C 语言词法分析程序的设计与实现

目录

实验内容及要求

- 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。
- 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。

实验环境

环境/工具	说明	版本/网址
Windows11	OS	220916
TDM-gcc	编译器	10.3.0
Flex	LEX	2.5.4
FSM simulator	NFA 绘制	http://ivanzuzak.info/noam/webapps/fsm_simulator/

程序设计说明

一、总体设计

1.1、总体思路

通过良好的高层封装实现对各单词符号的提取和处理，将各类记号的读取分为多个模块，降低耦合性。同时模仿 Java 语言中 `java.util.Scanner` 类的读取模式 `next()`，调用 `nextSymbol()`函数进行单词符号获取，实现流畅且清晰的单词符号处理。

1.2、类设计

1.2.1、Symbol 类

- **类说明**

  记录单词符号的**详细信息**，包括：
  - 单词符号的**内容**
  - 单词符号的**记号类型**
  - 单词符号在原文件中的**行列位置**

  单词符号的记号类型包括：

```
enum class Type
{
    END,            // 结束符
    ERROR,          // 错误符
    KEYWORD,        // 关键字
    IDENTIFIER,     // 标识符
    NUMBER,         // 数字
    OPERATOR,       // 操作符
    LIMIT,          // 界限符
    STRING,         // 字符
    COMMENT,        // 注释
};
```

- **主要函数**

```
void append(char ch)      // 向字符数组中添加字符
char rollBack()           // 从字符数组中回退一个字符
Type attr()               // 获取记号类型（属性）
char operator[](int pos) // 重载[]运算符直接读取类字符数组内容
```

- **源代码**

1.2.2、Scanner 类

- **类说明**

  读取输入的文件流，通过词法分析，以**记号**为单位获取源文件信息。
  词法分析支持:
  - 标识符（英文）
  - 数字，包括十六进制、八进制、二进制前缀以及各类后缀
  - 操作符
  - 界限符
  - 字符串
  - 注释

- **主要函数**

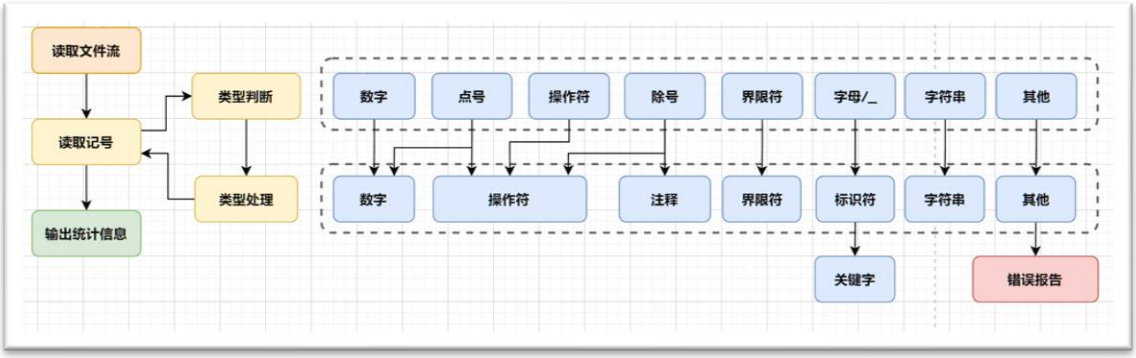
```
char _nextChar();           // 从缓冲区中读取字符
void _rollBack();           // 缓冲区指针回退一个字符
static bool _isXXX(char);  // 判断字符类型
void _procXXX ();           // 各类记号处理
void _reportError(ErrorType e);      // 报告错误信息

bool open(const char* file_name);    // 打开文件
Symbol nextSymbol();                 // 读取下一记号
```

- **源代码**

```
class Scanner
{
    const static int __buffer_length = 64;  // 缓冲区长度
    const static int __buffer_mid = __buffer_length / 2;
private:
    enum class ErrorType{    //*错误类型
        INVALID_WORD,        // 非法标识符
        EXPECTED_END,        // 需要字符串/注释结尾
        INVALID_FLOAT,       // 浮点无效
        INVALID_HEX,         // 十六进制无效
        INVALID_OCTAL,       // 八进制无效
        INVALID_BINARY,      // 二进制无效
        CHAR_OVERFLOW,       // 字符常量超限
    };
    enum class PrefixType{   //*前缀类型
        DECIMAL,             // 十进制
        HEX,                 // 十六进制
        BINARY,              // 二进制
        OCTAL,               // 八进制
        _NAN,                // 非数字
    };
    enum class NumType{      //*数字类型
        INT,                 // 整形
        FLOAT,               // 浮点
    };
    // File stream
    std::string _file_name;              // 文件名
    std::ifstream _ifs, _ifs_line;       // 文件流(字符文件流)(行文件流)
    // String buffer
    char _line_buffer[256] = {0};        // 行缓冲区
    char _file_buffer[__buffer_length] = {0};   // 字符缓冲区
    int _ptr_buffer = -1;                // 字符缓冲区指针
    // Counter
    int _type_counter[9] = {0};          // 记号类型计数器
    int _total_char = 0;                 // 字符计数器
    int _roll_back_times = 0;            // 指针回退计数器
    // Current Indicator
    Indicator _cur_ind;                  // 行列指示器
    // Read symbol
    Symbol _token;                       // 记号缓冲
```

```
    // 字符类型判断
    static bool _isNum(char);        // 数字
    static bool _isLetter(char);     // 字母/下划线
    static bool _isLimit(char);      // 界限符
    static bool _isOperator(char);   // 操作符
    static bool _isSpace(char);      // 空格/换行
    static bool _isHex(char);        // 十六进制
    static bool _isSuffix(char);     // 后缀

    void _rollBack();            // 字符指针回退
    char _nextChar();            // 字符读取

    void _procDot();             // 点号处理 (→数字/操作符)

    void _procNumber();          // 数字处理
    PrefixType _procPrefix();    // 前缀处理
    void _procSuffix(NumType);   // 后缀处理

    void _procIdentifier();     // 标识符处理
    void _procLimit();          // 界限符处理

    void _procDiv();            // 除号处理  (→操作符/注释)
    void _procOperator(char);   // 操作符处理
    void _procString(char start);     // 字符串处理
    void _procAnnotaion(char start); // 注释处理

    void _procError(ErrorType);          // 错误处理
    void _reportError(ErrorType e);     // 错误报告

public:

    bool open(const char* file_name);    // 打开文件
    void close();                        // 关闭文件
    int totalChar();                     // 字符总数
    int totalRow();                      // 行总数
    void typeStatistic(int[]);           // 记号计数
    Symbol nextSymbol();                 // 下一记号
};
```

1.3、流程设计



二、主要模块设计

2.1、字符缓冲区

2.1.1、设计模式

参考教材中的双段缓冲区进行设计，采用单指针，通过操作记号类 Symbol 的 **append(char)** 和 **rollBack()** 方法完成**字符同步**。

