Capítulo 2

Linguagens Livres-do-Contexto

No Capítulo 1 introduzimos dois métodos diferentes, embora equivalentes, de descrever linguagens: ***autômatos finitos*** e ***expressões regulares***. Mostramos que muitas linguagens podem ser descritas dessa maneira mas que algumas linguagens simples, tais como $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$, não podem.

Neste capítulo introduzimos ***gramáticas livres-do-contexto***, um método mais poderoso de descrever linguagens. Tais gramáticas podem descrever certas características que têm uma estrutura recursiva o que as torna úteis em uma variedade de aplicações.

Gramáticas livres-do-contexto foram primeiramente usadas no estudo de linguagens humanas. Uma maneira de entender o relacionamento de termos como *nome*, *verbo*, e *preposição* e suas respectivas frases leva a uma recursão natural porque frases nominais podem aparecer dentro de frases verbais e vice-versa. Gramáticas livres-do-contexto podem capturar aspectos importantes desses relacionamentos.

Uma aplicação importante de gramáticas livres-de-contexto ocorre na especificação e compilação de linguagens de programação. Uma gramática para uma linguagem de programação frequentemente aparece como uma referência para pessoas que estão tentando aprender a sintaxe da linguagem. Projetistas de compiladores e interpretadores para linguagens de programação frequemente começam obtendo uma gramática para a linguagem. A maioria dos compiladores e interpretadores contêm uma componente chamada de ***analisador*** que extrai o significado de um programa antes de gerar o código compilado ou realizar a execução interpretada. Um número de metodologias facilitam a construção de um analisador uma vez que uma gramática livre-do-contexto está disponível. Algumas ferramentas até geram automaticamente o analisador a partir da gramática.

A coleção de linguagens associadas a gramáticas livres-do-contexto são chamadas as ***linguagens livres-do-contexto***. Elas incluem todas as linguagens regulares e muitas linguagens adicionais. Neste capítulo, damos uma definição formal de gramáticas livres-do-contexto e estudamos as propriedades de linguagens livres-do-contexto. Também introduzimos ***autômatos a pilha***, uma classe de máquinas que reconhece as linguagens livres-do-contexto. Autômatos a pilha são úteis porque eles nos permitem aumentar nosso conhecimento sobre o poder de gramáticas livres-do-contexto.

2.1 Gramáticas livres-do-contexto

O que segue é um exemplo de uma gramática livre-do-contexto, que chamaremos G_1.

$$\begin{array}{lcl} A & \to & \texttt{0}A\texttt{1} \\ A & \to & B \\ B & \to & \# \end{array}$$

Uma gramática consiste de uma coleção de ***regras de substituição***, também chamadas *produções*. Cada regra aparece como uma linha na gramática e compreende um símbolo e uma cadeia, separados por uma seta. O símbolo é chamado uma ***variável***. A cadeia consiste de variáveis e outros símbolos chamados ***terminais***. Os símbolos variáveis são frequentemente representados por letras maiúsculas. Os terminais são análogos ao alfabeto de entrada e frequentemente são representados por letras minúsculas, números, ou símbolos especiais. Uma variável é designada a ***variável inicial***. Ela usualmente ocorre no lado esquerdo de uma das primeiras regras da gramática. Por exemplo, a gramática G_1 contém três regras. As variáveis de G_1 são A e B, onde A é a variável inicial. Seus terminais são `0`, `1`, e #.

Você usa uma gramática para descrever uma linguagem gerando cada cadeia daquela linguagem da seguinte maneira:

1. Escreva a variável inicial. Ela é a variável no lado esquerdo da primeira regra, a menos que seja especificado o contrário.
2. Encontre uma variável que está escrita e uma regra que começa com aquela variável. Substitua a variável escrita pelo lado direito daquela regra.
3. Repita o passo 2 até que nenhuma variável permaneça.

Por exemplo, a gramática G_1 gera a cadeia `000#111`. A seqüência de substituições para obter uma cadeia é chamada uma ***derivação***. Uma derivação da cadeia `000#111` na gramática G_1 é

$$A \Rightarrow \texttt{0}A\texttt{1} \Rightarrow \texttt{00}A\texttt{11} \Rightarrow \texttt{000}A\texttt{111} \Rightarrow \texttt{000}B\texttt{111} \Rightarrow \texttt{000\#111}$$

Você pode também representar a mesma informação de uma maneira mais pictórica usando uma ***árvore sintática***. Um exemplo de uma árvore sintática aparece na Figura 2.1.

Figura 2.1: Árvore sintática para `000#111` na gramática G_1

Todas as cadeias geradas dessa maneira constituem a ***linguagem da gramática***. Escrevemos $L(G_1)$ para a linguagem da gramática G_1. Alguma experimentação com a gramática G_1 nos mostra que $L(G_1)$ é $\{0^n\#1^n \mid n \geq 0\}$. Qualquer linguagem que pode ser gerada por alguma gramática livre-do-contexto é chamada uma ***linguagem livre-do-contexto*** (LLC). Por conveniência ao apresentar uma gramática livre-do-contexto, abreviamos várias regras com a mesma variável do lado esquerdo, tais como $A \to \texttt{0}A\texttt{1}$ e $A \to B$, em uma única linha $A \to \texttt{0}A\texttt{1} \mid B$, usando o símbolo "|"como um "ou."

O que segue é um segundo exemplo de uma gramática livre-do-contexto chamada G_2, que descreve um fragmento da língua inglesa.

$$
\begin{array}{rcl}
\langle \text{SENTENCE} \rangle & \to & \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB- PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle & \to & \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX- NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle & \to & \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX- VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle & \to & \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle & \to & \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle & \to & \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle & \to & \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle & \to & \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle & \to & \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle & \to & \texttt{with}
\end{array}
$$

A gramática G_2 tem dez variáveis (os termos gramaticais em maiúsculas escritos entre colchetes); 27 terminais (o alfabeto inglês padrão mais um caracter de espaço); e dezoito regras. Cadeias em $L(G_2)$ incluem os três exemplos seguintes.

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

Cada uma dessas cadeias tem uma derivação na gramática G_2. O que segue é uma derivação da primeira cadeia nessa lista.

$$
\begin{array}{rl}
\langle \text{SENTENCE} \rangle & \Rightarrow \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB- PHRASE} \rangle \\
& \Rightarrow \langle \text{CMPLX-NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
& \Rightarrow \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
& \Rightarrow \texttt{a}\ \langle \text{NOUN} \rangle \langle \text{VERB-PHRASE} \rangle \\
& \Rightarrow \texttt{a boy}\ \langle \text{VERB-PHRASE} \rangle \\
& \Rightarrow \texttt{a boy}\ \langle \text{CMPLX-VERB} \rangle \\
& \Rightarrow \texttt{a boy}\ \langle \text{VERB} \rangle \\
& \Rightarrow \texttt{a boy sees}
\end{array}
$$

Definição formal de uma gramática livre-do-contexto

Vamos formalizar nossa noção de uma gramática livre-do-contexto (LLC).

Definição 2.1 . . . . . .
Uma ***gramática livre-do-contexto*** é uma 4-upla (V, Σ, R, S), onde

1. V é um conjunto finito chamado as ***variáveis***,
2. Σ é um conjunto finito, disjunto de V, chamado os ***terminais***,
3. R é um conjunto finito de ***regras***, com cada regra sendo uma variável e uma cadeia de variáveis e terminais, e
4. $S \in V$ é o símbolo inicial.

Se u, v, e w são cadeias de variáveis e terminais, e $A \to w$ é uma regra da gramática, dizemos que uAv ***produz*** uwv, escrito $A \Rightarrow uwv$. Escreva $u \overset{*}{\Rightarrow} v$ se $u = v$ ou se uma seqüência $u_1, u_2, \ldots, u_k$ existe para $k \geq 0$ e

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

A ***linguagem da gramática*** é $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

Na gramática G_1, $V = \{A, B\}$, $\Sigma = \{\mathtt{0}, \mathtt{1}, \#\}$, $S = A$, e R é a coleção de três regras que aparecem na página ??. Na gramática G_2,

$$\begin{aligned} V = \quad & \{\langle\text{SENTENCE}\rangle, \langle\text{NOUN-PHRASE}\rangle, \langle\text{VERB- PHRASE}\rangle, \\ & \langle\text{PREP-PHRASE}\rangle, \langle\text{CMPLX-NOUN}\rangle, \langle\text{CMPLX- VERB}\rangle, \\ & \langle\text{ARTICLE}\rangle, \langle\text{NOUN}\rangle, \langle\text{VERB}\rangle, \langle\text{PREP}\rangle\}. \end{aligned}$$

e $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}, \text{“ ”}\}$. O símbolo “ ”é o símbolo branco, colocado invisivelmente entre cada palavra (a, boy, etc.), de forma que palavras não entram umas pelas outras.

