to é resolvido em favor da palavra-chave. Em geral, esta estratégia de resolução de conflitos torna fácil reservar palavras-chave listando-as à frente dos padrões para os identificadores.

Como um outro exemplo, suponhamos que <= sejam os dois primeiros caracteres lidos. Enquanto o padrão < reconhece o primeiro caractere, o mesmo não é o padrão mais longo que reconhece um prefixo da entrada. Conseqüentemente, a estratégia de Lex em selecionar o prefixo mais longo reconhecido por um padrão facilita a resolução do conflito entre < e <= da forma esperada – pela seleção de <= como o próximo *token*. □

O Operador *Lookahead*

Como vimos na Seção 3.1, os analisadores léxicos para certas construções de linguagem de programação precisam examinar além do fim do lexema antes que um *token* possa ser determinado sem sombra de dúvida. Relembremos o exemplo da Fortran a respeito do par de enunciados

```
DO 5 I = 1.25
DO 5 I = 1,25
```

Como em Fortran os espaços não são significativos fora dos comentários e das cadeias Hollerith, suponhamos, então, que todos os espaços removíveis sejam estirpados antes que a análise léxica comece. Os enunciados acima apareceriam para o analisador léxico como

```
DO5I=1.25
DO5I=1,25
```

No primeiro enunciado, não podemos dizer, até que tenhamos examinado o ponto decimal, que a cadeia `DO` seja parte do identificador `DO5I`. No segundo enunciado, `DO` é uma palavra-chave por si mesma.

Em Lex, podemos escrever um padrão da forma r_1/r_2, onde r_1 e r_2 sejam expressões regulares, significando reconhecer uma cadeia em r_1 somente se seguida por uma cadeia reconhecida em r_2. A expressão regular r_2, após o operador *lookahead* /, indica que contexto à direita deve ser usado no reconhecimento; mas esse contexto deve ser usado apenas para restringir o reconhecimento, não para fazer parte do mesmo. Por exemplo, uma especificação Lex que reconheça a palavra-chave `DO` no contexto acima é

```
DO/({letra} | {dígito})* = ({letra} | {dígito})*,
```

Com essa especificação, o analisador léxico irá procurar à frente em seu *buffer* de entrada por uma seqüência de letras e dígitos seguida por um sinal de igual, seguida por letras e dígitos e seguida por uma vírgula, a fim de assegurar que não haja um enunciado de atribuição (e sim o comando repetitivo `DO`). Então, somente os caracteres `D` e `O`, precedendo o operador *lookahead* /, seriam parte do lexema reconhecido. Após um reconhecimento com sucesso, `yytext` aponta para o `D` e `yyleng` = 2. Note-se que esse simples padrão de esquadrinhamento adiante permite que `DO` seja reconhecido quando seguido por lixo, como `Z4=6Q`, mas jamais irá reconhecer o `DO` que seja parte de um identificador.

Exemplo 3.12. O operador *lookahead* pode ser usado para colaborar em outro difícil problema de análise léxica em Fortran: distinguir palavras-chave e identificadores. Por exemplo, a entrada

```
IF(I, J) = 3
```

é um enunciado de atribuição perfeitamente válido em Fortran e não um enunciado lógico `IF`. Uma forma de especificar a palavra-chave `IF` em Lex é definir seus possíveis contextos à direita usando o operador *lookahead*. A forma simples do enunciado IF lógico é

```
IF(condição) enunciado
```

Fortran 77 introduziu uma outra forma de enunciado IF lógico:

```
IF (condição) THEN
   bloco_then
ELSE
   bloco_else
END IF
```

Notamos que cada enunciado Fortran sem rótulo se inicia por uma letra e que cada parênteses à direita usado para a subscrição ou agrupamento de operandos precisa ser seguido por um símbolo de operação, tal como =, + ou vírgula, outro parênteses à direita ou o fim do enunciado. Um tal parênteses à direita não pode ser seguido por uma letra. Nesta situação, a fim de confirmar que `IF` é uma palavra-chave ao invés de um nome de *array*, esquadrinhamos à frente procurando por um parênteses à direita seguido por uma letra antes de vermos um caractere de avanço de linha (assumimos que os cartões de continuação "cancelam" o caractere de avanço de linha prévio). Esse padrão para a palavra-chave `IF` pode ser escrito como

```
IF/ \(.* \) {letra}
```

O ponto figura no lugar de "qualquer caractere menos avanço de linha" e as barras invertidas à frente dos parênteses informam ao compilador Lex para tratar esses últimos literalmente, não como metassímbolos, ao agrupar expressões regulares (ver Exercício 3.10). □

Outra forma de atacar o problema imposto pelos enunciados `IF` em Fortran é, após enxergar `IF`, determinar se o mesmo foi declarado como um *array*. Esquadrinhamos o padrão completo indicado acima somente se assim o tiver sido declarado. Tais testes tornam mais difícil a implementação automática de um analisador léxico a partir de uma especificação Lex e podem custar tempo numa perspectiva mais ampla, pois verificações freqüentes precisam ser feitas no programa que simula um diagrama de transições, a fim de determinar se algum de tais testes é necessário. Deveria ser notado que tokenizar Fortran é uma tarefa tão irregular que freqüentemente é mais fácil se escrever um analisador léxico *ad hoc* para Fortran numa linguagem convencional de programação do que usar um gerador automático de analisadores léxicos.

3.6 AUTÔMATOS FINITOS

Um *reconhecedor* para uma linguagem é um programa que toma como entrada uma cadeia x e responde "sim" se x for uma sentença da linguagem e "não" em caso contrário. Compilamos expressões regulares num reconhecedor através da construção de um diagrama de transições generalizado chamado de autômato finito. Um autômato finito pode ser determinístico ou não-determinístico, onde "não-determinístico" significa que mais de uma transição para fora de um estado pode ser possível para o mesmo símbolo de entrada.

Tanto os autômatos finitos determinísticos quanto os não-determinísticos são capazes de reconhecer precisamente os conjuntos regulares. Conseqüentemente, ambos podem reconhecer exatamente o que as expressões regulares podem denotar. Entretanto, existe uma barganha tempo-espaço: enquanto os autômatos finitos determinísticos podem levar a reconhecedores mais rápidos do que os não-determinísticos, um autômato finito-determinístico pode ser muito maior do que um autômato finito não determinístico equivalente. Na próxima seção, apresentamos métodos para converter expressões regulares em ambos os tipos de autômatos finitos. A conversão num autômato finito não determinístico é mais direta e, então, discutiremos esses autômatos primeiro.

Os exemplos desta seção e da próxima lidam primariamente com a linguagem denotada pela expressão regular $(a \mid b)^*abb$, consistindo no conjunto de todas as cadeias de a's e b's terminadas em abb. Linguagens similares emergem na prática. Por exemplo, uma expressão

regular para os nomes de todos os arquivos que terminem em .o é da forma (. | o | *c*)*.o, com *c* representando qualquer caractere que não um ponto ou um . o. Como outro exemplo, os comentários em C consistem em qualquer seqüência de caracteres começada por /* e terminada por */, com a exigência adicional de que nenhum prefixo próprio termine em */.

Autômatos Finitos Não-Determinísticos

Um *autômato finito não-determinístico* (AFN, simplificadamente) é um modelo matemático que consiste em

1. um conjunto de *estados S*
2. um conjunto de símbolos de entrada Σ (o *alfabeto de símbolos de entrada*)
3. uma função de transição, *movimento*, que mapeia pares estado-símbolo em conjuntos de estados
4. um estado s_0 que é distinguido como o *estado de partida* (ou *inicial*).
5. um conjunto de estados *F* distinguidos como *estados de aceitação* (ou *finais*).

