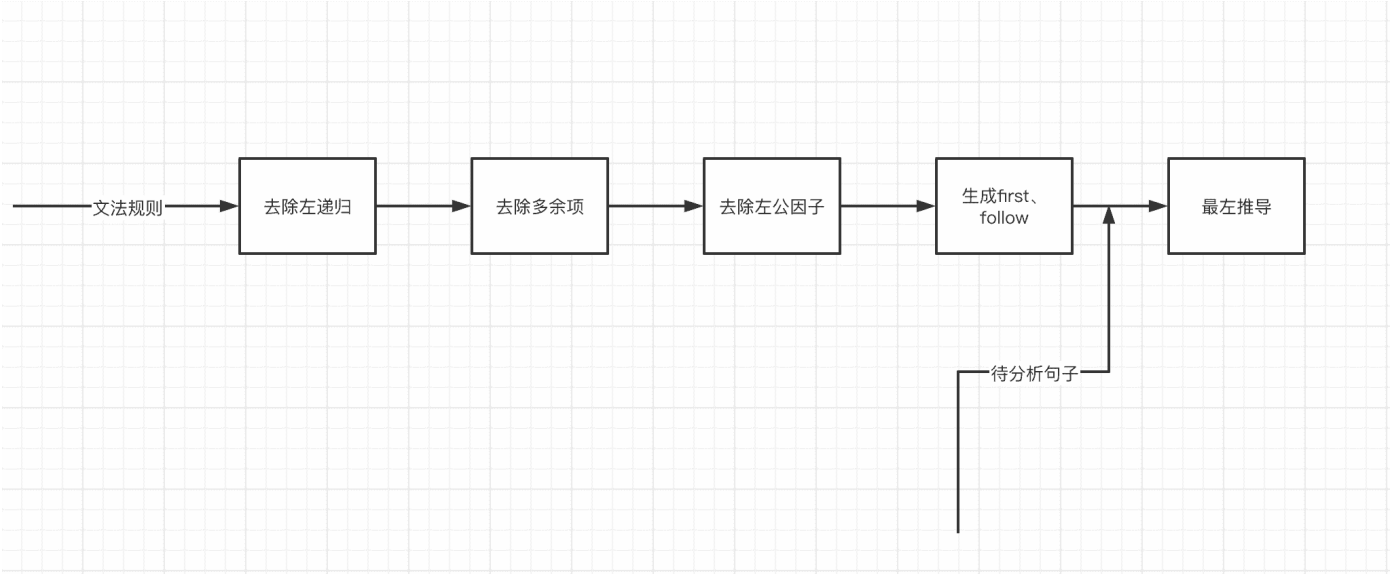
文法问题处理器

20192131031 梁诺明

一、实验内容：
设计一个应用软件，以实现文法的化简及各种问题的处理。

二、实验要求：
（1）系统需要提供一个文法编辑界面，让用户输入文法规则（可保存、打开存有文法规则的文件）
（2）化简文法：检查文法是否存在有害规则和多余规则并将其去除。系统应该提供窗口以便用户可以查看文法化简后的结果。
（3）检查该文法是否存在着左公共因子（可能包含直接和间接的情况）。如果存在，则消除该文法的左公共因子。系统应该提供窗口以便用户可以查看消除左公共因子的结果。
（4）检查该文法是否存在着左递归（可能包含直接和间接的情况），如果存在，则消除该文法的左递归。系统应该提供窗口以便用户可以查看消除左递归后的结果。
（5）求出经过前面步骤处理好的文法各非终结符号的first集合与follow集合，并提供窗口以便用户可以查看这些集合结果。【可以采用表格的形式呈现】
（6）对输入的句子进行最左推导分析，系统应该提供界面让用户可以输入要分析的句子以及方便用户查看最左推导的每一步推导结果。【可以采用表格的形式呈现推导的每一步结果】
（7）应该书写完善的软件文档

流程图



存储结构设计

文法规则存储结构

使用string来存储左边部分，用vector<vector<string>>来存储右边部分。

对于文法规则 A -> acd | efg，则右边部分的存储结构为[[a, c, d], [e, f, g]]

类声明如下：

```

#define EPSILON "@"
class LinkNode
{
public:
    //左部
    string left;
    //右部
    vector<vector<string>> right;

    LinkNode(string str);
    //右边添加规则
    void insert(vector<string> &nodes);

    bool includesEpsilon();

    string toString();
};
```

