
**Universidade do Estado do Rio de Janeiro**
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística
Departamento de Informática e Ciência da Computação

Juan Pedro Alves Lopes

# Uma Implementação de Expressões Regulares em Tempo Polinomial

Rio de Janeiro

2013

Juan Pedro Alves Lopes

# Uma Implementação de Expressões Regulares em Tempo Polinomial

Monografia apresentada como requisito parcial para obtenção do título de Bacharel, ao Instituto de Matemática e Estatística, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Paulo Eustáquio Duarte Pinto

Coorientadora: Prof[a]. Maria Alice Silveira de Brito

Rio de Janeiro

2013

Juan Pedro Alves Lopes

# Uma Implementação de Expressões Regulares em Tempo Polinomial

Monografia apresentada como requisito parcial para obtenção do título de Bacharel, ao Instituto de Matemática e Estatística, da Universidade do Estado do Rio de Janeiro.

Aprovada em 17 de Maio de 2013

Banca Examinadora:

---

Prof. Paulo Eustáquio Duarte Pinto (Orientador)
IME/DICC/UERJ

---

Profª. Maria Alice Silveira de Brito (Coorientadora)
IME/DICC/UERJ

---

titulação membro1
afiliação1

Rio de Janeiro

2013

# Dedicatória

À minha esposa, Jacqueline.

Simplicity is a great virtue but it requires hard work
to achieve it and education to appreciate it. And to
make matters worse: complexity sells better.
– *Edsger W. Dijkstra*

# Resumo

LOPES, Juan Pedro Alves. *Uma Implementação de Expressões Regulares em Tempo Polinomial*. 2013. Monografia (Bacharelado em Informática e Tecnologia da Informação) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro.

Na teoria, expressões regulares constituem uma notação para definir linguagens regulares em termos de operações recursivas simples. São equivalentes a autômatos finitos em seu poder expressivo. Na prática, apesar de serem ferramentas extremamente populares, as implementações modernas afastaram-se bastante da teoria que as originou. A maior parte das mudanças aconteceu para permiti-las alcançar um poder de expressão maior. Entretanto, esta conveniência trouxe o custo de tornar o *match* um problema mais difícil computacionalmente do que a teoria descreve. Na maior parte das linguagens modernas o *match* de *regexes* (como são comumente conhecidas) é um problema NP-difícil. Além disso, por particularidades nas implementações, mesmo expressões que poderiam ser reconhecidas em tempo polinomial são resolvidas com soluções exponenciais.

Neste trabalho são analisadas as armadilhas comuns no desenvolvimento de expressões regulares, realizando benchmarks entre diversas implementações populares, apontando funcionalidades que impediriam sua execução polinomial. É sugerida uma implementação simples e didática que garante tempo de execução polinomial, no pior caso.

Palavras-chave: Teoria da computação. Expressões regulares. Linguagens regulares. Autômatos finitos. Complexidade computacional. Algoritmos. NP-difícil.

# Abstract

In theory, regular expressions are a notation to define regular languages in terms of simple recursive operations. They are equivalent to finite automata in its expressive power. In practice, modern regular expressions implementations deviated from the original theory. Most changes were made to allow them reach a greater expressive power. This convenience, however, came at the cost of making the match a harder problem than it could be. In most modern languages, the regex match is a NP-hard problem. Besides, the way they are implemented makes even expressions that could be recognized in polynomial time to be solved with exponential algorithms.

In this work, we analyze the common pitfalls in regular expressions development, measuring performance with benchmarks of the most popular implementations, pointing features that could not be implemented in polynomial time. Also it suggests a simple and didactic implementation which has superior worst-case performance than most popular implementations.

Keywords: Computation theory. Regular expressions. Regular languages. Finite automata. Computational complexity. Algorithms. NP-hard.

# Lista de figuras


Figura 1 - Exemplo de autômato para expressão regular $(a|a) + b$ . . . . . 11

Figura 2 - Tempo de execução (em escala logarítmica) para *match* de $a^n$ contra $(a?a) + b$ . . . . . 12

Figura 3 - Representação da inclusão de conjuntos na hiearquia de Chomsky . . . 19

Figura 4 - Árvore de avaliação para expressão $ab|cd*$ . . . . . 22

Figura 5 - Diagrama de estados de um autômato não-determinístico . . . . . 24

Figura 6 - Diagrama de estados de um autômato não-determinístico com entrada $\epsilon$ 24

Figura 7 - Árvore de avaliação para expressão $(a|a) + b$ . . . . . 26

Figura 8 - Representação de estados na nova notação . . . . . 26

Figura 9 - Mesmo autômato representado com as duas notações . . . . . 27

Figura 10 - Exemplo completo de autômato para a expressão $(a|a) + b$ . . . . . 28

Figura 11 - Passos do reconhecimento da expressão $(a|a) + b$ contra a entrada $aab$ . 32

Figura 12 - Representação do autômato como diagrama de estados . . . . . 36

Figura 13 - Autômato para expressão $a + b + c+$ . . . . . 39

Figura 14 - Exemplo de visualização lendo o 4º caractere da string . . . . . 39

Figura 15 - Exemplo de visualização lendo o 7º caractere da string . . . . . 40

Figura 16 - Diagrama de conjuntos caso $P \neq NP$ e $P = NP$, respectivamente . . . 43

Figura 17 - Tempo de execução (em escala logarítmica) para match de $a^n$ contra $(a?a) + b$ . . . . . 47

Figura 18 - Tempo de execução (em escala logarítmica) para match de $a^n$ contra $a * b$ 48

Figura 19 - Tempo de execução (em escala logarítmica) para match de $a^n$ contra $a * a * a * a * a * b$ . . . . . 49

# Lista de listagens


1 Exemplo de uso da biblioteca PyRex . . . . . 34

2 Implementação da regra *sequence* . . . . . 35

3 Exemplo de tratamento de erros no parser . . . . . 35

4 Representação do autômato como array . . . . . 36

5 Representação do autômato como instruções . . . . . 36

6 Implementação do método *match* . . . . . 37

7 Implementação do método *addnext* . . . . . 38

8 Implementação do método *matcher* . . . . . 38

9 Implementação de 3SAT com expressões regulares . . . . . 45

10 Código-fonte da biblioteca PyRex . . . . . 53

11 Código-fonte do Visualizador . . . . . 55

12 Benchmark 1: (a?a)+b . . . . . 56

13 Benchmark 2: a*b . . . . . 56

14 Benchmark 3: a*a*a*a*a*b . . . . . 56

15 Benchmark runner . . . . . 57

16 Benchmark com PyRex . . . . . 57

17 Benchmark com Python . . . . . 58

18 Benchmark com Ruby . . . . . 58

19 Benchmark com Java (compilador) . . . . . 58

20 Benchmark com Java (fonte) . . . . . 58

# Sumário


**1 Introdução** . . . . . 11

1.1 Motivação . . . . . 11

1.2 Objetivo . . . . . 13

1.3 Estrutura . . . . . 14

**2 Fundamentação Teórica** . . . . . 15

2.1 Linguagens Formais . . . . . 15

2.1.1 *Palavra* . . . . . 15

2.1.2 *Linguagem* . . . . . 16

2.2 Gramáticas Gerativas . . . . . 17

2.3 Hierarquia de Chomsky . . . . . 18

2.3.1 *Gramáticas Regulares* . . . . . 19

2.3.2 *Gramáticas Livres de Contexto* . . . . . 20

2.3.3 *Gramáticas Sensíveis ao Contexto* . . . . . 20

2.3.4 *Gramáticas Recursivamente Enumeráveis* . . . . . 20

2.3.5 *Tabela de Referência* . . . . . 21

2.4 Expressões Regulares . . . . . 21

2.5 Autômatos Finitos . . . . . 23

2.6 Método de Construção de Thompson . . . . . 25

2.6.1 *Analise sintática* . . . . . 25

2.6.2 *Notação* . . . . . 26

2.6.3 *Construção* . . . . . 27

2.7 Simulação do Autômato . . . . . 29

2.7.1 *Backtracking* . . . . . 29

2.7.2 *Backtracking com Memorização* . . . . . 30

2.7.3 *Conversão para DFA* . . . . . 30

2.7.4 *Simulação paralela* . . . 31
2.7.5 *Conversão preguiçosa para DFA* . . . 31
**3 Implementação** . . . 33
3.1 Funcionalidades Implementadas . . . 33
3.2 Estrutura do Código . . . 34
3.3 Análise sintática . . . 35
3.4 Representação . . . 36
3.5 Simulação . . . 37
3.6 Visualização . . . 39
**4 Análise dos Resultados** . . . 41
4.1 Retorno do Método *match* . . . 41
4.2 Reconhecimento de *backreferences*: um problema NP-difícil . . . 42
4.2.1 *Problemas NP-difíceis* . . . 42
4.2.2 *Reconhecimento de* backreferences . . . 44
4.2.3 *Implementação de* 3SAT *com expressões regulares* . . . 45
4.3 Benchmarks . . . 46
4.3.1 *Benchmark 1: (a?a)+b* . . . 46
4.3.2 *Benchmark 2: a*b* . . . 47
4.3.3 *Benchmark 3: a*a*a*a*a*b* . . . 48
**5 Conclusão** . . . 50
5.1 Contribuição . . . 50
5.2 Funcionalidades não implementadas . . . 50
5.3 Trabalhos Futuros . . . 51
**Apêndice** A – Código-fonte da Biblioteca PyRex . . . 53
**Apêndice** B – Código-fonte do Visualizador . . . 55
**Apêndice** C – Código-fonte dos Benchmarks . . . 56
**Apêndice** D – Tabelas dos Benchmarks . . . 59
**Referências** . . . 62

# Capítulo 1
# Introdução

Neste capítulo serão apresentados a motivação, objetivos, e a estrutura do projeto.

## 1.1 Motivação

Expressões regulares fazem parte do ferramental da maioria das linguagens e plataformas de desenvolvimento modernas. Seu uso é bastante difundido na indústria para os mais variados fins, desde o reconhecimento de padrões, passando pela extração de símbolos para análise sintática de linguagens formais, até a sanitização de entradas do usuário para fins de segurança.

Seu extensivo uso vem precedido por uma forte base teórica na área de autômatos finitos, introduzida nos anos 40 por McCulloch e Pitts (MCCULLOCH; PITTS, 1943) e formalizada na década seguinte por Kleene (KLEENE, 1956).

Expressões regulares e autômatos finitos compartilham uma relação estreita. São equivalentes em poder expressivo, e a conversão do primeiro para o segundo é trivial, como pode ser visto na Figura 1. Ken Thompson demonstrou isso em sua implementação no final dos anos 60 (THOMPSON, 1968), enquanto desenvolvia o editor de texto *QED*. As mesmas expressões regulares foram, posteriormente, portadas para os mais conhecidos *ed* e *grep*, integrantes do sistema operacional Unix.

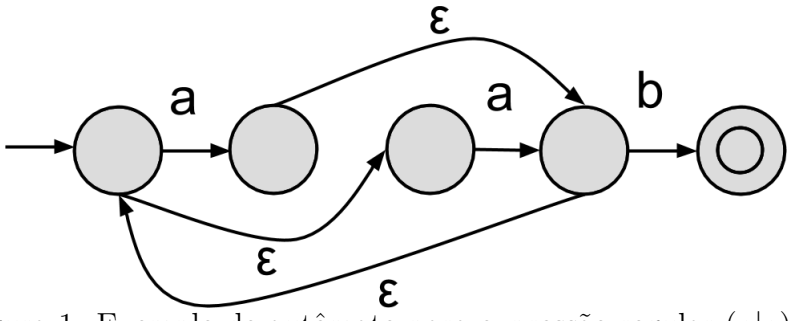



Figura 1: Exemplo de autômato para expressão regular $(a|a) + b$

Desde então, conforme as implementações foram evoluindo, muitas funcionalidades foram adicionadas à linguagem de descrição de expressões regulares que as afastaram da teoria original. Enquanto originalmente descreviam linguagens estritamente regulares, a implementação mais difundida atualmente (*PCRE*) é capaz não só de reconhecer qualquer

linguagem livre de contexto, como também algumas sensíveis ao contexto. (POPOV, 2012)

Uma das consequências mais notáveis desta evolução não planejada é que o reconhecimento de strings da linguagem, um problema com solução linear originalmente, ganhou soluções exponenciais em um grande número de implementações modernas, que inclui muitas das mais usadas. Talvez a funcionalidade mais perigosa neste sentido sejam as *backreferences*, que não só impedem soluções polinomiais como tornam o problema de reconhecimento NP-difícil. Entretanto, mesmo nas expressões que poderiam ser reconhecidas estritamente com autômatos finitos, certas particularidades de implementação as tornam potencialmente exponenciais, no pior caso (COX, 2007).

Linguagens como Python, Java e Ruby são amplamente utilizadas pelo mercado, porém suas bibliotecas-padrão implementam expressões regulares vulneráveis. A Figura 2 mostra o crescimento no tempo de execução entre suas implementações e a implementação proposta neste projeto (*pyrex*).

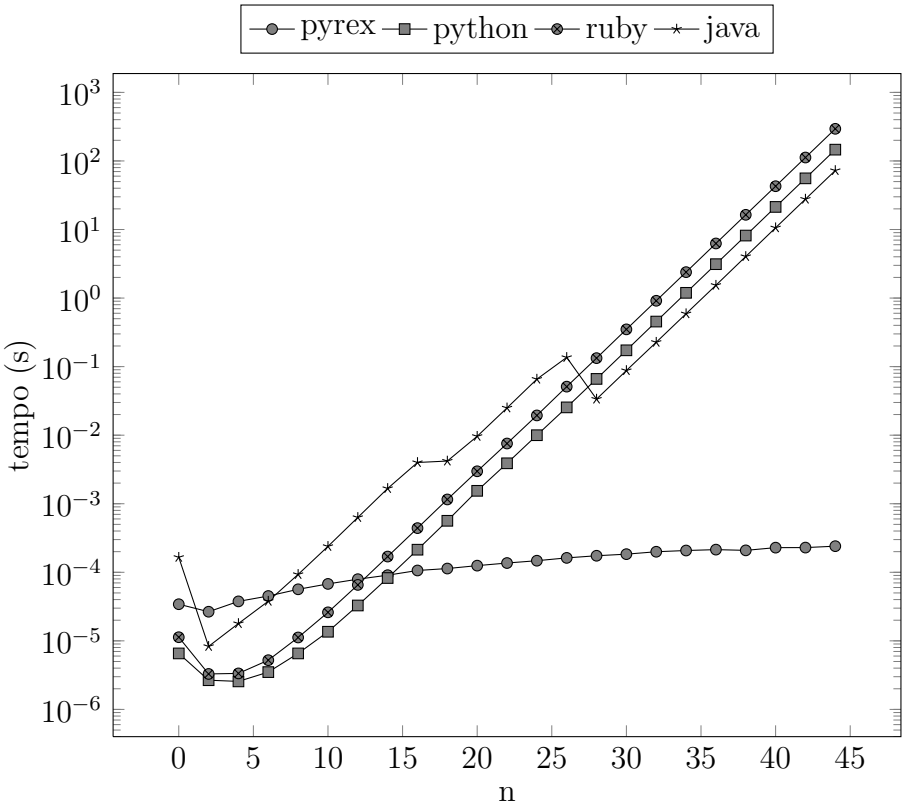



Figura 2: Tempo de execução (em escala logarítmica) para *match* de $a^n$ contra $(a?a) + b$

Estas implementações tornam o uso de expressões regulares potencialmente inseguro em situações ora triviais. Um usuário mal intencionado pode ser capaz de forçar a execução de uma expressão com caráter exponencial para efetuar um ataque de negação de serviço em um servidor web.

