REFERÊNCIAS

Algumas das primeiras publicações sobre autômatos finitos foram:

- G. H. Mealy "A method for synthesizing sequential circuits", *Bell System Technical Journal*, 34, 5, pp. 1045-1079, 1955, e
- E. F. Moore "Gedanken experiments on sequential machines", *Automata Studies*, ed. C. E. Shannon and J. McCarthy, pp. 129-53. Princeton: Princeton University Press, 1956.

A obra clássica sobre autômatos finitos (contendo o Teorema 2.2. 1) é:

- M. O. Rabin and D. Scott "Finite automata and their decision problems", *IBM Journal of Research and Development*, 3, pp. 114-25, 1959.

O teorema 2.3.2, afirmando que autômatos finitos aceitam linguagens regulares, é de Kleene:

- S. C. Kleene "Representation of events by nerve nets", in *Automata Studies*, ed. C. E. Shannon and J. McCarthy, pp. 3-42. Princeton: Princeton University Press, 1956.

Nossa prova desse teorema segue a obra:

- R. McNaughton and H. Yamada "Regular expressions and state graphs for automata", *IEEE Transactions on Electronic Computers*, EC-9, 1 pp. 39-47, 1960.

O teorema 2.4.1 (o "lema do bombeamento") é da autoria de:

- Y. Bar-Hillel, M. Perls, and E. Shamir "On formal properties of simple phrase structure grammars", Zeitschrift fur Phonetik, Sprachwissenschaft, und Kommunikationsforschung, 14, pp. 143-172, 1961.

Os transdutores de estados finitos (Problema 2.1.4) foram apresentados pela primeira vez em:

- S. Ginsburg "Examples of abstract machines", IEEE Transactions on Electronic Computers, EC-11, 2, pp. 132-135, 1962.

Os autômatos de estados finitos de duas fitas (Problemas 2.1.5 e 2.4.7) foram examinados em:

- M. Bird "The equivalence problem for deterministic two-tape automata", *Journal of Computer and Systems Sciences*, 7, pp. 218-236, 1973.

O teorema de Myhill-Nerode (Teorema 2.5.2) é da autoria de:

- A. Nerode "Linear automaton transformations", *Proc. AMS*, 9, pp.541-544, 1958.

O algoritmo para minimizar autômatos finitos provém do trabalho de Moore, citado acima. Um algoritmo mais eficiente é encontrado em:

- J. E. Hopcroft "An $n \log n$ algorithm for minimizing the states in a finite automaton", in *The Theory of Machines and Computations*, ed. Z. Kohavi. New York: Academic Press, 1971.

A simulação de autômatos não-determinísticos (Teorema 2.6.3) baseia-se em:

- K. Thompson "Regular Expression search algorithms", *Communications of the ACM*, 11, 6, pp. 419-422, 1968.

O algoritmo rápido de correspondência de padrões, do Problema 2.6.3, encontra-se em:

- D. E. Knuth, J. H. Morris, Jr, V. R. Pratt "Fast pattern matching in strings", *SIAM J. on Computing*, 6, 2, pp. 323-350, 1976.

A equivalência dos autômatos finitos de mão única com os de mão dupla (Problema 2.5 - 4) é mostrada em:

- J. C. Shepherdson "The reduction of two-way automata to one-way automata", *IBM Journal of Research and Development*, 3, pp. 198-200, 1959.

3 Linguagens livres de contexto

3.1 GRAMÁTICAS LIVRES DE CONTEXTO

Imagine-se um processador de linguagem. Você poderia reconhecer uma sentença válida em português, ao ouvir: "o gato está no chapéu", você entenderia que essa frase está no mínimo sintaticamente correta (seja ou não verdade aquilo que ela diz), mas "chapéu o no está gato" não teria sentido. De fato, você utiliza um idioma para fazer isso e pode dizer, imediatamente, ao ler uma sentença, se ela foi ou não formada de acordo com as regras geralmente aceitas para estruturar sentenças. Sob esse enfoque, você está atuando como um **reconhecedor de linguagem**: um dispositivo que reconhece cadeias válidas. Os autômatos finitos do capítulo anterior são formalizados como uma possível forma de um reconhecedor de linguagem.

Mas você também é capaz de produzir sentenças válidas da língua portuguesa. Novamente, a razão por que você iria querer fazer isso e a forma como observa um idioma para poder fazê-lo não são do nosso interesse; mas o fato é que você de fato fala ou escreve sentenças, e, em geral, estas estão sintaticamente corretas (mesmo quando falsas). Sob esse aspecto, você está atuando como um **gerador de linguagem**. Nesta seção, iremos estudar alguns tipos de geradores de linguagens formais. Um dispositivo dessa natureza começa a operar ao receber algum sinal "inicia", e passa então a construir uma cadeia da linguagem. Sua operação não fica completamente determinada *a priori*, mas é, não obstante, condicionada por um conjunto fixo de regras. Ao final, esse processo pára, tendo o dispositivo produzido uma cadeia completa. A linguagem definida pelo dispositivo é o conjunto de todas as possíveis cadeias que ele pode produzir.

Um reconhecedor ou um gerador para uma linguagem natural qualquer, como, por exemplo, o idioma inglês, não é uma coisa fácil de construir; de fato, projetar esses dispositivos para grandes subconjuntos de linguagens naturais tem sido um permanente desafio na pesquisa de ponta há várias décadas. Contudo, a idéia de uma linguagem geradora tem alguma força explicativa em uma tentativa de discutir a linguagem humana. Mais importante para nós, entretanto, é a teoria dos geradores de linguagens "artificiais" formais, incluindo as linguagens regulares e a importante classe das linguagens "livres de contexto" a serem apresentadas adiante. Essa teoria complementará elegantemente o estudo dos autômatos, que reconhece linguagens e é também de valor prático na especificação e na análise das linguagens de computador.

Expressões regulares podem ser vistas como geradores de linguagem. Por exemplo, considere-se a expressão regular $a(a^* \cup b^*)b$. Uma descrição verbal de como gerar uma cadeia, de acordo com essa expressão, seria a seguinte:

Primeiramente, emita um *a*. Então, faça uma das seguintes duas coisas:
Emita um número arbitrário (ou nulo) de *a*'s ou então um número arbitrário (ou nulo) de *b*'s.
Finalmente, emita um *b*.

A linguagem associada a esse gerador de linguagem – isto é, o conjunto de todas as cadeias que podem ser produzidas pelo processo que acabamos de descrever – é, exatamente a linguagem regular definida pela expressão regular $a(a^* \cup b^*)b$.

Neste capítulo, estudaremos um tipo mais complexo de geradores de linguagem, chamados **gramáticas livres de contexto**, as quais materializam um completo entendimento do procedimento de construção das cadeias pertencentes à linguagem. Utilizando novamente o exemplo da linguagem gerada por $a(a^* \cup b^*)b$, note que qualquer cadeia dessa linguagem consiste de um *a* inicial, seguido por uma *parte intermediária* – gerada por $(a^* \cup b^*)$ – seguido por um *b* final. Definindo *S* como um novo símbolo que pode ser interpretado como "qualquer cadeia na linguagem" e *M* como um símbolo que significa "parte intermediária", então podemos expressar essa observação escrevendo

$$S \rightarrow aMb$$

onde $\rightarrow$ é lido como "pode ser". Uma expressão assim construída denomina-se **regra**. O que pode ser *M*, a parte intermediária? A resposta é: ou uma cadeia de *a*'s, ou uma cadeia de *b*'s. Podemos expressar esse fato adicionando as regras

$$M \rightarrow A$$
$$M \rightarrow B$$

onde *A* e *B* são novos símbolos que representam, respectivamente uma cadeia de *a*'s ou *b*'s. Agora, o que é uma cadeia de *a*'s? Ela pode ser a cadeia vazia

$$A \rightarrow \varepsilon$$

ou pode consistir de um *a* inicial seguido por uma outra cadeia complementar de *a*'s:

$$A \rightarrow aA$$

Analogamente, para *B*, temos as duas regras seguintes:

$$B \rightarrow \varepsilon$$
$$B \rightarrow bB$$

A linguagem denotada pela expressão regular $a(a^* \cup b^*)b$ pode, então, ser definida alternativamente pela seguinte linguagem geradora:

Inicie-se com a cadeia elementar formada apenas pelo símbolo *S*. Localize-se um símbolo na cadeia atual, que aparece à esquerda do símbolo $\rightarrow$ em qualquer das regras acima. Substitua-se uma ocorrência desse símbolo pela cadeia que aparece à direita de $\rightarrow$ na mesma regra. Repita-se esse processo, até que tal símbolo não possa mais ser encontrado.

Por exemplo, para gerar a cadeia *aaab*, começamos com *S*, como especificado; então substituímos *S* por *aMb*, de acordo com a primeira regra, $S \rightarrow aMb$.

Para *aMb*, aplicamos a regra $M \rightarrow A$ e obtemos *aAb*. Então, aplicamos a regra $A \rightarrow aA$ duas vezes para obter a cadeia *aaaAb*. Finalmente, aplicamos a regra $A \rightarrow \varepsilon$. Na cadeia resultante, *aaab*, não podemos identificar qualquer símbolo que apareça à esquerda de $\rightarrow$ em qualquer das regras. Portanto, a operação da nossa linguagem geradora terminou e *aaab* foi produzida, como prometido.

Uma **gramática livre de contexto** é uma linguagem geradora que opera nos moldes descritos acima, com algum conjunto de regras. Vamos fazer uma pausa para explicar agora por que tal linguagem geradora é chamada livre de contexto. Considere-se a cadeia *aaAb*, obtida em um estágio intermediário da geração de *aaab*. É natural chamar as cadeia *aa* e *b*, que envolvem o símbolo *A*, como sendo o contexto de *A nessa* particular cadeia. Agora, a regra $A \rightarrow aA$ diz que podemos substituir *A* pela cadeia *aA*, não importa quais sejam as cadeias que o envolvam. Em outras palavras, a substituição pode ser feita *independentemente* do *contexto de A*. No Capítulo 4, examinaremos uma gramática na qual a substituição pode ser condicionada à existência de um contexto apropriado.

Em uma gramática livre de contexto, alguns símbolos aparecem à esquerda do símbolo $\rightarrow$ nas regras – *S*, *M*, *A* e *B*, em nosso exemplo – enquanto outros – *a* e *b* – não. Os símbolos deste último tipo são chamados **terminais**, uma vez que a produção de uma cadeia que consista somente de tais símbolos indica o término do processo de geração. Todas essas idéias são expostas formalmente na próxima definição.

Definição 3.1.1: Uma **gramática livre de contexto** *G* é uma quádrupla (V, Σ, R, S), onde

V é um alfabeto,
Σ (o conjunto de **terminais**) é um subconjunto de V,
R (o conjunto de **regras**) é um subconjunto finito de $(V - \Sigma) \times V^*$, e
S (o **símbolo inicial**) é um elemento de $V - \Sigma$.

Os membros de $V - \Sigma$ são chamados **não-terminais**. Para quaisquer $A \in V - \Sigma$ e $u \in V^*$, escreve-se $A \rightarrow_G u$, sempre que $(A, u) \in R$. Para quaisquer cadeias $u, v \in V^*$, escreve-se $u \Rightarrow_G v$ se e somente se há cadeias $x, y \in V^*$ e $A \in V - \Sigma$, tais que $u = xAy$, $v = xv'y$ e $A \rightarrow_G v'$. A relação $\Rightarrow_G^*$ é o fechamento reflexivo e transitivo de $\Rightarrow_G$. Finalmente, $L(G)$, a linguagem **gerada** por *G*, é $\{w \in \Sigma^*: S \Rightarrow_G^* w\}$; Diz-se também que *G* **gera** cada cadeia de $L(G)$. Uma linguagem *L* é dita **livre de contexto** se $L = L(G)$ para alguma gramática livre de contexto *G*.

Quando não houver dúvida sobre qual é a gramática que se está considerando, escreve-se $A \rightarrow w$ e $u \Rightarrow v$, em vez de $A \rightarrow_G w$ e $u \Rightarrow_G v$.

Chama-se qualquer seqüência da forma

$$w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$$

de **derivação**, em *G*, da cadeia w_n a partir de w_0. Aqui $w_0, \cdots, w_n$ podem ser quaisquer cadeias de V^*, e *n*, o **comprimento** da derivação, pode ser qualquer número natural, incluindo zero. Também se diz, neste caso, que a derivação tem *n* **passos**.

Exemplo 3.1.1: Considere-se a gramática livre de contexto $G = (V, \Sigma, R, S)$, onde $V = \{S, a, b\}$, $\Sigma = \{a, b\}$ e R consiste das regras $S \to aSb$ e $S \to \varepsilon$. Uma possível derivação é, neste caso,

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Neste exemplo, os primeiros dois passos de derivação aplicam a regra $S \to aSb$, enquanto o último emprega a regra $S \to \varepsilon$. De fato, não é difícil ver que $L(G) = \{a^n b^n : n \geq 0\}$. Daí se pode concluir que algumas linguagens livres de contexto não são regulares. ✧

Em breve será constatado, entretanto, que todas as linguagens regulares são livres de contexto.

Exemplo 3.1.2: Seja G a gramática (W, Σ, R, S), onde

$W = \{O, A, N, V, S\} \cup \Sigma$,
$\Sigma = \{$Jim, grande, verde, queijo, comeu$\}$,
$R = \{O \to NVN$,
$N \to S$,
$N \to NA$,
$A \to$ grande,
$A \to$ verde,
$N \to$ queijo,
$N \to$ Jim,
$V \to$ comeu$\}$

Aqui, G foi construída como uma gramática que representa uma parte do idioma português: O significa *oração*, A, *adjetivo*, S, *substantivo*, V, *verbo* e N, *sintagma nominal*. As seguintes são algumas cadeias em $L(G)$.

Jim comeu queijo
Jim grande comeu queijo verde
queijo grande comeu Jim

Infelizmente, as cadeias seguintes também são sentenças em $L(G)$:

Queijo grande comeu queijo grande verde grande verde verde
Jim verde comeu Jim grande verde

Exemplo 3.1.3: Programas de computador, escritos em qualquer linguagem de programação, devem satisfazer algum critério rígido a fim de serem considerados sintaticamente corretos e, portanto, compatíveis com uma interpretação mecânica. Felizmente, a sintaxe da maioria das linguagens de programação pode, diferentemente do que ocorre com as linguagens humanas, ser capturada por gramáticas livre de contexto. Devemos ver na Seção 3.7 que ser livre de contexto é extremamente útil quando se trata de analisar sintaticamente um programa, isto é, analisá-lo para entender sua sintaxe. Aqui, oferecemos uma gramática que gera um fragmento de muitas linguagens usuais de programação. Essa linguagem consiste em todas as cadeias sobre o alfabeto {(,), +, *, id} que representam expressões aritméticas

sintaticamente corretas que se utilizam dos operadores + e *. O símbolo id representa qualquer *identificador*, ou seja, nome de variável.† Exemplos dessas cadeias são id e id * (id * id + id), mas não *id + (ou + *id.

Suponha que $G = (V, \Sigma, R, E)$ onde V, Σ e R são como segue.

$V = \{+, *, (,), \text{id}, T, F, E\}$,
$\Sigma = \{+, *, (,), \text{id}\}$,
$R = \{E \to E + T$, (R1)
$E \to T$, (R2)
$T \to T * F$, (R3)
$T \to F$, (R4)
$F \to (E)$, (R5)
$F \to \text{id}\}$. (R6)

Os símbolos E, T e F são abreviações de *expressão*, *termo* e *fator*, respectivamente.

A gramática G gera a cadeia (id * id + id) * (id + id) pela seguinte derivação.

$E \Rightarrow T$	pela Regra R2
$\Rightarrow T * F$	pela Regra R3
$\Rightarrow T * (E)$	pela Regra R5
$\Rightarrow T * (E + T)$	pela Regra R1
$\Rightarrow T * (T + T)$	pela Regra R2
$\Rightarrow T * (F + T)$	pela Regra R4
$\Rightarrow T * (\text{id} + T)$	pela Regra R6
$\Rightarrow T * (\text{id} + F)$	pela Regra R4
$\Rightarrow T * (\text{id} + \text{id})$	pela Regra R6
$\Rightarrow F * (\text{id} + \text{id})$	pela Regra R4
$\Rightarrow (E) * (\text{id} + \text{id})$	pela Regra R5
$\Rightarrow (E + T) * (\text{id} + \text{id})$	pela Regra R1
$\Rightarrow (E + F) * (\text{id} + \text{id})$	pela Regra R4
$\Rightarrow (E + \text{id}) * (\text{id} + \text{id})$	pela Regra R6
$\Rightarrow (T + \text{id}) * (\text{id} + \text{id})$	pela Regra R2
$\Rightarrow (T * F + \text{id}) * (\text{id} + \text{id})$	pela Regra R3
$\Rightarrow (F * F + \text{id}) * (\text{id} + \text{id})$	pela Regra R4
$\Rightarrow (F * \text{id} + \text{id}) * (\text{id} + \text{id})$	pela Regra R6
$\Rightarrow (\text{id} * \text{id} + \text{id}) * (\text{id} + \text{id})$	pela Regra R6

Veja o Problema 3.1.8 para gramática livre de contexto que gera grandes subconjuntos de linguagens de programação. ✧

Exemplo 3.1.4: A seguinte gramática gera todas as cadeias que equilibram adequadamente os parênteses da esquerda com os da direita: cada parêntese esquerdo deve ser associado com um único subseqüente parêntese da direita, e cada parêntese direito pode ser pareado com um único parêntese

† A propósito, a descoberta desses identificadores (ou palavras reservadas da linguagem ou, ainda, constantes numéricas) pelo programa é realizada no estágio inicial da *análise léxica*, por algoritmos baseado em expressões regulares e autômatos finitos.

esquerdo precedente. Além disso, a cadeia compreendida entre qualquer um desses pares de parênteses deve apresentar a mesma propriedade. Seja $G = (V, \Sigma, R, S)$, onde

$$\begin{aligned} V &= \{S, (,)\}, \\ \Sigma &= \{(,)\}, \\ R &= \{S \to \varepsilon, \\ &\quad S \to SS, \\ &\quad S \to (S)\}. \end{aligned}$$

Duas das possíveis derivações nessa gramática são

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow (S)\,(()) \Rightarrow ()\,(())$$

e

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()\,((S)) \Rightarrow ()\,(())$$

Portanto, a mesma cadeia pode ter várias derivações, com base em uma dada gramática livre de contexto; na próxima subseção, serão discutidas as intrincadas maneiras pelas quais tais derivações podem ser relacionadas.

Observe-se que, $L(G)$ é outra linguagem livre de contexto que não é regular (o fato de que ela não é regular é explorado no Problema 2.4.6). ✧

Exemplo 3.1.5: Está claro que existem linguagens livre de contexto que não são regulares (já foram vistos dois exemplos). Entretanto, *todas as linguagens regulares são livres de contexto.* No decorrer deste capítulo, serão encontradas várias provas desse fato. Por exemplo, a Seção 3.3 mostra que linguagens livres de contexto são precisamente as linguagens aceitas por certos dispositivos reconhecedores de linguagens chamados *autômatos de pilha.* Agora deve-se também destacar que tal aceitador é uma generalização do autômato finito, no sentido de que qualquer autômato finito pode ser normalmente considerado como um caso particular de autômato de pilha. Portanto, pode-se dizer que todas as linguagens regulares são livres de contexto.

Para outra prova, devemos ver na Seção 3.5 que a classe de linguagens livres de contexto é fechada sob as operações de união, concatenação e fechamento (estrela) de Kleene (Teorema 3.5.1); além disso, as linguagens triviais $\emptyset$ e $\{a\}$ são, definitivamente, livres de contexto (respectivamente, geradas pela gramática livre de contexto sem regras, e pela gramática livre de contexto que contém somente a regra $S \to a$). Portanto, a classe de linguagens livres de contexto deve conter todas as linguagens regulares, e corresponde ao fechamento das linguagens normais sob essas operações.

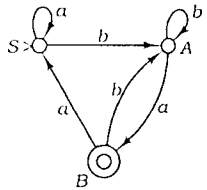


Figura 3-1

Mas vamos agora mostrar *por construção direta* que todas as linguagens regulares são livres de contexto. Considere-se a linguagem regular aceita pelo autômato finito determinístico $M = (K, \Sigma, \delta, s, F)$. A mesma linguagem é gerada pela gramática $G(M) = (V, \Sigma, R, S)$, onde $V = K \cup \Sigma$, $S = s$ e R consiste destas regras:

$$R = \{q \to ap : \delta(q,a) = p\} \cup \{q \to \varepsilon : q \in F\}.$$

Isto é, os não-terminais correspondem aos estados do autômato. Da mesma forma que ocorre com as regras, para cada transição de q para p, como resposta à entrada a, temos em R a regra $q \to ap$. Por exemplo, para o autômato na Figura 3-1 constroi-se a seguinte gramática:

$$S \to aS,\ S \to bA,\ A \to aB,\ A \to bA,\ B \to aS,\ B \to bA,\ B \to \varepsilon.$$

Deixa-se como exercício, mostrar que a gramática livre de contexto assim obtida gera precisamente a linguagem aceita pelo autômato (ver o Problema 3.1.10 para um tratamento geral de gramáticas livres de contexto como $G(M)$ acima, e seu parentesco com os autômatos finitos). ✧

Problemas para a Seção 3.1

3.1.1 Considere a gramática $G = (V, \Sigma, R, S)$, onde

$$\begin{aligned} V &= \{a, b, S, A\}, \\ \Sigma &= \{a, b\}, \\ R &= \{S \to AA, \\ &\quad A \to AAA, \\ &\quad A \to a, \\ &\quad A \to bA, \\ &\quad A \to Ab\}. \end{aligned}$$

(a) Quais cadeias de $L(G)$ podem ser produzidas por derivações de quatro passos ou menos?
(b) Forneça no mínimo quatro derivações distintas para a cadeia *babbab*.
(c) Para qualquer $m, n, p > 0$, descreva uma derivação em G da cadeia $b^m a b^n a b^p$.

3.1.2 Considere a gramática (V, Σ, R, S), onde V, Σ e R são definidas como segue:

$$\begin{aligned} V &= \{a, b, S, A\}, \\ \Sigma &= \{a, b\}, \\ R &= \{S \to aAa, \\ &\quad S \to bAb, \\ &\quad S \to \varepsilon, \\ &\quad A \to SS\}. \end{aligned}$$

Forneça uma derivação da cadeia *baabbb* em G. (Note que, diferente do que ocorre com todas as outras linguagens livres de contexto que foram vistas até aqui, essa é uma linguagem muito difícil de descrever em português.)

3.1.3 Construa uma gramática livre de contexto que defina cada uma das seguintes linguagens.
(a) $\{wcw^R: w \in \{a, b\}^*\}$
(b) $\{ww^R: w \in \{a, b\}^*\}$
(c) $\{w \in \{a, b\}^*: w = w^R\}$

3.1.4 Considere o alfabeto $\Sigma = \{a, b, (,), \cup, *, \varnothing\}$. Construa uma gramática livre de contexto que gere todas as cadeias em Σ^* que são expressões regulares sobre $\{a, b\}$.

3.1.5 Considere a gramática livre de contexto $G = (V, \Sigma, R, S)$, onde

$$\begin{aligned} V &= \{a, b, S, A, B\}, \\ \Sigma &= \{a, b\}, \\ R &= \{S \to aB, \\ &\quad S \to bA, \\ &\quad A \to a, \\ &\quad A \to aS, \\ &\quad A \to BAA, \\ &\quad B \to b, \\ &\quad B \to bS, \\ &\quad B \to ABB\}. \end{aligned}$$

(a) Mostre que $ababba \in L(G)$.
(b) Prove que $L(G)$ é o conjunto de todas as cadeias em $\{a, b\}$ que têm igual número de ocorrências de a e b.

3.1.6 Seja G uma gramática livre de contexto e que $k > 0$. Considere-se $L_k(G) \subseteq L(G)$ como o conjunto de todas as cadeias que têm uma derivação em G com k ou menos passos.
(a) O que é $L_5(G)$, onde G é a gramática do Exemplo 3.1.4
(b) Mostre que, para todas as gramáticas livre de contexto G e todos os $k > 0$, $L_k(G)$ é finito.

3.1.7 Suponha que $G = (V, \Sigma, R, S)$, onde $V = \{a, b, S\}$, $\Sigma = \{a, b\}$ e $R = \{S \to aSb, S \to aSa, S \to bSa, S \to bSb, S \to \varepsilon\}$. Mostre que $L(G)$ é regular.

