Sumário

Introdução

O que são Autômato Finito Não Determinístico ?

Um Autômato Finito Não Determinístico (AFN) é um modelo matemático utilizado para representar sistemas que possuem um conjunto finito de estados e processam uma sequência de entradas, determinando se a sequência é aceita ou rejeitada

O que são expressões regulares (regex)?

Elas são compostas por uma combinação de caracteres e meta caracteres que formam uma espécie de "linguagem" para especificar padrões de correspondência. Amplamente utilizadas em diversas áreas da computação

Fundamentação Teórica

AFN

é composto por:

- Fita
  - Disposito de entrada
  - Contém a informações a ser processada
- Unidade de controle
  - Reflete o estado corrente da máquina
  - Possui unidade leitura, cabeça da fita
  - Acessa uma cédula da fita de cada vez
  - Movimenta-se exclusivamente para a direita
- programa(função programa ou transição)
  - Comanda as leituras
  - Define o estado da máquina

REGEX

As expressões regulares (regex) são ferramentas que descrevem padrões de strings de forma concisa e poderosa. Elas são usadas para busca, validação, substituição e extração de dados textuais em diversas linguagens de programação e sistemas

sintaxe e componentes

- Literais: Caracteres simples que devem aparecer exatamente como estão na string. Ex: a, b
- Metacaracteres: Símbolos com significados especiais

Ex

- .: Qualquer caractere
- *:Zero ou mais ocorrências do elemento maior
- +:Uma ou mais ocorrências
- ?:Zero ou uma ocorrência
- /d:Um caractere alfanumérico ou sublinhado.

Desenvolvimento

Requisitos para o código:

- Receber uma entrada do usuário, a expressão regular
- Percorrer a expressão regular
- Identificar símbolos e operadores
- implementar 3 tipos de operadores: Concatenação(“.”), União(“|”) e Fecho de Kleene(“*”).
- Implementar tratamento de grupos com parênteses
- Construir uma AFN com base na expressão
- Saída legível que mostre as transições dos estados

Estrutura do código:

- **Classe AFN**

  Define uma estrutura para representar um AFN com três atributos:
  - Estado_inicial: o estado inicial do autômato
  - Estado_final: o estado final do autômato
  - Transicao: um dicionário que mapeia estados para as transições Cada transição consiste em um símbolo (incluindo 'ε' para transições vazias) associado a um conjunto de estados de destino

Estrutura do código:

- **Função:**

  regex_para_afn(regex: str) -> AFN:

  - A função utiliza uma abordagem baseada em pilhas para construir o AFN de forma incremental, respeitando a precedência e a associatividade dos operadores.
  - Símbolos alfanuméricos: São transformados em AFNs básicos
  - Operadores(“.”,”|”,”*”): São armazenados em uma pilha de operadores, garantindo que operações de maior precedência sejam executadas primeiro
  - Parênteses: São usados para agrupar operações e controlados por empilhamento/desempilhamento
  - Ao final da varredura, qualquer operador remanescente na pilha é processado

Execução

- **Entrada**

```
# Entrada do usuário para a expressão regular.
regex = input("Digite a expressão regular: ")
afn = regex_para_afn(regex)
```

- **Principal**

```
    # Itera sobre a expressão regular para construir o AFN.
    for char in regex:
        if char.isalnum():  # Se for um símbolo, cria um AFN básico.
            operandos.append(criar_afn_para_simbolo(char))
        elif char in {'*', '|', '.'}:  # Operadores da regex.
            # Aplica operadores de maior ou igual precedência no topo da pilha.
            while operadores and operadores[-1] != '(' and prioridade[operadores[-1]] >= prioridade[char]:
                aplicar_operador(operadores, operandos)
            operadores.append(char)
        elif char == '(':
            operadores.append(char)  # Abre um grupo.
        elif char == ')':
            # Fecha um grupo e aplica operadores dentro do parêntese.
            while operadores[-1] != '(':
                aplicar_operador(operadores, operandos)
            operadores.pop()  # Remove o '(' da pilha.

    # Aplica operadores restantes.
    while operadores:
        aplicar_operador(operadores, operandos)

    return operandos[0]  # Retorna o AFN final.
```

Execução

- **Função ID para estados**

```python
# Gera IDs únicos para estados
id_generator = count(1)

def prox_estado_id() -> str:
    return f'S{next(id_generator)}'  # Retorna um novo ID de estado.
```

- **Criar AFN para o símbolo**

```python
def criar_afn_para_simbolo(simbolo: str) -> AFN:
    # Cria um AFN básico para um único símbolo.
    estado_inicial = prox_estado_id()
    estado_final = prox_estado_id()
    transicao = {}
    adicionar_transicao(transicao, estado_inicial, simbolo, {estado_final})
    return AFN(estado_inicial, estado_final, transicao)
```