Por exemplo, ao permitir que entradas do usuário sejam interpretadas como expressões regulares, um usuário mal intencionado pode injetar a expressão $(a?a) + b$, que por sua complexidade exponencial em implementações vulneráveis, ocuparia 100% de uma CPU por um tempo virtualmente infinito. Vários ataques deste tipo podem causar instabilidade e queda do serviço web.

Muitas vezes a prevenção para esse tipo de ataque pode não ser trivial, e.g. em Java, onde métodos comumente usados sobre objetos oriundos da entrada do usuário, como *replaceAll* e *split*, da classe *String* são implementados com expressões regulares vulneráveis a esse tipo de ataque.

Mesmo em casos onde o usuário mal intencionado não tem acesso à escrita de expressões regulares, um erro de programação pode deixar o sistema vulnerável a negação de serviço. Este problema torna-se especialmente notável pelo frequente compartilhamento em repositórios online, que contêm muitas expressões vulneráveis (KIRRAGE; RATHNAYAKE; THIELECKE, 2013; WEIDMAN, 2010).

## 1.2 Objetivo

Este projeto tem dois objetivos principais. O primeiro é demonstrar através de testes pontuais e benchmarks os problemas fundamentais nas implementações de expressões regulares em diversas linguagens modernas, provando inclusive que o problema de reconhecimento de expressões regulares com *backreferences* pertence à classe de problemas NP-difícil. O segundo objetivo é alcançar uma implementação didática e minimalista das expressões regulares, abordando as propostas por Kleene e utilizando o método descrito por Thompson para construção e simulação do autômato.

A implementação neste projeto não inclui certas funcionalidades comuns nos *sabores* mais modernos de expressão regular. Algumas destas funcionalidades podem ser implementadas sem sacrificar eficiência de execução. Outras, introduzem certa complexidade, porém mantêm a execução do algoritmo polinomial. O restante, entretanto, não é possível implementar sem tornar o algoritmo exponencial no pior caso. Todas as funcionalidades intencionalmente excluídas serão listadas e propostas de soluções serão apresentadas quando cabível.

Deseja-se mostrar com esse projeto que o problema de reconhecimento de expressões regulares pode ser resolvido de maneira eficiente, mesmo com as implementações mais simples, desde que seja observada a base teórica que as deu origem.

## 1.3 Estrutura

O projeto está divido em cinco capítulos. Cada um dos próximos capítulos está descrito abaixo.

**Capitulo 2 – Fundamentação Teórica:** base teórica por trás das expressões regulares, contando sua história, passando pelo método de construção do autômato de Thompson, e exibindo um algoritmo trivial (porém ineficiente) para sua solução.

**Capitulo 3 – Implementação:** implementação apresentada neste trabalho, discutindo certos aspectos específicos desta implementação que não tenham sido abordados na parte de teoria.

**Capitulo 4 – Análise dos Resultados:** diferenças entre as implementações modernas e a implementação proposta neste trabalho. Neste capítulo é provada a pertinência em NP-difícil do *match* de expressões com *backreferences* e são mostrados benchmarks comparando as diversas implementações (inclusive a aqui apresentada), que evidenciam uma diferença fundamental na forma como o problema é abordado em cada uma delas.

**Capitulo 5 – Conclusão:** descrição das contribuições deste trabalho, bem como as dificuldades encontradas durante o projeto. Também serão listadas e descritas as funcionalidades não implementadas, possivelmente sugerindo formas eficientes de implementá-las, ou mesmo formas diferentes de simular o autômato que melhorem a eficiência geral da execução.

# Capítulo 2
# Fundamentação Teórica

Neste capítulo será introduzida um pouco da base teórica que guiou a implementação deste projeto.

## 2.1 Linguagens Formais

Linguagens formais é um tema que mistura elementos da ciência da computação, matemática e linguística. Seu papel na computação é fundamental pela relação de seus problemas com os descritos pela teoria da computação.

Uma *linguagem formal* é um conjunto – potencialmente infinito – de *palavras*, que por sua vez são combinações de elementos de um conjunto finito de símbolos chamado *alfabeto*.

### 2.1.1 *Palavra*

Palavras são combinações entre os elementos de um alfabeto. Por exemplo, com um alfabeto $\Sigma = \{0, 1\}$, é possível formar palavras que representam números binários.

Uma palavra é representada pelos símbolos que a compõem. Logo, 01101 é uma palavra formada pelos símbolos do alfabeto descrito anteriormente. Uma palavra especial, denotada $\epsilon$ (às vezes $\lambda$) representa uma palavra vazia.

Dado um alfabeto com $k$ símbolos, é possível formar até $k^n$ palavras de tamanho $n$. Logo, existem

$$\sum_{i=0}^{n} k^i = \frac{k^{n+1} - 1}{k - 1}$$

palavras de até $n$ caracteres num alfabeto com $k$ símbolos.

O conjunto formado por todas as palavras de um alfabeto é infinito, porém enumerável. Uma possível estratégia é numerá-los primeiro por seu tamanho, então por sua ordem lexicográfica.

A operação básica entre palavras é a *concatenação*. Por exemplo,

$$w_1 = aaa, w_2 = bbb \Rightarrow w_1 w_2 = aaabbb$$
$$w_1 = aaa, w_2 = \epsilon \Rightarrow w_1 w_2 = aaa$$
$$w_1 = \epsilon, w_2 = bbb \Rightarrow w_1 w_2 = bbb$$

Concatenação é associativa, então

$$(w_1 w_2) w_3 = w_1 (w_2 w_3),$$

porém não comutativa, logo

$$w_1 w_2 \neq w_2 w_1.$$

### 2.1.2 *Linguagem*

Linguagens são conjuntos de palavras sobre algum alfabeto. A mesma notação de conjuntos é utilizada para elas. Por exemplo, $A \cup B$ representa a linguagem formada pela união de todas as palavras de A e B, assim como $w \in A$ é uma proposição que afirma que a palavra $w$ está presente na linguagem A.

É possível concatenar linguagens. Dadas linguagens $L_1$ e $L_2$,

$$L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$$

E.g.: se $A = \{a, aa\}$ e $B = \{b, bb\}$, então $AB = \{ab, aab, abb, aabb\}$.

Com base na concatenação, uma operação importante em linguagens é o fecho Kleene (ou fecho de concatenação, $L^*$), que pode ser definido recursivamente como:

$$L^0 = \{\epsilon\}$$
$$L^i = L^{i-1} L$$
$$L^* = \bigcup_{i \in \mathbb{N}} L^i$$

E.g.: se $A = \{a, aa\}$, então $A^* = \{\epsilon, a, aa, aaa, \ldots\}$.

Existem duas formas principais para se definir linguagens formais. *Gramáticas gerativas*, que geram palavras da linguagem a partir de sistemas de reescrita; e *Autômatos*, que reconhecem palavras da linguagem.

Além destas, existem diversas outras formas de definir linguagens. As próprias expressões regulares denotam linguagens formais. Outra forma bastante é comum é através da sintaxe de geração de conjuntos, e.g. $L = \{a^n \mid n \in \mathbb{N}\}$.

## 2.2 Gramáticas Gerativas

Uma das formas de representar linguagens formais é através de gramáticas gerativas. O termo "gramática formal" é comumente utilizado para referir-se a elas. No entanto, existem outras formas de representar gramáticas formais além deste.

Gramáticas gerativas são sistemas de reescrita, i.e. a partir de um símbolo inicial, é possível usar as regras de reescrita para produzir todas as palavras de uma linguagem. Esta operação é, de certa forma, o problema dual de reconhecer uma palavra como parte da linguagem através de um autômato (RUOHONEN, 2009).

Uma gramática gerativa é formada por uma tupla $G = (N, \Sigma, P, S)$. Onde:

- N é o conjunto de símbolos não terminais. Eles não fazem parte da linguagem original, então precisam ser sempre substituídos para alcançar uma palavra válida na linguagem.
- $\Sigma$ é o conjunto de símbolos terminais da linguagem. Também conhecido como alfabeto.
- P é o conjunto de regras de produção, geralmente na forma $\alpha \rightarrow \beta \mid \alpha, \beta \in (N \cup \Sigma)*$.
- S é o símbolo inicial do sistema de reescrita. $S \in N$.

Um exemplo de gramática é a que define expressões da forma $a^n b^n, n > 0$:

$$G = (\{S\}, \{a, b\}, P, S)$$

Onde P é definido por:

$$S \rightarrow aSb$$

$$S \rightarrow ab$$

Iniciando do símbolo S, é possível começar com a regra $ab$ ou $aSb$. A primeira já forma uma palavra na linguagem que queremos descrever ($ab$), a segunda possui um símbolo não terminal. A partir dele é possível gerar as palavras na linguagem: $aabb$, $aaabbb$, etc...

Perceba que na gramática todas as regras do lado esquerdo possuem apenas um símbolo não terminal. Se fôssemos escrever uma gramática para expressões na forma $a^n b^n c^n$, precisaríamos de uma gramática mais rebuscada:

$$G = (\{S, T\}, \{a, b, c\}, P, S)$$

Onde P é definido por:

$$S \rightarrow aTSc$$

$$S \rightarrow abc$$

$$Ta \rightarrow aT$$

$$Tb \rightarrow bb$$

O fato desta gramática definir sequências de símbolos à esquerda com mais de um símbolo a torna especial. Segundo a hiearquia de Chomsky, a primeira gramática é livre de contexto, enquanto a segunda é sensível ao contexto.

## 2.3 Hierarquia de Chomsky

Segundo Noam Chomsky (CHOMSKY, 1957), uma gramática formal pode ser classificada em um dos quatro tipos baseados em sua capacidade expressiva:

**Tipo 0:** gramáticas recursivamente enumeráveis (ou irrestritas)

**Tipo 1:** gramáticas sensíveis ao contexto

**Tipo 2:** gramáticas livres de contexto

**Tipo 3:** gramáticas regulares

Como pode ser visto no diagrama da Figura 3, cada conjunto é propriamente incluso no seu conjunto superior na hierarquia. Há entretanto uma ligeira exceção no caso das

gramáticas sensíveis ao contexto, pois estas não podem gerar a string $\epsilon$, enquanto as livres de contexto podem. Gramáticas livres de contexto que geram $\epsilon$ constituem a única exceção da pertinência às gramáticas sesíveis ao contexto.

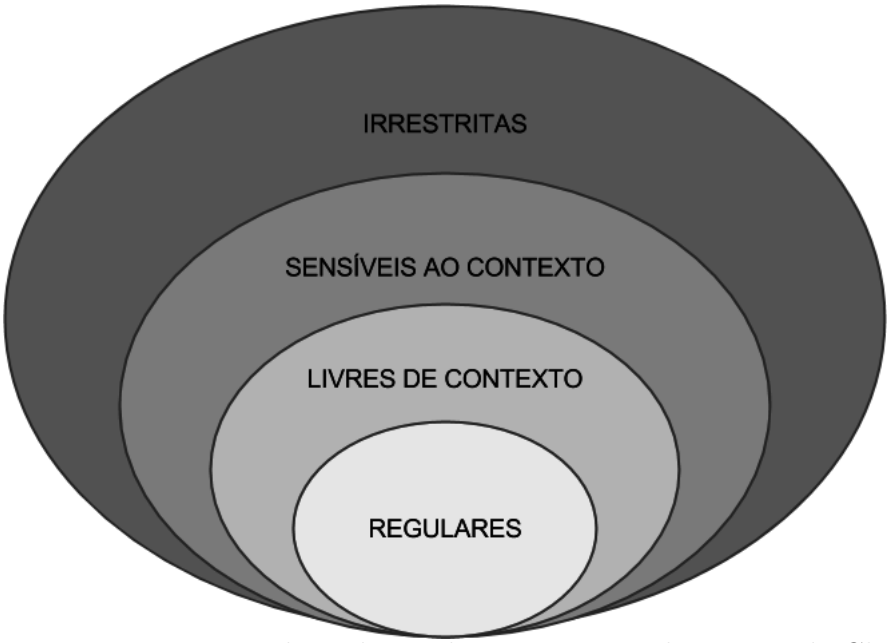



Figura 3: Representação da inclusão de conjuntos na hiearquia de Chomsky

### 2.3.1 *Gramáticas Regulares*

Gramáticas regulares são aquelas cujas regras de produção são na forma $S \to a$ ou $S \to aT$, onde $S$ e $T$ são não-terminais e $a$ é um terminal na linguagem. Estas gramáticas são equivalentes em poder de expressão a *autômatos finitos*.

Opcionalmente a regra $S \to aT$ pode ser substituída por $S \to Ta$, mas nunca as duas podem ocorrer na mesma gramática. Se a primeira ocorre, a gramática é conhecida como *regular à direita*. Caso contrário, é *regular à esquerda*.

Alguns autores também incluem regras do tipo $S \to \epsilon$, desde que S não apareça do lado direito de nenhuma regra.

Exemplos de linguagens regulares incluem: endereços de email, repetição de caracteres ($a^n$), repetição independente de dois caracteres ($a^n b^m$).

### 2.3.2 *Gramáticas Livres de Contexto*

Gramáticas livres de contexto são aquelas onde todas as regras de produção são do tipo $S \rightarrow \gamma$, onde $S$ é um não-terminal e $\gamma$ é uma sequência composta por terminais e não-terminais.

Isto na prática significa que a substituição de um não-terminal é sempre feita independente do contexto onde é encontrado. Gramáticas deste tipo são equivalentes em poder expressivo a *autômatos de pilha*. Elas compõem a base teórica para a maioria das linguagens de programação.

Entre alguns exemplos clássicos de linguagens livres de contexto figuram: parênteses aninhados, repetição condicionada de dois caracteres ($a^n b^n$), expressões aritméticas.

### 2.3.3 *Gramáticas Sensíveis ao Contexto*

Gramáticas sensíveis ao contexto são aquelas onde todas as regras de produção são do tipo $\alpha S \beta \rightarrow \alpha \gamma \beta$, onde $\alpha$, $\gamma$ e $\beta$ são sequências compostas por terminais e não-terminais. Elas diferem das gramáticas livres de contexto por definir – exatamente – contexto para que um não-terminal S seja substituído por uma sequência $\gamma$.

Apesar de terem um escopo bem mais amplo, gramáticas deste tipo ainda são passíveis de ser computadas usando um *autômato linearmente limitado*.

Exemplos de liguagens sensíveis ao contexto: *squares* (*DogDog, CatCat, WikiWiki...*), *matching triplets* ($a^n b^n c^n$).

### 2.3.4 *Gramáticas Recursivamente Enumeráveis*

As gramáticas para as linguagens recursivamente enumeráveis não apresentam restrições em suas regras, o que faz com que essas linguagens sejam também chamadas de "sem restrição". Portanto, qualquer sequência de terminais e não-terminais $\alpha$ pode ser substituída por outra $\beta$. Gramáticas assim são reconhecíveis por uma *máquina de Turing*.

Diferente dos tipos anteriores, quando apresentada a uma sequência que não faça parte da linguagem, não há garantia de parada da máquina, definindo o que é chamado de *conjunto semi-decidível*.

Um dos exemplos mais notáveis de linguagens recursivamente enumeráveis que não

são sensíveis ao contexto é a sintaxe de templates do C++ (VELDHUIZEN, 2003).

