王正今

seu-lex & seu-yacc

项目文档

目录

[**一、项目简介 2**](#_Toc134705498)

[**1.整体流程 2**](#_Toc134705499)

[**2.编译对象和编译功能 2**](#_Toc134705500)

[**二、项目特色 3**](#_Toc134705501)

[**三、项目设计 4**](#_Toc134705502)

[**1.seu-lex 4**](#_Toc134705503)

[**(1)概要设计 4**](#_Toc134705504)

[**(2)详细设计 4**](#_Toc134705505)

[**2.seu-yacc 8**](#_Toc134705506)

[**(1)概要设计 8**](#_Toc134705507)

[**(2)详细设计 8**](#_Toc134705508)

[**四、使用说明 12**](#_Toc134705509)

[**五、测试用例 12**](#_Toc134705510)

[**1.DFA可视化测试 12**](#_Toc134705511)

[**2.LR1可视化测试 13**](#_Toc134705512)

[**3.c99文法解析 15**](#_Toc134705513)

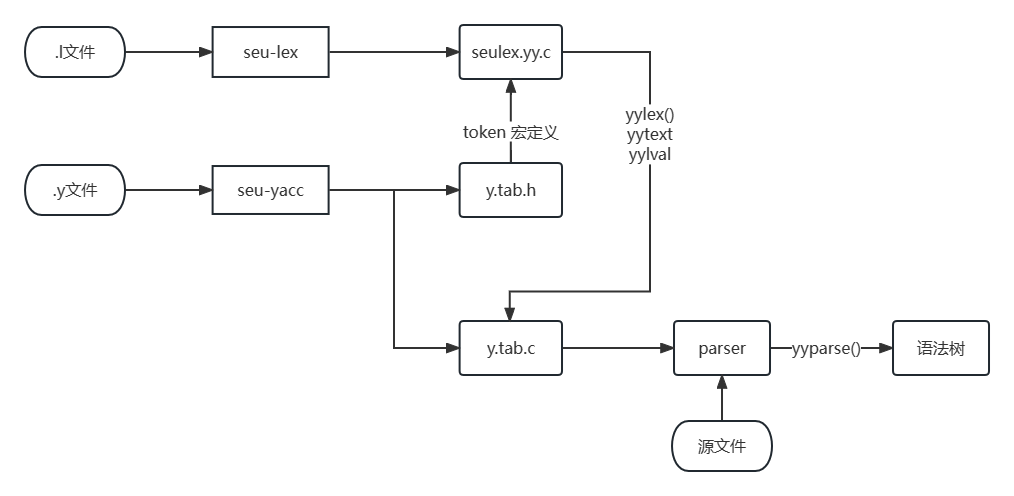
[**4.calculator文法解析 17**](#_Toc134705514)

[**5.symbol\_chart\_demo 18**](#_Toc134705515)

[**六、总结 19**](#_Toc134705516)

# 一、项目简介

## 1.整体流程



如图，将.l文件和.y文件分别输入seu-lex和seu-yacc，可生成seulex.yy.c、y.tab.h、y.tab.c三个文件，对生成的C文件进行联合编译，可以得到文法解析程序，在命令行中运行该程序即可实现对源文件的文法解析。

## 2.编译对象和编译功能

**编译对象：**

项目提供了三个demo，分别是c99，calculator，symbol\_chart\_demo。

c99文法规则与示例文件的一致，仅修改了部分语义动作执行的部分。

calculator用于演示如何实现语义动作的执行，该demo可以完成加减乘除和括号运算。

symbol\_chart\_demo用于演示用户如何定义符号表，该demo用线性表的方式存储符号表，在定义变量或重新为变量赋值时会输出提示信息，在使用未定义的变量名时会有异常提示。

**编译功能：**

**seu-lex**

LexParser.java 解析.l文件，拆分四部分，并替换正则表达式规则部分的别名

Regex.java 构造函数参数为正则表达式字符串，构造时会解析正则表达式，最终得到后缀形式的表达式，用于作为NFABuilder的输入

NFABuilder.java 将.l文件中单行的正则表达式（后缀形式）做计算，生成NFA，最后把所有NFA并联起来

DFABuilder.java 使用子集法将NFA转换为DFA，并提供最小化功能

CodeGenerator.java 生成解析词法的seulex.yy.c文件

**seu-yacc**

YaccParser.java 解析.y文件，拆分四部分，解析头部定义信息（如%token %union等），提取终结符、非终结符、产生式、语义动作

LR1Builder.java 为符号、产生式分配id，求出非终结符first集和lr(1)状态转移图，提供lr(1)转换lalr(1)功能

TableGenerator.java 根据lr(1)状态转移图生成action表和goto表

CodeGenerator.java 根据符号定义、语义动作、action goto表生成y.tab.h和y.tab.c

# 二、项目特色

支持可视化功能，包括NFA、DFA状态转移图可视化，LR(1)和LALR(1)状态转移图和action表goto表的可视化，以及语法树的可视化，项目使用graphviz作为可视化工具。

可以处理较大的语言子集，如c99

**seu-lex:**

正则表达式支持常用运算符? \* + | () “” 字符集合[] 字符集合取反[^] 转义符\

支持特殊符号.，表示除换行符外的任意字符

支持最小化DFA

支持别名定义

实现最长匹配原则

支持yytext yyleng yylineno yyless() yymore()

**seu-yacc:**

支持LR(1)转换为LALR(1)

支持头部定义规则，终结符声明%token，文法开始符号声明%start，符号类型声明%union %type，优先级定义%left %right %nonassoc。

支持二义性文法根据优先级解决冲突

支持语义动作执行，并使用$$,$1等获取产生式某一位置的元素，并根据该符号的类型声明确定使用union的哪个域

类型声明可以使用户自定义符号表，示例在demo中的symbol\_chart\_demo

可以使用yylineno检测异常出现的行号

文法符号（包括终结符和非终结符）在解析中用struct Node保存，方便语义动作的执行和语法树的遍历

action表用负数表示规约，非负数代表移进，简化了代码

用户可以使用outputGraphvizFile()内置函数在解析源文件后生成语法树可视化图片

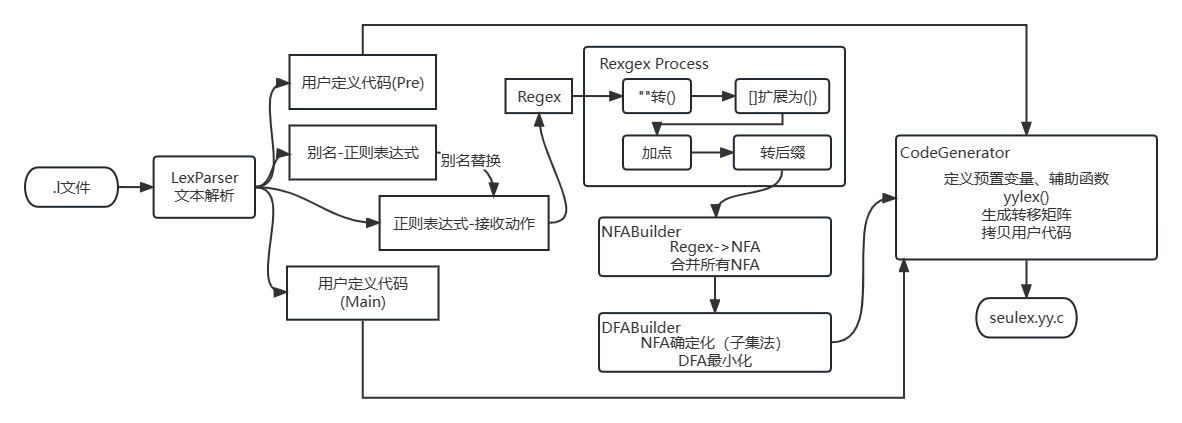
# 三、项目设计

## 1.seu-lex

### (1)概要设计

**子模块:**

主要分为.l文件解析，正则表达式转换，NFA构建，DFA构建，c代码生成五个模块。



**包结构:**

constant: 保存一些特殊常量，如正则表达式的运算符，用于字符判断

core: 核心处理函数，解析.l文件，转换NFA、DFA等

dto: 数据承载对象，包含了主要的数据结构，如Regex、NFA、DFA

utils: 可视化工具和字符判断

### (2)详细设计

**LexParser.java**

外部调用的是其中的getParseResult()方法，该方法首先按%%将文件划分为三部分（因此在每部分内部不要出现%%，不然会解析错误）。对于第一部分，调用detachDefinitionPart()方法会按%}再划分为用户代码部分和别名替换部分，并返回别名-正则表达式的Map集合，aliasMap。对先前解析的第二部分使用getRegexAction()方法处理，会得到正则表达式-接受态动作的Map集合，rawRegexAction。将rawRegexAction和aliasMap作为参数调用getRegexAction()方法就可以完成rawRegexAction中的别名替换了。

