**软件课程设计(Ⅱ)报告**

**姓 名：\_\_\_\_\_\_\_\_\_\_\_\_曾子昕\_\_\_\_\_\_\_\_\_\_**

**学 号：\_\_\_\_\_\_\_\_\_\_918106840617\_\_\_\_\_\_**

**成 绩：\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_**

**任务一：词法分析器**

1. **要求**

创建一个词法分析程序，该程序支持分析常规单词。必须使用 DFA（确定性有限自动机）或 NFA（不确定性有限自动机）来实现此程序。 程序有两个输入：一个文本文档，包括一组 3º型文法（正规文法）的产生式；一个源代码文本文档，包含一组需要识别的字符串（程序代码）。 程序的输出是一个 token（令牌）表，该表由 5 种 token 组成：关键词，标识符，常量，限定符和运算符。

要求：词法分析程序可以准确识别:科学计数法形式的常量（如 0.314E+1），复数常量（如10+12i），可检查整数常量的合法性，标识符的合法性（首字符不能为数字等），尽量符合真实常用高级语言要求的规则。

1. **基本思路**

程序使用的语法分析器基于有限自动机（finite automata）思想，由于利用

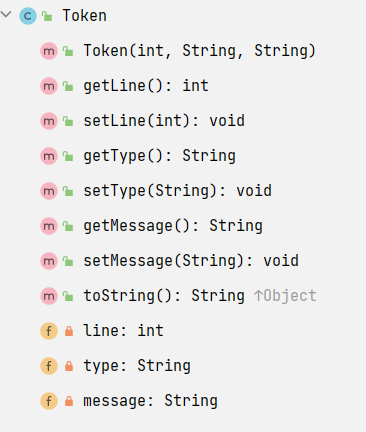
正规文法去表示 ID 和 CONSTANT 比较麻烦，于是本程序使用了正则表达式，基于了java 的正则表达式的 NFA 对输入文件进行逐字符处理，从而根据判断的顺序/条件得到识别的 token 串，能识别 常量*(*包括浮点数、科学计数法*)*、变量、关键词、操作符、分隔符。行注释、块注释，检测整数常量的合法性，标识符的合法性，生成Token序列。

首先从输入文本中按行读入待识别的代码，遇到块注释进入匹配块注释状态，直至块注释结束。遇到行注释进入匹配行注释状态直至行注释结束。其他情况下在 *关键字-》常量-》标识符-》操作符 -》分隔符-》错误* 状态间转换。

