南 京 理 工 大 学

编译原理课程设计

|  |  |  |  |
| --- | --- | --- | --- |
| **姓 名:** | 蒋旭钊 | **学 号:** | 918106840727 |
| **学院(系):** | 计算机科学与工程学院 | | |
| **专 业:** | 计算机科学与技术 | | |
| **课 程:** | 软件课程设计（Ⅱ） | | |

2021 年 5 月

**实验1：词法分析**

# 一、需求分析

要求：创建一个词法分析程序，该程序支持分析常规语法。

1. 设计实现类高级语言的词法分析器，基本功能为识别以下几类单词：
   1. 关键字

①类型关键字：整型、浮点型、布尔型、记录型；

②分支结构中的if和else；

③循环结构中的do和while。

* 1. 标识符

由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。

* 1. 常量

无符号整数和浮点数等。

* 1. 限定符

①用于赋值语句的界符，如“=”；

②用于句子结尾的界符，如“；”。

* 1. 运算符

①算术运算符；

②关系运算符；

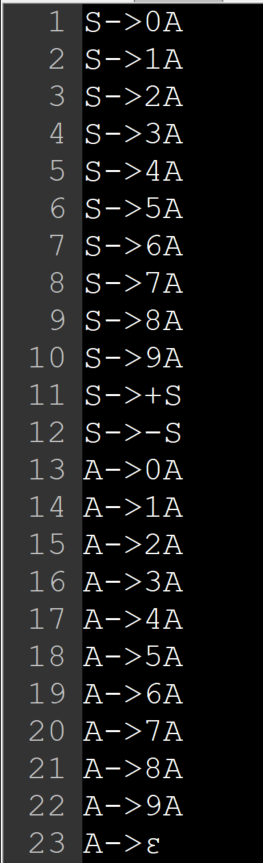
③逻辑运算。

1. 具体要求
2. 要求基于DFA技术设计词法分析器。
3. 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。

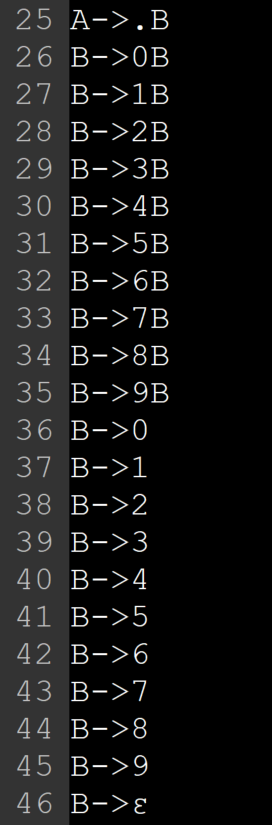
# 二、文法设计

文法产生式详见lexicalgram.txt。开始符号为S，相关内容如下：

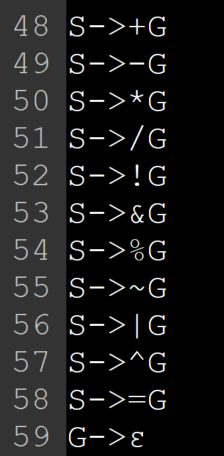
1. 整数



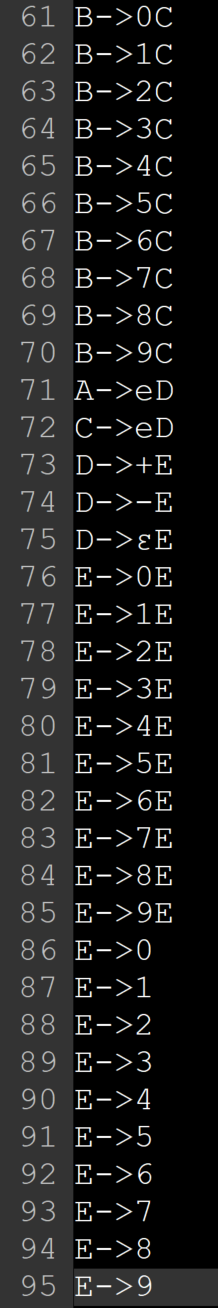
1. 小数



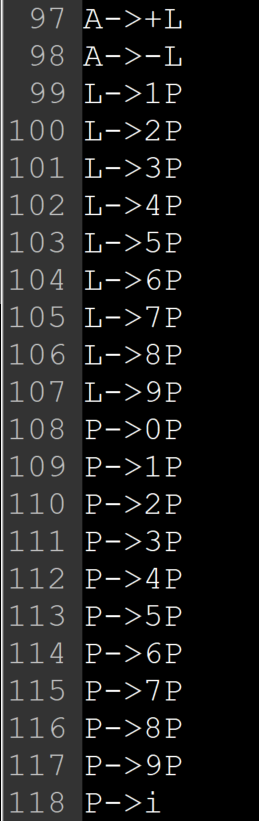
1. 运算符



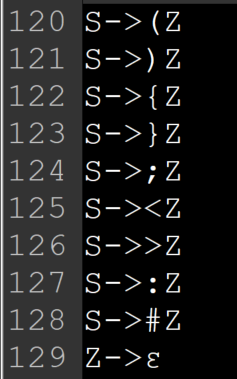
1. 科学计数法



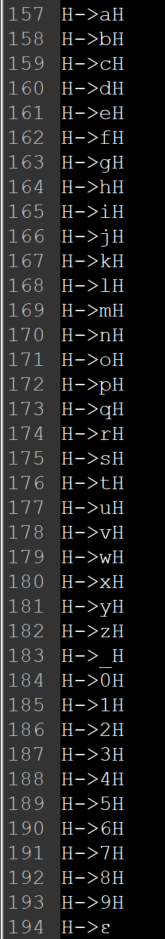
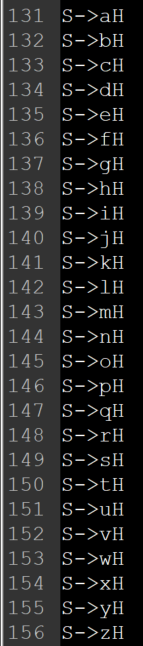
1. 复数



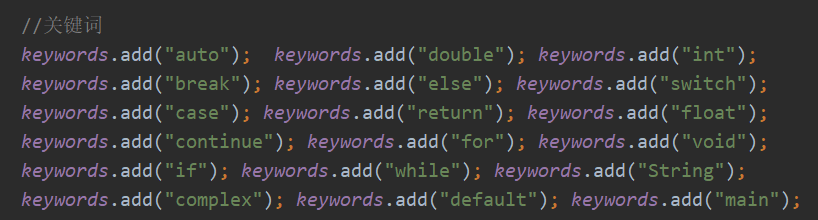
1. 界符



（7）标识符



（8）关键字内置



# 三、系统设计

3.1 设计方案

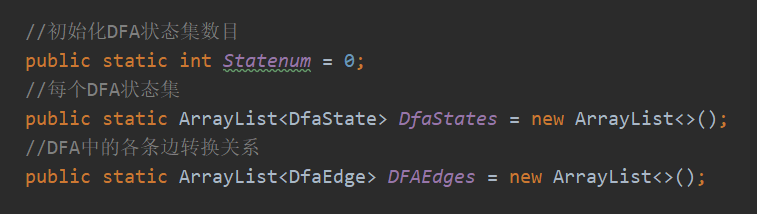
1. 程序使用右线性的正规文法作为输入，调用LexicalGram类中的read方法从文本文件中逐行读取产生式并将产生式用LexicalProc类保存，最终所有类都存储在ArrayList<LexicalProc> *LexProcF*。
2. 按照正规文法到NFA的转换规则构造NFA的边，用Edge类存储NFA边，所有边存储在NFAutil类定义的ArrayList<Edge> *NFAEdge*容器中。
3. 再调用NFAtoDFA类中定义的闭包函数Closure和Move函数，将NFA确定化为DFA，所有DFA状态用DfaState类存储，所有状态存储在NFAtoDFA类中的ArrayList<DfaState> DfaStates容器中，然后用DfaEdge类存储DFA边，所有边存储在NFAtoDFA类定义的ArrayList<DfaEdge> *DFAEdges*容器中。
4. 最后只需要调用Lexical类中的lex()函数，根据建立好的DFA状态对输入的test.txt进行词法分析，将类别区分开来，同时标记出符号和常量，将最终结果用GUI展示出来。

3.2核心数据结构

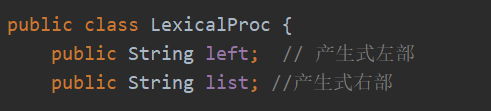
1. NFAutil中：



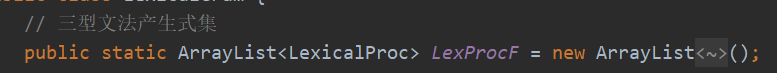
1. NFAtoDFA中：



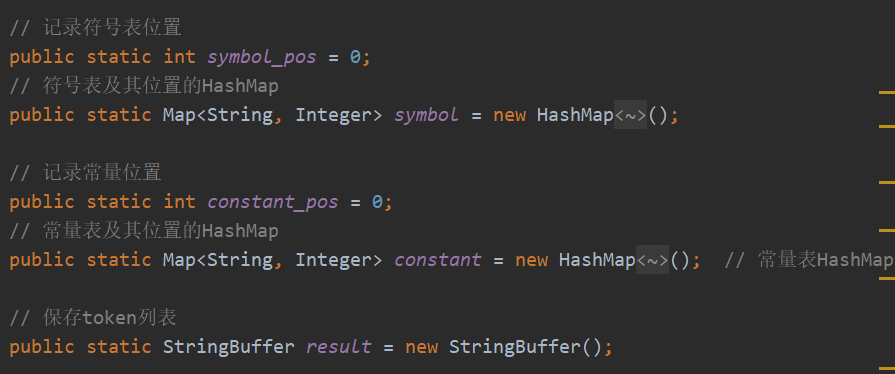
1. LexicalProc:



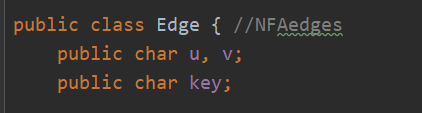
1. LexicalGram:



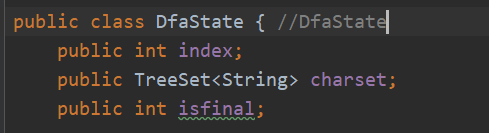
1. Lexical:



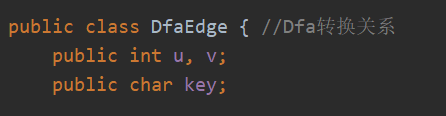
1. Edge:



1. DfaState:

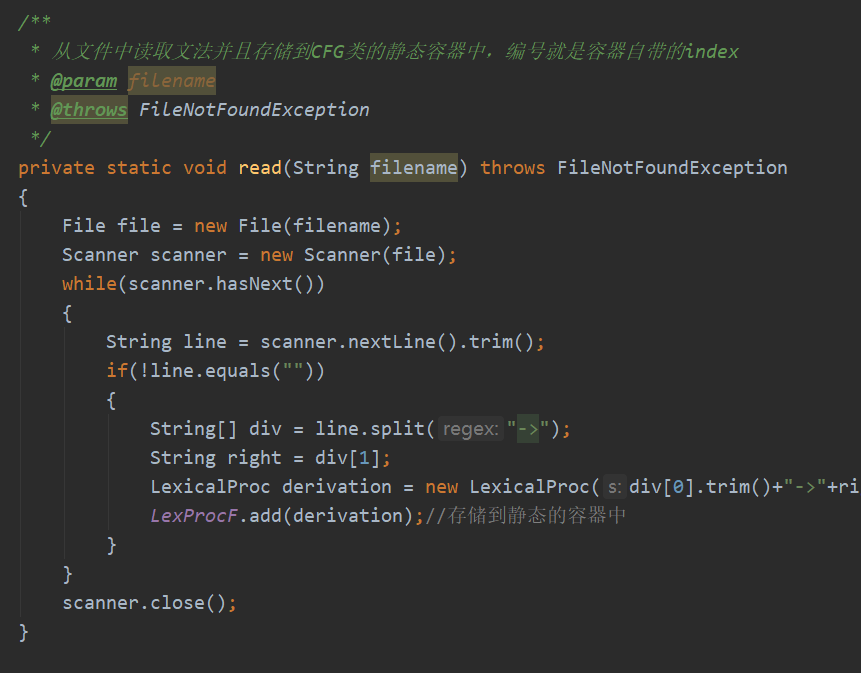


1. DfaEdge:

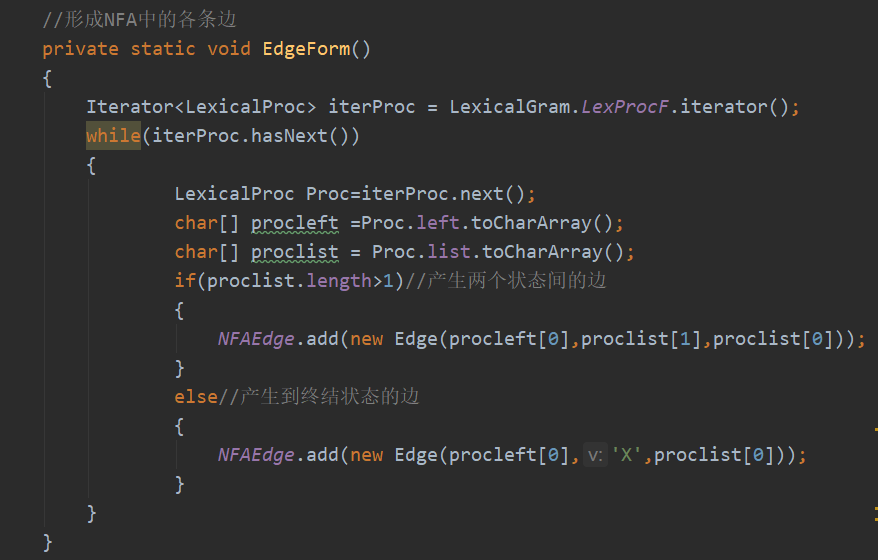


3.3主要函数及其功能

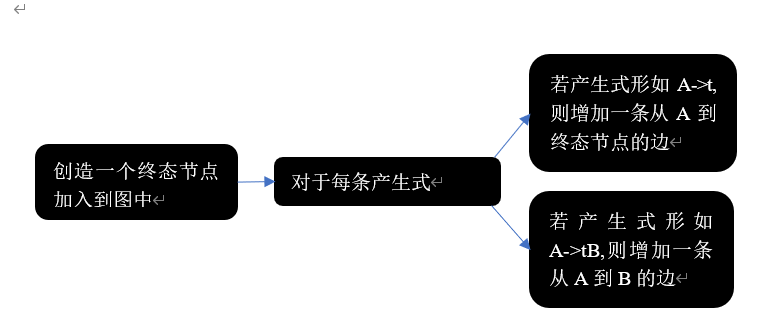
1. Void Read()



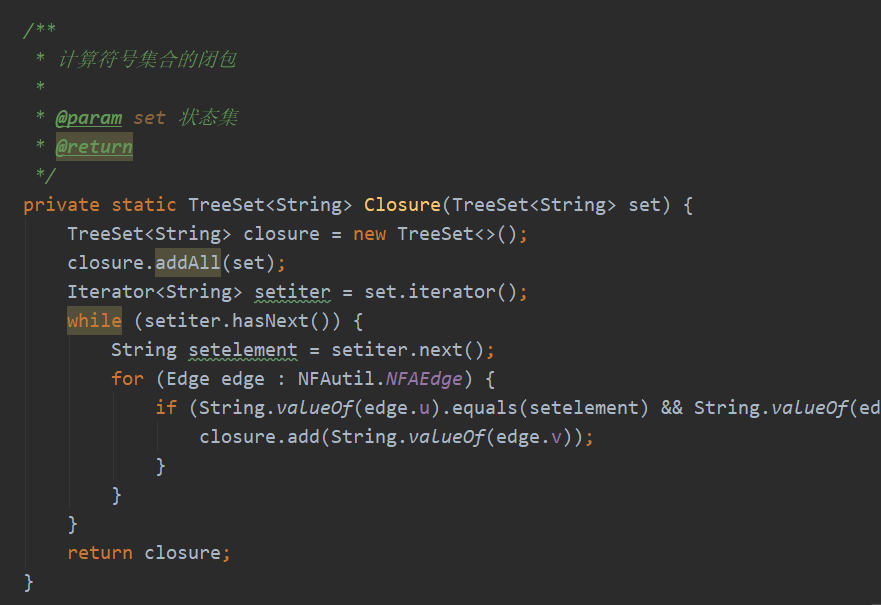
1. Void EdgeForm()



原理如下：

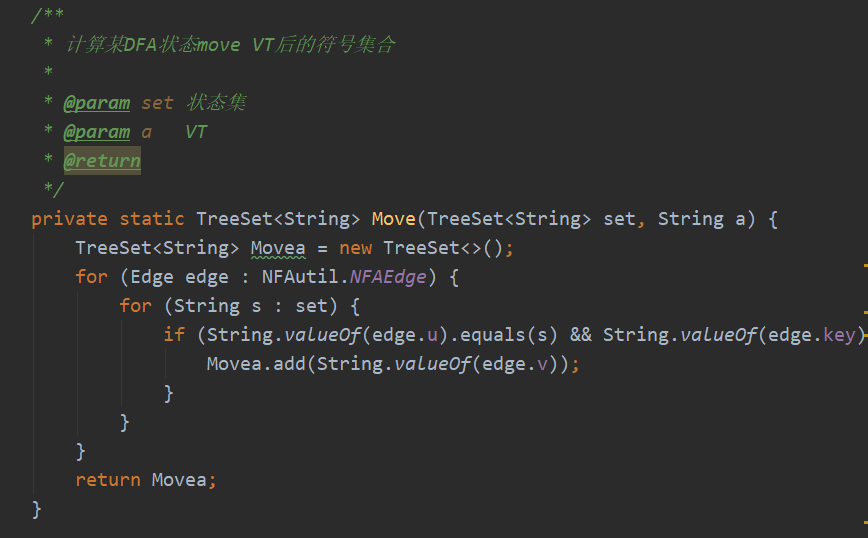


1. TreeSet<String> Closure(TreeSet<String> set)



首先将初始符号集合加入到闭包里，然后遍历该初始符号集，同时对于其中的每个符号setelement，遍历所有的NFAEdge,若NFAEdge中有出发状态u和setelement相同并且该边的key为emp的，就将结束状态v加入到closure中，最终返回closure.

