技术报告

韩露露

1.作业一：CFG 转 PDA

1.1 环境要求

1.C++版本至少是 C++11 以上，使用了新特性，低于此版本的 C++对某些新特性语法识别编译不了。

2.Qt 版本使用 5.9.9,由于 Qt 版本更新迭代的问题，所以不建议使用更高的版本。

3.由于 Qt 的跨版本特性，操作系统不限，支持 Linux，Windows，Android，Mac 等平台，但具体依赖需要自行配置。

1.2 主要参考资料

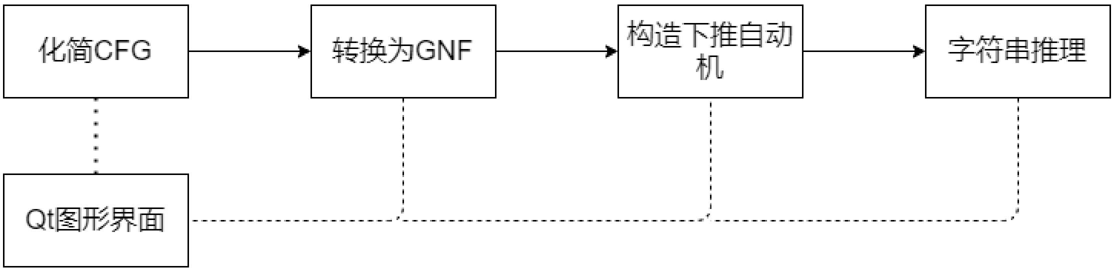
1.吴哲辉《形式语言与自动机理论》，北京，机械工业出版社，2007 年 4 月.

2.J.Hopcroft， J.D.Ullman，《Introduction to Automata Theory， Language and Computation》California，Addison-Wesley Publishing Company，1979.（2002 年清华，影印版）.

3.刘田译，自动机理论、语言和计算导论，机械工业出版社，2005.

4.蒋宗礼，形式语言与自动机理论，清华大学出版社，2003.

1.3 设计流程



算法过程：化简 CFG→转换为 GNF→构造自动机→字符串推理。整个过程使用 Qt 实现图形界面。

1.3.1 化简 CFG

该算法分为三部分：消除 ε 产生式→消除单产生式→消除无用产生式。并且经过资料参考和验证，该顺序不可随意调换，否则化简不完全。

1.消除 ε 产生式

A→ε 型的产生式称为空产生式。对于一个文法 G=(V,T,P,S)，如果 ε 不属于 L(G)，那么 G 中的空产生式就没有存在的必要，应该从文法的产生式集中删去。但是简单地划去空产生式，会使得新的文法丢失了一些文法的推导，因此当我们在一个文法中删去空产生式时，还需要添加上另外一些产生式，以便原文法中出现的各种可能的推导在新的文法中有所替代。

1.1.化简方法如下：

1)找出 G 中那些能够推导出空串的变量，记作这些变量的集合为：$V_\varepsilon=\{A\in V|A^*\Longrightarrow\varepsilon\}$

2)对于每个 $X_i\in V_\varepsilon$，当 G 中存在产生式 $A\to X_1\cdots X_{i-1}X_iX_{i+1}\cdots X_n$ 时，在 G’中增加一个产生式 $A\to X_1\cdots X_{i-1}X_{i+1}\cdots X_n$ 从 G 中直接删除所有的空产生式，得到 G’。

注：如果文法本身能够接受空串，那么实际上不可以消除空产生式，但是 GNF 的表达式本身不存在空产生式，所以文法一开始消除空产生式是合理的。

1.2.部分代码如下:

```
//寻找所有的可以推导空字符的可空变量
QSet<QChar> CFG::nullAbleV(){
    QSet<QChar> old_set;
    QSet<QChar> new_set;
    for(int i=0;i<products.size();i++){
        if(products[i].right=="#")
            new_set.insert(products[i].left);
    }
    while(old_set!=new_set){
        old_set=new_set;
        for(int i=0;i<products.size();i++)
        {
            if(isInSet(products[i].right,old_set))
                new_set.insert(products[i].left);
        }
    }
    return new_set;
}
//对一条产生式进行去除空产生式并派生新的产生式的操作
void CFG::dealOnePro(Production p){
    QQueue<int> queue;//记录可空变量在产生式右边的位置
    for(int i=0;i<p.right.length();i++){
        QChar ch=p.right[i];
        if(nullable_V.contains(ch))
             queue.push_back(i);
    }
    if(!queue.size()){
        if(!isInNOepsilonPro(p))//产生式右边无可空变量，则直接添加
            noEpsilonPro.push_back(p);
        return;
    }
    else{
        int count=0;//count 记录将符号删除的次数
        recurDerive(p,queue,count);
    }
}
//递归的对产生式右部的可空符号进行依次删除和派生新产生式
void CFG::recurDerive(Production p, QQueue<int> queue,int count)
{
    if(!queue.size())
        return;
    int pos=queue.front();//可空变量的原始位置
    queue.pop_front();
```

```
    Production temp=p;
    if(!isInNOepsilonPro(temp)){//如果没有可删除的可空变量
        noEpsilonPro.push_back(temp);
    }
    recurDerive(temp,queue,count);//原始式子递归
    int index=pos-count;//现在可空变量的位置是原始位置减去变短的长度
    temp.right=p.right.remove(index,1);//删除可空变量
    if(!isInNOepsilonPro(temp)&&temp.right.length()!=0)
    {
        noEpsilonPro.push_back(temp);
    }
    recurDerive(temp,queue,count++);//对删除可空变量的式子递归
}
//消除空产生
void CFG::removeEposilonPro(){
    nullable_V=nullAbleV();//返回可空符号集
    for(int i=0;i<products.size();i++){
        if(products[i].right=="#")
            continue;
         dealOnePro(products[i]);
    }
}
```

