|  |
| --- |
| 编译原理课程设计  实验报告  **指导教师： 刘磊**  **年 级： 2018级**  **班 级： 27班**  **小组编号： 第9组**  **组长学号姓名：21180123-李伟涵**  **组员学号姓名：21180813-郑志硕**  **2021年 4月 25 日**  **计算机科学与技术学院** |

|  |  |  |
| --- | --- | --- |
| **完成实验内容** | | |
| 设计并实现SNL程序设计语言的编译程序：  1.词法分析模块  2.语法分析模块（递归下降方法）  3.语法分析模块（LL(1)方法）  4.语义分析模块 | | |
| **小组成员任务完成情况** | | |
| 姓名 | 具体完成任务 | 工作量  百分比 |
| **李伟涵** | 词法分析-语法分析（递归下降方法）-语义分析 | 70% |
| **郑志硕** | 语法分析（LL(1)方法） | 30% |
| **小组成员协作情况** | | |
| **李伟涵：**词法分析-语法分析（递归下降方法）-语义分析  **郑志硕：**语法分析（LL(1)方法）  成员之间代码可以根据导入和输出的文件相联系 | | |
| **实验平台与编程语言** | | |
| 实验平台：Visual Studio2019/CodeBlocks  编程语言：C++ | | |
| **实验方案设计** | | |
| 词法分析：从左向右将字符逐个读入，依据词法规则识别单词。执行词法分析的程序称为词法分析器，将给定的程序通过词法分析器，识别出对应的单词符号，根据其读入内容，判断单词为标识符、整型数、单字符分界符、双字符分界符等类别，对于其他的输入符号，则根据不同错误类型报错。结果有输出到控制台或者输出到文件两种形式。  语法分析：在词法分析之后，根据词法分析的结果和定义的语法规则判断输入的程序是否有语法错误。  LL(1)语法分析：语法分析按自左至右的顺序扫描输入符号串，并在此过程中产生一个句子的最左推导，在分析过程中，每进行一步推导，只要向前查看一个输入符号，便能确定当前所应选用的产生式规则。根据SNL语言的上下文无关文法求出其first，follow，predict集合，读取不同语言的上下文无关文法文件，可以实现求出不同语言的三个集合，完成对不同语言的分析。如果有语法错误，显示错误行数与内容。 getFirst  1. 所有终结符的First集为其本身，对于所有的非终结符，如果还未生成first集则执行第二步 2. 对于非终结符α，First（α)=∅，对其所有产生式，如α=x1x2...xn，如果x1~xn是终结符或者已经生成first集，则执行 第三步，否则对于其中所有未生成first集的非终结符执行第二步   3.  C:\Users\liweihan\Desktop\捕获.PNG  getFollow  实现方法：   1. 初始化，对于开始符S（即Program），Follow[S]={#}，其它非终结符Follow集为空集 2. 对于所有的产生式，例如α=x1x2...xn，执行第三步 3. 对于x1~xn,依次对其中的非终结符xi执行第四步 4. 如果xi等于xn，则   C:\Users\liweihan\Desktop\捕获.PNG  5. 对于xj(i<j<=n)  C:\Users\liweihan\Desktop\捕获.PNG  如果xj(i<j<=n)是非终结符，且xj的First集不包含ε，则并直接返回第三步  6.当处理完一遍所有的产生式后，如果有Follow集产生了变化，则重新开始执行第二步。 getPredict  1. 对于所有的产生式，如第k个产生式α=x1x2...xn，执行第二步 2. 如果x1~xn的First集均包含ε,则   C:\Users\liweihan\Desktop\捕获.PNG  3.如果x1~xi-1的First集均包含ε，xi的First集不包含ε,则  C:\Users\liweihan\Desktop\捕获.PNG  LL（1）   1. 将'#'和开始符S(Program)依次入栈，按照Predict集生成LL(1)分析表   C:\Users\liweihan\Desktop\捕获.PNG  2.如果栈为空，或者输入流处理完毕执行第三步。否则从栈中弹出一个元素，  如果是终结符，判断是否匹配当前输入流字符，不匹配则进行错误处理，匹配则后移输入流，继续执行第二步。 如果是非终极符，判断当前输入流字符是否在该非终极符的LL(1)分析表中，如果在则将对应的产生式如α=x1x2...xn，将x1x2...xn，倒着依次入栈，如果不在则进行错误处理  3.如果栈非空或者输入流未处理完，语法存在错误，否则语法分析成功。  递归下降语法分析：读入词法分析的tokenlist，并构建语法树的根节点。在读入tokenlist过程中，若进行需要对终极符进行匹配，就进行opposition++，匹配当前终极符并读入下一个token；否则调用对应的函数，继续对当前token进行对应的处理。在递归下降过程中，会对所有的终极符（符号，变量，函数名等）均进行创建语法树节点的操作。若有错误，程序将会报错并无法向下进行语义分析。最终，程序将会在控制台输出得到的语法树。  语义分析：根据上一步得到的语法树，通过读入root，进行语义的分析。程序将会对声明部分创建符号表并入栈，在函数体部分对所有的变量进行向低层的查找、若变量、函数未声明将会报错，在当前层结束并删除时会输出这一层的符号表并删除当前层符号表 | | |
| **程序界面及运行截图** | | |
