Relatório de Desenvolvimento do Simulador de Linguagens Regulares

Alunos:

- Marcus Vinicius Ramos de Araujo
- Guilherme Colonhese Camargo

Introdução

Este relatório apresenta o desenvolvimento de um simulador de linguagens regulares que permite a criação, edição e simulação de expressões regulares, autômatos finitos (determinísticos e não determinísticos) e gramáticas regulares. O objetivo principal é proporcionar uma ferramenta interativa que auxilie no aprendizado e compreensão dos conceitos fundamentais da Teoria da Computação.

Funcionalidades Implementadas

1. **Expressões Regulares:**
    - **Entrada e Validação:** Entrada e validação de expressões regulares seguindo a sintaxe padrão.
    - **Simulação de Correspondência:** Simulação de correspondência de expressões regulares com cadeias de entrada fornecidas pelo usuário.
    - **Indicação de Resultados:** Indicação clara de aceitação ou rejeição das cadeias em relação à expressão regular.
2. **Autômatos Finitos:**
    - **Editor Visual:** Editor visual para criação e edição de autômatos finitos, permitindo adicionar estados, transições e definir estados iniciais e de aceitação.
    - **Múltiplos Símbolos e Self-Loops:** Suporte a múltiplos símbolos em uma única transição, inclusive para auto-transições (self-loops).
    - **AFD e AFND:** Opção para definir o autômato como determinístico (AFD) ou não determinístico (AFND).
    - **Múltiplos Estados Iniciais:** Permissão para múltiplos estados iniciais, conforme necessário.
    - **Simulação Avançada:** Simulação de autômatos com múltiplas entradas, exibindo resultados individuais para cada uma.
    - **Simulação Passo a Passo:** Simulação passo a passo (teste de mesa), permitindo ao usuário acompanhar a execução do autômato em cada etapa.
    - **Feedback Visual:** Destaque visual dos estados atuais durante a simulação, facilitando a compreensão do funcionamento do autômato.

3. **Gramáticas Regulares:**
    - **Editor de Gramáticas:** Editor para definição de gramáticas regulares, incluindo não-terminais, terminais e produções.
    - **Conversão Automática:** Conversão automática de gramáticas regulares em autômatos finitos não determinísticos.
    - **Visualização do Autômato:** Exibição do autômato resultante da gramática definida.
    - **Simulação:** Simulação do autômato gerado a partir da gramática com cadeias de entrada fornecidas pelo usuário.

Descrição das Técnicas Utilizadas

1. Front-End com React e TypeScript

- **React:** Utilizado para construir a interface de usuário de forma modular e reativa, facilitando a manutenção e escalabilidade da aplicação.
- **TypeScript:** Adicionado para melhorar a robustez do código com tipagem estática, reduzindo erros em tempo de desenvolvimento e facilitando a colaboração.

2. Manipulação de Grafos com `react-flow-renderer`

- **Biblioteca `react-flow-renderer`:** Escolhida para a criação e manipulação de grafos que representam os autômatos finitos. Essa biblioteca permite uma interação intuitiva com nodos e arestas, além de suportar funcionalidades avançadas como arrastar e soltar, zoom e pan.
- **Componentes Customizados:** Desenvolvimento de componentes personalizados (`StateNode`, `StartNode`, `SelfLoopEdge`) para aprimorar a visualização e interatividade dos autômatos.

3. Design de Autômatos e Conversão de Gramáticas

- **Modelagem de Autômatos:** Implementação de estruturas de dados para representar estados, transições e propriedades (inicial, aceitação).
- **Conversão de Gramáticas Regulares para AFND:** Desenvolvimento de algoritmos que transformam gramáticas regulares em autômatos finitos não determinísticos, seguindo as regras de conversão da Teoria da Computação.

4. Simulação de Autômatos

- **Simulação Determinística (AFD):** Implementação de algoritmos que processam cadeias de entrada determinando a aceitação ou rejeição com base nas transições definidas.
- **Simulação Não Determinística (AFND):** Desenvolvimento de mecanismos para lidar com múltiplas transições possíveis e fechaduras ε, permitindo simulações mais flexíveis e abrangentes.

- **Simulação Passo a Passo:** Criação de um sistema que permite ao usuário avançar manualmente através de cada etapa da simulação, acompanhando o estado atual e as transições efetuadas.

