C语言词法分析程序的设计与实现

> **北京邮电大学 2019213688 池纪君**

实验题目、要求
实现方法要求：
C语言程序设计说明
    语言说明
    缓冲区设置
    状态自动机
        自动机说明
        自动机图示
    词法错误的检测和恢复
C语言程序运行说明
    运行结果
Lex程序说明
    Lex程序规则和写法
    Lex工作原理和在Linux下的使用
    Lex语言程序运行结果(分析目标文件同上)
源程序
    C语言程序 c_lex.cpp
    lex语言源程序 c_lex.l

实验题目、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。

C语言程序设计说明

语言说明

C语言中的记号（Tokens）：

1. 标识符：以字母、下划线开头的，以字母、下划线和数字组成的符号串。
2. 关键字：标识符集合的子集。本程序的C语言定义关键字如下：

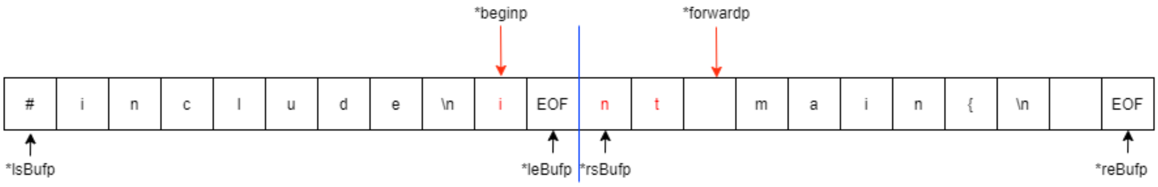
```
"auto",       "break",     "case",          "char",
"const",      "continue",  "default",       "do",
"double",     "else",      "enum",          "extern",
"float",      "for",       "goto",          "if",
"inline",     "int",       "long",          "register",
"restrict",   "return",    "short",         "signed",
"sizeof",     "static",    "struct",        "switch",
"typedef",    "union",     "unsigned",      "void",
"volatile",   "while",     "_Alignas",      "_Alignof",
"_Atomic",    "_Bool",     "_Complex",      "_Generic",
"_Imaginary", "_Noreturn", "_Static_assert", "_Thread_local"
```

3. 常量：C语言中的常量包括数字（number），字符（character）和字符串（character string）
    a. 数字：包括整数和浮点数。
    b. 字符：本程序只考虑整型字符，即类型为int的字符常量，包含在' '中。
    c. 字符串：本程序只考虑常规字符串里，即包含在" "中的字符集合，不考虑宽字符构成的字符串（Wide-character string）
4. 符号：其他C语言程序中出现的具有功能的合法符号。本程序中考虑的所有符号如下：

```
"*",  "%",  "!",  "^",  "=",  "+",  "-",  "<",   ">",   "|",   "&",
"/",  "\'", "\"", "#",  "(",  ")",  "[",  "]",   "{",   "}",   ";",
",",  "~",  "++", "--", "&&", "||", "<<", ">>",  "==",  "<=",  "!=",
">=", "+=", "-=", "*=", "/=", "%=", "^=", "&&=", "||=", "<<=", ">>="
```

缓冲区设置

- 本程序中，将一个缓冲区分成了两个大小为1KB的半缓冲区，设置缓冲区头指针 `lsBufp`， `rsBufp` 和缓冲区尾指针 `leBufp`， `reBufp`。程序将依次在左，右两个半缓冲区中缓存数据并一次读入状态机中。
- 记录当前串的指针为 `beginp` 和 `forwardp`，每次读取下一个字符时， `forwardp` 指针会将所在位置的字符加入当前串中，并前移一位。
- 状态机结束前，会将$[beginp, forwardp)$加入到对应的符号表中。如图所示，倘若在当前状态结束该标识符的读取，则将插入串 `int`，而又因为 `int` 属于关键字，则程序输出 `int keyword int`。

