

《编译原理》研究性学习专题实验报告

实验5： 基于SLR(1)分析法的语法制导翻译及中间代码生成程序设计原理与实现

学 院：计算机与信息技术学院

专 业： 计算机科学与技术

学生姓名： 杜永坤

学 号： 16281002

班 级： 计算机1601班

指导教师： 于双元

北京交通大学

2019年06月04日

[一、实验要求： 3](#_Toc10552256)

[1.[实验项目] 3](#_Toc10552257)

[2.[设计说明] 3](#_Toc10552258)

[3.[设计要求] 3](#_Toc10552259)

[二、实验过程： 4](#_Toc10552260)

[1、程序功能描述 4](#_Toc10552261)

[2、主要数据结构描述 4](#_Toc10552262)

[2.1主要使用的java数据结构类型 4](#_Toc10552263)

[2.1.1 List 4](#_Toc10552264)

[list中是否包含某个元素 4](#_Toc10552265)

[list获取长度： 4](#_Toc10552266)

[list中查看（判断）元素的索引 5](#_Toc10552267)

[根据元素索引位置进行的判断 5](#_Toc10552268)

[判断list是否为空 6](#_Toc10552269)

[2.1.2 Map 6](#_Toc10552270)

[HashMap 6](#_Toc10552271)

[常用API 6](#_Toc10552272)

[Map的遍历 7](#_Toc10552273)

[Map获取元素 7](#_Toc10552274)

[2.2 二元式文件结构 8](#_Toc10552275)

[1）FirstAndFollow类： 8](#_Toc10552276)

[2）ProductionFormula类： 9](#_Toc10552277)

[3）AssignmentGrammar类： 9](#_Toc10552278)

[4）StateDFA类： 10](#_Toc10552279)

[5）DFAManager类： 11](#_Toc10552280)

[状态图： 15](#_Toc10552281)

[6）SLR1AnalysisTable类： 15](#_Toc10552282)

[7）AssignmentTranslationGrammar类： 17](#_Toc10552283)

[8）SLR1Analyzer类： 17](#_Toc10552284)

[9）TranslationMain类： 19](#_Toc10552285)

[3、程序结构描述 19](#_Toc10552286)

[1）FirstAndFollow类中 19](#_Toc10552287)

[2）ProductionFormula类： 20](#_Toc10552288)

[3）AssignmentGrammar类： 20](#_Toc10552289)

[4）StateDFA类： 21](#_Toc10552290)

[5）DFAMananger类： 21](#_Toc10552291)

[6）SLR1AnalysisTable类： 22](#_Toc10552292)

[7）AssignmentTranslationGrammar类： 22](#_Toc10552293)

[8）SLR1Analyzer类： 23](#_Toc10552294)

[9）TranslationMain类： 24](#_Toc10552295)

[三、程序测试 24](#_Toc10552296)

[3.1、正确样例 24](#_Toc10552297)

[3.2、错误样例 25](#_Toc10552298)

[3.2.1缺少 = 25](#_Toc10552299)

[3.2.1缺少操作符 26](#_Toc10552300)

### 一、实验要求：

1.[实验项目]

完成以下描述赋值语句 SLR(1)文法语法制导生成中间代码四元式的过程。G[E]:

S→V=E

E →E+T∣E-T∣T

T→T\*F∣T/F∣F

F→(E)∣i

V=I

2.[设计说明]

终结符号i 为用户定义的简单变量,即标识符的定义。

3.[设计要求]

（1）构造文法的 SLR(1)分析表，设计语法制导翻译过程，给出每一产生式对应的语义动作；

（2）设计中间代码四元式的结构；

（3）输入串应是词法分析的输出二元式序列，即某赋值语句“专题 1”的输出结果，输出为赋值语句的四元式序列中间文件；

（4）设计两个测试用例（尽可能完备），并给出程序执行结果四元式序列。

### 二、实验过程：

#### 1、程序功能描述

1、本程序能够根据所给定的文法构造其拓展文法

2、本程序能够根据拓展文法自动生成识别活前缀的DFA

3、本程序能够根据活前缀的DFA构造SLR（1）分析表

4、本程序能够进行SLR（1）语法分析，并将赋值语句翻译成四元式

#### 2、主要数据结构描述

##### 2.1主要使用的java数据结构类型

###### 2.1.1 List

//终结符号集

**private** List<Character> Vt = **new** ArrayList<Character>();

//非终结符号集

**private** List<Character> Vn = **new** ArrayList<Character>();

使用List保存文法的非终结符号和终结符号，一旦文法给定，非终结符号集合和终结符号集合也就确定，所以对Vt、Vn的操作一般是查找元素和遍历元素。

list中是否包含某个元素

方法：.contains（Object o）； 返回true或者false

例如下面在程序中使用到的：

Vn.contains(nArryStr[i].charAt(0))&&Vt.contains(nArryStr[i].charAt(1))

list获取长度：

Vt.size()