### 2.3.5 *Tabela de Referência*

É possível resumir os tipos de gramáticas da hierarquia de Chomsky numa tabela de referência quanto à máquina que as reconhece e os tipos de regras da gramática que as gera.

| | **Linguagem** | **Máquina** | **Regras** |
|---|---|---|---|
| 0 | Recursivamente enumerável | Máquina de Turing | $\alpha \rightarrow \beta$ |
| 1 | Sensível ao contexto | Autômato limitado linearmente | $\alpha S \beta \rightarrow \alpha \gamma \beta$ |
| 2 | Livre de contexto | Autômato de pilha | $S \rightarrow \gamma$ |
| 3 | Regular | Autômato finito | $S \rightarrow aT$ ou $S \rightarrow a$ |

## 2.4 Expressões Regulares

Expressões regulares são sequências de caracteres que denotam linguagens regulares. Estas sequências obedecem a algumas regras de criação. Alguns caracteres são interpretados literalmente, enquanto outros são considerados metacaracteres que controlam como os trechos da expressão se relacionam.

Por definição:

- Sequências vazias são expressões regulares.
- Caracteres unitários são expressões regulares. E.g. $a$.
- Expressões regulares podem ser concatenadas para formar novas expressões. E.g. se $e_1$ reconhece $a$ e $e_2$ reconhece $b$, então $e_1 e_2$ reconhece $ab$.
- Expressões regulares podem ser alternadas para formar novas expressões. E.g. se $e_1$ reconhece $a$ e $e_2$ reconhece $b$, então $e_1|e_2$ reconhece $a$ e $b$.
- Pode-se aplicar o fecho Kleene para formar novas expressões. E.g. se $e_1$ reconhece $a$, então $e_1*$ reconhece $a$, $aa$, $aaa$... etc., bem como string vazia.

Perceba que é possível definir outros operadores comuns, como ? e +. Mas é possível fazê-lo a partir das operações descritas acima. O operador $e_1?$ pode ser definido como a alternância entre $e_1$ e uma string vazia. O operarador $e_1+$ pode ser definido como $e_1 e_1*$.

Outros operadores sobre caracteres como o . (qualquer caractere) e as classes de caractere (e.g. $[a-zA-Z]$) são trivialmente implementados usando alternação de expressões.

A precedência natural dos operadores, da mais fraca para a mais forte, é: alternância, concatenação, repetição. Assim, uma expressão como $ab|cd*$ é interpretada como exibido na Figura 4.

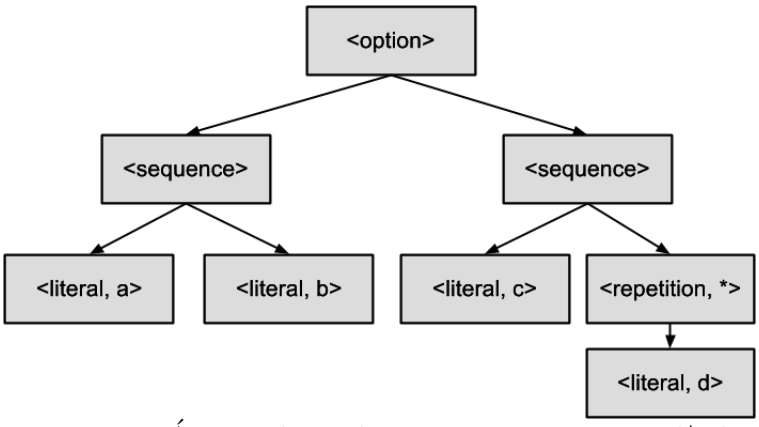



Figura 4: Árvore de avaliação para expressão $ab|cd*$

É possível agrupar sub-expressões utilizando parênteses, tal qual é feito com expressões aritméticas. Assim $a(b|c)d*$ tem um sentido completamente diferente de $ab|cd*$.

Estas operações definem inequivocamente expressões regulares. É possível representar qualquer linguagem regular utilizando apenas estes operadores. Mais importante: é teoricamente possível reconhecer qualquer símbolo desta linguagem com apenas uma passagem pela entrada (sem *backtracking*).

Representando apenas linguagens regulares, as expressões regulares conduzem a um paradoxo sobre si mesmas: elas não são capazes de representar a própria linguagem que as define. A razão disso é a presença dos parênteses agrupadores, tornando esta gramática propriamente pertencente ao conjunto das gramáticas livres de contexto.

Novas implementações de expressões regulares adicionaram novos operadores que tornam a sintaxe mais concisa. Porém, essas adições normalmente não alteram o poder expressivo das *regexes*, sendo encontradas adições que aumentam o poder expressivo, como é o caso das *backreferences*, que permitem o reconhecimento de algumas linguagens não-regulares. *Backreferences* consistem em referências *submatches* reconhecidas anteriormente.

Pode ser observado o exemplo de uma linguagem composta por palavras repetidas, como *DogDog* ou *CatCat*, para o qual é possível definir uma expressão regular $/(.*)\backslash 1/$ que reconhece esta linguagem. $\backslash 1$ instrui o motor de reconhecimento a esperar uma nova palavra exatamente igual à obtida pelo reconhecimento do primeiro grupo $(.*)$.

Esta linguagem não apenas fica fora do conjunto das linguagens regulares, como também não é uma linguagem livre de contexto.

Tal poder expressivo tem seu preço. O reconhecimento de *backreferences* é um problema NP-difícil, logo, os melhores algoritmos conhecidos ainda têm pior caso exponencial. É possível, porém, escrever algoritmos polinomiais para expressões que não fazem uso de *backreferences*.

Para suportar *backreferences*, a implementação mais comum consiste em *backtracking* com memória das capturas anteriores, o que impede a utilização de *memorização*, técnica que poderia tornar polinomial a complexidade deste tipo de implementação. Mais detalhes sobre os tipos de implementação são discutidos na Seção 2.7, na Página 29.

Uma outra solução seria utilizar duas implementações, uma garantidamente polinomial (para casos simples) e outra possivelmente exponencial (para casos mais complexos), porém este tipo de solução exige grande esforço de engenharia para manter concordantes os resultados das duas implementações.

## 2.5 Autômatos Finitos

Autômatos finitos representam um modelo matemático de computação. Eles representam máquinas abstratas constituídas de um número finito de estados. Formalmente, autômatos finitos são uma tupla $M = (Q, \Sigma, q_0, \delta, A)$, onde:

- Q é o conjunto de estados do autômato. Sempre finito.
- $\Sigma$ é o alfabeto da linguagem reconhecida pelo autômato.
- $q_0$ é o estado inicial do autômato. $q_0 \in Q$.
- $\delta$ é a função de transição, que mapeia cada $(q_i, a) \in Q \times \Sigma$ a um subconjunto de Q. Se esse subconjunto tem apenas zero ou um elementos, o autômato é *determinístico*, caso contrário, ele é *não-determinístico*. Cabe notar que para cada autômato não-determinístico, há um autômato determinístico equivalente (RABIN; SCOTT, 1959).
- A é o conjunto de estados de aceite. $A \subseteq Q$.

Por exemplo:

O autômato na Figura 5 pode ser descrito por $M = (\{1, 2, 3, 4\}, \{a, b, c\}, 1, \delta, \{4\})$, onde $\delta$ é definida pela tabela de transições:

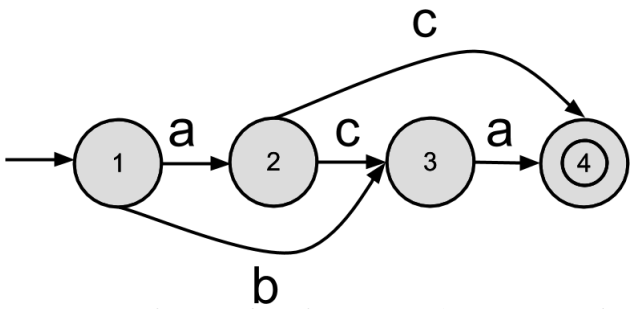



Figura 5: Diagrama de estados de um autômato não-determinístico

| $\delta$ | **a** | **b** | **c** |
|---|---|---|---|
| 1 | 2 | 3 | |
| 2 | | | 3, 4 |
| 3 | 4 | | |
| 4 | | | |

Este autômato é *não determinístico*, pois se a máquina reconhecedora estiver no estado 2 e for lido o símbolo $c$, esse par de parâmetros (estado, entrada) causam na função de transição um resultado com dois estados alternativos, o 3 ou o 4, caracterizando um indeterminismo.

Autômatos finitos não-determinísticos também aceitam um tipo de entrada especial $\epsilon$, que não consome caracteres da entrada, como pode ser visto na Figura 6. Perceba que este autômato *não é equivalente* ao definido anteriormente. Ele aceita uma palavra $b$ que o primeiro não aceita.

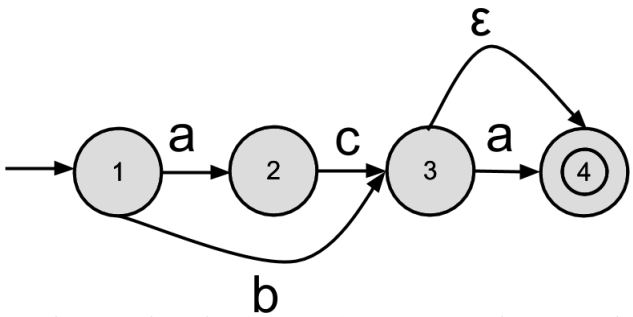



Figura 6: Diagrama de estados de um autômato não-determinístico com entrada $\epsilon$

Autômatos finitos podem ser utilizados para reconhecer linguagens regulares. É possível construir um autômato que, dada uma linguagem, termina em estados de aceite somente se a palavra fizer parte da linguagem representada.

O autômato descrito na Figura 6, por exemplo, reconhece uma linguagem finita definida por $L = \{ac, aca, b, ba\}$. A linguagem que um autômato finito reconhece é *sempre regular*. E *toda linguagem regular* pode ser reconhecida com autômatos finitos (KLEENE,

1956).

## 2.6 Método de Construção de Thompson

Pela sinergia entre expressões regulares e autômatos finitos e pela facilidade de lidar com cada um deles em situações específicas, é útil poder converter entre um e outro. Ken Thompson (THOMPSON, 1968) descreveu um método para construir autômatos a partir de expressões regulares.

Na época em que foi escrito, o artigo descrevia como gerar código de máquina para um IBM 7094. Isto era necessário por restrições de hardware da época. Hoje não é mais estritamente necessário. Este trabalho irá se limitar a descrever como construir o autômato a partir da expressão regular.

O método consite em construir o autômato recursivamente através da análise da árvore sintática da expressão, procedimento que precisa ser efetuado antes do começo da construção do autômato. No artigo original de Thompson, este passo era feito construindo a expressão em notação *infixa*. Entretanto, para permitir extensibilidade do formato da expressão, iremos definir completamente a gramática da expressão regular que aceitaremos.

### 2.6.1 *Analise sintática*

O primeiro passo para a construção do autômato é analisar sintaticamente a expressão, construindo a árvore sintática.

Podemos definir uma gramática gerativa simples pela notação EBNF (*Extended Backus–Naur Form*) que representa a linguagem formada pelas expressões regulares.

```
<option>     ::= <sequence> { "|" <sequence> }*
<sequence>   ::= <repetition> { <repetition> }*
<repetition> ::= <primary> { "*" | "?" | "+" }*
<primary>    ::= "." | <literal> | "(" <option> ")"
```

Assim, a expressão pode ser facilmente analisada. Por exemplo, a expressão $(a|a) + b$ seria avaliada como a árvore descrita na Figura 7.

Utilizando a EBNF, esta árvore pode ser produzida por diversos tipos de ferramentas geradoras de compiladores presentes em várias linguagens. Exemplos notáveis incluem

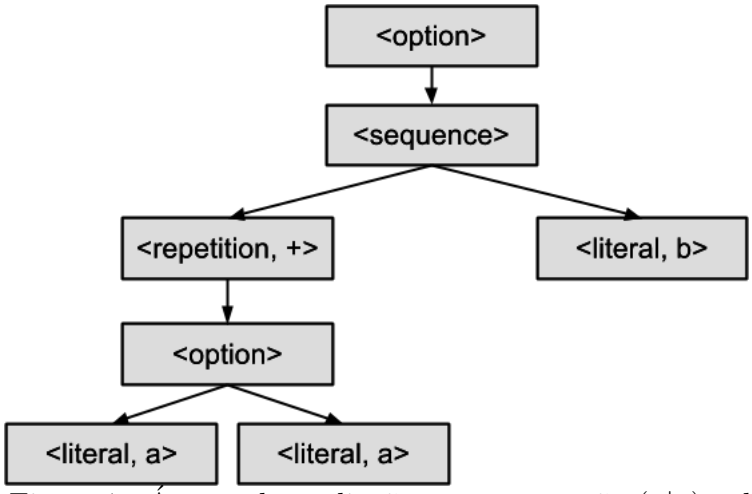


Figura 7: Árvore de avaliação para expressão $(a|a) + b$

o YACC (*Yet Another Compiler Compiler*), o JAVACC (*Java Compiler Compiler*) e o ANTLR (*ANother Tool for Language Recognition*).

Com esta árvore de avaliação é possível fazer a conversão para o autômato de forma bastante trivial, como será discutido nas próximas seções.

### 2.6.2 *Notação*

1. Estado de *consumo*, com apenas uma transição ($a \in \Sigma, S \in N$).
2. Estado de *decisão*, com uma transição $\epsilon$ para um estado ou duas transições $\epsilon$ para estados distintos.
3. Estado de aceite, com nenhuma transição

A Figura 8 mostra como podem ser representados os estados desta nova notação no diagrama de estados.

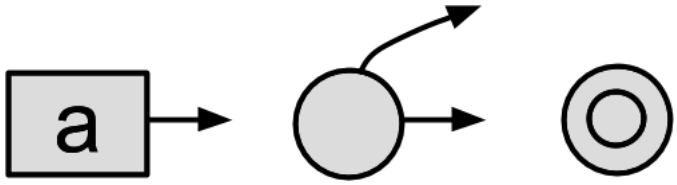


Figura 8: Representação de estados na nova notação

Esta representação facilita a visualização por omitir as transições $\epsilon$ e ressaltar a diferença entre os estados de consumo e os estados de decisão. Um exemplo de conversão para a nova notação pode ser visto na Figura 9.

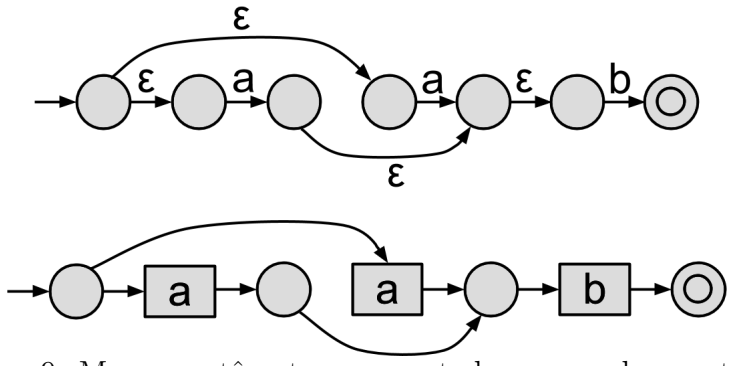


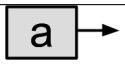
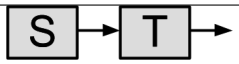
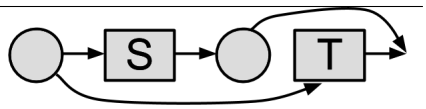
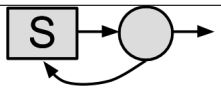
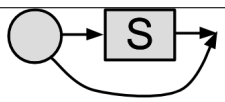
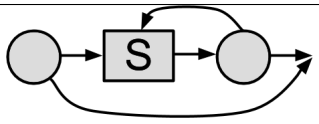

Figura 9: Mesmo autômato representado com as duas notações

### 2.6.3 *Construção*

A construção do autômato se dá pela conversão recursiva de cada um dos nós na árvore por uma construção de autômato predeterminada. Cada uma destas construções tem apenas um estado de entrada e todas as transições que não apontem para um nó dentro da própria construção, apontam para um único nó de saída.