解析.l文件实际上也是解析文法的一个过程，尤其是第二部分提取正则表达式-接受态动作部分，严格来说同样需要构造一个自动机。在编写代码时引入了一些状态变量，如status、bracket、quote、slash等，用于记录状态和记录当前是否遇到某些特殊字符，如遇到\时会将slash设置为true，读入下一个字符时，由于slash为true，会忽视字符的语义，直接读入字符，然后将slash设置为false，后续对转义字符处理基本都使用该方式。状态有5个，主要的状态是读取正则表达式阶段和读取动作阶段。在测试阶段，发现动作阶段可能出现”}”、’}’这样的字符干扰解析，因此做出了一定的修正。目前的测试没有遇到新问题，但不排除有改进的可能。

**Regex.java**

调用Regex构造方法时，传递一个正则表达式字符串就可以获得一个解析好的regex对象。解析分为四步，双引号替换，方括号展开，加点处理，中缀转后缀。

1.双引号替换。该阶段把未转义的””替换为()，同时把未转义的.替换为特殊字符SpAlpha.ANY。主要作用是减少运算符数量，方便最后后缀表达式运算。

2.方括号展开。同样也是减少运算符，首先要解析方括号的内容，判断字符范围以及范围是否合理，然后将这些字符用(|)，即数个字符相或的形式表示出来，替换原来的方括号部分。如果是[^..]则要取字符补集，这时要遍历ASCII码，将不在范围内的字符用(|)形式表示，如果遇到正则表达式特殊字符，要在前边加一个转义字符\，方便后续处理。在生成NFA计算后缀表达式时，\会被视为一个特殊符号，对紧邻的下一个字符转义。

3.加点处理。除了以下三种情况，都需要在当前字符后加点。

当前字符为非转义的（和|

当前字符为正规表达式最后一个字符

当前字符的后一个字符为|、)、\*、+、?

代码中使用SpAlpha.CONCAT作为点号，以区别ASCII字符，由于Java中char类型占两个字节，使用Unicode字符集，SpAlpha.CONCAT被设置为中文字符“连”，便于调试。

4.中缀转后缀。对非操作符，直接加到字符串尾部，遇到\会对下个字符进行转义处理。对操作符会先比较符号栈栈顶元素，如果栈不为空且栈顶元素优先级大于等于当前操作符，把栈顶弹出追加到字符串尾部，直到栈为空或栈顶元素优先级小于当前操作符。最后当前操作符会push到符号栈中。完成后缀表达式转换，也去掉了括号。此时表达式只含有普通字符和| 点 \* + ? 五种操作符和\转义操作符。

**NFABuilder.java**

外部调用buildNFA()方法，先对.l文件中列出的每个正则表达式做buildSingleNFA()操作,再将所有NFA并联得到最终的NFA。下面介绍buildSingleNFA()的实现。

buildSingleNFA()参数为regex和regex对应的action，主体是对regex中的后缀表达式计算。计算结果保存在栈中，栈内的元素类型是NFA类。读入普通字符时，会调用createAtomNFA()创建一个原子NFA（初态 –字符-> 终态）；读入转义字符\，会把下一个字符当作普通字符处理（如果是n r t会做转义）；读入操作符时，会从栈顶取出一或两个元素做计算，再放回栈中。具体操作类型有：

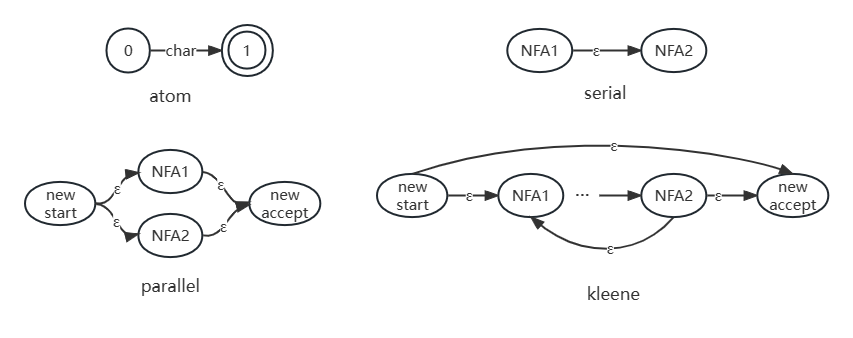
1. |运算，把栈顶两个元素取出做并联，并联后会产生新的初态和接受态。

2. 点运算，把栈顶两个元素取出做串联，用一条空串边连接两个NFA即可，需要设置初态为前一个NFA的初态，终态为后一个NFA的终态。栈顶元素是后一个状态，栈第二个元素是前一个状态，要注意顺序。

3. \*运算，取出栈顶一个元素，对原NFA做kleene闭包，产生新的初态和接受态，具体见图示。

4. +运算，取出栈顶一个元素，首先调用copyNFA()方法对原NFA做深拷贝，对深拷贝后的NFA做kleene闭包，再将两个NFA串联

5. ?运算，取出栈顶一个元素，将初态和接受态增加一条空串边连接即可。



后缀表达式计算完毕时，栈内应当只有一个NFA，将该NFA取出并设置定义的action，就完成了单个NFA的构建。

**DFABuilder.java**

提供了buildDFA()和minimizeDFA()两个方法。

buildDFA()使用子集法将NFA确定化，得到DFA。在实现时，需要调用NFA类的两个成员方法，getEpsilonClosure()和move()，分别用来求一个状态集的epsilon闭包和求一个状态集经过字符c后，得到的下一个已求闭包的状态集合。而move()方法有两个步骤，首先调用moveOneStep()方法，获取当前状态集的直接转移状态集，再对直接转移状态集用getEpsilonClosure()求得闭包。

在NFA转换DFA时，首先对nfa的初始状态求epsilon闭包，得到I0，然后把I0加入到队列中，并且要构建一个<Set<Integer>,Integer>的Map，来保存NFA状态集与新的DFA的状态号的映射，同时也可以用来查找某个转移后的状态集之前有没有被发现过。

接下来进行BFS（广度优先搜索）操作，从队列中取出头部状态集，用NFA字符表里所有字符（除epsilon）对当前状态集做move()操作，得到新状态，如果该状态没有被求过，则把它添加到队列中，并加入到map中作为一个已经求过的状态，同时也要在DFA中添加这个状态。之后无论该状态是否被求过，都需要在新的DFA中建立一条当前状态—c-->新状态的边。当队列为空的时候，转换就完成了，这时要遍历之前所有出现的NFA状态集，如果状态集存在NFA的终态，需要把该状态集对应的DFA状态号设为终态。一个状态集下，可能有不同action的终态，这时需要用到解析.l文件时对规则的优先级定义，即先出现的规则具有更高的优先级，较高优先级的动作会被视作DFA中该终态应执行的动作。

minimizeDFA()用划分的方式将DFA最小化。初始时并非只有终态集合和非终态集合两个集合，而是根据action不同将终态集合也分成了数个不同的状态集，否则不同action对应的终态最终可能无法区分。最小化过程同样用到了BFS算法，最终我们想得到的是划分好的状态集的集合。算法实现是按轮次进行的，每轮会遍历所有的状态，为当前状态所属的状态集编一个临时的状态集号，得到一个map，临时是因为每一轮的状态集号会发生改变，也就是每一轮迭代后都可能会出现新的划分，如果没有新的划分就转换完成了。接下来，只遍历还没确定好的状态集（实际上，确定好的状态集不在队列中，因此也不会在这时被遍历到），按照DFA中的字符集进行遍历，计算当前状态集下的每个状态会被转移到哪个状态，这时候需要用到之前的<状态号，所属状态集临时编号>map，此时可以知道当前状态集下的一个单状态经过字符c转移后会转移到哪个状态集上，如果在某个字符c下，不同单状态转移到不同的临时状态集号（如果单状态没有c的出边，就转移到死状态，用-1作为死状态编号），那么就根据临时状态集号对当前状态集进行划分。如果经过所有字符c后，状态集没被划分，说明它已经不会再被划分，就将它加入到finalStates（确定好的状态集）中，否则把新划分出来的状态继续加到队列中。如此循环操作，直到队列为空，此时划分便结束了。上述过程中，一个轮次的临时状态集号是不变的，即使在当前轮次下产生新状态，也是在下个轮次考虑，而不会在当前轮次考虑。这里实际上有个疑问，如果状态集Ia{1,2}在所有字符下都转移到状态集Ib{3,4}，但其中1转移到3，2转移到4，而在处理Ib时，Ib又被划分成{3}，{4}两个子状态集，算法还成立吗？Ia应该被划分吗？或者在这种情况下，Ib是不会被划分成两个状态集的？或许这里还有思考的空间。

