**课程编号：A0801041040**

**编译方法实验报告**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **姓名** | **柳成林** | **学号** | | **20206722** |
| **班级** | **软件2001** | **指导教师** | | **刘洪娟** |
| **实验名称** | **扫描器设计及中间代码生成器设计** | | | |
| **开设学期** | **2022-2023秋季学期** | | | |
| **开设时间** | **第5周和第8周** | | | |
| **报告日期** | **2022年9月29日** | | | |
| **评定成绩** |  | | **评定人** | **刘洪娟** |
| **评定日期** | **2022年10月30日** |

**东北大学软件学院**

实验一：扫描器的设计

1. **实验目的**

通过学习有限自动机原理、词法分析器原理，结合课堂知识与自己的编程语言知识，阅读所给的参考程序代码，实现一个扫描器（词法分析程序）。

1. **实验内容**

分析、理解、学习实验指导书中给出的参考代码，设计扫描器的有限自动机，设计翻译、生成Token的算法，对完成的代码进行调试，保证能够正确解析并通过，输入为源程序文件，输出为解析出的关键字表、解析出的界符表、符号表和常数表。

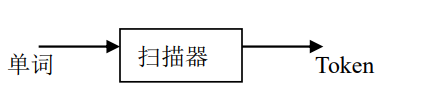
1. **实验原理及基本步骤**

实验原理：有限自动机原理。

有限自动机对应了正规文法的表述，通常用来识别正规语言，在词法分析时特别有效，一个设计良好的有限自动机应该只存在一个开始态，转换边尽量少，一般需要进行确定化和最小化。

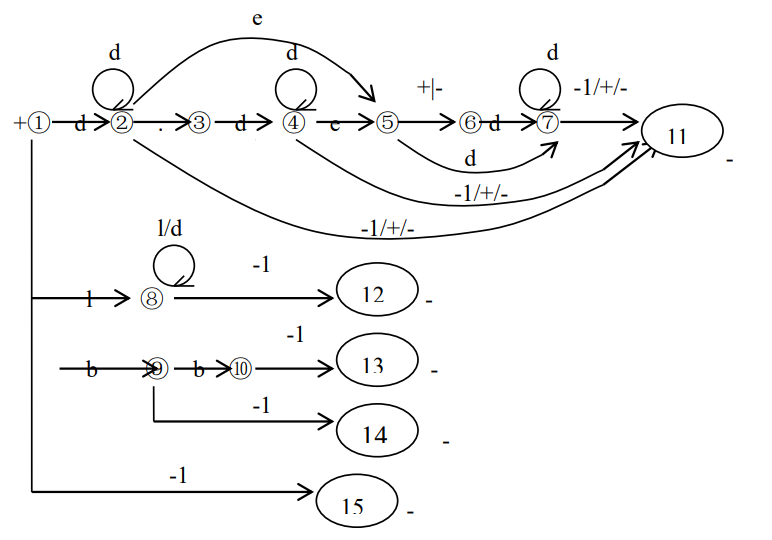
实验基本步骤：

1. 体会并设计系统输入输出模式；



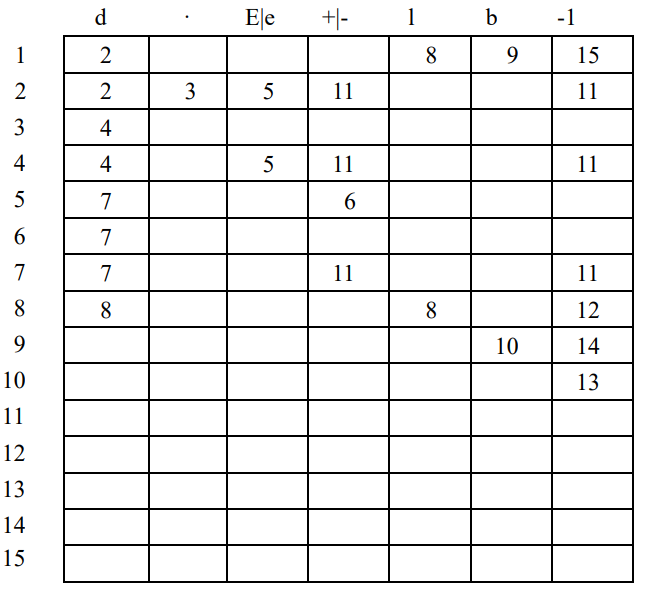
1. 了解需实现的有限自动机的状态转换图、状态转换矩阵；

画出有限自动机的状态转换图：



其中：d为数字，l为字母，b为界符，-1代表其它符号（如在状态8处遇到了非字母或数字的其它符号，会变换到状态12）。

写出状态转换矩阵：



1. 定义关键字表和界符表；

|  |  |
| --- | --- |
| Program | ; |
| Begin | : |
| End | ( |
| Var | ) |
| While | , |
| Do | := |
| Repeat | + |
| Until | - |
| For | \* |
| To | / |
| If | > |
| Then | >= |
| Else | == |
|  | < |
|  | <= |

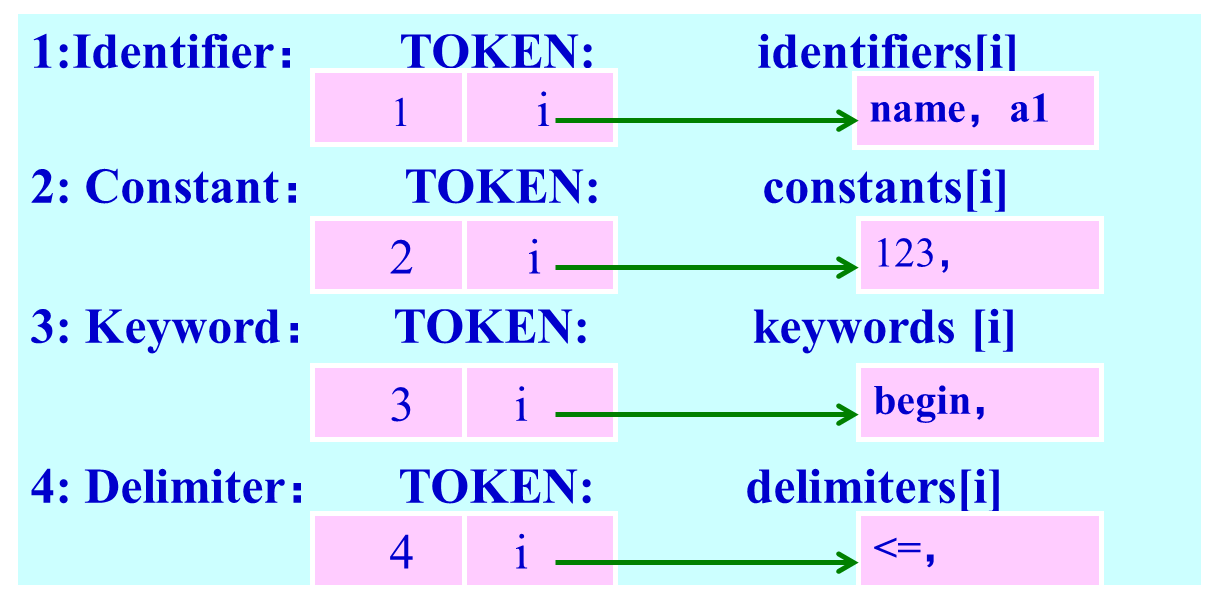
1. 根据给出的关键词、界符等信息设计数据结构；
2. 阅读并运行给出的C语言代码，**发现代码中的错误**以及不合理之处；
3. 参考给出的程序代码，**使用Java语言**完成代码的编写，如果存在不合理的数据结构，进行数据结构的替换；
4. 对Java实现的代码进行进一步**改进创新**。
5. 调试修改后的代码，输入多种源文件测试并修改，记录输出结果情况，总结反思。
6. **数据结构设计**

**Token类：**

这里对token的属性进行了重新设定，没有仿照实例代码。

包含两个int类型的属性，code代表token的类型，有1、2、3、4四种类型，分别代表标识符、常量、关键词和界符；value代表token对应的表中值的索引。

如下图所示：左边为4种类型的token结构，右边为对应的字符串数组。



其中，identifiers、constants、keywords和delimiters分别代表存储标识符、常量、关键词和界符的字符串数组，所有token也使用数组存储。这里采用将关键词和界符分开存储的方法（实例代码中存在了一个数组中），并分别对应不同的token，这样定义更加合理，也使输出的结果更加直观明了。

