UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

GUILHERME AUGUSTO ANÍCIO DRUMMOND DO NASCIMENTO
Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro
Coorientadora: Profª. Dra. Valéria de Carvalho Santos

# ELABORAÇÃO DE ALGORITMOS PARA CRIAÇÃO DE EXERCÍCIOS SOBRE AUTÔMATOS FINITOS DETERMINÍSTICOS

Ouro Preto, MG
2024

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

GUILHERME AUGUSTO ANÍCIO DRUMMOND DO NASCIMENTO

# ELABORAÇÃO DE ALGORITMOS PARA CRIAÇÃO DE EXERCÍCIOS SOBRE AUTÔMATOS FINITOS DETERMINÍSTICOS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

**Orientador:** Prof. Dr. Rodrigo Geraldo Ribeiro
**Coorientadora:** Profª. Dra. Valéria de Carvalho Santos

Ouro Preto, MG
2024

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

# FOLHA DE APROVAÇÃO

**Guilherme Augusto Anício Drummond do Nascimento**

**Elaboração de algoritmos para criação de exercícios sobre Autômatos Finitos Determinísticos**

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

Aprovada em 16 de Fevereiro de 2024.

Membros da banca:

Rodrigo Geraldo Ribeiro (Orientador) - Doutor - Universidade Federal de Ouro Preto
Valéria de Carvalho Santos (Coorientadora) - Doutora - Universidade Federal de Ouro Preto
Leonardo Vieira dos Santos Reis (Examinador) - Doutor - Universidade Federal de Juiz de Fora
Reinaldo Silva Fortes (Examinador) - Doutor - Universidade Federal de Ouro Preto

Rodrigo Geraldo Ribeiro, Orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 16/02/2024.

Documento assinado eletronicamente por **Rodrigo Geraldo Ribeiro**, **PROFESSOR 3 GRAU**, em 20/02/2024, às 07:13, 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 **0667181** e o código CRC **C10F907E**.



R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

# Agradecimentos

Aos meus orientadores, Rodrigo e Valéria, pela paciência e dedicação ao me guiarem na construção desse trabalho e por confiarem e incentivarem meu potencial, tanto neste trabalho, quanto em minha carreira acadêmica.

A minha família de sangue e de coração, em especial aos meus pais, meu irmão e minha cunhada, pelo apoio incondicional durante meu período na UFOP. Aos meus pais, Aparecida e Rogério, por me guiar, incentivar e aconselhar todo esse tempo, ao meu irmão Rogério e minha cunhada Clara, pelo apoio incondicional, pelos conselhos e ajudas, mesmo estando distantes, e à Imaculada, por cuidar de mim como se eu fosse um filho.

Aos amigos que fiz na UFOP, por todos os momentos, histórias e memórias. Aos meus amigos do ensino fundamental e médio, por sempre sempre lembrar de mim mesmo quando não nos vemos com tanta frequência como antes. E ao Henrique, por todas as conversas noturnas, por me escutar e ajudar durante o tempo que passamos juntos.

# Resumo

Teoria da Computação é um importante tópico para a Ciência da Computação e seu ensino, porém, é muito abstrato e matemático. A criação manual de exercícios é um processo complexo, que exige treinamento, experiência e recursos. Isso atrapalha e atrasa o uso de atividades educacionais (e.g. apresentar questões práticas) e novos avanços (como testes adaptativos) que requerem um grande conjunto de perguntas. Junto de sua função como ferramenta de avaliação, questões tem potencial para influenciar no aprendizado do estudante. Alguns dos benefícios são: 1) oferecer a oportunidade de praticar a recuperação de informações da memória; 2) prover aos estudantes um retorno sobre seus equívocos; 3) focar a atenção do estudante no material apresentado; 4) reforçar o aprendizado ao repetir conceitos básicos; e 5) motivar os estudantes a se envolverem nas atividades de aprendizado (e.g. leituras e discussões). Neste trabalho, além de apresentar uma ferramenta para gerar exercícios de construção de autômatos finitos determinísticos e minimização de autômatos finitos determinísticos usando o conceito de expressões regulares, foi também implementado um algoritmo genético para otimizar a geração automática de questões com níveis variados de dificuldade. Além disso, este projeto foi integrado a um outro orientado pelo mesmo professor e ao Jupyter Notebook, para criação de um ambiente interativo e prático para o estudo e prática de exercícios.

**Palavras-chave**: Autômatos Finitos Determinísticos. Expressões regulares. Derivadas de expressões regulares. Geração de exercícios. Teoria da Computação.

# Abstract

Computer Theory is an important topic for Computer Science and its teaching; however, it is highly abstract and mathematical. Manual exercise creation is a complex process requiring training, experience, and resources. This hinders and delays the use of educational activities (e.g., presenting practical questions) and new advancements (such as adaptive tests) that require a large set of questions. Alongside its function as an evaluation tool, questions have the potential to influence student learning. Some of the benefits include: 1) providing an opportunity to practice information retrieval from memory; 2) giving students feedback on their mistakes; 3) focusing students' attention on the presented material; 4) reinforcing learning by repeating basic concepts; and 5) motivating students to engage in learning activities (e.g., readings and discussions). In this work, in addition to presenting a tool for generating exercises on the construction and minimization of deterministic finite automata using regular expressions, a genetic algorithm was also implemented to optimize the automatic generation of questions with varying levels of difficulty. Furthermore, this project was integrated with another guided by the same professor and with Jupyter Notebook, to create an interactive and practical environment for studying and practicing exercises.

**Keywords**: Deterministic Finite Automata, Regular Expressions, Derivatives of Regular Expressions, Exercise Generation, Computational Theory.

# Lista de Ilustrações

# Lista de Tabelas

# List of Algoritmos

# Lista de Abreviaturas e Siglas

| | |
|---|---|
| DECOM | Departamento de Computação |
| UFOP | Universidade Federal de Ouro Preto |
| AFD | autômato finito determinístico |
| ER | expressão regular |
| AFN | autômato finito não determinístico |
| GAQ | geração automática de questões |
| AG | algoritmo genético |

# Lista de Símbolos

| | |
|---|---|
| $\lambda$ | Letra grega minúscula lambda, representa a palavra vazia |
| $\Sigma$ | Letra grega Sigma, representa um alfabeto |
| $\delta$ | Letra grega minúscula delta, representa a função de transição |
| $\nu$ | Letra grega minúscula nu, representa a função de anulabilidade |
| $\partial$ | Símbolo da derivada |
| $\nabla$ | Nabla, símbolo da derivada parcial |
| $\top$ | Tautologia |
| $\bot$ | Contradição ou absurdo |

# Sumário

# 1 Introdução

Teoria da Computação é um importante tópico para o entendimento da Ciência da Computação. Usualmente, o ensino deste assunto envolve conceitos matemáticos como demonstrações e algoritmos (Mohammed, 2020). Tradicionalmente, o ensino de Teoria da Computação é feito sem o auxílio de softwares e os estudantes trabalham nos exercícios propostos usando papel e lápis.

O uso deste modelo tradicional de exercícios possui alguns problemas. Primeiro, para uma melhor compreensão do conteúdo, alunos necessitam fazer uma grande quantidade de exercícios e ter retorno sobre a correção de suas soluções, o que pode sobrecarregar o docente que não deve apenas atestar se uma resposta está incorreta mas também justificar o motivo do erro com o intuito de orientar o aluno na produção de uma solução adequada. Outro aspecto a ser considerado é a dificuldade na criação de exercícios com diferentes níveis de dificuldade.

Dessa forma, a geração automática de questões (GAQ) surgiu como uma solução para os desafios presentes na produção de exercícios avaliativos. As técnicas de GAQ estão interessadas na construção de algoritmos para produzir questões de boa qualidade a partir de fontes de conhecimento. De acordo com Alsubait, Parsia e Sattler (2012), pesquisas sobre GAQ remontam à década de 70 e atualmente vem ganhando importância com o surgimento de cursos onlines abertos e outras tecnologias de aprendizado digital (Qayyum; Zawacki-Richter, 2018; Gaebel *et al.*, 2014; Goldbach; Lup, 2017).

## 1.1 Justificativa

A criação manual de exercícios que envolvem autômatos finitos determinísticos, abrangendo tanto sua construção quanto sua minimização, é um processo desafiador e intensivo em recursos. Teoria dos Autômatos, tópico tratado em Teoria da Computação, com seu caráter abstrato e matemático, frequentemente apresenta obstáculos significativos para o ensino e a aprendizagem. Um sistema de geração de exercícios dedicado a essa área, além de simplificar a elaboração de problemas práticos e teóricos, ofereceria a oportunidade de explorar uma ampla variedade de cenários de aprendizado.

Ao apresentar uma variedade de desafios, desde os fundamentos da construção de autômatos até a complexidade da minimização, a ferramenta pode incentivar os estudantes a praticar a aplicação dos conceitos teóricos, promovendo uma abordagem mais ativa e interativa. Além disso, com a integração com outro trabalho orientado pelo mesmo professor (Costa, 2023), o sistema também é capaz de oferecer *feedback* imediato, permitindo aos estudantes aprimorarem suas habilidades e corrigirem equívocos de maneira eficiente.

## 1.2 Objetivos

O principal objetivo deste trabalho é projetar e desenvolver uma ferramenta para criação automática de exercícios sobre construção e minimização de autômatos finitos determinísticos. Essa ferramenta deverá produzir uma série de exercícios e suas respectivas soluções.

### 1.2.1 Objetivos específicos

São objetivos específicos do trabalho:

1. A criação de um sistema que gere expressões regulares aleatórias que serão utilizadas para construção de enunciados para questões.
2. Implementação de algoritmos para geração de autômatos determinísticos a partir de expressões regulares. Os autômatos produzidos serão utilizados como gabaritos para questões produzidas e para construção de questões de minimização de autômatos.
3. Integração da ferramenta desenvolvida com o Jupyter Notebook.

## 1.3 Organização do Trabalho

O capítulo 2 apresenta uma revisão bibliográfica sobre geração automática de questões e ferramentas de simulação de autômatos, além da fundamentação teórica necessária para compreensão do trabalho. No capítulo 3, é apresentado o desenvolvimento do código e problemas encontrados ao longo do caminho. Em seguida, no capítulo 4 são apresentados os resultados e funcionalidades desenvolvidas e, por fim, o capítulo 5 apresenta a conclusão e os trabalhos futuros relacionados.

O código desenvolvido está disponível no repositório *automata-language* <https://github.com/lives-group/automata-language>.

# 2 Revisão Bibliográfica

Neste capítulo são abordados os conceitos necessários para a compreensão do trabalho desenvolvido. A seção 2.1 revisa conceitos da teoria de autômatos finitos, expressões regulares e suas derivadas, que são utilizadas no processo de produzir um autômato equivalente a uma dada expressão regular. Por sua vez, a seção 2.2 apresenta uma introdução à linguagem Racket utilizada no desenvolvimento deste trabalho. Trabalhos relacionados são discutidos na seção 2.3.

## 2.1 Fundamentação Teórica

No estudo de linguagens formais, os conceitos de alfabeto, palavra e linguagem são fundamentais e sua definição é apresentada a seguir.

**Definição 2.1** (Alfabeto)**.** *Um alfabeto $\Sigma$ é um conjunto finito não vazio de símbolos.*

**Definição 2.2** (Palavra)**.** *Uma palavra é uma sequência finita de símbolos de um determinado alfabeto. A notação $|w|$ denota o número de símbolos que forma a palavra $w$ e $\lambda$ representa a palavra vazia, isto é, $|\lambda| = 0$.*

**Definição 2.3** (Linguagem)**.** *Uma linguagem é um conjunto de palavras sobre um alfabeto. Definimos o conjunto de todas as palavras sobre um alfabeto $\Sigma$ como $\Sigma^*$.*

**Exemplo 2.1.** *Exemplos de possíveis alfabetos são $\Sigma_1 = \{0, 1\}$ ou $\Sigma_2 = \{a, e, i, o, u\}$. Note que conjuntos como $\emptyset$ ou $\mathbb{N}$ não podem ser considerados como alfabetos.*

*Considere o alfabeto $\Sigma_1 = \{0, 1\}$. Exemplos de palavras sobre esse alfabeto são $w_1 = 110$, $w_2 = 00$ e $w_3 = \lambda$. Ainda considerando o alfabeto $\Sigma_1$, a seguir será apresentado alguns exemplos de linguagens: $\{110, 00, \lambda, 1\}$, $\emptyset$ e $\{0^n \mid n \geq 10\}$.*

Intuitivamente, um autômato pode ser entendido como uma máquina abstrata para determinar se uma certa palavra pertence ou não a uma linguagem. A seguir, será apresentada a definição formal e exemplos de autômatos finitos.

### 2.1.1 Autômatos Finitos Determinísticos

Um Autômato Finito Determinístico (AFD) pode ser definido formalmente como uma quíntupla $M = (E, \Sigma, \delta, i, F)$, conforme a definição a seguir:

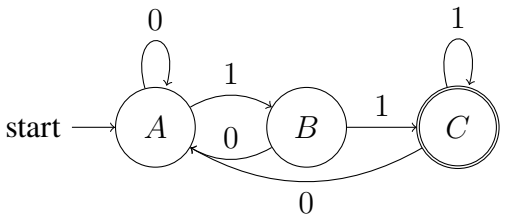
**Definição 2.4** (Autômato Finito Determinístico)**.** *Um AFD é uma quíntupla $(E, \Sigma, \delta, i, F)$, em que:*

- *$E$: é o conjunto de estados.*

- $\Sigma$*: é o alfabeto.*
- $\delta : E \times \Sigma \to E$*: função de transição, uma função total.*
- $i \in E$*: é o estado inicial.*
- $F \subseteq E$*: conjunto de estados finais.*

**Exemplo 2.2.**

*Considere a seguinte linguagem sobre* $\Sigma = \{0, 1\}$*:* $L = \{0, 1\}^*\{11\}$*. Ou seja, a linguagem de palavras sobre* $\{0, 1\}$ *que terminam em* $11$*. Um AFD que reconhece palavras desta linguagem é apresentado a seguir.*




*Usualmente, AFDs são apresentados utilizando grafos direcionados em que nós denotam estados do autômato e arestas rotuladas sua função de transição. Formalmente, o AFD anterior é descrito pela seguinte quíntupla:* $(\{A, B, C\}, \{0, 1\}, \delta, A, \{C\})$*, em que a função de transição* $\delta$ *é definida pelas seguinte tabela:*

| $\delta$ | $0$ | $1$ |
|---|---|---|
| $A$ | $A$ | $B$ |
| $B$ | $A$ | $C$ |
| $C$ | $A$ | $C$ |

Para definir a linguagem aceita por uma AFD, é necessário estender a função de transição, que opera sobre símbolos de um alfabeto, para palavras. A definição a seguir apresenta este conceito.

