Introdução à Teoria de Autômatos, Linguagens e Computação

John E. Hopcroft
Cornell University

Jeffrey D. Ullman
Stanford University

Rajeev Motwani
Stanford University

Introdução à teoria de
UESC
99238
99238 UESC

EDITORA CAMPUS

TRADUÇÃO DA SEGUNDA EDIÇÃO AMERICANA

Capítulo 1

Autômatos: os métodos e a loucura

A teoria de autômatos é o estudo dos dispositivos de computação abstratos, ou "máquinas". Antes de existirem os computadores, na década de 1930, A. Turing estudou uma máquina abstrata que tinha todas as características dos computadores atuais, pelo menos no que se refere ao quanto eles poderiam calcular. O objetivo de Turing era descrever com exatidão o limite entre o que uma máquina de computação podia fazer e aquilo que ela não podia fazer; suas conclusões se aplicam não apenas às suas *máquinas de Turing* abstratas, mas também às máquinas reais de hoje.

Nas décadas de 1940 e 1950, tipos de máquinas mais simples, que hoje chamamos "autômatos finitos", foram estudados por diversos pesquisadores. Esses autômatos, propostos originalmente para modelar a função do cérebro, se mostraram extremamente úteis para uma grande variedade de outros propósitos, que mencionaremos na Seção 1.1. Também no final dos anos 50, o lingüista N. Chomsky iniciou o estudo de "gramáticas" formais. Embora não sejam estritamente máquinas, essas gramáticas têm relacionamentos estreitos com os autômatos abstratos e hoje servem como a base de alguns importantes componentes de software, incluindo algumas partes dos compiladores.

Em 1969, S. Cook estendeu o estudo de Turing do que podia e do que não podia ser calculado. Cook conseguiu separar os problemas que podem ser resolvidos de forma eficiente por computadores daqueles problemas que podem em princípio ser resolvidos mas que, na prática, levam tanto tempo que os computadores são inúteis para solucionar todas as instâncias do problema, exceto aquelas muito pequenas. Os problemas dessa última classe são chamados "intratáveis" ou "NP-difíceis" (NP-hard). É altamente improvável que até mesmo a melhoria exponencial na velocidade de computação que o hardware de computadores vem alcançando ("Lei de Moore") tenha impacto significativo sobre nossa habilidade para resolver grandes instâncias de problemas intratáveis.

Todos esses desenvolvimentos teóricos têm relação direta com aquilo que os cientistas da computação fazem hoje. Alguns conceitos, como autômatos finitos e certos tipos de gramáticas formais, são usados no projeto e na construção de importantes componentes de software. Outros conceitos, como a máquina de Turing, ajudam a entender o que podemos esperar de nosso software. Em especial, a teoria de problemas intratáveis nos permite deduzir se temos a chance de, ao nos depararmos com um problema, sermos capazes de escrever um programa para resolvê-lo (porque ele não pertence à classe intratável), ou se teremos de descobrir algum modo de contornar o problema intratável: encontrar uma aproximação, usar uma heurística ou empregar algum outro método para limitar o período de tempo que o programa despenderá para resolver o problema.

Neste capítulo introdutório, começaremos com uma visão de alto nível do objeto de estudo da teoria dos autômatos e de quais são seus usos. Grande parte do capítulo é dedicada a uma pesquisa a respeito de técnicas de prova (ou demonstração) e de artifícios para descobrir provas. Abordaremos as provas dedutivas, a reformulação de enunciados, as provas por contradição, as provas por indução e outros conceitos importantes. Uma seção final introduz os conceitos que permeiam a teoria de autômatos: alfabetos, strings e linguagens.

1.1 Por que estudar a teoria de autômatos?

Há várias razões pelas quais o estudo de autômatos e complexidade é uma parte importante do núcleo da Ciência da Computação. Esta seção serve para apresentar ao leitor a principal motivação, e também descreve os tópicos mais importantes abordados neste livro.

1.1.1 Introdução aos autômatos finitos

Os autômatos finitos constituem um modelo útil para muitos elementos importantes de hardware e software. Veremos, a partir do Capítulo 2, exemplos de como os conceitos são usados. Por enquanto, vamos apenas listar alguns dos elementos mais importantes:

1. Software para projetar e verificar o comportamento de circuitos digitais.
2. O "analisador léxico" de um compilador típico, isto é, o componente do compilador que divide o texto de entrada em unidades lógicas, como identificadores, palavras-chave e pontuação.
3. Software para examinar grandes corpos de texto, como coleções de páginas da Web, a fim de encontrar ocorrências de palavras, frases ou outros padrões.

4. Software para verificar sistemas de todos os tipos que têm um número finito de estados distintos, como protocolos de comunicações ou protocolos para troca segura de informações.

Logo encontraremos uma definição precisa de autômatos de vários tipos, mas, enquanto isso, vamos iniciar nossa introdução informal com um esboço do que um autômato finito é e do que ele faz. Existem muitos sistemas ou componentes, como aqueles enumerados anteriormente, que podemos considerar como estando, a todo momento, em um de um número finito de "estados". O propósito de um estado é memorizar a porção relevante da história do sistema. Tendo em vista que existe apenas um número finito de estados, a história inteira em geral não pode ser memorizada, e assim o sistema tem de ser projetado com cuidado, a fim de memorizar o que é importante e esquecer o que não é. A vantagem de se ter apenas um número finito de estados é que podemos implementar o sistema com um conjunto fixo de recursos. Por exemplo, poderíamos implementá-lo em hardware como um circuito, ou como uma forma simples de programa que possa tomar decisões examinando somente uma quantidade limitada de dados ou usando a posição no próprio código para tomar a decisão.

Exemplo 1.1: Talvez o autômato finito não trivial mais simples seja um interruptor liga/desliga. O dispositivo memoriza se está no estado "ligado" o no estado "desligado", e permite ao usuário pressionar um botão cujo efeito é diferente, dependendo do estado do interruptor. Isto é, se o interruptor está no estado desligado, pressionar o botão o leva ao estado ligado e, se o interruptor se encontra no estado ligado, pressionar o mesmo botão o leva ao estado desligado.

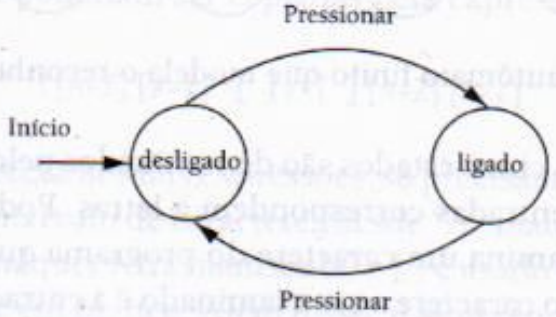

Figura 1.1: Um autômato finito que modela um interruptor liga/desliga

O modelo de autômato finito para o interruptor é mostrado na Figura 1.1. Como ocorre em todos os autômatos finitos, os estados são representados por círculos; nesse exemplo, denominamos os estados de *ligado* e *desligado*. Arcos entre estados são identificados por "entradas", que representem influências externas sobre o sistema. Aqui, ambos os arcos são identificados pela entrada *Pressionar*, que representa um usuário pressionando o botão. A intenção dos dois

arcos é que, seja qual for o estado em que o sistema se encontre, quando a entrada *Pressionar* é recebida, ele vai para o outro estado.

Um dos estados é designado como o "estado inicial ", o estado em que o sistema é colocado inicialmente. Em nosso exemplo, o estado inicial é *desligado*, e por convenção indicamos o estado inicial pela palavra *Início* e por uma seta que leva a esse estado.

Com freqüência, é necessário indicar um ou mais estados como estados "finais", ou de "aceitação". A chegada em um desses estados após uma seqüência de entradas indica que a seqüência de entrada é boa em algum aspecto. Por exemplo, poderíamos ter considerado o estado *ligado* na Figura 1.1 como um estado de aceitação porque, nesse estado, o dispositivo que está sendo controlado pelo interruptor irá operar. É usual designar estados de aceitação por um círculo duplo, embora não tenhamos feito tal designação na Figura 1.1. □

Exemplo 1.2: Às vezes, o que é memorizado por um estado pode ser muito mais complexo que uma escolha entre ligado/desligado. A Figura 1.2 mostra outro autômato finito que poderia fazer parte de um analisador léxico. O trabalho desse autômato é reconhecer a palavra-chave `then`. Desse modo, ele precisa de cinco estados, cada um dos quais representa uma posição diferente na palavra `then`, conforme o que foi atingido até o momento. Essas posições correspondem aos prefixos da palavra, variando desde o string vazio (isto é, nenhuma parte da palavra foi atingida até agora) até a palavra completa.

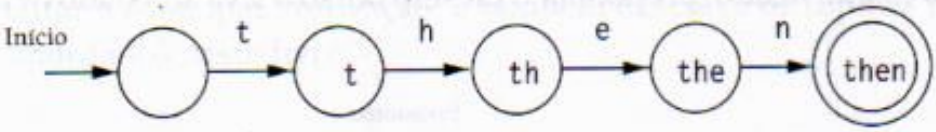

Figura 1.2: Um autômato finito que modela o reconhecimento de `then`

Na Figura 1.2, os cinco estados são denominados pelo prefixo de `then` visto até o momento. As entradas correspondem a letras. Podemos imaginar que o analisador léxico examina um caractere do programa que está compilando de cada vez, e o próximo caractere a ser examinado é a entrada para o autômato. O estado inicial corresponde ao string vazio, e cada estado tem uma transição na próxima letra de `then` para o estado que corresponde ao prefixo seguinte. O estado denominado `then` é atingido quando a entrada grafa a palavra completa `then`. Tendo em vista que o trabalho desse autômato é reconhecer o momento em que `then` é visto, poderíamos considerar esse estado como o único estado de aceitação.

1.1.2 Representações estruturais

Há dois formalismos importantes que não são semelhantes a autômatos, mas que desempenham um importante papel no estudo de autômatos e suas aplicações.

1. As *gramáticas* são modelos úteis quando se projeta software que processa dados com uma estrutura recursiva. O exemplo mais conhecido é um "analisador sintático" (parser), o componente de um compilador que lida com as características recursivamente aninhadas de uma linguagem de programação típica, como as expressões – aritméticas, condicionais e assim por diante. Por exemplo, uma regra gramatical na forma $E \Rightarrow E + E$ estabelece que uma expressão pode ser formada tomando-se duas expressões quaisquer e conectando-as por um sinal de adição; essa regra é típica do modo como são formadas as expressões de linguagens de programação reais. Introduzimos as gramáticas livres de contexto, como são chamadas usualmente, no Capítulo 5.
2. As *expressões regulares* também denotam uma estrutura de dados, especialmente strings de texto. Como veremos no Capítulo 3, os padrões de strings que elas descrevem são exatamente iguais aos que podem ser descritos por autômatos finitos. O estilo dessas expressões difere significativamente do estilo das gramáticas, e nos contentaremos aqui com um exemplo simples. A expressão regular no estilo do UNIX `'[A-Z][a-z] *[ ] [A-Z][A-Z]'` representa palavras em maiúsculas e minúsculas seguidas por um espaço e duas letras maiúsculas. Essa expressão representa padrões em texto que poderiam ser uma cidade e um estado, por exemplo, `Recife PE`. Ela omite nomes de cidades com várias palavras, como `Porto Alegre RS`, que poderiam ser captadas pela expressão mais complexa

   `'([A-Z][a-z]*[ ])*[ ][A-Z][A-Z]'`

   Quando interpretamos tais expressões, só precisamos saber que `[A-Z]` representa um intervalo de caracteres desde "A" maiúscula até "Z" maiúscula (isto é, qualquer letra maiúscula) e [ ] é usado para representar apenas o caractere em branco. Além disso, o símbolo * representa a repetição por "qualquer número de" vezes da expressão precedente. Os parênteses são usados para agrupar componentes da expressão; eles não representam caracteres do texto descrito.

1.1.3 Autômatos e complexidade

Os autômatos são essenciais para o estudo dos limites de computação. Como mencionamos na introdução ao capítulo, existem duas questões importantes:

1. O que um computador pode fazer, afinal? Esse estudo é chamado "decidibilidade", e os problemas que podem ser resolvidos pelo computador são chamados "decidíveis". Esse tópico é tratado no Capítulo 9.
2. O que um computador pode fazer de forma eficiente? Esse estudo é chamado "intratabilidade", e os problemas que podem ser resolvidos por um computador sem utilizar mais tempo que alguma função lentamente crescente do tamanho da entrada são chamados "tratáveis". Com freqüência, consideramos todas as funções polinomiais "lentamente crescentes", enquanto funções que crescem com rapidez maior que qualquer polinômio são julgadas como sendo de crescimento rápido demais. Esse assunto é estudado no Capítulo 10.

1.2 Introdução às provas formais

Se estudasse geometria plana no ensino médio em qualquer época antes da década de 1990, é bem provável que você tivesse de realizar algumas "provas dedutivas" minuciosas, em que mostraria a verdade de uma afirmação (ou declaração) por meio de uma seqüência detalhada de etapas e raciocínios. Embora a geometria tenha seu lado prático (por exemplo, você precisa conhecer a regra para calcular a área de um retângulo, se tiver necessidade de comprar a quantidade correta de carpete para forrar uma sala), o estudo de metodologias de provas formais foi importante ao menos como uma razão para abordar esse ramo da matemática no ensino médio.

Na década de 1990 nos EUA se tornou popular ensinar a demonstração como se ela fosse uma questão de sentimentos pessoais a respeito da afirmação. Embora seja bom sentir a verdade de uma afirmação que você precisa usar, as técnicas de prova importantes não são mais dominadas na escola secundária. Ainda assim, a prova é algo que todo cientista da computação precisa entender. Alguns cientistas da computação adotam a visão extrema de que uma prova formal da correção de um programa deve seguir lado a lado com a escrita do próprio programa. Duvidamos que isso seja produtivo. Por outro lado, existem aqueles que dizem que a prova não tem lugar na disciplina de programação. O dito "se não tiver certeza de que seu programa está correto, execute-o e veja" é oferecido comumente por esse grupo.

Nossa posição está entre esses dois extremos. Testar programas é sem dúvida essencial. No entanto, os testes vão apenas até certo ponto, pois não é possível experimentar um programa para toda entrada. Mais importante ainda, se o seu programa é complexo – digamos, uma recursão ou iteração complicada – então, se não entender o que está acontecendo enquanto se percorre um loop ou se chama uma função recursivamente, é improvável que você escreva o código de modo correto. Quando seus testes o informarem de que o código está incorreto, você ainda precisará corrigi-lo.

Para tornar correta sua iteração ou recursão, você precisará configurar uma hipótese indutiva, e é útil para raciocinar, seja formal ou informalmente, que a hipótese é coerente com a iteração ou recursão. Esse processo de entender o funcionamento de um programa correto é em essência igual ao processo de provar teoremas por indução. Desse modo, além de lhe fornecer modelos que serão úteis para certos tipos de software, tornou-se tradicional em um curso sobre teoria de autômatos cobrir as metodologias de prova formal. Talvez mais do que outros temas centrais da Ciência da Computação, a teoria de autômatos se presta a provas naturais e interessantes, tanto do tipo *dedutivo* (uma seqüência de etapas justificadas) quanto do tipo *indutivo* (provas recursivas de uma afirmação parametrizada que utilizam a própria afirmação com valores "mais baixos" do parâmetro).

1.2.1 Provas dedutivas

Como mencionamos anteriormente, uma prova dedutiva consiste em uma seqüência de afirmações cuja verdade nos leva de alguma afirmação inicial, chamada *hipótese* ou *declaração(ões) dada(s)*, a uma afirmação de *conclusão*. Cada etapa da prova deve se seguir, por algum princípio lógico aceito, dos fatos dados, de algumas das afirmações anteriores na prova dedutiva, ou de uma combinação desses elementos.

A hipótese pode ser verdadeira ou falsa, em geral dependendo dos valores de seus parâmetros. Com freqüência, a hipótese consiste em várias afirmações independentes conectadas por um AND lógico. Nesses casos, dizemos que cada uma dessas afirmações é uma hipótese ou uma declaração dada.

O teorema que é provado quando vamos de uma hipótese H até uma conclusão C é a afirmação "se H então C". Dizemos que C é *deduzido* a partir de H. Um exemplo de teorema da forma "se H então C" ilustrará esses pontos.

Teorema 1.3: Se $x \geq 4$, então $2^x \geq x^2$. □

Não é difícil nos convencermos informalmente de que o Teorema 1.3 é verdadeiro, embora uma prova formal exija indução e seja adiada até o Exemplo 1.17. Primeiro, note que a hipótese H é "$x \geq 4$". Essa hipótese tem um parâmetro x e, portanto, não é verdadeira nem falsa. Em vez disso, sua verdade depende do valor do parâmetro x; por exemplo, H é verdadeira para $x = 6$ e falsa para $x = 2$.

Da mesma forma, a conclusão C é "$2^x \geq x^2$". Essa afirmação também utiliza o parâmetro x e é verdadeira para certos valores de x e falsa para outros. Por exemplo, C é falsa para $x = 3$, pois $2^3 = 8$, que não é tão grande quanto $3^2 = 9$. Por outro lado, C é verdadeira para $x = 4$, pois $2^4 = 4^2 = 16$. Para $x = 5$, a afirmação também é verdadeira, pois $2^5 = 32$ é pelo menos tão grande quanto $5^2 = 25$.

Você já deve ter visto o argumento intuitivo que nos leva à conclusão de que $2^x \geq x^2$ será verdadeira sempre que $x \geq 4$. Já vimos que ela é verdadeira para $x = 4$. À medida que x cresce além de 4, o lado esquerdo 2^x é duplicado a cada vez que x aumenta em uma unidade. Entretanto, o lado direito, x^2, cresce de acordo com a taxa $\left(\frac{x+1}{x}\right)^2$. Se $x \geq 4$, então $(x+1)/x$ não pode ser maior que 1,25 e, portanto, $\left(\frac{x+1}{x}\right)^2$ não pode ser maior que 1,5625. Como $1{,}5625 < 2$, toda vez que x aumenta acima de 4, o lado esquerdo 2^x cresce mais que o lado direito x^2. Desse modo, desde que comecemos com um valor como $x = 4$, no qual a desigualdade $2^x \geq x^2$ já é satisfeita, poderemos aumentar x como preferirmos, pois a desigualdade ainda será satisfeita.

Completamos assim uma prova informal, embora precisa do Teorema 1.3. Voltaremos à prova e a tornaremos mais precisa no Exemplo 1.17, depois que introduzirmos as provas "indutivas".

O Teorema 1.3, como todos os teoremas interessantes, envolve um número infinito de fatos inter-relacionados, nesse caso a afirmação "se $x \geq 4$, então $2^x \geq x^2$" para todos os inteiros x. De fato, não precisamos supor que x é um inteiro, mas a prova mencionou repetidamente o aumento de x em 1, começando em $x = 4$, e assim só estamos realmente interessados na situação em que x é um inteiro.

O Teorema 1.3 pode ser usado para ajudar a deduzir outros teoremas. No próximo exemplo, consideraremos uma prova dedutiva completa de um teorema simples que utiliza o Teorema 1.3.

Teorema 1.4: Se x é a soma dos quadrados de quatro inteiros positivos, então $2^x \geq x^2$.

PROVA: A idéia intuitiva da prova é que, se a hipótese é verdadeira para x, isto é, x é a soma dos quadrados de quatro inteiros positivos, então x deve ser pelo menos 4. Então, a hipótese do Teorema 1.3 é válida e, como acreditamos nesse teorema, podemos afirmar que sua conclusão também é verdadeira para x. O raciocínio pode ser expresso como uma seqüência de etapas. Cada etapa é a hipótese do teorema a ser provado, parte dessa hipótese, ou uma afirmação que decorre de uma ou mais afirmações anteriores.

Por "decorre" queremos dizer que, se a hipótese de algum teorema é uma afirmação anterior, então a conclusão desse teorema é verdadeira, e pode ser enunciada como uma afirmação de nossa prova. Essa regra lógica é freqüentemente chamada *modus ponens*; ou seja, se sabemos que H é verdadeira e sabemos que "se H então C" é verdadeira, podemos concluir que C é verdadeira. Também permitimos que algumas outras etapas lógicas sejam usadas na criação de uma afirmação que decorre de uma ou mais afirmações anteriores. Por exemplo, se A e B são duas afirmações anteriores, então podemos deduzir e enunciar a afirmação "A e B".

A Figura 1.3 mostra a seqüência de afirmações de que precisamos para provar o Teorema 1.4. Embora em geral não sejam provados teoremas nessa forma tão estilizada, ela ajuda a pensar em provas como listas muito explícitas de afirmações, cada qual com uma justificativa precisa. Na etapa (1), repetimos uma das declarações dadas do teorema: que x é a soma dos quadrados de quatro inteiros. Muitas vezes é de grande ajuda nas provas nomear quantidades referenciadas, mas não identificadas, e faremos isso aqui, dando aos quatro inteiros os nomes a, b, c e d.

	Afirmação	Justificativa
1.	$x = a^2 + b^2 + c^2 + d^2$	Dado
2.	$a \geq 1; b \geq 1; c \geq 1; d \geq 1$	Dado
3.	$a^2 \geq 1; b^2 \geq 1; c^2 \geq 1; d^2 \geq 1$	(2) e propriedades da aritmética
4.	$x \geq 4$	(1), (3) e propriedades da aritmética
5.	$2^x \geq x^2$	(4) e Teorema 1.3

Figura 1.3: Uma prova formal do Teorema 1.4

Na etapa (2), consideramos a outra parte da hipótese do teorema: que cada um dos valores que estão sendo elevados ao quadrado é pelo menos 1. Tecnicamente, essa afirmação representa quatro afirmações distintas, uma para cada um dos quatro inteiros envolvidos. Assim, na etapa (3), observamos que, se um número é pelo menos 1, então seu quadrado também é pelo menos 1. Usamos como justificativa o fato de que a afirmação (2) é válida, e também "propriedades da aritmética". Isto é, supomos que o leitor conhece ou pode provar afirmações simples sobre como as desigualdades funcionam, tais como a afirmação "se $y \geq 1$, então $y^2 \geq 1$".

A etapa (4) utiliza as afirmações (1) e (3). A primeira afirmação nos diz que x é a soma dos quatro quadrados em questão, e a afirmação (3) nos diz que cada um dos quadrados é pelo menos 1. Usando mais uma vez propriedades bem conhecidas da aritmética, concluímos que x é pelo menos $1 + 1 + 1 + 1$, ou seja, 4.

Na última etapa, (5), usamos a afirmação (4), que é a hipótese do Teorema 1.3. O teorema propriamente dito é a justificativa para se escrever sua conclusão, pois sua hipótese é uma afirmação anterior. Tendo em vista que a afirmação (5), que é a conclusão do Teorema 1.3, também é a conclusão do Teorema 1.4, provamos assim o Teorema 1.4. Ou seja, começamos com a hipótese desse teorema e conseguimos deduzir sua conclusão. □

1.2.2 Redução a definições

Nos dois teoremas anteriores, a hipótese usou termos que devem ter sido familiares: por exemplo, inteiros, adição e multiplicação. Em muitos outros teoremas,

inclusive vários da teoria de autômatos, os termos usados no enunciado podem ter implicações menos óbvias. Um modo útil de proceder em muitas provas é:

- Se não tiver certeza de como iniciar uma prova, converta todos os termos da hipótese em suas definições.

Aqui está um exemplo de um teorema simples de provar, uma vez que tenhamos expressado seu enunciado em termos elementares. Ele utiliza as duas definições a seguir:

1. Um conjunto S é *finito* se existe um inteiro n tal que S tem exatamente n elementos. Escrevemos $||S|| = n$, onde $||S||$ é usado para denotar o número de elementos em um conjunto S. Se o conjunto S não é finito, dizemos que S é *infinito*. Intuitivamente, um conjunto infinito é um conjunto que contém mais que qualquer número inteiro de elementos.
2. Se S e T são ambos subconjuntos de algum conjunto U, então T é o *complemento* de S (em relação a U) se $S \cup T = U$ e $S \cap T = \emptyset$. Isto é, cada elemento de U está em exatamente um conjunto dentre S e T; em outras palavras, consiste exatamente nos elementos de U que não estão em S.

Teorema 1.5: Seja S um subconjunto finito de algum conjunto infinito U. Seja T o complemento de S em relação a U. Então, T é infinito.

PROVA: Intuitivamente, esse teorema diz que, se você tem um suprimento infinito de algo (U) e joga fora uma quantidade finita (S), então ainda lhe restará uma quantidade infinita. Vamos começar reafirmando os fatos do teorema como na Figura 1.4.

Afirmação original	Nova afirmação
S é finito	Existe um inteiro n tal que $\|\|S\|\| = n$
U é infinito	Para nenhum inteiro p temos $\|\|U\|\| = p$
T é o complemento de S	$S \cup T = U$ e $S \cap T = \emptyset$

Figura 1.4: Reafirmando os dados do Teorema 1.5

Ainda estamos sem saída, e assim precisamos usar uma técnica de prova comum chamada "prova por contradição". Nesse método de prova, que será discutido em detalhes na Seção 1.3.3, supomos que a conclusão é falsa. Então, usamos essa suposição, junto com partes da hipótese, para provar o oposto de uma das afirmações feitas na hipótese. Em seguida, mostramos que é impossível todas as partes da hipótese serem verdadeiras e a conclusão ser falsa ao mesmo tempo. A única possibilidade que resta é a conclusão ser verdadeira sempre que a hipótese é verdadeira. Isto é, o teorema é verdadeiro.

No caso do Teorema 1.5, a contradição da conclusão é "T é finito". Vamos supor que T é finito, juntamente com a afirmação da hipótese que diz que S é finito; ou seja, $||S|| = n$ para algum inteiro n. De modo semelhante, podemos reafirmar a suposição de que T é finito como $||T|| = m$ para algum inteiro m.

Agora uma das afirmações dadas nos diz que $S \cup T = U$ e $S \cap T = \emptyset$. Isto é, os elementos de U são exatamente os elementos de S e T. Desse modo, deve haver $n + m$ elementos de U. Como $n + m$ é um inteiro, e mostramos que $||U|| = n + m$, segue –se que U é finito. Mais precisamente, mostramos que o número de elementos em U é um algum inteiro, que é a definição de "finito". Porém, a afirmação de que U é finito contradiz a afirmação dada de que U é infinito. Portanto, usamos a contradição de nossa conclusão para provar a contradição de uma das afirmações dadas na hipótese e, pelo princípio de "prova por contradição", podemos concluir que o teorema é verdadeiro. □

As provas não têm de ser tão prolixas. Tendo visto as idéias por trás da prova, vamos voltar a provar o teorema em poucas linhas.

PROVA: (do Teorema 1.5) Sabemos que $S \cup T = U$ e que S e T são disjuntos, e assim $||S|| + ||T|| = ||U||$. Tendo em vista que S é finito, $||S|| = n$ para algum n e, como U é infinito, não existe nenhum inteiro p tal que $||U|| = p$. Assim, suponha que T seja finito; ou seja, $||T|| = m$ para algum inteiro m. Então, $||U|| = ||S|| + ||T|| = n + m$, o que contradiz a afirmação dada de que não existe nenhum inteiro p igual a $||U||$.

1.2.3 Outras formas de teoremas

A forma de teorema "se-então" é mais comum em áreas típicas da matemática. No entanto, também encontramos outros tipos de afirmações provadas como teoremas. Nesta seção, examinaremos as formas mais comuns de afirmações e o que em geral precisamos fazer para prová-las.

Maneiras de dizer "se-então"

Primeiro, existem vários tipos de enunciados de teoremas que parecem diferentes de uma simples forma "se H então C", mas que estão de fato dizendo o mesmo: se a hipótese H é verdadeira para um dado valor do(s) parâmetro(s), então a conclusão C é verdadeira para o mesmo valor. Aqui estão algumas outras maneiras de representar "se H então C".

1. H implica C.
2. H somente se C.
3. C se H.
4. Sempre que H é verdadeira, segue-se C.

Também podemos ver muitas variantes da forma (4), como "se H é verdadeira, então decorre C " ou "sempre que H é verdadeira, C é verdadeira".

Exemplo 1.6: O enunciado do Teorema 1.3 seria representado nessas quatro formas como:

1. $x \geq 4$ implica $2^x \geq x^2$.
2. $x \geq 4$ somente se $2^x \geq x^2$.
3. $2^x \geq x^2$ se $x \geq 4$.
4. Sempre que $x \geq 4$, segue-se que $2^x \geq x^2$.

□

Afirmações com quantificadores

Muitos teoremas envolvem afirmações que utilizam os *quantificadores* "para todo" e " existe", ou variações semelhantes, como "para cada" em lugar de "para todo". A ordem em que esses quantificadores aparecem afeta o significado da afirmação. Freqüentemente, é útil ver afirmações com mais de um quantificador como um "jogo" entre dois participantes – para todos e existe – que especificam alternadamente valores para os parâmetros mencionados no teorema. "Para-todo" deve considerar todas as opções possíveis, e assim as escolhas de "para-todo" em geral são deixadas como variáveis. Entretanto, "existe" só tem de escolher um valor, que pode depender dos valores escolhidos pelos jogadores anteriormente. A ordem em que os quantificadores aparecem na afirmação determina quem joga primeiro. Se o último jogador a fazer uma escolha sempre puder encontrar algum valor permitido, então a afirmação é verdadeira.

Por exemplo, considere uma definição alternativa de "conjunto infinito": o conjunto S é *infinito* se e somente se, para todo inteiro n, existe um subconjunto T de S com exatamente n elementos. Aqui, "para-todo" precede "existe", e assim devemos considerar um inteiro arbitrário n. Desta forma, "existe" escolhe um subconjunto T, e pode usar o conhecimento de n para fazer isso. Por exemplo, se S fosse o conjunto de inteiros, "existe" poderia escolher o subconjunto $T = \{1, 2, \ldots, n\}$ e assim obter sucesso, independente de n. Essa é uma prova de que o conjunto de inteiros é infinito.

A afirmação a seguir é semelhante à definição de "infinito", mas é *incorreta* porque inverte a ordem dos quantificadores: "existe um subconjunto T do conjunto S tal que, para todo n, o conjunto T tem exatamente n elementos". Agora, dado um conjunto S como o dos inteiros, o jogador "existe" pode escolher qualquer conjunto T; digamos que $\{1, 2, 5\}$ seja escolhido. Para essa escolha, o jogador "para-todo" deve mostrar que T tem n elementos para *todo* n possível. Porém, "para-todo" não pode fazê-lo. Por exemplo, a afirmação é falsa para $n = 4$ ou, de fato, para qualquer $n \neq 3$.

Além disso, em lógica formal, com freqüência vemos o operador $\rightarrow$ em lugar de "se-então". Isto é, a afirmação "se H então C" poderia aparecer como $H \rightarrow C$ em alguma literatura matemática; não a utilizaremos aqui.

Afirmações se-e-somente-se

Às vezes, encontramos uma afirmação da forma "A se e somente se B". Outras formas dessa afirmação são "A iff B"[1], "A é equivalente a B" ou "A exatamente quando B". Na realidade, essa afirmação é um par de afirmações se-então: "se A então B" e "se B então A". Provamos que "A se e somente se B" provando estas duas afirmações:

> **O quanto as provas precisam ser formais?**
>
> A resposta a essa pergunta não é fácil. A essência das provas é sua finalidade de convencer alguém, seja ele um avaliador do seu trabalho ou você mesmo, da correção de uma estratégia que você está utilizando em seu código. Se for convincente, ela será suficiente; se ela deixar de convencer o "consumidor" da prova, então ela será totalmente ignorada.
>
> A parte da incerteza relativa a provas vem do conhecimento diferente que o consumidor pode ter. Desse modo, no Teorema 1.4, supomos que você sabia tudo sobre aritmética e que acreditaria em uma afirmação como "se $y \geq 1$ então $y^2 \geq 1$". Se não estivesse familiarizado com a aritmética, teríamos de provar essa afirmação usando algumas etapas de nossa prova dedutiva.
>
> Contudo, existem certos detalhes obrigatórios em provas, e omiti-los sem dúvida torna a prova inadequada. Por exemplo, qualquer prova dedutiva que utilize afirmações não justificadas pelas afirmações dadas, ou anteriores, não pode ser adequada. Quando fazemos uma demonstração de uma afirmação "se e somente se", seguramente devemos ter uma prova da parte "se" e outra prova da parte "somente-se". Como um exemplo adicional, as provas indutivas (discutidas na Seção 1.4) exigem provas das partes base e indução.

1. A *parte se*: "se B então A", e
2. A *parte somente-se*: "se A então B", que freqüentemente é enunciada na forma equivalente "A somente se B".

As provas podem ser apresentadas em qualquer ordem. Em muitos teoremas, uma parte é decididamente mais fácil que a outra, e é habitual apresentar primeiro a direção fácil e deixá-la de lado.

---

[1] Iff (sse), abreviatura de "if and only if" ("se e somente se"), é uma não-palavra utilizada em alguns compêndios de matemática para ajudar a torná-los mais sucintos.

Em lógica formal, pode-se observar o operador $\leftrightarrow$ ou $\equiv$ denotar uma afirmação "se-e-somente-se". Isto é, $A \equiv B$ e $A \leftrightarrow B$ significam o mesmo que "A se e somente se B".

Ao provar uma afirmação se-e-somente-se, é importante lembrar que você deve provar tanto a parte "se" quanto a parte "somente-se". Às vezes, você achará útil desmembrar uma afirmação se-e-somente-se em uma sucessão de várias equivalências. Ou seja, para provar "A se e somente se B", você poderia provar primeiro "A se e somente se C", e depois provar "C se e somente se B". Esse método funciona, desde que você lembre que cada etapa se-e-somente-se deve ser provada em ambos os sentidos. Provar qualquer etapa em apenas um dos sentidos invalida a prova inteira.

O exemplo a seguir mostra uma prova se-e-somente-se simples. Ela usa as notações:

1. $\lfloor x \rfloor$, o *piso* do número real x, é o maior inteiro igual ou menor que x.
2. $\lceil x \rceil$, o *teto* do número real x, é o menor inteiro igual ou maior que x.

Teorema 1.7: Seja x um número real. Então, $\lfloor x \rfloor = \lceil x \rceil$ se e somente se x é um inteiro.

PROVA: (Parte somente-se) Nesta parte, supomos $\lfloor x \rfloor = \lceil x \rceil$ e tentamos provar que x é um inteiro. Usando as definições de piso e teto, notamos que $\lfloor x \rfloor \leq x$ e $\lceil x \rceil \geq x$. Porém, foi dado que $\lfloor x \rfloor = \lceil x \rceil$. Desse modo, podemos substituir o teto pelo piso na primeira desigualdade para concluir que $\lceil x \rceil \leq x$. Tendo em vista que $\lceil x \rceil \leq x$ e $\lceil x \rceil \geq x$ são válidas, podemos concluir pelas propriedades das desigualdades aritméticas que $\lceil x \rceil = x$. Como $\lceil x \rceil$ é sempre um inteiro, x também tem de ser um inteiro nesse caso.

(Parte se) Agora, supomos que x é um inteiro e tentamos provar $\lfloor x \rfloor = \lceil x \rceil$. Essa parte é fácil. Pelas definições de piso e teto, quando x é um inteiro, $\lfloor x \rfloor$ e $\lceil x \rceil$ são iguais a x e, portanto, são iguais entre si. □

1.2.4 Teoremas que parecem não ser afirmações se-então

Às vezes, encontramos um teorema que parece não ter uma hipótese. Um exemplo é o bem conhecido fato da trigonometria:

Teorema 1.8: $\text{sen}^2\,\theta + \cos^2\theta = 1$. □

Na realidade, essa afirmação tem uma hipótese, e a hipótese consiste em todas as afirmações que você precisa conhecer para interpretar a afirmação. Em particular, a hipótese oculta é que θ é um ângulo e, assim, as funções seno e cosseno têm seu significado habitual para ângulos. A partir das definições desses

termos e do Teorema de Pitágoras (em um triângulo retângulo, o quadrado da hipotenusa é igual à soma dos quadrados dos outros dois lados), você poderia provar o teorema. Em essência, a forma se-então do teorema é realmente: "se θ é um ângulo, então $\text{sen}^2\,\theta + \cos^2\theta = 1$".

1.3 Outras formas de provas

Nesta seção, estudaremos vários tópicos adicionais relativos à elaboração de provas:

1. Provas sobre conjuntos.
2. Provas por contradição.
3. Provas por contra-exemplo.

1.3.1 Provando equivalências entre conjuntos

Na teoria de autômatos, freqüentemente somos levados a provar um teorema que afirma que conjuntos construídos de duas maneiras diferentes são idênticos. Muitas vezes, esses conjuntos são conjuntos de strings de caracteres, e tais conjuntos são chamados "linguagens"; porém, nesta seção, a natureza dos conjuntos não é importante. Se E e F são duas expressões que representam conjuntos, a afirmação $E = F$ significa que os dois conjuntos representados são iguais. Mais precisamente, todo elemento no conjunto representado por E está no conjunto representado por F, e todo elemento no conjunto representado por F está no conjunto representado por E.

Exemplo 1.9: A *lei comutativa da união* afirma que podemos tomar a união de dois conjuntos R e S em qualquer ordem. Ou seja, $R \cup S = S \cup R$. Nesse caso, E é a expressão $R \cup S$ e F é a expressão $S \cup R$. A lei comutativa da união afirma que $E = F$. □

Podemos escrever uma igualdade de conjuntos $E = F$ como uma afirmação se-e-somente-se: um elemento x está em E se e somente se x está em F. Como conseqüência, vemos o esboço de uma prova de qualquer enunciado que afirma a igualdade de dois conjuntos $E = F$; ela segue a mesma forma de qualquer prova se-e-somente-se:

1. Provar que, se x está em E, então x está em F.
2. Provar que, se x está em F, então x está em E.

Como um exemplo desse processo de prova, vamos provar a *lei distributiva da união sobre a interseção*:

Teorema 1.10: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

PROVA: As duas expressões de conjuntos envolvidas são $E = R \cup (S \cap T)$ e

$$F = (R \cup S) \cap (R \cup T).$$

Provaremos cada uma das duas partes do teorema. Na parte "se" supomos que o elemento x está em E e mostramos que ele está em F. Essa parte, resumida na Figura 1.5, usa as definições de união e interseção; supomos que você esteja familiarizado com elas.

Então, temos de provar a parte "somente-se" do teorema. Aqui, supomos que x está em F e mostramos que ele está em E. As etapas são resumidas na Figura 1.6. Tendo em vista que provamos ambas as partes da afirmação se-e-somente-se, a lei distributiva da união sobre a interseção fica provada. □

1.3.2 A contrapositiva

Toda afirmação se-então tem uma forma equivalente que, em algumas circunstâncias, é mais fácil de provar. A *contrapositiva* da afirmação "se H então C" é "se não C então não H". Uma afirmação e sua antítese são ambas verdadeiras ou ambas falsas, e assim podemos provar qualquer uma delas para provar a outra.

	Afirmação	Justificativa
1.	x está em $R \cup (S \cap T)$	Dado
2.	x está em R ou x está em $R \cup (S \cap T)$	(1) e definição de união
3.	x está em R ou x está em S e T	(2) e definição de interseção
4.	x está em $R \cup S$	(3) e definição de união
5.	x está em $R \cup T$	(3) e definição de união
6.	x está em $(R \cup S) \cap (R \cup T)$	(4), (5) e definição de interseção

Figura 1.5: Etapas na parte "se" do Teorema 1.10

	Afirmação	Justificativa
1.	x está em $(R \cup S) \cap (R \cup T)$	Dado
2.	x está em $R \cup S$	(1) e definição de interseção
3.	x está em $R \cup T$	(1) e definição de interseção
4.	x está em R ou x está em S e T	(2), (3) e raciocínio sobre uniões
5.	x está em R ou x está em $S \cap T$	(4) e definição de interseção
6.	x está em $R \cup (S \cap T)$	(5) e definição de união

Figura 1.6: Etapas da parte "somente-se" do Teorema 1.10

Para ver por que "se H então C" e "se não C então não H" são logicamente equivalentes, primeiro observe que há quatro casos a considerar:

1. H e C verdadeiras.
2. H verdadeira e C falsa.
3. C verdadeira e H falsa.
4. H e C falsas.

Só existe uma maneira de tornar falsa uma afirmação se-então; a hipótese deve ser verdadeira e a conclusão falsa, como no caso (2). Nos outros três casos, inclusive no caso (4) em que a conclusão é falsa, a afirmação se-então propriamente dita é verdadeira.

Agora, considere em que casos a antítese "se não C, então não H" é falsa. Para que essa afirmação seja falsa, sua hipótese (que é "não C") deve ser verdadeira, e sua conclusão (que é "não H") deve ser falsa. Entretanto, "não C" é verdadeira exatamente quando C é falsa, e "não H" é falsa exatamente quando H é verdadeira. Essas duas condições são novamente o caso (2), o que mostra que, em cada um dos quatro casos, a afirmação original e sua antítese são ambas verdadeiras ou ambas falsas; isto é, elas são logicamente equivalentes.

Como dizer "se-e-somente-se" para conjuntos

Como mencionamos, os teoremas que declaram equivalências de expressões sobre conjuntos são afirmações se-e-somente-se. Desse modo, o Teorema 1.10 poderia ter sido declarado como: um elemento x está em $R \cup (S \cap T)$ se e somente se x está em

$$(R \cup S) \cap (R \cup T)$$

Outra expressão comum de uma equivalência entre conjuntos pode ser feita com a locução "todos-e-somente". Por exemplo, o Teorema 1.10 também poderia ser declarado como "os elementos de $R \cup (S \cap T)$ são todos e somente os elementos de

$$(R \cup S) \cap (R \cup T)$$

A recíproca

Não confunda os termos "contrapositiva" e "recíproca". A *recíproca* de uma afirmação se-então é o "outro sentido"; isto é, a recíproca de "se H então C" é "se C então H". Diferente da contrapositiva, que é logicamente equivalente à afirmação original, a recíproca *não* é equivalente à afirmação original. De fato, as duas partes de uma prova se-e-somente-se são sempre alguma afirmação e sua recíproca.