**ColMap类：**

参考实例代码中的map结构体，因为字符到状态转换矩阵需进行映射。包含两个属性：字符串str代表所有可能的字符连接而成的字符串，col代表列数。

同时，每一条转换路线所需要的字符是不一样的，而且他们的-1代表的其他字符也是不一样的，需要定义3种不同的map数组（这里采用List存储）。

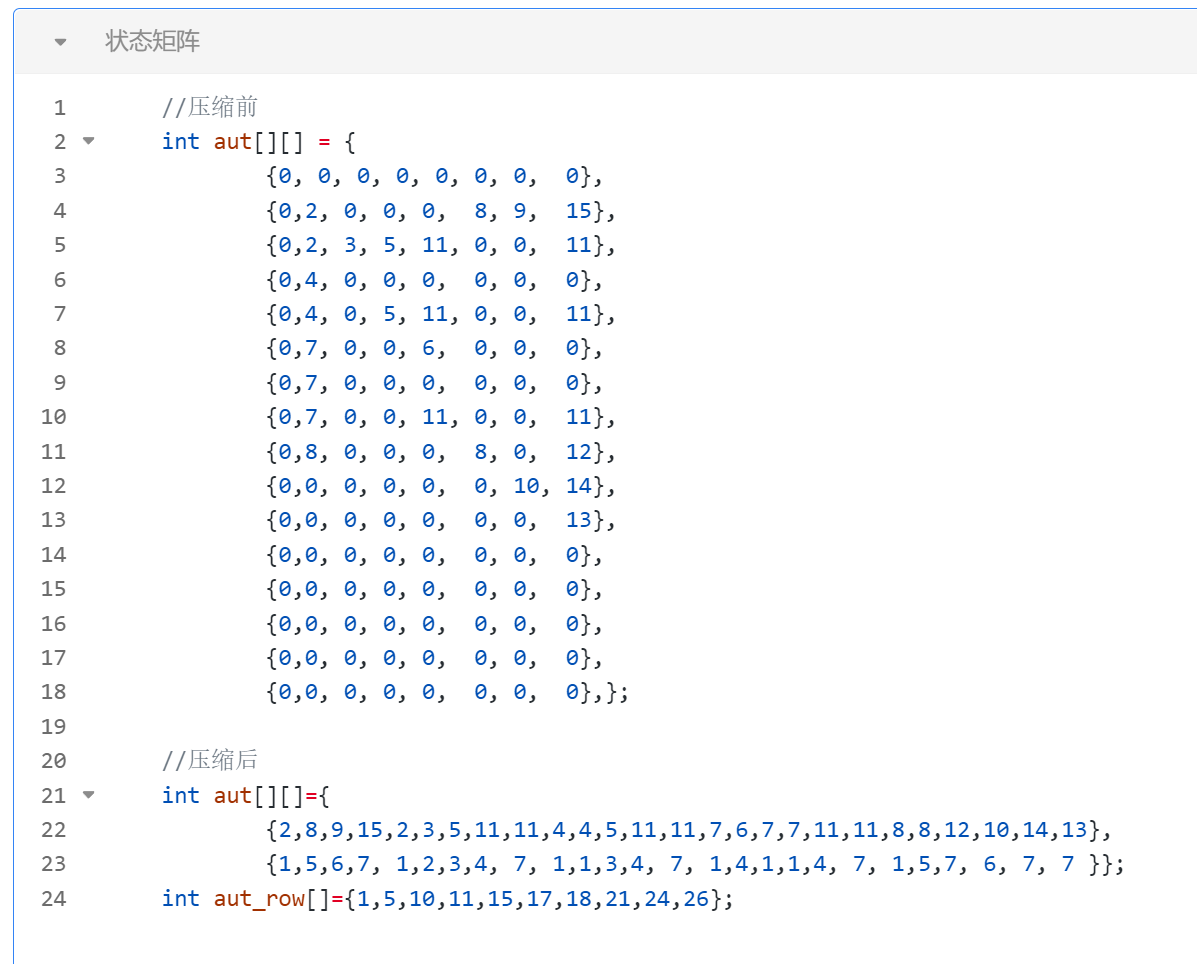


**状态转换矩阵：**

首先采用二维数组的形式存储状态转换矩阵，由于状态命名方式由1开始，故这里也采用从1开始索引的方式。但发现完成一个状态矩阵的定义需要16\*8的二维数组，且其中较大空间被0所占，明显是一个稀疏矩阵。

作为改进，我**采用了压缩矩阵的方式**进行存储，压缩存储稀疏矩阵的方法是：只存储矩阵中的非0元素，与前面的存储方法不同，稀疏矩阵非0元素的存储需同时存储该元素所在矩阵中的行标和列标。

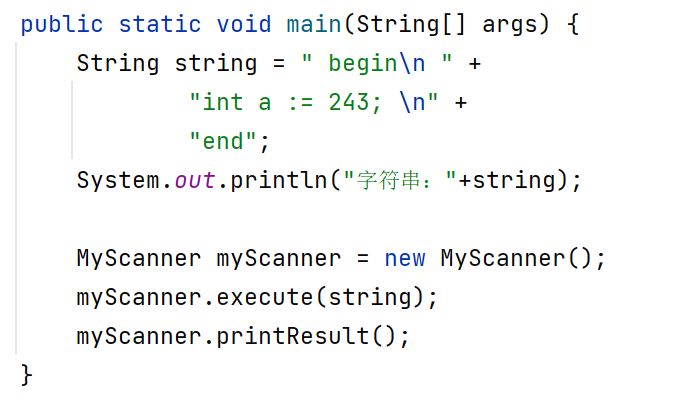
对该矩阵进行按行存取，aut矩阵第一行为所有非0元素按行排列，，第二行为所有元素在原来矩阵中的列标。aut\_row数组存储每一行第一个非0元素在aut矩阵中的位置。



上图为压缩矩阵前与压缩矩阵后，同时对压缩的矩阵需要重写state的判断方法，具体的实现在类MyScanner1中。

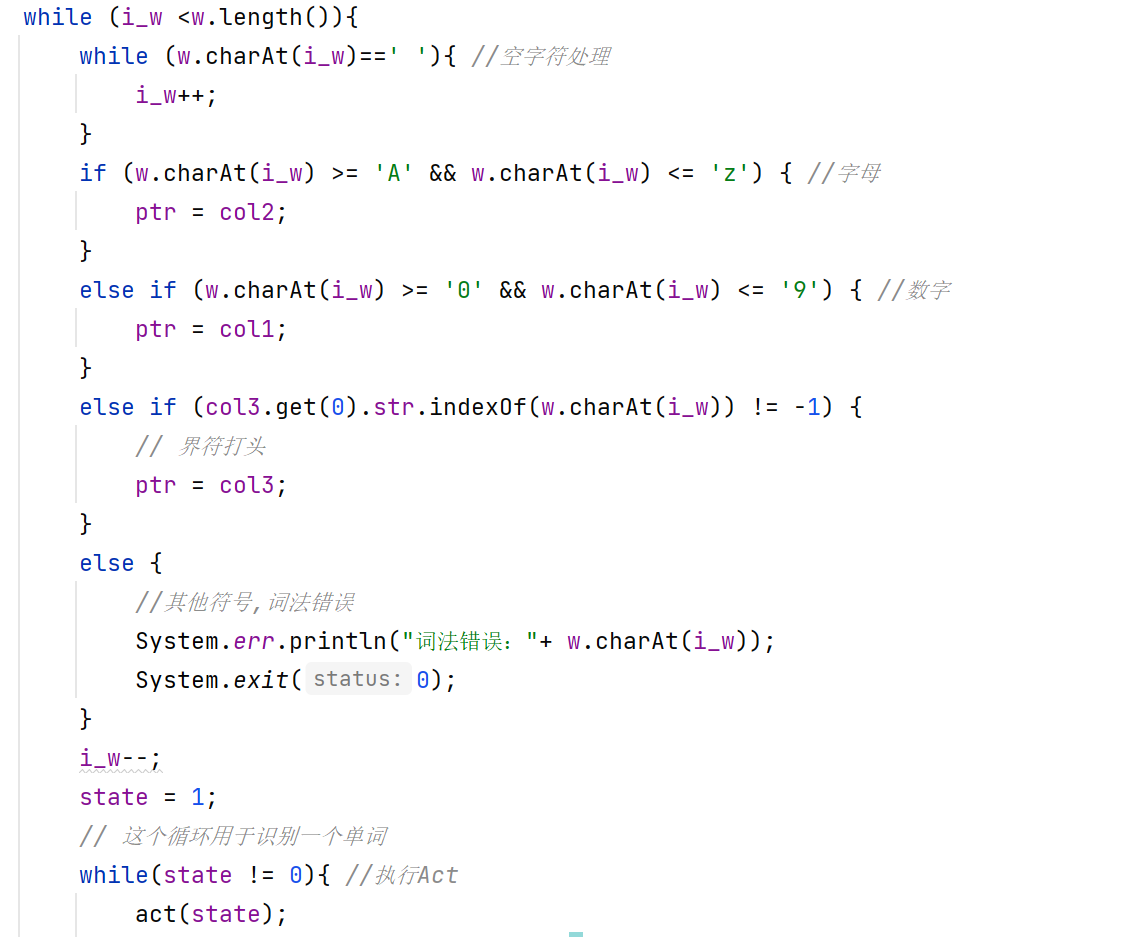
1. **关键代码分析（带注释）及运行结果**

**Main函数主要运行流程：**



* + - 1. 定义需要扫描的字符串，这里为了方便，直接采用字符串定义，也可以选择文件读取。
      2. 创建MyScanner扫描器对象，扫描器对象封装了一些状态属性和方法函数，可以对传入的字符串进行词法分析。
      3. 扫描器执行execute方法，对字符串进行词法分析。
      4. 扫描器执行print方法，输出结果。