单指针方式在回退操作位于边界附近时容易触发二次文件读取，因此设计有**回退计数器**，只有在回退计数器为 0 时才进行文件读取。

2.1.2、源代码

```
41  {
42      _ptr_buffer = (_ptr_buffer + 1) % __buffer_length;
43      if(_roll_back_times != 0){  // 有回退记录，不进行文件读取
44          --_roll_back_times;     // 回退消除
45      }else{
46          if(_row_flag){
47              _ifs_line.getline(_line_buffer,__buffer_length);    // 行缓冲，用于错误报告
48          }
49          if(_ptr_buffer == __buffer_mid - 1 && !_ifs.eof())
50          {
51              memset(&_file_buffer[__buffer_mid], EOF, __buffer_mid); //段重置
52              _ifs.read(&_file_buffer[__buffer_mid], __buffer_mid);
53          }
54          else if(_ptr_buffer == __buffer_length - 1 && !_ifs.eof())
55          {
56              memset(&_file_buffer[0], EOF, __buffer_mid);    // 段重置
57              _ifs.read(_file_buffer, __buffer_mid);
58          }
59      }
60
61      if(_row_flag){
62          _cur_ind.nextRow();     // 行计数
63          _row_flag = false;
64      }
65      else
66          _cur_ind.nextCol();     // 列计数
67
68      char ch = _file_buffer[_ptr_buffer];
69      if(ch != ' ' && ch != '\n' && ch != '\t' && ch != EOF)
70          ++_total_char;  // 字符计数
71      if(ch == '\n')
72          _row_flag = true;
73      return ch;
74  }
```

2.2、类型判断

参考设计流程图，在读取到第一个字符时进行**类型初判**：

```
Symbol Scanner::nextSymbol()
{
    _token.clear();

    char ch;
    do
        ch = _nextChar();
    while(ch == ' ' || ch == '\n'); // 跳过开始读取时的空格/换行
    _token.indicator = _cur_ind;
    if(ch == EOF)           // 文件结束
        return _token.end();
    _token.append(ch);
    if(ch == '.')           // 点号（→操作符/数字）
        _procDot();
    else if(_isNum(ch))     // 数字
        _procNumber();
    else if(_isLetter(ch))  // 字母/_
        _procIdentifier();
    else if(ch == '/')      // 除号（→操作符/注释）
        _procDiv();
    else if(ch == '\"' || ch == '\'')   // 字符串
        _procString(ch);
    else if(_isLimit(ch))   // 界限符
        _procLimit();
    else if(_isOperator(ch))    // 操作符
        _procOperator(ch);
    else                        // 非法字符
        _procError(ErrorType::INVALID_WORD);

    ++_type_counter[(int)_token.attr()];    // 类型计数
    if(_token.attr() == Symbol::Type::COMMENT)
        return nextSymbol();    // 跳过注释
    return _token;
}
```

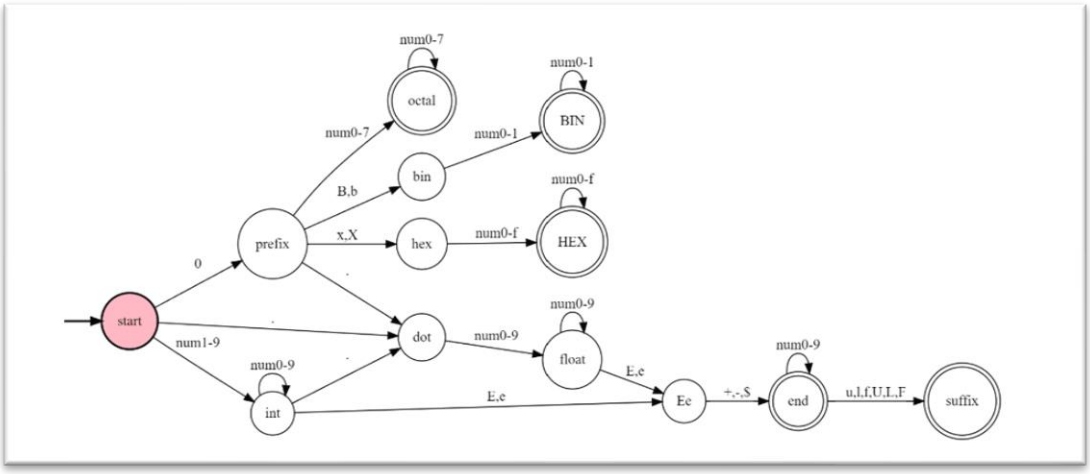
2.3、数字处理

2.3.1、设计模式

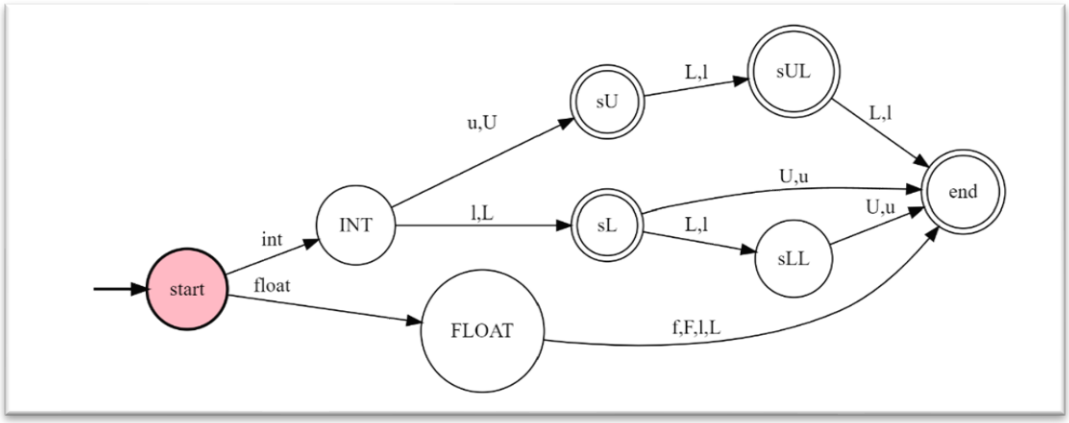
数字处理模块支持 C 语言前缀和后缀，主要分为三部分处理：**前缀处理，中间数处理，后缀处理**。

2.3.2、有限状态自动机

图示为前缀和中间数处理的有限状态自动机，由于大小限制，**对于需要回退的非接收结果未标出。**



图示为后缀处理的有限状态自动机，为上图后缀处理的细化。其中的 **float/int** 状态是在数字处理阶段定义，目的是**支持词法分析阶段的后缀错误检测**。同样的，由于大小限制，**对于需要回退的非接收结果未标出**。



