Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Um Simulador de APNs

Matheus Henrique Vieira Ribeiro

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Elton Máximo Cardoso

Julho, 2026
João Monlevade–MG

Matheus Henrique Vieira Ribeiro

Um Simulador de APNs

Orientador: Elton Máximo Cardoso

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto
João Monlevade
Julho de 2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Matheus Henrique Vieira Ribeiro

Um Simulador de APNs

Monografia apresentada ao Curso de Sistemas de Informação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação

Aprovada em 30 de Julho de 2026

Membros da banca

Dr. Elton Máximo Cardoso - Orientador(a) (Universidade Federal de Ouro Preto)
Dr. Rodrigo Geraldo Ribeiro - (Universidade Federal de Ouro Preto)
Dr. Roberto Gomes Ribeiro (Universidade Federal de Ouro Preto)

Elton Máximo Cardoso, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 30/07/2026

Documento assinado eletronicamente por **Elton Maximo Cardoso**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 31/07/2026, às 19:51, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1150124** e o código CRC **FEC4BB08**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

Agradecimentos

Agradeço, em primeiro lugar, a Deus, por ter sido meu guia, fortaleza e por me conceder a sabedoria e a resiliência necessárias para superar cada desafio ao longo de toda a minha trajetória acadêmica.

Aos meus amados pais, Jacqueline Ribeiro e Norival Ribeiro, por serem o meu porto seguro e a base de tudo o que sou. Obrigado pelo amor incondicional, pelos sacrifícios invisíveis e por sempre acreditarem nos meus sonhos e no meu potencial.

À minha querida irmã, Iara Ribeiro, pelo companheirismo, apoio constante e por estar ao meu lado dividindo os sorrisos e as ansiedades ao longo desses anos de graduação.

À minha namorada, Emilly Cury, pelo amor, pela paciência infinita e por ser meu braço direito em cada noite em claro dedicada a este projeto. Sua presença e incentivo foram fundamentais para que eu não desanimasse nos momentos mais difíceis.

Ao meu orientador, Prof. Elton M. Cardoso, pela confiança depositada em mim, pela paciência na condução deste trabalho e pelo compartilhamento de seu vasto conhecimento, fundamentais para a consolidação deste projeto.

A todos que, de forma direta ou indireta, fizeram parte desta caminhada e torceram pela minha vitória, expresso minha mais sincera gratidão.

"Science is more than a body of knowledge; it is a way of thinking."

— Carl Sagan (1934 – 1996),
in: The Demon-Haunted World: Science as a Candle in the Dark.

Resumo

A disciplina de Teoria da Computação é fundamental na formação de discentes da área de tecnologia, fornecendo o embasamento teórico para o desenvolvimento de compiladores modernos. Contudo, conceitos relacionados às Linguagens Livres de Contexto (LLCs) e aos seus reconhecedores, os Autômatos de Pilha Não Determinísticos (APNs), frequentemente impõem barreiras ao aprendizado devido ao seu alto nível de abstração e à complexidade inerente de visualizar múltiplas computações paralelas. Este trabalho apresenta o desenvolvimento de uma ferramenta pedagógica baseada na web voltada à modelagem gráfica e à simulação em tempo real de APNs. A metodologia consistiu na implementação de uma aplicação executada integralmente no lado do cliente (*frontend*), utilizando JavaScript puro e a biblioteca Cytoscape para a renderização de grafos. O sistema permite ao usuário desenhar autômatos, definir diferentes critérios de aceitação e analisar dinamicamente as ramificações geradas pelo não-determinismo por meio de estruturas denominadas *snapshots*. Como resultado, a plataforma viabiliza a exportação e importação de modelos em formato JSON e o gerenciamento de exercícios de forma totalmente síncrona e gratuita. Embora o processamento em *thread* única imponha limitações de desempenho frente a volumes massivos de dados, a ferramenta cumpre com sucesso o seu propósito didático de mitigar as dificuldades de visualização estática tradicionais. Como trabalhos futuros, propõem-se a realização de testes empíricos com os estudantes para validação da eficácia pedagógica, a expansão do escopo para outras máquinas de estados e a arquitetura de um *backend* para otimização do desempenho.

Palavras-chave: Teoria da Computação. Autômatos de Pilha Não Determinísticos. Ferramenta Didática. Visualização de Algoritmos. Plataforma Web.

Abstract

The study of Theory of Computation is essential for students in computer-related fields, providing the theoretical foundation required to design modern compilers. However, concepts associated with Context-Free Languages (CFLs) and their recognizers, Non-Deterministic Pushdown Automata (NPDAs), frequently pose significant learning barriers due to their high level of abstraction and the inherent complexity of tracking multiple parallel computations. This work presents the development of a web-based educational tool designed for the graphical modeling and real-time simulation of NPDAs. The methodology consisted of developing a client-side (*frontend*) application using vanilla JavaScript and the Cytoscape library for graph rendering. The system enables users to design automata, define distinct acceptance criteria, and dynamically analyze the computation branches generated by non-determinism through structures called *snapshots*. As a result, the platform allows exporting and importing models in JSON format, as well as managing exercises in a completely synchronous and free environment. Although single-thread processing introduces performance constraints when dealing with massive data volumes, the tool successfully fulfills its pedagogical purpose of mitigating traditional static visualization obstacles. Future works include conducting empirical evaluations with students to validate its pedagogical effectiveness, expanding the scope to cover other state machines, and architecting a *backend* for performance optimization.

Keywords: Theory of Computation. Non-Deterministic Pushdown Automata. Educational Tool. Algorithm Visualization. Web Platform.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AFD Autômato Finito Determinístico

AFN Autômato Finito não Determinístico

AFNL Autômato Finito não Determinístico Com Transições λ

ER Expressão Regular

APD Autômato de Pilha Determinístico

APN Autômato de Pilha não Determinístico

LLC Linguagem Livre de Contexto

GLC Gramática Livre de Contexto

MT Máquina de Turing

jFAST Java Finite Automata Simulation Tool

FSMD Finite State Machine Designer

GAM Ginux Abstract Machine

Lista de símbolos

ϵ	Letra grega epsilon
α	Letra grega alfa
β	Letra grega beta
γ	Letra grega gamma
λ	Letra grega lambda
$\in$	Pertence

Sumário

1 Introdução

A teoria da computação estuda as propriedades dos sistemas computacionais. Essa teoria foi formalizada de forma efetiva, pela primeira vez, em 1936, por Alan Turing e Alonzo Church, que construíram as bases do que hoje chamamos de computação por meio da elaboração do conceito de linguagens formais.

Linguagens formais são reconhecidas por meio de máquinas de estados e descritas por meio de gramáticas formais. Neste trabalho, serão enfatizadas as máquinas de estados, mais especificamente o Autômato de Pilha não Determinístico (APN), por apresentarem maior nível de abstração e representação mais complexa do que as demais máquinas.

Sendo a teoria da computação a base das tecnologias atuais, torna-se essencial que os discentes dos cursos da área de computação, como Sistemas de Informação e Engenharia da Computação, dominem esses conceitos para se tornarem profissionais qualificados. Afinal, tais conceitos não apenas fornecem o embasamento teórico necessário, mas também promovem habilidades de pensamento crítico e análise rigorosa, conferindo ao estudante a capacidade de formalizar problemas, definir algoritmos e analisar sua eficiência.

Como forma de auxiliar os discentes a visualizarem de maneira mais clara os autômatos de pilha não determinísticos, este trabalho tem como objetivo desenvolver uma ferramenta web para a modelagem e visualização de APNs em execução.

1.1 Organização do capítulo

Neste capítulo, será abordada a motivação deste trabalho na Seção 1.2, os objetivos propostos na Seção 1.3 e a metodologia adotada para o desenvolvimento do projeto na Seção 1.4.

1.2 O problema de pesquisa

Ciente da importância do estudo de linguagens formais e máquinas de estados — utilizadas hoje como ferramentas fundamentais para o desenvolvimento de compiladores de linguagens de programação modernas —, é imprescindível que os alunos dos cursos superiores da área de computação dominem esses assuntos. Contudo, eles normalmente enfrentam desafios complexos durante esse aprendizado.

Linguagens formais e máquinas de estados são conceitos teóricos rígidos e abstratos. Mesmo com o auxílio de diagramas, o processo de ensino-aprendizagem costuma ser desafiador (PILLAY, 2010), especialmente no que tange à Linguagem Livre de Contexto (LLC).

A máquina reconhecedora dessas linguagens, o Autômato de Pilha Não Determinístico, possui uma série de execuções simultâneas em que cada ramificação carrega seu próprio conjunto de informações (como o topo da pilha e o estado atual). Assimilar toda essa execução paralela de forma puramente teórica constitui um grande obstáculo no aprendizado da disciplina.

1.3 Objetivos

Dadas as dificuldades dos alunos na assimilação das linguagens livres de contexto e de suas respectivas máquinas de estados, este trabalho visa criar uma plataforma web na qual seja possível projetar esses modelos e demonstrar sua execução em tempo real. Desse modo, busca-se tornar viável a visualização detalhada de cada uma das múltiplas execuções simultâneas dos autômatos de pilha não determinísticos.

Para alcançar o objetivo geral, este trabalho possui os seguintes objetivos específicos:

- Criar um software gráfico que auxilie os alunos a visualizar a execução de autômatos de pilha em tempo real;
- Explorar as derivadas propostas por (BRZOZOWSKI, 1964) para a geração de palavras pertencentes a uma linguagem formal;
- Disponibilizar o sistema de forma web e gratuita para facilitar ao máximo o seu acesso.

1.4 Metodologia

A metodologia deste trabalho consiste no desenvolvimento de um sistema de criação e visualização de autômatos e de sua respectiva execução, servindo como um meio didático para que o estudante consiga identificar erros em suas tarefas de forma rápida, visual e automatizada.

1.5 Organização do trabalho

O Capítulo 2 apresenta os conceitos básicos necessários para a compreensão do trabalho, bem como as ferramentas já existentes com fins semelhantes; o Capítulo 3 detalha as ideias gerais da implementação da ferramenta de simulação proposta; o Capítulo 4 discute brevemente os resultados obtidos durante o desenvolvimento; por fim, o Capítulo 5 apresenta as conclusões e as considerações finais deste trabalho.