2. 消除单一产生式

在 CFG 中，对 A,B∈V，A→B 称为单产生式。单产生式在文法中也是不必要的。在文法中消去单产生式的工作相对比较容易，下面分两种情况化简。

2.2.算法：

1)构造映射 Φ(X)={v|v∈{所有单产生式的右边变量}}；

2)将所有的非单产生式加入结果集中；

3)若 V∈Φ(X)，将所有 V 的非单产生的右部作为 X 产生式的右部添加到结果集中。

2.3.部分代码如下：

```
//消除单产生式
void CFG::removeSinglePro(){
    QMap<QChar,QSet<QChar>> map;
    QSet<QChar> new_set,old_set;
    for(int i=0;i<V_vec.length();i++){
        QChar ch=V_vec[i];
        old_set.clear();
        new_set.clear();
        new_set.insert(ch);
        //找到所有的单产生式
        while(new_set!=old_set){
            old_set=new_set;
```

```
            for(int j=0;j<noEpsilonPro.size();j++){
                QChar left=noEpsilonPro[j].left;
                QString right=noEpsilonPro[j].right;
                if(!old_set.contains(left)||right.length()>1)
                    continue;
                //如果产生式右边等于左边则添加进入单一映射的集合中
                QChar temp=right[0];
                if(V_set.contains(temp)){
                    new_set.insert(temp);
                }
                else
                continue;
            }
        }
        //添加到映射中
        map[ch]=old_set;
        Production p;
        p.left=ch;
        for(int j=0;j<noEpsilonPro.size();j++){
            Production p1=noEpsilonPro[j];
            if(!map[ch].contains(p1.left))
                continue;
            //将所有非单产生式加入
            if(p1.right.length()>1){
                p.right=p1.right;
                if(!isInNOsinglePro(p))
                    noSinglePro.push_back(p);
            }
            else{
                if(V_vec.contains(p1.right[0]))
                    continue;
                else
                    p.right=p1.right;
                if(!isInNOsinglePro(p))
                    noSinglePro.push_back(p);
            }
        }
    }
}
```

3.消除无用产生式

字符 $X \in V \cup T$，如果存在一个推导：$S \overset{*}{\Longrightarrow} \alpha X \beta \overset{*}{\Longrightarrow} w$，$\alpha, \beta \in (V \cup T)^*, w \in T^*$那么就说 X 是文法 G 中的一个有用字符，否则说 X 是一个无用字符，包括两种情况。

3.1.消除非“产生的”符号和产生式：

1)每个 T 中符号都是产生的;
2)A→α∈P，如果 α 中每个符号都是可产生的，则 A 是可产生的。
对 A∈V，不存在 w∈T*使得 A*⟹w。这种情况的无用字符只可能是变量。
算法就是找出那些不能推导出终极符串的变量,基本思想是:
1)若存在产生式 A→w,(w∈T^*)，那么从 A 可以推导出终极符串;
2)若已求出部分可以推导出终极符串的变量，且有产生式 A→α，其中 α 只含终极符和那些已知可推导出终极符串的变量，则从 A 也可以推导出终极符串;
3)反复使用 2)，直到再也不能求出新的可推出终极符串的变量。剩下的变量就为所求。

3.2.消除非“可达的”符号和产生式:

1)符号 S 是可达的;
2)A→α∈P，如果 A 是可达的，则 A 是可达的，则 α 中每个符号都是可达的。
对∀α∈$(V\cup T)^*$，若 S⟹α，则 α 中不含 X。即 X 是不出现在任何句型中的字符。
这种情况下的无用符号 X 既可能是变量，也可以是终极符。
算法就是寻找这种不出现在任何句型中的变量或终极符。基本思想是:
1)所有出现在 S-产生式右边的变量和终极符都可以出现在句型中;
2)若已知 A 是可以出现在句型中的变量，则所有出现在 A-产生式右边的变量和终极符都是可以出现在句型中的字符;
3)反复使用 2)，直到找不出新的字符为止，剩下的变量或终极符就是那些不出现在任何句型中的字符。

3.3.部分代码如下:

```
void  CFG::removeNotUsePro(){
    QSet<QChar> T_use;//有用终结符集合
    QSet<QChar> V_use;//有用非终结符集合
    QVector<Production> pTemp;
    QSet<QChar> vTemp;
    bool flag;
    //先删除非产生的式子和集合元素
    //遍历第一次找到 A->a 的可产生变量
    for (auto i :noSinglePro) { //遍历上一次处理的产生式
        if (isInSet(i.right, T_set)&&(!vTemp.contains(i.left)))
//判断右边都是终结符
            vTemp.insert(i.left);//添加可产生的变量
    }
     //循环迭代，反复查找在右边是否有可产生的变量，并添加
     do{
        flag=false;
        for (auto i :noSinglePro) { //遍历产生式
            if (isInSet(i.right,
vTemp+T_set)&&(!vTemp.contains(i.left))){//判断右边只有可产生的变量和
终结符
                flag=true;
                vTemp.insert(i.left);//添加可产生的变量
            }
```

```
        }
    }while(flag);
    for (auto i :noSinglePro) { //添加有用产生式
       if (isInSet(i.left,vTemp)&&isInSet(i.right,vTemp+T_set))
       pTemp.append(i);
    }
    //再删除不可达的式子和集合元素
    //首先放入 S
    V_use.insert('S');
    do{
       flag=false;
       for (auto i :pTemp) //遍历产生式
          if(isInSet(i.left,V_use))//左边在可达的变量集合中
             for(auto k:i.right){
                if(isInSet(k,T_set)&&(!T_use.contains(k))){//把右
边终结符放到可达终结符集合中
                   flag=true;
                   T_use.insert(k);
                }
                if(isInSet(k,vTemp)&&(!V_use.contains(k))){//把右
边非终结符放到可达非终结符集合中
                   flag=true;
                   V_use.insert(k);
                }
             }
       } while(flag);
     for (auto i :pTemp) { //添加有用产生式
        if (isInSet(i.left,V_use)&&isInSet(i.right,V_use+T_use))
        useProducts.append(i);
     }
     //删除符号向量中未出现的字符
     QVector<QChar> temp=T_vec;
     for(QVector<QChar>::iterator
i=T_vec.begin();i!=T_vec.begin();i++){
        if(!T_use.contains(*i))
           temp.erase(i);
     }
     T_vec=temp;
     temp=V_vec;
     for(QVector<QChar>::iterator
i=V_vec.begin();i!=V_vec.begin();i++){
        if(!V_use.contains(*i))
           temp.erase(i);
     }
```

```
        V_vec=temp;
}
```