使用size()即可返回List的长度

list中查看（判断）元素的索引

　　注意：.indexOf（）； 和 lastIndexOf（）的不同；

List<String> names=new ArrayList<>();

names.add("刘备"); //索引为0

names.add("关羽"); //索引为1

names.add("张飞"); //索引为2

names.add("刘备"); //索引为3

names.add("张飞"); //索引为4

System.out.println(names.indexOf("刘备"));

System.out.println(names.lastIndexOf("刘备"));

System.out.println(names.indexOf("张飞"));

System.out.println(names.lastIndexOf("张飞"));

根据元素索引位置进行的判断

if (names.indexOf("刘备")==0) {

System.out.println("刘备在这里");

}else if (names.lastIndexOf("刘备")==3) {

System.out.println("刘备在那里");

}else {

System.out.println("刘备到底在哪里？");

}

在程序中使用OPGtable[Vt.indexOf(Vt1)][Vt.indexOf(Vt2)]

判断list是否为空

//空则返回true，非空则返回false

if (person.isEmpty()) {

System.out.println("空的");

}else {

System.out.println("不是空的");

}

###### 2.1.2 Map

HashMap

最常用的Map,它根据键的HashCode 值存储数据,根据键可以直接获取它的值，具有很快的访问速度。HashMap最多只允许一条记录的键为Null(多条会覆盖);允许多条记录的值为 Null。非同步的。

常用API

|  |  |
| --- | --- |
| clear() | 从 Map 中删除所有映射 |
| remove(Object key) | 从 Map 中删除键和关联的值 |
| put(Object key, Object value) | 将指定值与指定键相关联 |
| putAll(Map t) | 将指定 Map 中的所有映射复制到此 map |
| entrySet() | 返回 Map 中所包含映射的 Set 视图。Set 中的每个元素都是一个 Map.Entry 对象，可以使用 getKey() 和 getValue() 方法（还有一个 setValue() 方法）访问后者的键元素和值元素 |
| keySet() | 返回 Map 中所包含键的 Set 视图。删除 Set 中的元素还将删除 Map 中相应的映射（键和值） |
| values() | 返回 map 中所包含值的 Collection 视图。删除 Collection 中的元素还将删除 Map 中相应的映射（键和值） |
| get(Object key) | 返回与指定键关联的值 |
| containsKey(Object key) | 如果 Map 包含指定键的映射，则返回 true |
| containsValue(Object value) | 如果此 Map 将一个或多个键映射到指定值，则返回 true |
| isEmpty() | 如果 Map 不包含键-值映射，则返回 true |
| size() | 返回 Map 中的键-值映射的数目 |

在程序中使用了MAP:

//用map来重新构造文法

**private** Map<Character, String> mapGrammar = **new** HashMap<Character, String>();

//First集

**private** Map<Character,Set<Character>> First = **new** HashMap<Character,Set<Character>>();

//Follow集

**private** Map<Character,Set<Character>> First = **new** HashMap<Character,Set<Character>>();

使用map保存文法，和First, First。

Map的遍历

mapGrammar.forEach((k,v)->{

//处理k，v

});

使用forEach方法，参数(k,v)

其中k为每个values的索引值，通过forEach的方法，遍历map的所有元素，就返回每个元素的key和values，并赋给参数（k,v）。

Map获取元素

Map获取元素一般根据key值获取对应的values：

例如：

First.get(k.charAt(0))

结果返回的是，key对应的values，而且values是声明map时设置的类型对应，

比如：

**private** Map<Character,Set<Character>> First = **new** HashMap<Character,Set<Character>>();

key的类型为Character

values的类型为Set<Character>

所以通过First.get(key)获得values时，得到类型为Set<Character>。

##### 2.2 二元式文件结构

##### 1）FirstAndFollow类：

根据所给定的文法，构造非终结符号的FIRST集和FOLLOW集。

FIRST集和FOLLOW集的数据结构：

//First集

**private** Map<Character,Set<Character>> First = **new** HashMap<Character,Set<Character>>();

//Follow集

**private** Map<Character,Set<Character>> Follow = **new** HashMap<Character,Set<Character>>();

主要采用Map数据结构来处理FIRST和FOLLOW，map的操作在[2.1.2Map](#_2.1.2_Map)中有详细的叙述。

##### 2）ProductionFormula类：

对字符串类型的产生式进行封装，由于之后需要对产生式进行“.”的移动操作，因此在本类中设计了一个movePoint方法，用于在本产生式的基础上右移“.”产生新的产生式。

