编译原理程序作业
报 告

姓名：高谦

班级：计科 1601

学号：2016014302

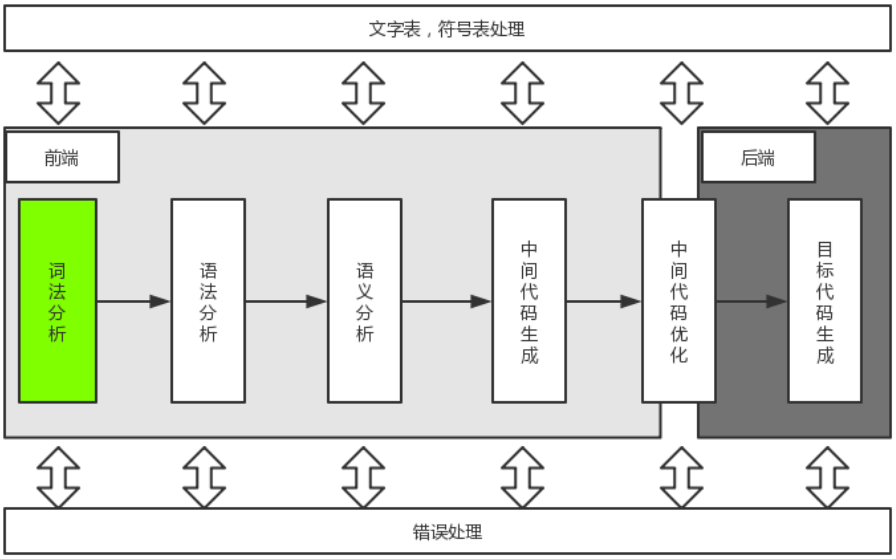
指导教师：胡伟

目录

提示：本文档有清晰的标题索引，使用 word 打开标题栏目，可以方便的跳转查看！

词法分析

词法分析地位：



词法分析作用：读入输入字符，产生记号序列，供语法分析使用

词法分析的过程：

正则表达式-> NFA->DFA->最小化 DFA->用最小化的 DFA 进行分析

编程实现

题目要求：

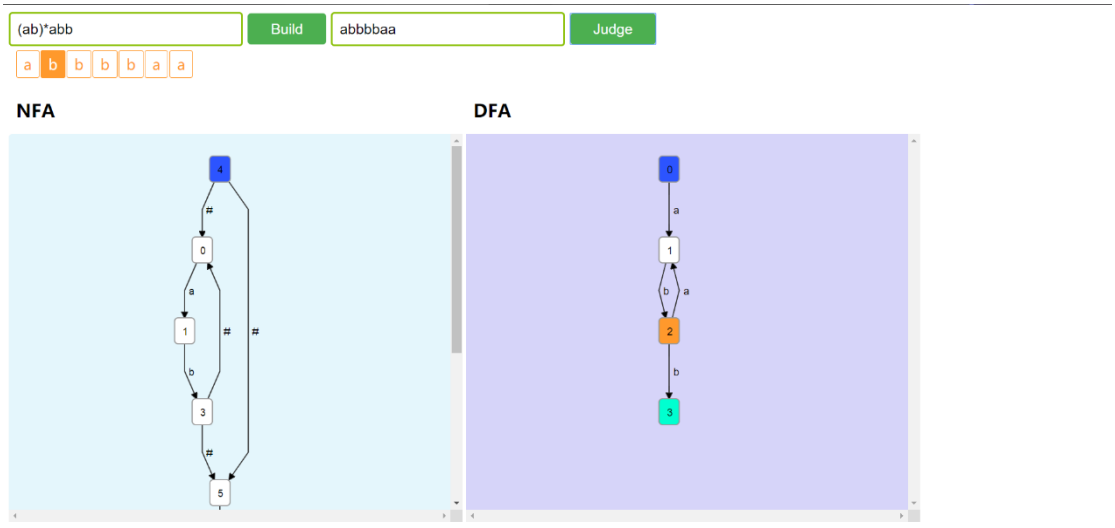
分析用户的输入的正则表达式，（包含基本运算）对操作进行理解
将正则表达式转换到 NFA
将 NFA 转换到 DFA
将 DFA 转换到最小化 DFA
利用最小化 DFA 进行分析
附加要求：程序可以实现交互性

最终实现效果展示：

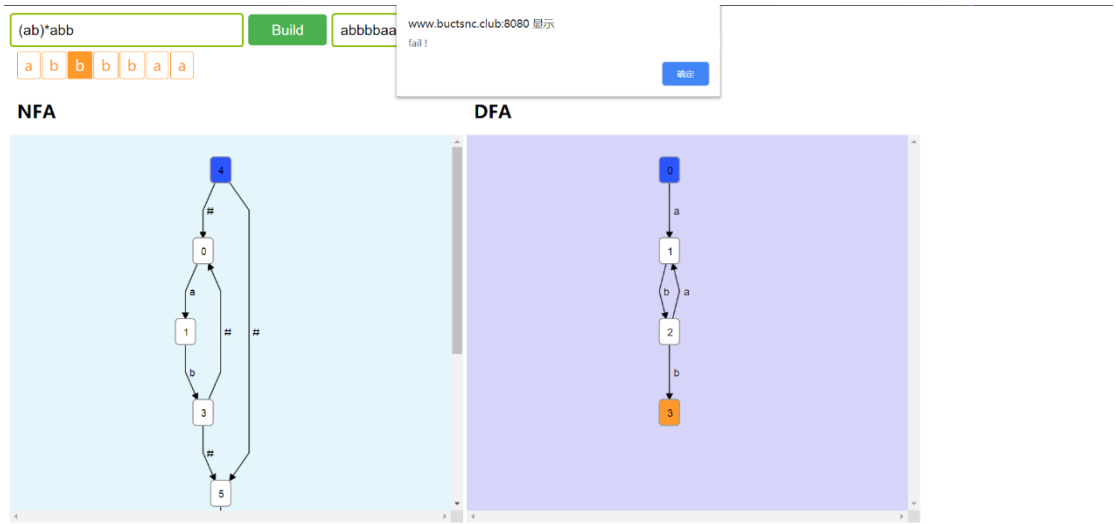
最终的效果是一个 web 网页，输入正则表达式点击 build 后端 Java 程序构建 NFA 和最小化 DFA 并发送数据到前端，运用 JavaScript 生成图片，输入要分析的字符串，点击 judge 即可进行匹配分析，分析过程中会把分析的过程详细的展示出来，包括输入指针所在的位置，DFA 所在的状态，采用全自动刷新，只需要等待程序运行出来结果即可。程序采用的是麻省理工学院的开源 JavaScript 图形布局算法包，运用 Java web 架构进行后台的开发。

展示图：

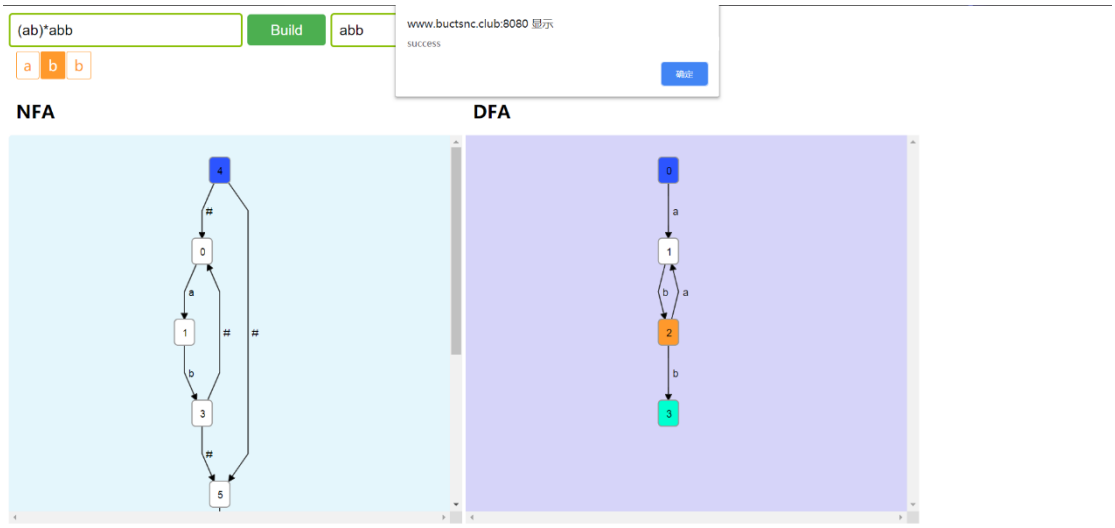
程序匹配运行截图。



程序失败的结果运行图：



程序成功的结果运行图：



程序亮点：

1）可以实现 DFA 和 NFA 的清晰图形化展示，方便结果的验证

2）界面大方的网页，跨平台使用没问题，只需输入网址即可在线使用，方便快捷

3）实现了用户的交互功能，匹配输入的字符串的时候采取输入指针和 DFA 状态同步变化的功能，使得分析的过程，分析的结果更加清晰化，可以作为非常好的教学示范工具来讲解词法分析。

程序具体编写过程（只包含算法部分）：

一、从正则表达式到 NFA

1. 构建字母表

只需要扫描一遍字符串，即可得出这个表达式都有哪些字母

2. 将正则表达式补充完整：

部分正则表达式没有”.”操作符号，所以要补充完整才可以进行分析

补充完整部分的代码如下：

//得到完整正则表达式的判断函数：

```
//得到完整正则表达式的判断函数：
public Boolean isleft(char s)
{
```

```
if(s==')'||s>='a'&&s<='z'||s=='*'||s>='A'&&s<='Z'||s=='_'||s>='0'&&s<='9')
    {
        return true;
    }
    return false;
}
public Boolean isright(char s)
{
    if(s>='a'&&s<='z'||s=='('||s=='_'||s>='A'&&s<='Z'||s>='0'&&s<='9')
    {
        return true;
    }
    return false;
}
//得到包括"."的完整正则表达式：
public String getRE(String s)
{
    StringBuffer res=new StringBuffer();
    res.append(s.substring(0, 1));
    for(int i=1;i<s.length();i++)
    {
        if(isleft(s.charAt(i-1))&&isright(s.charAt(i)))
        {
            res.append(".".toString());
            res.append(s.substring(i,i+1));
        }
        else
            res.append(s.substring(i,i+1));
    }
    System.out.println("得到的完整的表达式是：");
    System.out.println(res.toString());
    return res.toString();
}
```

以上代码写的是什么时候需要加一个点运算符表示连接运算符，只有当左边是右括号，字母，数字，下划线或者*号，并且右边是左括号，字母，数字或者下划线才在两者之间插入一个连接符号，这样就可以生成完整的正则表达式。

