Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Институт информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"

профиль "Программное обеспечение средств вычислительной техники и автоматизированных систем"

Кафедра прикладной математики и кибернетики

**Курсовая работа по дисциплине  
 Теория языков программирования и методы трансляции**

Вариант 4

Выполнил:

Cтудент гр. ИП-813 \_\_\_\_\_\_\_\_\_\_\_\_\_/Бурдуковский И.А./

ФИО студента

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Проверил:

Ассистент кафедры ПМиК \_\_\_\_\_\_\_\_\_\_\_\_\_/Павлова У. В./

ФИО преподавателя

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г. Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Новосибирск 2021 г.

# Задание

*Вариант 4*

Написать программу, которая по предложенному описанию языка построит регулярную грамматику (ЛЛ или ПЛ – по заказу пользователя), задающую этот язык, и позволит сгенерировать с её помощью все цепочки языка в заданном диапазоне длин. Предусмотреть возможность поэтапного отображения на экране процесса генерации цепочек. Варианты задания языка:

Алфавит, начальная и конечная подцепочки и кратность длины всех цепочек языка.

# Описание алгоритма решения задачи

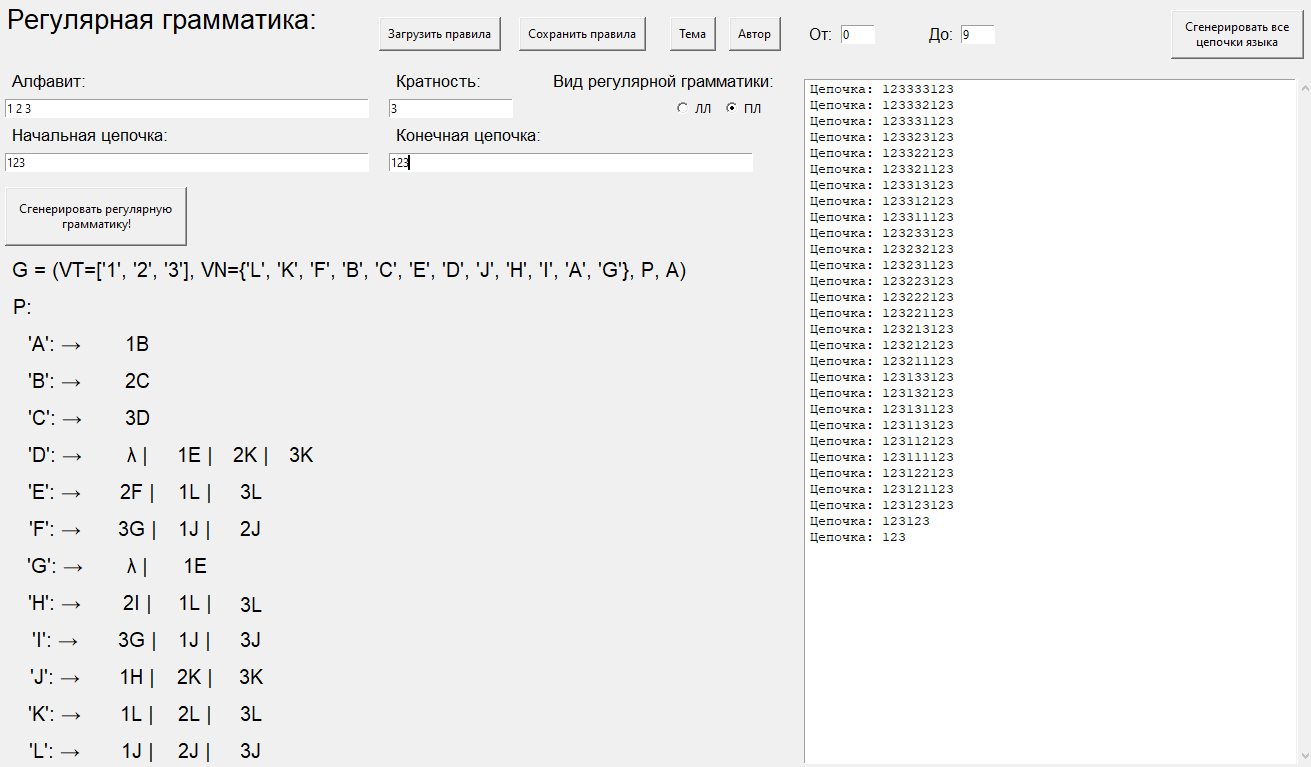
Регулярная грамматика строится следующему принципу:

1. Определяется сколько символов в начальной и конечной цепочке могут схлопнуться воедино.
2. Строятся правила для начальной подцепочки с переходами по целевым символам (символ по которому оно должно переходить в следующее правило подцепочки). Если подцепочки полностью схлопнулись, сохранив кратность, то конец начальной цепочки становится также конечным состоянием.
3. Строятся правила для конечной подцепочки с переходами по целевым символам, последнее правило становится конечным. Если подцепочки смогли объединиться друг с другом полностью или частично (с сохранением кратности) – то строится лишь часть правил из конечной цепочки (которая образует это объединение) и создаются ещё правила зеркальной конечной цепочки – они будут использоваться для генераций остальных цепочек нужной кратности, чтобы они смогли корректно оканчиваться на нужную подцепочку.
4. Для соблюдения кратности k для всей цепочки, строится k дополнительных правил, каждое по неиспользуемым в начале цепочки символам создают переход к следующему правилу. Последнее дополнительное правило переходит в первое, образуя петлю, позволяющую генерировать цепочку бесконечной длины, соблюдая кратность. Из одного дополнительного правила, сохраняющего кратность, по первому символу из конечной цепочки идёт переход к правилам конечной цепочки.
5. После генерации всех вспомогательных правил нужно пройтись по правилам конечной цепочки (зеркальной тоже) и по не целевым символам образовать переходы в дополнительные правила. Дополнительные правила выбираются так, чтобы сохранять кратность при генерации.

Для лево-линейной грамматики был придуман следующий алгоритм:

1. Входные начальная и конечная подцепочка меняются местами и отзеркаливаются.
2. Строится право-линейная грамматика по предложенным входным данным.
3. Во всех полученных правилах нетерминальный символ меняется местами с терминальным

После построения регулярной грамматики по полученным правилам проходит алгоритм генерации всех определённой длины из 1ой лабораторной работы.



Первым делом генерируется последовательность правил начальной подцепочки. Так как кратность равна 3 и подцепочки полностью схлопываются, то генерируется 4 правила и последнее будет иметь лямбда-выход:

A -> 1B

B -> 2C

C -> 3D

D -> λ | 1E

Затем к созданным правилам добавляются правила конечной подцепочки:

E -> 2F

F -> 3G

G -> λ

Далее создаются зеркальные конечные правила, т.к. подцепочки схлапываются:

H -> 2I

I -> 3G

Генерируются дополнительные правила, сохраняющие кратность и образующие возможность генерации бесконечно длинных цепочек.

J -> 1H | 2K | 3K

K -> 1L | 2L | 3L

G -> 1J | 2J | 3J

И наконец ранее сгенерированным правилам конечной цепочки (а также зеркальным) добавляем переходы в дополнительные состояния.