顺序不可调换，先消除非“产生的”，再消除非“可达的”，否则化简可能不完整。

1.3.2 转换为 GNF

参考资料分为三个步骤：

1.步骤 1

构造 $G_1=(V_1,T,P_1,S)$，使得 $L(G_1)=L(G)$，并且 G_1 中的产生式都形如：

$$A \to A_1A_2\cdots A_m$$

$$A \to a\, A_1A_2\cdots A_{m-1}$$

$$A \to a$$

其中 $A,A_1,A_2,\cdots,A_m \in V_1$,$a \in T$，$m \geq 2$。

1.1.算法如下：

1)对于 P 中的每一个产生式 $A \to \alpha$，如果 $\alpha \in T \cup V^+ \cup TV^+$，则直接将 $A \to \alpha$ 放入 P_1；

2)否则，对 $A \to \alpha$ 进行如下处理：

设 $\alpha=X_1X_2\cdots X_m$,则对于每一个 $X_i,i\geq 2$，如果 $X_i= a \in T$ ，则引入新变量 A_a（将它放入 V_1）和产生式 $A_a \to a$（将它放入 P_1），并且用 A_a 替换产生式 $A \to \alpha$ 中的 X_i

3)将处理后的形如 $A \to A_1A_2\cdots A_m$ 或者 $A \to a\, A_1A_2\cdots A_{m-1}$ 的产生式放入 P_1

1.2.部分代码如下：

```
void GNF::generateG1 (QVector<QChar> T, QVector<QChar> V,
QVector<Production> p){
    for(auto i:T){//将终结符中'a'->"a"
      QString temp;
      temp+=i;
      t_vec.push_back(temp);
      t_set.insert(temp);
      trans[i]=temp;
    }
    for(auto i:V){//将变量中'S'->"A1"
       QString temp;
       temp="A"+QString::number(A_count);
       v_set.insert(temp);
       A_count++;
       trans[i]=temp;
    }
    for(auto i:p){
       GNFProduction gnf_p;
       gnf_p.left=trans[i.left];
       if(isInVset(i.right)){
          for(auto j:i.right)
             gnf_p.right.push_back(trans[j]);
       }
```

```
        else{
            gnf_p.right.push_back(trans[i.right[0]]);
            for(int j=1;j<i.right.size();j++){
                QChar ch=i.right[j];
                if(!(ch>='A'&&ch<='Z')){//新增非终结符替换终结符
                    if(!t_replace.contains(ch)){
                        QString temp="A"+QString::number(A_count);
                        v_set.insert(temp);
                        v_vec.push_back(temp);
                        A_count++;
                        t_replace[ch]=temp;
                        GNFProduction p;
                        p.left=temp;
                        p.right.push_back(QString(ch));
                        G1.push_back(p);
                    }
                  gnf_p.right.push_back(t_replace[ch]);
                }
                else
                 gnf_p.right.push_back(trans[ch]);
            }
        }
        G1.push_back(gnf_p);
    }
}
```

2.步骤 2

设 V1={$A_1,A_2,\cdots,A_m$}，构造 $G_2=(V_2,T,P_2,S)$，使得 $L(G_2)=L(G_1)$ ，并且 G_2 中的产生式都形如：

$$A_i \to A_j\alpha \ (i<j)$$
$$A_i \to a\alpha$$
$$B_i \to \alpha$$

其中 V2=V1∩{$B_1,B_2,\cdots,B_m$}，$a\in T$, $\alpha \in V_2^*$

2.1.算法如下：

1）标记产生式 $A_k \to A_j\alpha$ (k>j)。设 $A_j \to \gamma_1|\gamma_2|\cdots|\gamma_n$ 为所有的 A_j 产生式,将产生式组 $A_k \to \gamma_1\alpha|\gamma_2\alpha|\cdots|\gamma_n\alpha$ 添加到产生式集合 P_2 中；

2）设 $A_k \to A_k\alpha_1|A_k\alpha_2|\cdots|A_k\alpha_p$ 是所有的右部第一个字符为 A_k 的 A_k 产生式，$A_k \to \beta_1|\beta_2|\cdots|\beta_q$ 是所有其他的 A_k 产生式。标记所有的 A_k 产生式，并引入新的变量 B，将下列产生式添加到产生式集合 P_2 中：

$$A_k \to \beta_1 |\beta_2|\cdots|\beta_q,$$
$$A_k \to \beta_1 B|\beta_2 B|\cdots|\beta_q B,$$
$$B \to \alpha_1| \alpha_2|\cdots| \alpha_p,$$
$$B \to \alpha_1 B| \alpha_2 B|\cdots| \alpha_p B$$