2 Revisão bibliográfica

2.1 Trabalhos correlatos

Foram encontrados 10 sistemas com objetivos similares ao proposto por este projeto. Ao longo desta seção, cada um deles será exposto, podendo ser visualizada uma comparação simples dos principais pontos de cada um por meio da tabela 1.

O primeiro foi o Language Emulator (VIEIRA; VIEIRA; VIEIRA, 2003), projeto desenvolvido em 2003 na UFMG utilizando a linguagem de programação Java pelos pesquisadores Luiz Filipe Menezes Vieira, Marcos Augusto Menezes Vieira e Newton José Vieira. O projeto se trata de um simulador de autômatos que tem foco nas principais operações envolvendo autômatos finitos determinísticos, autômatos finitos não determinísticos com e sem transição lambda, máquinas de Moore e máquinas de Mealy, juntamente com as gramáticas regulares e as expressões regulares. As operações realizadas por esse sistema incluem transformações entre diferentes tipos de máquinas de estados e gramáticas, assim como a capacidade de determinar se uma palavra pertence ou não a essa gramática. Apesar de ser um projeto bastante completo em relação às suas funcionalidades, ele não apresenta uma interface que detalhe com diagramas essas operações, sendo interativa apenas por meio de botões e textos.

O segundo sistema foi AUTOMATOGRAPH (KIENETZ; CANAL, 2004), desenvolvido por Eduardo Bacchi Kienetz e Ana Paula Canal na UFRGS no ano de 2006 utilizando a linguagem de programação Java. Ele consegue reconhecer palavras de uma linguagem regular pertencentes à linguagem de um autômato informado pelo usuário e gerar uma gramática regular a partir desse autômato; porém, assim como o anterior, não possui uma interface gráfica que represente o diagrama desses autômatos.

Em seguida, o Ginux Abstract Machine (GAM) (JUKEMURA; NASCIMENTO; UCHôA, 2005), que é um sistema desenvolvido por Anibal Jukemura, Hugo Nascimento e Joaquim Uchôa na UFLA no ano de 2005. Tem a capacidade de representar em forma de grafos direcionados Autômato Finito Determinístico (AFD), Autômato Finito não Determinístico (AFN), Autômato Finito não Determinístico Com Transições λ (AFNL), Autômato de Pilha Determinístico (APD), APN e Máquina de Turing (MT), além de reconhecer se uma palavra pertence ou não à máquina de estados representada por esse grafo. O GAM também consegue transformar AFNs em AFDs e minimizar AFDs.

O FSM Simulator (ZUZAK; JANKOVIC, 2025) é um aplicativo web desenvolvido por Ivan Zuzak e Vedrana Janković utilizando HTML, JavaScript e Bootstrap. O FSM Simulator consegue representar o funcionamento e criar aleatoriamente AFDs, AFNs e

expressões regulares. Além disso, consegue criar palavras aleatórias e realizar testes. Sua representação se dá por meio de dígrafos estáticos.

AutoSim ou Automaton Simulator (BURCH, 2008) é um software desenvolvido em Java por Carl Burch na Universidade Carnegie Mellon no ano de 2006. AutoSim é capaz de construir e simular de forma dinâmica AFDs, AFNs, APDs e MTs por meio de grafos direcionados.

Java Finite Automata Simulation Tool (jFAST) (WHITE; WAY, 2006) é um software que foi desenvolvido na Universidade de Villanova por Timothy M. White e Thomas P. Way no ano de 2006 utilizando a linguagem de programação Java e traz a proposta de construir diagramas dinâmicos para simular o comportamento de AFDs, AFNs, APDs e máquinas de Turing.

Finite State Machine Designer (FSMD) é um aplicativo web desenvolvido por Evan Wallace em 2010 com o objetivo de criar com facilidade diagramas que representassem máquinas de estados em geral e portar esses diagramas para vários formatos como PNG, JSON e LaTeX, porém não simula seu funcionamento, apenas desenha as máquinas.

Automaton Simulator (DICKERSON, 2021) é um aplicativo web criado por Kyle Dickerson em 2021 e tem o propósito de simular o comportamento de AFDs, AFNs e APDs por meio de diagramas.

UC Davis Automaton Simulator (UCDAS) (DOTY, 2020) é uma aplicação web criada em 2020 por David Doty utilizando a linguagem de programação Elm na Universidade da Califórnia em Davis e se propõe a ser um simulador que demonstra o comportamento de alguns autômatos e gramáticas, sendo: AFDs, AFNs, Regex, GLCs e MTs. Porém, não possui uma interface com diagramas, sendo sua interatividade realizada por meio de texto e botões.

Por fim, o sistema mais completo é o JFLAP (RENSSELAER; UNIVERSITY, 2018), que teve seu desenvolvimento iniciado em 1990 na Rensselaer por Dan Caugherty e desde então teve a contribuição de vários desenvolvedores utilizando várias tecnologias como C++, Java e JavaScript, tendo o desenvolvimento transferido para a Duke University em 1995 e sua última atualização até o momento em 2018. Atualmente, o JFLAP consegue criar e simular AFD, AFN, APD, APN e MT, além de transformar cada autômato em sua respectiva gramática e transformar autômatos que são equivalentes entre si, como AFD em AFN e outras operações como minimizar AFDs. Ele realiza todas essas funções com diagramas e elementos gráficos que facilitam bastante o entendimento.

Após visitar estes sistemas, foi possível perceber que apenas o simulador GAM simulava a execução de autômatos de pilha não determinísticos; porém, ele não possuía uma visualização satisfatória do não determinismo desse tipo de autômato, pois cada execução paralela possui sua própria pilha, estado atual e leitura de caractere da palavra, sendo difícil

Sistema	Ano	Linguagem	Idioma	Autômatos	Operações	Interface
Language Emulator	2003	Java	Português	3	Sim	Apenas texto
AUTOMATOGRAPH	2004	Java	Português	2	Não	Apenas texto
GAM	2005	-	Português	6	Sim	Gráfica estática
FSM Simulator	-	JavaScript	Inglês	2	Não	Gráfica estática
AutoSim	2006	Java	Inglês	4	Não	Gráfica dinâmica
jFAST	2006	Java	Inglês	4	Não	Gráfica dinâmica
FSMD	2010	JavaScript	Inglês	1	Não	Gráfica dinâmica
Automaton Simulator	2021	JavaScript	Inglês	3	Não	Gráfica dinâmica
UCDAS	2020	Elm	Inglês	4	Não	Apenas texto
JFLAP	1990-2018	-	Inglês	5	Sim	Gráfica dinâmica

Tabela 1 – Trabalhos relacionados

visualizar tantos dados simultaneamente de cada uma das execuções paralelas. Portanto, optou-se por criar um sistema que tivesse como um dos pontos principais representar esse não determinismo de forma aprimorada.

2.2 Expressões regulares

Esta seção apresenta uma rápida introdução ao conceito de expressões regulares, incluindo a definição, e um sumário das propriedades de linguagens regulares. Como este trabalho se destina principalmente às linguagens livres de contexto, não entraremos em detalhes técnicos e formais a respeito desta matéria. O principal objetivo desta seção é prover ao leitor o conhecimento mínimo necessário para o conceito de derivadas que é apresentado a seguir, já que os autores consideram ser mais natural abordar o tema de derivadas em linguagens regulares antes de se apresentar o conceito sobre derivadas de linguagens livres de contexto.

Dado um alfabeto Σ, uma Expressão Regular (ER) sobre Σ é definida indutivamente

pelo seguinte conjunto de regras:

- $\emptyset$ denota a linguagem vazia.
- A palavra vazia ϵ é uma ER.
- Se $a \in \Sigma$, então a é uma ER.
- Se e_1 e e_2 são expressões regulares, então a concatenação $e_1\, e_2$ e a união $e_1 \,|\, e_2$ também são expressões regulares.
- Se e é uma expressão regular, então o fecho de Kleene e^* também é uma expressão regular.
- Apenas são expressões regulares aquelas obtidas por uma sequência de aplicações finitas das regras anteriormente descritas.

Sintaticamente, parênteses podem ser empregados para desambiguar o significado das expressões. Vamos considerar como ordem de precedência, da mais alta para a mais baixa, os operadores Kleene, sequência e alternativa. Por exemplo, $ab|cd*$ é equivalente a $(ab)|c(d*)$.

A linguagem denotada por uma expressão regular e, dada pela notação $L(e)$, pode ser expressa, por meio de equivalência à teoria de conjuntos, da seguinte forma:

- $L(\emptyset) = \{\}$. A ER para a linguagem vazia denota o conjunto vazio.
- $L(\epsilon) = \{\epsilon\}$, onde ϵ é a palavra vazia. A ER para a palavra vazia denota um conjunto cujo único elemento é a palavra vazia (ou equivalentemente o conjunto unitário vazio).
- $L(a), \ \texttt{com}\ a \in \Sigma = \{a\}$. A ER para um símbolo a é um conjunto contendo apenas o referido símbolo.
- $e_1\, e_2 = \{w_1 w_2 \mid w_1 \in L(e_1) \wedge w_2 \in L(e_2)\}$. A ER para a concatenação de duas linguagens é o conjunto formado pela justaposição de uma palavra de e_1 seguida por uma palavra de e_2, nessa ordem.
- $e_1 \,|\, e_2 = L(e_1) \cup L(e_2)$. A ER para a alternativa entre duas ERs e_1 e e_2 é dada pela união das palavras em $L(e_1)$ e $L(e_2)$.
- $e^* = \{\epsilon\} \cup L(e) \cup L(e\, e) \cup L(e\, e\, e) \cup \ldots$. O fecho de Kleene pode ser entendido como uma união de concatenações da expressão e consigo mesma, de zero até infinitas vezes.