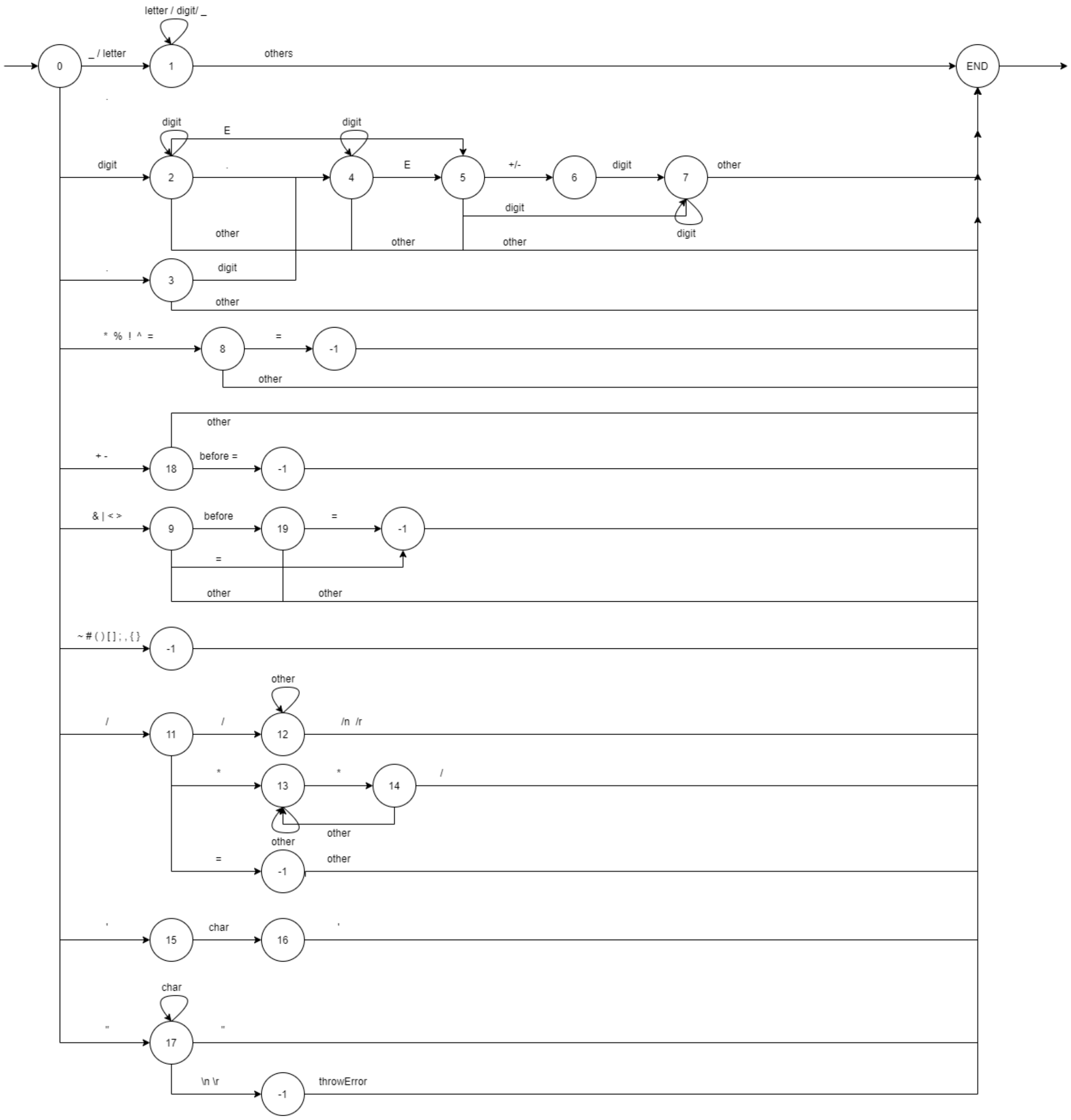


状态自动机

自动机说明

- 状态机常规状态主要分为五大类状态，分别表示标识符、数字常量、符号、常量字符和常量字符串。
- 图示中的-1态为直接返回0状态的临时态，加入改状态的目的在于区别到达此状态的字符和其他字符。当状态机到达该态时，需要结束之前的状态，将已识别出的符号、标识符或常量存入对应的符号表中，并使向前指针退后，重新读取刚才的字符以进入新的一轮状态转移中。
- **注释内容识别和跳过**：当从初始状态出发，自动机识别到//或/*的字符时，将进入注释状态并对注释中的内容不加处理地跳过。当在接受/*的状态下接收到结束注释的*/符号时，自动机将终止注释状态，并返回初始状态。

自动机图示



词法错误的检测和恢复

- 当检测到词法错误时，程序将输出`An error occurred in Line {lineCnt}: {charInLine}`并输出报错信息。同时，自动机将**恢复到初始状态**（状态0）继续读取源程序。
- 该程序主要处理的错误有以下几类：
    - 标识符过长：标识符过长将导致缓冲区错误，可能导致程序运行出错，此时程序将输出`Identifier too long. (Max size: 1024)`，并忽略已存储的标识符，继续读取源程序。
    - 标识符不合法：常见的标识符错误为以数字开头的标识符。当检测到数字出现在字母前的情况时，将输出错误信息：`Illegal number.`并忽略已经读取的数字。（注：为完整分析源程序，词法分析将继续分析数字后的字母并将其视为标识符，例：片段`5a5`会产生报错信息，但仍会被解析为标识符`a5`）
    - 不合法的字符：当在非注释状态读取到非C语言中的字符时，将输出错误信息：`Unkown sign`
    - 数值常量表示不合法：形如 .、$3e$、$5e+$ 等数值常量被读取时，将输出错误信息：`Illegal number.`
    - 字符串常量不合法：在字符串状态下，若在行末（'\n', '\r'）前没有解析到右引号"，则输出错误信息：`Lack of "`
    - 字符常量过长：在字符状态下，若输入字符过长，则输出错误信息`Too many characters entered.`
- 在分析完源程序后，将输出词法错误出现的次数。

C语言程序运行说明

程序接受命令行参数，由用户进行输入和输出地址控制。

- 在运行程序的命令中加入`-input`即可指定输入的文件路径（默认为`test0.c`），`-output`指定输出文件路径（默认为`result.txt`）。
    - 例如：用户将C语言代码存在`./test0.c`中，希望将分析结果存储在`./result.txt`中，则他需要输入命令语句为：`./c_lex.exe -input test0.c -output result.txt`
- 此法程序输出为如下形式：
    - 输出行号；
    - 对所有记号，程序将输出形如<x. token, property>的三元组；
        - 对标识符，状态机将输出<标识符名称, id, 该标识符在标识符表中的索引>。若该标识符属于关键字，则输出<关键字名称, keyword, 关键字含义>；
        - 对常量数字，状态机将输出<数值, 数据类型, 该数值在数值表中的索引>；数据类型包括`int`，`double`，`long long`
        - 对常量字符和常量字符串，状态机将输出<字符(串)值, 字符(串)类型, 该字符(串)在字符串表中的索引>；
        - 对符号，状态机将输出<符号, op, 符号含义>；
    - **输出源程序词法分析统计结果**：从上到下依次为：源程序的语句行数、记号数、关键字数、标识符数、常量数字数、符号数、常量字符串数、常量字符数、词法错误个数。

