**编译原理第一次实验报告**

**Experiment1 Lexical Analyse**

19计算机科学与技术（1）班 梅雨欣 2019329621004

目录

[一、实验要求 2](#_Toc86779486)

[二、相关知识 2](#_Toc86779487)

[2.1 Lex 2](#_Toc86779488)

[2.2 词法分析 5](#_Toc86779489)

[三、实验准备与设计 6](#_Toc86779490)

[3.1 第三方工具Linux、gcc、Flex 6](#_Toc86779491)

[3.2 词法分析器设计 7](#_Toc86779492)

[3.3 DFA 7](#_Toc86779493)

[四、完整代码 9](#_Toc86779494)

[五、实验结果 18](#_Toc86779495)

[六、实验总结 21](#_Toc86779496)

# 一、实验要求

**实验目的**

根据算术表达式的需求，设计并实现适用于表达式的词法分析器，读入源程序（表达式），根据构词规则拆分出单词。

一、确定其状态转换图，用状态转换图的方法实现词法分析。

二、用自动生成器实现词法分，析程序（可以是lex，但不限于lex）。

三、注意程序结构的设计思想将整个系统模块化，并考虑接口。

**2 个能力要求考核：**

1、调查研究（investigation）的能力：能够针对实验要求，设计一个完整的实验(20 分)。要求能够体现实验设计的科学性和完整性，分析解释实验数据（20 分）并能给出合理的结论。（30 分）

2、三方工具的使用能力。总结使用三方工具的优缺点。指出使用三方工具需要具备哪些能力。并且评价自已在本次实验中的表现。（30 分）

# 二、相关知识

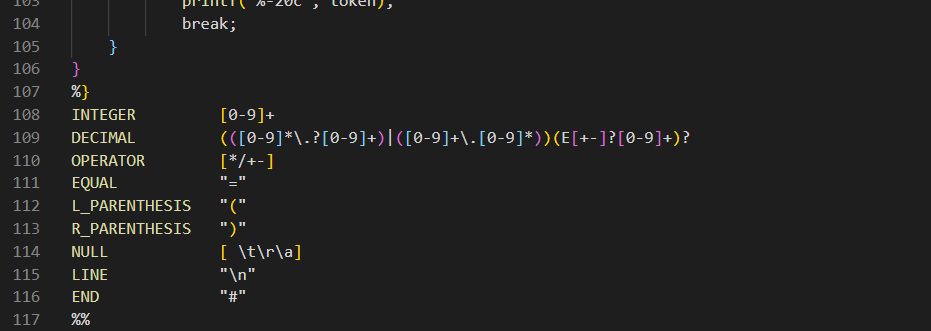
## 2.1 Lex

* Lex代表Lexical Analyzar。
* 使用如Flex的工具，生成程序叫词法分析器。
* 词法分析器将输入拆分成一个个记号（token），语法分析器根据已有的规则，分析这些token的组合，是否符合语法规范。
* 规则文件（.l文件）分为3个部分，每个部分之间使用%%分割。结构如下：

|  |
| --- |
| **声明定义部分（definition）**  %%  **识别规则部分（rules）**  %%  **辅助函数部分（user code）** |

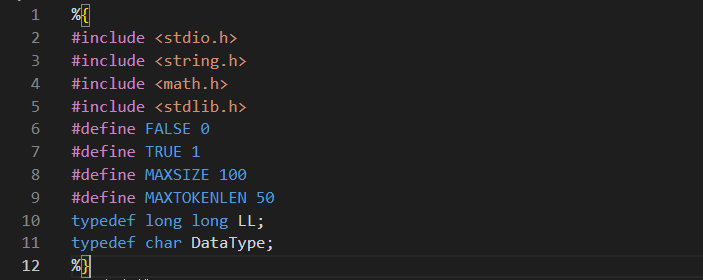
* **声明定义部分（definition）**

定义部分对规则部分要引用的文件和变量进行说明，通常可包含头文件、常数定义、全局变量定义、外部变量定义以及正规式定义等。正规式定义用来定义在规则部分引用的正规式，类似于C语言中的宏定义，所以也称**宏定义**。



图表 1 lex定义声明部分的宏定义

除了宏定义外，定义部分的其他代码需要用“%{”和“%}”括起来，LEX会将这部分内容直接复制到C文件生成的lex.yy.c文件中，通常用来放置include、define信息。例如：