G -> λ | 3E

H -> 2I | 1L | 3L

I -> 3G | 1J | 3J

# 

# Описание основных блоков программы

Содержание файла kurs.py:

Функции:

* Machine\_input() – вводит начальные значения из файла
* Machine\_output() – сохраняет начальные значения в файл
* Generate\_chain\_button() – генерирует цепочки определённый длины по имеющимся правилам
* Generate\_grammar\_clicked() – считывает начальные данные для генерации грамматики из пользовательского интерфейса
* Find\_effective\_chain() – находит максимально возможное схлопывание подцепочек
* Generate\_grammar() – генерирует грамматику по начальным данным

Классы:

* Grammar является классом-контейнером для грамматики и позволяет строить цепочки по грамматике
* Rule является классом-контейнером для описания правил грамматики
* ExtraRule является классом-контейнером для описания дополнительных правил грамматики

# **Текст программы**

## **main.py**

from dataclasses import dataclass  
from typing import Dict, List, Any  
from tkinter import \*  
import string  
from os import path  
from tkinter import filedialog, messagebox  
from functools import partial  
import json  
  
min\_chain = 0  
start\_chain\_len = 0  
end\_chain\_len = 0  
window = Tk()  
normilize\_grammar = dict()  
  
entry\_alpabet = Entry(window, width=60)  
entry\_multiplicity = Entry(window, width=20)  
entry\_start\_chain = Entry(window, width=60)  
entry\_end\_chain = Entry(window, width=60)  
entry\_left\_border = Entry(window, width=5)  
entry\_right\_border = Entry(window, width=5)  
  
lbl\_err = Label(window, text="", font=("Arial", 15))  
lbl\_grammar = Label(window, text="", font=("Arial", 15), padx=15, pady=0)  
frame = Frame(master=window, padx=10, pady=5)  
  
text = Text(master=window, width=60, height=10, padx=5)  
  
r\_var = BooleanVar()  
r\_var.set(1)  
Radiobutton\_LL = Radiobutton(text='ЛЛ', variable=r\_var, value=0)  
Radiobutton\_PL = Radiobutton(text='ПЛ', variable=r\_var, value=1)  
  
  
@dataclass  
class ExtraRule:  
 uniq\_sym: str  
 multiplicity\_count: int  
 Rules: List[str]  
  
  
@dataclass  
class Rule:  
 uniq\_sym: str  
 key\_rule: bool  
 multiplicity\_count: int  
 next\_rule: str  
 Rules: List[str]  
  
  
@dataclass  
class Grammar:  
 VT: List[str]  
 VN: set()  
 Rules: List[Rule]  
 Extra\_Rules: List[ExtraRule]  
 Start\_state: str  
  
  
grammar = Grammar(list(), set(), list(), list(), str())  
  
  
def machine\_input():  
 filename = filedialog.askopenfilename(filetypes=[("Json Files", "\*.json"), ("All Files", "\*.\*")],  
 initialdir=path.dirname(\_\_file\_\_))  
 if not filename:  
 return  
 try:  
 with open(filename, "r") as json\_file:  
 data = json.load(json\_file)  
 except FileNotFoundError:  
 print("Файл с данными не найден.")  
 exit(-1)  
 entry\_alpabet.delete(0, END)  
 entry\_multiplicity.delete(0, END)  
 entry\_start\_chain.delete(0, END)  
 entry\_end\_chain.delete(0, END)  
 entry\_alpabet.insert(0, data["alpabet"])  
 entry\_multiplicity.insert(0, data["multiplicity"])  
 entry\_start\_chain.insert(0, data["start\_chain"])  
 entry\_end\_chain.insert(0, data["end\_chain"])  
  
  
def machine\_output():  
 filename = filedialog.askopenfilename(filetypes=[("Json Files", "\*.json"), ("All Files", "\*.\*")],  
 initialdir=path.dirname(\_\_file\_\_))  
 if not filename:  
 return  
 data = {}  
 data["alpabet"] = entry\_alpabet.get()  
 data["multiplicity"] = entry\_multiplicity.get()  
 data["start\_chain"] = entry\_start\_chain.get()  
 data["end\_chain"] = entry\_end\_chain.get()  
 try:  
 with open(filename, "w") as json\_file:  
 json.dump(data, json\_file)  
 except FileNotFoundError:  
 print("Файл с данными не найден.")  
 exit(-1)  
  
  
# Отрисовывает таблицу переходов  
def generate\_func\_tab(frame):  
 lbl\_sigma = Label(frame, text=f"P:", font=("Arial", 15), pady=5)  
 lbl\_sigma.grid(row=1, column=0, sticky="w", padx=5)  
 i = 2  
 for rule in grammar.Rules:  
 lbl\_alphabet = Label(frame, text=f"'{rule.uniq\_sym}': →", font=("Arial", 15), padx=5, pady=5)  
 lbl\_alphabet.grid(row=i, column=0, padx=15)  
 for j in range(len(rule.Rules)):  
 if j != len(rule.Rules) - 1:  
 lbl\_current = Label(frame, text=f" {rule.Rules[j]} |", font=("Arial", 15), padx=5, pady=5)  
 else:  
 lbl\_current = Label(frame, text=f" {rule.Rules[j]}", font=("Arial", 15), padx=5, pady=5)  
 lbl\_current.grid(row=i, column=1 + j)  
 i += 1  
 for rule in grammar.Extra\_Rules:  
 lbl\_alphabet = Label(frame, text=f"'{rule.uniq\_sym}': →", font=("Arial", 15), padx=5, pady=5)  
 lbl\_alphabet.grid(row=i, column=0)  
 for j in range(len(rule.Rules)):  
 if j != len(rule.Rules) - 1:  
 lbl\_current = Label(frame, text=f" {rule.Rules[j]} |", font=("Arial", 15), padx=5, pady=5)  
 else:  
 lbl\_current = Label(frame, text=f" {rule.Rules[j]}", font=("Arial", 15), padx=5, pady=5)  
 lbl\_current.grid(row=i, column=1 + j)  
 i += 1  
  
  
def count\_non\_term\_sym(gram, sequence):  
 length = 0  
 for sym in sequence:  
 if sym in gram.VT:  
 length += 1  
 return length  
  
  
def generate\_chain\_button():  
 left\_border = int(entry\_left\_border.get())  
 right\_border = int(entry\_right\_border.get())  
 text.delete('1.0', END)  
  
 rules = list(grammar.Start\_state)  
 used\_sequence = set()  
 while rules:  
 sequence = rules.pop()  
 if sequence in used\_sequence:  
 continue  
 used\_sequence.add(sequence)  
 no\_term = True  
 for i, symbol in enumerate(sequence):  
 # print("symbol: " + symbol)  
 if symbol in grammar.VN or symbol == "λ":  
 no\_term = False  
 for elem in normilize\_grammar[symbol]:  
 temp = sequence[:i] + elem + sequence[i + 1:]  
 # print(len(temp), right\_border+1)  
 if count\_non\_term\_sym(grammar, temp) <= right\_border and temp not in rules:  
 rules.append(temp)  
 elif symbol not in grammar.VT:  
 no\_term = False  
 print("цепочка " + sequence + " не разрешима")  
 break  
 # for elem in rules:  
 # print("rule: " + elem)  
 # print("\*\*\*\*\*\*")  
 if no\_term and left\_border <= len(sequence) <= right\_border:  
 text.insert(END, f"Цепочка: {sequence if sequence else 'λ'}\n")  
 print(sequence if sequence else "лямбда")  
  
  
def generate\_grammar\_clicked():  
 lbl\_err.grid\_remove()  
 lbl\_grammar.grid\_remove()  
 normilize\_grammar.clear()  
 text.delete('1.0', END)  
 for widget in frame.winfo\_children():  
 widget.destroy()  
  