3.1.8 Um programa em uma linguagem real de programação, como C ou Pascal, é composto de instruções de vários tipos:
(1) instrução de atribuição, na forma id:= E, onde E é qualquer expressão aritmética (gerada pela gramática do Exemplo 3.1.3).
(2) instrução condicional, por exemplo na forma if $E < E$ then instrução ou uma instrução while na forma while $E < E$ do instrução.
(3) instrução goto; além disso, cada instrução pode ser precedida por um rótulo.
(4) instrução composta, isto é, uma seqüência de instruções, separadas por um ";", precedida por um begin, seguida por um end.
Forneça uma gramática livre de contexto que gere todas as possíveis instruções na linguagem de programação simplificada acima descrita.

3.1.9 Prove que as seguintes linguagens são livres de contexto, mostrando gramáticas livres de contexto capazes de gerá-las.
(a) $\{a^m b^n: m \geq n\}$
(b) $\{a^m b^n c^p d^q: m + n = p + q\}$
(c) $\{w \in \{a,b\}^*: w$ tem duas vezes muitos b's como a's$\}$
(d) $\{uawb: u, w \in \{a, b\}^*, |u| = |w|\}$
(e) $\{w_1 c w_2 c \ldots c w_k c c w_j^R: k \geq 1, 1 \leq j \leq k, w_i \in \{a, b\}^+$ para $i = 1, \ldots, k\}$
(f) $\{a^m b^n: m \leq 2n\}$

3.1.10 Uma gramática livre de contexto $G = (V, \Sigma, R, S)$ **é dita regular** (ou **linear à direita**), se $R \subseteq (V - \Sigma) \times \Sigma^*((V - \Sigma) \cup \{\varepsilon\})$; isto é, se cada transição tem um lado direito que consiste em uma cadeia de terminais seguida por, no máximo, um não-terminal.
(a) Considere a gramática regular $G = (V, \Sigma, R, S)$, onde

$$\begin{aligned} V &= \{a, b, A, B, S\} \\ \Sigma &= \{a,b\} \\ R &= \{S \to abA, S \to B, S \to baB, S \to \varepsilon, \\ &\quad A \to bS, B \to aS, A \to b\}. \end{aligned}$$

Construa um autômato finito não-determinístico M, tal que $L(M) = L(G)$. Levante a seqüência de transições de M que conduzem à aceitação da cadeia $abba$ e compare com uma derivação da mesma cadeia em G.
(b) Prove que uma linguagem é regular se e somente se houver uma gramática regular capaz de gerá-la. (*Sugestão*: Lembre-se do Exemplo 3.1.5.)
(c) Uma gramática livre de contexto $G = (V, \Sigma, R, S)$ é dita **linear à esquerda**, se, e somente se, $R \subseteq (V - \Sigma) \times ((V - \Sigma) \cup \{\varepsilon\})\Sigma^*$. Mostre que uma linguagem é regular se e somente se ela for gerada por alguma gramática linear à esquerda.
(d) Seja $G = (V, \Sigma, R, S)$ uma gramática livre de contexto tal que cada regra em R seja da forma $A \to wB$, $A \to B\varepsilon$ ou $A \to \varepsilon$, onde, em cada caso $A, B \in V - \Sigma$ e $w \in \Sigma^*$. É $L(G)$ necessariamente regular? Prove isso, ou apresente um contra-exemplo.

3.2 ÁRVORES DE ANÁLISE SINTÁTICA

Seja G uma gramática livre de contexto. Uma cadeia $w \in L(G)$ pode apresentar muitas derivações em G. Por exemplo, se G for a gramática livre de contexto que gera a linguagem de parênteses equilibrados (Veja o Exemplo 3.1.4), então a cadeia () () pode ser derivada de S por meio de, no mínimo, duas derivações distintas, a saber,

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

e

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow () ()$$

Entretanto, essas duas derivações são, em um certo sentido, "as mesmas". As regras utilizadas são as mesmas e aplicam-se nos mesmos trechos da cadeia intermediária. A única diferença, neste caso, está na *ordem* em

que as regras são aplicadas. Intuitivamente, ambas as derivações podem ser representadas como mostra a Figura 3-2.

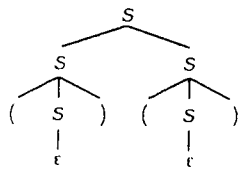


Figura 3-2

A esse diagrama dá se o nome de **árvore de análise sintática**, os nós dessa árvore são chamados **vértices**; cada vértice traz um **rótulo**, que é um símbolo em V. O vértice mais alto é chamado **raiz** e os vértices periféricos são chamados **folhas**. Todas as folhas são identificadas por *terminais* ou eventualmente pela cadeia vazia ε. Concatenando os rótulos das folhas, da esquerda para a direita, obtemos uma cadeia de terminais, a qual é chamada **resultado** da árvore de análise sintática.

Mais formalmente, para uma gramática arbitrária livre de contexto $G = (V, \Sigma, R, S)$, definimos suas árvores de análise sintática e suas raízes, folhas e resultados, como segue:

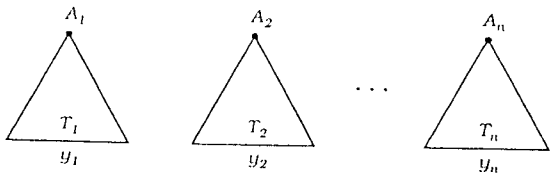
1. $\circ\, a$

Essa é uma árvore de análise sintática para cada $a \in \Sigma$. O único vértice dessa árvore de análise sintática é a raiz e também é uma folha. O resultado dessa árvore de análise sintática é a.

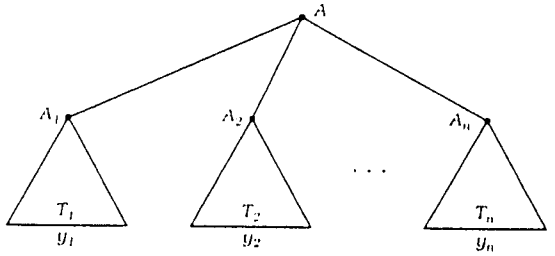
2. Se $A \to \varepsilon$ é uma regra em R, então

é uma árvore de análise sintática; sua raiz é o vértice rotulado A, sua única folha é o vértice rotulado ε, e seu resultado é ε, a cadeia vazia.

3. Se



forem árvores de análise sintática, onde $n \geq 1$, com raízes rotuladas $A_1, \cdots, A_n$, respectivamente, e com resultados $y_1, \cdots, y_n$, e se $A \to A_1 \ldots A_n$ for uma regra em R, então



será também uma árvore de análise sintática. Sua raiz será um novo vértice, rotulado A, suas folhas serão as folhas das árvores de análise sintática que a constituem e seu resultado será $y_1 \ldots y_n$.

4. Nada mais é considerado árvore de análise sintática.

Exemplo 3.2.1: Lembre-se da gramática G, que gera todas as expressões aritméticas sobre id (Exemplo 3.1.3). Uma árvore de análise sintática com resultado id * (id + id) é mostrada na Figura 3-3. ✧

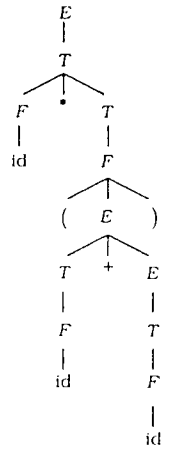


Figura 3-3

Intuitivamente, árvores de análise sintática constituem uma forma de representar essas derivações das cadeia de $L(G)$, de tal modo que sejam suprimidas as diferenças superficiais entre derivações, devidas à ordem de aplicação das regras. Para colocar de outro modo, as árvores de análise sintática representam *classes de equivalência de derivações*. Tornamos essa intuição mais precisa a seguir.

Seja $G = (V, \Sigma, R, S)$ uma gramática livre de contexto. Sejam $D = x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n$ e $D' = x'_1 \Rightarrow x'_2 \Rightarrow \cdots x'_n$ duas derivações em G, onde $x_i, x'_i \in V^*$ para $i = 1, \ldots, n$, $x_1, x'_1 \in V - \Sigma$ e $x_n, x'_n \in \Sigma^*$. Em outras palavras, ambas são derivações de uma cadeia de terminais a partir de um não-terminal simples. Dizemos que D **precede** D' (denota-se como $D \prec D'$) se $n > 2$ e se houver um inteiro k, $1 < k < n$, tal que

(1) para todo $i \neq k$ valer $x_i = x'_i$;
(2) $x_{k-1} = x'_{k-1} = uAvBw$, onde $u, v, w \in V^*$, e $A, B, \in V - \Sigma$;
(3) $x_k = uyvBw$, onde $A \rightarrow y \in R$;
(4) $x'_k = uAvzw$ onde $B \rightarrow z \in R$;
(5) $x_{k+1} = x'_{k+1} = uyvzw$.

Assim, as duas derivações são idênticas, exceto quanto a dois passos consecutivos, durante os quais os dois não-terminais são substituídos pelas mesmas duas cadeias, *mas em ordem invertida nas duas derivações*. Diz-se, nesse caso, que a derivação na qual o não-terminal mais à esquerda é substituído em primeiro lugar, *precede* a outra.

Exemplo 3.2.2: Considerem-se as três derivações seguintes: D_1, D_2 e D_3, na gramática G, que gera todas as cadeias de parênteses equilibrados:

$$D_1 \Rightarrow S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$$
$$D_2 \Rightarrow S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$$
$$D_3 \Rightarrow S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$$

Temos que $D_1 \prec D_2$ e $D_2 \prec D_3$. Entretanto, *não* é verdade que $D_1 \prec D_3$, uma vez que as duas últimas derivações diferem em mais de uma cadeia intermediária. Note-se que as três derivações apresentam a mesma árvore de análise sintática, como mostrado na Figura 3-4. ✧

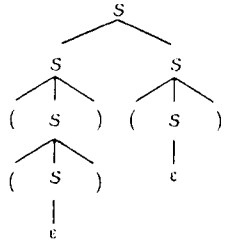


Figura 3-4

Diz-se que duas derivações D e D' são **similares** quando o par (D, D') pertencer ao *fechamento transitivo reflexivo simétrico de* $\prec$. Como o fechamento transitivo reflexivo simétrico de qualquer relação é por definição reflexivo, simétrico e transitivo, a similaridade será uma relação de equivalência. Em outras palavras, duas derivações são similares se elas puderem ser transformadas uma na outra por meio de uma seqüência de alterações na ordem em que as regras são aplicadas. Tal alteração pode substituir uma derivação por alguma outra que a preceda, ou então por alguma outra que ela precede.

Exemplo 3.2.2 (continuação): As árvores de análise sintática capturam exatamente, via um isomorfismo natural, as classes de equivalência da relação de "similaridade" entre derivações de uma cadeia, conforme foi definido acima. A classe de equivalência das derivações de (())(), correspondente à árvore na Figura 3-4, contém as derivações D_1, D_2, D_3 mostradas acima, e também as seguintes outras sete:

$$D_4 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$$
$$D_5 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$$
$$D_6 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$
$$D_7 = S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$$
$$D_8 = S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$$
$$D_9 = S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$
$$D_{10} = S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$

Essas dez derivações são relacionadas por $\prec$ como mostrado na Figura 3-5.

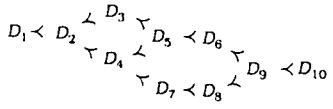


Figura 3-5

Todas essas dez derivações são similares, porque, informalmente, elas representam aplicações das mesmas regras sobre a mesma região das cadeias, somente diferindo na ordem relativa dessas aplicações; equivalentemente, pode-se ir de qualquer uma delas para qualquer outra seguindo repetidamente um $\prec$ ou um $\prec$ inverso. Não há outras derivações similares a essas.

Há, entretanto, para o exemplo acima, outras derivações de (())() que não são similares – e, portanto, não são capturadas pela árvore de análise sintática mostrada na Figura 3-4. Um exemplo é a seguinte derivação: $S \Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S((S))S \Rightarrow S(())S \Rightarrow S(())(S) \Rightarrow S(())() \Rightarrow (())()$. Sua árvore de análise sintática é mostrada na Figura 3-6 (compare com a Figura 3-4). ✧

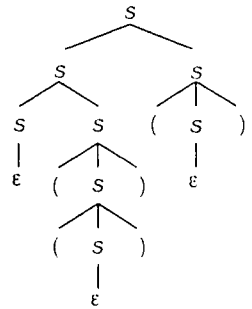


Figura 3-6

Cada classe de equivalência quanto à similaridade de derivações, isto é, cada árvore de análise sintática, contém uma derivação que é máxima sob $\prec$; ou seja, ela não é precedida por qualquer outra derivação. Essa derivação é chamada **derivação mais à esquerda**. Existe uma derivação mais à esquerda em cada árvore de análise sintática, e pode ser obtida conforme descrito a seguir. Começando pelo rótulo da raiz A, substituem-se, repetidamente os não-terminais mais à esquerda na cadeia corrente, de acordo com a regra sugerida pela árvore de análise sintática. Analogamente, uma **derivação mais à direita** é aquela que não precede qualquer outra derivação; ela pode ser obtida, da árvore de análise sintática, expandindo-se sempre os não-terminais mais à direita na cadeia corrente. Cada árvore de análise sintática tem uma e apenas uma derivação mais à esquerda e uma e apenas uma derivação mais à direita. Isso ocorre porque a derivação mais à esquerda de uma árvore de análise sintática é univocamente determinada, uma vez que, a cada passo, há exatamente um não-terminal para ser substituído: aquele que figurar mais à esquerda na cadeia. Ocorre o mesmo em relação à derivação mais à direita. No exemplo acima, D_1 é uma derivação mais à esquerda, e D_{10} é uma derivação mais à direita.

É fácil dizer quando um passo de derivação pode ser parte de uma derivação mais à esquerda: é o não-terminal mais à esquerda que deve ser substituído. Escreve-se $x \overset{L}{\Rightarrow} y$ se e somente se $x = wA\beta$, $y = w\alpha\beta$, onde $w \in \Sigma^*$, $\alpha, \beta \in V^*$, $A \in V - \Sigma$ e $A \to \alpha$ é uma regra de G. Portanto, se $x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n$ é uma derivação mais à esquerda, então, de fato, $x_1 \overset{L}{\Rightarrow} x_2 \overset{L}{\Rightarrow} \cdots \overset{L}{\Rightarrow} x_n$. Uma configuração análoga se aplica às derivações mais à direita (a notação é $x \overset{R}{\Rightarrow} y$).

Para resumir as idéias sobre árvores de análise sintática e derivações, apresentadas nesta seção, apresenta-se, sem prova formal, o seguinte teorema.

Teorema 3.2.1: *Seja* $G = (V, \Sigma, R, S)$ *uma gramática livre de contexto, e sejam* $A \in V - \Sigma$ *e* $w \in \Sigma^*$. *Então, as seguintes afirmações são equivalentes:*

(a) $A \Rightarrow^* w$.
(b) Existe uma árvore de análise sintática com raiz A *e resultado* w.
(c) Existe uma derivação mais à esquerda $A \overset{L}{\Rightarrow}^* w$.
(d) Existe uma derivação mais à direita $A \overset{R}{\Rightarrow}^* w$.

Ambigüidade

Foi visto, no Exemplo 3.2.2, que pode haver cadeias da linguagem geradas por uma gramática livre de contexto que apresentam derivações *não* similares, ou seja, derivações *com árvores de análise sintática distintas*, ou, equivalentemente, com derivações mais à direita distintas (e derivações mais à esquerda distintas). Como exemplo mais concreto, lembre-se da gramática G que gera todas as expressões aritméticas sobre id no Exemplo 3.1.3 e considere outra gramática, G', que gera a mesma linguagem, através do seguinte conjunto de regras:

$$E \to E + E,\quad E \to E * E,\quad E \to (E),\quad E \to \text{id}.$$

Não é difícil ver que $L(G') = L(G)$. Além disso, há importantes diferenças entre G e G'. Intuitivamente, a gramática G', eliminando a distinção explícita entre fatores (F) e termos (T), dá margem a uma interpretação inadequada quanto à precedência da multiplicação sobre adição. De fato, há duas árvores de análise sintática para a expressão id + id * id em G', ambas mostradas na Figura 3-7. Uma delas, 3-7(a), corresponde ao significado natural dessa expressão (com a operação * assumindo precedência sobre +). A outra fornece uma interpretação incorreta.

Gramáticas como G', com cadeias que apresentam duas ou mais árvores de análise sintática distintas, são ditas **ambíguas**. Como será visto extensivamente na Seção 3.7, determinar uma árvore de análise sintática para uma dada cadeia da linguagem – isto é, *analisar sintaticamente* tal cadeia – é o primeiro passo importante em direção ao entendimento da estrutura da cadeia e o motivo pelo qual ela pertence à linguagem – em última instância, sua interpretação. Isso tem, naturalmente, uma especial importância no caso das gramáticas como G e G' acima, que geram fragmentos de linguagens de programação. Gramáticas ambíguas tais como G' não são desejáveis para análise sintática, uma vez que elas não determinam uma única árvore de análise sintática ou seja, uma única interpretação, para cada cadeia da linguagem.

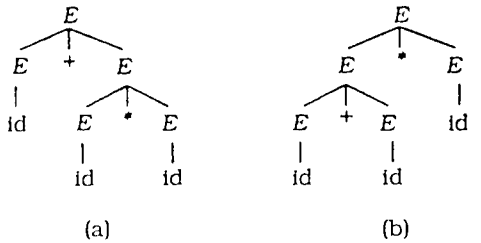


Figura 3-7

Felizmente há, neste caso, um modo de eliminar a ambigüidade de G', introduzindo os novos não-terminais T e F (lembre-se da gramática G do Exemplo 3.1.3). Em outras palavras, existe nesse caso uma gramática *não-ambígua*, que gera a mesma linguagem (trata-se da gramática G, definida no Exemplo 3.1.3; para uma prova de que a gramática G é, de fato, não-ambígua, veja o Problema 3.2.1). Analogamente, a gramática dada no Exemplo 3.1.4, que gera cadeias com parênteses balanceados, a qual também é ambígua (conforme foi discutido no final do Exemplo 3.2.2), pode ser facilmente tornada não-ambígua (veja o Problema 3.2.2). Por outro lado, existem linguagens livres de contexto tais que *todas as* gramáticas livres de contexto que as geram são obrigatoriamente ambíguas. Tais linguagens são chamadas **inerentemente ambíguas**. Felizmente, linguagens de programação nunca são inerentemente ambíguas.

Problemas para a Seção 3.2

3.2.1 Mostre que não é ambígua a gramática livre de contexto G, dada no Exemplo 3.1.3, a gual gera todas as expressões aritméticas sobre id.

3.2.2 Mostre que é ambígua a gramática livre de contexto dada no Exemplo 3.1.4, a qual gera todas as cadeias de parênteses balanceadas. Apresente uma gramática equivalente, não-ambígua.

3.2.3 Considere a gramática do Exemplo 3.1.3. Forneça duas derivações da cadeia id * id + id: uma que seja mais à esquerda e outra que não seja mais à esquerda.

3.2.4 Desenhe árvores de análise sintática para cada afirmação.
(a) A Gramática do Exemplo 3.1.2 e a cadeia "*Jim grande comeu queijo verde*".
(b) A Gramática do Exemplo 3.1.3 e as cadeias id + (id + id) * id e (id * id + id * id).

3.3 AUTÔMATOS DE PILHA

Nem toda linguagem livre de contexto pode ser reconhecida por um autômato finito pois, como já vimos, algumas delas não são regulares. Que tipo de dispositivo mais poderoso poderia ser utilizado para reconhecer linguagens livres de contexto arbitrárias? Ou, mais especificamente, que propriedade precisamos adicionar ao autômato finito para que ele possa aceitar qualquer linguagem livre de contexto?

Utilizando um exemplo particular, considere-se a linguagem $\{ww^R: w \in \{a, b\}^*\}$. Ela é livre de contexto, pois é gerada pela gramática com as regras $S \rightarrow aSa$, $S \rightarrow bSb$ e $S \rightarrow \varepsilon$. É intuitivo que qualquer dispositivo que reconheça as cadeias dessa linguagem, lendo-as da esquerda para a direita, deve memorizar a primeira metade da cadeia de entrada para poder posteriormente compará-la – na ordem inversa – com a sua segunda metade. Não surpreende que essa função não possa ser executada por um autômato finito. Se, entretanto, a máquina fosse capaz de acumular sua cadeia de entrada à medida que esta vai sendo lida, acrescentando símbolos, um por vez, a uma cadeia memorizada (veja a Figura 3-8), então essa máquina poderia, não-deterministicamente, determinar o momento em que o centro da entrada foi alcançado e, daí em diante, comparar um de cada vez, sucessivamente, os símbolos retirados da sua memória. O dispositivo de armazenamento não precisa ser de propósito geral. É suficiente uma pilha, a qual, permita acesso de leitura e gravação apenas ao símbolo contido em seu topo*.

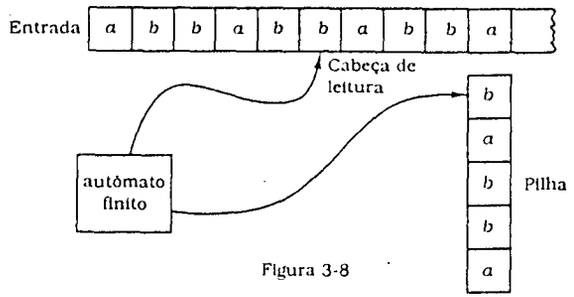


Figura 3-8

* N. de R.T. Em inglês, o termo "pilha" tem duas formas usuais: "pushdown store" – neste caso o único acesso direto permitido ocorre no topo da área de armazenamento e "stack" – neste caso, é possível acessar diretamente elementos que não estejam no topo.

Como um outro exemplo, pode-se considerar o conjunto de cadeias de parênteses balanceados (Exemplo 3.1.4), que também não é regular. Entretanto, os programadores de computador estão familiarizados com um algoritmo simples para reconhecer essa linguagem: começando em zero, adicione a uma variável uma unidade a cada parênteses esquerdo e subtraia uma unidade a cada parênteses direito encontrado. Se o contador se tornar negativo em qualquer momento ou se ao final não for nulo, então a cadeia deve ser rejeitada por não estar balanceada. Caso contrário, ela deve ser aceita. Observe-se que um contador pode ser considerado como um caso muito particular de uma pilha, no qual somente um tipo de símbolo pode ser gravado.

Tratando essa questão com um outro enfoque, regras como $A \rightarrow aB$ são fáceis de simular com um autômato finito, da seguinte forma: "Estando no estado A, recebendo o símbolo a, vá para o estado B". O que se deve fazer ao encontrar uma regra cujo lado direito não seja um terminal seguido por um não-terminal, como é o caso, por exemplo, da regra $A \rightarrow aBb$? Certamente a máquina deve ir novamente do estado A para o estado B, lendo a, mas, o que se deve fazer quanto ao símbolo b? Que ação adicional nos permitiria lembrar da presença de b nessa regra? Uma pilha seria bastante conveniente aqui: adicionando-se b no topo de tal pilha, pode-se memorizar esse símbolo, e levá-lo em consideração ao ser encontrado novamente na cadeia de entrada.

Essa idéia de um autômato, com uma pilha funcionando como armazenamento auxiliar, pode ser formalizada como segue:

Definição 3.3.1: Define-se um **autômato de pilha** como uma sêxtupla $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde
K é um conjunto finito de **estados**,
Σ é um alfabeto (os **símbolos de entrada**),
Γ é um alfabeto (os **símbolos de pilha**),
$s \in K$ é o **estado inicial**,
$F \subseteq K$ é o conjunto de **estados finais**, e
Δ, a **relação de transição**, é um subconjunto finito de $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*)$.

Intuitivamente, se $((p, a, \beta), (q, \gamma)) \in \Delta$, então sempre que o autômato M estiver no estado p, com β no topo da pilha, poderá ler o símbolo a da fita de entrada (se $a = \varepsilon$, então a entrada não será consultada), substituindo β por γ no topo da pilha e passando para o estado q. O par $((p, a, \beta), (q, \gamma))$ é chamado **transição** de M; considerando que várias transições de M possam ser simultaneamente aplicadas, as máquinas assim descritas serão não-determinísticas em sua operação. (Devemos mais tarde alterar essa definição para definir um dispositivo mais restrito, o autômato de pilha determinístico.)

Utiliza-se o termo **empilhar** para indicar que um símbolo deve ser **adicionado** ao topo da pilha; e o termo **desempilhar** para indicar que um símbolo deve ser **removido** do topo da pilha. Por exemplo, a transição $((p, u, \varepsilon), (q, a))$ empilha o símbolo a, enquanto $((p, u, a), (q, \varepsilon))$ desempilha o símbolo a.