Em uma implementação otimizada, este passo pode ser substituído por geração de código de máquina (como no artigo original). Entretanto, para fins didáticos, abordaremos algumas representações úteis do autômato gerado.

A tabela a seguir mostra a conversão para cada tipo de nó. Para fins de notação, as letras maiúsculas representam autômatos complexos produzidos em passos anteriores, mas que seguem a mesma regra definida no parágrafo anterior.

| **Tipo** | **Expressão** | **Construção** |
|---|---|---|
| Literal | $a$ |   |
| Sequência | $ST$ |   |
| Opção | $S\|T$ |   |
| Kleene+ | $S+$ |   |
| Opcional | $S?$ |   |
| Kleene* | $S*$ |   |

Para converter um autômato usando estas regras, basta percorrer a árvore de avaliação

de baixo para cima, inicialmente transformando todos os literais e recursivamente usando-os como insumo para os nós superiores. A Figura 10 mostra um exemplo de conversão utilizando a expressão $(a|a) + b$.

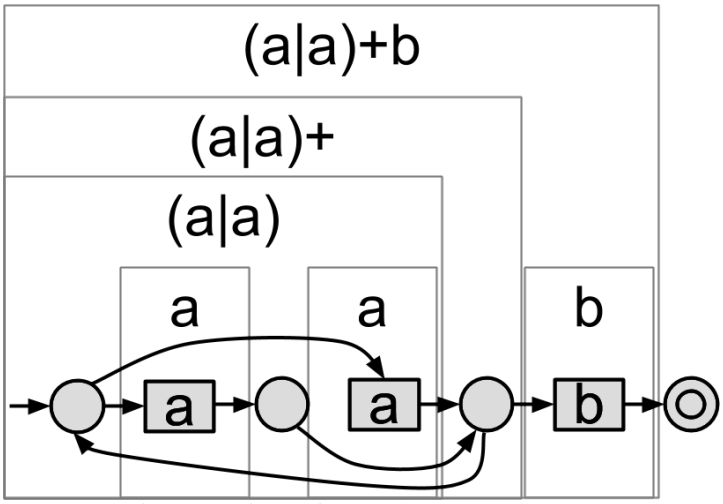



Figura 10: Exemplo completo de autômato para a expressão $(a|a) + b$

Uma das vantagens desta forma de construir é que ela pode ser escrita como uma sequência de instruções para serem interpretadas por uma máquina virtual. Cada um dos três tipos de estados definidos pela notação na seção anterior podem ser substituídos por uma entre três instruções. Estados de consumo como instruções *CONSUME*, estados de decisão como instruções *JUMP* e estados de aceite como instruções *MATCH*.

Por exemplo, o autômato representado pela Figura 10 pode ser reescrito como as instruções:

```
0000: JUMP (1, 3)
0001: CONSUME a
0002: JUMP (2,)
0003: CONSUME a
0004: JUMP (1, -4)
0005: CONSUME b
0006: MATCH!
```

As instruções *JUMP* são os que representam o não-determinismo destes autômatos, pois ao serem executadas, podem desviar para qualquer um dos rótulos para os quais apontam. Existem diversas formas de simular a execução desses programas não-determinísticos usando uma máquina de Turing determínistica. E é exatamente neste ponto onde a maior parte das implementações torna-se exponencial desnecessariamente.

## 2.7 Simulação do Autômato

Simular a execução de um autômato não-determinístico utilizando uma máquina determinística é um problema por si só complicado. Existe potencialmente um número exponencial de caminhos possíveis num autômato não-determinístico (RABIN; SCOTT, 1959).

Dentre as opções para simulação da execução do autômato, estão:

- *Backtracking*: tentar todos os caminhos possíveis, voltando para a última escolha feita em caso de falha.
- *Backtracking com memorização.*
- *Conversão para DFA*: converter para um autômato determinístico antes de executar.
- *Simulação paralela*: simular a execução de todos os estados simultaneamente.
- *Conversão preguiçosa para DFA*: semelhante a execução paralela, mas construindo um cache dos estados alcançados

Cada uma destas opções tem caracteristicas próprias. Suas vantagens e desvantagens serão discutidas separadamente.

### 2.7.1 *Backtracking*

Backtracking é talvez a forma mais trivial de implementar a simulação de autômatos não determinísticos. Consiste basicamente em ao ser apresentado a uma escolha, decidir por uma delas, mas manter uma pilha de escolhas feitas para retroceder e escolher novamente caso não tenha sucesso.

*Vantagem*: Este método é o mais fácil de implementar, pois como a chamada de subrotinas na maior parte das linguagens já implementa uma pilha, uma simples recursão é o suficiente para implementar esta técnica.

*Desvantagem*: Pela natureza do autômato, cada escolha pode gerar vários fluxos independentes, então uma sequência de n escolhas (assumindo que sejam binárias) pode gerar até $2^n$ fluxos diferentes. Além disso, esta técnica requer múltiplas passagens pela mesma string, o que é trivial se ela estiver em memória, mas complicado se estiver vindo de um fluxo de rede ou pipes do sistema. E por fim, a pilha reservada para chamadas de métodos

é compartilhada com outros recursos e limitada em diversos ambientes, impossibilitando, por exemplo, o reconhecimento de strings muito grandes.

### 2.7.2 *Backtracking com Memorização*

O maior problema de usar backtracking talvez seja a quantidade de fluxos diferentes que a implementação pode alcançar. Entretanto, para o problema de reconhecimento, a resposta é invariante se definirmos em termos de $s_i, q_j$ tal que $s_i \in S, q_j \in Q$, onde $S$ é a entrada e $Q$ o conjunto de estados do autômato.

Definindo assim, se $n = |S|, m = |Q|$, então existem apenas $n \times m$ possíveis configurações para o problema, que podem ser devidamente memorizadas. Então um problema que era $O(2^n)$ torna-se $O(n \times m)$.

*Vantagem*: Esta técnica é tão fácil de implementar quanto o backtracking puro e tem complexidade bem menor.

*Desvantagem*: Ainda é necessário passar multiplas vezes pela mesma string. E a pilha ainda pode ser um fator limitante.

### 2.7.3 *Conversão para DFA*

Autômatos finitos determinísticos são muito mais fáceis de simular, visto que se aproximam muito mais intimamente aos computadores que conhecemos atualmente.

É sabido que autômatos finitos determinísticos (DFA) e não-determinísticos (NFA) são equivalentes em termos de poder expressivo. É possível converter entre um e outro com métodos conhecidos. NFAs tem baixa complexidade de memória, mas são mais custosos em tempo de execução. DFAs, entretanto permitem uma execução mais rápida ao custo de uma alta complexidade de memória.

Um NFA com m estados, quando convertido para um DFA pode ter ate $2^m$ estados. Em termos de memória utilizada, NFAs são exponencialmente mais expressivos que DFAs. Entretanto, DFAs tem execução com menor complexidade que NFAs. Para uma entrada com n caracteres, NFAs podem ser simulados em $O(n \times m)$, enquanto DFAs em $O(m)$.

A conversão entre os dois se dá através do algoritmo conhecido como *Rabin-Scott powerset construction* (RABIN; SCOTT, 1959). Este algoritmo consiste basicamente em simular a execução paralela do autômato para todas as entradas possíveis.

*Vantagem*: Depois de construído o DFA, ele pode ser usado para reconhecer qualquer string em $O(m)$.

*Desvantagem*: A construção do DFA tem complexidades de tempo e espaço exponenciais. E é bem mais complexa do que o backtracking puro.

### 2.7.4 *Simulação paralela*

Uma forma de simular a execução de autômatos é, sempre que confrontado com uma decisão, escolher as duas opções simultaneamente, e manter fluxos paralelos sendo executados.

Inicialmente, pode parecer que esta solução leve a um conjunto de até $2^n$ fluxos executando simultaneamente, porém, como um autômato tem um número finito de estados, existem no máximo $m$ fluxos simultâneos para cada caractere. Eventualmente, múltiplas decisões podem colidir num mesmo estado, eliminando assim o caráter exponencial da avaliação. Assim, para uma entrada com n caracteres avaliada contra um autômato de m estados, no pior caso, a complexidade de tempo é de $O(n \times m)$.

O objetivo é que durante a execução, seja lido um caractere por vez. Então, antes de cada execução é preciso avaliar todas as transições $\epsilon$ e deixar todos os fluxos em espera posicionados em estados do tipo *consumo*.

A Figura 11 mostra a execução do autômato gerado para a expressão regular $(a|a)+b$ contra a entrada $aa$.

*Vantagem*: Tem tempo de execução polinomial ($O(n \times m)$). Só precisa ler a entrada uma vez. Não requer uso de pilha.

*Desvantagem*: Precisa de memória adicional para guardar os estados executados ($O(m)$). Em comparação com execução com DFA puro, é mais lento.

### 2.7.5 *Conversão preguiçosa para DFA*

O maior perigo da conversão completa para DFA é o potencial explosivo que a geração do autômato determinístico tem. Até $2^n$ estados podem ser gerados, e tempo proporcional a isso será gasto. Por outro lado, a simulação parelela parece gastar uma quantidade linear de memória e tempo polinomial para simular o autômato.

Para a maior parte das aplicações práticas, dificilmente a quantidade de estados no

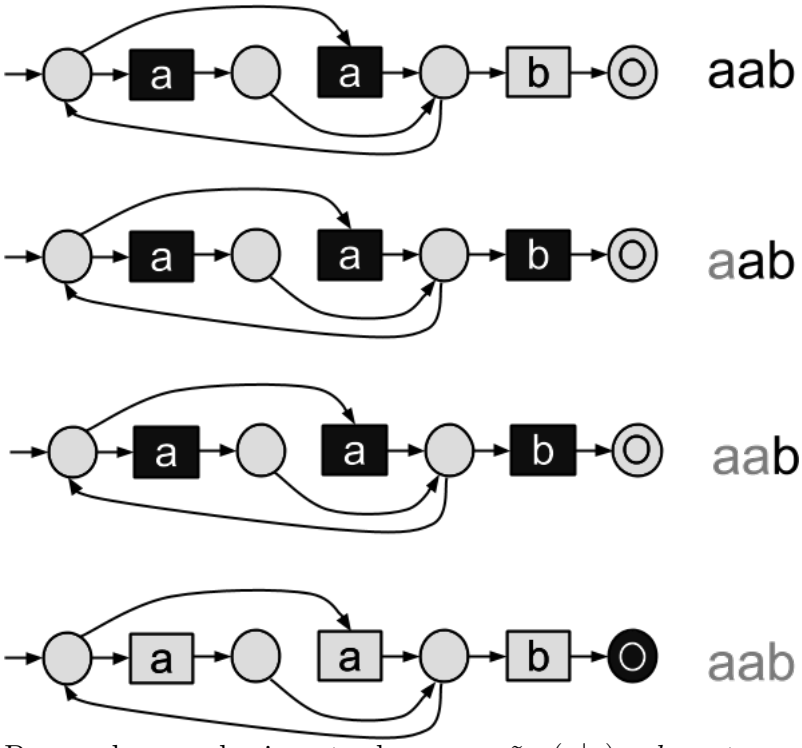



Figura 11: Passos do reconhecimento da expressão $(a|a) + b$ contra a entrada $aab$

DFA será tão grande. Por isso, uma aproximação combinada entre as duas abordagens poderia trazer vantagens.

Durante a simulação paralela, os estados ativados simultaneamente correspondem exatamente a um estado no respectivo DFA. Ao realizar cache destes estados (porém mantendo uma política de gerência de memória, descartando os mais antigos) a implementação estará mantendo a complexidade de execução da simulação paralela, porém aumentando as chances de eventualmente gerar o DFA completamente.

*Vantagem*: Tem tempo de execução linear no caso médio ($O(n)$). Só precisa ler a entrada uma vez. Não requer uso de pilha.

*Desvantagem*: Pior caso ainda é $O(n \times m)$. Precisa de memória adicional para guardar os estados executados e o cache de estados do DFA. Implementação consideravelmente mais complexa.

# Capítulo 3
# Implementação

Neste capítulo será apresentada a implementação da biblioteca *PyRex* (expressões regulares em Python). A implementação na íntegra pode ser visualizada no Apêndice A.

Neste projeto o objetivo era alcançar uma implementação completamente polinomial para expressões regulares. Alcançado o primeiro objetivo, a implementação foi focada na simplicidade. Foram evitados truques de implementação que a tornassem mais rápida em detrimento da legibilidade. Todas as escolhas foram para reduzir a complexidade da implementação. Mesmo assim, o algoritmo utilizado foi eficiente o bastante para atingir o objetivo inicial.

A linguagem escolhida para implementar o projeto foi Python. É uma linguagem de máquina virtual, tal qual o Java. Entretanto, sua máquina virtual não tem as otimizações em tempo de execução que as implementações mais famosas de Java têm. Além disso, a resolução dinâmica de tipos a faz ter uma performance inferior à de outras linguagens com resolução estática de tipos.

Mesmo assim, pela reduzida quantidade de caracteres especiais em sua sintaxe, os programas escritos nela são em geral mais legíveis e simples. Por isso, toda a implementação do PyRex tem menos de 90 linhas de código (contando as em branco) e menos de 3KB. Esta implementação abrange desde a análise sintática da expressão regular até a simulação do autômato retornando, inclusive, a região do reconhecimento da expressão.

O projeto foi implementado utilizando apenas as bibliotecas padrão disponíveis no Python 2.7, porém ele também funciona no Python 3.0 e superiores.

## 3.1 Funcionalidades Implementadas

Foi escolhido um conjunto de funcionalidades que fosse capaz de mostrar o ponto de falha das implementações modernas de expressões regulares e ainda assim ser simples o suficiente para ter uma implementação que pudesse ser explicada em alguns minutos.

Assim, foi implementado o reconhecimento de:

- Literais (e.g. $a$)
- Sequências (e.g. $ab$)

- Alternâncias (e.g. $ab|cd$)
- Agrupamentos (e.g. $a(bc)d$)
- Expressão opcional (e.g. $a?$)
- Kleene* (e.g. $a*$)
- Kleene+ (e.g. $a+$)

Estas funcionalidades são bem próximas às definidas por Thompson (THOMPSON, 1968) em seu artigo e permitem denotar inequivocamente linguagens regulares.

## 3.2 Estrutura do Código

O código está dividido em basicamente duas partes: um *parser* e um *matcher*.

O *parser* é responsável por receber uma string com a expressão regular, analisá-la sintaticamente e retornar um objeto (o *matcher*) capaz de reconhecer strings que façam parte da linguagem regular definida.

O *matcher* contém informações do autômato gerado e funções que sabem simular a execução do autômato. Ele é implementado como uma classe Python com um método *match*, além da sobrecarga do operador *__repr__* do Python, que provê uma visualização útil do objeto em tempo de depuração.

O uso desta biblioteca se dá pela compilação do autômato usando a função *rex* e a posterior avaliação de strings usando o método *match* da classe *Machine*. Um exemplo de uso pode ser visto na listagem 1.

```
1 import pyrex
2 machine = pyrex.rex('a(ab)+')
3 print machine.match('aabab')
```

Listagem 1: Exemplo de uso da biblioteca PyRex

O retorno da chamada da função match pode ser tanto uma tupla com dois elementos (índices de início e final do match) ou uma referência nula (*None* em Python). No exemplo da Listagem 1, o retorno seria uma tupla $(0, 5)$, indicando que o match começa no índice 0 da string e termina no índice 4. Perceba que a tupla denota um intervalo aberto à direita.

