# 自动机实验二

2020211597吴清柳 2020211257尹睿 2020211599谢卫凯

## 实验环境

本次实验采用python实现所需功能，环境如下：

* Python 3
* PyCharm / VsCode
* Windows /MacOS

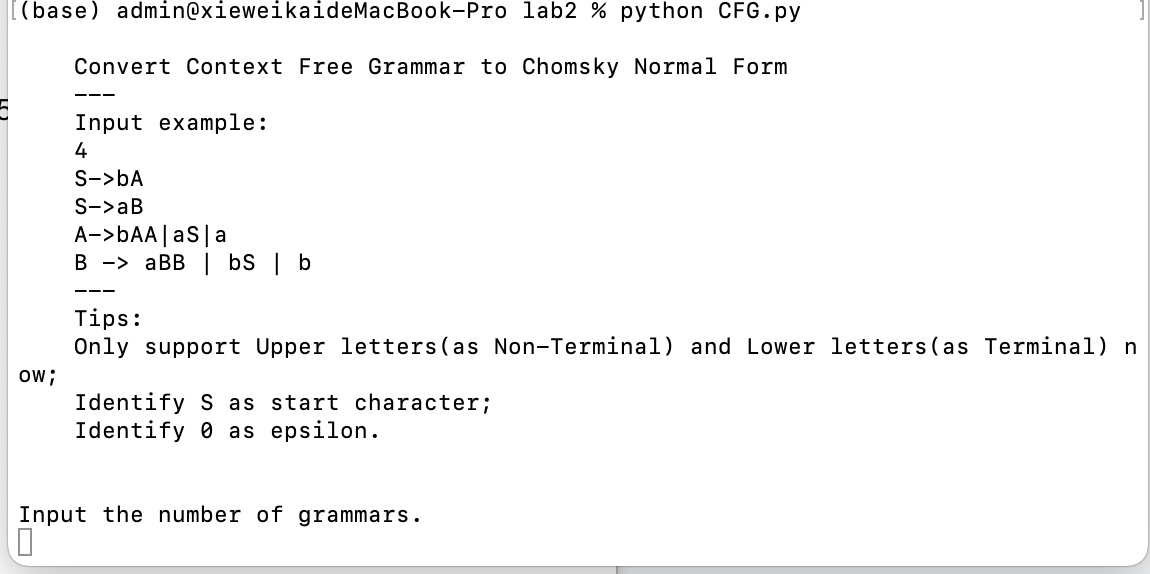
## 程序使用方法

### 程序功能

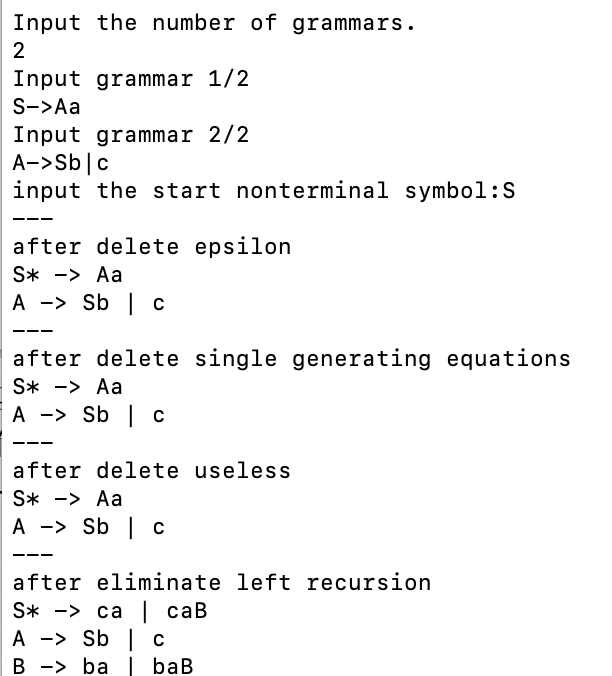
本次实验完成了所有实验要求的功能：去除epsilon产生式、去除单产生式、去除无用符号。同时额外添加了消除左递归的功能, 以及生成Chomsky范式的功能。

