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 14

Выполнил:

Студент гр. ИП-014 \_\_\_\_\_\_\_\_\_\_\_\_\_/Обухов А. И./

ФИО студента

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Проверил:

Старший преподаватель кафедры ПМиК \_\_\_\_\_\_\_\_\_\_\_\_\_/Павлова У. В./

ФИО преподавателя

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2023 г. Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Новосибирск 2023 г.

**Содержание**

**1.** Описание задания……………………………………………………..4 **2.** Описание алгоритма……………………………………………….….5 **3.** Описание функций………………………………………………….…6  
**4.** Результат работы………………………………………………………7  
**5.** Листинг…………………………………………………………………9

**Описание задания**

Написать программу, которая по заданному регулярному выражению построит эквивалентную грамматику (по желанию разработчика – грамматика может быть контекстно-свободной или регулярной). Программа должна сгенерировать по построенной грамматике и регулярному выражению множества всех цепочек в указанном диапазоне длин, проверить их на совпадение. Процесс построения цепочек отображать на экране. Для подтверждения корректности выполняемых действий предусмотреть возможность корректировки любого из построенных множеств пользователем (изменение цепочки, добавление, удаление…). При обнаружении несовпадения в элементах множеств должна выдаваться диагностика различий – где именно несовпадения и в чём они состоят.

**Описание алгоритма**

Пользователь должен ввести регулярное выражение для построения КС грамматики. Поля проходят валидацию – при необрабатываемых входных данных действия не будут выполнены.

Один из алгоритмов генерирует слова на основе заданной контекстно-свободной грамматики (CFG), представленной набором продукций. Алгоритм использует подход динамического программирования для генерации слов путем поэтапного создания слов разной длины. Он возвращает сгенерированные слова по мере выполнения. Метод также обрабатывает пустые символы в грамматике и прекращает генерацию слов, если не произошло никаких изменений в сгенерированных словах в течение определенного количества итераций.

Алгоритм для формирования цепочек по регулярному выражению сначала разбивает регулярное выражение на список подстрок, а затем рекурсивно генерирует все возможные комбинации этих подстрок. Функция generate\_chains рекурсивно генерирует все возможные последовательности слов, используя параметры service\_symbols\_rec и split\_chain\_rec, которые содержат информацию о том, какие подстроки являются необязательными и какие могут повторяться.

Для генерации цепочек в определённом диапазоне необходимо ввести два числа (поля проходят валидацию). Все возможные для алфавита цепочки генерируются в рекурсивном цикле, который проходит по всему алфавиту, учитывая, что их длина должна быть кратна введённому числу.

**Описание функций**

Класс Regex, который представляет собой регулярное выражение. Он принимает строку регулярного выражения в качестве аргумента и строит соответствующую CFG (Контекстно-свободная грамматика) для регулярного выражения. Грамматика состоит из двух типов правил: правила производства и сервисные символы. Правило производства представлено парой последовательностей, где первая последовательность - нетерминальный символ, а вторая - список возможных терминальных символов, которые могут заменить нетерминальный символ. Сервисные символы представляются последовательностями, которые не следуют никакому конкретному формату, и используются для захвата специальных случаев в регулярном выражении, таких как операторы звездочки Kleene (\*) и положительного опережения (+).

Метод generate\_words принимает строку регулярного выражения, минимальную длину слова и максимальную длину слова в качестве аргументов и генерирует все возможные слова, соответствующие регулярному выражению.

Метод split\_reg\_ex принимает список сервисных символов и список цепочек разделов в качестве аргументов и рекурсивно разбивает сервисные символы и цепочки разделов на меньшие подпоследовательности. Это делается для обеспечения того, чтобы сгенерированные слова были действительными согласно регулярному выражению.

CFG - это класс, представляющий контекстно-свободную грамматику. Он имеет следующие атрибуты:

variables: Множество всех переменных в грамматике.

terminals: Множество всех терминалов в грамматике.

productions: Множество всех правил вывода в грамматике.

start\_symbol: Начальный символ грамматики.

Класс CFG предоставляет несколько методов для манипулирования и анализа грамматик. Некоторые из этих методов:

remove\_useless\_symbols(): Удаляет переменные и терминалы, которые не появляются ни в одном правиле вывода.

remove\_epsilon(): Удаляет правила вывода, которые генерируют пустую строку.

eliminate\_unit\_productions(): Удаляет правила вывода, которые имеют только один терминал в теле.

get\_words(max\_length=-1): Генерирует все возможные строки, которые могут быть сгенерированы грамматикой, до заданной длины.

