|  |
| --- |
| 编译原理  实验报告  （第四周）  **指导教师：刘宇舟**  **年 级： 大三**  **班 级：八班**  **小组编号：8班6组**  **组长学号姓名：55190814 曹伟**  **组员学号姓名：55190828 张天浩**  **组员学号姓名：55190815 张津赫**  **2022年 6 月 8 日**  **软件学院** |

|  |  |  |  |
| --- | --- | --- | --- |
| **实验目的** | | | |
| 1、通过设计编制完成对SNL语言的词法分析程序和语法分析程序，加深对词法分析和语法分析原理的理解。理解其功能和实现方法。  2、通过本课程实验，加深对编译原理课程所学理论知识的理解，熟练掌握和巩固编译原理的基本知识，深刻体会编译原理的本质和思想。将所学理论知识应用于实践，提高学生的动手能力。 | | | |
| **实验内容** | | | |
| 总体内容：设计并实现SNL程序设计语言的编译程序。  具体内容：   1. 根据SNL语言特点，设计TOKEN序列结构。根据标识符/保留字种类划分，完成词法分析，实现对SNL的词法分析程序，能够识别出关键字，标识符，常数，运算符和分界符等常用语言单词符号 2. 根据实验给出的SNL 104条文法规则，以及实验指导书，完成LL（1）分析表的建立，用于实现LL（1）语法分析。LL（1）语法分析输入TOKEN序列，输出由终结符和非终结符组成的语法树（层次输出），能够识别出声明语句、分支语句、循环语句等多种常用语句 3. 根据实验给出的SNL语言的104条语法规则，递归下降法根据predict集选择合适的子程序进行调用，对每个非终极符按其产生式结构构造对应语法分析子程序，并产生过程调用命令，对每个终极符产生匹配命令。 | | | |
| **小组成员任务占比及组内评分** | | | |
| 学号\_姓名 | 占比 | 评分 | 评分理由 |
| 曹伟 | 34 | 100 | 按要求实现了LL1语法分析（104条文法的LL1分析表，104条文法的录入）工作量较大，完成了程序的合并与调试。 |
| 张津赫 | 33 | 100 | 完成词法分析工作，设计的TOKEN结构完备详尽，参与整体程序的结构设计工作，为后序语法分析提供了极大的便利。 |
| 张天浩 | 33 | 100 | 按要求实现了递归下降语法分析程序，在工作量很大的情况下（104个文法函数）。依然高效准确的完成了分析程序。 |
| **程序设计思想** | | | |
| 我们组完成了**词法分析程序**和**LL1，递归下降两种语法分析程序。**  descript  对于一个SNL源程序：  **词法分析：**  词法分析就是对输入的字符串形式的源程序按顺序进行扫描，根据源程序的词法规则识别具有独立意义的单词（符号），并输出与其等价的 Token 序列。  下面是能识别SNL中各类单词的DFA，也是词法分析的核心思路。  descript  **语法分析：**  **递归下降方法**：每个非终极符按其Predict集选择合适的产生式，构造相应的语法分析子程序，其中对应的终极符进行match匹配，对应的非终极符调用对应的函数，直到所有的非终极符均转换为终极符并match成功或者match出错程序结束。  **LL（1）方法：**根据文法规则，求出Predict集，填写LL（1）分析表。编写驱动程序完成语法分析和语法树的构建（边分析边构建）。在分析结束后若成功则打印语法树。失败则输出在何处失败，何种原因导致失败。  其中，LL1语法分析主要就是对LL1分析表的构造。语法分析表用一个二维矩阵表示。LL1语法分析的主要过程就是先初始化符号栈，然后调用创建LL1符号表函数CreatLL1Table()，生成语法树的根节点。然后利用LL1分析表和符号栈进行语法分析，并处理标识符不匹配的错误和文件提前结束错误，函数处理完毕后，整个语法分析结束，函数返回生成的语法分析树。  主要流程图如下：  descript | | | |
| **实验步骤** | | | |
| 第一周：设计TOKEN数据结构。查阅资料课件，了解具体实现编译程序的方法。搞清词法分析文法分析步骤。  第二周：  词法分析：学习SNL文法，理解各种符号和保留字的含义。  语法分析：  1.递归下降：对语法树进行了解，完成token序列的读入和read\_next函数（读入下一个单词）和部分语法树的工作。  2.LL1分析：设计文法结构，录入文法，将LL1表和终极符非终极符数组进行映射，便于后续语法分析。  第三周：  词法分析：编写token类和词法分析器类。定义函数内容，实现了getToken函数的部分功能。  语法分析：  1.递归下降：根据predict集完成大部分调用子函数。  2.LL1分析：编写驱动程序（主要通过语法栈和语法树栈进行分析和建树），处理TOKEN序列，完成LL1语法分析。  第四周：  词法分析：完成了剩余的函数功能，对程序进行调试，测试结果。  语法分析：  1.递归下降：完成剩余的函数并对语法树进行测试。  2.LL1分析：进行测试，编写语法树输出程序。  进行程序三个部分的汇总和调试。 | | | |
| **数据结构及算法设计** | | | |
| **数据结构设计：**  **TOKEN种类：**  typedef enum {  /\*薄记单词符号\*/  ENDFILE, ERROR,  /\*保留字\*/  PROGRAM, PROCEDURE, TYPE, VAR, IF,  THEN, ELSE, FI, WHILE, DO,  ENDWH, BEGINl, ENDl, READ, WRITE,  ARRAY, OF, RECORD, RETURNl,  //类型  INTEGER, CHARl,  /\*多字符单词符号\*/  ID, INTC, CHARC, //字母开头的包含字母数字的字符串为ID，整数为INTC，单引号内的字母数字为CHARC  /\*特殊符号\*/  ASSIGN, EQ, LT, PLUS, MINUS, // := = < + -  TIMES, OVER, LPAREN, RPAREN, DOT, // \* / ( ) .  COLON, SEMI, COMMA, LMIDPAREN, RMIDPAREN, //: ; , [ ]  UNDERANGE // ..  } TokenType;  **TOKEN结构：**  class Token {  public:  TokenType type;  string context;  int line; //所在代码行数  };  **递归下降语法树节点结构：**  typedef struct treeNode  {  struct treeNode \* child[3]; //子节点指针  struct treeNode \* sibling; //兄弟节点指针  int lineno; //源代码行号  NodeKind nodekind; //节点类型  //ProK,PheadK,TypeK,VarK,ProcDecK,StmLK,DecK,StmtK,ExpK,ProcK  int idnum; //标识符个数  string name[20];//标识符名  union  {  DecKind dec;  //ArrayK,CharK,IntegerK,RecordK,IdK  StmtKind stmt;  //IfK,WhileK,AssignK,ReadK,WriteK,CallK,ReturnK  ExpKind exp;  //OpK,ConstK,VariK  } kind;//具体类型    struct  {  struct  {  int low; //数组下界  int up; // 数组上界  DecKind childtype; // 数组的子类型  }ArrayAttr; //数组属性    struct  {  ParamType paramt; //参数类型  //valparamType,varparamType  }ProcAttr; //过程属性    struct  {  TokenType op; //表达式的操作符  int val; //表达式的值  VarKind varkind; // 变量的类别  //IdV,ArrayMembV,FieldMembV  ExpType type; // 用于类型检查  //Void,Integer,Boolean  }ExpAttr; //表达式属性  char type\_name[10]; //类型名是标识符    } attr;  }TreeNode;  **LL1文法结构：**  static class Rule { // SNL的语法规则  public:  string A;//左  vector<string> B;//右  };  **终极符非终极符数组：**  vector<string> terminal; // 终极符  vector<string> nonTerminal; // 非终极符  LL1分析表：  一个二维数组，逻辑上行号和终极符数组下标对应。列号与非终极符下标对应。LL1表中存放的内容为相应文法下标（文法在文法数组中下标）。在逻辑上完美实现LL1分析表的构建。  **LL1语法树节点结构：**  static struct node  {  string name;//节点名称  string nodekind;//节点类型  vector<node\*>child;//子节点列表  };  typedef node\* TreeNode;  static struct AST  {  TreeNode root;  };  **算法设计：**  **词法分析DFA：**  Token Analyzer::getNextToken() {  state = START;  while (1) {  if (next < 0) {  next = 0;  }  if (next + 1 > source.length()) {  return Token(ENDFILE, "", line);  }  if (state == START) {  while (source.at(next) == ' ' || source.at(next) == '\n' || source.at(next) == '\t') {  if (source.at(next) == '\n') {  line++;  }  next++;  if (next + 1 > source.length()) {  return Token(ENDFILE, "", line);  }  }  if (isLetter(source.at(next))) {  state = INID;  }else if (isNumber(source.at(next))) {  state = INNUM;  }  else {  switch (source.at(next++)) {  //单个符号：完成状态，此处代码简化  case'+':  return Token(PLUS, "", line);  break;  case'-':  return Token(MINUS, "", line);  break;  case'\*':  return Token(TIMES, "", line);  break;  case'/':  return Token(OVER, "", line);  break;  case'(':  return Token(LPAREN, "", line);  break;  case')':  return Token(RPAREN, "", line);  break;  case';':  return Token(SEMI, "", line);  break;  case'[':  return Token(LMIDPAREN, "", line);  break;  case']':  return Token(RMIDPAREN, "", line);  break;  case'=':  return Token(EQ, "", line);  break;  case'<':  return Token(LT, "", line);  break;  case',':  return Token(COMMA, "", line);  break;  case':':  state = INASSIGN;  break;  case'{':  state = INCOMMENT;  break;  case'.':  state = INRANGE;  break;  case'\'':  state = INCHAR;  break;  default:  string t(1, source.at(next - 1));  return Token(ERROR, t, line);  }  }  }  else if (state == INASSIGN) {  if (source.at(next) == '=') {  next++;  return Token(ASSIGN, "", line);  }  else {  return Token(COLON, "", line);  }  }  else if (state == INCOMMENT) {  while (source.at(next) != '}' && next < source.length()) {  if (source.at(next) == '\n')  line++;  next++;  }  next++;  state = START;  }  else if (state == INNUM) {  string number = "";  number += source.at(next++);  while (next < source.length() && isNumber(source.at(next))) {  number += source.at(next++);  }  return Token(INTC, number, line);  }  else if (state == INID) {  string id = "";  id += source.at(next++);  while (next < source.length() && (isNumber(source.at(next)) || isLetter(source.at(next)))) {  id += source.at(next++);  }  TokenType type = reservedLookup(id); //若不是，返回ID  if (type == ID) {  return Token(ID, id, line);  }  else {  return Token(type, "", line);  }  }  else if (state == INCHAR) {  string chars = "";  while (next < source.length() && (isNumber(source.at(next)) || isLetter(source.at(next)))) {  chars += source.at(next++);  }  if (next < source.length() && source.at(next++) == '\''){  return Token(CHARC, chars, line);  }else{  return Token(ERROR, "char error", line);  }  }  else if (state == INRANGE) {  if (source.at(next) == '.') {  next++;  return Token(UNDERANGE, "", line);  }  else {  return Token(DOT, "", line);  }  }  }  }  **LL1语法分析：**  void initialize() // 初始化LL1分析所要用到的基本数据结构，包含文法规则，终极符非终极符数组、LL1分析表。  static int findindex(string str, vector<string> str\_base)//完成从具体的token到下标的转换。  int parsing(vector<string> token)//输入为一个变形过的TOKEN序列（只用终极符和非终极符替换每个TOKEN），建立语法栈和语法树栈，完成具体的语法分析过程。建立语法树。  descript  void parser(vector<string> token)  {  //读取token  basedata.initialize();  cout << "分析结果：";  switch (parsing(token))  {  case 0:  cout << "语法正确" << endl;  cout << endl << "语法树如下" << endl;  printTree();  break;  case 1:  cout << "无对应产生式" << endl;  break;  case 2:  cout << "错误原因：含有非法字符" << endl;  break;  case 3:  cout << "语法错误" << endl;  break;  default:  cout << "error" << endl;  }  }  //语法分析结果处理程序，对于产生的错误进行输出，如果成功完成语法分析则打印语法树。  void printTree()//层次遍历打印语法树  {  ofstream outfile;  outfile.open("C:/Users/17363/Desktop/lexical analysis/res\_ll1.txt", ofstream::app);    queue<TreeNode> que;//使用队列完成层次遍历  int dept = 0;  int num0 = 1;  int num1 = 0;  que.push(ast.root);  cout << ast.root->name <<"("<<ast.root->child.size() << "个)" << endl;  outfile << ast.root->name << "(" << ast.root->child.size() << "个)" << endl;  while (!que.empty())  {  TreeNode cur\_node = que.front();  num0--;  for (int i = cur\_node->child.size() - 1; i >= 0; i--)  {  cout << cur\_node->child[i]->name << "(" << cur\_node->child[i]->child.size() << "个子节点)" << " ";  outfile << cur\_node->child[i]->name << "(" << cur\_node->child[i]->child.size() << "个子节点)" << " ";  que.push(cur\_node->child[i]);  }  num1 = num1 + cur\_node->child.size();  if(cur\_node->child.size()!=0)  cout << "||| ";  //cout<< cur\_node->child.size()<<" ";  if (num0 == 0)  {  if (num1 != 0)  {  cout << "::第" << dept + 1 << "层" << endl;  outfile << "::第" << dept + 1 << "层" << endl;  }  dept++;  num0 = num1;  num1 = 0;    }        que.pop();    }  outfile.close();  }  **递归下降法语法分析：**  parse()：读入单词，调用program()程序。  