### 使用方法

本实验的程序源代码为cfg.py,使用时只需要简单执行该文件即可，执行后输出如下



输入方式：首先输入产生式的条目数，接着输入和产生式条目数匹配的产生式，注意输入的产生式中大写字母代表非终结符、小写字母代表终结符，字符0代表epsilon(代表epsilon的字符可以在源代码开头处修改)。在输入完产生式后还要求输入起始非终结符。按照以上要求输入后可以得出各个功能执行后产生的结果，如下图

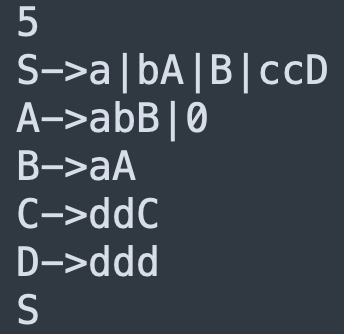


可以看到程序会依次输出消除epsilon产生式、消除单产生式、消除无用符号、生成Chomsky范式以及消除左递归后的结果。注意程序输出会给起始符号标\*。

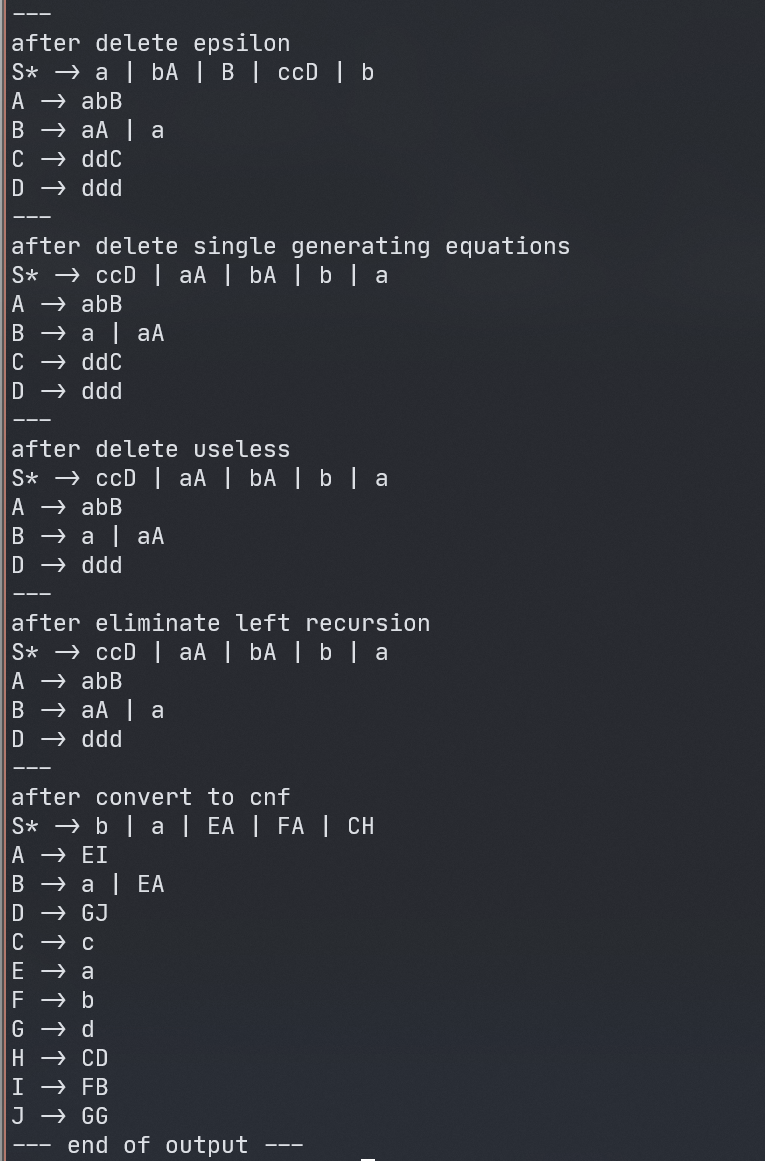
## 测试

### 例1

程序首先测试了实验安排要求中的测试样例，输入样例如下图



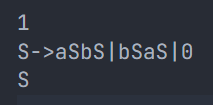
程序执行结果如下图