Frequentemente especificamos uma gramática escrevendo apenas suas regras. Podemos identificar as variáveis como os símbolos que aparecem no lado esquerdo das regras e os terminais como os símbolos remanescentes. Por convenção, a variável inicial é a variável no lado esquerdo da primeira regra.

Exemplo 2.2 .....
Considere a gramática $G_2 = (\{S\}, \{\mathtt{a}, \mathtt{b}\}, R, S)$. O conjunto de regras, R, é

$$S \to \mathtt{a}S\mathtt{b} \mid SS \mid \varepsilon$$

Essa gramática gera cadeias tais como abab, aaabbb, aababb. Você pode ver mais facilmente o que essa linguagem é se você pensar em a como sendo um abre-parênteses “(”e b como um fecha-parênteses “)”. Vista dessa maneira, $L(G_3)$ é a linguagem de todas as cadeias de parênteses apropriadamente aninhados.

Exemplo 2.3 .....
Considere a gramática $G_4 = (V, \Sigma, R, \langle\text{EXPR}\rangle)$.

V é $\{\langle\text{EXPR}\rangle, \langle\text{TERMO}\rangle, \langle\text{FATOR}\rangle\}$ e Σ é $\{\mathtt{a}, +, \times, (,)\}$. As regras são

$$\begin{aligned} \langle\text{EXPR}\rangle &\to \langle\text{EXPR}\rangle + \langle\text{TERMO}\rangle \mid \langle\text{TERMO}\rangle \\ \langle\text{TERMO}\rangle &\to \langle\text{TERMO}\rangle \times \langle\text{FATOR}\rangle \mid \langle\text{FATOR}\rangle \\ \langle\text{FATOR}\rangle &\to (\ \langle\text{EXPR}\rangle\ ) \mid \mathtt{a} \end{aligned}$$

As duas cadeias a+a×a e (a+a)×a podem ser geradas com a gramática G_4. As árvores sintáticas são mostradas na figura a seguir.

Figura 2.2: Árvores sintáticas para as cadeias a+a×a e (a+a)×a

Um compilador traduz código escrito numa linguagem de programação para uma outra forma, usualmente uma forma mais adequada para execução. Para fazer isso o compilador extrai o significado do código para ser compilado em um processo chamado ***parsing***. Uma representação desse significado é a árvore sintática para o código, no contexto da gramática livre-do-contexto para a linguagem de programação. Discutimos um algoritmo que analisa linguagens livres-do-contexto mais adiante no Teorema 7.14 e no Problema 7.38.

A gramática G_4 descreve um fragmento de uma linguagem de programação que lida com expressões aritméticas. Observe como as árvores sintáticas da Figura 2.2 “agrupam” as operações. A árvore para a+a×a agrupa o operador × e seus operandos (os dois segundos a’s) juntos como um operando do operador +. Na árvore para

`(a+a)×a`, o agrupamento é revertido. Esses agrupamentos estão de acordo com a precedência padrão da multiplicação antes da adição e o uso de parênteses para sobrepor a precedência padrão. A gramática G_4 é projetada para capturar essas relações de precedência.

Projetando gramáticas livres-do-contexto

Como no projeto de autômatos finitos, discutido na página 41 da Seção 1.1, o projeto de gramáticas livres-do-contexto requer criatividade. De fato, gramáticas livres-do-contexto são até mais complicadas de construir que autômatos finitos porque estamos mais acostumados a programar uma máquina para tarefas específicas do que a descrever linguagens com gramáticas. As seguintes técnicas são úteis, em separado ou em combinação, quando você se depara com o problema de construir uma GLC.

Primeiro, muitas GLC's são a união de GLC's mais simples. Se você tem que construir uma GLC para uma LLC que você pode partir em partes mais simples, faça isso e aí construa gramáticas separadas para cada parte. Essas gramáticas separadas podem ser facilmente combinadas em uma gramática para a linguagem original juntando todas as regras e variáveis iniciais das gramáticas separadas. Resolver vários problemas mais simples é frequentemente mais fácil que resolver um problema complicado.

Por exemplo, para obter uma gramática para a linguagem $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\} \cup \{\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$, primeiro construa a gramática

$$S_1 \to \mathtt{0}S_1\mathtt{1} \mid \varepsilon$$

para a linguagem $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ e a gramática

$$S_2 \to \mathtt{1}S_2\mathtt{0} \mid \varepsilon$$

para a linguagem $\{\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$ e aí adicione as regras $S \to S_1 \mid S_2$ para dar a gramática

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to \mathtt{0}S_1\mathtt{1} \mid \varepsilon \\ S_2 &\to \mathtt{1}S_2\mathtt{0} \mid \varepsilon \end{aligned}$$

Segundo, construir uma GLC para uma linguagem que acontece de ser regular é fácil se você puder primeiro construir um AFD para aquela linguagem. Você pode converter um AFD em uma GLC equivalente da seguinte forma. Crie uma variável R_i para cada estado q_i do AFD. Adicione a regra $R_i \to aR_j$ à GLC se $\delta(q_i, a) = q_j$ é uma transição do AFD. Adicione a regra $R_i \to \varepsilon$ se q_i é um estado de aceitação do AFD. Crie a variável inicial R_0 da gramática, onde q_0 é o estado inicial da máquina. Verifique você mesmo que a GLC resultante gera a mesma linguagem que o AFD reconhece.

Terceiro, certas linguagens livres-do-contexto contêm cadeias com duas subcadeias que são "ligadas" no sentido de que uma máquina para uma tal linguagem necessitaria de memorizar uma quantidade ilimitada de informação sobre uma das subcadeias para verificar que ela corresponde propriamente à outra subcadeia. Essa situação ocorre na linguagem $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ porque uma máquina precisaria memorizar o número de `0`'s de modo a verificar que ele é igual ao número de `1`'s. Você pode construir uma GLC para lidar com essa situação usando uma regra da forma $R \to uRv$, que gera cadeias nas quais a parte contendo as u's corresponde à parte contendo as v's.

Finalmente, em linguagens mais complexas, as cadeias podem conter certas estruturas que aparecem recursivamente como parte de outras (ou as mesmas) estruturas.

Essa situação ocorre na gramática que gera expressões aritméticas no Exemplo 2.3. Em qualquer hora que o símbolo a aparece, um expressão parentizada inteira pode aparecer recursivamente ao invés do a. Para obter esse efeito, coloque a símbolo de variável gerando a estrutura no local das regras correspondendo a onde aquela estrutura pode recursivamente aparecer.

Ambigüidade

Às vezes uma gramática pode gerar a mesma cadeia de diversas maneiras diferentes. Tal cadeia terá árvores sintáticas diferentes e por conseguinte vários significados diferentes. Esse resultado pode ser indesejável para certas aplicações, tais como linguagens de programação, onde um dado programa deve ter uma única interpretação.

Se uma gramática gera a mesma cadeia de várias maneiras diferentes, dizemos que a cadeia é derivada *ambiguamente* naquela gramática. Se uma gramática gera alguma cadeia ambiguamente dizemos que a gramática é *ambígua*.

Por exemplo, vamos considerar a gramática G_5:

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \texttt{a}$$

Essa gramática gera a cadeia a+a×a ambiguamente. A seguinte figura mostra as duas árvores sintáticas.

Figura 2.3: As duas árvores sintáticas para a cadeia a+a×a

Essa gramática não captura as relações de precedência usuais e portanto pode agrupar o $+$ antes do $\times$ ou vice-versa. Diferentemente, a gramática G_4 gera exatamente a mesma linguagem, mas toda cadeia gerada tem uma única árvore sintática. Portanto G_4 é não-ambígua, enquanto que G_5 é ambígua.

A gramática G_2 da página 93 é um outro exemplo de uma gramática ambígua. A sentença `the girl touches the boy with the flower` tem duas derivações diferentes. No Exercício 2.8 pede-se para dar as duas árvores sintáticas e observar sua correspondência com as duas maneiras diferentes de ler aquela sentença.

Agora formalizamos a noção de ambigüidade. Quando dizemos que uma gramática gera uma cadeia ambiguamente, queremos dizer que a cadeia tem duas árvores sintáticas diferentes, e não duas derivações diferentes. Duas derivações podem diferir meramente na ordem em que elas substituem variáveis e mesmo assim não diferir na sua estrutural global. Para nos concentrarmos sobre a estrutura definimos um tipo de derivação que substitui variáveis numa ordem fixada. Uma derivação de uma cadeia w em uma gramática G é uma ***derivação à esquerda*** se a cada passo a variável mais à esquerda remanescente é aquela a ser substituída. A derivação na página 94 é uma derivação à esquerda.