| **词法分析：**  **例子文件：**  **C:\Users\liweihan\Desktop\捕获.PNG**  **得到的token文件：**  **C:\Users\liweihan\Desktop\捕获.PNG**  **若输入词法错误文件：**  **C:\Users\liweihan\Desktop\捕获.PNG**  **则会在文件中显示错误:**  **C:\Users\liweihan\Desktop\捕获.PNG**  **语法分析(LL1)：**  导入正确文件（将词法分析和LL1语法分析结果存入文件）：  QQ截图20210426112424  QQ截图20210426112616  QQ截图20210426112703  导入语法错误文件（将词法分析和LL1语法分析结果存入文件）：  语法错误会把错误行数与错误信息输出到控制台与文件之中。  QQ截图20210426112758  QQ截图20210426112830  导入词法错误文件（将词法分析和LL1语法分析结果存入文件）：  词法错误会在控制台输出报错并把正确结果输出到文件，但是不能往下继续进行接下来的语法分析。  QQ截图20210426112905  导入正确文件（将词法分析结果输出到控制台和LL1语法分析结果存入文件）：  QQ截图20210426113218  导入词法错误文件（将词法分析结果输出到控制台和LL1语法分析结果存入文件）：  QQ截图20210426113241  导入语法错误文件（将词法分析结果输出到控制台和LL1语法分析结果存入文件）：  QQ截图20210426113306  **语法分析(递归下降):**  递归下降程序将根据词法结果（tokenlist）构建语法树：  例子程序：  其中包含数组，嵌套定义函数，while，if等  C:\Users\liweihan\Desktop\捕获.PNG  以下为构建语法树的结果:  C:\Users\liweihan\Desktop\捕获.PNG  若出现语法错误(v1:=1之后缺失；)：  C:\Users\liweihan\Desktop\捕获.PNG  则在构建语法树时会返回null并报错，不会再显示语法树  C:\Users\liweihan\Desktop\捕获.PNG  **语义分析:**  语义分析通过读取语法树根节点root来进行，输出符号表:  实例程序同上，不再粘贴  符号表：  C:\Users\liweihan\Desktop\捕获.PNG  实例程序主函数中k并未给出声明，因此会显示出现语义错误  图中分别显示了嵌套函数（第一层）和主函数（第零层）的符号表  主函数中可见嵌套函数q的声明，层数为1 | | |
| **源程序核心代码** | | |
| **词法分析：**  void Scanner::WordAnalyse(FILE\* fpin) {  int line = 1;//确定当前行数  int index = 0;//tokenlist数组的下标  char ch = fgetc(fpin);  while (ch != EOF) {  if (IsFilter(ch)) {//判断是否是空格等字符 这些字符不需要放入tokenlist中  if (ch == '\n') {  line++;  }  ch = fgetc(fpin);  }  else if (IsLetter(ch)) {//判断是否是标识符或关键字  string name = "";  name += ch;  ch = fgetc(fpin);  while (IsLetter(ch) || IsDigit(ch)) {//表示符第一个为字母，之后为字母或数字  name += ch;  ch = fgetc(fpin);  }  //ch不再是字母或数字，此时ch已经完成了取下一个字符的操作，不用再来一次ch = fgetc(fpin)  //再去判断这个标识符是否是关键字  int judge = IfisreservedWords(name);//若为关键字，judge保存对应关键字在reservedWords数组中的编号  if (judge>=0) {//若为关键字  tokenlist[index].lineshow = line;  tokenlist[index].word = reservedWords[judge];  index++;  }  else {//若为普通的标识符  tokenlist[index].lineshow = line;  tokenlist[index].word.str = name;  tokenlist[index].word.tok = ID;  index++;  }  }  else if (IsSeparater(ch)) {//判断是否为单分界符  LexType type = GetSeparaterType(ch);  tokenlist[index].word.str = ch;  tokenlist[index].word.tok = type;  tokenlist[index].lineshow = line;  index++;  ch = fgetc(fpin);  }  else if (IsDoubleSeparater(ch)) {//判断是否为双分界符  string arr = "";  if (ch == '.') {//判断数组下标限界符  arr += ch;  ch = fgetc(fpin);  if (ch == '.') {//是数组下标限界符  arr += ch;  tokenlist[index].lineshow = line;  tokenlist[index].word.str = arr;  tokenlist[index].word.tok = UNDERANGE;  ch = fgetc(fpin);  index++;  }  else {//只是一个点  tokenlist[index].lineshow = line;  tokenlist[index].word.str = '.';  tokenlist[index].word.tok = DOT;  index++;  }  }  else if (ch == ':') {//判断是否为赋值号  //string test = "";  //test += ch;  ch = fgetc(fpin);  tokenlist[index].lineshow = line;  if (ch == '=') {  tokenlist[index].word.str = ":=";  tokenlist[index].word.tok = ASSIGN;  ch = fgetc(fpin);  }  else {//发生错误  //test += ch;  tokenlist[index].word.str = ":";  tokenlist[index].word.tok = ERROR;  //若发生错误，不应该取下一个字符，说不定只有一个:，这个ch取到的是一串字符的开头  }  index++;  }  else if (ch == '\'') {//判断字符串 