Slide 11 - Conjunto de Cadeias

Questão 1

Seja $\Sigma = \{'a', 'b'\}$ um alfabeto. Escreva um programa em Python para gerar o conjunto Σ^n, ou seja, que contém todas as possíveis cadeias de caracteres formadas pelos elementos de Σ de comprimento n

```
# Definindo o alfabeto
sigma = ['a', 'b']

# Função para gerar cadeias do alfabeto de um comprimento específico
def generate_specific_chains(alfabeto, n):
    if n == 0:
        return ['']
    if n == 1:
        return alfabeto

    # Iniciar com cadeias de comprimento 1
    prev_chains = alfabeto.copy()
    for _ in range(1, n): # 2
        new_chains = []
        # []
        for chain in prev_chains: #[a, b]
            # a
            for character in alfabeto: # [a, b]
                # a
                new_chains.append(chain + character)
        prev_chains = new_chains

    return new_chains

# Exemplo de uso da função para gerar Sigma^2
n = 3
sigma = generate_specific_chains(sigma, n)
print(f"Sigma^{n} =")
print(sigma)
```

```
Sigma^3 =
['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']
```

Explicação

- A função `generate_specific_chains` cria cadeias de comprimento `n` usando o alfabeto fornecido.

- Começa com o alfabeto como as cadeias de comprimento `1` e, em seguida, constrói cadeias de comprimento maior ao concatenar cada caractere do alfabeto a cada cadeia existente.
- Este processo é repetido `n` vezes para alcançar o comprimento desejado.
- Para `n = 2` e o alfabeto `{'a', 'b'}`, a função produzirá as cadeias `'aa'`, `'ab'`, `'ba'`, e `'bb'`.

Questão 2

Seja $\Sigma = \{'a', 'b'\}$ um alfabeto. Escreva um programa em Python para gerar o conjunto Σ^* até Σ^3, que contém todas as possíveis cadeias de caracteres formadas pelos elementos de Σ até o comprimento 3.

Resolução 1

Podemos chamar a função acima para cada sigma e irmos armazenando os resultados em um dicionário. Ao final, imprimir o dicionário.

```python
# Definindo o alfabeto
sigma = ['a', 'b']

# Função para gerar cadeias de um comprimento específico
def generate_specific_chains(alfabeto, n):
    if n == 0:
        return ['']
    if n == 1:
        return alfabeto.copy()

    prev_chains = alfabeto.copy()
    for _ in range(1, n):
        new_chains = []
        for chain in prev_chains:
            for character in alfabeto:
                new_chains.append(chain + character)
        prev_chains = new_chains

    return new_chains

# Gerar e exibir os resultados de Sigma^1 a Sigma^3
sigma_dict = {}

for i in range(0, 4):  # de 0 até 3
    cadeias = generate_specific_chains(sigma, i)
    sigma_dict[i] = cadeias
    if i > 0:
        print(f"Sigma^{i} =", cadeias)

# Exibir o dicionário completo incluindo Sigma^0
print(sigma_dict)
```

```
Sigma^1 = ['a', 'b']
Sigma^2 = ['aa', 'ab', 'ba', 'bb']
Sigma^3 = ['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']
{0: [''], 1: ['a', 'b'], 2: ['aa', 'ab', 'ba', 'bb'], 3: ['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']}
```

Resolução 2

Podemos resolver usando a função `itertools.product()` do python. `itertools.product` é uma função da biblioteca itertools no Python que é usada para realizar o `produto cartesiano` entre iteráveis. Quando você passa um `iterável` (como uma lista) e um `argumento repeat=n`, itertools.product gera todas as combinações possíveis dos elementos do iterável com eles mesmos, repetidos `n` vezes. Cada combinação é retornada como uma tupla de elementos.

```python
import itertools

# Definindo o alfabeto
sigma = ['a', 'b']

# Função para gerar cadeias do alfabeto até um comprimento máximo
def generate_chains(alfabeto, max_length):
    chains = ['']  # Inicia com a cadeia vazia (para Sigma^0)
    for length in range(1, max_length + 1):
        for element in itertools.product(alfabeto, repeat=length):
            chains.append(''.join(element))
    return chains

# Gerando e exibindo cadeias até o comprimento 3
sigma_star = generate_chains(sigma, 3)
print("Cadeias de caracteres formadas pelo alfabeto até o comprimento 3:")
for chain in sigma_star:
    print(chain)
```

```
Cadeias de caracteres formadas pelo alfabeto até o comprimento 3:

a
b
aa
ab
ba
bb
aaa
aab
aba
abb
baa
bab
bba
bbb
```

O que itertools.product faz? Por exemplo, se você tem um alfabeto $\Sigma = \{a, b\}$ e usa itertools.product(sigma, repeat=2), o resultado será todas as possíveis combinações de 2 elementos, que são:

('a', 'a') ('a', 'b') ('b', 'a') ('b', 'b') Isso é equivalente ao produto cartesiano $\Sigma \times \Sigma$.

Resolução 3

Podemos fazer um código que faça uma geração direta de todas as cadeias. Mas essa resolução adiciona mais complexidade.

Comece a programar ou gere código com IA.

```
Início: Sigma^0 = {''}

Gerando Sigma^1 (comprimento 1):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Sigma^1 = ['a', 'b']

Gerando Sigma^2 (comprimento 2):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Adicionando 'aa' à lista de cadeias
Adicionando 'ab' à lista de cadeias
Adicionando 'ba' à lista de cadeias
Adicionando 'bb' à lista de cadeias
Sigma^2 = ['a', 'b', 'aa', 'ab', 'ba', 'bb']

Gerando Sigma^3 (comprimento 3):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Adicionando 'aa' à lista de cadeias
Adicionando 'ab' à lista de cadeias
Adicionando 'ba' à lista de cadeias
Adicionando 'bb' à lista de cadeias
Adicionando 'aa' à lista de cadeias
Adicionando 'ab' à lista de cadeias
Adicionando 'ba' à lista de cadeias
Adicionando 'bb' à lista de cadeias
Adicionando 'aaa' à lista de cadeias
Adicionando 'aab' à lista de cadeias
Adicionando 'aba' à lista de cadeias
Adicionando 'abb' à lista de cadeias
Adicionando 'baa' à lista de cadeias
Adicionando 'bab' à lista de cadeias
Adicionando 'bba' à lista de cadeias
Adicionando 'bbb' à lista de cadeias
Sigma^3 = ['a', 'b', 'aa', 'ab', 'ba', 'bb', 'aa', 'ab', 'ba', 'bb', 'aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']

Cadeias de caracteres formadas pelo alfabeto até o comprimento 3 (Sigma^*):
['', 'a', 'b', 'a', 'b', 'aa', 'ab', 'ba', 'bb', 'a', 'b', 'aa', 'ab', 'ba', 'bb', 'aa', 'ab', 'ba', 'bb', 'aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']
```

Parece que há uma confusão está no loop que gera as novas cadeias, pois algumas cadeias estão sendo repetidas. Dessa forma, precisamos garantir que cada conjunto de cadeias de um determinado comprimento seja gerado corretamente antes de prosseguir para o próximo comprimento. Vamos ajustar o código para corrigir isso:

Comece a programar ou gere código com IA.

```
Início: Sigma^0 = {''}

Gerando Sigma^1 (comprimento 1):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Sigma^1 = ['a', 'b']

Gerando Sigma^2 (comprimento 2):
Adicionando 'aa' à lista de cadeias
Adicionando 'ab' à lista de cadeias
Adicionando 'ba' à lista de cadeias
Adicionando 'bb' à lista de cadeias
Sigma^2 = ['aa', 'ab', 'ba', 'bb']

Gerando Sigma^3 (comprimento 3):
Adicionando 'aaa' à lista de cadeias
Adicionando 'aab' à lista de cadeias
Adicionando 'aba' à lista de cadeias
Adicionando 'abb' à lista de cadeias
Adicionando 'baa' à lista de cadeias
Adicionando 'bab' à lista de cadeias
Adicionando 'bba' à lista de cadeias
Adicionando 'bbb' à lista de cadeias
Sigma^3 = ['aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']

Cadeias de caracteres formadas pelo alfabeto até o comprimento 3 (Sigma^*):
['', 'a', 'b', 'aa', 'ab', 'ba', 'bb', 'aaa', 'aab', 'aba', 'abb', 'baa', 'bab', 'bba', 'bbb']
```

Neste código, ajustamos a lógica para que previous_chains armazene as cadeias do comprimento imediatamente anterior, evitando a repetição de cadeias ao gerar as de comprimento superior. Agora, para cada novo comprimento, o programa concatena caracteres apenas às cadeias do comprimento anterior, o que deve eliminar as repetições e erros na geração das cadeias. Podemos ainda usar uma estrutura de dados chamada `set`. Essa estrutura de dados evita de guardarmos elementos repetidos, mesmo que tentemos fazer. Veja como ficaria simples.

Comece a programar ou gere código com IA.

```
Início: Sigma^0 = {''}

Gerando Sigma^1 (comprimento 1):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Sigma^1 = ['a', 'b']
```

```
Gerando Sigma^2 (comprimento 2):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Adicionando 'aa' à lista de cadeias
Adicionando 'ab' à lista de cadeias
Adicionando 'ba' à lista de cadeias
Adicionando 'bb' à lista de cadeias
Sigma^2 = ['bb', 'ba', 'aa', 'b', 'ab', 'a']

Gerando Sigma^3 (comprimento 3):
Adicionando 'a' à lista de cadeias
Adicionando 'b' à lista de cadeias
Adicionando 'bba' à lista de cadeias
Adicionando 'bbb' à lista de cadeias
Adicionando 'baa' à lista de cadeias
Adicionando 'bab' à lista de cadeias
Adicionando 'aaa' à lista de cadeias
Adicionando 'aab' à lista de cadeias
Adicionando 'ba' à lista de cadeias
Adicionando 'bb' à lista de cadeias
Adicionando 'aba' à lista de cadeias
Adicionando 'abb' à lista de cadeias
Adicionando 'aa' à lista de cadeias
Adicionando 'ab' à lista de cadeias
Sigma^3 = ['baa', 'bb', 'abb', 'ba', 'bba', 'aa', 'b', 'aab', 'bbb', 'ab', 'aba', 'aaa', 'bab', 'a']

Cadeias de caracteres formadas pelo alfabeto até o comprimento 3 (Sigma^*):
['', 'a', 'aa', 'aaa', 'aab', 'ab', 'aba', 'abb', 'b', 'ba', 'baa', 'bab', 'bb', 'bba', 'bbb']
```

Slide 18 - Operações e Propriedades

Questão 3

Considere os conjuntos de strings $L = \{"001", "10", "111"\}$ e $M = \{\epsilon, "001"\}$, onde ϵ representa a string vazia. Escreva um programa em Python que calcule a concatenação de L e M, representada por LM. LM é o conjunto de todas as strings formadas pela concatenação de cada elemento em L com cada elemento em M.

Resolução 1

Usando `set`

```
Comece a programar ou gere código com IA.
```

```
LM = {'10001', '001', '10', '111', '001001', '111001'}
```

Explicação

- `L` e `M` são definidos como conjuntos de strings.
- A função `concatenate_sets` recebe dois conjuntos, `set1` e `set2`, e itera sobre cada elemento de set1 e set2 para concatenar os elementos correspondentes, formando assim um novo conjunto que contém todos os resultados da concatenação.
- A concatenação de `L` e `M (LM)` é então calculada usando essa função.
- O resultado, `LM`, é impresso e deve conter todas as strings resultantes da concatenação de cada string em `L` com cada string em `M`.
- No exemplo dado, `LM` incluirá as strings `"001", "001001", "10", "10001", "111", e "111001"`, correspondendo à concatenação de todos os elementos de `L` com todos os elementos de `M`.

Resolução 2

Usando `itertools.product` para realizar o produto cartesiano

```
Comece a programar ou gere código com IA.
```

```
LM = {'10001', '001', '10', '111', '001001', '111001'}
```

- **`L` e `M`** são definidos como conjuntos de strings, com `M` contendo a string vazia ϵ e "001".
- Utilizamos a função `product` do módulo `itertools` para criar o produto cartesiano de `L` e `M`. Isso nos dá todos os pares possíveis de elementos entre `L` e `M`.
- A compreensão de conjunto `{''.join(pair) for pair in itertools.product(L, M)}` concatena cada par de strings formando o conjunto `LM`.
- `LM` contém todas as strings resultantes da concatenação de elementos de `L` com elementos de `M`, como "001", "001001", "10", "10001", "111", e "111001".

Slide 19 - Fechamento reflexivo e transitivo

Linguagem

O **fechamento reflexivo e transitivo** é um conceito da teoria de linguagens formais e autômatos, geralmente aplicado a conjuntos de cadeias ou relações. Ou seja, dada uma linguagem L, é basicamente uma maneira de obter todas as possíveis cadeias (sequências de símbolos) que você pode formar usando as cadeias originais em L, quantas vezes quiser, incluindo a possibilidade de não usar nenhuma cadeia (que é representada pela cadeia vazia ϵ). Vamos explorar esse conceito em detalhes e compará-lo com a reflexividade e transitividade em funções ou relações.

Fechamento Reflexivo

- **Linguagens Formais**: No contexto de linguagens formais, o fechamento reflexivo de uma linguagem L permite a inclusão da cadeia vazia ϵ. Isso significa que, independentemente de não selecionar nenhum elemento de L, a cadeia vazia é considerada uma

concatenação válida de zero elementos de L.
- **Relações**: Em teoria de relações, uma relação é considerada reflexiva se cada elemento estiver relacionado a si mesmo. Por exemplo, na relação de igualdade, cada número é igual a ele mesmo.

Fechamento Transitivo

- **Linguagens Formais**: O fechamento transitivo de uma linguagem L envolve criar novas cadeias por meio da concatenação repetida de elementos de L. Se $L = \{a, b\}$, então L^* (o fechamento transitivo de L) incluiria cadeias como aa, ab, ba, bb, aaa, e assim por diante, abrangendo todas as concatenações possíveis dos elementos de L.
- **Relações**: Na teoria de relações, transitividade implica que se um elemento a está relacionado a um elemento b, e b está relacionado a um elemento c, então a deve estar relacionado a c. Isso é diferente da concatenação em linguagens, que se concentra na combinação de cadeias.

