

[一、实验内容 3](#_Toc119947062)

[1.概述 3](#_Toc119947063)

[2.必做内容 3](#_Toc119947064)

[3.选做内容 4](#_Toc119947065)

[4.备注 4](#_Toc119947066)

[二、实验目的 4](#_Toc119947067)

[三、实验过程 4](#_Toc119947068)

[（一）理论准备与算法 4](#_Toc119947069)

[0.数据结构 4](#_Toc119947070)

[（1）定义 4](#_Toc119947071)

[（2）存储结构 4](#_Toc119947072)

[1.化简文法 5](#_Toc119947073)

[（1）有害规则 5](#_Toc119947074)

[（2）多余规则 5](#_Toc119947075)

[2.求出first集合 5](#_Toc119947076)

[（1）定义 5](#_Toc119947077)

[（2）手工算法 6](#_Toc119947078)

[（3）代码算法 6](#_Toc119947079)

[3.求出follow集合 6](#_Toc119947080)

[（1）定义 6](#_Toc119947081)

[（2）手工算法 7](#_Toc119947082)

[（3）代码算法 7](#_Toc119947083)

[4.消除左公因子 7](#_Toc119947084)

[（1）定义 7](#_Toc119947085)

[（2）手工算法 7](#_Toc119947086)

[（3）代码算法 7](#_Toc119947087)

[5.消除左递归 8](#_Toc119947088)

[（1）定义 8](#_Toc119947089)

[（2）手工算法 8](#_Toc119947090)

[（3）代码算法 8](#_Toc119947091)

[6.生成自动机 9](#_Toc119947092)

[（1）将文法规则转换为正则表达式 9](#_Toc119947093)

[（2）生成NFA图 9](#_Toc119947094)

[（3）生成DFA图 10](#_Toc119947095)

[（4）生成最小DFA图 10](#_Toc119947096)

[（二）代码细节 11](#_Toc119947097)

[1.界面与文件存取 11](#_Toc119947098)

[（1）界面设计 11](#_Toc119947099)

[（2）文件读取 11](#_Toc119947100)

[2.文法处理器 11](#_Toc119947101)

[（1）数据结构 12](#_Toc119947102)

[（2）基础工具代码 12](#_Toc119947103)

[（3）细节-遍历的过程中改动链表 13](#_Toc119947104)

[（4）技巧-遍历文法 13](#_Toc119947105)

[3.自动机 14](#_Toc119947106)

[（三）测试数据 14](#_Toc119947107)

[（五）遇到的问题 15](#_Toc119947108)

[四、实验总结 16](#_Toc119947109)

[（一）学习感悟 16](#_Toc119947110)

[（二）编码心得 16](#_Toc119947111)

[五、参考文献 16](#_Toc119947112)

# 一、实验内容

## 1.概述

设计一个应用软件，以实现文法的化简及各种问题的处理。

## 2.必做内容

（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。【从某个非终结符号开始只需要消除做4次推导内就可以发现的左公共因子，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即可】

（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）如果输入的文法规则是线性规则（左线性规则或右线性规则），则将该文法对应的有穷自动机生成出来，系统应该提供界面让用户可以查看转换得到的有穷自动机（可以使用状态表的方法来呈现）。【注意：如果该有穷自动机是NFA，还需要把该NFA转为DFA并呈现，并再进行DFA的最小化并呈现，你可以使用实验二的算法来完成】

