**《编译原理》实验报告**

|  |  |
| --- | --- |
| **实验名称** | 递归下降分析子程序 |
| 1. **实验目的：**   掌握最基本的自顶向下分析方法，即递归下降子程序方法，理解其特点和适用范围（回溯，左递归等现象），锻炼递归调用程序的构造方法。  给定表达式文法G[E]:  E→E+T | T  T→T\*F |F  F→( E ) | i  根据该文法，编写递归下降分析子程序。  1. 输入：任意的符号串( 上述文法的终结符号“(”、“)”、“i”、“+”、“\*”)  2. 处理：调用递归下降分析程序  3. 输出：判断输入串是否合法表达式 | |
| 1. **实验过程及步骤：** 2. 将给定的文法存储在.txt文件中格式如下：   6  E:E+T  E:T  T:T\*F  T:F  F:(E)  F:i  文本的第一行表示文法的条数，接下来6行分别表示文法的一个产生式   1. 编写dic和usetochange函数，用来构造消除左递归时使用的   转换字符   1. 编写消除直接左递归、消除间接左递归、提取左公共因子函数 2. 在提取左公共因子函数中，需要设置一个int regk来记录产生式的右部相等到第几个字符过，如果两个产生式左部相等且右部第一个字符相等，则存在左公共因子，这是遍历两个产生式的右部，如果符号相等则regk加一，以便截取相等的字符串加以改造。 3. 记录转换过后的文法的非终结符号，为每个非终结符号构造一个动作，递归下降分析输入   **程序源代码如下：**  package source\_code;  import java.io.BufferedReader;  import java.io.File;  import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.IOException;  import java.io.InputStreamReader;  import java.util.ArrayList;  import java.util.HashSet;  import java.util.List;  public class TopToDown {  //得到已被使用的字符之外的字符，用来改造  public List<Character> usetochange (List<Character> afterRemoveAllLeftRecursionRightpart){  List<Character> list = new ArrayList<>();  for (int i = 0; i < 26; i++) {  list.add((char) (65+i));  }  for(int i = 0 ; i < list.size();i++) {  for(char t:afterRemoveAllLeftRecursionRightpart) {  if(list.get(i).equals(t)) {  list.remove(i);  }  }  }  return list;  }  public List<String> ExtractionLeftFactor(List<String> mat, List<Character> Leftpart) {  List<String> myregList = new ArrayList<String>();  String myfirstString = new String();  int regk = 0; //用来指向第几个字符，用来分割分发  List<Integer> myintlist = new ArrayList<Integer>();//记录那几条文法参与了改造  // 遍历list中每一个项  for (int i = 0; i < mat.size(); i++) {  for (int j = i; j < mat.size(); j++) {  // 如果右部第一个字符相等且左部相等  if (mat.get(i).charAt(0) == mat.get(j).charAt(0) && mat.get(i).charAt(2) == mat.get(j).charAt(2)  && i != j) {  myintlist.add(i);  myintlist.add(j);  // 遍历每一个字符，如果字符相等，且字符序号相等  for (int k = 0; k < mat.get(i).length(); k++) {  for (int l = k; l < mat.get(j).length(); l++) {  if (mat.get(i).charAt(k) == mat.get(j).charAt(l) && k == l) {  myfirstString += mat.get(i).charAt(k); // 把相等的字符加入string  regk = k + 1;  }  }  }  if (mat.get(i).equals(myfirstString)) {//如果一个文法包含另一个文法，加入推出#和推出剩余不相等的符号  myregList.add(usetochange(Leftpart).get(i).toString() + ':' + '#');  myregList.add(usetochange(Leftpart).get(i).toString() + ':' + mat.get(j).substring(regk));  } else if (mat.get(j).equals(myfirstString)) {  myregList.add(usetochange(Leftpart).get(i).toString() + ':' + '#');  myregList.add(usetochange(Leftpart).get(i).toString() + ':' + mat.get(i).substring(regk));  } else {//如果两个文法没有包含关系  myregList.add(usetochange(Leftpart).get(i).toString() + ':' + mat.get(i).substring(regk));  myregList.add(usetochange(Leftpart).get(i).toString() + ':' + mat.get(j).substring(regk));  }  myfirstString += usetochange(Leftpart).get(i);  myregList.add(myfirstString);  myfirstString = "";  } else {  myregList.add(mat.get(j));  }  }  }  for (int i = 0; i < myregList.size(); i++) {  