2.3.3、部分源代码

通过该自动机设计 C++源代码，本次多采用 if-else 结构辅以部分标志位来完成。

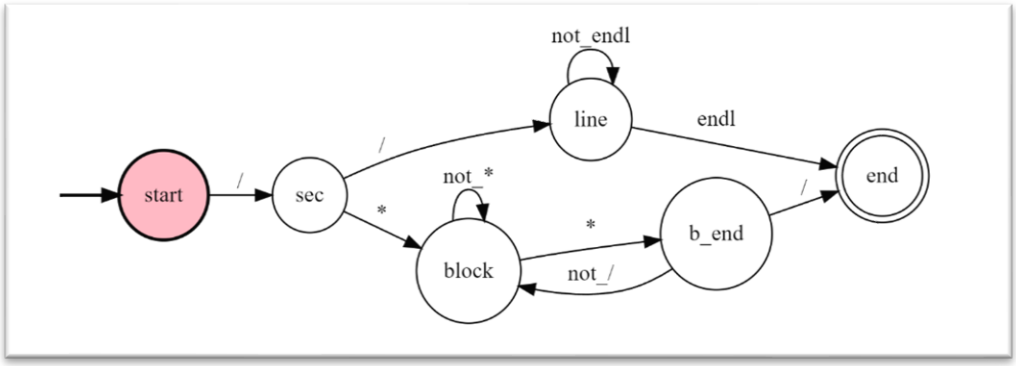
```cpp
/* 数字处理 */
void Scanner::_procNumber()
{
    _token.setAttr(Symbol::Type::NUMBER);
    int type = 0;
    int part_int = 1, part_float = 2, part_e = 3, part_end = 4;
    int part = part_int;
    PrefixType ptype = _procPrefix();
    NumType ntype = NumType::INT;
    if(ptype == PrefixType::_NAN)   return;
    for(char ch;;)
    {
        ch = _nextChar();
        if(ch == EOF) return;
        if(_isSuffix(ch)){
            _rollBack();
            return _procSuffix(ntype);
        }
        if(part == part_int)
        {
            if(ptype == PrefixType::BINARY && !(ch == '0' || ch == '1')){
                if(_isNum(ch))
                    _procError(ErrorType::INVALID_BINARY);
                return _rollBack();
            }else if(ptype == PrefixType::HEX && !_isHex(ch)){
                if(_isHex(ch))
                    _procError(ErrorType::INVALID_HEX);
                return _rollBack();
            }else if(ptype == PrefixType::OCTAL && !((ch >= '0' && ch <= '7') || ch == '.')){
                if(_isNum(ch))
                    _procError(ErrorType::INVALID_OCTAL);
                return _rollBack();
            }
            // Decimal
            if(ch != '.' && ch != 'E' && ch != 'e' && !_isNum(ch))
                return _rollBack();
            part = (ch == '.' ? part_float : ((ch == 'e' || ch == 'E') ? part_e : part_int));
        }
        else if(part == part_float)
```

2.4、注释处理

2.4.1、设计模式

注释处理基于类型判断模块的除号判断，通过后续字符判断其是注释还是操作符。

2.4.2、有限状态自动机



2.4.3、部分源代码

对于状态机图中 block 状态和 b_end 状态的转移，在 C++源代码中通过 end 标志位进行处理。

```
68  {
69      _token.setAttr(Symbol::Type::COMMENT);
70      _token.append(type);
71      if(type == '/')
72      {
73          for(char ch;;)
74          {
75              ch = _nextChar();
76              if(ch == '\n' || ch == EOF)
77                  return;
78          }
79      }
80
81      bool end = false;
82      for(char ch;;)
83      {
84          ch = _nextChar();
85          if(ch == '*')
86              end = true;
87          else if(end){
88              if(ch == '/')   return;
89              end = false;
90          }
91          else if(ch == EOF)
92          {
93              _token.indicator.setCol(_cur_ind.col());
94              return _reportError(ErrorType::EXPECTED_END);
95          }
96      }
97  }
```

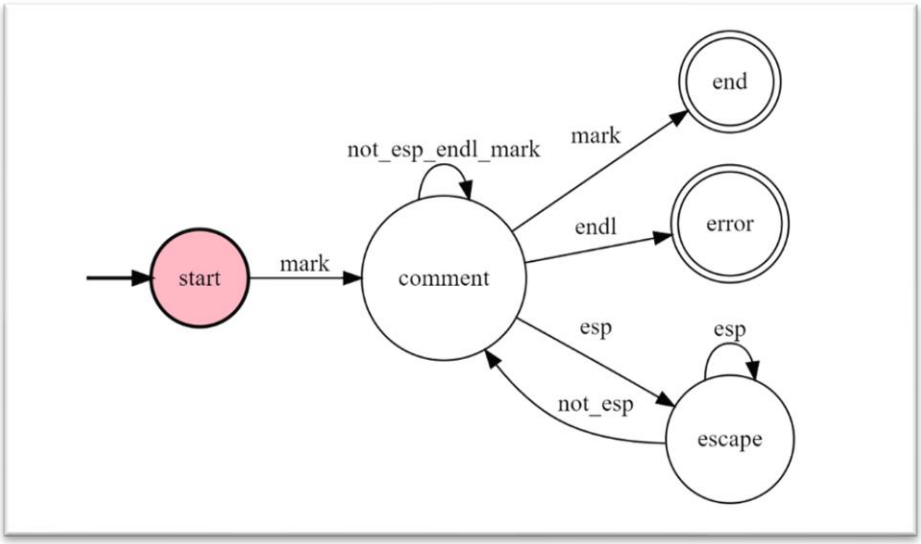
2.5、字符串处理

2.5.1、设计模式

字符串处理基于类型判断模块的引号判断，支持转义换行以及单引号内字符个数超限警告。在实际开发过程中，应当考虑 Unicode 字符以及各类不同编码字符的处理，尤其是对于有字符数量限制的单引号字符。不过考虑到本程序设计的重点，本次实验未实现 Unicode 字符的解析。

2.5.2、有限状态自动机

下图中 mark 为前后匹配的双引号或单引号，esp 为转义字符'\'



2.5.3、源代码

对于状态机图中 comment 状态和 escape 状态的转移，即多个转义符的识别，在 C++源代码中通过 escape 标志位进行处理。

```
void Scanner::_procString(char start)
{
    _token.setAttr( Symbol::Type::STRING);
    bool escape = false;
    for(;;)
    {
        char ch = _nextChar();
        if(ch == ' ' || ch == '\n' || ch == '\t')
            ++_total_char;          // 字符串中的空字符计入字符总数
        _token.append(ch);

        if(escape){
            escape = (ch == '\\' ? true:false);
            if(ch == '\n'){         // 隐藏token中的转义\n
                _token.rollBack();
                _token.rollBack();
            }
        }
        else if(!escape && ch == '\\')
            escape = true;
        else if(ch == EOF || (!escape && ch == '\n'))
        {
            _token.rollBack();
            _token.indicator.setCol(_cur_ind.col());
            ++_type_counter[(int)Symbol::Type::ERROR];
            return _reportError(ErrorType::EXPECTED_END);
        }
        else if(!escape && ch == start){
            if(start == '\''){
                int l = _token.content().length();
                if(l > 4 || (l == 4 && _token[1] != '\\'))
                    return _reportError(ErrorType::CHAR_OVERFLOW);
            }
            return;
        }

    }
}
```

