ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ ПО КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ

КОМПЮТЪРНО И СОФТУЕРНО ИНЖЕНЕРСТВО

Формални езици и езикови процеси:

Курсова работа:

„ Графично представяне на регулярни изрази.
Синтактичен граф генератор“

Съставил:	Факултетен номер:	Група:
Ангел Любомиров Стойнов	121222150	40

Съдържание

Въведение:

Регулярни изрази:

Регулярните изрази са удобен начин за задаване на регулярни множества. Те се образуват от операнди, представляващи множества от низове, и знаците за операции (подредени според естествения приоритетен ред):

Основни операции при регулярните изрази:

- **Дизюнкция (Alternation):** Представлява избора между два или повече варианта. Например, изразът a|b ще съвпадне с 'a' или 'b'.
- **Конкатенация (Concatenation):** Последователно свързване на символи или изрази. Например, ab ще съвпадне с низа 'ab'.
- **Итерация (Kleene Star):** Позволява повторение на предходния елемент нула или повече пъти. Например, a* ще съвпадне с '', 'a', 'aa', 'aaa' и т.н.

Приоритетният ред може да се промени чрез въвеждане на скоби ( ) за формиране на под изрази. РИ са средство за задаване, описание, определение на РМ. Формалната дефиниция на РИ включват 5 правила, които съответстват на 5-те правила на определение на РМ.

Изразите се определят рекурсивно по следния начин:

1. ∅ е регулярен израз, означаващ регулярното множество ∅;
2. ε е регулярен израз, означаващ регулярното множество {ε};
3. Ако a ∈ Σ, a е регулярен израз, означаващ множеството {a};
4. Ако p и q са регулярни изрази, означаващи регулярните множества P и Q, тогава:
   - (p|q) е регулярен израз, означаващ P|Q;
   - (pq) е регулярен израз, означаващ P.Q;
   - (p)* е регулярен израз, означаващ P*;

Графично представяне на регулярни изрази и граматики:

Синтактични графи (Syntax Graphs):

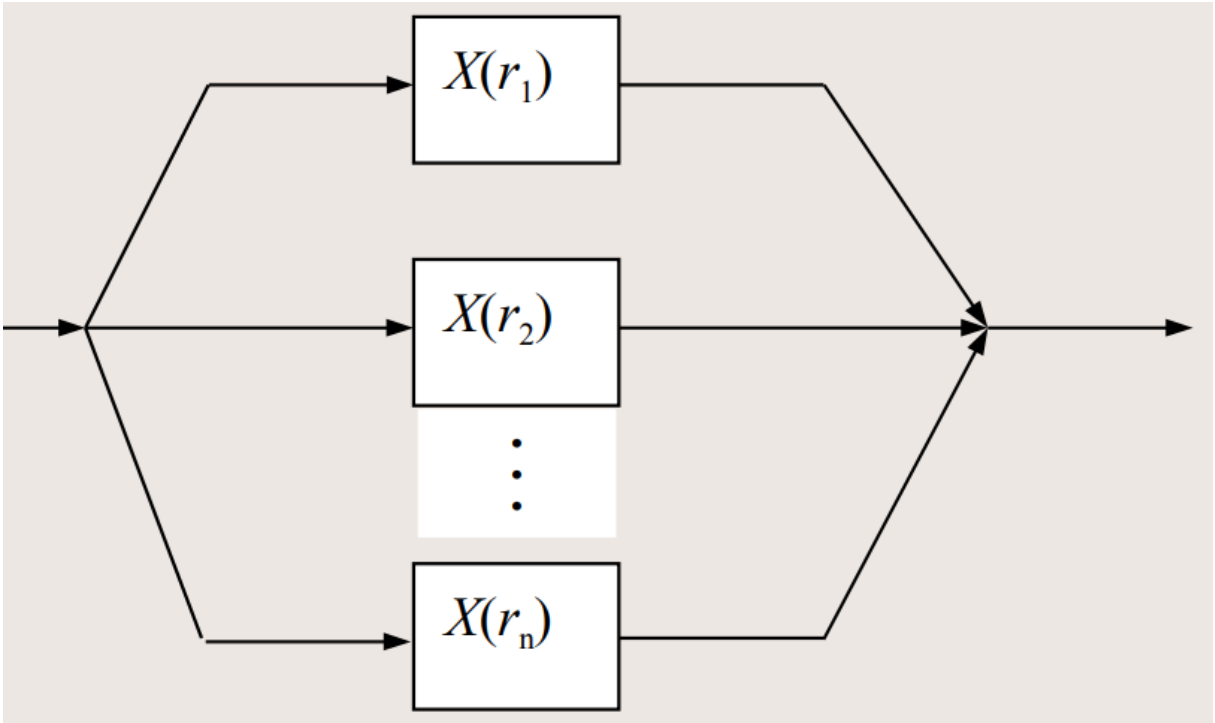
Синтактичните графи представляват графично представяне на регулярни изрази (РИ), използвани като генератори на низове. Те са съставени от колекция възли, свързани чрез насочени дъги, които определят посоката на преминаване през графа.

