号的形式出每个单词符号。

可以识别并跳过源程序中的注释。

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。

检查源程序中存在的词法错误，并报告错误所在的位置。

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。

1.2 实验要求

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、C++程序设计说明

2.1 设计思路

为了能够将尽可能简洁易处理的记号流交给语法分析程序，记号流的类型和属性都应尽量用数而非字符串表示，属性部分可采用 union 类型存储，用于存储不同类型的数。因此需要为每种记号设计定义常量，为标志符、字符串常量设计表格，并将对应位置输出作为记号属性。

为了进一步减少语法分析程序的工作量，可将使用场景相同的符号归为一类，例如关系运算符、赋值运算符。

最后，依据 C 语言的词法规则构造状态转移图，依据状态转移图编写词法分析函数。

2.2 数据结构

2.2.1 结构体

```
union value_type
{
    int i;
    unsigned int ui;
    long l;
    unsigned long int ul;
```

```
    unsigned char c;
    float f;
    double d;
};

struct token
{
    word_type type;    //记号类型
    value_type value;  //记号属性
};
```

2.2.2 常量

```
//关键字表，顺序排列
const vector<string> KEYWORD_LIST = { "auto", "break", "case", "char", "const",
"continue", "default", "do", "double", "else", "enum", "extern", "float", "for",
"goto", "if", "int", "long", "register", "return", "short", "signed", "sizeof",
"static", "struct", "switch", "typedef", "union", "unsigned", "void", "volatile",
"while" };

const int WORD_TYPE_AMOUNT = 40; // 记号类型数量，不包含注释和具体的关系运算符和赋
值运算符
enum word_type
{
    KEYWORD,                    //关键字
    ID,                         //标志符
    STRING,                     //字符串常量
    CHAR,                       //字符常量
    INT,                        //整型常量
    UINT,                       //无符号整型常量
    LONG,                       //长整型常量
    ULONG,                      //无符号长整型
    FLOAT,                      //单精度浮点数
    DOUBLE,                     //双精度浮点数
    RELATION_OPERATOR,           //关系运算符
    ASSIGN_OPERATOR,             //赋值运算符
    PLUS,                       //"+"
    MINUS,                      //"-"
    MULTIPLY,                    //"*"
    DIVIDE,                     //"/"
    MOD,                        //"%"
    INC,                        //"++"
    DEC,                        //"--"
    LOGICAL_AND,                 //"&&"
    LOGICAL_OR,                  //"||"
```

```
    LOGICAL_NEGATION,          //"!"
    BITWISE_AND,               //"&"
    BITWISE_OR,                //"|"
    BITWISE_NEGATION,          //"~"
    BITWISE_XOR,               //"^"
    BITWISE_LSHIFT,            //"<<"
    BITWISE_RSHIFT,            //">>"
    QUESTION_MARK,             //"?"
    COLON,                     //":"
    SEMICOLON,                 //";"
    LEFT_SQUARE_BRACKET,       //"["
    RIGHT_SQUARE_BRACKET,      //"]"
    LEFT_PARENTHESE,           //"("
    RIGHT_PARENTHESE,          //")"
    LEFT_BRACE,                //"{"
    RIGHT_BRACE,               //"}"
    DOT,                       //"."
    COMMA,                     //","
    ARROW,                     //"->"
    ANNOTATION,                //注释

    //关系运算符属性
    GREATER,                   //">"
    GREATER_EQUAL,             //">="
    LESS,                      //"<"
    LESS_EQUAL,                //"<="
    EQUAL,                     //"=="
    UNEQUAL,                   //"!="

    //赋值运算符属性
    SIMPLE_EQUAL,              //"="
    PLUS_EQUAL,                //"+="
    MINUS_EQUAL,               //"-="
    MULTIPLY_EQUAL,            //"*="
    DIVIDE_EQUAL,              //"/="
    MOD_EQUAL,                 //"%="
    AND_EQUAL,                 //"&="
    OR_EQUAL,                  //"|="
    XOR_EQUAL,                 //"^="
    LSHIFT_EQUAL,              //"<<="
    RSHIFT_EQUAL,              //">>="
};
```