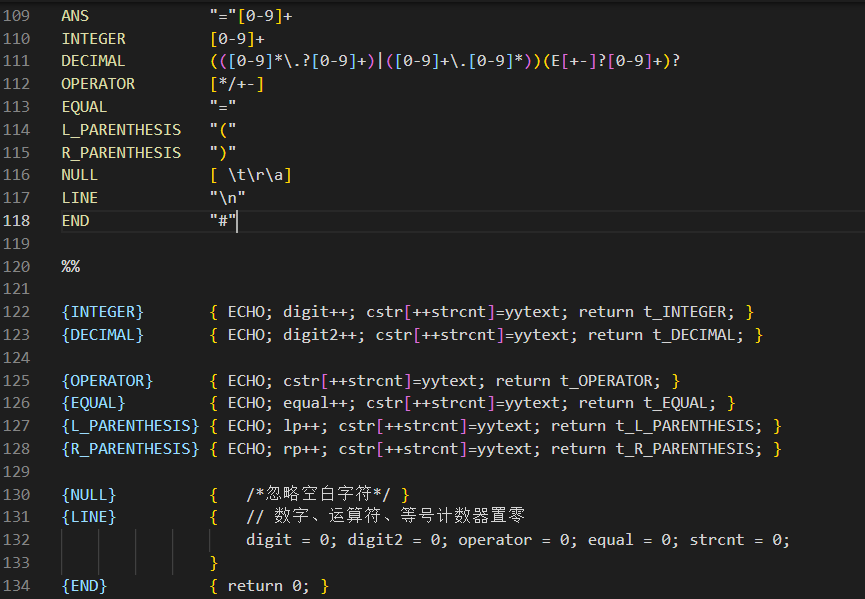


图表 2 lex定义声明部分的C部分

* **识别规则部分（rules）**

这部分用来定义对每一种词法单元的识别动作，也就是词法分析器要执行的操作。规则由模式（pattern）和动作（action）两个部分组成。模式是正则表达式，动作一般是C语句。模式指出了一个单词是如何构成的，当分析出一个符合该规则的单词时，就执行相应的动作。例如：

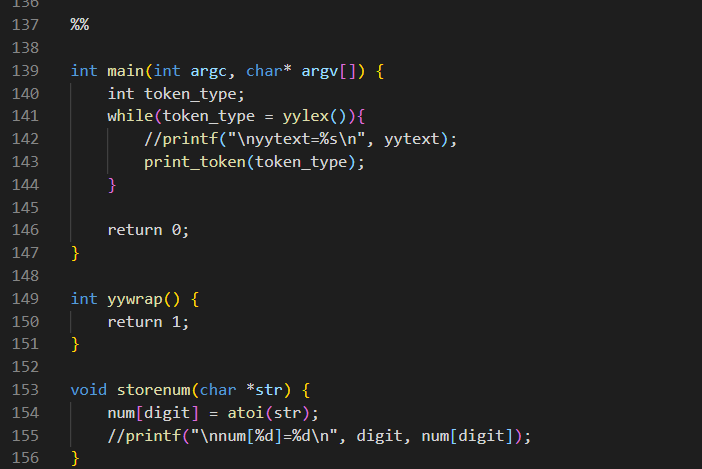
[0-9] {printf(“NUMBER %s\n”,yytext);}



图表 3 lex识别规则部分

* **辅助函数部分（user code）**

如果用到的函数不是库函数，则需要在这部分给出函数定义，由用户用C语言编写，它们会直接复制到输出的C程序文件（lex.yy.c）最后。例如：



图表 4 lex辅助函数部分

补充lex中常用变量和函数：

|  |  |
| --- | --- |
| **函数名** | **定义** |
| yyin | FILE\*类型，指向LEX输入文件，缺省情况下指向标准输入 |
| yyout | FILE\*类型，指向LEX输出文件，缺省情况下指向标准输出 |
| yytext | char\*类型，指向识别规则中的一个正规式匹配的单词的首字符 |
| yyleng | int类型，记录与识别规则中正规式匹配的单词的长度 |
| yylex() | 从该函数开始分析，可由lex自动生成 |
| yywrap() | 文件结束处理函数，如果返回值为1就停止解析。  可以用来解析多个文件。 |
| ECHO | 将yytext打印到yyout |

## 2.2 词法分析

**概念：**Flex是用来生成程序的工具，生成的程序叫词法分析器，也称为扫描器。其功能是从左往右逐个字符地对源程序进行扫描，然后按照源程序的构词规则识别单词符号，把作为字符串的源程序等价转换为单词符号串的中间程序。

**单词符号**是程序设计语言中的基本语法单元。词法分析阶段编译器从左向右扫描源文件，将其字符流分割成一个个的词（token、记号）。**token**就是源文件中不可再进一步分割的一串字符。

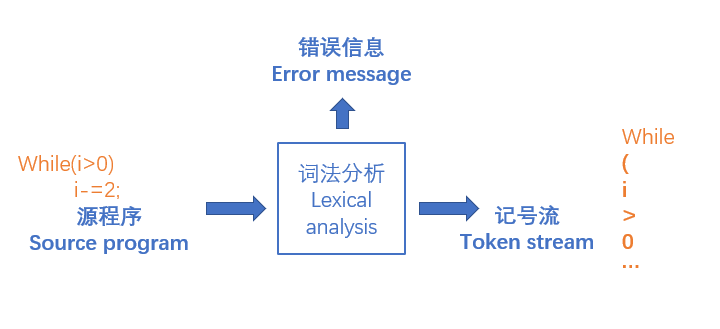
**词法单元**也叫种别码。词法单元中每个单词符号也叫**词素**，每个词素应该具备不同的属性值，这个值是反映单词特征或者特性的值，是编译中其他阶段需要的信息。如果一个词法单元只含有一个词素，那么不必设置属性值，如果一个词法单元含有多个词素，那么就应该给每个词素一个属性值用以区分他们。

一般来说程序语言中的token 有：常数（整数、小数等），操作符（算术操作符、比较操作符、逻辑操作符），分隔符（逗号、分号、括号等），标识符（变量名、函数名、类名等）等。如：