运行结果

源程序代码(test.c)：

```
#include<stdio.h>

#include "5-6.h"
#include "5-12.h"
#include "5-1.h"

void Swap ( int x, int y )
{
  int temp;

  temp=x;
  x=y;
  y=temp;
}
int main() {
    double a = .3;//
    double b = 4e;/*  */
    double c = 6.;/*  */
    5a5=3;
    double d = .3e3;
    printf("%d %d", a, b);
    char s[20]="asdfasas\'dfasdf";
    char s1 = 'a';
    char s2 = "asdf\nasdf   zxcvaerh"
    ;
    int a = 2, b = 3, 0c=4;
    a++ <<= b;
    a >>=b--;
    /*  */printf("%s", s);z
    printf("璇ユ枃妗ｅ叡鍑虹幇 %d 涓□湁鏁堝崟璇?", wordcount);
    Show( H, 10.0/100 );  /* 鏄剧ず璇嶉□鍓?0%鐨勫壁鏈夊崟璇?*/
    DestroyTable( H );    /* 閿€姣佸暅鍒楄〃 */

    return 0;
}
```

结果程序代码(result.c)：

可以看到，程序对错误`5a5`进行了处理并跳过`5`，继续处理后续字符。

```
Line 0:
# op #
include id 0
< op <
stdio id 1
```

```
. op .
h id 2
> op >

Line 1:

Line 2:
# op #
include id 0
"5-6.h" const_char[7] 0

Line 3:
# op #
include id 0
"5-12.h" const_char[8] 1

Line 4:
# op #
include id 0
"5-1.h" const_char[7] 2

Line 5:

Line 6:
void keyword void
Swap id 3
( op (
int keyword int
x id 4
, op ,
int keyword int
y id 5
) op )

Line 7:
{ op {

Line 8:
int keyword int
temp id 6
; op ;

Line 9:

Line 10:
temp id 6
= op =
x id 4
; op ;

Line 11:
x id 4
= op =
y id 5
; op ;

Line 12:
y id 5
= op =
temp id 6
; op ;

Line 13:
} op }

Line 14:
int keyword int
main id 7
( op (
) op )
{ op {

Line 15:
double keyword double
a id 8
= op =
0.300000 double 0
; op ;

Line 16:
double keyword double
b id 9
= op =
4 int 0
; op ;

Line 17:
```

```
double keyword double
c id 10
= op =
6 int 1
; op ;

Line 18:
An error occurred in Line 18:6:  Illegal number.
a5 id 11
= op =
3 int 2
; op ;

Line 19:
double keyword double
d id 12
= op =
300.000000 double 1
; op ;

Line 20:
printf id 13
( op (
"%d %d" const_char[7] 3
, op ,
a id 8
, op ,
b id 9
) op )
; op ;

Line 21:
char keyword char
s id 14
[ op [
20 int 3
] op ]
= op =
"asdfasas\'dfasdf" const_char[18] 4
; op ;

Line 22:
char keyword char
s1 id 15
= op =
' Char 0
; op ;

Line 23:
char keyword char
s2 id 16
= op =
"asdf\nasdf   zxcvaerh" const_char[23] 5

Line 24:
; op ;

Line 25:
int keyword int
a id 8
= op =
2 int 4
, op ,
b id 9
= op =
3 int 2
, op ,
An error occurred in Line 25:33:  Illegal number.
c id 10
= op =
4 int 0
; op ;

Line 26:
a id 8
++ op ++
<<= op <<=
b id 9
; op ;

Line 27:
a id 8
>>= op >>=
b id 9
-- op --
; op ;
```

```
Line 28:
printf id 13
( op (
"%s" const_char[4] 6
, op ,
s id 14
) op )
; op ;
z id 17

Line 29:
printf id 13
( op (
"璇ユ枃妗ｅ叡鍑虹幇 %d 涓□湁鏁堝崟璇?" const_char[55] 7
, op ,
wordcount id 18
) op )
; op ;

Line 30:
Show id 19
( op (
H id 20
, op ,
10.000000 double 2
/ op /
100 int 5
) op )
; op ;

Line 31:
DestroyTable id 21
( op (
H id 20
) op )
; op ;

Line 32:

Line 33:
return keyword return
0 int 6
; op ;

Line 34:
} op }

Total:
  34 line
  154 tokens
  14 keywords
  44 identifiers
  12 constants
  75 operator
  8 string
  1 char
Total error: 2
```

Lex程序说明

Lex程序规则和写法

Lex源程序必须按照Lex语言的规范来写，其核心是一组词法规则（正规式）。一般而言，一个Lex源程序分为三部分，三部分之间以符号%%分隔。

```
[第一部分：定义段]
%%
[第二部分：词法规则段]
%%
[第三部分：辅助函数段]
```

- 定义段可以分为两部分：
  - 第一部分以符号%{和%}包裹，里面为以C语法写的一些定义和声明：例如，文件包含，宏定义，常数定义，全局变量及外部变量定义，函数声明等。这一部分被Lex翻译器处理后会全部拷贝到文件lex.yy.c中。特殊括号%{和%}都必须位于行首。

    - 第二部分是一组正规定义和状态定义。正规定义是为了简化后面的词法规则而给部分正规式定义了名字。每条正规定义也都要顶着行首写。例如下面这组正规定义分别定义了letter，digit和id所表示的正规式：
- 词法规则段列出的是词法分析器需要匹配的正规式，以及匹配该正规式后需要进行的相关动作。
- 辅助函数段用C语言语法来写，辅助函数一般是在词法规则段中用到的函数。这一部分一般会被直接拷贝到lex.yy.c中。