Execução

- **Executar operação**

```
def aplicar_operador(operadores, operandos):
    # Aplica o operador do topo da pilha de operadores aos operandos apropriados.
    operador = operadores.pop()
    if operador == '.':
        operandos.append(concatenar_afn(operandos.pop(-2), operandos.pop()))
    elif operador == '|':
        operandos.append(uniao_afn(operandos.pop(-2), operandos.pop()))
    elif operador == '*':
        operandos.append(asterisco_afn(operandos.pop()))
```

Execução

- **Operação de Concatenação**

```python
def concatenar_afn(afn1: AFN, afn2: AFN) -> AFN:
    # Concatena dois AFNs adicionando uma transição ε do estado final de afn1 para o inicial de afn2.
    adicionar_transicao(afn1.transicao, afn1.estado_final, 'ε', {afn2.estado_inicial})
    return AFN(afn1.estado_inicial, afn2.estado_final, {**afn1.transicao, **afn2.transicao})
```

- **Operação de União**

```python
def uniao_afn(afn1: AFN, afn2: AFN) -> AFN:
    # Cria um novo AFN que representa a união (|) de dois AFNs.
    estado_inicial = prox_estado_id()
    estado_final = prox_estado_id()
    transicao = {estado_inicial: {'ε': {afn1.estado_inicial, afn2.estado_inicial}},
                 afn1.estado_final: {'ε': {estado_final}},
                 afn2.estado_final: {'ε': {estado_final}},
                 **afn1.transicao, **afn2.transicao}
    return AFN(estado_inicial, estado_final, transicao)
```

Execução

- **Operação Fecho de Kleener**

```python
def asterisco_afn(afn: AFN) -> AFN:
    # Cria um novo AFN que representa a operação de fecho de Kleene (*) sobre um AFN.
    estado_inicial = prox_estado_id()
    estado_final = prox_estado_id()
    transicao = {estado_inicial: {'ε': {afn.estado_inicial, estado_final}},
                 afn.estado_final: {'ε': {afn.estado_inicial, estado_final}},
                 **afn.transicao}
    return AFN(estado_inicial, estado_final, transicao)
```

Execução

- **Retorno da função**

```
    # Aplica operadores restantes.
    while operadores:
        aplicar_operador(operadores, operandos)

    return operandos[0]  # Retorna o AFN final.
```

- **Saída**

```
# Exibe os detalhes do AFN gerado.
print("AFN:")
print(f"Estado Inicial: {afn.estado_inicial}")
print(f"Estado Final: {afn.estado_final}")
print("Transições:")
print(formatar_transicoes(afn.transicao))
```

Resultados

O código foi capaz de realizar o objetivo proposto, fazendo todas as operações esperadas, exemplos:

Exemplo Simples:
A.B

```
AFN:
Estado Inicial: S1
Estado Final: S4
Transições:
S1 --A--> S2
S2 --ε--> S3
S3 --B--> S4
```

Exemplo
União:
A|B

```
AFN:
Estado Inicial: S5
Estado Final: S6
Transições:
S5 --ε--> S1, S3
S2 --ε--> S6
S4 --ε--> S6
S1 --A--> S2
S3 --B--> S4
```

Resultados

Exemplo Fecho de Kleene:
A*

```
AFN:
Estado Inicial: S3
Estado Final: S4
Transições:
S3 --ε--> S1, S4
S2 --ε--> S1, S4
S1 --A--> S2
```

Exemplo Complexo:
(A|B)*.C

```
AFN:
Estado Inicial: S7
Estado Final: S10
Transições:
S7 --ε--> S8, S5
S6 --ε--> S8, S5
S5 --ε--> S1, S3
S2 --ε--> S6
S4 --ε--> S6
S1 --A--> S2
S3 --B--> S4
S8 --ε--> S9
S9 --C--> S10
```

Conclusão

A implementação apresentada converte expressões regulares em Autômatos Finitos Não Determinísticos (AFN), demonstrando a relação prática entre linguagens formais e autômatos

O código é modular, escalável e utiliza pilhas para respeitar a precedência dos operadores, garantindo a correta construção do AFN

Além de reforçar conceitos teóricos, a solução serve como base para futuras extensões, como conversão para AFD ou reconhecimento de linguagens, evidenciando a aplicação prática de ideias teóricas em Ciência da Computação

Referencias

- COMPUTANDO! Autômatos #01: Algoritmo 01 - Expressão Regular para Autômato Finito Não-Determinístico Teoria. Disponível em: <https://www.youtube.com/watch?v=zwcdwEQEayM>. Acesso em: 14 jan. 2025.
- DAVI ROMERO DE VASCONCELOS. Linguagens Regulares: Convertendo Expressão Regular em Autômato Finito Não-Determinístico. Disponível em: <https://www.youtube.com/watch?v=JWttym_qNI0>. Acesso em: 14 jan. 2025.
- Regular expression to ∈-NFA. Disponível em: <https://www.geeksforgeeks.org/regular-expression-to-nfa/>.
- DARKGEEKMS. GitHub - DarkGeekMS/regex-to-nfa: A python tool for Regex conversion to nondeterministic finite automaton (NFA). Disponível em: <https://github.com/DarkGeekMS/regex-to-nfa/tree/main>. Acesso em: 14 jan. 2025.