2.6、标识符处理

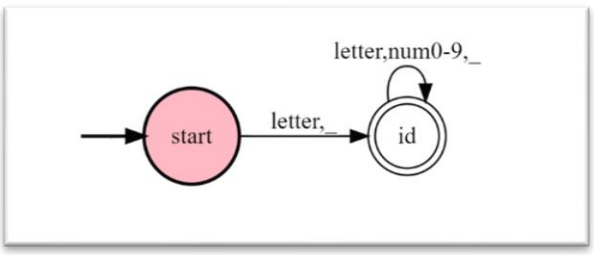
2.6.1、设计模式

标识符处理基于 C 语言标识符定义要求：必须由字母（区分大小写）、数字、下划线组成，且标识符的第一个字符不可以是数字。该版本设计中未支持 UTF-8 模式，即未支持中文标识符。

在标识符成功识别后，会通过扫描函数判断该标识符是否为 C 语言关键字。

在实际开发过程中，应当考虑 Unicode 字符以及各类不同编码字符的处理，不过考虑到本程序设计的重点，本次实验未实现 Unicode 字符的解析。

2.6.2、有限状态自动机



2.6.3、源代码

```
206  void Scanner::_procIdentifier()
207  {
208      _token.setAttr(Symbol::Type::IDENTIFIER);
209      for(char ch;;)
210      {
211          ch = _nextChar();
212          if(ch == EOF)
213              break;
214          if(_isLetter(ch) || _isNum(ch))
215              _token.append(ch);
216          else{
217              _rollBack();
218              break;
219          }
220      }
221      if(__is_keyword(_token.content().c_str()))
222          _token.setAttr(Symbol::Type::KEYWORD);
223
224  }
225
```

2.7、操作符/界限符处理

2.7.1、设计模式

由于长度较为固定，操作符/界限符的识别采用单段 if-else 模式，辅以 rollBack()回退以支持不同长度的字符识别。

由于仅要求输出单词与记号，因此该设计中未区分不同操作符。

2.7.2、部分源代码

```
357  void Scanner::_procOperator(char start)
358  {
359      _token.setAttr(Symbol::Type::OPERATOR);
360      bool append_flag = true;
361      char ch = _nextChar();
362      if(ch == EOF)   return;
363      if(start == '-')
364          append_flag = (ch == '-' || ch == '=' || ch == '>');
365      else if(start == '*' || start == '/' || start == '!' || start == '%' || start == '^')
366          append_flag = (ch == '=');
367      else if(start == '>' || start == '<' || start == '=' || start == '|' || start == '&' || start == '+')
368          append_flag = (ch == '=' || ch == start);
369
370      if((start == '<' && ch == '<') || (start == '>' && ch == '>'))
371      {
372          _token.append(ch);
373          ch = _nextChar();
374          append_flag = (ch == '=');
375      }
376
377      return append_flag ? _token.append(ch) : _rollBack();
378  }
```

2.8、错误处理

2.8.1、设计模式

在前序记号处理的过程中，如果遇到错误，则会让字符缓冲区指针回退，同时立即进行错误处理并报告错误到控制台。后续将从回退的位置继续重新开始进行单词分析。

参考 **gcc/g++**的错误警告格式，在 Scanner 类中设计有行缓冲区，用于标明字符

缓冲区指针所在行，即同时也可以**标明错误所在行**。支持部分词法分析阶段可以发现的错误：

- 非法字符
- 字符串/注释结尾缺失
- 字符常量超限
- 浮点常量无效
- 十六进制错误
- 八进制错误
- 二进制错误

对于**浮点常量无效、十六进制/八进制/二进制错误，分析器仅给出警告信号**，不计入错误统计。

2.8.2、显示效果

```
scripts/origin.c:30:16: error: invalid character '#'
   30 |     int a$23 = #4;
      |                ^
scripts/origin.c:31:18: warning: invalid float number
   31 |     float b = 1.2UL, f=1lf;
      |                  ^
scripts/origin.c:32:16: warning: invalid hex number
   32 |     int c = 0x1.4, d=078, e=0b13;
      |                ^
```

2.8.3、部分源代码

```
void Scanner::_procError(Scanner::ErrorType type)
{
    if(type == ErrorType::CHAR_OVERFLOW || type == ErrorType::EXPECTED_END || type == ErrorType::INVALID_WORD)
        _token.setAttr(Symbol::Type::ERROR) ;
    _token.indicator.setCol(_cur_ind.col());    // 标明错误所在行列
    _reportError(type);                         // 打印错误
}

void Scanner::_reportError(ErrorType e)
{
    std::string msg = "",type = BOLDYELLOW "warning: " RESET;
    /* 错误类型 & 信息注入 */
    if(e == ErrorType::INVALID_WORD){
        msg = "invalid character '" BOLDWHITE + _token.content() + RESET "'";
        type = BOLDRED "error: " RESET;
    }
    else if(e == ErrorType::EXPECTED_END){
        std::string end = std::string(1,_token[0]);
        msg = "expected '" BOLDWHITE + (end == "*"?"*/":end) + RESET "'";
        type = BOLDRED "error: " RESET;
    }else if(e == ErrorType::INVALID_BINARY){
        msg = "invalid binary number";
    }else if(e == ErrorType::INVALID_FLOAT){
        msg = "invalid float number";
    }else if(e == ErrorType::INVALID_HEX){
        msg = "invalid hex number";
    }else if(e == ErrorType::INVALID_OCTAL){
        msg = "invalid octal number";
    }else if(e == ErrorType::CHAR_OVERFLOW){
        msg = "too many characters";
    }
    /* 高亮显示 */
    printf(BOLDWHITE "%s:%d:%d: " "%s" "%s\n", _file_name.c_str(),_token.indicator.row(), _token.indicator.col(),type.c_str(),  msg.c_str());
    printf("%5d |%s\n",_token.indicator.row(),_line_buffer);
    __print_space(6);
    printf("|");
    __print_space(_token.indicator.col()-1);
    printf(BOLDRED "^\n" RESET);
}
```

三、运行结果及分析

3.1、输入源程序

输入源程序包括**各类标识符、操作符识别测试**以及各类**错误测试**，具体如下：

Lexer > scripts > C origin.c > test

```c
struct  test
{
    double pt;
};

int main(){
    // short comment test
    /* short block comment test */
    /*
        long block comment test *\/ escpate test
    */
    //id test
    int _a0, b;
    //operator test
    struct test* t = malloc(sizeof(struct test));
    t->pt = .1e64;
    (*t).pt = 0.2;
    _a0 >>= 2;
    // suffix test
    int a_ = 10uL, b_ = 100uL;
    // num type test
    int c_ = 0x1e8, d_=070, e_ = 0b0011;
    float f_ = 1.3f, g_ = 1.1l;
    //string test
    printf("print test \"\n");
    char str[] = "string \\t\
        test\
    ";
    //error test
    int a$23 = #4;
    float b = 1.2UL, f=1lf;
    int c = 0x1.4, d=078, e=0b13;
    char str2[] = "no end test \"

    return 0;
}

