
UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

# Simulador de Linguagens Regulares

**Projeto Semestral**

**Discentes**

Lucas Gonçalves Ferreira - 191250422

Matheus Magalhães da Rocha - 191254754

Graduação em Ciência da Computação

FCT/UNESP

**Responsável pela Disciplina**

Prof. Celso Olivete Junior

Departamento de Matemática e Computação

FCT/UNESP

Presidente Prudente - SP

2021

LUCAS GONÇALVES FERREIRA - 191250422
MATHEUS MAGALHÃES DA ROCHA - 191254754

# Simulador de Linguagens Regulares

Projeto Semestral

Projeto Semestral da disciplina de Linguagens Formais e Teoria da Computação, apresentado como parte integrante da nota final da disciplina, além de necessário para aprovação na mesma.

Docente: Prof. Celso Olivete Junior

Presidente Prudente - SP
2021

## SUMÁRIO

## INTRODUÇÃO

O presente trabalho é um relatório do projeto semestral da disciplina de Linguagens Formais e Teoria da Computação(LFTC), proposto aos alunos com a finalidade de reforçar o conhecimento deles sobre os temas estudados na disciplina, em outras palavras, o objetivo do projeto é desenvolver um simulador de linguagens regulares, que consiga simular Expressões Regulares (ER), Autômatos Finitos Determinísticos (AFD) e Não-Determinísticos (AFND) e Gramáticas Regulares (GR).

Este corrente trabalho foi dividido da seguinte forma: na seção 1 é apresentado um resumo geral sobre o projeto(ferramentas,linguagens usadas…), na seção 2 são apresentadas as expressões regulares e como elas funcionam na ferramenta, a seção 3 descreve o funcionamento da ferramenta para as gramáticas regulares, e finalmente a seção 4 explica o uso da ferramenta para autômatos finitos.

# 1. SIMULADOR DE LINGUAGENS REGULARES

Como discorrido na introdução, o presente trabalho é um simulador de linguagens regulares, que realizará a simulação de ER's, GR's, AFD's e AFND's, por meio de algoritmos desenvolvidos pelos integrantes do projeto.

Nesta seção serão apresentados alguns detalhes gerais de desenvolvimento dos simuladores e também uma breve explicação das telas dos mesmos.

## 1.1 Detalhes do desenvolvimento

A ferramenta em questão, é um website construído a partir de linguagens voltadas ao desenvolvimento web (HTML,CSS,Javascript), principalmente Javascript, uma vez que o grupo optou pela utilização da biblioteca de código aberto voltada a construção de interfaces, ReactJs e também da linguagem extensão do CSS, Sass.

Para a representação de grafos no simulador de autômatos finitos foi utilizado a biblioteca graphviz-react que permite o uso de um componente para trabalhar com as funcionalidades da biblioteca d3-graphviz, voltada para criação de grafos, dígrafos, entre outros.

Os membros do grupo também utilizaram o editor de texto/código-fonte Visual Studio Code(vscode) para implementação do website, e o sistema de controle de versões distribuído, Git, com um repositório de código sendo hospedado no github.

## 1.2 Funcionamento geral do website

O website como um todo apresenta quatro telas/interfaces principais sendo: uma tela(menu) para escolher qual simulador usar, uma tela para o simulador de ER, uma tela para o simulador de GR e uma tela para o simulador de AF. Obs: será especificado melhor sobre os simuladores em cada seção.

### 1.2.1 Tela de home/menu

A tela inicial que o usuário tem contato, onde ele pode escolher qual simulador ele deseja testar.


**Figura 1** - Tela de menu do website

### 1.2.2 Simulador de ER

A seguinte tela é relativa ao simulador de ER, onde é possível adicionar uma expressão regular e diversos testes(strings de teste) para o mesmo.


**Figura 2** - Tela do Simulador de ER

### 1.2.3 Simulador de GR

Na tela do Simulador de GR, é possível adicionar/excluir diversas regras de produção, e depois escrever strings na área de testes. O resultado do processamento das strings será listado na tabela mais à direita.


**Figura 3** - Tela do Simulador de GR

### 1.2.4 Simulador de AF

Na tela do Simulador de AF, é possível adicionar/excluir estados e transições, além de testar apenas uma entrada ou múltiplas entradas.


**Figura 4** - Tela do Simulador de AF

### 1.2.5 Menu lateral

Embora não seja diretamente uma tela, o menu lateral do site permite trocar de tela rapidamente para facilitar a experiência do usuário.


**Figura 5** - Menu lateral

Obs: É preferível rodar o website em questão em um computador ou notebook, pois o mesmo não foi desenvolvido pensando em telas de tablet ou dispositivos mobile.

## 1.3 Informações para executar e testar o simulador

Para executar o simulador basta apenas seguir os passos abaixo:

1. Possuir o Node.js e o gerenciador de pacotes “yarn” instalados em seu computador.
2. Extrair o arquivo comprimido do trabalho.
3. Entrar na pasta do projeto por um terminal/prompt de comando.
4. Ao se certificar de estar no mesmo diretório que o “package.json”, digitar o comando “yarn install”.
5. Após o processo de instalação acabar, digitar: “yarn start” e uma janela no browser será aberta, com o website rodando.

A fim de facilitar o processo de execução do simulador, a equipe hospedou o website desenvolvido, dessa forma o mesmo pode ser acessado sem a necessidade de baixar nada.

O grupo recomenda acessar o website pelo endereço abaixo mesmo, pois além de ser mais prático, não precisa de todo o passo-a-passo acima para usufruir do mesmo. Obs: O website também está muito pesado, logo baixar pode ficar fora de mão e também podem ocorrer problemas de dependências.

O endereço para o website do simulador é:
https://simulador-er-af-gr.web.app/simulador-af

# 2. EXPRESSÕES REGULARES

Dentro do contexto de linguagens regulares têm-se as Expressões Regulares(ER), uma espécie de notação para representar padrões de strings, e além de denotarem, ou seja, poderem “representar” linguagens regulares, são muito conhecidas e utilizadas em processamento de strings, pesquisas em textos, validações de campos e muitas outras finalidades.