Definição 2.4 .....

Uma cadeia w é derivada ***ambiguamente*** na gramática livre-do-contexto G se ela tem duas ou mais derivações à esquerda. A gramática G é ***ambígua*** se ela gera alguma cadeia ambiguamente.

Às vezes quando temos uma gramática ambígua podemos encontrar uma gramática não-ambígua que gera a mesma linguagem. Algumas linguagens livres-do-contexto, entretanto, somente podem ser geradas por gramáticas ambíguas. Tais linguagens são chamadas ***inerentemente ambíguas***. O Problema 2.24 pede para você provar que a linguagem $\{\mathtt{0}^i\mathtt{1}^j\mathtt{2}^k \mid i = j \text{ ou } j = k\}$ é inerentemente ambígua.

Forma normal de Chomksy

Quando se trabalha com gramáticas livres-do-contexto, é frequentemente conveniente se tê-las em forma simplificada. Uma das formas mais simples e mais úteis é chamada a forma normal de Chomsky. Vamos achar a forma normal de Chomsky útil quando estivermos dando algoritmos para trabalhar com gramáticas livres-do-contexto nos Capítulos 4 e 7.

Definição 2.5 .....
Uma gramática livre-do-contexto está na ***forma normal de Chomsky*** se toda regra está na forma

$$A \to BC$$
$$A \to a$$

onde a é qualquer terminal e A, B, e C são quaisquer variáveis—exceto que B e C não podem ser a variável inicial. Adicionalmente permitimos a regra $S \to \varepsilon$, onde S é a variável inicial.

Teorema 2.6 .....
Qualquer linguagem livre-do-contexto é gerada por uma gramática na forma normal de Chomsky.

.....
Idéia da prova. Podemos converter qualquer gramática G na forma normal de Chomsky. A conversão tem diversos estágios nos quais as regras que violam as condições são substituídas por regras equivalentes que são satisfatórias. Primeiro, adicionamos um novo símbolo inicial. Então, eliminamos todas as ***regras*** ε da forma $A \to \varepsilon$. Também eliminamos todas as ***regras unitárias*** da forma $A \to B$. Em ambos os casos a gramática é então consertada para garantir que ela ainda gera a mesma linguagem. Finalmente, convertemos as regras remanescentes na forma apropriada.

Prova. Primeiro, adicionamos um novo símbolo inicial S_0 e a regra $S_0 \to S$, onde S era o símbolo inicial original. Essa mudança garante que o símbolo inicial não ocorre no lado direito de uma regra.

Segundo, cuidamos de todas as regras ε. Removemos uma ε-regra $A \to \varepsilon$, onde A não é o símbolo inicial. Então para cada ocorrência de um A no lado direito de uma regra, adicionamos uma nova regra com aquela ocorrência apagada. Em outras palavras, se $R \to uAv$ é uma regra na qual u e v são cadeias de variáveis e terminais, adicionamos a regra $R \to uv$. Fazemos isso para cada *ocorrência* de um A, de modo que a regra $R \to uAvAw$ nos leve a adicionar $R \to uvAw$, $R \to uAvw$, e $R \to uvw$. Se tivermos a regra $R \to A$, adicionamos $R \to \varepsilon$ a menos que tivéssemos removido previamente a regra $R \to \varepsilon$. Repetimos esses passos até que eliminemos todas as regras ε que não envolvem a variável inicial.

Terceiro, manuseamos as regras unitárias. Removemos uma regra unitária $A \to B$. Então, sempre que uma regra $B \to u$ aparece, adicionamos a regra $A \to u$ a menos

que essa fosse uma regra unitária previamente removida. Como antes, u é uma cadeia de variáveis e terminais. Repetimos esses passos até que eliminemos todas as regras unitárias.

Finalmente, convertemos todas as regras remanescentes para a forma apropriada. Substituimos cada regra $A \to u_1u_2\cdots u_k$ onde $k \geq 3$ e cada u_i é uma variável ou um símbolo terminal, com as regras $A \to u_1A_1$, $A_1 \to u_2A_2$, $A_2 \to u_3A_3$, ..., $A_{k-2} \to u_{k-1}u_k$. Os A_i's são novas variáveis. Se $k \geq 2$, substituimos qualquer terminal u_i na(s) regra(s) precedente(s) por novas variáveis U_i e adicionamos a regra $U_i \to u_i$.

. . . . . .

Exemplo 2.7 . . . . . .

Suponha que G_6 seja a GLC a seguir e converta-a para a forma normal de Chomsky usando o procedimento de conversão que acaba de ser dado. A seguinte série de gramáticas ilustra os passos na conversão. As regras sublinhadas acabaram de ser adicionadas. As regras mostradas em $\square$ acabaram de ser removidas.

1. A GLC G_6 original é mostrada à esquerda. O resultado de se aplicar o primeiro passo para acrescentar um novo símbolo original aparece à direita.

	$\mathbf{S_0 \to S}$
$S \to ASA \mid \mathsf{a}B$	$S \to ASA \mid \mathsf{a}B$
$A \to B \mid S$	$A \to B \mid S$
$B \to \mathsf{b} \mid \varepsilon$	$B \to \mathsf{b} \mid \varepsilon$

2. Remova as regras ε, mostradas à esquerda, e $A \to \varepsilon$, mostrada à direita.

$S_0 \to S$	$S_0 \to S$
$S \to ASA \mid \mathsf{a}B \mid \mathsf{a}$	$S \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid \mathbf{SA} \mid \mathbf{AS} \mid \mathbf{S}$
$A \to B \mid S \mid \varepsilon$	$A \to B \mid S \mid \boxed{\varepsilon}$
$B \to \mathsf{b} \mid \boxed{\varepsilon}$	$B \to \mathsf{b}$

3a. Remova as regras unitárias $S \to S$, mostradas à esquerda, e $S_0 \to S$, mostrada à direita.

$S_0 \to S$	$S_0 \to \boxed{S} \mid \mathbf{ASA} \mid \mathsf{a}\mathbf{B} \mid \mathsf{a} \mid \mathbf{SA} \mid \mathbf{AS}$
$S \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid SA \mid AS \mid \boxed{S}$	$S \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid \mathbf{SA} \mid \mathbf{AS}$
$A \to B \mid S$	$A \to B \mid S$
$B \to \mathsf{b}$	$B \to \mathsf{b}$

3b. Remova as regras unitárias $A \to B$ e $A \to S$.

$S_0 \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid SA \mid AS$	$S_0 \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid SA \mid AS$
$S \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid SA \mid AS$	$S \to ASA \mid \mathsf{a}B \mid \mathsf{a} \mid SA \mid AS$
$A \to B \mid S \mid \mathsf{b}$	$A \to S \mid \mathsf{b} \mid \mathbf{ASA} \mid \mathsf{a}\mathbf{B} \mid \mathsf{a} \mid \mathbf{SA} \mid \mathbf{AS}$
$B \to \mathsf{b}$	$B \to \mathsf{b}$

4. Converta as regras remanescentes na forma apropriada acrescentando variáveis e regras adicionais. A gramática final na forma normal de Chomsky é equivalente a G_6 e aparece a seguir. (Na verdade o procedimento dado no Teorema 2.6 produz diversas variáveis U_i juntamente com várias regras $U_i \rightarrow \mathsf{a}$. Simplificamos a gramática resultante usando uma única variável U e uma única regra $U \rightarrow \mathsf{a}$.)

$$
\begin{array}{rcl}
S_0 & \rightarrow & AA_1 \mid UB \mid \mathsf{a} \mid SA \mid AS \\
S & \rightarrow & AA_1 \mid UB \mid \mathsf{a} \mid SA \mid AS \\
A & \rightarrow & \mathsf{b} \mid AA_1 \mid UB \mid \mathsf{a} \mid SA \mid AS \\
A_1 & \rightarrow & SA \\
U & \rightarrow & \mathsf{a} \\
B & \rightarrow & \mathsf{b}
\end{array}
$$

2.2 Autômatos a pilha . . . . . .

Nesta seção introduzimos um novo tipo de modelo computacional chamado ***autômato a pilha***. Esses autômatos são como autômatos finitos não-determinísticos mas têm um componente extra chamado uma ***pilha***. A pilha provê memória adicional além do autômato finito disponível no controle. A pilha permite autômatos a pilha reconhecerem algumas linguagens não-regulares.