2.2.3 主函数变量

```
ifstream program;  //程序输入流
```

```
vector<struct token> token_stream;  //记号流
vector<string> id_list;  //标志符表
vector<string> str_list;  //字符串表
int line_num = 0;  //行数
int char_num = 0;  //字符数
vector<int> word_type_num(WORD_TYPE_AMOUNT);  //各记号类型数量
```

2.3 函数设计

inline bool is_digit(char ch)
功能：判断字符是否为数字
参数：char ch - 指定字符
返回：是否为数字

inline bool is_letter(char ch)
功能：判断字符是否为字母
参数：char ch - 指定字符
返回：是否为字母

inline char get_char(int& char_num, ifstream& program)
功能：读取一个字符
参数：int& char_num - 字符数
ifstream& program - 程序输入流
返回：读取的字符

inline void retract(int& char_num, ifstream& program)
功能：回退一个字符
参数：int& char_num - 字符数
ifstream& program - 程序输入流

inline void error(const string& str, const int& line_num)
功能：错误处理
参数：const string& str - 不符合词法规则的字符串
const int& line_num - 当前行数

int reserve(const string& str)
功能：二分搜索 str 在 KEYWORD_LIST 的位置
参数：const string& str - 指定字符串
返回：若搜索到返回位置，否者返回-1

int table_insert(vector<string>& table, const string& str)
功能：搜索 str 在 table 的位置，若搜索到返回位置，否者插入到表格末尾
参数：vector<string>& table - 字符串表
const string& str - 指定字符串
返回：返回字符串所在位置

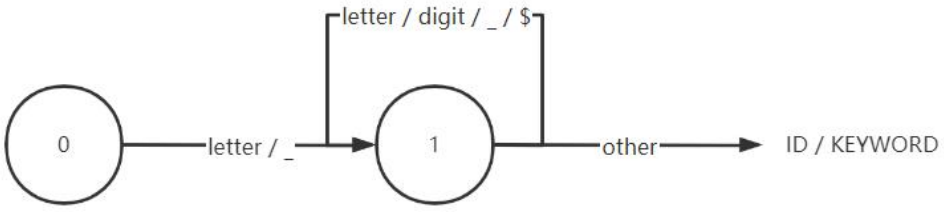
```
void word_analysis(vector<struct token>& token_stream, vector<string>& id_list,
vector<string>& str_list, vector<int>& word_type_num, const int& line_num, const
word_type& type, const string& buf = "", const int& num_base = 10)
```
功能：分析记号具体属性，并将其加入各类表格
参数：`vector<struct token>& token_stream` - 记号流
`vector<string>& id_list` - 标志符表
`vector<string>& str_list` - 字符串表
`vector<int>& word_type_num` - 各记号类型数量
`int& line_num` - 行数
`const word_type& type` - 记号类型
`const string& buf` - 需要判断属性的字符串
`const int& num_base` - 数字的进制

```
void lexical_analysis(vector<struct token>& token_stream, vector<string>& id_list,
vector<string>& str_list, int& line_num, vector<int>& word_type_num, int& char_num,
ifstream& program);
```
功能：对输入程序进行词法分析，输出对应记号流，统计源程序中的语句行数、各类单词的个数、以及字符总数，同时检查源程序中存在的词法错误，并报告错误所在的位置
参数：`vector<struct token>& token_stream` - 记号流
`vector<string>& id_list` - 标志符表
`vector<string>& str_list` - 字符串表
`int& line_num` - 行数
`vector<int>& word_type_num` - 各记号类型数量
`int& char_num` - 字符数
`ifstream& program` - 程序输入流