3. 构建逆波兰表达式

这里运用的是调度场算法，算法完整代码如下：

```
//求后缀表达式的比较函数：
public int adv(char s)
{
    if(s=='(')
        return -1;
    if(s=='*')
        return 4;
    if(s=='.')
        return 2;
    if(s=='|')
        return 0;
    return -1;
}
//获得后缀表达式
public String getPostfix(String s)
{
    StringBuffer res =new StringBuffer();
    Stack<String> stack=new Stack<String>();
    for(int i=0;i<s.length();i++)
    {
        if(isInput(s.charAt(i)))
        {
            res.append(s.substring(i, i+1));
        }
        else
        {
            if(s.charAt(i)=='(')
```

```
            {
                stack.push(s.substring(i,i+1));
                continue;
            }
            if(s.charAt(i)==')')
            {
                String temp=stack.peek();
                while(!temp.equals("("))
                {
                    res.append(temp);
                    stack.pop();
                    if(stack.empty())
                        break;
                    temp=stack.peek();
                }
                stack.pop();
                continue;
            }
            if(stack.empty())
            {
                stack.push(s.substring(i,i+1));
            }
            else
            {
                String temp=stack.peek();
                while(adv(temp.charAt(0))>=adv(s.charAt(i)))
                {
                    res.append(temp);
                    stack.pop();
                    if(stack.empty())
                        break;
                    temp=stack.peek();
                }
                stack.push(s.substring(i,i+1));
```

```
            }
        }
    }
    while(!stack.empty())
    {
        String temp=stack.peek();
        res.append(temp);
        stack.pop();
    }
    return res.toString();
}
```

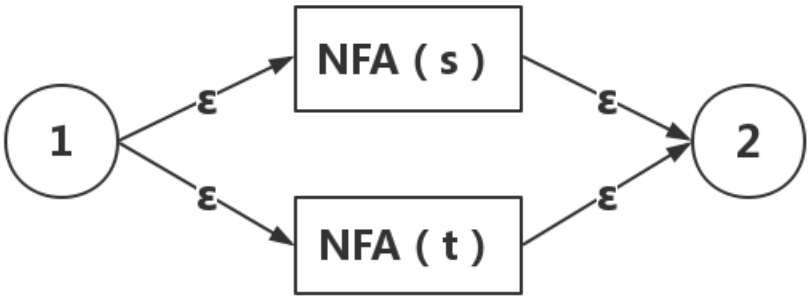
求后缀表达式的比较函数规定了各个正则表达式运算符的优先级，在这里，*号运算符优先级最大，连接运算符次之，最小的是|运算符。

构建逆波兰表达式的时候需要一个栈的数据结构，具体思路是如果当前输入符号不是一个运算符或者括号，直接加到结果后缀表达式中，如果是运算符，则要根据优先级别进行判断，然后进行出栈和入栈操作，具体操作如下：

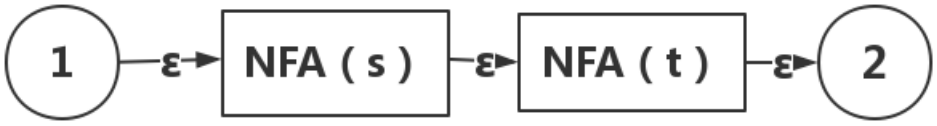
1) 如果当前运算符是左括号，直接入栈。
2) 如果当前是右括号，依次出栈，并将出栈的元素加到结果后缀表达式的后边，直到当前栈顶元素是左括号，直接出栈左括号，输入指针向前，判断下一个输入字符
3) 如果当前运算符优先级大于栈顶的运算符优先级，直接入栈，输入指针前进，如果小于等于当前栈顶的运算符，一直出栈，并将出栈元素加入结果后缀表达式中，直到不满足栈顶运算符优先级大于当期输入串运算符号为止。
4) 特别注意处理栈空的情况，做好判断才可以得出运算结果。

4.正则表达式转换为 NFA 的方法：

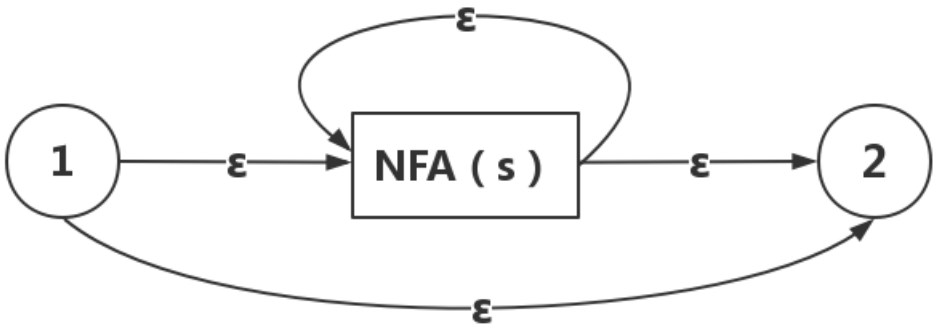
1）对于形如 R=s|t 的表达式，NFA（R）如下：



2）对于形如 R=st 的表达式，NFA（R）如下：



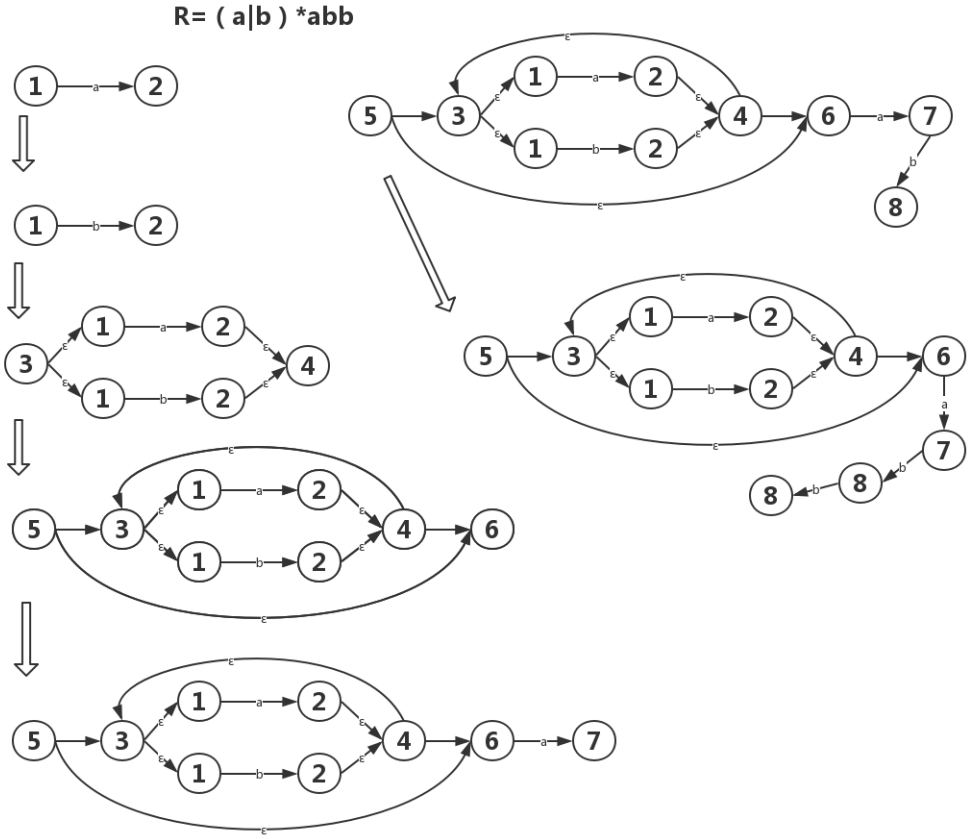
3）对于形如 R=s*的表达式，NFA（R）如下：



5.例子讲解：

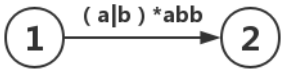
为正则表达式 R=（a|b）*abb 构造一个 NFA

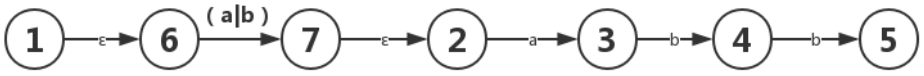
方法一：（推荐编程使用）



方法二（推荐考试使用）

R = (a|b) *abb





6.逆波兰表达式转换为 NFA 方法

1）遇到字母，构建基本的 NFA 入栈，也就是例子讲解中第一种法法的第一个式子和第二个式子，都是基本的 NFA，构建一个栈用来存储基本 NFA。

2）遇到运算符，根据运算符的结合性，判断出栈的数目，比如*是一目运算符，只出栈一个基本 NFA，连接和|运算符都是双目运算符，出栈两个，构建好 NFA 后继续入栈，知道扫描完逆波兰表达式为止，这样，栈的顶端（其实此时栈中只有一个元素）存储的就是构造好的 NFA。

构造 NFA 的算法代码如下：

```
public stackNode getNFA(String s) {
    Stack<stackNode> stack=new Stack<stackNode>();
    for(int i=0;i<s.length();i++)
```

```
{
    //构造单个字母的状态转换图
    if(isInput(s.charAt(i)))
    {
        Node a;
        stackNode stacknode;
        a = new Node();
        stacknode=new stackNode();
        stacknode.left=a;
        Node b=new Node();
        Line newline=new Line();
        newline.next=b;
        newline.tranval=s.substring(i, i+1);
        a.line.add(newline);
        stacknode.right=b;
        stack.push(stacknode);
    }
    else
    {
        if(s.charAt(i)=='|')
        {
            //要进行 "|"运算的两个 NFA;
            stackNode node1=stack.pop();
            stackNode node2=stack.pop();
            //两个新的状态
            Node newnode1=new Node();
            Node newnode2=new Node();
            //四条新边
            Line newline1=new Line();
            Line newline2=new Line();
            Line newline3=new Line();
            Line newline4=new Line();
            //右边相连
            newline3.next=newnode2;
```

```
            newline3.tranval="ε".toString();
            newline4.next=newnode2;
            newline4.tranval="ε".toString();
            node1.right.line.add(newline3);
            node2.right.line.add(newline4);
            //左边相连：
            newline1.next=node1.left;
            newline1.tranval="ε".toString();
            newline2.next=node2.left;
            newline2.tranval="ε".toString();
            newnode1.line.add(newline1);
            newnode1.line.add(newline2);
            //创造新的压栈元素，并压栈。
            stackNode newstacknode=new stackNode();
            newstacknode.left=newnode1;
            newstacknode.right=newnode2;
            stack.push(newstacknode);
            continue;
        }
        if(s.charAt(i)=='.')
        {
            //这里的顺序要搞清楚。
            stackNode node2=stack.pop();
            stackNode node1=stack.pop();
            for(int j=0;j<node2.left.line.size();j++)
            {
                Line newline=new Line();
                newline.next=node2.left.line.get(j).next;
                newline.tranval=node2.left.line.get(j).tranval;
                node1.right.line.add(newline);
            }
            /*Line newline=new Line();
            newline.next=node2.left;
            newline.tranval="ε".toString();
```

```
            node1.right.line.add(newline);*/
            node1.right=node2.right;
            stack.push(node1);
            continue;
        }
        if(s.charAt(i)=='*')
        {
            stackNode node=stack.pop();
            Line newline=new Line();
            Line newline1=new Line();
            Line newline2=new Line();
            Line newline3=new Line();
            newline.next=node.left;
            newline.tranval="ε".toString();
            node.right.line.add(newline);
            Node newnode1=new Node();
            Node newnode2=new Node();
            newline1.next=node.left;
            newline1.tranval="ε".toString();
            newnode1.line.add(newline1);
            newline2.next=newnode2;
            newline2.tranval="ε".toString();
            node.right.line.add(newline2);
            newline3.next=newnode2;
            newline3.tranval="ε".toString();
            newnode1.line.add(newline3);
            node.left=newnode1;
            node.right=newnode2;
            stack.push(node);
        }
    }
}
stackNode node =stack.pop();
```

```
    return node;
}
```