**Execute方法（部分代码）：**

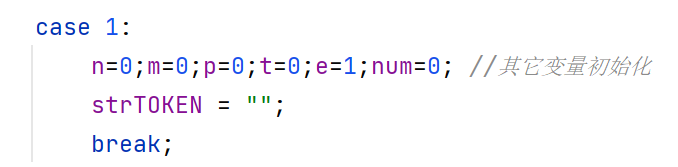


包含了程序主要的执行逻辑。具体的执行逻辑是：首先将字符串/字符串的一部分读取到缓冲区中，外层while循环读取缓冲区中的每一个字符；然后先对空字符进行过滤，按字母>数字>界符>其他符号的顺序进行读取，对于每种不同的情况，改变对应的map列表。之后对读取到的单词进行拼接等处理，执行Act方法并生成相应的token。

**Act方法（部分代码）：**



Act方法的执行逻辑非常简单，就是针对不同的state进行不同的操作。要注意的是，当state为1时，必须进行初始化操作。因为每一个token的识别都从state=1开始，必须清除上一个token识别产生的临时值。



除此之外，我还对关键词、界符等识别逻辑进行了修改（实例代码有一定的不合理和错误）。Case13判断识别到的界符是单界符还是双界符，此时识别到的token字符串中应当包含两个界符字符，长度为2。当reserveDel返回的值为-1时，说明界符表中不存在该双界符，所以可以认为字符串中的两个字符都为单界符，此时需要将i进行减1，回退一个字符。Case14是针对识别到单个界符字符的情况，此时还需要加上一个判断逻辑（该界符是否在表中），即错误处理。



**其他方法**

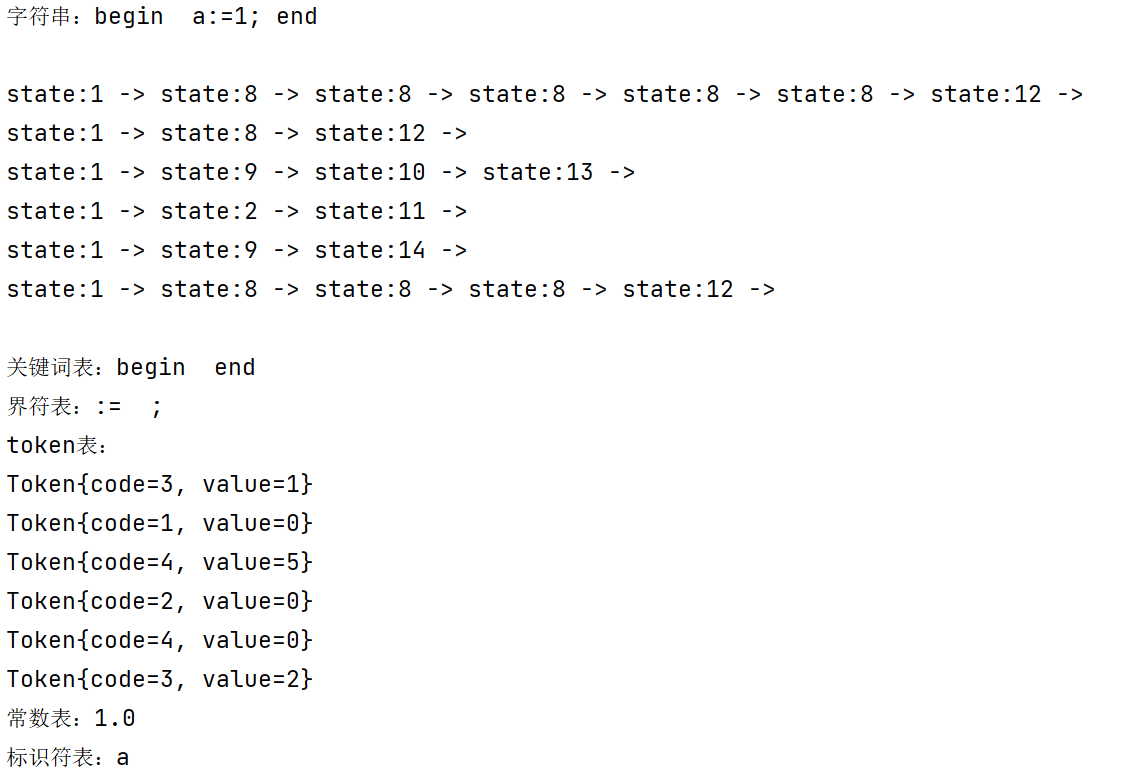
|  |  |  |  |
| --- | --- | --- | --- |
| **方法名** | **传入参赛** | **返回值** | **作用** |
| reserveKey | String str | int value | 判断关键词是否存在，返回关键词在关键词表中的位置 |
| reserveDel | String str | int value | 判断界符是否存在，返回界符在界符表中的位置 |
| insertIdentifier | String str | int value | 插入新的标识符，返回标识符在标识符表中的位置 |
| insertConstant | String str | int value | 插入新的常量，返回常量在常量表中的位置 |

以上方法在Act方法中调用。

**运行情况**

**（1）字符串：begin a:=1; end**

输出state的转换过程、每一个表的内容。



**（2）字符串：**

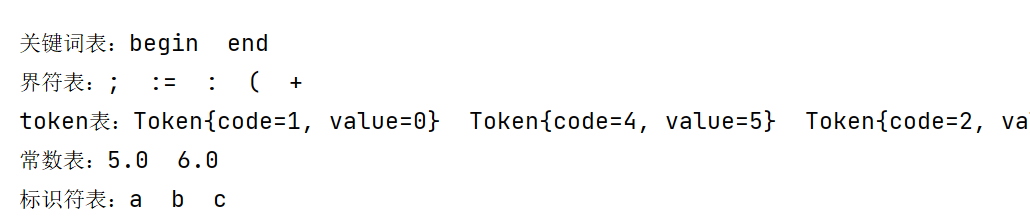
**a:=5;**

**b:=6;**

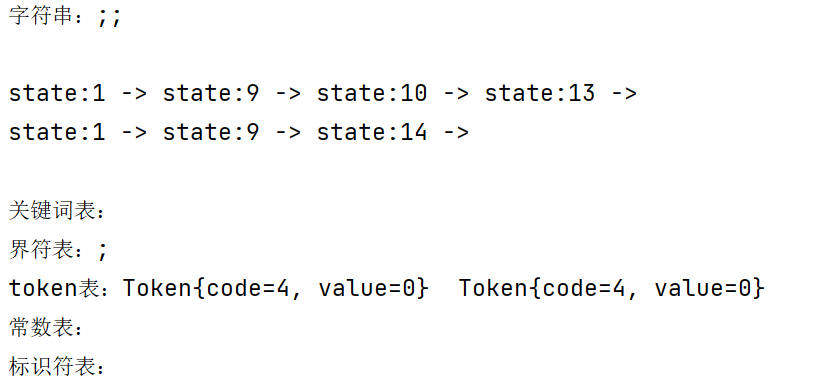
**begin**

**c:=a+b;**

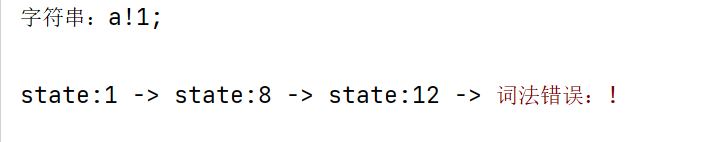
**end;**



**（3）字符串： ;;** (两个单界符)



（4）字符串： **a!1;**  (词法错误，不存在！)



由运行结果可以得出，上述情况的词法分析都正确。

1. **总结与分析**

在本次实验中我通过java实现了简单扫描器，java相比于c语言，在某些变量的定义上比较简单，比如说直接使用字符串String，相比于c语言中的字符数组更加方便。同时，我还对实例代码中不合理的地方进行了分析，并加入了自己的创新，例如重写判断逻辑、压缩矩阵存储等。具体修改的地方有：

* + - 1. 将常数的类型改成double；
      2. 将关键词表和界符表分开定义；
      3. 对不同类型token的格式重新定义；
      4. 修改主循环的部分判断逻辑，例如修改字母的识别范围（大小写），修改字符的判断顺序；
      5. 修改一些错误处理的判断，例如case14中非界符的判断。

通过本实验深刻地了解到有限自动机原理及词法分析的过程，通过合理的方法除去无用的字符序列，根据状态矩阵完成状态的转换。

1. **思考题**

**（1）扫描器的任务是什么？**

扫描器又名词法分析器，是进行词法分析的程序或者函数，它的任务主要是负责将字符流转化为记号流，即通过识别把字符序列转化为Token，其中还包括进行词法检查，删除回车等其他非实质性字符、删除注释，识别标识符，报告所发现的错误等内容。

**（2）扫描器、识别器、翻译器三者之间的关系是怎样的？**

扫描器是词法分析器，扫描器的有限自动机是识别器，对识别出的进行翻译、生成Token的算法是翻译器。可以看出，扫描器是最底层的部分，识别器的实现依赖于扫描器，翻译器的实现依赖于识别器。