1. TreeSet<String> Move(TreeSet<String> set, String a)



首先遍历所有NFAEdge，然后对于每条边，遍历set中的每个元素，如果存在NFAEdge中有出发状态u和s相同，该边的key和a相同，就将结束状态加入Movea中，最终返回。

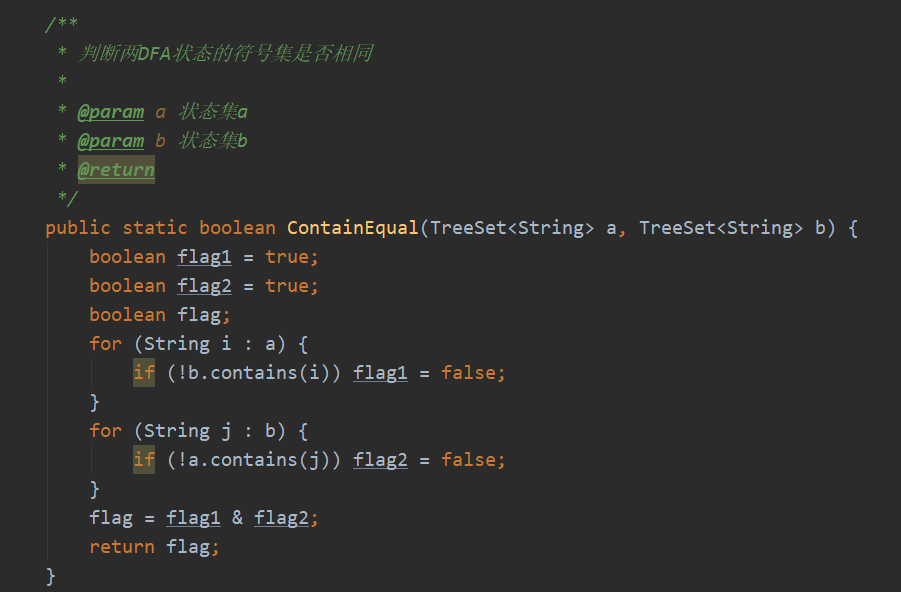
1. void formDfa()



创建初始的DFA状态，在初始状态中输入符号S，建立闭包，并加入DfaStates中，同时创建新状态队列newqueue，对所有新状态进行闭包运算。取出队首的状态，遍历所有VT，对于该状态的符号集进行Move,Closoure运算，如果在已有的所有DfaStates状态中，已经有此符号集对应的状态，则只记录DfaEdge。如果在已有的所有DfaStates状态中，不存在此符号集对应的状态，则生成新状态，产生新的DfaEdge边，同时将新状态加入到队列newqueue以便下次遍历。

1. formDfa（）中用到的几个函数



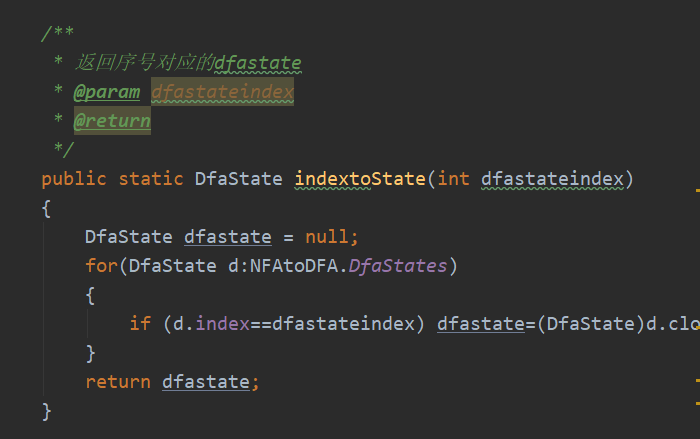


1. 根据DfaState结束状态判断识别类型

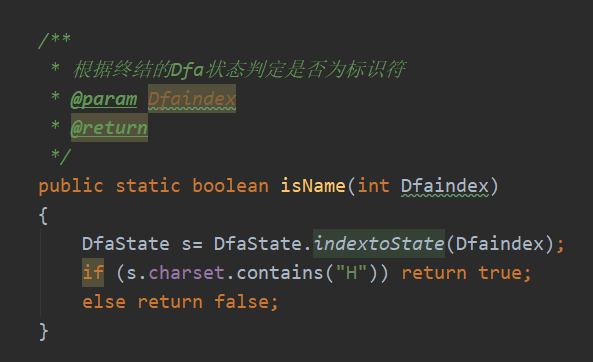
定义相应的终态符号：

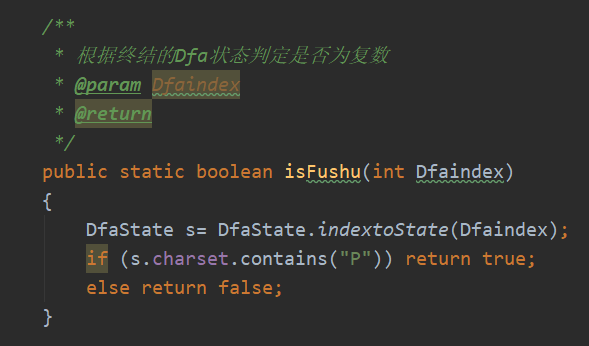


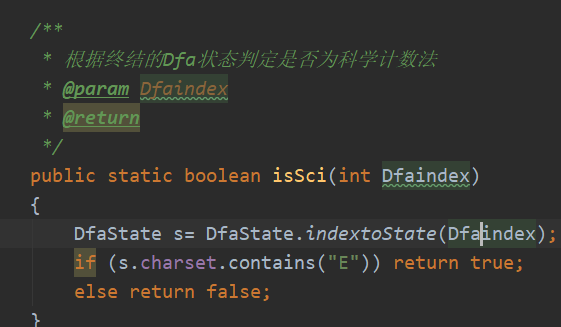
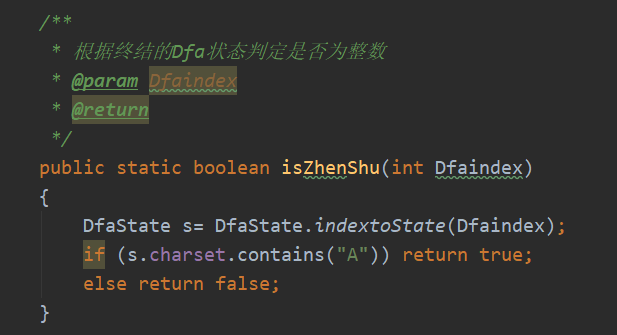
写出根据产生式序号找到产生式的函数：