```

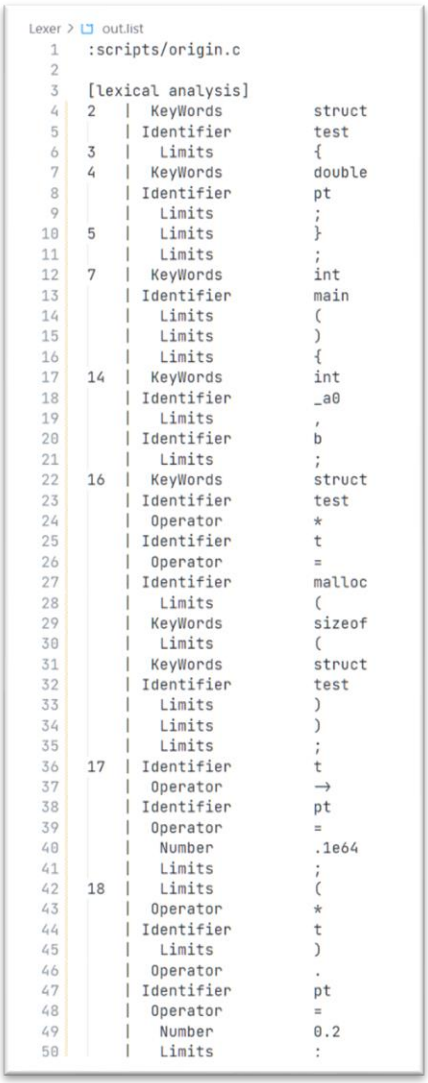
3.2、执行结果

将源文件以参数形式传递给词法分析程序处理，分析结果将同时在控制台和文件中输出。其中控制台主要用于显示错误信息和统计信息；文件中包括各记号的详细信息和统计信息

3.2.1、控制台输出

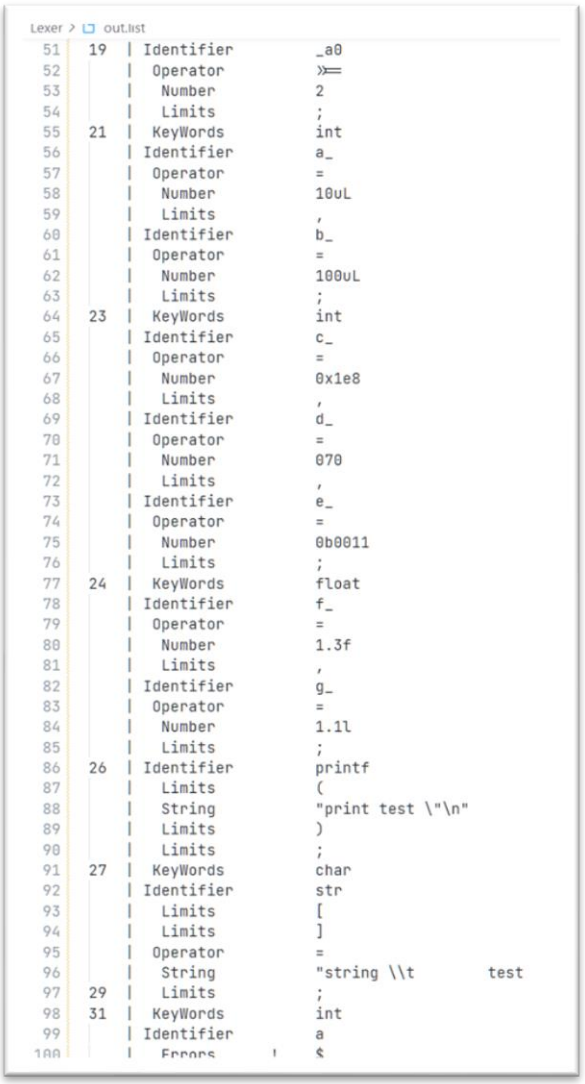
```
PS G:\Coding\CProject\Compile\Lexer> ./out.exe scripts/origin.c
scripts/origin.c:30:10: error: invalid character '$'
   30 |     int a$23 = #4;
      |          ^
scripts/origin.c:30:16: error: invalid character '#'
   30 |     int a$23 = #4;
      |                ^
scripts/origin.c:31:18: warning: invalid float number
   31 |     float b = 1.2UL, f=1lf;
      |                  ^
scripts/origin.c:32:16: warning: invalid hex number
   32 |     int c = 0x1.4, d=078, e=0b13;
      |               ^
scripts/origin.c:32:24: warning: invalid octal number
   32 |     int c = 0x1.4, d=078, e=0b13;
      |                        ^
scripts/origin.c:32:32: warning: invalid binary number
   32 |     int c = 0x1.4, d=078, e=0b13;
      |                                ^
scripts/origin.c:33:34: error: expected '"'
   33 |     char str2[] = "no end test \"
      |                                  ^
scripts/origin.c: statistic:
   [row]   37
   [char]  471
scripts/origin.c: types:
   [  Errors  ]   3
   [ KeyWords ]   16
   [Identifier]   32
   [  Number  ]   21
   [ Operator ]   23
   [  Limits  ]   42
   [  String  ]   2
PS G:\Coding\CProject\Compile\Lexer>
```

3.2.2、文件输出



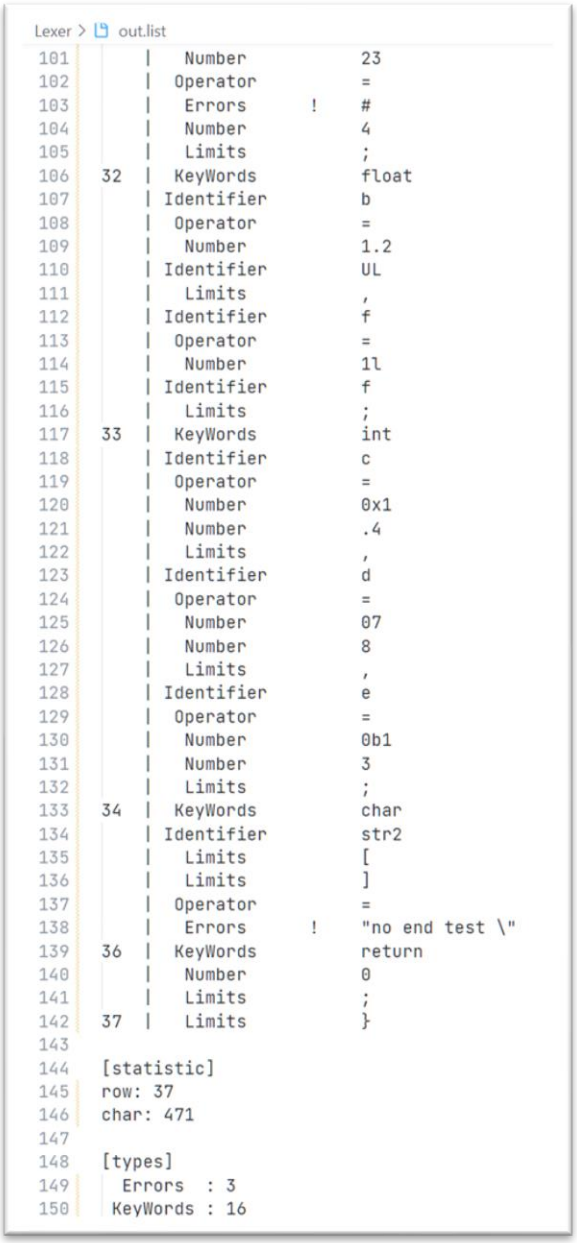
Lexer > out.list

```
:scripts/origin.c

[lexical analysis]
2   |  KeyWords          struct
    | Identifier         test
3   |   Limits           {
4   |  KeyWords          double
    | Identifier         pt
    |   Limits           ;
5   |   Limits           }
    |   Limits           ;
7   |  KeyWords          int
    | Identifier         main
    |   Limits           (
    |   Limits           )
    |   Limits           {
14  |  KeyWords          int
    | Identifier         _a0
    |   Limits           ,
    | Identifier         b
    |   Limits           ;
16  |  KeyWords          struct
    | Identifier         test
    |  Operator          *
    | Identifier         t
    |  Operator          =
    | Identifier         malloc
    |   Limits           (
    |  KeyWords          sizeof
    |   Limits           (
    |  KeyWords          struct
    | Identifier         test
    |   Limits           )
    |   Limits           )
    |   Limits           ;
17  | Identifier         t
    |  Operator          →
    | Identifier         pt
    |  Operator          =
    |   Number           .1e64
    |   Limits           ;
18  |   Limits           (
    |  Operator          *
    | Identifier         t
    |   Limits           )
    |  Operator          .
    | Identifier         pt
    |  Operator          =
    |   Number           0.2
    |   Limits           :
```