 alpabet\_parse = entry\_alpabet.get()  
 multiplicity\_parse = entry\_multiplicity.get()  
 start\_chain\_parse = entry\_start\_chain.get()  
 end\_chain\_parse = entry\_end\_chain.get()  
  
 alpabet = alpabet\_parse.split()  
 multiplicity\_split = re.findall("\d+", multiplicity\_parse)  
  
 error\_string = str()  
 if not alpabet:  
 error\_string = "Отсутствует алфавит"  
 elif not multiplicity\_split:  
 error\_string = "Отсутствует кратность"  
 elif any(i not in alpabet for i in list(start\_chain\_parse)):  
 error\_string = "В начальной цепочке содержатся символы, отсутствующие в алфавите"  
 elif any(i not in alpabet for i in list(end\_chain\_parse)):  
 error\_string = "В конечной цепочке содержатся символы, отсутствующие в алфавите"  
  
 if error\_string:  
 lbl\_err.config(text=error\_string)  
 lbl\_err.grid(row=8, column=0, sticky="w", padx=5, pady=10)  
 return  
  
 print(start\_chain\_parse, end\_chain\_parse)  
 multiplicity = int(multiplicity\_split[0])  
  
 generate\_grammar(alpabet, multiplicity, start\_chain\_parse, end\_chain\_parse)  
 grammar\_text = f"G = (VT={grammar.VT}, VN={grammar.VN}, P, {grammar.Start\_state})"  
 lbl\_grammar.config(text=grammar\_text)  
 lbl\_grammar.grid(row=8, column=0, columnspan=2, sticky="w")  
  
 generate\_func\_tab(frame)  
 frame.grid(row=9, column=0, sticky="w")  
  
 print(normilize\_grammar)  
  
 lbl\_left\_border = Label(window, text=f"От: ", font=("Arial", 12))  
 lbl\_left\_border.grid(row=0, column=2, sticky="w", padx=20)  
 entry\_left\_border.grid(row=0, column=2, sticky="w", padx=55)  
 entry\_left\_border.delete(0, END)  
 entry\_left\_border.insert(0, "0")  
 lbl\_right\_border = Label(window, text=f"До: ", font=("Arial", 12))  
 lbl\_right\_border.grid(row=0, column=2, sticky="w", padx=140)  
 entry\_right\_border.grid(row=0, column=2, sticky="w", padx=175)  
 entry\_right\_border.delete(0, END)  
 entry\_right\_border.insert(0, start\_chain\_len + end\_chain\_len + multiplicity)  
  
 btn\_generate\_chain = Button(window, text="Сгенерировать все\nцепочки языка", command=partial(generate\_chain\_button),  
 padx=10, pady=5)  
 btn\_generate\_chain.grid(row=0, column=2, sticky="e", padx=10, pady=10)  
  
 text.grid(row=2, column=2, rowspan=8, sticky="N" + "S", padx=18, pady=10)  
 scroll = Scrollbar(command=text.yview)  
 scroll.grid(row=2, column=2, rowspan=8, sticky="N" + "S" + "E", pady=10)  
 text.config(yscrollcommand=scroll.set)  
  
  
# находит в списке самое короткое объединение начальной и конечной цепочки, соответствующее кратности  
# если такой нет, то возвращает объединёённые цепочки без схлопывания  
def find\_effective\_chain(max\_union\_chain\_list, multiplicity, start\_chain\_parse, end\_chain\_parse):  
 if len(max\_union\_chain\_list) != 0:  
 for chain in max\_union\_chain\_list:  
 # print(chain)  
 if len(chain) % multiplicity == 0:  
 return chain  
 return start\_chain\_parse + end\_chain\_parse  
  
  
def generate\_grammar(alpabet, multiplicity, start\_chain\_parse, end\_chain\_parse):  
 global min\_chain, start\_chain\_len, end\_chain\_len  
 min\_chain = 0  
 Unique\_sym\_counter = 0  
 General\_rules\_counter = 0  
 grammar.Rules.clear()  
 grammar.Extra\_Rules.clear()  
 grammar.VT.clear()  
 grammar.VN.clear()  
 start\_state = "A"  
  
 # если выбрана лево-линейная цепочка  
 if r\_var.get() != 1:  
 temp\_reverse = start\_chain\_parse[::-1]  
 start\_chain\_parse = end\_chain\_parse[::-1]  
 end\_chain\_parse = temp\_reverse  
  
 start\_chain = list(start\_chain\_parse)  
 end\_chain = list(end\_chain\_parse)  
 print(alpabet, multiplicity, start\_chain, end\_chain)  
  
 same\_start\_end = False  
 grammar.VT = alpabet  
  
 start\_chain\_len = len(start\_chain\_parse)  
 end\_chain\_len = len(end\_chain\_parse)  
 chains\_min\_len = start\_chain\_len if start\_chain\_len <= end\_chain\_len else end\_chain\_len  
 start\_len\_dif = start\_chain\_len - chains\_min\_len  
 end\_len\_dif = end\_chain\_len - chains\_min\_len  
  
 max\_union\_chain\_list = list()  
  
 print()  
 print(f"начальная цепочка: {start\_chain} - Длина: {start\_chain\_len}")  
 print(f"начальная цепочка: {end\_chain} - Длина: {end\_chain\_len}")  
 print(start\_len\_dif, end\_len\_dif)  
  
 # если у нас пустая начальная или конечная цепочка  
 if start\_chain\_len == 0 and end\_chain\_len == 0:  
 # генерация доп правил  
 for i in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 if i == 0:  
 new\_Extra\_rule.Rules += ["λ"]  
 for v in grammar.VT:  
 if i == multiplicity - 1:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[0]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[i + 1]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # если не указан только начальная цепочка  
 elif start\_chain\_len == 0:  
 rules\_to\_add = (multiplicity - (end\_chain\_len % multiplicity)) % multiplicity  
 print(f"Правил не хватает до кратности: {rules\_to\_add}")  
 rules\_to\_add += 1  
  
 same\_sym\_counter = 1  
 cycle\_last\_rule = False  
 counter\_block = False  
 for i in range(1, end\_chain\_len):  
 if end\_chain[i] == end\_chain[0] and not counter\_block:  
 same\_sym\_counter += 1  
 else:  
 counter\_block = True  
 if same\_sym\_counter == end\_chain\_len and multiplicity == 1:  
 cycle\_last\_rule = True  
 print("Конечная цепочка состоит из одного повторяющегося символа")  
 else:  
 print(f"В конечной цепочке идёт {same\_sym\_counter} первых символа подряд")  
  
 # генерация конечных правил  
 for i in range(1, end\_chain\_len + 1):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = (rules\_to\_add + General\_rules\_counter) % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[i]  
 if i == end\_chain\_len:  
 new\_rule.Rules += ["λ"]  
 else:  
 new\_rule.Rules += [end\_chain[i] + string.ascii\_uppercase[i]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # генерация дополнительных правил  
 for i in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = i % multiplicity  
 for v in grammar.VT:  
 if (new\_Extra\_rule.multiplicity\_count + 1) % multiplicity == grammar.Rules[  
 0].multiplicity\_count and v == end\_chain[0]:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[0]]  
 elif i == multiplicity - 1:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[General\_rules\_counter]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
 Unique\_sym\_counter += 1  
  