Lex工作原理和在Linux下的使用

- Lex基本工作原理为：由正规式生成NFA，将NFA变换成DFA，DFA经化简后，模拟生成词法分析器。
- 在Linux下的使用：
    - `sudo install flex` 安装lex翻译器
    - `flex c_lex.l` 将lex源文件翻译成一个名为lex.yy.c的C语言源文件，此文件含有两部分内容：一部分是根据正规式所构造的DFA状态转移表，另一部分是用来驱动该表的总控程序yylex()
    - `gcc lex.yy.c -o example -lyl` 将lex.yy.c程序用C编译器进行编译，并将相关支持库函数连入目标代码，其中`-lfl`是链接flex的库函数的，库函数中可能包含类似`yywrap()`一类的标准函数。
    - `./example < test.c > result.txt` 运行词法分析程序。当主程序需要从输入字符流中识别一个记号时，只需要调用一次`yylex()`。

Lex语言程序运行结果(分析目标文件同上)

Lex语言程序中没有对错误进行特殊处理，故只按照给定正规表达式分析。

```
(OP #)
(ID include)
(RELOP <)
(ID stdio)
(OP .)
(ID h)
(RELOP >)
(OP #)
(ID include)
(STRING_LITERAL "5-6.h")
(OP #)
(ID include)
(STRING_LITERAL "5-12.h")
(OP #)
(ID include)
(STRING_LITERAL "5-1.h")
(KEYWORD, void)
(ID Swap)
(OP ()
(KEYWORD, int)
(ID x)
(OP ,)
(KEYWORD, int)
(ID y)
(OP ))
(OP {)
(KEYWORD, int)
(ID temp)
(OP ;)
(ID temp)
(OP =)
(ID x)
(OP ;)
(ID x)
(OP =)
(ID y)
(OP ;)
(ID y)
(OP =)
(ID temp)
(OP ;)
(OP })
(KEYWORD, int)
(ID main)
(OP ()
(OP ))
(OP {)
(KEYWORD, double)
(ID a)
(OP =)
```

```
(NUM .3)
(OP ;)
(KEYWORD, double)
(ID b)
(OP =)
(NUM 4e)
(OP ;)
(KEYWORD, double)
(ID c)
(OP =)
(NUM 6.)
(OP ;)
(NUM 5)
(ID a5)
(OP =)
(NUM 3)
(OP ;)
(KEYWORD, double)
(ID d)
(OP =)
(NUM .3e3)
(OP ;)
(ID printf)
(OP ()
(STRING_LITERAL "%d %d")
(OP ,)
(ID a)
(OP ,)
(ID b)
(OP ))
(OP ;)
(KEYWORD, char)
(ID s)
(OP [)
(NUM 20)
(OP ])
(OP =)
(STRING_LITERAL "asdfasas\'dfasdf")
(OP ;)
(KEYWORD, char)
(ID s1)
(OP =)
(C_CHAR 'a')
(OP ;)
(KEYWORD, char)
(ID s2)
(OP =)
(STRING_LITERAL "asdf\nasdf   zxcvaerh")
(OP ;)
(KEYWORD, int)
(ID a)
(OP =)
(NUM 2)
(OP ,)
(ID b)
(OP =)
(NUM 3)
(OP ,)
(NUM 0)
(ID c)
(OP =)
(NUM 4)
(OP ;)
(ID a)
(OP ++)
(OP <<=)
(ID b)
(OP ;)
(ID a)
(OP >>=)
(ID b)
(OP --)
(OP ;)
(ID printf)
(OP ()
(STRING_LITERAL "%s")
(OP ,)
(ID s)
(OP ))
(OP ;)
(ID z)
(ID printf)
(OP ()
(STRING_LITERAL "璇ユ枃浠ｅ叡鏈夊 %d 涓□湁鏁堝崟璇?")
(OP ,)
(ID wordcount)
(OP ))
```

```
(OP ;)
(ID Show)
(OP ()
(ID H)
(OP ,)
(NUM 10.0)
(OP /)
(NUM 100)
(OP ))
(OP ;)
(ID DestroyTable)
(OP ()
(ID H)
(OP ))
(OP ;)
(KEYWORD, return)
(NUM 0)
(OP ;)
(OP })

Total:
14 keywords
44 identifiers
14 constants
75 operator
8 string
1 char
```

源程序

C语言程序 c_lex.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int HALFBUFSIZE = 1024;
const int OPTABLESIZE = 45;
const int UNARYOPSIZE = 25;
const int BINARYOPSIZE = 20;
const int KEYWORDSIZE = 44;
const int MAXIDLENGTH = 1024;

FILE *infp, *outfp;
char *inputPath, *outputPath, *lsBufp, *leBufp, *rsBufp, *reBufp, *beginp,
    *forwardp;
char buffer[HALFBUFSIZE * 2 + 2], nowC;
int charcCnt, charInLine, lineCnt, keywordCnt, idCnt, numCnt, opCnt, charCnt,
    strCnt, wordCnt, errorCnt;
int state, crossFlag;
int numInt, numK, numE, numF;
double numDoub;
// symbol table
vector<string> identifier, constStr;
vector<double> constDouble;
vector<int> constInt;
vector<long long> constLong;
vector<char> constChar;

int unaryOpNxtState[UNARYOPSIZE] = {8,  8,  8,  8,  8,  18, 18, 3,  9,
                                    9,  9,  9,  11, 15, 17, 10, 10, 10,
                                    10, 10, 10, 10, 10, 10, 10};
char opTable[OPTABLESIZE][5] = {
    "*",  "%",  "!",  "^",  "=",  "+",   "-",   ".",   "<",  ">",  "|",  "&",
    "/",  "\'", "\"", "#",  "(",  ")",   "[",   "]",   "{",  "}",  ";",  ",",
    "~",  "++", "--", "&&", "||", "<<",  ">>",  "==",  "<=", "!=", ">=", "+=",
    "-=", "*=", "/=", "%=", "^=", "&&=", "||=", "<<=", ">>="};