（7）软件的操作界面应该是windows界面

（8）应该书写完善的软件文档

## 3.选做内容

（1）对于必做内容中的第（6）个要求，使用画图的算法设计把DFA图，NFA图等直观图示呈现在窗口中。

## 4.备注

（1）注意事项：文法规则为了处理上的简单，输入时可以只输入单一个大写字母作为非终结符号，单一个小写字母作为终结符号，用@表示空串。

如，可以这样输入：

E->E+T

T->a

（2）补充说明：4次推导才发现左公共因子的举例：

S->Aa | Ba

A->Cb

B->Dc

C->Ed

D->Fe

E->fe

F->fd

如果超过4次才发现的左公共因子的就产生出错，如果少于等于4次推导的就要把左公共因子进行消除。

# 二、实验目的

（1）理解文法问题的解决算法

（2）加深对状态机的NFA图、DFA图的理解

（3）增强编码能力

# 三、实验过程

## （一）理论准备与算法

### 0.数据结构

#### （1）定义

文法：描述语言语法结构的一系列形式规则。

上下文无关文法CFG是一个四元组G=（VN非终结符的集合，VT终结符的集合，P产生式的集合，S开始符号）。

整个实验都是在分析和重构文法。那么就要尤其注意**存储**文法和**分析**文法的数据结构。

#### （2）存储结构

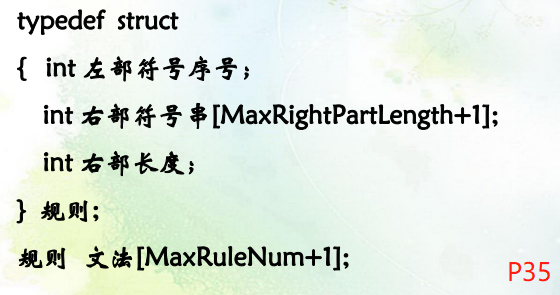
文法规则有数组和链表两种结构，我选择链表。

首先，用一个**字典**存储读入的字符并为其安排对应的序号。终结符号的序号大于1小于100，非终结符号的序号大于100。这样易于划分终结符和非终结符，程序也有可拓展性。

|  |
| --- |
| map<int, char> v; |

接着，定义一个**结构体**，名为“规则”，参考：

|  |
| --- |
| struct Rule{  int leftChar;  vector<int> rightChar;  };  List<Rule> grammar; |



// vector适合随机存取，list适合插入和删除

为了便于将来的操作，需要为非终结符排序。

|  |
| --- |
| list<Rule> vn; |

另外，可以为它定义一个**打印函数**PrintGrammar()，打印A->a格式的字符串。

### 1.化简文法

化简文法就是去除其中的无效规则，其中无效规则包括了有害规则和多余规则。

#### （1）有害规则

定义：导致文法出现二义性的规则。例如A->A，B->B。

算法：直接找左部符号和右部符号一样的文法规则结构体，删了。【比较】

#### （2）多余规则

定义：文法中任何句子的推导都不会用到的规则，又分为不可到达规则和不可终止规则。

算法：

①去除不可到达规则：

将S放入“可到达集合”中。从S开始遍历每个文法，若左部在集合中，则将右部的每个非终结符也放入“可到达集合”中。再次遍历每个文法，若左部不在集合中，则删去。【并查集】

②去除不可终止规则：

【深度优先搜索】【递归】

①从开始符号为左部起步，遍历其对应的每个文法。设当前左部符号为A，检查是否满足A->αAβ形式。

②若满足，则标记该文法，查找以A为左部、可终止的文法。若不存在可终止，则删除所有以A为左部，或右部含有A的文法，返回。

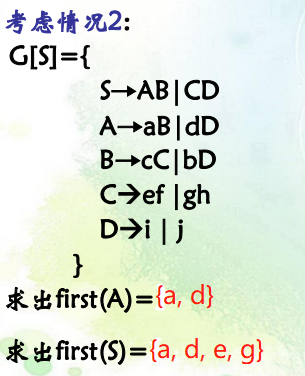
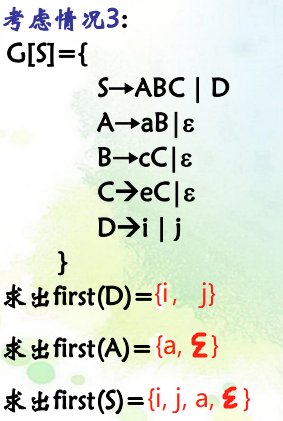
③若不满足，且右部存在非终结符B，则以B为左部重复步骤①②。