Diferenças e Semelhanças

- **Operação vs. Propriedade**: O fechamento reflexivo e transitivo em linguagens é uma operação que gera um conjunto novo a partir do original, enquanto reflexividade e transitividade em relações são propriedades que descrevem como os elementos de um conjunto estão interconectados.
- **Resultado vs. Condição**: No fechamento reflexivo e transitivo, estamos interessados no resultado (o conjunto de todas as cadeias possíveis), enquanto em reflexividade e transitividade, o foco está em se a relação satisfaz certas condições.
- **Completude**: Ambos os conceitos lidam com a ideia de completude. No fechamento reflexivo e transitivo, trata-se da completude das cadeias que podem ser formadas; nas relações reflexivas e transitivas, é sobre a completude das conexões dentro do conjunto.

Essas explicações ilustram como o fechamento reflexivo e transitivo expande um conjunto de cadeias em linguagens formais, enquanto a reflexividade e a transitividade em relações descrevem as conexões dentro de um conjunto.

Se você tem um conjunto L com algumas cadeias de caracteres, então $L*$ incluirá:

- A cadeia vazia ϵ
- Todas as cadeias em L (isso é L^1)
- Todas as cadeias que podem ser formadas concatenando duas cadeias em L (isso é L^2) E assim por diante, para 3 cadeias, 4 cadeias, etc.

Questão 4

Seja L o conjunto de todas as cadeias formadas apenas pelo símbolo '0'. Calcule L^*, o fechamento reflexivo e transitivo de L, o que representa todas as cadeias que podem ser formadas concatenando zero ou mais vezes as cadeias em L.

```
# Definindo a linguagem L
L = {'aba', 'bab'}

# Função para calcular o fechamento reflexivo e transitivo de L
def reflexive_transitive_closure(language, max_length=5):
```

```
    closure = {''}
    print("Início: L^0 = {''}")

    for i in range(1, max_length + 1):

        if not new_elements:
            print(f"Nenhum novo elemento adicionado para L^{i}")
            break


        print(f"L^{i} = {sorted(closure)}")

    return closure
L_star = reflexive_transitive_closure(L)
print("\nL* =", sorted(L_star))

"""
Saida esperada:
Início: L^0 = {''}
Adicionando 'bab' à L^1
Adicionando 'aba' à L^1
L^1 = ['', 'aba', 'bab']
Adicionando 'ababab' à L^2
Adicionando 'abaaba' à L^2
Adicionando 'babbab' à L^2
Adicionando 'bababa' à L^2
L^2 = ['', 'aba', 'abaaba', 'ababab', 'bab', 'bababa', 'babbab']
Adicionando 'babbabbab' à L^3
Adicionando 'babbababa' à L^3
Adicionando 'abaababab' à L^3
Adicionando 'abaabaaba' à L^3
Adicionando 'babababab' à L^3
Adicionando 'bababaaba' à L^3
Adicionando 'abababbab' à L^3
Adicionando 'ababababa' à L^3
L^3 = ['', 'aba', 'abaaba', 'abaabaaba', 'abaababab', 'ababab', 'ababababa', 'abababbab', 'bab', 'bababa', 'bababaaba', 'babababab', 'babbab', 'babbababa', 'babbabbab']
Adicionando 'abaabaababab' à L^4
Adicionando 'abaabaabaaba' à L^4
Adicionando 'abaabababbab' à L^4
Adicionando 'abaababababa' à L^4
Adicionando 'babbabababab' à L^4
Adicionando 'babbababaaba' à L^4
Adicionando 'bababaababab' à L^4
Adicionando 'bababaabaaba' à L^4
Adicionando 'babababab bab' à L^4
Adicionando 'bababababababa' à L^4
Adicionando 'abababababab' à L^4
Adicionando 'ababababaaba' à L^4
Adicionando 'babbabbabbab' à L^4
Adicionando 'babbabbababa' à L^4
Adicionando 'abababbabbab' à L^4
```

```
Adicionando 'abababbababa' à L^4
L^4 = ['', 'aba', 'abaaba', 'abaabaaba', 'abaabaabaaba', 'abaabaababab', 'abaababab', 'abaabababaaba', 'abaabababbab', 'ababab', 'ababababa', 'ababababaaba', 'abababababab', 'ababa
Adicionando 'babbabbabbabbab' à L^5
Adicionando 'babbabbabbababa' à L^5
Adicionando 'abaabaabababbab' à L^5
Adicionando 'abaabaababababa' à L^5
Adicionando 'babbabbabababab' à L^5
Adicionando 'babbabbababaaba' à L^5
Adicionando 'bababaabaababab' à L^5
Adicionando 'bababaabaabaaba' à L^5
Adicionando 'abababbabababab' à L^5
Adicionando 'abababbababaaba' à L^5
Adicionando 'abaabaabaababab' à L^5
Adicionando 'abaabaabaabaaba' à L^5
Adicionando 'abaababababab' à L^5
Adicionando 'abaabababaaba' à L^5
Adicionando 'ababababababbab' à L^5
Adicionando 'abababababababa' à L^5
Adicionando 'babababbabbab' à L^5
Adicionando 'bababababbababa' à L^5
Adicionando 'bababaabababbab' à L^5
Adicionando 'bababaababababa' à L^5
Adicionando 'babbababaababab' à L^5
Adicionando 'babbababaabaaba' à L^5
Adicionando 'ababababaababab' à L^5
Adicionando 'ababababaabaaba' à L^5
Adicionando 'babbabababbab' à L^5
Adicionando 'babbababababa' à L^5
Adicionando 'babababababab' à L^5
Adicionando 'bababababaaba' à L^5
Adicionando 'abababbabbabbab' à L^5
Adicionando 'abababbabbababa' à L^5
Adicionando 'abaabababbabbab' à L^5
Adicionando 'abaabababbababa' à L^5
L^5 = ['', 'aba', 'abaaba', 'abaabaaba', 'abaabaabaaba', 'abaabaabaabaaba', 'abaabaabaababab', 'abaabaababab', 'abaabaababababa', 'abaabaabababbab', 'abaababab', 'abaabababaaba', '

L* = ['', 'aba', 'abaaba', 'abaabaaba', 'abaabaabaaba', 'abaabaabaabaaba', 'abaabaabaababab', 'abaabaababab', 'abaabaababababa', 'abaabaabababbab', 'abaababab', 'abaabababaaba', 'a
"""
```

```
Início: L^0 = {''}
Adicionando 'bab' à L^1
Adicionando 'aba' à L^1
L^1 = ['', 'aba', 'bab']
Adicionando 'ababab' à L^2
Adicionando 'abaaba' à L^2
Adicionando 'babbab' à L^2
Adicionando 'bababa' à L^2
L^2 = ['', 'aba', 'abaaba', 'ababab', 'bab', 'bababa', 'babbab']
Adicionando 'babbabbab' à L^3
Adicionando 'babbababa' à L^3
Adicionando 'abaababab' à L^3
Adicionando 'abaabaaba' à L^3
```

```
Adicionando 'babababab' à L^3
Adicionando 'bababaaba' à L^3
Adicionando 'abababbab' à L^3
Adicionando 'ababababa' à L^3
L^3 = ['', 'aba', 'abaaba', 'abaabaaba', 'abaababab', 'ababab', 'ababababa', 'abababbab', 'bab', 'bababa', 'bababaaba', 'babababab', 'babbab', 'babbababa', 'babbabbab']
Adicionando 'abaabaababab' à L^4
Adicionando 'abaabaabaaba' à L^4
Adicionando 'abaababab bab' à L^4
Adicionando 'abaabababaaba' à L^4
Adicionando 'babbabababab' à L^4
Adicionando 'babbababaaba' à L^4
Adicionando 'bababaababab' à L^4
Adicionando 'bababaabaaba' à L^4
Adicionando 'babababab bab' à L^4
Adicionando 'bababababaaba' à L^4
Adicionando 'abababababab' à L^4
Adicionando 'ababababaaba' à L^4
Adicionando 'babbabbabbab' à L^4
Adicionando 'babbabbababa' à L^4
Adicionando 'abababbabbab' à L^4
Adicionando 'abababbababa' à L^4
L^4 = ['', 'aba', 'abaaba', 'abaabaaba', 'abaabaabaaba', 'abaabaababab', 'abaababab', 'abaabababaaba', 'abaababab bab', 'ababab', 'ababababa', 'ababababaaba', 'abababababab',
Adicionando 'babbabbabbabbab' à L^5
Adicionando 'babbabbabbababa' à L^5
Adicionando 'abaabaabababbab' à L^5
Adicionando 'abaabaababababa' à L^5
Adicionando 'babbabbabababab' à L^5
Adicionando 'babbabbababaaba' à L^5
Adicionando 'bababaabaababab' à L^5
Adicionando 'bababaabaabaaba' à L^5
Adicionando 'abababbabababab' à L^5
Adicionando 'abababbababaaba' à L^5
Adicionando 'abaabaabaababab' à L^5
Adicionando 'abaabaabaabaaba' à L^5
Adicionando 'abaabababababab' à L^5
Adicionando 'abaababababaaba' à L^5
Adicionando 'abababababab bab' à L^5
Adicionando 'ababababababa ba' à L^5
Adicionando 'babababab babbab' à L^5
Adicionando 'babababab bababa' à L^5
Adicionando 'bababaabababbab' à L^5
Adicionando 'bababaababababa' à L^5
Adicionando 'babbababaababab' à L^5
Adicionando 'babbababaabaaba' à L^5
```