3)将 P_1 中未被标记的产生式全部都添加到产生式集合 P_2 中

2.2.部分代码如下：

```
void GNF::generateG3(){
    int size=A_count-1;
    sort(G2.begin(),G2.end(),comp);
    QVector< QVector<GNFProduction> > v1(size);
    QVector<GNFProduction>  vb;
    QVector< QVector<GNFProduction> > v2(size);
    QVector<GNFProduction>  vb2;
    for(auto i:G2){
       if(i.left[0]=='A'){
          int n=i.left.mid(1).toInt();
          v1[n-1].push_back(i);
       }
       if(i.left[0]=='B'){
          vb.push_back(i);
       }
    }
    v2[size-1]=v1[size-1];//下标最大的 Ai 右侧第一个字符一定是终结符，直接加入
    for(int i=size-2;i>=0;i--){//从下标第二大的 Ai 开始遍历
        for(auto k:v1[i]){
            if(t_set.contains(k.right[0]))//右侧第一个字符为终结符，直接加入
            {
                v2[i].push_back(k);
                continue;
            }
            else
            {
                int m=k.right[0].mid(1).toInt();//获取右侧第一个非终结符的下标，一定大于左侧非终结符的下标;
                k.right.removeFirst();
                for(auto n:v2[m-1]){//遍历下标为 m 的已经符合要求的产生式
                    GNFProduction p;
                    p.left=k.left;
                    p.right+=n.right;
                    p.right+=k.right;
                    v2[i].push_back(p);
                }
            }
        }
    }
    for( auto i:vb){
```

```
        if(t_set.contains(i.right[0])){//右侧第一个字符非终结符，直接加入
            vb2.push_back(i);
            continue;
        }
        else{
            int n=i.right[0].mid(1).toInt();//获取右侧第一个字符的下标
            i.right.removeFirst();
            for(auto j:v2[n-1]){
                GNFProduction p;
                p.left=i.left;
                p.right+=j.right;
                p.right+=i.right;
                vb2.push_back(p);
            }
        }
    }
    for(auto i:v2)
        G3+=i;
   G3+=vb2;
}
```

3.步骤 3

构造 $G_3=(V_3,T,P_3,S)$，使得 $L(G_3)= L(G_2)$，并且 G_3 中的产生式都形如：

$$A\rightarrow a\ A_1A_2\cdots A_{m-1}$$

$$A\rightarrow a$$

其中 $A,A_1,A_2,\cdots,A_m\in V_3$,$a\in T$，$m\geq 1$。

3.1.算法如下：

1)如果 $A_k\rightarrow A_j\beta$ $(k<j)$标记所有的 A_j 产生式 $A_j\rightarrow\gamma$，将产生式 $A_k\rightarrow\gamma\beta$ 放入 P_3;

2)用 P_3 中的产生式将所有的 B 产生式变换成满足 GNF 要求的形式。

3.2.部分代码如下:

```
void GNF::generateG2(){
    QVector<GNFProduction> old;//上一次的产生式
    G2=G1;
    //生成所有的 Ak->Ak;
    do{
        old=G2;//每次计算前赋值上一次产生式
        G2.clear();
    for(auto k :old){ //遍历产生式
        if((k.right[0].length()<=1)) { //左边第一个只有终结符
            if(!G2.contains(k))//去重
                G2.append(k);
        }
        else { //否则说明左边第一个是非终结符
if((readNumber(k.left)>readNumber(k.right[0]))&&(k.right[0][0]==k
.left[0])){//消除间接左递归
```

```
                for(auto j:old){
                    if(j.left==k.right[0]){ //找到所有 Aj 的产生式
                     QVector<QString> a=k.right;
                     a.removeFirst(); //保留右边第一个以后的式子
                     QVector<QString> r=j.right;
                     //添加 Ak->ra
                     GNFProduction Ak;
                     Ak.left=k.left;
                     Ak.right=r+a;
                     if(!G2.contains(Ak))//去重
                     G2.append(Ak);
                    }
                }
            }
            else{
                //其他的产生式加入 P2，不做处理
                if(!G2.contains(k))//去重
                    G2.append(k);
            }
        }
    }
    }while (old!=G2);//不再增加跳出循环
    //处理所有的 Ak->Ak;
    do {
        old=G2;//每次计算前赋值上一次产生式
        for(auto k :old){ //遍历产生式
if((readNumber(k.left)==readNumber(k.right[0]))&&(k.right[0][0]==
k.left[0])){ //找到所有的 Ak->Ak
            static int num=readNumber(k.left);
            if(num!=readNumber(k.left)){
                B_conut++;
            }
            QString B=QString("B%1").arg(B_conut);
            QVector<GNFProduction> Temp;
            for(QVector<GNFProduction>::iterator
m=G2.begin();m!=G2.end();m++){//删除 AK->Ak 式子
                if((*m).left==(*m).right[0]&&(*m).left==k.left){
                    //满足则删除
                }
                else{
                    Temp.append(*m);
                }
            }
            G2=Temp;
```

```
            for(auto j:old){ //找到所有Ak->AK的其他产生式
                if(j.left==k.right[0]&&j.left!=j.right[0]){
                    QVector<QString> a=k.right;
                    a.removeFirst(); //保留右边第一个以后的式子
                    //添加所有的Ak->b
                    if(!G2.contains(j))//去重
                    G2.append(j);
                    //添加Ak->bB
                    GNFProduction temp;
                    temp.left=j.left;
                    temp.right=j.right;
                    temp.right.append(B);
                    if(!G2.contains(temp))//去重
                    G2.append(temp);
                    //添加B->a;
                    temp.left=B;
                    temp.right=a;
                    if(!G2.contains(temp))//去重
                    G2.append(temp);
                    //添加B->aB;
                    temp.right.append(B);
                    if(!G2.contains(temp))//去重
                    G2.append(temp);
                }
            }
        }
        }
    }while(old!=G2);
}
```

1.3.3 构造 PDA

1) PDA 规则

非终结符:

产生式为 $A\to\alpha$，则添加规则：$\delta(q, \varepsilon, A) = (q, \alpha)$

终结符:

添加规则：$\delta(q, a, a) = (q, \varepsilon)$

2)判断字符串

使用递归进行推理字符串是否被文法接受

bool PDA::inference(QString str,int ptr, QStack<QString> stack, int count)