二、NFA 转换为 DFA

1.转换方法：

1） λ（ε）合并（如果有 S1 通过空串可以到 S2,则把 S2 状态合并到 S1 状态）
2） 符号合并。

2.基本概念：

令 I 是一个状态集的子集，定义 ε-closure（I）为：
1）若 s∈I，则 s∈ ε-closure（I）;
2）若 s∈I，则从 s 出发经过任意条 ε 弧能够到达的任何状态都属于 ε-closure（I）
状态集 ε-closure（I）称为 **I 的 ε-闭包**
令 I 是 NFA M’的状态集的一个子集，a∈Σ **定义：** **Ia= ε-closure(J)**
其中 J = ∪T(s,a)
1）J 是从状态子集 I 中的每个状态出发，经过标记为 a 的弧而达到的状态集合。
2）Ia 是状态子集,其元素为 J 中的状态，加上从 J 中每一个状态出发通过 ε 弧到达的状态。

3.实现代码：

（部分函数省略，请看注释）

```
public ArrayList<DFAStatus []> getDFA(Node head,Node bottom)
{
    //首先，找到起始状态的ε闭包，作为一个新的起点。
    TreeSet<Integer> Start=new TreeSet<Integer>();
    getClosure(head.num,Start);
```

```
Start.add(head.num);
DFAStatus start=new DFAStatus(Start);
start.initSymble();
DFAStatus [] row=new DFAStatus[Head.size()+1];
row[0]=start;
ArrayList<DFAStatus []> DFAform=new ArrayList<DFAStatus[]>();
DFAform.add(row);
for(int i=0;i<DFAform.size();i++)
{
    for(int j=0;j<Head.size()-1;j++)
    {
        TreeSet<Integer> elem=new TreeSet<Integer>();
        //找到当前状态开始。
        Iterator it=DFAform.get(i)[0].closure.iterator();
        while(it.hasNext())
        {
            Integer newnum=Integer.parseInt(it.next().toString());
            getClosure(newnum,Head.get(j),elem);
        }
        if(elem.size()==0) {
            TreeSet<Integer> Null=new TreeSet<Integer>();
            DFAStatus NULL=new DFAStatus(Null);
            NULL.setSymble(-1);
            DFAform.get(i)[j + 1]=NULL;
        }
        else
        {
            DFAStatus status=new DFAStatus(elem);
            Boolean flag=true;
            for(int k=0;k<DFAform.size();k++)
            {
                if(DFAform.get(k)[0].isEqual(status))
                {
                    status.setSymble(DFAform.get(k)[0].symble);
```

```
                    DFAform.get(i)[j+1]=status;
                    flag=false;
                    break;
                }
            }
            if(flag)
            {
                status.initSymble();
                DFAform.get(i)[j+1]=status;
                DFAStatus [] newrow=new DFAStatus[Head.size()+1];
                newrow[0]=status;
                DFAform.add(newrow);
            }
        }
    }
    for(int i=0;i<DFAform.size();i++)
    {
        DFAform.get(i)[0].judgeAcceptable(bottom.num);
    }
    DFAFORM=DFAform;
    getSimpleDfaArray();
    getSimpleDfaArray();
    return DFAform;
}
```

三、DFA 的最小化

一个有穷自动机可以通过消除多余的状态和状态合并转换为一个与之等价的最小的有穷自动机。

多余状态：从有穷自动机的初始状态开始，不能到达的状态成为多余状态。

1. 等价状态：

两个状态是等价状态的条件是：

1）一致性条件

状态 s 和状态 t 必须同时为可接受状态或者不可接受状态才有可能等价。

2）蔓延性条件

状态 s 和状态 t 对于所有的输入符号，都必须转移到等价的状态里面。

2.最小化 DFA 的方法：（分割法）

把一个 DFA(不含多余状态)的状态分割成一些不相关的子集，使得任何不同的两个子集状态都是可区别的，而同一个子集中的任何状态都是等价的.初始时先将 DFA 的终态和非终态划分为两个子集，然后分别从两个子集中寻找等价状态进行合并。

3. 最小化 DFA 实现代码（部分代码省略，请看注释）

```
public ArrayList<Block> getMinDFA() {
    //将原始的集合划分为可接受集合和不可接受集合：
    Block block1=null;
    Boolean havaBlock1=false;
    for(int i=0;i<DFAFORM.size();i++)
    {
        if(!simpleDFA[i][0].isAccept){
            block1=new Block();
            havaBlock1=true;
            break;
        }
    }
    Block block2=new Block();
    if(havaBlock1) {
        for (int i = 0; i < DFAFORM.size(); i++) {
            if (simpleDFA[i][0].isAccept) {
                block2.addStatus(simpleDFA[i]);
```

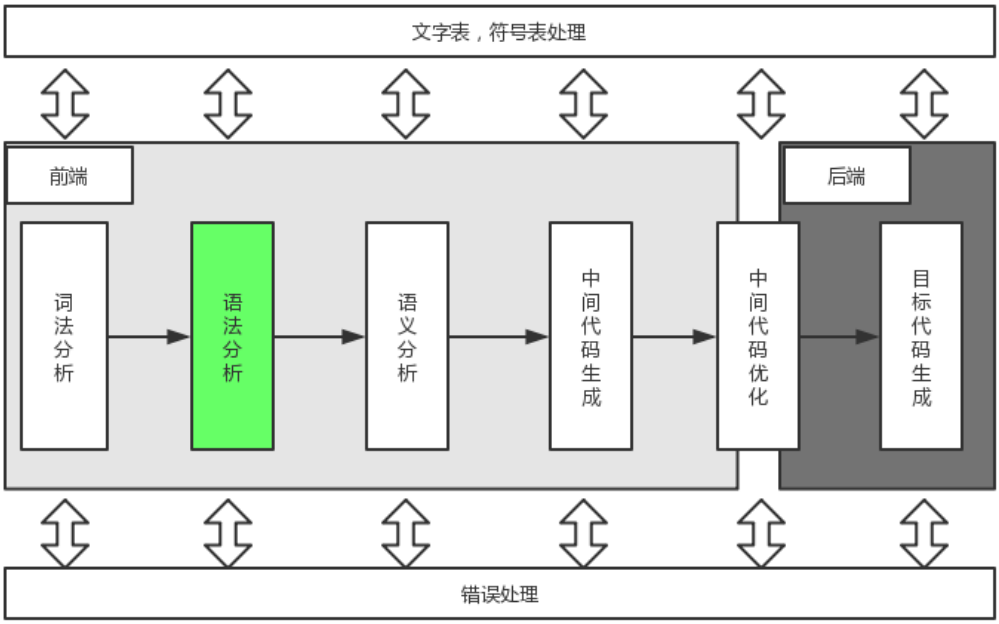
```
        } else {
            block1.addStatus(simpleDFA[i]);
        }
    }
}
else
{
    for(int i=0;i<DFAFORM.size();i++)
        block2.addStatus(simpleDFA[i]);
}
//初始化列表，只有两个区，可接受区和不可接受区域。
ArrayList<Block> BlockSet=new ArrayList<Block>();
if(block1!=null)
    BlockSet.add(block1);
BlockSet.add(block2);
System.out.println("划分为终态和非终态：");
PrintBlock(BlockSet);
System.out.println("每一次的划分信息如下：");
int i=0;
//遍历这些区块：
while(i<BlockSet.size())
{
    //如果当前区块只有一个状态：直接看下一个区块：
    if(BlockSet.get(i).statusSet.size()==1)
    {
        i++;
        continue;
    }
    //如果区块中有多个状态，寻找等价状态：
    else {
        Integer index = 1;
        StringBuffer sb = new StringBuffer();
        SimpleDFA[] begin = BlockSet.get(i).statusSet.get(0);
        //将每一个状态通过输入符号转换后的状态所在的块号存入一个字符串中：
```

```
            for (int j = 1; j < begin.length; j++) {
                sb.append(begin[j].getBlocknum(BlockSet).toString());
            }
            //将这个字符串与剩下的状态生成的字符串进行比较：如果相同的话说明两个状态暂时还是一个等价的状态
            //如果不同的话需要将这一个区块划分为两个区块，划分的位置就是当前位置。
            for (int j = 1; j < BlockSet.get(i).statusSet.size(); j++) {
                SimpleDFA[] row = BlockSet.get(i).statusSet.get(j);
                StringBuffer sb2 = new StringBuffer();
                for (int k = 1; k < row.length; k++) {
                    sb2.append(row[k].getBlocknum(BlockSet).toString());
                }
                //如果相同，就直接继续查找不同的状态出现的地方。
                if (sb.toString().equals(sb2.toString())) {
                    index++;
                    continue;
                }
                //如果不同，找到了划分的位置，就可以进行划分了：
                else {
                    break;
                }
            }
            //这个说明当前块内的所有状态都是等价状态，不需要划分：
            if (index == BlockSet.get(i).statusSet.size()) {
                i++;
            }
            //如果需要划分的话：
            else {
                Block newblock = new Block();
                for (int m = 0; m < index; m++) {
                    newblock.addStatus(BlockSet.get(i).statusSet.get(m));
                    BlockSet.get(i).deleteStatus(m);
                }
                newblock.setNum(i);
```

```
                //BlockSet.add(i, newblock);
                ArrayList<SimpleDFA[]> newstatusset = new
ArrayList<SimpleDFA[]>();
                for (int m = index; m < BlockSet.get(i).statusSet.size();
m++) {
                    newstatusset.add(BlockSet.get(i).statusSet.get(m));
                }
                BlockSet.get(i).statusSet = newstatusset;
                for (int q = i; q < BlockSet.size(); q++) {
                    BlockSet.get(q).setNum();
                }
                BlockSet.add(i,newblock);
                i = 0;
            }
        }
        System.out.println("*******************************");
        PrintBlock(BlockSet);
    }
    System.out.println("最终划分的结果是：");
    PrintBlock(BlockSet);
    return BlockSet;
}
```

语法分析：

语法分析地位



程序完成情况汇报：

1. 完成了所有语法分析的程序：包括 LL1，LR0，SLR1，LR1 和 LALR1
2. 语法分析基本要求的程序在程序检查课的时候全部运行正确（结果已发群里，下面各部分的讲解也会附上结果图片）。
3. 附加分数的检查部分（LR1 和 LALR1）由于输入格式不友好，只检查了两个，7.3 和 7.2，结果全部正确。之后课下自己检查发现 LL1 存在问题，现在已经修复，并且在最后一次课程给老师演示了全部附加分数要求的题目，全部运行正确。
4. 程序采用的是控制台输出，保证结果的正确性（老师出的全部题目现在可以全部运行正确）和程序完成度很高（五个程序全部完成）的情况下确实没有时间做优美的图形界面，计划之后有时间做。（形式和词法分析一样，清晰展示，可以交互）。