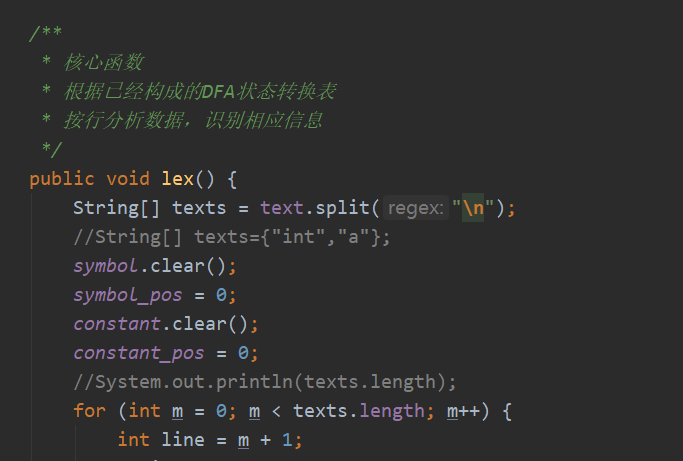
类别判断函数：

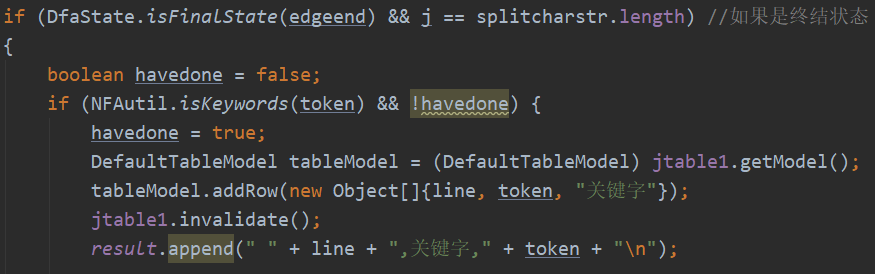


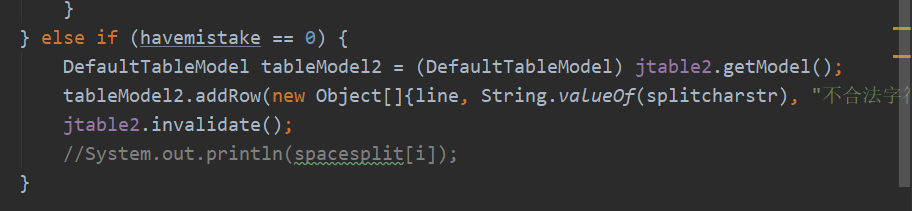




1. Lex()识别核心函数：

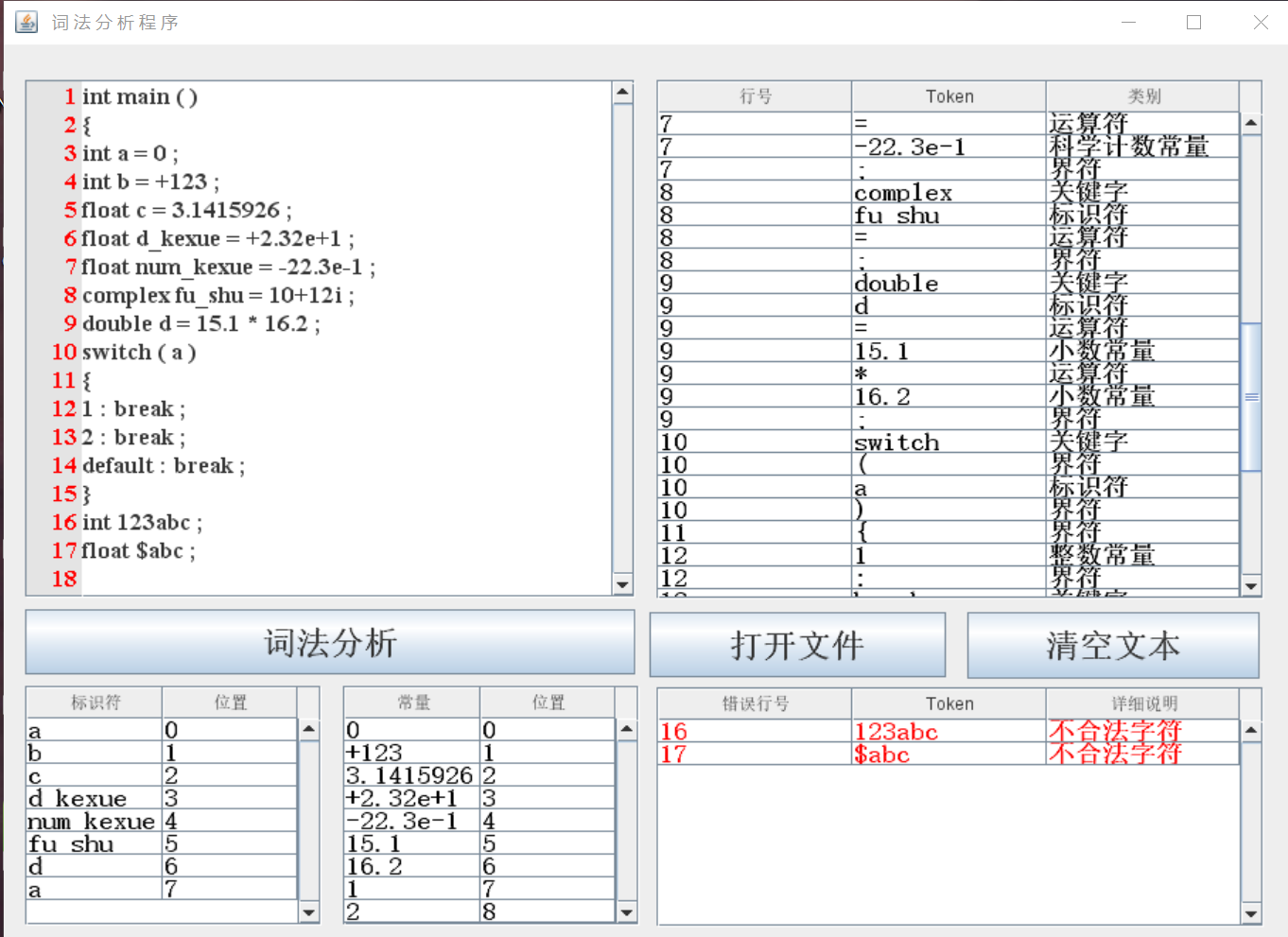




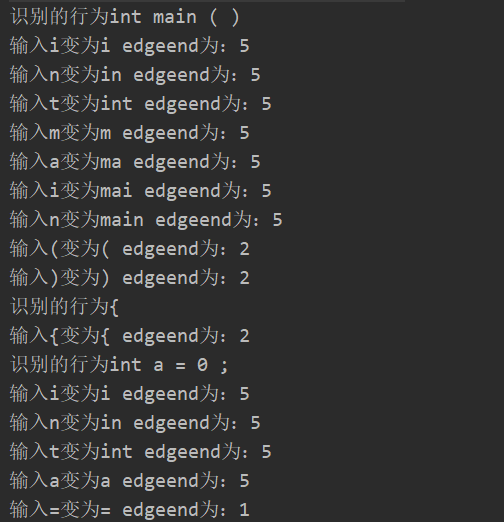


将test.txt中的每一行读入，将每一行的字符进行识别，产生相应的状态，若最终识别不出来，则进行错误提示。

# 四、结果分析



结果成功的识别了相应的文法符号，并且清楚地进行了展示。



同时在控制台中输出了相应的DFA状态转换关系以便查错。

**实验2：语法分析**

# 一、需求分析

要求：创建一个使用LR(1) 方法的语法分析程序。

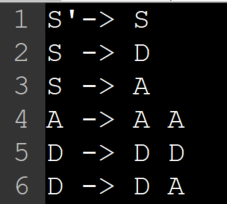
1. 具体要求

能够根据用户输入的2型文法，生成ACTION表和GOTO表，能够演示其中间过程。

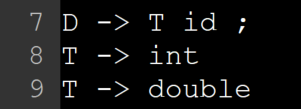
# 二、文法设计

拓广文法产生式详见grammar.txt。开始符号为S’，相关内容如下：

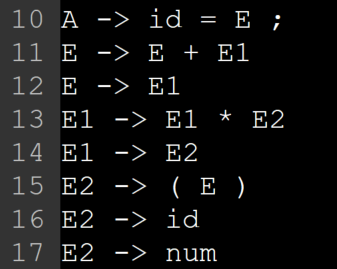
1. 产生多行变量命名和变量赋值：



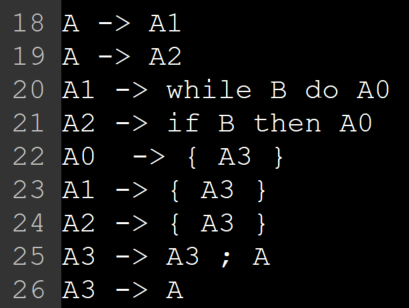
1. 变量命名和类型表示



1. 变量赋值以及代数运算和嵌套



1. 循环语句以及嵌套



1. Bool判断和bool运算语句



# 三、系统设计

3.1 设计方案

语法识别主要分为两部分：

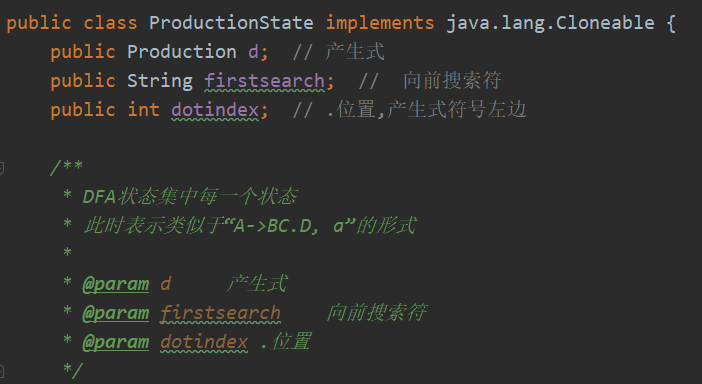
一是要根据输入的二型文法构造语法动作表。

二是要根据构造的语法动作表对输入的测试集进行相应的状态转换动作。

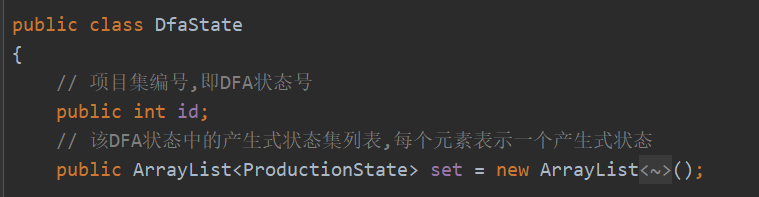
1. 首先读入二型文法产生式，用Production类存储，并将所有产生式存储在ProcsandFirst类的ArrayList<Production> GramProcF容器中。
2. 同时在ProcsandFirst类中添加使用的所有终结符和非终结符，同时调用GenerateFirst（）产生所有符号的First集，存在HashMap<String,TreeSet<String> > firstMap中。
3. 在TableForm类中构造语法动作表，首先用createTableHeader（）函数建立ACTION表和GOTO表表头，然后调用InitiateDfa()构造所有项目集族及其转换关系，将关系记录在ArrayList<gramDfaEdge> gramDfaEdges中。根据上面的基础，调用createAnalyzeTable()调充语法分析表。
4. 在SyntaxParser类中进行语法分析，用ReadToken()不断读入Token符号，构造相应的状态栈Stack<Integer>stateStack,符号栈Stack<String>tokenStack。在核心函数analyze()中，根据动作表产生相应的动作。