1. **主要类**

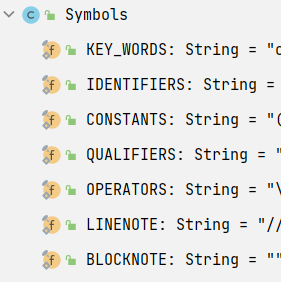
**3.1 Token**

该类是token的基本信息。保存token的行号、类别、内容。



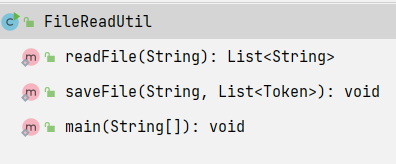
**3.2 Symbols**

定义了正则文法的规则。包括 关键字、标识符、常量、分隔符、操作符、行注释、块注释。



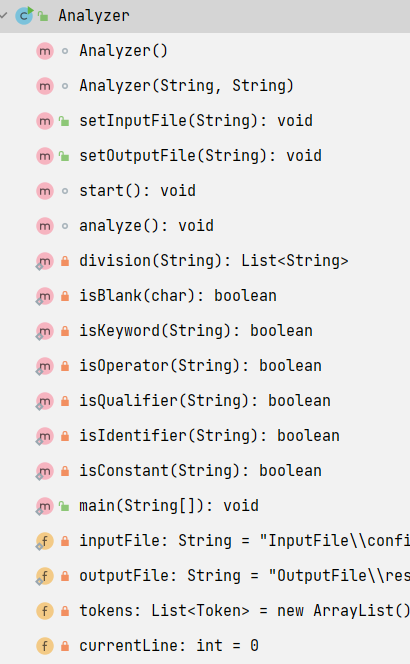
**3.3 FileReadUtil**

操作文件的工具类。能从配置文件中读取待操作的代码，也能保存识别到的token序列。



**3.4 Analyze**

完成词法分析的主类。包含字符切分、类别判断、词法分析。



1. **主要代码**

**4.1文法规则**

*\* 关键字*

public static final String *KEY\_WORDS* = "class|void|function|static|private|public|protected|char|int|boolean|double|bool|interface|var|if|else|while|do|for|this|return|null|print"; *\* 标识符, 正则表达式*public static final String *IDENTIFIERS* = "[a-zA-Z\_$][a-zA-Z\_0-9$]\*"; *\*常量，正则*public static final String *CONSTANTS* = "((\"(.\*)\")|('(\\\\)?[\\p{ASCII}]')|(\\d+(\\.\\d\*)?i)|(\\d\*(\\.)?\\d\*(E([+\\-])\\d\*(\\.)?\\d\*)?)|false|true)"; *\*分隔符，终结 { } . , ; ( ) [ ]*public static final String *QUALIFIERS* = ",|;|\\(|\\)|\\[|\\]|\\{|\\}|'|\"";  
 *\* 运算符*public static final String *OPERATORS* = "\\+|-|\\\*|/|\\+=|-=|\\\*=|/=|\\+\\+|--|&&|&|<|>|=|<=|>=|==|!=|~|\\||\\|||<<|>>"; *\* 行注释*public static final String *LINENOTE* = "//[^\\n]\*";

**4.2 divition()**

简单切分单词，将每行代码按空格、分隔符 切分存入list。

private static List*<*String*>* division*(*String s*) {* char*[]* chars = s.trim*()*.toCharArray*()*;  
 boolean isNote = false;  
 int lastIndex = 0;  
 *//去除首尾空格并转化为字符数组* List*<*String*>* list = new ArrayList*<>()*;  
 *//保存组合出的单词和字符* StringBuilder sb = new StringBuilder*()*;  
 for *(*int i = 0; i < chars.length; i++*) {* if*( isQualifier(*String.*valueOf(*chars*[*i*]))* ||*isBlank(*chars*[*i*])){* if *(*sb.length*()* != 0*)* list.add*(*sb.toString*()*.replaceAll*(*" ", ""*))*;  
 if *(*!*isBlank(*chars*[*i*]) )* list.add*(*String.*valueOf(*chars*[*i*]))*;  
 sb.delete*(*0, sb.length*())*; *//清空StringBuilder* continue;  
 *}* sb.append*(*chars*[*i*])*;  
 *}* return list;  
 *}*

**4.3 analyze()**

基本步骤如下：

1.按行读取读取程序；

2.如果遇到 块注释开始符 /\* 进入块注释匹配状态，直到匹配到块注释结束 \*/ ；

3.如果匹配到 行注释开始符 // 则将改行剩余字符全放入行注释；

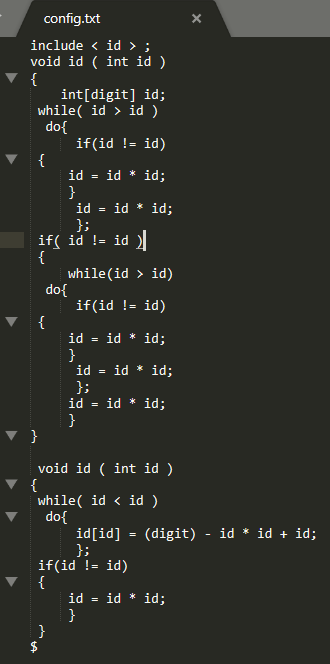
4.不在注释状态下，利用 java 自带的正则匹配，优先匹配出关键字，再是常量、标识符、操作符、分隔符；

5.都不是，则提示报错。

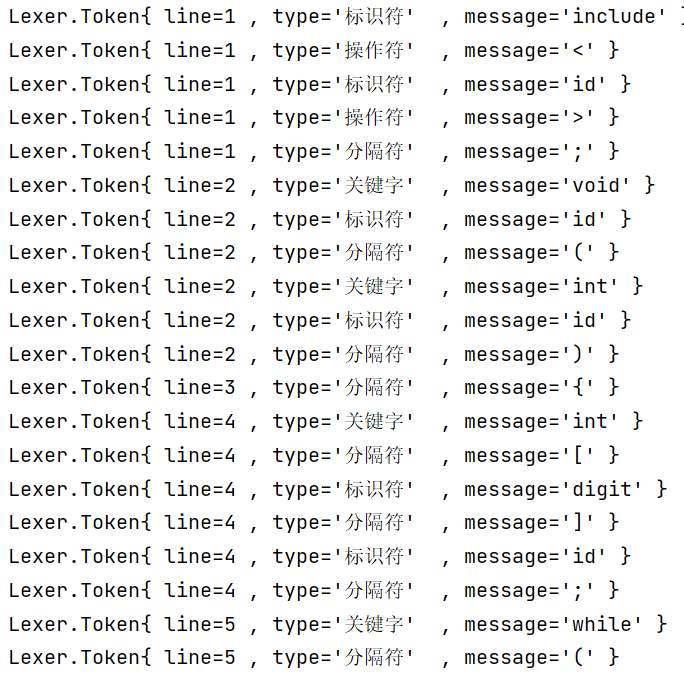
void analyze*()* throws IOException *{* List*<*String*>* lineList = new ArrayList*<>()*; *//* List*<*String*>* wordList = new ArrayList*<>()*; *//* Token currentToken = null;  
 String tem = "";  
 boolean isBlockNote = false;  
 try*{* lineList = FileReadUtil.*readFile(inputFile)*; *//按行读取文件* for *(*String line : lineList*) {* line = line.trim*()*;  
 currentLine++;  
  