Um AFN pode ser representado diagramaticamente por um grafo dirigido e rotulado, chamado de *grafo de transições*, no qual os nós são os estados e os lados rotulados representam a função de transição. Esse grafo se parece com um diagrama de transições, mas o mesmo caractere pode rotular duas ou mais transições para fora de um mesmo estado e os lados podem ser rotulados pelo símbolo especial ϵ bem como pelos símbolos de entrada.

Um grafo de transições para um AFN que reconheça a linguagem (*a* | *b*)**abb* é mostrado na Fig. 3.19. O conjunto de estados do AFN é {0, 1, 2, 3} e o alfabeto de símbolos de entrada é {*a*, *b*}. O estado 0 na Fig. 3.19 é distinguido como o estado de partida e o estado de aceitação 3 é indicado por um círculo duplo.

Ao descrever um AFN, usamos a representação de grafos de transições. Como veremos, a função de transição de um AFN pode ser implementada de várias formas diferentes num computador. A implementação mais fácil é a *tabela de transições*, na qual existe uma linha para cada estado e uma coluna para cada símbolo de entrada e para ε se necessário. A entrada para a linha *i* e símbolo *a* na tabela é o conjunto de estados (ou mais provavelmente na prática, um apontador para um conjunto de estados) que podem ser atingidos através de uma transição a partir do estado *i* e entrada *a*. A tabela de transições para o AFN da Fig. 3.19 é mostrada na Fig. 3.20.

A representação sob a forma de tabela de transições possui a vantagem de providenciar acesso rápido às transições de um dado estado e caractere; sua desvantagem é que pode ocupar um grande espaço quando o alfabeto de entrada for grande e a maioria das transições for para o conjunto vazio. Representações da função de transição sob a forma de listas de adjacências de transição providenciam implementações mais compactas, mas o acesso a uma dada transição é mais lento. Deveria ficar claro que podemos facilmente converter qualquer uma dessas implementações de um autômato finito para outra.

Um AFN *aceita* uma cadeia de entrada *x* se e somente se existir algum percurso no grafo de transições, a partir do estado inicial até algum estado de aceitação, tal que os rótulos dos lados ao longo do

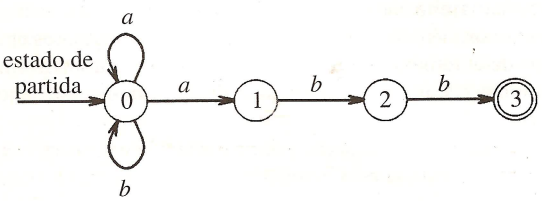


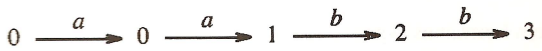
Fig. 3.19. Um autômato finito não-determinístico.

Estado	Símbolo de Entrada	
	a	*b*
0	{0,1}	{0}
1	—	{2}
2	—	{3}

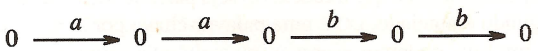
Fig. 3.20. Tabela de transições para o autômato finito da Fig. 3.19.

percurso soletrem a cadeia *x*. O AFN da Fig. 3.19 aceita as cadeias de entrada *abb, aabb, babb, aaabb, ...* . Por exemplo, *aabb* é aceita pelo percurso a partir de 0, seguindo o lado rotulado *a* até o estado 0 de novo e, então, para os estados 1, 2 e 3 através dos lados rotulados *a, b* e *b*, respectivamente.

Um percurso pode ser representado por uma seqüência de transições de estado chamadas *movimentos*. O seguinte diagrama mostra os movimentos realizados ao se aceitar a cadeia de entrada *aabb*:



Em geral, mais de uma seqüência de movimentos pode levar a um estado de aceitação. Note-se que várias outras seqüências de movimentos podem ser realizadas para a cadeia de entrada *aabb*, mas a nenhuma dessas outras ocorre terminar num estado de aceitação. Por exemplo, uma outra seqüência de movimentos para a entrada *aabb* se mantém reentrando no estado não final 0:



A *linguagem definida por* um AFN é o conjunto de cadeias de entrada que o mesmo aceita. Não é difícil mostrar que o AFN da Fig. 3.19 aceita (*a* | *b*)**abb*.

Exemplo 3.13. Na Fig. 3.21, vemos um AFN para reconhecer *aa** | *bb**. A cadeia *aa* é aceita através da movimentação dos estados 0, 1, 2, 2 e 2. Os rótulos desses lados são ϵ, *a*, *a* e *a*, cuja concatenação é *aaaa*. Note-se que os ϵ's "desaparecem" na concatenação. □

Autômatos Finitos Determinísticos

Um *autômato finito determinístico* (AFD, simplificadamente) é um caso especial de autômato finito não-determinístico, no qual

1. nenhum estado possui uma transição-ϵ, isto é, uma transição à entrada ϵ, e
2. para cada estado *s* e símbolo de entrada *a* existe no máximo *um* lado rotulado *a* deixando *s*.

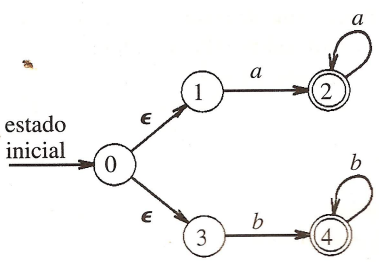


Fig. 3.21. AFN que aceita *aa** | *bb**.

Um autômato finito possui no máximo uma transição, a partir de cada estado, para qualquer símbolo de entrada. Se estivermos usando uma tabela de transições para representar a função de transição de um AFD, então cada entrada na tabela de transições será um único estado. Como conseqüência, é muito fácil determinar se um autômato finito determinístico aceita uma cadeia de entrada, dado que existe no máximo um único percurso, rotulado por aquela cadeia, a partir do estado inicial. O algoritmo seguinte mostra como simular o comportamento de um AFD, dada uma cadeia de entrada.

Algoritmo 3.1. Simulando um AFD.

Entrada. Uma cadeia de entrada x terminada por um caractere de fim de arquivo **eof.** Um AFD D com estado de partida s_0 e conjunto de estados de aceitação F.

Saída. A resposta "sim" se D aceitar x, "não" em caso contrário.

Método. Aplicar o algoritmo da Fig. 3.22 para a cadeia de entrada x. A função *movimento* (s,c) fornece o estado para o qual existe uma transição a partir do estado s e caractere de entrada c. A função *próximo* retorna o próximo caractere da cadeia de entrada x. □

Exemplo 3.14. Na Fig. 3.23, vemos um grafo de transições de um autômato finito determinístico que aceita a mesma linguagem $(a \mid b)^*abb$, aceita pelo AFN da Fig. 3.19. Com este AFD e a cadeia de entrada *ababb*, o Algoritmo 3.1 segue a seqüência de estados 0, 1, 2, 1, 2, 3 e retorna "sim". □

Conversão de um AFN num AFD

Note-se que o AFN da Fig. 3.19 possui duas transições do estado 0 e entrada a; ou seja, pode-se ir para o estado 0 ou 1. Similarmente, o AFN da Fig. 3.21 possui duas transições em ϵ a partir do estado 0. Conquanto não tenhamos mostrado um exemplo, uma situação onde pudéssemos escolher entre uma transição em ϵ ou num símbolo real de entrada também causaria ambigüidade. Essas situações, nas quais a função de transição é multiavaliada, tornam difícil simular um AFN com um programa de computador. A definição de aceitação meramente estabelece que precisa existir algum percurso rotulado pela cadeia de entrada em questão, indo do estado inicial até um estado de aceitação. Mas, se existirem vários percursos que soletrem a mesma cadeia de entrada, podemos ter que considerá-los todos antes de encontrarmos um que leve à aceitação ou descobrir que nenhum deles o faz.

Apresentamos agora um algoritmo para construir um AFD a partir de um AFN que reconheça a mesma linguagem. Esse algoritmo, freqüentemente chamado de *construção de subconjuntos*, é útil na simulação de um AFN por um programa de computador. Um algoritmo estreitamente relacionado desempenha um papel fundamental na construção de *parsers* LR, no próximo capítulo.