在划分结束后，需要从划分好的状态集中选一个状态号作为该状态集的代表状态，选取的状态号可以是任意的，代码中就将状态集中的第一个状态号作为代表状态。选取时，需要建立<状态集中的旧状态号，新的代表状态号>map，所有状态集都选出代表状态后，遍历所有代表状态号，找到原DFA转移图中代表状态号的所有边，把边指向的旧状态号换成map中映射到的新状态号，以构造新的DFA边集，至此DFA最小化完成。

**CodeGenerator.java**

负责生成seulex.yy.c，包含三部分：用户代码声明部分，seu-lex生成代码，用户代码定义部分。第一部分和第三部分由.l文件解析拷贝得到。第二部分包括预置内容、辅助函数、状态转移表和yylex()

1.预置内容，包括头文件引入，宏定义，yylex()需要用到的变量。

2.辅助函数，包括yyless()和yymore()

3.状态转移表，由DFA的状态转移图产生，由于状态从0开始，行号就代表状态号，列号代表转移字符的ASCII码，表中的值为某个状态读入某个字符后转移到的下一个状态，如果没有该字符对应的边，则用-1指代无法转移。可表示为 transfer\_matrix[cur\_state][cur\_char] = next\_state;

考虑到ASCII码0一般不会作为读入的字符，列号0对应的列表示为SpAlpha.ANY转移到的状态，当cur\_state不能经过cur\_char转移到下一个状态（转移矩阵值为-1），会尝试找transfer\_matrix[cur\_state][0]的值是否为-1，也就是能不能通过ANY这条边转移到下一个状态。

4.yylex()，词法解析的核心函数，根据读入字符和当前状态不断进行状态转移，直到匹配。最长匹配原则是在状态转移时自然遵循的，只要DFA可以接收当前字符，无论当前处于终态还是非终态，都会继续读入下一个字符，直到读入当前状态下不能被接收的字符，就会开始匹配。如果当前状态是接受态，会执行相应动作（返回终结符编号），并把字符流指针回退一格，即把当前字符放回流中；如果当前状态不是接受态，说明此时违反了定义的词法规则，应当进行异常处理（返回异常编码）。如果当前状态是接受态，但是在switch-case语句中该状态不返回终结符，说明这是一个可以被忽视的词元，一般是空格或制表符这种空白符，可以返回一个预置的空白符编码。yylex()返回值会给yyparse()，用作后续的语法分析处理。

**VisualizeUtils.java**

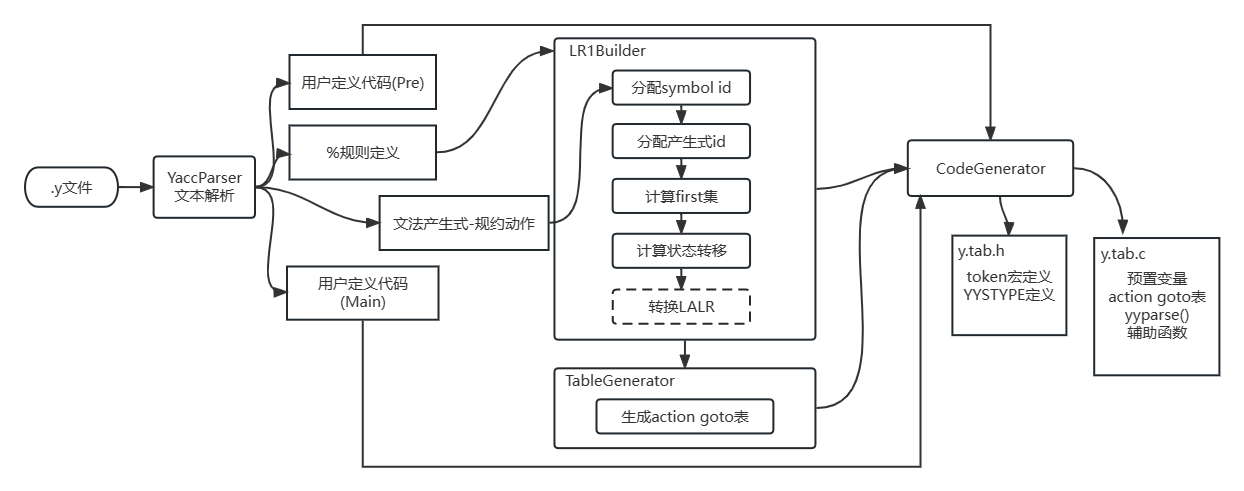
根据DFA状态转移图，先生成graphviz可以识别，符合dot语法的dot文件，然后调用Java的Process类执行dot命令，由dot文件生成图片。

## 2.seu-yacc

### (1)概要设计

**子模块:**

主要分为.y文件解析，LR1/LALR构建，action-goto表构建，c代码生成四个模块。



**包结构:**

constant: 保存特殊符号，如空串，$符，以及结合性的枚举类

core: 核心处理函数，解析.y文件，转换LR、LALR等

dto: 数据承载对象，包含了主要的数据结构，如LR1、LR1State、LR1Item

utils: 可视化工具和字符判断

### (2)详细设计

**YaccParser.java**

与seu-lex类似，外部调用getParseResult()来解析.y文件，同样也要把文件划分为三个部分。第一部分要分离用户定义代码和文法预设信息，并解析语法预设的信息，如终结符声明，文法开始符号声明，符号优先级定义等，然后把这些信息注入到parseResult当中。

第二部分为语法规则部分，它的解析较为复杂，一个文法规则分为左部和右部，左部是一个非终结符，右部以:作为开始，并用空格分割多个词元，其中可能还包括{}包括的语义动作，不同的产生式用|分割，文法规则结束用;标记，如果最后的|后边紧邻着;，说明这是空产生式。为此，代码中同样设计多个状态变量，在读入特殊字符时会改变状态，此外还要考虑{}””嵌套的影响。解析完成后，会得到所有的非终结符、产生式、对应的语义动作及优先级（通过%prec定义)。产生式右部的词元要么是在第一部分%token定义好的终结符，要么是出现在产生式左部的非终结符，也可以是用单引号’’括起来的单个ASCII字符，ASCII字符不需要在%token中定义。

第三部分是用户代码部分，划分出来后直接拷贝到最后的代码文件即可，无需额外处理。最终得到解析结果parseResult，接下来通过LR1Builder进一步处理。

**LR1Builder.java**

LR1Bulder根据parseResult来生成LR1状态图，主要分为四个步骤：分配符号id，产生式编码，计算first集，生成LR1状态图。

1.分配符号id。这一步需要将非终结符和终结符用整型代替，因为action goto表中符号作为列的索引出现，所以要进行编码。其中终结符用大于等于0的整型表示，0-127是默认的ASCII编码字符，从128开始分配自定义的终结符，在此阶段还要分配两个特殊符号，epsilon表示空串，$表示文法结束符号；非终结符用小于0的整型表示，因为一个产生式里既可能有终结符，也可能有非终结符，使用这种编码方式在后续处理产生式词元时可以方便判断。此外，如果终结符有%left %right %nonassoc的优先级/结合性声明，也要在相应编码中设置，如果.y文件中有%union的声明，也要予以保留，后续作为YYSTYPE宏定义的联合体类型。