3.2核心数据结构

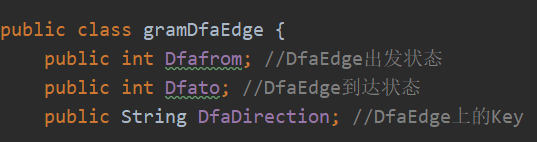
1. ProductionState类



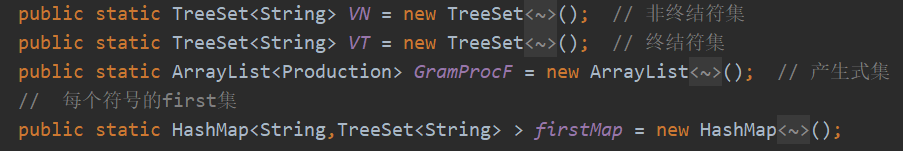
1. DfaState类



1. gramDfaEdge



1. 非终结符，终结符和First集合



1. 语法分析表和状态栈、符号栈

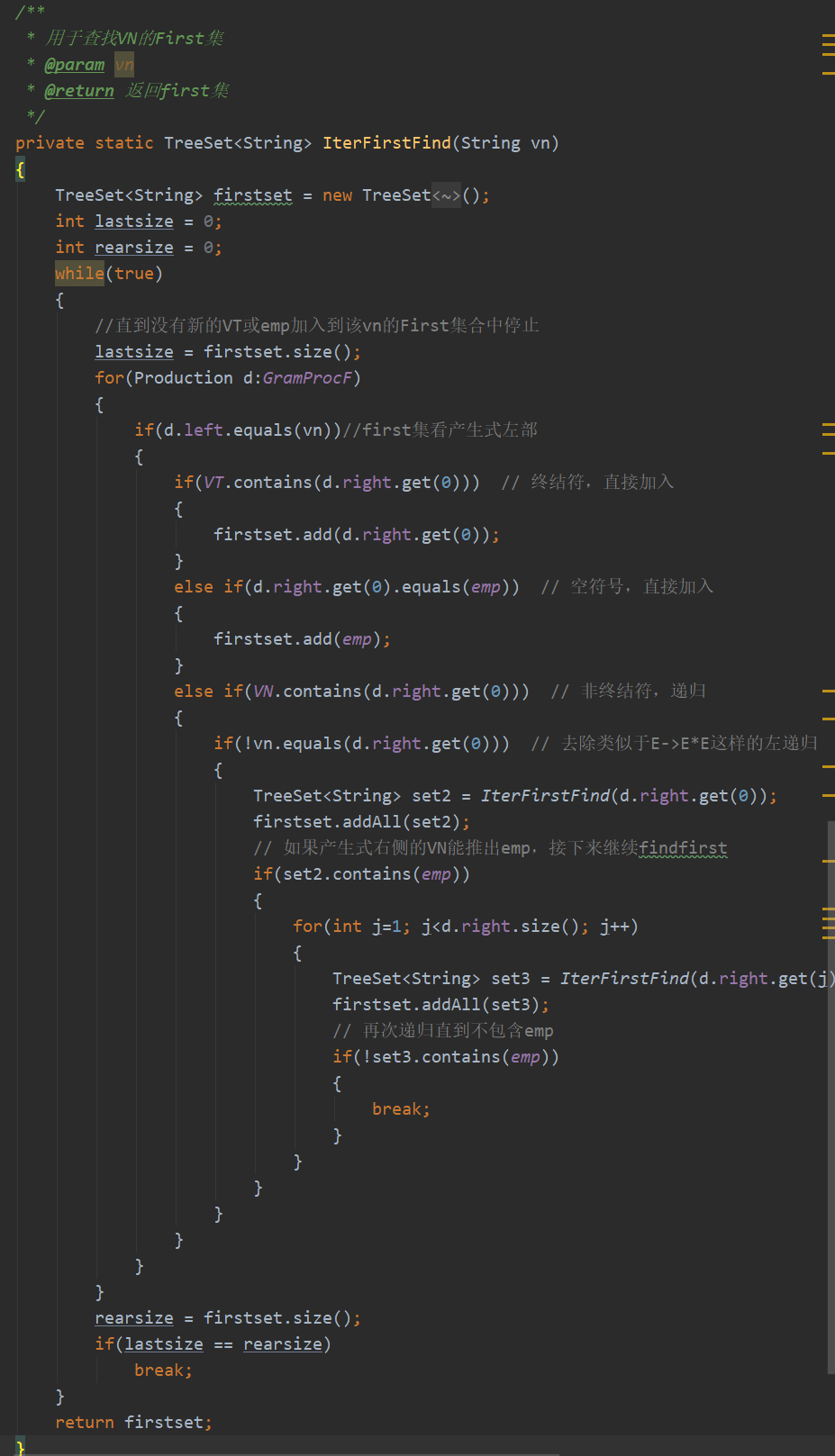


3.3主要函数及其功能

1. GenerateFirst()



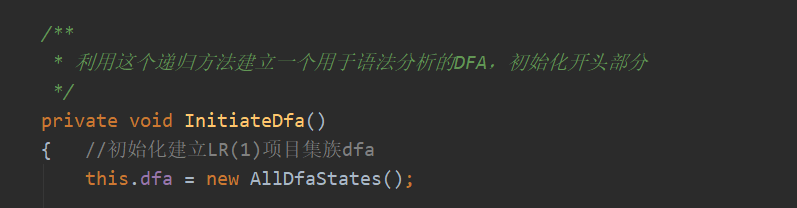
1. TreeSet<String> IterFirstFind(String vn)



计算非终结符VN的First集合，While(True)循环，终止条件是First集合中不会有新的元素加入。

* 对于某一个VN，遍历所有产生式，找到产生式左边和VN相等的，查看产生式右部的第一个符号。
* 如果是VT和emp直接加入。
* 如果是VN，则需要进一步递归，但是考虑到左递归的死循环但是对求First无影响，所以在不是左递归的情况下，求出右侧VN的First集合加入结果中，如果包含emp，则需要依产生式右部一直往下求，直到某一个符号的FirstSet不包含emp退出。

1. InitiateDfa()

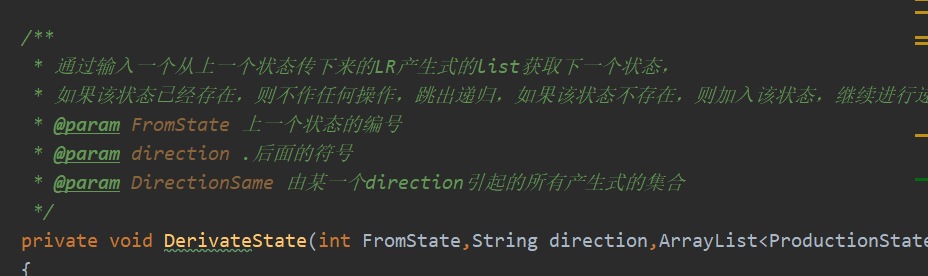


* 从state0开始建立，将S'->.S,#加入state0状态中，接着遍历该state0的所有产生式。
* 对于所有的非规约状态，即dotindex比right.size小，我们才可能会有新的产生式状态加入。
* 求取该产生式的向前搜索符searchfirst，类似于“A->B.C, #”的状态,向前搜索符不做改动，直接用原来的，否则类似求first，分情况如下：

1. 首先设置一个标志位flag=true，对于.后面的每一个符号遍历调用firstofV方法得到ProcsandFirst里面已经求好的first集，将这些first集加入到searchfirst集合中，在此过程中，如果该first集合不包含emp则置flag=false跳出循环，最终first集合包含前面所求。
2. 如果遍历完了之后每个都包含emp，则最后first集合中还要加入原来的向前搜索符集合firstsearch。

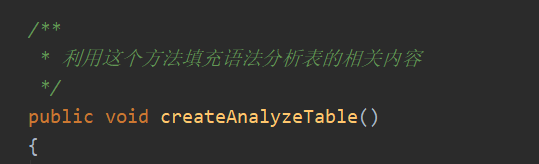
* 接着就是衍生产生式状态的加入，只有在.后面是VN时，才需要加入新的产生式状态，用leftvgetDerivation(dotA)找到.之后这个VN为左部的所有产生式，遍历所有产生式，遍历所有之前求得的searchfirst,产生新产生式状态procstate1,若原产生式推出emp，则直接把.加在最后，若原产生式不能推出emp，则将.加到最前面。
* 用lrdStatecontains判断新产生是状态procstate1是否已经存在state0中，若有则不添加，若无则添加。
* 最后将state0加入到dfa状态表中，同时获取该状态中，所有.后面的符号，以此为direction，同时对于每一种符号，用getLRDmatchs得到这一类产生式状态，最后调用DerivateState衍生其他状态。

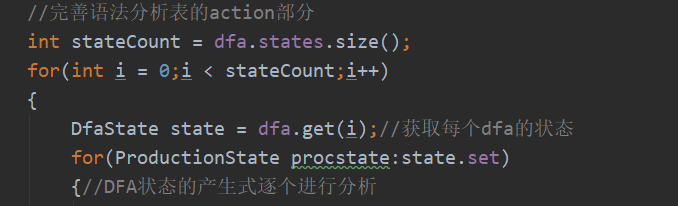
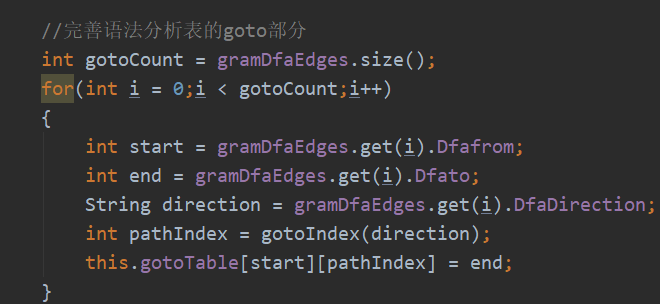
1. DerivateState(intFromState,String,direction,ArrayList<ProductionState> DirectionSame)