Autômatos a pilha são equivalentes em poder a gramáticas livres-do-contexto. Essa equivalência é útil porque ela nos dá duas opções para provar que uma linguagem é livre-do-contexto. Podemos ou dar uma gramática livre-do-contexto que a gera ou um autômato a pilha que a reconhece. Certas linguagens são mais facilmente descritas em termos de geradores, enquanto que outras são mais facilmente descritas em termos de reconhecedores.

A figura a seguir é uma representação esquemática de um autômato finito. O controle representa os estados e a função de transição, a fita contém a cadeia de entrada, e a seta representa a cabeça de entrada, apontando para o próximo símbolo de entrada a ser lido.

Figura 2.4: Esquemática de um autômato finito

Com a adição de um componente pilha obtemos uma representação esquemática de um autômato a pilha, como mostrado na figura a seguir.

Figura 2.5: Esquemática de um autômato a pilha

Um autômato a pilha (AP) pode escrever símbolos na pilha e lê-los de volta mais adiante. Escrever um símbolo "empilha" todos os outros símbolos sobre a pilha. A qualquer tempo o símbolo no topo da pilha pode ser lido e removido. Os símbolos remanescentes então sobem. O ato de escrever um símbolo na pilha é frequentemente referido como ***empilhar*** o símbolo, e o ato de remover um símbolo é referido como

desempilhá-lo. Note que todo acesso à pilha, tanto para ler quanto para escrever, pode ser feito somente no topo. Em outras palavras, uma pilha é um dispositivo de armazenamento "último que entra, primeiro que sai". Se uma certa informação for escrita na pilha e informação adicional for escrita depois, a informação mais antiga torna-se inacessível até que a informação mais nova seja removida.

Os pratos num balcão de servir de uma cafeteria ilustram uma pilha. A pilha de pratos repousa sobre uma fonte de modo que quando um novo prato é colocado sobre a pilha, os pratos abaixo dele descem. A pilha em um autômato a pilha é como uma pilha de pratos, com cada prato tendo um símbolo escrito nele.

Uma pilha é valorosa porque ela pode armazenar uma quantidade ilimitada de informação. Lembre-se que um autômato finito é incapaz de reconhecer a linguagem $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ porque ele não pode armazenar números muito grandes de `0`'s que ele tenha visto. Por conseguinte a natureza ilimitada de uma pilha permite ao AP armazenar números de tamanho ilimitado. A seguinte descrição informal mostra como o autômato para essa linguagem funciona.

> Leia símbolos da entrada. À medida que cada `0` é lido, empilhe-o na pilha. Assim que `1`'s são vistos, desempilhe um `0` da pilha para cada `1` lido. Se a leitura da entrada termina exatamente quanto a pilha fica vazia de `0`'s, aceite a entrada. Se a pilha fica vazia enquanto `1`'s permanecem ou se os `1`'s terminam enquanto a pilha ainda contém `0`'s ou se quaisquer `0`'s aparecem na entrada seguindo `1`'s, rejeite a entrada.

Como mencionado anteriormente, autômatos a pilha podem ser não-determinísticos. Essa característica é crucial porque, ao contrário da situação dos autômatos finitos, não-determinismo adiciona poder à capacidade que autômatos a pilha teriam se lhes permitissem somente ser determinísticos. Algumas linguagens, tais como $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$, não requerem não-determinismo, mas outras sim. Damos uma linguagem que requer não-determinismo no Exemplo 2.11.

Definição formal de um autômato a pilha

A definição formal de um autômato a pilha é semelhante àquela de um autômato finito exceto pela pilha. A pilha é um dispositivo contendo símbolos tirados de algum alfabeto. A máquina pode usar alfabetos diferentes para sua entrada e sua pilha, portanto agora especificamos tanto o alfabeto de entrada Σ quanto o alfabeto da pilha Γ.

No coração de qualquer definição formal de um autômato está a função de transição, pois ela descreve seu comportamento. Lembre-se que $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. O domínio da função de transição é $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Por conseguinte o estado atual, o próximo símbolo de entrada a ser lido, e o símbolo no topo da pilha determinam o próximo movimento de um autômato a pilha. O símbolo pode ser ε causando a máquina a mover sem ler um símbolo da entrada, ou a mover sem ler um símbolo da pilha.

Para o codomínio da função de transição precisamos considerar o que permite ao autômato fazer quando ele está numa dada situação. Ele pode entrar em algum novo estado e possivelmente escrever um símbolo no topo da pilha. A função δ pode indicar essa ação retornando um membro de Q juntamente com um membro de Γ_ε, ou seja, um membro de $Q \times \Gamma_\varepsilon$. Devido ao fato de que permitimos não-determinismo nesse modelo, uma situação pode ter várias movimentos próximos legais. A função de transição incorpora não-determinismo da maneira usual, retornando um conjunto de membros de $Q \times \Gamma_\varepsilon$, ou seja, um membro de $\mathcal{P}(Q \times \Gamma_\varepsilon)$. Colocando tudo junto, nossa função de transição δ toma a forma $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

Definição 2.8 . . . . . .
Um ***autômato a pilha*** é uma 6-upla $(Q, \Sigma, \Gamma, \delta, q_0, F)$, onde Q, Σ, Γ, e F são todos conjuntos finitos, e

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

Um autômato a pilha $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computa da seguinte forma. Ele aceita a entrada w se w pode ser escrita como $w = w_1 w_2 \cdots w_n$, onde cada $w_i \in \Sigma_\varepsilon$ e seqüências de estados $r_0, r_1, \ldots, r_m \in Q$ e cadeias $s_0, s_1, \ldots, s_m \in \Gamma^*$ existem que satisfazem as três condições abaixo. As cadeias s_i representam a seqüência de conteúdos da pilha que M tem no ramo de aceitação da computação.

1. $r_0 = q_0$ e $s_0 = \varepsilon$. Esta condição significa que M começa de forma apropriada, no estado inicial e com a pilha vazia.
2. Para $i = 0, \ldots, m-1$, temos $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, onde $s_i = at$ e $s_{i+1} = bt$ para algum $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$. Esta condição enuncia que M move apropriadamente conforme o estado, a pilha, e o próximo símbolo de entrada.
3. $r_m \in F$. Essa condição enuncia que um estado de aceitação ocorre no final da entrada.

Exemplos de autômatos a pilha

Exemplo 2.9 . . . . . .
O que segue é uma descrição formal do AP da página ?? que reconhece a linguagem $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. Suponha que M_1 seja $(Q, \Sigma, \Gamma, \delta, q_1, F)$ onde

$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{\mathtt{0}, \mathtt{1}\}$,

$\Gamma = \{\mathtt{0}, \$\}$,

$F = \{q_1, q_4\}$, e

δ seja dada pela seguinte tabela, na qual uma entrada em branco significa $\emptyset$.

Entrada:	0			1			ε		
Pilha:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2		$\{(q_2, \mathtt{0})\}$			$\{(q_3, \varepsilon)\}$				
q_3					$\{(q_3, \varepsilon)\}$			$\{(q_4, \varepsilon)\}$	
q_4									

Podemos também usar um diagrama de estados para descrever um AP, como mostrado nas Figuras 2.6, 2.7 e 2.8. Tais diagramas são semelhantes aos diagramas de estado usados para descrever autômatos finitos, modificados para mostrar como o AP usa sua pilha quando vai de um estado para outro. Escrevemos "$a, b \to c$" para significar que quando a máquina está lendo um a da entrada ela pode substituir o símbolo b no topo da pilha por um c. Quaisquer dos a, b, e c pode ser ε. Se a é ε, a máquina pode fazer sua transição sem ler qualquer símbolo da entrada. Se b é ε, a máquina não escreve qualquer símbolo na pilha quando faz essa transição.

Figura 2.6: Diagrama de estados para o AP M_1 que reconhece $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$

A definição formal de um AP não contém qualquer mecanismo explícito para permitir que o AP teste por uma pilha vazia. Esse AP é capaz de obter o mesmo efeito colocando inicialmente um símbolo especial \$ na pilha. Então se ele por acaso vê \$ novamente, ele sabe que a pilha efetivamente está vazia. Subsequentemente, quando nos referimos a testar por uma pilha vazia em uma descrição informal de um AP, implementamos o procedimento da mesma maneira.

Igualmente, AP's não podem testar explicitamente por ter atingido o final da cadeia de entrada. Esse AP é capaz de alcançar esse efeito porque o estado de aceitação toma efeito somente quando a máquina está no final da cadeia. Por conseguinte de agora por diante, assumimos que AP's podem testar pelo final da entrada, e sabemos que podemos implementá-lo da mesma maneira.