这一步里还需要引入一个自定的文法开始符号S’,并增加S’-> S（原文法开始符号）的产生式，构成增广文法，以便LR状态图的产生。

2.产生式编码。parseResult中的产生式是List<String>类型，词元用原始字符串表示，下标为0的元素是左部非终结符，下标大于等于1的部分就是产生式的右部。在这一段使用第一步里的符号编码来替换产生式里的字符串，也就是把List<String>转为List<Integer>。并且产生式也要进行编码，序号从1开始，因为在后续生成action表时，会把规约动作表示为负的产生式序号，所以产生式序号不能为0。编码后的产生式会被加入一个Map<Integer, List<Integer>> productionIdtoProduction，通过产生式编号就能得到具体的产生式，而具体产生式里的符号都是已编码的。在这一阶段还要建立一个非终结符编号-产生式编号集合的Map<Integer, List<Integer>> nonTerminalToProductionIds，结合productionIdtoProduction可以通过某个非终结符编号获得它有哪些具体的产生式，为后续计算first集做准备。

3.计算first集。计算first集需要遍历某个非终结符的所有产生式，对右部用求first集的算法求解。然而如果遇到右部的一个非终结符词元，需要获得对应的first集，因此刚开始的思路为动态规划，当求解到一个非终结符的first集把它标记为searched，再使用该非终结符时直接返回已记录的first集即可。但用minic文法测试时，发现这个算法不能应对左递归文法（包括直接左递归和间接左递归）。在网上查阅资料[https://www.zhihu.com/question/407704983]得知，可以先求非终结符是否会推导出空串，在求解函数中，还要加一个preNonTerminals参数，来保存当前路径已经访问过的非终结符，如果此时遇到的右部非终结符出现在preNonTerminals中，说明发生左递归，会返回false，表示不再考虑这条路径，从其他路径获取该非终结符是否会产生空串。最终得到每个非终结符是否会产生空串的信息。接下来，再去求解first集，求解函数也需要preNonTerminals参数，在没有遇到左递归的情况，按原算法求解，如果发生左递归，根据该非终结符是否能产生空串来判断，若不产生空串，返回空集合即可，它的first集在之前也一定被求过，若产生空串，则还有继续向右寻找first集的可能，需要递归向下搜索。

4.生成LR1状态图。在实现构建LR1状态图的算法前，需要定义两个数据结构LR1State和LR1Item。LR1State有状态编号stateId以及内部所有的项items（由LR1Item组成），LR1Item包括三个属性，产生式编号、点的位置、预测符，其中每一个item只记录一个预测符，不做合并，可以方便LR1State的比较，两个LR1State的所有item是一样的则两个状态相同，用于后续产生新状态时判断它是否在之前出现过。此外还定义了一种数据结构LR1ItemCore，它只包含产生式编号和点的位置，用作可视化和LALR的生成。

同时，还要对LR1State设计两个函数，求项集闭包，代码中称作内扩展innerExpand，求项集的所有移进后状态，代码中称为外扩展outerExpand。innerExpand用到BFS算法，先将LR1State的初始items添加到队列中，然后开始BFS，取出队头，检查当前item的当前点位置后边是否是一个非终结符，若是则可以进一步扩展，需要遍历该终结符的所有产生式，建立新的item，并检查这个item是否在之前出现过，没有出现过的item才会被添加到队列中，并添加到当前的LR1State中。outerExpand则是遍历LR1State的所有item，将点的位置向后移动，建立一个移进符号-新产生的item集合的映射，有多少个移进符号就有多少个新的状态，把移进符号对应的item集合作为新状态的初始items，再进行innerExpand，然后检查这是否是一个重复状态，没有重复就添加到返回值集合中，而无论是否是重复状态，都要建立一条当前状态->下一状态的边。

构造LR1状态图用到了BFS算法，开始构建0号状态作为开始状态，并添加一个item S’-> ·S, $，再进行内扩展，然后把它加入队列，开始BFS，对队头的每一个状态进行outerExpand，outerExpand会返回所有不重复的新状态，把这些状态放入队列继续循环直到队列为空，最终构建了整个LR1状态图。

代码中还实现了有LR1生成LALR，原理和最小化DFA类似，首先要找出所有LR1的core，把相同core的LR1划分在一起，然后从中选一个代表状态作为新状态，并需要建立旧状态-新状态的映射。然后遍历所有新状态，根据旧的LR1状态转移图，把出边指向的状态替换为映射到的新状态，实现新的LALR状态图构建。

**TableGenerator.java**

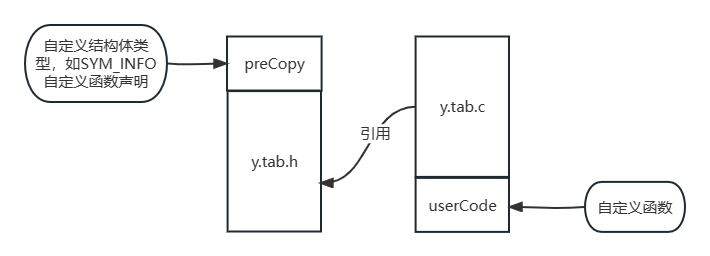
TableGenerator的构造函数需要传递两个参数，LR1和TransGraph，以此来确定生成的是LR1还是LALR的转移表。生成action表时，要遍历每个状态，由于状态号从0开始，可以直接把状态号作为action表的行号（goto表也同样），列号也可以直接使用终结符的编号。对每个状态，首先确定它的移进项，找到所有的出边，把出边符号作为列号，指向的状态作为填表的值，然后确定规约项，要找到状态中所有点移动到最后的item，并根据预测符来填值，值为item对应产生式编号的负值。这里可能会发生移进-规约冲突或者规约-规约冲突，这时就要用到之前的符号和产生式的优先级以及符号的结合性来做冲突处理的判断了。

goto表则是只看状态的非终结符出边，由于非终结符是负值，因此对应的列号被设为(-编号-1)，填值就是状态的转移号。

最后会生成两个二维数组，分别代表action表和goto表，可以通过tableGenerator.getActionTable()和getGotoTable()来获取。

**CodeGenerator.java**

CodeGenerator会生成两个文件，y.tab.h和y.tab.c，其中y.tab.c包含用户定义代码的第一部分，以及生成的自定义终结符的宏定义，还要%union自定义类型的YYSTYPE宏定义。y.tab.c则包含解析文法的主要函数yyparse()，一些数据结构和action goto表，还有异常处理函数。代码结构如下图所示：



下面介绍y.tab.c的主要构成

1.preset，预置部分。主要包含头文件引入、声明变量、异常处理函数和主要数据结构及函数。主要数据结构有两部分，第一部分是struct Node，用来表示语法树中的一个节点，包含它的符号信息，语义值(YYSTYPE类型)，如果是非终结符还有产生式以及子节点指针。并设有新建节点的函数。第二部分是状态栈和符号栈，并设有放入、弹出、获得栈顶元素的函数。

2.table，表信息。除了action表和goto表外，还有非终结符和goto表列转换的函数，以及产生式-产生式长度表和产生式-非终结符表，用来确定规约时状态栈、符号栈弹出元素的数量，规约后加入符号栈的非终结符号。此外还有编号-字符串表，用作可视化。

3.actionSwitch，语义动作。生成一个doSemanticAction()函数，规约时会用产生式编号作为参数调用这个函数，如果该产生式有定义的规约时动作便会执行。在这里$$、$1..会分别被解析成Node的val或者Node的某个孩子节点的val，并会根据产生式中的符号事先声明好的类型，确定使用union中的哪个域。

4.printGrammarTree，打印语法树的函数。在分析文法的过程中，同时也会建立Node之间的关系，据此可以对树进行遍历，输出可视化的结果。

5.yyparse，解析文法的核心函数，一次性解析整个文件的文法。主体是一个循环，每次循环都会调用yylex()获取当前token，根据当前token与当前状态查action表确定下一步要执行的动作。

如果action<0，是规约动作，要根据产生式长度弹出状态和符号栈里的Node，并新建一个语法树节点，把弹出的Node都作为它的子节点，然后执行相应的语法动作，然后把非终结符加入符号栈中。规约完成后还要查goto表，转移到下一个状态，并再根据action表查是否会进行规约，这同样也是一个循环，直到action >= 0才会跳出循环。