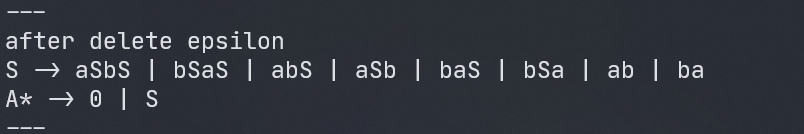
可以看到该示例程序执行正确。

### 例2

使用例题测试消除epsilon生成式



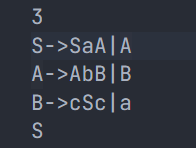
程序执行结果如下:



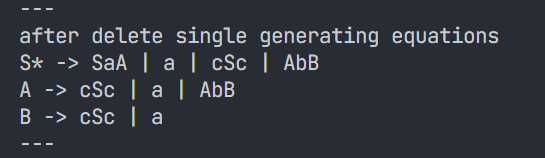
与课本答案一致

### 例3

使用例题测试消除单生成式



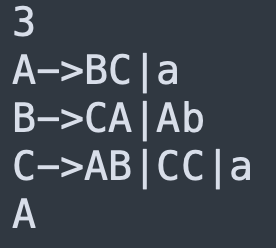
程序执行结果如下

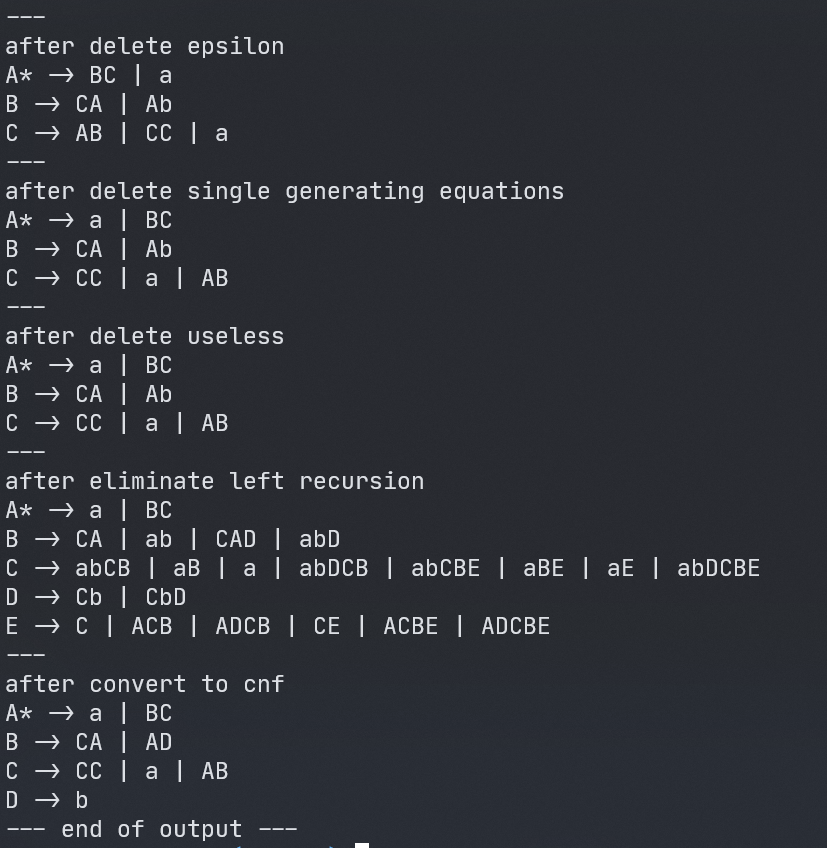


与课本答案一致。

### 例4

尝试使用书本中消除左递归的习题来进行测试，输入样例如下

程序执行结果如下



经检验，程序成功的消除了左递归。（注：因为程序中使用单个字符做非终结符，故不方便使用类似A’之类的符号，所以在消除左递归时没有形如A’的符号，全部采用未使用过的英文大写字母代替。