同时之后也会设计到产生式的查询和比较，因此，在此重写该类的hashCode方法和equals方法。

##### 3）AssignmentGrammar类：

该类是对字符串文法进行处理的基本类，将文法拆解成没有|符号的文法，生成含有“.”符号的拓展文法，统计文法中的终结符号，非终结符号等等。

该类涉及到了文法的数据结构和终结文法和非终结符号的数据结构：

**private** String []grammarnArry = {

"A->V=E",

"E->E+T|E-T|T",

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

"F->(E)|i",

"V->i"

};

**private** Character startSymbol = 'G'; //文法的 开始符号

//去掉|后的基本文法

**private** List<String> basicGrammar = **new** ArrayList<String>();

//基于基本文法形成的第一批拓展文法，其中s'用符号G代替

**private** **static** List<ProductionFormula> *expendGrammar* = **new** ArrayList<ProductionFormula>() ;

//终结符号集

**private** Set<Character> Vt = **new** HashSet<Character>();

//非终结符号集

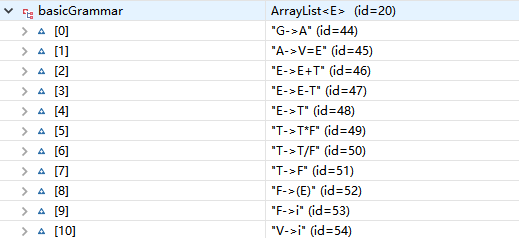
**private** Set<Character> Vn = **new** HashSet<Character>();

//所有符号集

**private** Set<Character> V = **new** HashSet<Character>();

文法通过手动设置，保存到grammarnArry中。

Vt、Vn根据文法进行设置。V为Vt和Vn的并集。

basicGrammar为将|去掉的后的文法，且将扩展文法加入G→A，其内容为：

*expendGrammar*为加入·后的文法，构造如下：

G->.A

A->.V=E

E->.E+T

E->.E-T

E->.T

T->.T\*F

T->.T/F

T->.F

F->.(E)

F->.i

V->.i

##### 4）StateDFA类：

该类用于描述DFA状态装换图中的一个结点，主体部分是一个产生式的集合也就是对应状态的项目集，以及一个字符到另一个StateDFA对象的Map，用来表示DFA中的状态转换机制。

//该状态对应的项目集

**private** Set<ProductionFormula> productionSet = **new** HashSet<ProductionFormula>();

//状态转换关系

**private** Map<Character,StateDFA> transform = **new** HashMap<Character,StateDFA>();

例如状态10：

状态10:

T->.T\*F E->E+.T T->.T/F T->.F F->.(E) F->.i

状态10--T-->状态15

状态10--F-->状态6

状态10--(-->状态7

状态10--i-->状态8

productionSet对应为：

T->.T\*F E->E+.T T->.T/F T->.F F->.(E) F->.i

Transform对应为：key为经过的非终结符号或者终结符号，values为到达的状态：

Key：T values：状态15

Key：F values：状态6

Key：( values：状态7

Key：I values：状态8

##### 5）DFAManager类：

指导和管理StateDFA对象的生成，建立整个DFA状态转换图。

**private** List<StateDFA> StateList = **new** ArrayList<StateDFA>(); //全局的状态链

该文法的DFA为：

状态0:

V->.i G->.A A->.V=E

状态0--A-->状态1

状态0--V-->状态3

状态0--i-->状态2

状态1:

G->A.

本结点为终端结点

状态2:

V->i.

本结点为终端结点

状态3:

A->V.=E

状态3--=-->状态4

状态4:

A->V=.E E->.E+T T->.T\*F E->.E-T T->.T/F T->.F F->.(E) E->.T F->.i

状态4--T-->状态9

状态4--E-->状态5

状态4--F-->状态6

状态4--(-->状态7

状态4--i-->状态8

状态5:

A->V=E. E->E.-T E->E.+T

状态5--+-->状态10

状态5----->状态11

状态6:

T->F.