5. Interface de Usuário com `react-bootstrap`

- **Componentes UI:** Utilização de componentes estilizados para formulários, botões, modais e listas, garantindo uma experiência de usuário agradável e consistente.
- **Responsividade:** Design responsivo para garantir a usabilidade em diferentes dispositivos e tamanhos de tela.

Qualidade da Solução Implementada

1. Usabilidade

- **Interface Intuitiva:** A utilização de `react-flow-renderer` proporciona uma interação direta e intuitiva na criação e edição de autômatos, permitindo que usuários visualizem claramente os estados e transições.
- **Feedback Visual:** Destaques e animações durante a simulação auxiliam na compreensão do fluxo do autômato, facilitando o aprendizado.

2. Robustez

- **Validação de Entradas:** Implementação de validações rigorosas para expressões regulares, gramáticas e transições, prevenindo erros e inconsistências nos autômatos.
- **Gestão de Estados:** Controle eficaz de estados internos para garantir a consistência dos dados e evitar estados inválidos ou duplicados.

3. Desempenho

- **Otimização de Renderização:** Utilização eficiente do React e suas práticas de renderização para minimizar reflows e repaints, garantindo uma experiência fluida mesmo com autômatos complexos.
- **Algoritmos de Simulação:** Implementação de algoritmos otimizados para simulação de AFD e AFND, assegurando tempos de resposta rápidos durante a simulação.

4. Manutenibilidade e Escalabilidade

- **Modularidade:** Estruturação do código em componentes reutilizáveis e independentes facilita a manutenção e futuras expansões da aplicação.
- **TypeScript:** A tipagem estática proporciona segurança adicional, facilitando a identificação de erros em tempo de desenvolvimento e tornando o código mais legível e compreensível.

5. Acessibilidade

- **Design Responsivo:** Garantia de que a aplicação seja acessível em diferentes dispositivos, promovendo a inclusão de diversos tipos de usuários.
- **Compatibilidade com Navegadores:** Testes realizados em múltiplos navegadores asseguram que a aplicação funcione corretamente em diferentes ambientes.

Estruturação do Código

1. Arquitetura de Componentes

- **Componentes Principais:**
  - **`RegexInput` e `RegexSimulator`:** Gerenciam a entrada e simulação de expressões regulares.
  - **`AutomatonEditor`:** Facilita a criação e edição visual de autômatos finitos.
  - **`AutomatonSimulator`:** Responsável pela simulação de autômatos com entradas fornecidas pelo usuário.
  - **`GrammarEditor`:** Permite a definição de gramáticas regulares e sua conversão para autômatos.
- **Componentes Customizados:**
  - **`StateNode`, `StartNode`, `SelfLoopEdge`:** Melhoram a representação visual dos estados e transições no grafo.

2. Gerenciamento de Estado

- **Hooks do React:**
  - Utilização de hooks como `useState` e `useEffect` para gerenciar estados locais e efeitos colaterais.
  - Hooks específicos do `react-flow-renderer` (`useNodesState`, `useEdgesState`) para gerenciar nodos e arestas de forma eficiente.
- **State Lifting:**
  - Compartilhamento de estados entre componentes através de props e funções de callback, garantindo a sincronização dos dados em toda a aplicação.

3. Separation of Concerns (Separação de Responsabilidades)

- **Modularidade:** Cada componente possui uma responsabilidade clara e distinta, facilitando a manutenção e a escalabilidade.
- **Utilitários e Serviços:**
  - Implementação de funções utilitárias para conversão de gramáticas, validação de expressões e simulação de autômatos.
  - Serviços para persistência de dados, como salvar e carregar autômatos a partir do `localStorage` ou arquivos JSON.

4. Boas Práticas de Desenvolvimento

- **Tipagem Estrita:** Uso extensivo de TypeScript para garantir a consistência dos tipos e prevenir erros.
- **Documentação Inline:** Comentários claros e precisos dentro do código facilitam a compreensão e a colaboração entre desenvolvedores.
- **Consistência de Estilo:** Adoção de padrões de codificação consistentes, facilitando a leitura e a manutenção do código.

5. Testes e Validações

- **Testes Unitários:** Implementação de testes para funções críticas, garantindo que os algoritmos de conversão e simulação funcionem conforme o esperado.
- **Testes de Integração:** Verificação da interação entre componentes, assegurando que a aplicação funcione de forma integrada e sem falhas.