下面将分开展开各个程序编写的思路，运行截图以及编程序时遇到的问题和解决办法。

第一部分：LL1 语法分析

一、LL1 文法分析概述：

LL1 分析方法使用自顶向下的分析方法，使用显示的栈来实现分析，而不是地柜调用的方法,，具体步骤可以分为修改文法（消除左递归，提取左因子），求取 first 集合，求取 follow 集合，求取 LL1 分析表，根据分析表进行分析。

二、具体过程和程序实现

1.消除左递归

左递归的意思是形如 exp → exp addop term | term 的产生式，产生式右端的第一个符号还是左端的非终结符号，这样的话会导致循环替换，不能结束，最终导致栈溢出的错误，需要修改文法。左递归可以分两部分讲，简答直接左递归和一般左递归，下面分别展开详细叙述：

直接左递归

直接左递归是一类比较简单，可以直接被看出来的左递归方式。
形如：A → Aα | β
这一类左递归消除的办法就是将文法做如下修改：

```
A → βA’
A’ → αA’| ε
```

这里需要特别注意的是千万不要把空串忘记了，否则前功尽弃！

一般左递归

一般左递归是含有多个产生式，多个非终结符号，他们之间可能存在着左递归，需要哟弄一定的策略来消除所有隐含的左递归
形如：

```
A→Ba| Aa| c
B→Bb| Ab| d
```

消除的方法如下：

首先消除 A 的直接左递归，这很简单，结果为：

```
A→BaA’| c A’
A’→aA’| ε
B→Bb| Ab| d
```

A 的简单直接左递归消除完毕后要及时替换 B 中所有的产生式中的 A 为 A 左递归消除之后的右边的产生式，否则的话可能还会有间接左递归的存在。

将 B 的所有产生式的右边的所有 A 替换之后的结果为：

```
A→BaA’| c A’
A’→aA’| ε
B→Bb| BaA’b|cA’b| d
```

这样就不会存在间接左递归了，只要继续仿照上面的过程处理简单直接左递归即可，处理的结果如下：

```
A→BaA’| c A’
A’→aA’| ε
B→cA’bB’| dB’
B’→bB’ |aA’bB’|ε
```

例子以两个产生式为例，实际上多个产生式用一个循环加上判断就可以得出正确的结果。

实现代码讲解：

```
def RemoveLeftRecursion(gramma):
    print("消除左递归：")
    i=0
    length=len(gramma)
    while i<length:
        j=0
        while j<i:
            k =0
            current=gramma[i].right
            productionlen=len(current)
            newright=[]
            while k<productionlen:
                if current[k][0]==gramma[j].value:
                    for sub in gramma[j].right:
                        newright.append(sub+current[k][1:])
                else:
                    newright.append(current[k])
                k+=1
            gramma[i].right=newright
```

```
            j+=1
        a=[]
        b=[]
        for production in gramma[i].right:
            if production[0]==gramma[i].value:
                a.append(production)
            else:
                b.append(production)
        # 存在左递归
        if len(a)!=0:
            for temp in b:
                temp.append(gramma[i].value+'~')
            for temp in a:
                temp.remove(gramma[i].value)
                temp.append(gramma[i].value+'~')
            a.append(["ε"])
            newlabel=label(gramma[i].value+'~',a)
            gramma[i].right=b
            gramma.append(newlabel)
        i+=1
    gramma.sort(key=label.op)
    printGramma(gramma)
    return gramma
```

2. 提取左因子

左因子：

形如 if-stmt → if (exp) statement

| if (exp) statement else statement 的产生式，可以看到，产生式右端的两个式子存在着公共的因子 if（exp）statement，可以将两者的公共部分提取出来，修改文法得到如下结果：

if-stmt →if（exp）statement other

other→else statement | ε

代码实现：

```
def RemoveLeftFactor(gramma):
    print("提取左因子: ")
    rownum=0
```

```
    length=len(gramma)
    for row in gramma:
        if rownum>=length:
            break
        minlen=len(row.right[0])
        for i in row.right:
            if len(i)<minlen:
                minlen=len(i)
        ml=0
        while ml<minlen:
            if not judge(row.right,ml):
                break
            ml+=1
        if ml!=0:
            substr=row.right[0][0:ml]
            newright=[substr+[row.value+'^']]
            newelemright=[]
            for right in row.right:
                if len(right[ml:])==0:
                    newelemright.append(["ε"])
                else:
                    newelemright.append(right[ml:])
            gramma.append(label(row.value+'^',newelemright))
            gramma[rownum].right=newright
        rownum+=1
    printGramma(gramma)
    return gramma
```

3.First 集合的获取

定义：

FIRST(α) = {a | α => * aβ， a T， α,β (T N)* }若 α => * ε，则规定 ε FIRST(α) 该集合称为α的头符号集合

通俗解释和计算方法：

1）终结符和非终结符都有 first 集合，终结符的 first 集合就是它本身，非终结符的 first 集合要看它存在的所有产生式。

2）假设存在一个产生式，A->BCDE,现在要求 A 的 first 集合，首先可以确定的是 A 的 first 集合中包含 B 的 first 集合减去空串，即 first（A）中包含 first（B）-{ε}，还没有结束，需要判断一下 first（B）中有没有空串，有的话说明 first（A）中还应该包含 first（C）-{ε}，一直这么判断，如果 BCDE 的 first 集合中都包含空串的话，那么可以确定 first（A）中存在空串。

代码实现：

```
def getFirst(gramma,tset,nset):
    print("获取 First 集合")
    First={}
    for i in tset:
        First[i]=[i]
    for i in nset:
        First[i]=[]
    loop=True
    while True:
        if not loop:
            break
        loop = False
        for Nlabel in gramma:
            for production in Nlabel.right:
                Continue=True
                n=len(production)
                labelindex=0
                while Continue and labelindex<n:
                    if len(First[production[labelindex]])==0:
                        Continue=False
                    else:
                        res,Continue=ExceptNull(First[production[labelindex]])
                        First[Nlabel.value],ischange=add(First[Nlabel.value],res)
                        if ischange:
                            loop=True
                    labelindex+=1
                if Continue:
                    First[Nlabel.value],ischange=add(First[Nlabel.value],["ε"])
        if not loop:
            continue
        for i in First:
            print(i,end=" : ")
            print(First[i])
```

```
        print("++++++++++++++++++++++++++++++++")
    printFirst(First,nset)
    print("----------------------------------")
    return First
```

4.Follow 集合的获取：

follow 集合只针对非终结符集合，终结符没有 follow 集合（因为终结符不会出现在产生式的左边）

算法描述：

不断应用下面的规则，知道没有新的终结符可以被加入到任何 follow 集合中为止：

1）将开始符号的 follow 集合中加入$符号。

2）如果存在一个产生式 A->aBC，则 follow（B）中应该包含 first（C）-{ε}。

3）如果存在一个产生式 A->aBC 且 follow(C)包含空串,则可以确定的是 follow（B）包含 follow（A）的全部内容。

算法实现：

根据以上描述，可以很简答的写出得到 follow 集合的算法：

```
def getFollow(gramma,tset,nset,First,start):
    print("得到 Follow 集合")
    Follow={}
    #初始化 Follow 集合：
    for i in nset:
        if i==start:
            Follow[i]=['$']
        else:
            Follow[i]=[]
    loop=True
    while True:
        if not loop:
            break
        loop=False
        for i in nset:
            for row in gramma:
                right=row.right
```

```
                for production in right:
                    labelindex=0
                    length=len(production)
                    while labelindex <length:
                        if production[labelindex]==i:
                            if labelindex==length-1:
                                # print("将产生式左端的 follow 集合添加到 i 的 follow 集
合")

Follow[i],ischange=add(Follow[i],Follow[row.value])
                                if ischange:
                                    loop=True
                            else:
                                # print("将 labelindex+1 的 First 集合减去空 添加到 i
的 follow 集合: ")

res,contain=ExceptNull(First[production[labelindex+1]])
                                Follow[i],ischange=add(Follow[i],res)
                                if ischange:
                                    loop=True
                                if labelindex+1==length-1 and contain:
                                    # print("将产生式左端的 follow 集合添加到 i 的
follow 集合")
                                    Follow[i], ischange = add(Follow[i],
Follow[row.value])
                                    if ischange:
                                        loop=True
                        labelindex+=1
    for i in Follow:
        print(i,end=" : ")
        print(Follow[i])
    print("----------------------------------")
    return Follow
```

5. 构建分析表

构建方法，LL1 分析表的每一行是一个非终结符，每一列是一个终结符，表格中内容是应用哪一条产生式进行替换

分析表构建方法：

对于每一个非终结符 A，以及这个非终结符具有的产生式： $A \rightarrow \alpha$.

1）对于 First(α)中的每个记号 a，添加 A→α 到 M[A,a].

2）如果 ε 在 First(α)中，对于 Follow(A) 中每个元素(a token or $)，添加 A→α 到 M[A,a].

代码实现：

```
def getTable(follow,first,gramma,nset,tset):
    table=[]
    head=["M[N,T]"]
    for i in tset:
        if i !="ε":
            head.append(i)
    head.append("$")
    table.append(head)
    productionSet={}
    i=0
    #构造 productionSet 集合:
    for row in gramma:
        for production in row.right:
            productionSet[i]=[row.value,production]
            i+=1
    for i in nset:
        newrow=[i]
        for temp in range(1,len(head)):
            newrow.append("")
        table.append(newrow)
        #遍历 first 集合:
        for elem in first[i]:
            # content 是在表格中填的内容
            content=0
            length=len(productionSet)
            #设置是否找对应的右边是终结符的产生式编号:
            flag=False
            # 遍历所有的产生式
            while content<length:
                #如果产生式的左边和这一行的表头是一样的:
                if productionSet[content][0]==i:
                    #如果产生式右边第一个就是要找的终结符元素:
                    if productionSet[content][1][0]==elem:
                        #如果这个终结符是空串，应该引入 Follow 集合:
                        if not head.__contains__(elem):
                            flag=True
                            for fo in follow[i]:
```

```
                            #这里应该加一个是否不是 LL1 文法的条件
                            if fo != "ε":
                                if newrow[head.index(fo)] != "":
                                    newrow[head.index(fo)] += "/" +
str(content)
                                    PrintProductionSet(productionSet)
                                    PrintTable(table)
                                    exit("这不是一个 LL1 文法！")
                                else:
                                    newrow[head.index(fo)] = str(content)

                        #不是空串的话，说明一下找到了，并把产生式编号填进表格：
                        else:
                            #这里应该判断一下是是不是 LL1 文法
                            if newrow[head.index(elem)]!="":
                                newrow[head.index(elem)] += "/"+str(content)
                                PrintTable(table)
                                PrintProductionSet(productionSet)
                                exit("这不是一个 LL1 文法！")
                            newrow[head.index(elem)]=str(content)
                            flag=True
                            #如果没有这个 break，默认使用多个候选产生式的第一个产生式：
                            #break
                content+=1
            #如果没有找到匹配的右边是终结符的产生式。
            if not flag:
                content=0
                while content<length:
                    if productionSet[content][0]==i and \
                            productionSet[content][0] in nset and elem!="ε":
                        newrow[head.index(elem)]=str(content)
                        break
                    content+=1
    return (table,productionSet)
```