文法处理器存储结构

文法处理器的功能包括对文法的存储以及各种处理操作。

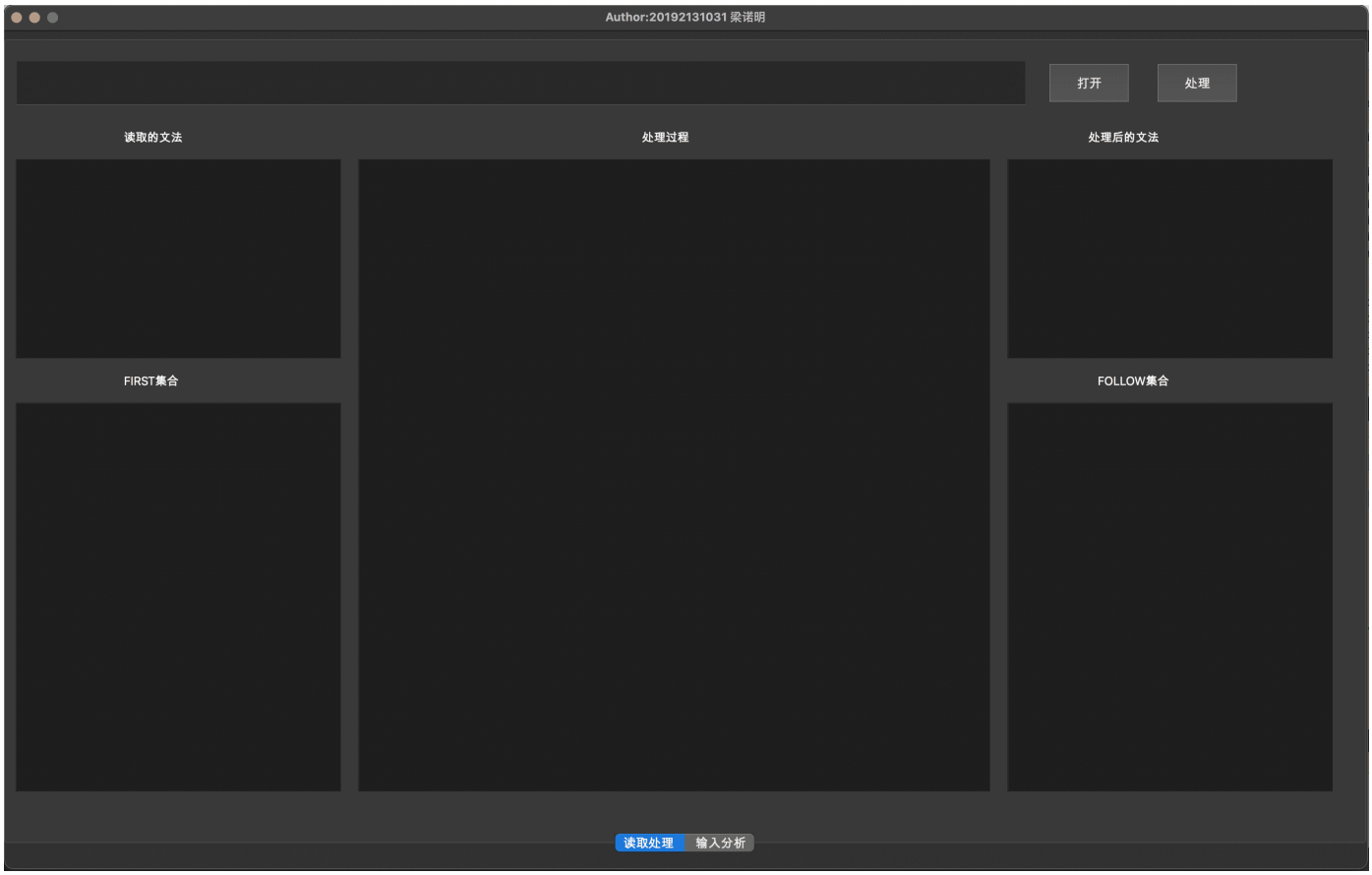
1. **使用vector<LinkNode>存储多条文法规则**
2. **使用vector<string>存储用户输入的待推导句子、终结符号、非终结符号**
3. **使用map<string, set<string>>存储各个非终结符号对应的first集合和follow集合**

