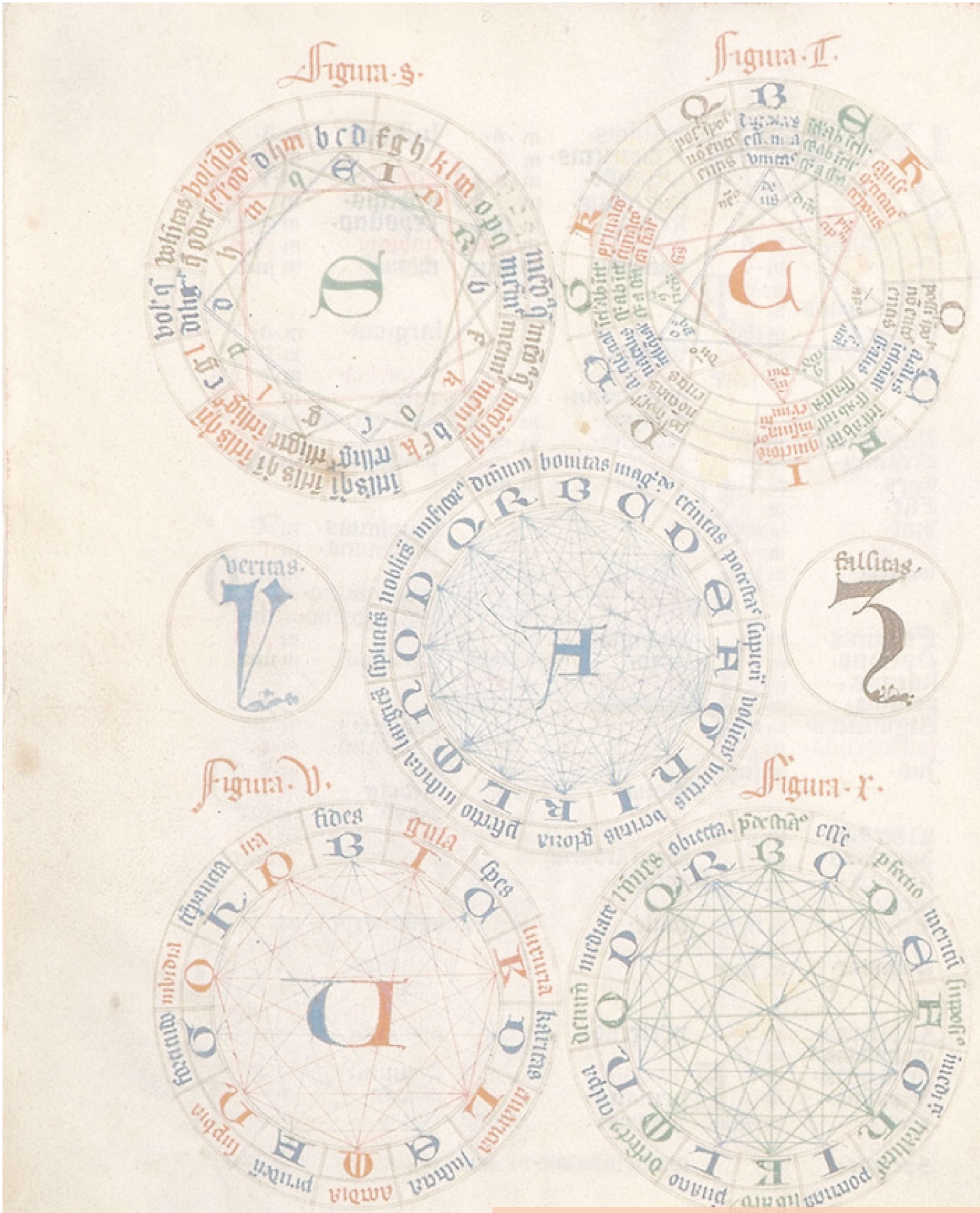
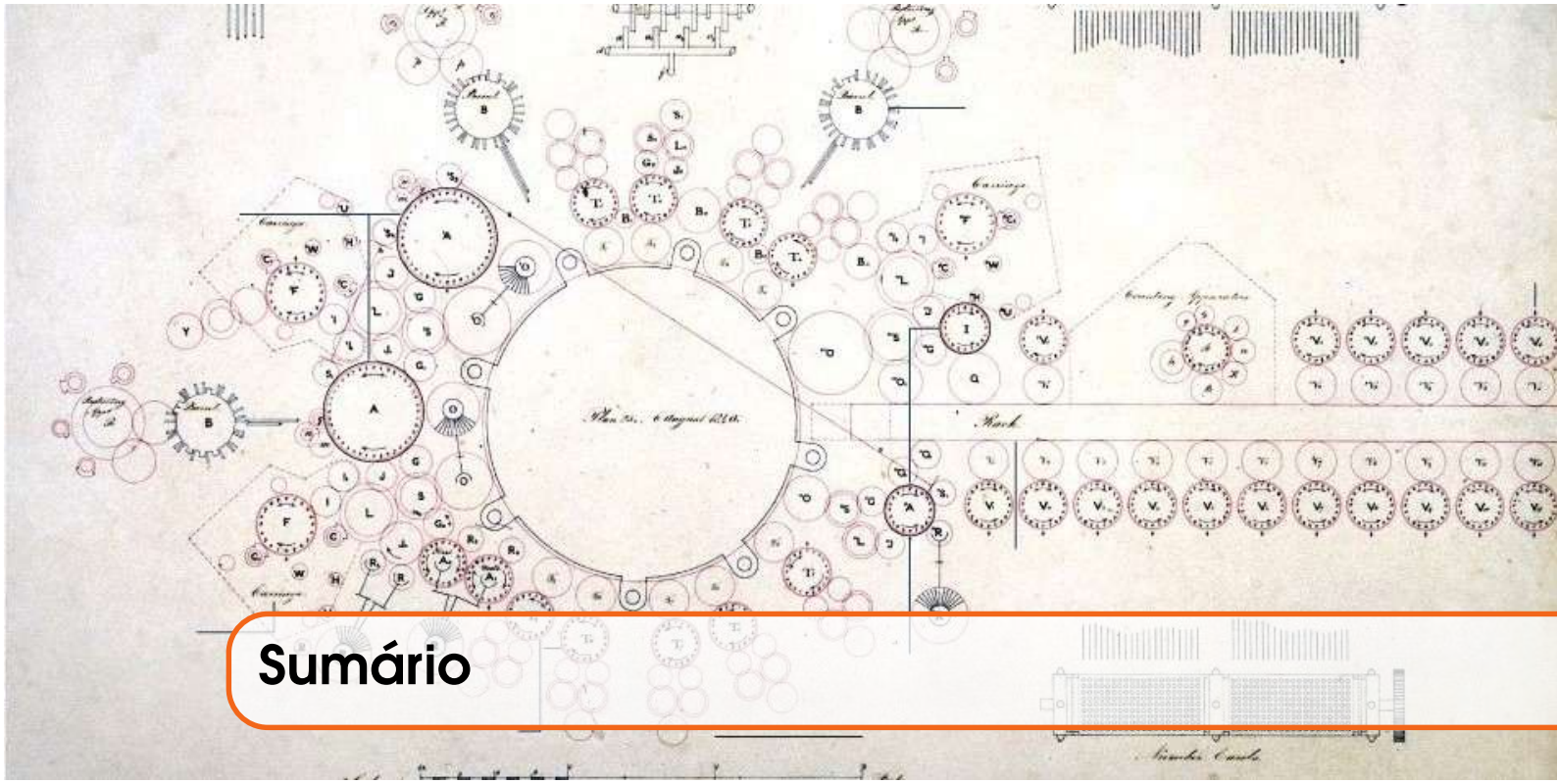


Autômatos, computabilidade e complexidade computacional

Murilo V. G. da Silva

AUTÔMATOS, COMPUTABILIDADE E COMPLEXIDADE COMPUTACIONAL
© MURILO VICENTE GONÇALVES DA SILVA 2017 – 2019

Este texto está licenciado sob a Licença *Attribution-NonCommercial 3.0 Unported License (the "License")* da *Creative Commons*. Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite `http://creativecommons.org/licenses/by-nc/3.0`.



Sumário

1 Prólogo 7

1.1 O que é computação? 7

1.2 Algoritmos, problemas e computadores 8

I Parte 1: Teoria de linguagens e autômatos

2 Alfabetos, Strings e Linguagens 15

2.1 Alfabetos e strings 15

2.1.1 Exercícios 18

2.2 Linguagens 18

2.2.1 Exercícios 21

2.2.2 Operações com linguagens 21

3 Autômatos e Linguagens Regulares 23

3.1 Autômatos Finitos Determinísticos (AFDs) 23

3.1.1 Modelando matematicamente autômatos 24

3.1.2 Aceitação e rejeição de strings: ideia intuitiva 26

3.1.3 Aceitação e rejeição de strings: definição formal 27

3.1.4 Exercícios 28

3.2 Autômatos Finitos não Determinísticos (AFNs) 28

3.2.1 Definição formal para autômatos finitos não determiníticos 30

3.2.2 Aceitação e rejeição de strings por AFNs 33

3.2.3 Exercícios 33

3.3 Equivalência entre AFDs e AFNs 34
3.3.1 Algoritmo de construção de conjuntos . . . 34
3.3.2 Algoritmo de construção de conjuntos: versão melhorada . . . 35
3.3.3 Exercícios . . . 37
3.4 Autômatos Finitos não Determinísticos com transições ε 37
3.5 Equivalência entre AFDs e ε-AFNs 40
3.5.1 Exercícios . . . 41
3.6 Expressões Regulares (ERs) 42
3.6.1 Expressões regulares para linguagens de autômatos . . . 44
3.6.2 Autômatos para linguagens de expressões regulares . . . 47
3.6.3 Exercícios . . . 50

4 Para além das Linguagens Regulares . . . 53
4.1 O Lema do Bombeamento para Linguagens Regulares 53
4.2 Autômato com Pilha (AP) 56
4.2.1 O modelo matemático para autômatos com pilha . . . 56
4.2.2 Computação com Autômatos com Pilha . . . 58
4.2.3 Aceitação por pilha vazia . . . 60
4.2.4 Autômatos com pilha determinísticos (APDs) . . . 61
4.3 Gramáticas Livre de Contexto 61
4.3.1 Definição formal de gramáticas livre de contexto . . . 61
4.3.2 Derivações de uma gramática . . . 62
4.3.3 Derivação mais a direita e mais a esquerda . . . 63
4.3.4 Árvores de análise sintática . . . 64
4.3.5 Ambiguidade de Gramáticas . . . 64
4.3.6 Equivalência entre APs e gramáticas livre de contexto . . . 64
4.3.7 APDs e ambiguidade de gramáticas . . . 64
4.3.8 Exercícios . . . 65

II Parte 2: Máquinas de Turing e Computabilidade

5 A Máquina de Turing . . . 69

6 A Tese de Church-Turing . . . 71

7 Computabilidade . . . 73

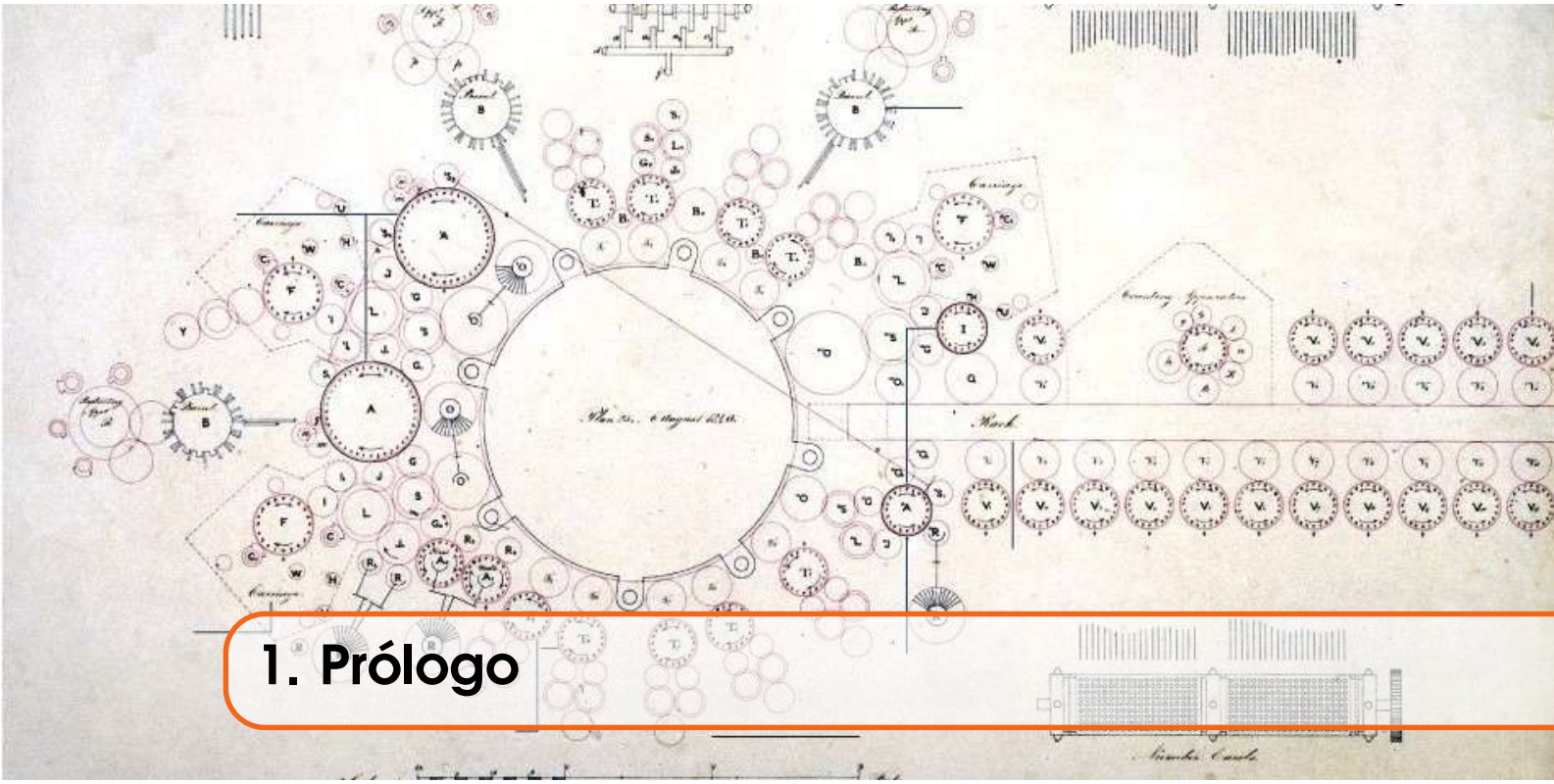
III Parte 3: Complexidade Computacional

8 Complexidade de Tempo e Espaço . . . 77

9 A classe NP . . . 79

10 NP-completude . . . 81

Bibliografia 83
Livros 83



1. Prólogo

1.1 O que é computação?

A maioria de nós tem uma noção intuitiva do que é um algoritmo ou que é um computador. Estes dois conceitos são tão corriqueiros que é comum não pararmos para pensar a fundo sobre eles. Entretanto, como veremos neste livro, quando tentamos responder perguntas mais profundas a respeito de computação, estas nossas noções intuitivas sobre algoritmos e computadores não nos são suficientes. Por exemplo, será que existe algum problema para o qual não exista nenhum algoritmo eficiente que o resolva? Ou mesmo, será que existe algum problema para o qual não exista nenhum algoritmo que o resolva (independente de ser ou não eficiente)? Um dos objetivos deste texto é apresentar conceitos fundamentais da computação de maneira matematicamente precisa e clara e apresentar respostas para perguntas como estas.

Um dos primeiros cuidados que precisamos tomar é pensar em algoritmos e computadores sem nos atermos à artefatos tecnológicos como linguagens programação modernas, laptops ou smartphones. A ideia é começar a enxergar computação como uma ciência que transcende tais tecnologias. Seria possível estudar ciência da computação sem nos preocuparmos com artefatos tecnológicos? Um bom ponto de partida para entendermos que isso é possível, é a observação de que isso, de fato, já tem sido feito a muito tempo. De fato, o próprio conceito de *informação*, algo intimamente ligado à computação, já era estudado antes de termos computadores digitais.

> O CONCEITO DE INFORMAÇÃO NA CIÊNCIA
>
> A ideia de informação, que é algo natural e antigo, começou a entrar de maneira mais regular no vocabulário de alguns cientistas no final do século XIX, especialmente no vocabulário dos físicos, quando estes começaram a esbarrar em conceitos como entropia e termodinâmica.
>
> A partir daí, uma série de descobertas científicas do último século, como por exemplo a descoberta das leis da mecânica quântica (leis em que a ideia de *informação* quântica é central), ou a descoberta da molécula de DNA (molécula que está intrinsicamente ligada a ideia de *informação* genética), começaram a solidificar a intuição que temos hoje de que informação é algo fundamental e que permeia o mundo natural em diferentes níveis de análise.

Com a popularização dos computadores na segunda metade do século XX, conceitos como informação e computação tornaram-se populares. Isso pode ser positivo por um lado, mas, por outro, acaba nos enviesando e nos induz a associar conceitos como informação e computação unicamente à artefatos tecnológicos, que é algo que queremos evitar neste livro. Para ilustrar o que queremos dizer, observe que não é incomum que em situações em que estamos descrevendo processos computacionais no mundo natural, como moléculas de DNA *armazenando informação*, ou cérebros *processando informação*, nós venhamos a pensar que estamos meramente usando metáforas inspiradas pela nossa tecnologia corrente. Embora exista, de fato, um contingência histórica que nos faz criar tais associações[1], isto pode obscurecer o fato de que os conceitos de informação e de computação estão presentes no mundo (na matemática, na natureza, etc), e podem ser estudados como objetos em si, independentes de qualquer contingência tecnológica. Para isso, uma das tarefas essenciais que empreendemos neste texto é tal estudo, de maneira esclarecer o que são, fundamentalmente, algoritmos e computadores.

Além de algoritmos e computadores, um outro conceito que normalmente usamos de maneira intuitiva e que iremos tornar preciso neste texto é o conceito de *problema computacional*. Por exemplo, considere o problema de testar se um dado número é primo, ou o problema de testar se existe um caminho ligando dois vértices em um grafo. Em nosso estudo de teoria da computação uma questão que surgirá naturalmente neste contexto é definir exatamente o que é, de *maneira genérica*, um problema computacional. O quadro abaixo esclarece isso um pouco.

> DEFINIÇÕES GENÉRICAS E DEFINIÇÕES ESPECÍFICAS
>
> No Capítulo 2 iremos desenvolver algumas ferramentas matemáticas que nos ajudarão a dar definições precisas para o conceito de problema computacional. Entretanto, antes de termos estas ferramentas em mãos, vamos esclarecer aonde estamos querendo chegar com isso.
>
> Vamos usar uma analogia com algo bastante familiar. Pense na função $f(x) = x^2$ e na função $g(x) = \log x$. Temos aqui dois casos *específicos* e matematicamente bem definidos de funções. Entretanto, sabemos que é perfeitamente possível sermos mais *genéricos* e fornecermos uma definição para o conceito *genérico* de função[1] tal que as funções $f(x)$ e $g(x)$ anteriores são casos particulares (ou instanciações) de tal definição. Ou seja, podemos falar de objetos matemáticos específicos, como $f(x) = x^2$ e $g(x) = \log x$, mas também do objeto matemático genérico que chamamos de *função*. A analogia que queremos fazer aqui é que é possível dar uma definição genérica para o conceito de problema computacional, tal que os problemas específicos, como testar primalidade de números ou testar a conectividade de um grafo, sejam *especificações* do conceito geral de problema computacional. Da mesma forma, neste curso vamos lidar não somente com algoritmos específicos, mas também com uma *definição matemática genérica para o próprio conceito de algoritmo*. Em nossos estudos, a flexibilidade mental de entender esta distinção entre casos gerais e casos particulares é bastante importante.
>
> [1] O leitor aqui pode estar um pouco enferrujado e não lembrar exatamente qual é definição matemática de uma função. Na verdade, isso não importa tanto aqui, pois o que queremos enfatizar neste ponto é que uma função, enquanto objeto matemático, *pode* ser precisamente definida. Curiosamente, funções podem também ser definidas como casos particulares objetos matemáticos ainda mais genéricos, conhecidos como relações.

1.2 Algoritmos, problemas e computadores

Uma vez que temos em mãos definições precisas para algoritmos e para problemas computacionais, uma pergunta surge naturalmente é a seguinte: Será que existe algum problema para o

[1] Em particular, é preciso ter clareza filosófica em temas como este (no Capítulo 6 apresentamos uma breve discussão em torno de questões de interesse filosófico).

qual não exista nenhum algoritmo que o resolva? Note que não estamos falando de problemas para os quais nós, hoje, ainda não conhecemos algoritmos que os solucionem, mas que no futuro os venhamos a descobrir. O que estamos perguntando aqui é se existem problemas que não podem ser resolvidos por nenhum algoritmo (dentre os infinitos possíveis). Um dos resultados que veremos neste livro é que, de fato, existem problemas assim. Uma das razões para se estudar teoria da computação é a possibilidade de termos *certeza matemática* de tais afirmações a respeito das bases da ciência da computação.

No Capítulo 5, veremos uma descrição matemática genérica para o conceito de algoritmo, de maneira que, qualquer algoritmo, com o naturalmente concebemos, acaba sendo um caso particular desta definição genérica. Mas, antes disso, no Capítulo 2, veremos definições matemáticas simplificadas de algoritmos (definições que "capturam" apenas um subconjunto de todos os possíveis algoritmos). Há duas razões para vermos estes modelos matemáticos simplificados: (1) Uma razão é pedagógica, de forma que iremos primeiro ganhamos intuição estudando modelos mais simples, para somente depois ver a definição precisa para o conceito de algoritmo do Capítulo 5; (2) A segunda razão é que estes modelos são úteis não apenas em teoria da computação, mas também em outras áreas da computação (e.g., processamento de texto, construção de compiladores, etc).

Refletindo um pouco: limite tecnológico ou princípio fundamental?

Na discussão anterior mencionamos a possibilidade de que existam problemas insolúveis. Mas a existência destes problemas não refletiria apenas uma limitação para o que atualmente entendemos por computadores e algoritmos? Ou seja, será que tais problemas não seriam apenas insolúveis apenas atualmente? Estes problemas não poderiam ser resolvidos no futuro por computadores "exóticos"descritos por modelos matemáticos correspondendo a leis da física que ainda não conhecemos? Ou a insolubilidade de certos problemas seria algo mais fundamental? Até que ponto podemos descartar a possibilidade de que seja possível construir objetos exóticos que poderíamos usar como computadores para resolver estes problemas que os computadores atuais não resolvem?

O consenso científico atual, chamado de Tese de Church-Turing (TCT), é que qualquer problema solúvel poderia ser resolvido, *em princípio*, por um computador como concebemos hoje. Este tipo de questão, é claro, como qualquer questão científica corrente, é passível de debate. Entretanto, a maioria das propostas que aparecem na literatura questionando a TCT, embora matematicamente bem definidas, tendem a cair, em última análise, em variações de ideias que podem ser inviáveis de ser realizadas fisicamente[a]. O Capítulo 6 deste livro é dedicado a discussão da Tese de Church-Turing.

Entretanto, isso tudo não exclui a possibilidade de que possamos construir computadores *fundamentalmente* muito mais *eficientes* que os atuais. A pesquisa em computação quântica, por exemplo, investiga precisamente a possibilidade de construção de computadores *exponencialmente* mais rápidos do que os atuais na resolução de *alguns* problemas.

[a] Um caso comum se refere a um modelo matemático que requer a existência de "oráculos mágicos" que executam passos computacionais sem nenhuma explicação. Um outro caso comum é o de um modelo matemático que requer a realização de computação "verdadeiramente analógica", uma ideia que parece contradizer alguns pressupostos da física contemporânea.

Embora um curso de teoria da computação seja um curso de matemática, e não de física, é importante observarmos que objetos abstratos estudados em computação, como algoritmos e informação, se manifestam, de alguma forma no mundo físico[2].