Considere a expressão regular $(a \mid b) * a$. Podemos facilmente verificar que a palavra $a\,b\,a$ pertence à linguagem gerada por essa expressão regular. Mas a palavra b não pertence.

$$(a \mid b) * a$$

$$(a \mid b)\,(a \mid b) * a$$

$$(a \mid b)\,(a \mid b) * a$$

$$(a \mid b)\,(a \mid b)\,\epsilon\,a$$

$$a\,(a \mid b)\,\epsilon\,a$$

$$a\,b\,\epsilon\,a$$

$$a\,b\,a$$

Expressões regulares são particularmente úteis em diversos sistemas computacionais como um meio de pesquisa avançada em textos. Por exemplo, o comando grep (Free Software Foundation, 2026) do Linux, que permite listar apenas as linhas de um arquivo onde um padrão dado por uma expressão ocorre.

Como resultados consolidados da teoria da computação, sabe-se que expressões regulares denotam o mesmo conjunto das linguagens regulares, o mesmo reconhecido por autômatos finitos determinísticos (AFD) e autômatos finitos não determinísticos (AFN). Também sabe-se que o conjunto das linguagens regulares é fechado sob uma coleção de operações como união, interseção, complemento, concatenação, reverso, intercalação, entre outras. Esses resultados podem ser facilmente encontrados em livros de teoria, como (VIEIRA, 2006) e (HOPCROFT; MOTWANI; ULLMAN, 2006), e não serão replicados neste trabalho.

2.3 Linguagens Livres de Contexto

Linguagens livres de contexto LLC, são uma classe de linguagens formais que podem ser geradas por gramáticas livres de contexto e podem ser reconhecidas por autômatos de pilha. Linguagens livres de contexto são fechadas sob as operações de união, concatenação e fecho de Kleene, ou seja, a aplicação dessas operações resulta em uma linguagem que certamente pertence ao grupo das LLCs. Porém, LLCs não são fechadas sob as operações de interseção e complemento, então não há garantias de que a linguagem resultante dessas operações continue sendo livre de contexto.

2.3.1 Autômatos de Pilha

Um APD é uma máquina de estados formalmente denotada por uma sêxtupla $\langle E, \Sigma, \Gamma, \delta, i, F \rangle$ em que:

- E é um conjunto finito de elementos, ditos estados.
- Σ, também chamado de alfabeto, é um conjunto finito de elementos distintos, ditos símbolos.
- Γ, também chamado de alfabeto da pilha, é um conjunto finito de elementos distintos, ditos símbolos.
- $\delta : E \times \Sigma_\lambda \times \Gamma_\lambda \to E \times \Gamma_\lambda$

Uma configuração instantânea tem como função demonstrar o estado atual do autômato a cada passo de sua execução. Em um APD, é uma tripla $[e, y, z]$ em que e é o estado atual, y é a palavra que ainda não foi lida e z é a palavra que representa o conteúdo da pilha.

Uma das linguagens que é possível reconhecer com um APD é a linguagem $\{a^n b^n \mid n \geq 0\}$, que pode ser definida formalmente pela seguinte sêxtupla: $(\{1, 2\}, \{a, b\}, \{X\}, \delta, 1, \{1, 2\})$, em que δ é dada por:

$\delta(1, a, \lambda) = [1, X]$;

$\delta(1, b, X) = [2, \lambda]$;

$\delta(2, b, X) = [2, \lambda]$;

Para a palavra $aabb$, a execução do APD é dada por:

$[1, aabb, \lambda]$

$\vdash [1, abb, X]$

$\vdash [1, bb, XX]$

$\vdash [2, b, X]$

$\vdash [2, \lambda, \lambda]$

A diferença entre um autômato de pilha determinístico e um não determinístico é a presença de transições compatíveis, ou seja, duas ou mais transições que podem ser aplicadas a uma mesma configuração instantânea do autômato, gerando duas configurações

instantâneas diferentes. Isso acontece quando existem duas ou mais transições que possuem o mesmo estado de origem, o mesmo símbolo lido da palavra e o mesmo símbolo no topo da pilha.

Uma das linguagens que é possível reconhecer com um APN é a linguagem $\{w \in \{0,1\}^* \mid n_0(w) = n_1(w)\}$, que pode ser definida formalmente pela seguinte sêxtupla: $(\{=, \neq\}, \{0,1\}, \{Z, U\}, \delta, 1, \{=, =\})$, em que δ é dada por:

$$\delta(=, 0, \lambda) = [\neq, Z];$$

$$\delta(=, 1, \lambda) = [\neq, U];$$

$$\delta(\neq, 0, Z) = [\neq, ZZ];$$

$$\delta(\neq, 0, U) = [\neq, \lambda];$$

$$\delta(\neq, 1, U) = [\neq, UU];$$

$$\delta(\neq, 1, Z) = [\neq, \lambda];$$

$$\delta(\neq, \lambda, \lambda) = [=, \lambda];$$

Para a palavra 0101, a execução do APN é dada por:

$$[=, 0101, \lambda]$$

$$\vdash [\neq, 101, Z]$$

$$\vdash [\neq, 01, \lambda]$$

$$\vdash [\neq, 1, U]$$

$$\vdash [\neq, \lambda, \lambda]$$

A linguagem aceita por um APD/APN pode ser definida de uma das seguintes formas:

- **Aceitação por estado final** : Conjunto de todas as palavras tais que o APD/APN consome toda a palavras e termina em estado final.
- **Aceitação por pilha vazia** : Conjunto de todas as palavras tais que o APD/APN consome toda a palavras e termina com pilha vazia.
- **Aceitação por estado final e pilha vazia**: Conjunto de todas as palavras tais que o APD consome toda a palavras e termina em estado final e com pilha vazia.

No caso particular de um APN considera-se que se qualquer caminho de computação satisfaz a condição de aceitação escolhida, então a palavra é aceita.

A possibilidade de transições compatíveis faz com que APNs sejam máquinas de estados mais poderosas que os APDs, de forma que os APDs não reconhecem todo o conjunto de linguagens livres de contexto, enquanto os APNs reconhecem. Por exemplo, a linguagem de palíndromos $\{ww^r \mid |w| \geq 0\}$ (em que a notação w^r é usada para denotar o reverso de w) é uma linguagem que não pode ser reconhecida por um APD, mas pode ser reconhecida por um APN.

2.3.2 Gramáticas livres de contexto

Uma Gramática Livre de Contexto (GLC) é um conjunto de regras que descrevem a formação de palavras em uma linguagem livre de contexto, em que cada regra da gramática tem no lado esquerdo um único símbolo não terminal e, no lado direito, uma sequência de símbolos terminais e não terminais. As GLCs podem ser representadas formalmente por uma quádrupla (V, Σ, R, S), em que V é um conjunto finito de símbolos não terminais, Σ é um conjunto finito de símbolos terminais, R é um conjunto finito de regras de produção e S é o símbolo inicial da gramática.

A linguagem livre de contexto $\{0^n 1^n \mid n \geq 0\}$, por exemplo, pode ser gerada pela GLC $G = (\{P\}, \{0, 1\}, R, P)$, em que R é dado por:

$$P \rightarrow 0P1 \mid \lambda$$

O ato de gerar uma palavra a partir de uma GLC é chamado de derivação, que consiste em aplicar as regras da gramática repetidamente partindo do símbolo inicial. Um exemplo de derivação para a palavra 0011 é dado por:

$$\begin{array}{c} P \\ | \\ 0\ P\ 1 \\ | \\ 0\ P\ 1 \\ | \\ \lambda \end{array}$$

2.4 Derivadas de Expressões Regulares

Derivadas de linguagens regulares foram primeiramente descritas por Brzozowski (BRZOZOWSKI, 1964) como um meio tanto para determinar a pertinência de uma palavra a um conjunto quanto como forma de deduzir um AFD diretamente a partir de uma expressão regular. Uma derivada de uma linguagem L em relação a um símbolo a pode ser definida como:

$$d(a, L) = \{w \mid aw \in L\} \tag{2.1}$$

Uma derivada em relação a um símbolo a é um conjunto de palavras (e, portanto, uma linguagem) de todos os sufixos w tais que aw faz parte da linguagem original. Antes de definirmos formalmente a derivada, vamos primeiro apresentar a definição da função $null : e \to \{T, F\}$ que, dada uma expressão regular e, retorna T se e produz a palavra vazia ou F caso contrário. Chamamos expressões regulares que podem produzir a palavra vazia de anuláveis.

$$\begin{aligned}
&null_g(\epsilon) = T \\
&null_g(a) = F \\
&null_g(\alpha\,\beta) = null_g(\alpha) \wedge null_g(\beta) \\
&null_g(\alpha \mid \beta) = null_g(\alpha) \vee null_g(\beta) \\
&null_g(v) = null_g(\alpha), \quad \texttt{se } v \to \alpha \in R
\end{aligned}$$

A derivada de uma expressão regular pode ser computada por meio de um conjunto de regras de equivalência que computam uma ER para a linguagem derivada. Considere a expressão que denota a linguagem que contém apenas a palavra vazia ϵ. Pela definição de derivada (2.1), podemos concluir que não existe nenhuma palavra w tal que, para qualquer símbolo do alfabeto, $aw \in \{\epsilon\}$; logo, $d(a, \epsilon) = \emptyset$, para qualquer símbolo $a \in \Sigma$.

O mesmo raciocínio pode ser usado em ER de um único símbolo do alfabeto. Nesse caso, seja $d(a, b)$, em que a e b são símbolos do alfabeto. Se $b = a$, então a derivada deve ser ϵ, pois $b\,\epsilon \equiv b \in \{b\}$. Por outro lado, se $a \neq b$, então não existe um sufixo w tal que $aw \in \{b\}$.

Um cuidado especial deve ser tomado com a concatenação devido à possibilidade de uma das expressões produzir uma palavra vazia. Ao determinar $d(a, e_1\,e_2)$, se e_1 é anulável, devemos considerar a possibilidade de o símbolo a também ser um prefixo de e_2. Caso contrário, a derivada da sequência será a concatenação $d(a, e_1)\,e_2$. Como a alternativa representa a união de duas linguagens, a derivada da alternativa é expressa como a alternativa das derivadas das respectivas subexpressões.