Alfabeto

- **Conjunto Vazio** $\varnothing$:
  - O conjunto vazio $\varnothing$ contém zero cadeias e representa a menor linguagem possível definida sobre um alfabeto Σ. Ele é crucial em teoria da computação porque serve como a base para construir linguagens mais complexas, representando a ideia de "nada" ou

"ausência" em termos de cadeias de caracteres.

- **Fechamento Reflexivo e Transitivo** Σ^*:
  - Σ^* é o conjunto de todas as possíveis cadeias que podem ser formadas com os elementos de Σ, incluindo a cadeia vazia ϵ. Este conjunto é a maior de todas as linguagens que se pode definir sobre Σ, pois inclui todas as combinações possíveis de elementos em Σ em qualquer comprimento, começando do zero (cadeia vazia).
- **Conjunto Potência** 2^{Σ^*}:
  - 2^{Σ^*} representa o conjunto de todos os subconjuntos possíveis formados a partir de Σ^*, incluindo o próprio Σ^* e o conjunto vazio $\varnothing$. Ele corresponde ao conjunto de todas as possíveis linguagens que podem ser definidas sobre Σ, abrangendo todas as variações e combinações de cadeias possíveis.
  - É interessante notar que $\varnothing$ e Σ^* são elementos de 2^{Σ^*}, evidenciando a abrangência deste conjunto, que vai da menor à maior linguagem possível sobre o alfabeto Σ.

Essas observações realçam a estrutura hierárquica e inclusiva dos conjuntos em teoria das linguagens formais, desde o conjunto mais simples $\varnothing$ até o conjunto potência 2^{Σ^*}, que encapsula todas as linguagens possíveis.

Considere o alfabeto $\Sigma = \{a, b, c\}$ e uma propriedade P que define que todas as cadeias devem iniciar com o símbolo 'a'. Vamos explorar algumas linguagens derivadas de Σ e como elas se relacionam com a propriedade P:

- **Linguagem** L_0:
  - Definida como $L_0 = \varnothing$, é a menor linguagem possível sobre Σ. Ela não contém nenhuma cadeia e, portanto, não contribui para a propriedade P.
- **Linguagem** L_1:
  - Contém cadeias que começam com 'a' e segue a propriedade P: $L_1 = \{a, ab, ac, abc, acb\}$. Esta é uma linguagem finita e satisfaz a condição de iniciar todas as cadeias com 'a'.
- **Linguagem** L_2:
  - É uma linguagem infinita que também obedece a P, definida como $L_2 = \{a\}\{a\}^*\{b\}^*\{c\}^*$. Isso significa que L_2 começa com 'a', seguido por zero ou mais 'a's, 'b's e 'c's, em qualquer quantidade.
- **Linguagem** L_3:
  - A maior linguagem que observa P, definida por $L_3 = \{a\}\{a, b, c\}^*$, consiste em todas as cadeias que começam com 'a' seguido por qualquer combinação de 'a', 'b', e 'c'. Ela é infinita e engloba todas as cadeias que podem ser formadas segundo P.
- **Subconjuntos e Pertinência**:
  - Todas estas linguagens, L_0, L_1, L_2, e L_3, são subconjuntos de Σ^* e elementos do conjunto potência 2^{Σ^*}, que representa todas as possíveis linguagens definidas sobre Σ.
- **Diversidade de Linguagens**:

- Além de L_0, L_1, L_2 e L_3, existem muitas outras linguagens que podem ser construídas a partir de Σ seguindo diferentes regras ou propriedades.

Este exemplo demonstra a variedade e a complexidade das linguagens que podem ser definidas a partir de um alfabeto básico, dependendo das propriedades ou regras especificadas, desde o conjunto vazio até linguagens infinitamente expansíveis.

Questão 5 - Slide 26

Exemplo de Fechamento Transitivo

Explore o conceito de fechamento transitivo no alfabeto $\Sigma = \{n, (,), +, *, -, /\}$:

Resposta
O fechamento transitivo é uma operação fundamental em linguagens formais que gera um conjunto contendo todas as possíveis sequências (ou cadeias) de elementos de um alfabeto, concatenadas em qualquer quantidade, incluindo a cadeia vazia (ϵ) para o fechamento reflexivo. Vamos explorar esse conceito usando o alfabeto:

- Σ^* representa o conjunto de todas as possíveis cadeias formadas pelos símbolos em Σ, incluindo a cadeia vazia (ϵ). Exemplos de cadeias em Σ^* incluem "n", "n+n", "-n)", "*/", e "n())", representando a vasta gama de expressões que podem ser criadas.
- Σ^+ é similar ao Σ^*, mas não inclui a cadeia vazia (ϵ). Assim, contém cadeias como "n", "n+n", "-n)", "n()", e "n-(n*n)", refletindo todas as combinações possíveis de elementos em Σ com um ou mais símbolos.
- A relação $\Sigma^+ = \Sigma^* - \{\epsilon\}$ demonstra que Σ^+ pode ser derivado de Σ^* pela remoção da cadeia vazia, ressaltando a conexão entre esses dois conjuntos no contexto do fechamento transitivo.