Основни характеристики:

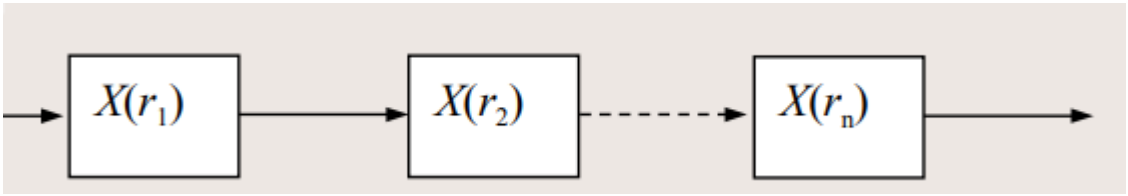
- Насочен граф – дъгите имат зададена посока.
- **Типове възли**:
  - **Елементарен генератор** – генерира конкретен символ или поредица от символи.
  - **Възел разклонение** – позволява избор между няколко алтернативни пътя (дизюнкция).
  - **Възел обединение** – обединява различни последователности (конкатенация).

Поддържани операции:

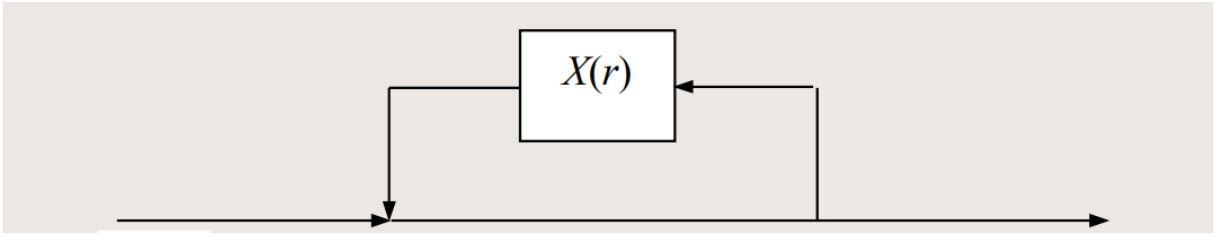
- **Дизюнкция (алтернатива)** – представя избор между няколко възможни пътя.



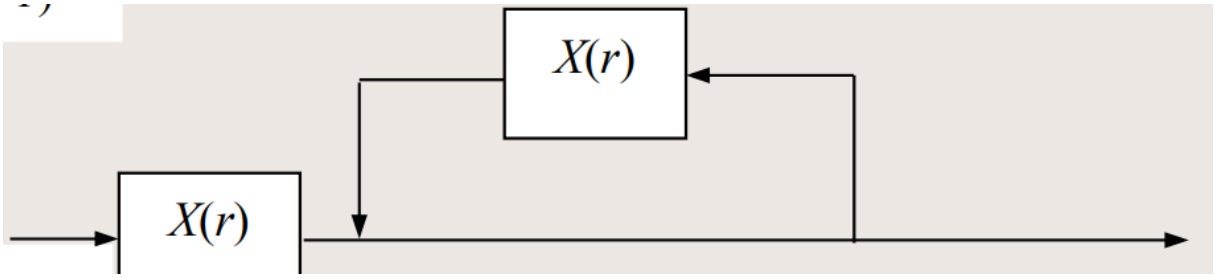
- **Конкатенация** – последователност от символи или групи.



