编译方法说明

#### 3. 设计思路

本次实验设计流程为：

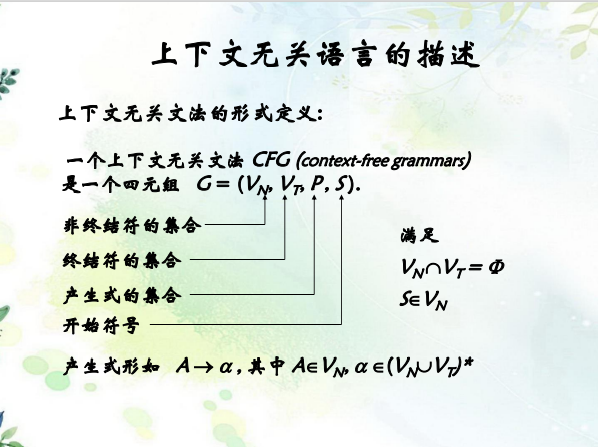
用户输入文法—》程序保存文法—》用户选择处理文法：

* 化简文法
* 消除左公共因子
* 消除左递归
* 求非终结符号的first集合与follow集合
* 转换得到的有穷自动机—》DFA—》最小化DFA

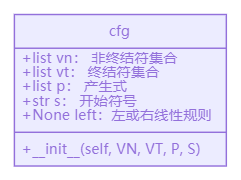
逐步设计算法处理，最后将结果展现到windows界面上

##### 3.1 储存结构

cfg文法的定义如下：



要设计文法，首先需要考虑文法的储存结构，我设计了一个cfg文法的类结构：



其中提前设置好left变量以便后续判断是左或者右线性规则，亦或者不是线性规则

**3.2 用户输入和保存文法**

用户输入界面如下：



用户可通过输入文法，或打开保存文法的文件填写输入框

点击”初始化“按钮后准备对文法做后续处理。

文法文件规则如下：

第一行：vn

第二行：vt

第三行：开始符号s

第四行及以下：产生式p

**应严格按照以上规则保存文法**

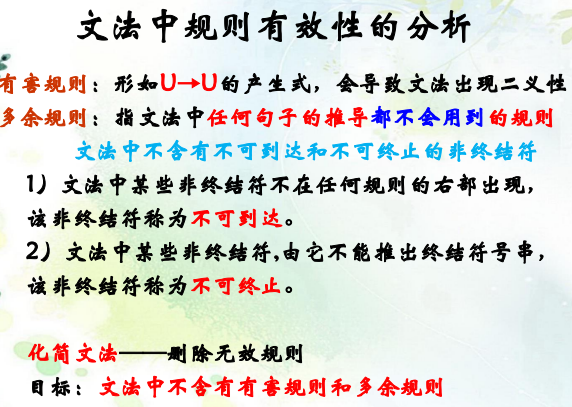
**3.3 化简文法**

这部分的算法我写入了文件：**simplify\_cfg.py**

其中编写的函数如下：

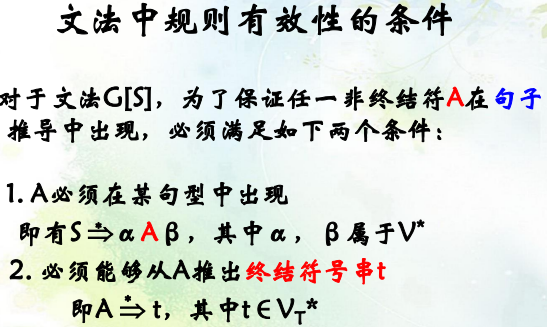
def if\_senten\_tofinal(cfg, senten, used\_ch): # 判断一个规则能否推出终结符
  