举例：S->Be，B->Ce，B->Af，A->e ，A->Ae，A->E，E->e，C->Cf，D->f。

### 2.求出first集合

#### （1）定义

可以从**非**终结符号或符号串X**推导出**的所有串首**终结符**构成的**集合**。如果X->∗ε，那么 ε也在FIRST（X）中。举例：

#### （2）手工算法

以X为目标，从右部符号串取第一个字符a，若a是终结符或ε，则填入Follow集中；若a是非终结符，则继续重复上述步骤，以a为左部，向右推导，直到其右部符号串的**串首**是终结符或ε。

#### （3）代码算法

定义【递归】函数GetFirst(int X)。遍历以X为左部的文法，取文法右部第一个字符假设为a。若a是终结符或ε，则填入当前Follow集中；若为非终结符且不为X（**避免进入死循环**），则递归进入GetFirst(int a)。遍历结束则返回。

### 3.求出follow集合

#### （1）定义

给定一个在右部的非终结符A，则Follow(A)为**紧跟**在其后的每个**终结符号**或$（右端结束标记）的**集合**。举例：

|  |  |  |
| --- | --- | --- |
| E -> TE’  E’-> +TE’| ε  T -> FT’  T’-> \*FT’| ε  F -> (E) | a  Follow(E)={ $, ) }  Follow(E’)={ $, ) }  Follow(T)=  { +, $, ) }  Follow(T’)=  { +, $, ) }  Follow(F)=  { \*, +, $, ) }  (来源：[编译原理中Follow集的求法\_杨博东的博客的博客-CSDN博客\_follow集](https://blog.csdn.net/yangbodong22011/article/details/52950436?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-52950436-blog-106379883.pc_relevant_recovery_v2&spm=1001.2101.3001.4242.1&utm_relevant_index=3)) | S→ABc  A→a|ε  B→b|ε  Follow（S）=  ｛＃｝  Follow（A）=  ｛b，c｝  Follow（B）=  ｛c｝  （来源：[编译原理 First集 Follow集 select集 通俗易懂的讲解 + 实例\_CooperNiu的博客-CSDN博客\_select集](https://blog.csdn.net/CooperNiu/article/details/78524688?spm=1001.2101.3001.6650.6&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-6-78524688-blog-115911002.pc_relevant_aa2&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7EBlogCommendFromBaidu%7ERate-6-78524688-blog-115911002.pc_relevant_aa2&utm_relevant_index=7)） | S→AB  S→bC  A→ε  A→b  B→ε  B→aD  C→AD  C→b  D→aS  D→c  FOLLOW(S)={ $ }  FOLLOW(A)= { a, $, c }  FOLLOW(B)= { $ }  FOLLOW(C)={ $ }  FOLLOW(D)={ $ }  （来源：[判断LL(1)文法（first集、follow集、select集）\_内存不足°的博客-CSDN博客\_判断ll(1)文法](https://blog.csdn.net/qq_44922497/article/details/111880076?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-4-111880076-blog-78524688.pc_relevant_recovery_v2&spm=1001.2101.3001.4242.3&utm_relevant_index=7)） |

#### （2）手工算法

不断应用下面的规则，直到再没有新的终结符号可以被加入到任意的follow集合中为止：

①将 **$** 放到follow（S）中，其中S是文法的开始符号。

②当A是最右部的时候，将 **$** 加入到follow(A)中

③如果存在一个产生式A→α**B**β，那么follow（B）**包含first（β）-ε**。（ follow(B)是求跟在B后的终结符或$组成的集合，因此对于跟在B后的β，它的first集合就是follow(B)的子集 ）

④如果存在一个产生式A→α**B**，或存在产生式A→αBβ且first（β）包含ε，那么follow（B）**包含follow（A）**。（对于A→αBβ,且β多步推导出ε ，那么可以用αB替换A, B后面紧跟的字符就是A后面紧跟的字符）