## 设计思路及核心算法

### CFG的表示

在程序中，使用一个CFG class来表示一个CFG，该类包含两个重要的属性

* start：起始非终结符对应的字符
* grammar:一个dict,用于标识产生式

若文法中包含形如A->w1 | w2这样的产生式，则grammar以非终结符对应符号为Key,即以A为key,value为一个list,list中包含w1,w2。程序中所有的算法均以这小节中描述的结构为基础。

### 消除epsilon产生式

程序首先将能一步推出epsilon的非终结符放入set中，接着不断尝试通过set中已有的非终结符能否找到更多的推出epsilon的非终结符，指导找不到为止。接着通过这个可空非终结符的集合，来依次改变grammar中的各条产生式（将各个可空非终结符去掉变成额外一条产生式，但不能出现全部去掉的情况，细节不在此多言）。

### 消除单产生式

对于每一个非终结符A，维护一个集合set,首先将A放入set,接着尝试通过set中的非终结符对应的产生式能否找到更多非终结符B使得A=>B,将B放入set中，不断寻找知道set不再变换为止。对于所有B属于set,若B->a,不为单产生式，则将A->a设为A的产生式。

### 去除无用符号

首先维护一个set,往内放入所有可以直接产生出终结符的非终结符，接着依旧是通过不断扩大集合的办法找到所有可以产生出终结符的非终结符，将所有非终结符集合和set作差，得到不可推出终结符号串的非终结符集合，将grammar中所有包含该集合内非终结符的产生式全部删除。

接着维护一个新的set,将起始符号放入，还是用扩大集合的办法，找出所有可达的非终结符和终结符集合。将所有不可达非终结符对应的产生式删除。

### 生成Chomsky范式

在前三步的铺垫后, 上下文无关文法已经满足了转换为Chomsky范式的前置需求。之后执行如下两步操作:

第一步, 遍历所有非终结符及其生成式, 对其每一个生成式, 如果含有终结符, 且生成式包含超过两个符号, 则将此处的此终结符(记为e)用如下的方式转换为非终结符: 如果已经存在非终结符, 它有可以生成此终结符的单生成式, 则替换为此非终结符; 否则, 引入新非终结符(如E)和单生成式E->e。重复此操作, 直到所有非终结符及其文法都被遍历。

第二步, 将所有非终结符及其生成式遍历, 在每个非终结符E的遍历期间维护一个队列, 首先进行初始化: 对于其每一个生成式, 如果长度大于2, 则放入队列, 并删除此生成式。之后对队列进行操作: 弹出队首元素, 将其生成式右端前两个字符用一个非终结符代替, 这个非终结符有生成代替的两个字符的生成式。如果这个非终结符和生成式是新添加的, 则添加到文法中。如果代替后的生成式长度大于2, 放入队列。反之, 放入非终结符E的生成式列表中。如此操作直到队列清空, Chomsky范式转换完成。

### 消除左递归

将非终结符按一定顺序排列（即将grammar的key随意进行一个排序），按照这个排序依次处理非终结符，处理过程如下

1. 若存在A->w 且w的第一个符号是排在A前面的非终结符，将该非终结符代入，使得不存在这种产生式。
2. 消除直接左递归，即将形如A->Aa | b转化为A->b | bA’ A’->a | aA’的形式（由于该程序中使用单一符号代表非终结符，故消除左递归是没有A’形式的非终结符，而是用一个新的大写字母代替A’这样形式的非终结符）。
3. 若全部非终结符处理完毕，结束，否则转1处理下一个非终结符。