- **Итерация** – повторение на елементите нула или повече пъти (*).



- **Непразна итерация** – повторение поне веднъж (+).



Синтактични диаграми (Railroad Diagrams)

Синтактичните диаграми, известни още като Railroad diagrams, представляват визуален метод за описание на безконтекстни граматики. Те са графична алтернатива на формати като BNF (Backus–Naur Form) и EBNF и често се използват поради своята интуитивност и лесно разбиране, особено от неспециалисти.

Основни характеристики:

- Визуално представяне на синтактичните правила на граматики.
- Често използвани в документация и спецификации заради своята яснота и удобство за четене.
- Съдържат последователности, алтернативи и повторения, изобразени чрез разклонения и циклични пътища.

История:

Стивън Клийн е изобретил регулярните изрази в средата на 50-те години на миналия век, като обозначение за крайни автомати. Всъщност, те са еквивалентни на крайните автомати по отношение на това, което представят – формално описание на множеството от низове, които автоматът може да разпознае.

Първоначално регулярните изрази се появяват в програмен контекст във версията на текстовия редактор QED, разработена от Кен Томпсън, през средата на 60-те години. През 1967 г. Томпсън подава заявка за патент за механизъм за бързо съвпадение на текст, базиран на регулярни изрази. Патентът е издаден през 1971 г. и се счита за един от първите патенти за софтуер.

Тези иновативни идеи не само са положили основите за теорията на формалните езици и крайните автомати, но и значително повлияват на развитието на инструменти за текстова обработка. Работата на Томпсън допринася за създаването на Unix-базирани програми, като grep, sed и awk, които и до днес са ключови в анализа и манипулирането на текстови данни.

Защо го ползваме?

Регулярните изрази и синтактичните диаграми намират широко приложение поради следните причини:

- **Ефективност и прецизност:** Регулярните изрази позволяват бързо и точно търсене и обработка на текстови данни, като разпознават специфични шаблони без необходимост от сложна логика.
- **Гъвкавост в обработката на данни:** С тях може лесно да се проверява валидността на входни данни (например имейли, URL адреси, телефонни номера), да се извличат конкретни части от текст или да се извършват замени.
- **Приложение в множество технологии:** Много езици за програмиране – като Python, JavaScript, C# – вграждат поддръжка за регулярни изрази, което улеснява бързото им интегриране във всякакви приложения.
- **Подобряване на визуализацията на граматики:** Синтактичните диаграми предоставят визуално представяне на граматиката, което може да бъде по-разбираемо за неспециалисти и полезно при обучението или дизайна на езици за програмиране.
- **Поддръжка при разработката на инструменти:** Те се използват от генератори на парсъри и компилатори за визуализиране на синтактичната структура на езиците, което спомага за по-доброто разбиране и поддръжка на сложни системи.

Документация по кода:

Използвани технологии за реализация на задачата:

- Python
- Graphviz е open source граф визуализиращ софтуер. Част от Python библиотеката – Diagraph*

*** Линк към graphviz за повече информация.**

Цели и функционалности:

- Графично представяне на регулярни изрази чрез синтактичен граф генератор.
- Потребителят въвежда през конзолата регулярен израз, като получава графичното представяне в .pdf файл.

Преди стартиране на проекта:

1. Трябва да имате свален Python*
2. Трябва да имате свален graphviz и да го конфигурирате в user/system environment variables**

*** Сваляне на Python**

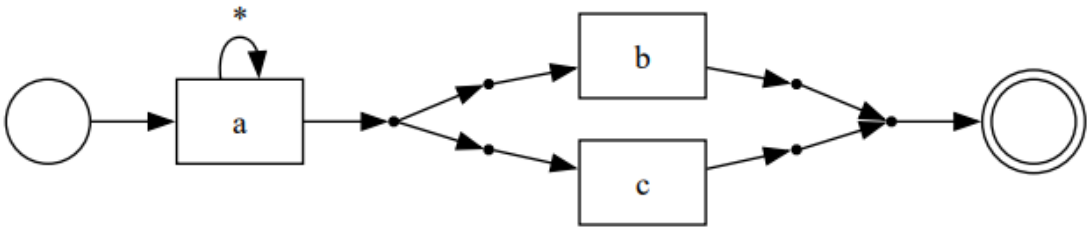
**** Сваляне на graphyz**

Как да конфигурираме environment variables.