本结点为终端结点

状态7:

E->.E+T T->.T\*F E->.E-T T->.T/F T->.F F->.(E) F->(.E) E->.T F->.i

状态7--T-->状态9

状态7--E-->状态12

状态7--F-->状态6

状态7--(-->状态7

状态7--i-->状态8

状态8:

F->i.

本结点为终端结点

状态9:

T->T./F E->T. T->T.\*F

状态9--\*-->状态13

状态9--/-->状态14

状态10:

T->.T\*F E->E+.T T->.T/F T->.F F->.(E) F->.i

状态10--T-->状态15

状态10--F-->状态6

状态10--(-->状态7

状态10--i-->状态8

状态11:

T->.T\*F E->E-.T T->.T/F T->.F F->.(E) F->.i

状态11--T-->状态16

状态11--F-->状态6

状态11--(-->状态7

状态11--i-->状态8

状态12:

F->(E.) E->E.-T E->E.+T

状态12--)-->状态17

状态12--+-->状态10

状态12----->状态11

状态13:

T->T\*.F F->.(E) F->.i

状态13--F-->状态18

状态13--(-->状态7

状态13--i-->状态8

状态14:

F->.(E) F->.i T->T/.F

状态14--F-->状态19

状态14--(-->状态7

状态14--i-->状态8

状态15:

T->T./F E->E+T. T->T.\*F

状态15--\*-->状态13

状态15--/-->状态14

状态16:

T->T./F E->E-T. T->T.\*F

状态16--\*-->状态13

状态16--/-->状态14

状态17:

F->(E).

本结点为终端结点

状态18:

T->T\*F.

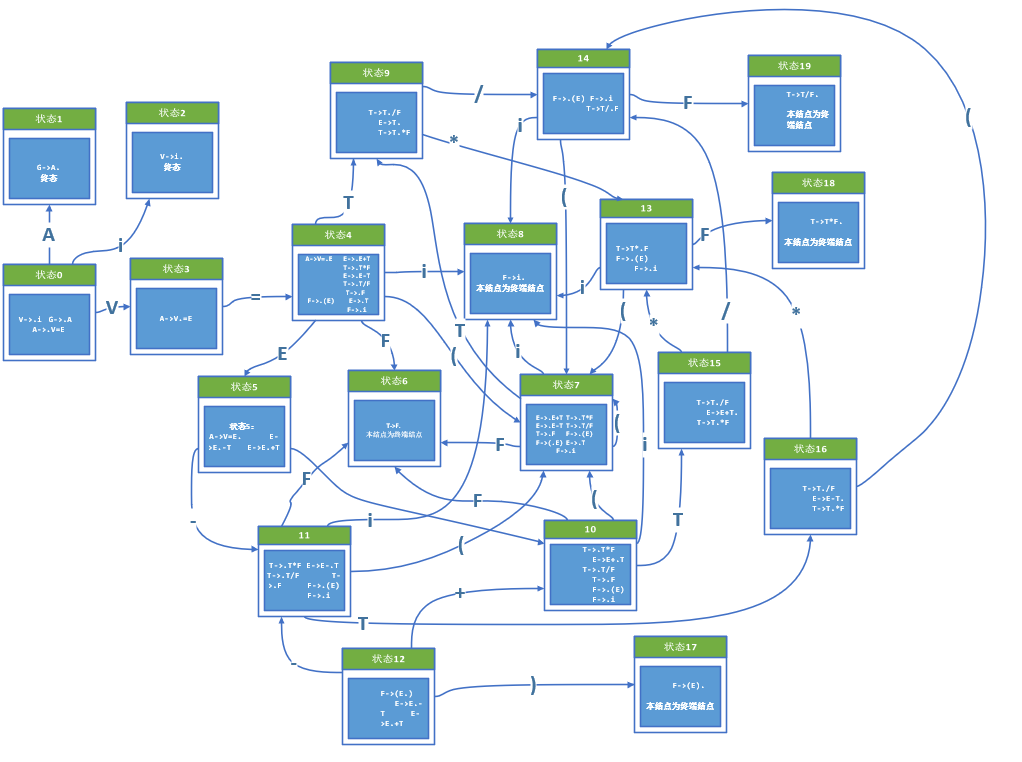
本结点为终端结点

状态19:

T->T/F.