#### （3）代码算法

定义【递归】函数GetFollow(int X)。

若X是文法开始符号，则将$填入当前Follow集中。遍历每个文法。

若X在文法最右部，则将$填入当前Follow集中。

若存在A->αXβ形式，则将GetFirst(β)-ε填入当前Follow集中。

若GetFirst(β)包含ε，则将GetFollow(A) 填入当前Follow集中。

若存在A->αX形式，则将GetFollow(A) 填入当前Follow集中。

### 4.消除左公因子

#### （1）定义

一个或多个文法规则共享一个通用前缀串。

举例：A->**ab**，A->**ac**。B->**ac**mm，B->**ac**d。

#### （2）手工算法

提取左公因子，将后缀改为“左部符号**’**”，并新增文法：左部符号’->后缀

举例（承接上面的例子）：A->a**A’**，A’ ->b | c。B->ac**B’**，B’ ->mm | d。

#### （3）代码算法

定义【迭代】函数Remove\_ Left\_ Common\_Factor ()。

①遍历每个非终结符号，设当前符号为Vni

②遍历以Vni为左部的文法，装入容器list<Rule>中并统计个数n。

③若个数n≠1，则定义字典map<vector, int>，和数组union[ Index ]（初始化为0）。遍历以Vni为左部的文法，若**右部第一个字符**为**非终结字符Vnj**，则在字典添加First(Vnj)；若为**终结字符a**，则在字典添加{a}，并分别赋序号Index1, Index2。若字典已有某个集合，且其对应的序号为Index3，则union[Index 3]= Index 1，union[Index 1] = -1。【并查集】

④遍历union，若发现union[Index 1]=-1，找到第Index 1个文法，其原形式为A->aB，新增非终结符**C**，将A->aB改为A->aC，新增C->B。找到所有union[…]=n的文法A->aB’，改为C->B’。

### 5.消除左递归

#### （1）定义

左递归：文法经过一次或多次推导之后，出现如下形式A->Aα，则称该文法是左递归的。左递归会在自顶向下分析的过程中产生不好的回溯。

①直接左递归：经过一次推导就可以看出文法存在左递归。如A->Aα|β。递归结果为A->Aα->Aαα->Aααα（总在左边增加α）->βααα= β{α}

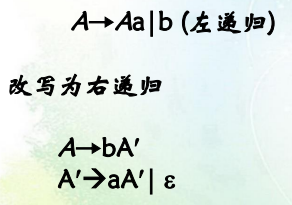
②间接左递归：需多次推导才可以看出文法存在左递归。如文法：S->Qc | c，Q->Rb | b，R->Sa | a，有S->Qc->Rbc->Sabc。

#### （2）手工算法

①消除直接左递归：

举一个最简单的例子，把A->Aa | β直接改为A->βA’，A’ ->aA’ | ε（右递归）。

更一般化的形如P → P**X** | **Y**（其中X和Y看作一个整体，比如：P → Pabc|ab|b，X就是abc，Y就是ab|b），可改写为P->**Y**P'，P'->**X**P' | **ε**。

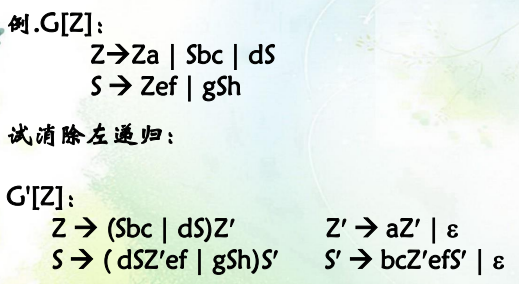


②消除间接左递归：

1）若消除过程中出现了直接左递归，则按照左递归的方法直接消除

2）将消除直接左递归后的新文法代入未解决的文法中（即间接左递归），得到新的直接左递归，按照步骤1再次消除

3）反复实施，直到不可代入



#### （3）代码算法

定义【迭代】函数Remove\_ Left\_Recursion()。

需要给符号排序。

①遍历每个非终结符号，设当前符号为Vni

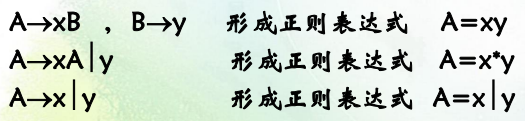
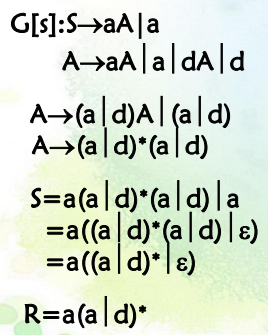
②遍历该符号之前的非终结符号Vnj