## 3.3 Análise sintática

Análise sintática é feita pela função *rex*. Ela implementa um parser recursivo descendente rudimentar. A BNF está descrita abaixo:

```
<option>     ::= <sequence> { "|" <sequence> }*
<sequence>   ::= { <repetition> }*
<repetition> ::= <primary> { "*" | "?" | "+" }*
<primary>    ::= "." | <literal> | "(" <option> ")"
```

É importante chamar a atenção para o detalhe da regra *sequence* que permite strings vazias. Esta é uma mudança que permite strings vazias como expressão regular (o que é útil em alguns casos). Entretanto, esta mesma mudança permite expressões regulares do tipo $a||b$. O que não chega a constituir um erro. A Listagem 2 mostra a implementação desta regra.

```
1 def sequence():
2     e = []
3     while tokens and tokens[0] not in '|)':
4         e += repetition()
5     return e
```

Listagem 2: Implementação da regra *sequence*

A implementação é bem simples, usa uma instância da classe *deque* como buffer de tokens e chamadas recursivas para implementar as regras. Para cada uma das quatro regras defindas pela BNF, há um método aninhado em *rex* para representá-la.

O tratamento de erros é simples porém eficiente. Sempre que um caractere não esperado é encontrado na entrada, ele é reportado. Um dos mecanismos pode ser observado no código principal do parser (Listagem 3), onde após consumir todos os caracteres possíveis, se ainda existirem remanescentes na entrada, uma exceção é lançada.

```
1 e = option()
2 if tokens:
3     raise Exception('Not expected: "{}"'.format(''.join(tokens)))
4
5 return Machine(e)
```

Listagem 3: Exemplo de tratamento de erros no parser

## 3.4 Representação

Durante o parse, a expressão regular é transformada numa representação do autômato relacionado. Este autômato é representado como um *array de objetos*, onde cada posição do array representa um estado do autômato.

Como definido na Seção 2.6.2 (Página 26), a notação utilizada neste autômato prevê três tipos de estado: consumo, decisão e aceite. Estados de consumo são representados pelo caractere que consomem. Estados de decisão são representados por uma tupla de inteiros que informam quantos estados à frente ou atrás deve-se 'pular'. Estados de aceite não precisam de representação, pois sempre ocorrem no final do autômato (segundo esta notação).

A Figura 12 e as listagens 4 e 5 representam a expressão regular $(a|a) + b$. Apenas a forma em array é armazenada na instância da classe Machine para execução. Entretanto, a forma em instruções pode ser gerada através do método *__repr__* da mesma classe.

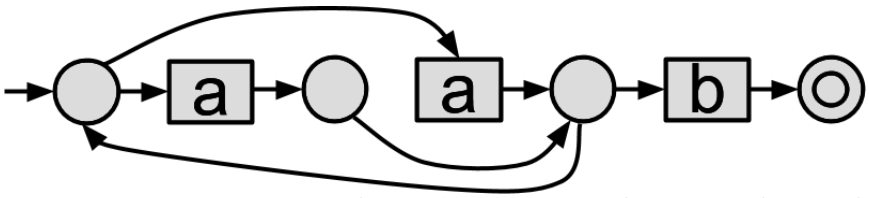



Figura 12: Representação do autômato como diagrama de estados

```
1  [(1, 3), 'a', (2,), 'a', (1, -4), 'b']
```

Listagem 4: Representação do autômato como array

```
1 0000: JUMP (1, 3)
2 0001: CONSUME a
3 0002: JUMP (2,)
4 0003: CONSUME a
5 0004: JUMP (1, -4)
6 0005: CONSUME b
7 0006: MATCH!
```

Listagem 5: Representação do autômato como instruções

## 3.5 Simulação

A simulação é executada pelos métodos *matcher* e *match* da classe *Machine*. A diferença entre os dois é que o primeiro retorna um iterador que avalia o estado da execução do autômato para cada caractere da entrada. O segundo apenas retorna o resultado do match no final. Na implementação, o método *match* utiliza o método *matcher*, como mostra a Listagem 6.

```
1 def match(self, string):
2     return reduce(lambda answer, s: s[1], self.matcher(string), None)
```

Listagem 6: Implementação do método *match*

O método implementado para a simulação do autômato é a *simulação paralela*. Este método foi escolhido por combinar uma complexidade assintótica polinomial com uma facilidade de implementação razoável.

Como é costumeiro na implementação de expressões regulares, o match é realizado caso qualquer substring do input pertença à linguagem definida. O método de simulação paralela leva vantagem por necessitar de pouca alteração para suportar este modo de execução.

A implementação baseia-se em ciclos de consumo, onde um conjunto de estados iniciais passa simultâneamente por uma transição pelo mesmo caractere $c$ para um conjunto de estados sequentes.

Para representar os estados, utilizamos duas listas, A e B, que em todo momento representam respectivamente: o conjunto de estados *de consumo* alcançados até o início da avaliação do caractere atual (A) e o conjunto de estados que a avaliação do caractere atual irá alcançar (B). Após a avaliação de cada caractere estas listas são invertidas.

Cada estado nestas listas vem acompanhado da posição na string original onde o match começou. Esta informação é criada quando o estado inicial é colocado na lista para cada um dos caracteres. Ela é então copiada cada vez que o estado é resolvido e avança para seus sucessores. Na lista, o estado é representado por uma tupla $(start, j)$, onde $start$ representa a posição na entrada onde o match começou e $j$ é o índice do estado atual.

É importante ressaltar que um mesmo estado nunca entra na lista B duas vezes no mesmo ciclo de consumo. Esta verificação é efetuada pelo array V, que controla o índice do último caractere onde cada estado entrou na lista de estados. Inicialmente, esta lista está preenchida com valores -1.

Como apenas estados de consumo entram nas listas de estados para participar do ciclo de consumo (por razões óbvias), é preciso avaliar todos os estados de decisão antes do início do ciclo de consumo.

A Listagem 7 mostra a implementação do método *addnext* que adiciona estados à próxima lista de consumo (B), resolvendo recursivamente estados de decisão em seus respectivos próximos estados de consumo. Este método também retorna o número de vezes que o estado de aceite foi alcançado.

```
1  def addnext(start, i, j):
2      if j==self.n: return 1
3      if V[j] == i: return 0
4      V[j] = i
5
6      if isinstance(self.states[j], tuple):
7          return sum(addnext(start, i, j+k) for k in self.states[j])
8
9      B.append((start, j))
10     return 0
```

Listagem 7: Implementação do método *addnext*

Assim, cada ciclo de consumo é marcado pelos passos:

- Iniciar novo fluxo no estado inicial com o caractere atual.
- Inverter as listas de consumo, limpando a lista de próximo (B).
- Avaliar o consumo do caractere atual adicionando os estados subsequentes na lista de próximo.

A Listagem 8 mostra a implementação destes passos. A linha 5 adiciona um novo fluxo começando no caractere $i$. As linhas 8 e 9 invertem as listas de consumo. Finalmente, as linhas 11-13 avaliam os estados de consumo contra a entrada, avançam os estados devidos, além de atualizar a melhor resposta, caso algum desses estados alcance o estado de aceite.

```
1  def key(a): return (a[1]-a[0], -a[0]) if a else (0, 0)
2
3  answer = None
4  for i, c in enumerate(string):
5      addnext(i, i, 0)
6      yield i, answer, B
7
8      A, B = B, A
9      del B[:]
10
11     for start, j in A:
12         if self.states[j] in (None, c) and addnext(start, i+1, j+1):
13             answer = max(answer, (start, i+1), key=key)
```

```
14
15 yield len(string), answer, B
```

Listagem 8: Implementação do método *matcher*

## 3.6 Visualização

Valendo-se da implementação do método *matcher* foi possível construir uma visualização em modo texto, passo-a-passo, da execução do algoritmo, que visa ajudar na compreensão. Trata-se de uma ferramenta de linha de comando (também escrita em Python) que utiliza o PyRex .

É possível chamar o visualizador com o comando *./view.py (regex) (input)*, e.g.:

```
./view.py a+b+c+ aabbbcccc
```

No exemplo, a execução do programa iria mostrar os passos do match desta expressão regular. O autômato gerado pode ser visualizado na Figura 13. Alguns passos da execução podem ser vistos nas Figuras 14 e 15.

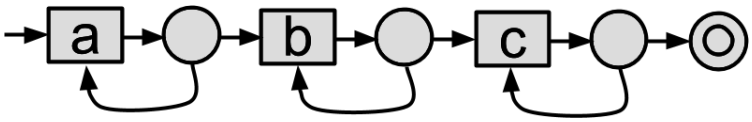



Figura 13: Autômato para expressão $a + b + c+$

```
Best answer: <none>
Input: aabbbcccc
          ^ (3)

  3 >0000: CONSUME a
    0001: JUMP (1, -1)
  0 >0002: CONSUME b
    0003: JUMP (1, -1)
  0 >0004: CONSUME c
    0005: JUMP (1, -1)
    0006: MATCH!
```

Figura 14: Exemplo de visualização lendo o 4º caractere da string

Esta forma de exibir mostra o estado atual de todos os fluxos sendo executados pelo autômato. O número exibido ao lado de cada instrução é o índice da entrada onde aquele fluxo começou. No exemplo específico, há três fluxos em execução:

- O que começa com cada caractere da string. (3)
- O que continua lendo caracteres b da entrada, enquanto houver. (0)
- O que está pronto para ler caracteres c, quando começarem. (0)

```
Best answer: aabbbc (0, 6)
Input: aabbbcccc
             ^ (6)

  6 >0000: CONSUME a
    0001: JUMP (1, -1)
    0002: CONSUME b
    0003: JUMP (1, -1)
  0 >0004: CONSUME c
    0005: JUMP (1, -1)
    0006: MATCH!
```

Figura 15: Exemplo de visualização lendo o 7º caractere da string

Neste momento, já existe um match para a string (visto que já existem caracteres c lidos). O melhor match até o momento é mostrado na linha *Best answer*. Este valor é atualizado a cada passo. Além disso, os seguintes estados estão ativos:

- O que começa com cada caractere da string. (6)
- O que continua lendo caracteres c da entrada, enquanto houver. (0)

A ferramenta utiliza o método matcher, que retorna um iterador que, a cada iteração, retorna uma tupla $(i, answer, state)$, onde $i$ é o índice atual que está sendo lido na entrada, $answer$ é a melhor resposta conseguida até o momento e $state$ é a lista de estados atuais, que é composta de várias tuplas $(start, j)$.

O código-fonte completo da ferramenta pode ser visto no Apêndice B.

# Capítulo 4
# Análise dos Resultados

Neste capítulo serão abordadas as principais diferenças entre a implementação proposta neste trabalho em comparação com outras implementações populares.

Algumas comparações com implementações específicas serão feitas, por isso, faz-se necessário definir claramente as caracteristicas e versões destas implementações. Segue uma tabela que clarifica tais questões:

| **Implementação** | **Ambiente** | **Linguagem original** |
|---|---|---|
| pyrex | Python 2.7.3 | Python |
| re | Python 2.7.3 | C |
| Oniguruma | Ruby 1.9.3 | C |
| java.util.regex | Java 1.6.0_45 | Java |

## 4.1 Retorno do Método *match*

O problema de reconhecer palavras usando expressões regulares é uma problema de *decisão*. Isto significa que ele aceita apenas uma entre duas respostas: *sim* ou *não*. Entretanto, provou-se útil ao longo do tempo não só reconhecer palavras em linguagens regulares, mas também qualquer sub-palavra pertencente a ela, retornando, então, sua localização na palavra original.

Uma expressão regular *abcd* retornaria um match válido numa palavra *zzabcdzz*. Este comportamento permite que múltiplas soluções sejam válidas. Por exemplo, a mesma expressão poderia encontrar dois resultados diferentes na palavra *abcdabcd*. Muitos resultados poderiam se sobrepor, inclusive. A expressão $a*$, ao avaliar a palavra *aaaa* pode levar a 10 resultados diferentes. A implementação utilizada influencia bastante na escolha de qual dos resultados deve ser retornado.

A implementação proposta neste projeto sempre favorece as respostas mais longas. Em caso de empate, é favorecida a resposta que inicie mais à esquerda na entrada. Este comportamento é garantido pela função *key*, representada na Listagem 8 na Página 38.

Este comportamento é coerente com o artigo original de Thompson e, consequentemente, com as implementações de ferramentas como *sed* e *grep* do sistema operacional Unix. Por outro lado, diverge de outras implementações usuais baseadas na do Perl,

que geralmente retornam a primeira resposta encontrada seguindo uma ordem gulosa de avaliação (COX, 2007).

Uma forma fácil de verificar esta diferença é comparar o match da expressão $a*(b|abc)$ contra a string $abc$. A implementação padrão do Python retornaria a string $ab$. A implementação neste projeto encontrará a string $abc$. A diferença é que a primeira tenta avaliar o máximo que pode na repetição $a*$, enquanto a segunda consegue visualizar globalmente os matches possíveis.

## 4.2 Reconhecimento de *backreferences*: um problema NP-difícil

No Capítulo 2 foram mostradas diversas formas de simular o autômato construído a partir da expressão regular. Cada uma delas tem características particulares de complexidade de tempo e espaço, além de dificuldades de implementação diversas.

O que se propõe nesta seção é demonstrar que qualquer implementação de expressões regulares que suporte *backreferences* necessariamente torna o *match* um problema NP-difícil. Este resultado mostra que qualquer algoritmo conhecido é suficientemente ineficiente para aplicações onde o tempo de execução é crítico.

### 4.2.1 *Problemas NP-difíceis*

A classe de problemas NP-difíceis (*Non-deterministic Polynomial-time hard*) são aqueles para os quais existem problemas NP-completos redutíveis a eles em tempo polinomial. Nem todos os problemas em NP-difícil são de decisão, diferentemente dos problemas em NP.

A classe NP diz respeito a todos os problemas de *decisão* que podem ser resolvidos em tempo polinomial em uma máquina de Turing não-determinística. Isto inclui também os problemas de decisão que podem ser resolvidos em máquinas de Turing determinísticas em tempo polinomial (conjunto P). Não há até o momento qualquer prova conclusiva se há algum problema em NP que não esteja também em P. Também não se sabe se todos os problemas de NP estão em P. A Figura 16 mostra como seria o diagrama de conjuntos em ambos os casos.

Uma característica importante dos problemas em NP é que se a resposta para o problema for *sim*, dada um *certificado* (ou *testemunha*) que mostre tal fato, é possível

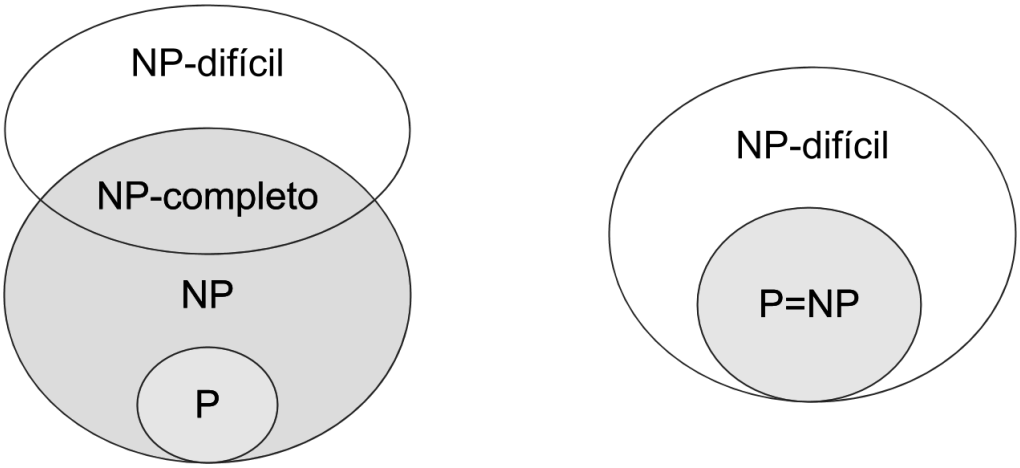



Figura 16: Diagrama de conjuntos caso $P \neq NP$ e $P = NP$, respectivamente

determinar em tempo polinomial se esta resposta está correta ou não utilizando uma máquina de Turing determinística.