Exemplo 1.11: Lembramos o Teorema 1.3, cuja afirmação era: "se $x \geq 4$, então $2^x \geq x^2$". A contrapositiva dessa afirmação é "se não $2^x \geq x^2$ então não $x \geq 4$". Em termos mais coloquiais, fazendo uso do fato de que "não $a \geq b$" é o mesmo que $a < b$, a contrapositiva é "se $2^x < x^2$ então $x < 4$". □

Quando somos levados a provar um teorema se-e-somente-se, o uso da contrapositiva em uma das partes nos permite várias opções. Por exemplo, suponha que queremos provar a equivalência de conjuntos $E = F$. Em vez de provar "se x está em E então x está em F e se x está em F então x está em E", também poderíamos colocar um sentido na contrapositiva. Uma forma de prova equivalente é:

- Se x está em E então x está em F, e se x não está em E então x não está em F. Também poderíamos intercambiar E e F na afirmação anterior.

1.3.3 Prova por contradição

Outro modo de provar uma afirmação da forma "se H então C" é provar a afirmação

- "H e não C implica falsidade".

Ou seja, comece por supor a hipótese H e a negação da conclusão C. Complete a prova mostrando que algo que se sabe ser falso decorre logicamente de H e não C. Essa forma de prova é chamada *prova por contradição*.

Exemplo 1.12: Lembre-se do Teorema 1.5, em que provamos a afirmação se-então com a hipótese H = "U é um conjunto infinito, S é um subconjunto finito de U e T é o complemento de S em relação a U". A conclusão C era "T é infinito". Vamos provar esse teorema por contradição. Supomos "não C"; isto é, supomos que T era finito.

Nossa prova foi derivar uma falsidade de H e não C. Primeiro mostramos, a partir das suposições de que S e T são ambos finitos, que U também tem de ser finito. Contudo, como U foi declarado na hipótese H ser infinito e um conjunto não pode ser ao mesmo tempo finito e infinito, provamos que a afirmação lógica era "falsa". Em termos lógicos, temos ao mesmo tempo uma proposição p (U é finito) e sua negação, não p (U é infinito). Então, usamos o fato de que "p e não p" é logicamente equivalente a "falso". □

Para ver por que as provas por contradição são logicamente corretas, lembramos que na Seção 1.3.2 existem quatro combinações de valores verdade para H e C. Somente o segundo caso, H verdadeira e C falsa, torna falsa a afirmação "se H então C". Mostrando que H e não C leva à falsidade, estamos mostrando que o caso 2 não pode ocorrer. Portanto, as únicas combinações possíveis de valores verdade para H e C são as três combinações que tornam "se H então C" verdadeira.

1.3.4 Contra-exemplos

Na vida real, não somos levados a provar um teorema. Em vez disso, ficamos diante de algo que parece verdadeiro – por exemplo, uma estratégia para implementar um programa – e precisamos decidir se o "teorema" é ou não verdadeiro. Para resolver a questão, temos a alternativa de tentar provar o teorema e, se não pudermos, tentar provar que sua afirmação é falsa.

Em geral, os teoremas são afirmações sobre um número infinito de casos, talvez todos os valores de seus parâmetros. De fato, a convenção matemática rígida só irá distinguir uma afirmação com o título de "teorema" se ela tiver um número infinito de casos; as afirmações que não têm nenhum parâmetro, ou que se aplicam apenas a um número finito de valores de seus parâmetros, são chamadas *observações*. É suficiente mostrar que um suposto teorema é falso em qualquer caso isolado para mostrar que ele não é um teorema. A situação é análoga para programas, pois em geral consideramos que um programa tem um bug se ele deixa de operar corretamente até mesmo para uma única entrada com a qual deveria funcionar.

Com freqüência, é mais fácil provar que uma afirmação não é um teorema do que provar que ela *é* um teorema. Como mencionamos, se S é qualquer afirmação, então a afirmação "S não é um teorema" é ela própria uma afirmação sem parâmetros e, portanto, pode ser considerada uma observação em vez de um teorema. Aqui estão dois exemplos, o primeiro de um não-teorema óbvio, e o segundo de uma afirmação que simplesmente deixa de ser um teorema, e que exige alguma investigação antes que se resolva se a questão é ou não um teorema.

Suposto Teorema 1.13: Todos os primos são ímpares. (Mais formalmente, poderíamos dizer: se o inteiro x é primo, então x é ímpar.)

REFUTAÇÃO: O inteiro 2 é primo, mas é par. □

Agora, vamos discutir um "teorema" que envolve aritmética modular. Não existe uma definição essencial que tenhamos de estabelecer primeiro. Se a e b são inteiros positivos, então $a \bmod b$ é o resto da divisão de a por b; isto é, o único inteiro r entre 0 e $b-1$ tal que $a = qb + r$ para algum inteiro q. Por exemplo, $8 \bmod 3 = 2$, e $9 \bmod 3 = 0$. Nosso primeiro teorema proposto, que definiremos como falso, é:

Suposto Teorema 1.14: Não existe par de inteiros a e b tais que

$$a \bmod b = b \bmod a$$

□

Quando somos solicitados a realizar ações com pares de objetos, como a e b neste caso, com freqüência é possível simplificar o relacionamento entre os

dois, tirando proveito da simetria. Nesse caso, podemos nos concentrar na situação em que $a < b$ pois, se $b < a$, podemos trocar a e b e obter a mesma equação do Suposto Teorema 1.14. Entretanto, devemos ser cuidadosos para não esquecer o terceiro caso, em que $a = b$. Esse caso acaba sendo fatal para nossas tentativas de prova.

Vamos supor que $a < b$. Então, $a \bmod b = a$ pois, na definição de $a \bmod b$, temos $q = 0$ e $r = a$. Ou seja, quando $a < b$, temos $a = 0 \times b + a$. Porém, $b \bmod a < a$, pois nenhum inteiro mod a está entre 0 e $a - 1$. Desse modo, quando $a < b$, $b \bmod a < a \bmod b$, e assim $a \bmod b = b \bmod a$ é impossível. Usando o argumento de simetria anterior, também sabemos que $a \bmod b \neq b \bmod a$ quando $b < a$.

No entanto, considere o terceiro caso: $a = b$. Tendo em vista que $x \bmod x = 0$ para qualquer inteiro x, *temos* $a \bmod b = b \bmod a$ se $a = b$. Desse modo, temos uma refutação do suposto teorema:

REFUTAÇÃO: (do Suposto Teorema 1.14) Seja $a = b = 2$. Então

$$a \bmod b = b \bmod a = 0$$

□

No processo de encontrar o contra-exemplo, descobrimos de fato as condições exatas sob as quais o suposto teorema é verdadeiro. Aqui estão a versão correta do teorema e sua prova.

Teorema 1.15: $a \bmod b = b \bmod a$ se e somente se $a = b$.

PROVA: (Parte se) Suponha $a = b$. Então, como observamos anteriormente, $x \bmod x = 0$ para qualquer inteiro x. Desse modo, $a \bmod b = b \bmod a = 0$ sempre que $a = b$.

(Parte somente-se) Agora suponha que $a \bmod b = b \bmod a$. A melhor técnica é uma prova por contradição, e assim supomos além disso a negação da conclusão; isto é, supomos $a \neq b$. Então, como $a = b$ foi eliminado, só temos de considerar os casos $a < b$ e $b < a$.

Já observamos antes que, quando $a < b$, temos $a \bmod b = a$ e $b \bmod a < a$. Portanto, essas afirmações, em conjunto com a hipótese $a \bmod b = b \bmod a$ nos permitem derivar uma contradição.

Por simetria, se $b < a$, então $b \bmod a = b$ e $a \bmod b < b$. Novamente derivamos uma contradição da hipótese e concluímos que a parte somente-se também é verdadeira. Assim, provamos ambos os sentidos e concluímos que o teorema é verdadeiro. □

1.4 Provas indutivas

Existe uma forma especial de prova, chamada "indutiva", que é essencial ao se lidar com objetos definidos recursivamente. Muitas das provas indutivas mais

familiares lidam com inteiros mas, na teoria de autômatos, também precisamos de provas indutivas sobre conceitos definidos recursivamente, tais como árvores e expressões de vários tipos, como as expressões regulares que foram mencionadas rapidamente na Seção 1.1.2. Nesta seção, introduziremos o assunto de provas indutivas, iniciando com induções "simples" sobre inteiros. Em seguida, mostraremos como executar induções "estruturais" sobre qualquer conceito definido recursivamente.

1.4.1 Induções sobre inteiros

Suponha que temos de provar uma afirmação $S(n)$ sobre um inteiro n. Uma abordagem comum envolve provar duas coisas:

1. A *base*, onde mostramos $S(i)$ para um inteiro i específico. Em geral, $i = 0$ ou $i = 1$, mas há exemplos em que queremos começar com algum i mais alto, talvez porque a afirmação S é falsa para alguns inteiros pequenos.
2. A *etapa indutiva*, em que supomos $n \geq i$, onde i é o inteiro da base, e mostramos que "se $S(n)$ então $S(n + 1)$".

Intuitivamente, essas duas partes devem nos convencer de que $S(n)$ é verdadeira para todo inteiro n que seja igual ou maior que o inteiro i da base. Podemos demonstrar isso da seguinte forma. Suponha que $S(n)$ fosse falsa para um ou mais desses inteiros. Então, teria de haver um valor mínimo de n, digamos j, para o qual $S(j)$ seria falsa, e ainda $j \geq i$. Evidentemente, j não poderia ser i, porque provamos na parte da base que $S(i)$ é verdadeira. Portanto, j deve ser maior que i. Agora, sabemos que $j - 1 \geq i$, e $S(j - 1)$ é verdadeira.

No entanto, provamos na parte indutiva que, se $n \geq i$, então $S(n)$ implica $S(n + 1)$. Suponha que fazemos $n = j - 1$. Então, sabemos a partir da etapa indutiva que $S(j - 1)$ implica $S(j)$. Tendo em vista que também conhecermos $S(j - 1)$, podemos concluir $S(j)$.

Nesta demonstração supusemos a negação daquilo que queríamos provar; isto é, que $S(j)$ era falsa para algum $j \geq i$. Em cada caso, derivamos uma contradição, e assim temos uma "prova por contradição" de que $S(n)$ é verdadeira para todo $n \geq i$.

Infelizmente, existe uma falha lógica sutil no raciocínio anterior. Nossa suposição de que podemos escolher um $j \geq i$ mínimo para o qual $S(j)$ é falsa depende em primeiro lugar de nossa crença no princípio da indução. Ou seja, a única maneira de provar que podemos encontrar tal j é usar um método que seja essencialmente uma prova indutiva. Porém, a "prova" descrita anteriormente faz sentido em termos intuitivos e corresponde à nossa compreensão do mundo real. Assim, em geral a tomamos como uma parte integral de nosso sistema de raciocínio lógico:

- *Princípio de indução*: Se provamos $S(i)$ e provamos que, para todo $n \geq i$, $S(n)$ implica $S(n+1)$, então podemos concluir que $S(n)$ para todo $n \geq i$.

Os dois exemplos seguintes ilustram o uso do princípio de indução para provar teoremas sobre inteiros.

Teorema 1.16: Para todo $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

PROVA: A prova é feita em duas partes: a base e a etapa indutiva; provaremos cada uma separadamente.

BASE: Para a base, escolhemos $n = 0$. Pode parecer surpreendente que o teorema até mesmo faça sentido para $n = 0$, pois o lado esquerdo da Equação (1.1) é $\sum_{i=1}^{0}$ quando $n = 0$. Contudo, existe um princípio geral de que, quando o limite superior de um somatório (0, nesse caso) for menor que o limite inferior (no caso, 1), o somatório não tem nenhum termo e, portanto, é 0. Isto é, $\sum_{i=1}^{0} i^2 = 0$.

O lado direito da Equação (1.1) também é 0, pois $0 \times (0+1) \times (2 \times 0 + 1)/6 = 0$. Desse modo, a Equação (1.1) é verdadeira quando $n = 0$.

INDUÇÃO: Agora, suponha $n \geq 0$. Devemos provar a etapa indutiva, de que a Equação (1.1) implica a mesma fórmula com $n + 1$ em lugar de n. A última fórmula é:

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

Podemos simplificar as Equações (1.1) e (1.2) expandindo as somas e os produtos no lado direito. Essas equações se tornam:

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n) / 6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6) / 6 \tag{1.4}$$

Precisamos provar (1.4) usando (1.3), pois, no princípio de indução, essas são as afirmações $S(n+1)$ e $S(n)$, respectivamente. O "truque" é desmembrar o somatório até $n + 1$ no lado direito de (1.4) em um somatório até n mais o $(n + 1)$-ésimo termo. Portanto, podemos substituir o somatório até n pelo lado esquerdo de (1.3) e mostrar que (1.4) é verdadeira. Essas etapas são as seguintes:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6) / 6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n) / 6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6) / 6 \tag{1.6}$$

A verificação final de que (1.6) é verdadeira só exige álgebra polinominal simples no lado esquerdo para mostrar que ele é idêntico ao lado direito. □

Exemplo 1.17: No próximo exemplo, provamos o Teorema 1.3 da Seção 1.2.1. Lembre-se de que esse teorema afirma que, se $x \geq 4$, então $2^x \geq x^2$. Fornecemos uma prova informal baseada na idéia que a razão $x^2/2^x$ diminui à medida que x cresce acima de 4. Podemos tornar a idéia precisa se provarmos a afirmação $2^x \geq x^2$ por indução sobre x, começando com a base $x = 4$. Note que a afirmação é realmente falsa para $x < 4$.

BASE: Se $x = 4$, então 2^x e x^2 são ambos 16. Desse modo, $2^4 \geq 4^2$ é verdadeira.

INDUÇÃO: Suponha para algum $x \geq 4$ que $2^x \geq x^2$. Com essa afirmação como hipótese, precisamos provar a mesma afirmação, com $x + 1$ em lugar de x; ou seja, $2^{[x+1]} \geq [x+1]^2$. Essas são as afirmações $S(x)$ e $S(x+1)$ do princípio de indução; o fato de estarmos usando x em lugar de n como parâmetro não deve ser motivo de preocupação: x ou n é apenas uma variável local.

Como no Teorema 1.16, devemos reescrever $S(x+1)$ de modo que ela possa usar $S(x)$. Nesse caso, podemos escrever $2^{[x+1]}$ como 2×2^x. Tendo em vista que $S(x)$ nos informa que $2^x \geq x^2$, podemos concluir que $2^{[x+1]} = 2 \times 2^x + 1 \geq 2x^2$.

Porém, precisamos de algo diferente; precisamos mostrar que $2^{x+1} \geq (x+1)^2$. Um modo de provar essa afirmação é provar que $2x^2 \geq (x+1)^2$ e depois usar a transitividade de $\geq$ para mostrar que $2^{x+1} \geq 2x^2 \geq (x+1)^2$. Em nossa prova de que

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

podemos usar a suposição de que $x \geq 4$. Comece simplificando (1.7):

$$x^2 \geq 2x + 1 \tag{1.8}$$

Divida (1.8) por x para obter:

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

Como $x \geq 4$, sabemos que $1/x \leq 1/4$. Desse modo, o lado esquerdo de (1.9) é pelo menos 4, e o lado direito é no máximo 2,25. Assim, provamos a verdade de (1.9). Logo, as Equações (1.8) e (1.7) também são verdadeiras. Por sua vez, a Equação (1.7) nos dá $2x^2 \geq (x+1)^2$ para $x \geq 4$ e nos permite provar a afirmação $S(x+1)$ que, como sabemos, era $2^{x+1} \geq (x+1)^2$. □

Inteiros como conceitos definidos recursivamente

Mencionamos que as provas indutivas são úteis quando o tema é definido recursivamente. Entretanto, nossos primeiros exemplos foram induções sobre inteiros, que normalmente não imaginamos como "definidos recursivamente". Contudo, existe uma definição recursiva natural de quando um número é um inteiro não negativo, e essa definição realmente corresponde ao modo como ocorrem as induções sobre inteiros: dos objetos definidos primeiro para aqueles definidos mais tarde.

BASE: 0 é um inteiro.

INDUÇÃO: Se n é um inteiro, então $n + 1$ também é.

1.4.2 Formas mais gerais de induções sobre inteiros

Às vezes, uma prova indutiva só se torna possível usando-se um esquema mais geral que o proposto na Seção 1.4.1, na qual provamos uma afirmação S para um valor de base e então provamos que "se $S(n)$ então $S(n + 1)$". Duas generalizações importantes desse esquema são:

1. Podemos usar vários casos de base. Isto é, provamos $S(i), S(i + 1),..., S(j)$ para algum $j > i$.

2. Na prova de $S(n + 1)$, podemos usar a verdade de todas as afirmações

$$S(i), S(i + 1),..., S(n)$$

em vez de usar apenas $S(n)$. Além disso, se provamos casos de base até $S(j)$, então podemos supor que $n \geq j$, em vez de $n \geq i$.

A conclusão a ser tirada dessa base e da etapa indutiva é que $S(n)$ é verdadeira para todo $n \geq i$.

Exemplo 1.18: O exemplo a seguir ilustrará o potencial de ambos os princípios. A afirmação $S(n)$ que gostaríamos de provar é que, se $n \geq 8$, então n pode ser escrito como uma soma de valores 3 e 5. A propósito, note que 7 não pode ser escrito como uma soma de valores 3 e 5.

BASE: Os casos de base são $S(8)$, $S(9)$ e $S(10)$. As provas são $8 = 3 + 5$, $9 = 3 + 3 + 3$ e $10 = 5 + 5$, respectivamente.

INDUÇÃO: Suponha que $n \geq 10$ e que $S(8), S(9),..., S(n)$ são verdadeiras. Devemos provar $S(n + 1)$ a partir desses fatos dados. Nossa estratégia é subtrair 3 de $n + 1$, observar que deve ser possível escrever esse número como uma soma de valores 3 e 5, e adicionar mais um 3 à soma para obter um modo de escrever $n + 1$.

Mais formalmente, observe que $n - 2 \geq 8$, e assim podemos supor que $S(n - 2)$. Isto é, $n - 2 = 3a + 5b$ para alguns inteiros a e b. Então, $n + 1 = 3 + 3a + 5b$, e assim $n + 1$ pode ser escrito como a soma de $a + 1$ valores 3 e b valores 5. Isso prova $S(n + 1)$ e conclui a etapa indutiva. □

1.4.3 Induções estruturais

Na teoria de autômatos, existem várias estruturas definidas recursivamente das quais precisamos para provar afirmações. As noções familiares de árvores e expressões são exemplos importantes. Como induções, todas as definições recursivas têm um caso de base, no qual uma ou mais estruturas elementares são definidas, e uma etapa indutiva, em que estruturas mais complexas são definidas em termos das estruturas definidas previamente.

Exemplo 1.19: Aqui está a definição recursiva de uma árvore:

BASE: Um único nó é uma árvore, e esse nó é a *raiz* da árvore.

INDUÇÃO: Se $T_1, T_2, \ldots, T_k$ são árvores, então podemos formar uma nova árvore da seguinte forma:

1. Comece com um novo nó N, a raiz da árvore.
2. Adicione cópias de todas as árvores $T_1, T_2, \ldots, T_k$.
3. Adicione arestas desde o nó N até as raízes de cada uma das árvores $T_1, T_2, \ldots, T_k$.

A Figura 1.7 mostra a construção indutiva de uma árvore com raiz N a partir de k árvores menores. □

Exemplo 1.20: Aqui temos outra definição recursiva. Dessa vez, definimos expressões usando os operadores aritméticos + e *, com números e variáveis permitidas como operandos.

BASE: Qualquer número ou letra (isto é, uma variável) é uma expressão.

INDUÇÃO: Se E e F são expressões, então $E + F$, $E * F$ e (E) também o são.

Por exemplo, 2 e x são expressões pela base. A etapa indutiva nos diz que $x + 2$, $(x + 2)$ e $2 * (x + 2)$ também são expressões. Note que cada uma dessas expressões depende das anteriores também o serem. □

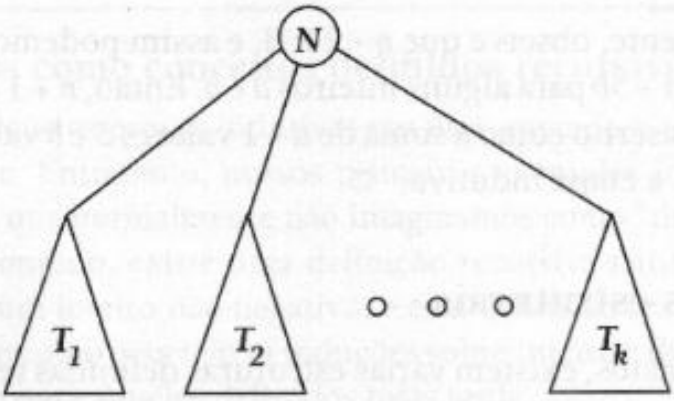

Figura 1.7: Construção indutiva de uma árvore

Intuição por trás da indução estrutural

Podemos sugerir informalmente por que a indução estrutural é um método de prova válido. Imagine a definição recursiva estabelecendo que cada uma de certas estruturas $X_1, X_2, \ldots$ satisfazem à definição. Os elementos de base vêm primeiro, e o fato de que X_i está no conjunto definido de estruturas só pode depender da pertinência ao conjunto definido de estruturas que precedem X_i na lista. Vista desse modo, uma indução estrutural não é nada além de uma indução sobre o inteiro n da afirmação $S(X_n)$. Essa indução pode ser da forma generalizada descrita na Seção 1.4.2, com vários casos de base e uma etapa indutiva que usa todas as instâncias anteriores da afirmação. No entanto, devemos lembrar, como explicamos na Seção 1.4.1, que essa intuição não é uma prova formal e, de fato, temos de supor a validade desse princípio de indução como fizemos no caso da validade do princípio de indução original daquela seção.

Quando temos uma definição recursiva, podemos provar teoremas sobre ela, usando a forma de prova a seguir, chamada *indução estrutural*. Seja $S(X)$ uma afirmação sobre as estruturas X definidas por alguma definição recursiva específica.

1. Como base, prove $S(X)$ para a(s) estrutura(s) de base X.
2. Para a etapa indutiva, tome uma estrutura X que a definição recursiva nos diz que é formada a partir de $Y_1, Y_2, \ldots, Y_K$. Suponha as afirmações $S(Y_1), S(Y_2), \ldots, S(Y_K)$ e use essas afirmações para provar $S(X)$.

Nossa conclusão é que $S(X)$ é verdadeira para todo X. Os dois teoremas seguintes são exemplos de fatos que podem ser provados sobre árvores e expressões.

Teorema 1.21: Toda árvore tem um nó a mais que seu número de arestas.

PROVA: A afirmação formal $S(T)$ que precisamos provar por indução estrutural é: "se T é uma árvore, e T tem n nós e e arestas, então $n = e + 1$".

BASE: A base ocorre quando T é um único nó. Então, $n = 1$ e $e = 0$, e assim o relacionamento $n = e + 1$ é verdadeiro.

INDUÇÃO: Seja T uma árvore construída pela etapa indutiva da definição, a partir do nó de raiz N e de k árvores menores $T_1, T_2, \ldots, T_k$. Podemos supor que as afirmações $S(T_i)$ são verdadeiro para $i = 1,2,\ldots,k$. Isto é, seja T_i com n_i nós e e_i arestas; então $n_i = e_i + 1$.

Os nós de T são o nó N e todos os nós das árvores T_i. Desse modo, existem $1 + n_1 + n_2 + \ldots + n_k$ nós em T. As arestas de T são as k arestas que adicionamos explicitamente na etapa de definição indutiva, mais as arestas das árvores T_i. Conseqüentemente, T tem

$$k + e_1 + e_2 + \ldots + e_k \tag{1.10}$$

arestas. Se substituirmos n_i por $e_i + 1$ na contagem do número de nós de T, descobriremos que T tem

$$1 + [e_1 + 1] + [e_2 + 1] + \cdots + [e_k + 1] \tag{1.11}$$

nós. Tendo em vista que existem k termos "+1" em (1.10), podemos reagrupar (1.11) como

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

Essa expressão é exatamente uma unidade maior que a expressão de (1.10) que foi dada para o número de arestas de T. Assim, T tem um nó a mais que seu número de arestas. □

Teorema 1.22: Toda expressão tem um número igual de parênteses à esquerda e à direita.

PROVA: Formalmente, provamos a afirmação $S(G)$ sobre qualquer expressão G definida pela recursão do Exemplo 1.20: os números de parênteses à esquerda e à direita em G são iguais.

BASE: Se G é definida pela base, então G é um número ou uma variável. Essas expressões têm 0 parênteses à esquerda e 0 parênteses à direita, e assim os números são iguais.

INDUÇÃO: Há três regras pelas quais a expressão G pode ter sido construída de acordo com a etapa indutiva na definição:

1. $G = E + F$.
2. $G = E * F$.
3. $G = (E)$.

Podemos supor que $S(E)$ e $S(F)$ são verdadeiras; isto é, E tem o mesmo número de parênteses à esquerda e à direita, digamos n de cada, e F também tem o mesmo número de parênteses à esquerda e à direita, digamos m de cada. Então, podemos calcular os números de parênteses à esquerda e à direita em G para cada um dos três casos, como:

1. Se $G = E + F$, então G tem $n + m$ parênteses à esquerda e $n + m$ parênteses à direita; de cada tipo, n vem de E e m vem de F.
2. Se $G = E * F$, a contagem de parênteses para G é mais uma vez $n + m$ de cada, pela mesma razão do caso (1).
3. Se $G = (E)$, então existem $n + 1$ parênteses à esquerda em G – um parênte-se à esquerda é mostrado explicitamente, e os outros n estão presentes em E. Do mesmo modo, existem $n + 1$ parênteses à direita em G; um é explícito, e os outros n estão em E.

Em cada um dos três casos, vemos que os números de parênteses à esquerda e à direita em G são iguais. Essa observação completa a etapa indutiva e também a prova. □

1.4.4 Induções mútuas

Às vezes não podemos provar uma única afirmação por indução, mas em vez disso precisamos provar um grupo de afirmações $S_1(n), S_2(n), \ldots, S_k(n)$ juntas por indução sobre n. A teoria dos autômatos proporciona muitas dessas situações. No Exemplo 1.23, mostramos a situação comum em que precisamos explicar o que um autômato faz provando um grupo de afirmações, uma para cada estado. Essas afirmações informam sob quais seqüências de entradas o autômato entra em cada um dos estados.

No sentido estrito, provar um grupo de afirmações não é diferente de provar a *conjunção* (AND lógico) de todas as afirmações. Por exemplo, o grupo de afirmações $S_1(n), S_2(n), \ldots, S_k(n)$ poderia ser substituído pela única afirmação $S_1(n)$ AND $S_2(n)$ AND ... AND $S_k(n)$. Porém, quando há realmente várias afirmações independentes a provar, em geral é menos confuso manter as afirmações separadas e provar todas elas em suas próprias partes das etapas de base e indutivas. Chamamos esse tipo de prova de *indução mútua*. Um exemplo ilustrará as etapas necessárias de uma recursão mútua.

Exemplo 1.23: Vamos rever o interruptor liga/desliga, que representamos como um autômato no Exemplo 1.1. O próprio autômato é reproduzido na Figura 1.8. Tendo em vista que pressionar o botão alterna o estado entre *ligado* e *desligado*, e como o interruptor começa no estado *desligado*, esperamos que as afirmações a seguir expliquem em conjunto a operação do interruptor:

$S_1(n)$: O autômato está no estado *desligado* após n acionamentos se e somente se n é par.

$S_2(n)$: O autômato está no estado *ligado* após n acionamentos se e somente se n é ímpar.

Figura 1.8: Repetição do autômato da Figura 1.1

Poderíamos supor que S_1 implica S_2 e *vice-versa*, pois sabemos que um número n não pode ser ao mesmo tempo par e ímpar. Contudo, nem sempre é verdade que um autômato está em um e somente um estado. Ocorre que o autômato da Figura 1.8 está sempre em exatamente um estado, mas esse fato tem de ser provado como parte da indução mútua.

Damos a parte base e a parte indutiva das provas das afirmações $S_1(n)$ e $S_2(n)$ a seguir. As provas dependem de vários fatos sobre inteiros ímpares e pares: se adicionarmos ou subtrairmos 1 de um inteiro par, obteremos um inteiro ímpar e, se adicionarmos ou subtrairmos 1 de um inteiro ímpar, obteremos um inteiro par.

BASE: Para a base, escolhemos $n = 0$. Tendo em vista que existem duas afirmações, cada uma das quais deve ser provada em ambos os sentidos (porque S_1 e S_2 são afirmações "se-e-somente-se"), existem realmente quatro casos para a base e também quatro casos para a indução.

1. [S_1; Se] Como 0 é de fato par, devemos mostrar que, depois de 0 acionamentos, o autômato da Figura 1.8 está no estado *desligado*. Como esse é o estado inicial, o autômato realmente está no estado *desligado* depois de 0 acionamentos.

2. [S_1; Somente-se] O autômato está no estado *desligado* depois de 0 acionamentos; assim, devemos mostrar que 0 é par. Contudo, 0 é par pela definição de "par", e assim não há mais nada a provar.

3. [S2; Se] A hipótese da parte "se" de S_2 é que 0 é ímpar. Como essa hipótese H é falsa, qualquer afirmação da forma "se H então C" é verdadeira,

como discutimos na Seção 1.3.2. Desse modo, essa parte da base também é verdadeira.

4. [S_2; Somente-se] A hipótese de que o autômato está no estado *ligado* depois de 0 acionamentos também é falsa, pois o único modo de chegar ao estado *ligado* é seguir um arco identificado como *Pressionar*, o que exige que o botão seja pressionado pelo menos uma vez. Tendo em vista que a hipótese é falsa, novamente podemos concluir que a afirmação se-então é verdadeira.

INDUÇÃO: Agora, supomos que $S_1(n)$ e $S_2(n)$ são verdadeiras e tentamos provar $S_1(n+1)$ e $S_2(n+1)$. Novamente, a prova se divide em quatro partes.

1. [$S_1(n+1)$; Se] A hipótese para essa parte é que $n+1$ é par. Desse modo, n é ímpar. A parte "se" da afirmação $S_2(n)$ diz que, depois de n acionamentos, o autômato está no estado *ligado*. O arco de *ligado* até *desligado* identificado como *Pressionar* nos informa que o $(n+1)$-ésimo acionamento fará o autômato entrar no estado *desligado*. Isso completa a prova da parte "se" de S_1 $(n+1)$.

2. [S_1 $(n+1)$; Somente-se] A hipótese é que o autômato está no estado *desligado* depois de $n+1$ acionamentos. A inspeção do autômato da Figura 1.8 nos diz que o único modo de chegar ao estado *desligado* depois de um ou mais movimentos é estar no estado *ligado* e receber uma entrada *Pressionar*. Desse modo, se estamos no estado *desligado* depois de $n+1$ acionamentos, devemos ter ficado no estado *ligado* depois de n acionamentos. Então, podemos usar a parte "somente-se" da afirmação $S_2(n)$ para concluir que n é ímpar. Conseqüentemente, $n+1$ é par, que é a conclusão desejada para a parte somente-se de $S_1(n+1)$.

3. [$S_2(n+1)$; Se] Essa parte é em essência igual à parte (1), com as funções das afirmações S_1 e S_2 permutadas, e com as funções de "ímpar" e "par" permutadas. O leitor deve ser capaz de construir essa parte da prova facilmente.

4. [$S_2(n+1)$; Somente-se] Essa parte é em essência igual à parte (2), com as funções das afirmações S_1 e S_2 permutadas, e com as funções de "ímpar" e "par" permutadas.

□

Podemos abstrair do Exemplo 1.23 o padrão para todas as induções mútuas:

- Cada uma das afirmações deve ser provada separadamente na base e na etapa indutiva.
- Se as afirmações são "se-e-somente-se", então ambos os sentidos de cada afirmação devem ser provados, tanto na base quanto na indução.

1.5 Os conceitos centrais da teoria de autômatos

Nesta seção, introduziremos as definições mais importantes dos termos que permeiam a teoria de autômatos. Esses conceitos incluem o "alfabeto" (um conjunto de símbolos), "strings" (uma lista de símbolos de um alfabeto) e "linguagem" (um conjunto de strings de um mesmo alfabeto).

1.5.1 Alfabetos

Um *alfabeto* é um conjunto de símbolos finito e não-vazio. Convencionalmente, usamos o símbolo Σ para um alfabeto. Os alfabetos comuns incluem:

1. $\Sigma = \{0, 1\}$, o alfabeto *binário*.
2. $\Sigma = \{a, b, \ldots, z\}$, o conjunto de todas as letras minúsculas.
3. O conjunto de todos os caracteres ASCII, ou o conjunto de todos os caracteres ASCII imprimíveis.

1.5.2 Strings

Um *string* (ou às vezes *palavra* ou também *cadeia*) é uma seqüência finita de símbolos escolhidos de algum alfabeto. Por exemplo, 01101 é um string do alfabeto binário $\Sigma = \{0, 1\}$. O string 111 é outro string escolhido nesse alfabeto.

O string vazio

O *string vazio* é o string com zero ocorrências de símbolos. Esse string, denotado por ε, é um string que pode ser escolhido de qualquer alfabeto.

Comprimento de um string

Com freqüência, é útil para classificar strings por seu *comprimento*, isto é, o número de posições para símbolos no string. Por exemplo, 01101 tem comprimento 5. É comum dizer que o comprimento de um string é o "número de símbolos" no string; essa afirmação é coloquialmente aceita, mas não é estritamente correta. Desta forma, existem só dois símbolos, 0 e 1, no string 01101, mas existem cinco *posições* para símbolos, e seu comprimento é 5. Porém, em geral você deve esperar que o "número de símbolos" possa ser usado quando o significado for "número de posições".

A notação padrão para o comprimento de um string w é $|w|$. Por exemplo, $|011| = 3$ e $|\varepsilon| = 0$.

Potências de um alfabeto

Se Σ é um alfabeto, podemos expressar o conjunto de todos os strings de um certo comprimento a parte desse alfabeto, usando uma notação exponencial. Definimos Σ^k como o conjunto de strings de comprimento k, e o símbolo de cada um deles está em Σ.

Exemplo 1.24: Observe que $\Sigma^0 = \{f\}$, independente de qual alfabeto seja Σ. Isto é, ε é o único string cujo comprimento é 0.

Se $\Sigma = \{0,1\}$, então $\Sigma^1 = \{0,1\}$, $\Sigma^2 = \{00,01,10,11\}$,

$$\Sigma^3 = \{000,001,010,011, 100,101,110,111\}$$

e assim por diante. Observe que existe uma ligeira confusão entre Σ e Σ^1. O primeiro é um alfabeto; seus elementos 0 e 1 são símbolos. O outro é um conjunto de strings; seus elementos são os strings 0 e 1, cada um dos quais tem comprimento 1. Não tentaremos usar notações distintas para os dois conjuntos, confiando no contexto para tornar claro se $\{0, 1\}$ ou conjuntos semelhantes são alfabetos ou conjuntos de strings. □

Convenção de tipo para símbolos e strings

Comumente, usaremos letras minúsculas do início do alfabeto (ou dígitos) para denotar símbolos, e letras minúsculas próximas ao fim do alfabeto, em geral w, x, y e z, para denotar strings. Você deve tentar se acostumar a essa convenção, para ajudá-lo a se lembrar dos tipos de elementos que estão sendo discutidos.

O conjunto de todos os strings sobre um alfabeto Σ é denotado convencionalmente por Σ^*. Por exemplo, $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000,...\}$. Em outros termos,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Às vezes, desejamos excluir o string vazio do conjunto de strings. O conjunto de strings não vazios do alfabeto Σ é denotado por Σ^+. Desse modo, duas equivalências apropriadas são:

- $\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$.
- $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

Concatenação de strings

Sejam os strings x e y. Então, xy denota a *concatenação* de x e y, isto é, o string formado tomando-se uma cópia de x e acrescentando-se a ela uma cópia de y. Mais precisamente, se x é o string composto de i símbolos $x = a_1a_2 \ldots a_i$ e y é o string composto de j símbolos $y = b_1b_2 \ldots b_j$, então xy é o string de comprimento $i + j$: $xy = a_1a_2 \ldots a_i\, b_1b_2 \ldots b_j$.

Exemplo 1.25: Sejam x = 01101 e y = 110. Então xy = 01101110 e yx = 11001101. Para qualquer string w, as equações $\varepsilon w = w\varepsilon = w$ são válidas. Isto é, ε é a *identidade para a concatenação*, pois, quando é concatenado com qualquer string, ele produz o mesmo string como resultado (de forma análoga ao modo como 0, a identidade para a adição, pode ser somado a qualquer para número x e produz x como resultado). □

1.5.3 Linguagens

Um conjunto de strings, todos escolhidos a partir de algum Σ^*, onde Σ é um alfabeto específico, é chamado *linguagem*. Se Σ é um alfabeto, e $L \subseteq \Sigma^*$, então L é uma *linguagem sobre* Σ. Note que uma linguagem sobre Σ não precisa incluir strings com todos os símbolos de Σ; assim, uma vez que estabelecemos que L é uma linguagem sobre Σ, também sabemos que ela é uma linguagem sobre qualquer alfabeto que seja um superconjunto de Σ.

A escolha do termo "linguagem " pode parecer estranha. Porém, linguagens comuns podem ser vistas como conjuntos de strings. Um exemplo é o português, no qual uma coleção de palavras válidas em português é um conjunto de strings sobre o alfabeto que consiste em todas as letras. Outro exemplo é C, ou qualquer outra linguagem de programação, na qual os programas válidos são um subconjunto dos strings possíveis que podem ser formados a partir do alfabeto da linguagem. Esse alfabeto é um subconjunto dos caracteres ASCII. O alfabeto exato pode diferir ligeiramente entre diferentes linguagens de programação mas, em geral, inclui as letras maiúsculas e minúsculas, os dígitos, a pontuação e símbolos matemáticos.

Contudo, também existem muitas outras linguagens que surgem quando estudarmos autômatos. Algumas são exemplos abstratos, como:

1. A linguagem de todos os strings que consistem em n 0's seguidos por n valores 1, para algum $n \geq 0$: $\{\varepsilon, 01, 0011, 000111, \ldots\}$.

2. O conjunto de strings de 0's e 1's com um número igual de cada um deles:

$$\{\varepsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

3. O conjunto de números binários cujo valor é um número primo:

$$\{10,11,101,111,1011,...\}$$

4. Σ^* é uma linguagem para qualquer alfabeto Σ.
5. $\emptyset$, a linguagem vazia, é uma linguagem sobre qualquer alfabeto.
6. $\{\varepsilon\}$, a linguagem que consiste apenas no string vazio, também é uma linguagem sobre qualquer alfabeto. Note que $\emptyset \neq \{\varepsilon\}$: o primeiro não tem nenhum string, enquanto o outro tem um único string.

A única restrição importante sobre o que pode ser uma linguagem é que todos os alfabetos são finitos. Desse modo, embora as linguagens possam ter um número infinito de strings, são restritas a strings tirados de um alfabeto fixo e finito.

1.5.4 Problemas

Na teoria de autômatos, um *problema* é a questão de decidir se um dado string é elemento de alguma linguagem específica. Ocorre, como veremos, que tudo aquilo que chamamos em termos mais coloquiais um "problema" pode ser expresso como pertinência a uma linguagem. Mais precisamente, se Σ é um alfabeto e L é uma linguagem sobre Σ, então o problema L é:

- Dado um string w em Σ^*, definir se w está ou não em L.

Exemplo 1.26: O problema de testar o caráter primo pode ser expresso pela linguagem L_p que consiste em todos os strings binários cujo valor como um número binário é primo. Ou seja, dado um string de 0's e 1's, diremos "sim" se o string for a representação binária de um primo e diremos "não" se não for. Para alguns strings, essa decisão é fácil. Por exemplo, 0011101 pode ser a representação de um primo, pela simples razão de que todo inteiro, exceto 0, tem uma representação binária que começa com 1. Entretanto, é menos óbvio se o string 11101 pertence a L_p, e assim qualquer solução para esse problema terá de usar recursos computacionais significativos de algum tipo: tempo e/ou espaço, por exemplo. □

Um aspecto potencialmente insatisfatório de nossa definição de "problema" é que em geral os problemas não são considerados como questões de decisão (o que se segue é ou não verdadeiro?), mas como solicitações para calcular ou transformar alguma entrada (encontre a melhor maneira de realizar essa tarefa). Por exemplo, a tarefa do analisador sintático em um compilador C pode ser imaginada como um problema em nosso sentido formal, onde se recebe um string ASCII e se deve decidir se o string é ou não um elemento de L_c, o conjunto de programas válidos em C. Porém, o analisador faz mais do que decidir. Ele produz uma árvore de análise sintática, as entradas em uma tabela de símbolos e talvez mais. Pior ain-

da, o compilador como um todo resolve o problema de transformar um programa C em código-objeto para alguma máquina, o que está longe de simplesmente responder "sim" ou "não" sobre a validade de um programa.

Formadores de conjuntos como um modo de definir linguagens

É comum descrever uma linguagem usando um "formador de conjuntos":

$$\{w \mid \text{algo sobre } w\}$$

Essa expressão é lida como "o conjunto de palavras w tais que (seja o que for dito sobre w à direita da barra vertical)". Alguns exemplos são:

1. $\{w \mid w$ consiste em um número igual de 0's e 1's$\}$.
2. $\{w \mid w$ é um número inteiro binário primo$\}$.
3. $\{w \mid w$ é um programa em C sintaticamente correto $\}$.

Também é comum substituir w por alguma expressão com parâmetros e descrever os strings na linguagem declarando condições sobre os parâmetros. Aqui estão alguns exemplos; o primeiro com o parâmetro n, o segundo com os parâmetros i e j:

1. $\{0^n1^n \mid n \geq 1\}$. Lê-se "o conjunto de 0 elevado a n 1 elevado a n tal que n é maior que ou igual a 1"; essa linguagem consiste nos strings {01,0011,000111,...}. Note que, como ocorre com os alfabetos, podemos elevar um único símbolo a uma potência n para representar n cópias desse símbolo.
2. $\{0^i1^j \mid 0 \leq i \leq j\}$. Essa linguagem consiste em strings com alguns 0's (possivelmente nenhum) seguidos por um número igual ou superior de 1's.

É uma linguagem ou um problema?