 *//进行块注释处理* if*(*!isBlockNote && line.indexOf*(*"/\*"*)* != -1*) {* isBlockNote = true;  
 tem = "";  
 *}* if*(*isBlockNote*){* tem += line;  
 if*(*line.lastIndexOf*(*"\*/"*)* == line.length*()*-2*){* isBlockNote = false;  
 currentToken = new Token*(*currentLine,"块注释", tem*)*;  
 tokens.add*(*currentToken*)*;  
 *//System.out.println(currentToken);  
 }* continue;  
 *}  
  
  
 //判断行注释* if*(*Pattern.*matches(*Symbols.*LINENOTE*,line*) ){* currentToken = new Token*(*currentLine,"行注释", line*)*;  
 tokens.add*(*currentToken*)*;  
 *//System.out.println(currentToken);* continue;  
 *}  
  
 //进行词法分析* wordList = *division(*line*)*;  
 for *(*String word : wordList*) {  
 // 关键字-》常量-》标识符-》操作符 -》分隔符-》错误* if *( isKeyword(*word*) ) {* currentToken = new Token*(*currentLine,"关键字", word*)*;  
 *}*else if*( isConstant(*word*) ){* currentToken = new Token*(*currentLine,"常 量", word*)*;  
 *}*else if*( isIdentifier(*word*)){* currentToken = new Token*(*currentLine,"标识符", word*)*;  
 *}*else if *( isOperator(*word*) ) {* currentToken = new Token*(*currentLine,"操作符", word*)*;  
 *}* else if *( isQualifier(*word*)) {* currentToken = new Token*(*currentLine,"分隔符", word*)*;  
 *}*else *{* currentToken = new Token*(*currentLine,"错 误", word*)*;  
 *}* tokens.add*(*currentToken*)*;  
 *//System.out.println(currentToken);  
 }  
 }  
 }*catch*(*IOException e*){* e.printStackTrace*()*;  
 *}  
  
}*

1. **部分截图**

**5.1 配置文件**



**5.2 Token序列**



任务二：语法分析器

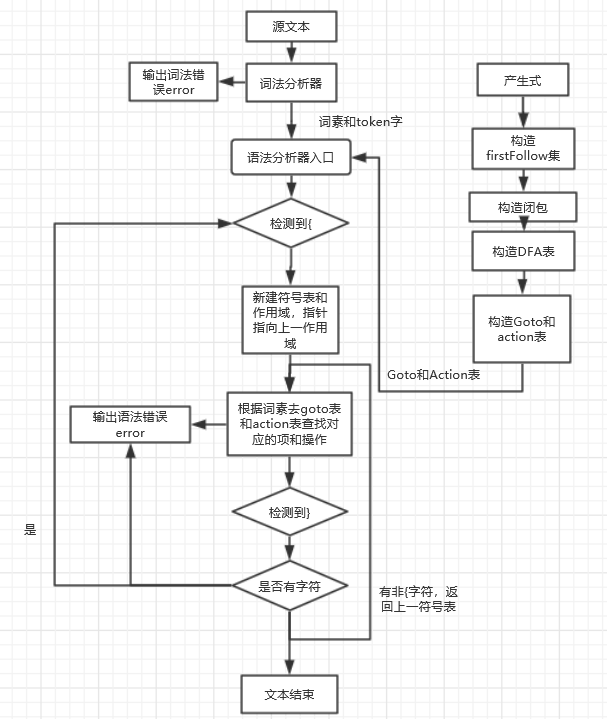
1. **要求**

创建一个使用 LL(1) 方法或 LR(1) 方法的语法分析程序。

程序有两个输入：1）一个是文本文档，其中包含 2º型文法（上下文无关法的产生式集合；2）任务 1 词法分析程序输出的（生成的）token 令牌表。程序的输出包括：YES 或 NO（源代码字符串符合此 2º型文法，或者源代码字符串不符合此 2º型文法）；错误提示文件，如果有语法错标示出错行号，并给出大致的出错原因。

1. **基本思路**

本程序根据LR1语言，输入文法规则、源代码程序，生成Closure集合，goto集合和action集合（根据翻译的产生式，从第一个非终结符S开始进行构建）然后把经过转换后的程序代码，根据goto表和action表进行规约，如果能规约到起始状态S，提示接受，否则报错。实现对 循环语句、if-else的分析。



1. **主要类**

**---Entity**

**FirstFollow**

**Item**

**ItemSet**

**Production**

**ProductionList**

**---Utils**

**FileOpt**

**FileUtils**

**MyStack**

**LRClouse**

**Goto**

**GotoTable**

**ActionTable**

**ItemTable**

**LR**

**3.1 Production**

产生式的基本结构类。

public class Production {  
 private String left; // 产生式左边  
 private String[] right; // 产生式右边