Lexer > out.list

```
19  | Identifier         _a0
    |  Operator          >>=
    |   Number           2
    |   Limits           ;
21  |  KeyWords          int
    | Identifier         a_
    |  Operator          =
    |   Number           10uL
    |   Limits           ,
    | Identifier         b_
    |  Operator          =
    |   Number           100uL
    |   Limits           ;
23  |  KeyWords          int
    | Identifier         c_
    |  Operator          =
    |   Number           0x1e8
    |   Limits           ,
    | Identifier         d_
    |  Operator          =
    |   Number           070
    |   Limits           ,
    | Identifier         e_
    |  Operator          =
    |   Number           0b0011
    |   Limits           ;
24  |  KeyWords          float
    | Identifier         f_
    |  Operator          =
    |   Number           1.3f
    |   Limits           ,
    | Identifier         g_
    |  Operator          =
    |   Number           1.1l
    |   Limits           ;
26  | Identifier         printf
    |   Limits           (
    |   String           "print test \"\n"
    |   Limits           )
    |   Limits           ;
27  |  KeyWords          char
    | Identifier         str
    |   Limits           [
    |   Limits           ]
    |  Operator          =
    |   String           "string \\t        test
29  |   Limits           ;
31  |  KeyWords          int
    | Identifier         a
    |   Errors           $
```



```
Lexer > out.list
101      |   Number          23
102      |  Operator         =
103      |   Errors      !   #
104      |   Number          4
105      |   Limits          ;
106   32 |  KeyWords         float
107      | Identifier        b
108      |  Operator         =
109      |   Number          1.2
110      | Identifier        UL
111      |   Limits          ,
112      | Identifier        f
113      |  Operator         =
114      |   Number          1l
115      | Identifier        f
116      |   Limits          ;
117   33 |  KeyWords         int
118      | Identifier        c
119      |  Operator         =
120      |   Number          0x1
121      |   Number          .4
122      |   Limits          ,
123      | Identifier        d
124      |  Operator         =
125      |   Number          07
126      |   Number          8
127      |   Limits          ,
128      | Identifier        e
129      |  Operator         =
130      |   Number          0b1
131      |   Number          3
132      |   Limits          ;
133   34 |  KeyWords         char
134      | Identifier        str2
135      |   Limits          [
136      |   Limits          ]
137      |  Operator         =
138      |   Errors      !   "no end test \"
139   36 |  KeyWords         return
140      |   Number          0
141      |   Limits          ;
142   37 |   Limits          }
143
144   [statistic]
145   row: 37
146   char: 471
147
148   [types]
149     Errors  : 3
150    KeyWords : 16
```

```
Lexer > out.list
150    KeyWords : 16
151   Identifier: 32
152     Number  : 21
153    Operator : 23
154     Limits  : 42
155     String  : 2
156
```

3.3、分析说明

本词法分析程序将单词记号分为：

- 关键字　[KeyWords]　**(int, while, if …)**
- 标识符　[Identifier]　**(a0_, b, c …)**
- 数字　[Number]　**(10ll, .1e64, 1.2f …)**
- 操作符　[Operator]　**(+=, >>=, ->, …)**
- 界限符　[Limits]　**({, [, (, ;, …)**
- 字符串　[String]　**(“xxx”, ‘x’)**

对于**单词符号的分析**，可以看到，词法分析的输出文件中能够标识每行的各类单词符号，并在末尾记录总行数、总字符数，各记号个数。

对于**错误报告**，可以看到词法分析程序能够进行错误恢复，并准确标明**错误的所在位置和类型**。同时在输出末尾标明**错误的总个数**

四、LEX 实现

4.1、正则表达式

根据LEX程序的要求，结合实验中的词法分析程序，设计了基于正则表达式的LEXER。关键匹配项说明：

- **前序定义**

  提前将出**现频率高**的字符定义为宏，包括 0-9 数字、界限符号、操作符号、关键字以及后缀。

```
NUM         [0-9]
LETTER      [A-Za-z]
LIMIT       [\{\}\[\(\)\[\],;]
OP          (<|>|=|\+|-|\*|\/|\%|&|\|\!|\^|\.)|(>=|<=|==|!=|\+\+|--|&&|\|\||\+=|-=|\*=|\/=|%=|&=|\|=|\^=|>>|<<|->)|(>>=|<<=)
I_SUFFIX    (u|U|l|L|ll|LL|ul|uL|UL|lu|Lu|LU|ull|uLL|ULL|llu|LLu|LLU)
F_SUFFIX    (f|F|l|L)
KEY         (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|union|unsigned|void|volatile|while)
```

- **数字**

  在该项匹配中，分为十进制整数、十进制浮点、十六进制、八进制、二进制数分别构造正则表达式，在统计环节**统一统计为数字类别**：

```
hex         (0x|0X)[0-9A-Fa-f]+
octal       0[0-7]+
binary      (0b|0B)[0-1]+
```

```
integer     0|([1-9]+{NUM}*(([Ee][+-]?){NUM}+)?{I_SUFFIX}?)
float       {NUM}*\.{NUM}+(([Ee][+-]?){NUM}+)?{F_SUFFIX}?
```

- **字符串**

为了实现实验中的**字符串闭合缺失**的错误警告，在定义字符串的正则表达式的同时还另外定义了缺失闭合型字符串。

```
string      (\'\\?{LETTER}?\')|(\"([^\n\\\"]*(\\.|\\+\n)?)*\")
string_err  \"([^\n\\\"]*(\\.)?)*\n
```

