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 $\{0, 1\}$.

1. $0^*10^* = \{w \mid w$ tem exatamente um único $1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w$ tem pelo menos um $1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w$ contém a cadeia 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. $01 \cup 10 = \{01, 10\}$.
7. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w$ começa e termina com o mesmo símbolo$\}$.
8. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.

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

    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 $R = 0$, então $L(R) = \{0\}$ mas $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

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

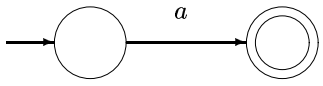
Esse teorema tem duas direções. Enunciamos e provamos cada direção como um lema separado.

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 $L(R)$.

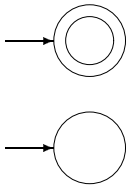


   Note que essa máquina se encaixa na definição de um AFN mas não na de um AFD pois ela tem alguns estados sem seta saindo para cada símbolo de entrada possível. É claro que poderíamos ter apresentado aqui um AFD equivalente mas um AFN é tudo do que precisamos por agora, e é mais fácil descrever.

   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 $(\mathtt{ab} \cup \mathtt{a})^*$ 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 $(\mathtt{a} \cup \mathtt{b})^*\mathtt{aba}$ 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

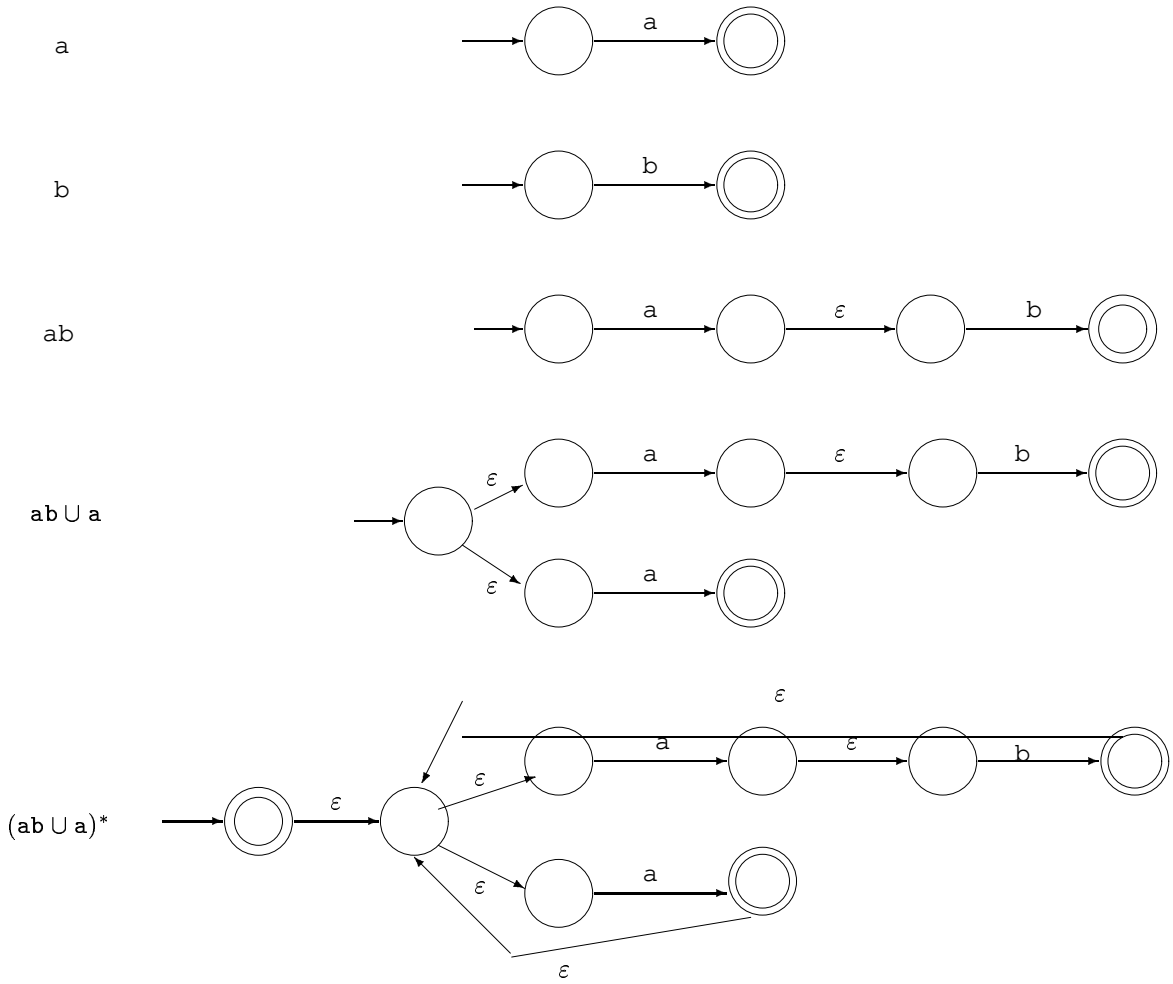


Figura 1.27: Construindo um AFN a partir da expressão regular $(\mathtt{ab} \cup \mathtt{a})^*$

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 $R_1 =$ `b`, $R_2 =$ `a∪b`, $R_3 = \varepsilon$, e $R_4 = \emptyset$. Por conseguinte o novo rótulo sobre a seta de 1 para a é `(b)(a∪b)*`$(\varepsilon)\cup\emptyset$. Simplificamos essa expressão regular para `b(a ∪ 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 $B = \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$. 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 $\Sigma = \{\texttt{0}, \texttt{1}\}$:

$C = \{w \mid w$ 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$\}$.