Nos anos 70, Stephen Cook introduziu o conceito de problemas NP-completos (COOK, 1971). São problemas em NP tais que qualquer outro problema em NP pode ser reduzido a eles em tempo polinomial. Qualquer solução polinomial de um problema NP-completo demonstraria que todos os problemas em NP podem ser resolvidos polinomialmente em máquinas de Turing determinísticas.

Cook também provou provou que o problema de satisfabilidade booleana (SAT) é NP-completo. Resultado parecido com o que Leonid Levin alcançou independentemente em 1973 (LEVIN, 1973). Hoje este resultado é conhecido como teorema de Cook–Levin.

O problema de satisfabilidade consiste em determinar se existe uma interpretação que satisfaça uma expressão booleana, i.e. se existe uma atribuição de variáveis que avalie tal expressão como *verdadeira*. Neste problema, o *certificado* consiste no conjunto de valores a se atribuir às variáveis para que a expressão seja avaliada como verdadeira. Para verificá-la, basta substituir por tais valores e avaliar a expressão resultante.

Certas variações de SAT podem pertencer a classes de problemas diferentes. Uma variação possível é restringir às expressões booleanas a sempre estarem na CNF (*Conjunctive Normal Form* ou Forma Normal Conjuntiva).

Se a expressão booleana estiver escrita na 2CNF (forma normal conjuntiva com no máximo 2 variáveis por cláusula) este problema torna-se polinomial. Neste caso, é referido como 2-satisfabilidade (2SAT). Entretanto, se a expressão estiver escrita na 3CNF, o problema (3SAT) continua NP-completo (KARP, 1972).

Uma expressão está na 3CNF se ela está na forma:

$$(A_{11} \vee A_{12} \vee A_{13}) \wedge (A_{21} \vee A_{22} \vee A_{23}) \wedge \cdots \wedge (A_{n1} \vee A_{n2} \vee A_{n3}),$$

onde $A_{ij}$ pode ser tanto uma variável ($X$) quanto a negação de uma variável ($\neg X$).

### 4.2.2 *Reconhecimento de* backreferences

Durante o reconhecimento de expressões regulares, uma funcionalidade comum em implementações modernas é a capacidade de definir grupos de captura. Uma expressão $(a+)(b+)(c+)$ teria quatro grupos de captura, numerados de 0 a 3. E o reconhecimento de uma string com esta expressão resultaria em uma tupla com o valor capturado por cada um dos grupos. A string *aabbbcccc* resultaria nas capturas $(aa, bbb, cccc)$.

Baseado nesta funcionalidade, muitas implementações modernas adicionam a capacidade de referenciar tais grupos de captura ainda durante o match de uma string. Por exemplo, dada uma linguagem $L = \{a^n b a^n \mid n \in \mathbb{N}\}$, seria possível denotá-la com a expressão $(a*)b\backslash 1$. A referência ao grupo de captura 1 $(a*)$ se dá pela expressão $\backslash 1$.

Denotaremos aqui o problema de reconhecimento de expressões regulares com *backreferences* como MATCH.

---

MATCH: Dada uma expressão regular $E$ com *backreferences* abrangendo $n$ grupos de captura e $S$ a string a ser reconhecida, encontrar tupla $C = (c_1, c_2, \cdots, c_n)$, onde $c_i$ é substring de $s$ relativa ao reconhecimento do $i$-ésimo grupo de captura.

---

**Teorema 1** MATCH *é NP-difícil.*

*Demonstração*: 3SAT é redutível para MATCH utilizando a seguinte lógica: para resolver uma fórmula com $n$ variáveis e $m$ cláusulas, define-se a expressão regular

$$E = \wedge(x?)^n.*; (? : \backslash 1 v_{11} | \backslash 2 v_{12} | \backslash 3 v_{13}), (? : \backslash 1 v_{21} | \backslash 2 v_{22} | \backslash 3 v_{23}), \ldots (? : \backslash 1 v_{m1} | \backslash 2 v_{m2} | \backslash 3 v_{m3}), \$$$

e a string a ser reconhecida

$$S = x^n; x,^m$$

onde $v_{ij} = x$ se a variável $j$ da cláusula $i$ aparecer negada na fórmula, caso contrário, $v_{ij} = \epsilon$. Isto força o reconhecimento da *backreference* $\backslash j$ como $x$ ou $\epsilon$ respectivamente, porém pelo menos uma das variáveis precisa reconhecer a substring $x$, definida para cada cláusula na string $S$. Assim, existem grupos de captura $C = (c_1, c_2, \cdots, c_n)$ se e somente se for possível reconhecer tanto as expressões dos grupos de captura ($x$ ou $\epsilon$ para *verdadeiro* ou $falso$, respectivamente), como as *backreferences* que representam as cláusulas da fórmula de forma consistente. Como tanto a expressão $E$ quanto a string $S$ podem ser construídas em $O(n + m)$ MATCH é NP-difícil. ∎

### 4.2.3 *Implementação de* 3SAT *com expressões regulares*

Para exemplificar a demonstração da seção anterior, será descrita aqui uma implementação que permite a solução de instâncias do problema 3SAT utilizando as expressões regulares da linguagem Python.

Como exemplo, pode-se observar a fórmula:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3).$$

É possível construir uma expressão regular que represente esta expressão, e.g.

$$\wedge(x?)(x?)(x?).*; (? : \backslash 1|\backslash 2|\backslash 3x), (? : \backslash 1|\backslash 2x|\backslash 3), (? : \backslash 1x|\backslash 2x|\backslash 3), (? : \backslash 1x|\backslash 2x|\backslash 3x), \$$$

Ao tentar realizar match desta expressão com a string $xxx; x, x, x, x$, o resultado será a tupla de strings $(`x', '', `x')$, denotando que caso $x_1 = verdadeiro, x_2 = falso, x_3 = verdadeiro$, a expressão original será satisfeita.

A primeira parte da expressão denota os grupos iniciais onde entre 0 e 3 variáveis podem ser verdadeiras (reconhecendo 'x'). A segunda parte da expressão define as cláusulas. Toda cláusula começa com a definição de um grupo que não gera captura, (? : .... Então cada variável é descrita com uma altenação. $\backslash i$ referencia a variável $x_i$, enquanto $\backslash ix$ denota sua negação.

A Listagem 9 mostra uma implementação em Python de 3SAT utilizando expressões regulares.

```
1 #!/usr/bin/env python
2 import re
3
```

```
clauses = [[1,2,-3], [1,-2,3], [-1,-2,3], [-1,-2,-3]]
#(x1 or not x2 or x3) and (x1 or not x2 or x3) and
#(not x1 or not x2 or x3) and (not x1 or not x2 or not x3)

v = reduce(lambda a, b: max(a, *b), clauses, 0)
s = v * 'x' + ';' + len(clauses) * 'x,'
e = '^' + v * '(x?)' + '.*;' + ''.join(
    '(?:' + '|'.join(
        '\\' + (str(-x) + 'x' if x < 0 else str(x))
        for x in clause) + '),'
    for clause in clauses)

print re.match(e, s).groups()
```

Listagem 9: Implementação de 3SAT com expressões regulares

## 4.3 Benchmarks

Apesar do match de *backreferences* ser um problema NP-difícil, match de expressões regulares por si só não é. O principal objetivo neste trabalho é mostrar que mesmo em casos onde as expressões poderiam ser avaliadas em tempo polinomial, elas não o são em uma grande parte de implementações modernas.

Para demonstrar este ponto, foram feitos alguns testes utilizando a mesma expressão regular em diversas implementações e comparando seus tempos de execução em relação ao crescimento da string de entrada. O código-fonte de todos os testes pode ser encontrado no Apêndice C. Todas as tabelas de valores podem ser encontradas no Apêndice D.

Todos os gráficos estão em escala logarítmica, para facilitar a visualização do grande intevalo de valores que cada implementação representa.

### 4.3.1 *Benchmark 1: (a?a)+b*

Neste teste a string $a^n$ foi avaliada contra a expressão regular $(a?a) + b$. O objetivo é mostrar o comportamento exponencial de implementações baseadas em backtracking. A expressão escolhida força a escolha para cada caractere da entrada sobre usar ou não o caractere $a$ na expressão $a?$.

A Figura 17 mostra o tempo de execução para as diferentes implementações. A tabela na Seção D.1 (Página 59) contém os dados para este gráfico.

Um ponto interessante a ser observado é que a implementação em Java, que inicial-

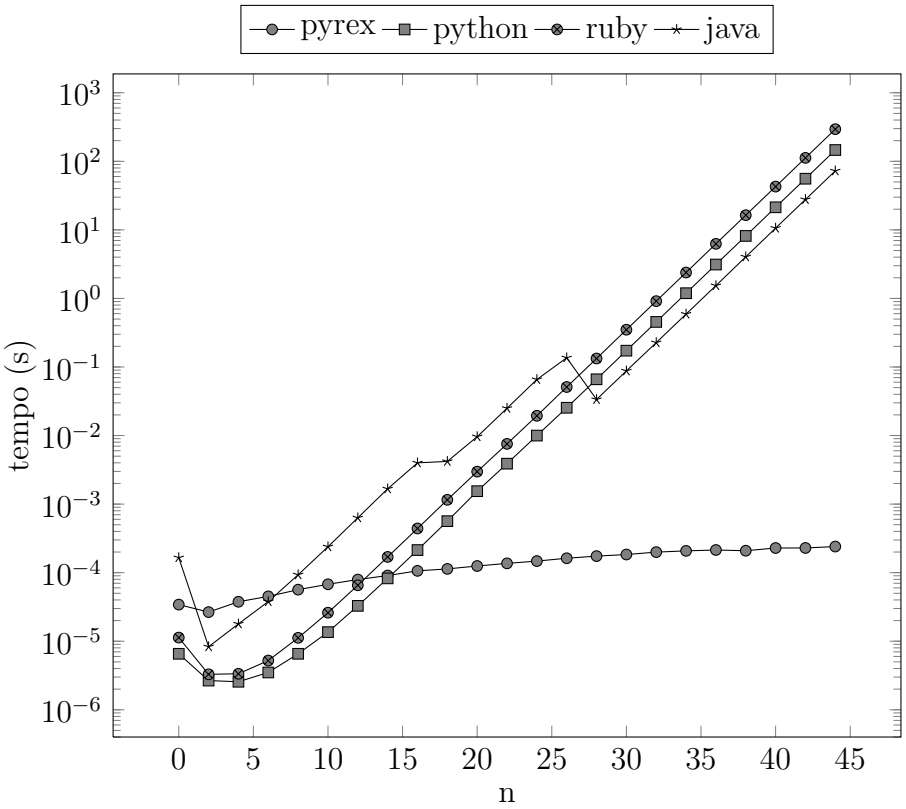



Figura 17: Tempo de execução (em escala logarítmica) para match de $a^n$ contra $(a?a) + b$

mente tem a pior constante de todas, rapidamente sofre compilação JIT (*Just in Time*), que faz com que o tempo de execução caia bastante, porém mantendo seu caráter exponencial.

### 4.3.2 ***Benchmark 2: a*b***

Neste teste, o objetivo era demonstrar a alta constante da implementação PyRex em casos onde o tempo de execução é linear para todas as implementações.

A Figura 18 mostra o tempo de execução para as diferentes implementações. A tabela na Seção D.2 (Página 60) contém os dados para este gráfico.

É possível observar uma clara desvantagem de implementações em código gerenciado (PyRex e Java) contra outras implementações escritas em C (Ruby e Python) para casos onde a complexidade de tempo é linear.

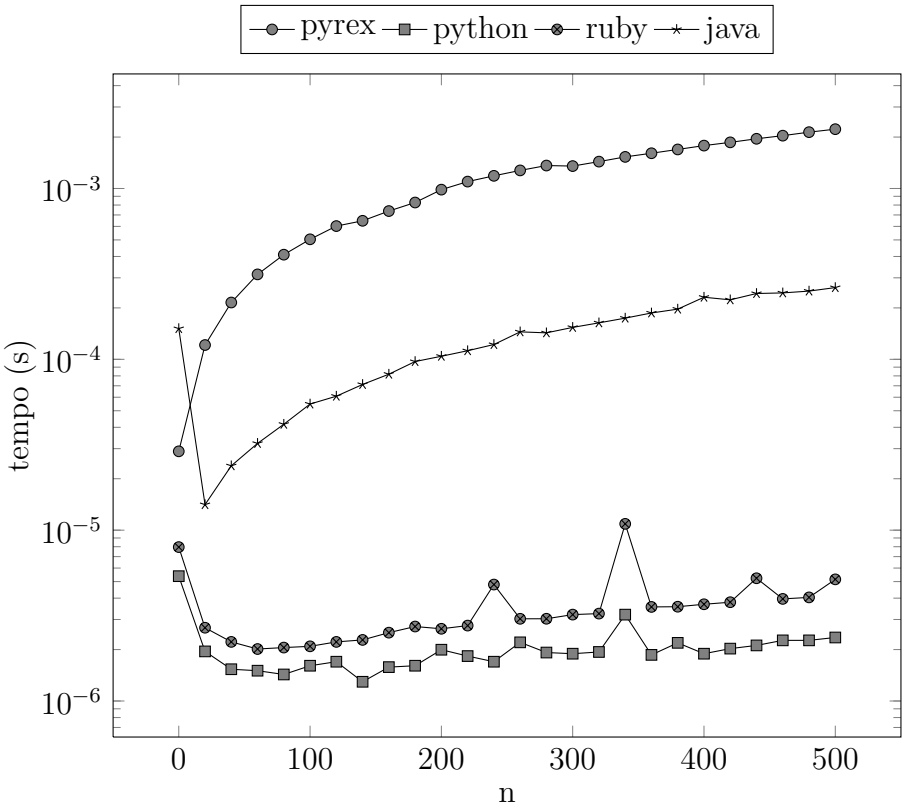



Figura 18: Tempo de execução (em escala logarítmica) para match de $a^n$ contra $a * b$

### 4.3.3 *Benchmark 3: a*a*a*a*a*b*

Neste teste, o objetivo era demonstrar como mesmo em casos onde implementações baseadas em backtracking não geram tempos de execução exponenciais, ainda assim é possível alcançar tempos de execução maiores que $O(n)$. Esta expressão força um backtracking para consumo de várias repetições $a*$. O tempo de execução esperado para implementações baseadas em backtracking é $O(n^5)$.

A Figura 19 mostra o tempo de execução para as diferentes implementações. A tabela na Seção D.3 (Página 61) contém os dados para este gráfico.

Este resultado mostra ainda como certas otimizações estáticas permitem que a implementação do Ruby mantenha seu tempo de execução linear enquanto a implementação do Python, que também é escrita em C, tenha uma complexidade de tempo maior que a do PyRex.