Como o fecho de Kleene pode ser visto como uma união de concatenações, a derivada para uma e^* pode ser vista como a derivada para ee^*. A derivada de uma expressão regular em relação a um símbolo a pode ser definida como uma função $d : \Sigma \times e \to e$. Essa função é definida a seguir:

$$
\begin{aligned}
&d(a, \epsilon) = \emptyset \\
&d(a, a) = \epsilon \\
&d(a, b) = \emptyset, \ \texttt{se} \ a \neq b \\
&d(a, e_1\, e_2) = \begin{cases} d(a, e_1)\, e_2, \ \texttt{se} \ \neg null(e_1) \\ d(a, e_1)\, e_2 \mid d(a, e_2), \ \texttt{se} \ null(e_1) \end{cases} \\
&d(a, e_1 \mid e_2) = d(a, e_1) \mid d(a, e_2) \\
&d(a, e^*) = d(a, e)\, e^*
\end{aligned}
$$

Como exemplo, vamos derivar a linguagem $(a|\epsilon)bc$ em relação ao símbolo b.

$$
\begin{aligned}
&d(b, (a \mid \epsilon)\, b\, c) \equiv \\
&d(b, (a \mid \epsilon))\, b\, c \mid d(b, b\, c) \equiv \\
&(d(b, a) \mid d(b, \epsilon))b\, c \mid \epsilon d(b, b\, c) \equiv \\
&(\emptyset \mid \emptyset)b\, c \mid d(b, b\, c) \equiv \\
&\emptyset \mid c \equiv c
\end{aligned}
$$

Pode-se determinar se uma dada palavra formada pelos símbolos $a_1...a_n$ é gerada pela expressão regular e aplicando sucessivamente a derivada e testando se a expressão resultante é anulável, ou seja, $a_1...a_n \in L(e)$ se e somente se a expressão regular $d(a_n, d(a_{n-1}, \ldots d(a_1, e)) \ldots)$ for anulável.

2.5 Derivadas para Linguagens Livres de Contexto

O conceito de derivadas pode ser estendido para linguagens livres de contexto, adicionando-se os casos necessários para a derivação de símbolos não terminais que estão presentes nessas gramáticas. Uma GLC é formada por $\langle \Sigma, V, R, P \rangle$.

Essa mudança afeta tanto a definição de anulável quanto a definição de derivadas. Intuitivamente, a nova definição de anulável para GLC deve receber uma forma sentencial e retornar `T` se essa forma puder derivar ϵ. Vamos usar letras gregas do início do alfabeto

(α e β) para denotar formas sentenciais da gramática e a letra v para denotar um não terminal. Logo, a nova definição de anuláveis seria $null_g : \{\Sigma \cup V\}^* \to \{T, F\}$:

$$
\begin{aligned}
&null_g(\epsilon) = T \\
&null_g(a) = F \\
&null_g(\alpha\,\beta) = null_g(\alpha) \wedge null_g(\beta) \\
&null_g(\alpha \mid \beta) = null_g(\alpha) \vee null_g(\beta) \\
&null_g(v) = null_g(\alpha), \quad \texttt{se } v \to \alpha \in R
\end{aligned}
$$

Subsequentemente, a definição de derivada se torna:

$$
\begin{aligned}
&d_g(a, \epsilon) = \emptyset \\
&d_g(a, a) = \epsilon \\
&d_g(a, b) = \emptyset, \quad \texttt{se } a \neq b \\
&d_g(a, \alpha\,\beta) = \begin{cases} d_g(a, \alpha)\,\beta, & \texttt{se } \neg null_g(\alpha) \\ d_g(a, \alpha)\,\beta \mid d_g(a, \beta), & \texttt{se } null_g(\alpha) \end{cases} \\
&d_g(a, \alpha \mid \beta) = d_g(a, \alpha) \mid d_g(a, \beta) \\
&d_g(a, v) = d_g(a, \alpha) \quad \texttt{se } v \to \alpha \in R
\end{aligned}
$$

Essas definições são matematicamente corretas, mas, como observado por Might (MIGHT; DARAIS; SPIEWAK, 2011), computacionalmente elas podem levar à recursão infinita caso a gramática possua recursão à esquerda. A implementação proposta por Might tem como objetivo o uso de derivadas para parsing, e emprega o uso de linguagem funcional com suporte a avaliação lazy e memoização para contornar esses problemas. Além desses problemas, Might também precisou implementar simplificações após cada passo da derivada para evitar o crescimento rápido da gramática e a degradação do tempo de parsing.

Uma segunda abordagem para derivadas, proposta por Henriksen (HENRIKSEN; BILARDI; PINGALI, 2019), descreve um método com custo quadrático em espaço e cúbico em tempo para a realização de parsing. A abordagem consiste em construir uma nova GLC a partir da gramática original. Além disso, cada não terminal é anotado com a sequência de símbolos sobre os quais foi derivado até um dado momento. Essa abordagem permite que se apliquem derivadas a gramáticas com recursão à esquerda. Uma segunda otimização é a eliminação de regras desnecessárias, que são inatingíveis a partir do símbolo de partida.

Para formalizar a abordagem de Henriksen, definimos a construção de uma gramática derivada $G_a = \langle \Sigma, V', R', S_a \rangle$ a partir da gramática original G, em relação

a um símbolo de entrada $a \in \Sigma$. O novo conjunto de não terminais V' passa a conter símbolos anotados com o histórico de derivação (como v_a).

A geração das novas regras baseia-se em uma função de derivação simbólica $\mathcal{D}_a$, que mapeia o lado direito das regras originais para o lado direito das novas regras. Em estilo declarativo, para cada regra na gramática original, geramos uma regra correspondente na gramática derivada:

$$\frac{v \to \alpha \in R}{v_a \to \mathcal{D}_a(\alpha) \in R'}$$

Onde $\mathcal{D}_a$ é definida por casos, dependendo se o primeiro símbolo da forma sentencial é um terminal ($b \in \Sigma$) ou um não terminal ($u \in V$):

$$\mathcal{D}_a(\epsilon) = \emptyset$$

$$\mathcal{D}_a(b\,\beta) = \begin{cases} \beta, \texttt{se } a = b \\ \emptyset, \texttt{se } a \neq b \end{cases}$$

$$\mathcal{D}_a(u\,\beta) = \begin{cases} u_a\,\beta \mid \mathcal{D}_a(\beta), \texttt{se } null_g(u) \\ u_a\,\beta, \texttt{se } \neg null_g(u) \end{cases}$$

$$\mathcal{D}_a(\alpha \mid \beta) = \mathcal{D}_a(\alpha) \mid \mathcal{D}_a(\beta)$$

A grande vantagem dessas regras está no tratamento de não terminais (os casos envolvendo $u\,\beta$). Em vez de expandir o não terminal u recursivamente — o que causaria o loop infinito em gramáticas com recursão à esquerda —, o método introduz o novo não terminal u_a (que simboliza "u derivado por a"). Isso posterga a avaliação e transfere a resolução da recursão para a própria estrutura da nova gramática, transformando o problema de parsing em sucessivas reescritas da GLC.

Exemplo de Derivação Simbólica

Considere uma gramática muito simples com recursão à esquerda para expressões matemáticas:

$$\begin{aligned} S &\to S + T \mid T \\ T &\to x \end{aligned}$$

Vamos calcular a gramática derivada para o símbolo de entrada x. Sabendo que nem S nem T são anuláveis ($\neg null_g(S)$ e $\neg null_g(T)$), aplicamos a função $\mathcal{D}_x$ às regras originais para obter os novos não terminais S_x e T_x:

1. **Regra** $S \to S + T$**:** Como S é não terminal e não anulável, geramos $S_x \to \mathcal{D}_x(S + T) \Rightarrow S_x \to S_x + T$.
2. **Regra** $S \to T$**:** Como T é não terminal e não anulável, geramos $S_x \to \mathcal{D}_x(T) \Rightarrow S_x \to T_x$.
3. **Regra** $T \to x$**:** Como x é um terminal e coincide com o símbolo de entrada, geramos $T_x \to \mathcal{D}_x(x) \Rightarrow T_x \to \epsilon$.

A gramática derivada resultante (focando nos estados atingíveis a partir do novo símbolo inicial S_x) é finita e bem definida:

$$
\begin{aligned}
S_x &\to S_x + T \mid T_x \\
T_x &\to \epsilon \\
T &\to x \quad \texttt{(mantida pois é referenciada em } S_x\texttt{)}
\end{aligned}
$$

Este exemplo ilustra como a abordagem de Henriksen contorna a recursividade à esquerda ao gerar o símbolo S_x de forma simbólica, evitando a expansão contínua que ocorreria nas derivadas estritas.

3 Desenvolvimento

A proposta da ferramenta é permitir que o aluno explore o funcionamento de um APN, com clareza a respeito do seu não determinismo. Adicionalmente, pretende-se que o aluno seja capaz de testar a corretude de um autômato. Este último passo requer uma especificação da linguagem que o APN deve aceitar. Para tanto, empregamos GLCs como uma forma de especificar a linguagem a ser implementada.

O processo de correção automatizado consiste na geração de palavras que devem ser aceitas pelo APN. Para esta etapa, empregamos um método baseado em derivadas livres de contexto.

Este capítulo apresenta como a ferramenta foi desenvolvida e descreve a arquitetura de software utilizada, assim como as bibliotecas e técnicas de programação. Ele está organizado da seguinte forma: a seção 3.1 descreve a implementação da ferramenta; a subseção 3.1.1 descreve de forma geral como o APN e sua execução foram modelados; por fim, a seção 3.1.2 apresenta a implementação do gerador de palavras a partir de uma dada GLC.

3.1 Implementação

A ferramenta foi desenvolvida utilizando a linguagem de programação JavaScript, com foco na orientação a objetos, com o auxílio de HTML5 e CSS para a sua apresentação visual. A aplicação ocorre completamente no *front-end* e não possui partes executadas em servidor. Também não foi utilizado nenhum tipo de *framework* para auxiliar na organização do código. Além das tecnologias nativas, foi utilizada a biblioteca JavaScript Cytoscape.