**Результат работы**

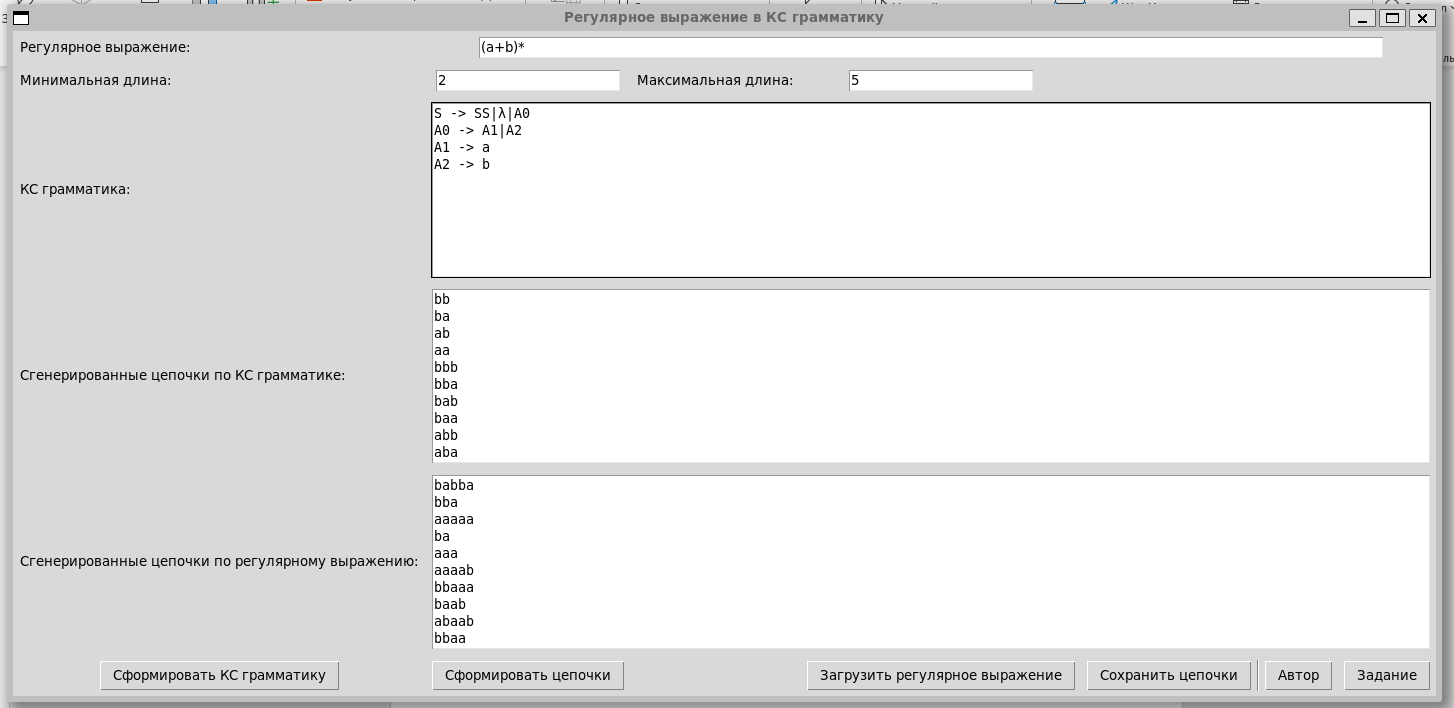


Рис. 1 – Результат работы программы

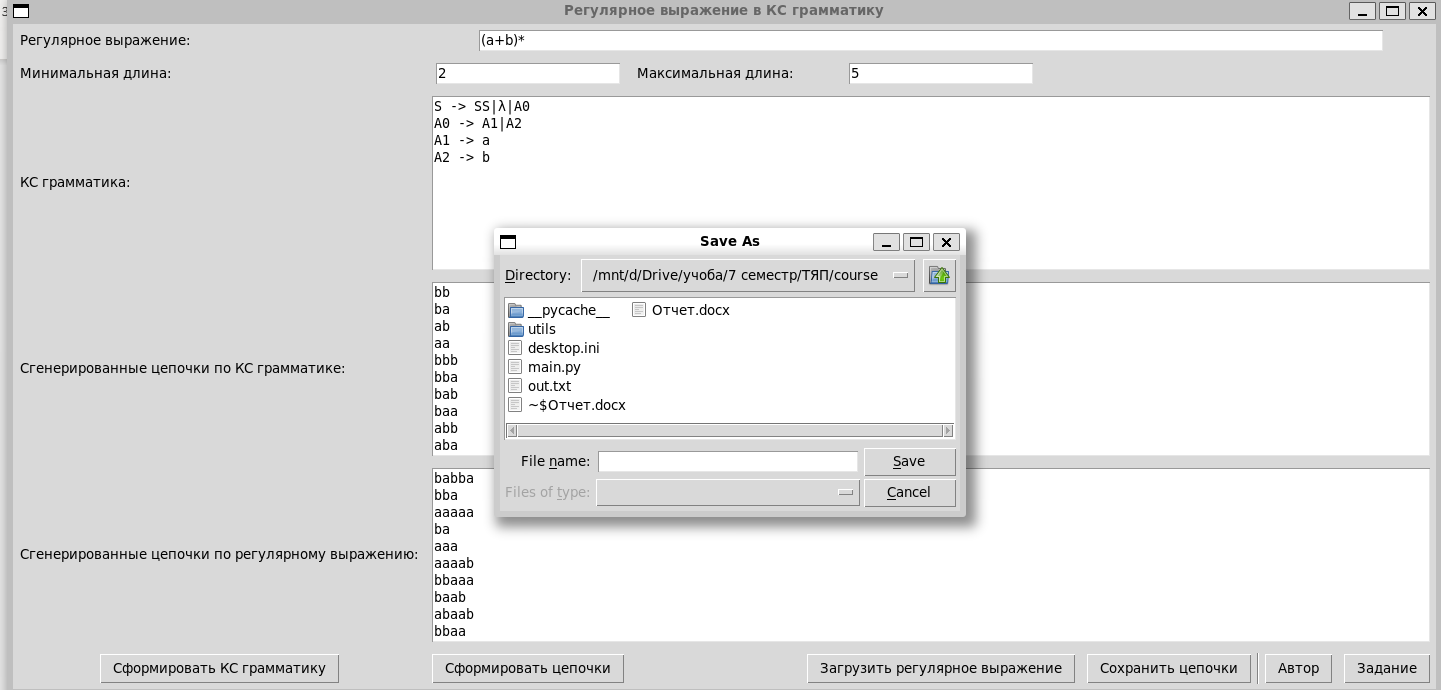
****

Рис. 2 – Сохранение цепочек в файл

**Листинг**

import tkinter as tk

from tkinter import messagebox

from tkinter import filedialog

from tkinter import ttk

from utils.regular\_expression import Regex

import tkinter as tk

from tkinter import messagebox

from utils.regular\_expression import Regex

class RegexGeneratorGUI:

    def \_\_init\_\_(self):

        self.cfg = None

        self.regex = None

        self.root = tk.Tk()

        self.root.title('Регулярное выражение в КС грамматику')

        self.root.resizable(False, False)