如此，便可以在字符串闭合失败时进行错误警告，同时不会将字符串行的内容重新进行其他类型的匹配。

```
[string]        "string \\t\
        test\
    "
```

```
********** expected end: 33:"no end test \"
```

- 注释

注释区分了单行和多行注释，同时在正则表达式中考虑了**多个*号以及转义字符**的影响。

```
comment     (\/\/[^\n]*)|(\/\*([^\*]*(\*+[^\/]+)*)*\*\/)
```

4.2、辅助函数

为实现**字符统计、行数统计以及记号统计**功能，在 LEX 源程序中添加了各类计数器以及辅助函数。

```
%{/*2022-10-01*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int chars = 0,rows = 1,errors = 0;
int keywords = 0,ids = 0,nums = 0,operators = 0,limits = 0;
int strings = 0,comment = 0;
int find(const char* input,char c){ // 字符查找
    int count = 0;
    for(int i = 0; i < strlen(input); i++)
        if(input[i] == '\n')    count++;
    return count;
}
int length(const char* input){      // 字符长度计算(去除空格、换行)
```

```
    int count = 0;
    for(int i = 0;;i++){
        if(input[i] == 0)   break;
        if(input[i] != '\n' && input[i] != ' ' && input[i] != '\t')
            count ++;
    }
    return count;
}
#define ADDROW() {rows += find(yytext,'\n');};  // 行数统计
#define ADD(X) {X++;chars += length(yytext);};  // 字符统计
%}
```

4.3、错误警告

在所有预置匹配项均未成功时，将在控制台输出错误信息。

```
. {printf("***************
error: %d:%s\n",rows,yytext);ADD(errors);rows+=find(yytext,'\n
');}
```

4.4、运行结果

4.4.1、输入程序

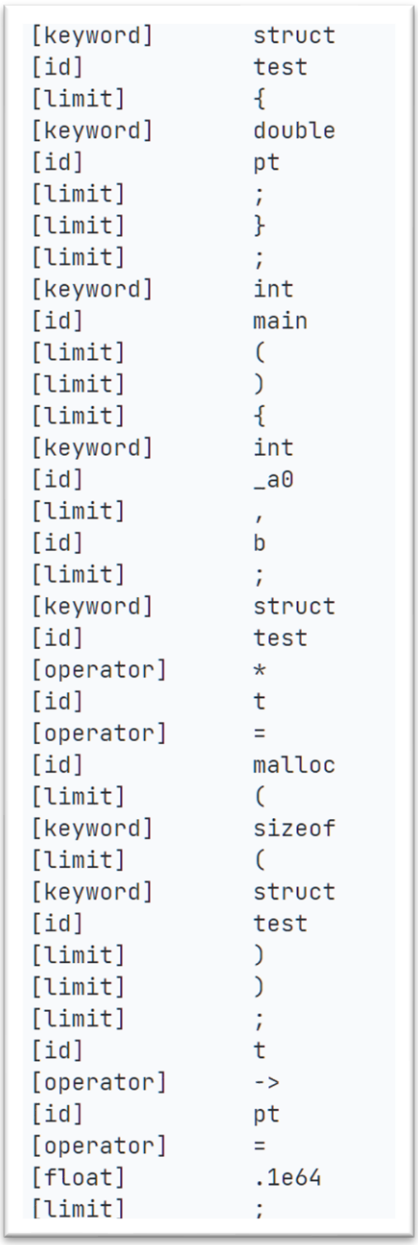
输入程序与测试 C++实现的词法分析程序相同。

4.4.2、运行结果

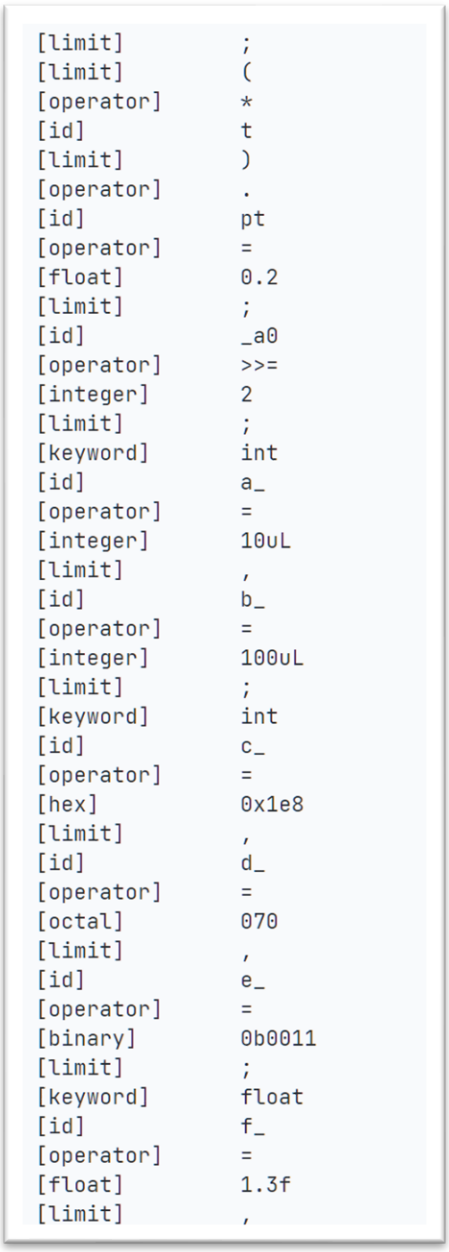
编译生成：

```
PS D:\Flex\GnuWin32\bin> .\flex G:\Coding\CProject\Compile\lex.l
PS D:\Flex\GnuWin32\bin> gcc lex.yy.c -o out
PS D:\Flex\GnuWin32\bin> .\out G:\Coding\CProject\Compile\Lexer\scripts\origin.c
```