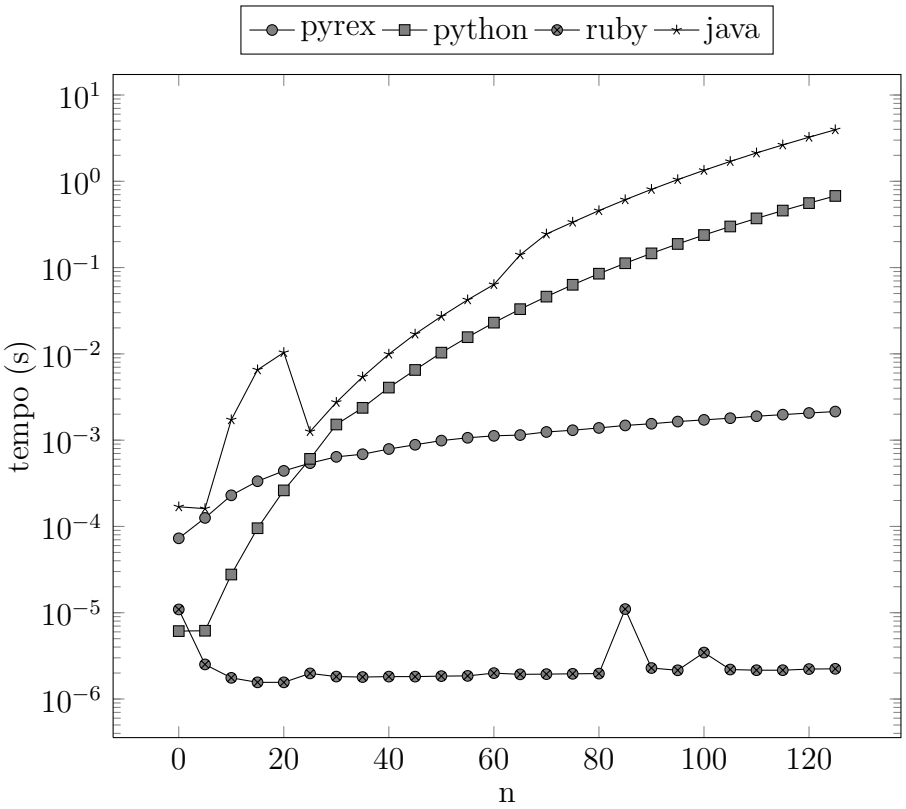



Figura 19: Tempo de execução (em escala logarítmica) para match de $a^n$ contra $a * a * a * a * a * b$

# Capítulo 5
# Conclusão

## 5.1 Contribuição

A principal contribuição neste trabalho foi uma implementação didática de expressões regulares utilizando o método de construção de Thompson. Nela foi utilizada uma forma simples de representar os estados do autômato finito não-determinístico como instruções de uma máquina abstrata que pode ser implementada em qualquer plataforma. A implementação foi feita em Python para fins de simplicidade e didática.

Os resultados alcançados foram bastante satisfatórios, mostrando que é possível implementar expressões regulares com algoritmos polinomiais. A implementação, apesar de não ser a mais eficiente para casos triviais, mostrou-se muito útil para demonstrar didaticamente a teoria apresentada no Capítulo 2.

No Capítulo 4 mostrou-se com sucesso que o problema reconhecimento de expressões regulares com *backreferences* é NP-difícil; mostrou-se também a incapacidade de implementações em linguagens como Python, Ruby e Java de reconhecê-las em tempo polinomial, mesmo para expressões sem *backreferences*.

Estes resultados confirmam a necessidade de parcimônia ao utilizar expressões regulares, principalmente em softwares que compartilham recursos de hardware entre vários usuários, e.g. sites e outros serviços de rede.

## 5.2 Funcionalidades não implementadas

O principal objetivo neste trabalho era apresentar uma implementação simples de expressões regulares, algumas funcionalidades comuns em implementações modernas foram propositalmente ignoradas. Enquanto algumas delas podem ser implementadas facilmente sem sacrificar complexidade de tempo, outras necessariamente tornariam o match exponencial. A seguir são comentadas algumas funcionalidades não implementadas.

**Classes de caracteres:** Para facilitar a escrita de expressões, muitas implementações disponibilizam conjuntos pré-definidos de caracteres. Exemplos incluem $\backslash w$, que inclui letras, números e underscore; ou $\backslash d$, que inclui todos os caracteres numéricos. Também deveria ser possível reconhecer classes customizadas, e.g. $[0-9a-fA-F]$,

para caracteres aceitáveis num número na base hexadecimal.

**Suporte a Unicode:** Uma implementação moderna de expressões regulares precisa dar suporte a strings Unicode. Apesar de não mudar a complexidade de execução do algoritmo, a implementação precisa ter cuidado com muitos detalhes, e.g. a definição do que é um caractere aceitável numa palavra ($\backslash w$) varia conforme a codificação.

**Captura de grupos:** A possibilidade de capturar grupos pode ser importante para extrair informações de strings. Por exemplo, para capturar números separados por vírgula, seria possível escrever a expressão $(\backslash d+),(\backslash d+)$. Assim, a primeira captura seria o primeiro número e a segunda o segundo número.

**Repetições contadas:** É util poder definir a repetição de um certo grupo de captura um número finito de vezes, e.g. a expressão $a\{2,4\}$ deve reconhece exatamente as strings $aa$, $aaa$ e $aaaa$. É uma funcionalidade de simples implementação usando autômatos, bastando por exemplo reescrever a expressão como $aaa?a?$. Esta reescrita, porém, pode ter um crescimento exponencial. Um exemplo é a expressão $a\{1000000\}$. Onde uma expressão com 10 caracteres geraria um autômato com no mínimo um milhão de estados.

***Zero-width assertions*:** Algumas vezes pode ser necessário condicionar o match de um grupo ao match de outras strings em volta da região do match principal. Exemplos incluem o match de fronteiras de palavra ($\backslash b$), que pode ser útil ao buscar por uma palavra inteira numa string. A expressão $\backslash bpalavra\backslash b$, por exemplo, procura por *palavra* na entrada desde que não seja uma substring de nenhuma outra palavra.

***Backreferences*:** Como já mencionado no Capítulo 4, *backreferences* são a capacidade de referenciar outros grupos de captura. Foi provado também que esta funcionalidade torna o match um problema NP-difícil.

## 5.3 Trabalhos Futuros

A implementação apresentada no Capítulo 3 abordou os pontos mais básicos do reconhecimento de expressões regulares. Devido à alta densidade de funcionalidades em implementações modernas de expressões regulares, uma nova implementação precisaria resolver outros problemas, antes de ser útil para a industria.

Nesta seção serão discutidas propostas de trabalhos futuros que podem dar continuidade à implementação apresentada, de forma a torná-la apta ao uso em programas comerciais.

**Reimplementar a máquina em C:** Uma das vantagens de descrever a expressão regular como instruções de uma máquina abstrata é a capacidade de implementá-la facilmente em qualquer linguagem. Durante os benchmarks, um dos pontos mais visíveis foi a desvantagem de uma implementação puramente em Python ao competir com outras implementações em C nos casos mais triviais. Uma proposta de novo trabalho seria reimplementar esta máquina de execução utilizando uma linguagem de mais baixo nível.

**Implementar funcionalidades comuns:** Certas funcionalidades presentes em implementações modernas permitem uma representação mais concisa de certas linguagens sem sacrificar a complexidade de tempo do algoritmo. Alguns exemplos notáveis incluem: captura de grupos (sem *backreferences*); alguns tipos de *zero-width assertions*, como \$ ou $\backslash b$.

**Realizar um mapeamento completo de funcionalides:** Ao longo do tempo, as funcionalidades em implementações expressões regulares começaram a divergir entre as diversas implementações desenvolvidas. Cada biblioteca implementa uma certa porção utilizando um certo algoritmo. Uma proposta de trabalho seria mapear todas essas funcionalidades, catalogando seu impacto na complexidade dos algoritmos que as implementam.

# Apêndice A
# Código-fonte da Biblioteca PyRex

```
# -*- coding: utf-8 -*-
from collections import deque
from functools import reduce

def rex(pattern):
    tokens = deque(pattern)

    def walk(chars):
        while tokens and tokens[0] in chars:
            yield tokens.popleft()

    def option():
        e = sequence()
        for token in walk('|'):
            e2 = sequence()
            e = [(1, len(e)+2)] + e + [(len(e2)+1,)] + e2
        return e

    def sequence():
        e = []
        while tokens and tokens[0] not in '|)':
            e += repetition()
        return e

    def repetition():
        e = primary()
        for token in walk('?*+'):
            if token in '+*': e = e + [(1, -len(e))]
            if token in '?*': e = [(1, len(e)+1)] + e
        return e

    def primary():
        token = tokens.popleft()
        if token == '.': return [None]
        if token == '(': return [option(), tokens.popleft()][0]
        if token not in '?*+)|': return [token]
        raise Exception('Not expected: "{}"'.format(token))

    e = option()
    if tokens:
        raise Exception('Not expected: "{}"'.format(''.join(tokens)))

    return Machine(e)

class Machine(object):
    def __init__(self, states):
        self.states = states
        self.n = len(states)

    def matcher(self, string):
        A, B, V = list(), list(), [-1]*len(self.states)

```

```
53          def addnext(start, i, j):
54              if j==self.n: return 1
55              if V[j] == i: return 0
56              V[j] = i
57
58              if isinstance(self.states[j], tuple):
59                  return sum(addnext(start, i, j+k) for k in self.states[j])
60
61              B.append((start, j))
62              return 0
63
64          def key(a): return (a[1]-a[0], -a[0]) if a else (0, 0)
65
66          answer = None
67          for i, c in enumerate(string):
68              addnext(i, i, 0)
69              yield i, answer, B
70
71              A, B = B, A
72              del B[:]
73
74              for start, j in A:
75                  if self.states[j] in (None, c) and addnext(start, i+1, j+1):
76                      answer = max(answer, (start, i+1), key=key)
77
78          yield len(string), answer, B
79
80      def match(self, string):
81          return reduce(lambda answer, s: s[1], self.matcher(string), None)
82
83      def source(self):
84          for s in self.states:
85              yield ('JUMP ' if isinstance(s, tuple) else 'CONSUME ') + str(s)
86          yield 'MATCH!'
87
88      def __repr__(self):
89          return '\n'.join('{:04d}: {}'.format(i, s) for i, s in enumerate(self.source()))
```

Listagem 10: Código-fonte da biblioteca PyRex

# Apêndice B
# Código-fonte do Visualizador

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function
import sys, pyrex

try: input = raw_input
except NameError: pass

def simulate(machine, string):
    for i, answer, state in machine.matcher(string):
        best = (string[answer[0]:answer[1]] + ' ' + str(answer) if answer else '<none>')
        print('Best answer: ' + best)
        print('Input: ' + string)
        print('       ' + i*' ' + '^' + ' (' + str(i) + ')')
        print()

        per_line = {item: str(start)+' >' for start, item in state}

        for j, line in enumerate(machine.source()):
            print('{:>5}{:04d}: {}'.format(per_line.get(j, ''), j, line))

        print('-----')
        input()

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print('usage: simulator.py <regex> <input>')
        sys.exit(1)

    print('-----')
    print('Matching pattern "{}" against input "{}"'.format(*sys.argv[1:]))
    print('-----')
    print()
    simulate(pyrex.rex(sys.argv[1]), sys.argv[2])
```

Listagem 11: Código-fonte do Visualizador

# Apêndice C
# Código-fonte dos Benchmarks

```
#!/bin/bash

echo '(a?a)+b' > input1.txt
echo 'n' > bench1.txt

for i in $(seq 0 2 44);
do
    eval printf 'a%0.s' {1..$i} >> input1.txt
    echo >> input1.txt
    echo $i >> bench1.txt
done

./bench_run input1.txt bench1.txt ${1:-16}
rm input1.txt
```

Listagem 12: Benchmark 1: (a?a)+b

```
#!/bin/bash

echo 'a*b' > input2.txt
echo 'n' > bench2.txt

for i in $(seq 0 20 500);
do
    eval printf 'a%0.s' {1..$i} >> input2.txt
    echo >> input2.txt
    echo $i >> bench2.txt
done

./bench_run input2.txt bench2.txt ${1:-16}
rm input2.txt
```

Listagem 13: Benchmark 2: a*b

```
#!/bin/bash

echo 'a*a*a*a*a*b' > input3.txt
echo 'n' > bench3.txt

for i in $(seq 0 5 125);
do
    eval printf 'a%0.s' {1..$i} >> input3.txt
    echo >> input3.txt
    echo $i >> bench3.txt
done

./bench_run input3.txt bench3.txt ${1:-16}
rm input3.txt
```

Listagem 14: Benchmark 3: a*a*a*a*a*b

```
#!/bin/bash

if [ $# -lt 3 ]; then
    echo 'bench <input> <output> <runs>'
    echo 'e.g.: bench input1.txt bench1.txt 20'
    exit 0
fi

function average {
    while read -a line ; do
        sum=0
        for ((i=0; i<${#line[@]}; i++)) ; do
            let sum+=line[i]
        done;
        echo $(bc -l <<< "$sum/$i/1000000000")
    done
}

RUNNERS=('pyrex' 'python' 'ruby' 'java')

for runner in "${RUNNERS[@]}"
do
    echo "Running ${runner}... "
    truncate result_${runner}.txt -s 0

    for i in $(seq 1 $3);
    do
        echo -n "    execution #$i... "
        TIMEFORMAT='%lU'
        time (
            mv result_${runner}.txt result_${runner}.tmp;
            "./bench_${runner}" < $1 | paste result_${runner}.tmp - > result_${runner}.txt;
            rm result_${runner}.tmp;
        )
    done

    mv $2 $2.tmp
    cat result_${runner}.txt | (echo ${runner}; average) | paste $2.tmp - > $2

    rm result_${runner}.txt
    rm $2.tmp
done
```

Listagem 15: Benchmark runner

```
#!/usr/bin/env python
from timeit import timeit
import os, sys
sys.path.append(os.path.join(os.path.dirname(__file__), '../pyrex'))
import pyrex

r = pyrex.rex(sys.stdin.readline())

for line in iter(sys.stdin.readline, ''):
    print int(timeit(lambda: r.match(line), number=1)*1e9)
```

Listagem 16: Benchmark com PyRex

```python
#!/usr/bin/env python
from timeit import timeit
import os, sys, re

r = re.compile(sys.stdin.readline())

for line in iter(sys.stdin.readline, ''):
    print int(timeit(lambda: r.match(line), number=1)*1e9)
```

Listagem 17: Benchmark com Python

```ruby
#!/usr/bin/env ruby
r = Regexp.new(readline.strip)

while true
    starttime = Time.now
    begin
        s = readline.strip
    rescue EOFError
        break
    end
    r.match(s)
    endtime = Time.now
    puts "#{((endtime - starttime)*1e9).to_i}"
end
```

Listagem 18: Benchmark com Ruby

```bash
#!/bin/bash

javac java_source.java
java java_source
rm java_source.class
```

Listagem 19: Benchmark com Java (compilador)

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.regex.Pattern;

public class java_source {
    public static void main(String... args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        Pattern pattern = Pattern.compile(reader.readLine());
        String input;
        while ((input = reader.readLine()) != null) {
            long start = System.nanoTime();
            pattern.matcher(input).matches();
            System.out.println(System.nanoTime() - start);
        }
    }
}
```

Listagem 20: Benchmark com Java (fonte)

