Parte Um:
Autômatos e Linguagens

Capítulo 1

Linguagens Regulares

A teoria da computação começa com uma pergunta: O que é um computador? É talvez uma pergunta boba, pois mesmo minha filha de 4 anos de idade sabe que essa coisa sobre a qual estou teclando é um computador. Mas esses computadores reais são bastante complicados—demasiado a ponto de nos permitir estabelecer uma teoria matemática manuseável sobre eles diretamente. Ao invés, usamos um computador idealizado chamado um ***modelo computacional***. Como com qualquer modelo em ciência, um modelo computacional pode ser preciso de algumas formas mas talvez não em outras. Portanto usaremos vários modelos computacionais diferentes, dependendo das características sobre as quais desejamos focar. Começamos com o modelo mais simples, chamado ***máquina de estados finitos*** ou ***autômato finito***.

1.1 Autômatos finitos

Autômatos finitos são bons modelos para computadores com uma quantidade de memória extremamente limitada. O que pode um computador fazer com uma memória tão pequena? Muitas coisas úteis! Na verdade, interagimos com tais computadores o tempo todo, pois eles residem no coração de vários dispositivos eletromecânicos.

Como mostrado nas figuras a seguir, o controlador para uma porta automática é um exemplo de tal dispositivo. Frequentemente encontradas em entradas e saídas de supermercados, portas automáticas abrem deslizando quando uma pessoa está se aproximando. Uma porta automática tem um tapete na frente para detectar a presença de uma pessoa que está próxima a atravessar a passagem. Um outro tapete está localizado atrás da passagem de modo que o controlador pode manter a porta aberta um tempo suficiente para que a pessoa atravesse toda a passagem e também de modo que a porta não atinja alguém que está atrás no momento que ela abre.

O controlador está em um dos dois estados: “ABERTA” ou “FECHADA,” representando a condição correspondente da porta. Como mostrado nas figuras a seguir, existem quatro condições possíveis: “FRENTE” (significando que uma pessoa está pisando no tapete na frente da passagem), “ATRÁS” (significando que uma pessoa está

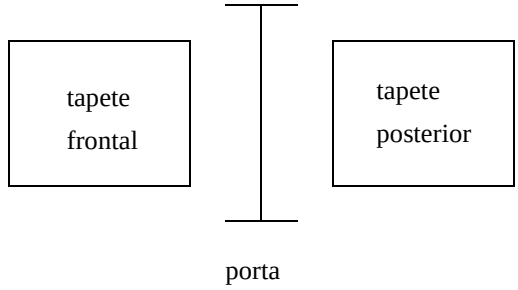


Figura 1.1: Vista superior de uma porta automática

pisando no tapete após a passagem), “AMBOS” (significando que as pessoas estão pisando em ambos os tapetes), e “NENHUM” (significando que ninguém está pisando em qualquer dos tapetes.

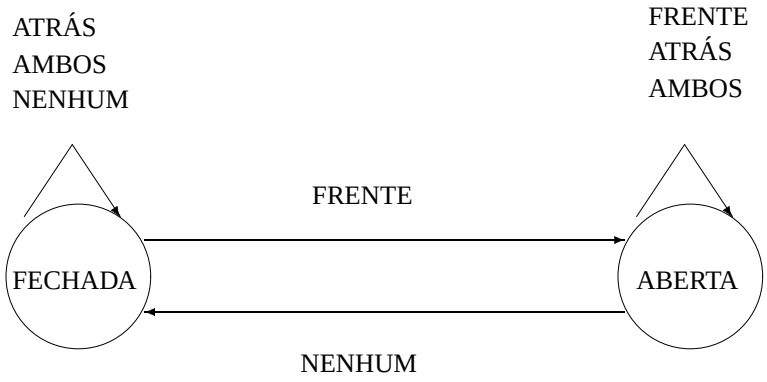


Figura 1.2: Diagrama de estados para o controlador de porta automática

		NENHUM	FRENTE	ATRÁS	AMBOS
estado	FECHADA	FECHADA	ABERTA	FECHADA	FECHADA
	ABERTA	FECHADA	ABERTA	ABERTA	ABERTA

Figura 1.3: Tabela de transição de estado para o controlador de porta automática

O controlador move de estado para estado, dependendo da entrada que ele recebe. Quando no estado “FECHADA” e recebendo uma entrada NENHUM ou ATRÁS, ele permanece no estado FECHADA. Adicionalmente, se a entrada AMBOS é recebida, ele permanece FECHADA porque a porta corre o risco de atingir alguém sobre o tapete de trás. Mas se a entrada FRENTE chega, ele move para o estado ABERTA. No estado ABERTA, se a entrada FRENTE, ATRÁS, ou AMBOS é recebida, ele permanece em ABERTA. Se a entrada NENHUM chega, ele retorna a FECHADA.

Por exemplo, um controlador pode começar no estado FECHADA e receber a seguinte série de sinais de entrada: FRENTE, ATRÁS, NENHUM, FRENTE, AMBOS, NENHUM, ATRÁS, NENHUM. Ele então passaria pela série de estados: FE-

CHADA (começando), ABERTA, ABERTA, FECHADA, ABERTA, ABERTA, FECHADA,
FECHADA, FECHADA.

Pensando num controlador de porta automática como um autômato finito, é útil porque sugere formas padronizadas de representação como nas Figuras 1.2 e 1.3. Esse controlador é um computador que tem somente um bit de memória, capaz de gravar em quais dos dois estados o controlador está. Outros dispositivos comuns têm controladores com memórias algo maiores. Em um controlador de elevador um estado pode representar o andar no qual o elevador está e as entradas pode ser os sinais recebidos dos botões. Esse computador pode precisar de vários bits para guardar essa informação. Controladores para vários dispositivos domésticos tais como lavadoras de prato e termostatos eletrônicos, assim como peças de relógios digitais e calculadoras, são exemplos adicionais de computadores com memórias limitadas. O desenho de tais dispositivos requer que se mantenha em mente a metodologia e a terminologia de autômatos finitos.

Autômatos finitos e suas contrapartidas probabilísticas ***cadeias de Markov*** são ferramentas úteis quando estamos tentando reconhecer padrões em dados. Esses dispositivos são usados em processamento de voz e em reconhecimento de caracteres óticos. Cadeias de Markov têm sido usadas para modelar e prever mudanças de preço em mercados financeiros.

Agora vamos dar uma olhada mais próxima em autômatos finitos sob uma perspectiva matemática. Desenvolveremos uma definição precisa de um autômato finito, terminologia para descrever e manipular autômatos finitos, e resultados teóricos que descrevem seu poder e suas limitações. Além de nos dar um entendimento mais claro do que são autômatos finitos, e do que eles podem e não podem fazer, o desenvolvimento teórico nos permite praticar e nos tornar mais confortáveis com definições matemáticas, teoremas, e provas em um cenário relativamente simples.

Ao começar a descrever a teoria matemática de autômatos finitos, fazemos isso no nível abstrato, sem referência a qualquer aplicação específica. A seguinte figura mostra um autômato finito chamado M_1.

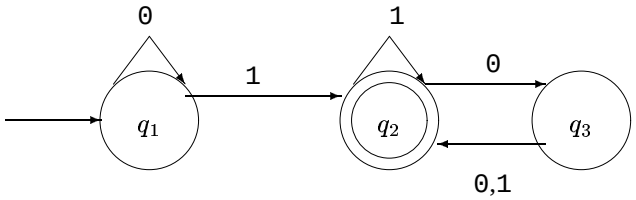


Figura 1.4: Um autômato finito chamado M_1 que tem três estados

A Figura 1.4 é chamada ***diagrama de estado*** de M_1. O autômato tem três ***estados***, rotulados q_1, q_2 e q_3. O ***estado inicial***, q_1, é indicado pela seta apontando para ele a partir do nada. O ***estado de aceitação***, q_2, é aquele com um duplo círculo. As setas saindo de um estado para outro são chamadas ***transições***.

Quando esse autômato recebe uma cadeia de entrada tal como `1101`, ele processa essa cadeia e produz uma saída. A saída é ***aceita*** ou ***rejeita***. Consideraremos apenas esse tipo de saída sim/não por enquanto de modo a manter as coisas simples. O processamento começa no estado inicial de M_1. O autômato recebe os símbolos da cadeia

de entrada um por um da esquerda para a direita. Após ler cada símbolo, M_1 move de um estado para outro por meio da transição que tem aquele símbolo como seu rótulo. Quando ele lê o último símbolo, M_1 produz sua saída. A saída é *aceita* se M_1 está agora em um estado de aceitação e *rejeita* se não está.

Por exemplo, quando alimentamos a cadeia de entrada `1101` na máquina M_1 da Figura 1.4, o processamento procede da seguinte forma:

1. começa no estado q_1;
2. lê `1`, segue transição de q_1 para q_2;
3. lê `1`, segue transição de q_2 para q_2;
4. lê `0`, segue transição de q_2 para q_3;
5. lê `1`, segue transição de q_3 para q_2;
6. *aceita* porque M_1 está em um estado de aceitação q_2 no final da entrada.

Experimentando com essa máquina sobre uma variedade de cadeias de entrada revela que ela aceita as cadeias `1`, `01`, `11`, e `0101010101`. Na verdade, M_1 aceita qualquer cadeia que termina com um `1`, pois ela vai para seu estado de aceitação q_2 sempre que ela lê o símbolo `1`. Adicionalmente, ela aceita as cadeias `100`, `0100`, `110000`, e `0101000000`, e qualquer cadeia que termine com um número par de `0`'s após o último `1`. Ela rejeita outras cadeias, tais como `0`, `10`, `101000`. Você pode descrever a linguagem consistindo de todas as cadeias que M_1 aceita? Faremos isso brevemente.

Definição formal de um autômato finito

Na seção precedente usamos diagramas de estado para introduzir autômatos finitos. Agora definimos autômatos finitos formalmente. Embora diagramas de estado sejam mais fáceis de compreender intuitivamente, precisamos de uma definição formal também, por duas razões específicas.

Primeiro, uma definição formal é precisa. Ela resolve quaisquer incertezas sobre o que é permitido num autômato finito. Se você estivesse incerto sobre se autômatos finitos pudessem ter 0 estados de aceitação ou se eles têm que ter exatamente uma transição saindo de todo estado para cada símbolo de entrada possível, você poderia consultar a definição formal e verificar que a resposta é sim em ambos os casos. Segundo, uma definição formal provê notação. Boa notação ajuda a você pensar e expressar seus pensamentos claramente.

A linguagem de uma definição formal é um tanto misteriosa, tendo alguma semelhança com a linguagem de um documento legal. Ambos necessitam ser precisos, e todo detalhe deve ser explicitado.

Um autômato finito tem várias partes. Tem um conjunto de estados e regras para ir de um estado para outro, dependendo do símbolo de entrada. Tem um alfabeto de entrada que indica os símbolos de entrada permitidos. Tem um estado inicial e um conjunto de estados de aceitação. A definição formal diz que um autômato finito é uma lista daqueles cinco objetos: conjunto de estados, alfabeto de entrada, regras para movimentação, estado inicial, e estados de aceitação. Em linguagem matemática uma lista de cinco elementos é frequentemente chamada 5-upla.

Usamos algo chamado de uma ***função de transição***, frequentemente denotado por δ, para definir as regras para movimentação. Se o autômato finito tem uma seta de um

estado x para um estado y rotulada com o símbolo de entrada `1`, isso significa que, se o autômato está no estado x quando ele lê um `1`, ele então move para o estado y. Podemos indicar a mesma coisa com a função de transição dizendo que $\delta(x, \mathtt{1}) = y$. Essa notação é uma espécie de abreviação matemática. Juntando tudo chegamos na definição formal de autômatos finitos.

Definição 1.1 . . . . . .
Um ***autômato finito*** é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$ onde

1. Q é um conjunto finito chamado de ***os estados***,
2. Σ é um conjunto finito chamaddo de ***o alfabeto***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ é a ***função de transição***,[1]
4. $q_0 \in Q$ é o ***estado inicial***, e
5. $F \subseteq Q$ é o ***conjunto de estados de aceitação***.[2]

A definição formal descreve precisamente o que queremos dizer por um autômato finito. Por exemplo, voltando à questão anterior sobre se 0 estados de aceitação é permissível, você pode ver que fazendo F ser o conjunto vazio $\emptyset$ resulta em 0 estados de aceitação, o que é permissível. Além do mais a função de transição δ especifica exatamente um estado para cada combinação possível de um estado e um símbolo de entrada. Isso responde à nossa outra questão afirmativamente, mostrando que exatamente uma seta de transição sai de todo estado para cada símbolo de entrada possível.

Podemos usar a notação da definição formal para descrever autômatos finitos individualmente especificando cada uma das partes listadas na Definição 1.1. Por exemplo, vamos retornar ao autômato finito M_1 na Figura 1.4.

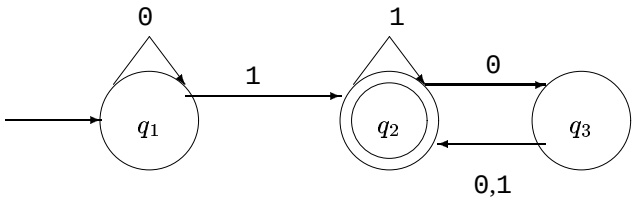


Figura 1.5: O autômato finito M_1

Podemos descrever M_1 formalmente escrevendo $M_1 = (Q, \Sigma, \delta, q_1, F)$, onde

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{\mathtt{0}, \mathtt{1}\}$,
3. δ é descrita como

[1] Remeta-se de volta à página 7 se você está incerto sobre o significado de $\delta : Q \times \Sigma \longrightarrow Q$.
[2] Estados de aceitação às vezes são chamados de ***estados finais***.

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 é o estado inicial, e
5. $F = \{q_2\}$.

Se A é o conjunto de todas as cadeias que a máquina M aceita, dizemos que A é a ***linguagem da máquina*** M e escrevemos $L(M) = A$. Dizemos que M ***reconhece*** A ou que M ***aceita*** A. Devido ao fato de que o termo *aceita* tem significados diferentes quando nos referimos a máquinas aceitando cadeias e máquinas aceitando linguagens, preferimos o termo *reconhece* para linguagens de modo a evitar confusão.

Uma máquina pode aceita diversas cadeias, mas ela sempre reconhece apenas uma linguagem. Se a máquina não aceita nenhuma cadeia, ela ainda reconhece uma linguagem, a saber a linguagem vazia $\emptyset$.

Em nosso exemplo, faça

$$A = \{w \mid w\text{contém pelo menos um } \texttt{1} \text{ e}$$

$$\text{um número par de } \texttt{0}\text{'s segue o último } \texttt{1}\}$$

Então $L(M_1) = A$, ou equivalentemente, M_1 reconhece A.

Exemplos de autômatos finitos

O diagrama abaixo é o diagrama de estados do autômato finito M_2.

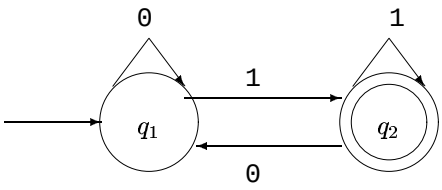


Figura 1.6: Diagrama de estados do autômato finito M_2 de dois estados

Na descrição formal $M_2 = (\{q_1, q_2\}, \{\texttt{0}, \texttt{1}\}, \delta, q_1, \{q_2\})$. A função de transição δ é

	0	1
q_1	q_1	q_2
q_2	q_1	q_2

Lembre-se que o diagrama de estados de M_2 e a descrição formal de M_2 contêm a mesma informação, apenas de forma diferente. Você pode sempre ir de uma para a outra se necessário.

Uma boa maneira de começar entendendo qualquer máquina é testá-la com algumas cadeias de entrada. Quando você faz esses "experimentos" para ver como a máquina está funcionando, seu método de funcionamento frequentemente se torna aparente. Sobre a cadeia de amostra `1101` a máquina M_2 começa no seu estado inicial q_1 e procede

primeiro para o estado q_2 após ler o primeiro `1`, e então para os estados q_2, q_1, e q_2 após ler `1`, `0`, e `1`. A cadeia é aceita porque o estado q_2 é um estado de aceitação. Mas a cadeia `110` deixa M_2 no estado q_1, portanto ela é rejeitada. Após tentar alguns poucos exemplos mais, você verá que M_2 aceita todas as cadeias que terminam com um `1`. Por conseguinte $L(M_2) = \{w \mid w \text{ termina com um } \texttt{1}\}$.

Exemplo 1.3 . . . . . .
Considere o autômato finito M_3.

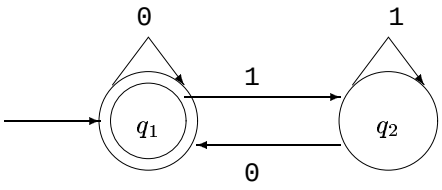


Figura 1.7: Diagrama de estados do autômato finito M_3 de dois estados

A máquina M_3 é semelhante a M_2 exceto pela localização do estado final. Como de costume, a máquina aceita todas as cadeias que a deixam em um estado de aceitação quando ela termina de ler. Note que, devido ao fato de que o estado inicial é também um estado de aceitação, M_3 aceita a cadeia vazia ε. Assim que a máquina começa a ler a cadeia vazia ela já está no fim, portanto se o estado inicial é um estado de aceitação, ε é aceita. Além da cadeia vazia, essa máquina aceita qualquer cadeia terminando com um `0`. Aqui,

$$L(M_3) = \{w \mid w \text{ é a cadeia vazia } \varepsilon \text{ ou termina com um } \texttt{0}\}.$$

Exemplo 1.4 . . . . . .
Considere o autômato finito M_4.