O seu laptop rodando Windows é o caso mais óbvio do que podemos entender como um processo computacional ocorrendo em um meio físico. Entretanto, tal caso não é único, e este é exatamente o ponto que queremos ressaltar aqui. O cálculo do logaritmo de um número ocorrendo em uma máquina com engrenagens e graxa, como a máquina de Charles Babbage, do século XIX, também é um outro exemplo de um processo computacional ocorrendo no meio físico. Poderíamos também pensar no cérebro de um primata processando sinais elétricos sensoriais, ou moléculas de DNA executando rotinas biológicas, ou mesmo em um punhado de átomos, elétrons e fótons interagindo sistematicamente em algum protótipo de computador quântico, de maneira que a manipulação da informação quântica de destas partículas fazem parte do processo de execução de um algoritmo quântico. A Figura 1.1 ilustra tais exemplos. O quadro "*Computação: instanciando objetos abstratos em objetos físicos*", abaixo, descreve tais casos de maneira unificada.

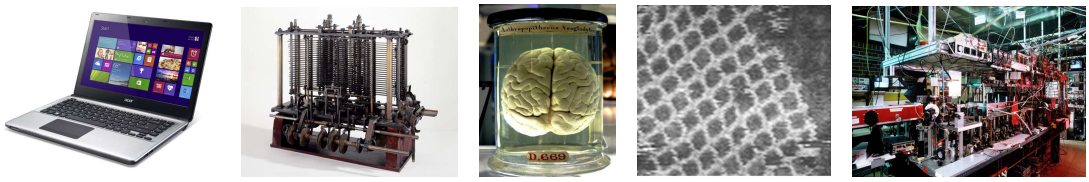


Figure 1.1: Computação ocorrendo em diferentes meios físicos: circuitos eletrônicos, engrenagens mecânicas, estruturas biológicas, moléculas de DNA e partículas subatômicas.

> COMPUTAÇÃO: INSTANCIANDO OBJETOS ABSTRATOS EM OBJETOS FÍSICOS
>
> Podemos pensar em um computador como uma maneira de instanciar objetos abstratos e o conjunto de possíveis relacionamentos matemáticos destes objetos abstratos (i.e., informação e algoritmos) em objetos físicos e o conjunto dos possíveis graus de movimento (ou graus de liberdade de movimento) destes objetos físicos.

O objetivo da teoria da computação é prover modelos matemáticos para processos computacionais que sejam *independentes do substrato físico* em que a computação esteja ocorrendo. O nosso objetivo principal é ter um modelo matemático que seja independente do substrato físico, mas que, ao mesmo tempo, seja abrangente e realista o suficiente para capturar qualquer possível computação ocorrendo em qualquer meio físico possível. Por que isso seria o ideal? Porque uma vez que temos um modelo matemático com estas características, bastaríamos nos focar neste modelo para poder saber o que pode e o que não pode ser *efetivamente* computado.

[2] O ponto de contato entre a natureza matemática da teoria da computação e questões de natureza empírica é bastante sutil e será discutido com mais cuidado no Capítulo 6. Sem entrar em uma discussão filosófica mais profunda, o ponto que queremos chamar atenção aqui é simplesmente que existe conexão entre modelos matemáticos estudados em computação e objetos do "mundo físico".

Claramente este objetivo parece ser bastante ambicioso, pois estamos atrás de um modelo matemático que possa descrever todo e qualquer tipo de manipulação de informação em qualquer tipo de substrato físico! Entretanto, veremos que na década de 1930 foi concebido um modelo que, apesar de ser razoavelmente simples, se propõe a ter tais propriedades[3]. Este modelo é conhecido como *Máquina de Turing*. A tese científica que afirma que Máquinas de Turing atingem tal nível de generalidade em termos de poder representar qualquer processo computacional (i.e., processos para manipular/tranformar informação) é conhecida hoje em dia como *Tese de Church-Turing*[4].

Os computadores que usamos no dia a dia são objetos precisos e sabemos rigorosamente como eles funcionam (afinal, fomos nós quem os construímos!) e, portanto, temos modelos matemáticos que os descrevem com exatidão. Veremos que os modelos matemáticos que descrevem nossos computadores atuais são equivalentes a certas Máquinas de Turing conhecidas como Máquinas de Turing Universais. A propriedade essencial de tais máquinas universais é que elas podem simular[5] qualquer outra Máquina de Turing, e, portanto, de acordo com a TCT, poderiam simular qualquer outro processo computacional com alguma correspondência em algum substrato físico. Isso significa que, pelo menos em princípio, não existem instanciações físicas que possam resolver problemas que não possam ser resolvidos por computadores atuais devidamente programados e com as condições de eficiência e de memória adequadas[6]. A definição matemática de uma Máquina de Turing Universal é o modelo matemático para o que chamamos de *computador*.

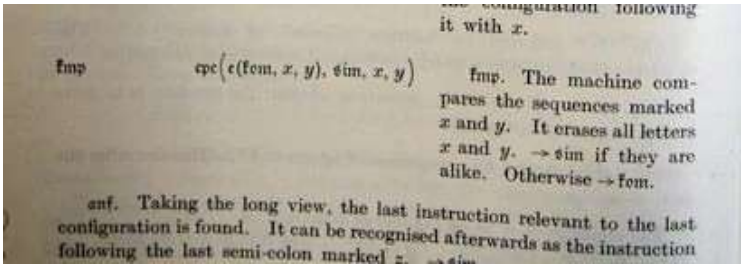
[3]Observe que empreendimentos semelhantes também ocorrem em outras ciências. Por exemplo, na física, algumas equações se propõe a descrever "todo e qualquer movimento", "todo e qualquer estado de um sistema", e assim por diante.

[4]Para ser mais preciso, atualmente há duas interpretações para o que se entende por Tese de Church-Turing (TCT). Estas duas interpretações serão vistas com cuidado no Capítulo 6. Uma interpretação é que a TCT é uma *definição matemática* e a outra, que é a que nos referimos neste parágrafo, é que ela é uma *afirmação sobre a realidade física*. Um ponto fundamental desta segunda versão da TCT é que, na possibilidade dela ser falsa, em princípio existe meios de refutá-la. Isso é relevante para hipóteses no contexto de ciências empíricas. Veremos também quais foram as razões que foram tornando esta tese mais e mais sólida com o tempo.

[5]Aqui a palavra "simular" tem um significado matematicamente preciso que, a grosso modo, significa uma certa *correspondência de um para um* entre dois modelos matemáticos.

[6]Claramente afirmações assim podem gerar discussões de natureza filosófica (com direito a bastante confusão e imprecisão em muitos casos!)

O fato de que existem Máquinas de Turing Universais, que é um dos resultados mais importantes no artigo original escrito por Alan Turing (a Figura 1.2 mostra um fragmento de tal artigo), é conhecido como *universalidade* das Máquinas de Turing.



configuration following it with x.

fmp cpc(c(fom, x, y), sim, x, y)

fmp. The machine compares the sequences marked x and y. It erases all letters x and y. → sim if they are alike. Otherwise → fom.

anf. Taking the long view, the last instruction relevant to the last configuration is found. It can be recognised afterwards as the instruction following the last semi-colon marked z. → sim

Figure 1.2: Um trecho do famoso artigo publicado em 1936 por Alan Turing, entitulado *On computable numbers, with an application to the Entscheidungsproblem*. Neste artigo é que foram estabelecidas as bases de toda ciência da computação.

A Tese de Church-Turing e a Universalidade da Computação

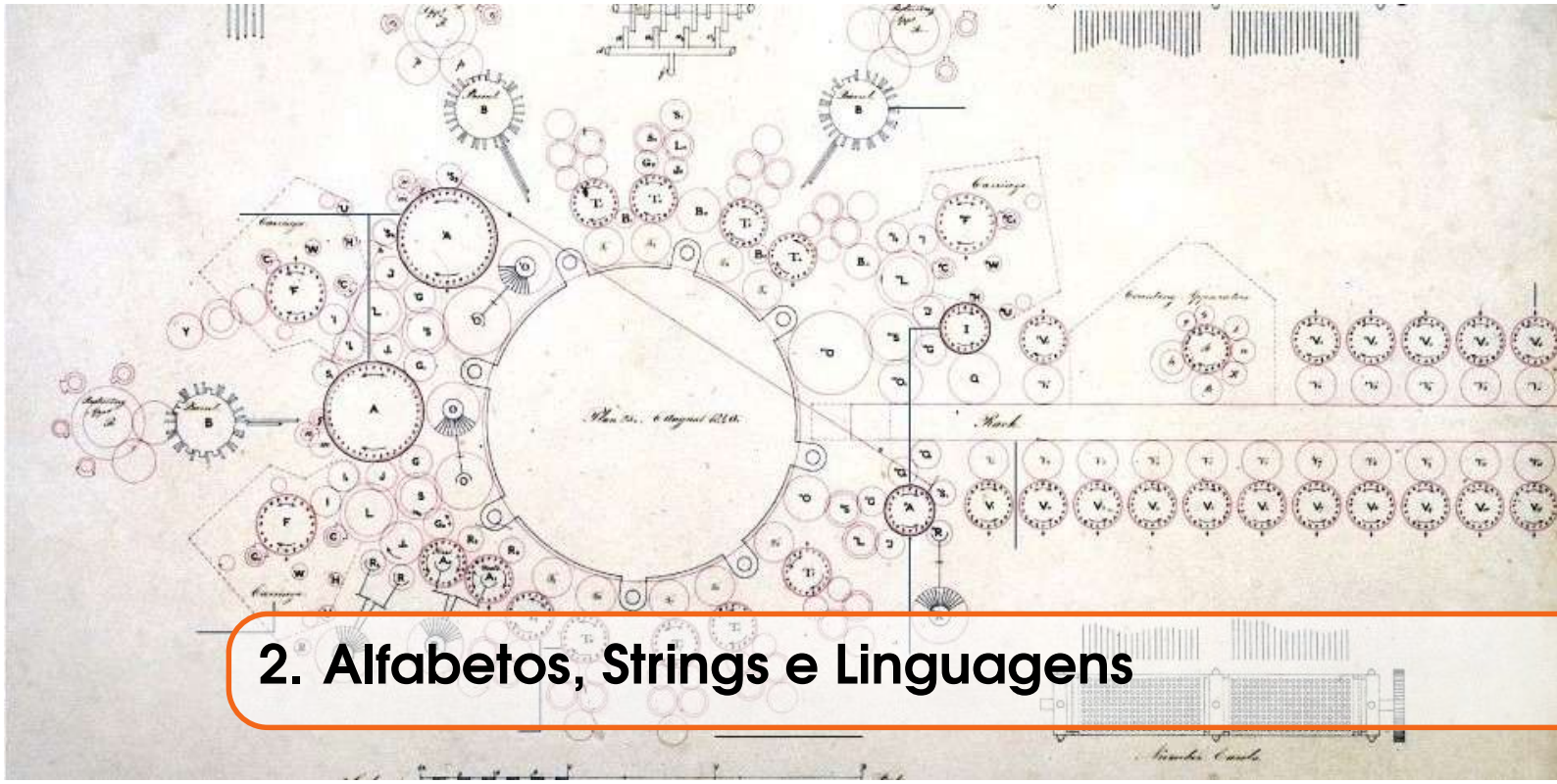
"A lição que tomamos da universalidade da computação é que podemos construir um objeto físico, que chamamos de computador, que pode simular qualquer outro processo físico, e o conjunto de todos os possíveis movimentos deste computador, definido pelo conjunto de todos os possíveis programas concebíveis, está em ***correspondência de um para um*** *com o conjunto de todos os possíveis movimentos de qualquer outro objeto físico concebível".* – David Deutsch

Parte 1: Teoria de linguagens e autômatos

2 Alfabetos, Strings e Linguagens 15
2.1 Alfabetos e strings
2.2 Linguagens

3 Autômatos e Linguagens Regulares . . 23
3.1 Autômatos Finitos Determinísticos (AFDs)
3.2 Autômatos Finitos não Determinísticos (AFNs)
3.3 Equivalência entre AFDs e AFNs
3.4 Autômatos Finitos não Determinísticos com transições $\boldsymbol{\varepsilon}$
3.5 Equivalência entre AFDs e ε-AFNs
3.6 Expressões Regulares (ERs)

4 Para além das Linguagens Regulares . 53
4.1 O Lema do Bombeamento para Linguagens Regulares
4.2 Autômato com Pilha (AP)
4.3 Gramáticas Livre de Contexto



2. Alfabetos, Strings e Linguagens

2.1 Alfabetos e strings

Algoritmos, computadores e problemas computacionais são objetos matemáticos complexos. Nosso primeiro passo será definir os conjuntos elementares de símbolos que usaremos para construir tais objetos. Ou seja, apresentar os tijolos básicos que serão necessários para construirmos todo o nosso edifício intelectual. Estes tijolos básicos serão chamados de símbolos e conjuntos finitos destes símbolos serão chamados de alfabetos.

Definição 2.1.1 — Alfabeto e símbolos. Um *alfabeto* é um conjunto finito qualquer e *símbolos* são elementos deste conjunto.

■ **Exemplo 2.1** Considere o conjunto $X = \{a, b, c, d\}$. Os elementos a, b, c, d do conjunto X são chamados de *símbolos do alfabeto* X. ■

■ **Exemplo 2.2** Considere o conjunto $\Sigma = \{0, 1\}$. Os elementos $0, 1$ do conjunto Σ são chamados de *símbolos do alfabeto* Σ. ■

Note que os conceitos de alfabeto e símbolos nada mais são do que sinônimos dos conceitos matemáticos de conjuntos e elementos. Iremos usar vários alfabetos neste livro, entretanto, o mais utilizado será o alfabeto *alfabeto binário*, apresentando no Exemplo 2.2. Um exemplo de alfabeto bastante comum no mundo das linguagens de programação é o seguinte: $\Sigma_C = \{0, 1, 2, ..., 9, a, b, ..., z, A, B, ..., Z, ..., \&, \#, \%, >, <, ...\}$. A ideia é que este último alfabeto trata-se do conjunto de todos caracteres válidos em um programa escrito na linguagem C no padrão ANSI C. Poderíamos pensar em ou série de outros exemplos semelhantes.

Assim como não é possível construir um edifício usando apenas um tijolo, objetos matemáticos complexos não podem ser descritos usando-se apenas um símbolo. Para descrever objetos matemáticos complexos, iremos utilizar sequências de símbolos. A segunda definição que veremos neste capítulo se refere exatamente a isso.

Definição 2.1.2 — Strings. Dado um alfabeto Σ, uma *string sobre* Σ é uma sequência de símbolos de Σ justapostos.

Por exemplo, *aaba* é uma string sobre o alfabeto $X = \{a, b, c, d\}$, do Exemplo 2.1. A string 00101 é uma string sobre o alfabeto binário. Um outro exemplo de string é um programa escrito em linguagem C (note que tal programa é uma sequência de símbolos ANSI, incluindo os símbolos especiais usados para quebra de linha, indentação e espaçamento), que pode ser visto matematicamente como uma string sobre o alfabeto Σ_C, que mencionamos anteriormente.

Notação 2.1. *Em geral usamos a letra grega Σ para denotar alfabetos. No caso em que o alfabeto não estiver explicitamente definido, a letra grega Σ denotará, por convenção, o alfabeto binário.*

Terminologia 2.1. *Embora estejamos usando "string" para se referir a uma sequência de símbolo, observamos que em textos em português é comum também que se use "palavra" para se referir a estes mesmos objetos matemáticos.*

Definição 2.1.3 — Substring e concatenação de strings. Uma *substring* de w é uma subsequência de símbolos de w. A concatenação de duas strings x e y, denotada xy, é a string resultante da justaposição das strings x e y.

■ **Exemplo 2.3** As strings *ab*, *bb* e *bbc* são algumas das substrings da string *abbbbc*. ■

■ **Exemplo 2.4** Para um exemplo de concatenação, considere as strings $x = 111$ e $y = 000$. Neste caso, a string $xy = 111000$ é a concatenação de x e y e a string 000000 é a concatenação de y com ela mesma (i.e., $yy = 000000$). ■

Observe que concatenção não é uma operação comutativa. A partir do Exemplo 2.4, temos que $yx = 000111$, portanto $xy \neq yx$. Por outro lado, para quaisquer strings x, y, z, $(xy)z = x(yz)$, ou seja, a operação de concatenação de strings é associativa.

TEORIA DE LINGUAGENS FORMAIS

Em alguns livros de Teoria da Computação a definição de concatenação de strings é apresentada de maneira um pouco mais formal (veja o livro [Sud05], por exemplo) e em outros de maneira um pouco mais "intuitiva" (veja o livro [Sip06]), por exemplo). Esta variação é uma questão da ênfase que o autor quer dar ao assunto.

Na Definição 2.1.3 optamos pela versão mais intuitiva. De maneira semelhante, quando dissemos que concatenação é uma operação associativa nós não apresentamos uma demonstração matemática para isso (no livro [Sud05] tal demonstração é feita). Esta opção que fizemos aqui não tem a ver com a importância em sermos formais, pois seremos bastante formais no decorrer do texto. A ideia é que o nosso curso tem um enfoque mais na linha de cobrir em *amplitude as bases da teoria da computação* e menos na linha de cobrir *em profundidade a teoria de linguangens formais*.

Ainda assim, no Exercício 2.3 pedimos por uma definição mais formal para a operação de concatenação de strings. O intuito disso é mais na linha de fazer o aluno pensar um pouco no assunto e perceber que podemos dar definições diferentes para um mesmo conceito, e menos na linha de tratar em detalhes formais os desdobramentos do assunto em questão. Para os alunos interessados, um tratamento um pouco mais aprofundado a respeito de teoria linguagens formais pode ser visto em mais detalhes em [Sud05].

Definição 2.1.4 — Tamanho de uma string. O *tamanho* de uma string w é o número de símbolos de w, e é denotado por $|w|$.

■ **Exemplo 2.5** Considere as strings $aabc$ e 01. Nestes casos escrevemos $|aabc| = 4$ e $|001| = 2$. ■

Definição 2.1.5 — String nula. Denotamos a *string nula*, ou seja, a string que não contém nenhum símbolo, por ε.

Observe que, de acordo com nossa definição, $|\varepsilon| = 0$. Note também que ε é substring de qualquer possível string.

A seguir, definiremos o que é a reversa de uma string. Intuitivamente, a ideia é simples: dada uma string w, a reversa de w, denotada por w^{R}, é a string lida de trás para frente (e.g., se $w = 1110$, então $w^{\mathrm{R}} = 0111$). Vamos formalizar isso usando uma definição indutiva[1]:

Definição 2.1.6 — Reversa de uma string. A reversa de uma string w, denotada w^{R}, é definida recursivamente da seguinte maneira:

Base: Se $w = \varepsilon$, então $w^{\mathrm{R}} = \varepsilon$.
Caso Geral: Seja w uma string com pelo menos um símbolo. Se a string w tem a forma $w = xa$, tal que x é uma string e a é o último símbolo de w, então $w^{\mathrm{R}} = ax^{\mathrm{R}}$.

Notação 2.2. *Seja Σ um alfabeto e $a \in \Sigma$. Diremos que uma string sobre este alfabeto é da forma a^n se a string é formada por n símbolos a consecutivos. De maneira análoga, se w é uma string sobre Σ, ao escrevermos w^n estamos nos referindo à concatenação de n cópias da string w.*

■ **Exemplo 2.6** A string 11111 pode ser escrita como 1^5. ■

■ **Exemplo 2.7** Considere a string $w = 10$. Com isso $w^6 = 101010101010$. ■

Note que, em particular, $x^0 = \varepsilon$ para qualquer string x. No Exercício 2.3, pedimos para o aluno fornecer uma definição formal para o conceito de concatenação de strings.

Definição 2.1.7 — Potência de um alfabeto. Dado um alfabeto Σ, o conjunto Σ^i de strings w sobre Σ, tal que $|w| = i$, é dito uma *potência de* Σ. Definimos $\Sigma^0 = \{\varepsilon\}$.

Por exemplo, dado o alfabeto binário Σ, o conjunto Σ^3 é o seguinte conjunto de strings: $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

Dado um alfabeto Σ e número natural i, note que Σ^i é um conjunto finito. A seguir vamos definir o conjunto infinito de todas as strings sobre Σ, ou seja, $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$. A definição formal é a seguinte.

Definição 2.1.8 — O conjunto Σ^*. Dado um alfabeto Σ, o conjunto de todas as strings sobre Σ é definido como

$$\Sigma^* = \bigcup_{i \geq 0} \Sigma^i.$$

[1] Uma definição indutiva é a mesma coisa que uma definição recursiva. Em teoria da computação será bastante comum usar raciocínio indutivo. Este tipo de raciocínio será útil não apenas em demonstrações de teoremas, mas também em várias definições.

2.1.1 Exercícios

Exercício 2.1 Seja $\Sigma = \{0,1\}$, $x \in \Sigma^*$ e $y \in \{a,b,c\}^*$. Responda verdadeiro ou falso.

(a) Se w é uma string da forma $x01$, então $|w| > 2$?
(b) Se w é uma string da forma $x010$, então $|w| \geq 3$?
(c) Se w é uma string da forma $y010$, então w é uma string sobre Σ?

■

Exercício 2.2 Seja Σ um alfabeto e i um número natural. Qual é a cardinalidade de Σ^i? ■

Exercício 2.3 Nesta seção, fornecemos uma definição informal para a concatenação de uma string x com ela mesmo n vezes, denotada x^n. Forneça uma definição formal indutiva para x^n. ■

Exercício 2.4 Apresente uma definição formal recursiva para Σ^* diferente da Definição 2.1.8. ■

Exercício 2.5 É verdade que $x^0 = y^0$ para qualquer par de strings x e y? ■

Exercício 2.6 Seja Σ um alfabeto qualquer, i um número natural e $w \in \Sigma^*$. Mostre que $(w^i)^R = (w^R)^i$? ■

2.2 Linguagens

Na seção anterior começamos com elementos fundamentais chamados símbolos e os usamos para construir objetos complexos chamados de strings. Agora vamos usar strings como elementos fundamentais para construir objetos ainda mais complexos chamados de *linguagens*.

Definição 2.2.1 — Linguagem. Seja Σ um alfabeto. Uma linguagem é um conjunto $L \subseteq \Sigma^*$.

Em outras palavras, dado um alfabeto, uma linguagem é um conjunto qualquer de strings sobre este alfabeto. Por exemplo, $L = \{1010, 11111, 0000001\}$ é uma linguagem sobre o alfabeto binário. Um exemplo de linguagem sobre o alfabeto Σ_{ANSI} é $A = \{a, aa, aaa, aaa, aaaa, \ldots\}$, ou seja, o conjunto de strings da forma a^i, para $i \geq 1$.

Observe que strings e linguagens são objetos matemáticos diferentes. Enquanto strings são *sequências*, linguagens são *conjuntos*. Enquanto strings tem tamanho *finito*, linguagens podem ter tamanho *finito ou infinito*.

Considere o alfabeto binário $\Sigma = \{0,1\}$. Nos Exemplos 2.8, 2.9, 2.10 e 2.11 apresentamos algumas linguagens binárias que serão bastante comuns neste curso:

■ **Exemplo 2.8** $L_{01} = \{\varepsilon, 01, 0011, 000111, \ldots\} = \{w \in \Sigma^* : w \text{ é da forma } 0^n1^n, \text{ para } n \geq 0\}$. ■

■ **Exemplo 2.9** $L_{\text{PAL}} = \{\varepsilon, 0, 1, 00, 11, 010, 101, 000, 111, \ldots\} = \{w \in \Sigma^* : w \text{ é um palíndromo}\}$. ■

■ **Exemplo 2.10** $L_{\text{P}} = \{10, 11, 101, 111, 1011, \ldots\} = \{w \in \Sigma^* : w \text{ é um número primo escrito em binário}\}$. ■

■ **Exemplo 2.11** $L_{\text{SQ}} = \{0, 1, 100, 1001, 10000, \ldots\} = \{w \in \Sigma^* : w \text{ é um número quadrado perfeito escrito em binário}\}$. ■

LINGUAGENS NATURAIS E LINGUAGENS FORMAIS

Observe que chamamos conjuntos de strings de linguagens. A razão disso é que há uma certa analogia com o conceito que intuitivamente conhecemos como linguagem, usada naturalmente em nosso dia a dia. Vamos a alguns exemplos que nos ajudarão a tornar isso mais claro.