 # добавляем ссылки на доп правила из конечной цепочки если алфавит состоит из больше чем 1го символа  
 if len(grammar.VT) != 1:  
 for i in range(end\_chain\_len):  
 for v in grammar.VT:  
 if i == end\_chain\_len - 1:  
 if cycle\_last\_rule and v == end\_chain[0]:  
 grammar.Rules[i].Rules += [v + grammar.Rules[i].uniq\_sym]  
 else:  
 grammar.Rules[i].Rules += [v + grammar.Extra\_Rules[0].uniq\_sym]  
 else:  
 grammar.Rules[i].Rules += [  
 v + grammar.Extra\_Rules[(grammar.Rules[i].multiplicity\_count + 1) % multiplicity].uniq\_sym]  
 # иначе если кратность равна 1 - циклим последнее правило  
 elif multiplicity == 1:  
 grammar.Rules[end\_chain\_len - 1].Rules += [end\_chain[0] + grammar.Rules[end\_chain\_len - 1].uniq\_sym]  
 elif end\_chain\_len > multiplicity:  
 grammar.Rules[end\_chain\_len - 1].Rules += [  
 end\_chain[0] + grammar.Rules[end\_chain\_len % multiplicity].uniq\_sym]  
 else:  
 grammar.Rules[end\_chain\_len - 1].Rules += [end\_chain[0] + grammar.Extra\_Rules[1 % multiplicity].uniq\_sym]  
  
 # изменяем начальное состояние на дополнительное правило для сохранения кратностт  
 start\_state = grammar.Extra\_Rules[0].uniq\_sym  
  
 # если не указан только конечная цепочка  
 elif end\_chain\_len == 0:  
  
 # генерация начальных правил  
 for i in range(start\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[i + 1]  
 new\_rule.Rules += [start\_chain\_parse[i] + string.ascii\_uppercase[i + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # генерация дополнительных правил  
 for i in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = (General\_rules\_counter + i) % multiplicity  
 if new\_Extra\_rule.multiplicity\_count == 0:  
 new\_Extra\_rule.Rules += ["λ"]  
 for v in grammar.VT:  
 if i == multiplicity - 1:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[General\_rules\_counter]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
 Unique\_sym\_counter += 1  
  
 # иначе если начальная и конечная заполнены  
 else:  
 # находит смежное количество символов междду начальной и конечной цепочкой  
 for i in range(chains\_min\_len):  
 print(i + start\_len\_dif, chains\_min\_len - i, " : ", start\_chain\_parse[start\_len\_dif + i:],  
 end\_chain\_parse[:chains\_min\_len - i])  
 if start\_chain\_parse[start\_len\_dif + i:] == end\_chain\_parse[:chains\_min\_len - i]:  
 collective\_sym\_count = chains\_min\_len - i  
 print(f"Общее количество символов у двух подцепочек: {collective\_sym\_count}")  
 max\_union\_chain = start\_chain\_parse[:start\_chain\_len - collective\_sym\_count] + end\_chain\_parse  
 print(f"Объединённая цепочка: {max\_union\_chain}")  
 max\_union\_chain\_list.append(max\_union\_chain)  
 # break  
 print(f"Все возможные сочетания начальной и конечной цепочек: {max\_union\_chain\_list}")  
  
 # подбираем самое эффективное сочетание начальной и конечной цепочки:  
 # > наименьшего размера  
 # > соответствует кратности  
 # если такой не нашлось, то цепочки просто складываются друг за другом  
 max\_union\_chain = find\_effective\_chain(max\_union\_chain\_list, multiplicity, start\_chain\_parse, end\_chain\_parse)  
 print(f"Самое эффективное сочетание цепочки: {max\_union\_chain}")  
  
 # генерация общих начальных правил  
 for i in range(start\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.Rules.append(start\_chain\_parse[i] + string.ascii\_uppercase[i + 1])  
 new\_rule.next\_rule = string.ascii\_uppercase[i + 1]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 rules\_count\_to\_add = (multiplicity - (len(max\_union\_chain) % multiplicity)) % multiplicity  
 print(f"\nДобавить правил: {rules\_count\_to\_add}")  
  
 # если полное счетание начальной и конечной цепочек меньше кратности  
 if start\_chain\_len + end\_chain\_len < multiplicity:  
  
 Extra\_rules\_counter = 0  
 start\_General\_rules\_counter = General\_rules\_counter % multiplicity  
 for i in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = (start\_General\_rules\_counter + i) % multiplicity  
  
 # если это не последнее доп правило то добавляем ссылку на следующее доп правило по всем символам  
 if i != multiplicity - 1:  
 # если это доп правило, которое по ключевому символу переходит в конечную цепочку  
 for v in grammar.VT:  
 # если это ключевой символ для перехода в конечную цепочку  
 if i == rules\_count\_to\_add % multiplicity and v == end\_chain\_parse[0]:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[start\_chain\_len + multiplicity]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 # иначе добавляем ссылки по всем символам на первое доп правило  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter - Extra\_rules\_counter]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
 Unique\_sym\_counter += 1  
 Extra\_rules\_counter += 1  
 # если доп правило является частью конечной цепочки для завершения кратности  
 if i <= rules\_count\_to\_add:  
 General\_rules\_counter += 1  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 for v in grammar.VT:  
 if v == end\_chain[i]:  
 new\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 for j in grammar.Extra\_Rules:  
 if (new\_rule.multiplicity\_count + 1) % multiplicity == j.multiplicity\_count:  
 needed\_sym = j.uniq\_sym  
 new\_rule.Rules += [v + needed\_sym]  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # финальное правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.Rules += ["λ"]  
 for j in grammar.Extra\_Rules:  
 if (new\_rule.multiplicity\_count + 1) % multiplicity == j.multiplicity\_count:  
 needed\_sym = j.uniq\_sym  
 for v in grammar.VT:  
 new\_rule.Rules += [v + needed\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # если сочетание начальной и конечной цепочек не соответствует кратности  
 # например:  
 # начальная: 123  
 # конечная: 356  
 # кратность: 4  
 # то добавляем правила закрывающие начальную цепочку  
 # и генерируем доп правила, закрывающие конечную цепочку и прокручивающие символы для кратности  
 elif rules\_count\_to\_add != 0:  
 rules\_count\_to\_add\_left = (multiplicity - (start\_chain\_len % multiplicity)) % multiplicity  
 rules\_count\_to\_add\_right = (multiplicity - (end\_chain\_len % multiplicity)) % multiplicity  
 # если случится что начальную цепочку по кратности дополняет конечная цепочка  
 # то делаем смещение и обнуляем количество доп правил для конечной цепочки  
 # иначе кратность будут дважды соблюдаться  
 rules\_count\_to\_add\_if\_collision = 0  
 if start\_chain\_len > multiplicity or end\_chain\_len > multiplicity:  
 rules\_count\_to\_add\_right = 0  
 rules\_count\_to\_add\_if\_collision = end\_chain\_len % multiplicity  
  
 print(f"Добавить правил слева: {rules\_count\_to\_add\_left - rules\_count\_to\_add\_if\_collision}")  
 print(f"Добавить правил справа: {rules\_count\_to\_add\_right}")  
 print(f"Количество правил из конечной цепочки, завершающие начальную: {rules\_count\_to\_add\_if\_collision}")  
 print(f"Необходимо добавление правил")  
  
 Extra\_rules\_counter = 0  
 # если у конечной цепочки нужно добавить правила для сохранения кратности  
 # то доп правила, которые крутят цепочку до бесконечности будут генерироваться там  
 if rules\_count\_to\_add\_right != 0:  
  