Na realidade, linguagens e problemas são a mesma coisa. O termo que preferimos usar depende de nosso ponto de vista. Quando nos preocupamos apenas com strings em si, por exemplo, no conjunto $\{0^n1^n \mid n \geq 1\}$, tendemos a pensar no conjunto de strings como uma linguagem. Nos últimos capítulos deste livro, adotaremos a tendência de atribuir uma "semântica" aos strings; por exemplo, pensaremos em strings como a codificação de grafos, expressões lógicas ou mesmo inteiros. Nesses casos, em que nos importamos mais com o que o string representa do que com o próprio string, tenderemos a pensar em um conjunto de strings como um problema.

Não obstante, a definição de "problemas" como linguagens resistiu à prova do tempo como o modo apropriado de lidar com importantes questões da teoria da complexidade. Nessa teoria, estamos interessados em provar limites inferiores sobre a complexidade de certos problemas. São especialmente importantes as técnicas para provar que certos problemas não podem ser resolvidos em um período de tempo menor que o exponencial no tamanho de sua entrada. Ocorre que a versão sim/não ou baseada em linguagens de problemas conhecidos são quase tão difíceis nesse sentido quanto suas versões do tipo "resolva isso".

Ou seja, se pudermos provar que é difícil decidir se um dado string pertence à linguagem L_X de strings válidos na linguagem de programação X, então é lógico que não será mais fácil converter programas na linguagem X para código-objeto. Se fosse fácil gerar código, poderíamos executar o conversor e concluir que a entrada era um elemento válido de L_X exatamente quando o conversor tivesse sucesso em produzir o código-objeto. Tendo em vista que a etapa final de determinar se o código-objeto foi produzido não pode ser difícil, podemos usar o algoritmo rápido de geração do código-objeto para decidir de modo eficiente a pertinência a L_X. Desse modo, contradizemos a suposição de que testar a pertinência a L_X é difícil. Temos uma prova por contradição da afirmação "se o teste de pertinência a L_X é difícil, então compilar programas na linguagem de programação X é difícil".

Essa técnica, que consiste em mostrar que um problema é difícil pelo uso de um algoritmo supostamente eficiente para resolver com eficiência outro problema que já se sabe ser difícil, é chamada "redução" do segundo problema ao primeiro. Ela é uma ferramenta essencial no estudo da complexidade de problemas, e sua aplicação é muito facilitada por nossa noção de que os problemas são questões sobre pertinência a uma linguagem, e não tipos mais gerais de questões.

1.6 Resumo do Capítulo 1

- *Autômatos finitos*: Os autômatos finitos envolvem estados e transições entre estados em resposta a entradas. Eles são úteis na elaboração de vários tipos diferentes de software, inclusive o componente de análise léxica de um compilador e sistemas para verificar a correção de circuitos ou protocolos, por exemplo.
- *Expressões regulares*: Essas expressões formam uma notação estrutural para descrever os mesmos padrões que podem ser representados por autômatos finitos. Elas são usadas em muitos tipos comuns de software que incluem, por exemplo, ferramentas para pesquisa de padrões em textos ou em nomes de arquivos.

- *Gramáticas livres de contexto*: Essas gramáticas constituem uma notação importante para descrever a estrutura de linguagens de programação e conjuntos de strings inter-relacionados; elas são usados para elaborar o componente analisador sintático de um compilador.
- *Máquinas de Turing*: Essas máquinas são autômatos que modelam o poder dos computadores reais. Elas nos permitem estudar a decidibilidade, a questão do que pode ou não pode ser feito por um computador. Elas também tornam possível distinguir problemas tratáveis – aqueles que podem ser resolvidos em tempo polinomial – dos problemas intratáveis – aqueles que não o podem.
- *Provas dedutivas*: Esse método básico de prova se dá pela listagem de afirmações que são dadas como verdadeiras ou que decorrem logicamente de algumas das afirmações anteriores.
- *Prova de afirmações se-então*: Muitos teoremas têm a forma "se (algo) então (alguma outra coisa)". A afirmação ou afirmações que seguem o "se" são a hipótese, e o que segue "então" é a conclusão. As provas dedutivas de afirmações se-então começam com a hipótese e continuam com afirmações que se seguem logicamente da hipótese e de afirmações anteriores, até se provar a conclusão como uma dessas afirmações.
- *Prova de afirmações se-e-somente-Se*: Há outros teoremas da forma "(algo) se e somente se (alguma outra coisa)". Eles são provados mostrando-se afirmações se-então em ambos os sentidos. Uma espécie similar de teorema afirma a igualdade dos conjuntos descritos de dois modos diferentes; eles são provados pela demonstração de que cada um dos dois conjuntos está contido no outro.
- *Provando a contrapositiva*: Às vezes, é mais fácil provar uma afirmação da forma "se *H* então *C*" provando-se a afirmação equivalente: "se não *C* então não *H*". Essa última é chamada *contrapositiva* da primeira.
- *Prova por contradição*: Outras vezes, é mais conveniente provar a afirmação "se *H* então *C*" provando-se "se *H* e não *C* então (algo sabidamente falso)". Uma prova desse tipo é chamada prova por contradição.
- *Contra-exemplos*: Às vezes, somos solicitados a mostrar que uma certa afirmação não é verdadeira. Se a afirmação tem um ou mais parâmetros, então podemos mostrar que ela é falsa como uma generalidade, fornecendo simplesmente um contra-exemplo, isto é, uma atribuição de valores aos parâmetros que torna a afirmação falsa.
- *Provas indutivas*: Uma afirmação que tem um parâmetro inteiro *n* pode muitas vezes ser provada por indução sobre *n*. Provamos que a afirmação

é verdadeira para a base, um número finito de casos para valores específicos de n, e depois provamos a etapa indutiva: se a afirmação é verdadeira para valores até n, então ela é verdadeira para $n + 1$.

- *Induções estruturais*: Em algumas situações, inclusive muitas apresentadas neste livro, o teorema a ser provado indutivamente trata de alguma construção definida recursivamente, como árvores. Podemos provar um teorema a respeito dos objetos construídos por indução sobre o número de etapas usadas em sua construção. Esse tipo de indução é chamado estrutural.
- *Alfabetos*: Um alfabeto é qualquer conjunto finito não-vazio de símbolos.
- *Strings*: Um string é uma seqüência de comprimento finito de símbolos.
- *Linguagens e problemas*: Uma linguagem é um (possivelmente infinito) conjunto de strings, os quais escolhem seus símbolos a partir de algum alfabeto único. Quando os strings de uma linguagem têm de ser interpretados de algum modo, a questão de saber se um string pertence à linguagem às vezes se denomina um problema.

1.7 Referência para o Capítulo 1

Para ampliar a cobertura do material deste capítulo, incluindo conceitos matemáticos subjacentes à Ciência da Computação, recomendamos [1].

1. A. V. Aho e J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, Nova York, 1994.

Capítulo 2

Autômatos finitos

Este capítulo introduz a classe de linguagens conhecidas como "linguagens regulares". Essas linguagens são exatamente aquelas que podem ser descritas por autômatos finitos, que mostramos rapidamente na Seção 1.1.1. Depois de um exemplo completo que dará motivação para o estudo subseqüente, definiremos autômatos finitos de maneira formal.

Como mencionamos anteriormente, um autômato finito tem um conjunto de estados, e seu "controle" se desloca de estado para estado em resposta a "entradas" externas. Uma das distinções cruciais entre classes de autômatos finitos é o fato desse controle ser "determinístico", significando que o autômato não pode estar em mais de um estado em qualquer instante, ou "não-determinístico", significando que ele pode estar em vários estados ao mesmo tempo. Descobriremos que a adição do não-determinismo não nos permite definir qualquer linguagem que não possa ser definida por um autômato finito determinístico, mas pode haver uma eficiência substancial na descrição de uma aplicação usando-se um autômato não-determinístico. Na realidade, o não-determinismo nos permite "programar" soluções para problemas usando uma linguagem de alto nível. O autômato finito não-determinístico é então "compilado", por um algoritmo que estudaremos neste capítulo, em um autômato determinístico que pode ser "executado" em um computador convencional.

Concluímos este capítulo com um estudo de um autômato não-determinístico estendido que tem a opção adicional de fazer uma transição de um estado para outro espontaneamente, isto é, considerando o string vazio como "entrada". Esses autômatos também não aceitam nada além das linguagens regulares. Porém, veremos que eles são bastante importantes quando estudarmos expressões regulares e sua equivalência a autômatos, no Capítulo 3.

O estudo das linguagens regulares continua no Capítulo 3. Lá, apresentaremos outro formalismo importante para descrever linguagens regulares: a notação algébrica conhecida como expressões regulares. Depois de discutir as expressões regulares e de mostrar sua equivalência a autômatos finitos, utilizare-

mos autômatos e expressões regulares como ferramentas no Capítulo 4, a fim de mostrar certas propriedades importantes das linguagens regulares. São exemplos de tais propriedades as propriedades de "fechamento", que nos permitem afirmar que uma linguagem é regular porque uma ou mais linguagens diferentes são conhecidas como regulares, e também as propriedades de "decisão". Essas últimas são algoritmos para responder a perguntas sobre autômatos ou expressões regulares como, por exemplo, se dois autômatos ou expressões representam a mesma linguagem.

2.1 Uma descrição informal dos autômatos finitos

Nesta seção, estudaremos um exemplo estendido de um problema real cuja solução utiliza autômatos finitos de modo significativo. Investigaremos protocolos que admitem o "dinheiro eletrônico" – arquivos que um cliente pode usar para pagar por mercadorias na Internet, e que o vendedor pode receber com a garantia de que o "dinheiro" é real. O vendedor deve saber que o arquivo não foi forjado, nem foi copiado e enviado para o vendedor, enquanto o cliente conservava uma cópia do mesmo arquivo para gastar novamente.

A impossibilidade de forjar o arquivo é algo que deve ser garantido por um banco e por uma política de criptografia. Isto é, um terceiro participante, o banco, deve emitir e criptografar os arquivos de "dinheiro", para que a fraude não seja um problema. No entanto, o banco tem uma segunda função importante: ele deve manter um banco de dados de todo o dinheiro válido que emitiu, para poder confirmar para uma loja que o arquivo que ela recebeu representa dinheiro real e pode ser creditado na conta da loja. Não examinaremos os aspectos criptográficos do problema, nem nos preocuparemos com o modo como o banco pode armazenar e recuperar o que poderia se tornar bilhões de "cédulas eletrônicas de dinheiro". Esses problemas provavelmente não representam impedimentos a longo prazo para o conceito de dinheiro eletrônico, e exemplos de seu uso em pequena escala já existiam desde o final da década de 1990.

Porém, para se usar dinheiro eletrônico, é preciso criar protocolos que permitam a manipulação do dinheiro em uma variedade de meios desejados pelos usuários. Como os sistemas monetários sempre convidam à fraude, temos de verificar qualquer política que seja adotada com relação ao modo de usar o dinheiro. Isto é, precisamos provar que as únicas coisas que podem acontecer são aquelas que pretendemos que aconteçam – coisas que não permitam a um usuário inescrupuloso roubar de outras pessoas ou "fabricar" dinheiro. No restante desta seção, introduziremos um exemplo muito simples de um protocolo (ruim) de dinheiro eletrônico, modelaremos esse protocolo com autômatos finitos e mostraremos como construções sobre autômatos podem ser usadas para confirmar protocolos (ou, nesse caso, para descobrir que o protocolo tem um bug).

2.1.1 As regras básicas

Há três participantes: o cliente, a loja e o banco. Pressupomos, por simplicidade, que existe apenas um arquivo de "dinheiro". O cliente pode decidir transferir esse arquivo de dinheiro para a loja, que então resgatará o arquivo do banco (isto é, fará o banco emitir um novo arquivo de dinheiro pertencente à loja, e não ao cliente) e enviará as mercadorias ao cliente. Além disso, o cliente terá a opção de cancelar o arquivo. Isto é, o cliente poderá pedir ao banco para colocar o dinheiro de volta em sua conta, tornando o dinheiro indisponível. A interação entre os três participantes é portanto limitada a cinco eventos:

1. O cliente pode decidir *pagar*. Isto é, o cliente envia o dinheiro à loja.
2. O cliente pode decidir *cancelar*. O dinheiro é enviado ao banco com uma mensagem informando que o valor do dinheiro deve ser adicionado à conta bancária do cliente.
3. A loja pode *enviar* mercadorias ao cliente.
4. A loja pode *resgatar* o dinheiro. Isto é, o dinheiro é enviado ao banco com um pedido para que seu valor seja entregue à loja.
5. O banco pode *transferir* o dinheiro, criando um novo arquivo de dinheiro criptografado adequadamente e enviando esse arquivo à loja.

2.1.2 O protocolo

Os três participantes devem projetar seus comportamentos cuidadosamente, ou podem ocorrer erros. Em nosso exemplo, supomos de forma razoável que não é possível confiar que o cliente agirá com responsabilidade. Em particular, o cliente pode tentar copiar o arquivo de dinheiro, usá-lo para pagar várias vezes ou ainda pagar e cancelar o dinheiro, obtendo assim as mercadorias "de graça".

O banco deve se comportar com responsabilidade, ou não pode ser um banco. Em particular, ele tem de assegurar que duas lojas não poderão resgatar o mesmo arquivo de dinheiro, e ele não deve permitir que o dinheiro seja ao mesmo tempo cancelado e resgatado. A loja também deve ser cuidadosa. Especificamente, não deve enviar mercadorias enquanto não tiver certeza de ter recebido o dinheiro correspondente ao seu pagamento.

Protocolos desse tipo podem ser representados como autômatos finitos. Cada estado representa uma situação em que um dos participantes pode estar. Isto é, o estado "lembra" que certos eventos importantes aconteceram e que outros ainda não aconteceram. As transições entre estados acontecem quando ocorre um dos cinco eventos descritos. Vamos considerar esses eventos "externos" aos autômatos que representam os três participantes, embora cada participante seja responsável por iniciar um ou mais dos eventos. O que é importante a

respeito do problema é saber quais seqüências de eventos podem acontecer, e não quem tem permissão para iniciá-las.

A Figura 2.1 representa os três participantes por autômatos. Nesse diagrama, mostramos apenas os eventos que afetam um participante. Por exemplo, a ação *pagar* afeta somente o cliente e a loja. O banco não sabe que o dinheiro foi enviado pelo cliente à loja; ele só descobre esse fato quando a loja executa a ação *resgatar*.

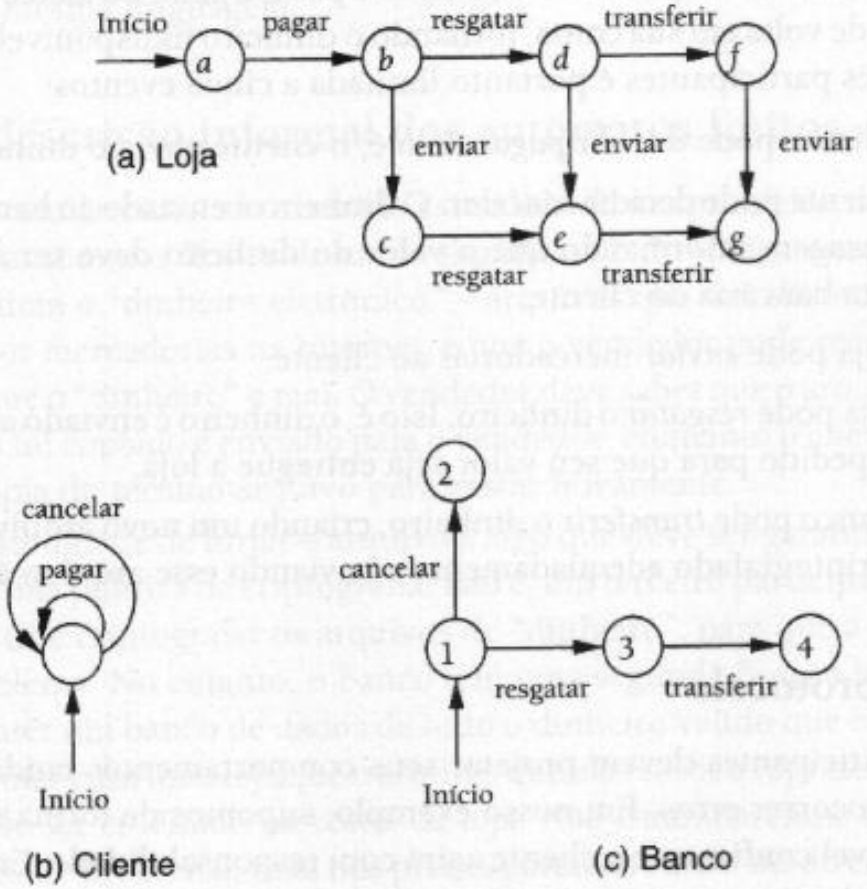

Figura 2.1: Os autômatos finitos que representam um cliente, uma loja e um banco

Vamos examinar primeiro o autômato (c) correspondente ao banco. O estado inicial é o estado 1; ele representa a situação em que o banco emitiu o arquivo de dinheiro em questão, mas ainda não recebeu a solicitação para resgatá-lo ou cancelá-lo. Se uma solicitação *cancelar* for enviada ao banco pelo cliente, então o banco irá devolver o dinheiro à conta do cliente e entrará no estado 2. Esse último estado representa a situação em que o dinheiro foi cancelado. O banco, sendo responsável, não deixará o estado 2 depois de ter entrado nele, pois o banco não deve permitir que o mesmo dinheiro seja novamente cancelado ou gasto pelo cliente.[1]

[1] Você deve se lembrar de que toda esta discussão é sobre um único arquivo de dinheiro. Na realidade, o banco estará executando o mesmo protocolo com um grande número de itens eletrônicos de dinheiro, mas o funcionamento do protocolo ainda será o mesmo para cada um deles, e assim podemos discutir o problema como se existisse apenas um item de dinheiro eletrônico.

Alternativamente, quando se encontra no estado 1 o banco pode receber uma solicitação *resgatar* da loja. Nesse caso, ele irá para o estado 3 e rapidamente enviará à loja uma mensagem *transferir*, com um novo arquivo de dinheiro que agora pertencerá à loja. Depois de enviar a mensagem de transferência, o banco irá para o estado 4. Nesse estado, ele não aceitará solicitações *cancelar* ou *resgatar*, nem executará qualquer outra ação relativa a esse arquivo de dinheiro específico.

Agora, vamos considerar a Figura 2.1(a), o autômato que representa as ações da loja. Embora o banco sempre faça tudo certo, o sistema da loja tem alguns defeitos. Imagine que as operações financeira e envio sejam feitas por processos separados, de forma que existisse a oportunidade para a ação *enviar* ser realizada antes, depois ou durante o resgate do dinheiro eletrônico. Essa política permite à loja entrar em uma situação na qual ela já enviou as mercadorias e depois descobriu que o dinheiro era falso.

A loja começa no estado *a*. Quando o cliente envia as mercadorias executando a ação *pagar*, a loja entra no estado *b*. Nesse estado, a loja inicia os processos de envio e resgate. Se as mercadorias forem enviadas primeiro, a loja entrará no estado *c*, onde ela ainda tem de resgatar o dinheiro do banco e receber a *transferência* de um arquivo de dinheiro equivalente do banco. Como outra alternativa, a loja pode enviar primeiro a mensagem *resgatar*, entrando no estado *d*. Do estado *d*, a loja pode em seguida enviar, entrando no estado *e*, ou pode receber a transferência de dinheiro do banco, entrando no estado *f*. Do estado *f*, esperamos que a loja eventualmente envie, entrando no estado *g*, em que a transação se completa e nada mais ocorre. No estado *e*, a loja está esperando pela *transferência* do banco. Infelizmente, as mercadorias já foram enviadas e, se a *transferência* nunca ocorrer, a loja estará sem sorte.

Por fim, observe o autômato correspondente ao cliente, na Figura 2.1(b). Esse autômato só tem um estado, refletindo o fato de que o cliente "pode fazer qualquer coisa". O cliente pode executar as ações *pagar* e *cancelar* qualquer número de vezes, em qualquer ordem, e permanecer em seu único estado depois de cada ação.

2.1.3 Permitindo que os autômatos ignorem ações

Enquanto os três autômatos da Figura 2.1 refletem os comportamentos dos três participantes de forma independente, estão faltando certas transições. Por exemplo, a loja não é afetada por uma mensagem *cancelar*; assim, se a ação *cancelar* for executada pelo cliente, a loja deve permanecer no estado em que se encontra. Contudo, na definição formal de um autômato finito, que estudaremos na Seção 2.2, sempre que uma entrada *X* é recebida por um autômato, o autômato deve seguir um arco identificado por *X* desde o estado em que se encontra até algum novo estado. Desse modo, o autômato correspondente à loja precisa de um arco

adicional a partir de cada estado para ele mesmo, identificado por *cancelar*. Portanto, sempre que a ação *cancelar* for executada, o autômato da loja pode fazer uma "transição" sobre essa entrada, com o efeito de permanecer no mesmo estado em que estava. Sem esses arcos adicionais, sempre que a ação *cancelar* fosse executada, o autômato da loja "morreria"; isto é, o autômato não estaria em nenhum estado, e ações adicionais executadas por esse autômato seriam impossíveis.

Outro problema potencial é que um dos participantes pode, intencional ou erroneamente, enviar uma mensagem inesperada, e não queremos que essa ação faça um dos autômatos morrer. Por exemplo, suponha que o cliente decidisse executar a ação *pagar* uma segunda vez, enquanto a loja estivesse no estado *e*. Tendo em vista que esse estado não tem nenhum arco de saída com o rótulo *pagar*, o autômato da loja morreria antes de poder receber a transferência do banco. Em resumo, devemos adicionar aos autômatos da Figura 2.1 loops em certos estados, com rótulos para todas as ações que devem ser ignoradas quando nesse estado; os autômatos completos são mostrados na Figura 2.2. Para economizar espaço, combinamos os rótulos em um único arco, em vez de mostrarmos vários arcos com os mesmos inícios e finais, mas com rótulos diferentes. Os dois tipos de ações que devem ser ignorados são:

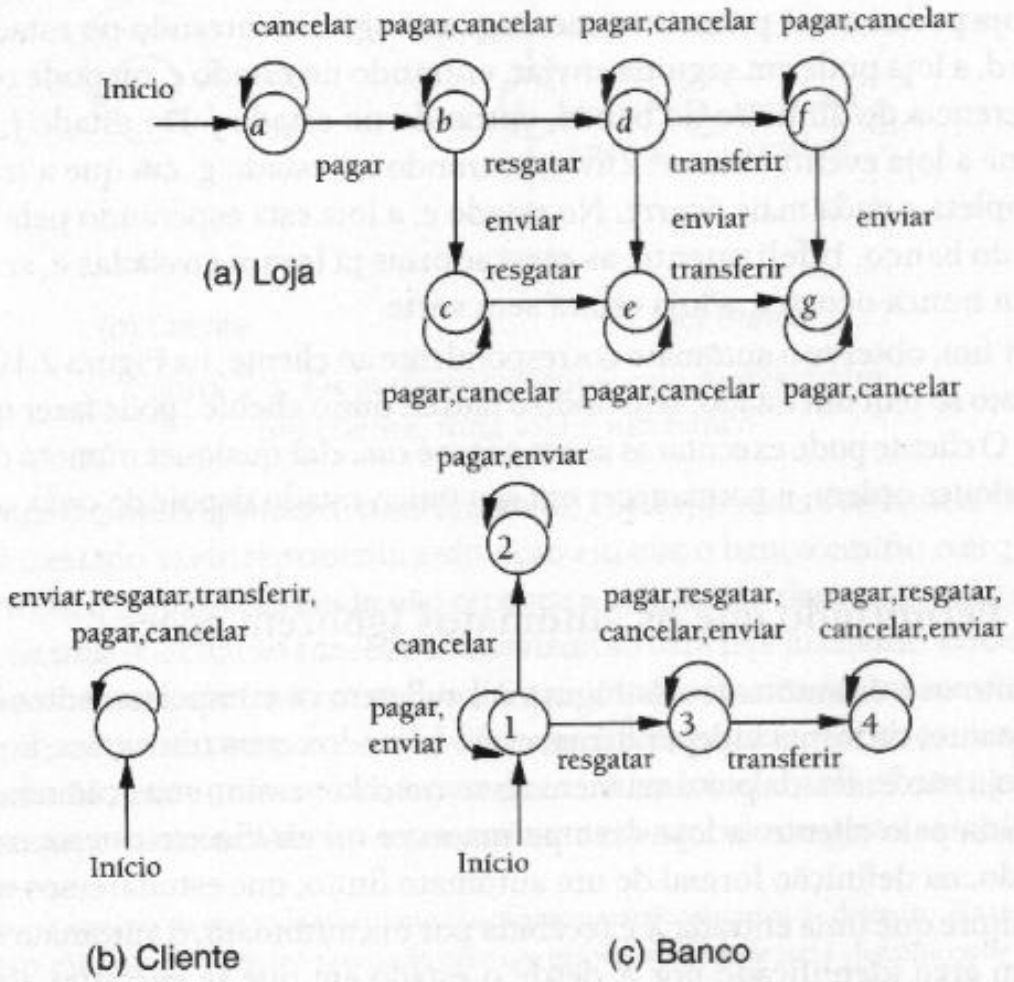

Figura 2.2: Os conjuntos completos de transições para os três autômatos

1. *Ações irrelevantes para os participantes envolvidos.* Como vimos, a única ação irrelevante para a loja é *cancelar*, e assim cada um de seus sete estados tem um loop identificado por *cancelar*. Para o banco, tanto *pagar* quanto *enviar* irrelevantes; portanto, colocamos em cada um dos estados do banco um arco identificado por *pagar*, *enviar*. Para o cliente, *enviar*, *resgatar* e *transferir* são todas irrelevantes, e então adicionamos arcos com esses rótulos. Na realidade, ele permanece em seu único estado em qualquer seqüência de entradas e, portanto, o autômato do cliente não tem nenhum efeito sobre a operação do sistema global. É claro que o cliente ainda é um participante, pois é ele que inicia as ações *pagar* e *cancelar*. No entanto, como mencionamos, a questão de quem inicia as ações não tem nenhuma relação com o comportamento dos autômatos.

2. *Ações que não devem ter permissão para eliminar um autômato.* Como mencionamos, não devemos permitir ao cliente eliminar o autômato da loja executando novamente *pagar*, e assim acrescentamos loops com o rótulo *pagar* a todos os estados com exceção de *a* (onde a ação *pagar* é esperada e relevante). Também adicionamos loops com rótulos *cancelar* aos estados 3 e 4 do banco, a fim de impedir que o cliente elimine o autômato do banco, tentando cancelar dinheiro que já foi resgatado. O banco corretamente ignora tal uma solicitação. Da mesma forma, os estados 3 e 4 têm loops em *resgatar*. A loja não deve tentar resgatar o mesmo dinheiro duas vezes mas, se o fizer, o banco irá ignorar corretamente a segunda solicitação.

2.1.4 O sistema inteiro como um autômato

Embora agora tenhamos modelos para mostrar o comportamento dos três participantes, ainda não temos uma representação para a interação dos três participantes. Como mencionamos, considerando que o cliente não tem nenhuma restrição de comportamento, esse autômato só tem um estado, e qualquer seqüência de eventos permite que ele fique nesse estado; ou seja, não é possível para o sistema como um todo "morrer", porque o autômato do cliente não tem nenhuma resposta para uma ação. Porém, tanto a loja quanto o banco se comportam de modo complexo, e não é imediatamente óbvio em que combinações de estados esses dois autômatos podem estar.

O caminho normal para explorar a interação de autômatos como esses é construir o autômato *produto*. Os estados desse autômato representam um par de estados, um da loja e um do banco. Por exemplo, o estado (3, *d*) do autômato produto representa a situação em que o banco está no estado 3, e a loja no estado *d*. Como o banco tem quatro estados e a loja tem sete, o autômato produto tem $4 \times 7 = 28$ estados.

Mostramos o autômato produto na Figura 2.3. Por clareza, organizamos os 28 estados em uma matriz. A linha corresponde ao estado do banco e a coluna ao estado da loja. Para poupar espaço, também abreviamos os rótulos nos arcos, com *P*, *S*, *C*, *R* e *T* significando pagar, enviar, cancelar, resgatar e transferir, respectivamente.

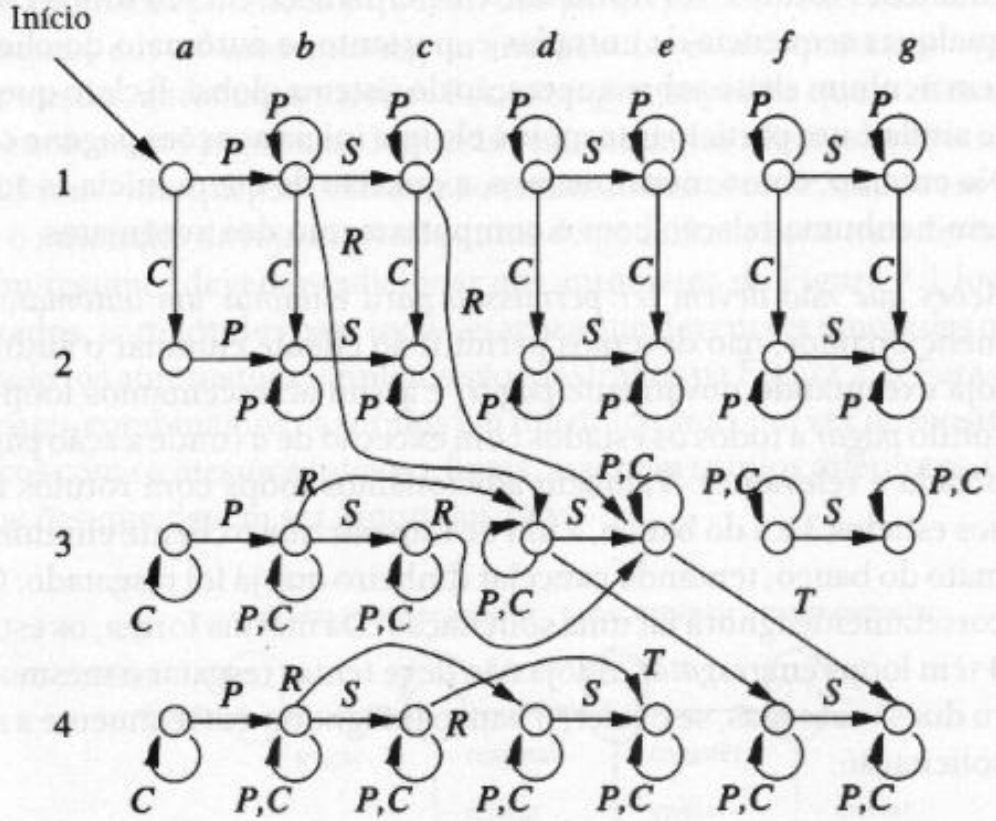

Figura 2.3: O autômato produto para a loja e para o banco

Para construir os arcos do autômato produto, precisamos executar os autômatos do banco e da loja "em paralelo". Cada um dos dois componentes do autômato produto faz independentemente transições a partir das várias entradas. No entanto, é importante notar que, se uma ação de entrada é recebida e um dos dois autômatos não tem nenhum estado para ir sobre essa entrada, então o autômato produto "morre"; ele não tem nenhum estado para onde ir.

Para tornar precisa essa regra de transições de estados, suponha que o autômato produto esteja no estado (i, x). Esse estado corresponde à situação na qual o banco se encontra no estado i e a loja no estado x. Seja Z uma das ações de entrada. Vamos examinar o autômato correspondente ao banco e ver se existe uma transição para fora do estado i com o rótulo Z. Suponha que exista, e que ela leve ao estado j (que pode ser igual a i se o banco tiver um loop para a entrada Z). Então, examinamos a loja e vemos se existe um arco identificado por Z que leve a algum estado y. Se tanto j quanto y existem, então o autômato produto tem um arco do estado (i, x) até o estado (j, y), identificado por Z. Se o estado j ou y não

existir (porque o banco ou a loja não têm nenhum arco de saída de *i* ou *x*, respectivamente, para a entrada *Z*), então não existe nenhum arco de saída de (*i*, *x*) identificado por *Z*.

Agora, podemos ver como os arcos da Figura 2.3 foram selecionados. Por exemplo, na entrada *pagar*, a loja vai do estado *a* até *b*, mas permanece no mesmo lugar se estiver em qualquer outro estado além de *a*. O banco fica em qualquer estado em que esteja quando a entrada é *pagar*, porque essa ação é irrelevante para o banco. Essa observação explica os quatro arcos identificados como *P* nas extremidades esquerda das quatro linhas da Figura 2.3, e os loops identificados por *P* em outros estados.

Para analisar um outro exemplo de como os arcos são selecionados, considere a entrada *resgatar*. Se o banco receber uma mensagem *resgatar* quando estiver no estado 1, ele irá para o estado 3. Se estiver nos estados 3 ou 4, ele ficará lá, enquanto no estado 2 o autômato do banco morre; isto é, ele não tem nenhum lugar para onde ir. Por outro lado, a loja pode fazer transições do estado *b* para *d* ou de *c* para *e* quando a entrada resgatar é recebida. Na Figura 2.3, vemos seis arcos rotulados como *resgatar*, correspondentes às seis combinações dos três estados do banco e de dois estados da loja que têm arcos de saída identificados por *R*. Por exemplo, no estado (1, *b*), o arco rotulado por *R* leva o autômato para o estado (3, *d*), pois *resgatar* leva o banco do estado 1 para 3 e a loja de *b* para *d*. Em outro exemplo, existe um arco identificado por *R* de (4, *c*) para (4, *e*), pois *resgatar* leva o banco do estado 4 de volta ao estado 4, enquanto leva a loja do estado *c* para o estado *e*.

2.1.5 Usando o autômato produto para validar o protocolo

A Figura 2.3 nos mostra alguns detalhes interessantes. Por exemplo, dos 28 estados, apenas dez deles podem ser acessados a partir do estado inicial, que é (1, *a*) – a combinação dos estados inicias dos autômatos banco e loja. Note que estados como de (2, *e*) e (4, *d*) não são *alcançáveis*, isto é, não existe nenhum caminho até eles a partir do estado inicial. Estados não *alcançáveis* não precisam ser incluídos no autômato, e só o fizemos nesse exemplo para sermos sistemáticos.

Porém, o propósito real de analisar um protocolo como esse utilizando autômatos é formular e responder perguntas tais como "é possível ocorrer o tipo de erro a seguir?" No exemplo, poderíamos perguntar se é possível a loja enviar mercadorias e nunca ser paga. Isto é, o autômato produto pode entrar em um estado no qual a loja tenha enviado (ou seja, o estado está na coluna *c*, *e* ou g) e ainda não tenha sido feita ou venha a ser feita qualquer transição sobre a entrada *T*?

Por exemplo, no estado (3, *e*), as mercadorias foram enviadas, mas haverá eventualmente uma transição sobre a entrada *T* para o estado (4, *g*). Em termos do que o banco está fazendo, uma vez que ele entrou no estado 3, ele recebeu a solicitação *resgatar* e a processou. Isso significa que ele deve ter passado pelo es-

tado 1 antes de receber a mensagem *resgatar* e, portanto, a mensagem *cancelar* não foi recebida e será ignorada se for recebida no futuro. Desse modo, o banco executará eventualmente a transferência de dinheiro para a loja.

Contudo, o estado $(2, c)$ é um problema. O estado é alcançável, mas o único arco de saída leva de volta a esse mesmo estado. Esse estado corresponde a uma situação na qual o banco recebeu uma mensagem *cancelar* antes de uma mensagem *resgatar*. Porém, a loja recebeu uma mensagem *pagar*; isto é, o cliente estava foi desonesto e gastou e cancelou o mesmo dinheiro ao mesmo tempo. A loja tolamente enviou a mercadoria antes de tentar resgatar o dinheiro e, quando ela executar a ação *resgatar*, o banco nem mesmo reconhecerá a mensagem, porque está no estado 2, onde cancelou o dinheiro e não processará uma solicitação *resgatar*.

2.2 Autômatos finitos determinísticos

Agora é hora de apresentar a noção formal de um autômato finito, de forma que vamos começar a tornar precisos alguns dos argumentos e descrições informais que vimos nas Seções 1.1.1 e 2.1. Começamos introduzindo o formalismo de um autômato finito determinístico, aquele que se encontra em um único estado depois de ler uma seqüência qualquer de entradas. O termo "determinístico" se refere ao fato de que, para cada entrada, existe um e somente um estado ao qual o autômato pode transitar a partir de seu estado atual. Em contraste, autômatos finitos "não-determinísticos", o assunto da Seção 2.3, podem estar em vários estados ao mesmo tempo. A expressão "autômato finito" irá se referir à variedade determinística, embora utilizemos "determinístico" ou a abreviatura *DFA* normalmente, para lembrar ao leitor o tipo de autômato mencionando.

2.2.1 Definição de um autômato finito determinístico

Um *autômato finito determinístico* consiste em:

1. Um conjunto finito de *estados*, freqüentemente denotado por Q.
2. Um conjunto finito de *símbolos de entrada*, freqüentemente denotado por Σ.
3. Uma *função de transição* que toma como argumentos um estado e um símbolo de entrada e retorna um estado. A função de transição será comumente denotada por δ. Em nossa representação gráfica informal de autômatos, δ foi representada pelos arcos entre estados e pelos rótulos nos arcos. Se q é um estado, e a é um símbolo de entrada, então $\delta(q, a)$ é o estado p tal que existe um arco identificado por a de q até p.[2]

[2] Mais precisamente, o grafo é uma representação de alguma função de transição δ, e os arcos do grafo são construídos para refletir as transições especificadas por δ.

4. Um *estado inicial*, um dos estados em Q.

5. Um conjunto de estados *finais* ou de *aceitação* F. O conjunto F é um subconjunto de Q.

Um autômato finito determinístico será com freqüência referido pelo seu acrônimo: *DFA*. A representação mais sucinta de um DFA é uma listagem dos cinco componentes anteriores. Em provas, mencionaremos com freqüência um DFA pela notação "de tupla de cinco elementos":

$$A = (Q, \Sigma, \delta, q_0, F)$$

onde A é o nome do DFA, Q é seu conjunto de estados, Σ é seu conjunto de símbolos de entrada, δ sua função de transição, q_0 é seu estado inicial e F é seu conjunto de estados de aceitação.

2.2.2 Como um DFA processa strings

O primeiro detalhe que precisamos entender sobre um DFA é como o DFA decide se deve ou não "aceitar" uma seqüência de símbolos de entrada. A "linguagem" do DFA é o conjunto de todos os strings que o DFA aceita. Suponha que $a_1a_2 \ldots a_n$ seja uma seqüência de símbolos de entrada. Começamos com o DFA em seu estado inicial, q_0. Consultamos a função de transição δ, digamos $\delta(q_0, a_1) = q_1$, para encontrar o estado em que o DFA A entra depois de processar o primeiro símbolo de entrada a_1. Processamos o próximo símbolo de entrada, a_2, avaliando $\delta(q_1, a_2)$; vamos supor que esse estado seja q_2. Continuamos dessa maneira, encontrando os estados $q_3, q_4, \ldots, q_n$ tais que $\delta(q_{i-1}, a_i) = q_i$ para cada i. Se q_n é um elemento de F, então a entrada $a_1a_2 \ldots a_n$ é aceita e, se não, ela é "rejeitada".

Exemplo 2.1: Vamos especificar formalmente um DFA que aceita todos e somente os strings de 0's e 1's que têm a seqüência 01 em algum lugar no string. Podemos escrever essa linguagem L como:

$$\{w \mid w \text{ é da forma } x01y \text{ para alguns strings } x \text{ e } y \text{ que consistem somente em 0's e 1's}\}$$

Outra descrição equivalente, usando parâmetros x e y à esquerda da barra vertical, é:

$$\{x01y \mid x \text{ e } y \text{ são quaisquer strings de 0's e 1's}\}$$

Os exemplos de strings na linguagem incluem 01, 11010 e 100011. Os exemplos de strings que *não* estão na linguagem incluem ε, 0 e 111000.

O que sabemos sobre um autômato que pode aceitar essa linguagem L? Primeiro, seu alfabeto de entrada é $\Sigma = \{0, 1\}$. Ele tem algum conjunto de estados, Q, dos quais um estado, digamos q_0, é o estado inicial. Esse autômato tem de memorizar os fatos importantes sobre a parte da entrada que viu até o momento. Para decidir se 01 é um substring da entrada, A precisa lembrar:

1. Ela já viu 01? Nesse caso, ele aceita toda seqüência de entradas adicionais; isto é, ele só estará em estados de aceitação de agora em diante.

2. Ele nunca viu 01, mas sua entrada mais recente foi 0; assim, se agora ele vir o 1, terá visto 01 e poderá aceitar tudo que vir daqui por diante?

3. Ele nunca viu 01, mas sua última entrada ou foi não-existente (ele apenas iniciou), ou viu um 1? Nesse caso, A não pode aceitar até ver primeiro um 0 seguido de um 1.

Cada uma dessas três condições pode ser representada por um estado. A condição (3) é representada pelo estado inicial, q_0. Certamente, quando apenas iniciamos, precisamos ver um 0 seguido de um 1. Contudo, se no estado q_0 vemos em seguida um 1, então não estamos mais próximos de ver 01, e assim devemos ficar no estado q_0. Isto é, $\delta(q_0, 1) = q_0$.

No entanto, se estamos no estado q_0 e em seguida vemos um 0, estamos na condição (2). Isto é, nunca vimos 01, mas temos nosso 0. Desse modo, vamos usar q_2 para representar a condição (2). Nossa transição de q_0 sobre a entrada 0 é $\delta(q_0, 0) = q_2$.

Agora, vamos considerar as transições a partir do estado q_2. Se vemos um 0, não estamos em situação melhor do que antes, mas ela também não é pior. Não vimos 01, mas 0 foi o último símbolo, e assim ainda estamos esperando por um 1. O estado q_2 descreve essa situação perfeitamente, e então queremos $\delta(q_2, 0) = q_2$. Se estamos no estado q_2 e vemos uma entrada1, sabemos agora que existe um 0 seguido por 1. Podemos entrar em um estado de aceitação, que chamaremos q_1, e que corresponde à condição (1) anterior. Ou seja, $\delta(q_2, 1) = q_1$.