Nesta seção serão apresentadas informações a respeito do funcionamento e da implementação do simulador de Expressões Regulares.

## 2.1 Funcionamento do Simulador de ER

O funcionamento da ferramenta é bem simples, é necessário apenas inserir a expressão regular no campo indicado e então a string de teste(palavra de entrada) nos campos de teste, que a ferramenta irá informar imediatamente se a string inserida é válida ou não de acordo com a ER informada previamente.

A ferramenta funciona de forma semelhante às disponíveis na internet como a regex101 ou a regexp_br (ferramenta apresentada pelo professor), portanto, os resultados delas serão os mesmos que os da ferramenta desenvolvida pelo grupo.

Obs: para dar match/validar exatamente as strings inseridas, é necessário inserir “^” e “$” ao redor da expressão, ou seja: “^expressão$” ou “^(expressão)$” , caso contrário não serão obtidos os valores desejados. O simulador foi feito assim justamente para seguir o padrão de outros simuladores de ER, portanto, tais tratamentos ficam por parte do usuário.

Para testar as funcionalidades do simulador vamos definir uma ER “^abc(a|b)c*$” e algumas strings de teste como: “abcbccc” e “abcc”.

Esta ER definida permite strings que começam obrigatoriamente com “**abc**”, seguido por ou “**a**” ou “**b**” uma única vez, e terminando com qualquer quantidade de “**c**”, inclusive nenhuma.

Podemos ver no simulador que a primeira string “**abcbccc**” está correta de acordo com a ER informada, e então a ferramenta destaca com a cor verde o campo onde ela se encontra.


**Figura 6** - String válida

Todavia a segunda string “**abcc**” não está de acordo com a ER informada, pois após o “abc”, deveria vir ou um “**a**” ou um “**b**”, e nesse caso veio um “**c**”, portando a entrada não é válida  e então a ferramenta destaca com a cor vermelha o campo onde a string se encontra.


**Figura 7** - String inválida

Caso a expressão regular inserida seja inválida, ou seja, sintaticamente incorreta, o interpretador irá informar uma mensagem de erro ao usuário.


**Figura 8** - Erro de sintaxe

Também é possível inserir múltiplas strings de teste ao mesmo tempo e validá-las em tempo real na ferramenta, basta clicar no botão “+Add TEST”, logo abaixo dos campos de teste e um novo campo será adicionado. Para remover um campo, pode-se clicar no ícone de exclusão ao lado de cada campo.


**Figura 9** - Múltiplas strings

Os caracteres válidos para o campo de Expressão Regular são letras, números, símbolos, e também os caracteres especiais das expressões regulares:

| | |
|---|---|
| **^** | Corresponde ao início da linha ou string de texto de uma expressão regular. |
| **$** | Corresponde ao final da linha ou string de texto de uma expressão regular. |
| **\|** | Ou |
| * | Corresponde a qualquer quantidade de um certo caractere, inclusive nenhuma |
| + | Corresponde a qualquer quantidade de um certo caractere. É necessário ter no mínimo “um” caractere, vazio não é válido. |
| **[...]** | Corresponde a qualquer caractere de um conjunto de caracteres |
| () | Utilizado para agrupar expressões como (a\|c) |

Existem muitos outros caracteres especiais de ER que não foram listados aqui, e todos irão funcionar. Desde que a expressão regular esteja sintaticamente correta, o compilador não retornará nenhum erro.

Obs: Para validar apenas uma string de teste vazia usar o regex: “^$”.

### 2.2 Implementação do Simulador de ER

Para a implementação do Simulador de Expressões Regulares foram usados os próprios recursos da linguagem, no caso os recursos do Javascript.

Como sugerido pelo professor, foi o usado o regex nativo da linguagem, e o algoritmo se resume a: pegar o texto do campo do regex, compilar o mesmo, caso se tenha algum erro(expressão regex inválida) é retornado um erro, caso contrário segue a validação, onde será verificado se a string/entrada de teste fornecida pelo usuário é válida ou não.

Obs: caso o campo do regex esteja vazio, não é retornado nenhum resultado ao usuário.

```
const validate = value => {
    try{
        let regex_input = (document.getElementsByName("regex-input")[0]).value;
        let regex = new RegExp(regex_input);
        return (regex_input === "") ? "" : (regex.test(value) ? "true"  : "false");
    }
    catch(e){
        return "error";
    }

}
```

**Figura 10** - Algoritmo para validar ER

# 3. GRAMÁTICAS REGULARES

Uma linguagem regular também pode ser gerada/reconhecida pelo formalismo de uma Gramática Regular(GR), que consiste em uma ou mais variáveis que representam linguagens, e é caracterizada pela quádrupla ordenada G = ( {V}, {T}, P, S), onde V representa representa o vocabulário não terminal da gramática( variáveis), T representa o vocabulário terminal, contendo os símbolos que constituem as sentenças da linguagem, P representa o conjunto de todas as leis de formação (regras de produção) utilizadas pela gramática para definir a linguagem e S representa o símbolo de início.

As Gramáticas Regulares podem ser classificadas nas categorias:

- ☐ GLD: o lado direito da regra é formado por N símbolos terminais seguido de UM símbolo não terminal OU formado apenas por N símbolos terminais.
- ☐ GLE: o lado direito da regra é formado por UM símbolo não terminal seguido de N símbolos terminais OU formado apenas por N símbolos terminais.
- ☐ GLUD: o lado direito da regra é formado por no máximo UM símbolo terminal seguido de UM símbolo não terminal OU formado apenas por UM símbolo terminal.
- ☐ GLUE: o lado direito da regra é formado por UM símbolo não terminal seguido por no máximo UM símbolo terminal OU formado apenas por UM símbolo terminal.