**Definição 2.5** (Função de transição estendida)**.** *Seja* $M = (E, \Sigma, \delta, i, F)$ *um AFD qualquer. A função de transição estendida para* $M$*,* $\hat{\delta} : E \times \Sigma^* \to E$*, retorna o estado* $e_2 \in E$ *no qual a computação de uma palavra* $w$ *termina em* $M$ *ao ser inciado o processamento de* $w$ *em um estado* $e$*. Formalmente,* $\hat{\delta}$ *é definido por recursão sobre a estrutura da palavra* $w$*.*

$$\hat{\delta}(e, \lambda) = e \quad (1)$$
$$\hat{\delta}(e, ay) = \hat{\delta}(\delta(e, a), y),\ a \in \Sigma, y \in \Sigma^* \quad (2)$$

**Exemplo 2.3.** *Considere o AFD apresentado no exemplo 2.2. Utilizando a função de transição estendida podemos obter o estado em que a computação deste AF termina com a palavra* $011$ *ao*

*iniciar seu processamento no estado inicial, como se segue:*

$$\begin{array}{lcl}
\hat{\delta}(A, 011) & = & \textit{equação (2) de } \hat{\delta} \\
\hat{\delta}(\delta(A, 0), 11) & = & \textit{def. de } \delta \\
\hat{\delta}(A, 11) & = & \textit{equação (2) de } \hat{\delta} \\
\hat{\delta}(\delta(A, 1), 1) & = & \textit{def. de } \delta \\
\hat{\delta}(B, 1) & = & \textit{equação (2) de } \hat{\delta} \\
\hat{\delta}(\delta(B, 1), \lambda) & = & \textit{def. de } \delta \\
\hat{\delta}(C, \lambda) & = & \textit{equação (1) de } \hat{\delta} \\
C & &
\end{array}$$

Utilizando a função de transição estendida, podemos definir a linguagem aceita por um AFD.

**Definição 2.6** (Linguagem aceita por um AFD)**.** *Seja $M = (E, \Sigma, \delta, i, F)$ um AFD qualquer. A linguagem aceita por $M$, $L(M)$, é definida como:*

$$L(M) = \{w \in \Sigma^* \mid \hat{\delta}(i, w) \in F\}$$

### 2.1.2 Expressões Regulares

Dizemos que uma linguagem $L \subseteq \Sigma^*$ é regular se existe um autômato finito determinístico $M$ tal que $L(M) = L$. Dessa forma, temos que autômatos são uma maneira de especificar uma linguagem regular. Outro formalismo amplamente utilizado para definir linguagens regulares são as chamadas expressões regulares, que permitem uma especificação de linguagens utilizando uma notação algébrica. A seguir, apresentamos a definição de expressões regulares.

**Definição 2.7** (Expressões regulares)**.** *Seja $\Sigma$ um alfabeto qualquer. O conjunto de expressões regulares sobre $\Sigma$ é definido indutivamente da seguinte forma:*

- *O conjunto vazio, representado por $\emptyset$.*
- *O conjunto contendo apenas a palavra vazia, que não possui nenhum símbolo do alfabeto, representado por $\lambda$.*
- *O conjunto que contém apenas o símbolo $a$, $a \in \Sigma$, representado pelo próprio caractere $a$.*
- *Sejam $r$ e $s$ duas expressões regulares quaisquer sobre $\Sigma$:*
  - *$r \cdot s$, representa o conjunto das palavras que podem ser obtidas concatenando palavras pertencentes a $r$ com palavras pertencentes a $s$. Comumente, o símbolo da concatenação é omitido, ou seja, $r \cdot s$ também pode ser escrito como $rs$.*
  - *$r + s$, representa o conjunto obtido pela união de $r$ e $s$.*
  - *$r^*$, representa o conjunto obtido pela concatenação de $r$ zero ou mais vezes, chamado de Fecho de Kleene. $r^*$ possui a palavra vazia, mesmo que $r$ não possua.*
  - *$r$ & $s$, representa o conjunto das palavras que pertencem a $r$ e a $s$ simultaneamente.*
  - *$\neg\, r$, representa o conjunto de palavras que não pertencem a $r$*

*Representamos a linguagem denotada por uma expressão regular* $e$ *por* $L(e)$.

#### 2.1.2.1 Derivadas de Expressões Regulares

A equivalência entre expressões regulares e autômatos finitos é um resultado clássico da teoria da computação. Ao invés de utilizar a equivalência normalmente apresentada em livros clássicos (Sipser, 2013), vamos usar a baseada no conceito de derivadas. A derivada de um conjunto de strings $S$ com respeito a um símbolo $a$, $\partial_a(S)$, é o conjunto de strings geradas ao retirar o $a$ inicial das strings de $S$ que começam com $a$. Formalmente:

$$\partial_a(S) = \{y \in \Sigma^* \mid ay \in S\}$$

Se $S$ for um conjunto regular, i.e., um conjunto definido por uma expressão regular (ER), sua derivada também é um conjunto regular. Usando as derivadas de expressões regulares, é possível simular a execução de um autômato sobre uma palavra de entrada $w$ e também construir um autômato finito determinístico equivalente a ER, usando um método proposto por Brzozowski (1964). Para definirmos a operação de derivada sobre uma expressão regular, primeiramente vamos apresentar o conceito auxiliar de anulabilidade:

**Definição 2.8** (Anulabilidade)**.** *Dizemos que uma expressão regular* $e$ *é anulável se* $\lambda \in L(e)$. *A função* $nullable : e \to \{\bot, \top\}$ *determina se uma expressão regular é ou não anulável e é definida como se segue:*

$$\begin{aligned}
nullable(\lambda) &= \top \\
nullable(a) &= \bot \\
nullable(\emptyset) &= \bot \\
nullable(r \cdot s) &= nullable(r) \wedge nullable(s) \\
nullable(r + s) &= nullable(r) \vee nullable(s) \\
nullable(r^*) &= \top \\
nullable(r \& s) &= nullable(r) \wedge nullable(s) \\
nullable(\neg r) &= \neg nullable(r)
\end{aligned}$$

*Usando* $nullable$*, definimos:*

$$\nu(e) = \begin{cases} \lambda & \textit{se } nullable(e) \\ \emptyset & \textit{caso contrário} \end{cases}$$

A seguir, apresentamos um exemplo que ilustra o uso desta função sobre uma expressão regular.

**Exemplo 2.4.**

*Considere a expressão regular* $L = (0+1)^*11$.

$$\begin{aligned}
&nullable((0+1)^*11) &=\\
&nullable((0+1)^*) \wedge nullable(11) &=\\
&nullable((0+1)^*) \wedge (nullable(1) \wedge nullable(1)) &=\\
&\top \wedge (\bot \wedge \bot) &=\\
&\top \wedge \bot &=\\
&\bot
\end{aligned}$$

*Portanto,* $L = (0+1)^*11$ *não é anulável.*

Utilizando a função $nullable$ definimos uma função que obtém a derivada de uma expressão regular.

**Definição 2.9** (Derivada de ER com respeito a um símbolo $a$)**.** *A derivada de uma expressão regular com respeito a um símbolo* $a$*,* $\partial_a(e)$*, é definida pela seguinte função:*

$$\begin{aligned}
\partial_a(\lambda) &= \emptyset\\
\partial_a(b) &= \begin{cases} \lambda & \textit{se } a = b\\ \emptyset & \textit{caso contrário} \end{cases}\\
\partial_a(\emptyset) &= \emptyset\\
\partial_a(r \cdot s) &= \partial_a(r) \cdot s + \nu(r) \cdot \partial_a(s)\\
\partial_a(r^*) &= \partial_a(r) \cdot r^*\\
\partial_a(r + s) &= \partial_a(r) + \partial_a(s)\\
\partial_a(r \& s) &= \partial_a(r) \& \partial_a(s)\\
\partial_a(\neg r) &= \neg(\partial_a(r))
\end{aligned}$$

A seguir, apresentamos um exemplo de cálculo da derivada de uma expressão regular.

**Exemplo 2.5.**

*Considere a expressão regular* $r = (0+1)^*11$.

*Calculando a derivada dessa expressão regular com respeito ao símbolo 1.*

$$
\begin{array}{ll}
\partial_1((0+1)^*11) & = \\
(\partial_1((0+1)^*)\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
((\partial_1(0+1)\cdot(0+1)^*)\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
(((\partial_1(0)+\partial_1(1))\cdot(0+1)^*)\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
(((\emptyset+\partial_1(1))\cdot(0+1)^*)\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
(((\emptyset+\lambda)\cdot(0+1)^*)\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
((\lambda\cdot(0+1)^*)\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
((0+1)^*\cdot 11) + (\nu((0+1)^*)\cdot\partial_1(11)) & = \\
((0+1)^*\cdot 11) + (\lambda\cdot\partial_1(11)) & = \\
((0+1)^*\cdot 11) + (\lambda\cdot 1) & = \\
(0+1)^*\cdot 11 + 1 &
\end{array}
$$

*Logo* $\partial_1((0+1)^*11) = (0+1)^*11+1$. *Observe que o resultado ao se processar um símbolo* $1$ *de palavras pertencentes a* $(0+1)^*11$ *são todas palavras que pertencem a este conjunto e a palavra* $1$*, resultado de processarmos o primeiro* $1$ *da menor palavra pertencente a esta linguagem,* $11$.

Derivadas fornecem uma maneira elegante para verificarmos se uma string pertence ou não à linguagem de uma expressão regular. Para isso, basta aplicar a operação de derivada recursivamente sobre cada símbolo pertencente à string. Caso a expressão regular resultante seja anulável, temos que a string original pertence à linguagem descrita pela expressão regular. A definição a seguir formaliza essa ideia.

**Definição 2.10** (Função de derivada estendida)**.** *É possível estender a função de derivada para uma string da seguinte forma:*

$$
\begin{aligned}
\partial^*_\lambda(r) &= r \\
\partial^*_{ab}(r) &= \partial^*_b(\partial_a(r))
\end{aligned}
$$

*Também é possível usar a função estendida para determinar se uma palavra pertence a linguagem definida pela expressão regular, simulando a execução de um AFD:*

$$
\begin{aligned}
\hat{\partial}_\lambda(r) &= \nu(r) \\
\hat{\partial}_{ab}(r) &= \hat{\partial}_b(\partial_a(r))
\end{aligned}
$$

**Exemplo 2.6.**

*Considere a expressão regular* $r = (0+1)^*11$ *e a palavra 1011.*

*Calculando a derivada dessa linguagem com respeito a essa palavra.*

$$\begin{aligned}
&\partial^*_{1011}((0+1)^*11) &&= \\
&\partial^*_{011}(\partial^*_1((0+1)^*11)) &&= \\
&\partial^*_{011}((0+1)^*11+1) &&= \\
&\partial^*_{11}(\partial^*_0((0+1)^*11+1) &&= \\
&\partial^*_{11}((0+1)^*11) &&= \\
&\partial^*_1(\partial^*_1((0+1)^*11) &&= \\
&\partial^*_1((0+1)^*11+1) &&= \\
&\partial^*_\lambda(\partial^*_1((0+1)^*11+1) &&= \\
&\partial^*_\lambda((0+1)^*11+1+\lambda) &&= \\
&(0+1)^*11+1+\lambda
\end{aligned}$$

*Dessa forma* $\partial^*_{1011}((0+1)^*11) = (0+1)^*11+1+\lambda$*. E como a expressão regular* $(0+1)^*11+1+\lambda$ *é anulável, i.e.,* $\nu((0+1)^*11+1+\lambda) = \lambda$*, a palavra 1011 pertence a linguagem definida pela expressão regular.*

Algoritmos clássicos para obtenção de autômatos equivalentes a expressões regulares possuem o inconveniente de produzir autômatos não determinísticos. Derivadas permitem a obtenção de um autômato determinístico diretamente a partir de uma expressão regular. A próxima seção apresenta esta técnica.

#### 2.1.2.2 Construção do Autômato Finito Determinístico equivalente à Expressão Regular

Formalmente o autômato finito determinístico equivalente a uma expressão regular $e$ é definido como se segue:

**Definição 2.11.** *Seja* $e$ *uma expressão regular qualquer sobre* $\Sigma$*. O autômato finito determinístico equivalente a* $e$*,* $M = (D(e), \Sigma, \delta, i, F)$*, em que:*

- $D(e)$*: conjunto finito de todas as derivadas da expressão regular* $e$*.*
- $\delta(e, a) = \partial_a(e)$
- $i = e$
- $F = \{e' \in D(e) \,|\, \nu(e') = \lambda\}$

Intuitivamente, a definição anterior mostra que o conjunto de estados do autômato equivalente é o conjunto de todas as derivadas de uma expressão regular. Transições do autômato são representadas pela aplicação da operação de derivada sobre um símbolo do alfabeto. O conjunto de estados finais são todos os estados correspondentes a expressões regulares anuláveis e o estado inicial é o que representa a expressão regular original. A seguir apresentamos um algoritmo para construção de um AFD usando derivadas.

**Algoritmo 1** Construção de AFD com derivadas de ER (adaptado de (Owens; Reppy; Turon, 2009))

**função** VISITA($e, c, (E, \delta)$)
    seja $e_c$ = $\partial_c(e)$
    em
    **se** $\exists e' \in E$ tal que $e' \equiv e_c$ **então**
        $(E, \delta \cup \{(e, c) \mapsto e_c\})$
    **senão**
        seja $E' = E \cup \{e_c\}$
        seja $\delta' = \delta \cup \{(e, c) \mapsto e_c\}$
        em *explore*$(E', \delta', e_c)$
    **fim se**
**fim função**

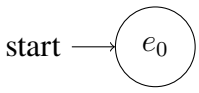
**função** EXPLORE($E, \delta, e$)
    fold (*visita* $e$) $(E, \delta)$ $\Sigma$
**fim função**

**função** CONSTROI-AFD(r)
    seja $i = r$
    seja $(E, \delta)$ = explore $(\{i\}, \{\}, i)$
    seja $F$ = $\{e \mid e \in E \wedge \nu(e) = \lambda\}$
    em $\langle E, i, F, \delta \rangle$
**fim função**

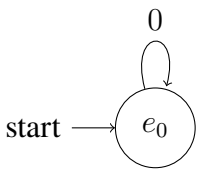
A seguir apresentamos um exemplo de uso deste algoritmo para construção do autômato equivalente a uma expressão regular usando derivadas.