Exemplo 2.10 .....
Este exemplo ilustra um autômato a pilha que reconhece a linguagem

$$\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ e } i = j \text{ ou } i = k\}.$$

Informalmente o AP para essa linguagem funciona primeiro lendo e empilhando os a's. Quando os a's terminam a máquina tem todos eles na pilha de modo que ela pode emparelhá-los ou com os b's ou com os c's. Essa manobra é um pouco complicada porque a máquina não sabe com antecipação se emparelha os a's com os b's ou com os c's. Não-determinismo vem para ajudar aqui.

Usando seu não-determinismo, o AP pode adivinhar se emparelha os a's com os b's ou com os c's, como mostrado na Figura 2.7. Pense na máquina como tendo dois ramos de seu não-determinismo, um para cada palpite possível. Se um deles emparelha, aquele ramo aceita e a máquina inteira aceita. Na verdade poderíamos mostrar, embora não o fazemos, que não-determinismo é *essencial* para reconhecer essa linguagem com um AP.

Figura 2.7: Diagrama de estados para o AP M_2 que reconhece $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \geq 0 \text{ e } i = j \text{ ou } i = k\}$

Exemplo 2.11 . . . . . .
Neste exemplo damos um AP M_3 que reconhece a linguagem $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$. Lembre-se que $w^{\mathcal{R}}$ significa w escrita de trás para frente. A descrição informal do AP segue.

> Comece empilhando os símbolos que são lidos na pilha. A cada ponto não-deterministicamente estipule que o meio da cadeia foi atingido e então mude para desempilhar a pilha para cada símbolo lido, verificando para ver se eles são os mesmos. Se eles fossem sempre o mesmo símbolo e a pilha se esvazia ao mesmo tempo em que a entrada termina, aceite; caso contrário, rejeite.

A seguir está o diagrama dessa máquina.

Figura 2.8: Diagrama de estados para o AP M_3 que reconhece $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

Equivalência com gramáticas livres-do-contexto

Nesta seção mostramos que gramáticas livres-do-contexto e autômatos a pilha são equivalentes em poder. Ambos são capazes de descrever a classe das linguagens livres-do-contexto. Mostramos como converter qualquer gramática livre-do-contexto em um autômato a pilha que reconhece a mesma linguagem e vice-versa. Relembrando que definimos uma linguagem livre-do-contexto como sendo qualquer linguagem que pode ser descrita por uma gramática livre-do-contexto, nosso objetivo é o seguinte teorema.

Teorema 2.12 . . . . . .
Uma linguagem é livre-do-contexto se e somente se algum autômato a pilha a reconhece.

Como de costume para teoremas "se e somente se", temos duas direções para provar. Neste teorema, ambas as direções são interessantes. Primeiro, vamos fazer a parte mais fácil que é a ida.

Lema 2.13 . . . . . .
Se uma linguagem é livre-do-contexto, então algum autômato a pilha a reconhece.

. . . . . .
Idéia da prova. Seja A uma LLC. Da definição sabemos que A tem uma GLC, G, que a gera. Mostramos como converter G em um AP equivalente, que chamamos P.

O AP P que agora descrevemos funcionará aceitando sua entrada w, se G gera aquela entrada, determinando se existe uma derivação para w. Lembre-se que uma derivação é simplesmente a seqüência de substituições feitas à medida que a gramática gera uma cadeia. Cada passo da derivação dá origem a uma ***cadeia intermediária*** de variáveis e terminais. Projetamos P para determinarse alguma série de substituições usando as regras G podem levar da variável inicial para w.

Uma das dificuldades em se testar se existe uma derivação para w está em descobrir quais substituições fazer. O não-determinismo do AP o permite estipular a seqüência

de substituições corretas. A cada passo da derivação uma das regras para uma dada variável é selecionada não-deterministicamente e usada para substituir aquela variável.

O AP P começa escrevendo a variável inicial na sua pilha. Ela passa por uma série de cadeias intermediárias, fazendo uma substituição após outra. Em algum ponto ele pode chegar a uma cadeia que contém somente símbolos terminais, o que quer dizer que ele derivou uma cadeia usando a gramática. Então P aceita se essa cadeia for idêntica à cadeia que ela recebeu como entrada.

Implementar essa estratégia em um AP requer uma idéia adicional. Precisamos ver como o AP armazena as cadeias intermediárias à medida que ela vai de uma para outra. Simplesmente usar a pilha para armazenar cada cadeia intermediária é tentador. Entretanto, isso não chega a funcionar porque o AP precisa encontrar as variáveis na cadeia intermediária e fazer substituições. O AP pode acessar somente o símbolo no topo da pilha e esse pode ser um símbolo terminal ao invés de uma variável. A forma de contornar esse problema é manter somente *parte* da cadeia intermediária na pilha: os símbolos começando com a primeira variável na cadeia intermediária. Quaisquer símbolos terminais aparecendo antes da primeira variável são emparelhados imediatamente com símbolos na cadeia de entrada. A figura abaixo mostra o AP P.

Figura 2.9: P representando a cadeia intermediária $\texttt{01}A\texttt{1}A\texttt{0}$

A seguir vai uma descrição informal de P.

1. Coloque o símbolo marcador \$ e a variável inicial na pilha.
2. Repita os seguintes passos para sempre.
   a. Se o topo da pilha é um símbolo de variável A, não-deterministicamente selecione uma das regras para A e substitua A pela cadeia no lado direito da regra.
   b. Se o topo da pilha é um símbolo terminal a, leia o próximo símbolo da entrada e compare-o com a. Se eles casam, repita. Se eles não casam, rejeite nesse ramo do não-determinismo.
   c. Se o topo da pilha é o símbolo \$, entre no estado de aceitação. Fazendo isso aceite a cadeia se ela foi toda lida.

Prova. Agora damos os detalhes formais da construção do autômato a pilha $P = (Q, \Sigma, \Gamma, \delta, q_1, F)$. Para tornar a construção mais clara usamos notação abreviada para a função de transição. Essa notação provê uma maneira de escrever uma cadeia inteira na pilha em um passo da máquina. Podemos simular essa ação introduzindo estados adicionais para escrever a cadeia um símbolo por vez, como implementado na seguinte construção formal.

Sejam q e r estados do AP, e suponha que a esteja em Σ_ε e s esteja em Γ_ε. Digamos que desejamos que o AP vá de de q para r quando ele lê a e desempilha s. Além do mais desejamos que ele empilhe a cadeia inteira $u = u_1 \cdots u_l$ na pilha ao mesmo tempo. Podemos implementar essa ação introduzindo novos estados $q_1, \ldots, q_{l-1}$ e montando a função de transição da seguinte forma

$$\begin{aligned}
&\delta(q, a, s) \text{ deve conter } (q_1, u_l),\\
&\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\},\\
&\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\},\\
&\qquad\vdots\\
&\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

Usamos a notação $(r, u) \in \delta(q, a, s)$ para representar que quando q é o estado do autômato, a é o próximo símbolo, e s é o símbolo no topo da pilha, o AP pode ler o a e desempilhar o s, então empilhamos a cadeia u na pilha e continuar para o estado r. A Figura 2.10 mostra essa implementação pictorialmente.

Figura 2.10: Implementando a abreviação $(r, xyz) \in \delta(q, a, s)$

Os estados de P são $Q = \{q_{\text{inicio}}, q_{\text{laco}}, q_{\text{aceita}}\} \cup E$ onde E, é o conjunto de estados de que precisamos para implementar a abreviação que acaba de ser descrita. O estado inicial é q_{inicio}. O único estado de aceitação é q_{aceita}.

A função de transição é definida da seguinte forma. Começamos inicializando a pilha para conter os símbolos \$ e S, implementando o passo 1 na descrição informal: $\delta(q_{\text{inicio}}, \varepsilon, \varepsilon) = \{(q_{\text{laco}}, S\$)\}$. Então colocamos transições para o laço principal do passo 2.

Primeiro, lidamos com o caso (a) no qual o topo da pilha contém uma variável. Faça $\delta(q_{\text{laco}}, \varepsilon, A) = \{(q_{\text{laco}}, w) \mid \text{ onde } A \to w \text{ é uma regra em } R\}$.

Segundo, lidamos com o caso (b) no qual o topo da pilha contém um terminal. Faça $\delta(q_{\text{laco}}, a, a) = \{(q_{\text{laco}}, \varepsilon)\}$.

Finalmente, lidamos com o caso (c) no qual o marcador de pilha vazia \$ está no topo da pilha. Faça $\delta(q_{\text{laco}}, \varepsilon, \$) = \{(q_{\text{aceita}}, \varepsilon)\}$.

O diagrama de estados é mostrado na Figura 2.11.

Figura 2.11: Diagrama de estados de P

. . . . . .

Exemplo 2.14 . . . . . .

