|  |
| --- |
| 编译原理  实验报告  （第 三 周）  **指导教师：杨冬**  **年 级：2019级**  **班 级：五班**  **小组编号： 2**  **组长学号姓名：55190521季圣鹏**  **组员学号姓名：55190516 孟令震**  **组员学号姓名：55190512 贾鹏程**  **2022年 6 月 2 日**  **软件学院** |

|  |  |  |
| --- | --- | --- |
| **完成实验内容** | | |
| 完成语法分析器。实现了读入token序列。动态地打印语法分析树的功能。  进行部分错误处理，进行 | | |
| **小组成员任务完成情况** | | |
| 姓名 | 具体完成任务 | 工作量  百分比 |
| 季圣鹏 | 完成语法分析器的编写。 | 90% |
| 孟令震 | 进行测试。  完成语法分析器的根节点函数。 | 5% |
| 贾鹏程 | 完成树型节点的定义。  实现数据结构定义。 | 5% |
| **小组成员协作情况** | | |
| 总体一起讨论设计，季圣鹏负责主要开发。协作情况较好。 | | |
| **实验中出现的问题及解决方案** | | |
| 1  将生成树节点时用malloc来分配内存的语句都换成用new分配内存。  原因是因为结构体的string类型不定长，不能动态分配内存。  malloc只是分配内存。 new除了分配内存还会调用构造函数的。  2.发现写语法分析器的时候必须加上,(小逗号)。不然程序遇到这种语句var integer v1,v2;肯定会错。所以回去重新修改词法分析器。  3.  参考编译程序的设计与实现这本书时，发现函数名节点没有单独作为一个节点，考虑到很有可能过程声明不止一个节点，我们设计时单独增加一个函数名节点HDeck， | | |
| **程序界面及运行截图** | | |
|  | | |
| **已完成部分的源程序代码** | | |
| //完成词法分析器  #include <iostream>  #include <vector>  #include <windows.h>  using namespace std;  #define distant 4 //控制\*的长度  //定义token序列的结构  struct Token {  string value1;  string value2;  };  //节点的具体类型  struct Specificnode {  string dec="";  string stmt="";  string exp="";  };  //定义语法分析树的节点  struct Treenode {  Treenode\* child[10];  Treenode\* sibling;  int idchild=0;//儿子的个数  string nodekind;//节点类型  Specificnode specificnode;//详细类型  int idnum=0;//标识符个数  string value;//节点的值（有的节点只有一个标识符。也用value了）  string id[10];//部分标识符的值  };  //定义语法分析器的函数  Treenode\* programhead(); //程序头  Treenode\* declarehead(); //声明  //Treenode\* programbody(); //程序体  Treenode\* typedec(); //类型声明  Treenode\* vardec(); //变量声明  Treenode\* procdec();//过程声明  Treenode\* typedeclist(); //类型声明中的部分函数  Treenode\* vardeclist(); //变量声明中的部分函数  Treenode\* paramdeclist(); //过程声明中的部分函数  Treenode\* paramlist(); //过程声明中的形参函数  //Treenode\* procdecpart(); //过程声明中的变量声明  Treenode\* probody(); //过程声明中的函数体,后面可以直接用作程序体  //Treenode\* stmtlist(); //语句列表  Treenode\* stmt(); //生成一个语句节点  Treenode\* assign1(); //生成赋值表达式  Treenode\* write1(); //生成读写表达式  Treenode\* read1(); //生成读写表达式  Treenode\* if1(); //生成选择表达式  //Treenode\* stmtmore(); //生成更多的语句节点  //全局变量  int size1= 0;//token序列的长度  vector<Token> token;//token序列  int subscript=0;//语法分析程序中的下标，标记读到哪个token序列  int size2 = 0;//这个用来控制树的层次结构  //分界符的命名+ | - | \*| / | ( | ) | [ | ] | ; | . | < | : | = | ' | := | > | " | ,  char SingleDelimiter[18][20]= {"PLUS","MINUS","TIMES","OVER","LPAREN","RPAREN",  " LMIDPAREN","RMIDPAREN", "SEMI","DOT","LT","COLON","EQ","COMMA","ASSIGN","RT","SY","JSP1"};  // 保留字的命名,为了实现映射关系  string reservedWords[21] = { "program","type","var","procedure","begin","end","array","of","record","if","then","else",  "fi","while","do","endwh","read","write","return","integer","char" };  string reservedWords1[21] = { "PROGRAM","TYPE","VAR","PROCEDURE","BEGIN","END","ARRAY","OF","RECORD","IF","THEN","ELSE",  "FI","WHILE","DO","ENDWH","READ","WRITE","RETURN","INTEGER","CHAR" };  //打印出错函数  void printwrong() {  cout << "你输入的程序段有词法错误" << endl;  }  //打印语法树的空格  void printq(int size3) {  //Sleep(1000);  for (int w = 0; w < distant \* size3; w++) {  cout << " ";  }  }  //打印token序列到控制台  void printtoken(vector<Token>token) {  