代码中加入了判断，如果不是 LL1 语法，构建分析表的时候会出现错误，程序会正常停止。

6. 分析过程和结果展示：

是 LL1 文法的情况：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LL1.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
S->( S ) S|ε
exit
解析输入文法:
('S', [['(', 'S', ')', 'S'], ['ε']])
--------------------------------
消除左递归:
('S', [['(', 'S', ')', 'S'], ['ε']])
--------------------------------
提取左因子:
('S', [['(', 'S', ')', 'S'], ['ε']])
--------------------------------
获取 First 集合
( : ['(']
) : [')']
ε : ['ε']
S : ['(', 'ε']
++++++++++++++++++++++++++++++++
S : ['(', 'ε']
------------------------------------
得到 Follow 集合
S : ['$', ')']
----------------------------------
产生式集合编号如下所示:
 0 : S -> ( S ) S
 1 : S -> ε
分析表如下:
----------+----------+----------+----------+
  M[N,T]  |    (     |    )     |    $     |
----------+----------+----------+----------+
    S     |    0     |    1     |    1     |
----------+----------+----------+----------+
请输入要匹配的字符串，输入 exit 结束输入,注意不同的输入要有空格
( )
分析栈:$ S
输入队列:( ) $
```

```
动作:S->( S ) S
--------------------------------
分析栈:$ S ) S (
输入队列:( ) $
动作: 匹配
--------------------------------
分析栈:$ S ) S
输入队列:) $
动作:S->ε
--------------------------------
分析栈:$ S ) ε
输入队列:) $
动作: 匹配
--------------------------------
分析栈:$ S )
输入队列:) $
动作: 匹配
--------------------------------
分析栈:$ S
输入队列:$
动作:S->ε
--------------------------------
分析栈:$ ε
输入队列:$
动作: 匹配
--------------------------------
分析栈:$
输入队列:$
动作: 接受
--------------------------------
```

不是 LR1 文法的情况：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LL1.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
S->A a|b A c|B c|b B a
A->d
B->d
exit
这不是一个 LL1 文法!
解析输入文法:
('S', [['A', 'a'], ['b', 'A', 'c'], ['B', 'c'], ['b', 'B', 'a']])
('A', [['d']])
```

```
('B', [['d']])
------------------------------
消除左递归：
('A', [['d']])
('B', [['d']])
('S', [['A', 'a'], ['b', 'A', 'c'], ['B', 'c'], ['b', 'B', 'a']])
------------------------------
提取左因子：
('A', [['d']])
('B', [['d']])
('S', [['A', 'a'], ['b', 'A', 'c'], ['B', 'c'], ['b', 'B', 'a']])
------------------------------
获取 First 集合
a : ['a']
b : ['b']
c : ['c']
d : ['d']
A : ['d']
B : ['d']
S : ['d', 'b']
++++++++++++++++++++++++++++++++
A : ['d']
B : ['d']
S : ['d', 'b']
------------------------------------
得到 Follow 集合
A : ['a', 'c']
B : ['c', 'a']
S : ['$']
------------------------------------
分析表如下：
----------+----------+----------+----------+----------+----------+
  M[N,T]  |    a     |    b     |    c     |    d     |    $     |
----------+----------+----------+----------+----------+----------+
    A     |          |          |          |    0     |          |
----------+----------+----------+----------+----------+----------+
    B     |          |          |          |    1     |          |
----------+----------+----------+----------+----------+----------+
    S     |          |   3/5    |          |    2     |          |
----------+----------+----------+----------+----------+----------+
产生式集合编号如下所示：
 0 : A -> d
 1 : B -> d
 2 : S -> A a
```

```
3 : S -> b A c
4 : S -> B c
5 : S -> b B a
```

如果有冲突，会在表中显示出冲突存在的位置

第二部分：LR0 语法分析

一、LR0 分析方法概述：

LR0 分析方法是一种自底向上的分析方法，通过构造识别文法活前缀的 DFA 来构造分析表，根据分析表来决定当分析栈和输入字符串已知的情况下选择移进和规约的操作。LR0 分析方法的过程包括扩展文法，构造识别文法活前缀的 DFA，构造 LR0 分析表，通过 LR0 分析表进行分析。由于之后要介绍的 SLR1，LR1，LALR1 中有很多步骤和 LR0 分析方法一致，故仅在 LR0 分析方法中介绍，之后的文法分析算法不再介绍这些过程。

二、具体过程和算法实现：

基础概念定义：

项目：含有”.”的产生式是一个项目，因点的位置的不同而不同，点后面如果是一个非终结符，则这个状态的项目还需要通过闭包运算来进行扩充运算。

1.扩展文法：

这一步非常的简单，只需要更换一个开始符号即可。例如：

形如

```
S → aA
A → cA | d
```

的产生式，扩展文法之后的文法如下：

```
S’→S
S → aA
A → cA | d
```

2.构造识别文法活前缀的 DFA

构造的方法可以按照书上的概念理解，也可以直接进行构建，构建的方法也很简单，这里采用简单构建的方法来进行叙述：

1）DFA 的每一个状态中都有很多条产生式，每一个状态都有一个唯一的标识，状态有很多的出边，出边指向另外一个状态。需要特别注意的是状态中每一个产生式并不是普通的产生式，产生式中还包括当前所处的位置信息，即 DFA 中的产生式是一个项目。

2）DFA 中每一个状态的产生式（项目）集合都是经过闭包运算形成的完整的项目集合，经过出边来实现跳转到另外一个状态。出边上有相应转移符号，表示当前状态经过这个符号之后会到达那个状态。

3. 构造 DFA 代码实现

```
def getDFA(productionset,tset,nset,gramma):
    start=Status()
    start.productionSet.append(Item(productionset[0][0],productionset[0][1]))
    #创造开始状态的所有产生式:
    for i in start.productionSet:  #遍历 item
        if i.right[i.index] in nset:
            #修复了产生式不断添加到 productionSet 的 bug，防止电脑蓝屏:
            godown=True
            for temp in start.productionSet:
                if temp.left==i.right[i.index]:
                    godown=False
            if not godown:
                continue
            for row in gramma:
                if row.value==i.right[i.index]:
                    for k in row.right:
                        start.productionSet.append(Item(row.value,k))
    start.initid()
    resultSet=[]
    resultSet.append(start)
    #这是构造的递归算法。
    getNextStatus(start,tset,nset,gramma,resultSet)
    print("识别文法活前缀的 DFA 状态集合: ")
    print("————————————————————————————")
    for i in resultSet:
        print("状态 id: "+str(i.static_id))
        print("状态产生式集合以及点所处的位置")
        for j in i.productionSet:
            print((j.left,j.right,j.index))
        print("状态的出边和所指向的状态标号")
        for k in i.line:
            print((k.tranval,k.next.static_id))
```

```
        print("--------------------------")
    return (resultSet,start)
```

getNextStatus 函数：

```
def getNextStatus(status,tset,nset,gramma,resultSet):
    # 首先判断是不是终止状态：status 的 production 中只有一个产生式，并且产生式的点在最右端
    # 是终止状态直接返回
    length=len(status.productionSet)
    currentpro=status.productionSet[0]
    if length == 1 and currentpro.index == currentpro.maxindex:
        return
    # 遍历当前状态 productionSet 中的所有 item
    # 根据点的位置创造出下一个状态
    # 用一条边将当前状态和下一个状态连接在一起（边的 next 指向下一个状态，边的值就是当前 index 所指的符号的值）
    # 将边添加到当前状态的出边集合中
    # 这里特别需要注意的是判断一下新的状态是不是可能就是当前状态，是的话就别创建了，如果不想 stackoverflow 的话
    i=0
    plength=len(status.productionSet)
    while i<plength:
        if status.productionSet[i].right[status.productionSet[i].index]=='ε':
            i+=1
            continue

isrepeat,newset,nextstatus=checkRepeat(resultSet,status.productionSet,i,gramma,nset,tset)
        #如果不是重复的话：
        if not isrepeat:
            #创建新的状态：
            newstatus=Status()
            newstatus.productionSet=newset
            newstatus.initid()
            #添加到最终的结果集合
            resultSet.append(newstatus)
            #创建指向新状态的边：

L1=Line(status.productionSet[i].right[status.productionSet[i].index],newstatus)
            # status.line.append(L1)
            status.addline(L1)
            getNextStatus(newstatus,tset,nset,gramma,resultSet)
        else:
            #如果是一个终结符号：
            if status.productionSet[i].right[status.productionSet[i].index] in
```

```
tset:

L1=Line(status.productionSet[i].right[status.productionSet[i].index],nextstatus
)
                # status.line.append(L1)
                status.addline(L1)
            else:
            # 如果是一个非终结符号：
            # 创建指向自己的一个边：

L1=Line(status.productionSet[i].right[status.productionSet[i].index],nextstatus
)
                # status.line.append(L1)
                status.addline(L1)
        i+=1
```

4.构造 LR0 分析表：

LR0 分析表的构建过程比较简单，LR0 分析表的每一行对应着识别文法或前缀的 DFA 的一个状态，每一列对应着文法的符号序列，分为两个部分，action 部分和 goto 部分，action 部分包括所有的终结符，goto 部分包括所有的非终结符（不包括扩展之后的符号）表格中的内容指的是从一个状态经过某一个符号要进行的操作，包括 S（移进操作）和 R（规约操作），S 后面跟的数字代表移进之后到达哪一个状态，R 后面跟的数字代表使用哪一个产生式进行规约操作。

构造算法：

1）如果当前状态只有一个产生式，并且产生式是一个完全项目（点的位置在最后面），则要进行规约操作，从文法的全部产生式中挑选出和当前状态完全项目具有相同形态的产生式，将 R+标号添加到表中。

2）如果当前产生式有多个状态，并且多个状态中没有完全状态，则可以确定这个状态要进行移进的操作，遍历这个状态的出边，到达新的一个状态，将 S+新状态标号添加到表格中，特别注意，当出边经过的符号是非终结符号的时候，使用的是 goto 规则，则填表的时候不用填 S+标号，直接填标号即可。

3）如果一个状态中既有完全产生式，也有不完全产生式，则可以说明存在移进和规约的冲突，程序可以停止运行并输出部分 LR0 分析表了。

5. 构造分析表代码实现：

```
def getTable(resultSet,tset,nset,productionSet):
    head = ["status"]
    for i in tset:
        if i == 'ε':
            continue
        head.append(i)
    head.append("$")
    for i in nset[1:]:
        head.append(i)
    table = []
    table.append(head)
    for i in resultSet:
        newrow = [i.static_id]
        for j in head[1:]:
            newrow.append("")
        table.append(newrow)
    # 遍历所有的状态
    row=1
    for status in resultSet:
        for index in range(len(status.line)):
            getshift2(status, table, row, index)
        for pro in status.productionSet:
            if pro.index==pro.maxindex or pro.right[0]=="ε":
                getreduce2(productionSet,table,row,pro)
        row+=1
    PrintTable(table)
    return table
```

6. 分析过程和结果：

成功的分析情况：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LR0.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
S->a A
A->c A|d
exit