A biblioteca Cytoscape é utilizada para a manipulação de desenhos em HTML5, com foco na visualização de grafos. Ela foi escolhida por ser totalmente *open source* e porque grafos são a forma mais comum de representação de autômatos, utilizada, por exemplo, no livro didático (VIEIRA, 2006). A biblioteca foi empregada apenas para o desenho dos grafos, tendo toda a parte de simulação de autômatos sido desenvolvida exclusivamente durante este trabalho. O código-fonte está disponível no GitHub, e o software encontra-se hospedado de forma gratuita por meio do GitHub Pages para testes.

O sistema desenvolvido originalmente possuía os seis tipos de máquinas de estados estudados na disciplina de Fundamentos Teóricos da Computação na UFOP. Porém, devido ao tempo limitado e ao escopo abrangente, foi tomada a decisão de seguir apenas com a representação de APNs e a correção de exercícios. Caso haja interesse, essa versão beta pode ser encontrada neste repositório, ao passo que o código mais recente está disponível

neste repositório.

3.1.1 Modelagem do APD e sua execução

Ao desenvolver este projeto, foi feita uma divisão entre a representação lógica e a representação gráfica do APN. A representação lógica do APN e a sua execução dentro do sistema acontecem por meio de quatro classes principais: `Transition`, `APN`, `Snapshot` e `APNRunner`. Ao passo que as classes `Transition` e `APN` descrevem os aspectos estáticos do autômato, as classes `Snapshot` e `APNRunner` são usadas para emular o comportamento do autômato sobre uma dada entrada.

Cada transição foi representada utilizando objetos da classe `Transition`, que possui quatro atributos simples: `sym`, que é o caractere a ser lido na palavra; `stkR`, que é o valor que deverá ser desempilhado da pilha; `stkW`, que é o valor que deverá ser empilhado na pilha; e o atributo `state`, que é o estado de destino daquela transição.

A classe `APN` ficou responsável por reunir os estados e as transições, além de alguns métodos auxiliares. Essa classe contém os seguintes campos privados:

- `#states`: Uma lista de estados. Cada estado é representado por um valor numérico.
- `#transitions`: Um mapa que associa a cada estado, uma lista de objetos que representam transições.
- `#finals`: Uma lista de estados finais.
- `#start`: Uma lista de estados iniciais.

O método `addState(s)` permite adicionar um novo estado, desde que `s` seja um número e já não esteja presente no conjunto de estados. A função `removeState(s)` procura por um estado e o remove da lista de estados, observando a necessidade de se atualizar a lista de transições de todos os demais estados, a lista de estados finais e iniciais. A função `addTransition(s,t)` adiciona um objeto de transição `t` à lista de transições de `s`. A função `removeTransition(s,t)` remove o objeto de transição cuja referência é dada por `t` da lista de transições do estado `s`. Além desses métodos, há métodos para definir e obter estados iniciais e finais. Tais métodos são triviais (e bem documentados na implementação).

A classe `APN` possui ainda o método `delta`, que utiliza os atributos do seu mapa de transições para determinar quais transições podem ser atingidas a partir de um estado s, um caractere de palavra *chr* e um topo de pilha r, retornando esse conjunto de transições na forma de um *array*. Esse método percorre cada transição que tem sua chave no mapa igual a s (já que sua chave no mapa é seu estado de origem) e, em cada uma, verifica se o atributo `sym` é igual a *chr* ou à string vazia, e se o atributo `stkR` é igual a r ou à

string vazia. Se ambas as condições forem satisfeitas simultaneamente, essa transição é considerada válida e é adicionada ao *array* que será retornado ao final do método.

O maior desafio ao representar APNs neste projeto foi modelar o seu não determinismo, dado que existem muitas execuções simultâneas do mesmo autômato. A forma adotada para representar cada uma das execuções simultâneas causadas pelo não determinismo foi a criação da classe `Snapshot`, na qual cada `Snapshot` representa um ponto da execução de um dos possíveis caminhos da máquina, podendo vários `Snapshots` existir simultaneamente.

Para modelar isso, a classe `Snapshot` conta com os seguintes atributos: `state`, que representa o estado em que a execução se encontra; `pos`, que representa a posição do caractere lido na palavra que está sendo processada; e `stack`, que representa o array com o conteúdo da pilha naquele momento da execução. Além de armazenar os dados, a classe `Snapshot` dispõe de métodos utilitários essenciais, como o `top()`, que retorna o valor atual no topo da pilha (ou a *string* vazia, caso não haja elementos), e o método `equal(snp)`, que realiza a comparação profunda de conteúdo entre duas instâncias, garantindo que configurações instantâneas sejam consideradas idênticas apenas se possuírem o mesmo estado, a mesma posição de leitura e o mesmo arranjo na pilha.

Para a visualização do usuário, é criado um grafo de *snapshots* construído na classe `Graph`. Essa classe pode ser utilizada para a criação de grafos de qualquer tipo de objeto JavaScript, sendo composta por dois mapas: um que tem como chave um ID e como valor um único objeto que representa um nó; e um segundo mapa, que representa as arestas entre os nós utilizando os valores de ID do primeiro mapa, no qual a chave é o nó de origem e o valor é o nó de destino da aresta.

A classe `APNRunner` é a principal responsável por emular o comportamento dinâmico do autômato sobre uma dada entrada. Esta classe contém os seguintes campos privados principais:

- `#apn`: Um objeto da classe `APN` que será executado.
- `#input`: A palavra (cadeia de caracteres) que será processada.
- `#snaps`: Um *array* de objetos da classe `Snapshot`, armazenando as configurações instantâneas atuais.
- `#graph`: O grafo rastreador da execução, construído para visualização.
- `#accType`: O tipo de critério de aceitação esperado (`'F'`, `'S'` ou `'FS'`).
- `#limit`: O limite de *loops* estipulado para a simulação, prevenindo execuções infinitas.

Na sua inicialização, por meio do método `start()`, a classe prepara a simulação iterando sobre os estados iniciais do autômato para instanciar os *snapshots* de partida, registrando-os imediatamente como os primeiros nós no grafo.

O principal método encarregado de avançar a execução é o `step`. Ele percorre cada um dos *snapshots* atuais e verifica as transições válidas a partir deles por meio do método `delta` da classe `APN`. Para processar a mudança de estado e manipular a pilha, o `step` utiliza a função auxiliar `transition(t, snp)`. Essa função recebe um objeto de transição e um *snapshot*, verifica se a regra consome um caractere da palavra (ou se é uma transição vazia) e, se aplicável, gera um novo *snapshot* devidamente atualizado — desempilhando e empilhando os símbolos definidos pela transição e avançando o ponteiro de leitura da entrada. As transições válidas resultantes passam a ser os *snapshots* atuais e são adicionadas ao grafo, criando-se uma aresta entre cada *snapshot* original e os gerados a partir dele.

A simulação acontece por meio do método `runUntilAcc()`, que realiza sucessivas chamadas à função `step` até que uma de três condições de parada seja atingida: (I) todos os caminhos possíveis se esgotam (*snapshots* atuais ficam vazios); (II) o limite de *loops* de execução definido pelo usuário é atingido; ou (III) a palavra é aceita. Alternativamente, a execução pode ser controlada passo a passo através do método `next()`.

Por fim, a verificação de aceitação da palavra é flexível e feita a cada etapa. A classe `APNRunner` oferece três métodos distintos baseados na propriedade `accType`: o método `acceptedF()`, que verifica se algum *snapshot* concluiu a leitura da palavra e parou em um estado final; o `acceptedS()`, que checa se a palavra foi totalmente lida e a pilha se encontra vazia; e o `acceptedFS()`, que exige rigorosamente a satisfação de ambas as condições simultaneamente.

3.1.1.1 Exemplo de Uso: Linguagem $a^n b^n$

Para ilustrar o funcionamento integrado dessas classes, apresenta-se a seguir a instanciação de um Autômato de Pilha (neste caso, determinístico) projetado para reconhecer a linguagem $L = \{a^n b^n \mid n > 0\}$. O autômato possui três estados: o estado 0 empilha o caractere ‘A‘ ao ler o primeiro ‘a‘; o estado 1 continua empilhando ‘A‘ para leituras subsequentes de ‘a‘ e transita para o estado 2 ao ler o primeiro ‘b‘ (desempilhando); e o estado 2 continua desempilhando ao ler o caractere ‘b‘. A aceitação ocorre por estado final (2) e pilha vazia (`FS`).

O fragmento de código em JavaScript a seguir demonstra a construção da lógica e o laço para a sua execução passo a passo utilizando o método `next()`:

```
1 import { APN, Transition, APNRunner } from './APN.js';
2
3 // 1. Criação do autômato e definição dos estados
```

```
let apn = new APN();
apn.addState(0);
apn.addState(1);
apn.addState(2);

apn.addInitialState(0);
apn.addFinalState(2);

// 2. Definição das transições
// Estado 0: Lê o primeiro 'a', empilha 'A' e vai para o estado 1
apn.addTransition(0, new Transition('a', '', ['A'], 1));

// Estado 1: Lê 'a' adicionais, empilha 'A' e continua no 1
apn.addTransition(1, new Transition('a', '', ['A'], 1));

// Estado 1: Lê o primeiro 'b', desempilha 'A' e vai para o estado 2
apn.addTransition(1, new Transition('b', 'A', [], 2));

// Estado 2: Lê 'b' adicionais, desempilha 'A' e continua no 2
apn.addTransition(2, new Transition('b', 'A', [], 2));

// 3. Configuração da execução
let input = "aabb";
let accType = "FS"; // Aceitação por Estado Final e Pilha Vazia
let limit = 20;

let runner = new APNRunner(apn, input, accType, limit);
console.log("Iniciando execução para a palavra: " + input);

// 4. Laço de execução passo a passo
while (!runner.exausted() && !runner.acceptedFS()) {
    runner.next();
    // Imprime o estado das configurações atuais e passadas
    console.log(runner.lastStepString());
}
// 5. Verificação do critério de aceitação
if (runner.acceptedFS()) {
    console.log("Palavra aceita pelo autômato!");
} else {
    console.log("Palavra rejeitada!");
}
```

3.1.2 Gerando palavras a partir de uma GLC

O processo para geração de palavras consiste no uso de derivadas para sintetizar uma palavra a partir de uma GLC. Essa abordagem é simples e tem a vantagem de garantir,

por definição, que a palavra sintetizada deverá necessariamente ser aceita por qualquer APN que implemente a respectiva GLC. A técnica consiste em, iterativamente, derivar a GLC em relação a um símbolo proveniente do conjunto *First* do símbolo de partida. A cada passo, inspecionamos se o símbolo de partida é anulável; em caso afirmativo, a concatenação de todos os símbolos derivados até o momento forma uma palavra de entrada válida que deve ser aceita pelo APN.