// Standard C keywords
char keywords[KEYWORDSIZE][15] = {
    "auto",       "break",     "case",           "char",
    "const",      "continue",  "default",        "do",
    "double",     "else",      "enum",           "extern",
    "float",      "for",       "goto",           "if",
    "inline",     "int",       "long",           "register",
    "restrict",   "return",    "short",          "signed",
    "sizeof",     "static",    "struct",         "switch",
    "typedef",    "union",     "unsigned",       "void",
    "volatile",   "while",     "_Alignas",       "_Alignof",
    "_Atomic",    "_Bool",     "_Complex",       "_Generic",
    "_Imaginary", "_Noreturn", "_Static_assert", "_Thread_local"};

int argPos(char *str, int argc, char **argv) {
  for (int i = 0; i < argc; ++i) {
    if (!strcmp(str, argv[i])) {
      if (i == argc - 1) {
        printf("Argument missing for %s\n", str);
        exit(1);
      }
      return i;
    }
  }
  return -1;
}

void getArg(int argc, char **argv) {
  int a;
  if ((a = argPos(const_cast<char *>("-input"), argc, argv)) > 0) {
    inputPath = argv[a + 1];
  } else {
    inputPath = const_cast<char *>("test.c");
  }
  if ((a = argPos(const_cast<char *>("-output"), argc, argv)) > 0) {
    outputPath = argv[a + 1];
  } else {
    outputPath = const_cast<char *>("result.txt");
  }
}

// half buffer end with EOF
// valid space: HALFBUFSIZE - 1
void getHalfBuffer(char *buffp) {
  int cnt = fread(buffp, 1, HALFBUFSIZE - 1, infp);
  buffp[cnt] = EOF;
}

void init() {
  infp = fopen(inputPath, "r");
  outfp = fopen(outputPath, "w");
  lsBufp = buffer, leBufp = buffer + HALFBUFSIZE - 1,
  rsBufp = buffer + HALFBUFSIZE, reBufp = buffer + HALFBUFSIZE * 2 - 1;
  forwardp = lsBufp;
  *leBufp = *reBufp = EOF;
  printf("Input file: %s\nOutput file: %s\n", inputPath, outputPath);
  getHalfBuffer(lsBufp);
  fprintf(outfp, "Line 0:\n");
}

void getNextChar() {
  nowC = *forwardp++;
  charcCnt++, charInLine++;
  if (*forwardp == EOF) {
    if (forwardp == leBufp) {
      if (!crossFlag) getHalfBuffer(rsBufp);
      forwardp = rsBufp;
      crossFlag = 0;
    } else if (forwardp == reBufp) {
      if (!crossFlag) getHalfBuffer(lsBufp);
      forwardp = lsBufp;
      crossFlag = 0;
    }
  }
}

void retreatPtr() {
  if (forwardp == lsBufp) {
    forwardp = reBufp - 1;
    crossFlag = 1;
  } else if (forwardp == rsBufp) {
    forwardp = leBufp - 1;
    crossFlag = 1;
  } else {
    forwardp--;
  }
}

// find pointer back x position from *forwardp
char *backPtr(int x) {
  char *p = forwardp;
  for (int i = 1; i <= x; i++) {
    if (p == lsBufp)
      p = reBufp - 1;
    else if (forwardp == rsBufp)
      p = leBufp - 1;
    else
      p = p - 1;
  }
  return p;
}
```

```
bool isLetter() {
  return (nowC >= 'a' && nowC <= 'z') || (nowC >= 'A' && nowC <= 'Z') ||
         (nowC == '_');
}

bool isDigit() { return nowC >= '0' && nowC <= '9'; }

bool isWhiteSpace() {
  if (nowC == ' ' || nowC == '\t' || nowC == '\n' || nowC == '\r') return true;
  return false;
}

// return -1 if nowC isn't a operator, otherwise return index of nowC
int isUnaryOperator() {
  for (int i = 0; i < UNARYOPSIZE; i++)
    if (nowC == opTable[i][0]) return unaryOpNxtState[i];
  return -1;
}

// throw error retreat forwardptr and turn to state 0
void throwError(string str) {
  errorCnt++;
  retreatPtr();
  fprintf(outfp, "An error occurred in Line %d:%d: ", lineCnt, charInLine);
  fprintf(outfp, " %s\n", str.c_str());
  state = 0;
}

// return the string between beginp -> forwardp-1
char *getString() {
  char *str = static_cast<char *>(malloc(sizeof(char) * MAXIDLENGTH));
  int len;
  if ((beginp < forwardp && forwardp < leBufp) ||
      (beginp < forwardp && beginp > leBufp)) {
    strncpy(str, beginp, forwardp - beginp);
    len = forwardp - beginp;
  } else if (beginp < leBufp && forwardp >= rsBufp) {
    char tmp[MAXIDLENGTH];
    strncpy(str, beginp, leBufp - beginp);
    len = leBufp - beginp;
    if (forwardp != rsBufp) {
      strncpy(tmp, rsBufp, forwardp - rsBufp);
      strcat(str, tmp);
      len += forwardp - rsBufp;
    }
  } else if (beginp >= rsBufp && forwardp < leBufp) {
    char tmp[MAXIDLENGTH];
    strncpy(str, beginp, reBufp - beginp);
    len = reBufp - beginp;
    if (forwardp != lsBufp) {
      strncpy(tmp, lsBufp, forwardp - lsBufp);
      strcat(str, tmp);
      len += forwardp - lsBufp;
    }
  } else
    throwError("Identifier too long. (Max size: 1024)");
  str[len] = '\0';
  return str;
}