TreeNode \*parse()  {  TreeNode \* tree=NULL;  read\_next();  tree = program();  return tree;  }  getTreeNode（）：进行语法树结点的初始化  TreeNode \*getTreeNode(NodeKind n)  {  TreeNode \*t=new TreeNode();  t->child[0]=NULL;  t->child[1] = NULL;  t->child[2] = NULL;  t->idnum=0;  t->sibling=NULL;  t->nodekind=n;  t->lineno=nexttoken.line;  return t;  }  read\_next()：单词使用vector数组存储，用i标识处理到哪一个单词  void read\_next()  {  if (number<v.size()) {  nexttoken = v[number];  number++;  }  }  match（）：匹配函数  void match(TokenType type)  {  if(nexttoken.type==type)  read\_next();  else  {  cout<<"第"<<nexttoken.line<<"行匹配出错!";  exit(0);  }  }  program（）：调用处理程序头、声明部分、程序体的函数  TreeNode \* program()  {  TreeNode \*tree=getTreeNode(ProK);//建立根结点  tree->child[0]=programHead();  tree->child[1]=declarePart();  tree->child[2]=programBody();  treePrint(tree);  match(DOT);  return tree;  }  programHead（）：处理程序头，并match标识符类型的函数名  TreeNode \* programHead(void)  {  TreeNode \*t=getTreeNode(PheadK);  match(PROGRAM);  if(nexttoken.type==ID)  t->name[t->idnum++]=nexttoken.context;//函数名  match(ID);  return t;  }  declarePart（）：根据不同的声明调用对应的函数  TreeNode \* declarePart(void)  {  if(nexttoken.type==TYPE)  return typeDec();  else if(nexttoken.type==VAR)  return varDec();  else if(nexttoken.type==PROCEDURE)  return procDec();  else return NULL;  }  programBody（）：//处理程序体语句序列  TreeNode \*programBody()  {  TreeNode \*t=getTreeNode(StmLK);  match(BEGINl);  if(t!=NULL)  t->child[0]=stmList();  match(ENDl);  return t;  }  stm（）：处理语句结构，调用循环、分支等函数。  TreeNode \* stm(void){  switch(nexttoken.type){  case IF:return conditionalStm();  case WHILE:return loopStm();  case READ:return inputStm();  case WRITE:return outputStm();  case RETURNl:return returnStm();  case ID:temp\_char=nexttoken.context;match(ID);return assCall();  default:  cout << "第" << nexttoken.line << "行没有匹配成功";  exit(0);  }  }  treePrint():打印语法树主程序  void treePrint(TreeNode \*t){  cout << "第" << t->lineno << "行 ProK" << endl;  headPrint(t->child[0]);  typePrint(t->child[1],1);  varPrint(t->child[1],1);  procPrint(t->child[1],1);  BodyPrint(t->child[2],1);  }  varIdList（）：处理标识符存入name数组，并记录个数  void varIdList(TreeNode \* t)  {  if (nexttoken.type==ID)  {  t->name[(t->idnum)] =nexttoken.context;  match(ID);  t->idnum =(t->idnum ) + 1;  }  else  {  cout<<"需要id!"<<endl;  read\_next();  }  varIdMore(t);  }  typeDecList、typeDecMore：处理类型声明和标识符  TreeNode \* typeDecList(void){  TreeNode \*t=getTreeNode(DecK);  typeId(t);match(EQ);typeDef(t);match(SEMI);  t->sibling=typeDecMore();  return t;  }  TreeNode \* typeDecMore(void){  if(nexttoken.type==ID)  return typeDecList();  else  return NULL;  }  **其余打印函数和识别关键字的函数未列出。** | | | |
| **实验结果截图** | | | |
| 一个简单SNL程序运行结果：  descript  输入SNL源程序：（冒泡排序）  descript  词法分析结果：TOKEN序列  descript  递归下降语法树：  descript  LL1层次遍历语法树。  descript  出错情况：  descript | | | |
| **实验结论** | | | |
| 本实验在学习完理论知识后马上进行实践，帮助我们更好地理解编译的过程，使我们深入了解了编译过程中的词法分析和语法分析两部分。我们小组较好的完成了必做内容（语法词法分析，语法树的输出），并且在语法分析部分分别使用LL（1）法和递归下降法进行语法树的构建。美中不足的是，由于时间原因，没有完成语义分析。第二LL1语法树采用命令行层次输出的方式，整体程序没做用户界面而是采用命令行形式，直观性美观性有待加强，递归下降法利用缩进的方式来展示语法树并且标明了语法树结点在源程序中对应的行数。  总结：通过本次实验，从无到有的实现了编译器的部分功能（词法分析语法分析），对于编译器的工作原理和编译原理的实际使用有了较深的认识。在实验过程中，我们小组三人充分参与进行数据结构的讨论与确定，逐步完成小组间程序交互的任务。在程序成功运行，完成词法分析语法分析，真正的看到语法树的输出的那一刻，满足感爆棚。 | | | |