■ **Exemplo 2.12** Seja $\Sigma_P = \{a, b, c, ..., z\}$ um alfabeto (i.e., o alfabeto da língua portuguesa). ■

Considere agora o conjunto L contendo todas as palavras da língua portuguesa. Note que este conjunto, por se tratar de um conjunto de strings sobre Σ_P, de acordo com Definição 2.2.1 é formalmente uma linguagem (por questão de simplicidade, estamos ignorando aqui acentos, hífens e outras sutilezas). Poderíamos chamar este conjunto de strings de "linguagem portuguesa". Por exemplo, observe que $bola \in L$, $caneta \in L$, $inconstitucional \in L$.

■ **Exemplo 2.13** Seja $\Sigma_P = \{a, b, c, ..., z, \sqcup\}$. ■

Ao incluirmos o símbolo "$\sqcup$" em nosso alfabeto, temos algo para ser usado de separador de palavras. Com isso, poderíamos montar um frase como $a \sqcup bola \sqcup verde \sqcup escura$. De maneira simplificada, novamente, poderíamos pensar na língua portuguesa como sendo o conjunto L' de todas as strings que correspondem a frases válidas em português.

Neste ponto, um linguista nos chamaria atenção (com razão!) para o fato que tentar definir matematicamente a língua portuguesa não é uma tarefa tão fácil e argumentaria que o conteúdo das linguagens L e L' que acabamos de ver não é tão bem definido como estamos sugerindo. Poderíamos até tentar contra-argumentar dizendo que, pelo menos no caso de L, seria possível sermos formais definindo a linguagem como sendo o conjunto de todas as palavras de algum dicionário específico fixado a priori (já para o caso de fornecer uma definição exata para L' a tarefa seria bem mais difícil).

Entretanto, o nosso foco neste livro não são linguagens naturais como o português ou qualquer outra língua natural, sendo que os exemplos vistos acima são úteis apenas para ilustrar a analogia que há entre o conceito matemático de linguagens e o conceito intuitivo de linguagens naturais. Ainda assim, existem muitas linguagens que nós, profissionais de computação, lidamos no dia a dia e que são completamente formais. Voltando à um exemplo que vimos anteriormente, considere o alfabeto Σ_C da Seção 2.1. A linguagem sobre Σ_C, definida abaixo é matematicamente precisa:

$$L_C = \{w \in \Sigma_C^* : \text{"}w \text{ é um programa válido em linguagem C"}\}$$

A linguagem L_C, definida acima, consiste de todos os (infinitos) possíveis programas válidos escritos em linguagem C. Embora tenhamos escrito textualmente a expressão informal "w é um programa válido em linguagem C" na nossa especificação das strings contidas no conjunto L_C, esta linguagem pode, sim, ser matematicamente bem definida. Para tal precisamos de algumas ferramentas matemáticas, especialmente um conceito matemático conhecido como *gramática formal*. Mais adiante neste livro veremos formalmente o que é uma gramática formal e discutiremos brevemente algumas aplicações disto na especificação de linguagens de programação e na construção de compiladores.

Em diversas situações, nós queremos interpretar strings binárias não apenas como meras sequências de 0's e 1's, mas como números naturais representados em base binária. Em tais situações a seguinte notação será conveniente.

Notação 2.3. *O número natural representado pela string binária w é denotado por $N(w)$.*

Por exemplo, se $w_1 = 101$ e $w_2 = 1111$, então $N(w_1) = 5$ e $N(w_2) = 15$. Note que N não é uma bijeção, pois várias strings podem estar associadas à um mesmo número. Por exemplo, $N(01) = N(1) = 1$).

Convenção: $N(\varepsilon) = 0$.

A vantagem da Notação 2.3, é tornar a nossa escrita mais concisa na definição de algumas linguagens. Por exemplo, a linguagem dos números primos e a linguagens do números múltiplos de 3 podem ser denotadas, respectivamente, por $L_P = \{w : N(w)$ é um número primo$\}$ e $L_3 = \{w : N(w)$ é um múltiplo de $3\}$.

REFLETINDO UM POUCO: PROBLEMAS COMPUTACIONAIS E LINGUAGENS

O que significa *"o problema de testar se um número n é primo"*? Uma maneira de tentar formalizar isso é pensar que isso é equivalente ao seguinte problema: dada uma string w, decidir se w é a representação binária de um número primo, ou seja, testar se a string w pertence a linguagem L_P do Exemplo 2.10.

Para àqueles que não gostam de números binários, poderíamos alternativamente definir a linguagem dos números primos usando outros alfabetos (e.g., o alfabeto dos dígitos de 0 a 9), mas isso realmente não é tão importante aqui. O ponto central é que uma vez que fixamos um alfabeto, digamos o alfabeto binário, *a linguagem L_P incorpora a propriedade "ser primo"*, pois todos os objetos contidos neste conjunto tem tal propriedade e todos os objetos que não fazem parte deste conjunto não tem tal propriedade. Portanto, responder se um número é primo se reduz a distinguir se uma dada string pertence ou não ao conjunto L_P.

Em teoria da computação é bastante comum pensamos em linguagens como sinônimos de problemas computacionais. Embora esta ideia pareça simples, ela também é bastante poderosa. Existem outras maneiras de tornar formal o conceito de problema computacional e, no decorrer deste livro, veremos outras definições mais sofisticadas para tal. Entretanto, em boa parte do curso o uso de linguagens será o formalismo padrão para representar problemas.

Para quem ainda não está convencido de que existe tal conexão direta entre o conceito de linguagem e o conceito de problema, uma maneira intuitiva de se pensar a respeito disso segue na seguinte linha: No funcionamento interno de um computador, qualquer objeto matemático, em última análise, é uma sequência de bits. No caso particular do exemplo acima, quando escrevemos um programa para testar se um dado número é primo, o que o programa faz é testar se uma sequência de bits da memória do computador representa ou não representa um número primo em binário. Ou seja, no final das contas, o que o programa está fazendo é testar se uma string do alfabeto binário pertence ou não à linguagem L_P. Não é muito difícil de ver que esse tipo de raciocínio pode ser generalizado para uma série de outros problemas. De fato, qualquer problema cuja solução envolva testar se um dado objeto matemático tem uma certa propriedade pode ser modelado usanddo uma linguagem.

2.2.1 Exercícios

Exercício 2.7 Seja Σ um alfabeto arbitrário. Com relação a linguagens sobre Σ, responda:
(a) Σ^* é uma linguagem?
(b) $\varnothing$ é uma linguagem?
(c) $\{\varepsilon\}$ é uma linguagem?
(d) O conjunto potência Σ^4 é uma linguagem? ■

Exercício 2.8 Forneça uma definição formal para a linguagem sobre $\Sigma = \{0,1\}$, das strings que representam números múltiplos de 4 escritos em binário. ■

Exercício 2.9 Seja Σ um alfabeto qualquer. É verdade que a linguagem $\{w : \exists x \in \Sigma^* \text{ tal que } w = xx^R\}$ é a linguagem de todos os palíndromos construídos com símbolos de Σ? Em caso afirmativo, prove. Em caso negativo, forneça um contra-exemplo e, em seguida, forneça uma definição formal adequada para a linguagem dos palíndromos construídos com símbolos de Σ. ■

2.2.2 Operações com linguagens

Assim como podemos criar strings maiores a partir da operação de concatenação de strings menores, vamos definir agora uma operação análoga para linguagens, chamada de concatenação de linguagens.

Definição 2.2.2 A *concatenação de duas linguagens* L e M é o conjunto de todas as strings que podem ser formadas tomando-se uma string x de L e uma string y de M e fazendo a concatenação xy. Denotamos a linguagem resultante por LM ou $L \cdot M$.

■ **Exemplo 2.14** Considere as linguagens $L = \{00,0\}$ e $L' = \{1,111\}$. A concatenação destas duas linguagens é $LL' = \{001,00111,01,0111\}$. ■

■ **Exemplo 2.15** Seja $L = \{a,aa,aaa,...\}$ e $M = \{\varepsilon,x,xx,xxx,...\}$. A linguagem LM contém as strings a, ax, axx, $axxx$, ..., e também as strings aa, aax, $aaxx$, $aaxxx$, ... e assim por diante. De maneira compacta: $LM = \{a^i x^k \ : \ i \geq 1 \text{ e } k \geq 0\}$ ■

Exercício resolvido 2.1 Seja Σ o alfabeto binário e L_3 e R as linguagens sobre Σ definidas a seguir: $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$ e $R = \{0\}$. Qual é a linguagem L_3R?

Solução: As strings contidas na linguagem da concatenção L_3R são as strings de L_3 adicionadas de um 0 ao final. Note que quando adicionarmos 0 ao final de um número binário o resultado é um novo número binário que é dobro do número original. Portanto a linguagem resultante é: $L_3R = \{w : N(w) \text{ é um múltiplo de } 6\}$. ■

Exercício 2.10 Seja Σ o alfabeto binário e L_3 e R as linguagens sobre Σ definidas a seguir: $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$ e $R = \{\varepsilon,0\}$. Qual é a linguagem L_3R? ■

Exercício resolvido 2.2 Seja Σ o alfabeto binário. Seja $R = \{0\}$ uma linguagem sobre Σ. Qual é a linguagem $\Sigma^* R$? E qual é a linguagem $\Sigma^* RR$?

Solução: Lembrando que as strings binárias terminadas em 0 são precisamente os múltiplos de 2 em binário e a o multiplicarmos múltiplos de 2, temos múltiplos de 4, temos que:

- $\Sigma^* R = \{w : N(w)$ é um múltiplo de $2\}$.
- $\Sigma^* RR = \{w : N(w)$ é um múltiplo de $4\}$.

■

Notação 2.4. *Usamos a notação L^i para denotar a concatenação de uma linguagem L consigo mesmo i vezes, para uma constante i, isto é, $L^i = \underbrace{LL...L}_{i \text{ vezes}}$.*

A operação de concatenação de uma linguagem com ela mesma pode ser feita infinitamente, como definido a seguir:

Definição 2.2.3 Seja L uma linguagem. O Fechamento (ou Fecho de Kleene) de L, denotado por L^*, é o conjunto de strings que podem ser formadas tomando-se qualquer número de strings de L (possivelmente com repetições) e as concatenando. Isto é,

$$L^* = \bigcup_{i \geq 0} L^i, \quad \text{sendo que } L^0 = \{\varepsilon\} \text{ e } L^i = \underbrace{LL...L}_{\text{i vezes}}.$$

■ **Exemplo 2.16** Seja $L = \{000, 111\}$. Então L^* = {ε, 000, 111, 000000, 000111, 111000, 111111, 000000000, 000000111, 000111000, ... }. ■

Observe que ao escrevermos A^*, devemos especificar exatamente o que significa A. Se A é um alfabeto, então A^* é a união infinita de conjuntos potência de A. Por outro lado se A é uma linguagem, então A^* é o fecho de A, ou seja, a sequência infinita de concatenações de A consigo mesma. Entretanto, em ambos os casos, note que A^* é uma linguagem. Mais precisamente, a linguagem de todas as strings construídas usando os elementos do conjunto A como "tijolos básicos". Caso A seja um alfabeto, estes tijolos básicos são símbolos, caso A seja uma linguagem, estes tijolos básicos são strings.

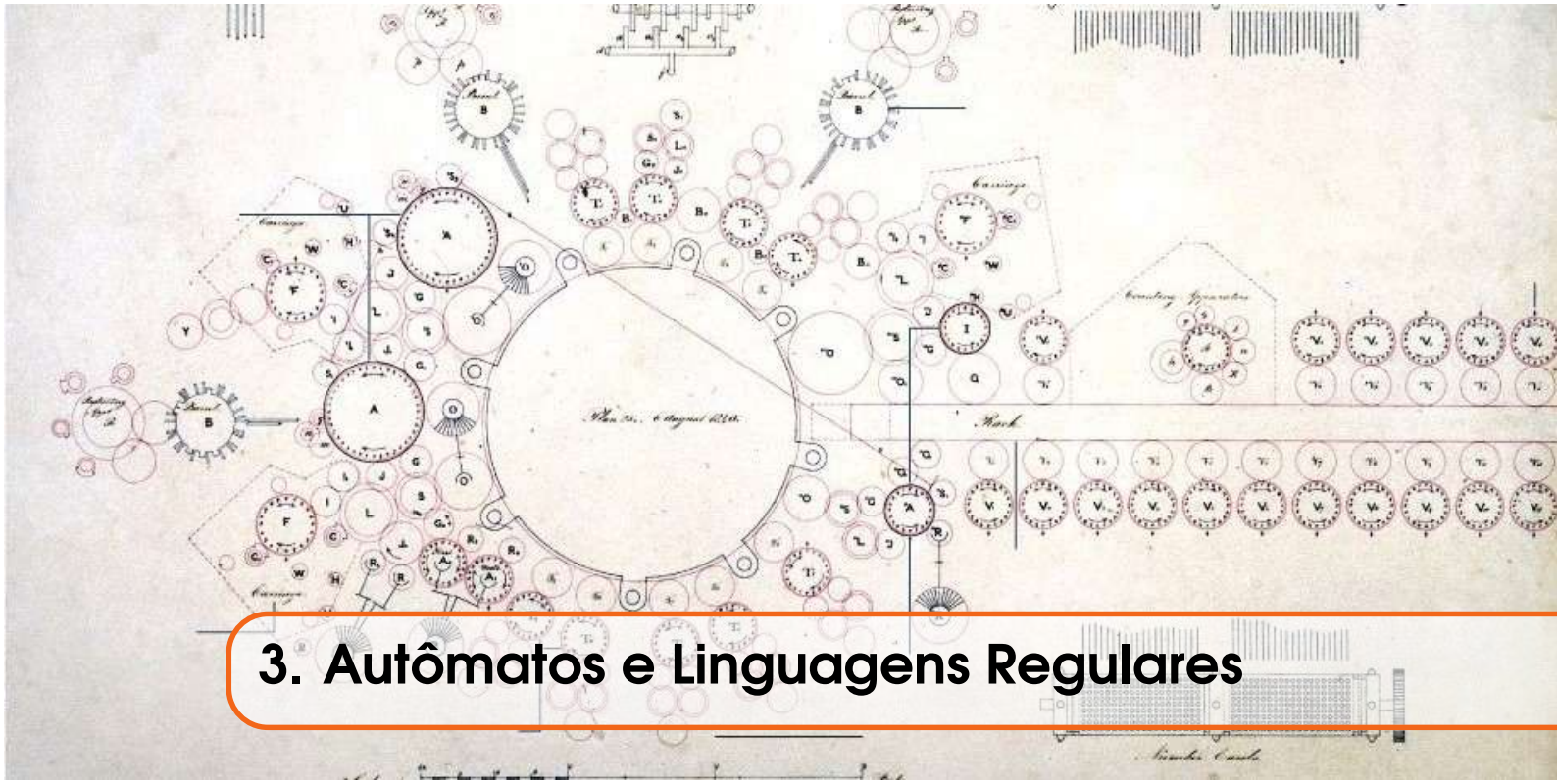
Exercício resolvido 2.3 Considere as linguagens $L = \{1\}$ e $R = \{0\}$. Qual é a linguagem LR^*?

Solução: A linguagem é $LR^* = \{w \mid N(w)$ é uma potência de $2\}$ ■

Exercício 2.11 Nas questões abaixo, A é sempre um alfabeto e L é sempre uma linguagem.

- Para todo A, é verdade que $A = A^*$?
- Existe A, tal que $A = A^*$?
- Para todo A, é verdade que $A^* = (A^*)^*$
- Para todo L, é verdade que $L^* = (L^*)^*$
- Para todo L e A tal que L é uma linguagem sobre A, é verdade que $A^* = L^*$
- Existe uma linguagem L sobre A tal que $A^* = L^*$.
- Seja L uma linguagem não vazia. Qual é a linguagem resultante da concatenação de L com o conjunto vazio?

■



3. Autômatos e Linguagens Regulares

Um dos objetivos que queremos alcançar neste livro é prover uma definição formal para o conceito de algoritmo. Nós alcançaremos este objetivo no Capítulo 5, com a definição das Máquinas de Turing. O que veremos agora é a definição de *autômatos finitos*, que é um modelo matemático com poder de expressão menor do que as Máquinas de Turing. Por poder de expressão menor, queremos dizer que apenas um subconjunto de todos os possíveis algoritmos podem ser representados por autômatos finitos. Entretanto, estudar este modelo será útil para nos familiarizarmos com os conceitos matemáticos que serão muito utilizados no Capítulo 5. Além disso, por si só, o assunto que veremos agora é bastante útil em aplicações práticas, como busca de padrões em textos e construção de analisadores léxicos para compiladores.

3.1 Autômatos Finitos Determinísticos (AFDs)

Considere uma máquina que vende chocolates que custam 6 reais. Abaixo mostramos uma figura ilustrando tal máquina juntamente com um diagrama que descreve o mecanismo de funcionamento interno dela.

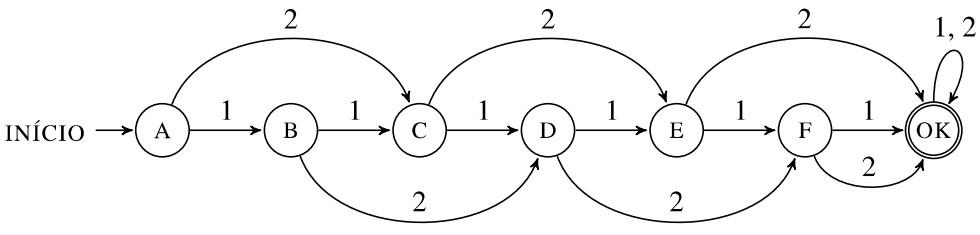


Figure 3.1: Uma máquina que vende chocolates de R$ 6 e aceita moedas de R$ 1 e R$ 2. Ao lado o diagrama simplificado descrevendo o seu mecanismo de funcionamento interno. Por exemplo, se inserirmos uma moeda de R$ 1 e uma moeda de R$ 2, a máquina, atingirá o estado D. A máquina atingirá o estado OK se for inserida uma sequência de moedas cuja soma seja pelo menos R$ 6.

Diagramas como o da Figura 3.1 são conhecidos como *máquinas de estados*, *máquinas de estados finitos* ou *autômatos finitos determinísticos*. A ideia é entender que a entrada da máquina é uma sequência de moedas (podemos também pensar que a entrada é uma "string de moedas"), que podem ser de 1 ou 2 reais. A máquina libera o chocolate se as moedas somarem pelo menos 6 reais. Há uma indicação de INÍCIO no estado A, pois este é o estado que a máquina se encontra antes de todo processo começar. A partir do estado A, a máquina faz uma transição para cada moeda que é fornecida como entrada. A máquina libera o chocolate ao final do processo se, e somente se, ela atingir o *estado* OK. Note que estamos em um cenário simplificado em que a máquina não precisa fornecer troco.

> REFLETINDO UM POUCO: MODELOS MATEMÁTICOS E OBJETOS FÍSICOS
>
> No Capítulo 1 dissemos que podemos pensar em computadores como "instanciações de objetos abstratos e seus possíveis relacionamentos matemáticos em objetos físicos e seu conjunto de possíveis graus de movimento". Note que a Figura 3.1 apresenta exatamente isso, pois:
>
> (1) A máquina de chocolates é um objeto físico (embora devemos estar atentos ao fato de que ela não é um computador de propósito geral, pois trata-se se uma máquina que implementa um único algoritmo: o algoritmo que testa se a soma das moedas de entrada é maior ou igual a 6 reais).
> (2) O diagrama faz o papel do objeto matemático abstrato, em particular, um objeto matemático que consiste de um conjunto de estados e um conjunto de símbolos (o conjunto $\{1, 2\}$ dos símbolos que aparecem nos rótulos das transições). O diagrama também representa certas relações matemáticas, como os relacionamentos entre estados e símbolos indicados pelas setas saindo de um estado e indo para outro. A definição exata destas relações ficará clara no decorrer desta seção.
>
> O ponto central aqui é que *o objeto matemático e suas relações matemáticas estão "amarradas" em uma correspondência de um para um com o objeto físico e seus graus de movimento*. Os graus de liberdade de movimento são os tipos de movimentações que os mecanismos internos da máquina de vender chocolate tem.

3.1.1 Modelando matematicamente autômatos

Nesta seção veremos uma definição precisa para os objetos matemáticos ilustrados por diagramas como vistos na seção anterior.

Definição 3.1.1 — Autômato Finito Determinístico (AFD). Uma Autômato Finito Determinístico, também chamado de AFN, é uma 5-tupla $D = (Q, \Sigma, \delta, q_0, F)$, tal que:

Q é o conjunto de *estados*;

Σ é o conjunto de *símbolos de entrada*;

δ é a *função de transição* $\delta : Q \times \Sigma \to Q$;

$q_0 \in Q$ é o *estado inicial*;

$F \subseteq Q$ é o conjunto de *estados finais*.

Normalmente, diremos simplesmente *autômatos* ou usaremos simplesmente a sigla *AFD* para se referir aos autômatos finitos determinísticos[1]. Algo importante de se observar é que esta definição é genérica, ou seja, se quisermos uma definição matemática particular para o diagrama da nossa máquina de chocolates, teríamos que dar uma *instância específica da Definição 3.1.1*. O exemplo a seguir ilustra o que queremos dizer com isso.

[1] Algumas vezes usa-se o acrônimo *DFA*, que vem da expressão em inglês "*deterministic finite automata*."

■ **Exemplo 3.1 — Máquina de vender chocolates.** Uma definição matemática para o exemplo da Figura 3.1 é o autômato finito determinístico $C = (Q_c, \Sigma_c, \delta_c, \mathrm{A}, F_c)$, sendo que os componentes da 5-tupla são:

$Q = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{OK}\}$;
$\Sigma = \{1, 2\}$;
$\delta_c : Q \times \Sigma \to Q$ é a função definida caso a caso a seguir:
$\delta_c(\mathrm{A}, 1) = \mathrm{B}$, $\delta_c(\mathrm{A}, 2) = \mathrm{C}$, $\delta_c(\mathrm{B}, 1) = \mathrm{C}$, $\delta_c(\mathrm{B}, 2) = \mathrm{D}$, $\delta_c(\mathrm{C}, 1) = \mathrm{D}$, $\delta_c(\mathrm{C}, 2) = \mathrm{E}$, $\delta_c(\mathrm{D}, 1) = \mathrm{E}$, $\delta_c(\mathrm{D}, 2) = \mathrm{F}$, $\delta_c(\mathrm{E}, 1) = \mathrm{F}$, $\delta_c(\mathrm{E}, 2) = \mathrm{OK}$, $\delta_c(\mathrm{F}, 1) = \mathrm{OK}$, $\delta_c(\mathrm{F}, 2) = \mathrm{OK}$, $\delta_c(\mathrm{OK}, 1) = \mathrm{OK}$, $\delta_c(\mathrm{OK}, 2) = \mathrm{OK}$.
O estado A é o estado inicial;
O conjunto unitário $\{\mathrm{OK}\}$ é o conjunto de estados finais. ■

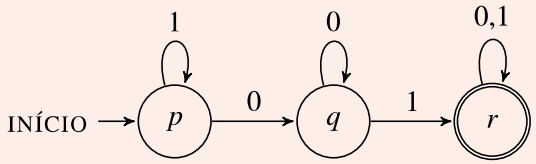
Neste momento pode ser instrutivo lembrar da analogia que fizemos no Capítulo 1, quando discutimos definições genéricas e definições específicas. Naquela ocasião mencionamos a definição matemática genérica de "função" e as definições matemáticas de funções particulares, como x^2 ou $\log x$. Na definição 3.1.1 temos a definição para um AFD em geral. Por outro lado, no exemplo 3.1, a 5-tupla $C = (Q_c, \Sigma_c, \delta_c, \mathrm{A}, F_c)$ é uma definição matemática do AFD especificamente correspondente a máquina de vender chocolates.