Slide 28 - Reversão

Questão 6

Dada a linguagem $L_2 = \{\epsilon, a, ab, abc\}$, escreva uma função em Python que determine o reverso dessa linguagem, denotado por L_2^R. O reverso de uma linguagem é um conjunto de cadeias onde cada cadeia é o inverso das cadeias originais de L_2.

```
# Linguagem original
L2 = ['', 'a', 'ab', 'abc']

# Função para reverter uma linguagem
def reverso_linguagem(linguagem):
    return [cadeia[::-1] for cadeia in linguagem]

# Calcular o reverso de L2
LR2 = reverso_linguagem(L2)
```

```python
LR2 = reverse_language(L2)

# Exibir o resultado
print("L2 =", L2)
print("LR2 =", LR2)
```

```
L2 = ['', 'a', 'ab', 'abc']
LR2 = ['', 'a', 'ba', 'cba']
```

Explicação do Código

- A linguagem L_2 é inicialmente definida como um conjunto de cadeias: ϵ (cadeia vazia), 'a', 'ab', e 'abc'.
- A função `reverse_language` recebe a linguagem L_2 como argumento e utiliza a compreensão de conjunto para criar um novo conjunto. Para cada sentença em L_2, a sentença é invertida (`sentence[::-1]`) e adicionada ao novo conjunto.
- O resultado `L1` é o conjunto de todas as sentenças de L_2 invertidas, ou seja, L_2^R.
- Ao imprimir `L1`, obtemos o reverso de L_2, que neste caso será $\{\epsilon, a, ba, cba\}$.

Slide 29 - Propriedade de Prefixo e Sufixo Próprio

Prefixo Próprio: Uma cadeia α é um prefixo próprio de outra cadeia $\alpha\beta$ se β não é vazia (ϵ). Isso significa que α é o início de $\alpha\beta$, mas $\alpha\beta$ contém mais caracteres além de α. Em uma linguagem com a propriedade de prefixo próprio, nenhuma cadeia na linguagem é um prefixo próprio de outra cadeia dentro dessa mesma linguagem.

Sufixo Próprio: Similarmente, uma cadeia α é um sufixo próprio de outra cadeia $\beta\alpha$ se β não é vazia. Aqui, α aparece no final de $\beta\alpha$, e $\beta\alpha$ contém mais caracteres antes de α. Uma linguagem com a propriedade de sufixo próprio não tem nenhuma cadeia que seja sufixo próprio de outra cadeia na linguagem.

Dada a linguagem $L = \{a, ab, abc, bc, c\}$, escreva uma função em Python que determine se L tem a propriedade de prefixo próprio e sufixo próprio. Mostre os elementos que violam estas propriedades, se houver.

```python
L = {'a', 'ab', 'abc', 'bc', 'c'}

# Verificando a propriedade de prefixo próprio
def has_proper_prefix(language):
    """
    Verifica se a linguagem possui a propriedade de prefixo próprio.

    Um conjunto possui essa propriedade se nenhuma cadeia for prefixo próprio de outra.
    Imprima print(f"'{element}' é um prefixo de '{other}'")

    :param language: conjunto de cadeias (strings)
    :return: True se não há prefixo próprio, False caso contrário
    """
```

```python
    return True

# Verificando a propriedade de sufixo próprio
def has_proper_suffix(language):
    """
    Verifica se a linguagem possui a propriedade de sufixo próprio.

    Um conjunto possui essa propriedade se nenhuma cadeia for sufixo próprio de outra.
    Imprima print(f"'{element}' é um sufixo de '{other}'")

    :param language: conjunto de cadeias (strings)
    :return: True se não há sufixo próprio, False caso contrário
    """
    return True

print("L tem a propriedade de prefixo próprio?", has_proper_prefix(L))
print("L tem a propriedade de sufixo próprio?", has_proper_suffix(L))
```

```
'ab' é um prefixo próprio de 'abc'
L tem a propriedade de prefixo próprio? False
'bc' é um sufixo próprio de 'abc'
L tem a propriedade de sufixo próprio? False
```

Questão 7

Dada as linguagens abaixo disposta em um dicionário, escreva uma função em Python que determine se cada uma delas tem a propriedade de prefixo próprio e sufixo próprio. Mostre os elementos que violam estas propriedades, se houver.

```python
languages = [
    ('L1', {'prefix', 'pref', 'xfix'}),
    ('L2', {'prefix', 'pref', 'fix'}),
    ('L3', {'alpha', 'beta', 'gamma'}),
    ('L4', {"bat","sat","at"})
]

# Função para verificar a propriedade de prefixo próprio
def has_proper_prefix(language):
    # comment via docstring pep8
    """
    @brief Para cada elemento na linguagem, verifica se existe outro elemento que seja diferente dele mesmo e que começa com ele.
    Se encontrar, imprime a relação e retorna False.
    Se não encontrar, retorna True. Imprima dentro da função print(f"'{x}' é um prefixo próprio de '{y}'")

    :param language: Conjunto de cadeias da linguagem.
    :return: True se a linguagem não tem prefixos próprios, False caso contrário.
    """
    return True
```

```python
# Função para verificar a propriedade de sufixo próprio
def has_proper_suffix(language):
    # comment via docstring pep8
    """
    @brief Para cada elemento na linguagem, verifica se existe outro elemento que seja diferente dele mesmo e que termina com ele.
    Se encontrar, imprime a relação e retorna False.
    Se não encontrar, retorna True. Imprima dentro da função print(f"'{x}' é um sufixo próprio de '{y}'")

    :param language: Conjunto de cadeias da linguagem.
    :return: True se a linguagem não tem sufixos próprios, False caso contrário.
    """
    return True

# Testando as propriedades para cada linguagem
for name, lang in languages:
    print(f"\n{name} tem a propriedade de prefixo próprio? {has_proper_prefix(lang)}")
    print(f"{name} tem a propriedade de sufixo próprio? {has_proper_suffix(lang)}")

    prefix = has_proper_prefix(lang)
    suffix = has_proper_suffix(lang)

    if prefix and suffix:
        print(f"Portanto, {name} possui ambas as propriedades de prefixo e sufixo próprio.\n")
    elif prefix:
        print(f"Portanto, {name} tem somente a propriedade de prefixo próprio.\n")
    elif suffix:
        print(f"Portanto, {name} tem somente a propriedade de sufixo próprio.\n")
    else:
        print(f"Portanto, {name} não possui as propriedades de prefixo e sufixo próprio.\n")
# Exemplo de uso
```

```
'pref' é um prefixo próprio de 'prefix'
L1 tem a propriedade de prefixo próprio? False
L1 tem a propriedade de sufixo próprio? True
Portanto, L1 tem somente a propriedade de sufixo próprio.

'pref' é um prefixo próprio de 'prefix'
'fix' é um sufixo próprio de 'prefix'
L2 tem a propriedade de prefixo próprio? False
L2 tem a propriedade de sufixo próprio? False
Portanto, L2 não possui as propriedades de prefixo e sufixo próprio.

L3 tem a propriedade de prefixo próprio? True
L3 tem a propriedade de sufixo próprio? True
Portanto, L3 possui ambas as propriedades de prefixo e sufixo próprio.

'at' é um sufixo próprio de 'sat'
L4 tem a propriedade de prefixo próprio? True
L4 tem a propriedade de sufixo próprio? False
Portanto, L4 tem somente a propriedade de prefixo próprio.
```

Slide 31 - Quociente de Linguagens

Definição Sejam L_1 e L_2 duas linguagens. O quociente de L_1 por L_2, denotado por L_1/L_2, é definido como o conjunto de todas as cadeias x tal que a concatenação de x com qualquer cadeia y de L_2 resulta em uma cadeia que pertence a L_1.

Formulação Matemática

$$L_1/L_2 = \{x \mid xy \in L_1, y \in L_2\}$$

Exemplo Prático

Considere as linguagens:

- $L = \{a, aab, baa\}$
- $A = \{a\}$

Para calcular o quociente L/A, procuramos por cadeias em L que, ao serem concatenadas com 'a' (os elementos de A), ainda pertencem a L.