cout << "输入程序段经过词法分析器之后的Token序列为:" << endl;  for (int i = 0; i < size1-1; i++) {  cout << token[i].value1 << "," << token[i].value2 << endl;  }  cout << token[size1 - 1].value1 << "," << token[size1 - 1].value2 << endl<<"EOF";    }  //通过一个连续字符串,生成token序列,词法分析器的核心程序  void generatetoken(vector<char> input, int len) {  int i = 0;  while (i < len) {  //如果遇到'\n'和空格  if (input[i] == ' ' || input[i] == '\n') {  i = i + 1;  continue;  }  //如果遇到数字  if (input[i] >= '0' && input[i] <= '9') {  string tmp = "";  tmp += input[i];  int j = i + 1;  while (j < len) {  if (input[j] >= '0' && input[j] <= '9') {  tmp += input[j];  j++;  }  else {  i = j;  break;  }  }  //下面要判断他是数学下标还是数字，这个地方的下标已经是i+1了，所以不用再变  if (i >= len) {  Token token1;  token1.value1 = "NUM";  token1.value2 = tmp;  size1 = size1 + 1;  token.push\_back(token1);  }  else {  //如果这个右边是]，则判断他是数组类型的下表  if (input[i] == ']') {  Token token1;  token1.value1 = "UNDERANGE";  token1.value2 = tmp;  size1 = size1 + 1;  token.push\_back(token1);  }  else {  Token token1;  token1.value1 = "NUM";  token1.value2 = tmp;  size1 = size1 + 1;  token.push\_back(token1);  }  }  continue;  }  //如果遇到字符  if ((input[i] >= 'a' && input[i] <= 'z') || (input[i] >= 'A' && input[i] <= 'Z')) {  string tmp = "";  tmp += input[i];  int j = i + 1;  while (j < len) {  if ((input[j] >= 'a' && input[j] <= 'z') || (input[j] >= 'A' && input[j] <= 'Z') || (input[j] >= '0' && input[j] <= '9')) {  tmp += input[j];  j = j + 1;  }  else {  i = j;  break;  }  }  //cout << "\n" << "\*\*\*\*\*\*" << tmp << endl;  if (tmp.length()==1) { //如果是char类型的  Token token1;  token1.value1 = "CHAR";  token1.value2 = tmp;  size1 = size1 + 1;  token.push\_back(token1);  }  else {  //判断他个是保留字  bool flag2 = false;  for (int k = 0; k < 21; k++) {  if (tmp == reservedWords[k]) {  Token token1;  token1.value1 = "reserved word";  token1.value2 = reservedWords1[k]; //这里将保留字的小写转变成大写  size1 = size1 + 1;  token.push\_back(token1);  flag2 = true;  break;  }  }  //如果他就是普通字符串  if (!flag2) {  Token token1;  token1.value1 = "ID";  token1.value2 = tmp;  size1 = size1 + 1;  token.push\_back(token1);  }  }  continue;  }  //如果遇到左注释和出错处理  if (input[i] == '{') {  int j = i; //通过双指针找到下一个}  bool flag = false; //标识是否找到了}  while (j < len) {  if (input[j] == '}') {  i = j + 1;  flag = true;  break;  }  j = j + 1;  }  if (!flag) {  printwrong();  }  continue;  }  //如果遇到右注释,出错处理  if (input[i] == '}') {  i = i + 1;  continue;  }  //如果遇到单分界符+  if (input[i] == '+') {  Token token1;  token1.value1 = SingleDelimiter[0];  token1.value2 = '+';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符-  if (input[i] == '-') {  Token token1;  token1.value1 = SingleDelimiter[1];  token1.value2 = '-';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符\*  if (input[i] == '\*') {  Token token1;  token1.value1 = SingleDelimiter[2];  token1.value2 = '\*';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符/  if (input[i] == '/') {  Token token1;  token1.value1 = SingleDelimiter[3];  token1.value2 = '/';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符(  if (input[i] == '(') {  