Na tabela de transições de um AFN, cada entrada é um conjunto de estados; na tabela de transições de um AFD, cada entrada é exatamente um único estado. A idéia geral por trás da construção AFN-para-AFD é que cada estado do AFD corresponde a um conjunto de estados do AFN. O AFD usa seu estado para controlar todos os possíveis estados que o AFN poderia estar após ler cada símbolo de entrada. Isto significa dizer, após ler a entrada $a_1a_2 \ldots a_n$, o AFD estará num estado que representa o subconjunto T dos estados do AFN que são atingíveis a partir do estado inicial do AFN juntamente com algum percurso rotulado $a_1a_2 \ldots a_n$. O número de estados do AFD pode ser uma exponencial do número de estados do AFN, mas na prática esse caso extremo ocorre raramente.

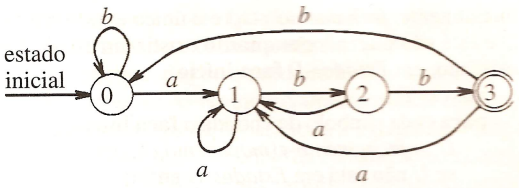


Fig. 3.23. AFD que aceita $(a \mid b)^*abb$.

```
s := s0;
c := próximo_caractere;
enquanto c ≠ eof faça
        s := movimento (s, c);
        c := próximo_caractere
fim;
se s estiver em F então
        retornar "sim"
senão retornar "não";
```

Fig. 3.22. Simulando um AFD.

Algoritmo 3.2. (*Construção de subconjuntos*). Construindo um AFD a partir de um AFN.

Entrada. Um AFN N.

Saída. Um AFD D aceitando a mesma linguagem.

Método. Nosso algoritmo constrói uma tabela de transições *Dtran* para D. Cada estado do AFD é um conjunto de estados do AFN e construímos *Dtran* de tal forma que D simule em "paralelo" todos os possíveis movimentos que N possa fazer a uma dada cadeia de entrada.

Usamos as operações da Fig. 3.24 para controlar os conjuntos de estados do AFN (s representa um estado do AFN e T, o conjunto de estados do AFN).

Antes que o mesmo enxergue o primeiro símbolo de entrada, N pode estar em qualquer um dos estados no conjunto *fechamento*-ϵ (s_0), onde s_0 é o estado inicial de N. Suponhamos que exatamente os estados do conjunto T sejam atingíveis a partir de s_0 a uma dada seqüência de símbolos de entrada e seja a o próximo símbolo de entrada. Ao enxergar a, N pode ir para qualquer um dos estados do conjunto *movimento* (T, a). Quando permitimos transições-ϵ, N pode estar em qualquer um dos estados em *fechamento*-ϵ *(movimento* (T, a)*)*, após ver a.

Construímos *Estados-D*, o conjunto de estados de D, e *Dtran*, a tabela de transições para D, da seguinte maneira. Cada estado de D corresponde a um conjunto de estados do AFN que poderiam ser atin-

OPERAÇÃO	DESCRIÇÃO
fechamento-$\epsilon(s)$	Conjunto de estados do AFN atingíveis a partir de um estado s (do AFN) somente através de transições-ϵ.
fechamento-$\epsilon(T)$	Conjunto de estados do AFN atingíveis a partir de algum estado s do AFN, pertencente a T, somente em transições-ϵ.
movimento (T, a)	Conjunto de estados do AFN para o qual existe uma transição no símbolo de entrada a, a partir de algum estado s do AFN, pertencente a T.

Fig. 3.24. Operações sobre os estados do AFN.

```
inicialmente, fechamento-ϵ(s0) é o único estado em Estados-D e está não marcado; enquanto existir um estado T não
marcado em Estados-D faça início
    marcar T;
    para cada símbolo de entrada a faça início
        U := fechamento-ϵ(movimento (T, a));
        se U não está em Estados-D então
            adicione U como um estado não marcado a
Estados-D;
        Dtran[T, a] := U
    fim
fim
```

Fig. 3.25. A construção de subconjuntos.

```
empilhar todos os estados em T na pilha;
inicializar fechamento-ϵ (T) com T;
enquanto pilha não estiver vazia executar início
        desempilhar t, o elemento de topo, para fora da pilha;
        para cada estado u com um lado de t para u rotulado ϵ
        executar se u não estiver em fechamento-ϵ(T) executar
            início adicionar u ao fechamento-ϵ(T);
            empilhar u na pilha
        fim
fim
```

Fig. 3.26. Cômputo de *fechamento*-ϵ.

gidos em N após ler alguma seqüência de símbolos de entrada, incluindo todas as possíveis transições-ϵ antes ou depois dos símbolos terem sido lidos. O estado de partida de D é *fechamento*-ϵ (s_0). Os estados e transições são adicionados a D usando-se o algoritmo da Fig. 3.25. Um estado de D é de aceitação se for um conjunto de estados do AFN contendo pelo menos um estado de aceitação de N.

O cômputo do *fechamento*-ϵ (T) é um processo típico de busca dos nós atingíveis a partir de um dado conjunto de nós num grafo. Nesse caso, os estados de T são os conjuntos de nós dados e o grafo consiste exatamente nos lados rotulados ϵ no AFN. Um algoritmo simples para computar o *fechamento*-ϵ (T) usa uma pilha para guardar os estados que ainda não foram checados pela existência de lados representando transições ϵ. Tal procedimento é mostrado na Fig. 3.26. □

Exemplo 3.15. A Fig. 3.27 mostra outro AFN N que aceita a linguagem $(a \mid b)^*abb$. (Acontece que é o mesmo AFN da próxima seção, que será construído mecanicamente a partir da expressão regular.) Vamos aplicar o Algoritmo 3.2 a N. O estado inicial do AFD equivalente é *fechamento*-ϵ (0), que é A = {0, 1, 2, 4, 7}, já que esses são exatamente os estados atingíveis a partir do estado 0 através de um percurso no qual cada lado é rotulado ϵ. Note-se que um percurso pode não ter lados, de tal forma que 0 é atingido a partir de si mesmo dessa forma.

O alfabeto de símbolos de entrada aqui é $\{a, b\}$. O algoritmo da Fig. 3.25 nos diz para marcar A e, em seguida, computar

fechamento-ϵ *(movimento (A, a)).*

Primeiro computamos *movimento (A, a),* o conjunto de estados de N tendo transições em a a partir de membros da A. Dentre os estados 0, 1, 2, 4 e 7, somente 2 e 7 têm tais transições, para 3 e 8, e, então,

fechamento-ϵ *(movimento* ({0, 1, 2, 4, 7}, *a*)) = *fechamento*-ϵ ({3,8}) = {1, 2, 3, 4, 6, 7, 8}

Vamos chamar esse conjunto de B. Então, *Dtran* $[A, a] = B$.

Dos estados em A, somente 4 possuem uma transição em b para 5, e, conseqüentemente, o AFD possui uma transição em b de A para

$$C = \textit{fechamento}\text{-}\epsilon\ (\{5\}) = \{1,2,4,5,6,7\}$$

Dessa forma, *Dtran* $[A, b] = C$.

Se continuarmos esse processo com os conjuntos ainda não marcados B e C, atingiremos eventualmente o ponto onde todos os conjuntos que sejam estados do AFD estejam marcados. Isto é certo, dado que existem "somente" 2^{11} subconjuntos diferentes num conjunto de onze estados e um conjunto, uma vez marcado, está marcado para sempre. Os cinco conjuntos de estados diferentes que efetivamente construímos são:

$A = \{0, 1, 2, 4, 7\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$
$C = \{1, 2, 4, 5, 6, 7\}$
$D = \{1, 2, 4, 5, 6, 7, 9\}$
$E = \{1, 2, 4, 5, 6, 7, 10\}$

O estado A é o estado de partida e o estado E é o único estado de aceitação. A tabela de transições completa *Dtran* é mostrada na Fig. 3.28.