* 首先将所有产生式状态的.后移，然后将所有产生式加入到新状态newderiState状态中。
* 对与newderiState状态进行产生式状态的衍生和加入，类似于上面state0,如果最后产生的newderiState和以前已经有的dfa状态一样，则存储DfaEdge转换关系后，该函数返回。
* 如果产生的状态不一样，则添加新dfastate的序号和内容,增加和新状态有关的新边转换关系。
* 最后就是对于新加入的状态，继续调用调用DerivateState衍生其他DFA状态。

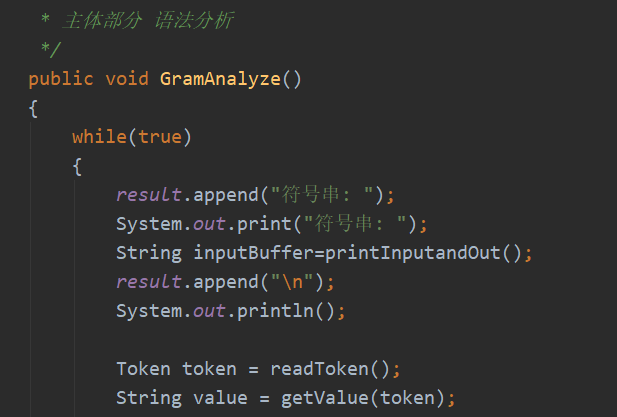
1. void createAnalyzeTable()

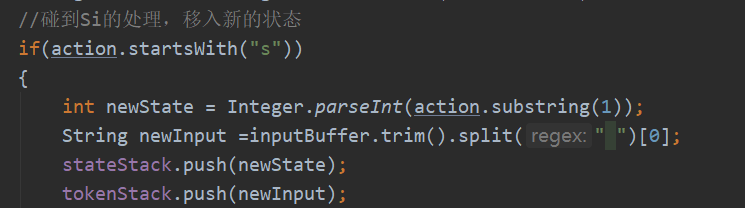


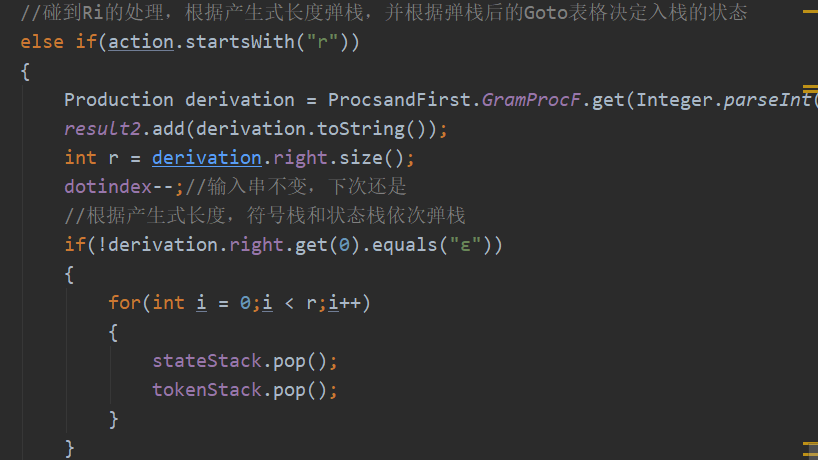


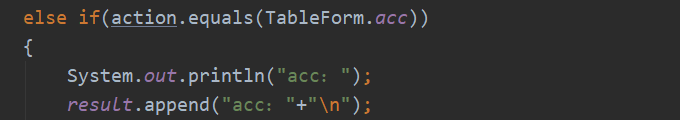
* 在GOTO表的构造中，遍历所有的DFA状态转换边gramDfaEdges，将其中的出发状态，到达状态以及输入符号key存入GOTO表中。
* 在ACTION表的构造中，遍历所有DFA状态DfaStates，在此基础上，遍历每个DFA状态的每条边。

1. 如果是规约产生式，即procstate.dotindex == procstate.d.right.size()，并且产生式左端不是S’，则找到该产生式的编号，同时将ACTION表中所有以该DFA状态i为行，该产生式向前搜索符集actionIndex(procstate.firstsearch)为列的ACTION表置为“规约状态r”；如果产生式左端是S’，设为“接受状态acc”。
2. 如果不是规约产生式，则获取.后面的符号next，如果是非终结符的话，则在ACTION表中添加以i为行，actionIndex(next)为列的“移进状态s”。
3. GramAnalyze()语法分析





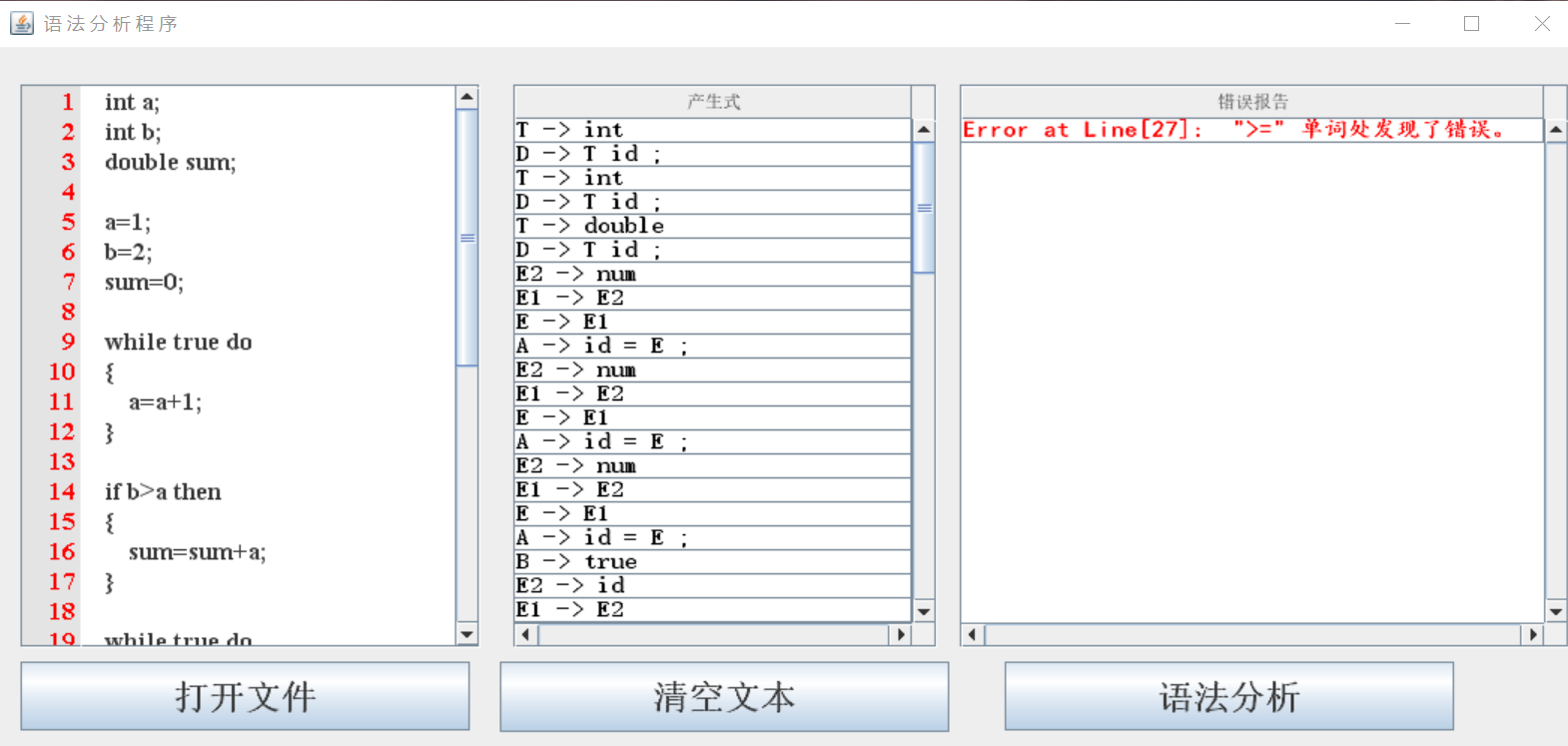




* 在while(true)循环中进行语法分析，终止条件是接受状态或者error()。
* 首先将状态0压入statestack，将符号#压入tokenstack，读取第一个符号，通过getActionElement(state, value)找到ACTION中的内容。
* 如果碰到Si，则将i新状态压入状态栈中，将移入符号串中的第一个token压入符号栈。
* 如果碰到Ri,则获取到第i个产生式，将该产生式右部的长度获得，将tokenstack中的token弹出同时将statestack中的state弹出，最后根据getGotoElement(stateStack.lastElement(), derivation.left)得到规约后状态，压入状态栈，此时analyzeindex—保证下次移入符号串分析不动。
* 如果碰到acc，则规约完成break;
* 否则碰到error，则通过error()给出错误报告。