如果action==0，说明文法解析出错，会抛出异常。

如果action>0，则是移进动作，新建一个语法树节点，把它加入符号栈，根据yylex()得到的yylval设置节点的语义值，然后把转移后的状态号加入状态栈中。

当输入文件的文法能被解析成功时，在规约阶段会得到action表中的-1，表示规约1号产生式，也就是S’-> S，这时相当于成功接收，函数会返回0。

**VisualizeUtils.java**

和seu-lex类似，根据LR状态转移图，先生成graphviz可以识别，符合dot语法的dot文件，然后调用Java的Process类执行dot命令，由dot文件生成图片。每个状态下都有多个item，包括产生式、点的位置和预测符，其中点用中文符号·表示，因此在dot文件需要定义使用SimHei字体。由于LR1State中相同core不同预测符的item被视作不同的实体，为增强状态图可读性，对同一core的item做了合并操作。

# 四、使用说明

项目需要maven和graphviz依赖。

1.在两个子文件夹seu-lex和seu-yacc下分别执行mvn package -DskipTests，得到两个jar包。

2.使用java -jar seu-lex-1.0-SNAPSHOT.jar <.l文件名> [options]

java -jar seu-yacc-1.0-SNAPSHOT.jar <.y文件名> [options]

可以在命令行中运行seu-lex和seu-yacc工具

seu-lex 默认输出seulex.yy.c一个文件

seu-yacc 默认输出 y.tab.h y.tab.c 两个文件

seu-lex 可选参数:

-v 输出DFA可视化图片

seu-yacc 可选参数

-v 输出lr状态转移图可视化图片

-lalr 构造lalr(1)分析表

3.将y.tab.c seulex.yy.c联合编译即可输出解析程序（请在.l文件中引入y.tab.h）

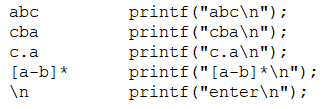
例 gcc y.tab.c seulex.yy.c -o parser.exe

在demo文件夹下可以运行示例。

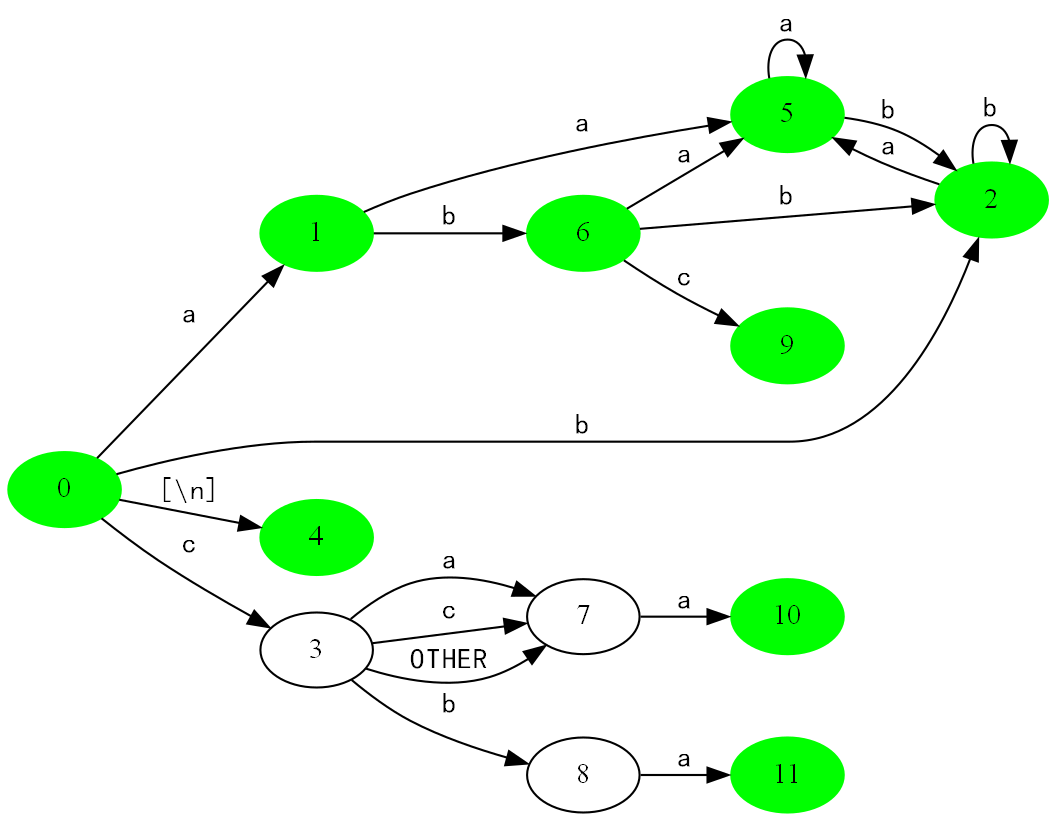
# 五、测试用例

## 1.DFA可视化测试

正则表达式规则定义：

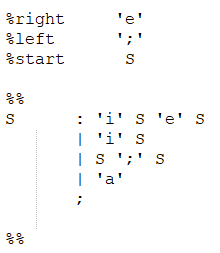


DFA状态转移图（绿色为终态，白色为非终态）：

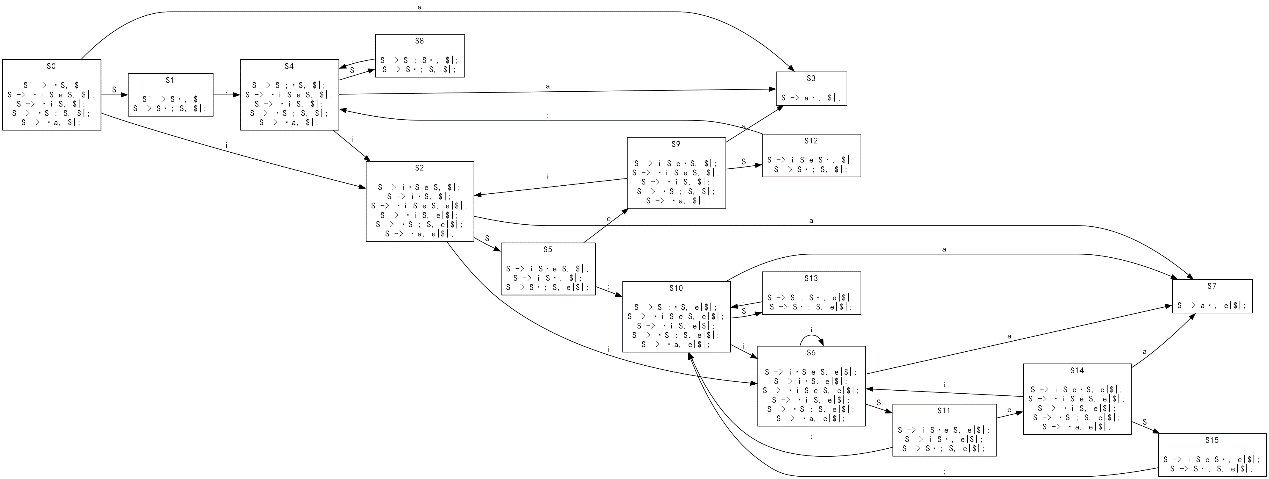


## 2.LR1可视化测试

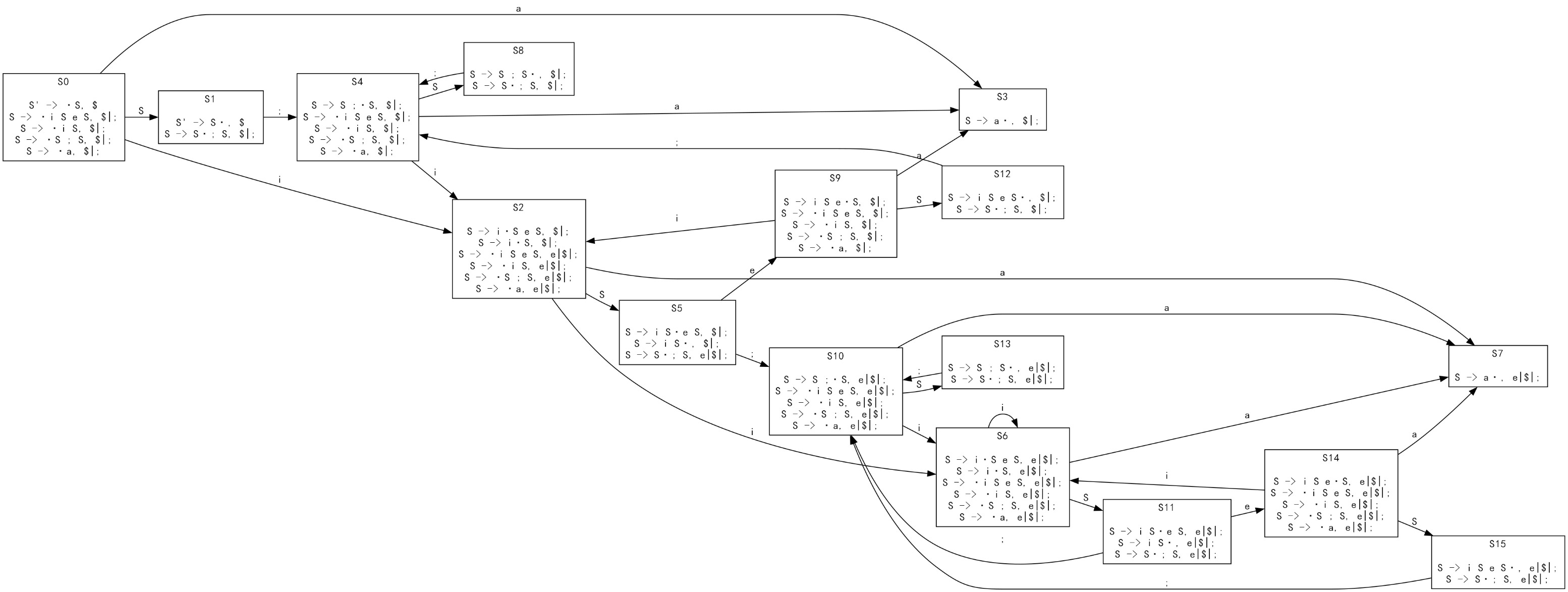
文法规则定义（源自作业题）：



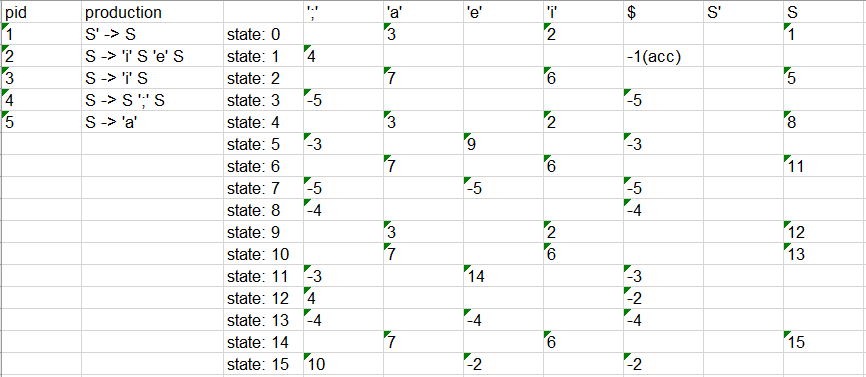
LR(1)状态转移图：



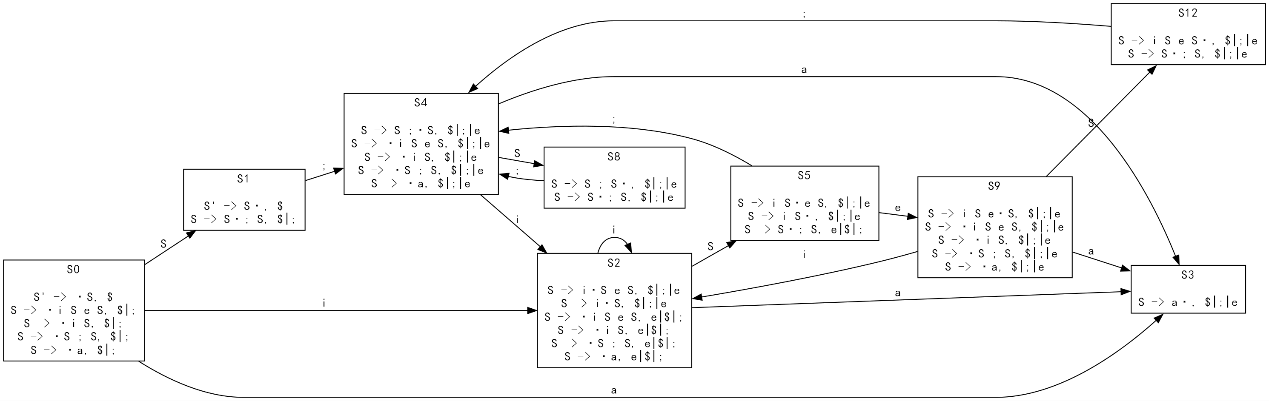
局部放大：



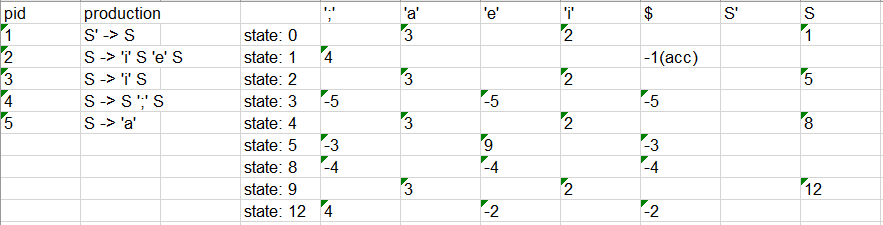
LR(1) action goto 表：