**（3）为什么说有限自动机是词法分析的基础？**

词法分析要通过对源程序读取并分析，共分为两个阶段：扫描阶段、词法分析阶段。扫描阶段中扫描器通常基于有限自动机，扫描到的字符可以完全枚举，第二阶段的词法分析需要转化为Token，此时仍会用到第一阶段的扫描器，所以两个阶段都与有限自动机密切相关，有限自动机是词法分析的基础。

实验二：中间代码生成器设计

1. **实验目的**

熟悉算术表达式得语法分析与中间代码生成原理，实现算术表达式的中间代码生成器。

1. **实验内容**

在本实验中我使用Java语言实现LL(1)和LR()两种分析方法，并在实验时进行了两种方法的对比和分析，理解了语法分析的过程。

在这里我主要介绍LR()分析法的实现步骤：

* 1. 拓展文法
  2. 构建属性翻译文法
  3. 构造句柄分析器
  4. 构建LR()分析表
  5. 构建LR()控制器
  6. 拓展LR()分析器（r(j)中加入属性翻译动作）
  7. 构建语法制导翻译器
  8. 处理输入的表达式，生成对应四元式序列和分析过程

1. **实验原理及基本步骤**

我在实验中采用自顶向下的LL(1)分析法和自底向上的LR()文法分析方法分别实现了简单的中间代码生成器。

如果使用LL(1)，我们要保证文法不能有左递归。如果有左递归，需要进行文法变换；之后还要求select集判断该文法是否可以使用LL(1)分析法，所以LL(1)分析法对文法的限制很高。而LR()分析法相比LL(1)则对文法的限制很少，而且分析速度快，但LR()分析法需要构造较为复杂的句柄识别器，两者各有优缺。

在这里我主要介绍LR()分析法的原理和步骤：

G(E)： E 🡪 E + T | T

T 🡪 T \*F | F

F 🡪 i | (E)

拓展文法：

G(Z)： Z 🡪 E

E 🡪 E + T | T

T 🡪 T \*F | F

F 🡪 i | (E)

设计属性翻译文法：

G(Z)： Z 🡪 E

E 🡪 E + T{GEQ(+)} ① | T ②

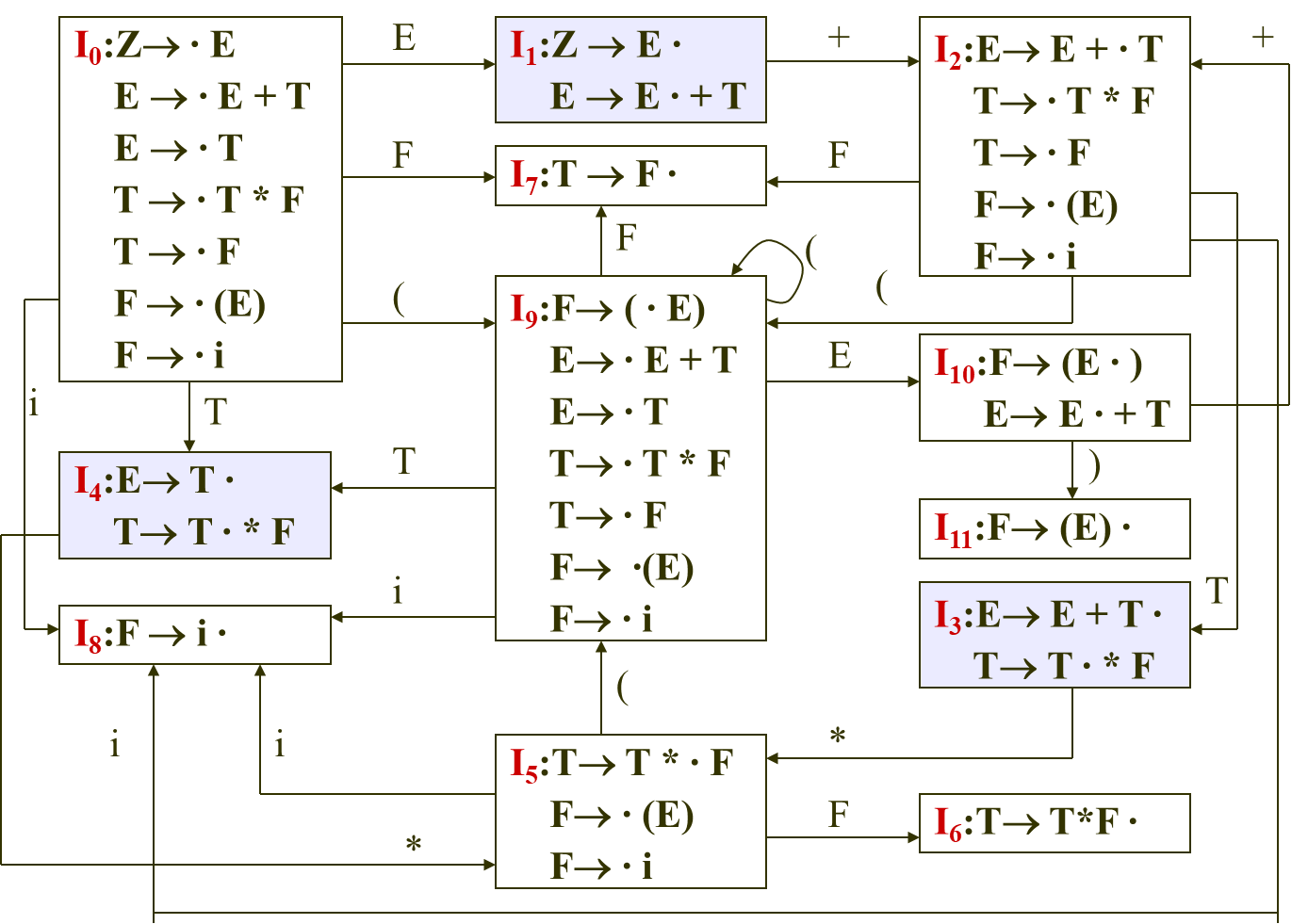
T 🡪 T \* F{GEQ(\*)} ③ | F ④

F 🡪 i{PUSH(i)} ⑥ | ( E ) ⑤

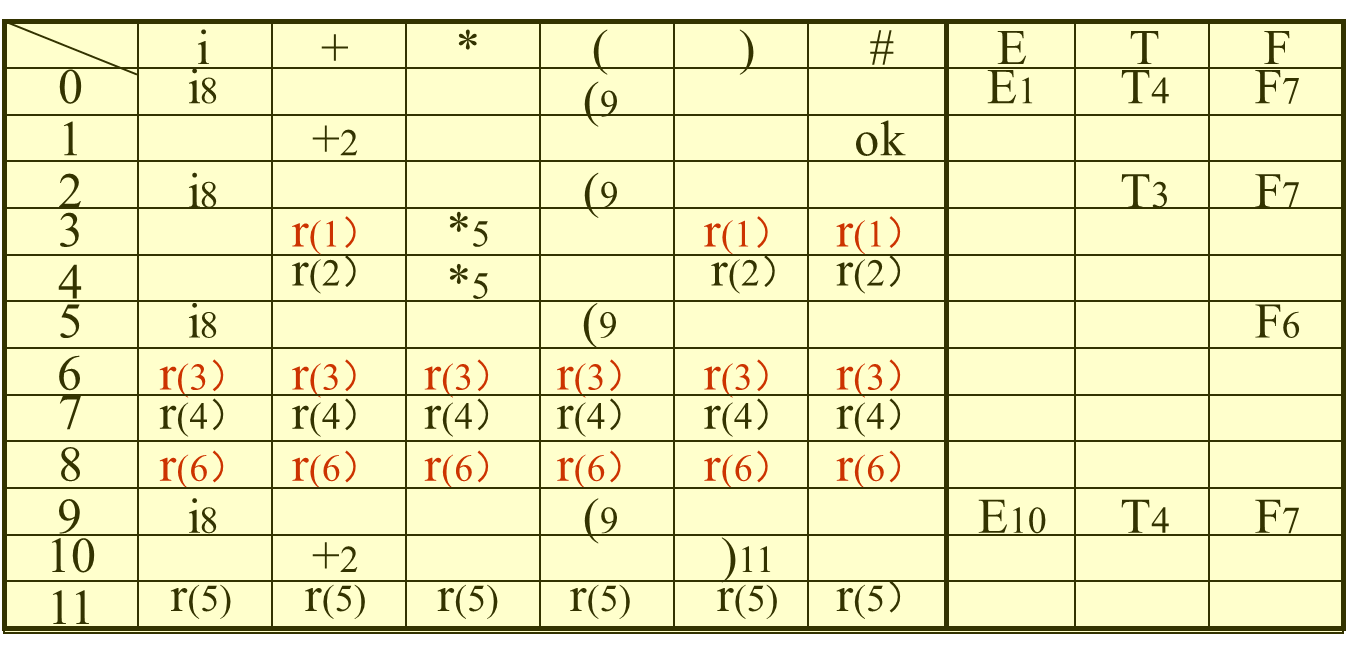
在LR()分析法中动作符号只能位于产生式的最右端，在这里我设计了+和\*两种算符。