# 四、结果分析

输入testerror.txt，带错误报告的分析如下：



输入testperfect.txt，语法分析如下：



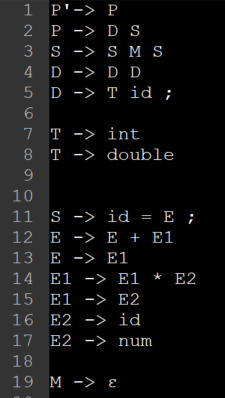
**实验3：语义分析**

# 一、需求分析

要求：创建符合属性文法的语义分析程序，可以进行简单的表达式计算，最终输出四元式序列。

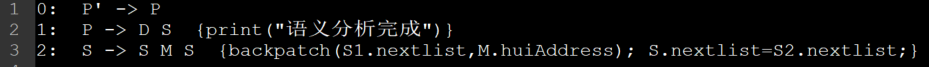
# 二、文法设计

二型文法详见grammar.txt，相应的语义动作见语义动作.txt：

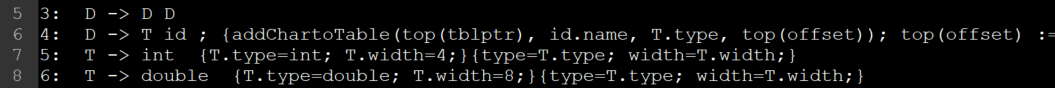


语义动作的含义：

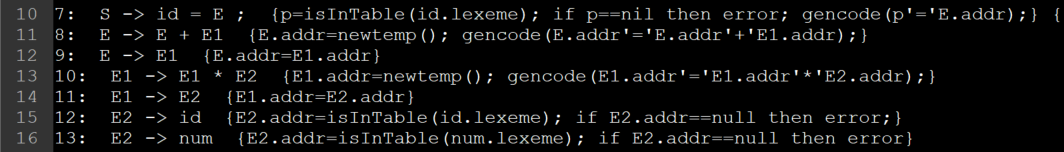
1. 全局动作



1. 声明语句



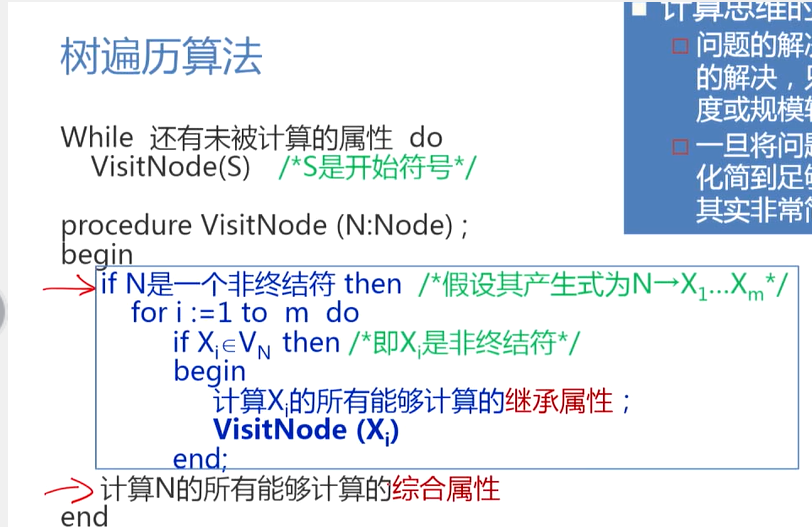
1. 赋值及算数表达式语句



# 三、系统设计

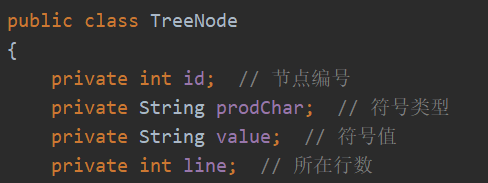
3.1 设计方案

基于属性文法的语义分析程序可以看做是LR(1)语法分析程序的一种拓展，我们只需要在语法分析的时候，在语法树上建立相应的属性文法树节点即可。同时需要事先定义好语义动作，相关的语义动作在Semantic类中分别予以实现。接着只需要深度遍历语法树，计算出全部的文法属性，执行相应的语义动作，就能够实现语义分析。

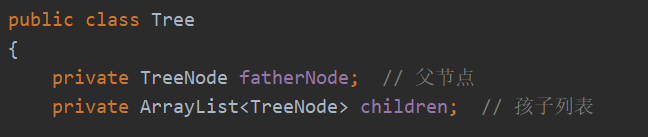


3.2核心数据结构

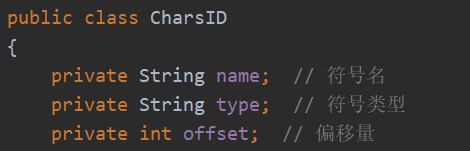
1. Treenode类



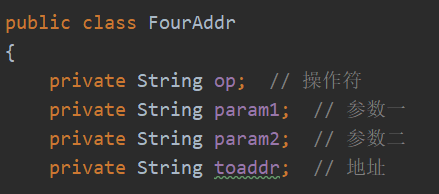
1. Tree类



1. CharsID类，保存符号表中的每个符昊



1. FourAddr类

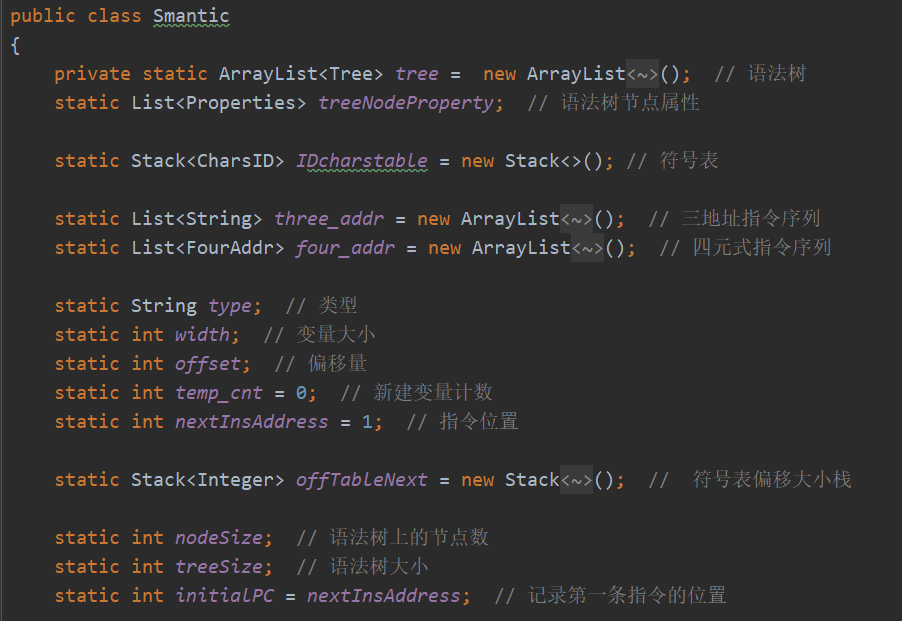


1. Properties类

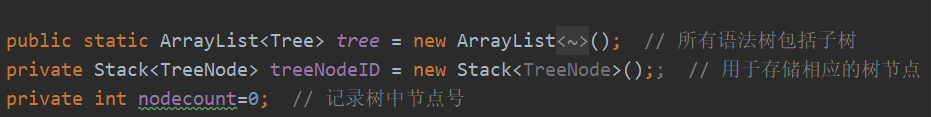


用以保存树上每个节点的属性，根据语义动作的执行填入相应的属性。

1. Semantic类，语义分析核心函数



1. 语法树建立时



存储所有语法子树以及所有的结点的结点栈Stack<TreeNode>treeNodeID。

3.3主要函数及其功能

1. 在语法分析时同时建立语法树

* 碰到s状态，直接将该VT作为树节点压入

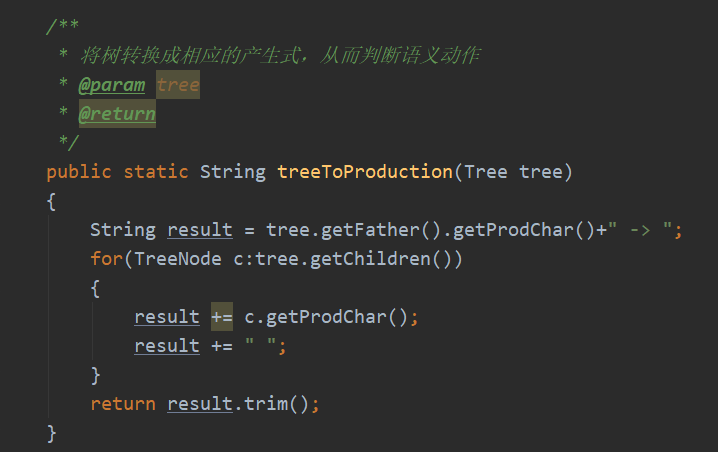


* 碰到r状态，则先获取该规约状态产生式右部的长度，将Stack<TreeNode>treeNodeID中的VT弹出作为树的子节点，将规约后的VN作为树的父亲结点，存储到树列表ArrayList<Tree>tree中。