**3.2 ProductionList**

产生式表。用于构建、获取产生式。

public class ProductionList {

List<Production> productions = new ArrayList<Production>();

**3.3 Item**

LR1分析的项目类。

public class Item {  
 int index; // 点的位置  
 private String Left; // 产生式左边  
 private String[] alpha = {}; // 字母表  
 private String[] B; // 产生式右边的第一个变量和它前面的值  
 private String[] beta = {}; // 点右边 剩下的  
 private String[] a = {"$"}; // 多看的一项，向前搜索符  
 List<String> terminator; ///终结符表

**3.4 ItemSet**

项目集。通过循环遍历生成项目集。

public class ItemSet {  
 HashSet<Item> items = new HashSet<>(); // 项目集  
 HashSet<String> used = new HashSet<>(); // 已经添加过的

**3.5 FirstFollow**

求First、Follow 集。

public class FirstFollow *{* List*<*Production*>* proList;

**3.6 LRClouse**

求闭包。构造步骤：

(1)I中的所有项目都属于CLOSURE(I)；

(2)若项目[A→a.Bβ,a]属于CLOSURE(I)，B→ξ是一个产生式，那么，对于FIRST<βa>中的每一个中介符b,如果[β→.ξ,b]原来不在CLOSURE(I)中，则把它加进去；

(3)重复执行步骤（2），直到CLOSURE(I)不再增大为止。

public class LRClosure *{* ProductionList productionList;  
 List*<*Production*>* productions;  
 List*<*Item*>* items;  
 FirstFollow firstFollow;  
 List*<*String*>* used;

**3.7 ItemTabel**

项目集表。

public class ItemTable *{* LRClosure lrClosure; *// 初始的I0集合* Map*<*String, LRClosure*>* map; *// 闭包 键值对* int key = 0;  
 Map*<*GoTo, String*>* gotoMap; *// goto表*

**3.8 ActionTable**

action表。

public class ActionTable *{* int line; *// 列* int row; *// 行* String*[][]* actionTable; *// action表* List*<*Production*>* proList; *// 产生式集合* List*<*String*>* terminator; *// 终结符符号* Map*<*GoTo, String*>* gotoMap; *// goto图，保存了所有跳转信息* Map*<*String, LRClosure*>* lrMap; *// 闭包集合，包含整个dfa的每个项目集合*

**3.9 GotoTable**

goto表。

public class GoToTable *{* int line;*// 列* int row;*// 行* String*[][]* gotoTable; *// goto表* List*<*Production*>* proList; *// 产生式集合* Map*<*GoTo, String*>* gotoMap; *// goto图，保存了所有跳转信息* Map*<*String, LRClosure*>* lrMap; *// 闭包集合，包含整个dfa的每个项目集合* List*<*String*>* variables; *// 变量符号*

**3.10 MyStack**

分析栈。

public class MyStack *{* Stack*<*String*>* wordStack = new Stack*<>()*;

**3.11** **FileUtils**

读取、保存文件。

**3.12 LR**

语法分析的主类。在实现GO(I,X)时，记录下状态的转化。得到分析表中的移进部分。然后再扫描所有的项目集，找到其中包含归约项目的哪些项目集，根据其中项目，得到分析表中那些规约的部分。

1. **重点代码**

**4.1 获取产生式集合**

**ProductionList::setProList**

*/\*\*  
 \* 从 文件 里获取且设置产生式表，获得产生式集合,对产生式文件进行解析。  
 \*/*public void setProList*()* throws IOException *{* String productions = new FileOpt*()*.getProduces*(*path*)*;  
 String*[]* produces = productions.split*(*"\n"*)*;  
 String*[]* part; *// 切分左右* String*[]* rightItems; *// 右边的项分组* String*[]* rightPro; *// 右边* String*[]* leftPro; *// 左边* for *(*int i = 0; i < produces.length; i++*) {  
 //System.out.println(produces[i]);* part = produces*[*i*]*.split*(*" -> "*)*;  
 leftPro = part*[*0*]*.split*(*" "*)*;  
 if *(*part.length == 2*) {* rightItems = part*[*1*]*.split*(*"\\|"*)*;  
 for *(*int j = 0; j < rightItems.length; j++*) {* rightPro = rightItems*[*j*]*.split*(*" "*)*;  
 Production production = new Production*(*part*[*0*]*, rightPro*)*;  
 this.productions.add*(*production*)*;  
 *}  
 }* else *{* throw new Error*(*"一行产生式出现了两个箭头！读取错误"*)*;  
 *}  
 }  
}*