本结点为终端结点

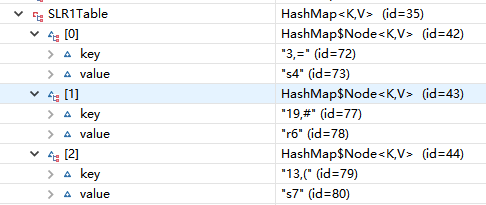
###### 状态图：



##### 6）SLR1AnalysisTable类：

用一个DFAManager对象生成一个该对象的特定的SLR(1)分析表，数据的组织方式依然采用Map的形象，即“[状态，输入字符]->动作”的形式存储。

**private** Map<String,String> SLR1Table = **new** HashMap<String,String>();

 内容为：

可以看到key为3,= 3表示状态，=为状态字符，value为s4，也就是下一个状态。

SLR1Table保存了完整的SLR(1)分析表为：

ACTION GOTO

( ) i \* + = - / # | A T E V F G

0 s2 | 1 3

1 acc |

2 r10 |

3 s4 |

4 s7 s8 | 9 5 6

5 s10 s11 r1 |

6 r7 r7 r7 r7 r7 r7 |

7 s7 s8 | 9 12 6

8 r9 r9 r9 r9 r9 r9 |

9 r4 s13 r4 r4 s14 r4 |

10 s7 s8 | 15 6

11 s7 s8 | 16 6

12 s17 s10 s11 |

13 s7 s8 | 18

14 s7 s8 | 19

15 r2 s13 r2 r2 s14 r2 |

16 r3 s13 r3 r3 s14 r3 |

17 r8 r8 r8 r8 r8 r8 |

18 r5 r5 r5 r5 r5 r5 |

19 r6 r6 r6 r6 r6 r6 |

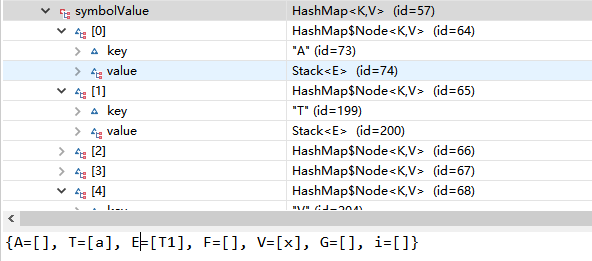
##### 7）AssignmentTranslationGrammar类：

该类存放了每条产生式对应的语义子程序，以及一张描述文法符号属性值的表。辅助语法分析的同时生成四元式。

//带有栈的符号表，因为对于复杂的算术表达式在归约时会由于归约为相同的左部而对其他值产生影响，因此将Map的对应项制成栈的结构

**private** Map<String,Stack<String>> symbolValue = **new** HashMap<String,Stack<String>>();

//语法制导翻译得到的四元式序列

**private** List<String> fourYuanShi = **new** ArrayList<String>();

##### 8）SLR1Analyzer类：

语法制导翻译的主类，即SLR(1)分析器，通过添加一个SLR(1)分析表，运行SLR(1)分析程序。

该类主要涉及二元式文件的读入问题，二元式的数据结构如下：

二元式文件通过专题1的词法分析程序得到：

其中一个测试用例为：

(1,x)

(13,=)

(1,a)

(4,+)

(1,31)

(10,\*)

(3,()

(1,c)

(5,-)

(1,d)

(3,))

(11,/)

(1,num2)

表达式为：x=a+31\*(c-d)/num2

二元式文件内容被录入到

**private** **static** List<String> *InputStream* = **new** ArrayList<String>(); //从二元式文件中拆解的符号穿输入流

二元式文件的录入和专题2一样，InputStream是一个Java List列表的一个对象，list列表是一系列的String类型的字符串，具体的操作：

br = **new** BufferedReader(**new** InputStreamReader(**new** FileInputStream(fp.getName())));

String erYuanShi = "";

**while**((erYuanShi=br.readLine())!=**null**) {

//截取符号串

*InputStream*.add(erYuanShi.substring(erYuanShi.indexOf(",") + 1, erYuanShi.lastIndexOf(")")));

}

*InputStream*.add("#"); //末尾添加#号

br 为一个文件的读入流，通过使用br.readLine()方法读入二元式文件当前行内容并返回给String类型的变量erYuanShi，然后每一行的内容类似为(1,num1)的形式，但是我们需要就是num1，所以通过erYuanShi.substring(erYuanShi.indexOf(",") + 1, erYuanShi.lastIndexOf(")"))方法将num1截取下来，放入List列表对象*InputStream*中，继续读文件，直到读取结束。这样就将二元式文件的内容读取到了List列表对象*InputStream中。*