- Da cadeia 'a' em L, removendo 'a' de A, sobra o ϵ (cadeia vazia).
- Da cadeia 'aab' em L, removendo 'a' de A, sobra 'ab'. No entanto, 'ab' não é incluída porque não satisfaz a condição de formar uma cadeia em L ao ser concatenada com 'a'.
- Da cadeia 'baa' em L, removendo 'a' de A, sobra 'ba'.

Portanto, o quociente L/A é $\{\epsilon, ba\}$.

Questão 8 - Slide 32

Considere as linguagens seguintes:

- $L_1 = \{a^i b \mid i \geq 0\}$
- $L_2 = \{a^i bc^i \mid i \geq 0\}$
- $L_3 = \{b\}$
- $L_4 = \{a^i b \mid i \geq 1\}$
- $L_5 = \{bc^i \mid i \geq 0\}$
- $L_6 = \{c^i b \mid i \geq 0\}$
- $L_7 = \{a^i \mid i \geq 0\}$

Descrição das Linguagens

- $L_1 = \{a^i b \mid i \geq 0\}$

Esta linguagem consiste em cadeias formadas pela repetição de 'a' zero ou mais vezes seguida por um 'b'. Isso inclui 'b' (quando $i = 0$), 'ab' ($i = 1$), 'aab' ($i = 2$), e assim por diante.

- $L_2 = \{a^i bc^i \mid i \geq 0\}$

  Aqui, as cadeias começam e terminam com o mesmo número de 'a's e 'c's respectivamente, com um 'b' no meio. Exemplos incluem 'b' ($i = 0$), 'abc' ($i = 1$), 'aabcc' ($i = 2$), etc.

- $L_3 = \{b\}$

  Esta linguagem é a mais simples e contém uma única cadeia: 'b'.

- $L_4 = \{a^i b \mid i \geq 1\}$

  Similar a L_1, mas aqui temos pelo menos um 'a' antes do 'b', ou seja, não inclui apenas 'b'. Exemplos são 'ab' ($i = 1$), 'aab' ($i = 2$), etc.

- $L_5 = \{bc^i \mid i \geq 0\}$

  Consiste em cadeias que começam com 'b' seguidas por zero ou mais 'c's. Inclui 'b' ($i = 0$), 'bc' ($i = 1$), 'bcc' ($i = 2$), e assim por diante.

- $L_6 = \{c^i b \mid i \geq 0\}$

  Similar a L_5, mas aqui 'c's precedem o 'b'. Isso inclui 'b' ($i = 0$), 'cb' ($i = 1$), 'ccb' ($i = 2$), etc.

- $L_7 = \{a^i \mid i \geq 0\}$

  Esta linguagem inclui somente cadeias de 'a's de qualquer comprimento, incluindo a cadeia vazia ϵ quando $i = 0$, 'a' ($i = 1$), 'aa' ($i = 2$), e assim por diante.

Quocientes de Linguagens: Exemplos Detalhados

1. $L_1 / L_3 = ?$

   - $L_1 = \{a^i b \mid i \geq 0\}$ inclui cadeias como 'b', 'ab', 'aab', 'aaab', etc.
   - $L_3 = \{b\}$ contém apenas a cadeia 'b'.

   O quociente L_1 / L_3 procura cadeias em L_1 que, ao serem concatenadas com 'b' de L_3, resultam em cadeias ainda em L_1. Portanto, o resultado é composto por cadeias de 'a's de qualquer comprimento.

   Resposta: L_1 / L_3 são todas as cadeias formadas por 'a', que é $L_7 = \{a^i \mid i \geq 0\}$.

2. $L_1 / L_4 = ?$ Considere as seguintes linguagens:

   - $L_1 = a^i b \mid i \geq 0$: Contém cadeias como 'b', 'ab', 'aab', etc., onde 'b' é precedido por zero ou mais 'a's.
   - $L_4 = a^i b \mid i \geq 1$: Similar a L_1, mas começa com pelo menos um 'a', então não inclui 'b' sozinha. Contém 'ab', 'aab', 'aaab', etc.
     - O Quociente L_1 / L_4

       Ao calcular L_1 / L_4, procuramos por cadeias em L_1 que, ao serem concatenadas com qualquer cadeia de L_4, resultem em uma cadeia que ainda pertence a L_1.

- **Exemplos Detalhados para L_1/L_4:**
    - Cadeia 'b' em L_1: Não pode ser formada removendo algo de L_4.
    - Cadeia 'ab' em L_1: Forma ϵ ao remover 'ab', então $\epsilon \in L_1/L_4$.
    - Cadeia 'aab' em L_1: Ao remover 'ab', restam 'a', então 'a $\in L_1/L_4$.
    - Cadeia 'aaab' em L_1: Removendo 'ab', restam 'aa', então 'aa' está em L_1/L_4.
    - Cadeia 'aaaab' em L_1: Removendo 'ab', restam 'aaa', assim 'aaa' está em L_1/L_4.
    - Continuando assim, todas as cadeias de 'a's de qualquer comprimento estão em L_1/L_4.

  Portanto, L_1/L_4 é composto por todas as cadeias de 'a's, ou seja, é igual a L_7.

- *Conclusão:**
- Cada cadeia em L_1/L_4 é formada pela remoção de 'ab' das cadeias em L_1, resultando em cadeias compostas apenas por 'a's.
- Portanto, $L_1/L_4 = L_7 = a^i \mid i \geq 0$, que representa todas as possíveis sequências de 'a's, incluindo a cadeia vazia.

3. $L_5/L_7 =?$
    - $L_5 = \{bc^i \mid i \geq 0\}$ começa com 'b' seguido por zero ou mais 'c's. (b, bc, bcc, bccc)
    - $L_7 = \{a^i \mid i \geq 0\}$ inclui somente cadeias de 'a's de qualquer comprimento (ϵ, a, aa, aaa...)

   Para L_5/L_7, como L_7 contém apenas cadeias de 'a's, e nenhuma cadeia em L_5 pode resultar em uma cadeia em L_5 ao ser concatenada com 'a's, o resultado é o conjunto vazio.

   Resposta: L_5/L_7 é $\varnothing$, pois não há como obter cadeias de L_5 ao concatenar com cadeias de L_7.

4. $L_2/L_6 =?$
    - $L_2 = \{a^i bc^i \mid i \geq 0\}$ onde o número de 'a's e 'c's é o mesmo, envolvendo um 'b'.
      (b, abc, aabcc, aaabccc, ...)
    - $L_6 = \{c^i b \mid i \geq 0\}$ contém cadeias que começam com zero ou mais 'c's seguidos por 'b'.
      (b, cb, ccb, cccb, ...)

   Processo para encontrar L_2/L_6

   Ao tentar formar o quociente L_2/L_6, encontramos dificuldades:

    - As cadeias em L_2 são estruturadas de forma que 'b' esteja sempre entre o mesmo número de 'a's e 'c's.
    - As cadeias em L_6 não se encaixam como sufixos diretos em L_2 porque os 'c's em L_2 estão sempre precedidos por 'b'.
    - Portanto, não há uma correspondência direta que permita remover um sufixo de L_6 de uma cadeia em L_2 mantendo a cadeia resultante dentro de L_2.
    - *Conclusão:**
    - Nenhuma cadeia em L_6 se ajusta perfeitamente como um sufixo removível das cadeias em L_2 devido à estrutura específica de L_2 que requer um número igual de 'a's e 'c's separados por um único 'b'.