**4.2 获取First集**

**FirstFollow::getFirst**

*/\*\*  
 \* 获取first集合  
 \*/*public List*<*String*>* getFirst*(*String str*) {* List*<*String*>* list = new ArrayList*<>()*;  
 if *(*!isVariable*(*str*)) {* list.add*(*str*)*;  
 return list;  
 *}* List*<*Production*>* productions = findLeft*(*str*)*;  
 for *(*Iterator*<*Production*>* iterator = productions.iterator*()*; iterator  
 .hasNext*()*;*) {* Production production = *(*Production*)* iterator.next*()*;  
 if *(*isEmpty*(*production.getRight*())) {* list.add*(*"null"*)*;  
 *}* else if *(*!isVariable*(*production.getRight*()[*0*])* && !isEmpty*(*production.getRight*())) {* list.add*(*production.getRight*()[*0*])*;  
 *}* else *{* list.addAll*(*getFirstItem*(*production*))*;  
 *}  
 }* return list;  
*}*

**4.3 获取一个Item的闭包**

**LRclouse::getNextClosureItem**

*/\*\*  
 \* 从一个item出发，求他的闭包,嵌套求到全部的闭包  
 \*/*public List*<*Item*>* getNextClosureItem*(*Item item*) {* List*<*Item*>* items = new ArrayList*<>()*;  
 List*<*String*>* beta\_a = new ArrayList*<>()*;  
 for *(*int i = 0; i < item.getBeta*()*.length; i++*) {* beta\_a.add*(*item.getBeta*()[*i*])*;  
 *}* beta\_a = firstFollow.getFirst*(*beta\_a*)*;  
  
 if *(*beta\_a.isEmpty*()) {* for *(*int i = 0; i < item.getA*()*.length; i++*) {* if *(*!beta\_a.contains*(*item.getA*()[*i*])) {* beta\_a.add*(*item.getA*()[*i*])*;  
 *}  
 }  
 }* for *(*Iterator*<*Production*>* iterator =  
 findProduction*(*item.getNextB*())*.iterator*()*; iterator.hasNext*()*;*) {* Production pro = *(*Production*)* iterator.next*()*;  
 beta\_a.remove*(*""*)*;  
 Item tmp = new Item*(*pro, beta\_a, productionList.getTerminator*())*;  
 items.add*(*tmp*)*;  
 used.add*(*tmp.getLeft*())*;  
 *}* return items;  
*}*

**4.4 设置Action表**

**ActionTable::setActionTable**

*/\*\*  
 \* 设置action表  
 \*/*public void setActionTable*() {* for *(*Iterator*<*String*>* iterator = lrMap.keySet*()*.iterator*()*; iterator  
 .hasNext*()*;*) {* String setName = *(*String*)* iterator.next*()*;  
 LRClosure tmp = lrMap.get*(*setName*)*;  
 for *(*Iterator*<*Item*>* iterator2 = tmp.items.iterator*()*; iterator2  
 .hasNext*()*;*) {  
 // System.out.println("ActionTable.setActionTable()");* Item item = *(*Item*)* iterator2.next*()*;  
 *// System.out.println(item);  
 // System.out.println("----------------------------------------");  
 // System.out.println(item.getNextB());* if *(*item.getNextB*()*.equals*(*""*)* && !item.getLeft*()*.equals*(*"S'"*)) {  
 // 如果[A->alpha·,a]在Ii里，而且A!=S’,那么将Action[i,a]设置为规约A->alpha* StringBuffer stringBuffer = new StringBuffer*()*;  
 for *(*int i = 0; i < item.getAlpha*()*.length; i++*) {* stringBuffer.append*(*item.getAlpha*()[*i*])*;  
 *}* for *(*int i = 0; i < item.getA*()*.length; i++*) {* if *(*!setTable*(*setName, item.getA*()[*i*]*,  
 "r" + getProductionId*(*item*))) {* throw new Error*(*"规约表设置出现了错误！！产生了语义冲突！！"*)*;  
 *}  
 }  
 }* else if *(*item.getLeft*()*.equals*(*"S'"*)* && item.getNextB*()*.equals*(*""*)* && item.setCheckAcc*()) {* setTable*(*setName, "$", "ACC"*)*;  
 *}* else if *(*terminator.contains*(*item.getNextB*())) {// 如果不是变量，就去找goto表  
 // 查找有没有GoTo（Ii，a）=Ij,有的话就把action[i,a]设置为移入j，sj* String result = findGotoTable*(*setName, item.getNextB*())*;  
 *// System.out.println("????????????????????");  
 // System.out.println(result);* if *(*!result.equals*(*""*)) {  
 // 把action[i,a]设置为移入j，sj* if *(*!setTable*(*setName, item.getNextB*()*, "s" + result*)) {* throw new Error*(*"移入表设置出现了错误！！产生了语义冲突！！！！"*)*;  
 *}  
 }  
 }  
 }  
 }  
}*