③遍历以Vnj为左部的文法，**代入**到形式为Vni→αVnj β的文法中，得到新的Vni的文法

④判断Vni是否存在形式为A→ A**X**的文法。若存在，则新增非终结符B，将文法A->AX修改为文法B->XB，新增文法B->ε。遍历其它以Vni为左部的文法，将B追加到最右部。（**消除直接左递归**）

### 6.生成自动机

#### （1）将文法规则转换为正则表达式

①遍历每个非终结符A->……，形成正则表达式A=……

②处理S，将各表达式代入到S的表达式中

#### （2）生成NFA图

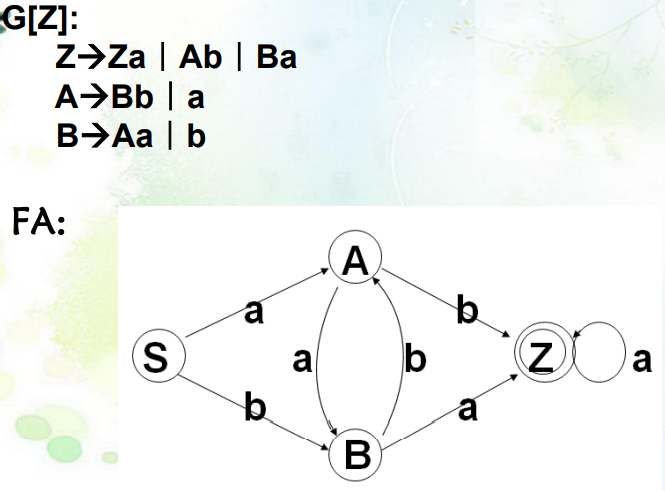
①左线性

形如A->a或A->Ba的文法，即右部第一个字符为终结符或非终结符，之后的每个字符都是终结符，对应生成S--a-->A，或B--a-->A。

算法：

遍历每条文法，检测是否满足“右部最左的符号是非终结符号”并且“右部其它符号都是终结符号”。若满足，则判定为左线性文法。

新增非终结符S作为开始结点。生成含有非终结符个数的结点的图。遍历每个文法，A->Bc生成从以B为起点，转移条件为c，指向A结点的路径（填表）。



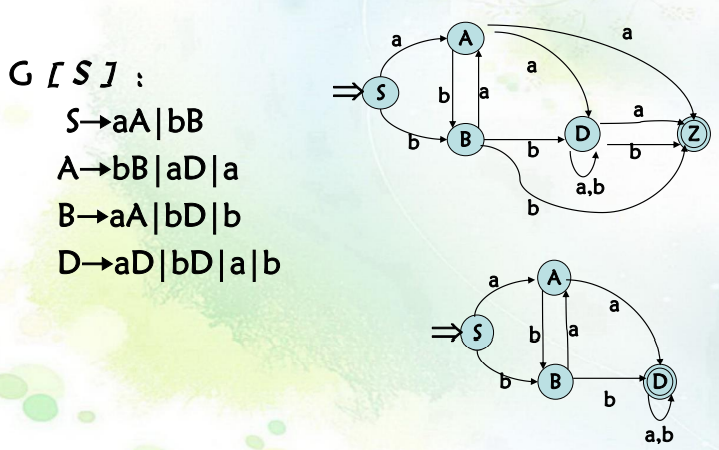
②右线性

形如A->a或A->aB的文法，即右部第一个字符为终结符或非终结符，之后的每个字符都是终结符，对应生成A--a-->Z，或A--a-->B。

算法：

遍历每条文法，检测是否满足“右部最右的符号是非终结符号”并且“右部其它符号都是终结符号”。若满足，则判定为右线性文法。

新增非终结符Z作为终态结点。生成含有非终结符个数的结点的图。遍历每个文法，A->cB生成从以A为起点，转移条件为c，指向B结点的路径（填表）。



#### （3）生成DFA图

NFA转成DFA的目的就是消除ε转换。

算法：

①求出第一列的ε闭包，放入闭包队列中。

②取出队头的闭包。若该闭包没有在DFA的第一列没有出现，则填入DFA的第一列。

③根据第一列最新的闭包，把对应的NFA格子中的字符填到DFA中，注意去重。填完之后求ε闭包。若闭包不为空，则放入闭包队列中。

④重复②③，直到队列为空。

#### （4）生成最小DFA图

DFA转最小DFA的目的就是把条件转换相同的结点合在一起。对于表格，就是合并行相同的两行（手工算法）。

代码算法：因为我存储图的表格是个二维字符数组vector<vector<char>>，第[i][j]格就是vector[ i \* col + j ]。要想比较两行是不是相同，我直接把每行的字符从左到右抄进string里，遇到空单元格就抄空格进去。然后比较哪两个字符串是相同的。如果相同，就把对应的DFA行合并填写到minDFA中。

## （二）代码细节

### 1.界面与文件存取

#### （1）界面设计



#### （2）文件读取

定义一个Loader类，**专门**作为读取文件的工具，因此将其设为**单例**。C++的饿汉式单例模式实现起来非常简单粗暴，用静态变量和静态函数即可。但存在线程安全问题，但不在本次作业的考虑范围内。

|  |
| --- |
| class **Loader**  {  private:  static char\* filePath;  public:  static char\* **Input**(const char\* filePath);  static void **Output**(const char\* compressed);  static char\* **GetFilepath**();  }; |

### 2.文法处理器

定义一个类专门负责处理文法规则，名为GrammarProcessor。大部分算法的思路在理论准备部分**代码算法**中已写明，并在源代码中也备注了清晰的注释。在此我只分析有技巧的代码段。

#### （1）数据结构

参考理论准备部分，定义存储结构如下：

|  |
| --- |