类声明如下：

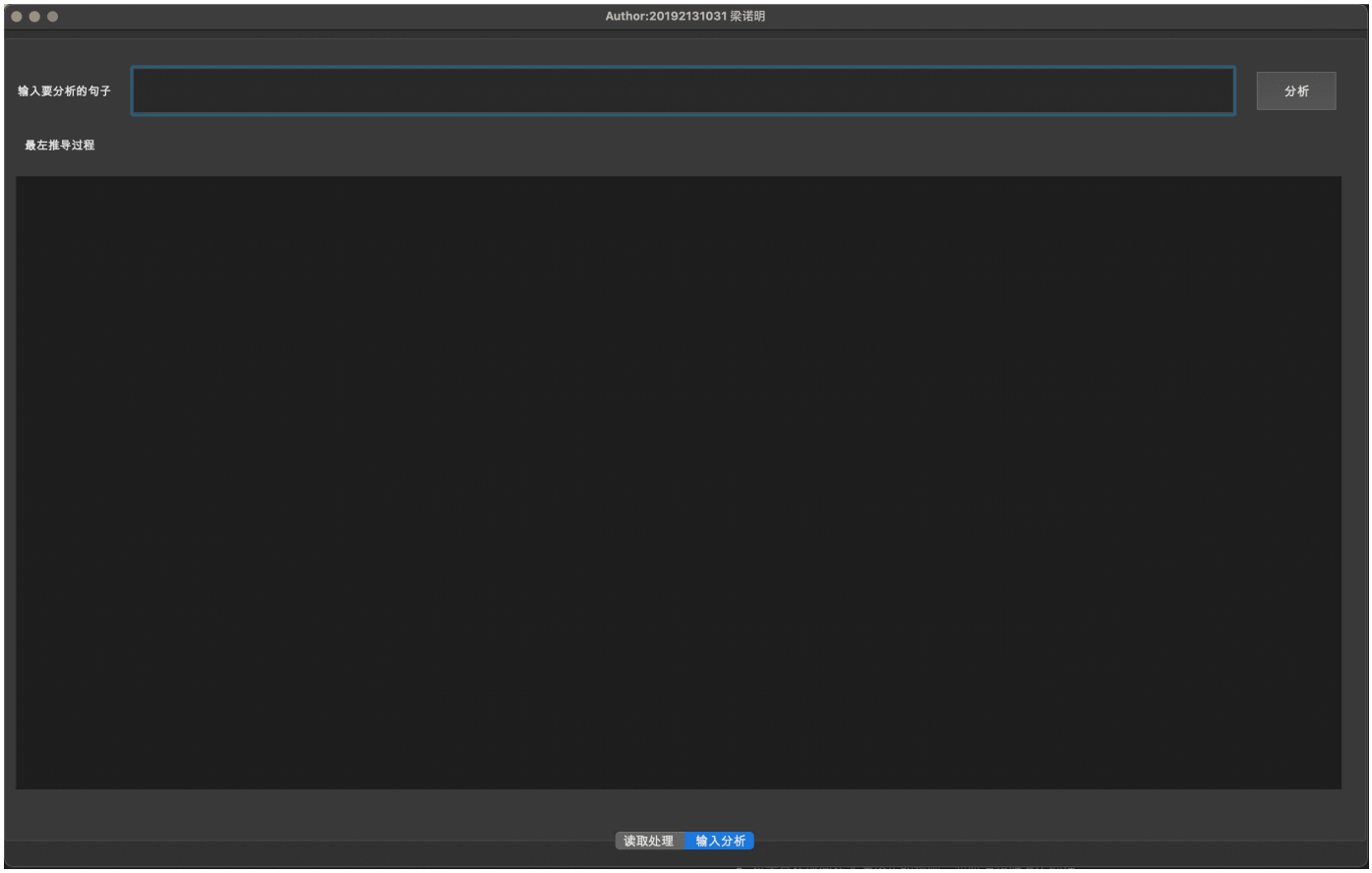
```
class Processor
{
public:
    vector<string>input;//存用户输入的待分析句子
    int startWordIndex;//开始节点索引
    string startNode;//开始结点值
    vector<LinkNode> grammers;//存文法
    vector<string> finalWord;//终结符号
    vector<string> nonFinalWord;//非终结符号
    map<string, set<string>> first;//First集合
    map<string, set<string>> follow;//Follow集合
    ofstream log;//文件流，用于写出日志
    Processor(string filePath);//构造函数，参数是文件路径
    void init();//初始化终结符号和非终结符号
    void dealLeftRecursion();//处理左递归
    void dealLeftCommonFacotr();//处理左公因子
    void simplify();//化简
    void getFirst();//生成First集合
    void getFollow();//生成Follow集合
    bool isFinalWord(string word);//判断是不是终结符号
    bool isWord(string word);//判断是不是该文法能处理的符号
    //按照指定符号拆分字符串
    vector<string> splitString(string str, string spliter);
    void printGrammers();//输出当前文法信息到log文件
    vector<vector<string>> leftMostDerivation();//最左推导
private:
    //判断是否访问过
    bool visited[MAX_NODE_NUMBER];
    //找到非终结符号word对应的是第几条规则
    int getIndex(string word);//配合getFirst()函数使用，递归实现求某条规则的first集合
    void findFirst(int i);
    void dfs(int x);//深度优先遍历
    //处理单条文法的左公因子
    vector<LinkNode> dealCommonFactor(LinkNode&node);
    //生成一个没有使用过的非终结符号名
    string generateNewName(string source);
    //将str1的follow添加到str2的follow集合
    void append(const string &str1, const string &str2);
};