* 3和255是常数token
* “Apple”和“wail”是字符串token
* stack和queue是标识符token

上述的3、“Apple”、stack称为token的字面值。部分token只有一个字面值，如保留字的token，其他token则有不同字面值，如整数token。

词法分析器每扫描到一个完整的token后，立即新建一个TokenRecord，将此token的类型记录在此结构的type域中，将其字面值记录在value域中对应的子域内，并将TokenRecord结构传递给下一阶段的语法分析模块使用，然后接着扫描下一个token。这样从语法分析模块的角度来看，源程序就变成了一个连续的token stream了。流程如下图所示。



图表 5 词法分析流程

当有多条规则的模式被匹配到时，yylex会选择**匹配长度最长的那条规则**，如果有匹配长度相等的规则，则选择**排在最前面的规则**。

# 三、实验准备与设计

## 3.1 第三方工具Linux、gcc、Flex

在Linux系统下使用flex完成实验内容。因为Linux编译执行lex文件只需要3条命令，很**方便使用**。而Vistual Studio过于庞大，**占用内存**和**性能速度**远不如Linux直接使用terminal命令进行处理。

|  |
| --- |
| **lex exp1.lex（执行完后会生成lex.yy.c）**  **gcc -o exp1 lex.yy.c（执行完后可执行文件即为exp1）**  **./exp1（执行文件）** |

Flex是一个自动化的**快速词法分析生成器**，它可以将用户用正则表达式写的分词匹配模式构造成一个**有限状态自动机**，可以按照定义好的规则自动生成一个C函数yylex()，也成为扫描器（Scanner）。正则表达式中用双引号括起来的字符串就是原始字符串，里面的特殊字符是不需要转义的，而**双引号本身必须转义**（必须用“\”或\042），这是Flex中不同于常规的正则表达式的一个特性。



图表 6 flex编译流程

## 3.2 词法分析器设计

此次实验需要根据算术表达式的需求，设计并实现适用于表达式的词法分析器，读入源程序（表达式），根据构词规则拆分出单词。我们尝试设计一个能扫描算术表达式的词法分析器，单词分为以下几种：

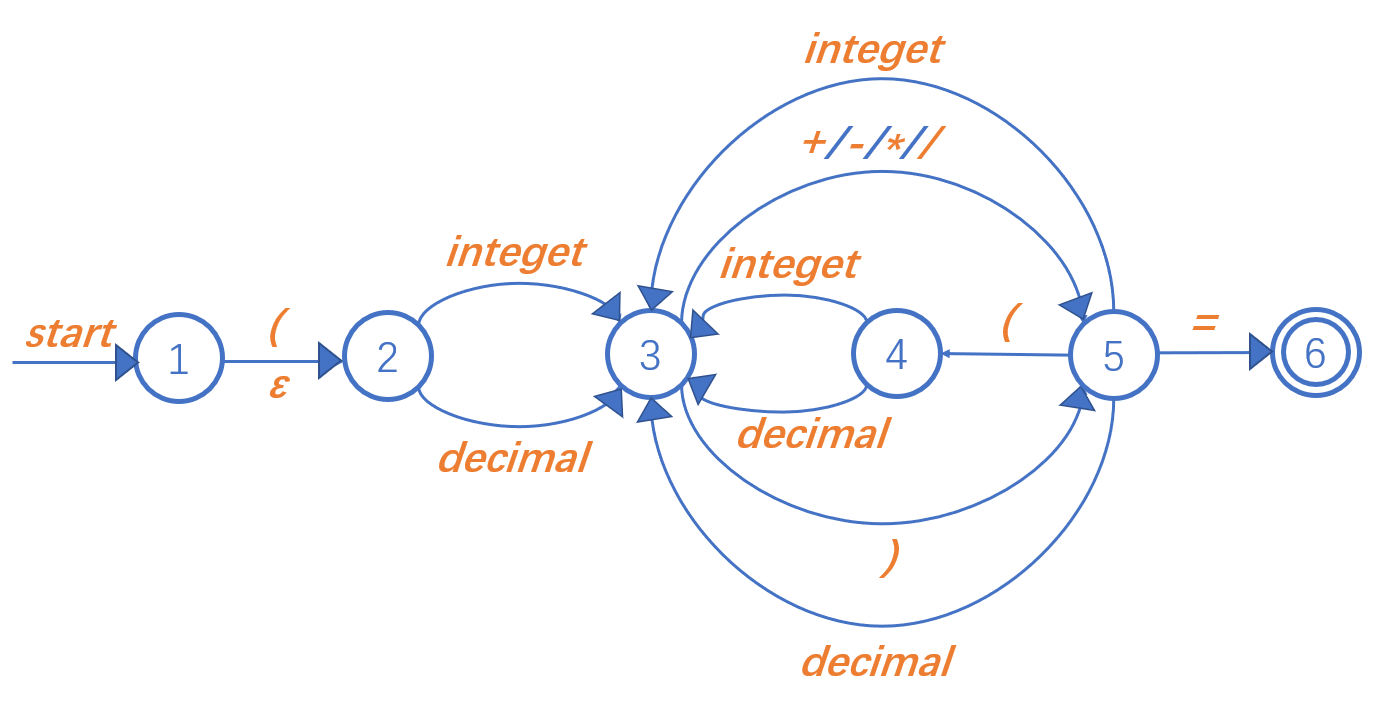
1. 整数interget：如123
2. 小数decimal：如123.456
3. 运算符：需要考虑优先级，加减乘除+-\*/
4. 左右括号：需要考虑优先级，(、)
5. 等号：=
6. 空白符：如空格tab等
7. 换行符：表示新一次扫描判断
8. 结束符：终止程序的执行