**4.5 设置Goto表**

**GotoTable::setGotoTable**

*/\*\*  
 \* 如果GOTO(Ii,A)=Ij,那么GOTO[i,A]=j  
 \*/*public void setGotoTable*() {* for *(*Iterator*<*GoTo*>* iterator = gotoMap.keySet*()*.iterator*()*; iterator  
 .hasNext*()*;*) {* GoTo goTo = *(*GoTo*)* iterator.next*()*;  
 if *(*variables.contains*(*goTo.getPath*())) {  
 // 如果是变量的话* if *(*!setTable*(*goTo.getClosureID*()*, goTo.getPath*()*, gotoMap.get*(*goTo*))) {* throw new Error*(*"goto设置出现了错误！！产生冲突！！"*)*;  
 *}  
 }  
 }  
  
}*

**4.5 设置Item项目集**

**ItemSetTable::setItemSet**

*/\*\*  
 \*  
 \* 完成goto表的构建，  
 \* 这个函数负责从初始开始构建一个第一层closure，也就是从I0到后面的一级推导，  
 \* 然后它把得到的map传到一个新的函数，这个map包含了第二层的闭包，  
 \* 新的函数负责从一个map的闭包集合求解下一层闭包，如果这个闭包集合已经存在，就直接建立goto表，  
 \* 如果不存在，就新建一个map存下一层的闭包，然后迭代自身求解，直到闭包集合不再增加为止。  
 \*  
 \*/*public Map*<*String, LRClosure*>* setItemSet*(*LRClosure closure, String setName*) {* Map*<*String, LRClosure*>* lrClosure = new HashMap*<>()*;  
 map.put*(*setName, closure*)*;  
 key++;  
 closure.setClosureItem*(*closure.productions.get*(*0*))*;*// 初始第一个闭包* for *(*Iterator*<*String*>* iterator = closure.gotoPath*()*.iterator*()*; iterator  
 .hasNext*()*;*) {* String path = *(*String*)* iterator.next*()*;  
 *// closure.getNextClosure(type);* LRClosure tmp = closure.getNextClosure*(*path*)*;  
 if *(*!map.containsValue*(*tmp*)) {* String name = new String*(*"I" + key*)*;  
 map.put*(*name, tmp*)*;  
 lrClosure.put*(*name, tmp*)*;  
 gotoMap.put*(*new GoTo*(*setName, path*)*, name*)*;  
 key++;  
 *}* else *{* gotoMap.put*(*new GoTo*(*setName, path*)*, getOutClosure*(*map, tmp*))*;  
 *}  
 }* setItemSetItem*(*lrClosure*)*; *// 新的函数负责从一个map的闭包集合求解下一层闭包  
 /\* 5.4 可能会用\*\*/  
 //System.out.println(map);  
 //System.out.println(gotoMap);* return map;  
*}*

**4.6 进行分析**

**LR::analyze**

public static void analyse*(*String grammarPath,String sourcePath, String actionPath, String gotoPath*){* ProductionList productionList = new ProductionList*(*grammarPath*)*;  
 LR LR = new LR*(*productionList*)*;  
  
 MyStack stack = new MyStack*()*;  
 stack.push*(*"I0"*)*;  
  
 String action = "";  
 int wordID = 0;  
 List*<*String*>* words = null;  
  
 *//读取 词法分析器产生 的token* try *{* words = FileUtils.*readWord(*sourcePath*)*;  
 *}* catch *(*IOException e*) {* e.printStackTrace*()*;  
 *}* while *(*!action.equals*(*"ACC"*)* && wordID <= words.size*()) {  
 // 读取当前输入符号和栈顶部状态* String word = words.get*(*wordID*)*;  
 String top = stack.getTop*()*;  
 action = LR.searchActionTable*(*word, top*)*;if *(*action.equals*(*"ACC"*)) {* break;  
 *}* else if *(*action.contains*(*"s"*)) {* action = action.replaceAll*(*"s", ""*)*;  
 System.*out*.println*(*"移入: " + word*)*;  
 wordID++;  
 stack.push*(*action*)*;  
 *}* else if *(*action.contains*(*"r"*)) {* stack.printStack*()*;  
 int productionId = Integer.*parseInt(*action.replaceAll*(*"r", ""*))*;  
 Production production = productionList.getProductions*()*.get*(*productionId*)*;  
 System.*out*.println*(*"规约: " + production*)*;  
 int r = productionList.getProductions*()*.get*(*productionId*)*.getRight*()*.length;  
 while *(*r > 0*) {* System.*out*.println*(*"Parse.main()"*)*;  
 System.*out*.println*(*"栈顶: " + stack.pop*() )*;  
 r--;  
 *}* System.*out*.println*(*"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"*)*;  
 stack.push*(*LR.searchGotoTable*(*stack.getTop*()*,  
 productionList.getProductions*()*.get*(*productionId*)*.getLeft*()))*;  
 *}* else *{* throw new Error*(*"出现错误！"*)*;  
 *}  
 }* if *(*action.equals*(*"ACC"*)* && wordID == words.size*()* - 1*) {* System.*out*.println*()*;  
 System.*out*.println*(*"-----------------YES------------------"*)*;  
 *}*