Da mesma forma que ocorre com autômatos finitos, durante uma computação, a parte já lida da entrada não mais afeta a operação subseqüente da máquina. Correspondentemente, uma configuração de um autômato de pilha é definida como sendo um membro de $K \times \Sigma^* \times \Gamma^*$: o primeiro componente corresponde ao estado da máquina, o segundo, à parte da entrada que

ainda não foi lida, e o terceiro, ao conteúdo previamente armazenado da pilha, lido a partir do seu topo. Por exemplo, se a configuração fosse (q, w, abc), o símbolo a estaria no topo da pilha e o símbolo c, na parte inferior da mesma. Se (p, x, α) e (q, y, ς) são configurações de M, dizemos que (p, x, α) **produz, em um passo, o resultado** (q, y, ς) (notação: $(p, x, \alpha) \vdash_M (q, y, \varsigma)$ se há uma transição $((p, a, \beta), (q, \gamma)) \in \Delta$ tal que $x = ay$, $\alpha = \beta\eta$ e $\varsigma = \gamma\eta$ para algum $\eta \in \Gamma^*$. Denotamos o fechamento transitivo reflexivo de $\vdash_M$ por $\vdash_M^*$. Dizemos que M **aceita** uma cadeia $w \in \Sigma^*$ se e somente se $(s, w, \varepsilon) \vdash_M^* (p, \varepsilon, \varepsilon)$ para algum estado $p \in F$. Em outras palavras, M aceita uma cadeia w se e somente se houver uma seqüência de configurações $C_0, C_1, \ldots, C_n$ $(n > 0)$ tal que $C_0 \vdash_M C_1 \vdash_M \ldots \vdash_M C_n$, $C_0 = (s, w, \varepsilon)$ e $C_n = (p, \varepsilon, \varepsilon)$ para algum $p \in F$. Qualquer seqüência de configurações $C_0, C_1, \ldots, C_n$ tal que $C_i \vdash_M C_{i+1}$ para $i = 0, \ldots, n-1$ será denominada uma **computação** executada por M, e dizemos que tal computação tem **comprimento** n, ou seja, que tem n **passos**. A linguagem aceita por M, denotada por $L(M)$, é o conjunto de todas as cadeias aceitas por M.

Para não complicar a notação, sempre que não houver ambigüidade, escrevemos $\vdash$ e $\vdash^*$ em lugar de $\vdash_M$ e $\vdash_M^*$.

Exemplo 3.3.1: Vamos projetar um autômato de pilha M para aceitar a linguagem $L = \{wcw^R: w \in \{a, b\}^*\}$. Por exemplo, $ababcbaba \in L$, mas $abcab \notin L$ e $cbc \notin L$. Seja $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde $K = \{s, f\}$, $\Sigma = \{a, b, c\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, e Δ contém as seguintes cinco transições:

(1) $((s, a, \varepsilon), (s, a))$
(2) $((s, b, \varepsilon), (s, b))$
(3) $((s, c, \varepsilon), (f, \varepsilon))$
(4) $((f, a, a), (f, \varepsilon))$
(5) $((f, b, b), (f, \varepsilon))$

Esse autômato opera do seguinte modo: à medida que ele lê a primeira metade de sua entrada, o autômato permanece em seu estado inicial s, e usa as transições 1 e 2 para transferir os símbolos lidos da cadeia de entrada para a pilha. Note-se que essas transições são aplicadas sem levar em conta o conteúdo correntemente armazenado na pilha, uma vez que é vazia a cadeia a ser comparada com o conteúdo do topo dessa pilha. Quando a máquina encontra um símbolo c na cadeia de entrada, move-se do estado s para o estado f, sem utilizar-se da sua pilha. Daí em diante, somente as transições 4 e 5 poderão ser aplicadas, pois permitem a remoção do símbolo contido no topo da pilha, que deveria ser o mesmo encontrado no próximo símbolo da cadeia de entrada. Se o símbolo corrente de entrada não corresponder ao símbolo previamente armazenado no topo da pilha, nenhuma operação adicional será possível. Se o autômato alcançar a configuração $(f, \varepsilon, \varepsilon)$ – estado final, fim da cadeia de entrada, pilha vazia – então a entrada fornecida estava, de fato na forma wcw^R, e o autômato deve aceitá-la; por outro lado, se o autômato detectar uma não-correspondência entre símbolos da cadeia de entrada e da pilha, ou então se a entrada se esgotar antes de a pilha ter sido esvaziada, então não haverá aceitação da cadeia de entrada apresentada.

Para ilustrar a operação de M, apresenta-se a seqüência de transições executada pela máquina acima quando alimentada pela cadeia de entrada $abbcbba$.

Estado	Entrada ainda não-lida	Pilha	Transição utilizada
s	$abbcbba$	ε	-
s	$bbcbba$	a	1
s	$bcbba$	ba	2
s	$cbba$	bba	2
f	bba	bba	3
f	ba	ba	5
f	a	a	5
f	ε	ε	4

Exemplo 3.3.2: Neste exemplo, construiremos um autômato de pilha para aceitar $L = \{ww^R: w \in \{a, b\}^*\}$. Assim, as cadeias aceitas por essa máquina são similares àquelas aceitas pela máquina do exemplo anterior, exceto pela ausência do símbolo c, que marcava o centro das cadeias fornecidas. Portanto, a nova máquina deverá "adivinhar" o momento em que irá alcançar o centro da cadeia de entrada, e mover-se do estado s para o estado f, de um modo não-determinístico. Portanto, $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde $K = \{s, f\}$, $\Sigma = \{a, b\}$, $F = \{f\}$, e Δ é o conjunto das seguintes cinco transições.

(1) $((s, a, \varepsilon), (s, a))$
(2) $((s, b, \varepsilon), (s, b))$
(3) $((s, \varepsilon, \varepsilon), (f, \varepsilon))$
(4) $((f, a, a), (f, \varepsilon))$
(5) $((f, b, b), (f, \varepsilon))$

Assim, essa máquina é idêntica à do último exemplo, exceto quanto à transição 3. Sempre que a máquina estiver no estado s, poderá não-deterministicamente, escolher entre adicionar o próximo símbolo de entrada à pilha ou mover-se para o estado f sem consumir qualquer símbolo da entrada. Deste modo, em cada começo de uma cadeia da forma ww^R, M irá executar computações que não a conduzirão à configuração $(f, \varepsilon, \varepsilon)$; mas existirá alguma computação que leva M para essa configuração se, e somente se, a cadeia de entrada for da forma ww^R. ✧

Exemplo 3.3.3: O autômato de pilha deste exemplo aceita a linguagem $\{w \in \{a, b\}^*: w$ tem o mesmo número de a's e b's$\}$. Para isto, ou uma cadeia de a's, ou uma cadeia de b's, é mantida por M em sua pilha. Uma cadeia de a's na pilha indica o excesso de a's sobre b's lidos até o momento, e uma pilha de b's indica o excesso de b's em relação aos a's lidos. Em qualquer caso, M mantém um símbolo especial c na parte inferior da pilha, que funciona como um marcador. Seja $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde $K = \{s, q, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, c\}$, $F = \{f\}$, e Δ é o conjunto listado a seguir.

(1) $((s, \varepsilon, \varepsilon), (q, c))$
(2) $((q, a, c), (q, ac))$
(3) $((q, a, a), (q, aa))$
(4) $((q, a, b), (q, \varepsilon))$
(5) $((q, b, c), (q, bc))$
(6) $((q, b, b), (q, bb))$
(7) $((q, b, a), (q, \varepsilon))$
(8) $((q, \varepsilon, c), (f, \varepsilon))$

A transição 1 leva M ao estado q, além de introduzir o marcador c na parte inferior da pilha. No estado q, quando M lê um a, inicia uma pilha de a's a partir da parte inferior, preservando o marcador c (transição 2), ou então adiciona um símbolo a a uma pilha de a's (transição 3), ou ainda remove um símbolo b de uma pilha de b's (transição 4). Ao ler um b da entrada, a máquina opera de forma análoga, adicionando um b á pilha de b's ou à pilha vazia que consiste apenas de um marcador c na sua parte inferior, ou então removendo um a de uma pilha de a's (transições 5, 6 e 7, respectivamente). Por fim, quando o símbolo c figura no topo da pilha (e for, portanto, o único), a máquina M pode removê-lo e passar para um estado final (transição 8). Se nesse ponto todas as entradas já tiverem sido lidas, então a configuração $(f, \varepsilon, \varepsilon)$ foi alcançada, e a cadeia de entrada é aceita.

A tabela abaixo ilustra a operação de M.

Estado	Entradas não lidas	Pilha	Transição	Comentários
s	$abbbabaa$	ε	-	Configuração inicial.
q	$abbbabaa$	c	1	Marcador.
q	$bbbabaa$	ac	2	Começa uma pilha de a's.
q	$bbabaa$	c	7	Remove um a.
q	$babaa$	bc	5	Começa uma pilha de b's.
q	$abaa$	bbc	6	
q	baa	bc	4	
q	aa	bbc	6	
q	a	bc	4	
q	ε	c	4	
f	ε	ε	8	Aceita.

✧

Exemplo 3.3.4: Qualquer autômato finito pode ser considerado como um autômato de pilha que nunca faz uso de sua pilha. Para ser preciso, seja $M = (K, \Sigma, \Delta, s, F)$ um autômato finito não-determinístico e M' o autômato de pilha $(K, \Sigma, \varnothing, \Delta', s, F)$, onde $\Delta' = \{((p, u, \varepsilon), (q, \varepsilon)): (p, u, q) \in \Delta\}$. Em outras palavras, M' sempre adiciona e remove uma cadeia vazia à sua pilha, simulando, dessa maneira, as transições de M. Essa é uma forma direta de verificar que M e M' aceitam precisamente a mesma linguagem. ✧

Problemas para a Seção 3.3

3.3.1 Considere-se o autômato de pilha $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde

$K = \{s, f\}$,
$F = \{f\}$,
$\Sigma = \{a, b\}$,
$\Gamma = \{a\}$,
$\Delta = \{((s, a, \varepsilon), (s, a)), ((s, b, \varepsilon), (s, a)), ((s, a, \varepsilon), (f, \varepsilon)),$
$((f, a, a), (f, \varepsilon)), ((f, b, a), (f, \varepsilon))\}$.

(a) Determine todas as possíveis seqüências de transições de M para a entrada aba.
(b) Mostre que $aba, aa, abb \in L(M)$, mas $baa, bab, baaaa \notin L(M)$.
(c) Descreva $L(M)$ em português.

3.3.2 Construa um autômato de pilha que aceite cada uma das seguintes linguagens:
(a) A linguagem gerada pela gramática $G = (V, \Sigma, R, S)$, onde

$V = \{S, (,), [,]\}$,
$\Sigma = \{(,), [,]\}$,
$R = \{S \to \varepsilon,$
$S \to SS,$
$S \to [S],$
$S \to (S)\}$.

(b) A linguagem $\{a^m b^n: m \leq n \leq 2m\}$.
(c) A linguagem $\{w \in \{a, b\}^*: w = w^R\}$.
(d) A linguagem $\{w \in \{a, b\}^*: w$ tem duas vezes mais b's do que a's$\}$.

3.3.3 Seja $M = (K, \Sigma, \Gamma, \Delta, s, F)$ um autômato de pilha. A linguagem aceita **por M pelo critério do estado final** é definida como segue:

$$L_f(M) = \{w \in \Sigma^*: (s, w, \varepsilon) \vdash_M^* (f, \varepsilon, \alpha) \text{ para algum } f \in F, \alpha \in \Gamma^*\}.$$

a) Mostre que existe um autômato de pilha M', tal que $L(M') = L_f(M)$.
b) Mostre que existe um autômato de pilha M'', tal que $L_f(M'') = L(M)$.

3.3.4 Seja que $M = (K, \Sigma, \Gamma, \Delta, s, F)$ um autômato de pilha. A linguagem aceita **por M pelo critério da pilha vazia** é definida como segue:

$$L_\varepsilon(M) = \{w \in \Sigma^*: (s, w, \varepsilon) \vdash_M^* (q, \varepsilon, \varepsilon) \text{ para algum } q \in K\}.$$

(a) Mostre que existe um autômato de pilha M', tal que $L_\varepsilon(M') = L(M)$.
(b) Mostre que existe um autômato de pilha M'', tal que $L(M'') = L_\varepsilon M$.
(c) Mostre por um contra-exemplo que nem sempre $L_\varepsilon(M) = L(M) \cup \{\varepsilon\}$.

3.4 AUTÔMATOS DE PILHA E GRAMÁTICAS LIVRES DE CONTEXTO

Nesta seção, mostraremos que o autômato de pilha apresenta-se como formalismo necessário e suficiente para aceitar linguagens arbitrárias livres de contexto. Esse fato é de grande importância, tanto matemática como prática: matemática, porque oferece duas visualizações formais diferentes da mesma classe de linguagens; e prática, porque estabelece as bases do estudo de analisadores sintáticos para linguagens livres de contexto encontradas no dia-a-dia, como é o caso das linguagens de programação† (veja mais na Seção 3.7).

Teorema 3.4.1: *A classe de linguagens aceitas por autômatos a pilha é exatamente a classe de linguagens livres de contexto.*

† N. de R.T. Na verdade, as linguagens de programação não são simples linguagens livres de contexto. O autor alude, neste parte, às formalizações usualmente encontradas das linguagens de programação, em que somente são formalizados os aspectos livres de contexto, formando assim versões simplificadas e portanto incompletas de linguagens. Todas as características adicionais da linguagem costumam, neste caso, ser descritas à parte, usualmente através de exemplos ou então de textos em linguagem natural.

Prova: Dividimos essa prova em duas partes.

Lema 3.4.1: *Toda linguagem livre de contexto é aceita por algum autômato de pilha.*

Prova: Seja que $G = (V, \Sigma, R, S)$ seja uma gramática livre de contexto: devemos construir um autômato de pilha M, tal que $L(M) = L(G)$. A máquina que construímos tem somente dois estados, p e q, e permanece sempre no estado q após seu primeiro movimento. Além disso, M usa V, o conjunto de terminais e não-terminais, como seu alfabeto de pilha. Seja $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, onde Δ contém as seguintes transições:

(1) $((p, \varepsilon, \varepsilon), (q, S))$
(2) $((q, \varepsilon, A), (q, x))$ para cada regra $A \rightarrow x$ em R.
(3) $((q, a, a), (q, \varepsilon))$ para cada $a \in \Sigma$.

O autômato de pilha M inicia sua operação empilhando S, o símbolo inicial de G, à sua pilha, inicialmente vazia, e posicionando-se no estado q (transição 1). A cada passo subseqüente, M substitui o símbolo A do topo da pilha pela cadeia x, que forma o lado direito de alguma regra $A \rightarrow x$ em R (transições do tipo 2), caso A seja um não-terminal, ou então remove (*desempilha*) o símbolo contido no topo da pilha, desde que tal símbolo seja idêntico ao próximo símbolo da cadeia de entrada (transições de tipo 3), caso A seja um símbolo terminal. As transições de M são projetadas para que a cadeia armazenada na pilha represente sempre uma derivação mais à esquerda da cadeia de entrada, quando da validação de algum não-terminal. Assim, intermitentemente, M realiza passos da derivação na pilha e, entre dois desses passos, remove do topo da pilha quaisquer símbolos terminais aí encontrados e os compara com os símbolos na cadeia de entrada. A remoção dos terminais da pilha tem o efeito de expor os não-terminais mais à esquerda, permitindo, assim, com que o processo continue.

Exemplo 3.4.1: Considere-se a gramática $G = (V, \Sigma, R, S)$ com $V = \{S, a, b, c\}$, $\Sigma = \{a, b, c\}$ e $R = \{S \rightarrow aSa,\ S \rightarrow bSb,\ S \rightarrow c\}$, que gera a linguagem $\{wcw^R: w \in \{a, b\}^*\}$. O correspondente autômato de pilha, de acordo com a construção acima, é $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, com

$$\begin{aligned}\Delta = \{&((p, \varepsilon, \varepsilon), (q, S)), && (T1)\\ &((q, \varepsilon, S), (q, aSa)), && (T2)\\ &((q, \varepsilon, S), (q, bSb)), && (T3)\\ &((q, \varepsilon, S), (q, c)), && (T4)\\ &((q, a, a), (q, \varepsilon)), && (T5)\\ &((q, b, b), (q, \varepsilon)), && (T6)\\ &((q, c, c), (q, \varepsilon))\} && (T7).\end{aligned}$$

A cadeia *abbcbba* é aceita por M através da execução da seguinte seqüência de movimentos:

Estado	Entradas não lidas	Pilha	Transição utilizada
p	*abbcbba*	ε	-
q	*abbcbba*	S	T1
q	*abbcbba*	aSa	T2
q	*bbcbba*	Sa	T5
q	*bbcbba*	$bSba$	T3
q	*bcbba*	Sba	T6
q	*bcbba*	$bSbba$	T3
q	*cbba*	$Sbba$	T6
q	*cbba*	*cbba*	T4
q	*bba*	*bba*	T7
q	*ba*	*ba*	T6
q	*a*	*a*	T6
q	ε	ε	T5

Compare-se com a operação, sobre a mesma cadeia, do autômato de pilha do Exemplo 3.3.1.✧

Para continuar a prova do Lema 3.4.1, com vistas à constatação de que $L(M) = L(G)$, provaremos a seguinte afirmação.

Afirmação. Seja que $w \in \Sigma^*$ e $\alpha \in (V - \Sigma)V^* \cup \{\varepsilon\}$. Nessas condições, $S \overset{L}{\Rightarrow}{}^{+} w\alpha$, se, e somente se, $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$.

Essa afirmação será suficiente para a prova do Lema 3.4.1, uma vez que resultará (adotando-se $\alpha = \varepsilon$) que $S \overset{L}{\Rightarrow}{}^{*} w$ se e somente se $(q, \varepsilon, S) \vdash_M^* (q, \varepsilon, \varepsilon)$ – em outras palavras, $w \in L(G)$ se e somente se $w \in L(M)$.

(*Somente se*) Seja $S \overset{L}{\Rightarrow}{}^{*} w\alpha$, onde $w \in \Sigma^*$ e $\alpha \in (V - \Sigma)V^* \cup \{\varepsilon\}$. Devemos mostrar por indução no comprimento da derivação mais à esquerda de w a partir de S que $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$.

Base de Indução. Se a derivação é de comprimento 0, então, $w = \varepsilon$, $\alpha = S$ e, portanto, obviamente, $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$.

Hipótese de Indução. Se $S \overset{L}{\Rightarrow}{}^{*} w\alpha$ por uma derivação de comprimento menor ou igual a n, com $n \geq 0$, então $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$.

Passo de indução. Seja

$$S = u_0 \overset{L}{\Rightarrow} u_1 \overset{L}{\Rightarrow} \ldots \overset{L}{\Rightarrow} u_n \overset{L}{\Rightarrow} u_{n+1} = w\alpha$$

uma derivação mais à esquerda de $w\alpha$ a partir de S. Seja A o não-terminal mais à esquerda de u_n. Então, $u_n = xA\beta$ e $u_{n+1} = x\gamma\beta$, onde $x \in \Sigma^*$, $\beta, \gamma \in V^*$ e $A \rightarrow \gamma$ é uma regra contida em R. Como há uma derivação mais à esquerda, de comprimento n, de $u_n = xA\beta$ a partir de S, pela hipótese da indução

$$(q, x, S) \vdash_M^* (q, \varepsilon, A\beta). \qquad (2)$$

Como $A \rightarrow \gamma$ é uma regra contida em R,

$$(q, \varepsilon, A\beta) \vdash_M (q, \varepsilon, \gamma\beta), \qquad (3)$$

pela aplicação de uma transição do tipo 2.

* N. de R.T. O posicionamento do símbolo de fechamento transitivo reflexivo está incorreto no original em inglês.

Agora note-se que u_{n+1} é $w\alpha$, mas é também $xy\beta$. Portanto, há uma cadeia $y \in \Sigma^*$, tal que $w=xy$ e $y\alpha = \gamma\beta$. Portanto, é possível reescrever (2) e (3) como

$$(q, w, S) \vdash_M^* (q, y, \gamma\beta). \tag{4}$$

Entretanto, $y\alpha = \gamma\beta$, logo pode-se obter

$$(q, y, \gamma\beta) \vdash_M^* (q, \varepsilon, \alpha), \tag{5}$$

aplicando-se uma seqüência de $|y|$ transições do tipo 3. A combinação de (4) e (5) completa o passo de indução.

(*Se*) Seja agora $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$ com $w \in \Sigma^*$ e $\alpha \in (V - \Sigma)V^* \cup \{\varepsilon\}$: mostraremos que $S \overset{L}{\Rightarrow}^* w\alpha$. Novamente, a prova será feita por indução, mas, desta vez, sobre o *número de transições de tipo 2* na computação por M.

Base de indução. Como o primeiro movimento em qualquer computação é promovido por uma transição do tipo 2, se $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$ sem utilizar transições do tipo 2, então $w = \varepsilon$ e $\alpha = S$, e nosso enunciado é verdadeiro.

Hipótese de indução. Se $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$ para uma computação que tenha, no máximo, n passos do tipo 2, $n \geq 0$, então $S \overset{L}{\Rightarrow}^* w\alpha$.

Passo de indução. Seja $(q, w, S) \vdash_M^* (q, \varepsilon, \alpha)$ em $n + 1$ transições do tipo 2, e explicite-se a transição diferenciada:

$$(q, w, S) \vdash_M^* (q, y, A\beta) \vdash_M (q, y, \gamma\beta) \vdash_M^* (q, \varepsilon, \alpha),$$

onde $w = xy$ para algum x, $y \in \Sigma^*$ e $A \to \gamma$ é uma regra da gramática. Pela hipótese de indução temos que $S \overset{L}{\Rightarrow}^* xA\beta$ e, portanto, $S \overset{L}{\Rightarrow}^* x\gamma\beta$. Considerando que, $(q, y, \gamma\beta) \vdash_M^* (q, \varepsilon, \alpha)$, é o resultado da aplicação exclusiva de transições do tipo 3, segue-se que $y\alpha = \gamma\beta$ e, assim, $S \overset{L}{\Rightarrow}^* xy\alpha = w\alpha$, completando a prova do Lema 3.4.1 e, com ela, metade da prova do Teorema 3.4.1. ■

Voltemo-nos agora para a prova da outra metade do Teorema 3.4.1

Lema 3.4.2: *Se uma linguagem é aceita por um autômato de pilha, então ela é uma linguagem livre de contexto.*

Prova: Será útil impor algumas restrições sobre o nosso autômato de pilha. Vamos definir um autômato de pilha **simples** como sendo aquele para o qual vale a seguinte afirmação:

> Sempre que $((q, a, \beta), (p, \gamma))$ é uma transição do autômato de pilha e q não é o estado inicial, então $\beta \in \Gamma$ e $|\gamma| \leq 2$.

Em outras palavras, a máquina sempre consulta (apenas) o símbolo contido no topo da pilha e o substitui ou por ε, ou por um símbolo do alfabeto de pilha, ou ainda por um par de símbolos do alfabeto de pilha. É fácil constatar que não nos interessa que um autômato de pilha possua somente transições desse tipo, uma vez que, nesse caso, tal autômato não seria capaz de operar nas situações em que a pilha estiver vazia (por exemplo, tal autômato não seria sequer capaz de iniciar uma computação, já que, nessa ocasião a pilha está sempre vazia). Essa é a razão pela qual não imporemos restrições sobre as transições que partem do estado inicial.

Desejamos provar que, se uma linguagem for aceita por um autômato de pilha irrestrito, então ela será aceita por um autômato de pilha simples. Assim, seja $M = (K, \Sigma, \Gamma, \Delta, s, F)$ qualquer autômato de pilha; devemos construir um autômato de pilha simples $M' = (K', \Sigma, \Gamma \cup \{Z\}, \Delta', s', \{f'\})$ que também aceite $L(M)$; aqui s' e f' são novos estados que não pertencem a K, e Z é um novo símbolo de pilha, o símbolo escolhido como indicador do fundo da pilha, também não contido em Γ. Incluamos inicialmente em Δ a transição $((s', \varepsilon, \varepsilon), (s, Z))$; essa transição inicia a computação inserindo o símbolo indicado do fundo da pilha, onde permanecerá por todo o processo. Nenhuma outra regra de Δ irá adicionar o símbolo Z à pilha – exceto para substituí-lo no fundo da mesma. Também incluiremos em Δ as transições $((f, Z, \varepsilon), (f', \varepsilon))$ para cada $f \in F$. Essas transições terminarão a computação removendo Z da pilha e aceitando a cadeia de entrada que acaba de ser processada.