**Exemplo 2.7.**

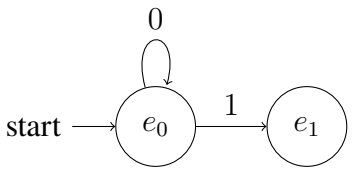
*Considere a expressão regular* $r = (0 + 1)^*11$. *A construção do AFD começa com o estado inicial* $e_0 = r = (0 + 1)^*11$ *e segue da seguinte forma:*



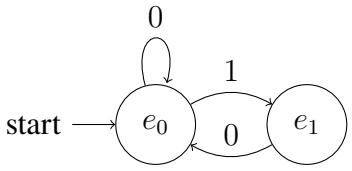

*Calcule* $\partial_0(e_0) = \partial_0((0 + 1)^*11) = (0 + 1)^*11 = e_0$



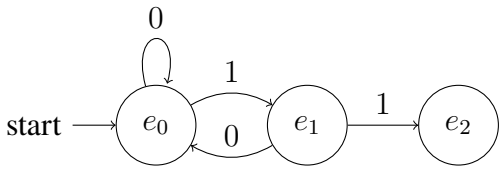

*Calcule* $\partial_1(e_0) = \partial_1((0 + 1)^*11) = (0 + 1)^*11 + 1$, *que é novo, então chamamos de* $e_1$



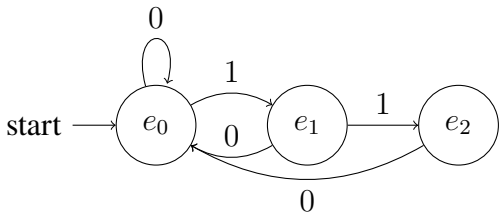

*Calcule* $\partial_0(e_1) = \partial_0((0+1)^*11+1) = (0+1)^*11 = e_0$



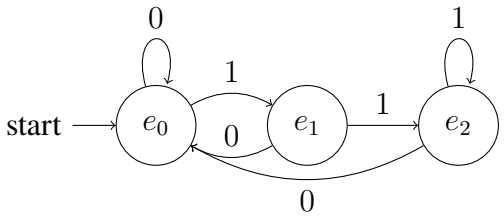

*Calcule* $\partial_1(e_1) = \partial_1((0+1)^*11+1) = (0+1)^*11+1+\lambda$*, que é novo, então chamamos de* $e_2$




*Calcule* $\partial_0(e_2) = \partial_0((0+1)^*11+1+\lambda) = (0+1)^*11 = e_0$



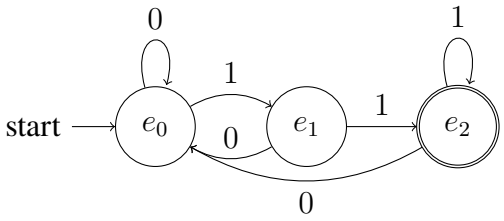

*Calcule* $\partial_1(e_2) = \partial_1((0+1)^*11+1+\lambda) = (0+1)^*11+1+\lambda = e_2$




*Por fim, olhamos quais estados são anuláveis:*

$nullable(e_0) = nullable((0+1)^*11) = \bot$
$nullable(e_1) = nullable((0+1)^*11+1) = \bot$
$nullable(e_2) = nullable((0+1)^*11+1+\lambda) = \top$

*Como $e_2$ é anulável, então ele também é um estado final.*




#### 2.1.2.3 Derivadas Parciais de Expressões Regulares

Antimirov (1996) introduziu a noção de derivadas parciais, uma generalização das derivadas de Brzozowski, e um método para construção do Autômato Finito Não-Determinístico (AFN) equivalente à ER. A principal diferença é que a derivada parcial produz um conjunto de ERs, enquanto a derivada produz uma única ER.

**Definição 2.12** (Derivada parcial de ER com respeito a um símbolo $a$)**.** *A derivada parcial de uma expressão regular com respeito a um símbolo $a$, $\partial_a(e)$, é definida pela seguinte função:*

$$
\begin{aligned}
\nabla_a(\lambda) &= \emptyset \\
\nabla_a(b) &= \begin{cases} \{\lambda\} & \textit{se } a = b \\ \emptyset & \textit{caso contrário} \end{cases} \\
\nabla_a(\emptyset) &= \emptyset \\
\nabla_a(r \cdot s) &= \begin{cases} \nabla_a(r) \odot s & \textit{se } nullable(r) = \bot \\ \nabla_a(r) \odot s \cup \nabla_a(s) & \textit{caso contrário} \end{cases} \\
\nabla_a(r^*) &= \nabla_a(r) \odot r^* \\
\nabla_a(r + s) &= \nabla_a(r) \cup \nabla_a(s) \\
\nabla_a(r \& s) &= \nabla_a(r) \dot{\cap} \nabla_a(s) \\
\nabla_a(\neg r) &= \dot{\neg}(\nabla_a(r))
\end{aligned}
$$

*Onde $\odot$ é definida da seguinte forma:*

$$R \odot s = \{r \cdot s \,|\, r \in R\}$$

*$\dot{\cap}$ é definida da seguinte forma:*

$$R \dot{\cap} S = \{r \& s \,|\, r \in R, s \in S)\}$$

*E $\dot{\neg}$ é definida recursivamente da seguinte forma:*

$$
\begin{aligned}
\dot{\neg}\emptyset &= \Sigma^* \\
\dot{\neg}\{r\} &= \{\neg r\} \\
\dot{\neg} R &= \{\neg r_1 \cap \cdots \cap \neg r_n\}
\end{aligned}
$$

**Exemplo 2.8.**

*Considere a expressão regular* $r = (0+1)^*11$.

*Calculando a derivada dessa expressão regular com respeito ao símbolo 1.*

$$\begin{array}{ll}
\nabla_1((0+1)^*11) & = \\
(\nabla_1((0+1)^*) \odot 11) \cup (\nabla_1(11)) & = \\
((\nabla_1((0+1)) \odot (0+1)^*) \odot 11) \cup (\nabla_1(11)) & = \\
(((\nabla_1(0) \cup \nabla_1(1)) \odot (0+1)^*) \odot 11) \cup (\nabla_1(11)) & = \\
(((\emptyset \cup \nabla_1(1)) \odot (0+1)^*) \odot 11) \cup (\nabla_1(11)) & = \\
(((\emptyset \cup \{\lambda\}) \odot (0+1)^*) \odot 11) \cup (\nabla_1(11)) & = \\
((\{\lambda\} \odot (0+1)^*) \odot 11) \cup (\nabla_1(11)) & = \\
(\{(0+1)^*\} \odot 11) \cup (\nabla_1(11)) & = \\
\{(0+1)^*11\} \cup (\nabla_1(11)) & = \\
\{(0+1)^*11\} \cup \{1\} & = \\
\{(0+1)^*11, 1\} & =
\end{array}$$

*Logo* $\nabla_1((0+1)^*11) = \{(0+1)^*11, 1\}$. *Observe que o resultado ao se processar um símbolo* $1$ *de palavras pertencentes a* $(0+1)^*11$ *são todas palavras que pertencem a este conjunto e a palavra* $1$*, resultado de processarmos o primeiro* $1$ *da menor palavra pertencente a esta linguagem,* $11$.

Assim como as derivadas fornecem uma maneira de verificar se uma string pertence ou não à linguagem de uma ER, as derivadas parciais também fornecem. Porém, como a derivada parcial retorna um conjunto, as chamadas recursivas da derivada parcial devem ser feitas sobre todas as ERs do conjunto e a string pertence a linguagem caso pelo menos uma das expressões regulares do conjunto seja anulável. A definição a seguir formaliza essa ideia.

**Definição 2.13** (Função de derivada parcial estendida)**.** *Primeiro, vamos estender a função de derivada para uma string:*

$$\begin{array}{rcl}
\nabla^*_\lambda(R) & = & R \\
\nabla^*_{ab}(R) & = & \nabla^*_b(\bigcup_{r \in R}(\nabla_a(r)))
\end{array}$$

*Usando a função estendida para determinar se uma palavra pertence a linguagem definida pela expressão regular, simulando a execução de um AFN:*

$$\begin{array}{rcl}
\hat{\nabla}_\lambda(R) & = & \bigvee_{r \in R}(\nu(r)) \\
\hat{\nabla}_{ab}(R) & = & \hat{\nabla}_b(\bigcup_{r \in R}(\nabla_a(r)))
\end{array}$$

**Exemplo 2.9.**

*Considere a expressão regular* $r = (0+1)^*11$ *e a palavra 1011.*

*Calculando a derivada dessa linguagem com respeito a essa palavra.*

$$\begin{aligned}
&\nabla^*_{1011}((0+1)^*11) &&= \\
&\nabla^*_{011}(\nabla^*_1((0+1)^*11)) &&= \\
&\nabla^*_{011}(\{(0+1)^*11, 1\}) &&= \\
&\nabla^*_{11}(\nabla^*_0(\{(0+1)^*11, 1\})) &&= \\
&\nabla^*_{11}(\{(0+1)^*11\}) &&= \\
&\nabla^*_1(\nabla^*_1(\{(0+1)^*11\})) &&= \\
&\nabla^*_1(\{(0+1)^*11, 1\}) &&= \\
&\nabla^*_\lambda(\nabla^*_1(\{(0+1)^*11, 1\})) &&= \\
&\nabla^*_\lambda(\{(0+1)^*11, 1, \lambda\}) &&= \\
&\{(0+1)^*11, 1, \lambda\}
\end{aligned}$$

*Dessa forma* $\nabla_{1011}((0+1)^*11) = \{(0+1)^*11, 1, \lambda\}$. *E como pelo menos uma das expressões regulares de* $\{(0+1)^*11, 1, \lambda\}$ *é anulável, a palavra 1011 pertence a linguagem definida pela expressão regular.*

#### 2.1.2.4 Construção do Autômato Finito Não-Determinístico equivalente à Expressão Regular

Formalmente o autômato finito não-determinístico equivalente a uma expressão regular $e$ é definido como se segue:

**Definição 2.14.** *Seja* $e$ *uma expressão regular qualquer sobre* $\Sigma$. *O autômato finito não-determinístico equivalente a* $e$, $M = (PD(e), \Sigma, \delta, I, F)$, *em que:*

- $PD(e)$*: conjunto finito de todas as derivadas parciais da expressão regular* $e$.
- $\delta(e, a) = \nabla_a(e)$
- $I = \{e\}$
- $F = \{E \in PD(e) \mid \exists e' \in E \wedge \nu(e') = \lambda\}$

Assim como nas derivadas de Brzozowski, a definição anterior mostra que o conjunto de estados do autômato equivalente é o conjunto de todas as derivadas de uma expressão regular. Transições do autômato são representadas pela aplicação da operação de derivada sobre um símbolo do alfabeto. O conjunto de estados finais são todos os estados correspondentes a expressões regulares anuláveis e o conjunto de estados iniciais contém apenas a expressão regular original. A seguir apresentamos um algoritmo para construção de um AFN usando derivadas parciais.

**Algoritmo 2** Construção de AFN com derivadas parciais de ER

**função** VISITA($e, c, e_c, (E, \delta)$)
    **se** $\exists e' \in E$ tal que $e' \equiv e_c$ **então**
        $(E, \delta \cup \{(e, c) \mapsto e_c\})$
    **senão**
        seja $E' = E \cup \{e_c\}$
        seja $\delta' = \delta \cup \{(e, c) \mapsto e_c\}$
        em *explore*$(E', \delta', e_c)$
    **fim se**
**fim função**

**função** VISITA-TODOS($E, \delta, e, c$)
    seja $e_c$ = $\nabla_c(e)$
    fold (*visita* $e\, c$) $(E, \delta)$ $e_c$
**fim função**

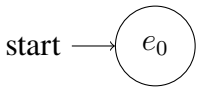
**função** EXPLORE($E, \delta, e$)
    fold (*visita-todos* $e$) $(E, \delta)$ $\Sigma$
**fim função**

**função** CONSTROI-AFD(r)
    seja $i = r$
    seja $(E, \delta)$ = explore $(\{i\}, \{\}, i)$
    seja $F$ = $\{e \mid e \in E \wedge \nu(e) = \lambda\}$
    em $\langle E, \{i\}, F, \delta \rangle$
**fim função**

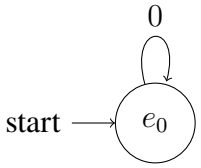
A seguir apresentamos um exemplo de uso deste algoritmo para construção do autômato equivalente a uma expressão regular usando derivadas parciais.

**Exemplo 2.10.**

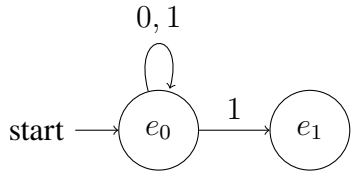
*Considere a expressão regular* $r = (0 + 1)^*11$. *A construção do AFN começa com o estado inicial* $e_0 = r = (0 + 1)^*11$ *e segue da seguinte forma:*




*Calcule* $\nabla_0(e_0) = \nabla_0((0 + 1)^*11) = \{(0 + 1)^*11\} = \{e_0\}$

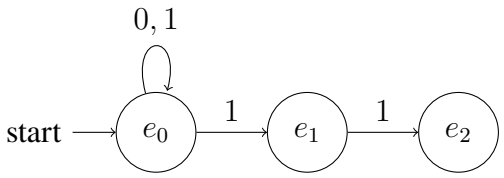


*Calcule* $\nabla_1(e_0) = \nabla_1((0+1)^*11) = \{(0+1)^*11, 1\}$. *A expressão regular* $1$ *é nova, então a chamamos de* $e_1$. *Vamos adicionar um estado e duas transições.*




*Calcule* $\nabla_0(e_1) = \nabla_0(1) = \emptyset$. *Nenhum estado ou transição nova é adicionada.*

*Calcule* $\nabla_1(e_1) = \nabla_1(1) = \{\lambda\}$. *A expressão regular* $\lambda$ *é nova, então a chamamos de* $e_2$



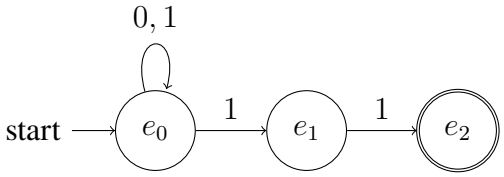

*Calcule* $\nabla_0(e_2) = \nabla_0(\lambda) = \emptyset$. *Nenhum estado ou transição nova é adicionada.*

*Calcule* $\nabla_1(e_2) = \nabla_1(\lambda) = \emptyset$. *Nenhum estado ou transição nova é adicionada.*

*Por fim, olhamos quais estados são anuláveis:*

$nullable(e_0) = nullable((0+1)^*11) = \bot$
$nullable(e_1) = nullable(1) = \bot$
$nullable(e_2) = nullable(\lambda) = \top$

*Como* $e_2$ *é anulável, então ele também é um estado final.*




### 2.1.3 Algoritmos genéticos

Algoritmos genéticos (AG) referem-se a uma meta-heurística inspirada pelo processo de seleção natural que pertence à classe de algoritmos evolutivos. Dada uma população, indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos aptos tendem a desaparecer. Analogamente, algoritmos genéticos, através de operações como seleção, mutação e recombinação, criam soluções cada vez melhores para um determinado problema, comumente de otimização ou busca.

Um algoritmo genético inicia a busca da solução a partir uma população inicial aleatória - a população no tempo 0. A cada geração, uma porção da população existente é escolhida para gerar

uma nova geração e esse processo termina quando: 1) uma solução é encontrada; 2) o número máximo de iterações é atingido; 3) Recursos disponíveis esgotaram; 4) o melhor indivíduo da população alcançou ou vai alcançar um platô e não vai melhorar mais nas próximas iterações; 5) inspeção manual; ou 6) uma combinação dos anteriores.

## 2.2 Introdução à linguagem Racket

A linguagem de programação Racket é uma linguagem de programação funcional dinamicamente tipada e consiste de um dialeto moderno da conhecida linguagem LISP. Assim como outros dialetos de LISP, Racket possui uma sintaxe simples e um sistema de macros que permite incluir novos elementos sintáticos à linguagem.