拓广文法并给产生式编号:
 0 : S* -> S
```

```
 1 : S -> a A
 2 : A -> c A
 3 : A -> d
分析表如下：
----------+----------+----------+----------+----------+----------+----------+
  status  |    a     |    c     |    d     |    $     |    S     |    A     |
----------+----------+----------+----------+----------+----------+----------+
    0     |    s2    |          |          |          |    s1    |          |
----------+----------+----------+----------+----------+----------+----------+
    1     |          |          |          |   ACC    |          |          |
----------+----------+----------+----------+----------+----------+----------+
    2     |          |    s4    |    s6    |          |          |    s3    |
----------+----------+----------+----------+----------+----------+----------+
    3     |    r1    |    r1    |    r1    |    r1    |          |          |
----------+----------+----------+----------+----------+----------+----------+
    4     |          |    s4    |    s6    |          |          |    s5    |
----------+----------+----------+----------+----------+----------+----------+
    5     |    r2    |    r2    |    r2    |    r2    |          |          |
----------+----------+----------+----------+----------+----------+----------+
    6     |    r3    |    r3    |    r3    |    r3    |          |          |
----------+----------+----------+----------+----------+----------+----------+
请输入要进行分析的字符串：
a c c d
开始分析：
------------------------------
分析栈：$ s0
输入队列：a c c d $
动作：shift
------------------------------------
分析栈：$ s0 a s2
输入队列：c c d $
动作：shift
------------------------------------
分析栈：$ s0 a s2 c s4
输入队列：c d $
动作：shift
------------------------------------
分析栈：$ s0 a s2 c s4 c s4
输入队列：d $
动作：shift
------------------------------------
分析栈：$ s0 a s2 c s4 c s4 d s6
输入队列：$
动作：reduce: A->d
```

```
-----------------------------------
分析栈：$ s0 a s2 c s4 c s4 A s5
输入队列：$
动作：reduce: A->cA
-----------------------------------
分析栈：$ s0 a s2 c s4 A s5
输入队列：$
动作：reduce: A->cA
-----------------------------------
分析栈：$ s0 a s2 A s3
输入队列：$
动作：reduce: S->aA
-----------------------------------
分析栈：$ s0 S s1
输入队列：$
动作：reduce: S*->S
-----------------------------------
分析栈：$ s0
输入队列：$
动作：接受
-----------------------------------
请输入要进行分析的字符串：
a c
开始分析：
------------------------------
分析栈：$ s0
输入队列：a c $
动作：shift
-----------------------------------
分析栈：$ s0 a s2
输入队列：c $
动作：shift
-----------------------------------
分析栈：$ s0 a s2 c s4
输入队列：$
动作：op 异常:分析表中这个位置是空的
不可以接受！
```

失败的分析情况：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LR0.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
S->( S ) S|ε
exit
```

```
拓广文法并给产生式编号：
 0 : S* -> S
 1 : S -> ( S ) S
 2 : S -> ε
分析表如下：
----------+----------+----------+----------+----------+
  status  |    (     |    )     |    $     |    S     |
----------+----------+----------+----------+----------+
    0     |s2/reduce2|          |          |    s1    |
----------+----------+----------+----------+----------+
    1     |          |          |          |          |
----------+----------+----------+----------+----------+
    2     |          |          |          |          |
----------+----------+----------+----------+----------+
    3     |          |          |          |          |
----------+----------+----------+----------+----------+
    4     |          |          |          |          |
----------+----------+----------+----------+----------+
    5     |          |          |          |          |
----------+----------+----------+----------+----------+
存在冲突，程序停止运行！

Process finished with exit code 1
```

程序可以看很清楚的看出冲突存在的位置以及冲突的类型是移进-规约冲突

第三部分：SLR1 语法分析

一、SLR1 分析方法概述：

SLR1 分析算法和 LR0 分析算法过程类似，唯一不同的地方是构建分析表的时候，当要进行规约操作的时候，不是单纯的将这个状态所在行的全部列填入规约项目，而是引入了 follow 集合的概念，只有完成项左边非终结符的 follow 集合中的元素才进行规约操作，其他的地方空着，这样的话就允许一些文法一个状态中既有完成项目，也有非完全项目。由于 SLR1 分析方法和 LR0 分析方法的过程如出一辙，并且求取 follow 集合的过程也和前面的 LL1 分析方法一致（我直接 import 的之前写好的函数），所以这里不再详细赘述，LR0 和 SLR1 分析方法主要的不同点在于分析表的构建，所以只给出分析表的构建算法。

二、具体过程和程序实现：

1. 分析表的构建算法：

SLR 分析表的构建方法是先求出所有非终结符的 follow 集合，follow 集合中的终结符（不包含空串）在规约的时候要填内容的列对应的终结符号。因此相对于 LR0 算法来讲，最终的分析表不如 LR0 分析表密集，缺少的部分就是不在产生式左端非终结符号 follow 集合中的列。由于这个特性，允许一个状态中既有完全项目也有非完全项目（当然这只是部分文法才可以，还是没办法完全避免移进和规约的冲突，后面介绍的 LR1 算法可以更好的解决这些问题）

2. SLR 文法特点

1) S 状态中当前状态所有的非完成的产生式，下一个指向的终结符不包含在完成产生式的左端的 Follow 集合中。（没有移进和规约的冲突）
2) 两个完成产生式的左端的 Follow 集合交集为空，交集为空的话就没有规约冲突，有交集的话会产生冲突。
3) 例子：
   若有效项目集中存在冲突动作：

```
I =
{ X ->a.bC,
  A ->a.c,
  B ->a.
}
设当前输入符号为 a,
1. 若 a = b, 则移进;
2. 若 a 属于 Follow(A), 则用 A->a 进行归约;
3. 若 a 属于 Follow(B), 则用 B->a 进行归约;
4. 其余情况报错.
```

3. 实现代码：

```
def getTable(resultSet,tset,nset,productionSet,follow):
    #初始化表头:
    head=["status"]
    for i in tset:
```

```
        if i=='ε':
            continue
        head.append(i)
    head.append("$")
    for i in nset[1:]:
        head.append(i)
    #建立一个新表：
    table=[]
    table.append(head)
    for i in resultSet:
        newrow=[i.static_id]
        for j in head[1:]:
            newrow.append("")
        table.append(newrow)
    #开始在表中填内容：
    i=0
    length=len(resultSet)
    while i<length:
        currentstatus=resultSet[i]
        Set=currentstatus.productionSet
        firstPro=Set[0]
        #如果当前状态产生式的左端有多个产生式，则要具体区分两种情况：
        #1.没有完成项，只有未完成项，也就是点的位置没有到最后的项
        #2.有完成项和未完成项，贼烦，要考虑有没有移进和规约的冲突，还要考虑所有完成式之间有没有规约冲突
        #详见 ppt3.2   32 页
        if len(Set) !=1:
            if len(currentstatus.fin_pro) ==0:
                #没有完成项目：直接 shift 就好，不会有冲突：
                for j in currentstatus.line:
                    val=j.tranval
                    nextid=j.next.static_id
                    index=head.index(val)
                    table[i+1][index]="s"+str(nextid)
            else:
                #检查是不是有移进和规约的冲突：
                for ufin in currentstatus.unfin_pro:
                    if ufin.right[ufin.index] in tset:
                        for fin in currentstatus.fin_pro:
                            if ufin.right[ufin.index] in follow[fin.left]:
                                raise RuntimeError("存在移进和规约的冲突，程序正常停止")
                #检查是不是有规约的冲突：
                w=0
```

```
                n=0
                while w<len(currentstatus.fin_pro):
                    while n<len(currentstatus.fin_pro):
                        if w==n:
                            n+=1
                            continue
                        else:
                            for elem in follow[currentstatus.fin_pro[w].left]:
                                if elem in follow[currentstatus.fin_pro[n].left]:
                                    raise RuntimeError("存在规约的冲突，两个完成式左
端非终结符的 follow 集合有交集！ ")
                        n+=1
                    w+=1
                #程序运行到这里，说明没有移进和规约的冲突，也没有规约的冲突,可以开始填
表了：
                for fin in currentstatus.fin_pro:
                    for r in productionSet:
                        if productionSet[r][0]==fin.left and
productionSet[r][1]==fin.right:
                            if r == 0:
                                table[i + 1][len(tset)] = "ACC"
                            else:
                                col=1
                                while col<=len(tset):
                                    if head[col] in follow[fin.left]:
                                        table[i+1][col]="r"+str(r)
                                    col+=1
                            break
                #shift 操作：
                for j in currentstatus.line:
                    val = j.tranval
                    nextid = j.next.static_id
                    index = head.index(val)
                    table[i + 1][index] = "s" + str(nextid)
        #如果一个状态只包含一个产生式
        else:
            if firstPro.index!=firstPro.maxindex:
                #如果产生式的右端不是空的，那么就说明要进行移进的操作了。
                if firstPro.right[0]!="ε":
                    for j in currentstatus.line:
                        val=j.tranval
                        nextid=j.next.static_id
                        index=head.index(val)
                        table[i+1][index]="s"+str(nextid)
```

```
                #否则要进行的操作是进行规约操作：
                else:
                    for r in productionSet:
                        # 找到对应的状态的 id ， r 就是 id ， 这里要注意的是当 r=0 的时候是开始状态，应该把 ACC 填到这一行的
                        #  $ 符号对应的位置上。
                        # print(productionSet[r])
                        if productionSet[r][0] == firstPro.left and productionSet[r][1] == firstPro.right:
                            if r == 0:
                                table[i + 1][len(tset)] = "ACC"
                            else:
                                col = 1
                                while col <= len(tset):
                                    # 在当前产生式的左端的 follow 集合中填入这个规约条件：
                                    if head[col] in follow[firstPro.left]:
                                        table[i + 1][col] = "r" + str(r)
                                    col += 1
                            break
            # 这个产生式一定是一个完成产生式，只能进行规约操作
            # 根据 Follow 集合填表，不再和 LR0 算法一样，暴力全填上。
            else:
                for r in productionSet:
                    #找到对应的状态的 id ， r 就是 id ， 这里要注意的是当 r=0 的时候是开始状态，应该把 ACC 填到这一行的
                    #  $ 符号对应的位置上。
                    # print(productionSet[r])
                    if productionSet[r][0]==firstPro.left and productionSet[r][1]==firstPro.right:
                        if r==0:
                            table[i+1][len(tset)]="ACC"
                        else:
                            col=1
                            while col <=len(tset):
                                #在当前产生式的左端的 follow 集合中填入这个规约条件：
                                if head[col] in follow[firstPro.left]:
                                    table[i+1][col]="r"+str(r)
                                col+=1
                        break
        i+=1
    PrintTable(table)
    return table
```

可以非常明显的看到,SLR 分析表的构建比 LR0 分析表构建的算法代码要长很多，就是因为在里面约束了只有产生式左端 follow 集合中的元素才可以进行进行规约操作。

4. 分析过程和结果：

成功的分析过程：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/SLR.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
E->E + n|n
exit