这里认为'也是双分界符（需要两个且中间的是charc）  tokenlist[index].lineshow = line;  tokenlist[index].word.tok = CHARC;  string charc = "";  ch = fgetc(fpin);  while (ch != '\'') {  charc += ch;  ch = fgetc(fpin);  }  tokenlist[index].word.str = charc;  index++;  ch = fgetc(fpin);  }  }  else if (IsDigit(ch)) {//判断是否为无符号整数  string num = "";  num += ch;  ch = fgetc(fpin);  while (IsDigit(ch) || IsLetter(ch)) {//为保证后面的词法分析正确，即使出现如：“123ret2”这种，我们也应该把它全部放入num中，再给它一个ERROR，而不应该将123与ret2拆开  num += ch;  ch = fgetc(fpin);  }  //再去检查num中是否全是数字  bool flag = true;  for (int i = 0; i < num.length(); i++) {  if (!IsDigit(num[i])) {//若出现第i位不是数字  flag = false;  break;  }  }  if (flag) {  tokenlist[index].word.tok = INTC;  }  else {  tokenlist[index].word.tok = ERROR;  }  tokenlist[index].lineshow = line;  tokenlist[index].word.str = num;  index++;  //同样，由于while循环，不需要再进行一次ch = fgetc(fpin);  }  else if (ch == '{') {//去注释  while (ch != '}') {  ch = fgetc(fpin);  if (ch == '\n')  line += 1;  }  ch = fgetc(fpin);  }  else {//未知字符  tokenlist[index].lineshow = line;  tokenlist[index].word.str = ch;  tokenlist[index].word.tok = ERROR;  index++;  ch = fgetc(fpin);  }  }  //最后增加文件结尾  tokenlist[index].lineshow = line;  tokenlist[index].word.str = ch;  tokenlist[index].word.tok = ENDFILE;  }  **语法分析（LL1）：**  int main()  {  lex\_analysis\* l\_a = new lex\_analysis();  string str\_FileName="111.txt";  l\_a->Scan(str\_FileName);  l\_a->ResultPrint(1);  //l\_a->ResultPrint(0);  cout<<"结束词法分析"<<endl;  if(l\_a->iswrong())  return 0;  cout<<"词法分析结果成功记录在lex\_result.txt中"<<endl;  wenfazhuanhuan();  getfirst();  getfollow();  getpredict();  ci\_yu(l\_a);  delete l\_a;  cout<<"结束语法分析"<<endl;  cout<<"语法分析结果记录在par\_result.txt中"<<endl;  if(LL1())  cout<<"success! "<<endl;  else  cout<<"fail！ "<<endl;  return 0;  }  int LL1()  {  ofstream output("answer/par\_result.txt",ios::trunc);  int flag=1;  memset(table, -1, sizeof(table));  rep(i, 0, tot)  {  int l = fu[wenfa[i][0]];  rep(j, 0, Predict[i].size())  {  int r = Predict[i][j];  table[l][r]=i;  }  }  int i=0;  sta.push(fu["#"]);  sta.push(fu[wenfa[0][0]]);  while(!sta.empty())  {  int now = sta.top();  sta.pop();  if(now==s[i])  i++;  else if(table[now][s[i]]!=-1)  {  int k = table[now][s[i]];  output<<wenfa[k][0]<<"->";  rep(j, 1, wenfa[k].size())  output<<wenfa[k][j]<<" ";  output<<endl;  per(j, 1, wenfa[k].size())  {  if(wenfa[k][j]!="KONG")  sta.push(fu[wenfa[k][j]]);  }  }  else  {  if(i==len\_s-1)  {  cout<<"第"<<yang[i-1].i\_line<<"行 "<<"语法错误";  output<<"第"<<yang[i-1].i\_line<<"行 "<<"语法错误";  }  else  {  cout<<"第"<<yang[i].i\_line<<"行 "<<"语法错误: "<<yang[i].str\_sem<<endl;  output<<"第"<<yang[i].i\_line<<"行 "<<"语法错误: "<<yang[i].str\_sem<<endl;  }  while(table[now][s[i]]==-1&&now!=s[i]&&i<len\_s)  i++;  flag=0;  }  if(i>=len\_s)  break;  }  output.close();  if(i!