

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:

Студент 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 пройти 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.

Регулярная грамматика:

Загрузить правила Сохранить правила Тема Автор От: 0 До: 9 Сгенерировать все цепочки языка

Алфавит: 1 2 3 Кратность: 3 Вид регулярной грамматики: ЛЛ ПЛ

Начальная цепочка: 123 Конечная цепочка: 123

Сгенерировать регулярную грамматику!

$G = (V_T = \{'1', '2', '3'\}, V_N = \{'L', 'K', 'F', 'B', 'C', 'E', 'D', 'J', 'H', 'I', 'A', 'G'\}, P, A)$

P:

- 'A': $\rightarrow 1B$
- 'B': $\rightarrow 2C$
- 'C': $\rightarrow 3D$
- 'D': $\rightarrow \lambda \mid 1E \mid 2K \mid 3K$
- 'E': $\rightarrow 2F \mid 1L \mid 3L$
- 'F': $\rightarrow 3G \mid 1J \mid 2J$
- 'G': $\rightarrow \lambda \mid 1E$
- 'H': $\rightarrow 2I \mid 1L \mid 3L$
- 'I': $\rightarrow 3G \mid 1J \mid 3J$
- 'J': $\rightarrow 1H \mid 2K \mid 3K$
- 'K': $\rightarrow 1L \mid 2L \mid 3L$
- 'L': $\rightarrow 1J \mid 2J \mid 3J$

Цепочка: 123333123
Цепочка: 123332123
Цепочка: 123331123
Цепочка: 123323123
Цепочка: 123322123
Цепочка: 123321123
Цепочка: 123313123
Цепочка: 123312123
Цепочка: 123311123
Цепочка: 123233123
Цепочка: 123232123
Цепочка: 123231123
Цепочка: 123223123
Цепочка: 123222123
Цепочка: 123221123
Цепочка: 123213123
Цепочка: 123212123
Цепочка: 123211123
Цепочка: 123133123
Цепочка: 123132123
Цепочка: 123131123
Цепочка: 123113123
Цепочка: 123112123
Цепочка: 123111123
Цепочка: 123122123
Цепочка: 123121123
Цепочка: 123123123
Цепочка: 123123
Цепочка: 123

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 \rightarrow 1B
B \rightarrow 2C
C \rightarrow 3D
D $\rightarrow \lambda \mid 1E$

Затем к созданным правилам добавляются правила конечной подцепочки:

E \rightarrow 2F
F \rightarrow 3G
G $\rightarrow \lambda$

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

$H \rightarrow 2I$

$I \rightarrow 3G$

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

$J \rightarrow 1H \mid 2K \mid 3K$

$K \rightarrow 1L \mid 2L \mid 3L$

$G \rightarrow 1J \mid 2J \mid 3J$

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

$G \rightarrow \lambda \mid 3E$

$H \rightarrow 2I \mid 1L \mid 3L$

$I \rightarrow 3G \mid 1J \mid 3J$

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 определённой 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="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 = True
                        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 '\'}\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.wininfo_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 объединённые 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

```

```

кратности
    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 между 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 сочетание 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
        #     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 =
(string.ascii_uppercase[Unique_sym_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 цепочки 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 второстепенным
                # символам ссылаемся на себя же
                if multiplicity == 1:
                    new_Extra_rule.Rules += [v +
string.ascii_uppercase[Unique_sym_counter]]
                # иначе на следующее 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

# добавляем 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]

        # добавляем 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 цепочки 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 поглощается 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 сгенерированная цепочка соответствовала кратности
        if multiplicity == 1 and full_end_in_start and len(alphabet) != 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()

```