M_4 tem dois estados de aceitação, q_1 e r_1 e opera sobre o alfabeto $\Sigma = \{\texttt{a}, \texttt{b}\}$. Um pouco de experimentação mostra que ela aceita as cadeias `a`, `b`, `aa`, `bb`, e `bab`, mas não as cadeias `ab`, `ba`, ou `bbba`. Essa máquina começa no estado s, e depois que ela lê o primeiro símbolo na entrada, ela ou vai para a esquerda para os estados q ou para a direita para os estados r. Em qualquer dos casos ela nunca pode retornar ao estado inicial (em contraste com o caso anterior), pois ela não tem maneira de sair de qualquer outro estado e voltar para s. Se o primeiro símbolo na cadeia de entrada é `a`, então ela vai para a esquerda e aceita quando a cadeia termina com um `a`. Similarmente, se o primeiro símbolo é um `b`, a máquina vai para a direita e aceita quando a cadeia termina em `b`. Portanto M_4 aceita todas as cadeias que começam e terminam com `a`, ou que começam e terminam com `b`. Em outras palavras, M_4 aceita cadeias que começam e terminam com o mesmo símbolo.

Exemplo 1.5 . . . . . .
O diagrama de estados a seguir mostra a máquina M_5, que tem um alfabeto de entrada de quatro símbolos $\Sigma = \{\langle\text{ZERA}\rangle, \texttt{0}, \texttt{1}, \texttt{2}\}$. Tratamos $\langle\text{ZERA}\rangle$ como um único símbolo.

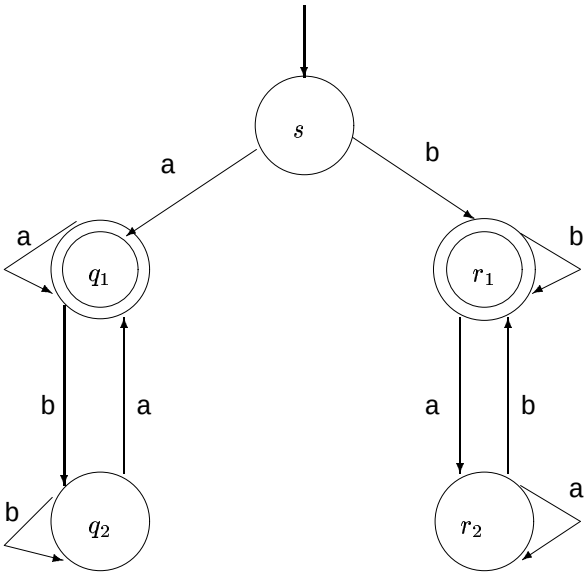


Figura 1.8: Autômato finito M_4

M_5 mantém um contador para a soma dos símbolos numéricos de entrada que ela lê, módulo 3. Toda vez que ela recebe o símbolo ⟨ZERA⟩ ela recoloca o contador em 0. Ela aceita se a soma é 0, módulo 3, ou em outras palavras, se a soma é um múltiplo de 3.

Exemplo 1.6 .....

Descrever um autômato finito por diagrama de estados não é possível em alguns casos. Isso pode ocorrer quando o diagrama seria grande demais para desenhar ou se, como nesse exemplo, a descrição depende de algum parâmetro não-especificado. Nesses casos recorremos a uma descrição formal para especificar a máquina.

Considere uma generalização do Exemplo 1.5 usando os mesmos quatro símbolos do alfabeto Σ. Para cada $i \geq 1$ seja A_i a linguagem de todas as cadeias onde a soma dos números é um múltiplo de i, exceto que a soma é zerada sempre que o símbolo ⟨ZERA⟩ aparece. Para cada A_i damos um autômato finito B_i reconhecendo A_i. Descrevemos a máquina B_i formalmente da seguinte maneira: $B_i = (Q_i, \Sigma, \delta_i, q_0, \{q_0\})$, onde Q_i é o conjunto de i estados $\{q_0, q_1, q_2, \ldots, q_{i-1}\}$, e montamos a função de transição δ_i de modo que para cada j, se B_i está em q_j, a soma corrente é j, módulo i. Para cada q_j faça

$\delta_i(q_j, \texttt{0}) = q_j$
$\delta_i(q_j, \texttt{1}) = q_k$ onde $k = j + 1$ módulo i,
$\delta_i(q_j, \texttt{2}) = q_k$ onde $k = j + 2$ módulo i, e
$\delta_i(q_j, \langle\text{ZERA}\rangle) = q_0$.

Definição formal de computação

Até agora descrevemos autômatos finitos informalmente, usando diagramas de estado, e com uma definição formal, como uma 5-upla. A descrição informal é mais fácil de compreender inicialmente, mas a definição formal é útil pois torna a noção totalmente

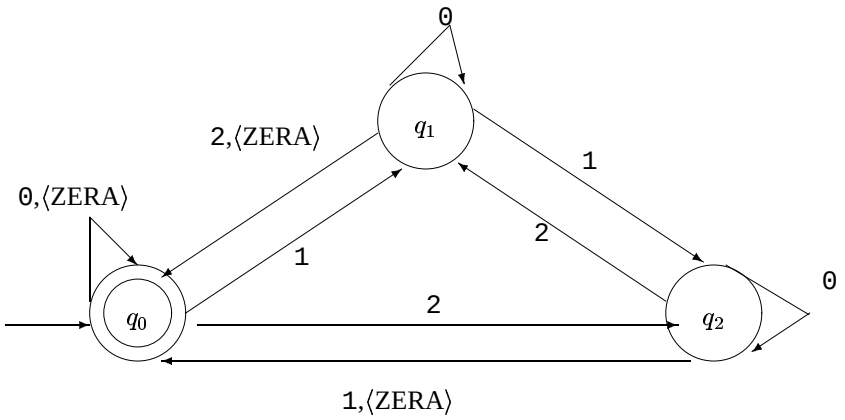


Figura 1.9: Autômato finito M_5

precisa, resolvendo quaisquer ambigüidades que podem ter ocorrido na descrição informal. A seguir fazemos o mesmo para uma computação de um autômato finito. Já temos uma idéia informal da maneira com que ele computa, e agora a formalizamos matematicamente.

Seja $M = (Q, \Sigma, \delta, q_0, F)$ um autômato finito e $w = w_1 w_2 \cdots w_n$ uma cadeia sobre o alfabeto Σ. Então M ***aceita*** w se uma seqüência de estados $r_0, r_1, \ldots, r_n$ existe em Q com as seguintes três condições:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ para $i = 0, \ldots, n-1$, e
3. $r_n \in F$.

Condição 1 diz que a máquina começa no estado inicial. Condição 2 diz que a máquina vai de estado para estado conforme a função de transição. Condição 3 diz que a máquina aceita sua entrada se ela termina em um estado de aceitação. Dizemos que M ***reconhece a linguagem*** A se $A = \{w \mid M \text{ aceita } w\}$.

Definição 1.7 . . . . . .
Uma linguagem é chamada de uma ***linguagem regular*** se algum autômato finito a reconhece.

Exemplo 1.8 . . . . . .
Tome a máquina M_5 do Exemplo 1.5. Seja w a cadeia

$$\texttt{10}\langle\text{ZERA}\rangle\texttt{22}\langle\text{ZERA}\rangle\texttt{012}$$

Então M_5 aceita w conforme a definição formal de computação porque a seqüência de estados em que ela entra quando computando sobre w é

$$q_0, q_1, q_1, q_0, q_2, q_1, q_0, q_0, q_1, q_0,$$

que satisfaz as três condições. A linguagem de M_5 é

$$L(M_5) = \{w \mid \begin{array}{l}\text{a soma dos símbolos em } w \text{ é 0 módulo 3,} \\ \text{exceto que } \langle\text{ZERA}\rangle \text{ zera o contador}\}.\end{array}$$

Como M_5 reconhece essa linguagem, ela é uma linguagem regular.

Projetando autômatos finitos

Seja de um autômato ou de um trabalho artístico, projetar é um processo criativo. Como tal ele não pode ser reduzido a uma simples receita ou fórmula. Entretanto, você poderia achar uma determinada abordagem benéfica quando projetando vários tipos de autômatos. Isto é, ponha-*se* no lugar da máquina que você está tentando projetar e então veja como você iria realizar a tarefa da máquina. Fazendo de conta que você é a máquina é um truque psicológico que ajuda a engajar sua mente inteira no processo de projetar.

Vamos projetar um autômato finito usando o método do "leitor como autômato" que acaba de ser descrito. Suponha que lhe é dada uma linguagem e você deseja projetar um autômato finito que a reconheça. Fazendo de conta ser o autômato, você recebe uma cadeia de entrada e tem que determinar se ela é um membro da linguagem que o autômato deve reconhecer. Você vê os símbolos na cadeia um por um. Após cada símbolo você tem que decidir se a cadeia vista até então está na linguagem. A razão é que você, como a máquina, não sabe quando o final da cadeia está vindo, daí você tem que estar sempre pronto com a resposta.

Primeiro, de modo a tomar essas decisões, você tem que identificar o que você precisa memorizar sobre a cadeia à medida que você está lendo. Por que não simplesmente memorizar tudo o que você viu? Tenha em mente que você está fazendo de conta que é um autômato finito e que esse tipo de máquina tem apenas uma quantidade finita de estados, o que significa uma memória finita. Imagine que a entrada seja extremamente longa, digamos, daqui até a lua, de modo que você não poderia de forma alguma memorizar a coisa inteira. Você tem uma memória finita, digamos, uma única folha de papel, que tem uma capacidade de memória limitada. Felizmente, para muitas linguagens você não precisa memorizar a entrada inteira. Você somente necessita de memorizar uma certa informação crucial. Exatamente que informação é crucial depende da linguagem específica considerada.

Por exemplo, considere que o alfabeto é $\{\mathtt{0}, \mathtt{1}\}$ e que a linguagem consiste de todas as cadeias com um número ímpar de `1`'s. Você deseja construir um autômato finito E_1 para reconhecer essa linguagem. Fazendo de conta que é o autômato, você começa obtendo uma cadeia de entrada de `0`'s e `1`'s símbolo por símbolo. Você precisa memorizar a cadeia inteira vista até então de modo a determinar se o número de `1`'s é ímpar? Claro que não. Simplesmente memorize se o número de `1`'s vistos até então é par ou ímpar e mantenha um registro dessa informação à medida que você lê novos símbolos. Se você ler um `1`, inverta a resposta, mas se você ler um `0`, deixe a resposta do jeito que está.

Mas como isso ajuda a você projetar E_1? Uma vez que você determinou a informação necessária para memorizar a cadeia como ela está sendo lida, você representa essa informação como uma lista finita de possibilidades. Nessa instância, as possibilidades seriam

1. par até agora, e
2. ímpar até agora.

Então você associa um estado a cada uma das possibilidades. Esses são os estados de E_1, como mostrado na figura seguinte.

A seguir, você associa as transições vendo como ir de uma possibilidade para outra ao ler um símbolo. Portanto, se o estado q_{par} representa a possibilidade par e o estado

Figura 1.10: Os dois estados q_{par} e q_{impar}

q_{impar} respresenta a possibilidade ímpar, você faria com que as transições trocassem de estado sobre um `1` e permanecessem onde estavam sobre um `0`, como mostrado na figura seguinte.

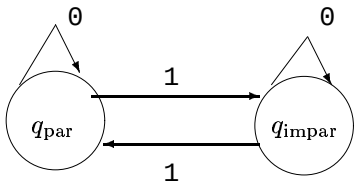


Figura 1.11: Transições dizendo como as possibilidades se rearranjam

A seguir, você marca como estado inicial o estado correspondente à possibilidade associada com ter visto 0 símbolos até então (a cadeia vazia ε). Neste caso o estado inicial corresponde ao estado q_{par} porque 0 é um número par. Por último, marque os estados de aceitação como sendo aqueles correspondentes às possibilidades nas quais você deseja aceitar a cadeia de entrada. Marque q_{impar} como sendo um estado de aceitação porque você deseja aceitar quando você tiver visto um número ímpar de `1`'s. Essas adições são mostradas na figura a seguir.

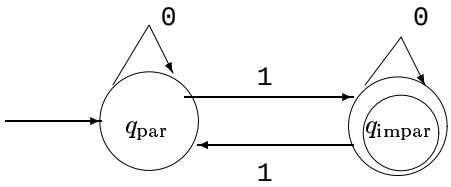


Figura 1.12: Adicionando os estados inicial e de aceitação

Exemplo 1.9 . . . . . .
Esse exemplo mostra como desenhar um autômato finito E_2 para reconhecer a linguagem regular de todas as cadeias que contêm a cadeia `001` como uma subcadeia. Por exemplo, `0010`, `1001`, `001`, e `11111110011111` estão todas na linguagem, mas `11` e `0000` não estão. Como você reconheceria essa linguagem se você fosse fazer de conta que era E_2? À medida que símbolos chegam, você inicialmente pularia sobre todos os `1`'s. Se você chegasse a um `0`, então você anotaria que você pode ter acabado de ver o primeiro dos três símbolos no padrão `001` que você está buscando. Se nesse ponto você vê um `1`, houve poucos `0`'s, portanto você volta a pular sobre os `1`'s. Mas se você vê um `0` naquele ponto, você deve memorizar que você acabou de ver dois

símbolos do padrão. Agora você simplesmente precisa continuar a varrer até que você veja um `1`. Se você o encontrar, memorize que você foi bem sucedido em encontrar o padrão e continue a ler a cadeia de entrada até que você chegue ao fim.

Portanto existem quatro possibilidades: Você

1. não chegou a ver quaisquer símbolos do padrão,
2. acabou de ver um `0`,
3. acabou de ver `00`, ou
4. viu o padrão inteiro `001`.

Associe os estados q, q_0, q_{00}, e q_{001} a essas possibilidades. Você pode associar as transições observando que de q lendo um `1` você permanece em q, mas lendo um `0` você move para q_0. Em q_0 lendo um `1` você retorna a q, mas lendo um `0` você move para q_{00}. Em q_{00} lendo um `1` você move para q_{001}, mas lendo um `0` deixa você em q_{00}. Finalmente, em q_{001} lendo um `0` ou um `1` deixa você em q_{001}. O estado inicial é q, e o único estado de aceitação é q_{001}, como mostrado na figura seguinte.

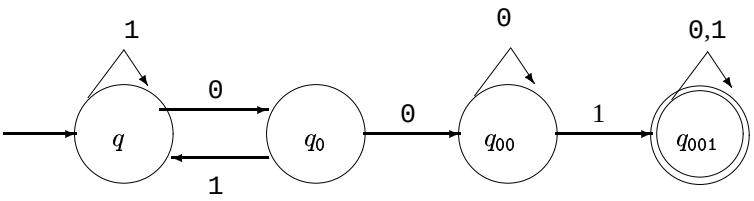


Figura 1.13: Aceita cadeias contendo `001`

As operações regulares

Nas duas seções precedentes introduzimos e definimos autômatos finitos e linguagens regulares. Agora começamos a investigar suas propriedades. Fazer assim ajudará a desenvolver uma caixa de ferramentas de técnicas para usar quando você projeta autômatos para reconhecer linguagens específicas. A caixa de ferramentas também incluirá maneiras de provar que certas outras linguagens são não-regulares (i.e, além da capacidade de autômatos finitos).

Em aritmética, os objetos básicos são números e as ferramentas são operações para manipulá-los, tais como $+$ e $\times$. Na teoria da computação os objetos são linguagens e as ferramentas incluem operações especificamente desenhadas para manipulá-las. Definimos três operações sobre linguagens, chamadas de as ***operações regulares***, e as usamos para estudar propriedades das linguagens regulares.

Definição 1.10
Sejam A e B linguagens. Definimos as operações regulares ***união***, ***concatenação***, e ***estrela*** da seguinte forma.

- **União**: $A \cup B = \{x \mid x \in A \text{ ou } x \in B\}$.
- **Concatenação**: $A \circ B = \{xy \mid x \in A \text{ e } y \in B\}$.

- **Estrela**: $A^* = \{x_1 x_2 \ldots x_k \mid k \geq 0 \text{ e cada } x_i \in A\}$.

Você já está familiar com a operação de união. Ela simplesmente toma todas as cadeias em ambas A e B e junta todas em uma linguagem.

A operação de concatenação é um pouco mais complicada. Ela acrescenta uma cadeia de A na frente de uma cadeia de B de todas as maneiras possíveis para obter as cadeias na nova linguagem.

A operação estrela é um pouco diferente das outras duas porque ela se aplica a uma única linguagem ao invés de duas. Ou seja, a operação estrela é uma ***operação unária*** ao invés de uma ***operação binária***. Ela funciona juntando um número qualquer de cadeias de A para obter uma cadeia na nova linguagem. Devido ao fato de que o termo "um número qualquer" inclui 0 como uma possibilidade, a cadeia vazia ε é sempre um membro de A^*, independente do que A seja.