Nesta seção serão apresentadas informações a respeito do funcionamento e da implementação do simulador de Gramática Regular.

## 3.1 Funcionamento do Simulador de GR

Antes de comentar a respeito do funcionamento do Simulador de Gramáticas Regulares, é necessário saber que a implementação do grupo valida apenas as categorias GLUD e GLUE.

Após a informação anterior ter sido bem elucidada, podemos começar a discorrer sobre o funcionamento do simulador.

Este simulador consiste em três partes fundamentais, sendo basicamente os campos de regras, o campo de strings/entradas de teste e a tabela de correspondência.

Nos campos de regras é possível adicionar e remover regras de produção em sua gramática. O campo de regras não aceita “|”, logo é necessário colocar cada "parâmetro” em uma linha diferente. O caractere de vazio(ε) também é comportado aqui e inserido por padrão a cada nova regra adicionada.

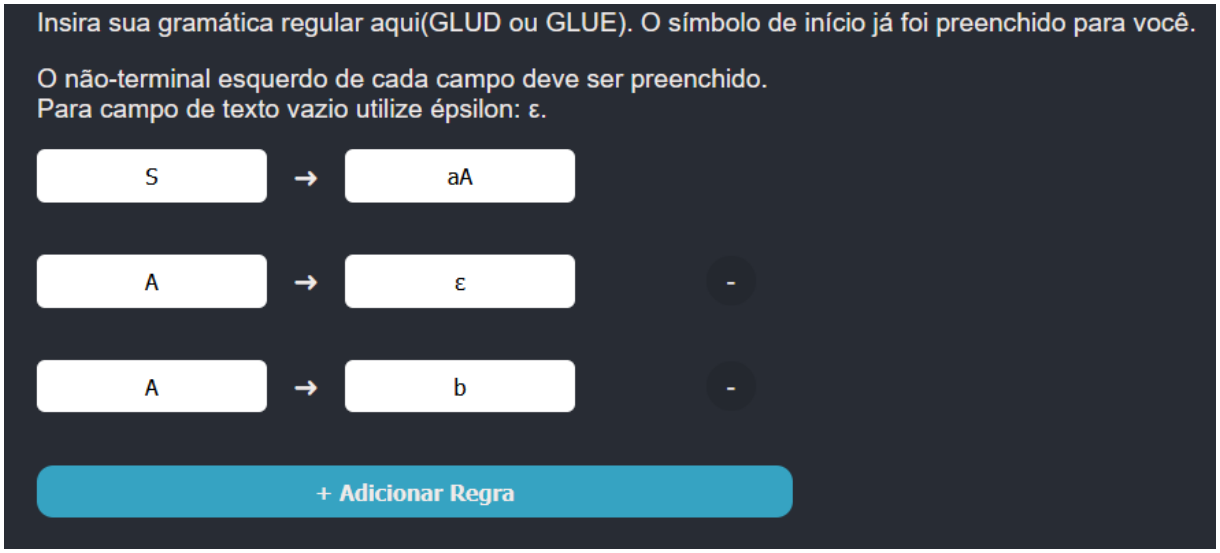



**Figura 11** - Campos de regras

Nos campos de strings de entrada são adicionadas as strings que serão testadas com base nas regras de produção inseridas.

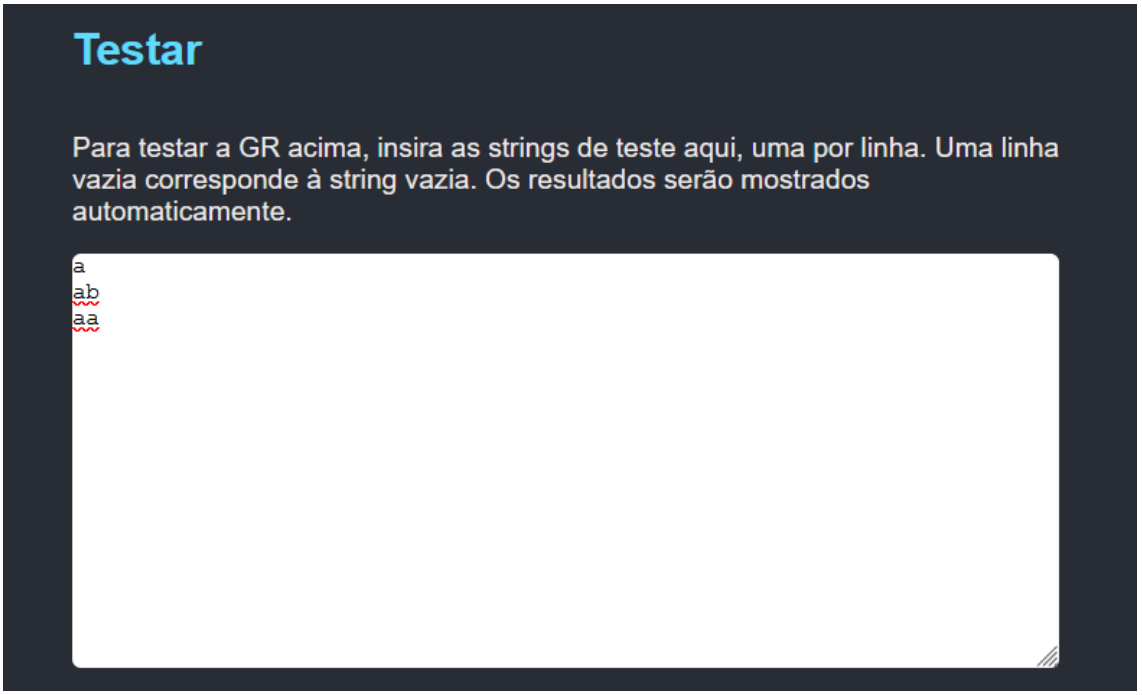



**Figura 12** - Campos de strings de entrada

Cada linha corresponde a uma string diferente e para inserir uma string vazia, basta apenas não digitar nada na linha. Obs: como é possível perceber, podemos inserir tanto uma única string de teste, como várias strings.