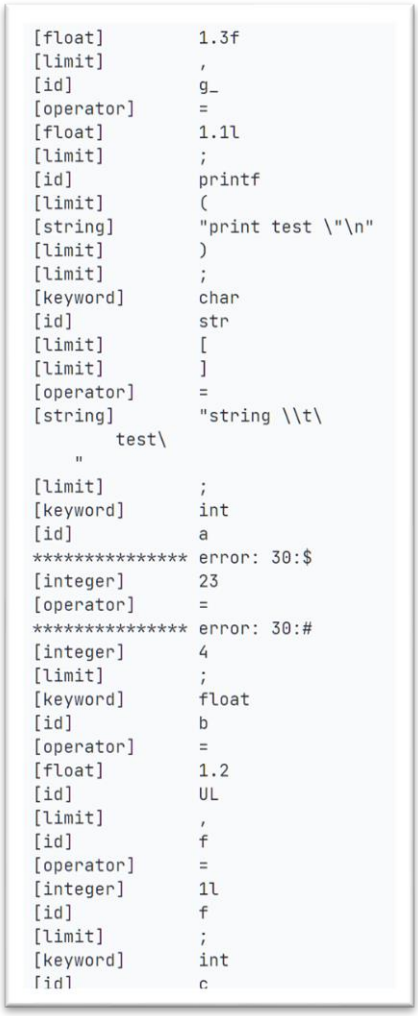
运行结果：



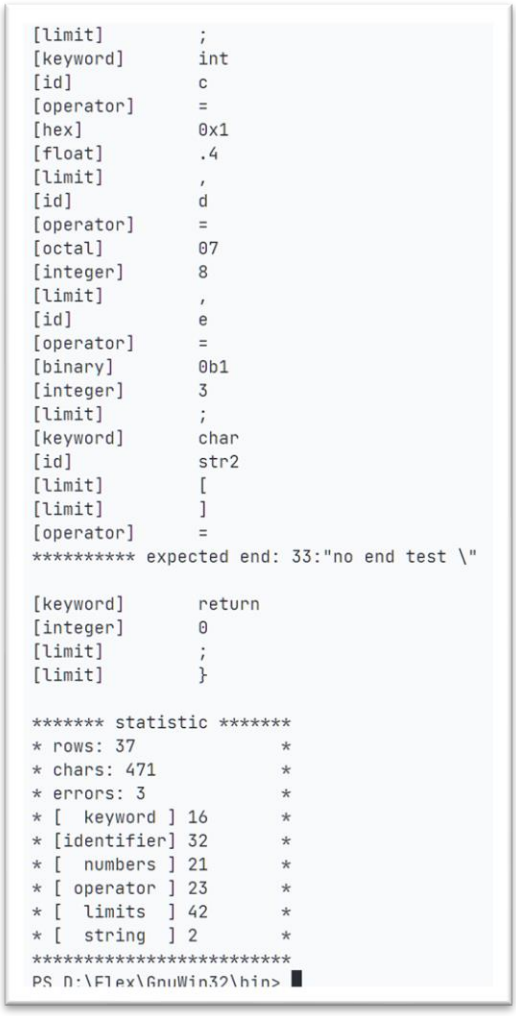
```
[keyword]       struct
[id]            test
[limit]         {
[keyword]       double
[id]            pt
[limit]         ;
[limit]         }
[limit]         ;
[keyword]       int
[id]            main
[limit]         (
[limit]         )
[limit]         {
[keyword]       int
[id]            _a0
[limit]         ,
[id]            b
[limit]         ;
[keyword]       struct
[id]            test
[operator]      *
[id]            t
[operator]      =
[id]            malloc
[limit]         (
[keyword]       sizeof
[limit]         (
[keyword]       struct
[id]            test
[limit]         )
[limit]         )
[limit]         ;
[id]            t
[operator]      ->
[id]            pt
[operator]      =
[float]         .1e64
[limit]         ;
```



```
[limit]         ;
[limit]         (
[operator]      *
[id]            t
[limit]         )
[operator]      .
[id]            pt
[operator]      =
[float]         0.2
[limit]         ;
[id]            _a0
[operator]      >>=
[integer]       2
[limit]         ;
[keyword]       int
[id]            a_
[operator]      =
[integer]       10uL
[limit]         ,
[id]            b_
[operator]      =
[integer]       100uL
[limit]         ;
[keyword]       int
[id]            c_
[operator]      =
[hex]           0x1e8
[limit]         ,
[id]            d_
[operator]      =
[octal]         070
[limit]         ,
[id]            e_
[operator]      =
[binary]        0b0011
[limit]         ;
[keyword]       float
[id]            f_
[operator]      =
[float]         1.3f
[limit]         ,
```



```
[float]         1.3f
[limit]         ,
[id]            g_
[operator]      =
[float]         1.1l
[limit]         ;
[id]            printf
[limit]         (
[string]        "print test \"\n"
[limit]         )
[limit]         ;
[keyword]       char
[id]            str
[limit]         [
[limit]         ]
[operator]      =
[string]        "string \\t\
        test\
    "
[limit]         ;
[keyword]       int
[id]            a
*************** error: 30:$
[integer]       23
[operator]      =
*************** error: 30:#
[integer]       4
[limit]         ;
[keyword]       float
[id]            b
[operator]      =
[float]         1.2
[id]            UL
[limit]         ,
[id]            f
[operator]      =
[integer]       1l
[id]            f
[limit]         ;
[keyword]       int
[id]            c
```

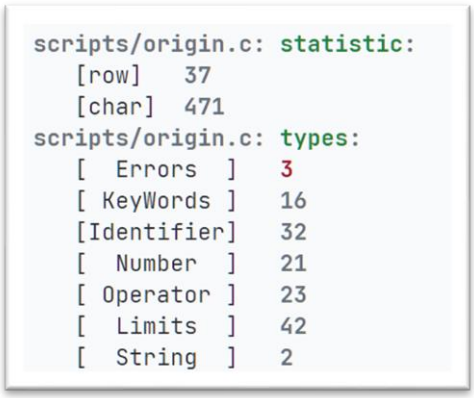


```
[limit]         ;
[keyword]       int
[id]            c
[operator]      =
[hex]           0x1
[float]         .4
[limit]         ,
[id]            d
[operator]      =
[octal]         07
[integer]       8
[limit]         ,
[id]            e
[operator]      =
[binary]        0b1
[integer]       3
[limit]         ;
[keyword]       char
[id]            str2
[limit]         [
[limit]         ]
[operator]      =
********** expected end: 33:"no end test \"

[keyword]       return
[integer]       0
[limit]         ;
[limit]         }

******* statistic *******
* rows: 37              *
* chars: 471            *
* errors: 3             *
* [  keyword ] 16       *
* [identifier] 32       *
* [  numbers ] 21       *
* [ operator ] 23       *
* [  limits  ] 42       *
* [  string  ] 2        *
************************
PS D:\Flex\GnuWin32\bin>
```

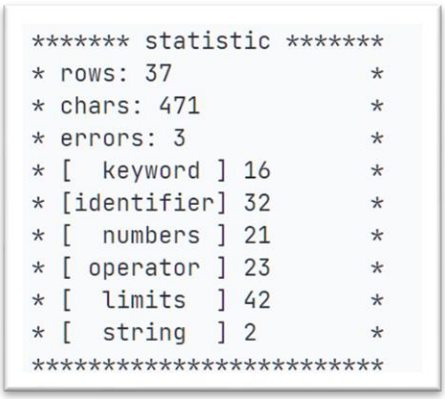
4.5、对比分析

在控制台输出结果中可以清楚的看到，LEX 程序能够准确识别各类记号，同时能够准确的统计各类记号的数量和错误。

统计情况对比：



```
scripts/origin.c: statistic:
   [row]   37
   [char]  471
scripts/origin.c: types:
   [  Errors  ]   3
   [ KeyWords ]   16
   [Identifier]   32
   [  Number  ]   21
   [ Operator ]   23
   [  Limits  ]   42
   [  String  ]   2
```



```
******* statistic *******
* rows: 37              *
* chars: 471            *
* errors: 3             *
* [  keyword ] 16       *
* [identifier] 32       *
* [  numbers ] 21       *
* [ operator ] 23       *
* [  limits  ] 42       *
* [  string  ] 2        *
************************
```

能够看到，C++实现的词法分析程序和 LEX 实现的词法分析器具有相同的识别结果，这也一定程度上验证了两类程序的正确性。