Finalmente, devemos projetar as transições sobre o estado q_1. Nesse estado, já vimos uma seqüência 01; então, independente do que acontecer, ainda estaremos em uma situação na qual vimos 01. Isto é, $\delta(q_1, 0) = \delta(q_1, 1) = q_1$.

Dessa forma, $Q = \{q_0, q_1, q_2\}$. Como dissemos, q_0 é o estado inicial, e o único estado de aceitação é q_1; isto é, $F = \{q_1\}$. A especificação completa do autômato A que aceita a linguagem L de strings que têm como substring 01, é

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

onde δ é a função de transição descrita anteriormente. □

2.2.3 Notações mais simples para DFA's

Especificar um DFA como uma tupla de cinco elementos, com uma descrição detalhada da função de transição δ, é ao mesmo tempo tedioso e difícil de ler. Há duas notações preferenciais para descrever autômatos:

1. Um *diagrama de transições*, que é um grafo como aqueles que vimos na Seção 2.1.
2. Uma *tabela de transições*, que é uma listagem tabular da função δ que, por implicação, nos informa o conjunto de estados e o alfabeto de entrada.

Diagramas de transições

Um *diagrama de transições* para um DFA $A = (Q, \Sigma, \delta, q_0, F)$ é um grafo definido como a seguir:

a) Para cada estado em Q existe um nó correspondente.

b) Para cada estado q em Q e para cada símbolo de entrada a em Σ, seja $\delta(q, a) = p$. Então, o diagrama de transições tem um arco do nó q para o nó p, rotulado por a. Se existem vários símbolos de entrada que causam transições de q para p, então o diagrama de transições pode ter um arco rotulado pela lista desses símbolos.

c) Existe uma seta no estado inicial q_0, identificada como *Início*. Essa seta não se origina em nenhum nó.

d) Os nós correspondentes aos estados de aceitação (aqueles em F) são marcados por um círculo duplo. Estados que não estão em F têm um único círculo.

Exemplo 2.2: A Figura 2.4 mostra o diagrama de transições para o DFA que projetamos no Exemplo 2.1. Vemos nesse diagrama os três nós que correspondem aos três estados. Há uma seta *Início* entrando no estado inicial, q_0, e o único estado de aceitação, q_1, é representado por um círculo duplo. Fora de cada estado há um arco rotulado por 0 e um arco rotulado por 1 (embora os dois arcos estejam combinados em um só com um rótulo duplo no caso de q_1). Cada um dos arcos corresponde a um dos valores de δ desenvolvidos no Exemplo 2.1. □

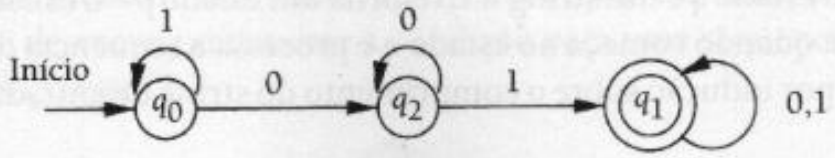

Figura 2.4: O diagrama de transições para o DFA que aceita todos os strings que contêm o substring 01

Tabelas de transições

Uma *tabela de transições* é uma representação convencional e tabular de uma função como δ que recebe dois argumentos e retorna um valor. As linhas da tabela correspondem aos estados, e as colunas correspondem às entradas. A entrada para a linha correspondente ao estado q e para a coluna correspondente à entrada a é o estado $\delta(q, a)$.

Exemplo 2.3: A tabela de transições correspondente à função δ do Exemplo 2.1 é mostrada na Figura 2.5. Também mostramos duas outras características de uma tabela de transições. O estado inicial é marcado com uma seta, e os estados de aceitação estão marcados com um asterisco. Como podemos deduzir os conjuntos de estados e os símbolos de entrada examinando os nomes das linhas e colunas, podemos ler na tabela de transições todas as informações de que precisamos para especificar o autômato finito de modo único. □

	0	1
$\rightarrow q_0$	q_2	q_0
$^*q_1$	q_1	q_1
q_2	q_2	q_1

Figura 2.5: Tabela de transições para o DFA do Exemplo 2.1

2.2.4 Estendendo a função de transição aos strings

Explicamos informalmente que o DFA define uma linguagem: o conjunto de todos os strings que resultam em uma seqüência de transições de estado, desde o estado inicial até um estado de aceitação. Em termos do diagrama de transições, a linguagem de um DFA é o conjunto de rótulos ao longo de todos os caminhos que levam do estado inicial a qualquer estado de aceitação.

Precisamos agora tornar exata a noção da linguagem de um DFA. Para isso, definimos uma *função de transição estendida* que descreve o que acontece quando começamos em qualquer estado e seguimos qualquer seqüência de entradas. Se δ é nossa função de transição, então a função de transição estendida construída a partir de δ será chamada $\hat{\delta}$. A função de transição estendida é uma função que toma um estado q e um string w e retorna um estado p – o estado que o autômato alcança quando começa no estado q e processa a seqüência de entradas w. Definimos $\hat{\delta}$ por indução sobre o comprimento do string de entrada, da seguinte forma:

BASE: $\hat{\delta}(q, \varepsilon) = q$. Isto é, se estamos no estado q e não lemos nenhuma entrada, então ainda estamos no estado q.

INDUÇÃO: Suponha que w é um string da forma xa; ou seja, a é o último símbolo de w, e x é o string que consiste em tudo, menos o último símbolo.[3] Por exemplo, $w = 1101$ é desmembrado em $x = 110$ e $a = 1$. Então:

$$\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a) \tag{2.1}$$

Agora, (2.1) pode parecer bastante complicada, mas a idéia é simples. Para calcular $\hat{\delta}(q, w)$, primeiro calculamos $\hat{\delta}(q, x)$, o estado em que o autômato se encontra depois de processar tudo, exceto o último símbolo de w. Suponha que esse estado seja p; ou seja, $\hat{\delta}(q, x) = p$. Então $\hat{\delta}(q, w)$ é o que obtemos fazendo uma transição do estado p sobre a entrada a, o último símbolo de w. Isto é, $\hat{\delta}(q, w) = \delta(p, a)$.

Exemplo 2.4: Vamos projetar um DFA para aceitar a linguagem

$$L = \{w \mid w \text{ tem ao mesmo tempo um número par de 0's é um número par de 1's}\}$$

Não deve surpreender o fato de que a função dos estados desse DFA é contar o número de 0's e o número de 1's, numa operação em módulo 2. Isto é, o estado é usado para lembrar se o número de 0's vistos até agora é par ou ímpar, e também para lembrar se o número de 1's vistos até agora é par ou ímpar. Portanto, existem quatro estados, que podem ter as seguintes interpretações:

q_0: O número de 0's vistos até agora e o número de 1's vistos até agora são ambos pares.

q_1: O número de 0's vistos até agora é par, mas o número de 1's vistos até agora é ímpar.

q_2: O número de 1's vistos até agora é par, mas o número de 0's vistos até agora é ímpar.

q_3: O número de 0's vistos até agora e o número de 1's vistos até agora são ambos ímpares.

O estado q_0 é ao mesmo tempo o estado inicial e o único estado de aceitação. Ele é o estado inicial porque, antes de ler quaisquer entradas, os números de 0's e 1's vistos até o momento são ambos zero, e zero é par. É o único estado de aceitação, porque descreve exatamente a condição para uma seqüência de 0's e 1's pertencer à linguagem L.

---

[3] Lembre-se de nossa convenção de que letras no início do alfabeto são símbolos, e que as letras que estão próximas ao final do alfabeto são strings. Precisamos dessa convenção para dar sentido à expressão "da forma xa".

Agora sabemos quase tudo o que é necessário para especificar um DFA para linguagem L. Ele é:

$$A = (\{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_0\})$$

onde a função de transição δ é descrita pelo diagrama de transições da Figura 2.6. Note como cada entrada 0 faz o estado cruzar a linha tracejada horizontal. Desse modo, depois de ver um número par de 0's, sempre estamos acima da linha, no estado q_0 ou q_1; por outro lado, depois de ver um número ímpar de 0's, sempre estamos abaixo da linha, no estado q_2 ou q_3. Da mesma forma, todo 1 faz o estado cruzar a linha tracejada vertical. Desse modo, depois de ver um número par de 1's, sempre estamos à esquerda, no estado q_0 ou q_2; por outro lado, depois de ver um número ímpar de 1's, estamos à direita, no estado q_1 ou q_3. Essas observações são uma prova informal de que os quatro estados têm as interpretações atribuídas a eles. Entretanto, seria possível provar formalmente a correção de nossas afirmações sobre os estados, por uma indução mútua, no espírito do Exemplo 1.23.

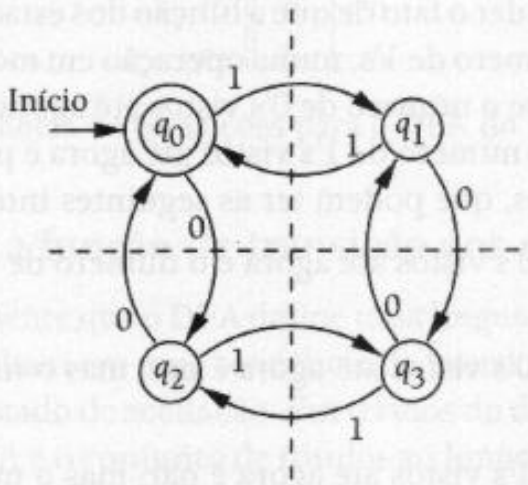

Figura 2.6: Diagrama de transições para o DFA do Exemplo 2.4

Também podemos representar esse DFA por uma tabela de transições. A Figura 2.7 mostra essa tabela. Porém, não estamos apenas preocupados com o projeto desse DFA; queremos usá-lo para ilustrar a construção de $\hat{\delta}$ a partir de sua função de transição δ. Suponha que a entrada seja 110101. Como esse string tem números pares de 0's e 1's, esperamos que ele esteja na linguagem. Desse modo, esperamos que $\hat{\delta}(q_0, 110101) = q_0$, pois q_0 é o único estado de aceitação. Vamos agora verificar essa afirmação.

	0	1
$*\rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Figura 2.7: Tabela de transições para o DFA do Exemplo 2.4

Notação padrão e variáveis locais

Depois de ler esta seção, você pode imaginar que nossa notação habitual é obrigatória; isto é, você *tem* de usar δ para a função de transição, usar A como nome de um DFA e assim por diante. Tendemos a usar as mesmas variáveis para denotar os mesmos itens em todos os exemplos, porque isso ajuda você a lembrar os tipos de variáveis, de modo semelhante a uma variável i em um programa, que quase sempre é do tipo inteiro. Porém, somos livres para chamar os componentes de um autômato, ou qualquer outro item, pelo nome que desejarmos. Desse modo, você é livre para denominar um DFA M e sua função de transição T, se preferir.

Acima de tudo, você não deve se surpreender com o fato da mesma variável ter significados diferentes em diferentes contextos. Por exemplo, os DFAs dos Exemplos 2.1 e 2.4 recebem ambos uma função de transição chamada δ. Contudo, as duas funções de transição são variáveis locais, pertencentes apenas a seus exemplos. Essas duas funções de transição são muito diferentes e não guardam nenhum relacionamento entre si.

A verificação envolve a computação de $\hat{\delta}(q_0, w)$ para cada prefixo w de 110101, começando em δ e aumentando de tamanho. O resumo desse cálculo é:

- $\hat{\delta}(q_0,\varepsilon) = q_0$.
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \varepsilon), 1) = \delta(q_0, 1) = q_1$.
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$.
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_2$.
- $\hat{\delta}(q_0,1101) = \delta(\hat{\delta}(q_0,110), 1) = \delta(q_2, 1) = q_3$.
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101),0) = \delta(q_3, 0) = q_1$.
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$.

□

2.2.5 A linguagem de um DFA

Agora, podemos definir a *linguagem* de um DFA $A = (Q, \Sigma, \delta, q_0, F)$. Essa linguagem é denotada por $L(A)$ e definida por:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \text{ está em } F\}$$

Isto é, a linguagem de A é o conjunto de strings w que levam o estado inicial q_0 até um dos estados de aceitação. Se L é $L(A)$ para algum DFA A, dizemos que L é uma *linguagem regular*.

Exemplo 2.5: Como mencionamos antes, se A é o DFA do Exemplo 2.1, então $L(A)$ é o conjunto de todos os strings de 0's e 1's que contêm um substring 01. Em vez disso, se A é o DFA do Exemplo 2.4, então $L(A)$ é o conjunto de todos os strings de 0's e 1's cujos números de símbolos 0 e 1 são ambos pares. □

2.2.6 Exercícios para a Seção 2.2

Exercício 2.2.1: Na Figura 2.8 temos um conhecido brinquedo. Uma bola de gude é solta em A ou B. As alavancas de x_1, x_2 e x_3 fazem a bolinha cair para a esquerda ou para a direita. Sempre que uma bolinha encontra uma alavanca, ela faz a alavanca se inverter após a passagem da bolinha; assim, a próxima bolinha passará pelo desvio oposto.

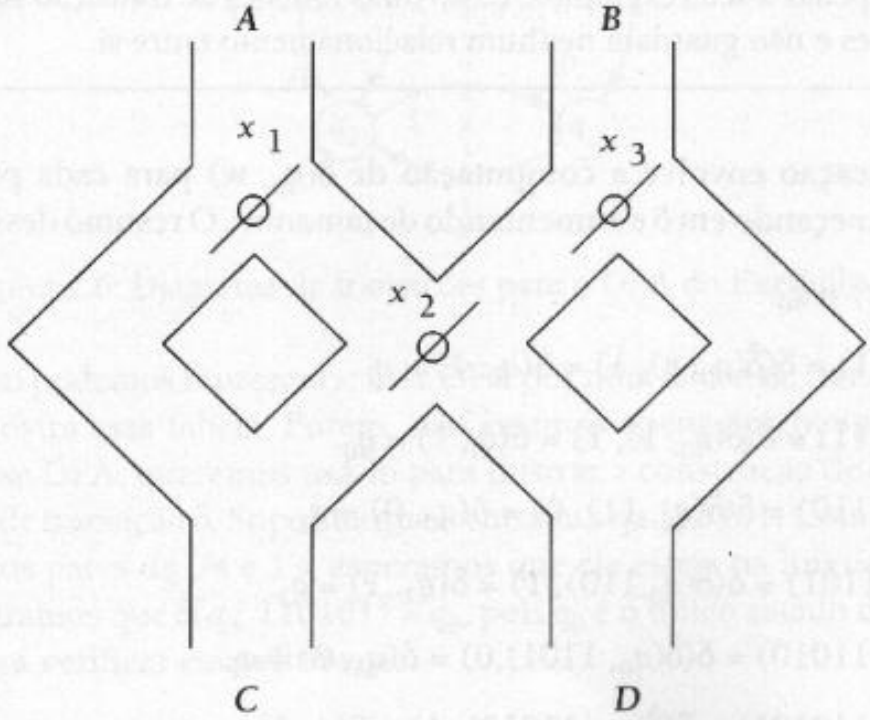

Figura 2.8: Um brinquedo de bolinhas de gude

* a) Modele esse brinquedo por um autômato finito. Sejam A e B as representações das entradas em que a bolinha pode ser solta. Seja o estado de aceitação correspondente à bolinha saindo em D; a não-aceitação representada por uma bolinha saindo em C.

! b) Descreva informalmente a linguagem deste autômato.

c) Suponha que, em vez disso, as alavancas fossem comutadas *antes* de permitir a passagem da bolinha. De que forma suas respostas para as partes (a) e (b) mudariam?

*! **Exercício 2.2.2**: Definimos $\hat{\delta}$ dividindo o string de entrada em qualquer string seguido por um único símbolo (na parte indutiva, Equação 2.1). No entanto, pensamos informalmente em $\hat{\delta}$ como a descrição do que acontece ao longo de um caminho com um certo string de rótulos e, nesse caso, não deve importar o modo como dividimos o string de entrada na definição de $\hat{\delta}$. Mostre que, de fato, $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$ para qualquer estado q e strings x e y. *Sugestão*: Execute uma indução sobre $|y|$.

! **Exercício 2.2.3**: Mostre que, para qualquer estado q, string x e símbolo de entrada a, $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$. *Sugestão*: Use o Exercício 2.2.2.

Exercício 2.2.4: Forneça os DFAs que aceitam as seguintes linguagens sobre o alfabeto $\{0,1\}$:

* a) O conjunto de todos os strings que terminam em 00.

b) O conjunto de todos os strings com três 0's consecutivos (não necessariamente no final).

c) O conjunto de strings que têm 011 como um substring.

! **Exercício 2.2.5**: Forneça os DFAs que aceitam as seguintes linguagens sobre o alfabeto $\{0,1\}$:

a) O conjunto de todos os strings tais que cada bloco de cinco símbolos consecutivos contém pelo menos dois 0's.

b) O conjunto de todos os strings cujo décimo símbolo a partir da extremidade direita é 1.

c) O conjunto de strings que começam ou terminam (ou ambos) com 01.

d) O conjunto de strings tais que o número de 0's é divisível por 5, e o número de 1's é divisível por 3.

!! **Exercício 2.2.6**: Forneça os DFAs que aceitam as seguintes linguagens sobre o alfabeto $\{0,1\}$:

* a) O conjunto de todos os strings que começam com um 1 e que, quando interpretados como inteiros binários, são múltiplos de 5. Por exemplo, strings 101, 1010 e 1111 estão na linguagem; 0, 100 e 111 não estão.

b) O conjunto de todos os strings que, quando interpretados *em ordem inversa* como inteiros binários, são divisíveis por 5. Os exemplos de strings na linguagem são 0, 10011, 1001100 e 0101.

Exercício 2.2.7: Seja A um DFA e q um estado específico de A, tal que $\delta(q, a) = q$ para todos os símbolos de entrada a. Mostre por indução sobre o comprimento da entrada que, para todos os strings de entrada w, $\hat{\delta}(q, w) = q$.

Exercício 2.2.8: Seja A um DFA e a um símbolo de entrada específico de A, tal que, para todos os estados q de A, temos $\delta(q, a) = q$.

a) Mostre por indução sobre n que, para todo $n \geq 0$, $\hat{\delta}(q, a^n) = q$, onde a^n é o string que consiste numa seqüência de n símbolos a.

b) Mostre que $\{a\}^* \subseteq L(A)$ ou $\{a\}^* \cap L(A) = \emptyset$

***! Exercício 2.2.9**: Seja $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ um DFA e suponha que, para todo a em Σ, temos $\delta(q_0, a) = \delta(q_f, a)$.

a) Mostre que, para todo $w \neq \varepsilon$, temos $\hat{\delta}(q_0, w) = \hat{\delta}(q_f, w)$.

b) Mostre que, se x é um string não vazio em $L(A)$, então, para todo $k > 0$, x^k (isto é, x escrito k vezes) também está em $L(A)$.

***! Exercício 2.2.10**: Considere o DFA com a seguinte tabela de transições:

	0	1
$\rightarrow A$	A	B
$*B$	B	A

Descreva informalmente a linguagem aceita por esse DFA e prove por indução sobre o comprimento de um string de entrada que sua descrição é correta. *Sugestão*: Ao configurar a hipótese indutiva, é sensato fazer uma declaração sobre as entradas que levam você a cada estado, não apenas sobre as entradas que o levam ao estado de aceitação.

Exercício 2.2.11: Repita o Exercício 2.2.10 para a seguinte tabela de transições:

	0	1
$\rightarrow *A$	B	A
$*B$	C	A
C	C	C

2.3 Autômatos finitos não-determinísticos

Um autômato finito "não-determinístico" (*NFA*) tem o poder de estar em vários estados ao mesmo tempo. Essa habilidade é expressa com freqüência como a capacidade de "adivinhar" algo sobre sua entrada. Por exemplo, quando o autômato é usado para procurar certas seqüências de caracteres (por exemplo, palavras-chave) em um longo string de texto, é útil para "adivinhar" que estamos no início de um desses strings e usar uma seqüência de estados apenas para verificar se o string aparece, caractere por caractere. Veremos um exemplo desse tipo de aplicação na Seção 2.4.

Antes de examinarmos aplicações, precisamos definir autômatos finitos não-determinísticos e mostrar que cada um aceita uma linguagem que também é aceita por algum DFA. Isto é, os NFAs aceitam exatamente as linguagens regulares, da mesma maneira que fazem os DFAs. No entanto, há razões para pensar nos NFAs. Freqüentemente eles são mais sucintos e mais fáceis de projetar que os DFAs. Além disso, embora sempre possamos converter um NFA em um DFA, esse último pode ter exponencialmente mais estados que o NFA; felizmente, casos desse tipo são raros.

2.3.1 Uma visão informal dos autômatos finitos não-determinísticos

Analogamente aos DFA's, um NFA tem um conjunto finito de estados, um conjunto finito de símbolos de entrada, um estado inicial e um conjunto de estados de aceitação. Ele também tem uma função de transição, que comumente chamaremos δ. A diferença entre um DFA e um NFA está no tipo de δ. Para um NFA, δ é uma função que recebe um estado e um símbolo de entrada como argumentos (da mesma forma que a função de transição do DFA), mas retorna um conjunto de zero, um ou mais estados (em vez de retornar exatamente um estado, como um DFA deve fazer). Começaremos com um exemplo de um NFA, e depois tornaremos as definições precisas.

Exemplo 2.6: A Figura 2.9 mostra um autômato finito não-determinístico, cujo trabalho é aceitar todos os strings de 0's e 1's que terminam em 01 e somente eles. O estado q_0 é o estado inicial, e podemos pensar no autômato como estando no estado q_0 (e talvez entre outros estados) sempre que ele ainda não tiver "adivinhado" que o 01 final começou. É sempre possível que o próximo símbolo não inicie o 01 final, mesmo que esse símbolo seja 0. Desse modo, o estado q_0 pode fazer uma transição para ele mesmo em 0 e em 1.

Porém, se o próximo símbolo é 0, esse NFA também adivinha que o 01 final começou. Um arco identificado como 0 leva portanto de q_0 ao estado q_1. Note

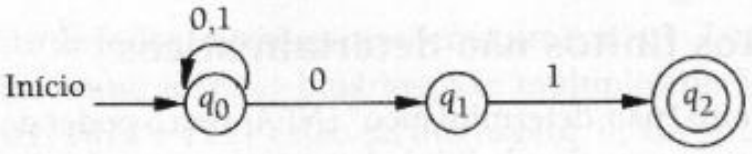

Figura 2.9: Um NFA que aceita todos os strings que terminam em 01

que existem dois arcos rotulados como 0 saindo de q_0. O NFA tem a opção de ir para q_0 ou para q_1 e, de fato, ele segue os dois caminhos, como veremos quando tornarmos as definições precisas. No estado q_1, o NFA verifica se o próximo símbolo é 1 e, nesse caso, vai para o estado q_2 e aceita a entrada.

Observe que não existe nenhum arco saindo de q_1 rotulado com 0, e não existe nenhum arco saindo de q_2. Nessas situações, o encadeamento no NFA correspondente a estes estados simplesmente "morre", embora outros encadeamentos possam continuar a existir. Enquanto um DFA tem exatamente um arco saindo de cada estado para cada símbolo de entrada, um NFA não tem a mesma restrição; vimos na Figura 2.9 casos em que o número de arcos é zero, um e dois, por exemplo.

A Figura 2.10 sugere como um NFA processa entradas. Mostramos o que acontece quando o autômato da Figura 2.9 recebe a seqüência de entrada 00101. Ele só começa em seu estado inicial q_0. Quando o primeiro 0 é lido, o NFA pode ir para o estado q_0 ou o estado q_1, e assim ele segue os dois caminhos. Esses dois encadeamentos são sugeridos pela segunda coluna na Figura 2.10.

Em seguida, o segundo 0 é lido. O estado q_0 pode novamente ir para q_0 e q_1. Porém, o estado q_1 não tem nenhuma transição em 0, e assim ele "morre". Quando a terceira entrada, um símbolo 1, ocorre, temos de considerar transições de q_0 e q_1. Descobrimos que q_0 só vai para q_0 em 1, enquanto q_1 só vai para q_2. Desse modo, depois de ler 001, o NFA se encontra nos estados q_0 e q_2. Como q_2 é um estado de aceitação, o NFA aceita 001.

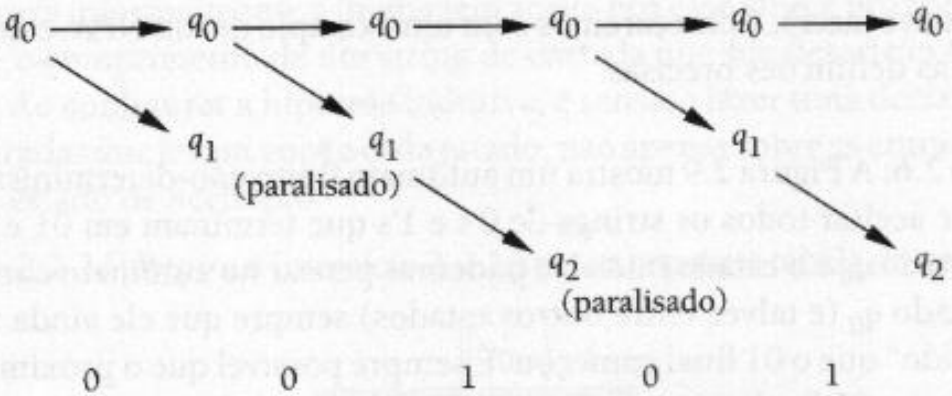

Figura 2.10: Os estados em que um NFA se encontra durante o processamento da seqüência de entrada 00101

Contudo, a entrada não está encerrada. A quarta entrada, um 0, faz o encadeamento de q_2 morrer, enquanto q_0 vai para q_0 e q_1. A última entrada, um 1, envia q_0 para q_0 e q_1 para q_2. Tendo em vista que estamos novamente em um estado de aceitação, 00101 é aceito. □

2.3.2 Definição de autômato finito não-determinístico

Agora, vamos introduzir as noções formais associadas com autômatos finitos não-determinísticos. As diferenças entre DFAs e NFAs serão indicadas à medida que prosseguirmos. Um NFA é essencialmente representado como um DFA:

$$A = (Q, \Sigma, \delta, q_0, F)$$

onde:

1. Q é um conjunto finito de *estados*.
2. Σ é um conjunto finito de *símbolos de entrada*.
3. q_0, um elemento de Q, é o *estado inicial*.
4. F, um subconjunto de Q, é o conjunto de estados *finais* (ou *de aceitação*).
5. δ, a *função de transição*, é uma função que recebe um estado em Q e um símbolo de entrada em Σ como argumentos e retorna um subconjunto de Q. Note que a única diferença entre um NFA e um DFA está no tipo de valor que δ retorna: um conjunto de estados no caso de um NFA e um único estado no caso de um DFA.

Exemplo 2.7: O NFA da Figura 2.9 pode ser formalmente especificado como

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

onde a função de transição δ é dada pela tabela de transições da Figura 2.11. □

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

Figura 2.11: A tabela de transições para um NFA que aceita todos os strings que terminam em 01

Note que as tabelas de transições podem ser usadas para especificar a função de transição para um NFA, bem como para um DFA. A única diferença é que cada entrada na tabela para o NFA é um conjunto, ainda que este conjunto seja *unitário* (tenha um único elemento). Note também que, quando não existe nenhuma transição de um dado estado sobre um dado símbolo de entrada, a entrada adequada é $\emptyset$, o conjunto vazio.

2.3.3 A função de transição estendida

Como ocorre para DFAs, precisamos estender a função de transição δ de um NFA para uma função $\hat{\delta}$ que recebe um estado q e um string de símbolos de entrada w, e retorna o conjunto de estados em que o NFA se encontra, se ele começa no estado q e processa o string w. A idéia foi sugerida pela Figura 2.10; em essência, $\hat{\delta}(q, w)$ é a coluna de estados encontrados após a leitura de w, se q é o único estado na primeira coluna. Por exemplo, a Figura 2.10 sugere que $\hat{\delta}(q_0, 001) = \{q_0, q_2\}$. Formalmente, definimos $\hat{\delta}$ para a função de transição δ de um NFA por:

BASE: $\hat{\delta}(q, \varepsilon) = \{q\}$. Isto é, sem ler quaisquer símbolos de entrada, só ficamos no estado em que começamos.

INDUÇÃO: Suponha que w seja da forma $w = xa$, onde a é o último símbolo de w e x é o restante de w. Suponha também que $\hat{\delta}(q, x) = \{p_1, p_2, ..., p_k\}$. Seja

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

Então, $\hat{\delta}(q, w) = \{r_1, r_2, ..., r_m\}$. De modo menos formal, calculamos $\hat{\delta}(q, w)$ calculando primeiro $\hat{\delta}(q, x)$, e depois seguindo todas as transições de quaisquer desses estados que seja rotulada por a.

Exemplo 2.8: Vamos usar $\hat{\delta}$ para descrever o processamento da entrada 00101 pelo NFA da Figura 2.9. Um resumo das etapas é:

1. $\hat{\delta}(q_0, \varepsilon) = \{q_0\}$.
2. $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$.
3. $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.
4. $\hat{\delta}(q_0, 001) = \delta(q_0, 0) \cup \delta(q_1, 1) = \{q_0\} \cup \emptyset = \{q_2\} = \{q_0, q_2\}$.
5. $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$.
6. $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$.

A linha (1) é a regra da base. Obtemos a linha (2) aplicando δ ao único estado, q_0, que está no conjunto anterior, e obtemos $\{q_0, q_1\}$ como resultado. A linha

(3) é obtida tomando-se a união sobre os dois estados no conjunto anterior do que obtemos quando aplicamos δ a eles com a entrada 0. Ou seja, $\delta(q_0, 0) = \{q_0, q_1\}$, enquanto $\delta(q_1, 0) = \emptyset$. Para a linha (4), tomamos a união de $\delta(q_0, 1) = \{q_0\}$ e $\delta(q_1, 1) = \{q_2\}$. As linhas (5) e (6) são semelhantes às linhas (3) e (4). □

2.3.4 A linguagem de um NFA

Como sugerimos, um NFA aceita um string w se é possível tomar qualquer seqüência de escolhas do próximo estado, enquanto são lidos os caracteres de w, e ir do estado inicial para algum estado de aceitação. O fato de outras escolhas usando os símbolos de entrada de w levarem a um estado de não aceitação ou não levarem a nenhum estado em absoluto (isto é, a seqüência de estados "morre"), não impede w de ser aceito pelo NFA como um todo. Formalmente, se $A = (Q, \Sigma, \delta, q_0, F)$ é um NFA, então

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Isto é, $L(A)$ é o conjunto de strings w em Σ^* tais que $\hat{\delta}(q_0, w)$ contém pelo menos um estado de aceitação.

Exemplo 2.9: Como um exemplo, vamos provar formalmente que o NFA da Figura 2.9 aceita a linguagem $L = \{w \mid w \text{ termina em } 01\}$. A prova é uma indução mútua das três declarações seguintes que caracterizam os três estados:

1. $\hat{\delta}(q_0, w)$ contém q_0 para todo w.
2. $\hat{\delta}(q_0, w)$ contém q_1 se e somente se w termina em 0.
3. $\hat{\delta}(q_0, w)$ contém q_2 se e somente se w termina em 01.

Para provar essas afirmações, precisamos considerar o modo como A pode atingir cada estado; isto é, qual foi o último símbolo de entrada, e em que estado estava A imediatamente antes de ler esse símbolo?

Tendo em vista que a linguagem desse autômato é o conjunto de strings w tais que $\hat{\delta}(q_0, w)$ contém q_2 (porque q_2 é o único estado de aceitação), a prova dessas três afirmações, e em particular a prova de (3), garante que a linguagem desse NFA é o conjunto de strings terminados em 01. A prova do teorema é uma indução sobre $|w|$, o comprimento de w, começando com o comprimento 0.

BASE: Se $|w| = 0$, então $w = \varepsilon$. A afirmação (1) nos diz que $\hat{\delta}(q_0, \varepsilon)$ contém q_0, o que ocorre pela parte base da definição de $\hat{\delta}$. Para a afirmação (2), sabemos que ε não termina em 0, e também sabemos que $\hat{\delta}(q_0, \varepsilon)$ não contém q_1, e novamente pela parte base da definição de $\hat{\delta}$. Desse modo, as hipóteses em ambos os senti-

dos da afirmação se-e-somente-se são falsas, e portanto ambos os sentidos da afirmação são verdadeiros. A prova da afirmação (3) para $w = \varepsilon$ é essencialmente igual à prova anterior da afirmação (2).

INDUÇÃO: Suponha que $w = xa$, onde a é um símbolo, 0 ou 1. Podemos supor que as afirmações de (1) a (3) são válidas para x e precisamos prová-las para w. Ou seja, supomos $|w| = n + 1$, e então $|x| = n$. Suporemos a hipótese indutiva para n e a demonstraremos para $n + 1$.

1. Sabemos que $\hat{\delta}(q_0, x)$ contém q_0. Como existem transições em 0 e 1 a partir de q_0 para ele mesmo, segue-se que $\hat{\delta}(q_0, w)$ também contém q_0, e assim a afirmação (1) é provada para w.
2. (Se) Suponha que w termine em 0; isto é, $a = 0$. Pela afirmação (1) aplicada a x, sabemos que $\hat{\delta}(q_0, x)$ contém q_0. Como existe uma transição de q_0 para q_1 para a entrada 0, concluímos que $\hat{\delta}(q_0, w)$ contém q_1.

   (Somente-se) Suponha que $\hat{\delta}(q_0, w)$ contém q_1. Se examinarmos o diagrama da Figura 2.9, veremos que o única maneira de entrar no estado q_1 é a seqüência de entrada w ter a forma $x0$. Isso é suficiente para provar a parte "somente-se" da afirmação (2).
3. (Se) Suponha que w termine em 01. Então se $w = xa$, sabemos que $a = 1$ e x termina em 0. Pela afirmação (2) aplicada a x, sabemos que $\hat{\delta}(q_0, x)$ contém q_1. Tendo em vista que existe uma transição de q_1 para q_2 para a entrada 1, concluímos que $\hat{\delta}(q_0, w)$ contém q_2.

   (Somente-se) Suponha que $\hat{\delta}(q_0, w)$ contém q_2. Examinando o diagrama da Figura 2.9, descobrimos que o único caminho para chegar ao estado q_2 é percorrido quando w tem a forma $x1$, onde $\hat{\delta}(q_0, x)$ contém q_1. Pela afirmação (2) aplicada a x, sabemos que x termina em 0. Portanto, w termina em 01 e provamos a afirmação (3).

□

2.3.5 Equivalência entre autômatos finitos determinísticos e não-determinísticos

Embora existam muitas linguagens para as quais um NFA é mais fácil de construir que um DFA, como a linguagem (Exemplo 2.6) de strings que terminam em 01, é um fato surpreendente que toda linguagem que pode ser descrita por algum NFA também possa ser descrita por algum DFA. Além disso, na prática um DFA tem quase tantos estados quantos tem o NFA correspondente, embora com freqüência tenha mais transições. Entretanto, no pior caso, o menor DFA pode ter 2^n estados, enquanto o menor NFA para a mesma linguagem tem apenas n estados.

A prova de que os DFAs podem fazer tudo que os NFAs podem fazer envolve uma "construção" importante, chamada *construção de subconjuntos*, porque inclui a construção de todos os subconjuntos do conjunto de estados do NFA. Em geral, muitas provas sobre autômatos envolvem a construção de um autômato a partir de outro. É importante observarmos a construção de subconjuntos como um exemplo de como se descreve formalmente um autômato em termos dos estados e transições de outro, sem conhecer os detalhes específicos desse último autômato.

A construção de subconjuntos começa a partir de um NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$. Sua meta é a descrição de um DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tal que $L(D) = L(N)$. Note que os alfabetos de entrada dos dois autômatos são os mesmos, e o estado inicial de D é o conjunto que contém apenas o estado inicial de N. Os outros componentes de D são construídos como a seguir.

- Q_D é o conjunto de subconjuntos de Q_N; isto é, Q_D é o *conjunto potência* de Q_N. Note que, se Q_N tem n estados, então Q_D terá 2^n estados. Com freqüência, nem todos esses estados estão acessíveis a partir do estado inicial de Q_D. Os estados inacessíveis podem ser "descartados"; assim, o número de estados de D pode ser efetivamente muito menor que 2^n.

- F_D é o conjunto de subconjuntos S de Q_N tais que $S \cap F_N \neq \emptyset$. Isto é, F_D representa todos os conjuntos de estados de N que incluem pelo menos um estado de aceitação de N.

- Para cada conjunto $S \subseteq Q_N$ e para cada símbolo de entrada a em Σ,

$$\delta_D(S, a) = \bigcup_{p \text{ em } S} \delta_N(p, a)$$

Isto é, para calcular $\delta_D(S,a)$, examinamos todos os estados p em S, vemos para quais estados N vai a partir de p sobre a entrada a e fazemos a união de todos esses estados.

Exemplo 2.10: Seja N o autômato da Figura 2.9 que aceita todos os strings que terminam em 01. Como o conjunto de estados de N é $\{q_0, q_1, q_2\}$, a construção de subconjuntos produz um DFA com $2^3 = 8$ estados, correspondendo a todos os subconjuntos desses três estados. A Figura 2.12 mostra a tabela de transições para esses oito estados; mostraremos em breve os detalhes de como algumas dessas entradas são calculadas.

Note que essa tabela de transições pertence a um autômato finito determinístico. Embora as entradas na tabela sejam conjuntos, os estados do DFA construído *são* conjuntos. Para tornar esse ponto mais claro, podemos criar novos nomes para esses estados; por exemplo, A para $\emptyset$, B para $\{q_0\}$ e assim por diante.

A tabela de transições do DFA da Figura 2.13 define exatamente o mesmo autômato que a Figura 2.12, mas torna claro o fato de que as entradas na tabela são estados únicos do DFA.

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

Figura 2.12: A construção completa de subconjuntos a partir da Figura 2.9

	0	1
A	A	A
$\rightarrow B$	E	B
C	A	D
$*D$	A	A
E	E	F
$*F$	E	B
$*G$	A	D
$*H$	E	F

Figura 2.13: Renomeando os estados da Figura 2.12

Dos oito estados da Figura 2.13, começando no estado inicial B, só podemos acessar os estados B, E e F. Os outros cinco estados são inacessíveis a partir do estado inicial e não precisam constar da tabela. Freqüentemente, podemos evitar a etapa de tempo exponencial da construção de entradas da tabela de transições para todo o subconjunto de estados, se executarmos uma "avaliação ociosa" nos subconjuntos, como a seguir.

BASE: Sabemos com certeza que o conjunto unitário que consiste apenas no estado inicial de N é acessível.

INDUÇÃO: Suponha que determinamos que o conjunto S de estados é acessível. Então, para cada símbolo de entrada a, calcule o conjunto de estados $\delta_D(S, a)$; sabemos que esses conjuntos de estados também serão acessíveis.

No exemplo, sabemos que $\{q_0\}$ é um estado do DFA D. Descobrimos que $\delta_D(\{q_0\}, 0) = \{q_0, q_1\}$ e $\delta_D(\{q_0\}, 1) = \{q_0\}$. Esses fatos são estabelecidos examinando-se o diagrama de transições da Figura 2.9 e observando-se que em 0 existem arcos de saída de q_0 para q_0 e q_1, enquanto em 1 existe apenas um arco para q_0. Desse modo, temos uma linha da tabela de transições para o DFA: a segunda linha na Figura 2.12.

Um dos dois conjuntos que calculamos é "antigo"; $\{q_0\}$ já foi considerado. Porém, o outro – $\{q_0, q_1\}$ – é novo e suas transições devem ser calculadas. Encontramos $\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1\}$ e $\delta_D(\{q_0, q_1\}, 1) = \{q_0, q_2\}$. Por exemplo, para ver o último cálculo, sabemos que

$$\delta_D(\{q_0,q_1\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0,q_2\}$$

Temos agora a quinta linha da Figura 2.12, e descobrimos um novo estado de D, que é $\{q_0, q_2\}$. Um cálculo semelhante nos diz que

$$\delta_D(\{q_0,q_2\}, 0) = \delta_N(q_0, 0) \cup \delta_N(q_2, 0) = \{q_0,q_1\} \cup \emptyset = \{q_0,q_1\}$$

$$\delta_D(\{q_0,q_2\}, 1) = \delta_N(q_0, 1) \cup \delta_N(q_2, 1) = \{q_0\} \cup \emptyset = \{q_0\}$$

Esses cálculos nos fornecem a sexta linha da Figura 2.12, mas ela nos dá apenas conjuntos de estados que já foram encontrados.

Desse modo, a construção de subconjuntos convergiu; conhecemos todos os estados acessíveis e suas transições. O DFA inteiro é mostrado na Figura 2.14. Observe que ele só tem três estados, o que é por coincidência exatamente o mesmo número de estados que o NFA da Figura 2.9, a partir do qual ele foi construído. Por outro lado, o DFA da Figura 2.14 tem seis transições, comparadas com as quatro transições da Figura 2.9.

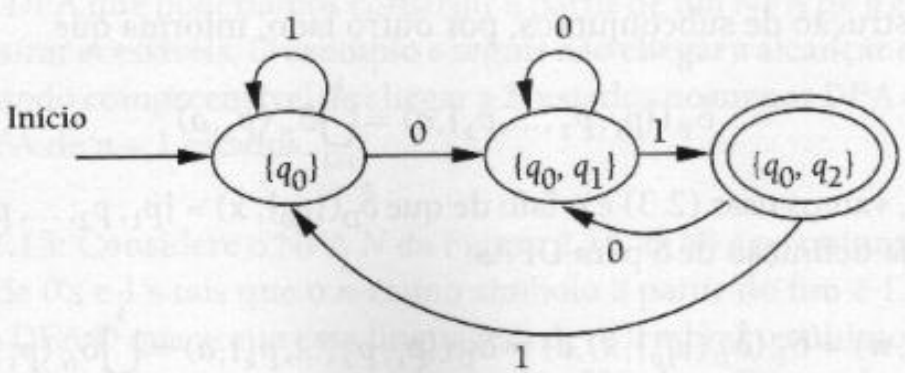

Figura 2.14: O DFA construído a partir do NFA da Figura 2.9

Precisamos mostrar formalmente que a construção de subconjuntos funciona, embora a intuição tenha sido sugerida pelos exemplos. Depois de ler a seqüência de símbolos de entrada w, o DFA construído está em um estado que é o conjunto de estados do NFA em que o NFA estaria após a leitura de w. Como os estados de aceitação do DFA são os conjuntos que incluem pelo menos um estado de aceitação do NFA, e como o NFA também aceita se entra em pelo menos um de seus estados de aceitação, podemos então concluir que o DFA e o NFA aceitam exatamente os mesmos strings, e portanto aceitem a mesma linguagem.