Exemplo 1.11 .....

Seja o alfabeto Σ constituído das 23 letras $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. Se $A = \{\mathtt{legal}, \mathtt{ruim}\}$ e $B = \{\mathtt{garoto}, \mathtt{garota}\}$, então

$A \cup B = \{\mathtt{legal}, \mathtt{ruim}, \mathtt{garoto}, \mathtt{garota}\}$,
$B \circ A = \{\mathtt{garotolegal}, \mathtt{garotalegal}, \mathtt{garotoruim}, \mathtt{garotaruim}\}$, e
$A^* = \{\varepsilon, \mathtt{legal}, \mathtt{ruim}, \mathtt{legallegal}, \mathtt{legalruim}, \mathtt{ruimlegal}, \mathtt{ruimruim},$
$\mathtt{legallegallegal}, \mathtt{legallegalruim}, \mathtt{legalruimlegal}, \mathtt{legalruimruim}\}$.

Seja $\mathcal{N} = \{1, 2, 3, \ldots\}$ o conjunto dos números naturais. Quando dizemos que $\mathcal{N}$ é *fechado sob multiplicação* queremos dizer que, para quaisquer x e y em $\mathcal{N}$, o produto $x \times y$ também está em $\mathcal{N}$. Em geral, uma coleção de objetos é ***fechada*** sob alguma operação se aplicando-se aquela operação a membros da coleção retorna um objeto ainda na coleção. Mostramos que a coleção de linguagens regulares é fechada sob todas as três operações regulares. Na Seção 1.3 mostramos que essas são ferramentas úteis para manipulação de linguagens regulares e para o entendimento do poder dos autômatos finitos. Começamos com a operação de união.

Teorema 1.12 .....

A classe das linguagens regulares é fechada sob a operação de união.
Em outras palavras, se A_1 e A_2 são linguagens regulares, então $A_1 \cup A_2$ também o é.

.....

Idéia da prova. Temos duas linguagens regulares A_1 e A_2 e desejamos mostrar que $A_1 \cup A_2$ também é regular. Como A_1 e A_2 são regulares, sabemos que algum autômato finito M_1 reconhece A_1 e algum autômato finito M_2 reconhece A_2. Para provar que $A_1 \cup A_2$ é regular exibimos um autômato finito, chame-o M, que reconhece $A_1 \cup A_2$.

Essa é uma prova por construção. Construímos M a partir de M_1 e M_2. A máquina M tem que aceitar sua entrada exatamente quando uma das duas M_1 ou M_2 aceitar de modo a reconhecer a linguagem da união. Ela funciona *simulando* ambas M_1 e M_2 e aceitando se uma das duas simulações aceita.

Como podemos fazer a máquina M simular M_1 e M_2? Talvez ela primeiro deva simular M_1 sobre a entrada e então simular M_2 sobre a entrada. Mas temos que ser cuidadosos aqui! Uma vez que os símbolos da entrada são lidos e usados para simular M_1, não podemos "reenrolar a fita de entrada" para tentar a simulação sobre M_2. Precisamos de uma outra abordagem.

Faça de conta que você é M. À medida que os símbolos de entrada chegam um por um, você simula ambas M_1 e M_2 simultaneamente. Dessa maneira somente uma

passagem sobre a entrada é necessária. Mas você pode manter o registro de ambas as simulações com memória finita? Tudo o que você precisa memorizar é o estado em que cada máquina estaria se ela tivesse lido até aquele ponto na entrada. Por conseguinte você precisa memorizar um par de estados. Quantos pares possíveis existem? Se M_1 tem k_1 estados e M_2 tem k_2 estados, o número de pares de estados, um de M_1 e o outro de M_2, é o produto $k_1 \times k_2$. Esse produto será o número de estados em M, um para cada par. As transições de M vão de par para par, atualizando o estado corrente para ambas M_1 e M_2. Os estados de aceitação de M são aqueles pares nos quais M_1 ou M_2 está num estado de aceitação.

Prova.
Suponha que M_1 reconhece A_1, onde $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, e
M_2 reconhece A_2, onde $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$
Construa M para reconhecer $A_1 \cup A_2$, onde $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ e } r_2 \in Q_2\}$.

   Esse conjunto é o *produto cartesiano* dos conjuntos Q_1 e Q_2, e é escrito $Q_1 \times Q_2$. É o conjunto de todos os pares, o primeiro de Q_1 e o segundo de Q_2.

2. Σ, o alfabeto, é o mesmo que em M_1 e M_2. Neste teorema e em todos os teoremas subseqüentes similares, assumimos por simplicidade que ambos M_1 e M_2 têm o mesmo alfabeto de entrada Σ. O teorema permanece verdadeiro se eles tiverem alfabetos diferentes, Σ_1 e Σ_2. Modificaríamos então a prova para fazer $\Sigma = \Sigma_1 \cup \Sigma_2$.

3. δ, a função de transição, é definida como segue. Para cada $(r_1, r_2) \in Q$, e cada $a \in \Sigma$, faça
   $$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$
   Donde δ pega um estado de M (que na verdade é um par de estados de M_1 e M_2), juntamente com um símbolo de entrada, e retorna o estado seguinte de M.

4. q_0 é o par (q_1, q_2).

5. F é o conjunto dos pares nos quais um dos membros é um estado de aceitação de M_1 ou M_2.

   Podemos escrevê-lo como

   $$F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ ou } r_2 \in F_2\}.$$

   Essa expressão é a mesma que $F = (F_1 \times Q_1) \cup (Q_2 \times F_2)$. (Note que ela *não* é a mesma que $F = F_1 \times F_2$. O que essa última nos daria diferentemente da primeira?[3])

Isso conclui a construção do autômato finito M que reconhece a união de A_1 e A_2. Essa construção é bastante simples, e por conseguinte sua corretude é evidente da estratégia que é descrita na idéia da prova. Construções mais complicadas requerem discussão adicional para provar corretude. Uma prova formal de corretude para

[3]Essa expressão definiria os estados de aceitação de M como sendo aqueles para os quais *ambos* os membros do par são estados de aceitação. Nesse caso M aceitaria a cadeia somente se ambos M_1 *e* M_2 o aceitassem, portanto a linguagem resultante seria a *interseção* e não a união. Na realidade, esse resultado prova que a classe de linguagens regulares é fechada sob interseção.

uma construção desse tipo usualmente procede por indução. Para um exemplo de uma construção provada correta, veja a prova do Teorema 1.28. A maior parte das construções que você encontrará neste curso são bastante simples e portanto não requerem uma prova formal de corretude.

. . . . . .

Acabamos de mostrar que a união de duas linguagens regulares é regular, mostrando portanto que a classe das linguagens regulares é fechado sob a operação de união. Agora nos voltamos para a operação de concatenação e tentamos mostrar que a classe das linguagens regulares é fechada sob aquela operação também.

Teorema 1.12 . . . . . .
A classe das linguagens regulares é fechada sob a operação de concatenação.
Em outras palavras, se A_1 e A_2 são linguagens regulares então $A_1 \circ A_2$ também o é.

Para provar esse teorema vamos tentar algo na linha da prova do caso da união. Como antes, podemos começar com autômatos finitos M_1 e M_2 reconhecendo as linguagens regulares A_1 e A_2. Porém agora, ao invés de construir o autômato M para aceitar sua entrada se M_1 ou M_2 aceita, ele tem que aceitar se sua entrada pode ser quebrada em duas partes, onde M_1 aceita a primeira parte e M_2 aceita a segunda parte. O problema é que M não sabe onde quebrar sua entrada (i.e., onde a primeira parte termina e a segunda começa). Para resolver esse problema introduzimos uma técnica chamada não-determinismo.

1.2 Não-determinismo . . . . . .

Não-determinismo é um conceito útil que tem tido grande impacto sobre a teoria da computação. Até agora em nossa discussão, todo passo de uma computação segue de uma forma única do passo precedente. Quando uma máquina está num dado estado e lê o próximo símbolo de entrada, sabemos que estado será o próximo—ele está determinado. Chamamos isso de computação *determinística*. Em uma máquina ***não-determinística***, várias escolhas podem existir para cada próximo estado em qualquer ponto.

Não-determinismo é uma generalização de determinismo, portanto todo autômato finito determinístico é automaticamente um autômato finito não-determinístico. Como a figura a seguir mostra, autômatos finitos não-determinísticos podem ter características adicionais.

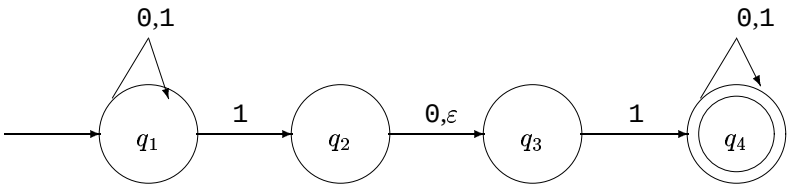


Figura 1.14: O autômato finito não-determinístico N_1

A diferença entre um autômato finito determinístico, abreviado por AFD, e um autômato finito não-determinístico, abreviado por AFN, é imediatamente aparente. Primeiro, todo estado de um AFD sempre tem exatamente uma seta de transição saindo para cada símbolo no alfabeto. O autômato não-determinístico mostrado na Figura 1.14 viola essa regra. O estado q_1 tem apenas uma seta saindo para o símbolo `0`, mas tem duas para `1`; q_2 tem uma seta para `0`, mas não tem nenhuma para `1`. Em um AFN um estado pode ter zero, um, ou mais setas saindo para cada símbolo do alfabeto.

Segundo, em um AFD, rótulos sobre as setas de transição são símbolos do alfabeto. Esse AFN tem uma seta com o rótulo ε. Em geral, um AFN pode ter setas rotuladas com membros do alfabeto ou com ε. Zero, uma, ou mais setas podem sair de cada estado com o rótulo ε.

Como um AFN computa? Suponha que você esteja rodando um AFN sobre uma cadeia de entrada e chegue num estado com múltiplas maneiras de prosseguir. Por exemplo, digamos que estamos no estado q_1 no AFN N_1 e que o próximo símbolo de entrada seja um `1`. Após ler esse símbolo, a máquina divide-se em múltiplas cópias de si mesma e segue *todas* as possibilidades em paralelo. Cada cópia da máquina toma uma das possíveis maneiras de prosseguir e continua como antes. Se existirem escolhas subseqüentes, a máquina divide-se novamente. Se o próximo símbolo de entrada não aparece sobre qualquer das setas saindo do estado ocupado por uma cópia da máquina, aquela cópia da máquina morre, juntamente com o ramo da computação associado a ela. Finalmente, se *qualquer uma* dessas cópias da máquina está em um estado de aceitação no final da entrada, o AFN aceita a cadeia de entrada.

Se um estado com um símbolo ε sobre uma seta saindo é encontrado, algo semelhante acontece. Sem ler qualquer entrada, a máquina divide-se em múltiplas cópias, uma seguindo cada uma das setas saindo rotuladas com ε e uma permanecendo no estado corrente. Então a máquina prossegue não-deterministicamente como antes.

Não-determinismo pode ser visto como uma espécie de computação paralela na qual vários "processos" podem estar rodando concorrentemente. Quando o AFN divide-se para seguir várias escolhas, isso corresponde a um processo "bifurcando" em vários filhotes, cada um prosseguindo separadamente. Se pelo menos um desses processos aceita então a computação inteira aceita.

Uma outra maneira de pensar em uma computação não-determinística é como uma árvore de possibilidades. A raiz da árvore corresponde ao início da computação. Todo ponto de ramificação na árvore corresponde a um ponto na computação no qual a máquina tem múltiplas cópias. A máquina aceita se pelo menos um dos ramos da computação termina em um estado de aceitação, como mostrado na Figura 1.15.

Vamos considerar algumas rodadas amostrais do AFN N_1 mostrado na Figura 1.14. Sobre a entrada `010110` comece no estado inicial q_1 e leia o primeiro símbolo `0`. A partir de q_1 existe somente um lugar para ir sobre `0`, a saber, de volta a q_1, portanto permaneça lá.

A seguir leia o segundo símbolo `1`. Em q_1 sobre um `1` existem duas escolhas: ou permaneça em q_1 ou mova para q_2. Não-deterministicamente, a máquina divide-se em duas para seguir cada escolha. Mantenha o registro das possibilidades colocando um dedo sobre cada estado onde a máquina poderia estar. Portanto você agora tem dedos sobre os estados q_1 e q_2. Uma seta ε sai do estado q_2 de modo que a máquina se divide novamente; mantenha um dedo sobre q_2, e mova o outro para q_3. Você agora tem dedos sobre q_1, q_2, e q_3.

Quando o terceiro símbolo `0` é lido, tome cada dedo por vez. Mantenha o dedo sobre q_1 no lugar, mova o dedo sobre q_2 para q_3, e remova o dedo que estava sobre q_3.

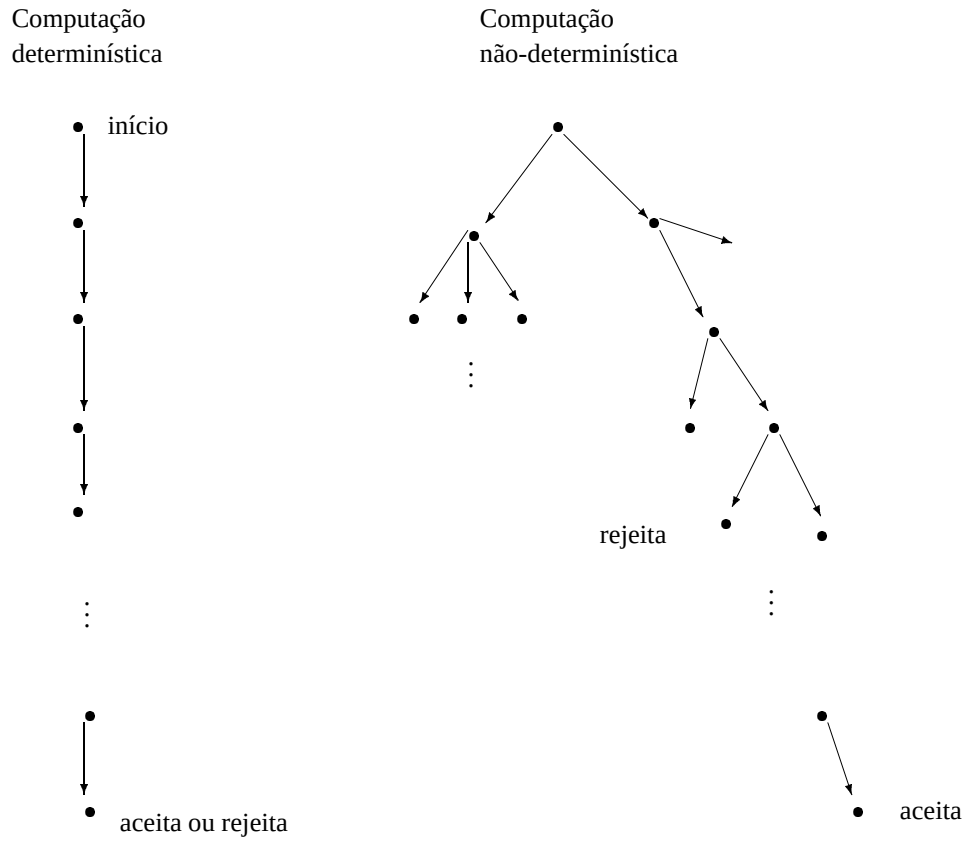


Figura 1.15: Computações determinísticas e não-determinísticas com um ramo de aceitação

Esse último dedo não tinha nenhuma seta `0` para seguir e corresponde a um processo que simplesmente "morre." Nesse ponto você tem dedos sobre os estados q_1 e q_3.

Quando o quarto símbolo `1` é lido, divida o dedo sobre q_1 em dedos sobre q_1 e q_2, e divida ainda o dedo sobre q_2 para seguir a seta ε para q_3, e mova o dedo que estava sobre q_3 para q_4. Você agora tem um dedo sobre cada um dos quatro estados.

Quando o quinto símbolo `1` é lido, os dedos sobre q_1 e q_3 resultam em dedos sobre os estados q_1, q_2, q_3, e q_4, como você viu com o quarto símbolo. O dedo sobre o estado q_2 é removido. O dedo que estava sobre q_4 permanece sobre q_4. Agora você tem dois dedos sobre q_4, portanto remova um, pois você apenas precisa memorizar que q_4 é um estado possível nesse ponto, não que ele é possível por múltiplas razões.

Quando o sexto e último símbolo `0` é lido, mantenha o dedo sobre q_1 no lugar, mova aquele sobre q_2 para q_3, remova aquele que estava sobre q_3, e deixe aquele sobre q_4 no lugar. Você está agora no final da cadeia, e você aceita se algum dedo está sobre um estado de aceitação. Você tem dedos sobre os estados q_1, q_3, e q_4, e como q_4 é um estado de aceitação, N_1 aceita essa cadeia. A computação de N_1 sobre a entrada `010110` está ilustrada na Figura 1.16.