Exercício resolvido 3.1 Desenhe o diagrama do autômato A definido a seguir:

$A = (Q, \Sigma, \delta, p, \{r\})$, tal que $Q = \{p, q, r\}$, $\Sigma = \{0, 1\}$ e a função δ é definida abaixo:

$\delta(p, 1) = p$
$\delta(p, 0) = q$
$\delta(q, 0) = q$
$\delta(q, 1) = r$
$\delta(r, 0) = r$
$\delta(r, 1) = r$

Solução:



■

A função δ do enunciado do Exercício 3.1 é um tipo de função que é definida ponto a ponto (ou seja, ela não tem uma regra geral que descreve o comportamento da função, como é o caso de funções que estamos acostumados a estudar em cálculo, como, por exemplo, $f(x) = x^2$). Em teoria da computação, funções que não podem ser facilmente descritas por regras gerais serão bastante comuns. Para facilitar a nossa vida, vamos descrever δ usando uma tabela, como esta apresentada no exemplo a seguir:

Na primeira coluna da Tabela 3.1.1, em negrito, temos os estados p, q, r do autômato. A seta ao lado do estado p indica que ele é o estado inicial e o asterisco ao lado do estado r indica que ele é um estado final. No topo da tabela, também em negrito, temos os símbolos do alfabeto, ou seja, 0 e

	0	1
→ **p**	q	p
q	q	r
***r**	r	r

Table 3.1: Tabela de transições da função δ do AFD do Exercício 3.1.

1. Preenchemos a posição correspondente a linha do estado p e a coluna do símbolo 0 com o valor q, pois $\delta(p,0) = q$. As demais linhas da tabela são preenchidas de maneira semelhante.

No decorrer deste livro, ao nos referirmos à um AFD, estritamente falando, estaremos nos referindo à uma 5-tupla. Porém, por questões de conveniência, será bastante comum apresentarmos a tabela de transições ou mesmo o diagrama, para descrever o AFD em questão.

Exercício 3.1 Projete uma máquina que venda chocolates que custem 2 reais. Ela deve aceitar moedas de 10 e 50 centavos e moedas de 1 real. Você deve tanto desenhar o diagrama quanto apresentar a definição formal. ■

3.1.2 Aceitação e rejeição de strings: ideia intuitiva

Agora que nós já temos o nosso modelo matemático para autômatos finitos, vamos interpretá-lo. Mas o que queremos dizer com "interpretá-lo"? O que vamos fazer é apresentar definições matemáticas para que possamos entender este modelo matemático como um *modelo de computação* e não como uma mera 5-tupla. Antes de formalizarmos isso (Seção 3.1.3), vamos analisar o processo computacional em questão com cuidado.

A ideia central é enxergar o AFD como um objeto que receba uma string $w = w_1w_2...w_n$ como entrada, processe um a um os símbolos de w, e produza uma saída. Durante o processamento de w, cada passo consiste em ler um símbolo w_i, fazer uma mudança de estado e descartar w_i. O AFD irá, passo a passo, descartar símbolos de w e irá também mudando de estados neste processo. O ponto central é que podemos fazer a seguinte pergunta:

- Em que estado o AFD se encontra após o descarte do último símbolo de w?

Digamos que depois da última transição de estados o AFD atinja um dado estado q. A ideia aqui é que se o estado q for um estado final, nós iremos dizer que o AFD *aceitou* a string w. Por outro lado, se q não for um estado final, nós iremos dizer que o AFD *rejeitou* a string w. Os estados finais também são chamados de estados de aceitação do autômato.

Exercício resolvido 3.2 Seja $\Sigma = \{0,1\}$. Considere a linguagem binária L das strings que contém a substring 01, ou seja, $L = \{w : w \text{ é da forma } x01y, \text{ sendo que } x,y \in \Sigma^*\}$. Construa um AFD que aceite todas e somente as strings de L.

Solução: Podemos usar o mesmo AFD usado no Exercício 3.1. ■

Exercício 3.2 Seja $\Sigma = \{a,b,c\}$. Seja $L = \{w \in \Sigma^* : w \text{ é da forma } xbbya, \text{ sendo que } x,y \in \Sigma^*\}$. Construa um AFD que aceite a string w fornecida como entrada se, e somente se, $w \in L$. ■

Exercício 3.3 Seja $\Sigma = \{0,1\}$. Construa AFDs que aceite a string fornecida como entrada se, e somente se, a string pertence a linguagem L, para cada um dos casos abaixo:

(a) $L = \{w : w \text{ tenha um número ímpar de 1's}\}$.
(b) $L = \{w : |w| \leq 3\}$.
(c) $L = \{w : w \text{ tem ao mesmo tempo um número par de 0's e um número par de 1's}\}$
(d) $L = \{w : \text{se } w' \text{ é uma substring de } w \text{ com } |w'| = 5, \text{ então } w' \text{ tem pelo menos dois 0's}\}$.
(e) $L = \{w : w \text{ é terminada em } 00\}$.
(f) $L = \{w : \text{o número de 0's em } w \text{ é divisível por 3 e o número de 1's é divisível por 5}\}$.
(g) $L = \{w : w \text{ contém a substring } 011\}$.
(h) $L = \Sigma^*$

O que precisamos fazer agora é transformar esta ideia intuitiva de que um AFD aceita algumas strings e rejeita outras strings em definições matemáticas precisas. Ou seja, queremos saber matematicamente o que significa um certo AFD $D = (Q, \Sigma, \delta, q_0, F)$ aceitar ou rejeitar uma dada string w.

3.1.3 Aceitação e rejeição de strings: definição formal

Voltando a nossa máquina de vender chocolates, considere o seguinte: Se a máquina estiver no estado B e receber 3 moedas de 1 real, em que estado a máquina vai parar? A resposta é que a máquina irá atingir o estado E. O mais importante aqui é obvervar a *pergunta* que fizemos: *dado um estado e uma sequência de moedas, qual é o estado resultante?* O que vamos fazer agora é formalizar esta idéia de que dado um estado q e uma sequência de símbolos w, obtemos o estado resultante é q'. O objeto matemático para modelar esta ideia é uma função $f : Q \times \Sigma^* \to Q$ de forma que $f(q,w) = q'$. Uma vez a definição de tal função depende da função δ original da máquina, vamos chamar esta nova função de $\hat{\delta}$.

Definição 3.1.2 — Função de trasição estendida. Dado um AFD $D = (Q, \Sigma, \delta, q_0, F)$, a *função de transição estendida* $\hat{\delta}$ de D é a função $\hat{\delta} : Q \times \Sigma^* \to Q$ definida indutivamente:

Caso base: $w = \varepsilon$.
$\hat{\delta}(q, \varepsilon) = q$

Indução: $|w| > 0$.
Seja w uma string da forma $w = xa$, sendo que $x \in \Sigma^*$ e $a \in \Sigma$. Então $\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x), a)$.

A razão de termos apresentado a definição 3.1.2 é que agora temos uma maneira precisa de dizer o que significa um AFD D aceitar ou rejeitar uma string. A maneira precisa é a seguinte:

Definição 3.1.3 — Aceitação e rejeição de strings. Seja $D = (Q, \Sigma, \delta, q_0, F)$ um AFD e $w \in \Sigma^*$. Se $\hat{\delta}(q_0, w) \in F$, então dizemos que D *aceita* a string w. Se $\hat{\delta}(q_0, w) \notin F$, então dizemos que D *rejeita* a string w.

Podemos generalizar a ideia de aceitação de string para a ideia de aceitação de uma linguagem. Se L é o conjunto de todas as strings aceitas por D, então dizemos que D aceita a linguagem L. Neste caso dizemos que L é a linguagem de D. Isto é definido formalmente a seguir.

Definição 3.1.4 — Linguagem de um AFD. A linguagem de um AFD D, denotada por $L(D)$, é definida como $L(D) = \{w : \hat{\delta}(q_0, w) \in F\}$.

Se L é a linguagem de um AFD D, dizemos que D *decide* a linguagem L. Dizemos também que D *aceita* a linguagem L[2]. Agora segue a definição mais importante deste capítulo:

[2] Alguns textos utilizam também a expressão "L é reconhecida por D"

Definição 3.1.5 — Liguagem Regular. Dada uma linguagem L, se existe um AFD D tal que $L = L(D)$, então L é dita uma *linguagem regular*.

3.1.4 Exercícios

Exercício 3.4 Seja um AFD com alfabeto Σ e conjunto de estados Q. Sejam $x, y \in \Sigma^*$, $a \in \Sigma$ e $q \in Q$ quaisquer.

(a) Mostre que $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q,x), y)$.

(b) Mostre que $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q,a), x)$. ■

Exercício 3.5 Considere o seguinte AFD $D = (Q, \Sigma, \delta, q_0, F)$ tal que

- $Q = \{q_0, q_1, q_2, ..., q_6\}$
- $\Sigma = \{0, 1, 2, ..., 6\}$
- $F = \{q_0\}$
- Função de transição δ definida a seguir:
  $\delta(q_j, i) = q_k$, sendo que $k = (j + i) \bmod 7$

Qual é a linguagem aceita por D? ■

Exercício 3.6 Forneça um AFD com alfabeto binário que aceite a linguagem: $L_3 = \{w : N(w)$ é um múltiplo de $3\}$. ■

Exercício 3.7 Prove formalmente que a sua solução para o Exercício 3.6 é correta. ■

Exercício 3.8 Seja uma linguagem L sobre o alfabeto Σ. O complemento da linguagem L, denotado por $\overline{L}$, é defindo da seguinte maneira: $\overline{L} = \Sigma^* \setminus L$. Prove que se L é regular, então $\overline{L}$ também é regular. ■

Exercício 3.9 Considere os seguintes AFDs: $D_Q = (Q, \Sigma, \delta_Q, q_0, F_Q)$ e $D_P = (P, \Sigma, \delta_P, p_0, F_P)$ sendo $Q = \{q_0, q_1, ..., q_k\}$ e $P = \{p_0, p_1, ..., p_h\}$.

Construiremos agora um terceiro AFD D_A a partir dos dois AFDs anteriores. A ideia é que dada qualquer string w, o AFD D_A vai simular ao mesmo tempo a computação de D_Q com w e D_P com w. Por exemplo, se na terceira transição D_Q estiver no estado q_1 e D_P estiver no estado p_5, então na terceira transição o AFD D_A estará em um estado chamado (q_1, p_5). A definição formal dos componentes de $D_A = (A, \Sigma, \delta_A, a_0, F_A)$ segue abaixo:

- $A = Q \times P$ (ou seja, para cada $q_i \in Q$ e $p_j \in P$, temos um estado $(q_i, p_j) \in A$)
- $a_0 = (q_0, p_0)$
- $F_A = \{$"conjunto de todos os elementos (q_i, p_j) tal que $q_i \in F_Q$ e $p_j \in F_P$"$\}$.
- Definição de δ_A: Para todo $q_i, q_j \in Q$ e todo $p_r, p_t \in P$ e todo símbolo $s \in \Sigma$ temos que: Se $\delta_Q(q_i, s) = q_j$ e $\delta_P(p_r, s) = p_t$, então $\delta_A((q_i, p_r), s) = (q_j, p_t)$.

Sendo $L_Q = L(D_Q)$ e $L_P = L(D_P)$, responda: Qual é a linguagem aceita por D_A? ■

3.2 Autômatos Finitos não Determinísticos (AFNs)

A computação com AFDs é completamente determinística, ou seja, para cada par (q, a), sendo q um estado e a um símbolo, temos exatamente uma transição definida no ponto (q, a). E se

quiséssemos definir um modelo abstrato de autômato que em certos momentos possa escolher uma entre várias transições possíveis de maneira não determinística? E se este autômato tivesse uma habilidade "mágica" de advinhar qual é a transição correta a ser executada no momento? Um exemplo de um diagrama de um autômato com as propriedades que queremos é o seguinte:

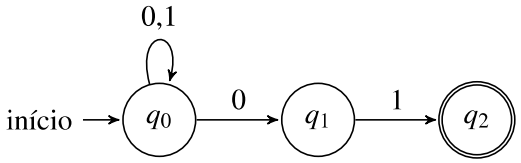


Figure 3.2: Um autômato finito não determinístico.

Imediatamente notamos uma diferença no autômato da Figura 3.2 em relação aos autômatos da seção anterior. Neste exemplo, se o autômato estiver no estado q_0 e o símbolo lido for 0, existem duas possibilidades: (1) o autômato pode continuar no estado q_0; (2) o autômato pode mudar para o estado q_1. A pergunta chave é a seguinte: dado um autômato com estas características, quais são as strings que o autômato aceita?

Para responder esta pergunta, vamos agora ser mais precisos sobre o que queremos dizer com "o autômato advinha" qual transição fazer. A ideia é que, dada uma string w, se existe uma sequência de passos que leve o autômato a atingir um estado final ao finalizar o processamento de w, então o autômato irá escolher, a cada momento, uma transição que leve a computação em um caminho correto. Em tais casos, dizemos que string pertence a linguagem do autômato.

Mesmo com esta habilidade nova, pode ocorrer que não exista nenhuma sequência de transições que leve o autômato a aceitar certas strings. Por exemplo, o autômato da Figura 3.2 não "consegue" aceitar a string 100. Neste caso diremos que tais strings não estão na linguagem do autômato.

Se prestarmos atenção no autômato da Figura 3.2, veremos que ele tem a habilidade de aceitar exatamente as strings terminadas em 01. Dada uma string da forma $x01$, tal que x é uma substring qualquer, o autômato pode processar toda a substring x usando o "laço" sobre o estado q_0. Quando faltarem apenas os dois símbolos finais, o autômato utiliza a transição que vai ao estado q_1 e depois a transição que vai ao estado q_2. Observe que se a string não é terminada em 01, o autômato não conseque atingir o estado final em hipótese alguma.

Observe que a habilidade do autômato poder escolher entre duas transições possíveis não é a única diferença que este novo modelo tem em relação aos autômatos determinísticos da seção anteriror. Uma outra situação que pode ocorrer neste modelo e que não ocorria no caso determinístico é aquela em que o autômato não tenha nenhuma transição definida para um determinado par (estado, símbolo). Por exemplo, se o autômato da Figura 3.2 estiver no estado q_1 e próximo símbolo a ser lido for 0, o autômato não terá nenhuma opção de transição para realizar. Em tal caso, diremos que o autômato *morre*.

NÃO DETERMINISMO?

Claramente, do ponto de vista prático, um autômato não determinístico não parece ser um modelo realista de computação correspondendo a algo concreto do mundo real. Ainda assim, o estudo deste modelo matemático será bastante útil.

Em teoria da computação este tipo de situação é bastante comum. O ponto chave é que estes modelos "não realistas" podem ser pensados, em última análise, como *ferramentas matemáticas úteis*, inclusive úteis para nos ajudar a entender modelos "realistas" de computação. No Capítulo 8, veremos que Máquinas de Turing não determinísticas podem ser usadas para definir um conjunto de linguagens conhecido como *NP*, que é parte da famoso (e concreto) problema *P vs NP*. Em um curso mais aprofundado de complexidade computacional a definição de modelos "irreais" de computação, mas que ainda assim sejam matematicamente úteis para lidar com problemas ou modelos concretos de computação, acontece com muita frequência.

3.2.1 Definição formal para autômatos finitos não determiníticos

Autômatos finitos não determinísticos tem uma definição formal muito parecida com a definição dos AFDs. A única diferença é que dado um par (q,a), sendo q um elemento do conjunto Q de estados do autômato e a um símbolo, pode ser que exista zero, um, ou mais estados possíveis de serem atingidos por uma transição com rótulo a. Mais precisamente, a função de transição tem a forma $\delta(q,a) = S$, sendo que S é um conjunto qualquer de estados. Por exemplo, a função δ do autômato da Figura 3.2, quando aplicada a $(q_0,0)$, deve ter a forma $\delta(q_0,0) = \{q_0,q_1\}$. Note que, como os elementos da imagem de δ são conjuntos, o contradomínio da função δ é conjunto de todos os subconjuntos de Q, ou seja, o conjunto potência $\mathscr{P}(Q)$.

Definição 3.2.1 — Autômato Finito não Determinístico (AFN). Um Autômato Finito não Determinístico, também chamado de AFN, é uma 5-tupla $A = (Q,\Sigma,\delta,q_0,F)$, tal que:

Q é o conjunto de *estados*;

Σ é o conjunto de *símbolos de entrada*;

δ é a *função de transição* $\delta : Q \times \Sigma \to \mathscr{P}(Q)$;

$q_0 \in Q$ é o *estado inicial*;

$F \subseteq Q$ é o conjunto de *estados finais*.

■ **Exemplo 3.2** A definição formal para o diagrama da Figura 3.2 é o AFN $N = (Q,\Sigma,\delta,q_0,F)$, tal que $Q = \{q_0,q_1,q_2\}$, $\Sigma = \{0,1\}$ e a função δ é definida abaixo:

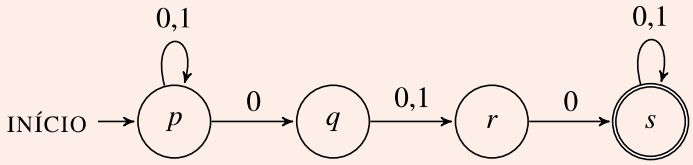
$\delta(q_0,0) = \{q_0,q_1\}$
$\delta(q_0,1) = \{q_0\}$
$\delta(q_1,1) = \{q_2\}$
$\delta(q_1,0) = \delta(q_2,0) = \delta(q_2,1) = \emptyset$ ■

Para simplificar, nós usaremos tanto 5-tuplas como tabelas de transições como definições formais para AFNs.

Exercício resolvido 3.3 Desenhe o diagrama do AFN definido pela seguinte tabela de transições:

	0	1
$\rightarrow p$	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\varnothing$
$*s$	$\{s\}$	$\{s\}$

Solução:



Uma maneira bastate útil para se descrever os ramos possíveis de computação de um AFN A com uma string w é a utilização de uma ávore, chamada de *árvore de computações possíveis de A com w*. Antes de definir este conceito, vamos apresentar a definição um pouco mais geral.

Definição 3.2.2 — (A,q,w)-árvores. Seja (A,q,w) uma tripla tal que A é um AFN, q um é estado de A e w um string do alfabeto de A. Uma (A,q,w)-*árvore* é uma árvore enraizada em q definida da seguinte maneira:

Base: $w = \varepsilon$

A árvore contém apenas o nó raíz q

Indução: $w \neq \varepsilon$

Suponha que w é uma string da forma ax, sendo que $a \in \Sigma$ e $x \in \Sigma^*$ e seja δ a função de transição do AFN. Se $\delta(q,a) = \{p_1, p_2, ..., p_k\}$, então então a árvore contém o nó q e, além disso, q possui os filhos $p_1, p_2, ..., p_k$ que são raízes de uma (A, p_i, x)-árvore.

Definição 3.2.3 — Árvore de computações possíveis. Seja $A = (Q, \Sigma, \delta, q_0, F)$ um AFN, $w \in \Sigma^*$. Uma *árvore de computações possíveis de A com w* é uma (A, q_0, w)-árvore.

Observe que o *nível 0* da árvore (i.e., a raiz) é o estado inicial do AFN e o nível i contém todos os estados que o AFN pode estar depois de i transições do autômato ao processar os i primeiros símbolos da string de entrada. Em particular, dada uma string de tamanho n, o AFN aceita a string se e somente se o n-ésimo nível da árvore contém **pelo menos um estado final**. No exemplo da Figura 3.3 o nível 5 da árvore contém o estado q_2. O fato de que neste nível existe apenas um nó que é um estado final significa que o AFN tem exatamente uma computação possível que aceita a string 01001. A computação é definida pela sequência de estados do caminho ligando a raíz ao estado final em questão.

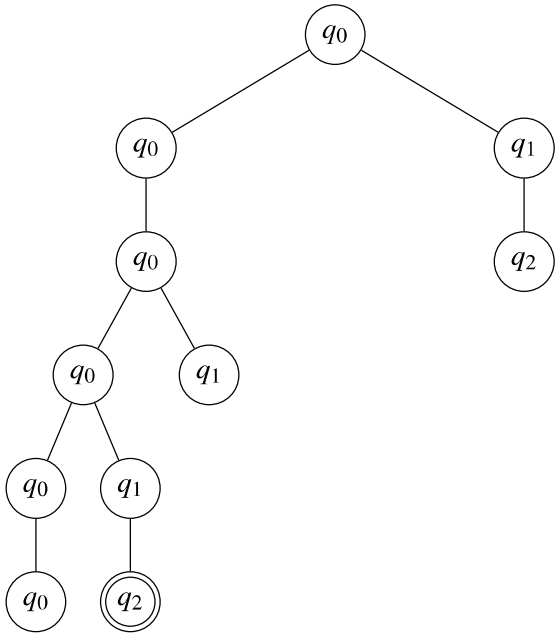


Figure 3.3: A árvore de computações possíveis do AFN N da Figura 3.2 com a string 01001.

Exercício resolvido 3.4 Apresente uma definição formal para um AFN cujo diagrama seja idêntico ao diagrama do AFD obtido na solução do Exercício 3.1.

Solução: Relembramos que o AFD do Exercício 3.1 é definido pela seguinte tabela:

	0	1
$\rightarrow p$	q	p
q	q	r
$*r$	r	r

Para obtermos um AFN cujo diagrama seja idêntido ao diagrama do AFD definido pela tabela acima, basta fazer o seguinte: se no AFD a função de transição é $\delta(x,y) = z$, defina o AFN de maneira que sua função de transição seja $\delta(x,y) = \{z\}$. A definição formal do AFN é dada pela seguinte tabela:

	0	1
$\rightarrow p$	$\{q\}$	$\{p\}$
q	$\{q\}$	$\{r\}$
$*r$	$\{r\}$	$\{r\}$

■

A Figura 3.4 mostra o diagrama do AFN do exercício 3.4.

Exercício 3.10 Desenhe a árvore de computações possíveis para o AFN do Exercício 3.4 com a string de entrada 111010. ■

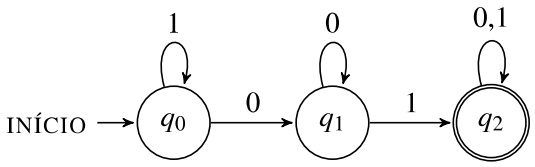


Figure 3.4: Um caso particular em que um AFN que se comporta de maneira determinística.

3.2.2 Aceitação e rejeição de strings por AFNs

A definição da função de transição estendida de um AFN é um pouco mais complicada que a definição que vimos no caso dos AFDs. A ideia básica é a seguinte: dado um estado q e uma string x, queremos saber qual é o conjunto de estados que o autômato pode estar após o processamento da string x se a computação começou no estado q.

Definição 3.2.4 Dado um AFN $N = (Q, \Sigma, \delta, q_0, F)$, definimos $\hat{\delta} : Q \times \Sigma^* \to \mathscr{P}(Q)$:

Base: $w = \varepsilon$.
$\hat{\delta}(q, \varepsilon) = \{q\}$

Indução: $|w| > 0$.
Seja $w \in \Sigma^*$ da forma xa, onde $x \in \Sigma^*$ e $a \in \Sigma$. Suponha $\hat{\delta}(q,x) = \{p_1, p_2, ..., p_k\}$. Então
$\hat{\delta}(q,w) = \bigcup_{i=1}^{k} \delta(p_i, a)$

Exercício 3.11 Calcule $\hat{\delta}(q_0, 00101)$ do AFN do Exemplo 3.2. ■

Definição 3.2.5 — Linguagem de um AFN. Se $N = (Q, \Sigma, \delta, q_0, F)$ é um AFN, então $L(N) = \{w \ : \ \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$ é a linguagem de N.

Se L é a linguagem de um AFN N, dizemos que N *decide* a linguagem L. Dizemos também que N *aceita* a linguagem L.