// insert identifier into table
// *beginp  -> *forwardp
void insertId() {
  char *idf = getString();
  int flag = 0;
  for (int i = 0; i < KEYWORDSIZE; i++)
    if (strcmp(idf, keywords[i]) == 0) {  // find keywords
      flag = 1;
      fprintf(outfp, "%s keyword %s\n", idf, idf);
      keywordCnt++;
      break;
    }
  std::string tmp = idf;
  for (int i = 0; i < identifier.size(); i++)
    if (identifier[i] == tmp) {
      flag = 1;
      idCnt++;
      fprintf(outfp, "%s id %d\n", idf, i);
      break;
    }
  if (!flag) {
    idCnt++;
    identifier.push_back(tmp);
    fprintf(outfp, "%s id %d\n", idf, identifier.size() - 1);
  }
  free(idf);
  state = 0;
```

```
}

double qkpow(double x, int k) {
  double ans = 1;
  while (k) {
    if (k & 1) ans *= x;
    k >>= 1;
    x *= x;
  }
  return ans;
}

// insert identifier into table and turn to state 0
void insertConstant() {
  double tmp = (numInt + numDoub) * qkpow(10, numE * numF);
  int flag = 0;
  numCnt++;
  state = 0;
  if (numK) {
    for (int i = 0; i < constDouble.size(); i++)
      if (constDouble[i] == tmp) {
        flag = 1;
        fprintf(outfp, "%lf double %d\n", tmp, i);
        break;
      }
    if (!flag) {
      constDouble.push_back(tmp);
      fprintf(outfp, "%lf double %d\n", tmp, constDouble.size() - 1);
    }
  } else if (tmp > INT_MAX) {  // long long
    for (int i = 0; i < constLong.size(); i++)
      if (constLong[i] == tmp) {
        flag = 1;
        fprintf(outfp, "%lld long long %d\n", (long long)tmp, i);
        break;
      }
    if (!flag) {
      constLong.push_back(tmp);
      fprintf(outfp, "%lld long long %d\n", (long long)tmp,
              constLong.size() - 1);
    }
  } else {  // int
    for (int i = 0; i < constInt.size(); i++)
      if (constInt[i] == tmp) {
        flag = 1;
        fprintf(outfp, "%d int %d\n", (int)tmp, i);
        break;
      }
    if (!flag) {
      constInt.push_back(tmp);
      fprintf(outfp, "%d int %d\n", (int)tmp, constInt.size() - 1);
    }
  }
}

// insert operator(delimiter) into table and turn to state 0
void insertOp() {
  char *op = getString();
  for (int i = 0; i < OPTABLESIZE; i++)
    if (strcmp(op, opTable[i]) == 0) {
      fprintf(outfp, "%s op %s\n", op, op);
      break;
    }
  free(op);
  opCnt++;
  state = 0;
}

// insert string into table and turn to state 0
void insertStr() {
  char *str = getString();
  std::string tmpstr = str;
  int flag = 0;
  for (int i = 0; i < constStr.size(); i++)
    if (constStr[i] == tmpstr) {
      fprintf(outfp, "%s const_char[%d] %d\n", str, (int)strlen(str), i);
      flag = 1;
      break;
    }
  if (!flag) {
    constStr.push_back(tmpstr);
    fprintf(outfp, "%s const_char[%d] %d\n", str, (int)strlen(str),
            constStr.size() - 1);
  }
  free(str);
  strCnt++;
  state = 0;
```

```
}

// insert char into char table and turn to state 0
void insertChar() {
  char ch = *(backPtr(1));
  int flag = 0;
  for (int i = 0; i < constChar.size(); i++)
    if (constChar[i] == ch) {
      fprintf(outfp, "%c Char %d\n", ch, i);
      flag = 1;
      break;
    }
  if (!flag) {
    constChar.push_back(ch);
    fprintf(outfp, "%c Char %d\n", ch, constChar.size() - 1);
  }
  charCnt++;
  state = 0;
}

void lexAnalysis() {
  int flag;
  do {
    getNextChar();
    // if (nowC == EOF) {  // touch EOF!
    //   printf("%d %d %d %d %d %d\n", beginp, forwardp, state, buffer, leBufp,
    //          reBufp);
    // }
    switch (state) {
      case 0:
        if (isWhiteSpace()) {
          if (nowC == '\n' || nowC == '\r') {
            fprintf(outfp, "\nLine %d:\n", ++lineCnt);
            charInLine = 0;
          }
          continue;
        }
        beginp = backPtr(1);
        if (isLetter()) {
          state = 1;
        } else if (isDigit()) {
          numInt = nowC - '0', numDoub = numK = numE = 0, numF = 1;
          state = 2;
        } else if (nowC == '.') {
          numInt = 0, numDoub = numK = numE = 0, numF = 1;
          state = 3;
        } else if ((flag = isUnaryOperator()) != -1) {
          state = flag;
        } else if (nowC != EOF) {
          throwError("Unkown sign.");
          getNextChar();  // need go forward
        }

        break;