构造句柄分析器和LR()分析表：

通过闭包函数和状态转移函数可以求出句柄识别器，进一步得到LR()分析表，过程略。



句柄识别器



LR()分析表

这里直接采用了课件中构建好的分析表，在后续代码中将这一过程省略。

LR()分析器的扩展：

※ LR()分析表中的 r(i) 执行下述两种操作：

① 首先执行动作符号(翻译函数)；

② 然后执行归约操作(按产生式 i 归约)。

最后，结合属性翻译文法，完成语法制导翻译器的构造。

**分析过程的实现逻辑：**

基本逻辑：在对输入串的一次从左到右的扫描过程中，语法分析器将输入内容移到栈的顶端，直到他可以对栈顶的一个文法符号进行规约为止，然后将其规约为某个产生式的左部，并逐渐重复这个循环。

在我们已知LR分析表的基础上，在代码中，我们首先定义以下数据结构：

语法栈：SYN[n] ;

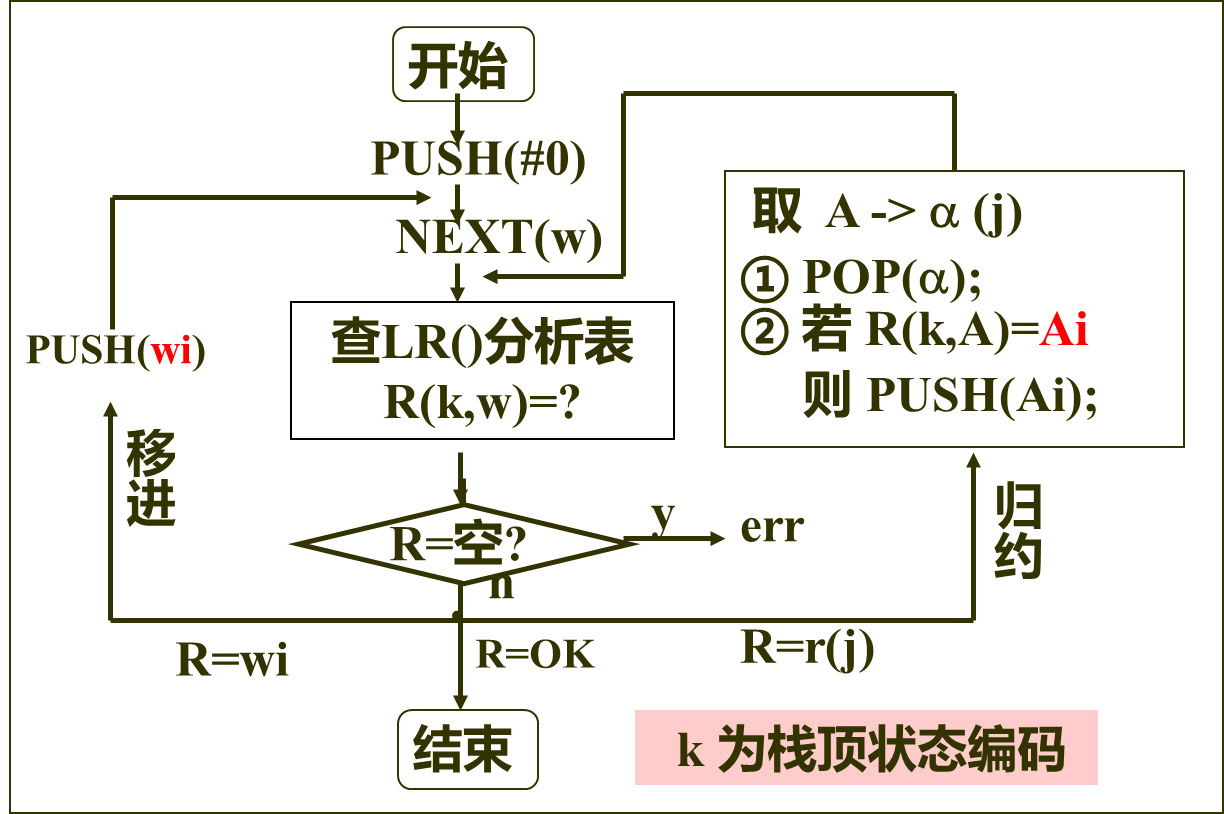
语义栈：SEM[m];

四元式区：QT[q];

在主要代码逻辑中，我们首先在SYN中压入#0，表示初始状态为0状态，接下来读取一个单词w（需要事先通过词法分析获得单词），根据当前状态和这个单词的类别查找LR分析表，根据表格返回的内容，可以产生4种结果。

* + - 1. 查找表格结果为空，出现语法错误
      2. R=OK，说明分析结束
      3. R=r(j)，执行动作符号和规约过程，然后对规约结果进一步查表
      4. R=(wi)，移进SYN，并读取下一个单词w，然后查表