Token token1;  token1.value1 = SingleDelimiter[4];  token1.value2 = '(';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符)  if (input[i] == ')') {  Token token1;  token1.value1 = SingleDelimiter[5];  token1.value2 = ')';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符[  if (input[i] == '[') {  Token token1;  token1.value1 = SingleDelimiter[6];  token1.value2 = '[';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符]  if (input[i] == ']') {  Token token1;  token1.value1 = SingleDelimiter[7];  token1.value2 = ']';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符:  if (input[i] == ';') {  Token token1;  token1.value1 = SingleDelimiter[8];  token1.value2 = ';';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符.  if (input[i] == '.') {  Token token1;  token1.value1 = SingleDelimiter[9];  token1.value2 = '.';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符<  if (input[i] == '<') {  Token token1;  token1.value1 = SingleDelimiter[10];  token1.value2 = '<';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符:这个地方得考虑一下是不是:=  if (input[i] == ':') {  int j = i + 1;  if (j >= len) {  Token token1;  token1.value1 = SingleDelimiter[11];  token1.value2 = ':';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  }  else {  if (input[j] == '=') {  Token token1;  token1.value1 = SingleDelimiter[14];  token1.value2 = ":=";  size1 = size1 + 1;  token.push\_back(token1);  i = i + 2;  }  else {  Token token1;  token1.value1 = SingleDelimiter[11];  token1.value2 = ':';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  }  }  continue;  }  //如果遇到单分界符=  if (input[i] == '=') {  Token token1;  token1.value1 = SingleDelimiter[12];  token1.value2 = '=';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符'  if (input[i] == '\'') {  Token token1;  token1.value1 = SingleDelimiter[13];  token1.value2 = '\'';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符>  if (input[i] == '>') {  Token token1;  token1.value1 = SingleDelimiter[15];  token1.value2 = '>';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符"  if (input[i] == '"') {  Token token1;  token1.value1 = SingleDelimiter[16];  token1.value2 = '"';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //如果遇到单分界符,  if (input[i] == ',') {  Token token1;  token1.value1 = SingleDelimiter[17];  token1.value2 = ',';  size1 = size1 + 1;  token.push\_back(token1);  i = i + 1;  continue;  }  //printwrong();  i = i + 1;  }  }  //通过下标，每次得到一个token  Token gettoken(int m) {  return token[m];  }  //创建语法分析树的根节点  Treenode\* program() {  Treenode\* root = new Treenode;  if (root) {  cout << "\n" << "程序经过语法分析器之后的语法分析树为:" << endl;  cout << "Prok"<<endl;  root->idchild = 3;  root->child[0] = programhead();  root->child[1] = declarehead();  root->child[2] = probody();  root->sibling = NULL;  root->nodekind = "Prok";//节点类型为根节点  if (gettoken(subscript).value1 == "DOT") {  cout << "语法分析树生成成功";  return root;  }  else {  return NULL;  }  }  }  //创建程序头头部分析函数节点  Treenode\* programhead() {  Treenode\* tmp ;  tmp = new Treenode;  if (tmp) {  if (token[subscript].value2 == "PROGRAM") {  size2 = size2 + 1;  printq(size2);//调用打印空格  cout << "PheadK"<<" ";  tmp->idchild = 0;  tmp->nodekind = "PheadK";  //tmp->sibling = declarehead();  subscript = subscript + 1;  cout << token[subscript].value2<<endl;  size2 = size2 - 1;//回退  tmp->value = token[subscript].value2;  subscript = subscript + 1;  }  else {  return NULL;  }  }  return tmp;  }  //创建程序声明函数节点  Treenode\* declarehead() {  Treenode\* tmp1 = new Treenode;  Treenode\* tmp2 = new Treenode;  Treenode\* tmp3 = new Treenode;;  Treenode\* tp1 = typedec();  Treenode\* tp2 = vardec();  Treenode\* tp3 = procdec();  //对过程声明节点的细化  if (tmp3) {  tmp3->child[0] = tp3;  tmp3->sibling = NULL;  tmp3->nodekind = "Proc";  }  //对变量声明节点的细化  if (tmp2) {  tmp2->child[0] = tp2;  tmp2->sibling = tmp3;  tmp2->nodekind = "Varc";  }  //对类型声明节点的细化  if (tmp1) {  tmp1->child[0] = tp1;  tmp1->nodekind = "TypeK";  tmp1->sibling = tmp2;  }  //cout << 3;  //判断不同情况得到的返回值  if (tp1 != NULL) {  return tmp1;  }  else {  if (tp2 != NULL) {  return tmp2;  }  else {  return tmp3;//不管是不是NULL都可以这么返回  }  }  return NULL;  }  //  ////创建程序体函数节点  //Treenode\* programbody() {  // return NULL;  //}  //类型声明部分处理程序  Treenode\* typedec() {  Treenode\* t = new Treenode;  if (token[subscript].value2 == "TYPE") {  size2 = size2 + 1;  printq(size2);//调用打印空格  cout << "TYPE" << endl;  subscript=subscript+1;  //生成类型声明节点  t->nodekind = "Typek";  t->child[0] = typedeclist();  size2 = size2 - 1;//注意回退的时机  return t;  }  else return NULL;  }  //变量声明处理程序  Treenode\* vardec() {  Treenode\* t = new Treenode;  if (token[subscript].value2 == "VAR") {  size2 = size2 + 1;  printq(size2);//调用打印空格  cout << "VAR" << endl;  subscript = subscript + 1;  //生成类型声明节点  t->nodekind = "Vark";  t->child[0] = vardeclist();  size2 = size2 - 1;//注意回退的时机  return t;  }  else return NULL;  }  //函数声明处理程序  Treenode\* procdec() {  Treenode\* t = new Treenode;  if (token[subscript].value2 == "PROCEDURE") {  size2 = size2 + 1;  printq(size2);//调用打印空格  cout << "PROCEDURE" << endl;  subscript = subscript + 1;  //生成类型声明节点  t->nodekind = "PROCEDURE";  t->child[0] = paramdeclist();  size2 = size2 - 1;//注意回退的时机  return t;  }  else return NULL;  }  //类型声明的具体函数  Treenode\* typedeclist() {  if (token[subscript].value2 == "VAR" || token[subscript].value2 == "PROCEDURE" ||  token[subscript].value2 == "BEGIN" || token[subscript].value1 == "DOT") {//这里要细节一点记住value1和value2  return NULL;  }  else {  Treenode\* t = new Treenode;  if (t) {  //生成信息  t->value = token[subscript].value2;  subscript = subscript + 2;  t->nodekind = "Deck";  t->specificnode.dec = token[subscript].value2;  subscript = subscript + 2;  //打印  size2 = size2 + 1;  printq(size2);  cout << t->nodekind << " " << t->specificnode.dec << " " << t->value<<endl;  size2 = size2 - 1;//注意回退时机  //递归调用  t->sibling = typedeclist();  }  return t;  }  }  //变量声明的具体函数  Treenode\* vardeclist() {  if (token[subscript].value2 == "PROCEDURE" || token[subscript].value2 == "BEGIN" || token[subscript].value1 == "DOT") {//这里要细节一点记住value1和value2  return NULL;  }  else {  Treenode\* t = new Treenode;  if (t) {  //生成这个节点的具体域  t->nodekind = "Deck";  