      case 1:
        if (isWhiteSpace() || (!isLetter() && !isDigit())) {
          retreatPtr();
          insertId();
        }
        break;

      case 2:
        if (isDigit()) {
          numInt = numInt * 10 + nowC - '0';
        } else if (nowC == '.') {
          state = 4;
        } else if (nowC == 'E' || nowC == 'e') {
          state = 5;
        } else if (isLetter()) {
          throwError("Illegal number.");
        } else {
          retreatPtr();
          insertConstant();
        }
        break;

      case 3:
        if (isDigit()) {
          numDoub += qkpow(0.1, ++numK) * (nowC - '0');
          state = 4;
        } else {
          retreatPtr();
          insertOp();
        }
        break;
      case 4:
        if (isDigit()) {
          numDoub += qkpow(0.1, ++numK) * (nowC - '0');
        } else if (nowC == 'E' || nowC == 'e') {
          state = 5;
        } else {
          retreatPtr();
          insertConstant();
        }
        break;

      case 5:
        if (isDigit()) {
          numE = numE * 10 + nowC - '0';
          state = 7;
        } else if (nowC == '+' || nowC == '-') {
          if (nowC == '-') numF = -1;
          state = 6;
        } else {
          retreatPtr();
          insertConstant();
        }
        break;
      case 6:
        if (isDigit()) {
          numE = numE * 10 + nowC - '0';
          state = 7;
        } else
          throwError("Illegal number.");
        break;
      case 7:
        if (isDigit()) {
          numE = numE * 10 + nowC - '0';
        } else {
          retreatPtr();
          insertConstant();
        }
        break;

      case 8:
        if (nowC != '=') retreatPtr();
        insertOp();
        break;
      case 9:
        if (nowC == (*backPtr(2))) {
          state = 19;
        } else if (nowC == '=') {
          insertOp();
        } else {
          retreatPtr();
          insertOp();
        }
        break;
      case 10:
        retreatPtr();
        insertOp();
        break;
      case 11:
        switch (nowC) {
          case '/':
            state = 12;
            break;
          case '*':
            state = 13;
            break;
          case '=':
            insertOp();
            break;
          default:
            retreatPtr();
            insertOp();
            break;
        }
        break;
      case 12:
        if (nowC == '\n' || nowC == '\r') {
          fprintf(outfp, "\nLine %d:\n", ++lineCnt);
          charInLine = 0;
          state = 0;
        }
        break;
      case 13:
        if (nowC == '*') state = 14;
        break;
      case 14:
        if (nowC == '/')
          state = 0;
        else
          state = 13;
        break;
      case 15:
        state = 16;
        break;
      case 16:
        if (nowC == '\'')
          insertChar();
        else
          throwError("Too many characters entered.");
        break;
      case 17:
        if (nowC == '\"')
          insertStr();
        else if (nowC == '\n' || nowC == '\r')
          throwError("Lack of \"");
        break;
      case 18:
        if (nowC != (*backPtr(2)) && nowC != '=') retreatPtr();
        insertOp();
        break;
      case 19:
        if (nowC != '=') retreatPtr();
        insertOp();
        break;
      default:
        break;
    }
  } while (nowC != EOF);
  wordCnt = keywordCnt + idCnt + numCnt + opCnt + strCnt + charCnt;
  fprintf(outfp,
          "\nTotal:\n\t%d line\n\t%d tokens\n\t%d keywords\n\t%d "
          "identifiers\n\t%d "
          "constants\n\t%d operator\n\t%d string\n\t%d char\nTotal error: %d",
          lineCnt, wordCnt, keywordCnt, idCnt, numCnt, opCnt, strCnt, charCnt,
          errorCnt);
}

int main(int argc, char **argv) {
  getArg(argc, argv);
  init();
  lexAnalysis();
  fclose(infp);
  fclose(outfp);
  printf("Lexical analysis succeed. Result saved to %s", outputPath);
  return 0;
}
```

lex语言源程序 c_lex.l

```
%{

#include <stdio.h>

#define KEYWORD       1
#define ID              2
#define NUMBER          3
#define OP_ASSIGN     4
#define RELOP           5
#define OP_OTHER        6
#define STRING_LITERAL  7
#define C_CHAR          8
#define ERRORCHAR       9

int yylval;
int keyCnt = 0, idCnt = 0, numCnt = 0, strCnt = 0, charCnt = 0, opCnt = 0, errorCnt = 0;

%}

delim   [ \t \n]
ws      {delim}+
letter      [A-Za-z_]
digit   [0-9]
id      {letter}({letter}|{digit})*
D     [0-9]
L     [a-zA-Z_]
H     [a-fA-F0-9]
E     ([Ee][+-]?{D}*)
P           ([Pp][+-]?{D}+)
FS      (f|F|l|L)
IS          ((u|U)|(u|U)?(l|L|ll|LL)|(l|L|ll|LL)(u|U))
key         auto|break|case|char|const|continue|default|do|double|else|enum|extern|float|for|goto|if|inline|int|long|register|restrict
```

```
%%

"/*"              { comt(); }
"//"[^\n]*              { /* consume //-comment */ }

{ws}                  {;/* Go ahead */}
{key}                 {yylval = KEYWORD; return (KEYWORD);}
{id}                  {yylval = installID (); return (ID);}