| map<int, char> v; // **字典映射表**  list<Rule> grammar; // **文法规则**  int n\_num = 0, t\_num = 100; // 非终结符起始符号，终结符起始符号  char n\_char = 0; // 记录最新的非终结符，以便分配新的非终结符字符（n\_char + 1）  int temp\_set[200]; // 算法中多次用到并查集的思想，所以预先分配一个数组  list<int> first; // first集采用深度优先算法生成，故在函数外定义存储结构  vector<vector<char>> NFA; // **NFA表格** |

其中，文法结构体定义如下：

|  |
| --- |
| struct **Rule**{  int left;  vector<int> right;  bool operator == (const Rule& a) const{  if(left == a.left && right == a.right)  return true;  return false;  }  bool operator < (const Rule& a) const{  if(left < a.left)  return true;  return false;  }  }; |

#### （2）基础工具代码

①内部函数

根据理论准备部分的分析，可知以下函数需要经常被用到：

|  |
| --- |
| /\*工具函数\*/  void **AddVn**(char c); // 添加非终结符  void **AddVt**(char c); // 添加终结符  int **value2key**(char c); // 根据符号查找映射表中的序号  bool **isVn**(int num); // 判断是否为非终结符  bool **isVt**(int num); // 判断是否为终结符  // 查询容器中是否含有某值（不知为何Qt中用不了std::find函数，因此我重写）  bool **find**(vector<int>::iterator a, vector<int>::iterator b, int value);  void **OrganizeDict**(); // 整理映射表，去除已经被删去的字符索引  void **OrganizeGrammar**(); // 排序文法规则 |

②字符串之间的类型转换

QString转char

|  |
| --- |
| QByteArray fn\_qba = fileName.toLatin1();  char\* fn\_cc = fn\_qba.data();  char\* c = Loader::Input(fn\_cc); |

char转QString

|  |
| --- |
| QString(c) |

String转QString

|  |
| --- |
| QString::fromStdString(gm->PrintGrammar()) |

（3）核心算法函数分配

|  |
| --- |
| private:  void **RemoveHarmfulRules**();  void **RemoveUnreachableRules**();  bool **RemoveUnterminableRules\_sub**(int vn, int depth);  void **RemoveUnterminableRules**();  void **GetFirst\_sub**(int x);  list<int>& **GetFirst**(int x);  void **GetFollow\_sub**(int x);  list<int> **GetFollow**(int x);  bool **LL2NFA**();  bool **RL2NFA**();  public:  **GrammarProcessor**(char \*c);  string **SimplifyGrammar**();  string **GetFirst**();  string **GetFollow**();  string **RemoveLeftCommonFactor**();  string **RemoveLeftRecursion**();  string **GetRegularExpression**();  vector<vector<char>> **GetNFA**();  string **PrintGrammar**(); |