=len\_s||flag==0)  return 0;  return 1;  }  void lex\_analysis::ResultPrint(int type)//默认参数0表示输出到控制台，1表示输出到文本  {  Token \* t\_temp=t\_first;  if(type == 0){  while(t\_temp!=t\_last)  {  cout<<t\_temp->i\_line<<'\t'<<t\_temp->str\_type<<'\t'<<t\_temp->str\_sem<<endl;  t\_temp=t\_temp->t\_next;  }  cout<<t\_temp->i\_line<<'\t'<<t\_temp->str\_type<<'\t'<<t\_temp->str\_sem<<endl;  }  else if(type == 1){  ofstream output("answer/lex\_result.txt",ios::trunc);  while(t\_temp!=t\_last)  {  output<<t\_temp->i\_line<<'\t'<<t\_temp->str\_type<<'\t'<<t\_temp->str\_sem<<endl;  t\_temp=t\_temp->t\_next;  }  output<<t\_temp->i\_line<<'\t'<<t\_temp->str\_type<<'\t'<<t\_temp->str\_sem<<endl;  output.close();  }  }  #include<bits/stdc++.h>  using namespace std;  struct Token{  int i\_line; //行号  string str\_type; //类型  string str\_sem; //单词语义  Token\* t\_next; //链表next指针  };  class lex\_analysis{  public:  lex\_analysis();  ~lex\_analysis();  int Scan(string file);  void AddNewToken(int row,string str1,string str2);  void ResultPrint(int type=0); //输出结果，  void result\_hjz(vector<Token> &x);  Token\* GetToken();  int iswrong();  private:  int i\_Line; //当前行数  int i\_IsWrong; //判断是否出错  int i\_SmallBr; // ()  int i\_MidBr; // []  int i\_BigBr; // {}  int i\_SgQu; // ''  Token \* t\_first;  Token \* t\_last;  };  **语法分析（递归下降）:**  函数过多，只展示Program()，和语法树输出函数  TreeNode\* Parse::Program() {  TreeNode \*programhead, \*declarepart, \*programbody;  programhead = ProgramHead();  declarepart = DeclarePart();  programbody = ProgramBody();  TreeNode \*t = new TreeNode;  t->nodekind = ProK;//根节点设为根标志节点  //应判断新语法树root是否创建成功，此处未进行判断  if (programhead == NULL) {//若programhead为空，应进行报错  cout << "ProgramHead为空" << "\n";  return NULL;  }  else {  t->child.push\_back(programhead);  }  if (declarepart == NULL) {//若declarepart为空，应进行报错  cout << "DeclarePart为空" << "\n";  return NULL;  }  else {  t->child.push\_back(declarepart);  }  if (programbody == NULL) {//若programbody为空，应进行报错  cout << "programbody为空" << "\n";  return NULL;  }  else {  t->child.push\_back(programbody);  }  //match(DOT);//当前单词和DOT进行匹配  opposition++;  return t;  }  oid Parse::PrintParseTree() { //导一个文件指针进来，应该可以通过递归实现  cout << "执行输出语法树" << "\n";  TreeNode \*nowop = root;  if (nowop == NULL) {  cout << "空了??" << "\n";  }  int depth = 0;//用于记录当前的语法树节点的深度,用于控制输出空格数量  DFS(nowop, 0);  }  void Parse::DFS(TreeNode \*t, int depth) {  while (t != NULL) {  for (int i = 0; i < depth; i++) {  cout << " ";  }  cout << transform(t->nodekind);  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //以下是对各个节点的具体输出处理  if (t->nodekind == PheadK || t->nodekind == ProcDecK) {  cout << " " << t->name[0] << "\n";  }  else if (t->nodekind == DecK) {  cout << " " << transform\_deckind\_to\_string(t->k.dec);  for (int j = 0; j < t->name.size(); j++) {  cout << " " << t->name[j];  }  cout << "\n";  }  else if (t->nodekind == StmtK) {  //cout << " " << t->op << "\n";  if (t->k.stmt == IfK) {  cout << " " << "If" << "\n";  //if的条件表达式、条件为真的处理语句和条件为假的处理语句都放在if节点的child中  //if中存在问题，如果每一种情况中都有多条语句，输出会混乱  }  else if (t->k.stmt == WhileK) {  cout << " " << "While" << "\n";  //while的条件表达式和循环语句部分均放在while节点的child中  }  else if (t->k.stmt == AssignK) {  cout << " " << "Assign" << "\n";  /\*  //assign的赋值号左侧的变量在name中，因此要先输出一下下一层的等号左侧  for (int i = 0; i <= depth; i++) {  cout << " ";  }  cout << "ExpK" << " " << t->name[0] << " IdV" << "\n";  //这里面好像并没有a[3]:=?这种操作 具体看函数stm和asscall 没有匹配[的操作  \*/  //现在等号左侧也被放到child中  }  else if (t->k.stmt == ReadK) {  cout << " " << "Read" << " " << t->name[0] << "\n";  }  else if (t->k.stmt == WriteK) {  cout << " Write" << "\n";  //write语句的表达式在child中，本层不做处理  }  else if (t->k.stmt == CallK) {  cout << " Call" << "\n";  /\*  //函数名被放在了call节点的name中，因此应先输出下一层的函数名  for (int i = 0; i <= depth; i++) {  cout << " ";  }  cout << "ExpK " << t->name[0] << " IdV" << "\n";  \*/  //现在改为函数名也放到child中  }  else if (t->k.stmt == ReturnK) {  cout << " Return" << "\n";  //? 我也不知道咋整啊，应该是这样  }  }  else if (t->nodekind == ExpK) {  if (t->k.EXP == OpK) {  cout << " OP " << tranform\_op\_to\_string(t->op) << "\n";  //不论是exp，simple\_exp，term还是factor，运算符两边的数或变量均放在child中  }  else if (t->k.EXP == ConstK) {  cout << " Const " << t->num << "\n";  }  else if (t->k.EXP == IdEK) {  if (t->vark == ArrayMembV) {  cout << " " << t->name[0] << "[] IdV\n";  }  else {  cout << " " << t->name[0] << " IdV\n";  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*存在问题！  //并没有考虑a[i+1]和a.field的情况，先这么放着  }  }  else {  cout << "\n";  }  //其他语法树节点后不会跟其他东西，只有语法树节点类型  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  for (int j = 0; j < t->child.size(); j++) {//对每一个t的孩子均进行遍历  DFS(t->child[j], depth + 1);  }  t = t->sibling;//去遍历t的兄弟节点  }  }  **语义分析:**  程序过长，只展示初始函数  void Semantic::Analyze(TreeNode \*currentP) {//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  SymbTable \* Entry = NULL;  TreeNode \* p = NULL;  TreeNode \* pp = currentP;  CreateTable();  initialize();  //循环处理主程序的声明部分  //直接在声明节点开始  //第一个节点是ProgramHead，没有用  p = currentP->child[1];//处理DeclarePart  while (p != NULL) {  if (p->nodekind == TypeK) {  TypeDecPart(p->child[0]);  }  else if (p->nodekind == VarK) {  VarDecList(p->child[0]);  }  else if (p->nodekind == ProcDecK) {  ProcDecPart(p);  }  else {  cout << "语法树中没有这种类型的节点" << endl;  }  p = p->sibling;  }  //if(Error) return;  //实例程序中，若出现错误，直接返回  //处理主程序体  p = currentP->child[2];//处理ProgramBody  if (p->nodekind == StmLK) {  Body(p);  }  PrintSymbTable();  //撤销符号表  if (level != -1)  DestroyTable();  } | | |