Exercício resolvido 3.5 Seja $D{=}(Q, \Sigma, \delta, q_0, F)$ um AFD. Apresente um AFN N, tal que $L(D){=}L(N)$.

Solução: A ideia é generalizar o que fizemos no Exercício Resolvido 3.4. O AFN N que aceita a mesma linguagem de D é o seguinte: $N = (Q, \Sigma, \delta', q_0, F)$, tal que a função δ' é a seguinte: se $\delta(x,y) = z$, então $\delta'(x,y) = \{z\}$. ■

3.2.3 Exercícios

Exercício 3.12 Suponha que queiramos mudar a definição de AFNs para que a *função de transição* tenha a seguinte forma: $\delta : Q \times \Sigma \to (\mathscr{P}(Q) \setminus \varnothing)$. Observe que autômatos segundo esta nova definição nunca morrem. Prove que dado um AFN N, podemos construir um AFN N' segundo nossa nova definição que aceita a mesma linguagem de N. ■

Exercício 3.13 No caso particular de AFNs que são determinísticos (i.e., AFNs que podem ser obtidos de AFDs como no Exercício 3.5), qual é o formato da árvore de computações possíveis? ■

3.3 Equivalência entre AFDs e AFNs

Algo que vamos lidar com frequência neste curso é a diferença de expressividade entre diferentes modelos de computação. Veremos no Capítulo 4 que existem algoritmos que, embora não possam ser expressos na forma de um AFD, podem ser expressos em outros modelos matemáticos que generalizam AFDs. Em tais casos normalmente dizemos que estes outros modelos são mais poderosos (ou mais expressivos) que o modelo de AFD.

> EXPRESSIVIDADE DE MODELOS DE COMPUTAÇÃO
>
> A ideia de expressividade de um modelo de computação é central em Teoria da Computação. Embora AFDs sejam capazes de realizar tarefas interessantes, como testar se um dado número é divisível por k (desde que k seja fixado *a priori*), veremos no Capítulo 4 que AFDs não são capazes de "testar primalidade" de números. De maneira formal, provaremos que não existe um AFD D tal que $L(D) = L_P$, ou seja, um AFD que tome uma string binária e decida se ela representa um número primo. Como sabemos que podemos escrever algoritmos (expressando eles em linguagem C, por exemplo) para testar primalidade de números, concluímos que o formalismo de AFD não é o formalismo mais geral possível para expressar algoritmos.
>
> A limitação de AFDs, neste momento do curso, não é o mais importante aqui. O que queremos observar aqui é que sempre que apresentamos um novo modelo de computação, como o modelo de AFNs, algo que sempre nos preocuparemos é como tal modelo se compara com outros modelos de computação já conhecidos, com AFDs.

No *Exercício Resolvido* 3.5 o objetivo foi mostrar que qualquer linguagem que um AFD reconheça, também pode ser reconhecida por um AFN. Isso significa que AFNs são pelo menos tão poderosos como AFDs. Isso é natural, pois AFNs são generalizações de AFDs. A pergunta óbvia que devemos fazer é: AFNs são *estritamente* mais poderosos que AFDs? Veremos nesta seção que a resposta é **não**. Ou seja, podemos mostrar que se uma linguagem pode ser aceita por um AFN, então existe algum AFD que aceita a mesma linguagem. Isso pode parecer um pouco surpreendente, pois AFNs, em algumas situações, tem a capacidade de advinhar qual transição deve fazer, uma capacidade que AFDs não tem.

3.3.1 Algoritmo de construção de conjuntos

O Algoritmo 1 apresentado a seguir recebe um AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ como entrada e retorna um AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ como saída tal que $L(N) = L(D)$. A ideia central é construir um AFD D que possa "simular" N usando a seguinte ideia: No momento em que o AFN N vai processar o i-ésimo símbolo da string de entrada $w = w_1....w_i...w_n$, os possíveis estados que N pode estar em tal instante são dados pelos nós do i-ésimo nível da árvore de computações possíveis de N com w. Seja P_i o conjunto dos estados do i-ésimo nível desta árvore e P_{i+1} o conjunto de nós do $(i+1)$-ésimo nível desta mesma árvore. O AFD D contruído pelo algoritmo, terá uma transição de um estado chamado P_i para outro chamado P_{i+1} (tais estados correspondem aos conjuntos de estados P_i e P_{i+1} da árvore de computações possíveis mencionados anteriormente), e esta transição terá o rótulo w_i. Em outras palavras, $\delta_D(P_i, w_i) = P_{i+1}$. O que o algoritmo faz é usar força bruta e aplicar a função δ_N em todas as combinações possíveis de pares (P_i, a), tal que $P_i \subseteq Q$, $a \in \Sigma$ para que possa determinar P_{i+1}.

Algorithm 1 Construindo um AFD a partir de um AFN.

AFN_AFD $(Q_N, \Sigma, \delta_N, q_0, F_N)$

1: $Q_D = \mathscr{P}(Q_N)$
2: **for all** $S \subseteq Q_D$ **do**
3: **for all** $a \in \Sigma$ **do**
4: Defina a função δ_D no par (S,a) da seguinte maneira: $\delta_D(S,a) = \bigcup_{p \in S} \delta_N(p,a)$
5: $F_D = \{S \in \mathscr{P}(Q_N) : S \cap F_N \neq \varnothing\}$
6: $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$
7: **Return** D

Vamos aplicar o algoritmo no autômato da Figura 3.2, cuja tabela de transições é a seguinte.

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$^*q_2$	$\varnothing$	$\varnothing$

Ao aplicarmos o Algoritmo 1 no AFN da tabela anterior, obtemos como saída o AFD definido pela seguinte tabela de transições:

	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$^*\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$^*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$^*\{q_1, q_2\}$	$\varnothing$	$\{q_2\}$
$^*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Observe que os estados do AFD obtido são conjuntos. Isso não é um problema, pois os estados de um AFD podem ser qualquer coisa, desde que os elementos da imagem da função δ sejam do mesmo tipo que os estados do AFD (note que neste caso os próprios estados são conjuntos). O que não devemos fazer é confundir com o caso dos AFNs em que os elementos da imagem da função δ são de natureza diferente dos estados.

3.3.2 Algoritmo de construção de conjuntos: versão melhorada

Alguns alunos mais observadores devem ter notado que não precisamos de todos os estados do AFD resultante. Precisamos apenas dos estados "alcançáveis" a partir do estado inicial $\{q_0\}$. Olhando a tabela, notamos que na linha do estado inicial $\{q_0\}$, atingimos o estado $\{q_0, q_1\}$ e o próprio estado $\{q_0\}$, dependendo do símbolo lido. Ao olharmos a tabela, na linha do estado $\{q_0, q_1\}$, vemos que atingimos o estado $\{q_0, q_2\}$ se o símbolo lido for 1. Se o autômato estiver no estado $\{q_0, q_2\}$, independente do símbolo lido, os únicos estados alcançáveis são estados que já mencionamos (i.e., $\{q_0\}$, $\{q_0, q_1\}$ e $\{q_0, q_2\}$). Portanto, como a computação sempre começa no estado $\{q_0\}$, os únicos três estados atingíveis para qualquer string de entrada são $\{q_1\}$, $\{q_0, q_1\}$ e

$\{q_0, q_2\}$. Com isso, podemos eliminar os estados desnecessários e simplificar a tabela da seguinte maneira:

	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$^*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Não vamos nos preocupar tanto com formalismo neste ponto, mas se quiséssemos fornecer uma definição formal para o coneceito de estado alcançável poderíamos usar a ideia de um vértice alcançável por uma busca em largura em um grafo direcionado. Pense no AFD obtido pela saída do Algoritmo 1 como um grafo direcionado em que os vértices são os estados e as arestas direcionadas do grafo ligam o vértice p ao vértice q caso ocorra que para algum símbolo a, $\delta(p,a) = q$. Com isso, os *estados alcançáveis* são os vértices alcançáveis por algum caminho direcionado a partir do vértice correspondente ao estado inicial.

Convenção: A partir de agora, sempre que formos construir um AFD equivalente a um dado AFN, vamos construir o AFD passo a passo, mantendo apenas os estados alcançáveis.

Teorema 3.3.1 Se o AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é obtido pelo Algoritmo 1 a partir de um AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, então $L(N) = L(D)$.

Exercício 3.14 Prove o Teorema 3.3.1
(Dica: prove por indução que $\forall w \in \Sigma^*$, $\hat{\delta}_D(q_0, w) \in F_N \Leftrightarrow \hat{\delta}_N(\{q_0\}, w) \cap F_N)$ ∎

O próximo teorema enuncia a equivalência entre AFDs e AFNs.

Teorema 3.3.2 Uma linguagem L é aceita por um AFD se e somente se L é aceita por um AFN.

Prova: Consequência do Teorema 3.3.1 e do *Exercício Resolvido* 3.5.

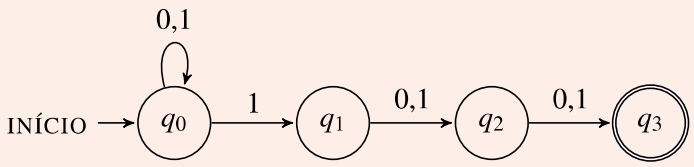
3.3.3 Exercícios

Exercício 3.15 Considere o o AFN dado pela tabela abaixo:

	0	1
$\rightarrow p$	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\varnothing$
$*s$	$\{s\}$	$\{s\}$

Apresente um AFD que aceite a mesma linguagem do AFN acima. ■

Exercício 3.16 Qual a linguagem aceita pelo AFN abaixo?



■

Exercício 3.17 Forneça um AFD equivalente ao AFN do Exercício 3.16. ■

3.4 Autômatos Finitos não Determinísticos com transições ε

Vamos considerar agora um diagrama de um AFN que contém algumas transições com o rótulo ε. Estas transições são chamadas de *transições* ε e um autômato que tenha tais transições será chamado de ε-AFN.

A ideia é tentar incrementar ainda mais o poder do nosso modelo de computação. A nova habilidade que vamos incluir em nosso autômato é a possibilidade de arbitrariamente fazer certas mudanças de estado sem que nenhum símbolo seja consumido. Para apresentar este conceito vamos usar um exemplo do livro de Hopcroft, Motwani e Ullman [HMU06]. A ideia é construir um ε-AFN que aceita strings que representam números decimais que estejam na forma descrita a seguir:

(1) O número pode ter sinal "+" ou "−", mas este sinal é opcional;
(2) Em seguida, há um string de dígitos (os dígitos da parte inteira do número), que também é opcional;
(3) Após esta sequência de dígitos há um ponto decimal "." obrigatório;
(4) Opcionalmente, há outra string de dígitos (os dígitos da parte decimal do número);
(5) Faz-se a restrição extra de que pelo menos uma das strings de dígitos de (2) e (4) é não seja vazia.

Observe que o alfabeto do autômato é $\{0,1,2,3,4,5,6,7,8,9,\cdot,+,-\}$. Alguns exemplos de strings aceitas são 5.72, +5.72, -12., -1. e -.5 enquanto alguns exemplos de strings não aceitam são +-5.72, -12, 8.0- e ..56. O autômato é o seguinte:

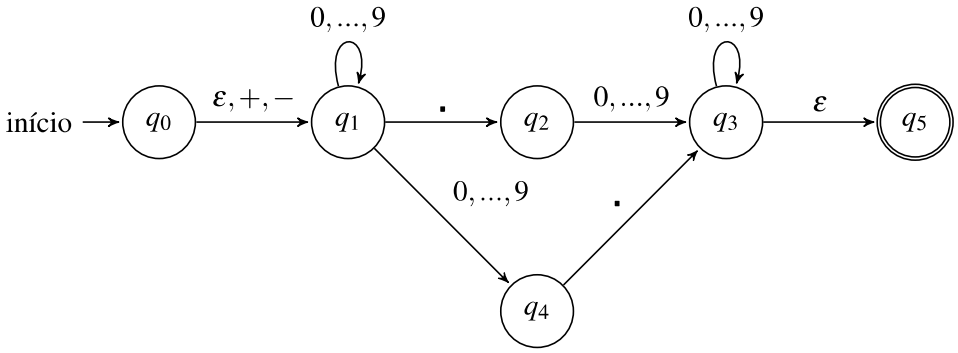


Figure 3.5: Um ε-AFN que aceita números decimais.

A definição formal para ε-AFNs é bastante semelhante a definição dos AFNs:

Definição 3.4.1 — ε-AFN. Um ε-AFN é uma 5-tupla $E = (Q, \Sigma, \delta, q_0, F)$ tal que cada um dos componentes tem a mesma interpretação que um AFN, exceto pelo fato que δ é uma função do tipo $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$, sendo que $\varepsilon \notin \Sigma$.

■ **Exemplo 3.3** O ε-AFN do exemplo anterior é $E = (\{q_0, ..., q_5\}, \{., +, -, 0, ..., 9\}, \delta, q_0, \{q_5\})$ tal que a tabela de transições de δ é a seguinte:

	ε	*sinal*	.	dígito
q_0	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
$*q_5$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

■

Observe que no Exemplo 3.3, se quiséssemos ser totalmente rigorosos, a tabela de transições teria que ter 14 colunas: exatamente 13 colunas para o alfabeto do autômato (uma coluna para cada um dos 10 dígitos, duas colunas para os sinais e uma para o ponto) além de uma coluna extra para ε). Entretanto, para simplificar, agrupamos todos os dígitos em apenas uma coluna e os dois possíveis sinais em uma outra coluna, pois transições para cada elemento destes grupos são as mesmas.

O nosso próximo passo agora é definir o conceito de função de transição estendida. Para que possamos apresentar tal definição, vamos antes definir o que é o ε-*Fecho* de um estado. A ideia é que o fecho de um estado q é o conjunto de todos os possíveis estados que podem ser atingidos a partir de q (incluindo o próprio q) utilizando-se uma quantidade arbitrária de trasições ε.

Definição 3.4.2 — ε-Fecho de estados. Seja um ε-AFN com conjunto de estados Q e seja $q \in Q$. Vamos definir o conjunto ε-Fecho(q) de maneira indutiva:

Base: $q \in \varepsilon$-Fecho(q)

Indução: se $p \in \varepsilon$-Fecho(q) e $r \in \delta(p, \varepsilon)$, então $r \in \varepsilon$-Fecho(q).

Agora que temos a Definição 3.4.2 em mãos, podemos definir o conceito de função de transição

estendida de um ε-AFN. A ideia é parecida com a definição de função de transição estendida de um AFN, com o cuidado extra de adicionar os estados que podem ser atingidos por trasições ε.

Definição 3.4.3 — Função de transição estendida em ε-AFNs. Dado um ε-AFN $N = (Q, \Sigma, \delta, q_0, F)$, definimos $\hat{\delta} : Q \times \Sigma^* \to \mathscr{P}(Q)$ indutivamente:

Base: $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-Fecho}(q)$

Indução: Seja $w \in \Sigma^*$ da forma xa, onde $x \in \Sigma^*$ e $a \in \Sigma$. Suponha $\hat{\delta}(q,x) = \{p_1, p_2, ..., p_k\}$.

Suponha $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, ..., r_m\}$

Então

- $\hat{\delta}(q,w) = \bigcup_{j=1}^{m} \varepsilon\text{-Fecho}(r_j)$

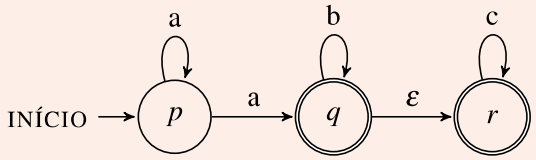
O próximo passo é definir o que é a linguagem de um ε-AFN.

Definição 3.4.4 — Linguagem de ε-AFNs. Seja $E = (Q, \Sigma, \delta, q_0, F)$ um ε-AFN. Definimos $L(E) = \{w \ : \ \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$ como sendo a linguagem de E.

Dada uma linguagem L, se existe um ε-AFN tal que $L = L(E)$, então dizemos que E aceita L.

Exercício resolvido 3.6 Seja $\Sigma = \{a, b, c\}$. Apresente um ε-AFN para a linguagem das strings sobre Σ que têm a forma $a^n b^m c^l$, tal que $n > 0$, $m \geq 0$, $l \geq 0$.

Solução:



■

3.5 Equivalência entre AFDs e ε-AFNs

Nesta seção vamos construir um AFD D a partir de um ε-AFN $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. A ideia aqui é praticamente a mesma da que vimos na seção 3.3. A única diferença é que temos que tomar cuidado com as transições ε.

Algorithm 2 Construindo um AFD a partir de um ε-AFN

AFN_AFD $(Q_E, \Sigma, \delta_E, q_0, F_E)$

1: $Q_D = \mathscr{P}(Q_E)$
2: $q_0 = \varepsilon$-Fecho(q_0)
3: **for all** $S \subseteq Q_N$ **do**
4: **for all** $a \in \Sigma$ **do**
5: $R = \bigcup_{q_i \in S} \delta_E(q_i, a)$
6: Defina a função δ da seguinte maneira: $\delta_D(S,a) = \bigcup_{r_j \in R} \varepsilon$-Fecho$(r_j)$
7: $F_D = \{S \in Q_D \ : \ S \cap F_E \neq \varnothing\}$
8: Elimine os estados não alcançáveis
9: **Return** $(Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$

Convenção: De maneira semelhante ao Algoritmo 1, a partir de agora sempre que formos construir um AFD equivalente a um dado ε-AFN, vamos construir o AFD passo a passo, mantendo apenas os estados alcançáveis.

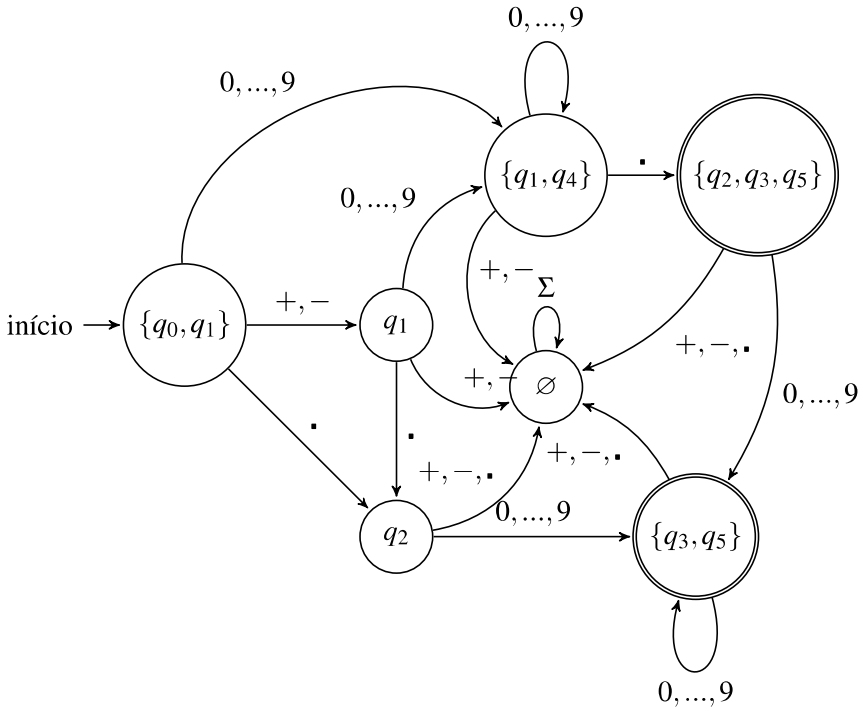
Teorema 3.5.1 Se o AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é obtido pelo Algoritmo 2 quando toma como entrada o ε-AFN $E = (Q_N, \Sigma, \delta_N, q_0, F_N)$, então $L(D) = L(N)$.

Teorema 3.5.2 Uma linguagem é aceita por um AFD se e somente se é aceita por um ε-AFN.

Vamos construir um AFD equivalente ao ε-AFN da Figura 3.5 usando o Algoritmo 2. A tabela do AFD que o algoritmo retorna é o seguinte:

	$+,-$	$0,\ldots,9$	.
$\rightarrow \{q_0, q_1\}$	$\{q_1\}$	$\{q_1, q_4\}$	$\{q_2\}$
$\{q_1\}$	$\varnothing$	$\{q_1, q_4\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\varnothing$	$\{q_1, q_4\}$	$\{q_2, q_3, q_5\}$
$\{q_2\}$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$^*\{q_2, q_3, q_5\}$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$
$^*\{q_3, q_5\}$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$

O diagrama do AFD obtido pelo Algoritmo 2 é o seguinte:



3.5.1 Exercícios

Exercício 3.18 Considere o ε-AFN dado pela tabela abaixo:

	ε	a	b	c
$\to p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\varnothing$
$*r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

(a) Desenhe o diagrama do ε-AFN
(b) Calcule o ε-fecho de cada estado.
(c) Forneça todas strings w tal que $|w| \leq 2$ aceitas pelo autômato.
(d) Converta o ε-AFN em um AFD.

■

Exercício 3.19 Repita a questão 1 para o ε-AFN dado pela tabela abaixo:

	ε	a	b	c
$\to p$	$\{q,r\}$	$\varnothing$	$\{q\}$	$\{r\}$
q	$\varnothing$	$\{p\}$	$\{r\}$	$\{p,q\}$
$*r$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

■

Exercício 3.20 Apresente um AFD que decida a mesma linguagem do ε-AFN do Exercício Resolvido 3.6 ∎

Exercício 3.21 Apresente a árvore de computações possíveis para ε-AFN da figura 3.5 com a string -0.95. ∎

3.6 Expressões Regulares (ERs)

Uma *Expressão regular* é uma expressão matemática que representa uma linguagem. O nosso objetivo agora é definir indutivamente quais expressões matemáticas são consideradas expressões regulares válidas. Algo bastante importante em nossa definição é que no mesmo momento tivermos estabelecido que uma certa expressão regular R é válida, também vamos estabelecer exatamente qual linguagem $L(R)$ corresponde a tal expressão.

Definição 3.6.1 — Expressões Regulares (ER).
Seja Σ um alfabeto qualquer. Uma expressão matemática consistindo de símbolos de Σ, parêntesis e '$+$' e '$*$' é uma expressão regular se tem forma descrita recursivamente a seguir:

Base:
(1) As expressões $\underline{\varepsilon}$ e $\underline{\varnothing}$ são expressões regulares que correspondem, respectivamente, às linguagens $L(\underline{\varepsilon}) = \{\varepsilon\}$ e $L(\underline{\varnothing}) = \varnothing$.
(2) Para todo símbolo a de Σ, a expressão $\underline{a}$ é uma expressão regular correspondente a linguagem $L(\underline{a}) = \{a\}$.

Passo indutivo:
(1) Se E e F são expressões regulares, então $E+F$ é uma expressão regular representando a união de $L(E)$ e $L(F)$. Isto é, $L(E+F) = L(E) \cup L(F)$.
(2) Se E e F são expressões regulares, então EF é uma expressão regular denotando a concatenação de $L(E)$ e $L(F)$. Isto é, $L(EF) = L(E)L(F)$.
(3) Se E é uma expressão regular, então E^* é uma expressão regular que representa o fechamento de $L(E)$. Isto é, $L(E^*) = (L(E))^*$.
(4) Se E é uma expressão regular, então (E) também é uma expressão regular, representando a mesma linguagem que E representa. Isto é, $L((E)) = L(E)$.

Terminologia 3.1. *As expressões regulares definidas na base da definição são chamadas de expressões regulares elementares. As expressões regulares definidas no passo indutivo são chamadas de expressões regulares compostas.*

Exercício resolvido 3.7 A expressão regular para a linguagem L das "strings contendo 0's e 1's alternados" é $\underline{(01)^* + (10)^* + 0(10)^* + 1(01)^*}$. Justifique isto passo a passo.

Solução: Vamos ser bastante cuidadosos e resolver passo a passo.

Passo 1: Segundo a Base da Definição 3.6.1 (item 2), se $0 \in \Sigma$, então $\underline{0}$ é uma expressão regular válida. Ainda segundo esta definição, a expressão $\underline{0}$ representa a linguagem $\{0\}$.