##### 9）TranslationMain类：

本程序的入口方法类，任务是创建文法的识别活前缀的DFA对象，利用该DFA对象构造一个SLR(1)分析表对象，再构造一个SLR1Analyzer对象，并添加SLR(1)分析表，运行语法翻译制导程序。

#### 3、程序结构描述

该部分会详细介绍各个类文件中主要方法的功能以及程序设计。

##### 1）FirstAndFollow类中

要是一个描述产生式的Map<Character,String> mapGrammar和要在这个类的方法中生成的First集和Follow集，也采用Map的数据结构。buildFirst方法构造First集，buildFollow方法构造Follow集。

buildFirst方法实现概述：

while{

截取产生式左部left;

截取产生式右部,将右部通过“|”分开，得到对立的产生式数组rightnArry;

foreach rightnArry[i]:

对于每一个产生式右部挨个检查其字符cha

do

if （指针移到产生式尾部）

说明产生式中所有的非终结符号的First集都包含$,

将$加入到left的First集

break;

if (cha 属于终结符号)

将cha加入left的First集，记录一下集合大小size

else if (cha 属于非终结符号)

将cha的First集（除$外）都加入left的First集,

记录一下结合大小size

继续判断下一个字符

while(cha的First集包含$)

if(size 没有增大)

退出循环

处理下一条产生式

}

buildFollow方法实现概述：

先将“#”号加入文法开始符号的Follow集

while{

截取产生式左部left

截取产生式右部,将右部通过“|”分开，得到对立的产生式数组rightnArry;

foreach rightnArry[i]:

对每一个产生式右部，找到其第一个非终结符号firstChar,

然后找到其下一个字符secondChar

If(secondChar已经超出产生式的长度)

说明firstChar是产生式的最后一个非终结符号，

将left的First集加入到firstChar的Follow集

else if(secondChar属于终结符号)

将secondChar移入到firstChar的Follow集

else if(secondChar属于非终结符号)

将secondChar的First集元素（除$）全部移入到

firstChar的Follow集

if(secondChar的First集中有$)

将left的Follow集全部加入到FirstChar的Follow集

继续下一个产生式

判断集合有无变长，无变长退出

}

##### 2）ProductionFormula类：

对基本的String类型的产生式的封装，写了一个movePoint（Character）方法，在本产生式的基础上，根据传入的参数是否匹配本产生式中“.”后的字符，如果匹配就将“.”后移一位构造一个新的ProductionFormula对象并返回，否则返回null。

##### 3）AssignmentGrammar类：

该类中重要的是成员变量List<ProducionFormula> expendGrammar，由输入的文法构造的拓展文法，用于之后识别活前缀的DFA已经其他需要用到拓展文法的地方使用。

##### 4）StateDFA类：

相当于DFA状态转换图中的一个状态结点，有以下主要数据结构：

//状态的类别码，唯一标识一个状态的类别码

**private** **int** categoryID;

//产生式集

**private** Set<ProductionFormula> productionSet = **new** HashSet<ProductionFormula>();

//状态转换关系

**private** Map<Character,StateDFA> transform = **new** HashMap<Character,StateDFA>();

//全局的状态结点表

**public static List<StateDFA> AllStateList**

在生成DFA时，除了一个一个结点是人为添加，之后的每个状态都是由当前状态对象的createNewStateDFA方法生成，遍历所以符号结点，对产生集中的每个产生式测试movePoint方法，然后要求新状态的产生式闭包，即将“.”后面的所有产生式再加入进来，重复这个过程，直到产生式集合不在增大。之后比较该新的状态结点是否已经存在，即和AllStateList表中的元素进行比较，如果已经存在了，isExist方法会返回其状态类别码，那么新对象就复制成相同的状态，然后调用buildRelation方法建立两个状态的映射关系，如果不存在，isExist方法会返回-1，新状态会分配新的状态类别码，并将本状态加入到AllStateList表中

##### 5）DFAMananger类：

该类的构造函数中要首先添加第一个状态结点，之后，从AllStateList表中不断取出StateDFA对象，使其构造新的结点对象又插入到AllStateList表中，即广度优先搜索算法。当取完表中的最后一个StateDFA对象之后，整个DFA的状态转换图就已经实现了，并被保存在List<StateDFA> StateList表中。