LALR(1)状态转移图：

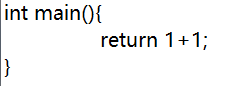


LALR(1) action goto 表

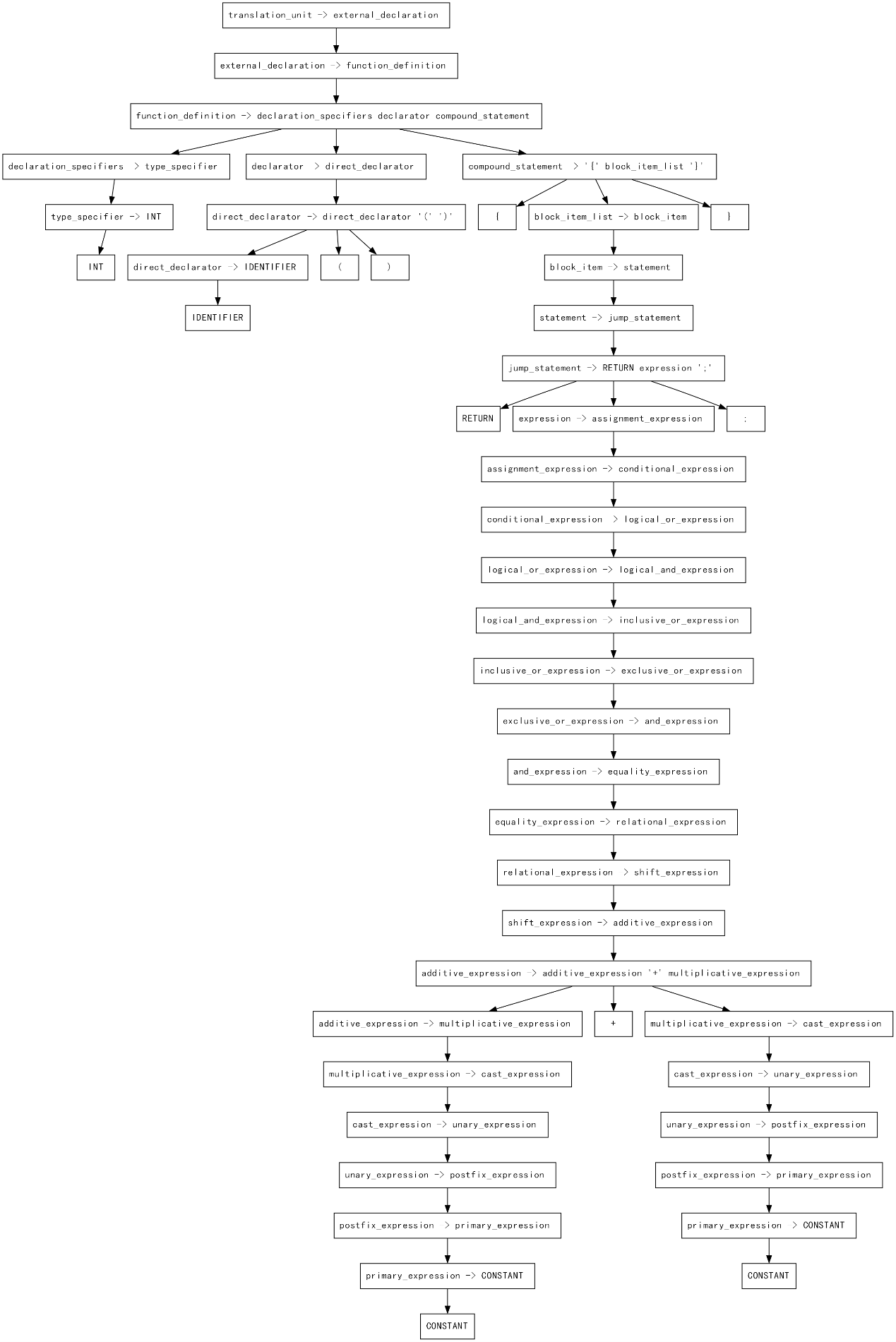


## 3.c99文法解析

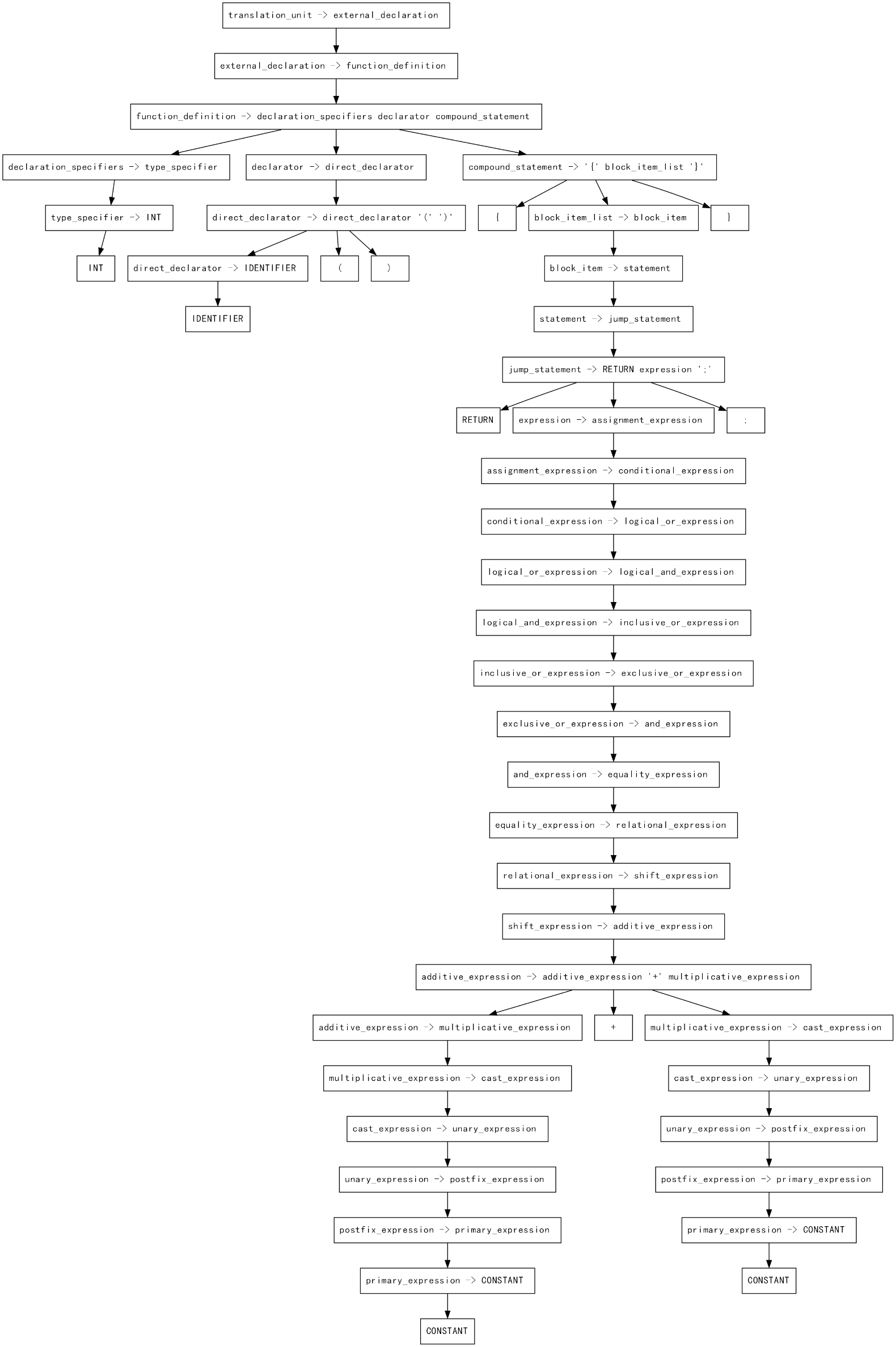
输入文件：



语法树：



局部放大：

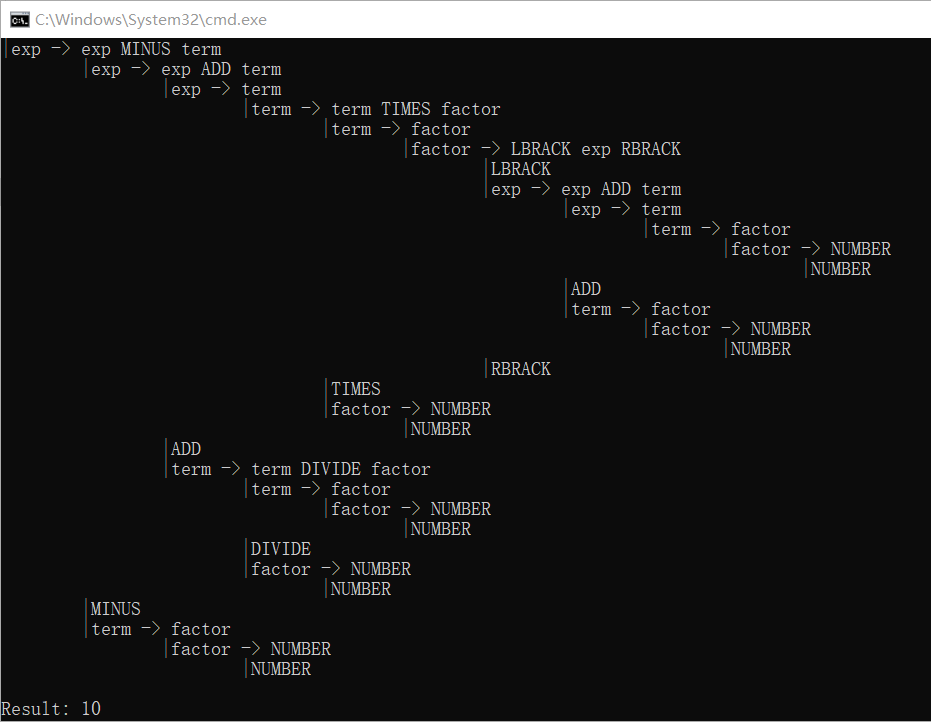


## 4.calculator文法解析

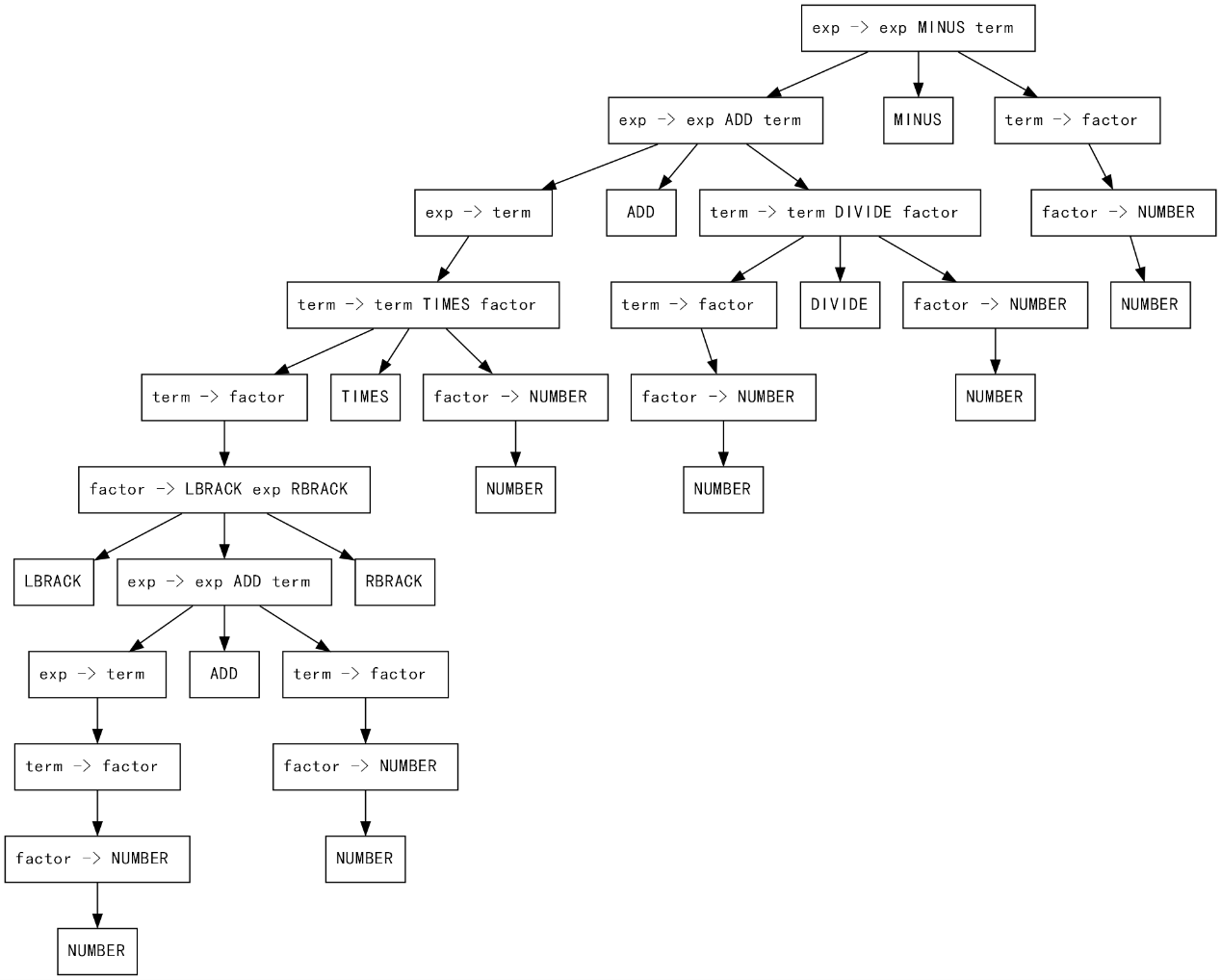
输入文件：



执行结果：



语法树：

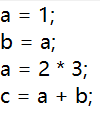


**结果分析：**

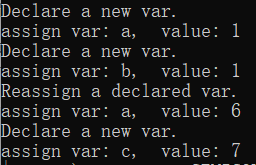
解析程序正确地输出算术表达式计算结果，实现了语义动作执行。

## 5.symbol\_chart\_demo

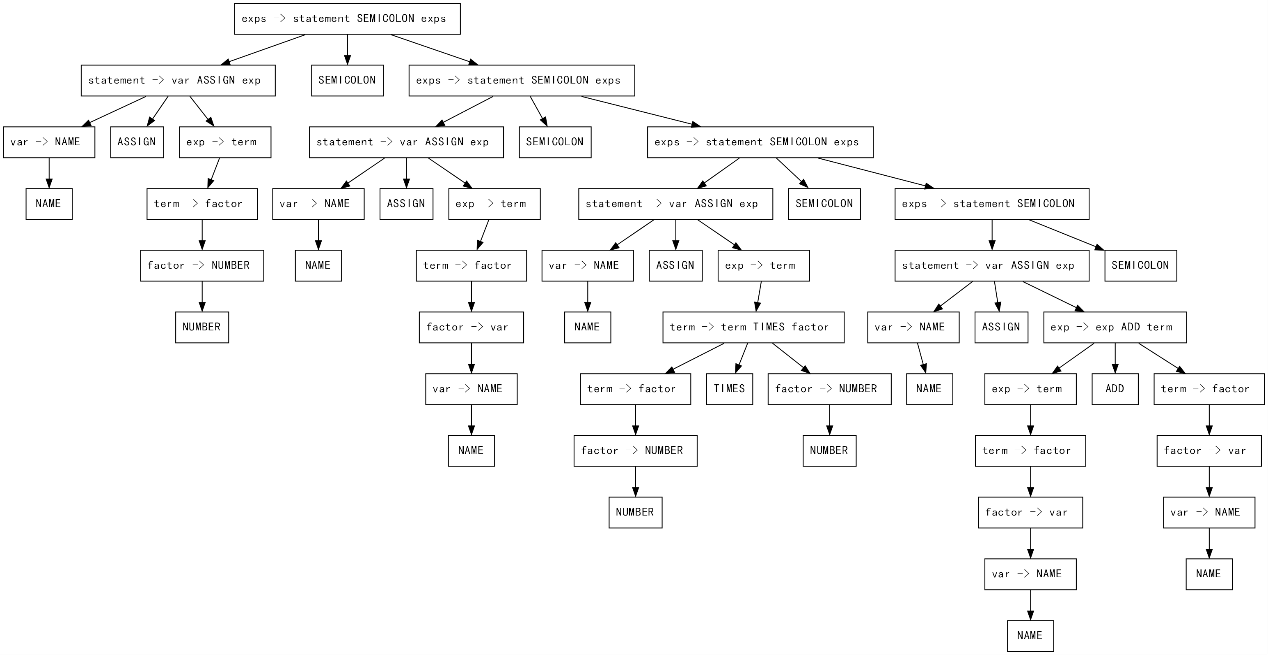
输入文件：



执行结果：



语法树：



**结果分析：**

当定义变量或对变量重新赋值时，会输出相应提示，并且可以验证得出，被定义的变量在后续代码中可以被正确地引用，并取到事先所赋的值。因此用户可以通过自定义.y文件实现符号表。