## 改进思路

程序在实现消除单产生式是，对于每一个非终结符A,都从头求出了其对应的N\_A集合，事实上在这里可以对性能进行进一步的改进提升。容易发现，在求N\_A时其实可以顺带求出N\_B,其中B为满足A=>B的非终结符，在程序中可以稍作改进，不必每一次都从头求起，重复计算浪费了时间。一种思路是同时维护所有的集合N\_A,其中A为该CFG的非终结符。在求某个N\_A时，同时改变多个集合，避免重复计算。若同时想节省空间，另一种思路是维护一种链式结构，每个非终结符都能指向直接可以通过一个单产生式直接产生的非终结符，通过这种链式结构很容易得到每个非终结符A对应的N\_A内的内容。

另外，可以稍微对程序存储语法的结构稍加修改，使得程序支持形如A’ A\_1等等这样的非终结符形式，让程序更加灵活

## 分工

吴清柳：程序的主体设计, 和组员的协调分工。Chomsky范式转换部分。对应的文档编写。

尹睿：消除epsilon产生式和单生成式部分。程序的样例设计, 测试和程序改进。对应的文档编写。

谢卫凯：消除左递归部分和去除无用符号部分。程序的改进。以及对应功能的文档编写。

### Source Code List

grammars = {}

epsilon = '0'

l\_alphabet\_N = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',

                'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']

l\_alphabet\_T = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',

                'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

alphabet\_N = set(l\_alphabet\_N)

alphabet\_T = set(l\_alphabet\_T)

INF = 26

def replace\_idx(text, idx=0, pat=''):

*"""*

*replace pattern at text[idx]*

*"""*

*return* '%s%s%s' % (text[:idx], pat, text[idx+1:])

def comparator\_n\_idx(a):

*if* a[0] in l\_alphabet\_N:

*return* l\_alphabet\_N.index(a[0])

*else*:

*return* INF

class CFG:

    def \_\_init\_\_(self, grammar, start):

*self*.grammar = grammar

*self*.start = start

*self*.is\_CNF = False

*self*.is\_GNF = False

*# 消除epsilon生成式*

    def delete\_epsilon(self):

*"""*

*delete epsilon in CFG*

*"""*

        set0 = set(epsilon)

        set1 = set(epsilon)

*# use algorithm 1 to collect non-terminal A if there is a A->epsilon*

*for* item *in* *self*.grammar:

*if* epsilon in *self*.grammar[item]:

                set1.add(item)

*# recursively add all symbols can arrive epsilon*

*while* set0 != set1:

            set0 = set1.copy()

*for* k, v *in* *self*.grammar.items():

*for* jtem *in* v:

*if* *self*.in\_set(jtem, set0):

                        set1.add(k)

*break*

*# collect all non-terminal symbols that can arrive epsilon*

        set2 = set1.copy()

*# clean epsilon grammars*

        k\_to\_del = set()

*for* k, v *in* *self*.grammar.items():

            to\_del = set()

*for* jtem *in* v:

*if* k != *self*.start and jtem == epsilon:

                    to\_del.add(jtem)

*continue*

*for* ktem *in* set2:

                    i = jtem.find(ktem)

*while* i != -1 and len(jtem) > 1:

                        new\_n = replace\_idx(jtem, i)

*if* new\_n not in v:

                            v.append(new\_n)

                        i = jtem.find(ktem, i+1)

*for* jtem *in* to\_del:

                v.remove(jtem)

*# remove keys whose value list is empty*

*if* not v:

                k\_to\_del.add(k)

*# remove keys whose value list is empty*

*for* key *in* k\_to\_del:

*self*.grammar.pop(key, 0)

*# add new start symbol if there is a grammar: S->epsilon*

*if* *self*.start in set2:

*self*.grammar[*self*.start].remove(epsilon)