Partindo para a última parte fundamental da Gramática, temos a tabela de correspondência, nela são mostrados os resultados da strings de entrada com base nas regras de produção inseridas. Caso a string de entrada esteja de acordo com as regras estabelecidas o campo correspondência é marcado como “sim” e a linha é pintada de verde, caso a string de entrada não esteja de acordo com as regras estabelecidas o campo correspondência é marcado como “não” e a linha é pintada de vermelho.


**Figura 13** - Tabela de correspondência

Dependendo das regras inseridas, o simulador pode acusar um erro de Sintaxe, ou simplesmente invalidar todas as strings de teste, já que se as regras estão erradas, não tem como validar ninguém:


**Figura 14** - Erro de sintaxe

Para fazer um teste um pouco mais profundo no simulador vamos usar o seguinte GR:

Cadeias de comprimento múltiplo de 3.

**GLUD**
G = ( {S,A,B,C}, {a,b,c}, P, S )
P = { S →A,
A → aB | bB | cB | ε,
B → aC | bC | cC,
C → aA | bA | cA }

Strings de teste(strings válidas): {bca, acababababb, ε, acabab, cabacacbb, aabcbabaccba, baaaba, aaa, bbbbbb, aabaacbab, aac, ... }

Primeiramente devem-se inserir as regras nos campos de regras.

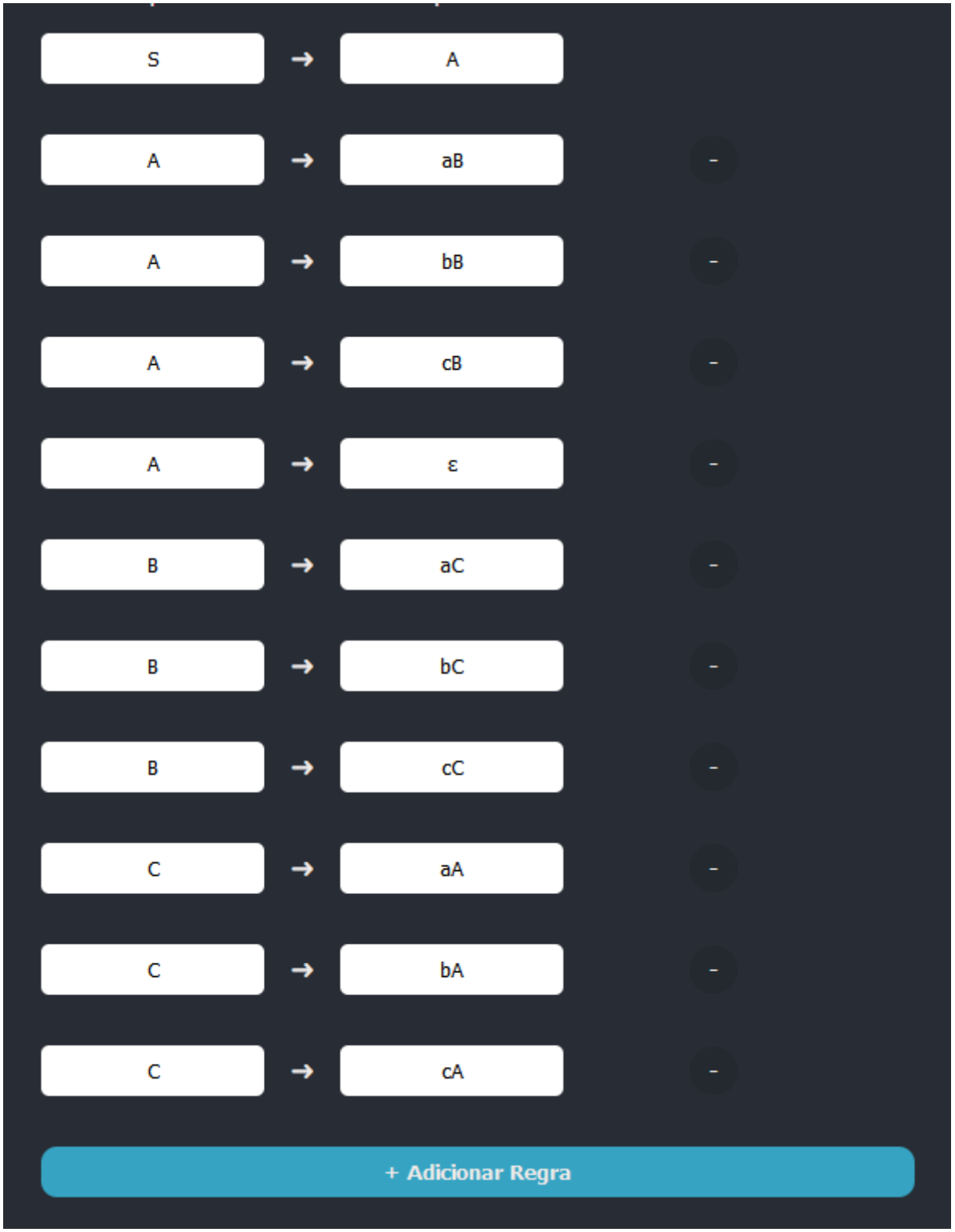



**Figura 15** - Inserindo as regras

Logo em seguida devem-se inserir as strings de testes nos campos de strings de entrada.