#### （3）细节-遍历的过程中改动链表

定义函数RemoveHarmfulRules()，按照理论准备部分编写代码。其中，关键代码如下：

|  |
| --- |
| if((\*it).right.size() == 1 && (\*it).left == (\*it).right[0]) // U->U  {  it = grammar.erase(it);  it--;  } |

当链表移除某一元素时，当前的it指针会发生变动，导致指向错误。使用erase()函数，会返回删除元素的下一元素的迭代器。我以此给it重新赋值，并将it指向上一元素，这样就能正常遍历了。

#### （4）技巧-遍历文法

为了避免过多的遍历，我先确保文法规则已排序，相同的左部**聚集**在一处。遍历文法规则的过程中，若找寻到目标（以A为左部的文法），则将tag标记为true。当遍历到左部不是A的文法时，且tag为true，说明已经遍历完A->…，之后都不会再有A->…了，则结束遍历。

|  |
| --- |
| bool tag = false;  for(auto& g : grammar)  {  if(g.left == vn) // 遍历以A为左部的每个文法  {  tag = true;  ……  }  else if(tag == true) //（剪枝操作）  {  break;  }  } |

（5）获取NFA

因为每个非终结符的编号为0~99，每个终结符的编号为101~199（第100号是$），因此可以直接把他们抄到表格里。

### 3.自动机

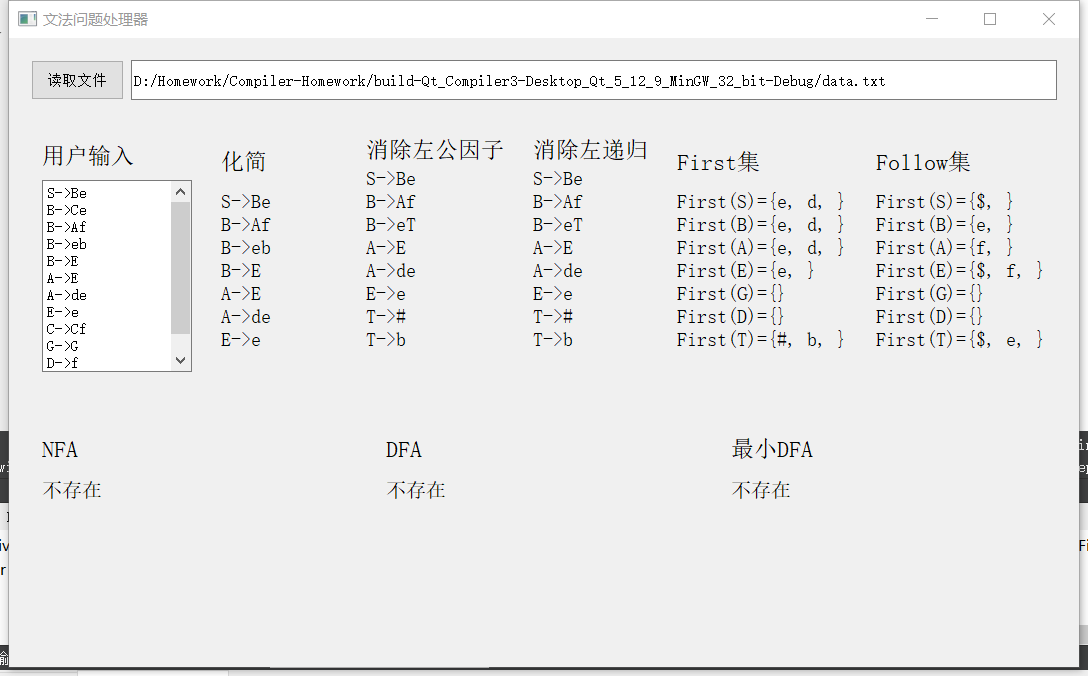
定义一个负责转化表达式、NFA、DFA的自动机类，重写实验2的RegularExpression2NFA类的代码。