Uma característica importante de Racket é que sua sintaxe base utiliza as chamadas *S-expressions*, em que o código fonte possui uma estrutura simples que torna a representação de sua árvore de sintaxe como uma lista de símbolos. Ilustraremos esses conceitos utilizando alguns exemplos. A seguir, apresentamos a definição de uma função para cálculo do fatorial utilizando Racket:

**Algoritmo 3** Cálculo de Fatorial em Racket

```
1: (define (factorial n)
2:         (if (= n 0)
3:                 1
4:                 (* n (factorial (- n 1)))))
```

A palavra reservada `define` inicia a definição de uma função de nome `fatorial`, que recebe um argumento de nome `n`. Em linguagens da família LISP, utilizamos parêntesis para representar a chamada de funções, que devem sempre ser feitas como um prefixo de seus argumentos. Por exemplo, a expressão aritmética $n - 1$ é representada, em Racket, como `(- n 1)`, em que o parêntesis é utilizado para representar a chamada da função `-` sobre os argumentos `n` e `1`. Adicionalmente, essa estrutura da chamada de funções permite enxergá-la como uma lista de símbolos em que o primeiro é a função `-`; o segundo, a variável `n` e o último o número 1. Essa representação é o que permite que linguagens da família LISP possuam bons recursos para meta-programação. Em particular, Racket combina o mecanismo de macros com um sistema de módulos para permitir que um mesmo programa seja desenvolvido em diferentes linguagens, que internamente são todas traduzidas para Racket.

Também é possível definir registros usando a palavra reservada `struct`. Por exemplo, o trecho de código seguinte define um registro para representar informações sobre pessoas. No exemplo, o registro pessoa é formado por três campos chamados nome, idade e cidade.

**Algoritmo 4** Estrutura Pessoa

```
1: (struct Pessoa (nome idade cidade))
```

Então, toda vez que for necessário adicionar informações sobre uma pessoa, é criada uma nova estrutura, como `(Pessoa "Guilherme Augusto" 21 "Belo Horizonte")`.

Neste trabalho, adotamos a linguagem Racket por sua simplicidade e extensibilidade que será útil no contexto de criação de uma plataforma para criação e correção automática de exercícios sobre autômatos finitos determinísticos.

## 2.3 Trabalhos relacionados

Como dito anteriormente, estudantes trabalham nos exercícios propostos com papel e lápis, sem um retorno automático sobre suas soluções. Por isso, há um atraso entre um modelo ser introduzido e o aluno receber um retorno sobre tarefas relacionadas. Nesse período de tempo, o curso já avançou muito e os alunos ficam progressivamente mais confusos (Bezáková *et al.*, 2020). Além disso, as turmas podem ser grandes e o professor não consegue tirar as dúvidas de todos os alunos.

Estudantes aprendem melhor vendo representações de um conceito, sejam textuais, visuais ou animadas. Os livros didáticos podem oferecer as representações textuais e algumas visuais, enquanto os softwares podem fornecer representações visuais e animadas. No entanto, apenas observar não é suficiente, os alunos devem ser capazes de interagir com o conceito de alguma forma e receber *feedback* para verificar sua compreensão do conceito (Rodger, 2002).

Chakraborty, Saxena e Katti (2011) listam diversas ferramentas de simulação de autômatos e algumas características sobre elas, como os tipos de autômatos suportados (autômatos finitos, autômatos de pilha, máquinas de Turing), suporte para não-determinismo, suporte para sub-máquinas e linguagem de implementação. Dentre eles, o único que suporta todas essas funcionalidades é o Java Formal Languages and Automata Package (JFLAP) (Rodger, 2002) [1], além de ser uma das mais populares para o ensino de Teoria da Computação. Junto as funcionalidades já citadas, o JFLAP também oferece produto de autômatos, conversão de autômatos finitos não-determinísticos (AFN) para autômatos finitos determinísticos, conversão de AFD para gramáticas e vice-versa, entre outras.

O SimStudio (Chudá; Rodina, 2010) oferece a possibilidade de descrever os autômatos tanto por meio de linguagens simbólicas quanto visualmente, criando o diagrama, porém não possui todas as funcionalidades que o JFLAP oferece. E o Language Emulator (Vieira; Vieira; José, 2003) aceita as especificações de um autômato por meio de uma interface interativa e simula o seu comportamento, mas também não oferece todas as funções que o JFLAP oferece.

Porém, nenhum deles suporta a geração automática de questões. Todos os exercícios propostos para serem resolvidos neles devem ser criados manualmente. Adithi *et al.* (2015) levanta duas questões: (1) é possível o professor transmitir, de maneira não ambígua, a descrição

[1] Disponível em https://jflap.org/

do problema de uma forma segura (i.e., sem revelar sua própria solução)? (2) essa ferramenta pode operar offline (i.e., sem conexão com a internet)? A questão (1) por si só pode ser resolvida permitindo que alunos enviem suas tentativas para um servidor, onde podem ser comparadas com a solução do professor para equivalência, porém tal *feedback* não pode ser obtido por alunos sem conexão confiável à internet.

Mazidi e Nielsen (2014) e Ali, Chali e Hasan (2010) geram perguntas factuais (perguntas do tipo "por que", "quando", "onde", "quem", "quanto", etc.) a partir de uma sentença fonte, fazendo análise semântica e sintática do texto para criar perguntas independentes de domínio (i.e., independente do tema do texto de entrada). Tais perguntas, embora interessantes para determinados casos, não exercitam diretamente a construção e minimização de AFDs e, portanto, não são interessantes para este trabalho.

Shenoy *et al.* (2016) apresenta uma ferramenta que recebe um problema de construção de autômato finito determinístico $P$ como entrada e gera um número arbitrário de problemas $P_1, P_2, ...$ em ordem decrescente de similaridade a $P$. A saída da ferramenta pode ser restringida a problemas que são mais fáceis, mais difíceis ou tão difíceis quanto $P$. A partir de uma base de problemas, composta de 107 problemas "sementes", novos exercícios são gerados combinando e alterando esses problemas. Porém, é mostrado que esses exercícios podem gerar linguagem que não são regulares, ou seja, não seria possível construir um AFD para resolver o exercício.

Owens, Reppy e Turon (2009) apresenta as derivadas de expressões regulares e um sistema de classes de caracteres para testar equivalência de expressões regulares para criar AFDs equivalentes. Seus resultados mostram que os AFDs gerados são muitas vezes ótimos no número de estados e são uniformemente melhores do que aqueles gerados por ferramentas anteriores.

## 2.4 Conclusão

Neste capítulo foi apresentado o referencial teórico deste trabalho: inicialmente foram revisados os principais conceitos sobre teoria de linguagens formais e autômatos finitos. Na sequência, o conceito de derivada e como esse pode ser utilizado para obtenção de autômatos determinísticos a partir de expressões regulares foi apresentado. Adicionalmente, foram apresentados os principais trabalhos relacionados a ferramentas de ensino de linguagens formais e do uso de derivadas de expressões regulares.

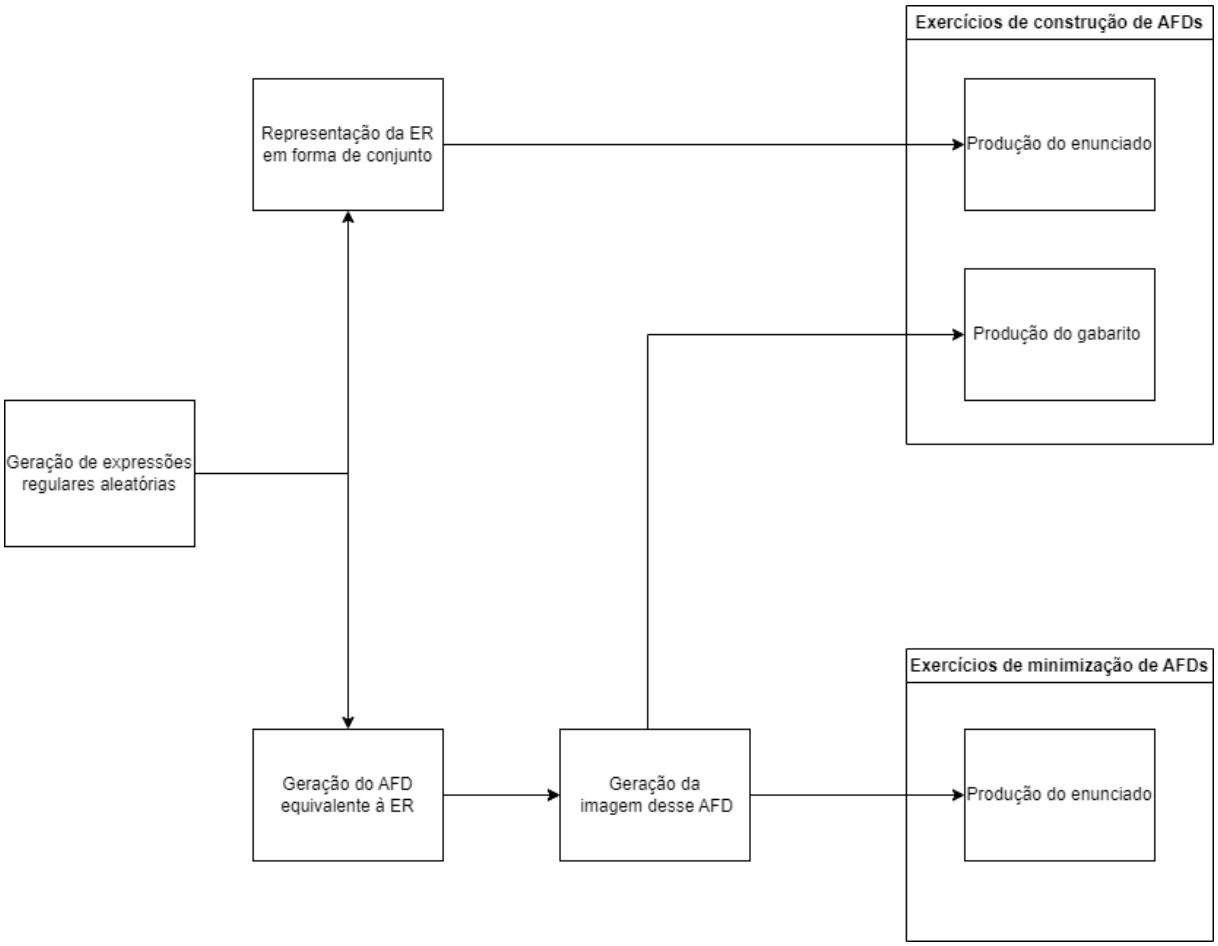
# 3 Desenvolvimento

Neste capítulo é apresentada a versão atual da ferramenta desenvolvida como parte deste trabalho monográfico. A Seção 3.1 apresenta uma visão geral da ferramenta. Detalhes de implementação discutidos na Seção 3.2 e a Seção 3.3 conclui o capítulo.

## 3.1 Visão geral da ferramenta proposta

Nesta seção é apresentada uma visão geral da ferramenta proposta por este trabalho. A Figura 3.1 ilustra os seus principais componentes.

Figura 3.1 – Visão geral da ferramenta




A ferramenta proposta visa produzir dois tipos de questões: 1) elaborar um AFD a partir da especificação de uma linguagem utilizando teoria de conjuntos e 2) obter o autômato mínimo equivalente a um AFD. Para produção de ambos os tipos de questões, primeiro é necessário gerar uma expressão regular aleatória e, a partir desta, construir exercícios do primeiro tipo por

traduzir uma expressão para sua representação utilizando teoria de conjuntos ou do segundo tipo produzindo AFDs.

É importante notar que expressões regulares são também a base para criação de gabaritos para questões. Gabaritos são produzidos por converter uma expressão regular no AFD equivalente, utilizando a construção baseada em derivadas apresentada na Seção 2.1.2.2. A tarefa de produção de gabaritos para minimização de autômatos é deixada como trabalho futuro.

Na próxima seção é discutido detalhes da implementação do processo de geração de exercícios.

## 3.2 Detalhes de implementação

Para a implementação da ferramenta proposta, a linguagem Racket será utilizada. A Seção 3.2.1 descreve as estruturas de dados para representar expressões regulares e algumas funções sobre esses tipos. A Seção 3.2.2 apresenta a implementação de um algoritmo para obter um autômato finito determinístico equivalente a uma expressão regular. Em seguida, na Seção 3.2.3, é apresentada uma estratégia para geração aleatória de expressões regulares e também a necessidade de renomeamento de estados para permitir a visualização de autômatos utilizando imagens produzidas pela ferramenta GraphViz. Por fim, na Seção 3.2.6, é apresentada a geração das questões.

### 3.2.1 Operações sobre expressões regulares

Expressões regulares são representadas pelo seguinte conjunto de registros, apresentados no Algoritmo 5:

**Algoritmo 5** Estruturas para representar uma ER

```
(struct EMPTY ())
(struct LAMBDA ())
(struct SYMBOL (a))
(struct CONCATENATION (r s))
(struct KLEENE-CLOSURE (r))
(struct UNION (r s))
(struct INTERSECTION (r s))
(struct COMPLEMENT (r))
```

As estruturas `EMPTY` e `LAMBDA` representam as expressões regulares $\emptyset$ e $\lambda$, respectivamente. A estrutura `SYMBOL (a)` representa o símbolo `a`, tal que `a` é um caractere. As estruturas `CONCATENATION (r s)`, `UNION (r s)`, `KLEENE-CLOSURE (r)` e `COMPLEMENT (r)` representam, respectivamente, as operações sobre ERs concatenação ($r \cdot s$), união ($r + s$), fecho de Kleene ($r^*$) e complemento ($\neg r$).

**Exemplo 3.1.**

*A expressão regular* $L = (0 + 1)^*11$ *pode ser representada como*

*(CONCATENATION*
*(KLEENE−CLOSURE (UNION (SYMBOL 0) (SYMBOL 1)))*
*(CONCATENATION (SYMBOL 1) (SYMBOL 1)))*

Em seguida, a implementação dos algoritmos para o cálculo da derivada de expressões regulares é apresentada. Primeiramente, a função $\nu(r)$ e um trecho da função $nullable(r)$ são apresentados a seguir, no Algoritmo 6, onde $e$ é uma expressão regular qualquer:

**Algoritmo 6** Cálculo de anulabilidade de uma ER

```
1: (define (nullable? e)
2:     (match e
3:       [(EMPTY) #f]
4:       [(LAMBDA) #t]
5:       [(SYMBOL a) #f]
6:       [(CONCATENATION r s) (and (nullable? r) (nullable? s))]
7:       [(KLEENE-CLOSURE r) #t]
8:       ...))

1: (define (v e)
2:     (if (nullable? e) (LAMBDA) (EMPTY)))
```

A definição de $nullable$ utiliza casamento de padrão para representar a função descrita na Definição 2.8. O cálculo de derivadas segue a mesma estrutura. Alguns trechos da definição de derivada são apresentados a seguir, no Algoritmo 7:

**Algoritmo 7** Cálculo da derivada de uma ER

```
1: (define (derivative a e)
2:     (match e
3:       [(EMPTY) (EMPTY)]
4:       [(LAMBDA) (EMPTY)]
5:       [(SYMBOL b) (if (char=? a b) (LAMBDA) (EMPTY))]
6:       [(CONCATENATION r s) (union-regex
7:          (concat-regex (derivative a r) s)
8:          (concat-regex (v r) (derivative a s)))]
9:       ...))
```

As função `union-regex` e `concat-regex` são funções auxiliares para realizar simplificações sobre expressões regulares. Por exemplo, a função `union-regex` transforma expressões da forma $e + \emptyset$ em $e$ e `concat-regex` reescreve $e\lambda = e$ e $e\emptyset = \emptyset$. Tais igualdades preservam a semântica da expressão obtida e evitam o acréscimo de estados desnecessários durante a criação do autômato.