        # Create labels for entries

        self.regex\_label = tk.Label(self.root, text="Регулярное выражение:")

        self.min\_length\_label = tk.Label(self.root, text="Минимальная длина:")

        self.max\_length\_label = tk.Label(self.root, text="Максимальная длина:")

        # Create the input fields for regex and word lengths

        self.regex\_entry = tk.Entry(self.root, width=100)

        self.min\_length\_entry = tk.Entry(self.root)

        self.max\_length\_entry = tk.Entry(self.root)

        # Create separate text fields for cfg and generated words

        self.cfg\_text = tk.Text(self.root, width=50, height=10)

        self.generated\_words\_text = tk.Text(self.root, width=50, height=10)

        self.generated\_words\_text2 = tk.Text(self.root, height=10)

        self.generated\_words\_text.config(state='disabled')

        self.cfg\_text.config(state='disabled')

        # Create buttons for generating context-free grammar and words

        self.generate\_cfg\_button = tk.Button(

            self.root, text='Сформировать КС грамматику', command=self.generate\_cfg

        )

        self.generate\_words\_button = tk.Button(

            self.root, text='Сформировать цепочки', command=self.generate\_words

        )

        self.upload\_regex\_button = tk.Button(

            self.root, text='Загрузить регулярное выражение', command=self.upload\_regex

        )

        self.save\_word\_button = tk.Button(

            self.root, text='Сохранить цепочки', command=self.save\_chains

        )

        self.author\_button = tk.Button(

            self.root, text='Автор', command=self.author

        )

        self.task\_button = tk.Button(

            self.root, text='Задание', command=self.task

        )

        # Place labels and entry fields on the window

        self.regex\_label.grid(row=0, column=0, padx=5, pady=5, sticky=tk.W)

        self.regex\_entry.grid(row=0, column=1, columnspan=10, padx=5, pady=5)

        self.min\_length\_label.grid(row=1, column=0, padx=5, pady=5, sticky=tk.W)

        self.min\_length\_entry.grid(row=1, column=1, padx=5, pady=5)

        self.max\_length\_label.grid(row=1, column=2, padx=5, pady=5, sticky=tk.W)

        self.max\_length\_entry.grid(row=1, column=3, padx=5, pady=5)

        self.cfg\_label = tk.Label(self.root, text="КС грамматика:")

        self.generated\_words\_label = tk.Label(self.root, text="Сгенерированные цепочки по КС грамматике:")

        self.generated\_words\_label2 = tk.Label(self.root, text="Сгенерированные цепочки по регулярному выражению:")

        self.separator = ttk.Separator(self.root, orient='vertical')

        self.separator2 = ttk.Separator(self.root, orient='vertical')

        self.cfg\_label.grid(row=2, column=0, padx=5, pady=5, sticky=tk.W)

        self.cfg\_text.grid(row=2, column=1, columnspan=10, padx=5, pady=5, sticky="nsew")

        self.generated\_words\_label.grid(row=3, column=0, padx=5, pady=5, sticky=tk.W)

        self.generated\_words\_label2.grid(row=4, column=0, padx=5, pady=5, sticky=tk.W)

        self.generated\_words\_text.grid(row=3, column=1, columnspan=10, padx=5, pady=5, sticky="ew")

        self.generated\_words\_text2.grid(row=4, column=1, columnspan=10, padx=5, pady=5, sticky="ew")

        self.generated\_words\_text.config(state='disabled')

        self.generated\_words\_text2.config(state='disabled')

        self.generate\_cfg\_button.grid(row=5, column=0, padx=5, pady=5)

        self.generate\_words\_button.grid(row=5, column=1, padx=5, pady=5)

        self.upload\_regex\_button.grid(row=5, column=3, padx=5, pady=5, sticky='nws')

        self.save\_word\_button.grid(row=5, column=4, padx=5, pady=5)

        self.separator2.grid(row=5, column=5, padx=0, pady=5, sticky='ns')

        self.author\_button.grid(row=5, column=6, padx=5, pady=5)

        self.task\_button.grid(row=5, column=7, padx=5, pady=5, sticky='nwe')

        self.root.mainloop()

    def generate\_cfg(self):

        try:

            self.cfg\_text.config(state='normal')

            self.generated\_words\_text.config(state='normal')

            self.generated\_words\_text2.config(state='normal')

            self.cfg\_text.delete(1.0, tk.END)

            self.generated\_words\_text.delete(1.0, tk.END)

            self.generated\_words\_text2.delete(1.0, tk.END)

            regex = self.regex\_entry.get()

            if not regex:

                messagebox.showerror('Ошибка', 'Не указано регулярное выражение!')

            self.regex = Regex(regex)

            self.cfg = Regex(regex).to\_cfg()

            formatted\_cfg\_dict = {}

            for production in self.cfg.productions:

                variable = production.head.to\_text()

                if variable not in formatted\_cfg\_dict.keys():

                    formatted\_cfg\_dict[variable] = []

                if len(production.body) > 0:

                    formatted\_cfg\_dict[variable].append(

                        "".join([x.to\_text() for x in production.body])

                    )

                else:

                    formatted\_cfg\_dict[variable].append("λ")

            output\_strings = []

            for key, value in formatted\_cfg\_dict.items():

                output\_strings.append(key + " -> " + "|".join(value))

            formatted\_cfg\_string = "\n".join(output\_strings)

            self.cfg\_text.insert(2.0, formatted\_cfg\_string)

            self.cfg\_text.config(state='disabled')

            self.generated\_words\_text.config(state='disabled')

            self.generated\_words\_text2.config(state='disabled')

        except:

            messagebox.showerror('Ошибка', 'Регулярное выражение некорректно! Повторите ввод')

    def generate\_words(self):

            self.generated\_words\_text.config(state='normal')

            self.generated\_words\_text.delete(1.0, tk.END)

            self.generated\_words\_text2.config(state='normal')

            self.generated\_words\_text2.delete(1.0, tk.END)

            if not self.cfg:

                messagebox.showerror('Ошибка', 'Не указана КС грамматика!')