# Apêndice D
# Tabelas dos Benchmarks

## D.1 Benchmark 1: (a?a)+b

| n | pyrex | python | ruby | java |
|---|---|---|---|---|
| 0 | $3.423 \cdot 10^{-5}$ | $6.556 \cdot 10^{-6}$ | $1.13 \cdot 10^{-5}$ | $1.655 \cdot 10^{-4}$ |
| 2 | $2.66 \cdot 10^{-5}$ | $2.667 \cdot 10^{-6}$ | $3.292 \cdot 10^{-6}$ | $8.307 \cdot 10^{-6}$ |
| 4 | $3.765 \cdot 10^{-5}$ | $2.563 \cdot 10^{-6}$ | $3.351 \cdot 10^{-6}$ | $1.79 \cdot 10^{-5}$ |
| 6 | $4.508 \cdot 10^{-5}$ | $3.502 \cdot 10^{-6}$ | $5.224 \cdot 10^{-6}$ | $3.784 \cdot 10^{-5}$ |
| 8 | $5.649 \cdot 10^{-5}$ | $6.556 \cdot 10^{-6}$ | $1.12 \cdot 10^{-5}$ | $9.311 \cdot 10^{-5}$ |
| 10 | $6.77 \cdot 10^{-5}$ | $1.357 \cdot 10^{-5}$ | $2.606 \cdot 10^{-5}$ | $2.388 \cdot 10^{-4}$ |
| 12 | $7.935 \cdot 10^{-5}$ | $3.28 \cdot 10^{-5}$ | $6.546 \cdot 10^{-5}$ | $6.313 \cdot 10^{-4}$ |
| 14 | $9.075 \cdot 10^{-5}$ | $8.252 \cdot 10^{-5}$ | $1.696 \cdot 10^{-4}$ | $1.671 \cdot 10^{-3}$ |
| 16 | $1.063 \cdot 10^{-4}$ | $2.136 \cdot 10^{-4}$ | $4.409 \cdot 10^{-4}$ | $3.996 \cdot 10^{-3}$ |
| 18 | $1.134 \cdot 10^{-4}$ | $5.639 \cdot 10^{-4}$ | $1.155 \cdot 10^{-3}$ | $4.195 \cdot 10^{-3}$ |
| 20 | $1.25 \cdot 10^{-4}$ | $1.543 \cdot 10^{-3}$ | $2.974 \cdot 10^{-3}$ | $9.629 \cdot 10^{-3}$ |
| 22 | $1.365 \cdot 10^{-4}$ | $3.89 \cdot 10^{-3}$ | $7.559 \cdot 10^{-3}$ | $2.498 \cdot 10^{-2}$ |
| 24 | $1.476 \cdot 10^{-4}$ | $9.988 \cdot 10^{-3}$ | $1.942 \cdot 10^{-2}$ | $6.565 \cdot 10^{-2}$ |
| 26 | $1.622 \cdot 10^{-4}$ | $2.544 \cdot 10^{-2}$ | $5.104 \cdot 10^{-2}$ | 0.136 |
| 28 | $1.743 \cdot 10^{-4}$ | $6.617 \cdot 10^{-2}$ | 0.133 | $3.358 \cdot 10^{-2}$ |
| 30 | $1.839 \cdot 10^{-4}$ | 0.173 | 0.349 | $8.771 \cdot 10^{-2}$ |
| 32 | $1.994 \cdot 10^{-4}$ | 0.454 | 0.913 | 0.227 |
| 34 | $2.078 \cdot 10^{-4}$ | 1.192 | 2.389 | 0.591 |
| 36 | $2.141 \cdot 10^{-4}$ | 3.121 | 6.257 | 1.537 |
| 38 | $2.089 \cdot 10^{-4}$ | 8.159 | 16.382 | 4.041 |
| 40 | $2.289 \cdot 10^{-4}$ | 21.345 | 42.788 | 10.606 |
| 42 | $2.293 \cdot 10^{-4}$ | 55.744 | 112.208 | 27.734 |
| 44 | $2.404 \cdot 10^{-4}$ | 146.132 | 294.275 | 72.533 |

## D.2 Benchmark 2: a*b

| n | pyrex | python | ruby | java |
|---|---|---|---|---|
| 0 | $2.894 \cdot 10^{-5}$ | $5.379 \cdot 10^{-6}$ | $7.961 \cdot 10^{-6}$ | $1.513 \cdot 10^{-4}$ |
| 20 | $1.213 \cdot 10^{-4}$ | $1.952 \cdot 10^{-6}$ | $2.685 \cdot 10^{-6}$ | $1.411 \cdot 10^{-5}$ |
| 40 | $2.151 \cdot 10^{-4}$ | $1.534 \cdot 10^{-6}$ | $2.218 \cdot 10^{-6}$ | $2.381 \cdot 10^{-5}$ |
| 60 | $3.142 \cdot 10^{-4}$ | $1.505 \cdot 10^{-6}$ | $2.015 \cdot 10^{-6}$ | $3.22 \cdot 10^{-5}$ |
| 80 | $4.098 \cdot 10^{-4}$ | $1.43 \cdot 10^{-6}$ | $2.055 \cdot 10^{-6}$ | $4.167 \cdot 10^{-5}$ |
| 100 | $5.041 \cdot 10^{-4}$ | $1.609 \cdot 10^{-6}$ | $2.089 \cdot 10^{-6}$ | $5.473 \cdot 10^{-5}$ |
| 120 | $6.025 \cdot 10^{-4}$ | $1.698 \cdot 10^{-6}$ | $2.218 \cdot 10^{-6}$ | $6.084 \cdot 10^{-5}$ |
| 140 | $6.47 \cdot 10^{-4}$ | $1.296 \cdot 10^{-6}$ | $2.278 \cdot 10^{-6}$ | $7.126 \cdot 10^{-5}$ |
| 160 | $7.377 \cdot 10^{-4}$ | $1.579 \cdot 10^{-6}$ | $2.512 \cdot 10^{-6}$ | $8.183 \cdot 10^{-5}$ |
| 180 | $8.278 \cdot 10^{-4}$ | $1.609 \cdot 10^{-6}$ | $2.729 \cdot 10^{-6}$ | $9.705 \cdot 10^{-5}$ |
| 200 | $9.845 \cdot 10^{-4}$ | $1.996 \cdot 10^{-6}$ | $2.647 \cdot 10^{-6}$ | $1.042 \cdot 10^{-4}$ |
| 220 | $1.097 \cdot 10^{-3}$ | $1.832 \cdot 10^{-6}$ | $2.763 \cdot 10^{-6}$ | $1.123 \cdot 10^{-4}$ |
| 240 | $1.186 \cdot 10^{-3}$ | $1.698 \cdot 10^{-6}$ | $4.804 \cdot 10^{-6}$ | $1.221 \cdot 10^{-4}$ |
| 260 | $1.276 \cdot 10^{-3}$ | $2.205 \cdot 10^{-6}$ | $3.028 \cdot 10^{-6}$ | $1.448 \cdot 10^{-4}$ |
| 280 | $1.362 \cdot 10^{-3}$ | $1.922 \cdot 10^{-6}$ | $3.03 \cdot 10^{-6}$ | $1.429 \cdot 10^{-4}$ |
| 300 | $1.353 \cdot 10^{-3}$ | $1.892 \cdot 10^{-6}$ | $3.204 \cdot 10^{-6}$ | $1.538 \cdot 10^{-4}$ |
| 320 | $1.436 \cdot 10^{-3}$ | $1.937 \cdot 10^{-6}$ | $3.248 \cdot 10^{-6}$ | $1.635 \cdot 10^{-4}$ |
| 340 | $1.531 \cdot 10^{-3}$ | $3.203 \cdot 10^{-6}$ | $1.089 \cdot 10^{-5}$ | $1.742 \cdot 10^{-4}$ |
| 360 | $1.608 \cdot 10^{-3}$ | $1.862 \cdot 10^{-6}$ | $3.554 \cdot 10^{-6}$ | $1.868 \cdot 10^{-4}$ |
| 380 | $1.692 \cdot 10^{-3}$ | $2.19 \cdot 10^{-6}$ | $3.564 \cdot 10^{-6}$ | $1.967 \cdot 10^{-4}$ |
| 400 | $1.783 \cdot 10^{-3}$ | $1.892 \cdot 10^{-6}$ | $3.683 \cdot 10^{-6}$ | $2.309 \cdot 10^{-4}$ |
| 420 | $1.862 \cdot 10^{-3}$ | $2.026 \cdot 10^{-6}$ | $3.787 \cdot 10^{-6}$ | $2.23 \cdot 10^{-4}$ |
| 440 | $1.954 \cdot 10^{-3}$ | $2.115 \cdot 10^{-6}$ | $5.229 \cdot 10^{-6}$ | $2.428 \cdot 10^{-4}$ |
| 460 | $2.038 \cdot 10^{-3}$ | $2.264 \cdot 10^{-6}$ | $3.96 \cdot 10^{-6}$ | $2.448 \cdot 10^{-4}$ |
| 480 | $2.137 \cdot 10^{-3}$ | $2.264 \cdot 10^{-6}$ | $4.04 \cdot 10^{-6}$ | $2.508 \cdot 10^{-4}$ |
| 500 | $2.224 \cdot 10^{-3}$ | $2.354 \cdot 10^{-6}$ | $5.153 \cdot 10^{-6}$ | $2.63 \cdot 10^{-4}$ |

## D.3 Benchmark 3: a*a*a*a*a*b

| n | pyrex | python | ruby | java |
|---|---|---|---|---|
| 0 | $7.284 \cdot 10^{-5}$ | $6.124 \cdot 10^{-6}$ | $1.091 \cdot 10^{-5}$ | $1.689 \cdot 10^{-4}$ |
| 5 | $1.255 \cdot 10^{-4}$ | $6.184 \cdot 10^{-6}$ | $2.516 \cdot 10^{-6}$ | $1.609 \cdot 10^{-4}$ |
| 10 | $2.294 \cdot 10^{-4}$ | $2.772 \cdot 10^{-5}$ | $1.759 \cdot 10^{-6}$ | $1.722 \cdot 10^{-3}$ |
| 15 | $3.342 \cdot 10^{-4}$ | $9.534 \cdot 10^{-5}$ | $1.56 \cdot 10^{-6}$ | $6.532 \cdot 10^{-3}$ |
| 20 | $4.405 \cdot 10^{-4}$ | $2.618 \cdot 10^{-4}$ | $1.56 \cdot 10^{-6}$ | $1.037 \cdot 10^{-2}$ |
| 25 | $5.413 \cdot 10^{-4}$ | $6.071 \cdot 10^{-4}$ | $1.981 \cdot 10^{-6}$ | $1.257 \cdot 10^{-3}$ |
| 30 | $6.406 \cdot 10^{-4}$ | $1.518 \cdot 10^{-3}$ | $1.82 \cdot 10^{-6}$ | $2.746 \cdot 10^{-3}$ |
| 35 | $6.87 \cdot 10^{-4}$ | $2.369 \cdot 10^{-3}$ | $1.797 \cdot 10^{-6}$ | $5.401 \cdot 10^{-3}$ |
| 40 | $7.894 \cdot 10^{-4}$ | $4.067 \cdot 10^{-3}$ | $1.818 \cdot 10^{-6}$ | $9.926 \cdot 10^{-3}$ |
| 45 | $8.842 \cdot 10^{-4}$ | $6.509 \cdot 10^{-3}$ | $1.817 \cdot 10^{-6}$ | $1.695 \cdot 10^{-2}$ |
| 50 | $9.88 \cdot 10^{-4}$ | $1.033 \cdot 10^{-2}$ | $1.842 \cdot 10^{-6}$ | $2.721 \cdot 10^{-2}$ |
| 55 | $1.065 \cdot 10^{-3}$ | $1.561 \cdot 10^{-2}$ | $1.853 \cdot 10^{-6}$ | $4.229 \cdot 10^{-2}$ |
| 60 | $1.122 \cdot 10^{-3}$ | $2.301 \cdot 10^{-2}$ | $1.996 \cdot 10^{-6}$ | $6.36 \cdot 10^{-2}$ |
| 65 | $1.146 \cdot 10^{-3}$ | $3.297 \cdot 10^{-2}$ | $1.932 \cdot 10^{-6}$ | 0.14 |
| 70 | $1.244 \cdot 10^{-3}$ | $4.606 \cdot 10^{-2}$ | $1.943 \cdot 10^{-6}$ | 0.245 |
| 75 | $1.303 \cdot 10^{-3}$ | $6.32 \cdot 10^{-2}$ | $1.959 \cdot 10^{-6}$ | 0.336 |
| 80 | $1.386 \cdot 10^{-3}$ | $8.495 \cdot 10^{-2}$ | $1.971 \cdot 10^{-6}$ | 0.457 |
| 85 | $1.481 \cdot 10^{-3}$ | 0.112 | $1.102 \cdot 10^{-5}$ | 0.612 |
| 90 | $1.55 \cdot 10^{-3}$ | 0.146 | $2.285 \cdot 10^{-6}$ | 0.806 |
| 95 | $1.642 \cdot 10^{-3}$ | 0.188 | $2.152 \cdot 10^{-6}$ | 1.046 |
| 100 | $1.718 \cdot 10^{-3}$ | 0.238 | $3.456 \cdot 10^{-6}$ | 1.339 |
| 105 | $1.797 \cdot 10^{-3}$ | 0.299 | $2.196 \cdot 10^{-6}$ | 1.698 |
| 110 | $1.894 \cdot 10^{-3}$ | 0.372 | $2.157 \cdot 10^{-6}$ | 2.132 |
| 115 | $1.975 \cdot 10^{-3}$ | 0.458 | $2.157 \cdot 10^{-6}$ | 2.637 |
| 120 | $2.065 \cdot 10^{-3}$ | 0.559 | $2.223 \cdot 10^{-6}$ | 3.242 |
| 125 | $2.144 \cdot 10^{-3}$ | 0.676 | $2.239 \cdot 10^{-6}$ | 3.957 |

# Referências

CHOMSKY, N. I.r.e. transactions on information theory. *Bulletin of Mathematical Biophysics*, v. 2, p. 113–124, 1957.

COOK, S. A. The complexity of theorem-proving procedures. In: ACM. *Proceedings of the third annual ACM symposium on Theory of computing*. [S.l.], 1971. p. 151–158.

COX, R. *Regular Expression Matching Can Be Simple And Fast*. 2007. Disponível em: <http://swtch.com/~rsc/regexp/regexp1.html>. Acesso em: 17 mai. 2013.

KARP, R. M. *Reducibility among combinatorial problems*. [S.l.]: Springer, 1972.

KIRRAGE, J.; RATHNAYAKE, A.; THIELECKE, H. Static analysis for regular expression denial-of-service attacks. *Springer LNCS*, n. 7873, 2013.

KLEENE, S. C. Representation of events in nerve nets and finite automata. *In Automata Studies, Ann. Math. Stud.*, Princeton U. Press, Princeton, N.J., n. 34, p. 3–41, 1956.

LEVIN, L. A. Universal sorting problems. *Problems of Information Transmission*, v. 9, n. 3, p. 265–266, 1973.

MCCULLOCH, W.; PITTS, W. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, v. 5, p. 127–147, 1943.

POPOV, N. *The true power of regular expressions*. 2012. Disponível em: <http://nikic-.github.io/2012/06/15/The-true-power-of-regular-expressions.html>. Acesso em: 17 mai. 2013.

RABIN, M. O.; SCOTT, D. Finite automata and their decision problems. *IBM journal of research and development*, IBM, v. 3, n. 2, p. 114–125, 1959.

RUOHONEN, K. Formal languages. *lecture notes for the TUT course "Formaalit kielet"*, 2009.

THOMPSON, K. Regular expression search algorithm. *Communications of the ACM*, New York, v. 11, n. 6, p. 419–422, jun. 1968.

VELDHUIZEN, T. L. C++ templates are turing complete. *Available at citeseer. ist. psu. edu/581150. html*, Indiana University Computer Science, 2003.

WEIDMAN, A. *Regular expression Denial of Service - ReDoS*. 2010. Disponível em: <https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS>. Acesso em: 17 mai. 2013.