Usamos o procedimento desenvolvido no Lema 2.13 para construir um AP P_1 a partir da seguinte GLC G.

$$\begin{aligned}
S &\to \mathsf{a}T\mathsf{b} \mid \mathsf{b}\\
T &\to T\mathsf{a} \mid \varepsilon
\end{aligned}$$

A função de transição é mostrada na Figura 2.12.

Figura 2.12: Diagrama de estados de P_1

Agora provamos a direção reversa do Teorema 2.12. Para a direção da ida demos um procedimento para converter um GLC em um AP. A idéia principal era projetar o autômato de modo que ele simula a gramática. Agora desejamos dar um procedimento para ir na outra direção: converter um AP em uma GLC. Projetamos a gramática para simular o autômato. Essa tarefa é um pouco complicada porque "programar" um autômato é mais fácil que "programar" uma gramática.

Lema 2.15

Se um autômato a pilha reconhece uma dada linguagem, então ela é livre-do-contexto.

Idéia da prova. Temos um AP P, e desejamos construir uma GLC G que gera todas as cadeias que P aceita. Em outras palavras, G deve gerar uma cadeia se aquela cadeia faz o AP ir do seu estado inicial para um estado de aceitação.

Para atingir esse resultado projetamos uma gramática que faz algo mais. Para cada par de estados p e q em P a gramática terá uma variável A_{pq}. Essa variável gera todas as cadeias que pode levar P de p com uma pilha vazia a q com uma pilha vazia. Observe que tais cadeias podem também levar P de p para q, independente do conteúdo da pilha em p, deixando a pilha em q na mesma condição em que ela estava em p.

Primeiro, simplificamos nossa tarefa modificando P levemente para lhe dar as seguintes características.

1. Ele tem um único estado de aceitação, q_{aceita}.
2. Ele esvazia sua pilha antes de aceitar.
3. Cada transição ou empilha um símbolo na pilha (um movimento de *empilhar*) ou desempilha um da pilha (um movimento de *desempilhar*), mas não faz ambos ao mesmo tempo.

Dar a P as características 1 e 2 é fácil. Para lhe dar a característica 3, substituimos cada transição que simultaneamente desempilha e empilha com uma seqüência de duas transições que passa por um novo estado, e substituimos cada transição que nem empilha nem desempilha por uma seqüência de duas transições que empilha e depois desempilha um símbolo de pilha arbitrário.

Para projetar G de modo que A_{pq} gere todas as cadeias que levam P de p para q, começando e terminando com uma pilha vazia, temos que entender como P opera sobre essas cadeias. Para cada tal cadeia x, o primeiro movimento de P sobre x tem que ser um empilha, pois todo movimento é um empilha ou um desempilha e P não pode desempilhar uma pilha vazia. Igualmente o último movimento sobre x tem que ser um desempilha, pois a pilha termina vazia.

Duas possibilidades ocorrem durante a computação de P sobre x. Ou o símbolo desempilhado no final é o símbolo que foi empilhado no início, ou não. Se for, a pilha está vazia somente no início e no fim da computação de P sobre x. Se não for, o símbolo inicialmente empilhado tem que ser empilhado em algum ponto antes do final de x e portanto a pilha se torna vazia nesse ponto. Simulamos a primeira possibilidade com a regra $A_{pq} \to aA_{rs}b$ onde a é o símbolo de entrada lido no primeiro movimento, b é o símbolo lido no último movimento, r é o estado seguinte a p, e s o estado que precede q. Simulamos a segunda possibilidade com a regra $A_{pq} \to A_{pr}A_{rq}$, onde r é o estado no qual a pilha se torna vazia.

Prova. Digamos que $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{\text{aceita}}\})$ e vamos construir G. As variáveis de G são $\{A_{pq} \mid p, q \in Q\}$. A variável inicial é $A_{q_0, q_{\text{aceita}}}$. Agora descrevemos as regras de G.

- Para cada $p, q, r, s \in Q$, $t \in \Gamma$, e $a, b \in \Sigma_\varepsilon$, se $\delta(p, a, \varepsilon)$ contém (r, t) e $\delta(s, b, t)$ contém (q, ε) ponha a regra $A_{pq} \to aA_{rs}b$ em G.
- Para cada $p, q, r \in Q$ ponha a regra $A_{pq} \to A_{pr}A_{rq}$ em G.
- Finalmente, para cada $p \in Q$ ponha a regra $A_{pp} \to \varepsilon$ em G.

Você pode ganhar alguma intuição para essa construção a partir das Figuras 2.13 e 2.14.

Figura 2.13: A computação de um AP correspondendo à regra $A_{pq} \to A_{pr}A_{rq}$

Figura 2.14: A computação de um AP correspondendo à regra $A_{pq} \to aA_{rs}b$

Agora provamos que essa construção funciona demonstrando que A_{pq} gera x se e somente se (sse) x pode trazer P de p com a pilha vazia para q com a pilha vazia. Consideramos cada direção do sse como uma afirmação separada.

Afirmação 2.16 .....
Se A_{pq} gera x, então x pode trazer P de p com a pilha vazia para q com a pilha vazia.

Provamos essa afirmação por indução sobre o número de passos na derivação de x a partir de A_{pq}.

Base: A derivação tem 1 passo.
A derivação com um único passo tem que usar uma regra cujo lado direito não contém variáveis. As únicas regras em G nas quais nenhuma variável ocorre no lado direito são $A_{pp} \to \varepsilon$. Claramente, a entrada ε leva P de p com a pilha vazia para q com a pilha vazia, portanto a base está provada.

Passo da indução: Assuma que a afirmação é verdadeira para derivações de comprimento no máximo k, onde $k \geq 1$, e prove que é verdadeira para derivações de comprimento $k + 1$.
Suponha que $A \stackrel{*}{\Rightarrow} aA_{rs}b$ com $k + 1$ passos. O primeiro passo nessa derivação é ou $A_{pq} \Rightarrow aA_{rs}b$ ou $A_{pq} \Rightarrow A_{pr}A_{rs}$. Vamos tratar esses dois casos separadamente.

No primeiro caso, considere a porção y de x que A_{rs} gera, tal que $x = ayb$. Devido ao fato de que $A \stackrel{*}{\Rightarrow} y$ com k passos, a hipótese da indução nos diz que P pode ir de r com a pilha vazia para s com a pilha vazia. Dado que $A_{pq} \to aA_{rs}b$ é uma regra de G, $\delta(p, a, \varepsilon)$ contém (r, t) e $\delta(s, b, t)$ contém (q, ε). Daí, se P começa em p com a pilha vazia, após ler a ele pode ir para o estado r e empilhar t na pilha. Então ler a cadeia y pode levá-lo a s e deixar t na pilha. Então após ler b ele pode ir para o estado

q e desempilhar t da pilha. Por conseguinte x pode levá-lo de p com pilha vazia para q com pilha vazia.

No segundo caso, considere as porções y e z de x que A_{pr} e A_{rs} respectivamente geram, portanto $x = yz$. Devido ao fato de que $A_{pr} \stackrel{*}{\Rightarrow} y$ em no máximo k passos e $A_{rs} \stackrel{*}{\Rightarrow} z$ em no máximo k passos, a hipótese de indução nos diz que y pode trazer P de p para r, e z pode trazer P de r para q, com pilhas vazias no início e no fim. Daí x pode trazê-lo de p com pilha vazia para q com pilha vazia. Isso completa o passo da indução.

Afirmação 2.17 .....
Se x pode trazer P de p com pilha vazia para q com pilha vazia, A_{pq} gera x.

Provamos essa afirmação por indução sobre o número de passos na computação de P que leva de p para q com pilhas vazias sobre a entrada x.

Base: A computação tem 0 passos.
Se uma computação tem 0 passos, ela começa e termina no mesmo estado, digamos p. Portanto temos que mostrar que $A_{pp} \stackrel{*}{\Rightarrow} x$. Em 0 passos, P apenas tem tempo de ler a cadeia vazia, portanto $x = \varepsilon$. Por construção, G tem a regra $A_{pp} \rightarrow \varepsilon$, portanto a base está provada.

Passo da indução: Assuma que a afirmação é verdadeira para computações de comprimento no máximo k, onde $k \geq 0$, e prove que é verdadeira para computações de comprimento $k + 1$.
Suponha que P tenha uma computação na qual x traz p para q com pilhas vazias em $k + 1$ passos. Ou a pilha está vazia somente no início e no fim dessa computação, ou ela se torna vazia em outro lugar também.

No primeiro caso, o símbolo que é empilhado no primeiro movimento tem que ser o mesmo que o símbolo que é desempilhado no último movimento. Chame esse símbolo t. Seja a a entrada lida no primeiro movimento, b a entrada lida no último movimento, r o estado após o primeiro movimento, e s o estado antes do último movimento. Então $\delta(p, a, \varepsilon)$ contém (r, t) e $\delta(s, b, t)$ contém (q, ε), e portanto a regra $A_{pq} \rightarrow aA_{rs}b$ está em G.