Intuitivamente, o conjunto *First*, frequentemente empregado em reconhecedores sintáticos LL(1) (AHO et al., 2006), pode ser entendido como o conjunto de todos os símbolos terminais que podem ser produzidos imediatamente por uma forma sentencial. Por exemplo, uma regra na forma $A \rightarrow aAb \mid c$ possui como conjunto *First* os elementos $\{a, c\}$. Dada uma gramática $G = \langle \Sigma, V, R, P \rangle$, seja $F(A)$ o conjunto *First* do não terminal A. Inicialmente, para cada $A \in V$, $F(A) = \emptyset$. Formalmente, o conjunto *First(X)* pode ser definido pela aplicação das seguintes regras, repetidas até que nenhum dos conjuntos sofra alterações:

1. Se X é um terminal, então $First(X) = \{X\}$.
2. Se X é um não terminal com regra $X \rightarrow Y_1\, Y_2\, \ldots\, Y_k$ (para algum $k \geq 1$), e se existe algum j tal que $\neg nullable(Y_j)$ com $0 \leq j \leq k$, então $First(X) = First(X) \cup First(Y_i)$ para $0 \leq i \leq j$.

O conjunto de não terminais anuláveis (*nullables*) de uma GLC é o conjunto de todos os não terminais que podem gerar a palavra vazia (ϵ). Esse conjunto pode ser computado pela aplicação das seguintes regras até que o conjunto pare de mudar:

1. Para toda produção na forma $A \rightarrow \epsilon$, temos que $A \in nullables$.
2. Para toda produção na forma $A \rightarrow B_1 \ldots B_n$, se $B_1 \ldots B_n \in nullables$, então $A \in nullables$.

A implementação desses conceitos em JavaScript se beneficia do uso do tipo de dados `Set` para a implementação de *nullables*, e de `Map` para a implementação do *First*. Esses conceitos foram codificados em uma classe que modela uma GLC, formada por um `Map` que armazena as produções da gramática. A API desenvolvida permite a adição de regras e a definição do símbolo de partida, além de disponibilizar o método `derivate`, que cria uma cópia da gramática contendo as regras derivadas.

A implementação do método da derivada também copia a tabela *First* da gramática original. Como o *First* é computado por um algoritmo de ponto fixo e todas as produções da gramática base são copiadas para a derivada, o novo conjunto *First* não precisa ser computado do zero, baseando-se no conjunto original. Isso permite que apenas adicionemos

à tabela as novas produções mais relevantes. A API da gramática contém ainda o método `prune`, responsável por remover produções inatingíveis a partir do símbolo de partida.

Finalmente, o processo para a geração de uma palavra pode ser descrito pela seguinte sequência de passos, dada a gramática $G = \langle \Sigma, V, R, P \rangle$:

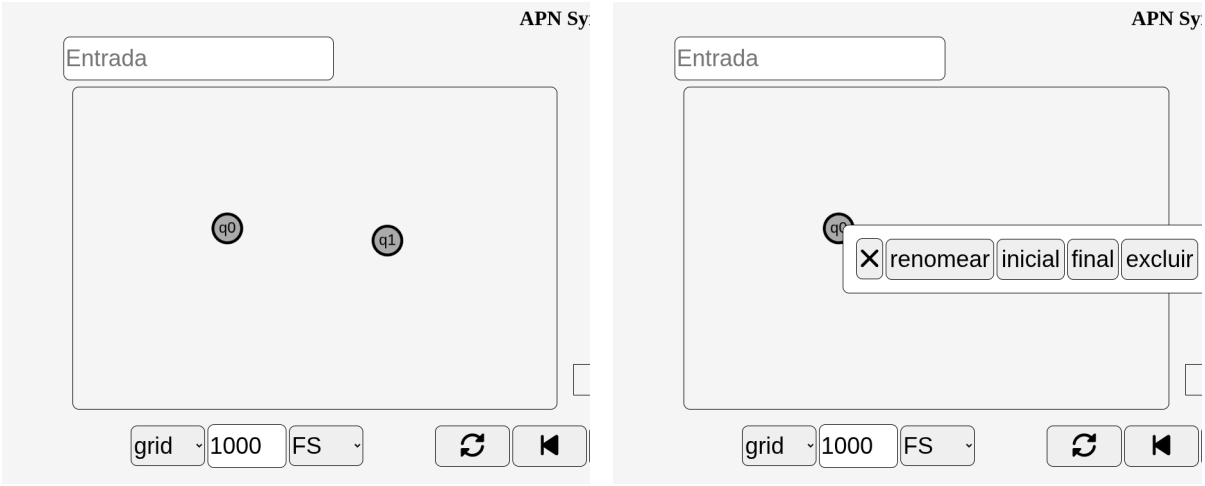
1. Inicialize a palavra w com ϵ.
2. Inicialize o conjunto $WRDS$ com $\emptyset$.
3. Se P pode gerar ϵ, faça $WRDS = WRDS \cup \{w\}$.
4. Compute o *First(P)*.
5. Escolha, aleatoriamente, um símbolo a de *First(P)*.
6. Compute a derivada de G em relação a a.
7. Remova as produções inatingíveis de G.
8. Atualize a lista de produções anuláveis.
9. Retorne ao passo 3.

3.2 Uso do software

A interface do software pode ser dividida em duas partes principais: criação/edição de APNs e execução do APN com base em uma palavra. A parte de criação permite: instanciar estados; editar o nome do estado; defini-lo como final, inicial, ambos ou nenhum; excluí-lo; criar transições; editar valores da transição; excluir transições; baixar o APN desenvolvido; importar um APN desenvolvido anteriormente; mover livremente os elementos; e aplicar *layouts* prontos do Cytoscape para organizar a visualização.

Já a parte de execução do APN permite: definir a palavra de entrada; estipular o número limite de *loops* para a função `step`; escolher a condição de aceitação entre FS (estado final e pilha vazia), S (pilha vazia) ou F (estado final); iniciar e interromper a simulação; avançar ou retroceder um `step`; reiniciar a execução; e visualizar os dados completos de cada *snapshot* individualmente ao clicar nele.

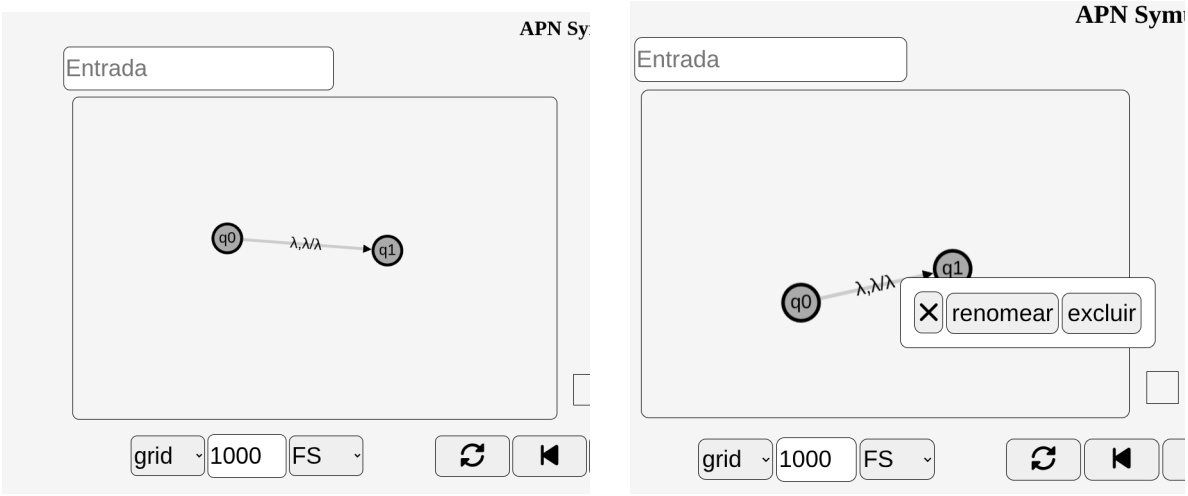
Para exemplificar a utilização, vamos ao cenário de um autômato simples. Para criar cada estado na ferramenta, basta clicar no painel esquerdo do sistema, conforme a Figura 1a. Clicando em um estado com o botão direito, abrem-se as opções de edição: "inicial"e "final"(que alternam o estado daquele nó), além de "renomear"e "excluir". Essas funções são ilustradas na Figura 1b.



(a) (b)

Figura 1 – a Criação do estado, b Editando o estado

Para criar uma transição, clique no estado de origem com o botão esquerdo e, em seguida, no estado de destino. A transição gerada tem, por padrão, o valor λ em todos os elementos, como mostra a Figura 2a. Ao clicar em uma transição com o botão direito, suas opções de edição são exibidas (Figura 2b): "excluir"e "renomear", que permite alterar o caractere de leitura, o caractere a ser desempilhado e o caractere a ser empilhado, caso deixado em branco o campo será considerado como λ. Um exemplo de autômato simples construído na ferramenta pode ser visto na Figura 3.



(a) (b)

Figura 2 – a Criação da transição, b Editando a transição

Ao inserir a palavra de leitura no campo "Entrada"(como a palavra "0101", mostrada na Figura 4) e clicar no botão inferior com ícone de *play*, o sistema calcula a execução completa do autômato. Após clicar no botão com simbolo de *play*, o botão transforma-se no ícone de *stop*, permitindo interromper a demonstração se desejado.