def is\_tofinal(cfg, ch, used\_ch): # 判断一个非终结字符的能否推出终结符
  
def delete\_ch(cfg, P, ch): # 删除与终结字符有关的规则
  
def simplify\_cfg(cfg): # 化简文法，主函数

参考上课的pdf：



第一步是去除文法的有害规则，首先遍历所有的产生式，去除形如U->U的有害规则。

第二步是根据以下条件遍历消除不可终结的无用符号和无用产生式：



删除某个非终结符后需要同时将与其有关的规则删除！

##### 注意

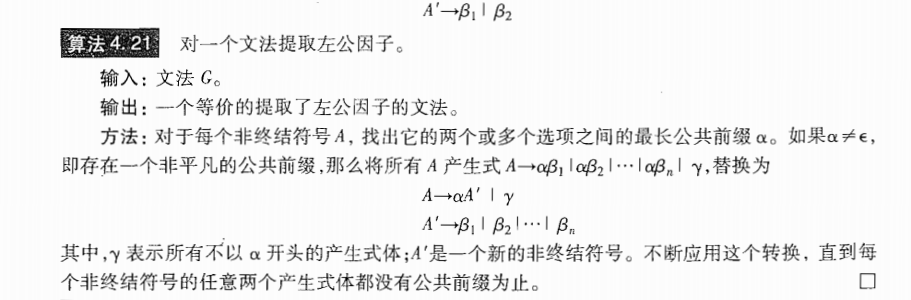
• 先寻找并消除全部不可终止的符号  
 • 再寻找并消除全部不可到达的符号  
 • 否则可能消除不完整

**3.4 消除左公共因子**

这部分的算法我写入了文件：**del*left*common\_factor.py**

其中编写的函数如下：

def mfun2(strs): # 寻找列表的最长公共子串
  
def left\_common\_factor(cfg):

主要算法简介如下：

对每条规则进行判断，若规则以非终结符起始要进行推导，将其用终结符起始的规则替换，从某个非终结符号开始做4次推导内发现有左公共因子即可消除，如果超过4次推导还存在左公共因子就直接产生出错处理；直到遍历完所有规则后结束