Seja y a porção de x sem a e b, de modo que $x = ayb$. A entrada y pode trazer P de r para s sem tocar o símbolo t que está na pilha e portanto P pode ir de r com a pilha vazia para s com a pilha vazia sobre a entrada y. Removemos o primeiro e o último passos dos $k + 1$ passos na computação original sobre x portanto a computação sobre y tem $(k + 1) - 2 = k - 1$ passos. Por conseguinte a hipótese de indução nos diz que $A_{rs} \stackrel{*}{\Rightarrow} y$. Donde $A_{pq} \stackrel{*}{\Rightarrow} x$.

No segundo caso, seja r um estado no qual a pilha se torna vazia, e diferente do início ou do final da computação sobre x. Então as porções da computação de p para r e de r para q cada uma contém no máximo k passos. Digamos que y é a entrada lida durante a primeira porção e z é a entrada lida durante a segunda porção. A hipótese de indução nos diz que $A_{pr} \stackrel{*}{\Rightarrow} y$ e $A_{rq} \stackrel{*}{\Rightarrow} z$. Devido ao fato de que a regra $A_{pq} \rightarrow A_{pr}A_{rq}$ está em G, $A_{pq} \stackrel{*}{\Rightarrow} x$, e a prova está completa.

Isto completa a prova do Lema 2.15 e do Teorema 2.12.

.....

Acabamos de provar que autômatos a pilha reconhecem a classe das linguagens livres-do-contexto. Essa prova nos permite estabelecer um relacionamento entre as linguagens regulares e as linguagens livres-do-contexto. Devido ao fato de que toda linguagem regular é reconhecida por um autômato finito e todo autômato finito é automaticamente um autômato a pilha que simplesmente ignora sua pilha, agora sabemos que toda linguagem regular é também uma linguagem livre-do-contexto.

Corolário 2.18 .....
Toda linguagem regular é livre-do-contexto.

Figura 2.15: Relacionamento entre linguagens regulares e linguagens livres-do-contexto

2.3 Linguagens não-livres-do-contexto .....

Nesta seção apresentamos uma técnica para provar que certas linguagens não são livres-do-contexto. Lembre-se que na Seção 1.4 introduzimos o lema do bombeamento para mostrar que certas linguagens não são regulares. Aqui apresentamos um lema do bombeamento semelhante para linguagens livres-do-contexto. Ele enuncia que toda linguagen livre-do-contexto tem um valor especial chamado o ***comprimento de bombeamento*** tal que todas as cadeias de comprimento maior que esse valor na linguagem podem ser "bombeadas." Dessa vez o significado de *bombeada* é um pouco mais complicado. Significa que a cadeia pode ser dividida em cinco partes de modo que a segunda e a quarta partes podem ser repetidas um número qualquer de vezes e a cadeia resultante ainda permanece na linguagem.

O lema do bombeamento para linguagens livres-do-contexto

Teorema 2.19 .....
Lema do bombeamento para linguagens livres-do-contexto Se A é uma linguagem livre-do-contexto, então existe um número p (o comprimento de bombeamento) onde, se s é uma cadeia qualquer de A de comprimento pelo menos p, então s pode ser dividida em cinco partes $s = uvxyz$ satisfazendo as condições:

1. Para cada $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, e
3. $|vxy| \leq p$.

Quando s está sendo dividida em $uvxyz$, a condição 2 diz que v ou y não é a cadeia vazia. Do contrário o teorema seria trivialmente verdadeiro. A condição 3 enuncia que as partes v, x, e y juntas têm comprimento no máximo p. Essa condição técnica às vezes é útil ao provar que certas linguagens são não-livres-do-contexto.

.....

Idéia da prova. Seja A uma LLC e suponha que G seja uma GLC que a gera. Temos que mostrar que qualquer cadeia suficientemente longa s em A pode ser bombeada e permanecer em A. A idéia por trás dessa abordagem é simples.

Seja s uma cadeia muito longa em A. (Esclarecemos mais adiante o que queremos dizer por "muito longa.") Devido ao fato de que s está em A, ela é derivável de G e portanto tem uma árvore sintática. A árvore sintática para s tem que ser muito alta porque s é muito longa. Ou seja, a árvore sintática tem que conter algum caminho longo da variável inicial na raiz da árvore para um dos símbolos terminais em uma folha. Sobre esse caminho longo algum símbolo de variável R deve se repetir devido ao princípio da casa-de-pombos. Como a Figura 2.16 mostra essa repetição nos permite substituir a subárvore embaixo da segunda ocorrência de R pela subárvore embaixo da primeira ocorrência de R e ainda obter uma árvore sintática legal. Por conseguinte podemos cortar s em cinco pedaços $uvxyz$ como a figura indica, e podemos repetir o segundo e o quarto pedaços e obter uma cadeia ainda na linguagem. Em outras palavras, uv^ixy^iz está em A para qualquer $i \geq 0$.

Figura 2.16: Cirurgia em árvores sintáticas

Vamos agora voltar para os detalhes para se obter todas as três condições do lema do bombeamento. Mostramos também como calcular o comprimento de bombeamento p.

Prova. Seja G uma GLC a LLC A. Seja b o número máximo de símbolos no lado de direito de uma regra. Podemos assumir que $b \geq 2$. Em qualquer árvore sintática usando essa gramática sabemos que um nó não pode ter mais que b filhos. Em outras palavras no máximo b folhas estão 1 passo da variável inicial; no máximo b^2 folhas estão a no máximo 2 passos da variável inicial; e no máximo b^h folhas estão a no máximo h passos da variável inicial. Portanto, se a altura da árvore sintática é no máximo h, o comprimento da cadeia gerada é no máximo b^h.

Seja $|V|$ o número de variáveis em G. Fazemos p ser $b^{|V|+2}$. Devido ao fato de que $b \geq 2$, sabemos que $p > b^{|V|+1}$, portanto uma árvore sintática para qualquer cadeia em A de comprimento pelo menos p requer altura no mínimo $|V| + 2$.

Suponha que s seja uma cadeia em A de comprimento no mínimo p. Agora mostramos como bombear s. Seja τ a árvore sintática para s. Se s tem várias árvores sintáticas, escolhemos τ como sendo a árvore sintática que tem o menor número de nós. Como $|s| \geq p$, sabemos que τ tem altura no mínimo $|V| + 2$, portanto o caminho mais longo em τ tem comprimento pelo menos $|V| + 2$. Esse caminho tem que ter pelo menos $|V| + 1$ variáveis porque somente a folha é um terminal. Com G tendo somente $|V|$ variáveis, alguma variável R aparece mais que uma vez no caminho. Por conveniência mais adiante, selecionamos R como sendo a variável que se repete entre as $|V| + 1$ variáveis mais baixas nesse caminho.

Dividimos s em $uvxyz$ conforme a Figura 2.16. Cada ocorrência de R tem uma subárvore sob ela, gerando uma parte da cadeia s. A ocorrência mais superior de R tem uma subárvore maior e gera vxy, enquanto que a ocorrência mais inferior gera somente x com uma subárvore menor. Ambas essas subárvores são geradas pela mesma variável, portanto podemos substituir uma pela outra e ainda assim obter uma árvore

sintática válida. Substituindo a menor pela maior repetidamente dá árvores sintáticas para as cadeias uv^ixy^iz a cada $i > 1$. Substituindo a maior pela menor gera a cadeia uxz. Isso estabelece a condição 1 do lema. Agora voltamos para as condições 2 e 3.

Para obter a condição 2, temos que assegurar que ambas v e y não são ε. Se fossem, a árvore sintática obtida substituindo-se a menor subárvore pela maior teria menos nós que τ tem e ainda geraria s. Esse resultado não é possível porque já tínhamos escolhido τ como sendo uma árvore sintática para s com o menor número de nós. Essa é a razão para se selecionar τ dessa maneira.

De modo a obter a condição 3 precisamos nos assegurar de que vxy tem comprimento no máximo p. Na árvore sintática para s a ocorrência mais superior de R gera vxy. Escolhemos R de modo que ambas as ocorrências estejam entre as $|V| + 1$ variáveis de baixo no caminho, e escolhemos o caminho mais longo na árvore sintática, portanto a subárvore onde R gera vxy tem altura no máximo $|V|+2$. Uma árvore dessa altura pode gerar uma cadeia de comprimento no máximo $b^{|V|+2} = p$.

.....