 # добавляем правила для окончания начальной цепочки  
 for i in range(rules\_count\_to\_add\_left):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = False  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 for j in grammar.VT:  
 new\_rule.Rules += [j + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # генерируем дополнительные правила, сохраняющие кратность конечной цепочки и раскручивающие бесконечную генерацию  
 # количество доп. правил будет равно кратности  
 # последнее доп правило будет ссылаться:  
 # по ключевому символу: на первое правило для конечной цепочки  
 # по второстепенному: на первое дополнительное правило  
 for j in range(multiplicity + 1):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = j  
  
 # если это не последнее доп правило то добавляем ссылку на следующее доп правило по всем символам  
 if j != multiplicity:  
 # если это доп правило, которое по ключевому символу переходит в конечную цепочку  
 if j == rules\_count\_to\_add\_right:  
 for v in grammar.VT:  
 # если это ключевой символ для перехода в конечную цепочку  
 if v == end\_chain\_parse[0]:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[  
 Unique\_sym\_counter + (multiplicity - rules\_count\_to\_add\_right) + 1]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 # иначе добавляем ссылки по всем символам на первое доп правило  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [  
 v + string.ascii\_uppercase[Unique\_sym\_counter - Extra\_rules\_counter + 1]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
  
 # если доп правило является частью конечной цепочки для завершения кратности  
 if j <= rules\_count\_to\_add\_right:  
 General\_rules\_counter += 1  
 Unique\_sym\_counter += 1  
 Extra\_rules\_counter += 1  
  
 # генерируем правила конечной цепочки и из каждого правила ссылаемся на доп правила по свободным символам  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 for j in grammar.VT:  
 if j != end\_chain\_parse[i]:  
 new\_rule.Rules += [j + grammar.Extra\_Rules[new\_rule.multiplicity\_count + 1].uniq\_sym]  
 else:  
 new\_rule.Rules += [j + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # добавляем завершающее правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.Rules += ["λ"]  
 for j in grammar.VT:  
 new\_rule.Rules += [j + grammar.Extra\_Rules[new\_rule.multiplicity\_count + 1].uniq\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
 else:  
 # генерируем дополнительные правила, сохраняющие кратность начальной цепочки и раскручивающие бесконечную генерацию  
 # количество доп. правил будет равно кратности  
 # последнее доп правило будет ссылаться:  
 # по ключевому символу: на первое правило для конечной цепочки  
 # по второстепенному: на первое дополнительное правило  
 start\_General\_rules\_counter = General\_rules\_counter % multiplicity  
 for j in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = (start\_General\_rules\_counter + j) % multiplicity  
  
 # если это не последнее доп правило то добавляем ссылку на следующее доп правило по всем символам  
 if j != multiplicity - 1:  
 # если это доп правило, которое по ключевому символу переходит в конечную цепочку  
 if j + rules\_count\_to\_add\_if\_collision == rules\_count\_to\_add\_left:  
 for v in grammar.VT:  
 # если это ключевой символ для перехода в конечную цепочку  
 if v == end\_chain\_parse[0]:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + (  
 multiplicity - rules\_count\_to\_add\_left + rules\_count\_to\_add\_if\_collision)]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 # иначе добавляем ссылки по всем символам на первое доп правило  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [  
 v + string.ascii\_uppercase[Unique\_sym\_counter - Extra\_rules\_counter]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
  
 # если доп правило является частью конечной цепочки для завершения кратности  
 if j < rules\_count\_to\_add\_left:  
 General\_rules\_counter += 1  
 # print(General\_rules\_counter)  
  
 Unique\_sym\_counter += 1  
 Extra\_rules\_counter += 1  
  
 print(f"start {start\_General\_rules\_counter}, current {General\_rules\_counter}")  
  
 # генерируем правила конечной цепочки и из каждого правила ссылаемся на доп правила по свободным символам  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = (General\_rules\_counter + 1) % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 for j in grammar.VT:  
 if j != end\_chain\_parse[i]:  
 new\_rule.Rules += [j + grammar.Extra\_Rules[((multiplicity - start\_General\_rules\_counter)  
 + new\_rule.multiplicity\_count - rules\_count\_to\_add\_if\_collision + 1) % multiplicity].uniq\_sym]  
 else:  
 new\_rule.Rules += [j + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 print(f"start {start\_General\_rules\_counter}, current {General\_rules\_counter}")  
 # добавляем завершающее правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = (General\_rules\_counter + 1) % multiplicity  
 new\_rule.Rules += ["λ"]  
 for j in grammar.VT:  
 new\_rule.Rules += [j + grammar.Extra\_Rules[((multiplicity - start\_General\_rules\_counter)  
 + new\_rule.multiplicity\_count - rules\_count\_to\_add\_if\_collision + 1) % multiplicity].uniq\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # иначе, если начальные и конечные цепочки никак не схлопываются и сохраняют кратность  
 elif start\_chain\_len + end\_chain\_len == len(max\_union\_chain):  
 rules\_count\_to\_add\_left = (multiplicity - (start\_chain\_len % multiplicity)) % multiplicity  
 rules\_count\_to\_add\_right = 0  
 rules\_count\_to\_add\_if\_collision = end\_chain\_len % multiplicity  
  
 print(f"Добавить правил слева: {rules\_count\_to\_add\_left - rules\_count\_to\_add\_if\_collision}")  
 print(f"Добавить правил справа: {rules\_count\_to\_add\_right}")  
 print(f"Количество правил из конечной цепочки, завершающие начальную: {rules\_count\_to\_add\_if\_collision}")  
 print(f"Добавление правил не нужно")  
  
 same\_sym\_counter = 1  
 cycle\_last\_rule = False  
 counter\_block = False  
 for i in range(1, end\_chain\_len):  
 if end\_chain[i] == end\_chain[0] and not counter\_block:  
 same\_sym\_counter += 1  
 else:  
 counter\_block = True  
 if same\_sym\_counter == end\_chain\_len and multiplicity == 1:  
 cycle\_last\_rule = True  
 print("Конечная цепочка состоит из одного повторяющегося символа")  
 else:  
 print(f"В конечной цепочке идёт {same\_sym\_counter} первых символа подряд")  
  
 Extra\_rules\_counter = 0  
 # генерируем дополнительные правила, сохраняющие кратность начальной цепочки и раскручивающие бесконечную генерацию  
 # количество доп. правил будет равно кратности  
 # последнее доп правило будет ссылаться:  
 # по ключевому символу: на первое правило для конечной цепочки  
 # по второстепенному: на первое дополнительное правило  
 start\_General\_rules\_counter = General\_rules\_counter % multiplicity  
 for j in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = (start\_General\_rules\_counter + j) % multiplicity  
  
 # если это не последнее доп правило то добавляем ссылку на следующее доп правило по всем символам  
 if j != multiplicity - 1:  
 # если это доп правило, которое по ключевому символу переходит в конечную цепочку  
 if j + rules\_count\_to\_add\_if\_collision == rules\_count\_to\_add\_left:  
 for v in grammar.VT:  
 # если это ключевой символ для перехода в конечную цепочку  
 if v == end\_chain\_parse[0]:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + (  
 multiplicity - rules\_count\_to\_add\_left + rules\_count\_to\_add\_if\_collision)]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 # иначе добавляем ссылки по всем символам на первое доп правило  
 for v in grammar.VT:  
 if multiplicity == 1 and v == end\_chain[0]:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 new\_Extra\_rule.Rules += [  
 v + string.ascii\_uppercase[Unique\_sym\_counter - Extra\_rules\_counter]]  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
  