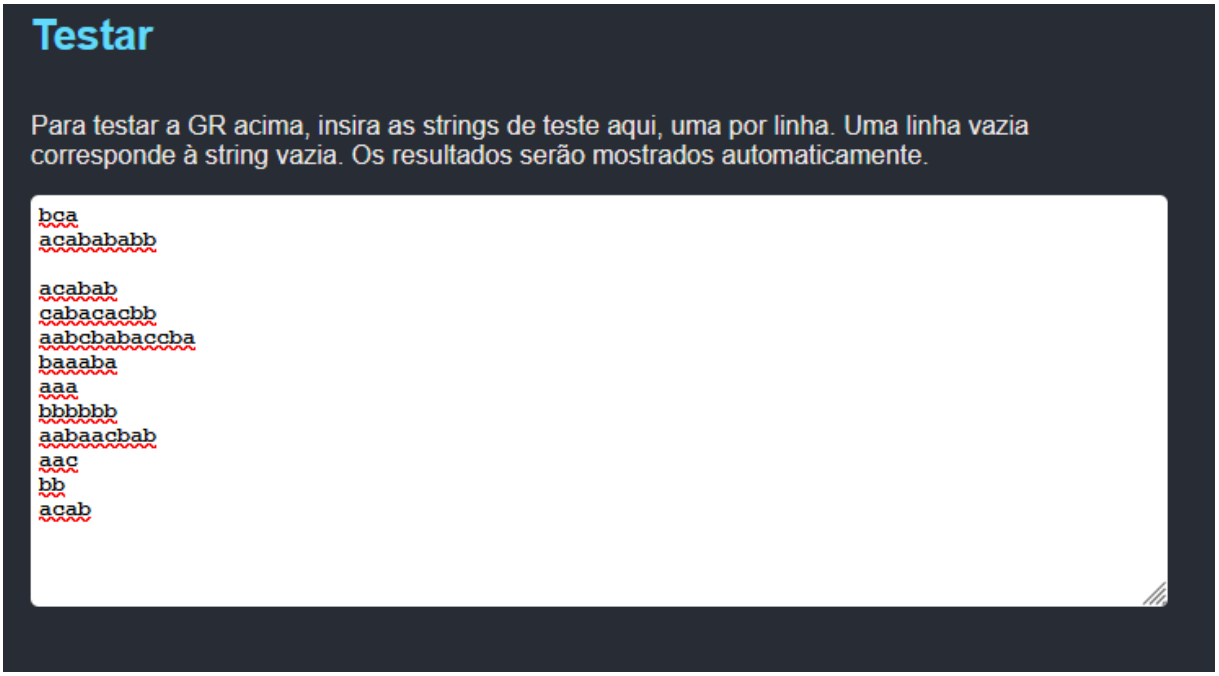
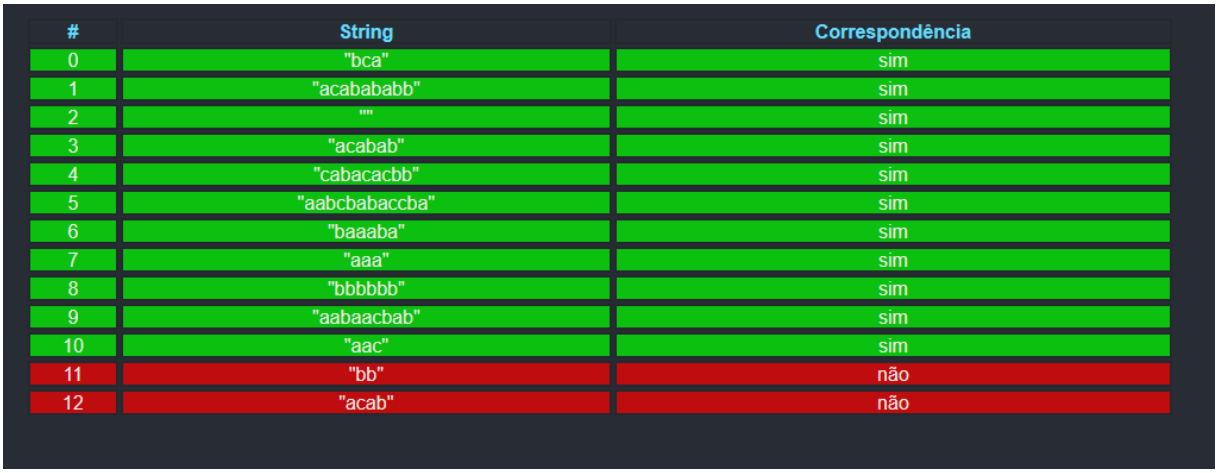



**Figura 16** - Inserindo strings de entrada

E por fim olhar os resultados na tabela de correspondências.



| # | String | Correspondência |
|---|---|---|
| 0 | "bca" | sim |
| 1 | "acababababb" | sim |
| 2 | "" | sim |
| 3 | "acabab" | sim |
| 4 | "cabacacbb" | sim |
| 5 | "aabcbabaccba" | sim |
| 6 | "baaaba" | sim |
| 7 | "aaa" | sim |
| 8 | "bbbbbb" | sim |
| 9 | "aabaacbab" | sim |
| 10 | "aac" | sim |
| 11 | "bb" | não |
| 12 | "acab" | não |

**Figura 17** - Olhando resultados na tabela de correspondência

Como é possível ver na figura, todas as strings que deveriam ser aceitas foram validadas corretamente pelo algoritmo de gramática, o que prova o funcionamento e precisão do mesmo.

### 3.2 Implementação do Simulador de GR

Para a implementação do simulador de Gramática Regular foram utilizados princípios de recursão, onde tem-se duas funções, sendo elas: **grammarValidate** e **validate**.

A função **grammarValidate** possui como objetivo tratar a grammar inserida, determinando seu tipo através de uma lista chamada de validator, caso a gramática inserida seja GLUD será adicionado um parâmetro “D” na chamada da função **validate**, se a gramática inserida for GLUE será adicionado o parâmetro “E” na chamada da função **validate**, e após feita a determinação do tipo, a função irá chamar a função **validate** dentro de um for com todas as regras do símbolo de início.

A função **validate** possui como objetivo validar se as regras percorridas irão formar a string inserida para teste, então as verificações para validar o caso foram centradas nisso, a cada nova chamada recursiva dessa função a string de regra chamada de rule irá concatenar com a próxima regra. As chamadas recursivas irão parar quando a string não satisfazer às regras ou quando as regras forem satisfeitas e não ter mais um símbolo não terminal à direita ou esquerda (dependendo do tipo de gramática inserida).