上述过程会一直持续，直到返回OK或者出现错误，其具体的逻辑也可以用下图表示。



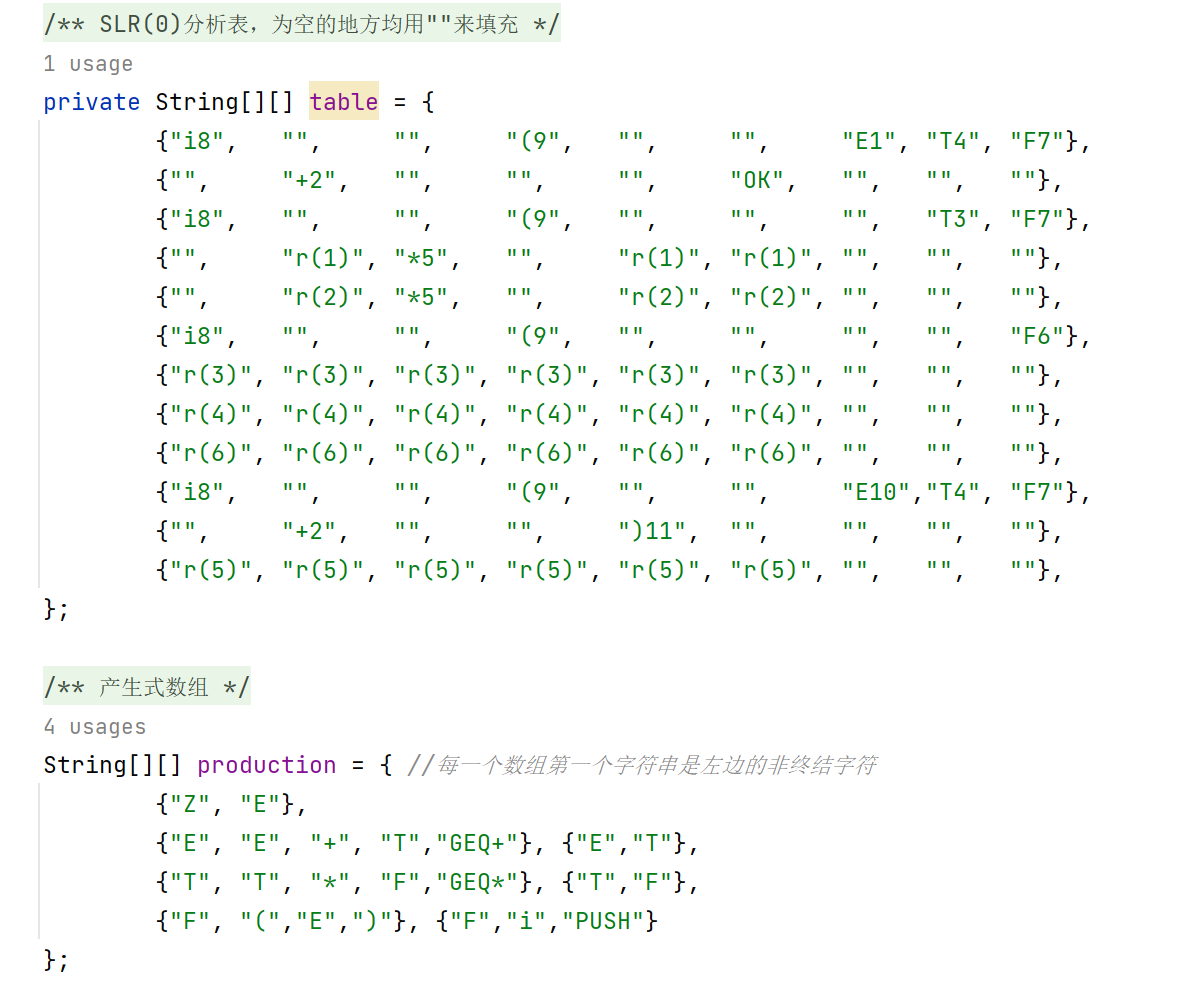
LR()控制器流程

其中比较重要的点为数据结构的设计、查表方法、规约方法和整体的循环方法。

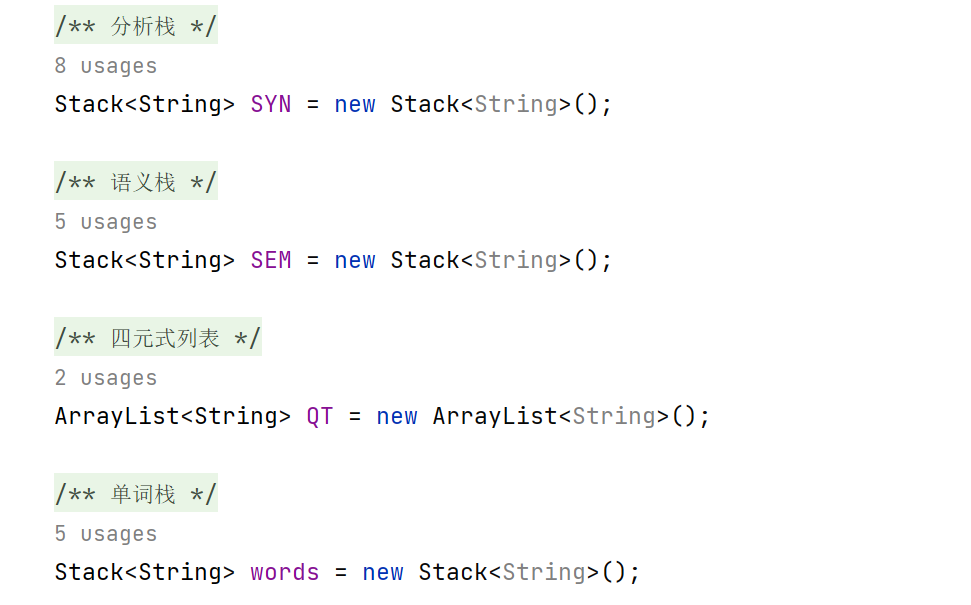
1. **数据结构设计**

SLR(0)分析表[table]：采用二维数组定义，同时用一个一维数组保存列标。

产生式表[production]：也采用二维数组，每一行是一个产生式，每一行的每一个字符串代表产生式的一个终结符号或者非终结符号，每一行的第一个字符串为产生式左部，从第二个开始为产生式右部，如果有动作符号，则放在最右边。



其他数据结构：



分析栈（语法栈）：SYN

语义栈：SEM

四元式列表：QT

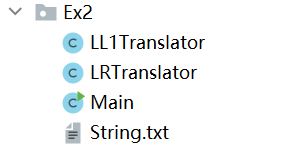
单词栈：words（词法分析得到的单词列表）

以上变量都作为类的属性，其他变量略。

在LL(1)分析器中也采用类似的设计：只是产生式的左部并不需要保存，这里直接保存右部。



**类的设计：**



LL1Translator和LRTranslator分别代表两种分析器的封装类，Main函数中生成他们的对象并执行分析，String代表要处理的表达式。两个分析器类的结构大致相同，只是主要函数的执行逻辑和判断条件不同。

1. **关键代码分析（带注释）及运行结果**

**LR主程序控制函数**：根据分析表返回的结果有四个条件判断

|  |
| --- |
| /\*\*  \* 主程序控制函数  \*  \*/  public void analysis() {  proc(string);//将字符串转为单词序列  //首先将#0压栈  SYN.push("#0");  //得到第一个单词  w = words.pop();  while(!words.isEmpty()||!SYN.isEmpty()) {  String x = SYN.peek(); // 得到栈顶字符串  currentState = Integer.parseInt(x.substring(1)); //取栈顶状态数字  printSYN(); //输出一列结果  String result = findTable(currentState,w); //查表得到结果  if (result.equals("")) {  //出错  err();  } else if (result.equals("OK")) {  //完成，跳出while  break;  } else if (result.startsWith("r(")) {  //R=r(j) 规约  reduction(result);  } else {  // R=wi 状态转换  SYN.push(result); //移进语法栈  lastW = w; //记录当前字符  w = words.pop();//读下一个字符  }  }//while  if (w.equals("#")) {  System.out.println("四元式：");  for (String string : QT) {  System.out.println(string);  }  } else {  err();  }  } |

**LR规约函数reduction**：执行动作符号和规约操作

|  |
| --- |
| /\*\*  \* 规约  \* @param result 表里返回的结果  \*/  public void reduction(String result){  //result为 r(i) 格式  int index = Integer.parseInt(result.split("\\(")[1].split("\\)")[0]); //得到表达式的index  String left = production[index][0]; //产生式最左边的非终结符  String right = production[index][production[index].length-1]; //产生式最右边的符号，要判断是不是动作符号  int flag = 0;  //先执行动作符号，如果有，判断是哪种，并执行  if(right.startsWith("GEQ")||right.equals("PUSH")){  flag = 1;  if (right.startsWith("GEQ")) {  GEQ(right.substring(3)); //GEQ方法，需要传入生成式的算符  }  else if (right.equals("PUSH")) {  SEM.push(String.valueOf(lastW)); //压入上一个单词到语义栈  }  }  //接下来执行规约操作，如果有动作符号flag=1  for (int i = 1; i < production[index].length-flag; i++) { //对产生式右侧的字符串进行循环  SYN.pop();  }  //规约之后，需要计算新的栈顶状态  String x = SYN.peek(); // 得到栈顶字符串  currentState = Integer.parseInt(x.substring(1)); //取栈顶状态数字  result = findTable(currentState,left);  SYN.push(result); //当前状态遇到规约后的非终结符发生的状态变化  } |

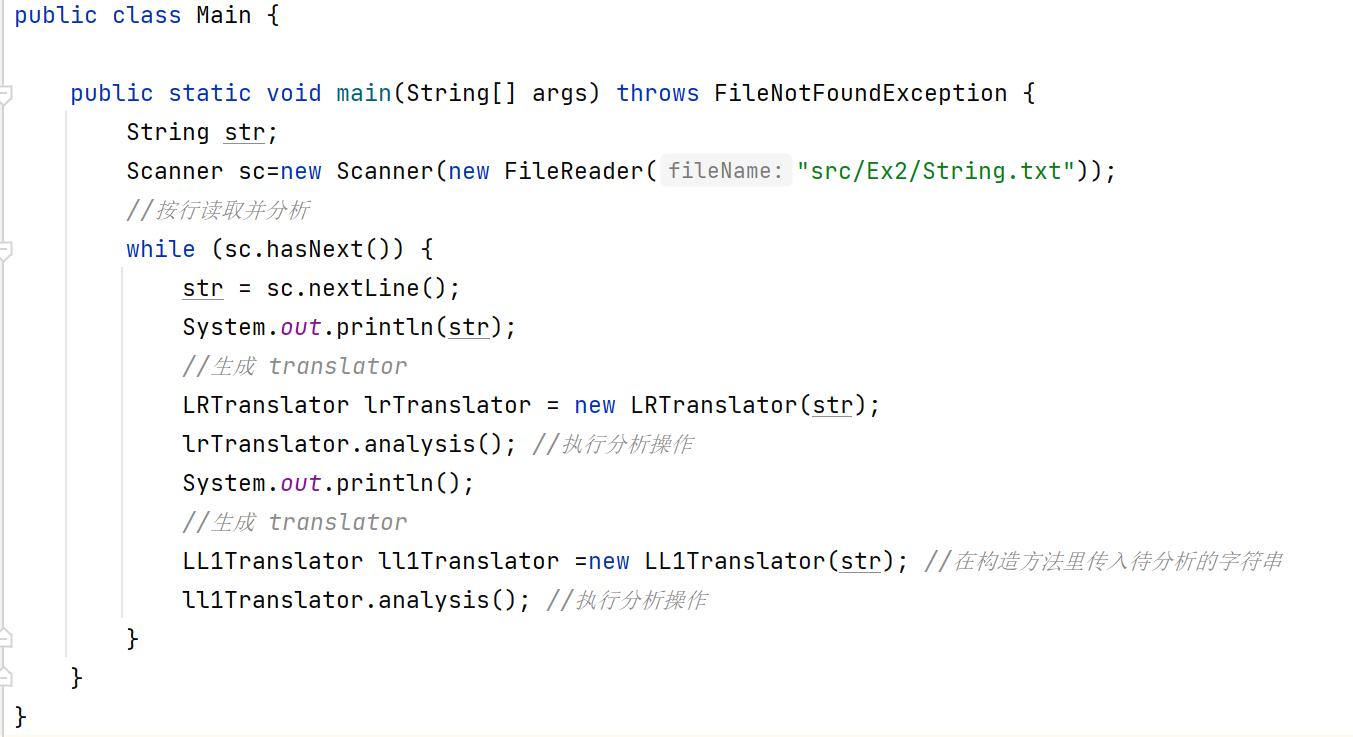
**LR查表函数findTable**：根据状态和遇到的字符返回表的结果，通过列名找到下标

|  |
| --- |
| /\*\*  \*  \* @param state 状态  \* @param w 遇到的字符  \* @return  \*/  private String findTable(int state, String w) {  // colName = {"i", "+", "\*", "(", ")", "#", "E", "T", "F"};  int row = state, column = -1;  if ((w.charAt(0) >= '0' && w.charAt(0) <= '9') ||  (w.charAt(0) >= 'a' && w.charAt(0) <= 'z')) {  column = 0;//字母和数字  }else {  //循环找列标  for (int i = 0; i < colName.length; i++) {  if (colName[i] .equals(w) ) {  column = i ;  }  }  }  if(column == -1){  err();  }  return table[row][column]; //返回查询到的状态  } |