对于一个算术表达式来说，数字和运算符都没有限制数量，判断的流程也比较复杂，因此我使用等号“=”判断是否为一个完整算术表达式，定义等号行为是输出该算术表达式的正确答案。

## 3.3 DFA

确定有限状态自动机或确定有限自动机（英语：deterministic finite automaton，DFA）是一个能实现状态转移的自动机。对于一个给定的属于该自动机的状态和一个属于该自动机字母表的字符，它都能根据事先给定的转移函数转移到下一个状态（这个状态可以是先前那个状态）。

我们根据实验目的和词法分析器的设计画出以下DFA：

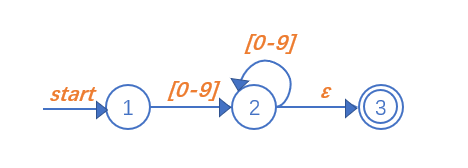


图表 7 算术表达式DFA

状态转换矩阵如下：

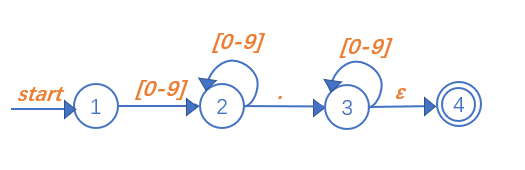
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **ε** | **(** | **)** | **integet** | **decimal** | **+/-/\*//** | **=** |
| **1** | 2 | 2 |  |  |  |  |  |
| **2** |  |  |  | 3 | 3 |  |  |
| **3** |  |  | 5 |  |  | 5 |  |
| **4** |  |  |  | 3 | 3 |  |  |
| **5** |  | 4 |  | 3 | 3 |  | 5 |
| **6** |  |  |  |  |  |  |  |