Um grafo de transições para o AFD resultante é mostrado na Fig. 3.29. Deveria ser notado que o AFD da Fig. 3.23 também aceita $(a \mid b)^*abb$ e possui um estado a menos. Discutiremos a questão da minimização do número de estados do AFD na Seção 3.9. □

3.7 DE UMA EXPRESSÃO REGULAR PARA UM AFN

Existem muitas estratégias para se construir um reconhecedor a partir de uma expressão regular, cada uma com suas próprias fraquezas e vantagens. Uma estratégia que tem sido usada em alguns programas de edição de texto é a de construir um AFN a partir de uma expressão regular e então simular o comportamento do AFN numa cadeia de entrada usando os algoritmos 3.3 e 3.4 desta seção. Se o tempo de execução for essencial, podemos converter o AFN num AFD usando a construção de subconjuntos da seção anterior. Na Seção 3.9, examinamos uma alter-

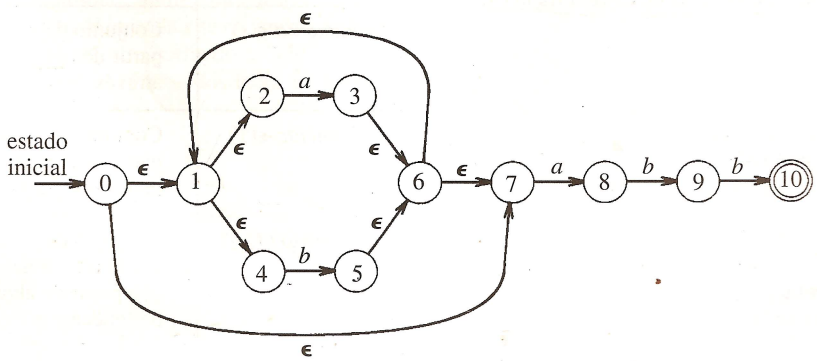


Fig. 3.27. AFN N para $(a \mid b)^*abb$.

ESTADO	SÍMBOLO DE ENTRADA	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*E*
E	*B*	*C*

Fig. 3.28. Tabela de transições *Dtran* para o AFD.

nativa de implementação de um AFD a partir de uma expressão regular, na qual um AFN intermediário não é explicitamente construído. Esta seção conclui com uma discussão sobre as barganhas de tempo-espaço na implementação de reconhecedores baseados em AFNs e AFDs.

Construção de um AFN a partir de uma Expressão Regular

Daremos agora um algoritmo para construir um AFN a partir de uma expressão regular. Existem muitas variantes desse algoritmo, mas aqui apresentamos uma versão simples, que é fácil de implementar. O algoritmo é dirigido pela sintaxe na medida em que usa a estrutura sintática da expressão regular para guiar o processo de construção. As alternativas no algoritmo seguem as alternativas na definição de uma expressão regular. Primeiro mostramos como construir autômatos que reconheçam ϵ e qualquer símbolo no alfabeto. Em seguida, mostramos como construir autômatos para expressões contendo um operador de alternação, concatenação e de fechamento Kleene. Por exemplo, para a expressão $r \mid s$, construímos um AFN indutivamente a partir do AFN para r e s.

À medida que a construção prossegue, cada passo introduz pelo menos dois novos estados, e, então, o AFN resultante, construído a partir de uma expressão regular, possui como número de estados, no máximo o dobro do número de símbolos e de operadores existentes na expressão regular.

Algoritmo 3.3. (*Construção de Thompson*). Um AFN a partir de uma expressão regular.

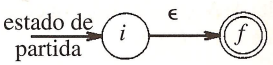
Entrada. Uma expressão regular r sobre um alfabeto Σ.

Saída. Um AFN N que aceita $L(r)$.

Método. Primeiro, dividimos r gramaticalmente em suas expressões constituintes. Em seguida, usando as regras (1) e (2) abaixo, construímos AFNs para cada um dos símbolos básicos em r (aqueles que sejam ϵ ou um símbolo de alfabeto). Os símbolos básicos correspondem às partes (1) e (2) na definição de uma expressão regular. É importante compreender que se um símbolo a ocorrer várias vezes em r, um AFN separado é construído para cada ocorrência.

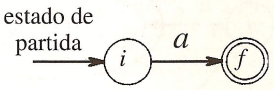
Em seguida, guiados pela estrutura sintática da expressão regular r, combinamos esses AFNs indutivamente usando a regra (3) abaixo, até que obtenhamos o AFN para toda a expressão. Cada AFN intermediário produzido durante o curso da construção corresponde a uma subexpressão de r e possui várias propriedades importantes; possui exatamente um estado final, nenhum lado entra no estado de partida e nenhum lado deixa o estado final.

1. Para ϵ, construímos o AFN



Aqui, i é um novo estado de partida e f um novo estado de aceitação. Este AFN claramente reconhece $\{\epsilon\}$.

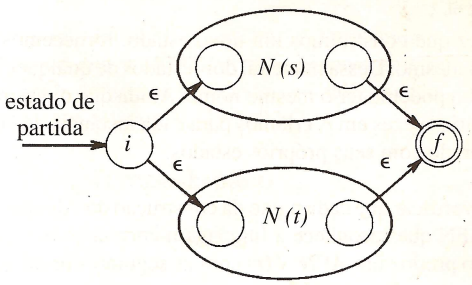
2. Para a em Σ, construímos o AFN



De novo, i é um novo estado e f um novo estado de aceitação. Essa máquina reconhece $\{a\}$.

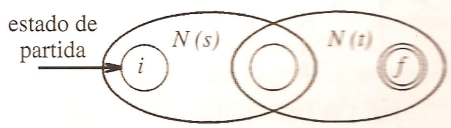
3. Suponhamos que $N(s)$ e $N(t)$ sejam AFNs para as expressões regulares s e t.

a) Para a expressão regular $s \mid t$, construímos o seguinte AFN composto $N(s \mid t)$:



Aqui, i é um novo estado de partida e f um novo estado de aceitação. Existe uma transição em ϵ a partir de i para os estados de partida de $N(s)$ e $N(t)$. Existe uma transição em ϵ a partir dos estados de aceitação de $N(s)$ e $N(t)$ para o novo estado de aceitação f. Os estados inicial e de aceitação de $N(s)$ e de $N(t)$ não são os estados inicial e de aceitação de $N(s \mid t)$. Note-se que qualquer percurso de i para f precisa passar exclusivamente através de $N(s)$ ou de $N(t)$. Dessa forma, vemos que o AFN composto reconhece $L(s) \cup L(t)$.

b) Para a expressão regular st, construímos o AFN composto $N(st)$:



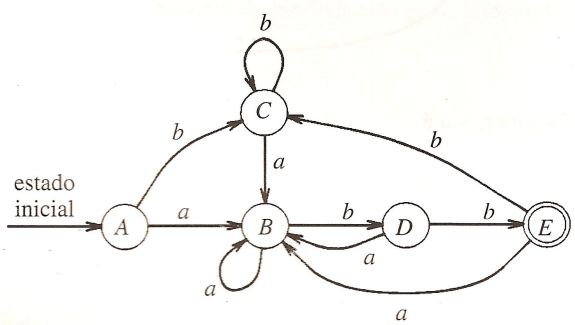
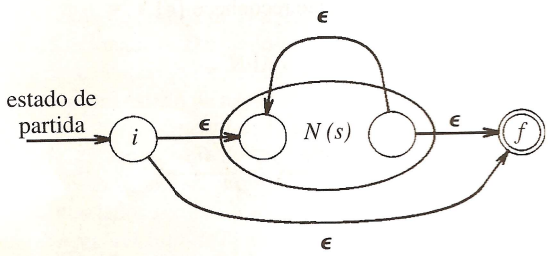


Fig. 3.29. Resultado da aplicação da construção de subconjuntos à Fig. 3.27.