Teorema 2.11: Se $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é o DFA construído a partir do NFA $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ pela construção de subconjuntos, então $L(D) = L(N)$.

PROVA: O que realmente provaremos primeiro, por indução sobre $|w|$, é que

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

Note que cada uma das $\hat{\delta}$ funções retorna um conjunto de estados de Q_N, mas $\hat{\delta}_D$ interpreta esse conjunto como um dos estados de Q_D (que é o conjunto potência de Q_N), enquanto $\hat{\delta}_N$ interpreta esse conjunto como um subconjunto de Q_N.

BASE: Seja $|w| = 0$; isto é, $w = \varepsilon$. Pelas definições base de $\hat{\delta}$ para DFAs e NFAs, tanto $\hat{\delta}_D(\{q_0\}, \varepsilon)$ quanto $\hat{\delta}_N(q_0, \varepsilon)$ são iguais a $\{q_0\}$.

INDUÇÃO: Considere que w tem comprimento $n + 1$ e suponha o enunciado para o comprimento n. Considere w na forma $w = xa$, onde a é o último símbolo de w. Pela hipótese indutiva, $\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(\{q_0\}, x)$. Sejam esses dois conjuntos de estados de N indicados por $\{p_1, p_2, \ldots, p_k\}$.

A parte indutiva da definição de $\hat{\delta}$ para NFAs nos diz que

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.2}$$

A construção de subconjuntos, por outro lado, informa que

$$\hat{\delta}_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a) \tag{2.3}$$

Agora, vamos usar (2.3) e o fato de que $\hat{\delta}_D(\{q_0\}, x) = \{p_1, p_2, \ldots, p_k\}$ na parte indutiva da definição de $\hat{\delta}$ para DFAs:

$$\hat{\delta}_D(\{q_0\}, w) = \delta_D(\hat{\delta}_D(\{q_0\}, x), a) = \delta_D(\{p_1, p_2, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_1, a) \tag{2.4}$$

Desse modo, as Equações (2.2) e (2.4) demonstram que $\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$. Quando observamos que tanto D quanto N aceitam w se e somente se $\hat{\delta}_D(\{q_0\}, w)$ ou $\hat{\delta}_N(q_0, w)$, respectivamente, contêm um estado em F_N, temos uma prova completa de que $L(D) = L(N)$. □

Teorema 2.12: Uma linguagem L é aceita por algum DFA se e somente se L é aceita por algum NFA.

PROVA: (Se) A parte "se" é constituída pela construção de subconjuntos e pelo Teorema 2.11.

(Somente-se) Essa parte é fácil; temos apenas de converter um DFA em um NFA idêntico. Intuitivamente, se temos o diagrama de transições para um DFA, também podemos interpretá-lo como o diagrama de transições de um NFA, que tem exatamente uma opção de transição em cada situação. Mais formalmente, seja $D = (Q, \Sigma, \delta_D, q_0, F)$ um DFA. Defina $N = (Q, \Sigma, \delta_N, q_0, F)$ como o NFA equivalente, onde δ_N é definido pela regra:

- Se $\delta_D(q, a) = p$, então $\delta_N(q, a) = \{p\}$.

Então é fácil mostrar por indução sobre $|w|$ que, se $\hat{\delta}_D(q_0, w) = p$, então

$$\hat{\delta}_N(q_0, w) = \{p\}$$

Deixamos a prova para o leitor. Como conseqüência, w é aceito por D se e somente se é aceito por N; isto é, $L(D) = L(N)$. □

2.3.6 Um caso ruim para a construção de subconjuntos

No Exemplo 2.10, descobrimos que o DFA não tinha mais estados que o NFA. Como mencionamos, é bastante comum na prática que o DFA tenha aproximadamente o mesmo número de estados que o NFA a partir do qual é construído. Porém, o crescimento exponencial no número de estados é possível; todos os 2^n estados do DFA que poderíamos construir a partir de um NFA de n estados podem se mostrar acessíveis. O exemplo a seguir não chegar a alcançar esse limite, mas é um modo compreensível de chegar a 2^n estados no menor DFA equivalente a um NFA de $n + 1$ estados.

Exemplo 2.13: Considere o NFA N da Figura 2.15. $L(N)$ é o conjunto de todos os strings de 0's e 1's tais que o n-ésimo símbolo a partir do fim é 1. Intuitivamente, um DFA D que aceita essa linguagem deve lembrar os últimos n símbolos que leu. Tendo em vista que qualquer dos 2^n subconjuntos dos últimos n símbolos poderia ter sido 1, se D tivesse menos de 2^n estados, então haveria al-

gum estado q tal que D pudesse estar no estado q depois de ler duas seqüências diferentes de n bits, digamos $a_1a_2 \ldots a_n$ e $b_1b_2 \ldots b_n$.

Como as seqüências são diferentes, elas devem diferir em alguma posição, digamos $a_i \neq b_i$. Suponha (por simetria) que $a_i = 1$ e $b_i = 0$. Se $i = 1$, então q deve ser ao mesmo tempo um estado de aceitação e um estado de não aceitação, pois $a_1a_2 \ldots a_n$ é aceita (o n-ésimo símbolo a partir do fim é 1) e $b_1b_2 \ldots b_n$ não é. Se $i > 1$, então considere o estado p em que D entra depois de ler $i - 1$ símbolos 0. Então, p deve ser ao mesmo tempo um estado de aceitação e de não-aceitação, pois $a_ia_{i+1} \ldots a_n00 \ldots 0$ é aceita e $b_ib_{i+1} \ldots b_n00 \ldots 0$ não é.

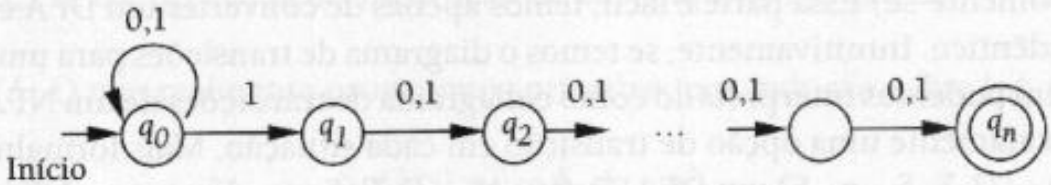

Figura 2.15: Esse NFA não tem nenhum DFA equivalente com menos de 2^n estados

Agora, vamos ver como o NFA N da Figura 2.15 funciona. Existe um estado q_0 em que o NFA sempre está, independente das entradas que foram lidas. Se a próxima entrada for 1, N também poderá "adivinhar" que esse 1 será o n-ésimo símbolo a partir do fim, e assim ele vai para o estado q_1 e também para q_0. A partir do estado q_1, qualquer entrada leva N para q_2, a próxima entrada o leva até q_3 e assim por diante até que, $n - 1$ entradas depois, ele está no estado de aceitação q_n. A afirmação formal a respeito do que estados de N fazem é:

1. N está no estado q_0 depois de ler qualquer seqüência de entradas w.
2. N está no estado q_i, para $i = 1,2,\ldots, n$, depois de ler a seqüência de entrada w se e somente se o i-ésimo símbolo a partir do fim de w é 1; isto é, w é da forma $x1a_1a_2 \ldots a_{i-1}$, onde os valores a_j correspondem a cada símbolo de entrada.

Não provaremos essas afirmações formalmente; a prova é uma indução fácil sobre $|w|$, imitando o Exemplo 2.9. Para completar a prova de que o autômato aceita exatamente os strings com um valor 1 na n-ésima posição a partir do fim, vamos considerar a afirmação (2) com $i = n$. Ela informa que N se encontra no estado q_n se e somente se o n-ésimo símbolo a partir do fim é 1. Porém, q_n é o único estado de aceitação, de forma que a condição também caracteriza exatamente o conjunto de strings aceitos por N.

O princípio da casa de pombo

No Exemplo 2.13, usamos uma importante técnica de raciocínio chamada *princípio da casa de pombo*. Coloquialmente, se você tem mais pombos que compartimentos em um pombal e cada pombo voa para algum compartimento, então deve haver pelo menos um compartimento com mais de um pombo. Em nosso exemplo, os "pombos" são as seqüências de n bits, e os "compartimentos" são os estados. Tendo em vista que existem menos estados que seqüências, um estado deve ser atribuído a duas seqüências.

O princípio da casa de pombo pode parecer óbvio, mas ele realmente depende do número de compartimentos ser finito. Desse modo, ele funciona para autômatos de estados finitos, com os estados sendo os compartimentos, mas não se aplica a outros tipos de autômatos que têm um número infinito de estados.

Para ver por que a condição finita do número de compartimentos é essencial, considere a situação infinita na qual os compartimentos correspondem a inteiros 1, 2,.... Numere os pombos como 0, 1, 2,..., de modo que exista um pombo a mais que o número de compartimentos. Porém, podemos enviar o pombo i para o compartimento $i + 1$, para todo $i \geq 0$. Então, cada pombo do número infinito de pombos terá um compartimento, e não haverá dois pombos que compartilhem um compartimento.

2.3.7 Exercícios para a Seção 2.3

* **Exercício 2.3.1**: Converta o seguinte NFA em um DFA:

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

Exercício 2.3.2: Converta o seguinte NFA em um DFA:

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

Estados mortos e DFAs que omitem algumas transições

Definimos formalmente um DFA para ter uma transição de qualquer estado, sobre qualquer símbolo de entrada, para exatamente um estado. Contudo, às vezes, é mais conveniente projetar o DFA para "morrer" em situações nas quais sabemos que é impossível qualquer extensão da seqüência de entrada ser aceita. Por exemplo, observe o autômato da Figura 1.2, que fez seu trabalho reconhecendo uma única palavra-chave, `then`, e nada mais. Tecnicamente, esse autômato não é um DFA, porque carece de transições na maioria dos símbolos de cada um de seus estados.

Porém, tal autômato é um NFA. Se usarmos a construção de subconjunto para convertê-lo em um DFA, o autômato terá quase a mesma aparência, mas incluirá um *estado morto*, isto é, um estado de não-aceitação que vai para ele mesmo a partir de cada símbolo de entrada possível. O estado morto corresponde a $\emptyset$, o conjunto vazio de estados do autômato da Figura 1.2.

Em geral, podemos adicionar um estado morto a qualquer autômato que tenha *não mais* de uma transição para qualquer estado e símbolo de entrada. Então, adicione uma transição ao estado morto a partir de cada estado q diferente dele, sobre todos os símbolos de entrada para os quais q não tem nenhuma outra transição. O resultado será um DFA no sentido estrito. Desse modo, às vezes faremos referência a um autômato como sendo um DFA, se ele tiver *no máximo* uma transição para fora de qualquer estado para qualquer símbolo, e não se ele tiver *exatamente uma* transição.

! **Exercício 2.3.3**: Converta o NFA seguinte em um DFA e descreva informalmente a linguagem que ele aceita.

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

! **Exercício 2.3.4**: Forneça autômatos finitos não-determinísticos que aceitem as linguagens a seguir. Procure tirar proveito do não-determinismo, tanto quanto possível.

* a) O conjunto de strings sobre o alfabeto $\{0, 1, \ldots, 9\}$ tal que o dígito final tenha aparecido antes.

b) O conjunto de strings sobre o alfabeto $\{0, 1, \ldots, 9\}$ tal que o dígito final *não* tenha aparecido antes.

c) O conjunto de strings de 0's e 1's tais que não existam dois 0's separados por um número de posições que seja múltiplo de 4. Observe que 0 é um múltiplo permitido de 4.

Exercício 2.3.5: Na parte somente-se do Teorema 2.12 omitimos a prova por indução sobre $|w|$ de que, se $\hat{\delta}_D(q_0, w) = p$, então $\hat{\delta}_N(q_0, w) = \{p\}$. Forneça essa prova.

! Exercício 2.3.6: No quadro sobre " Estados mortos e DFAs que omitem algumas transições", afirmamos que, se N é um NFA que tem no máximo uma opção de estado para qualquer estado e símbolo de entrada (isto é, $\delta(q, a)$ nunca tem tamanho maior que 1), então o DFA D construído a partir de N pela construção de subconjuntos tem exatamente os estados e as transições de N, mais as transições para um novo estado morto sempre que N estiver omitindo uma transição para um dado estado e símbolo de entrada. Prove essa afirmação.

Exercício 2.3.7: No Exemplo 2.13, afirmamos que o NFA N está no estado q_i, para $i = 1, 2, \ldots, n$, depois de ler a seqüência de entrada w, se e somente se o i-ésimo símbolo a partir do fim de w é 1. Prove essa afirmação.

2.4 Uma aplicação: busca em textos

Nesta seção, veremos que o estudo abstrato da seção anterior, em que consideramos o "problema" de definir se uma seqüência de bits termina em 01, é na realidade um excelente modelo para diversos problemas reais que também aparecem em aplicações, como busca na Web e extração de informações em textos.

2.4.1 Localização de strings no texto

Um problema comum na era da Web e de outros repositórios de textos on-line é apresentado a seguir. Dado um conjunto de palavras, encontre todos os documentos que contenham uma (ou todas) dessas palavras. Uma máquina de busca é um exemplo popular desse processo. O mecanismo de busca utiliza uma tecnologia específica, os chamados *índices invertidos*, em que, para cada palavra que aparece na Web (existem 100.000.000 de palavras diferentes), uma lista de todos os lugares em que essa palavra ocorre é armazenada. Máquinas com quantidades muito grandes de memória principal mantêm disponíveis as listas mais comuns, permitindo a muitas pessoas pesquisarem documentos ao mesmo tempo.

As técnicas de índices invertidos não fazem uso de autômatos finitos, mas também exigem períodos de tempo muito grandes para que robôs de busca

(crawlers) possam copiar a Web e configurar os índices. Há várias aplicações inter-relacionadas que são inadequadas para índices invertidos, mas que são boas aplicações para técnicas baseadas em autômatos. As características que tornam uma aplicação apropriada para buscas que usam autômatos são:

1. O repositório em que a pesquisa é conduzida está mudando rapidamente. Por exemplo:
   (a) A cada dia, os analistas de notícias querem pesquisar artigos de notícias on-line do dia, em busca de tópicos relevantes. Por exemplo, um analista financeiro poderia procurar certos símbolos de ações ou nomes de empresas.
   (b) Um "robô de compras" quer pesquisar os preços atuais referentes aos itens que seus clientes solicitam. O robô recuperará páginas de catálogo atualizadas na Web e então pesquisará nessas páginas palavras que sugerem um preço para um item específico.
2. Os documentos a serem pesquisados não podem ser catalogados. Por exemplo, Amazon.com não facilita a localização pelos robôs de busca de todas as páginas correspondentes a todos os livros que a empresa vende. Em vez disso, essas páginas são geradas "on the fly" em resposta a consultas. Porém, poderíamos enviar uma consulta a respeito de livros sobre um certo tópico, digamos "autômatos finitos", e então pesquisar nas páginas recuperadas, em busca de certas palavras como, por exemplo, "excelente" na parte correspondente às resenhas críticas.

2.4.2 Autômatos finitos não-determinísticos para busca em textos

Suponha que temos um conjunto de palavras, que chamaremos *palavras-chave*, e queremos encontrar ocorrências de quaisquer dessas palavras. Em aplicações como essas, um modo útil de proceder é projetar um autômato finito não-determinístico, que sinaliza, entrando em um estado de aceitação, que encontrou uma das palavras-chave. O texto de um documento é transferido, um caractere de cada vez, para esse NFA, que então reconhece ocorrências das palavras-chave nesse texto. Existe uma forma simples para um NFA que reconhece um conjunto de palavras-chave.

1. Há um estado inicial com uma transição para ele mesmo sobre todo símbolo de entrada; por exemplo, todo caractere ASCII imprimível se estamos examinando textos. Intuitivamente, o estado inicial representa um "palpite" de que ainda não começamos a ver uma das palavras-chave, ainda que tenhamos visto algumas letras de uma dessas palavras.

2. Para cada palavra-chave $a_1a_2,...,a_k$ há k estados, digamos $q_1,q_2,...,q_k$. Existe uma transição desde o estado inicial até q_1 para o símbolo a_1, uma transição de q_1 para q_2 para o símbolo a_2 e assim por diante. O estado q_k é um estado de aceitação e indica que a palavra-chave $a_1a_2 \ldots a_k$ foi encontrada.

Exemplo 2.14: Suponha que queremos projetar um NFA para reconhecer ocorrências das palavras `web` e `ebay`. O diagrama de transições para o NFA projetado com o uso das regras anteriores é apresentado na Figura 2.16. O estado 1 é o estado inicial, e usamos Σ para representar o conjunto de todos os caracteres ASCII imprimíveis. Os estados 2 a 4 têm o trabalho de reconhecer `web`, enquanto os estados 5 a 8 reconhecem `ebay`. □

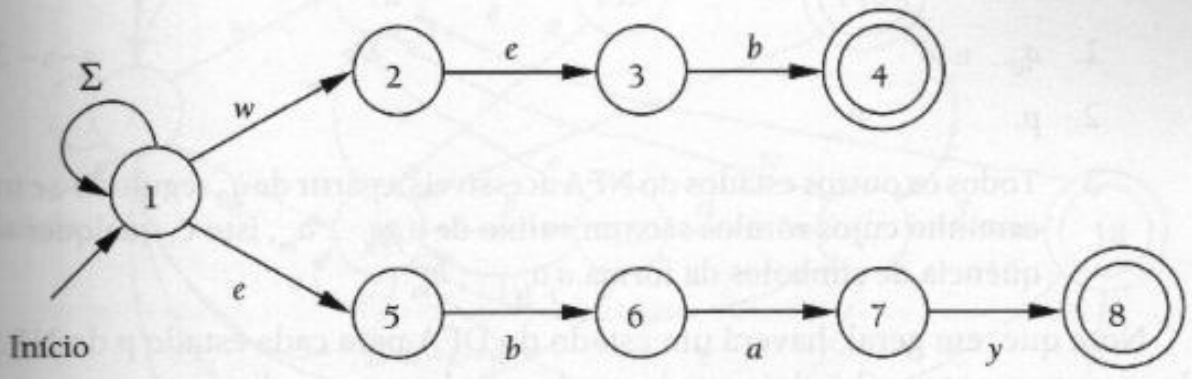

Figura 2.16: Um NFA que busca as palavras `web` e `ebay`

É claro que o NFA não é um programa. Temos duas opções importantes para uma implementação desse NFA.

1. Escrever um programa que simule esse NFA calculando o conjunto de estados em que ele está depois de ler cada símbolo de entrada. A simulação foi sugerida na Figura 2.10.

2. Converter o NFA em um DFA equivalente, usando a construção de subconjuntos. Em seguida, simular o DFA diretamente.

Alguns programas de processamento de textos, tais como formas avançados do comando `grep` do UNIX (`egrep` e `fgre`), na realidade usam uma mistura dessas duas abordagens. Porém, para nossos propósitos, a conversão em um DFA é fácil e oferece a garantia de não aumentar o número de estados.

2.4.3 Um DFA para reconhecer um conjunto de palavras-chave

Podemos aplicar a construção de subconjuntos a qualquer NFA. Entretanto, quando aplicamos essa construção a um NFA projetado a partir de um conjunto de palavras-chave, de acordo com a estratégia da Seção 2.4.2, descobrimos que o

número de estados do DFA nunca é maior que o número de estados do NFA. Tendo em vista que, no pior caso, o número de estados aumenta exponencialmente quando passamos para o DFA, essa observação é uma boa notícia, e explica por que o método de projetar um NFA para palavras-chave e depois construir um DFA a partir dele é usado com freqüência. As regras para construir o conjunto de estados do DFA são dadas a seguir.

a) Se q_0 é o estado inicial do NFA, então $\{q_0\}$ é um dos estados do DFA.

b) Suponha que p seja um dos estados do NFA, e que ele seja acessado a partir do estado inicial, ao longo de um caminho cujos símbolos são $a_1a_2 \ldots a_m$. Então, um dos estados do DFA é o conjunto de estados do NFA que consiste em:

   1. q_0.
   2. p.
   3. Todos os outros estados do NFA acessíveis a partir de q_0 seguindo-se um caminho cujos rótulos são um sufixo de $a_1a_2 \ldots a_m$, isto é, qualquer seqüência de símbolos da forma $a_ja_{j+1} \ldots a_m$.

Note que, em geral, haverá um estado do DFA para cada estado p do NFA. No entanto, na etapa (b), dois estados podem realmente produzir o mesmo conjunto de estados do NFA e, portanto, se tornar um único estado do DFA. Por exemplo, se duas palavras-chave começam com a mesma letra, digamos a, então os dois estados do NFA acessados a partir de q_0 por um arco rotulado como a produzirão o mesmo conjunto de estados do NFA e, desse modo, serão mesclados no DFA.

Exemplo 2.15: A construção de um DFA a partir do NFA da Figura 2.16 é mostrado na Figura 2.17. Cada um dos estados do DFA está localizado na mesma posição que o estado p a partir do qual ele foi derivado, com o uso da regra (b) anterior. Por exemplo, considere o estado 135, que é a nossa abreviação para $\{1,3,5\}$. Esse estado foi construído a partir do estado 3. Ele inclui o estado inicial, 1, porque todo conjunto de estados do DFA o inclui. Ele também inclui o estado 5, porque esse estado é acessado a partir do estado 1 por um sufixo, e, do string we que acessa o estado 3 na Figura 2.16.

As transições para cada um dos estados do DFA podem ser calculadas de acordo com a construção de subconjuntos. Porém, a regra é simples. A partir de qualquer conjunto de estados que inclui o estado inicial q_0 e alguns outros estados $\{p_1,p_2,\ldots,p_n\}$, determine, para cada símbolo x, onde ficam os p_i's no NFA, e faça com que esse estado do DFA tenha uma transição rotulada como x para o estado do DFA que consiste em q_0 e em todos os destinos dos p_i's para o símbolo x. Em todos os símbolos x tais que não existe nenhuma transição saindo de qual-

quer dos valores p_i rotulada por x, faça com que esse estado do DFA tenha uma transição para x de modo que o estado do DFA consistindo em q_0 e em todos os estados que são acessados a partir de q_0 no NFA sigam um arco rotulado por x.

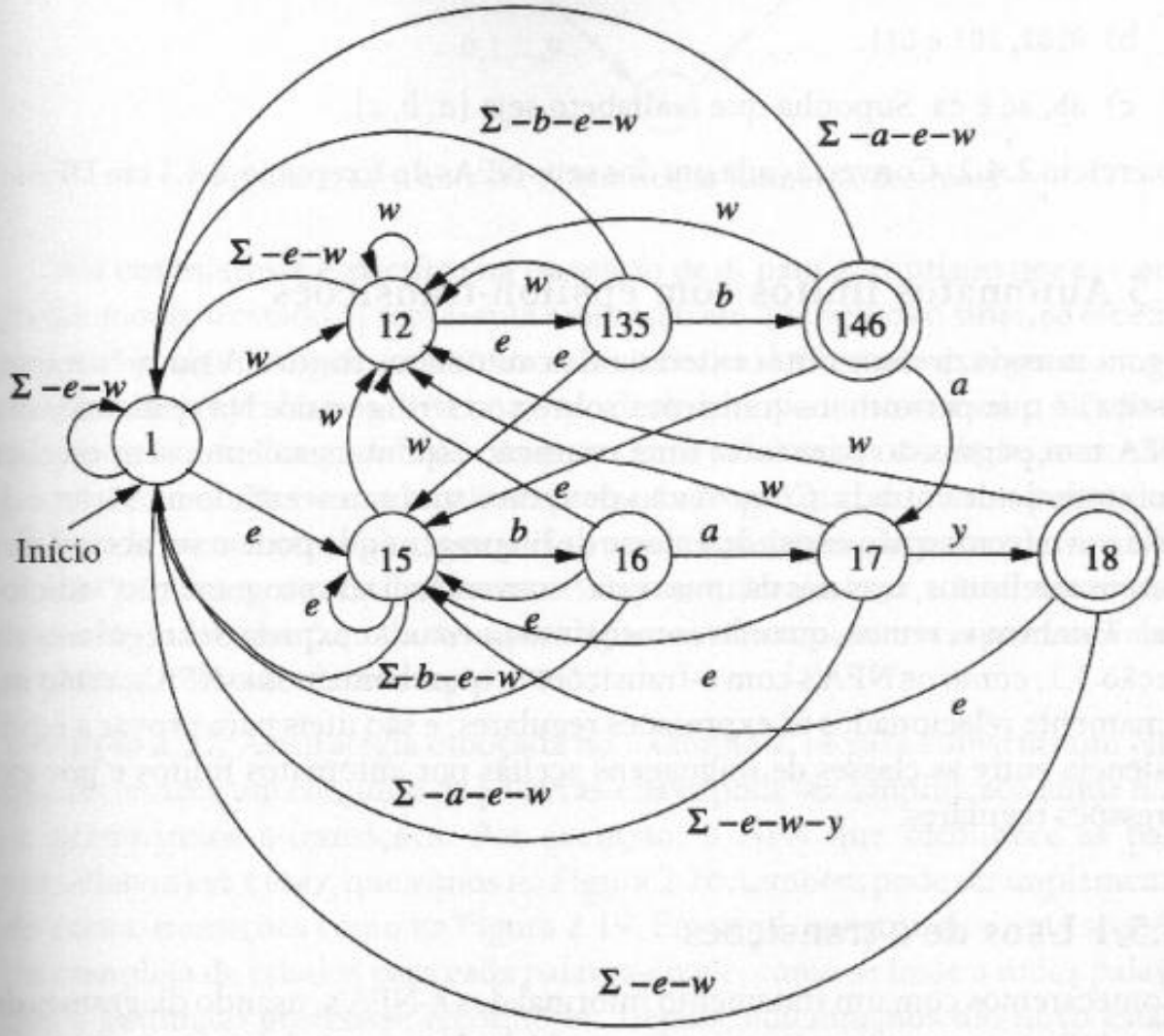

Figura 2.17: Conversão do NFA da Figura 2.16 em um DFA

Por exemplo, considere o estado 135 da Figura 2.17. O NFA da Figura 2.16 tem transições para o símbolo b dos estados 3 e 5 para os estados 4 e 6, respectivamente. Então, para o símbolo b, 135 vai para 146. Para o símbolo e, não existe nenhuma transição do NFA saindo de 3 ou 5, mas existe uma transição de 1 para 5. Desse modo, no DFA, 135 vai para 15 para a entrada e. De modo semelhante, sobre a entrada w, 135 vai para 12.

Para todos os outros símbolos x, não há nenhuma transição saindo de 3 ou 5, e o estado 1 vai apenas para ele mesmo. Desse modo, existem transições de 135 para 1 para todo símbolo em Σ diferente de b, e e w. Usamos a notação $\Sigma - b - e - w$ para representar esse conjunto, e utilizamos representações similares de outros conjuntos em que alguns símbolos são removidos de Σ. □

2.4.4 Exercícios para a Seção 2.4

Exercício 2.4.1: Projete NFAs que reconheçam os conjuntos de strings a seguir.

* a) abc, abd e aacd. Suponha que o alfabeto seja $\{a, b, c, d\}$.

b) 0101, 101 e 011.

c) ab, ac e ca. Suponha que o alfabeto seja $\{a, b, c\}$.

Exercício 2.4.2: Converta cada um dos seus NFAs do Exercício 2.4.1 em DFAs.

2.5 Autômatos finitos com epsílon-transições

Agora introduziremos outra extensão dos autômatos finitos. A nova "característica" é que permitimos transições sobre ε, o string vazio. Na realidade, um NFA tem permissão para fazer uma transição espontaneamente, sem receber um símbolo de entrada. Como o não-determinismo acrescentado na Seção 2.3, esse novo recurso não expande a classe de linguagens que podem ser aceitas por autômatos finitos, mas nos dá uma certa "conveniência de programação" adicional. Também veremos, quando começarmos a estudar expressões regulares na Seção 3.1, como os NFA's com ε-transições δ, que chamamos *ε-NFA's*, estão intimamente relacionados às expressões regulares, e são úteis para provar a equivalência entre as classes de linguagens aceitas por autômatos finitos e por expressões regulares.

2.5.1 Usos de ε-transições

Começaremos com um tratamento informal dos ε-NFA's, usando diagramas de transições que permitem ter ε como um rótulo. Nos exemplos a seguir, imagine que o autômato aceite as seqüências de rótulos ao longo dos caminhos desde o estado inicial até um estado de aceitação. Porém, cada ε encontrado ao longo de um caminho é "invisível", isto é, ele não contribui com nada para o string formado ao longo do caminho.

Exemplo 2.16: Na Figura 2.18, temos um ε -NFA que aceita números decimais consistindo em:

1. Um sinal + ou – opcional.
2. Um string de dígitos.
3. Um ponto decimal.
4. Outro string de dígitos. Esse string de dígitos ou o string (2) podem ser vazios, mas pelo menos um dos dois strings deve ser não-vazio.

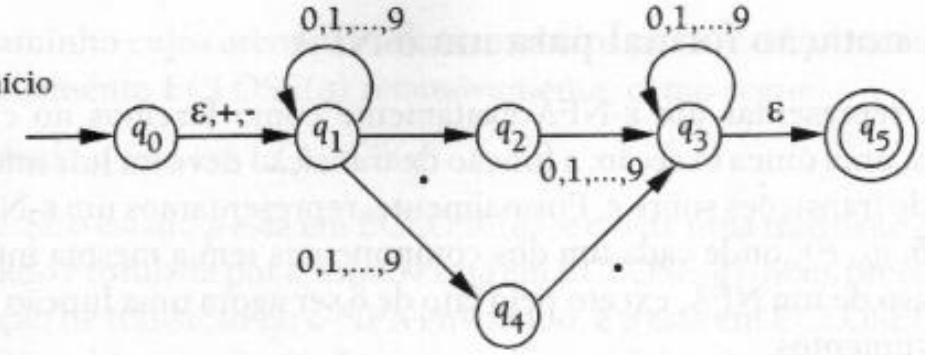

Figura 2.18: Um ε-NFA que aceita números decimais

Há um interesse específico na transição de q_0 para q_1 rotulado por ε, + ou -. Desse modo, o estado q_1 representa a situação em que vemos o sinal, se ele existe, e talvez alguns dígitos, mas não o ponto decimal. O estado q_2 representa a situação em que acabamos de ver o ponto decimal, e podemos ter visto ou não dígitos anteriores. Em q_4, definitivamente vimos pelo menos um dígito, mas não o ponto decimal. Desse modo, a interpretação de q_3 é que vimos um ponto decimal e pelo menos um dígito, esteja ele ou antes ou depois do ponto decimal. Podemos permanecer em q_3 lendo outros dígitos que existirem, e também temos a opção de "adivinhar" que o string de dígitos está completo e ir espontaneamente para q_5, o estado de aceitação. □

Exemplo 2.17: A estratégia esboçada no Exemplo 2.14 para construir um NFA que reconhece um conjunto de palavras-chave pode ser simplificada ainda mais se permitirmos ε-transições. Por exemplo, o NFA que reconhece as palavras-chaves web e ebay, que vimos na Figura 2.16, também pode ser implementado com ε-transições como na Figura 2.19. Em geral, construímos uma seqüência completa de estados para cada palavra-chave, como se fosse a única palavra que o autômato precisasse reconhecer. Depois, adicionamos um novo estado inicial (o estado 9 na Figura 2.19), com ε-transições para os estados iniciais dos autômatos correspondentes a cada uma das palavras-chave. □

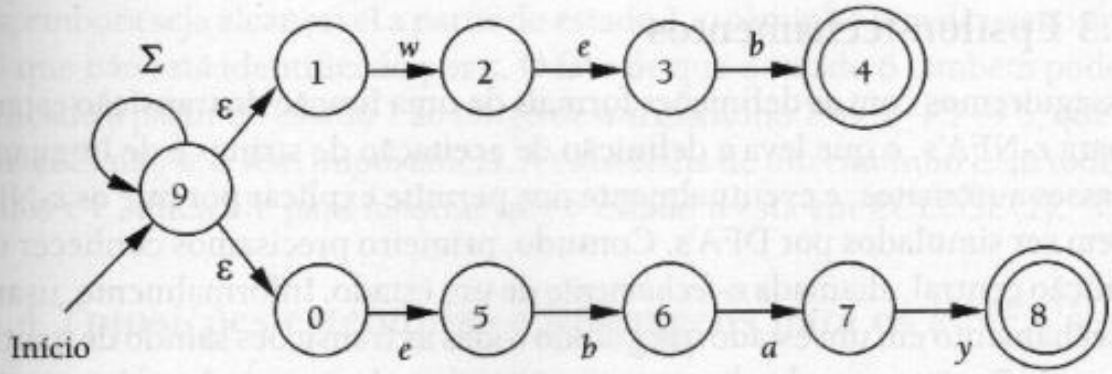

Figura 2.19: O uso de ε-transições para ajudar a reconhecer palavras-chave

2.5.2 A notação formal para um δ-NFA

Podemos representar um ε-NFA exatamente como fazemos no caso de um NFA, com uma única exceção: a função de transição deve incluir informações a respeito de transições sobre ε. Formalmente, representamos um ε-NFA A por $A = (Q, \Sigma, \delta, q_0, F)$, onde cada um dos componentes tem a mesma interpretação que no caso de um NFA, exceto pelo fato de δ ser agora uma função que recebe como argumentos:

1. Um estado em Q.
2. Um elemento de $\Sigma \cup \{\varepsilon\}$, isto é, um símbolo de entrada ou o símbolo ε. Exigimos que ε, o símbolo para o string vazio, não pode ser um elemento do alfabeto Σ, para não gerar nenhuma confusão.

Exemplo 2.18: O ε-NFA da Figura 2.18 é representado formalmente como

$$E = (\{q_0, q_1, \ldots, q_5\}, \{\cdot, +, -, 0, 1, \ldots, 9\}, \delta, q_0, \{q_5\})$$

onde δ é definido pela tabela de transições da Figura 2.20. □

	ε	+, −	.	0, 1, ..., 9
q_0	$\{q_1\}$	$\{q_1\}$	∅	∅
q_1	∅	∅	$\{q_2\}$	$\{q_1, q_4\}$
q_2	∅	∅	∅	$\{q_3\}$
q_3	$\{q_5\}$	∅	∅	$\{q_3\}$
q_4	∅	∅	$\{q_3\}$	∅
q_5	∅	∅	∅	∅

Figura 2.20: Tabela de transições correspondente à Figura 2.18

2.5.3 Epsílon-fechamentos

Prosseguiremos com as definições formais de uma função de transição estendida para ε-NFA's, o que leva à definição de aceitação de strings e de linguagens por esses autômatos, e eventualmente nos permite explicar por que os ε-NFA's podem ser simulados por DFA's. Contudo, primeiro precisamos conhecer uma definição central, chamada *ε-fechamento* de um estado. Informalmente, usamos o ε-fechamento em um estado q seguindo todas as transições saindo de q rotuladas por ε. Porém, quando chegamos a outros estados seguindo ε, acompanhamos as transições ε que saem desses estados, e assim por diante, encontrando eventualmente todo estado que pode ser alcançado a partir de q ao longo de

qualquer caminho cujos arcos são todos rotulados por ε. Formalmente, definimos o ε-fechamento ECLOSE(q) recursivamente, como segue:

BASE: O estado q está em ECLOSE(q).

INDUÇÃO: Se o estado p está em ECLOSE(q), e existe uma transição do estado p para o estado r rotulada por ε, então r está em ECLOSE(q). Mais precisamente, se δ é a função de transição do ε-NFA envolvido, e p está em ECLOSE(q), então ECLOSE(q) também contém todos os estados em $\delta(p, \varepsilon)$.

Exemplo 2.19: Para o autômato da Figura 2.18, cada estado é seu próprio ε-fechamento, com duas exceções: ECLOSE(q_0) = $\{q_0, q_1\}$ e ECLOSE(q_3) = $\{q_3, q_5\}$. A razão é que existem apenas duas ε-transições, uma que adiciona q_1 a ECLOSE(q_0) e a outra que adiciona q_5 a ECLOSE(q_3).

Um exemplo mais complexo é dado na Figura 2.21. Para essa coleção de estados, que pode fazer parte de algum δ-NFA, podemos concluir que

$$\text{ECLOSE}(1) = \{1, 2, 3, 4, 6\}$$

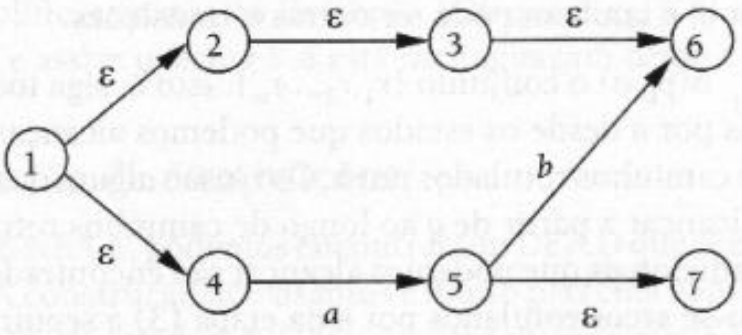

Figura 2.21: Alguns estados e transições

Cada um desses estados pode ser alcançado a partir do estado 1 ao longo de um caminho exclusivamente identificado por ε. Por exemplo, o estado 6 é alcançado pelo caminho $1 \to 2 \to 3 \to 6$. O estado 7 não está em ECLOSE(1), pois, embora seja alcançável a partir do estado 1, o caminho tem de usar o arco $4 \to 5$ que não está identificado por ε. O fato de que o estado 6 também pode ser alcançado a partir do estado 1 ao longo de um caminho $1 \to 4 \to 5 \to 6$, que tem transições não-ε, é sem importância. A existência de um caminho com todos os rótulos ε é suficiente para mostrar que o estado 6 está em ECLOSE(1). □

2.5.4 Transições estendidas e linguagens para os ε-NFA's

O ε-fechamento nos permite explicar facilmente qual será a aparência das transições de um ε-NFA quando é dada uma seqüência de entradas (não-vazia). A partir daí, podemos definir o que significa um ε-NFA aceitar sua entrada.

Suponha que $E = (Q, \Sigma, \delta, q_0, F)$ seja um ε-NFA. Primeiro, definimos $\hat{\delta}$, a função de transição estendida, a fim de refletir o que acontece em uma seqüência de entradas. O objetivo é fazer com que $\hat{\delta}(q, w)$ seja o conjunto de estados que podem ser alcançados ao longo de um caminho cujos rótulos, quando concatenados, formam o string w. Como sempre, os valores ε ao longo desse caminho não contribuem para w. A definição recursiva apropriada de $\hat{\delta}$ é:

BASE: $\hat{\delta}(q, \varepsilon) = \text{ECLOSE}(q)$. Isto é, se o rótulo do caminho é ε, então podemos seguir apenas arcos rotulados por ε que se estendem desde o estado q; isso é exatamente o que ECLOSE faz.

INDUÇÃO: Suponha que w tenha a forma xa, onde a é o último símbolo de w. Note que a é um elemento de Σ; ele não pode ser ε, que não está em Σ. Calculamos $\hat{\delta}(q, w)$ da seguinte forma:

1. Seja $\{p_1, p_2, \ldots, p_k\}$ o valor de $\hat{\delta}(q, x)$. Isto é, os p_i's são todos e somente os estados que podemos alcançar a partir de q seguindo um caminho rotulado por x. Esse caminho pode terminar com uma ou mais transições rotuladas por ε, e também pode ter outras ε-transições.
2. Seja $\bigcup_{i=1}^{k} \delta(p_i, a)$ o conjunto $\{r_1, r_2, \ldots r_m\}$. Isto é, siga todas as transições rotuladas por a desde os estados que podemos alcançar a partir de q ao longo de caminhos rotulados por x. Os r_j's são *alguns* dos estados que podemos alcançar a partir de q ao longo de caminhos rotulados por w. Os estados adicionais que podemos alcançar são encontrados a partir dos r_j, seguindo-se arcos rotulados por ε na etapa (3) a seguir.
3. Então, $\hat{\delta}(q, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$. Essa etapa adicional de fechamento inclui todos os caminhos desde q rotulados por w, considerando-se a possibilidade de existirem arcos adicionais rotulados por ε que podemos seguir após efetuar uma transição sobre o último símbolo "real", a.

Exemplo 2.20: Vamos calcular $\hat{\delta}(q_0, 5.6)$ para o ε-NFA da Figura 2.18. Um resumo das etapas necessárias é o seguinte:

- $\hat{\delta}(q_0, \varepsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$.
- Calcule $\hat{\delta}(q_0, 5)$ da seguinte forma:

1. Primeiro, calcule as transições sobre a entrada 5 a partir dos estados q_0 e q_1 que obtivemos no cálculo de $\hat{\delta}(q_0, \varepsilon)$. Isto é, calculamos $\delta(q_0, 5) \cup \delta(q_1, 5) = \{q_1, q_4\}$.
2. Em seguida, faça o ε-fechamento dos elementos do conjunto calculado na etapa (1). Obtemos $\text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$.

Esse conjunto é $\hat{\delta}(q_0, 5)$. Esse padrão de duas etapas se repete para os dois símbolos seguintes.

- Calcule $\hat{\delta}(q_0, 5.)$ como segue:
  1. Primeiro, calcule $\delta(q_1, .) \cup \{q_4, .\} = \{q_2\} \cup \{q_3\} = \{q_2, q_3\}$.
  2. Em seguida, calcule

$$\hat{\delta}(q_0, 5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}.$$

- Calcule $\hat{\delta}(q_0, 5.6)$ da seguinte forma:
  1. Primeiro, calcule $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \emptyset = \{q_3\}$.
  2. Em seguida, calcule $\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$.

□

Agora podemos definir a linguagem de um ε-NFA $E = (Q, \Sigma, \delta, q_0, F)$ da maneira esperada: $L(E) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$. Ou seja, a linguagem de E é o conjunto de strings w que levam do estado inicial a pelo menos um estado de aceitação. Para exemplificar, vimos no Exemplo 2.20 que $\hat{\delta}(q_0, 5.6)$ contém o estado de aceitação q_5, e assim o string 5.6 está na linguagem desse ε-NFA.