*}*

1. **部分截图**

**程序的产生式：**

S' -> <程序>

<程序> -> <声明列表>|<程序> <函数>

<声明列表> -> <声明>|<声明列表> <声明>

<声明> -> include < <标识符> > ;

<函数> -> <修饰符> <标识符> <形式参数> <复合语句>

<修饰符> -> void|int|char|float|double

<标识符> -> id

<形式参数> -> ( <数据类型> <标识符> )

<数据类型> -> int|float|double|char|<数据类型> C1

C1 -> [ digit ]

<复合语句> -> { <语句列表> }|{ }

<复合语句> -> { <声明语句> <语句列表> }|{ }

<声明语句> -> <数据类型> <标识符> ;|<数据类型> <赋值语句>

<语句列表> -> <语句>|<语句列表> <语句>

<语句> -> <条件语句>|<赋值语句>|<循环语句>

<赋值语句> -> <Left> = <表达式> ;

<循环语句> -> while <b> do { <语句列表> } ;

<条件语句> -> <S1>|<S2>

<S1> -> if <b> { <语句列表> } else { <语句列表> }

<S2> -> if <b> { <语句列表> }

<b> -> ( <逻辑表达式> )

<逻辑表达式> -> <标识符> <逻辑运算符> <标识符>

<逻辑运算符> -> >=|<=|<|!|==|!=|>

<表达式> -> <表达式> + T|T|<表达式> - T

T -> F \* F|F / F|F

F -> digit|<标识符>|( <表达式> )

<Left> -> Elist ]|id

Elist -> id [ <表达式>

**待分析源代码：**

include <id> ;

void id (int id)

{

int[digit] id;

while(id > id)

do{

if(id != id)

{

id=id\*id;

}

id=id\*id;

};

if(id != id)

{

while(id > id)

do{

if(id != id)

{

id=id\*id;

}

id=id\*id;

};

id=id\*id;

}

}

void id (int id)

{

while(id < id)

do{

id[id]=(digit)-id\* id+id;

};

if(id != id)

{

id=id\*id;