##### 6）SLR1AnalysisTable类：

根据一个DFAManager对象以及产生式文法的Follow集，构建用于SLR1分析器分析的分析表。该表的数据结构为

Map<String,String> SLR1Table

存储方式为“[状态，输入字符]->动作”，通过遍历stateList表得到DFA中的所有状态装换信息进行总结，就得到了SLR1分析表,其概括SLR(1)规则如下：

**对任何输入符号a：**

**(1) 当a=b时，置ACTION［i,b］=“移进”；**

**(2) 当a∈FOLLOW(V)时，置ACTION［i,a］={按产生式V→α归约}；**

**(3) 当a∈FOLLOW(W)时，置ACTION［i,a］={按产生式W→α归约}；**

**(4) 当a不属于上述三种情况之一时，置ACTION［i,a］=“ERROR”。**

##### 7）AssignmentTranslationGrammar类：

该类是在进行SLR1分析时，进行语法制导翻译，产生四元式的主类，该类中提供了文法中各个产生式的语义子程序。类中的方法translateAction(String pro)根据传入的产生式字符串检索产生式表grammar，从而选择相应的子程序执行。该类中设计了一个语义信息表

Map<String,Stack<String>> symbolValue

该表记录了符号到属性信息的映射，即X到X.Place的映射。在实际的测试中发现，对于一个复杂一点的赋值语句，归约的过程中可能出现符号栈中可能同时存在两个相同符号的情况。为了解决这个问题，我对书中的翻译过程进行了详细的剖析，发现如果进行归约，归约前的符号属性值就无效了，因为它已经被在归约完成后被传递到归约后的符号中去了，而且翻译过程中总是先归约后进来的符号，和栈的压栈弹栈有着高度的类似，因此我创意性的将X.Place的数据结构改成栈的结构，对于原本的简单传值过程，比如“X.Place=Y.Place”修改为栈操作，即“X.push(Y.pop())”,从而解决了复杂赋值语句的翻译问题。

##### 8）SLR1Analyzer类：

该类是分析器进行分析动作的主类，设置状态栈stateStcak,符号栈symbolStack,分析开始先将状态栈和符号栈分别压入“0”和“#”。分析时将状态栈的栈顶元素和输入符号组成的字符串“状态，字符”，查询SLR1分析表，获得将要进行的动作。程序的大致流程如下所示：

#入符号栈，

0入状态栈

while{

inputSymbol=输入字符

if(inputSymbol是标识符)

将该标识压入“i”对应的属性值栈,

inputSymbol=’i’

keyState=状态栈栈顶元素

key=keyState,inputSymbol;

choice=查询SLR1表，找到key对应的动作

switch(choice)

{

case 移进：

输入符号压入符号栈

移进的状态压入状态栈

case 归约：

根据归约的序号查询产生式表，得到使用的产生式

状态栈弹栈，

符号栈弹栈

将归约符号压入符号栈栈，匹配状态栈栈顶与归约符号，得到

新的状态压入状态栈栈顶

此时要调用接口完成归约产生式的相应语法制导翻译

case 接受：

输出成功，不再执行

case 错误：

输出错误，不再执行

}

}

##### 9）TranslationMain类：

该程序的入口方法类，过程分三步：

1、根据文法创建识别活前缀的DFA

2、根据创建的DFA构建SLR（1）分析表

3、根据SLR(1)分析表构建SLR（1）分析器，并运行分析程序。

### 三、程序测试

#### 3.1、正确样例

本程序设计了两个测试文件：

Zhuanti5\_1.tys:

(1,x)

(13,=)

(1,a)

(4,+)

(1,31)

(10,\*)

(3,()

(1,c)

(5,-)

(1,d)