# 4. AUTÔMATOS FINITOS

Por fim, outro formalismo das Linguagens Regulares são os Autômato Finitos(AF), que podem validar por meio de estados e transições padrões de strings/valores fornecidos.

Os AF podem ser classificados em determinísticos e não-determinísticos, sendo o último caracterizado por ter duas ou mais transições de mesmo caractere que partem de um mesmo estado e vão para outros estados, além de apresentar transições em vazio.

Nesta seção serão apresentadas informações a respeito do funcionamento e da implementação do simulador de Autômatos Finitos.

## 4.1 Funcionamento do Simulador de AF

Como previamente dito na seção 1, para a representação de grafos no simulador de autômatos finitos foi utilizado a biblioteca graphviz-react, um port de outra biblioteca de mesmo nome para react. Esta biblioteca permite o desenho de arcos, círculos, retas, polígonos, estados, transições entre outros.

Junto com a graphviz-react também foi usado a biblioteca react-modal, para permitir a criação de modais que interagem diretamente com o usuário para criar estados, criar transições, deletar estados e transições e selecionar estados iniciais e finais.

Este simulador consiste em duas partes fundamentais, sendo basicamente o campo de criação de autômato, e o campo de strings/entradas de teste.

Começando pela primeira parte fundamental, temos o campo de criação de autômato, onde o usuário tem um menu que pode usar para escolher diversas opções diferentes para se criar e personalizar um autômato finito.


**Figura 18** - Criação de Autômato

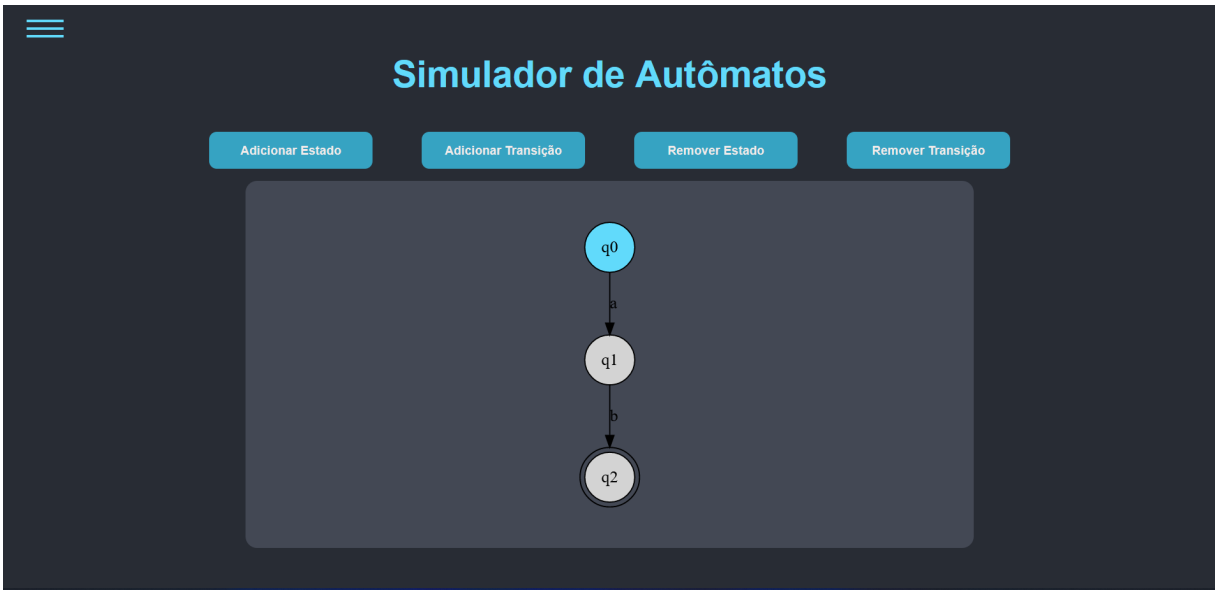



**Figura 19** - Exemplo de Autômato

A primeira opção presente aqui é a “Adicionar Estado”, onde após apertar o botão o usuário é redirecionado para um modal que permite inserir um estado no Autômato Finito.