其中 str 是推断字符串，ptr 是读头，stack 是栈，count 是计数器，计数 stack 里的非终结符个数。

对于 $A\to\alpha$ 产生式直接压栈，读头不动，并且 count 计数增加，若读头字符与栈顶字符相同则进行弹栈，

读头右移，count 计数减少。进行函数递归，每次递归一次创建一个临时栈 stk 保存栈内容，防止破坏原有栈。

如果读头到最右边且 count=0，同时栈内符号都能推成 ε,则可以清空栈，返回 True，否则 False。

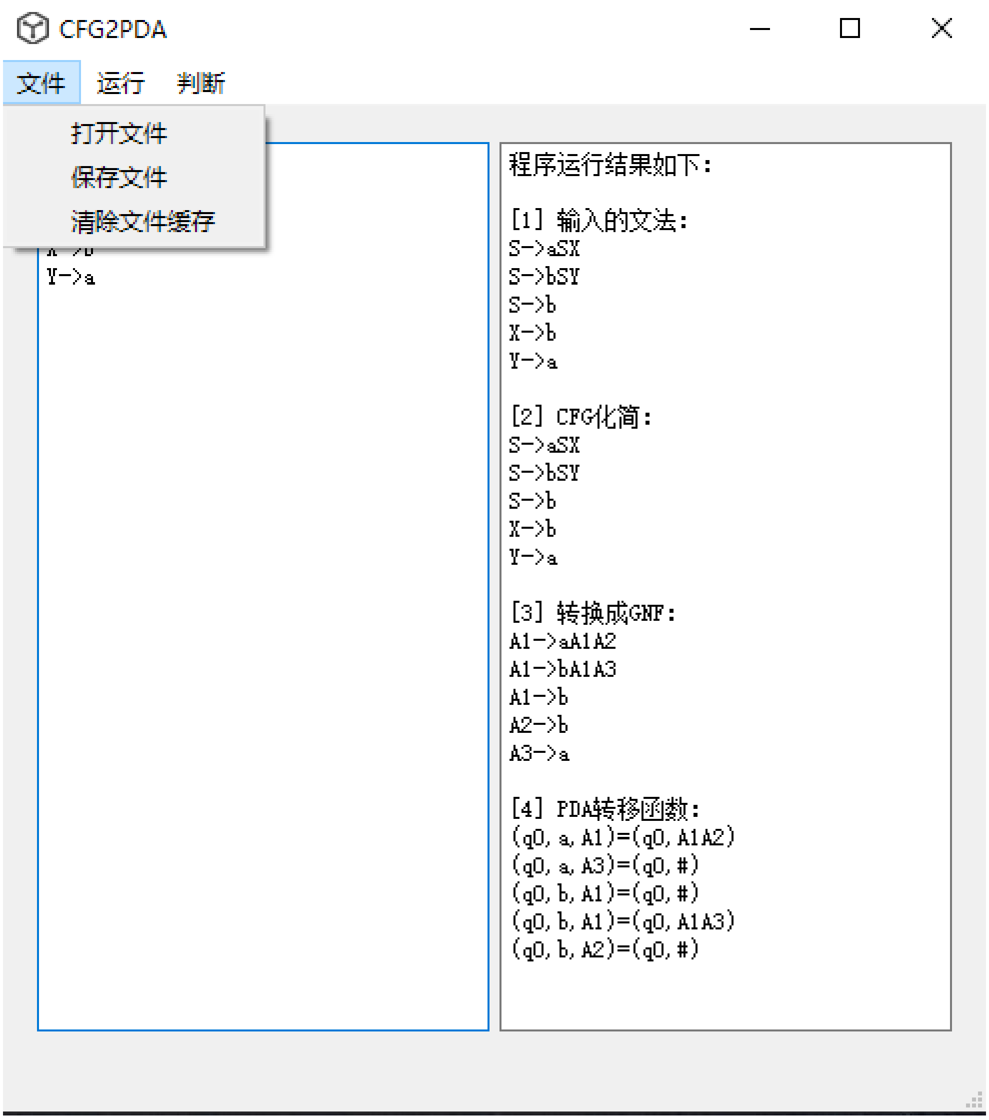
部分代码如下：

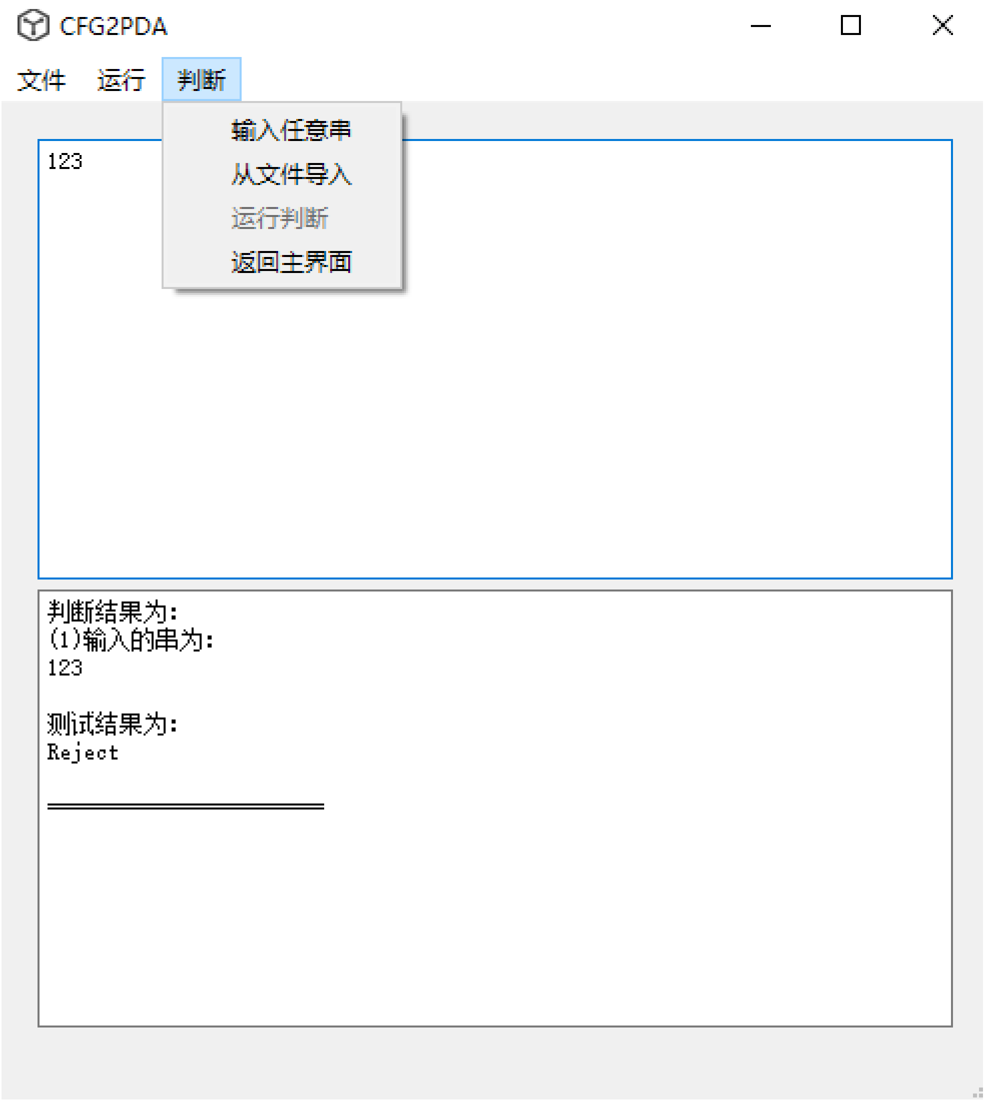
```
//迭代推理字符串是否被目标文法接收
bool PDA::inference(QString str,int ptr, QStack<QString> stack,
int count)
{
    bool flag = false;
    QVector<QVector<QString>> deltas;
    if(!str.length()){
        qDebug()<<"Error"<<endl;
        Result.append("Error\n");
        return false;
    }
    for(auto i:str){
        if(!this->t_set.contains(i)){
            qDebug()<<"Error"<<endl;
            Result.append("Error\n");
            return false;
        }
    }
    QStack<QString> temp=stack;
    while(!temp.isEmpty()){
        Result.append(temp.top());
        qDebug()<<temp.pop()<<",";
    }
    if(!stack.isEmpty())
    Result.append("->");
    qDebug()<<endl;
    //如果匹配完成,栈内没有残留的非终结符,且栈内符号都能变成ε,则可以清空栈
   if (ptr == str.length()&&count==0) {
       //因为 count=0,所以栈内都是非终结符
       QStack<QString> stk = stack;
       while (!stk.empty()){
           bool epsilonFlag=false;
           QString tos = stk.pop();
           deltas = delta(EPSILON, tos);
           for (auto delta : deltas) {
               if (delta[0]==EPSILON){
                   epsilonFlag=true;
                   break;
               }
           }
           if (!epsilonFlag){
```

```
                qDebug()<<"Error"<<endl;
                Result.append("Error\n");