其中整数integet的DFA如下：



图表 8 整数integet的DFA

小数decimal的DFA如下：



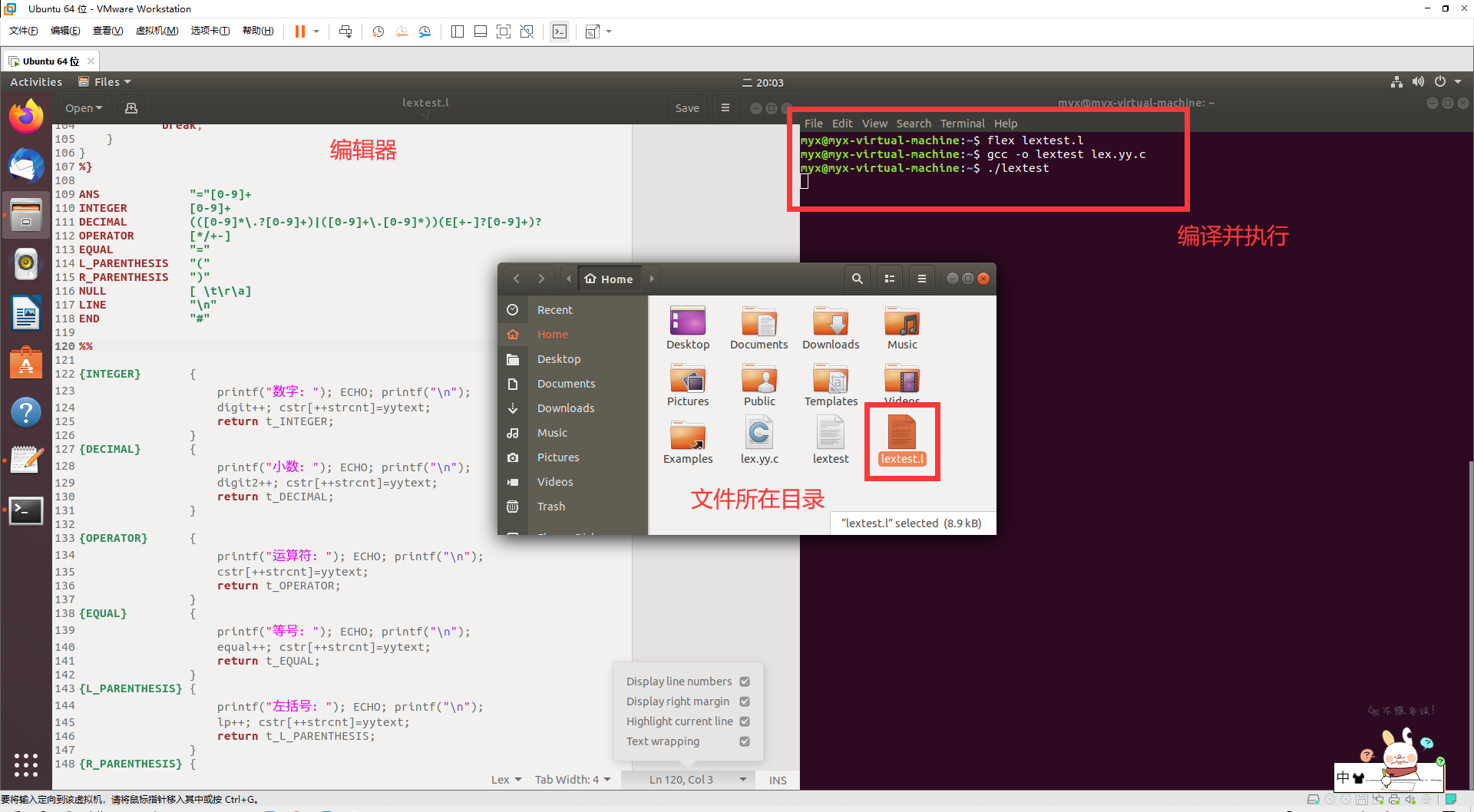
图表 9 小数decimal的DFA

# 四、完整代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425  426 | %{  #include <stdio.h>  #include <string.h>  #include <math.h>  #include <stdlib.h>  #define FALSE 0  #define TRUE 1  #define MAXSIZE 100  #define MAXTOKENLEN 50  typedef long long LL;  typedef char DataType;  // 定义栈  typedef struct stack{      DataType data[MAXSIZE];      int top;  }SeqStack;  // 操作数  typedef struct Operand{      int Data[MAXSIZE];      int top;  }Rand;  // 运算符  typedef struct Operator{      char Data[MAXSIZE];      int top;  }Rator;  typedef enum {      t\_ANS           = 1000, // 答案      t\_INTEGER       = 1001, // 整数，123      t\_DECIMAL       = 1002, // 小数，123.554      t\_OPERATOR      = 1003, // 操作符，+-/\*      t\_EQUAL         = 1004, // 等号      t\_L\_PARENTHESIS = 1005, // 左括号，(      t\_R\_PARENTHESIS = 1006  // 右括号，)  } TokenType;  //定义栈存放操作数和运算符  Rand operands;  Rator operators;  // 数字 运算符 等号 括号计数器  int digit = 0, digit2 = 0, operator = 0, equal = 0, lp = 0, rp = 0;  int num[MAXSIZE];  double dou[MAXSIZE];  char\* cstr[MAXSIZE] = {NULL};  int strcnt = 0;  char string[MAXSIZE\*10];  int stringtmp = 0;  void storenum(char \*str);                       // 存储整数  void storedouble(char \*str);                    // 存储小数  void InitOperand(Rand \*ra);                     //初始化栈  void InitOperator(Rator \*op);  int EmptyRand(Rand \*ra);                        //判栈空  int EmptyRator(Rator \*op);  int PushRand(Rand \*ra,int e);                   //进栈  int PushRator(Rator \*op,char e);  int PopRand(Rand \*ra,int \*e);                   //出栈  int PopRator(Rator \*op,char \*e);  int GetTopRand(Rand \*ra);                       //取栈顶元素  char GetTopRator(Rator \*op);  int InOp(char ch);                              //判断字符是否为运算符  int Priority(char s);                           //判断运算符优先级  int Precede(char op1,char op2);                 //比较运算符优先级  int Calculation(int a,int b,char c);            //判断符号并运算  int ExpCalculation(Rand \*ra,Rator \*op, char \*str);      //计算表达式  void InitStack(SeqStack \*s);                    // 初始化栈  int IsEmpty(SeqStack \*s);                       // 判断栈是否为空  int IsFull(SeqStack \*s);                        // 判断栈是否已满  void PopStack(SeqStack \*s, DataType temp);      // 出栈  void PushStack(SeqStack \*s,DataType temp);      // 压栈  void GetTopStack(SeqStack \*s , DataType \*temp); // 取栈顶元素  int BracketMatch(char \*str);                    // 括号是否匹配  // 根据token执行对应行为  static void print\_token(int token) {      switch(token) {          case 1000:              //while( num[digit] / 10 != 0){              //}              break;          case 1001:              storenum(yytext);              break;          case 1002:              storedouble(yytext);              break;          case 1003:              break;          case 1004:              InitOperand(&operands);              InitOperator(&operators);              ExpCalculation(&operands, &operators, cstr[1]);              EmptyRand(&operands);              EmptyRator(&operators);              break;          case 1005:              break;          case 1006:              break;          default:              printf("%-20c", token);              break;      }  }  %}  ANS             "="[0-9]+  INTEGER         [0-9]+  DECIMAL         (([0-9]\*\.?[0-9]+)|([0-9]+\.[0-9]\*))(E[+-]?[0-9]+)?  OPERATOR        [\*/+-]  EQUAL           "="  L\_PARENTHESIS   "("  R\_PARENTHESIS   ")"  NULL            [ \t\r\a]  LINE            "\n"  END             "#"  %%  {INTEGER}       {                      printf("数字: "); ECHO; printf("\n");                      digit++; cstr[++strcnt]=yytext;                      return t\_INTEGER;                  }  {DECIMAL}       {                      printf("小数: "); ECHO; printf("\n");                      digit2++; cstr[++strcnt]=yytext;                      return t\_DECIMAL;                  }  {OPERATOR}      {                      printf("运算符: "); ECHO; printf("\n");                      cstr[++strcnt]=yytext;                      return t\_OPERATOR;                  }  {EQUAL}         {                      printf("等号: "); ECHO; printf("\n");                      equal++; cstr[++strcnt]=yytext;                      return t\_EQUAL;                  }  {L\_PARENTHESIS} {                      printf("左括号: "); ECHO; printf("\n");                      lp++; cstr[++strcnt]=yytext;                      return t\_L\_PARENTHESIS;                  }  {R\_PARENTHESIS} {                      printf("右括号: "); ECHO; printf("\n");                      rp++; cstr[++strcnt]=yytext;                      return t\_R\_PARENTHESIS;                  }  {NULL}          { printf("空白字符\n"); }  {LINE}          {   // 数字、运算符、等号计数器置零                      printf("新的一行\n");                      digit = 0; digit2 = 0; operator = 0; equal = 0; strcnt = 0;                  }  {END}           { printf("结束\n"); return 0; }  <<EOF>>         { return 0; }  %%  int main(int argc, char\* argv[]) {      int token\_type;      while(token\_type = yylex()){          //printf("\nyytext=%s\n", yytext);          print\_token(token\_type);      }      return 0;  }  int yywrap() {      return 1;  }  void storenum(char \*str) {      num[digit] = atoi(str);      //printf("\nnum[%d]=%d\n", digit, num[digit]);  }  void storedouble(char \*str) {      char \*ptr;      dou[digit2] = strtod(str, &ptr);      //printf("\ndou[%d]=%lf\n", digit2, dou[digit2]);  }  //初始化栈  void InitOperand(Rand \*ra) {      ra->top = -1;  }  void InitOperator(Rator \*op) {      op->top = -1;  }  //栈空  int EmptyRand(Rand \*ra) {      ra->top = -1;      return 0;  }  int EmptyRator(Rator \*op) {      op->top = -1;      return 0;  }  //进栈  int PushRand(Rand \*ra,int e) {      if(ra->top == MAXSIZE-1)          return 0;      ra->top++;      ra->Data[ra->top] = e;      return 1;  }  int PushRator(Rator \*op,char e) {      if(op->top == MAXSIZE-1)          return 0;      op->top++;      op->Data[op->top] = e;      return 1;  }  //出栈  int PopRand(Rand \*ra,int \*e) {      if(ra->top == -1)          return 0;      \*e = ra->Data[ra->top];      ra->top--;      return 1;  }  int PopRator(Rator \*op,char \*e) {      if(op->top == -1)          return 0;      \*e = op->Data[op->top];      op->top--;      return 1;  }  //取栈顶元素  int GetTopRand(Rand \*ra) {      if(ra->top == -1)          return 0;      return ra->Data[ra->top];  }  char GetTopRator(Rator \*op) {      if(op->top == -1)          return 'N';      return op->Data[op->top];  }  //判断字符是否为运算符  int InOp(char ch) {  if(ch == '(' || ch == ')' || ch == '+' || ch == '-'  || ch == '\*' || ch == '/' || ch == '#')          return 1;      return 0;  }  //判断运算符优先级  int Priority(char s) {      switch(s) {      case '(':          return 4;      case '\*':      case '/':          return 3;      case '+':      case '-':          return 2;      case ')':          return 1;      default:          return 0;      }  }  //比较运算符优先级  int Precede(char op1,char op2) {      if(Priority(op1) < Priority(op2))          return 0;      return 1;  }  //判断符号并运算  int Calculation(int a, int b, char c) {      switch(c) {      case '+':          return a+b;      case '-':          return a-b;      case '\*':          return a\*b;      case '/':          if(b == 0)              exit(1);          return a/b;      }  }  //计算表达式  int ExpCalculation(Rand \*ra,Rator \*op, char \*str) {      int a, b, f = 0;      SeqStack st;      char s, x;      PushRator(op, '=');      InitStack(&st);      for(int j = 0 ; str[j] != '\0'; j++ ) {          switch(str[j]){              case '(':                  PushStack(&st, str[j]);                  break;              case ')':                  if(IsEmpty(&st)) {                      printf("\n右括号多余\n");                      f = 1;                  }else{                      GetTopStack(&st, &x);                      PopStack(&st, x);                  }                  break;          }      }      if(IsEmpty(&st)){          //printf("\n括号匹配\n");          f = 0;      }else{          printf("\n左括号多余\n");          f = 1;      }      if( f != 0 || str[yyleng-1] == '+'){          printf("false\n");          return 0;      }      int i = 0;      while(str[i] != '=' || GetTopRator(op) != '='){          if(!InOp(str[i])) {              int temp = str[i] - '0';              i++;              while(!InOp(str[i]) && str[i] != '=') {                  temp = temp \* 10 + str[i] - '0';                  i++;              }              PushRand(ra, temp);              if(str[i] == '=' || GetTopRator(op) == '(')break;          }          else {              if(str[i] == '=' || GetTopRator(op) == '(')break;              if(GetTopRator(op) != '(') {                  if(str[i] == ')')                      PopRator(op, &s);                  else                      PushRator(op, str[i]);                  i++;              }              else {                  if(!Precede(GetTopRator(op), str[i])) {                      PushRator(op,str[i]);                      i++;                  }else {                      PopRand(ra, &b);                      PopRand(ra, &a);                      PopRator(op, &s);                      PushRand(ra, Calculation(a, b, s));                  }              }          }  }      printf("=%d\n", GetTopRand(ra));      return 0;  }  // 判断两个括号是否匹配  int Match(char chone,char chtwo) {      if(chone=='(' && chtwo==')')          return TRUE;      else          return FALSE;  }  // 初始化栈  void InitStack(SeqStack \*s) {      s->top = -1;  }  // 判断栈是否为空  int IsEmpty(SeqStack \*s) {      if(s->top == -1)          return TRUE;      return FALSE;  }  //判断栈是否已满  int IsFull(SeqStack \*s) {      if(s->top == MAXSIZE-1)          return TRUE;      return FALSE;  }  // 出栈  void PopStack(SeqStack \*s, DataType temp) {      if(IsEmpty(s))          return;      temp = s->data[s->top];      s->top --;  }  // 压栈  void PushStack(SeqStack \*s,DataType temp) {      if(IsFull(s))          return;      s->top++;      s->data[s->top] = temp;  }  // 取栈顶元素  void GetTopStack(SeqStack \*s , DataType \*temp) {      if(IsEmpty(s))          return;      \*temp = s->data[s->top];  } |