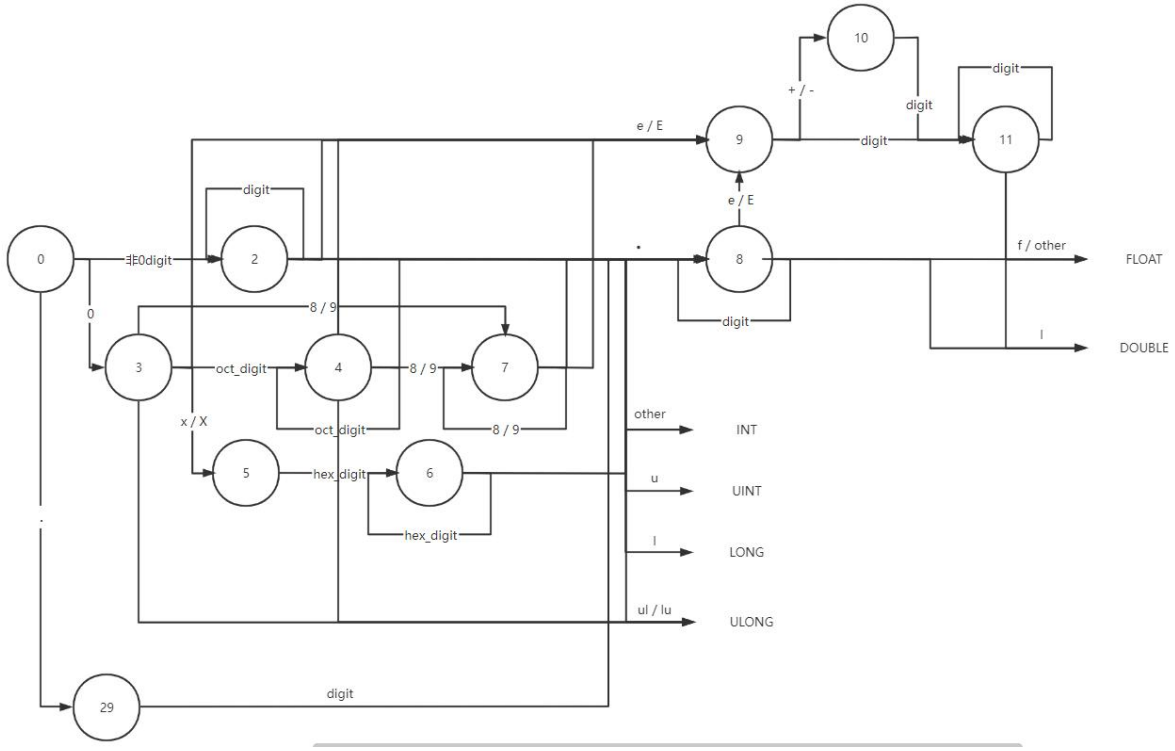
2.4 详细设计

2.4.1 状态转移图分析记号类型

（1）标志符和关键字



标志符以字母或_为首，由字母、数字、_、$组成。
（2）整型常量和实型常量

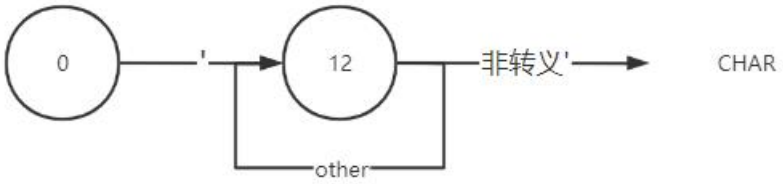


较为复杂的地方在于整型常量的判断和实型常量的判断。

整型常量具有十进制，八进制和十六进制三种表示形式，十进制表示不能以 0 开头，八进制以 0 开头同时只含有 0 到 7 的数，若含有其他数则必须为实型常量，十六进制以 0x 开头，三种表示形式都允许以 u 和 l 组合结尾，分别代表无符号和长整型。

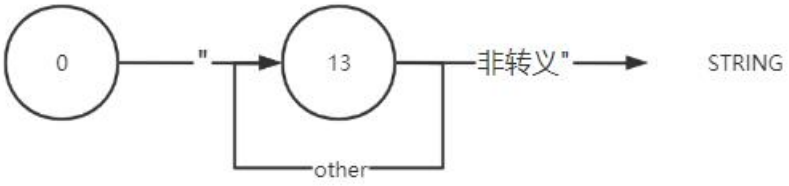
实型常量允许小数点左右有一边没有数字，允许用科学计数法表示，允许以 f 或 l 结尾，分别表示单精度和双精度。