2.5.5 Eliminação de ε-transições

Dado qualquer ε-NFA E, podemos encontrar um DFA D que aceita a mesma linguagem que E. A construção que usamos é muito parecida com a construção de subconjuntos, pois os estados de D são subconjuntos dos estados de E. A única diferença é que devemos incorporar as ε-transições de E, o que fazemos por meio do mecanismo de ε-fechamento.

Seja $E = (Q_E, \Sigma\ \delta_E, q_0, F_E)$. Então o DFA equivalente

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

é definido como a seguir:

1. Q_D é o conjunto de subconjuntos de Q_E. Mais precisamente, descobriremos que todos os estados acessíveis de D são subconjuntos com *ε-fechamento* de Q_E; isto é, conjuntos $S \subseteq Q_E$ tais que $S = \text{ECLOSE}(S)$. Em outras palavras, os conjuntos de estados S com ε-fechamento são aqueles em que qualquer ε-transição saindo de um dos estados em S leva a um estado que também está em S. Observe que $\emptyset$ é um conjunto ε-fechado.
2. $q_D = \text{ECLOSE}(q_0)$; isto é, obtemos o estado inicial de D fechando o conjunto que consiste apenas no estado inicial de E. Observe que essa regra

difere da construção de subconjuntos original, em que o estado inicial do autômato construído era apenas o conjunto que continha o estado inicial do NFA dado.

3. F_D representa os conjuntos de estados que contêm pelo menos um estado de aceitação de E. Ou seja, $F_D = \{S \mid S \text{ está } Q_D \text{ e } S \cap F_E \neq \emptyset\}$.

4. $\delta_D(S, a)$ é calculado, para todo a em Σ e todos os conjuntos S em Q_D por:

   (a) Seja $S = \{p_1, p_2, \ldots, p_k\}$.

   (b) Calcule $\bigcup_{i=1}^{k} \delta_E(p_1, a)$; seja esse conjunto $\{r_1, r_2, \ldots, r_m\}$.

   (c) Então $\delta_D(S, a) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

Exemplo 2.21: Vamos eliminar as ε-transições do ε-NFA da Figura 2.18, que chamaremos E no texto a seguir. A partir de E, construímos um DFA D, mostrado na Figura 2.22. Entretanto, para evitar confusão, omitimos da Figura 2.22 o estado morto $\emptyset$ e todas as transições para este estado. Você deve imaginar que, para cada estado mostrado na Figura 2.22, há transições adicionais de qualquer estado para $\emptyset$ para quaisquer símbolos de entrada em que uma transição não é indicada. Além disso, o estado $\emptyset$ tem transições para ele mesmo para todos os símbolos de entrada.

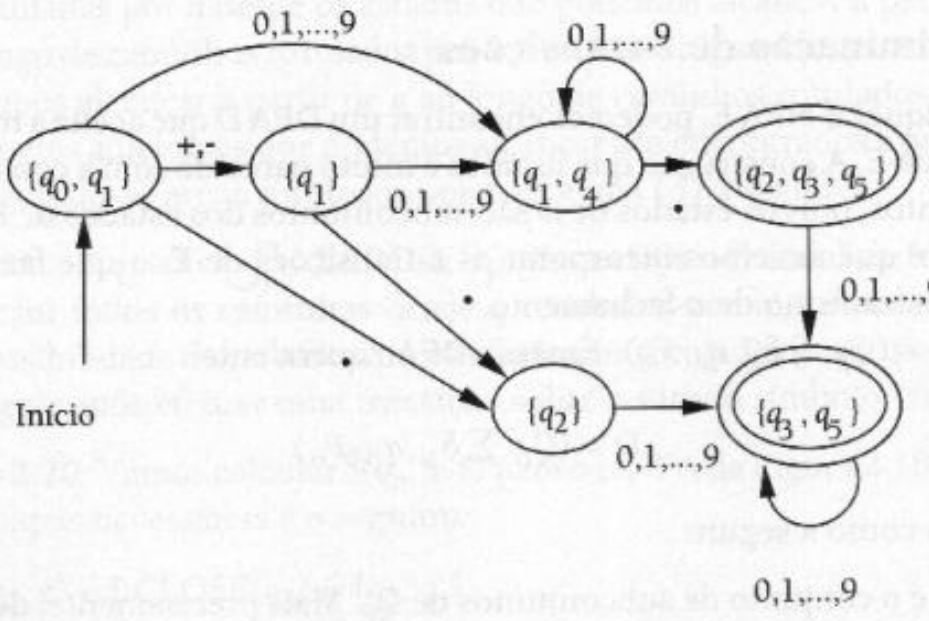

Figura 2.22: O DFA D que elimina ε-transições da Figura 2.18

Tendo em vista que o estado inicial de E é q_0, o estado inicial de D é ECLOSE(q_0), que é $\{q_0, q_1\}$. Nosso primeiro trabalho é encontrar os sucessores de q_0 e q_1 para os vários símbolos em Σ; observe que esses símbolos são os sinais de mais e menos, o ponto e os dígitos de 0 a 9. Sobre + e −, q_1 não vai para lugar nenhum na Figura 2.18, enquanto q_0 vai para q_1. Desse modo, para calcular

$\delta_D(\{q_0, q_1\}, +)$, começamos com $\{q_1\}$ e usamos o ε-fechamento . Como não existe nenhuma ε-transição saindo de q_1, temos $\delta_D(\{q_0, q_1\}, +) = \{q_1\}$. De modo semelhante, $\delta_D(\{q_0, q_1\}, -) = \{q_1\}$. Essas duas transições são indicadas por um único arco na Figura 2.22.

Em seguida, precisamos calcular $\delta_D(\{q_0, q_1\}, .)$. Tendo em vista que q_0 não vai para lugar nenhum para o ponto, e como q_1 vai para q_2 na Figura 2.18, devemos usar o ε-fechamento em $\{q_2\}$. Como não existe nenhuma ε-transição saindo de q_2, esse estado é seu próprio fechamento, e assim $\delta_D(\{q_0, q_1\}, .) = \{q_2\}$.

Por fim, devemos calcular $\delta_D(\{q_0, q_1\}, 0)$, como um exemplo das transições de $\{q_0,q_1\}$ sobre todos os dígitos. Descobrimos que q_0 não vai para nenhum lugar sobre os dígitos, mas q_1 vai para q_2 e q_4. Como nenhum desses estados tem em saída ε-transições, concluímos que $\delta_D(\{q_0, q_1\}, 0) = \{q_1, q_4\}$, e da mesma forma, para os outros dígitos.

Assim, explicamos os arcos saindo de $\{q_0,q_1\}$ da Figura 2.22. As outras transições são calculadas de modo semelhante, e vamos deixá-las para você verificar. Como q_5 é o único estado de aceitação de E, os estados de aceitação de D são aqueles estados acessíveis que contêm q_5. Vemos que os dois conjuntos $\{q_3, q_5\}$ e $\{q_2, q_3, q_5\}$ são indicados por círculos duplos na Figura 2.22. □

Teorema 2.22: Uma linguagem L é aceita por algum ε-NFA se e somente se L é aceita por algum DFA.

PROVA: (Se) Esse sentido é fácil. Suponha que $L = L(D)$ para algum DFA. Transforme D em um ε-DFA E adicionando transições $\delta(q, \varepsilon) = \emptyset$ para todos os estados q de D. Tecnicamente, também devemos converter as transições de D em símbolos de entrada, por exemplo, $\delta_D(q, a) = p$ em uma transição de NFA para o conjunto que contém apenas p, isto é, $\delta_E(q, a) = \{p\}$. Desse modo, as transições de E e D são iguais, mas E estabelece explicitamente que não existe nenhuma transição para fora de qualquer estado para ε.

(Somente se) Seja $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$ um ε-NFA. Aplique a construção de subconjuntos modificada descrita anteriormente para produzir o DFA

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

Precisamos mostrar que $L(D) = L(E)$, e o fazemos mostrando que as funções de transição estendida de E e D são iguais. Formalmente, mostramos $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_0, w)$ por indução sobre o comprimento de w.

BASE: Se $|w| = 0$, então $w = \varepsilon$. Sabemos que $\hat{\delta}_E(q_0, \varepsilon) = \text{ECLOSE}(q_0)$. Também sabemos que $q_D = \text{ECLOSE}(q_0)$, porque esse é o modo como o estado inicial de D é definido. Por fim, para um DFA, sabemos que $\hat{\delta}(p, \varepsilon) = p$ para qualquer estado p e assim, em particular, $\hat{\delta}_D(q_D, \varepsilon) = \text{ECLOSE}(q_0)$. Provamos portanto que $\hat{\delta}_E(q_D, \varepsilon) = \hat{\delta}_D(q_D, \varepsilon)$.

INDUÇÃO: Suponha que $w = xa$, onde a é o último símbolo de w, e que o enunciado seja verdadeiro para x. Isto é, $\hat{\delta}_E(q_0, x) = \hat{\delta}_D(q_D, x)$. Sejam esses dois conjuntos de estados $\{p_1, p_2, \ldots p_k\}$.

Pela definição de $\hat{\delta}$ para ε-NFAs, calculamos $\hat{\delta}_E(q_0, w)$ por:

1. Seja $\{r_1, r_2, \ldots, r_m\}$ o conjunto $\bigcup_{i=1}^{k} \delta_E(p_i, a)$.
2. Então, $\hat{\delta}_E(q_0, w) = \bigcup_{j=1}^{m} \text{ECLOSE}(r_j)$.

Se examinarmos a construção do DFA D na construção de subconjuntos modificada anterior, veremos que $\delta_D(\{p_1, p_2, \ldots, p_k\}, a)$ é construída pelas mesmas etapas (1) e (2) anteriores. Desse modo, $\hat{\delta}_D(q_D, w)$, que é $\hat{\delta}_D(\{p_1, p_2, \ldots, p_k\}, a)$, é o mesmo conjunto que $\hat{\delta}_E(q_0, w)$. Assim provamos que $\hat{\delta}_E(q_0, w) = \hat{\delta}_D(q_D, w)$ e concluímos a parte indutiva. □

2.5.6 Exercícios para a Seção 2.5

* **Exercício 2.5.1**: Considere o seguinte ε-NFA.

	ε	a	b	c
$\rightarrow p$	$\emptyset$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\emptyset$
$*r$	$\{q\}$	$\{r\}$	$\emptyset$	$\{p\}$

a) Calcule o ε-fechamento de cada estado.

b) Forneça todos os strings de comprimento três ou menos aceitos pelo autômato.

c) Converta o autômato em um DFA.

Exercício 2.5.2: Repita o Exercício 2.5.1 para o seguinte ε-NFA:

	ε	a	b	c
$\rightarrow p$	$\{q, r\}$	$\emptyset$	$\{q\}$	$\{r\}$
q	$\emptyset$	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Exercício 2.5.3: Projete ε-NFAs para as linguagens a seguir. Procure usar ε-transições para simplificar seu projeto.

a) O conjunto de strings que consiste em zero ou mais a's seguidos por zero ou mais b's, seguidos por zero ou mais c's.

! b) O conjunto de strings que consiste em 01 repetido uma ou mais vezes, ou em 010 repetido uma ou mais vezes.

! c) O conjunto de strings de 0's e 1's tais que pelo menos uma das dez últimas posições é um 1.

2.6 Resumo do Capítulo 2

- *Autômatos finitos determinísticos*: Um DFA tem um conjunto finito de estados e um conjunto finito de símbolos de entrada. Um estado é designado como estado inicial, e zero ou mais estados são estados de aceitação. Uma função de transição determina como o estado se altera toda vez que um símbolo de entrada é processado.

- *Diagramas de transições*: É conveniente representar autômatos por um grafo no qual os nós são os estados, e os arcos são rotulados por símbolos de entrada, indicando as transições desse autômato. O estado inicial é designado por uma seta, e os estados de aceitação por círculos duplos.

- *Linguagem de um autômato*: O autômato aceita strings. Um string é aceito se, começando no estado inicial, as transições causadas pelo processamento dos símbolos desse string um de cada vez levam a um estado de aceitação. Em termos do diagrama de transições, um string é aceito se for o rótulo de um caminho do estado inicial até algum estado de aceitação.

- *Autômatos finitos não-determinísticos*: O NFA difere do DFA pelo fato de que o NFA pode ter qualquer número de transições (inclusive zero) para os estados seguintes, a partir de um dado estado ou de um dado símbolo de entrada.

- *A construção de subconjuntos*: Tratando conjuntos de estados de um NFA como estados de um DFA, é possível converter qualquer NFA em um DFA que aceite a mesma linguagem.

- *ε-transições* : Podemos estender o NFA permitindo transições para uma entrada vazia, isto é, sem qualquer símbolo de entrada. Esses NFA's estendidos podem ser convertidos em DFA's que aceitam a mesma linguagem.

- *Aplicações de busca em textos*: Os autômatos finitos não-determinísticos constituem uma forma útil de representar um comparador de padrões (*pattern-matcher*) que examina um grande corpo de texto em busca de uma ou mais palavras-chave. Esses autômatos são simulados diretamente em software ou são convertidos primeiro em um DFA, que é então simulado.

Capítulo 3

Expressões regulares e linguagens

Começamos esse capítulo introduzindo a notação chamada "expressões regulares". Essas expressões são outro tipo de notação para a definição de linguagens, que mostramos brevemente na Seção 1.1.2. As expressões regulares também podem ser consideradas uma "linguagem de programação", na qual expressamos algumas aplicações importantes, como aplicações de pesquisa em textos ou componentes de compiladores. As expressões regulares estão intimamente relacionadas aos autômatos finitos não-determinísticos e podem ser consideradas uma alternativa "amigável para o usuário" para a notação dos NFA, a fim de descrever componentes de software.

Neste capítulo, depois de definir expressões regulares, mostraremos que elas são capazes de definir todas e somente as linguagens regulares. Discutiremos o modo como as expressões regulares são usadas em vários sistemas de software. Em seguida, examinaremos as leis algébricas que se aplicam a expressões regulares. Elas têm semelhança significativa com as leis algébricas da aritmética, ainda que também existam algumas diferenças importantes entre as álgebras de expressões regulares e de expressões aritméticas.

3.1 Expressões regulares

Neste ponto, deslocamos nossa atenção das descrições de linguagens semelhantes a máquinas – autômatos finitos determinísticos e não-determinísticos – para uma descrição algébrica: a "expressão regular". Descobriremos que as expressões regulares podem definir exatamente as mesmas linguagens que as diversas formas de autômatos descrevem: as linguagens regulares. Porém, as expressões regulares oferecem algo que os autômatos não oferecem: um modo declarativo de expressar os strings que queremos aceitar. Dessa maneira, as expressões regulares servem como a linguagem de entrada para muitos sistemas que processam strings. Os exemplos incluem:

1. Comandos de pesquisa como o comando `grep` do UNIX ou comandos equivalentes para localizar strings que são vistos em navegadores da Web ou sistemas de formatação de textos. Esses sistemas usam uma notação semelhante à de expressões regulares para descrever padrões que o usuário quer encontrar em um arquivo. Diferentes sistemas de pesquisa convertem a expressão regular em um DFA ou NFA e simulam esse autômato no arquivo que está sendo pesquisado.
2. Geradores de analisadores léxicos, como Lex ou Flex. Lembre-se de que um analisador léxico é o componente de um compilador que divide o programa-fonte em unidades lógicas (chamadas *tokens*) de um ou mais caracteres que têm um significado compartilhado. Os exemplos de símbolos incluem palavras-chave (por exemplo, `while`), identificadores (por exemplo, qualquer letra seguida por zero ou mais letras e/ou dígitos) e sinais, como + ou <=. Um gerador de analisador léxico aceita descrições das formas de tokens que são em essência expressões regulares, e produz um DFA que reconhece o token que aparece em seguida na entrada.

3.1.1 Os operadores de expressões regulares

As expressões regulares denotam linguagens. Como um exemplo simples, a expressão regular **01* + 10*** denota a linguagem que consiste em todos os strings que são um único 0 seguido por qualquer número de 1's ou um único 1 seguido por qualquer número de 0's. Não esperamos que você saiba neste momento como interpretar expressões regulares, e assim nossa afirmação sobre a linguagem dessa expressão deve ser aceita em confiança, por enquanto. Em breve definiremos todos os símbolos usados nessa expressão, e assim você poderá ver por que nossa interpretação dessa expressão regular é a correta. Antes de descrever a notação de expressões regulares, precisamos conhecer as três operações sobre linguagens que os operadores de expressões regulares representam. Essas operações são:

1. A *união* de duas linguagens L e M, denotadas por $L \cup M$, é o conjunto de strings que estão em L ou M, ou em ambas. Por exemplo, se $L = \{001, 10, 111\}$ e $M = \{\varepsilon, 001\}$, então $L \cup M = \{\varepsilon, 10, 001, 111\}$.
2. A *concatenação* de linguagens L e M é o conjunto de strings que podem ser formados tomando-se qualquer string em L e concatenando-se esse string com qualquer string em M. Vimos na Seção 1.5.2 a definição de concatenação de um par de strings; um string é seguido pelo outro para formar o resultado da concatenação. Denotamos a concatenação de linguagens por um ponto ou sem nenhum operador, embora o operador de concatenação seja chamado com freqüência "ponto". Por exemplo, se

$L = \{001, 10, 111\}$ e $M = \{\varepsilon, 001\}$, então $L.M$, ou simplesmente LM, é $\{001, 10, 111, 001001, 10001, 111001\}$. Os três primeiros strings em LM são os strings de L concatenados com ε. Tendo em vista que ε é a identidade para a concatenação, os strings resultantes são iguais aos strings de L. No entanto, os três últimos strings em LM são formados tomando-se cada string em L e concatenando-se esse string com o segundo string em M, que é 001. Por exemplo, 10 de L concatenado com 001 de M nos dá 10001 para LM.

3. O *fechamento* (ou *estrela*, ou *fechamento de Kleene*)[1] de uma linguagem L é denotado L^* e representa o conjunto dos strings que podem ser formados tomando-se qualquer número de strings de L, possivelmente com repetições (isto é, o mesmo string pode ser selecionado mais de uma vez) e concatenando-se todos eles. Por exemplo, se $L = \{0, 1\}$, então L^* representa todos os strings de 0's e 1's. Se $L = \{0, 11\}$, então L^* consiste nos strings de 0's e 1's tais que os símbolos 1 formam pares; por exemplo, 011, 11110 e ε, mas não 01011 ou 101. Mais formalmente, L^* é a união infinita $U_{i \geq 0} L^i$, onde $L^0 = \{\varepsilon\}$, $L^1 = L$ e L^i para $i > 1$ é $LL \ldots L$ (a concatenação de i cópias de L).

Exemplo 3.1: Tendo em vista que a idéia do fechamento de uma linguagem é um pouco complicada, vamos estudar alguns exemplos. Primeiro, seja $L = \{0, 11\}$. $L^0 = \{\varepsilon\}$, independente de qual linguagem é L; a 0-ésima potência representa a seleção de zero strings de L. $L^1 = L$, que representa a escolha de um string de L. Desse modo, os dois primeiros termos na expansão de L^* nos dão $\{\varepsilon, 0, 11\}$.

Em seguida, considere L^2. Escolhemos dois strings de L, com repetições permitidas, e assim há quatro opções. Essas quatro seleções nos dão a $L^2 = \{00, 011, 110, 1111)$. De modo semelhante, L^3 é o conjunto de strings que podem ser formados fazendo-se três escolhas dos dois strings em L e nos dá

$$\{000, 0011, 0110, 1100, 01111, 11011, 11110, 111111\}$$

Para calcular L^*, devemos calcular L^i para cada i, e tomar a união de todas essas linguagens. L^i tem 2^i elementos. Embora cada L^i seja finito, a união do número infinito de termos L^i é em geral uma linguagem infinita, como em nosso exemplo.

Seja agora L o conjunto de todos os strings de 0's. Observe que L é infinito, diferente do nosso exemplo anterior, que é uma linguagem finita. Entretanto, não é difícil descobrir o que é L^*. $L^0 = \{\varepsilon\}$, como sempre. $L^1 = L$. L^2 é o conjunto

[1] O termo "fechamento de Kleene" se refere a S. C. Kleene, que deu origem à notação de expressões regulares e a esse operador.

de strings que podem ser formados tomando-se um string de 0's e concatenando-se esse string com outro string de 0's. O resultado ainda é um string de 0's. De fato, todo string de 0's pode ser escrito como a concatenação de dois strings de 0's (não se esqueça de que ε é um "string de 0's"; esse string sempre pode ser um dos dois strings que concatenamos). Desse modo, $L^2 = L$. Da mesma forma, $L^3 = L$ e assim por diante. Assim, a união infinita $L^* = L^0 \cup L^1 \cup L^2$ é $L \cup \ldots$ é L no caso particular em que a linguagem L é o conjunto de todos os strings de 0's.

Como um último exemplo, $\emptyset^* = \{\varepsilon\}$. Observe que $\emptyset^* = \{\varepsilon\}$, enquanto $\emptyset^i$, para qualquer $i \geq 1$, é vazio, pois não podemos selecionar quaisquer strings do conjunto vazio. Na verdade, $\emptyset$ é uma das duas únicas linguagens cujo fechamento *não* é infinito. □

3.1.2 Construindo expressões regulares

As álgebras de todos os tipos começam com algumas expressões elementares, normalmente constantes e/ou variáveis. As álgebras então nos permitem construir mais expressões aplicando um certo conjunto de operadores a essas expressões elementares e às expressões previamente construídas. Em geral, também é necessário algum método para agrupar operadores com seus operandos, tais como parênteses. Por exemplo, a álgebra aritmética familiar começa com constantes como inteiros e números reais, além de variáveis, e elabora expressões mais complexas com operadores aritméticos como + e ×.

O uso do operador estrela

Vimos o operador estrela pela primeira vez na Seção 1.5.2, em que o aplicamos a um alfabeto, por exemplo, Σ^*. Esse operador formava todos os strings cujos símbolos eram escolhidos no alfabeto Σ. O operador de fechamento é essencialmente igual, embora exista uma distinção sutil de tipos.

Suponha que L seja a linguagem que contém strings de comprimento 1 e, para cada símbolo a em Σ, existe um string a em L. Então, embora L e Σ "pareçam" iguais, eles são de tipos diferentes; L é um conjunto de strings, e Σ é um conjunto de símbolos. Por outro lado, L^* denota a mesma linguagem de Σ^*.

A álgebra de expressões regulares segue esse padrão, usando constantes e variáveis que denotam linguagens, e operadores para as três operações da Seção 3.1.1 – união, ponto e estrela. Podemos descrever as expressões regulares recursivamente, como a seguir. Nessa definição, não só descrevemos quais são as expressões regulares válidas mas, para cada expressão regular E, descrevemos a linguagem que ela representa, que denotamos por $L(E)$.

BASE: A base consiste em três partes:

1. As constantes ε e $\emptyset$ são expressões regulares, denotando as linguagens $\{\varepsilon\}$ e $\emptyset$, respectivamente. Isto é, $L(\varepsilon) = \{\varepsilon\}$, e $L(\emptyset) = \emptyset$.
2. Se a é qualquer símbolo, então **a** é uma expressão regular. Essa expressão denota a linguagem $\{a\}$. Isto é, $L(\mathbf{a}) = \{a\}$. Note que usamos a fonte em **negrito** para denotar uma expressão correspondente a um símbolo. A correspondência, por exemplo, o fato de que **a** se refere a a, deve ser óbvia.
3. Uma variável, em geral escrita em maiúsculas e em itálico, como L, é uma variável que representa qualquer linguagem.

INDUÇÃO: Há quatro partes para a etapa indutiva, uma para cada um dos três operadores e uma para a introdução de parênteses.

1. Se E e F são expressões regulares, então $E + F$ é uma expressão regular denotando 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, $L(EF) = L(E)L(F)$.

Expressões e suas linguagens

No sentido estrito, uma expressão regular E é apenas uma expressão, não uma linguagem. Devemos usar $L(E)$ quando quisermos nos referir à linguagem que E denota. Porém, é prática comum fazer referência a, digamos, "E" quando realmente queremos nos referir a "$L(E)$". Usaremos essa convenção desde que fique claro que estamos nos referindo a uma linguagem, e não a uma expressão regular.

Observe que o ponto pode opcionalmente ser usado para denotar o operador de concatenação, seja como uma operação sobre linguagens ou como o operador em uma expressão regular. Por exemplo, **0.1** é uma expressão regular que significa o mesmo que **01** e representa a linguagem $\{01\}$. Porém, evitaremos o ponto como concatenação em expressões regulares.[2]

3. Se E é uma expressão regular, então E^* é uma expressão regular, denotando o fechamento de $L(E)$. Isto é, $L(E^*) = (L(E))^*$.
4. Se E é uma expressão regular, então (E), E entre parênteses, também é uma expressão regular, denotando a mesma linguagem que E. Formalmente, $L((E)) = L(E)$.

[2] Na verdade, as expressões regulares do UNIX utilizam o ponto para uma finalidade completamente diferente: representar qualquer caractere ASCII.

Exemplo 3.2: Vamos escrever uma expressão regular para o conjunto de strings que consistem em 0's e 1's alternados. Primeiro, vamos desenvolver uma expressão regular para a linguagem que consiste no único string 01. Podemos então usar o operador estrela para obter uma expressão correspondente a todos os strings da forma 0101 ... 01.

A regra básica para expressões regulares nos diz que **0** e **1** são expressões que denotam as linguagens {0} e {1}, respectivamente. Se concatenarmos as duas expressões, obtermos uma expressão regular para a linguagem {01}; essa expressão é **01**. Como regra geral, se quisermos uma expressão regular para a linguagem consistimos apenas no string *w*, usaremos o próprio *w* como a expressão regular. Observe que, na expressão regular, os símbolos de *w* serão escritos normalmente em **negrito**, mas a mudança de fonte serve apenas para ajudá-lo a distinguir expressões de strings e não deve ser considerada significativa.

Agora, para obter todos os strings que consistem em zero ou mais ocorrências de 01, usamos a expressão regular $(\mathbf{01})^*$. Note que primeiro colocamos parênteses em torno de **01**, a fim de evitar confusão com a expressão $\mathbf{01}^*$, cuja linguagem representa todos os strings que consistem em 0 e qualquer número de 1's. A razão para essa interpretação é explicada na Seção 3.1.3 mas, em poucas palavras, a estrela tem precedência sobre o ponto e, por conseguinte, o argumento da estrela é selecionado antes da execução de quaisquer concatenações.

Porém, $L((\mathbf{01})^*)$ não é exatamente a linguagem que queremos. Ela inclui apenas os strings de 0's e 1's alternadas que começam com 0 e terminam com 1. Também precisamos considerar a possibilidade de haver 1 no início e/ou 0 no final. Uma abordagem é construir mais três expressões regulares que tratem as outras três possibilidades. Isto é, $(\mathbf{10})^*$ representa os strings alternados que começam com 1 e terminam com 0, enquanto $\mathbf{0}(\mathbf{10})^*$ pode ser usado para strings que começam e terminam com 0 e $\mathbf{1}(\mathbf{01})^*$ serve para strings que começam e terminam com 1. A expressão regular completa é

$$(\mathbf{01})^* + (\mathbf{10})^* + \mathbf{0}(\mathbf{10})^* + \mathbf{1}(\mathbf{01})^*$$

Observe que usamos o operador + para fazer a união das quatro linguagens que juntas nos dão todos os strings com 0's e 1's alternados.

Contudo, existe outra abordagem que gera uma expressão regular de aparência bem diferente e que também é um pouco mais sucinta. Comece novamente com a expressão $(\mathbf{01})^*$. Podemos adicionar um 1 opcional no início, se concatenarmos à esquerda com a expressão $\varepsilon + \mathbf{1}$. Da mesma forma, adicionamos um 0 opcional no final com a expressão $\varepsilon + \mathbf{0}$. Por exemplo, usando a definição do operador +:

$$L(\varepsilon + \mathbf{1}) + L(\varepsilon) \cup L(\mathbf{1}) = \{\varepsilon\} \cup \{1\} = \{\varepsilon, 1\}$$

Se concatenarmos essa linguagem com qualquer outra linguagem L, a escolha de ε nos dará todos os strings em L, enquanto a escolha de 1 nos dará $1w$ para todo string w em L. Desse modo, outra expressão para o conjunto de strings que alternam 0's e 1's é:

$$(\varepsilon + 1)\,(01)^*(\varepsilon + 0)$$

Observe que precisamos de parênteses em torno de cada uma das expressões adicionadas, a fim de assegurar que os operadores serão agrupados corretamente. □

3.1.3 Precedência de operadores em expressões regulares

Como em outras álgebras, os operadores nas expressões regulares têm uma ordem de "precedência" predefinida, o que significa que os operadores estão associados com seus operandos em uma ordem específica. Estamos familiarizados com a noção de precedência desde as expressões aritméticas comuns. Por exemplo, sabemos que $xy+z$ agrupa o produto xy antes da soma, e assim é equivalente à expressão com parênteses $(xy) + z$ e não à expressão $x(y + z)$. De modo semelhante, agrupamos dois operadores iguais a partir da esquerda em aritmética, e então $x - y - z$ é equivalente a $(x - y) - z$, e não a $- (y - z)$. No caso de expressões regulares, utiliza-se a seguinte ordem de precedência para os operadores:

1. O operador estrela é o de precedência mais alta. Isto é, ele se aplica apenas à menor seqüência de símbolos à sua esquerda que seja uma expressão regular bem formada.
2. Em seguida na precedência temos o operador de concatenação ou "ponto". Depois de agrupar todas as estrelas a seus operandos, agrupamos os operadores de concatenação a seus operandos. Isto é, todas as expressões que são *justapostas* (adjacentes, sem nenhum operador interveniente) são agrupadas juntas. Tendo em vista que o operador de concatenação é um operador associativo, não importa em que ordem agrupamos concatenações consecutivas, embora, se existir uma opção, você deva agrupá-los a partir da esquerda. Por exemplo, 012 é agrupado como (01)2.
3. Finalmente, todas as uniões (operadores +) são agrupadas com seus operandos. Tendo em vista que a união também é associativa, novamente é pouco importante a ordem em que são agrupadas uniões consecutivas, mas vamos pressupor o agrupamento a partir da esquerda.

É claro que às vezes não queremos que o agrupamento em uma expressão regular siga as regras de precedência dos operadores. Nesse caso, somos livres

para usar parênteses com a finalidade de agrupar operandos exatamente como pretendemos. Além disso, não há nada de errado em colocar parênteses ao redor de operandos que você deseja agrupar, mesmo que o agrupamento desejado esteja implícito pelas regras de precedência.

Exemplo 3.3: A expressão $01^* + 1$ é agrupada como $(0(1^*)) + 1$. O operador estrela é agrupado primeiro. Tendo em vista que o símbolo 1 imediatamente à sua esquerda é uma expressão regular válida, esse é o único operando da estrela. Em seguida, agrupamos a concatenação entre 0 e (1^*), o que nos dá a expressão $(0(1^*))$. Finalmente, o operador de união conecta essa última expressão e a expressão à sua direita, que é 1.

Note que a linguagem de uma expressão dada, agrupada de acordo com as regras de precedência, é o string 1 mais todos os strings que consistem em 0 seguido por qualquer número de 1's (inclusive nenhum). Se tivéssemos optado por agrupar o ponto antes da estrela, poderíamos ter usado parênteses, como $(01)^* + 1$. A linguagem dessa expressão é o string 1 e todos os strings que repetem 01, zero ou mais vezes. Se tivéssemos escolhido agrupar a união primeiro, poderíamos ter acrescentado parênteses em torno da união para tornar a expressão $0(1^* + 1)$. A linguagem dessa expressão é o conjunto de strings que começam com 0 e têm qualquer número de 1's em seguida. □

3.1.4 Exercícios para a Seção 3.1

Exercício 3.1.1: Escreva expressões regulares correspondentes às seguintes linguagens:

* a) O conjunto de strings sobre o alfabeto $\{a, b, c\}$ que contém pelo menos um a e pelo menos um b.

b) O conjunto de strings de 0's e 1's cujo décimo símbolo a partir da extremidade direita é 1.

c) O conjunto de strings de 0's e 1's com no máximo um par de 1's consecutivos.

! Exercício 3.1.2: Escreva expressões regulares correspondentes às seguintes linguagens:

* a) O conjunto de todos os strings de 0's e 1's tais que todo par de 0's adjacentes aparece antes de qualquer par de 1's adjacentes.

b) O conjunto de strings de 0's e 1's cujo número de 0's é divisível por 5.

!! Exercício 3.1.3: Escreva expressões regulares correspondentes às seguintes linguagens:

a) O conjunto de todos os strings de 0's e 1's que não contêm 101 como um substring.

b) O conjunto de todos os strings com um número igual de 0's e 1's, tais que nenhum prefixo tenha dois 0's a mais que os 1's, nem dois 1's a mais que os 0's.

c) O conjunto de strings de 0's e 1's cujo número de 0's é divisível por 5 e cujo número de 1's é par.

! **Exercício 3.1.4:** Forneça descrições em português das linguagens correspondentes às seguintes expressões regulares:

* a) $(1 + \varepsilon)(00^*1)^*0^*$.

b) $(0^*1^*)^*000(0 + 1)^*$.

c) $(0+10)^*1^*$.

*! **Exercício 3.1.5:** No Exemplo 3.1, destacamos que $\emptyset$ é uma das duas linguagens cujo fechamento é finito. Qual é a outra?

3.2 Autômatos finitos e expressões regulares

Embora a abordagem de expressões regulares para descrever linguagens seja fundamentalmente distinta da abordagem de autômatos finitos, essas duas notações representam exatamente o mesmo conjunto de linguagens, que denominamos "linguagens regulares". Já mostramos que os autômatos finitos determinísticos e os dois tipos de autômatos finitos não-determinísticos – com e sem ε-transições – aceitam a mesma classe de linguagens. Para mostrar que as expressões regulares definem a mesma classe, devemos mostrar que:

1. Toda linguagem definida por um desses autômatos também é definida por uma expressão regular. Para essa prova, podemos supor que a linguagem é aceita por algum DFA.
2. Toda linguagem definida por uma expressão regular é definida por um desses autômatos. Para essa parte da prova, é mais fácil mostrar que existe um NFA com ε-transições que aceita a mesma linguagem.

A Figura 3.1 mostra todas as equivalências que provamos ou provaremos. Um arco da classe X para a classe Y significa que provamos que toda linguagem definida pela classe X também é definida pela classe Y. Tendo em vista que o grafo é fortemente conectado (isto é, podemos ir de cada um dos quatro nós a qualquer outro nó), vemos que todas as quatro classes são na realidade a mesma classe.

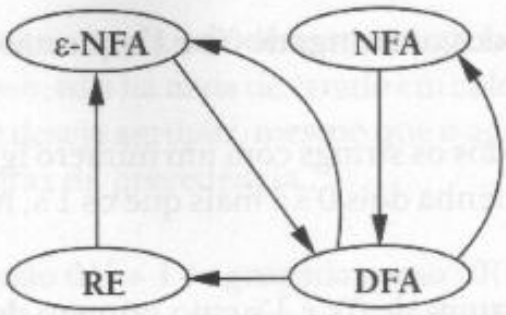

Figura 3.1: Plano para mostrar a equivalência entre as quatro notações diferentes para linguagens regulares

3.2.1 De DFA's para expressões regulares

A construção de uma expressão regular para definir a linguagem de qualquer DFA é surpreendentemente complicada. Em linhas gerais, construímos expressões que descrevem conjuntos de strings que identificam certos caminhos no diagrama de transições do DFA. Porém, só se permite que os caminhos passem por um subconjunto limitado dos estados. Em uma definição indutiva dessas expressões, começamos com as expressões mais simples que descrevem caminhos que não podem passar por *quaisquer* estados (isto é, eles são nós únicos ou arcos únicos) e construímos indutivamente as expressões que permitem a passagem dos caminhos por conjuntos de estados progressivamente maiores. Por fim, se permite que os caminhos passem por qualquer estado; isto é, as expressões que geramos no final representam todos os caminhos possíveis. Essas idéias aparecem na prova do teorema a seguir.

Teorema 3.4: Se $L = L(A)$ para algum DFA A, então existe uma expressão regular R tal que $L = L(R)$.

PROVA: Vamos supor que os estados de A sejam $\{1, 2, ..., n\}$ para algum inteiro n. Não importa quais sejam realmente os estados de A, haverá n deles para algum n finito e, renomeando os estados, podemos nos referir aos estados dessa maneira, como se eles fossem os primeiros n inteiros positivos. Nossa primeira e mais difícil tarefa é construir uma coleção de expressões regulares que descreva conjuntos de caminhos progressivamente mais amplos no diagrama de transições de A.

Vamos usar $R_{ij}^{(k)}$ como o nome de uma expressão regular cuja linguagem é o conjunto de strings w tais que w é o rótulo de um caminho do estado i ao estado j em A, e esse caminho não tem nenhum nó intermediário cujo número seja maior que k. Observe que os pontos de início e término do caminho não são "intermediários", e assim não há nenhuma restrição de que i e/ou j seja menor que ou igual a k.

A Figura 3.2 sugere o requisito sobre os caminhos representados por $R_{ij}^{(k)}$. Na figura, a dimensão vertical representa o estado, desde 1 na parte inferior até n na parte superior, e a dimensão horizontal representa o percurso ao longo do caminho. Note que nesse diagrama mostramos ao mesmo tempo i e j maiores que k, mas um deles ou ambos poderiam ser iguais ou menores que k. Observe também que o caminho passa duas vezes pelo nó k, mas nunca passa por um estado mais alto que k, exceto nos extremos.

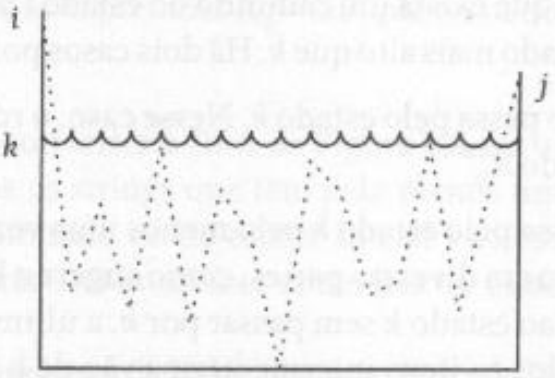

Figura 3.2: Um caminho cujo rótulo está na linguagem da expressão regular $R_{ij}^{(k)}$

Para construir as expressões $R_{ij}^{(k)}$, usamos a definição indutiva a seguir, começando em $k = 0$ e alcançando finalmente $k = n$. Observe que, quando $k = n$, não existe nenhuma restrição sobre os caminhos representados, pois *não existem* estados maiores que n.

BASE: A base é $k = 0$. Tendo em vista que todos os estados são numerados com 1 ou algum valor superior, a restrição sobre os caminhos é que o caminho não deve ter absolutamente nenhum estado intermediário. Só existem dois tipos de caminhos que satisfazem a tal condição:

1. Um arco do nó (estado) i para o nó j.
2. Um caminho de comprimento 0 que consiste apenas em algum nó i.

Se $i \neq j$, apenas o caso (1) é possível. Devemos examinar o DFA A e encontrar os símbolos de entrada a tais que exista uma transição do estado i para o estado j para o símbolo a.

a) Se não existe tal símbolo a, então $R_{ij}^{(0)} = \emptyset$.

b) Se existe exatamente um símbolo a, então $R_{ij}^{(0)} = \mathrm{a}$.

c) Se existem símbolos $a_1, a_2, \ldots, a_k$ que rotulam arcos do estado i para o estado j, então $R_{ij}^{(0)} = \mathrm{a}_1 + \mathrm{a}_2, + \ldots + \mathrm{a}_k$.

Porém, se $i = j$, então os caminhos válidos são o caminho de comprimento 0 e todos os loops de i até ele mesmo. O caminho de comprimento 0 é representado pela expressão regular ε, pois esse caminho não contém nenhum símbolo. Desse modo, adicionamos ε às várias expressões criadas em (a) até (c) anteriormente. Isto é, no caso (a) [nenhum símbolo a] a expressão se torna ε; no caso (b) [um símbolo a] a expressão se torna $\varepsilon + \mathbf{a}$ e, no caso (c) [vários símbolos] a expressão se torna $\varepsilon + \mathbf{a}_1 + \mathbf{a}_2 + \ldots + \mathbf{a}_k$.

INDUÇÃO: Suponha que exista um caminho do estado i para o estado j que não passe por nenhum estado mais alto que k. Há dois casos possíveis a considerar:

1. O caminho não passa pelo estado k. Nesse caso, o rótulo do caminho está na linguagem de $R_{ij}^{(k-1)}$.
2. O caminho passa pelo estado k pelo menos uma vez. Assim, podemos dividir o caminho em diversas partes, como sugere a Figura 3.3. A primeira vai do estado i ao estado k sem passar por k, a última vai de k a j sem passar por k, e todos os itens intermediários vão de k até eles mesmos, sem passarem por k. Note que, se o caminho passar pelo estado k apenas uma vez, não haverá nenhum item "intermediário", mas apenas um caminho de i a k e um caminho de k a j. O conjunto de rótulos para todos os caminhos desse tipo é representado pela expressão regular $R_{ik}^{(k-1)} R_{kk}^{(k-1)*} R_{kj}^{(k-1)}$. Ou seja, a primeira expressão representa a parte do caminho que chega ao estado k pela primeira vez, a segunda representa a parte que vai de k para ele mesmo, zero vezes, uma vez, ou mais de uma vez, e a terceira expressão representa a parte do caminho que deixa k pela última vez e vai para o estado j.