 # если доп правило является частью конечной цепочки для завершения кратности  
 if j <= rules\_count\_to\_add\_left:  
 General\_rules\_counter += 1  
  
 Unique\_sym\_counter += 1  
 Extra\_rules\_counter += 1  
  
 # генерируем правила конечной цепочки и из каждого правила ссылаемся на доп правила по свободным символам  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = (General\_rules\_counter + 1) % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 print(new\_rule.uniq\_sym)  
 for v in grammar.Extra\_Rules:  
 if (new\_rule.multiplicity\_count + 1) % multiplicity == v.multiplicity\_count:  
 needed\_sym = v.uniq\_sym  
 for v in grammar.VT:  
 if v != end\_chain[i]:  
 if i == same\_sym\_counter and v == end\_chain[0] and multiplicity == 1:  
 new\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter]]  
 else:  
 new\_rule.Rules += [v + needed\_sym]  
 else:  
 new\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # добавляем завершающее правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = (General\_rules\_counter + 1) % multiplicity  
 new\_rule.Rules += ["λ"]  
 for v in grammar.Extra\_Rules:  
 if (new\_rule.multiplicity\_count + 1) % multiplicity == v.multiplicity\_count:  
 needed\_sym = v.uniq\_sym  
 for v in grammar.VT:  
 if cycle\_last\_rule and v == end\_chain[0]:  
 new\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter]]  
 else:  
 new\_rule.Rules += [v + needed\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # Иначе если цепочки полностью схлопнулись с сохранением кратности  
 elif start\_chain\_parse == max\_union\_chain and end\_chain\_parse == max\_union\_chain:  
 print("\nЦепочки схлопнулись")  
  
 same\_sym\_counter = 1  
 counter\_block = False  
 add\_exit\_to\_all\_rules = False  
 for i in range(1, end\_chain\_len):  
 if end\_chain[i] == end\_chain[0] and not counter\_block:  
 same\_sym\_counter += 1  
 else:  
 counter\_block = True  
 if same\_sym\_counter == end\_chain\_len:  
 add\_exit\_to\_all\_rules = True  
 print("Конечная цепочка состоит из одного повторяющегося символа")  
 else:  
 print(f"В конечной цепочке идёт {same\_sym\_counter} первых символа подряд")  
  
 # добавляем правило для перехода в конечную цепочку, либо в доп бесконечную генерацию  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 # т.к. обе подцепочки кратны и полностью схлопываются, то добавляем доп выход  
 new\_rule.Rules += ["λ"]  
 # если цепочка состоит из одного повторяющегося символа, кратность равна 1 и в алфавите нету других символов  
 # то просто оставляем крутиться по конечному правилу  
 if add\_exit\_to\_all\_rules and multiplicity == 1 and len(grammar.VT) == 1:  
 new\_rule.Rules += [end\_chain[0] + string.ascii\_uppercase[Unique\_sym\_counter]]  
 grammar.Rules.append(new\_rule)  
 else:  
 new\_rule.Rules += [end\_chain[0] + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
 # запоминаем позицию этого правила  
 end\_postition\_of\_start = Unique\_sym\_counter - 1  
  
 # генерируем правила конечной цепочки  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 if add\_exit\_to\_all\_rules and i % multiplicity == 0:  
 new\_rule.Rules += ["λ"]  
 new\_rule.Rules += [end\_chain[i] + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # добавляем завершающее правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.Rules += ["λ"]  
 if len(end\_chain) == 1:  
 new\_rule.Rules += [end\_chain[0] + string.ascii\_uppercase[Unique\_sym\_counter]]  
 else:  
 new\_rule.Rules += [end\_chain[0] + grammar.Rules[end\_postition\_of\_start + 1].uniq\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
 # запоминаем позицию финального правила  
 final\_rule\_position = Unique\_sym\_counter - 1  
  
 # если в алфавите больше одного символа, то генерируем хвостову. конечную цепочку и доп правила для раскрутки бексконечности  
 if len(grammar.VT) > 1:  
  
 start\_tail\_index = Unique\_sym\_counter  
 print(f"Начало хвоста конечной цепочки: {start\_tail\_index}")  
 # дополняем хвостовыми правилами для конечной цепочки  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = i % multiplicity  
 if i != end\_chain\_len - 1:  
 new\_rule.Rules += [end\_chain[i] + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 else:  
 new\_rule.Rules += [end\_chain[i] + string.ascii\_uppercase[final\_rule\_position]]  
 new\_rule.next\_rule = string.ascii\_uppercase[final\_rule\_position]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
 end\_tail\_index = Unique\_sym\_counter  
 print(f"Конец хвоста конечной цепочки: {end\_tail\_index}")  
  
 # если хвоста у конечной цепоки нет - то исправляем индексы  
 if end\_chain\_len == 1:  
 start\_tail\_index -= 1  
 end\_tail\_index -= 1  
  
 Extra\_rules\_counter = 0  
 for i in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = i % multiplicity  
  
 # если это первое доп правило, то ссылаемся по ключевому символу на первое правило конечной цепочки,  
 # а по остальным - на другие доп правила  
 if (new\_Extra\_rule.multiplicity\_count + 1) % multiplicity == grammar.Rules[  
 start\_tail\_index].multiplicity\_count:  
 for v in grammar.VT:  
 if v != end\_chain\_parse[0]:  
 # если кратность 1, то по по второстепенным символам ссылаемся на себя же  
 if multiplicity == 1:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter]]  
 # иначе на следущее доп правило  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 new\_Extra\_rule.Rules += [v + grammar.Rules[start\_tail\_index].uniq\_sym]  
 # если это не последнее доп правило то добавляем ссылку на следующее доп правило по всем символам  
 elif i != multiplicity - 1:  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 # иначе ссылаемся по всем символам на первое доп правило  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + grammar.Extra\_Rules[0].uniq\_sym]  
  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
 Unique\_sym\_counter += 1  
 Extra\_rules\_counter += 1  
  
 # добаляем ссылку для правил конечной цепочки  
 for i in range(end\_postition\_of\_start, final\_rule\_position):  
 for v in grammar.VT:  
 # если символ, повторяющийся в начале конечной цепочки  
 if v == end\_chain[0] and i == end\_postition\_of\_start + same\_sym\_counter:  
 # если повторений первого символа в конечной цепочке меньше чем кратность (т.е. кратность не равна 1)  
 # то ссылаемся на доп правила  
 if same\_sym\_counter < multiplicity:  
 grammar.Rules[i].Rules += [v + grammar.Extra\_Rules[  
 (grammar.Rules[i].multiplicity\_count + 1) % multiplicity].uniq\_sym]  
 # иначе в зависимости от кратности ссылаем на предыдущее правило, которое сохраняет нашу кратность  
 else:  
 grammar.Rules[i].Rules += [v + grammar.Rules[i - multiplicity + 1].uniq\_sym]  
 elif v != end\_chain[i - end\_postition\_of\_start]:  
 grammar.Rules[i].Rules += [v + grammar.Extra\_Rules[  
 (grammar.Rules[i].multiplicity\_count + 1) % multiplicity].uniq\_sym]  
  