                return false;
            }
        }
        return true;
    }
     if (!stack.empty()) {
         if (!t_set.contains(stack.top())) {
             //空移动
             QString tos = stack.pop();
             deltas=delta(EPSILON,tos);
             for (auto delta : deltas) { //遍历所有的表达式
                 QStack<QString> stk=stack;
                 int cnt = count;
                 for (int i = 0; i < delta.length(); ++i) {
                     //从右向左入栈
                     if (this->t_set.contains(delta[delta.length() - i
- 1])) {//如果是终结符
                         cnt++;
                     }
                     stk.push(delta[delta.length() - i - 1]);
                 }
                 if (cnt > str.length() - ptr) {
                     qDebug()<<"Error"<<endl;
                     Result.append("Error\n");
                     return false;
                 }
                 flag = inference(str, ptr, stk, cnt);
                 if (flag) {
                     break;
                 }
             }
         } else {
             //判断是否符合输入串
             if (stack.top() == str[ptr]) {
                 QStack<QString> stk=stack;
                 QString toss=stk.pop();
                 ptr++;
                 count--;
                 flag=inference(str,ptr,stk,count);
             } else {
                 qDebug()<<"Error"<<endl;
                 Result.append("Error\n");
```

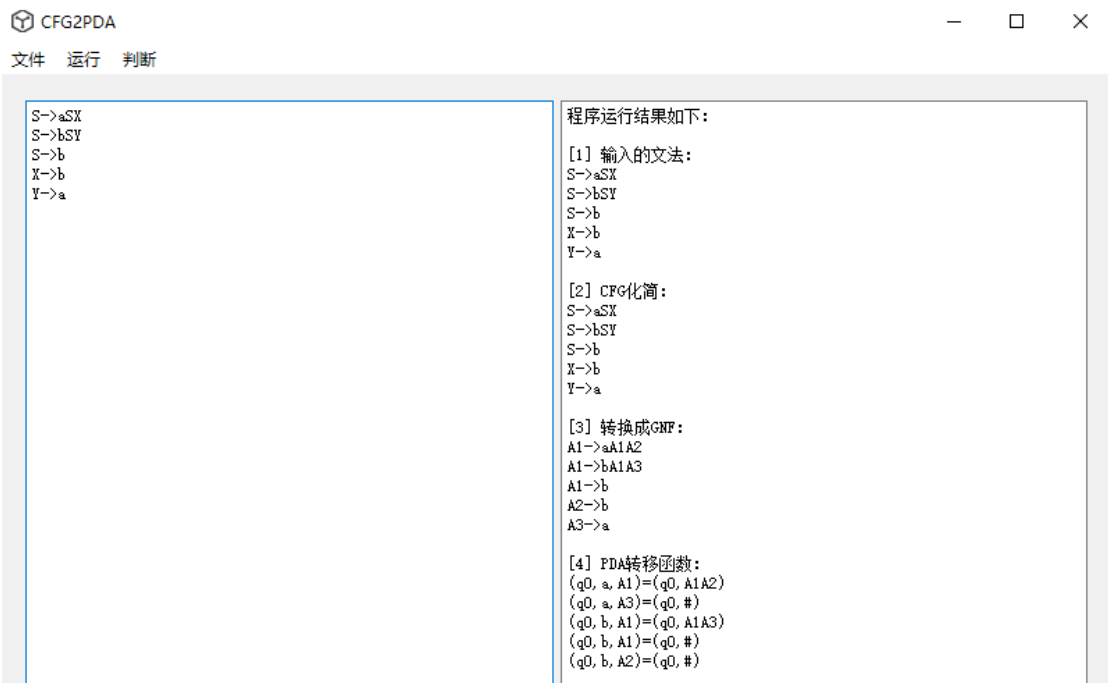
```
                return false;
            }
        }
    }
    return flag;
}
```