Изглед на приложението:

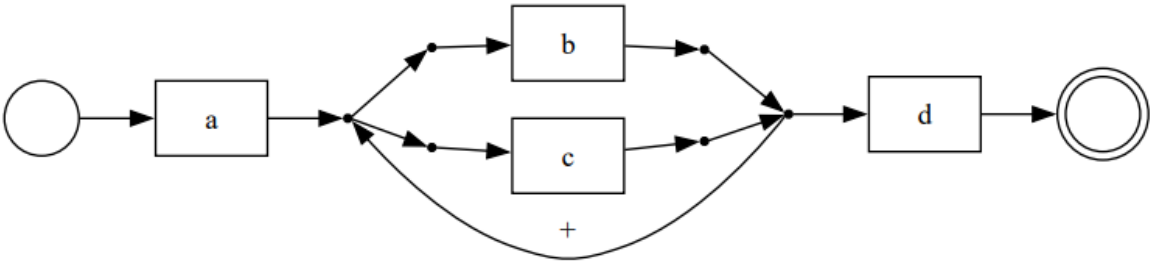
При въвеждане на a*(b|c):

```
Enter a simple regular expression (supports a-z, *, +, |, () ): a*(b|c)
Diagram saved to: /mnt/data/user_regexper_diagram.pdf
```



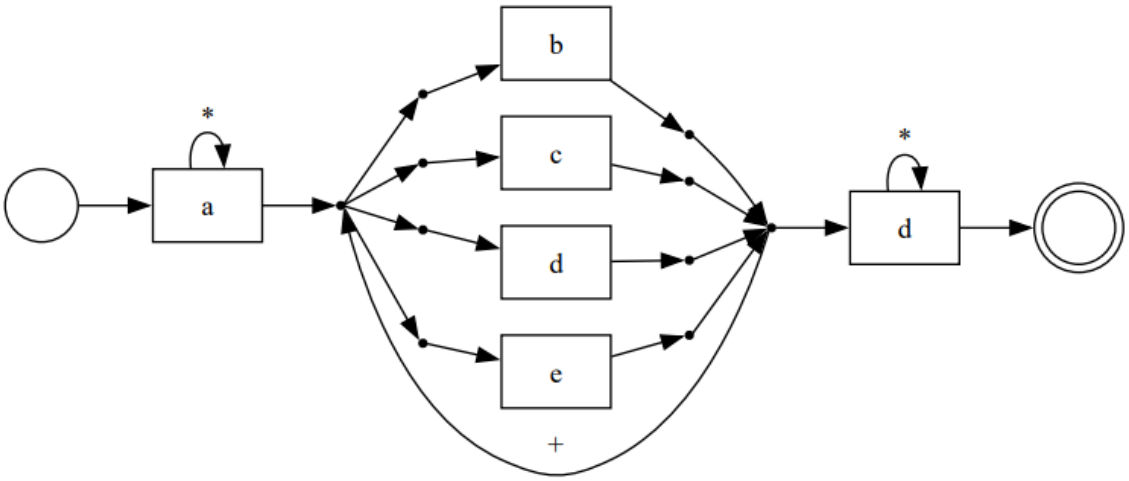
При въвеждане на a(b|c)*d

```
Enter a simple regular expression (supports a-z, *, +, |, () ): a(b|c)+d
Diagram saved to: /mnt/data/user_regexper_diagram.pdf
```



При въвеждане на по-сложен пример a*(b|c|d|e)+d*

```
Enter a simple regular expression (supports a-z, *, +, |, () ): a*(b|c|d|e)+d*
Diagram saved to: /mnt/data/user_regexper_diagram.pdf
```



При въвеждане на некоректен пример:

```
Enter a simple regular expression (supports a-z, *, +, |, () ): a#(b|c)@d+
Error: Invalid characters detected!
Enter a simple regular expression (supports a-z, *, +, |, () ):
```

Реализация:

Линк към проекта: https://github.com/StoynovAngel/FEEP

```
from graphviz import Digraph
import re
class RegexperStyleDiagram:
    REGEX_PATTERN = r'^[a-zA-Z()*+|]+$'
    def __init__(self):
        self.dot = Digraph(graph_attr={'rankdir': 'LR'})
        self.node_id = 0
    def new_node(self, label='', shape='box'):
        name = f"n{self.node_id}"
        self.node_id += 1
        self.dot.node(name, label=label, shape=shape)
        return name
    def build(self, regex: str):
        start = self.new_node('', shape='circle')
        end = self.new_node('', shape='doublecircle')
        self._process_expr(regex, start, end, self.dot)
        return self.dot
    def process_alternatives(self, parts: list[str], entry: str, exit: str, g: Digraph):
        for part in parts:
            branch_start = self.new_node('', shape='point')
            branch_end = self.new_node('', shape='point')
            g.edge(entry, branch_start)
            self._process_expr(part, branch_start, branch_end, g)
            g.edge(branch_end, exit)
```

```
def _process_expr(self, expr: str, entry: str, exit: str, g: Digraph):
    parts = logical_or(expr)
    if len(parts) > 1:
        self.process_alternatives(parts, entry, exit, g)
    else:
        i = 0
        prev = entry
        while i < len(expr):
            c = expr[i]
            if c == '(':
                j = i + 1
                depth = 1
                while j < len(expr):
                    if expr[j] == '(':
                        depth += 1
                    elif expr[j] == ')':
                        depth -= 1
                    if depth == 0:
                        break
                    j += 1
                group_content = expr[i + 1:j]
                group_entry = self.new_node('', shape='point')
                group_exit = self.new_node('', shape='point')
                g.edge(prev, group_entry)
                self._process_expr(group_content, group_entry, group_exit, g)
                if j + 1 < len(expr) and expr[j + 1] in '*+':
                    apply_quantifier(g, group_exit, group_entry, expr[j + 1], is_group=True)
                    i = j + 1
                else:
                    i = j
                prev = group_exit
            elif c.isalpha():
                char_node = self.new_node(c)
                g.edge(prev, char_node)

                if i + 1 < len(expr) and expr[i + 1] in '*+':
                    apply_quantifier(g, char_node, prev, expr[i + 1], is_group=False)
                    i += 1
                prev = char_node
            i += 1
        g.edge(prev, exit)
```

```
def logical_or(expr: str) -> list[str]:
    parts = []
    depth = 0
    current = ''
    for c in expr:
        if c == '(':
            depth += 1
        elif c == ')':
            depth -= 1
        if c == '|' and depth == 0:
            parts.append(current)
            current = ''
        else:
            current += c
    parts.append(current)
    return parts

def apply_quantifier(g: Digraph, current_node: str, prev_node: str, quantifier: str,
is_group: bool):
    if quantifier in ('*', '+'):
        if is_group:
            g.edge(current_node, prev_node, label=quantifier, constraint='false')
        else:
            g.edge(current_node, current_node, label=quantifier, constraint='false')

def user_input():
    regex_input = input("Enter a simple regular expression (supports a-z, *, +, |, () ): ")
    if not re.match(RegexperStyleDiagram.REGEX_PATTERN, regex_input):
        print("Error: Invalid characters detected!")
        return user_input()

    try:
        diagram = RegexperStyleDiagram()
        return diagram.build(regex_input)
    except Exception as e:
        print("Error during processing:", e)
        return user_input()
```

```
def create_pdf(graph):
    output_path = '/mnt/data/user_regexper_diagram'
    graph.render(output_path, view=True, format='pdf')
    print(f"Diagram saved to: {output_path}.pdf")


if __name__ == "__main__":
    graph = user_input()
    create_pdf(graph)
```

Използвана литература:

[1] - https://to6esko.github.io/09_regexp.html

[2] - https://pyfmi.org/media/materials/14._%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D0%B8_%D0%B8%D0%B7%D1%80%D0%B0%D0%B7%D0%B8.pdf

[3] - https://regexper.com/documentation.html

[4] - https://schupen.net/lib/tu/KST_all/Semesters/semestar%206/FEEP/FEEPLecture3_RE.pdf