for (int j = 0; j < myintlist.size(); j++) {  if (myregList.get(i).equals(mat.get(myintlist.get(j)))) {  myregList.remove(i);  }  }  }  return myregList;  }  public List<String> RemoveLeftRecursion(List<String> mat) {// 消除直接在左递归函数  List<String> after = new ArrayList<String>();  List<String> myreg = new ArrayList<String>();  char splitcharacter = ':';  for (int i = 0; i < mat.size(); i++) {  if (mat.get(i).charAt(2) == mat.get(i).charAt(0)) {  // 形如 E：E+T 。 转换为 e:+Te  String a = "";  a += dic(mat.get(i).charAt(0)); // 调用转换函数  a += splitcharacter; // ：  a += mat.get(i).substring(3); // 截取左递归之后的string  a += dic(mat.get(i).charAt(0)); // 调用转换函数  after.add(a);  String b = "";  b += dic(mat.get(i).charAt(0));  b += splitcharacter;  b += '#';  after.add(b);  myreg.add(mat.get(i));// 寄存一条数据,后面用来判断是否要添加集合  for (int j = 0; j < mat.size(); j++) {  // 形如E：T 当T不等于E且E == E 时  if (mat.get(i).charAt(0) != mat.get(j).charAt(2) && mat.get(j).charAt(0) == mat.get(i).charAt(0)) {  // 形如 E:T 转换为 E：Te  after.add(mat.get(j) + dic(mat.get(i).charAt(0)));  myreg.add(mat.get(j));  }  }  }  }  for (int i = 0; i < mat.size(); i++) {  for (int j = 0; j < myreg.size(); j++) {  if (mat.get(i).equals(myreg.get(j))) {  mat.remove(i);  }  }  }  for (int i = 0; i < mat.size(); i++) {  after.add(mat.get(i));  }  after = new ArrayList<String>(new HashSet<String>(after));  return after;  }  //消除所有左递归  public List<String> RemoveAllRecursion(List<String> mat) {  List<String> afterall = new ArrayList<String>();  List<String> mynob = new ArrayList<String>();  for (int i = 0; i < mat.size(); i++) {  for (int j = 0; j < mat.size(); j++) {  if (mat.get(i).charAt(2) == mat.get(j).charAt(0)) {  String a = "";  a += mat.get(i).charAt(2);  afterall.add(mat.get(i).replaceAll(a, mat.get(j).substring(2)));  mynob.add(mat.get(i));  }  }  }  mynob = new ArrayList<String>(new HashSet<String>(mynob));  for (int i = 0; i < mat.size(); i++) {  for (int j = 0; j < mynob.size(); j++) {  if (mat.get(i).equals(mynob.get(j))) {  mat.remove(i);  }  }  }  for (String t : mat) {  afterall.add(t);  }  for (int i = 0; i < afterall.size(); i++) {  for (int j = 0; j < afterall.get(i).length(); j++) {  if (afterall.get(i).charAt(j) == '#') {  if (j + 1 < afterall.get(i).length()) {  String a = "";  a += afterall.get(i).substring(0, j);  a += afterall.get(i).substring(j + 1);  afterall.remove(i);  afterall.add(a);  }  }  }  }  return afterall;  }  public char dic(char A) {// 转换函数,形如E转换成e  return (char) (A ^ ' ');  }  int myPointer = 0, myerrorreg = 0;//记录匹配到哪一个字符，记录错误信息  public void E(String myInputString) {  if (myInputString.charAt(myPointer) == 'i') {  System.out.println("E->ite");  myPointer++;  t(myInputString);//  e(myInputString);  } else {  if (myInputString.charAt(myPointer) == '(') {  System.out.println("E->(E)te");  myPointer++;  E(myInputString);  if (myInputString.charAt(myPointer) == ')') {//  myPointer++;  t(myInputString);  e(myInputString);  } else if (myInputString.charAt(myPointer) != '$') {  