（3）字符常量

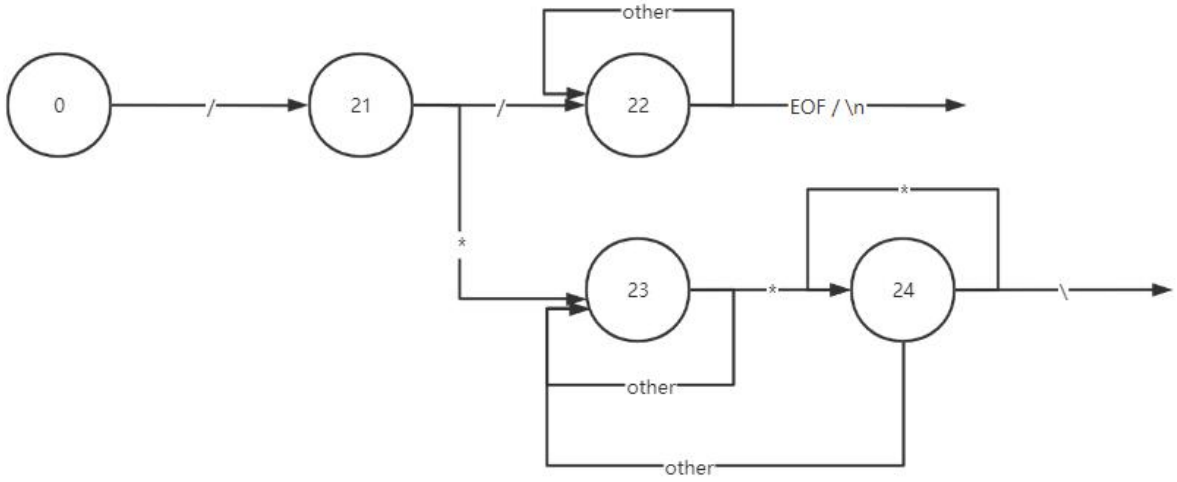


只要两个非转移单引号包围的字符串都作为字符常量，具体属性和是否合法放在分析记号属性中分析。

（4）字符串常量



（5）注释



（6）其余符号

其余符号状态转移较简单，这里就不再列出。

2.4.2 分析记号属性

在分析出记号类型后需要分析记号的对应属性。

（1）标志符和关键字

查找标志符在关键字表中的位置，若找到则将关键字表中的位置作为属性，否则查找其在标志符表中的位置，若没找到插入该标志符，最后将标志符表中的位置作为属性。

（2）整型常量和实型常量

将字符串转换为对应类型的数作为属性。

（3）字符常量

若第一个字符非\，将第一个字符作为字符常量属性，否则需要分析转义字符，并将对应的单字节十进制数作为字符常量的属性。

（4）字符串常量

查找字符串在字符串表中的位置，若未找到插入该字符串，最后将字符串表中的位置作为属性。

（5）赋值运算符和关系运算符

由于赋值运算符和关系运算符归为了一类，因此需要将具体的运算符类型作为记号属性。

（6）其余符号均不设置属性。

三、LEX 程序设计说明

3.1 设计思路

lex 程序设计思路基本与 C++相同，只需要将词法分析中状态转移表的实现部分修改为 lex 的正规文法，并将 C++中的数据结构使用 C 语言实现。但由于时间限制，现采用更简单的实现方式。识别出一个记号后就将其类型和读取的字符输出，不再分析并存储其属性。

3.2 正规文法设计

```
delim [ \t]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id ({letter}|_)({letter}|{digit}|_|\$)*
keyword (auto)|(break)|(case)|(char)|(const)|(continue)|(default)|(do)|
(double)|(else)|(enum)|(extern)|(float)|(for)|(goto)|(if)|(int)|(long)|
(register)|(return)|(short)|(signed)|(sizeof)|(static)|(struct)|(switch)|
(typedef)|(union)|(unsigned)|(void)|(volatile)|(while)
hex_digit [0-9a-fA-F]
hex_digits (0x{hex_digit}+)
octal_digit [0-7]
octal_digits (0{octal_digit}*)
decimal_digits ([1-9]{digit}*)
int {hex_digit}|{octal_digits}|{decimal_digits}
uint {int}u
long {int}l
ulong {int}((ul)|(lu))
real_num
(({digit}+E[\+\-]?{digit}+)|(({digit}+\.{digit}*)|(\.{digit}+))(E[\+\-]?{digit
}+)?)
float {real_num}f?
double {real_num}l
char \'[^']*\'
str \"[^"]*\"
relop (\>)|(\>=)|(\<)|(\<=)|(==)|(!=)
assop (=)|(\+=)|(\-=)|(\*=)|(\/=)|(\%=)|(&=)|(\|=)|(\^=)|(\<\<=)|(\>\>=)
singleline_annotation "//"(.*)
multiline_annotation "/*"([^*]*[*]+)+"/"
other .
```