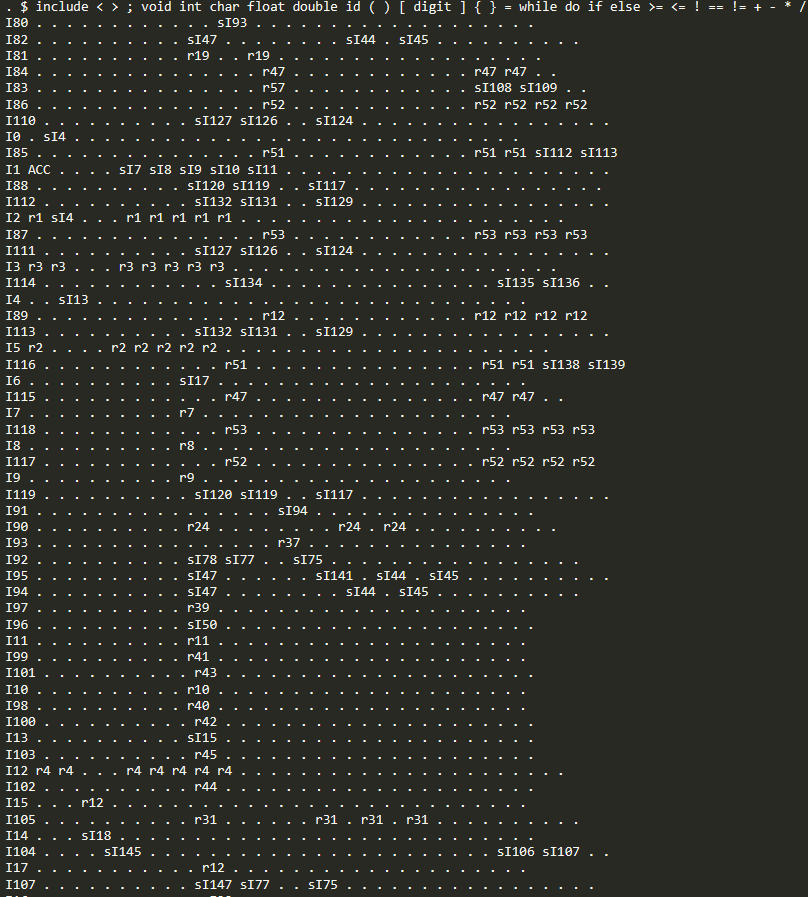
}

}

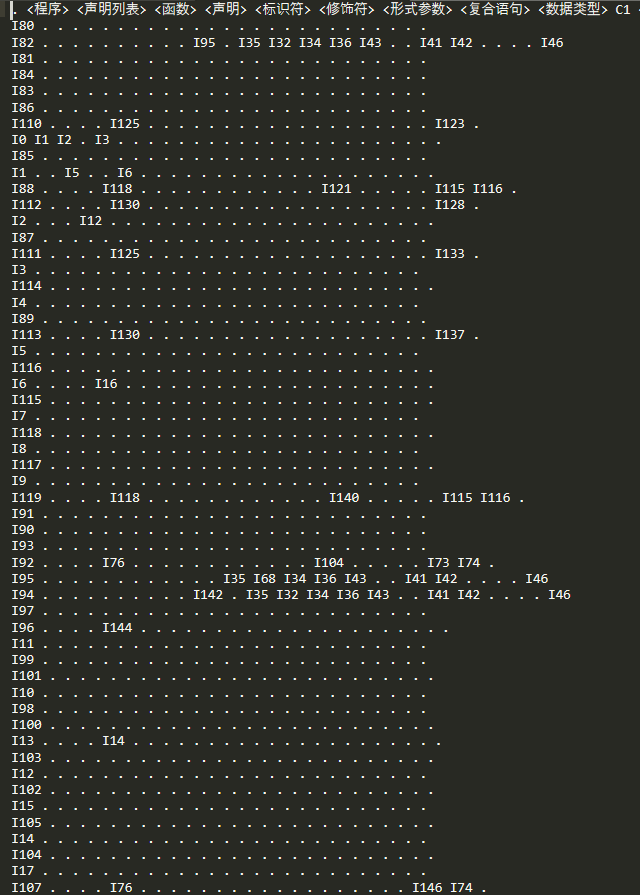
$

**Action表：**

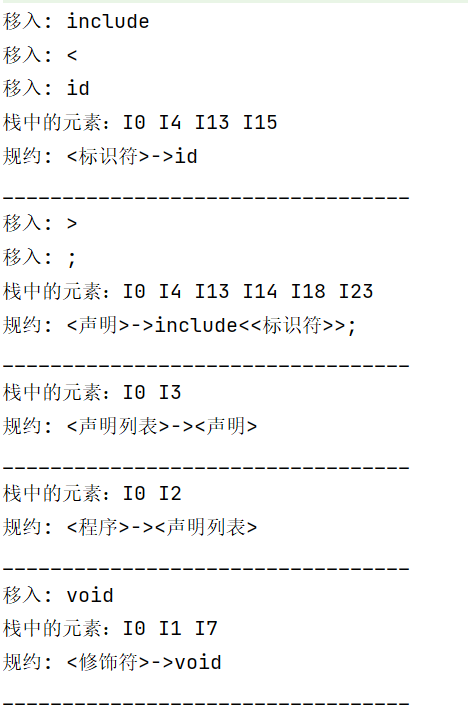
（完整版太长，仅截取部分展示）



**Goto表：(部分)**

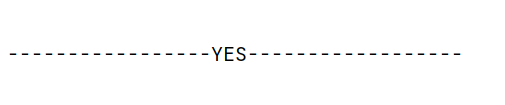


**部分规约步骤：**

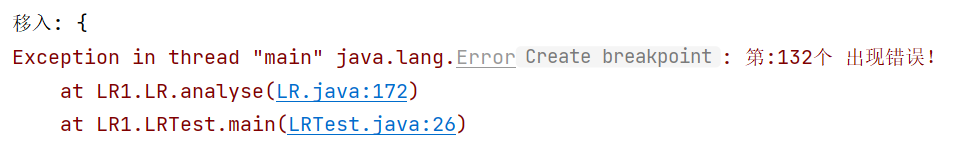


**分析结果：**

**成功：**



**失败：**



个人小结

与前两次的C++语言、java web课设相比，本次课设的难度可以说是高了一个档次。在上学期，我自认为在编译原理这门课上花费了足够多的时间，然而最终的考试成绩却告诉我努力的方向恐怕错了,所以在一开始对本次课设我是十分惧怕的。

由于以前对编译的认知仅停留在书本的题目上，看到要求的第一时间顿感无从下手。任务的要求也很模糊，只说了输入输出，没有一个样例可以参考，最终只得从网络上根据他人的一点点摸索。

本次的两个任务难度都很大。完成词法分析器用了四天，语法分析器断断续续花了六天。我也从此次课程设计中受益良多，从一开始的不知道从何入手，再到决定用的编程语言、设计程序流程、调试，最后到程序运行成功。较好的文法分析器是功能全面的能自动构造其项目集和转换函数并构造分析表，然后进行分析的程序。在编程过程中也遇到了很多困难，如对某些终结符的动作或是规约函数的实现，最后通过请教同学和上网查找参考资料，都得到了解决。

底层知识的学习是痛苦而又难以迅速见效的，但它们对计算机思维的培养却是无法匹敌的。可惜学业繁重，时间紧张，无法继续钻研，希望日后有缘再见！