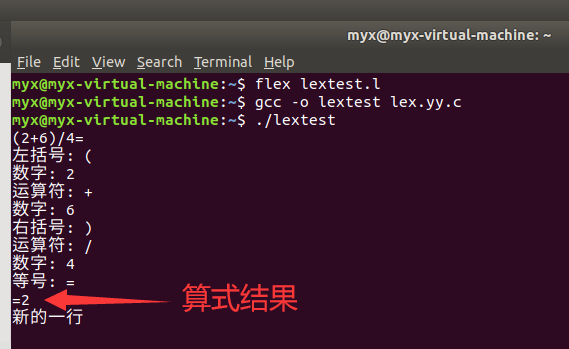
# 五、实验结果

预先设计该词法分析器能够识别各词，包括整数、运算符、等号、括号等，并执行对应指定行为，最后输出正确答案或提示非算式。有运算操作符优先级（\*/高于+-），可以判断多层括号并优先括号结果。



* **输入：(2+6)/4=**

, 简单含括号运算，应先计算括号内2+6再除以4，输出正确结果2，同时识别出各token并输出提示。



* **输入：((33+12)/2=**

输出false表示**不是算术表达式**，同时输出错误提示：“**左括号多余**”。



* **输入：5+14/7=**

输入算术表达式有**运算符号优先级**，应先算14/7，最后输出正确结果为7。



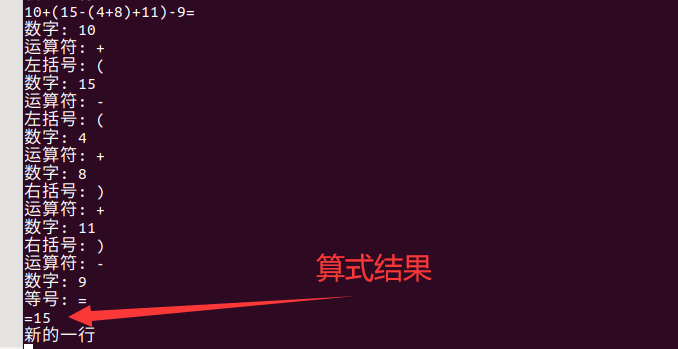
* **输入：10-(14/7+2)+8=**

输入算术表达式包括优先级与括号，最后正确输出算式结果为14。



* **输入：10+(15-(4+8)+11)-9=**

该式较复杂，多层括号嵌套的优先级处理，最后正确输出答案为15。



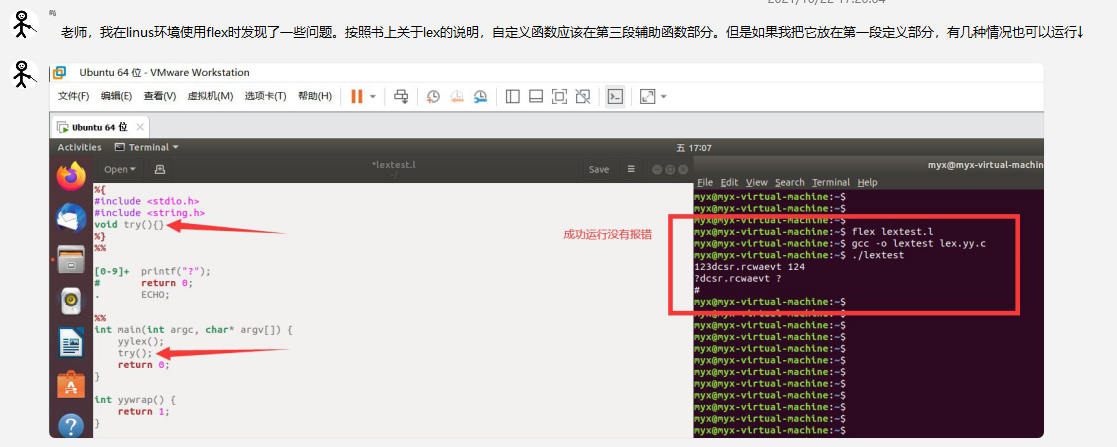
通过以上几个例子证明了该词法分析器设计成功，能够识别**多层括号嵌套**，还能**对括号、运算符的优先级**进行相应处理，最终**输出正确答案**，或因括号数目不匹配等原因**输出“false”**表示并非算术表达式。