 # добаляем ссылку для хвостовых правил конечной цепочки  
 for i in range(start\_tail\_index, end\_tail\_index):  
 for v in grammar.VT:  
 # если символ, повторяющийся в начале конечной цепочки  
 if v == end\_chain[0] and i == start\_tail\_index + same\_sym\_counter - 1:  
 # если повторений первого символа в конечной цепочке меньше чем кратность (т.е. кратность не равна 1)  
 # то ссылаемся на доп правила  
 if same\_sym\_counter - 1 < multiplicity:  
 grammar.Rules[i].Rules += [v + grammar.Extra\_Rules[  
 (grammar.Rules[i].multiplicity\_count + 1) % multiplicity].uniq\_sym]  
 # иначе в зависимости от кратности ссылаем на предыдущее правило, которое сохраняет нашу кратность  
 else:  
 grammar.Rules[i].Rules += [v + grammar.Rules[i - multiplicity + 1].uniq\_sym]  
 elif v != end\_chain[i - start\_tail\_index]:  
 grammar.Rules[i].Rules += [v + grammar.Extra\_Rules[  
 (grammar.Rules[i].multiplicity\_count + 1) % multiplicity].uniq\_sym]  
  
 # иначе если цепочки схлопнулись до нужной кратности  
 else:  
 print("\nЦепочки схлопнулись частично")  
  
 start\_collective\_node\_position = len(max\_union\_chain) - end\_chain\_len  
 print(f"Позиция первого общего правила у цепочек {start\_collective\_node\_position}")  
 end\_collective\_node\_position = Unique\_sym\_counter  
 print(f"Позиция последнего общего правила у цепочек {end\_collective\_node\_position}")  
  
 # если конечная цепочка полностью сливается с начальной  
 # например  
 # начальная: 1234  
 # конечная: 234  
 # кратность: 4  
 # то уменьшаем позицию последнего коллективного правила  
 full\_end\_in\_start = False  
 if end\_collective\_node\_position - start\_collective\_node\_position == end\_chain\_len:  
 end\_collective\_node\_position -= 1  
 full\_end\_in\_start = True  
  
 # догенерируем правила для частично-схлопнутой  
 for i in range(start\_chain\_len, len(max\_union\_chain)):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 new\_rule.Rules += [max\_union\_chain[i] + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 final\_position = Unique\_sym\_counter  
 print(f"Позиция финального правила: {final\_position}")  
  
 # финальное правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.Rules += ["λ"]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 start\_tail\_index = Unique\_sym\_counter  
 print(f"Начало хвоста конечной цепочки: {start\_tail\_index}")  
 # дополняем хвостовыми правилами для конечной цепочки  
 print(start\_collective\_node\_position, end\_collective\_node\_position)  
 for i in range(start\_collective\_node\_position, end\_collective\_node\_position):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = (i + 1) % multiplicity  
 if i != end\_collective\_node\_position - 1:  
 new\_rule.Rules += [max\_union\_chain[i + 1] + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 else:  
 new\_rule.Rules += [  
 max\_union\_chain[i + 1] + string.ascii\_uppercase[end\_collective\_node\_position + 1]]  
 new\_rule.next\_rule = string.ascii\_uppercase[end\_collective\_node\_position + 1]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
 end\_tail\_index = Unique\_sym\_counter  
 print(f"Конец хвоста конечной цепочки: {end\_tail\_index}")  
  
 # если хвоста у конечной цепоки нет - то исправляем индексы  
 if end\_chain\_len == 1:  
 start\_tail\_index -= 1  
 end\_tail\_index -= 1  
  
 Extra\_rules\_counter = 0  
 # генерируем дополнительные правила, сохраняющие кратность начальной цепочки и раскручивающие бесконечную генерацию  
 # количество доп. правил будет равно кратности  
 # последнее доп правило будет ссылаться:  
 # по ключевому символу: на первое правило для конечной цепочки  
 # по второстепенному: на первое дополнительное правило  
 for j in range(multiplicity):  
 new\_Extra\_rule = ExtraRule(str(), int(), list())  
 new\_Extra\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_Extra\_rule.key\_rule = False  
 new\_Extra\_rule.multiplicity\_count = (grammar.Rules[  
 end\_collective\_node\_position].multiplicity\_count + j + 1) % multiplicity  
  
 # если это первое общее правило, то ссылаемся по ключевому символу на первое правило конечной цепочки, а по остальным - на следующее доп правило  
 if (new\_Extra\_rule.multiplicity\_count + 1) % multiplicity == grammar.Rules[  
 start\_tail\_index].multiplicity\_count:  
 for v in grammar.VT:  
 if v != end\_chain\_parse[0]:  
 if multiplicity == 1:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 new\_Extra\_rule.Rules += [v + grammar.Rules[start\_tail\_index].uniq\_sym]  
 # если это не последнее доп правило то добавляем ссылку на следующее доп правило по всем символам  
 elif j != multiplicity - 1:  
 for v in grammar.VT:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 else:  
 # иначе ссылаемся если конечная цепочка не полностью поглощается начальной -  
 # то по ключевому символу ссылаемся на первое правило конечной цепочки,  
 # а по свободным символам - на первое доп правило  
 for v in grammar.VT:  
 if v != end\_chain\_parse[0] or full\_end\_in\_start:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter - j]]  
 else:  
 new\_Extra\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
  
 grammar.Extra\_Rules.append(new\_Extra\_rule)  
 Unique\_sym\_counter += 1  
 Extra\_rules\_counter += 1  
  
 # если конечная цепочка не полностью поглащается начальной, то  
 # добавляем ссылки на доп правила из правил конечной цепочки  
 if not full\_end\_in\_start:  
 for i in range(end\_collective\_node\_position, len(max\_union\_chain)):  
 print(grammar.Rules[i].uniq\_sym)  
 for j in grammar.Extra\_Rules:  
 if (grammar.Rules[i].multiplicity\_count + 1) % multiplicity == j.multiplicity\_count:  
 needed\_sym = j.uniq\_sym  
 for v in grammar.VT:  
 if v != max\_union\_chain[i]:  
 if multiplicity == 1 and v == end\_chain[0]:  
 grammar.Rules[i].Rules += [end\_chain[0] + grammar.Rules[start\_tail\_index].uniq\_sym]  
 else:  
 grammar.Rules[i].Rules += [v + needed\_sym]  
  
 # добавление ссылки на доп правила для хвоста конечных правил, если хвост вообще есть  
 if end\_chain\_len != 1:  
 rule\_index = 1  
 for i in range(start\_tail\_index, end\_tail\_index):  
 for j in grammar.Extra\_Rules:  
 if (grammar.Rules[i].multiplicity\_count + 1) % multiplicity == j.multiplicity\_count:  
 needed\_sym = j.uniq\_sym  
 for v in grammar.VT:  
 if multiplicity == 1 and i == start\_tail\_index and v == end\_chain[0]:  
 grammar.Rules[i].Rules += [v + grammar.Rules[i].uniq\_sym]  
 elif v != max\_union\_chain[start\_collective\_node\_position + rule\_index]:  
 grammar.Rules[i].Rules += [v + needed\_sym]  
 rule\_index += 1  
  