A execução será feita com base em mais dois valores mostrados nessa figura, onde

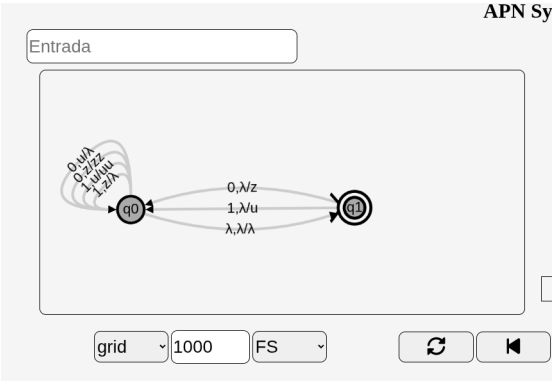


Figura 3 – Um APN simples

está escrito 1000 está definido o número maximo de passos de execução que esse autômato pode executar antes que a sua parada seja forçada, dado que APNs podem entrar em loop infinito. Além disso existe o seletor onde está escrito FS onde é informado o tipo de aceitação do sistema, sendo, FS estado final e pilha vazia, F apenas estado final e S apenas pilha vazia.

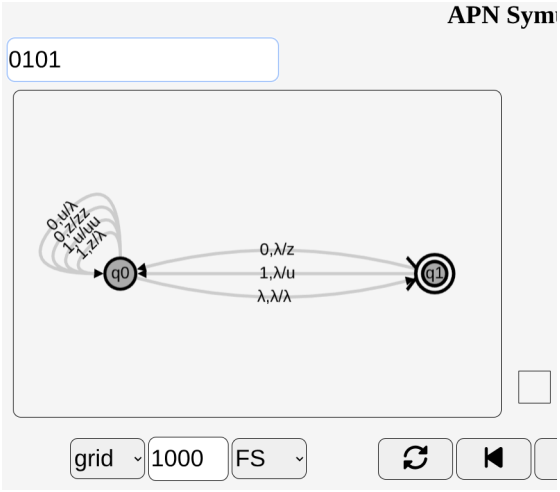
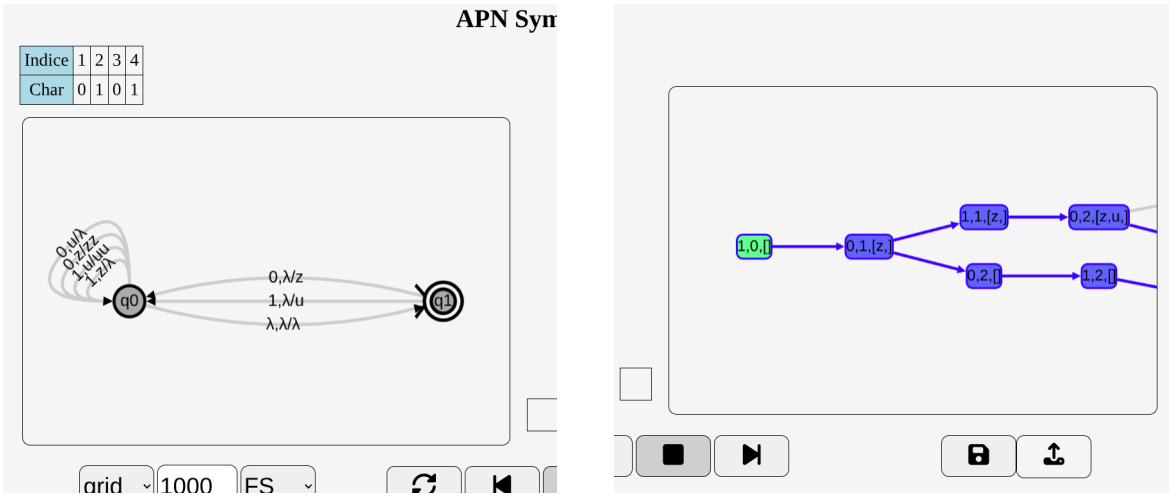


Figura 4 – Entrada da palavra 0101

Durante a execução, o campo da palavra é substituído por uma tabela detalhando cada caractere e seu respectivo índice, destacando o último símbolo lido para facilitar a compreensão (Figura 5a). Além disso, no painel direito é possível visualizar as etapas de execução (os *snapshots*). Aqueles associados ao caractere recém-lido são destacados em verde (Figura 5b). Em cada nó do painel é possível ver: o estado atual na máquina, o índice do caractere lido na palavra e o topo da pilha (até três elementos).

No painel direito, as arestas entre os *snapshots* desenhadas em azul indicam os caminhos que podem levar a um estado de aceitação (Figura 6a), enquanto as cinzas representam ramos sem sucesso. Ao clicar em um *snapshot*, é possível ver sua pilha completa no centro da tela e visualizar o estado em questão destacado no desenho principal (Figura 6b). Os botões de avançar e retroceder permitem iterar passo a passo pelo

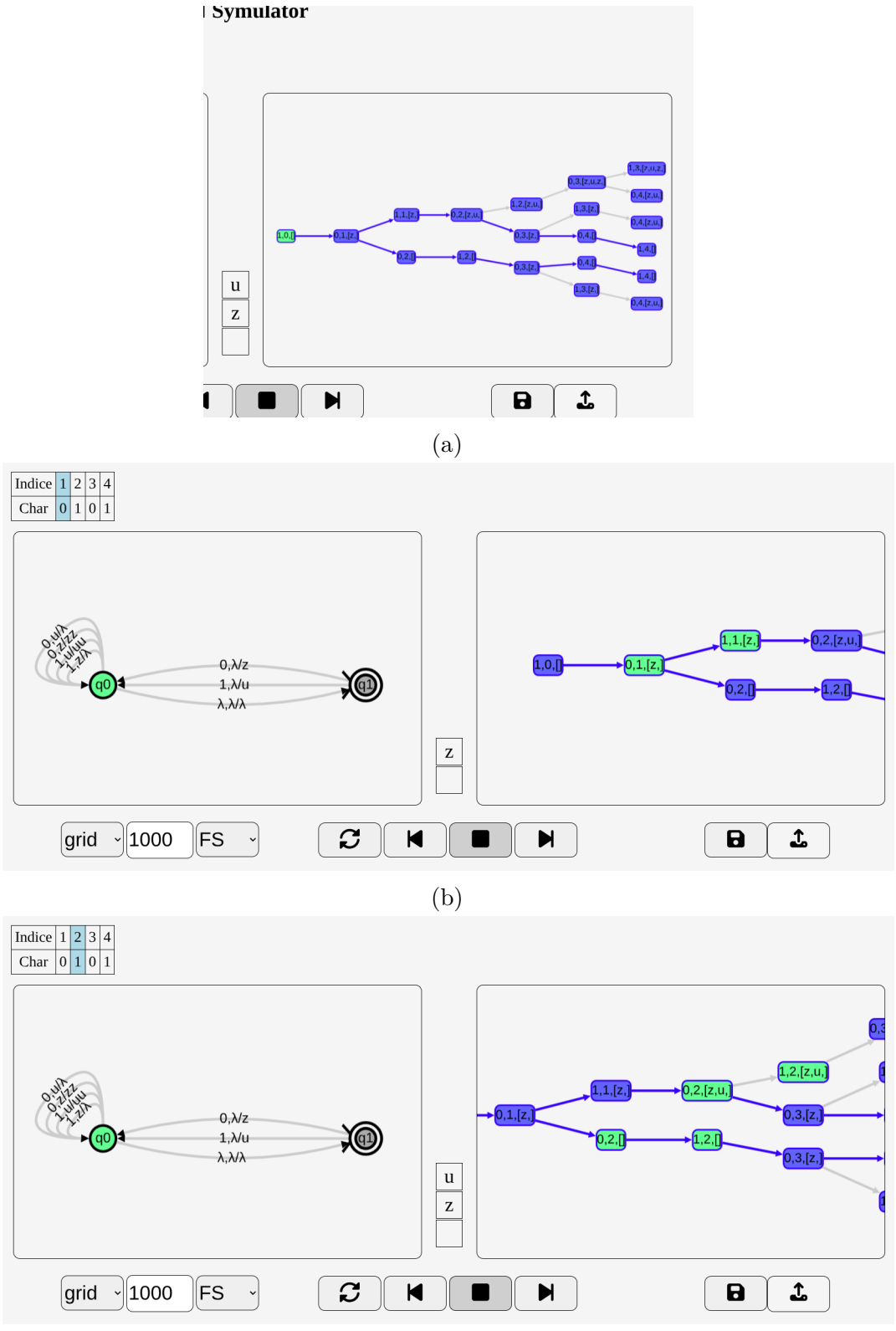


(a) (b)

Figura 5 – a Tabela de caracteres, b Visualização dos *snapshots*

processamento, atualizando os nós em verde (Figura 6c).

No canto inferior esquerdo, encontram-se três botões de configuração (Figura 7a): o primeiro ajusta o *layout* visual; o segundo define o limite numérico de *loops* para interromper execuções infinitas; e o terceiro seleciona o critério de aceitação (FS, S ou F). Por fim, no canto inferior direito, há dois botões (Figura 7b): o ícone de disquete realiza o *download* do autômato criado em formato JSON, e o ícone de seta faz o *upload* de um JSON contendo uma máquina previamente salva.