*self*.add\_s1()

    def add\_n(self, set=set()):

*"""*

*add new unused Non-Terminal*

*"""*

        set\_n = *self*.get\_set\_n()

        set\_n = set\_n.union(set)

*for* i *in* range(len(l\_alphabet\_N)):

*if* l\_alphabet\_N[i] not in set\_n:

*return* l\_alphabet\_N[i]

*return* str()

    def add\_s1(self):

*"""*

*add new start non-terminal*

*"""*

        new\_n = *self*.add\_n()

*self*.grammar.update({new\_n: [epsilon]})

*self*.grammar[new\_n].append(*self*.start)

*self*.start = new\_n

*return*

    def get\_set\_n(self):

*"""*

*get non-terminal set*

*"""*

        set\_n = set()

*for* k, v *in* *self*.grammar.items():

*if* k in alphabet\_N:

                set\_n.add(k)

*for* jtem *in* v:

*for* ktem *in* jtem:

*if* ktem in alphabet\_N:

                        set\_n.add(ktem)

*return* set\_n

    def get\_set\_t(self):

*"""*

*get terminal set*

*"""*

        set\_t = set()

*for* k, v *in* *self*.grammar.items():

*for* jtem *in* v:

*for* ktem *in* jtem:

*if* ktem in alphabet\_T:

                        set\_t.add(ktem)

*return* set\_t

    def in\_set(self, jtem, set):

*"""*

*if all elements in jtem belong with set, return true*

*else, return false*

*"""*

*# return set(jtem) <= set*

        flag = True

*for* c *in* jtem:

*if* c not in set:

                flag = False

*break*

*return* flag

*# 消除单生成式*

    def delete\_single\_generator(self):

*"""*

*消除单生成式*

*"""*

*for* item *in* *self*.grammar:

            set0 = set()

            set1 = set(item)

*while* set0 != set1:

                set0 = set1.copy()

*for* jtem *in* set0:

*if* jtem not in *self*.grammar:

*continue*

*for* ktem *in* *self*.grammar[jtem]:

*if* ktem not in set0 and len(ktem) == 1 and ktem in alphabet\_N:

                            set1.add(ktem)

*for* jtem *in* set1:

*if* jtem not in *self*.grammar:

*continue*

*if* len(*self*.grammar[jtem]) > 0:

*self*.grammar[item].extend(*self*.grammar[jtem])

*self*.grammar[item] = list(set(*self*.grammar[item]))  *# 去除重复*

*for* jtem *in* set1:

*if* jtem in *self*.grammar[item]:

*self*.grammar[item].remove(jtem)

*# 消除无用符号*

    def delete\_useless(self):

*"""*

*delete useless non-terminal and terminal symbols*

*"""*

        set\_n0 = set()

        set\_n1 = set()

*# collect terminal symbols*

*for* k, v *in* *self*.grammar.items():

*for* jtem *in* v:

*if* *self*.in\_set(jtem, alphabet\_T):

                    set\_n1.add(k)

*break*

*# collect non-terminal symbols can arrive terminals*

*while* set\_n0 != set\_n1:

            set\_n0 = set\_n1.copy()

*for* k, v *in* *self*.grammar.items():

*for* jtem *in* v:

*if* *self*.in\_set(jtem, alphabet\_T.union(set\_n0)):

                        set\_n1.add(k)

*break*

*# delete useless non-terminal symbol grammars*

        useless\_n = *self*.get\_set\_n() - set\_n1

*for* item *in* useless\_n:

*self*.grammar.pop(item, 0)

*# delete useless symbols in grammars*

*for* k, v *in* *self*.grammar.items():

            s = set()

*for* jtem *in* v:

*for* ktem *in* useless\_n:

*if* ktem in jtem:

                        s.add(jtem)

*for* jtem *in* s:

                v.remove(jtem)

*# end of algorithm 1 part*

        set\_0 = set(*self*.start)

        set\_1 = set(*self*.start)