                return

            try:

                min\_length = int(self.min\_length\_entry.get())

                if min\_length < 0:

                    raise 'Error'

            except:

                messagebox.showerror('Ошибка', 'Минимальная длина некорректна!')

                return

            try:

                max\_length = int(self.max\_length\_entry.get())

                if max\_length < 0 or max\_length < min\_length:

                    raise 'Error'

            except:

                messagebox.showerror('Ошибка', 'Максимальная длина некорректна!')

                return

            cfg\_words = []

            for word in self.cfg.get\_words(max\_length):

                if len(word) < min\_length:

                    continue

                word = "".join([x.to\_text() for x in word])

                cfg\_words.append(word)

                self.generated\_words\_text.insert(tk.END, f"{word}\n")

            regex\_words = []

            for word in self.regex.generate\_words(min\_length, max\_length):

                regex\_words.append(word)

                self.generated\_words\_text2.insert(tk.END, f"{word}\n")

            self.generated\_words\_text.config(state='disabled')

            self.generated\_words\_text2.config(state='disabled')

            if self.compare\_chains(cfg\_words, regex\_words):

                messagebox.showinfo('Результат', 'КС и регулярное выражение совпадают!')

    def upload\_regex(self):

        root = tk.Tk()

        root.withdraw()

        self.regex\_entry.delete(0, tk.END)

        file\_path = filedialog.askopenfilename()

        if file\_path:

            with open(file\_path, 'r') as file:

                regex = file.read()

            self.regex\_entry.insert(0, regex)

        else:

            return None

    def save\_chains(self):

        root = tk.Tk()

        root.withdraw()

        words = self.generated\_words\_text.get(1.0, tk.END)

        file\_path = filedialog.asksaveasfilename(defaultextension=".txt")

        if file\_path:

            with open(file\_path, 'w') as file:

                file.write(words)

        else:

            return None

    def author(self):

        messagebox.showinfo('Автор', 'ИП-014 Обухов А.И.')

    def task(self):

        task\_string = """Написать программу, которая по заданному регулярному выражению построит эквивалентную грамматику (по желанию разработчика – грамматика может быть контекстно-свободной или регулярной).

Программа должна сгенерировать по построенной грамматике и регулярному выражению множества всех цепочек в указанном диапазоне длин,

проверить их на совпадение. Процесс построения цепочек отображать на экране.

Для подтверждения корректности выполняемых действий предусмотреть возможность корректировки любого из построенных множеств пользователем (изменение цепочки, добавление, удаление…).

При обнаружении несовпадения в элементах множеств должна выдаваться диагностика различий – где именно несовпадения и в чём они состоят."""

        messagebox.showinfo('Задание', task\_string)

    def compare\_chains(self, a, b):

        return set(a) == set(b)

if \_\_name\_\_ == '\_\_main\_\_':

    window = RegexGeneratorGUI()

import random

from typing import Iterable

from utils.cfg.cfg import CFG

from utils import cfg

from utils.regular\_expression.regex\_reader import RegexReader

class Regex(RegexReader):

    def \_\_init\_\_(self, regex):

        self.head = None

        self.sons = None

        super().\_\_init\_\_(regex)

        self.\_counter = 0

        self.\_regex = self.\_regex.replace(" ", "")

        print(self.\_regex)

    def to\_cfg(self, starting\_symbol="S") -> CFG:

        productions, \_ = self.\_get\_production(starting\_symbol)

        cfg\_res = cfg.CFG(start\_symbol=cfg.utils.to\_variable(starting\_symbol),

                          productions=set(productions))

        return cfg\_res

    def \_get\_production(self, current\_symbol, count=0):

        next\_symbols = []

        next\_productions = []

        for son in self.sons:

            next\_symbol = "A" + str(count)

            count += 1

            new\_prods, count = son.\_get\_production(next\_symbol, count)

            next\_symbols.append(next\_symbol)

            next\_productions += new\_prods

        new\_prods = self.head.get\_cfg\_rules(current\_symbol, next\_symbols)

        next\_productions += new\_prods

        return next\_productions, count

    def \_\_repr\_\_(self):

        return self.head.get\_str\_repr([str(son) for son in self.sons])

    def from\_string(self, regex\_str: str):

        return Regex(regex\_str)

    def generate\_words(self, min\_length, max\_length):

        range\_1 = min\_length

        range\_2 = max\_length

        reg\_ex\_chain = self.\_regex

        split\_chain = []

        service\_symbols = []

        counter = 0

        start\_position = 0

        sym\_num = 0

        while sym\_num < len(reg\_ex\_chain):

            if reg\_ex\_chain[sym\_num] == '(':

                counter += 1

            elif reg\_ex\_chain[sym\_num] == ')':

                counter -= 1

            if counter == 0:

                if sym\_num == len(reg\_ex\_chain) - 1 or reg\_ex\_chain[sym\_num+1] == '(' or reg\_ex\_chain[sym\_num+1] == '+' or reg\_ex\_chain[sym\_num+1] == '\*' or reg\_ex\_chain[sym\_num+1] == '⁺':

                    if reg\_ex\_chain[start\_position] != '(':

                        split\_chain.append([[reg\_ex\_chain[start\_position:sym\_num+1]]])

                    else:

                        split\_chain.append([[reg\_ex\_chain[(start\_position+1):sym\_num]]])

                    if sym\_num == len(reg\_ex\_chain) - 1:

                        service\_symbols.append([['-']])

                        start\_position = sym\_num + 1

                    elif reg\_ex\_chain[sym\_num+1] == '(':

                        service\_symbols.append([['-']])

                        start\_position = sym\_num + 1

                    elif reg\_ex\_chain[sym\_num+1] == '+':

                        service\_symbols.append([['+']])

                        sym\_num += 1

                        start\_position = sym\_num + 1

                    elif reg\_ex\_chain[sym\_num+1] == '\*':