O que N_1 faz sobre a entrada `010`? Comece com um dedo sobre q_1. Após ler o `0` você ainda tem um dedo somente sobre q_1, mas após o `1` existem dedos sobre q_1, q_2, e q_3 (não esqueça a seta ε). Após o terceiro símbolo `0`, remova o dedo sobre q_3, mova

o dedo sobre q_2 para q_3, e deixe o dedo sobre q_1 onde está. Nesse ponto você está no final da entrada, e como nenhum dedo está sobre um estado de aceitação, N_1 rejeita essa entrada.

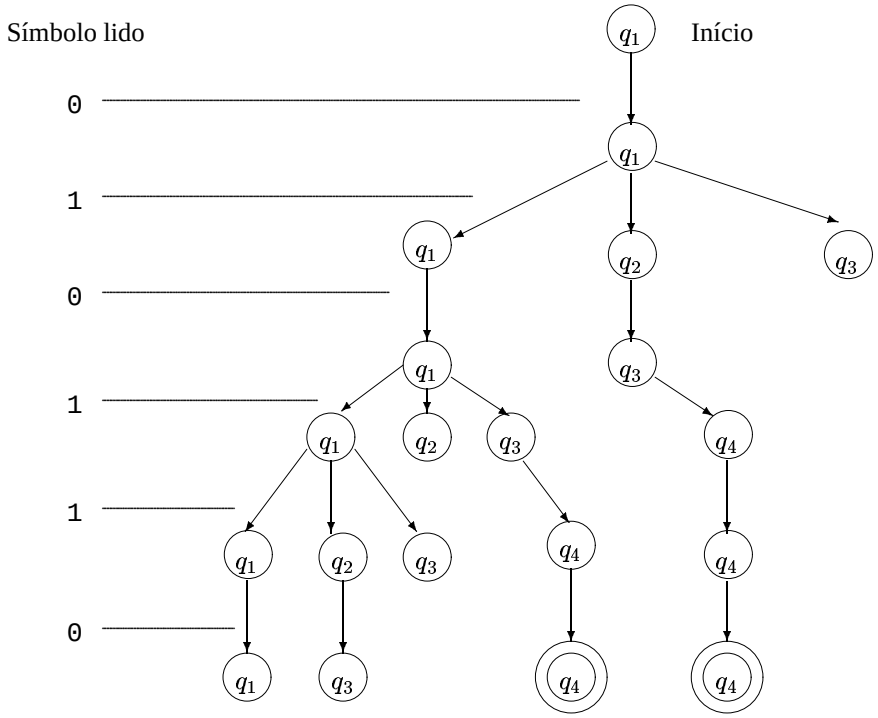


Figura 1.16: A computação de N_1 sobre a entrada `010110`

Continuando a experimentar dessa maneira, você verá que N_1 aceita todas as cadeias que contêm `101` ou `11` como subcadeia.

Autômatos finitos não-determinísticos são úteis em vários aspectos. Como mostraremos, todo AFN pode ser convertido em um AFD equivalente, e construir AFN's é às vezes mais fácil que construir diretamente AFD's. Um AFN pode ser muito menor que sua contrapartida determinística, ou seu funcionamente ser mais fácil de entender. Não-determinismo em autômatos finitos é também uma boa introdução a não-determinismo em modelos computacionais mais poderosos porque autômatos finitos são especialmente fáceis de entender. Agora nos voltamos para vários exemplos de AFN's.

Exemplo 1.14 .....
Seja A a linguagem consistindo de todas as cadeias sobre $\{\mathtt{0},\mathtt{1}\}$ contendo um `1` na terceira posição a partir do final (e.g., `000100` está em A mas `0011` não está). O seguinte AFN N_2 de quatro estados reconhece A.

Uma maneira boa de ver a computação desse AFN é dizer que ele permanece no estado inicial q_1 até que ele "adivinha" que ele está a três posições do final. Naquele ponto, se o símbolo de entrada é um `1`, ele ramifica para o estado q_2 e usa q_3 e q_4 para "verificar" se seu palpite estava correto.

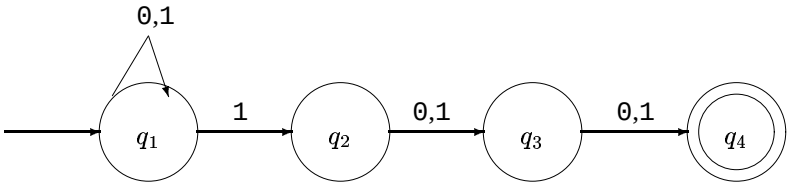


Figura 1.17: O AFN N_2 que reconhece A

Conforme mencionado anteriormente, todo AFN pode ser convertido em um AFD equivalente, mas às vezes aquele AFD pode ter muito mais estados. O menor AFD para A contém oito estados. Além disso, entender o funcionamento do AFD é muito mais fácil, como você pode ver examinando a Figura 1.18 para o AFD.

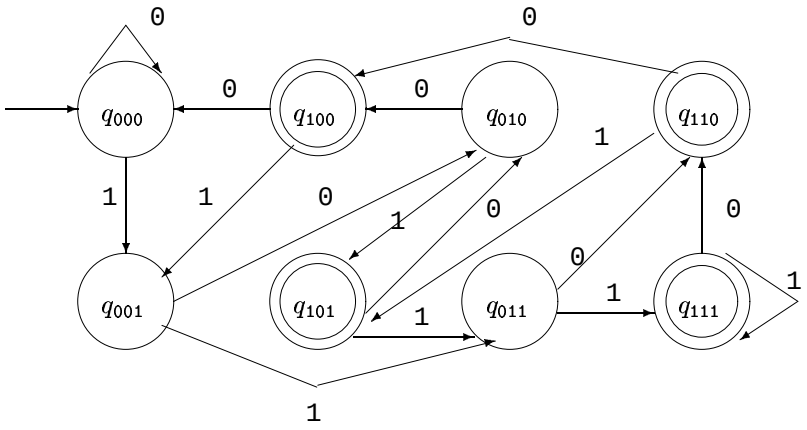


Figura 1.18: Um AFD que reconhece A

Suponha que adicionássemos ε aos rótulos sobre as setas indo de q_2 para q_3 e de q_3 para q_4 na máquina N_2 na Figura 1.17. Em outras palavras, ambas as setas teriam então o rótulo $\mathtt{0}, \mathtt{1}, \varepsilon$ ao invés de $\mathtt{0}, \mathtt{1}$. Que linguagem N_2 reconheceria com essa modificação? Tente modificar o AFD na Figura 1.18 para reconhecer essa linguagem.

Exemplo 1.15 . . . . . .
Considere o seguinte AFN N_3 que tem como alfabeto de entrada $\{\mathtt{0}\}$ consistindo de um único símbolo. Um alfabeto contendo apenas um símbolo é chamado de alfabeto *unário*.

Essa máquina demonstra a conveniência de se ter setas ε. Ela aceita cadeias da forma $\mathtt{0}^k$ onde k é um múltiplo de 2 ou 3. (Lembre-se que o expoente denota repetição, e não exponenciação numérica.) Por exemplo, N_2 aceita as cadeias ε, $\mathtt{00}$, $\mathtt{000}$, $\mathtt{0000}$, e $\mathtt{000000}$, mas não $\mathtt{0}$ ou $\mathtt{00000}$.

Pense na máquina operando de forma a inicialmente adivinhar se testa por um múltiplo de 2 ou um múltiplo de 3 ramificando ou no laço superior ou no laço inferior, e então verificando se seu palpite estava correto. É claro que poderíamos substituir essa

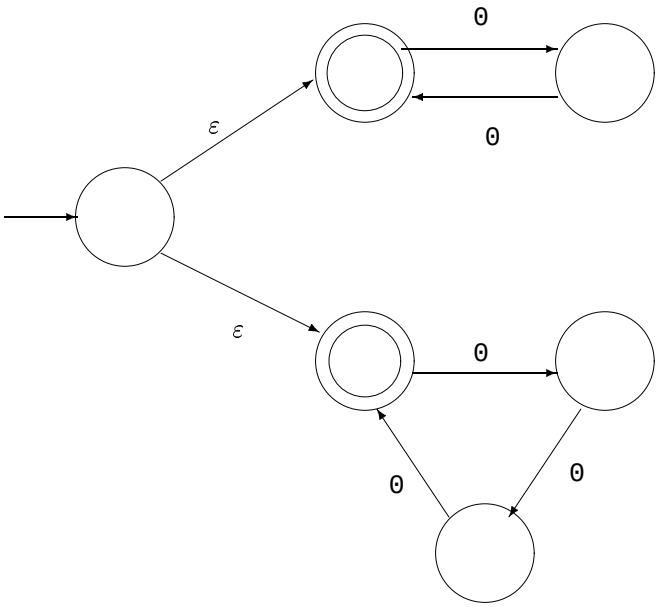


Figura 1.19: O AFN N_3

máquina por uma que não tem setas ε ou mesmo qualquer não-determinismo mesmo, mas a máquina mostrada é a mais fácil de entender essa linguagem.

Exemplo 1.16 .....

Damos um outro exemplo de um AFN na figura seguinte. Pratique com ele para se satisfazer de que ele aceita as cadeias ε, a, baba, e baa, mas que ele não aceita as cadeias b, bb, e babba. Mas adiante usamos essa máquina para ilustrar o procedimento para converter AFN's em AFD's.

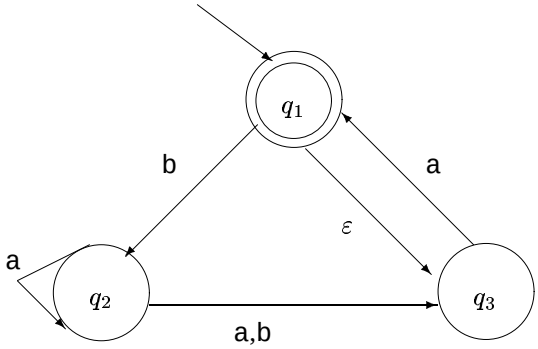


Figura 1.20: O AFN N_4

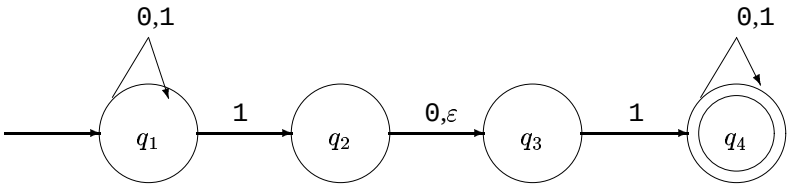
Definição formal de um autômato finito não-determinístico

A definição formal de um autômato finito não-determinístico é semelhante àquela de um autômato finito determinístico. Ambos têm estados, um alfabeto de entrada, uma função de transição, um estado inicial, e uma coleção de estados de aceitação. Entretanto, eles diferem de uma maneira essencial: no tipo de função de transição. Em um AFD a função de transição toma um estado e um símbolo de entrada e produz o próximo estado. Em um AFN a função de transição toma um estado e um símbolo de entrada *ou a cadeia vazia* e produz *o conjunto de estados seguintes possíveis*. De modo a escrever a definição formal, precisamos de fixar um pouco de notação adicional. Para qualquer conjunto Q escrevemos $\mathcal{P}(Q)$ como sendo a coleção de todos os subconjuntos de Q. Aqui $\mathcal{P}(Q)$ é chamado de *conjunto potência* de Q. Para qualquer alfabeto Σ escrevemos Σ_ε como sendo $\Sigma \cup \{\varepsilon\}$. Agora podemos facilmente escrever a definição formal do tipo de função de transição em um AFN. Ela é: $\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$, e estamos prontos para dar a definição formal.

Definição 1.17 . . . . . .
Um ***autômato finito não-determinístico*** é uma 5-upla $(Q, \Sigma, \delta, q_0, F)$ onde

1. Q é um conjunto finito de estados,
2. Σ é um alfabeto finito,
3. $\delta : Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ é a ***função de transição***,
4. $q_0 \in Q$ é o estado inicial, e
5. $F \subseteq Q$ é o conjunto de estados de aceitação.

Exemplo 1.18 . . . . . .
Retomemos o autômato N_1:



A descrição formal de N_1 é $(Q, \Sigma, \delta, q_1, F)$, onde

1. $q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0, 1\}$,
3. δ é dada como

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$

4. q_1 é o estado inicial, e
5. $F = \{q_4\}$.

A definição formal de computação de um AFN também é semelhante àquela para um AFD. Seja $N = (Q, \Sigma, \delta, q_0, F)$ um AFN e w uma cadeia sobre o alfabeto Σ. Então dizemos que N ***aceita*** w se podemos escrever w como $w = y - 1y_2 \cdots y_m$, onde cada y_i é um membro de Σ_ε e uma seqüência de estados $r_0, r_1, \ldots, r_m$ existe em Q com as seguintes três condições:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$ para $i = 0, \ldots, m-1$, e
3. $r_m \in F$.

Condição 1 diz que a máquina começa no estado inicial. Condição 2 diz que o estado r_{i+1} é um dos estados seguintes permissíveis quando N estado no estado r_i e está lendo y_{i+1}. Observe que $\delta(r_i, y_{i+1})$ é o *conjunto* de estados seguintes permissíveis e portanto dizemos que r_{i+1} é um membro daquele conjunto. Finalmente, Condição 3 diz que a máquina aceita sua entrada se o último estado é um estado de aceitação.

Equivalência de AFN's e AFD's

Autômatos finitos determinísticos e não-determinísticos reconhecem a mesma classe de linguagens. Tal equivalência é tanto surpreendente quanto útil. Ela é surpreendente porque AFN's parecem ter mais poder que AFD's, portanto poderíamos esperar que AFN's reconhecessem mais linguagens. É útil porque descrever um AFN para uma dada linguagem às vezes é muito mais fácil que descrever um AFD para aquela linguagem.

Digamos que duas máquinas são *equivalentes* se elas reconhecem a mesma linguagem.

Teorema 1.19 .....
Todo autômato finito não-determinístico tem um autômato finito determinístico equivalente.

.....

Idéia da prova. Se uma linguagem é reconhecida por um AFN, então temos que mostrar a existência de um AFD que também a reconhece. A idéia é converter o AFN em um AFD equivalente que simula o AFN.

Lembre-se da estratégia "leitor como autômato" para projetar autômatos finitos. Como você simularia o AFN se você estivesse fazendo de conta que é um AFD? O que você precisaria memorizar à medida que a cadeia de entrada é processada? Nos exemplos de AFN's você memorizava as várias ramificações da computação colocando um dedo sobre cada estado que poderia estar ativo em dados pontos na entrada. Você atualizava os dedos movendo-, adicionando-, e removendo-os conforme a maneira com que o AFN opera. Tudo o que você precisava memorizar era o conjunto de estados com dedos sobre eles.

Se k é o número de estados do AFN, ele tem 2^k subconjuntos de estados. Cada subconjunto corresponde a uma das possibilidades de que o AFD tem que se lembrar, portanto o AFD que simula o AFN terá 2^k estados. Agora você precisa identificar

qual será o estado inicial e os estados de aceitação do AFD, e qual será sua função de transição. Podemos discutir isso mais facilmente fixando um pouco de notação formal.

Prova.
Seja $N = (Q, \Sigma, \delta, q_0, F)$ o AFN que reconhece uma dada linguagem A. Construimos um AFD M que reconhece A. Antes de realizar toda a construção, vamos primeiro considerar o caso mais fácil no qual N não tem setas ε. Mais adiante levaremos em conta as setas ε.

Construa $M = (Q', \Sigma, \delta', q_0', F')$.

1. $Q' = \mathcal{P}(Q)$.

   Todo estado de M é um conjunto de estados de N. Lembre-se que $\mathcal{P}(Q)$ é o conjunto de todos os subconjuntos de Q.

2. Para $R \in Q'$ e $a \in \Sigma$ seja $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ para algum } r \in R\}$.

   Se R é um estado de M, ele também é um conjunto de estados de N. Quando M lê um símbolo a no estado R, ele mostra para onde a leva cada estado em R. Devido ao fato de que cada estado pode ir para um conjunto de estados, tomamos a união de todos esses conjuntos. Uma outra forma de escrever essa expressão é

   $$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).^4$$

3. $q_0' = \{q_0\}$.

   M começa no estado correspondente à coleção contendo apenas o estado inicial de N.

4. $F' = \{R \in Q' \mid R \text{ contém um estado de aceitação de } N\}$.

   A máquina M aceita se um dos estados possíveis em que N poderia estar nesse ponto é um estado de aceitação.

Agora precisamos considerar as setas ε. Para fazer isso fixemos um pouco mais de notação. Para qualquer estado R de M definimos $E(R)$ como sendo a coleção de estados que podem ser atingidos a partir de R percorrendo somente setas ε, incluindo os próprios membros de R. Formalmente, para $R \subseteq Q$ faça

$$E(R) = \{q \mid q \text{ pode ser atingido a partir de } R \text{ passando por } 0 \text{ ou mais setas } \varepsilon\}.$$

Então modificamos a função de transição de M para colocar dedos a mais sobre todos os estados que podem ser atingidos passando por setas ε após cada passo. Substituindo $\delta(r, a)$ por $E(\delta(r, a))$ faz esse efeito. Por conseguinte

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ para algum } r \in R\}.$$

Adicionalmente precisamos modificar o estado inicial de M para mover os dedos inicialmente para todos os estados possíveis que podem ser atingidos a partir do estado inicial de N ao longo de setas ε. Mudando q_0' para $E(\{q_0\})$ faz esse efeito. Agora completamos a construção do AFD M que simula o AFN N.