Passo 2: Usando o mesmo argumento do Passo 1, concluímos que $\underline{1}$ é uma expressão regular válida que representa a linguagem $\{1\}$.

Passo 3: Até este ponto já sabemos que $\underline{0}$ e $\underline{1}$ são expressões regulares válidas que representam as linguagens $\{0\}$ e $\{1\}$, respectivamente. Segundo a Definição 3.6.1 (Indução, item 2),

concluímos que $\underline{01}$ também é uma expressão válida. A linguagem que esta expressão representa é $L(\underline{01}) = \{0\} \cdot \{1\} = \{01\}$.

Passo 4: A partir da expressão obtida no passo Passo 3, podemos aplicar a Definição 3.6.1 (Indução, item 3) e concluir que $\underline{(01)^*}$ é uma expressão válida, e $L(\underline{(01)^*}) = \{\varepsilon, 01, 0101, 010101, ...\}$.

Passo 5: Usando argumentos semelhantes aos anteriores, concluímos que $\underline{(10)^*}$ também é uma expressão válida, e $L(\underline{(10)^*}) = \{\varepsilon, 10, 1010, 101010, ...\}$.

Passo 6: Pelos Passos 4 e 5, $\underline{(01)^*}$ e $\underline{(10)^*}$ são expressões válidas que representam as linguagens $\{\varepsilon, 01, 0101, 010101, ...\}$ e $\{\varepsilon, 10, 1010, 101010, ...\}$, respectivamente. Pela Definição 3.6.1 (Indução, item 1), $\underline{(01)^* + (10)^*}$ é uma expressão válida e $L(\underline{(01)^* + (10)^*}) = \{\varepsilon, 01, 0101, 010101, ...\} \cup \{\varepsilon, 10, 1010, 101010, ...\} = \{\varepsilon, 01, 10, 0101, 1010, 010101, 1010101, ...\}$.

Passo 7: Podemos concluir que $\underline{0(10)^*}$ é uma expressão regular a partir do fato que $\underline{0}$ e $\underline{(10)^*}$ são expressões válidas (concluímos isso nos Passos 1 e 5) e a linguagem que expressão representa é $\{0, 010, 01010, 0101010, ...\}$. Usando argumentos semelhantes, podemos concluir que $\underline{1(01)^*}$ é uma expressão regular válida e $L(\underline{1(01)^*}) = \{1, 101, 10101, 1010101, ...\}$. Com isso, podemos aplicar a Definição 3.6.1 (Indução, item 1) nas expressões $\underline{0(10)^*}$ e $\underline{1(01)^*}$ para concluir que $\underline{0(10)^* + 1(01)^*}$ é uma expressão regular válida representando a linguagem $\{0, 010, 01010, 0101010, ...\} \cup \{1, 101, 10101, 1010101, ...\}$.

Passo 8: Aplicando a Definição 3.6.1 (Indução, item 1) nas expressões obtidas no Passo 6 e 7, concluímos que $E = \underline{(01)^* + (10)^* + 0(10)^* + 1(01)^*}$ é uma expressão regular válida e $L(E) = \{\varepsilon, 0, 1, 01, 10, 010, 101, 0101, 1010, 010101, ...\}$. ■

Assim como ocorre em expressões aritméticas, operadores em expressões regulares obedecem regras de precedência. As regras são as seguintes:

- O operador * tem precedência mais alta e, portanto, deve ser o primeiro a ser aplicado. Com isso, por exemplo, a expressão $\underline{01^*}$ é equivalente a $\underline{0(1)^*}$;
- O operador com segunda maior precedência é o operador de concatenação. Com isso, por exemplo, a expressão $\underline{a + bc}$ é equivalente a $\underline{a + (bc)}$;
- O operador de menor precedência é o operador $+$;
- Como de custume, usamos parentesis para alterar a precedência de operadores.

Exercício 3.22 Seja $\Sigma = \{a, b, c\}$. Apresente a expressão regular para a linguagem das strings sobre Σ que começam e terminam com a e têm pelo menos um b. ■

Exercício 3.23 Seja $\Sigma = \{a, b\}$. Apresente a expressão regular para a linguagem das strings sobre Σ que tem tamanho ímpar. ■

Algo importante a ser mencionado é que, embora seja verdade que dada uma expressão regular, existe exatamente uma linguagem que corresponde a tal expressão, o oposto não é verdade. Isto é, dada uma linguagem L, não é verdade que existe exatamente uma expressão regular para L. De fato, dada uma linguagem L pode existir mais de uma expressão regular R tal que $L(R) = L$ (ou mesmo pode não existir nenhuma expressão regular para L). A exata relação entre expressões regulares e linguagens é estabelecida pelo seguinte teorema, que é central em teoria de linguagens regulares:

Teorema 3.6.1 Uma linguagem L é regular se e somente se existe uma expressão regular R tal que $L = L(R)$.

Para demonstrar o Teorema 3.6.1, precisamos demonstrar que as afirmações (1) e (2) abaixo são verdadeiras:

(1) Se L é uma linguagem regular, então existe uma expressão regular R tal que $L = L(R)$.
(2) Se R é uma expressão regular qualquer, então $L(R)$ é uma linguagem regular.

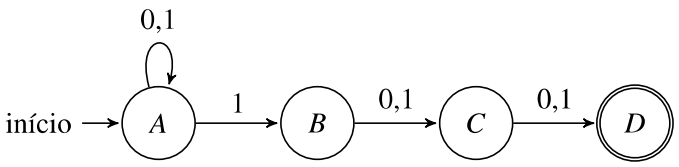
Apresentamos as provas das Afirmações (1) e (2) nas Seções 3.6.1 e 3.6.2, repectivamente.

3.6.1 Expressões regulares para linguagens de autômatos

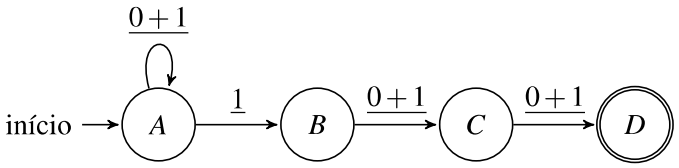
Nesta seção vamos demonstrar a seguinte afirmação: *se L é uma linguagem regular, então existe uma expressão regular R tal que $L = L(R)$*. Por definição, uma linguagem regular é uma linguagem aceita por um AFD. Entretanto, já vimos que AFDs e ε-AFNs aceitam exatamente o mesmo conjunto de linguagens (ver Teorema 3.5.2) e, portanto, para demonstrarmos a afirmação acima, basta que provemos o seguinte teorema:

Teorema 3.6.2 Seja $L(E)$ a linguagem de um $\varepsilon-$AFN E qualquer. Então existe uma expressão regular R tal que $L(R) = L(E)$.

Para provar o teorema acima, o que faremos é apresentar um algoritmo que, dado como entrada um $\varepsilon-$AFN E qualquer é capaz de produzir como saída uma expressão regular R tal que $L(R) = L(E)$. Para que possamos entender como funciona tal algoritmo, vamos, incialmente, considerar um caso simples. Seja E o autômato não determinístico abaixo:

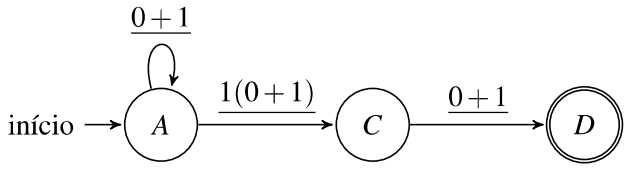


A partir de E, vamos apresentar um *autômato generalizado* E_0. A ideia é que um autômato generalizado seja uma variação dos autômatos que já conhecemos, mas que tenha expressões regulares ao invés símbolos em suas transições. O autômato generalizado E_0 é apresentado abaixo:



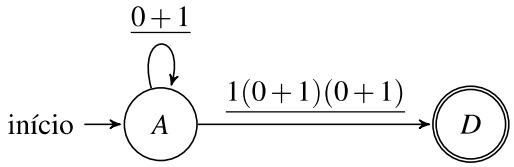
Observe que agora temos expressões regulares como rótulos das transições. Por exemplo, observe o conjunto de símbolos $\{0,1\}$ que figuravam como rótulo da transição do estado C para o estado D foram trocados pela expressão regular $\underline{0+1}$. Note, em particular, que $L(\underline{0+1}) = \{0,1\}$. De maneira mais geral, a ideia é que se para se Q e Q' são dois estados do autômato original E tal que o conjunto $S_{Q,Q'}$ de símbolos figure como rótulo na transição $Q \to Q'$, fazemos que no autômato generalizado E_0 a expressão regular $R_{Q,Q'}$, tal que $L(R_{Q,Q'}) = S_{Q,Q'}$, figure como rótulo da transição correspondente.

A ideia chave agora é produzir sequência de autômatos generalizados $E_0, E_1, E_2, \ldots$ de forma que todos sejam equivalentes, mas que tenham cada vez menos estados. Por exemplo, considere o autômato generalizado E_1 abaixo:



Observe que obtemos E_1 a partir de E_0 da seguinte forma: removemos o estado B e todas as transições que envolviam este estado (ou seja, $A \to B$ e $B \to C$) e incluímos a transição $A \to C$ com uma nova expressão como rótulo: $\underline{1(0+1)}$. A ideia é que a expressão $\underline{1(0+1)}$ é a concatenação das expressões $\underline{1}$ e $\underline{(0+1)}$. A ideia fundamental é que o conjunto de strings que faziam E_1 sair de A e chegar em C é representado pela expressão $\underline{1(0+1)}$, que agora figura no rótulo de $A \to C$, ou seja, strings de dois bits, tal que o primeiro bit é obrigatoriamente 1.

Agora o próximo passo é remover o estado C de E_1 e obter o autômato generalizado E_2 abaixo:

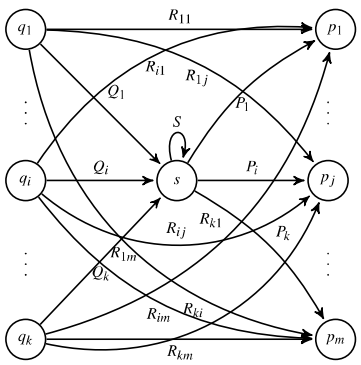


No caso do autômato acima, fica claro que as strings aceitas, isto é, as que fazem o autômato sair de A e atingir D são da forma xy, onde $x \in \{0,1\}^*$ e $y \in L(\underline{1(0+1)(0+1)})$. A ideia é que o autômato processe x fazendo transições do tipo $A \to A$ e, quando faltarem os três últimos símbolos da string de entrada, use a transição $A \to B$ para processar a string y restante.

Fazendo a simplificação dos autômatos generalizados

No exemplo visto anteriormente, as "atualizações" que fazemos para obter um autômato generalizado E_{i+1} a partir de um autômato E_i são razoavelmente simples. De maneira geral, ao removermos um estado s de E_i, precisamos observar todo par de estado q_i, p_j tal que ambas $p_i \to s$ e $s \to p_j$ sejam transições.

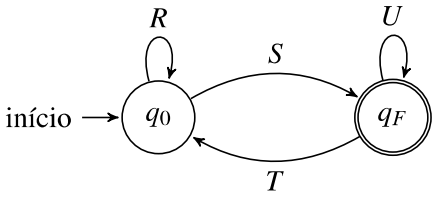
De acordo com a figura abaixo, suponha que os rótulos das transições $p_i \to s$ e $s \to p_j$ no autômato E_i são, respectivamente, Q_1 e P_1. Suponha também que o rótulo da transição $p_i \to q_j$ seja R_{ij} e da transição $s \to s$ seja S (lembrando que se alguma transição não existe no autômato, podemos pensar que ela tem rótulo $\underline{\varnothing}$). Com isso, ao remover o estado s o rótulo da transição de $p_i \to q_j$ no autômato E_{i+1} passa a ser $R_{ij} + Q_i S^* P_j$.



Usando a ideia acima para fazer as atualizações das transições, vamos ver o algoritmo. Entretanto, precisamos antes tomar cuidado como o estado inicial e os estados finais do autômato. Vamos começar apresentando o algoritmo para o caso que o autômato em questão tenha apenas um estado final. Vamos considerar dois casos: no primeiro caso, o estado inicial não é o estado final e no segundo caso o estado inicial e o estado final coincidem.

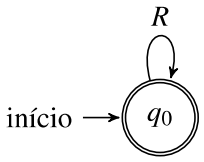
Caso 1: Algoritmo para obter ER a partir de um ε-AFN $(Q,\Sigma,\delta,q_0,\{q_F\})$, com $q_0 \neq q_F$:

1. Troque os rótulos das transições de E por expressões regulares equivalentes de forma que agora E seja um autômato generalizado
2. A cada passo remova um estado $q \in Q \setminus \{q_0, q_F\})$ e atualize as transições e as espressões regulares dos rótulos do autômato
3. Quando restar apenas o estado inicial q_0 e o estado final q_F, como o da figura abaixo, a expressão regular final é $(R+SU^*T)^*SU^*$



Caso 2: Algoritmo para obter ER a partir de ε-AFN (Q,Σ,δ,q_0,F) com $|F|=1$ e $q_0 \in F$:

1. Troque os rótulos das transições de E por expressões regulares equivalentes de forma que agora E seja um autômato generalizado
2. A cada passo remova um estado $q \in Q \setminus \{q_0\}$ e atualize as transições e as espressões regulares dos rótulos do autômato
3. Quando restar apenas o estado inicial (e final) q_0, como o da figura abaixo, a expressão regular final é R^*.



A ideia agora é usar o algoritmos vistos para os dois casos anteriores e apresentar um algoritmo que funciona para qualquer autômato.

Caso geral: Algoritmo para obter ER a partir de ε-AFN $E=(Q,\Sigma,\delta,q_0,F)$ qualquer

Suponha que $F=\{f_1,f_2,...,f_k\}$. Agora tome uma string $w \in L(E)$ e seja f_i o estado atingido pelo autômato quando w é aceita, tomando i o menor possível[3]. A ideia chave é que o autômato $E_i=(Q,\Sigma,\delta,q_0,\{q_i\})$ tem a seguinte propriedade:

(1) E_i também aceita a string w;

(2) E_i não aceita nenhuma string que não seja aceita por E.

Usando essa ideia, o algoritmo para obter a expressão regular de E é o seguinte:

- A partir de E, crie k autômatos $E_1,E_2,...,E_k$, onde E_i é o o autômato $(Q,\Sigma,\delta,q_0,\{q_i\})$
- Encontre as expressões regulares $R_1,R_2,...,R_k$ para cada autômato E_1 usando os algoritmos para os casos especiais de autômatos com apenas um estado final
- A partir daí, e ER para E é $R_1+R_2+...+R_k$

[3]Note que como o autômato não é determinístico, podem haver vários estados de aceitação possível, mas precisamos de apenas um deles e, arbitrariamente, tomamos o estado f_i para o menor índice i possível.

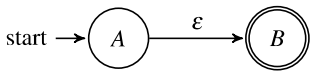
3.6.2 Autômatos para linguagens de expressões regulares

Nesta seção, nosso objetivo é demonstrar a seguinte afirmação: *Se R é uma expressão regular qualquer, então $L(R)$ é uma linguagem regular*. Isto é enunciado de maneira precisa abaixo:

Teorema 3.6.3 Seja $L(R)$ é a linguagem representada por uma expressão regular R qualquer. Então existe um $\varepsilon-$AFD E tal que $L(E) = L(R)$.

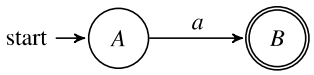
Para provarmos o teorema, vamos mostar que o $\varepsilon-$AFD com alfabeto Σ que aceita a linguagem de R, de mesmo alfabeto, pode ser construído recursivamente a partir da própria definição recursiva para as expressões regulares (Definição 3.6.1).

- Se R é uma expressão regular elementar, então $\varepsilon-$AFD tem uma das três formas abaixo:



Se $R = \underline{\varepsilon}$, então o autômato acima aceita a linguagem $L(\underline{\varepsilon}) = \{\varepsilon\}$.

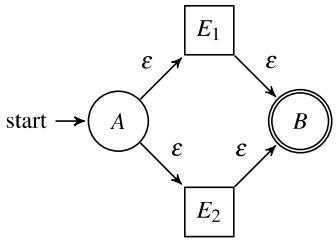
Se $R = \underline{\emptyset}$, então o autômato acima aceita a linguagem $L(\underline{\emptyset}) = \emptyset$.



Se $R = \underline{a}$, para algum $a \in \Sigma$, então o autômato acima aceita $L(\underline{a}) = \{a\}$.

- Se R é uma expressão regular composta, então $\varepsilon-$AFD pode ser construído recursivamente usando a seguinte estratégia:

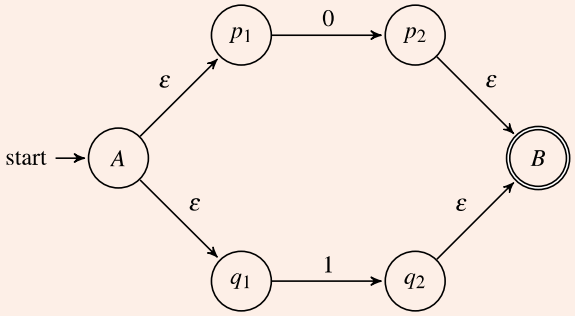
Caso 1. Suponha que e a expressão regular tem a forma composta $R = R_1 + R_2$. Suponha por indução que os $\varepsilon-$AFDs E_1 e E_2 aceitam, respectivamente, $L(R_1)$ e $L(R_2)$. O diagrama abaixo descreve a forma para o $\varepsilon-$AFD E que aceita a expressão composta R:



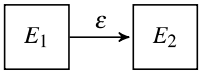
Observe o seguinte na construção acima: Os nós E_1 e E_2 representam autômatos para as linguagens $L(R_1)$ e $L(R_2)$. A transição $A \to R_1$ deve ser entendida como saindo do estado A e tendo destino o estado inicial do autômato E_1 e a transição $E_1 \to B$ deve ser entendida como saindo do estado final de E_1 e tendo como destino B. A mesma ideia vale para as transições $A \to R_2$ e $E_2 \to B$. Note também que nesta construção os estados que originalmente eram iniciais em E_1 e E_2 não são mais iniciais no autômato composto resultante.

Exercício resolvido 3.8 Apresente um autômato para aceitar a linguagem da expressão regular $R = \underline{0+1}$ sabendo que os dois autômatos E_1 e E_2 abaixo aceitam as linguagens das expressões $\underline{0}$ e $\underline{1}$, respectivamente:

Solução: Usando a construção para o Caso 1, obtemos abaixo o autômato E que aceita a linguagem da expressão $\underline{0+1}$:



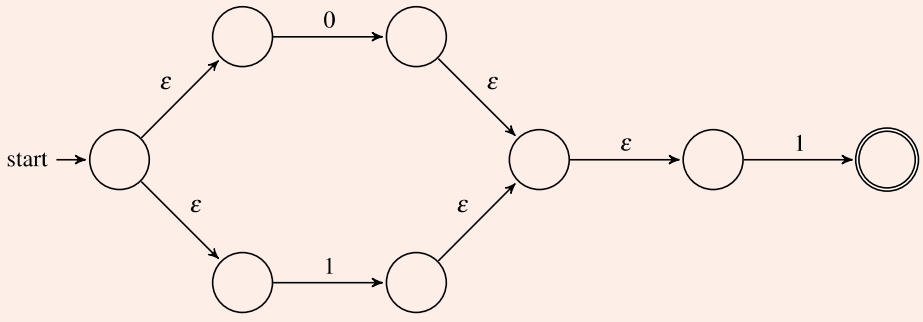
Caso 2. Suponha que e a expressão regular tem a forma composta $R = R_1R_2$. Suponha por indução que os $\varepsilon-$AFDs E_1 e E_2 aceitam, respectivamente, $L(R_1)$ e $L(R_2)$. O diagrama abaixo descreve a forma para o $\varepsilon-$AFD E que aceita a expressão composta R:



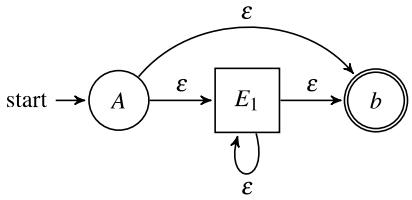
Observe o seguinte no esquema da construção acima: Os nós E_1 e E_2 representam autômatos para as linguagens $L(R_1)$ e $L(R_2)$. A transição $E_1 \to E_2$ deve ser entendida como saindo do final do autômato E_1 e tendo destino o estado inicial do autômato E_2. Note também que nesta construção o estado que originalmente era inicial em E_2 não é mais inicial no autômato composto resultante.

Exercício resolvido 3.9 Apresente um autômato para aceitar a linguagem de $R = \underline{(0+1)1}$.

Solução: Sabendo que os autômatos E e E_1 apresentados no Exercício Resolvido 3.8 aceitam as linguagens das expressões $\underline{(0+1)}$ e $\underline{1}$ obtemos o seguinte autômato para R:



Caso 3. Suponha que e a expressão regular tem a forma composta $R = R_1^*$. Suponha por indução que o $\varepsilon-$AFD E_1 aceita $L(R_1)$. O diagrama abaixo descreve a forma para o $\varepsilon-$AFD E que aceita a expressão composta R:

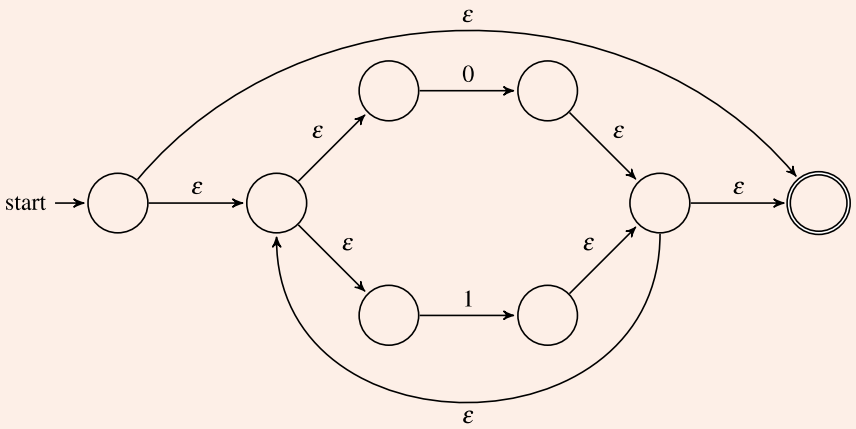


Observe o seguinte no esquema da construção acima: O nó E_1 representa o autômato para a linguagem $L(R_1)$. A transição $E_1 \to E_1$ deve ser entendida como saindo do final do autômato E_1 e tendo destino o estado inicial do autômato E_1. A transição $A \to E_1$ sai de A e chega no estado inicial de E_1 e a transição $E_1 \to B$ sai do estado final de E_1 e chega em B. Note também que nesta construção o estado que originalmente era inicial em E_1 não é mais inicial no autômato composto resultante.

Exercício resolvido 3.10 Apresente um autômato para aceitar a linguagem de $R = \underline{(0+1)^*}$.

Solução: Sabendo que o autômato E apresentados na solução do Exercício Resolvido 3.8 aceita a linguagem da expressão $\underline{(0+1)}$ obtemos o seguinte autômato composto para R:

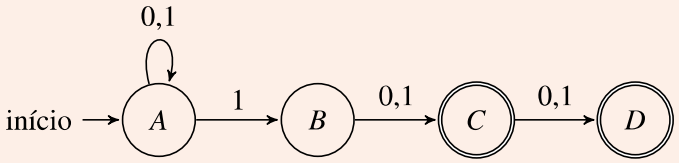
Solução:



■

3.6.3 Exercícios

Exercício 3.24 Obtenha uma expressão regular que represente a linguagem do seguinte ε-AFN:



■

Exercício 3.25 Obtenha um ε-AFN cuja linguagem seja a mesma representada pela expressão regular $(0+1)^*1(0+1)$. ■

Exercício 3.26 Considere os alfabetos $\Sigma_1 = \{a,b,c\}$ e $\Sigma_2 = \{0,1\}$. Forneça expressões regulares para as seguintes linguagens:

(a) $L \subseteq \Sigma_1^*$ tal que toda string de L tem pelo menos um a e um b.
(b) Conjunto de strings sobre Σ_2 tal que o terceiro símbolo de trás para frente é 1.
(c) $L \subseteq \Sigma_2^*$ definido por $L = \{w \mid w$ tenha um número par de 0's e um número par de 1's$\}$.