**Exemplo 3.2.** *A expressão regular* $L = a+b$ *poderia ser representada como* (UNION (SYMBOL 'a') (SYMBOL 'b')).

*Sua derivada com respeito ao símbolo* $a$ *é* $\lambda$ (LAMBDA).

*Porém, sem essas funções auxiliares, a função retornaria* (UNION (LAMBDA) (EMPTY)).

*Esse resultado não está errado, pois as expressões* (LAMBDA) *e* (UNION (LAMBDA) (EMPTY)) *são equivalentes, mas dificulta o processo de testar equivalência de ERs durante o processo de geração do AFD.*

Trechos da implementação de `union-regex` e `concat-regex` são apresentados a seguir, no Algoritmo 8:

**Algoritmo 8** Funções auxiliares para simplificação de uma ER

```
(define (union-regex r s)
  (match (cons r s)
    ...
    [(cons _ (EMPTY)) r]
    [(cons (LAMBDA) e) (if (nullable? e) e (UNION r s))]
    ...
    [any (if (equivalent? r s)
       r
       (UNION r s))]))
```

```
(define (concat-regex r s)
  (match (cons r s)
    ...
    [(cons _ (EMPTY)) (EMPTY)]
    [(cons (LAMBDA) _) s]
    ...
    [any (CONCATENATION r s)]))
```

Além das funções `union-regex` e `concat-regex`, também foram criadas as funções `inter-regex`, `kleene-regex` e `complement-regex`. Estas funções reescrevem igualdades envolvendo as operações de interseção, fecho de Kleene e complemento.

Um componente importante do processo de simplificação de expressões regulares é determinar quando duas expressões regulares são ou não equivalentes. A seguir, no Algoritmo 9, são apresentados trechos de código da função que realiza esse teste. Intuitivamente, a função percorre a estrutura recursiva das expressões para determinar se estas são ou não equivalentes.

**Algoritmo 9** Equivalência de ERs

```
(define (equivalent? r s)
  (match (cons r s)
    ...
    [(cons (EMPTY) (EMPTY)) #t]
    [(cons (SYMBOL a) (SYMBOL b)) (char=? a b)]
    ...
    [(cons (UNION a b) (UNION c d))
      (or
        (and (equivalent? a c) (equivalent? b d))
        (and (equivalent? a d) (equivalent? b c)))]
    ...
    [(cons (COMPLEMENT a) (COMPLEMENT b)) (equivalent? a b)]
    [any #f]))
```

Porém, simplesmente percorrer a estrutura recursiva de expressões para o teste de equivalência não é suficiente. Algumas operações sobre expressões regulares são comutativas (união e interseção) e associativas (união, interseção e concatenação) o que impede uma implementação direta do teste. Para contornar esse problema, foi adicionada uma etapa de reescrita que aninha expressões evolvendo concatenação, união e interseção à direita. Parte do código deste sistema de reescrita é apresentado a seguir, no Algoritmo 10:

**Algoritmo 10** Reescrita de uma ER

```
(define (rewrite-re e)
  (match e
    ...
    [(CONCATENATION r s) (list-to-concat
                                (map rewrite-re
                                  (concat-to-list e)))]
    [(UNION r s) (list-to-union
                  (remove-duplicates
                    (sort
                      (map rewrite-re
                        (union-to-list e))
                      regex-smaller?)
                    equivalent?))]
    ...
    [(COMPLEMENT r) (complement-regex (rewrite-re r))]))
```

Para aninhar a concatenação a direita, é usada o par de funções `concat-to-list` e `list-to-concat`. A função `concat-to-list` converte uma árvore de concatenação em uma lista que contém todas as folhas, enquanto a função `list-to-concat` converte essa lista em uma concatenação, com os elementos aninhados a direita. Elas são apresentadas no Algoritmo 11.

**Algoritmo 11** Aninhamento à direita de uma ER

```
1: (define (union-to-list e)
2:     (match e
3:         [(UNION r s) (append (union-to-list r) (union-to-list s))]
4:         [any (list e)]))
```

```
1: (define (list-to-union l)
2:     (match l
3:         [(list) (EMPTY)]
4:         [(list r) r]
5:         [(list r s) (union-regex r s)]
6:         [(cons x xs) (union-regex x (list-to-union xs))]))
```

As funções `inter-to-list` e `list-to-inter` (para interseção) e `concat-to-list` e `list-to-concat` (para concatenação) funcionam da mesma forma.

**Exemplo 3.3.** *A regex*

*(CONCATENATION*
*(CONCATENATION (SYMBOL 'a') (UNION (SYMBOL 'a') (SYMBOL 'b')))*
*(CONCATENATION (SYMBOL 'c') (SYMBOL 'b')))*

*seria reescrita na forma de*

*(CONCATENATION*
*(SYMBOL 'a')*
*(CONCATENATION*
*(UNION (SYMBOL 'a') (SYMBOL 'b'))*
*(CONCATENATION (SYMBOL 'c') (SYMBOL 'b'))))*

Além disso, para união e interseção, dois passos adicionais são feitos: 1) a ordenação dos elementos de forma arbitrária (e igual para todas as ERs), para que o teste de equivalência possa ser feito percorrendo linearmente as expressões regulares e 2) a remoção de elementos duplicados que são equivalentes.

Sobre o ponto 1), a ordenação pode ser feita pois essas operações são associativas e comutativas e é feita usando a função `regex-smaller?`, que impõe uma ordem sobre expressões regulares. A ordem estabelecida por `regex-smaller?` determina a seguinte prioridade: símbolo, concatenação, interseção, união, complemento, fecho de Kleene, $\lambda$ e $\emptyset$. Expressões regulares do mesmo tipo são ordenadas lexicograficamente pelo primeiro símbolo que elas contém. Um trecho da função é apresentado no Algoritmo 12.

**Algoritmo 12** Comparação de ERs

```
(define (regex-smaller? x y)
  (match (cons x y)
    [(cons (EMPTY) _) #f]
    [(cons _ (EMPTY)) #t]
    ...
    [(cons (SYMBOL a) (SYMBOL b)) (char<? a b)]
    [(cons (SYMBOL _) s) #t]

    [(cons (KLEENE-CLOSURE r) (KLEENE-CLOSURE s)) (regex-smaller? r s)]
    [(cons (KLEENE-CLOSURE r) _) #f]
    [(cons _ (KLEENE-CLOSURE r)) #t]
    ...
    [(cons (CONCATENATION r s) (CONCATENATION r1 s1)) (regex-smaller? r r1)]
    [(cons (CONCATENATION r s) _) #f]
    [(cons _ (CONCATENATION r s)) #t]))
```

O ponto 2) é necessário pois, embora as funções auxiliares mencionadas anteriormente já simplifiquem idempotência, como os elementos são aninhados a direita, elementos equivalentes podem ficar em níveis diferentes da árvore.

**Exemplo 3.4.** *A expressão regular*

*(UNION*
*(SYMBOL 'a')*
*(UNION (SYMBOL 'a') (SYMBOL 'b')))*

*pode ser simplificada para*

*(UNION (SYMBOL 'a') (SYMBOL 'b'))*

*mas como* (SYMBOL 'a') $\not\equiv$ (UNION (SYMBOL 'a') (SYMBOL 'b'))*, a função* union-regex *não faz essa redução.*

### 3.2.2 Construção do AFD equivalente à ER

A função `re-sigma-to-dfa` gera o AFD a partir de uma ER qualquer e do alfabeto, representado por uma lista de caracteres. A função `mk-dfa2` age como um construtor, retornando uma estrutura que representa o AFD e é apresentada no Algoritmo 13.

**Algoritmo 13** Construção do AFD equivalente à ER

```
1: (define (re-sigma-to-dfa sigma re)
2:     (define start re)
3:     (define graph (mk-graph sigma (fa-graph (list start) '()) start))
4:     (define Q (fa-graph-states graph))
5:     (define d (fa-graph-transitions graph))
6:     (define F (filter nullable? Q))
7:     (mk-dfa2 Q sigma d start F))
```

`fa-graph` é um estrutura que representa o grafo criado ao longo do processo de geração do AFD por meio de derivadas. Inicialmente esse grafo começa apenas com um único estado (que é o estado inicial do AFD) e nenhuma transição. A função `mk-graph` explora esse grafo por meio de uma busca em profundidade e é apresentada no Algoritmo 14.

**Algoritmo 14** Construção do AFD com derivadas

```
1: (define (mk-transition sigma q c graph)
2:     (define qc (rewrite-re (derivative c q)))
3:     (if (state-exists qc graph)
4:         (add-transition (cons (cons q c) qc) graph)
5:         (let ([graph-n (add-transition (cons (cons q c) qc) (add-state qc graph))])
6:             (mk-graph sigma graph-n qc))))
```

```
1: (define (mk-graph sigma graph q)
2:     (foldl (curry mk-transition sigma q)
3:     graph
4:     sigma))
```

Para obter o alfabeto de uma ER foi criada a função `get-alphabet`, que retorna uma lista de todos os caracteres distintos que estão na ER.

### 3.2.3 Geração de uma ER

Para produzir ERs foi adotada uma abordagem simples: expressões geradas aleatoriamente. Para isso, a biblioteca de testes baseados em propriedades, rackcheck foi utilizada. A escolha de uso desta biblioteca é motivada pelo fato desta possuir geradores de valores aleatórios para diversos tipos primitivos da linguagem Racket e funções para combinar geradores para criar funções que produzem tipos de dados quaisquer. Um trecho da função é apresentado no Algoritmo 15.

**Algoritmo 15** Geração de uma ER aleatória

```
(define gen:symbol
   (gen:one-of (list
            (LAMBDA)
            (SYMBOL '1')
            (SYMBOL '0'))))
```

```
(define (gen:re h)
   (if (<= h 1)
      gen:symbol
      (let* ([h2 (quotient h 2)])
         (gen:frequency `((25 . ,(gen:let ([r (gen:re h2)]
                  [s (gen:re h2)])
                  (gen:const (UNION r s))))
            ...
            (10 . ,(gen:let ([r (gen:re h2)])
                  (gen:const (COMPLEMENT r)))))))))
```

A função `gen:symbol` gera um símbolo (a, b, c, 0 ou 1), $\lambda$ ou $\emptyset$. A função `gen:re` gera uma ER e recebe como entrada a altura máxima `h` da ER que deve ser gerada. Se `h` for menor ou igual a 1, será gerado um símbolo, $\lambda$ ou $\emptyset$. Caso contrário, será gerado aleatoriamente uma união (com frequência de 40), uma interseção (com frequência de 5), uma concatenação (com frequência de 30), um fecho de Kleene (com frequência de 15) ou um complemento (com frequência de 10). Para gerar as sub-expressões é usada recursivamente a função `gen:re`, dividindo o valor de `h` por 2.

### 3.2.4 Renomeação dos estados de um AFD

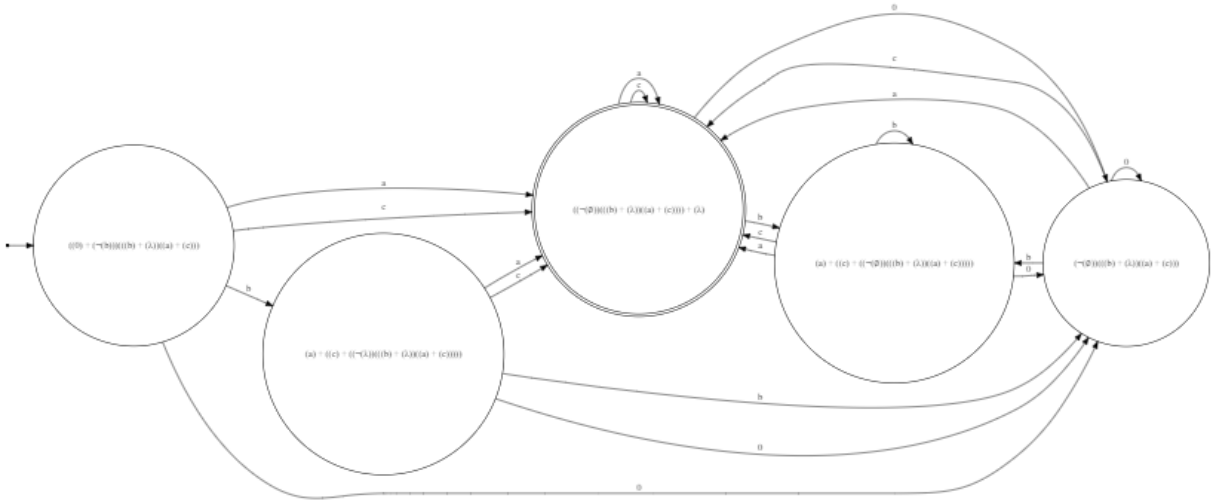
Para gerar imagens com o intuito de criar exercícios do tipo "minimize o autômato", os autômatos gerados com a função `re-to-dfa` criam imagens muito difíceis de serem lidas, pois as expressões regulares podem ficar grandes, fazendo com que o tamanho dos estados gerados pela ferramenta `dot` também fique muito grande.

**Exemplo 3.5.** *A expressão regular* $(0 + \neg b) \cdot (b + \lambda) \cdot (a + c)$*, gerada pelo sistema apresentado anteriormente, representada por meio das estruturas como:*

```
(CONCATENATION
   (UNION (SYMBOL '0') (COMPLEMENT (SYMBOL 'b')))
   (CONCATENATION
      (UNION (SYMBOL 'b') (LAMBDA))
      (UNION (SYMBOL 'a') (SYMBOL 'c'))))
```

*criaria o seguinte AFD (Figura 3.2):*

Figura 3.2 – AFD sem renomeamento dos estados



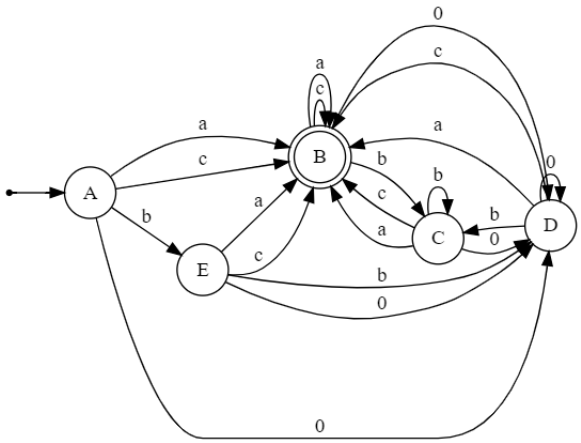
Para melhorar a leitura do AFD, foi desenvolvida uma função para renomear os estados do AFD. A função `dfa-rename-states` retorna o AFD com os estados renomeados para letras maiúsculas (A, B, C .. Z) e uma tabela hash indicadando qual ER cada letra representa. O código da função é apresentado no Algoritmo 16.