**对于产生新的终结符，用终结符+数字表示，如A->A0**

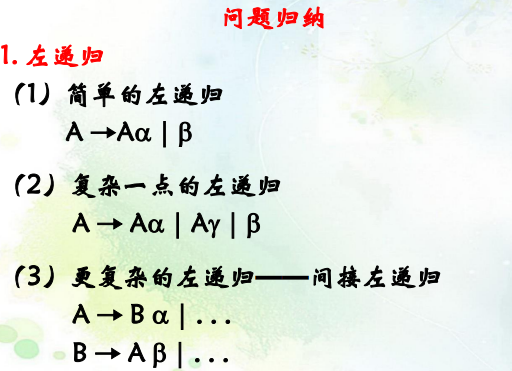
**3.5 消除左递归**

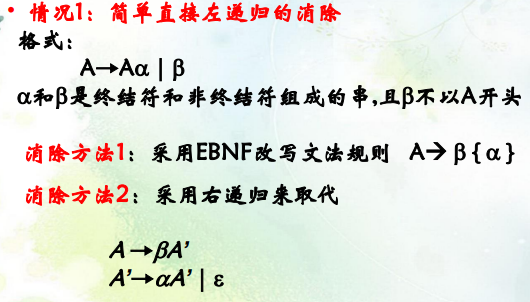
这部分的算法我写入了文件：**del*left*recursion.py**

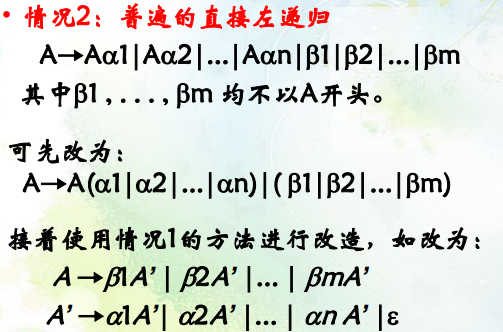
其中编写的函数如下：

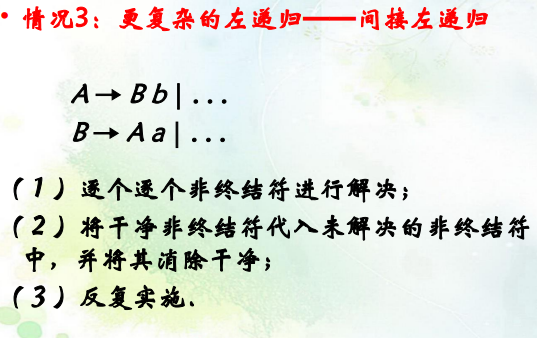
def left\_recursion(cfg): # 算法函数
  
def if\_recursion(cfg, left\_ch, production, used\_ch, finished): # 递归判断函数

递归分为以下情况：

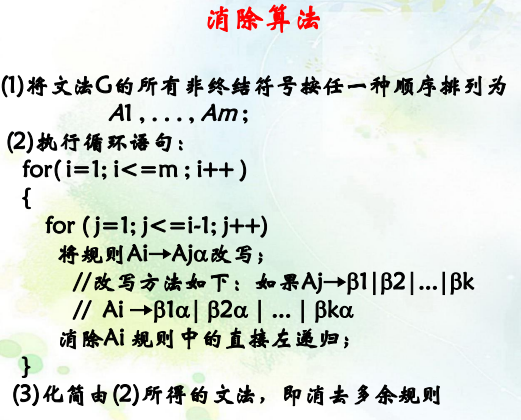








参考以下伪代码:



遍历每条规则，先对规则判断：if if*recursion(cfg, production[0], production[1], used*ch, finished):, 若没有递归不处理该规则，解决完一个非终结符后将其加入finished集合，若后续有递归的规则首字符出现finished集合中的规则，需要带入消除。

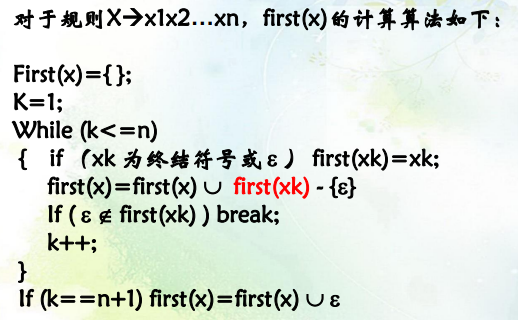
**3.6 求first集合与follow集合**

这部分的算法我写入了文件：**get*first*and\_follow.py**

其中编写的函数如下：

def find\_first(cfg, left\_ch, ch\_first, used\_ch): # 递归
  
def get\_first(cfg, first):
  
def find\_follow(cfg, ch, first, ch\_follow, used\_ch): # 递归
  
def get\_follow(cfg, first, follow):

**求first集合的伪代码：**

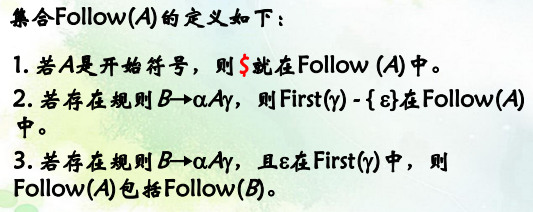


**求first集合的结构图：**



因此目标很明确，遍历产生式，对于规则首部是终结符直接加入first集合，对于非终结符则将该终结符的first集合加入，因此这部分要用到递归循环判断，注意要及时跳出循环。

**求follow集合有三种情况：**



伪代码如下：



同样使用了递归，不过比求first集合复杂一些

**3.7 生成有穷自动机**

这部分的算法我写入了文件：**finite*machine*create.py**

其中编写的函数如下：

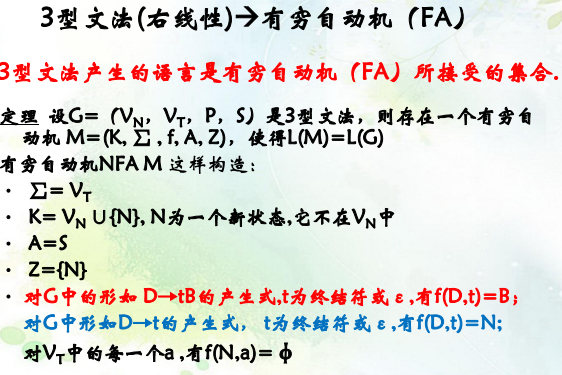
def is\_regluar\_grammer(cfg): # 判断线性规则
  
def finite\_machine(cfg, fa\_value): # 生成有穷自动机图和nfa表

线性规则的定义如下：



因此首先要判断该文法好是否线性规则，并且要判断是左或右线性规则

左右线性规则的处理方法相反：





右线性规则要添加多一个终止结点，左线性规则要添加起始结点

同时顺便构造fa表，以便后续转为DFA并呈现，并再进行DFA的最小化并呈现

**3.7 设计界面**

界面左边用户输入或保存文法规则，右边做文法规则处理：