拓广文法并给产生式编号:
 0 : E* -> E
 1 : E -> E + n
 2 : E -> n
分析表如下:
----------+----------+----------+----------+----------+
  status  |    +     |    n     |    $     |    E     |
----------+----------+----------+----------+----------+
    0     |          |    s4    |          |    s1    |
----------+----------+----------+----------+----------+
    1     |    s2    |          |   ACC    |          |
----------+----------+----------+----------+----------+
    2     |          |    s3    |          |          |
----------+----------+----------+----------+----------+
    3     |    r1    |          |    r1    |          |
----------+----------+----------+----------+----------+
    4     |    r2    |          |    r2    |          |
----------+----------+----------+----------+----------+
请输入要进行分析的字符串:
n + n
开始分析:
-----------------------------
分析栈: $ s0
输入队列: n + n $
动作: shift
-----------------------------------
分析栈: $ s0 n s4
输入队列: + n $
动作: reduce: E->n
```

```
-----------------------------------
分析栈：$ s0 E s1
输入队列：+ n $
动作：shift
-----------------------------------
分析栈：$ s0 E s1 + s2
输入队列：n $
动作：shift
-----------------------------------
分析栈：$ s0 E s1 + s2 n s3
输入队列：$
动作：reduce: E->E+n
-----------------------------------
分析栈：$ s0 E s1
输入队列：$
动作：reduce: E*->E
-----------------------------------
分析栈：$ s0
输入队列：$
动作：接受
-----------------------------------
请输入要进行分析的字符串：
n n
开始分析：
------------------------------
分析栈：$ s0
输入队列：n n $
动作：shift
-----------------------------------
分析栈：$ s0 n s4
输入队列：n $
动作：op 异常:分析表中这个位置是空的
不可以接受！
请输入要进行分析的字符串：
```

失败的分析过程：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/SLR.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
S->A a|b A c|B c|b B a
A->d
B->d
exit
----------------------------
```

```
拓广文法并给产生式编号:
 0 : S* -> S
 1 : S -> A a
 2 : S -> b A c
 3 : S -> B c
 4 : S -> b B a
 5 : A -> d
 6 : B -> d
RuntimeError: 存在规约的冲突，两个完成式左端非终结符的 follow 集合有交集!
```

第四部分：LR1 语法分析

一、LR1 分析方法概述：

SLR1 分析方法仅仅简单的考察下一个输入符号是不是属于与规约项目 A->a 相关联的 follow（A），但是 b 输入 follow（A）只是采用 A->a 进行规约的一个必要条件，非充分条件，实际上 ，真正需要规约的列只是 follow（A）的一个子集，所以 SLR0 算法只是排除了一些不合理的规约，但是不能保证一定会正确。因此引入了 LR1 分析方法，LR1 分析方法在分析过程中每一个产生式都记录了采用这个产生式进行规约需要后面所跟的后缀符号，称为展望符，规约的时候只有展望符所在的列才进行规约动作。

LR1 项目的定义：

将一个一般形式为[A->a.B,c]的项称为 LR1 项，其中 A->aB 是一个产生式，c 是一个终结符（视$为一个特殊的终结符）它表示在当前状态下，A 后面要求紧跟的终结符，称为该项目的展望符。

LR1 分析法的提出：

对于产生式 A->a 的规约，在不同的使用位置，A 要求不同的后继符号。例如：DFA 的一个状态如下：

S’ ->S

S ->L=R

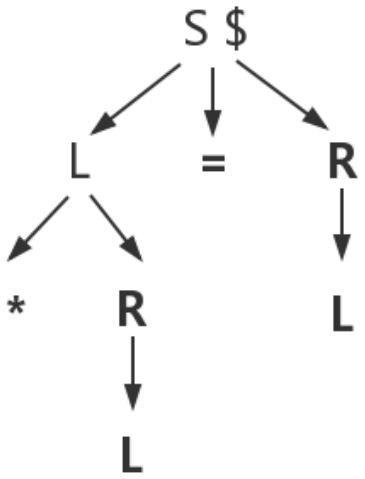
S ->R

L ->*R

L ->id

R ->L

在特定位置，A 的后继符集合是 Follow（A）的子集。



所以 LR1 分析方法在进行分析的时候在产生式的后端加入了展望符来表示不同的位置采取不同的规约。

二、具体过程和程序实现：

LR1 算法和 LR0，SLR 算法的唯一不同点是构造识别文法活前缀的 DFA 的时候要考虑展望符，所以在原有的数据结构中增加一个存储展望符号的字段，另外 LL1 分析表在构建的时候，判断两个状态是不是同一个状态的方法也不一样，需要考虑展望符，所以之前的函数也要进行修改。整体来说，LR1 分析方法也包括构架识别文法活前缀的 DFA，构建 LR1 分析表，根据分析表进行分析。这里主要说一下 DFA 的构建和 LR1 分析表的构建方法。

1.识别文法活前缀的 DFA：

构造识别文法活前缀的 DFA 和之前的方法没有什么不同，只是加入了一个展望符的运算，展望符实际上就是求的项目的点之后连同原展望符的字符串的 first 集合，该 first 集合中的全部符号都会和一个项目构成不同的项目。如果点后面的符号没有后继符号，则直接继承原来的展望符（在这里可以认为其后面有一个空

串，空串和原展望符构成的字符串的 first 集合就是原来的展望符，如此就可以和第一个方法统一了）。

2. 代码实现：

```
def getLR1DFA(gramma,first,nset):
    productionset=[]
    for i in gramma:
        for j in i.right:
            productionset.append([i.value,j])
    S=Status()
    S.add(Production(productionset[0][0],productionset[0][1],['$']))
    S.getClosure(productionset,first,nset)
    S.init_status()
    resultSet=[]
    resultSet.append(S)
    getNextStatus(resultSet,S,productionset,first,nset)
    print("识别文法活前缀的 DFA 状态集合：")
    print("————————————————————————————")
    for i in resultSet:
        print("状态 id: " + str(i.staticid))
        print("状态产生式集合以及点所处的位置")
        for j in i.productionSet:
            print((j.left, j.right, j.index,j.lookahead))
        print("状态的出边和所指向的状态标号")
        for k in i.line:
            print((k.tranval, k.next.staticid))
        print("----------------------------")
    return (resultSet,productionset)
```

识别文法活前缀的 DFA 的算法如上，可以看出较之前的代码有了很大的简化，通过前面的学习，对文法分析有了很深刻的认识，设计出了比较好的数据结构，运用面向对象的方法，将很多操作写成了类的成员函数，程序更加的易读，相关造作已经隐藏在了类中。构造算法主要运用了两个函数，一个是类的 getclosure（）成员函数，一个是 getnextstatus 函数，分别用来获得一个新状态的闭包和获取下一个状态。下面分别展示这两个函数的代码：

Status 类中的 getclosure 函数：

```
def getClosure(self,proSets,first,nset):
    for i in self.productionSet:
        if i.index==i.maxindex:
            continue
        if i.right[i.index] in nset:
```

```
            add=True
            #检查当前的集合中是否已经有了这个产生式：
            for j in self.productionSet:
                look=self.getLookAhead(i,first)
                if j.index==0 and j.left==i.right[i.index] and
set(look).issubset(set(j.lookahead)):
                    add=False
                    break
            if add:
                for item in proSets:
                    if item[0]==i.right[i.index]:
                        lookahead=self.getLookAhead(i,first)

self.productionSet.append(Production(item[0],item[1],lookahead))
```

getnextstatus 函数：

```
def getNextStatus(resultSet,status,proSet,first,nset):
    Set=status.productionSet
    #遍历所有产生式
    try:
        for i in range(len(Set)):
            #如果这是一个终止产生式，啥也不用做。
            if Set[i].index==Set[i].maxindex or Set[i].right==['ε']:
                continue
            #如果不是终止产生式，则要进行下一个状态的构造：
            tranval=Set[i].right[Set[i].index]
            newstatus=Status()
            newproduction=Production(Set[i].left,Set[i].right,Set[i].lookahead)
            newproduction.setindex(Set[i].index+1)
            newstatus.add(newproduction)
            #查找当前集合中是不是还有其他产生式经过一个相同的符号会转移
            for j in range(len(Set)):
                if i==j:
                    continue
                if Set[j].index==Set[j].maxindex:
                    continue
                if Set[j].right[Set[j].index]!=tranval:
                    continue

newproduction=Production(Set[j].left,Set[j].right,Set[j].lookahead)
                newproduction.setindex(Set[j].index+1)
                newstatus.add(newproduction)
            #状态得到闭包：
            newstatus.getClosure(proSet,first,nset)
            #状态增加
```

```
            add=True
            for s in range(len(resultSet)):
                if resultSet[s].same(newstatus):
                    add=False
                    nextstatus=resultSet[s]
                    line=Line(tranval,nextstatus)
                    status.addline(line)
                    break
            if add:
                newstatus.init_status()
                line=Line(tranval,newstatus)
                status.addline(line)
                resultSet.append(newstatus)
                getNextStatus(resultSet,newstatus,proSet,first,nset)
    except Exception as e:
        exit(str(e))
```

3. LR1 分析表的构建：

构建 LR1 分析表的方法和之前的方法略有不同，代码改动不大，主要做的工作是在进行规约填表的时候，之后产生式展望符的集合中的元素才可以被填上规约项目，LR0 简单的全部规约，SLR1 根据 follow 集合去掉了不合理的规约，而 LR1 运用展望符较准确的判断出产生式进行规约时候后边必须要跟随的符号的信息，使得分析更加的准确，适用的范围更加的广泛。

4. LR1 分析表构建代码：

```
def getLR1Table(resultSet,productionSet,tset,nset,start):
    table=[]
    head=[]
    head.append("STATUS")
    if 'ε' not in tset:
        tset.append('ε')
    for i in tset:
        if i!='ε':
            head.append(i)
    head.append("$")
    for i in nset:
        head.append(i)
    table.append(head)
    for i in resultSet:
```

```
        temp=[]
        temp.append(i.staticid)
        for j in head[1:]:
            temp.append("")
        table.append(temp)
    for i in range(len(productionSet)):
        print(i,':',productionSet[i][0],"->",productionSet[i][1])
    for i in range(len(resultSet)):
        cStatus=resultSet[i]
        for line in cStatus.line:
            if table[i+1][head.index(line.tranval)]=="":
                table[i+1][head.index(line.tranval)]="s"+str(line.next.staticid)
            else:
                print("存在冲突，请修改文法！")
                table[i + 1][head.index(line.tranval)] += "/s" +
str(line.next.staticid)
                PrintTable(table)
                exit(0)
        for pro in cStatus.productionSet:
            if pro.index==pro.maxindex or pro.right==['ε']:
                for look in pro.lookahead:
                    if table[i+1][head.index(look)]!="":
                        print("存在冲突，请修改文法：")
                        table[i + 1][head.index(look)] += "/r" +
str(getReduceIndex(productionSet, pro))
                        PrintTable(table)
                        exit(0)

table[i+1][head.index(look)]="r"+str(getReduceIndex(productionSet,pro))
    for i in range(len(resultSet)):
        for pro in resultSet[i].productionSet:
            if pro.left==start+"*" and pro.maxindex==pro.index:
                table[i+1][len(tset)]="ACC"
                break
    PrintTable(table)
    return table
```

5. 分析过程和结果：

成功的分析过程

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LR1.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
```