[4] A notação $\bigcup_{r \in R} \delta(r, a)$ significa: a união dos conjuntos $\delta(r, a)$ para cada possível r em R.

A construção de M obviamente funciona corretamente. Em todos os passos da computação de M sobre uma entrada, ela claramente entra em um estado que corresponde ao subconjunto de estados em que N poderia estar naquele ponto. Por conseguinte nossa prova está completa.

Se a construção usada na prova anterior fosse mais complexa precisaríamos de provar que ela funciona tal qual reivindicado. Usualmente tais provas procedem por inducção sobre o número de passos da computação. A maioria das construções que usamos neste livro são simples e portanto não requerem tal prova de corretude. Para ver um exemplo de uma construção mais complexa que provamos correta vá para a prova do Teorema 1.28.

O Teorema 1.19 enuncia que todo AFN pode ser convertido num AFD equivalente. Por conseguinte autômatos finitos não-determinísticos dão uma maneira alternativa de caracterizar linguagens regulares. Enunciamos esse fato como um corolário do Teorema 1.19.

Corolário 1.20

Uma linguagem é regular se e somente se algum autômato finito não-determinístico a reconhece.

Uma direção do "se e somente se" enuncia que uma linguagem é regular se algum AFN a reconhece. O Teorema 1.19 mostra que qualquer AFN pode ser convertido para um AFD equivalente, portanto se um AFN reconhece uma dada linguagem, o mesmo acontece com algum AFD, e portanto a linguagem é regular. A outra direção enuncia que uma linguagem é regular somente se algum AFN a reconhece. Ou seja, se uma linguagem é regular, algum AFN tem que a estar reconhecendo. Obviamente, essa condição é verdadeira pois uma linguagem regular tem um AFD que a reconhece e qualquer AFD é também um AFN.

Exemplo 1.21

Vamos ilustrar o procedimento de converter um AFN para um AFD usando a máquina N_4 que foi dada no Exemplo 1.16. Para maior clareza, vamos renomear os estados de N_4 para $\{1,2,3\}$. Por conseguinte, a descrição formal de $N_4 = (Q, \{\mathsf{a},\mathsf{b}\}, \delta, 1, \{1\})$, o conjunto de estados Q é $\{1,2,3\}$ como mostrado na Figura 1.21.

Para construir um AFD D que é equivalente a N_4, primeiro determinamos os estados de D. N_4 tem três estados, $\{1,2,3\}$, portanto construimos D com oito estados, um para cada subconjunto dos estados de N_4. Rotulamos cada um dos estados de D com o subconjunto correspondente. Por conseguinte o conjunto de estados de D é

$$\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}.$$

A seguir, determinamos os estados inicial e de aceitação de D. O estado inicial é $E(\{1\})$, o conjunto de estados que são atingíveis a partir de 1 viajando por setas ε, mais o próprio 1. Uma seta ε vai de 1 para 3, portanto $E(\{1\}) = \{1,3\}$. Os novos estados de aceitação são aqueles contendo o estado de aceitação de N_4; por conseguinte $\{\{1\},\{1,2\},\{1,3\},\{1,2,3\}\}$.

Finalmente, determinamos a função de transição de D. Cada um dos estados de D vai para um lugar sobre a entrada a, e um lugar para a entrada b. Ilustramos o processo de determinar a colocação das setas de transição de D com uns poucos exemplos.

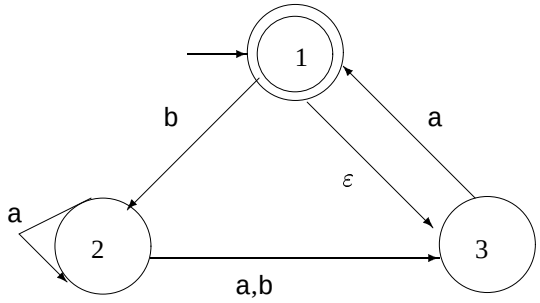


Figura 1.21: O AFN N_4

Em D, o estado $\{2\}$ vai para $\{2,3\}$ sobre a entrada a, porque em N_4 o estado 2 vai tanto para 2 quanto para 3 sobre a entrada a e não podemos ir além de 2 ou 3 ao longo de setas ε. O estado $\{2\}$ vai para o estado $\{3\}$ sobre a entrada b, porque em N_4 o estado 2 vai somente para o estado 3 sobre a entrada b e não podemos ir além de 3 ao longo de setas ε.

O estado $\{1\}$ vai para $\emptyset$ sobre a, porque nenhuma seta a sai dele. Ele vai para $\{2\}$ sobre b.

O estado $\{3\}$ vai para $\{1,3\}$ sobre a, porque em N_4 o estado 3 vai para 1 sobre a e 1 por sua vez vai para 3 com uma seta ε. O estado $\{3\}$ sobre b vai para $\emptyset$.

O estado $\{1,2\}$ sobre a vai para $\{2,3\}$ porque 1 não aponta para nenhum estado com setas a e 2 aponta tanto para 2 quanto para 3 com setas a e nenhum aponta para lugar nenhum com setas ε. O estado $\{1,2\}$ sobre b vai para $\{2,3\}$. Continuando dessa forma obtemos o seguinte diagrama para D.

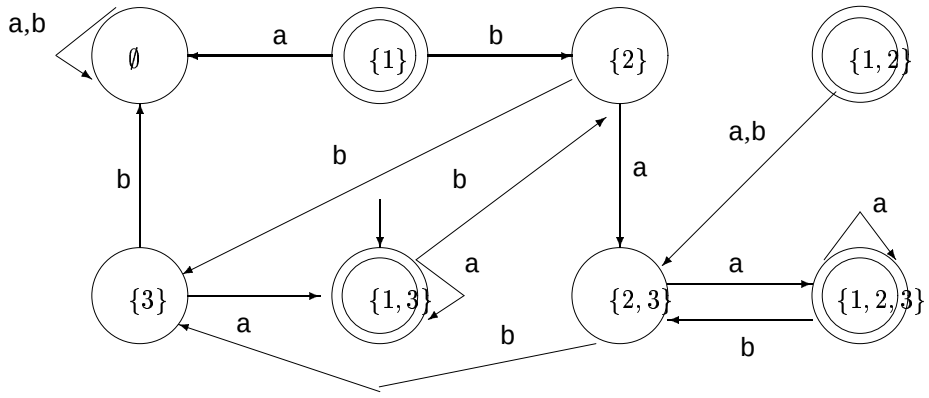


Figura 1.22: Um AFD D que é equivalente ao AFN N_4

Podemos simplificar essa máquina observando que nenhuma seta aponta para os estados $\{1\}$ e $\{1,2\}$, portanto eles podem ser removidos sem afetar o desempenho da máquina. Fazendo isso obtém-se a seguinte figura:

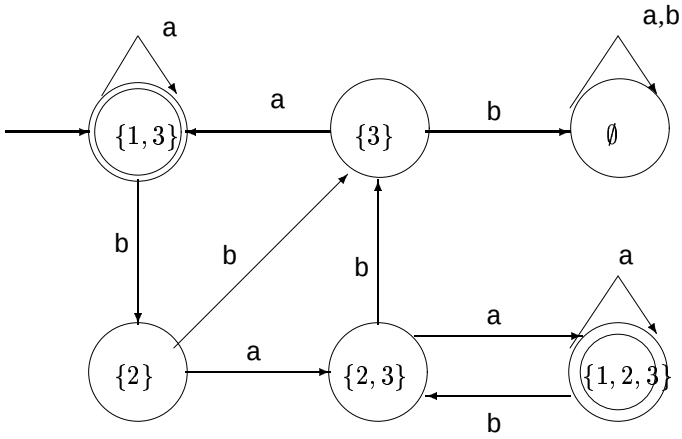


Figura 1.23: O AFD D após a remoção de estados desnecessários

Fecho sob as operações regulares

Agora voltamos ao fecho da classe das linguagens regulares sob as operações regulares que começamos na Seção 1.1. Nosso objetivo é provar que a união, concatenação, e estrela de linguagens regulares são ainda regulares. Abandonamos a tentativa original de fazer isso quando lidar com a operação de concatenação era complicado demais. O uso de não-determinismo torna as provas mais fáceis.

Primeiro, vamos considerar novamente o fecho sob união. Anteriormente provamos o fecho sob união simulando deterministicamente ambas as máquinas simultaneamente via uma construção de produto cartesiano. Agora damos uma nova prova para ilustrar a técnica do não-determinismo. Revisar a primeira prova, na página ??, pode valer a pena para ver quão mais fácil e mais intuitiva a nova prova é.

Teorema 1.22 .....
A classe de linguagens regulares é fechada sob a operação de união.

.....

Idéia da prova. Temos linguagens regulares A_1 e A_2 e desejamos provar que $A_1 \cup A_2$ é regular. A idéia é tomar dois AFN's, N_1 e N_2 para A_1 e A_2, e combiná-los em um novo AFN N.

A máquina N tem que aceitar sua entrada se N_1 ou N_2 aceita sua entrada. A nova máquina tem um novo estado inicial que ramifica para os estados iniciais das máquinas antigas com setas ε. Dessa maneira a nova máquina não-deterministicamente adivinha qual das duas máquinas aceita a entrada. Se uma delas aceita a entrada, N a aceitará também.

Representamos essa construção na Figura 1.24. À esquerda indicamos os estados inicial e de aceitação das máquinas N_1 e N_2 com círculos maiores e alguns estados adicionais com círculos menores. À direita mostramos como combinar N_1 e N_2 em N adicionando setas extra de transição.

Prova.
Seja $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, e

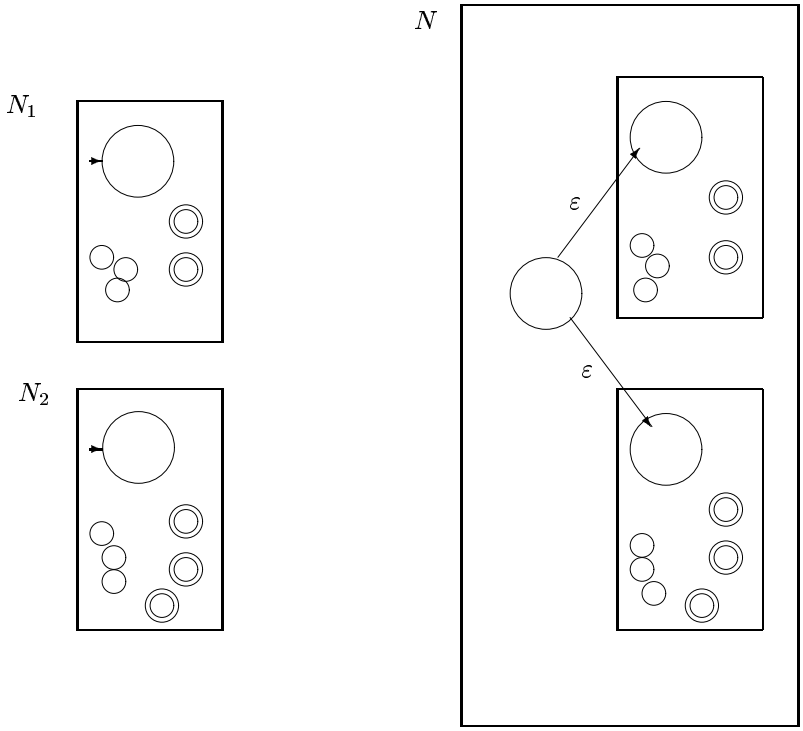


Figura 1.24: Construção de um AFN N para reconhecer $A_1 \cup A_2$

$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construa $N = (Q, \Sigma, \delta, q_0, F)$ para reconhecer $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.

   Os estados de N são todos estados de N_1 e N_2, com a adição de um novo estado q_0.

2. O estado q_0 é o estado inicial de N.

3. Os estados de aceitação $F = F_1 \cup F_2$.

   Os estados de aceitação de N são todos os estados de aceitação de N_1 e N_2. Dessa forma N aceita se N_1 aceita ou N_2 aceita.

4. Defina δ de modo que para qualquer $q \in Q$ e qualquer $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

. . . . . .

Agora podemos provar o fecho sob concatenação. Lembre-se que anteriormente, sem não-determinismo, completar a prova teria sido difícil.

Teorema 1.23 .....
A classe de linguagens regulares é fechada sob a operação de concatenação.

.....

Idéia da prova. Temos linguagens regulares A_1 e A_2 e desejamos provar que $A_1 \circ A_2$ é regular. A idéia é tomar dois AFN's N_1 e N_2 para A_1 e A_2, e combiná-los em um novo AFN N como fizemos para o caso da união, mas dessa vez de uma forma diferente, como mostrado na Figura 1.25.

Associe o estado inicial de N ao estado inicial de N_1. Os estados de aceitação de N_1 têm setas ε adicionais que não-deterministicamente permitem ramificação para N_2 sempre que N_1 está em um estado de aceitação, significando que ela encontrou uma parte inicial da cadeia que constitui uma cadeia em A_1. Os estados de aceitação de N são estados de aceitação de N_2 somente. Por conseguinte ele aceita quando a entrada pode ser dividida em duas partes, a primeira aceita por N_1 e a segunda aceita por N_2. Podemos pensar em N como adivinhando não-deterministicamente onde fazer a divisão.

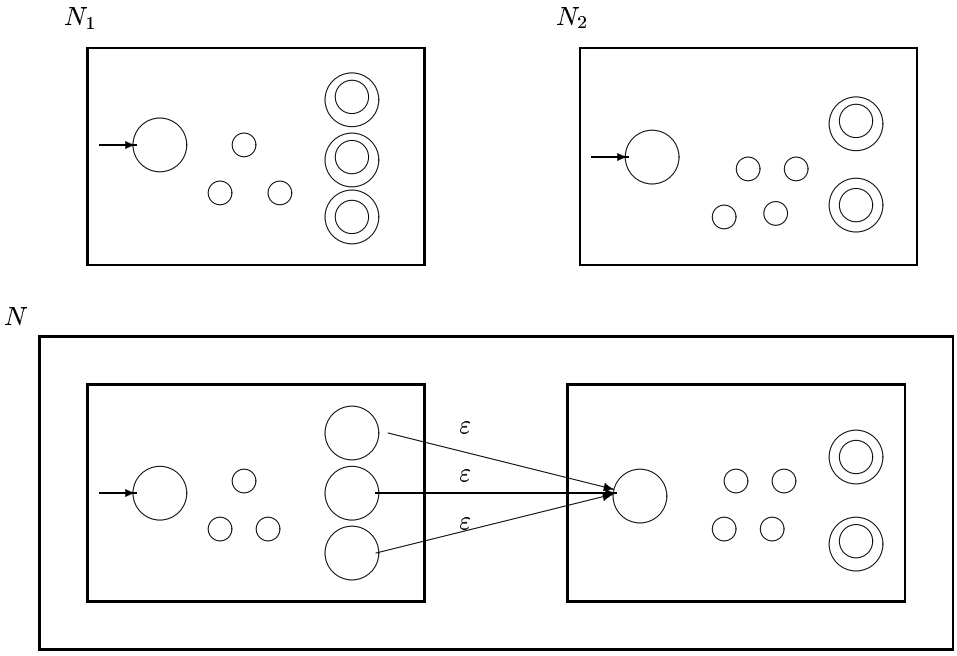


Figura 1.25: Construção de N para reconhecer $A_1 \circ A_2$

Prova.
Suponha que N_1 $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ reconhece A_1, e
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ reconhece A_2.
Construa $N = (Q, \Sigma, \delta, q_1, F_2)$ para reconhecer $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.

   Os estados de N são todos os estados de N_1 e N_2.

2. O estado q_1 é o mesmo que o estado inicial de N_1.

3. Os estados de aceitação F_2 são os mesmos que os estados de aceitação de N_2.

4. Defina δ tal que para qualquer $q \in Q$ e qualquer $a \in \Sigma_\varepsilon$

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_2(q,a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ e } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

. . . . . .

Teorema 1.23 . . . . . .
A classe de linguagens regulares é fechada sob a operação estrela.

. . . . . .

Idéia da prova. Temos uma linguagem regular A_1 e queremos provar que A_1^* também é regular. Tomamos um AFN N_1 para A_1 e o modificamos para reconhecer A_1^*, como mostrado na Figura 1.26. O AFN resultante N aceitará sua entrada sempre que ela puder ser partida em vários pedaços em N_1 aceita cada pedaço.