- Consequentemente, não é possível formar uma cadeia em L_2 removendo partes que são consistentes com L_6.
- Dado que não podemos satisfazer a condição para o quociente L_2/L_6 com as cadeias fornecidas em ambas as linguagens, o resultado é o conjunto vazio $\varnothing$.
- Assim, temos $L_2/L_6 = \varnothing$, indicando que não existem cadeias em L_2 das quais podemos remover um sufixo de L_6 e ainda termos uma cadeia que pertença a L_2.

🧠 Exercício: Quociente de Linguagens

Objetivo

Implementar a operação de **quociente de linguagens**, definida formalmente como:

$$L_1/L_2 = \{x \mid \exists y \in L_2,\ xy \in L_1\}$$

Ou seja, o conjunto de todas as cadeias `x` tal que existe uma cadeia `y ∈ L2` com `x + y ∈ L1`.

O que você deve fazer

1. Implementar uma função auxiliar `is_suffix(suffix, word)` que:
   - Verifica se `suffix` é sufixo de `word`;
   - Retorna o prefixo restante (`x`) se for;
   - Caso contrário, retorna `None`.
2. Usar essa função dentro da implementação de `quociente(L1, L2)`.
3. Validar o funcionamento com testes automatizados utilizando `unittest`.

```python
def is_suffix(suffix, word):
    """
    Verifica se 'suffix' é um sufixo de 'word'.
    Se for, retorna o prefixo restante (word - suffix).
    Caso contrário, retorna None.
    """
    return None

def quociente(L1, L2):
    """
    Calcula L1 / L2 = { x | existe y ∈ L2 tal que xy = w ∈ L1 },
    usando is_suffix para extrair x de forma direta.
    """
    resultado = set()
    return resultado
```

Testes Automatizados

```python
import unittest

class TestQuociente(unittest.TestCase):

    def test_exemplo_classico(self):
        L1 = {'a', 'aab', 'baa'}
        L2 = {'a'}
        esperado = {'ε', 'ba'}
        self.assertEqual(quociente(L1, L2), esperado)

    def test_sufixo_vazio(self):
        L1 = {'a', 'b'}
        L2 = {''}
        esperado = {'a', 'b'}
        self.assertEqual(quociente(L1, L2), esperado)

    def test_sem_sufixo_em_comum(self):
        L1 = {'abc', 'def'}
        L2 = {'x', 'y'}
        esperado = set()
        self.assertEqual(quociente(L1, L2), esperado)

    def test_multiplos_sufixos_validos(self):
        L1 = {'banana', 'bandana', 'ana'}
        L2 = {'ana'}
        esperado = {'ban', 'band', 'ε'}
        self.assertEqual(quociente(L1, L2), esperado)
```

```python
    def test_varias_possibilidades(self):
        L1 = {'hello', 'hell', 'low', 'yellow'}
        L2 = {'lo', 'low'}
        esperado = {'hel', 'yel', 'ε'}
        self.assertEqual(quociente(L1, L2), esperado)

unittest.main(argv=[''], exit=False)
```

```
.........
----------------------------------------------------------------------
Ran 9 tests in 0.007s

OK
<unittest.main.TestProgram at 0x1ef06268b80>
```

Slide 33 - Substituição

A **substituição** é uma operação fundamental em muitos aspectos da computação e análise de linguagens formais onde você associa cada símbolo de um alfabeto a um conjunto de cadeias de outro alfabeto. Pode-se pensar nisso como uma regra que diz como cada letra de um alfabeto pode ser transformada em palavras de outro alfabeto. Ou seja, é uma maneira de transformar cadeias de um alfabeto em cadeias de outro alfabeto, seguindo um conjunto de regras predefinidas.

Definição Formal Se temos dois alfabetos:

- Σ_1: O alfabeto de origem.
- Σ_2: O alfabeto de destino.

Uma substituição s é uma função que para cada símbolo em Σ_1, associa um conjunto de cadeias (palavras) formadas pelos símbolos em Σ_2. Em termos matemáticos, a substituição é descrita como:

$$s : \Sigma_1 \to 2^{\Sigma_2^*}$$

onde $2^{\Sigma_2^*}$ representa o conjunto de todos os subconjuntos de cadeias possíveis em Σ_2.

Exemplo Prático Considerando:

- $\Sigma_1 = \{a, b, c\}$
- $\Sigma_2 = \{x, y, z\}$

Podemos definir uma substituição s como:

- $s(a) = \{x\}$: O símbolo 'a' é substituído por 'x'.
- $s(b) = \{y, yy\}$: O símbolo 'b' pode ser substituído por 'y' ou 'yy'.
- $s(c) = \{z, zz, zzz\}$: O símbolo 'c' pode ser substituído por 'z', 'zz', ou 'zzz'.

Como Funciona

- Quando aplicamos a substituição, cada letra do alfabeto Σ_1 é transformada nas possíveis cadeias definidas pela função s.

- Por exemplo, se temos uma cadeia 'abc' em Σ_1, após a substituição, poderíamos obter cadeias como 'xyz', 'xyzz', 'xyzzz', etc., dependendo de como escolhemos substituir cada letra.

Agora vamos aplicar essa substituição em várias cadeias do alfabeto Σ_1:

1. Para a cadeia 'a' em Σ_1:
   - Após a substituição, obtemos 'x', porque $s(a) = \{x\}$.
2. Para a cadeia 'b' em Σ_1:
   - Podemos obter 'y' ou 'yy', porque $s(b) = \{y, yy\}$.
3. Para a cadeia 'ab' em Σ_1:
   - Podemos obter 'xy' ou 'xyy', combinando as possibilidades de substituição para 'a' e 'b'.
4. Para a cadeia 'bc' em Σ_1:
   - As opções incluem 'yz', 'yzz', 'yzzz', 'yyz', 'yyzz', e 'yyzzz', combinando as substituições para 'b' e 'c'.
5. Para a cadeia 'abc' em Σ_1:
   - Podemos ter 'xyz', 'xyzz', 'xyzzz', 'xyyz', 'xyyzz', 'xyyzzz', e assim por diante, escolhendo uma substituição para cada símbolo em 'abc'.

Além de aplicar substituições a elementos individuais de um alfabeto, também podemos aplicar uma substituição a uma cadeia inteira. Esta operação é definida de maneira indutiva, ou seja, construída passo a passo, aplicando a substituição a cada símbolo da cadeia.

Definição Indutiva

Seja s uma substituição e w uma cadeia, a aplicação de s em w, denotada por $s(w)$, segue estas regras:

- Para a cadeia vazia ϵ, $s(\epsilon) = \epsilon$.
- Para uma cadeia $a\alpha$, onde a é um símbolo do alfabeto Σ_1 e α é uma subsequência de cadeias em Σ_1^*, temos:

$$s(a\alpha) = s(a)s(\alpha)$$

Ou seja, a substituição de uma cadeia é o resultado da concatenação das substituições de cada um de seus símbolos.

Exemplo Prático

Suponha a cadeia $w = abc$ e a seguinte substituição s:

- $s(a) = \{x\}$
- $s(b) = \{y, yy\}$
- $s(c) = \{z, zz, zzz\}$

Então, para aplicar a substituição em w:

1. Começamos com $s(abc)$.

2. Aplicamos a substituição ao primeiro símbolo e ao restante da cadeia:
$$s(abc) = s(a)s(bc)$$
3. Continuamos aplicando a substituição para cada parte:
$$s(a) = \{x\}$$
$$s(bc) = s(b)s(c)$$
$$s(b) = \{y, yy\}$$
$$s(c) = \{z, zz, zzz\}$$
4. Combinando todas as possíveis substituições de $s(a)$ e $s(b)$ com $s(c)$, obtemos um conjunto de cadeias resultantes de $s(abc)$.