Inicialmente, Δ' consiste das transições inicial e final descritas acima e de todas as transições contidas em Δ. Devemos, em seguida, substituir todas as transições presentes em Δ', que violem a condição de simplicidade do autômato por transições equivalentes que satisfaçam tal condição. Deve-se, primeiramente, substituir as transições com $|\beta| \geq 2$. Em seguida, eliminar transições com $|\gamma| > 2$ sem introduzir transições com $|\beta| \geq 2$. Finalmente, remover transições com $\beta = \varepsilon$ sem introduzir transições com $|\beta| \geq 2$ ou $|\gamma| > 2$.

Considere-se uma transição qualquer da forma $((q, a, \beta), (p, \gamma)) \in \Delta'$, onde $\beta = B_1 \cdots B_n$ com $n > 1$. Ela será substituída por novas transições que removam um a um os símbolos individuais $B_1 \ldots B_n$, em lugar de removê-los todos em uma só operação. Especificamente, adicionaremos a Δ' as transições seguintes:

$$\begin{array}{l}((q, \varepsilon, B_1), (q_{B_1}, \varepsilon)),\\ ((q_{B_1}, \varepsilon, B_2), (q_{B_1 B_2}, \varepsilon)),\\ \vdots\\ ((q_{B_1 B_2 \ldots B_{n-2}}, \varepsilon, B_{n-1}), (q_{B_1 B_2 \cdots B_{n-1}}, \varepsilon)),\\ ((q_{B_1 B_2 \ldots B_{n-1}}, a, B_n), (p, \gamma)),\end{array}$$

onde, para $i = 1, \ldots, n-1$, $q_{B_1 B_2 \cdots B_i}$ é um novo estado com o significado intuitivo de "estado q atingido após os símbolos $B_1 B_2 \cdots B_i$ terem sido removidos". Repetimos isso para todas as transições $((q, a, \beta), (p, \gamma)) \in \Delta'$ com $|\beta| > 1$. É claro que o autômato de pilha resultante é equivalente ao original.

Analogamente, substituímos as transições da forma $((q, a, \beta), (p, \gamma))$ com $\gamma = C_1 \cdots C_m$ e $m \geq 2$ por transições

$$\begin{array}{l}((q, a, \beta), (r_1, C_m)),\\ ((r_1, \varepsilon, \varepsilon), (r_2, C_{m-1})),\\ \vdots\\ ((r_{m-2}, \varepsilon, \varepsilon), (r_{m-1}, C_2)),\\ ((r_{m-1}, \varepsilon, \varepsilon), (p, C_1)),\end{array}$$

onde $r_1, \ldots, r_{m-1}$ são novos estados. Note-se que todas as transições $((q, a, \beta), (p, \gamma)) \in \Delta$ têm agora $|\gamma| \leq 1$ – uma exigência mais restritiva do que

a de simplicidade, a qual realmente traria uma perda de generalidade. Será restaurada a condição $|\gamma| \leq 2$ no próximo estágio da transformação. Nenhuma transição da forma $((q, a, \beta), (p, \gamma))$ com $|\beta| > 1$ foi adicionada.

Por fim, considere-se qualquer transição da forma $((q, a, \varepsilon), (p, \gamma))$ com $q \neq s'$ – as únicas possíveis violações remanescentes da condição de simplicidade. Substitua-se qualquer transição desse tipo por um conjunto de transições da forma $((q, a, A), (p, \gamma A))$, para todos os $A \in \Gamma \cup \{Z\}$. Isto significa que se o autômato pode transitar sem consultar sua pilha, também pode executar uma transição equivalente consultando o símbolo do topo da pilha, seja ele qual for, e substituí-lo imediatamente por ele mesmo. E sabemos que por construção, exceto nas transições inicial e final: existe no mínimo, um símbolo na pilha durante toda a computação a pilha nunca se esvazia. Note-se também que, nesse estágio podemos introduzir γ's de dois comprimentos, os quais não violam a condição de simplicidade, mas são necessários para obter o autômato de pilha geral.

É fácil ver que essa construção resulta em um autômato de pilha simples M', tal que $L(M) = L(M')$. Para continuar a prova do lema, devemos mostrar que pode ser construída uma gramática livre de contexto G, tal que $L(G) = L(M')$; isso concluirá a prova do lema e com ele a do Teorema 3.4.1.

Seja $G = (V, \Sigma, R, S)$, onde V contém, além de um novo símbolo S e os símbolos em Σ, um novo símbolo $\langle q, A, p\rangle$ para todo $q, p \in K'$ e cada $A \in \Gamma \cup \{\varepsilon, Z\}$. Para entender o papel do não-terminal $\langle q, A, p\rangle$, convém lembrar que G *deve* gerar todas as cadeias aceitas por M'. Portanto, os não-terminais de G significam diferentes partes das cadeias de entrada que são aceitas por M'. Em particular, se $A \in \Gamma$, então os não-terminais $\langle q, A, p\rangle$ representam qualquer parte da cadeia de entrada que pode ser lida desde a ocasião em que M' esteja no estado q com A no topo da sua pilha, e aquela em que M' remove essa ocorrência de A da pilha e transita para o estado p. Se $A = \varepsilon$, então $\langle q, \varepsilon, p\rangle$ denota uma parte da cadeia de entrada que pode ser lida desde o momento em que M' esteja no estado q, e outro, em que atinge o estado p com o mesmo conteúdo da pilha, sem que nesse interim, essa parte da pilha seja consultada ou modificada.

As regras em R são de quatro tipos.

(1) A regra $S \to \langle s, Z, f'\rangle$, onde s é o estado inicial do autômato de pilha original M e f', o novo estado final.
(2) Para cada transição $((q, a, B), (r, \varepsilon))$, onde $q, r \in K'$, $a \in \Sigma \cup \{\varepsilon\}$, $B, C \in \Gamma \cup \{\varepsilon\}$ e para cada $p \in K'$, adicionamos a regra $\langle q, B, p\rangle \to a\langle r, C, p\rangle$.
(3) Para cada transição $((q, a, B), (r, C_1 C_2))$, onde $q, r \in K'$, $a \in \Sigma \cup \{\varepsilon\}$, $B, C \in \Gamma \cup \{\varepsilon\}$ e $C_1, C_2 \in \Gamma$ para cada $p, p' \in K'$, acrescentamos a regra $\langle q, B, p\rangle \to a\langle r, C_1, p'\rangle\langle p', C_2, p\rangle$.
(4) Para cada $q \in K'$, a regra $\langle q, \varepsilon, q\rangle \to \varepsilon$.

Note que, como M' é simples, suas transições são apenas do tipo 2 ou do tipo 3.

Uma regra do tipo 1 afirma essencialmente que qualquer cadeia de entrada que possa ser lida por M' leva o autômato do estado s para o estado final, enquanto, ao mesmo tempo, o efeito resultante na pilha é que o símbolo da parte inferior removido é uma cadeia da linguagem $L(M')$. Uma regra do tipo 4 indica que nenhuma computação é necessária para levar o autômato de um estado qualquer para si próprio sem modificar o conteúdo da pilha. Finalmente, uma regra do tipo 2 ou 3 diz que, se $((q, a, B), (p, \gamma)) \in \Delta'$, então uma das possíveis computações que conduz do estado q para o estado p, consumindo B (possivelmente vazio) do topo da pilha, inicia lendo a entrada a, substituindo B por γ, transitando para o estado γ e, então, prosseguindo para consumir γ e acabando no estado p – lendo qualquer entrada que seja apropriada durante tal computação. Se $\gamma = C_1 C_2$, essa última computação pode, a princípio, passar por qualquer estado p' imediatamente após C_1 ser removido (esta é uma regra do tipo 3).

Esses lembretes intuitivos são formalizados na seguinte afirmação.

Afirmação. Para qualquer $q, p \in K', A \in \Gamma \cup \{\varepsilon\}$ e $x \in \Sigma^*$,

$$\langle q, A, p\rangle \Rightarrow_G^* x \text{ se e somente se } (q, x, A) \vdash_M^* (p, \varepsilon, \varepsilon).$$

O Lema 3.4.2, e com ele o Teorema 3.4.1, são decorrentes imediatos da afirmação, uma vez que $\langle s, \varepsilon, f\rangle \Rightarrow_G^* x$ para algum $f \in F$ se e somente se $(s, x, \varepsilon) \vdash_M^* (f, \varepsilon, \varepsilon)$; isto é, $x \in L(G)$ se e somente se $x \in L(M')$.

Ambas as direções da afirmação podem ser provadas por indução sobre o comprimento da derivação de G ou da computação de M'; Essas demonstrações são deixadas como exercício para o leitor (Problema 3.4.5). ■

Problemas para a Seção 3.4

3.4.1 Execute a construção apresentada no Lema 3.4.1 para a gramática do Exemplo 3.1.4. Determine a operação do autômato que você construiu aplicando-o a cadeia de entrada (()()).

3.4.2 Execute a construção apresentada no Lema 3.4.2 para o autômato de pilha do Exemplo 3.3.2, sendo G a gramática resultante, qual é o conjunto $\{w \in \{a, b\}^*: \langle q, a, p\rangle \Rightarrow_G^* w\}$? Compare com a prova do Lema 3.4.2.

3.4.3 Execute a construção apresentada no Lema 3.4.2 para o autômato de pilha do Exemplo 3.3.3. A gramática resultante terá 25 regras, muitas das quais podem ser eliminadas por serem inúteis. Mostre uma derivação da cadeia *aababbba* nessa gramática. (Você pode alterar os nomes dos não-terminais para aumentar a clareza.)

3.4.4 Mostre que, se $M = (K, \Sigma, \Gamma, \Delta, s, F)$ é um autômato de pilha, então há outro autômato de pilha $M' = (K', \Sigma, \Gamma, \Delta', s, F)$, tal que $L(M') = L(M)$ e, *para todos os* $((q, u, \beta), (p, \gamma)) \in \Delta'$, $|\beta| + |\gamma| \leq 1$.

3.4.5 Complete a prova do Lema 3.4.2.

3.4.6 Uma gramática livre de contexto é dita **linear** se e somente se nenhuma regra tiver como lado direito uma cadeia com mais de um não-terminal. Diz-se que um autômato de pilha $(K, \Sigma, \Gamma, \Delta, s, F)$ é **de turno único** se e somente se $(s, w, \varepsilon) \vdash^* (q_1, w_1, \gamma_1) \vdash (q_2, w_2, \gamma_2) \vdash^* (q_3, w_3, \gamma_3)$ e $|\gamma_2| < |\gamma_1|$, então $|\gamma_3| \leq |\gamma_2|$ –. (Isto é, uma vez que a pilha começa a diminuir em altura, nunca aumentará novamente). Mostre que uma linguagem é gerada por uma gramática livre de contexto linear se, e somente se, ela é aceita por um autômato de pilha de turno único.

3.5 LINGUAGENS QUE SÃO LIVRES DE CONTEXTO E LINGUAGENS QUE NÃO SÃO LIVRES DE CONTEXTO

Propriedades de fechamento

Na última seção, mostrou-se a equivalência de duas visualizações das linguagens livres de contexto: como linguagens geradas por gramáticas livre de contexto e como linguagens aceitas por autômatos de pilha. Essas caracterizações enriquecem nosso entendimento das linguagens livres de contexto, uma vez que validam dois diferentes métodos para reconhecê-las como tal. Por exemplo, a representação gramatical é mais natural e atraente para descrever um fragmento de linguagem de programação, tal como no Exemplo 3.1.3; mas a representação em termos de autômato de pilha é mais fácil de visualizar no caso de $\{w \in \{a, b\}^* : w$ tem um mesmo número de a's e b's$\}$ (veja Exemplo 3.3.3). Nesta subseção, devemos apresentar ferramentas adicionais para caracterizar as linguagens livres de contexto: mostraremos algumas *propriedades de fechamento* das linguagens livres de contexto sob as operações usuais de linguagem, como foi feito no caso das propriedades de fechamento das linguagens regulares. Na próxima subseção, devemos provar um *teorema de bombeamento* mais poderoso, o qual nos permite mostrar que algumas linguagens *não* são livres de contexto.

Teorema 3.5.1: *As linguagens livres de contexto são fechadas em relação às operações de união, concatenação e estrela de Kleene.*

Prova: Sejam $G_1 = (V_1, \Sigma_1, R_1, S_1)$ e $G_2 = (V_2, \Sigma_2 R_2, S_2)$ duas gramáticas livres de contexto. Sem perda de generalidade, vamos admitir que elas têm conjuntos disjuntos de não-terminais – isto é, $V_1 - \Sigma_1$ e $V_2 - \Sigma_2$, são disjuntos.

União. Seja S um novo símbolo e $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$, onde $R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$. Gostaríamos de provar que $L(G) = L(G_1) \cup L(G_2)$. Como as únicas regras que envolvem S são $S \to S_1$ e $S \to S_2$, então $S \Rightarrow_G^* w$ se e somente se $S_1 \Rightarrow_{G_1}^* w$ ou $S_2 \Rightarrow_{G_2}^* w$. Uma vez que G_1 e G_2 têm conjuntos disjuntos de não-terminais, a última disjunção é equivalente a dizer que $w \in L(G_1) \cup L(G_2)$.

Concatenação. A construção é similar: $L(G_1)\,L(G_2)$ é gerada pela gramática

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S).$$

Estrela de Kleene. $L(G_1)^*$ é gerada por

$$G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \to \varepsilon, S \to SS_1\}, S).$$

■

Como devemos ver brevemente, a classe de linguagens livres de contexto *não* é fechada em relação às operações de intersecção ou complementação. Isso não é muito surpreendente: lembre-se de que nossa prova de que linguagens regulares são fechadas em relação à operação de intersecção depende do fechamento sob complementação e de que essa construção exigia que o autômato fosse *determinístico*. E nem todas as linguagens livres de contexto são aceitas por autômatos de pilha determinísticos (veja o corolário do Teorema 3.7.1).

Há uma interessante prova direta do fechamento de linguagens regulares sob intersecção, que não se apóia no fechamento sob complementação, mas em uma construção direta de um autômato finito cujo conjunto de estados é o *produto cartesiano* dos conjuntos de estados dos autômatos finitos envolvidos (lembre-se do Problema 2.3.3). Essa construção não pode, naturalmente, ser estendida para o autômato de pilha – o autômato produto teria necessariamente *duas pilhas*. Entretanto, pode-se aplicar uma técnica *quando um dos dois autômatos é finito*:

Teorema 3.5.2: *A intersecção de uma linguagem livre de contexto com uma linguagem regular é uma linguagem livre de contexto.*

Prova: Se L é uma linguagem livre de contexto e R é uma linguagem regular, então $L = L(M_1)$ para algum autômato de pilha $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$, e $R = L(M_2)$ para algum autômato finito determinístico $M_2 = (K_2, \Sigma, \delta, s_2, F_2)$. A idéia é combinar essas máquinas para formar um autômato de pilha simples M que realiza em paralelo as computações de M_1 e M_2, e que aceita a cadeia somente se esta for aceita tanto por M_1 como por M_2. Seja: $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde

$K = K_1 \times K_2$, é o produto cartesiano dos conjuntos de estados de M_1 e M_2;
$\Gamma = \Gamma_1$;
$s = (s_1, s_2)$;
$F = F_1 \times F_2$, e

Δ é a relação de transição, definida como segue. Para cada transição do autômato de pilha $((q_1, a, \beta), (p_1, \gamma) \in \Delta_1$ e para cada estado $q_2 \in K_2$ inclui-mos em Δ a transição $(((q_1, q_2), a, \beta), ((p_1, \delta(q_2, a)), \gamma))$; e para cada transição da forma $((q_1, \varepsilon, \beta), (p_1, \gamma) \in \Delta_1$ e cada estado $q_2 \in K_2$, incluímos a transição $(((q_1, q_2), \varepsilon, \beta), ((p_1, q_2), \gamma))$. Em outra palavras, M passa do estado (q_1, q_2) para o estado (p_1, p_2) nas mesmas condições em que M_1 passa do estado q_1 para p_1, porém, adicionalmente M acompanha as correspondentes alterações no estado de M_2, causadas pela leitura da mesma entrada.

É fácil ver que, de fato, $w \in L(M)$ se e somente se $w \in L(M_1) \cap L(M_2)$. ■

Exemplo 3.5.1: Suponha que a linguagem L compreende todas as cadeias de a's e b's com igual número de a's e b's, mas não contendo as subcadeias *abaa* ou *babb*. Então L é livre de contexto, uma vez que é a intersecção da linguagem aceita pelo autômato de pilha do Exemplo 3.3.3 com a linguagem regular $\{a, b\}^* - \{a, b\}^*(abaa \cup babb)\{a, b\}^*$. ✧

Um teorema de bombeamento

Infinitas linguagens livres de contexto apresentam um tipo de periodicidade mais sutil que a existente nas linguagens regulares. Para entender

3.5 LINGUAGENS QUE SÃO LIVRES DE CONTEXTO E LINGUAGENS QUE NÃO SÃO LIVRES DE CONTEXTO

Propriedades de fechamento

Na última seção, mostrou-se a equivalência de duas visualizações das linguagens livres de contexto: como linguagens geradas por gramáticas livre de contexto e como linguagens aceitas por autômatos de pilha. Essas caracterizações enriquecem nosso entendimento das linguagens livres de contexto, uma vez que validam dois diferentes métodos para reconhecê-las como tal. Por exemplo, a representação gramatical é mais natural e atraente para descrever um fragmento de linguagem de programação, tal como no Exemplo 3.1.3; mas a representação em termos de autômato de pilha é mais fácil de visualizar no caso de $\{w \in \{a, b\}^*: w$ tem um mesmo número de a's e b's$\}$ (veja Exemplo 3.3.3). Nesta subseção, devemos apresentar ferramentas adicionais para caracterizar as linguagens livres de contexto: mostraremos algumas *propriedades de fechamento* das linguagens livres de contexto sob as operações usuais de linguagem, como foi feito no caso das propriedades de fechamento das linguagens regulares. Na próxima subseção, devemos provar um *teorema de bombeamento* mais poderoso, o qual nos permite mostrar que algumas linguagens *não* são livres de contexto.

Teorema 3.5.1: *As linguagens livres de contexto são fechadas em relação às operações de união, concatenação e estrela de Kleene.*

Prova: Sejam $G_1 = (V_1, \Sigma_1, R_1, S_1)$ e $G_2 = (V_2, \Sigma_2 R_2, S_2)$ duas gramáticas livres de contexto. Sem perda de generalidade, vamos admitir que elas têm conjuntos disjuntos de não-terminais – isto é, $V_1 - \Sigma_1$ e $V_2 - \Sigma_2$, são disjuntos.

União. Seja S um novo símbolo e $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$, onde $R = R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}$. Gostaríamos de provar que $L(G) = L(G_1) \cup L(G_2)$. Como as únicas regras que envolvem S são $S \to S_1$ e $S \to S_2$, então $S \Rightarrow_G^* w$ se e somente se $S_1 \Rightarrow_G^* w$ ou $S_2 \Rightarrow_G^* w$. Uma vez que G_1 e G_2 têm conjuntos disjuntos de não-terminais, a última disjunção é equivalente a dizer que $w \in L(G_1) \cup L(G_2)$.

Concatenação. A construção é similar: $L(G_1)\,L(G_2)$ é gerada pela gramática

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1 S_2\}, S).$$

Estrela de Kleene. $L(G_1)^*$ é gerada por

$$G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \to \varepsilon, S \to SS_1\}, S).$$

■

Como devemos ver brevemente, a classe de linguagens livres de contexto *não* é fechada em relação às operações de intersecção ou complementação. Isso não é muito surpreendente: lembre-se de que nossa prova de que linguagens regulares são fechadas em relação à operação de intersecção depende do fechamento sob complementação e de que essa construção exigia que o autômato fosse *determinístico*. E nem todas as linguagens livres de contexto são aceitas por autômatos de pilha determinísticos (veja o corolário do Teorema 3.7.1).

Há uma interessante prova direta do fechamento de linguagens regulares sob intersecção, que não se apóia no fechamento sob complementação, mas em uma construção direta de um autômato finito cujo conjunto de estados é o *produto cartesiano* dos conjuntos de estados dos autômatos finitos envolvidos (lembre-se do Problema 2.3.3). Essa construção não pode, naturalmente, ser estendida para o autômato de pilha – o autômato produto teria necessariamente *duas pilhas*. Entretanto, pode-se aplicar uma técnica *quando um dos dois autômatos é finito*:

Teorema 3.5.2: *A intersecção de uma linguagem livre de contexto com uma linguagem regular é uma linguagem livre de contexto.*

Prova: Se L é uma linguagem livre de contexto e R é uma linguagem regular, então $L = L(M_1)$ para algum autômato de pilha $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$, e $R = L(M_2)$ para algum autômato finito determinístico $M_2 = (K_2, \Sigma, \delta, s_2, F_2)$. A idéia é combinar essas máquinas para formar um autômato de pilha simples M que realiza em paralelo as computações de M_1 e M_2, e que aceita a cadeia somente se esta for aceita tanto por M_1 como por M_2. Seja: $M = (K, \Sigma, \Gamma, \Delta, s, F)$, onde

$K = K_1 \times K_2$, é o produto cartesiano dos conjuntos de estados de M_1 e M_2;
$\Gamma = \Gamma_1$;
$s = (s_1, s_2)$;
$F = F_1 \times F_2$, e

Δ é a relação de transição, definida como segue. Para cada transição do autômato de pilha $((q_1, a, \beta), (p_1, \gamma) \in \Delta_1$ e para cada estado $q_2 \in K_2$ incluí-mos em Δ a transição $(((q_1, q_2), a, \beta), ((p_1, \delta(q_2, a)), \gamma))$; e para cada transição da forma $((q_1, \varepsilon, \beta), (p_1, \gamma) \in \Delta_1$ e cada estado $q_2 \in K_2$, incluímos a transição $(((q_1, q_2), \varepsilon, \beta), ((p_1, q_2), \gamma))$. Em outra palavras, M passa do estado (q_1, q_2) para o estado (p_1, p_2) nas mesmas condições em que M_1 passa do estado q_1 para p_1, porém, adicionalmente M acompanha as correspondentes alterações no estado de M_2, causadas pela leitura da mesma entrada.

É fácil ver que, de fato, $w \in L(M)$ se e somente se $w \in L(M_1) \cap L(M_2)$. ■

Exemplo 3.5.1: Suponha que a linguagem L compreende todas as cadeias de a's e b's com igual número de a's e b's, mas não contendo as subcadeias *abaa* ou *babb*. Então L é livre de contexto, uma vez que é a intersecção da linguagem aceita pelo autômato de pilha do Exemplo 3.3.3 com a linguagem regular $\{a, b\}^* - \{a, b\}^*(abaa \cup babb)\{a, b\}^*$. ✧

Um teorema de bombeamento

Infinitas linguagens livres de contexto apresentam um tipo de periodicidade mais sutil que a existente nas linguagens regulares. Para entender

esse aspecto característico das linguagens livres de contexto começamos analisando um fenômeno quantitativo familiar observado nas árvores de análise sintática. Seja $G = (V, \Sigma, R, S)$ uma gramática livre de contexto. O **leque** (fanout) de G, denotado $\phi(G)$, é o número de símbolos no lado direito da mais longa regra do conjunto R. Um **caminho** em uma árvore de análise sintática é uma seqüência de vértices distintos, cada um conectado ao anterior por um arco: o primeiro vértice de um caminho denomina-se raiz, e o último vértice, folha. O **comprimento** do caminho é o número de arcos que o formam. A **altura** de uma árvore de análise sintática é o comprimento do mais longo dos caminhos de que esta é formada.

Lema 3.5.1: *O contorno*† *de qualquer árvore de análise sintática de G de altura h tem comprimento máximo* $\phi(G)^h$.

Prova: A prova é por indução em h. Quando $h = 1$, então a árvore de análise sintática é uma regra da gramática (este é o caso 2 da definição da árvore de análise sintática) e, portanto, seu contorno possui, no máximo, $\phi(G)^h = \phi(G)$ símbolos.