其他符号由于较为简短，不设置非终结符，直接作为终结符放在翻译规则部分识别。

四、程序测试

4.1 测试案例

```
struct st
{
    int a;
};
```

```
int main()
{
    char c = '\n';
    int i = 12;
    //这是一条单行注释
    unsigned int ui = 017u;
    long l = 0x12al;
    unsigned long ul = 14ul;
    float f = 1.;
    struct st s1;
    struct st* s2 = &s1;
    s2->a = 15;
    /*
    这是
    一条
    多行
    注释
    */
    double d = .12E+10l;
    if (c <= '\xff')
    {
        c = '\x100';
    }
    else if (!i || l == 12)
    {
        i += 09;
        ui += 0xr;
        f = 1.2Ea;
        d = 1.3E+a;
    }

    i++;
    ui << 1;
    f = (i == 1) ? 1.0 : 0;
    printf("This is a string");
    return 0;
}
```

该测试用例包含部分可能遇到的情况，一共有 5 处错误，第 27 行’\x100’超过 255 字符常量过大，第 31 行 09 十进制整型常量不能以 0 开头，第 32 行 0x 后面未跟数字，第 33 行 1.2E 后面未跟数字或+-，第 34 行 1.3E+后面未跟数字。

4.2 测试结果

4.2.1 C++程序测试结果

首先输出出错位置和出错字符串，与预想的结果相同。

```
Designed by CHEN YU, built: Oct  3 2022 18:25:11
error 27: \x100
error 31: 09
error 32: 0x
error 33: 1.2E
error 34: 1.3E+
```

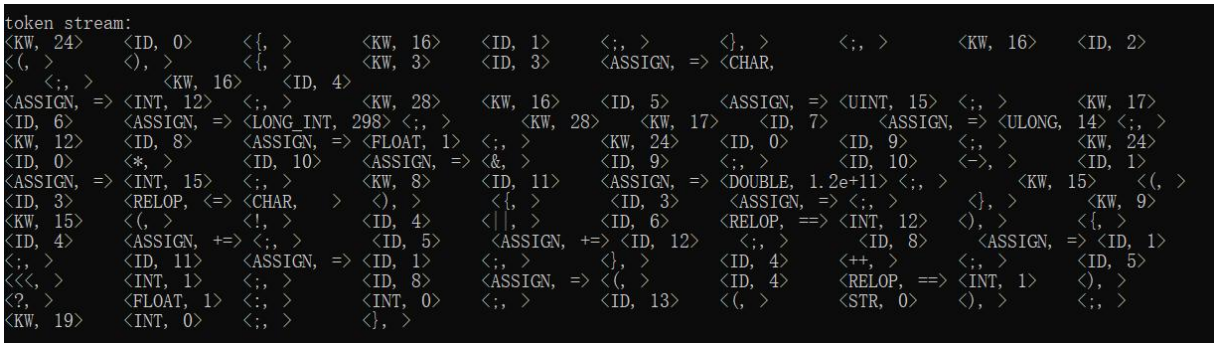
接着输出关键字表、标志符表和字符串表。

```
keyword list:
0         auto
1         break
2         case
3         char
4         const
5         continue
6         default
7         do
8         double
9         else
10        enum
11        extern
12        float
13        for
14        goto
15        if
16        int
17        long
18        register
19        return
20        short
21        signed
22        sizeof
23        static
24        struct
25        switch
26        typedef
27        union
28        unsigned
29        void
30        volatile
31        while
```