```

界面设计

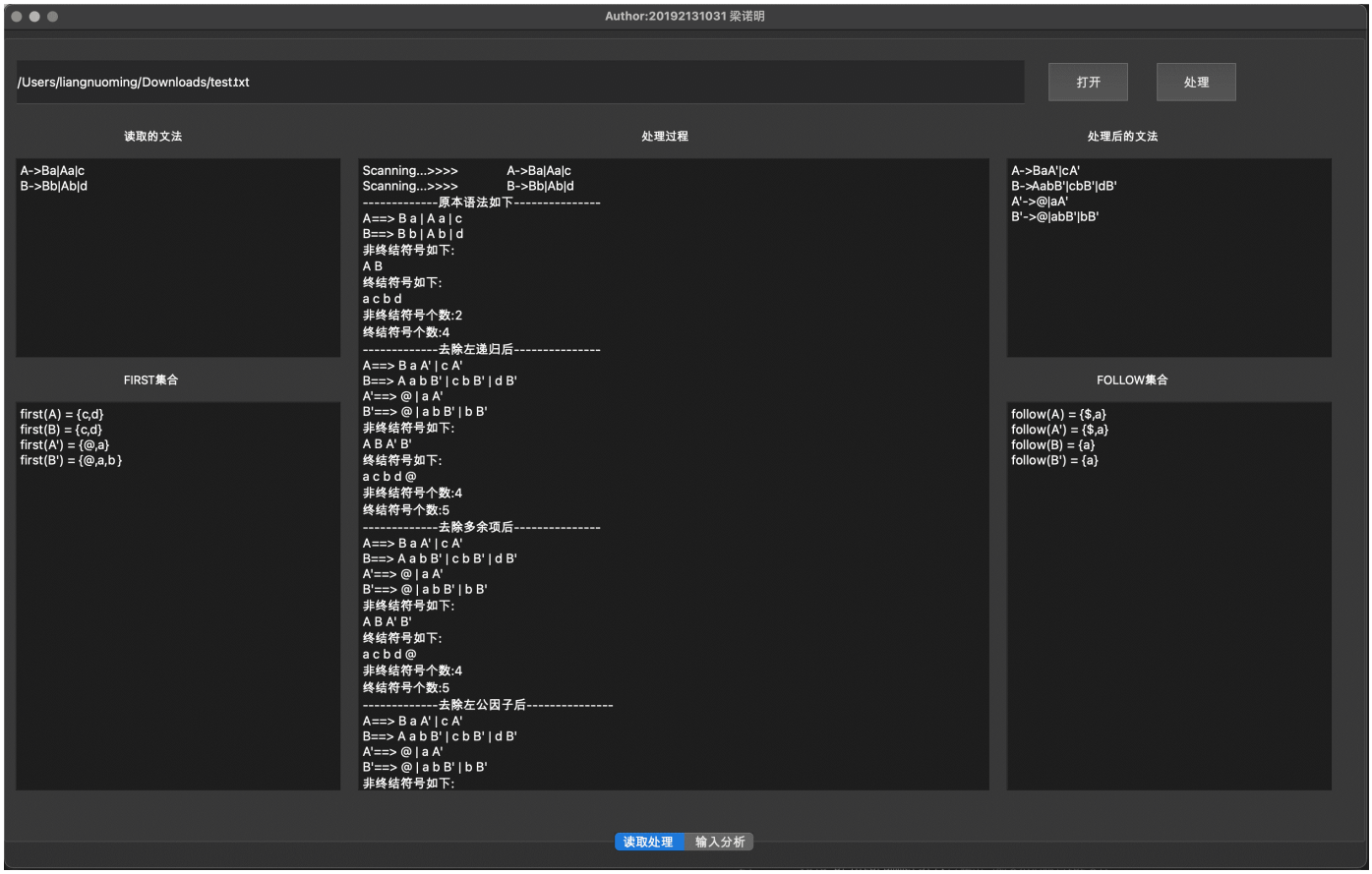
处理文法的界面



最左推导界面


Author:20192131031 梁诺明
输入要分析的句子
分析
最左推导过程
读取处理
输入分析

结果展示


Author:20192131031 梁诺明
/Users/liangnuoming/Downloads/test.txt
打开
处理
读取的文法
A->Ba|Aa|c
B->Bb|Ab|d
处理过程
Scanning...>>>> A->Ba|Aa|c
Scanning...>>>> B->Bb|Ab|d
----------原本语法如下----------
A==> B a | A a | c
B==> B b | A b | d
非终结符号如下:
A B
终结符号如下:
a c b d
非终结符号个数:2
终结符号个数:4
----------去除左递归后----------
A==> B a A' | c A'
B==> A a b B' | c b B' | d B'
A'==> @ | a A'
B'==> @ | a b B' | b B'
非终结符号如下:
A B A' B'
终结符号如下:
a c b d @
非终结符号个数:4
终结符号个数:5
----------去除多余项后----------
A==> B a A' | c A'
B==> A a b B' | c b B' | d B'
A'==> @ | a A'
B'==> @ | a b B' | b B'
非终结符号如下:
A B A' B'
终结符号如下:
a c b d @
非终结符号个数:4
终结符号个数:5
----------去除左公因子后----------
A==> B a A' | c A'
B==> A a b B' | c b B' | d B'
A'==> @ | a A'
B'==> @ | a b B' | b B'
非终结符号如下:
处理后的文法
A->BaA'|cA'
B->AabB'|cbB'|dB'
A'->@|aA'
B'->@|abB'|bB'
FIRST集合
first(A) = {c,d}
first(B) = {c,d}
first(A') = {@,a}
first(B') = {@,a,b}
FOLLOW集合
follow(A) = {$,a}
follow(A') = {$,a}
follow(B) = {a}
follow(B') = {a}
读取处理
输入分析

主要算法说明

求first集合

对于规则X -> x1x2...xn，first(x)的计算方法如下：

```
void Processor::getFirst()
{
    memset(visited, false, sizeof(visited));
    //对所有规则左边的非终结符号求first集合
    for(size_t i = 0; i < grammers.size(); i++)
        findFirst(i);
    //记录日志信息
    map<string, set<string>>::iterator it = first.begin();
    for(; it!=first.end(); it++)
    {
        log << "FIRST(" << it->first << ")={";
        set<string> &temp = it->second;
        set<string>::iterator it1 = temp.begin();
        bool flag = false;
        while(it1!= temp.end())
        {
            if(flag)    log << ",";
            log << *it1;
            flag = true;
            it1++;
        }
        log << "}" << '\n';
    }
}