                        if sym\_num == len(reg\_ex\_chain) - 2:

                            service\_symbols.append([['\*']])

                            sym\_num += 1

                            start\_position = sym\_num + 1

                        elif reg\_ex\_chain[sym\_num+2] != '+':

                            service\_symbols.append([['\*']])

                            sym\_num += 1

                            start\_position = sym\_num + 1

                        else:

                            service\_symbols.append([['\*+']])

                            sym\_num += 2

                            start\_position = sym\_num + 1

                    elif reg\_ex\_chain[sym\_num+1] == '⁺':

                        if sym\_num == len(reg\_ex\_chain) - 2:

                            service\_symbols.append([['⁺']])

                            sym\_num += 1

                            start\_position = sym\_num + 1

                        elif reg\_ex\_chain[sym\_num+2] != '+':

                            service\_symbols.append([['⁺']])

                            sym\_num += 1

                            start\_position = sym\_num + 1

                        else:

                            service\_symbols.append([['⁺+']])

                            sym\_num += 2

                            start\_position = sym\_num + 1

                    else:

                        raise 'Error'

            sym\_num += 1

        def split\_reg\_ex(service\_symbols\_rec, split\_chain\_rec):

            for ss\_num in range(0,len(service\_symbols\_rec)):

                if service\_symbols\_rec[ss\_num][0][0] != '-':

                    split\_chain\_rec[ss\_num].append([])

                    service\_symbols\_rec[ss\_num].append([])

                    counter = 0

                    start\_position = 0

                    sym\_num = 0

                    while sym\_num < len(split\_chain\_rec[ss\_num][0][0]):

                        if split\_chain\_rec[ss\_num][0][0][sym\_num] == '(':

                            counter += 1

                        elif split\_chain\_rec[ss\_num][0][0][sym\_num] == ')':

                            counter -= 1

                        if counter == 0:

                            if sym\_num == len(split\_chain\_rec[ss\_num][0][0]) - 1 or split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '(' or split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '+' or split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '\*' or split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '⁺':

                                if split\_chain\_rec[ss\_num][0][0][start\_position] != '(':

                                    split\_chain\_rec[ss\_num][1].append([[split\_chain\_rec[ss\_num][0][0][start\_position:sym\_num+1]]])

                                else:

                                    split\_chain\_rec[ss\_num][1].append([[split\_chain\_rec[ss\_num][0][0][(start\_position+1):sym\_num]]])

                                if sym\_num == len(split\_chain\_rec[ss\_num][0][0]) - 1:

                                    service\_symbols\_rec[ss\_num][1].append([['-']])

                                    start\_position = sym\_num + 1

                                elif split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '(':

                                    service\_symbols\_rec[ss\_num][1].append([['-']])

                                    start\_position = sym\_num + 1

                                elif split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '+':

                                    service\_symbols\_rec[ss\_num][1].append([['+']])

                                    sym\_num += 1

                                    start\_position = sym\_num + 1

                                elif split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '\*':

                                    if sym\_num == len(split\_chain\_rec[ss\_num][0][0]) - 2:

                                        service\_symbols\_rec[ss\_num][1].append([['\*']])

                                        sym\_num += 1

                                        start\_position = sym\_num + 1

                                    elif split\_chain\_rec[ss\_num][0][0][sym\_num+2] != '+':

                                        service\_symbols\_rec[ss\_num][1].append([['\*']])

                                        sym\_num += 1

                                        start\_position = sym\_num + 1

                                    else:

                                        service\_symbols\_rec[ss\_num][1].append([['\*+']])

                                        sym\_num += 2

                                        start\_position = sym\_num + 1

                                elif split\_chain\_rec[ss\_num][0][0][sym\_num+1] == '⁺':

                                    if sym\_num == len(split\_chain\_rec[ss\_num][0][0]) - 2:

                                        service\_symbols\_rec[ss\_num][1].append([['⁺']])

                                        sym\_num += 1

                                        start\_position = sym\_num + 1

                                    elif split\_chain\_rec[ss\_num][0][0][sym\_num+2] != '+':

                                        service\_symbols\_rec[ss\_num][1].append([['⁺']])

                                        sym\_num += 1

                                        start\_position = sym\_num + 1

                                    else:

                                        service\_symbols\_rec[ss\_num][1].append([['⁺+']])

                                        sym\_num += 2

                                        start\_position = sym\_num + 1

                                else:

                                    raise 'Error'

                        sym\_num += 1

                    split\_reg\_ex(service\_symbols\_rec[ss\_num][1], split\_chain\_rec[ss\_num][1])

        split\_reg\_ex(service\_symbols, split\_chain)

        def generate\_chains(service\_symbols\_rec, split\_chain\_rec, range\_1, range\_2, level = 0, cnt = 0):

            chains = ['']

            is\_pluses = False

            ss\_num = 0

            for a in service\_symbols\_rec:

                if a[0][0] == '+' or a[0][0] == '-':

                    cnt += 1

            while ss\_num < len(service\_symbols\_rec):

                if service\_symbols\_rec[ss\_num][0][0] == '-':

                    buf = split\_chain\_rec[ss\_num][0][0].split('+')

                elif service\_symbols\_rec[ss\_num][0][0] == '+':

                    buf = split\_chain\_rec[ss\_num][0][0].split('+')

                    is\_pluses = True

                elif service\_symbols\_rec[ss\_num][0][0] == '⁺':

                    buf = generate\_chains(service\_symbols\_rec[ss\_num][1], split\_chain\_rec[ss\_num][1], range\_1, range\_2, level+1, cnt)

                    size = len(buf)

                    size\_2 = len(buf)

                    start\_pos = 0

                    is\_next = True

                    while is\_next:

                        is\_next = False

                        for b\_num in range(0, size):

                            for b\_num\_2 in range(start\_pos, size\_2):

                                if len(buf[b\_num]+buf[b\_num\_2]) <= range\_2:

                                    buf.append(buf[b\_num]+buf[b\_num\_2])

                                    is\_next = True

                        start\_pos = size\_2-1

                        size\_2 = len(buf) - size\_2

                elif service\_symbols\_rec[ss\_num][0][0] == '\*':

                    buf = generate\_chains(service\_symbols\_rec[ss\_num][1], split\_chain\_rec[ss\_num][1], range\_1, range\_2, level+1, cnt)

                    size = len(buf)

                    size\_2 = len(buf)

                    start\_pos = 0

                    is\_next = True

                    while is\_next:

                        is\_next = False

                        for b\_num in range(0, size):

                            for b\_num\_2 in range(start\_pos, size\_2):

                                if len(buf[b\_num]+buf[b\_num\_2]) <= range\_2:

                                    buf.append(buf[b\_num]+buf[b\_num\_2])

                                    is\_next = True

                        start\_pos = size\_2-1

                        size\_2 = len(buf) - size\_2

                    buf.append('')

                    buf\_3 = []

                    if len(service\_symbols\_rec)-1 > ss\_num:

                        if service\_symbols\_rec[ss\_num+1][0][0] == '+':

                            buf\_2 = split\_chain\_rec[ss\_num+1][0][0].split('+')

                            is\_pluses = True

                            for buf\_num in range(0,len(buf)):

                                for buf\_2\_num in range(0,len(buf\_2)):

                                    buf\_3.append(buf[buf\_num]+buf\_2[buf\_2\_num])

                            buf = buf\_3

                            ss\_num += 1

                        elif service\_symbols\_rec[ss\_num+1][0][0] == '-':

                            buf\_2 = split\_chain\_rec[ss\_num+1][0][0].split('+')

                            for buf\_num in range(0,len(buf)):

                                for buf\_2\_num in range(0,len(buf\_2)):

                                    buf\_3.append(buf[buf\_num]+buf\_2[buf\_2\_num])

                            buf = buf\_3

                            ss\_num += 1

                elif service\_symbols\_rec[ss\_num][0][0] == '\*+':

                    buf = generate\_chains(service\_symbols\_rec[ss\_num][1], split\_chain\_rec[ss\_num][1], range\_1, range\_2, level+1, cnt)

                    size = len(buf)

                    size\_2 = len(buf)

                    start\_pos = 0

                    is\_next = True

                    while is\_next:

                        is\_next = False

                        for b\_num in range(0, size):

                            for b\_num\_2 in range(start\_pos, size\_2):

                                if len(buf[b\_num]+buf[b\_num\_2]) <= range\_2:

                                    buf.append(buf[b\_num]+buf[b\_num\_2])

                                    is\_next = True

                        start\_pos = size\_2-1

                        size\_2 = len(buf) - size\_2

                    buf.append('')

                    is\_pluses = True

                elif service\_symbols\_rec[ss\_num][0][0] == '⁺+':

                    buf = generate\_chains(service\_symbols\_rec[ss\_num][1], split\_chain\_rec[ss\_num][1], range\_1, range\_2, level+1, cnt)

                    size = len(buf)

                    size\_2 = len(buf)

                    start\_pos = 0

                    is\_next = True

                    while is\_next:

                        is\_next = False

                        for b\_num in range(0, size):

                            for b\_num\_2 in range(start\_pos, size\_2):

                                if len(buf[b\_num]+buf[b\_num\_2]) <= range\_2:

                                    buf.append(buf[b\_num]+buf[b\_num\_2])

                                    is\_next = True

                        start\_pos = size\_2-1

                        size\_2 = len(buf) - size\_2

                    is\_pluses = True

                if not is\_pluses:

                    counter = len(chains)

                    while counter != 0:

                        for elem in buf:

                            if len(chains[0]+elem) <= range\_2:

                                chains.append(chains[0]+elem)

                        chains.pop(0)

                        counter -= 1

                else:

                    for elem in buf:

                        chains.append(chains[0]+elem)

                ss\_num += 1

            if is\_pluses:

                chains.pop(0)

            chains\_set = set(chains)

            chain\_list = []

            if level == 0:

                for chain in list(chains\_set):

                    if len(chain) >= range\_1 and len(chain) <= range\_2:

                        chain\_list.append(chain)

            else:

                for chain in list(chains\_set):

                    if len(chain) <= range\_2:

                        chain\_list.append(chain)

            return(chain\_list)

        resulting\_chains = generate\_chains(service\_symbols, split\_chain, range\_1, range\_2)

        return resulting\_chains

import string

from copy import deepcopy

from typing import AbstractSet, Iterable, Tuple, Dict, Any

import networkx as nx

from .cfg\_object import CFGObject

from .epsilon import Epsilon

from .production import Production

from .terminal import Terminal

from .utils import to\_variable, to\_terminal

from .utils\_cfg import remove\_nullable\_production, get\_productions\_d

from .variable import Variable

EPSILON\_SYMBOLS = ["\_"]

class NotParsableException(Exception):

    """"""

def is\_special\_text(text):

    return len(text) > 5 and \

        (text[0:5] == '"VAR:' or text[0:5] == '"TER:') and \

        text[-1] == '"'

class CFG:

    def \_\_init\_\_(self,

                 variables: AbstractSet[Variable] = None,

                 terminals: AbstractSet[Terminal] = None,

                 start\_symbol: Variable = None,

                 productions: Iterable[Production] = None):

        if variables is not None:

            variables = {to\_variable(x) for x in variables}

        self.\_variables = variables or set()

        self.\_variables = set(self.\_variables)

        if terminals is not None:

            terminals = {to\_terminal(x) for x in terminals}

        self.\_terminals = terminals or set()

        self.\_terminals = set(self.\_terminals)

        if start\_symbol is not None:

            start\_symbol = to\_variable(start\_symbol)

        self.\_start\_symbol = start\_symbol

        if start\_symbol is not None:

            self.\_variables.add(start\_symbol)