```
S->A a|b A c|B c|b B a
A->d
B->d
exit
产生式编号如下：
---------------------------
0 : S* -> ['S']
1 : S -> ['A', 'a']
2 : S -> ['b', 'A', 'c']
3 : S -> ['B', 'c']
4 : S -> ['b', 'B', 'a']
5 : A -> ['d']
6 : B -> ['d']
分析表如下：
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
  STATUS  |    a     |    b     |    c     |    d     |    $     |    S     |    A
|    B     |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
    0     |          |    s4    |          |   s12    |          |    s1    |    s2
|   s10    |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
    1     |          |          |          |          |   ACC    |          |
|          |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
    2     |    s3    |          |          |          |          |          |
|          |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
    3     |          |          |          |          |    r1    |          |
|          |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
    4     |          |          |          |    s9    |          |          |    s5
|    s7    |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
    5     |          |          |    s6    |          |          |          |
|          |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
```

```
   6    |         |         |         |         |   r2    |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
   7    |   s8    |         |         |         |         |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
   8    |         |         |         |         |   r4    |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
   9    |   r6    |         |   r5    |         |         |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
   10   |         |         |   s11   |         |         |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
   11   |         |         |         |         |   r3    |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
   12   |   r5    |         |   r6    |         |         |         |
|         |
----------+----------+----------+----------+----------+----------+----------+--
--------+----------+
请输入要进行分析的字符串：
b d c
开始分析：
------------------------------
分析栈：$ s0
输入队列：b d c $
动作：shift
------------------------------------
分析栈：$ s0 b s4
输入队列：d c $
动作：shift
------------------------------------
分析栈：$ s0 b s4 d s9
输入队列：c $
动作：reduce: A->d
------------------------------------
```

```
分析栈：$ s0 b s4 A s5
输入队列：c $
动作：shift
----------------------------------
分析栈：$ s0 b s4 A s5 c s6
输入队列：$
动作：reduce: S->bAc
----------------------------------
分析栈：$ s0 S s1
输入队列：$
动作：reduce: S*->S
----------------------------------
分析栈：$ s0
输入队列：$
动作：接受
请输入要进行分析的字符串：
b d c c
开始分析：
------------------------------
分析栈：$ s0
输入队列：b d c c $
动作：shift
----------------------------------
分析栈：$ s0 b s4
输入队列：d c c $
动作：shift
----------------------------------
分析栈：$ s0 b s4 d s9
输入队列：c c $
动作：reduce: A->d
----------------------------------
分析栈：$ s0 b s4 A s5
输入队列：c c $
动作：shift
----------------------------------
分析栈：$ s0 b s4 A s5 c s6
输入队列：c $
动作：op 异常:分析表中这个位置是空的
不可以接受！
请输入要进行分析的字符串：
```

失败的分析过程

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LR1.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
```

```
S->S + a T|a T|+ a T
T->+ a T|+ a
exit

产生式编号如下：
--------------------------
0 : S* -> ['S']
1 : S -> ['S', '+', 'a', 'T']
2 : S -> ['a', 'T']
3 : S -> ['+', 'a', 'T']
4 : T -> ['+', 'a', 'T']
5 : T -> ['+', 'a']
存在冲突，请修改文法：
分析表如下：
----------+----------+----------+----------+----------+----------+
  STATUS  |    +     |    a     |    $     |    S     |    T     |
----------+----------+----------+----------+----------+----------+
    0     |   s10    |    s8    |          |    s1    |          |
----------+----------+----------+----------+----------+----------+
    1     |    s2    |          |    r0    |          |          |
----------+----------+----------+----------+----------+----------+
    2     |          |    s3    |          |          |          |
----------+----------+----------+----------+----------+----------+
    3     |    s5    |          |          |          |    s4    |
----------+----------+----------+----------+----------+----------+
    4     |    r1    |          |    r1    |          |          |
----------+----------+----------+----------+----------+----------+
    5     |          |    s6    |          |          |          |
----------+----------+----------+----------+----------+----------+
    6     |  s5/r5   |          |    r5    |          |    s7    |
----------+----------+----------+----------+----------+----------+
    7     |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+
    8     |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+
    9     |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+
    10    |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+
    11    |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+
    12    |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+
```

```
Process finished with exit code 0
```

由上图可知，程序可以很清楚的判断出冲突所在的位置正常退出。

第五部分：LALR1 语法分析

一、LALR1 分析方法概述：

由于 LR1 分析方法会生成很多的状态，因此编译的过程会有些不方便，为了减少这些状态，引入 LALR 分析方法，合并同心项目，达到简化状态数目的目的，采用的方法是在 LR1 的识别文法活前缀的 DFA 构建完成之后进行合并操作，和并项目的展望符。之后的操作和前面的分析方法一致。

LALR 分析方法基本思想：

1）寻找相同核心的 LR1 项目集，并将这些项集合并为一个项集，所谓项集的核心就是其第一分量的集合
2）根据合并后的项集构造语法分析表
3）如果语法分析表中没有语法分析动作冲突，给定的文法就成为 LALR 文法，可以根据分析表进行语法分析。

二、具体过程和程序实现：

由于 LALR 分析方法仅仅是合并 LR1 分析方法产生的状态，所以这里仅仅给出 LALR 合并同心项的代码：

1.合并同心项代码

```
def getLALRDFA(resultSet):
    result=[]
    delete =[]
    print("找到的可以合并的同心项如下：")
    print("++++++++++++++++++++++++++++++++++++++++++++")
    for i in range(len(resultSet)):
        if resultSet[i].staticid in delete:
            continue
        for j in range(len(resultSet)):
            if i==j:
```

```
                continue
            if resultSet[j].staticid in delete:
                continue
            if resultSet[i].sameHeart(resultSet[j]):
                print((i,j))
                delete.append(resultSet[j].staticid)
                proSet=resultSet[j].productionSet
                getUnionLook(resultSet[i],proSet)
                fixline(resultSet,i,j)
    print("识别文法活前缀的 LALR DFA 状态集合： ")
    print("————————————————————————————")
    for i in resultSet:
        if i.staticid in delete:
            continue
        result.append(i)
        print("状态 id: " + str(i.staticid))
        print("状态产生式集合以及点所处的位置")
        for j in i.productionSet:
            print((j.left, j.right, j.index, j.lookahead))
        print("状态的出边和所指向的状态标号")
        for k in i.line:
            print((k.tranval, k.next.staticid))
        print("----------------------------")
    return result
```

2. 分析过程和结果：

成功的分析：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe
D:/Data/Coder/python/GrammaAnalysis/LALR1.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格:
S->A a A b|B b B a
A->ε
B->ε
exit

产生式的编号如下：
---------------------------
0 : S* -> ['S']
1 : S -> ['A', 'a', 'A', 'b']
2 : S -> ['B', 'b', 'B', 'a']
3 : A -> ['ε']
4 : B -> ['ε']
```

```
分析表如下：
----------+----------+----------+----------+----------+----------+----------+
  STATUS  |    a     |    b     |    $     |    S     |    A     |    B     |
----------+----------+----------+----------+----------+----------+----------+
    0     |    r3    |    r4    |          |    s1    |    s2    |    s6    |
----------+----------+----------+----------+----------+----------+----------+
    1     |          |          |   ACC    |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
    2     |    s3    |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
    3     |          |    r3    |          |          |    s4    |          |
----------+----------+----------+----------+----------+----------+----------+
    4     |          |    s5    |          |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
    5     |          |          |    r1    |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
    6     |          |    s7    |          |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
    7     |    r4    |          |          |          |          |    s8    |
----------+----------+----------+----------+----------+----------+----------+
    8     |    s9    |          |          |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
    9     |          |          |    r2    |          |          |          |
----------+----------+----------+----------+----------+----------+----------+
请输入要进行分析的字符串：
a b
开始分析：
------------------------------
分析栈：$ s0
输入队列：a b $
动作：reduce: A->ε
------------------------------------
分析栈：$ s0 A s2
输入队列：a b $
动作：shift
------------------------------------
分析栈：$ s0 A s2 a s3
输入队列：b $
动作：reduce: A->ε
------------------------------------
分析栈：$ s0 A s2 a s3 A s4
输入队列：b $
动作：shift
------------------------------------
```

```
分析栈：$ s0 A s2 a s3 A s4 b s5
输入队列：$
动作：reduce: S->AaAb
----------------------------------
分析栈：$ s0 S s1
输入队列：$
动作：reduce: S*->S
----------------------------------
分析栈：$ s0
输入队列：$
动作：接受
----------------------------------
请输入要进行分析的字符串：
b b
开始分析：
-----------------------------
分析栈：$ s0
输入队列：b b $
动作：reduce: B->ε
-----------------------------------
分析栈：$ s0 B s6
输入队列：b b $
动作：shift
-----------------------------------
分析栈：$ s0 B s6 b s7
输入队列：b $
动作：op 异常:分析表中这个位置是空的
不可以接受！
请输入要进行分析的字符串：
```

失败的分析：

```
D:\Data\Coder\python\GrammaAnalysis\venv\Scripts\python.exe D:/Data/Coder/python/GrammaAnalysis/LALR1.py
请输入文法，例如：A->a|A b,输入 exit 结束输入(注意,不同的符号之间要有空格：
S->A a|b A c|B c|b B a
A->d
B->d
exit
--------------------------
0 : S* -> ['S']
1 : S -> ['A', 'a']
2 : S -> ['b', 'A', 'c']
3 : S -> ['B', 'c']
4 : S -> ['b', 'B', 'a']
5 : A -> ['d']
```

```
6 : B -> ['d']
存在冲突，请修改文法：
分析表如下：
```

STATUS	a	b	c	d	$	S	A	B
0		s4		s9		s1	s2	s10
1					r0			
2	s3							
3					r1			
4				s9			s5	s7
5			s6					
6					r2			
7	s8							
8					r4			
9	r6/r5		r5					
10								
11								

```
Process finished with exit code 0
```

LALR 的冲突一定是规约和规约的冲突，由于合并了两个项目的展望符，不能确定合并之后不会有冲突，如果无冲突，则说明文法是一个 LALR 文法，使用 LALR 分析方法主要是为了简化 LR1 分析方法中繁多的状态。更有实用性。

总结和展望：

通过对词法分析和语法分析的程序编写，我最大的收获是非常清晰的学会了这几个分析方法，几乎可以不用再看就可以参加考试，果然需要动手去做才会学得更加的透彻。这份文档与其说是一份报告，倒不如说是我复习的总结，写这份

报告的时候我最初只是想做一下简答的介绍，后来我发现我可以写的详细一点，清楚 一点，之后这篇文档说不定可以帮助其他同学更好的理解编译原理，帮助他们写出程序，所以这篇文档我写的很详细，在写的过程中，我进一步的展开了复习，对此法分析和语法分析部分有了更加深刻的认识，之前上课不太懂的东西现在完全懂了（不懂的话就写不出来程序，即使写出来了，结果也不一定完全正确），希望考试顺利！

很高兴的一件事是词法分析部分的程序界面做的比较友好，并且具有交互能力，得到了老师的认可和表扬，有时间也会继续完善语法分析部分的代码，帮助其他同学更好的理解词法分析和语法分析。目前全部代码已经放到了 github 上面，可以在上面下载完整的代码。