Para algumas dicas sobre o uso do lema do bombeamento para provar que linguagens são não-livres-do-contexto, reveja a página ?? onde discutimos o problema relacionado de se provar não-regularidade com o lema do bombeamento para linguagens regulares.

Exemplo 2.20 .....

Use o lema do bombeamento para mostrar que a linguagem $B = \{\mathsf{a}^n\mathsf{b}^n\mathsf{c}^n \mid n \geq 0\}$ é não-livre-do-contexto.

Assumimos que B é uma LLC e obtemos uma contradição. Seja p o comprimento de bombeamento para B que é garantido existir pelo lema do bombeamento. Selecione a cadeia $s = \mathsf{a}^p\mathsf{b}^p\mathsf{c}^p$. Claramente s é um membro de B e de comprimento pelo menos p. O lema do bombeamento enuncia que s pode ser bombeada, mas mostramos que ela não pode. Em outras palavras, mostramos que independetemente de como dividimos s em $uvxyz$, uma das três condições do lema é violada.

Primeiro, a condição 2 estipula que ou v ou y seja não-vazia. Então consideramos um dos dois casos, dependendo se subcadeias v e y contêm mais que um tipo de símbolo do alfabeto.

1. Quando tanto v quanto y contêm somente um tipo de símbolos do alfabeto, v não contém ambos a's e b's ou ambos b's e c's, e o mesmo se verifica para y. Nesse caso a cadeia uv^2xy^2z não pode conter igual número de a's, b's, e c's. Por conseguinte ela não pode ser um membro de B. Isso viola a condição 1 do lema e é portanto uma contradição.

2. Quando v ou y contém mais que um tipo de símbolo uv^2xy^2z pode conter iguais números dos três símbolos do alfabeto mas não contê-los-á na ordem correta. Daí ela não pode ser um membro de B e uma contradição ocorre.

Um desses casos tem que ocorrer. Devido ao fato de que ambos os casos resultam numa contradição, uma contradição é inevitável. Portanto a suposição de que B é uma LLC tem que ser falsa. Por conseguinte provamos que B não é uma LLC.

Exemplo 2.21 .....

Seja $C = \{\mathsf{a}^i\mathsf{b}^j\mathsf{c}^k \mid 0 \leq i \leq j \leq k\}$. Usamos o lema do bombeamento para mostrar que C não é uma LLC. Essa linguagem é semelhante à linguagem B no Exemplo 2.20, mas provar que ela não é livre-do-contexto é um pouco mais complicado.

Assuma que C é uma LLC e obtenha uma contradição. Seja p o comprimento de bombeamento dado pelo lema do bombeamento. Usamos a cadeia $\mathtt{a}^p\mathtt{b}^p\mathtt{c}^p$ que usamos anteriormente, mas dessa vez temos que "bombear para baixo" assim como "bombear para cima." Seja $s = uvxyz$ e novamente considere os dois casos que ocorreram no Exemplo 2.20.

1. Quando ambas v e y contêm somente um tipo de símbolo do alfabeto, v não contém ambos a's e b's ou ambos b's e c's, e o mesmo se verifica para y. Note que o raciocínio usado previamente no caso 1 não mais se aplica. A razão é que C contém cadeias com números diferentes de a's, b's, e c's desde que os números não sejam decrescentes. Temos que analisar a situação mais cuidadosamente para mostrar que s não pode ser bombeada. Observe que devido ao fato de que v e y contêm somente um tipo de símbolo do alfabeto, um dos símbolos a, b, ou c não aparece em v ou y. Dividimos mais ainda esse caso em três subcasos de acordo com qual símbolo não aparece.

   a. Os a's não aparece. Então tentamos bombear para baixo para obter a cadeia $uv^0xy^0z = uxz$. Essa contém o mesmo número de a's que s contém, mas contém menos b's ou menos c's. Por conseguinte ela não é um membro de C, e uma contradição ocorre.

   b. Os b's não aparecem. Então ou a's ou c's têm que aparecer em v ou y porque não pode ser que ambos sejam a cadeia vazia. Se a's aparecem, a cadeia uv^2xy^2z contém mais a's que b's, portanto ela não está em C. Se c's aparecem, a cadeia uv^0xy^0z contém mais b's que c's, portanto ela não está em C. Em qualquer dos casos, uma contradição ocorre.

   c. Os c's não aparecem. Então a cadeia uv^2xy^2z contém mais a's ou mais b's que c's, portanto ela não está em C, e uma contradição ocorre.

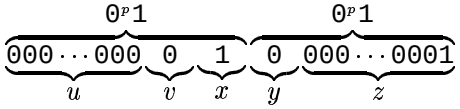
2. Quando v ou y contém mais que um tipo de símbolo, uv^2xy^2z não conterá os símbolos na ordem correta. Daí ela não pode ser um membro de C, e uma contradição ocorre.

Por conseguinte mostramos que s não pode ser bombeada o que viola o lema do bombeamento e que C não é livre-do-contexto.

Exemplo 2.22 .....

Seja $D = \{ww \mid w \in \{0,1\}^*\}$. Use o lema do bombeamento para mostrar que D não é uma LLC. Assuma que D é uma LLC e obtenha uma contradição. Seja p o comprimento de bombeamento dado pelo lema do bombeamento.

Dessa vez escolher a cadeia s é menos óbvio. Uma possibilidade é a cadeia 0^p10^p1. Ela é um membro de D e tem comprimento maior que p, portanto ela parece ser uma boa candidata. Mas essa cadeia *pode* ser bombeada dividindo-a da seguinte forma, de modo que ela não é adequada para nossos propósitos.



Vamos tentar uma outra candidata para s. Intuitivamente, a cadeia $0^p1^p0^p1^p$ parece capturar mais da "essência" da linguagem D que a candidata anterior. Na verdade, podemos mostrar que essa cadeia não funciona, como segue.

Mostramos que a cadeia $s = \texttt{0}^p\texttt{1}^p\texttt{0}^p\texttt{1}^p$ não pode ser bombeada. Dessa vez usamos a condição 3 do lema do bombeamento para restringir a maneira pela qual s pode ser dividida. Ela diz que podemos bombear s dividindo $s = uvxyz$, onde $|vxy| \leq p$.

Primeiro, mostramos que a subcadeia vxy tem que passar do meio da cadeia s. Caso contrário, se a cadeia ocorre somente na primeira metade de s, bombeando s para cima até uv^2xy^2z move um 1 para a primeira posição da segunda metade, e portanto ela não pode ser da forma ww. Igualmente, se vxy ocorre na segunda metade de s, bombeando s para cima até uv^2xy^2z move um 0 para a última posição da primeira metade, e portanto ela não pode ser da forma ww.

Mas se a subcadeia vxy passa do meio da cadeia s, quando tentamos bombear s para baixo até uxz ela tem a forma $\texttt{0}^p\texttt{1}^i\texttt{0}^j\texttt{1}^p$, onde i e j não podem ser ambos p. Essa cadeia não é da forma ww. Por conseguinte s não pode ser bombeada, e D não é uma LLC.

. . . .

Exercícios

2.1 Retomemos a GLC G_4 que demos no Exemplo 2.3. Por conveniência, vamos renomear suas variáveis com letras da seguinte forma:

$$\begin{aligned} E &\to E + T \mid T \\ T &\to T \times F \mid F \\ F &\to (E) \mid \texttt{a} \end{aligned}$$

Dê árvores sintáticas e derivações para cada cadeia:

a. a

b. a+a

c. a+a+a

d. ((a))

2.2 a. Use as linguagens $A = \{\texttt{a}^m\texttt{b}^n\texttt{c}^n \mid m, n \geq 0\}$ e $B = \{\texttt{a}^n\texttt{b}^n\texttt{c}^m \mid m, n \geq 0\}$ juntamente com o Exemplo 2.20 para mostrar que a classe das linguagens livres-do-contexto não é fechada sob interseção.

b. Use a parte (a) e a lei de De Morgan (Teorema 0.10) para mostrar que a classe das linguagens livres-do-contexto não é fechada sob complementação.

2.3 Responda cada item para a seguinte gramática livre-do-contexto G:

$$\begin{aligned} R &\to XRX \mid S \\ S &\to \texttt{a}T\texttt{b} \mid \texttt{b}T\texttt{a} \\ T &\to XTX \mid X \mid \varepsilon \\ X &\to \texttt{a} \mid \texttt{b} \end{aligned}$$

a. Quais são as variáveis e terminais de G? Qual é o símbolo inicial?

b. Dê três exemplos de cadeias em $L(G)$.

c. Dê três exemplos de cadeias que *não estão* em $L(G)$.

d. Verdadeiro ou Falso: $T \Rightarrow \texttt{aba}$.