|  |
| --- |
| class **RegularExpressionProcessor**  {  vector<vector<char>> NFA;  vector<vector<char>> DFA;  vector<vector<char>> minDFA;  string **PrintFATable**(vector<vector<char>>::iterator it);  bool **find**(vector<char>::iterator begin, vector<char>::iterator end, char value);  public:  void **SetNFA**(vector<vector<char>> nfa);  string **GetNFA**();  string **GetDFA**();  string **GetMinDFA**();  }; |

## （三）测试数据

一个数据文件涉及的处理过程足够繁杂，且本人能力有限，再次仅设计3组测试数据。

（1）data1.txt

 测试：消除有害规则（G->G）、不可达规则（D->f）、不可终止规则（B->Ce、C->Cf），消除左公因子（B->eb、B->E、E->e）。

|  |
| --- |
| S->Be  B->Ce  B->Af  B->eb  B->E  A->E  A->de  E->e  C->Cf  G->G  D->f |

（2）data2.txt

测试：消除左公因子（A->dB、A->dC），求出NFA、DFA、最小DFA。

|  |
| --- |
| A->dB  A->dC  A->a  B->c  C->c |



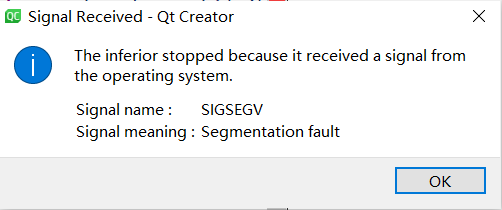
（3）data3.txt

测试：消除左递归。

|  |
| --- |
| A->Aa  A->Bbc  A->c  B->Aef  B->gAh  B->a  C->c |

## （五）遇到的问题

（1）



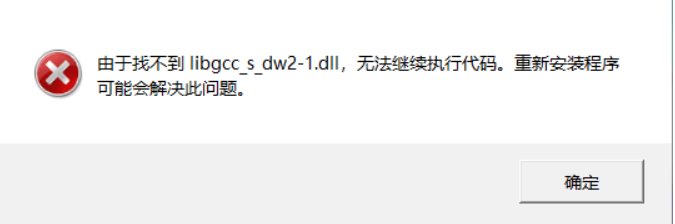
老师说，没遇到过段错误，都不能说自己敲的代码多。

情况一：Debug之后发现是指针问题，把**新开辟**的数组赋值给了错误的变量a，然后再把这个变量a赋值给别的变量b，就出问题了。

情况二：在疯狂Debug又终止又Debug的过程中，内存会抽搐，随便运行一行赋值代码都会栈溢出。这种情况下重启电脑就好了。

最后，我终于明白，虽然每次出错都会跳出这样的框并进入一堆完全看不懂的底层代码段里，但是Qt下方的debug框都很多行，双击点开可以一步一步找到出现错误的上层代码段，就像python。

（2）将Qt程序打包之后运行出现报错



我用Qt 5.12.9 (MinGW 7.3.0 **64**-bit)打包就会出现这样的问题，换成用32-bit打包就没问题了。

# 四、实验总结

## （一）学习感悟

感觉文法是一种逻辑特别清晰的东西，它虽然很难，但是按照算法一步一步写出来之后，它就能以不变应万变。有时候我自己都没反应过来的结果，代码却帮我算对了。

## （二）编码心得

一个星期，为了一个小作业，写了一千多行代码。因为提前编写的实验文档做理论准备，所以写起来特别的思路明确，也考虑到了松耦合。有种一点一点盖房子的感觉了，蛮有成就感。

# 五、参考文献

（1）[(2条消息) 包教包会！十分钟搞定自顶向下分析——编译原理速成计划\_smilestruggler的博客-CSDN博客\_编译原理速成](https://blog.csdn.net/toohandsomeIeaseId/article/details/105010226)

（2）老师的PPT