**Algoritmo 16** Renomeação dos estados de um AFD

```
(define (dfa-rename-states dfa)
  (define Q (dfa-states dfa))
  (define sigma (dfa-sigma dfa))
  (define d (dfa-delta dfa))
  (define start (dfa-start dfa))
  (define F (dfa-final dfa))
  (define table (hash-states Q))
  (define Q2 (map (curry hash-ref table) Q))
  (define d2 (map
               (lambda (t) (cons
                 (cons (hash-ref table (car (car t)))
                   (cdr (car t)))
                 (hash-ref table (cdr t))))
               d))
  (define start2 (hash-ref table start))
  (define F2 (map (curry hash-ref table) F))
  (define dfa2 (mk-dfa2 Q2 sigma d2 start2 F2))
  (list dfa2 table))
```

**Exemplo 3.6.** *Com o uso da função de renomeamento, a expressão regular* $(0+\neg b)\cdot(b+\lambda)\cdot(a+c)$*, gera o seguinte AFD (Figura 3.3):*

Figura 3.3 – AFD com renomeamento dos estados




*A tabela a seguir indica qual expressão regular cada estado representa:*

| Estado | Expressão regular |
|---|---|
| A | $((0) + (\neg(b)))(((b) + (\lambda))((a) + (c)))$ |
| B | $((\neg(\emptyset))(((b) + (\lambda))((a) + (c)))) + (\lambda)$ |
| C | $(a) + ((c) + ((\neg(\emptyset))(((b) + (\lambda))((a) + (c)))))$ |
| D | $(\neg(\emptyset))(((b) + (\lambda))((a) + (c)))$ |
| E | $(a) + ((c) + ((\neg(\lambda))(((b) + (\lambda))((a) + (c)))))$ |

Tabela 3.1 – Relação entre estados e expressões regulares que eles representam.

### 3.2.5 Operações de *crossover* e mutação

Para explicar o funcionamento dessas operações, considere o seguinte: as estruturas usadas para representar as operações levam, de forma simples, a uma representação de árvore das expressões regulares.

**Exemplo 3.7.**

*A representação usando as estruturas e a árvore da expressão regular* $(0 + 1)^*11$.

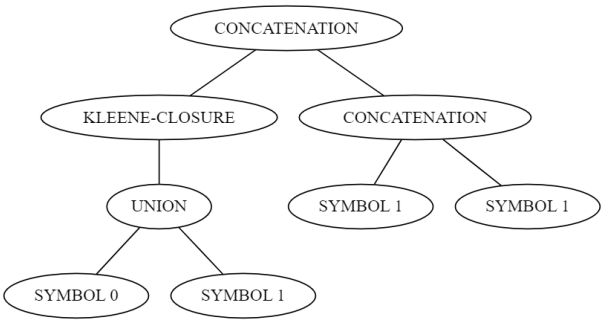
*(CONCATENATION*
*(KLEENE−CLOSURE (UNION (SYMBOL 0) (SYMBOL 1)))*
*(CONCATENATION (SYMBOL 1) (SYMBOL 1)))*

Dessa forma, o *crossover* é feito trocando duas subárvores das árvores das expressões $r$ e $s$.

**Exemplo 3.8.** *Considere as expressões* $r = (0 + 1)^*11$ *e* $s = 10 + 01$.

*As Figuras 3.5 e 3.6 apresentam as árvores das expressões* $r$ *e* $s$*, respectivamente. Os nós preenchidos em cinza são os nós raízes das sub árvores trocadas.*

Figura 3.4 – Árvore da expressão $(0+1)^*11$




*As Figuras 3.7 e 3.8 apresentam as árvores geradas pelo crossover:* $r' = (10)^*11$ *e* $s' = 0+1+01$.

Figura 3.5 – Árvore da expressão $r = (0+1)^*11$

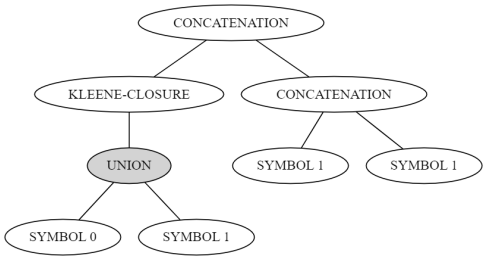



Figura 3.6 – Árvore da expressão $s = 10+01$

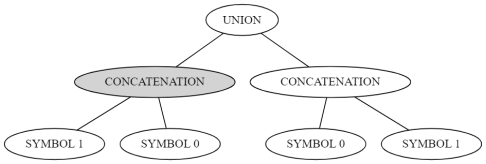



Figura 3.7 – Árvore da expressão $r' = (10)^*11$

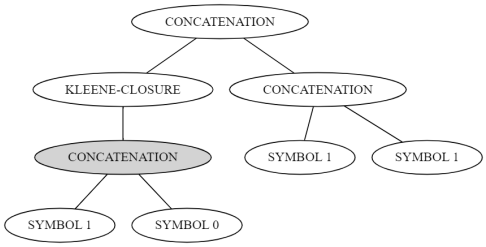
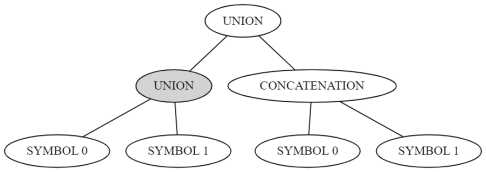



Figura 3.8 – Árvore da expressão $s' = 0+1+01$




E a mutação é feita trocando a operação, caso seja um nó interno da árvore, ou o símbolo, caso seja um nó folha.

**Exemplo 3.9.** *Considere a expressão* $r = (0+1)^*11$.

*A Figura 3.9 apresenta a árvore dessa expressão e a Figura 3.10 apresenta o resultado da mutação,* $r' = (0+1)^* + 11$. *O nó preenchido em cinza foi o nó que sofreu mutação.*

A seguir, é apresentada uma forma alternativa para realizar o *crossover*: dada duas expressões $r$ e $s$ e os nós $A_r$, $B_r$, $A_s$ e $B_s$, de forma que $B_r$ esteja na sub árvore cuja raiz é $A_r$ e

Figura 3.9 – Árvore da expressão $r = (0 + 1)^*11$

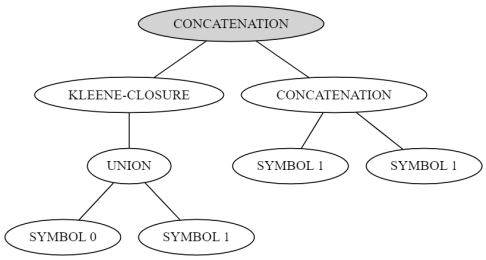
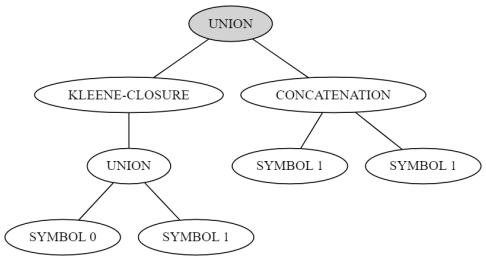



Figura 3.10 – Árvore da expressão $r' = (0 + 1)^* + 11$




$B_s$ esteja na sub árvore cuja raiz é $A_s$, trocando o trecho das árvores que se encontra entre $A_r$ e $B_r$ e entre $A_s$ e $B_s$ nas duas árvores, incluindo $A_r$, $B_r$, $A_s$ e $B_s$.

**Exemplo 3.10.** *Considere as expressões* $r = (11 + 0)^*0$ *e* $s = (\neg(000))^*$.

*As Figuras 3.11 e 3.12 apresentam as árvores das expressões* $r$ *e* $s$*, respectivamente. Os nós preenchidos em cinza são os nós que delimitam o trecho que será trocado.*

*As Figuras 3.13 e 3.14 apresentam as árvores geradas pelo crossover:* $r' = \neg(110)0$ *e* $s' = (((0 + 0)0)^*)^*$.

Figura 3.11 – Árvore da expressão $r = (11 + 0)^*0$

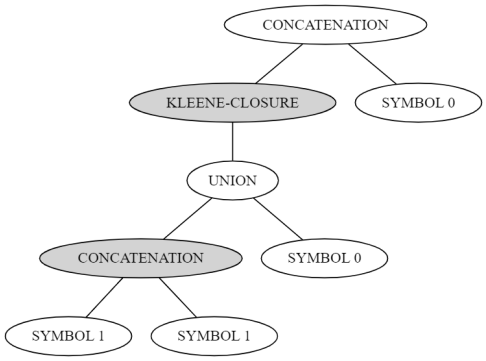
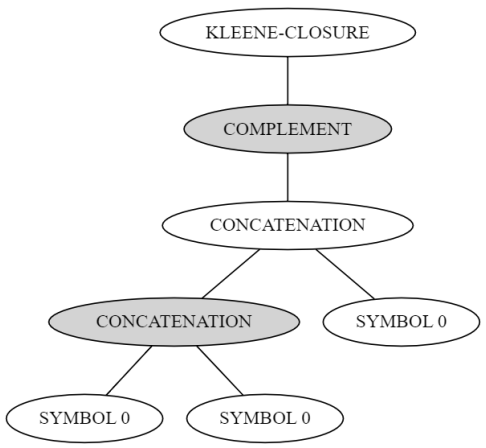



Figura 3.12 – Árvore da expressão $s = (\neg(000))^*$




### 3.2.6 Criação das questões

Por terem enunciados similares, apenas as expressões são necessárias para criação das questões. Basta variar a expressão sobre a qual o exercício trata para criar novos exercícios.

No Algoritmo 17, são apresentadas algumas funções auxiliares para o desenvolvimento do algoritmo genético. As funções `objective-easy`, `objective-medium` e `objective-hard` são funções objetivos do algoritmo genético, que pode gerar questões com dificuldade variada. A dificuldade da questão é definida pelo número de estados do autômato equivalente a expressão

Figura 3.13 – Árvore da expressão $r' = \neg(110)0$

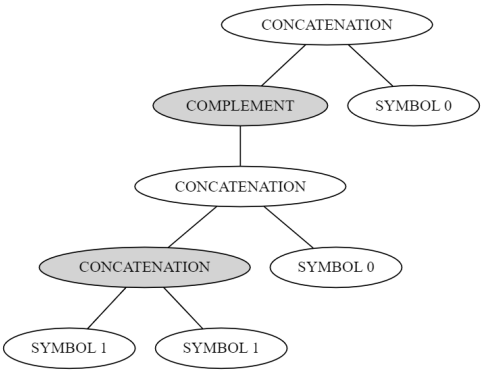
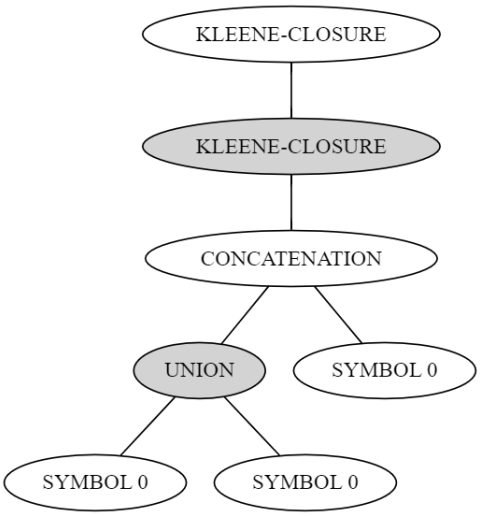



Figura 3.14 – Árvore da expressão $s' = (((0 + 0)0)^*)^*$




regular do enunciado da questão. Um número de estados menor ou igual a quatro é considerado um exercício fácil, de cinco a oito estados é considerado um exercício médio e de nove a doze é considerado um exercício difícil. O limite de 12 estados para questões difíceis existe para não criar autômatos com estados demais, o que torna a resolução do exercício em um processo mais tedioso do que desafiador.

Não foram encontrada na literatura formas de avaliar a dificuldade a criação do AFD equivalente a expressão regular e, por isso, foi utilizado esse método. Porém, usando esse método, autômatos finitos com muitos estados que apresentam a forma de uma lista seriam categorizados como difíceis, mesmo que na realidade sejam simples, por serem lineares.

A função `initial-population` cria uma nova população inicial para o algoritmo genético, de tamanho *size*. As funções `fitnessess`, `population-fitness` e `best-individual` recebem como argumento qual função de *fitness* será usada para fazer o cálculo. A função `fitnessess` calcula a *fitness* de uma população e retorna uma lista com a *fitness* de cada expressão, enquanto a função `population-fitness` retorna um único inteiro, que representa a *fitness* da própria população, calculada pela soma das *fitnesses* das expressões que a compõe. A função `best-individual` retorna o melhor indivíduo de uma população.

**Algoritmo 17** Funções auxiliares para o Algoritmo Genético

```
1: (define (objective-easy re)
2:     (if (<= (number-states re) LIMIT-EASY) 0 1))

1: (define (objective-medium re)
2:     (if (<= (add1 LIMIT-EASY) (number-states re) LIMIT-MEDIUM) 0 1))

1: (define (objective-hard re)
2:     (if (<= (add1 LIMIT-MEDIUM) (number-states re) LIMIT-HARD) 0 1))

1: (define (fitnesses fitness-function population)
2:     (map fitness-function population))

1: (define (population-fitness fitness-function population)
2:     (apply + (fitnesses fitness-function population)))

1: (define (best-individual population fitness-function)
2:     (first (sort population (curry sort-by fitness-function))))

1: (define (initial-population size [re-length MAX-RE-LENGTH])
2:     (sample (gen:re re-length) size))
```

O *crossover* de duas ERs foi definido como a troca entre subexpressões das expressões. O Algoritmo 18 apresenta o código desenvolvido para realizar o *crossover*. A função `choose-path` cria uma lista de 0's e 1's, que representa um caminho na árvore: 0 representa caminhar pelo filho a esquerda e 1 pelo filho a direita. A função `get-subtree-path`, dada uma ER e um caminho *path*, retorna a sub-árvore no fim do caminho *path* e a função `replace-subtree-path` substitui a sub-árvore no fim do caminho pela sub-árvore *new*. Por fim, a função `crossover-path` recebe duas expressões $r$ e $s$ e retorna as novas expressões geradas pelo *crossover*.