**Main函数：**

读取文件中的字符串，按行分析；

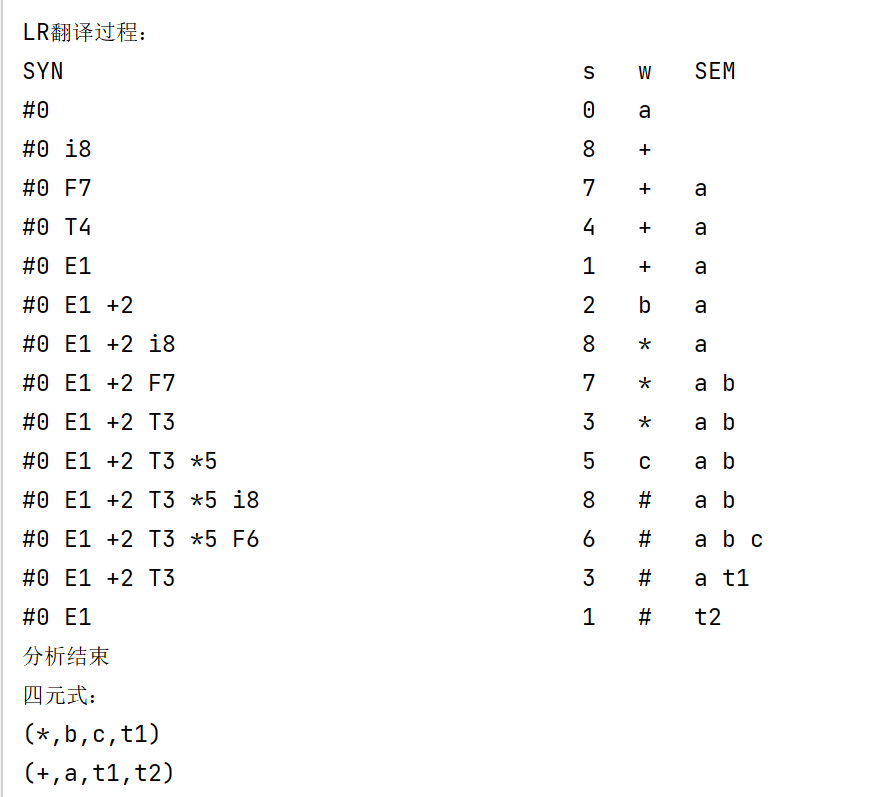
生成分析器的实例，在构造方法中传入字符串，执行analysis方法。



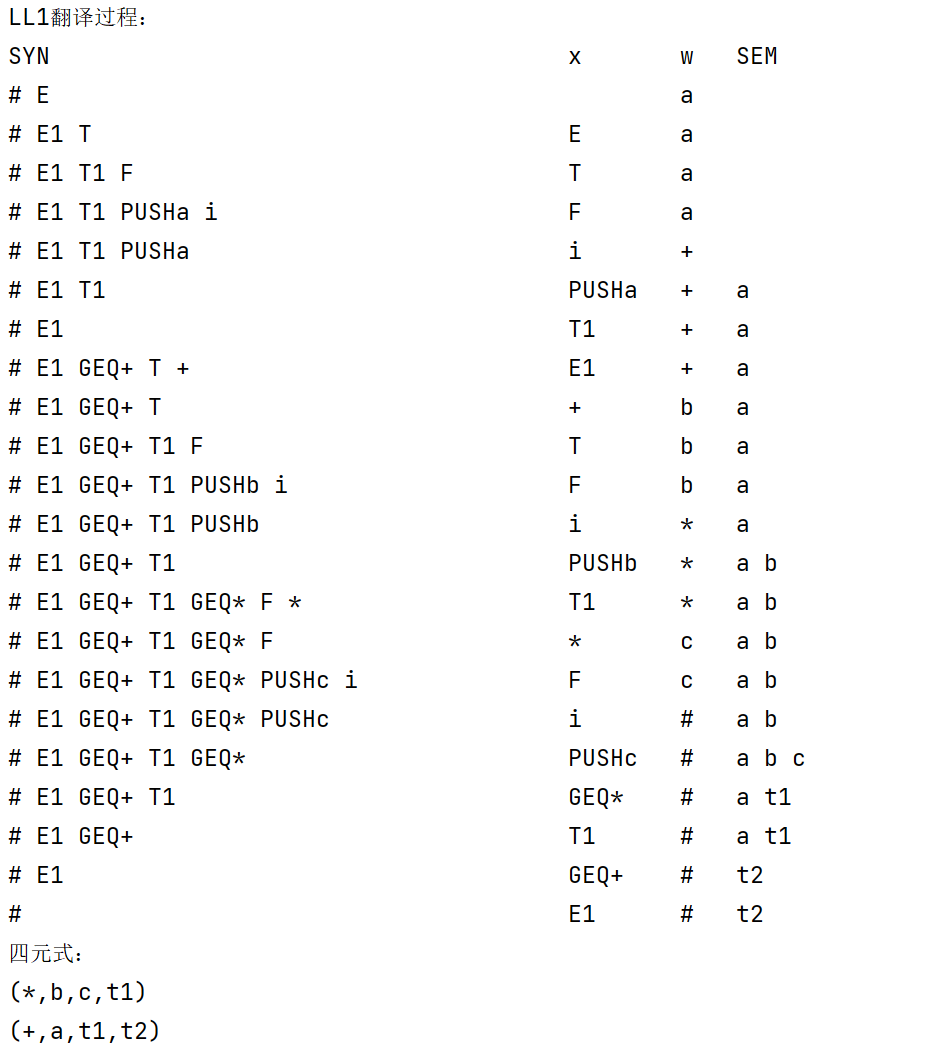
**运行结果：**

**1. a+b\*c# 分析过程：**

LR():