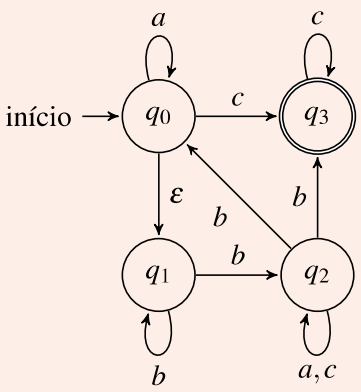
■

Exercício 3.27 Dado o AFD abaixo, encontre uma expressão regular equivalente. Você deve mostrar o desenvolvimento passo a passo de solução.

	0	1
$\rightarrow^* p$	s	p
q	p	s
r	r	q
s	q	r

■

Exercício 3.28 Considere o ε-AFN $E = (\{q_0,q_1,q_2,q_3\},\{a,b,c\},\delta,q_0,\{q_3\})$ abaixo:

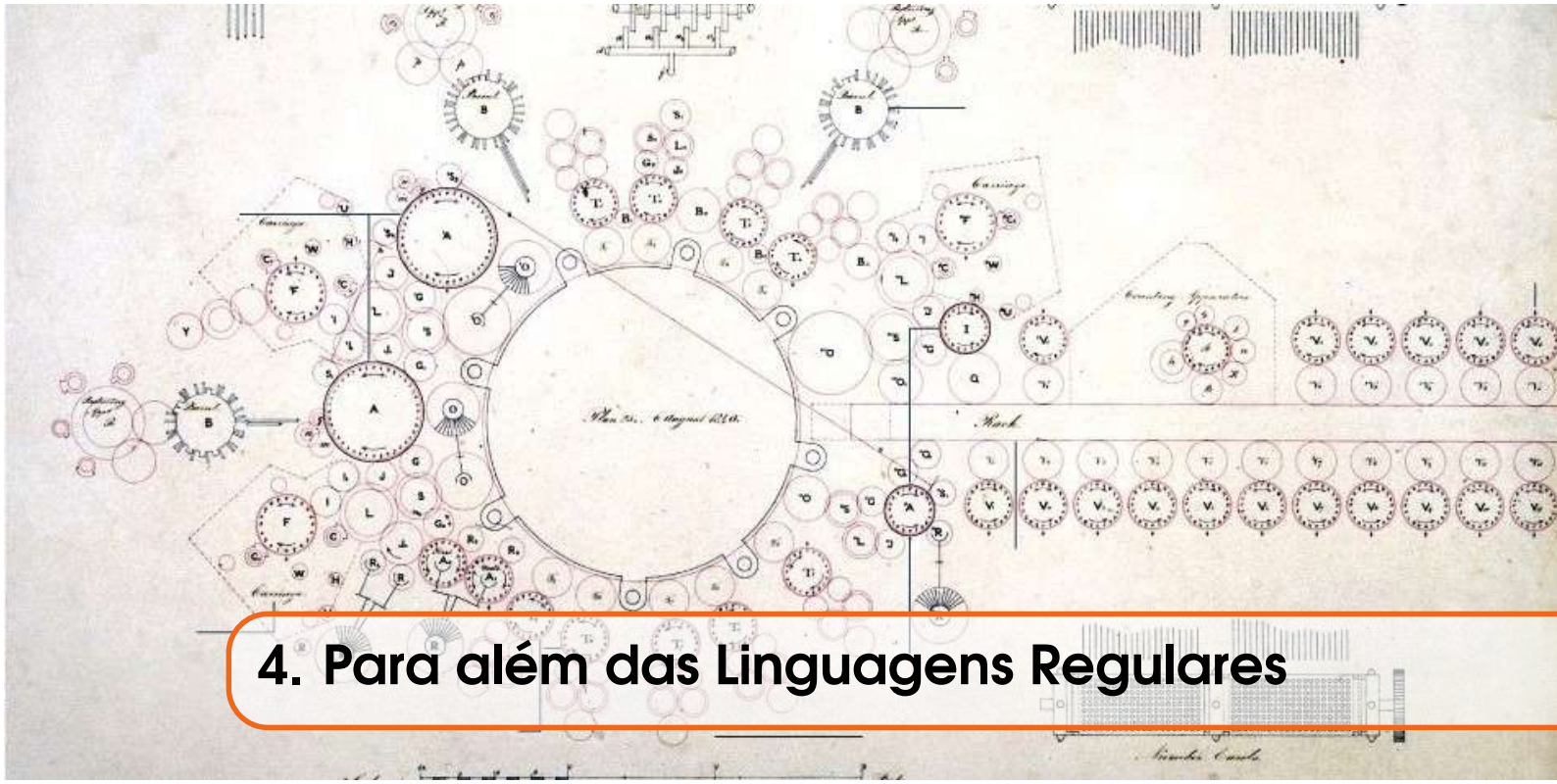


(a) Apresente a tabela de transições da função δ do autômato E.

(b) Forneça uma expressão regular R tal que $L(R) = L(E)$. Mostre passo a passo o desenvolvimento da sua solução. ■

Exercício 3.29 Forneça um ε-AFN que aceite a mesma linguagem da expressão regular $\underline{(00+11)^* + (111)^*0}$. ■

Exercício 3.30 Construa um ε-AFN que aceite a mesma linguagem da expressão regular $\underline{00(0+1)^*}$. ■



4. Para além das Linguagens Regulares

Existem linguagens não regulares? Nas seções anteriores nós já antecipamos que a resposta é sim. Veremos neste capítulo que algumas linguagens extremamente simples como, por exemplo, $L_{01} = \{0^n 1^n \mid n \geq 1\}$, não são regulares.

Dada uma linguagem L, se quisermos mostrar que L é regular, basta explicitamente apresentarmos um AFD a aceite. Entretanto, se quisermos mostrar que L não é regular, temos que provar que não existe nenhum AFD que aceite L. Ou seja, precisamos usar um argumento que exclua logicamente a possibilidade de que cada um dos infinitos possíveis AFDs tenha a propriedade de ser um AFD que aceite L. A demonstração deste tipo de afirmação tende a ser mais difícil de se obter. Nosso primeiro objetivo neste capítulo é apresentar uma ferramenta matemática, conhecida como *Lema do Bombeamento*, que será extremamente útil para demonstrarmos que certos AFDs não existem.

4.1 O Lema do Bombeamento para Linguagens Regulares

O seguinte lema será bastante útil para provarmos que certas linguagens não são regulares:

Lema 4.1.1 — Lema do Bombeamento (LB). Seja L uma linguagem regular. Então existe uma constante t tal que $\forall w \in L$, com $|w| \geq t$, o seguinte é verdadeiro:

$\exists x, y, z \in \Sigma^*$ tal que $w = xyz$ e as três condições abaixo são satisfeitas:

(1) $y \neq \varepsilon$ (2) $|xy| \leq t$ (3) $\forall k \geq 0, xy^k z \in L$

No decorrer deste capítulo nos referiremos ao Lema 4.1.1 como Lema do Bombeamento ou simplesmente LB. Agora vamos mostrar como podemos fazer uso do LB para mostrar que uma linguagem não é regular. A vantagem de se usar o LB é que não precisamos mostrar explicitamente que um certo AFD não existe. O que acontece aqui é que todo trabalho da prova de inexistência do AFD fica "encapsulada" dentro da demonstração do LB. Segue um exemplo de como fazer uso do

Lema do Bombeamento para provar que uma determinada linguagem não é regular.

Teorema 4.1.2 $L_{01} = \{0^i 1^i \mid i \geq 1\}$ não é regular.

Prova: Suponha que L_{01} é regular. Portanto, usando o LB, sabemos que existe $t \in \mathbb{N}$, tal que se tomaramos uma string w de L "grande o suficiente" ou seja, tal que $|w| \geq t$, deve existir $x, y, z \in \Sigma^*$ tal que w pode ser escrita como $w = xyz$ de maneira que as três afirmações abaixo são verdadeiras:

(1) $y \neq \varepsilon$ *(2)* $|xy| \leq t$ *(3)* $\forall k \geq 0, xy^k z \in L_{01}$.

Considere a string $w = 0^t 1^t$. Note que $w \in L_{01}$ e $|w| \geq t$. Portanto podemos aplicar o LB e, com isso, w pode ser escrita na forma $w = xyz$ tal que as três afirmações acima são verdadeiras.

Pela condição *(2)*, temos que $|xy| \leq t$ e portanto a string xy contém apenas 0's. Portanto, todos os símbolos 1 da string w estão contidos em z (note não **necessariamente** z contém apenas símbolos 1, mas isso não é relevante aqui).

Pela condição *(3)*, a string $xy^k z$ deve pertencer a L_{01} para qualquer $k \geq 0$. Portanto, em particular, $xy^0 z \in L_{01}$. Com isso, temos que $xz \in L_{01}$.

Note que $|xyz| = 2t$. Pela condição *(1)*, $|xz| < |xyz|$ e portanto $|xz| < 2t$. Como z tem t símbolos 1, a string xz pode ter no máximo t-1 símbolos 0. Isso é uma contradição, pois $xz \in L_{01}$. Logo L_{01} não é regular. □

Vamos utilizar agora o LB em uma linguagem um pouco mais interessante:

Teorema 4.1.3 $L_p = \{1^p \mid p \text{ é um número primo }\}$ não é regular.

Prova: Suponha que L_p é regular. Então o LB nos diz que existe $t \in \mathbb{N}$, tal que se tomaramos uma string w de L_p tal que $|w| \geq t$, então $\exists x, y, z \in \Sigma^*$ tal que w pode ser escrita como $w = xyz$ e:

(1) $y \neq \varepsilon$ *(2)* $|xy| \leq t$ *(3)* $\forall k \geq 0, xy^k z \in L_p$.

Considere a string $w = 1^p$ para algum primo $p \geq t$. Note que w é uma string para a qual podemos aplicar o LB, pois $w \in L_P$ e $|w| \geq t$. Portanto w pode ser escrita na forma $w = xyz$ satisfazendo condições acima.

Pela condição *(3)*, a string $xy^k z$ deve pertencer a L_p para qualquer $k \geq 0$. Em particular, para $k = p+1$, podemos concluir que $xy^{p+1} z \in L_p$.

Note que $|xy^{p+1} z| = |xz| + |y^{p+1}|$. Seja $|y| = n$. Com isso temos:

$$\begin{aligned} |xy^{p+1} z| &= |xz| + |y^{p+1}| \\ &= (p-n) + n \cdot (p+1) \\ &= p - n + n + tp \\ &= p + np \\ &= p \cdot (1+n) \end{aligned}$$

Como p é primo, $p \geq 2$. Além disso, a condição *(1)* diz que $n \geq 1$, e portanto $(1+n) \geq 2$. Como ambos p e $(1+n)$ são maiores ou iguais a dois, o produto $p \cdot (1+n) = |xy^{p+1} z|$ não é um número primo. Isso contradiz o fato que $xy^{p+1} z \in L_p$. □

O Teorema 4.1.3 mostra que o problema de reconhecer se um determinado número é primo não pode ser solucionado usando um algoritmo (ou uma máquina) cujo funcionamento possa ser descrito por um autômato finito. Entretanto, observe que aqui estamos permitido que o alfabeto

contenha apenas símbolos 1, de maneira que os números primos são representados pelas strings 1^p, onde p é um primo. Também podemos provar algo mais "natural", que é supor que o alfabeto de entrada é $\Sigma = \{0,1\}$ e os primos são as strings binárias que representem números primos. Embora a prova disto seja um pouco complicada (veja o Exercício Opcional 4.6), vamos enunciar este teorema abaixo. No enunciado do teorema, relembramos que $N(w)$ é o número natural que a string binária w representa.

Teorema 4.1.4 $L_{\text{P}} = \{w \mid N(w) \text{ é um número primo}\}$ não é regular.

Prova: Exercício 4.6. □

Exercício 4.1 Prove que $L_{\text{RR}} = \{\text{"strings da forma } ww^R\text{"}\}$ não é regular. ■

Exercício 4.2 Prove que $L_{\text{EQ}} = \{w \mid w \text{ tem o mesmo número de 0's e 1's}\}$ não é regular. ■

Exercício 4.3 Prove que $L_{\text{DOBRO}} = \{w \mid \text{o número de 0's em } w \text{ é o dobro do número de 1's}\}$ não é regular. ■

Exercício 4.4 $L_p = \{1^p \mid p \text{ é um quadrado perfeito}\}$ não é regular. ■

Exercício 4.5 (OPCIONAL) Prove que a seguinte versão mais forte do Lema do Bombeamento é verdadeira:

> Seja L uma linguagem regular. Então existe uma constante t tal que $\forall w \in L$, com $|w| \geq t$, o seguinte é verdadeiro:
>
> $\exists u,x,y,z,v \in \Sigma^*$ tal que $w = xw'z = uxyzv$ e as três condições abaixo são satisfeitas:
>
> (1) $y \neq \varepsilon$ (2) $|xy| \leq t$ (3) $\forall k \geq 0, ux(w')^k zv \in L$

■

Exercício 4.6 (OPCIONAL) Prove que $L_{\text{P}} = \{w \mid N(w) \text{ é um número primo}\}$ não é regular. Dica: Use a versão do Lema do Bombeamento do Exercício 4.5. Além disso, para resulver esse exercício também podem ser úteis o Teorema de Dirichlet e o Pequeno Teorema de Fermat. ■

Exercício 4.7 Considere o alfabeto $\{0,1,M\}$ e a seguinte linguagem sobre este alfabeto: $L_{\text{RMR}} = \{wMw^R \mid \text{tal que } w \in \{0,1\}^*\}$. Use o Lema do Bombeamento para mostrar que L_{RMR} não é regular. ■

Exercício 4.8 Considere o alfabeto $\{0,1,M\}$ e a seguinte linguagem sobre este alfabeto: $L_{\text{0M1}} = \{0^iM1^i \mid \text{tal que } i \geq 1\}$. Use o Lema do Bombeamento para mostrar que L_{0M1} não é regular. ■

Exercício 4.9 Use o Lema do Bombeamento (LB) para provar que a seguinte linguagem sobre $\Sigma = \{0,1\}$ não é regular: $L = \{0^i1^j \mid i > j\}$. ■

4.2 Autômato com Pilha (AP)

No capítulo 3 nós apresentamos a definição de AFDs e definimos conjunto das linguagens regulares como sendo exatamente o conjunto de linguagens aceitas por AFDs. Depois disso, modificamos a nossa definição de autômatos: primeiro incluímos a habilidade do autômato "advinhar" qual transição fazer e, em seguida, adicionamos a possibilidade do autômato fazer transições ε. Entretanto, o conjunto de linguagens aceitas por tais autômatos continua sendo as linguagens regulares. Agora apresentar um cenário diferente: Vamos dar aos ε-AFNs uma habilidade que os tornarão mais poderosos, isto é, os tornarão capazes de aceitar linguagens que não são regulares.

A habilidade que vamos dar ao autômato agora é a possibilidade de armazenar informação e recuperar informação em uma memória. Entretanto, o tipo de memória que o autômato terá é bastante restrito: uma pilha de dados (veja Figura 4.1). O ponto chave aqui é que, embora o autômato tenha acesso a um tipo de memória bastante restrito, este fato será suficiente para aumentar o seu poder de computação.

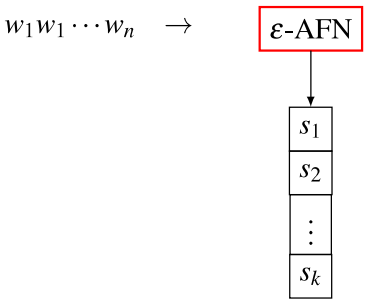


Figure 4.1: Esquema conceitual de autômatos com pilha.

Primeiramente, observamos que este novo modelo é capaz de realizar tarefas que um ε-AFN já realizava, ou seja, reconhecer linguagens regulares, pois podemos fazer computação com o autômato simplesmente ignorando a pilha de dados. Entretanto, quando fazendo uso da pilha de dados, veremos que este modelo de computação é capaz de resolver problemas que não era possível resolver com ε-AFNs.

4.2.1 O modelo matemático para autômatos com pilha

Nesta seção apresentaremos a definição matemática para um autômato com pilha. Para tal, precisamos esclarecer antes como este autômato funciona. Note primeiamente que durante a computação, um ε-AFN executa duas ações em cada um de seus passos:

(1) Processa um ou zero símbolos da string de entrada;
(2) Faz uma transição de estado.

Relembre que a quantidade de símbolos processados na ação (1) depende da transição efetuada (um símbolo nas transições comuns e zero símbolos nas transições ε). Quanto a ação (2), relembre também que a transição pode ser fazer com o autômato fique no mesmo estado que já se encontra. Autômatos com pilha, por sua vez, executam quatro ações a cada passo:

(1) Consome um ou zero símbolos da string de entrada;
(2) Desempilha o símbolo do topo da pilha;
(3) Faz uma transição de estado;

(4) Empilha uma quantidade finita de símbolos na pilha.

No caso dos ε-AFNs, as ações executadas a cada passo da dependem de um par (*estado atual*, *símbolo consumido*) e por isso, tem o domínio da função de transição igual a $Q \times \Sigma \cup \{\varepsilon\}$. No caso dos APs[1], as ações executadas a cada passo da computação do AP é determinado por uma tripla com a seguinte forma: (*estado atual, símbolo consumido, símbolo desempilhado*). Por este motivo, o domínio da função de transição δ do autômato com pilha deve ser $Q \times \Sigma \cup \{\varepsilon\} \times \Gamma$. A cada passo, um autômato com pilha, além de mudar de estado, também empilha símbolos na pilha de dados (formalmente, empilham uma string de símbolos). Mais precisamente, como estamos lidando com um modelo de computação não determinístico, dado (*estado, símbolo da entrada, símbolo da pilha*) a função δ devolve um conjunto de possíveis ações a serem executadas pelo autômato, onde cada elemento é do tipo (*novo estado, string empilhada*).

Definição 4.2.1 — Autômatos com Pilha (AP). Um autômato com pilha P é uma 7-tupla $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tal que:

Q, Σ, q_0, F: Tem a mesma interpretação que em um ε-AFN.

Γ é o alfabeto da pilha.

$Z_0 \in \Gamma$ é o *símbolo inicial da pilha*. Adicionalmente, temos que $Z_0 \notin \Sigma$.

$\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to \{(q_1, \gamma_1), (q_2, \gamma_2), ..., (q_k, \gamma_k)\}$, tal que $q_i \in Q$ e $\gamma_i \in \Gamma^*$.

Na definição acima, o símbolo Z_0 é, por convenção, o único símbolo presente na pilha no início da computação. Assim como fazemos com os demais autômatos vistos anteriormente, vamos usar diagramas para representar APs. Nos diagramas, o rótulo de uma transição do tipo $p \to q$ contém, além do símbolo de Σ processado na transição, o símbolo de Γ e a string que deve ser empilhada quando a transição é efetuada.

Vamos apresentar um diagrama de um AP com um exemplo instrutivo. Seja $\Sigma = \{0, 1\}$. Considere a linguagem $L_{RR} = \{w = xx^R : x \in \Sigma^*\}$. Observamos que esta linguagem não é regular (a demonstração deste fato é o objetivo do Exercício 4.1). O AP P_{RR}, da Figura 4.2 decide L_{RR}.

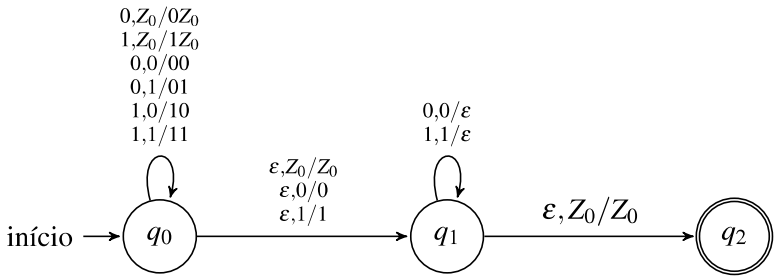


Figure 4.2: Diagrama do autômato com pilha P_{RR} que aceita a linguagem L_{RR}.

RÓTULOS NOS DIAGRAMAS DE AUTÔMATOS COM PILHA

No caso de diagramas de APs, os rótulos nas transições tem a forma $a, b/w$, onde $a, b \in \Sigma$ e $w \in \Sigma^*$. Por exemplo, na transição $q_1 \to q_1$, aparecem dois rótulos: $0, 0/\varepsilon$ e $1, 1/\varepsilon$.

Os rótulos do tipo $a, b/w$ devem ser interpretados da seguinte maneira: a transição é executada consumindo o símbolo a da string, desempilhando o símbolo b e empilhando a string w. A sequência da símbolos de w é empilhada de trás para frente. Por exemplo, se $w = 0Z_0$

[1] Estamos usando "AP" para nos referir aos autômatos com pilha. Entretanto, em alguns textos usa-se o acrônimo PDA, que vem do ingês *Pushdown Automata*.

(esta é a string w no primeiro dos seis rótulos existentes na transição $q_0 \to q_0$), o símbolo Z_0 é empilhado por primeiro e depois é empilhado o símbolo 0.

Quando projetamos um AP para realizar uma tarefa é comum tomar cuidado para que o símbolo Z_0 sempre esteja no "fundo" da pilha para que possamos saber que a pilha está vazia. Isso não é obrigatório, mas é uma boa prática no projeto de APs, pois o AP, a cada passo, sempre remove um símbolo da pilha por padrão. Mas o que acontece quando o AP tenta fazer uma trasição e a pilha está vazia? Aqui temos uma situação que não é muito diferente ε-AFNs tentando realizar uma transição que não está definida: o autômato morre.

Como já observamos, o modelo da Definição 4.2.1 não é determinítico, pois ele é uma generalização de um modelo que já não era determinístico. Uma pergunta importante é se este não determinismo faz diferença, ou seja, se podemos fornecer uma versão determinística de um AP que aceite as mesmas linguagens que os APs da Definição 4.2.1. Na Seção 4.2.4 daremos uma definição para APs determinísticos e veremos que tais APs **não** aceitam as mesmas linguagens aceitas por APs não determinísticos, embora ainda aceitem um conjunto maior de linguagens do que as linguagens regulares.

4.2.2 Computação com Autômatos com Pilha

A computação de um autômato com pilha a computação começa com o autômato no estado inicial q_0 com a string de entrada ainda sem ter sido processada e com a pilha do autômato contendo apenas o símbolo Z_0. A partir daí, a cada passo, a "situação" que a computação se encontra, será chamada de *configuração* da computação, pode ser sempre descrita dado:

(1) O estado q em que o AP se encontra;
(2) A string w de símbolos ainda não processada pelo AP;
(3) A string γ de símbolos que está armazenada na pilha de dados.

A configuração do autômato (q, w, γ) pode ser vista como uma "fotografia" do autômato, capturando a situação da computação naquele momento do tempo[2]. Formalizamos isso a seguir.

Definição 4.2.2 — Configuração de um AP. Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP e $q \in Q$, $w \in \Sigma^*$, $\gamma \in \Gamma^*$. Uma tripla (q, w, γ) é chamada de uma configuração de P.

Vamos agora definir o símbolo $\vdash_P$ abaixo:

Definição 4.2.3 — Um passo computacional. Seja um AP $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. O símbolo $\vdash_P$, usado para representar um passo que leva a computação de certa configuração para uma outra configuração, é definido da seguinte maneira:

Se $(p, \alpha) \in \delta(q, a, X)$, então $\forall w \in \Sigma^*$ e $\forall \beta \in \Gamma^*$, temos $(q, aw, X\beta) \vdash_P (p, w, \alpha\beta)$.