Suponhamos que o enunciado se aplique a árvores de análise sintática de altura $h \geq 1$. Para o passo de indução, qualquer árvore de análise sintática de altura $h + 1$ consistirá de uma raiz conectada a, no máximo, $\phi(G)$ árvores menores cada qual com altura máxima, h (esse é caso 3 da definição da árvore de análise sintática). Por indução, todas essas "subárvores" de análise sintática têm contornos de comprimento máximo $\phi(G)^h$ cada. Segue-se que o contorno global da árvore de análise sintática é, de fato, $\phi(G)^{h+1}$, completando a indução. ■

Em outras palavras a árvore de análise sintática de qualquer cadeia $w \in L(G)$ com $|w| > \phi(G)^h$, deve apresentar ao menos um caminho de comprimento maior que h. Isso é crucial na prova do seguinte *teorema de bombeamento* para linguagens livres de contexto.

Teorema 3.5.3: *Seja* $G = (V, \Sigma, R, S)$ *uma gramática livre de contexto. Então qualquer cadeia* $w \in L(G)$, *de comprimento maior que* $\phi(G)^{|V-\Sigma|}$ *pode ser reescrita como* $w = uvxyz$, *com* v *ou* y *não-vazio, e nessas condições,* uv^nxy^nz *pertence a* $L(G)$ *para todo* $n \geq 0$.

Prova: Seja w tal cadeia e T aquela árvore de análise sintática, com raiz S e com contorno w, *que apresenta o menor número de folhas* dentre todas as árvores de análise sintática com a mesma raiz e contorno. Como o contorno de T é maior que $\phi(G)^{|V-\Sigma|}$, segue-se que T inclui algum caminho de comprimento maior ou igual a, $|V - \Sigma| + 1$, ou seja, T possui no mínimo $|V - \Sigma| + 2$ vértices. Somente um desses vértices pode ser rotulado com um terminal e, portanto, os demais são rotulados como não-terminais. Como há mais vértices no caminho do que não-terminais, então haverá dois vértices desse

† N. de R. – Como "contorno" de uma árvore de análise sintática estamos designando a cadeia de símbolos cuja análise sintática produz a árvore em questão. Consiste da seqüência de terminais representados pelas folhas da árvore sintática, lidos seqüencialmente da esquerda para a direita.

caminho rotulados com um mesmo membro A de $V - \Sigma$.† Examinemos tal caminho mais detalhadamente (veja a Figura 3-9):

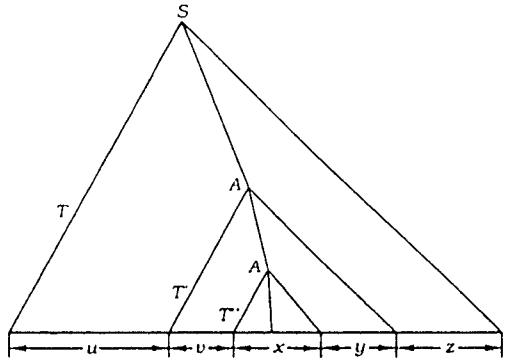


Figura 3-9

Vamos chamar u, v, x, y e z de partes do contorno de T, conforme mostrado na figura. Assim, x é o contorno da subárvore T'' cuja raiz é o mais baixo vértice rotulado com A; v é a parte do contorno da árvore T' originados no vértice A superior, que antecede o contorno de T''; u é o contorno de T onde o contorno de T' inicia; e z é o restante do contorno de T.

Agora está claro que a parte de T, excluindo T'', pode ser repetida qualquer número de vezes, *inclusive nenhuma*, para produzir outras árvores de análise sintática de G cujo contorno sejam as cadeias de forma uv^nxy^nz, $n \geq 0$. Isso completa a prova, exceto quanto à imposição de que $vy \neq \varepsilon$. Mas, se $vy = \varepsilon$, então haverá uma árvore com raiz S e contorno w com menos folhas do que T – exatamente aquela que resulta da omissão, em T, da parte de T que exclui T'' – contrariando nossa hipótese de que T é a menor árvore desse tipo. ■

Exemplo 3.5.2: Exatamente da mesma forma que o teorema de bombeamento para linguagens regulares (Teorema 2.4.1), esse teorema é útil para mostrar que certas linguagens não são livres de contexto. Por exemplo, vamos mostrar que $L = \{a^nb^nc^n: n \geq 0\}$ não é livre do contexto. Para tanto, vamos admitir que seja $L = L(G)$ para alguma gramática livre de contexto $G = (V, \Sigma, R, S)$. Suponhamos que $n > \frac{\phi(G)^{|V-\Sigma|}}{3}$. Então $w = a^nb^nc^n$ pertence a $L(G)$ e tem, portanto, uma representação da forma $w = uvxyz$, tal que v ou y é não-vazio, e uv^ixy^iz pertence a $L(G)$ para $n = 0, 1, 2, \ldots$. Há dois casos, ambos levando a uma contradição. Se vy contém ocorrências de todos os três símbolos a, b, c, então, no mínimo, um deles deve conter ocorrências de pelo menos dois desses símbolos. Mas uv^2xy^2z conterá, nesse caso, duas ocorrências fora de sua ordem correta – um b antes de um a ou um c antes de um a ou de um b. Se vy contiver ocorrências de alguns mas não todos os três símbolos, então uv^2xy^2z terá diferentes números de a's, b's e c's. ✧

Exemplo 3.5.3: A linguagem $L = \{a^n: n \geq 1$ é um número primo$\}$ não é livre de contexto. Para constatar essa afirmação tomemos um número primo p maior

† N. de R.T. – Na verdade pode haver mais de dois vértices com o mesmo rótulo.

que $\phi(G)^{|V-\Sigma|}$, onde $G = (V, \Sigma, R, S)$ seria uma gramática livre de contexto que supostamente gera L. Então a cadeia $w = a^p$ pode ser decomposta, conforme o enunciado do teorema, na forma, $w = uvxyz$, onde todos os componentes de w são cadeias de a's, e $vy \neq \varepsilon$. Consideremos que $vy = a^q$ e $uxz = a^r$, onde q e r são números naturais e $q > 0$. Então o teorema afirma que $r + nq$ é um número primo, para todo o $n \geq 0$, o que foi demonstrado como sendo um absurdo no Exemplo 2.4.3.

Não foi acidental que, em nossa prova de que $\{a^n: n \geq 1$ é um número primo$\}$ não é livre de contexto, tivéssemos recorrido a um argumento muito similar ao do Exemplo 2.4.3, que mostrava que a mesma linguagem não é regular. Revela-se que *qualquer* linguagem livre de contexto sobre um alfabeto unitário é regular; portanto, o resultado do presente exemplo segue-se imediatamente desse fato e do Exemplo 2.4.3. ✧

Exemplo 3.5.4: A seguir, mostraremos que a linguagem $L = \{w \in \{a, b, c\}^*: w$ tem o mesmo número de a's, b's e c's$\}$ não é livre de contexto. Desta vez, empregamos o resultado do Teorema 3.5.2: se L fosse livre de contexto, então sua intersecção com o conjunto regular $a^*b^*c^*$ também o seria. Mas foi demonstrado que essa linguagem $\{a^nb^nc^n: n \geq 0\}$ não é livre de contexto, no Exemplo 3.5.2 acima. ✧

Esses fatos negativos também expõem a carência de algumas propriedades de fechamento apresentada pela classe das linguagens livres de contexto:

Teorema 3.5.4: *As linguagens livres de contexto não gozam de propriedade de fechamento em relação às operações de intersecção e complementação.*

Prova: Evidentemente $\{a^nb^nc^m: m, n \geq 0\}$ e $\{a^mb^nc^n: m, n \geq 0\}$ são linguagens livres de contexto. A intersecção dessas duas linguagens livres de contexto é a linguagem $\{a^nb^nc^n: n \geq 0\}$, que acabamos de provar não ser uma linguagem livre de contexto. Como,

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

então, se as linguagens livres de contexto fossem fechadas em relação à operação de complementação, também seriam fechadas em relação à operação de intersecção (sabemos que elas são fechadas em relação à operação da união, conforme o Teorema 3.5.1). ■

Problemas para a Seção 3.5

3.5.1 Utilizar a propriedade de fechamento em relação à operação da união para mostrar que as seguintes linguagens são livres de contexto:
(a) $\{a^mb^n: m \neq n\}$
(b) $\{a, b\}^* - \{a^nb^n: n \geq 0\}$
(c) $\{a^mb^nc^pd^q: n = q$ ou $m \leq p$ ou $m + n = p + q\}$
(d) $\{a, b\}^* - L$, onde L é a linguagem
$\{babaabaaab \ldots ba^{n-1}ba^nb: n \geq 1\}$
(e) $\{w \in \{a, b\}^*: w = w^R\}$

3.5.2 Utilizar os Teoremas 3.5.2 e 3.5.3 para mostrar que as seguintes linguagens não são livres de contexto:
(a) $\{a^p: p$ é um número primo$\}$
(b) $\{a^{n^2}: n \geq 0\}$
(c) $\{www: w \in \{a, b\}^*\}$
(d) $\{w \in \{a, b, c\}^*: w$ tem o mesmo número de a's, b's e c's$\}$

3.5.3 Um **homomorfismo** é uma função h de cadeias para cadeias tal que para quaisquer duas cadeias v e w, $h(vw) = h(v)h(w)$. Portanto, um homomorfismo é determinado pelos valores que produz quando aplicado a símbolos simples: se $w = a_1 \ldots a_n$ onde cada a_i é um símbolo isolado, então $h(w) = h(a_1) \cdots h(a_n)$. Note-se que um homomorfismo pode "apagar" símbolos: $h(w)$ pode ser ε, mesmo que w não o seja. Mostre que, se L é uma linguagem livre de contexto e h é um homomorfismo, então
(a) $h(L)$ *é uma linguagem livre de contexto*;
(b) $h^{-1}(L)$ (ou seja, $\{w \in \Sigma^*: h(w) \in L\}$) é uma linguagem livre de contexto. (*Sugestão*: Comece com um autômato de pilha M que aceita L. Construa outro autômato de pilha, similar a M, exceto pelo fato de que ele lê sua entrada não da fita de entrada, mas de uma memória finita que é ocasionalmente reabastecida de alguma forma. Complete as idéias e os detalhes formais restantes.)

3.5.4 Na prova do Teorema 3.5.2, por que razão foi feita a hipótese de que M_2 era determinístico?

3.5.5 Mostre que a linguagem $L = \{babaabaaab \cdots ba^{n-1}ba^nb: n \geq 1\}$ não é livre de contexto
(a) aplicando o Teorema de bombeamento (3.5.3);
(b) aplicando o resultado do Problema 3.5.3. (*Sugestão*: O que significa $h(L)$, onde $h(a) = aa$ e $h(b) = a$?)

3.5.6 Se $L_1, L_2 \subseteq \Sigma^*$ são linguagens, o **quociente direito de** L_1 **por** L_2 é definido como segue.

$$L_1/L_2 = \{w \in \Sigma^*: \text{existe algum } u \in L_2, \text{ tal que } wu \in L_1\}$$

(a) Mostre que, se L é livre de contexto e R é regular, então L/R é livre de contexto.
(b) Prove que $\{a^pb^n: p$ é um número primo, e $n > p\}$ não é livre de contexto.

3.5.7 Prove a seguinte versão mais poderosa do Teorema de bombeamento 3.5.3: Seja G uma gramática livre de contexto. Então existem números K e k, tais que qualquer cadeia $w \in L(G)$ com $w \geq K$ pode ser reescrita como $w = uvxyz$ com $vxy \leq k$, de tal modo que ou v ou y não seja vazio, e $uv^nxy^nz \in L(G)$ para cada $n \geq 0$.

3.5.8 Utilize o Problema 3.5.7 para mostrar que a linguagem $\{ww: w \in \{a, b\}^*\}$ não é livre de contexto.

3.5.9 Seja $G = (V, \Sigma, R, S)$ uma gramática livre de contexto. Um não-terminal A de G é dito auto-recursivo central (*self-embedding*) se e somente se $A \Rightarrow_G^+ uAv$ para algum $u, v \in V^*$.

(a) Construa um algoritmo para testar se um dado não-terminal de uma gramática livre de contexto é auto-recursivo central.
(b) Mostre que se G não apresenta não-terminais auto-recursivos centrais, então $L(G)$ é uma linguagem regular.

3.5.10 Dizemos que a gramática livre de contexto $G = (V, \Sigma, R, S)$ está na **forma normal de Greibach** se cada uma de suas regras estiver na forma $A \rightarrow w$ com $w \in \Sigma(V - \Sigma)^*$.
(a) Mostre que para cada gramática livre de contexto G, há uma gramática livre de contexto G' na forma normal de Greibach, tal que $L(G') = L(G) - \{\varepsilon\}$.
(b) Mostre que, se M for construído como na prova do Lema 3.4.1, a partir de uma gramática expressa na forma normal de Greibach, então o número de passos executados por M em qualquer computação sobre uma entrada w, pode ser limitado por alguma função do comprimento de w.

3.5.11 Transdutores determinísticos de estado finito foram introduzidos no Problema 2.1.4. Mostre que se L é livre de contexto e f é computada por um transdutor de estados finitos determinístico, então
(a) $f[L]$ é livre de contexto;
(b) $f^{-1}[L]$ é livre de contexto.

3.5.12 Desenvolva uma versão do Teorema de bombeamento para linguagens livres de contexto, na qual o comprimento da "parte bombeada" é o mais longo possível.

3.5.13 Sejam M_1 e M_2 autômatos de pilha. Mostre como construir autômatos de pilha que aceitem $L(M_1) \cup L(M_2)$, $L(M_1)\,L(M_2)$ e $L(M_1)^*$, oferecendo, assim, outra prova do Teorema 3.5.1.

3.5.14 Qual ou quais das seguintes linguagens são livres de contexto? Justifique brevemente cada caso.
(a) $\{a^m b^n c^p: m = n \text{ ou } n = p \text{ ou } m = p\}$
(b) $\{a^m b^n c^p: m \neq n \text{ ou } n \neq p \text{ ou } m \neq p\}$
(c) $\{a^m b^n c^p: m = n \text{ e } n = p \text{ e } m = p\}$
(d) $\{w \in \{a, b, c\}^*: w \text{ não contém o mesmo número de símbolos } a, b \text{ e } c\}$
(e) $\{w \in \{a, b\}^* : w = w_1 w_2 \cdots w_m \text{ para algum } m \geq 2, \text{ tal que } |w_1| = |w_2| = \ldots = |w_m| \geq 2\}$

3.5.15 Sejam duas linguagens: L livre de contexto e R regular. $L - R$ é necessariamente livre de contexto? E quanto a $R - L$? Justifique suas respostas.

3.6 ALGORITMOS PARA GRAMÁTICAS LIVRES DE CONTEXTO

Nesta seção, estudaremos os problemas computacionais relacionados com as linguagens livres de contexto, desenvolveremos algoritmos para resolver esses problemas e analisaremos sua complexidade. Ao final, estabeleceremos o seguinte resultado:

Teorema 3.6.1: *(a) Existe um algoritmo polinomial que, dada uma gramática livre de contexto, constrói um autômato de pilha equivalente.*
(b) Existe um algoritmo polinomial que, dado um autômato de pilha, constrói uma gramática livre de contexto equivalente.
(c) Existe um algoritmo polinomial que, dada uma gramática livre de contexto G e uma cadeia x, decide se $x \in L(G)$ ou não.

É instrutivo comparar o Teorema 3.6.1 com o seu correspondente Teorema 2.6.1, relativo aos aspectos algoritmicos dos autômatos finitos. Identificamos inicialmente algumas similaridades: em ambos os casos há algoritmos que transformam reconhecedores em geradores e vice-versa, no caso, autômatos finitos em expressões regulares e expressões regulares em autômatos finitos; agora autômatos de pilha em gramáticas livres de contexto e vice-versa. As diferenças, porém são mais evidentes. Primeiramente, no Teorema 2.6.1 não houve necessidade de uma parte análoga ao item (c) acima, uma vez que linguagens regulares são representadas por autômatos finitos determinísticos, que implementam um algoritmo eficiente para decidir precisamente a pertinência de uma cadeia à linguagem. Em contraste, para linguagens livres de contexto, introduzimos até aqui somente reconhecedores não-determinísticos, os autômatos de pilha. Mostraremos na próxima subseção, em resposta ao item (c) acima que para qualquer linguagem livre de contexto podemos construir um reconhecedor determinístico; a construção não é direta e é relativamente sofisticada, e o algoritmo resultante, embora polinomial, não é mais linear com relação ao comprimento da entrada.

Uma segunda diferença importante entre o Teorema 2.6.1 e o Teorema 3.6.1 é que neste último caso seguer mencionamos qualquer algoritmo para testar se duas gramáticas livres de contexto quaisquer (ou dois autômatos de pilha) são equivalentes; nem conjecturamos que possa haver algoritmos para minimizar o número dos estados de um autômato de pilha. Deveremos constatar, no Capítulo 5, que tais questões sobre gramáticas livre de contexto e autômatos de pilha não apresentam nenhuma solução algorítmica – ainda que ineficiente!

O algoritmo de programação dinâmica

Voltamo-nos agora para a prova da parte (c) do Teorema (as partes (a) e (b) são conseqüências simples e diretas das construções utilizadas nas provas dos Lemas 3.4.1 e 3.4.2). Nosso algoritmo para decidir sobre linguagens livres de contexto será baseado em um modo útil de "padronização" de gramáticas livres de contexto.

Definição 3.6.1: Dizemos que uma gramática livre de contexto $G = (V, \Sigma, R, S)$ está na forma normal de Chomsky se $R \subseteq (V - \Sigma) \times V^2$.

Em outras palavras, o lado direito de qualquer regra de uma gramática livre de contexto, dada na forma normal de Chomsky, deve ter comprimento dois. Note-se que nenhuma gramática na forma normal de Chomsky seria capaz de produzir cadeias de comprimento menor que dois, como a, b ou ε; por-

tanto, as linguagens livres de contexto que contêm essas cadeias não podem ser geradas por gramáticas na forma normal de Chomsky. Entretanto, o próximo teorema estabelece que essa é a única perda de generalidade introduzida pela forma normal de Chomsky:

Teorema 3.6.2: *Para qualquer gramática livre de contexto G, existe uma gramática livre de contexto G' na forma normal de Chomsky tal que* $L(G') = L(G) - (\Sigma \cup \{\varepsilon\})$. *Além disso, a construção de G' pode ser realizada em tempo polinomial em relação ao tamanho de G.*

Em outras palavras, G' gera exatamente as cadeias que G gera, exceto as cadeias de comprimento menor que dois: Estando na forma normal de Chomsky, sabemos que G' não pode gerar tais cadeias.

Prova: Devemos mostrar como transformar qualquer gramática livre de contexto $G = (V, \Sigma, R, S)$ em uma gramática livre de contexto na forma normal de Chomsky. Há três maneiras pelas quais o lado direito de uma regra $A \to x$ de G pode violar as limitações da forma normal de Chomsky: *regras longas* (aquelas cujo lado direito tem comprimento três ou mais), *regras* ε (na forma $A \to \varepsilon$) e *regras curtas* (na forma $A \to a$ ou $A \to B$). Devemos mostrar como remover essas violações uma por uma.

Primeiramente consideremos as regras longas de G. Seja $A \to B_1 B_2 \cdots B_n \in R$, onde $B_1, \ldots, B_n \in V$, e $n \geq 3$. Substituímos essa regra por $n - 1$ novas regras, a saber:

$$A \to B_1 A_1,$$
$$A_1 \to B_2 A_2,$$
$$\vdots$$
$$A_{n-2} \to B_{n-1} B_n,$$

onde $A_1, \ldots, A_{n-2}$ são novos não-terminais não utilizados em qualquer outro lugar na gramática. Como a regra $A \to B_1 B_2 \ldots B_n$ pode ser simulada pelas regras acima introduzidas, e como esse é o único modo pelo qual tais novas regras podem ser utilizadas, fica evidente que a gramática livre de contexto resultante é equivalente à original. Repetimos isso para cada regra longa da gramática. A gramática resultante é equivalente à original e tem regras cujos lados direitos têm comprimento dois ou menos.

Exemplo 3.6.1: Tomemos a gramática que gera o conjunto de parênteses balanceados, com regras $S \to SS$, $S \to (S)$, $S \to \varepsilon$. Há somente uma regra longa, $S \to (S)$. Ela é substituída pelas duas regras $S \to (S_1$ e $S_1 \to S)$.✧

Devemos, em seguida, tratar das regras ε. Para esse fim, primeiro determinamos *o conjunto* N_ε *de não-terminais apagáveis*

$$N_\varepsilon = \{A \in V - \Sigma : A \Rightarrow^* \varepsilon\},$$

isto é, o conjunto de todos os não-terminais que podem derivar a cadeia vazia. Isso pode ser feito através de um simples cálculo de fechamento:

$N_\varepsilon := \emptyset$
enquanto existir uma regra $A \to \alpha$ com $\alpha \in N_\varepsilon^*$ e
  $A \notin N_\varepsilon$ acrescentar A **a** N_ε.

Uma vez construído o conjunto N_ε, excluímos de G todos as regras ε e repetimos o seguinte: para cada regra na forma $A \to BC$ ou $A \to CB$ com $B \in N_\varepsilon$ e $C \in V$, adicionamos à gramática a regra $A \to C$. Qualquer derivação que possa ser executada pela gramática original pode ser simulada pela nova gramática e vice-versa – com uma exceção: ε não pode ser mais derivado na linguagem, uma vez que podemos ter omitido a regra $S \to \varepsilon$ durante esse passo. Felizmente, o enunciado do Teorema permite essa exclusão.

Exemplo 3.6.1 (continuação): Vamos continuar a partir da gramática anteriormente obtida com as regras

$$S \to SS,\ S \to (S_1,\ S_1 \to S),\ S \to \varepsilon.$$

Começamos computando o conjunto N_ε de não-terminais apagáveis: inicialmente $N_\varepsilon = \emptyset$; então $N_\varepsilon = \{S\}$, devido à regra $S \to \varepsilon$; e esse é o valor final de N_ε. Omitimos da gramática as regras ε (das quais há somente uma, $S \to \varepsilon$) e adicionamos variantes de todas as regras com uma ocorrência de S, nas quais essa ocorrência é omitida. O novo conjunto de regras é

$$S \to SS,\ S \to (S_1,\ S_1 \to S),\quad S \to S,\ S_1 \to).$$

A regra $S \to S$ foi adicionada devido à regra $S \to SS$ com $S \in \varepsilon$; ela é naturalmente inútil e pode ser omitida. A regra $S_1 \to)$ foi adicionada por causa da regra $S_1 \to S)$ com $S \in N_\varepsilon$.

Por exemplo, a derivação na gramática original

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow ()$$

pode agora ser simulada por

$$S \Rightarrow (S_1 \quad \ldots \quad \ldots$$

— omitindo a parte $S \Rightarrow SS$, uma vez que o primeiro S acabaria sendo apagado – e, finalmente,

$$(S_1 \Rightarrow ()$$

— utilizando a regra $S_1 \to)$ para antecipar o apagamento do S na regra $S_1 \to S)$.✧

Agora nossa gramática tem apenas regras cujos lados direitos apresentam comprimento um e dois. Devemos, em seguida, eliminar as regras curtas, aquelas cujos lados direitos têm comprimento um. Realizamos isso como segue: para cada $A \in V$ computamos, novamente pela aplicação de um simples algoritmo de fechamento, o conjunto $\mathcal{D}(A)$ *de símbolos que podem ser derivados de* A na gramática, $\mathcal{D}(A) = \{B \in V : A \Rightarrow^* B\}$, como segue:

$\mathcal{D}(A) := \{A\}$
enquanto existir uma regra $B \to C$ com $B \in \mathcal{D}(A)$ e
  $C \notin \mathcal{D}(A)$ acrescentar C **a** $\mathcal{D}(A)$.

Note-se que, para todos os símbolos A, $A \in \mathcal{D}(A)$; e, se a é um terminal, então $\mathcal{D}(a) = \{a\}$.

No terceiro e último passo da transformação de nossa gramática para a forma normal de Chomsky, omitimos todas as regras curtas da gramática e substituímos cada regra da forma $A \to BC$ por todas as possíveis regras da forma $A \to B'C$, onde $B' \in \mathcal{D}(B)$ e $C' \in \mathcal{D}(C)$. Tal regra simula o efeito da regra original $A \to BC$, com a seqüência de regras curtas que produz B' a partir de B e C' a partir de C. Finalmente, adicionamos as regras $S \to BC$ para cada regra $A \to BC$, tal que $A \in \mathcal{D}(S) - \{S\}$.