# 六、总结

作为一个单人完成的项目，能完成到这里我觉得很不容易。十分感谢17级学长的开源分享，没有它们的项目参考我在构建项目以及面临一些问题时很可能无从下手。也感谢老师提供的参考书籍还有graphviz可视化工具，帮我了解了flex和bison的特性，可视化也方便了调试。在项目中，我尽可能运用我所了解的软件开发规范，并借助版本控制工具git记录开发过程，这也是我一次难得的项目实践经验。项目开发中，我亲自把课上学到的各种算法用代码实现出来，只有动手实现才会发现算法中的一些细节，这是做练习题时感受不到的。比如求first集时我遇到了左递归问题，在做题时文法规则简单，并不会仔细考虑，但程序设计中却是一个必须解决的问题，在参考了网上资料后，我了解到应该先求非终结符的first集是否含epsilon，再去解决左递归问题。在实现时，我也用到了Java中的各类数据结构，让我更加熟悉Java的特性。比如当把自定义的LR1ItemCore作为HashMap的key时，equals()和hashCode()方法都应该进行重写。同时我也使用编码技巧简化了开发，如把非终结符编号设为负数，终结符设为非负数，同样规约动作在action表中是负值，移进动作是非负值，在判断时就较为方便。

在完成项目后，我对编译原理的理解又加深一步，不过，仍有很多知识我还没有触碰。我们现在学习的内容主要还是在前端，在项目中我也尚未考虑中间代码和目标代码生成的部分，而后端优化是现在编译领域更有价值的研究方向。希望自己将来可以继续向底层探进，了解编译器工作的全貌。