O estado de partida de $N(s)$ e o de aceitação de $N(t)$ se tornam, respectivamente, os estados de partida e de aceitação do AFN composto. O estado de aceitação de $N(s)$ é fundido ao estado de partida de $N(t)$; isto é, todas as transições que emanam do estado de partida de $N(t)$ se tornam transições provenientes do estado de aceitação de $N(s)$. O novo estado resultante da fusão perde tanto o status de estado de aceitação (de $N(s)$) quanto o de partida (de $N(t)$).Um percurso de i para f precisa ir primeiro através de $N(s)$ e, em seguida, através de $N(t)$, de tal forma que o rótulo daquele percurso será uma cadeia em $L(s)\,L(t)$. Como nenhum lado entra no estado de partida de $N(t)$ ou deixa o estado de acei tação de $N(s)$, não pode haver percurso de i para f que trafegue de volta de $N(t)$ para $N(s)$. Por conseguinte, o AFN composto reconhece $L(s)\,L(t)$.

c) Para uma expressão regular s^*, construímos o AFN composto $N(s^*)$:



Aqui, i é o novo estado de partida e f, o novo estado de aceitação. No AFN composto, podemos ir diretamente de i para f ao longo do lado rotulado ϵ, representando o fato de que ϵ está em $(L(s))^*$, ou podemos ir de i para f passando através de $N(s)$ uma ou mais vezes. O AFN composto claramente reconhece $(L(s))^*$.

d) Para uma expressão regular parentetizada (s), usamos o próprio $N(s)$ como o AFN.

A cada vez que construímos um novo estado, fornecemos um nome distinto ao mesmo. Dessa maneira, dois estados de qualquer AFN componente não poderão ter o mesmo nome. Ainda que o mesmo símbolo apareça várias vezes em r, criamos para cada instância do símbolo um AFN separado com seus próprios estados. □

Podemos verificar que cada passo da construção do Algoritmo 3.3 produz um AFN que reconhece a linguagem correta. Adicionalmente, a construção produz um AFN $N(r)$ com as seguintes propriedades:

1. $N(r)$ possui, como número de estados, no máximo o dobro de símbolos e operadores em r. Isto segue do fato de que cada passo da construção cria no máximo dois novos estados.

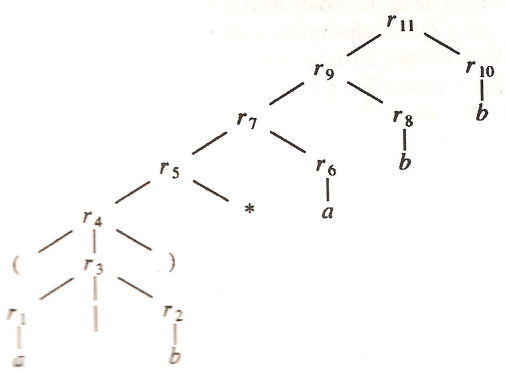
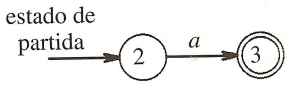


Fig. 3.30. Decomposição de $(a \mid b)^*abb$.

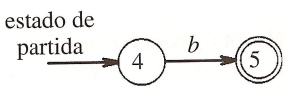
2. $N(r)$ possui exatamente um estado de partida e um de aceitação. O estado de aceitação não possui transições para fora. Esta propriedade vigora igualmente em cada um dos autômatos constituintes.

3. Cada estado de $N(r)$ possui ou uma transição para fora num símbolo de Σ ou no máximo duas transições para fora em ϵ.

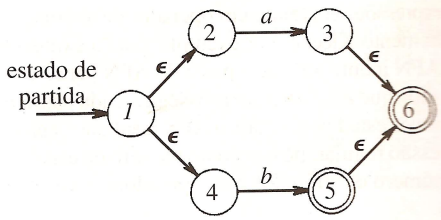
Exemplo 3.16. Vamos usar o Algoritmo 3.3 para construir $N(r)$ para a expressão regular $r = (a \mid b)^*abb$. A Fig. 3.30 mostra uma árvore gramatical para r que é análoga às árvores gramaticais construídas para as expressões aritméticas na Seção 2.2. Para o constituinte r_1, o primeiro a, construímos o AFN



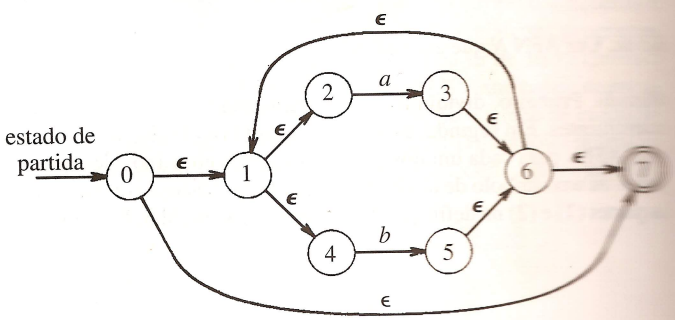
Para r_2, construímos



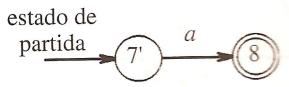
Podemos agora combinar $N(r_1)$ e $N(r_2)$ usando a regra da união a fim de obter o AFN para $r_3 = r_1 \mid r_2$



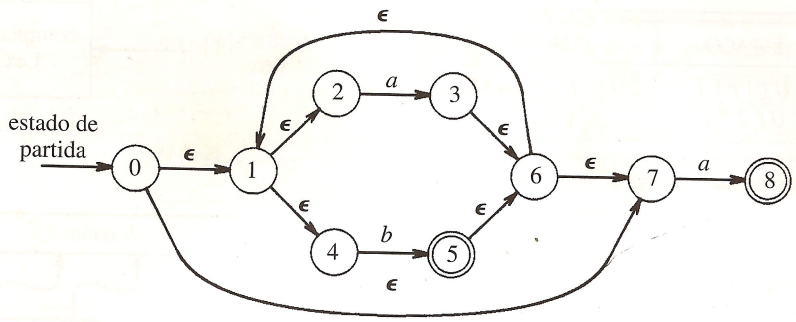
O AFN para (r_3) é o mesmo que o para r_3. O AFN para $(r_3)^*$ é, por conseguinte:



O AFN para $r_6 = a$ é



A fim de se obter o autômato para r_5r_6, combinamos os estados 7 e 7', chamando o estado resultante de 7, para obter



Continuando dessa maneira obtemos o AFN para $r_{11} = (a \mid b)^*abb$, que foi primeiramente exibido na Fig. 3.27. □

Simulação em Duas Pilhas de um AFN

Apresentamos agora um algoritmo que, dado um AFN N construído pelo Algoritmo 3.3, com uma cadeia de entrada x, determina se N aceita x. O algoritmo trabalha através da leitura de um caractere de entrada de cada vez e computa o conjunto completo de estados em que N poderia estar após ter lido cada prefixo da entrada. O algoritmo aproveita a vantagem das propriedades especiais do AFN produzido pelo Algoritmo 3.3 para computar cada conjunto de estados não determinísticos eficientemente. Pode ser implementado para rodar num tempo proporcional a $|N| \times |x|$ onde $|N|$ é o número de estados em N e $|x|$ é o comprimento de x.

Algoritmo 3.4. Simulação de um AFN.

Entrada. Um AFN construído p[illegible] e uma cadeia de entrada x. Assumimos que x seja terminada pelo caractere de fim de arquivo **eof.** N possui estado de partida s_0, conjunto de estados de aceitação F.

Saída. A resposta "sim" se N aceita x; "não" em caso contrário.