*# collect all symbols start symbol can arrive directly*

*for* item *in* set\_0:

*for* jtem *in* *self*.grammar[item]:

*for* ktem *in* jtem:

                    set\_1.add(ktem)

*# collect all symbols start symbol can arrive*

*while* set\_0 != set\_1:

            set\_0 = set\_1.copy()

*for* item *in* set\_0:

*if* item in alphabet\_N:

*for* jtem *in* *self*.grammar[item]:

*for* ktem *in* jtem:

                            set\_1.add(ktem)

        use\_n = set\_1.intersection(*self*.get\_set\_n())

        useless\_n = *self*.get\_set\_n() - use\_n

*# delete symbols can't be accessed*

*for* item *in* useless\_n:

*self*.grammar.pop(item, 0)

*# 转换为Chomsky范式*

    def conv2cnf(self):

*"""*

*convert CFG without epsilon, useless symbols and 单生成式 to Chomsky Normal Form*

*"""*

        set\_t\_epsilon = alphabet\_T.copy()

        set\_t\_epsilon.add(epsilon)

        dict\_new\_g = dict()

*# convert terminal symbols appearing at the right end of the generating equations to non-terminal symbols*

*for* k, v *in* *self*.grammar.items():

            set\_del = set()

            set\_add = set()

*for* jtem *in* v:

*if* not *self*.in\_set(jtem, alphabet\_N):

*if* not jtem in set\_t\_epsilon:

                        new\_jtem = str(jtem)

                        set\_del.add(jtem)

*for* ktem *in* new\_jtem:

*if* ktem in set\_t\_epsilon:

                                new\_fr = *self*.t\_to\_n(ktem, dict\_new\_g)

                                new\_jtem = new\_jtem.replace(ktem, new\_fr, 1)

*if* new\_jtem not in v:

                            set\_add.add(new\_jtem)

*for* jtem *in* set\_del:

                v.remove(jtem)

*for* jtem *in* set\_add:

                v.append(jtem)

*self*.grammar.update(dict\_new\_g)

*# end of step 1*

*# shorten the length of the right end of the generating equations to no more than 2*

        dict\_new\_g = dict()

*for* k, v *in* *self*.grammar.items():

            q = set()

            set\_add = set()

            to\_del = set()

*for* jtem *in* v:

*if* len(jtem) > 2:

                    q.add(jtem)

                    to\_del.add(jtem)

*for* jtem *in* to\_del:

                v.remove(jtem)

*while* len(q) > 0:

                jtem = q.pop()

                new\_jtem = *self*.shorten\_g(jtem, dict\_new\_g)

*if* len(new\_jtem) > 2:

                    q.add(new\_jtem)

*else*:

                    set\_add.add(new\_jtem)

*for* jtem *in* set\_add:

                v.append(jtem)

*for* item *in* dict\_new\_g:

*self*.grammar.update({item: [dict\_new\_g[item]]})

    def shorten\_g(self, jtem, dict):

*"""*

*single step in Chomsky Normal Form conversion*

*"""*

        to\_shorten = jtem[1:3]

*# shorten the length of the right end of the generating equations by one at a time*

*for* k, v *in* *self*.grammar.items():

*if* len(v) == 1 and v[0] == to\_shorten:

                s = replace\_idx(jtem, 1, k)

                s = replace\_idx(s, 2)

*return* s

*for* k, v *in* dict.items():

*if* len(v) == 1 and v[0] == to\_shorten:

                s = replace\_idx(jtem, 1, k)

                s = replace\_idx(s, 2)

*return* s

        new\_n = *self*.add\_n(set(dict.keys()))

        dict.update({new\_n: to\_shorten})

        s = replace\_idx(jtem, 1, new\_n)

        s = replace\_idx(s, 2)

*return* s

    def t\_to\_n(self, terminal, dict):

*"""*

*return N if there is an N which has only one grammar N->terminal; if hasn't, add new grammar N->terminal, return N.*