Novamente, a gramática resultante é equivalente àquela que existia antes da omissão das regras curtas, uma vez que o efeito de uma regra curta passou a ser simulado por meio da "antecipação" de sua utilização na primeira vez que o lado esquerdo aparece na derivação (se o lado esquerdo é S e, portanto, inicia a derivação, as regras $A \to BC$ que foram adicionadas na última etapa dessa construção são suficientes para garantir a equivalência). Há, novamente, uma única exceção: podemos ter removido uma regra $S \to \alpha$, e, portanto, omitido a cadeia α da linguagem gerada por G. Mais uma vez, felizmente, essa omissão é permitida pelo enunciado do Teorema.

Exemplo 3.6.1 (continuação): Em nossa gramática modificada, agora com as regras

$$S \to SS,\ S \to (S_1,\ S_1 \to S),\quad S_1 \to)$$

tem-se $\mathcal{D}(S_1) = \{S_1,)\}$ e $\mathcal{D}(A) = \{A\}$ para todos os $A \in V - \{S_1\}$. Omitimos agora todas as regras de comprimento um, das quais há somente uma, $S_1 \to)$. O único não-terminal com um conjunto incomum, $\mathcal{D}(S_1)$, aparece no lado direito da segunda regra apenas. Essa regra é, portanto, substituída pelas duas regras $S \to (S_1$, $S \to ($, correspondendo aos dois elementos de $\mathcal{D}(S_1)$. A gramática final, expressa na forma normal de Chomsky, apresenta as seguintes regras.

$$S \to SS,\ S \to (S_1,\ S_1 \to S),\quad S \to ()$$

✧

Após a aplicação dos três passos da transformação, a gramática está na forma normal de Chomsky e, exceto pela possível omissão de cadeias de comprimento menor que dois, ela gera a mesma linguagem que a original.

A fim de completar a prova do teorema, devemos estabelecer que essa transformação pode ser realizada em tempo polinomial em relação ao *tamanho da gramática original G*. Por "tamanho de G" queremos dizer o comprimento de uma cadeia que seja suficiente para descrever *por inteiro a gramática* G – isto é, a soma dos comprimentos de todas as regras de G. Suponha que n seja esse comprimento. A primeira parte da transformação (eliminação das regras longas), consome um tempo $O(n)$ e cria uma gramática de tamanho $O(n)$ novamente. A segunda parte, a eliminação das regras ε, consome um tempo $O(n^2)$ para a computação de fechamento ($O(n)$ interações, cada uma realizável em tempo $O(n)$), mais $O(n)$ para adicionar as novas regras. Finalmente, a terceira parte (tratamento das regras curtas) pode também ser realizada em tempo polinomial ($O(n)$ computações de fechamento, cada uma levando o tempo $O(n^2)$). Isso completa a prova do teorema.■

A vantagem da forma normal de Chomsky é que ela permite utilizar um simples algoritmo polinomial para decidir se uma cadeia pode ou não ser

gerada pela gramática. Suponha-mos que nos seja dada uma gramática livre de contexto $G = (V, \Sigma, R, S)$ na forma normal de Chomsky, e que nos seja perguntado se a cadeia $x = x_1 \ldots x_n$, com $n \geq 2$, pertence a $L(G)$. O algoritmo é mostrado a seguir. Ele decide se $x \in L(G)$ *analisando todas as sub-cadeias de* x. Para cada i e s, tal que $1 \leq i \leq i + s \leq n$, define-se $N[i, i+s]$ como sendo os conjuntos de todos os símbolos em V que podem derivar em G a cadeia $x_i \ldots x_{i+s}$. O algoritmo obtém esses conjuntos. Ele procede computando $N[i, i+s]$ de cadeias curtas (s pequeno) para cadeias cada vez mais longas. Esse método geral de resolver um problema começando pelos seus subproblemas menores e construindo soluções para subproblemas cada vez maiores, até que todo o problema seja resolvido, é conhecida como **programação dinâmica**.

Para i:= 1 até n faça $N[i, i]$:= $\{x_i\}$; todos os outros $N[i, j]$ são inicialmente vazios
Para s:= 1 até $n - 1$ faça
  Para i:= 1 até $n - s$ faça
    Para k:= i até $i+s-1$ faça
    Se existe uma regra $A \to BC \in R$ com $B \in N[i, k]$ e $C \in N[k+1, i+s]$,
    Então acrescentar A ao conjunto $N[i, i+s]$.
  Aceite x se $S \in N[1, n]$.

Para garantir que o algoritmo acima determina corretamente se $x \in L(G)$, devemos provar a seguinte afirmação:

Afirmação: *Para cada número natural* s *com* $0 \leq s \leq n$, *após a* s*-ésima iteração do algoritmo, para todos os* $i = 1, \ldots, n - s$, vale o seguinte:

$$N[i, i+s] = \{A \in V: A \Rightarrow^* x_i \ldots x_{i+s}\}.$$

***Prova da afirmação*:** a prova dessa afirmação é por indução em s.

Base de indução. Quando $s = 0$ – onde por "zero-ésima iteração do algoritmo" entendemos a primeira linha (de preparação inicial) – a afirmação é verdadeira: Como G se encontra na forma normal de Chomsky, o único símbolo que pode gerar o terminal x_i é o próprio x_i.

Passo de indução: Suponha que a afirmação seja verdadeira para todos os inteiros menores que $s > 0$. Considere uma derivação da subcadeia $x_i \ldots x_{i+s}$, por exemplo a partir de um não-terminal A. Como G está na forma normal de Chomsky, a derivação é iniciada com uma regra da forma $A \to BC$, isto é,

$$A \Rightarrow BC \Rightarrow^* x_i \ldots x_{i+s},$$

onde $B, C \in V$. Portanto, para algum k com $i < k \leq i + s$, temos

$$B \Rightarrow^* x_i \ldots x_k \text{ e } C \Rightarrow^* x_{k+1} \ldots x_{i+s}.$$

Concluímos que $A \in \{A \in V : A \Rightarrow^* x_i \ldots x_{i+s}\}$ se e somente se, existir um inteiro k, $i \leq k < i + s$, e dois símbolos $B \in \{A \in V: A \Rightarrow^* x_i \ldots x_k\}$ e $C \in \{A \in V: A \Rightarrow^* x_{k+1} \ldots x_{i+s}\}$, tais que $A \to BC \in R$. Podemos reescrever a

cadeia $x_i \ldots x_k$ como $x_i \ldots x_{i+s'}$, onde $s' = k - i$ e a cadeia $x_{k+1} \ldots x_{i+s}$, como $x_{k+1} \ldots x_{k+1+s''}$, onde $s'' = i + s - k - 1$. Note que, uma vez que $i \leq k < i + s$, devemos ter $s', s'' < s$. *Portanto, a hipótese de indução se aplica!*

Pela hipótese de indução, $\{A \in V: A \Rightarrow^* x_i \ldots x_k\} = N[i, k]$ e $\{A \in V: A \Rightarrow^* x_{k+1} \ldots x_{i+s}\} = N[k+1, i+s]$. Concluímos que $A \in \{A \in V: A \Rightarrow^* x_i \ldots x_{i+s}\}$, se, e somente se, há um inteiro k, $i \leq k < i + s$ e dois símbolos $B \in N[i, k]$ e $C \in N[k + 1, i + s]$, tais que $A \rightarrow BC \in R$. Mas essas são precisamente as circunstâncias sob as quais nosso algoritmo adiciona A a $N[i, i + s]$. Portanto, a afirmação aplica-se a s também, e isso conclui a prova da hipótese da indução – e da afirmação. ■

Segue-se imediatamente à afirmação que o algoritmo acima decide corretamente se $x \in L(G)$: no fim, o conjunto $N[1, n]$ conterá todos os símbolos que derivam a cadeia $x_i \ldots x_n = x$. Portanto, $x \in L(G)$, se, e somente se, $S \in N[1, n]$.

Para analisarmos o desempenho em tempo desse algoritmo, notemos que ele consta de três laços aninhados, cada qual limitado por $|x| = n$. No núcleo do laço, devemos examinar cada regra da forma $A \rightarrow BC$, se $B \in N[i, j]$ e $C \in N[j + 1, i + s]$; isso pode ser realizado em tempo proporcional ao tamanho da gramática G. Concluímos que o número total de operações é $O(|x|^3 |G|)$ – um polinômio, *tanto* no comprimento de x *como* no tamanho de G. Para qualquer gramática *fixa* G (isto é, quando considerarmos $|G|$ como sendo uma constante), o algoritmo é executado em tempo $O(n^3)$. ■

Exemplo 3.6.1 (continuação): Vamos aplicar o algoritmo de programação dinâmica à gramática dos parênteses balanceados, expressa na forma normal de Chomsky através das regras

$$S \Rightarrow SS,\ S \Rightarrow (S_1,\ S_1 \rightarrow S),\ S \Rightarrow ().$$

Suponha que desejássemos verificar se a cadeia (()(())) pode ou não ser gerada por G. Mostramos na Figura 3.10 os valores de $N[i, i + s]$ para $1 \leq i \leq j \leq n = 8$, resultantes das interações do algoritmo. A computação procede ao longo de diagonais paralelas da tabela. A diagonal principal, correspondente a $s = 0$, contém a cadeia a ser analisada sintaticamente. Para preencher uma célula, por exemplo, [2, 7], consultamos todos os pares de células da forma $N[2, k]$ e $N[k + 1, 7]$ com $2 \leq k < 7$. Todas essas células estão à esquerda ou acima da célula a ser preenchida. Para $k = 3$, notamos que $S \in N[2,3]$, $S \in N[4,7]$, e $S \rightarrow SS$ é uma regra; portanto, devemos adicionar o lado esquerdo de S à célula $N[2, 7]$. E assim por diante. O extremo inferior direito é $N[1, n]$ e ele realmente contém S; portanto, a cadeia está realmente em $L(G)$. De fato, inspecionando essa tabela é fácil construir a derivação da cadeia (()(())) em G. O algoritmo de programação dinâmica pode ser facilmente modificado para efetuar tal derivação; veja o Problema 3.6.2. ✧

A Parte (c) do Teorema 3.6.1 agora se aplica, combinando os Teoremas 3.6.2 e a afirmação acima: dada uma gramática livre de contexto G e uma cadeia x, determinamos se $x \in L(G)$ como segue: primeiro, transformamos G em uma gramática livre de contexto equivalente a G', porém expressa na forma normal de Chomsky, conforme a técnica utilizada na prova do Teorema 3.6.2, em tempo polinomial. No caso especial em que $|x| \leq 1$, já podemos decidir se $x \in L(G)$: ele pertence, se e somente se, durante a transformação tivemos de excluir uma regra $S \rightarrow x$. De outro modo, executamos o algoritmo de programação dinâmica descrito acima para a gramática G' e a cadeia x. O número total de operações utilizadas pelo algoritmo é limitado por um polinômio sobre o tamanho da gramática original G e sobre o comprimento da cadeia x. ■

	1	2	3	4	5	6	7	8
1	(	Ø	Ø	Ø	Ø	Ø	Ø	S
2	(	S	Ø	Ø	Ø	Ø	S	S_1
3	)	Ø	Ø	Ø	Ø	Ø	Ø	Ø
4	(	Ø	Ø	S	S	S_1	Ø	
5	(	S	S_1	Ø				
6	)	Ø	Ø					
7	)	Ø						
8	)							

Figura 3-10

Problemas para a Seção 3.6

3.6.1 Converta a gramática livre de contexto G, dada no Exemplo 3.1.3, que gera expressões aritméticas, para uma gramática livre de contexto equivalente, expressa na forma normal de Chomsky. Aplique o algoritmo de programação dinâmica para decidir se a cadeia x = (id + id + id) * (id) está em $L(G)$.

3.6.2 Como se pode modificar o algoritmo de programação dinâmica para que, quando a entrada x de fato pertencer à linguagem gerada por G, então o algoritmo construa a correta derivação de x em G?

3.6.3 (a) Seja $G = (V, \Sigma, R, S)$ uma linguagem livre de contexto. Diz-se que um não-terminal $A \in V - \Sigma$ é *produtivo*, se $A \Rightarrow^*_G x$ para algum $x \in \Sigma^*$. Forneça um algoritmo polinomial G que determina todos os não-terminais produtivos de G. (*Sugestão*: Este é um algoritmo de fechamento.)

(b) Apresente um algoritmo polinomial que, dada uma gramática G livre de contexto, decide se $L(G) = \emptyset$.

3.6.4 Descreva um algoritmo que, dada uma gramática G livre de contexto, decide se $L(G)$ é infinita. (*Sugestão*: por exemplo usar o Teorema de bombeamento.) Qual é a complexidade de seu algoritmo? Você pode construir um algoritmo equivalente, que opere em tempo polinomial?

3.7 DETERMINISMO E ANÁLISE SINTÁTICA

As gramáticas livres de contexto são extensivamente utilizadas para a modelagem da sintaxe de linguagens de programação, como foi sugerido pelo Exemplo 3.1.3.† Um compilador para tais linguagens de programação deve, então, incorporar um **analisador sintático**, isto é, um algoritmo para determinar se uma dada cadeia pertence à linguagem gerada por uma gramática livre de contexto, e, em caso afirmativo, construir a árvore de análise sintática da cadeia. (O compilador iria, então, prosseguir traduzindo a árvore de análise sintática para a forma de um programa denotado em alguma linguagem mais básica, como por exemplo, uma linguagem de montagem.) O analisador sintático geral livre de contexto que desenvolvemos na seção anterior, o algoritmo de programação dinâmica, apesar de polinomial, é muito lento para manipular programas com dezenas de milhares de instruções (lembre-se de seu comportamento cúbico em relação ao comprimento da cadeia). Muitas técnicas de análise sintática foram desenvolvidas pelos projetistas de compiladores nas últimas quatro décadas. Uma das mais bem-sucedidas fundamenta-se na idéia do autômato de pilha, dada a equivalência entre autômatos de pilha e gramáticas livres de contexto, provada na Seção 3.4. Entretanto, um autômato de pilha não se aplica imediatamente na construção de analisadores sintáticos, pois opera como um *dispositivo não-determinístico*. Surge a seguinte questão: seria possível, no caso geral, que autômatos de pilha operem deterministicamente (como no caso dos autômatos finitos)?

O objetivo principal desta seção é investigar os autômatos de pilha determinísticos. Iremos constatar a existência de linguagens livres de contexto que não podem ser aceitas por autômatos de pilha determinísticos. Isso é bastante desapontador, e sugere que o método de transformação de gramáticas em autômatos, desenvolvida na Seção 3.4, não pode fundamentar um método prático. Entretanto, constata-se que, para a maioria das linguagens de programação, é possível construir autômatos de pilha determinísticos que aceitem programas sintaticamente corretos. Adiante, nesta seção, apresentaremos algumas *heurísticas* – regras práticas – que se mostram úteis para a construção de autômatos de pilha determinísticos a partir de gramáticas livres de contexto particulares. Essas regras nem sempre produzirão autômatos de pilha práticos a partir de qualquer gramática livre de contexto; como foi dito, isso é impossível, embora sejam elas utilizadas com freqüência na construção de compiladores para linguagens de programação.

Linguagens determinísticas

Um autômato de pilha M é dito **determinístico** quando cada uma de suas configurações apresenta, no máximo, alguma configuração que pode sucedê-la em qualquer computação executada por M. Isso pode ser expresso de outra forma equivalente: Dizemos que duas cadeias são **consistentes** se a primeira for um prefixo da segunda ou vice-versa. Dizemos que duas transições $((p, a, \beta), (q, \gamma))$ e $((p, a', \beta'), (q', \gamma'))$ **são compatíveis** quando a e a' forem

† N. de R. Como foi comentado anteriormente, a modelagem de linguagens de programação por meio de gramáticas livres de contexto é apenas parcial, exigindo complementos para a representação integral da linguagem.

consistentes, assim como β e β', ou, em outras palavras, se houver uma situação na qual ambas as transições sejam aplicáveis. Portanto, M será determinístico se não apresentar transições compatíveis distintas.

Por exemplo, a máquina construída no Exemplo 3.3.1 para aceitar a linguagem $\{wcw^R: w \in \{a, b\}^*\}$ é determinística: para cada possível escolha do estado e do símbolo de entrada, há somente uma transição possível. Por outro lado, a máquina construída no Exemplo 3.3.2 para aceitar $\{ww^R: w \in \{a, b\}^*\}$ não é determinística, pois a transição 3 é compatível com as transições 1 e 2: note-se que essas são as transições que "adivinham" a localização do centro da cadeia – uma atividade intuitivamente não-determinística.

Linguagens livres de contexto determinísticas são essencialmente aquelas aceitas por autômatos de pilha determinísticos. Entretanto, por motivos que em breve se tornarão claros, é preciso modificar ligeiramente o critério de aceitação. Diz-se que uma linguagem livre de contexto é determinística se ela for reconhecida por um autômato determinístico de pilha, que apresente também a capacidade extra de *localizar o final da cadeia de entrada*. Formalmente, diz-se que uma linguagem $L \subseteq \Sigma^*$ é **determinística e livre de contexto**, se $L\$ = L(M)$ para algum autômato determinístico de pilha M. Aqui \$ é um novo símbolo, não contido em Σ, que é acrescentado ao final de cada cadeia de entrada com o propósito de identificar essa posição.

Toda linguagem determinística livre de contexto, caracterizada da forma que acabamos de definir, é também uma linguagem livre de contexto. Para constatar esse fato, seja um autômato de pilha determinístico M, que aceita $L\$$. Então, é possível construir um autômato de pilha (não-determinístico) M' que aceita L. Em qualquer ponto, M' pode "imaginar" um símbolo \$ na entrada e transitar para um novo conjunto de estados, a partir dos quais ele não mais aceita símbolos de entrada adicionais.

Se, por outro lado, não adotassemos esse critério especial de aceitação, muitas linguagens livres de contexto, que são intuitivamente determinísticas, não seriam consideradas determinísticas segundo a nossa definição. Um exemplo é $L = a^* \cup \{a^n b^n: n \geq 1\}$. Um autômato determinístico de pilha não tem como manter o registro do número de a's já recebidos, com a finalidade de verificar o comprimento da cadeia de b's que poderá ser encontrada a seguir, e, ao mesmo tempo estar pronto para aceitar a cadeia (pelo critério da pilha vazia) no caso de nenhum b estar presente na cadeia. Entretanto, pode-se facilmente projetar um autômato de pilha determinístico que aceite $L\$$: se um \$ é encontrado enquanto a máquina está ainda acumulando a's, então a entrada pode ser classificada como uma cadeia da forma a^*. Se isso acontecer, a pilha deve ser esvaziada, e a cadeia de entrada, aceita.

Uma questão natural nesse ponto é se cada linguagem livre de contexto não seria determinística – de modo similar ao que ocorre com as linguagens regulares que sempre podem ser aceitas por um autômato finito determinístico. Seria muito interessante se esse fato ocorresse. Considere, por exemplo, a linguagem livre de contexto

$$L = \{a^n b^m c^p: m, n, p \geq 0 \text{ e } m \neq n \text{ ou } m \neq p\}$$

Pode parecer, à primeira vista que um autômato de pilha poderia aceitar essa linguagem somente adivinhando quais dos blocos de símbolos compa-

rar: os a's com os b's *ou* os b's com os c's. Sem lançar mão do não-determinismo, pode parecer que essa máquina não tem como comparar os b's com os a's, ao mesmo tempo que se prepara para comparar os b's com os c's. Entretanto, provar que L não é determinístico requer uma argumentação menos imediata: *o complemento de L não é livre de contexto.*

Teorema 3.7.1: *A classe das linguagens livres de contexto determinísticas é fechada em relação à operação de complementação.*

Prova: Suponha que $L \subseteq \Sigma^*$ seja uma linguagem tal, que $L\$$ é aceita por um autômato de pilha determinístico $M = (K, \Sigma, \Gamma, \Delta, s, F)$. Será conveniente assumir, como foi feito na prova do Lema 3.4.2, que M é *simples*, isto é, nenhuma transição de M remove mais que um símbolo da pilha, enquanto uma transição inicial empilha o símbolo Z (que marca o fundo da pilha), o qual será removido antes do final da computação; é fácil constatar que o método empregado para isso, na prova do Lema 3.4.2, não altera as características, relativas ao determinismo, da máquina M.

Sendo M, por hipótese, um autômato de pilha determinístico, à primeira vista bastaria, para obter um dispositivo que aceitasse $(\Sigma^* - L)\$$, inverter os estados de aceitação e rejeição – como foi feito com os autômatos finitos determinísticos na prova do Teorema 2.3.1(d), e como será feito novamente no próximo capítulo, com dispositivos determinísticos mais complexos. Neste caso, entretanto, uma simples inversão não é suficiente, uma vez que um autômato de pilha determinístico pode rejeitar uma entrada não somente por alcançar um estado de não-aceitação ao final da leitura da cadeia de entrada, mas também por *não ter a possibilidade de terminar de ler sua entrada.* Essa situação pode ocorrer em duas ocasiões: na primeira, M pode ser exposta a uma configuração C na qual nenhuma das transições contidas na relação Δ é aplicável. Na segunda, e talvez mais intrigante, M pode encontrar-se diante de uma configuração em que venha a executar uma seqüência interminável de transições em vazio (transições da forma (q, ε, α) (p, β)).

Vamos chamar uma configuração $C = (q, w, \alpha)$ de M de **sem saída**, se o seguinte é verdadeiro: se $C \vdash_M^* C'$ para alguma outra configuração $C' = (q', w', \alpha')$, então $w' = w$ e $|\alpha'| \geq |\alpha|$. Isto é, dizemos que uma configuração é sem saída se nenhum progresso pode ser feito a partir dela, tanto lendo uma entrada como reduzindo a altura da pilha. Obviamente, se M está em uma configuração sem saída, então ele irá de fato falhar ao ler sua entrada até o fim. Inversamente, não é difícil ver que, se M não tem configurações sem saída, então ele definitivamente lerá toda sua entrada. Essa é a razão por que, na falta de configurações sem saída, em todas as vezes há uma vez no futuro em que ou o próximo símbolo de entrada será lido ou a altura da pilha será diminuída – e a segunda opção pode somente ser assumida um número finito de vezes, uma vez que o comprimento da pilha não pode ser diminuído infinitamente.

Vamos mostrar como transformar qualquer autômato de pilha determinístico simples M em um autômato de pilha determinístico equivalente, isento de configurações sem saída. Considerando que M é, por hipótese, um autômato simples, o fato de uma configuração ser ou não sem saída *depende somente do estado corrente, do próximo símbolo de entrada e do símbolo contido no topo da pilha.* Em particular, suponha que $q \in K$ seja um estado, $a \in \Sigma$ um símbolo de entrada e $A \in \Gamma$ um símbolo de pilha. Dizemos que a tripla (q, a, A) é um término por morte se não houver um estado p e uma cadeia de símbolos de pilha, tal que a *configuração* (q, a, A) resulta em (p, ε, α) ou (p, a, ε). Em outras palavras, uma tripla (q, a, A) é sem saída se for um término por morte quando considerada como configuração. Seja $D \subseteq K \times \Sigma \times \Gamma$ o conjunto de todas as triplas término por morte. Note-se que estamos declarando que é possível afirmar, examinando uma tripla, se ela pertence ou não ao conjunto D: tudo que estamos dizendo é que o conjunto D é um conjunto de triplas bem definido e finito.

Propomos a seguinte modificação de M: para cada tripla $(q, a, A) \in D$, removemos de Δ todas as transições compatíveis com (q, a, A) e adicionamos a Δ a transição $((q, a, A), (r, \varepsilon))$, onde r é um novo estado de rejeição. Por fim, adicionamos a Δ estas transições: $((r, a, \varepsilon), (r, \varepsilon))$ para todo $a \in \Sigma$, $((r, \$, \varepsilon), (r', \varepsilon))$ e $((r', \varepsilon, A), (r', \varepsilon))$ a cada $A \in \Gamma \cup \{Z\}$, onde r' é outro estado novo não-aceitável. Essas transições permitem que M', quando no estado r, leia toda a entrada (sem consultar a pilha), e, ao ler um \$, esvaziar a pilha e rejeitar. O resultado dessa operação é o autômato de pilha M'.

É fácil verificar que M' é determinístico e que aceita a mesma linguagem que de M (M' simplesmente rejeita, de modo explícito, aquelas cadeias que M tenha rejeitado implicitamente por não ser capaz de ter toda a cadeia da entrada). Além disso, M' foi construído de modo tal que não tenha configurações sem saída. Dessa forma, M sempre terminará lendo toda sua entrada. Agora, a inversão dos papéis dos estados de aceitação e da rejeição de M' produzirá finalmente, um autômato determinístico de pilha que aceita $(\Sigma^* - L)\$$, completando a prova. ■

O Teorema 3.7.1, de fato, estabelece que a linguagem livre de contexto $L = \{a^n b^n c^p : m \neq n \text{ ou } m \neq p\}$ acima não é determinística: se L fosse determinística, então sua linguagem complemento $\overline{L}$ também seria determinística e livre de contexto, e, portanto, certamente livre de contexto. Desse modo, a intersecção de $\overline{L}$ com a linguagem regular $a^*b^*c^*$ seria livre de contexto, pelo Teorema 3.5.2. Mas é fácil ver que $\overline{L} \cap a^*b^*c^* = \{a^n b^n c^n : n \geq 0\}$, a qual sabemos que não é livre de contexto. Concluímos que a linguagem L livre de contexto não pode ser uma linguagem livre de contexto determinística.