Método. Aplicar o algoritmo delineado na Fig. 3.31 para a cadeia de entrada x. O algoritmo, com efeito, realiza a construção de subconjuntos em tempo de execução. Computa, em dois estágios, uma transição a partir do conjunto de estados correntes S para o próximo conjunto. Primeiro, determina *movimento* (S, a), isto é, todos os estados que podem ser alcançados a partir de qualquer estado em S através de uma transição em a, o caractere corrente de entrada. Em seguida, computa o *fechamento*-ϵ de *movimento* (S, a), ou seja, todos os estados que podem ser atingidos a partir de *movimento* (S, a) através de zero ou mais transições ϵ. O algoritmo usa a função *próximo_caractere* para ler os caracteres de x, um de cada vez. Quando todos os caracteres de x tiverem sido examinados, o algoritmo retorna "sim" se um estado de aceitação estiver no conjunto S de estados correntes; "não" em caso contrário. □

O Algoritmo 3.4 pode ser eficientemente implementado usando-se duas pilhas e um vetor de *bits* indexado pelos estados do AFN.

```
S := fechamento ϵ ({s0} );
a := próximo_caractere;
enquanto a ≠ eof faça início
        S := fechamento ϵ (movimento (S, a) );
        a := próximo_caractere;
fim
se S ∩ F ≠ ∅ então
        retornar "sim";
senão retornar "não";
```

Fig. 3.31. Simulação de um AFN do Algoritmo 3.3.

Usamos uma pilha para controlar o conjunto corrente de estados não-determinísticos e a outra pilha para computar o próximo conjunto de estados não-determinísticos. Podemos usar o algoritmo da Fig. 3.26 para computar o *fechamento*-ϵ. O vetor de *bits* pode ser usado para determinar em tempo constante se um estado não-determinístico já está na pilha, de forma a não o adicionarmos duas vezes a mesma. Uma vez que tenhamos computado o próximo estado numa segunda pilha, podemos intercambiar os papéis das duas. Como cada estado não-determinístico possui no máximo duas transições para fora, cada estado pode dar origem a, no máximo, dois novos estados numa transição. Vamos denominar de $|N|$ o número de estados de N. Como podem existir no máximo $|N|$ estados numa pilha, o cômputo do próximo conjunto de estados a partir do conjunto de estados corrente pode ser feito num tempo proporcional a $|N|$. Conseqüentemente, o tempo total necessário para simular o comportamento de N para a entrada x é proporcional a $|N| \times |x|$.

Exemplo 3.17. Seja N o AFN da Fig. 3.27 e seja x a cadeia constituída [illegible]. O estado de partida é o *fechamento*-ϵ $(\{0\}) = \{0, 1, 2, 4, 7\}$. Ao símbolo de entrada a existe uma transição de 2 para 3 e [illegible] seguinte, T é $\{3, 8\}$. Tomando o *fechamento* -ϵ de [illegible] estados $\{1, 2, 3, 4, 6, 7, 8\}$. Como nenhum desses estados não-determinísticos é de aceitação, o algoritmo retorna "não". [illegible] algoritmo 3.4 realiza a construção de subconjuntos em tempo de execução. Por exemplo, comparemos as transições acima com os estados do AFD da Fig. 3.29 construído a partir do AFN da Fig. 3.27. Os conjuntos de estados inicial e próximo à entrada a correspondem aos estados A e B do AFD. □

Barganhas de Tempo-Espaço

Dada uma expressão regular r e uma cadeia de entrada x, temos agora dois métodos para determinar se x está em $L(r)$. Um enfoque é o de usar o Algoritmo 3.3 para construir um AFN N a partir de r. Esta construção pode ser feita num tempo $O(|r|)$ (proporcional a $|r|$), onde $|r|$ é o comprimento de r. N possui um número de estados que é no máximo o dobro de $|r|$ e também, duas transições, no máximo, a partir de cada estado, de tal forma que a tabela de transições para N pode ser armazenada num espaço $O(|r|)$. Podemos, então, usar o Algoritmo 3.4 para determinar se N aceita x num tempo $O(|r| \times |x|)$. Dessa forma, usando esta abordagem, podemos determinar se x está em $L(r)$ num tempo total proporcional ao comprimento de r vezes o comprimento de x. Esse enfoque tem sido usado em alguns editores de texto a fim de procurar por padrões de expressões regulares quando a cadeia-alvo x não é geralmente muito longa.

Uma segunda abordagem é a de construir um AFD a partir de uma expressão regular r através da aplicação da construção de Thompson para r e, em seguida, usar a construção de subconjuntos, Algoritmo 3.2, para o AFN resultante (uma implementação que evita a construção explícita de um AFN é dada na Seção 3.9). Implementando a função de transição com uma tabela de transições, podemos usar o Algoritmo 3.1 para simular o AFD para a entrada x num tempo proporcional ao comprimento de x, independentemente do número de estados do AFD. Esse enfoque tem sido freqüentemente usado em programas de

AUTÔMATO	ESPAÇO	TEMPO
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA	$O(2^{\|r\|})$	$O(\|x\|)$

Fig. 3.32. Espaço e tempo requeridos para reconhecer expressões regulares.

reconhecimento de padrões que percorrem arquivos de texto procurando padrões de expressões regulares. Uma vez que o autômato finito tenha sido construído, a procura pode proceder muito rapidamente, e, então, esse enfoque é vantajoso quando a cadeia-alvo x é muito longa.

Existem, entretanto, certas expressões regulares cujo menor AFD possui um número de estados cujo número é uma exponencial do tamanho da expressão regular. Por exemplo, a expressão regular $(a \mid b)^*a(a \mid b)(a \mid b)\ldots(a \mid b)$, onde existem $n-1$ $(a \mid b)$'s ao fim, não possui um AFD com menos de 2^n estados. Essa expressão regular denota qualquer cadeia de a's e b's na qual o enésimo caractere a partir da extremidade à direita é um a. Não é difícil provar que qualquer AFD para esta expressão precisa controlar os últimos n caracteres que examina na entrada; de outra forma, poderia fornecer uma resposta errada. Claramente, são necessários pelo menos 2^n estados para controlar todas as possíveis seqüências de n a's e b's. Felizmente, expressões como essa não ocorrem muito freqüentemente em aplicações de análise léxica, mas existem aplicações onde expressões similares efetivamente emergem.

Um terceiro enfoque é o de usar um AFD, mas evitar construir toda a tabela de transições utilizando uma técnica chamada "avaliação preguiçosa de transições". Aqui, as transições são computadas em tempo de execução, mas uma transição a partir de um dado estado e caractere não é determinada até que seja efetivamente necessitada. As transições computadas são armazenadas num *cache*. A cada vez que uma transição estiver prestes a ser realizada, o *cache* é consultado. Se a transição não estiver lá, é então computada e armazenada no *cache*. Se o *cache* encher, podemos apagar algumas transições previamente computadas a fim de abrir espaço para a nova transição.

A Fig. 3.32 sumariza o pior caso de exigência de espaço e tempo para se determinar se uma cadeia de entrada x está na linguagem denotada por uma expressão regular r, usando reconhecedores construídos a partir de autômatos determinísticos e não-determinísticos. A técnica "preguiçosa" combina as exigências de espaço do método do AFN com as de tempo da abordagem do AFD. Sua exigência de espaço é o tamanho da expressão regular mais o tamanho do *cache;* seu tempo de execução observado é quase tão curto quanto aquele de um reconhecedor determinístico. Em algumas aplicações, a técnica "preguiçosa" é consideravelmente mais rápida do que o enfoque do AFD, porque nenhum tempo é desperdiçado computando-se transições de estados que jamais serão usadas.