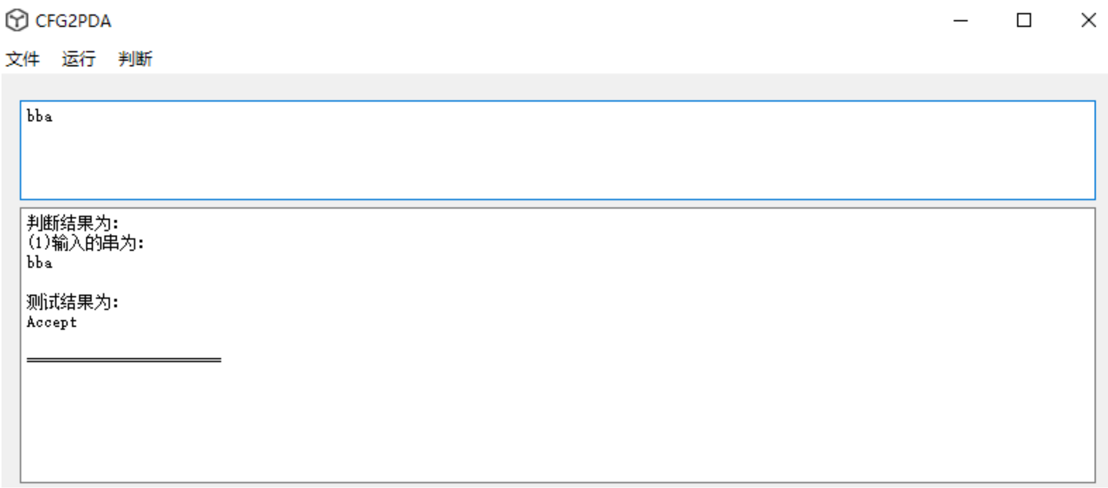
1.4GUI 界面设计

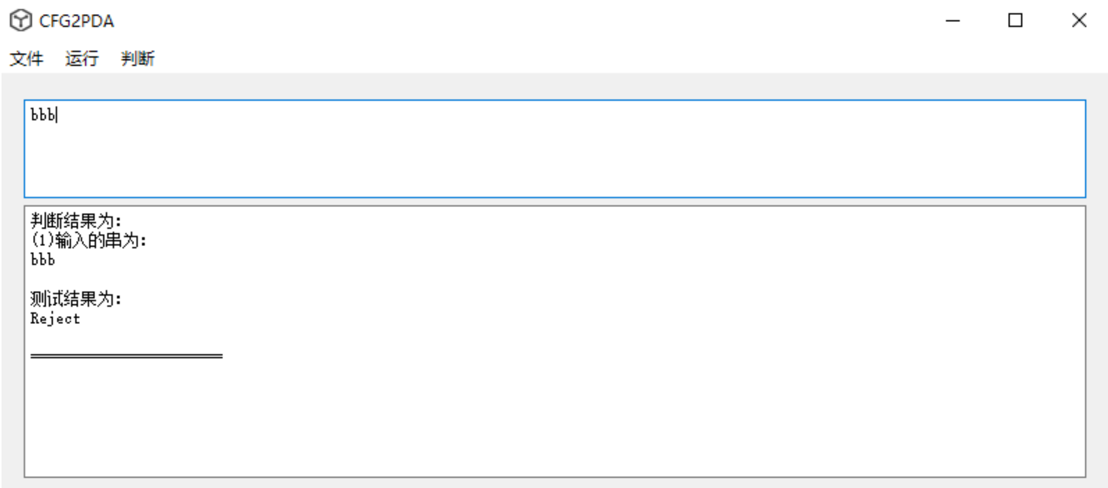



CFG2PDA
文件
运行
判断
输入任意串
从文件导入
运行判断
返回主界面
123
判断结果为:
(1)输入的串为:
123
测试结果为:
Reject

1.5 结果展示


CFG2PDA
文件
运行
判断
S->aSX
S->bSY
S->b
X->b
Y->a
程序运行结果如下:
[1] 输入的文法:
S->aSX
S->bSY
S->b
X->b
Y->a
[2] CFG化简:
S->aSX
S->bSY
S->b
X->b
Y->a
[3] 转换成GNF:
A1->aA1A2
A1->bA1A3
A1->b
A2->b
A3->a
[4] PDA转移函数:
(q0,a,A1)=(q0,A1A2)
(q0,a,A3)=(q0,#)
(q0,b,A1)=(q0,A1A3)
(q0,b,A1)=(q0,#)
(q0,b,A2)=(q0,#)





2.作业二：图灵机计算 x^y

2.1 环境要求

1.C++版本至少是 C++11 以上，使用了新特性，低于此版本的 C++对某些新特性语法识别编译不了。

2.2 主要参考资料

1.吴哲辉《形式语言与自动机理论》，北京，机械工业出版社，2007 年 4 月.

2.J.D.Ullman，J.D.Ullman，《Introduction to Automata Theory，Language and Computation》California，Addison-Wesley Publishing Company，1979.（2002 年清华，影印版）.

3.刘田译，自动机理论、语言和计算导论，机械工业出版社，2005.

4.蒋宗礼，形式语言与自动机理论，清华大学出版社，2003.