*"""*

*for* k, v *in* *self*.grammar.items():

*if* len(v) == 1 and v[0] == terminal:

*return* k

*for* k, v *in* dict.items():

*if* len(v) == 1 and v[0] == terminal:

*return* k

        new\_n = *self*.add\_n(set(dict.keys()))

        dict.update({new\_n: [terminal]})

*return* new\_n

*# convertion: var->Aw  A->b  =>  var->bw*

    def replace\_first\_var(self, var, A):

        res = []

*for* item *in* *self*.grammar[var]:

*if* item[0] == A:

                tmp = [v + item[1:] *for* v *in* *self*.grammar[A]]

                res.extend(tmp)

*else*:

                res.append(item)

        res = list(set(res))

*self*.grammar[var] = res

*# 消除直接左递归*

    def eliminate\_direct\_left\_recursion(self, var):

        beta = []

        alpha = []

*for* item *in* *self*.grammar[var]:

*if* item[0] == var:

                alpha.append(item[1:])

*else*:

                beta.append(item)

*if* alpha == []:  *# no direct\_left\_recursion*

*return*

        new\_n = *self*.add\_n()

        beta.extend([v + new\_n *for* v *in* beta])

*self*.grammar[var] = beta

        alpha.extend([v + new\_n *for* v *in* alpha])

*self*.grammar[new\_n] = alpha

*# 消除左递归*

    def eliminate\_left\_recursion(self):

        sorted\_var = sorted(*self*.grammar)

*for* i *in* range(len(sorted\_var)):

*for* j *in* range(i):

*self*.replace\_first\_var(sorted\_var[i], sorted\_var[j])

*self*.eliminate\_direct\_left\_recursion(sorted\_var[i])

    def cfg\_to\_cnf(self):

*self*.delete\_epsilon()

*self*.delete\_single\_generator()

*self*.delete\_useless()

*self*.conv2cnf()

*self*.is\_CNF = True

    def printer(self):

*for* k, v *in* *self*.grammar.items():

            start\_flag = '\*' *if* k == *self*.start *else* ''

            print(f'{k}{start\_flag} -> {" | ".join(v)}')

def helper():

    print("""

    Context Free Grammar Converter

    - Elimite epsilon-generation

    - Elimite single-generation

    - Elimite useless symbols

    - Convert Context Free Grammar to Chomsky Normal Form

    - Elimite left recursion

    ---

    Input example:

    4

    S->bA

    S->aB

    A->bAA|aS|a

    B -> aBB | bS | b

    S

    ---

    Tips:

    Only support Upper letters(as Non-Terminal) and Lower letters(as Terminal) now;

    Identify S as start character;

    Identify 0 as epsilon.

    """)

def main():

    n = int(input("Input the number of grammars.\n"))

*for* i *in* range(n):

        str = input("Input grammar {}/{}\n".format(i+1, n))

        sstr = str.split('->')

        key = sstr[0].strip()

        vals = sstr[1].split('|')

        vals = [j.strip() *for* j *in* vals]

*if* key in grammars:

            grammars[key].extend(vals)

*else*:

            grammars.update({key: vals})

    start = input("input the start non-terminal symbol:")

    g = CFG(grammars, start)

    g\_cfg = CFG(grammars.copy(), start)

    print()

    print("---")

    g.delete\_epsilon()

    print("after delete epsilon")

    g.printer()

    print("---")

    g.delete\_single\_generator()

    print("after delete single generating equations")

    g.printer()

    print("---")

    g.delete\_useless()

    print("after delete useless")

    g.printer()

    print('---')

    g.eliminate\_left\_recursion()

    print("after eliminate left recursion")

    g.printer()

    print('---')

    g\_cfg.cfg\_to\_cnf()

    print("after convert to cnf")

    g\_cfg.printer()

    print('--- end of output ---')

*if* \_\_name\_\_ == "\_\_main\_\_":

    helper()

    main()