TOP SECRET

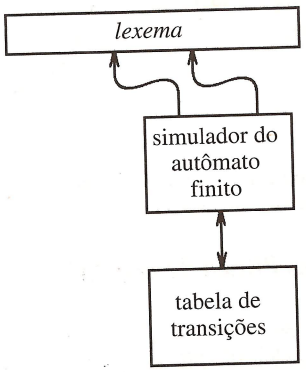
3.8 O PROJETO DE UM GERADOR DE ANALISADORES LÉXICOS

Nesta seção, consideramos o projeto de uma ferramenta de *software* que constrói automaticamente um analisador léxico a partir de um programa na linguagem Lex. Apesar de discutirmos vários métodos, e de nenhum ser precisamente idêntico àquele usado pelo comando Lex do sistema UNIX, referimo-nos a esses programas para construir analisadores léxicos como compiladores Lex.

Assumimos que temos uma especificação de um analisador léxico da forma

$$\begin{array}{ll} p_1 & \{\text{ação}_1\} \\ p_2 & \{\text{ação}_2\} \\ \cdots & \cdots \\ p_n & \{\text{ação}_n\} \end{array}$$

(a) Compilador Lex.



(b) Analisador léxico esquematizado.

Fig. 3.33. Modelo de um compilador Lex.

onde, como na Seção 3.5, cada padrão p_i é uma expressão regular e cada *ação*$_i$ é um fragmento de programa que deve ser executado sempre que um lexema reconhecido por p_i for encontrado na entrada.

Nosso problema é o de construir um reconhecedor que procure por lexemas no *buffer* de entrada. Se mais de um padrão for reconhecido, o reconhecedor deverá escolher o mais longo lexema encontrado. Se existirem dois ou mais padrões que reconheçam o lexema mais longo, o primeiro padrão de reconhecimento listado será escolhido.

Um autômato finito é um modelo natural em torno do qual se pode construir um analisador léxico e aquele construído por nosso compilador Lex possui a forma mostrada na Fig. 3.33 (b). Lá está o *buffer* de entrada com dois apontadores para o mesmo, o início_de_lexema e o apontador_adiante, como discutido na Seção 3.2. O compilador Lex constrói uma tabela de transições para um autômato finito a partir de padrões de expressões regulares na especificação Lex. O analisador léxico por si só consiste em um simulador de autômato finito que usa esta tabela de transições para procurar pelos padrões de expressões regulares no *buffer* de entrada.

O resto desta seção mostra que a implementação de um compilador *Lex* pode ser baseada quer num autômato determinístico, quer num não-determinístico. Ao fim da última seção, vimos que a tabela de transições de um AFN para um padrão de expressão regular podia ser consideravelmente menor do que a de um AFD, mas que o AFD possuía a decidida vantagem de ser capaz de reconhecer padrões mais rapidamente do que um AFN.

Reconhecimento de Padrões Baseado em AFN's

Um método é o de construir a tabela de transições de um autômato finito não-determinístico N para o padrão composto $p_1 \mid p_2 \mid \ldots \mid p_n$. Isto pode ser feito primeiro criando-se um AFN $N(p_i)$ para cada padrão p_i, usando o Algoritmo 3.3 e, em seguida, adicionando-se um novo estado de partida s_0 e, finalmente, ligando-se s_0 ao estado de partida de cada $N(p_i)$ através de uma transição ϵ, como mostrado na Fig. 3.34.

Para simular este AFN, podemos usar uma modificação do Algoritmo da Fig. 3.4. A modificação assegura que o AFN combinado reconheça o mais longo prefixo da entrada que seja reconhecido por um padrão. No AFN combinado, existe um estado de aceitação para cada padrão p_i. Ao simularmos o AFN usando o Algoritmo 3.4, cons-

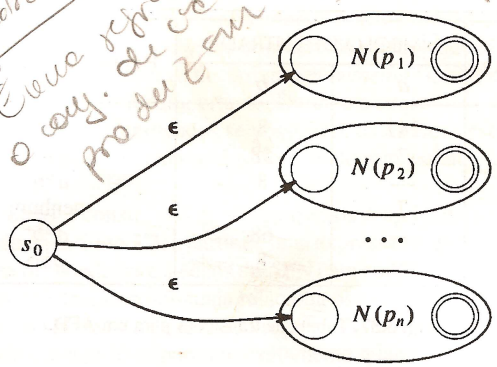


Fig. 3.34. AFN construído a partir de uma especificação Lex.

truímos a seqüência de conjuntos de estados em que o AFN combinado possa estar após examinar cada caractere de entrada. Mesmo se encontrarmos um conjunto de estados que contenha um estado de aceitação, para obtermos a correspondência mais longa precisaremos continuar a simular o AFN até que o mesmo atinja a *terminação,* isto é, o conjunto de estados a partir do qual não existam transições no símbolo corrente de entrada.

Presumimos que a especificação Lex seja projetada de tal forma que um programa válido de entrada não possa preencher inteiramente o *buffer* de entrada sem que o AFN tenha atingido a terminação. Por exemplo, cada compilador coloca alguma restrição no comprimento de um identificador e as violações desse limite serão detectadas quando o *buffer* de entrada estourar a capacidade de armazenamento, se não mais cedo.

Para atingir o reconhecimento correto, fazemos duas modificações ao Algoritmo 3.4. Primeiro, sempre que adicionarmos um estado de aceitação ao conjunto corrente de estados, registramos a posição corrente de entrada e o padrão p_i correspondente a esse estado de aceitação. Se o conjunto corrente entrada já contiver um estado de aceitação, então somente o padrão que aparece primeiro na especificação de Lex é registrado. Segundo, continuamos fazendo transições até encontrarmos a terminação. Ao final, retraímos o apontador adiante para a posição na qual o último reconhecimento ocorreu. O padrão que realiza esse reconhecimento identifica o *token* encontrado e o lexema reconhecido é a cadeia entre os apontadores de início de lexema e adiante.

Usualmente, a especificação Lex é tal que algum padrão, possivelmente um padrão de erro, irá sempre reconhecer. Se nenhum padrão o fizer, entretanto, temos uma condição de erro para a qual nenhuma provisão foi feita e o analisador léxico deveria transferir o controle para alguma rotina *default* de recuperação de erros.

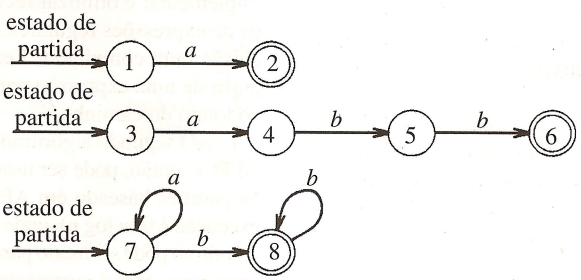
Exemplo 3.18. Um único exemplo ilustra as idéias acima. Suponhamos ter o seguinte programa Lex, consistindo em três expressões regulares e nenhuma definição regular.

a	{ } /* as ações são omitidas aqui */
abb	{ }
a^*b^+	{ }

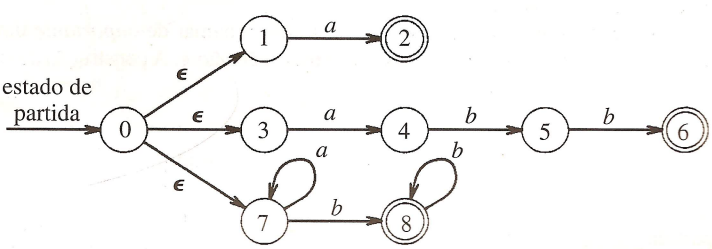
Os três *tokens* acima são reconhecidos pelos autômatos da Fig. 3.35(a). Simplificamos o terceiro autômato em algumas partes em relação àquele que teria sido produzido pelo Algoritmo 3.3. Como indicado acima, podemos converter os AFNs da Fig. 3.35(a) no único AFN combinado *N* mostrado à Fig. 3.35(b).