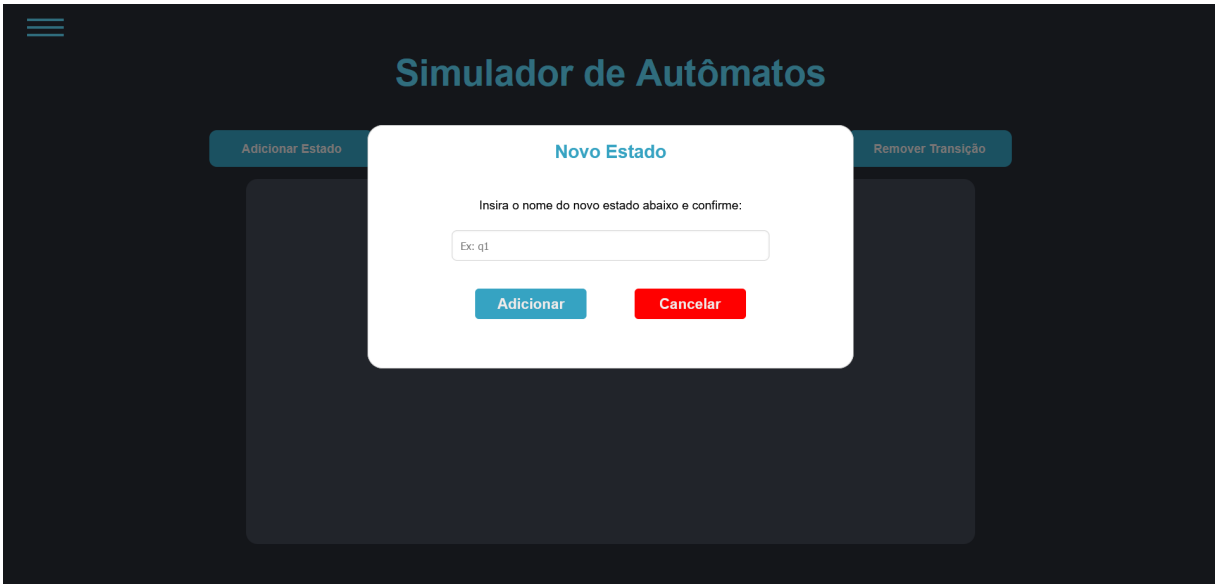



**Figura 20** - Adicionar Estado

A opção seguinte aqui é a “Adicionar Transição”, que permite ao usuário adicionar transições ao autômato caso o mesmo tenha algum estado já criado.


**Figura 20** - Adicionar Transição

Também têm-se a opção de “Remover Estado”, onde o usuário pode excluir algum estado existente(junto com todas suas transições). Esta opção está apenas disponível se algum estado já tiver sido criado.


**Figura 21** - Remover Estado

Assim como “Remover estado”, a opção “Remover Transição” também está disponível. Basta simplesmente selecionar uma das transições listadas no modal que aparecerá na tela. Obs: “label” na foto é o caractere daquela transição.

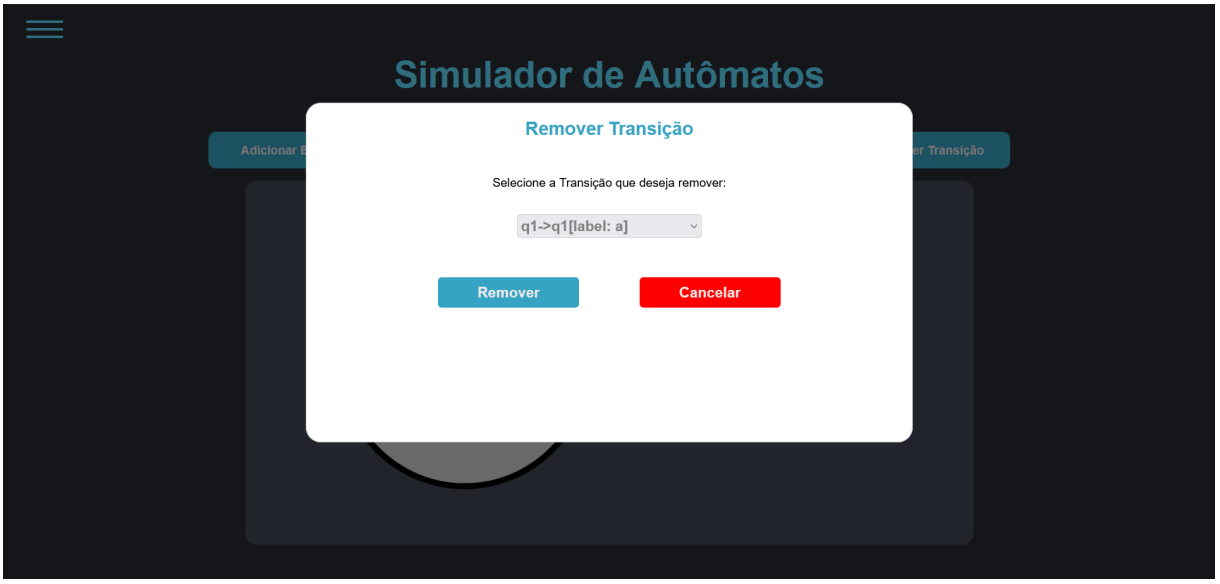



**Figura 22** - Remover Transição

Por fim, ao clicar **duas vezes seguidas** no nome de um estado, uma janela onde é possível escolher se tal estado será inicial ou final aparecerá.