Podemos construir N como N_1 com setas ε adicionais retornando ao estado inicial a partir do estado de aceitação. Dessa maneira, quando o processamento chega ao final de um pedaço que N_1 aceita, a máquina N tem a opção de pular para o estado inicial e tentar ler um outro pedaço que N_1 aceita. Adicionalmente temos que modificar N de tal modo que ela aceite ε, que sempre será um membro de A_1^*. Uma idéia (não muito boa) é simplesmente adicionar o estado inicial ao conjunto de estados de aceitação. Essa abordagem certamente adiciona ε à linguagem reconhecida, mas pode também adicionar outras cadeias indesejadas. O Exercício 1.11 pede um exemplo da falha dessa idéia. A forma de consertar esse problema é adicionar um novo estado inicial, que também é um estado de aceitação, e que tenha uma seta ε para o antigo estado inicial. Essa solução tem o efeito desejado de adicionar ε à linguagem sem adicionar qualquer outra coisa.

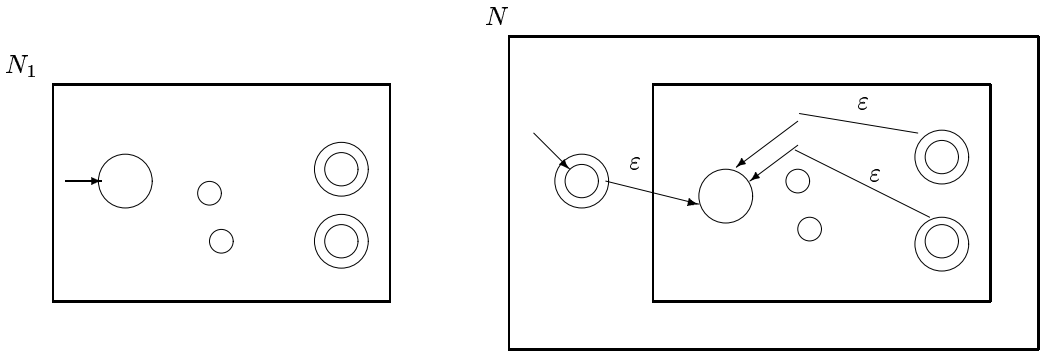


Figura 1.26: Construção de N para reconhecer A_1^*

Prova.
Suponha que $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ reconhece A_1.
Construa $N = (Q, \Sigma, \delta, q_0, F)$ para reconhecer A_1^*.

1. $Q = \{q_0\} \cup Q_1$.

   Os estados de N são os estados de N_1 mais um novo estado inicial.

2. O estado q_0 é o novo estado inicial.

3. $F = \{q_0\} \cup F_1$.

   Os estados de aceitação são os antigos estados de aceitação mais o novo estado inicial.

4. Defina δ tal que para qualquer $q \in Q$ e qualquer $a \in \Sigma_\varepsilon$

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ e } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

. . . . . .

1.3 Expressões regulares . . . . . .

Em aritmética, podemos usar as oprações $+$ e $\times$ para construir expressões tais como

$$(5 + 3) \times 4$$

Igualmente, podemos usar operações regulares para construir expressões descrevendo linguagens, que são chamadas *expressões regulares*. Um exemplo é

$$(\mathtt{0} \cup \mathtt{1})\mathtt{0}^*$$

O valor da expressão aritmética é o número 32. O valor de uma expressão regular é uma linguagem. Nesse caso o valor é a linguagem consistindo de todas as cadeias com um `0` ou um `1` seguido de um número qualquer de `0`'s. Obtemos esse resultado dissecando a expressão em suas partes. Primeiro, os símbolos `0` e `1` são abreviações para os conjuntos $\{\mathtt{0}\}$ e $\{\mathtt{1}\}$. Portanto $(\mathtt{0} \cup \mathtt{1})$ significa $(\{\mathtt{0}\} \cup \{\mathtt{1}\})$. O valor dessa parte é a linguagem $\{\mathtt{0}, \mathtt{1}\}$. A parte $\mathtt{0}^*$ significa $\{\mathtt{0}\}^*$, e seu valor é a linguagem consistindo de todas as cadeias contendo um número qualquer de `0`'s. Segundo, como o símbolo $\times$ em álgebra, o símbolo de concatenação $\circ$ frequentemente está implícito nas expressões regulares. Por conseguinte $(\mathtt{0} \cup \mathtt{1})\mathtt{0}^*$ na verdade é uma abreviação para $(\mathtt{0} \cup \mathtt{1}) \circ \mathtt{0}^*$. A concatenação junta as cadeias das duas partes para obter o valor da expressão inteira.

Expressões regulares têm um papel importante em aplicações em ciência da computação. Em aplicações envolvendo texto, usuários podem querer buscar por cadeias que satisfazem certos padrões. Utilitários como AWK e GREP em UNIX, linguagens de programação como PERL, e editores de texto, todos provêem mecanismos para a descrição de padrões usando expressões regulares.

Exemplo 1.25 . . . . . .
Um outro exemplo de uma expressão regular é

$$(\mathtt{0} \cup \mathtt{1})^*$$

Ela começa com a linguagem $(\mathtt{0} \cup \mathtt{1})$ e aplica a operação $*$. O valor dessa expressão é a linguagem consistindo de todas as cadeias possíveis de 0's e 1's. Se $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, podemos escrever Σ como abreviação para a expressão regular $(\mathtt{0} \cup \mathtt{1})$. De um modo geral, se Σ é um alfabeto qualquer, a expressão regular Σ descreve a linguagem consistindo de todas as cadeias de comprimento 1 sobre esse alfabeto, e Σ^* descreve a linguagem consistindo de todas as cadeias sobre aquele alfabeto. Igualmente $\Sigma^*\mathtt{1}$ é a linguagem que contém todas as cadeias que terminam em um 1. A linguagem $(\mathtt{0}\Sigma^*) \cup (\Sigma^*\mathtt{1})$ consiste de todas as cadeias que ou começam com um 0 ou terminam com um 1.

Em aritmética, dizemos que $\times$ tem precedência sobre $+$ para dizer que, quando existe uma escolha, fazemos a operação $\times$ primeiro. Por conseguinte em $2 + 3 \times 4$ o 3×4 é realizado antes da adição. Para ter a adição realizada primeiro temos que adicionar parênteses para obter $(2 + 3) \times 4$. Em expressões regulares, a operação estrela é realizada primeiro, seguida por concatenação, e finalmente união, a menos que parênteses sejam usados para mudar a ordem usual.

Definição formal de uma expressão regular

Definição 1.26 .....
Digamos que R é uma ***expressão regular*** se R é

1. a para algum a no alfabeto Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, onde R_1 e R_2 são expressões regulares,
5. $(R_1 \circ R_2)$, onde R_1 e R_2 são expressões regulares, ou
6. (R_1^*), onde R_1 é uma expressão regular.

Nos itens 1 e 2, as expressões regulares a e ε representam as linguagens $\{a\}$ e $\{\varepsilon\}$, respectivamente. No item 3, a expressão regular $\emptyset$ representa a linguagem vazia. Nos itens 4,5, e 6, as expressões representam as linguagens obtidas tomando-se a união ou concatenação das linguagens R_1 e R_2, ou a estrela da linguagem R_1, respectivamente.

Não confunda as expressões regulares ε e $\emptyset$. A expressão ε representa a linguagem contendo uma única cadeia, a saber, a cadeia vazia, enquanto que $\emptyset$ representa a linguagem que não contém qualquer cadeia.

Aparentemente, estamos sob risco de definir a noção de expressão regular em termos de si própria. Se verdadeiro, teríamos uma ***definição circular***, o que seria inválido. Entretanto, R_1 e R_2 são sempre menores que R. Por conseguinte estamos na verdade definindo expressões regulares em termos de expressões regulares menores e dessa forma evitando circularidade. Uma definição desse tipo é chamada de ***definição indutiva***.

Parênteses em uma expressão regular podem ser omitidos. Se assim for, o cálculo é feito na ordem de precedência: estrela, então concatenação, então união.

Quando desejamos deixar claro uma distinção entre uma expressão regular R e a linguagem que ela descreve, escrevemos $L(R)$ como sendo a linguagem de R.

Exemplo 1.27
Nos exemplos abaixo assumimos que o alfabeto Σ é $\{\mathtt{0}, \mathtt{1}\}$.

1. $\mathtt{0}^*\mathtt{10}^* = \{w \mid w$ tem exatamente um único $\mathtt{1}\}$.
2. $\Sigma^*\mathtt{1}\Sigma^* = \{w \mid w$ tem pelo menos um $\mathtt{1}\}$.
3. $\Sigma^*\mathtt{001}\Sigma^* = \{w \mid w$ contém a cadeia $\mathtt{001}$ como uma subcadeia$\}$.
4. $(\Sigma\Sigma)^* = \{w \mid w$ é uma cadeia de comprimento par$\}$.[5]
5. $(\Sigma\Sigma\Sigma)^* = \{w \mid$ o comprimento de w é um múltiplo de três$\}$.
6. $\mathtt{01} \cup \mathtt{10} = \{\mathtt{01}, \mathtt{10}\}$.
7. $\mathtt{0}\Sigma^*\mathtt{0} \cup \mathtt{1}\Sigma^*\mathtt{1} \cup \mathtt{0} \cup \mathtt{1} = \{w \mid w$ começa e termina com o mesmo símbolo$\}$.
8. $(\mathtt{0} \cup \varepsilon)\mathtt{1}^* = \mathtt{01}^* \cup \mathtt{1}^*$.

   A expressão $\mathtt{0} \cup \varepsilon$ descreve a linguagem $\{0, \varepsilon\}$, de modo que a operação de concatenação adiciona $\mathtt{0}$ ou ε antes de toda cadeia em $\mathtt{1}^*$.
9. $(\mathtt{0} \cup \varepsilon)(\mathtt{1} \cup \varepsilon) = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}\}$.
10. $\mathtt{1}^*\emptyset = \emptyset$.

    Concatenar o conjunto vazio a um conjunto qualquer resulta no conjunto vazio.
11. $\emptyset^* = \varepsilon$.

    A operação estrela junta qualquer número de cadeias da linguagem para obter uma cadeia no resultado. Se a linguagem é vazia, a operação estrela pode juntar 0 cadeias, dando apenas a cadeia vazia.

Se supormos que R é uma expressão regular qualquer, temos as seguintes identidades. Elas são bons testes para ver se você entende a definição.

$R \cup \emptyset = R$
Adicionar a linguagem vazia a qualquer outra linguagem não a modificará.

$R \circ \varepsilon = R$.
Adicionar a cadeia vazia a qualquer cadeia não a modificará.

Entretanto, trocando $\emptyset$ por ε e vice-versa nas identidades acima pode fazer as igualdades falharem.

$R \cup \varepsilon$ pode não ser igual a R.
Por exemplo, se $R = \mathtt{0}$, então $L(R) = \{\mathtt{0}\}$ mas $L(R \cup \varepsilon) = \{\mathtt{0}, \varepsilon\}$.

$R \circ \emptyset$ pode não ser igual a R.
Por exemplo, se $R = \{\mathtt{0}\}$, então $L(R) = \{\mathtt{0}\}$ mas $L(R \circ \emptyset) = \emptyset$.

Expressões regulares são ferramentas úteis no desenho de compiladores para linguagens de programação. Objetos elementares em uma linguagem de programação, chamados de ***tokens***, tais como nomes de variáveis e de constantes, podem ser descritos com expressões regulares. Por exemplo, uma constante numérica que pode incluir uma parte fracionária e/ou um sinal pode ser descrita como um membro da linguagem

$$(+ \cup - \cup \varepsilon)(DD^* \cup DD^*\mathtt{.}D^* \cup D^*\mathtt{.}DD^*),$$

[5] O *comprimento* de uma cadeia é o número de símbolos que ela contém.

onde $D = \{0, 1, 2, 3.4, 5, 6, 7, 8, 9\}$ é o alfabeto de dígitos decimais. Exemplos de cadeias geradas são: `72`, `3,14159`, `+7,`, e `-,01`.

Uma vez que a sintaxe dos tokens da linguagem de programação tenha sido descrita com expressões regulares, sistemas automáticos podem gerar o ***analisador léxico***, a parte de um compilador que inicialmente processa o programa de entrada.

Equivalência com autômatos finitos

Expressões regulares e autômatos finitos são equivalentes no seu poder descritivo. Esse fato é um tanto notável, porque autômatos finitos e expressões regulares aparentam superficialmente ser um tanto diferentes. Entretanto, qualquer expressão regular pode ser convertida em um autômato finito que reconhece a linguagem que ela descreve, e vice-versa. Lembre-se que uma linguagem regular é aquela que é reconhecida por algum autômato finito.

Teorema 1.28 . . . . . .
Uma linguagem é regular se e somente se alguma expressão regular a descreve.

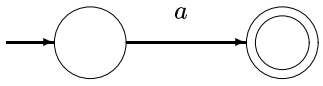
Esse teorema tem duas direções. Enunciamos e provamos cada direção como um lema separado.

Lema 1.29 . . . . . .
Se uma linguagem é descrita por uma expressão regular, então ela é regular.

. . . . . .
Idéia da prova. Digamos que temos uma expressão regular R descrevendo uma certa linguagem A. Mostramos como converter R em um AFN que reconhece A. Pelo Corolário 1.20, se um AFN reconhece A então A é regular.

Prova. Vamos converter R em um AFN N. Consideramos os seis casos na definição formal de expressões regulares.

1. $R = a$ para algum $a \in \Sigma$. Então $L(R) = \{a\}$, e o AFN abaixo reconhece $L(R)$.

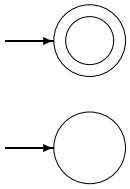


   Note que essa máquina se encaixa na definição de um AFN mas não na de um AFD pois ela tem alguns estados sem seta saindo para cada símbolo de entrada possível. É claro que poderíamos ter apresentado aqui um AFD equivalente mas um AFN é tudo do que precisamos por agora, e é mais fácil descrever.

   Formalmente, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, onde descrevemos δ dizendo que $\delta(q_1, a) = \{q_2\}$, $\delta(r, b) = \emptyset$ para $r \neq q_1$ ou $b \neq a$.

2. $R = \varepsilon$. Então $L(R) = \{\varepsilon\}$, e o seguinte AFN reconhece $L(R)$.

   Formalmente, $N = (\{q_1\}, \Sigma, \delta, q - 1, \{q_1\})$, onde $\delta(r, b) = \emptyset$ para qualquer r e b.



3. $R = \emptyset$. Então $L(R) = \emptyset$, e o seguinte AFN reconhece $L(R)$.

   Formalmente $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, onde $\delta(r, b) = \emptyset$ para qualquer r e b.

4. $R = R_1 \cup R_2$.

5. $R = R_1 \circ R_2$.

6. $R = R_1^*$.

Para os três últimos casos usamos as construções dadas nas provas de que a classe das linguagens regulares é fechada sob as operações regulares. Em outras palavras, construimos o AFN para R a partir dos AFN's para R_1 e R_2 (ou somente R_1 no caso 6) e a construção de fecho apropriada.

. . . . . .

Isso conclui a primeira parte da prova do Teorema 1.28, dando a direção mais fácil do se e somente se. Antes de prosseguir para a outra direção vamos considerar alguns exemplos através dos quais usamos esse procedimento para converter uma expressão regular num AFN.

Exemplo 1.30 . . . . . .
Convertemos a expressão regular $(\mathsf{ab} \cup \mathsf{a})^*$ em um AFN em uma seqüência de estágios. Montamos a partir das subexpressões menores para as subexpressões maiores até que tenhamos um AFN para a expressão original, como mostrado no diagrama abaixo. Note que esse procedimento geralmente não dá o AFN com o mínimo de estados. Neste exemplo, o procedimento dá um AFN com oito estados, mas o menor AFN equivalente tem apenas dois estados. Você pode encontrá-lo?

Exemplo 1.31 . . . . . .
Neste segundo exemplo convertemos a expressão regular $(\mathsf{a} \cup \mathsf{b})^*\mathsf{aba}$ em um AFN. (Veja Figura 1.28.) Alguns dos passos intermediários não são mostrados.

Agora vamos nos voltar para a outra direção da prova do Teorema 1.28.

Lema 1.32 . . . . . .
Se uma linguagem é regular, então ela é descrita por uma expressão regular.