Sejam C_i e C_j configurações de P. A expressão $C_i \vdash_P C_j$ é lida "C_i *produz* C_j com um passo computacional". A seguir vamos definir o símbolo $\vdash_P^*$ para generalizar esta ideia para um número arbitrário de passos:

Definição 4.2.4 Sejam C_i, C_j, C_k configurações de um autômato P.

Base: $C_i \vdash_P^* C_i$

Indução: $C_i \vdash_P^* C_j$ se $\exists C_k$ tal que $C_i \vdash_P C_k$ e $C_k \vdash_P^* C_j$.

[2] A tripla contendo estes três elementos também é chamada em alguns textos de *descrição instantânea* do autômato.

A expressão $C_i \vdash_P^* C_j$ é lida "C_i *produz* C_j".

■ **Exemplo 4.1** Sabemos que o AP P_{RR} da Figura 4.2 aceita a string 010010. Portanto, podemos escrever o seguinte: $(q_0, 010010, Z_0) \vdash_{P_{\text{RR}}}^* (q_2, \varepsilon, Z_0)$. ■

Definição 4.2.5 — Árvore de computações possíveis de APs. Dado um AP P e uma string w. A árvore de computações possíveis de $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ com a string $w \in \Sigma^*$ é definida indutivamente:

Base: Inclua na árvore o nó raíz correspondente à configuração $C_0 = (q_0, w, Z_0)$.

Indução: Seja C_i um nó da árvore. Se $C_i \vdash_P C_j$, então inclua o nó C_j como filho de C_i na árvore.

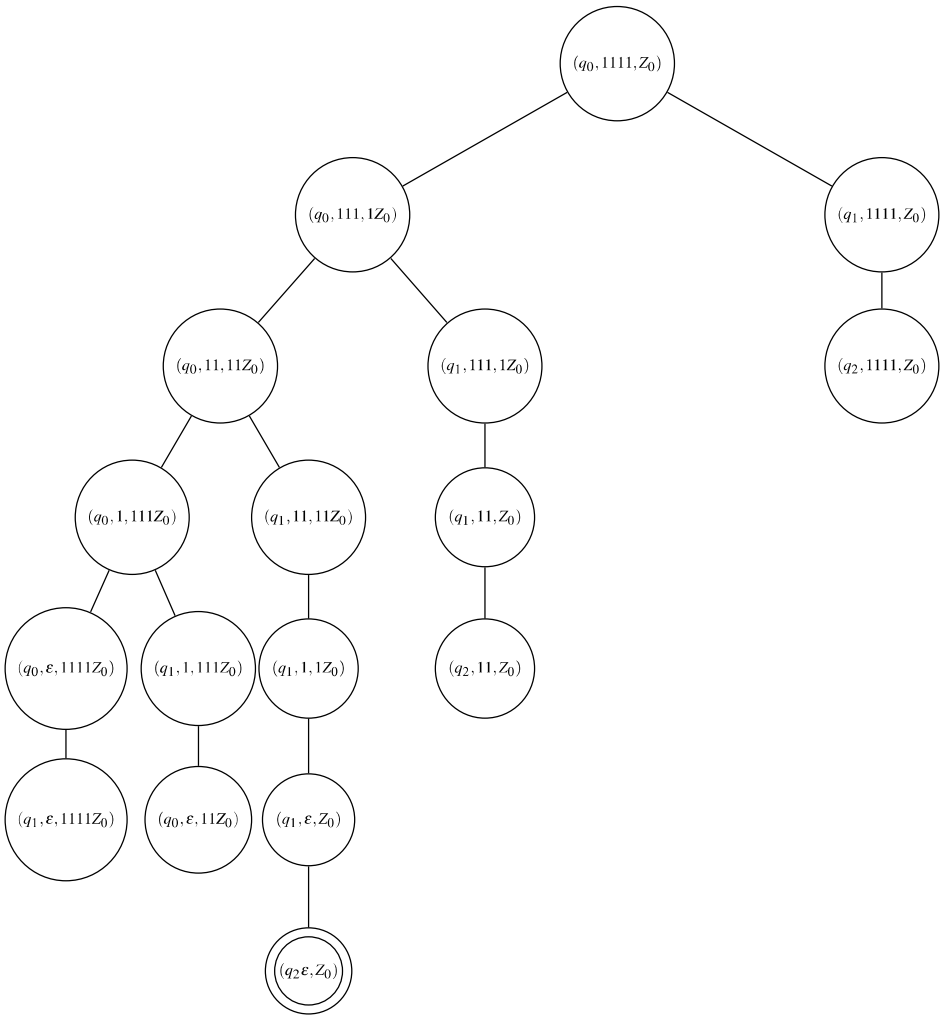


Figure 4.3: A árvore de computações possíveis do AP P_{RR} da Figura 4.2 com a string 1111. O nó marcado com dois círculos concêntricos é um nó que corresponde a uma configuração em que o AP aceita a string de entrada.

Definição 4.2.6 — Aceitação e rejeição de strings. Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Dado $w \in \Sigma^*$, dizemos que P *aceita* w se $(q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha)$ para $q \in F$ e $\alpha \in \Gamma^*$ qualquer. Caso contrário, dizemos que P *rejeita* w.

A linguagem de um AP é conjunto de strings que ele aceita, conforme a definição a seguir.

Definição 4.2.7 — Linguagens de APs. Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Definimos a linguagem de P como sendo $L(P) = \{w \in \Sigma^* : (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha)$ para $q \in F$ e $\alpha \in \Gamma^*$ qualquer$\}$.

Exercício 4.10 Mostre que toda linguagem regular é aceita por um autômato com pilha. ■

Exercício 4.11 O conjunto das linguagens regulares está estritamente contido no conjunto das linguagens que podem ser aceitas por APs? Justifique sua resposta. ■

4.2.3 Aceitação por pilha vazia

Relembramos que quando projetamos um AP, normalmente tomamos cuidado para que o fundo da pilha sempre contenha o símbolo Z_0, pois o AP morre quando tenta fazer uma trasição com a pilha vazia. O que vamos fazer agora é mudar completamente como entendemos a computação feita pelo AP. Vamos projetar APs de maneira que eles morram exatamente quando terminarem de ler a string de entrada.

Neste contexto, a pergunta chave que queremos fazer agora é a seguinte: quais são as strings que fazem com que um determinado AP esvazie completamente sua pilha (e por consequência morra no passo seguinte)? Dado um AP P, vamos definir a seguir $N(P)$ como sendo o conjunto contendo toda string que faz o AP morrer com a pilha vazia. Observe que o conjunto $N(P)$ não é a linguagem do AP, embora, em algumas situações, este possa ser o caso.

Definição 4.2.8 Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Então $N(P) = \{w \ ; \ (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \varepsilon)$ para $q \in Q$ qualquer$\}$.

■ **Exemplo 4.2** Como o AP P_{RR} da Figura 4.2 nunca esvazia a pilha, então $N(P_{\text{RR}}) = \varnothing$. ■

Observe que as strings de entrada de um dado AP P caem em dois casos: as strings que esvaziam a pilha de P e as strings que não esvaziam a pilha de P. Assim como temos pensado no AP P como tendo um certo *poder de computação* capaz de distinguir se uma dada string pertence ou não pertence a $L(P)$, podemos também pensar que P também tem um certo poder de computação para distinguir se a dada string pertence ou não pertence a $N(P)$. A partir de agora, vamos formalizar esta ideia e dizer que as strings de $N(P)$ são as strings que P *aceita por pilha vazia*. Observe que o fato de que P aceite por pilha vazia uma dada string w, não implica necessariamente que P aceite (no sentido da string estar na linguagem do AP) esta mesma string w. Entretanto, os seguintes teoremas mostram há uma certa equivalência entre os dois modos de aceitação de strings.

Teorema 4.2.1 Seja L a linguagem de um AP P. Então existe um AP P' tal que $N(P') = L$.

Teorema 4.2.2 Seja $N(P)$ o conjunto de strings aceitas por pilha vazia de AP P. Então existe um AP P' tal que $L(P') = N(P)$.

4.2.4 Autômatos com pilha determinísticos (APDs)

Vamos finalizar esta seção sobre APs seguindo a direção oposta que vínhamos fazendo desde o Capítulo 3. Até agora vínhamos, pouco a pouco, *generalizando* o nosso modelo de computação. Agora, vamos dar "um passo para trás" e definir um modelo de computação um pouco mais restrito. Chamaremos este modelo de *Autômato com Pilha Determinístico (APD)*.

Definição 4.2.9 — Autômatos com Pilha Determinísticos (APDs). Um *Autômato com Pilha Determinístico (APD)* é um Autômato com Pilha $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ cuja função de transição δ tem as seguintes restrições:

1. Para quaisquer, $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ e $X \in \Gamma$, temos $|\delta(q,a,X)| \leq 1$,
2. Se existe $q \in Q$, $a \in \Sigma$ e $X \in \Gamma$ tal que $\delta(q,a,X) \neq \varnothing$, então $\delta(q,\varepsilon,X) = \varnothing$.

Exercício 4.12 Mostre que toda linguagem regular pode ser decidida por um APD. ■

Exercício 4.13 Considere o alfabeto $\{0,1,M\}$ e a seguinte linguagem sobre este alfabeto: $L_{\text{RMR}} = \{wMw^R \mid \text{tal que } w \in \{0,1\}^*\}$. Mostre um APD que decida L_{RMR}. ■

Teorema 4.2.3 O conjunto de linguagens decididas por APs é estritamente maior do que o conjunto das linguagens regulares.

Prova: A demonstração é consequência direta dos Exercícios 4.8, 4.12 e 4.13. □

Teorema 4.2.4 Toda linguagem decidida por um APD também é decidida por um AP.

Prova: Como APDs são casos particulares de APs, trivialmente toda linguagem decidida por um APD também é decidida por um AP. □

A pergunta natural agora é se toda linguagem decidida por AP também pode ser decidida por um APD. O teorema a seguir enuncia que isto não é verdadeiro.

Teorema 4.2.5 Não existe APD que decida a linguagem L_{RR}.

4.3 Gramáticas Livre de Contexto

Na Seção 3.6 nós vimos que podemos usar expressões regulares para reprentar linguagens. Mais precisamente, vimos que as linguagens representáveis por expressões regulares são exatamente as linguagens aceitas por autômatos finitos. Neste seção vamos fazer algo semelhante. Vamos apresentar um outro formalismo matemático, chamado de *Gramáticas Livres de Contexto* (GLC), que também pode ser usado para representar linguagens. Na sequência veremos que as linguagens representáveis por GLCs são precisamente as linguagens aceitas por APs.

4.3.1 Definição formal de gramáticas livre de contexto

O objeto matemático que usaremos para representar linguagens nesta seção é uma quádrupla. Vamos primeiramente apresentar a definição matemática deste objeto e, em seguida, explicaremos como associamos uma linguagem a estas quádruplas.

Definição 4.3.1 Uma *Gramática Livre de Contexto* é uma quádrupla $G = (V, T, P, S)$ tal que:

- V é o conjunto de *variáveis*
- T é o conjunto de *símbolos terminais*.
- P é o conjunto de *regras de produção*, sendo que cada elemento de P é uma expressão da forma $X \to W$, onde $X \in V$ e W é uma string de $(V \cup T)^*$.
- S é a variável inicial, onde $S \in V$.

Para simplificar, muitas vezes diremos apenas "gramáticas", ao invés de gramáticas livres de contexto e apenas "regras" ao invés de regras de produção.

■ **Exemplo 4.3** $G_P = (\{P\}, \{0,1\}, A, P)$ é uma gramática onde A contém as seguintes regras:

$P \to \varepsilon$
$P \to 0$
$P \to 1$
$P \to 0P0$
$P \to 1P1$ ■

Na Seção 4.3.2 veremos como formalmente podemos associar linguagens a gramáticas, como a que acabamos de ver no Exemplo 4.3. Por enquanto vamos apenas nos certificar que entendemos exatamente que tipo de objeto matemático é uma gramática. Vamos a mais um exemplo:

■ **Exemplo 4.4** $G_1 = (\{I,E\}, \{a,b,0,1,+,*,(,)\}, A_1, E)$ onde as regras de A_1 são:

$E \to I$
$E \to E + E$
$E \to E * E$
$E \to (E)$
$I \to a$
$I \to b$
$I \to Ia$
$I \to Ib$
$I \to I0$
$I \to I1$ ■

Mais uma vez, para simplificar, em vez de escrever a lista completa de regras de um dado conjunto de regras, vamos utilizar uma notação mais compacta. No caso do Exemplo 4.4, ao invés de gastar dez linhas para descrever o conjunto de regras A_1, poderíamos ter usado apenas duas linhas e o descrito da seguinte maneira:

$E \to I|E+E|E*E|(E)$
$I \to a|b|Ia|Ib|I0|I1$

De maneira semelhante, o conjunto de regras do Exemplo 4.3 é descrito por $P \to \varepsilon|0|1|0P0|1P1$.

4.3.2 Derivações de uma gramática

Dada uma gramática G, veremos agora qual é a linguagem $L(G)$ que esta gramática representa. Para chegarmos a tal definição, vamos definir o conceito de *derivação* de strings. A ideia é que as strings deriváveis de G são as strings que pertencem a linguagem $L(G)$.

Definição 4.3.2 Seja $G = (V,T,P,S)$ uma gramática e seja uma string $\alpha A\beta$ onde $A \in V$ e $\alpha,\beta \in (V \cup T)^*$. Seja $A \to \gamma$ uma regra de P. Então dizemos que $\alpha A\beta \Rightarrow_G \alpha\gamma\beta$

No exemplo acima, dizemos que a string $\alpha\gamma\beta$ pode ser *derivada* da string $\alpha A\beta$ na gramática G. Alternativamente a a string $\alpha A\beta$ *produz* a string $\alpha\gamma\beta$ na gramática G. Observe que tanto $\alpha\gamma\beta$ quanto $\alpha A\beta$ são strings cujos símbolos pertencem ao conjunto $V \cup T$. Estas strings são chamadas de *termos sentenciais*. As strings cujos símbolos pertencem apenas ao conjunto T são chamadas de *strings terminais*. Note que uma derivação pode ser aplicada apenas a strings que são termos sentenciais. Por outro lado, quando formos definir quais são as strings que pertencem a uma dada gramática, estaremos interessados apenas em strings terminais.

Antes de seguir em frente, vamos mostrar como usar a Definição 4.3.2 dizer formalmente que a $a*(a+b00)$ pode ser derivada da gramática G_1 do Exemplo 4.4.

■ **Exemplo 4.5** A string $a*(a+b00)$ pode ser derivada da gramática G_1, pois:

$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow a*(a+I) \Rightarrow a*(a+I0) \Rightarrow a*(aI00) \Rightarrow a*(a+b00)$ ■

A próxima definição será útil para indicar que uma string pode derivada de uma dada gramática usando uma ou mais derivações.

Definição 4.3.3 Sejam $\alpha,\beta,\gamma \in (V \cup T)^*$, definimos indutivamente $\Rightarrow_G^*$:

BASE: $\alpha \Rightarrow_G^* \alpha$
INDUÇÃO: se $\alpha \Rightarrow_G^* \beta$ e $\beta \Rightarrow_G \gamma$, então $\alpha \Rightarrow_G^* \gamma$

No Exemplo 4.5 vimos que podemos derivar $a*(a+b00)$ de G_1 Usando uma série de derivações. Agora temos uma maneira formal de dizer isso: $E \Rightarrow_G^* a*(a+b00)$. Com isso podemos definir qual é linguagem associada a uma dada gramática.

Definição 4.3.4 — A Linguagem de uma Gramática. Dada uma gramática $G = (V,T,P,S)$, a linguagem de G é $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$.

4.3.3 Derivação mais a direita e mais a esquerda

Em algumas situações pode ser útil fixarmos que em uma derivação de uma string α nós sempre escolhemos aplicar a regra de produção à variável mais a esquerda presente em α (caso haja mais de uma variável na string α). Tal derivação é dita *mais a esquerda*. Escrevemos $\Rightarrow_{lm}$ e $\Rightarrow_{lm}^*$ (assumindo que G é conhecida). Similarmente temos *derivações mais a direita* e podemos representá-las usando os símbolos $\Rightarrow_{rm}$ e $\Rightarrow_{rm}^*$.

Exercício 4.14 Verifique que a derivação de G_1 do Exemplo 4.5 é mais a esquerda ■

Exercício 4.15 Forneça uma derivação mais a direita para a gramática G_1 do Exemplo 4.4. ■

Definição 4.3.5 — Linguagens Livre de Contexto. Se L é a linguagem de uma gramática livre de contexto G, então dizemos que L é uma *Linguagem Livre de Contexto*. Neste casos dizemos que a gramática G *gera* a linguagem L.

■ **Exemplo 4.6** As linguagens $L(G_p)$ e $L(G_1)$ das gramáticas dos Exemplos 4.3 e 4.4 são linguagens livre de contexto, pois são geradas por gramáticas livre de contexto. ■

4.3.4 Árvores de análise sintática

Exercício 4.16 Veja no livro [Sip06] o conceito de árvore de análise sintática (também conhecidas como árvores de derivação ou *parse trees*) de gramáticas livres de contexto. ■

4.3.5 Ambiguidade de Gramáticas

Seja $G = (V, T, P, S)$ uma gramática e $w \in L(G)$. Se existem duas árvores de derivação diferentes para w, então dizemos que G é ambígua. Se existe exatamente uma árvore de derivação para cada string $w \in L(G)$, então G é uma gramática *não ambígua*.

OBSERVAÇÕES IMPORTANTES:

- Mesmo que uma string tenha várias derivações diferentes, isso não necessariamente quer dizer que a gramática seja ambígua. A gramática somente é ambígua se existir uma string que admita mais do que uma árvore de derivação.
- Por outro lado, no caso de apenas permitirmos derivações mais a esquerda e ainda assim pudermos obter mais do que uma derivação de uma dada string, podemos concluir que a gramática é ambígua. O mesmo pode ser dito no caso em que permitimos apenas derivações mais a direita.

Dada uma linguagem livre de contexto L, por definição existe uma gramática livre de contexto G tal que $L(G) = L$. Podemos nos perguntar se sempre é possível obter G que gere L tal que G não seja ambígua. A resposta é não, pois pois existem linguagens livre de contexto que são ditas *inerentemente* ambíguas. Mais precisamente, isso quer dizer que existem linguagens livre de contexto L tal que para toda gramática G tal que $L(G) = L$, a gramática G é ambígua. A linguagem $L = \{a^n b^n c^m d^m ; n > 0, m > 0\} \cup \{a^n b^m c^m d^n ; n > 0, m > 0\}$ é um exemplo de tal linguagem. Entretanto não vamos apresentar aqui a demonstração de que esta linguagem é inerentemente ambígua, pois isto é bastante complicado e está fora do escopo deste curso.

4.3.6 Equivalência entre APs e gramáticas livre de contexto

Enunciamos abaixo o teorema mais importante a respeito de APs e gramáticas:

Teorema 4.3.1 Uma linguagem L é livre de contexto $\Leftrightarrow$ existe um AP que aceita L.

4.3.7 APDs e ambiguidade de gramáticas

O Teorema 4.3.1 mostra que o conjunto de linguagens aceitas por APs é exatamente o mesmo conjunto das linguagens expressa por gramáticas. Por outro lado, vimos anteriormente que o conjunto das linguagens aceitas por APs não é o mesmo que o conjunto de linguagens aceitas por APDs. Além disso, vimos que algumas gramáticas são inerentemente ambíguas. Em outras palavras, o mundo das linguagens livres de contexto é bem mais sutil do que o mundo das linguagens regulares. Enunciamos a seguir dois teoremas que mostram a relação entre ambiguidade de gramáticas e APDs.

Teorema 4.3.2 Se $L = L(P)$ para algum APD P, então existe uma gramática livre de contexto não ambígua G tal que $L(G) = L$.

Teorema 4.3.3 Existe uma gramática livre de contexto não ambígua G tal que não existe nenhum APD que aceite G.

4.3.8 Exercícios

Exercício 4.17 Forneça um AP que aceite a linguagem $\{0^n 1^n \mid n \geq 1\}$. ■

Exercício 4.18 Forneça um AP que aceite a linguagem $L = \{0^i 1^j \mid i > j\}$. ■

Exercício 4.19 Considere o alfabeto Σ que contém os símbolos de "abertura" e "fechamento" de parêntesis, ou seja $\Sigma = \{(,)\}$. Forneça um AP que aceite a linguagem das strings de parêntesis balanceados sobre σ. Por exemplo, a string $(()(()))$ deve ser aceita, enquanto a string $())($ não deve ser aceita. ■

Exercício 4.20 Seja $A = \{a,b,c\}$ e seja L a linguagem $\{a^n bc^n \mid n \geq 1\}$ sobre o alfabeto A. Forneça um AP que aceite a linguagem L. ■

Exercício 4.21 Apresente um AP para a linguagem das strings palíndromas sobre o alfabeto $\Sigma = \{0,1\}$. ■

Exercício 4.22 Forneça uma CFG para a linguagem $\{0^n 1^n \mid n \geq 1\}$. ■

Exercício 4.23 Observe que a linguagem da expressão regular $\underline{0^*1(0+1)^*}$ é a mesma linguagem da *gramática regular* $G = (V,T,P,S)$ com as regras abaixo:

$$S \to A1B$$
$$A \to 0A \mid \varepsilon$$
$$B \to 0B \mid 1B \mid \varepsilon$$

Forneça uma derivação mais a direita e uma derivação mais a esquerda da string 00101. ■

Exercício 4.24 Considere a gramática $G = (V,T,P,S)$, onde $V = \{S\}$, $T = \{0,1\}$ e as regras de P são:

$$S \to \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$$

Considere agora a seguinte definição: $S(G) = \{w \in (V \cup T)^* \mid S \Rightarrow_G^* w\}$. Com isso em mente, forneça um exemplo de string w tal que $w \in S(G) \setminus L(G)$. ■

Exercício 4.25 Forneça uma gramática que tem como linguagem expressões bem formadas em lógica proposicional. Lembre que em uma expressão bem formada em lógica proposicional pode ter variáveis $x_1, x_2, x_3, \ldots$, operadores binários $\wedge, \vee, \Rightarrow, \Leftrightarrow$, operador unário $\neg$ e abertura e fechamento de parênteses. ■

Exercício 4.26 Considere a gramática $G = (V, T, P, S)$, onde

- $V = \{$ [STMT], [IF-THEN], [IF-THEN-ELSE], [ASSIGN] $\}$
- $T = \{$ $a = 1$, if, condition, then, else $\}$
- $S =$ [STMT]

Onde as regras são:

[STMT] → [IF-THEN] | [IF-THEN-ELSE] | [ASSIGN]

[IF-THEN] → if condition then [STMT]

[IF-THEN-ELSE] → if condition then [STMT] else [STMT]

[ASSIGN] → a=1

A gramática G acima é naturalmente o fragmento de uma linguagem de programação, entretanto G é ambígua. Mostre que a gramática é ambígua e forneça uma outra gramática não ambígua equivalente. ■

Exercício 4.27 Para responder as questões abaixo assuma que o alfabeto é $\Sigma = \{a, b, c\}$.

(a) Forneça um AP P tal que $L(P) = \{a^m b^n c^n;\ m, n \geq 0\}$.

(b) Forneça uma Gramática G tal que $L(G) = \{a^n b^n c^m;\ m, n \geq 0\}$.

■

Exercício 4.28 (OPCIONAL) Prove que $L = \{a^n b^n c^n;\ n \geq 0\}$ não é livre de contexto. ■

Exercício 4.29 A interseção de duas linguagens regulares também é uma liguagem regular. Por ter esta propriedade, as linguagens regulares são ditas *fechadas sob interseção*. Nesta questão você deve provar que esta propriedade de ser fechada sob interseção não vale para linguagens livre de contexto, ou seja, você deve provar que existem linguagens livre de contexto L_1 e L_2 tal que $L_1 \cap L_2$ não é uma linguagem livre de contexto. Dica: Tente usar o fato de que a linguagem da questão 4.28 não é livre de contexto. ■