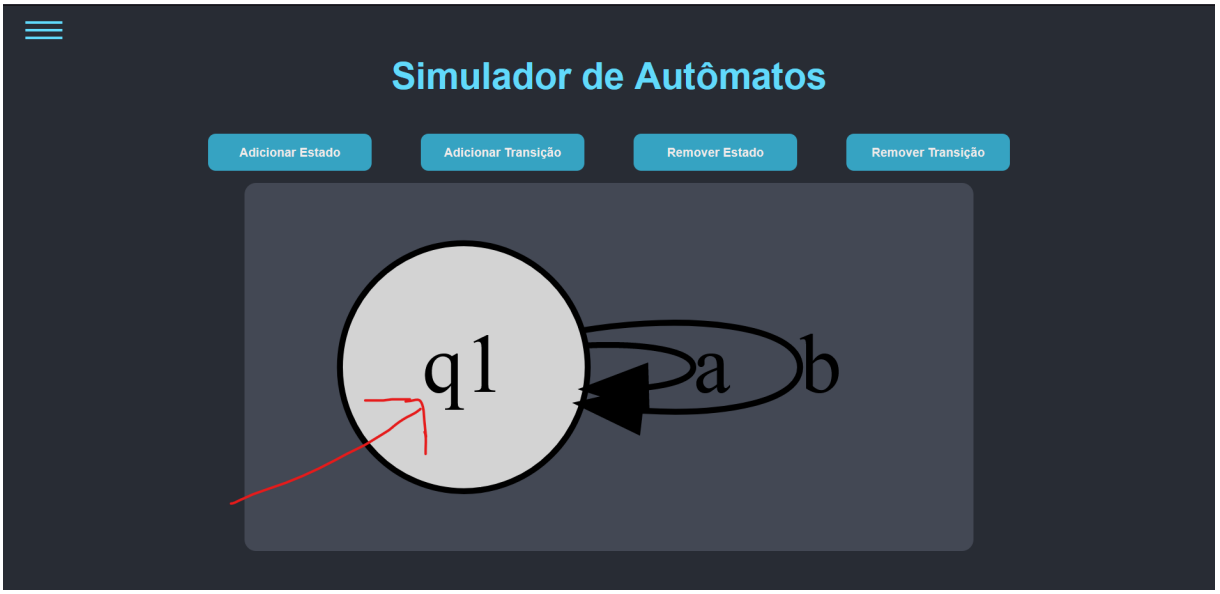



**Figura 23** - Duplo clique no nome do estado

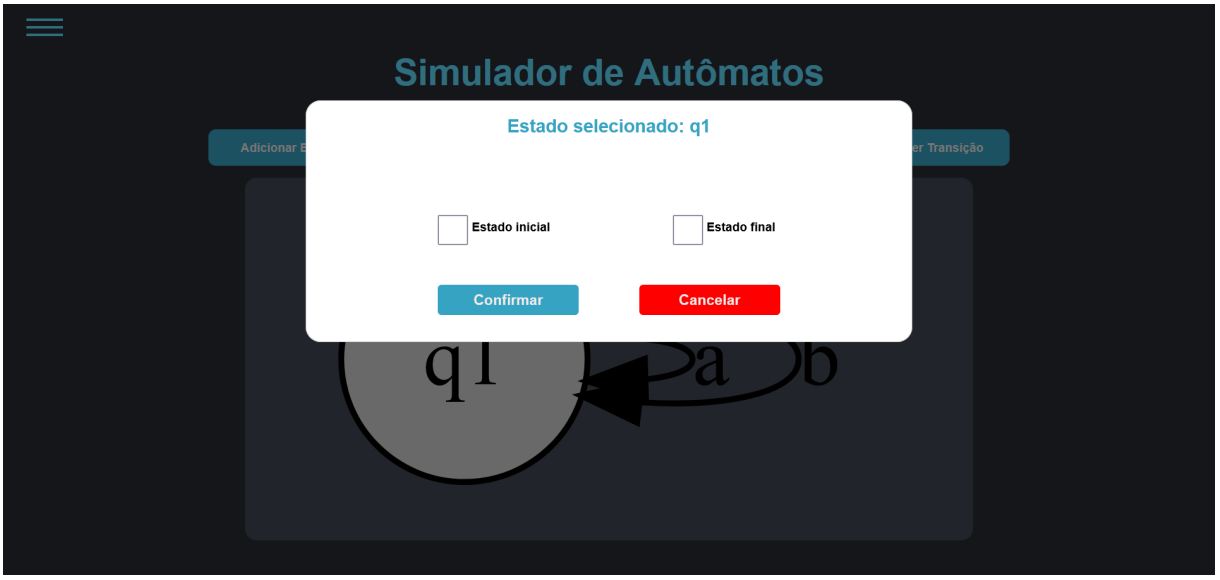



**Figura 24** - Selecionar se o estado é inicial, final ou ambos

Obs: É possível dar dois cliques no estado novamente e mudar tais configurações(clicando nas checkbox presentes no modal), tirando o status de “estado inicial” ou “estado final” do estado em questão.

No campo de testes de string/entradas os usuários podem inserir uma única string para ser testada/validada pelo autômato ou então usar o painel de entrada múltipla para testar múltiplas entradas ao mesmo tempo.


**Figura 25** - Entrada única


**Figura 25** - Entrada múltipla

Caso o autômato valide as strings, os campos de teste serão pintados de verde, caso a string seja rejeitada os campos de teste serão pintados de vermelho.

### 4.2 Implementação do Simulador de AF

Para a implementação do simulador de Autômatos Finitos foram utilizadas **validateAF** (**validateAFMult** no caso de multi-testes, sendo somente uma adaptação da principal), **AFtoGLUD** e **validateRule**.

A função **validateAF** possui como objetivo tratar o AF inserido, transformando ele em uma gramática regular do tipo GLUD através da função **AFtoGLUD** e em seguida a função irá chamar a função **validateRule** dentro de um for com todas as regras do símbolo de início.

A função **AFtoGLUD** inicia-se fazendo cópias dos vetores de estados e transições e manipulando-os para que dessa forma possam ser uma gramática regular do tipo GLUD, o intuito dessa função é apenas renomear os estados e ordenar de forma que fique compatível com o formato exigido pela **validateRule**, função que é baseada na **validate** do simulador de Gramática Regular, porém sendo apenas para GLUD, tendo em vista que a única conversão feita do AF é para GLUD.

Como dito anteriormente, a função **validateRule** possui os mesmos princípios da função **validate** da Gramática Regular, portanto, seu objetivo é validar se as regras percorridas irão formar a string inserida para teste, então as verificações para validar o caso foram centradas nisso, a cada nova chamada recursiva dessa função a string de regra chamada de “rule” irá concatenar com a próxima regra. As chamadas recursivas irão parar quando a string não satisfazer às regras ou quando as regras forem satisfeitas e não ter mais um símbolo não terminal à direita.

## CONCLUSÃO

Neste trabalho foi implementado um simulador de linguagens regulares, onde além de simular Expressões Regulares , Gramáticas Regulares e Autômatos Finitos, estudamos e reforçamos diversos temas da disciplina de Linguagens Formais e Teoria da Computação. Outro ponto também muito importante no trabalho foram os testes, uma vez que o grupo precisou elaborar os mais variados casos de teste para verificar e validar suas implementações e soluções.

Ao longo do projeto, todos os objetivos definidos pela equipe foram concluídos, e embora tenham ocorrido certos problemas em algumas etapas, no fim, tudo foi corrigido e configurado corretamente de acordo com a visão inicial do grupo.

De forma geral, este trabalho foi muito importante para o aprofundamento do nosso conhecimento, uma vez que permitiu-nos compreender melhor a disciplina Linguagens Formais e Teoria da Computação e seus tópicos relacionados, além de ter-nos possibilitado desenvolver competências de investigação, seleção, organização e até comunicação.