**Algoritmo 18** Funções auxiliares para o Algoritmo Genético

```
(define (choose-path r s)
    (define smaller (min (re-height r) (re-height s)))
    (define length (random 1 (add1 smaller)))
    (build-list length (lambda (x) (random 2))))
```

```
(define (get-subtree-path re path)
    (if (empty? path)
        re
        (let* ([head (first path)] [tail (rest path)])
            (match re
                [(EMPTY) re]
                [(LAMBDA) re]
                [(SYMBOL _) re]
                [(CONCATENATION r s) (get-subtree-path (if (= head 0) r s) tail)]
                [(UNION r s) (get-subtree-path (if (= head 0) r s) tail)]
                [(INTERSECTION r s) (get-subtree-path (if (= head 0) r s) tail)]
                [(KLEENE-CLOSURE r) (get-subtree-path r tail)]
                [(COMPLEMENT r) (get-subtree-path r tail)]))))
```

```
(define (replace-subtree-path re new path)
    (if (empty? path)
        new
        (let* ([head (first path)] [tail (rest path)])
            (match re
                [(EMPTY) new]
                [(LAMBDA) new]
                [(SYMBOL _) new]
                [(CONCATENATION r s) (if (= head 0)
                        (CONCATENATION (replace-subtree-path r new tail) s)
                        (CONCATENATION r (replace-subtree-path s new tail)))]
                [(UNION r s) (if (= head 0)
                        (UNION (replace-subtree-path r new tail) s)
                        (UNION r (replace-subtree-path s new tail)))]
                [(INTERSECTION r s) (if (= head 0)
                        (INTERSECTION (replace-subtree-path r new tail) s)
                        (INTERSECTION r (replace-subtree-path s new tail)))]
                [(KLEENE-CLOSURE r) (replace-subtree-path r new tail)]
                [(COMPLEMENT r) (replace-subtree-path r new tail)]))))
```

```
(define (crossover-path r s)
    (define path (choose-path r s))
    (define r-subtree (get-subtree-path r path))
    (define s-subtree (get-subtree-path s path))
    (define r2 (replace-subtree-path r s-subtree path))
    (define s2 (replace-subtree-path s r-subtree path))
    (list r2 s2))
```

A mutação de uma ER foi definida como a alteração de uma subexpressão da ER. O Algoritmo 19 apresenta o código desenvolvido para realizar a mutação. A função `choose-path-mutate` cria um caminho, nos moldes da função choose-path, mas recebe apenas uma expressão como argumento. A função `mutate-subtree-path`, dada uma ER e um caminho *path*, altera a subárvore no fim do caminho *path*. Por fim, a função `mutate` recebe uma expressão $r$ e retorna a nova expressão gerada pela mutação e a função `mutate-population` aplica o processo de mutação em cima de uma população.

**Algoritmo 19** Funções auxiliares para o Algoritmo Genético

```
(define (mutate-subtree-path re path)
  (if (empty? path)
    (match re
      [(EMPTY) re]
      [(LAMBDA) re]
      [(SYMBOL _) re]
      [(CONCATENATION r s) (if (nullable? r)
        (if (< (random) 0.1) (UNION r s) (INTERSECTION r s))
        (if (< (random) 0.1) (INTERSECTION r s) (UNION r s)))]
      [(UNION r s)
        (if (< (random) 0.1) (INTERSECTION r s) (CONCATENATION r s)) ]
      [(INTERSECTION r s)
        (if (< (random) 0.1) (UNION r s) (CONCATENATION r s))]
      [(KLEENE-CLOSURE r)
        (if (= 0 (random 2)) (KLEENE-CLOSURE r) (COMPLEMENT r))]
      [(COMPLEMENT r)
        (if (= 0 (random 2)) (KLEENE-CLOSURE r) (COMPLEMENT r))])
    (let* ([head (first path)]
        [tail (rest path)])
      (match re
      [(EMPTY) re]
      [(LAMBDA) re]
      [(SYMBOL _) re]
      [(CONCATENATION r s) (if (= head 0)
        (CONCATENATION (mutate-subtree-path r tail) s)
        (CONCATENATION r (mutate-subtree-path s tail)))]
      [(UNION r s) (if (= head 0)
        (UNION (mutate-subtree-path r tail) s)
        (UNION r (mutate-subtree-path s tail)))]
      [(INTERSECTION r s) (if (= head 0)
        (INTERSECTION (mutate-subtree-path r tail) s)
        (INTERSECTION r (mutate-subtree-path s tail)))]
      [(KLEENE-CLOSURE r) (KLEENE-CLOSURE (mutate-subtree-path r tail))]
      [(COMPLEMENT r) (COMPLEMENT (mutate-subtree-path r tail))]))))
```

```
(define (choose-path-mutate r)
  (define length (random 1 (add1 (re-height r))))
  (build-list length (lambda (x) (random 2))))
```

```
(define (mutate r)
  (define path (choose-path-mutate r))
  (mutate-subtree r path))
```

```
(define (mutate-population population)
  (map (lambda (x) (if (< (random) MUTATION-RATE) (mutate x) x)) population))
```

No Algoritmo 20, é apresentado o código do algoritmo genético. A função `crossover-torneio` realiza o *crossover* na população, usando a função `torneio-escolhe-pai`, que recebe um argumento $n$, indicando o número de indivíduos no torneio, para selecionar, permitindo repetição, os indivíduos que serão usados. A função `new-population` gera uma nova população usando a função `gen-torneio`, que realiza $\frac{L}{2}$ *crossovers*, onde $L$ é o tamanho da população. A função `stop` define os critérios de parada do algoritmo genético e a função `gen` executa o processo do algoritmo genético como um todo. E, por fim, a função `generate-questions`, apresentada no Algoritmo 21, cria as expressões usadas para montar o enunciado das questões.

**Algoritmo 20** Algoritmo genético

```
(define (torneio-escolhe-pai population fitness-function n)
    (define positions (build-list n (lambda (x) (random (length population)))))
    (define candidatos (map (lambda (p) (list-ref population p)) positions))
    (best-individual candidatos fitness-function))

(define (crossover-torneio population step fitness-function n)
    (define pai1 (torneio-escolhe-pai population fitness-function n))
    (define pai2 (torneio-escolhe-pai (remove pai1 population) fitness-function n))
    (define novo-par (if (< (random) CROSSOVER-RATE)
        (crossover-path pai1 pai2)
        (list pai1 pai2)))
    (if (= step 0)
        (list)
        (append novo-par (crossover-torneio population (sub1 step) fitness-function n)))))

(define (gen-torneio population fitness-function [n 2])
    (mutate-population
        (crossover-torneio
            population
            (quotient (length population) 2)
            fitness-function
            n)))

(define (new-population population fitness-function)
    (gen-torneio population fitness-function))

(define (stop population fitness-function current-generation)
    (or
        (= (population-fitness fitness-function population) 0)
        (= current-generation MAX-GENERATIONS)))

(define (gen population fitness-function [current-generation 0] [return-generation #f])
    (if (stop population fitness-function current-generation)
        (if return-generation (list population current-generation) population)
            (gen
                (new-population population fitness-function)
                fitness-function
                (add1 current-generation)
                return-generation)))
```

**Algoritmo 21** Geração das questões

```
1: (define (generate-questions number-easy number-medium number-hard [re-length MAX-RE-
   LENGTH])
2:   (append
3:     (build-list
4:       number-easy
5:         (lambda (x)
6:           (best-individual
7:             (gen (initial-population NUMBER-QUESTIONS re-length) objective-easy)
8:             objective-easy)))
9:     (build-list
10:      number-medium
11:        (lambda (x)
12:          (best-individual
13:            (gen (initial-population NUMBER-QUESTIONS re-length) objective-
   medium)
14:            objective-medium)))
15:    (build-list
16:      number-hard
17:        (lambda (x)
18:          (best-individual
19:            (gen (initial-population NUMBER-QUESTIONS re-length) objective-hard)
20:              objective-hard)))))
```

### 3.2.7 Criação do Jupyter Notebook

O Jupyter Notebook é um arquivo JSON no formato descrito em <https://nbformat.readthedocs.io/en/latest/format_description.html>. Para trabalhar com JSON em Racket, foi utilizada a biblioteca JSON, que representa o JSON como uma tabela hash. A biblioteca usa o termo *jsexpr* para se referir a representação interna de um JSON.

No Algoritmo 22, é apresentada as funções `re->jsexpr`, que converte uma ER para *jsonexpr*, `re->json`, que retorna uma *jsexpr* contendo a própria expressão, a expressão convertida para uma *jsexpr* e a representação em texto da expressão. A função `re-list->jsexpr` gera e converte as expressões que serão usadas para montar a lista de exercícios. Por fim, a função `re-as-string` transforma a representação usada para ER em um texto.

**Algoritmo 22** Conversão de ER para *jsexpr*

```
(define (re->jsexpr re)
  (match re
    [(EMPTY) (hasheq 'EMPTY "EMPTY")]
    [(LAMBDA) (hasheq 'LAMBDA "LAMBDA")]
    [(SYMBOL a) (hasheq 'SYMBOL (string a))]
    [(CONCATENATION r s)
      (hasheq 'CONCATENATION (list (re->jsexpr r) (re->jsexpr s)))]
    [(KLEENE-CLOSURE r)
      (hasheq 'KLEENE-CLOSURE (re->jsexpr r))]
    [(UNION r s)
      (hasheq 'UNION (list (re->jsexpr r) (re->jsexpr s)))]
    [(INTERSECTION r s)
      (hasheq 'INTERSECTION (list (re->jsexpr r) (re->jsexpr s)))]
    [(COMPLEMENT r)
      (hasheq 'COMPLEMENT (re->jsexpr r))]))
```

```
(define (re->json re)
  (hasheq 're re
    're-json (re->jsexpr (rewrite-re re))
    'string (re->string (rewrite-re re))))
```

```
(define (re-list->jsexpr number-easy number-medium number-hard [re-length MAX-RE-LENGTH])
  (define questions
    (generate-questions number-easy number-medium number-hard re-length))
  (map re->json questions))
```

```
(define (re-as-string re)
  (match re
    [(EMPTY) "(EMPTY)"]
    [(LAMBDA) "(LAMBDA)"]
    [(SYMBOL a) (string-append "(SYMBOL #\"(string a) ")")]
    [(CONCATENATION r s)
      (string-append "(CONCATENATION "(re-as-string r)  (re-as-string s) ")")]
    [(KLEENE-CLOSURE r)
      (string-append "(KLEENE-CLOSURE "(re-as-string r) ")")]
    [(UNION r s)
      (string-append "(UNION "(re-as-string r)  (re-as-string s) ")")]
    [(INTERSECTION r s)
      (string-append "(INTERSECTION "(re-as-string r)  (re-as-string s) ")")]
    [(COMPLEMENT r)
      (string-append "(COMPLEMENT "(re-as-string r) ")")]))
```

Nos Algoritmo 23 e 24 são apresentadas as funções usadas para montar o notebook. As funções `re->cell` e `generate-cells` convertem uma lista de ERs em questões do tipo "Crie

um AFD equivalente a expressão X". A função `generate-metadata` retorna os metadados necessários para a interpretação do notebook. A função `generate-notebook` retorna uma jsexpr contendo todos os dados necessários para a criação de um notebook válido. Por fim, a função `save-notebook` salva um notebook em um arquivo. A Figura 3.15 mostra um exemplo de um notebook gerado.

**Algoritmo 23** Geração do Jupyter Notebook

```
(define (re->cell question num)
   (list (hasheq 'cell_type "markdown"
            'metadata (hasheq)
            'source (list (string-append "#### Questão "(number->string num))))
         (hasheq 'cell_type "markdown"
            'metadata (hasheq)
            'source (list (string-append
               "Crie um AFD equivalente a seguinte expressão regular: "
               (hash-ref question 'string))))
         (hasheq 'cell_type "code"
            'execution_count 0
            'metadata (hasheq 'vscode (hasheq 'languageId "racket"))
            'outputs (list)
            'source (list (string-append
               "(define resposta"
               (number->string num))
               "null"
               ")"))
         (hasheq 'cel_type "code"
            'executio_count 0
            'metadata (hasheq 'vscode (hasheq 'languageId "racket"))
            'outputs (list)
            'source (list (string-append "(automaton-correction resposta"
                     (number->string num)
                     "(re-to-dfa "
                     (re-as-string (hash-ref question 're)) "))")))))
```

```
(define (generate-cells number-easy number-medium number-hard [re-length MAX-RE-LENGTH])
   (define questions-list (re-list->jsexpr
                                          number-easy
                                          number-medium
                                          number-hard
                                          re-length))
   (define require-setup (hasheq 'cell_type "code"
                  'execution_count 0
                  'metadata (hasheq 'vscode (hasheq 'languageId "racket"))
                           'outputs (list)
                           'source (list "(require \"re.rkt\"\"re-to-dfa.rkt\"\"../dfa/automaton-correction.rkt\"\"../dfa/image-builder.rkt\")")))
   (flatten (list require-setup (map re->cell
                              questions-list
                              (build-list (length questions-list) (lambda (x) (add1 x)))))))
```

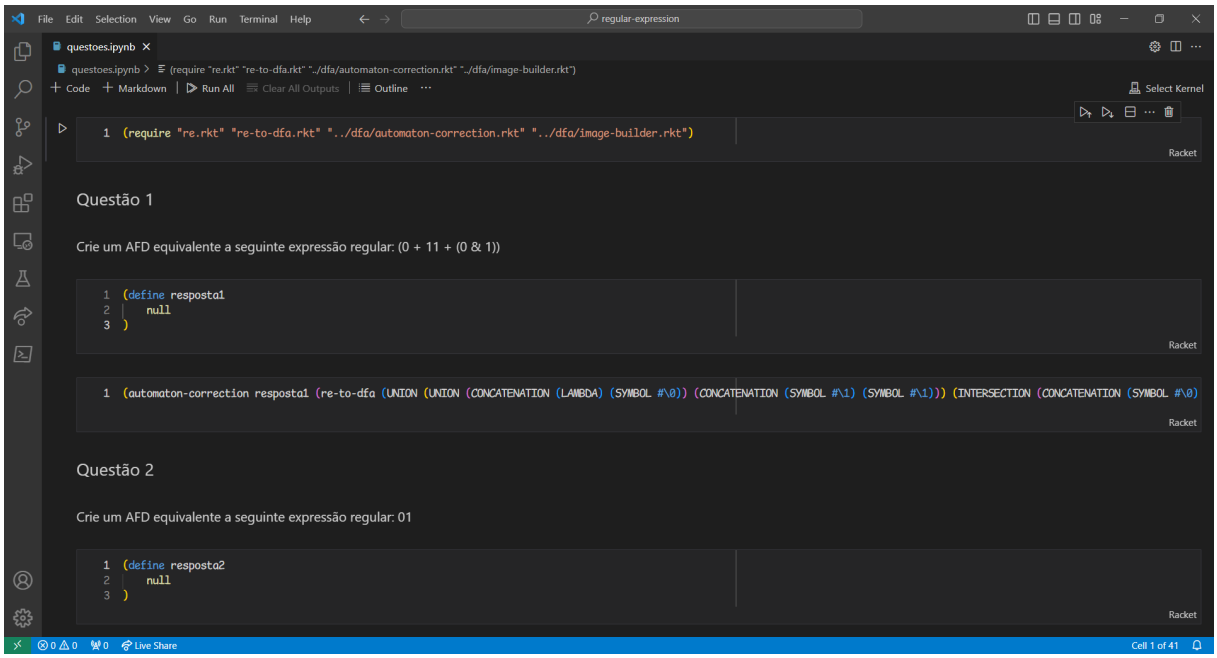
**Algoritmo 24** Geração do Jupyter Notebook - parte 2

```
(define (generate-metadata)
    (hasheq 'kernelspec (hasheq 'display_name "Racket"
                                'language "racket"
                                'name "racket")
            'language_info (hasheq 'codemirror_mode "scheme"
                                   'file_extension ".rkt"
                                   'mimetype "text/x-racket"
                                   'name "Racket"
                                   'pygments_lexer "racket"
                                   'version "8.10")))
```

```
(define (generate-notebook number-easy number-medium number-hard [re-length MAX-RE-LENGTH])
    (hasheq 'cells (generate-cells number-easy number-medium number-hard re-length)
            'metadata (generate-metadata)
            'nbformat 4
            'nbformat_minor 2))
```

```
(define (save-notebook notebook filename)
    (define out (open-output-file filename #:mode 'text #:exists 'truncate))
    (write-json notebook out)
    (close-output-port out))
```

Figura 3.15 – Exemplo de um Jupyter Notebook gerado



## 3.3 Conclusão

Neste capítulo foi apresentado o desenvolvimento da ferramenta para criação de exercícios sobre autômatos finitos determinísticos. O capítulo apresentou uma visão geral do funcionamento da ferramenta proposta e sua implementação utilizando a linguagem Racket.