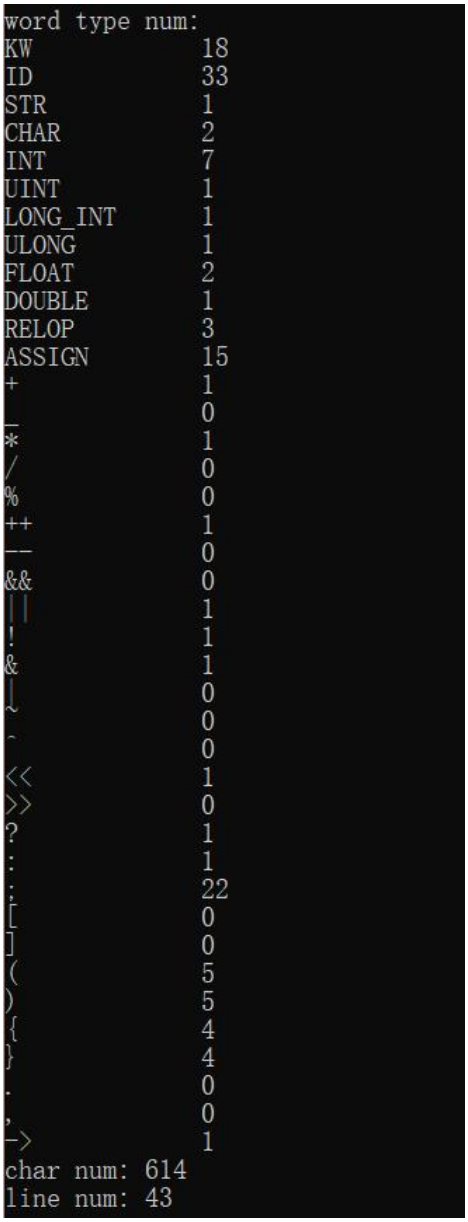
```
ID list:
0         st
1         a
2         main
3         c
4         i
5         ui
6         l
7         ul
8         f
9         s1
10        s2
11        d
12        r
13        printf

string list:
0         This is a string
```

然后开始输出记号流，记号以<类型，属性>的格式输出，图中第二行 CHAR 类型的属性为\n，因此输出后实现了换行。



最后输出各记号类型数量，行数和字符总数



经过查验，程序能够正常执行并得到正确结果，能够跳过注释，能够进行错误检测并输出出错位置。

4.4.2 lex 程序测试结果

首先，同时输出记号流和错误信息。

```
<KW, struct>    <ID, st>        <{, >   <KW, int>       <ID, a> <;, >   <}, >   <;, >   <KW, int>       <ID, main>
        <(,     <), >   <{, >   <KW, char>      <ID, c> <ASSIGN_OPERATOR, =>    <CHAR, '\n'>    <;, >   <KW, int>
        <ID, i> <ASSIGN_OPERATOR, =>    <INT, 12>       <;, >   <KW, unsigned>  <KW, int>       <ID, ui>        <ASSIGN_
OPERATOR, =>    <UINT, 017u>    <;, >   <KW, long>      <ID, l> <ASSIGN_OPERATOR, =>    <LONG, 0x12al>  <;, >   <KW, uns
igned>  <KW, long>      <ID, ul>        <ASSIGN_OPERATOR, =>    <ULONG, 14ul>   <;, >   <KW, float>     <ID, f> <ASSIGN_
OPERATOR, =>    <FLOAT, 1.>     <;, >   <KW, struct>    <ID, st>        <ID, s1>        <;, >   <KW, struct>    <ID, st>
        <*, >   <ID, s2>        <ASSIGN_OPERATOR, =>    <&, >   <ID, s1>        <;, >   <ID, s2>        <->, >  <ID, a>
        <ASSIGN_OPERATOR, =>    <INT, 15>       <;, >   <KW, double>    <ID, d> <ASSIGN_OPERATOR, =>    <DOUBLE, .12E+10
1>      <;, >   <KW, if>        <(,     <ID, c> <RELATION_OPERATOR, <=> <CHAR, '\xff'>  <), >   <{, >   <ID, c> <ASSIGN_
OPERATOR, =>    <CHAR, '\x100'> <;, >   <}, >   <KW, else>      <KW, if>        <(,     <!, >   <ID, i> <||, >  <ID, 1>
        <RELATION_OPERATOR, ==> <INT, 12>       <), >   <{, >   <ID, i> <ASSIGN_OPERATOR, +=>   <INT, 0>       <INT, 9>
        <;, >   <ID, ui>        <ASSIGN_OPERATOR, +=>   <INT, 0>        <ID, xr>        <;, >   <ID, f> <ASSIGN_OPERATOR
, =>    <FLOAT, 1.2>    <ID, Ea>        <;, >   <ID, d> <ASSIGN_OPERATOR, =>    <FLOAT, 1.3>    <ID, E> <+, >   <ID, a>
        <;, >   <}, >   <ID, i> <++, >  <;, >   <ID, ui>        <<<,    <INT, 1>        <;, >   <ID, f> <ASSIGN_OPERATOR
, =>    <(,     <ID, i> <RELATION_OPERATOR, ==> <INT, 1>        <), >   <?, >   <FLOAT, 1.0>    <:, >   <INT, 0>
        <;, >   <ID, printf>    <(,     <STRING, "This is a string">    <), >   <;, >   <KW, return>    <INT, 0>
        <;, >   <}, >
```