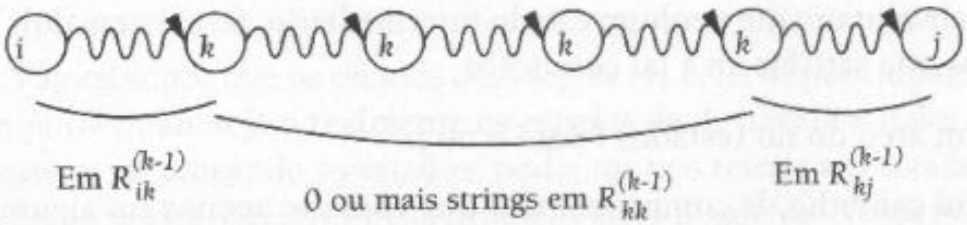

Figura 3.3: Um caminho de i até j pode ser dividido em segmentos em cada ponto no qual ele passa pelo estado k

Quando combinamos as expressões correspondentes aos caminhos dos dois tipos anteriores, temos a expressão

$$R_{ij}^{(k)} = R_{ij}^{(k-1)} + R_{ik}^{(k-1)} (R_{kk}^{(k-1)})^* R_{kj}^{(k-1)}$$

para os rótulos de todos os caminhos desde o estado i até o estado j que não passam por nenhum estado mais alto que k. Se construirmos essas expressões em ordem crescente do sobrescrito, como cada $R_{ij}^{(k)}$ depende apenas de expressões com um sobrescrito menor, todas as expressões estarão disponíveis quando houver necessidade de sua utilização.

Eventualmente, teremos $R_{ij}^{(n)}$ para todo i e j. Podemos supor que o estado 1 é o estado inicial, embora os estados de aceitação possam formar qualquer conjunto dos estados. A expressão regular para a linguagem do autômato é então a soma (união) de todas as expressões $R_{ij}^{(n)}$ tais que o estado j é um estado de aceitação. □

Exemplo 3.5: Vamos converter o DFA da Figura 3.4 em uma expressão regular. Esse DFA aceita todos os strings que têm pelo menos um valor 0. Para ver por quê, observe que o autômato vai do estado inicial 1 ao estado de aceitação 2 assim que vê uma entrada 0. O autômato então fica no estado 2 sobre todas as seqüências de entrada.

Aqui estão as expressões de base na construção do Teorema 3.4.

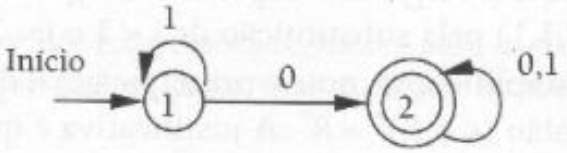

Figura 3.4: Um DFA que aceita todos os strings que têm pelo menos um 0

$R_{11}^{(0)}$	$\varepsilon + 1$
$R_{12}^{(0)}$	0
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$(\varepsilon + 0 + 1)$

BIBLIOTECA 99238 UESC

Por exemplo, $R_{11}^{(0)}$ tem o termo ε porque os estados inicial e final são o mesmo estado 1. Ele tem o termo 1 porque existe um arco do estado 1 ao estado 1 sobre a entrada 1. Como outro exemplo, $R_{12}^{(0)}$ é 0 porque existe um arco rotulado por 0 do estado 1 para o estado 2. Não é nenhum termo ε porque os estados inicial e final são diferentes. Como um terceiro exemplo, $R_{21}^{(0)} = \emptyset$, porque não existe nenhum arco do estado 2 para o estado 1.

Agora, devemos executar a parte de indução, criando expressões mais complexas que primeiro levam em conta caminhos que passam pelo estado 1, e depois caminhos que passam pelos estados 1 e 2, isto é, por qualquer caminho. As regras para calcular as expressões $R_{ij}^{(1)}$ são instâncias da regra geral dada na parte indutiva do Teorema 3.4:

$$R_{ij}^{(1)} = R_{ij}^{(0)} + R_{i1}^{(0)} (R_{11}^{(0)})^* R_{1j}^{(0)} \tag{3.1}$$

A tabela da Figura 3.5 apresenta primeiro as expressões calculadas por substituição direta na fórmula anterior, e depois uma expressão simplificada que podemos mostrar, por um raciocínio *ad hoc*, que representa a mesma linguagem que a expressão mais complexa.

	Por substituição direta	Justificativa
$R_{11}^{(1)}$	$\varepsilon + 1 + (\varepsilon + 1)(\varepsilon + 1)^*(\varepsilon + 1)$	1^*
$R_{12}^{(1)}$	$0 + (\varepsilon + 1)(\varepsilon + 1)^*0$	1^*0
$R_{21}^{(1)}$	$\emptyset + \emptyset\,(\varepsilon + 1)^*(\varepsilon + 1)$	$\emptyset$
$R_{22}^{(1)}$	$\varepsilon + 0 + 1 + \emptyset\,(\varepsilon + 1)^*0$	$\varepsilon + 0 + 1$

Figura 3.5: Expressões regulares correspondentes a caminhos que só podem passar pelo estado 1

Por exemplo, considere $R_{12}^{(1)}$. Sua expressão é $R_{12}^{(0)} + R_{11}^{(0)} (R_{11}^{(0)})^* R_{12}^{(0)}$, que obtemos a partir de (3.1) pela substituição de $i = 1$ e $j = 2$.

Para entender a simplificação, note o princípio geral de que, se R é qualquer expressão regular, então $(\varepsilon + R)^* = R^*$. A justificativa é que ambos os lados da equação descrevem a linguagem que consiste em qualquer concatenação de zero ou mais strings de $L(R)$. Em nosso caso, temos $(\varepsilon + 1)^* = 1^*$; note que ambas as expressões denotam qualquer número de 1's. Além disso, $(\varepsilon + 1)1^* = 1^*$. Mais uma vez, podemos observar que ambas as expressões denotam "qualquer número de 1's". Desse modo, a expressão original $R_{12}^{(1)}$ é equivalente a $0 + 1^*0$. Essa expressão denota a linguagem que contém o string 0 e todos os strings que consistem em um 0 precedido por qualquer número de 1's. Essa linguagem também é representada pela expressão mais simples 1^*0.

A simplificação de $R_{11}^{(1)}$ é semelhante à simplificação de $R_{12}^{(1)}$ que acabamos de considerar. A simplificação de $R_{21}^{(1)}$ e $R_{22}^{(1)}$ depende de duas regras sobre o modo como $\emptyset$ opera. Para qualquer expressão regular R:

1. $\emptyset R = R\emptyset = \emptyset$. Isto é, $\emptyset$ é um *aniquilador* para concatenação; ele resulta na concatenação de si próprio, seja à esquerda ou à direita, com qualquer expressão. Essa regra faz sentido porque, para um string estar presente no resultado de uma concatenação, devemos encontrar strings de ambos os argumentos da concatenação. Sempre que um dos argumentos for $\emptyset$, será impossível encontrar um string a partir desse argumento.
2. $\emptyset + R = R + \emptyset = R$. Ou seja, $\emptyset$ é a identidade para a união; ele resulta na outra expressão sempre que aparece em uma união.

Como resultado, uma expressão como $\emptyset(\varepsilon+1)^*(\varepsilon+1)$ pode ser substituída por $\emptyset$. As duas últimas simplificações devem assim estar claras.

Agora vamos calcular as expressões $R_{ij}^{(2)}$. A regra indutiva aplicada com $k = 2$ nos dá:

$$R_{ij}^{(2)} = R_{ij}^{(1)} + R_{i2}^{(1)} (R_{22}^{(1)})^* R_{2j}^{(1)} \tag{3.2}$$

Se substituirmos as expressões simplificadas da Figura 3.5 em (3.2), obteremos as expressões da Figura 3.6. Essa figura também mostra simplificações que seguem os mesmos princípios descritos para a Figura 3.5.

	Por substituição direta	Justificativa
$R_{11}^{(2)}$	$1^* + 1^*0\,(\varepsilon + 0 + 1)\,\emptyset$	1^*
$R_{12}^{(2)}$	$1^*0 + 1^*0\,(\varepsilon + 0 + 1)^*(\varepsilon + 0 + 1)$	$1^*0(0 + 1)$
$R_{21}^{(2)}$	$\emptyset + (\varepsilon + 0 + 1)(\varepsilon + 0 + 1)^*\emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\varepsilon + 0 + 1 + (\varepsilon + 0 + 1)(\varepsilon + 0 + 1)^*(\varepsilon + 0 + 1)$	$(0 + 1)^*$

Figura 3.6: Expressões regulares para caminhos que podem passar por qualquer estado

A última expressão regular equivalente ao autômato da Figura 3.4 é construída tomando-se a união de todas as expressões em que o primeiro estado é o estado inicial e o segundo estado é de aceitação. Nesse exemplo, como 1 é o estado inicial e 2 é o único estado de aceitação, só precisamos da expressão $R_{12}^{(2)}$. Essa expressão é $1^*0(0+1)^*$. É simples interpretar essa expressão. Sua linguagem consiste em todos os strings que começam com zero ou mais 1's, depois têm um 0 e depois qualquer string constituído de 0's e 1's. Em outras palavras, a linguagem consiste em todos os strings de 0's e 1's com pelo menos um valor 0. □

3.2.2 Conversão de DFAs em expressões regulares por eliminação de estados

O método da Seção 3.2.1 para converter um DFA em uma expressão regular sempre funciona. De fato, como você deve ter notado, ele realmente não depende de o autômato ser determinístico, e poderia ter sido igualmente aplicado a um NFA ou mesmo a um ε-NFA. Porém, a construção da expressão regular é dispendiosa. Não apenas temos de construir cerca de n^3 expressões para um autômato de n estados, mas o comprimento da expressão pode crescer por um fator de 4 em média, em cada uma das n etapas indutivas, se não houver nenhuma

simplificação das expressões. Desse modo, as próprias expressões poderiam chegar à ordem de 4^n símbolos.

Existe uma abordagem semelhante que evita a duplicação do trabalho em alguns pontos. Por exemplo, todas as expressões com sobrescrito $(k + 1)$ na construção do Teorema 3.4 utilizam a mesma subexpressão $(R_{kk}^{(k)})^*$; o trabalho de escrever essa expressão é portanto repetido n^2 vezes.

A abordagem de construção de expressões regulares que vamos aprender agora envolve a eliminação de estados. Quando eliminamos um estado s, todos os caminhos que passaram por s não mais existem no autômato. Se a linguagem do autômato não tiver de mudar, devemos incluir, em um arco que vá diretamente de q a p, os rótulos de caminhos que foram de algum estado q para o estado p passando por s. Tendo em vista que o rótulo desse arco pode agora envolver strings, em vez de símbolos isolados, e pode haver até mesmo um número infinito de tais strings, não podemos simplesmente listar os strings como um rótulo. Felizmente, há um modo simples e finito de representar todos esses strings: usar uma expressão regular.

Desse modo, somos levados a considerar autômatos que têm expressões regulares como rótulos. A linguagem do autômato é a união sobre todos os caminhos desde o estado inicial até um estado de aceitação da linguagem formada por concatenação das linguagens das expressões regulares ao longo desse caminho. Observe que essa regra é coerente com a definição da linguagem para qualquer das variedades de autômatos que consideramos até agora. Cada símbolo a, ou ε, se ele for permitido, pode ser considerado uma expressão regular cuja linguagem é um único string, seja ele $\{a\}$ ou $\{\varepsilon\}$. Podemos considerar essa observação a base de um procedimento de eliminação de estados que descreveremos em seguida.

A Figura 3.7 mostra um estado genérico s prestes a ser eliminado. Supomos que o autômato do qual s é um estado tem estados predecessores $q_1, q_2, \ldots, q_k$ e estados sucessores $p_1, p_2, \ldots, p_m$ para s. É possível que alguns dos q's também sejam p's, mas supomos que s não está entre os valores de q ou de p, mesmo que exista um loop de s para ele mesmo, como sugere a Figura 3.7. Também mostramos uma expressão regular em cada arco de um dos q's para s; a expressão Q_i rotula o arco de q_i. Da mesma forma, mostramos uma expressão regular P_j rotulado o arco de s a p_i, para todo i. Mostramos um loop em s com o rótulo S. Finalmente, existe uma expressão regular R_{ij} no arco de q_i até p_j, para todo i e todo j. Note que alguns desses arcos podem não existir no autômato e, nesse caso, tomaremos a expressão nesse arco como $\emptyset$.

A Figure 3.8 mostra o que acontece quando eliminamos o estado s. Todos os arcos que envolvem o estado s são eliminados. Para compensar introduzimos, para cada predecessor q_i de s e para cada sucessor p_j de s, uma expressão regular que representa todos os caminhos que começam em q_i, vão até s, talvez façam um loop em torno de s zero ou mais vezes, e finalmente vão até p_j. A expressão

para esses caminhos é $Q_i S^* P_j$. Essa expressão é adicionada (com o operador de união) ao arco que vai de q_i a p_j. Se não houver nenhum arco $q_i\ p_j$, então primeiro introduziremos esse arco com a expressão regular $\emptyset$.

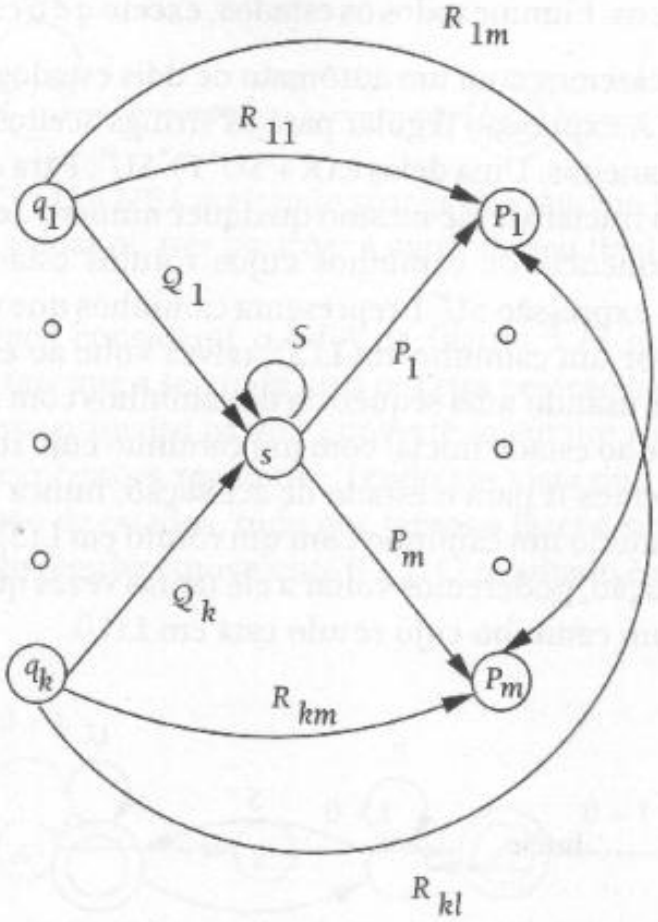

Figura 3.7: Um estado s prestes a ser eliminado

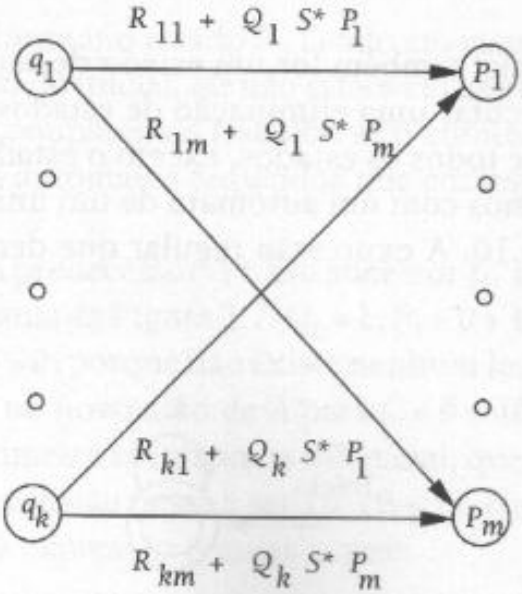

Figura 3.8: Resultado da eliminação do estado s da Figura 3.7

A estratégia para construir uma expressão regular a partir de um autômato finito é:

1. Para cada estado de aceitação q, aplique o processo de redução anterior para produzir um autômato equivalente com rótulos de expressões regulares nos arcos. Elimine todos os estados, exceto q e o estado inicial q_0.

2. Se $q \neq q_0$, ficaremos com um autômato de dois estados semelhante ao da Figura 3.9. A expressão regular para os strings aceitos pode ser descrita de várias maneiras. Uma delas é $(R + SU^*T)^*SU^*$. Para explicar, podemos ir do estado inicial até ele mesmo qualquer número de vezes, percorrendo uma seqüência de caminhos cujos rótulos estão em $L(R)$ ou em $L(SU^*T)$. A expressão SU^*T representa caminhos que vão até o estado de aceitação por um caminho em $L(S)$, talvez volte ao estado de aceitação várias vezes usando uma seqüência de caminhos com rótulos $L(U)$, e depois retorne ao estado inicial com um caminho cujo rótulo está em $L(T)$. Então, devemos ir para o estado de aceitação, nunca retornar ao estado inicial, seguindo um caminho com um rótulo em $L(S)$. Uma vez no estado de aceitação, poderemos voltar a ele tantas vezes quantas desejarmos, seguindo um caminho cujo rótulo está em $L(U)$.

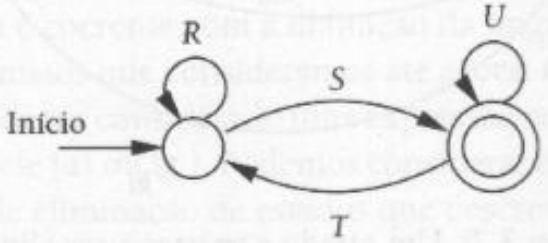

Figura 3.9: Um autômato genérico de dois estados

3. Se o estado inicial também for um estado de aceitação, então também deveremos executar uma eliminação de estados no autômato original que nos livre de todos os estados, exceto o estado inicial. Quando o fizermos, ficaremos com um autômato de um único estado, semelhante ao da Figura 3.10. A expressão regular que denota os strings que ele aceita é R^*.

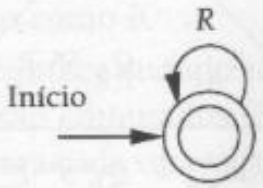

Figura 3.10: Um autômato genérico de um estado

4. A expressão regular desejada é a soma (união) de todas as expressões derivadas dos autômatos reduzidos correspondentes a cada estado de aceitação, pelas regras (2) e (3).

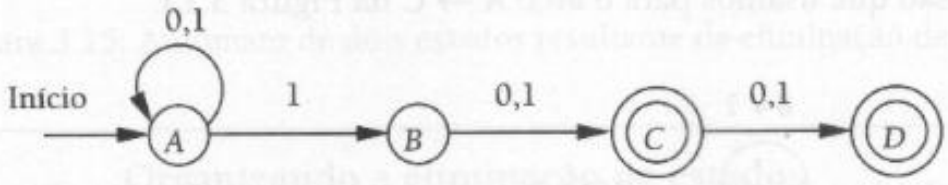

Figura 3.11: Um NFA aceitando strings que têm um símbolo 1 a duas ou três posições a partir do seu final

Exemplo 3.6: Vamos considerar o NFA da Figura 3.11 que aceita todos os strings de 0's e 1's tais que a segunda ou a terceira posição a partir do final tem um símbolo 1. Nosso primeiro passo é convertê-lo em um autômato com rótulos formados por expressões regulares. Tendo em vista que não foi executada nenhuma eliminação de estados, tudo que temos a fazer é substituir os rótulos "0,1" pela expressão regular equivalente $0 + 1$. O resultado é mostrado na Figura 3.12.

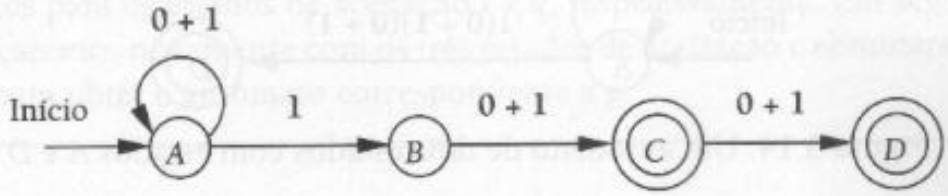

Figura 3.12: O autômato da Figura 3.11 com rótulos formados por expressões regulares

Primeiro, vamos eliminar o estado B. Tendo em vista que esse estado não é de aceitação nem é o estado inicial, ele não estará em nenhum dos autômatos reduzidos. Desse modo, pouparemos trabalho se o eliminarmos primeiro, antes de desenvolver os dois autômatos reduzidos que correspondem aos dois estados de aceitação.

O estado B tem um predecessor A e um sucessor C. Em termos das expressões regulares no diagrama da Figura 3.7: $Q_1 = 1, P_1 = 0 + 1, R_{11} = \emptyset$ (pois o arco de A para C não existe) e $S = \emptyset$ (porque não existe nenhum loop no estado B). Como resultado, a expressão no novo arco de A para C é $\emptyset + 1\emptyset^*(0 + 1)$.

Para simplificar, primeiro eliminamos o $\emptyset$ inicial, que pode ser ignorado em uma união. Assim, a expressão passa a ser $1\emptyset^*(0 + 1)$. Note que a expressão regular $\emptyset^*$ é equivalente à expressão regular ε, pois

$$L(\emptyset*) = \{\varepsilon\} \cup L(\emptyset) \cup L(\emptyset)L(\emptyset) \ldots$$

Tendo em vista que todos os termos exceto o primeiro são vazios, vemos que $L(\emptyset^*) = \{\varepsilon\}$, que é igual a $L(\varepsilon)$. Portanto, $1\emptyset^*(0+1)$ é equivalente a $1(0+1)$, a expressão que usamos para o arco $A \rightarrow C$ na Figura 3.13.

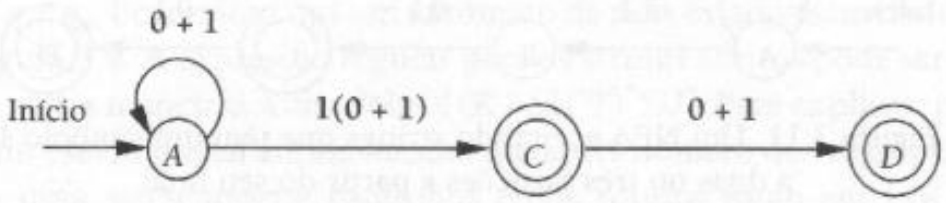

Figura 3.13: Eliminação do estado B

Agora, devemos nos desviar, eliminando os estados C e D em reduções separadas. Para eliminar o estado C, a mecânica é semelhante à que utilizamos anteriormente para eliminar o estado B, e o autômato resultante é mostrado na Figura 3.14.

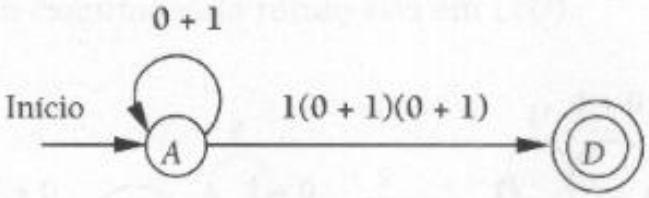

Figura 3.14: Um autômato de dois estados com estados A e D

Em termos do autômato genérico de dois estados da Figura 3.9, as expressões regulares da Figura 3.14 são: $R = 0 + 1$, $S = 1(0+1)(0+1)$, $T = \emptyset$ e $U = \emptyset$. A expressão U^* pode ser substituída por ε, isto é, eliminada em uma concatenação; a justificativa é que $\emptyset^* = \varepsilon$, como vimos anteriormente. Além disso, a expressão SU^*T é equivalente a $\emptyset$ porque T, um dos termos da concatenação, é $\emptyset$. Portanto, a expressão genérica $(R + SU^*T)^*SU^*$ é simplificada nesse caso para R^*S, ou $(0 + 1)^*1(0 + 1)(0 + 1)$. Em termos informais, a linguagem dessa expressão é qualquer string que termina em 1, seguido por dois símbolos dos quais cada um é ou 0 ou 1. Essa linguagem é uma parte dos strings aceitos pelo autômato da Figura 3.11: os strings cuja terceira posição a partir da extremidade final tem um símbolo 1.

Agora, devemos voltar à Figura 3.13 e eliminar o estado D em vez de C. Tendo em vista que D não tem nenhum sucessor, uma inspeção da Figura 3.7 nos informa que não haverá alterações em arcos, e o arco de C para D será eliminado juntamente com o estado D. O autômato de dois estados resultante é mostrado na Figura 3.15.

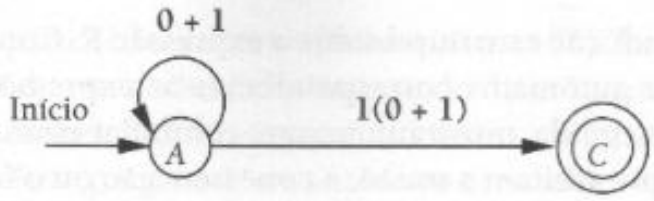

Figura 3.15: Autômato de dois estados resultante da eliminação de D

Organizando a eliminação de estados

Como observamos no Exemplo 3.6, quando um estado não é o estado inicial nem um estado de aceitação, ele é eliminado em todos os autômatos derivados. Desse modo, uma das vantagens do processo de eliminação de estados em comparação com a geração mecânica de expressões regulares que descrevemos na Seção 3.2.1 é que podemos começar eliminando definitivamente todos os estados que não são o estado inicial nem estados de aceitação, um de cada vez. Só teremos de começar a duplicar o esforço de redução quando precisarmos eliminar algum estado de aceitação.

Mesmo nesse caso, podemos combinar algumas partes do trabalho. Por exemplo, se houver três estados de aceitação p, q e r, poderemos eliminar p e depois efetuar ramificações para eliminar q ou r, produzindo assim os autômatos para os estados de aceitação r e q, respectivamente. Em seguida, começaremos novamente com os três estados de aceitação e eliminaremos q e r para obter o autômato correspondente a p.

Esse autômato é muito semelhante ao da Figura 3.14; somente o rótulo no arco do estado inicial até o estado de aceitação é diferente. Desse modo, podemos aplicar a regra para autômatos de dois estados e simplificar a expressão, obtendo $(0 + 1)^*1(0 + 1)$. Essa expressão representa o outro tipo de strings que o autômato aceita: aqueles que têm 1 na segunda posição a partir da extremidade final.

Só falta somar as duas expressões para obter a expressão correspondente ao autômato completo da Figura 3.11. Essa expressão é

$$(0 + 1)^*1(0 + 1) + (0 + 1)^*1(0 + 1)(0 + 1)$$

□

3.2.3 Conversão de expressões regulares em autômatos

Agora completaremos o plano da Figura 3.1, mostrando que toda linguagem L que é $L(R)$ para alguma expressão regular R, também é $L(E)$ para algum ε-NFA

E. A prova é uma indução estrutural sobre a expressão R. Começamos mostrando como construir autômatos correspondentes às expressões base: símbolos isolados, ε e $\emptyset$. Em seguida, mostramos como combinar esses autômatos em autômatos maiores que aceitam a união, a concatenação ou o fechamento da linguagem aceita por autômatos menores.

Todos os autômatos que construímos são ε-NFAs com um único estado de aceitação.

Teorema 3.7: Toda linguagem definida por uma expressão regular também é definida por um autômato finito.

PROVA: Suponha que $L = L(R)$ para uma expressão regular R. Mostramos que $L = L(E)$ para algum ε-NFA E com:

1. Exatamente um estado de aceitação.
2. Nenhum arco chega no estado inicial.
3. Nenhum arco sai do estado de aceitação.

A prova é por indução estrutural sobre R, seguindo a definição recursiva de expressões regulares que vimos na Seção 3.1.2.

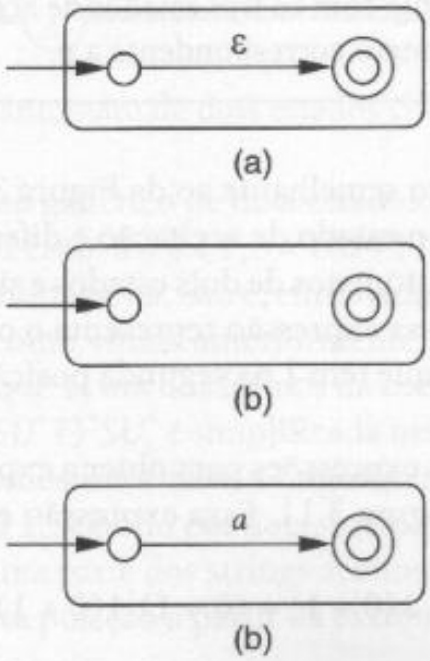

Figura 3.16: A base da construção de um autômato a partir de uma expressão regular

BASE: Há três partes para a base, mostradas na Figura 3.16. Na parte (a), vemos como manipular a expressão ε. Vemos facilmente que a linguagem do autômato deve ser $\{\varepsilon\}$, pois o único caminho do estado inicial até um estado de aceitação é

rotulado por ε. A parte (b) mostra a construção para ∅. Claramente, não existe nenhum caminho do estado inicial a um estado de aceitação, e assim ∅ é a linguagem desse autômato. Por fim, a parte (c) fornece o autômato para uma expressão regular a. A linguagem desse autômato consiste evidentemente no único string a, que também é $L(a)$. É fácil verificar que todos esses autômatos satisfazem às condições (1), (2) e (3) da hipótese indutiva.

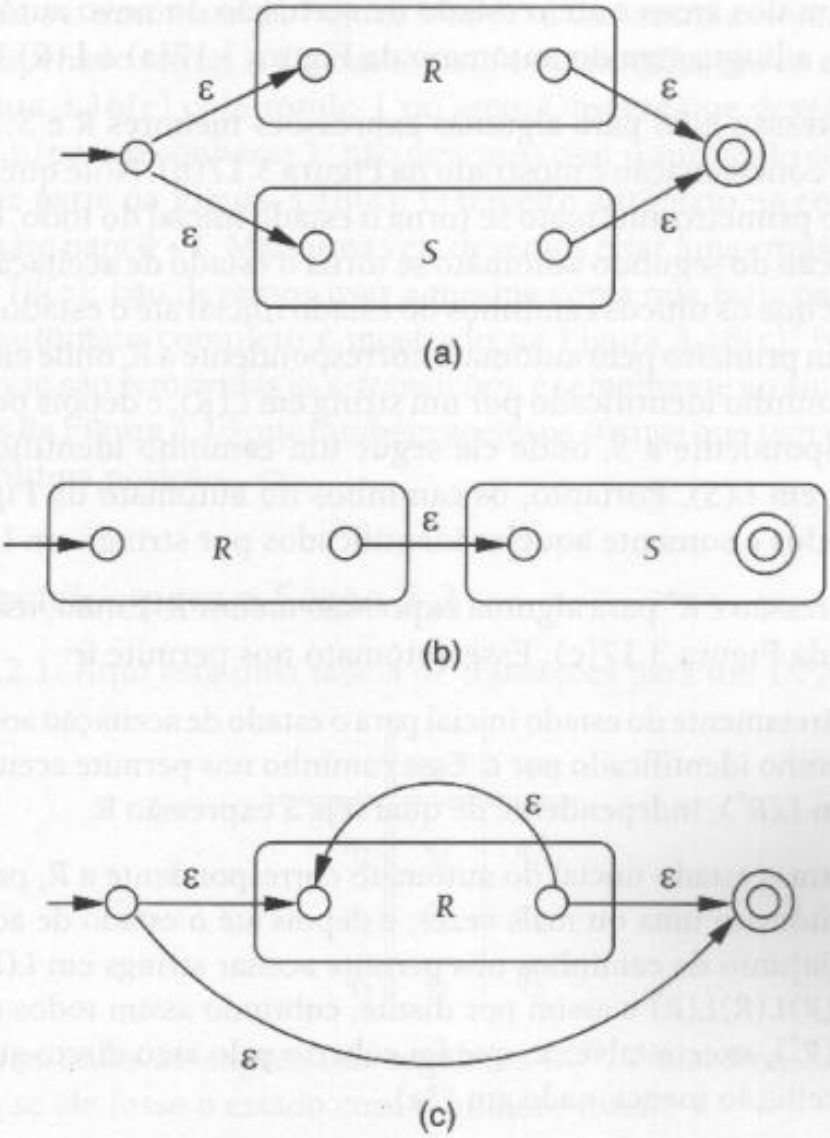

Figura 3.17: A etapa indutiva na construção de um ε-NFA a partir de uma expressão regular

INDUÇÃO: As três partes da indução são mostradas na Figura 3.17. Supomos que a afirmação do teorema é verdadeira para as subexpressões imediatas de uma dada expressão regular; isto é, as linguagens dessas subexpressões também são as linguagens dos ε-NFA's com um único estado de aceitação. Os quatro casos são:

1. A expressão é $R + S$ para algumas expressões menores R e S. Então o autômato da Figura 3.17(a) é adequado. Isto é, começando no novo estado inicial, podemos ir para o estado inicial do autômato correspondente a R ou do autômato correspondente a S. Em seguida, alcançamos o estado de aceitação de um desses autômatos, percorrendo um caminho identificado por algum string em $L(R)$ ou $L(S)$, respectivamente. Uma vez que alcançamos o estado de aceitação do autômato para R ou S, podemos seguir um dos arcos ε até o estado de aceitação do novo autômato. Desse modo, a linguagem do autômato da Figura 3.17(a) é $L(R) \cup L(S)$.

2. A expressão é RS para algumas expressões menores R e S. O autômato para a concatenação é mostrado na Figura 3.17(b). Note que o estado inicial do primeiro autômato se torna o estado inicial do todo, e o estado de aceitação do segundo autômato se torna o estado de aceitação do todo. A idéia é que os únicos caminhos do estado inicial até o estado de aceitação passem primeiro pelo autômato correspondente a R, onde ele deve seguir um caminho identificado por um string em $L(R)$, e depois pelo autômato correspondente a S, onde ele segue um caminho identificado por um string em $L(S)$. Portanto, os caminhos no autômato da Figura 3.17(b) são todos e somente aqueles identificados por strings em $L(R)L(S)$.

3. A expressão é R^* para alguma expressão menor R. Então, usamos o autômato da Figura 3.17(c). Esse autômato nos permite ir:

   (a) Diretamente do estado inicial para o estado de aceitação ao longo um caminho identificado por ε. Esse caminho nos permite aceitar ε, que está em $L(R^*)$, independente de qual seja a expressão R.

   (b) Para o estado inicial do autômato correspondente a R, passar por esse autômato uma ou mais vezes, e depois até o estado de aceitação. Esse conjunto de caminhos nos permite aceitar strings em $L(R)$, $L(R)L(R)$, $L(R)L(R)L(R)$ e assim por diante, cobrindo assim todos os strings em $L(R^*)$, exceto talvez ε, que foi coberto pelo arco direto até o estado de aceitação mencionado em (3a).

4. A expressão é (R) para alguma expressão menor R. O autômato correspondente a R também serve como autômato para (R), pois os parênteses não alteram a linguagem definida pela expressão.

É simples observar que os autômatos construídos satisfazem às três condições dadas na hipótese indutiva – um estado de aceitação, sem arcos chegando no estado inicial ou saindo do estado de aceitação. □

Exemplo 3.8: Vamos converter a expressão regular $(0 + 1)^* 1(0 + 1)$ em um ε-NFA. Nosso primeiro passo é construir um autômato para $0 + 1$. Usamos dois

autômatos construídos de acordo com a Figura 3.16(c), um com o rótulo **0** no arco e outro com o rótulo **1**. Esses dois autômatos são então combinados com o uso da construção de união da Figura 3.17(a). O resultado é mostrado na Figura 3.18(a).

Em seguida, aplicamos à Figura 3.18(a) a construção estrela da Figura 3.17(c). Esse autômato é mostrado na Figura 3.18(b). As duas últimas etapas envolvem a aplicação da construção de concatenação da Figura 3.17(b). Primeiro, conectamos o autômato da Figura 3.18(b) a um outro autômato projetado para aceitar apenas o string 1. Esse autômato é outra aplicação da construção de base da Figura 3.16(c) com rótulo **1** no arco. Observe que devemos criar um *novo* autômato para reconhecer 1; não devemos usar o autômato correspondente a 1 que faz parte da Figura 3.18(a). O terceiro autômato na concatenação é outro autômato para **0** + **1**. Mais uma vez, devemos criar uma cópia do autômato da Figura 3.18(a); não devemos usar a mesma cópia que fazia parte da Figura 3.18(b). O autômato completo é mostrado na Figura 3.18(c). Note que esse ε-NFA, quando são removidas as ε-transições, é semelhante ao autômato muito mais simples da Figura 3.15 que também aceita os strings que têm um símbolo 1 em sua penúltima posição. □

3.2.4 Exercícios para a Seção 3.2

Exercício 3.2.1: Aqui está uma tabela de transições para um DFA:

	0	1
$\rightarrow q_1$	q_2	q_1
q_2	q_3	q_1
$*q_3$	q_3	q_2

* a) Forneça todas as expressões regulares $R_{ij}^{(0)}$. *Nota*: Pense no estado q_i como se ele fosse o estado com o número inteiro i.

* b) Forneça todas as expressões regulares $R_{ij}^{(1)}$. Procure simplificar ao máximo as expressões.

c) Forneça todas as expressões regulares $R_{ij}^{(2)}$. Procure simplificar ao máximo as expressões.

d) Forneça uma expressão regular para a linguagem do autômato.

* e) Construa o diagrama de transições para o DFA e forneça uma expressão

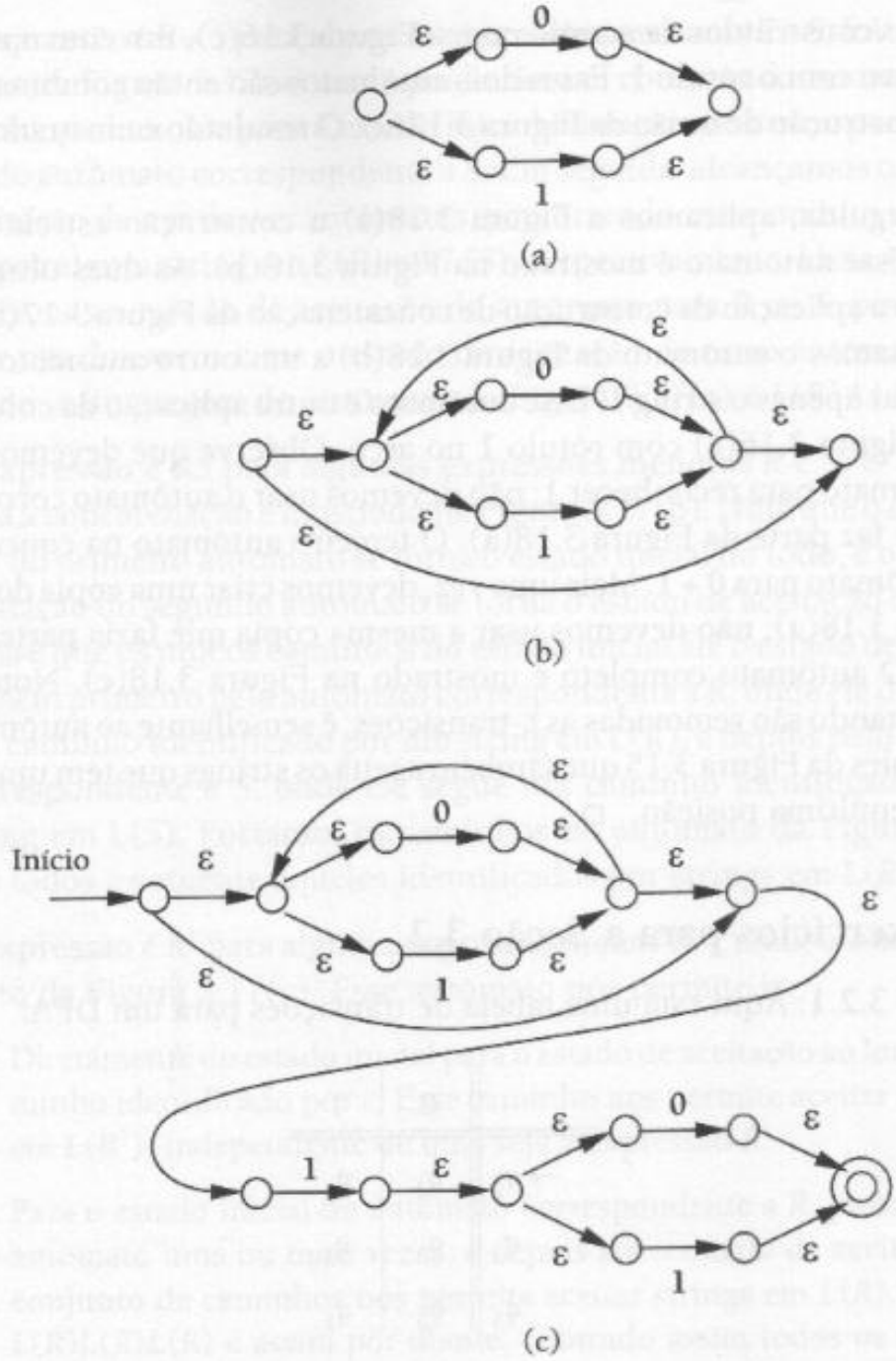

Figura 3.18: Autômatos construídos para o Exemplo 3.8

Exercício 3.2.2: Repita o Exercício 3.2.1 para o seguinte DFA:

	0	1
$\rightarrow q_1$	q_2	q_3
q_2	q_1	q_3
$*q_3$	q_2	q_1

Exercício 3.2.3: Converta o seguinte DFA em uma expressão regular, usando a técnica de eliminação de estados da Seção 3.2.2.

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

Exercício 3.2.4: Converta as expressões regulares a seguir em NFA's com ε-transições.

* a) 01^*.

b) $(0 + 1)01$.

c) $00(0 + 1)^*$.

Exercício 3.2.5: Elimine ε-transições dos seus ε-NFA's do Exercício 3.2.4. Uma solução para a parte (a) encontra-se na página Web do livro.

! **Exercício 3.2.6:** Seja $A = (Q, \Sigma, \delta, q_0, \{q_f\})$ um ε-NFA tal que não existe nenhuma transição chegando em q_0 e nenhuma transição saindo de q_f. Descreva a linguagem aceita por cada uma das seguintes modificações de A, em termos de $L = L(A)$:

*a) O autômato construído a partir de A pela adição de uma ε-transição de q_f para q_0.

* b) O autômato construído a partir de A pela adição de uma ε-transição de q_0 para todo estado alcançável a partir de q_0 (ao longo de um caminho cujos rótulos podem incluir símbolos de Σ, bem como ε).

c) O autômato construído a partir de A pela adição de uma ε-transição para q_f a partir de cada estado que possa alcançar q_f ao longo de algum caminho.

d) O autômato construído a partir de A fazendo-se ao mesmo tempo (b) e (c).