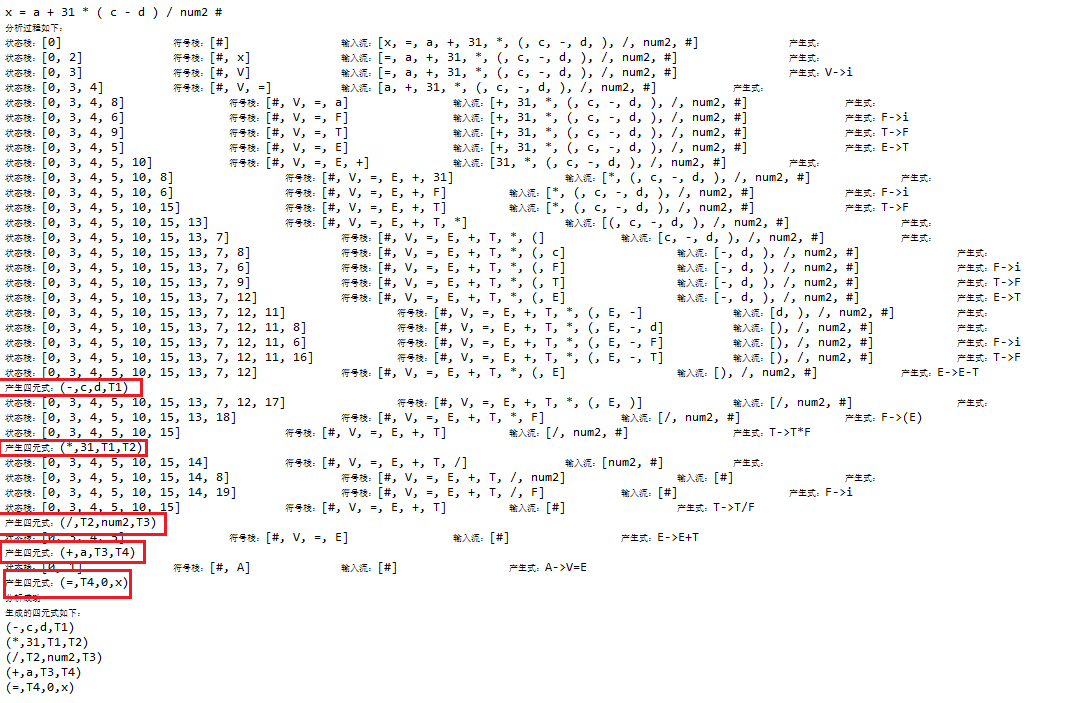
(3,))

(11,/)

(1,num2)

原本的赋值语句表达式为x = a + 31 \* ( c - d ) / num2

程序运行结果如下：



#### 3.2、错误样例

##### 3.2.1缺少 =

将二元式文件的中的等号去掉：

(1,x)

(1,a)

(4,+)

(1,31)

(10,\*)

(3,()

(1,c)

(5,-)

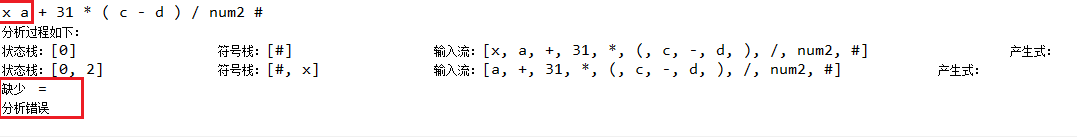
(1,d)

(3,))

(11,/)

(1,num2)

运行得到结果和提示：



##### 3.2.1缺少操作符

将正确的二元式文件某个操作符去掉：

(1,x)

(13,=)

(1,a)

(4,+)

(1,31)

(10,\*)

(3,()

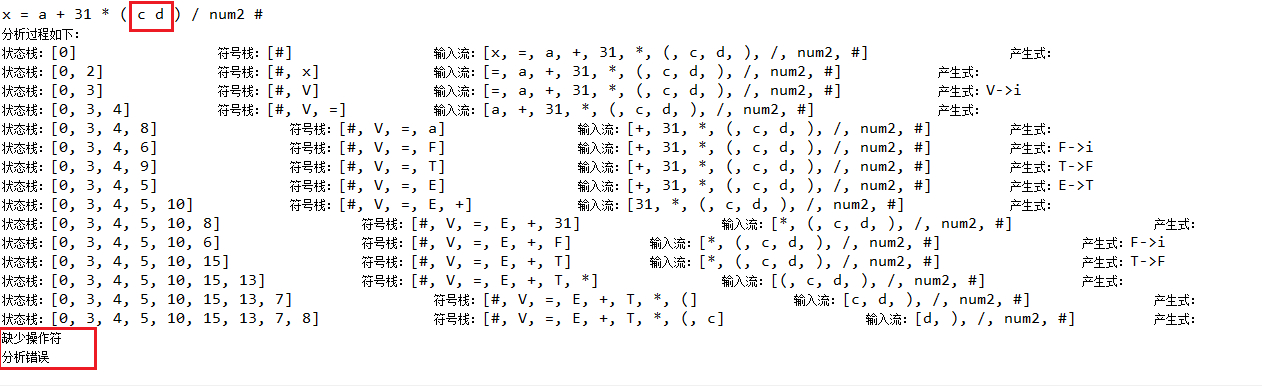
(1,c)

(1,d)

(3,))

(11,/)

(1,num2)

运行得到结果和提示：