Desafios e Soluções

1. Representação de Múltiplos Símbolos em Transições

- **Desafio:** Garantir que múltiplos símbolos em uma transição sejam representados de forma clara e sem sobreposição de arestas.
- **Solução:** Combinar símbolos em uma única aresta entre dois estados, concatenando os símbolos no rótulo e evitando a criação de arestas duplicadas.

2. Visualização de Auto-Transições (Self-Loops)

- **Desafio:** Desenhar auto-transições que sejam visualmente claras e com rótulos corretamente orientados.
- **Solução:** Desenvolver o componente `SelfLoopEdge` personalizado, ajustando o caminho da curva Bezier para posicionar o loop e corrigir a orientação do texto.

3. Simulação Passo a Passo

- **Desafio:** Permitir que o usuário acompanhe a execução do autômato em cada etapa, visualizando os estados atuais e transições.
- **Solução:** Implementar estados internos no `AutomatonSimulator` para controlar o índice atual, estados ativos e conclusão da simulação, além de destacar os estados ativos no editor visual.

4. Múltiplos Estados Iniciais

- **Desafio:** Adaptar a lógica e visualização para suportar múltiplos estados iniciais sem comprometer a usabilidade.
- **Solução:** Atualizar o modelo de dados para permitir múltiplos estados iniciais e ajustar a visualização, criando nodos e arestas de início para cada estado inicial.

5. Conversão de Gramáticas para Autômatos

- **Desafio:** Desenvolver algoritmos que convertem gramáticas regulares em autômatos finitos não determinísticos de forma eficiente e precisa.
- **Solução:** Implementar a lógica de conversão seguindo as regras da Teoria da Computação, garantindo que cada produção da gramática seja corretamente traduzida para transições no autômato.

6. Gestão de Estado e Performance

- **Desafio:** Manter a aplicação responsiva e eficiente mesmo com autômatos complexos.
- **Solução:** Otimizar a renderização dos componentes React e utilizar algoritmos eficientes para a simulação dos autômatos, além de minimizar re-renderizações desnecessárias.

Testes Realizados

1. **Validação de Expressões Regulares:**
    - **Testes com Expressões Válidas e Inválidas:** Assegurou que a validação está funcionando conforme esperado.
    - **Verificação da Correspondência:** Confirmou a correspondência de cadeias de entrada com as expressões regulares.
2. **Simulação de Autômatos:**
    - **Criação de AFD e AFND:** Incluindo casos com múltiplos estados iniciais e transições com múltiplos símbolos.
    - **Simulação de Entradas Válidas e Inválidas:** Confirmou a aceitação ou rejeição conforme esperado.
    - **Simulação Passo a Passo:** Verificou a atualização correta dos estados ativos e a progressão através dos símbolos de entrada.
3. **Conversão de Gramáticas em Autômatos:**
    - **Definição de Diversas Gramáticas Regulares:** Convertidas para autômatos finitos não determinísticos.
    - **Simulação dos Autômatos Gerados:** Validou a correspondência com as linguagens definidas pelas gramáticas.
4. **Usabilidade e Feedback Visual:**
    - **Interação com o Editor Visual:** Testou a criação, edição e remoção de estados e transições, garantindo uma experiência fluida.
    - **Feedback Visual durante a Simulação:** Confirmou que os estados ativos são destacados corretamente e que as transições são exibidas de forma clara.

Conclusão

O desenvolvimento do simulador de linguagens regulares proporcionou uma compreensão aprofundada dos conceitos teóricos relacionados a expressões regulares, autômatos finitos e gramáticas regulares. A implementação das funcionalidades exigidas, bem como a superação dos desafios encontrados, resultou em uma ferramenta robusta e interativa que pode ser utilizada como suporte educacional no estudo da Teoria da Computação.

A utilização de tecnologias modernas como React, TypeScript e `react-flow-renderer` permitiu a criação de uma interface intuitiva e eficiente, enquanto a adoção de boas práticas de desenvolvimento assegurou a qualidade e a manutenibilidade do código. A capacidade de simular tanto autômatos determinísticos quanto não determinísticos, juntamente com a conversão automática de gramáticas regulares, amplia significativamente o potencial de aprendizado proporcionado pela ferramenta.