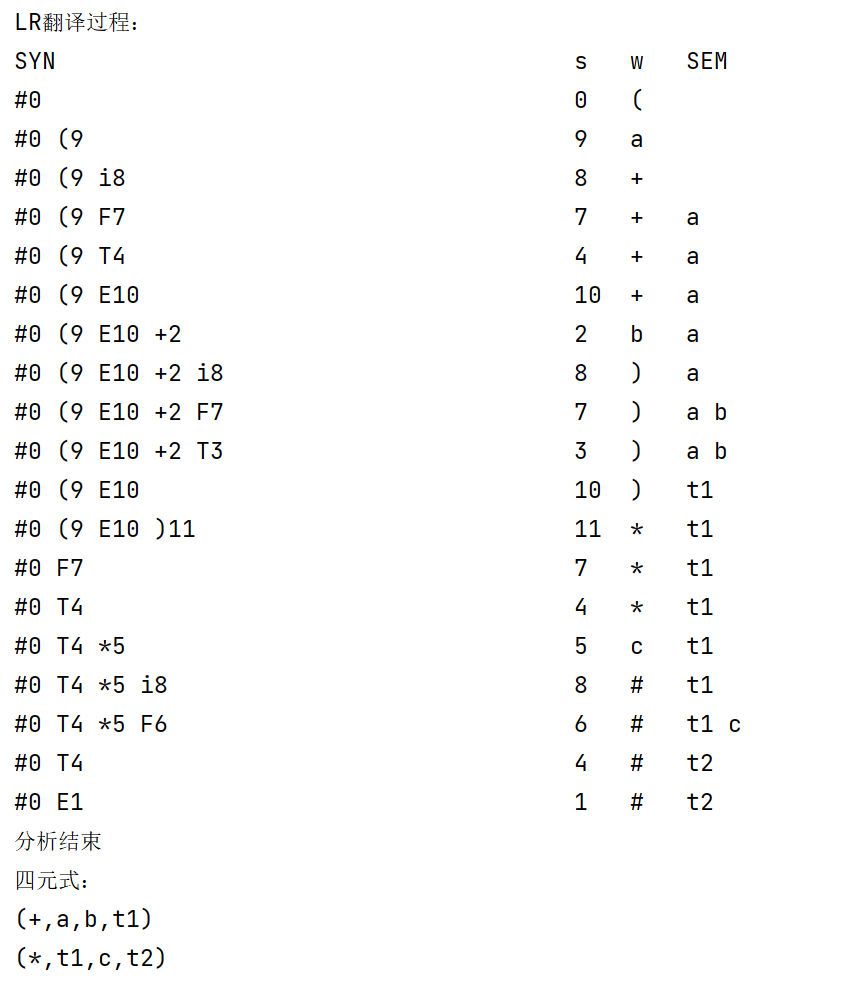


LL(1)：

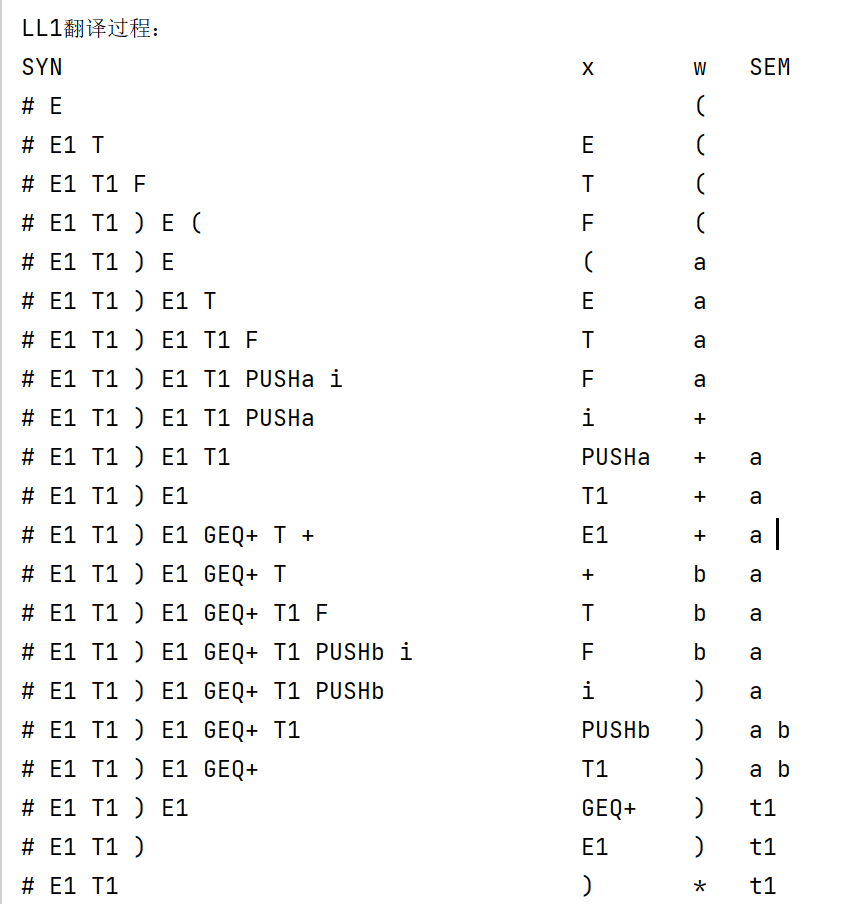


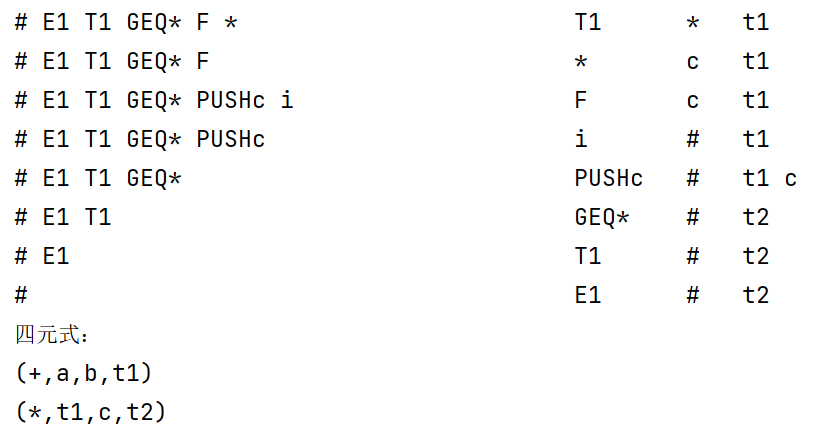
**2. (a+b)\*c# 分析过程：**

LR():



LL(1):



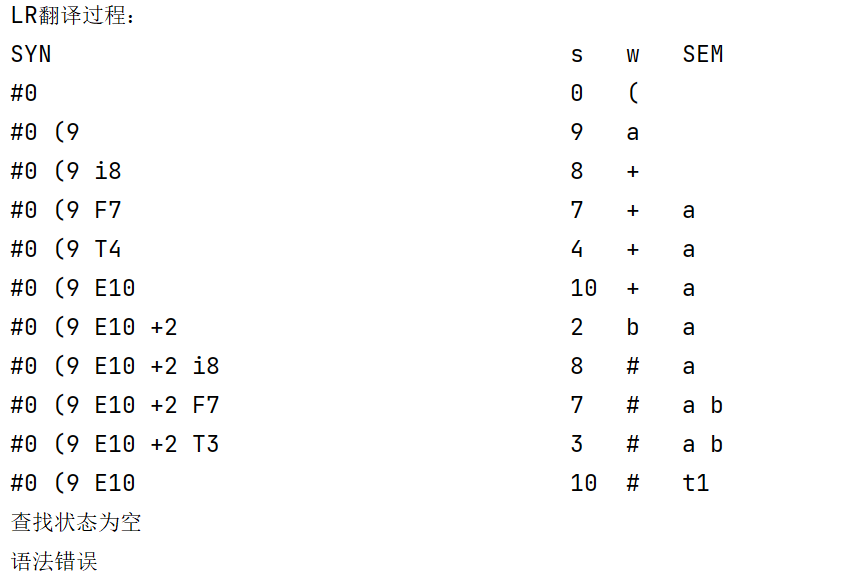


通过分析四元式列表和分析过程表可以得出，两次执行都是正确的。

通过对比两种方法可以得出，LL(1)的分析过程明显比LR()的分析过程长，由于每执行一次while方法就会打印出一行结果，也就是说分析表越长执行while循环的次数就越多，这可以说明LR()的分析速度的确优于LL(1)。

**3. (a+b# 分析：**

明显有语法错误：没有右括号，程序执行也判断出了语法错误。



1. **总结与分析**

在本实验中我使用Java语言实现LL(1)和LR()两种分析方法，并在实验时进行了两种方法的对比和分析，理解了语法分析的过程，在报告中主要介绍了LR()方法的实现过程。而LL(1)的代码实现也并不复杂，二者实际上都是通过一个while循环实现分析过程。通过实现两种分析器，使我的代码能力得到了提高。通过两种方法的对比输出，使我更加清晰的理解了不同分析方法的适用条件，执行逻辑，优缺点等等，使我对语法分析有了更为深刻的认识。

但是我的程序本身依然有很多不足，比如程序的词法分析过程比较简单，程序直接使用构建好的分析表等。

1. **思考题**

**（1）语法分析分为几类？其关键技术各是什么？**

语法分析可分为两类：自顶向下（Bottom-up）与自底向上（Top-down）。

自顶向下分析法关键技术为最左推导。自顶向下分析法要从文法的开始符号出发，并以此为根节点，使用产生式，采用最左推导方法，自上而下得构造出语法树；动作符号可以插入到产生式右部任何位置。

自底向上分析法关键技术在于确定句柄。自底向上分析法要从左到右把输入串的符号一一移进符号栈里，如果栈顶此时形成可规约串，就用规约符号代替此串，重复直到完成解析（或输入串语法错误）；其中动作符号只能在产生式最右端。

**（2）什么是递归下降子程序法，什么是LL(1)分析法？二者对文法各有什么要求？**

递归下降子程序法将每个非终结符构造出独立的子程序，当遇到非终结符时，进入独立子程序中执行，产生式右部构造子程序内容，当某个非终结符存在多个候选时，按照LL(1)形式可唯一确定选择某个候选进行推导。

LL(1)分析法是一种从左到右扫描、推导过程中使用最左推导的一种分析方法，向右看一个符号就可以判定下一个推导所用的产生式。

递归下降子程序法要求LL(1)文法，不存在左递归；LL(1)分析法要求LL(1)文法，非终结符的各个产生时select集相交为空。

**（3）比较LL(1)分析法和递归下降子程序法的异同。**

相同点：

都是自顶向下的语法分析方法；都需要LL(1)文法

不同点：

递归下降子程序法需要提前构建子程序，然后在失败过程中调用子程序来实现状态转换，LL(1)分析法首先建立分析表，在栈操作时按照分析表进行推导操作。

**（4）什么是语法制导翻译技术？其核心技术是什么？**

语法制导翻译技术是一种源语言代码的翻译，完全由语法分析器驱动的编译器实现方法，在语法分析过程中，随着分析的进行，每次使用一条产生式进行推导，就执行该产生式对应的语义动作，完成相应的翻译工作。

其核心技术是在原文法产生式中插入语义动作符号（翻译子程序），说明属性文法中属性求值时机和顺序。

**（5）表达式的四元式属性翻译文法如何设计？**

以实验二文法为例：

设计SEM(m) — 语义栈(属性传递、赋值场所)；QT[q] — 四元式区；

G(Z)： Z 🡪 E

E 🡪 E + T{GEQ(+)} ① | T ②

T 🡪 T \* F{GEQ(\*)} ③ | F ④

F 🡪 i{PUSH(i)} ⑥ | ( E ) ⑤

其中：

(1) PUSH(i)– 压栈函数(把当前 i 压入语义栈)；

(2) GEQ(w) – 表达式四元式生成函数:

① t = NEWT; {申请临时变量函数}

② SEND(w,SEM[m-1]（次栈顶）,SEM[m]（栈顶）,t)— 生成四元式QT[q]

③ POP;POP;PUSH(t)

**成绩评价表格**：

|  |  |
| --- | --- |
| 考核标准 | 得分 |
| （1）正确理解和掌握实验所涉及的基本概念和原理；（10%）； |  |
| （2）程序结构合理,要设计各种测试用例，考虑词法、语法和语义正确和不正确等各种情况（30%）； |  |
| （3）认真记录实验结果，对实验结果的分析准确（20%）； |  |
| （4）实验过程中，具有严谨、认真的学习态度，不做与实验内容无关的其他事情（10%）； |  |
| （5）实验1的程序能够正常运行,实验2应具有一定的创新性（10%）； |  |
| （6）实验报告内容充实、格式符合规范（20%）。 |  |