        self.\_productions = productions or set()

        self.\_productions = self.\_productions

        for production in self.\_productions:

            self.\_\_initialize\_production\_in\_cfg(production)

        self.\_normal\_form = None

        self.\_generating\_symbols = None

        self.\_nullable\_symbols = None

        self.\_impacts = None

        self.\_remaining\_lists = None

        self.\_added\_impacts = None

    def \_\_initialize\_production\_in\_cfg(self, production):

        self.\_variables.add(production.head)

        for cfg\_object in production.body:

            if isinstance(cfg\_object, Terminal):

                self.\_terminals.add(cfg\_object)

            else:

                self.\_variables.add(cfg\_object)

    def get\_generating\_symbols(self) -> AbstractSet[CFGObject]:

        if self.\_generating\_symbols is None:

            self.\_generating\_symbols = self.\_get\_generating\_or\_nullable(False)

        return self.\_generating\_symbols

    def \_get\_generating\_or\_nullable(self, nullable=False):

        """ Merge of nullable and generating """

        to\_process = [Epsilon()]

        g\_symbols = {Epsilon()}

        self.\_set\_impacts\_and\_remaining\_lists()

        for symbol in self.\_added\_impacts:

            if symbol not in g\_symbols:

                g\_symbols.add(symbol)

                to\_process.append(symbol)

        if not nullable:

            for terminal in self.\_terminals:

                g\_symbols.add(terminal)

                to\_process.append(terminal)

        processed\_with\_modification = []

        while to\_process:

            current = to\_process.pop()

            for symbol\_impact, index\_impact in self.\_impacts.get(current, []):

                if symbol\_impact in g\_symbols:

                    continue

                processed\_with\_modification.append(

                    (symbol\_impact, index\_impact))

                self.\_remaining\_lists[symbol\_impact][index\_impact] -= 1

                if self.\_remaining\_lists[symbol\_impact][index\_impact] == 0:

                    g\_symbols.add(symbol\_impact)

                    to\_process.append(symbol\_impact)

        # Fix modifications

        for symbol\_impact, index\_impact in processed\_with\_modification:

            self.\_remaining\_lists[symbol\_impact][index\_impact] += 1

        g\_symbols.remove(Epsilon())

        return g\_symbols

    def \_set\_impacts\_and\_remaining\_lists(self):

        if self.\_impacts is not None:

            return

        self.\_added\_impacts = set()

        self.\_remaining\_lists = {}

        self.\_impacts = {}

        for production in self.\_productions:

            head = production.head  # Should check if head is not Epsilon?

            body = production.body

            if not body:

                self.\_added\_impacts.add(head)

                continue

            temp = self.\_remaining\_lists.setdefault(head, [])

            temp.append(len(body))

            index\_impact = len(temp) - 1

            for symbol in body:

                self.\_impacts.setdefault(symbol, []).append(

                    (head, index\_impact))

    def get\_reachable\_symbols(self) -> AbstractSet[CFGObject]:

        r\_symbols = set()

        r\_symbols.add(self.\_start\_symbol)

        reachable\_transition\_d = {}

        for production in self.\_productions:

            temp = reachable\_transition\_d.setdefault(production.head, [])

            for symbol in production.body:

                if not isinstance(symbol, Epsilon):

                    temp.append(symbol)

        to\_process = [self.\_start\_symbol]

        while to\_process:

            current = to\_process.pop()

            for next\_symbol in reachable\_transition\_d.get(current, []):

                if next\_symbol not in r\_symbols:

                    r\_symbols.add(next\_symbol)

                    to\_process.append(next\_symbol)

        return r\_symbols

    def remove\_useless\_symbols(self) -> "CFG":

        generating = self.get\_generating\_symbols()

        productions = [x for x in self.\_productions

                       if x.head in generating and

                       all(y in generating for y in x.body)]

        new\_var = self.\_variables.intersection(generating)

        new\_ter = self.\_terminals.intersection(generating)

        cfg\_temp = CFG(new\_var, new\_ter, self.\_start\_symbol, productions)

        reachables = cfg\_temp.get\_reachable\_symbols()

        productions = [x for x in productions

                       if x.head in reachables]

        new\_var = new\_var.intersection(reachables)

        new\_ter = new\_ter.intersection(reachables)

        return CFG(new\_var, new\_ter, self.\_start\_symbol, productions)

    def get\_nullable\_symbols(self) -> AbstractSet[CFGObject]:

        if self.\_nullable\_symbols is None:

            self.\_nullable\_symbols = self.\_get\_generating\_or\_nullable(True)

        return self.\_nullable\_symbols

    def remove\_epsilon(self) -> "CFG":

        new\_productions = []

        nullables = self.get\_nullable\_symbols()

        for production in self.\_productions:

            new\_productions += remove\_nullable\_production(production,

                                                          nullables)

        return CFG(self.\_variables,

                   self.\_terminals,

                   self.\_start\_symbol,

                   new\_productions)

    def get\_unit\_pairs(self) -> AbstractSet[Tuple[Variable, Variable]]:

        unit\_pairs = set()

        for variable in self.\_variables:

            unit\_pairs.add((variable, variable))

        productions = [x

                       for x in self.\_productions

                       if len(x.body) == 1 and isinstance(x.body[0], Variable)]

        productions\_d = get\_productions\_d(productions)

        to\_process = list(unit\_pairs)

        while to\_process:

            var\_a, var\_b = to\_process.pop()

            for production in productions\_d.get(var\_b, []):

                temp = (var\_a, production.body[0])

                if temp not in unit\_pairs:

                    unit\_pairs.add(temp)

                    to\_process.append(temp)

        return unit\_pairs

    def eliminate\_unit\_productions(self) -> "CFG":

        unit\_pairs = self.get\_unit\_pairs()

        productions = [x

                       for x in self.\_productions

                       if len(x.body) != 1

                       or not isinstance(x.body[0], Variable)]

        productions\_d = get\_productions\_d(productions)