1. String treeToProduction(Tree tree)



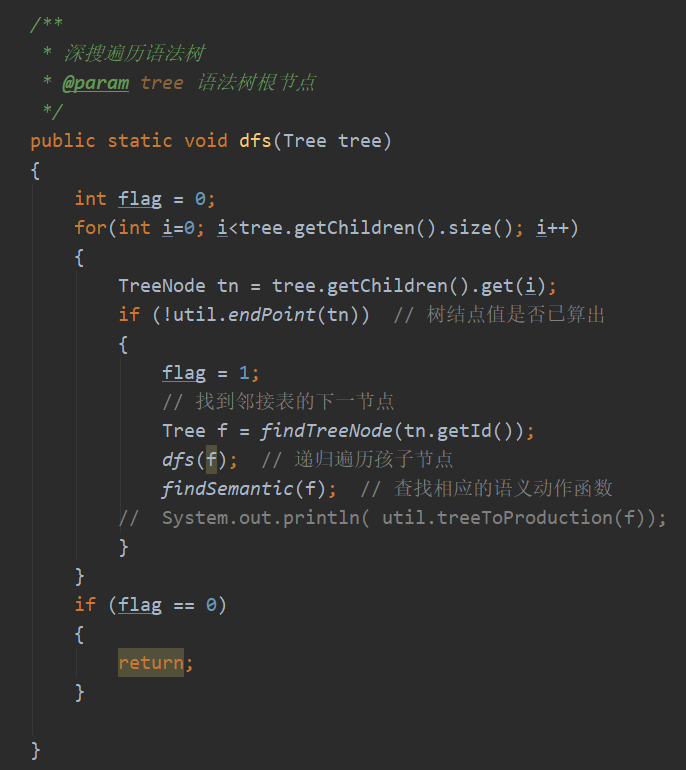
将树结构转换成相应的String，可以和Semantic里的语义执行动作匹配执行。

1. Boolean endPoint(TreeNode t)



用以判断某个树节点是否有值了，以此作为深度搜索结束的标志。

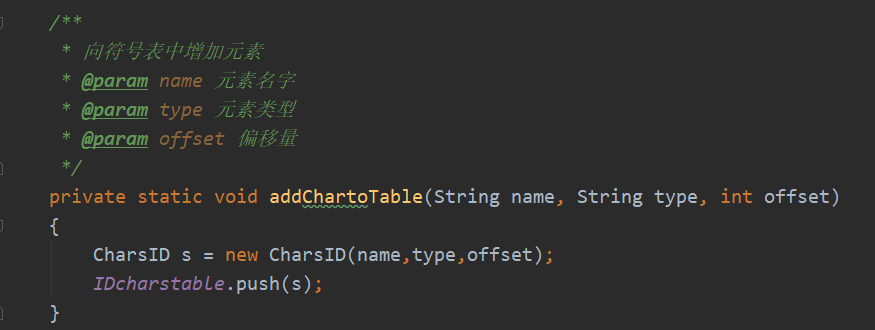
1. void dfs(Tree tree)



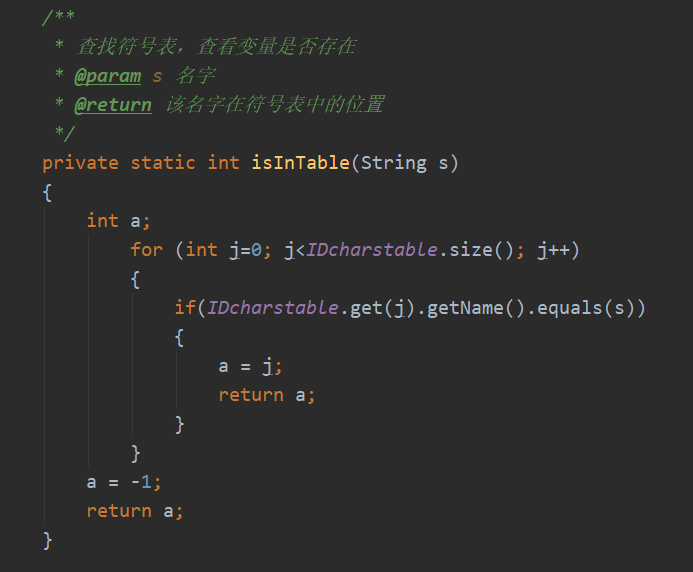
* 深度遍历语法树，如果该树的所有孩子结点的值都已经算出，则return。
* 否则以孩子结点为树节点进行递归dfs，同时用findSemantic查找相应的语义动作进行执行。

1. 实现语义动作要用到的函数

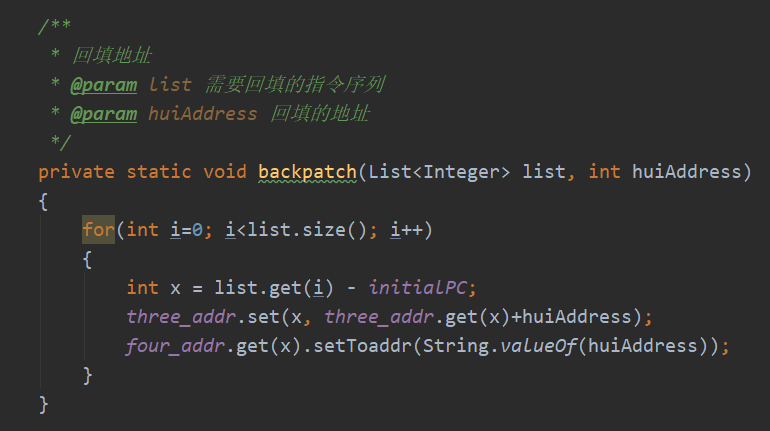
* void addChartoTable(String name, String type, int offset)



* int isInTable(String s)

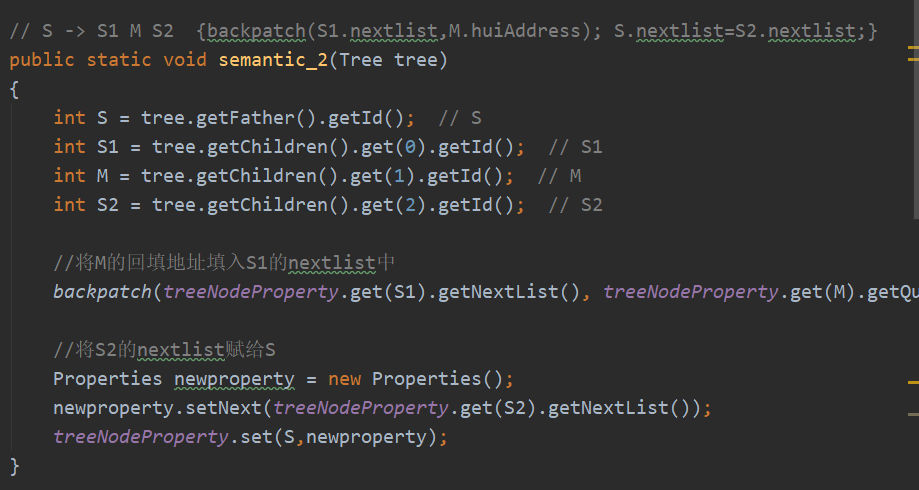


* void backpatch(List<Integer> list, int huiAddress)



1. 几类语义动作

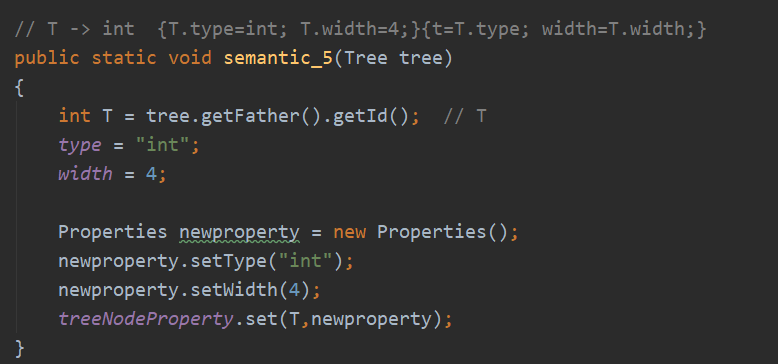
* 语句回填



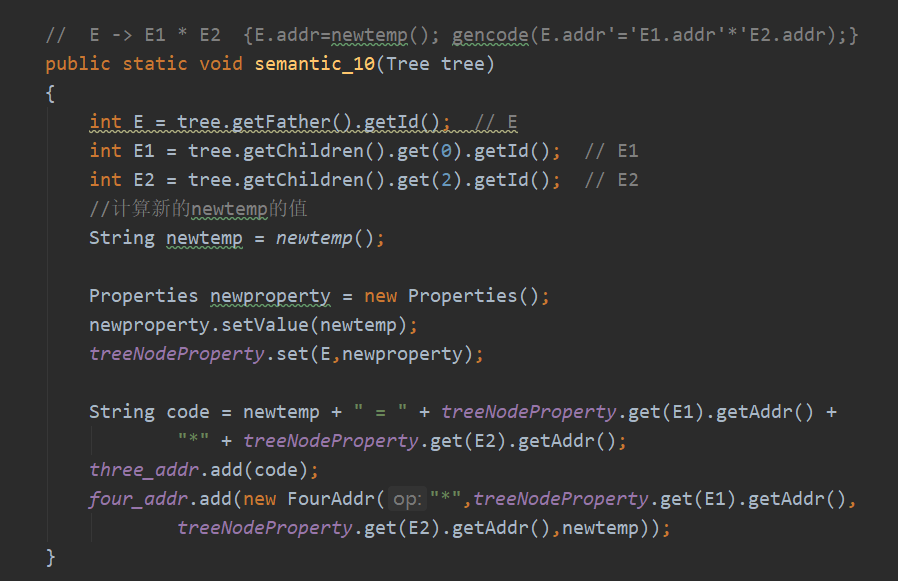
* 符号表添加



* 声明类型



* 算术运算



# 四、结果分析

将test.txt输入查看结果