. . . . . .
Idéia da prova. Precisamos mostrar que, se uma linguagem A é regular, uma expressão regular a descreve. Devido ao fato de que A é regular, ela é aceita por um

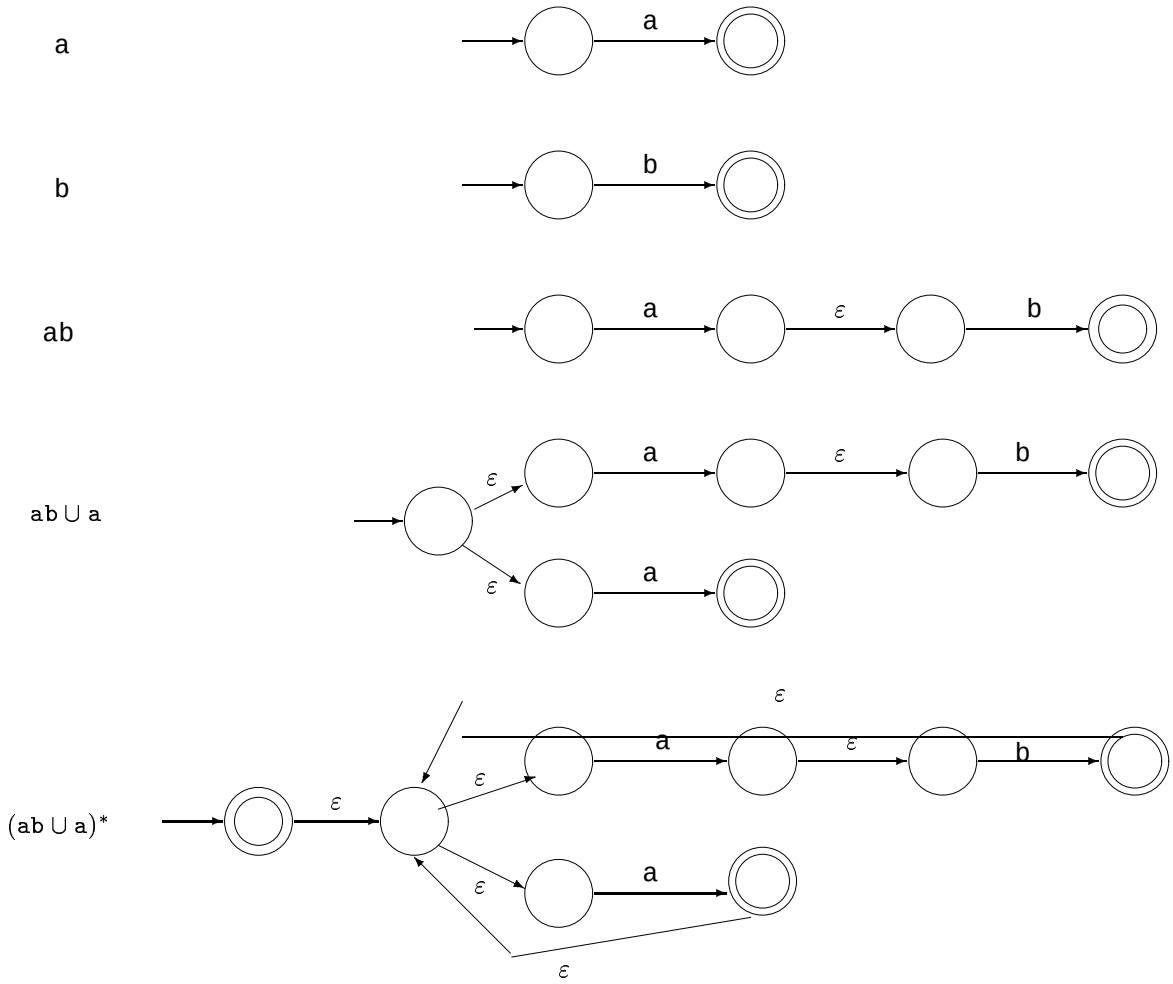


Figura 1.27: Construindo um AFN a partir da expressão regular $(\mathtt{ab} \cup \mathtt{a})^*$

AFD. Descrevemos um procedimento para converter AFD's em expressões regulares equivalentes.

Dividimos esse procedimento em duas partes, usando um novo tipo de autômato finito chamado de ***autômato finito não-determinístico generalizado***, AFNG. Primeiro mostramos como converter AFD's em AFNG's e daí AFNG's em expressões regulares.

Autômatos finitos não-determinísticos generalizados são simplesmente autômatos finitos não-determinísticos nos quais as setas de transição podem ter qualquer expressão regular como rótulo, ao invés de apenas membros do alfabeto ou ε. O AFNG lê blocos de símbolos da entrada, não necessariamente apenas um símbolo a cada vez como num AFN comum. O AFNG se move ao longo de uma seta de transição conectando dois estados após ler um bloco de símbolos da entrada, o qual constitui uma cadeia descrita pela expressão regular sobre aquela seta. Um AFNG é não-determinístico e portanto pode ter várias maneiras diferentes de processar a mesma cadeia de entrada. Ele aceita sua entrada se seu processamento pode levar o AFNG a estar em um estado de aceitação no final da entrada. A Figura 1.29 apresenta um exemplo de um AFNG.

Por conveniência requeremos que AFNG's sempre tenham um formato especial que atende às seguintes condições:

- O estado inicial tem setas de transição indo para todos os outros estados mas nenhuma chegando de nenhum outro estado.
- Existe apenas um único estado de aceitação, e ele tem setas vindo de todos os outros estados mas nenhuma seta saindo para nenhum outro estado. Além disso, o estado de aceitação não é o mesmo que o estado inicial.
- Exceto os estados inicial e de aceitação, uma seta vai de todo estado para todos os outros estados e de cada estado para si próprio.

Podemos facilmente converter um AFD em um AFNG no formato especial. Simplesmente adicionamos um novo estado inicial com uma seta ε ao antigo estado inicial e um novo estado de aceitação com setas ε a partir dos antigos estados de aceitação. Se quaisquer setas têm múltiplos rótulos (ou se existem múltiplas setas indo entre os mesmos dois estados na mesma direção), substituimos cada uma por uma única seta cujo rótulo é a união dos rótulos anteriores. Finalmente, adicionamos setas rotuladas com $\emptyset$ entre estados que não tinham setas. Esse último passo não modificará a linguagem reconhecida porque uma transição rotulada com $\emptyset$ nunca pode ser usada. Daqui por diante assumimos que todos os AFNG's estão no formato especial.

Agora vamos mostrar como converter um AFNG em uma expressão regular. Digamos que o AFNG tem k estados. Então, devido ao fato de que um AFNG tem que ter um estado inicial e um estado de aceitação e eles têm que ser diferentes um do outro, sabemos que $k \geq 2$. Se $k > 2$, construimos um AFNG equivalente com $k - 1$ estados. Esse passo pode ser repetido sobre o novo AFNG até ele ser reduzido a dois estados. Se $k = 2$, o AFNG tem uma única seta que vai do estado inicial para o estado de aceitação. O rótulo dessa seta é a expressão regular equivalente. Por exemplo, os estágios na conversão de um AFD com três estados para uma expressão regular equivalente são mostrados na Figura 1.30.

O passo crucial é na construção de um AFNG equivalente com um estado a menos quando $k > 2$. Fazemos isso selecionando um estado, removendo-o da máquina, e reparando o restante de modo que a mesma linguagem seja ainda reconhecida. Qualquer estado serve, desde que ele não seja o estado inicial ou o estado de aceitação. Temos garantido de que tal estado existirá pois $k > 2$. Vamos chamar esse estado removido q_{rem}.

Após remover q_{rem} reparamos a máquina alterando as expressões regulares que rotulam cada uma das setas remanescentes. Os novos rótulos compensam a ausência de q_{rem} adicionando de volta as computações perdidas. O novo rótulo indo de um estado q_i para um estado q_j é uma expressão regular que descreve todas as cadeias que levariam a máquina de q_i para q_j ou diretamente ou via q_{rem}. Ilustramos essa abordagem na Figura 1.31.

Na máquina antiga se q_i vai de q_{rem} com uma seta rotulada R_1, q_{rem} vai para si próprio com uma seta rotulada R_2, q_{rem} vai para q_j com uma seta rotulada R_3, e q_i vai para q_j com uma seta rotulada R_4, então na nova máquina a seta de q_i para q_j recebe o rótulo

$$(R_1)(R_2)^*(R_3) \cup (R_4).$$

Fazemos essa mudança para cada seta indo de qualquer estado q_i para qualquer estado q_j, incluindo o caso em que $q_i = q_j$. A nova máquina reconhece a linguagem original.

Prova. Vamos agora realizar essa idéia formalmente. Primeiro, para facilitar a prova, definimos formalmente o novo tipo de autômato introduzido. Um AFNG é semelhante a um autômato finito não-determinítico exceto pela função de transição, que tem a forma

$$\delta : (Q - \{q_{\text{aceita}}\}) \times (Q - \{q_{\text{inicio}}\}) \longrightarrow \mathcal{R}.$$

O símbolo $\mathcal{R}$ é a coleção de todas as expressões regulares sobre o alfabeto Σ, e q_{aceita} e q_{inicio} são os estados inicial e de aceitação. Se $\delta(q_i, q_j) = R$, a seta do estado q_i para o estado q_j tem a expressão regular R como seu rótulo. O domínio da função de transição é $(Q - \{q_{\text{aceita}}\}) \times (Q - \{q_{\text{inicio}}\})$ porque uma seta conecta todo estado para todos os outros estados, exceto que nenhuma seta está vindo de q_{aceita} ou chegando em q_{inicio}.

Definição 1.33 . . . . . .
Um ***autômato finito não-determinístico generalizado***, $(Q, \Sigma, \delta, q_{\text{inicio}}, q_{\text{aceita}})$, é uma 5-upla onde

1. Q é o conjunto finito de estados,
2. Σ é o alfabeto de entrada,
3. $\delta : (Q - \{q_{\text{aceita}}\}) \times (Q - \{q_{\text{inicio}}\}) \longrightarrow \mathcal{R}$ é a função de transição,
4. q_{inicio} é o estado inicial, e
5. q_{aceita} é o estado de aceitação.

Um AFNG aceita uma cadeia w em Σ^* se $w = w_1 w_2 \cdots w_k$, onde cada w_i está em Σ^* e uma seqüência de estados $q_0, q_1, \ldots, q_k$ existe tal que

1. $q_0 = q_{\text{inicio}}$ é o estado inicial,
2. $q_k = q_{\text{aceita}}$ é o estado de aceitação, e
3. para cada i, temos $w_i \in L(R_i)$, onde $R_i = \delta(q_{i-1}, q_i)$; em outras palavras, R_i é a expressão sobre a seta de q_{i-1} para q_i.

Voltando à prova do Lema 1.32, supomos que M seja o AFD para a linguagem A. Então convertemos M para um AFNG G adicionando um novo estado inicial e um novo estado de aceitação e setas adicionais de transição conforme necessário. Usamos o procedimento $CONVERT(G)$, que toma um AFNG como entrada e retorna uma expressão regular equivalente. Esse procedimento usa ***recursão***, o que significa que ele chama a si próprio. Um laço infinito é evitado porque o procedimento chama a si próprio somente para processar o AFNG que tem um estado a menos. O caso em que o AFNG tem dois estados é tratado sem recursão.

$CONVERT(G)$:

1. Seja k o número de estados de G.

2. Se $k = 2$, então G tem que consistir de um estado inicial, um estado inicial, e uma única seta conectando-os e rotulada com uma expressão regular R.

   Retorne a expressão R.

3. Se $k > 2$, selecionamos qualquer estado $q_{\text{rem}} \in Q$ diferente de q_{inicio} e q_{aceita}, e montamos G' como sendo o AFNG $(Q', \Sigma, \delta', q_{\text{inicio}}, q_{\text{aceita}})$, onde

   $$Q' = Q - \{q_{\text{rem}}\},$$

   e para qualquer $q_i \in Q' - \{q_{\text{aceita}}\}$ e qualquer $q_j \in Q' - \{q_{\text{inicio}}\}$ faça

   $$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$

   para $R_1 = \delta(q_i, q_{\text{rem}})$, $R_2 = \delta(q_{\text{rem}}, q_{\text{rem}})$, $R_3 = \delta(q_{\text{rem}}, q_j)$, e $R_4 = \delta(q_i, q_j)$.

4. Calcule $CONVERT(G')$ e retorne esse valor.

Agora provamos que $CONVERT(G)$ retorna o valor correto.

Afirmação 1.34 .....
Para qualquer AFNG G, $CONVERT(G)$ é equivalente a G.

Provamos essa afirmação por indução sobre k, o número de estados do AFNG.

Base. Prove que a afirmação é verdadeira para $k = 2$ estados. Se G tem apenas dois estados, ele pode ter apenas uma única seta, que vai do estado inicial para o estado de aceitação. O rótulo sob forma de expressão regular sobre essa seta descreve todas as cadeias que permitem que G chegue ao estado de aceitação. Donde essa expressão é equivalente a G.

Passo indutivo. Assuma que a afirmação é verdadeira para $k - 1$ estados e use essa suposição para provar que a afirmação é verdadeira para k estados. Primeiro mostramos que G e G' reconhecem a mesma linguagem. Suponha que G aceita uma entrada w. Então em um ramo de aceitação da computação G entra numa seqüência de estados

$$q_{\text{inicio}}, q_1, q_2, q_3, \ldots, q_{\text{aceita}}.$$

Se nenhum deles é o estado removido q_{rem}, claramente G' também aceita w. A razão é que cada uma das novas expressões regulares rotulando as setas de G' contém a expressão regular antiga como parte de uma união.

Se q_{rem} realmente aparece, removendo cada rodada e estados q_{rem} consecutivos forma uma computação de aceitação para G'. Os estados q_i e q_j iniciando e terminando uma rodada têm uma nova expressão regular sobre a seta entre eles que descreve todas as cadeias levando q_i para q_j via q_{rem} sobre G. Portanto G' aceita w.

Para a outra direção, suponha que G' aceita uma cadeia w. Como cada seta entre quaisquer dois estados q_i e q_j em G' descreve a coleção de cadeias levando q_i para q_j em G, ou diretamente ou via q_{rem}, G tem que aceitar w também. Por conseguinte G e G' são equivalentes.

A hipótese de indução enuncia que quando o algoritmo chama a si próprio recursivamente sobre a entrada G', o resultado é uma expressão regular que é equivalente a G' porque G' tem $k - 1$ estados. Daí a expressão regular também é equivalente a G, e o algoritmo está provado correto.

Isso conclui a prova da Afirmação 1.34, Lema 1.32, e Teorema 1.28.

.....

Exemplo 1.35 . . . . . .
Neste exemplo usamos o algoritmo precedente para converter um AFD em uma expressão regular. Começamos com o AFD de dois-estados na Figura 1.32(a).

Em (b) montamos um AFNG de quatro-estados adicionando um novo estado inicial e um novo estado de aceitação, chamados i e a ao invés de q_{inicio} e q_{aceita} de modo que podemos desenhá-los convenientemente. Para evitar carregar demasiadamente a figura, não desenhamos as setas que são rotuladas $\emptyset$, muito embora elas estejam presentes. Note que substituimos o rótulo a, b sobre o auto-laço no estado 2 do AFD pelo rótulo a∪b no ponto correspondente sobre o AFNG. Fazemos isso porque o rótulo do AFD representa duas transições, uma para a e outra para b, enquanto que o AFNG pode ter somente uma única transição saindo de 2 para si próprio.

Em (c) removemos o estado 2, e atualizamos os rótulos de setas remanescentes. Neste caso, o único rótulo que muda é aquele de 1 para a. Em (b) era $\emptyset$, mas em (c) é b(a∪b)*. Obtemos esse resultado seguindo o passo 3 do procedimento $CONVERT$. O estado q_i éo estado 1, o estado q_j é a, e q_{rem} é 2, portanto $R_1 = \mathsf{b}$, $R_2 = \mathsf{a}\cup\mathsf{b}$, $R_3 = \varepsilon$, e $R_4 = \emptyset$. Por conseguinte o novo rótulo sobre a seta de 1 para a é $(\mathsf{b})(\mathsf{a}\cup\mathsf{b})^*(\varepsilon)\cup\emptyset$. Simplificamos essa expressão regular para $\mathsf{b}(\mathsf{a}\cup\mathsf{b})^*$.

Em (d) removemos o estado 1 de (c) e seguimos o mesmo procedimento. Devido ao fato de que somente os estados inicial e de aceitação permanecem, o rótulo sobre a seta que os conecta é a expressão regular que é equivalente ao AFD original.

Exemplo 1.36 . . . . . .
Neste exemplo começamos com um AFD de três-estados. Os passos na conversão aparecem na Figura 1.33.

1.4 Linguagens não-regulares . . . . . .

Para entender o poder de autômatos finitos você também tem que entender suas limitações. Nesta seção mostramos como provar que certas linguagens não podem ser reconhecidas por nenhum autômato finito.

Vamos tomar a linguagem $B = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. Se tentarmos contrar um AFD que reconhece B, descobrimos que a máquina parece necessitar de memorizar quantos 0's foram vistos até então à medida que ela lê a entrada. Devido ao fato de que o número de 0's não é limitado, a máquina terá que manter registro de um número ilimitado de possibilidades. Mas ela não pode fazer com um nenhuma quantidade finita de estados.

A seguir apresentamos um método para provar que linguagens como B não são regulares. O argumento que acaba de ser dado não já prova não-regularidade, pois o número de 0's é ilimitado? Não, naõ prova. Simplesmente porque a linguagem parece requerer uma quantidade ilimitada de memória não significa que ela necessariamente é assim. Acontece de ser verdadeiro para a linguagem B, mas outras linguagens parecem requerer um número ilimitado de possibilidades, e ainda assim na verdade são regulares. Por exemplo, considere duas linguagens sobre o alfabeto $\Sigma = \{\mathtt{0}, \mathtt{1}\}$:

$C = \{w \mid w$ tem o mesmo número de 0's e 1's$\}$, e

$D = \{w \mid w$ tem o mesmo número de ocorrências de 01 e 10 como subcadeias$\}$.