```

求follow集合

计算follow集合的算法如下：

```
void Processor::getFollow()
{
    follow[grammers[startWordIndex].left].insert("$");
    //用于标记follow集合的值是否发生变化
    bool changed;
    while(true)
```

```
    {
        changed = false;
        //遍历每条文法
        for(size_t i = 0; i < grammers.size(); i++)
        {
            //左边部分
            string &left = grammers[i].left;
            //右边部分
            vector<vector<string>> &right = grammers[i].right;
            vector<vector<string>>::iterator it = right.begin();
            //遍历A -> bcd|efg|hij的后边三个部分，分别是bcd，efg，hij
            while(it != right.end())
            {
                //first(Xi+1,...,Xn)是否包含EPSILON
                bool flag = false;
                vector<string> &words = *it;
                //遍历每一个Xi
                for(size_t j = 0; j < it->size(); j++)
                {
                    flag = false;
                    string word = words[j];
                    // 如果当前符号是非终结符号
                    if(isFinalWord(words[j]) == false)
                    {
                        if(j == it -> size() - 1)
                        {
                            //如果是最后一个符号
                            //A->aB，就把Follow(A)加入Follow(B)
                            int sizeBefore = follow[word].size();
                            append(left, it->at(j));
                            int sizeAfter = follow[word].size();
                            if(sizeAfter > sizeBefore)
                                changed = true;
                        }
                        //将First(Xi+1,...,Xn)-EPSILON加入到follow(Xi)
                        for(size_t k = j + 1; k < it->size(); k++)
                        {
                            //first(Xi)是否包含EPSILON
                            bool flagi = false;
                            string nextWord = words[k];
                            if(isFinalWord(nextWord) == false)
                            {
                                set<string> &from = first[nextWord];
                                set<string> &to = follow[word];
                                set<string>::iterator it1 = from.begin();
                                int sizeBefore = follow[word].size();
                                for(; it1 != from.end(); it1++)
```

```
54                                  {
55                                      if(*it1 != EPSILON)
56                                          to.insert(*it1);
57                                      else
58                                      {
59                                          //包含了EPSILON
60                                          flagi = true;
61                                      }
62                                  }
63                                  int sizeAfter = follow[word].size();
64                                  if(sizeAfter > sizeBefore)
65                                      changed = true;
66                                  //如果first(Xi+1)中包含EPSILON，则继续把first(Xi+2)加入到
    follow(Xi)
67                                  //如果不包含EPSILON，则直接break;
68                                  if(flagi == false)
69                                      break;
70                                  if(flagi == true && k == it -> size() - 1)
71                                      flag = true;
72                              }
73                              else
74                              {
75                                  int sizeBefore = follow[word].size();
76                                  follow[word].insert(nextWord);
77                                  int sizeAfter = follow[word].size();
78                                  if(sizeAfter > sizeBefore)
79                                      changed = true;
80                                  break;
81                              }
82                          }
83                          // 如果First(Xi+1,...Xn)包含EPSILON
84                          if(flag)
85                          {
86                              size_t sizeBefore = follow[word].size();
87                              //把A的follow集合加入到B中
88                              append(left, it->at(j));
89                              size_t sizeAfter = follow[word].size();
90                              if(sizeAfter > sizeBefore)
91                                  //还在变化，要继续
92                                  changed = true;
93                          }
94                      }
95                  }
96                  it++;
97              }
98          }
99          if(changed == false)
```

```
            break;
        }
        //输出follow集合到log文件
        map<string, set<string>>::iterator it = follow.begin();
        for(; it != follow.end(); it++)
        {
            log << "FOLLOW(" << it->first << ")={";
            set<string> &temp = it->second;
            bool flag = false;
            for(set<string>::iterator it1 = temp.begin(); it1 != temp.end(); it1++)
            {
                if(flag) log << ",";
                log << *it1;
                flag = true;
            }
            log << "}" << endl;
        }
    }
```

最左推导

设计一个计数器，表示当前匹配到第几个字符。从开始符号开始执行，每次执行找到当前已生成符号串从左到右第一个非终结符号，查看该终结符号的文法规则：

- 若第一个字符是终结符号，且和待匹配字符相同，就选择该转换
- 若第一个字符是非终结符号，且first集合包含待匹配字符，就选择该转换
- 若第一个字符是非终结符号，且follow集合包含待匹配字符，就选择ε转换

去除无效文法

- 对于形如A -> A的有害规则，在读取规则时就可以将其去除
- 对于直接或间接无法终止的规则，暂时没想好怎么处理……
- 对于用不到的无效规则，可以采用深度优先的方法，设置一个bool数组用于标记是否访问过各条文法规则，从开始结点遍历文法规则，当遍历结束时，对于没有访问过的文法规则，就将其删除

去除左递归

- 先将文法右边的非终结符号用其对应的转换规则取代，可以使得间接的左递归也转化成直接左递归
- 然后再开始消除直接左递归，例如A -> Aa|b，就转换成A -> bA‘，A' -> aA’ | ε

去除左公因子

处理的思路是每次只寻找一个左公因子

- 例如A -> ab | abc
- 第一次处理后变成A -> aA'，A' -> b | bc
- 第二次处理后变成A -> aA'，A‘ -> bA''，A’’ -> ε | c

定义从调函数，其功能是对于某条文法规则，若有左公因子，就返回处理好后的（一条变多条）文法规则，若没有左公因子，就返回一个空的vector；定义一个主调函数，定义变量i，当i小于文法数量时，就一直执行从调函数，若从调函数返回的向量为空，就说明此文法规则无左公因子，那么i++，否则就将目前处理的文法规则替换为从调函数返回的（多条）文法规则