myerrorreg++;  }  }  }  }  public void T(String myInputString) {  if (myInputString.charAt(myPointer) == 'i') {  System.out.println("T->it");  myPointer++;  t(myInputString);  } else {  if (myInputString.charAt(myPointer) == '(') {  System.out.println("T->(E)t");  myPointer++;  E(myInputString);  if (myInputString.charAt(myPointer) == ')') {  myPointer++;  t(myInputString);  } else {  myerrorreg++;  }  }  }  }  public void F(String myInputString) {  if (myInputString.charAt(myPointer) == '(') {  System.out.println("F->(E)");  myPointer++;  E(myInputString);  if (myInputString.charAt(myPointer) == ')') {  myPointer++;  } else {  myerrorreg++;  }  } else if (myInputString.charAt(myPointer) == 'i') {  System.out.println("F->i");  myPointer++;  // System.out.println(myPointer);  } else {  myerrorreg++;  }  }  public void t(String myInputString) {  if (myInputString.charAt(myPointer) == '\*') {  System.out.println("t->\*Ft");  myPointer++;  F(myInputString);  // System.out.println(myPointer);  t(myInputString);  } else if (myInputString.charAt(myPointer) == ')' || myInputString.charAt(myPointer) == '+'  || myInputString.charAt(myPointer) == '$') {  System.out.println("t->#");  } else {  myerrorreg++;  }  }  public void e(String myInputString) {  if (myInputString.charAt(myPointer) == '+') {  System.out.println("e->+Te");  myPointer++;  T(myInputString);  e(myInputString);  } else if (myInputString.charAt(myPointer) == ')' || myInputString.charAt(myPointer) == '$') {  System.out.println("e->#");  } else {  myerrorreg++;  }  }  public static void main(String[] args) throws IOException {  TopToDown toptodown = new TopToDown();  List<String> allline = new ArrayList<String>();// 文法  List<Character> notfinallist = new ArrayList<Character>();// 非终结符号  List<Character> leftpart = new ArrayList<Character>();// 文法左部  List<String> rightpart = new ArrayList<String>();// 文法右部  List<String> afterRemoveDirectLeftRecursion = new ArrayList<String>();// 文法消除直接左递归  List<String> regallline = new ArrayList<String>();// 寄存文法  List<String> afterRemoveAllLeftRecursion = new ArrayList<String>();// 文法消除所有  List<Character> afterRemoveAllLeftRecursionLeftpart = new ArrayList<Character>(); // 消除左递归后的文法左部  List<String> afterRemoveAllLeftRecursionRightpart = new ArrayList<String>();// 消除左递归后的文法右部  List<String> testfile = new ArrayList<String>();  String pathname = "E://tt.txt";  File filename = new File(pathname);  InputStreamReader read;  read = new InputStreamReader(new FileInputStream(filename));  BufferedReader br = new BufferedReader(read);  String oneline = "";  oneline = br.readLine();  int firstoneline = Integer.parseInt(oneline);  // allline 读取文法  for (int i = 0; i < firstoneline; i++) {  oneline = br.readLine();  allline.add(oneline);  }  String myTestPath = "E://test.txt";  File testfilename = new File(myTestPath);  InputStreamReader testread;  testread = new InputStreamReader(new FileInputStream(testfilename));  BufferedReader testbr = new BufferedReader(testread);  for (int i = 0; i < 5; i++) {  String testline = "";  testline = testbr.readLine();  testfile.add(testline);  }  // 寄存allline读取的文法  for (int i = 0; i < allline.size(); i++) {  regallline.add(allline.get(i));  }  // 读取文法的左部  for (int i = 0; i < allline.size(); i++) {  leftpart.add(allline.get(i).charAt(0));  }  // 读取文法的右部  for (int i = 0; i < allline.size(); i++) {  rightpart.add(allline.get(i).substring(2));  }  // 保存文法的非终结符号  notfinallist = new ArrayList<Character>(new HashSet<Character>(leftpart));  System.out.println("文法");  for (String t : allline) {  System.out.println(t);  }  System.out.println("非终结符号");  for (char t : notfinallist) {  System.out.println(t);  }  // 调用函数消除直接左递归  System.out.println("消除直接左递归");  afterRemoveDirectLeftRecursion = toptodown.RemoveLeftRecursion(regallline);  for (String t : afterRemoveDirectLeftRecursion) {  System.out.println(t);  }  System.out.println("消除所有左递归");  afterRemoveAllLeftRecursion = toptodown.RemoveAllRecursion(afterRemoveDirectLeftRecursion);  for (int i = 0; i < afterRemoveAllLeftRecursion.size(); i++) {  for (int j = 0; j < afterRemoveAllLeftRecursion.size(); j++) {  if (afterRemoveAllLeftRecursion.get(i).charAt(2) == afterRemoveAllLeftRecursion.get(j).charAt(0)) {  afterRemoveAllLeftRecursion = toptodown.RemoveAllRecursion(afterRemoveAllLeftRecursion);  }  }  }  for (String t : afterRemoveAllLeftRecursion) {  System.out.println(t);  }  // 存入左部和右部  for (String t : afterRemoveAllLeftRecursion) {  afterRemoveAllLeftRecursionLeftpart.add(t.charAt(0));  }  for (String t : afterRemoveAllLeftRecursion) {  afterRemoveAllLeftRecursionRightpart.add(t.substring(2));  }    System.out.println("提取左公共因子");  List<String> aftermmm = new ArrayList<String>();  aftermmm = toptodown.ExtractionLeftFactor(afterRemoveAllLeftRecursion, afterRemoveAllLeftRecursionLeftpart);  aftermmm = new ArrayList<String>(new HashSet<String>(aftermmm));  for (String t : aftermmm) {  System.out.println(t);  }  /\*  \* for (int i = 0; i < afterRemoveAllLeftRecursion.size(); i++) { for (int j =  \* 2; j < afterRemoveAllLeftRecursion.get(i).substring(2).length(); j++) { for  \* (int k = 0; k < afterRemoveAllLeftRecursionLeftpart.size(); k++) { if  \* (afterRemoveAllLeftRecursion.get(i).charAt(j) ==  \* afterRemoveAllLeftRecursionLeftpart.get(k)) {  \*  \* } } } }  \*/  // E(String myInputString, int myPointer, int myerrorreg)  //  System.out.println("测试句子：");  System.out.println(testfile.get(2));  System.out.println("测试结果：");  toptodown.E(testfile.get(2));  if (toptodown.myerrorreg > 0) {  System.out.println("error");  } else if (toptodown.myerrorreg == 0) {  System.out.println("right");  }  }  }  **测试程序如下：**  (i)\*i\*i\*i$  (i)$  (i\*i+i)\*i+(i)\*i$  (+i)$  (i+i)(i\*i)+i$ | |
| 1. **实验结果：** | |
| 1. **实验总结：**   通过这次实验，我对递归下降分析过程有了更进一步的了解。这次实验前，花了一个半小时理解书中的理论并且自己在纸上模拟了一遍分析过程。递归下降分析就是对每个非终结符按其产生式结构来构造相应语法分析子程序，其中终结符产生匹配命令，而非终极符则产生过程调用命令。因为文法递归相应子程序也递归，所以称这种方法为递归子程序下降法或递归下降法。有了前面的基础，在编程的时候对程序流程就比较熟悉了。通过这次实验，对递归下降分析程序的设计方法有了更深的理解，同时，也提高了自己的程序设计能力和调试技巧。总之，这次实习，我收获很大。 | |