  /*
  L?'(\\.|[^\\'\n])+'         { yylval = NUMBER; return(NUMBER); }
  This project consider char as character(s) rather than an unsigned integer and abandoned this rule.
  */
{D}+{E}{FS}?            { yylval = NUMBER; return(NUMBER); }
0[xX]{H}+{IS}?            { yylval = NUMBER; return(NUMBER); }
{D}*"."{D}+{E}?{FS}?      { yylval = NUMBER; return(NUMBER); }
{D}+"."{D}*{E}?{FS}?      { yylval = NUMBER; return(NUMBER); }
0[xX]{H}+{P}{FS}?         { yylval = NUMBER; return(NUMBER); }
0[xX]{H}*"."{H}+{P}{FS}?    { yylval = NUMBER; return(NUMBER); }
0[xX]{H}+"."{H}*{P}{FS}?    { yylval = NUMBER; return(NUMBER); }
0[0-7]*{IS}?            { yylval = NUMBER; return(NUMBER); }
[1-9]{D}*{IS}?            { yylval = NUMBER; return(NUMBER); }

L?\"(\\.|[^\\"\n])*\"     { yylval = STRING_LITERAL;return(STRING_LITERAL); }

  /*
  C_CHAR is added to ANSI C
  */
L?\'(\\.|[^\\"\n])*\'     { yylval = C_CHAR; return(C_CHAR); }

"..."     { yylval = OP_ASSIGN; return(OP_ASSIGN); }
">>="     { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"<<="     { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"+="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"-="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"*="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"/="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"%="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"&="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"^="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"|="      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
">>"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"<<"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"++"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"--"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"->"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"&&"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }
"||"      { yylval = OP_ASSIGN; return(OP_ASSIGN); }

"<"             { yylval = RELOP; return (RELOP);}
"<="          { yylval = RELOP; return (RELOP);}
"=="          { yylval = RELOP; return (RELOP);}
"!="          { yylval = RELOP; return (RELOP);}
">"             { yylval = RELOP; return (RELOP);}
">="          { yylval = RELOP; return (RELOP);}

"#"             { yylval = OP_OTHER; return (OP_OTHER);}
";"         { yylval = OP_OTHER; return (OP_OTHER);}
("{"|"<%")      { yylval = OP_OTHER; return (OP_OTHER);}
("}"|"%>")      { yylval = OP_OTHER; return (OP_OTHER);}
","         { yylval = OP_OTHER; return (OP_OTHER);}
":"         { yylval = OP_OTHER; return (OP_OTHER);}
"="         { yylval = OP_OTHER; return (OP_OTHER);}
"("         { yylval = OP_OTHER; return (OP_OTHER);}
")"         { yylval = OP_OTHER; return (OP_OTHER);}
("["|"<:")      { yylval = OP_OTHER; return (OP_OTHER);}
("]"|":>")      { yylval = OP_OTHER; return (OP_OTHER);}
"."         { yylval = OP_OTHER; return (OP_OTHER);}
"&"         { yylval = OP_OTHER; return (OP_OTHER);}
"!"         { yylval = OP_OTHER; return (OP_OTHER);}
"~"         { yylval = OP_OTHER; return (OP_OTHER);}
"-"         { yylval = OP_OTHER; return (OP_OTHER);}
"+"         { yylval = OP_OTHER; return (OP_OTHER);}
"*"         { yylval = OP_OTHER; return (OP_OTHER);}
"/"         { yylval = OP_OTHER; return (OP_OTHER);}
"%"         { yylval = OP_OTHER; return (OP_OTHER);}
"^"         { yylval = OP_OTHER; return (OP_OTHER);}
"|"         { yylval = OP_OTHER; return (OP_OTHER);}
"?"         { yylval = OP_OTHER; return (OP_OTHER);}

.         {yylval = ERRORCHAR; return ERRORCHAR;}

%%

int installID () {
  return ID;
}
```

```
int installNum () {
  return NUMBER;
}

int yywrap (){
  return 1;
}

void writeout(int c){
  switch(c){
    case ERRORCHAR: errorCnt++; fprintf(yyout, "(ERRORCHAR, %s)\n", yytext);break;
    case OP_ASSIGN:
    case OP_OTHER: opCnt++; fprintf(yyout, "(OP %s)\n", yytext); break;
    case RELOP: opCnt++; fprintf(yyout, "(RELOP %s)\n", yytext); break;
    case KEYWORD: keyCnt++; fprintf(yyout, "(KEYWORD, %s)\n", yytext); break;
    case NUMBER: numCnt++; fprintf(yyout, "(NUM %s)\n", yytext); break;
    case ID: idCnt++; fprintf(yyout, "(ID %s)\n", yytext); break;
    case STRING_LITERAL: strCnt++; fprintf(yyout, "(STRING_LITERAL %s)\n", yytext);break;
  case C_CHAR: charCnt++; fprintf(yyout, "(C_CHAR %s)\n", yytext); break;
    default:break;
  }
  return;
}

void comt()
{
  char c, prev = 0;

  while ((c = input()) != 0)      /* (EOF maps to 0) */
  {
    if (c == '/' && prev == '*')
      return;
    prev = c;
  }
  printf("unterminated comment\n");
}

int main (int argc, char ** argv){
  int c,j=0;
  if (argc>=2){
    if ((yyin = fopen(argv[1], "r")) == NULL){
      printf("Can't open file %s\n", argv[1]);
      return 1;
    }
    if (argc>=3){
      yyout=fopen(argv[2], "w");
    }
  }

  while (c = yylex()){
    writeout(c);
  }
  if(argc>=2){
    fclose(yyin);
    if (argc>=3) fclose(yyout);
  }
    fprintf(yyout,
          "\nTotal:\n%d keywords\n%d "
          "identifiers\n%d "
          "constants\n%d operator\n%d string\n%d char\n",
          keyCnt, idCnt, numCnt, opCnt, strCnt, charCnt);
  return 0;
}
```