 # добавляем ссылки на доп правила для конечного правила  
 for j in grammar.Extra\_Rules:  
 if (grammar.Rules[final\_position].multiplicity\_count + 1) % multiplicity == j.multiplicity\_count:  
 needed\_sym = j.uniq\_sym  
 for v in grammar.VT:  
 if multiplicity == 1:  
 if len(alpabet) == 1:  
 grammar.Rules[final\_position].Rules += [v + string.ascii\_uppercase[final\_position]]  
 elif end\_chain[end\_chain\_len - 1] == end\_chain[0] and v == end\_chain[1]:  
 if full\_end\_in\_start:  
 grammar.Rules[final\_position].Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter]]  
 elif collective\_sym\_count >= 2:  
 grammar.Rules[final\_position].Rules += [v + string.ascii\_uppercase[start\_tail\_index + 1]]  
 else:  
 grammar.Rules[final\_position].Rules += [  
 v + string.ascii\_uppercase[end\_collective\_node\_position + 1]]  
 elif v == end\_chain[0]:  
 if full\_end\_in\_start:  
 grammar.Rules[final\_position].Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter]]  
 else:  
 grammar.Rules[final\_position].Rules += [v + string.ascii\_uppercase[start\_tail\_index]]  
 else:  
 grammar.Rules[final\_position].Rules += [v + needed\_sym]  
 else:  
 grammar.Rules[final\_position].Rules += [v + needed\_sym]  
  
 # если кратность равна 1 и вся конечная цепочка является частью начальной, то добавляем ещё правила  
 # чтобы любая сгенерированаяцепочка соответствовала кратности  
 if multiplicity == 1 and full\_end\_in\_start and len(alpabet) != 0:  
 # дополнительные конечные правила  
 for i in range(1, end\_chain\_len):  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = 0  
 new\_rule.next\_rule = string.ascii\_uppercase[Unique\_sym\_counter + 1]  
 for v in grammar.VT:  
 if v == end\_chain\_parse[i]:  
 new\_rule.Rules += [v + string.ascii\_uppercase[Unique\_sym\_counter + 1]]  
 elif v == end\_chain\_parse[0] and i == 1:  
 new\_rule.Rules += [v + grammar.Rules[start\_tail\_index + i - 1].uniq\_sym]  
 else:  
 new\_rule.Rules += [v + grammar.Extra\_Rules[0].uniq\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 # финальное правило  
 new\_rule = Rule(str(), bool(), int(), str(), list())  
 new\_rule.uniq\_sym = string.ascii\_uppercase[Unique\_sym\_counter]  
 new\_rule.key\_rule = True  
 new\_rule.multiplicity\_count = General\_rules\_counter % multiplicity  
 new\_rule.Rules += ["λ"]  
 for j in grammar.Extra\_Rules:  
 if (grammar.Rules[final\_position].multiplicity\_count + 1) % multiplicity == j.multiplicity\_count:  
 needed\_sym = j.uniq\_sym  
 for v in grammar.VT:  
 new\_rule.Rules += [v + grammar.Extra\_Rules[0].uniq\_sym]  
 grammar.Rules.append(new\_rule)  
 Unique\_sym\_counter += 1  
 General\_rules\_counter += 1  
  
 grammar.Start\_state = start\_state  
 for rule in grammar.Rules:  
 grammar.VN.add(rule.uniq\_sym)  
  
 # если выбрана лево-линейная цепочка  
 if r\_var.get() != 1:  
 for i in range(len(rule.Rules)):  
 rule.Rules[i] = rule.Rules[i][::-1]  
  
 normilize\_grammar[rule.uniq\_sym] = rule.Rules  
 print(rule)  
 print()  
 for rule in grammar.Extra\_Rules:  
 grammar.VN.add(rule.uniq\_sym)  
  
 # если выбрана лево-линейная цепочка  
 if r\_var.get() != 1:  
 for i in range(len(rule.Rules)):  
 rule.Rules[i] = rule.Rules[i][::-1]  
  
 normilize\_grammar[rule.uniq\_sym] = rule.Rules  
 print(rule)  
 normilize\_grammar["λ"] = [""]  
 return  
  
  
def author\_clicked():  
 var = messagebox.showinfo("Автор", "Бурдуковский Илья Александрович\nИП-813")  
  
  
def theme\_clicked():  
 var = messagebox.showinfo("Тема",  
 "Написать программу, которая по предложенному описанию языка построит регулярную грамматику, "  
 "задающую этот язык, и позволит сгенерировать с её помощью все цепочки языка в заданном диапазоне длин. "  
 "Предусмотреть возможность поэтапного отображения на экране процесса генерации цепочек. \n"  
 "Варианты задания языка: \n"  
 "(4) Алфавит, начальная и конечная подцепочки и кратность длины всех цепочек языка.")  
  
  
# ζ δ ε  
if \_\_name\_\_ == '\_\_main\_\_':  
 ls = list()  
 ls.append("Z")  
 ls = ls[1:]  
 # window.columnconfigure(1, minsize=400, weight=1)  
 # window.rowconfigure(0, minsize=250, weight=1)  
 window.title("Добро пожаловать на сервер ТЯПофриния")  
 # window.geometry('1000x550')  
 lbl = Label(window, text="Регулярная грамматика:", font=("Arial Bold", 20), padx=10)  
 lbl.grid(row=0, column=0, sticky="nw")  
  
 btn\_author = Button(window, text="Автор", command=author\_clicked, padx=5, pady=5)  
 btn\_author.grid(row=0, column=1, sticky="e", padx=5, pady=5)  
  
 btn\_author = Button(window, text="Тема", command=theme\_clicked, padx=5, pady=5)  
 btn\_author.grid(row=0, column=1, sticky="e", padx=70, pady=5)  
  
 btn\_author = Button(window, text="Загрузить правила", command=machine\_input, padx=5, pady=5)  
 btn\_author.grid(row=0, column=1, sticky="e", padx=300, pady=5)  
  
 btn\_author = Button(window, text="Сохранить правила", command=machine\_output, padx=5, pady=5)  
 btn\_author.grid(row=0, column=1, sticky="e", padx=140, pady=5)  
  
 lbl\_alpabet = Label(window, text=f"Алфавит: ", font=("Arial", 13), padx=15)  
 lbl\_alpabet.grid(row=2, column=0, sticky="w")  
 entry\_alpabet.grid(row=3, column=0, padx=10, pady=5)  
  
 lbl\_multiplicity = Label(window, text=f"Кратность: ", font=("Arial", 13), padx=15)  
 lbl\_multiplicity.grid(row=2, column=1, sticky="w")  
 entry\_multiplicity.grid(row=3, column=1, padx=10, pady=5, sticky="w")  
  
 lbl\_radiobutton = Label(window, text=f"Вид регулярной грамматики: ", font=("Arial", 13), padx=5)  
 lbl\_radiobutton.grid(row=2, column=1, sticky="e")  
 Radiobutton\_LL.grid(row=3, column=1, sticky="e", padx=70)  
 Radiobutton\_PL.grid(row=3, column=1, sticky="e", padx=20)  
  
 lbl\_start\_chain = Label(window, text=f"Начальная цепочка: ", font=("Arial", 13), padx=15)  
 lbl\_start\_chain.grid(row=4, column=0, sticky="w")  
 entry\_start\_chain.grid(row=5, column=0, padx=10, pady=5)  
  
 lbl\_end\_chain = Label(window, text=f"Конечная цепочка: ", font=("Arial", 13), padx=15)  
 lbl\_end\_chain.grid(row=4, column=1, sticky="w")  
 entry\_end\_chain.grid(row=5, column=1, padx=10, pady=5)  
  
 btn\_generate\_grammar = Button(window, text="Сгенерировать регулярную\n грамматику!",  
 command=generate\_grammar\_clicked, padx=10, pady=10)  
 btn\_generate\_grammar.grid(row=7, column=0, sticky="w", padx=10, pady=10)  
  
 window.mainloop()