t->specificnode.dec = token[subscript].value2;  subscript = subscript + 1;  while (token[subscript].value1 != "SEMI") {  //当这个token序列中有,而不是;时  t->id[t->idnum] = token[subscript].value2;  t->idnum = t->idnum + 1;  subscript = subscript+1;  //这里必须拉出来单独判断  if (token[subscript].value1 == "JSP1") {  subscript = subscript + 1;  }  }  //从;到下一个token  subscript = subscript + 1;  //打印序列  size2 = size2 + 1;  printq(size2);  cout << "DecK" << " " << t->specificnode.dec;  for (int w = 0; w < t->idnum; w++) {  cout << " " << t->id[w];  }  cout << endl;  size2 = size2 - 1;  //递归调用  t->sibling = vardeclist();  }  }  }  //过程声明中的部分函数  Treenode\* paramdeclist() {  if (token[subscript].value1 == "ID" || token[subscript].value1 == "CHAR") {  //生成函数名节点部分信息  Treenode\* t = new Treenode;  t->nodekind = "HDeck";//这个地方是我自创的一个函数名节点  t->value = token[subscript].value2;  subscript = subscript + 1;  //打印信息  size2 = size2 + 1;  printq(size2);  cout << "HDeck" <<" "<< t->value<<endl;  //生成形参，变量声明，函数体部分，作为儿子节点  t->child[0]= paramlist(); //过程声明中的形参函数  t->child[1]= vardec(); //过程声明中的变量声明,这个是之前写好了的  t->child[2]= probody(); //过程声明中的函数体  size2 = size2 - 1;  return t;  }  else return NULL;  }  //过程声明中的形参函数  Treenode\* paramlist() {  //进去之后首先是一个左括号  subscript = subscript + 1;  //当有好多个形参时,生成节点  Treenode\* t = new Treenode;  while (token[subscript].value1 != "RPAREN") {  if (token[subscript].value1 == "JSP1") {  //遇到,就跳过  subscript = subscript + 1;  }  t->id[t->idnum] = token[subscript].value2;  t->idnum = t->idnum + 1;  subscript = subscript + 1;  }  //现在token到右括号了  subscript = subscript + 2;  //打印信息  size2 = size2 + 1;  printq(size2);  cout << "DecK" << " " << "value" << " " << "param";  for (int w = 0; w < t->idnum; w++) {  cout << " " << t->id[w];  }  cout << endl;  size2 = size2 - 1;  return t;  }  //过程声明中的函数体  Treenode\* probody() {  Treenode\* t = new Treenode;  if (token[subscript].value2 == "BEGIN") {  size2 = size2 + 1;  printq(size2);//调用打印空格  cout << "StmLK" << endl;  subscript = subscript + 1;  //生成函数体类型声明节点  t->nodekind = "StmLK";  //这边进行判断，是不是语句都结束了  while (token[subscript].value2 != "END" && token[subscript].value2 != "ENDWH") {  t->child[t->idchild] = stmt();  t->idchild = t->idchild + 1;  }  subscript = subscript + 1;  size2 = size2 - 1;//注意回退的时机  return t;  }  else return NULL;  }  ////返回语句列表  //Treenode\* stmtlist() {  // Treenode\* tmp1 = new Treenode;  // Treenode\* tmp2 = new Treenode;  // tmp1 = stmt();  // tmp2 = stmtmore();  // tmp1->sibling = tmp2;  // return tmp1;  //}  //得到一个语句节点  Treenode\* stmt(){  if (token[subscript].value1 == "ID"||token[subscript].value1=="CHAR") {  //说明是赋值节点语句,生成赋值节点  Treenode\* tmp1 = new Treenode;  tmp1->nodekind = "StmtK";  tmp1->specificnode.stmt = "AssignK";  //打印这个  size2 = size2 + 1;  printq(size2);  //这边加一些判断来区分函数调用和赋值语句  if (token[subscript+1].value1 == "LPAREN") {  cout << "StmtK" << " " << "CALL" << endl;  }  else {  cout << "StmtK" << " " << "AssignK" << endl;  }  //cout << "StmtK" << " " << "AssignK" << endl;  //继续生成赋值节点细的部分；  tmp1->child[0] = assign1();  size2 = size2 - 1;  return tmp1;  }  else {  if (token[subscript].value2 == "WRITE") {  //说明是写出节点  Treenode\* tmp1 = new Treenode;  tmp1->nodekind = "StmtK";  tmp1->specificnode.stmt = "WRITE";  subscript = subscript + 1;  //打印这个  size2 = size2 + 1;  printq(size2);  cout << "StmtK" << " " << "WRITE" << endl;  //继续生成写节点细的部分；  tmp1->child[0] = write1();  size2 = size2 - 1;  return tmp1;  }  else {  if (token[subscript].value2 == "READ") {  //说明是读入节点  Treenode\* tmp1 = new Treenode;  tmp1->nodekind = "StmtK";  tmp1->specificnode.stmt = "READ";  subscript = subscript + 1;  //打印这个  size2 = size2 + 1;  printq(size2);  cout << "StmtK" << " " << "READ" << endl;  //继续生成读节点细的部分；  tmp1->child[0] = read1();  size2 = size2 - 1;  return tmp1;  }  else {  if (token[subscript].value2 == "IF") {  //说明是判断节点  Treenode\* tmp1 = new Treenode;  tmp1->nodekind = "StmtK";  tmp1->specificnode.stmt = "IF";  subscript = subscript + 1;  size2 = size2 + 1;  printq(size2);  cout << "StmtK" << " " << "IF" << endl;  //继续生成选择节点if判断部分；  tmp1->child[0] = if1();  //打印then部分  size2 = size2 + 1;  printq(size2);  cout << "StmtK" << " " << "THEN" << endl;  subscript = subscript + 1;  tmp1->child[2] = assign1();  size2 = size2 - 1;  //如果有else  if (token[subscript].value2 == "ELSE") {  size2 = size2 + 1;  printq(size2);  cout << "StmtK" << " " << "ELSE" << endl;  subscript = subscript + 1;  tmp1->child[3] = assign1();  size2 = size2 - 1;  }  //对于FI的处理  subscript = subscript + 2;  //别忘了一开始的if判断  size2 = size2 - 1;  return tmp1;  }  else {  subscript = subscript + 1;  return NULL;  }  }  }    }  }  //生成赋值表达式节点  Treenode\* assign1() {  //想的简单一点，就遇到;就停止  Treenode\* tmp1 = new Treenode;  while (token[subscript].value1 != "SEMI"&& token[subscript].value2 != "END" && token[subscript].value2 != "ENDWH"  && token[subscript].value2 != "ELSE" && token[subscript].value2 != "FI") {  if (token[subscript].value1 == "JSP1" || token[subscript].value1 == "ASSIGN" || token[subscript].value1 == "EQ"  || token[subscript].value1 == "LPAREN" || token[subscript].value1 == "RPAREN"||token[subscript].value1=="SY"  ||token[subscript].value1=="COMMA") {  subscript = subscript + 1;  }  else {  tmp1->nodekind = "ExpK";  tmp1->specificnode.stmt = "IdK";  size2 = size2 + 1;  printq(size2);  //这边对赋值语句的不同符号进行区分  //加减乘除  if ((token[subscript].value1 == "PLUS") || (token[subscript].value1 == "MINUS") || (token[subscript].value1 == "TIMES")  || (token[subscript].value1 == "OVER")) {  cout << "ExpK" << " " << "OP" << " " << token[subscript].value2 << endl;  }  else {  //整数  if (token[subscript].value1 == "NUM") {  cout << "ExpK" << " " << "const" << " " << token[subscript].value2 << endl;  }  else {  cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  }  //cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  }  subscript = subscript + 1;  size2 = size2 - 1;  }  //tmp1->nodekind = "ExpK";  //tmp1->specificnode.stmt = "IdK";  //size2 = size2 + 1;  //printq(size2);  //if ((token[subscript].value1 == "PLUS") || (token[subscript].value1 == "MINUS") || (token[subscript].value1 == "TIMES")  // || (token[subscript].value1 == "OVER")) {  // cout << "ExpK" << " " << "OP" << " " << token[subscript].value2 << endl;  //}  //else {  // cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  //}  //subscript = subscript + 1;  //size2 = size2 - 1;  }  if (token[subscript].value1 == "SEMI") {  subscript = subscript + 1;  }  return tmp1;  }  //生成写表达式  Treenode\* write1() {  Treenode\* tmp1 = new Treenode;  while ((token[subscript].value1 != "SEMI" && token[subscript].value2 != "END" && token[subscript].value2 != "ENDWH")) {  if (token[subscript].value1 == "LPAREN" || token[subscript].value1 == "RPAREN") {  subscript = subscript + 1;  }  else {  tmp1->nodekind = "ExpK";  tmp1->specificnode.stmt = "IdK";  size2 = size2 + 1;  printq(size2);  cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  subscript = subscript + 1;  size2 = size2 - 1;  }  }  if (token[subscript].value1 == "SEMI") {  subscript = subscript + 1;  }  return tmp1;  }  //生成读表达式  Treenode\* read1() {  Treenode\* tmp1 = new Treenode;  while ((token[subscript].value1 != "SEMI" && token[subscript].value2 != "END" && token[subscript].value2 != "ENDWH")) {  if (token[subscript].value1 == "LPAREN" || token[subscript].value1 == "RPAREN") {  subscript = subscript + 1;  }  else {  tmp1->nodekind = "ExpK";  tmp1->specificnode.stmt = "IdK";  size2 = size2 + 1;  printq(size2);  cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  subscript = subscript + 1;  size2 = size2 - 1;  }  }  if (token[subscript].value1 == "SEMI") {  subscript = subscript + 1;  }  return tmp1;  }  //生成if表达式  Treenode\* if1() {  Treenode\* tmp1 = new Treenode;  while (token[subscript].value2 != "THEN") {  if (token[subscript].value1 == "LPAREN" || token[subscript].value1 == "RPAREN") {  subscript = subscript + 1;  }  else {  tmp1->nodekind = "ExpK";  size2 = size2 + 1;  printq(size2);  //这边对赋值语句的不同符号进行区分  //加减乘除  if ((token[subscript].value1 == "PLUS") || (token[subscript].value1 == "MINUS") || (token[subscript].value1 == "TIMES")  || (token[subscript].value1 == "OVER")|| (token[subscript].value1 == "LT")|| (token[subscript].value1 == "RT")) {  cout << "ExpK" << " " << "OP" << " " << token[subscript].value2 << endl;  }  else {  //整数  if (token[subscript].value1 == "NUM") {  cout << "ExpK" << " " << "const" << " " << token[subscript].value2 << endl;  }  else {  cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  }  //cout << "ExpK" << " " << token[subscript].value2 << " " << "IdK" << endl;  }  subscript = subscript + 1;  size2 = size2 - 1;  }  }  return tmp1;  }  ////得到更多的语句节点  //Treenode\* stmtmore() {  // if (token[subscript].value2 == "END" || token[subscript].value2 == "ENDWH") {  // subscript = subscript + 1;  // return NULL;  // }  // else {  //  // }  //}  int main() {  //存放输入字符串  vector<char> input;  char tmp = ' ';  cout << "请输入SNL程序片段：" << endl;  while (tmp != '.') {  tmp = getchar();  input.push\_back(char(tmp));    }  //打印看一下  int len = input.size();  //for (int i = 0; i < len; i++) {  // cout << input[i];  //}  generatetoken(input, len); //将字符串变成token序列  //system("cls");  printtoken(token); //将token进行输出  //Sleep(5000);  //语法分析器  Treenode\* root = program();  return 0;    } | | |