Vamos considerar o comportamento de *N* na cadeia de entrada *aaba* usando nossa modificação do Algoritmo 3.4. A Fig. 3.36 mostra os conjuntos de estados e padrões que reconhecem à medida que cada caractere da entrada *aaba* é processado. Essa figura mostra que o conjunto inicial de estados é {0, 1, 3, 7}. Os estados 1, 3 e 7 têm cada um uma transição em *a* para os estados 2, 4 e 7, respectivamente. Como o estado 2 é o de aceitação para o primeiro padrão, registramos o fato de que o primeiro padrão reconhece positivamente após a leitura do primeiro *a*.

Entretanto, existe uma transição a partir do estado 7 para o estado 7 no segundo caractere de entrada e, por conseguinte, precisamos continuar realizando as transições. Existe uma transição a partir do estado 7 para o 8 no caractere de entrada *b*. O estado 8 é o estado de aceitação para o terceiro padrão. Uma vez que atinjamos o estado 8, não existem transições possíveis para o próximo caractere de entrada *a* e, nesse caso, atingimos terminação. Como o último reconhecimento ocorreu após lermos o terceiro caractere de entrada, relatamos que o terceiro padrão reconheceu o lexema *aab*. □



(a) AFN para *a*, *abb* e a^*b^+.



(b) AFN combinado.

Fig. 3.35. AFN reconhecendo três diferentes padrões.

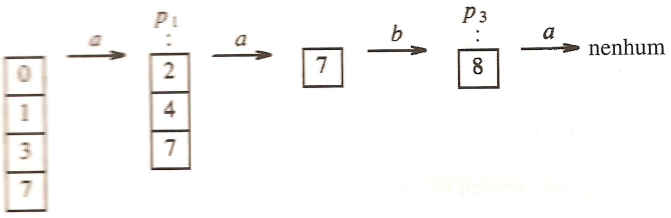


Fig. 3.36. Seqüência de estados atingidos no processamento da entrada *aaba*.

O papel da *ação*$_i$ associada ao padrão p_i na especificação Lex é como se segue. Quando uma instância de p_i é reconhecida, o analisador léxico executa o programa associado *ação*$_i$. Note-se que *ação*$_i$ não é executada somente porque o AFN entra num estado que inclua um estado de aceitação para p_i; a *ação*$_i$ somente é executada se p_i vier a se tornar o padrão que produza o reconhecimento mais longo.

AFDs para Analisadores Léxicos

Um outro enfoque para a construção de um analisador léxico é o de usar um AFD para realizar o reconhecimento de padrões. A única nuança é certificar que encontramos os padrões de reconhecimento adequados. A situação é completamente análoga à simulação modificada do AFN já descrito. Quando convertemos um AFN para um AFD usando a construção de subconjuntos do Algoritmo 3.2, pode haver vários estados de aceitação num dado subconjunto de estados não-determinísticos. Em tal situação, o estado de aceitação correspondente ao padrão listado à frente na especificação Lex possui prioridade. Como na simulação do AFN, a única outra modificação que precisamos fazer é a de continuar realizando transições até que tenhamos atingido um estado sem nenhum próximo estado (isto é, o estado $\varnothing$) para o símbolo corrente de entrada. Para encontrar o lexema reconhecido, retornamos à última posição de entrada a qual o AFD entrou num estado de aceitação.

Exemplo 3.19. Se convertermos o AFN da Fig. 3.35 para um AFD, obtemos a tabela de transições da Fig. 3.37, onde os estados do AFD foram designados por listas de estados do AFN. A última coluna da Fig. 3.37 indica um dos padrões reconhecidos ao se entrar naquele estado do AFD. Por exemplo, dentre os estados 2, 4 e 7 do AFN, somente 2 é um estado de aceitação e é o estado de aceitação do autômato para a expressão regular *a* na Fig. 3.35(a). Conseqüentemente, o estado 247 do AFD reconhece o padrão *a*.

Note-se que a cadeia *abb* é reconhecida por dois padrões, *abb* e a^*b^+, reconhecidos pelos estados 6 e 8 do AFN. O estado 68 do AFD, na última linha da tabela de transições inclui, pois, dois estados de aceitação para o AFN. Notamos que *abb* aparece antes de a^*b^+ nas regras de tradução de nossa especificação Lex e, então, anunciamos que *abb* foi encontrado no estado 68 do AFD.

Com a cadeia de entrada *aaba*, o AFD entra nos estados sugeridos pela simulação do AFN mostrada na Fig. 3.36. Consideremos um segundo exemplo, a cadeia de entrada *aba*. O AFD da Fig. 3.37 começa no estado 0137. A entrada *a* vai para o estado 247. Em seguida, à entrada *b*, progride até o estado 58 e, à entrada *a*, não há próximo estado. Atingimos, por conseguinte, a terminação, progredindo através dos estados 0137, em seguida 247 e então 58. O último desses estados inclui

Estado	Símbolo de Entrada		Padrão Anunciado
	a	*b*	
0137	247	8	nenhum
247	7	58	*a*
8	—	8	a^*b^+
7	7	8	nenhum
58	—	68	a^*b^+
68	—	8	*abb*

Fig. 3.37. Tabela de transições para um AFD.

o estado de aceitação 8 da Fig. 3.35(a). Conseqüentemente, no estado 58, o AFD anuncia que o padrão a^*b+ foi reconhecido e seleciona *ab*, o prefixo da entrada que levou ao estado 58, como o lexema. □

Implementando o Operador *Lookahead*

Relembremos da Seção 3.4 que o operador *lookahead* / é necessário em algumas situações, dado que o padrão que denota um *token* particular pode precisar descrever algum contexto após o lexema efetivo. Quando convertemos um padrão com / para um AFN, podemos tratar a / como se fosse ϵ, e, então, não precisamos efetivamente procurar por / à entrada. Entretanto, se uma cadeia denotada por esta expressão regular for reconhecida no *buffer* de entrada, o final do lexema não é a posição do estado de aceitação do AFN. Ao invés, é a última ocorrência de estado deste AFN que tenha uma transição (imaginária) em /.

Exemplo 3.20. O AFN que reconhece o padrão `IF` dado no Exemplo 3.12 é mostrado na Fig. 3.38. O estado 6 indica a presença da palavra-chave `IF`; entretanto, encontramos o *token* `IF` esquadrinhando de volta até a última ocorrência do estado 2. □

3.9 OTIMIZAÇÃO DE RECONHECEDORES DE PADRÕES BASEADOS EM AFDs

Nesta seção, apresentamos três algoritmos que têm sido usados para implementar e otimizar reconhecedores de padrões construídos a partir de expressões regulares. O primeiro algoritmo é adequado para inclusão num compilador Lex, porque constrói um AFD diretamente a partir de uma expressão regular, sem construir um AFN intermediário ao longo do caminho.

O segundo algoritmo minimiza o número de estados de qualquer AFD, e, então, pode ser usado para reduzir o tamanho do reconhecedor de padrões baseado em AFDs. O algoritmo é eficiente; seu tempo de execução é $O(n \log n)$, onde n é o número de estados do AFD. O terceiro algoritmo pode ser usado para produzir representações rápidas porém mais compactas para a representação da tabela de transições do AFD do que uma representação direta sob a forma de uma tabela bidimensional.

Estados Importantes de um AFN

Vamos chamar de *importante* um estado de um AFN que tenha uma transição não-ϵ. A construção de subconjuntos na Fig. 3.25 usa somente

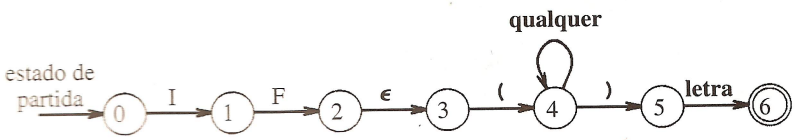


Fig. 3.38. AFN reconhecendo a palavra-chave Fortran `IF`.