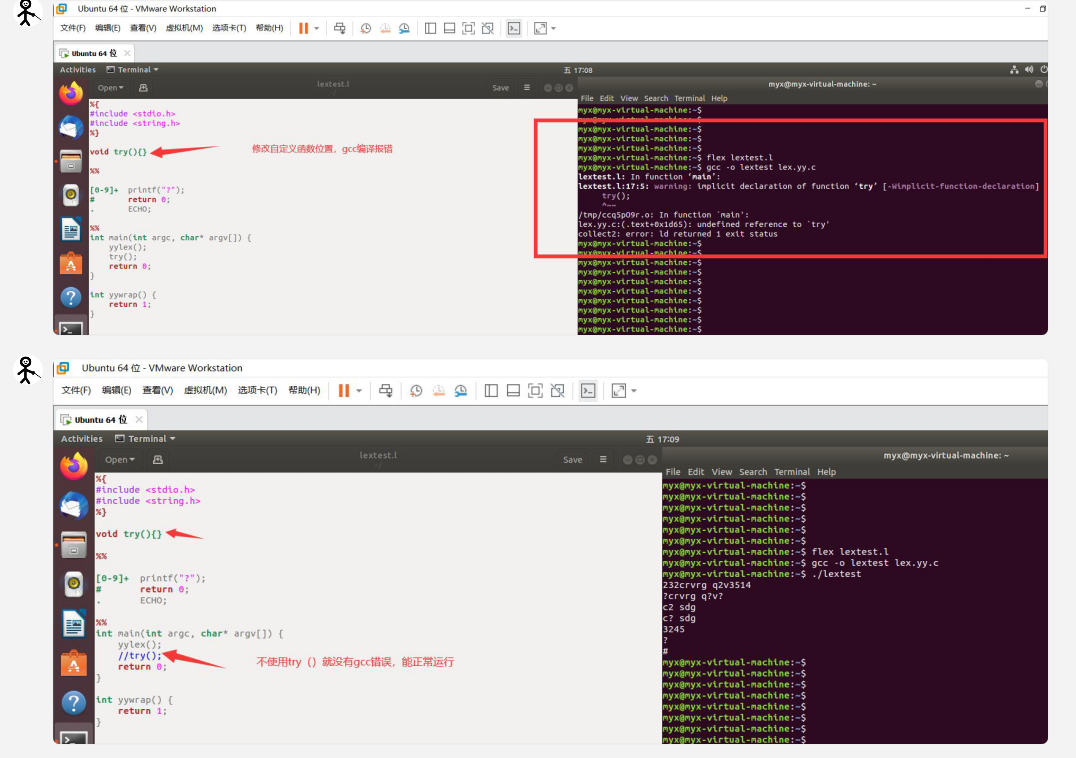
# 六、实验总结

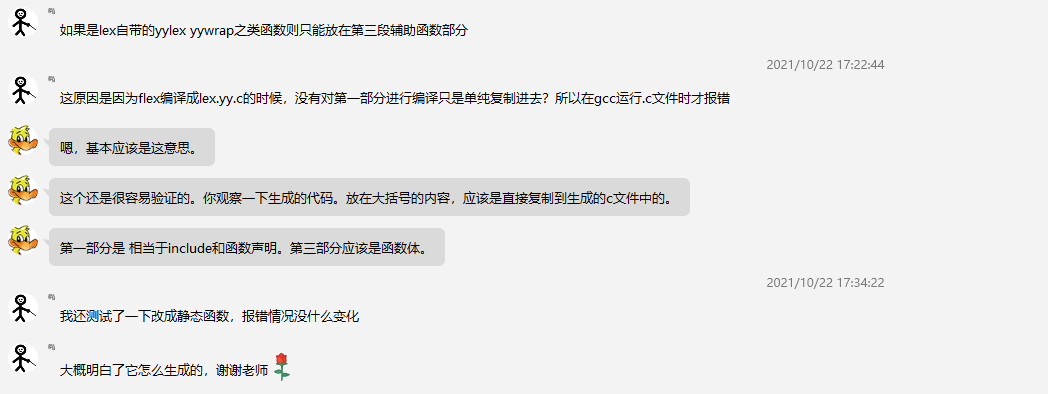
在此次实验中我也遇到了很多问题。对于实验中的flex工具是第一次进行使用，开始第一天上午踩了很多的坑，在最开始尝试使用Visual Studio来配置进行编译，我按照网上提供的方式对Visual Studio进行了配置。它将flex的.l文件编译**生成了.cpp文件**，并且编译后的.cpp文件可以正常运行，但是平台要求的格式是.c文件。因此我上网查询后在Visual Studio中设置输出文件的格式，设置不使用预编译头，这次再次编译后成功**生成了.c文件（lex.yy.c）**，并且可以正常运行。但是在一开始学习lex的过程中，我发现Vistual Studio固然方便，但是占用内存和速度方面还是**稍有欠缺**。

由于我的电脑是Windows10操作系统，于是我使用VMware的Linux操作系统，采用直接在terminal输入命令的方法，先后安装配置完成flex工具，进行测试也能运行成功，而且根据命令我也更好地理解flex的**编译方式**其实非常简单，在Windows或者说是Visual Studio中，只需要将写好的.l文件直接拖拽到flex\_win.exe上就会自动生成.c文件。

编写lex源代码文件的过程中我也有不少问题，通过查阅资料、自主测试和老师的解答下更深刻地理解了**lex三段结构**。







总的来说，在我完成这个实验过程中，感觉受益匪浅，不仅熟悉了正则表达式和DFA以及NFA之间的关系，更对编译器内部的词法扫描原理有了更加深刻的理解。