Portanto, $s(abc)$ gera um conjunto de cadeias que inclui combinações como 'xyz', 'xyzz', 'xyzzz', 'xyyz', 'xyyzz', 'xyyzzz', e assim por diante.

Vamos aplicar a substituição em uma cadeia mais complexa, digamos $w = abac$, usando a mesma substituição s do exemplo anterior:

- $s(a) = \{x\}$
- $s(b) = \{y, yy\}$
- $s(c) = \{z, zz, zzz\}$

A cadeia $w = abac$ será analisada e transformada pela substituição s.

1. Começamos decompondo a cadeia e aplicando a substituição passo a passo:
$$s(abac) = s(a)s(bac)$$
2. Aplicando a substituição ao primeiro símbolo e ao restante da cadeia sequencialmente:
   - Para o primeiro 'a': $s(a) = \{x\}$
   - Para o restante 'bac':
$$s(bac) = s(b)s(ac)$$
3. Continuamos desdobrando cada parte:
   - Para 'b': $s(b) = \{y, yy\}$
   - Para 'ac':
$$s(ac) = s(a)s(c)$$
   - Para o segundo 'a': $s(a) = \{x\}$
   - Para 'c': $s(c) = \{z, zz, zzz\}$
4. Agora, combinamos as substituições de cada símbolo para obter o resultado final:
   - Combinando $s(a)$, $s(b)$, e $s(ac)$, temos múltiplas combinações possíveis, dependendo das escolhas em $s(b)$ e $s(c)$.

Exemplo Concreto de Combinações

- Considerando uma das possíveis substituições para 'b' e 'c', podemos ter:
   - Se escolhemos 'y' para 'b' e 'z' para 'c', uma das combinações possíveis para $s(abac)$ seria 'xyxz'.
   - Se escolhemos 'yy' para 'b' e 'zzz' para 'c', outra combinação possível seria 'xyyxzzz'.

Assim, a substituição s é aplicada a cada parte da cadeia $w = abac$, gerando um conjunto de cadeias resultantes que refletem todas as combinações possíveis de substituições conforme definido por s.

Slide 35

A operação de substituição, previamente discutida no contexto de cadeias individuais, pode ser ampliada para ser aplicada a uma linguagem completa. Isso nos permite transformar todas as cadeias em uma linguagem de acordo com regras de substituição específicas.

Definição para Linguagens

Seja s uma substituição e L uma linguagem, então a aplicação de s em L, denotada por $s(L)$, é definida como:

$$s(L) = \{y | y = s(x) \ para \ x \in L\}$$

Isso significa que para cada cadeia x em L, aplicamos a substituição s para gerar uma nova cadeia y, e o conjunto de todas essas novas cadeias forma a linguagem $s(L)$.

Exemplo Prático

Suponha a linguagem $L = \{a^i b^i c^i | i \geq 1\}$ sobre o alfabeto $\Sigma_1 = \{a, b, c\}$. Cada cadeia em L consiste em 'a's, 'b's e 'c's em número igual, mas variável.

Se definirmos uma substituição s tal que:

- $s(a) = \{x\}$
- $s(b) = \{y, yy\}$
- $s(c) = \{z, zz, zzz\}$

Então, ao aplicar s a L, consideramos todas as possíveis combinações geradas pela substituição:

- Para uma cadeia $a^i b^i c^i$ em L, onde $i \geq 1$, s mapeia cada 'a' para 'x', cada 'b' para 'y' ou 'yy', e cada 'c' para 'z', 'zz' ou 'zzz'.
- Assim, $s(L)$ incluirá cadeias como $x^i y^j z^k$ onde $i \geq 1$, $i \leq j \leq 2i$, e $i \leq k \leq 3i$ refletindo as múltiplas possibilidades de substituição para 'b' e 'c'.

Conclusão

Aplicar uma substituição a uma linguagem permite a transformação de todas as suas cadeias em novas cadeias, seguindo as regras de substituição definidas. Isso resulta em uma nova linguagem que pode ter características e estruturas complexas, dependendo das regras de substituição aplicadas.

🧠 Exercício: Substituição de Cadeias entre Alfabetos

Você deve implementar uma função chamada `aplicar_substituicao(cadeia, substituicoes)` que recebe:

- Uma **cadeia** formada por símbolos de um alfabeto Σ_1 (por exemplo: `'abc'`), e
- Um **dicionário de substituições** que mapeia cada símbolo de Σ_1 para **um conjunto de cadeias** (strings) sobre um alfabeto Σ_2.

A função deve **gerar todas as cadeias possíveis** resultantes da substituição de cada símbolo da cadeia original por uma das opções possíveis no dicionário.

✅ Regras

- Para cada símbolo da cadeia original, substitua-o por **todas as cadeias possíveis** associadas a ele.
- O resultado final deve ser **todas as combinações possíveis**.
- A ordem das combinações não importa.
- Retorne a lista **sem repetições**.

🧪 Exemplo de uso

```
substituicoes = {
    'a': ['x'],
    'b': ['y', 'yy'],
    'c': ['z', 'zz', 'zzz']
}

resultado = aplicar_substituicao('abc', substituicoes)
print(sorted(resultado))
```

Saída esperada (ordenada):

```
['xyz', 'xyyzz', 'xyyzzz', 'xyyz', 'xyzz', 'xyzzz']
```

💡 Dica

Use `itertools.product` para gerar as combinações possíveis.

Coloque a sua resposta na célula abaixo

```
from itertools import product

def aplicar_substituicao(cadeia, substituicoes):

    return list(resultados)
```

✅ Casos de Teste Automatizados

```python
import unittest
## Se der OK ao final do teste, significa que o código está correto.
class TestSubstituicao(unittest.TestCase):

    def test_exemplo_base(self):
        substituicoes = {
            'a': ['x'],
            'b': ['y', 'yy'],
            'c': ['z', 'zz', 'zzz']
        }
        cadeia = 'abc'
        resultado = aplicar_substituicao(cadeia, substituicoes)
        esperado = {'xyz', 'xyzz', 'xyzzz', 'xyyz', 'xyyzz', 'xyyzzz'}
        self.assertEqual(set(resultado), esperado)

    def test_com_caracteres_repetidos(self):
        substituicoes = {
            'a': ['0', '1'],
            'b': ['x'],
        }
        cadeia = 'aab'
        resultado = aplicar_substituicao(cadeia, substituicoes)
        esperado = {
            '00x', '01x', '10x', '11x'
        }
        self.assertEqual(set(resultado), esperado)

    def test_unico_simbolo(self):
        substituicoes = {'a': ['p', 'q', 'r']}
        cadeia = 'a'
        resultado = aplicar_substituicao(cadeia, substituicoes)
        esperado = {'p', 'q', 'r'}
        self.assertEqual(set(resultado), esperado)

    def test_sem_substituicoes(self):
        substituicoes = {}
        cadeia = ''
        resultado = aplicar_substituicao(cadeia, substituicoes)
        self.assertEqual(resultado, [''])

unittest.main(argv=[''], exit=False)
```

```
....
----------------------------------------------------------------------
```