        for var\_a, var\_b in unit\_pairs:

            for production in productions\_d.get(var\_b, []):

                productions.append(Production(var\_a, production.body,

                                              filtering=False))

        return CFG(self.\_variables,

                   self.\_terminals,

                   self.\_start\_symbol,

                   productions)

    def \_get\_productions\_with\_only\_single\_terminals(self):

        term\_to\_var = {}

        new\_productions = []

        for terminal in self.\_terminals:

            var = Variable(str(terminal.value) + "#CNF#")

            term\_to\_var[terminal] = var

        used = set()

        for production in self.\_productions:

            if len(production.body) == 1:

                new\_productions.append(production)

                continue

            new\_body = []

            for symbol in production.body:

                if symbol in term\_to\_var:

                    new\_body.append(term\_to\_var[symbol])

                    used.add(symbol)

                else:

                    new\_body.append(symbol)

            new\_productions.append(Production(production.head,

                                              new\_body))

        for terminal in used:

            new\_productions.append(

                Production(term\_to\_var[terminal], [terminal]))

        return new\_productions

    def \_get\_next\_free\_variable(self, idx, prefix):

        idx += 1

        temp = Variable(prefix + str(idx))

        while temp in self.\_variables:

            idx += 1

            temp = Variable(prefix + str(idx))

        return idx, temp

    def \_decompose\_productions(self, productions):

        idx = 0

        new\_productions = []

        done = {}

        for production in productions:

            body = production.body

            if len(body) <= 2:

                new\_productions.append(production)

                continue

            new\_var = []

            for \_ in range(len(body) - 2):

                idx, var = self.\_get\_next\_free\_variable(idx, "C#CNF#")

                new\_var.append(var)

            head = production.head

            stopped = False

            for i in range(len(body) - 2):

                temp = tuple(body[i + 1:])

                if temp in done:

                    new\_productions.append(Production(head,

                                                      [body[i], done[temp]]))

                    stopped = True

                    break

                new\_productions.append(Production(head, [body[i], new\_var[i]]))

                done[temp] = new\_var[i]

                head = new\_var[i]

            if not stopped:

                new\_productions.append(Production(head, [body[-2], body[-1]]))

        return new\_productions

    def to\_normal\_form(self) -> "CFG":

        if self.\_normal\_form is not None:

            return self.\_normal\_form

        nullables = self.get\_nullable\_symbols()

        unit\_pairs = self.get\_unit\_pairs()

        generating = self.get\_generating\_symbols()

        reachables = self.get\_reachable\_symbols()

        if (len(nullables) != 0 or len(unit\_pairs) != len(self.\_variables) or

                len(generating) !=

                len(self.\_variables) + len(self.\_terminals) or

                len(reachables) !=

                len(self.\_variables) + len(self.\_terminals)):

            if len(self.\_productions) == 0:

                self.\_normal\_form = self

                return self

            new\_cfg = self.remove\_useless\_symbols() \

                .remove\_epsilon() \

                .remove\_useless\_symbols() \

                .eliminate\_unit\_productions() \

                .remove\_useless\_symbols()

            cfg = new\_cfg.to\_normal\_form()

            self.\_normal\_form = cfg

            return cfg

        # Remove terminals from body

        new\_productions = self.\_get\_productions\_with\_only\_single\_terminals()

        new\_productions = self.\_decompose\_productions(new\_productions)

        cfg = CFG(start\_symbol=self.\_start\_symbol,

                  productions=set(new\_productions))

        self.\_normal\_form = cfg

        return cfg

    @property

    def variables(self) -> AbstractSet[Variable]:

        return self.\_variables

    @property

    def terminals(self) -> AbstractSet[Terminal]:

        return self.\_terminals

    @property

    def productions(self) -> AbstractSet[Production]:

        return self.\_productions

    @property

    def start\_symbol(self) -> Variable:

        return self.\_start\_symbol

    def get\_words(self, max\_length: int = -1):

        nullables = self.get\_nullable\_symbols()

        if self.start\_symbol in nullables:

            yield []

        if max\_length == 0:

            return

        cfg = self.to\_normal\_form()

        productions = cfg.productions

        gen\_d = {}

        for production in productions:

            if production.head not in gen\_d:

                gen\_d[production.head] = [[]]

            if len(production.body) == 2:

                for obj in production.body:

                    if obj not in gen\_d:

                        gen\_d[obj] = [[]]

        for production in productions:

            body = production.body

            if len(body) == 1:

                if len(gen\_d[production.head]) == 1:

                    gen\_d[production.head].append([])

                if body not in gen\_d[production.head][-1]:

                    gen\_d[production.head][-1].append(list(body))

                    if production.head == cfg.start\_symbol:

                        yield list(body)

        current\_length = 2

        total\_no\_modification = 0

        while current\_length <= max\_length or max\_length == -1:

            was\_modified = False

            for gen in gen\_d.values():

                if len(gen) != current\_length:

                    gen.append([])

            for production in productions:

                body = production.body

                if len(gen\_d[production.head]) != current\_length + 1:

                    gen\_d[production.head].append([])

                if len(body) != 2:

                    continue

                for i in range(1, current\_length):

                    j = current\_length - i

                    for left in gen\_d[body[0]][i]:

                        for right in gen\_d[body[1]][j]:

                            new\_word = left + right

                            if new\_word not in gen\_d[production.head][-1]:

                                was\_modified = True

                                gen\_d[production.head][-1].append(new\_word)

                                if production.head == cfg.start\_symbol:

                                    yield new\_word

            if was\_modified:

                total\_no\_modification = 0

            else:

                total\_no\_modification += 1

            current\_length += 1

            if total\_no\_modification > current\_length / 2:

                return

    def to\_text(self):

        res = []

        for production in self.\_productions:

            res.append(str(production.head) + " -> " +

                       " ".join([x.to\_text() for x in production.body]))

        return "\n".join(res) + "\n"