# 4 Resultados

Neste capítulo são apresentados os testes realizados e seus resultados. A Seção 4.1 descreve os testes realizados, seus resultados e alterações necessárias para melhorar o desempenho da geração de questões e a Seção 4.2 conclui o capítulo.

## 4.1 Testes

Nesta seção são apresentados os testes realizados para avaliar a eficácia do algoritmo genético desenvolvido. O objetivo principal desses testes foi gerar um conjunto diversificado de questões, com a proporção de questões fáceis, média e difíceis escolhidas pelo usuário. Por exemplo, em uma lista de 10 exercícios, o aluno pode querer 3 questões fáceis, 3 médias e 4 difíceis.

A população inicial do algoritmo genético é gerada usando a função `gen:re` (Função 15) e seus indivíduos são as diferentes expressões regulares geradas aleatoriamente. O *crossover* consiste em, dada duas expressões $r$ e $s$, trocar subexpressões delas, gerando duas novas expressões $r'$ e $s'$, enquanto a mutação consiste em trocar uma operação ou um símbolo da expressão.

Para simplificar a geração das questões, o algoritmo genético é executado uma vez para cada questão da lista a ser gerada, variando a dificuldade das questões geradas para manter a proporção escolhida pelo usuário, e o melhor indivíduo da população final é escolhido para entrar na lista.

A escolha do nó onde as operações de *crossover* e mutação acontecem é aleatória e os resultados iniciais podem ser observados na Tabela 4.1. Inicialmente, devido a utilização da função de reescrita de ER, que aninhava as expressões para a direita, um número $n$ entre 1 e a altura da folha mais a direita é gerado e o $n$-ésimo nó no caminho formado pelos galhos mais à direita na árvore é escolhido para ser o ponto.

A Tabela 4.1 apresenta o resultado de uma execução do algoritmo genético para gerar questões de nível fácil. É possível ver que embora o algoritmo tenha gerado 10 ERs, existem apenas 3 expressões distintas. Para tentar aumentar a variedade nas expressões geradas, o *crossover* foi modificado para ser feito em 2 pontos das expressões, como explicado na Seção 3.2.5.

A Tabela 4.2 apresenta os resultados, arredondados, de 100 execuções do algoritmo genético para gerar questões fáceis. Ela apresenta a média aritmética e desvio padrão do número de estados dos AFDs equivalentes às expressões geradas. A geração final é a quantidade de iterações executadas pelo algoritmo genético até sua convergência. A média de estados mostra que as expressões geradas são satisfatórias, atendendo o requisito para questões fáceis. Além disso, o número pequeno de iterações executadas mostra que o problema é relativamente simples

| Identificador | Expressão |
|---|---|
| 1 | $\lambda$ |
| 2 | $\lambda$ |
| 3 | $(1)^*$ |
| 4 | $\lambda$ |
| 5 | $(0 + (11 \,\&\, (1)^*) + \lambda)$ |
| 6 | $\lambda$ |
| 7 | $(0 + (11 \,\&\, (1)^*) + \lambda)$ |
| 8 | $\lambda$ |
| 9 | $(1)^*$ |
| 10 | $\lambda$ |

Tabela 4.1 – Exemplos de expressões de nível fácil geradas com simplificação das expressões e apenas um ponto de *crossover*.

| | Média | Desvio padrão |
|---|---|---|
| Número de estados | 3 | 0.95 |
| Geração final | 4 | 2 |

Tabela 4.2 – Média e desvio padrão do número de estados das expressões e geração em que o algoritmo genético terminou usando as características do gerador na Tabela 4.1.

de resolver, obtendo soluções satisfatórias para o problema. Porém, como dito antes, as expressões geradas apresentam muitas repetições.

A tabela 4.3 mostra algumas expressões de dificuldade fácil geradas por esse método. Ela mostra que, embora a variedade de expressões geradas tenha aumentado, as expressões ainda se repetem.

A Tabela 4.4 apresenta os resultados, arredondados, de 100 execuções do algoritmo genético para gerar questões fáceis, usando o método descrito anteriormente. A média de estados para questões fáceis ficou acima do limite esperado para questões fáceis, gerando mais questões de dificuldade média. Além disso, o número de iterações executadas mostra que dessa forma, o algoritmo precisou executar bem mais iterações e, em alguns casos, alcançou o número máximo de iterações antes de convergir.

Durante os testes, foi observado que a reescrita estava diminuindo o tamanho das expressões, devido às simplificações realizadas. Também foi observado que a escolha de caminhar pelos galhos mais a direita fez com que o início das expressões (o lado esquerdo da árvore) não mudasse ao longo das gerações do algoritmo genético. Esses dois fatores fizeram com que o algoritmo genético convergisse muito rápido e gerasse muitas expressões idênticas ao final da execução.

Para tentar reduzir esse efeito, as expressões não são mais reescritas e o método para escolher o ponto para *crossover* e mutação foi alterado para gerar um caminho na árvore, na forma de uma lista de 0's e 1's (0 representa caminhar pelo galho da esquerda e 1 caminhar pelo

| Identificador | Expressão |
|---|---|
| 1 | $((0 + \lambda))^*$ |
| 2 | $(0 + \lambda)$ |
| 3 | $((0 + ((0 + 00))^*)\ \&\ (1)^*)$ |
| 4 | $((0 + \lambda))^*$ |
| 5 | $\lambda$ |
| 6 | $\neg((0 + 1))((0 + 1 + \lambda))^*$ |
| 7 | $(1)^*$ |
| 8 | $\neg((0 + 1))((0 + 1))^*$ |
| 9 | $\lambda$ |
| 10 | $\lambda$ |

Tabela 4.3 – Exemplos de expressões de nível fácil geradas com simplificação das expressões e dois pontos de *crossover*.

| | Média | Desvio padrão |
|---|---|---|
| Número de estados | 3 | 6 |
| Geração final | 6 | 3 |

Tabela 4.4 – Média e desvio padrão do número de estados das expressões e geração em que o algoritmo genético terminou usando as características do gerador na Tabela 4.3.

galho da direita) e o nó que estivesse no fim do caminho seria o nó escolhido. Isso gerou mais diversidade nas expressões geradas na população final do algoritmo genético, mas frequentemente essas expressões podiam ser simplificadas para outras expressões que acabavam repetindo, i.e., mesmo as expressões sendo diferentes, ainda eram equivalentes. Para tentar aumentar o número de expressões não equivalentes, foram feitos diversos ajustes na frequência das operações e nos símbolos que podem ser gerados durante a criação da população inicial.

A tabela 4.5 apresenta os resultados dessas mudanças. A Tabela 4.6 apresenta os resultados, arredondados, de 100 execuções do algoritmo genético para gerar questões fáceis. A média de estados mostra que as expressões geradas são satisfatórias, atendendo o requisito para questões fáceis. E o número pequeno de iterações executadas mostra que com esse método o algoritmo voltou a convergir mais rapidamente.

## 4.2 Conclusão

Este capítulo abordou a elaboração de algoritmos para a geração de exercícios sobre AFDs, com foco na sua construção e minimização. A implementação bem-sucedida dos algoritmos representou um avanço significativo, proporcionando uma base sólida para a criação de um ambiente de aprendizagem envolvente e prático na área de Teoria da Computação.

Além disso, visando a expansão e integração de nossos esforços, foi feita uma colaboração com outro projeto orientado pelo mesmo professor, para fornecer *feedback* automático para os alunos sobre suas soluções para os exercícios propostos. A integração desses projetos permite a

| Identificador | Expressão |
|---|---|
| 1 | $(0 + ((0 + \lambda)\ \&\ (1 + \lambda)))$ |
| 2 | $(0 + 1)$ |
| 3 | $\neg(\lambda)(1)^*$ |
| 4 | $(1 + \lambda)$ |
| 5 | $(1)^*(1 + \lambda)$ |
| 6 | $\neg(\lambda)(0 + \lambda)$ |
| 7 | $(10 + \lambda)$ |
| 8 | $(0 + (1\ \&\ (1 + \lambda)))\neg(0)$ |
| 9 | $(0 + ((0 + \lambda)\ \&\ (1 + \lambda)))\neg(0)$ |
| 10 | $(0 + ((0 + \lambda)\ \&\ (1)^*))\neg(0)$ |

Tabela 4.5 – Exemplos de expressões de nível fácil geradas sem simplificação das expressões e *crossover* por caminho.

| | Média | Desvio padrão |
|---|---|---|
| Número de estados | 3 | 0.8 |
| Geração final | 4 | 2 |

Tabela 4.6 – Média e desvio padrão do número de estados das expressões e geração em que o algoritmo genético terminou usando as características do gerador na Tabela 4.5.

troca de conhecimentos, a otimização de recursos e a ampliação das possibilidades de aprendizado. Além disso, também foi integrado com o *kernel* IRacket para o Jupyter Notebook, no Visual Studio Code, agregando uma camada adicional de eficiência e facilidade no desenvolvimento e execução dos exercícios.

# 5 Considerações Finais

Durante o desenvolvimento deste trabalho, foi explorada a criação de uma ferramenta dinâmica para auxiliar na compreensão de AFDs. A implementação bem-sucedida de um sistema capaz de gerar expressões regulares aleatórias, com níveis variados de dificuldade, e criar autômatos equivalentes promove uma abordagem prática e interativa para abordar conceitos muitas vezes abstratos e desafiadores. Além disso, a visualização das estruturas por meio de imagens reforça a compreensão visual e abre caminho para treinamento da minimização dos autômatos.

Com a integração ao IRacket e ao trabalho, foi gerado um Jupyter Notebook contendo exercícios de dificuldades variadas, com o total de exercícios de cada dificuldade escolhidos pelo próprio aluno, o que aumentou ainda mais a utilidade e impacto da ferramenta, possibilitando uma adaptação mais personalizada às necessidades de cada estudante e ampliando as possibilidades de interação com o ambiente de aprendizado. Por fim, também é possível criar diferentes tipos de questões, como gerar o AFD equivalente à ER, minimização de AFDs e converter AFN para AFD, entre outros.

Para trabalhos futuros, ficam as sugestões de utilizar a ferramenta em cursos de Teoria da Computação para avaliar seu impacto no aprendizado dos alunos e avaliar outros métodos para definir a dificuldade de uma expressão gerada, além de avaliar outros métodos para geração das expressões da dificuldade desejada.

O código desenvolvido está disponível no repositório *automata-language* <https://github.com/lives-group/automata-language>.

# Referências


ADITHI, G. *et al.* Secure, offline feedback to convey instructor intent. In: IEEE. **2015 IEEE Seventh International Conference on Technology for Education (T4E)**. Warangal, India, 2015. p. 105–108.

ALI, H.; CHALI, Y.; HASAN, S. A. Automatic question generation from sentences. In: **Actes de la 17e conférence sur le Traitement Automatique des Langues Naturelles. Articles courts**. Montréal, Canada: ATALA, 2010. p. 213–218. Disponível em: <https://aclanthology.org/2010.jeptalnrecital-court.36>.

ALSUBAIT, T.; PARSIA, B.; SATTLER, U. Automatic generation of analogy questions for student assessment: an ontology-based approach. **Research in Learning Technology**, v. 20, 2012.

ANTIMIROV, V. Partial derivatives of regular expressions and finite automaton constructions. **Theoretical Computer Science**, Elsevier, v. 155, n. 2, p. 291–319, 1996.

BEZáKOVá, I. *et al.* Prototype of an automated feedback tool for intro cs theory. In: **Proceedings of the 51st ACM Technical Symposium on Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCSE '20), p. 1311. ISBN 9781450367936. Disponível em: <https://doi.org/10.1145/3328778.3372598>.

BRZOZOWSKI, J. A. Derivatives of regular expressions. **Journal of the ACM (JACM)**, ACM New York, NY, USA, v. 11, n. 4, p. 481–494, 1964.

CHAKRABORTY, P.; SAXENA, P. C.; KATTI, C. P. Fifty years of automata simulation: a review. **acm inroads**, ACM New York, NY, USA, v. 2, n. 4, p. 59–70, 2011.

CHUDá, D.; RODINA, D. Automata simulator. In: **Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies**. New York, NY, USA: Association for Computing Machinery, 2010. (CompSysTech '10), p. 394–399. ISBN 9781450302432. Disponível em: <https://doi.org/10.1145/1839379.1839449>.

COSTA, C. S. **Desenvolvimento de algoritmos para correção automática de exercécios sobre autômatos finitos determinísticos.** 2023. Disponível em: <http://www.monografias.ufop.br/handle/35400000/5992>.

GAEBEL, M. *et al.* E-learning in european higher education institutions: Results of a mapping survey conducted in october-december 2013. **European University Association**, ERIC, 2014.

GOLDBACH, I.; LUP, F. H. Survey on e-learning implementation in eastern-europe spotlight on romania. In: **The Ninth International Conference on Mobile, Hybrid, and On-line Learning, eLmL**. Nice, França: [s.n.], 2017. p. 05–13.

MAZIDI, K.; NIELSEN, R. D. Linguistic considerations in automatic question generation. In: **Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)**. Baltimore, Maryland: Association for Computational Linguistics, 2014. p. 321–326. Disponível em: <https://aclanthology.org/P14-2053>.

MOHAMMED, M. K. O. Teaching formal languages through visualizations, simulators, auto-graded exercises, and programmed instruction. In: **Proceedings of the 51st ACM Technical Symposium on Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCSE '20), p. 1429. ISBN 9781450367936. Disponível em: <https://doi.org/10.1145/3328778.3372711>.

OWENS, S.; REPPY, J.; TURON, A. Regular-expression derivatives re-examined. **Journal of Functional Programming**, Cambridge University Press, v. 19, n. 2, p. 173–190, 2009.

QAYYUM, A.; ZAWACKI-RICHTER, O. **Open and distance education in Australia, Europe and the Americas: National perspectives in a digital age**. Singapore: Springer Nature, 2018.

RODGER, S. H. Using hands-on visualizations to teach computer science from beginning courses to advanced courses. In: **Second Program Visualization Workshop**. Hornstrup Center, Dinamarca: [s.n.], 2002. p. 103–112.

SHENOY, V. *et al.* Generating dfa construction problems automatically. In: **2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)**. Mumbai, India: IEEE, 2016. p. 32–37.

SIPSER, M. Introduction to the theory of computation (3rd international ed.). **Cengage Learning**, 2013.

VIEIRA, L. F. M.; VIEIRA, M. A. M.; JOSé, N. Language emulator, uma ferramenta de auxílio no ensino de teoria da computação. Belo Horizonte, MG, Brasil, 01 2003.