Corolário: *A classe de linguagens livres de contexto determinísticas está propriamente contida na classe de linguagens livres de contexto*

Em outras palavras, *o não-determinismo é mais poderoso do que o determinismo, no âmbito dos autômatos de pilha.* Em contraste, foi constatado, no último capítulo, que o não-determinismo nada adicionava ao poder dos autômatos finitos, a não ser que o número de estados seja levado em consideração. Sob esse ângulo, o não-determinismo é *exponencialmente* mais poderoso.

Essa intrigante questão do poder diferenciado do não-determinismo em vários contextos computacionais é talvez a mais importante apresentada neste livro.

Análise sintática descendente ("*top-down*")

Uma vez constatado que nem toda linguagem livre de contexto pode ser aceita por um autômato de pilha determinístico, consideramos a seguir algumas linguagens que não estão sujeitas a essa restrição. Nossa meta para o restante deste capítulo é estudar gramáticas livres de contexto que podem ser convertidas em autômatos de pilha determinísticos aplicáveis ao reconhecimento de linguagens em aplicações reais. Vamos, no entanto, nos afastar um pouco do estilo que caracteriza o restante desta publicação: haverá menos provas e não tentaremos concluir todos os aspectos da teoria anteriormente introduzida. Apresentamos algumas heurísticas, que nem sempre serão aplicáveis, e não nos preocuparemos em precisar as condições exatas de sua aplicabilidade. Em lugar disso pretendemos apresentar algumas aplicações sugestivas da teoria desenvolvida anteriormente neste capítulo, mas apenas a título de introdução.

Vamos começar com um exemplo. A linguagem $L = \{a^n b^n\}$ é gerada pela gramática livre de contexto $G = (\{a, b, S\}, \{a, b\}, R, S)$, onde R contém as regras $S \rightarrow aSb$ e $S \rightarrow \varepsilon$. Sabemos como construir um autômato de pilha que aceita L: basta executar o método utilizado no Lema 3.4.1 para a gramática G. O resultado é

$$M_1 = (\{p, q\}, \{a, b\}, \{a, b, S\}, \Delta_1, p, \{q\}),$$

onde

$$\Delta_1 = \{((p, \varepsilon, \varepsilon), (q, S)), ((q, \varepsilon, S), (q, aSb)) ((q, \varepsilon, S), (q, \varepsilon)), ((q, a, a), (q, \varepsilon)), ((q, b, b), (q, \varepsilon))\}.$$

Como M_1 apresenta duas transições diferentes cujos primeiros componentes são idênticos – os correspondentes às duas regras de G que têm lados esquerdos idênticos – ela não é determinística.

Contudo, L é uma linguagem livre de contexto determinística e M_1 pode ser modificada para tornar-se um autômato de pilha determinístico M_2 que aceite $L\$$. Intuitivamente, toda a informação de que M_1 necessita, em cada ponto, para poder decidir qual das duas transições deve aplicar está restrita ao próximo símbolo de entrada. Se esse símbolo for um a, então M_1 deve substituir S por aSb em sua pilha, para que seja mantida a esperança de uma computação aceitável. Por outro lado, se o próximo símbolo de entrada for um b, então a máquina deve remover o símbolo S. M_2 executa com êxito essa consulta prévia à cadeia de entrada (*lookahead*), consumindo antecipadamente um símbolo da entrada e incorporando essa informação em seu estado. Formalmente,

$$M_1 = (\{p, q, q_a, q_b, q_S\}, \{a, b\}, \{a, b, S\}\ \Delta_2, p, \{q_S\}),$$

onde Δ_2 contém as seguintes transições:

(1) $((p, \varepsilon, \varepsilon), (q, S))$
(2) $((q, a, \varepsilon), (q_a, \varepsilon))$
(3) $((q_a, \varepsilon, a), (q, \varepsilon))$
(4) $((q, b, \varepsilon), (q_b, \varepsilon))$
(5) $((q_b, \varepsilon, b), (q, \varepsilon))$
(6) $((q, \$, \varepsilon), (q_S, \varepsilon))$
(7) $((q_a, \varepsilon, S), (q_a, aSb))$
(8) $((q_b, \varepsilon, S), (q_b, \varepsilon))$

A partir do estado q, M_2 lê um símbolo de entrada e, sem alterar o conteúdo da pilha, transita para um dos três novos estados q_a, q_b ou q_S. Ele usa essa informação para diferenciar as duas transições compatíveis $((q, \varepsilon, S), (q, aSb))$ e $((q, \varepsilon, S), (q, \varepsilon))$: a primeira transição é efetivada somente a partir do estado q_a. E a segunda, somente a partir do estado q_b. Assim, constatamos que M_2 é determinístico, e aceita a entrada abS como segue.

Passo	Estado	Entrada ainda não lida	Pilha	Transição utilizada	Regra de G
0	p	abS	ε	-	
1	q	abS	S	1	
2	q_a	bS	S	2	
3	q_a	bS	aSb	7	$S \rightarrow aSb$
4	q	bS	Sb	3	
5	q_b	S	Sb	4	
6	q_b	S	b	8	$S \rightarrow \varepsilon$
7	q	S	ε	5	
8	q_S	ε	ε	6	

Dessa forma, M_2 pode operar como um dispositivo determinístico para reconhecer cadeias da forma $a^n b^n$. Além disso, memorizando para cada transição de M_2 as regras da gramática das quais foram obtidas (última coluna da tabela acima), podemos, a partir do rastreamento da operação de M_2, reconstruir uma derivação mais à esquerda da cadeia de entrada. Especificamente, os passos da computação nos quais um não-terminal é substituído no topo da pilha (passos 3 e 6 no exemplo) correspondem à construção de uma árvore de análise sintática, partindo-se da raiz em direção às folhas (veja a Figura 3-11(a)).

Dispositivos como M_2, que corretamente decidem se uma cadeia pertence ou não a uma linguagem livre de contexto, e, no caso afirmativo, produzem a correspondente árvore de análise sintática, são denominados **analisadores sintáticos**. Em particular, M_2 é um **analisador sintático** descendente uma vez que o rastreamento dos passos de sua operação nos quais não-terminais são substituídos na pilha, permite a construção de uma árvore de análise sintática partindo da raiz para as folhas, e *da esquerda para a direita* (veja na Figura 3-11(b) uma representação sugestiva do progresso da análise sintática descendente). Estudaremos um exemplo mais completo adiante.

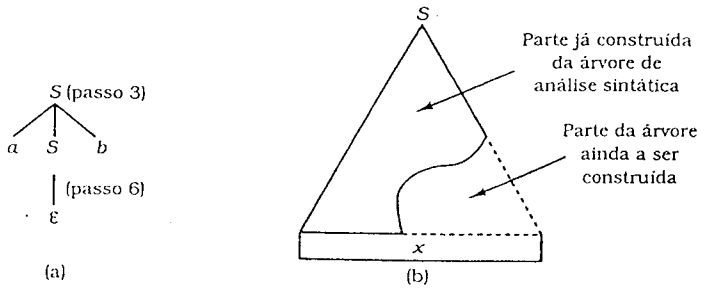


Figura 3-11

Naturalmente, nem todas as linguagens livres de contexto têm reconhecedores determinísticos que podem ser derivados a partir de um autômato de pilha não-determinístico por meio da técnica de antecipação *(lookahead)*. Por exemplo, vimos na subseção anterior que algumas linguagens livres de contexto nem sequer são determinísticas. Mesmo para certas linguagens livres de contexto determinísticas, a antecipação de apenas um símbolo pode não ser suficiente para eliminar todas as incertezas. Por motivos superficiais, que podem ser removidos modificando ligeiramente a gramática, algumas linguagens, não se mostram diretamente compatíveis com a análise sintática por *lookahead*. Focalizaremos esses casos a seguir.

Tomemos a gramática G que gera expressões aritméticas com operações + e * (Exemplo 3.1.3). Vamos enriquecer essa gramática com a regra adicional seguinte:

$$F \rightarrow \text{id(E)} \qquad (M)$$

projetada para permitir que *chamadas de função* – tais como *sqrt* $(x^* x + 1)$ e $f(z)$ – possam ser incluídas em nossas expressões aritméticas.

Vamos tentar construir um analisador sintático descendente a partir dessa gramática. A técnica descrita na Seção 3.4 produz o autômato de pilha

$$M_3 = (\{p, q\}, \Sigma, \Gamma, \Delta, p, \{f\}),$$

com

$$\Sigma = \{(,), +, *, \text{id}\},$$
$$\Gamma = \Sigma \cup \{E, T, F\},$$

e Δ conforme a lista abaixo.

(0) $((p, \varepsilon, \varepsilon), (q, E))$
(1) $((q, \varepsilon, E), (q\ E + T))$
(2) $((q, \varepsilon, E), (q, T))$
(3) $((q, \varepsilon, T), (q, T * F))$
(4) $((q, \varepsilon, T), (q, F))$
(5) $((q, \varepsilon, F), (q, (E)))$
(6) $((q, \varepsilon, F), (q, \text{id}))$
(7) $((q, \varepsilon, F), (q, \text{id}(E)))$

Por fim, $((q, a, a), (q, \varepsilon)) \in \Delta$ para todos os $a \in \Sigma$. O não-determinismo de M_3 é manifestado pelos conjuntos de transições 1-2, 3-4 e 5-6-7 que têm os primeiros componentes idênticos. *Agrava a situação o fato de que a escolha correta da regra não pode ser feita apenas com base no símbolo seguinte de entrada.* Examinenos com mais cuidado as razões desse problema.

Transições 6 e 7. Seja (q, id, F) uma configuração de M_3. Nessa situação, M_3 pode agir de acordo com qualquer uma das transições 5, 6 ou 7. Procurando o próximo símbolo de entrada – id –, M_3 pode excluir a transição 5, uma vez que essa transição requer que o próximo símbolo seja (. M_3 não será, porém, capaz de escolher dentre as transições 6 e 7, pois da execução de qualquer uma delas resulta um topo de pilha que pode ser identificado com o próximo símbolo de entrada – id . O problema surge porque as regras $F \rightarrow$ id e $F \rightarrow$ id(E) de G não têm apenas lados esquerdos idênticos, mas também o mesmo primeiro símbolo em seus lados direitos.

Há uma forma muito simples de contornar esse problema: basta substituir as regras $F \rightarrow$ id e $F \rightarrow$ id(E) em G pelas regras $F \rightarrow$ id$A$, $A \rightarrow \varepsilon$ e $A \rightarrow (E)$, onde A é um novo não-terminal. Isso tem o efeito de "adiar" a decisão entre as regras $F \rightarrow$ id e $F \rightarrow$ id(E), até que todas as informações necessárias para essa decisão estejam disponíveis. Um autômato de pilha M_3' modificado resulta agora dessa gramática transformada, na qual as transições 6 e 7 são substituídas pelas seguintes:

(6') $((q, \varepsilon, F), (q, \text{id}A))$
(7') $((q, \varepsilon, A), (q, \varepsilon))$
(8') $((q, \varepsilon, A), (q, (E)))$

Consultar um símbolo antecipadamente é suficiente para que M_3' possa decidir a ação correta a ser tomada. Por exemplo, a configuração $(q, \text{id (id)}, F)$ iria resultar $(q, \text{id (id)}, \text{id}A), (q, \text{(id)}, A), (q, \text{(id)}, (E))$ e assim por diante.

Essa técnica para evitar não-determinismos é conhecida como **fatoração à esquerda**, e pode ser resumida como segue.

Heurística 1: *Sempre que* $A \rightarrow \alpha\beta_1$, $A \rightarrow \alpha\beta_2, \ldots, A \rightarrow \alpha\beta_m$ *forem regras com* $\alpha \neq \varepsilon$ *e* $n \geq 2$, *então deve-se substituí-las pelas regras* $A \rightarrow \alpha A'$ *e* $A' \rightarrow \beta_i$ *para* $i = 1, \ldots, n$, *onde* A' *é um novo não-terminal.*

É fácil constatar que a aplicação da heurística 1 não altera a linguagem gerada pela gramática.

Agora passaremos a examinar o segundo tipo de anomalia que nos impede de transformar M_3 em um analisador sintático determinístico.

Transições 1 e 2. Essas transições apresentam um problema mais sério. Se o autômato recebe id como símbolo da entrada, e o conteúdo da pilha é apenas E, ele pode tomar diversas atitudes: Ele pode executar a transição 2, substituindo E por T (isso seria justificado caso a entrada seja id). Ele pode alternativamente substituir E por $E + T$ (transição 1) e, então substituir o topo E por T (isso deve ser feito se a entrada for id + id). Ele pode ainda executar a transição 2 duas vezes e a transição 1 uma vez (entrada id + id + id), e assim por diante. Não há limite para o quanto adiante o autômato deve consultar para poder decidir sobre a ação correta a ser tomada. Isso se deve à regra $E \rightarrow E + T$, na qual o não-terminal indicado em sua parte esquerda ocorre novamente como primeiro símbolo da sua parte direita. Esse fenômeno é denominado **recursão à esquerda**, e pode ser removido fazendo-se uma alteração adicional na gramática.

Para remover a recursão à esquerda da regra $E \rightarrow E + T$, basta substituí-la pelas regras $E \rightarrow TE'$, $E' \rightarrow +TE'$ e $E' \rightarrow \varepsilon$, onde E' é um novo não-terminal. Pode-se de mostrar que essas transformações não alteram a linguagem produzida pela gramática. O mesmo método também deve ser aplicado à outra regra de G que apresenta recursão à esquerda: $T \rightarrow T * F$. Assim, obtém-se gramática $G' = (V', \Sigma, R', E)$, onde $V' = \Sigma \cup \{E, E', T, T', F, A\}$, e as regras são as seguintes:

(1) $E \rightarrow TE'$
(2) $E' \rightarrow + TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow *FT'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow \text{id}A$
(9) $A \rightarrow \varepsilon$
(10) $A \rightarrow (E)$

A técnica acima utilizada para a remoção das recursões à esquerda presentes em uma gramática livre de contexto pode ser expressa como segue.†

Heurística 2: *Seja $A \rightarrow A\alpha_1, \ldots, A \rightarrow A\alpha_n$ e $A \rightarrow \beta_1, \ldots, A \rightarrow \beta_m$ todas as regras com A em seu lado esquerdo, onde os β_i's não começam com o símbolo A, e $m > 0$ (isto é, existe no mínimo uma regra recursiva à esquerda). Nesse caso, substituam-se essas regras por $A \rightarrow \beta_1 A', \ldots, A \rightarrow \beta_m A'$ e $A' \rightarrow \alpha_1 A', \ldots, A' \rightarrow \alpha_n A'$ e $A' \rightarrow \varepsilon$, onde A' é um novo não-terminal.*

Adicionalmente, a gramática G' de nosso exemplo apresenta regras com lados esquerdos idênticos, porém agora todas as incertezas podem ser resolvidas consultando antecipadamente o próximo símbolo de entrada. Podemos, portanto, construir o seguinte autômato de pilha determinístico M_4 que aceita $L(G)\$$.

$$M_4 = (K, \Sigma \cup \{\$\}, V', \Delta, p, \{q_\$\}),$$

onde

$$K = \{p, q, q_{\text{id}}, q_+, q_*, q_), q_(, q_\$\},$$

e Δ é relacionado abaixo.

$((p, \varepsilon, \varepsilon), (q, E))$
$((q, a, \varepsilon), (q_a, \varepsilon))$ para cada $a \in \Sigma \cup \{\$\}$
$((q_a, \varepsilon, a), (q, \varepsilon))$ para cada $a \in \Sigma$
$((q_a, \varepsilon, E), (q_a, TE'))$ para cada $a \in \Sigma \cup \{\$\}$
$((q_+, \varepsilon, E'), (q_+, +TE'))$
$((q_a, \varepsilon, E'), (q_a, \varepsilon))$ para cada $a \in \{), \$\}$
$((q_a, \varepsilon, T), (q_a, FT'))$ para cada $a \in \Sigma \cup \{\$\}$
$((q_*, \varepsilon, T'), (q_*, *FT'))$
$((q_a, \varepsilon, T'), (q_a, \varepsilon))$ para cada $a \in \{+,), \$\}$
$((q_(, \varepsilon, F), (q_(, (E)))$
$((q_{\text{id}}, \varepsilon, F), (q_{\text{id}}, \text{id}A))$
$((q_(, \varepsilon, A), (q_(, (E)))$
$((q_a, \varepsilon, A), (q_a, \varepsilon))$ para cada $a \in \{+, *,), \$\}$

Então, M_4 é um analisador sintático para G'. Por exemplo, a cadeia de entrada id * (id)$ é aceita como mostrado na tabela seguinte.

† Assumimos aqui que não há regras da forma $A \rightarrow A$.

Passo	Estado	Entrada ainda não-lida	Pilha	Regra de G'
0	p	id * (id)$	ε	
1	q	id * (id)$	E	
2	q_{id}	* (id)$	E	
3	q_{id}	* (id)$	TE'	1
4	q_{id}	* (id)$	$FT'E'$	4
5	q_{id}	* (id)$	id$AT'E'$	8
6	q	* (id)$	$AT'E'$	
7	q_*	(id)$	$AT'E'$	
8	q_*	(id)$	$T'E'$	9
9	q_*	(id)$	$*FT'E'$	5
10	q	(id)$	$FT'E'$	
11	$q_($	id)$	$FT'E'$	
12	$q_($	id)$	$(E)T'E'$	7
13	q	id)$	$E)T'E'$	
14	q_{id}	)$	$E)T'E'$	
15	q_{id}	)$	$TE)T'E'$	1
16	q_{id}	)$	$FT'E)T'E'$	4
17	q_{id}	)$	id$AT'E)T'E'$	8
18	q	)$	$AT'E)T'E'$	
19	$q_)$	$	$AT'E)T'E'$	
20	$q_)$	$	$T'E)T'E'$	9
21	$q_)$	$	$E)T'E'$	6
22	$q_)$	$	$)T'E'$	3
23	q	$	$T'E'$	
24	$q_\$$	ε	$T'E'$	
25	$q_\$$	ε	E'	6
26	$q_\$$	ε	ε	3

Aqui indicamos os passos da computação nos quais um não-terminal foi substituído na pilha de acordo com uma regra de G'. Aplicando as regras de G' indicadas na última coluna dessa tabela na mesma seqüência em que aparecem, obtemos uma derivação mais à esquerda da cadeia de entrada:†

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \text{id A } T'E' \text{ id } T'E' \Rightarrow \text{id} * FT'E' \Rightarrow \text{id} * (E)T'E' \Rightarrow$$

$$\text{id} * (TE')T'E' \Rightarrow \text{id} * (FT'E')T'E' \Rightarrow \text{id} * (\text{id A } T'E')\ T'E' \Rightarrow \text{id} * (\text{id}T'E')\ T'E' \Rightarrow$$

$$\text{id} * (\text{id}E')\ T'E' \Rightarrow \text{id} * (\text{id})\ T'E' \Rightarrow \text{id} * (\text{id})E' \Rightarrow \text{id} * (\text{id})$$

Assim, uma árvore *de análise sintática* da entrada pode ser construída (veja a Figura 3-12; o passo do autômato de pilha correspondente à expansão de cada vértice da árvore de análise sintática é também mostrado ao lado do vértice). Note-se que esse analisador sintático constrói a árvore de análise sintática da entrada partindo da raiz para as folhas, e da esquerda para a direita partindo da raiz E e repetidamente aplicando regra a apropriada, sempre substituindo o não-terminal mais à esquerda.

† N. de R.T. A versão original, em inglês, omite algumas das derivações e indica incorretamente as regras utilizadas nos passos 20 e 23.

(1) $E \rightarrow TE'$
(2) $E' \rightarrow + TE'$
(3) $E' \rightarrow \varepsilon$
(4) $T \rightarrow FT'$
(5) $T' \rightarrow *FT'$
(6) $T' \rightarrow \varepsilon$
(7) $F \rightarrow (E)$
(8) $F \rightarrow \text{id}A$
(9) $A \rightarrow \varepsilon$
(10) $A \rightarrow (E)$

A técnica acima utilizada para a remoção das recursões à esquerda presentes em uma gramática livre de contexto pode ser expressa como segue.†

Heurística 2: *Seja $A \rightarrow A\alpha_1, \ldots, A \rightarrow A\alpha_n$ e $A \rightarrow \beta_1, \ldots, A \rightarrow \beta_m$ todas as regras com A em seu lado esquerdo, onde os β_i's não começam com o símbolo A, e $m > 0$ (isto é, existe no mínimo uma regra recursiva à esquerda). Nesse caso, substituam-se essas regras por $A \rightarrow \beta_1 A', \ldots, A \rightarrow \beta_m A'$ e $A' \rightarrow \alpha_1 A', \ldots, A' \rightarrow \alpha_n A'$ e $A' \rightarrow \varepsilon$, onde A' é um novo não-terminal.*

Adicionalmente, a gramática G' de nosso exemplo apresenta regras com lados esquerdos idênticos, porém agora todas as incertezas podem ser resolvidas consultando antecipadamente o próximo símbolo de entrada. Podemos, portanto, construir o seguinte autômato de pilha determinístico M_4 que aceita $L(G)\$$.

$$M_4 = (K, \Sigma \cup \{\$\}, V', \Delta, p, \{q_\$\}),$$

onde

$$K = \{p, q, q_{\text{id}}, q_+, q_*, q_), q_(, q_\$\},$$

e Δ é relacionado abaixo.

$((p, \varepsilon, \varepsilon), (q, E))$
$((q, a, \varepsilon), (q_a, \varepsilon))$ para cada $a \in \Sigma \cup \{\$\}$
$((q_a, \varepsilon, a), (q, \varepsilon))$ para cada $a \in \Sigma$
$((q_a, \varepsilon, E), (q_a, TE'))$ para cada $a \in \Sigma \cup \{\$\}$
$((q_+, \varepsilon, E'), (q_+, +TE'))$
$((q_a, \varepsilon, E'), (q_a, \varepsilon))$ para cada $a \in \{), \$\}$
$((q_a, \varepsilon, T), (q_a, FT'))$ para cada $a \in \Sigma \cup \{\$\}$
$((q_*, \varepsilon, T'), (q_*, *FT'))$
$((q_a, \varepsilon, T) (q_a, \varepsilon))$ para cada $a \in \{+,), \$\}$
$((q_(, \varepsilon, F), (q_(, (E)))$
$((q_{\text{id}}, \varepsilon, F), (q_{\text{id}}, \text{id}A))$
$((q_(, \varepsilon, A), (q_(, (E)))$
$((q_a, \varepsilon, A), (q_a, \varepsilon))$ para cada $a \in \{+, *,), \$\}$

Então, M_4 é um analisador sintático para G'. Por exemplo, a cadeia de entrada id * (id)$ é aceita como mostrado na tabela seguinte.

† Assumimos aqui que não há regras da forma $A \rightarrow A$.