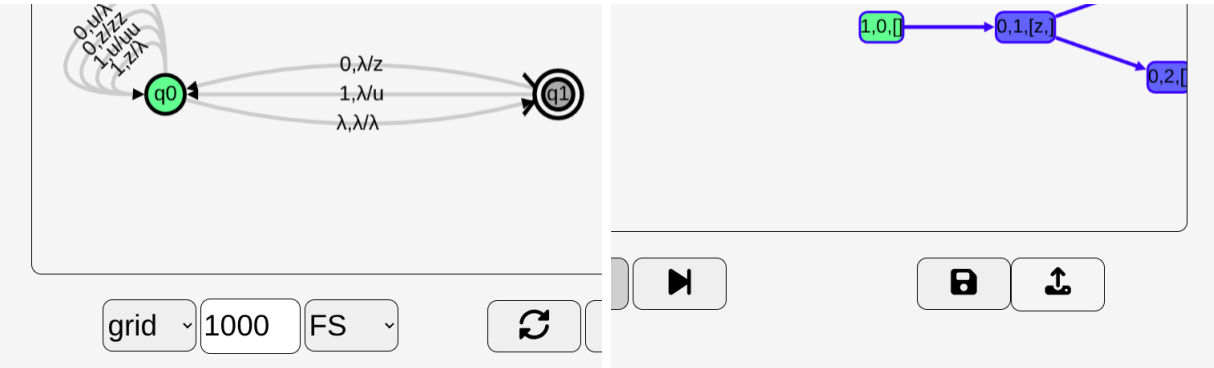


(a)

(b)

(c)

Figura 6 – a Caminho de aceitação, b Clique em *snapshot*, c Botão de avançar



(a) (b)

Figura 7 – a Botões de configuração, b Botões de *download*/*upload*

4 Resultados

Este capítulo descreve os resultados finais do desenvolvimento da ferramenta proposta, abordando as funcionalidades implementadas, as limitações identificadas, bem como possíveis trabalhos futuros e expansões do projeto.

Foi desenvolvido um construtor gráfico de APN baseado em grafos manipuláveis pelo usuário, além de um painel que ilustra de forma dinâmica a execução de uma determinada palavra e os múltiplos caminhos possíveis gerados pelo não-determinismo. Esse painel também destaca visualmente os caminhos que resultaram em aceitação. Ao submeter uma palavra à análise do autômato, o usuário pode selecionar entre três critérios de aceitação: pilha vazia (S), estado final (F) ou ambos (FS). O design da interface foi projetado priorizando a usabilidade e a agilidade nas interações em detrimento de configurações excessivamente detalhadas, focando na redução do número de cliques necessários para executar cada ação.

Os APNs modelados na ferramenta podem ser exportados em formato JSON e importados localmente para dar continuidade a estudos anteriores. Como o sistema foi concebido para fins essencialmente didáticos e de apoio gráfico, o foco recai sobre autômatos de pilha de pequeno porte, podendo apresentar degradação de desempenho conforme o grau de não-determinismo aumenta ou dependendo do comprimento da palavra analisada.

Dessa forma, a aplicação não foi projetada para testes massivos de modelos robustos, embora sua eficiência computacional possa ser refinada em etapas posteriores para cenários mais exigentes. O sistema também não mitiga de forma nativa a ocorrência de *loops* infinitos no autômato; por essa razão, foi integrado um parâmetro configurável de limite máximo de iterações. Essa decisão justifica-se pelo fato de que a detecção e prevenção absoluta de ciclos infinitos em APNs constitui um problema complexo na literatura da área. Ademais, como os navegadores web modernos operam sua interface (*frontend*) prioritariamente em uma única *thread* e o projeto prescinde de um *backend* — decisão estratégica para viabilizar a hospedagem gratuita e estável do sistema —, todo o processamento dos caminhos não-determinísticos é executado de forma síncrona no cliente.

O desenvolvimento do *software* para abranger outras máquinas de estados chegou a ser iniciado, contudo, em razão do cronograma reduzido, optou-se pela delimitação do escopo aos APNs. A integração dessas demais máquinas permanece como proposta para trabalhos futuros. Por fim, ressalta-se que não foi conduzido um estudo empírico com estudantes para avaliar a real eficácia pedagógica da ferramenta no ensino da disciplina, consolidando a validação com usuários como uma das principais vertentes para a continuidade desta pesquisa.

5 Conclusão

Este trabalho teve como objetivo desenvolver uma ferramenta para auxiliar os discentes dos cursos da área de computação no aprendizado de linguagens formais e máquinas de estados. O foco principal recaiu sobre os autômatos de pilha não determinísticos (APNs), visto que esses conhecimentos, embora fundamentais, caracterizam-se por sua alta complexidade e abstração.

Entre as principais contribuições do projeto desenvolvido, destacam-se: a modelagem gráfica de APNs; a visualização, passo a passo, das múltiplas ramificações de execução do autômato; e a funcionalidade de criação e correção de exercícios. Todos esses recursos foram integrados em uma plataforma web, eliminando qualquer necessidade de instalação prévia no computador do usuário.

Apesar dos resultados satisfatórios, o sistema apresenta algumas limitações. A principal delas diz respeito à execução de autômatos que podem gerar um número exponencial de caminhos ao computarem palavras muito extensas. Como a aplicação é executada integralmente no lado do cliente (*frontend*), essa restrição varia significativamente de acordo com a capacidade do *hardware* do usuário. No entanto, essa limitação não representa um obstáculo significativo, uma vez que a ferramenta foi concebida com propósitos didáticos para a simulação de exemplos acadêmicos, e não para o processamento de modelos de grande porte voltados à pesquisa.

Dessa forma, conclui-se que a ferramenta desenvolvida demonstra grande potencial para auxiliar os alunos na compreensão dos autômatos de pilha não determinísticos. Contudo, ressalta-se que ainda não foi realizada uma pesquisa prática com o público-alvo para determinar empiricamente sua eficácia e efetividade. Sendo assim, a validação com os usuários consolida-se como um dos principais tópicos para trabalhos futuros. Adicionalmente, sugere-se a implementação de outras máquinas de estados além dos APNs, bem como o desenvolvimento de um *backend* para o sistema, o que otimizaria o desempenho da aplicação e permitiria a execução de autômatos mais complexos.

Referências

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811. Citado na página 32.

BRZOZOWSKI, J. A. Derivatives of regular expressions. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 11, n. 4, p. 481–494, out. 1964. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/321239.321249>. Citado 2 vezes nas páginas 13 e 22.

BURCH, C. *Autosim*. 2008. <https://cburch.com/proj/autosim/index.html>. Acessado em: 24 jun. 2025. Citado na página 15.

DICKERSON, K. *Automaton Simulator*. 2021. <https://automatonsimulator.com/>. Acessado em: 24 jun. 2025. Citado na página 15.

DOTY, D. *UC Davis Automaton Simulator*. 2020. <https://web.cs.ucdavis.edu/~doty/automata/>. Acessado em: 24 jun. 2025. Citado na página 15.

Free Software Foundation. *grep(1) — Linux User's Manual*. [S.l.], 2026. Disponível offline via comando 'man grep'. Disponível em: <https://man7.org/linux/man-pages/man1/grep.1.html>. Citado na página 18.

HENRIKSEN, I.; BILARDI, G.; PINGALI, K. Derivative grammars: a symbolic approach to parsing with derivatives. *Proc. ACM Program. Lang.*, Association for Computing Machinery, New York, NY, USA, v. 3, n. OOPSLA, out. 2019. Disponível em: <https://doi.org/10.1145/3360553>. Citado na página 24.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3rd. ed. Boston, MA, USA: Addison-Wesley, 2006. ISBN 0-321-46225-4. Citado na página 18.

JUKEMURA, A. S.; NASCIMENTO, H. A. D. do; UCHôA, J. Q. Gam - um simulador para auxiliar o ensino de linguagens formais e de autômatos. *25° Congersso da Sociedade Brasileira de Computação*, 2005. Acessado em: 24 jun. 2025. Disponível em: <https://www.academia.edu/download/69969649/GAM-Um_Simulador_para_Auxiliar_o_Ensino_20210920-880-j6ritm.pdf>. Citado na página 14.

KIENETZ, E. B.; CANAL, A. P. Simulador de autômatos finitos determinísticos - automatograph. *Disc. Scientia. Série: Ciências Naturais e Tecnológicas*, v. 5, n. 1, p. 1–9, 2004. ISSN 1519-0625. Trabalho de Iniciação Científica - PROBIC. Acessado em: 24 jun. 2025. Disponível em: <https://periodicos.ufn.edu.br/index.php/disciplinarumNT/article/download/1176/1113>. Citado na página 14.

MIGHT, M.; DARAIS, D.; SPIEWAK, D. Parsing with derivatives: a functional pearl. In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: Association for Computing Machinery, 2011. (ICFP '11), p. 189–195. ISBN 9781450308656. Disponível em: <https://doi.org/10.1145/2034773.2034801>. Citado na página 24.

PILLAY, N. Learning difficulties experienced by students in a course on formal languages and automata theory. *SIGCSE Bull.*, Association for Computing Machinery, New York, NY, USA, v. 41, n. 4, p. 48–52, jan. 2010. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/1709424.1709444>. Citado na página 12.

RENSSELAER; UNIVERSITY, D. *JFLAP*. 2018. <https://www.jflap.org/>. Acessado em: 24 jun. 2025. Citado na página 15.

VIEIRA, L. F. M.; VIEIRA, M. A. M.; VIEIRA, N. J. Language emulator, uma ferramenta de auxílio no ensino de teoria da computação. *Anais do Congresso*, 2003. Departamento de Ciência da Computação, UFMG. Acessado em: 24 jun. 2025. Disponível em: <http://homepages.dcc.ufmg.br/~mmvieira/publications/Wei-languageEmulator.pdf>. Citado na página 14.

VIEIRA, N. *Introdução aos fundamentos da computação: linguagens e máquinas*. Pioneira Thomson Learning, 2006. ISBN 9788522105083. Disponível em: <https://books.google.com.br/books?id=62ieMQAACAAJ>. Citado 2 vezes nas páginas 18 e 27.

WHITE, T. M.; WAY, T. P. jfast: a java finite automata simulator. *SIGCSE Bull.*, Association for Computing Machinery, New York, NY, USA, v. 38, n. 1, p. 384–388, mar. 2006. ISSN 0097-8418. Disponível em: <https://doi.org/10.1145/1124706.1121460>. Citado na página 15.

ZUZAK, I.; JANKOVIC, V. *FSM Simulator*. 2025. <https://ivanzuzak.info/noam/webapps/fsm_simulator/>. Acessado em: 24 jun. 2025. Citado na página 14.