À primeira vista uma máquina reconhecedora aparenta necessitar contar em cada caso, e por conseguinte nenhuma linguagem aparenta ser regular. Como esperado, C não é regular, mas surpreendentemente D é regular![6] Por conseguinte nossa intuição pode às vezes nos levar para fora dos trilhos, motivo pelo qual precisamos de provas matemáticas para ter certeza. Nesta seção mostramos como provar que certas linguagens não são regulares.

O lema do bombeamento para linguagens regulares

Nossa técnica para provar não-regularidade provém de um teorema sobre linguagens regulares, tradicionalmente chamado ***lema do bombeamento***. Esse teorema enuncia que todas as linguagens regulares têm uma propriedade especial. Se pudermos mostrar que uma linguagem não tem essa propriedade, estamos garantidos de que ela não é regular. A propriedade enuncia que todas as cadeias na linguagem podem ser "bombeadas" se elas são no mínimo tão longas quanto um certo valor, chamado o ***comprimento de bombeamento***. Isso significa que cada cadeia dessa contém uma parte que pode ser repetida um número qualquer de vezes com a cadeia resultante permanecendo na linguagem.

Teorema 1.37

Lema do bombeamento Se A é uma linguagem regular, então existe um número p (o comprimento de bombeamento) onde, se s é uma cadeia qualquer de A de comprimento pelo menos p, então s pode ser dividida em três partes, $s = xyz$, satisfazendo as seguintes condições:

1. para cada $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, e
3. $|xy| \leq p$.

Lembre-se da notação onde $|s|$ representa o comprimento da cadeia s, y^i significa que i cópias de y são concatenadas umas às outras, e y^0 é igual a ε.

Quando s é dividida em xyz, ou x ou z pode ser ε, mas a condição 2 diz que $y \neq \varepsilon$. Observe que sem essa condição 2 o teorema seria trivialmente verdadeiro. A condição 3 enuncia que as partes x e y juntas têm comprimento no máximo p. É uma condição técnica extra que ocasionalmente achamos útil ao provar que certas linguagens não são regulares. Veja o Exemplo 1.39 para uma aplicação da condição 3.

Idéia da prova. Seja $M = (Q, \Sigma, \delta, q_1, F)$ um AFD que reconhece A. Atribuimos o comprimento de bombeamento p como sendo o número de estados de M. Mostramos que qualquer cadeia s em A de comprimento pelo menos p pode ser quebrada nas três partes xyz satisfazendo nossas três condições. E se nenhuma cadeia em A é de comprimento pelo menos p? Então nossa tarefa é ainda mais fácil porque o teorema se torna *vacuamente* verdadeiro: obviamente as três condições se verificam para todas as cadeias de comprimento pelo menos p se não existem tais cadeias.

Se s em A tem comprimento pelo menos p, considere a seqüência de estados pelos quais M passa quando computando com a entrada s. Ele começa com o estado inicial q_1, e então vai para, digamos, q_3, e daí para, digamos, q_{20}, e então para q_9, e assim por

[6] Veja o Problema 1.41.

diante, até que ele atinge o final de s no estado q_{13}. Com s em A, sabemos que M aceita s, portanto q_{13} é um estado de aceitação.

Se dissermos que n é o comprimento de s, a seqüência de estados $q_1, q_3, q_{20}, q_9, \ldots,$ q_{13} tem comprimento $n+1$. Devido ao fato de que n é pelo menos p, sabemos que $n+1$ é maior que p, o número de estados de M. Por conseguinte a seqüência tem que conter um estado repetido. Esse resultado é um exemplo do ***princípio da casa-de-pombos***, um nome pomposo para o fato um tanto óbvio de que se p pombos são colocados em menos que p casas, alguma casa tem que ter mais que um pombo.

A Figura 1.34 mostra a cadeia s e a seqüência de estados pelos quais M passa quando processando s. O estado q_9 é um que se repete.

Agora dividimos s em três partes x, y, e z. A parte x é a parte de s que aparece antes de q_9, a parte y é a parte entre as duas aparições de q_9, e a parte z é a parte remanescente de s, vindo após a segunda ocorrência de q_9. Portanto x leva M do estado q_1 para q_9, y leva M de q_9 de volta para q_9 e z leva M de q_9 para o estado de aceitação q_{13}, como mostrado na Figura 1.35.

Vamos ver por que essa divisão satisfaz as três condições. Suponha que rodemos M sobre a entrada $xyyz$. Sabemos que x leva M de q_1 para q_9, e então o primeiro y o leva de q_9 de volta a q_9, o mesmo fazendo o segundo y, e então z o leva para q_{13}. Com q_{13} sendo um estado de aceitação, M aceita a cadeia $xyyz$. Igualmente, ele aceitará xy^iz para qualquer $i > 0$. Para o caso $i = 0$, $xy^iz = xz$, que é aceita por razões semelhantes. Isso estabelece a condição 1.

Verificando a condição 2, vemos que $|y| > 0$, pois era a parte de s que ocorria entre duas ocorrências diferentes do estado q_9.

De modo a obter a condição 3, asseguramos que q_9 seja a primeira repetição na seqüência. Pelo princípio da casa-de-pombos, os primeiros $p + 1$ estados na seqüência têm que conter uma repetição. Por conseguinte $|xy| \leq p$.

Prova. Seja $M = (Q, \Sigma, \delta, q_1, F)$ um AFD que reconhece A e p o número de estados de M.

Seja $s = s_1 s_2 \cdots s_n$ uma cadeia em A de comprimento n, onde $n \geq p$. Seja $r_1, \ldots, r_{n+1}$ a seqüência de estados nos quais M entra enquanto processa s, portanto $r_{i+1} = \delta(r_i, s_i)$ para $1 \leq i \leq n$. Essa seqüência tem comprimento $n + 1$, o que é pelo menos $p + 1$. Entre os primeiros $p + 1$ elementos na seqüência, dois têm que ser o mesmo estado, pelo princípio da casa-de-pombos. Chamamos o primeiro desses de r_j e o segundo de r_j. Devido ao fato de que r_l ocorre entre as primeiras $p + 1$ posições na seqüência começando com r_1, temos $l \leq p + 1$. Agora faça $x = s_1 \cdots s_{j-1}$, $y = s_j \cdots s_{l-1}$, e $z = s_l \cdots s_n$.

Como x leva M de r_1 para r_j, y leva M de r_j para r_j, e z leva M de r_j para r_{n+1}, que é um estado de aceitação, M tem que aceitar xy^iz para $i \geq 0$. Sabemos que $j \neq l$, portanto $|y| > 0$; e $l \leq p + 1$, logo $|xy| \leq p$. Por conseguinte satisfizemos todas as condições do lema do bombeamento.

. . . . . .

Para usar o lema do bombeamento para provar que uma linguagem B não é regular, primeiro assuma que B é regular de modo a obter uma contradição. Então use o lema do bombeamento para garantir a existência de um comprimento de bombeamento p tal

que todas as cadeias de comprimento p ou mais em B podem ser bombeadas. A seguir, encontre uma cadeia s em B que tem comprimento p ou mais porém que não pode ser bombeada. Finalmente, demonstre que s não pode ser bombeada considerando todas as maneiras de se dividir x em x, y, e z (levando a condição 3 do lema do bombeamento em consideração se conveniente) e, para cada tal divisão, encontrando um valor i onde $xy^iz \notin B$. Esse passo final frequentemente envolve agrupar as várias maneiras de se dividir s em vários outros casos e analisá-los individualmente. A existência de s contradiz o lema do bombeamento se B fosse regular. Donde B não pode ser regular.

Encontrar s às vezes requer um pouco de pensamento criativo. Você pode precisar tentar vários candidatos para s antes de você descobrir um que funciona. Tente membros de B que parecem exibir a "essência" da não-regularidade de B. Discutimos mais a tarefa de encontrar s em alguns dos exemplos a seguir.

Exemplo 1.38 .....

Seja B a linguagem $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. Usamos o lema do bombeamento para provar que B não é regular. A prova é por contradição.

Assuma, ao contrário, que B é regular. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Escolha s como sendo a cadeia $\mathtt{0}^p\mathtt{1}^p$. Devido ao fato de que s é um membro de B e que s tem comprimento maior que p, o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, onde para qualquer $i \geq 0$ a cadeia xy^iz está em B. Consideramos três casos para mostrar que esse resultado é impossível.

1. A cadeia y consiste de apenas `0`'s. Neste caso a cadeia xyz tem mais `0`'s que `1`'s e portanto não é um membro de B, violando a condição 1 do lema do bombeamento. Este caso é uma contradição.

2. A cadeia y consiste de somente `1`'s. Este caso também dá uma contradição.

3. A cadeia y consiste de `0`'s e `1`'s. Neste caso a cadeia $xyyz$ pode ter o mesmo número de `0`'s e `1`'s, mas eles estarão fora de ordem com alguns `1`'s antes de `0`'s. Daí ela não é um membro de B, o que é uma contradição.

Por conseguinte uma contradição é inevitável se tomamos a suposição de que B é regular, portanto B não é regular.

Neste exemplo, encontrar a cadeia s foi fácil, porque qualquer cadeia em B de comprimento p ou mais funcionaria. Nos próximos dois exemplos algumas escolhas para s não funcionam, portanto cuidado adicional é necessário.

Exemplo 1.39 .....

Seja $C = \{w \mid w$ tem o mesmo número de `0`'s e `1`'s$\}$. Usamos o lema do bombeamento para provar que C não é regular. A prova é por contradição.

Assuma, ao contrário, que C é regular. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Como no Exemplo 1.38, seja s a cadeia $\mathtt{0}^p\mathtt{1}^p$. Com s sendo um membro de C e tendo comprimento maior que p, o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, onde para qualquer $i \geq 0$ a cadeia xy^iz está em C. Gostaríamos de mostrar que esse resultado é impossível. Mas espere, ele *é* possıvel! Se fizermos x e z serem a cadeia vazia e y ser a cadeia $\mathtt{0}^p\mathtt{1}^p$, então xy^iz sempre terá o mesmo número de `0`'s e de `1`'s e portanto está em C. Portanto *parece* que s pode ser bombeada.

Aqui a condição 3 no lema do bombeamento é útil. Ela estipula que quando bombear s ela tem que ser dividida de tal modo que $|xy| p$. Essa restrição na forma em que s pode ser dividida torna mais fácil mostrar que a cadeia $s = \mathtt{0}^p\mathtt{1}^p$ que selecionamos não pode ser bombeada. Se $|xy| \leq p$, então y tem que consistir somente de 0's, portanto $xyyz \notin C$. Por conseguinte s não pode ser bombeada. Isso nos dá a contradição desejada.[7]

Selecionar a cadeia s neste exemplo exigiu mais cuidado do que no Exemplo 1.38. Se, ao contrário, tivéssemos escolhido $s = (\mathtt{01})^p$, teríamos esbarrado em problemas pois precisamos de uma cadeia que *não pode* ser bombeada e essa cadeia *pode* ser bombeada, mesmo levando em conta a condição 3. Você pode ver como bombeá-la? Uma maneira de fazê-lo é $x = \varepsilon$, $y = \mathtt{01}$, e $z = (\mathtt{01})^{p-1}$. Então $xy^iz \in C$ para todo valor de i. Se você falhar na sua primeira tentativa de encontrar uma cadeia que não pode ser bombeada, não se desespere. Tente outra!

Um método alternativo de provar que C é não-regular segue de nosso conhecimento de que B é não-regular. Se C fosse regular, $C \cap \mathtt{0}^*\mathtt{1}^*$ também seria regular. As razões são que a linguagem $\mathtt{0}^*\mathtt{1}^*$ é regular e que a classe das linguagens regulares é fechada sob interseção (provado na nota de pé de página na página ??). Mas $C \cap \mathtt{0}^*\mathtt{1}^*$ é igual a B, e sabemos que B é não-regular do Exemplo 1.38.

Exemplo 1.40 .....

Seja $F = \{ww \mid w \in \{\mathtt{0},\mathtt{1}\}^*\}$. Mostramos que F é não-regular usando o lema do bombeamento.

Assuma, ao contrário, que F é regular. Seja p o comprimento do lema do bombeamento dado pelo lema do bombeamento. Seja s a cadeia $\mathtt{0}^p\mathtt{01}^p\mathtt{1}$. Devido ao fato de que s é um membro de F e s tem comprimento maior que p, o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, satisfazendo as três condições do lema. Mostramos que esse resultado é impossível.

A condição 3 é novamente crucial, pois sem ela poderíamos bombear s se fizéssemos x e z ser a cadeia vazia. Com a condição 3 a prova segue porque y tem que consistir somente de 0's, portanto $xyyz \notin F$.

Observe que escolhemos $s = \mathtt{0}^p\mathtt{10}^p\mathtt{1}$ como sendo uma cadeia que exibe a "essência" da não-regularidade de F, ao invés de, digamos, a cadeia $\mathtt{0}^p\mathtt{0}^p$. Muito embora $\mathtt{0}^p\mathtt{0}^p$ seja um membro de F, ela não serve para demonstrar uma contradição porque ela pode ser bombeada.

Exemplo 1.41 .....

Aqui demonstramos uma linguagem unária não-regular. Seja $D = \{\mathtt{1}^{n^2} \mid n \geq 0\}$. Em outras palavras, D contém todas as cadeias de 1's cujo comprimento é um quadrado perfeito. Usamos o lema do bombeamento para provar que D não é regular. A prova é por contradição.

Assuma, ao contrário, que D é regular. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Seja s a cadeia $\mathtt{1}^{p^2}$. Devido ao fato de que s é um membro de D e s tem comprimento pelo menos p, o lema do bombeamento garante que s pode ser dividida em três partes, $s = xyz$, onde para qualquer $i \geq 0$ a cadeia xy^iz está em D. Como nos exemplos precedentes, mostramos que esse resultado é impossível. Fazer isso neste caso requer um pouco mais de raciocínio com a seqüência de quadrados perfeitos:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

[7] Poderíamos ter usado a condição 3 no Exemplo 1.38 também, para simplificar sua prova.

Note a crescente distância entre membros sucessivos dessa seqüência. Membros grandes dessa seqüência não podem estar próximos um do outro.

Agora considere as duas cadeias xy^iz e $xy^{i+1}z$. Essas cadeias diferem entre si de uma única repetição de y, e consequentemente seus comprimentos diferem entre si do comprimento de y. Se escolhermos i muito grande, os comprimentos de xy^iz e $xy^{i+1}z$ não podem ser ambos quadrados perfeitos porque eles estão próximos demais. Por conseguinte xy^iz e $xy^{i+1}z$ não pode estar ambos em D, uma contradição.

Para tornar essa idéia uma prova, calculamos um valor de i que dê a contradição. Se $m = n^2$ é um quadrado perfeito, a diferença entre ele e o próximo quadrado perfeito $(n+1)^2$ é

$$\begin{aligned}(n+1)^2 - n^2 &= n^2 + 2n + 1 - n^2 \\ &= 2n + 1 \\ &= 2\sqrt{m} + 1\end{aligned}$$

O lema do bombeamento enuncia que ambos $|xy^iz|$ e $|xy^{i+1}z|$ são quadrados perfeitos para qualquer i. Mas, fazendo $|xy^iz|$ ser m como acima, vemos que eles *não podem* ser ambos quadrados perfeitos se $|y| < 2\sqrt{|xy^iz|} + 1$ porque eles estariam próximos demais.

Calculando o valor para i que leva a uma contradição agora é fácil. Observe que $|y| \leq |s| = p^2$. Seja $i = p^4$; então

$$\begin{aligned}|y| \leq p^2 &= \sqrt{p^4} \\ &< 2\sqrt{p^4} + 1 \\ &\leq 2\sqrt{|xy^iz|} + 1.\end{aligned}$$

Exemplo 1.41 .....

Às vezes "bombear para baixo" é útil quando aplicamos o lema do bombeamento. Usamos o lema do bombeamento para mostrar que $E = \{\mathtt{0}^i\mathtt{1}^j \mid i > j\}$ não é regular. A prova é por contradição.

Assuma que E é regular. Seja p o comprimento de bombeamento para E dado pelo lema do bombeamento. Seja $s = \mathtt{0}^{p+1}\mathtt{1}^p$. Então s pode ser dividida em xyz, satisfazendo as condições do lema do bombeamento. Pela condição 3, y consiste de apenas `0`'s. Vamos examinar a cadeia $xyyz$ para ver se ela pode estar em E. Adicionando uma cópia extra de y aumenta o número de `0`'s. Mas, E contém todas as cadeias em $\mathtt{0}^*\mathtt{1}^*$ que têm mais `0`'s que `1`'s, portanto aumentando o número de `0`'s ainda dará uma cadeia em E. Nenhuma contradição ocorre. Precisamos tentar algo diferente.

O lema do bombeamento enuncia que $xy^iz \in E$ mesmo quando $i = 0$, portanto vamos considerar a cadeia $xy^0z = xz$. Remover a cadeia y diminui o número de `0`'s em s. Lembre-se que s tem somente um `0` a mais que `1`'s. Por conseguinte, xz não pode ter mais `0`'s que `1`'s, portanto ela não pode ser um membro de E. Por conseguinte obtemos uma contradição.

. . . . . .

Exercícios

1.1 Os diagramas a seguir são diagramas de estado de dois AFD's, M_1 e M_2. Responda às seguintes questões sobre essas máquinas.