Passo	Estado	Entrada ainda não-lida	Pilha	Regra de G'
0	p	id * (id)$	ε	
1	q	id * (id)$	E	
2	q_{id}	* (id)$	E	
3	q_{id}	* (id)$	TE'	1
4	q_{id}	* (id)$	$FT'E'$	4
5	q_{id}	* (id)$	id$AT'E'$	8
6	q	* (id)$	$AT'E'$	
7	q_*	(id)$	$AT'E'$	
8	q_*	(id)$	$T'E'$	9
9	q_*	(id)$	$*FT'E'$	5
10	q	(id)$	$FT'E'$	
11	$q_($	id)$	$FT'E'$	
12	$q_($	id)$	$(E)T'E'$	7
13	q	id)$	$E)T'E'$	
14	q_{id}	)$	$E)T'E'$	
15	q_{id}	)$	$TE)T'E'$	1
16	q_{id}	)$	$FT'E)T'E'$	4
17	q_{id}	)$	id$AT'E)T'E'$	8
18	q	)$	$AT'E)T'E'$	
19	$q_)$	$	$AT'E)T'E'$	
20	$q_)$	$	$T'E)T'E'$	9
21	$q_)$	$	$E)T'E'$	6
22	$q_)$	$	$)T'E'$	3
23	q	$	$T'E'$	
24	$q_\$$	ε	$T'E'$	
25	$q_\$$	ε	E'	6
26	$q_\$$	ε	ε	3

Aqui indicamos os passos da computação nos quais um não-terminal foi substituído na pilha de acordo com uma regra de G'. Aplicando as regras de G' indicadas na última coluna dessa tabela na mesma seqüência em que aparecem, obtemos uma derivação mais à esquerda da cadeia de entrada:†

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow \text{id A } T'E' \text{ id } T'E' \Rightarrow \text{id} * FT'E' \Rightarrow \text{id} * (E)T'E' \Rightarrow$$

$$\text{id} * (TE)T'E' \Rightarrow \text{id} * (FT'E)T'E' \Rightarrow \text{id} * (\text{id A } T'E')\ T'E' \Rightarrow \text{id} * (\text{id}T'E')\ T'E' \Rightarrow$$

$$\text{id} * (\text{id}E')\ T'E' \Rightarrow \text{id} * (\text{id})\ T'E' \Rightarrow \text{id} * (\text{id})E' \Rightarrow \text{id} * (\text{id})$$

Assim, uma árvore *de análise sintática* da entrada pode ser construída (veja a Figura 3-12; o passo do autômato de pilha correspondente à expansão de cada vértice da árvore de análise sintática é também mostrado ao lado do vértice). Note-se que esse analisador sintático constrói a árvore de análise sintática da entrada partindo da raiz para as folhas, e da esquerda para a direita partindo da raiz E e repetidamente aplicando regra a apropriada, sempre substituindo o não-terminal mais à esquerda.

† N. de R.T. A versão original, em inglês, omite algumas das derivações e indica incorretamente as regras utilizadas nos passos 20 e 23.

Em geral, dada uma gramática G, pode-se tentar construir um analisador sintático descendente a parte de G como segue: eliminam-se as recursões à esquerda presentes em G, aplicando repetidamente a heurística 2 a todos os não-terminais A recursivos à esquerda em G. Aplica-se a heurística 1 para efetuar as fatorações à esquerda sempre que necessário. Então, examina-se a gramática resultante quanto à propriedade de poder decidir dentre as regras que apresentem o mesmo lado esquerdo, com o auxílio da consulta antecipada do próximo símbolo de entrada. As gramáticas que exibem essa propriedade são ditas $LL(1)$. Apesar de não termos especificado exatamente como determinar se uma gramática é de fato $LL(1)$ – nem como construir o correspondente analisador sintático determinístico caso ela de fato seja $LL(1)$ – há métodos sistemáticos para executar tal tarefa. De qualquer forma, a inspeção da gramática e um pouco de prática serão suficientes em muitos casos.

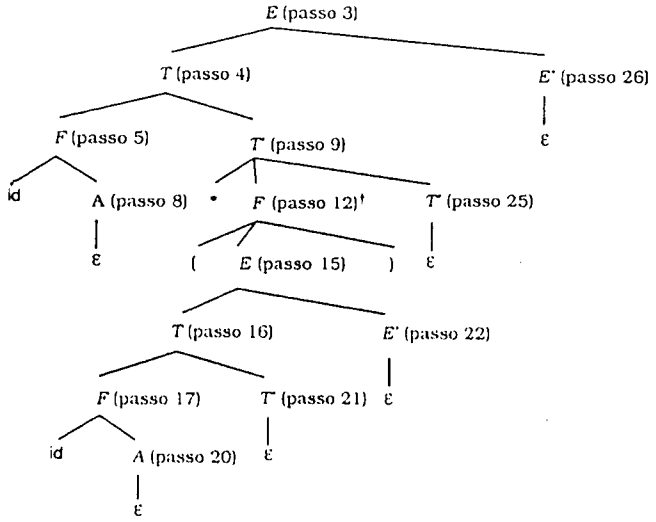


Figura 3-12

Análise sintática ascendente (*"bottom-up"*)

Não há um método de análise sintática livre de contexto que possa ser classificada como "o melhor", cada qual podendo ser mais adequado a cada uma das diferentes gramáticas. Concluímos este capítulo apresentando resumidamente algumas alternativas completamente diferentes do método apresentado da análise sintática decendente. Contudo, todos eles também se originam e se baseiam na construção de um autômato de pilha.

Existe uma técnica simétrica à empregada no Lema 3.4.1 para a construção de um autômato de pilha que aceite a linguagem gerada por uma dada gramática livre de contexto. O autômato obtido pelo método descendente opera realizando derivações mais à esquerda na pilha; à medida que

† N. de R.T. A indicação do passo 12 não consta no original.

são gerados os símbolos terminais, vão sendo comparados com a cadeia de entrada. Na técnica descrita abaixo, o autômato consulta a entrada inicialmente e, com base no símbolo de entrada encontrado, determina a derivação que deve realizar. O efeito final da aplicação desta técnica, como veremos, é o de construir uma árvore de análise sintática partindo das folhas em direção à raiz, razão pela qual essa classe de métodos é dita **ascendente**.

O autômato de pilha ascendente é construído conforme descrito a seguir. Seja $G(V, \Sigma, R, S)$ uma gramática livre de contexto; Seja $M = (K, \Sigma, s, F)$, onde $K = p, q$, $\Gamma = V$, $F = \{q\}$, e Δ constituída das seguintes transições:

(1) $((p, a, \varepsilon), (p, a))$ para cada $a \in \Sigma$.
(2) $((p, \varepsilon, \alpha^R), (p, A))$ para cada regra $A \rightarrow \alpha$ em R.
(3) $((p, \varepsilon, S), (q, \varepsilon))$.

Antes de iniciar a demonstração, comparemos essas transições com as do autômato construído segundo o método utilizado na prova do Lema 3.4.1. As transições do tipo 1 aqui empilham símbolos de entrada; transições de tipo 3 do Lema 3.4 desempilham símbolos terminais sempre que eles se identificam com os símbolos de entrada. Transições do tipo 2 substituem o lado direito de uma regra, contido na pilha, pelo correspondente lado esquerdo. O conteúdo do lado direito é encontrado ao revés na pilha; Transições do tipo 2 no Lema 3.4.1 substituem na pilha o não terminal correspondente ao lado esquerdo de uma regra pelo correspondente ao lado direito. Transições do tipo 3 aqui encerram uma computação transitando para o estado final, deixando na pilha apenas o símbolo inicial; transições de tipo 1 no Lema 3.4.1 iniciam a computação empilhando o símbolo inicial em uma pilha inicialmente vazia. Assim, a máquina ascendente opera de forma perfeitamente simétrica àquela desenvolvida no Lema 3.4.1.

Lema 3.7.1: *Sejam G e M aderentes à descrição acima. Nessas condições,* $L(M) = L(G)$.

Prova: Qualquer cadeia pertencente a $L(G)$ tem, para seu símbolo inicial, uma derivação mais à direita. Portanto, a prova da seguinte afirmação é suficiente para estabelecer o lema.

Afirmação: Para qualquer $x \in \Sigma^*$ e $\gamma \in \Gamma^*$, $(p, x, \gamma) \vdash_M^* (p, \varepsilon, S)$ se e somente se $S \overset{R}{\Rightarrow}_G \gamma^R x$.

Sendo x uma entrada de M e $\gamma = \varepsilon$, então, como q funciona apenas como estado final, podendo ser atingido apenas por meio da execução da transição 3, a afirmação implica a aceitação de x por M se e somente se G gera x. A prova da parte *somente se* da afirmação pode ser feita por indução sobre o número de passos de computação de M, enquanto a direção *se* pode ser provada por indução sobre o número de passos da derivação mais à direita de x de S. ■

Consideramos novamente a gramática de expressões aritméticas (Exemplo 3.1.3, sem a regra $F \rightarrow \mathrm{id}(E)$ da subseção anterior). As regras dessa gramática são as seguintes:

$$E \rightarrow E + T \quad (R1)$$
$$E \rightarrow T \quad (R2)$$
$$T \rightarrow T * F \quad (R3)$$
$$T \rightarrow F \quad (R4)$$
$$F \rightarrow (E) \quad (R5)$$
$$F \rightarrow \text{id} \quad (R6)$$

Aplicando-se a essa gramática o método que acabamos de apresentar, obtém-se o seguinte conjunto de transições:

$$(p, a, \varepsilon), (p, a)) \text{ para cada } a \in \Sigma \quad (\Delta 0)$$
$$(p, \varepsilon, T + E), (p, E) \quad (\Delta 1)$$
$$(p, \varepsilon, T), (p, E) \quad (\Delta 2)$$
$$(p, \varepsilon, F * T), (p, T) \quad (\Delta 3)$$
$$(p, \varepsilon, F), (p, T) \quad (\Delta 4)$$
$$(p, \varepsilon,)E(), (p, F) \quad (\Delta 5)$$
$$(p, \varepsilon, \text{id}), (p, F) \quad (\Delta 6)$$
$$(p, \varepsilon, E), (q, \varepsilon) \quad (\Delta 7)$$

Denominemos M a esse autômato de pilha. A entrada id * (id) é aceita por M, conforme mostrado na tabela abaixo:

Passo	Estado	Entrada ainda não-lida	Pilha	Transição utilizada	Regra de G
0	p	id*(id)	ε		
1	p	*(id)	id	Δ0	
2	p	*(id)	F	Δ6	R6
3	p	*(id)	T	Δ4	R4
4	p	(id)	*T	Δ0	
5	p	id)	(*T	Δ0	
6	p	)	id(*T	Δ0	
7	p	)	F(*T	Δ6	R6
8	p	)	T(*T	Δ4	R4
9	p	)	E(*T	Δ2	R2
10	p	ε	E(*T	Δ0	
11	p	ε	F*T	Δ5	R5
12	p	ε	T	Δ3	R3
13	p	ε	E	Δ2	R2
14	p	ε	ε	Δ7	

Como se pode facilmente notar, M é um dispositivo não-determinístico: transições do tipo Δ0 são compatíveis com todas as transições dos tipos Δ1 a Δ8. Além disso, sua forma geral de operação é sugestiva. Em qualquer ponto, M pode empilhar ***(shift)*** um símbolo terminal de sua cadeia de entrada (transições do tipo Δ0, utilizadas no exemplo acima de computação, nos Passos 1, 4, 5, 6 e 10). Por outro lado, M pode ocasionalmente reconhecer, no topo da pilha, alguns símbolos como sendo uma cadeia correspondente ao lado direito invertido de uma regra de G podendo, então, **reduzir** essa cadeia ao

correspondente lado esquerdo da regra (transições dos tipos Δ2 a Δ6, utilizadas na computação do exemplo, indicada na coluna "regra de G" mais à direita).

A seqüência de regras, correspondentes aos passos de redução, espelha, em ordem reversa, uma derivação mais à direita da cadeia de entrada. Em nosso exemplo, tal derivação mais à direita é a seguinte.

$$\begin{aligned} E &\Rightarrow T \\ &\Rightarrow T * F \\ &\Rightarrow T * (E) \\ &\Rightarrow T * (T) \\ &\Rightarrow T * (F) \\ &\Rightarrow T * (\text{id}) \\ &\Rightarrow F * (\text{id}) \\ &\Rightarrow \text{id} * (\text{id}) \end{aligned}$$

Essa derivação pode ser extraída, da tabela que ilustra a computação, aplicando-se as regras registradas na coluna mais à direita da tabela, iniciando na parte inferior da tabela, para a superior, sempre considerando à substituição do não-terminal mais à direita. Equivalentemente, esse processo pode ser interpretado como sendo a construção, partindo das folhas em direção à raiz e da esquerda para a direita, de uma árvore de análise sintática (isto é, exatamente simétrica à Figura 3-12(b)).

Para construirmos um analisador sintático com utilidade prática para $L(G)$, devemos converter M em um dispositivo determinístico que aceite $L(G)\$$. A exemplo do que fizemos em nosso tratamento dos analisadores sintáticos descendentes, não iremos apresentar um procedimento sistemático para essa conversão mas utilizaremos o exemplo de G, destacando as heurísticas básicas que governam o procedimento.

Inicialmente deveremos escolher um critério de decisão entre a aplicação dos dois tipos básicos de movimento, a saber: *empilhar* o próximo símbolo de entrada e *reduzir* alguns símbolos a partir do topo da pilha a um não-terminal de acordo com alguma regra da gramática. Uma possibilidade consiste em pesquisar duas partes da informação: o próximo símbolo de entrada – que chamamos b – e o símbolo do topo da pilha – que chamamos a. (O símbolo a pode ser um não-terminal.) A decisão entre empilhar e reduzir é, então, feita por consulta a uma relação $P \subseteq V \times (\Sigma \cup \{\$\})$, denominada **relação de precedências** P. Se $(a, b) \in P$, então reduzimos; caso contrário, empilhamos b. A relação correta de precedências para a gramática de nosso exemplo é dada na tabela abaixo. Intuitivamente, $(a, b) \in P$ significa que essa situação representa uma derivação mais à direita da forma

$$S \overset{R}{\Rightarrow}{}^{*}_{G} \beta A b x \overset{R}{\Rightarrow}_{G} \beta \gamma a b x.$$

Como estamos construindo derivações mais à direita (porém em sentido inverso), faz sentido desfazer a aplicação da regra $A \rightarrow \gamma a$, sempre que observarmos que um a precede imediatamente b. Há maneiras sistemáticas para efetuarmos o cálculo da relação de precedências, bem como para descobrir, como é o caso desse exemplo, em que situações uma relação de precedências é suficiente para efetuar a opção entre empilhar e reduzir. Na realidade, porém, a simples

inspeção e um pouco de prática geralmente levam à construção da tabela correta em um grande número de casos.

	(	)	id	+	*	$
(						
)		√		√	√	√
id		√		√	√	√
+						
*						
E						
T		√		√		√
F		√		√	√	√

Aqui devemos enfrentar uma nova fonte de não-determinismos: ao optar pela redução, como escolher qual dos prefixos empilhados devemos substituir por um não-terminal? Por exemplo, se a pilha contém a cadeia $F * T + E$ e a tabela indicar que devemos, efetuar uma redução, temos uma opção entre reduzir F para T (Regra R4) ou reduzir $F * T$ para T (Regra R3). Para nossa gramática, a ação correta é escolher sempre o prefixo *mais longo* do conteúdo da pilha que corresponda ao inverso do lado direito de alguma regra e substituí-lo (reduzi-lo) para o lado esquerdo dessa mesma regra. Portanto, no caso acima, devemos adotar a segunda alternativa, e reduzir $F * T$ para T.

Com essas duas regras (efetuar a redução sempre que o topo da pilha e o próximo símbolo de entrada estiverem relacionados em P, caso contrário empilhar; e, ao reduzir, reduzir o maior número possível de símbolos a partir do topo da pilha), a operação do autômato de pilha M torna-se completamente determinística. De fato, podemos projetar um autômato de pilha determinístico que incorpore essas duas regras (veja o Problema 3.7.9).

Lembramos, outra vez, que as duas heurísticas que descrevemos: – (1) utilizar uma relação de precedências para optar entre empilhar e reduzir e (2), sempre que houver escolha, efetuar a redução mais longa possível – não são eficazes em todas as situações. As gramáticas para as quais elas realmente se aplicam são chamadas **gramáticas de precedência fraca**; na prática, muitas gramáticas relacionadas às linguagens de programação são, ou podem ser facilmente convertidas em gramáticas de precedência fraca. Há muitos outros métodos, ainda mais sofisticados, para construir analisadores sintáticos descendentes, que se aplicam a classes de gramáticas mais amplas.

Problemas para a Seção 3.7

3.7.1 Mostre que as seguintes linguagens são livres de contexto determinísticas.
(a) $\{a^m b^n: m \neq n\}$
(b) $\{wcw^R: w \in \{a,b\}^*\}$
(c) $\{ca^m b^m: m \neq n\} \cup \{da^m b^{2m}: m \geq 0\}$
(d) $\{a^m cb^m: m \neq n\} \cup \{a^m db^{2m}: m \geq 0\}$

3.7.2 Mostre que a classe das linguagens livres de contexto determinísticas não é fechada em relação à operação de homomorfismo.

3.7.3 Mostre que, se L é uma linguagem livre de contexto determinística, então L não é inerentemente ambígua.

3.7.4 Mostre que o autômato de pilha M' construído na Seção 3.7.1, aceita a linguagem L, dado que M aceita $L\$$.

3.7.5 Considere a seguinte gramática livre de contexto: $G = (V, \Sigma, R, S)$, onde $V = \{(,), a, S, A\}$, $\Sigma = \{(,),a\}$ e $R = \{S \rightarrow (), S \rightarrow a, S \rightarrow (A), A \rightarrow S, A \rightarrow A.S\}$ (Para os leitores familiarizados com a linguagem de programação LISP, $L(G)$ contém todos os átomos e listas, em que o símbolo a representa qualquer átomo não-vazio.)
(a) Aplique as heurísticas 1 e 2 para G. Seja G' a gramática resultante, prove que G' é $LL(1)$. Construa um autômato de pilha M determinístico que aceite $L(G)\$$. Estude a computação de M sobre a cadeia $((()).a)$.
(b) Repita o item (a) para a gramática que se obtém substituindo-se a primeira regra de G por $A \rightarrow \varepsilon$.
(c) Repita o item (a) para a gramática que se obtém substituindo-se a última regra de G por $A \rightarrow S.A$.

3.7.6 Considere novamente a gramática G do Problema 3.7.5. Mostre que G é uma gramática de precedência fraca, com a relação de precedência mostrada abaixo. Construa a partir dessa tabela um autômato de pilha determinístico que aceite $L(G)\$$.

	a	(	)	.	$
a		√		√	√
(					
)		√		√	√
.					
A					
S		√		√	

3.7.7 Seja $G' = (V, \Sigma, R', S)$ uma gramática com as regras $S \rightarrow (A)$, $S \rightarrow a$, $A \rightarrow S.A$, $A \rightarrow \varepsilon$. G' é uma gramática de precedência fraca? Em caso afirmativo apresente uma relação de precedência adequada; caso contrário, explique por que não.

3.7.8 A aceitação pelo critério do estado final é definida no Problema 3.3.3. Mostre que L é determinístico, se, e somente se, L é aceito, por estado final, por algum autômato de pilha determinístico.

3.7.9 Forneça explicitamente um autômato de pilha determinístico que aceite a linguagem das expressões aritméticas, com base no autômato de pilha não-determinístico M e da tabela de precedência P da última subseção. Esse autômato deve efetuar consulta antecipada à cadeia de entrada, absorvendo o próximo símbolo de entrada, de maneira similar à que foi feita no autômato de pilha M_4 da seção anterior.

3.7.10 Considere as seguintes classes de linguagens:
(a) regular
(b) livre de contexto
(c) a classe dos *complementos* das linguagens livres de contexto
(d) livre de contexto deterministicas

Construa um *diagrama de Venn* para essas classes, ou seja, represente cada classe por uma "bolha", desenhadas em posições relativas adequadas de modo que as inclusões, intersecções, etc. das diversas classes sejam precisamente refletidas. Você pode dar um exemplo de linguagem que corresponde a cada região não vazia de seu diagrama?

REFERÊNCIAS

As gramáticas livres de contexto foram criadas por Noam Chomsky:

- N. Chomsky "Three models for the description of languages", *IRE Transactions on Information Theory*, 2, 3, pp. 113-124, 1956.
- N. Chomsky "On certain formal properties of grammars", *Information and Control*, 2, 137-167, 1959.

No último trabalho, a forma normal de Chomsky também foi introduzida. Uma notação, intimamente relacionada com a sintaxe das linguagens de programação, denominada BNF (sigla para Backus Normal Form ou Backus-Naur Form) também foi inventada no fim da década de 1950:

- P. Naur, ed. "Revised report on the algorithmic language Algol 60", *Communications of the ACM*, 6, 1, pp. 1-17, 1963, reprinted in S. Rosen, ed., *Programming Systems and Languages* New York: McGraw-Hill, pp. 79-118, 1967.

O Problema 3.1.9 sobre a equivalência entre gramáticas regulares e autômatos finitos foi extraído do:

- N. Chomsky, G. A. Miller "Finite-state languages", *Information and Control*, 1, pp. 91-112, 1958.

O autômato de pilha foi introduzido em:

- A. G. Oettinger "Automatic syntactic analysis and the pushdown store", *Proceedings of Symposia on Applied Mathematics, Vol.* 12, Providence, R.I.: American Mathematical Society, 1961.

O Teorema 3.4.1 sobre a equivalência entre linguagens livres de contexto e autômatos de pilha foi provado independentemente por Schutzenberger, Chomsky e Evey:

- M. P. Schutzenberger "On context-free languages and pushdown automata", *Information and Control*, 6, 3, pp. 246-264, 1963.
- N. Chomsky "Context-free grammar and a pushdown storage", Quarterly Progress Report, 65, pp. 187-194, M.I.T. Research Laboratory in Electronics, Cambridge, Mass., 1962
- J. Evey "Application of a pushdown store machines", *Proceedings of the 1963 Fall Joint Computer Conference*, pp. 215-217, Montreal: AFIPS Press, 1963.

As propriedades de fechamento apresentadas na subseção 3.5.1, juntamente com muitas outras, foram extraídas da:

- V. Bar-Hillel, M. Perles & E. Shamir "On formal properties of simple phrase structure grammars", *Zeitschrift Ar Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14, pp. 143-172, 1961.

No mesmo trabalho, encontra-se a versão mais poderosa do Teorema 3.5.3 (o Teorema de bombeamento para gramáticas livres de contexto; veja também o Problema 3.5.7). Uma versão ainda mais poderosa desse teorema aparece em:

- W. G. Ogden "A helpful result for proving inherent ambiguity", *Mathematical Systems Theory*, 2, pp. 191-194, 1968.

O algoritmo de programação dinâmica para o reconhecimento de linguagens livres de contexto foi descoberto independentemente por:

- T. Kasami "An efficient recognition and syntax algorithm for context-free languages", Report AFCRL-65-758 (1965), Air Force Cambridge Research Laboratory, Cambridge, Mass.
- D. H. Younger "Recognition and parsing of context-free languages in time n^3," *Information and Control*, 10, 2, pp. 189-208, 1967.

Uma variante desse algoritmo é mais rápida quando a gramática subjacente é não-ambígua

- J. Earley "An efficient context-free parsing algorithm", *Communications of the ACM*, 13, pp. 94-102, 1970.

O mais eficiente algoritmo geral de reconhecimento que se conhece é creditado a Valiant. Ele é executado em tempo proporcional necessário para se multiplicar duas matrizes $n \times n$, ou seja $O(n^{2.3...})$.

- L. G. Valiant "General context-free recognition in less than cubic time", *Journal of Computer and Systems Sciences*, 10, 2, pp. 308-315, 1975.

Os analisadores sintáticos LL(1) foram introduzidos em:

- P. M. Lewis II, R. E. Stearns "Syntax-directed transduction", *Journal of the ACM*, 15, 3, pp. 465-488, 1968.
- D. E. Knuth "Top-down syntax analysis", *Acta Informatica*, 1, 2, pp. 79-110, 1971.

Os analisadores sintáticos de precedência fraca foram propostos em:

- J. D. Ichbiah and S. P. Morse "A technique for generating almost optimal Floyd Evans productions for precedence grammars", *Communications of the ACM*, 13, 8, pp. 501-508, 1970.

O seguinte é um livro clássico sobre compiladores:

- A. V. Aho, R. Sethi, J. D. Ullman *Principles of Compiler Design*, Reading, Mass.: Addison-Wesley, 1985.

A Ambigüidade e a ambigüidade inerente foram estudadas em primeiro lugar em:

- N. Chomsky and M. P. Schutzenberger "The algebraic theory of context-free languages", in *Computer Programming and Formal Systems (pp.* 118-161), ed. P. Braffort, D. Hirschberg, Amsterdam: North Holland, 1963.
- S. Ginsburg and J. S. Ullian "Preservation of unambiguity and inherent ambiguity in context-free languages", *Journal of the ACM*, 13, 1, pp. 62-88, 1966.

A forma normal de Greibach (Problema 3.5) encontra-se em:

- S. Greibach "A new normal form theorem for context-free phrase structure grammars", *Journal of the ACM*, 12, 1, pp. 42-52, 1965.

Dois livros avançados sobre linguagens livres de contexto são:

- S. Ginsburg The Mathematical Theory of Context-free Languages, New York: McGraw-Hill, 1966.
- M. A. Harrison *Introduction to Formal Language Theory*, Reading, Massach.: Addison-Wesley, 1978.