由于 lex 分析程序不对字符常量作属性分析，因此'0x100'的错误无法发现。同时，lex 分析程序将 09 分析为两个整型常量 0 和 9，将 0xr 分析为一个整型常量 0 和一个标志符 xr，将 1.2Ea 分析为一个实型常量 1.2 和一个标志符 Ea，将 1.3E+a 分析为一个实型常量 1.3、一个标志符 E、一个加号+和一个标志符 Ea，因此 5 个错误均为被查出。该 lex 分析程序只能分析出非法字符，而不能分析不全的记号。

最后输出各记号类型数量，行数和字符总数

```
word type num:
KW      18
ID      34
STR     1
CHAR    3
INT     10
UINT    1
LONG    1
ULONG   1
FLOAT   4
DOUBLE  1
RELOP   3
ASSIGN  15
+       1
-       0
*       1
/       0
%       0
++      1
--      0
&&      0
||      1
!       1
&       1
|       0
~       0
^       0
<<      1
>>      0
?       1
:       1
);      22
[       0
]       0
(       5
)       5
{       4
}       4
.       0
,       0
->      1
char num: 614
line num: 43
```

除了由于将本应出错的字符串拆成多种记号，部分记号类型数量比 C++程序多，其余均可正常执行，并得到正确结果。

五、实验总结

本次课程实验中，本人同时使用 C++程序和 lex 程序实现了词法分析。代码均由本人一人完成，修修补补共写了 1000 多行代码，最终将笔者能够想到的功能全部实现在实验过程中也遇到了不少的问题。

首先是记号种类的划分，个人理解划分记号种类的目的是为了减少语法分析的工作量，因此应尽可能将使用场景相同的记号种类划分在一起，最终发现只有关系运算符和赋值运算符能够划分在一起。

然后是对于常量十进制、八进制和十六进制的分析，这一部分是整个实验耗时最多的部分，再一次次尝试各种组合是否能被 gcc 编译通过的同时，逐步完善常量的识别，包括分析各种无法编译通过的组合。

接着是如何存储常量属性的问题，常量包含整型，单精度浮点型，双精度浮点型等多种类型，只能采用 union 存储。但由于 union 难以存储 string 类型，因此设立字符串表，并将字符串位置作为其属性。

最后就是代码结构上的问题，为了建立起清晰易读的代码架构耗费了不少心思，最后采用先分析记号类型，后分析记号属性的方式建立程序架构。

总的来说，从本次试验中学习到不少东西，首先是对于词法分析有了更深的理解，其次在不断挑战 gcc 编译器的识别边界的同时对于 c 语言有了更深的了解，最后提升了自己的代码水平，采用自顶向下的方式构建代码，基本写完就能跑，减少了不少调试时间。

不过还有一点不足就是由于时间关系，lex 程序完善的并不充分，完成度远不及 C++程序，仅当对于 lex 使用的一次实践。