!! **Exercício 3.2.7:** Existem algumas simplificações para as construções do Teorema 3.7, em que convertemos uma expressão regular em um ε-NFA. Aqui estão três delas:

1. Para o operador de união, em vez de criar um novo estado inicial e novos

combine os dois estados de aceitação, fazendo todas as transições irem para o estado combinado em vez de irem para um dos estados de aceitação.

2. Para o operador de concatenação, combine o estado de aceitação do primeiro autômato com o estado inicial do segundo.

3. Para o operador de fechamento, simplesmente adicione ε-transições do estado de aceitação para o estado inicial e *vice-versa*.

Cada uma dessas simplificações, por si só, ainda gera uma construção correta; isto é, o ε-NFA resultante para qualquer expressão regular aceita a linguagem da expressão. Que subconjuntos das alterações (1), (2) e (3) podem ser reunidos e aplicados à construção, produzindo ainda um autômato correto para toda expressão regular?

*!! **Exercício 3.2.8**: Forneça um algoritmo que receba um DFA A e calcule o número de strings de comprimento n (para algum n dado, não relacionado ao número de estados de A) aceitos por A. Seu algoritmo deve ser polinomial tanto em n quanto no número de estados de A. *Sugestão*: Use a técnica sugerida pela construção do Teorema 3.4.

3.3 Aplicações de expressões regulares

Uma expressão regular que apresenta um "quadro" do padrão que queremos reconhecer é o meio preferido para aplicações que buscam padrões em textos. As expressões regulares são então compiladas, nos bastidores, em autômatos determinísticos ou não-determinísticos, que são então simulados para produzir um programa que reconhece padrões em textos. Nesta seção, consideraremos duas classes importantes de aplicações baseadas em expressão regulares: os analisadores léxicos e a busca em textos.

3.3.1 Expressões regulares no UNIX

Antes de examinarmos as aplicações, apresentaremos a notação do UNIX para expressões regulares estendidas. Essa notação nos dá uma série de recursos adicionais. Na verdade, as extensões do UNIX incluem certos recursos, especialmente a habilidade de nomear e fazer referência a strings anteriores que corresponderam a um padrão, o que realmente permite o reconhecimento de linguagens não regulares. Não examinaremos esses recursos aqui; em vez disso, introduziremos apenas as simplificações que permitem a escrita sucinta de expressões regulares complexas.

O primeiro aperfeiçoamento na notação de expressões regulares diz respeito ao fato de que a maioria das aplicações reais trabalha sobre o conjunto de ca-

racteres ASCII. Nossos exemplos utilizaram tipicamente um alfabeto pequeno, como {0, 1}. A existência de apenas dois símbolos nos permitiu escrever expressões sucintas como 0 + 1 para "qualquer caractere". Porém se houvesse, digamos, 128 caracteres, a mesma expressão envolveria a listagem de todos eles, e seria altamente inconveniente escrevê-la. Desse modo, as expressões regulares do UNIX nos permitem escrever *classes de caracteres* para representar grandes conjuntos de caracteres da forma mais sucinta possível. As regras para classes de caracteres são:

- O símbolo · (ponto) representa "qualquer caractere".
- A seqüência $[a_1 a_2 a_k]$ representa a expressão regular

$$a_1 + a_2 + \ldots + a_k$$

Essa notação poupa cerca de metade dos caracteres, pois não temos de escrever os sinais de "+". Por exemplo, poderíamos expressar os quatro caracteres usados em operadores de comparação da linguagem C por `[< >=!]`.

- Entre os colchetes, podemos colocar um intervalo da forma *x-y* para indicar todos os caracteres desde *x* até *y* na seqüência ASCII. Tendo em vista que os dígitos têm códigos em ordem numérica, como as letras maiúsculas antes das letras minúsculas, podemos expressar muitas das classes de caracteres que realmente nos interessam com apenas algumas seqüências de teclas. Por exemplo, os dígitos podem ser expressos por `[0-9]`, as letras maiúsculas podem ser expressas `[A-Z]` e o conjunto de todas as letras e dígitos pode ser expresso por `[A-Za-z0-9]`. Se quisermos incluir um sinal de subtração entre uma lista de caracteres, poderemos incluí-lo em primeiro ou em último lugar, de forma que ele não seja confundido com seu uso para formar um intervalo de caracteres. Por exemplo, o conjunto de dígitos, mais o ponto, o sinal de adição e o sinal de subtração que são usados para formar números decimais com sinal podem ser expressos por `[-+.0-9]`. Colchetes ou outros caracteres, que têm significados especiais nas expressões regulares do UNIX, podem ser representados como caracteres colocando-se antes deles um sinal de barra invertida (\).
- Há notações especiais para várias classes de caracteres mais comuns. Por exemplo:

a) `[:digit:]` é o conjunto de dez dígitos, igual a `[0-9]`.[3]

[3] A notação `[:digit:]` tem a vantagem de, no caso algum outro código diferente de ASCII seja usado, em que os dígitos não tivessem códigos consecutivos, `[:digit:]` ainda representaria `[0123456789]`, enquanto `[0-9]` representaria quaisquer caracteres que tivessem códigos entre os códigos correspondentes a 0 e 9, inclusive.

b) `[:alpha:]` representa qualquer caractere alfabético, da mesma forma que `[A-Za-z]`.

c) `[:alnum:]` representa os dígitos e as letras (caracteres alfabéticos e numéricos), da mesma forma que `[A-Za-z0-9]`.

Além disso, há vários operadores que são usados em expressões regulares do UNIX que não encontramos anteriormente. Nenhum desses operadores amplia o modo como as linguagens podem ser expressas, mas às vezes eles tornam mais fácil expressar o que queremos.

1. O operador | é usado em lugar de + para denotar união.
2. O operador ? significa "zero ou um de". Desse modo, *R*? no UNIX é igual a $\varepsilon + R$ na notação de expressões regulares deste livro.
3. O operador + significa "um ou mais de". Desse modo, *R*+ no UNIX é a abreviatura correspondente a RR^* em nossa notação.
4. O operador {*n*} significa "*n* cópias de". Desse modo, *R*{5} no UNIX é a abreviatura de *RRRRR*.

Observe que as expressões regulares do UNIX permitem o uso de parênteses para agrupar subexpressões, exatamente como ocorre no caso das expressões regulares descritas na Seção 3.1.2, e a mesma precedência de operadores é usada (com ?, + e {*n*} tratados como * no que se refere à precedência). O operador estrela (*) é usado no UNIX (sem estar sobrescrito, é claro) com o mesmo significado que usamos.

3.3.2 Análise léxica

Uma das aplicações mais antigas de expressões regulares era na especificação do componente de um compilador chamado "analisador léxico". Esse componente examina o programa-fonte e reconhece todos os *tokens*, os substrings de caracteres consecutivos que estão logicamente juntos. As palavras-chave e os identificadores são exemplos comuns de símbolos, mas existem muitos outros.

O conjunto completo de expressões regulares do UNIX

O leitor que quiser conhecer a lista completa de operadores e abreviaturas disponíveis na notação de expressões regulares do UNIX poderá encontrá-los nas páginas dos manuais correspondentes a diversos comandos. Existem algumas diferenças entre as várias versões do UNIX, mas um comando como `man grep` apresentará a notação usada para o comando `grep`, que é fundamental. A propósito, "Grep" significa "Global (search for) Regular Expression and Print (busca e impressão global de expressões regulares)".

O comando do UNIX `lex` e sua versão GNU `flex` aceitam como entrada uma lista de expressões regulares, no estilo do UNIX, cada uma seguida por uma seção de código entre colchetes que indica o que o analisador léxico deve fazer quando encontrar uma instância desse símbolo. Tal recurso é chamado *gerador de analisador léxico*, porque toma como entrada uma descrição de alto nível de um analisador e produz a partir dele uma função que é um analisador léxico operacional.

Comandos como `lex` e `flex` se mostraram muito úteis, porque a notação de expressões regulares é tão eficiente quanto necessário para descrever símbolos. Esses comandos podem usar o processo de conversão de expressão regular para DFA com o objetivo de gerar uma função eficiente que divide programas-fonte em tokens. Eles tornam a implementação de um analisador léxico o trabalho de uma tarde, enquanto antes do desenvolvimento dessas ferramentas baseadas em expressões regulares, a geração manual de um analisador léxico podia demorar meses. Além disso, se precisarmos modificar o analisador léxico por qualquer razão, com freqüência será uma simples questão de alterar uma expressão regular ou duas, em vez de ter de entrar nos mistérios de um código para corrigir um bug.

Exemplo 3.9: Na Figura 3.19 temos um exemplo de uma entrada parcial para o comando `lex`, descrevendo alguns dos tokens encontrados na linguagem C. A primeira linha trata a palavra-chave `else`, e a ação é retornar uma constante simbólica (`ELSE` nesse exemplo) para o processamento adicional pelo analisador sintático. A segunda linha contém uma expressão regular descrevendo identificadores: uma letra seguida por zero ou mais letras e/ou dígitos. A ação efetuada primeiro é a inclusão desse identificador na tabela de símbolos, se ele ainda não estiver na tabela; o `lex` isola o símbolo encontrado em um buffer, de modo que esse fragmento de código saiba exatamente qual identificador foi encontrado. Finalmente, o analisador léxico retorna a constante simbólica `ID`, escolhida nesse exemplo para representar identificadores.

A terceira entrada na Figura 3.19 corresponde ao sinal `>=` , um operador de dois caracteres. O último exemplo que mostramos corresponde ao sinal `=`, um operador de um caractere. Na prática, haveria expressões descrevendo cada uma das palavras-chave, cada um dos sinais e símbolos de pontuação, como vírgulas e parênteses, e famílias de constantes como números e strings. Vários desses itens são muito simples, apenas uma seqüência de um ou mais caracteres específicos. Porém, alguns têm características similares às dos identificadores, exigindo toda a capacidade da notação de expressões regulares para descrevê-los. Os inteiros, números em ponto flutuante, strings de caracteres e comentários são outros exemplos de conjuntos de strings que lucram com os recursos de expressões regulares presentes em comandos como `lex`. □

```
else                    {return(ELSE);}

[A-Za-z] [A-Za-z0-9]*   {código para inserir o identificador encontrado
                         na tabela de símbolos;
                         return(ID);
                        }

>=                      {return(GE);}

=                       {return(EQ);}

. . .
```

Figura 3.19: Um exemplo do lex

A conversão de uma coleção de expressões em um autômato, como aquelas sugeridas na Figura 3.19, se dá aproximadamente da maneira formal descrita nas seções anteriores. Começamos construindo um autômato para a união de todas as expressões. Em princípio, esse autômato nos diz apenas que *algum* símbolo foi reconhecido. Contudo, se seguirmos a construção do Teorema 3.7 para a união de expressões, o estado de ε-NFA informará exatamente qual símbolo foi reconhecido.

O único problema é que mais de um símbolo pode ser reconhecido ao mesmo tempo; por exemplo, o string else corresponde não apenas à expressão regular **else**, mas também à expressão relativa a identificadores. A solução padrão determina que o gerador de analisador léxico deve dar prioridade à primeira expressão listada. Desse modo, se quisermos que palavras-chaves como else sejam *reservadas* (não utilizáveis como identificadores), simplesmente iremos listá-las à frente da expressão dos identificadores.

3.3.3 Busca de padrões em textos

Na Seção 2.4.1, introduzimos a noção de que autômatos poderiam ser usados para procurar de forma eficiente um conjunto de palavras em um grande repositório como a Web. Embora as ferramentas e a tecnologia para fazer isso não estejam tão bem desenvolvidas quanto às de analisadores léxicos, a notação de expressões regulares é valiosa para descrever pesquisas de padrões interessantes. Quanto aos analisadores léxicos, a capacidade de ir da notação natural e descritiva de expressões regulares para uma implementação eficiente (baseada em autômatos) oferece um significativo impulso intelectual.

O problema geral para o qual a tecnologia de expressões regulares tem se mostrado útil é a descrição de uma classe vagamente definida de padrões em textos. A incerteza da descrição virtualmente garante que a princípio não descreveremos o padrão de forma correta – talvez nunca possamos obter exata-

mente a descrição correta. Usando-se a notação de expressões regulares, torna-se fácil descrever os padrões em alto nível, com pouco esforço, e modificar a descrição com rapidez quando algo der errado. Um "compilador" para expressões regulares é útil na transformação das expressões que escrevemos em código executável.

Vamos explorar um exemplo estendido do tipo de problema que surge em muitas aplicações da Web. Suponha que queremos examinar um número muito grande de páginas da Web e detectar endereços. Poderíamos simplesmente desejar criar uma listagem de mala direta. Ou talvez estejamos procurando classificar empresas comerciais por sua localização, para podermos responder a consultas como "encontrar um restaurante situado no máximo a 10 minutos de carro do local em que estou agora".

Em particular, vamos nos concentrar no reconhecimento de endereços residenciais. O que é um endereço residencial? Teremos de descobrir isso e, enquanto testamos o software, se chegarmos à conclusão que omitimos alguns casos, teremos de modificar as expressões para capturar o que perdemos. Para começar, um endereço residencial provavelmente se inicia com "Rua" ou sua abreviatura "R.". Porém, algumas pessoas vivem em "Avenidas" ou "Estradas", e esses itens também podem estar abreviados no endereço. Desse modo, poderíamos usar como início de nossa expressão regular algo como:

```
Rua|R\.|Avenida|Av\.|Estrada|Estr\.
```

Na expressão anterior, usamos a notação no estilo do UNIX, com a barra vertical, em lugar de +, como o operador de união. Note também que os pontos são inseridos como *caracteres de escape*, com uma barra invertida antes deles, porque o ponto tem o significado especial de "qualquer caractere" em expressões do UNIX e, nesse caso, realmente queremos apenas o caractere de ponto para finalizar as três abreviaturas.

A designação `Rua` deve ser seguida pelo nome da rua. Normalmente, o nome é uma letra maiúscula seguida por algumas letras minúsculas. Podemos descrever esse padrão pela expressão do UNIX `[A-Z][a-z]*`. Porém, algumas ruas têm um nome que consiste em mais de uma palavra, como Avenida Princesa Isabel em Recife, PE. Portanto, depois de descobrir que estamos omitindo endereços que têm essa forma, poderemos revisar nossa descrição de nomes de ruas para

```
'[A-Z][a-z]*(  [A-Z][a-z]*)*'
```

A expressão anterior começa com um grupo que consiste em uma letra maiúscula e zero ou mais letras minúsculas. Seguem-se zero ou mais grupos que consistem em um branco, outra letra maiúscula e zero ou mais letras minúsculas. O branco é um caractere comum em expressões do UNIX mas, para evitar que a expressão anterior se assemelhe a duas expressões separadas por

um branco em uma linha de comandos do UNIX, somos obrigados a incluir sinais apóstrofos em torno da expressão inteira. Os apóstrofos não fazem parte da expressão em si.

Precisamos agora incluir o número da casa como parte do endereço. A maioria dos números de casas é um string de dígitos. Porém, alguns terão uma letra em seguida, como em "Av. Brasil 123A". Desse modo, a expressão que usamos para números tem uma letra maiúscula opcional no fim: `[0-9]+[A-Z]?`. Note que usamos o operador + do UNIX para indicar "um ou mais" dígitos e o operador ? para indicar "zero ou uma" letra maiúscula. A expressão inteira que desenvolvemos para endereços residenciais é:

```
'(Rua|R\.|Avenida|Av\.|Estrada|Estr\.)
[A-Z][a-z]*(  [A-Z][a-z]*)* [0-9]+[A-Z]? '
```

Se trabalharmos com essa expressão, nós o faremos muito bem. Contudo, poderemos eventualmente descobrir que estamos deixando de considerar:

1. Endereços representados por algo diferente de uma rua, avenida ou estrada. Por exemplo, deixaremos de reconhecer "Boulevard", "Praça", "Beco" e suas abreviaturas.
2. Nomes de ruas que são numéricos ou parcialmente numéricos, como "Rua 42".
3. Caixas postais e roteiros de entrega rural.
4. Nomes de ruas que não têm nenhuma relação com algo como "Rua". Um exemplo é Trevo do Cunha, em Porto Alegre, ou ainda Ladeira do Segredo, no Rio de Janeiro.
5. Todos os tipos de itens estranhos que sequer podemos imaginar.

Desse modo, um compilador de expressões regulares pode tornar muito mais fácil o processo de lenta convergência para o identificador completo de endereços do que se tivéssemos de recodificar cada mudança diretamente em uma linguagem de programação convencional.

3.3.4 Exercícios para a Seção 3.3

! **Exercício 3.3.1**: Forneça uma expressão regular para descrever números de telefones em todas as diversas formas que você possa imaginar. Considere números internacionais, bem como o fato de diferentes países terem números de dígitos diferentes em códigos regionais e em números de telefones locais.

!! **Exercício 3.3.2**: Forneça uma expressão regular para representar salários como eles aparecem em anúncios de empregos. Considere que salários podem ser pa-

gos por hora, semana, mês ou ano. Eles podem ou não aparecer com um símbolo de cifrão ou ainda com outra unidade como "mil" em lugar de três zeros. Talvez haja uma palavra ou palavras próximas que identifiquem um salário. *Sugestão*: Examine os anúncios classificados em um jornal ou as listagens de ofertas de empregos on-line para ter uma idéia dos padrões que poderiam ser úteis.

! **Exercício 3.3.3**: No final da Seção 3.3.3, apresentamos alguns exemplos de melhorias que seriam possíveis para a expressão regular que descreve endereços. Modifique a expressão desenvolvida naquela seção para incluir todas as opções mencionadas.

3.4 Leis algébricas para expressões regulares

No Exemplo 3.5, vimos a necessidade de simplificar expressões regulares, a fim de manter administrável o tamanho das expressões. Naquele exemplo, fornecemos alguns argumentos *ad hoc* para explicar por que uma expressão poderia ser substituída por outra. Em todos os casos, a questão básica era que as duas expressões fossem *equivalentes*, no sentido de que elas definissem as mesmas linguagens. Nesta seção, ofereceremos uma coleção de leis algébricas que levam a um nível mais alto a questão de quando duas expressões regulares são equivalentes. Em vez de examinar expressões regulares específicas, consideraremos pares de expressões regulares tendo variáveis como argumentos. Duas expressões com variáveis são *equivalentes* se quaisquer que sejam as linguagens usadas para substituir as variáveis, os resultados das duas expressões são a mesma linguagem.

Um exemplo desse processo na álgebra da aritmética é dado a seguir. É comum dizer que $1 + 2 = 2 + 1$. Esse é um exemplo da lei comutativa da adição, e é fácil verificá-lo pela aplicação do operador de adição em ambos os lados, obtendo-se $3 = 3$. Porém, a *lei comutativa da adição* diz mais que isso; ela afirma que $x + y = y + x$, onde x e y são variáveis que podem ser substituídas por dois números quaisquer. Isto é, não importa quais são os dois números que somamos, obtemos o mesmo resultado independente da ordem em que efetuamos a soma.

Como as expressões aritméticas, as expressões regulares têm várias leis que as definem. Muitas dessas leis são semelhantes às leis da aritmética, se pensarmos na união como adição e na concatenação como multiplicação. No entanto, existem alguns lugares em que a analogia falha, e também existem algumas leis que se aplicam a expressões regulares mas não têm nenhuma analogia na aritmética, em especial quando se trata do operador de fechamento. As próximas seções formam um catálogo das leis mais importantes. Concluímos com uma discussão de como é possível verificar se uma lei proposta para expressões regulares é realmente uma lei; isto é, se ela será válida para todas as linguagens que possamos usar em substituição às variáveis.

3.4.1 Associatividade e comutatividade

A *comutatividade* é a propriedade de um operador que nos diz que podemos trocar a ordem de seus operandos e obter o mesmo resultado. Um exemplo para a aritmética foi dado anteriormente: $x + y = y + x$. A *associatividade* é a propriedade de um operador que nos permite reagrupar os operandos quando o operador é aplicado duas vezes. Por exemplo, a lei associativa da multiplicação é $(x \times y) \times z = x \times (y \times z)$. Aqui estão três leis desses tipos que são válidas para expressões regulares:

- $L + M = M + L$. Essa lei, a *lei comutativa para a união*, nos diz que podemos considerar a união de duas linguagens em qualquer ordem.
- $(L + M) + N = L + (M + N)$. Essa lei, a *lei associativa para a união*, nos diz que podemos considerar a união de três linguagens tomando inicialmente a união das duas primeiras ou tomando inicialmente a união das duas últimas. Observe que, junto com a lei comutativa para a união, concluímos que podemos tomar a união de qualquer coleção de linguagens com qualquer ordem e agrupamento, e o resultado será o mesmo. Intuitivamente, um string pertence a $L_1 \cup L_2 \cup \ldots \cup L_k$ se e somente se ele pertence a uma ou mais das linguagens L_i.
- $(LM)N = L(MN)$. Essa lei, a *lei associativa para a concatenação*, nos diz que podemos concatenar três linguagens concatenando inicialmente as duas primeiras ou as duas últimas.

Está faltando nessa lista a "lei" $LM = ML$, que nos diria que a concatenação é comutativa. Porém, essa lei é falsa.

Exemplo 3.10: Considere as expressões regulares **01** e **10**. Essas expressões denotam as linguagens {01} e {10}, respectivamente. Tendo em vista que as linguagens são diferentes, a lei geral $LM = ML$ não pode ser válida. Se fosse, poderíamos substituir L pela expressão regular **0** e M por **1** e concluiríamos falsamente que **01** = **10**. □

3.4.2 Identidades e aniquiladores

Uma *identidade* (ou elemento neutro) para um operador é um valor tal que, quando o operador é aplicado à identidade e a algum outro valor, o resultado é o outro valor. Por exemplo, 0 é a identidade para a adição, pois $0 + x = x + 0 = x$, e 1 é a identidade para a multiplicação, pois $1 \times x = x \times 1 = x$. Um *aniquilador* (ou elemento absorvente) para um operador é um valor tal que, quando o operador é aplicado ao aniquilador e a algum outro valor, o resultado é o aniquilador. Por exemplo, 0 é um aniquilador para a multiplicação, pois $0 \times x = x \times 0 = 0$. Não existe nenhum aniquilador para a adição.

Há três leis para expressões regulares envolvendo esses conceitos; essas leis estão listadas a seguir.

- $\emptyset + L = L + \emptyset = L$. Essa lei afirma que $\emptyset$ é a identidade para a união.
- $\varepsilon L = L\varepsilon = L$. Essa lei afirma que ε é a identidade para a concatenação.
- $\emptyset L = L\emptyset = \emptyset$. Essa lei afirma que $\emptyset$ é o aniquilador para a concatenação.

Essas leis são ferramentas poderosas em simplificações. Por exemplo, se tivermos uma união de várias expressões, algumas das quais são ou foram simplificadas até $\emptyset$, então os símbolos de $\emptyset$ podem ser descartados da união. Da mesma forma, se tivermos uma concatenação de várias expressões, algumas das quais são ou foram simplificadas até ε, poderemos retirar os símbolos ε da concatenação. Finalmente, se tivermos uma concatenação de qualquer número de expressões, e se uma delas for $\emptyset$, a concatenação inteira poderá ser substituída por $\emptyset$.

3.4.3 Leis distributivas

Uma *lei distributiva* envolve dois operadores, e afirma que um operador pode ser empurrado para dentro até ser aplicado a cada argumento do outro operador individualmente. O exemplo mais comum da aritmética é a lei distributiva da multiplicação sobre a adição, ou seja, $x \times (y + z) = x \times y + x \times z$. Tendo em vista que a multiplicação é comutativa, não importa se a multiplicação está à esquerda ou à direita da soma. Porém, existe uma lei análoga para expressões regulares, que temos de enunciar em duas formas, pois a concatenação não é comutativa. Essas leis são:

- $L(M + N) = LM + LN$. Essa lei, é a *lei distributiva à esquerda da concatenação sobre a união*.
- $(M + N)L = ML + NL$. Essa lei, é a *lei distributiva à direita da concatenação sobre a união*.

Vamos provar a lei distributiva à esquerda; a outra é provada de modo semelhante. A prova fará referência apenas a linguagens; ela não depende das linguagens terem expressões regulares.

Teorema 3.11: Se L, M e N são quaisquer linguagens, então

$$L(M \cup N) = LM \cup LN$$

PROVA: A prova é semelhante à outra prova sobre a lei distributiva que vimos no Teorema 1.10. Primeiro, precisamos mostrar que um string w está em $L(M \cup N)$ se e somente se ele está em $LM \cup LN$.

(Somente se) Se w está em $L(M \cup N)$, então $w = xy$, onde x está em L e y está em M ou N. Se y está em M, então xy está em LM e portanto em $LM \cup LN$. Da mesma forma, se y está em N, então xy está em LN e portanto em $LM \cup LN$.

(Se) Suponha que w está em $LM \cup LN$. Então, w está em LM ou em LN. Primeiro, suponha que w está em LM. Então, $w = xy$, onde x está em L e y está em M. Como y está em M, ele também está em $M \cup N$. Desse modo, xy está em $L(M \cup N)$. Se w não está em LM, certamente está em LN, e um argumento semelhante mostra que ele está em $L(M \cup N)$. □

Exemplo 3.12: Considere a expressão regular $0 + 01^*$. Podemos "fatorar um 0" da união, mas primeiro temos de reconhecer que a expressão 0 sozinha é na realidade a concatenação de 0 com algo, especificamente ε. Isto é, usamos a lei de identidade para a concatenação com o objetivo de substituir 0 por 0ε, o que nos dá a expressão $0\varepsilon + 01^*$. Agora, podemos aplicar a lei distributiva à esquerda para substituir essa expressão por $0(\varepsilon + 1^*)$. Se reconhecermos que ε está em $L(1^*)$, então $\varepsilon + 1^* = 1^*$, e podemos simplificar a expressão até 01^*. □

3.4.4 A lei de idempotência

Um operador é dito *idempotente* se o resultado da aplicação de dois valores iguais como argumentos é esse mesmo valor. Os operadores aritméticos comuns não são idempotentes; $x + x \neq x$ em geral e $x \times x \neq x$ em geral (embora existam *alguns* valores de x para os quais a igualdade é válida, como $0 + 0 = 0$). Porém, a união e a interseção são exemplos comuns de operadores idempotentes. Desse modo, para expressões regulares, podemos enunciar a seguinte lei:

- $L + L = L$. Essa lei, a *lei de idempotência para a união*, afirma que, se tomarmos a união de duas expressões idênticas, poderemos substituí-las por uma única cópia da expressão.

3.4.5 Leis envolvendo fechamentos

Existem várias leis envolvendo os operadores de fechamento e suas variantes + e ? no estilo do UNIX. Vamos listá-las aqui e dar algumas explicações para mostrar por que elas são verdadeiras.

- $(L^*)^* = L^*$. Essa lei nos diz que fechar uma expressão que já está fechada não muda a linguagem. A linguagem de $(L^*)^*$ são todos os strings criados por concatenação de strings pertencentes à linguagem de L^*. Contudo, esses strings são eles próprios compostos de strings de L. Desse modo, o string em $(L^*)^*$ também é uma concatenação de strings de L e portanto está na linguagem de L^*.

- $\emptyset^* = \varepsilon$. O fechamento de $\emptyset$ contém apenas o string ε, como vimos no Exemplo 3.6.
- $\varepsilon^* = \varepsilon$. É fácil verificar que o único string que pode ser formado pela concatenação de qualquer número de cópias do string vazio é o próprio string vazio.
- $L^+ = LL^* = L^*L$. Lembre-se de que L^+ é definido como $L + LL + LLL + \ldots$. Além disso, $L^* = \varepsilon + L + LL + LLL + \ldots$. Desse modo,

$$LL^* = L\varepsilon + LL + LLL + LLLL + \ldots$$

  Quando lembramos que $L\varepsilon = L$, vemos que as expansões infinitas para LL^* e para L^+ são iguais. Isso prova que $L^+ = LL^*$. A prova de que $L^+ = L^*L$ é semelhante.[4]
- $L^* = L^+ + \varepsilon$. A prova é fácil, pois a expansão de L^+ inclui todo termo na expansão de L^*, exceto ε. Observe que, se a linguagem L contém o string ε, então o termo adicional "$+\varepsilon$" não é necessário; ou seja, $L^+ = L^*$ nesse caso especial.
- $L? = \varepsilon + L$. Na realidade, essa regra é a definição do operador ?.

3.4.6 Descobrindo leis para expressões regulares

Cada uma das leis anteriores foi provada, formal ou informalmente. Porém, existe uma variedade infinita de leis sobre expressões regulares que poderiam ser propostas. Existe uma metodologia geral que facilite nossas provas das leis corretas? Ocorre que a verdade de uma lei se reduz a uma questão da igualdade de duas linguagens específicas. De maneira interessante, a técnica está intimamente ligada aos operadores de expressões regulares e não pode ser estendida a expressões que envolvem alguns outros operadores, como a interseção.

Para ver como esse teste funciona, vamos considerar uma lei proposta, como

$$(L + M)^* = (L^*M^*)^*$$

Essa lei nos diz que, se tivermos duas linguagens quaisquer L e M e fecharmos sua união, obteremos a mesma linguagem que obteríamos se tomássemos a linguagem L^*M^*, isto é, todos os strings compostos por zero ou mais escolhas de L seguidas por zero ou mais escolhas de M, e fechássemos essa linguagem.

---

[4]Note que, como conseqüência, qualquer linguagem L comuta (sob a concatenação) com seu próprio fechamento; $LL^* = L^*L$. Essa regra não contradiz o fato de que, em geral, a concatenação não é comutativa.

Para provar essa lei, primeiro suponha que o string w está na linguagem de $(L+M)^*$.[5] Podemos escrever $w = w_1 w_2 \ldots w_k$ para algum k, onde cada w_i está em L ou M. Segue-se que cada w_i está na linguagem de L^*M^*. Para ver por que, se w está em L, escolha um string, w_i de L; esse string também está em L^*. Não escolha nenhum string de M; isto é, escolha ε de M^*. Se w_i estiver em M, o argumento será semelhante. Uma vez que todo w_i está em L^*M^*, segue-se que w está no fechamento dessa linguagem.

Para completar a prova, também temos de provar a recíproca: que strings em $(L^*M^*)^*$ também estão em $(L+M)^*$. Omitimos essa parte da prova, pois nosso objetivo não é provar a lei, mas notar a seguinte propriedade importante de expressões regulares.

Qualquer expressão regular com variáveis pode ser imaginada como uma expressão regular *concreta*, que não tem nenhuma variável, imaginando cada variável como se ela fosse um símbolo distinto. Por exemplo, a expressão $(L + M)^*$ pode ter variáveis L e M substituídas por símbolos a e b, respectivamente, dando a expressão regular $(\mathbf{a} + \mathbf{b})^*$.

A linguagem da expressão concreta nos orienta em relação à forma de strings em qualquer linguagem que é formada a partir da expressão original, quando substituímos as variáveis por linguagens. Desse modo, em nossa análise de $(L+M)^*$, observamos que qualquer string w composto por uma seqüência de escolhas de L ou M, estaria na linguagem de $(L + M)^*$. Podemos chegar a essa conclusão observando a linguagem da expressão concreta, $L((\mathbf{a} + \mathbf{b})^*)$, que é evidentemente o conjunto de todos os strings de a's e b's. Poderíamos substituir qualquer ocorrência de a em um desses strings por qualquer string em L, e poderíamos substituir qualquer ocorrência de b por qualquer string em M, possivelmente com escolhas diferentes de strings para ocorrências diferentes de a ou b. Essas substituições, aplicadas a todos os strings em $(\mathbf{a} + \mathbf{b})^*$ nos dá todos os strings formados pela concatenação de strings de L e/ou M, em qualquer ordem.

A afirmação anterior pode parecer óbvia mas, como indicamos no quadro "Extensões do teste além das quais as expressões regulares podem falhar", ela não é nem mesmo verdadeira quando alguns outros operadores são adicionados aos três operadores de expressões regulares. Provaremos o princípio geral para expressões regulares no próximo teorema.

Teorema 3.13: Seja E uma expressão regular com variáveis $L_1, L_2, \ldots, L_m$. Forme uma expressão regular concreta C substituindo cada ocorrência de L_i pelo símbolo a_i, para $i = 1, 2, \ldots, m$. Então, para quaisquer linguagens $L_1, L_2, \ldots, L_m$, todo string w em $L(E)$ pode ser escrito como $w = w_1 w_2 \ldots w_k$, onde cada w_i está em uma

[5]Por simplicidade, identificaremos as expressões regulares e suas linguagens, e evitaremos dizer "a linguagem de" antes de toda expressão regular.

das linguagens, digamos L_{j_i}, e o string $a_{j_1} a_{j_2} \ldots a_{j_k}$ está na linguagem $L(C)$. Menos formalmente, podemos construir $L(E)$ começando com cada string em $L(C)$, digamos $a_{j_1} a_{j_2} \ldots a_{j_k}$, substituindo cada um dos a_{j_i}'s por qualquer string da linguagem correspondente L_{j_i}.

PROVA: A prova é uma indução estrutural sobre a expressão E.

BASE: Os casos base são aqueles em que E é ε, $\emptyset$ ou uma variável L. Nos dois primeiros casos, não existe nada para provar, pois a expressão concreta C é igual a E. Se E é uma variável L, então $L(E) = L$. A expressão concreta C é apenas a, onde a é o símbolo correspondendo a L. Desse modo, $L(C) = \{a\}$. Se substituirmos por qualquer string em L o símbolo a nesse único string, obteremos a linguagem L, que também é $L(E)$.

INDUÇÃO: Há três casos, dependendo do último operador de E. Primeiro, suponha que $E = F + G$; isto é, uma união é o último operador. Sejam C e D as expressões concretas formadas a partir de F e G, respectivamente, substituindo-se por símbolos concretos as variáveis de linguagens nessas expressões. Observe que o mesmo símbolo deve substituir todas as ocorrências da mesma variável, tanto em F quanto em G. Então, a expressão concreta que obtemos a partir de E é $C + D$, e $L(C + D) = L(C) + L(D)$.

Suponha que w seja um string em $L(E)$ quando as variáveis de linguagens de E são substituídas por linguagens específicas. Então w está em $L(F)$ ou em $L(G)$. Pela hipótese indutiva, w é obtido começando-se com um string concreto em $L(C)$ ou $L(D)$, respectivamente, substituindo-se os símbolos por strings nas linguagens correspondentes. Assim, em qualquer caso, o string w pode ser construído começando-se com um string concreto em $L(C + D)$, e fazendo-se as mesmas substituições de símbolos por strings.

Também devemos considerar os casos em que E é FG ou F^*. Entretanto, os argumentos são semelhantes ao caso da união anterior, e vamos deixá-los para o leitor. □

3.4.7 O teste de uma lei algébrica para expressões regulares

Agora, podemos enunciar e provar o teste para saber se uma lei para expressões regulares é ou não verdadeira. O teste para saber se $E = F$ é verdadeira, onde E e F são duas expressões regulares com o mesmo conjunto de variáveis, é:

1. Converter E e F nas expressões regulares concretas C e D, respectivamente, substituindo cada variável por um símbolo concreto.

2. Testar se $L(C) = L(D)$. Nesse caso, então $E = F$ é uma lei verdadeira e, se não, a "lei" é falsa. Observe que só veremos o teste para saber se duas expressões regulares denotam a mesma linguagem na Seção 4.4. Porém, podemos usar meios *ad hoc* para definir a igualdade dos pares de linguagens que realmente nos interessam. Lembre-se de que, se as linguagens *não* são iguais, é suficiente fornecer um contra-exemplo: um único string que esteja em uma linguagem, mas não na outra.

Teorema 3.14: O teste anterior identifica corretamente as leis verdadeiras para expressões regulares.

PROVA: Mostraremos que $L(E) = L(F)$ para quaisquer linguagens em lugar das variáveis de E e F, se e somente se $L(C) = L(D)$.

(Somente-se) Suponha que $L(E) = L(F)$ para todas as escolhas de linguagens em substituição às variáveis. Em particular, escolha para cada variável L o símbolo concreto a que substitui L nas expressões C e D. Então, para essa escolha, $L(C) = L(E)$ e $L(D) = L(F)$. Tendo em vista que $L(E) = L(F)$ é dada, segue-se que $L(C) = L(D)$.

(Se) Suponha que $L(C) = L(D)$. Pelo Teorema 3.13, $L(E)$ e $L(F)$ são construídas substituindo-se os símbolos concretos de strings em $L(C)$ e $L(D)$, respectivamente, por strings nas linguagens que correspondem a esses símbolos. Se os strings de $L(C)$ e $L(D)$ são iguais, então as duas linguagens construídas dessa maneira também serão iguais; isto é, $L(E) = L(F)$. □

Exemplo 3.15: Considere a provável lei $(L + M)^* = (L^*M^*)^*$. Se substituirmos as variáveis L e M por símbolos concretos a e b respectivamente, obteremos as expressões regulares $(\mathbf{a} + \mathbf{b})^*$ e $(\mathbf{a}^*\mathbf{b}^*)^*$. É fácil verificar que essas duas expressões denotam a linguagem com todos os strings de a's e b's. Desse modo, as duas expressões concretas denotam a mesma linguagem, e a lei é válida. □

Como outro exemplo de uma lei, considere $L^* = L^*L^*$. As linguagens concretas são $\mathbf{a}^*$ e $\mathbf{a}^*\mathbf{a}^*$, respectivamente, e cada um deles é o conjunto de todos os strings de valores a. Novamente, a lei é valida; isto é, a concatenação de uma linguagem fechada com ela própria produz essa mesma linguagem.

Por fim, considere a provável lei $L + ML = (L + M)L$. Se escolhermos os símbolos a e b para as variáveis L e M, respectivamente, teremos as duas expressões regulares concretas $\mathbf{a} + \mathbf{ba}$ e $(\mathbf{a} + \mathbf{b})\mathbf{a}$. Porém, as linguagens dessas expressões não são iguais. Por exemplo, o string aa está na segunda, mas não na primeira. Desse modo, esta lei é falsa. □

3.4.8 Exercícios para a Seção 3.4

Exercício 3.4.1: Verifique as identidades a seguir que envolvem expressões regulares.

* a) $R + S = S + R$.

b) $(R + S) + T = R + (S + T)$.

c) $(RS)T = R(ST)$.

d) $R(S + T) = RS + RT$.

e) $(R + S)T = RT + ST$.

* f) $(R^*)^* = R^*$.

g) $(\varepsilon + R)^* = R^*$.

h) $(R^*S^*)^* = (R + S)^*$.

Extensões do teste além das expressões regulares podem falhar

Vamos considerar uma álgebra estendida de expressões regulares que inclui o operador de interseção. De modo interessante, a adição de $\cap$ aos três operadores de expressões regulares não aumenta o conjunto de linguagens que podemos descrever, como veremos no Teorema 4.8. No entanto, ele torna inválido o teste de leis algébricas.

Considere a "lei" $L \cap M \cap N = L \cap M$; ou seja, a interseção de três linguagens quaisquer é igual à interseção das duas primeiras linguagens. Essa "lei" é evidentemente falsa. Por exemplo, sejam $L = M = \{a\}$ e $N = \emptyset$. Porém, o teste baseado na concretização das variáveis deixaria de ver a diferença. Isto é, se substituíssemos L, M e N pelos símbolos a, b e c, respectivamente, testaríamos se $\{a\}\} \cap \{b\} \cap \{c\} = \{a\} \cap \{b\}$. Tendo em vista que ambos os lados são o conjunto vazio, a igualdade de linguagens é verdadeira e o teste implicaria que a "lei" é verdadeira.

! **Exercício 3.4.2**: Prove ou refute cada uma das afirmações a seguir sobre expressões regulares.

* a) $(R + S)^* = R^* + S^*$.

b) $(RS + R)^*R = R(SR + R)^*$.

* c) $(RS + R)^*RS = (RR^*S)^*$.

d) $(R+S)^*S = (R^*S)^*$.

e) $S(RS + S)^*R = RR^*S(RR^*S)^*$.

Exercício 3.4.3: No Exemplo 3.6, desenvolvemos a expressão regular

$$(0 + 1)^*1(0 + 1) + (0 + 1)^*1(0 + 1)\ (0 + 1)$$

Use as leis distributivas para desenvolver duas expressões diferentes, mais simples e equivalentes.

Exercício 3.4.4: No início da Seção 3.4.6, apresentamos parte de uma prova de que $(L^*M^*)^* = (L + M)^*$. Complete a prova, mostrando que strings em $(L^*M^*)^*$ também estão em $(L + M)^*$.

! **Exercício 3.4.5**: Complete a prova do Teorema 3.13, tratando os casos em que a expressão regular E é da forma FG ou da forma F^*.

3.5 Resumo do Capítulo 3

- *Expressões regulares*: Essa notação algébrica descreve exatamente as mesmas linguagens que os autômatos finitos: as linguagens regulares. Os operadores de expressões regulares são união, concatenação (ou "ponto") e fechamento (ou "estrela").
- *Expressões regulares na prática*: Sistemas como UNIX e vários de seus comandos usam uma linguagem estendida de expressões regulares que fornece abreviaturas para muitas expressões comuns. As classes de caracteres permitem a expressão fácil de conjuntos de símbolos, enquanto operadores como um-ou-mais-de e no máximo-um-de ampliam os operadores habitualmente usados em expressões regulares.
- *Equivalência de expressões regulares e autômatos finitos*: Podemos converter um DFA em uma expressão regular por meio de uma construção indutiva na qual as expressões correspondentes aos rótulos de caminhos que podem passar por conjuntos cada vez maiores de estados são construídos. De modo alternativo, podemos usar um procedimento de eliminação de estados para elaborar a expressão regular correspondente a um DFA. No outro sentido, podemos construir recursivamente um ε-NFA a partir de expressões regulares, e depois converter o ε-NFA em um DFA, se desejarmos.
- *A álgebra de expressões regulares*: As expressões regulares obedecem a muitas leis algébricas da aritmética, embora existam diferenças. A união e a concatenação são associativas, mas só a união é comutativa. A concatenação se distribui sobre a união. A união é idempotente.
- *Testando identidades algébricas*: Podemos saber se uma equivalência de expressões regulares envolvendo variáveis como argumentos é verdadeira substituindo as variáveis por constantes distintas e testando se as linguagens resultantes são iguais.