2.3 设计思路

本设计采用了三带图灵机结构，减少了很多状态，同时大幅度减少了读头的来回移动，实际的物理图灵机效率会提高。

2.3.1 初始化：

第一条带上放置 y 的数值，用 1 的个数表示 y；

第二条带上放置 x 的数值，用 1 的个数表示 x；

第三条带上放置输出的结果，在初始时设置成 100，其中 0 为分割符号，1 为初始 x^0 的值；

1	1	…	1	1	1	

y

1	1	…	1	1	1	

x

1	0	0				

x^{n-1}(初始为 $x^0=1$)

2.3.2 算法思路：

算法采取图灵机计算乘法的过程，带 1 读头每次将 y 中的 1 变为 a，则进行一次乘法，将带 2 上的 x 与带 3 上的 x^{n-1} 做乘法，然后将结果 x^n 覆盖掉带 3 上的 x^{n-1}，继续重复直到 y 中的 1 全变为 a，则带 3 最后的结果为 x^y。

2.3.3 算法流程：

1)读头 1 在带 1 上将 y 中 1 变为 a，并向右移动，进入 2)。若所有 1 变为 a，则停机;

2)读头 2 在带 2 上将 x 中的 1 变为 b，并向右移动，同时读头 3 在带 3 上将最右边的 0 替换为 $x^{n-1}0$;

3)重复 2)直到带 2 中的 1 全部为 b，得到 x^n0;

4)读头 3 用 x^n 替换带 3 的 x^{n-1}，读头 2 向左走带 2 上将 x 中的 b 全部变为 1,然后回到 1)。

2.3.4 转移函数：

$$\delta(q0, (1,1,1)) \vdash (q1, (a,1,1),(R,S,S))$$
$$\delta(q1, (1,1,1)) \vdash (q2, (1,b,1),(S,R,S))$$
$$\delta(q2, (1,1,1)) \vdash (q3, (1,1,c),(S,S,R))$$
$$\delta(q3, (1,1,1)) \vdash (q3, (1,1,1),(S,S,R))$$
$$\delta(q3, (1,1,0)) \vdash (q4, (1,1,0),(S,S,R))$$
$$\delta(q4, (1,1,0)) \vdash (q5, (1,1,1),(S,S,R))$$
$$\delta(q4, (1,1,1)) \vdash (q4, (1,1,1),(S,S,R))$$
$$\delta(q4, (1,1,\square)) \vdash (q5, (1,1,1),(S,S,R))$$
$$\delta(q5, (1,1,\square)) \vdash (q6, (1,1,0),(S,S,L))$$
$$\delta(q6, (1,1,0)) \vdash (q6, (1,1,0),(S,S,L))$$
$$\delta(q6, (1,1,1)) \vdash (q6, (1,1,1),(S,S,L))$$
$$\delta(q6, (1,1,c)) \vdash (q2, (1,1,c),(S,S,R))$$
$$\delta(q2, (1,1,0)) \vdash (q7, (1,1,0),(S,S,L))$$
$$\delta(q7, (1,1,c)) \vdash (q7, (1,1,1),(S,S,L))$$
$$\delta(q7, (1,1,\square)) \vdash (q1, (1,1,\square),(S,S,R))$$
$$\delta(q1, (1,\square,1)) \vdash (q8, (1,\square,1),(S,L,S))$$
$$\delta(q8, (1,b,1)) \vdash (q8, (1,1,1),(S,L,S))$$
$$\delta(q8, (1,\square,1)) \vdash (q9, (1,\square,1),(S,R,S))$$
$$\delta(q9, (1,1,1)) \vdash (q8, (1,1,d),(S,S,R))$$
$$\delta(q9, (1,1,0)) \vdash (q10, (1,1,1),(S,S,R))$$
$$\delta(q10, (1,1,1)) \vdash (q10, (1,1,1),(S,S,R))$$

$\delta(q10, (1, 1, 0)) \vdash (q11, (1, 1, \square), (S,S,L))$
$\delta(q11, (1, 1, 1)) \vdash (q12, (1, 1, 0), (S,S,L))$
$\delta(q12, (1, 1, 1)) \vdash (q12, (1, 1, 1), (S,S,L))$
$\delta(q12, (1, 1, d)) \vdash (q10, (1, 1, 1), (S,S,R))$
$\delta(q12, (1, 1, \square)) \vdash (q0, (1, 1, \square), (S,S,R))$
$\delta(q0, (1, 1, \square)) \vdash (qf, (1, 1, \square), (S,S,S))$

2.3.5 部分代码如下：

```
//使用向量作为带
class TM {
public:
    TM(int, int);
    void run();
    void print();
    int result();
private:
    int rw1, rw2, rw3;
    vector<char> tape1, tape2, tape3;
    int state;
};
void TM::run() {
    while (state != 13) {
        switch (state) {
            case 0:
                if (rw1 == tape1.size()) {
                    state = 13;
                } else if (tape1[rw1] == '1') {//q0 带 1 所有 1 依次变为
a,进入 q1
                    state = 1;
                    tape1[rw1] = 'a';
                    rw1++;
                }
                break;
                …
                …
                …
        {
{
```

2.3 程序结果：

```
hanlulu@Ltopia:/mnt/c/Users/HanLulu/Desktop/TuringMachine$ ./TM
compute x^y, please input x and y, Enter is end!
x:2
y:2
initializing...
print tape1:
11
print tape2:
11
print tape3:
100
computing...
print tape1:
aa
print tape2:
11
print tape3:
11110
result:
4
```

```
compute x^y, please input x and y, Enter is end!
x:5
y:5
initializing...
print tape1:
11111
print tape2:
11111
print tape3:
100
computing...
result:
3125
hanlulu@Ltopia:/mnt/c/Users/HanLulu/Desktop/TuringMachine$ ./TM2
compute x^y, please input x and y, Enter is end!
x:6
y:6
initializing...
print tape1:
111111
print tape2:
111111
print tape3:
100
computing...
result:
46656
```

由于图灵机的特性，计算较大数时带会很长很长，所以会需要很长的时间，验证计算 6^6 时间等待可忍受（15s）。